Audio/Video Connectivity Solutions for the Broadcast Industry

Reference Designs for Advanced Xilinx Products

XAPP514 (v3.0) August 31, 2006
Xilinx is disclosing this Document and Intellectual Property (hereinafter “the Design”) to you for use in the development of designs to operate on, or interface with Xilinx FPGAs. Except as stated herein, none of the Design may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Any unauthorized use of the Design may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

Xilinx does not assume any liability arising out of the application or use of the Design; nor does Xilinx convey any license under its patents, copyrights, or any rights of others. You are responsible for obtaining any rights you may require for your use or implementation of the Design. Xilinx reserves the right to make changes, at any time, to the Design as deemed desirable in the sole discretion of Xilinx. Xilinx assumes no obligation to correct any errors contained herein or to advise you of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or technical support or assistance provided to you in connection with the Design.

THE DESIGN IS PROVIDED “AS IS” WITH ALL FAULTS, AND THE ENTIRE RISK AS TO ITS FUNCTION AND IMPLEMENTATION IS WITH YOU. YOU ACKNOWLEDGE AND AGREE THAT YOU HAVE NOT RELIED ON ANY ORAL OR WRITTEN INFORMATION OR ADVICE, WHETHER GIVEN BY XILINX, OR ITS AGENTS OR EMPLOYEES. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DESIGN, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT OF THIRD-PARTY RIGHTS.

IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOST DATA AND LOST PROFITS, ARISING FROM OR RELATING TO YOUR USE OF THE DESIGN, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THE TOTAL CUMULATIVE LIABILITY OF XILINX IN CONNECTION WITH YOUR USE OF THE DESIGN, WHETHER IN CONTRACT OR TORT OR OTHERWISE, WILL IN NO EVENT EXCEED THE AMOUNT OF FEES PAID BY YOU TO XILINX HEREREUNDER FOR USE OF THE DESIGN. YOU ACKNOWLEDGE THAT THE FEES, IF ANY, REFLECT THE ALLOCATION OF RISK SET FORTH IN THIS AGREEMENT AND THAT XILINX WOULD NOT MAKE AVAILABLE THE DESIGN TO YOU WITHOUT THESE LIMITATIONS OF LIABILITY.

The Design is not designed or intended for use in the development of on-line control equipment in hazardous environments requiring fail-safe controls, such as in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support, or weapons systems (“High-Risk Applications”). Xilinx specifically disclaims any express or implied warranties of fitness for such High-Risk Applications. You represent that use of the Design in such High-Risk Applications is fully at your risk.

© 2006 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. PowerPC is a trademark of IBM, Inc. All other trademarks are the property of their respective owners.
## Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>04/15/05</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
<tr>
<td>03/06/06</td>
<td>2.0</td>
<td>Re-released as XAPP514.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Expanded Introduction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added section “Digital Audio” and Chapter 18 (AES3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Revised and updated Chapter 10, Chapter 13, and Chapter 15.</td>
</tr>
<tr>
<td>08/31/06</td>
<td>3.0</td>
<td>- Revised Chapter 17, “HDTV Video Pattern Generator.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Split off introductory material from the AES3 audio chapter and created a new Chapter 18, “Introduction to Digital Audio for Video Broadcasting.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added a new section to Chapter 18, “Audio Sample Rate Conversion.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- “AES3 Serial Digital Audio Interfaces for Xilinx FPGAs” becomes Chapter 19.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added Chapter 20, “Asynchronous Sample Rate Converter.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added Chapter 21, “AES3 Audio Demultiplexer for Standard-Definition Digital Audio.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Resectioned book with separator pages for easier perception of section contents.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Minor updates throughout.</td>
</tr>
</tbody>
</table>
Table of Contents

Schedule of Figures .......................................................... 19
Schedule of Tables ............................................................ 29

Preface: About This Guide
  Guide Contents ............................................................ 33
  Additional Resources ........................................................ 35
  Conventions ..................................................................... 35
    Typographical ............................................................... 35
    Online Document .......................................................... 36

Chapter 1: Introduction

Section I:
  SD-SDI

Chapter 2: SD-SDI Physical Layer Implementation
  Summary ............................................................................. 41
  Introduction ....................................................................... 41
    Measuring SDI Transmitter and Receiver Performance ............ 43
    SDI Cable Equalization ................................................... 44
    Jitter Reduction ............................................................. 46
    Clock Multiplication ....................................................... 47
    SDI Cable Driver ............................................................ 49
    Clock and Data Recovery .................................................. 50
      External CDR ............................................................... 50
      Internal CDR ............................................................... 51
      Data Recovery Only ....................................................... 54
  Reference Designs ............................................................ 56
  Conclusions ....................................................................... 58
  Design Files ....................................................................... 58
  Appendix A: Test Equipment ................................................ 58
    Test Equipment Used for Receiver Input Jitter Tolerance Measurements .................................................. 58
    Test Equipment Used for Transmitter Output Jitter Measurements .......................................................... 58

Chapter 3: SD-SDI Video Encoder
  Summary ............................................................................. 59
  SDI Introduction ............................................................... 60
    Digital Video Formats ....................................................... 60
    Encoding and Decoding .................................................... 60
    Framing and TRS Clipping ................................................ 61
Chapter 4: SD-SDI Video Decoder

Summary ................................................................. 71
SDI Introduction ....................................................... 71
  Digital Video Formats ............................................ 71
  Encoding and Decoding .......................................... 72
  Framing .............................................................. 73
  SDI Bit Rates ...................................................... 73
  Error Detection .................................................... 74
Reference Design ..................................................... 74
  Serial Descrambler Implementation: ser_descrambler.* .......... 74
  Parallel Descrambler Implementation: par_descrambler.* ........ 75
  Framer Implementation ............................................ 76
    Serial Framer: ser_framer.* ..................................... 77
    Parallel Framer: par_framer.* .................................... 78
  CLC011 Example: An Alternative Solution ......................... 79
  CY7C9335 Example: An Alternative Solution ....................... 79
  Reference Design Results ........................................ 81
  Testing .............................................................. 81
Conclusion ............................................................ 81
Design Files .......................................................... 81

Chapter 5: SD-SDI Video Flywheel

Summary ................................................................. 83
Introduction .......................................................... 83
Digital Video Standards ............................................ 84
Video Standard Detection .......................................... 85
Flywheel Video Decoder ............................................. 86
  Basic Video Decoding ............................................. 86
  Using a Flywheel for Noise Immunity ............................ 87
  Synchronous Switching Considerations ......................... 87
  Tolerating an Early Falling Transition of the V Bit .......... 88
Reference Design ..................................................... 88
  Trs_detect Module ............................................... 89
  Autodetect Module ................................................. 91
Chapter 6: SD-SDI Ancillary Data and EDH Processors

Summary ................................................................. 101
Introduction .......................................................... 101
ANC Packets .......................................................... 102
ANC Packet Format .................................................... 102
Non-Conforming ANC Packets ....................................... 104
Another Start and End Marker Protocol ............................ 105
8-bit Considerations .................................................. 106
ANC Packet Positioning .............................................. 106
ANC Packet Insertion Rules .......................................... 108
ANC Packet Deletion Rules .......................................... 109
Synchronous Switching Considerations ............................ 109
Error Detection and Handling (EDH) ................................ 110
CRC Checkword Calculations ........................................ 111
Error Flags ............................................................... 118
edh — Error Detected Here ........................................ 118
ed — Error Detected Already ....................................... 118
idh — Internal Error Detected Here ................................ 118
ida — Internal Error Detected Already ............................ 118
ues — Unknown Error Status ........................................ 118
EDH Packet Format .................................................... 119
Reference Design .......................................................... 120
ANC and EDH Processor ............................................. 120
edh_check Module ..................................................... 120
anc_demux Module ...................................................... 121
anc_mux Module .......................................................... 121
edh_gen Module .......................................................... 122
edh_processor Module ................................................ 122
Results ........................................................................ 122
Conclusion .................................................................... 123
Design Files ................................................................. 123
Appendix A .................................................................... 124
Additional Reference Design Information .......................... 124
edh_processor Module ..................................................... 124
edh_rx Module ............................................................... 125
anc_rx Module ............................................................... 127
edh_loc Module .............................................................. 128
edh_crc Module ............................................................... 129
anc_mux Module .............................................................. 130
anc_insert Module .......................................................... 131
anc_pkt_gen Module ........................................................ 134
anc_demux and anc_extract Modules ................................. 137
anc_edh_processor Module ............................................... 139
edh_gen Module .............................................................. 140
edh_tx Module ............................................................... 140
Section II: HD-SDI

Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Summary ................................. 181
Introduction ............................. 181
HD-SDI Data Format ..................... 182
   HD-SDI Supported Video Formats 182
   HD-SDI Data Format .................. 184
   Channel Interleaving .................. 186
   Encoding ............................ 187
HD-SDI Transmitter Requirements .... 187
   Electrical Requirements ............. 187
   Jitter Requirements ................ 188
   Cable Driver ......................... 188
Clocks ................................. 189
   Reference Clocks ..................... 190
   User Clocks .......................... 191
   Jitter Reduction ..................... 192
Reference Designs ..................... 193
   Jitter Performance .................. 196
   Reference Design Size ............... 197
Conclusion ............................ 198
Design Files ........................... 198
Appendix: Reference Design Details .. 198
   Video Format Detection and Line Number Generation 198
   HD-SDI Encoder ...................... 201

Chapter 10: HD-SDI Receiver Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Summary ................................. 203
Introduction ............................ 203
HD-SDI Receiver Functions .......... 204
   Cable Equalization .................. 204
   Clock and Data Recovery .......... 205
Chapter 11: HD-SDI Integration Examples for the Serial Digital Video Demonstration Board

Summary ................................................................. 231
Introduction ............................................................ 231
Separate HD-SDI Tx and Rx Application ......................... 232
  Clocks ................................................................. 233
  Transmitter Section ............................................... 233
  Receiver Section .................................................. 234
  Design Size .......................................................... 234
HD-SDI Pass-Through Application .................................. 235
  Clocks ................................................................. 236
  PLL ................................................................. 236
  Design Size .......................................................... 237
HD-SDI Pass-Through Using the ICS664 Application ............ 237
  Design Size .......................................................... 239
Conclusion .............................................................. 240
Design Files ........................................................... 240
Section III: Multi-Rate HD/SD-SDI

Chapter 12: Multi-Rate HD/SD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Summary .............................................................. 243
Introduction .......................................................... 243
SD-SDI and HD-SDI Similarities and Differences ...................... 244
Generating SD-SDI Bitstreams with the RocketIO Transmitter ........ 244
Cable Driver ......................................................... 245
Clocks ................................................................. 246
  Reference Clocks ................................................ 246
  User Clocks ...................................................... 247
  Clocking Example ............................................. 247
  More Thoughts on Clocks .................................... 249
Reference Designs .................................................. 249
  Multi-Rate Encoder ............................................ 249
  SD-SDI Bit Replication ......................................... 251
  Multi-Rate SDI Transmitter Example ........................... 252
  SD-SDI Transmitter Example Using the RocketIO Transceiver .... 253
  Jitter Performance ............................................. 253
  Reference Design Size ......................................... 254
Conclusions ................................................................ 254
Design Files .......................................................... 255

Chapter 13: Multi-Rate HD/SD-SDI Receiver Using RocketIO Multi-Gigabit Transceivers

Summary .............................................................. 257
Introduction .......................................................... 257
SD-SDI and HD-SDI Similarities and Differences ...................... 258
Multi-Rate SDI Receiver Functions .................................... 258
  Cable Equalization .............................................. 258
  Clock and Data Recovery ...................................... 259
  Deserialization ................................................... 259
  Decoding .......................................................... 259
  Framing ............................................................ 260
  Error Checking ................................................... 261
  Rate Selection .................................................... 262
Implementing the Multi-Rate SDI Receiver ............................. 262
  Cable Equalization .............................................. 262
  Receiving SD-SDI Bitstreams with the RocketIO Transceiver .... 264
    DRU Version 1 .................................................. 264
    DRU Version 2 .................................................. 266
  RocketIO Transceiver Clocks .................................... 267
    Reference Clocks .............................................. 267
    User Clocks .................................................... 269
Chapter 14: Multi-Rate SDI Integration Examples for the Serial Digital Video Demonstration Board

Summary ................................................................. 281
Introduction ......................................................... 281
Application Example 1 (EDH Processor from Chapter 6) ................................................................. 283
  HD-SDI Mode ...................................................... 283
  SD-SDI Mode ...................................................... 284
  Clocks ................................................................. 285
  Design Size ........................................................ 287
Application Example 2 (EDH Processor from Chapter 7) ................................................................. 288
  Clocks ................................................................. 288
  Design Size ........................................................ 289
Application Example 3 (Using ICS664) ................................................................. 291
  Clocks ................................................................. 291
  Design Size ........................................................ 293
Conclusion .............................................................. 293
Design Files ........................................................... 293

Section IV: DVB-ASI

Chapter 15: DVB-ASI Physical Layer Implementation

Introduction to DVB-ASI .................................................... 297
Implementing the DVB-ASI Physical Layer with SelectIO Features ................................................................. 299
  Introduction ........................................................ 299
  SelectIO DVB-ASI Receiver .......................................... 299
    Reference Design Details .......................................... 300
    Cable Equalization .............................................. 300
    Clocking .......................................................... 301
    Data Recovery Module ........................................... 302
    8B/10B Decoder .................................................. 307
    Sync Byte Insertion / Deletion .................................. 310
Section V: Video Test Pattern Generators

Chapter 16: SDTV Video Pattern Generators

Summary ................................................................. 349

A Brief Component Digital Video Primer .......................... 349
  Component Digital Video Standards ........................... 349
  Color Space .................................................. 350
  Sampling Schemes ........................................... 350
  Video Format ................................................ 351
  Numbering Quirks ......................................... 354

Video Test Pattern Standards ..................................... 355
  Standards for Color Bar Test Patterns ...................... 355
  SDI Pathological Test Patterns ............................ 357

Reference Designs ................................................. 358
  Limiting Signal Transition Rates ......................... 358

User Instructions for Instantiating Core Generator FIFO .... 310
SelectIO DVB-ASI Transmitter .................................. 313
  Line Driver .................................................. 314
  Clocking .................................................... 315
  8B/10B Encoder ............................................ 315
  User Instructions for Instantiating Core Generator 8B/10B Encoder ................................. 316
Pass-Through Mode ............................................... 318
  Modes of Operation ....................................... 318
  BIST Test Bitstream Generator and Checker ............. 319
  BIST Parameter Setting ................................... 319
  Reference Design ........................................... 321

Conclusion ......................................................... 326

Implementing the DVB-ASI Physical Layer with RocketIO Transceivers ........... 327
  Introduction .................................................. 327
  RocketIO DVB-ASI Receiver ................................ 327
    Cable Equalization ...................................... 328
    Data Recovery Unit (DRU) ............................... 329
    RocketIO Transceiver Clocks ............................ 331
    Deserialization .......................................... 332
    User Instructions for Instantiating Core Generator 8B/10B Decoder .................. 334
    Sync Byte Insertion / Deletion .......................... 336
    User Instructions for Instantiating Core Generator FIFO .......................... 338
  RocketIO DVB-ASI Transmitter .............................. 340
    Cable Driver .............................................. 341
    Clocking Requirements .................................. 341
    8B/10B Encoder ........................................... 342
    User Instructions for Instantiating Core Generator 8B/10B Encoder ................ 343
    DVB-ASI Bit Replication ................................ 344
    Jitter Performance ....................................... 345
    Resource Utilization ..................................... 345
    Conclusions .............................................. 345

Design Files ....................................................... 345
Section VI: Digital Audio

Chapter 18: Introduction to Digital Audio for Video Broadcasting

Introduction to the AES3 Digital Audio Standard ........................................... 393
Data Format ......................................................................................... 393
Subframes, Frames, and Blocks ................................................................. 394
Preambles ......................................................................................... 394
Biphase-Mark Encoding .......................................................................... 395
Valid Bit ............................................................................................ 395
Channel Status .................................................................................. 395
User Data ........................................................................................... 395
Parity Bit ............................................................................................ 395
Data Rate ............................................................................................ 396
SMPTE 272M: Embedded Digital Audio for SD-SDI ................................. 396
SD Embedded Audio Packets ................................................................. 397
SD Audio Data Packets .......................................................................... 397
SD Extended Data Packets .................................................................... 399
SD Audio Control Packets .................................................................... 400
SD Audio Sample Distribution ............................................................... 402
SMPTE 299M: Embedded Digital Audio for HD-SDI ................................. 402
HD Audio Data Packets ......................................................................... 402
HD Audio Control Packets ..................................................................... 403
HD Audio Sample Distribution ............................................................... 403
Audio Sample Rate Conversion .............................................................. 404
Introduction ....................................................................................... 404
Clarification of Terms .......................................................................... 405
Methods of Sample Rate Conversion ..................................................... 405
Synchronous ...................................................................................... 405
Asynchronous ...................................................................................... 406
ASRC Operation .................................................................................... 408
Ratio Control ....................................................................................... 409
Resampler ......................................................................................... 409
Synchronous SRC Operation ................................................................. 411
Lagrange Interpolation of Filter Coefficients ........................................... 412
Performance Factors .......................................................................... 413
Prototype Filter Design ........................................................................ 414
Typical Applications ............................................................................ 414
Digital Audio Reference Designs ........................................................... 416

Chapter 19: AES3 Serial Digital Audio Interfaces for Xilinx FPGAs

Summary .............................................................................................. 417
Reference Design .................................................................................. 417
Receiver .............................................................................................. 417
Transmitter .......................................................................................... 424
Channel Status CRC ............................................................................ 426
Spartan-3E SDV Board AES3 Demonstration ........................................ 428
Chapter 20: Asynchronous Sample Rate Converter

Summary .................................................................................................................. 433
Structure .................................................................................................................. 433
  Modules .................................................................................................................. 434
Functional Description .......................................................................................... 436
  Ratio Control Functional Block ............................................................................ 436
    Ratio Detection .................................................................................................. 436
    Ratio Calculation ............................................................................................... 438
    Ratio Filtering for Jitter Tolerance ...................................................................... 439
    Ratio Regulation ............................................................................................... 441
    Lock Status Indicators ...................................................................................... 442
  Input Sample Storage ............................................................................................ 442
    Clock Domain Considerations .......................................................................... 445
  Resampler Functional Block ................................................................................ 446
    Prototype Filter ................................................................................................ 446
    Coefficient Interpolation ................................................................................... 449
    FIR Filter ......................................................................................................... 450
    Shared Divider ................................................................................................. 451
  Control .................................................................................................................. 451
  Interface Timing ................................................................................................... 453
Performance ............................................................................................................ 454
  THD + N ................................................................................................................ 454
  Latency .................................................................................................................. 456
FPGA Resource Utilization and Performance ....................................................... 457
  Additional Channels ............................................................................................ 457
  Data Flow Spreadsheet ....................................................................................... 459
    Filter Phase Interpolation and FIR Filter ............................................................ 460
    Output Sample Normalization .......................................................................... 460
Design Files ............................................................................................................. 460
Conclusion ............................................................................................................... 460

Chapter 21: AES3 Audio Demultiplexer for Standard-Definition Digital Audio

Specifications and Features ..................................................................................... 461
Usage Models .......................................................................................................... 463
  One Video Stream with Video Rate Clock ............................................................ 463
  One Video Stream with Faster Clock .................................................................... 463
  Multiple Synchronous Video Streams .................................................................. 464
  Multiple Asynchronous Video Streams ................................................................. 466
  Clock Requirements for Processing Multiple Video Streams ............................. 466
Modules ................................................................................................................... 466
I/O Port Description ................................................................................................. 469
  Clock and Control Signals .................................................................................... 469
  Input Video Streams ............................................................................................. 470
Audio Sample Output Ports ................................................................. 471
Channel Pair Present Flags ............................................................... 473
Channel Pair Demux Control Ports ..................................................... 474
Input Packet Error Ports ................................................................. 475
Audio Control Packet Ports .............................................................. 477
Audio Packet Deletion Ports ............................................................ 478

Parameters ................................................................................. 478
Sample Buffer Size ......................................................................... 479

FPGA Resource Requirements ......................................................... 480

Theory of Operation ......................................................................... 480
  Overview ................................................................................... 480
  Input State Machine ................................................................... 482
  Output State Machine ............................................................... 485
  Audio Packet Deletion ............................................................... 485
  Input Stream Control Module ..................................................... 486

Design Files .................................................................................. 487

Conclusions ................................................................................... 487

Section VII:
Appendixes

Appendix A: References ................................................................. 491
Schedule of Figures

Chapter 1:  Introduction

Chapter 2:  SD-SDI Physical Layer Implementation

Figure 2-1:  SDI Block Diagram and SD-SDI Section Chapters .......................... 41
Figure 2-2:  SDI Pass-Through Block Diagram ................................................. 42
Figure 2-3:  Pathological Waveforms .............................................................. 44
Figure 2-4:  Cable Equalization Example .......................................................... 45
Figure 2-5:  Jitter Reduction ............................................................................. 46
Figure 2-6:  Using DCM and DDR for SDI Serializer ........................................... 48
Figure 2-7:  SDI Cable Driver Example ............................................................. 49
Figure 2-8:  SDI Encoding and Decoding ............................................................ 50
Figure 2-9:  Interfacing an External SDI Reclocker to Xilinx FPGAs ....................... 51
Figure 2-10: Internal CDR for SDI Receiver ....................................................... 52
Figure 2-11: 270 MHz VCO for SDI CDR ........................................................... 53
Figure 2-12: XAPP250 Input Jitter Tolerance ....................................................... 54
Figure 2-13: XAPP224 and VCO Implement CDR ............................................... 56
Figure 2-14:  XAPP250-Based SDI Receiver Reference Design .............................. 56
Figure 2-15:  SDI Receiver Reference Design for External SDI Reclocker ................. 57

Chapter 3:  SD-SDI Video Encoder

Figure 3-1:  SDI Block Diagram and SD-SDI Section Chapters .......................... 59
Figure 3-2:  SDI Encoding and Decoding Processes ............................................. 61
Figure 3-3:  Example SDI Transmitter with TRS Clipper ...................................... 63
Figure 3-4:  Bit-Rate Serial SDI Scrambler ......................................................... 64
Figure 3-5:  SDI Scrambler Processing Two Bits Per Clock Cycle ......................... 64
Figure 3-6:  Parallel Scrambler Block Diagram ................................................. 65
Figure 3-7:  X9002 Example Block Diagram ...................................................... 66
Figure 3-8:  X7C9235 Example Block Diagram .................................................. 67
Figure 3-9:  SDI Test-Bench Block Diagram ...................................................... 68

Chapter 4:  SD-SDI Video Decoder

Figure 4-1:  SDI Block Diagram and SD-SDI Section Chapters .......................... 71
Figure 4-2:  Example SDI Encoder and Decoder Processes ................................... 73
Figure 4-3:  Parallel Descrambler Block Diagram .............................................. 75
Figure 4-4:  Serial Framer Block Diagram ......................................................... 77
Figure 4-5:  SMPTE 259M-1997 Parallel Framer Block Diagram ......................... 79
Figure 4-6:  X7C9335 Block Diagram ............................................................... 80
Chapter 5: SD-SDI Video Flywheel

Figure 5-1: SDI Block Diagram and Application Notes ................................. 83
Figure 5-2: Video Decoder Block Diagram .................................................. 89
Figure 5-3: Timing Diagram for video_decode ............................................. 89
Figure 5-4: trs_detect Block Diagram ......................................................... 90
Figure 5-5: trs_detect Module Timing ......................................................... 91
Figure 5-6: autodetect Block Diagram ....................................................... 92
Figure 5-7: autodetect FSM ACQUIRE Loop ............................................. 93
Figure 5-8: autodetect FSM LOCKED Loop ............................................... 94
Figure 5-9: flywheel Block Diagram ......................................................... 95
Figure 5-10: flywheel FSM State Diagram Main Loop ................................ 96
Figure 5-11: flywheel FSM State Diagram Synchronous Switching .............. 97

Chapter 6: SD-SDI Ancillary Data and EDH Processors

Figure 6-1: SDI Block Diagram and Related Application Notes ..................... 101
Figure 6-2: EDH and ANC Processor Block Diagram .................................. 102
Figure 6-3: ANC Packet Format ............................................................... 103
Figure 6-4: Non-conforming ANC Packets ............................................... 105
Figure 6-5: Available ANC Spaces in NTSC Frame .................................... 107
Figure 6-6: Overwriting an ANC Packet Marked for Deletion ...................... 109
Figure 6-7: CRC Calculations ................................................................. 112
Figure 6-8: NTSC 13.5 MHz 4:2:2 CRC Calculations and EDH Packet Positions 112
Figure 6-9: NTSC 18 MHz 4:2:2 CRC Calculations and EDH Packet Position 113
Figure 6-10: NTSC 4:4:4:4 CRC Calculations and EDH Packet Positions ....... 114
Figure 6-11: PAL 13.5 MHz 4:2:2 CRC Calculations and EDH Packet Positions 115
Figure 6-12: PAL 18 MHz 4:2:2 CRC Calculations and EDH Packet Positions 116
Figure 6-13: PAL 4:4:4:4 CRC Calculations and EDH Packet Positions ........ 117
Figure 6-14: Error Flag Forwarding .......................................................... 119
Figure 6-15: EDH Packet Format ............................................................. 119
Figure 6-16: ANC and EDH Processor Block Diagram ............................... 120
Figure 6-17: EDH Processor Block Diagram ............................................ 124
Figure 6-18: edh_rx Block Diagram .......................................................... 125
Figure 6-19: edh_rx State Diagram ......................................................... 126
Figure 6-20: anc_rx Block Diagram .......................................................... 127
Figure 6-21: anc_rx State Diagram ........................................................... 128
Figure 6-22: edh_loc Block Diagram ....................................................... 129
Figure 6-23: edh_crc Block Diagram ....................................................... 130
Figure 6-24: anc_mux Block Diagram ....................................................... 131
Figure 6-25: anc_insert Block Diagram .................................................... 132
Figure 6-26: anc_insert State Diagram ..................................................... 133
Figure 6-27: anc_pkt_gen Block Diagram .................................................. 134
Figure 6-28: anc_pkt_gen State Diagram ................................................... 136
Figure 6-29: anc_extract Block Diagram .................................................. 137
Chapter 7: Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor

Figure 7-1: Video Timing Signals .................................................. 152
Figure 7-2: rtx EDH Packet Flags and Error Detection Logic .......... 154
Figure 7-3: EDH and Error Flags Timing ...................................... 154
Figure 7-4: rtx EDH Processor Block Diagram ............................. 157
Figure 7-5: Horizontal Position Registers .................................. 161
Figure 7-6: Vertical Line Count Registers ................................... 162
Figure 7-7: std_reg, ctrl0_reg, and sync_switch Registers .......... 163
Figure 7-8: Simplified State Diagram (a) ..................................... 168
Figure 7-9: Simplified State Diagram (b) ................................. 169
Figure 7-10: Simplified State Diagram (c) .................................... 170

Chapter 8: SD-SDI Integration Example for the Serial Digital Video Demonstration Board

Figure 8-1: SD-SDI Application Block Diagram .......................... 173
Figure 8-2: Alternative DRU Clocking Scheme ............................ 176

Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Figure 9-1: HD-SDI Line Format .................................................. 184
Figure 9-2: XYZ Word Format .................................................... 185
Figure 9-3: Line Number Format ................................................ 185
Figure 9-4: CRC Calculation ....................................................... 186
Figure 9-5: CRC Format ............................................................. 186
Figure 9-6: Interleaved Data Stream ......................................... 186
Figure 9-7: HD-SDI Encoder Serial Implementation ..................... 187
Figure 9-8: HD-SDI Transmitter Output Jitter Bands .................... 188
Figure 9-9: Interfacing the GS1528 Cable Driver to the RocketIO Transmitter ..................... 189
Figure 9-10: REFCLK MUXes .................................................... 190
Figure 9-11: Jitter Reduction ...................................................... 192
Figure 9-12: Xilinx SDV Board HD-SDI Transmitter Reference Design ............. 193
Figure 9-13: hdsdi_tx Block Diagram .......................................... 194
Figure 9-14: hdsdi_tx_path Block Diagram ................................. 195
Figure 9-15: hdsdi_rio Module .................................................... 196
Figure 9-16: Xilinx SDV Demo Board Eye Diagram Measurements ................. 197
Figure 9-17: hdsdi_autodetect_In Module ................................. 199
Chapter 10: HD-SDI Receiver Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Figure 10-1: HD-SDI Receiver Block Diagram .............................................. 204
Figure 10-2: HD-SDI Decoding Algorithm .................................................. 205
Figure 10-3: XYZ Word Format ................................................................. 206
Figure 10-4: Interleaved Data Stream ........................................................ 206
Figure 10-5: Framer Example ................................................................. 207
Figure 10-6: SMPTE 292M Jitter Template ............................................... 209
Figure 10-7: Interfacing a Cable Equalizer to the RocketIO Receiver ........ 210
Figure 10-8: Typical RocketIO Transceiver Clock Connections for an HD-SDI Receiver ................................................................. 213
Figure 10-9: Xilinx SDV Demo Board HD-SDI Receiver Reference Design .... 214
Figure 10-10: hdsdi_rx Block Diagram .................................................... 215
Figure 10-11: hdsdi_rio Module ............................................................... 216
Figure 10-12: HD-SDI Receiver Input Jitter Tolerance ............................ 217
Figure 10-13: hdsdi_decoder Block Diagram ............................................ 218
Figure 10-14: hdsdi_framer Block Diagram ............................................. 219
Figure 10-15: hdsdi_framer_mult Barrel Shifter ..................................... 221
Figure 10-16: MULT18X18 used as Nine 2:1 MUXes .............................. 222
Figure 10-17: MULT18X18 used as Seven 12:1 MUXes ......................... 223
Figure 10-18: hdsdi_rx_autorate State Diagram .................................... 225
Figure 10-19: Problem with Sharing REFCLK Between Transmitter and Receiver ................................................................. 226
Figure 10-20: Using a VCXO to Lock the Reference Clock to RXRECCLK ... 227
Figure 10-21: ICS664 Providing REFCLK to RocketIO Transceiver .......... 228

Chapter 11: HD-SDI Integration Examples for the Serial Digital Video Demonstration Board

Figure 11-1: Separate HDS-SDI Tx and Rx Block Diagram ....................... 232
Figure 11-2: HD-SDI Pass-through Block Diagram .................................. 235
Figure 11-3: VCXO and Loop Filter ....................................................... 237
Figure 11-4: HD-SDI Pass-Through using ICS664 Block Diagram ............ 238
Figure 11-5: Loop Filter, 27 MHz VCXO, and ICS664-01 ....................... 239

Chapter 12: Multi-Rate HD/SD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Figure 12-1: RocketIO Transmitter Producing a 270 Mb/s SD-SDI Bitstream .... 245
Figure 12-2: Interfacing the GS1528 Cable Driver to the RocketIO Transmitter .... 246
Figure 12-3: Clocking Example .............................................................. 248
Chapter 13: Multi-Rate HD/SD-SDI Receiver Using RocketIO Multi-Gigabit Transceivers

Figure 12-4: SDI Encoding Algorithm ................................................................. 250
Figure 12-5: Multi-Rate SDI Encoder Module ............................................... 251
Figure 12-6: SD-SDI Bit Replication ................................................................. 252
Figure 12-7: Multi-Rate SDI Transmitter Example ........................................ 253
Figure 12-8: SD-SDI Transmitter Example ....................................................... 253

Chapter 13: Multi-Rate HD/SD-SDI Receiver Using RocketIO Multi-Gigabit Transceivers

Figure 13-1: Multi-rate SDI Receiver Block Diagram ....................................... 258
Figure 13-2: SDI Decoding Algorithm ............................................................... 259
Figure 13-3: XYZ Word Format ........................................................................ 260
Figure 13-4: HD-SDI Interleaved Data Stream ................................................ 261
Figure 13-5: SD-SDI Data Stream .................................................................... 261
Figure 13-6: Interfacing a Cable Equalizer to the RocketIO Transceiver ........... 263
Figure 13-7: RocketIO Transceiver and Data Recovery Unit ............................. 265
Figure 13-8: Synthesis of SD-SDI Recovered Clock (DRU Version 1) ............ 266
Figure 13-9: Clock Enable Timing (DRU Version 2) ......................................... 267
Figure 13-10: Reference Clock Selection ........................................................ 268
Figure 13-11: Xilinx SDV Demo Board Multi-Rate SDI Receiver Reference Design (DRU Version 1) ................................................................. 270
Figure 13-12: Xilinx SDV Demo Board Multi-Rate SDI Receiver Reference Design (DRU Version 2) ................................................................. 271
Figure 13-13: multi_sdi_rx Block Diagram ....................................................... 276
Figure 13-14: multi_sdi_decoder Block Diagram ............................................. 278
Figure 13-15: multi_sdi_framer Block Diagram .............................................. 279
Figure 13-16: multi_sdi_rx_autorate State Diagram ........................................ 280

Chapter 14: Multi-Rate SDI Integration Examples for the Serial Digital Video Demonstration Board

Figure 14-1: Reference Design Top-Level Block Diagram ................................. 283
Figure 14-2: Multi-Rate Receiver Module Block Diagram ............................... 285
Figure 14-3: 74.1758 MHz VCXO and Loop Filter ......................................... 286
Figure 14-4: 27 MHz VCXO, Loop Filter, ICS8745 PLL, and ICS664-01 .......... 287
Figure 14-5: Multi-Rate SDI Pass-Through Block Diagram ............................ 288
Figure 14-6: Multi-Rate Receiver Module with PicoBlaze EDH Processor .......... 289
Figure 14-7: Multi-Rate SDI Pass-Through Using ICS664 Block Diagram .......... 291
Figure 14-8: Multi-Rate Receiver Module for Application 3 ............................. 292

Chapter 15: DVB-ASI Physical Layer Implementation

Figure 15-1: DVB-ASI Protocol Stack ............................................................... 297
Figure 15-2: Data Path of DVB-ASI Transmitter and Receiver ....................... 298
Figure 15-3: DVB-ASI Receiver Block Diagram ............................................. 299
Figure 15-4: ASI Receiver ............................................................................... 300
Figure 15-46: DVB-ASI Transmitter Block Diagram .............................................. 340
Figure 15-47: RocketIO Transmitter Producing a 270 Mb/s DVB-ASI Bitstream ....... 340
Figure 15-48: Generating 8B/10B Encoder from Core Generator ....................... 343
Figure 15-49: Generating 8B/10B Encoder from Core Generator ....................... 344
Figure 15-50: DVB-ASI Bit Replication .............................................................. 344

Chapter 16: SDTV Video Pattern Generators

Figure 16-1: NTSC and PAL Video Line Detail .................................................... 351
Figure 16-2: NTSC Video Frame Details ............................................................ 353
Figure 16-3: PAL Video Frame Details ............................................................... 354
Figure 16-4: EIA-189-A Color Bar Pattern ......................................................... 356
Figure 16-5: SMPTE EG 1-1990 Color Bar Pattern ............................................ 357
Figure 16-6: Distributed RAM EG 1 Pattern Generator ................................. 359
Figure 16-7: Horizontal Regions of the EG 1 Test Pattern Generator .............. 360
Figure 16-8: Vertical Regions of the EG 1 Test Pattern Generator ................. 362
Figure 16-9: Video Pattern Generator Using Block RAMs ............................... 364

Chapter 17: HDTV Video Pattern Generator

Figure 17-1: HDTV Video Line Format .............................................................. 380
Figure 17-2: XYZ Word Format .......................................................................... 380
Figure 17-3: SMPTE RP 219-2002 Color Bar Pattern (16:9 Aspect Ratio) ...... 382
Figure 17-4: 75% Color Bars Pattern (16:9 Aspect Ratio) ............................. 382
Figure 17-5: HD-SDI Pathological Waveforms ................................................. 383
Figure 17-6: RP 198-1998 Checkfield ............................................................... 384
Figure 17-7: Video Pattern Generator Block Diagram ..................................... 386
Figure 17-8: Y-Ramp Generator ........................................................................ 388

Chapter 18: Introduction to Digital Audio for Video Broadcasting

Figure 18-1: AES3 Subframe Format ................................................................. 394
Figure 18-2: SD Audio Data Packet Format ...................................................... 398
Figure 18-3: SD Extended Data Packet Format ................................................ 399
Figure 18-4: SD Audio Control Packet Format ................................................ 400
Figure 18-5: HD Audio Data Packet Format ..................................................... 403
Figure 18-6: Sample Rate Conversion ............................................................... 404
Figure 18-7: Classic Up-Conversion ................................................................. 405
Figure 18-8: Classic Down-Conversion ............................................................ 405
Figure 18-9: Classic Sample Rate Conversion By a Rational Number ............ 405
Figure 18-10: Classic Method of Asynchronous Sample Rate Conversion ...... 406
Figure 18-11: Interpolated Coefficient FIR Filtering ........................................ 407
Figure 18-12: Prototype Filter Centered at Output Sample Position ............... 407
Figure 18-13: ASRC Inputs and Outputs .......................................................... 408
Figure 18-14: ASRC Major Functions .............................................................. 409
Figure 18-15: Up-Conversion Example ............................................................ 410
Chapter 19: AES3 Serial Digital Audio Interfaces for Xilinx FPGAs

Figure 19-1: AES3 Receiver Block Diagram ............................................................ 418
Figure 19-2: aes_rx Demultiplexed Mode Timing Diagram .............................. 420
Figure 19-3: aes_rx Multiplexed Mode Timing Diagram ............................... 421
Figure 19-4: Brute Force Deserialization of C and U Data ............................. 422
Figure 19-5: Using Dual-Port Distributed Memory for C and U Data ............. 423
Figure 19-6: Transmitter Timing Diagram (24.576 MHz Clock and 48 kHz Sample Rate) .......................................................... 424
Figure 19-7: Checking C CRC in an AES Receiver ........................................... 427
Figure 19-8: Generating C CRC for an AES Transmitter .............................. 427
Figure 19-9: Spartan-3E SDV Board AES3 Demonstration Block Diagram .... 429
Figure 19-10: ChipScope Bus Plot Window Showing Audio Data from aes_rx Module .......................................................... 429
Figure 19-11: Status LEDs and Pushbuttons in Spartan-3E SDV AES3 Audio Demo .......................................................... 431
Figure 19-12: DIP Switches in Spartan-3E SDV AES3 Audio Demo ................ 431

Chapter 20: Asynchronous Sample Rate Converter

Figure 20-1: ASRC Top Level Block Diagram ................................................ 434
Figure 20-2: Reference Design Module Hierarchy and Relation to Functional Blocks .......................................................... 435
Figure 20-3: Error Correction Curves ................................................................. 436
Figure 20-4: Ratio Detection Block Diagram ..................................................... 437
Figure 20-5: Ratio Calculation Detailed Block Diagram .................................... 438
Figure 20-6: Timing Diagram of Input Period Measurement with max_count = 4 .... 439
Figure 20-7: Ratio Filter .................................................................................. 440
Figure 20-8: Detailed Block Diagram of Ratio Regulation Section .................. 441
Figure 20-9: Ring Buffer ................................................................................ 443
Figure 20-10: Input Buffer Storage Block Diagram ......................................... 444
Figure 20-11: FIFO Level Calculation with Fractional Bits ............................ 444
Figure 20-12: Rising Edge Detect .................................................................. 445
Figure 20-13: Resampler ............................................................................... 446
Figure 20-14: Prototype Filter Frequency Response ....................................... 447
Figure 20-15(a): Prototype Filter Transition Band (Calculated) ................. 447
Figure 20-15(b): Prototype Filter Transition Band (Measured) ................. 448
Figure 20-16(a): Prototype Filter Passband (Calculated) ............................. 448
Chapter 21: AES3 Audio Demultiplexer for Standard-Definition Digital Audio

Figure 21-1: One-Input Video Stream with 27 MHz Video Clock ............................ 463
Figure 21-2: One-Input Video Stream with High-Speed Clock & Input Clock Enable 463
Figure 21-3: Multiple Synchronous Video Streams ................................. 465
Figure 21-4: Multiple Asynchronous Video Streams .............................. 467
Figure 21-5: Driving the Channel Pair Demux Enables with a Mux .................... 475
Figure 21-6: Driving the Channel Pair Demux Enables with a RAM .................. 475
Figure 21-7: pkt_start and pkt_cs_err Timing ............................................ 476
Figure 21-8: Audio Control Packet Output Timing ........................................ 477
Figure 21-9: sd_aes_demux Block Diagram ............................................... 481
Figure 21-10: Sample Buffer Data Formats .............................................. 481
Figure 21-11: Input FSM Audio Data Packet Processing .............................. 483
Figure 21-12: Input FSM Extended Data Packet Processing ........................... 484
Figure 21-13: Input FSM Audio Control Packet Processing ........................... 484
Figure 21-14: Output FSM State Diagram ................................................ 486
Schedule of Tables

Chapter 1: Introduction

Chapter 2: SD-SDI Physical Layer Implementation
  Table 2-1: Reference Design Implementation Results .......................... 57

Chapter 3: SD-SDI Video Encoder
  Table 3-1: SDI Standard Bit Rates ................................................. 62
  Table 3-2: Design Results .......................................................... 70

Chapter 4: SD-SDI Video Decoder
  Table 4-1: SDI Standard Bit Rates ................................................. 74
  Table 4-2: Design Results .......................................................... 81

Chapter 5: SD-SDI Video Flywheel
  Table 5-1: SDI Supported Digital Video Standards ......................... 84
  Table 5-2: XYZ Word Format for the 4:4:4:4 TRS Symbol .................... 85
  Table 5-3: Words Per Video Line .................................................. 86
  Table 5-4: Field Starting Line Numbers ........................................ 87
  Table 5-5: RP 168 Synchronous Switching Line Numbers .................... 88
  Table 5-6: Reference Design Results .......................................... 99

Chapter 6: SD-SDI Ancillary Data and EDH Processors

Chapter 7: Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor
  Table 7-1: I/O Signals .............................................................. 148
  Table 7-2: EDH Error Flag Port Bit Assignments ............................. 153
  Table 7-3: err_flg_en Bit Assignment ............................................ 155
  Table 7-4: FPGA Resource Usage ................................................ 159

Chapter 8: SD-SDI Integration Example for the Serial Digital Video Demonstration Board
  Table 8-1: FPGA Resources Used in the SD-SDI Demonstration Application ... 178

Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers
  Table 9-1: HD-SDI Compatible Video Formats from SMPTE 292M ............ 182
Table 9-2: Segmented Frame Video Formats from RP 211 ........................................ 183
Table 9-3: Typical HD-SDI Transmitter Output Jitter Values ................................. 197
Table 9-4: Reference Design Implementation Sizes ............................................. 198

Chapter 10: HD-SDI Receiver Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers
Table 10-1: Reference Design Implementation Sizes ........................................... 217

Chapter 11: HD-SDI Integration Examples for the Serial Digital Video Demonstration Board
Table 11-1: FPGA Resources Used by the Separate HD-SDI Tx and Rx Application . 234
Table 11-2: FPGA Resources Used by the HD-SDI Pass-Through Application ........ 237
Table 11-3: FPGA Resources Used by the HD-SDI Pass-Through with the ICS664-01 Application .......................................................... 239

Chapter 12: Multi-Rate HD/SD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers
Table 12-1: Typical SD-SDI Transmitter Output Jitter Values Using RocketIO MGTs 254
Table 12-2: Reference Design Implementation Sizes ............................................. 254

Chapter 13: Multi-Rate HD/SD-SDI Receiver Using RocketIO Multi-Gigabit Transceivers
Table 13-1: Version 2 DRU Filenames ................................................................. 273
Table 13-2: Reference Design Implementation Sizes ............................................. 277

Chapter 14: Multi-Rate SDI Integration Examples for the Serial Digital Video Demonstration Board
Table 14-1: FPGA Resources Used by Application Example 1 ............................ 287
Table 14-2: FPGA Resources Used by Application Example 2 ............................ 290
Table 14-3: FPGA Resources Used by Application Example 3 ............................ 293

Chapter 15: DVB-ASI Physical Layer Implementation
Table 15-1: DVB-ASI Receiver Implementation Details .................................... 300
Table 15-2: Table of User Parameters ............................................................... 307
Table 15-3: DVB-ASI Transmitter Implementation Details ................................. 313
Table 15-4: DVB-ASI Pass-Through Implementation Details .............................. 319
Table 15-5: User Parameters ............................................................................ 321
Table 15-6: SDV Demo Board Settings ............................................................. 322
Table 15-7: Default Parameter Settings ............................................................. 323
Table 15-8: Resource Utilization, RocketIO MGT Implementation ...................... 345
Chapter 16: SDTV Video Pattern Generators

Table 16-1: Common 4:2:2 Component Digital Video Standards ........................................ 349
Table 16-2: Reference Design Results ................................................................. 367

Chapter 17: HDTV Video Pattern Generator

Table 17-1: Common HD 4:2:2 Component Digital Video Standards .............................. 378
Table 17-2: Video Format Groups Supported by Video Pattern Generator ...................... 385
Table 17-3: Reference Design Results ................................................................. 390

Chapter 18: Introduction to Digital Audio for Video Broadcasting

Table 18-1: AES3 Bit Rates ..................................................................................... 396
Table 18-2: DID Values for SD Embedded Audio Packets ........................................ 397
Table 18-3: Audio Sample Rate Codes .................................................................... 401
Table 18-4: DID Values for HD Embedded Audio Packets ...................................... 402

Chapter 19: AES3 Serial Digital Audio Interfaces for Xilinx FPGAs

Table 19-1: aes_rx Module Ports ........................................................................... 419
Table 19-2: aes_tx Module Ports ........................................................................... 425
Table 19-3: aes_crc Module Ports ........................................................................ 428
Table 19-4: LED Displays for Various Audio Sample Rates .................................... 430
Table 19-5: FPGA Resource Usage for AES3 Modules ........................................ 432

Chapter 20: Asynchronous Sample Rate Converter

Table 20-1: Reference Design Module Descriptions .................................................. 434
Table 20-2: Prototype Filter Parameters .................................................................. 446
Table 20-3: THD + N Performance vs. Conversion Frequency .................................... 455
Table 20-4: Reference Design Resource Utilization .................................................. 457
Table 20-5: Reference Design Frequency Performance ............................................. 457
Table 20-6: 4-Channel Resource Utilization ............................................................ 458
Table 20-7: 8-Channel Resource Utilization ............................................................ 458

Chapter 21: AES3 Audio Demultiplexer for Standard-Definition Digital Audio

Table 21-1: List of Modules ..................................................................................... 468
Table 21-2: Clock and Control Signals ..................................................................... 469
Table 21-3: Input Video Stream for Single Stream Module (sd_aes_demux_1) .......... 471
Table 21-4: Input Video Streams for Asynchronous Multi-Stream Modules ............. 471
Table 21-5: Input Video Streams for Synchronous Multi-Stream Modules .............. 471
Table 21-6: Audio Sample Output Ports ................................................................. 472
Table 21-7: Channel Pair Present Flags for sd_aes_demux_1 ................................... 473
Table 21-8: Channel Pair Present Flags (Multi-Stream Modules) ............................. 473
Table 21-9: Channel Pair Present Flag Vector Mapping .......................................... 473
Table 21-10: Channel Pair Filtering Ports ................................. 474
Table 21-11: Input Packet Status Ports ................................. 475
Table 21-12: Audio Control Packet Ports ................................. 477
Table 21-13: Audio Control Packet Ports ................................. 478
Table 21-14: FPGA Resources ........................................... 480
Preface

About This Guide

Guide Contents

This manual contains the following sections and chapters. For reference purposes, chapters originally published as Xilinx Application Notes show the original XAPP number.

**Note:** The stand-alone application notes referenced are no longer current and have been retired. *The content of this volume has been updated and supersedes the original application notes in all cases.*

- Chapter 1, “Introduction”  
  by Gregg Hawkes

Section I: SD-SDI

- Chapter 2, “SD-SDI Physical Layer Implementation”  
  by John Snow (originally published as XAPP247)
- Chapter 3, “SD-SDI Video Encoder”  
  by John Snow (originally published as XAPP298)
- Chapter 4, “SD-SDI Video Decoder”  
  by John Snow (originally published as XAPP288)
- Chapter 5, “SD-SDI Video Flywheel”  
  by John Snow (originally published as XAPP625)
- Chapter 6, “SD-SDI Ancillary Data and EDH Processors”  
  by John Snow (originally published as XAPP299)
- Chapter 7, “Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor”  
  by John Snow (originally published as XAPP580)
- Chapter 8, “SD-SDI Integration Example for the Serial Digital Video Demonstration Board”  
  by John Snow (originally published as XAPP578)

Section II: HD-SDI

- Chapter 9, “HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers”  
  by John Snow (originally published as XAPP680)
- Chapter 10, “HD-SDI Receiver Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers”  
  by John Snow (originally published as XAPP681)
Preface: About This Guide

- Chapter 11, “HD-SDI Integration Examples for the Serial Digital Video Demonstration Board”
  by John Snow (originally published as XAPP577)

Section III: Multi-Rate HD/SD-SDI

- Chapter 12, “Multi-Rate HD/SD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers”
  by John Snow (originally published as XAPP683)
- Chapter 13, “Multi-Rate HD/SD-SDI Receiver Using RocketIO Multi-Gigabit Transceivers”
  by John Snow (originally published as XAPP684)
- Chapter 14, “Multi-Rate SDI Integration Examples for the Serial Digital Video Demonstration Board”
  by John Snow (originally published as XAPP579)

Section IV: DVB-ASI

- Chapter 15, “DVB-ASI Physical Layer Implementation”
  by Tze Yi Yeoh (originally published as XAPP509)

Section V: Video Test Pattern Generators

- Chapter 16, “SDTV Video Pattern Generators”
  by John Snow (originally published as XAPP248)
- Chapter 17, “HDTV Video Pattern Generator”
  by John Snow (originally published as XAPP682)

Section VI: Digital Audio

- Chapter 18, “Introduction to Digital Audio for Video Broadcasting”
  by John Snow and Reed Tidwell (initial Xilinx publication)
- Chapter 19, “AES3 Serial Digital Audio Interfaces for Xilinx FPGAs”
  by John Snow (initial Xilinx publication)
- Chapter 20, “Asynchronous Sample Rate Converter”
  by Reed Tidwell (initial Xilinx publication)
- Chapter 21, “AES3 Audio Demultiplexer for Standard-Definition Digital Audio”
  by John Snow (initial Xilinx publication)

Section VII: Appendixes

- Appendix A, “References”
Additional Resources

For additional information, go to http://www.xilinx.com/support/. The following table lists some of the resources you can access from this website. You can also directly access these resources using the provided URLs.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description/URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorials</td>
<td>Tutorials covering Xilinx design flows, from design entry to verification and debugging</td>
</tr>
<tr>
<td>Answer Browser</td>
<td>Database of Xilinx solution records</td>
</tr>
<tr>
<td>Application Notes</td>
<td>Descriptions of device-specific design techniques and approaches</td>
</tr>
<tr>
<td>Data Sheets</td>
<td>Device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp">http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp</a></td>
</tr>
<tr>
<td>Problem Solvers</td>
<td>Interactive tools that allow you to troubleshoot your design issues</td>
</tr>
<tr>
<td>Tech Tips</td>
<td>Latest news, design tips, and patch information for the Xilinx design environment</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.xilinx.com/xlnx/xil_tt_home.jsp">http://www.xilinx.com/xlnx/xil_tt_home.jsp</a></td>
</tr>
</tbody>
</table>

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning or Use</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courier font</td>
<td>Messages, prompts, and program files that the system displays</td>
<td>speed grade: - 100</td>
</tr>
<tr>
<td>Courier bold</td>
<td>Literal commands that you enter in a syntactical statement</td>
<td>ngdbuild design_name</td>
</tr>
<tr>
<td>Helvetica bold</td>
<td>Commands that you select from a menu</td>
<td>File →Open</td>
</tr>
<tr>
<td></td>
<td>Keyboard shortcuts</td>
<td>Ctrl+C</td>
</tr>
<tr>
<td>Convention</td>
<td>Meaning or Use</td>
<td>Example</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td><em>Italic font</em></td>
<td>Variables in a syntax statement for which you must supply values</td>
<td><code>ngdbuild design_name</code></td>
</tr>
<tr>
<td></td>
<td>References to other manuals</td>
<td>See the <em>Development System Reference Guide</em> for more information.</td>
</tr>
<tr>
<td></td>
<td>Emphasis in text</td>
<td>If a wire is drawn so that it overlaps the pin of a symbol, the two nets are <em>not</em> connected.</td>
</tr>
<tr>
<td>Square brackets [ ]</td>
<td>An optional entry or parameter. However, in bus specifications, such as <code>bus[7:0]</code>, they are required.</td>
<td><code>ngdbuild [option_name] design_name</code></td>
</tr>
<tr>
<td>Braces { }</td>
<td>A list of items from which you must choose one or more</td>
<td>`lowpwr = {on</td>
</tr>
<tr>
<td>Vertical bar</td>
<td>Separates items in a list of choices</td>
<td>`lowpwr = {on</td>
</tr>
</tbody>
</table>
| Vertical ellipsis . . . | Repetitive material that has been omitted | `IOB #1: Name = QOUT’
IOB #2: Name = CLKN’ .
. . .` |
| Horizontal ellipsis ... | Repetitive material that has been omitted | `allow block block_name loc1 loc2 ... locn;` |

**Online Document**

The following conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning or Use</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue text</td>
<td>Cross-reference link to a location in the current document</td>
<td>See the section “Additional Resources” for details. Refer to “Title Formats” in Chapter 1 for details.</td>
</tr>
<tr>
<td>Red text</td>
<td>Cross-reference link to a location in another document</td>
<td>See Figure 2-5 in the <em>Virtex-II Handbook</em>.</td>
</tr>
<tr>
<td>Blue, underlined text</td>
<td>Hyperlink to a website (URL)</td>
<td>Go to <a href="http://www.xilinx.com">http://www.xilinx.com</a> for the latest speed files.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Television is without doubt one of the truly world-shaping inventions of the 20th century, and quite a number of individual engineers in different countries have contributed to its evolution. Many engineers disagree on the future of television, just as historians might disagree on the exact date the television concept was created. However, no one can argue about its impact on the lives of billions of people in today’s world.

The first analog transmission networks for carrying television pictures and sound, just like the technology of television itself, began to come to life in the late 1930s with the invention of coaxial cable. By the 1990s, TV broadcast studios contained miles and miles of analog coaxial cable — but the world was fast becoming a digital landscape. To leverage the existing coaxial cable transport fabric, engineers created a method whereby digital streams of information could be transferred across legacy analog networks, and the Serial Digital Interface transmission standard (SDI) was born. Companies formed entire product lines to support the explosive popularity of this new standard. Specialized digital chips called Application-Specific Standard Products (ASSPs) addressed the complex needs of the protocol and found their way into every SDI system where a coaxial cable began or ended.

As entertainment consumers demanded more and more realistic immersion into the television experience, broadcasting movies in wide-screen dimensions promised more income for the television industry. Video resolution would increase, and aging coaxial cables would be forced to carry even more information. Engineers first responded to this demand by extending the bandwidth of the original SDI standards by a factor of 5 ½ times, and quickly a whole new standard for high-definition television transport emerged (HD-SDI), followed by a new generation of digital audio and video ASSPs.

The number of audio/video entertainment choices demanded by consumers was growing explosively, and the complexity of the transport networks had to keep up. Engineers responded by compressing digital video and audio information. The Digital Video Broadcast-Asynchronous Serial Interface standard (DVB-ASI) was developed to carry the newly compressed data over coaxial cables augmented with another new generation of ASSPs.

Modern computer networking technology has provided to the television industry an even lower-cost medium for transferring digital data: Ethernet. It is slowly creeping into broadcast studios and replacing the aging coaxial cables. The emerging standards for transmitting TV audio and video over an Ethernet fabric have been lumped together into the term Video over IP, meaning video and audio information transferred over an Ethernet fabric using the Internet Protocol (IP).

The ASSPs commonly used to implement digital video and audio transport at the hardware level have always been an expensive solution, and the industry needed another world-shaping technology to reduce the costs associated with digital transport. The Field-Programmable Gate Array (FPGA), invented by Xilinx in 1984, meets these needs perfectly. FPGAs are hardware “blank slates,” allowing engineers to innovate and try out new features inexpensively within the bounds of existing communication standards.
Chapter 1: Introduction

The Xilinx FPGA designers have embedded key circuits inside their devices to help audio and video engineers with their creations. For example, the integration into Xilinx FPGAs of RocketIO™ multi-gigabit serial transceivers and advanced SelectIO™ features has enabled the support of many more networking standards than previously possible. This now includes HD-SDI (SMPTE292M), SDI (SMPTE259M), and DVB-ASI, as well as others.

In addition, Xilinx application engineers have supported many audio and video standards with Verilog and VHDL reference designs that can be easily implemented in Xilinx FPGAs. These designs are available to customers at no cost.

This guide describes how to use Xilinx FPGA devices to implement various serial digital video interfaces commonly used in the professional video broadcast industry. The serial video interfaces described in this document are:

- **SD-SDI** (SMPTE-259M): Used to transport uncompressed standard-definition digital video.
- **HD-SDI** (SMPTE-292M): Used to transport uncompressed high-definition digital video.
- **DVB-ASI**: Used to transport compressed digital video.
- **AES**: Used to transport digital audio.

These popular standards offer high-bandwidth transmission between various video applications in the headend, edit suite, or studio, and can represent a significant portion of the bill of materials, particularly when there is a need for multiple channels. Integrating the network protocols into a Xilinx FPGA reduces the overall system cost considerably. Basing a system on Xilinx FPGAs gives the user the flexibility to meet exacting requirements, supporting standard interfaces, but still enabling differentiation from competitive offerings.

For each of these standards, this document describes how to implement the various functions needed in the transmitter and receiver. Reference designs are available in both Verilog and VHDL for these functions. Complete demonstration examples that run on Xilinx demonstration boards are also described. The source code for these demonstration examples is also available in both Verilog and VHDL.

Some of the interfaces, such as SD-SDI and DVB-ASI, can be implemented in most recent Xilinx FPGAs, including the Virtex™-II, Virtex-II Pro, Virtex-4, Virtex-5, and Spartan™-3 families. The higher-speed serial interfaces of HD-SDI require the RocketIO multi-gigabit serial transceivers found in Virtex-II Pro and Virtex-5 devices. RocketIO transceivers can also support the slower SD-SDI and DVB-ASI standards.
Section I: SD-SDI

Audio/Video Connectivity Solutions for the Broadcast Industry
Chapter 2

SD-SDI Physical Layer Implementation

Summary

The Serial Digital Interface (SDI) standard describes how to transport standard-definition digital video serially over coax cable. Equipment based on the SDI standard is commonly used in broadcast studios and video production centers.

This chapter describes implementations of the SDI physical layer using Xilinx FPGAs. The main topics covered by this chapter are: cable equalization, clock and data recovery, jitter reduction, clock multiplication for the transmitter, and cable drivers.

Introduction

SDI is defined by the SMPTE 259M and the ITU-R BT.656 standards.[1,2] SDI is widely used in broadcast studios and video production centers to transport digital video serially over coax cable. SMPTE 259M defines four standard bit rates for SDI ranging from 143 Mb/s to 360 Mb/s with the 270 Mb/s bit rate being, by far, the most common. Another standard, SMPTE 344M, adds a 540 Mb/s bit rate for SDI [Ref 1]. However, use of the 540 Mb/s bit rate is not yet widespread.

Figure 2-1 shows the correlation between the various SD-SDI chapters in this volume and the elements of the SDI link.

![Figure 2-1: SDI Block Diagram and SD-SDI Section Chapters](image)

The other chapters cover SDI encoding, decoding, error detection, and ancillary data handling. This chapter focuses on the physical layer and the details of sending and receiving the encoded SDI bitstream. Five different aspects of the physical layer are covered in this chapter. The block diagram shown in Figure 2-2 shows where these topics
fit in a piece of SDI equipment designed to receive and then retransmit an SDI bitstream (SDI pass-through).

- **Cable Equalization**: SDI receivers use adaptive cable length equalization to compensate for signal loss in the coax cable. An external cable equalizer can be used to equalize the bitstream before it is received by the FPGA. This chapter discusses techniques for interfacing cable equalizers to Xilinx FPGAs.

- **Clock & Data Recovery (CDR)**: After equalization, the SDI receiver must recover the data from the bitstream. Asynchronous data recovery is usually done by oversampling the bitstream and then looking for transitions. The data recovery unit tries to sample each bit in the middle of the bit period and as far away from the bit transitions as possible. Typically, a Phase-Locked Loop (PLL) is also used with the data recovery unit to recover the clock. However, in some cases, it can be assumed that the receiver and transmitter are running at the same frequency. If this is the case, then the receiver might not need to recover the clock. Instead, data-only recovery techniques can be used. This chapter discusses several clock and data recovery and data-only recovery techniques suitable for receiving SDI bitstreams with Xilinx FPGAs.

- **Jitter Reduction**: Parallel digital video supplied to an SDI transmitter either from an external video source or from an SDI receiver can contain a significant amount of jitter. An SDI transmitter is required to send the SDI bitstream with very little jitter. This can require the transmitter to reduce the amount of jitter on the video stream before transmission.

- **Clock Multiplication**: The SDI transmitter needs a bit-rate clock for its serializer. This usually requires the transmitter to multiply the incoming word-rate video clock by 10 to obtain the bit-rate clock. This multiplication must be done without adding too much jitter.

- **Cable Driver**: This chapter describes a method of interfacing Xilinx FPGAs to video coax cables.

![SDI Pass-Through Block Diagram](x247_02_041305)

**Figure 2-2**: **SDI Pass-Through Block Diagram**
The reference designs associated with this chapter have been tested on the Xilinx SDV demo board. This demo board is available from Cook Technologies (part number CTXIL103). Information is available at www.cook-tech.com.

This chapter is focused on implementations of SDI interfaces built using the fabric of Xilinx FPGAs. However, it is also possible to implement SDI receivers and transmitters using the Rocket IO transceivers in Virtex™-II Pro devices. Chapter 12 and Chapter 13 describe how to implement SDI using Virtex-II Pro RocketIO™ transceivers.

### Measuring SDI Transmitter and Receiver Performance

Before discussing how to implement SDI interfaces using Xilinx FPGAs, it is useful to have an understanding of the metrics used by the video industry to evaluate the performance of SDI transmitters and receivers. A very informative discussion of measuring SDI interface performance can be found in EBU Technical Document 3283 1996 [Ref 3].

The quality of an SDI transmitter is measured in two primary ways: the electrical performance of the output driver and the transmitter’s output jitter. The transmitter’s output driver must meet all of the electrical specifications defined in the SDI standard. These requirements are listed in the “SDI Cable Driver” section of this chapter. The transmitter must also produce an output bitstream that has less than 0.2 UI of output jitter. Lower transmitter output jitter results in more jitter margin in the SDI link.

SDI receiver performance is usually measured in three areas:

- Tolerance of waveform attenuation and distortion caused by long runs of coax cable
- Tolerance of jitter present on the input bitstream
- Tolerance of the SDI pathological waveforms.

The use of an SDI adaptive cable length equalizer takes care of the signal attenuation and phase distortion introduced by the coax cable.

Input jitter tolerance is the ability of the clock and data recovery unit to correctly receive an SDI bitstream with a significant amount of jitter distortion. The SMPTE 259M specification does not contain any requirement for how much input jitter must be tolerated by an SDI receiver. However, the standard does allow an SDI transmitter to have as much as 0.2 UI peak-to-peak of output jitter. Additional jitter is added by various sources such as reflections caused by impedance mismatches on the PCB board, connectors, and cable. An SDI receiver should be able to tolerate some amount of input jitter above the 0.2 UI allowed at the transmitter output. A good SDI receiver is usually capable of tolerating at least 0.5 UI of input jitter.

The pathological waveforms are defined by the SMPTE recommend practice RP 178-1996 [Ref 1]. These are two different worst-case waveforms that can be created by an SDI encoder and must be tolerated by SDI receivers.

One pathological waveform is poorly DC balanced and stresses the cable equalizer. This pattern is the top waveform in Figure 2-3. The waveform has one High bit followed by 19 UI stands for Unit Interval and is a term commonly used when discussing high-speed serial interfaces. One UI is equal to the time required to transmit one bit across the serial interface. At 270 Mb/s, 1.0UI is approximately 3.7 ns and 0.2 UI is 0.2 * 3.7 ns ~ 740 ps. The UI value is relative to the bit rate. At 360 Mb/s, 0.2 UI is about 556 ps.

2. The input jitter tolerance of an SDI receiver usually varies with the frequency of the jitter. An SDI receiver can usually tolerate more low-frequency jitter than high-frequency jitter. Typically, an SDI receiver can tolerate several UI of jitter when the sinusoidal jitter frequency is below 1 kHz. The jitter tolerance drops to something less than 1.0 UI as the jitter frequency increases above 10 kHz.
Low bits. This pattern can be repeated for the entire active portion of one video line. The opposite polarity is equally possible (one Low bit followed by 19 High bits). A properly designed cable equalizer must be able to deal with this waveform.

The second pathological waveform is a square wave with a period 40 bits long as shown in Figure 2-3. This square wave can repeat for the entire active portion of one line of video. Because of its relatively low density of transitions, this waveform can cause problems for the PLL in the CDR section of the SDI receiver. A properly designed CDR unit must be able to stay locked to the bitstream in the presence of this waveform.

Another type of jitter measurement applies when a piece of equipment receives an SDI bitstream and then retransmits this bitstream. The “jitter transfer function” of such a piece of equipment measures how much jitter present on the input SDI bitstream is transferred through the equipment to the output SDI bitstream for each different frequency component in the jitter spectrum. The jitter transfer function is very dependent upon the ability of the system to reduce jitter present on the input bitstream before re-transmission. Jitter reduction techniques, such as those described in the “Jitter Reduction” section of this chapter, can be used to modify the jitter transfer function of the system.

SDI Cable Equalization

The SDI standard allows digital video to be transmitted over long runs of video coax cable, up to 300 meters. Even with high-quality cable, the signal is significantly attenuated and distorted after passing through 300 meters of cable. The attenuation is frequency dependent, with the higher frequency components attenuated more than the lower frequency components of the signal. The coax cable also introduces frequency-dependent phase shifting, with the higher-frequency components phase-shifted more than the lower-frequency components.

The SDI standard requires that SDI receivers must be capable of working with input signals that have been attenuated by as much as 30 dB at one half the clock rate of the bitstream. To compensate for this amount of signal loss in the cable and to compensate for the frequency dependent phase shift introduced by the cable, SDI receivers usually employ
an adaptive cable length equalizer circuit. An adaptive cable length equalizer actively monitors the incoming signal and compensates for signal attenuation and phase shift caused by any cable length up to the maximum allowed.

SDI compliant adaptive cable length equalizers are not currently available in Xilinx FPGAs. It is recommended that an external SDI equalization circuit be used between the cable connector and the FPGA.

Figure 2-4 shows a National Semiconductor CLC014 SDI equalizer interfaced to a Xilinx Virtex-II Pro FPGA. The CLC014, like most SDI equalization chips, has differential PECL outputs. The peak-to-peak differential signal swing is compatible with some of the LVDS input standards supported by many Xilinx FPGAs. However, the common mode voltage is above the maximum allowed by low voltage Xilinx FPGAs, such as Virtex-II and Virtex-II Pro. In Figure 2-4, AC coupling is used to remove the DC offset from the signal. The parallel termination resistor networks on each leg of the LVDS input to the FPGA not only terminate the signal, but also bias the FPGA inputs to a common mode voltage that is compatible with the LVDS input buffers. The Virtex-II and Virtex-II Pro FPGAs have termination resistors for the LVDS input buffer built into the IOBs. These can be used instead of external resistor networks to terminate and bias the AC coupled signals from the equalizer.

The CLC014 generates a carrier detect (CD) signal. CD is asserted when a received signal is detected. In the example in Figure 2-4, the CD output of the CLC014 is connected to an FPGA input. A voltage divider is used to make the CD voltage level compatible with the FPGA input.

![Figure 2-4: Cable Equalization Example](image)

The Xilinx SDV demo board uses the CLC014 cable equalizer as shown in Figure 2-4. This design has been tested with cable lengths in excess of 300 meters.\(^1\)

---

1. Cable length testing of the SDI receiver was conducted on the Xilinx SDV demo board using a National Semiconductor CLC014 cable equalizer combined with the XAPP250-based SDI receiver checking for EDH errors.
Jitter Reduction

Prior to sending video data using an SDI transmitter, it might be necessary to reduce the amount of jitter present on the video stream. Digital video is usually provided to the SDI transmitter as 8-bit or 10-bit parallel video data words with an accompanying video clock running at the video word rate. Depending on the source of this video, the video clock can have a large amount of jitter. The SMPTE and ITU parallel video standards allow a large amount of jitter on the parallel video clock. For example, SMPTE 125M, a standard that defines an SDI compatible parallel video format for 4:2:2 component digital video, allows for as much as ±3 ns peak-to-peak clock jitter [Ref 1]. In contrast, the SDI standard requires that the bitstream generated by the SDI transmitter have no more than 0.2 UI peak-to-peak jitter (about 740 ps at 270 Mb/s). Simply multiplying the parallel video clock by 10 to obtain a bit-rate clock without also implementing jitter reduction would result in a bit-rate clock with as much as 0.8 UI of jitter, four times more than the entire output jitter budget of the SDI transmitter.

There are many sources of jitter in the digital video stream. Analog video can be passed through a video decoder to convert it to digital video. The video decoder might generate the digital video clock using a PLL that locks to the horizontal sync pulses present in the analog video. These circuits can produce a video clock with significant jitter.

Jitter is also present on the video stream recovered by an SDI receiver. SDI receivers usually are able to correctly receive SDI bitstreams that have 0.5 UI of jitter or more. The recovered clock generated by the receiver’s CDR circuit usually contains much of the jitter present on the incoming bitstream. So, in most cases, it is not possible to use the recovered clock from an SDI receiver directly as the clock for an SDI transmitter.

In all of these cases, the SDI transmitter should implement jitter reduction on the video before transmission. The classic jitter reduction technique for digital video is shown in Figure 2-5. The video clock is passed through a PLL designed to reduce jitter. The input video clock is also used to write the video data into an asynchronous FIFO. The data is read out of the FIFO using the low-jitter clock from the PLL. The jitter has been removed from the clock and the data has been resynchronized to the low-jitter clock. This technique is known as “reclocking”.

![Figure 2-5: Jitter Reduction](attachment:image.png)
The SDI transmitter must generate a bit-rate clock by multiplying the word-rate video clock by ten. Often, the PLL used to multiply the clock can also be designed to implement jitter reduction. In Figure 2-5, the PLL multiplies the high-jitter clock by 10 and also reduces the jitter on the clock. The bit-rate clock is used by the transmitter to serialize the data for transmission. The asynchronous FIFO and the input stages of the SDI transmitter must run at the word rate. The low-jitter bit-rate clock produced by the PLL can be divided by ten to produce a low-jitter word-rate clock. Or, as shown in Figure 2-5, a counter can generate a clock enable for one clock cycle out of every ten cycles for the FIFO and SDI transmitter.

The Xilinx Coregen tool can produce asynchronous FIFOs well suited for this jitter reduction application. The FIFO produced by Coregen can be implemented in either distributed RAM for shallow FIFOs or in block RAM when deeper FIFOs are required. In a jitter reduction application, the FIFO usually is not very deep. An asynchronous FIFO that is 10 bits wide and 15 locations or less deep is usually sufficient and uses very few FPGA resources when implemented in distributed RAM in Xilinx FPGAs.

A PLL with a narrow input bandwidth is required to implement clock jitter reduction. A DCM or DLL in Xilinx FPGAs cannot be used for jitter reduction. These digital circuits effectively pass any jitter present on the input clock directly through to the output clock.

It is possible to build a PLL in a Xilinx FPGA using an external VCO or VCXO as described in the “Internal CDR” section of this chapter. A PLL built using this technique can act as both a clock multiplier and a jitter reducer to meet the needs of the SDI transmitter.

Clock Multiplication

Every SDI transmitter needs to have both a word-rate clock and a bit-rate clock. The bit-rate clock runs at 10X the word-rate and is used to clock the serializer. It is important that the clock multiplier be capable of producing the bit-rate clock with low jitter. Jitter on the bit-rate clock becomes jitter on the SDI output bitstream.

As mentioned in the previous section, the jitter reduction PLL can often also be used to implement the clock multiplier for the bit-rate clock.

It is also possible to use the clock synthesis capability of the DCM in Virtex-II Pro to create the bit-rate clock for the SDI transmitter. The Virtex-II Pro DCM synthesizes an output clock that can be as fast as 360 MHz in the slowest speed grade Virtex-II Pro devices. The DCM in Virtex-II devices, however, should not be used for generating the SDI transmitter bit-rate clock. This older DCM design produces too much jitter in frequency synthesis mode.

When using the Virtex-II Pro DCM in this application, the 27 MHz clock cannot simply be multiplied by 10 to produce a 270 MHz bit-rate clock. For all current speed grades of Virtex-II Pro devices, the DCM must be in high-frequency mode to output a 270 MHz clock on the CLKFX output. However, in high-frequency mode, the input clock must be at least 50 MHz when using the CLKFX output.

There are two solutions to this problem. First, if a 2X version of the word-rate clock (54 MHz) is available to the FPGA, this can be used as the CLKIN to the DCM with the DCM in high-frequency mode. The DCM can multiply this clock by 5 to produce a 270 MHz clock. It is not viable to cascade two DCMs, one multiplying the input clock by 2 and the second by 5. Cascading DCMs produce too much jitter. Using a 54 MHz reference clock and using the DCM as a 5X multiplier, the typical output jitter was measured on the SDI transmitter of the Xilinx SDV demo board of less than 0.1 UI with a 270 Mb/s SDI bitstream.(1)
If a 2X word-rate clock is not available, then the DDR output capabilities of the Virtex-II Pro FPGA can be used as shown in Figure 2-6. In this example, the DCM is in low frequency mode and the 27 MHz clock is connected to the CLKIN of the DCM. The DCM multiplies CLKIN by 5X to produce a 135 MHz clock (one-half the bit-rate). The SDI transmitter’s serializer is clocked by the 135 MHz clock and shifts two bits every clock cycle. The two output bits from the serializer connect to the data inputs of a FDDRRSE DDR output primitive. The DDR primitive uses two phases of the bit-rate clock to multiplex the two bits together to produce a 270 Mb/s bitstream.

The Load Enable Logic block in Figure 2-6 generates a load enable to the serializer. The load enable signal is asserted for one clock cycle out of every five cycles of the 135 MHz clock. The load enable to the serializer is asserted during the clock cycle immediately after the rising edge of the 27 MHz clock.

This DDR technique also works for 360 MHz bit rates by multiplying the 36 MHz word-rate clock by 5 to produce a 180 MHz clock. It can also be used for 540 MHz bit rates, but the DCM must be in high-frequency mode for the CLKFX output to run at 270 MHz.

Using this DDR technique on the Xilinx SDV Demo Board, we have measured typical SDI transmitter jitter numbers of less than 0.1 UI for both timing and alignment jitter at 270 Mb/s. Keep in mind, however, that the DCM does not do jitter reduction. The word-rate clock supplied to the CLKIN input of the DCM must have little jitter in order to produce a bit-rate clock with low jitter and, subsequently, a low-jitter SDI output bitstream.

In Figure 2-6, the Q0 out of the serializer is the LSB and must be sent by the transmitter before Q1. This requires connecting Q0 to the D1 input of the FDDRRSE block and Q1 to the D0 input as shown.

---

1. All SDI transmitter output jitter measurements were conducted using the Xilinx SDV demo board. The FPGA was interfaced to a 75ΩBNC connector through a National Semiconductor CLC001 cable driver. The reference clock into the DCM clock multiplier was a crystal oscillator with less than 40ps pk-pk jitter. The jitter measurements were made with a Tektronix WFM700A analyzer connected to the output of the SDV demo board with 1 meter of RG179B/U coax cable.
SDI Cable Driver

The SMPTE 259M SDI standard requires that the signal produced by an SDI transmitter meet the following electrical specifications:

- Unbalanced (single-ended) output with a source impedance of 75\(\Omega\) and a return loss\(^{(1)}\) of at least 15 dB over a frequency range of 5 MHz to the clock frequency of the signal being transmitted.
- A peak-to-peak amplitude of 800 mV ±10%.
- DC offset of the mid-amplitude point of the signal of 0.0V ±0.5V. (This means that the transmitter is AC-coupled to the cable.)
- Rise and fall times (between 20% and 80% of the signal amplitude): 0.4 ns min and 1.5 ns max. Rise and fall times must not differ by more than 0.5 ns.
- Overshoot of the rising and falling edges shall not exceed 10% of the amplitude.

While it might be possible to AC couple the output of a Xilinx FPGA to the SDI coax cable using one of the existing Xilinx FPGA supported I/O standards, meeting all of the SDI electrical requirements is not a simple matter. It is much easier to use an external cable driver chip specifically designed to meet the electrical specifications of the SDI standard. Several vendors make SDI cable drivers.

With signaling rates as high as 360 MHz (or 540 MHz with SMPTE 344M), it is highly recommended that an LVDS output pair be used to drive the SDI bitstream out of the Xilinx FPGA. Using an LVDS output buffer produces the lowest jitter signal out of the FPGA. The National Semiconductor CLC001 is an SDI-compliant cable driver that has LVDS inputs. It is easy to interface the CLC001 to any Xilinx FPGA that supports the LVDS I/O standard. Figure 2-7 shows an example of using the National CLC001 with a Xilinx FPGA [Ref 4].

![Figure 2-7: SDI Cable Driver Example](image)

Note that the AC coupling capacitor used on the output of the CLC001 must be at least 1 \(\mu\)F in order to successfully pass the SDI pathological waveforms. The pathological waveforms are discussed in the “Clock and Data Recovery” section.

---

1. Return loss is a measurement of the amount of signal absorbed when a wave reflected by the receiver arrives back at the transmitter output. Higher return loss numbers are better since less of the reflected wave is reflected back to the receiver. Excessive reflections on the cable can cause waveform distortion (jitter), interfering with the SDI receiver’s ability to correctly receive the signal.
Chapter 2: SD-SDI Physical Layer Implementation

By using an SDI compliant cable driver and following the layout and circuit recommendations from the cable driver manufacturer, an SDI transmitter fully compliant with the SDI transmitter electrical specifications can easily be implemented.

Clock and Data Recovery

The SDI bitstream is encoded using a pseudo-random scrambler followed by a NRZ-to-NRZI conversion. Figure 2-8 shows how the SDI encoding and decoding algorithms are defined.

![SDI Encoder Diagram](image1)

![SDI Decoder Diagram](image2)

**Figure 2-8: SDI Encoding and Decoding**

The usual way to recover the data from SDI bitstream is to use a PLL-based CDR circuit. The PLL locks to the clock frequency of the incoming bitstream and provides a recovered bit-rate clock running at the clock rate of the bitstream. The data is recovered from the bitstream and is synchronized with the recovered clock.

After the data has been recovered, it is decoded using the inverse of the encoding algorithm. Finally, a “framer” must determine where the 10-bit word boundaries occur in the decoded bitstream. SDI decoder and framer circuits are described in Chapter 4, “SD-SDI Video Decoder.”

Two different CDR techniques for Xilinx FPGA-based SDI receivers are described in the following sections. The “External CDR” section uses an external CDR chip designed specifically for SDI. The “Internal CDR” section is based on Xilinx Application Note XAPP250 [Ref 5] and uses an inexpensive external VCO along with logic in the fabric of a Xilinx FPGA to implement a PLL-based CDR circuit.

In some cases, it might not be necessary to do clock recovery in the SDI receiver. A data recovery technique for SDI is described in the “Data Recovery Only” section.

External CDR

Several companies produce CDR devices designed specifically for SDI. The National Semiconductor CLC016 is an example of such a device. It supports the four standard bit rates of SMPTE 259M: 143 Mb/s, 177 Mb/s, 270 Mb/s, and 360 Mb/s. Devices such as the CLC016 are often called “reclockers,” because they generate a recovered clock and a copy of the original bitstream that has been “re clocked” or synchronized to the recovered clock.
The differential outputs of the CLC016 are compatible with PECL signaling. The common mode voltage of the CLC016’s differential outputs is too high to be used directly by low-voltage Xilinx FPGAs. Level shifting is required to reduce the common mode voltage of these outputs to a level compatible with the FPGA. This can be done by AC coupling the outputs of the CLC016 to the FPGA as shown in Figure 2-9.

![Figure 2-9: Interfacing an External SDI Reclocker to Xilinx FPGAs](image)

In Figure 2-9, a Xilinx Virtex-II Pro FPGA is interfaced to the CLC016. The two differential input pairs into the FPGA use the LVDSEXT_25_DCI standard. This standard supports differential input voltage swings up to 1V, sufficient to handle the 800 mV differential voltage swings of the CLC016. AC coupling is used to remove the DC offset on the outputs of the CLC016. The DCI input termination for the LVDS inputs of the FPGA biases each leg of the inputs to half the VCCO voltage, providing a suitable common mode voltage level on the LVDS inputs. External termination resistors can be used instead of the internal DCI termination, if desired. The AC coupling capacitors must be at least 1 μF in order to pass the pathological waveforms.

The recovered clock from the CLC016 is connected to a BUFG and distributed globally in the FPGA. The serial data comes into the LVDS buffer and passes through the IOBDELAY element in the IOB before being clocked into the input flip-flop (IFD) in the IOB. The IOBDELAY element matches the clock delay through the BUFG and global clock distribution network so that the setup and hold requirements of the IFD flip-flop are satisfied.

The input jitter tolerance of a solution using an external SDI CDR chip is dictated primarily by the specifications of the CDR chip.

**Internal CDR**

An external SDI CDR chip usually supports multiple bit rates. However, in many cases, an SDI receiver only needs to support a single bit rate, usually 270 Mb/s. This rate supports 4:2:2 component digital video for both NTSC and PAL. An SDI receiver or transmitter that only supports a single-bit rate is permitted by the SDI standard. Many pieces of video equipment with SDI interfaces are designed to support only the 270 Mb/s rate.
Chapter 2: SD-SDI Physical Layer Implementation

The main advantage of designing an SDI receiver supporting a single bit rate is cost. It is usually less expensive to implement a CDR circuit that only runs at one bit rate instead of multiple rates.

XAPP250 describes a technique that can be used to implement a 270 Mb/s CDR circuit using the fabric of a Xilinx FPGA with the addition of a few inexpensive external components [Ref 5]. The external components are a voltage controlled oscillator (VCO) and a loop filter to generate the control voltage into the VCO. Using these external components, a complete CDR circuit can be implemented in the fabric of the FPGA.

Figure 2-10 shows a block diagram of an SDI receiver using XAPP250 for the CDR function. Blocks in the shaded portion of the drawing are internal to the FPGA.

Initially, the phase detector controls the frequency of the VCO (divided by the M divider) to match the frequency of the reference oscillator divided by the N divider. After locking to the reference clock, the phase detector attempts to match the VCO to the frequency of the video bitstream coming from the cable equalizer. Using a reference clock to “spin-up” the PLL and to lock it when the input bitstream is missing provides several benefits. First, it allows the PLL to lock to the input bitstream faster. Second, it keeps the PLL running at 270 MHz even when the bitstream is missing, because the PLL reverts to locking to the reference clock when the bitstream stops.

The clock generated by the VCO is a bit-rate clock and is distributed by the global clock distribution network of the FPGA. A 10-bit shift register, clocked by the bit-rate clock, deserializes the data recovered by the phase detector, generating a 10-bit data word.

The VCO clock is divided by 10 to create a word-rate clock. The clock divider also generates a clock enable to the sync register. This register loads the 10-bit parallel data word from the shift register once every ten cycles of the VCO clock. The output of the sync register is fed into the SDI decoding and framing unit, clocked by the word-rate clock. The sync register is loaded at approximately the same time as the falling edge of the word-rate clock. This provides about half the word-rate clock period for data set up into SDI decoder. The other half of the word-rate clock period provides hold time on the data.

![Figure 2-10: Internal CDR for SDI Receiver](xilinx_fpga_cdr_diagram.png)
The CDR circuit described in XAPP250 uses a passive loop filter to generate the control voltage for the VCO. The loop filter converts two outputs of the FPGA into an analog control voltage to control the frequency of the VCO. In Figure 2-10, an active loop filter replaces the passive loop filter described in XAPP250. Active loop filters are generally superior to passive loop filters.

Figure 2-11 shows the active loop filter and VCO used on the Xilinx SDV demo board to implement the 270 MHz VCO for the SDI CDR circuit. The loop filter acts as an integrator, increasing the output control voltage when FASTER pulses are received from the phase detector in the FPGA and decreasing the control voltage when SLOWER pulses are received. The FASTER and SLOWER inputs to the loop filter are driven by 3.3V outputs from the FPGA.

Figure 2-11: 270 MHz VCO for SDI CDR

The active loop filter in Figure 2-11 has been optimized to provide good input jitter tolerance characteristics. Figure 2-12 shows the input jitter tolerance of an SDI receiver using the VCO and loop filter shown in Figure 2-11. The horizontal axis of the graph is the frequency of the jitter that has been added to the bitstream. The vertical axis is the...
amplitude of the jitter in UI. To make these measurements, the SDI bitstream is modulated with sinusoidal waveform of a certain frequency. This adds sinusoidal jitter to the bitstream at that particular frequency. The amplitude of the jitter is increased until the SDI receiver fails to accurately receive the signal. This point is recorded on the chart and indicates the input jitter tolerance of the receiver to jitter of that particular frequency. By measuring the input jitter tolerance of the SDI receiver at various jitter frequencies, a graph such as Figure 2-12 is generated.

![SDI Input Jitter Tolerance Graph]

**Figure 2-12: XAPP250 Input Jitter Tolerance**

Figure 2-12 shows that for jitter frequencies below 1 kHz, the CDR circuit can tolerate more than 15 UI of jitter. The jitter tolerance rolls off above 1 kHz then stabilizes at about 0.65 UI of input jitter tolerance for jitter frequencies above 100 kHz.

Data Recovery Only

In some cases, it might not be necessary to implement clock recovery for an SDI receiver. If the receiver and transmitter always run at the same frequency, then clock recovery is not necessary and data recovery only techniques can be used. Even if the transmitter and receiver vary slightly in frequency, various synchronization techniques can be used to make up for these minor frequency variations on a periodic basis.

Data recovery only could be used, for example, for an SDI link between a receiver and transmitter located in the same piece of equipment. Perhaps the receiver and transmitter

1. The input jitter tolerance graph shown for the XAPP250 style CDR was measured using the Xilinx SDV demo board. The external loop filter and VCO are implemented on this board as shown in Figure 2-11. The measurements were taken using a PRBS pattern using the equipment described in “Appendix A: Test Equipment.”.
are on different circuit boards in the same chassis. In this case, the transmitter and receiver would probably have access to the same video clock and data recovery techniques would work very well.

Another example occurs in broadcast studios where a studio time base signal, usually called “house sync,” is distributed to all the video equipment in the studio. Each piece of equipment uses this time base signal to generate local video timing signals including the video clock. In this scheme, a transmitter and receiver in different locations in the same broadcast studio have video clocks that are derived from the common time base and, over time, run at exactly the same frequency.

Xilinx has several application notes that describe different techniques for implementing data recovery units in Xilinx FPGAs. These techniques could be used in SDI receiver applications where clock recovery is not required.

Xilinx Application Note XAPP224 describes a technique for implementing data recovery in Virtex-II or Virtex-II Pro FPGAs [Ref 6]. This technique supports bit rates up to about 420 Mb/s. XAPP224 implements an oversampling technique and does not require that the input bitstream have any particular phase relationship to the local reference clock.

The primary disadvantage of XAPP224 is that it requires two DCMs to create different phases of the reference clock for oversampling the bitstream. However, the clocks generated by the DCMs can support multiple SDI receivers in the same FPGA. Asynchronous data recovery techniques like the one described in Xilinx Application Note XAPP671 can be used to implement data recovery for SDI without requiring any DCMs [Ref 7].

Both of these data recovery techniques allow for minor variations in the clock between the transmitter and the receiver. If it is known that, on average, the transmit and receive clocks are exactly the same frequency, a shallow FIFO is sufficient to smooth out the minor variations in the rate at which data is recovered. The data recovery unit normally produces 10 bits of data every word-rate clock cycle. However, occasionally the data recovery unit recovers either 9 or 11 bits in one word-rate clock cycle. By buffering the recovered data bits with a shallow FIFO, these minor variations are averaged out.

If latency in the SDI receiver is not an issue for a particular application, then it is possible to use a data recovery technique combined with a VCO to regenerate the video clock, in effect implementing clock and data recovery without a PLL. This technique is shown in Figure 2-13.

The data from the data recovery unit is written to an asynchronous FIFO. The block RAMs in Xilinx FPGAs are well suited for implementing asynchronous FIFOs. Initially, a controller allows the FIFO to become half full before any data words are read from the FIFO. After starting to read the FIFO, the controller monitors the data level in the FIFO and attempts to keep the FIFO half full by speeding up or slowing down the VCO that is generating the word-rate clock used to read the FIFO.
Chapter 2: SD-SDI Physical Layer Implementation

Reference Designs

Three reference designs are provided with this chapter: two SDI receivers and one SDI transmitter. One SDI receiver is based on XAPP250 [Ref 5]. The other receiver is intended for use with an external SDI reclocker.

Figure 2-14 is a block diagram of the XAPP250-based SDI receiver reference design. The CDR unit uses a clock recovery PLL built in the fabric of the FPGA. An external VCO and loop filter must be provided. Details of the VCO and loop filter are shown in Figure 2-11.

Figure 2-14:  XAPP250-Based SDI Receiver Reference Design

The second SDI receiver reference design uses an external reclocker for CDR. See Figure 2-15. The reclocker provides a recovered clock on a differential signal pair, and it provides the serial bitstream resynchronized to the recovered clock on another differential signal pair.

In the FPGA, the recovered clock is distributed through the global clock distribution network. The serial data is delayed by an IOBDELAY element. The IOBDELAY matches the...
delay added to the clock by the BUFG and clock distribution network. After the IOBDELAY, the serial data is clocked into the input flip-flip in the IOB. The serial bitstream from the flip-flop is deserialized and then decoded and framed. The recovered clock is divided by 10 by a ring counter to provide a word-rate recovered clock.

The SDI transmitter reference design uses a serializer implemented in the fabric of the FPGA and is identical to the SDI transmitter design shown in Figure 2-6. A DCM takes the 27 MHz word-rate clock and produces two phases of a 135 MHz clock that are 180 degrees out of phase from each other. A DDR primitive on the output of the serializer creates the 270 MHz SDI bitstream.

Table 2-1 shows the logic resources required to implement the reference designs. The SMPTE descrambler module and the framer module were taken directly from the reference designs discussed in Chapter 4, “SD-SDI Video Decoder.” In addition to the logic shown, both designs required one MULT18X18 primitive used by the barrel shifter in the par_framer_mult module. A framer design without a multiplier is also discussed in Chapter 4 and is slightly larger (see par_framer results).

The reference designs were targeted at Virtex-II Pro XC2VP4-5 and Virtex-II XC2V250-4 devices. In all cases, the synthesis was done using XST with the ISE 5.2i tools. Both devices met all timing constraints for operation at SDI bit rates up to 360 Mb/s.

The reference designs presented here are basic SDI transmitters and receivers and do not include more advanced features, such as EDH insertion and error detection, video standard detection, and flywheel video decoder. These advanced features can be added to the basic designs. The implementation sizes of these features can be found in Chapter 5, “SD-SDI Video Flywheel,” and Chapter 6, “SD-SDI Ancillary Data and EDH Processors.”

Table 2-1: Reference Design Implementation Results

<table>
<thead>
<tr>
<th>Logic Resource</th>
<th>Size LUTs</th>
<th>Size FFs</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDI Rx with external CDR (par_framer)</td>
<td>132</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>SDI Rx with external CDR (par_framer_mult)</td>
<td>78</td>
<td>109</td>
<td>1 MULT18X18</td>
</tr>
<tr>
<td>SDI Rx with XAPP250 CDR (par_framer)</td>
<td>352</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>SDI Rx with XAPP250 CDR (par_framer_mult)</td>
<td>270</td>
<td>262</td>
<td>1 MULT18X18</td>
</tr>
<tr>
<td>SDI Tx</td>
<td>31</td>
<td>43</td>
<td>1 DCM</td>
</tr>
</tbody>
</table>
Chapter 2: SD-SDI Physical Layer Implementation

Conclusions

The SDI chapters in this volume describe how to implement a complete SDI interface using Xilinx FPGAs. This chapter focused on the details of implementing the physical layer of the SDI interface.

Using Xilinx FPGAs to implement SDI interfaces in video equipment can provide many benefits. Xilinx FPGAs are large enough to allow multiple SDI interfaces to be implemented in the same FPGA device, reducing the number of devices required. Even in video equipment with only a single SDI interface, the FPGA can provide higher levels of integration by providing additional processing functions for the video prior to transmission over the SDI interface or after reception.

Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_sd-phy.zip.

Appendix A: Test Equipment

Test Equipment Used for Receiver Input Jitter Tolerance Measurements

All input jitter tolerance tests were made using the SDI receiver on the Xilinx SDV demo board (CTXIL103 from Cook Technologies).

Various test equipment was required to generate and measure the input jitter tolerance, including:

HP71603B Bit-Error Rate Tester for generating and verifying high-speed serial data. Includes the following modules:

- HP70311A Clock Source
- HP70841B Pattern Generator
- HP70842B Error Detector

Two HP8648D Signal generators to provide an accurate, low-jitter reference clock:

- Agilent 71501C jitter analysis system to add periodic jitter to the serial stream for receiver jitter tolerance testing
- Agilent 86100A Digital Communications Analyzer for measuring jitter, signal amplitude, eye width, and other signal characteristics.

Test Equipment Used for Transmitter Output Jitter Measurements

All transmitter output jitter measurements were made using the SDI transmitter on the Xilinx SDV demo board.

The transmitter output jitter was measured using a Tektronix WFM700A Waveform Monitor.
Chapter 3

**SD-SDI Video Encoder**

**Summary**

The ANSI/SMPTE 259M-1997 standard specifies a serial digital interface (SDI) for digital video equipment operating at either the 525-line, 60 Hz video standard or the 625-line, 50 Hz video standard [Ref 1]. The SDI standard describes how to transport both composite and component digital video over standard video coax. SDI is widely accepted and often forms the video transportation "backbone" of television studios and broadcast centers.

Figure 3-1 is a block diagram showing correlation between the various chapters in this volume and the elements of the SDI link.

This chapter focuses on the SDI encoder. The reference design includes several implementations of the SDI encoder optimized for use with the Virtex™-II FPGA series and other Xilinx FPGA families. Both serial (bit-rate) and parallel (word-rate) implementations of the SDI encoder are presented. Also included are examples illustrating using a Xilinx FPGA as an alternative to several commercially available SDI encoder devices, the Gennum GS9002 and the Cypress CY7C9235.

A test bench and several diagnostic modules are included for testing the SDI encoder modules described in this chapter, and the SDI decoder modules described in Chapter 4, “SD-SDI Video Decoder.”

![Figure 3-1: SDI Block Diagram and SD-SDI Section Chapters](image-url)
SDI Introduction

Digital Video Formats

The SDI standard describes how to transport standard definition digital video serially over a video coax cable. This standard describes the encoding and decoding processes performed on the video bitstream for transportation across the physical layer. The standard also describes the electrical and mechanical characteristics of the physical layer. However, it does not define the actual format of the digital video data. Additional standards for the definitions of SDI compatible digital video formats are:

- ANSI/SMPTE 125M, ANSI/SMPTE 267M, and ITU-R BT.601-5 for 4 x 3 and 16 x 9 aspect ratio 4:2:2 component digital video [Ref 1] [Ref 2]
- ANSI/SMPTE 244M for composite NTSC digital video [Ref 1]
- IEC 1179 (now called IEC 61179) for composite PAL digital video [Ref 3]

The SDI standard does not cover high-definition digital video. Another standard, SMPTE 292M, defines a serial digital interface standard for high-definition digital video, commonly called HD-SDI. The bandwidth requirements for high-definition video are significantly higher than for standard definition video. Also, video components in the HD-SDI standard are interleaved differently than in the SDI standard. Because implementing an HD-SDI encoder involves higher bandwidth requirements and different formats than a standard definition SDI encoder, it is not covered in this chapter.

All digital video formats supported by the SDI standard use either 8 bits or 10 bits per data word. Although the SDI standard always sends 10-bit data across the link, with proper handling it can transport 8-bit digital video formats. When 8-bit video is used, the two least-significant bits of a 10-bit input to the SDI encoder can be tied High or Low.

Encoding and Decoding

Prior to sending digital video serially across the physical layer, an SDI transmitter must encode the video according to the SDI standard. By design, this encoding process ensures that the serial bitstream has sufficient level transitions to allow the receiver to recover the clock and data. After the receiver captures the serial data, the decoder must reverse the encoding process to recover the original video data.

The SDI standard uses two generator polynomials, normally expressed as linear feedback shift registers (LFSR), to implement two separate encoding stages. First, the video bitstream is scrambled using the generator polynomial:

\[ G_1(x) = x^9 + x^4 + 1 \]

The output of this first encoding stage is referred to as the scrambled non-return-to-zero (NRZ) bitstream.

The second encoding stage uses the generator polynomial:

\[ G_2(x) = x + 1 \]

It converts the scrambled NRZ bitstream to a polarity-free scrambled NRZ-inverted (NRZI) bitstream. NRZI is DC balanced for transmission across the physical layer. If the bitstream is inverted between the transmitter and the receiver, then the polarity-free nature of the SDI bitstream allows the decoder to properly recover the original data.
The SDI decoder reverses the encoding process by using the same generator polynomials in reverse order: G2 to convert from NRZI to NRZ and then G1 to descramble the bitstream.

Figure 3-2 illustrates the encoding and decoding processes when implemented in LFSRs. The circles with plus symbols inside are exclusive-OR gates. The boxes represent individual flip-flops. The LSB of a data word is sent first.

Framing and TRS Clipping

After decoding the video bitstream, the receiver must determine where individual 10-bit words begin and end in the serial bitstream. This process is called framing. In order to frame the bitstream, a unique and recognizable pattern must be sent periodically for the framer to use as a framing reference.

All of the digital video formats supported by SDI share similar definitions for the timing reference signal (TRS) symbols. TRS symbols delineate between the active and inactive portions of the video. For component video standards, two TRS symbols are sent per line of video: one at the start of the active video called SAV, and one at the end of active video called EAV. For composite video standards, one TRS symbol is sent per line. A TRS symbol is sent as four consecutive words, formatted as:

```
3ff 000 000 XYZ
```

The first three words of the TRS symbol, called the preamble, form a unique sequence in the bitstream. The fourth word, called XYZ, varies depending on the specific digital video format being transported.

Since the TRS preamble is common across all the supported digital video formats, is sent on a regular basis, and is unique in the bitstream, it is used as the framing reference. Upon detecting a sequence of ten consecutive ones and twenty consecutive zeros, a framer in the SDI receiver can determine the proper boundaries of all subsequent data words in the bitstream.

When transporting 8-bit digital video formats, the SDI transmitter must convert the 8-bit video words into the native SDI 10-bit format. This must be done properly in order to generate valid 10-bit TRS preambles. The transformation of 8-bit digital video into SDI compatible 10-bit digital video is called TRS clipping. The SDI standard requires forcing all data values between hex 3fc and 3ff to a value of 3ff prior to encoding. Likewise, values between 000 and 003 must be forced to a value of 000.
Chapter 3: SD-SDI Video Encoder

SDI Bit Rates

The bit rates supported by SDI range from 143 Mb/s to 360 Mb/s, depending on the digital video format being transported. The SDI standard defines four different bit rates as 'support levels' (shown in Table 3-1). SDI compliant equipment is not required to support all bit rates. A piece of equipment supporting bit rates up to 270 Mb/s is said to conform to ANSI/SMPTE 259M-ABC, since it supports levels A, B, and C.

Table 3-1: SDI Standard Bit Rates

<table>
<thead>
<tr>
<th>Support Level</th>
<th>Bit Rate</th>
<th>Video Format</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level A</td>
<td>143 Mb/s</td>
<td>NTSC composite</td>
<td>ANSI/SMPTE 244M-1995</td>
</tr>
<tr>
<td>Level B</td>
<td>177 Mb/s</td>
<td>PAL composite</td>
<td>IEC 61179</td>
</tr>
</tbody>
</table>

Error Detection

The SDI standard does not mandate the use of an error detection mechanism. Some of the digital video standards, SMPTE 125M for example, specify error detection bits in the XYZ word of the TRS to determine the validity of the TRS symbol. However, the SDI standard highly recommends embedding error detection check words into the SDI video stream as described in SMPTE RP 165-1994. Techniques for generating and inserting these check words are described in Chapter 6, “SD-SDI Ancillary Data and EDH Processors.”

Clock Jitter Considerations

Any SDI encoder that generates a serial SDI bit stream generally requires a clock running at the SDI bit rate for the parallel-to-serial converter. A bit-rate serial clock can be generated in the FPGA by multiplying the parallel data clock using a Virtex-II DCM. Alternatively, an external clock multiplier circuit can be used to provide a serial clock to the FPGA.

The SDI standard allows for a maximum peak-to-peak jitter of 0.2 times the serial clock period. If the SDI link is running at 360 Mb/s, then the maximum jitter allowed is about 550 ps.

Parallel digital video standards allow relatively large amounts of clock jitter on the parallel clock. For example, the SMPTE 125M standard allows up to 3 ns of peak-to-peak jitter on the parallel clock. This can make the parallel clock unsuitable for use as a reference to the clock multiplier.

The Virtex-II DCM does not filter out clock jitter on the reference clock. Any jitter on the parallel clock becomes jitter on the serial clock, with additional jitter added by the DCM and the clock distribution network. If a parallel clock from a SMPTE 125M video source is used as the reference clock, the resulting serial clock jitter could greatly exceed the SDI jitter specification.

If the designer cannot ensure that the parallel clock has sufficiently low jitter to make it suitable for use as a reference to the DCM, then an external clock regenerator or clock multiplier capable of reducing the parallel clock jitter must be used.
Jitter considerations for SDI implementations in Xilinx FPGAs is covered in more detail in Chapter 2, "SD-SDI Physical Layer Implementation."

Reference Design

The reference design includes several different SDI encoder implementations, a TRS clipper module, diagnostic modules, and a test bench.

TRS Clipper

TRS clipping is required to support 8-bit digital video in the 10-bit SDI protocol. The trs_clipper module is a simple combinatorial design. If the eight most significant bits of the video word are all zeros, the module forces the two least significant bits to zeros. If the eight most significant bits are all ones, the two least significant bits are forced to ones. Otherwise, the two least significant bits pass through the module unchanged. An enable input to the TRS clipper module is provided to disable the TRS clipping function if desired.

Figure 3-3 shows a block diagram of how the TRS clipper is used in an SDI transmitter. This block diagram uses a DCM to multiply the parallel clock by five. The serializer shifts out two bits every five clock cycles into DDR flip-flops. This was done instead of using the DCM to multiply the parallel clock by ten because the CLKFX output of the Virtex-II DCM currently can not run fast enough to generate a bit-rate clock for the highest SDI bit rates. By using a half bit-rate clock and the DDR hardware in the Virtex-II IOBs, a Virtex-II design can easily serialize data at the maximum SDI bit-rate.

Bit-Rate Serial Scrambler

The scrambling process involves "division" of the incoming bitstream by the generator polynomials. A simple LFSR implementation is shown in Figure 3-4. A serial implementation results in a very small amount of hardware. However, a serial implementation must run at the full bit-rate of the SDI interface, up to 360 MHz.
Chapter 3: SD-SDI Video Encoder

The HDL files `ser_scrambler.*` contain the bit-rate serial SDI scrambler using a LFSR. As shown in “Reference Design Results”, this implementation is very small. In a Virtex-II FPGA, the serial scrambler runs fast enough to support the highest bit rate specified by the SDI standard.

The serial scrambler module has two control inputs, `scram` and `nrzi`, to enable the scrambler and the NRZ-to-NRZI conversion, respectively. These control signals allow the two encoding stages to be bypassed if the data to be sent is non-SDI compliant. In normal SDI operation, both inputs should be High.

### Half Bit-Rate Serial Scrambler

The Virtex-II architecture features double data-rate (DDR) output flip-flops and a DDR MUX in the IOB. A scrambler module that processes two bits per clock cycle can be used to drive the DDR flip-flops in an IOB. This allows the clock to the scrambler to run at one-half the SDI bit rate rather than at the full bit rate as required for a serial SDI scrambler.

Figure 3-5 shows a block diagram of an SDI scrambler that processes two bits per clock cycle. The HDL files `ser_scrambler2.*` contain the module shown in the block diagram. As the “Reference Design Results” section shows, this implementation is only slightly larger than the bit-rate serial scrambler previously described. Since it only needs to run at half the SDI bit-rate, this design can easily support the highest SDI bit-rates.

The `X9002` module described later in this chapter is an example of how to use the `ser_scrambler2` module with the Virtex-II DCM and DDR features to implement an SDI encoder.
Parallel Scrambler

The scrambler function can also be implemented in a parallel manner, processing one 10-bit word every clock cycle. This requires more hardware but only needs to run at one-tenth the bit rate of the SDI link.

In some situations, it is advantageous to use a larger parallel scrambler implementation. Since the parallel descrambler only has to run at the word rate, lower performance FPGAs can be used to support the highest SDI bit rates. With some lower performance FPGAs, it might be necessary to use an external device to serialize and transmit the encoded parallel data generated by the FPGA.

Figure 3-6 shows a block diagram of the module described in the `par_scrambler.*` files. This module accepts a 10-bit input word and generates a 10-bit output word every clock cycle. There are two clock cycles of latency through the scrambler. Also refer to Figure 3-3 for an example of using the `par_scrambler` in an SDI transmitter.

Ten 3-input XOR gates implement the SDI scrambler function. The incoming data bits are combined with the nine bits scrambled in the previous clock cycle and stored in the `scram_reg`. The least significant five bits of the incoming data word are scrambled and fed back into the scrambler to generate the five most significant data bits.

The NRZ-to-NRZI converter is implemented with ten two-input gates that XOR each bit with the bit that preceded it in the bitstream. This requires 11 bits to generate ten NRZI bits. The eleventh bit is the MSB stored in the `out_reg`.

![Parallel Scrambler Block Diagram](image.png)

**Figure 3-6:** Parallel Scrambler Block Diagram

X9002 Example: An Alternative Solution

The Gennum GS9002 was one of the first commercially available SDI encoder integrated circuits. It contained a TRS clipper, an SDI scrambler, and a PLL used to generate the serial clock from the parallel data clock. Although the GS9002 is now obsolete, Xilinx FPGAs can provide an alternative solution when redesigning equipment originally using a GS9002.

Figure 3-7 is a block diagram of the X9002 module provided in the reference design. A Virtex-II DCM multiplies the parallel data clock by five to synthesize a clock that runs at half the SDI bit rate. Two phases of this five times clock, 180° out of phase, are synthesized to drive the DDR logic. A TRS clipper circuit feeds clipped parallel video data to a `ser_scrambler2` module where it is encoded and serialized. Virtex-II DDR flip-flops and a DDR MUX are used to generate the SDI serial bitstream output.
A GS9002 provides a bit-rate clock through a differential PECL driver. This bit-rate clock is for reference and diagnostic purposes and is not required by the SDI standard. The X9002 module duplicates this functionality by using a second DCM to double the five times clock. Rather than using the second DCM to multiply the parallel clock by a factor of ten, the second DCM doubles the five times clock generated by the first DCM. This cascaded DCM configuration was used because the CLKFX output of the Virtex-II DCM currently is not capable of generating a 360 MHz output, but the CLK2X output is capable of this speed. DDR flip-flops are used to drive the serial clock out of the FPGA. This technique ensures a nearly 50% duty cycle on the clock output. This second DCM and associated global clock buffers and the DDR hardware can be easily removed from the design if the reference serial clock output is not required.

X7C9235 Example: An Alternative Solution

The Cypress CY7C9235 SMPTE 259M/DVB-ASI Scrambler-Controller is a parallel implementation of an SDI encoder. It accepts a 10-bit video word and generates a 10-bit encoded video word every clock cycle. The CY7C9235 is designed to operate in two modes, an SDI compliant mode and a DVB-ASI mode when used in conjunction with a Cypress
CY7B9234 transmitter. In SDI mode, the CY7B9234 simply serializes the SDI encoded data supplied by the CY7C9235. In DVB-ASI mode, the CY7C9235 passes the data to the CY7B9234 unmodified and the CY7B9234 performs 8B/10B encoding on the data before serializing it.

Figure 3-8 is a block diagram of the X7C9235 module provided in the reference design. The module includes an instance of the `par_scrambler` module to do parallel encoding of the video data. This design example does not implement the DVB-ASI mode features of the Cypress CY7C9235.

![Figure 3-8: X7C9235 Example Block Diagram](x298_08_101901)

Testing

Figure 3-9 shows the block diagram of a test bench developed for simulation verification of the SDI encoder modules discussed in this chapter, as well as the SDI decoder modules discussed in Chapter 4, “SD-SDI Video Decoder.”

The test bench includes a simple color bar test pattern generator to serve as the source of the video to be sent through the SDI link. The color bar generator and other test pattern generators are described in Chapter 16, “SDTV Video Pattern Generators.”

The video data generated by the color bar generator passes through a pathological TRS test case generator. This module passes the active video data and TRS symbols unchanged. During the horizontal blanking period, the module inserts pathological test sequences. These sequences resemble TRS symbols but differ by just a single bit.

The 10-bit digital video out of the pathological test case generator can be forced to a simulated worst case 8-bit value to exercise the TRS clipper. If the `eight_bit` signal is asserted, bit zero of the video is forced to a one and bit one is forced to a zero. This causes the TRS clipper to clip both the all-zeros and all-ones cases. When the pathological test case generator is actively inserting its test cases, as indicated by the assertion of the `replace` signal, the TRS clipper must be disabled. If it were not, the TRS clipper could turn some of the TRS-like test cases into valid TRS symbols.

The output from the TRS clipper is fed into the various SDI scrambler modules and to a FIFO module. The data in the FIFO module is used as a reference for comparison against the data recovered by the SDI decoders.

The 2-bit wide data path from the half bit-rate scrambler, `ser_scrambler2`, is connected to a DDR flip-flop pair and a MUX to generate a serial bitstream. This bitstream is directly compared against the serial bitstream from the serial scrambler, `ser_scrambler`, and any differences are reported as errors.
Figure 3-9: SDI Test-Bench Block Diagram
The bitstream from the ser_scrambler module passes through a noise generator module before passing to the serial decoders. This noise generator module is capable of corrupting the bitstream in two ways. The noise generator can inject a burst of noise that corrupts random bits in the bitstream. This is intended to simulate electrical noise injected onto the signal. This noise mode is not used in this testbench. It is intended for use with the error detection processors described in Chapter 6, “SD-SDI Ancillary Data and EDH Processors.”

The noise generator can also insert or remove and random number of consecutive bits (from one to nine bits) from the bitstream. This causes the data recovered by the SDI decoder to be unframed and invalid and forces the SDI decoder to reframe at the next TRS symbol. This simulates what happens if the SDI decoder becomes unsynchronized. In actual systems this can occur for a number of different reasons, such as when the video stream is switched to a different, unsynchronized video source. The noise generator only inserts or removes whole bits to test that the SDI decoder detects the offset and reframes. It is not intended to simulate partial bit jitter for testing the clock and data recovery unit. The period between bit insertion or removal is controlled by the OFF_PERIOD parameter in the noise generator module code.

The parallel scrambler module also encodes the video data from the trs_clipper module. The data from the parallel scrambler is serialized and then sent through a noise generator module. The resulting bitstream is called sdi2_stream and drives the parallel SDI decoder chains.

The SDI bitstreams are each descrambled and framed by the descrambler and framer modules from Chapter 4, “SD-SDI Video Decoder.” The data out of each framer is written into a FIFO module. The output of each FIFO module is compared against the reference video stream stored in the FIFO connected to the output of the trs_clipper module.

The X9002 and X7C9235 encoder modules contain their own TRS clipper circuits and are connected to the output of the 10-bit to 8-bit converter. There is a noise generator module connected to the output of each of these two modules. The serial bitstream from the X7C9235 encoder drives the X7C9335 decoder. The serial bitstream from the X9002 encoder drives the X011 decoder (see Chapter 4).

The sdi_fifo module is designed to simplify the comparison of video streams passing through SDI links that have different latencies. Framer designs react differently to the insertion of noise, especially offset noise. To make the comparison easier, the noise generators are monitored to determine when any of them inserts offset noise. As soon as this occurs, all the FIFOs are flushed and they stop storing data until the next TRS symbol is written into the FIFO. In this way, only properly framed data from the various SDI decoder chains are compared against the reference video data.

Verification of the actual hardware of an SDI link involves more than just verifying that a color bar pattern can be passed through the link. The primary areas of concern are the cable equalization and clock and data recovery. Refer to Chapter 2, “SD-SDI Physical Layer Implementation” for more information about testing these areas.

The patho_trs module can be synthesized and placed in an SDI transmitter design. When enabled, the patho_trs module’s pathological test cases are inserted during the horizontal blanking period. These test cases verify that the receiver’s framer function does not falsely detect a TRS symbol when it receives bit sequences similar to TRS symbols. The patho_trs module also generates bit sequences that differ from ANC symbols by one bit. An ANC symbol represents the beginning of an ancillary data block. The ANC-like test patterns generated by patho_trs can be used to verify the proper operation of the receiver logic that looks for ANC blocks.
Reference Design Results

Table 3-2 shows the results after place and route of the various modules implemented in this chapter. All results were obtained using the Verilog versions of the designs with Xilinx ISE version 4.1i. Results using the VHDL files are not shown but are essentially identical. Virtex-II results are for a -5 speed grade device. Spartan™-II results are for a -6 speed grade device.

The ser_scrambler module must run as fast as the bit rate of the SDI link. The ser_scrambler2 and X9002 modules run at half the bit rate. The par_scrambler, trs_clipper, X7C9235, and the patho_trs modules run at the word rate (one-tenth the bit rate).

<table>
<thead>
<tr>
<th>File Name</th>
<th>XST</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size LUTs/FFs</td>
<td>Virtex-II Speed</td>
<td>Spartan-II Speed</td>
</tr>
<tr>
<td>ser_scrambler.v</td>
<td>12/20</td>
<td>450 MHz</td>
<td>280 MHz</td>
</tr>
<tr>
<td>ser_scrambler2.v</td>
<td>14/21</td>
<td>300 MHz</td>
<td>N/A</td>
</tr>
<tr>
<td>par_scrambler.v</td>
<td>29/20</td>
<td>150 MHz</td>
<td>110 MHz</td>
</tr>
<tr>
<td>trs_clipper.v</td>
<td>8/0</td>
<td>2.5 ns</td>
<td>4.8 ns</td>
</tr>
<tr>
<td>X9002.v</td>
<td>28/33</td>
<td>250 MHz</td>
<td>N/A</td>
</tr>
<tr>
<td>X7C9235.v</td>
<td>35/34</td>
<td>150 MHz</td>
<td>110 MHz</td>
</tr>
<tr>
<td>patho_trs.v</td>
<td>55/85</td>
<td>150 MHz</td>
<td>120 MHz</td>
</tr>
</tbody>
</table>

Notes:
1. The ser_scrambler2 and X9002 modules use features unique to the Virtex-II series.

Conclusion

Virtex-II, Spartan-II, and newer FPGAs can implement the SDI encoder function. Because the SDI encoder uses very few FPGA resources, it can be placed in the same device along with other related video functions resulting in a highly integrated design.

Three different implementations of the SDI scrambler are described in this chapter. This allows designers to trade off clock rate versus FPGA area when creating an SDI encoder design. The half-bit rate scrambler is particularly well suited for use with the DDR support in the Virtex-II IOBs.

Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_sd-vid-encoder.zip.
**Chapter 4**

**SD-SDI Video Decoder**

**Summary**

The ANSI/SMPTE 259M-1997 standard specifies a serial digital interface (SDI) for digital video equipment operating at either the 525-line, 60 Hz video standard or the 625-line, 50 Hz video standard [Ref 1]. The SDI standard describes how to transport both composite and component digital video over standard video coax. SDI is widely accepted and often forms the video transportation “backbone” of television studios and broadcast centers.

The SDI standard can be broken down into three main functions: the encoder, the physical layer, and the decoder. Figure 4-1 shows the correlation between the various chapters in this volume and the elements of the SDI link.

This chapter focuses on the SDI decoder. The reference design includes several implementations of the SDI decoder optimized for use with the Virtex™-II family and other Xilinx family features. Both serial (bit-rate) and parallel (word-rate) implementations of the SDI decoder are presented. Design examples are included to illustrate alternative solutions for standard SDI decoder devices, the National CLC011 and the Cypress CY7C9335, by using the decoder implementations developed in this chapter.

![Figure 4-1: SDI Block Diagram and SD-SDI Section Chapters](image)

**SDI Introduction**

**Digital Video Formats**

The SDI standard describes how to transport standard definition digital video serially over a video coax cable. This standard describes the encoding and decoding processes...
performed on the video bitstream for transportation across the physical layer. The standard also describes the electrical and mechanical characteristics of the physical layer. However, it does not define the actual format of the digital video data. Refer to the following additional standards for the definition of SDI compatible digital video formats:

- ANSI/SMPTE 125M, ANSI/SMPTE 267M, and ITU-R BT.601-5 for 4 x 3 and 16 x 9 aspect ratio 4:2:2 component digital video [Ref 1] [Ref 2]
- ANSI/SMPTE 244M for composite NTSC digital video [Ref 1]
- IEC 1179 (now called IEC 61179) for composite PAL digital video [Ref 3]

The SDI standard does not cover high-definition digital video. Another standard, SMPTE 292M, defines a serial digital interface standard for high-definition digital video, commonly called HD-SDI. The bandwidth requirements for high-definition video are significantly higher than for standard definition video. Also, the HD-SDI standard differs from the SDI standard in the way that the video components are interleaved. Because of these higher bandwidth requirements and format differences, the implementation of a HD-SDI decoder has different considerations than that of a standard definition SDI decoder and is not covered in this chapter.

Any of the digital video formats supported by the SDI standard use either 8 bits or 10 bits per data word. The SDI standard is natively a 10-bit format, but allows 8-bit video to be transported across the interface.

**Encoding and Decoding**

Prior to sending digital video serially across the physical layer, a SDI transmitter must encode the video in accordance with the SDI standard. This encoding process is designed to insure that sufficient level transitions occur in the serial bitstream to allow the receiver to recover the clock and data. After the receiver captures the serial data, the decoder must reverse the encoding process to recover the original video data.

The SDI standard uses two generator polynomials, normally expressed as linear feedback shift registers (LFSR), to implement two separate encoding stages. First, the video bitstream is scrambled using the generator polynomial:

\[ G_1(x) = x^9 + x^4 + 1 \]

The output of this first encoding stage is referred to as the scrambled non-return-to-zero (NRZ) bitstream.

The second encoding stage uses the generator polynomial:

\[ G_2(x) = x + 1 \]

to convert the scrambled NRZ bitstream to a polarity-free scrambled NRZ-inverted (NRZI) bitstream. NRZI is DC balanced for transmission across the physical layer. If the bitstream is inverted between the transmitter and the receiver, then the polarity-free nature of the SDI bitstream allows the decoder to properly recover the original data.

The SDI decoder reverses the encoding process by using the same generator polynomials in reverse order: \( G_2 \) to convert from NRZI to NRZ and then \( G_1 \) to descramble the bitstream.
Figure 4-2 illustrates the encoding and decoding processes when implemented in LFSRs. Using standard LFSR notation, the circles with plus symbols inside are exclusive-OR gates. The boxes represent individual flip-flops. The LSB of a data word is sent first.

**Framing**

After decoding the video bitstream, the receiver determines where individual 10-bit words begin and end in the serial bitstream for data-word extraction. This process is called framing. In order to frame the bitstream, a unique and recognizable pattern must be sent periodically for the framer to use as a framing reference.

All of the digital-video formats supported by SDI share similar definitions for the timing reference signal (TRS) symbols. TRS symbols delineate between the active and inactive portions of the video. There are two TRS symbols sent per line of video: one at the start of active video called SAV, and one at the end of active video called EAV. A TRS symbol is sent as four consecutive words, formatted as:

```
3ff 000 000 XYZ
```

The first word transmitted is hexadecimal 3ff and the last word is XYZ. The first three words of the TRS symbol, called the preamble, form a unique bit sequence in the bitstream. The fourth word, called XYZ, varies depending on the specific digital video format being transported.

Since the TRS preamble is common across all the supported digital video formats, is sent on a regular basis, and is unique in the bitstream, it is used as the framing reference. Upon detecting a sequence of ten consecutive ones and twenty consecutive zeros, the framer can determine the proper boundaries of all subsequent data words in the bitstream.

**SDI Bit Rates**

The bit rates supported by SDI range from 143 Mb/s to 360 Mb/s, depending on the digital video format being transported. The SDI standard defines four different bit rates and identifies their "support levels" as shown in Table 4-1. SDI compliant equipment is not required to support all bit rates. A piece of equipment supporting bit rates up to 270 Mb/s is said to conform to ANSI/SMPTE 259M-ABC, meaning it supports levels A, B, and C.
Chapter 4: SD-SDI Video Decoder

Table 4-1: SDI Standard Bit Rates

<table>
<thead>
<tr>
<th>Support Level</th>
<th>Bit Rate</th>
<th>Video Format</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level A</td>
<td>143 Mb/s</td>
<td>NTSC composite</td>
<td>ANSI/SMPTE 244M-1995</td>
</tr>
<tr>
<td>Level B</td>
<td>177 Mb/s</td>
<td>PAL composite</td>
<td>IEC 61179</td>
</tr>
</tbody>
</table>

Error Detection

The SDI standard does not mandate the use of an error detection mechanism. Some of the digital video standards, SMPTE 125M for example, specify error detection bits in the XYZ word of the TRS symbol to determine the validity of the TRS symbol. However, the SDI standard highly recommends embedding error detection checkwords into the SDI video stream as described in SMPTE RP 165-1994.

At the receiving end of the SDI link, checksums are generated for the incoming bitstream and compared to the embedded checkwords. Error detection can only be done after the bitstream is descrambled and framed. An optional error detection module can simply be bolted onto the output of any of the framer modules described in this document. SDI error detection modules are described in Chapter 6, “SD-SDI Ancillary Data and EDH Processors.”

Reference Design

The descrambler and framer functions are implemented as separate modules. Serial and parallel implementations of both modules are provided.

Serial Descrambler Implementation: ser_descrambler.*

The descrambling process involves "division" of the incoming bitstream by the generator polynomials. This can be implemented very simply by a LFSR configured as shown in the SDI descrambler diagram in Figure 4-2, page 73. A serial implementation results in a very small amount of hardware. However, a serial implementation of the descrambler must run at the full bit rate of the SDI interface, up to 360 MHz. A serial implementation also requires the availability of a bit-rate clock for the FPGA. By using Virtex-II devices, a serial descrambler can be implemented to support SDI bit-rates up to 360 Mb/s.

The HDL files ser_descrambler.* contain direct implementations of the descrambler LFSR described in Figure 4-2. As shown in “Reference Design Results”, this implementation occupies eleven flip-flops and two LUTs for both Virtex-II and Spartan-II devices. In Virtex-II devices, the serial descrambler runs fast enough to support the highest bit rate supported by the SDI standard. In a Spartan-II device, it supports the 270 Mb/s rate.

It is also be possible to implement the LFSR using the SRL16 feature found in the Virtex architecture. An SRL16-based implementation is smaller than the implementation presented here, potentially about half the size. However, it is more difficult to get an SRL16-based implementation to run at the highest SDI bit-rates.
The ser_descrambler module contains two control inputs, NRZI and DESC, to enable the NRZI-to-NRZ conversion and the descrambler, respectively. These control signals allow the two functions to be bypassed if the incoming data is non-SDI compliant. In normal SDI operation, both inputs should be tied High.

**Parallel Descrambler Implementation: par_descrambler.***

The descrambler function can also be implemented in a parallel manner, processing one 10-bit word every clock cycle. This obviously requires more hardware but only needs to run at one-tenth the bit rate of the SDI link.

In some situations it is advantageous to use a larger parallel descrambler implementation. The FPGA could receive parallel data from some external SDI receiver device performing a serial-to-parallel conversion of the data. It might also be more economical to use a parallel implementation. Since the parallel descrambler only has to run at the word rate, lower performance FPGAs can be used to support the highest SDI bit rates. An optimized parallel descrambler is actually only about twice as much hardware as the serial descrambler described in the previous section when implemented in a Xilinx FPGA.

A block diagram of the module described in the par_descrambler.* files is shown in Figure 4-3. This module accepts a 10-bit input word and generates a 10-bit output word every clock cycle. There is a one-clock cycle latency through the descrambler, caused by the output register.

The NRZI-to-NRZ converter is implemented as ten 2-input gates that XOR each bit with the bit that preceded it in the bitstream. This requires the availability of eleven NRZI bits to generate ten NRZ bits. The eleventh bit comes from storing the MSB of the input-data word in a register making it available to be XORed with the LSB of the data word received in the next clock cycle.

Ten 3-input XOR gates form the SDI descrambler. These gates generate the ten descrambled output bits by combining 19 bits from the NRZI-to-NRZ converter. Ten of the input bits are from the current output of the NRZI-to-NRZ converter. The other nine-input bits were generated by the NRZI-to-NRZ converter during the previous clock cycle and are stored in the desc_in register.

As shown in “Reference Design Results”, this parallel descrambler turns out to be fairly small, surprisingly only about twice the size of the serial descrambler, and runs well above the 36 MHz clock rate required to support the highest SDI bit rate.

The parallel descrambler is implemented with a clock enable input called ld (Load). If ld is asserted for one clock cycle out of every ten, the parallel descrambler can be clocked by the bit-rate clock. If the parallel descrambler receives a word-rate clock, ld should be tied High.

![Figure 4-3: Parallel Descrambler Block Diagram](image-url)
Framer Implementation

The data stream coming out of the descrambler is unframed. There are no indications of where the actual word boundaries occur in the data. If the framer is receiving 10-bit wide parallel data words from the descrambler, the actual video word boundaries do not necessarily correspond to the arbitrary word boundaries of the incoming data.

The framer must scan the unframed data stream for the 30-bit TRS preamble that consists of ten consecutive "1" bits followed by twenty consecutive "0" bits. When a TRS preamble is detected, the framer can resynchronize to the 10-bit word boundaries in the video stream in order to generate properly framed video data words.

Most commercially available SDI framers allow control over whether the framer should resynchronize if it receives a TRS at a new offset. Sometimes an error in the video bitstream causes a false TRS to be detected. It is also possible that a non-SDI standard compliant data stream is occasionally transmitted over an SDI link. In this case, the receiver must temporarily disable resynchronization because the non-SDI data could contain bit sequences falsely detected as TRS symbols. These false TRS symbols can be ignored or filtered if the framer has an input to selectively control when resynchronization occurs.

The framer modules each have an input called $frame_en$ to control automatic resynchronization. If $frame_en$ is Low, the framer detects new TRS offsets, but it does not resynchronize; therefore, subsequent data words output by the framer are potentially not framed properly.

The framer module also has a new start position ($nsp$) output. This signal can also be interpreted as an indication of the presence of a framing error. It is asserted High when a TRS is detected at an offset different from the current offset used by the framer. It remains asserted until the offset error is corrected by either receiving another TRS matching the offset used by the framer, or by receiving another TRS when $frame_en$ is asserted High.

There are several ways to use the $nsp$ output and the $frame_en$ input to control how and when the framer modules respond to new TRS offsets:

1. If $frame_en$ is tied High, the framer always resynchronizes to new TRS offsets.
2. If $frame_en$ is tied Low, the framer does not resynchronize. This is primarily useful when the receiver knows the data sent over the interface is non-SDI compliant data. This is generally not useful without control logic to enable and disable the resynchronization at the appropriate times.
3. If $frame_en$ is tied to the $nsp$ output of the framer, automatic filtering of TRS offsets occurs. When a TRS symbol is detected that is at a new offset, $nsp$ is asserted High, but the framer does not change the offset until another TRS symbol is detected. This filters out one-time TRS offset errors.
4. More sophisticated TRS offset filtering algorithms can be implemented by designing a state machine to monitor the $nsp$ output and control $frame_en$.

During the time that the four words of the TRS symbol are present on the output, the framer module generates an asserted High $trs$ output. The $trs$ output is asserted High even if automatic resynchronization has been disabled ($frame_en$ Low) and the TRS is at a new offset. In this case, $trs$ is asserted, but the TRS symbol coming out of the framer is not properly framed. If it is desired that $trs$ only be asserted when framed TRS symbols are output, it is a simple matter to use $nsp$ to qualify the $trs$ output.

The framer modules generate 10-bit words. If 8-bit video is being sent through the SDI link, then use only the eight MSBs of the framer module output port.
As with the descrambler function, the framer function can be implemented both in a serial manner and in parallel. The same trade-offs apply. A serial framer is smaller but must run at the bit rate. A parallel framer is larger but only needs to run at $\frac{1}{10}$ the bit rate.

**Serial Framer:** ser_framer.*

The block diagram of the serial framer implementation is shown in Figure 4-4. The incoming video bitstream passes through a series of shift registers that, when combined, delay the video bitstream by 32-clock cycles. These shift registers are the three registers located along the top of Figure 4-4. The 32-bit delay generated by these video shift registers allows the TRS detection logic to examine 30 consecutive bits for a TRS symbol and determine if the framer needs to be resynchronized before any of those bits appear on the output port.

To detect a TRS preamble, a 10-bit wide AND gate and a 10-bit wide NOR gate determine if the contents of the trs_detect register contains all zeros or all ones. The output of the AND gate (all_ones) is delayed through a 20-bit long shift register (ones_reg). The output of the NOR gate (all_zeros) is delayed through a 10-bit long shift register (zeros_reg). By ANDing together the output of the ones_reg, zeros_reg and the all_zeros signal, a trs_detected signal is generated to only be asserted when the TRS preamble is contained in the video shift registers.

The offset logic block contains a 10-bit ring counter called bit_cntr. This counter is reset to its starting count when the framer resynchronizes. Otherwise, it causes the out_reg to load the 10 bits contained in the des_reg once every ten clock cycles.

The offset logic block generates the trs, nsp, and out_rdy outputs. The out_rdy signal is asserted for one clock cycle when the out_reg is reloaded. This signal is generated by assigning it to one of the bits of the bit_cntr. If downstream logic requires more setup time, the clock cycle when out_rdy is asserted can easily be changed by changing the bit_cntr bit assigned to out_rdy.

![Serial Framer Block Diagram](image)

Two slightly different implementations of this serial framer are provided. The ser_framer.* files are coded to cause the synthesis tool to infer flip-flops for all the shift registers in the module. This results in the best performance and, as can be seen from “Reference Design Results,” it runs at over 400 MHz in a Virtex-II device, sufficient to support the highest bit rates of the SDI specification.
Chapter 4: SD-SDI Video Decoder

A more compact, but potentially lower performance, implementation is provided in the `ser_framer_srl16.*` files. This version codes the `delay_reg`, `ones_reg`, and `zeros_reg` as arrays, allowing most synthesis tools to infer SRL16 blocks for these registers. This results in a significant reduction in the size of the module ("Reference Design Results"), but might not always produce the fastest results.

Parallel Framer: `par_framer.*`

Figure 4-5 shows a block diagram of a parallel implementation of a framer. The `par_framer.*` files contain the HDL descriptions of the parallel framer module.

The parallel framer accepts 10-bit unframed data words. It looks for 30-bit TRS preambles that can begin at any of the 10 bits in the input word and can span from the first word through the next two or three words. The TRS detection logic needs to look across a total of 39 bits to determine if a TRS symbol is present and to determine its offset.

The incoming data is pipelined through three cascaded registers called `in1_reg`, `in2_reg`, and `in3_reg`. The 30 bits from these three registers plus the nine LSBs from the input port form the 39-bit wide vector that the TRS detection logic examines.

A series of 10-bit wide AND and NOR gates examine the 39-bit input vector to determine if a TRS symbol is present. If so, an internal `trs_detected` signal is asserted, and the offset of the TRS symbol is determined. The offset encoder produces a numerical offset value indicating the starting bit position of the TRS symbol. The output of the offset encoder is compared to the current offset value stored in the offset register to determine if the newly detected TRS symbol is at a different offset position. The `nsp` logic uses the output of the comparator to generate the `nsp` signal and to load the offset register from the output of the offset encoder when resynchronization occurs. The offset register controls a barrel shifter that extracts the 10-bit output word from a 19-bit wide piece of the input video stream.

As shown in "Reference Design Results," the parallel framer is fast enough to support all bit rates of the SDI standard.

It is tempting to try to reduce the number of 10-bit wide NOR gates from twenty to ten by using one set of ten gates sequentially. This technique was explored and found to produce about the same size result. With some synthesis tools it actually produced a much larger implementation.

The parallel framer uses a barrel shifter to extract the framed data from the bitstream. In the original Verilog code, this was implemented with a simple assignment statement using the right shift operator. This produced widely varied results with different synthesis tools. The barrel shifter was subsequently re-coded with two levels of multiplexers. This produces good results in all synthesis tools and in both Verilog and VHDL.

When using Virtex-II technology, it is possible to use an embedded 18 x 18 multiplier to implement most of the barrel shifter, as described in Xilinx Application Note XAPP195 [Ref 4]. The files `par_framer_mult.*` use one 18 x 18 embedded multiplier to generate the nine LSBs of the multiplier. A 10-to-1 MUX is used to generate the tenth bit of the barrel shifter. If it is a Virtex-II design and a free multiplier is available, then using the embedded multiplier results in significant savings in the number of LUTs required to implement the parallel framer module.
CLC011 Example: An Alternative Solution

The National CLC011 Serial Digital Video Decoder is a serial implementation of an SDI decoder. It accepts a serial bitstream and a bit-rate clock and produces 10-bit-wide data words on its output.

Using the ser_descrambler and ser_framer modules, it is easy to implement an alternative to the CLC011 using Xilinx FPGAs. The inputs and outputs of these two modules are very similar in function to the corresponding signals on the CLC011. However, the CLC011 provides two output signals that are not generated by the ser_framer module: PCLK and EAV.

PCLK is used to indicate to downstream logic that data is valid on the PD outputs. The ser_framer generates an out_rdy signal that is similar to PCLK but does not have a 50% duty cycle. The out_rdy signal is better suited if the downstream logic is in the same FPGA as ser_framer since it can be used as a clock enable signal. If a true PCLK signal is required, it is quite simple to modify the ser_framer to generate a PCLK signal using the bit_cntr.

The EAV (end of active video) output is asserted Low during the time that the fourth word of a TRS symbol is present on the outputs and bit six (the H bit) of that word is Low.

The framer_X011 module generates both the PCLK and EAV outputs. The combination of the ser_descrambler module and the framer_X011 module completes an alternative implementation of a CLC011. The file X011.* contains the top level HDL descriptions.

CY7C9335 Example: An Alternative Solution

The Cypress CY7C9335 SMPTE 259M/DVB-ASI Descrambler/Framer-Controller is a parallel implementation of a SDI decoder. It accepts a 10-bit scrambled video word and generates a 10-bit descrambled and framed video word every clock cycle. The par_descrambler and par_framer modules plus a small amount of extra logic supplies most of the functionality of the CY7C9335. The file X7C9335.* files contain HDL descriptions of the X7C9335 design shown in Figure 4-6.
Chapter 4: SD-SDI Video Decoder

The CY7C9335 is designed to operate in two modes, an SDI compliant mode and a DVB-ASI mode when used in conjunction with a Cypress CY7B9334 receiver. The CY7B9334 receiver takes in the serial bitstream, does clock and data recovery, and deserializes the bitstream before sending it to the CY7C9335 decoder.

In the DVB-ASI mode, the SDI decoder (both the descrambler and the framer functions) are effectively bypassed and the CY7C9335 simply passes the incoming data straight through to its outputs. In this case, the CY7B9334 implements 8B/10B decoding as defined in the DVB-ASI standard. An input signal called DVB_EN puts the CY7C9335 into DVB-ASI mode. The CY7C9335 also bypasses the SDI decoding if the BYPASS input is High and DVB_EN is Low. A multiplexer prior to the output register implements the bypass functionality controlled by DVB_EN and BYPASS.

The input and output signals of the X7C9335 are similar in function to the signals of the CY7C9335. The following paragraphs describe these signals.

The X7C9335 has an input to control how framer resynchronization occurs. This signal, called sync_en, forces the X7C9335 to always resynchronize (sync_en Low) or to filter out single erroneous TRS symbols (sync_en High).

The CY7C9335 has an output called RF that is designed to connect to the CY7B9334. This output is the inverted and registered DVB_EN input. The X7C9335 replicates this signal with its rf output.

The X7C9335 has a horizontal sync output signal called h_sync. This signal toggles states every time a TRS symbol is detected by the framer. If sync filtering is enabled (sync_en High), h_sync still toggles even if the detected TRS symbol is at a new offset position. If the dvb_en input is Low, h_sync does not toggle.

The X7C9335 has a synchronization error output signal called sync_err. If TRS filtering is enabled (sync_en High), this signal pulses High for one (word-rate) clock cycle when the framer filters out a TRS symbol that is offset from the current framer reference.

The CY7C9335 contains a DVB-ASI mode state machine. This state machine generates an output called A/B used to control the CY7B9335 device. The purpose of this signal is to cause the CY7B9335 to invert the DVB-ASI data stream if too many errors occur. If DVB-ASI data streams are routed through SDI switches or repeaters, they can become inverted and can not be decoded by the CY7B9335’s 8B/10B decoder. By examining the data stream for errors, the state machine toggles the A/B signal if too many errors are detected in order to try and compensate for an inversion in the data stream. This A/B output and state machine have not been implemented in the X7C9335 design.

![X7C9335 Block Diagram](x288_05_033005)

**Figure 4-6: X7C9335 Block Diagram**
Reference Design Results

The Table 4-2 shows the results after place and route of the various modules implemented in this chapter. All results were obtained using the Verilog versions with Xilinx ISE version 3.3i. Results using the VHDL files are not shown but are essentially identical. Virtex-II results are for a -5 speed grade device. Spartan-II results are for a -6 speed grade device.

<table>
<thead>
<tr>
<th>File</th>
<th>XST Size in LUTs/FFs</th>
<th>Virtex-II Speed</th>
<th>Spartan-II Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ser_descrambler.v</td>
<td>2/11</td>
<td>490 MHz</td>
<td>300 MHz</td>
</tr>
<tr>
<td>par_descrambler.v</td>
<td>19/20</td>
<td>440 MHz</td>
<td>260 MHz</td>
</tr>
<tr>
<td>ser_framer.v</td>
<td>19/91</td>
<td>380 MHz</td>
<td>300 MHz</td>
</tr>
<tr>
<td>ser_framer_srl16.v</td>
<td>23/53</td>
<td>355 MHz</td>
<td>300 MHz</td>
</tr>
<tr>
<td>par_framer.v</td>
<td>84/49</td>
<td>100 MHz</td>
<td>80 MHz</td>
</tr>
<tr>
<td>par_framer_mult.v</td>
<td>51/55</td>
<td>85 MHz</td>
<td>N/A(1)</td>
</tr>
<tr>
<td>X011.v</td>
<td>24/103</td>
<td>390 MHz</td>
<td>300 MHz</td>
</tr>
<tr>
<td>X7C9335.v</td>
<td>115/106</td>
<td>100 MHz</td>
<td>75 MHz</td>
</tr>
</tbody>
</table>

Notes:
1. par_framer_mult is applicable only to Virtex-II devices since it uses an embedded 18 x 18 multiplier.

Testing

The best way to test the SDI decoder modules is in a test bench, connecting to an SDI encoder module being driven by a video generator. Chapter 3, “SD-SDI Video Encoder” describes not only the design of a SDI encoder module, but also details the implementation of the test bench to test the decoder modules implemented in this chapter.

Conclusion

Xilinx FPGAs can implement an SDI decoder function thus replacing costly external components. The Virtex-II devices are fast enough to implement an SDI standard decoder in a serial fashion thus producing a very compact implementation running at the full 360 Mb/s rate. Parallel implementations of the SDI decoder are also possible. These parallel implementations are somewhat larger, but only need to run at one-tenth the bit rate of the SDI link.

Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_sd-vid-decoder.zip.
Summary

The SMPTE 259M Serial Digital Interface (SDI) standard describes how to transport standard-definition digital video serially over video coax cable. SDI is commonly used as the video transportation backbone of most broadcast studios and video production centers. This chapter describes implementations of a video standard detector and a flywheel video decoder, suitable for use with Xilinx FPGAs.

Introduction

Figure 5-1 is a block diagram showing correlation between the various chapters in this volume and the elements of the SDI link.

Before transmission over an SDI link, digital video is usually processed to insert error detection checkwords. These checkwords allow the receiver to detect transmission errors. Ancillary data packets can also be inserted into the inactive (blanked) portions of the video to carry non-video data, such as digital audio. At the receiving end of the SDI link, the digital video is again processed to detect transmission errors, extract ancillary data, or insert additional types of ancillary data.

SDI is compatible with a variety of different digital video standards. Because the locations of the error detection and ancillary data packets vary with the digital video standard, a video processor must know which video standard is currently being processed. Most digital video processors have a video standard detector examining the video stream to automatically determine the video standard.
In addition to determining the video standard, a video processor must also synchronize itself to the input video stream. It must know the vertical and horizontal position of the current video sample. The video decoder function synchronizes to the input video stream and keeps running counts of the current video line number and current horizontal position. With this information, the video processor knows when to insert or extract the error detection checkwords and ancillary data packets. A special type of video decoder, the flywheel video decoder, is often used in video processors because it provides immunity from noisy, error prone, or briefly interrupted input video streams.

Digital Video Standards

There are many different video standards, both analog and digital. Today, most broadcast studios and video production centers use component digital video when creating, storing, and transporting video. Component digital video can be readily compressed using digital video compression standards. It can also be encoded into analog composite video for broadcast.

SDI supports a variety of standard-definition digital video standards. The ANSI/SMPTE 259M-1997 SDI standard [Ref 1] defines how to serially transport the digital standards listed in Table 5-1. The documents listed describe the parallel form of these video standards, and the SDI standard describes how to convert the parallel video data to a serial format.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Description</th>
<th>Serial Bit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMPTE 125M [Ref 2] and ITU-R BT.601-5 [Ref 3]</td>
<td>NTSC &amp; PAL 4x3 aspect ratio 4:2:2 component digital video</td>
<td>270 Mb/s</td>
</tr>
<tr>
<td>SMPTE 267M [Ref 4]</td>
<td>NTSC 16x9 aspect ratio 4:2:2 component digital video</td>
<td>360 Mb/s</td>
</tr>
<tr>
<td>SMPTE 244M [Ref 5]</td>
<td>NTSC Composite Digital Video</td>
<td>143 Mb/s</td>
</tr>
<tr>
<td>IEC 61179 [Ref 6] and EBU Tech. 3280-E [Ref 7]</td>
<td>PAL Composite Digital Video</td>
<td>177 Mb/s</td>
</tr>
</tbody>
</table>

SDI is compatible with both component and composite digital video. Component digital video is very commonly used in the broadcast industry, but composite digital video is not as common. The reference designs accompanying this chapter support only component digital video.

Since the original SDI specification was introduced, it has been adapted to accommodate some additional digital video standards. Among these are NTSC and PAL 4:4:4 component digital video standards. The parallel format for the NTSC and PAL 4:4:4 video standards are described in SMPTE RP 174-1993 [Ref 8] and ITU-R BT.799-3 [Ref 9]. The serial data rate for both 4:4:4 digital component video standards is 540 Mb/sec. The SDI specification was extended to cover the 540 Mb/sec data rate by SMPTE 344M-2000 [Ref 10].

Chapter 16, “SDTV Video Pattern Generators” contains descriptions of the 4 x 3 aspect ratio, 4:2:2 component digital video standards. The NTSC and PAL 16x9 aspect ratio 4:2:2 component digital video standards are very similar to the 4 x 3 aspect ratio standards and simply have more samples per line. The 4:4:4:4 standards differ more significantly from the 4:2:2 standards, as described below.
The 4:2:2 digital video standards use two data words per video sample. Each video sample contains a luma word (Y) and one chroma word. Consecutive video samples alternate between containing a blue color-difference chroma word (Cb) and a red color-difference chroma word (Cr).

In contrast, the 4:4:4:4 digital video standards have four data words per video sample. These standards support either the YCbCr color space or the RGB color space. If YCbCr is used, each video sample contains words for the Y, Cb, and Cr components plus a fourth auxiliary component designated as A. If the RGB color space is used, each video sample contains a word for each red, green, blue, and A component. The A component typically carries the key channel, indicating the transparency of the sample. In the RGB color space, this key channel is often called the alpha channel.

The format of the XYZ word of the timing reference signal (TRS) symbol is different for the 4:4:4:4 standards. A flag bit called S has been added to indicate the color space used in the video stream. The S bit is Low for RGB and High for YCbCr. The format of the XYZ word for the 4:4:4:4 TRS symbol is shown in Table 5-2.

The bits labeled P4 through P1 are protection bits calculated in the following manner:

\[
P_4 = V \oplus V \oplus H \\
P_3 = F \oplus V \oplus S \\
P_2 = V \oplus H \oplus S \\
P_1 = F \oplus H \oplus S
\]

**Video Standard Detection**

A video processor can be designed to support several different standards of digital video. To properly process the video stream, the processor must first determine the video standard of the input video stream.

Until very recently, digital video streams did not carry any explicit identification information to indicate the video standard. In 2001, the SMPTE 352M standard[Ref 11] was released. This standard describes a standard ancillary data packet carrying "video payload" identification information. When widely adopted, this standard will simplify the process of detecting the video standard of a digital video stream. The video processor will simply be able to look for and decode the identification information in the ancillary data packet.

For now, however, more traditional methods of video standard detection must still be used. The video processor must determine the video standard by examining the timing of the video stream. All digital video standards supported by SDI contain TRS symbols. These symbols occur in the video stream whenever the video timing signals change. There are three timing signals in the TRS symbol called F, V, and H. The F bit indicates the current field (odd or even). The V bit is asserted during the vertical blanking interval of each field. The H bit is asserted during the horizontal blanking interval of each line.

Each of the six component digital video standards supported by this chapter contains a different number of data words on a line of video. A video standard detector finds the TRS symbols marking the beginning of each video line and counts the number of words
between those symbols, then compares the results against the known video standards. Table 5-3 shows the number of samples on a line of video for each of the video standards supported by this chapter.

Table 5-3: Words Per Video Line

<table>
<thead>
<tr>
<th>NTSC/PAL</th>
<th>Sampling Scheme</th>
<th>Aspect Ratio</th>
<th>Words Per Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTSC</td>
<td>4:2:2</td>
<td>4:3</td>
<td>1716</td>
</tr>
<tr>
<td>NTSC</td>
<td>4:2:2</td>
<td>16:9</td>
<td>2288</td>
</tr>
<tr>
<td>NTSC</td>
<td>4:4:4:4</td>
<td>4:3</td>
<td>3432</td>
</tr>
<tr>
<td>PAL</td>
<td>4:2:2</td>
<td>4:3</td>
<td>1728</td>
</tr>
<tr>
<td>PAL</td>
<td>4:2:2</td>
<td>16:9</td>
<td>2304</td>
</tr>
<tr>
<td>PAL</td>
<td>4:4:4:4</td>
<td>4:3</td>
<td>3456</td>
</tr>
</tbody>
</table>

Most video standard detectors require some number of consecutive lines to contain the same number of samples before reporting that the video standard has been detected. This same function can be used to provide noise immunity by preventing the video standard detector from inadvertently switching to a new video standard when receiving a few lines containing the incorrect number of words, possibly caused by noise in the video stream.

If the video processor supports composite digital video, then it must determine whether the video stream is composite or component digital video. This can be done by examining the fourth word of any TRS symbol in the video stream. For composite digital video, all the bits of this word are zero. For component digital video, the most significant bit (bit 9) of the fourth word is always a one. Therefore, by simply looking at bit 9 of the fourth word of any TRS symbol, a video standard detector can determine whether the video stream contains composite or component digital video.

This design does not support composite digital video. The video standard detector in the reference design determines the type of video stream, and simply stays “unlocked” when it finds composite video.

## Flywheel Video Decoder

### Basic Video Decoding

Inserting and extracting information, such as ancillary data packets, requires the video processor to know exactly the current line number and horizontal position of the sample being received and processed. The primary purpose of the video decoder is to synchronize to the incoming video stream and provide the current horizontal and vertical position to other modules in the video processor.

To find the current horizontal position, a video decoder watches for a start of active video (SAV) TRS symbol. The sample immediately after the last word of the SAV symbol is the first sample in the active portion of the line (sample 0).

To find the current vertical position, the video decoder must watch for a transition of the F bit, indicating the beginning of a new field. When a field transition occurs, the video decoder can determine the current line number if it also knows the video standard. Table 5-4 shows the starting line numbers of each field for both NTSC and PAL video.
Using a Flywheel for Noise Immunity

A flywheel video decoder is often used in video processors to provide immunity from noise and interruptions in the input video stream. Flywheel video decoders use the same techniques just described to synchronize to the input video stream. Once synchronized, however, the flywheel generates its own video timing for the video stream. It continuously compares its internally generated video timing information with the input video stream. When a difference occurs, the flywheel decoder does not immediately resynchronize with the input video stream. Instead, it continues to generate video timing unchanged, as if it had momentum carrying it forward. If the input video stream contains only a few corrupted data words, it usually resumes in sync with the flywheel. However, if synchronization is lost because the video stream was switched to a different source or standard, the flywheel eventually determines that it must resynchronize with the incoming video stream.

Not only does the flywheel decoder provide noise immunity for the video decoder function, but it also can provide several other valuable functions.

The video timing information produced by the flywheel decoder can be used to repair damaged or invalid TRS symbols in the input video stream. Because the flywheel is generating video timing information, the video processor can generate correct TRS symbols. These can be inserted in place of the TRS symbols in the input video stream, thereby, repairing any TRS symbols that have been corrupted by noise.

The flywheel decoder continues to generate and insert TRS symbols into the video stream even when the input video stream is interrupted. Obviously, the visual information in the resulting video stream is invalid. But, the timing information contained in the TRS symbols is valid. This is useful because it keeps all downstream video equipment synchronized.

Synchronous Switching Considerations

SDI video streams are often sent through video routers. In most cases, broadcast studios take care to insure synchronization of the various video streams into the router. The input video streams usually are synchronized to the same video line. However, the video streams are not always synchronized to precisely the same horizontal position on the line.

When a router switches between these synchronous video sources, the receiving equipment sometimes detects some small horizontal offset of the EAV symbol on the line where the switch occurs. Normally, a flywheel decoder would ignore this EAV offset until it detected the offset occurring repeatedly over some number of consecutive lines. Only then would the flywheel resynchronize. However, when switching between closely synchronized video sources, it is better for the flywheel decoder to instantly resynchronize.

SMPTE recommended practice RP 168-1993 defines one line per field when synchronous switching is allowed to occur [Ref 12]. Table 5-5 shows the synchronous switching lines for both fields of both the NTSC and PAL video standards. These lines were carefully chosen to minimize disturbances to timing and other vital data. Other digital video standards

<table>
<thead>
<tr>
<th>NTSC/PAL</th>
<th>Odd Field Starting Line Number ((F = 0))</th>
<th>Even Field Starting Line Number ((F = 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTSC</td>
<td>4</td>
<td>266</td>
</tr>
<tr>
<td>PAL</td>
<td>1</td>
<td>313</td>
</tr>
</tbody>
</table>
forbid the placement of critical information on these synchronous switching lines, since these lines are subject to corruption during the switch.

### Table 5-5: RP 168 Synchronous Switching Line Numbers

<table>
<thead>
<tr>
<th>NTSC/PAL</th>
<th>Odd Field Synchronous Switching Line Number</th>
<th>Even Field Synchronous Switching Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTSC</td>
<td>10</td>
<td>273</td>
</tr>
<tr>
<td>PAL</td>
<td>6</td>
<td>319</td>
</tr>
</tbody>
</table>

According to RP 168, a synchronous switch must occur only during a window of a few hundred samples located in about the middle of the active portion of the synchronous switching line. This insures that the switch occurs well after the SAV symbol and well before the EAV symbol, thus minimizing the chance that these important video timing signals will be corrupted by the switch.

A video flywheel decoder should accommodate horizontal offsets that occur on these synchronous switching lines and immediately resynchronize to the incoming video stream, if such an offset is detected. If a vertical offset or field difference is detected on a synchronous switching line, the switch is asynchronous and the flywheel should implement its normal resynchronization process.

### Tolerating an Early Falling Transition of the V Bit

The current standards for NTSC component digital video, SMPTE 125M-1995 and ITU-R BT.601-5, require the V bit to transition from a High to a Low on lines 20 and 283, marking the end of the vertical blanking interval. Earlier versions of the NTSC component digital video specifications, however, allowed the V bit to fall Low on any line from 10 to 20 for the odd field and 273 to 283 for the even field.

The current specifications recommend tolerance of early V bit transitions to allow for compatibility with video equipment designed to earlier versions of the specifications. Since the video flywheel decoder is generating its own version of the V bit, it might detect a discrepancy in the V bit on video streams generated by older equipment. The flywheel decoder should tolerate these discrepancies, but only on those lines where the V bit was permitted to transition early in the previous standards.

PAL component digital video specifications have always precisely specified the line when the V bit should transition, so this does not apply to the PAL standards.

### Reference Design

The reference design contains a top-level module called `video_decode`. This module is a wrapper around three modules: the video standard detector (autodetect), the flywheel video decoder (flywheel), and a preprocessor module that is used to examine the video stream for TRS symbols and other special patterns (trs_detect). Figure 5-2 is a block diagram of the `video_decode` module.

There can be some cases where the autodetect module is not required. If an application always processes the same video standard, or if the video standard is provided some other way, by a front panel selector switch for example, the autodetect function can be eliminated from the design. In these cases, the `std_in` inputs to the flywheel module should be hardwired to the video standard or controlled by some external function. The `std_locked` signal should always be asserted. The `rx_xyz_err` input of the flywheel module should be connected to either the `rx_xyz_err` or the `rx_xyz_err_4444` output...
of the \textit{trs\_detect} module, depending on the video standard. Finally, the \texttt{s4444} input of the \textit{flywheel} module should be correctly controlled if one of the 4:4:4 formats is selected.

The \textit{video\_decode} module delays the video stream by six clock cycles. Figure 5-3 shows the timing relationships between many of the output signals of the \textit{video\_decode} module. The diagram shows an EAV symbol followed immediately by the first part of an ANC packet. If the ANC packet were an EDH packet, then the \texttt{edh\_next} signal would be asserted at the same time as the \texttt{anc\_next} signal. The diagram ends with an SAV symbol.

![Figure 5-2: Video Decoder Block Diagram](x626_02_008002)

![Figure 5-3: Timing Diagram for video\_decode](x625_03_033105)

\textbf{Trs\_detect Module}

The \textit{trs\_detect} module performs some preliminary parsing of the video stream, looking for certain special word patterns. This module has a four clock deep register pipeline that
delays the video while the module examines it. Figure 5-4 is a block diagram of the \textit{trs\_detect} module.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{trs_detect_block_diagram.png}
\caption{\textit{trs\_detect} Block Diagram}
\end{figure}

The \textit{trs\_detect} module performs the following functions:

\begin{itemize}
  \item The \textit{trs\_detect} module detects TRS symbols occurring in the video stream. TRS symbols are four words long. The first three words are a pattern unique in the video stream: \texttt{3FFh, 000h, and 000h}. When this pattern is detected, the \textit{trs\_detect} module asserts the \texttt{rx\_trs} signal while it outputs the first word of the TRS symbol.
  \item If a TRS symbol is detected, the \textit{trs\_detect} module decodes the fourth word of the TRS symbol, called the XYZ word. It asserts the \texttt{rx\_xyz} signal as it outputs the XYZ word. It also asserts either the \texttt{rx\_eav} or \texttt{rx\_sav} signals, depending on whether the TRS symbol is an EAV or SAV. The \texttt{rx\_eav} and \texttt{rx\_sav} signals are only asserted when the first word of a TRS symbol is output from the \textit{trs\_detect} module (when \texttt{rx\_trs} is asserted). These signals provide a look-ahead function for the video processor, indicating whether the TRS symbol is an EAV or SAV before the XYZ word of the TRS symbol appears in the output video stream from the \textit{trs\_detect} module.
  \item The \textit{trs\_detect} module checks the protection bits of the XYZ word to determine if it contains an error. The module asserts the \texttt{rx\_xyz\_err} signal if the XYZ word contains an error. This signal is only valid if the video standard is one of the 4:2:2 standards. A different error signal, \texttt{rx\_xyz\_err\_4444} indicates the detection of an error in the XYZ word for the 4:4:4:4 video standards. Because the \textit{trs\_detect} module does not know which video standard is being received, it always examines the XYZ word for errors in both formats. These two error signals are only valid when the \texttt{rx\_xyz} signal is asserted.
  \item The \textit{trs\_detect} module latches the F, V, and H bits from the TRS symbol’s XYZ word. These latched bits are output from the \textit{trs\_detect} module as the \texttt{rx\_f}, \texttt{rx\_v}, and \texttt{rx\_h} signals. These signals remain valid until the next TRS symbol is detected. These signals always transition at the beginning of the TRS symbol.
  \item The \textit{trs\_detect} module detects ancillary data (ANC) packets. An ANC packet begins with a three-word ancillary data flag (ADF). Similar to the TRS symbol, the ADF is unique in the video stream. The three words of the ADF are \texttt{000h, 3FFh, and 3FFh}. When an ADF is detected, the \textit{trs\_detect} module asserts the \texttt{rx\_anc} signal during the first word of the ADF.
\end{itemize}
When an ADF is detected, the `trs_detect` module examines the word immediately after the ADF to determine if the ANC packet is an error detection and handling (EDH) packet. The word immediately following the ADF in an ANC packet is called the Data Identification word (DID) identifying the type of packet. EDH packets have a DID value of 1F4h. If an EDH packet is found, the `trs_detect` module asserts the `rx_edh` signal during the first ADF word of the packet (when `rx_anc` is asserted).

Figure 5-5 shows a timing diagram of the inputs and outputs of the `trs_detect` module. It shows how the input video stream is delayed by four clock cycles before coming out of the module. An EAV TRS symbol is shown going into the `trs_detect` module. An ancillary data packet immediately follows the TRS symbol. The signals in parenthesis have the same timing as the signals listed above them.

![Figure 5-5: trs_detect Module Timing](image)

When the `trs_detect` module is looking for TRS symbols and ANC packets, it only examines the eight most significant bits of the video word to determine if the word contains all zeros or all ones. This is to provide compatibility with 8-bit video equipment. The digital video standards state that, when checking for TRS and ADF words, the least two significant bits should be ignored.

**Autodetect Module**

The `autodetect` module implements a video standard detector. This module examines the input video stream and decoded information from the `trs_detect` module and determines the video standard. Figure 5-6 is a block diagram of the `autodetect` module.
The autodetect module is based on a finite state machine (FSM). The FSM has two main loops, the ACQUIRE loop shown in Figure 5-7 and the LOCKED loop shown in Figure 5-8. The ACQUIRE loop tries to match the input video stream with one of the standards supported by the module. Once the video standard is determined, the FSM enters the LOCKED loop where it continuously monitors the input video stream for a change in the video standard.

To determine the video standard, the FSM determines the number of words between SAV symbols. Each video standard supported by the module has a unique number of words per video line. When the first SAV is detected, the horizontal counter is cleared (state ACQ3). The horizontal counter increments every clock cycle, counting the number of words on the video line. When the next SAV symbol is found, the horizontal counter value is captured in the saved_hcnt register (state ACQ4). The value in the saved_hcnt register is compared to the horizontal position of the six subsequent SAV symbols (state ACQ7). If the number of words on a line varies from the value in the saved_hcnt register during the acquisition process, the entire process is restarted.

**Figure 5-6: autodetect Block Diagram**

The autodetect module is based on a finite state machine (FSM). The FSM has two main loops, the ACQUIRE loop shown in Figure 5-7 and the LOCKED loop shown in Figure 5-8. The ACQUIRE loop tries to match the input video stream with one of the standards supported by the module. Once the video standard is determined, the FSM enters the LOCKED loop where it continuously monitors the input video stream for a change in the video standard.

To determine the video standard, the FSM determines the number of words between SAV symbols. Each video standard supported by the module has a unique number of words per video line. When the first SAV is detected, the horizontal counter is cleared (state ACQ3). The horizontal counter increments every clock cycle, counting the number of words on the video line. When the next SAV symbol is found, the horizontal counter value is captured in the saved_hcnt register (state ACQ4). The value in the saved_hcnt register is compared to the horizontal position of the six subsequent SAV symbols (state ACQ7). If the number of words on a line varies from the value in the saved_hcnt register during the acquisition process, the entire process is restarted.
After finding the number of words per line in the input video stream, the FSM compares this word count with the word counts of each of the supported video standards. The Samples ROM contains the word counts for each supported video standard. In state ACQ6, the FSM cycles through each entry in the ROM by incrementing the iteration counter and comparing the output of the ROM with the value in the saved_hcnt register. If a match is found, the value of the iteration counter is captured in the std register and is used as the output video standard code (vid_std). If no match is found after searching all the entries in the ROM, the FSM restarts the acquisition process.

After the FSM has acquired the video standard, it moves to the LOCKED loop. In this loop, the FSM determines if the number of words on each video line in the input video stream is correct for the current video standard. It also checks for errors in XYZ words of the TRS symbols. If an incorrect number of words is found on a line or an XYZ word error occurs, an error counter increments. When the number of consecutive video lines with errors exceeds the MAX_ERRS value, the FSM returns to the ACQUIRE loop to reacquire the standard.

By requiring that errors occur on some number of consecutive lines before beginning to reacquire the video standard, the FSM provides some noise immunity for the video standard detection function. It also, however, increases the amount of time required for a new video standard to be detected.

![Diagram](Figure 5-7: autodetect FSM ACQUIRE Loop)
Flywheel Module

The flywheel module implements a flywheel video decoder. This module provides video timing in the presence of noisy, error prone, or interrupted input video data. After the autodetect module has determined the video standard of the input video stream, the flywheel synchronizes to the input video stream. Once synchronized, the flywheel generates video timing that should correspond to the timing of the input video stream. If the flywheel detects mismatches between the input video stream and its internally generated video timing on four lines over a window of eight video lines, it begins to resynchronize. The flywheel provides noise immunity by requiring a significant number of errors to occur before resynchronizing.

The flywheel generates and inserts TRS symbols into the video stream, overwriting the data in the input video stream where the TRS symbols occur. This repairs any damaged or erroneous TRS symbols appearing in the input video stream. However, this can cause multiple copies of a TRS to appear in the resulting video stream if the TRS in the input video stream does not occur at the same time as the flywheel generated TRS. To prevent this, the flywheel implements TRS blanking. The flywheel generates black-level video
values and inserts them in place of a TRS in the input video stream, if the TRS does not occur at the same time as the TRS generated by the flywheel.

The flywheel reference design takes less than two fields to synchronize to a new input video stream. Because the flywheel must look for the start of a new field to synchronize vertically, the actual time to synchronize is anywhere from a few lines to a little over one field, depending on where the first field transition occurs in the input video stream.

Figure 5-9 is a block diagram of the flywheel module. The flywheel contains four modules implementing the state machine (fly_fsm), the field functions (fly_field), the vertical functions (fly_vert), and the horizontal functions (fly_horz).

Figure 5-9: flywheel Block Diagram
Figure 5-10: flywheel FSM State Diagram Main Loop
The FSM of the flywheel begins in the UNLOCK state. It remains in this state until the autodetect module signals detection of the input video standard by asserting the std_locked signal. The FSM then attempts to synchronize to the input video stream.

To synchronize to the input video stream, the FSM first synchronizes horizontally by looking for SAV symbols in the input video stream. The FSM resets the horizontal counter located in the fly_horz module whenever an SAV is received. This is repeated until the flywheel’s generated SAV occurs at the same time as the SAV in the input video stream. When the positions of the SAV symbols match, the FSM has synchronized the horizontal counter to the input video stream.

Next, the FSM attempts to synchronize vertically by waiting for a transition of the field bit (F) in the input video stream. The fly_field module contains field transition detection logic. This logic captures the F bit from every EAV in the input video stream and compares it to the F bit from the previous EAV. When the F bit changes, the start of a new field has been found and the fly_field module asserts the new_rx_field signal, causing the state machine to load the vertical counter. The value loaded into the vertical counter is determined by the current video standard and by the value of the F bit.

After the vertical counter has been loaded, the flywheel is synchronized to the input video stream. The FSM moves to the LOCK state and asserts the locked output. Once locked, the FSM remains locked until it detects differences between its internally generated TRS symbols and the TRS symbols in the input video stream on four lines over a rolling eight-line window. When too many errors are detected, the FSM negates the locked signal and repeats the synchronization process.

Once per field, on the synchronous switching line, the FSM moves to the SWITCH1 state to check for a synchronous switch. In this state, the FSM determines if the EAV in the input video stream and the internally generated EAV occur at the same time. If they do not, the
Chapter 5: SD-SDI Video Flywheel

FSM reloads the horizontal counter to match the position of the EAV symbol in the input video stream.

The FSM contains a fail-safe mechanism to allow it to continue to generate valid video timing even if the input video stream does not contain an EAV during the synchronous switching line. Without this fail-safe mechanism, the FSM would become stuck waiting for an EAV in the input video stream. The fail-safe mechanism is found in state SWITCH6. In this state, the FSM has already determined that the input EAV is overdue. If no EAV is received before it is time to generate an SAV, the FSM gives up on the synchronous switch and proceeds to the UNLOCK state. This is considered to be a failed synchronous switching event.

During the synchronous switching lines, the flywheel is designed to pass the EAV from the input video stream directly through to the output video stream. In the case of a failed synchronous switch, the input video stream does not contain an EAV. In this case, there is no EAV symbol in the output video on the failed synchronous switching line.

The flywheel module generates an output signal called sync_switch. This signal is asserted when the current video line contains the synchronous switching point. In an SDI receiver design, this signal should be used to disable TRS filtering in the SDI receiver’s framer function. TRS filtering generally forces the framer to receive at least two consecutive TRS symbols at a new bit offset in the serial data stream before reframing to this new offset. During the synchronous switching interval, the framer should immediately reframe if an offset is detected in the TRS symbol.

When the FSM moves to the UNLOCK state due to losing synchronization with the input video stream or because of a failed synchronous switching event, the flywheel continues to generate valid TRS symbols based on its internal timing until it regains synchronization.

The flywheel module is designed to tolerate early V-bit transitions. On those lines when the V bit is permitted to be either High or Low, the flywheel ignores the V bit when comparing its internally generated TRS symbols with the TRS symbols in the input video stream. This only applies to the NTSC standards. For all PAL video standards, the V bit is always checked.

Testbench

The test_vid_decode.v and test_vid_decode.vhd files each contain a testbench for the video decoder. The testbench contains an instance of the video_decode module and video generator module called multigen.

The multigen module generates video for each of the six digital component video standards supported by the video decoder. It also supports the option of generating an early transition on the V bit for the NTSC standards.

The video generated by the multigen module is connected to the input of the video decoder. It is also delayed by an amount equal to the latency of the video decoder and then compared with the output of the video decoder to detect any differences. The comparison is only done when the video decoder indicates that it is locked to the video standard. The comparison code also ignores legal early transitions of the V bit and differences during the synchronous switching interval.

The testbench cycles through all six supported video standards, determining if the video decoder recognizes and locks to each of them. During the NTSC 4:2:2 test, the testbench also performs tests of various other features of the video decoder: tolerance of early V-bit transitions, TRS blanking, synchronous switching, and generation of TRS symbols when the input video stream is interrupted. The code in the testbench that tests these features can be moved to allow testing of these features during any of the video standards.
Reference Design Results

Table 5-6 shows the results after place and route of the various modules implemented in this chapter. Results are given for the top-level video_decode module and individually for the three main blocks that make up the video decoder. All results were obtained using the Verilog versions of the designs with Xilinx ISE version 4.1i using XST as the synthesis tool. Results using the VHDL files are not shown but are essentially identical. Virtex™-II results are for a -5 speed grade device. Spartan™-II results are for a -6 speed grade device.

The video decoder runs at the word rate of the video interface. The highest word rate required for the six supported video standards is 54 MHz. As shown in Table 5-6, this is easily achievable with Xilinx FPGAs.

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Optimized for Area</th>
<th>Optimized for Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size LUTs/FFs</td>
<td>Speed: Virtex-II</td>
</tr>
<tr>
<td>trs_detect.v</td>
<td>30/60</td>
<td>140 MHz</td>
</tr>
<tr>
<td>autodetect.v</td>
<td>124/55</td>
<td>95 MHz</td>
</tr>
<tr>
<td>flywheel.v</td>
<td>211/127</td>
<td>75 MHz</td>
</tr>
<tr>
<td>video_decode.v</td>
<td>363/235</td>
<td>75 MHz</td>
</tr>
</tbody>
</table>

Conclusions

In an SDI transmission link, digital video is normally preprocessed prior to transmission to insert error detection checkwords and other types of ancillary data. At the receiving end of the SDI link, the data is again processed to check for transmission errors and possibly to extract the ancillary data. In order to carry out these functions, a digital video processor must be synchronized to the video stream, to determine the fixed locations of these types of data packets. This is the job of the video decoder. A flywheel video decoder adds noise immunity features to the basic video decoder. Most video processors also include a video standard detector, to automatically determine the standard of the input video stream.

These two functions can be used for more than just SDI-related video processing. Most video processing features require the two things provided by these functions: the standard of the input video stream and the current horizontal and vertical position of the input video stream.

Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_sd-flywheel.zip.
**Chapter 6**

**SD-SDI Ancillary Data and EDH Processors**

**Summary**

The SMPTE 259M Serial Digital Interface (SDI) Standard describes how to transmit standard-definition digital video serially over coax cable. SDI is commonly used to transport digital video in broadcast studios and video production centers.

This chapter describes implementations of an ancillary data packet processor and an error detection and handling processor for the SDI interface.

**Introduction**

Figure 6-1 is a block diagram showing correlation between the various chapters in this volume and the elements of the SDI link.

![SDI Block Diagram and Related Application Notes](image_url)

Before transmission over an SDI link, a digital video stream is usually processed to insert error detection packets. These packets contain checkwords allowing the receiver to detect transmission errors. Ancillary data packets can also be inserted into the inactive (blanked) portions of the video to carry non-video data such as digital audio. At the receiving end of the SDI link, the digital video stream is again processed to detect transmission errors, extract ancillary data, and possibly insert additional types of ancillary data.

The functions described in this chapter, combined with the video decoder described in Chapter 5, “SD-SDI Video Flywheel,” form a processor capable of implementing the SDI...
ANC Packets

Ancillary data (ANC) packets carry non-video information in the inactive portion of the digital video stream. ANC packets can carry any type of digital information. One of the most common uses of ANC packets is to carry the digital audio portion of the program. A number of commonly used ANC packet types have been standardized. User defined ANC packet types are also allowed.

The general format of ANC packets is defined in the SMPTE 291M [Ref 1] and the ITU-R BT.1364 [Ref 2] standards. These standards also describe the spaces where ANC packets are permitted in the video frame. These standards do not define the contents of any particular ANC packet type. Standard ANC packet types are typically defined in separate documents. For example, the ANC packet type for digital audio is defined in SMPTE 272M [Ref 3].

ANC Packet Format

Figure 6-3 shows the general format of an ANC packet. There are two nearly identical formats permitted, Type 1 and Type 2. In Type 1 packets, an 8-bit identification word identifies the contents of the packet. In Type 2 packets, the identification value is a 16-bit value sent in two separate words in the packet.
ANC Packets

Every ANC packet begins with a three-word ancillary data flag (ADF). The first word of the ADF is all zeros (000HEX). The second and third words of the ADF are all ones (3FFHEX). This three-word sequence is unique in the bitstream and only occurs at the beginning of an ANC packet.

The ADF is followed by three words that indicate the type and length of the packet. All three of these words contain 8-bit values located in the least significant 8 bits (bits 7 to 0). In all three of these words, bit 8 contains an even parity bit calculated from bits 7 through 0. Bit 9 is the complement of bit 8. Requiring bit 9 to be the complement of bit 8 prevents these words from ever having the values of all ones or all zeros — values restricted from occurring anywhere in the video stream except in the timing-reference signal (TRS) symbols and in the ADF of an ANC packet.

The word immediately after the ADF contains the Data ID (DID) value identifying the type of ANC packet. Usually, bit 7 of the DID value indicates whether the packet is a Type 1 packet (b7 = 1) or a Type 2 packet (b7 = 0). However, if the 8-bit DID value is 00HEX, this indicates an undefined packet type.

For Type 2 packets, the word after the DID contains the Secondary Data ID (SDID) value. The SDID value is combined with the DID to provide a 15-bit packet identification code. The identification code is effectively only 15 bits, because bit 7 of the DID word is always 0 for Type 2 packets.

For Type 1 packets, the word after the DID contains the Data Block Number (DBN) value. Use of the DBN is optional. The DBN is used to provide a block sequence number when a group of related Type 1 packets requires a continuity numbering system. Valid DBN values range from one through 255. When the DBN is unused, then the DBN value must be 00HEX.

![ ANC Packet Format Diagram ]

**Figure 6-3:** ANC Packet Format

<table>
<thead>
<tr>
<th>Type 1 ANC Packet</th>
<th>Type 2 ANC Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADF</td>
<td>DID</td>
</tr>
<tr>
<td>ADF</td>
<td>DID</td>
</tr>
<tr>
<td>Word Contents</td>
<td>b9</td>
</tr>
<tr>
<td>Ancillary Data Flag, Word 1 (000HEX)</td>
<td>0</td>
</tr>
<tr>
<td>Ancillary Data Flag, Word 3 (3FFHEX)</td>
<td>1</td>
</tr>
<tr>
<td>Secondary DID/Data Block Number</td>
<td>P</td>
</tr>
<tr>
<td>User Data Words (0 to 255 Words)</td>
<td></td>
</tr>
<tr>
<td>Note: P is an even parity bit for bits b7 through b0 and is located in b8. Words containing a P bit in b8, also have the inverse of b8 located in b9. x299_03_041802</td>
<td></td>
</tr>
</tbody>
</table>
The word following the SDID/DBN contains the Data Count (DC). The DC indicates the number of words in the payload portion of the packet. The DC can range from zero (indicating that the payload is empty) to 255. ANC packets are restricted to a maximum of 255 payload words.

The payload section begins immediately after the DC word. The words in the payload section are called User Data Words (UDW). The definition of the UDW data is completely dependent upon the packet format. User data words are not restricted to 8-bit values. All 10 bits of each UDW can be used.

The checksum word (CS) is immediately after the last UDW. The CS provides some error detection capabilities for the ANC packet. The CS is a 9-bit checksum value computed by adding the 9-bit values (bits 8 through 0) of all words in the ANC packet from the DID word through the last UDW, and discarding any carries that result from the additions. Bit 9 of the CS word is the complement of bit 8.

The checksum only provides limited error detection capabilities. The checksum calculation does not include the MSB of any of the words in the packet, so an error in the MSB of a word might go undetected. Many ANC packet formats simply follow the general format of the ANC packet and only carry 8-bit data in the UDW words, using bit 8 as a parity bit and bit 9 as the complement of the parity bit. Some ANC packet formats include error detection or even error correction information in the payload section itself.

Non-Conforming ANC Packets

The ITU-R BT.1364 standard describes a third type of ANC packet called a non-conforming packet. Use of non-conforming ANC packets is not recommended by the standard, but is tolerated. The main advantage of a non-conforming ANC packet is that it allows for a contiguous payload of more than 255 words.

A non-conforming ANC packet is preceded by an ANC packet called a start marker packet. The start marker packet is a standard Type 1 ANC packet with a DID value of 88HEX. The DC word of the start marker packet must be zero, indicating that the start marker packet contains no user data words. The length of the start marker packet is exactly seven words long including the ADF.

Immediately after the start marker packet, the non-conforming ANC data is inserted. This data has no ADF and no predefined format. However, the non-conforming data must not include words with values in the reserved ranges 000HEX to 003HEX and 3FCHEX to 3FFHEX.

The non-conforming ANC packet ends with the ADF of a conforming ANC packet. Normally, this is an end marker packet. The end marker packet has a DID value of 84HEX. The end marker packet is similar to the start marker packet. It has a DC value of zero, no user data words, and a total packet length of seven words including the ADF. It serves simply to denote the end of a non-conforming ANC packet.

When a processor inserts a new non-conforming ANC packet, it must always insert an end marker packet following the non-conforming data. However, downstream equipment that inserts a new conforming ANC packet can replace the end marker packet with the new conforming ANC packet, since any conforming ANC packet, end marker or otherwise, serves to mark the end of the non-conforming packet.

When two non-conforming ANC packets appear back-to-back in an ANC space, a start marker packet separates them. However, an end marker packet does not occur between them.

There are a number of disadvantages to non-conforming data packets. First, there is no standard method for identifying the contents of the non-conforming packet. It is just raw
data and does not contain any identification words in standard fixed locations. Second, there is no easy procedure for marking the non-conforming packet for deletion. To delete a non-conforming packet, the entire space occupied by the non-conforming packet must be filled with one or more conforming packets marked for deletion. If, for some reason, the non-conforming space is smaller than the minimum length of a conforming ANC packet (seven words) then the non-conforming packet would have to be merged with the preceding start marker packet and the combination marked for deletion.

Figure 6-4 shows a non-conforming ANC packet.

Another Start and End Marker Protocol

The SMPTE 291M standard optionally allows the start marker packet and end marker packet to be used to identify the starting and ending locations of an ancillary data space. In this usage, the start marker packet, if used, always occurs immediately after the TRS symbol that begins the ANC space. The end marker packet is placed after the last ANC packet (conforming or non-conforming) in the data space. The use of the start marker packet at the beginning of the ANC space is optional, even if the end marker packet is used. If there is insufficient space for an end marker packet at the end of the ANC space, the end marker packet is not inserted.

The implication of this optional protocol is that any piece of equipment can consider the rest of the ANC space empty if it finds an end marker packet. A piece of equipment designed to insert new ANC packets should, therefore, always overwrite an end marker packet when inserting a new packet, regardless of whether the equipment supports the start marker/end marker protocol allowed by SMPTE 291M. Inserting a new packet after an end marker packet, rather than overwriting the end marker packet, can result in other equipment not recognizing or overwriting the packet.
8-bit Considerations

ANC packets are primarily designed to work with 10-bit equipment, but there are provisions in the standards for dealing with ANC packets generated by 8-bit equipment.

When 8-bit equipment inserts an ANC packet, the 8-bit information is inserted into the eight MSBs of the video stream and the two least significant bits (bits 1 and 0) are invalid. This limits the DID value to 6 bits. Certain DID values have been reserved to identify 8-bit packets. DID values in the range of $04_{HEX}$ to $0F_{HEX}$ are reserved for 8-bit packets. Because the two LSBs have to be ignored and a DID value of zero is reserved, there are only three valid 8-bit DID values ($04_{HEX}$, $08_{HEX}$, and $0C_{HEX}$).

The SDID value in 8-bit packets is also limited to 6 bits. An SDID value of zero is reserved for an undefined format type, so only 63 valid 8-bit SDID values are allowed.

The DC value is also limited to 6 bits. In order to allow up to 255 user data words in 8-bit ANC packets, the DC value in an 8-bit packet indicates the number of blocks of four user data words in the payload. The 8-bit equipment that generates the packet must pad the payload to an even multiple of four words, if necessary, to make the payload section end on a four-word block boundary.

ANC Packet Positioning

There are two types of spaces in the video stream where ANC packets are allowed. The first is the horizontal blanking interval of the video line. This is called the horizontal ANC space (HANC). The second space is the active portion of those video lines in the vertical blanking interval. This is called the vertical ANC space (VANC). Some ANC packet formats are always placed in the HANC area, others always in the VANC area, while some can be placed in either area. Figure 6-5 shows available ANC spaces in NTSC frames.
In a particular ANC space, ANC packets must be contiguous with each other. For example, the HANC space of a line begins with an end-of-active-video (EAV) symbol and ends with a start-of-active-video (SAV) symbol. If there are any ANC packets in a line's HANC space, the first ANC packet must begin immediately after the last word of the EAV symbol. The next ANC packet must begin immediately after the last word (CS) of the first ANC packet, and so on. If an ADF does not occur at the beginning of the HANC space, the receiver can consider the HANC space to be empty. If an ADF does not occur immediately after the last word of an ANC packet, the receiver can consider the rest of the space to be empty. An ANC packet must fit entirely within the space. It cannot overwrite the TRS symbol that marks the end of the space.
There are some exceptions to the rule requiring all ANC packets to be contiguous. For example, the EDH packet is a Type 1 ANC packet, but it always occurs immediately before the SAV (at the end of the HANC space) on a specified line in each field. The HANC space preceding the EDH packet can be empty or it can contain normal contiguous ANC packets. However, the space reserved for the EDH packet must be respected and cannot be overwritten when inserting a new ANC packet.

Some older equipment designed prior to the formalization of the ANC packet standards might not always generate contiguous ANC packets. For example, a video stream containing ANC packets inserted by older equipment can contain a few samples of blank video at the beginning of the ANC space preceding the first ANC packet. While generally not a problem when detecting and extracting packets, this is a problem for equipment designed to insert new ANC packets. ANC packets that do not start at the locations defined by the ANC standards are subject to being overwritten by equipment that inserts new ANC packets according to the rules defined by the standards.

ANC Packet Insertion Rules

The following procedure is used to locate free ANC space for insertion of a new packet.

1. Locate the beginning of an appropriate ANC space by finding a TRS symbol (EAV for HANC or SAV for VANC). For VANC space, the video line must also be in the vertical blanking interval.

2. If an ADF does not occur immediately (beginning the word after the TRS or the CS of a preceding ANC packet), then the entire remaining space is available. Any new ANC packet inserted in this space must begin immediately after the TRS symbol or the end of a preceding ANC packet.

3. If an ADF is found immediately, the DID value of the ANC packet is checked to determine if the ANC packet is an *end marker*, *start marker*, or *deletion marker*.
   a. If a *start marker* for non-conforming ANC data is found, test each word after the *start marker* until another ADF is found, then repeat step 3. If the end of the ANC space is reached before another ADF is found, repeat step 1.
   b. If an *end marker* is found, the area occupied by the *end marker* plus the remaining area in the ANC space is available.
   c. If a packet marked for deletion is found, then the area occupied by the packet marked for deletion is available. However, the ANC packet deletion rules must be obeyed.

4. If an ADF is found that is not a *start marker*, *end marker*, or *deletion marker*, then use the DC word to locate the end of the ANC packet. At the end of the packet, repeat step 2.

After free space is found, the following rules must be used to determine if a new ANC packet can be inserted.

1. The space available must be sufficient to hold the entire ANC packet. The new packet cannot overwrite the TRS symbol that ends the space. If the line is the one line per field where an EDH packet should occur, the space reserved for the EDH packet cannot be overwritten, even if there is no EDH packet present.

2. An *end marker* ANC packet can be replaced by a newly inserted conforming ANC packet or by a *start marker* packet for non-conforming ANC data.

3. If a non-conforming ANC packet is to be inserted, it must always be preceded by a *start marker* ANC packet and followed by an *end marker* ANC packet. Prior to insertion, it must be determined that there is sufficient space for the *start marker* packet, *end marker* packet, and the non-conforming ANC data.
4. If a new ANC packet replaces a packet marked for deletion, then the rules for ANC packet deletion, described in the next section, must be followed.

**ANC Packet Deletion Rules**

To delete an ANC packet, the DID word is simply changed to a value of $80_{\text{HEX}}$ and the checksum word of the packet is updated (Figure 6-6). The deleted ANC packet still has a valid DC value and occupies the same amount of space, maintaining the contiguity of packets in the ANC space.

It is possible to insert a new ANC packet in the space occupied by an ANC packet that has been marked for deletion. In doing so, the contiguity of the packets in the ANC space must be maintained. The newly inserted packet must not be larger than the deleted packet — unless that packet is the last one in the ANC space. If the inserted packet is smaller than the deleted packet, then a dummy packet must fill the remainder of the space not filled by the newly inserted packet in order to maintain contiguity. The dummy packet has a DID value of $80_{\text{HEX}}$, the same as a packet marked for deletion. The minimum size of a dummy packet is seven words. Therefore, in order to replace a packet marked for deletion with a new ANC packet that is smaller, the new packet must be at least seven words smaller than the deleted packet in order to leave room for the dummy packet.

![ANC Space with Deleted ANC Packet](image1)

![After Insertion of New Packet Overwriting Part of Deleted Packet](image2)

**Figure 6-6: Overwriting an ANC Packet Marked for Deletion**

**Synchronous Switching Considerations**

The standards recommend against inserting ANC packets into those areas of the field that can be affected by synchronous video switching. SMPTE RP-168 [Ref 4] identifies a particular line in each video field where video-switching equipment should switch between synchronous video sources. Obviously, if a video stream is switched in the middle of an ANC packet, the packet is lost. Therefore, the standards recommend certain "keep-out" areas where ANC packets are not recommended. Table 1 shows those keep-out areas for various common video standards. These areas are also noted on the NTSC ANC space diagram in Figure 6-5.
Chapter 6: SD-SDI Ancillary Data and EDH Processors

Error Detection and Handling (EDH)

The SMPTE Recommended Practice RP 165-1994 [Ref 5] and the equivalent ITU standard ITU-R BT.1304 [Ref 6] define an error detection protocol which is primarily designed for use with SDI, but can also be used with parallel digital video interfaces. The purpose of the error detection protocol is to allow detection of defective equipment and noisy connections, not to prevent loss of data due to errors. There is no retransmission protocol that allows the fields containing errors to be retransmitted.

The error detection protocol standards define a special type of ANC packet called the error detection and handling (EDH) packet. An EDH packet is generated and inserted into the video stream once per field at a specific position defined by the standards. The packet contains two cyclic redundancy code (CRC) checkwords calculated from the previous field. The EDH packet also contains three sets of error flags used to forward error detection information to help identify faulty equipment and noisy connections.

Two different CRC checkwords are calculated on a field of digital video. One CRC checkword is calculated on only the active samples of the field and the other is calculated on the full field (actually most of the field). Both checkwords are provided to allow error detection to remain intact on the active portion of the field, even when a piece of equipment inserts new data (such as ANC packets) into the inactive portion of the field without updating the full-field CRC checkword in the EDH packet. Generally, video equipment that modifies the video stream in any way should calculate new CRC checkwords and update the EDH packet. However, equipment not supporting the EDH protocol could modify the inactive portion of the video without updating the EDH packet.

Three sets of error flags are provided in the EDH packet to forward error detection information. One set is related to the active-picture CRC checkword. Another set is related to the full-field CRC checkword. The third set of error flags is used to provide error detection information based on evaluating all the ANC packet checksums in the field. This third set of flags is optional when implementing EDH packets.

### Table 1: ANC Keep-Out Areas for Synchronous Switching

<table>
<thead>
<tr>
<th>Video Standard</th>
<th>ANC Keep-Out Areas</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lines</strong></td>
<td><strong>Sample Frequency</strong></td>
</tr>
<tr>
<td>Standard NTSC</td>
<td>10 and 273</td>
</tr>
<tr>
<td>525</td>
<td>13.5 MHz</td>
</tr>
<tr>
<td>Wide-screen NTSC</td>
<td>10 and 273</td>
</tr>
<tr>
<td>525</td>
<td>18 MHz</td>
</tr>
<tr>
<td>Standard PAL</td>
<td>6 and 319</td>
</tr>
<tr>
<td>625</td>
<td>13.5 MHz</td>
</tr>
<tr>
<td>Wide-screen PAL</td>
<td>6 and 319</td>
</tr>
<tr>
<td>625</td>
<td>18 MHz</td>
</tr>
</tbody>
</table>

The SMPTE Recommended Practice RP 165-1994 [Ref 5] and the equivalent ITU standard ITU-R BT.1304 [Ref 6] define an error detection protocol which is primarily designed for use with SDI, but can also be used with parallel digital video interfaces. The purpose of the error detection protocol is to allow detection of defective equipment and noisy connections, not to prevent loss of data due to errors. There is no retransmission protocol that allows the fields containing errors to be retransmitted.

The error detection protocol standards define a special type of ANC packet called the error detection and handling (EDH) packet. An EDH packet is generated and inserted into the video stream once per field at a specific position defined by the standards. The packet contains two cyclic redundancy code (CRC) checkwords calculated from the previous field. The EDH packet also contains three sets of error flags used to forward error detection information to help identify faulty equipment and noisy connections.

Two different CRC checkwords are calculated on a field of digital video. One CRC checkword is calculated on only the active samples of the field and the other is calculated on the full field (actually most of the field). Both checkwords are provided to allow error detection to remain intact on the active portion of the field, even when a piece of equipment inserts new data (such as ANC packets) into the inactive portion of the field without updating the full-field CRC checkword in the EDH packet. Generally, video equipment that modifies the video stream in any way should calculate new CRC checkwords and update the EDH packet. However, equipment not supporting the EDH protocol could modify the inactive portion of the video without updating the EDH packet.

Three sets of error flags are provided in the EDH packet to forward error detection information. One set is related to the active-picture CRC checkword. Another set is related to the full-field CRC checkword. The third set of error flags is used to provide error detection information based on evaluating all the ANC packet checksums in the field. This third set of flags is optional when implementing EDH packets.
**CRC Checkword Calculations**

Each of the CRC checkwords is calculated over a certain set of samples in a field. The starting and ending locations of these sample sets are specifically defined in the standards. These locations vary depending upon the video standard.

The standards also define the location of the EDH packet. The EDH packet location is immediately before the SAV on a specific line in each field.

*Figure 6-8 through Figure 6-13 show the starting and ending locations for the samples sets of each CRC checkword and the EDH packet position for various video standards.*

Each CRC checkword is a 16-bit value calculated using the CRC-CCITT polynomial generation method. *Figure 6-7* shows the equation for the CRC calculation and a conceptual logic diagram of how the CRC value is calculated.

The standards require that any the values between $3F_{\text{HEX}}$ and $3F_{\text{HEX}}$ must be regarded as equaling $3F_{\text{HEX}}$ for the purposes of the CRC calculation. This only affects the CRC generator and the actual value in the video stream does not need to be modified. This is done for compatibility between 8-bit and 10-bit video equipment.

The active-picture CRC only includes those samples in the active portion of the lines indicated in the drawings. The samples in the horizontal blanking interval of each line are not included in the active-picture CRC calculation.

In the NTSC video standards, lines 20 and 283 are not included in the active-picture CRC calculation. These lines are technically in the active portion of the field; the "V" bit in the TRS symbols on those lines is zero, indicating active video lines. Some video equipment manufacturers consider these two lines to be the last lines of the vertical blanking interval. Probably due to this ambiguity, the active-picture CRC calculations do not include these two lines. See Chapter 16, “SDTV Video Pattern Generators,” for a more detailed discussion of the active/inactive status of these two lines.

The full-field CRC calculation includes all samples, both active and inactive, from the starting point to the ending point shown in *Figure 6-7*. The full-field CRC includes those active samples that are also included in the active picture CRC calculation. The full-field CRC calculation does not include the line in each field defined by SMPTE RP 168 as the synchronous switching line nor the line immediately following. This is to prevent synchronous switching events from corrupting the CRC calculation. The line immediately before the synchronous switching line contains the EDH packet for the previous field. This line is also not included in the full-field CRC calculation. The following figures are included in this chapter:

*Figure 6-7: “CRC Calculations”*
*Figure 6-8: “NTSC 13.5 MHz 4:2:2 CRC Calculations and EDH Packet Positions”*
*Figure 6-9: “NTSC 18 MHz 4:2:2 CRC Calculations and EDH Packet Position”*
*Figure 6-10: “NTSC 4:4:4:4 CRC Calculations and EDH Packet Positions”*
*Figure 6-11: “PAL 13.5 MHz 4:2:2 CRC Calculations and EDH Packet Positions”*
*Figure 6-12: “PAL 18 MHz 4:2:2 CRC Calculations and EDH Packet Positions”*
*Figure 6-13: “PAL 4:4:4:4 CRC Calculations and EDH Packet Positions”*
Chapter 6: SD-SDI Ancillary Data and EDH Processors

Figure 6-7: CRC Calculations

The odd field full-field CRC includes all samples, both active and inactive, from word 1444 on line 12 to word 1439 on line 271.

The even field full-field CRC includes all samples, both active and inactive, from word 1444 on line 275 to word 1439 on line 8.

Words 1689-1711 on line 9 contain the EDH packet for the even field.

Words 1689-1711 on line 272 contain the EDH packet for the odd field.

The odd field active picture CRC includes only active samples from word 0 on line 21 to word 1439 on line 262.

The even field active picture CRC includes only active samples from word 0 on line 284 to word 1439 on line 525.

Figure 6-8: NTSC 13.5 MHz 4:2:2 CRC Calculations and EDH Packet Positions

CRC = x^{16} + x^{12} + x^{5} + 1
Figure 6-9: NTSC 18 MHz 4:2:2 CRC Calculations and EDH Packet Position
Figure 6-10: NTSC 4:4:4:4 CRC Calculations and EDH Packet Positions
Error Detection and Handling (EDH)

Figure 6-11: PAL 13.5 MHz 4:2:2 CRC Calculations and EDH Packet Positions

- Words 1701-1723 on line 5 contain the EDH packet for the even field.
- The odd field full-field CRC includes all samples, both active and inactive, from word 1444 on line 8 to word 1439 on line 317.
- The even field full-field CRC includes all samples, both active and inactive, from word 1444 on line 321 to word 1439 on line 4.
- The odd field active picture CRC includes only active samples from word 0 on line 24 to word 1439 on line 310.
- The even field active picture CRC includes only active samples from word 0 on line 336 to word 1439 on line 622.
Figure 6-12: PAL 18 MHz 4:2:2 CRC Calculations and EDH Packet Positions

Words 2277-2299 on line 5 contain the EDH packet for the even field.

The odd field full-field CRC includes all samples, both active and inactive, from word 1924 on line 8 to word 1919 on line 317.

The even field full-field CRC includes all samples, both active and inactive, from word 1924 on line 321 to word 1919 on line 4.

The odd field active picture CRC includes only active samples from word 0 on line 24 to word 1919 on line 310.

The even field active picture CRC includes only active samples from word 0 on line 336 to word 1919 on line 622.

Words 2277-2299 on line 318 contain the EDH packet for the odd field.

The odd field full-field CRC includes all samples, both active and inactive, from word 1924 on line 8 to word 1919 on line 317.

The even field active picture CRC includes only active samples from word 0 on line 336 to word 1919 on line 622.
Error Detection and Handling (EDH)

Figure 6-13: PAL 4:4:4:4 CRC Calculations and EDH Packet Positions

- The odd field full-field CRC includes all samples, both active and inactive, from word 2884 on line 8 to word 2879 on line 317.
- Words 3429-3451 on line 5 contain the EDH packet for the even field.
- The even field full-field CRC includes all samples, both active and inactive, from word 2884 on line 321 to word 2879 on line 4.
- Words 3429-3451 on line 318 contain the EDH packet for the odd field.
- The odd field active picture CRC includes only active samples from word 0 on line 24 to word 2879 on line 310.
- The even field active picture CRC includes only active samples from word 0 on line 336 to word 2879 on line 622.
Error Flags

An EDH packet contains three sets of error flags. One set is associated with the active picture (AP) CRC, one set is associated with the full field (FF) CRC, and one is associated with ANC packet errors. Each set of error flags contains five flags as described below.

**edh — Error Detected Here**

Any piece of equipment detecting a difference between the CRC value it calculates for the previous field and the CRC checkword located in the EDH packet sets (flag = 1) the edh flag. The ancillary data edh flag is set if a checksum error is detected in at least one ANC packet in the previous field.

**eda — Error Detected Already**

This flag indicates that some upstream piece of equipment detected an error. A video device processing an EDH packet having the edh flag set by the upstream device must set the eda flag in the packet and clear the edh flag unless it, too, detects an error. (See Figure 6-14.)

**idh — Internal Error Detected Here**

Any piece of equipment can assert the idh flag to indicate that some internal processing error, unrelated to the serial video transmission, has occurred. The idh flag is provided as a signaling mechanism to allow video equipment to indicate the occurrence of internal errors. These internal errors can be anything unrelated to the actual video stream, the detection of an over-heating condition, for example.

**ida — Internal Error Detected Already**

This flag indicates that some upstream piece of equipment detected an internal error. A video device processing an EDH packet having the idh flag set by the upstream device must set the ida flag and clear the idh flag unless it, too, detects an internal error.

**ues — Unknown Error Status**

This flag indicates that the video stream was received from equipment not supporting the EDH standard. For example, a device that receives a video stream without any EDH packets can generate and insert EDH packets into the video stream. It should, however, set the ues flag in the packets it creates to signify that the video stream was not previously protected by the EDH error detection protocol.

The flag pairs, edh/eda and idh/ida, can be used to track down faulty video equipment in the serial transmission chain. For example, if the eda flag is set at any location, then it is known that some upstream piece of equipment detected an error. If the errors are occurring repeatedly, each piece of video equipment can be checked, starting with the downstream device and moving upstream to see where the eda flag changes to an edh flag. The connection or piece of equipment prior to the device asserting the edh flag is suspect.

The EDH standards allow video equipment to implement only some or all of the defined error flags. If a piece of equipment does not support a particular flag, it must clear the flag to zero.
Error Detection and Handling (EDH)

The EDH packet has the same format as a standard Type 1 ANC packet. The format of an EDH packet is shown in Figure 6-15.

Each CRC value has an associated valid bit. The standards allow implementations of the EDH protocol where only one of the two CRC values is calculated. A CRC value that is not calculated is considered to be invalid and must have its "V" bit cleared to a zero.

### EDH Packet Format

<table>
<thead>
<tr>
<th>Word Contents</th>
<th>b9</th>
<th>b8</th>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ancillary Data Flag, Word 1 (000HEX)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ancillary Data Flag, Word 2 (3FFHEX)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ancillary Data Flag, Word 3 (3FFHEX)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Data ID (1F4HEX)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Block Number</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Data Count (16 Words of User Data)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Active-picture CRC bits [5:0]</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Active-picture CRC bits [11:6]</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Active-picture CRC bits [15:12]</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Full-field CRC bits [5:0]</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Full-field CRC bits [11:6]</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Full-field CRC bits [15:12]</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Ancillary Data Error Flags</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Active-picture Error Flags</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Full-field Error Flags</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Reserved Words (7 total)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Checksum</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

Notes:
1) P is an even parity bit for bits b7 through b0 and is located in b8. Words containing a P bit in b8, also have the inverse of b8 located in b9.
2) Each CRC value has an associated valid bit (V). If the CRC value is valid, V is set to 1.
Reference Design

ANC and EDH Processor

Figure 6-16 shows a block diagram of a complete ANC and EDH processor. The video decoder block is described in Chapter 5, “SD-SDI Video Flywheel.” This video decoder processes the incoming video stream to determine the video standard and to provide timing information about the video stream such as the current horizontal and vertical positions and locations of TRS symbols, ANC packets, and EDH packets.

Figure 6-16: ANC and EDH Processor Block Diagram

The `anc_edh_processor` design implements the complete ANC and EDH processor, including the video decoder. The various blocks that make up this processor are described below.

`edh_check` Module

This section calculates CRC values for the field, finds the EDH packets in the video stream, and compares the CRC values in the EDH packets with the calculated values. It also verifies the checksum values of every ANC packet in the video stream. Based on these checks, error flags are generated and provided to the `edh_gen` module for transmission to downstream equipment in the next EDH packet. The module maintains a running count of the number of fields with errors.

If the input video stream is known not to contain EDH packets, the `receive_mode` input of this module can be negated. This prevents the module from generating errors for each missing EDH packet.

The module has error flag inputs for any EDH flags not internally generated (`idh` flag). These inputs can be asserted by another module to send error information in the EDH packets.

The module has an error counter flag enable input for each of the various error condition flags. This allows selection of which error conditions increment the counter.

The `edh_check` section captures and outputs the flags from each EDH packet. These outputs can be used to determine what error conditions are being received in the EDH packets. The module also generates and outputs error signals related to the reception of the EDH packet.
itself. The parity, format, and checksum of the EDH packet are checked and separate error flags provided to indicate each of these error conditions. Another error flag is asserted when the EDH packet is missing from the video stream.

**anc_demux Module**

This module de-multiplexes ANC packets from the video stream. The module searches for and de-multiplex up to four different ANC packet types. The module has four sets of DID/SDID inputs used to specify which ANC packet types are to be de-multiplexed. The four sets of DID/SDID inputs are compared against the DID and SDID words in the ANC packets and matching packets are de-multiplexed. The module decodes the input DID values to determine whether to also use the SDID value in the matching process. The SDID value is only used for Type 2 ANC packets.

Each DID/SDID input set has an enable input. If the enable is Low, the DID/SDID pair is not compared with the incoming ANC packets.

Also associated with each DID/SDID input pair is a del_pkt input. If this input is asserted and the corresponding enable input is asserted, packets matching the DID/SDID pair are demultiplexed and marked for deletion in the video stream. The module changes the DID value of the packet in the video stream to mark it for deletion and calculates a new checksum value for the packet. The modified packet replaces the original packet in the video stream. The modified video stream is sent out on the module's vid_out port. The demultiplexed packet is sent out on the data_out port with its original DID and checksum values.

The module has a data_out_valid signal indicating when a de-multiplexed ANC packet is being sent out the data_out port. This signal becomes asserted when the DID word is available on the data_out port and stays asserted through the checksum word. This signal is not asserted during the three words of the ADF.

In addition to the data_out_valid signal, the module also provides a number of output signals indicating what is present on the data_out port. A 2-bit match_code value indicates which one of the four input DID/SDID pairs matched the de-multiplexed packet. A set of output signals (did, sdid, dbn, dc, udw, and cs) indicate which word of the packet is currently available on the data_out port.

**anc_mux Module**

This module multiplexes new ANC packets into the video stream.

When the module is ready to accept new packet data, it asserts the pkt_in_empty output. A new packet is formed by writing the DID, SDID/DBN, and DC words into the module's internal registers. These values are 8-bit values and must be placed on the eight least significant bits of the module's data_in port. Each word is loaded into the module by asserting the associated load signal (ld_did, ld_dbn, and ld_dc). The ld_dbn signal is used to load either the SDID word or the DBN word, depending on the type of packet.

The UDW words of the packet are written by placing the 10-bit words on the data_in port, placing the word number (0 for the first word, 1 for the second word, etc.) on the udw_wr_adr port, and asserting the ld_udw input. If the packet uses 8-bit UDW words with an even parity bit in bit 8 and the complement of the parity bit in bit 9, the module can automatically calculate and insert bits 8 and 9. This is done if the calc_udw_parity signal is asserted as the words are written to the module.

After the entire packet has been written to the module, the pkt_rdy_in signal must be asserted. At the same time, the hanc_pkt and vanc_pkt inputs must also be set appropriately to indicate whether the packet is to be inserted in HANC space, VANC space, or either.
module responds immediately by negating the \texttt{pkt\_in\_empty} signal. No new information can be written to the module until the \texttt{pkt\_in\_empty} signal is reasserted.

After \texttt{pkt\_rdy\_in} is asserted, the module looks for room in the specified ANC data spaces large enough to accommodate the packet. When an appropriate space is found, the packet is inserted. The module creates the ADF and calculates and inserts a checksum word for the packet.

This module is not designed to overwrite a packet marked for deletion. The module can, however, overwrite an \textit{end marker} packet.

\textbf{edh\_gen Module}

The \texttt{edh\_check} module calculates CRC values on the incoming video stream and compares them with the CRC values in the incoming EDH packets. However, the ANC MUX and DEMUX modules can modify the video stream, invalidating the CRC values in the EDH packets.

The \texttt{edh\_gen} module calculates new CRC values and uses them, along with the error flags generated by the \texttt{edh\_check} module, to update the contents of the EDH packets in the video stream. If no EDH packets are present in the video stream, the module generates new EDH packets and inserts them at the appropriate places in the video stream.

\textbf{edh\_processor Module}

If the ANC MUX and DEMUX functions are not used, the \texttt{edh\_processor} module is an efficient EDH-only processor. It is more efficient than simply combining the \texttt{edh\_check} and \texttt{edh\_gen} modules. This design uses the same submodules that make up the \texttt{edh\_check} and \texttt{edh\_gen} modules. However, only one CRC calculation is done since the video stream is not subject to modification by the ANC MUX and DEMUX processes. The CRC calculation done on the input video stream is valid for the output video stream.

The \texttt{edh\_processor} module has the same inputs as the \texttt{edh\_check} module.

\textbf{Results}

\textbf{Table 2} shows the results after place and route of the reference design. The \texttt{anc\_edh\_processor} results include the \texttt{video\_decode} module from Chapter 5, “SD-SDI Video Flywheel,” and both the \texttt{anc\_mux} and \texttt{anc\_demux} functions. The sizes of the \texttt{anc\_mux} and \texttt{anc\_demux} modules are shown separately so that an estimate can be made of how much smaller the \texttt{anc\_edh\_processor} would be with either of them removed. The \texttt{edh\_processor} results include the size of the \texttt{video\_decode} module from Chapter 5.

The \texttt{anc\_mux} module contains a RAM to store the user data words of the ANC packet. The module contains code to allow the RAM to be implemented as either distributed RAM or block RAM. Results for the \texttt{anc\_edh\_processor} and the \texttt{anc\_mux} are given with both block RAM and distributed RAM. The module’s UDW RAM fits in one block RAM.

The Virtex\textsuperscript{TM}-II results were achieved when the design was constrained to run at 54 MHz, allowing it to support the fastest SDI bit-rate. The Spartan\textsuperscript{TM}-II results were achieved when the design was constrained to run at 27 MHz, allowing support for the most commonly used 270 Mb/s SDI bit-rate.

All results were obtained using the Verilog versions of the designs with Xilinx ISE version 4.11 using XST as the synthesis tool. Results using the VHDL files are not shown but are essentially identical. Virtex-II results are for a -5 speed grade device. Spartan-II design results are for a -6 speed grade device.
Table 2: Reference Design Results

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Virtex-II (-5 Speed Grade)</th>
<th>Spartan-II (-6 Speed Grade)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size LUTs</td>
<td>Size FFs</td>
</tr>
<tr>
<td>anc_edh_processor.v</td>
<td>1496</td>
<td>856</td>
</tr>
<tr>
<td>anc_edh_processor.v</td>
<td>1326</td>
<td>846</td>
</tr>
<tr>
<td>anc_demux.v</td>
<td>136</td>
<td>179</td>
</tr>
<tr>
<td>anc_mux.v</td>
<td>448</td>
<td>105</td>
</tr>
<tr>
<td>anc_mux.v</td>
<td>260</td>
<td>95</td>
</tr>
<tr>
<td>edh_processor.v</td>
<td>810</td>
<td>537</td>
</tr>
</tbody>
</table>

Conclusion

In an SDI transmission link, digital video is normally preprocessed prior to transmission to insert error detection checkwords and ancillary data. At the receiving end of the SDI link, the data is again processed to check for transmission errors and possibly to extract the ancillary data.

The chapter demonstrates how to implement the EDH and ANC packet processors for an SDI link using Xilinx FPGAs.

Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_sd-edh.zip.
Appendix A

Additional Reference Design Information

edh_processor Module

The edh_processor contains the video_decode module from Chapter 5 plus the modules to do CRC checking on the input video stream, ANC packet checksum checking, and outgoing EDH packet generation (Figure 6-17).

Figure 6-17: EDH Processor Block Diagram
**edh_rx Module**

The `edh_rx` module (Figure 6-18) is included in both the `edh_processor` and `anc_edh_processor` modules. It monitors the input video stream until an EDH packet is found, then it captures the various CRC checkwords and flags from the EDH packet. It also performs various checks on the received EDH packet. It asserts the `edh_missing` signal if an EDH packet is not found where one is expected. It asserts the `edh_parity_err` signal if a parity error is detected in any parity protected word of the EDH packet. It asserts the `edh_chksum_err` signal if the checksum in the received EDH packet does not match the checksum calculated by the `edh_rx` module. It asserts the `edh_format_err` signal if the DBN or DC words do not match the proper values for an EDH packet.

The `edh_rx` module has an input signal called `reg_flags`. This signal affects the timing of the received flag output ports. When the module is used with the `edh_processor`, `reg_flags` is strapped Low. When the module is used with the `anc_edh_processor`, `reg_flags` is strapped High. Figure 6-19 is the state diagram for the finite state machine in the `edh_rx` module.

![Figure 6-18: edh_rx Block Diagram](image-url)
Figure 6-19: edh_rx State Diagram
anc_rx Module

The anc_rx module (Figure 6-20) is included in both the edh_processor and anc_edh_processor modules. It calculates the checksum for every received ANC packet and compares this calculated checksum with the CS word of the ANC packet. If they do not match, an error signal is sent to the edh_gen module allowing the error to be reported in the next outgoing EDH packet.

The finite state machine (shown in the state diagram Figure 6-21) in the anc_rx module waits until an ANC packet starts. It checks the parity on the parity-protected words. It calculates the checksum and compares it to the CS word. If either a parity error or a checksum error is detected the anc_edh_local output is asserted. This signal remains asserted until the next EDH packet has been sent — as signaled by the edh_packet signal from the edh_gen module going High then Low.

Figure 6-20: anc_rx Block Diagram
**edh_loc Module**

The `edh_loc` module (Figure 6-22) locates the position in each field where the EDH packet should occur. The `edh_rx` module uses this signal to determine if the EDH packet is present or missing in the input video stream. The `edh_gen` module uses this signal to determine when it is time to send an EDH packet.

**Figure 6-21: anc_rx State Diagram**
edh_crc Module

The edh_crc module (Figure 6-23) calculates the active picture and full-field CRC checkwords for each field of the video stream. In the anc_edh_processor design, this module is instanced twice. One instance calculates the CRC checkwords for the input video stream for comparison against the checkwords in the EDH packet. The second instance calculates the CRC checkwords for the output video stream for the edh_gen module to insert into the EDH packet. In the edh_processor design, only one instance of edh_crc is required because there is no ANC processing to modify the video stream. So, the CRC values calculated on the input video stream are valid for the output video stream as well.

The ITU and SMPTE standards require that any video word with 1s in all eight MSBs must also have 1s in the two LSBs for the purposes of CRC calculation. This makes the CRC calculation generate the same checkword regardless of whether the video stream was generated by 8-bit or 10-bit equipment. This requirement only applies to the input of the CRC generator and does not affect the actual words in the video stream.

The Valid Flag Logic section generates signals indicating whether the CRC checkwords are valid. The checkwords are considered valid as long as the video decoder's locked signal does not rise during the time when checkword is being calculated. A rising edge of the locked signal indicates a change in synchronization between the video decoder and the input video stream. In this case, any CRC checkword being calculated at the time of the rising edge of the locked signal was probably not calculated over the correct number of samples and should be considered invalid.

The actual CRC calculations are done in the edh_crc16 modules instanced in the edh_crc module. Each edh_crc16 module computes a 16-bit CRC value by combining the 10-bit video input word with the current 16-bit CRC value stored in the associated CRC register. At the beginning of a CRC region, the CRC register is cleared to zero to start a fresh CRC calculation. Load enable signals from the CRC Region Logic block control each CRC register to include into the CRC calculation only the appropriate video words.

The equations in the edh_crc16 module have been optimized for the four-input LUT structure of Xilinx FPGAs.
anc_mux Module

The anc_mux module (Figure 6-24) multiplexes new ANC packets into the video stream. The module contains two submodules, anc_pkt_gen and anc_insert. The anc_pkt_gen module accepts externally supplied raw ANC data, generates a properly formatted ANC packet from the raw data, and provides the formatted ANC packet to the anc_insert module. The anc_insert module searches the video stream for an appropriate ANC space large enough to hold the packet generated by anc_pkt_gen. When an appropriate space is found, the packet is transferred from anc_pkt_gen and inserted into the video stream.

The anc_insert module overwrites an end-marker ANC packet if one exists in the ANC space. However, it is not designed to overwrite an ANC packet marked for deletion.

Because the anc_insert module overwrites end-marker packets, it must tell the anc_pkt_gen module to begin sending the packet (by asserting send_pkt) before it can determine whether the new packet can overwrite the current packet. This determination is not made until anc_insert examines the DID word of the packet being overwritten to determine if it is an end-marker packet. If the packet cannot be overwritten, the anc_insert module asserts the abort_pkt signal, causing anc_pkt_gen to abort the packet and resend the same packet the next time send_pkt is asserted.

All the video timing signals from the video decoder pass through the anc_mux module, but are not registered. The current implementation of the anc_mux module does not add any

Figure 6-23: edh_crc Block Diagram
cycles of latency to the video signal so there is no need to delay the video timing signals. However, future versions of the \textit{anc_mux} module can add cycles of latency to the video signal. Passing the video timing signals through the \textit{anc_mux} module allows them to be delayed appropriately in the future to match the video without having to change any upper-level signal connections.

\textbf{anc_insert Module}

The \textit{anc_insert} module (Figure 6-25) multiplexes ANC packets generated by the \textit{anc_pkt_gen} module into the video stream.
A state machine (shown in the state diagram Figure 6-26) searches for EAV and SAV symbols in the video stream. An EAV symbol marks the beginning of HANC space and an SAV symbol marks the beginning of VANC space if the line is in the vertical blanking interval. If the \texttt{anc_pkt_gen} module asserts the \texttt{pkt_rdy_in} signal, the state machine determines if the packet can be inserted immediately after the EAV or SAV symbol. The packet can be inserted if there is no ANC packet already in the video stream immediately after the EAV or SAV. If the \texttt{pkt_rdy_in} signal becomes asserted after the state machine finds free ANC space, but before the end of the space, the ANC packet cannot be inserted. Doing so would violate the requirement for contiguity of the ANC packets.

If an ANC packet is found in the video stream, it is overwritten if it is an end-marker packet. Otherwise, the state machine examines the DC word of the packet to determine the length of the packet and waits until the end of the packet. If another ANC packet immediately follows, this procedure is repeated. If not, the state machine inserts the new packet if there is enough space remaining in the ANC space.

The state machine tells the \texttt{anc_pkt_gen} module to begin sending the new packet if there is chance that it can be inserted. If the state machine determines that the packet cannot be inserted, then the module asserts the \texttt{abort_pkt} signal to cancel the packet. The packet is aborted if the state machine finds an existing ANC packet in the video stream that is not an end-marker packet. The state machine also cancels the packet if the ANC space is part of the synchronous switching interval. The switching signal cannot be generated soon enough to determine start of the synchronous switching interval until the clock cycle after \texttt{send_pkt} signal must be asserted to the \texttt{anc_pkt_gen} module. The abort mechanism is used to cancel the packet if the switching signal is asserted.
Figure 6-26: anc_insert State Diagram
anc_pkt_gen Module

The anc_pkt_gen module (Figure 6-27) generates an ANC packet from raw ANC data.

An external processor or another module writes the raw ANC data into the anc_pkt_gen module. The external processor can begin loading the ANC data as soon as the anc_pkt_gen module asserts the pkt_in_empty signal. The external processor must provide 8-bit DID, DBN/SDID, and DC values. These values must be placed onto the LS 8 bits of the data_in port and the appropriate load signal (ld_did, ld_dbn, or ld_dc) must be asserted at the same time until the rising edge of the clock. The ld_dbn signal is used to load either the DBN or the SDID value. The user data words, if any, are written into the module one at a time. To write the UDWs, each 10-bit UDW is placed on the data_in port, the word number of the UDW is placed on the udw_wr_adr port (0 for the first word, 1 for the second word, etc.), and the ld_udw signal is asserted until the rising edge of the clock. The DID, DBN/SDID, DC, and UDW words can be written to the module in any order.

If the ANC packet format requires bit 8 of every UDW to be an even parity bit and bit 9 to be the complement of bit 8, the anc_pkt_gen module can calculate bits 8 and 9. The eight LSBs of each UDW are placed on the eight LSBs of the data_in port, and the calc_udw_parity signal is asserted by the external processor.

When all the data for the packet has been written to the module, the external processor must assert the pkt_rdy_in signal for one clock cycle. During the same cycle, the processor must also indicate whether the packet is to be inserted into HANC space by asserting the hanc_pkt input or VANC space by asserting the vanc_pkt input. If both hanc_pkt and vanc_pkt are asserted, the packet is inserted into the first ANC space that has sufficient room for the packet. The hanc_pkt and vanc_pkt signals are captured in a register in the anc_mux module when pkt_rdy_in is asserted and sent to the anc_insert module. The anc_pkt_gen module does not use these signals.

The anc_pkt_gen module stores the UDW values in a RAM. This RAM can be implemented in either distributed RAM or block RAM. The source files for this module contain code to allow either distributed RAM or block RAM to be inferred by the synthesizer. In the Verilog code, the following statement:

`define UDW_BLOCK_RAM
causes block RAM to be inferred if present. If this statement is commented out or deleted, distributed RAM is inferred. In the VHDL file, the two sections of code exist in the file with one of them commented out.

Different code is used to infer the two types of RAM rather then using synthesis options, because a common code base would infer a dual-port distributed RAM. Only a single-port distributed RAM is required and this is half the size of a dual-port distributed RAM.

The UDW RAM uses 2560 bits, 256 words times 10 bits each, to support the maximum number of UDW words allowed in an ANC packet. If an application always creates ANC packets with less than the maximum number of UDW words, the size of the UDW RAM could be made smaller, saving space if distributed RAM is used. Parameters or generics at the beginning of the module control the size of the UDW RAM and the width of the address bus or busses supplied to the RAM and to the module.

Only one 2560-bit UDW RAM fits in the 4096-bit block RAMs of the Virtex and Spartan-II families. However, the larger block RAM in Virtex-II family devices could hold multiple UDW RAMs. The current design does not allow multiple ANC packets to be written to the anc_pkt_gen module. Consequently, only one ANC packet can be inserted into any ANC space. However, the module could be modified to allow multiple ANC packets to be stored using a FIFO technique. Some modifications to the state machine would be required to allow the module to insert multiple consecutive ANC packets into the same ANC space.

There is another way to insert multiple ANC packets in an ANC space that requires no modification to the existing anc_mux design. Two or more anc_mux modules can be cascaded. The first anc_mux module inserts its ANC packet into the first available ANC space. The second anc_mux module inserts its ANC packet immediately after the ANC packet inserted by the first module, and so on.

Cascaded anc_mux modules inherently provide priority to the first anc_mux module. Consider what happens if the first anc_mux inserts its ANC packet, but there is no room in the same ANC space for the second anc_mux to insert its ANC packet. If a new ANC packet is written into the first anc_mux before the next ANC space occurs, then the first anc_mux inserts its new ANC packet into the video stream before the second anc_mux has a chance to insert its ANC packet. If such behavior is not desired, the pkt_rdy_in signals of the various anc_mux modules need to be carefully controlled to prevent the first anc_mux from always taking priority.

The current anc_mux module design provides a purely combinatorial path for the video signal. There is no input or output register on the video path. Cascading anc_mux modules increases the number of logic levels on the video path, making it more difficult to meet timing. A pipeline register can be required between cascaded anc_mux modules in order to meet timing. Be sure to delay all video timing signals as well, if a pipeline register is inserted into the video path to keep them synchronized.

Figure 6-28 shows the state diagram for the finite state machine in the anc_pkt_gen module.
Figure 6-28: anc_pkt_gen State Diagram
**anc_demux and anc_extract Modules**

The `anc_demux` module searches for certain types of ANC packets and demultiplexes them from the video stream. When a matching ANC packet is found, the module provides the ANC packet data to a separate output port, `data_out`. The module also provides a number of signals indicating when the ANC packet information is available on the `data_out` port and which word of the ANC packet is currently available. These signals can be used by another module or external processor to store or process the demultiplexed ANC packet. The demultiplexed ANC packet can either be left intact in the video stream or it can be marked for deletion.

The `anc_demux` module is actually a wrapper around the `anc_extract` module. The `anc_extract` module (Figure 6-29) does all the work of searching for and demultiplexing ANC packets. The `anc_extract` module introduces three clock cycles of latency to the video stream. The `anc_demux` module delays all the video timing signals by three clock cycles to match the delay added to the video stream by `anc_extract`.

The `anc_demux` module can search for and demultiplex up to four different ANC packet types. There are four sets of inputs allowing the ANC packet types to be specified. Each set contains a DID value, a SDID value, an enable signal, and a `del_pkt` signal. If the DID value indicates a Type 1 ANC packet, the SDID value is ignored. If the DID value indicates a Type 2 ANC packet, the SDID value is also used to find matching packets. If the enable signal for the set is Low, the DID and SDID input set are not used by the module when searching for matching ANC packets. If the `del_pkt` input is asserted High, any matching packets are marked for deletion as they are demultiplexed.

![Figure 6-29: anc_extract Block Diagram](image)

The `anc_demux` module can search for and demultiplex up to four different ANC packet types. There are four sets of inputs allowing the ANC packet types to be specified. Each set contains a DID value, a SDID value, an enable signal, and a `del_pkt` signal. If the DID value indicates a Type 1 ANC packet, the SDID value is ignored. If the DID value indicates a Type 2 ANC packet, the SDID value is also used to find matching packets. If the enable signal for the set is Low, the DID and SDID input set are not used by the module when searching for matching ANC packets. If the `del_pkt` input is asserted High, any matching packets are marked for deletion as they are demultiplexed.
The `anc_demux` module provides the demultiplexed data on the `data_out` port. The `data_out_valid` signal is asserted when the `data_out` port contains valid ANC data. This signal is asserted starting with the DID word and stays asserted through the CS word of the packet. It is not asserted for the three words of the ADF that marks the beginning of the packet. The `did`, `sdid`, `dbn`, `dc`, `udw`, and `cs` outputs of the module are asserted when the corresponding parts of the demultiplexed ANC packet are present on the `data_out` port. The module also places a value on the `match_code` output port to indicate which of the four input DID/SDID sets matched the packet: "00" for the A set, "01" for the B set, "10" for the C set, and "11" for the D set.

The `anc_extract` module calculates a new checksum for the ANC packet and inserts it into the ANC packet in the video stream. This is required because the module might modify the ANC packet if the packet is marked for deletion. The newly calculated checksum always replaces the CS word in every ANC packet, regardless of whether the packet is marked for deletion or not. If this behavior is not desired, the module must be modified to only replace the CS word in packets it marks for deletion.

If an application requires the `anc_demux` module to search for and demultiplex less than four different ANC packet types, the unused DID/SDID input sets must be disabled using the enable signal associated with each pair. However, the decoders and logic associated with the unused sets are still synthesized. Unused input sets can be removed from the `anc_demux` module code to save space.

If an application requires demultiplexing of more than four different ANC packet types, the `anc_demux` module can be modified to provide more DID/SDID input sets. Or, multiple `anc_demux` modules can be cascaded. Unlike the `anc_mux` module, the `anc_demux` has pipeline registers, so cascading `anc_demux` modules should not present any timing problems.

Figure 6-30 shows the state diagram for the finite state machine for the `anc_extract` module.
anc_edh_processor Module

The **anc_edh_processor** implements a complete ANC and EDH processor design. It implements the video decoding method discussed in Chapter 5, input EDH packet processing, ANC multiplexing, ANC demultiplexing, and output EDH packet generation.
The design can be easily modified to remove either the \texttt{ancmux} or \texttt{ancdemux} modules or to cascade multiple \texttt{ancmux} or \texttt{ancdemux} modules.

\textbf{edh\_gen Module}

The \texttt{edh\_gen} module (Figure 6-31) is used by the \texttt{anc\_edh\_processor} module. It calculates the CRC checkwords and generates a new EDH packet that is inserted into the outgoing video stream.

The \texttt{edh\_gen} module instances the \texttt{edh\_crc} module to calculate the CRC checkwords and the \texttt{edh\_tx} module to generate the EDH packets. It also provides an output register for the video path and the various video timing signals.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{edh_gen_block_diagram.png}
\caption{edh\_gen Block Diagram}
\end{figure}

\textbf{edh\_tx Module}

The \texttt{edh\_tx} module (Figure 6-32) generates new EDH packets and inserts them into the outgoing video stream. This module is used directly by the \texttt{edh\_processor} design. The \texttt{anc\_edh\_processor} instances an \texttt{edh\_gen} module. The \texttt{edh\_gen} module instances the \texttt{edh\_tx} module.

The \texttt{edh\_tx} module’s finite state machine (state diagram shown in Figure 6-33) waits for the \texttt{edh\_next} signal to be asserted. This signal is usually generated by an \texttt{edh\_loc} module and signals the \texttt{edh\_tx} module to output the first word of the EDH packet during the next clock cycle. The FSM contains a state for each word of the EDH packet and controls a big MUX to output the words of the EDH packet in the correct sequence.
Figure 6-32: edh_tx Block Diagram

Figure 6-33: edh_tx State Diagram
Chapter 7

Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor

Summary

The standard-definition serial digital interface (SD-SDI) standard is used to transport digital video serially over video coax cable. This standard is used to connect video equipment in broadcast studios and video production centers. The error detection and handling (EDH) protocol is an optional but commonly used addition to the SD-SDI standard. This protocol allows an SD-SDI receiver to verify that each field of video is received correctly.

Chapter 6, “SD-SDI Ancillary Data and EDH Processors” describes how to implement an EDH processor using the programmable logic of Xilinx FPGA devices. However, due to the complexity of the protocol, the EDH processor described in this chapter consumes a fairly large amount of FPGA resources.

This EDH processor design cuts the size of the EDH processor by 33% to 50%, depending on the feature set, by moving some of the logic into software running on a PicoBlaze™ processor [Ref 1]. This software-based approach also results in a more flexible design. As a result of this flexibility, several functions often required for high-definition SDI (HD-SDI) interfaces also can be implemented in the same hardware simply by adding a small amount of additional software.

Introduction

The SD-SDI EDH protocol is defined by SMPTE RP 165-1994 and the equivalent ITU standard ITU-R BT.1304 [Ref 2] [Ref 3]. The SD-SDI transmitter calculates two CRC values for each video field and places them in an EDH packet. The EDH packet is inserted at a specific location in each field of video. The SD-SDI receiver also generates the same two CRC values for each field and compares them against the CRC values in the received EDH packet to determine if each field of video is received without errors.

The EDH protocol does not provide for error correction, only error detection. Also, there is no mechanism in SD-SDI to allow a field containing errors to be retransmitted. EDH error detection is used primarily to assist in identifying faulty equipment in a video chain so that it can be quickly replaced or repaired.

Chapter 6 contains a thorough explanation of the EDH protocol. It includes diagrams showing how the two CRC values are calculated and where the EDH packet is located for each of six different SD video formats: NTSC 4:2:2, PAL 4:2:2, NTSC 4:2:2 wide-screen, PAL 4:2:2 wide-screen, NTSC 4:4:4:4, and PAL 4:4:4:4. Refer to Chapter 6 for details on how the
CRC values are calculated, the format and location of the EDH packets, and the operation of the EDH packet error flags.

The EDH processor design also implements several other video-related functions—many for both SD and HD video. The EDH processor produces various video timing signals. It generates the current horizontal position and vertical line number of the video stream. It identifies the format of the video for the six supported SD video formats and for the most commonly used HD video formats. Finally, it can identify when the current video position is in the synchronous switching interval as defined by SMPTE RP168-2002.

EDH Processor Description

Processor Selection

The Xilinx PicoBlaze processor is used as the basis for this EDH processor design. The EDH processor also can be implemented with the embedded PowerPC™ processors in Virtex™-II Pro and some Virtex-4 devices or with the MicroBlaze™ processor. However, the PicoBlaze processor is ideally suited to this application. Because multiple SDI channels are often implemented in one FPGA, the small size of the PicoBlaze processor allows the implementation of many PicoBlaze based EDH processors in the same FPGA.

There are some time-critical aspects to the EDH protocol. A PicoBlaze processor running at 27 MHz is just fast enough to handle these time-critical aspects. The simple instruction timing of the PicoBlaze processor (two clock cycles per instruction) makes it easy to determine if the PicoBlaze processor can meet the timing requirements.

The kcpsm3 version of the PicoBlaze processor is used in this chapter and is supported by the Virtex-II, Virtex-II Pro, Virtex-4, and Spartan™-3 FPGA families.

Using Software to Minimize EDH Processing

EDH processing involves the following processes:

1. Identify the format of the video. The position of the EDH packet and the details of which words are included in the CRC calculation are different for each video format.
2. Synchronize to the video stream so that the current horizontal and vertical positions are known.
3. Generate the two CRC values for each field of video.
4. If implementing receiver EDH checking, locate and capture the EDH packet from the incoming video stream, and compare the CRC values to detect errors.
5. Generate an output EDH packet and insert it into the correct position in the video stream. If an EDH packet already exists in the video, the status from the error detection process (#4 above) is combined with the error flags in the existing EDH packet to create the error flags for the output EDH packet. (See the Chapter 6 explanation of EDH error flags for details.)

The EDH processor in Chapter 6 has dedicated logic for each of these processes. However, it is possible to implement many of these processes sequentially in software rather than having parallel hardware capable of implementing all of these processes simultaneously. This is one of the keys to reducing the size of the EDH processor function. The PicoBlaze processor can sequentially identify the video format, then lock to the video stream, and then begin EDH processing rather than doing all of these processes in parallel.
The implementation of the horizontal and vertical counters and the process of identifying the location of the EDH packets provide additional examples of how a software-based EDH processor is more efficient.

The vertical counter initially must be synchronized to the incoming video stream. The V bit (located in the XYZ word of the EAV and SAV of each line) rises and falls on specific lines, unique to each video format. Once the video format is known, the EDH processor waits for a V-bit transition and then loads the proper value into the vertical counter.

After it has been initialized, the video counter is incremented once per video line when the EAV occurs. The vertical counter must roll over to line one when the maximum line number is exceeded (the maximum line number is specific to each video format).

A comparison is done between the vertical count and the known vertical location of the EDH packet (yet another video format specific value). The vertical count also is checked to determine if the current line contains the synchronous switching interval and if the CRC calculations are to be started or stopped.

In Chapter 6, these functionalities are implemented in hardware, requiring a 10-bit counter, as many as seven 10-bit comparators, and multiple lookup tables (LUTs) to provide the video format-specific values (vertical counter initialization value, maximum line count, EDH packet vertical position, synchronous switching interval position, and CRC calculation start and end lines for two different CRC calculations).

The PicoBlaze processor implements these functionalities in software. Two bytes of scratchpad memory are used to keep the current 10-bit vertical line count. The software initializes the vertical count appropriately when a transition of the V bit occurs after the video format is detected. The software determines when to roll the counter over to one. The software compares the current vertical count to the vertical position of the EDH packet, synchronous switching interval, and CRC calculation start and end lines.

In the horizontal direction, the PicoBlaze processor cannot replace the horizontal counter or the comparators because each must run at 27 MHz, but the PicoBlaze processor can replace the LUTs by providing the initialization and comparison values.

SD Functions

The PicoBlaze based EDH processor described in this chapter implements the following functions for standard-definition (SD) video streams:

- Video format detection – this EDH processor can detect the following SD video formats:
  - NTSC 13.5 MHz 4:2:2 (4:3 aspect ratio)
  - NTSC 18 MHz 4:2:2 (16:9 aspect ratio)
  - NTSC 4:4:4
  - PAL 13.5 MHz 4:2:2 (4:3 aspect ratio)
  - PAL 18 MHz 4:2:2 (16:9 aspect ratio)
  - PAL 4:4:4

- Video timing – the EDH processor synchronizes to the video stream, asserts a locked output signal when locked to the video stream, and generates the following video timing signals:
  - Current vertical line number
  - Current horizontal position
Chapter 7: Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor

- Field indicator (F), vertical blanking interval indicator (V), and horizontal blanking interval indicator (H) signals captured from the XYZ word of the EAV and SAV on each line
- Timing reference signal (TRS) identifying the occurrence of each EAV and SAV
- Synchronous switching interval indication signal that can be used by the SDI framer to disable erroneous TRS filtering on the line containing the synchronous switching interval

- EDH error checking – the EDH processor can detect the following errors:
  - CRC differences between calculated CRC values and the CRC values in the received EDH packet
  - EDA error bits (Error Detected Already by an upstream device)
  - IDA and IDH error flag handling
  - EDH packet missing
  - EDH packet parity error
  - EDH packet checksum error
  - EDH packet format error

- EDH packet generation – the EDH processor can generate a new EDH packet and insert it into the video stream or update an existing EDH packet with new CRC and error flag values.

HD Functions

Optionally, the PicoBlaze EDH processor can implement several functions for high-definition (HD) video streams. These features require only a minimal amount of additional hardware, but can replace several modules from the Xilinx HD-SDI reference designs, providing significant FPGA resource savings when the EDH processor is used in a multi-rate HD/SD-SDI interface.

The EDH processor module is configured by several parameters (Verilog) or generics (VHDL) that determine whether the HD features are supported by the hardware. Additionally, different software is used to support the HD functions. The HD software also contains all of the SD functionality, so there is no need to reload the EDH processor software when switching between SD and HD modes.

The HD functions supported by the EDH processor are:

- Video format detection. The HD video formats supported are:
  - SMPTE 295M 1080i 25 Hz
  - SMPTE 274M 1080i 30 Hz
  - SMPTE 274M 1080sF 30 Hz
  - SMPTE 274M 1080i 25 Hz
  - SMPTE 274M 1080sF 25 Hz
  - SMPTE 274M 1080sF 24 Hz
  - SMPTE 274M 1080p 30 Hz
  - SMPTE 274M 1080p 25 Hz
  - SMPTE 274M 1080p 24 Hz
  - SMPTE 296M 720p 60 Hz

The software can be modified to support additional HD video formats.
• Video timing. The EDH processor synchronizes to the video stream, asserts a locked output signal when locked to the video stream, and generates the following video timing signals:
  ♦ Current vertical line (suitable for insertion into each video line after the EAV as required by the SMPTE 292M HD-SDI standard)
  ♦ Current horizontal position
  ♦ F, V, and H signals captured from the EAV and SAV of each line
  ♦ TRS signal identifying the occurrence of each EAV and SAV
  ♦ Synchronous switching interval signal that can be used to configure the SDI framer for quick resynchronization on the line containing the synchronous switching interval

Reference Design

There are three different top-level designs of the PicoBlaze EDH processor with different capabilities. Each of the three designs can be configured through parameters or generics to include support for the optional HD functions. All of the source code is available in both Verilog and VHDL.

The three versions of the top-level design are:

• rxtx: This fully featured version supports both EDH error checking using EDH packets from the input video stream and updating of the EDH packet in the output video stream with corrected CRC values, updated error flags, and updated checksum. This version can be used in an SDI receiver or an SDI transmitter, but typically is used between a receiver and transmitter where the input video stream is checked for errors and the EDH packet is updated for retransmission. If EDH packets are not present in the input video stream, new EDH packets are generated.

• rxonly: This version supports only the receive side functionality. It checks for EDH errors by using the EDH packet in the input video stream. However, it does not update EDH packets nor does it generate new EDH packets.

• txonly: This version supports new EDH packet generation and insertion. It does not check for EDH errors in the input video stream. It also does not update existing EDH packets. Any EDH packets in the input video stream are overwritten with new EDH packets. This version typically is used in situations where a video source, such as a video test pattern generator or a video camera, is driving an SDI transmitter.

I/O Ports

The input and output ports of the EDH processor are listed in Table 7-1. The table also indicates which versions of the EDH processor include the port.

Unless otherwise noted, all signals are synchronous to the clk clock signal. Signals marked by * are synchronous to the cpuck clock. Those marked by ** can be synchronous to either the clk or the cpuck signal, depending on the current mode of operation. These signals are all related to the packet error flags and must be set up within 20 cpuck cycles after the occurrence of the EAV prior to the EDH packet position. They must remain stable until the SAV immediately after the EDH packet position.
## Table 7-1: I/O Signals

<table>
<thead>
<tr>
<th>Name</th>
<th>I/O</th>
<th>Width</th>
<th>rxtx</th>
<th>rxonly</th>
<th>txonly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clocks and Reset</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Video clock. Must run at the video word rate or an integer multiple of it.</td>
</tr>
<tr>
<td>cpuclk</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PicoBlaze clock. Must be at least as fast as the fastest supported SD video word rate – typically at least 27 MHz.</td>
</tr>
<tr>
<td>ce</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Clock enable for the clk signal. Does not affect cpuclk.</td>
</tr>
<tr>
<td>rst</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Asynchronous reset</td>
</tr>
<tr>
<td><strong>Control Inputs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>force_crc_err</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Forces an error in the output EDH packet.</td>
</tr>
<tr>
<td>enable_ues</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>When enable_ues is High, the UES bits in the AP and FF flag words are asserted in the output EDH packet if there are no EDH packets in the input video stream. When enable_ues is Low, the UES flags from the input packet are passed unchanged to the output packet.</td>
</tr>
<tr>
<td>en_sync_switch</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>When en_sync_switch is High, the video synchronizer function of the EDH processor resynchronizes immediately on the first EAV detected after the synchronous switching interval. Otherwise, the normal TRS error filtering process is used. This input does not affect the sync_switch output.</td>
</tr>
<tr>
<td><strong>Video I/O Ports</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vid_in</td>
<td>In</td>
<td>10</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Video input port. In HD mode, either the Y or C channel can be connected.</td>
</tr>
<tr>
<td>vid_out</td>
<td>Out</td>
<td>10</td>
<td>Yes</td>
<td>Yes(1)</td>
<td>Yes</td>
<td>Video output port. For the rxtx and txonly versions of the EDH processor, the video stream out of the vid_out port is modified with new or updated EDH packets. For the rxonly version, the output video stream is identical to the input video stream on the vid_in port, just delayed. For all versions, the video on the vid_out port is delayed four video clock cycles from the vid_in port.</td>
</tr>
<tr>
<td><strong>Video Format Signals</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hd_sd</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>This output is Low when the input video is SD or High when the input video is HD. This signal is only valid when std_locked is High.</td>
</tr>
</tbody>
</table>

(1) Valid only when std_locked is high.
Table 7-1:  I/O Signals (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>I/O</th>
<th>Width</th>
<th>rtx</th>
<th>rxonly</th>
<th>txonly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>std</td>
<td>Out</td>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>This output port indicates which video format is detected. If hd_sd is Low (SD mode), these bits are encoded as: 000: NTSC 4:2:2 001: not used 010: NTSC 4:2:2 widescreen 011: NTSC 4:4:4 100: PAL 4:2:2 101: not used 110: PAL 4:2:2 widescreen 111: PAL 4:4:4:4 If hd_sd is High (HD mode), these bits are encoded as: 000: SMPTE 295M 1080i 25 Hz 001: SMPTE 274M 1080i or 1080sF 30 Hz 010: SMPTE 274M 1080i or 1080sF 25 Hz 011: SMPTE 274M 1080p 30 Hz 100: SMPTE 274M 1080p 25 Hz 101: SMPTE 274M 1080p 24 Hz 110: SMPTE 296M 720p 60 Hz 111: SMPTE 274M 1080sF 24 Hz This output port is only valid when std_locked is High.</td>
</tr>
<tr>
<td>std_locked</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>This output is asserted High when the video synchronizer locks to the video stream and the video format detector detects a supported video format.</td>
</tr>
</tbody>
</table>

Video Timing Signals

<table>
<thead>
<tr>
<th>Name</th>
<th>I/O</th>
<th>Width</th>
<th>rtx</th>
<th>rxonly</th>
<th>txonly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>No(2)</td>
<td>Field indicator signal. This signal is the captured F bit from the EAV XYZ word of the current line.</td>
</tr>
<tr>
<td>v</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>No(2)</td>
<td>This output is High during the vertical blanking interval. This signal is the captured V bit from the EAV XYZ word of the current line.</td>
</tr>
<tr>
<td>h</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>No(2)</td>
<td>This output is High during the horizontal blanking interval.</td>
</tr>
<tr>
<td>trs</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>No(2)</td>
<td>This output is High when any of the four words of an EAV or SAV are present on the vid_out port.</td>
</tr>
<tr>
<td>h_pos</td>
<td>Out</td>
<td>12</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>This output port indicates the horizontal position (word number) of the video word present on the vid_out port.</td>
</tr>
<tr>
<td>v_pos</td>
<td>Out</td>
<td>10(3)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>This output port indicates the current video line number. It normally updates when the first word of the EAV is present on the vid_out port. However, when the EDH processor is synchronizing to a video stream, this output port can be updated at any time as part of the synchronization process.</td>
</tr>
<tr>
<td>sync_switch</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>This output is High during the synchronous switching interval. The actual synchronous switching interval is defined as a certain part of the active portion of particular video lines. However, the EDH processor asserts sync_switch during the entire active portion of the line. This is more useful for controlling SDI framers that must resynchronize immediately upon the first occurrence of an EAV on the synchronous switching line.</td>
</tr>
</tbody>
</table>
## Chapter 7: Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor

### Table 7-1: I/O Signals (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>I/O</th>
<th>Width</th>
<th>rxtx</th>
<th>rxonly</th>
<th>txonly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EDH Flags Signals</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>anc_edh_local**</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>This input is inserted into the EDH bit of the ANC flag word in the outgoing EDH packet.</td>
</tr>
<tr>
<td>anc_idh_local**</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>This input is inserted into the IDH bit of the ANC flag word in the outgoing EDH packet.</td>
</tr>
<tr>
<td>anc_ues_local**</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>This input is inserted into the UES bit of the ANC flag word in the outgoing EDH packet.</td>
</tr>
<tr>
<td>ap_idh_local**</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>This input is inserted into the IDH bit of the AP flag word in the outgoing EDH packet.</td>
</tr>
<tr>
<td>ff_idh_local**</td>
<td>In</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>This input is inserted into the IDH bit of the FF flag word in the outgoing EDH packet.</td>
</tr>
<tr>
<td>anc_flags(4)**</td>
<td>In</td>
<td>5</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>For the txonly version of the EDH processor, this port controls the five flag bits in the ANC flag word of the outgoing EDH packet.</td>
</tr>
<tr>
<td>ap_flags(4)**</td>
<td>In</td>
<td>5</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>For the txonly version of the EDH processor, this port controls the five flag bits in the AP flag word of the outgoing EDH packet.</td>
</tr>
<tr>
<td>ff_flags(4)**</td>
<td>In</td>
<td>5</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>For the txonly version of the EDH processor, this port controls the five flag bits in the FF flag word of the outgoing EDH packet.</td>
</tr>
<tr>
<td>anc_flags(4)**</td>
<td>Out</td>
<td>5</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>This output port of the rxtx and rxonly versions of the EDH processor represents the five flags bits from the ANC flag word of the last received EDH packet.</td>
</tr>
<tr>
<td>ap_flags(4)**</td>
<td>Out</td>
<td>5</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>This output port of the rxtx and rxonly versions of the EDH processor represents the five flags bits from the AP flag word of the last received EDH packet.</td>
</tr>
<tr>
<td>ff_flags(4)**</td>
<td>Out</td>
<td>5</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>This output port of the rxtx and rxonly versions of the EDH processor represents the five flags bits from the FF flag word of the last received EDH packet.</td>
</tr>
<tr>
<td><strong>Error Signals</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>packet_flags</td>
<td>Out</td>
<td>4</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>This output port contains four error flags asserted when certain error conditions are detected in the input EDH packet. The error flags are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bit 3: EDH packet format error (for example, wrong number of words, incorrect values in fixed value locations)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bit 2: EDH packet checksum error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bit 1: Parity error detected on a word in the EDH packet</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bit 0: EDH packet missing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For more details, refer to “Error Detection and Reporting.”</td>
</tr>
<tr>
<td>edh_ap_err</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>This signal is asserted when a CRC error is detected with the AP CRC. For more details, “Error Detection and Reporting.”</td>
</tr>
</tbody>
</table>
Video Timing Signals

The EDH processor produces a set of video timing signals useful to downstream video processing. All of these signals are synchronous with the vid_out port. For example, the trs output is asserted for four consecutive clock cycles while the four words of each EAV and SAV are present on the vid_out port.

The txonly version has a reduced set of timing signals, because it typically interfaces directly to an SDI encoder and transmitter where there is no need for the full set of video timing signals. However, most of the video timing signals are available internally in the txonly version and can be brought out of the module as output ports by modifying the design.

Figure 7-1 shows the relative timing of the video timing signals generated by the EDH processor. There is a four clock cycle latency through the EDH processor module – the data on the vid_out port is delayed from vid_in by four clock cycles. The trs output is asserted during all four words of the EAV or SAV sequence. The f and v signals change coincident with the first word of an EAV. The h signal always rises coincident with the first word of an EAV and always falls coincident with the first word of an SAV. The h signal is not a

Table 7-1: I/O Signals (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>I/O</th>
<th>Width</th>
<th>rxtx</th>
<th>rxonly</th>
<th>txonly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>edh_ff_err</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>This signal is asserted when a CRC error is detected with the FF CRC. For more details, refer to “Error Detection and Reporting.”</td>
</tr>
<tr>
<td>err_flg_en</td>
<td>In</td>
<td>16</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>This input vector provides error enables for each of the possible error conditions. If an error enable is High, the corresponding error condition causes the err_detected output to be asserted when detected. For more details, refer to “Error Detection and Reporting.”</td>
</tr>
<tr>
<td>err_detected</td>
<td>Out</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>This output signal is asserted when any enabled error condition (with its error enable driven High on the err_flg_en port) is detected. For more details, refer to “Error Detection and Reporting.”</td>
</tr>
</tbody>
</table>

Notes:

1. The EDH processor has a video output port even though this version does not modify the video by inserting or updating EDH packets. The video on the vid_out port, for all versions of the EDH processor, is delayed by four clock cycles from the video on the vid_in port, allowing the EDH processor to examine the video stream and properly generate the various video timing output signals. The video timing signal outputs match the delayed video present on the vid_out port.
2. The f, v, h, and trs video timing output signals are not part of the standard set of outputs of the txonly version of the EDH processor. However, these signals are present inside the module and can be brought out of the module if needed.
3. The v_pos port defaults to 10 bits wide. For HD support, this port should be 11 bits wide. The width of this port is controlled by the VWIDTH parameter or generic.
4. The anc_flags, ap_flags, and ff_flags ports are in all three versions of the EDH processor. However, the purpose of these three ports is different for the txonly version of the processor. For the txonly version, these ports are input ports. For the rxtx and rxonly versions, these ports are output ports. See “EDH Packet Error Flags,” for more details.
completely accurate indication of the horizontal blanking interval – defined as starting with the first word of the EAV and continuing through the last word of the SAV. However, as shown in Figure 7-1, it is easy to OR the \( h \) and \( \text{trs} \) signals together to generate a signal that is asserted for the duration of the horizontal blanking interval.

The \( v\_\text{pos} \) port changes values coincident with the second word of the EAV as shown in Figure 7-1. Therefore the new line number on the \( v\_\text{pos} \) port is ready in time to be inserted into an HD video stream immediately after the EAV, as required of HD-SDI transmitters by the HD-SDI standard.

![Figure 7-1: Video Timing Signals](image)

**EDH Packet Error Flags**

An EDH(1) packet contains three flag words: the ancillary data (ANC) flag word, the active picture (AP) flag word, and the full-field (FF) flag word. The flags in the AP flag word are associated with errors detected with the active-picture CRC value. Likewise, the flags in the FF flag word are associated with errors detected with the full-field CRC value. The flags in the ANC flag word are associated with errors detected in ancillary data packets. Each flag word contains five flags as described below.

**EDH – Error Detected Here**

Any piece of equipment detecting a difference between the calculated AP or FF CRC values for the field and the CRC values located in the received EDH packet sets the corresponding EDH flag in the AP or FF flag word. The ancillary data EDH flag is set if a checksum error is detected in at least one ANC packet in the previous field.

**EDA – Error Detected Already**

This flag indicates that some upstream device detected an error and then corrected the CRC value. A video device receiving an EDH packet with the EDH flag set by the upstream device must set the EDA flag in the output EDH packet and clear the EDH flag unless it, too, detects an error.

---

1. The acronym EDH is overused for three different purposes. First, EDH, as in Error Detection and Handling, is the name of the protocol used for error detection in SD-SDI. Second, the packets containing the error detection information are called EDH packets. Third, each of the three error flag words in the EDH packet contains an EDH (Error Detected Here) flag.
IDH – Internal Error Detected Here

Any piece of equipment can assert the IDH flag to indicate that some internal processing error, unrelated to the actual video stream data, occurred. The IDH flag is provided as a signaling mechanism to allow video equipment to indicate the occurrence of internal errors. These internal errors can be anything unrelated to the actual video stream, such as the detection of an overheating condition.

IDA – Internal Error Detected Already

This flag indicates that some upstream piece of equipment detected an internal error. A video device processing an EDH packet with the IDH flag set by an upstream device must set the IDA flag in the outgoing EDH packet. It must also clear the IDH flag in the outgoing packet unless it, too, detects an internal error.

UES – Unknown Error Status

This flag indicates that the video stream was received from equipment not supporting the EDH standard. A piece of equipment generating EDH packets and inserting them into a video stream that does not already contain EDH packets can assert the UES bit to indicate that the video stream might have been subject to undetected errors prior to the point where the EDH packets were inserted.

The handling of EDH flags is different for each version of the EDH processor.

rxtx EDH Error Flag Handling

The rxtx version of the EDH processor updates the EDH packets as they pass through the processor with new error flags as shown in Figure 7-2, page 154.

If the EDA or EDH flag is set in the ANC, FF, or AP flag words or the received packet, the corresponding EDA flag is set in the outgoing packet. Likewise, if the IDA or IDH flag is set in a flag word of the received packet, the output IDA flag is set in the outgoing packet.

The EDH processor generates the EDH flags in the FF and AP flag words of the output packet by comparing the CRC values calculated locally with those from the input packet.

The IDH flags in the FF and AP flag words of the output packets are controlled by the ap_idh_local and ff_idh_local inputs.

In the ANC flag word, the EDH and IDH flags in the output packet are controlled by the anc_edh_local and anc_idh_local inputs.

The EDH processor can generate automatically the UES flags in all three flag words. If the enable_ues input is High, then the EDH processor asserts the UES flags if the input video stream does not contain EDH packets. If the UES bit is set in a flag word in the input packet, the corresponding UES flag in the output packet also is set.

The rxtx EDH processor also captures the flags from the input EDH packet and outputs them on three 5-bit output ports called ap_flags, ff_flags, and anc_flags. The bit assignments for these ports are shown in Table 7-2.

Table 7-2: EDH Error Flag Port Bit Assignments

<table>
<thead>
<tr>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>UES</td>
<td>IDA</td>
<td>IDH</td>
<td>EDA</td>
<td>EDH</td>
</tr>
</tbody>
</table>
Chapter 7: Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor

The values of the ap_flags, ff_flags, and anc_flags ports are updated as each EDH packet is received. Thus, the values change during the time the EDH packet is being processed and are held until the next EDH packet is processed. The edh_packet output can be used to determine when EDH packet processing is taking place and when it is safe to examine these error flags. See Figure 7-3 for details.

Figure 7-2: rxtx EDH Packet Flags and Error Detection Logic

Figure 7-3: EDH and Error Flags Timing
**rxonly EDH Error Flag Handling**

The rxonly version of the EDH processor does not modify the EDH packets as they pass through the EDH processor. The rxonly version has the same flag processing logic as shown in Figure 7-2. It produces all of the output EDH packet flags, just as the rxtx version does, for error detection purposes. The only difference is that the output flags are used only for error detection and are not inserted into the video stream.

**txonly EDH Error Flag Handling**

The txonly version of the EDH processor always generates new EDH packets and inserts them into the video stream, overwriting any existing EDH packet if present. This version of the EDH processor has none of the logic shown in Figure 7-2. Instead, three 5-bit input ports called ap Flags, ff Flags, and anc Flags allow direct external control of all of the error flags in the three flag words of the output packet. The bit assignments for these ports are identical to those shown in Table 7-2.

**Error Detection and Reporting**

The rxtx and rxonly versions of the EDH processor implement a simple error detection and reporting mechanism. The txonly version of the EDH processor has no error detection capability because it does not process input EDH packets. The error detection mechanism for the rxtx and rxonly versions is shown on the right side of Figure 7-2.

The locally detected error conditions are all indicated by the packet Flags, edh_ap_err, and edh_ff_err signals on the output ports as described in Table 7-1. The ap Flags, ff Flags, and anc Flags output ports indicate the values of the error flags from the most recently received EDH packet. From this set of error signals, it is possible to implement any kind of error detection, counting, and reporting mechanism desired.

Internal to the EDH processor, the various error signals are each ANDed with the corresponding bit from the err_flg_en input port and then ORed together to generate the err detected output signal. The err_flg_en error mask does not affect any of the error output signals except err_detected. Table 7-3 shows the bit assignments of the err_flg_en error mask vector. The err_detected output is formed from a subset of the various error conditions available inside the EDH processor.

**Table 7-3: err_flg_en Bit Assignment**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDH packet checksum</td>
<td>Out AP Flags</td>
<td>Out FF Flags</td>
<td>Out ANC Flags</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UES</td>
<td>IDA</td>
<td>IDH</td>
<td>EDA</td>
<td>EDH</td>
<td>UES</td>
<td>IDA</td>
<td>IDH</td>
<td>EDA</td>
<td>EDH</td>
<td>UES</td>
<td>IDA</td>
<td>IDH</td>
<td>EDA</td>
<td>EDH</td>
<td></td>
</tr>
</tbody>
</table>

As shown in Figure 7-3, all of error and flag output ports (packet Flags, edh_ap_err, edh_ff_err, ap Flags, ff Flags, anc Flags, and err_detected) change during the period when the EDH packet is being updated or generated (when edh_packet is asserted). After EDH packet processing is completed, these signals retain the values determined during EDH packet processing until the next EDH packet is received. Thus, they can be examined by a processor or other hardware in the FPGA to accumulate error statistics.

**Supporting the Optional HD Functions**

Two parameters/generics are used to configure support for the optional HD functions. The following default values for these configure the EDH processor without HD support:
Chapter 7: Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor

SD_ONLY = 1
VWIDTH = 10

To support the HD functions, the default values should be overridden and set to:

SD_ONLY = 0
VWIDTH = 11

The VWIDTH parameter/generic defines the width of the v_pos output port. Only 10 bits are required for SD mode, and 11 bits are required to support HD mode. It is possible to specify a VWIDTH of 10 and support the HD functions if the v_pos output port is not used.

The SD_ONLY parameter/generic controls two options inside the EDH processor. The first is the hd_sd bit. If SD_ONLY = 1, this bit is forced to always be zero, otherwise this bit is controlled by a register loaded by the PicoBlaze processor. SD_ONLY also controls how the PicoBlaze EDH processor is instantiated. The customized kcpsm3 module provided with this reference design, called kcpsm3_edh, is identical to the standard PicoBlaze kcpsm3 module except that it allows a parameter/generic to control the size of the scratchpad memory. Three sizes of scratchpad memory are allowed: the normal 64 locations, 32 locations, or 16 locations. If SD_ONLY is 1, the scratchpad memory is generated with only 16 locations, an amount sufficient for the SD processing. To support the HD functions, the scratchpad memory must have 64 locations. Thus, setting SD_ONLY to 1 when only SD support is required makes the overall EDH processor design smaller when implemented in the FPGA.

There is one additional requirement for supporting the HD functions. The software stored in the PicoBlaze instruction code block RAM is different when HD support is enabled. The edh_code and edh_hd modules instantiate one block RAM with INIT statements generated by the PicoBlaze assembler. The edh_code module is used when only the SD functions are required. The edh_hd module must be used when the HD functions are to be supported. Never use the edh_hd module when the SD_ONLY parameter/generic is set to 1. The software in edh_hd attempts to use scratchpad memory locations that do not exist, causing the scratchpad memory address to wrap around, greatly confusing the software.

Hardware Description

Figure 7-4 is a color-coded composite block diagram of all three versions of the PicoBlaze based EDH processor. Blocks common to all three versions are shown in black. Parts of the design specific to certain versions are color-coded. For example, the input EDH packet receiver and checking logic block, shown in blue, is only used in the rxtx and the rxonly versions. Likewise, the output EDH packet RAM and video output MUX, shown in red, is used only for the versions that output EDH packets, namely the rxtx and txonly versions.

Refer to “Appendix A: Detailed Hardware Description,” page 160 for a detailed description of the EDH processor hardware.

The instruction ROM for the PicoBlaze processor is not included in the EDH processor module. Instead, the address and data buses of the PicoBlaze processor are connected to ports of the EDH processor module. The instruction ROM must be instantiated at a higher level of the project hierarchy, which allows two EDH processors to share one block RAM holding the EDH processing software. Because the block RAMs in Xilinx FPGAs are dual-ported, two EDH processors can share one block RAM. Make sure that the clock connected to the block RAM is the same clock as is connected to the cpuclk port of the EDH processor.
Software Description

The software running on the PicoBlaze processor is entirely interrupt driven and implements a complex state machine. The PicoBlaze processor normally receives two interrupts per video line, one at the beginning of the horizontal blanking interval when the EAV occurs and the other at the beginning of the active portion of the line when the SAV occurs. The software executes specific actions depending on the current state of the software state machine and on which of the two interrupts occurs.

A third type of interrupt, called the TRS time-out interrupt, also can occur if an EAV or SAV does not appear in the video stream within a certain period of time. The software uses this interrupt to determine that the video stream no longer contains valid video.

Refer to “Appendix B: Detailed Software Description,” page 165 for a detailed description of the EDH processor software.
Using the EDH Processor in an SDI Interface

The EDH processor must be connected to the video stream by connecting the 10-bit input video source to the vid_in port. If the video stream is only 8 bits wide, connect it to the most-significant 8 bits of the vid_in port and connect the two LSBs of the vid_in port to zero.

The EDH processor must have a word-rate video clock connected to its clk input port. This clock can be either exactly the video word-rate or a multiple of the video word rate. For example, for normal NTSC and PAL 4:2:2 video, the word rate is 27 MHz. The clk port can be clocked at 27 MHz or an integer multiple of 27 MHz, such as 54 MHz. If a 54 MHz clock signal is used, then the ce signal must be asserted every other clock cycle to run the video path at 27 MHz.

If the input video stream comes from a Virtex-II Pro RocketIO transceiver using an oversampling data recovery method, as described in Chapter 13, “Multi-Rate HD/SD-SDI Receiver Using RocketIO Multi-Gigabit Transceivers,” then it is common for the video clock to be running at some multiple of the video rate, 108 MHz for example. The data recovery unit described in Chapter 13 provides a clock enable signal that can be directly connected to the EDH processor’s ce input to throttle the video data path to 27 MHz.

The PicoBlaze processor does not have a clock enable. Thus, it runs independently of the video stream clock. A separate clock input, cpuclk, provides the clock to the PicoBlaze processor. The PicoBlaze processor clock can be the same as the video clock or it can be independent of the video clock. The EDH processor design allows these two clocks to be totally asynchronous, running at different frequencies. The PicoBlaze clock must be at least as fast as the fastest supported SD video word rate, otherwise the software running on the PicoBlaze processor does not execute fast enough during some time-critical processes.

Refer to Chapter 8, “SD-SDI Integration Example for the Serial Digital Video Demonstration Board” and Chapter 14, “Multi-Rate SDI Integration Examples for the Serial Digital Video Demonstration Board” for detailed examples using the EDH processor. The reference design discussed in Chapter 8 uses four EDH processors, two rxtx and two txonly. The reference designs in Chapter 14 are multi-rate HD-SDI and SD-SDI interfaces, and they use the optional HD features of the EDH processor to reduce the overall size of the design.

The EDH processor design in Chapter 6 works with the video flywheel described in Chapter 5, “SD-SDI Video Flywheel.” In fact, the Chapter 6 EDH processor requires the video flywheel to function correctly. This video flywheel generates EAV and SAV symbols and inserts them into the video stream if they are missing or damaged. It also handles video format detection and synchronization chores for the EDH processor. The PicoBlaze EDH processor can be used with the Chapter 5 video flywheel, if this feature is desired. However, the video flywheel is not required when using the PicoBlaze EDH processor because this processor design does its own video format detection and synchronization.

Chapter 6 also describes some ancillary data processing modules that can multiplex and demultiplex ancillary data packets with the video stream. The PicoBlaze EDH processor was not designed to work with these ANC modules. It is possible to modify the EDH processor design to incorporate these modules, but there currently is no reference design with these modifications.

Design Size

Table 7-4 shows the FPGA resources used by the various EDH processor versions both with and without the optional HD functions. All results were obtained with ISE 6.3i using XST with the Verilog versions of the EDH processor design. The results are for the EDH
processor only and do not include other elements commonly associated with an SDI receiver, such as the encoder, decoder, framer, and SerDes. Each EDH processor also requires a block RAM to hold the instruction code, and it is possible for two EDH processors to share one block RAM if they are running the same EDH processing software.

Table 7-4: FPGA Resource Usage

<table>
<thead>
<tr>
<th>EDH Processor Version</th>
<th>FFs</th>
<th>LUTs</th>
<th>Block RAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>rxtx</td>
<td>424</td>
<td>507</td>
<td>1</td>
</tr>
<tr>
<td>rxtx with HD functions</td>
<td>429</td>
<td>531</td>
<td>1</td>
</tr>
<tr>
<td>rxonly</td>
<td>412</td>
<td>379</td>
<td>1</td>
</tr>
<tr>
<td>rxonly with HD functions</td>
<td>417</td>
<td>403</td>
<td>1</td>
</tr>
<tr>
<td>txonly</td>
<td>271</td>
<td>349</td>
<td>1</td>
</tr>
<tr>
<td>txonly with HD functions</td>
<td>276</td>
<td>373</td>
<td>1</td>
</tr>
</tbody>
</table>

It is not always advantageous to share the PicoBlaze instruction code ROM between two EDH processors. This sharing tends to force these two EDH processors to be located close together in the FPGA to share the block RAM which, in fact, might not be the optimum placement of these EDH processors due to other design constraints.

**Design Speed**

The EDH processor has two clock domains, the video clock and the PicoBlaze clock.

The video clock rate is determined by the video word rate and never has to run faster than that rate. SD video rates are 27 MHz, 36 MHz, and 54 MHz, depending on the video format. In the worst case, the video path must run at the HD video rate of 74.25 MHz if the HD functions are supported. The video logic, clocked by the clk input, meets 74.25 MHz timing in all speed grades of Spartan-3, Virtex-II, Virtex-II Pro, and Virtex-4 FPGAs with ample margin.

The control section of the EDH processor, including the PicoBlaze processor, is clocked by an independent clock without a clock enable. This portion of the design only needs to run as fast as the fastest SD video word rate. Thus, if only 4:2:2 normal aspect ratio SD video is being supported, the PicoBlaze section only needs to run at 27 MHz. In the worst case, when supporting 4:4:4:4 SD video, the PicoBlaze processor must be clocked at a minimum of 54 MHz. The PicoBlaze clock does not have to be related to the video clock frequency. For example, it can be 33 MHz when the video clock is 27 MHz.

The control section can be clocked faster than the minimum speed requirements listed above. It is possible to clock the control section at over 90 MHz in -5 speed grade Virtex-II Pro devices and over 65 MHz in -4 speed grade Spartan-3 parts. Running this section at the slowest possible speed reduces power consumption.

**Conclusion**

The EDH processor presented here represents a significant savings in size over the previous EDH processor design from Chapter 6. By using a PicoBlaze processor to implement the functions that do not have to run at the video word rate, the size of the EDH processing function is reduced while, at the same time, making the design more flexible. Taking advantage of this flexibility, additional HD video processing functions are added to the design.
Chapter 7: Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor

Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_sd-edh-picoblaze.zip.

Appendix A: Detailed Hardware Description

The EDH processor consists of two main sections, as shown in Figure 7-4, page 157, the control section with the PicoBlaze processor and various registers and counters, and the video data path where the EDH packets are detected, processed, and created or updated.

Control Section

The PicoBlaze processor lies at the heart of the control section. Connected to its output port are various control and timing registers and counters. On its input port, the PicoBlaze processor can examine various video timing signals, interrupt flags, EDH packet flags, CRC values, and the value of the words_per_line register. Each version of the EDH processor has a different set of input port signals that it can examine.

The following is a description of each register that can be controlled or examined by the PicoBlaze processor. Registers written directly by the PicoBlaze processor are all clocked by the cpuclk signal. Registers and control signals that interface to the video data path or that are outputs from the module are clocked by the clk signal and enabled by the clock enable signal, ce. Various techniques are used to handle signals that cross between the two clock domains. For control signals and data that are written by the PicoBlaze processor but must be synchronized to clk, the register written by the PicoBlaze processor is clocked by cpuclk and then dual-rank synchronization registers, clocked by clk, are used to eliminate metastability problems. An example of this case is the std_reg register. In other cases, data is written into a register clocked by cpuclk, well in advance of when it is loaded into another register clocked by clk, allowing the data to settle before it is loaded into the final register. An example of this case is the data path from the eav_reload_val register to the h_count register. Finally, many of the inputs to the in_port bus of the PicoBlaze processor are signals in the clk domain. These signals are either synchronized to the cpuclk domain through dual-rank synchronizers (for example, the interrupt request status flags) or are examined by the PicoBlaze processor during periods when the software knows that they are stable for a sufficient period of time.

ms_reg Register

The EDH packet RAM and many of the registers that are written by the PicoBlaze processor are wider than the 8-bit out_port bus of the PicoBlaze processor. Some of these data paths are up to 12 bits wide. To write to wide data paths, the software first writes the most significant bits to the four-bit wide ms_reg register. Then, the software writes the least significant 8 bits to the actual register or RAM location. The upper bits from the ms_reg register are concatenated with the 8 bits from the out_port to form the full data vector. The ms_reg register is in the cpuclk domain.

h_count, eav_reload_val, words_per_line, and edh_position Registers

These registers, shown in Figure 7-5, are all related to the horizontal position of the video stream.
The h_count counter holds the current horizontal position of the video stream. The h_pos output port is always equal to the value of the internal h_count counter. Normally, the h_count counter increments by one every clk cycle when the ce signal is asserted. When the TRS detection module identifies that the current video word is the XYZ word of the SAV, then the h_count counter is cleared synchronously to zero on the next clk cycle.

During the synchronous switching interval, the h_count counter is loaded with the position of the EAV when the first word of the EAV is found, allowing the horizontal count to immediately resynchronize on the first EAV after the synchronous switching interval. This feature is only enabled when the en_sync_switch input is asserted. The EAV position is held in the eav_reload_val register, initialized by the PicoBlaze processor when the format of the video is detected.

If the h_count counter reaches its maximum count (all ones), the trs_timeout signal is asserted. This signal interrupts the PicoBlaze processor to tell it that an SAV did not occur within an acceptable amount of time.

The edh_position register is loaded by the PicoBlaze processor with a value one less than the horizontal position of the first word of the EDH packet when the software determines the video format. A value of one less than the first word location is used so that the hardware has a one clock cycle advance warning of the position of the EDH packet. The edh_pos_match signal is asserted when the h_count value matches the contents of the edh_position register.

The words_per_line register loads the h_count value during the XYZ word of the SAV on each line. This captures a value in the words_per_line register equal to one less than the total number of words found on the video line. The value of the words_per_line register can be read by the PicoBlaze processor. The software uses the number of words found on each line for two purposes. First, during video format detection, it uses this information to help identify the format of the incoming video stream. Second, after the EDH processor is

**Figure 7-5: Horizontal Position Registers**

If the h_count counter reaches its maximum count (all ones), the trs_timeout signal is asserted. This signal interrupts the PicoBlaze processor to tell it that an SAV did not occur within an acceptable amount of time.

The edh_position register is loaded by the PicoBlaze processor with a value one less than the horizontal position of the first word of the EDH packet when the software determines the video format. A value of one less than the first word location is used so that the hardware has a one clock cycle advance warning of the position of the EDH packet. The edh_pos_match signal is asserted when the h_count value matches the contents of the edh_position register.

The words_per_line register loads the h_count value during the XYZ word of the SAV on each line. This captures a value in the words_per_line register equal to one less than the total number of words found on the video line. The value of the words_per_line register can be read by the PicoBlaze processor. The software uses the number of words found on each line for two purposes. First, during video format detection, it uses this information to help identify the format of the incoming video stream. Second, after the EDH processor is
locked to the video stream, the word count of each video line is checked against the expected word count for the current video format. Any deviation from the expected value is treated as an error. When too many consecutive errors are encountered, the software tries to resynchronize to the video stream.

v_count and v_pos Registers

These registers, shown in Figure 7-6, are related to the vertical line number.

The v_count register is a simple register, not a counter. This register, written by the PicoBlaze processor, contains the line number of the next vertical line. The value of the v_count register is updated by the PicoBlaze processor during its processing of the SAV interrupt on each video line. Thus, the value in v_count register is set up near the beginning of the active portion of each line, well in advance of the actual time when the vertical line number changes.

The v_pos register loads from the line number value stored in v_count register when the EAV is detected. By definition, each new video line begins with the EAV. Thus the value of the line number should increment, or roll over to 1, synchronously with the beginning of the EAV. The value is transferred to the v_pos register synchronously with the beginning of the EAV. Thus, the v_pos register, not the v_count register, contains the actual vertical line number value.

The v_pos register loads after the first word of the EAV as shown in Figure 7-1. Therefore the v_pos register is stable by the end of the second word of the EAV, allowing the vertical line number to be set up well in advance of when it must be inserted into the HD video stream after the fourth word of the EAV if the application is using the EDH processor to generate line numbers for insertion into the HD video stream.

The PicoBlaze processor can force the v_pos register to load by asserting the force_v_pos ld signal. This assertion causes the v_pos register to load the value from the v_count register independently of when the EAV occurs. This operation is only done during the synchronization process when the software determines the vertical line count.
When the EDH processor is configured for SD only operation, the v_count and v_pos registers are 10 bits wide. If HD support is enabled, these registers are 11 bits wide to accommodate the larger line counts associated with HD video formats.

std_reg, ctrl0_reg, and sync_switch Registers

These three registers, shown in Figure 7-7, hold output and control signals written by the PicoBlaze processor.

Figure 7-7: std_reg, ctrl0_reg, and sync_switch Registers

The std_reg and the sync_switch registers are each a pipeline of three registers. The first register in each pipeline is in the cpuclk domain and is written directly by PicoBlaze processor. The second and third registers form dual-rank synchronizers to move the signals to the clk domain.

The ctrl0_reg register holds control signals related to generating the CRC values and controlling EDH packet insertion and the address counter to the EDH packet RAM. This register is in the cpuclk domain. Each control signal from this register is treated individually in terms of how it is synchronized to the clk domain.

Video Data Path

As shown in Figure 7-4, the video data path varies considerably between versions of the EDH processor, depending on whether the EDH processor is doing error detection, EDH packet updating, or simple EDH packet generation.

TRS Detection

In all versions, the video stream passes through a TRS detection module. This module examines the video stream and indicates when EAV or SAV symbols are encountered. For those versions of the EDH processor that process EDH packets present in the input video...
stream, the TRS detector also generates a signal indicating when the 4-word header of an ANC packet is detected. The TRS detector delays the video stream by four clock cycles.

**CRC Generation**

The CRC generators follow the TRS detector. These modules generate the AP and FF CRC values for each field of the input video stream. These calculated CRC values are compared against the CRC values stored in the EDH packet in the input video stream to detect errors. They are inserted into the EDH packets in the output video stream when new EDH packets are created or the existing EDH packets are updated.

The CRC generators are controlled by a combination of control signals from the control section and signals generated from the video timing. The PicoBlaze processor clears the CRC generators prior to beginning the CRC calculations for a new field of video by setting the clr_crc bit in the ctrl0_reg register. The PicoBlaze processor controls which video lines are included in the two CRC calculations by setting the enable_ff_crc and enable_ap_crc bits in the ctrl0_reg register. However, the horizontal start and stop points for each CRC calculation are controlled by logic based on the video timing signals. Refer to Chapter 6, “SD-SDI Ancillary Data and EDH Processors” for details of which words and line are included in each CRC calculation.

**EDH Packet Processing and Flags Handling**

For the rxtx and rxonly versions of the EDH processor, the edh_rx_soft module verifies and captures the important information from the EDH packets in the input video stream. This module is almost identical to the edh_rx module from Chapter 6 with a few extra timing signals brought out to output ports from the module. This module implements a state machine to process the input EDH packets. Refer to the description of the edh_rx module in Chapter 6 for more details.

The CRC values from the input EDH packet are captured by the edh_rx_soft module and then are compared with the CRC values calculated locally by the CRC generators to detect any errors in the transmission of the video field.

The edh_rx_soft module also captures the error flags from the three flag words in the input EDH packet. In the rxtx and rxonly versions of the EDH processor, these captured flags are output on the ap_flags, ff_flags, and anc_flags ports. These flags also are fed into the edh_flags2 module.

The edh_flags2 module is virtually identical to the edh_flags module from Chapter 6 and implements the EDH error flags protocol. This module generates new output EDH flags that are used both for error detection and for insertion into updated EDH packets.

**EDH Packet Generation and Insertion**

The rxtx version of the EDH processor updates existing EDH packets. If there are no EDH packets in the input video stream, this version of the EDH processor generates new EDH packets and inserts them into the video stream. The txonly version of the EDH processor always generates and inserts new EDH packets.

New EDH packets are assembled by the PicoBlaze processor and written into the EDH Packet RAM. This 32 x 10-bit RAM is written synchronously with the cpuclk signal through the out_port bus of the PicoBlaze processor. When writing the RAM, the most significant two bits of write data come from the ms_reg register and the least significant 8 bits come from the out_port bus. The write address comes from the port_id bus of the PicoBlaze processor.
Read operations from the EDH Packet RAM are done asynchronously, whenever the RAM is not being written. The read address comes from the edhram_adr_counter counter. This counter is cleared to zero by the clr_edhram_adr_counter control bit from the ctrl0_reg register. It increments on each clk cycle when both ce and insert_edh are asserted. The insert_edh signal indicates when the EDH packet actively is being generated and inserted into the video stream.

The CRC calculations for a field of video are completed by the last active word of the line prior to the line containing the EDH packet. The EDH packet must be assembled and ready for insertion by the time the EDH packet position arrives. The EDH packet must be inserted into the last 23 words of the horizontal blanking interval, just prior to the SAV. Thus, the PicoBlaze processor has only a portion of the horizontal blanking interval to generate the new EDH packet and write it to the EDH Packet RAM prior to the time it must be inserted into the video stream.

The insertion of the new EDH packet into the video stream and the updating of existing EDH packets are done by a MUX. The operation of this MUX is specific to the version of the EDH processor.

For the txonly version of the EDH processor or for the rxtx version when the input video stream does not contain EDH packets, a new EDH packet is created by the PicoBlaze processor and is stored in the EDH Packet RAM. All words of the new EDH packet are read from the EDH Packet RAM and inserted into the video stream by the MUX.

For the rxtx version of the EDH processor when there are existing EDH packets in the input video stream, the contents of the EDH Packet RAM are ignored. Instead, the MUX passes most of the words of the input EDH packet unchanged, replacing only the CRC and flag words with values generated by the CRC generators and edh_flags2 module. The checksum word of the EDH packet must be updated because some of the words of the EDH packet might have been modified. A checksum generator calculates the checksum for the modified packet. The MUX inserts the value from the checksum generator into the last word of the EDH packet.

The rxonly version of the EDH processor does not insert or update EDH packets. Therefore, it does not have an EDH Packet RAM, MUX, or checksum generator.

Appendix B: Detailed Software Description

The PicoBlaze software for the EDH processor implements an interrupt-driven state machine. The PicoBlaze processor is interrupted on each EAV and SAV and if a TRS time-out occurs. Thus, during normal operation, the software state machine is activated twice per video line, at the beginning of the horizontal blanking interval and at the beginning of the active portion of the line.

There are two different versions of the software, one without the optional HD functions and one with these HD functions included. All three versions of the EDH processor (rxtx, rxonly, and txonly) use the same software. Each version of the software is available in versions for both single-ported and dual-ported block RAM implementations.

Figure 7-8 is a simplified overall view of the state machine implemented by the software running on the PicoBlaze processor. In the figure, items shown in gray only appear in the HD version of the software.

After a reset, the state machine begins in the UNLOCKED state. At each EAV interrupt in this state, it checks the words per line (WPL) count against the WPL count from the previous line. Each time the WPL count matches the WPL count from the previous line, a match count is incremented. If the WPL counts do not match, the match count is reset to
zero. When the match count reaches the H_LOCK_MATCH constant value, then the state machine considers itself to be locked horizontally to the video stream and advances to the H_LOCKED state. If the WPL count matches a HD video format, the state machine moves to the H_LOCKED_HD1 state, instead.

In the H_LOCKED state, the state machine tries to lock vertically to the video stream. It waits for the V bit in the EAV XYZ word to rise. When it rises, the F bit is examined to determine the current field. Based on the detected video format and the status of the F bit, the software can determine the current line number of the video stream.

During SAV interrupts in the H_LOCKED state, the WPL count for the line is compared to the value found in the UNLOCKED state. If they don’t match, an error counter is incremented. If they do match, the error counter is cleared. If the error counter ever equals the MAX_ERR_COUNT constant value, the state machine goes back to the UNLOCKED state. This WPL checking routine is shown on the state diagram as a decision diamond called “WPL check” and occurs during the SAV interrupts of most states.

SAV interrupts in the LOCKED state cause the state machine to increment the v_count value, rolling it over to 1 if the maximum line count value is reached. This procedure occurs during the SAV interrupts of most states once the state machine has locked to the video stream. This keeps the v_count value synchronized to the video stream.

During EAV interrupts in the LOCKED state, the v_count value is compared against the starting line number of the FF CRC calculation. When the video stream is at the line before the start of the FF CRC calculation, the state machine sets the enable_ff_crc control bit to arm the FF CRC generator (it starts calculating the CRC at the next EAV) and the state machine moves to the FF1 state.

In the FF1 state, the FF CRC is being calculated. At each EAV interrupt in this state, the state machine compares the current v_count with the starting line number of the AP CRC calculation. When the video stream reaches the line where the AP CRC calculation should begin, the state machine sets the enable_ap_crc control bit to arm the AP CRC generator. The AP CRC calculation only includes the active video words of each line, so the state machine has ample time from the occurrence of the EAV interrupt to set the enable_ap_crc bit before the active portion of the line begins. The FF CRC calculation remains active during the time the AP CRC calculation takes place. The state machine moves to the AP state as soon as it sets the enable_ap_crc bit.

During the AP state, both the AP and FF CRC values are being calculated. On SAV interrupts, the current v_count is compared to the ending line number for the AP CRC calculation. The enable_ap_crc bit is cleared when a match occurs. The AP CRC generator continues to calculate the AP CRC until the EAV of this line. Once the enable_ap_crc bit is cleared, the state machine moves to the FF2 state.

During the FF2 state, the FF CRC calculation continues and the state machine watches for the end of the FF CRC calculation period. During EAV interrupts, the state machine compares the current v_count against the line number where the FF CRC calculation ends. If the end of the FF CRC calculation period has not yet arrived, the state machine checks to see if the current line is a line when the V bit in the XYZ word of the EAV should rise. If V is supposed to rise, but did not, then the state machine increments the v_error_cnt counter to keep track of the number of consecutive fields where the V bit did not rise when expected. If v_error_cnt reaches the MAX_V_ERR_COUNT constant value, the state machine considers itself unlocked from the video stream, at least in the vertical direction, and returns to the H_LOCKED state to try to resynchronize to the video stream.

Once the end of the FF CRC calculation period arrives, the state machine clears the enable_ff_crc bit. It prepares as much of the EDH packet as possible by writing the fixed value words of the packet into the EDH Packet RAM. The EDH packet is inserted just
before the SAV on the next line. The amount of time that the state machine has from the next EAV interrupt until the EDH packet position occurs in the video stream is very limited. Without doing some of the EDH packet preparation here in the FF2 state, the PicoBlaze processor does not have time to form the EDH packet after the next EAV. Once the EDH Packet preparation work is done, the state machine moves to the INSERT_EDH state.

As soon as the EAV interrupt occurs in the INSERT_EDH state, the FF CRC calculation is complete, and the software must complete the EDH packet stored in the EDH Packet RAM before the EDH packet position occurs in the video stream. The AP and FF CRC values are read from the input port, formatted, and stored into the proper locations in the EDH Packet RAM. Likewise, the AP, FF, and ANC flags are read from the input port, formatted, and stored in the EDH Packet RAM. Finally, the checksum is calculated for the new EDH packet. The state machine then sets the insert_edh_arm bit to arm the EDH packet insertion logic. Finally, the state machine moves to the NEW_FIELD state.

When the EAV interrupt occurs in the NEW_FIELD state, the state machine clears the insert_edh_arm bit and then updates the starting and ending line numbers of the AP and FF CRC calculations for the new field. It then moves to the SYNC_SWITCH state.

When the SAV interrupt occurs in the SYNC_SWITCH state, the state machine asserts the sync_switch signal. When the following EAV interrupt occurs, the state machine moves to the LOCKED state and begins processing the next field of video. The sync_switch signal remains asserted until the first SAV interrupt in the LOCKED state.

For HD video processing, the state machine moves into the H_LOCKED_HD1 state as soon as it is locked horizontally to the video stream and determines that the WPL count matches an HD video format instead of an SD video format. During EAV interrupts in the H_LOCKED_HD1 state, the state machine watches for the V bit in the EAV XYZ word to fall. As soon as V falls, the internal v_count value is cleared to zero (the internal v_count is kept in scratchpad memory and usually corresponds to the hardware v_count register, but in this case it is used to count the number of active vertical lines in the field). The state machine moves to the H_LOCKED_HD2 state after the fall of the V bit is detected.

During the H_LOCKED_HD2 state, the state machine counts the number of lines that occur before the V bit rises. Each EAV interrupt the software v_count value is incremented and then the software checks to see if the V bit rose. Once the V bit rises, the v_count value contains the number of active video lines in the field (for interlaced formats) or frame (for progressive formats). The state machine then tries to match the WPL and active line counts against the supported HD video formats. If a match is found, the state machine writes the video format code to the std_reg register, sets the std_locked signal, and moves to the LOCKED_HD state.

In the LOCKED_HD state, the state machine checks that it is still synchronized horizontally (by checking the WPL counts at each SAV interrupt) and vertically (by checking that the V bit rises when expected). Also during the SAV interrupt, the state machine checks to see if the current line contains the synchronous switching interval, setting the sync_switch output if so.

The TRS time-out handling is not shown in the state diagram. If a TRS time-out interrupt occurs, the error_cnt value is incremented. The error_cnt value is always cleared to zero when an SAV interrupt occurs. If error_cnt ever reaches the MAX_ERR_COUNT value, the state machine considers the video stream to be invalid and it moves to the UNLOCKED state.
Figure 7-8: Simplified State Diagram (a)
Figure 7-9: Simplified State Diagram (b)
Figure 7-10: Simplified State Diagram (c)
Chapter 8

SD-SDI Integration Example for the Serial Digital Video Demonstration Board

Summary

The standard-definition serial digital interface (SD-SDI) standard describes how to transport standard-definition (SD) digital video serially over video coax cable. SD-SDI is commonly used to connect SD video equipment in broadcast studios and video production centers.

Section I of this volume, comprising Chapter 2 through Chapter 8 inclusive, describes how to implement SD-SDI transmitter, receiver, and auxiliary functions in Xilinx FPGAs. In addition, Chapter 12 and Chapter 13 describe how the RocketIO™ transceivers found in Virtex™-II Pro devices can be used to implement SD-SDI interfaces. This chapter presents an application example showing how to use the modules from these various other chapters to form complete SD-SDI interfaces. This demonstration application is part of the standard demonstration suite for the Xilinx Serial Digital Video (SDV) board [Ref 1].

Introduction

SD-SDI, defined by the SMPTE 259M [Ref 2] and ITU-R BT.656 [Ref 3] standards, is the standard for transporting uncompressed SD digital video in the broadcast studio and video production center. SD-SDI transports 8-bit or 10-bit digital video serially over 75Ω video coax cable at distances of up to 300 meters. Various bit rates are supported, but the primary rate is 270 Mb/s. The Xilinx SD-SDI chapters have detailed information about the SD-SDI standard.

The SD-SDI application example, described here, is designed for the Xilinx SDV demonstration board. This design includes two SD-SDI transmitters and two SD-SDI receivers. One transmitter and receiver are built with standard I/O and logic resources in the FPGA. The other transmitter and receiver use RocketIO transceivers.

The video streams received by the SD-SDI receivers are checked for errors by error detection and handling (EDH) processors. When errors are detected, LEDs on the SDV board are illuminated. The video stream from either receiver can be retransmitted by the RocketIO based transmitter. Alternatively, internal video test pattern generators (from Chapter 16, “SDTV Video Pattern Generators”) can be used as the video source for the RocketIO based transmitter.

The standard non-RocketIO transmitter is always driven by an internal video pattern generator. This transmitter is referred to in this document as the “fabric” transmitter because it is built entirely using the programmable logic resources (the fabric) of the FPGA.
Chapter 8: SD-SDI Integration Example for the Serial Digital Video Demonstration Board

Four separate EDH processors are used in this application:

- One in each receiver for error detection and EDH packet regeneration
- One in each transmitter to generate and insert EDH packets into the video streams produced by the video pattern generators

The use of separate EDH processors for the receivers and transmitters allows the receivers and transmitters to be operated independently. Four of the EDH processors described in Chapter 6, combined with the other logic of this application, would not fit in the XC2VP4 device on the SDV board. Instead, a new, smaller EDH processor design, based on the Xilinx PicoBlaze™ processor, is used. This new EDH processor is described in Chapter 7, “Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor.”

This SD-SDI application example is designed specifically for the Xilinx SDV demonstration board. Some of the features of this implementation, especially the jitter reduction module, are designed to work around some limitations of the SDV board. However, the bulk of the application code is generic and can be easily adapted to customer applications.

Detailed Application Description

Figure 8-1 is a block diagram of the SD-SDI application. This application consists of four primary blocks. From top to bottom in Figure 8-1, these blocks are the fabric transmitter, the RocketIO transmitter, the RocketIO receiver, and the fabric receiver.

Fabric Transmitter

The fabric transmitter is a standalone SD-SDI transmitter not connected to either receiver. Video pattern generators, built in the FPGA, provide the video stream for this transmitter. Two video pattern generators are included, one for NTSC and one for PAL. These video pattern generators are fully described in Chapter 16. The video pattern generators can produce either standard SMPTE EG-1 color bars or SMPTE RP 178 SDI checkfield patterns.

An EDH processor generates and inserts EDH packets for error detection into the video generated by the video pattern generators prior to transmission. After EDH packet insertion, an SD-SDI encoder scrambles the video per the SD-SDI standard prior to transmission to ensure an adequate bit transition density. A serializer converts the encoded data into a 270 Mb/s serial bitstream.

RocketIO Transmitter

The RocketIO transmitter can be driven from one of the two video test pattern generators, one producing NTSC video and the other producing PAL video. As with the fabric transmitter, an EDH processor inserts EDH packets into the video from the video pattern generators. Alternatively, the video source for the RocketIO transmitter can be either SD-SDI receiver. The video streams from both receivers pass through jitter reduction modules. The jitter reduction module of the selected receiver controls a VCXO, used as part of a frequency-locked loop, to produce a low-jitter reference clock that is frequency-locked to the selected receiver’s recovered clock.
Figure 8-1: SD-SDI Application Block Diagram
The selected video stream, either from one of the video pattern generators or from one of the receivers, is encoded prior to being serialized by the RocketIO transmitter. A 54 MHz reference clock is multiplied by 20 in the RocketIO transceiver to generate a 1.08 GHz serial clock. This rate is four times the actual SD-SDI bit rate of 270 Mb/s. The RocketIO transceiver sends each encoded bit four times consecutively, which effectively produces a 270 Mb/s SD-SDI bitstream on the output of the RocketIO transmitter.

The video path of the transmitter, from the video pattern generators through the SD-SDI encoder, is clocked by the 54 MHz riotx_gclk signal. However, this video path must run at the 27 MHz SD-SDI word rate. A clock enable signal, asserted every other cycle of the 54 MHz clock, is generated and distributed to the various modules in the transmitter video path. This signal also controls a MUX on the output of the bit replication logic to select which half of the 40-bit vector produced by bit replication is loaded into the RocketIO transceiver on each cycle of the 54 MHz clock.

### RocketIO Receiver

The RocketIO receiver uses the oversampling technique described in Chapter 13, “Multi-Rate HD/SD-SDI Receiver Using RocketIO Multi-Gigabit Transceivers” to allow the RocketIO transceiver to be used with the 270 Mb/s bit rate of SD-SDI. In this technique, the RocketIO receiver is given a 108 MHz reference clock, causing it to oversample the input SD-SDI bitstream at eight times the actual SD-SDI bit rate. A data recovery unit (DRU), built in the fabric of the FPGA, recovers the actual SD-SDI data from the oversampled data captured by the RocketIO transceiver. The DRU produces a clock enable output that is asserted whenever the DRU has 10 bits of data ready on its output. This clock enable is usually asserted for one in every four gclk_108M clock cycles, clocking the SD-SDI data through the SD-SDI decoder, framer, and EDH processor at a 27 MHz clock rate.

The EDH processor generates two different CRC values for each field of video. These CRC values are compared against those created by the transmitter and inserted into an EDH packet at a known location in each field of video. The EDH processor reports any errors detected in this process by illuminating LEDs on the SDV board. It also can determine if the video format is PAL or NTSC and uses another LED to indicate which video format is detected.

As described above, the video path in the receiver is clocked by a 108 MHz clock with a clock enable from the DRU, driving the actual data rate to 27 MHz. The PicoBlaze processor in the EDH module, however, does not have a clock enable and does not meet timing at 108 MHz in a -5 Virtex-II Pro FPGA. Therefore, the EDH processor has a separate clock input for the PicoBlaze processor. This clock input can run at a different frequency than the video clock and does not have to be related to the video clock in any way. In this receiver, the PicoBlaze clock is the global 54 MHz clock.

The video stream from the receiver is written into a jitter reduction module for potential use as the video source for the RocketIO transmitter.

### Fabric Receiver

The fabric receiver uses an external, low-cost 270 MHz VCO as part of a clock and data recovery (CDR) unit as described in Chapter 2, “SD-SDI Physical Layer Implementation.” The 270 MHz recovered clock from the CDR unit is divided by 10 to provide a 27 MHz video clock for the receiver. The data from the CDR unit is deserialized then decoded and framed. The recovered video stream is checked for errors by an EDH processor with errors reported on an LED. The data from the receiver is written into a jitter reduction module for potential use as the video source for the RocketIO transmitter.
In Figure 8-1, the fabric receiver is shown with two global clocks, one for the 270 MHz bit-rate clock and the other for the 27 MHz word-rate clock. At first glance, it appears that the 270 MHz clock is lightly loaded and can be implemented as a local clock. However, the phase detector places more loads on the 270 MHz clock than might be expected, and it is not easy to arrange these loads so that they are all in close enough proximity that the skew on a local 270 MHz clock can be tolerated.

Another approach to reducing the number of global clocks needed by the fabric receiver might be to clock all the logic of the receiver, including the word-rate logic such as the decoder, framer, and EDH processor, with the 270 MHz bit-rate clock and use a clock enable, asserted once every 10 clock cycles, to drive the word-rate logic down to the 27 MHz word rate. However, extreme care must be taken to ensure that the clock enable signal can meet timing at all flip-flops in the receiver. Doing this with a 270 MHz clock is not easy.

Clocks

This section describes the various clocks used in this application.

- **clk_54M**
  A 54 MHz crystal oscillator on the SDV demonstration board is the source for most of the clocks used in this application. This low-jitter oscillator has a differential 3.3V LVPECL output. The oscillator is AC-coupled to a 2.5V LVDSEXT IBUFGDS input buffer on the FPGA. The 54 MHz signal from the input buffer is connected to the clock input of a DCM and to the REFCLK2 input of the RocketIO transceiver in the RocketIO based transmitter.

- **gclk_54M**
  This global 54 MHz clock from the CLK0 output of the DCM is buffered by a BUFG. This clock provides a reference clock to the CDR PLL in the fabric receiver. It also clocks the PicoBlaze processor in the EDH processor of the RocketIO based receiver.

- **gclk_270M**
  The DCM multiplies clk_54M by five to generate this 270 MHz clock. The DCM output is buffered by a BUFG. This clock drives the serializer of the fabric transmitter at the 270 MHz bit rate.

- **clk_108M**
  The DCM multiplies clk_54M by two to generate this 108 MHz clock. This clock is connected to the REFCLK input of the RocketIO transceiver in the RocketIO based SD-SDI receiver. It provides the correct reference clock frequency to cause the transceiver to oversample the SD-SDI bitstream at eight times the bit rate.

- **gclk_108M**
  This clock is the clk_108M clock from the DCM after being buffered by a BUFG. This signal is used in the RocketIO based receiver to clock the output side of the DRU and all logic in the receiver downstream from the DRU.

- **gclk_27M**
  The DCM divides clk_54M by two to generate this 27 MHz clock. This clock is the word-rate clock used in the fabric transmitter.

- **riorx_rxrecclk**
  The RocketIO transceiver in the RocketIO based receiver produces a recovered clock on its RXRECCLOCK output port. This recovered clock is buffered by a BUFG and is used...
to clock the interface between the transceiver and the DRU. Figure 8-1 shows the BUFG on this clock is optional. It is possible to eliminate this BUFG and use the RXRECLK output of the RocketIO transceiver to directly drive the input section of the DRU (and the user clock inputs of the RocketIO transceiver). riorx_rxrecclk has a total of 134 loads, and the maximum clock frequency is about 108 MHz. If the BUFG is eliminated, the DRU should be location-constrained to be adjacent to the associated RocketIO transceiver to minimize skew between the RXDATA output port of the transceiver and the inputs of the DRU.

- **fabrx_gclk**
  
  This 270 MHz bit-rate clock is from the fabric-based receiver’s clock and data recovery PLL. It is produced by a 270 MHz VCO on the SDV board and enters the FPGA through a IBUFGDS. It is buffered by a BUFG and clocks the bit-rate portions of the fabric receiver.

- **fabrx_gpclk**
  
  The 270 MHz fabrx_gclk is divided by 10 to produce a 27 MHz word-rate clock for the fabric-based receiver. This global clock is used to clock most of the fabric-based receiver logic.

- **riotx_gclk**
  
  This clock is the main clock for the RocketIO based transmitter. This 54 MHz clock is driven by a BUFGMUX. When the video source for the RocketIO based transmitter is one of the internal video pattern generators, the BUFGMUX selects clk_54M as the source for riotx_gclk. When retransmitting video from either SD-SDI receiver, the BUFGMUX selects an external VCXO as the clock source. The VCXO is used as part of a frequency-locked loop designed to reduce the jitter on recovered clock from the selected receiver. The SDV board has an external 27 MHz VCXO, but the RocketIO based transmitter needs a 54 MHz clock. So, the output of the VCXO is doubled by an external PLL before being connected to the BUFGMUX input in the FPGA.

### Alternative DRU Clock Scheme

Figure 8-2 shows an alternative clock arrangement for the DRU. Only the clock generation and RocketIO receiver sections from Figure 8-1 are shown here.
In this version, rather than using a global 108 MHz clock from the DCM to clock the output portion of the DRU and the receiver logic downstream from the DRU, the recovered clock from the RocketIO transceiver (RXRECCCLK) is used, eliminating the need for a global 108 MHz clock.

When this alternative clock scheme is used, the PicoBlaze based EDH processor must treat all signals crossing between the PicoBlaze CPU clock domain and the video clock domain asynchronously. Conversely, in the original version, the CPU clock and the video clock are phase-aligned because they are generated by the same DCM. Because the EDH processor is designed to handle the case where the two clocks are asynchronous, this is not a problem, so either clocking scheme can be used.

The alternative clocking scheme is closer to what is normally used in a multi-rate SDI receiver, where the RocketIO transceiver is used to receive both HD-SDI and SD-SDI bitstreams. Additionally, this alternative scheme uses one less global clock.

If this alternative clock scheme is used, then the BUFG on the RXRECCCLK output of the RocketIO transceiver is not optional because this clock signal now must drive approximately 450 loads.

### Jitter Reduction

When retransmitting a video stream from an SD-SDI receiver, the jitter present on the video stream must be reduced prior to retransmission. The DRU, used to recover SD-SDI data from the oversampled data captured by a RocketIO transceiver, can produce significant amounts of low-frequency jitter as it periodically reconciles slight frequency differences between the rate at which data is recovered and the local reference clock. This low-frequency jitter is not filtered out by the loop filters typically used in PLLs.

Many customer applications that use the RocketIO transceiver and DRU for SD-SDI reception often buffer the received video stream with a frame synchronizer. This has the desirable side effect of virtually eliminating all the jitter inherent in the received video stream and the jitter added by the DRU. However, the SDV board does not have sufficient memory available to implement a frame synchronizer, so another method of reducing jitter must be implemented.

The loop filter for the 27 MHz VCXO on the SDV board cannot filter out the low-frequency jitter from the DRU. Instead of using the 27 MHz VCXO as part of a PLL, it is used to implement a frequency-locked loop (FLL). The periodic phase changes (jitter) caused by the DRU are not tracked by the FLL. Instead, the FLL tracks the average frequency of the recovered video stream.

To implement the FLL, the video stream from the receiver is written into an asynchronous FIFO. Data is read from the FIFO using the clock from the 27 MHz VCXO. The frequency of the VCXO is controlled so that the data level of the FIFO is maintained at about half full.

The VCXO on the SD board used in the FLL runs at 27 MHz. A VCXO frequency of 54 MHz would be better because a 54 MHz reference clock is required by the RocketIO transmitter when transmitting SD-SDI bitstreams. Therefore the 27 MHz signal from the VCXO must be doubled before it can be used as the reference clock to the RocketIO transmitter. In this application, the 27 MHz clock from the VCXO is doubled using an ICS8745 PLL from the SDV demonstration board. A DCM also can be used to double the clock from the VCXO, but the ICS8745 produces a clock with less jitter than the DCMs in the Virtex-II Pro FPGA.
Design Size

Table 8-1 shows the FPGA resources used by this application. These results were obtained with ISE 6.3i using XST with the Verilog version of the design. Area optimization was used for XST. All timing constraints were met with a -5 speed grade Virtex-II Pro XC2VP4 device. Refer to the discussion in Chapter 13 regarding the use of 8X oversampling of 270 Mb/s bitstreams in -5 speed grade Virtex-II Pro devices.

<table>
<thead>
<tr>
<th>FF</th>
<th>LUT</th>
<th>Block RAM</th>
<th>MULT18X18</th>
</tr>
</thead>
<tbody>
<tr>
<td>2240</td>
<td>2459</td>
<td>12</td>
<td>2</td>
</tr>
</tbody>
</table>

Conclusion

This chapter presents an SD-SDI application designed for the Xilinx SDV demonstration board. The application shows how to integrate the various SD-SDI receiver, transmitter, and auxiliary function modules from the Xilinx SD-SDI chapters. SD-SDI receivers and transmitters using both the programmable fabric and regular I/O resources of the FPGA and using the RocketIO multi-gigabit transceivers are implemented in the example application.

Xilinx FPGAs are well-suited to implementing SD-SDI interfaces, allowing a high-level of integration between the SD-SDI interfaces and other video processing functions. It is possible, for example, to implement a very complex function, such as a video production switcher with a large number of SD-SDI inputs and outputs and all the video switching and blending effects, on a single Xilinx FPGA device.

Design Files

The reference design files can be downloaded from the Xilinx website at:


Open the ZIP archive and extract file `xapp514_integ-demobrd.zip`. 
Section II: HD-SDI

Audio/Video Connectivity Solutions for the Broadcast Industry
Chapter 9

HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Summary

The High-Definition Serial Digital Interface (HD-SDI) standard describes how to transport high-definition (HD) digital video serially over video coax cable. HD-SDI is used to connect HD video equipment in broadcast studios and video production centers. It is an evolution of the popular SDI standard that is widely used to transport standard-definition (SD) digital video in the broadcast industry.

The flexibility of RocketIO™ multi-gigabit transceivers available in the Virtex™-II Pro family devices combined with the programmable logic of Virtex-II Pro FPGAs makes it possible to implement HD-SDI interfaces. Because every Virtex-II Pro FPGA has multiple RocketIO transceivers, it is possible to integrate multiple HD-SDI interfaces into one Virtex-II Pro device along with other video processing functions.

This chapter describes the electrical specifications for HD-SDI transmitters and the HD-SDI data format. It also presents several implementation examples and reference designs for an HD-SDI transmitter implemented using the Virtex-II Pro FPGA.

Introduction

Use of HD-SDI, defined by the SMPTE 292M standard, is increasing rapidly in broadcast studios and video production centers as the broadcast industry ramps up support for HDTV broadcasting [Ref 1].

HD-SDI builds upon the widely used SDI standard used to transport SD digital video. It is now common to refer to the older SDI standard as SD-SDI to differentiate it from HD-SDI. The SD-SDI and HD-SDI standards share the same electrical characteristics and encoding scheme. However, HD-SDI uses a higher bit rate to accommodate the higher bandwidth requirements of uncompressed HD digital video signals. Because SD-SDI and HD-SDI share common electrical characteristics, it is possible to build video equipment that can support both standards through a single connection.

This chapter discusses the HD-SDI transmitter; Chapter 10 describes how to implement the HD-SDI receiver.

The HD-SDI standard supports both coax cable and optical fiber interfaces. So far, coax cable is the more popular of the two interfaces due to lower cost and commonality with SD-SDI. This chapter only discusses the implementation details for the coaxial cable interface. However, since the data formats and encoding schemes for the optical interface option are identical to the coaxial interface option, the reference designs presented in this
Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

As of this writing, there is a proposal for a new standard, SMPTE 372M, defining a dual-link HD-SDI interface. This proposal uses two HD-SDI interfaces to provide twice the bandwidth, allowing higher bandwidth video formats to be supported. This proposed new standard is not specifically addressed in this chapter, but the HD-SDI reference designs described here can be used as the building blocks for implementing a dual-link HD-SDI interface.

HD-SDI Data Format

This section describes the data format used by HD-SDI, including a discussion of the various video formats that are supported, how those video formats are mapped onto HD-SDI, and how the final bitstream is encoded for transmission over the coax cable.

HD-SDI Supported Video Formats

The SMPTE 292M standard identifies 13 different video formats that can be transported using HD-SDI. Table 9-1 shows these 13 HD-SDI compatible video formats.

Table 9-1: HD-SDI Compatible Video Formats from SMPTE 292M

<table>
<thead>
<tr>
<th>SMPTE Standard</th>
<th>260M</th>
<th>295M</th>
<th>274M</th>
<th>296M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format Designation</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>Format(1)</td>
<td>1035i</td>
<td>1035i</td>
<td>1080i</td>
<td>1080i</td>
</tr>
<tr>
<td>Frame Rate (Hz)</td>
<td>30</td>
<td>30/M</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>Sample Rate (MHz)</td>
<td>74.25</td>
<td>74.25/M</td>
<td>74.25</td>
<td>74.25</td>
</tr>
<tr>
<td>Active Samples per Line and Active Lines per Frame (words x lines)(2)</td>
<td>1920 x 1035</td>
<td>1920 x 1035</td>
<td>1920 x 1080</td>
<td>1920 x 1080</td>
</tr>
<tr>
<td>Total Samples per Line and Total Lines per Frame (words x lines)(2)</td>
<td>2200 x 1125</td>
<td>2200 x 1125</td>
<td>2376 x 1250</td>
<td>2200 x 1125</td>
</tr>
</tbody>
</table>

Notes:
1. The format designations follow the industry practice of using the number of active lines per frame plus either the letter “i” indicating interlaced scan or the letter “p” indicating progressive scan. Thus, a format listed as 1080i has 1080 active lines per frame and is interlaced, while a format given as 720p has 720 active lines per frame and is progressive scan.
2. The active samples per line and total samples per line shown are 2-word samples, one word of Y and one word of C. If there are 1920 active samples in a line, then there are 3840 10-bit active words per line after the channels have been interleaved.

As shown in Table 9-1, some frame rates and sample rates include a divisor called M. The value of M is exactly 1.001. Therefore, the frame rates for the two 1035i formats of SMPTE 260M are 30 Hz and 30 Hz / 1.001 (or approximately 29.97 Hz). The corresponding sample rates for these two standards are 74.25 MHz and 74.25 MHz / 1.001 (approximately 74.1758 MHz), respectively.

The reason for the M divisor is that some of the video formats are defined for true 60 Hz video rates (or 60 Hz derivative rates such as 30 Hz, 25 Hz, and 24 Hz) while others are
defined for the 59.94 Hz video rates and derivatives commonly used in North America. The NTSC TV standard used in North America historically has used a field rate of 59.94 Hz, which is carried forward into the HDTV video formats. The major implication for HD-SDI is that there are two different bit rates supported by the HD-SDI standard: 1.485 Gb/s and 1.485 / M Gb/s.

All video formats in Table 9-1 are 10-bit 4:2:2 component video. They use the YCbCr color space. The chroma (Cb and Cr) information is sampled at half the rate of the luma (Y) information. Each video sample is defined by two 10-bit words, one luma word and one chroma word. The chroma word of each sample is either a Cb or a Cr word. Consecutive samples alternate between Cb or Cr.

SMPTE recommended practice RP 211 adds five additional “segmented frame” video formats that are compatible also with HD-SDI. RP 211 takes the five SMPTE 274M 1080p standards and defines a segmented frame format corresponding to each of them. These segmented frame video formats have the same number of words and lines and the same frame rates as their 1080p counterparts. The primary difference is that the single progressive frame is divided or segmented into two fields, consisting of even lines in one field and odd lines in the other. Although this seems to be the same as interlaced video, there is a fundamental difference between interlaced video and segmented frame video. In interlaced video, the source image is rescanned for each field, so the two fields represent images that are separated in time. The two fields of a segmented frame format are both taken from the same progressive scan frame of the image. The two segmented fields represent the same image and are not separated in time.

The motivation behind defining the segmented frame formats was that HD equipment, particularly recorders, capable of handling the 1080p formats was almost non-existent early in the development of HD broadcasting. However, the 1080p format is an ideal format to use as a master format since it is easy to convert 1080p video to all the other video formats without artifacts. The segmented frame formats (1080sF) retain all the information of the 1080p format, but make the actual video stream appear to be an 1080i (interlaced) video stream. Video equipment and recorders supporting 1080i are more available than those supporting 1080p. Since 1080sF looks just like 1080i, a video recorder designed for 1080i can record the 1080sF video format.

Table 9-2 shows the five segmented frame video formats defined by SMPTE RP 211. The last row of the table shows the 1080p formats from Table 9-1 that correspond to each 1080sF format. In terms of data format, 1080sF formats with frame rates of 30, 30 / M, and 25 are indistinguishable from the 1080i formats D, E, and F, respectively. The 1080sF formats with frame rates of 24 and 24 / M have no “look-alike” 1080i formats.

Table 9-2:  Segmented Frame Video Formats from RP 211

<table>
<thead>
<tr>
<th>Format</th>
<th>1080sF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Rate (Hz)</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>30 / M</td>
</tr>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>24 / M</td>
</tr>
<tr>
<td>Sample Rate (MHz)</td>
<td>74.25</td>
</tr>
<tr>
<td></td>
<td>74.25 / M</td>
</tr>
<tr>
<td></td>
<td>74.25</td>
</tr>
<tr>
<td></td>
<td>74.25</td>
</tr>
<tr>
<td></td>
<td>74.25 / M</td>
</tr>
<tr>
<td>Active Samples per Line and Active Lines per Frame (words x lines)</td>
<td>1920 x 1080</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
</tr>
<tr>
<td>Total Samples per Line and Total Lines per Frame (words x lines)</td>
<td>2200 x 1125</td>
</tr>
<tr>
<td></td>
<td>2200 x 1125</td>
</tr>
<tr>
<td></td>
<td>2640 x 1125</td>
</tr>
<tr>
<td></td>
<td>2750 x 1125</td>
</tr>
<tr>
<td></td>
<td>2750 x 1125</td>
</tr>
</tbody>
</table>
Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Table 9-2: Segmented Frame Video Formats from RP 211

<table>
<thead>
<tr>
<th>Corresponding 1080p Format in Table 9-1</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Look-alike” 1080i Format in Table 9-1</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

HD-SDI Data Format

The high-definition video formats supported by HD-SDI keep the luma and chroma information in separate channels. The parallel video interfaces for these video standards carry the Y and C channels on separate wires. HD-SDI treats these two channels separately, but then interleaves them before encoding is done.

Each video line sent over an HD-SDI interface is divided into four areas as shown in Figure 9-1. The line begins with an EAV (end of active video) timing reference. The EAV includes the line number of the current line and a CRC value for the previous line. Following the EAV is the horizontal blanking interval. The horizontal blanking interval contains either blanking level video or ancillary data carrying information such as embedded digital audio. Following the horizontal blanking interval is the SAV (start of active video) timing reference followed immediately by the active portion of the line. The active portion of the line contains active video samples (two words per sample) if the line is an active line or it contains blanking level or ancillary data if the line is part of the vertical blanking interval.

Figure 9-1 shows just one of the two channels, either Y or C. Each channel has its own EAV, horizontal blanking, SAV, and active areas. The two channels are considered to be synchronous. For example, the Y word of sample zero is present on the Y channel at the same time that the C word of sample zero is present on the C channel.

The number of words in the horizontal blanking interval and the active portions of each video line are dependent upon the video format being transported. Refer to Table 9-1 for the number of words in these two regions for any supported video format.
The formats of the EAV and SAV timing references are shown in Figure 9-1. The first three words of these timing references are always fixed values: 3FF\text{H}, 000\text{H}, 000\text{H}. This sequence of three words is unique in the video stream and can occur only at the beginning of an EAV or SAV. The fourth word of the timing reference is commonly called the XYZ word and contains various flags that are used to describe the attributes of the current line. One bit in the XYZ word (the H bit) distinguishes between EAV and SAV. The format of the XYZ word is shown in Figure 9-2.

The EAV timing reference is followed by a two-word line number field called LN. The line number, required by HD-SDI, is used by the receiving equipment to synchronize more quickly to the video stream. The 11-bit line number value is encoded into the two 10-bit words as shown in Figure 9-3. Note that each channel, Y and C, has separate LN fields. The values of the line number for both Y and C should always be the same. Line numbers are assigned sequentially beginning with 1. The definition of which line in the frame is line 1 is video format dependent as specified in the SMPTE standard for each video format.

Two words that immediately follow the second LN word contain a cyclic redundancy code (CRC) for the previous line. The HD-SDI receiver uses the CRC value to detect transmission errors. As with the LN field, the Y and C channels each have their own CRC values. The CRC values for the two channels are calculated separately. Each CRC value is calculated using the following CRC polynomial:

\[ \text{CRC} = x^{18} + x^{5} + x^{4} + 1 \]

To calculate the CRC for a line, the initial CRC value is cleared to zero then the CRC is formed by including the first word of the active video portion of the line through and including the second word of the LN field. Figure 9-4 shows the words included in each
CRC calculation. Note that the CRC calculation does not include the two CRC words, the horizontal blanking interval, or the SAV. The SAV has its own parity bits that can be used to detect a corrupted SAV. Ancillary data packets inserted into the horizontal blanking interval usually have a CRC or checksum for each packet.

Figure 9-4: **CRC Calculation**

Figure 9-5 shows how the 18-bit CRC value is formatted into two 10-bit words when included into the HD-SDI video stream.

```
<table>
<thead>
<tr>
<th>b9</th>
<th>b8</th>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC word 0:</td>
<td>Not b8</td>
<td>CRC8</td>
<td>CRC7</td>
<td>CRC6</td>
<td>CRC5</td>
<td>CRC4</td>
<td>CRC3</td>
<td>CRC2</td>
<td>CRC1</td>
</tr>
<tr>
<td>CRC word 1:</td>
<td>Not b8</td>
<td>CRC17</td>
<td>CRC16</td>
<td>CRC15</td>
<td>CRC14</td>
<td>CRC13</td>
<td>CRC12</td>
<td>CRC11</td>
<td>CRC10</td>
</tr>
</tbody>
</table>
```

Figure 9-5: **CRC Format**

**Channel Interleaving**

After the line number and CRC fields are inserted into the two channels and prior to encoding the data, the Y and C channels are interleaved together as shown in Figure 9-6. For each sample, the word from the C channel is always sent first followed by the word from the Y channel.

Figure 9-6: **Interleaved Data Stream**
Encoding

The two-stage encoding scheme used by HD-SDI is identical to the one used by SD-SDI. First, a pseudo-random scrambler is used to scramble the data. Next, the non-return to zero (NRZ) data from the scrambler is converted to non-return to zero inverted (NRZI). The pseudo-random scrambler is very much like a linear feedback shift register (LFSR). The polynomial implemented by the scrambler is shown below as \( G_1 \), and the polynomial implemented by the NRZ-to-NRZI converter is shown as \( G_2 \):

\[
G_1(X) = X^9 + X^4 + 1
\]

\[
G_2(X) = X + 1
\]

NRZI bitstreams, such as HD-SDI, have an interesting property: they are polarity free. This means that the HD-SDI bitstream can be inverted between the transmitter and the receiver, and the receiver still correctly receives and decodes the data.

Figure 9-7 shows conceptually how the encoder would work if implemented serially. The boxes represent flip-flops and the \( \oplus \) symbols represent XOR gates. In the reference designs given in this chapter, the SDI encoder is implemented in a parallel fashion, encoding 20 bits per clock cycle.

**HD-SDI Transmitter Requirements**

This section describes the electrical requirements for the HD-SDI coaxial interface.

**Electrical Requirements**

The SMPTE 292M HD-SDI coaxial interface specifies the use of 75\( \Omega \) coax cable with BNC connectors. The BNC connectors used for HD-SDI usually have an impedance of 75\( \Omega \) although the specification does permit the use of 50\( \Omega \) connectors. When 75\( \Omega \) BNC connectors are used, they must be mechanically compatible with the 50\( \Omega \) BNC connector type defined by IEC 60169-8 [Ref 2].

The HD-SDI transmitter is unbalanced (singled-ended) with a source impedance of 75\( \Omega \). The transmitter must have a return loss of at least 15dB over a frequency range of 5 MHz to 1.485 GHz. Return loss is a measurement of the amount of signal that is absorbed by the transmitter when a reflected wave reaches the transmitter. The higher the return loss, the lower the amount of the reflected wave that is, in turn, reflected back to the receiver.

The transmitter signal amplitude is required to be 800mV \( \pm \)10%. The DC offset must be 0.0V \( \pm \)0.5V. This DC offset requirement implies that the transmitter is AC coupled to the coax cable.

The rise and fall times (20% to 80% amplitude) are required to be no slower than 270 ps. They cannot differ from each other by more than 100 ps. The overshoot of the rising and falling edges cannot exceed 10% of the amplitude of the signal.
Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Jitter Requirements

The SMPTE 292M specification places certain output jitter requirements on the HD-SDI transmitter. SMPTE 292M separates jitter into two bands called timing jitter and alignment jitter. Timing jitter includes all jitter frequency components from 10 Hz to 148.5 MHz. Alignment jitter is high-frequency jitter and begins at 100 kHz and goes to 148.5 MHz. As shown in Figure 9-8, both the timing and alignment jitter pass bands roll off at 20 dB / decade above and below the frequency limits just mentioned. Note that alignment jitter is a subset of the timing jitter.

![HD-SDI Transmitter Output Jitter Bands](image)

Figure 9-8: HD-SDI Transmitter Output Jitter Bands

The HD-SDI specification requires that the transmitter must have a timing jitter output of less than 1.0 UI\(^1\). Also, the total peak-to-peak alignment jitter must be less than 0.2 UI.

Usually, video equipment with HD-SDI interfaces have output jitter specifications well below the jitter requirements given in the HD-SDI spec, which allows for more jitter margin in the complete HD-SDI link. Good quality HD-SDI transmitters have timing jitter specifications of under 0.15 UI and alignment jitter specifications approaching 0.1 UI.

Cable Driver

The HD-SDI coaxial interface is single-ended, not differential. The RocketIO output driver is a differential driver. The results of trying to use just one side of the RocketIO output to

---

1. The term UI stands for Unit Interval and is equal to one bit period. For an HD-SDI interface running at 1.485 Gb/s, 1.0 UI is equal to about 673 ps.
drive the cable, even if the unused side is terminated to a 75Ω load, are unlikely to be acceptable. Trying to meet all the other electrical requirements of the HD-SDI transmitter, such as return loss and rise and fall times, by driving the coax cable directly from the RocketIO can be challenging. The RocketIO transmitter was designed to drive PCB traces that are, at most, about one meter long, whereas the HD-SDI specification allows for coax cable lengths of up to 100 meters. Therefore, Xilinx recommends the use of a cable driver specifically designed to meet the HD-SDI standards to interface the RocketIO output to the video coax cable.

The Gennum GS1528 cable driver is an example of such an HD-SDI compliant cable driver. The GS1528 can meet the requirements of both SD-SDI and HD-SDI. It has a rate control input that dynamically selects between HD and SD modes. This control signal changes the slew rate of the cable driver since the rise and fall time requirements of SD-SDI differ from those of HD-SDI.

This GS1528 cable driver has 3.3V LVPECL differential inputs. The CML output of the RocketIO transceiver is a 2.5V driver and is not LVPECL compatible. AC coupling must be used between the RocketIO driver and the input of the GS1528 in order to shift the signal levels from the 2.5V CML levels to the LVPECL levels. Figure 9-9 shows an example of interfacing the GS1528 to a RocketIO transmitter. Large AC coupling capacitor values, in the range of 1 μF to 4.7 μF, must be used in order to successfully pass the worst-case waveforms that can be generated by the HD-SDI and SD-SDI encoders. The rate control input pin of the GS1528 must be Low for HD-SDI. The RocketIO transceiver’s transmitter termination voltage (VTTX) should be set to 2.5V. The output impedance of the transceiver should be set to 50Ω.

![Figure 9-9: Interfacing the GS1528 Cable Driver to the RocketIO Transmitter](image)

Meeting the 15dB return loss requirement on the output of the cable driver requires careful layout of the cable driver and the output network between the cable driver and the BNC connector. Xilinx recommends that you carefully follow the recommendations from the cable driver manufacturer.

Clocks

One of the most important aspects of implementing an HD-SDI transmitter using the RocketIO transceivers is providing the right clocks to the transceivers. The RocketIO transceiver requires two types of clocks, reference clocks and user clocks. The reference

Audio/Video Connectivity Solutions
XAPP514 (v3.0) August 31, 2006
clocks are used to generate the bit-rate clock for the serializer. The user clocks are used to clock data from the fabric of the FPGA into the RocketIO transceiver.

The following sections contain descriptions of the clocking requirements of the RocketIO transceivers oriented towards implementing HD-SDI interfaces. More details about the clocking requirements of the RocketIO transceivers can be found in the RocketIO Transceiver User Guide [Ref 4].

For most HD-SDI transmitter applications, both the reference clock and user clocks are all derived from the parallel video clock running at either 74.25 MHz or 74.1758 MHz.

Reference Clocks

The reference clocks provide low-jitter frequency reference sources for the RocketIO transceiver. The RocketIO transceiver multiplies the selected reference clock by 20 to obtain a bit-rate clock for the transmitter’s serializer. Any jitter below about 10 MHz present on the reference clock is passed through to the transmitter output with little attenuation. Thus it is very important to use a low-jitter reference clock source.

As shown in Figure 9-10, each RocketIO transceiver has four reference clock inputs, although only one reference clock can be selected at a time. The four reference clocks are BREFCLK, BREFCLK2, REFCLK, and REFCLK2. Inside every RocketIO transceiver is a set of MUXes that select one of these four reference clocks. The MUXes are controlled by two signals:

- The REF_CLK_V_SEL signal is a configuration attribute of the RocketIO transceiver and can be changed only by reconfiguration.
- The REFCLKSEL signal is an input port to the RocketIO transceiver and can be controlled dynamically.

Keep in mind that although there are four reference clock inputs, you can switch only between two of them without reconfiguring the RocketIO transceiver. At configuration time, you must choose between using the REFCLKs or the BREFCLKs.

There is a significant difference between the REFCLKs and the BREFCLKs. The BREFCLK inputs provide the lowest jitter input path for the reference clocks, but at the cost of limited flexibility. The REFCLK inputs offer more flexibility, but at the cost of increased jitter.

The BREFCLK inputs use specific pins on the FPGA, and they can only use 2.5V differential I/O standards. The two BREFCLKs have dedicated routing resources inside
the FPGA from the IOBs to the RocketIO transceivers. Every Virtex-II Pro device has one BREFCLK input and one BREFCLK2 input on the top edge of the FPGA, and these inputs are only connected to the RocketIO transceivers along the FPGA’s top edge. Likewise, the bottom edge of the FPGA has one BREFCLK input and one BREFCLK2 input, and these inputs are connected only to the RocketIO transceivers on the bottom edge of the FPGA.

On the other hand, the REFCLK inputs do not use specific FPGA pins. They are connected through programmable routing resources inside the FPGA. The REFCLKLs can be used only when running the RocketIO transceivers at rates less than 2.5 Gb/s. The BREFCLK inputs must be used for bit rates above 2.5 Gb/s, but can be used at slower bit rates.

Because HD-SDI has two different bit rates, the RocketIO transceiver must have reference clocks for each of the supported bit rates. A reference clock source, external to the FPGA, must provide both 74.25 MHz and 74.25 / 1.001 MHz reference clock frequencies to the FPGA if both HD-SDI bit rates are supported. These two frequencies could come into the FPGA as one signal with something external to the FPGA selecting between them, or they could come into the FPGA as two separate clock sources, in which case the RocketIO reference clock MUX would be used to select between them.

It is very important that the reference clock has very low jitter. If this clock has too much jitter, the output jitter of the RocketIO transceivers might exceed the HD-SDI specifications. At HD-SDI speeds, Xilinx recommends that the peak-to-peak jitter of the reference clock should be less than 100 ps. On the Xilinx SDV demo board, the used reference clock sources had less than 40 ps peak-to-peak jitter.

If the reference clock MUX in the RocketIO is used to select between the two HD-SDI reference clock frequencies, you are highly recommended to use the BREFCLK inputs rather than the REFCLK inputs. Because the two HD-SDI reference clock frequencies differ by just 74.25 kHz, our experience has shown that these two clock sources tend to mix (heterodyne) if the REFCLK inputs are used. This results in excessive output jitter on the transmitter output with a large jitter component at 74.25 kHz. However, if the BREFCLK inputs are used, this mixing does not occur and the output jitter is much lower.

If only a single reference clock input is required for the RocketIO transceiver, such as when an external frequency synthesizer generates the reference clocks, then either the BREFCLK or REFCLK inputs can be used, producing about the same amount of output jitter.

One further recommendation is to use clock sources with differential LVDS or LVPECL outputs. The use of differential I/O standards results in significantly less jitter than when single-ended I/O standards are used for the reference clocks. In fact, the BREFCLK inputs must be differential.

**User Clocks**

The user clocks load data into the RocketIO transceiver from the fabric of the FPGA. Each RocketIO transceiver requires two user clocks on the transmitter side called TXUSRCLK and TXUSRCLK2. Each RocketIO transceiver also has two user clocks for the receiver called RXUSRCLK and RXUSRCLK2. If the receiver is not used, RXUSRCLK and RXUSRCLK2 still should be driven with valid clock signals. The easiest thing to do in this case is to connect RXUSRCLK to TXUSRCLK and RXUSRCLK2 to TXUSRCLK2.

TXUSRCLK always must be frequency-locked to the selected reference clock. There is no required phase relationship between TXUSRCLK and the selected reference clock, but they must have the same frequency.

The frequency and phase relationships between TXUSRCLK and TXUSRCLK2 depend on the width of the TXDATA input port of the RocketIO transceiver. For HD-SDI, it is usually most convenient to use a 20-bit wide TXDATA port because this matches the data word
width of HD-SDI (10 bits of Y and 10 bits of C). When using a 20-bit TXDATA port, TXUSRCLK2 must have the same frequency and phase as TXUSRCLK (simply connect TXUSRCLK and TXUSRCLK2 to the same clock source). Consult the RocketIO Transceiver User Guide for TXUSRCLK2 requirements when other TXDATA port widths are used.

The user clocks do not have to be low jitter. They are often connected to outputs of digital clock managers (DCMs) due to frequency and phase relationship requirement between TXUSRCLK and TXUSRCLK2 when using TXDATA port widths other than 20 bits.

When using two reference clock sources for HD-SDI, it is necessary to change the frequency of the TXUSRCLK and TXUSRCLK2 inputs when the reference clock source is changed. Figure 9-11 shows an example of the RocketIO clock connections when a single reference clock input is used and jitter reduction is being done on the reference clock by an external PLL. Figure 9-12 shows how to connect the reference clocks and user clocks when two clock sources are used.

Jitter Reduction

In many cases, the parallel video clock associated with the input parallel video stream might have too much jitter to be used directly as a reference clock to the RocketIO transceiver. For example, the SMPTE parallel interface standards for HD video, such as SMPTE 274M, are allowed to have jitter as high as 2 ns peak-to-peak on the parallel video clock. This is more than 10 times the acceptable amount of jitter for the reference clocks for the RocketIO transceiver. The SMPTE 292M standard specifically warns that jitter reduction must be implemented on this type of high-jitter parallel video interface before the video can be transmitted using HD-SDI.

Also jitter reduction is usually required when connecting the output of an HD-SDI receiver to an HD-SDI transmitter. The HD-SDI receiver recovers a clock from the incoming HD-SDI bitstream. Since the input bitstream might have a considerable amount of jitter after travelling down a long coax cable, the recovered clock also might have a great deal of jitter. It usually is not possible to directly use the recovered clock from receiver as the reference clock to the transmitter without reducing the jitter on the recovered clock.

Figure 9-11 shows a typical implementation of jitter reduction. The high-jitter parallel video clock passes through a PLL that is optimized for the desired jitter reduction characteristics. The high-jitter video data is written into an asynchronous FIFO using the high-jitter clock. Then the data is read out of the FIFO using the low-jitter clock from the PLL, thereby resynchronizing the data to the low-jitter clock.

![Figure 9-11: Jitter Reduction](image-url)
The RocketIO transmitter has a shallow FIFO on its input data path that is perfectly suited for implementing the asynchronous FIFO function shown in Figure 9-11. However, the Virtex-II Pro FPGA does not contain a PLL that can be used for clock jitter reduction. Thus, it is necessary to use an external PLL to implement jitter reduction on the video clock in those cases where jitter reduction is required. Any PLL circuit used for jitter reduction must produce an output that has 100 ps peak-to-peak of jitter, at most.

Reference Designs

The reference design is described at a top level in the following section. Detailed information about the reference design can be found in “Appendix: Reference Design Details.”

Most of the HD-SDI transmitter is contained in the module called hdsdi_tx. This module contains all of the formatting and encoding logic necessary for the HD-SDI transmitter, but it does not contain the RocketIO transceiver. The RocketIO transceiver is contained in another module called hdsdi_rio. The reason for keeping the RocketIO module separate from the rest of the HD-SDI transmitter is so that the receiver side of the transceiver can also be connected to an HD-SDI receiver module, if needed.

An example of how to connect the hdsdi_tx and hdsdi_rio modules is given in the file named sdv_hdsdi_tx. This HD-SDI transmitter example was designed specifically for the Xilinx SDV demo board. In this example, the video source for the HD-SDI transmitter comes from a video pattern generator module called multigenHD. The multigenHD module is discussed in Chapter 17, “HDTV Video Pattern Generator.”

Figure 9-12 is a block diagram of the sdv_hdsdi_tx design.
Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Figure 9-13 shows a block diagram of the main HD-SDI transmitter module, hdsdi_tx. This module contains three submodules described in the following paragraphs.

The hdsdi_trs_decode module examines the incoming video stream and identifies when EAV and SAV timing references occur. The EAV and SAV signals from this module are used by the other modules in the HD-SDI transmitter for such things as video format detection, line number insertion, and CRC generation and insertion.

The hdsdi_autodetect_ln module examines the incoming video stream and determines the format of the video. This module can identify all video formats listed in Table 9-1 and Table 9-2. Once the module determines the format of the video, it begins generating line number information that can be inserted into the video as required by the HD-SDI standard. This module does not insert the line number information. Insertion is done in the hdsdi_tx_path module. The hdsdi_autodetect_ln module is optional. If the video already contains line number information, then this module is not required.

Note that in the sdv_hdsdi_tx reference design, the multigenHD video pattern generator produces line numbers but does not insert them into the video. The multigenHD module also provides essentially the same video timing information that hdsdi_trs_decode provides. The hdsdi_trs_decode and hdsdi_autodetect_ln modules really are not necessary when the video is provided by a module such as multigenHD. However, these modules are included in this reference design to illustrate how they are used when video timing and line number information is not available on the input of the HD-SDI transmitter.

The third module in hdsdi_tx is called hdsdi_tx_path (see Figure 9-14). This module contains the main data path of the HD-SDI transmitter. In a design where line number information and video timing is provided at the input to the HD-SDI transmitter, the hdsdi_autodetect_ln and hdsdi_trs_decode modules could be eliminated and only hdsdi_tx_path would be required.
The hdsdi_tx_path module performs four main functions: line number insertion, CRC generation, CRC insertion, and encoding. The line number information is formatted and inserted into the Y and C channels by the hdsdi_insert_ln module. Once line number information is inserted, a CRC is computed for each channel by the hdsdi_crc modules, and then the CRCs are formatted and inserted into the Y and C channels by the hdsdi_insert_crc module. After CRC insertion, the video is encoded by the hdsdi_encoder module. After encoding, the data is sent to the hdsdi_rio module for transmission.

Figure 9-15 shows the block diagram of the hdsdi_rio module. This module is wrapped around the RocketIO transceiver primitive (GT_CUSTOM). In addition to the RocketIO primitive, the module contains bit swap functions on the input and output data ports of the RocketIO primitive and a reset delay circuit for the RocketIO transceiver.

The RocketIO transceiver transmits the MSB of the TXDATA port first. Likewise, the RocketIO receiver outputs the first bit it receives on the MSB of the RXDATA output port. HD-SDI always transmits the LSB first, just the opposite of how the RocketIO transceiver operates. The 20-bit output of the HD-SDI encoder must be bit swapped before being connected to the TXDATA port of the RocketIO primitive so that the LSB of the encoder is connected to the MSB of the RocketIO transceiver’s TXDATA port. The RocketIO receiver’s RXDATA output port also is bit swapped before leaving the hdsdi_rio module.

The TXRESET input of the GT_CUSTOM primitive must remain asserted for at least two TXUSRCLK cycles after all clock inputs become stable. The hdsdi_rio primitive contains some logic to keep the TXRESET input asserted until several clock cycles after the dcm_locked input becomes asserted. This input is called dcm_locked because it usually is driven from the LOCKED output of the DCM generating the TXUSRCLK and RXUSRCLK clock inputs to the RocketIO transceiver. If a DCM is not used to generate these clock signals, then the dcm_locked input can be connected to another appropriate signal that indicates when the clocks are stable or it can be tied High if the clocks are always running.
Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Jitter Performance

The output jitter of the HD-SDI transmitter is dependent upon the amount of jitter present on the reference clock, noise coupled into the reference clock from other sources (such as the other reference clock input), the intrinsic jitter of the RocketIO transceiver, and jitter added by the cable driver and board layout.

The Xilinx SDV demo board [Ref 5] was used to measure the typical transmitter output jitter of an HD-SDI transmitter implemented using the Virtex-II Pro device. This board uses low-jitter crystal oscillators for reference clocks to the RocketIO transceiver. The demo board uses a Gennum GS1528 cable driver to interface the RocketIO transceiver to the BNC connector.

The typical HD-SDI transmitter jitter values that were measured on the SDV demo board were verified over 30 different boards using several different HD-SDI waveform analyzers including the Tektronix WFM700 and the SyntheSys Research HD292 and MVA3000 analyzers. Figure 9-16 is the eye diagram display from a SyntheSys Research MVA3000 analyzer showing the output waveform and jitter measurements from a Xilinx SDV demo board running the HD-SDI transmitter reference design.

All jitter measurements were made with the SDV demo board connected to the analyzer using a 1-meter long, 75Ω coax cable. The HD-SDI transmitter was sending a 75% color bar pattern. The jitter values reported in Table 9-3 were the worst numbers measured across all 30 boards tested under room temperature conditions and nominal voltage conditions. These jitter numbers have been verified even when the FPGA was filled so that about 95% of the fabric, block RAM, and multipliers were being toggled actively and random data...
was output on about 50% of the IOBs. The reference clocks to the RocketIO transceiver had no more than 40 ps peak-peak of jitter.

Table 9-3: Typical HD-SDI Transmitter Output Jitter Values

<table>
<thead>
<tr>
<th>Jitter Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing Jitter</td>
<td>0.133 UI</td>
</tr>
<tr>
<td>Alignment Jitter</td>
<td>0.107 UI</td>
</tr>
</tbody>
</table>

Reference Design Size

Table 9-4 shows the FPGA resources used by the HD-SDI transmitter reference design. Two implementation sizes are shown, one with line number generation and insertion and one without. As can be seen by the results, the line number generation and insertion accounts for about half of the size of the design when included.

In both cases, the results were obtained using XST running under ISE 6.1. Area optimization was used. Both designs met the necessary timing constraints using a Virtex-II Pro -5 speed grade device.

The multiogenHD video pattern generator module, included in the sdv_hdsdi_tx reference design is not included in the reference design results shown in Table 9-4.
Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Conclusion

This chapter describes the implementation details of an HD-SDI transmitter using the RocketIO transceivers available in the Virtex-II Pro FPGA family. The RocketIO transceivers combined with an HD-SDI encoder and other support functions built in the fabric of the FPGA can easily implement an HD-SDI transmitter.

The HD-SDI transmitter reference design module requires very few resources in the FPGA, thus making it quite easy to implement multiple HD-SDI interfaces in even the smallest member of the Virtex-II Pro family or to integrate video processing functions and an HD-SDI transmitter all in the same part.

Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_hd-tx-mgt.zip.

Appendix: Reference Design Details

This appendix contains detailed design information for the hdsdi_autodetect_ln module and the hdsdi_encoder module.

Video Format Detection and Line Number Generation

If the video entering the HD-SDI transmitter does not contain line numbers after each EAV, then it necessary for the HD-SDI transmitter to decode the video stream and generate and insert line numbers into the video prior to transmission.

The module hdsdi_autodetect_ln is an HD video decoder. It can determine the incoming video format and generate line numbers. This module does not insert the line numbers. Line number insertion is done by the hdsdi_insert_ln module inside of hdsdi_tx_path.

The hdsdi_autodetect_ln module contains a finite state machine (FSM) that analyzes the incoming video and determines which video format the input video stream matches. The FSM counts the number of words per line and the number of lines per field (or frame for progressive scan formats). The FSM then compares these word and line counts to the known video formats. If it finds a match, it generates a 4-bit code indicating which video format was detected.

Once the FSM is locked to a video format, it generates line numbers for that video format. The line number generator looks for the first active line of a field. The first active line of a field is easy to detect because the V bit in the EAV symbol transitions from a 1 in the last line of the video blanking interval to a 0 in the first active line. When the first active line of a field is detected, the line number counter is loaded from a small ROM that contains the first active line number of each field for all of the known video formats. Once loaded from

<table>
<thead>
<tr>
<th>Reference Design</th>
<th>FFs</th>
<th>LUTs</th>
<th>Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>With LN generation and insertion</td>
<td>197</td>
<td>313</td>
<td>190</td>
</tr>
<tr>
<td>Without LN generation and insertion</td>
<td>126</td>
<td>144</td>
<td>85</td>
</tr>
</tbody>
</table>
this ROM, this line counter increments every time an EAV is detected unless the counter has reached the last line of the frame, in which case it is reloaded with a value of 1. Another ROM provides the maximum line number for each of the known video formats, so that the line counter can rollover to 1 at the appropriate time.

Figure 9-17 shows a block diagram of the hdsdi_autodetect_in module, and Figure 9-18 is the state diagram for the FSM.
Chapter 9: HD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

The FSM has two main loops, the acquire loop (ACQ) and the locked loop (LCK). Initially, the FSM starts in the ACQ loop, trying to match the video to a known format. As soon as the state machine sees the XYZ word of an SAV, and the current line is the first active line of a field (as indicated by the V bit transitioning from High to Low), the state machine

Figure 9-18: hdsdi_autodetect_in State Diagram

The FSM has two main loops, the acquire loop (ACQ) and the locked loop (LCK). Initially, the FSM starts in the ACQ loop, trying to match the video to a known format. As soon as the state machine sees the XYZ word of an SAV, and the current line is the first active line of a field (as indicated by the V bit transitioning from High to Low), the state machine
enables the word and line counters. The word counter increments every clock cycle and the line counter increments once per line. Words are counted until the XYZ word of the next SAV, at which time the word counter contains the total number of words per video line. Line counting continues until the next vertical blanking interval begins (V goes High), at which time the line counter contains the active lines per field (or frame if the video is progressive).

After counting the number of words and lines, the FSM moves to state ACQ4 where it sequentially compares the measured words and line counts to the corresponding values for each known video format. The “loops” counter is at zero when the FSM enters ACQ4. This counter provides the address to the ROMs that contain the word and line counts for each known format. The FSM stays in ACQ4, incrementing the loops counter to sequence through all of the known video formats, comparing the acquired words and lines value to the known values of each format, until either a match is found or the loops counter reaches its terminal count. If a match is found, the std register is loaded with the value of the loops counter so that it contains the code indicating the matching video format and the FSM moves to the LCK loop. If no match is found and the loops counter reaches its terminal count, the FSM moves back to state ACQ0 and begins the process again.

The LCK loop is similar to the ACQ loop. In the LCK loop, the FSM continuously counts the number of words per line and active lines per field and compares them to the known values for the current video format. If a mismatch is encountered, the FSM moves to the ERR state and the error counter is incremented. If the error counter reaches the MAX_ERRS value, then the FSM returns to the ACK loop, to acquire the new format.

Note that the error counter is reset in state LCK3 if a successful match is made between the measured words and lines and the expected values. Thus, the FSM must see MAX_ERRS consecutive fields with errors before it return to the ACQ loop. In some cases, other modules within the design could know when the video format changes. In this case, it is possible to immediately move the FSM to the ACQ loop in order to avoid the MAX_ERRS fields latency that the FSM normally waits before moving to the ACQ loop. This is done by asserting the reacquire input to the FSM. This input forces the FSM to the ACQ0 state whenever it is asserted High.

The hdsdi_autodetect_ln module can detect all the video formats shown in Table 9-1 and Table 9-2. However, several of the 1080sF formats cannot be distinguished from their “look-alike” 1080i formats by word and line counting. For the purpose of generating line numbers, however, it is not necessary to distinguish between the 1080sF formats and their corresponding 1080i formats.

HD-SDI Encoder

HD-SDI encoding is implemented in the hdsdi_encoder module. This encoder has control inputs that enable and disable both parts of the encoding algorithm: the scrambler and the NRZ-to-NRZI converter. For normal operation, these enable inputs should always be High. The scrambler and the NRZ-to-NRZI converter can be disabled for diagnostic purposes. For instance, if both of them are disabled, the video stream is directly serialized without any encoding, making it easier to determine if other portions of the transmitter are working correctly in simulation or by using a logic analyzer.

As described earlier in this chapter, the HD-SDI bitstream is interleaved and alternately contains a 10-bit word from the C channel followed by a 10-bit word from the Y channel. If the HD-SDI encoder ran at 2X the HD-SDI word rate (148.5 MHz or 148.5 / 1.001 MHz) then a single 10-bit encoder module could be used along with a MUX alternately feed C and Y channel words into the encoder. However, the hdsdi_encoder module has 20-bit input and output data paths and runs at the HD-SDI word rate. As shown in Figure 9-19,
this module contains two instances of a 10-bit encoder module called smpte_encoder. One smpte_encoder module encodes the C channel and the other encodes the Y channel. These two modules are interconnected so that encoded bits from the C channel encoder affect the encoding of the Y channel word (remember that each Y word is sent after the corresponding C word and the encoding results from the C word affect the encoding of the Y word). Likewise, the encoded results from the Y word are saved in a register and affect the encoding of the C word during the next clock cycle.

The smpte_encoder module implements the HD-SDI scrambling and NRZ-to-NRZI encoding algorithms. It operates at the word rate and encodes one 10-bit word per clock cycle. The smpte_encoder module has a two clock cycle latency and each 10-bit video word is processed in two stages. The first stage of the encoder is the scrambler and the second stage is the NRZ-to-NRZI converter. Figure 9-20 is a block diagram of the smpte_encoder module.
Chapter 10

HD-SDI Receiver Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Summary

The High-Definition Serial Digital Interface (HD-SDI) standard describes how to transport high-definition (HD) digital video serially over video coax cable. HD-SDI is used to connect HD video equipment in broadcast studios and video production centers. It is an evolution of the popular SDI standard that is widely used to transport standard-definition (SD) digital video in the broadcast industry.

The flexibility of the RocketIO™ multi-gigabit transceivers available in the Virtex™-II Pro family devices combined with the programmable logic of the Virtex-II Pro FPGAs makes it possible to implement HD-SDI interfaces. Because every Virtex-II Pro FPGA has multiple RocketIO transceivers, multiple HD-SDI interfaces can be integrated into one Virtex-II Pro device along with other video processing functions.

This chapter describes how to implement HD-SDI receivers. An HD-SDI receiver built in a Virtex-II Pro FPGA is presented as a reference design.

Introduction

Use of HD-SDI, defined by the SMPTE 292M standard, is increasing rapidly in broadcast studios and video production centers as the broadcast industry ramps up support for HDTV broadcasting [Ref 1].

HD-SDI builds upon the widely used SDI standard for transporting SD digital video. The older SDI standard is referred to as SD-SDI to differentiate it from HD-SDI. The SD-SDI and HD-SDI standards share the same electrical characteristics and encoding scheme. However, HD-SDI uses a higher bit rate to accommodate the higher bandwidth requirements of uncompressed HD digital video signals. Because SD-SDI and HD-SDI share common electrical characteristics, it is possible to build video equipment that can support both standards through a single connection.

This chapter discusses the HD-SDI receiver; Chapter 9 describes how to implement the HD-SDI transmitter.

The HD-SDI standard supports both coax cable and optical fiber interfaces. Coax cable has been the more popular of the two due to lower cost and commonality with SD-SDI. This chapter only discusses the implementation details for the coaxial cable interface. However, since the data formats and encoding schemes for the optical interface option are identical to the coaxial interface option, the reference design presented in this chapter is directly applicable to implementing an HD-SDI receiver with an optical fiber interface.
The dual link HD-SDI standard, SMPTE 372M, uses two HD-SDI interfaces to provide twice the bandwidth, allowing higher bandwidth video formats to be supported. The SMPTE 372M standard is not specifically addressed in this chapter, but the HD-SDI reference design described here can be used as a building block for implementing a dual-link HD-SDI interface.

**HD-SDI Receiver Functions**

This section describes the basic functions implemented by an HD-SDI receiver. Figure 10-1 is a block diagram of a typical HD-SDI receiver. Refer to the HD-SDI Data Format section of Chapter 9 for a description of the video formats supported by HD-SDI and the details of the format of the HD-SDI bitstream.

![HD-SDI Receiver Block Diagram](image)

**Cable Equalization**

HD-SDI uses two bit rates: 1.485 Gb/s and 1.485 / 1.001 Gb/s (approximately 1.4835 Gb/s). The HD-SDI bitstreams are sent serially using an unbalanced (single-ended) driver over 75Ω coaxial cable up to 100 meters in length.

The coax cable causes frequency-dependent attenuation of the signal, where the higher frequency components of the signal are attenuated more than the lower frequency components. The coax cable also causes frequency-dependent phase distortion, where the higher frequency components are phase shifted more than lower frequency components. After passing through 100 meters of coax cable, the HD-SDI signal is severely distorted and attenuated. The receiver must compensate for this attenuation and distortion before attempting to recover the signal.

Cable length equalization is used to compensate for the attenuation and distortion introduced by the coax cable. The SMPTE 292M HD-SDI standard states that receivers typically work with an attenuation of 20 dB at one-half the clock rate. Because this is not a requirement, the standard permits HD-SDI receivers that cannot recover a signal with 20 dB of attenuation.

Typically, an adaptive cable length equalizer is used in HD-SDI receivers. Such an equalizer actively monitors the amount of attenuation and distortion present on the incoming signal and applies the correct amount of equalization to the signal. The cable
length is allowed to change without requiring a change to the equalizer, as would be the case if fixed length equalization were used.

Clock and Data Recovery

After cable equalization, the HD-SDI receiver recovers the clock and data from the HD-SDI bitstream. This recovery typically is done with a PLL-based clock and data recovery (CDR) unit. A recovered clock usually is required for an HD-SDI receiver because the HD-SDI protocol has no provisions for clock correction to allow the incoming bitstream to be easily resynchronized to a local reference clock. Instead, the recovered clock from the CDR unit generally is used to clock all HD-SDI receiver logic downstream from the CDR unit.

When building an HD-SDI receiver using Virtex-II Pro devices, the RocketIO transceiver implements the CDR function and also deserializes the bitstream. The RocketIO transceiver provides a recovered clock that runs at the HD-SDI word rate (1/20th the bit rate). For HD-SDI, the recovered clock from the RocketIO transceiver runs at either 74.25 MHz or 74.25 / 1.001 MHz, depending on which bit rate is currently being received.

Decoding

As described in Chapter 9, HD-SDI uses a two-stage encoding algorithm, where the first stage performs pseudorandom scrambling and the second stage performs non-return-to-zero (NRZ) to non-return-to-zero-inverted (NRZI) conversion. After recovering the data, the HD-SDI receiver must decode it by reversing the two encoding steps: first it converts the NRZI data to NRZ, and then it undoes the pseudorandom scrambling. Figure 10-2 shows conceptually how the HD-SDI bitstream is decoded in a serial manner.

![HD-SDI Decoding Algorithm](image)

The RocketIO transceivers have built-in 8B/10B decoders. However, they do not have HD-SDI decoders. So, the recovered data from the RocketIO transceiver bypasses the decoding logic built into the RocketIO transceiver and is provided directly to the RXDATA port still encoded. The HD-SDI reference design described in this chapter implements the HD-SDI decoder in the fabric of the Virtex-II Pro FPGA. The data is decoded in a parallel manner, 20 bits per clock cycle.

Framing

The recovered data words from the CDR unit and from the HD-SDI decoder are not word aligned. The CDR unit has no concept of where the video sample boundaries are in the continuous stream of incoming bits. The decoder does not care where the sample boundaries are since it can decode the data without this information. However, after decoding, it is necessary to identify the sample boundaries and realign the data so that each 20-bit sample is properly aligned and contains a 10-bit Y word and a 10-bit C word. This process of realigning the data is called framing.

The framer in the HD-SDI receiver monitors the incoming data and looks for the bit sequences that mark the beginning of the timing references. There are two timing...
references per video line: the end-of-active video (EAV) and the start-of-active video (SAV).
Both the EAV and SAV have the same format and are four 10-bit words long. The first three
words are always fixed values. The first word of the timing reference is a word of all ones
and has a hex value of 3FFH. The second and third words of the timing reference are made
up of all zeros (000H). The fourth word of the timing reference is called the XYZ word.
Figure 10-3 shows the format of the XYZ word of the timing reference. The sequence of 10
‘1’ bits followed by 20 ‘0’ bits that marks the beginning of each timing reference is unique
in the HD-SDI video stream and can occur only at the beginning of the timing reference.

HD-SDI divides the video stream into separate channels called the luma (Y) channel and
the chroma (C) channel. Each channel has its own set of timing references. The channels are
considered to be synchronous so that the first word of the EAV, for example, would appear
on both the Y channel and the C channel at the same time.

Before transmission by the HD-SDI transmitter, the Y and C channels are interleaved so
that a C word is transmitted first followed immediately by the corresponding Y word.
Figure 10-4 shows the details of this interleaving.

The framer in the HD-SDI receiver must look for the unique 3FFh, 000h, 000h sequence
that marks the beginning of a timing reference. Only this unique pattern in the HD-SDI
bitstream can be used as a reference point for realigning the data. Due to the interleaving of
the Y and C channels, the framer sees the following sequence for each timing reference:
3FFh, 3FFh, 000h, 000h, 000h, 000h.

The framer looks for this unique pattern beginning at any possible bit position in the
recovered data coming from the HD-SDI decoder. Once this pattern is identified, the
framer knows the bit offset of the least significant bit of each sample in the data words coming from the decoder. A barrel shifter is used to realign each 20-bit sample.

Figure 10-5 shows how a framer correctly aligns the data from the decoder. The data going into the framer is unaligned and contains an EAV beginning at bit 12. The 20 ‘1’ bits and 40 ‘0’ bits that make up the first three words of the interleaved EAVs are shown in red. The 20 bits of the two XYZ words are shown in blue. After the framer, the data is realigned so that the first bit of the EAV is positioned as the least significant bit of the C channel.

In the HD-SDI receiver reference design, the framer function is implemented in the fabric of the Virtex-II Pro FPGA.

CRC Checking

After the HD-SDI video stream has been aligned by the framer, the receiver does cyclic-redundancy-code (CRC) checking to determine if any errors have occurred in the transmission of the data. Each video line contains an 18-bit CRC. The CRC is formatted into two 10-bit words located after the EAV of each line. The two words immediately after the XYZ word of the EAV contain the line number of the video line. The two words containing the CRC are located immediately after the line number words.

The Y and C channels each have their own CRCs. The receiver computes CRC values separately for both the Y and C channels as it receives a line of video. When the CRCs embedded in the video stream arrive, the receiver compares them to the CRCs that it has calculated. If the CRCs differ, an error has been detected.

Chapter 9 has more details on how the CRCs are computed and formatted.
Additional HD-SDI Receiver Functions

After CRC checking, the basic functions of the HD-SDI receiver are complete. Depending on the application, however, the HD-SDI receiver can also perform some additional functions. Some of these additional functions are described here.

The HD-SDI receiver can examine the video stream to determine its video format. The HD-SDI standard supports many different video formats. There are two ways to determine the video format:

- by the characteristics of the video itself (word/line counting) or
- by the finding a special ancillary data (ANC) packet that identifies the video format.

The SMPTE 352M standard specifies an ANC packet that can be used to uniquely identify the format of the video payload. However, if the video stream does not contain an SMPTE 352M payload ID packet, the video format can be identified by counting the number of words per line and lines per frame in the video. An example of such a video format detector is included in the reference design.

It is useful to derive some video timing information from the received video stream. A video decoder can generate various video timing signals, such as horizontal and vertical blanking, from the timing reference signals. A more sophisticated video decoder might implement a flywheel which keeps track of where the timing reference signals are expected, repairs defective timing references, and inserts timing references when they are missing. A simple video timing decoder is included in the reference design.

The SMPTE 352M video payload ID is one type of ancillary data that can be included in the horizontal and vertical blanking intervals of the HD-SDI data stream. Another common use of ancillary data packets is to carry embedded digital audio. Some HD-SDI receivers might need to detect certain types of ancillary data packets and separate that ancillary data from the main video stream. The general format of ancillary data packets is given in the SMPTE 291M standard. Ancillary data packets are easy to detect in the video stream because they begin with a unique three-word sequence, similar to the first three words of the timing reference. The first three words of an ancillary data packet are 000H, 3FFH, 3FFH.

HD-SDI Receiver Requirements

The SMPTE 292M document places a few requirements on the HD-SDI receiver. Basically, the receiver must be compatible with the single-ended, AC coupled electrical signal generated by the HD-SDI transmitter. The receiver must provide a 75Ω impedance to the cable interface with a return loss of at least 15 dB from 5 MHz to 1.485 GHz.

The SMPTE 292M standard states that it is typical for HD-SDI receivers to receive signals attenuated by up to 20 dB. Because this is not a requirement of the standard, receivers that cannot recover the data when the input signal has been attenuated by 20 dB are permitted.

The SMPTE 292M standard contains a jitter template describing the maximum amount of jitter that can be produced by the HD-SDI transmitter. HD-SDI receivers should have input jitter tolerance exceeding the maximum allowed transmitter jitter as described by the jitter template, although not specifically required by the standard. The amount by which the receiver exceeds the jitter template is the jitter margin of the receiver.

Figure 10-6 shows the SMPTE 292M jitter template. The horizontal axis is jitter frequency, and the vertical axis is jitter amplitude given in UI(1). The output jitter of an HD-SDI transmitter must be below the template for every jitter frequency within the specification. An HD-SDI receiver should be able to tolerate more jitter than the transmitter is allowed to produce at each jitter frequency. When the input jitter tolerance of an HD-SDI receiver is...
plotted onto Figure 10-6 (blue line), all points of the plot should be above the jitter template line. For any particular jitter frequency, the vertical distance between the receiver’s input jitter tolerance and the jitter template is the receiver’s jitter margin at that frequency.

![Jitter Template Diagram](image)

**Figure 10-6: SMPTE 292M Jitter Template**

SMPTE recommended practice RP 198 describes two worst-case pathological waveforms that can be produced by the HD-SDI encoder. One pathological waveform is poorly DC balanced and can cause problems with cable equalizers not designed to tolerate this waveform. Any cable equalizer designed specifically for HD-SDI use should be tolerant of this waveform. The second pathological waveform is essentially a low-frequency square wave consisting of 20 Low bits followed by 20 High bits. This square wave can repeat across the entire active portion of a video line. This low-frequency waveform can cause problems for the PLL inside of the CDR unit. The RocketIO CDR unit has been tested extensively with this waveform and is fully tolerant of it.

**Implementing the HD-SDI Receiver**

This section details how to implement an HD-SDI receiver using the RocketIO transceivers in Virtex-II Pro FPGAs. The reference design described here is implemented and tested on the Xilinx SDV Demo Board [Ref 2].

---

1. UI stands for Unit Interval. One UI is equal to the duration of one bit in the bitstream. For HD-SDI, one UI is about 673 ps.
Cable Equalizer

As previously described, an HD-SDI receiver usually has an adaptive cable length equalizer to compensate for attenuation and distortion of the signal caused by long runs of coax cable. The RocketIO transceivers in the Virtex-II Pro FPGA do not include adaptive cable length equalizers. So, an external cable equalizer must be used to interface the HD-SDI cable to the RocketIO receiver. As a side benefit, the cable equalizer also converts the single-ended HD-SDI signal into a differential signal. The CML inputs of the RocketIO receiver require a differential input signal. Most HD-SDI cable equalizers currently available have 3.3V LVPECL outputs that are not directly compatible with the 2.5V CML inputs of the RocketIO transceiver. AC coupling is used to interface the LVPECL outputs of the cable equalizer to the CML inputs of the RocketIO transceiver. Figure 10-7 shows a typical AC coupled interface between a Gennum GS1524 cable equalizer and a RocketIO receiver.

There are several important details in Figure 10-7:

- The recommendations given in the GS1524 data sheet [Ref 3] must be followed for the interface network between the BNC cable connector and the GS1524’s input.
- The GS1524 is a multi-rate cable equalizer capable of supporting both HD-SDI and SD-SDI. HD-SDI only cable equalizers are also available.
- The coupling capacitors between the GS1524 and the RocketIO receiver must be in the 1 μF to 10 μF range to pass the HD-SDI pathological waveforms without too much voltage drop. Typically, 4.7 μF capacitors are used.
- The input impedance of the RocketIO receiver must be set to 50Ω and the circuit board traces between the equalizer and the RocketIO receiver must have an impedance of 50Ω.
As described in the *RocketIO Transceiver User Guide* [Ref 4], when using AC coupling, the RocketIO receiver termination voltage (VTRX) must be between 1.6V to 1.8V. As shown in the figure, the required termination voltage can be generated from 2.5V using a voltage divider network. The resistor values shown are sized to supply the termination voltage to a single RocketIO receiver, so this resistor network must be duplicated for each RocketIO receiver used as an HD-SDI receiver.

In rare cases, it might not be necessary to use a cable equalizer. For example, if the HD-SDI bitstream is always sent over a very short length of cable or a backplane within a chassis, the transmission path length might be short enough that cable equalization is not required. In such cases, the incoming single-ended HD-SDI signal must be converted to a differential signal compatible with the CML inputs of the RocketIO receiver, unless the HD-SDI transmitter can be designed to provide a differential signal. While a differential HD-SDI signal is not within the HD-SDI specification, it would provide a superior solution inside of a proprietary chassis, especially when a cable equalizer is not used.

**RocketIO Transceiver Clocks**

The RocketIO transceiver requires two types of clocks: reference clocks and user clocks. The reference clocks are used by the RocketIO receiver as a reference for the CDR PLL. The user clocks are used to clock data out of the RocketIO receiver into the fabric of the FPGA. In addition, the RocketIO receiver also produces a recovered clock, called RXRECCLK.

The following sections describe the clocking requirements of the RocketIO transceivers oriented towards implementing HD-SDI interfaces. More details about the clocking requirements of the RocketIO transceivers can be found in the *RocketIO Transceiver User Guide*.

**Reference Clocks**

The RocketIO transceiver uses reference clocks for two different purposes:

1. In the transmitter, the reference clock provides a low-jitter frequency reference that the transmitter multiplies by 20 to obtain the bit-rate clock for the transmitter’s serializer.

2. On the receiver side, the reference clock is used to spin up the CDR circuit so that it can quickly lock to the bit rate of the incoming bitstream. The receiver’s PLL does not operate properly without a reference clock or if the reference clock frequency is not close enough to the frequency of the HD-SDI bitstream.

The reference clocks are required to be 1/20th the frequency of the bitstream ±100 ppm. Because HD-SDI has two different bit rates (1.485 Gb/s and 1.485 / 1.001 Gb/s), the RocketIO receiver must have both 74.25 MHz and 74.25 / 1.001 MHz reference clocks available if it is to support both HD-SDI bit rates.

As described in detail in the *RocketIO Transceiver User Guide* and in Chapter 9, each RocketIO transceiver has four reference clock inputs. A set of MUXes selects one of the four reference clock inputs as the active input. The method by which these MUXes are controlled limits dynamic switching to between two of the four inputs. Switching to the other set of two inputs requires reconfiguring the RocketIO transceiver. When implementing a RocketIO receiver, two reference clocks (either REFCLK and REFCLK2 or BREFCLK and BREFCLK2) typically are used, where one reference clock provides the 74.25 MHz reference frequency and the other provides 74.25 / 1.001 MHz.

Note that the selected reference clock is used by both the transmitter and the receiver in a RocketIO transceiver. It is not possible to select one reference clock for the transmitter and another reference clock for the receiver in a single RocketIO transceiver. However, different
RocketIO transceivers can have different reference clocks. This sharing of the reference clock by the transmitter and receiver has significant implications when trying to implement an HD-SDI transmitter and receiver in the same RocketIO transceiver. This topic is discussed in detail in “Appendix B: Implementing an HD-SDI Rx/Tx with One RocketIO Transceiver.”

The reference clocks have fairly stringent jitter requirements. At HD-SDI bit rates, the reference clock inputs should have no more than 100 ps of peak-peak jitter. Xilinx recommends the use of low-jitter oscillators with differential outputs to provide the reference clocks for the RocketIO transceivers. The transmitter section of the RocketIO transceiver actually imposes the most stringent requirements for low jitter on the reference clocks because jitter on the reference clock becomes jitter on the transmitter’s output. The RocketIO receiver is less sensitive to reference clock jitter. HD-SDI receivers have been successfully tested on the SDV board using singled-ended clock sources for the reference clocks. See “Appendix C: A Low-Cost Reference Clock Solution” for details.

User Clocks

The user clocks clock data out of the HD-SDI receiver and into the fabric of the FPGA. Each RocketIO transceiver requires two user clocks on the receiver side called RXUSRCLK and RXUSRCLK2. Each RocketIO transceiver also has two user clocks for the transmitter side called TXUSRCLK and TXUSRCLK2. If the transmitter portion of the transceiver is not used, the TXUSRCLK and TXUSRCLK inputs should still be driven with valid clock signals. In this case, simply connect TXUSRCLK to RXUSRCLK and TXUSRCLK2 to RXUSRCLK2.

RXUSRCLK is the clock signal that clocks data out of the RocketIO transceiver. The receiver output ports, such as RXDATA, change synchronously with the rising edge of RXUSRCLK. The frequency of RXUSRCLK is equal to the word rate of the HD-SDI interface, either 74.25 MHz or 74.25 / 1.001 MHz.

The frequency and phase relationships between RXUSRCLK and RXUSRCLK2 depend on the width of the RXDATA port of the RocketIO transceiver. For HD-SDI, a 20-bit RXDATA port is convenient to use because it matches the data word width of HD-SDI (10 bits of Y and 10 bits of C). When using a 20-bit RXDATA port, RXUSRCLK2 must have the same frequency and phase as RXUSRCLK (simply connect RXUSRCLK and RXUSRCLK2 to the same clock source). Consult the RocketIO Transceiver User Guide for RXUSRCLK2 requirements when other RXDATA port widths are used.

Note that the RocketIO transceiver’s RXDATA port is actually only 16 bits wide when a two-byte interface is selected. However, with the internal 8B/10B decoder bypassed, four additional output data bits are provided on other receiver output ports to form a 20-bit output data word. For simplicity, this chapter calls the entire 20-bit output port RXDATA.

In serial protocols that have clock correction capability, the RXUSRCLK and RXUSRCLK2 signals usually are derived from the same source as the reference clock. Then the RocketIO transceiver’s clock correction capability is used to occasionally insert or remove idle characters to compensate for the minor differences between the actual clock frequency of the incoming bitstream and the frequency of the local reference clock.

HD-SDI does not support clock correction. Therefore, deriving RXUSRCLK and RXUSRCLK2 from the reference clock quickly results in either an overflow or underflow condition on the output data port of the RocketIO receiver because RXUSRCLK and RXUSRCLK2 are running at a slightly different frequency than the frequency of the incoming bitstream.
When implementing an HD-SDI receiver, the recovered clock (RXRECCLK) from the RocketIO receiver is used as the source of RXUSRCLK and RXUSRCLK2. When connected in this manner, as shown in Figure 10-8, RXUSRCLK and RXUSRCLK2 always run at the same frequency as the incoming bitstream provided the RocketIO receiver’s CDR unit is locked to the bitstream. Thus underflow and overflow conditions are prevented on the RXDATA port of the RocketIO receiver.

**RXRECCLK**

The RXRECCLK output of the RocketIO transceiver is the recovered clock from the receiver’s CDR unit. When the CDR unit is locked to the incoming bitstream, this clock is exactly 1/20th the frequency of the bitstream. When the CDR unit is not locked to the bitstream frequency, RXRECCLK runs at the same frequency as the selected reference clock. Thus RXRECCLK always provides a word-rate clock for the HD-SDI receiver logic downstream from the RocketIO transceiver.

As shown in Figure 10-8, it is common to connect RXRECCLK to a BUFG global clock buffer. The output of the BUFG can be connected to the RXUSRCLK and RXUSRCLK2 inputs of the RocketIO transceiver and also to the clock inputs of the other portions of the HD-SDI receiver, such as the decoder, framer, and CRC checker.

### Reference Design

A high-level description of the reference design is in the following section. Detailed information about the reference design can be found in "Appendix A: Reference Design Details."

Most of the HD-SDI receiver reference design is contained in the module called hdsdi_rx. This module contains the HD-SDI decoder, framer, CRC checker, and the video format detector. It does not include the RocketIO transceiver module. The RocketIO module is kept separate from hdsdi_rx so that the RocketIO transceiver can be shared between an HD-SDI transmitter and receiver, if desired. Also not included in the hdsdi_rx module are the video timing decoder module (hdsdi_rx_timing) and a module (hdsdi_rx_autorate).
that automatically toggles between the reference clock inputs to the RocketIO transceiver until the HD-SDI receiver locks.

An example of how to connect the hdsdi_rx module to the RocketIO transceiver and the other modules is given in the sdv_hdsdi_rx module. This HD-SDI receiver example was designed specifically for the Xilinx SDV demo board. On the SDV board, there are no provisions for bringing the received parallel video out of the Virtex-II Pro FPGA. So, the received video is simply checked for CRC errors to determine correct reception. Figure 10-9 is a block diagram of the sdv_hdsdi_rx design.

Figure 10-9: Xilinx SDV Demo Board HD-SDI Receiver Reference Design

Figure 10-10 shows a block diagram of the main HD-SDI receiver module, hdsdi_rx. This module contains the four submodules described in the following paragraphs.
The 20-bit parallel data comes into the hdsdi_rx module from the RocketIO transceiver. The data is encoded and unframed at this point. The data words are sent into the hdsdi_decoder module. The decoder performs the two-step decoding process.

The output of the decoder is connected to the input of the framer module. There are two different framer modules provided in the reference design. They perform the same framing function but are implemented using different resources in the FPGA. The module called hdsdi_framer implements all framer functions using the fabric of the FPGA (LUTs and flip-flops). The hdsdi_framer_mult module provides an alternative implementation where six MULT18X18 multiplier blocks implement the barrel shifter function inside the framer, reducing the amount of FPGA fabric required for the framer by about one-half. This implementation can be a good trade-off if the multiplier blocks otherwise are unused.

The hdsdi_rx_crc module computes CRC values for both Y and C channels for each video line and compares them to the CRCs inserted into the video stream by the HD-SDI transmitter.

Finally, the hdsdi_autoformat_ln module examines the data and determines the video format. Once it recognizes the format, it asserts the std_locked output and outputs a 4-bit code indicating the format. This module is identical to the module of the same name described in detail in Chapter 9.

Figure 10-11 shows the block diagram of the hdsdi_rio module. This module is a wrapper around the RocketIO transceiver primitive (GT_CUSTOM). In addition to the RocketIO primitive, the module contains bit swap functions on the input and output data ports of the RocketIO primitive and a reset delay circuit for the RocketIO transceiver.
The RocketIO transceiver transmits the MSB of the TXDATA port first. Likewise, the RocketIO receiver outputs the first bit it receives on the MSB of the RXDATA port. HD-SDI always transmits the LSB first, just the opposite of how the RocketIO transceiver operates. In order to accommodate this difference, the hdsdi_rio module swaps the bit order on the input and output ports.

The RXRESET input of the GT_CUSTOM primitive must remain asserted for at least two RXUSRCLK cycles after all clock inputs become stable. The hdsdi_rio primitive contains some logic to keep the RXRESET input asserted until several clock cycles after the dcm_locked input becomes asserted. This input is called dcm_locked because it can be driven by the LOCKED output of a DCM, if a DCM is used to generate the RXUSRCLK and TXUSRCLK signals. If a DCM is not used to generate these clock signals, then the dcm_locked input either can be connected to another appropriate signal that indicates when the clocks are stable or can be tied High if the clocks are always running.

When an HD-SDI bitstream initially is connected to the RocketIO transceiver, the selected reference clock might not be the correct reference clock for the frequency of the bitstream. If the correct reference clock is selected, then the RocketIO transceiver quickly locks to the bitstream, and the HD-SDI receiver begins decoding and framing the data. However, if the wrong reference clock is selected, the HD-SDI receiver receives the video with many errors. The hdsdi_rx_autorate module examines the errors detected by the HD-SDI receiver and determines when it is appropriate to change the frequency of the reference clock. This module is described in more detail in “Appendix A: Reference Design Details.”

Figure 10-12 shows the results of input jitter tolerance measurements made on an HD-SDI receiver design implemented on the Xilinx SDV demo board. The input jitter tolerance of
the receiver was measured at different jitter frequencies and then plotted relative to the HD-SDI transmitter jitter template. As can be seen in the figure, the receiver’s input jitter tolerance is well above the HD-SDI jitter template for all jitter frequencies measured.

Reference Design Size

Table 10-1 shows the FPGA resources used by the HD-SDI receiver reference design. The complete reference design with all the modules is shown with the regular hdsdi_framer and hdsdi_framer_mult modules. The sizes of the optional modules (hdsdi_rx_timing, hdsdi_autodetect_ln, and hdsdi_rx_autorate) are also shown. If the optional modules are not included, their sizes can be subtracted from the implementation size of the full-featured design. The results shown in Table 10-1 were obtained with ISE 8.1 using XST for synthesis.

Table 10-1: Reference Design Implementation Sizes

<table>
<thead>
<tr>
<th>Design</th>
<th>LUTs</th>
<th>FFs</th>
<th>MULT18X18s</th>
</tr>
</thead>
<tbody>
<tr>
<td>HD-SDI receiver with all features using hdsdi_framer</td>
<td>608</td>
<td>385</td>
<td>0</td>
</tr>
<tr>
<td>HD-SDI receiver with all features using hdsdi_framer_mult</td>
<td>497</td>
<td>385</td>
<td>6</td>
</tr>
<tr>
<td>Size of optional hdsdi_rx_timing</td>
<td>17</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>Size of optional hdsdi_autodetect_ln</td>
<td>178</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>Size of optional hdsdi_rx_autorate</td>
<td>19</td>
<td>19</td>
<td>0</td>
</tr>
</tbody>
</table>
Conclusion

This chapter describes the implementation details of an HD-SDI receiver using the RocketIO multi-gigabit transceivers available in the Virtex-II Pro FPGA family. An HD-SDI receiver easily can be implemented from RocketIO transceivers combined with an HD-SDI decoder, framer, and other support functions built in the fabric of the FPGA.

The HD-SDI receiver reference design requires very few resources in the FPGA, making it quite easy to implement multiple HD-SDI interfaces in even the smallest member of the Virtex-II Pro family or to integrate video processing functions and an HD-SDI receiver all in the same part.

Design Files

The reference design files are available on the Xilinx website at:

www.xilinx.com/bvdocs/appnotes/xapp514.zip

Open the ZIP archive and extract file xapp514_hd-rx-mgt.zip.

Appendix A: Reference Design Details

This appendix contains detailed design information for the hdsdi_decoder, hdsdi_framer, and hdsdi_framer_mult modules.

hdsdi_decoder

The hdsdi_decoder module implements the two-stage decoding process to convert encoded HD-SDI data into decoded video data. The output data from the decoder is unaligned to word boundaries and must be framed by the hdsdi_framer module. Figure 10-13 shows a block diagram of the decoder module.

![hdsdi_decoder Block Diagram](x681_13_033005)

Figure 10-13: hdsdi_decoder Block Diagram

The decoder module first does the NRZI-to-NRZ conversion by XORing each bit with the previous bit in the bitstream. Remember that the LSB was the first bit received, so XORing d[1] with d[0] produces a new d[1] bit that has been converted to NRZ. The bit preceding d[0] is d[19] from the previous clock cycle. The prev_d19 register always captures the d[19]
bit every clock cycle so that it can be XORed with the \( d[0] \) bit of the next clock cycle to produce an NRZ \( d[0] \) bit.

Then the 20 NRZ bits are passed through the descrambler block. This block XORes each bit with two other bits to reverse the pseudorandom scrambling done by the HD-SDI encoder. In order to descramble all 20 bits, the descrambler needs the nine most significant bits produced by the NRZI-to-NRZ converter during the previous clock cycle. These bits are held in the prev_nrz register.

**hdsdi_framer**

The data from the RocketIO receiver is not aligned and usually contains bits from two different samples. The framer realigns the video samples so that each sample output from the framer contains the \( Y \) and \( C \) words from the same sample.

The framer searches for the bit pattern that marks the beginning of a timing reference, either EAV or SAV. When this unique pattern is located, the framer knows the offset of the sample’s least significant bit within the 20-bit data from the decoder. The framer uses this offset value to control a barrel shifter to realign all the subsequently received video samples.

Figure 10-14 shows a block diagram of the hdsdi_framer module. The framer has three main sections: the input pipeline registers, the timing reference signal (TRS) detector, and the barrel shifter.

Figure 10-14: **hdsdi_framer Block Diagram**

A TRS (either EAV or SAV) begins with the unique sequence of words: \( 3\text{FF}_{16}, 000_{16}, 000_{16} \). The \( Y \) and \( C \) channels each have a TRS that occurs at the same time in both channels. So the framer sees a 60-bit sequence like this: \( 3\text{FF}_{16}, 3\text{FF}_{16}, 000_{16}, 000_{16}, 000_{16}, 000_{16} \). The TRS detector matches the entire 60-bit HD-SDI TRS sequence.

The 20-bit input register, the two delay registers, and the LS 19 bits of the \( d \) input port form a 79-bit input vector that the TRS detector scans to find the 60-bit TRS. A "ones detector" looks for a run of 20 consecutive 1 bits; likewise, a "zeros detector" looks for 20-bit runs of 0 bits. The zeros detector is used twice, on consecutive clock cycles, to detect the complete...
40-bit run of zeros. The results of the ones detector and the two consecutive results from the zeros detector are combined to detect the complete 60-bit TRS sequence.

When a new TRS is detected, the TRS detector asserts the trs_detected signal and generates a binary code, called offset_val, indicating the bit position where the TRS was detected. This code is compared with the bit offset currently being used by the framer, stored in offset_reg. If the frame_en input to the framer is asserted, then a difference between the new offset_val and offset_reg causes offset_reg to load the new offset_val. If frame_en is not asserted, then offset_reg is not loaded, and the nsp output is asserted, indicating that a TRS was detected that did not match the current offset used by the framer.

The nsp output, combined with the frame_en input, can be used by control logic external to the framer module to implement simple or sophisticated TRS filtering. Sometimes, noise corrupts the HD-SDI bitstream and produces a bit sequence in the bitstream that looks like a TRS. If the framer always aligns to new TRS offsets, such an erroneous TRS could cause the framer to misalign data until the next TRS arrives. By controlling the frame_en input and monitoring when new TRS starting positions are detected based on the nsp output, control logic can prevent the framer from switching to a new TRS offset until some number of timing reference sequences arrive at the new offset.

The framer keeps the nsp output asserted until either the framer is allowed to reload offset_reg by the assertion of frame_en or when a TRS is detected that has a starting position that matches the current contents of offset_reg.

A simple TRS filtering scheme can be implemented by connecting nsp to frame_en. With this connection, the framer does not reload offset_reg when a TRS is detected that does not match offset_reg. Instead, nsp is asserted. With the assertion of nsp, frame_en is asserted, and the framer is allowed to reload offset_reg when the next TRS is detected. This next TRS either:

- matches the current offset_reg value if the TRS that caused the assertion of nsp was erroneous, thus filtering out the erroneous TRS, or
- forces the offset_reg to reload if the new TRS does not match the current offset_reg contents.

The offset value stored in offset_reg controls a barrel shifter. This barrel shifter realigns the video samples to their correct word alignment. The input vector for the barrel shifter is 39 bits wide. Depending on the value of offset_reg, the barrel shifter extracts a 20-bit output value from the 39-bit input vector.

In the hdsdi_framer module, the barrel shifter is made from three levels of MUXes. The first level consists of 2:1 MUXes that shift the input vector either 0 or 16 bit positions. The second level takes the output of the first level and shifts it 0, 4, 8, or 12 bit positions. Finally, the third level takes the output of the second level and applies the final 0, 1, 2, or 3 bit position shift.

The output of the barrel shifter is loaded into the barrel_out register, which drives the y and c output ports of the framer.

The framer module also contains some decoding logic that produces some TRS-related video timing signals. The trs output is asserted when all four samples of a TRS are output from the framer. The xyz output is asserted when the XYZ word of a TRS is output from the framer. The eav and sav outputs are asserted when the framer outputs the XYZ word of an EAV or SAV, respectively. Finally, the trs_err output is asserted when the XYZ word is output from the framer, if an error is detected in the XYZ word by examining the XYZ protection bits.
**hdsdi_framer_mult**

The hdsdi_framer_mult module is an alternate implementation of the framer. It is identical in function to hdsdi_framer with the only difference in how the barrel shifter is implemented. In the hdsdi_framer_mult module, six MULT18X18 multiplier blocks implement the barrel shifter rather than LUTs, as in hdsdi_framer. This can be a good trade-off if the multipliers are not required for other purposes since it essentially reduces the number of LUTs required to implement the framer in half.

A MULT18X18 block can be used as a barrel shifter by inputting the data to be shifted into one of the multiplier’s inputs and by putting a unary bit shift code into the other multiplier input. To shift the data zero positions, the shift code should be 1. To shift one position to the left, the shift code should be 2, and so on.

In the hdsdi_framer_mult module, the barrel shifter is implemented in two levels with three MULT18X18 blocks used in each level as shown in Figure 10-15. The top level of the barrel shifter shifts the input vector either 0 bit positions or 12 bit positions. The bottom level of the barrel shifter shifts the output of the top level from 0 to 11 bit positions. Thus, the barrel shifter can shift the input vector anywhere from 0 to 33 bit positions. However, the framer only requires shifts of 0 to 19 bit positions. So, the barrel shifter is not fully wired to support shifting by more than 19 bits.

![Figure 10-15: hdsdi_framer_mult Barrel Shifter](xsl81_15_120603)

In the top level of the barrel shifter, each multiplier acts like a 9-bit 2:1 MUX as shown in Figure 10-16. Note how every other output of the multiplier is used. The shift code, applied to the B input of the multiplier, only takes on the values of 1 or 2. When the shift code is 1, the X input bit of each MUX is selected and passes straight down to the output. When the shift code is 2, the Y input bit of each MUX is selected (by shifting it left one bit position).
With shift code = 1, the X inputs are transferred to the MUX outputs.

With shift code = 2, the Y inputs are transferred to the MUX outputs.

**Figure 10-16:** MULT18X18 used as Nine 2:1 MUXes
In the bottom level of the barrel shifter (see Figure 10-17), each multiplier acts like a 7-bit barrel shifter. Each multiplier has an 18-bit input vector connected to its A input and a shift code applied to its B input. If the shift code is 2048, the 7-bit output of the multiplier is equal to A[6:0]. If the shift code is 1024, the 7-bit output is equal to A[7:1], and so on until the shift code is 1 and the output is equal to A[17:11].

Figure 10-17: MULT18X18 used as Seven 12:1 MUXes
hdsdi_rx_autorate

This module selects the correct reference clock for the RocketIO transceiver. If the incorrect reference clock is selected, not matching the bitstream frequency, the RocketIO transceiver does not correctly recover the data.

Because the two HD-SDI frequencies are so close, much of the data is recovered correctly, even when the wrong reference clock is selected. The CDR unit in the RocketIO transceiver attempts to lock to the bitstream and recovers the data. However, after a certain period of time, the CDR unit determines that the frequency of the bitstream is not close enough to the frequency of the reference clock, and it switches back to using the reference clock to lock the CDR’s PLL. Once locked to the reference clock, the PLL then is freed to lock to the bitstream. This cycle repeats continuously since the reference clock is not close enough to the bitstream frequency.

When the PLL is close to the bitstream frequency, valid data is received. When the PLL is locked to the reference clock, the data is invalid.

The hdsdi_rx_autorate module examines the errors received by the HD-SDI receiver and determines when to toggle the REFCLKSEL input to the RocketIO transceiver. The CRC checkers in the HD-SDI receiver do a very good job of detecting errors in the received data. However, looking at CRC errors alone is not sufficient because the CRC checkers only work correctly if the TRS symbols are found by the receiver. When the wrong reference clock is selected, TRS symbols are detected roughly two-thirds of the time. Because these symbols are found so often, simply looking for some number of consecutive missing TRS symbols is not an adequate strategy.

Thus, the hdsdi_rx_autorate module uses a combination of missing TRS symbols and CRC errors to determine when to toggle the reference clock. A state machine looks for some number of video lines containing either CRC errors or a missing SAV. When this error threshold is reached, the state machine toggles the reference clock and again begins monitoring the HD-SDI output.

The error threshold is controlled by a counter called errcnt. The bit width of this counter is controlled by the parameter or constant ERRCNT_WIDTH. The maximum number of lines containing errors before the switching threshold is reached is controlled by MAX_ERRS. By default, the error counter is two bits wide, and the switching threshold is reached when three consecutive video lines containing errors have been detected. By changing the error counter width and maximum error threshold, you can trade off error tolerance for switching latency. Requiring more consecutive lines with errors before the switching threshold is reached decreases the likelihood that the reference clock is switched erroneously. However, this also increases the latency for the reference clock switch when the frequency of the bitstream does change.

Figure 10-18 shows the state diagram of the state machine in hdsdi_rx_autorate.
Appendix B: Implementing an HD-SDI Rx/Tx with One RocketIO Transceiver

In the RocketIO transceiver, the receiver and transmitter portions of the transceiver share the same reference clock. It is not possible to select one reference clock for the transmitter and a different reference clock for the receiver inside of the same RocketIO transceiver. This sharing of the reference clock can lead to complications when trying to implement an HD-SDI transmitter and receiver in the same RocketIO transceiver. Depending on the application, it might not be possible to place the transmitter and receiver in one RocketIO transceiver.

If it is required that the transmitter is running at one HD-SDI bit rate while the receiver is receiving a bitstream at the other HD-SDI bit rate, then the receiver and transmitter must be placed in separate RocketIO transceivers.

One case where it makes sense to put the HD-SDI transmitter and receiver in the same RocketIO transceiver is a pass-through HD-SDI interface. In a pass-through interface, the video received by the HD-SDI receiver is connected to the HD-SDI transmitter to be retransmitted. In such a configuration, the transmitter always runs at the same rate as the receiver. Some processing might be done on the video between the receiver and the transmitter.

**Figure 10-18: hdsdi_rx_autorate State Diagram**
Figure 10-19 shows an example where the data is received by the RocketIO transceiver. The video is decoded and framed by the HD-SDI receiver, then a logo “bug” is inserted into the video. New CRC values for the video lines are calculated, the video is encoded by the HD-SDI transmitter, and the encoded video is transmitted by the RocketIO transceiver.

As shown in Figure 10-19, the RXRECCCLK signal from the RocketIO receiver clocks all logic downstream from the receiver because RXRECCCLK is exactly the frequency of the video stream. RXRECCCLK clocks the HD-SDI transmitter and also clocks the encoded data into the RocketIO transmitter because it is connected to TXUSRCLK and TXUSRCLK2. The FIFO at the input of the RocketIO transmitter moves the data from the TXUSRCLK domain to the REFCLK domain. Data is clocked from the TXDATA port into the FIFO using TXUSRCLK (which is connected to RXRECCLK). Data is read from the FIFO using the selected reference clock. Data is read from the FIFO using the reference clock because the serializer is running synchronous to the reference clock, but at 20 times the frequency of the reference clock.

Herein lies the problem with sharing the reference clock between the receiver and the transmitter. If TXUSRCLK is not frequency locked to the reference clock, then the FIFO at the input of the RocketIO transmitter underflows or overflows. However, when using one RocketIO for both the transmitter and receiver in a pass-through HD-SDI interface, TXUSRCLK must run at the video rate, that is, it must be connected to RXRECCCLK. Because RXRECCCLK and the reference clock are rarely the same frequency, the RocketIO transmitter FIFO underflows or overflows in this configuration. There are two possible solutions to this problem.

The first solution is to resynchronize the video to the reference clock after it is received. Video resynchronization, sometimes done on a per line basis, usually is done on a frame basis. Resynchronizing video to the local reference clock requires a line buffer for line synchronization or a frame buffer for frame synchronization. Discussion of these video synchronization techniques is beyond the scope of this document.
A second solution is to somehow force REFCLK to track the frequency of RXRECLK. RXRECLK cannot be directly used as the reference clock because it has too much jitter.

Figure 10-20 shows a single RocketIO transceiver pass-through configuration that has been successfully tested on the Xilinx SDV demo board. The reference clocks are provided by VCXOs. The frequency of each VCXO is controlled by a phase detector, where the combination of the VCXO, the loop filter, and the phase detector forms a PLL. The phase detector compares the frequency of the VCXO and RXRECLK and adjusts the VCXO so that its frequency matches that of RXRECLK.

This scheme solves the initialization problem of trying to spin up the CDR unit in the RocketIO receiver prior to having a stable RXRECLK. Recall that during spin up, RXRECLK is equal to the reference clock. If RXRECLK is supplying the reference clock, then there is a cyclic problem with initializing the system. Because the pull range of a VCXO (the range over which the frequency of the VCXO can be adjusted by the control input) is limited to within something like ±100 ppm, the VCXO always is close in frequency to its normal center frequency, allowing the CDR unit to spin up and lock to the incoming bitstream frequency. During the initial stages of the CDR spin-up process, RXRECLK is set equal to the reference clock input (the VCXO’s output). The phase detector, seeing that RXRECLK is the same frequency as the VCXO, does not try to adjust the frequency of the VCXO. As the CDR unit begins to lock to the bitstream, RXRECLK starts to move towards the bitstream frequency, and the phase detector forces the VCXO to track this frequency change, keeping the reference clock locked to the frequency of RXRECLK.

As shown in Figure 10-20, this technique usually requires two VCXOs, one for each of the two HD-SDI reference clock frequencies because the pull range of most VCXOs is not sufficient to allow one VCXO to operate at both HD-SDI frequencies.

Figure 10-20: Using a VCXO to Lock the Reference Clock to RXRECLK
Using a VCO instead of a VCXO is a tempting consideration because VCOs typically have a much larger pull range than a VCXO, possibly allowing one VCO to run at both HD-SDI bit rates. However, VCOs typically have more jitter than VCXOs, so a low-jitter VCO is required. Also, some sort of mechanism is required to force the VCO to run at the two HD-SDI frequencies of 74.25 MHz and 74.1758 MHz under control of a supervisor circuit to provide the necessary reference for CDR spin up. After spin up, the VCO is controlled by the phase detector and tracks the RXRECCLK frequency.

Appendix C: A Low-Cost Reference Clock Solution

As documented earlier in this chapter, the RocketIO transceiver requires two reference clock frequencies of 74.25 MHz and 74.25 / 1.001 MHz in order to support the two HD-SDI bit rates. These two reference clock frequencies can be provided using either two crystal oscillators (one for each frequency) or a frequency synthesizer.

One possible low-cost frequency synthesizer is the ICS664 Digital Video Clock Source (http://www.icst.com). This device can synthesize a number of different video related frequencies from one input reference clock. The ICS664 can produce both 74.25 MHz and 74.25 / 1.001 MHz from either a 13.5 MHz or 27 MHz reference. It also can produce a 74.25 / 1.001 MHz from a 74.25 MHz reference clock and 74.25 MHz from a 74.25 / 1.001 MHz reference clock. The ICS664 comes in several versions. The ICS664-01 has a single-ended LVCMOS output. The ICS664-02 has a differential output and produces less jitter.

In Figure 10-21, an ICS664 uses a 27 MHz crystal from which it can synthesize either 74.25 MHz or 74.25 / 1.001 MHz on its output. The output of the ICS664 is connected to an IOB of the Virtex-II Pro FPGA. Internally, this signal is connected directly to the reference clock input of the RocketIO transceiver. A single output from the Virtex-II Pro FPGA commands the ICS664 to generate either 74.25 MHz or 74.25 / 1.001 MHz. This output can come from a module like hdsdi_rx_autorate, which toggles this signal periodically until the RocketIO transceiver locks to the incoming bitstream.

![Figure 10-21: ICS664 Providing REFCLK to RocketIO Transceiver](image-url)
In order to get the best performance from the ICS664, follow all the guidelines given in the ICS664 data sheet for power supply filtering and layout. Note that the REF clock output is disabled by grounding VDDR, which reduces the output jitter on the CLK output. Also note that the VDDO supply pin is connected to 2.5V to make the CLK output compatible with the 2.5V I/O standards of the Virtex-II Pro FPGA. The SELIN pin selects between using a crystal as the reference clock to the ICS664 (when High) or using an external clock source (when Low). SELIN has an internal pull-up resistor. The S[3:0] inputs also have internal pull up resistors. To generate 74.25 MHz and 74.25 / 1.001 MHz, the settings of S[3:0] are as follows:

- S3 and S2 are Low
- S1 is High
- S0 is Low to generate 74.25 MHz and High to generate 74.25 / 1.001 MHz

The S0 pin is driven with either the LVCMOS33 or the LVCMOS25 standard from the Virtex-II Pro FPGA.
Chapter 11

HD-SDI Integration Examples for the Serial Digital Video Demonstration Board

Summary

The high-definition serial digital interface (HD-SDI) standard describes how to transport high-definition (HD) digital video serially over video coax cables. HD-SDI is used to connect HD video equipment in broadcast studios and video production centers. It is an evolution of the popular SD-SDI standard that is widely used to transport standard-definition (SD) digital video in the broadcast industry.

Chapter 9 and Chapter 10 describe implementing the HD-SDI transmitter and receiver functions in Virtex™-II Pro FPGAs. This chapter presents three application examples that show how to use the HD-SDI transmitter and receiver blocks to form complete HD-SDI interfaces. These three demonstration applications are part of the standard demonstration suite for the Xilinx Serial Digital Video (SDV) demonstration board [Ref 1].

Introduction

HD-SDI, defined by the SMPTE 292M standard, is the standard transport protocol for uncompressed HD digital video in the broadcast studio and video production center [Ref 2].

This chapter describes three application examples built for the Xilinx SDV demonstration board:

1. Separate HD-SDI Tx and Rx
   This example contains one HD-SDI transmitter and one HD-SDI receiver. The video for the transmitter comes from an HD video test pattern generator (Chapter 17). The receiver section decodes the incoming video stream and checks it for CRC errors. The transmitter and receiver are separate from each other and can run at separate video rates (for example, one at 74.25 MHz and the other at 74.1758 MHz).

2. HD-SDI Pass-Through
   This example has one HD-SDI receiver and two transmitters. The video from the receiver is sent to one of the transmitters where it is re-encoded and retransmitted. The video source for the second transmitter comes from a video test pattern generator.

3. HD-SDI Pass-Through using ICS664
This example is identical to the pass-through configuration above, but uses the ICS664 clock synthesizer to generate the HD-SDI reference clocks for the pass-through receiver and transmitter [Ref 3].

All three applications are part of the Xilinx SD board demonstration suite. They are written specifically for the SD board demonstration board, but are easily adapted to customer applications.

Refer to Chapter 9, Chapter 10, and Chapter 17 for detailed descriptions of the various modules used in this application. Chapter 9 contains a detailed description of the clocking requirements of the RocketIO™ transceiver pertaining to using the transceiver in an HD-SDI interface. Both Chapter 9 and Chapter 10 provide more details about the HD-SDI protocol.

**Separate HD-SDI Tx and Rx Application**

Figure 11-1 is a block diagram of the separate HD-SDI Tx and Rx application.

---

**Figure 11-1:** Separate HD-SDI Tx and Rx Block Diagram
Clocks

Because the transmitter and receiver both support HD-SDI bit rates of 1.485 Gb/s and 1.485 / 1.001 Gb/s (~1.4835 Gb/s), the RocketIO transceivers for the transmitter and receiver must have two separate reference clocks of 74.25 MHz (1.485 GHz / 20) and 74.1758 MHz (1.4835 GHz / 20). On the SDV demonstration board, these two reference clocks are provided by low-jitter crystal oscillators. The oscillators have differential outputs connected to the specific IBUFGDS input buffers with dedicated routing to the BREFCLK and BREFCLK2 inputs of the RocketIO transceivers on the top edge of the XC2VP4 device used on the SDV board.

The clocks used in this application are summarized below:

- **clk_74_17M**
  This clock comes from the 74.1758 MHz XO on the SDV demonstration board. It provides the reference frequency needed to operate the RocketIO transceivers at the 1.4835 Gb/s HD-SDI bit rate. The signal from the oscillator is buffered by an IBUFGDS input buffer and is connected to one of the BREFCLK input ports of both RocketIO transceivers and to the BUFGMUX that drives the tx_usrclk signal.

- **clk_74_25M**
  This clock comes from the 74.25 MHz XO on the SDV demonstration board. It provides the reference frequency needed to operate the RocketIO transceivers at the 1.485 Gb/s HD-SDI bit rate. The signal from the oscillator is buffered by an IBUFGDS input buffer and is connected to one of the BREFCLK input ports of both RocketIO transceivers and to the BUFGMUX that drives the tx_usrclk signal.

- **tx_usrclk**
  This global clock signal is driven by a BUFGMUX. The inputs to the BUFGMUX are clk_74_25M and clk_74_17M. A DIP switch controls the select port of the BUFGMUX to choose the clock rate for the transmitter section. This signal clocks the video pattern generator and the HD-SDI transmitter logic. It also drives the TXUSRCLK inputs of the RocketIO transceiver so that the data from the HD-SDI encoder can be loaded synchronously into the RocketIO transmitter section.

- **rx_usrclk**
  The receiver section’s RocketIO transceiver produces a recovered clock on its RXRECLK output port. This clock runs at the word rate of the recovered data (1/20th the bit rate). The RXRECLK output is buffered by a BUFG, and the resulting global clock is used to clock all of the HD-SDI receiver logic in the FPGA. This signal also drives the RXUSRCLK inputs of the RocketIO transceiver so that the data emerging on the RXDATA port of the RocketIO transceiver is synchronous with this global clock.

- **gclk_33M**
  This global clock is used only to control the flash rate of the LEDs on the SDV board. The clock comes from a 33 MHz oscillator on the SDV demonstration board.

Transmitter Section

The bit rate of the transmitter section is selected by a DIP switch on the SDV board. This signal is connected to the REFCLKSEL input of the transmitter RocketIO transceiver to select between the two reference clocks. The same DIP switch also controls the select input of a BUFGMUX, selecting between the two reference clock sources to provide a global transmitter clock running at the HD video word rate.
The multigenHD video pattern generator module from Chapter 17 (slightly modified to optionally insert a Xilinx logo in a portion of the color bar test pattern) generates HD video for the transmitter section. DIP switches on the SDV board control the video pattern generator, selecting the video test pattern and the video format. The video pattern generator can produce all 13 video formats identified in the SMPTE 292M document as being compatible with HD-SDI plus the five HD-SDI compatible segmented-frame video formats defined in SMPTE RP 211. It can generate three different video test patterns: SMPTE RP 219-2002 color bars, 75% color bars, and the SMPTE RP 198-1998 HD digital checkfield pattern.

The video from the video pattern generator is encoded by the HD-SDI encoder. The encoded data stream is converted to a serial bitstream by the RocketIO transceiver. The output of the RocketIO transceiver is buffered by an external SDI cable driver on the SDV board and connects to a 75 Ω BNC connector.

### Receiver Section

The input HD-SDI signal comes from a BNC connector and passes through an SDI cable equalizer before entering the RocketIO transceiver. The RocketIO transceiver locks to the incoming bitstream and provides a recovered clock on the RXRECCCLK port, running at 1/20th the bit rate, and the recovered data, 20 bits per clock cycle, on the RXDATA port.

As described in Chapter 10, the RocketIO transceiver requires reference clocks close in frequency to the bitstream frequency. The transceiver in the HD-SDI receiver section is, therefore, provided with the same two reference clocks used by the transmitter’s transceiver. An automatic rate selection module controls the REFCLKSEL input of the transceiver, switching between the two reference clocks when appropriate.

The recovered clock from the RocketIO transceiver is buffered by a BUFG and drives all the logic in the receiver section at the HD video word rate of either 74.25 MHz or 74.1758 MHz. Notice how this global clock is connected to the RXUSRCLK and RXUSRCLK2 inputs of the RocketIO transceivers. This connection causes the data on the transceiver’s RXDATA output port to be synchronous with the global receiver clock.

The data from the RXDATA port of the transceiver is first descrambled and then framed to recover the original video stream. The video stream is fed into a CRC checker to detect transmission errors. Detection of any CRC error causes an LED on the SDV board to flash until cleared by pushing a button. The error detection signal is also present on a test point connector on the SDV board so that the actual error rate can be determined. The received video stream also feeds into a video format detection module. This module determines which of the HD-SDI compatible formats is being received. Several LEDs on the SDV board are driven by the format detector to indicate the received video format.

### Design Size

Table 11-1 shows the FPGA resources used by this application. These results were obtained with ISE 6.3i using XST with the Verilog version of the design. Area optimization was used for XST. The design meets all timing constraints in a -5 speed grade XC2VP4 device.

<table>
<thead>
<tr>
<th></th>
<th>FF</th>
<th>LUT</th>
<th>Block RAM</th>
<th>MULT18X18</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>525</td>
<td>881</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>
HD-SDI Pass-Through Application

Figure 11-2 is a block diagram of the HD-SDI pass-through application.

This application has two HD-SDI transmitters. One of them, shown at the top of Figure 11-2, is a standalone transmitter driven by the video pattern generator. This transmitter is identical to the transmitter from the HD-SDI Separate Tx and Rx application.

The second transmitter is coupled with the HD-SDI receiver to form a pass-through interface. The data from the receiver is descrambled and framed and then checked for errors before being re-encoded and retransmitted. A single RocketIO transceiver is used for both transmit and receive functions of the pass-through interface.
Clocks

The following paragraphs describe the various clocks used in this application:

- **clk_74_17M**
  This clock comes from the 74.1758 MHz XO on the SDV demonstration board. It provides the reference frequency needed to operate the RocketIO transceiver in the standalone transmitter section at the 1.4835 Gb/s HD-SDI bit rate. The signal from the oscillator is buffered by an IBUFGDS input buffer and is connected to one of the BREFCLK input ports of the standalone transmitter’s RocketIO transceiver and to the BUFGMUX that drives the tx2_usrclk signal.

- **clk_74_25M**
  This clock comes from the 74.25 MHz XO on the SDV demonstration board. It provides the reference frequency needed to operate the RocketIO transceiver in the standalone transmitter section at the 1.485 Gb/s HD-SDI bit rate. The signal from the oscillator is buffered by an IBUFGDS input buffer and is connected to one of the BREFCLK input ports of the standalone transmitter’s RocketIO transceiver and to the BUFGMUX that drives the tx2_usrclk signal.

- **tx2_usrclk**
  This global clock signal is driven by a BUFGMUX. The inputs to the BUFGMUX are clk_74_25M and clk_74_17M. A DIP switch controls the select port of the BUFGMUX to choose the clock rate for the standalone transmitter section. This signal clocks the video pattern generator and the HD-SDI transmitter logic in the standalone transmitter.

- **rx_usrclk**
  The receiver in the RocketIO transceiver produces a recovered clock on its RXRECCCLK output port. This clock runs at the word rate of the recovered data (1/20th the bit rate). This recovered clock is buffered by a BUFG, and the resulting global clock clocks all of the HD-SDI receiver and transmitter logic in the pass-through interface.

- **clk_hd_vcxo**
  This clock comes from a 74.1758 MHz VCXO on the SDV board. The VCXO is part of a PLL (see “PLL” section) that locks to the recovered clock from the receiver. clk_hd_vcxo is connected to the REFCLK input of the RocketIO transceiver in the pass-through HD-SDI interface.

- **gclk_33M**
  This global clock is used only to control the flash rate of the LEDs on the SDV board. The clock comes from a 33 MHz oscillator on the SDV demonstration board.

PLL

One interesting portion of this design is the phase-locked loop (PLL) that provides the reference clock for the RocketIO transceiver. This PLL serves two purposes in this application. First, it reduces the jitter on the recovered clock from the RocketIO transceiver so that a low-jitter reference clock can be provided to the transceiver’s transmitter section. Second, the way in which the PLL is placed between the recovered clock output of the transceiver and the reference clock input of the transceiver allows use of both the transmitter and receiver sections of the same RocketIO transceiver in this application. A complete description of this configuration can be found in “Appendix B: Implementing an HD-SDI Rx/Tx with One RocketIO Transceiver” in Chapter 10.
The PLL is made from a VCXO and loop filter external to the FPGA plus a phase detector built in programmable logic of the FPGA. The phase detector controls the VCXO so that its frequency and phase match the recovered clock from the receiver. When the receiver portion of the transceiver is locked to the incoming bitstream, the recovered clock on the RXRECCLK output port is locked to the frequency of the bitstream (divided by 20). The VCXO-based PLL locks to the recovered clock. The transmitter portion of the transceiver uses the clock signal from the VCXO to generate the serial clock for the output bitstream. Thus, the output bitstream frequency is locked to the frequency of the input bitstream.

Note that the SDV board has a 74.1758 MHz VCXO, but does not have a 74.25 MHz VCXO. Thus, in this application, the pass-through interface only works at the 1.4835 Gb/s bit rate. See the next application example using the ICS664 clock synthesizer for an example of how the SDV board can implement a pass-through interface supporting both HD-SDI bit rates. Figure 11-3 shows a portion of the SDV board schematic with the loop filter and VCXO used in this application.

![Figure 11-3: VCXO and Loop Filter](image)

The VCXO used on the SDV demonstration board has a 3.3V LVPECL output. AC coupling is used to interface this signal to a 2.5V LVDS input on the FPGA.

**Design Size**

Table 11-2 shows the FPGA resources used by this application. These results were obtained with ISE 6.3i using XST with the Verilog version of the design. Area optimization was used for XST. The design meets all timing constraints in a -5 speed grade XC2VP4 device.

**Table 11-2: FPGA Resources Used by the HD-SDI Pass-Through Application**

<table>
<thead>
<tr>
<th>FF</th>
<th>LUT</th>
<th>Block RAM</th>
<th>MULT18X18</th>
</tr>
</thead>
<tbody>
<tr>
<td>634</td>
<td>1075</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>

**HD-SDI Pass-Through Using the ICS664 Application**

Figure 11-4 is a block diagram of the HD-SDI pass-through application using the ICS664 frequency synthesizer device.
This application is almost identical to the previous pass-through application. However, the 74.1758 MHz VCXO is replaced with a combination of a 27 MHz VCXO and an ICS664-01 frequency synthesizer. The ICS664 takes in a 27 MHz reference clock and can generate the two HD-SDI reference clock frequencies (74.25 MHz and 74.1758 MHz). When driven from a 27 MHz VCXO as shown here, the output of the ICS664 can be varied by adjusting the frequency of the VCXO. By adding a loop filter on the input of the VCXO and a phase detector to control VCXO, a PLL can be created supporting both HD-SDI frequencies. Thus, in this application, the pass-through HD-SDI interface supports both HD-SDI bit rates.

The SDV demonstration board originally shipped with an older part, the ICS660. The ICS664 has significantly better jitter specifications than the older ICS660. Xilinx does not
recommend using the ICS660 in this configuration. The ICS664 comes in several varieties. To verify this design, the ICS660 on the SDV board was replaced with the pin and function compatible ICS664-01. For new designs, the ICS664-02 is recommended. The ICS664-02 has a differential LVPECL output that produces lower jitter than the singled-ended output of the ICS664-01.

Figure 11-5 is a portion of the SDV board schematics showing the loop filter, 27 MHz VCXO, and ICS664-01.

Figure 11-5: Loop Filter, 27 MHz VCXO, and ICS664-01

The 27 MHz VCXO used on the SDV demonstration board is the Pericom PI6CX100-27W. Several other companies make similar low-cost 27 MHz VCXOs. The flexibility of the ICS664 allows other VCXO frequencies to be used, including 13.5 MHz, 54 MHz, 74.25 MHz, and 74.1758 MHz.

An automatic rate detection module was added to this design, providing support for both HD-SDI bit rates. This allows the HD-SDI receiver to automatically lock to the incoming HD-SDI bitstream, even if it changes frequencies. The rate detection module, described in detail in Chapter 10, determines when the RocketIO transceiver is not locked to the incoming bitstream and begins toggling between the reference clock frequencies until the RocketIO transceiver is again locked to the bitstream.

Design Size

Table 11-3 shows the FPGA resources used by this application. These results were obtained with ISE 6.3i using XST with the Verilog version of the design. Area optimization was used for XST. The design meets all timing constraints in a -5 speed grade XC2VP4 device.

Table 11-3: FPGA Resources Used by the HD-SDI Pass-Through with the ICS664-01 Application

<table>
<thead>
<tr>
<th>FF</th>
<th>LUT</th>
<th>Block RAM</th>
<th>MULT18X18</th>
</tr>
</thead>
<tbody>
<tr>
<td>655</td>
<td>1068</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>
Conclusion

This chapter presents three different HD-SDI integration examples using the basic HD-SDI transmitter and receiver building blocks from Chapter 9 and Chapter 10. It shows how to combine these blocks to build complete HD-SDI interfaces in various configurations.

The flexibility of the Virtex-II Pro FPGA family, including the multi-gigabit RocketIO transceivers, allows customers to easily create customized HD-SDI interfaces. These interfaces use a small portion of the resources in the FPGA, allowing other video processing functions to be implemented in the same FPGA.

Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_hd-integ-demobrd.zip.
Section III: Multi-Rate HD/SD-SDI

Audio/Video Connectivity Solutions for the Broadcast Industry
Chapter 12

Multi-Rate HD/SD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

Summary

The SD-SDI standard is widely used in broadcast studios and video production centers to transport standard definition (SD) digital video serially over video coax cable. The HD-SDI standard is similar, but transports high-definition (HD) digital video. The SD-SDI and HD-SDI standards are similar enough that it is possible to implement interfaces for video equipment that support both standards through the same connector.

This chapter describes how to use the RocketIO™ multi-gigabit transceivers available in the Virtex™-II Pro family of FPGA devices to implement a transmitter that can support both SD-SDI and HD-SDI. The flexibility of the RocketIO transceivers, combined with the programmable logic of the Virtex-II Pro devices, makes it possible to implement multi-rate SDI interfaces.

Since all Virtex-II Pro devices have four or more RocketIO transceivers, it is possible to implement multiple HD-SDI and SD-SDI interfaces in a single FPGA.

Introduction

The SD-SDI standard, traditionally known simply as SDI, is defined by the SMPTE 259M standard [Ref 1] and also by the equivalent ITU-R BT.656 standard [Ref 2]. HD-SDI is defined by the SMPTE 292M standard. The SD-SDI standard is widely used in broadcast studios and video production centers today. Use of HD-SDI is increasing rapidly as the broadcast industry ramps up support for HDTV broadcasting.

Throughout this document, the term SD-SDI is used to refer to the SD version of the interface standard. SDI is used to refer generically to both SD-SDI and HD-SDI. And, multi-rate SDI is used to refer to support of both standards.

This chapter focuses specifically on the implementation of a multi-rate SDI transmitter using the RocketIO transceivers available in the Virtex-II Pro FPGA family. Chapter 13 describes the implementation of multi-rate SDI receivers, also using RocketIO transceivers.

This chapter refers to the Xilinx SD-SDI section of this volume (Chapter 2 through Chapter 8) and the Xilinx HD-SDI transmitter chapter, Chapter 9. Please refer to these chapters for more information.

The RocketIO transceivers are designed to support serial bit rates from 622 Mb/s to 3.125 Gb/s. HD-SDI, with bit rates of approximately 1.5 Gb/s, falls within this range. However, all of the current SD-SDI bit rates, ranging from 143 Mb/s to 540 Mb/s, are well...
Chapter 12: Multi-Rate HD/SD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

below the range supported by the RocketIO transceivers. Therefore, the main focus of this chapter is on a technique that allows the RocketIO transmitter to support these slower bit rates without violating any of the specifications of the RocketIO transceivers. Chapter 9 describes how to use the RocketIO transceiver to build an HD-SDI transmitter. The combination of the SD-SDI transmission technique described here and the HD-SDI transmitter from Chapter 9 results in a multi-rate SDI transmitter design. SD-SDI-only transmitters can also be built using the RocketIO transceivers, or they can be implemented in the fabric of the FPGA as described in the SD-SDI chapters.

Two reference designs are included with this chapter. One is a multi-rate SDI transmitter and the other is an SD-SDI only transmitter. These reference designs have been tested and verified on the Xilinx SDV demo board [Ref 3].

SD-SDI and HD-SDI Similarities and Differences

The basic electrical specifications of HD-SDI and SD-SDI are virtually identical, making it possible to design multi-rate SDI interfaces that support both standards. Both standards have the same basic electrical interface: a singled-ended signal with an 800 mV peak-to-peak swing centered around 0.0V. Both use 75Ω coaxial cable and BNC connectors. And, both use the same encoding algorithm.

The most obvious difference between HD-SDI and SD-SDI is the much higher bit rate required to support HD video. HD-SDI has two bit rates: 1.485 Gb/s and 1.485/1.001 Gb/s. SD-SDI bit rates range from 143 Mb/s to 540 Mb/s.

To support the higher bit rates, HD-SDI transmitters must have faster rise and fall times on their outputs. In fact, the slowest rise and fall times permitted by the HD-SDI standard are too fast to be legal for SD-SDI.

Another difference is that HD video is 20 bits wide with two 10-bit channels, one for chroma and one for luma. An HD-SDI transmitter encodes and transmits 20 bits for every video clock cycle, whereas SD-SDI only encodes and transmits 10 bits per video clock cycle.

Due to the higher bit rate of HD-SDI, the maximum coax cable length supported by HD-SDI is 100 meters versus the 300-meter maximum supported by SD-SDI. However, HD-SDI also permits the use of an optical interface to allow longer transmission distances.

Generating SD-SDI Bitstreams with the RocketIO Transmitter

The original SD-SDI standard supports four bit rates:

- 143 Mb/s for NTSC digital composite video
- 177.3 Mb/s for PAL digital composite video
- 270 Mb/s for NTSC and PAL digital component video
- 360 Mb/s for NTSC and PAL 16:9 aspect ratio digital component video

In addition, a more recent document (SMPTE 344M) adds a 540 Mb/s bit rate compatible with SD-SDI.

The digital composite video rates of 143 Mb/s and 177.3 Mb/s are rarely used. The most commonly used bit rate, by far, is 270 Mb/s. It is quite common for a piece of video equipment to support only the 270 Mb/s bit rate through its SDI interface. It is also common for equipment to support both 270 Mb/s and 360 Mb/s. The 540 Mb/s bit rate is relatively new and not widely supported, yet.
All of these bit rates are well below the 622 Mb/s minimum bit rate supported by RocketIO transceivers in Virtex-II Pro devices. Therefore, it is necessary to work around this minimum bit rate limitation in order to support SD-SDI with the RocketIO transceivers.

To transmit SD-SDI, the RocketIO transceiver is configured to run at some integer multiple of the SD-SDI bit rate and each bit is sent multiple times consecutively. For example, given a reference clock of 54 MHz, a RocketIO transceiver multiplies this reference clock by 20 resulting in a 1.08 Gb/s bitstream—exactly four times 270 Mb/s. If each encoded bit is transmitted by the RocketIO transmitter four times consecutively, the RocketIO transmitter produces a bitstream that is, in every way, equivalent to a normal 270 Mb/s SD-SDI bitstream. Figure 12-1 shows this in more detail.

Figure 12-1: RocketIO Transmitter Producing a 270 Mb/s SD-SDI Bitstream

Any bitstream frequency that is any integer multiple of the SD-SDI bit rate can be used, so long as it is fast enough to meet the minimum frequency requirements for the RocketIO transceiver. The “Clocks” section of this chapter discusses clock requirements in detail.

Cable Driver

Both SD-SDI and HD-SDI use singled-ended (unbalanced) signaling over 75Ω coaxial cable. As mentioned in Chapter 9, Xilinx does not recommend driving this single-ended interface directly using the RocketIO transmitter because the transceiver’s CML outputs are designed to drive differentially. Instead, an SDI cable driver with differential inputs should be used.

When implementing a multi-rate SDI transmitter, use of a multi-rate SDI cable driver is essential. SD-SDI and HD-SDI have the same basic electrical specifications for the transmitter, but they do differ on one important specification—rise and fall time. SMPTE 259M requires that the rise and fall times of the SD-SDI signal must be no less than 400 ps and no more than 1.50 ns. On the other hand, SMPTE 292M requires that the HD-SDI

1. SMPTE 344M defines a new SD-SDI bit rate of 540 Mb/s. To support this faster bit rate, SMPTE 344M requires compatible SDI transmitter to have rise and fall times of no less than 400 ps and no more than 800 ps. This is a subset of the 400 ps to 1.50 ns range allowed by SMPTE 259M.
Chapter 12: Multi-Rate HD/SD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers

signal rise and fall time must be no more than 270 ps. In order to meet these differing requirements, a cable driver with adjustable slew rates must be used.

Figure 12-2 shows how to interface a GS1528 multi-rate SDI cable driver to the RocketIO transmitter. The GS1528 cable driver has 3.3V LVPECL differential inputs. The transmitter output of the RocketIO transceiver is a differential 2.5V CML pair not directly compatible with the cable driver’s LVPECL inputs. AC coupling is used between the RocketIO transceiver’s output and the input of the GS1528 to shift the signal levels from the 2.5V CML levels to the LVPECL levels. Large AC coupling capacitor values, in the range of 1 µF to 4.7 µF, must be used in order to successfully pass the pathological waveforms that can be generated by the HD-SDI and SD-SDI encoders. The rate control input pin of the GS1528 sets the slew rate of the driver. This input must be High for SD-SDI and Low for HD-SDI.

Figure 12-2: Interfacing the GS1528 Cable Driver to the RocketIO Transmitter

Clocks

One of the most important aspects of implementing an SDI transmitter using RocketIO transceivers is providing the right clocks to the transceivers. RocketIO transceivers requires two types of clocks, reference clocks and user clocks. The reference clocks are used to generate the bit-rate clock for the serializer. The user clocks are used to clock data from the fabric of the FPGA into the RocketIO transceiver. More details about the clocking requirements of the RocketIO transceivers can be found in the RocketIO Transceiver User Guide [Ref 4] and in Chapter 9.

Reference Clocks

The reference clocks provide low-jitter frequency references for the RocketIO transceiver. The RocketIO transceiver multiplies the selected reference clock by 20 to obtain a bit-rate clock for the transmitter's serializer. Jitter present on the reference clock shows up as jitter on the transmitter output, so it is important that a low jitter reference clock be used.

When implementing an HD-SDI transmitter, the reference clock must be either 74.25 MHz or 74.25/1.001 MHz, depending on which HD-SDI bit rate (1.485 Gb/s or 1.485/1.001 Gb/s) is being transmitted. The 1.485 Gb/s bit rate supports 60 Hz video update rate (and derivatives of 60 Hz such as 30 Hz, 25 Hz and 24 Hz). The pathological waveforms for HD-SDI are documented in SMPTE RP 198. SMPTE RP 178 describes the similar set of pathological waveforms for SD-SDI. These waveforms are worst-case waveforms that can be generated by the SDI encoder and include a low-frequency square wave pattern and a poorly DC balanced pattern.

1. The pathological waveforms for HD-SDI are documented in SMPTE RP 198. SMPTE RP 178 describes the similar set of pathological waveforms for SD-SDI. These waveforms are worst-case waveforms that can be generated by the SDI encoder and include a low-frequency square wave pattern and a poorly DC balanced pattern.
1.485/1.001 Gb/s rate supports the 59.94 Hz video update rate (and derivatives) that are used primarily in North America.

For all of the SD-SDI bit rates, except 540 Mb/s, the reference clock must be some multiple of the SD-SDI word-rate clock since the SD video clocks are too slow to be used directly as reference clocks to the RocketIO transceiver.

The source of the reference clocks is application specific. In most cases parallel digital video is supplied to the transmitter with an associated word-rate clock from some external source. For HD-SDI, the word-rate clock would be exactly the correct frequency needed for the reference clock to the RocketIO transmitter. However, for SD-SDI, the word-rate clock would typically need to be multiplied by at least two to get a clock fast enough for the RocketIO transceiver. In either case, keep in mind that an external PLL might be required to reduce the jitter on video clocks so that they are suitable for the RocketIO transceiver reference clocks. Chapter 9 discusses jitter reduction requirements in more detail.

User Clocks

The user clocks load data into the RocketIO transceiver from the fabric of the FPGA. Each RocketIO transceiver requires two transmitter user clocks called TXUSRCLK and TXUSRCLK2.

TXUSRCLK must always be frequency locked to the selected reference clock. There is no required phase relationship between TXUSRCLK and the selected reference clock, but they must be exactly the same frequency.

The frequency and phase relationships between TXUSRCLK and TXUSRCLK2 depend on the width of the TXDATA input port of the RocketIO transceiver. For HD-SDI, it is usually most convenient to use a 20-bit wide TXDATA port as this matches the data word width of HD-SDI (10 bits of Y and 10 bits of C). When using a 20-bit TXDATA port, TXUSRCLK2 must have the same frequency and phase as TXUSRCLK (simply connect TXUSRCLK and TXUSRCLK2 to the same clock source). Consult the RocketIO Transceiver User Guide for TXUSRCLK2 requirements when other TXDATA port widths are used.

When implementing SD-SDI running at 270 Mb/s, it would be slightly more convenient to use a 40-bit wide TXDATA port. This is because each encoded bit must be replicated four times if the transceiver’s bit rate is four times the SD-SDI bit rate. Since each encoded SD-SDI word is 10 bits long, the resulting vector to the TXDATA port is 40 bits after bit replication. However, because the width of the TXDATA port is fixed at FPGA configuration time, a TXDATA port width that is suitable for both HD-SDI and SD-SDI must be chosen when implementing a multi-rate SDI interface. Because a 40-bit TXDATA port requires the use of a DCM to produce proper frequency and phase relationships between TXUSRCLK and TXUSRCLK2, it can be more desirable to use a 20-bit TXDATA port. So, the 40-bit SD-SDI bit vector resulting from bit replication must be sent to the TXDATA port as two 20-bit vectors, least significant half first.

The user clocks are usually global FPGA clocks that are also used to clock the parts of the SDI transmitter implemented in the FPGA fabric.

Clocking Example

There are many different ways to generate all the necessary reference clocks and user clocks needed for a multi-rate SDI transmitter. The details are application specific. The following example illustrates a common configuration used for a multi-rate SDI transmitter design.
In this example, shown in Figure 12-3, the parallel HD video and the parallel SD video enter the FPGA separately with separate video clocks. The HD video clock is either 74.25 MHz or 74.1758 MHz, depending on the video sample rate. In this example, the SD video clock is always 27 MHz and the SD-SDI bit rate is always 270 Mb/s.

The low-jitter HD video clock and the 54 MHz SD reference clock from the clock doubler are connected to the two REFCLK inputs of the RocketIO transceiver. The RocketIO transceiver contains a reference clock MUX that can be used to select between these reference clock sources.

The TXUSRCLK and TXUSRCLK2 inputs to the RocketIO transceiver must be the same frequency as the selected reference clock. The HD and SD reference clocks are multiplexed using a BUFGMUX to provide a global transmitter clock connected to the SDI encoder and the TXUSRCLK and TXUSRCLK2 inputs of the RocketIO transceiver. Note that much of the SD-SDI transmitter logic actually needs to run at 27 MHz, not 54 MHz. However, the global transmitter clock in the FPGA when running at SD-SDI rates is 54 MHz. So, a clock enable signal is generated from the 54 MHz clock that enables the clock of the SD-SDI transmitter logic every other clock cycle, effectively reducing the clock rate of the SD-SDI transmitter to 27 MHz. The SDI encoder module needs to have its clock enable asserted every other clock cycle for SD-SDI and every clock cycle for HD-SDI.
Since the CLK2X output of Virtex-II Pro’s DCMs doubles the input clock, it is tempting to think about using a DCM to generate the 54 MHz reference clock from the 27 MHz input clock. Normally, Xilinx does not recommend using a DCM to generate a reference clock for the RocketIO transceivers, due to the output jitter specs of the DCM. However, in this particular case, more jitter can be tolerated on the RocketIO transceiver's reference clock input because the amount of jitter allowed at 270 Mb/s is quite large.

There are some issues with the DCMs to keep in mind, however. First, the DCM does not reduce jitter. All jitter present on the DCM’s input clock is passed through the output clocks with some additional jitter added by the DCM. Thus, a low jitter input clock to the DCM is a requirement. Second, this should only be attempted with the CLK2X output of the DCM and never the CLKFX output since the CLKFX output produces more jitter. Third, use of the DCM as a clock doubler to produce a reference clock to the RocketIO transceiver has not been characterized in any way, and Xilinx cannot guarantee the RocketIO transmitter output jitter results obtained using this configuration.

More Thoughts on Clocks

In some applications, more than two reference clock sources might be needed for a multi-rate SDI transmitter. If, for example, the FPGA itself is the source of the video, such as from a video test pattern generator as described in Chapter 17, the FPGA might need both 74.25 MHz and 74.25/1.001 MHz reference clocks for HD-SDI and also 54 MHz for SD-SDI. The RocketIO transceiver actually has four reference clock inputs (two REFCLK and two BREFCLK). However, the RocketIO transceiver is configured during FPGA configuration to use either the two REFCLK or the two BREFCLK inputs. Thus, only two reference clock inputs can be used without reconfiguring the RocketIO transceiver. There are a few solutions to this problem and these are discussed in detail in Chapter 13.

For those applications where the FPGA is the source of video, a single reference clock frequency can support SD-SDI bit rates of 270 Mb/s, 360 Mb/s, and 540 Mb/s. The 1.08 Gb/s bitstream produced by the RocketIO transmitter when provided with a reference clock of 54 MHz, is exactly 4 * 270 Mb/s, 3 * 360 Mb/s, and 2 * 540 Mb/s. With a transmitter designed to replicate each encoded SD-SDI bit two times when transmitting 540 Mb/s, three times when transmitting 360 Mb/s, and four times when transmitting 270 Mb/s, all three bit rates can be derived from this one reference clock frequency.

Reference Designs

The reference design consists of several modules that are used to transmit SD-SDI using a RocketIO transceiver. These modules can be combined with the HD-SDI transmitter design described in Chapter 9 to implement a multi-rate SDI transmitter, or they can be used by themselves to implement an SD-SDI transmitter using a RocketIO transceiver.

An SDI transmitter that uses these modules usually also requires an error detection and handling (EDH) packet generator. EDH packets carry cyclic redundancy checksums (CRC) for each field allowing the receiver to check for transmission errors. EDH packets are used only for SD-SDI. HD-SDI uses a more robust error detection scheme with a CRC for each video line. An EDH processor that can generate and insert SD-SDI EDH packets is described in Chapter 6, “SD-SDI Ancillary Data and EDH Processors.”

Multi-Rate Encoder

Figure 12-4 shows the encoding algorithm used for both HD-SDI and SD-SDI. In this diagram, the encoding algorithm is shown as if implemented serially, one bit at a time.
However, in the reference design encoder, encoding is done in parallel, processing 20 bits per clock cycle for HD-SDI and 10 bits per clock cycle for SD-SDI.

![SDI Encoding Algorithm](image)

**Figure 12-4: SDI Encoding Algorithm**

Chapter 9 includes a 10-bit SDI encoder module called smpte_encoder. A single smpte_encoder module is used to encode SD-SDI. Two of these encoders are combined to implement an HD-SDI encoder as in the hdsdi_encoder module from Chapter 9.

One way to implement a multi-rate SDI encoder is to simply instantiate the hdsdi_encoder module from Chapter 9 and an additional smpte_encoder module for SD-SDI and then multiplex the outputs of the encoders into the RocketIO transceiver’s TXDATA port (after bit replication is done on the SD-SDI encoder output). The smpte_encoder module is small, so this method results in a fairly small multi-rate SDI encoder design.

However, a slightly more efficient implementation is found in the multi_sdi_encoder module. As shown in Figure 12-5, this module uses two smpte_encoder modules to encode the HD-SDI data, just like the hdsdi_encoder module. The Y channel smpte_encoder module is also used to encode the SD-SDI data. In order to switch between HD and SD modes of operation, two MUXes are added to the encoder, controlled by a signal that indicates whether the module is running in HD mode or SD mode. These MUXes select the feedback source for the Y encoder so that the feedback comes from the C encoder when encoding HD and from the Y encoder when encoding SD.
SD-SDI Bit Replication

For SD-SDI, each bit from the encoder must be replicated some number of times depending on the reference clock frequency and the SD-SDI bitstream frequency. With a 54 MHz reference clock, each bit must be replicated four times to generate a 270 Mb/s SD-SDI bitstream. This produces 40 bits from each 10-bit input word. If the TXDATA port of the RocketIO transmitter is 20 bits wide, the 40 bits from the bit replicator must be multiplexed to first provide the least significant 20 bits during one TXUSRCLK cycle and the most significant 20 bits during the next TXUSRCLK cycle.

The multi_sdi_bitrep_4X module (Figure 12-6) is an implementation of a bit replicator designed for 4X bit replication. It includes an output MUX to produce a 20-bit vector for the RocketIO transceiver’s TXDATA port. A control input indicates whether HD-SDI or SD-SDI is being encoded. If HD-SDI is encoded, the 20-bit input vector is simply passed through the module without any bit replication.

Because the transceiver transmits the MSB first, the bit replicator output vector must be bit-swapped to reverse the bit order before actually being connected to the TXDATA port of the RocketIO transceiver. The hdsdi_rio_refclk module used in the reference design includes this bit swap function. The hdsdi_rio_refclk module is a wrapper around the RocketIO...
primitive GT_CUSTOM. It is identical to the hdsdi_rio module from Chapter 9 except that the REFCLK inputs are active instead of the BREFCLK inputs. Due to the design of the SDV demo board, the REFCLK inputs must be used in this reference design instead of the BREFCLK inputs to the RocketIO transceiver.

Multi-Rate SDI Transmitter Example

The file sdv_multi_sdi_tx (Figure 12-7) contains an example of a multi-rate SDI transmitter. This file is designed to run on the Xilinx SDV demo board and the IOB constraints are specific to that board. The SDV demo board does not have the ability to accept parallel digital video, so in this example, video test pattern generators are used to produce the HD and SD video internally in the FPGA. Separate video pattern generators are used for SD-SDI (from Chapter 16) and for HD-SDI (from Chapter 17). In this example, the HD-SDI transmitter always runs at 74.25/1.001 MHz and the reference clock comes from a crystal oscillator on the SDV demo board. The 54 MHz reference clock for SD-SDI comes from another crystal oscillator.

This example includes CRC generation and insertion and line number insertion for HD-SDI as described in Chapter 9. It also includes the EDH processor design from Chapter 6 for generating and inserting EDH packets into the SD-SDI video.

In this example, txusrclk runs at 54 MHz in SD mode, twice the SD word rate. A flip-flop divides txusrclk in half to generate a clock enable signal for the SD video path. This signal enables the SD video pattern generator, EDH processor, and the encoder every other cycle of the 54 MHz clock. When running in HD mode, the SD clock enable is always negated to reduce the power consumed by the idle SD modules. The SD clock enable signal also controls the MUX in the bit replicator module (ms_bits input) to alternate between outputting the MS and LS encoded and replicated bits when running in SD mode.
A subset of the previous example illustrates how to implement an SD-SDI only transmitter using a RocketIO transceiver. This example is shown in Figure 12-8. The design includes the SD video test pattern generator and it also includes the EDH processor for EDH generation and insertion.

**Jitter Performance**

The output jitter of the RocketIO transmitter is dependent upon the amount of jitter present on the reference clocks, the intrinsic jitter of the RocketIO transceiver, jitter added by the cable driver, and any other jitter source.
The output jitter of an HD-SDI transmitter built with a RocketIO transceiver is documented in Chapter 9. When generating SD-SDI bitstreams using the technique described here, the RocketIO transmitter can produce extremely low amount of output jitter as compared to standard SD-SDI transmitters available on the market. Table 12-1 shows the SD-SDI transmitter output jitter typically measured using the Xilinx SDV demo board. The low-jitter (less than 40 ps peak-to-peak) crystal oscillator was used as the reference clock to the RocketIO transceiver for these measurements. The jitter measurements include all sources of jitter on the board, including the cable driver. The measurements were taken with an SDI waveform analyzer connected to the SDV demo board with 1 meter of coax cable.

Table 12-1: Typical SD-SDI Transmitter Output Jitter Values Using RocketIO MGTs

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing Jitter</td>
<td>0.059 UI</td>
</tr>
<tr>
<td>Alignment Jitter</td>
<td>0.049 UI</td>
</tr>
</tbody>
</table>

Reference Design Size

Table 12-2 shows the FPGA resources used by the reference design. The size for just the multi-rate SDI encoder plus bit replicator function is shown. Also shown are the sizes of the two SDI transmitter examples, the multi-rate SDI transmitter and the RocketIO-based SD-SDI.

The largest piece of both of the transmitter examples is the SD-SDI EDH processor. In fact, Chapter 6 shows that the implementation size of the EDH processor alone is actually larger than the entire SD-SDI transmitter example shown in Table 2 (and this example includes the EDH processor). In these examples, many of the receiver functions of the EDH processor module are not used and are optimized away by the implementation tools.

In all cases, the results were obtained using XST running under ISE 6.1. Area optimization was used. All designs were able to meet the necessary timing constraints using a Virtex-II Pro -5 speed grade device.

Table 12-2: Reference Design Implementation Sizes

<table>
<thead>
<tr>
<th>Reference Design</th>
<th>FFs</th>
<th>LUTs</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-rate encoder and bit replicator</td>
<td>61</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Multi-rate SDI Tx for SDV demo board</td>
<td>597</td>
<td>1013</td>
<td>6</td>
</tr>
<tr>
<td>SD-SDI only Tx for SDV demo board</td>
<td>420</td>
<td>725</td>
<td>3</td>
</tr>
</tbody>
</table>

Conclusions

This chapter describes a technique that can be used to allow slow SD-SDI bitstreams to be transmitted using the RocketIO multi-gigabit transceivers in the Virtex-II Pro FPGA family. Combined with the HD-SDI transmitter design described in Chapter 9, a multi-rate SDI transmitter can be implemented that supports both SD-SDI and HD-SDI.

All Virtex-II Pro devices contain multiple RocketIO transceivers, making it possible to implement four or more multi-rate SDI transmitters in a single Virtex-II Pro device. For applications that require multiple multi-rate SDI transmitters, this can provide a high level of integration, saving board space, power, and money as compared to discrete multi-rate SDI transmitter implementations.
Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_hdsd-tx-mgt.zip.
Chapter 13

Multi-Rate HD/SD-SDI Receiver Using RocketIO Multi-Gigabit Transceivers

Summary

The SD-SDI standard is widely used in broadcast studios and video production centers to transport standard definition (SD) digital video serially over video coax cable. The HD-SDI standard is similar, but transports high-definition (HD) digital video. The SD-SDI and HD-SDI standards are similar enough that it is possible to implement interfaces for video equipment that support both standards through the same connector.

This chapter describes how to use the RocketIO™ multi-gigabit transceivers available in the Virtex™-II Pro family of FPGA devices to implement a receiver that can support both SD-SDI and HD-SDI. The flexibility of the RocketIO transceivers, combined with the programmable logic of the Virtex-II Pro devices, makes it possible to implement multi-rate SDI interfaces.

Introduction

The SD-SDI standard, traditionally known simply as SDI, is defined by the SMPTE 259M standard [Ref 1] and also by the equivalent ITU-R BT.656 standard [Ref 2]. The SMPTE 292M standard defines HD-SDI. SD-SDI is widely used in broadcast studios and video production centers today. Use of HD-SDI is increasing rapidly as the broadcast industry ramps up support for HDTV broadcasting.

Throughout this document, the term SD-SDI is used to refer to the SD version of the interface standard. SDI is used to refer generically to both SD-SDI and HD-SDI. And, multi-rate SDI is used to refer to support of both standards.

This chapter focuses specifically on the implementation of a multi-rate SDI receiver using the RocketIO transceivers available in the Virtex-II Pro FPGA family. Chapter 12 describes the implementation of multi-rate SDI transmitters, also using RocketIO transceivers.

The RocketIO transceivers are designed to support serial bit rates from 622 Mb/s to 3.125 Gb/s. HD-SDI, with bit rates of approximately 1.5 Gb/s, falls within this range. However, all of the current SD-SDI bit rates, ranging from 143 Mb/s to 540 Mb/s, are well below the range supported by the RocketIO transceivers. Therefore, the main focus of this chapter is on a technique that allows the RocketIO receiver to support these slower bit rates without violating any of the specifications of the RocketIO transceivers. Chapter 10 describes how to use the RocketIO transceiver to build an HD-SDI receiver. The combination of the SD-SDI reception technique described here and the HD-SDI receiver from Chapter 10 results in a multi-rate SDI receiver design. SD-SDI only receivers can also be built using the RocketIO transceivers or they can be implemented in the fabric of the FPGA as described in the SD-SDI chapters.
The reference design presented here has been tested and verified on the Xilinx SDV demo board [Ref 3].

SD-SDI and HD-SDI Similarities and Differences

The basic electrical specifications of HD-SDI and SD-SDI are virtually identical, making it possible to design multi-rate SDI interfaces that support both standards. Both standards have the same basic electrical interface: a singled-ended signal with an 800 mV peak-to-peak swing centered around 0.0V. Both use 75Ω coaxial cable and BNC connectors. And, both use the same encoding algorithm.

The most obvious difference between HD-SDI and SD-SDI is the much higher bit rate required to support HD video. HD-SDI has two bit rates: 1.485 Gb/s and 1.485/1.001 Gb/s. SD-SDI bit rates range from 143 Mb/s to 540 Mb/s.

Another difference is that HD video is 20 bits wide with two 10-bit channels, one for chroma and one for luma. An HD-SDI transmitter encodes and transmits 20 bits for every video clock cycle, whereas SD-SDI only encodes and transmits 10 bits per video clock cycle.

Multi-Rate SDI Receiver Functions

This section describes the basic functions implemented by a multi-rate SDI receiver. Figure 13-1 is a block diagram of a typical multi-rate SDI receiver. The following sections describe the basic functions of the receiver.

Cable Equalization

The HD-SDI standard allows HD digital video to be sent over video coax cables up to 100 meters in length. SD-SDI, with its slower bit rates, allows maximum cable lengths of up to 300 meters.

The coax cable causes frequency-dependent attenuation of the signal, where the higher frequency components of the signal are attenuated more than the lower frequency components. The coax cable also causes frequency-dependent phase distortion, where the
higher frequency components are phase shifted more than the lower frequency components. After passing through long coax cables, the SDI signal is severely distorted and attenuated. The receiver must compensate for this attenuation and distortion before attempting to recover the signal.

Cable length equalization is used to compensate for the attenuation and distortion introduced by the coax cable. Typically, an adaptive cable length equalizer is used in SDI receivers. Such an equalizer actively monitors the amount of attenuation and distortion present on the incoming signal and applies the correct amount of equalization to the signal. The cable length can be changed without requiring a change to the equalizer, as would be the case if fixed length equalization were used.

Clock and Data Recovery

After cable equalization, the SDI receiver recovers the clock and data from the SDI bitstream. This is typically done with a PLL-based clock and data recovery (CDR) unit. A recovered clock is often required for the receiver because SDI does not support clock correction to allow the incoming bitstream to be easily resynchronized to a local reference clock. Instead, the recovered clock from the CDR unit is typically used to clock all the SDI receiver logic downstream from the CDR unit.

Deserialization

After the CDR unit, the serial bitstream is typically deserialized to produce a parallel data stream. While it is possible to implement SDI decoding and framing in a serial fashion, doing so at the HD-SDI clock rates of 1.485 Gb/s is not possible in today's FPGA technology. Instead, the HD-SDI bitstream is deserialized into 20-bit words for processing by the decoder and framer. The serial clock is divided by 20 to derive a word-rate clock for the downstream receiver functions. For SD-SDI, it is usually more convenient to deserializel the bitstream into 10-bit words and divide the clock by 10.

Decoding

Both HD-SDI and SD-SDI use the same two-stage encoding scheme. The first stage performs pseudorandom scrambling and the second stage performs non-return-to-zero (NRZ) to NRZ-inverted (NRZI) conversion. After recovering the data, the SDI receiver must decode the data by reversing the two encoding steps—first converting the NRZI data to NRZ and then descrambling the data. Figure 13-2 shows conceptually how the SDI bitstream is decoded in a serial manner.

The RocketIO transceivers have built-in 8B/10B decoders. However, they do not have SDI decoders. So, the recovered data from the RocketIO transceiver must bypass the decoding logic built into the RocketIO transceiver and be sent directly to the transceiver's RXDATA port still encoded. An SDI decoder built in the fabric of the FPGA implements the SDI
decoding algorithm on the recovered data. The data is decoded in a parallel manner—
20 bits are decoded per clock cycle for HD-SDI and 10 bits per clock cycle for SD-SDI.

Framing

The data words from the SDI decoder are not word aligned. The CDR unit has no concept
of where the video sample boundaries lie in the continuous stream of incoming bits. The
decoder does not care where the video sample boundaries lie since it can decode the data
without this information. However, after decoding, it is necessary to identify the sample
boundaries and realign the data. This process of realigning the data is called framing.

The framer in the SDI receiver monitors the decoded data and looks for the bit sequences
that mark the beginning of the timing references. There are two timing references per video
line: the end-of-active video (EAV) and the start-of-active video (SAV). Both the EAV and
SAV have the same format and are four 10-bit words long. The first three words are always
fixed values. The first word of the timing reference is a word of all ones and has a hex value
of 3FFh. The second and third words of the timing reference are made up of all zeros
(000h). The fourth word of the timing reference is called the XYZ word. Figure 13-3 shows
the format of the XYZ word of the timing reference.

In HD-SDI video streams, there are two separate 10-bit channels called the luma (Y)
channel and the chroma (C) channel. Each channel has its own set of timing references. The
channels are considered to be synchronous so that the first word of the EAV, for example,
would appear on both the Y channel and C channel simultaneously.

Before transmission by the HD-SDI transmitter, the Y and C channels are interleaved so
that a C word is transmitted first followed immediately by the corresponding Y word.
Figure 13-4 shows the details of this interleaving.
Multi-Rate SDI Receiver Functions

In SD-SDI video, there are also Y and C words for each sample, but the components are always interleaved and there is a single timing reference for the entire SD-SDI video stream, as shown in Figure 13-5.

The framer looks for the unique 3FFh, 000h sequence that marks the beginning of the EAV and SAV sequences. Only this unique pattern can be used as a reference point for realigning the data. For SD-SDI, the framer can simply look for this sequence, but for HD-SDI, due to the interleaving of the Y and C channels, the timing reference actually looks like this: 3FFh, 3FFh, 000h, 000h, 000h, 000h. A multi-rate SDI framer must be capable of detecting the timing reference sequence for both SD-SDI and HD-SDI.

The framer must look for the timing reference sequence pattern beginning at any possible bit position in the data words from the decoder. Once the pattern is identified, the framer knows the bit offset of the first bit of each data word. A barrel shifter is used to realign the data to proper word boundaries.

For more details about the framing function, refer to Chapter 10.

Error Checking

Error checking is done much differently in HD-SDI than it is in SD-SDI. Error checking in SD-SDI was added after the original SD-SDI specification was released. As a result, error checking in SD-SDI is quite cumbersome. In contrast, error checking was implementing from the beginning in HD-SDI in a much more elegant and robust manner.

In HD-SDI, the transmitter calculates two 18-bit CRC values for each video line, one for the Y channel and one for the C channel. Each CRC value is separately formatted into two 10-bit words and placed into the video stream two words after the end of the EAV. The two words between the EAV and the CRC contain a value indicating the line number. Chapter 9 has details of how the CRC values are computed for each line. The starting and ending
positions are all related to the positions of the EAV and SAV on each line, making it easy to compute the CRC values and to locate the position where the CRC values are located in the video stream. The transmitter and receiver don’t need to have the details of the video format, such as number of words per line and lines per frame, in order to generate or check the CRC values on each line.

In SD-SDI, the transmitter calculates two 16-bit CRC values for each frame of video. One CRC value is called the full-field (FF) CRC and is calculated for all words in most (but not all) of the video lines of a field. The other CRC value is called the active-picture (AP) CRC and is calculated on just the active portion of the field. These two CRC values, plus some additional error flags, are formatted into a 23-word long packet called the error detection and handling (EDH) packet. The EDH packet must be located immediately prior to the SAV on a specific video line in each field. The line where the EDH packet is located depends on the video format. Details of how EDH packets are calculated can be found in Chapter 6, “SD-SDI Ancillary Data and EDH Processors.”

In order to generate or check the EDH packet in an SD-SDI video stream, the transmitter or receiver must first determine the format of the video since the details of how the CRC values are calculated and the position of the EDH packet are different for each video format. This greatly complicates the process of error checking in SD-SDI. In fact, the EDH processor function accounts for almost half of the logic in the multi-rate SDI receiver reference design.

Because there is no commonality between HD-SDI and SD-SDI error checking, a multi-rate receiver generally needs two separate error checking modules, one for HD and one for SD.

Rate Selection

When implementing a multi-rate SDI receiver, the RocketIO transceiver requires a separate reference clock frequency for each bit rate that is to be received. For a typical SDI receiver capable of receiving both HD-SDI bit rates plus the 270 Mb/s SD-SDI bit rate, three reference clock frequencies are needed for the RocketIO transceiver.

In some cases, the SDI receiver can simply be told which bit rate to expect, making selection of the proper reference clock relatively straightforward. However, in most cases, SDI receivers are expected to automatically detect the frequency of the bitstream. This is the function of the rate selection block in Figure 13-1. It forms a feedback loop, monitoring the data being received for errors and cycling through the reference clocks until the receiver locks to the bitstream.

Implementing the Multi-Rate SDI Receiver

This section details how to implement a multi-rate SDI receiver using the RocketIO transceivers in the Virtex-II Pro FPGAs.

Cable Equalization

As previously described, an SDI receiver usually has an adaptive cable length equalizer to compensate for attenuation and distortion of the signal caused by long runs of coax cable. The RocketIO transceivers in the Virtex-II Pro FPGA do not include adaptive cable length equalizers. So, an external cable equalizer must be used to interface the SDI cable to the RocketIO transceiver. As a side benefit, the cable equalizer also converts the single-ended SDI signal into a differential signal. The CML inputs of the RocketIO receiver require a differential input signal. Most multi-rate SDI cable equalizers currently available have 3.3V LVPECL outputs that are not directly compatible with the 2.5V CML inputs of the
RocketIO transceiver. AC coupling can be used to interface the LVPECL outputs of the cable equalizer to the CML inputs of the RocketIO transceiver. Figure 13-6 shows a typical AC coupled interface between a Gennum GS1524 cable equalizer and a RocketIO transceiver.

There are several important details in Figure 13-6:

- The recommendations given in the GS1524 data sheet [Ref 4] must be followed for the interface network between the BNC cable connector and the GS1524's input.
- The AC coupling capacitors between the GS1524 and the RocketIO receiver must be in the 1 µF to 10 µF range to pass the SDI pathological waveforms without too much voltage droop. Typically, 4.7 µF capacitors are used.
- The input impedance of the RocketIO transceiver should be set equal to impedance of the circuit board traces between the cable equalizer and the transceiver's inputs. Normally this is 50Ω.
- As described in the RocketIO Transceiver User Guide [Ref 5], when using AC coupling, the RocketIO receiver termination voltage (VTRX) must be between 1.6V and 1.8V. As shown in Figure 13-6, the required termination voltage can be generated from 2.5V by using a voltage divider network. The resistor values shown are sized to supply the termination voltage to a single RocketIO transceiver, so this resistor network must be duplicated for each RocketIO transceiver used as an SDI receiver.
- Some Virtex-II Pro devices have internal power filter capacitors for the VTRX, VTTX, AVCCAUXRX, and AVCCAUXTX signals of each RocketIO transceiver. Consult the RocketIO Transceiver User Guide for more information.
- It is absolutely essential that all guidelines given in the RocketIO Transceiver User Guide for layout, bypass capacitors, and power regulation and filtering be observed. Do not attempt to provide power to the RocketIO transceiver from a switching regulator.
Receiving SD-SDI Bitstreams with the RocketIO Transceiver

The original SD-SDI standard supports four bit rates:

- 143 Mb/s for NTSC digital *composite* video
- 177.3 Mb/s for PAL digital *composite* video
- 270 Mb/s for NTSC and PAL digital *component* video
- 360 Mb/s for NTSC and PAL 16:9 aspect ratio digital *component* video

In addition, a more recent document (SMPTE 344M) adds a 540 Mb/s bit rate compatible with SD-SDI.

The digital composite video rates of 143 Mb/s and 177.3 Mb/s are rarely used. The most commonly used bit rate, by far, is 270 Mb/s. It is quite common for a piece of video equipment to support only the 270 Mb/s bit rate through its SDI interface. It is also common for equipment to support both 270 Mb/s and 360 Mb/s. The 540 Mb/s bit rate is relatively new and not widely supported, yet.

All of these bit rates are well below the 622 Mb/s minimum bit rate supported by RocketIO transceivers in Virtex-II Pro devices. Therefore, it is necessary to work around this minimum bit rate limitation in order to support SD-SDI with the RocketIO transceivers.

As described in Chapter 12, SD-SDI bitstreams can be transmitted with the RocketIO transceiver by simply over-clocking the transceiver at some multiple of the SD-SDI bit rate and then sending each encoded bit multiple times.

Trying to receive SD-SDI bitstreams by simply over-clocking the RocketIO transceiver and relying on the CDR unit of the transceiver to lock to a harmonic of the bitstream frequency does not work. The primary issue is run length limitations. SD-SDI has a maximum run length of 39 consecutive bits without a transition.\(^1\) This is well within the RocketIO transceiver’s maximum run length limit of 75 bits. However, when over-sampling the bitstream by 3X, the minimum over-sampling rate sufficient to meet the minimum frequency requirements of the RocketIO transceiver with a 270 Mb/s input bitstream, the 39-bit long maximum run length of the SD-SDI bitstream appears to the RocketIO transceiver as being three times as long (117 bits long). This clearly exceeds the maximum run length limit of the RocketIO transceiver by more than 50%. Thus, the CDR unit in the RocketIO transceiver does not maintain lock with a 270 Mb/s SD-SDI bitstream when simple over-sampling is attempted.

The CDR unit in the RocketIO transceiver was not designed to work with bitstreams with the frequency and characteristics of SD-SDI. To overcome this, Xilinx has developed an over-sampling technique that can be used with the RocketIO transceivers. This technique does not rely on the CDR section of the RocketIO transceiver for either clock or data recovery. The receiver section of the RocketIO transceiver is used as an asynchronous sampler. Data recovery is done in a data recovery unit (DRU) implemented in the fabric of the FPGA. The PLL in the CDR section of the RocketIO transceiver does not have to lock to the bitstream frequency for this technique to work. Therefore, the SD-SDI run lengths are not an issue for the RocketIO transceiver when using this technique.

The DRU takes the raw, over-sampled bitstream data from the output of the RocketIO transceiver. It scans the over-sampled data, looking for bit transitions. From these

---

1. SD-SDI bitstreams carrying PAL digital *composite* (\(4f_{sc}\)) video can have a maximum run length of 40 bits. However, this chapter is mainly targeted at digital *component* video, which has a worst case SD-SDI run length of 39 bits.
transitions, it determines the optimum place to sample each bit, producing a recovered bitstream on its output.

The over-sampling technique allows the RocketIO transceivers in the Virtex-II Pro devices to receive SD-SDI bitstreams of 360 Mb/s and slower. It does not, however, work for 540 Mb/s bitstreams. The minimum supported 8X over-sampling rate at 540 Mb/s exceeds the maximum bit rate supported by the RocketIO transceivers in the Virtex-II Pro.

There are two versions of the DRU. Version 1 synthesizes a recovered 27 MHz clock by dividing a global 270 MHz clock by 10. Version 2 generates a clock enable output. This clock enable is used with a global clock to allow all downstream logic in the SDI receiver to run at the 27 MHz SD rate.

**DRU Version 1**

Figure 13-7 shows an example of how the RocketIO transceiver and the DRU (version 1) work together to receive a 270 Mb/s SD-SDI bitstream. In this example, the RocketIO transceiver is over-sampling the bitstream by 8X (2.16 GHz sample rate).

![RocketIO Transceiver and Data Recovery Unit](image)

*Note: All frequencies listed are for 270 Mbps SD-SDI.

**Figure 13-7: RocketIO Transceiver and Data Recovery Unit**

The Xilinx DRU used in this application can work at sampling rates between 8 and 11 times the bitstream frequency. The RocketIO transceiver multiplies the selected reference clock by 20, so the correct RocketIO transceiver reference clock frequency is calculated as:

\[
\text{reference clock freq} = \frac{\text{bitstream freq} \times \text{over-sample rate}}{20} \quad \text{Eq. 1}
\]

For example, if 8X over-sampling is used on a 270 Mb/s bitstream, the reference clock must be 108 MHz. With a 108 MHz reference clock, the RocketIO transceiver is actually running at 2.16 Gb/s—that is, it is sampling the bitstream at a 2.16 GHz rate. At the time of this writing, -5 speed grade Virtex-II Pro parts have a maximum bit rate speed for their RocketIO transceivers of 2.0 Gb/s. However, since January 2004, selected -5 speed grade Virtex-II Pro devices can be obtained from Xilinx that have RocketIO transceivers tested to 2.5 Gb/s, allowing the -5 speed grade devices to receive 270 Mb/s SD-SDI bitstreams.

Version 1 of the DRU synthesizes a word-rate recovered clock called rdclk. To synthesize this clock, the DRU requires a reference clock (rdclk_ref) running at 10X the word-rate. This clock can come from a local oscillator and does not have to be frequency locked to the
The DRU normally divides \text{rdclk}_{\text{ref}} by 10 to produce the \text{rdclk} word-rate clock. However, since \text{rdclk}_{\text{ref}} is usually not running at exactly the same frequency as the bitstream, the DRU must occasionally make up for the slight differences between the bitstream frequency and the \text{rdclk}_{\text{ref}} frequency. So, occasionally, the recovered clock is 9 or 11 \text{rdclk}_{\text{ref}} cycles long, instead of the normal 10 cycles. This keeps the recovered clock in step with the rate that the data is being recovered by the DRU. The recovered data from the DRU always changes synchronously with the rising edge of \text{rdclk}, even when the period of \text{rdclk} is adjusted by the DRU as shown in Figure 13-8.

![Figure 13-8: Synthesis of SD-SDI Recovered Clock (DRU Version 1)](image)

The \text{rdclk}_{\text{ref}} signal doesn't have to come from a separate oscillator. In fact, since the frequency of this clock and the frequency of the reference clock required by the RocketIO transceiver are related, \text{rdclk}_{\text{ref}} can be generated in a DCM by multiplying the RocketIO transceiver reference clock by some factor. For example, in Figure 13-11, a single DCM generates both the 108 MHz RocketIO transceiver reference clock and the 270 MHz clock for the DRU from a 54 MHz source.

**DRU Version 2**

Version 2 of the DRU does not synthesize a recovered clock. Instead, it requires a reference clock called \text{rdclk}, which it divides by a set factor to produce a clock enable output called \text{rdclkDvEn}. If, for example, \text{rdclk} is running at 108 MHz, the DRU would divide it by 4 and assert the \text{rdclkDvEn} for one cycle of every four cycles of \text{rdclk}. The \text{rdclkDvEn} clock enable and \text{rdclk} can then be used to clock logic downstream from the DRU at 27 MHz. \text{rdclk} can be any integer multiple of the 27 MHz SD word rate from 4 to 10. Half-integer multiples (4.5X, 5.5X, etc) are also supported. This version of the DRU corrects for differences between the local \text{rdclk} frequency and the recovered data rate by sometimes inserting or removing one clock cycle between \text{rdclkDvEn} assertions. For example, if a clock divider of 4 is being used, occasionally, there are 2 or 4 \text{rdclk} cycles between assertions of \text{rdclkDvEn}, rather than the normal 3. The data out of the DRU changes synchronously with the rising edge of \text{rdclk} when \text{rdclkDvEn} is asserted. Figure 13-9 shows the timing of the version 2 DRU.
Implementing the Multi-Rate SDI Receiver

The rdclk reference clock can come from a local oscillator that is totally independent of the recovered clock from the RocketIO transceiver. However, one convenient way to use the version 2 DRU is to use the recovered clock (RXRECCLK) from the RocketIO transceiver as the rdclk reference clock. See the reference design description for more details.

RocketIO Transceiver Clocks

The RocketIO transceiver requires two types of clocks: reference clocks and user clocks. The reference clocks are used by the RocketIO transceiver as a reference for the CDR PLL. The user clocks are used to clock data out of the RocketIO transceiver and into the fabric of the FPGA. In addition, the RocketIO receiver also produces a recovered clock, called RXRECCLK.

The following sections describe the clocking requirements of the RocketIO transceivers oriented towards implementing multi-rate SDI interfaces. More details about the clocking requirements of the RocketIO transceivers can be found in the RocketIO Transceiver User Guide [Ref 5].

Reference Clocks

The RocketIO transceiver uses reference clocks for two different purposes:

1. In the transmitter, the reference clock provides a low-jitter frequency reference that the transmitter multiplies by 20 to obtain a bit-rate clock for the transmitter’s serializer.
2. In the receiver, the reference clock is used to spin up the CDR unit so that it quickly locks to the incoming bitstream. After the CDR unit is locked to the bitstream, the frequency of the PLL is constantly compared to the frequency of the reference clock to determine if the PLL is maintaining lock to a valid bitstream frequency.

The reference clocks are required to be 1/20th the frequency of the bitstream ±100 ppm.

For HD-SDI, there must be two reference clock frequencies, one for each of the two standard HD-SDI bitstream frequencies. The RocketIO transceiver needs reference clocks of 74.25 MHz and 74.25/1.001 MHz for HD-SDI. For SD-SDI, the reference clock determines the rate at which the bitstream is sampled by the RocketIO transceiver. The reference clock frequency calculation is shown in Equation 1.

Each RocketIO transceiver has four reference clock inputs from which a single reference clock is selected. There are two pairs of reference clocks called REFCLK and BREFCLK. The difference between the pairs of reference clocks is that the BREFCLK inputs are designed for lowest possible jitter. The BREFCLks can only come from certain special IOBs. By contrast, the REFCLks can come from any IOB or from anywhere in the FPGA.

*DRU frequency adjustment takes place here, shortening the clock period by one rdclk cycle.

![Figure 13-9: Clock Enable Timing (DRU Version 2)](Image)
As shown in Figure 13-10, a set of MUXes selects one active reference clock from the four reference clock inputs. An input signal called REFCLKSEL chooses one reference clock from each pair. A final MUX, controlled by a RocketIO attribute called REF_CLK_V_SEL, chooses between the REFCLK or BREFCLK input.

![Reference Clock Selection](x84_09_031704)

**Figure 13-10: Reference Clock Selection**

The REF_CLK_V_SEL attribute is initialized when the FPGA is configured and normally cannot be changed until the FPGA is configured again. So, in normal operation, the RocketIO transceiver is really limited to just two reference clock inputs that can be selected dynamically by the REFCLKSEL input. Since supporting two HD-SDI rates plus one SD-SDI rate requires the use of three reference clocks, a work-around to this limitation must be employed.

There are several possible solutions. The simplest solution is to multiplex two of the reference clocks together externally to the RocketIO transceiver or externally to the FPGA. Another solution is to reconfigure the RocketIO transceiver dynamically to change the REF_CLK_V_SEL attribute.

Xilinx Application Note XAPP660 [Ref 6] describes a method of using the embedded PowerPC processor available in most Virtex-II Pro devices to reconfigure one or more RocketIO transceivers dynamically. Since a PowerPC is not available in the smallest Virtex-II Pro device, a PicoBlaze soft processor can be used to reconfigure the RocketIO transceivers in this smallest device. There are some disadvantages to reconfiguring the RocketIO transceiver as opposed to MUXing the reference clocks externally. First, it takes several hundred microseconds for the PowerPC to reconfigure the RocketIO transceiver. Thus, the switching time of the external MUX solution is much faster. Second, if the PowerPC is being used for other purposes, then it usually cannot also implement the RocketIO reconfiguration.

The external MUX solution also has a disadvantage when multiple SDI interfaces are implemented in the same FPGA. Unless all the SDI interfaces can always run at the same HD-SDI bit rate, then a separate MUX is required for each RocketIO transceiver used as an SDI interface. Also, care must be taken that the two closely related HD-SDI reference clock frequencies do not mix (heterodyne) in the MUX, causing excessive jitter on the reference clock.

In the reference design section, an external frequency synthesizer generates the two HD-SDI reference frequencies based on a control signal from the RocketIO transceiver. This solution has the same advantages and disadvantages as the external MUX solution.
Since the CDR unit in the RocketIO transceiver is not really used for receiving SD-SDI, the normal reference clock requirements are not directly applicable. We recommend that the SD-SDI reference clock have no more than 200 ps peak-to-peak of jitter and that it be within ±1000 ppm of the calculated reference clock frequency. If the reference clock jitter is excessive, the PLL in the transceiver might not lock to the reference clock, causing the transceiver to sample the bitstream improperly.

User Clocks

The user clocks clock data out of the RocketIO transceiver and into the fabric of the FPGA. Each transceiver requires two user clocks on the receiver side called RXUSRCLK and RXUSRCLK2. Each transceiver also has two user clocks for the transmitter side called TXUSRCLK and TXUSRCLK2. If the transmitter portion of the transceiver is not used, the TXUSRCLK and TXUSRCLK2 inputs must still be driven with valid clock signals. In this case, simply connect TXUSRCLK to RXUSRCLK and TXUSRCLK2 to RXUSRCLK2.

RXUSRCLK is the clock signal that clocks data out of the RocketIO transceiver. The receiver’s output ports, such as RXDATA, change synchronously with the rising edge of RXUSRCLK. For HD-SDI, the frequency of RXUSRCLK is equal to the word-rate of the HD-SDI interface, either 74.25 MHz or 74.25/1.001 MHz. For SD-SDI, the frequency of RXUSRCLK is equal to reference clock frequency.

The frequency and phase relationships between RXUSRCLK and RXUSRCLK2 depend on the width of the RXDATA port of the RocketIO transceiver. For HD-SDI, a 20-bit RXDATA port is convenient to use because it matches the data word width of HD-SDI (10 bits of Y and 10 bits of C). The DRU used for SD-SDI expects the RXDATA port of the RocketIO transceiver to be 20 bits wide. When using a 20-bit wide output data path from the RocketIO transceiver, RXUSRCLK2 must have the same frequency and phase as RXUSRCLK (simply connect RXUSRCLK and RXUSRCLK2 to the same clock signal).

In serial protocols that have clock correction capability, the RXUSRCLK and RXUSRCLK2 signals usually are derived from the same source as the reference clock. The RocketIO transceiver’s clock correction capability is used to occasionally insert or remove idle characters to compensate for the minor differences between the actual clock frequency of the incoming bitstream and the frequency of the local reference clock.

SDI does not support clock correction. Therefore, deriving the transceiver’s user clocks from the reference clock would quickly result in an overflow or underflow condition on the output data port of the RocketIO receiver because RXUSRCLK and RXUSRCLK2 would have a slightly different frequency than the bitstream.

When implementing the HD-SDI receiver, the recovered clock (RXRECCLK) from the RocketIO receiver is used as the source of RXUSRCLK and RXUSRCLK2. When connected in this manner, RXUSRCLK and RXUSRCLK2 always run at the same frequency as the CDR PLL in the RocketIO transceiver. Thus, underflow and overflow conditions are prevented.

For SD-SDI, RXUSRCLK and RXUSRCLK2 should also be driven by RXRECCLK. For SD-SDI over-sampling, the PLL in the RocketIO transceiver is either locked to a harmonic of the SD-SDI bitstream or it is locked to the reference clock input. In either case, the RXRECCLK, which is derived from the PLL, indicates the rate at which the over-sampled data is being captured by the transceiver. RXRECCLK is used to clock data out of the transceiver, by connecting it to RXUSRCLK and RXUSRCLK2, and to clock data into the SD-SDI DRU.
Reference Design

The multi-rate SDI reference design supports the two HD-SDI bit rates and 270 Mb/s SD-SDI.

A high-level description of the reference design is given in the following section. Detailed information about the reference design can be found in “Appendix A: Reference Design Details.”

Multi-Rate SDI Reference Design

There are two versions of the reference design, one for each version of the DRU. Figure 13-11 shows the top level of the multi-rate SDI receiver reference design for version 1 of the DRU. This reference design was designed to run on the Xilinx SDV demo board, but can easily be adapted to other implementations. The top-level module is called sdv_multi_sdi_rx_v1. It contains the clock generation and distribution, the RocketIO module, the SD-SDI DRU, and a module called multi_sdi_rx containing the bulk of the multi-rate receiver logic.

Figure 13-12 shows the top level of the multi-rate SDI receiver design for version 2 of the DRU. It is nearly identical to version 1, with only minor changes involving the clocking of the DRU and downstream logic. Note that when using version 1 of the DRU, two global clock buffers are usually required per SDI receiver channel. With version 2, however, only one global clock is used by the receiver.
On the Xilinx SDV demo board, there are no provisions for bringing the received parallel video out of the Virtex-II Pro FPGA. So, the received video is simply checked for CRC and EDH errors to determine correct reception with LEDs indicating the error status.

Clocks (DRU Version 1)

On the SDV demo board, an ICS660 clock synthesizer is used to generate the two HD-SDI reference clocks of 74.25 MHz and 74.1758 MHz from a 27 MHz reference crystal. For new designs, the newer ICS664-01 and ICS664-02 are recommended because they have better jitter performance than the ICS660. The ICS664-02 has a differential output and the ICS664-01 has a single-ended output. An output from the multi_sdi_rx module, called hd_rate, selects which of the two HD-SDI reference clocks is to be generated by the ICS660.

To recover 270 Mb/s SD-SDI bitstreams with 8X over-sampling, the RocketIO module needs a 108 MHz reference clock. Ideally, a 108 MHz oscillator would be provided. However, the SDV demo board lacks such an oscillator. Instead, a 54 MHz oscillator is used. This oscillator is connected to the CLKIN of a DCM. The CLK2X output of the DCM provides the 108 MHz reference clock to the RocketIO. The CLKFX output of the DCM provides the 270 MHz rdclk_ref signal needed by the SD-SDI DRU by multiplying the 54 MHz input clock by 5.

While driving the RocketIO transceiver’s reference clock input with a DCM is not normally recommended, keep in mind that, in this application, the CDR PLL is not used for clock and data recovery. Our experimentation with this configuration has shown that jitter on the RocketIO reference clock does affect the SD-SDI DRU and providing a low-jitter reference clock for the RocketIO provides better input jitter tolerance for the DRU. Using the 2X clock output of a DCM, given a very low-jitter clock source as the reference to the DCM, does produce satisfactory results. However, do not attempt to use the CLKFX output.
output of the DCM for the RocketIO transceiver reference clock. The CLKFX output has too much jitter and does not work.

None of the outputs of the DCM need to be buffered by a BUFG buffer. The CLK0 and CLK2X outputs only drive single loads and the CLKFX output only drives 12 loads in the DRU.

The HD reference clock from the ICS660 is connected to the REFCLK2 input of the RocketIO transceiver. The 108 MHz SD reference clock from the DCM is connected to the REFCLK input of the RocketIO transceiver. An output from the multi_sdi_rx module, called hd_sd, is connected to the REFCLKSEL input of the Rocket transceiver to select between these two reference clock sources.

The RXRECLK output of the RocketIO transceiver runs at the word rate of the HD video stream when receiving HD-SDI. In HD mode, a BUFGMUX passes RXRECLK onto the multi_sdi_rx module as the receiver clock. In SD mode, the BUFGMUX passes the recovered clock from the SD-SDI DRU (rdclk_out) to the multi_sdi_rx module.

In SD mode, RXRECLK is about the same frequency as the SD reference clocks (108 MHz), although its frequency can vary slightly as the transceiver’s CDR PLL attempts to lock to the bitstream. RXRECLK is buffered by a BUFG and sent to the rxrecclk input of the SD-SDI DRU to clock the data from the RXDATA output port of the transceiver into the DRU.

For both HD and SD modes, the buffered RXRECLK output from the transceiver is connected to the four user clock inputs of the RocketIO transceiver. The data from the RXDATA port changes synchronously with the rising edge of the buffered RXRECLK signal.

Clocks (DRU Version 2)

Version 1 of the DRU needs a high-frequency rdclk_ref so that the output phase jitter caused by the clock correction in the DRU can be kept to a minimum. Version 2 of the DRU does not have this requirement. The rdclk input to the DRU can be driven by a clock running at any integer or half-integer multiple of the 27 MHz SD clock rate from 4X to 10X.

The clock source for rdclk is also used to clock all downstream logic in the SD receiver. The rdclkDvEn output from the DRU is used in conjunction with rdclk to cause this downstream logic to run at 27 MHz when running in SD mode.

Usually, the most convenient way to use the version 2 DRU is use the RXRECLK output of the RocketIO transceiver, buffered by a BUFG, as the rdclk source. This global RXRECLK signal can then drive all downstream receiver logic.

This is particularly convenient when building a multi-rate receiver. In HD mode, RXRECLK runs at exactly the word rate of the HD data being received by the RocketIO transceiver, and is the correct frequency needed to clock the downstream receiver logic. In SD mode, RXRECLK runs at the same frequency as the REFCLK input to the RocketIO transceiver. The DRU generates a clock enable output that causes the downstream receiver logic, clocked by RXRECLK, to run at the 27 MHz SD word rate.

This eliminates the need for a global 270 MHz reference clock, and also eliminates the need for the BUFGMUX used in the version 1 reference design to multiplex the RXRECLK from the RocketIO with the synthesized 27 MHz recovered clock from the DRU.

RocketIO Transceiver

The RocketIO transceiver primitive is included in the hdsdi_rio_refclk module. This module is a wrapper around the RocketIO GT_CUSTOM primitive. The module includes
the bit swap functions on the transceiver’s TXDATA input port and RXDATA output port. The bit swappers reverse the order of the input and output data vectors, compensating for the fact that the RocketIO transceiver sends and receives the MSB first, while SDI requires that the LSB be sent and received first.

SD-SDI DRU

At the current time, the source code for the DRU is not being provided. Instead, precompiled .ngc files for the DRU are provided in the reference design.

There are four .ngc files for the version 1 DRU, one for each oversampling rate from 8X to 10X. The file names are oversample_DRU_nX, where n is replaced by the oversampling rate.

There are 52 .ngc files for the version 2 DRU, 13 for each of the four supported oversampling rates. For each oversampling rate, there is a separate file for each rdclk division factor. The files are named oversample_DRU_nX_clkdv_m, where n represents the oversampling rate and m represents the clock divider used to produce the rdclkDvEn clock enable output. For example, the file oversample_DRU_11X_clkdv5_5 is a DRU supporting 11X oversampling and a clock divider of 5.5. Table 13-1 shows all the various version 2 DRU files and indicates the RocketIO reference clock frequencies and DRU rdclk frequencies required for each when running at the standard 270 Mb/s SD-SDI bit rate.

<table>
<thead>
<tr>
<th>Oversample Rate</th>
<th>Clock Divider</th>
<th>Filename</th>
<th>RocketIO REFCLK</th>
<th>DRU RDCLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>8X</td>
<td>4</td>
<td>oversample_DRU_8X_clkdv4</td>
<td>108.0 MHz</td>
<td>108.0 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>4.5</td>
<td>oversample_DRU_8X_clkdv4_5</td>
<td>108.0 MHz</td>
<td>121.5 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>5</td>
<td>oversample_DRU_8X_clkdv5</td>
<td>108.0 MHz</td>
<td>135.0 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>5.5</td>
<td>oversample_DRU_8X_clkdv5_5</td>
<td>108.0 MHz</td>
<td>148.5 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>6</td>
<td>oversample_DRU_8X_clkdv6</td>
<td>108.0 MHz</td>
<td>162.0 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>6.5</td>
<td>oversample_DRU_8X_clkdv6_5</td>
<td>108.0 MHz</td>
<td>175.5 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>7</td>
<td>oversample_DRU_8X_clkdv7</td>
<td>108.0 MHz</td>
<td>189.0 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>7.5</td>
<td>oversample_DRU_8X_clkdv7_5</td>
<td>108.0 MHz</td>
<td>202.5 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>8</td>
<td>oversample_DRU_8X_clkdv8</td>
<td>108.0 MHz</td>
<td>216.0 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>8.5</td>
<td>oversample_DRU_8X_clkdv8_5</td>
<td>108.0 MHz</td>
<td>229.5 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>9</td>
<td>oversample_DRU_8X_clkdv9</td>
<td>108.0 MHz</td>
<td>243.0 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>9.5</td>
<td>oversample_DRU_8X_clkdv9_5</td>
<td>108.0 MHz</td>
<td>256.5 MHz</td>
</tr>
<tr>
<td>8X</td>
<td>10</td>
<td>oversample_DRU_8X_clkdv10</td>
<td>108.0 MHz</td>
<td>270.0 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>4</td>
<td>oversample_DRU_9X_clkdv4</td>
<td>121.5 MHz</td>
<td>108.0 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>4.5</td>
<td>oversample_DRU_9X_clkdv4_5</td>
<td>121.5 MHz</td>
<td>121.5 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>5</td>
<td>oversample_DRU_9X_clkdv5</td>
<td>121.5 MHz</td>
<td>135.0 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>5.5</td>
<td>oversample_DRU_9X_clkdv5_5</td>
<td>121.5 MHz</td>
<td>148.5 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>6</td>
<td>oversample_DRU_9X_clkdv6</td>
<td>121.5 MHz</td>
<td>162.0 MHz</td>
</tr>
<tr>
<td>Oversample Rate</td>
<td>Clock Divider</td>
<td>Filename</td>
<td>RocketIO REFCLK</td>
<td>DRU RDCLK</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------</td>
<td>------------------------------</td>
<td>-----------------</td>
<td>-----------</td>
</tr>
<tr>
<td>9X</td>
<td>6.5</td>
<td>oversample_DRU_9X_clkdv6_5</td>
<td>121.5 MHz</td>
<td>175.5 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>7</td>
<td>oversample_DRU_9X_clkdv7</td>
<td>121.5 MHz</td>
<td>189.0 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>7.5</td>
<td>oversample_DRU_9X_clkdv7_5</td>
<td>121.5 MHz</td>
<td>202.5 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>8</td>
<td>oversample_DRU_9X_clkdv8</td>
<td>121.5 MHz</td>
<td>216.0 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>8.5</td>
<td>oversample_DRU_9X_clkdv8_5</td>
<td>121.5 MHz</td>
<td>229.5 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>9</td>
<td>oversample_DRU_9X_clkdv9</td>
<td>121.5 MHz</td>
<td>243.0 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>9.5</td>
<td>oversample_DRU_9X_clkdv9_5</td>
<td>121.5 MHz</td>
<td>256.5 MHz</td>
</tr>
<tr>
<td>9X</td>
<td>10</td>
<td>oversample_DRU_9X_clkdv10</td>
<td>121.5 MHz</td>
<td>270.0 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>4</td>
<td>oversample_DRU_10X_clkdv4</td>
<td>135.0 MHz</td>
<td>108.0 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>4.5</td>
<td>oversample_DRU_10X_clkdv4_5</td>
<td>135.0 MHz</td>
<td>121.5 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>5</td>
<td>oversample_DRU_10X_clkdv5</td>
<td>135.0 MHz</td>
<td>135.0 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>5.5</td>
<td>oversample_DRU_10X_clkdv5_5</td>
<td>135.0 MHz</td>
<td>148.5 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>6</td>
<td>oversample_DRU_10X_clkdv6</td>
<td>135.0 MHz</td>
<td>162.0 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>6.5</td>
<td>oversample_DRU_10X_clkdv6_5</td>
<td>135.0 MHz</td>
<td>175.5 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>7</td>
<td>oversample_DRU_10X_clkdv7</td>
<td>135.0 MHz</td>
<td>189.0 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>7.5</td>
<td>oversample_DRU_10X_clkdv7_5</td>
<td>135.0 MHz</td>
<td>202.5 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>8</td>
<td>oversample_DRU_10X_clkdv8</td>
<td>135.0 MHz</td>
<td>216.0 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>8.5</td>
<td>oversample_DRU_10X_clkdv8_5</td>
<td>135.0 MHz</td>
<td>229.5 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>9</td>
<td>oversample_DRU_10X_clkdv9</td>
<td>135.0 MHz</td>
<td>243.0 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>9.5</td>
<td>oversample_DRU_10X_clkdv9_5</td>
<td>135.0 MHz</td>
<td>256.5 MHz</td>
</tr>
<tr>
<td>10X</td>
<td>10</td>
<td>oversample_DRU_10X_clkdv10</td>
<td>135.0 MHz</td>
<td>270.0 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>4</td>
<td>oversample_DRU_11X_clkdv4</td>
<td>148.5 MHz</td>
<td>108.0 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>4.5</td>
<td>oversample_DRU_11X_clkdv4_5</td>
<td>148.5 MHz</td>
<td>121.5 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>5</td>
<td>oversample_DRU_11X_clkdv5</td>
<td>148.5 MHz</td>
<td>135.0 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>5.5</td>
<td>oversample_DRU_11X_clkdv5_5</td>
<td>148.5 MHz</td>
<td>148.5 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>6</td>
<td>oversample_DRU_11X_clkdv6</td>
<td>148.5 MHz</td>
<td>162.0 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>6.5</td>
<td>oversample_DRU_11X_clkdv6_5</td>
<td>148.5 MHz</td>
<td>175.5 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>7</td>
<td>oversample_DRU_11X_clkdv7</td>
<td>148.5 MHz</td>
<td>189.0 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>7.5</td>
<td>oversample_DRU_11X_clkdv7_5</td>
<td>148.5 MHz</td>
<td>202.5 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>8</td>
<td>oversample_DRU_11X_clkdv8</td>
<td>148.5 MHz</td>
<td>216.0 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>8.5</td>
<td>oversample_DRU_11X_clkdv8_5</td>
<td>148.5 MHz</td>
<td>229.5 MHz</td>
</tr>
<tr>
<td>11X</td>
<td>9</td>
<td>oversample_DRU_11X_clkdv9</td>
<td>148.5 MHz</td>
<td>243.0 MHz</td>
</tr>
</tbody>
</table>
The DRU provides a 10-bit recovered data word on its output. This 10-bit vector is multiplexed with the 20-bit RXDATA word from the RocketIO module and connected to the 20-bit data input of the multi_sdi_rx module. The 10-bit word from the SD-SDI DRU must be connected to the 10 most significant bits of the multi_sdi_rx module's 20-bit data input port in SD mode.

**multi_sdi_rx**

The multi_sdi_rx module contains all of the logic needed to descramble and frame the HD or SD video data and to check the video for errors.

The multi_sdi_decoder module descrambles the 20-bit HD data and the 10-bit SD data. The output of the decoder module is normally connected to the multi_sdi_framer module, although the decoder module can be bypassed for debugging purposes by asserting the dec_bypass input.

The multi_sdi_framer module implements the framing algorithm for both HD-SDI and SD-SDI. The Y channel output from the framer carries the 10-bit SD video when running in SD-SDI mode.

The framer resynchronizes immediately to a new EAV or SAV position if the frame_en input is asserted. The frame_en input can be used to implement various filtering functions to filter out single or multiple anomalous EAV or SAV sequences. One simple approach is to connect the nsp output of the multi_sdi_framer module to the frame_en input. The nsp output becomes asserted if the framer detects a timing reference signal at a new starting bit position when frame_en is not asserted and it stays asserted until the framer is synchronized to the bitstream. By connecting nsp to frame_en, a single anomalous timing reference does not cause the framer to resynchronize, but two consecutive timing references at a new starting position does cause the framer to resynchronize.

Note that there are two different implementations of the framer module provided in the reference design: multi_sdi_framer and multi_sdi_framer_mult. The latter uses several MULT18X18 multiplier blocks, available in the Virtex-II Pro, to implement the barrel shifter, reducing the amount of FPGA fabric required for the framer design.

In HD-SDI mode, the hdsdi_rx_crc module checks the video from the framer for CRC errors. This module also captures the line numbers embedded in the video stream and provides them on output ports. This module is fully described in Chapter 10.

The hdsdi_autodetect_ln module detects the current format of the HD video and provides a 4-bit code indicating which video format is being received. Refer to Chapter 10 for a detailed description of this module.

In SD-SDI mode, the EDH processor from Chapter 6 is used to detect the SD video format and to check the video stream for EDH errors. Refer to Chapter 6 for a detailed description of this module. For simplicity, many of the inputs and outputs of the EDH processor are not used in this design. All of the various function of the EDH processor are available by simply connecting the appropriate inputs and outputs.
A set of MUXes selects the appropriate HD or SD signals to drive the outputs of the multi_sdi_rx module. The various outputs of the framer module, such as y, trs, xyz, eav, and sav, are valid for both SD-SDI and HD-SDI, but in SD-SDI mode these MUXes select outputs from the EDH processor instead of from the framer. This is because the EDH processor delays the video stream (and modifies the video stream to correct EDH and TRS errors). In order to have the various timing signals, such as eav and sav, line up with the delayed video from the EDH processor, these signals are taken from the EDH processor rather than the framer in SD-SDI mode.

The multi_sdi_rx_autorate module is responsible for selecting the correct reference clock for the RocketIO transceiver to match the incoming bitstream frequency. It generates the hd_sd and hd_rate signals that, at the top level of the design, select between the various reference clocks for the RocketIO transceiver. This module cycles through the two HD-SDI bit rates and the 270 Mb/s SDI bit rate until valid data is detected coming from the framer module.

**Figure 13-13: multi_sdi_rx Block Diagram**

**Results**

The results shown in Table 13-2 were obtained using XST running under ISE 8.1. Area optimization was used. All designs were able to meet the necessary timing constraints using a Virtex-II Pro -5 speed grade device both HD-SDI bit rates and for 270 Mb/s.
SD-SDI. In order to support 360 Mb/s SD-SDI, the minimum speed grade required is -6, because 8X over-sampling requires RocketIO transceivers capable of running at 2.88 Gb/s.

Conclusions

This chapter describes a technique that can be used to allow SD-SDI bitstreams to be received using the RocketIO multi-gigabit transceivers in the Virtex-II Pro FPGA family. By combining this technique with the HD-SDI receiver design described in Chapter 10, a multi-rate SDI receiver can be implemented that supports both SD-SDI and HD-SDI.

All Virtex-II Pro devices contain multiple RocketIO transceivers, making it possible to implement multiple multi-rate SDI receivers in a single Virtex-II Pro device. For applications that require multiple SDI receivers, this can provide a high level of integration, saving board space, power, and reduce cost as compared to discrete multi-rate SDI receiver implementations.

Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file xapp514_hdsd-rx-mgt.zip.

Table 13-2: Reference Design Implementation Sizes

<table>
<thead>
<tr>
<th>Reference Design</th>
<th>FFs</th>
<th>LUTs</th>
<th>MULT18X18s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-rate SDI Rx for SDV demo board using multi_sdi_framer module</td>
<td>954</td>
<td>1750</td>
<td>0</td>
</tr>
<tr>
<td>Multi-rate SDI Rx for SDV demo board using multi_sdi_framer_mult module</td>
<td>934</td>
<td>1508</td>
<td>6</td>
</tr>
</tbody>
</table>
Appendix A: Reference Design Details

This appendix provides more details about the multi_sdi_decoder, multi_sdi_framer, and multi_sdi_rx_autorate modules.

**multi_sdi_decoder**

This module (see Figure 13-14) implements the SDI decoder algorithm for both SD-SDI and HD-SDI. It is based on the hdsdi_decoder module from Chapter 10. In fact, the only differences are MUXes (labeled Bit Selection) preceding the NRZI-to-NRZ converter and descrambler sections that select a different set of bits for use in SD-SDI mode.

The SD-SDI input data is only 10 bits wide and is required to be placed on the most-significant 10 bits of the d input port. The MUXes select the proper set of bits to perform the NRZI-to-NRZ and descrambling functions on a 10-bit vector instead of the 20-bit wide HD-SDI vector.

**multi_sdi_framer**

This module (see Figure 13-15, page 279) implements the SDI framer function for both SD-SDI and HD-SDI. It is based on the hdsdi_framer module from Chapter 10. An SD TRS detector has been added so that the framer can detect both HD and SD TRS sequences.

An alternate version of the framer, called multi_sdi_framer_mult, uses six MULT18X18 blocks to implement the barrel shifter, providing some savings in the amount of FPGA fabric consumed by the framer module.

Refer to Chapter 10 for more complete descriptions of the framer modules.

**multi_sdi_rx_autorate**

This module selects the correct bit rate for the multi-rate SDI receiver, cycling through the possible reference clock sources to the RocketIO transceiver until the receiver locks to the incoming bitstream. This module is similar to the hdsdi_autorate_rx module from Chapter 10. The Chapter 10 design simply switches between the two HD-SDI bit rates while the design used here also includes the 270 Mb/s SD-SDI bit rate.
The module relies on missing or erroneous SAVs and HD-SDI CRC errors to determine when the receiver is locked to the incoming bitstream. As described in Chapter 10, SAV errors alone are not sufficient for distinguishing whether the correct HD-SDI reference clock is selected. However, SAV errors are sufficient for determining if the correct selection is made between HD-SDI and SD-SDI.

The module consists of a state machine plus an error counter that tracks the number of consecutive video lines with erroneous or missing SAVs or CRC errors (HD-SDI only). When the maximum error threshold is exceeded, the state machine switches to the next reference clock in the cycle and tries again. The design allows the use of different maximum error thresholds depending on whether the receiver is already locked to the bitstream or whether it is unlocked and seeking the correct reference clock. This allows a bigger error threshold to be applied in the locked case to prevent accidentally unlocking in the presence of a burst of noise. It also allows a smaller error threshold to be applied to the unlocked case to make the search process quicker. The error thresholds are defined by the MAX_ERRS_LOCKED and MAX_ERRS_UNLOCKED parameters in the module, making it easy to modify these error thresholds. When modifying these parameters, make sure that the width of the error counter, defined by the ERRCNT_WIDTH parameter, is sufficient to handle the largest threshold value.

Figure 13-16, page 280 is the state diagram for the finite state machine in the multi_sdi_rx_autorate module.
sav_ok is asserted when an SAV is detected and its XYZ protection bits indicate no error.

trs_tc is asserted when a TRS timeout occurs - an SAV is not detected within a given timespan.

Figure 13-16: multi_sdi_rx_autorate State Diagram
Chapter 14

Multi-Rate SDI Integration Examples for the Serial Digital Video Demonstration Board

Summary

The high-definition serial digital interface (HD-SDI) and standard-definition serial digital interface (SD-SDI) standards are used to transport digital video serially over video coax cable. These two standards are used to connect video equipment in broadcast studios and video production centers. Increasingly, video equipment in the broadcast studio is required to support both high-definition (HD) and standard-definition (SD) video streams. SDI interfaces that can support both HD and SD can be implemented using Xilinx Virtex™-II Pro FPGA devices.

demonstrate how to implement the multi-rate SDI transmitter and receiver functions using the RocketIO™ multi-gigabit transceivers in Virtex-II Pro FPGAs. This chapter presents three application examples showing how to use the various SDI transmitter and receiver blocks to form complete multi-rate SDI interfaces. These demonstration applications are designed for the Xilinx Serial Digital Video (SDV) demonstration board [Ref 1]. They also serve as useful examples for any application incorporating multi-rate SDI interfaces.

Introduction

The HD-SDI standard is defined by the SMPTE 292M document [Ref 2]. SD-SDI is defined by both SMPTE 259M and ITU-R BT.656 documents [Ref 3]. HD-SDI and SD-SDI have similar electrical specifications. At the physical layer, they differ primarily in bit rate and in signal rise and fall time requirements at the transmitter output. HD-SDI has two bit rates: 1.485 Gb/s and 1.485/1.001 Gb/s (~1.4835 Gb/s). SD-SDI has different bit rates, but the most common is 270 Mb/s.

Due to the commonality of the physical layer specifications, it is possible to implement receivers and transmitters that can support both HD-SDI and SD-SDI. The receiver must be able to recover the data at both HD-SDI and SD-SDI bit rates. The transmitter must also support both bit rates, and it must have a cable driver with selectable slew rates to meet the differing rise and fall times of these two standards.

This chapter describes three multi-rate application examples. They are all similar in function, but vary slightly in the implementation details. All three applications are pass-through multi-rate SDI interfaces. These pass-through interfaces receive the SDI data, decode it, check it for errors, then re-encode and retransmit the data. In all three application examples, the RocketIO transceivers in the Virtex-II Pro FPGA device family
are used to implement the SDI interfaces. The three applications, described briefly below, are discussed in detail in the remaining sections:

- **Application 1:** This application has one multi-rate SDI receiver and one multi-rate SDI transmitter. The data from the receiver is decoded and checked for errors. In SD-SDI mode, the EDH processor design from Chapter 6 is used to check for errors. In HD-SDI mode, the CRC values from each video line are used to check for errors. This application uses the 74.1758 MHz VCXO on the SDV demonstration board for jitter reduction of the received HD-SDI signal and the 27 MHz VCXO for jitter reduction of the SD-SDI signal. Because the SDV demonstration board does not have a 74.25 MHz VCXO, the multi-rate receiver and transmitter in this application do not support the 1.485 Gb/s HD-SDI bit rate.

- **Application 2:** This application is the same as the first application, except that a PicoBlaze™ based EDH processor from Chapter 7 is used for error checking of the SD-SDI signal. This EDH processor design also implements the video format detection function for both HD-SDI and SD-SDI. Like Application 1, this application does not support the 1.485 Gb/s HD-SDI bit rate due to lack of a 74.25 MHz VCXO on the SDV demonstration board.

- **Application 3:** This application is similar to Application 2 in that it uses the Chapter 7 PicoBlaze EDH processor. The difference is that this application uses the ICS664-01 frequency synthesizer on the SDV demonstration board for jitter reduction of both HD-SDI bit rates [Ref 4]. This synthesizer allows the application to support both HD-SDI bit rates of 1.485 Gb/s and 1.4835 Gb/s as well as the 270 Mb/s SD-SDI bit rate. Not all SDV demonstration boards were built with the ICS664-01 device. Many were built with the older ICS660 device. This application example only works on SDV boards that have the ICS664-01. The ICS660 on the SDV board can be easily replaced by an ICS664-01 because these two parts are pin compatible.
Application Example 1 (EDH Processor from Chapter 6)

Figure 14-1 is the block diagram of the first application example using the Chapter 6 EDH processor.

Figure 14-1: Reference Design Top-Level Block Diagram

HD-SDI Mode

In HD-SDI mode, the RocketIO transceiver in the receiver section uses a 74.1758 MHz reference clock from a crystal oscillator, which provides the correct frequency to support reception of HD-SDI at 1.4835 Gb/s. The RocketIO transceiver recovers the clock and data from the HD-SDI bitstream. The recovered clock from the RocketIO transceiver is buffered by a BUFG and drives the multi-rate receiver module. This multi-rate receiver module, shown in Figure 14-2, decodes and frames the HD-SDI data and then checks the data for CRC errors. Detected errors are indicated by a flashing LED on the SDV board. Other LEDs indicate the video format detected by the receiver.

The recovered data from the receiver section is re-encoded by the HD-SDI encoder module and is sent to another RocketIO transceiver to be serialized and transmitted. The recovered...
clock from the receiver RocketIO transceiver is buffered by a BUFGMUX and drives the HD-SDI encoder module and the input ports of the RocketIO transceiver. The recovered clock from the RocketIO transceiver feeds into a PLL where the jitter present on the clock is reduced to provide a clean reference clock for the RocketIO transceiver in the transmitter section. The jitter reduction PLL is made from a phase detector, implemented in the FPGA, plus an external loop filter and 74.1758 MHz VCXO.

SD-SDI Mode

In SD-SDI mode, the receiver’s RocketIO transceiver is provided with a 108 MHz reference clock from the DCM, causing the transceiver to sample the SD-SDI bitstream at 2.16 GHz. This rate is eight times faster than the 270 Mb/s bit rate of SD-SDI. The oversampled data from the RocketIO transceiver is processed by a data recovery unit (DRU). The DRU recovers the SD-SDI data from the oversampled data captured by the RocketIO transceiver. The DRU is driven by the recovered clock from the RocketIO transceiver. The DRU asserts a clock enable output signal whenever the DRU has 10 bits of data available on its output port. This clock enable, combined with the recovered clock from the RocketIO transceiver, controls the data flow through the multi-rate receiver in SD-SDI mode. As noted above, the same recovered clock is used by multi-rate receiver block in HD-SDI mode. In HD-SDI mode, the recovered clock runs at 74.1578 MHz, and the clock enable from the DRU is ignored. In SD-SDI mode, the recovered clock runs at 108 MHz, and the clock enable from the DRU is usually asserted for only one out of every four clock cycles, resulting in a 27 MHz data rate through the multi-rate receiver module.

A frequency-locked loop (FLL) reduces the jitter on the recovered clock in SD-SDI mode. The decoded and framed SD data from the multi-rate receiver module is written into the jitter reduction module. This module contains an asynchronous FIFO. Data is written into the FIFO with the recovered clock. Data is read from the FIFO using a clock from a 27 MHz VCXO. The jitter reduction module controls the 27 MHz VCXO to keep the FIFO about half full, causing the VCXO to run at the same frequency as the video stream recovered by the receiver. The output of the VCXO is multiplied by two by an external PLL. The RocketIO transceiver in the transmitter requires a 54 MHz reference clock that is frequency-locked to the received video stream. The rest of the transmitter section, including the read side of the FIFO, is enabled every other cycle of the 54 MHz clock by a clock enable signal. A DCM also can be used to double the frequency of the VCXO, but the PLL produces less jitter than the DCM.

Ten bits of encoded data are produced by the SD-SDI encoder on every other cycle of the 54 MHz FLL clock. As described in Chapter 12, “Multi-Rate HD/SD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers,” the RocketIO transceiver’s transmitter cannot run at 270 Mb/s. Instead, the transmitter multiplies the 54 MHz reference clock from the FLL by 20 to produce a 1.08 Gb/s bitstream. This rate is four times faster than the 270 Mb/s SD-SDI bit rate. Each of the 10 encoded bits from the SD-SDI encoder must be sent four times consecutively to produce a 270 Mb/s SD-SDI bitstream from the RocketIO transceiver. The bit replication logic produces a 40-bit vector from the 10 encoded bits. These 40 bits are loaded into the RocketIO transmitter, 20 bits at a time, on consecutive cycles of the 54 MHz clock.
Below are brief descriptions of the clocks used in this application:

- **clk_74_17M**
  This clock comes from the 74.1758 MHz XO on the SDV demonstration board and provides the reference frequency needed to operate the RocketIO transceiver in the receiver section at the 1.4835 Gb/s HD-SDI bit rate.

- **mgt_rxrecclk**
  This signal is the recovered clock from the RocketIO receiver. In HD mode, the frequency of this clock is 74.1758 MHz. In SD mode, the frequency is 108 MHz. This clock is buffered by a BUFG to produce gclk_rxrecclk. It also connects to a BUFGMUX where it drives the tx_gclk in HD-SDI mode.

- **gclk_rxrecclk**
  This clock is mgt_rxrecclk after the BUFG. It drives all of the receiver logic at 74.1578 MHz in HD mode and 108 MHz in SD mode. In SD mode, the dru_rdclken signal from the DRU only allows the multi-rate SDI receiver logic to clock when the DRU has 10 bits of recovered data available, effectively forcing the data rate to 27 MHz.

*Figure 14-2: Multi-Rate Receiver Module Block Diagram*
• clk_ics8745
This signal is the 54 MHz clock from the FLL. It drives a reference clock input of the transmitter’s RocketIO transceiver. It also connects to the same BUFGMUX as mgt_rxrecclk. The BUFGMUX selects clk_ics8745 to drive the transmitter section in SD mode.

• tx_gclk
This signal is the global clock driving the transmitter section. It is driven by mgt_rxrecclk (74.1758 MHz) in HD mode and clk_ics8745 (54 MHz) in SD mode. In SD mode, a clock enable is asserted every other clock cycle to cause the transmitter section to run at 27 MHz.

• clk_hd_vcxo
This signal is the clock from the 74.1758 MHz VCXO. This VCXO is used to implement a PLL to reduce the jitter on the gclk_rxrecclk. The output of the VCXO is phase-locked to gclk_rxrecclk, but has low jitter so that it can be used as a reference clock to the transmitter’s RocketIO transceiver.

Figure 14-3 is a portion of the SDV demonstration board schematic showing the 74.1758 MHz VCXO and loop filter. Note that the VCXO has 3.3V LVPECL outputs. These are AC-coupled to a 2.5V LVDS input buffer on the FPGA.

![Figure 14-3: 74.1758 MHz VCXO and Loop Filter](Image)

Figure 14-4 is a portion of the SDV demonstration board schematic showing the 27 MHz VCXO, loop filter, and ICS8745 PLL. In new designs, it is recommended that a 54 MHz VCXO be used to eliminate the need to multiply the output of the VCXO by two. On the SDV board, the output of the 27 MHz VCXO is not directly connected to the ICS8745 PLL. Instead, the signal from the VCXO is connected through the FPGA to the ICS8745.

Figure 14-4 also shows how the ICS664-01 is connected between the 27 MHz VCXO and the FPGA. The ICS664-01 is not used in this application, but is used in Application 3.
Table 14-1 shows the FPGA resources used by Application Example 1. These results were obtained with ISE 6.3i using XST with the Verilog version of the design. XST optimized the design for area. The design meets all timing constraints in a -5 speed grade XC2VP4 device. See the notes in Chapter 13 regarding using -5 speed grade Virtex-II Pro devices for 8X oversampling of SD-SDI bitstreams.

Table 14-1: FPGA Resources Used by Application Example 1

<table>
<thead>
<tr>
<th>FF</th>
<th>LUT</th>
<th>Block RAM</th>
<th>MULTI18X18</th>
</tr>
</thead>
<tbody>
<tr>
<td>1323</td>
<td>1907</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

**Figure 14-4:** 27 MHz VCXO, Loop Filter, ICS8745 PLL, and ICS664-01
Application Example 2 (EDH Processor from Chapter 7)

Figure 14-5 is a block diagram of the second application example using the PicoBlaze EDH processor from Chapter 7. The chief advantage of the PicoBlaze based EDH processor is that it is considerably smaller than the EDH processor from Chapter 6.

At the top level, there are only two differences between Application 1 and Application 2. The first difference is that a global 54 MHz clock is generated by the DCM and is connected to the cpuclk port of the multi-rate receiver module, providing the clock for the PicoBlaze processor.

The second difference is that the 74.1758 MHz oscillator used in Application 1 to provide a reference clock to the RocketIO transceiver in the receiver section is eliminated. The 74.1758 MHz VCXO that is a part of the HD jitter reduction PLL provides the reference clock for the receiver’s RocketIO transceiver in this application. “Appendix A: Reference Design Details” in Chapter 10 describes how to use the output of the jitter reduction PLL to drive the reference clock inputs of the RocketIO transceivers in both the transmitter and receiver sections. Initially, the signal from the VCXO is approximately equal to the
bitstream frequency, regardless of the control voltage applied to the VCXO because the VCXO has a limited pull range (frequency variation). The frequency of the VCXO is close enough to the bitstream frequency to allow the RocketIO transceiver in the receiver section to lock to the bitstream and produce a recovered clock. The PLL locks to the recovered clock from the RocketIO transceiver and, from then on, the signal from the VCXO tracks the frequency input bitstream.

Figure 14-6 shows the block diagram of the multi-rate receiver module for Application 2. The Chapter 6 EDH processor, used in Application 1, has been replaced by the PicoBlaze EDH processor from Chapter 7. Application 1 uses an hdsdi_autodetect_in module for two purposes. First, it detects and reports the format of the HD video. Second, it locks to the HD video stream and generates expected line numbers that are compared against the actual line numbers from the video stream. The PicoBlaze EDH processor can implement both of these HD functions. So, the hdsdi_autodetect_in module is not required.

**Clocks**

The clocks for Application 2 are the same as for Application 1 except that clk_74_17M is gone (the HD reference clock to the receiver’s RocketIO transceiver is now clk_hd_vcxo). Also, this application has a global 54 MHz clock from the DCM called gclk_54M used as the PicoBlaze processor clock.
Design Size

Table 14-2 shows the FPGA resources used by this application. These results were obtained with ISE 6.3i using XST with the Verilog version of the design. XST optimized the design for area. The design meets all timing constraints in a -5 speed grade XC2VP4 device. Due to the size reduction afforded by the PicoBlaze EDH processor, this application uses about 12% fewer flip-flops and 23% fewer LUTs than Application 1.

Table 14-2: FPGA Resources Used by Application Example 2

<table>
<thead>
<tr>
<th>FF</th>
<th>LUT</th>
<th>BRAM</th>
<th>MULT18X18</th>
</tr>
</thead>
<tbody>
<tr>
<td>1163</td>
<td>1461</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>
Figure 14-7 is a block diagram of the third application example using the ICS664 frequency synthesizer device. Use of the ICS664 on the SDV demonstration board allows this application to support for both HD-SDI bit rates plus the 270 Mb/s SD-SDI bit rate.

This application is similar to Application 2, but its clocking is modified to use the ICS664 frequency synthesizer. Notice that the HD jitter reduction PLL, using the 74.1758 MHz VCXO, is removed from the design. Instead, the HD jitter reduction PLL is made from the 27 MHz VCXO and its associated loop filter plus the ICS664. The ICS664 uses the 27 MHz signal from the VCXO as a reference to generate either 74.25 MHz or 74.1758 MHz. The combination of the ICS664, 27 MHz VCXO, phase detector, and loop filter form a PLL running at either 74.25 MHz or 74.1758 MHz.
The 27 MHz VCXO is now at the heart of both the HD PLL and the SD FLL. Because the SDI interface never can be in both SD and HD mode at the same time, there are no conflicts when sharing the 27 MHz VCXO in this way. A MUX is used to drive the control signals to the loop filter, giving control of the VCXO to the SD jitter reduction module in SD mode and to the HD phase detector in HD mode. The clock paths involving the HD PLL and the SD FLL are coded as shown in the key in Figure 14-7. The key defines the HD PLL clock path, the SD PLL clock path, and components and paths used for both HD and SD, such as the 27 MHz VCXO.

As in Application 2, the HD jitter reduction PLL also serves as the HD reference clock for the receiver’s RocketIO transceiver. This allows this application to take advantage of the ICS664 in the jitter reduction PLL to generate the HD reference clock frequencies for both the transmitter and receiver sections.

Figure 14-8 is the block diagram of the multi-rate receiver module for this application. It is almost identical to the multi-rate receiver section from Application 2, but the automatic rate detection module must now select between three possible bit rates, 1.485 Gb/s, 1.8435 Gb/s, and 270 Mb/s. In addition to the hd_sd signal, the rate detection module must also produce an hd_rate signal to distinguish between the two HD-SDI bit rates. This hd_rate signal selects the frequency generated by the ICS664.

**Figure 14-8:** Multi-Rate Receiver Module for Application 3
The ICS664 is available in several varieties. The ICS664-01 has a singled-ended output and is used on the SDV demonstration board because it is pin compatible with the ICS660 that was originally designed into the SDV board. The ICS664-02 has a differential output and produces lower jitter. Xilinx recommends that new designs use the ICS664-02 to provide a lower jitter reference to the RocketIO transceiver.

Clocks

The clocks for this application are the same as for Application 2 except that clk_hd_vcxo is replaced by clk_ics664 from the ICS664-01 frequency synthesizer.

Design Size

Table 14-3 shows the FPGA resources used by this application. These results were obtained with ISE 6.3i using XST with the Verilog version of the design. XST optimized the design for area. The design meets all timing constraints in a -5 speed grade XC2VP4 device.

Table 14-3: FPGA Resources Used by Application Example 3

<table>
<thead>
<tr>
<th>FF</th>
<th>LUT</th>
<th>BRAM</th>
<th>MULT18X18</th>
</tr>
</thead>
<tbody>
<tr>
<td>1162</td>
<td>1451</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Conclusion

This chapter presents three different multi-rate SDI integration examples using the basic multi-rate SDI transmitter and receiver building blocks from Chapter 9, Chapter 10, Chapter 12, and Chapter 13. The chapter shows how to combine these blocks to build complete multi-rate SDI interfaces in various configurations.

The flexibility of the Virtex-II Pro FPGA family, including the multi-gigabit RocketIO transceivers, allows customers to easily create customized multi-rate SDI interfaces. These interfaces use a relatively small portion of the resources in the FPGA, allowing other video processing functions to be implemented in the same FPGA.

Design Files

The reference design files are available on the Xilinx website at:

www.xilinx.com/bvdocs/appnotes/xapp514.zip

Open the ZIP archive and extract file xapp514_hdsd-integ-demobrd.zip.
Section IV: DVB-ASI

Audio/Video Connectivity Solutions for the Broadcast Industry
DVB-ASI Physical Layer Implementation

Introduction to DVB-ASI

The DVB Project is an industry-led consortium of over 260 broadcasters, manufacturers, network operators, software developers, regulatory bodies, and others in over 35 countries committed to designing global standards for the delivery of digital television and data services.

DVB-ASI is a serial video communications standard defined by the DVB consortium for use in transporting MPEG-2 encoded video streams. The standard is most commonly used to connect cable head-end equipment that transports MPEG-2 streams. Data is transported at a rate of 270 Mb/s.

The DVB-ASI protocol stack is illustrated in Figure 15-1.

![Figure 15-1: DVB-ASI Protocol Stack](image-url)

The payload of the DVB-ASI protocol is the MPEG-2 packet. MPEG-2 packets are made up of 188 bytes in a standard transmission, and 204 bytes if the packets are Reed-Solomon encoded. Figure 15-2 illustrates the data path of an MPEG-2 packet as it traverses through the DVB-ASI transmitter and receiver.

The section entitled “Reference Design” describes implementation of the physical layer of the protocol. The physical layer employs 8B/10B encoding, which provides DC balancing of the link and a convenient error-checking mechanism. A complete discussion of DVB-ASI specifications is provided in “EN 50083-9: Cabled Distribution Systems for Television, Sound and Interactive Multimedia Signals,” available from the DVB Consortium.
Chapter 15: DVB-ASI Physical Layer Implementation

Figure 15-2: Data Path of DVB-ASI Transmitter and Receiver
Implementing the DVB-ASI Physical Layer with SelectIO Features

Introduction

This section describes the DVB-ASI physical layer as implemented with the SelectIO™ features available in all Xilinx FPGA families. The reference design is targeted towards the Virtex™-II and Virtex-II Pro families. The receiver and transmitter implementations are discussed in the first two sections. The final section discusses the receiver and transmitter configured in pass-through mode. The reference design provided with this section is the pass-through design. In the design, the user can select whether to implement the design in BIST mode (where the receiver and transmitter are separate) or pass-through mode. For more details on the SelectIO features available in Virtex-II and Virtex-II Pro devices, please refer to the respective user guides available on www.xilinx.com.

SelectIO DVB-ASI Receiver

Figure 15-3 shows the DVB-ASI receiver block diagram.

Figure 15-3: DVB-ASI Receiver Block Diagram

The DVB-ASI receiver performs two major tasks:

- Frame data in the correct word boundary
- Decode the 8B/10B encoded word

The receiver is comprised of the following modules:

- Data Recovery Module and Serial-to-Parallel Converter
- 8B/10B Decoder
- Sync Byte Insertion / Deletion Unit
- Elastic FIFO
Reference Design Details

Table 15-1 shows the DVB-ASI receiver reference design implementation details.

<table>
<thead>
<tr>
<th>Design Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Used in Implementation</td>
<td>Virtex-II Pro XC2VP4</td>
</tr>
<tr>
<td>Family Targeted</td>
<td>Virtex-II and Virtex-II Pro</td>
</tr>
<tr>
<td>Board Implemented</td>
<td>Cook Technologies SDV Demo Board</td>
</tr>
<tr>
<td>Resource Utilization</td>
<td>• 200 slices</td>
</tr>
<tr>
<td></td>
<td>• 2 Block RAMs</td>
</tr>
<tr>
<td></td>
<td>• 2 BUFGs</td>
</tr>
<tr>
<td></td>
<td>• 1 DCM</td>
</tr>
<tr>
<td>HDL Used</td>
<td>VHDL and Verilog</td>
</tr>
<tr>
<td>Synthesis Tool</td>
<td>Synplify 7.6.1</td>
</tr>
<tr>
<td>Implementation Tool</td>
<td>Xilinx ISE 6.3</td>
</tr>
</tbody>
</table>

Notes:
1. Synthesis of the reference design has been verified only with Synplify.

Cable Equalization

The receiver specification is described in the DVB-ASI protocol specification “EN 50083-9: Cabled Distribution Systems for Television, Sound and Interactive Multimedia Signals.” The specifications call for transformer coupling between the receiver and the connector. Figure 15-4 illustrates the DVB-ASI as implemented on the Cook Technologies Serial Digital Video (SDV) demonstration board.

Figure 15-4: ASI Receiver

The DVB-ASI protocol does not specify a cable length requirement. However, most users adhere to the same standards as SD-SDI when implementing their DVB-ASI design. SD-SDI, with its slower bit rates, allows maximum cable lengths of up to 300 meters. The coax cable causes frequency-dependent attenuation of the signal, where the higher frequency components of the signal are attenuated more than the lower frequency components. The coax cable also causes frequency-dependent phase distortion, where the higher frequency components are phase shifted more than the lower frequency components. After passing through long coax cables, the signal is severely distorted and attenuated. The receiver must compensate for this attenuation and distortion before attempting to recover the signal. Cable length equalization is used to compensate for the attenuation and distortion introduced by the coax cable.
Typically, an adaptive cable length equalizer is used in SDI receivers. Such an equalizer actively monitors the amount of attenuation and distortion present on the incoming signal and applies the correct amount of equalization to the signal. The cable length can be changed without requiring a change to the equalizer, as would be the case if fixed length equalization were used. The SDI specifications call for capacitive coupling between the connector and the receiver. Figure 15-5 illustrates the cable equalizer used on the SD-SDI as implemented on the Cook Technologies Serial Digital Video (SDV) demonstration board.

For more details on the receiver specifications for both ASI and SDI, please refer to the SDV Demo Board User Guide available from Cook Technologies (www.cook-tech.com).

Clocking

One digital clock manager (DCM) is used to generate the two required receive clocks, a 135 MHz clock and its inverse. The clocks are routed to global clock buffers for global distribution. Note that only one DCM is required, regardless of the number of DVB-ASI receivers instantiated.
Data Recovery Module

Figure 15-6 shows a block diagram of the data recovery module.

The main purpose of the data recovery module is to recover words from the incoming serial bitstream in the proper word boundary. Module components consist of the following:

- Asynchronous Tap Delay Line
- Data Extraction State Machine
- Serial-to-Parallel Converter

Basic Operation

The DCM generates the system clock, which samples incoming serial data. For the receive clock to capture incoming data correctly, the capture clock must sample data within the data valid window. To avoid sampling outside of the data valid window, the data recovery module continuously monitors its sampling point and determines whether an adjustment is necessary. A data transition edge indicates where the sampling point might be in danger of falling out of the data valid window. Therefore, the data recovery module creates multiple samples of the incoming data each clock period. From the set of samples created, it determines where the transition edges are, and it ultimately positions the sampling point away from the transition edges.

The data recovery module uses a tap delay line constructed from look-up tables (LUTs) that are connected in serial within the FPGA fabric to generate multiple samples of the incoming data stream. Each tap has a combined delay (LUT delay + interconnect delay) of approximately 700 ps.
This is the worst-case delay. The proper timing analysis of the tap delay line requires the calculation of the worst-case delay and the best-case delay. The worst-case timing analysis guarantees that there are at least two samples within the data-valid window. The best-case timing analysis is done using a common derating factor of 40%. This timing analysis guarantees the jitter tolerance of the tap delay line. In this design, the best-case timing analysis allows for peak-to-peak jitter tolerance of up to 0.5 UI. For more details on calculating the worst case and best case timing analysis, please refer to Xilinx Application Note XAPP671 [Ref 1].

The reference design uses two tap delay lines, each constructed from eight LUTs. Each tap delay line captures every other sample from the incoming bitstream, allowing both tap delay lines to run at half of the incoming data rate. One tap delay line is clocked at 135 MHz (clk135p), while the other is clocked at the inverse (clk135n). The data clocked on the inverse clock domain is eventually brought into the clk135p clock domain.

Edge information is created by pairwise XOR-ing each of the eight samples generated by the tap delay lines. A state machine polls the edge information and decides whether or not to adjust the current sampling point. Since this is an asynchronous system, when the incoming data rate is slightly different than the system frequency, occasional adjustments to the sampling rate are necessary. If the incoming data rate is faster than the sampling frequency, then an occasional three bits per 135 MHz clock period (instead of the usual two bits per 135 MHz clock period) is sampled to synchronize the data rate. Conversely, if the incoming data rate is slower than the sampling frequency, then an occasional one bit per 135 MHz clock period is sampled to synchronize the data rate.

Implementation Instructions

Proper operation of the data recovery module depends critically on the placement and routing of key routes. The relative location and routing constraints are predetermined and are built into the reference design. The only action the user must take is to set the proper parameters. Placement of the data recovery module is dependent on placement of the data IOBs. Two sets of placement and routing constraints are used for two placement scenarios:

- Data IOBs are placed in top or bottom I/O banks (Banks 0, 1, 4, and 5).
- Data IOBs are placed in right or left I/O banks (Banks 2, 3, 6, and 7).

All placement constraint parameters are found in the `des.vhd` HDL file. Parameters that determine the placement of the data recovery module are the SIDE parameter and the RLOC_ORIGIN_CONST parameter. The SIDE parameter tells the implementation tool whether to use the constraints for a top-bottom implementation (SIDE = 1) or a left-right implementation (SIDE = 0). The RLOC_ORIGIN_CONST parameter specifies the RLOC_ORIGIN attribute of the relative location macros that constitute the data recovery module.

To ensure optimum operation, the RLOC_ORIGIN parameter must be located as close to the data IOBs as possible. If possible, for right-left implementations the RLOC_ORIGIN parameter must be in the same CLB row as, or one row above or below, the IOB location. Similarly, for top-bottom implementations the RLOC_ORIGIN parameter must be in the same CLB column as, or one column to the right or left of, the IOB location.

In the case where data IOBs are placed in the top or bottom I/O banks, set the SIDE parameter to 1, and set RLOC_ORIGIN_CONST to the appropriate value. Figure 15-7 shows the relative location macro configuration for a top-bottom implementation.
In the case where data IOBs are placed in the right or left I/O banks, set the SIDE parameter to 0, and set RLOC_ORIG_CONST to the appropriate value. Figure 15-8 shows the relative location macro configuration for a left-right implementation.

Checking the routed design in FPGA Editor after place-and-route is complete helps ensure a balanced route from data IOB to the input of the first LUTs on each tap delay line (rising and falling). The difference between both routes must not be more than 500 ps. This can be achieved as follows:

1. Open the routed design in FPGA Editor. The net from IOB to the tap delay line is named sdatain.
2. Search for the sdatain net in the List window (see Figure 15-9) and highlight it.
3. Verify that this is indeed the net that connects the data IOB to the first LUT of the tap delay line. See Figure 15-10.
4. With the `sdatain` net highlighted, click on the **Delay** button. See Figure 15-11.

5. Verify that the delay of the route between the data IOB and the first LUT of both tap delay lines does not differ by more than 500 ps. This timing must be met. If it is not met, change the RLOC_ORIG_CONST parameter until timing is met.

The other parameter the user must set is the SYNC_MODE parameter, also found in the `des.vhd` source file. The SYNC_MODE parameter indicates whether a single sync byte is used to frame the initial data or whether two sync bytes within a 5-byte window are used, as per DVB-ASI specifications. Setting SYNC_MODE to 0 indicates that the receiver frames (locks) the first sync byte it detects. Setting SYNC_MODE to 1 indicates that the receiver frames only when two sync bytes are detected on the same byte boundary within a 5-byte window.
Table 15-2 summarizes user parameters in the design.

### Table 15-2: Table of User Parameters

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLOC.ORIG_CONST</td>
<td>String</td>
<td>Sets location (RLOC_ORIGIN) of the data recovery module.</td>
</tr>
<tr>
<td>SIDE</td>
<td>Bit</td>
<td>Indicates side of the FPGA on which the data recovery module is implemented:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Indicates left-right implementation (Data IOBs located in Banks 2, 3, 6, or 7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Indicates top-bottom implementation (Data IOBs located in Banks 0, 1, 4, or 5)</td>
</tr>
<tr>
<td>SYNC.MODE</td>
<td>Bit</td>
<td>Indicates whether a single sync byte is used to frame the initial data or whether two sync bytes within a 5-byte window are used:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Indicates the receiver frames (locks) the first sync byte it detects.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Indicates the receiver frames only when two sync bytes are detected on the same byte boundary within a 5-byte window.</td>
</tr>
</tbody>
</table>

**Notes:**
1. The location of the data recovery module should be as close to data input pins as possible.

### 8B/10B Decoder

As mentioned in “Introduction to DVB-ASI” the ASI packets are 8B/10B encoded. The 8B/10B code is DC-balanced allowing for cable equalization. Furthermore, 8B/10B transmissions have a limited run length: no more than five consecutive ones or zeroes.

The 8B/10B set of codes provides special comma (K) characters that are useful as packet delimiters. These 10-bit characters are guaranteed not to occur with any input combination. The K28.5 comma character is used as a packet delimiter in the DVB-ASI specifications.

The 8B/10B encoding scheme also provides a convenient error detection scheme, based on the concept of disparity. Disparity is defined as the difference between ones and zeroes in a word. Positive disparity refers to an excess of ones over zeroes. Negative disparity refers to an excess of zeroes over ones. During normal operation, a running disparity is stored, which serves as a disparity record for the aggregate of previously encoded symbols. The disparity of each decoded symbol is added to the running disparity.

Error detection in the decoder is achieved in two ways: code error and disparity error. A code error occurs when a 10-bit encoded symbol does not match any symbol in the code set. A disparity error occurs when the running disparity does not match a certain value. Using both of these error detection methods enables a very robust error detection scheme. In the reference design, code error and disparity error flags are OR-ed together to set the error_condition flag.
The 8B/10B encoding scheme provides all of these benefits with a very low (only 25%) overhead. The DVB-ASI specifications describe the 8B/10B encoding scheme in greater detail.

Core Generator 8B/10B Decoder Instantiation Instructions

Xilinx provides the 8B/10B decoder core free of charge. The core is available from the Core Generator tool. Instantiating the core in a design is achieved as follows:

1. See Figure 15-12. When opening the Core Generator tool for the first time, a project must be selected to which Core Generator writes all its files. Select the ISE project being worked in.

2. In Core Generator, navigate to the **Communications & Networking → Building Blocks** folder where the 8B/10B encoder and decoder cores are stored. Double-click the **Decode 8b/10b** core.
3. A dialog box similar to Figure 15-13 appears. Select the parameters as shown in the figure and click **Generate**. The 8B/10B Decoder core is generated.

![Generating 8B/10B Decoder from Core Generator](Image)
Sync Byte Insertion/Deletion

An elastic FIFO sits in the receive data chain to provide rate-matching between the incoming data rate and the system frequency. The FIFO state machine monitors the FIFO level and, when necessary, adjusts the incoming data stream to keep the FIFO from overflowing or underflowing. Figure 15-14 shows the state machine logic.

Figure 15-14: FIFO Control State Machine

When initialized, the FIFO state machine fills the FIFO to the halfway mark. From then on, the state machine monitors both high-water and low-water mark levels (see Figure 15-15). If the FIFO level reaches the high-water mark (indicating that the incoming data rate is faster than the system/sampling frequency), the state machine deletes one comma character. If the FIFO level reaches the low-water mark (indicating that the incoming data rate is slower than the system/sampling frequency), the state machine adds one comma character.

The FIFO state machine also stuffs the FIFO with comma characters whenever the system loses lock—for example, when a cable is unplugged or a link encounters errors.

User Instructions for Instantiating Core Generator FIFO

The FIFO used in the reference design is a 2047 x 9 asynchronous FIFO. Xilinx provides the Asynchronous FIFO core free of charge. The core is available from the Core Generator tool. Instantiating the core in a design is achieved as follows:

1. See Figure 15-16. When opening the Core Generator tool for the first time, a project must be selected to which Core Generator writes all its files. Select the ISE project being worked in.
Implementing the DVB-ASI Physical Layer with SelectIO Features

Figure 15-15: FIFO Levels

High-Water Mark
FIFO_LEVEL = 11

FIFO Midpoint
FIFO_LEVEL = 10

Low-Water Mark
FIFO_LEVEL = 01

Elastic FIFO

Figure 15-16: Generating FIFO from Core Generator
2. In Core Generator, navigate to the Memories & Storage Elements → FIFOs folder where the FIFO cores are stored. Double-click the Asynchronous FIFO core.

3. A dialog box similar to Figure 15-17 appears. Select the parameters as shown in the figure and click Generate. The 8B/10B Encoder core is generated.

![Figure 15-17: Generating FIFO from Core Generator](x508_14_021805)

Figure 15-17: Generating FIFO from Core Generator
Figure 15-18 shows handshaking options.

![Handshaking Options](image)

**Figure 15-18: Handshaking Options**

SelectIO DVB-ASI Transmitter

Figure 15-19 shows a block diagram for the DVB-ASI transmitter.

![DVB-ASI Transmitter Block Diagram](image)

**Figure 15-19: DVB-ASI Transmitter Block Diagram**

The main purpose of the DVB-ASI transmitter is to encode transmit words in 8B/10B format and serialize the data for transmission.

The transmitter is comprised of the following modules:

- 8B/10B Encoder
- Parallel-to-Serial Converter

Table 15-3 shows the DVB-ASI transmitter reference design implementation details.

### Table 15-3: DVB-ASI Transmitter Implementation Details

<table>
<thead>
<tr>
<th>Design Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Used in Implementation</td>
<td>Virtex-II Pro XC2VP4</td>
</tr>
<tr>
<td>Family Targeted</td>
<td>Virtex-II and Virtex-II Pro</td>
</tr>
<tr>
<td>Board Implemented</td>
<td>Cook Technologies SDV Demo Board</td>
</tr>
</tbody>
</table>
Table 15-3: DVB-ASI Transmitter Implementation Details (Continued)

<table>
<thead>
<tr>
<th>Design Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Utilization</td>
<td>- 50 slices</td>
</tr>
<tr>
<td></td>
<td>- 1 Block RAM</td>
</tr>
<tr>
<td></td>
<td>- 2 BUFGs</td>
</tr>
<tr>
<td></td>
<td>- 1 DCM</td>
</tr>
<tr>
<td>HDL Used</td>
<td>VHDL and Verilog</td>
</tr>
<tr>
<td>Synthesis Tool</td>
<td>XST and Synplify 7.6.1</td>
</tr>
<tr>
<td>Implementation Tool</td>
<td>Xilinx ISE 6.3</td>
</tr>
</tbody>
</table>

Line Driver

The line driver specification is described in the DVB-ASI protocol specification “EN 50083-9: Cabled Distribution Systems for Television, Sound and Interactive Multimedia Signals.” The specification calls for transformer coupling between the line driver and the connector.

Figure 15-20 illustrates the ASI line driver as implemented on the Cook Technologies Serial Digital Video (SDV) demonstration board.

![Figure 15-20: ASI Line Driver](x509_34_040505)

It is common practice to use the SDI line driver specifications to implement the DVB-ASI transmitter. This allows the user to implement a multi-standard SDI / ASI transmitter on a common connector. The SDI specifications call for capacitive coupling between the line driver and the connector. Figure 15-21 illustrates the cable equalizer used on the SD-SDI as implemented on the Cook Technologies Serial Digital Video (SDV) demonstration board.

![Figure 15-21: SDI Line Driver](x509_35_040505)
For more details on the transmitter specifications for both ASI and SDI, please refer to the SDV Demo Board User Guide available from Cook Technologies (www.cook-tech.com).

Clocking

One digital clock manager (DCM) is used to generate the two transmit clocks required: a 27 MHz word rate clock (which is also the system clock) and a 270 MHz serial transmit clock. The clocks are routed to global clock buffers for global distribution. Note that only one DCM is required, regardless of the number of DVB-ASI transmitters instantiated.

8B/10B Encoder

The method of encoding used in the DVB-ASI protocol is 8B/10B encoding. The 8B/10B code is DC-balanced allowing for cable equalization. Furthermore, 8B/10B transmissions have a limited run length; no more than five consecutive ones or zeroes. The 8B/10B set of codes also provides special comma (K) characters that are useful as packet delimiters. These 10-bit characters are guaranteed not to occur with any input combination. One comma character, the K28.5, is used as a packet delimiter in DVB-ASI specification.

The 8B/10B encoding scheme also provides a convenient error detection scheme. The error detection scheme is based on the concept of disparity. Disparity is defined as the difference between 1s and 0s in a word. Positive disparity refers to the excess of 1s over 0s. Negative disparity refers to the excess of 0s over 1s. During normal operation, a running disparity—the record of disparity for the aggregate of previously encoded symbols—is kept. For each decoded symbol, its disparity is added to the running disparity.

Error detection in the decoder is achieved via two means: code error and disparity error. A code error is called when the 10-bit encoded symbol does not match any symbol in the code set. A disparity error is called when the running disparity does not match a certain value. By using both these methods of error detection, a very robust error detection scheme is achieved. In the reference design, the code error and disparity error flags are ORed together to get the error flag error_condition. The 8B/10B encoding scheme provides all these benefits for a very low 25% overhead. The DVB-ASI specification mentioned in the introduction discusses the 8B/10B encoding scheme in greater detail.
User Instructions for Instantiating Core Generator 8B/10B Encoder

The 8B/10B encoder core is provided free of charge from Xilinx. The core is available from the Core Generator tool. Follow the instructions below to instantiate the core in your design.

1. See Figure 15-22. When opening the Core Generator tool for the first time, a project must be selected to which Core Generator writes all its files. Select the ISE project being worked in.

2. In Core Generator, navigate to the Communications & Networking → Building Blocks folder where the 8B/10B encoder and decoder cores are stored. Double-click the Encode 8b/10b core.

Figure 15-22: Generating 8B/10B Encoder from Core Generator
3. Doubleclick the **Encode 8B/10B** core. A dialog box similar to **Figure 15-23** displays. Select parameters as shown, and click **Generate**. The 8B/10B Encoder core is generated.

![Figure 15-23: Generating 8B/10B Encoder from Core Generator](image)
Pass-Through Mode

The DVB-ASI transmitter and receiver can be connected together in pass-through mode. Figure 15-24 illustrates the DVB-ASI pass-through design.

Figure 15-24: DVB-ASI Block Diagram

The design is comprised of the following modules:

- LVDS (low-voltage differential signaling) I/Os
- DVB-ASI Receiver
- DVB-ASI Transmitter
- Elastic FIFO
- BIST (Built-In Self Test) Test Bitstream Checker
- BIST (Built-In Self Test) Test Bitstream Generator
- Receive DCM (Digital Clock Manager)
- Transmitter DCM

Modes of Operation

Two modes of operation are available:

- BIST (Built-In Self Test) mode
- Pass-through mode

In BIST (Built-In Self Test) mode, the receiver and transmitter are separate elements and both modules can be run simultaneously. A BIST test bitstream generator and checker provides BIST capabilities.
When configured in pass-through mode, the receiver and transmitter are linked via an elastic FIFO. The FIFO provides rate-matching capabilities between the receiver and transmitter.

Table 15-3 shows the DVB-ASI transmitter reference design implementation details.

Table 15-4: DVB-ASI Pass-Through Implementation Details

<table>
<thead>
<tr>
<th>Design Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Used in Implementation</td>
<td>Virtex-II Pro XC2VP4</td>
</tr>
<tr>
<td>Family Targeted</td>
<td>Virtex-II and Virtex-II Pro</td>
</tr>
<tr>
<td>Board Implemented</td>
<td>Cook Technologies SDV Demo Board</td>
</tr>
<tr>
<td>Resource Utilization RX:</td>
<td>200 slices</td>
</tr>
<tr>
<td></td>
<td>2 Block RAMs</td>
</tr>
<tr>
<td></td>
<td>2 BUFGs</td>
</tr>
<tr>
<td></td>
<td>1 DCM</td>
</tr>
<tr>
<td>TX:</td>
<td>50 slices</td>
</tr>
<tr>
<td></td>
<td>1 Block RAM</td>
</tr>
<tr>
<td></td>
<td>2 BUFGs</td>
</tr>
<tr>
<td></td>
<td>1 DCM</td>
</tr>
<tr>
<td>HDL Used</td>
<td>VHDL and Verilog</td>
</tr>
<tr>
<td>Synthesis Tool</td>
<td>Synplify 7.6.1(1)</td>
</tr>
<tr>
<td>Implementation Tool</td>
<td>Xilinx ISE 6.3</td>
</tr>
</tbody>
</table>

Notes:
1. Synthesis of the reference design has been verified only with Synplify.

BIST Test Bitstream Generator and Checker

The BIST test bitstream generator outputs an MPEG-2 compliant packet with a known bitstream that can be checked by the pass-through receiver. Figure 15-25 shows the packet description.

![Figure 15-25: Test Packet Description](x509_19_022205)

BIST Parameter Setting

The DVB-ASI specification defines 2 modes of transport:

- Contiguous byte mode
• Interleave byte mode

When data is transmitted in contiguous byte mode, the entire 188-byte MPEG-2 data packet is transmitted as a block separated by sync bytes, as shown in Figure 15-26.

---

DVB-ASI specifications dictate that each MPEG-2 packet must be preceded by two sync bytes. This enables re-synchronization within one MPEG-2 packet, in case of a link failure. For more details on the DVB-ASI transport mode specification, refer to the DVB-ASI specifications.

In the reference design, the TRANSPORT_MODE parameter controls the transport mode of the test bitstream. This parameter is located in the `dvb_top.v` or `dvb_top.vhd` source file. When this parameter is set to 0, the BIST ASI transmitter transmits in contiguous byte mode (see Figure 15-28). When this parameter is set to 1, the BIST ASI transmitter transmits in interleave byte mode (Figure 15-29).
The TX_SEL parameter, located in the `dvb_top.v` or `dvb_top.vhd` source file, selects the transmit data path source. When this parameter is set to 0, the transmit data path is set to pass-through mode, which means the transmitter is sourced from the elastic FIFO on the receiver side. When this parameter is set to 1, the transmit data path is sourced from the BIST test bitstream generator. Table 15-5 summarizes these user parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSPORT_MODE</td>
<td>Bit</td>
<td>Indicates ASI transport mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 : Contiguous Byte mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 : Byte Interleave mode</td>
</tr>
<tr>
<td>TX_SEL</td>
<td>Bit</td>
<td>Indicates transmit data path source:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 : Pass-Through Mode (data sourced from Receive FIFO)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 : BIST Mode (data sourced from BIST Test pattern generator)</td>
</tr>
</tbody>
</table>

**Reference Design**

The reference design is provided in VHDL and Verilog. Figure 15-30 shows the design hierarchy for the VHDL and Verilog design files.

---

*Figure 15-30: Reference Design File Structure*

The data recovery module (`des.vhd` and `top_data_recovery.vhd`) is provided only in VHDL.

When synthesizing the Verilog design, mixed mode synthesis in Synplicity must be used.
SDV Demo Board Implementation

The DVB-ASI physical layer receiver and transmitter design is verified on the SDV Demo Board, available from Cook Technologies at www.cook-tech.com. The simplified board layout is shown in Figure 15-31. Proper settings for the board are shown in Table 15-6.

![SDV Demo Board Settings Diagram]

**Figure 15-31: SDV Demo Board Settings**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>All DIP switches set to ON</td>
</tr>
<tr>
<td>B</td>
<td>Push button switch 2 (SW2) System Reset</td>
</tr>
<tr>
<td>C</td>
<td>Push button switch 3 (SW3) Force TX Error</td>
</tr>
<tr>
<td>D</td>
<td>Push button switch 4 (SW4) Clear error LED</td>
</tr>
<tr>
<td>E</td>
<td>Rotary switch set to 0</td>
</tr>
<tr>
<td>F</td>
<td>8B/10B Disparity or Code Error LED</td>
</tr>
<tr>
<td>G</td>
<td>BIST Error LED</td>
</tr>
</tbody>
</table>

**Table 15-6: SDV Demo Board Settings**
For this demonstration, SD-SDI connectors are used instead of DVB-ASI compliant connectors. This is a common practice in the industry. Advantages of using the SD-SDI connectors include the following:

- Ability to use the National Semiconductor CLC014 cable equalizer on the SDI-RX connector (not available on the ASI-RX connector)
- Ability to drive two standards (SD-SDI and DVB-ASI) from the same pair of connectors

The CLC014 cable equalizer allows the user to drive up to 300 m of cable, instead of only 70 m on the ASI-RX connector.

For more details about the SDI I/Os on the SDV Demo Board, refer to the board documentation available from Cook Technologies at [www.cook-tech.com](http://www.cook-tech.com).

The default HDL code parameter settings are listed in Table 15-7:

### Table 15-7: Default Parameter Settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>File Location</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLOC_ORIG_CONST</td>
<td>des.vhd</td>
<td>X18Y0: Locks the placement of the tap delay line to the bottom side of the FPGA close to the SDI-RX pin.</td>
</tr>
<tr>
<td>SIDE</td>
<td>des.vhd</td>
<td>1: Indicates a bottom implementation.</td>
</tr>
<tr>
<td>SYNC_MODE</td>
<td>des.vhd</td>
<td>1: Receiver locks only when two sync bytes on the same byte boundary are found in a 5-byte window.</td>
</tr>
<tr>
<td>TRANSPORT_MODE</td>
<td>dvb_top.v</td>
<td>0: Indicates contiguous byte mode.</td>
</tr>
<tr>
<td></td>
<td>dvb_top.vhd</td>
<td></td>
</tr>
<tr>
<td>TX_SEL</td>
<td>dvb_top.v</td>
<td>0: Indicates pass-through mode (data from Receive FIFO).</td>
</tr>
<tr>
<td></td>
<td>dvb_top.vhd</td>
<td></td>
</tr>
</tbody>
</table>

### LVDS I/O Standards

When implementing the design on the SDV Demo Board with an Engineering Sample (ES) part, the LVDS_25_DCI I/O standard must be followed for all LVDS inputs. When implementing the design on the SDV Demo Board with a Production part, the LVDS_25_DT I/O standard must be followed for all LVDS inputs. The LVDS I/O standards for input are set in the `dvb_top.ucf` user constraint file.
In the `dvb_top.ucf` file, use the following constraints for ES silicon:

```plaintext
INST "DATAIN_IBUFDS" IOSTANDARD=LVDS_25_DCI;
INST "CLKIN_IBUFGDS" IOSTANDARD=LVDS_25_DCI;
```

For Production silicon, use the following constraints:

```plaintext
INST "DATAIN_IBUFDS" IOSTANDARD=LVDS_25_DT;
INST "CLKIN_IBUFGDS" IOSTANDARD=LVDS_25_DT;
```

Clocking

The reference design uses the 54 MHz clock source on the SDV Demo Board to derive all requisite clocks for both the receiver and transmitter. One DCM is used to generate the receive clocks and one DCM is used to generate the transmit clocks. Note that only one DCM each is required for the receiver and transmitter, regardless of how many receiver and transmitter modules are instantiated. Figure 15-32 and Figure 15-33 show DCM settings for both the receiver and transmitter clocks.

**Figure 15-32: Receiver DCM Settings**

![Receiver DCM Settings](x509_26_022205)

**Figure 15-33: Transmitter DCM Settings**

![Transmitter DCM Settings](x509_27_022205)

Error Detection

The reference design detects two types of errors: 8B/10B and BIST errors. 8B/10B errors are simply an OR function of the disparity error and code error flags of the 8B/10B decoder core. Whenever the receiver encounters a disparity error or code error, the 8B/10B error LED is asserted. The receiver automatically goes into reframe mode and starts looking for
sync bytes to which it can reframe. However, the error LED remains on and must be cleared by pushing the **Clear Error LED** button (SW4).

The BIST error LED is asserted when the receiver detects a BIST error. This LED remains on until the **Clear Error LED** button (SW4) is pushed.

When the design is first loaded onto the SDV Demo Board, either of the two error LEDs might light up. Push the **Clear Error LED** button (SW4) to clear them.

**BIST Test**

The BIST test is performed as follows:

1. Set the TRANSPORT_MODE parameter to either contiguous byte mode or interleave byte mode.
2. Set the TX_SEL parameter to 1.
3. Implement the code, and load the bitstream onto the SDV Demo Board.
4. Connect the BNC connector of the SDI-TX output to the SDI-RX input with a 75-ohm cable, as shown in **Figure 15-34**.

---

**Figure 15-34: BIST Setup**

Connect SDI-TX to SDI-RX with a 75 ohm cable for BIST
5. If either error LED lights up immediately upon downloading the bitstream, clear the LEDs by pushing the Clear Error LED button (SW4).
6. The BIST operation is running. Monitor the error LEDs for any transmission errors.

Conclusion

This section describes an implementation of the DVB-ASI physical layer using SelectIO features. Both the receiver and transmitter design are discussed. The design is targeted towards Virtex-II and Virtex-II Pro families. The reference design connects the receiver and transmitter in pass-through mode. Using parameters, users can easily configure the reference design in either pass-through mode or BIST mode. The reference design has been verified to synthesize with Synplify. The reference design is implemented on the SDV demonstration board available from Cook Technologies at www.cook-tech.com.
Implementing the DVB-ASI Physical Layer with RocketIO Transceivers

Introduction

The previous section, “Implementing the DVB-ASI Physical Layer with SelectIO Features,” describes the DVB-ASI physical layer implemented on the SelectIO resources that are available in both Virtex-II and Virtex-II Pro families. This section describes how to use the RocketIO™ multi-gigabit transceivers available in the Virtex-II Pro family of FPGA devices to implement a DVB-ASI receiver.

The RocketIO transceivers are designed to support serial bit rates from 622 Mb/s to 3.125 Gb/s. However, the current DVB-ASI bit rate, at 270 Mb/s, is well below the range supported by the RocketIO transceivers. Therefore, the main focus of this section is on a technique that allows the RocketIO transceiver to support this slower bit rate without violating any of its specifications. This technique was used in the multi-rate HD/SD-SDI application developed by John Snow (see Chapter 13, “Multi-Rate HD/SD-SDI Receiver Using RocketIO Multi-Gigabit Transceivers”). The method presented here can be used in conjunction with Chapter 13 to build a multi-standard receiver that can receive HD-SDI, SD-SDI and DVB-ASI video streams.

RocketIO DVB-ASI Receiver

Figure 15-35 shows the DVB-ASI receiver block diagram. Figure 15-36 provides a close-up of the ASI RX block.
Cable Equalization

The DVB-ASI protocol does not specify a cable length requirement. However, most users adhere to the same standards as SD-SDI when implementing their DVB-ASI design. SD-SDI allows maximum cable lengths of up to 300 meters. The coax cable causes frequency-dependent attenuation of the signal, where the higher frequency components of the signal are attenuated more than the lower frequency components. The coax cable also causes frequency-dependent phase distortion, where the higher frequency components are phase shifted more than the lower frequency components. After passing through long coax cables, the signal is severely distorted and attenuated. The receiver must compensate for this attenuation and distortion before attempting to recover the signal. Cable length equalization is used to compensate for the attenuation and distortion introduced by the coax cable.

Typically, an adaptive cable length equalizer is used in SDI receivers. Such an equalizer actively monitors the amount of attenuation and distortion present on the incoming signal and applies the correct amount of equalization to the signal. The cable length can be changed without requiring a change to the equalizer, as would be the case if fixed length equalization were used.

The RocketIO transceivers in the Virtex-II Pro FPGA do not include adaptive cable length equalizers. So, an external cable equalizer must be used to interface the SDI cable to the RocketIO transceiver. As a side benefit, the cable equalizer also converts the single-ended SDI signal into a differential signal. The CML inputs of the RocketIO receiver require a differential input signal. Most multi-rate SDI cable equalizers currently available have 3.3V LVPECL outputs that are not directly compatible with the 2.5V CML inputs of the RocketIO transceiver. AC coupling can be used to interface the LVPECL outputs of the cable equalizer to the CML inputs of the RocketIO transceiver. Figure 15-37 shows a typical AC-coupled interface between a Gennum GS1524 cable equalizer and a RocketIO transceiver.

There are several important details in Figure 15-37:

- The recommendations given in the GS1524 data sheet [Ref 7] must be followed for the interface network between the BNC cable connector and the GS1524’s input.
The AC coupling capacitors between the GS1524 and the RocketIO receiver must be in the 1 µF to 10 µF range to pass the SDI pathological waveforms without too much voltage droop. Typically, 4.7 µF capacitors are used.

The input impedance of the RocketIO transceiver should be set equal to impedance of the circuit board traces between the cable equalizer and the transceiver’s inputs. Normally this is 50Ω.

As described in the RocketIO Transceiver User Guide [Ref 8], when using AC coupling, the RocketIO receiver termination voltage (VTRX) must be between 1.6V and 1.8V. As shown in Figure 6, the required termination voltage can be generated from 2.5V by using a voltage divider network. The resistor values shown are sized to supply the termination voltage to a single RocketIO transceiver, so this resistor network must be duplicated for each RocketIO transceiver used as an SDI receiver.

Some Virtex-II Pro devices have internal power filter capacitors for the VTRX, VTTX, AVCCAUXRX, and AVCCAUTX signals of each RocketIO transceiver. Consult the RocketIO Transceiver User Guide for more information.

It is absolutely essential that all guidelines given in the RocketIO Transceiver User Guide for layout, bypass capacitors, and power regulation and filtering be observed. Do not attempt to provide power to the RocketIO transceiver from a switching regulator.

Data Recovery Unit (DRU)

After cable equalization, the ASI receiver recovers the data from the ASI bitstream. This is typically done with a PLL-based clock and data recovery (CDR) unit. In this case, the RocketIO transceivers are used to recover the data from the serial bitstream. The native DVB-ASI bit rate at 270 Mb/s is well below the 622 Mb/s minimum bit rate supported by RocketIO transceivers in Virtex-II Pro devices. Therefore, it is necessary to work around this minimum bit rate limitation in order to support DVB-ASI with the RocketIO transceivers.

The RocketIO transceiver can transmit the DVB-ASI bitstream by simply over-clocking the transceiver at some multiple of the DVB-ASI bit rate and then sending each encoded bit multiple times. On the receive side, the RocketIO transceiver’s native CDR circuit could theoretically be used to recover DVB-ASI bitstreams. This is because the maximum run length without transitions for 8B/10B-encoded DVB-ASI is 5 bits. Even when over-sampling the bitstream at 8X, this maximum run length multiplies out to only 40 bits, well under the RocketIO transceiver’s 75-bit maximum run length.

When implementing multiple standards such as SD-SDI on a single RocketIO transceiver, however, the transceiver’s native CDR circuit cannot be used, primarily due to run-length limitations. Although SD-SDI’s maximum run length of 39 consecutive bits without a transition (1) is well within the RocketIO transceiver’s 75-bit maximum run length limit, over-sampling the bitstream by at least 3X is required to meet the RocketIO transceiver’s minimum input frequency requirement of 270 Mb/s. Consequently, a 39-bit run length actually appears 117 bits long to the transceiver, clearly exceeding its maximum run length limit by more than 50%. Thus, the CDR unit in the RocketIO transceiver does not maintain lock on the bitstream when simple over-sampling is attempted.

1. SD-SDI bitstreams carrying PAL digital composite (4fsc) video can have a maximum run length of 40 bits. However, this chapter is mainly targeted at digital component video, which has a worst case SD-SDI run length of 39 bits.
Because the CDR unit in the RocketIO transceiver was not designed to work with the low bitstream frequencies and other characteristics typical of DVB-ASI, Xilinx has developed an oversampling technique that can be used with the RocketIO transceivers. This technique does not rely on the CDR section of the RocketIO transceiver for either clock or data recovery. The receiver section of the RocketIO transceiver is used as an asynchronous sampler. Data recovery is done in a data recovery unit (DRU) implemented in the fabric of the FPGA. The PLL in the CDR section of the RocketIO transceiver does not have to lock to the bitstream frequency for this technique to work. Therefore, the SD-SDI run lengths are not an issue for the RocketIO transceiver when using this technique. In the interest of making the DVB-ASI receiver integrable with SD-SDI receiver designs, this new oversampling technique is used as the data recovery unit.

The DRU takes the raw, over-sampled bitstream data from the output of the RocketIO transceiver. It scans the over-sampled data, looking for bit transitions. From these transitions, it determines the optimum place to sample each bit, producing a recovered bitstream on its output. The over-sampling technique allows the RocketIO transceivers in the Virtex-II Pro devices to receive DVB-ASI bitstreams at 270 Mb/s. The DRU generates a clock enable output. This clock enable is used with a global clock to allow all downstream logic in the DVB-ASI receiver to run at the 27 MHz rate.

The data recovery unit used in this design is identical to the one discussed in Chapter 13. Please refer to this chapter for more details.

Basic Operation

The DRU requires a reference clock called \( r\text{clk} \), which it divides by a set factor to produce a clock enable output called \( r\text{clkDvEn} \). If, for example, \( r\text{clk} \) is running at 108 MHz, the DRU would divide it by 4 and assert \( r\text{clkDvEn} \) for one cycle of every four cycles of \( r\text{clk} \). The \( r\text{clkDvEn} \) clock enable and \( r\text{clk} \) can then be used to clock logic downstream from the DRU at 27 MHz. \( r\text{clk} \) can be any integer or half-integer multiple of the 27 MHz DVB-ASI word rate from 4 to 10. In this design, the over-sampling rate is 8X. The DRU corrects for differences between the local \( r\text{clk} \) frequency and the recovered data rate by sometimes inserting or removing one clock cycle between \( r\text{clkDvEn} \) assertions. For example, if a clock divider of 4 is being used, occasionally, there are 2 or 4 \( r\text{clk} \) cycles between assertions of \( r\text{clkDvEn} \), rather than the normal 3. The data out of the DRU changes synchronously with the rising edge of \( r\text{clk} \) when \( r\text{clkDvEn} \) is asserted. Figure 15-38 shows the timing of the DRU.

![Figure 15-38: Clock Enable Timing](https://www.xilinx.com/xapp514/images/f15_38.png)

*DRU frequency adjustment takes place here, shortening to clock period by one rclk cycle.

The \( r\text{clk} \) reference clock can come from a local oscillator that is totally independent of the recovered clock from the RocketIO transceiver. However, one convenient way to use the DRU is to use the recovered clock (RXRECLK) from the RocketIO transceiver as the \( r\text{clk} \) reference clock. See the reference design description for more details.
RocketIO Transceiver Clocks

The RocketIO transceiver requires two types of clocks: reference clocks and user clocks. The reference clocks are used by the RocketIO transceiver as a reference for the CDR PLL. The user clocks are used to clock data out of the RocketIO transceiver and into the fabric of the FPGA. In addition, the RocketIO receiver also produces a recovered clock, called RXRECCLK. The following sections describe the clocking requirements of the RocketIO transceivers oriented towards implementing a DVB-ASI interface. More details about the clocking requirements of the RocketIO transceivers can be found in the RocketIO Transceiver User Guide [Ref 8].

Reference Clocks

The RocketIO transceiver uses reference clocks for two different purposes:

- In the transmitter, the reference clock provides a low-jitter frequency reference that the transmitter multiplies by 20 to obtain a bit-rate clock for the transmitter’s serializer.
- In the receiver, the reference clock is used to spin up the CDR unit so that it quickly locks to the incoming bitstream. After the CDR unit is locked to the bitstream, the frequency of the PLL is constantly compared to the frequency of the reference clock to determine if the PLL is maintaining lock to a valid bitstream frequency. The reference clocks are required to be 1/20th the frequency of the bitstream ±100 ppm.

For DVB-ASI, the reference clock determines the rate at which the bitstream is sampled by the RocketIO transceiver. The reference clock frequency calculation is shown in Equation 1.

\[
\text{reference clock frequency} = \frac{\text{(bitstream freq) \times (over-sample rate)}}{20} \quad \text{Eq. 1}
\]

Each RocketIO transceiver has four reference clock inputs from which a single reference clock is selected. There are two pairs of reference clocks called REFCLK and BREFCLK. The difference between the pairs of reference clocks is that the BREFCLK inputs are designed for lowest possible jitter. The BREFCLKs can only come from certain special IOBs. By contrast, the REFCLKs can come from any IOB or from anywhere in the FPGA. For this design, the clk_108M is used as the reference clock and is connected to the REFCLK input of the RocketIO transceiver. clk_108M is a 108 MHz clock generated by the digital clock manager (DCM) from the 54 MHz on-board crystal oscillator which gets multiplied by 20X inside the RocketIO to support an 8X oversampling of the 270 Mb/s bitstream.

User Clocks

The user clocks clock data out of the RocketIO transceiver and into the fabric of the FPGA. Each transceiver requires two user clocks on the receiver side called RXUSRCLK and RXUSRCLK2. Each transceiver also has two user clocks for the transmitter side called TXUSRCLK and TXUSRCLK2. If the transmitter portion of the transceiver is not used, the TXUSRCLK and TXUSRCLK2 inputs must still be driven with valid clock signals. In this case, simply connect TXUSRCLK to RXUSRCLK and TXUSRCLK2 to RXUSRCLK2. RXUSRCLK is the clock signal that clocks data out of the RocketIO transceiver. The receiver’s output ports, such as RXDATA, change synchronously with the rising edge of RXUSRCLK. For DVB-ASI the frequency of RXUSRCLK is equal to reference clock frequency. The frequency and phase relationships between RXUSRCLK and RXUSRCLK2 depend on the width of the RXDATA port of the RocketIO transceiver. The DRU used for DVB-ASI expects the RXDATA port of the RocketIO transceiver to be 20 bits wide. When using a 20-bit wide output data path from the RocketIO transceiver, RXUSRCLK2 must have the same frequency and phase as RXUSRCLK (simply connect RXUSRCLK and RXUSRCLK2 to the same clock signal).
In serial protocols that have clock correction capability, the RXUSRCLK and RXUSRCLK2 signals usually are derived from the same source as the reference clock. The RocketIO transceiver’s clock correction capability is used to occasionally insert or remove idle characters to compensate for the minor differences between the actual clock frequency of the incoming bitstream and the frequency of the local reference clock.

For DVB-ASI, RXUSRCLK and RXUSRCLK2 should be driven by RXRECCLK. For DVB-ASI over-sampling, the PLL in the RocketIO transceiver is either locked to a harmonic of the DVB-ASI bitstream or it is locked to the reference clock input. In either case, the RXRECCLK, which is derived from the PLL, indicates the rate at which the over-sampled data is being captured by the transceiver. Since the oversampling rate used in this design is 8X, the RXRECCLK frequency is 108 MHz. RXRECCLK is used to clock data out of the transceiver, by connecting it to RXUSRCLK and RXUSRCLK2, and to clock data into the DRU.

**Deserialization**

As the serial bitstream processes through the RocketIO transceiver and the CDR unit, it is deserialized to produce a parallel data stream. While it is possible to implement DVB-ASI decoding and framing in a serial fashion, doing so at the gigabit serial clock rates of 1.485 Gb/s is not possible in today’s FPGA technology. Instead, the DVB-ASI bitstream is deserialized into 10-bit words for processing by the framer and decoder. The clock driving downstream processes (framer and decoder) is the RocketIO recovered clock RXRECCLK which runs at 108 MHz. The clock enable signal from the DRU is used by the downstream processes to give an effective rate of 27 Mb/s, which is the word rate of the DVB-ASI bitstream.

**Parallel Framer**

Figure 15-39 shows a block diagram of a parallel implementation of a framer. The `par_framer.*` files contain the HDL descriptions of the parallel framer module. The parallel framer accepts 10-bit unframed data words. It looks for a 10-bit sync byte which is the 8B/10B K28.5 comma character that can begin at any of the 10 bits in the input word and can span from the first word through the next word. The comma detection logic needs to look across a total of 19 bits to determine if a comma symbol is present and to determine its offset. The incoming data is pipelined through a register called `in1_reg`. The 10 bits from this register plus the nine LSBs from the input port form the 19-bit wide vector that the comma detection logic examines.

A series of 10-bit wide AND and NOR gates examines the 19-bit input vector to determine if a comma symbol is present. If so, an internal comma_detected signal is asserted, and the offset of the comma symbol is determined. The offset encoder produces a numerical offset value indicating the starting bit position of the comma symbol. The output of the offset encoder is compared to the current offset value stored in the offset register to determine if the newly detected comma symbol is at a different offset position. The nsp logic uses the output of the comparator to generate the nsp signal and to load the offset register from the output of the offset encoder when resynchronization occurs. The offset register controls a barrel shifter that extracts the 10-bit output word from a 19-bit wide piece of the input video stream.

Once the first comma character is detected, the framer logic looks for a second comma character aligned on the same word boundary within a 5 byte window per ASI specifications. The purpose of this procedure is to reduce the possibility of detecting a false comma character which can happen when juxtaposing certain words in the 8B10B character set. If the second comma character is detected within specifications, data is framed and the ‘framed’ flag is asserted.
The framer used in this design is adapted from the parallel framer used in Chapter 4, “SD-SDI Video Decoder.” There is another option the user can choose to implement the parallel framer using the 18x18 multiplier as the barrel shifter. Please refer to Chapter 4 for more details.

**8B/10B Decoder**

The method of encoding used in the DVB-ASI protocol is 8B/10B encoding. The 8B/10B code is DC-balanced allowing for cable equalization. Furthermore, 8B/10B transmissions have a limited run length; no more than 5 consecutive ones or zeroes. The 8B/10B set of codes also provides special comma (K) characters that are useful for packet delimiters. These 10-bit characters are guaranteed not to occur with any input combination. One comma character, the K28.5 is used as a packet delimiter in DVB-ASI specification. The 8B/10B encoding scheme also provides a convenient error detection scheme. The error detection scheme is based on the concept of disparity. Disparity is defined as the difference between ones and zeroes in a word. Positive disparity refers to the excess of 1s over 0s. Negative disparity refers to the excess of 0s over 1s. During normal operation, a running disparity is kept. A running disparity is the record for disparity for the aggregate of previously encoded symbols. For each decoded symbol, its disparity is added to the running disparity. Error detection in the decoder is achieved via 2 means; code error and disparity error. A code error is called when the 10-bit encoded symbol does not match any symbol in the code set. A disparity error is called when the running disparity does not match a certain value. By using both these methods of error detection, a very robust error detection scheme is achieved. In the reference design, the code error and disparity error flags are OR-ed together to get the error flag, error_condition. The 8B/10B encoding

![Parallel Framer Diagram](image-url)
scheme provides all these benefits for a very low 25% overhead. The DVB-ASI specification mentioned in the introduction discussed the 8B/10B encoding scheme in greater detail.

User Instructions for Instantiating Core Generator 8B/10B Decoder

The 8B/10B decoder core is provided free of charge from Xilinx. The core is available from the Core Generator tool. Follow the instructions below to instantiate the core in your design.

1. See Figure 15-40. When opening the Core Generator tool for the first time, a project must be selected to which Core Generator writes all its files. Select the ISE project being worked in.

2. In Core Generator, navigate to the Communications & Networking → Building Blocks folder where the 8B/10B encoder and decoder cores are stored. Double-click the Decode 8b/10b core.

![Figure 15-40: Generating 8B/10B Encoder and Decoder from Core Generator](image)

The Xilinx 8b/10b Encoder Core implements the full code set proposed by A.X. Widmer and P.A. Franaszek. The core supports the encoding of an 8-b...
3. A dialog box similar to Figure 15-41 appears. Select the parameters as shown in the figure and click **Generate**. The 8B/10B Decoder core is generated.

![Figure 15-41: Generating 8B/10B Decoder from Core Generator](image-url)
Sync Byte Insertion / Deletion

An elastic FIFO sits in the receive data chain to provide rate-matching between the incoming data rate and the system frequency. The FIFO state machine monitors the FIFO level and adjusts the incoming data stream when necessary to keep the FIFO from overflowing or underflowing. The state machine logic is shown in Figure 15-42.

![FIFO Control State Machine](image_url)

**Figure 15-42:** FIFO Control State Machine

When initialized, the FIFO state machine fills the FIFO until the halfway mark. From then on, it monitors both the high and low water mark level Figure 15-43. If the FIFO reaches the high water mark level (indicating that the incoming data rate is faster than the system / sampling frequency), the state machine deletes one comma character. If the FIFO reaches the low water mark level (indicating that the incoming data rate is slower than the system / sampling frequency), the state machine adds one comma character.

The FIFO state machine also stuffs the FIFO with comma characters whenever the system loses lock (cable is unplugged or link encounters errors.)
Figure 15-43: FIFO Levels

- High-water mark: FIFO_LEVEL = 11
- FIFO Midpoint: FIFO_LEVEL = 10
- Low-water mark: FIFO_LEVEL = 01
User Instructions for Instantiating Core Generator FIFO

The FIFO used in the reference design is a 1024 x 10 asynchronous FIFO. The Asynchronous FIFO core is provided free of charge from Xilinx. The core is available from the Core Generator tool. Follow the instructions below to instantiate the core in your design.

1. See Figure 15-44. When opening the Core Generator tool for the first time, a project must be selected to which Core Generator writes all its files. Select the ISE project being worked in.

2. In Core Generator, navigate to the Memories & Storage Elements → FIFOs folder where the FIFO cores are stored. Double-click the Asynchronous FIFO core.

---

**Figure 15-44: Generating FIFO from Core Generator**
3. A dialog box similar to Figure 15-45 appears. Select the parameters as shown in the figure and click **Generate**. The 8B/10B Encoder core is generated.

![Generating FIFO from Core Generator](image_url)

**Figure 15-45**: Generating FIFO from Core Generator
RocketIO DVB-ASI Transmitter

Figure 15-46 shows the DVB-ASI transmitter block diagram.

The DVB-ASI bit rate at 270 Mb/s is well below the 622 Mb/s minimum bit rate supported by RocketIO transceivers in Virtex-II Pro devices. Therefore, it is necessary to work around this minimum bit rate limitation in order to support DVB-ASI with the RocketIO transceivers.

To transmit DVB-ASI, the RocketIO transceiver is configured to run at some integer multiple of the DVB-ASI bit rate and each bit is sent multiple times consecutively. For example, given a reference clock of 54 MHz, a RocketIO transceiver multiplies this reference clock by 20 resulting in a 1.08 Gb/s bitstream-exactly four times 270 Mb/s. If each encoded bit is transmitted by the RocketIO transmitter four times consecutively, the RocketIO transmitter produces a bitstream that is, in every way, equivalent to a normal 270 Mb/s DVB-ASI bitstream.

Figure 15-47 shows this in more detail.
Any bitstream frequency that is any integer multiple of the SD-SDI bit rate can be used, so long as it is fast enough to meet the minimum frequency requirements for the RocketIO transceiver. “User Clocks,” page 331 and “User Clocks,” page 341 discuss clock requirements in detail.

Cable Driver

It is very common in the industry to use cable drivers meant for SDI in DVB-ASI designs. This allows the channel to support a multi-standard (SDI and DVB-ASI) transmission. For this purpose, the DVB-ASI reference design uses the same cable driver as the multi-rate SD/HD-SDI design. Please refer to Chapter 12, “Multi-Rate HD/SD-SDI Transmitter Using Virtex-II Pro RocketIO Multi-Gigabit Transceivers” for more details.

Clocking Requirements

One of the most important aspects of implementing an SDI transmitter using RocketIO transceivers is providing the right clocks to the transceivers. RocketIO transceivers requires two types of clocks, reference clocks and user clocks. The reference clocks are used to generate the bit-rate clock for the serializer. The user clocks are used to clock data from the fabric of the FPGA into the RocketIO transceiver. More details about the clocking requirements of the RocketIO transceivers can be found in the RocketIO Transceiver User Guide and in Chapter 9.

Reference Clocks

The reference clocks provide low-jitter frequency references for the RocketIO transceiver. The RocketIO transceiver multiplies the selected reference clock by 20 to obtain a bit-rate clock for the transmitter’s serializer. Jitter present on the reference clock shows up as jitter on the transmitter output, so it is important that a low jitter reference clock be used.

For DVB-ASI, the reference clock must be some multiple of the DVB-ASI word-rate clock since the DVB-ASI video clocks are too slow to be used directly as reference clocks to the RocketIO transceiver.

The source of the reference clocks is application specific. In most cases parallel digital video is supplied to the transmitter with an associated word-rate clock from some external source. For DVB-ASI, the word-rate clock would typically need to be multiplied by at least two to get a clock fast enough for the RocketIO transceiver. In either case, keep in mind that an external PLL might be required to reduce the jitter on video clocks so that they are suitable for the RocketIO transceiver reference clocks. Chapter 9 discusses jitter reduction requirements in more detail.

User Clocks

The user clocks load data into the RocketIO transceiver from the fabric of the FPGA. Each RocketIO transceiver requires two transmitter user clocks called TXUSRCLK and TXUSRCLK2. TXUSRCLK must always be frequency locked to the selected reference clock. There is no required phase relationship between TXUSRCLK and the selected reference clock, but they must be exactly the same frequency. The frequency and phase relationships between TXUSRCLK and TXUSRCLK2 depend on the width of the TXDATA input port of the RocketIO transceiver.

When implementing DVB-ASI running at 270 Mb/s, it would be slightly more convenient to use a 40-bit wide TXDATA port. This is because each encoded bit must be replicated four times if the transceiver’s bit rate is four times the DVB-ASI bit rate. Since each encoded DVB-ASI word is 10 bits long, the resulting vector to the TXDATA port is 40 bits after bit
replication. The width of the TXDATA port is fixed at FPGA configuration time. Because a 40-bit TXDATA port requires the use of a DCM to produce proper frequency and phase relationships between TXUSRCLK and TXUSRCLK2, it can be more desirable to use a 20-bit TXDATA port. So, the 40-bit DVB-ASI bit vector resulting from bit replication must be sent to the TXDATA port as two 20-bit vectors, least significant half first.

The user clocks are global FPGA clocks generated by the digital clock manager (DCM) that are also used to clock the parts of the DVB-ASI transmitter implemented in the FPGA fabric. The clocks are 54 Mhz clocks identified as gclk_54M in the reference design.

8B/10B Encoder

The method of encoding used in the DVB-ASI protocol is 8B/10B encoding. The 8B/10B code is DC-balanced allowing for cable equalization. Furthermore, 8B/10B transmissions have a limited run length; no more than 5 consecutive ones or zeroes. The 8B/10B set of codes also provides special comma (K) characters that are useful for packet delimiters. These 10-bit characters are guaranteed not to occur with any input combination. One comma character, the K28.5 is used as a packet delimiter in DVB-ASI specification. The 8B/10B encoding scheme also provides a convenient error detection scheme. The error detection scheme is based on the concept of disparity. Disparity is defined as the difference between 1s and 0s in a word. Positive disparity refers to the excess of 1s over 0s. Negative disparity refers to the excess of 0s over 1s. During normal operation, a running disparity is kept. A running disparity is the record for disparity for the aggregate of previously encoded symbols. For each decoded symbol, its disparity is added to the running disparity. Error detection in the decoder is achieved via 2 means; code error and disparity error. A code error is called when the 10-bit encoded symbol does not match any symbol in the code set. A disparity error is called when the running disparity does not match a certain value. By using both these methods of error detection, a very robust error detection scheme is achieved. In the reference design, the code error and disparity error flags are OR-ed together to get the error flag, error_condition. The 8B/10B encoding scheme provides all these benefits for a very low 25% overhead. The DVB-ASI specification mentioned in the introduction discussed the 8B/10B encoding scheme in greater detail.
User Instructions for Instantiating Core Generator 8B/10B Encoder

The 8B/10B encoder core is provided free of charge from Xilinx. The core is available from the Core Generator tool. Follow the instructions below to instantiate the core in your design.

1. See Figure 15-48. When opening the Core Generator tool for the first time, a project must be selected to which Core Generator writes all its files. Select the ISE project being worked in.

2. In Core Generator, navigate to the Communications & Networking → Building Blocks folder where the 8B/10B encoder and decoder cores are stored. Double-click the Encode 8b/10b core.

![Figure 15-48: Generating 8B/10B Encoder from Core Generator](image)

---

**Figure 15-48:** Generating 8B/10B Encoder from Core Generator
3. A dialog box similar to Figure 15-49 appears. Select the parameters as shown in the figure and click **Generate**. The 8B/10B Encoder core is generated.

![Generating 8B/10B Encoder from Core Generator](x509_48_040705)

**Figure 15-49:** Generating 8B/10B Encoder from Core Generator

### DVB-ASI Bit Replication

For DVB-ASI, each bit from the encoder must be replicated some number of times depending on the reference clock frequency and the DVB-ASI bitstream frequency. With a 54 MHz reference clock, each bit must be replicated four times to generate a 270 Mb/s DVB-ASI bitstream. This produces 40 bits from each 10-bit input word. If the TXDATA port of the RocketIO transmitter is 20 bits wide, the 40 bits from the bit replicator must be multiplexed to first provide the least significant 20 bits during one TXUSRCLK cycle and the most significant 20 bits during the next TXUSRCLK cycle.

The bit replicator shown in **Figure 15-50** is an implementation of a bit replicator designed for 4X bit replication. It includes an output MUX to produce a 20-bit vector for the RocketIO transceiver's TXDATA port.

![DVB-ASI Bit Replication](x509_49_040705)

**Figure 15-50:** DVB-ASI Bit Replication
The bit replicator output vector must be bit-swapped to reverse the bit order before actually being connected to the TXDATA port of the RocketIO transceiver because the transceiver transmits the MSB first. The sdiasi_rio_refclk module used in the reference design includes this bit swap function. The sdiasi_rio_refclk module is a wrapper around the RocketIO primitive GT_CUSTOM. It is identical to the hdsdi_rio module from Chapter 9 except that the REFCLK inputs are active instead of the BREFCLK inputs. Due to the design of the SDV demo board, the REFCLK inputs must be used in this reference design instead of the BREFCLK inputs to the RocketIO transceiver.

### Jitter Performance

The physical layer design for DVB-ASI is identical to the design used in Chapter 12. Therefore the performance numbers for the DVB-ASI design is identical as well. Please refer to Chapter 12 for more details.

### Resource Utilization

**Table 15-8: Resource Utilization, RocketIO MGT Implementation**

<table>
<thead>
<tr>
<th></th>
<th>Slices</th>
<th>BUFG</th>
<th>Block RAM</th>
<th>DCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass-through:</td>
<td>480</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>RX only:</td>
<td>310</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TX only:</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Conclusions

This section describes a technique that can be used to allow DVB-ASI bitstreams to be transmitted and received using the RocketIO multi-gigabit transceivers in the Virtex-II Pro FPGA family. By combining this technique with the multi-rate HD/SD-SDI receiver design described in Chapter 12 and Chapter 13, a multi-standard receiver and transmitter can be implemented that supports HD-SDI, SD-SDI and DVB-ASI. All Virtex-II Pro devices contain multiple RocketIO transceivers, making it possible to implement multiple multi-standard receivers in a single Virtex-II Pro device. For applications that require multiple multi-standard transmitters and receivers, this can provide a high level of integration, saving board space, power, and reduce cost as compared to discrete multi-standard transmitter and receiver implementations.

### Design Files

The reference design files are available on the Xilinx website at:


Open the ZIP archive and extract file `xapp514_dvbasi-phy.zip`. 

Section V: Video Test Pattern Generators

Audio/Video Connectivity Solutions for the Broadcast Industry
SDTV Video Pattern Generators

Summary

This chapter describes methods of efficiently generating standard video test patterns in Xilinx FPGAs. Video test patterns are used to verify the proper operation of video equipment. Most video equipment capable of generating a video signal can produce one or more video test patterns to verify proper operation of the video generator and attached video equipment. Thus, there is often a need to have a video test pattern generator embedded in the video equipment.

Two basic video pattern generator designs have been described in this chapter. The first is based on distributed SelectRAM™ memory and is applicable to any current generation Xilinx FPGA family. The second design is based on the block SelectRAM memory in the Virtex™-II series. The design can implement sophisticated and flexible video pattern generation using very few Virtex-II device resources.

A Brief Component Digital Video Primer

Component Digital Video Standards

There are many different video standards, both analog and digital. Today, most broadcast studios and video production centers use component digital video when creating, storing, and transporting video. Component digital video can be readily compressed using digital video compression standards. It can also be encoded into analog composite video for broadcast.

Probably, the most common component digital video standards in use today are based on the 4:2:2 sampling scheme. The 4:2:2 component digital video format is used in various standards for 525-line (NTSC), 625-line (PAL), wide-screen NTSC and PAL, and HDTV. Table 16-1 lists some of the 4:2:2 component digital video standards.

Table 16-1: Common 4:2:2 Component Digital Video Standards

<table>
<thead>
<tr>
<th>Standard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMPTE 125M(1) and ITU-R BT.601-5(2)</td>
<td>NTSC &amp; PAL 4x3 aspect ratio 4:2:2 component digital video</td>
</tr>
<tr>
<td>SMPTE 267M</td>
<td>NTSC 16x9 aspect ratio 4:2:2 component digital video</td>
</tr>
</tbody>
</table>
Color Space

Black-and-white TV uses only intensity information, called luminance or luma designated with the letter Y. When color information was added, the luma signal was left intact for compatibility with existing equipment, and two components of color information, called U and V, were added. The two color components are often called color difference signals because they are derived by taking the difference between a color’s intensity and the overall luminance of the sample. The U component is the difference between blue and Y. The V component is the difference between red and Y.

The PAL and NTSC TV broadcast systems are both based on the YUV color space. NTSC can also optionally use a derivative of YUV, called YIQ. ’I’ stands for in-phase and ’Q’ for quadrature, reflecting the modulation method used to transmit the color information.

The YCbCr color space is commonly used in component digital video. YCbCr is a scaled and offset version of the YUV color space with a luma component (Y) and two chroma (color difference) components (Cb and Cr). The Y component has a nominal 8-bit range of 16 through 235. The two chroma components have nominal 8-bit ranges of 16 to 240. Some values above and below the nominal ranges are used to encode special signals.

Sampling Schemes

One of the key characteristics of digital component video formats is the sampling scheme. Component video sampling schemes are denoted with a sequence of numbers separated by colons, such as 4:2:2 and 4:4:4.

A 4:2:2 sampling scheme indicates that for every four samples of luma (Y), there are two samples each of the two chroma signals (Cb and Cr). In standard definition video, the luma is sampled at a 13.5-MHz rate, while each chroma component is sampled at half that rate. This takes advantage of the fact that the human eye is less sensitive to color than to intensity to reduce the signal bandwidth by sampling color components at a lower frequency than the luma components.

Other common video sampling schemes are 4:4:4, where there are an equal number of Y, Cb, and Cr samples, and 4:1:1, where there is only one sample of each chroma signal for every four luma samples. A sampling scheme called 4:2:0 is often used in digital video compression standards and involves compression of the chroma components in both the horizontal and vertical direction rather than just the horizontal direction as is in 4:2:2. However, 4:2:2 is the most common sampling scheme in use today for component digital video in broadcast studios and video production centers.
Video Format

For NTSC video, each video line contains 858 samples. As shown in Figure 16-1, a sample contains two words: a Y component word and a chroma component word, either Cb or Cr. Consecutive samples alternate between containing Cb or Cr components. The active video portion of the line consists of samples 0 through 719. The inactive portion or horizontal blanking interval of the line consists of samples 720 through 857.

For NTSC video, the four words of sample pairs 720/721 and 856/857 contain special codes called timing reference signals (TRS). The 720/721 pair contains the end of active video (EAV) TRS symbol, and the 856/857 pair contains the start of active video (SAV) TRS symbol. These TRS symbols are used to mark the transitions between the active and inactive portions of the line and also contain other timing information. When using 10-bit video words, the first three words of the TRS symbol are \(3FF_{HEX}, 000_{HEX}, \) and \(000_{HEX}\). The fourth word of the TRS symbol is called the XYZ word. Three bits of the XYZ word are used to indicate the status of the F, V, and H bits; four bits are used as error detection bits; and the remaining bits are fixed in value.

The F bit indicates whether field one \((F = 0)\) or field two \((F = 1)\) is active. The V bit is set to 1 in TRS symbols on lines that are part of the vertical blanking interval. On active video lines, the V bit is 0. The H bit distinguishes between EAV and SAV symbols. "H" is always a 1 in EAV symbols and always a 0 in SAV symbols.

The encoding of the TRS symbol's XYZ word is shown below:

**Figure 16-1: NTSC and PAL Video Line Detail**

The encoding of the TRS symbol's XYZ word is shown below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>F</td>
<td>V</td>
<td>H</td>
<td>P3</td>
<td>P2</td>
<td>P1</td>
<td>P0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Chapter 16: SDTV Video Pattern Generators

The bits labeled P3 through P0 are protection bits and are calculated in the following manner:

\[
\begin{align*}
P3 &= V \oplus H \\
P2 &= F \oplus H \\
P1 &= F \oplus V \\
P0 &= F \oplus V \oplus H \\
\end{align*}
\]

Figure 16-2 and Figure 16-3 show the arrangement of the vertical regions for both NTSC and PAL component digital video. The diagram shows the line numbers on which the F and V bits change values. For example, in NTSC video, lines 1 through 3 have both the F and V bits set to "1". On lines 4 through 19, the V bit is still a "1", but the F bit is a "0".

An NTSC video frame consists of 525 lines and is divided into two interlaced fields. Frames are drawn at a rate of 30 Hz. However, because new fields are drawn at a rate of 60 Hz, the flicker that the eye would perceive in a 30-Hz image is significantly reduced.

PAL video lines have the same number of active samples (720) as NTSC video. However, PAL has a few more inactive samples per line. PAL frames consist of 625 lines divided into two interleaved fields. The refresh rate of PAL is lower than NTSC with frames drawn at a 25-Hz rate (50-Hz field rate).
**Figure 16-2**: NTSC Video Frame Details

<table>
<thead>
<tr>
<th>Line</th>
<th>H=1</th>
<th>V=1</th>
<th>H=0</th>
<th>V=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EAV</td>
<td>F V</td>
<td>SAV</td>
<td>F V</td>
</tr>
<tr>
<td>2</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-18</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22-262</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>263</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>264</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>265</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>266</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>267</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>268-281</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>282</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>283</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>284</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>285-524</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>525</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Numbering Quirks

There are a few interesting quirks with the numbering of lines and samples in digital video. Lines are numbered beginning with one. In NTSC video, field 1 begins with line 4, not line 1. Field 1 includes lines 4 through 265 and field 2 includes lines 266 through 525 plus lines 1 through 3.

The samples along a horizontal line are numbered starting with zero. Sample 0 is the first sample of the active portion of the line. However, a new line does not actually begin at sample 0. Most digital video standards specify the beginning of the line as the first sample
of the EAV symbol, sample 720. This is because the EAV symbol’s V and F bits reflect the
status of the line that follows, so it is convenient to think of the EAV symbol as being the
beginning of the line.

In NTSC video, the definition of line 20 is somewhat unclear. Some documents define line
20 as the first active line of the odd field. Others define line 20 as the last line in the vertical
blanking interval and line 21 as the first active line. Some of this confusion arises from the
fact that the number of lines in the vertical blanking interval is given as a minimum of 19
lines in many NTSC standards, implying that it can be longer than 19 lines. If lines 1
through 19 are used as the vertical blanking interval, then line 20 is the first active line.
Some prefer to use line 21 as the first active line because this gives an equal number of
active lines (243) in each field. Using line 20 as the first active line gives 244 lines in the odd
field and 243 in the even field.

Another element in this confusion is that some earlier versions of the NTSC digital
component video standards ANSI/SMPTE 125M and ITU-R BT.656 allowed the V bit in
the TRS XYZ word to transition from 1 to 0, indicating the end of the vertical blanking
interval, on any line from 10 to 20 for the odd field and 273 to 283 for the even field. Current
versions of these documents are now very precise in specifying that the V bit should be 1
on line 19 and 0 on line 20, making line 20 the first active line of the odd field.

Because of the ambiguity surrounding line 20, some video equipment manufacturers
building NTSC digital video equipment treat line 20 as an active line, but avoid putting
critical video information in the active video portion of line 20. The video test pattern
generators in this chapter all treat NTSC line 20 as a valid active line.

Video Test Pattern Standards

Standards for Color Bar Test Patterns

Many of the most commonly used video test patterns fall into the class called "color bars." Color bar test patterns consist of several vertical bars filled with primary and
complementary colors. Color bar test patterns are particularly useful for verifying proper
operation of video encoders and decoders and for adjusting video monitors.

One of the early color bar standards, traditionally called RS-189-A, is now officially called
EIA-189-A [Ref 3]. Refer to Figure 16-4 for a diagram of the EIA-189-A color bar standard.
The EIA-189-A test pattern consists of seven vertical color bars that occupy the top 75% of the test pattern. These color bars are called “75% bars,” not because they occupy 75% of the picture, but because the luma value of these bars is set to 75% of the maximum luma value.

The bottom 25% of the test pattern consists of four bars of the colors –I, white, +Q, and black. The white bar is called 100% white because the luma component is set to 100%. The black bar is often called 0% black because the luma component is set to the black level or 0% luma. The –I and +Q colors represent full scale I and Q values in the YIQ color space. The –I color represents a signal with the maximum negative I value and a Q value of zero. The +Q signal represents a signal with a maximum positive Q value and an I value of zero.

SMPTE improved the EIA-189-A color bar pattern in engineering guideline, EG 1-1990. The SMPTE EG 1 test pattern is now one of the most commonly used video test patterns.

As shown in Figure 16-5, the EG 1 color bar pattern added to the EIA-189-A pattern a narrow middle band of color bars called the "new chroma set" bars. The new chroma set bars are arranged so that when the red and blue guns of a video monitor are turned off and only the blue gun is active, the brightness of each bar in the new chroma set should match the brightness of the 75% bar located immediately above it.

EG 1 also adds several narrow “near black” bars useful for setting the black level of monitors. These bars are sometimes called the PLUGE signal (Picture Line Up Generating Equipment). To adjust the black level of the monitor the brightness control is adjusted so that the black+4% (or whiter-than-black) bar is just visible but the black–4% (or blacker-than-black) bar is not distinguishable from the surrounding 0% black bars.
SDI Pathological Test Patterns

Many video test patterns have been developed to aid in testing specific aspects of video equipment performance. An example of this is the SMPTE RP 178-1996 Serial Digital Interface (SDI) Checkfield.

Equipment complying with the SMPTE 259M SDI standard is widely used in broadcast studios and production centers to transport digital video over standard video coax cable. The SDI standard defines how to send digital video serially at bit rates ranging from 143 Mb/s to 360 Mb/s. To compensate for signal loss in the coax cable, the SDI standard requires adaptive cable length equalization at the receiver. This equalization circuit can be stressed by waveforms that have a high amount of DC content.

SDI receivers also require a clock and data recovery (CDR) circuit, usually based on a Phase Locked Loop (PLL), to recover the serial bitstream at the receiver. The CDR circuit requires bit transitions periodically to stay locked to the bit rate of the bitstream. Low frequency waveforms with long runs of 1s or 0s stress the CDR’s ability to stay locked when few transitions are present in the bitstream.

The SMPTE recommended practice RP 178-1996 defines two test patterns, one to test the receiver equalization by producing a bitstream with a maximum amount of DC content.
and another to test the CDR circuit’s low frequency response by producing a bitstream with long runs of 1s or 0s. The SDI "checkfield", as the RP 178 test pattern is called, has a cable equalizer test pattern in the first half (top) of each active video field and a CDR test pattern in the second half (bottom) of each active video field.

In the cable equalizer test pattern, all chroma (Cb and Cr) components have values of \(300_{\text{HEX}}\) (all values are 10-bit values) and the luma (Y) components have values of \(198_{\text{HEX}}\). This pattern, when encoded by an SDI encoder, occasionally generates a repeating serial pattern that has 19 High bits followed by one Low bit or 19 Low bits followed by one High bit. This pattern produces a maximum amount of DC offset. To insure that both polarities of this pattern are generated, the entire video frame must have an odd number of 1 bits at the input to the SDI encoder. This is done by setting the Y component of the last sample on the first active line of the first field (line 20 for NTSC or line 23 for PAL) to a value of \(080_{\text{HEX}}\) instead of \(198_{\text{HEX}}\).

In the CDR test pattern, all chroma components have a value of \(200_{\text{HEX}}\) and all luma components have a value of \(110_{\text{HEX}}\). Feeding this pattern into an SDI encoder for one-half of a field produces several lines of a repeating waveform that has 20 consecutive bits of one polarity followed immediately by 20 consecutive bits of the opposite polarity, producing a minimum number of transitions to the CDR circuit.

The SDI encoding scheme uses a linear feedback shift register (LFSR) to scramble the video data. The starting state of the LFSR affects how any 10-bit video word is encoded by the SDI encoder. An SDI encoder has 511 different possible starting states. When the RP 178 patterns are encoded by an SDI encoder, they do not immediately nor consistently generate the pathologic waveforms. They only generate the pathological waveforms once the encoder has reached a certain starting state. In the half field where each of the two test patterns is applied to the SDI encoder, the pathological waveforms are only generated by the encoder during a few of the active video lines.

**Reference Designs**

Two basic video pattern generator designs are presented here, with a few minor variations of each design type also provided. The first type of pattern generator is based on distributed RAM found in most Xilinx FPGA families. The second basic type is based on the Virtex-II synchronous 18K-bit block RAM.

**Limiting Signal Transition Rates**

Many video standards require the video component values to have limited transition rates. This is because analog video devices have a limited amount of bandwidth and cannot handle video signals that transition quickly from one value to another. Color bar patterns have high transition rates at the borders between adjacent color bars.

A test pattern generator can limit the signal transition rates by ramping the value of the digital components at the color bar transitions. However, since video encoders often limit signal transition rates, it is often easier to let the video encoder perform this function.

The test pattern generators described in this chapter do not limit the transition rates of the signals. If transition rate limiting needs to be implemented in the FPGA, this can be done with a video FIR filter connected to the output of the test pattern generator. The FIR filter function must only be applied to the active video data and not to the timing reference signals or any non-video digital data that might be included in the blanking intervals. The design of a video FIR filter is outlined in other Xilinx applications notes.
Some test patterns, such as the RP 178 SDI test patterns, must not be filtered. Filtering the RP 178 patterns prior to SDI encoding does not achieve the correct test effect.

**Distributed RAM Video Pattern Generators**

Figure 16-6 provides a block diagram of a color bar pattern generator based on ROMs implemented in distributed RAM. This pattern generator produces the SMPTE EG 1 color bar pattern. The pattern generator can be broken down into three main sections: the horizontal section, the vertical section, and the component video generator section.

**Horizontal Section**

The horizontal section contains a horizontal counter and a horizontal state machine. The horizontal counter increments every clock cycle, counting the number of words (two words per video sample) on a horizontal video line. The two least significant bits (LSBs) of the horizontal counter are used to determine which component to output: Y (01 or 11), Cb (00), or Cr(10). The horizontal counter is reset to zero by the horizontal state machine when the end of the video line is reached.

The horizontal state machine sequences through a series of horizontal regions on each video line. The transition from one horizontal region to another is called a horizontal “event.” A horizontal event must be defined at each possible point on the line where the outputs of the horizontal state machine must change. These events occur where a new
color bar could begin or a TRS symbol must be generated. In this design, horizontal events can only occur at four clock boundaries. That is, horizontal regions can only begin where the least significant two bits of the horizontal counter are both zero.

**Figure 16-7** shows the EG 1 color bar pattern. Along the bottom of the drawing, the horizontal events and regions are defined. Each horizontal event is marked by a dotted line. The horizontal counter value for the beginning of each horizontal region is also shown. These counts are valid for NTSC video. Note how some color bars span multiple horizontal regions. For example, the top red color bar spans three regions because of the three small PLUGE bars located below it.

The last horizontal region on the right of the pattern is defined as just the last two samples of the line and is the horizontal region where the horizontal state machine asserts a signal to cause the vertical state machine to increment the line counter.

**Figure 16-7: Horizontal Regions of the EG 1 Test Pattern Generator**

The horizontal state machine consists of a horizontal region counter containing the current state (horizontal region) of the state machine and a ROM to decode the current state into the control outputs. One of the outputs of the ROM is a 9-bit “next-event” value. A comparator constantly compares the most significant nine bits from the horizontal counter to the next-event value from the ROM. When they match, an event has been reached and the horizontal region counter is incremented, moving the horizontal state machine to the next horizontal region. The horizontal region counter increments only when the least-significant two bits of the horizontal counter are both 1s.
The horizontal ROM also generates several other outputs:

- \( clr_h \) clears the horizontal counter when the end of the line is reached
- \( inc_v \) causes the vertical counter to increment to the next video line
- \( trs \) indicates that a TRS symbol is generated in the current horizontal region
- \( h \) H bit (horizontal blanking indicator) for the TRS XYZ word

**Vertical Section**

The vertical section contains a vertical counter that keeps track of the current line number. It increments from one to the maximum number of lines in the frame. The horizontal section controls when the vertical counter increments by asserting the \( inc_v \) signal at the end of each horizontal line. The vertical counter is cleared to a value of 1 (remember that the first video line is 1) when the vertical state machine asserts the \( clr_v \) signal indicating the end of the frame.

The vertical section also contains a vertical state machine that is almost identical to the horizontal state machine. A vertical region counter contains the current vertical region value.

A ROM decodes the vertical region value into a number of control bits, including a 10-bit next-event value that is constantly compared to the current value of the vertical counter. When the current line number matches the next-event value from the ROM, the vertical region counter is incremented. The vertical region counter only increments at the beginning of the video line as indicated by the horizontal state machine asserting the \( inc_v \) signal.

Different vertical regions are required to keep track of the changes in the V and F bits and for the different vertical patterns in the EG 1 pattern. Figure 16-8 shows an NTSC video frame with the 11 different vertical regions that the vertical state machine cycles through to process one frame of video. Note that the last region, region 10, is only active for the last video line. During this region, the \( clr_v \) signal is asserted to cause the vertical counter and the vertical state machine to reset to the beginning of the frame when the end of the line is reached.
The outputs of the vertical state machine are:

\[ f \] is the F bit (field indicator) for the TRS XYZ word

\[ v \] is the V bit (vertical blanking indicator) for the TRS XYZ word

\( clr_v \) clears the vertical counter and the vertical region counter at the end of the video frame

\( vband \) indicates which vertical region is currently active

The 2-bit \( vband \) signal indicates which of the three patterns (color bar sets) should be generated, based on the current vertical position. The EG 1 test pattern has three color bar sets located in three different rows on the screen. The fourth value that \( vband \) can assume indicates that the current vertical region is a vertical blanking interval.

**Component Video Generator Section**

The component video generator section converts the 2-bit sample code from the horizontal counter, the 4-bit horizontal region code, and the 2-bit \( vband \) code into actual video component values. Two ROMs are used in the component video generator section.

The color ROM converts the \( vband \) and horizontal region codes into a 4-bit color code. The EG 1 test pattern uses 13 different colors, leaving three unused color codes.

When generating colors, not TRS symbols, the 4-bit code from the color ROM and a 2-bit sample code derived from the two LSBs of the horizontal counter form the address into the video ROM. The video ROM generates the 10-bit value for each component of the color.
The sample code tells the video ROM whether to output the Y, Cb, or Cr components of the color. This uses three of the four possible values of the sample code. The fourth value of the sample code indicates that a TRS symbol should be generated. When the horizontal state machine asserts the TRS signal, a MUX located between the color ROM and the video ROM replaces the color code from the color ROM with the F, V, and H bits. The video ROM encodes the F, V, and H bits into a 10-bit XYZ word for the TRS symbol.

The video ROM only generates the XYZ word of the TRS symbol. It does not generate the first three words of the TRS symbol. The values of these first three words are $3\text{FF}_{\text{HEX}}, 000_{\text{HEX}},$ and $000_{\text{HEX}}$. These values are generated by a MUX on the output of the video ROM. When a TRS symbol is being generated, the MUX supplies 3FF for the first word, 000 for the second and third words, and the video ROM supplies the XYZ word for the fourth word. The use of a MUX to generate the trivial 3FF and 000 values reduces the amount of space needed in the video ROM.

The EG 1 color bar generator using distributed RAM is implemented in the `cb_eg1.v` and `cb_eg1.vhd` files. When generating NTSC or PAL video using this design, a 27-MHz clock should be used.

Generating the RP 178 SDI Checkfield

The RP 178 SDI checkfield pattern is relatively simple. It consists of one pattern during the first half of the active field and another pattern during the second half. However, there is one exception where the last Y component on the first active line of the first field has a different value than the other Y components in the cable equalization pattern.

In the `cb_eg1_rp178.*` files, an RP 178 pattern generator has been grafted onto the EG 1 color bar generator described above. An input signal to this module indicates whether the EG 1 or RP 178 pattern should be generated. The RP 178 generator simply looks at the horizontal and vertical counters to determine which video component values to output. This RP 178 generator only generates values during the active portion of the video. The regular color bar pattern generator takes over and generates the TRS symbols and blanking interval values.

A reference design that generates only the RP 178 SDI Checkfield test pattern is provided in the `rp178.v` and `rp178.vhd` files. This design is based on the distributed RAM test pattern generator design, but only generates the RP 178 test pattern, making it smaller than the combined EG 1 and RP 178 test pattern generator.

Simple Color Bars

The HDL files `colorbars.*` contain a simplified version of the EG 1 color bar generator. This version simplifies the bottom pattern so that a gray bar occupies the left half of the pattern and black bar occupies the right half. This eliminates the need to generate the colors $-I$, $+Q$, white, and the two near-black signals. This reduces the number of colors needed from 13 to 8, allowing the color code generated by the color ROM to be reduced from four to three. These changes reduce the size of the video ROM to half and eliminate one bit from the color ROM, resulting in a smaller implementation.

This simplified version can be used when space is at a premium in the FPGA and strict adherence to the EG 1 standard is not required.

Block RAM Video Pattern Generators

The dual-port Virtex-II block RAMs allow two independent test pattern generators to be implemented, using the same amount of hardware as required to implement one pattern.
generator. The second generator is essentially free. The two pattern generators must share the same ROM data, meaning they generate the same patterns but do not have to be synchronized in any way. An EG 1 color bar generator can be made using three Virtex-II block RAMs and very few other FPGA resources.

Figure 16-9 is a block diagram of a block RAM-based video pattern generator. The block diagram shows only one pattern generator, but this design implements two independent pattern generators in three block RAMs and four Virtex-II slices. If there are three unused block RAMs in a design, a video pattern generator can be added for almost no additional cost.

Figure 16-9: Video Pattern Generator Using Block RAMs

HROM

The HROM is a block RAM configured as a 1Kx18 device. It implements the horizontal state machine with the internal register of the block RAM serving as the current state register. Ten bits out of the HROM form the “next-state” value and wrap back around to the address input of the HROM. The HROM state machine advances one state every four clock cycles. A 2-bit sample counter is decoded to provide the clock enable signal for the HROM.

The HROM can implement up to 1024 states. Because each state lasts for two video samples, the HROM can accommodate test patterns that are up to 2048 samples wide, sufficient to cover most standard definition video formats. Some wide-screen standard definition video formats and some high-definition video formats have more than 2048 samples per line. There are two ways to adapt the design for these HDTV video formats. First, the number of samples per state could be increased to four, providing for up to 4096 samples. Doing so would require some changes to the design to correctly generate the TRS symbol during half of the state. Second, an additional HROM could be added to expand the HROM to 2Kx18, providing twice as many states.
The HROM generates an \textit{h\_region} code to indicate which horizontal region is currently active. The \textit{h\_region} code can be either four or five bits wide, depending on the requirements of the test pattern.

The HROM asserts a signal called \( h \) during the horizontal blanking interval. It also generates an enable signal to the vertical state machine. This enable signal indicates the end of the current line and causes the vertical state machine to increment to the next line when asserted.

**VROM**

The VROM is another 1Kx18 block RAM used to implement the vertical state machine. It is configured just like the HROM state machine with 10 bits out of the VROM forming the next state value and wrapping back around to the VROM’s address input.

The VROM can implement up to 1024 states. With each state corresponding to one video line, the VROM has enough states to cover most current video resolutions, but does not cover the 1080-line HDTV standards. To adapt this design to cover the higher resolution standards, the VROM can be implemented in two block RAMs each configured as 2Kx9, giving a total RAM space of 2Kx18.

The VROM generates a \textit{v\_region} code to indicate the current vertical region. The \textit{v\_region} code can be four or five bits wide. The VROM also generates a field indicator bit \((f)\) and a vertical blanking indicator bit \((v)\).

**CROM**

The CROM is a third block RAM in a 2Kx9 configuration and is used as the video component generator. The address inputs for CROM come from the 2-bit sample counter, the \textit{h\_region} code from the HROM, and the \textit{v\_region} code from the VROM. If both \textit{h\_region} and \textit{v\_region} are four bits wide, then there is an extra address input to the CROM available. This design example takes advantage of this extra address pin as a pattern select input, allowing either the EG 1 or RP 178 test pattern to be selected.

With the two independent test pattern generators available due to the dual-port nature of the block RAM, one generator can be generating the EG 1 pattern while the other is generating the RP 178 pattern. Or, they can both be generating the same pattern.

The CROM has a 9-bit wide output, so it can only produce 9-bit color components. While this is generally sufficient for most color bar applications, the RP 178 test patterns require all components to be generated at 10-bit resolution. TRS symbols should also be generated accurately to 10-bit resolution. There are several ways to solve this problem.

First, the CROM can be configured as a 1Kx18 part, allowing for more output resolution. This would limit the \textit{v\_region} and \textit{h\_region} codes to 4-bit values and would eliminate the ability to put two patterns in the CROM.

Second, an additional block RAM can be used to double the number of bits out of the CROM. Since many applications only require 10-bit video, it seems a waste to use an entire block RAM to generate one more bit.

Because generating video test patterns as efficiently as possible was a goal of this reference design, another technique was used. The LSB from the CROM is duplicated and used as both of the two LSBs of the component value. This can produce color component values that differ by one bit from the recommendation of some standards. However, it does accurately generate all TRS symbol words, and it also correctly generates all RP 178 test pattern component values.
A 3-bit output register is used to delay the $f$, $v$, and $h$ bits from the VROM and HROM by one clock cycle to match the clock cycle of delay in the CROM.

When generating NTSC or PAL video using the `vidgen` design, a 27-MHz clock should be used.

**Generating the ROM Contents**

Using block RAMs as the basis for a video test pattern generator makes a very flexible design. The test patterns can be changed simply by changing the initialization values of the RAMs, or by reloading the RAMs on the fly. The difficult part is coding the contents of these large RAMs by hand.

As part of the reference design, a utility called `cbgen` has been provided. This utility reads a text file that describes the test pattern and generates initialization files for the three RAMs. Two initialization files are generated for each RAM, one containing the initialization code for simulation and the other containing the synthesis initialization code. This utility generates the initialization files in either VHDL or Verilog and produces correct synthesis code for XST, Leonardo, FPGA Express, or Synplify.

The utility can also generate files compatible with the Xilinx XDL tool, which allows the initialization values of the ROMs to be changed without resynthesizing or rerouting the FPGA design. Refer to the `cbgen User Guide` for a complete description on how to use the `cbgen` utility.

The `vidgen.v` and `vidgen.vhd` files contain the HDL descriptions of the block RAM-based video pattern generator. The `vidgen.v` files contain the `include` directives that include the six RAM initialization files. Some Verilog synthesis tools do not implement the `include` directive. In these cases, the initialization files should be inserted directly into the `vidgen.v` file where the `include` directives currently exist.

VHDL lacks a file include directive, so the `*.vhd` initialization files should be inserted directly into the `vidgen.vhd` file at the places indicated by the comments.

The supplied RAM initialization files generate both the EG 1 and RP 178 test patterns for 4x3 aspect ratio, 525-line NTSC video. A pattern definition file that can be processed by `cbgen` is also provided for the EG 1 and RP 178 test patterns in 625-line PAL format.

The `cbgen` utility is provided pre-compiled and ready to run on a PC under Windows. The C source code for `cbgen` is also provided so that it can be compiled for use under other operating systems.

**Reference Design Results**

Table 16-2 shows the results after "place and route" of the various modules implemented in this chapter. All results were obtained using the Verilog versions of the designs with Xilinx ISE version 4.1i using XST as the synthesis tool. Results using the VHDL files are not shown, but are essentially identical. Virtex-II device results are for a -5 speed grade device. Spartan™-II device results are for a -6 speed grade device.
Conclusions

Video test pattern generators are often included in many types of video equipment, sometimes to provide a quick go/no-go test to determine if the equipment is functional, other times to provide sophisticated diagnostic capabilities.

Xilinx FPGAs are now commonly used in video equipment, so there is a need to efficiently implement video test pattern generators in Xilinx FPGAs. Two basic video pattern generator designs have been described in this chapter.

Design Files

The reference design files are available on the Xilinx website at:

www.xilinx.com/bvdocs/appnotes/xapp514.zip

Open the ZIP archive and extract file xapp514_sdtn-pattern-gen.zip.

Appendix A: cbgen User Guide

Introduction

The cbgen utility was developed to make it easier to generate the block RAM initialization files for the three ROMs in the vidgen video pattern generator reference design discussed in this chapter. The utility reads a text file that describes the video test pattern and generates two initialization files for each of the three ROMs, one for simulation and one for synthesis.

The utility generates the initialization files in either VHDL or Verilog or in a format compatible with the Xilinx XDL tool. XDL allows the ROM initialization values to be modified after synthesis and place and route.

The utility is written in C and the source code is provided to allow the utility to be modified or to be compiled for different operating systems.

This version of cbgen is limited to generating initialization files for designs with only one block RAM per section. It cannot support video formats with more than 1023 lines or 2048 samples per line.

---

Table 16-2: Reference Design Results

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Optimize for Area</th>
<th>Speed Virtex-II Device</th>
<th>Speed Spartan-II Device</th>
<th>Optimize for Speed</th>
<th>Speed Virtex-II Device</th>
<th>Speed Spartan-II Device</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size LUTs/FFs</td>
<td>Speed</td>
<td>Speed</td>
<td>Size LUTs/FFs</td>
<td>Speed</td>
<td>Speed</td>
</tr>
<tr>
<td>colorbars.v</td>
<td>116/42</td>
<td>100 MHz</td>
<td>60 MHz</td>
<td>117/46</td>
<td>140 MHz</td>
<td>80 MHz</td>
</tr>
<tr>
<td>cb_eg1.v</td>
<td>132/42</td>
<td>100 MHz</td>
<td>60 MHz</td>
<td>137/60</td>
<td>140 MHz</td>
<td>80 MHz</td>
</tr>
<tr>
<td>cb_eg1_rp178.v</td>
<td>160/43</td>
<td>90 MHz</td>
<td>60 MHz</td>
<td>171/47</td>
<td>140 MHz</td>
<td>80 MHz</td>
</tr>
<tr>
<td>rp178.v</td>
<td>82/41</td>
<td>140 MHz</td>
<td>90 MHz</td>
<td>86/41</td>
<td>165 MHz</td>
<td>100 MHz</td>
</tr>
<tr>
<td>vidgen.v</td>
<td>6/10</td>
<td>175 MHz</td>
<td>NA</td>
<td>6/10</td>
<td>200 MHz</td>
<td>NA</td>
</tr>
</tbody>
</table>
Input File Format

Basic Syntax

The cbgen input file is a text file that describes the video test pattern to be generated. The file can contain comments that are ignored by cbgen. The comment character is // and can occur anywhere on a line. Anything to the right of the comment character is ignored. Blank lines are ignored.

Generally, cbgen expects a command to exist on a single line of the text file. However, there is a line continuation character: \. Anything on a line to the right of the line continuation character is ignored. The following line is appended to any line with a line continuation character.

Some commands require an item to be named, such as the color definition lines in a PALETTE block. Names must only be a single word, so they cannot contain spaces.

The different elements on a command line must be separated by one or more space or tab characters.

File Sections

The input file is divided into different sections. The sections must appear in the proper order.

The first section contains a number of different commands that establish various parameters. These parameters include the number of words per video line, the number of lines in the video frame, and the names of the output files.

Next, the color palettes are defined using the PALETTE block. This block defines the colors that are used in the video test pattern.

The HORIZONTAL_REGIONS block must come after the palette blocks. This block describes where each horizontal region in the test pattern begins and ends on a video line.

The LINE_FORMATS block must come after the horizontal regions block. This block defines the possible formats that a video line can have. It defines what colors should be generated in each horizontal region for different line types. A different type of line (or format) occurs in each vertical region. For example, in the EG 1 test pattern, the lines in the 75% color bars pattern at the top of the screen have a different format than either the lines in the middle "new chroma set" pattern or the lines in the bottom pattern.

The last section of the input file is the VERTICAL_REGIONS block. This block is where the extent of each vertical region is defined. It also associates the lines within each vertical region with a line format.

Parameter Commands

At the beginning of the file, there must be several single-line parameter commands. These commands can come in any order relative to one another. These commands are described below.

**PALETTES num**

This command specifies how many color palettes are used in the test pattern. The numeric parameter can be either 1 or 2. The number of palettes used is usually one, unless two different test patterns are to be stored with a pattern select bit used to select between them.
The pattern select bit is an extra address bit to the CROM and essentially selects between the two possible color palettes.

Only one color palette can be used if either the hregion or vregion codes are five bits wide.

If the PALETTES command does not appear in the pattern definition file, the number of palettes defaults to 1.

**HREGION_BITS num**

This command specifies how many bits are used to encode the horizontal region. The numeric parameter can be either 4 or 5. Four bits allows 16 horizontal regions to be defined and five bits allows 32 horizontal regions to be defined. In determining how many horizontal regions to use, be sure to note that three horizontal regions are consumed by the EAV, horizontal blanking, and SAV regions. If five bits are used for the horizontal region code, only four bits can be used for the vertical region code, and only one color palette can be used.

If the HREGION_BITS command does not appear in the pattern definition file, the number of horizontal region code bits defaults to 4.

**VREGION_BITS num**

This command specifies how many bits are used to encode the vertical region. The numeric parameter can be either 4 or 5. Four bits allow 16 vertical regions to be defined and five bits allow 32 vertical regions to be defined.

If the VREGION_BITS command does not appear in the pattern definition file, the number of vertical region code bits defaults to 4.

**H_TOTAL num**

This command specifies the total number of words on a horizontal video line. For NTSC video, this value should be 1716. The value must be less than 2048. If the H_TOTAL command does not appear in the pattern definition file, the number of horizontal samples per line defaults to the NTSC value of 1716.

**V_TOTAL num**

This command specifies the total number of video lines in the frame. For NTSC video, this value should be 525. The value must be less than 1024. If the V_TOTAL command does not appear in the pattern definition file, the number of vertical lines per frame defaults to the NTSC value of 525.

**HROM_FILENAME "file name prefix"**

**VROM_FILENAME "file name prefix"**

**CROM_FILENAME "file name prefix"**

These commands specify the prefixes for the names of the output files. The file name prefix must be enclosed in quotation marks. cbgen appends to the supplied file name prefix either ".sim" for the simulation initialization file or ".syn" for the synthesis initialization file and the appropriate file extension type. Because the file name prefix string is enclosed in quotes, space characters are acceptable in the name. If these commands do not appear in the pattern definition file, default file names of "horz_rom", "vert_rom", and "comp_rom" are used.
HROM_INSTANCE "instance name"
VROM_INSTANCE "instance name"
CROM_INSTANCE "instance name"

These commands specify the instance names of the various ROMs. These instance names must match the instance names of the ROMs in the Verilog or VHDL code file. The instance name must be enclosed in quotation marks. If these commands do not appear in the pattern definition file, the instance names default to "HROM," "VROM," and "CROM".

HROM_INIT_STATE num

This command specifies the starting state for the HROM state machine. This is the state that the state machine enters after being reset. The state number must be less than 2048. If this command does not appear in the pattern definition file, the HROM init state defaults to zero.

VROM_INIT_STATE num

This command specifies the starting state of the VROM state machine. This is the state that the state machine enters after being reset. The state number must be less than 1024. If this command does not appear in the pattern definition file, the VROM init state defaults to the V_TOTAL value.

V_INCREMENT num

This command specifies the horizontal count on which the HROM asserts the inc_v signal to cause the VROM to increment to the next vertical line. The inc_v signal is actually asserted for four counts and the two least significant bits of the supplied numeric parameter are ignored. If this command does not appear in the pattern definition file, the value defaults to 1440.

Palette Blocks

The palette block defines the colors that are used in a pattern. Either one or two palette blocks can be defined as specified with the PALETTES command described previously. Generally only one palette is used, but if the pattern generator has a pattern select input to the CROM, this pattern select bit can select between two different color palettes.

A palette block begins with a line containing the PALETTE command and the name of the palette. A palette must be given a name.

After the PALETTE command line comes a series of color definition lines. One color in the palette is defined on each separate line. The palette block ends with a line containing the END command. Anything else after the END command on the same line is ignored, so a command like END PALETTE can be used.

Each color definition line begins with the name of the color followed by the color type as indicated by the reserved words TYPE0 and TYPE1. After the type code, an optional IS word can be used as a separator before either three or four numeric parameters are supplied to define the components of the color.

TYPE0 colors are specified with three components in the following order: Cb, Y, and Cr. The component values are specified in decimal and are 10-bit values. The single Y value is repeated for both samples of the color.

TYPE1 colors are specified with four components in the following order: Cb, Y0, Cr, and Y1. This type allows the specification of different Y values for the two samples of the color.
IMPORTANT: The first color definition line of every palette block must define a color named BLANK. This color is generated during the horizontal and vertical blanking intervals. All color component values must be supplied as 10-bit decimal numbers in the range 0 to 1023.

Below is an example of a palette block.

```
PALETTE eg1
    // name        type        Cb       Y      Cr       Y (type 1 only)
    // ----------- ------ ------ ------ ------ ------ ------ ------
    BLANK        TYPE0 IS    512      64     512
    gray         TYPE0 IS    512     721     512
    yellow       TYPE0 IS    176     674     543
    cyan         TYPE0 IS    589     581     176
    green        TYPE0 IS    253     534     207
    magenta      TYPE0 IS    771     251     817
    red          TYPE0 IS    435     204     848
    blue         TYPE0 IS    848     111     481
    black        TYPE0 IS    512     64     512
    i            TYPE0 IS    612     244     395
    q            TYPE0 IS    697     141     606
    white100     TYPE0 IS    512     940     512
    black-4      TYPE0 IS    512     29     512
    black+4      TYPE0 IS    512     99     512
END
```

Horizontal Regions Block

The horizontal regions block defines the different horizontal regions in a test pattern. A horizontal region must be defined for each possible place on a line where a different color can be generated. For example, in the EG 1 color bar pattern, the red bar in the top color bar pattern actually occupies three separate horizontal regions, one for each of the small black and near-black PLUGE bars below it in the bottom pattern.

The horizontal regions block begins with a command line containing the command HORIZONTAL_REGIONS and ends with the END command line. In between, each horizontal region is defined on an individual line. The horizontal regions of a test pattern are defined from left to right across the video line beginning with the first active sample of the line (count 0). All horizontal count values from zero to the value specified by the H_TOTAL command must be included in a horizontal region.

A horizontal region definition line begins with the horizontal region code to be associated with the region. The code value is specified in decimal and must be between 0 and 15 if HREGION_BITS is 4 or between 0 and 31 if HREGION_BITS is 5. The horizontal region code is used as an address into the CROM and tells it which color to generate based on which horizontal region is active. Horizontal regions can share the same code, so it is possible to define more than 16 or 32 horizontal regions. However, horizontal regions that share the same codes must always share the same color in each vertical region.

Three horizontal region codes must be reserved for the EAV, BLANK, and SAV horizontal regions and these three regions must be defined in the horizontal regions block. Because of how the horizontal code value is used in the LINE_FORMATS block, it is easier to assign the codes sequentially and to put the EAV, BLANK, and SAV codes together at the end of the code space as shown in the example below.
Following the code value on the horizontal region definition line, there can be an optional IS keyword. After that, the extent of the horizontal region is specified by a starting horizontal count value and an ending horizontal count value separated by the keyword TO. Horizontal count values are specified in decimal and indicate the actual horizontal count (there are two counts per video sample, one for the chroma component and one for the luma component). Horizontal regions must begin on a count value that is divisible by four and must end on a value that is one less than a value divisible by four. The first horizontal region should begin at 0 and the last horizontal region should end at one less than the H_TOTAL value.

After the horizontal region’s extent, the region type must be specified. Regions can be of type ACTIVE for regions in the active video space, BLANK for regions in the horizontal blanking region, EAV or SAV for the regions where the TRS symbols are generated. The EAV and SAV regions must have extents of exactly four counts.

```
HORIZONTAL_REGIONS
// code         start    end     type
// ----         -----   ----    ------
0   IS     0 TO  207    ACTIVE
1   IS    208 TO  259    ACTIVE
2   IS    260 TO  415    ACTIVE
3   IS    416 TO  519    ACTIVE
4   IS    520 TO  623    ACTIVE
5   IS    624 TO  779    ACTIVE
6   IS    780 TO  831    ACTIVE
7   IS    832 TO 1039    ACTIVE
8   IS   1040 TO 1107    ACTIVE
9   IS   1108 TO 1179    ACTIVE
10  IS   1180 TO 1247    ACTIVE
11  IS   1248 TO 1435    ACTIVE
12  IS   1436 TO 1439    ACTIVE
13  IS   1440 TO 1443    EAV
14  IS   1444 TO 1711    BLANK
15  IS   1712 TO 1715    SAV
END HORIZONTAL_REGIONS
```

Line Formats Block

The line formats block specifies the various formats used by the video lines in the test pattern. For example, the EG 1 test pattern contains three different patterns, the top 75% color bar pattern, the middle "new chroma set" pattern, and the bottom pattern with the PLUGE signals. To implement an EG 1 color bar pattern, three different line formats would be defined, one for each pattern in the EG 1 test pattern.

If two different test patterns are being defined for the pattern generator, the line formats block should have line formats defined for both patterns. The example below includes formats for both the EG 1 and the RP 178 test patterns in the same line formats block.

The line formats block begins with a LINE_FORMATS command line and ends with the END command line. In between, each line format is specified on an individual line format definition line. Because line format definitions can be long, it is often handy to use the line continuation character to format these lines.

A line format definition begins with a name to be given to the line format. After the name, there can be an optional IN reserved word followed by the name of the color palette to be used for this line format.
After the color palette name comes a list of colors to be used for each horizontal region on the line. A horizontal region is assigned to a color with the syntax:

```
region_code IS color_name
```

For example, the command “0 IS gray” assigns the color gray to the horizontal region code of 0 in this line format. All samples on the video line that fall in any horizontal region having a horizontal region code of 0 are gray in color.

It is common for several continuous regions to have the same color. If these regions have been assigned sequential horizontal region codes, a short hand command can be used to assign them all to the same color. This syntax has the format

```
region_code TO region_code ARE color_name
```

For example, the command “8 TO 10 ARE red” assigns horizontal region codes 8, 9, and 10 the color red.

Every horizontal region except the EAV, SAV, and BLANK regions must be assigned to a color on each line format definition line.

```
LINE_FORMATS

// name palette colors to use for each horizontal code
// ------ ------- ---------------------------------------------------
top_band IN eg1 0 IS gray 1 TO 2 ARE yellow 3 TO 4 ARE cyan \ 5 TO 6 ARE green 7 IS magenta 8 TO 10 ARE red \ 11 TO 12 ARE blue

mid_band IN eg1 0 IS blue 1 TO 2 ARE black 3 TO 4 ARE magenta\ 5 TO 6 ARE black 7 IS cyan 8 TO 10 ARE black \ 11 TO 12 ARE gray

bot_band IN eg1 0 TO 1 ARE i 2 TO 3 ARE white100 4 TO 5 ARE q\ 6 TO 7 ARE black 8 IS black-4 9 IS black \ 10 IS black+4 11 TO 12 ARE black

rp178_ceqx IN rp178 0 TO 11 ARE ceq 12 IS ceqx

rp178_ceq IN rp178 0 TO 12 ARE ceq

rp178_pll IN rp178 0 TO 12 ARE pll

END LINE_FORMATS
```

### Vertical Regions Block

The vertical regions block defines the extent of each vertical region in the video frame. Different vertical regions occur where the test pattern changes from one line format to another. Vertical regions must also be defined for the vertical blanking interval. Usually, more than one vertical region must be defined in each blanking interval because the field indicator bit (F) must transition during the vertical blanking interval.

The vertical region block begins with a line containing the VERTICAL_REGIONS command and ends with the END command line. In between, each vertical region is defined on a separate vertical region definition line.

The vertical region definition line begins with a vertical region code value to be assigned to the vertical region. Multiple vertical regions can be assigned the same vertical region code as long as they have the same attributes. In order to share a code, they must be in the same field and of the same type (ACTIVE or BLANK) and the video lines in the regions must use the same line format. Sharing vertical region codes is usually easier than sharing horizontal regions codes because there are often vertical regions with identical attributes.
Note in the example below how vertical region code 0 is used by both vertical blanking intervals in field 1.

After the vertical region code, there can be an optional IS keyword. This is followed by the vertical region extent definition. The extent is defined with the syntax:

```
start_line TO end_line
```

All lines in the frame from 1 to the last line must be included in a vertical region.

After the region extent definition is an optional IN reserved word followed by the field definition: FIELD0 or FIELD1. This defines whether the F bit generated by the VROM is 0 (FIELD0) or 1 (FIELD1).

After the field definition is the vertical region type, either ACTIVE for regions in the active video region or BLANK for regions in the vertical blanking interval.

ACTIVE region definitions must end with one or two format assignments, depending on how many palettes are defined. BLANK regions do not have a format assignment. The format assignment syntax is:

```
palette_name IS line_format_name
```

This specifies that for all video lines in this vertical region, if the named palette is selected, the given line format should be generated.

```
VERTICAL_REGIONS
// code start end     field    type    palette format    palette format
// ---- ----- ---     -----    ----    ------- ------    ------- ------
0 IS 1 TO 3  IN FIELD1   BLANK
1 IS 4 TO 19 IN FIELD0   BLANK
2 IS 20 TO 20 IN FIELD0   ACTIVE eg1 IS top_band rp178 IS rp178_ceqx
3 IS 21 TO 141 IN FIELD0   ACTIVE eg1 IS top_band rp178 IS rp178_ceq
4 IS 142 TO 196 IN FIELD0   ACTIVE eg1 IS top_band rp178 IS rp178_pll
5 IS 197 TO 217 IN FIELD0   ACTIVE eg1 IS mid_band rp178 IS rp178_pll
6 IS 218 TO 263 IN FIELD0   ACTIVE eg1 IS bot_band rp178 IS rp178_pll
1 IS 264 TO 265 IN FIELD0   BLANK
0 IS 266 TO 266 IN FIELD1   BLANK
7 IS 283 TO 402 IN FIELD1   ACTIVE eg1 IS top_band rp178 IS rp178_ceq
8 IS 403 TO 459 IN FIELD1   ACTIVE eg1 IS top_band rp178 IS rp178_pll
9 IS 460 TO 525 IN FIELD1   ACTIVE eg1 IS mid_band rp178 IS rp178_pll
10 IS 526 TO 525 IN FIELD1  ACTIVE eg1 IS bot_band rp178 IS rp178_pll
END VERTICAL_REGIONS
```

Running cbgen

After the pattern definition file has been created, the `cbgen` utility is used to generate the ROM initialization files. The `cbgen` utility is a command line utility and should be executed in a command line shell in Windows.

The syntax for executing the `cbgen` utility is:

```
cbgen [-s synth_tool] [-l language] input_filename
```

The optional `-s` flag specifies which synthesis tool to target with the initialization files. Different synthesis tools have slightly different syntax for initialization of Xilinx block RAMs. The choices are: `XST`, `SYNOPSYS` (for FPGA Express), `LEONARDO`, and `SYNPLIFY`. If the `-s` flag is not provided, the synthesis tool defaults to `XST`.

Running cbgen
The optional --l flag specifies which language to use for the initialization files. The choices are: VERILOG, VHDL, and XDL. If the --l flag is not provided, the language defaults to VERILOG. If the XDL option is chosen for the language, then the --s flag is ignored.

The input_filename must be the full name, including extension, of the pattern definition file.

**Using the RAM Initialization Files**

**Verilog**

The cbgen utility creates two Verilog files for each of three ROMs. One Verilog file contains the simulation initialization code and the other contains the synthesis initialization code. If the simulation and synthesis tools used supports the Verilog `include directive, then simply modify the six include directives in the vidgen.v file to include the correct file. If the tools do not support the include directive, insert the contents of the appropriate initialization file in place of each include directive in the vidgen.v file.

The simulation initialization code is surrounded by commands to cause the synthesis tool to ignore the simulation specific code. The synthesis specific code is written in the form of a Verilog comment block and is ignored by the simulation tool.

**VHDL**

The cbgen utility creates two VHDL files for each of the three ROMS. One VHDL file contains the simulation initialization code and the other contains the synthesis initialization code.

Unlike Verilog, VHDL does not have a file inclusion directive. So, you must use an editor to manually insert the files generated by cbgen into the vidgen.vhd file at the places indicated by the comments.

The initialization code for simulation is in the form of a generic map for each ROM. This generic map is surrounded by directives to cause the synthesis tool to ignore it.

The synthesis initialization code is a series of attribute definitions. These are user defined attributes and have to be declared before they can be used. The synthesis init file for the HROM defines all the attributes used by the initialization code. Therefore, the HROM synthesis file must be inserted in the vidgen.vhd file before the synthesis files for the other two ROMs.

**XDL**

XDL is a Xilinx utility included with ISE. XDL converts an NCD file to a text file so that it can be manually edited and then converts the text file back to an NCD file. This allows a design that has been run through the synthesis and place-and-route tools to be manually edited. Using the XDL files created by cbgen allows the block RAMs of the vidgen pattern generator to be updated with new initialization values without having to resynthesize or run PAR.

The procedure for using the XDL files is:

1. Run cbgen with the --l XDL flag to generate the three initialization XDL files, one for each ROM.
2. Convert the NCD file of the FPGA design to an XDL text file using the following command:

```
xdl -ncd2xdl ncd_filename
```
XDL creates a text file with the same name as the NCD file with a .xdl extension.

3. Open the XDL file created by the xdl utility in a text editor. Search for the HROM instance. The first few lines of the HROM instance look like this:

```plaintext
inst "HROM" "RAMB16", placed BMR8C1 RAMB16_X0Y0,
cfg "ENAINV::ENA CLKAINV::CLKA WEAINV::WEA SSRAINV::SSRA
CLKBINV::CLKB WEBINV::WEB ENBINV::ENB SSRBINV::SSRB
WRITEMODEA::READ_FIRST PORTA_ATTR::1024X18 RAMB16A:HROM.A:
WRITEMODEB::READ_FIRST PORTB_ATTR::1024X18 RAMB16B:HROM.B:
INIT_00::0X0010000F000E000D000C000B000A000900080007000600050004000300020001
INIT_01::0X0020000F0010001D001C001B001A001900180017001600150014001300120011
```

Replace all the lines starting with the line beginning with WRITEMODEA through, but not including the final line of the instance block, with the contents of the HROM XDL file. The last line of the instance block is a line with a semicolon (;) on a line by itself. Leave the last line intact.

Repeat step 3 for the VROM and CROM instances.

4. Convert the XDL file back to an NCD file using the XDL utility like this:

```plaintext
xdl –xdl2ncd xdl_filename
```

An optional ncd_filename can be supplied, otherwise the original NCD file is overwritten.

5. The resulting NCD file can be processed to generate a bit file that can be loaded into the FPGA.
Chapter 17

HDTV Video Pattern Generator

Summary

This chapter describes a technique for generating high-definition (HD) video test patterns with Xilinx FPGAs. Video test patterns are used to verify the proper operation of video equipment. Most video equipment capable of generating a video signal can also produce one or more video test patterns to verify proper operation of the video equipment.

The video test pattern generator presented here uses the 18 kb block SelectRAM memory present in the Virtex™-II, Virtex-II Pro, and Spartan™-3 FPGA families from Xilinx. The block memories are used to hold the video patterns. Several different video patterns can be stored in the block memories. The video pattern generator described in this chapter produces three commonly used HD video test patterns. The video test pattern generator supports all 18 of the HD digital component video formats supported by the HD-SDI (SMPTE 292M) standard [Ref 1]. The video test pattern generator uses very few FPGA resources, and it can easily be placed in the same device with other video processing functions.

HD Digital Component Video

Professional broadcast studios and video production centers typically use digital component video as the preferred video format for content creation, storage, and editing. Chapter 16, “SDTV Video Pattern Generators” contains a description of digital component video and describes a video pattern generator for standard-definition video.

Video Format Naming

There are two basic scanning methods used for video: interlaced and progressive. The industry standard naming scheme for video formats uses the number of active lines followed by either “i” for interlaced or “p” for progressive. An interlaced video format with 1080 active lines would be called 1080i. Often, the frame rate is also added to the end of the video format name. For example, a 1080i format with a 30 Hz frame rate would be called 1080i30.

In interlaced scanning, the frame is split into two fields with one field containing the odd lines and the other frame containing the even lines. When displayed on a monitor, all the lines from one field are drawn, followed by all the lines from the next field. The two fields are actually from different images taken at different times. There are two interlaced video formats currently defined for HDTV: 1035i with 1035 active lines, and 1080i with 1080 active lines. The 1035i format was an early HDTV format with non-square samples and has been almost entirely replaced by the 1080i formats.

In the progressive formats, the frame is not split into fields. All of the lines of the frame are from the same image. When displayed on a monitor, all of the lines of the frame are
displayed sequentially from top to bottom. There are two progressive video formats currently defined for HDTV: 720p with 720 active lines and 1080p with 1080 active lines.

SMPTPE recommended practice RP 211 adds a third scanning method called segmented frame. Segmented frame scanning is really progressive video that has been reformatted to make it compatible with interlaced equipment. In the early days of HDTV development, equipment for progressive scan video, particularly video recorders, was not as readily available as equipment for interlaced video. The segmented frame technique provided a standard method for taking progressive scan video and reformatting it to be compatible with video recorders and other equipment designed for interlaced video. In segmented frame video, each progressive scan frame is separated into two fields, one containing the even lines and the other containing the odd lines. Thus, the video appears to be interlaced. It is, in fact, indistinguishable from interlaced video. However, the two fields in the segmented frame represent one progressive frame and are taken from one image whereas, the two fields of an interlaced frame represent separate images taken at different times. Before being displayed, the two fields of a segmented frame are recombined into a progressive frame for display on a progressive scan monitor. The segmented frame formats use the letters "sF" in the format name like this: 1080sF30.

Digital Component Video Standards

The most commonly used digital component video formats used in the broadcast industry today are based on the 4:2:2 sampling scheme using the Y’Cb’Cr’ color space. Table 17-1 shows a list of commonly used HD 4:2:2 component digital video standards. All of these formats are compatible with the SMPTPE 292M HD-SDI standard for transporting HD digital video over coax cable or optical fiber. The last column shows the SMPTPE 292M format designations for these video standards. The format designation for the segmented frame formats show the equivalent 1080p standard.

<table>
<thead>
<tr>
<th>SMPTE Standard</th>
<th>Format</th>
<th>Frame Rate (Hz)</th>
<th>Sample Rate (MHz)</th>
<th>Active Samples/Line &amp; Active Lines/Frame (words x lines)</th>
<th>Total Samples/Line &amp; Total Lines/Frame (words x lines)</th>
<th>Format Designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMPTE 260M</td>
<td>1035i</td>
<td>30</td>
<td>74.25</td>
<td>1920 x 1035</td>
<td>2200 x 1125</td>
<td>A</td>
</tr>
<tr>
<td>SMPTE 260M</td>
<td>1035i</td>
<td>30/M</td>
<td>74.25/M</td>
<td>1920 x 1035</td>
<td>2200 x 1125</td>
<td>B</td>
</tr>
<tr>
<td>SMPTE 295M</td>
<td>1080i</td>
<td>25</td>
<td>74.25</td>
<td>1920 x 1080</td>
<td>2376 x 1250</td>
<td>C</td>
</tr>
<tr>
<td>SMPTE 274M</td>
<td>1080i</td>
<td>30</td>
<td>74.25</td>
<td>1920 x 1080</td>
<td>2200 x 1125</td>
<td>D</td>
</tr>
<tr>
<td>SMPTE 274M</td>
<td>1080i</td>
<td>30/M</td>
<td>74.25/M</td>
<td>1920 x 1080</td>
<td>2200 x 1125</td>
<td>E</td>
</tr>
<tr>
<td>SMPTE 274M</td>
<td>1080i</td>
<td>25</td>
<td>74.25</td>
<td>1920 x 1080</td>
<td>2640 x 1125</td>
<td>F</td>
</tr>
<tr>
<td>SMPTE 274M</td>
<td>1080p</td>
<td>30</td>
<td>74.25</td>
<td>1920 x 1080</td>
<td>2200 x 1125</td>
<td>G</td>
</tr>
<tr>
<td>SMPTE 274M</td>
<td>1080p</td>
<td>30/M</td>
<td>74.25/M</td>
<td>1920 x 1080</td>
<td>2200 x 1125</td>
<td>H</td>
</tr>
<tr>
<td>SMPTE 274M</td>
<td>1080p</td>
<td>25</td>
<td>74.25</td>
<td>1920 x 1080</td>
<td>2640 x 1125</td>
<td>I</td>
</tr>
<tr>
<td>SMPTE 274M</td>
<td>1080p</td>
<td>24</td>
<td>74.25</td>
<td>1920 x 1080</td>
<td>2750 x 1125</td>
<td>J</td>
</tr>
<tr>
<td>SMPTE 274M</td>
<td>1080p</td>
<td>24/M</td>
<td>74.25/M</td>
<td>1920 x 1080</td>
<td>2750 x 1125</td>
<td>K</td>
</tr>
<tr>
<td>SMPTE 296M</td>
<td>720p</td>
<td>60</td>
<td>74.25</td>
<td>1280 x 720</td>
<td>1650 x 750</td>
<td>L</td>
</tr>
<tr>
<td>SMPTE 296M</td>
<td>720p</td>
<td>60/M</td>
<td>74.25/M</td>
<td>1280 x 720</td>
<td>1650 x 750</td>
<td>M</td>
</tr>
</tbody>
</table>
Color Space and 4:2:2 Sampling

Monochrome TV signals contain only intensity information, called luminance, or luma, and designated with the letter Y’ (the apostrophe indicates that the component is gamma corrected). When color information was added to the TV signal, the luma signal was left intact for compatibility with existing equipment, and two components of color information, called U’ and V’, were added. The two color components are often called color difference signals. The U’ component is the difference between blue and luma. The V’ component is the difference between red and luma.

Digital component video uses a modified form of the Y’ U’ V’ color space. This color space, called Y’ Cb’ Cr’, is a scaled and offset version of Y’ U’ V’. The Cb’ component is the blue color difference component, similar to the U’ component in Y’ U’ V’. Likewise, Cr’ is the red color difference, similar to V’. In 10-bit digital video, the Y’ component has a range of 64 to 940. The Cr’ and Cb’ components have ranges of 64 to 960. Values above and below the specified component ranges are reserved.

In 4:2:2 digital component video, the Y’ component is sampled at twice the rate as each of the chroma components Cb’ and Cr’. For example, if the basic sample rate is 74.25 MHz, then the Y’ component is sampled at a rate of 74.25 MHz. The Cb’ and Cr’ components are sampled at half this rate, or 37.125 MHz. Because there are two chroma components that are each sampled at half the sample rate, there is an average of one chroma word per sample. So, each sample of video consists of one word of Y’ and one chroma word, either Cb’ or Cr’. Consecutive samples alternate between having a Cb’ word and a Cr’ word.

In the HDTV digital component video standards, the luma channel and the chroma channel are treated separately. For 10-bit digital video, there is a 10-bit luma channel and a 10-bit chroma channel. In Figure 17-1, notice how consecutive samples alternate between having a Cb’ word and a Cr’ word on the chroma channel.

### Table 17-1: Common HD 4:2:2 Component Digital Video Standards (Continued)

<table>
<thead>
<tr>
<th>SMPTE Standard</th>
<th>Format</th>
<th>Frame Rate (Hz)</th>
<th>Sample Rate (MHz)</th>
<th>Active Samples/Line &amp; Active Lines/Frame (words x lines)</th>
<th>Total Samples/Line &amp; Total Lines/Frame (words x lines)</th>
<th>Format Designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMPTE 296M</td>
<td>720p</td>
<td>50</td>
<td>74.25</td>
<td>1280 x 720</td>
<td>1980 x 750</td>
<td></td>
</tr>
<tr>
<td>SMPTE RP 211</td>
<td>1080sF</td>
<td>30</td>
<td>74.25</td>
<td>1920 x 1080</td>
<td>2200 x 1125</td>
<td>(G)</td>
</tr>
<tr>
<td>SMPTE RP 211</td>
<td>1080sF</td>
<td>30/M</td>
<td>74.25/M</td>
<td>1920 x 1080</td>
<td>2200 x 1125</td>
<td>(H)</td>
</tr>
<tr>
<td>SMPTE RP 211</td>
<td>1080sF</td>
<td>25</td>
<td>74.25</td>
<td>1920 x 1080</td>
<td>2640 x 1125</td>
<td>(I)</td>
</tr>
<tr>
<td>SMPTE RP 211</td>
<td>1080sF</td>
<td>24</td>
<td>74.25</td>
<td>1920 x 1080</td>
<td>2750 x 1125</td>
<td>(J)</td>
</tr>
<tr>
<td>SMPTE RP 211</td>
<td>1080sF</td>
<td>24/M</td>
<td>74.25/M</td>
<td>1920 x 1080</td>
<td>2750 x 1125</td>
<td>(K)</td>
</tr>
</tbody>
</table>
Timing Reference Signals

Each line of HD digital component video is formatted as shown in Figure 17-1. A line is divided into a horizontal blanking interval and the active video portion. The end-of-active video (EAV) and start-of-active video (SAV) timing reference signals serve to mark the transitions between the active and blanking portions of the line. Each EAV and SAV is four words long. The first three words are always 3FFh, 000h, and 000h in that order. The fourth word is called the XYZ word and contains three timing flags used to indicate which field the line is in, whether the line is an active video line or is in the vertical blanking interval, and whether this is an EAV or an SAV. Figure 17-2 shows the format of the XYZ word.

![Figure 17-1: HDTV Video Line Format](image1)

![Figure 17-2: XYZ Word Format](image2)

The timing reference signals play a key role in the HD-SDI protocol. The leading sequence of 3FFh, 000h, and 000h is unique in the video stream. This sequence can only occur legally at the beginning of a timing reference. HD-SDI receivers use this sequence to synchronize to the HD-SDI bit stream.
Video Test Patterns

Many different video test patterns have been developed to test different aspects of video transmission, reception, and display. This chapter focuses on three video test patterns commonly used in the broadcast studio, often in conjunction with the HD-SDI standard. The three test patterns are: SMPTE RP 219-2002 color bars, 75% color bars, and the SMPTE RP 198-1998 HD-SDI checkfield.

SMPTE RP 219-2002 Color Bar Pattern

SMPTE recommended practice RP 219-2002 specifies a color bar pattern that is compatible with both standard- and high-definition equipment. It is similar to the older standard-definition SMPTE EG-1 color bar pattern, but has been updated to include a 16:9 aspect ratio pattern and some new features. Figure 17-3 shows this test pattern.

The SMPTE RP 219-2002 document defines the sizes of the various areas in the color bar pattern and also explicitly specifies the digital luma and chroma values for each of the colors. Consult this document for more information about the pattern.

One of the features of this color bar pattern is the Y-Ramp pattern. This pattern ramps linearly from 0% luma (black) to 100% luma (white) from left to right across the pattern. Figure 17-3 shows the 16:9 aspect ratio version of this pattern. The 4:3 aspect ratio version leaves off the two columns (40% gray in the top portion) on either side of the pattern.

75% Color Bars Pattern

The top row of SMPTE RP 219-2002 is a 75% color bar pattern. Sometimes it is useful to remove the lower portions of the SMPTE RP 219-2002 pattern and have the 75% color bars pattern occupy the full height of the frame. In particular, the SMPTE 292M HD-SDI spec requires that the output jitter of an HD-SDI transmitter should only be measured with 75% color bars. Figure 17-4 shows this test pattern.
When -I is selected for the upper user option area, +Q must be displayed in the lower user option area and when +Q is selected for the lower user option area, -I must be displayed in the upper user option area.

**Figure 17-3:** SMPTE RP 219-2002 Color Bar Pattern (16:9 Aspect Ratio)

**Figure 17-4:** 75% Color Bars Pattern (16:9 Aspect Ratio)
HD-SDI Digital Checkfield Pattern

SMPTE recommended practice RP 198-1998 defines a special pattern designed to generate the HD-SDI pathological waveforms. There are two different pathological waveforms. One pattern stresses the HD-SDI receiver’s cable equalizer and the other stresses the receiver’s PLL. The two pathological waveforms are shown in Figure 17-5.

The equalizer pathological waveform is poorly DC balanced. It consists of a single High bit followed by 19 Low bits. This basic pattern is repeated continuously across a video line. Note that the opposite polarity of this waveform, a single Low bit followed by 19 High bits, is also possible and both polarities are generated by the RP 198-1998 checkfield.

The PLL pathological waveform is a square wave with 20 Low bits followed by 20 High bits. This basic pattern is repeated continuously across a video line. This pattern is much lower in frequency than the typical bit patterns seen by the receiver’s PLL. Poorly designed PLLs might not stay locked during this low-frequency waveform.

It is important to distinguish between the RP 198-1998 checkfield and the pathological waveforms. The pathological waveforms are serial bit patterns that are output by an HD-SDI transmitter after encoding and serialization. The checkfield is a video pattern that, when encoded by an HD-SDI transmitter, produces the pathological waveforms.

The HD-SDI encoding process is based on a pseudo-random scrambling algorithm. The encoding of each sample varies depending on the current state of the scrambler. The state of the scrambler is dependent upon the previous history of the data that has been encoded by the scrambler. Therefore, it is not possible to force the HD-SDI encoder to generate either pathological waveform “on demand”. The RP 198-1998 checkfield causes an HD-SDI
encoder to produce the pathological waveforms randomly with a certain statistical rate of occurrence.

The RP 198-1998 checkfield is shown in Figure 17-6. In the top half of each field or frame, the checkfield consists of repeatedly sending a luma value of 300h and a chroma value of 198h into the encoder to cause the it to occasionally generate the equalizer pathological waveform. These values can be swapped with the chroma channel set to 300h and the luma channel set to 198h; the order doesn’t matter. While encoding half a field or frame of this pattern, the equalizer pathological pattern is usually produced on several video lines in the frame.

In order to insure that both polarities of the equalizer pathological waveform are produced by the encoder, the 198h value in the first sample of the first active picture line in the first field of every other frame is changed to a value of 190h. For progressive scan formats, the sample that is changed is the first sample of the first active line of every other frame.

The bottom half of each field or frame in the checkfield consists of the pattern 200h, 110h. As with the equalizer pattern, it doesn’t matter which value is assigned to which channel. This sequence causes the HD-SDI encoder to produce the PLL pathological waveform during several lines of the frame (on average).

**Reference Design**

**Introduction**

The HD video pattern generator reference design can produce three different video patterns: SMPTE RP 219-2002 color bars, 75% color bars, and the SMPTE RP 198-1998 checkfield. In all cases, the pattern generator produces patterns with 16:9 aspect ratios.

The reference design can produce the video patterns for all of the video formats supported by HD-SDI (all of the formats listed in Table 17-1). The video pattern generator actually only has to produce 9 different video formats as listed in Table 17-2 because some of these video formats can be combined. For example, the only difference between format D (1080i 30 Hz) and E (1080i 30/M Hz) is the clock rate. The pattern generator produces exactly the same data for both of these formats and the frequency of the video clock must be switched externally to the video pattern generator. Similarly, with a static image, such as these video test patterns, the data that the video pattern generator produces for the "look-alike" segmented frame and interlaced formats is identical. So, for example, the video pattern generator produces the same data for both the 1080i 30 Hz format and the 1080sF 30 Hz format.
The video pattern generator reference design can produce all nine of the video formats listed in Table 17-2. However, it can only be loaded with eight of these formats at any one time. By default, the reference design includes neither the SMPTE 260M 1035i 30 Hz format (Table 17-2, group 0) nor the SMPTE 295M 1080i 25 Hz format (Table 17-2, group 1), because they are rarely used. However, it is a simple matter to change the reference design to include any set of eight video formats shown in Table 17-2.

Table 17-2: Video Format Groups Supported by Video Pattern Generator

<table>
<thead>
<tr>
<th>Group Number</th>
<th>Supported Video Formats (SMPTE 292M Format Designator Shown in Parenthesis)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Progressive or Interlaced Formats</td>
</tr>
<tr>
<td></td>
<td>74.25 MHz</td>
</tr>
<tr>
<td>0</td>
<td>1035i 30 Hz (A)</td>
</tr>
<tr>
<td>1</td>
<td>1080i 25 Hz (C)</td>
</tr>
<tr>
<td>2</td>
<td>1080i 30 Hz (D)</td>
</tr>
<tr>
<td>3</td>
<td>1080i 25 Hz (F)</td>
</tr>
<tr>
<td>4</td>
<td>1080p 30 Hz (G)</td>
</tr>
<tr>
<td>5</td>
<td>1080p 25 Hz (I)</td>
</tr>
<tr>
<td>6</td>
<td>1080p 24 Hz (J)</td>
</tr>
<tr>
<td>7</td>
<td>720p 60 Hz (L)</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Figure 17-7 is a block diagram of the reference design video pattern generator. The top level design files for this reference design are named multigenHD.v and multigenHD.vhd. This design is built around three 18 kb block RAMs. The block RAMs are used as ROMs and are initialized during FPGA configuration.
The video pattern generator consists of three main sections: the horizontal sequencer, the vertical sequencer, and the output generator. The horizontal sequencer keeps track of the current horizontal position and generates a horizontal region code indicating to the output section which horizontal region of the video pattern is currently being drawn. Likewise, the vertical sequencer keeps track of the current vertical position and generates a vertical band code indicating to the output section which vertical band of the video pattern is currently being drawn. The output section converts the horizontal and vertical coordinates of the video pattern into the luma and chroma video components appropriate for that portion of the pattern.

**Figure 17-7:  Video Pattern Generator Block Diagram**

The video pattern generator consists of three main sections: the horizontal sequencer, the vertical sequencer, and the output generator. The horizontal sequencer keeps track of the current horizontal position and generates a horizontal region code indicating to the output section which horizontal region of the video pattern is currently being drawn. Likewise, the vertical sequencer keeps track of the current vertical position and generates a vertical band code indicating to the output section which vertical band of the video pattern is currently being drawn. The output section converts the horizontal and vertical coordinates of the video pattern into the luma and chroma video components appropriate for that portion of the pattern.

**Horizontal Section**

One block RAM (the HROM) is used as part of the horizontal sequencer. A horizontal counter increments every clock cycle, keeping track of the current horizontal sample count. The HROM outputs a next event value indicating the horizontal sample number when the sequencer should be advanced to the next region of the pattern. A transition from one horizontal region to another takes place whenever the video output value needs to change, such as when moving from one color bar to the next. Separate regions exist for the EAV, SAV, and horizontal blanking portions of the video pattern, as well. The LSB of the horizontal counter is not used in the comparison with the horizontal next event. A
transition from one horizontal region to the next can only occur when the LSB of the horizontal counter is High.

The HROM implements a finite state machine (FSM). The current state register of the FSM is the block RAM’s internal input register. The next region output of the HROM is the next state of the FSM. This value wraps back to the address inputs of the HROM and is loaded into the current state register the next time the register is loaded. This only occurs when the horizontal counter matches the next event value from the HROM.

The HROM also outputs a value called h_region. This value indicates to the output section which region of the video pattern is currently active, horizontally. The h_region output from the HROM is modified in some cases to accommodate pattern modifications that the HROM can’t handle, such as for the user option fields of RP 219-2002.

The HROM also outputs several other control signals. The h_clr output of the HROM causes the horizontal counter to clear to zero when the end of the line is reached. The v_inc output causes the vertical counter in the vertical section to increment once per line.

**Vertical Section**

The vertical section is very similar to the horizontal section. Central to the vertical section is another block RAM-based FSM called the VROM. A vertical counter keeps track of the current line number. The counter is compared to a vertical next event value from the VROM to determine when to advance the state of the FSM. As with the HROM, the VROM outputs a next state value that is fed back to the VROM address inputs. Whenever the vertical counter matches the next event value from the VROM, the next state value is loaded into the current state register in the VROM.

The VROM outputs a vertical band code that indicates which vertical "band" of the video pattern is currently active. Different bands are used for the different vertical sections of the pattern. For example, the RP 219-2002 pattern has four vertical bands: the 75% color bars on top, the two narrow rows in the middle, and the bottom mostly black and white band. Additional vertical regions are used to indicate the vertical blanking intervals.

The VROM actually outputs two vertical band codes, one for the 75% color bars and the other for all other video patterns. A MUX selects between these two codes based on the pattern selection inputs.

**Output Section**

The output section of the video pattern generator uses another block RAM (the CROM) to generate the luma and chroma values. The address inputs to the CROM are the vertical band (v_band) from the vertical section, the horizontal region (h_region) from the horizontal FSM, and the LSB of the horizontal counter. The vertical band and the horizontal region uniquely identify the current rectangular block of the video pattern. The CROM outputs the luma and chroma components that correspond to this portion of the video pattern. Because 4:2:2 digital component video is being generated, consecutive samples of the chroma channel must alternate between containing a Cb’ word or a Cr’ word. The LSB of the horizontal counter tells the CROM which chroma word to output.

The CROM does not have the flexibility to generate the Y-Ramp portion of the RP 219-2002 color bar pattern so a Y-Ramp generator is added on the output of the CROM. Figure 17-8 shows the details of the Y-Ramp generator. The Y-Ramp generator is an accumulator with an output rounder. The accumulation register is initialized with an initial value called YRAMP_INIT when the CROM asserts its y_ramp_reload output. This occurs at the beginning of the Y-Ramp pattern. For each video sample after that, an increment value is
added to the current value of the accumulation register and the sum is loaded back into the accumulator register. The increment value is different for the video formats with 1280 active samples versus those with 1920 active samples.

To make the Y-Ramp pattern as linear as possible, the accumulation register and the increment value both have fractional bits. The output of the accumulation register is rounded to the nearest integer value before being output from the video pattern generator.

A MUX controls whether the luma output of the CROM or the rounded output of the Y-Ramp generator is used as the luma output. The MUX is controlled by the y_ramp_en output of the CROM so that the Y-Ramp generator output is only used during the Y-Ramp portion of the video pattern.

**Other Outputs**

The video pattern generator outputs several video timing signals:

- **trs**: Asserted when the four-word timing reference sequence is output by the video pattern generator.
- **xyz**: Asserted when the XYZ word of the timing reference sequence is output by the video pattern generator.
- **field**: Indicates whether the current field is the first (field = Low) or second (field = High) field. This signal is always Low for progressive formats.
- **v_blank**: Asserted during the vertical blanking interval.
- **h_blank**: Asserted during the horizontal blanking interval.

**Figure 17-8: Y-Ramp Generator**

![Diagram of Y-Ramp Generator](image-url)
The video pattern generator also outputs an 11-bit line_num value indicating the current video line number. HD-SDI requires that the line number be inserted into the two words that immediately follow EAV timing reference sequences. The video pattern generator does not insert the line number because this is only required when transmitting the video over HD-SDI. The line number is output from the video pattern generator so that logic in the HD-SDI transmitter can insert it into the video stream.

**Initialization File Generation**

The video pattern generator is a relatively simply design because all of the complexity of the video patterns is encoded into the block RAMs. The creation of these initialization files for the block RAMs is where all of the complexity occurs.

The initialization values for the three block RAMs are created by a Verilog file called `multigenHD_romgen.v`. This file, when compiled and run using a Verilog simulator, creates seven different initialization files for each block RAM:

- Verilog simulation init file (`xxxx_sim.v`)
- Verilog XST synthesis file (`xxxx_xst.v`)
- Verilog Synplify synthesis file (`xxxx_syn.v`)
- VHDL simulation init file (`xxxx_sim.vhd`)
- VHDL XST synthesis file (`xxxx_xst.vhd`)
- VHDL Synplify synthesis file (`xxxx_syn.vhd`)
- Verilog 2001 init file (`xxxx_v2001.v`)

While there is no VHDL version of the `multigenHD_romgen.v` file, this Verilog file creates both Verilog and VHDL initialization files. These files can be inserted into the appropriate places in the video pattern generator design files. The Verilog 2001 init file uses the Verilog 2001 parameter-passing syntax. Because most synthesis tools, including XST, and most simulators can use this syntax to define the default contents of the block RAMs, this has become a universal way to initialize the block RAMs. The Verilog 2001 file can take the place of the synthesis and simulation Verilog files.

The `multigenHD_romgen.v` file can produce initialization files for any subset of the nine different format groups listed in Table 17-2. The subset included in the initialization files can be easily changed and the mapping from the 3-bit standard input code to the selected video format can also be modified simply by changing the `convert_std` function at the end of the `multigenHD_romgen.v` file. This function consists of a simple case statement that maps the 3-bit standard code into a 4-bit video format code used by the ROM generation routines to create the initialization files.

To generate the initialization files, compile and load the `multigenHD_romgen.v` file into a Verilog simulator and then run the simulation to completion. The simulation automatically stops when the files are completed. The `multigenHD_romgen` file has been tested with the current version of ModelSim.

**Design Size**

Table 17-3 lists the FPGA resources required for the multigenHD video pattern generator. These results were obtained using XST and ISE 6.1. This reference design has been simulated using ModelSim. It has also been tested in hardware using the Xilinx SDV demo board [Ref 2].
Chapter 17: HDTV Video Pattern Generator

### Other Considerations

The block RAMs that are used as the three ROMs in this design are dual-ported. In the reference design, only one of the ports is used. However, it is possible to use the second port of each ROM to create a second, independent video pattern generator. The other logic in the video pattern generator must be duplicated for the second pattern generator, but using the same block RAMs for both pattern generators does save FPGA resources in applications that require two video pattern generators. The size of a dual video pattern generator can be estimated by simply doubling the LUT and flip-flop counts listed in Table 17-3.

The output values of the video pattern generator are not bandwidth limited—the rise and fall times are not shaped. When these patterns are displayed on an analog display, the sharp rise and fall times at the transitions between bars can cause ringing of the video image at these transitions. RP 219-2002 recommends that the rise and fall times be limited to the equivalent of 55 ns from 10% to 90% for all three components. If necessary, this can be done in the FPGA by filtering the three components with a digital filter. Note, however, that the RP 198-1998 checkfield should never be filtered. The values of this checkfield must enter the HD-SDI encoder exactly as they come from the video pattern generator.

The block RAMs in Virtex-5 devices are different than in previous-generation parts. The example Verilog files included in the reference design can be synthesized for either Virtex-5 RAMB18SDP primitives or for the older RAMB16_S36 primitives. A `define statement in the Verilog files indicates if Virtex-5 is the target device and uses the RAMB18SDP primitive if this is the case. Two sets of VHDL example files are included in the reference design, one set for Virtex-5 RAMB18SDP primitives and another set for the older RAMB16_S36 primitives.

### Conclusions

This chapter describes a reference design capable of generating common HDTV video test patterns for all 18 video formats supported by the HD-SDI standard. The reference design can produce the SMPTE RP 219-2002 color bar pattern along with two test patterns commonly used to test HD-SDI interface performance.

The reference design is based on three 18 kb block RAMs, allowing the video patterns and supported video formats to be changed by changing the block RAM initialization files. A Verilog file is used to create the block RAM initialization files. The use of block RAMs as the core of the video test pattern generator results in a small, efficient implementation.

### Design Files

The reference design files are available on the Xilinx website at:

www.xilinx.com/bvdocs/appnotes/xapp514.zip

Open the ZIP archive and extract file xapp514_hdtv-pattern-gen.zip.

---

**Table 17-3: Reference Design Results**

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>FFs</th>
<th>Block RAMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>multigenHD</td>
<td>119</td>
<td>100</td>
<td>3</td>
</tr>
</tbody>
</table>

---

www.BDTIC.com/XILINX
Section VI: Digital Audio

Audio/Video Connectivity Solutions for the Broadcast Industry
Chapter 18

Introduction to Digital Audio for Video Broadcasting

Audio and video are equally important elements of the programming content in the broadcast industry. At different places in the broadcast studio or production center, audio and video are combined into one digital signal. At other places, audio is transported and processed separately from video. Thus, there are standards for transporting digital audio separately from video and other standards for transporting digital audio embedded in the digital video signal.

The AES3 standard from the Audio Engineering Society is the professional digital audio transport standard. Each AES3 digital signal carries a pair of digital audio signals over twisted pair or coaxial cable.

Various SMPTE standards describe how to "embed" digital audio into the unused horizontal blanking intervals of digital video signals. These standards are based on the digital audio format specified by AES3.

Introduction to the AES3 Digital Audio Standard

AES3 is a professional standard for transporting digital audio serially over twisted pair or coaxial cable. Each AES3 audio link carries a stereo pair of digital audio channels and supports various audio sampling rates. AES3 is also called AES/EBU (Audio Engineering Society / European Broadcasting Union).

The consumer version of AES3 is called S/PDIF (Sony/Philips Digital Interface). S/PDIF is commonly used to move digital audio between pieces of consumer electronic equipment such as between a DVD player and a surround-sound receiver. The data format and data rates of S/PDIF are the same as AES3. S/PDIF differs from AES3 in the electrical and physical (cable and connector) specifications.

AES3 is defined by the AES3-2003 document from the Audio Engineering Society [Ref 1]. The nearly identical specification from EBU (Tech. 3250-E) describes AES/EBU [Ref 2].

Data Format

An AES3 interface carries two linear pulse coded modulation (PCM) audio channels, interleaved together in one serial bitstream. Additional standards from The Society of Motion Picture and Television Engineers (SMPTE) describe how to transport non-PCM multi-channel audio (surround sound) on AES3 interfaces. S/PDIF links commonly carry surround-sound audio. However, none of these multi-channel formats are officially part of the AES3 specification. The focus of this chapter and the reference designs in this section is the standard two-channel PCM audio described in the AES3 specification.
Subframes, Frames, and Blocks

The basic data structure of AES3 is called a subframe. Each 32-bit subframe carries a single audio sample word for one audio channel along with a few other bits of information (Figure 18-1). A subframe begins with a 4-bit preamble. The audio word can be either 24 bits or 20 bits. Following the audio word, are the valid bit (V), user data bit (U), channel status bit (C), and parity bit (P). Two consecutive subframes, one for each of the two audio channels form a complete frame. The subframe for channel 1 is always sent before the subframe for channel 2.

Figure 18-1 shows the bit order of the subframe from left to right as LSB to MSB. This corresponds to the order in which the bits are sent on the AES3 interfaces (LSB first).

Frames are grouped together in blocks of 192 frames. This grouping of frames into blocks serves to define the beginning and ending points for the sequence of channel status and user data bits. The channel status information for each channel is 192 bits long. With one channel status bit for each channel included in each frame, it takes 192 frames to transmit the 192 bits of channel status for each channel. Likewise, the user data for each channel is 192 bits long, with one user data bit for each channel present in each frame. The first frame of a block contains the LSBs of the channel status and user data for each channel. The MSBs are located in the last frame of the block.

Preambles

Preambles are unique sequences that do not occur anywhere except in the preamble portion of each subframe. Because preambles are unique, an AES3 receiver can find them and synchronize to the AES3 bitstream. Because there is a preamble at the beginning of each subframe, an AES3 receiver can quickly synchronize to the bitstream.

The first frame of a block is identified by a different preamble in the channel 1 subframe. In all but the first frame of a block, subframe 1 begins with an X preamble. In the first frame of a block, subframe 1 begins with a Z preamble instead of an X preamble. By detecting the Z preamble, an AES3 receiver can locate the LSB of the channel status and user data and thereby organize this information in the correct order. Every subframe 2 begins with a Y preamble.
Thus, the preambles serve three purposes:

- Unique preamble sequences allow the receiver to identify the beginning of each subframe.
- Different preambles distinguish subframe 1 from subframe 2 in each frame.
- The Z preamble, occurring only in subframe 1 of the first frame of a block, allows the receiver to locate the LSB of the channel status and user data.

**Biphase-Mark Encoding**

AES3 uses biphase-mark encoding. In this encoding scheme, each bit is encoded into a symbol consisting of two states. The first state of a symbol is the inverse of the second state of the previously transmitted symbol. For example, if the second state of the previous symbol was a 1, then the first state of next symbol is 0. This ensures that there always is a state transition between symbols. The second state of a symbol is the same as the first state if the bit being encoded is a 0. If the bit being encoded is a 1, then the second state of the symbol is the inverse of the first state.

The preambles at the beginning of each subframe are not biphase-mark encoded. In fact, they violate the rules of biphase-mark encoding, making them easily identifiable in the bitstream. Each preamble consists of eight consecutive states—the equivalent of four bit times.

**Valid Bit**

The valid bit in each subframe indicates whether the audio sample of the subframe is valid. If the valid bit is 0, then the sample is valid. If the valid bit is 1, the sample is invalid. So, for example, if only one channel of audio is being carried by an AES3 bitstream, the valid bits in all subframes of the second audio channel would be set to 1 to indicate that this audio channel is not valid.

**Channel Status**

The channel status information carries sideband data about each audio channel. The type of information carried by the channel status includes the audio sample rate and the number of bits in the audio sample word. The AES3-2003 document breaks the 192 bits of channel status information down into 24 bytes of 8 bits each and defines the purpose of each bit. Not every AES3 bitstream uses all 192-channel status bits and the AES3-2003 document identifies several subsets that are commonly used.

The last byte of channel status information is commonly used to hold a Cyclic Redundancy Check (CRC) word calculated from the other channel status bits. However, the CRC is not present in all AES3 bitstreams and should be filled with zeros if it is not used.

**User Data**

User data is similar to channel status data. The user data consists of 192 bits for each audio channel. User data can carry any type of information. The AES18-1996 document defines a standard way to create and insert data packets into the user data bits of an AES3 bitstream [Ref 3].

**Parity Bit**

Each subframe ends with a parity bit. This even parity bit is generated so that the 28 bits of the subframe, not including the preamble, have an even number of ones and zeros.
Data Rate

The data rate of an AES3 interface is dependent upon the audio sampling rate being carried. In theory, the audio can be sampled at any rate so the resulting AES3 bit rate can be anything. However, the AES5-2003 document [Ref 4] defines the standard audio sampling rates for AES3 audio as 32 kHz, 48 kHz, and 96 kHz. The audio sampling rate used on audio CDs is 44.1 kHz, so this sampling rate is also often carried on AES3 interfaces as are some multiples of 44.1 kHz, such as 88.2 kHz and 176.4 kHz. Also, some high-end applications are starting to use 192 kHz as the sampling rate. Table 18-1 shows the AES3 bit rates corresponding to various common audio sampling frequencies. The symbol rate of the AES3 signal is actually twice as fast as the bit rate because each bit is encoded as two states.

The AES3 spec also defines a single-channel double-sampling frequency mode. In this mode, the two subframes in a frame carry consecutive samples for one channel instead of samples from two different channels. This allows the sampling rate to double while maintaining the same AES3 bit rates.

<table>
<thead>
<tr>
<th>Audio Sampling Rate</th>
<th>AES3 Bit Rate(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 kHz</td>
<td>2.048 Mb/s</td>
</tr>
<tr>
<td>44.1 kHz</td>
<td>2.8224 Mb/s</td>
</tr>
<tr>
<td>48 kHz</td>
<td>3.072 Mb/s</td>
</tr>
<tr>
<td>96 kHz</td>
<td>6.144 Mb/s</td>
</tr>
<tr>
<td>192 kHz</td>
<td>12.288 Mb/s</td>
</tr>
<tr>
<td>96 kHz(2)</td>
<td>3.072 Mb/s</td>
</tr>
</tbody>
</table>

Notes:
1. The AES3 serial symbol rate is 2X the bit rate.
2. This entry shows the use of single channel double sampling frequency mode for 96 kHz.

SMPTE 272M: Embedded Digital Audio for SD-SDI

Digital video streams often carry non-video ancillary data embedded in the horizontal and vertical blanking intervals. One of the most common types of ancillary data is digital audio. Multiple channels of digital audio can be embedded in the horizontal blanking intervals of digital video signals carried by SD-SDI.

The SMPTE 291M specification describes the generic format of ancillary data packets for SD-SDI. SMPTE 272M specifies how AES3 digital audio is mapped to these ancillary data packets and embedded into SD-SDI video streams.

The SMPTE 291M standard allows 16 audio channels per video stream. These channels are divided into four audio groups, numbered 1 through 4. Audio channels 1 through 4 are assigned to audio group 1, channels 5 through 8 to audio group 2, channels 9 through 12 to audio group 3, and channels 13 through 16 to audio group 4. The four channels in each audio group are also grouped into two channel pairs. Each channel pair consists of two audio channels, usually a stereo pair derived from the same audio source. An AES3 signal carries two audio channels, and these two channels are typically mapped into one channel pair of an audio group when embedded in a video stream.
In the video broadcast industry, the standard audio sample rate is 48 kHz. SMPTE 272M also supports embedded audio sample rates of 44.1 kHz and 32 kHz.

SMPTE 272M supports both 20-bit and 24-bit audio samples. Some video equipment might support only 20-bit audio. SMPTE 272M embedded audio packets are designed so that 24-bit embedded audio is usable by equipment supporting only 20-bit audio.

SMPTE 272M also permits embedded audio that is not compliant with the AES3 specification, called *non-pulse-coded modulation* (non-PCM) data. SMPTE 337M describes how non-PCM data is mapped into the same data structures as defined by AES3, allowing it to be embedded in SD-SDI video streams. Non-PCM data is usually compressed multi-channel (surround sound) audio.

### SD Embedded Audio Packets

Three types of ancillary data packets are used for SMPTE 272M embedded audio: audio data packets, extended data packets, and audio control packets. For each embedded audio packet type, four different data identification (DID) values are defined, with a unique DID assigned to each audio group. Thus, the DID word identifies both the packet type and the audio group of the packet. Table 18-2 shows the DID values assigned to the various embedded audio ancillary data packet types.

<table>
<thead>
<tr>
<th></th>
<th>Audio Data Packets</th>
<th>Extended Data Packets</th>
<th>Audio Control Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>0x2FF</td>
<td>0x1FE</td>
<td>0x1EF</td>
</tr>
<tr>
<td>Group 2</td>
<td>0x1FD</td>
<td>0x2FC</td>
<td>0x2EE</td>
</tr>
<tr>
<td>Group 3</td>
<td>0x1FB</td>
<td>0x2FA</td>
<td>0x2ED</td>
</tr>
<tr>
<td>Group 4</td>
<td>0x2F9</td>
<td>0x1FB</td>
<td>0x1EC</td>
</tr>
</tbody>
</table>

### SD Audio Data Packets

An audio data packet contains audio samples for a single audio group. Audio samples from different audio groups cannot be mixed together in the same audio data packet.

Figure 18-2 shows the format of the audio data packet. The audio data packet is variable length. It can contain as many audio samples as fit, given the length limit of ancillary data packets (255 data words) and the space available in the horizontal ancillary data space (HANC). Other ancillary data packets, including audio data packets for other audio groups, might also be present in the same HANC, further limiting the size of an audio data packet. However, audio data packets are usually much smaller than the size of the HANC. With 48 kHz audio, an audio data packet commonly has three or four audio samples for each active channel in the group.

Each audio sample occupies three consecutive words in the audio data packet. Samples for both channels of a channel pair must be present in the packet, even if only one channel of the pair is active. The 2004 version of the SMPTE 272M standard recommends that both channel pairs of a group always be included in the audio data packet, but this is not strictly required. It is legal for an audio data packet to contain samples for only one channel pair.

The SMPTE 272M standard specifies that the two audio samples from the two channels of a channel pair must always be paired together and appear consecutively in the audio data packet. The sample from lower numbered channel of the channel pair must be first,
followed immediately by a sample from the higher numbered channel. For example, channels 1 and 2 are in channel pair 1 of audio group 1. In the audio data packet, a sample for channel 1 must be followed immediately by a sample for channel 2.

The SMPTE 272M-2004 revision of the standard recommends that within an audio data packet the channel pair containing the lower numbered channels precede the channel pair containing the higher numbered channels. For example, in an audio data packet for audio group 1, channels 1 and 2 should precede channels 3 and 4, but this is not a strict requirement.

The three words of an audio sample contain 20 bits of audio data plus additional information including a channel code, Z, V, U, and C bits, and a parity bit.

The 2-bit channel code indicates the audio sample’s channel number within the audio group. For example, if the audio data packet is from audio group 1, channel number values of 00, 01, 10, and 10 indicate audio channels 1, 2, 3, and 4 respectively. If the audio data packet is from audio group 2, channel numbers 00, 01, 10, and 10 indicate audio channels 5, 6, 7, and 8.

The C (channel status), U (user data), and V (valid) bits in the audio sample are identical to the AES3 C, U, and V bits. The V bit indicates whether the channel is valid (0 = valid, 1 = invalid). The channel status and user data are each organized into 192-bit data structures sent one bit per audio sample. In order to correctly interpret the C and U data, the receiver must know where the 192-bit block begins and ends. The Z bit is only High in the first audio sample of each 192-bit block, indicating that that the sample contains the LSB of the 192-bit C and U data. Typically, the Z bits of two audio samples in a channel pair are the same because AES3 requires this of the two channels carried on one AES3 interface. However, SMPTE 272M does allow the Z bits for the two channels of a channel pair to be set independently, meaning that the 192-sample blocks in the two channels in a channel pair are not necessarily aligned in the two channels of the pair.

Figure 18-2: SD Audio Data Packet Format

The SMPTE 272M-2004 revision of the standard recommends that within an audio data packet the channel pair containing the lower numbered channels precede the channel pair containing the higher numbered channels. For example, in an audio data packet for audio group 1, channels 1 and 2 should precede channels 3 and 4, but this is not a strict requirement.

The three words of an audio sample contain 20 bits of audio data plus additional information including a channel code, Z, V, U, and C bits, and a parity bit.

The 2-bit channel code indicates the audio sample’s channel number within the audio group. For example, if the audio data packet is from audio group 1, channel number values of 00, 01, 10, and 10 indicate audio channels 1, 2, 3, and 4 respectively. If the audio data packet is from audio group 2, channel numbers 00, 01, 10, and 10 indicate audio channels 5, 6, 7, and 8.

The C (channel status), U (user data), and V (valid) bits in the audio sample are identical to the AES3 C, U, and V bits. The V bit indicates whether the channel is valid (0 = valid, 1 = invalid). The channel status and user data are each organized into 192-bit data structures sent one bit per audio sample. In order to correctly interpret the C and U data, the receiver must know where the 192-bit block begins and ends. The Z bit is only High in the first audio sample of each 192-bit block, indicating that that the sample contains the LSB of the 192-bit C and U data. Typically, the Z bits of two audio samples in a channel pair are the same because AES3 requires this of the two channels carried on one AES3 interface. However, SMPTE 272M does allow the Z bits for the two channels of a channel pair to be set independently, meaning that the 192-sample blocks in the two channels in a channel pair are not necessarily aligned in the two channels of the pair.
SD Extended Data Packets

Audio data packets only provide 20 bits of audio data per sample. For 24-bit audio, the additional four bits of each audio sample (called the auxiliary or extended data) are embedded in a separate ancillary data packet type called the extended data packet. There are different extended data packet DID values for each audio group, so an extended data packet only carries the extended data for one audio group.

The four bits of extended data are the least significant four bits of the 24-bit audio sample. The most significant 20 bits are in the audio data packet. If the receiving equipment doesn't support 24-bit audio, it simply uses the 20 bits in the audio data packets and ignores the extended data packets, throwing away the least significant four bits of each audio sample.

When extended data packets are present, they are always paired with their associated audio data packet. An extended data packet immediately follows its associated audio data packet. If an extended data packet does not immediately follow an audio data packet, 20-bit audio is assumed.

In an extended data packet, each data word contains the extended data for both audio samples of a channel pair (Figure 18-3). The first data word of an extended data packet contains the extended data for the first two audio samples of the preceding audio data packet. The second data word of the extended data packet carries the extended data for the next two audio samples of the preceding audio data packet, and so on. A bit in each extended data packet data word indicates whether the data word is for the first channel pair of the audio group or the second. Because the order of the data words in the extended data packet exactly matches the order of the audio samples in the audio data packet, this bit primarily serves as to check that the ordering of extended data is correct.

![SD Extended Data Packet Format](image-url)
SD Audio Control Packets

Audio control packets carry additional information about the audio stream. Audio control packets are sent once per video field, always in the second HANC after the synchronous switching interval. Audio data packets and extended data packets are typically not sent in the HANC of this line.

As with the other embedded audio packet types, SMPTE 272M reserves four different DID values for audio control packets, one for each of the four audio groups. Thus, an audio control packet is specific to an audio group. If all four audio groups are present in the video stream, then four audio control packets, one for each audio group, are present in the HANC of the specified line.

Audio control packets are optional when the embedded audio conforms to the SMPTE 272M default 48 kHz synchronous audio. Otherwise, audio control packets are required.

The audio control packet is fixed in length. All data words in the packet have a defined meaning (Figure 18-4). Four types of information are contained in the audio control packet: audio frame number, audio sample rate, active channel indication, and audio delay.

Figure 18-4: SD Audio Control Packet Format
Audio Frame Number

The first word in the audio control packet payload contains the audio frame number for the first channel pair of the audio group. The second word contains the audio frame number for the second channel pair of the audio group. The audio frame numbers provide information about how many audio samples are present in video frame. For example, with NTSC video and 48 kHz audio, 8,008 audio samples must be sent every five video frames. Because 8,008 is not evenly divisible by 5, some video frames must carry more audio samples than other frames. SMPTE 272M strictly defines how many audio samples are to be carried in each video frame in the sequence of five frames. Frames 1, 3, and 5 of the five frame sequence must have exactly 1602 audio samples. Frames 2 and 4 must have 1601 audio samples. The audio frame number indicates the position of the current frame in the five frame sequence. A frame number of 0 indicates that the audio frame number is not used.

For different video frame rates and audio sample rates, different frame sequences are required. For example, with 44.1 kHz audio and NTSC video, 147,147 audio samples must be transmitted in 100 video frames. The audio frame sequence count, in this case, would range from 1 to 100.

Audio Sample Rate

The third data word of an audio control packet indicates the audio sample rates of both channel pairs. Three bits are allocated in this word for each channel pair indicating the audio sample rate. The encoding of these bits is shown in Table 18-3. An additional bit is provided for each channel pair. If this bit is 1, then the associated channel pair is operating asynchronously.

<table>
<thead>
<tr>
<th>Code</th>
<th>Sample Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>48 kHz</td>
</tr>
<tr>
<td>001</td>
<td>44.1 kHz</td>
</tr>
<tr>
<td>010</td>
<td>32 kHz</td>
</tr>
<tr>
<td>011 to 110</td>
<td>Reserved</td>
</tr>
<tr>
<td>111</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

Active Channels

The fourth data word of an audio control packet indicates which audio channels of the audio group are active. A bit is allocated to each of the four possible channels in the audio group to indicate whether the channel is active. The active bit for a channel is set to 1 if the channel is active.

Audio Delay

Following the active channel data word in the audio control packet, there are four audio delay values. Each audio delay value occupies three data words in the packet. The first three audio delay words indicate the audio delay for the first channel of the audio group. The next three words indicate the delay for the second channel of the audio group, and so on. Audio delay values are 26-bit two's-complement values indicating the accumulated audio processing delay relative to the video, expressed in audio sample intervals. An additional bit in each audio delay value indicates if the audio delay is valid. If an audio delay value is valid, the LSB of the first word of the delay value is set to 1.
Chapter 18: Introduction to Digital Audio for Video Broadcasting

SD Audio Sample Distribution

The audio samples should be evenly distributed throughout the video frame, otherwise there is a risk that the audio input buffer in the receiver could overflow. Some lines in the vertical blanking interval are not used to carry audio samples due to possible corruption of data on these lines during video switching events.

In addition, as described in the Audio Frame Number section, there are specific requirements regarding how many audio samples must be included in each video frame of an audio frame sequence. The length of the frame sequence and the audio sample count in each frame of the sequence are dependent upon the video frame rate and the audio sample rate.

SMPTE 299M: Embedded Digital Audio for HD-SDI

The SMPTE 299M specification specifies how to embed digital audio in HD-SDI video streams. It is similar to SMPTE 272M, but has some significant differences.

HD-SDI has separate HANC data spaces for the Y and C channels. SMPTE 299M audio data packets are only located in the HANC space of the C channel. Audio control packets are only located in the HANC space of the Y channel. There are no extended data packets for embedded audio in HD-SDI because the audio data packets support 24 bits of audio per sample. In SMPTE 299M, the audio data packets are fixed length and carry exactly one sample for each of the four channels in the audio group. The audio data packets also carry additional data and error correction that are not present in the SD audio data packets.

Table 18-4: DID Values for HD Embedded Audio Packets

<table>
<thead>
<tr>
<th></th>
<th>Audio Data Packets</th>
<th>Audio Control Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>0x2E7</td>
<td>0x1E3</td>
</tr>
<tr>
<td>Group 2</td>
<td>0x1E6</td>
<td>0x2E2</td>
</tr>
<tr>
<td>Group 3</td>
<td>0x1E5</td>
<td>0x2E1</td>
</tr>
<tr>
<td>Group 4</td>
<td>0x2E4</td>
<td>0x1E0</td>
</tr>
</tbody>
</table>

HD Audio Data Packets

The format of a HD audio data packet is shown in Figure 18-5.

The first two words of the audio data packet payload carry audio clock phase information that applies to all four audio samples in the packet. This information includes a 12-bit value indicating the number of video clocks that elapsed between the EAV and the occurrence of the audio sample. Normally, the audio data packet for a particular audio sample is located in the HANC space of the very next video line. However, due to restrictions on inserting audio data packets into the HANC space that follows the synchronous switching interval, audio packets that would have been placed in that HANC space are delayed by one line. In order to correctly interpret the audio clock phase information for these delayed audio packets, the multiplex position flag is set to 1 when audio packets are delayed by one video line.

Following the two audio clock phase data words, the payload has four user data words for each of the four audio channels in the audio group. The first group of four words contains the audio sample data for the first channel of the audio group; the second group of four words contains the sample data for the second channel of the group, and so on.
The audio sample data includes 24 bits of the actual audio information and the \( P, C, U, \) and \( V \) bits that are identical to these same bits in the AES3 audio stream. The \( Z \) bit in the first channel sample data is the \( Z \) frame indicator for both channels 1 and 2. The \( Z \) bit in the third channel sample data is the \( Z \) frame indicator for channels 3 and 4. Unlike SD-embedded audio samples, the \( Z \) frame of the two audio channels in a channel pair must always coincide.

Following the user data words for the fourth audio channel, the payload contains six words of error correction code for the audio data packet.

**HD Audio Control Packets**

HD audio control packets are identical in format to SD audio control packets but have different DID values. As with SD audio control packets, each HD audio control packet is specific to one audio group. The audio control packets are sent once per field in the HANC space of the second line after the synchronous switching interval. Audio control packets are sent in the \( Y \) component HANC space.

**HD Audio Sample Distribution**

HD audio sample distribution is more restrictive than for SD. Because of the inclusion of the clock phase data, audio samples must always be inserted in the HANC interface of the following video line except for those audio samples that occur during the video line containing the synchronous switching interval which are delayed by one line.
The maximum number of audio data packets for one audio group that can be located in one HANC space is two. When two audio data packets from the same audio group are present in the same HANC space, they must be ordered so that the audio data packet containing the earliest audio sample information occurs first.

Audio Sample Rate Conversion

Introduction

Sample rate conversion is required when audio applications need a different sample frequency from that of the audio source. Common sample rates include 32 kHz, 44.1 kHz, 48 kHz, 96 kHz. Often, source material recorded with one sample rate must be converted to another sample rate for processing. This is required in such systems as studio digital mixers, audio effects processors, digital audio broadcast equipment, and video systems with embedded audio. In some instances, the inputs and outputs can be synchronized. In others, synchronization is not possible or not desirable, or the sample frequency is time-varying. There are three basic categories for the state of synchronization:

- **Synchronous**: The input sample timing and output sample timing are based on a common timing source—i.e., the output sample rate is a fixed, rational multiple of the input sample rate, and there is no drift between them.

- **Plesiochronous**: The input sample rates are nominally the same, but they are based on two different oscillators. Due to tolerance variations and temperature/voltage drift in the two independent oscillators, the two frequencies are not locked, but differ by a small, varying amount.

- **Asynchronous**: The input rate and output rate are sourced from different oscillators, and also differ in nominal frequency. The sampling rate of the input or output can be variable.

The synchronous case can be handled by a synchronous, or fixed-ratio, sample rate converter. The plesiochronous and asynchronous cases require an asynchronous sample rate converter.

In either case, the function of the sample rate converter is to create a stream of samples timed to match the output timing source and to accurately represent the continuous-time signal embodied by the input samples. This is illustrated in Figure 18-6.

![Sample Rate Conversion](Figure 18-6: Sample Rate Conversion)
Clarification of Terms

In multirate digital signal processing, the term *interpolate* means to increase the sample rate of a signal. In general mathematics, interpolate has a similar but more general meaning: to construct new data points from a discrete set of known data points. For clarity, this application note uses the term *up-convert* to mean increasing the sample rate of an audio signal, and *interpolate* to mean constructing intermediate data points from a set of known data points.

For symmetry, *down-convert* is used to denote decreasing the sample rate of a signal, and implies a filtering process. *Down-sample* and *decimate* mean simply selecting every \( n \)th sample and discarding the rest.

Methods of Sample Rate Conversion

Synchronous

There are a number of approaches to sample rate conversion. In the classic configuration, up-conversion (Figure 18-7) has an up-sampler followed by a low-pass filter to eliminate spectral duplication or images.

![Figure 18-7: Classic Up-Conversion](image)

Down-conversion (Figure 18-8) uses a low-pass filter on the input samples to keep high-frequency components from aliasing into the base band of the output. This is followed by down-sampling by a factor, \( M \).

![Figure 18-8: Classic Down-Conversion](image)

The more general case of conversion by a rational number consists of up-conversion followed by down-conversion, as shown in Figure 18-9. The anti-aliasing filter and anti-imaging filters are combined into a single low-pass filter.

![Figure 18-9: Classic Sample Rate Conversion By a Rational Number](image)
In practice, there are many variations of implementation, most notably performing the up-sampling and down-sampling in multiple stages and moving the location of the low-pass filter, for efficiency, either to the first step or the last step (depending on which has the lower sampling rate). For example, up-conversion from 44.1 kHz to 48 kHz ($L/M = 160/147$) could be done in stages of $2:1$, $2:1$, $5:1$, and $8:147$, with the low-pass filter performed in the last stage. Similarly, down-conversion of 48 kHz to 44.1 kHz ($L/M = 147/160$) could be done in stages of $147:8$, $1:5$, $1:2$, and $1:2$, with the low-pass filter combined with the first stage. In both these cases, the output sample rate is a function of the input rate multiplied by a fixed rational number. In other words, they are synchronous.

Asynchronous

In many practical cases, the input-to-output ratio is not fixed and tends to wander slightly over time because the input is based on a different timing source than the output. In other words, the input and output are asynchronous. A sample rate converter with a fixed conversion ratio cannot handle such situations without one of these two scenarios occurring:

- The input-to-output latency changes due to accumulating delay.
- Artifacts are produced in the audio such as skipping samples or repeating samples.

Both of these cases represent undesirable distortions. In order to handle such cases without introducing artifacts, an asynchronous sample rate converter is required. Asynchronous sample rate converters operate with a variable conversion rate. Conceptually, they up-convert the input signal into a highly over-sampled signal, then down-convert by choosing the sample at the time nearest each output sample, as shown in Figure 18-10.

Thus, the phase of the generated output samples varies smoothly and virtually continuously with respect to the input samples.

The classic method for the up-conversion step is to up-sample by adding zero-valued samples at close intervals between the input samples, then convolving the resulting signal with a low-pass filter as shown in Figure 18-8. There are, however, other methods that are more computationally efficient.
Interpolated-Coefficient FIR Filter

The method described in this application note uses a polyphase filter approach in which several million possible phases of the prototype filter are interpolated at run time from a relatively small number of phases stored in RAM. These intermediate phases are produced by polynomial interpolation. As a result, the resample and filter operations are performed concurrently by a relatively small convolution operation of input samples with the sub-filter for each output sample. This process, shown in Figure 18-11, is referred to in this application note as interpolated coefficient FIR filtering (ICFIR). The conversion factor $L$ is referred to as the ratio. In general, this value can be greater than, less than, or equal to 1, and can vary over time.

![Interpolated Coefficient FIR Filtering](image)

**Figure 18-11:** Interpolated Coefficient FIR Filtering

A unique set of coefficients for the sub-filter is required for every phase of the output sample with respect to the input samples. The required sub-filter is identified by centering the prototype filter at the output sample that is to be calculated.

This is illustrated in Figure 18-12.

![Prototype Filter Centered at Output Sample Position](image)

**Figure 18-12:** Prototype Filter Centered at Output Sample Position
The prototype filter shown is shifted so that the center of the filter lines up with the output sample that is to be calculated. The input sample positions do not necessarily align with the coefficients of the prototype filter. A set of coefficients that do align with the input sample positions are interpolated from the stored coefficients, resulting in the sub filter shown at the bottom of Figure 18-12. When this sub filter is convolved with the corresponding input samples, the output sample of interest is produced. This process is repeated, with new sub-filter coefficients being interpolated for each output sample.

There are several advantageous characteristics of the ICFIR method:

- **Manageable coefficient storage**: For asynchronous conversion, the number of possible conversion ratios and phase relationships are infinite. Therefore storing and using fixed coefficient sets is not practical if reasonable distortion performance is to be attained. The ICFIR can handle millions of unique phases with only a few dozen stored phases.

- **Deterministic latency**: The number of taps used in the FIR convolution is fixed and relatively small. This results in low, deterministic latency relative to the performance achieved. Frequency drift in the input and output is tracked out by minute adjustments to the phase of the interpolated filter.

- **Simple interpolation of coefficients**: Since the prototype filter is smooth and highly sampled, interpolation to a high degree of precision can be accomplished with polynomial interpolation at a much lower computational complexity than an equivalent FIR interpolation.

- **Common sub-filter usage**: The computational cost of interpolating the filter coefficients can be amortized over multiple data streams. For an AES3 channel, pair, the same interpolated sub filter is used for both channels. Additional input channels in the same time domain can also use this same sub-filter.

The ICFIR method also works for synchronous sample rate conversion. In such instances, a major simplification can be made by using a fixed ratio rather than a variable one.

### ASRC Operation

The asynchronous nature of the ASRC means that the input sample rate and output sample rate have no prescribed relationship to one another. Indeed, they are generally controlled from independent timing sources and are both inputs to the ASRC. The ASRC does not have control over either rate, only the ability to monitor the ratio between the two and resample the input data at the output rate. This is illustrated in Figure 18-13.

As shown in Figure 18-14, there are two main components of the sample rate converter: ratio control and resampling. The ratio control section measures the ratio of output samples to input samples and controls how fast the filter moves through the input samples in order to keep the input sample buffer filled to a certain level. The resampler performs...
the interpolation of filter coefficients from a prototype low-pass filter and applies them to the input samples in an FIR convolution.

**Ratio Control**

The ratio control function measures the period of input and output clocks to obtain a ratio value. This value is used to determine where each output sample is located, in time, relative to the input samples. There must also be a buffer for storing input samples that are used in the FIR convolution of the resampler. The calculated ratio is adjusted in minute increments in order to keep the input-to-output latency constant. The ratio control function outputs a precise filter position (which in turn reflects the output sample position) relative to the input samples. This information is required to interpolate the appropriate sub filter. The set of input samples to be convolved with the sub filter coefficients is sent from the ratio control section.

**Resampler**

The resampler function interpolates the sub filter based on the filter position received from the ratio control function. It convolves the interpolated coefficients with the corresponding set of input samples to create the particular output sample of interest.

A set of coefficients representing the prototype low-pass filter is stored in memory. To form each output sample, the resampler receives the filter position relative to input samples. From this, a set of coefficients for the precise phase of output sample are calculated by interpolation of the stored coefficients. This is equivalent to selecting one of many subfilters.

Figure 18-15 shows a simplified graphic example of this process for up-sampling by a ratio of 2.4. The input sample times are denoted by green dots at the bottom of the figure. The output sample times are marked with x’s. The stored coefficients of the prototype filter are shown with the filter centered on the output sample of interest. In this example, the stored prototype filter consists of four phases with four taps in each phase. The particular set of coefficients required for the convolution represents a filter phase whose coefficients lie between the stored coefficients. The position of the coefficients is defined by delta, a fractional value that indicates the relative position, in time, of input samples to output samples. Based on this delta, the four interpolated coefficients are calculated at the locations shown. The convolution of these coefficients with the corresponding input samples produces the output sample of interest.
Filter Expansion for Down-Conversion

For the case of down-converting from a higher sample rate to a lower one, the cutoff frequency of the prototype filter must be lowered to avoid artifacts on the lower-bandwidth output. This is done by expanding the filter to cover more input samples, as shown in Figure 18-16. In this example, the down-conversion ratio is 3/7. The sub-filter...
has more than the four nominal coefficients. In fact, the prototype filter is expanded to
cover \( \sqrt{ratio} \), or 7/3, times as many input samples as in the up-conversion case. The result
in the frequency domain, by the scaling property of Fourier transforms, is compression of
the spectrum proportional to the expansion in time, thus reducing the cutoff frequency.
This is illustrated in Figure 18-17. This effect is equivalent to using a low-pass anti-aliasing
filter on the input samples, as shown in Figure 18-8. A side effect of the spreading of the
filter is that the additional terms in the convolution result in amplification of the result.
Hence, the result of the convolution must be attenuated in amplitude to compensate for the
increased length of the convolution.

<table>
<thead>
<tr>
<th>Filter Coefficients</th>
<th>Frequency Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="XAPP514_02_12_00206" alt="Filter Coefficients" /></td>
<td><img src="XAPP514_02_12_00206" alt="Frequency Response" /></td>
</tr>
</tbody>
</table>

**Figure 18-17: Time Domain Spreading Lowers Cutoff Frequency**

**Synchronous SRC Operation**

![Synchronous Sample Rate Converter](XAPP514_02_13_00206)

**Figure 18-18: Synchronous Sample Rate Converter**

For a synchronous SRC (Figure 18-18), the resampler function remains basically the same.
However, the ratio detector is replaced by a phase incrementer. Rather than actively
measuring the input and output period, this function increments the filter position and
pointers to the input samples based on a pre-determined ratio. The fixed ratio is arbitrary,
because the prototype filter of the resampler can be scaled to handle any ratio. The
resampler function interpolates the required filter coefficients and applies them to the
input sample set to produce the required output samples.

Lagrange Interpolation of Filter Coefficients

Lagrange polynomial interpolation (Figure 18-19) can be used efficiently to interpolate
filter coefficients from the stored prototype filter. For degree $n$, the Lagrange interpolation
calculates a value at a given point based on a function of degree $n$ that passes through the
neighboring $n+1$ points. Zero order equates to a hold interpolator, first order equates to
linear interpolation of 2 points, second order equates to interpolating quadratic equation
over 3 points, and so forth. Accuracy increases with increasing order for smooth sets of
points.

This method of interpolation is well suited to interpolating intermediate values from the
relatively smooth, over-sampled shape of filter coefficients. Lagrange interpolation can be
used successfully to generate intermediate coefficients, as shown in Figure 18-20. The
accuracy of the results is determined by the order of the interpolation.

It does not however, yield high quality results when applied directly to the generally
uneven audio samples. For example, consider the case of a high-frequency tone as shown
in the dashed line of Figure 18-20. The solid line shows the third order Lagrange
interpolation of four sample points. The error from the theoretical signal is significant and
readily apparent.
Performance Factors

The principal figures of merit for sample rate converters are:

- Total harmonic distortion plus noise (THD+N)
- Maximum conversion ratio, for up-conversion and down-conversion
- Maximum sample rate

THD+N, expressed in dB, defines the audio quality of the SRC. It is measured by passing a sine wave at a certain frequency through the SRC and measuring the spectrum of the result. The tone shows up as a high amplitude spike at the given frequency. The amplitude at all other frequencies represent distortion and noise and should be minimal. Since the inputs and outputs are all digital, the only noise is quantization noise due to the finite length of the digital words. For 24-bit audio data, the noise floor is 1 lsb or $2^{-24} = -144$ dB.

Many factors contribute to the harmonic distortion. For the sample rate converters discussed in this application note, the major factors that affect THD+N are number of stored coefficients in the prototype filter, the bit width of the coefficients, the accuracy and stability of input-to-output phase measurement, and the prototype filter function itself (e.g., width of passband, stopband attenuation).

The maximum conversion ratios and the maximum sample rate determine what rate combinations are possible. For each output sample, a complete coefficient set interpolation and FIR convolution must be performed. The time to perform these operations is a function of the number of taps in the FIR filter, and the amount of computational horsepower dedicated to the task (number of computational elements and the clock rate). The time to compute each output sample sets an upper limit on the output sample frequency.

For up-conversion, the input sampling frequency is, by definition, lower than the output frequency, so the input frequency has the same upper limit as the output frequency. For down-conversion, the input frequency is higher than the output frequency, however the FIR convolution is extended to cover more input samples, therefore taking longer to perform. This time increase is a function of the down-conversion ratio and has the effect of limiting the output rate to $\frac{1}{\text{ratio}} \times \text{up-convert maximum}$. This is equivalent to limiting the input ratio to the output sample frequency maximum. Though there are several limiting mechanisms, the net result is that the maximum sample frequency is approximately the same for the input as for the output.
For up-conversion, the maximum ratio is determined by the width of data paths that use the ratio. Down-conversion has the added limit of storage for the additional input samples required in the expanded FIR convolution.

Prototype Filter Design

The prototype filter can be designed using the Filter Design and Analysis tool of Matlab. It is a low-pass equiripple filter. It consists of N phases of M taps each. Higher M and N result in better performance, at the cost of more memory for storage. M is the number of taps in the FIR filter applied for up-sampling. N represents the number of phases to be stored as the prototype filter. Additional phases are produced at run time by interpolating between the stored phases. The prototype filter can be realized by specifying a filter order of $M \times N$ and frequency specs of $1 / (M \times N)$ times the desired response of each phase. The resulting coefficients must be scaled by a factor of N in order to fully utilize the coefficient bit width and to maintain the signal amplitude. For sample rate converters, the transition band is usually symmetric about the Nyquist frequency.

Passband Ripple vs. Stopband Attenuation

For a given filter order and transition band, increasing the stopband attenuation (how effectively high frequencies in the stop band are filtered out) results in increased passband ripple (undesirable variations in gain at different frequencies).

Stopband attenuation is important because it is a primary factor in THD + N performance. For up-conversion, up-sampling results in images of the fundamental spectrum at higher frequencies. For down-conversion, the input data stream can contain frequencies that are beyond the Nyquist rate of the output. In either case, the extraneous high-frequency energy is aliased back into the fundamental spectrum and manifested as harmonic distortion. Therefore, it must be filtered out. The stopband attenuation of the prototype filter directly affects the magnitude of this distortion.

Passband ripple is undesirable because it is a form of distortion; frequencies do not have the same gain. Passband ripple of a filter is normally specified as deviation above and below 0 dB. Digital audio, however, has fixed bit widths, therefore any calculations that would result in values above full scale must be limited or clamped to the full-scale value. This clamping of waves results in distortion artifacts that are unacceptable for sample rate conversion. Therefore, either the samples or the filter coefficients must be attenuated by a small amount (equal to the positive amount of passband ripple) at some point so that clamping, and the resulting clipping of waves does not occur. This means that a passband ripple of $+R$ is manifest as 0 to $-2R$ in the implementation. Ideally, the stopband attenuation would be infinite and the passband ripple would be 0, however in finite length filters, these must be balanced.

Typical Applications

Figure 18-22 shows a typical audio/video multiplexer. Several channels of digital audio enter the mux through AES3 audio receivers. The video signal comes from an SDI receiver and can be either HD or SD. The audio from the AES3 receivers is typically asynchronous to the video clock recovered by the SDI receiver so asynchronous audio sample rate converters are used to synchronize the audio to the video clock. If the audio sample rate is other than 48 kHz, the audio sample rate converters also usually convert the sample rate of the audio to 48 kHz. The audio mux block then creates embedded audio packets for the audio data and inserts them into the horizontal blanking intervals of the video signal. The video signal, with the embedded digital audio, is then transmitted by a SDI transmitter.
Figure 18-22: Example Audio Multiplexer Application

Figure 18-23 shows another typical application for audio multiplexing, demultiplexing, and sample rate conversion. This is a video frame sync application. The video signal, with embedded audio, enters the frame sync through an SDI receiver. The frame synchronizer stores frames of video in a buffer and then synchronizes the video to a local reference clock by occasionally dropping or repeating a frame of video.

The audio must be processed separately. Adding or dropping a frame of video containing embedded audio definitely causes an audible anomaly. An audio demultiplexer extracts the audio from the video signal before the video is written into the video buffer. The audio passes through a sample rate converter and possibly an audio delay buffer (to match the delay of the video signal). The audio is reinserted into the video by an audio multiplexer as the video is read from the frame buffer. The sample rate converter synchronizes the audio signal to the local video clock.

Figure 18-23: Example Frame Sync with Audio Path
Chapter 18: Introduction to Digital Audio for Video Broadcasting

Digital Audio Reference Designs

The following chapters provide reference designs for digital audio. Chapter 19, “AES3 Serial Digital Audio Interfaces for Xilinx FPGAs” provides a reference design for AES3 audio receivers and transmitters. Chapter 21, “AES3 Audio Demultiplexer for Standard-Definition Digital Audio” provides a reference design for an audio demultiplexer for SD video.
AES3 Serial Digital Audio Interfaces for Xilinx FPGAs

Summary

AES3 is a professional standard for transporting digital audio serially over twisted pair or coaxial cable. Each AES3 audio link carries a stereo pair of digital audio channels and supports various audio sampling rates. AES3 is also called AES/EBU (Audio Engineering Society / European Broadcasting Union).

The consumer version of AES3 is called S/PDIF (Sony/Philips Digital Interface). S/PDIF is commonly used to move digital audio between pieces of consumer electronic equipment such as between a DVD player and a surround-sound receiver. The data format and data rates of S/PDIF are the same as AES3. S/PDIF differs from AES3 in the electrical and physical (cable and connector) specifications.

This chapter describes how AES3 and S/PDIF receivers and transmitters can be implemented in Xilinx FPGA devices. The transmitter and receiver modules are very small and can easily be implemented in most current Xilinx FPGA families, including Virtex™-II Pro, Virtex-4, Spartan™-3, and Spartan-3E. The AES audio receiver design presented here requires no external PLL components because it uses digital oversampling techniques to recover the data.

Reference Design

The Xilinx reference designs for the AES3 receiver and transmitter are available at www.xilinx.com/bvdocs/appnotes/xapp514.zip. Open the ZIP archive and extract file xapp514_aes3-audio.zip.

These designs have been tested in hardware using Virtex-4, Virtex-5, and Spartan-3E devices. The reference designs can be implemented in almost any Xilinx FPGA family.

Receiver

The AES3 receiver consists of a data recovery unit, a framer, and a data formatter. The aes_rx module contains all three of these submodules plus some output registers and channel demultiplexing logic as shown in Figure 19-1.

The data recovery unit (aes_multirate_dru) digitally oversamples the AES3 bitstream. It first must determine the approximate length of a state (half a bit) in the bit stream. Logic in the DRU determines how many clock cycles are present in the minimum state width found in the AES3 bitstream. Once this measurement has been taken, the DRU uses that information to pick a sample point approximately in the center of the state to sample each
state’s value. The DRU is constantly measuring the minimum state length so that it can automatically adjust when the input audio sample rate changes.

The DRU produces one recovered bit (in this case, a recovered bit is one state of a biphase-mark encoded symbol) every time it asserts its data_valid output. The rate at which the data_valid output is asserted is dependent upon the audio sample rate. For example, with 192 kHz audio, the rate averages 24.576 MHz.

In order to provide good receiver jitter tolerance, it is recommended that the clock used for the aes_dru module be fast enough to sample the AES3 bitstream at least 4 times during each state of a biphase-mark encoded bit for the highest supported audio sample rate. For example, if 192 kHz is the fastest supported audio sample rate in an application, then a 100 MHz clock is sufficient. The bit rate of 192 kHz audio is 12.288 Mb/s, and the biphase-mark state rate is 24.576 MHz. A 100 MHz clock is just over 4 times 24.576 MHz. This 100 MHz clock is used by the DRU to receive all AES3 sample rates 192 kHz and lower. A DCM can be used to multiply a lower-frequency clock to frequencies sufficient to oversample the AES3 bitstream.

A previous version of the AES3 receiver reference design used a DRU that had a fixed 4X oversampling rate. This required a different clock frequency for each AES3 sample rate. This DRU was, however, somewhat smaller than the new multi-rate DRU and could be useful for certain applications where the audio sample rate is always fixed, typically at 48 kHz. The older DRU is still provided in the reference design files and is called aes_dru_fixed_4x. The two DRUs are interchangeable because they have exactly the same ports. When using aes_dru_fixed_4x, the clock provided to the AES receiver must be very close to 4X the biphase-mark bit rate of the desired bitstream. For 48 kHz sample rate audio, the clock should be 24.576 MHz when using the fixed rate DRU.

The recovered bits from the aes_dru module are sent to the framer module (aes_framer). This module detects the preamble sequences so the data can be aligned properly to the subframe boundaries. The framer decodes the biphase-mark symbols and then deserializes the data into 8-bit bytes properly aligned so that each subframe is output from

Figure 19-1: AES3 Receiver Block Diagram

The DRU produces one recovered bit (in this case, a recovered bit is one state of a biphase-mark encoded symbol) every time it asserts its data_valid output. The rate at which the data_valid output is asserted is dependent upon the audio sample rate. For example, with 192 kHz audio, the rate averages 24.576 MHz.
the framer in four consecutive bytes. The framer data_valid output is asserted whenever a 
byte of data is output on the framer data port.

The formatter module (aes_rx_formatter) takes the data from the framer and formats it into 
audio sample words, valid bits, channel status bits, and user data bits for each of the two 
audio channels. It also detects parity errors.

See Table 19-1 for details of the aes_rx module input and output ports. The two channels 
can be output from the module in a multiplexed manner or demultiplexed. An input port 
called mux_mode tells the aes_rx module which output mode to use.

In multiplexed mode, the audio data for both channels is output on the audio2 port. 
Likewise, the valid, user data, and channel status bits for both channels are output on the 
valid2, user2, and cs2 ports, respectively. When the data for channel 1 is present on these 
output ports, the chan1_en signal is High. When the data for channel 2 is present on the 
output ports, the chan2_en signal is High. These two enable signals can be used as clock 
enables to the downstream logic.

In demultiplexed mode, the data for the each channel is output from the receiver on 
separate ports. The chan2_en output is High when the data for both channels is present on 
the output ports. The chan1_en output is not used in this mode. Refer to Figure 19-2 for 
timing details of the demultiplexed mode and to Figure 19-3 for timing details of the 
multiplexed mode.

Table 19-1: aes_rx Module Ports

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>In</td>
<td>Oversampling clock. See the DRU description for clock frequency requirements.</td>
</tr>
<tr>
<td>rst</td>
<td>In</td>
<td>Asynchronous reset</td>
</tr>
<tr>
<td>din</td>
<td>In</td>
<td>AES3 serial input</td>
</tr>
<tr>
<td>mux_mode</td>
<td>In</td>
<td>When this input is Low, the two output channels are demultiplexed. When this input is High, the two output channels are multiplexed onto the channel 2 output ports.</td>
</tr>
<tr>
<td>locked</td>
<td>Out</td>
<td>High when the receiver is locked to the AES3 bitstream</td>
</tr>
<tr>
<td>chan1_en</td>
<td>Out</td>
<td>In mux mode, this output is High when the channel 1 data is present on the channel 2 output ports. In demux mode, this output is not used.</td>
</tr>
<tr>
<td>audio1[23:0]</td>
<td>Out</td>
<td>In demux mode, the audio word for channel 1 is present on this output port when chan2_en is High. In mux mode, this output port is not used.</td>
</tr>
<tr>
<td>valid1</td>
<td>Out</td>
<td>In demux mode, the valid bit for channel 1 is present on this output port when chan2_en is High. In mux mode, this output port is not used.</td>
</tr>
<tr>
<td>user1</td>
<td>Out</td>
<td>In demux mode, the user data bit for channel 1 is present on this output port when chan2_en is High. In mux mode, this output port is not used.</td>
</tr>
<tr>
<td>cs1</td>
<td>Out</td>
<td>In demux mode, the channel status bit for channel 1 is present on this output port when chan2_en is High. In mux mode, this output port is not used.</td>
</tr>
</tbody>
</table>
### Table 19-1: aes_rx Module Ports (Continued)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan2_en</td>
<td>Out</td>
<td>In mux mode, this output is High when the channel 2 data is present on the channel 2 output ports. In demux mode, this output is High when the channel 1 data is present on the channel 1 output ports and the channel 2 data is present on the channel 2 output ports.</td>
</tr>
<tr>
<td>audio2[23:0]</td>
<td>Out</td>
<td>This output port carries the audio for channel 2 in demux mode and for both channels in mux mode.</td>
</tr>
<tr>
<td>valid2</td>
<td>Out</td>
<td>This output port carries the valid bit for channel 2 in demux mode and for both channels in mux mode.</td>
</tr>
<tr>
<td>user2</td>
<td>Out</td>
<td>This output port carries the user data bit for channel 2 in demux mode and for both channels in mux mode.</td>
</tr>
<tr>
<td>cs2</td>
<td>Out</td>
<td>This output port carries the channel status bit for channel 2 in demux mode and for both channels in mux mode.</td>
</tr>
<tr>
<td>parity_err</td>
<td>Out</td>
<td>Asserted High when a parity error is detected on either channel.</td>
</tr>
<tr>
<td>frames[7:0]</td>
<td>Out</td>
<td>This port indicates the frame number of the data present on the output ports.</td>
</tr>
<tr>
<td>frame0</td>
<td>Out</td>
<td>This port is High when frames equals zero.</td>
</tr>
</tbody>
</table>

#### Figure 19-2: aes_rx Demultiplexed Mode Timing Diagram
The channel status and user data bits are grouped by the AES3 standard into 192-bit blocks. There are 192 bits of channel status and user data for each channel. The aes_rx module does not format the channel status and user data bits into 192-bit wide parallel data. Because these bits are only sometimes used and when they are used, only some of the bits are useful, it is wasteful for the reference design to format these into 192-bit parallel data when many applications do not use all of those bits. Instead, the aes_rx module provides the channel status and valid bits as single output ports for each channel, valid when the associated audio data word is present on the receiver module output. The aes_rx module also provides two ports, frames and frame0 that can be used by custom logic to store and format the channel status and user data as required by a particular application.

The 8-bit frame port indicates the current frame number from zero to 191. The frame0 output is asserted High when frames equals zero. There are many ways to use this information to format the channel status and user data. For example, Figure 19-4 shows how to deserialize the channel status and user data information using a 192-bit shift register. The shift register shifts in one bit into its MSB when the proper chan_en output from the receiver is asserted. When the frame0 output is High, the data from the shift register is transferred into a 192-bit wide output register. This brute force scheme makes all the data bits available to the application at any time. However, it does require 384 flip-flops for the channel status and user data for each channel—a total of 1,536 flip-flops for all channel status and user data bits for both channels.

**Figure 19-3: aes_rx Multiplexed Mode Timing Diagram**
There are other methods of capturing and storing the channel status and user data that are more efficient and more suitable to particular applications. In Figure 19-5, a dual-port memory made from the distributed RAM of Xilinx FPGAs holds the channel status or user data information in a very efficient manner. In this example, an 8-bit wide dual-port memory, 32-locations deep, is made from eight RAM32X1D primitives. The three LSBs of the frames port are decoded and used as write enables so that data bit from the channel status or user data ports is written one bit at a time. Bits [7:3] of the frames port are connected to the address lines of the RAM32X1D primitives. Once written to the RAM, the data can be read a byte at a time through the second port of the memory. This scheme is particularly useful if the data is to read by a PicoBlaze, MicroBlaze, or embedded PowerPC processor in the FPGA. The processor can read the data bytes in random order as needed. The frame0 output of aes_rx could be used as an interrupt to the processor to tell it when all 192 bits have been written into the dual-port RAM.

The example in Figure 19-5 shows how efficient the distributed memory of the Xilinx FPGA architecture can be. It uses just 40 LUTs to capture and store the channel status or user data for a channel, 32 LUTs for the dual-port RAM and another 8 for the bit enable decoding logic. If both the channel status and user data are being captured or this data is being captured for both channels, the 8 LUTs used to decode the RAM write enables can be reused for each dual-port memory array. Thus, it takes 136 LUTs to capture and store both the channel status and user data for both channels.

The aes_rx module has an output signal called locked. This signal goes High when the receiver locks to a valid AES3 bitstream. The locked signal goes high when the first Y preamble is detected. It stays high as long as Y preambles are detected periodically. It is not intended as a bitstream validity signal, but as a general indicator that the receiver is detecting an AES3 bitstream.

Notes:
1. In multiplexed mode, use chan1_en and cs2 to capture C bits for channel 1 and use chan2_en and cs2 for channel 2. In demultiplexed mode, use chan2_en and cs1 to capture C bits for channel 1 and use chan2_en and cs2 for channel 2.

Figure 19-4: Brute Force Deserialization of C and U Data
Figure 19-5: Using Dual-Port Distributed Memory for C and U Data
Transmitter

The AES3 transmitter module (aes_tx) takes in the audio data words for both channels along with the user data, channel status, and valid bits. It formats, encodes, and serializes the data. It calculates parity bits for each subframe. However, it does not generate CRC values for the channel status data. These CRC values need to be calculated externally and fed to the transmitter serially on the channel status input ports at the appropriate time. The transmitter has separate input ports for the two channels and does not have a multiplexed mode like the receiver section.

Table 19-2 shows the input and output ports for the aes_tx module. The clk input port must be a multiple of the AES3 bit rate. The transmitter must also be provided with three clock enables: ce_bp, ce_bit, and ce_word. The ce_bp clock enable must be asserted at the AES3 bitstream symbol rate (2X the bit rate). The ce_bit clock enable must be asserted at the bit rate, and the ce_word clock enable must be asserted at the audio sample rate.

Transmitter clocking example: A 24.576 MHz clock source is supplied to the clk port of aes_tx. The audio sample rate is 48 kHz. Table 18-1, page 396 shows that the AES bitstream bit rate carrying 48 kHz audio is 3.072 Mb/s and the bitstream symbol rate is 6.144 MHz. The 24.576 MHz clock is exactly four times faster than the symbol rate. Therefore, ce_bp must be asserted once every four clock cycles and ce_bit asserted once every eight clock cycles (Figure 19-6). The ce_word clock enable is asserted once every 512 clock cycles (24.576 MHz / 512 = 48 kHz).

\[
\begin{array}{c}
\text{clk} \\
(24.576 \text{ MHz})
\end{array}
\]

\[
\begin{array}{c}
\text{ce}_\text{bp} \\
(6.144 \text{ MHz})
\end{array}
\]

\[
\begin{array}{c}
\text{ce}_\text{bit} \\
(3.072 \text{ MHz})
\end{array}
\]

\[
\begin{array}{c}
\text{ce}_\text{word} \\
(48 \text{ kHz})
\end{array}
\]

\[
\begin{array}{c}
\text{tx_clken_gen}
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 511 & 0
\end{array}
\]

\[
\begin{array}{c}
x728_07_0130C
\end{array}
\]

**Figure 19-6: Transmitter Timing Diagram**
(24.576 MHz Clock and 48 kHz Sample Rate)

The Verilog code to produce the three clock enables for 48 kHz sample rate audio with a 24.576 MHz clock is given here:

```verilog
reg [8:0] tx_clken_gen = 9’d0; // 9-bit counter
wire ce_bp; // 6.144 MHz clock enable
wire ce_bit; // 3.072 MHz clock enable
wire ce_word; // 48 KHz clock enable

always @ (posedge gclk_audio_tx)
    tx_clken_gen <= tx_clken_gen + 9’d1;

assign ce_bp = &tx_clken_gen[1:0]; // asserted at 2X bit rate
```
The minimum frequency for the aes_tx clock is equal to the AES3 bitstream symbol rate. For 48 kHz sample rate audio, the minimum clock frequency is 6.144 MHz.

The module aes_tx_clkdiv, included with the reference design, also provides code to generate the three clock enables required by the AES transmitter. When provided with a 24.5786 MHz clock, this module can produce the correct clock enables for 192 kHz, 96 kHz, 48 kHz, 32 kHz, and 16 kHz sample rates. With a 22.5792 MHz clock, the module can produce clock enables for 176.4 kHz, 88.2 kHz, 44.1 kHz, and 22.05 kHz sample rates.

**Table 19-2: aes_tx Module Ports**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>In</td>
<td>Transmitter reference clock—must run at a multiple of the bit rate. Minimum clock frequency for 48 kHz sample rate audio is 6.144 MHz.</td>
</tr>
<tr>
<td>rst</td>
<td>In</td>
<td>Asynchronous reset</td>
</tr>
<tr>
<td>ce_word</td>
<td>In</td>
<td>Word (sample) rate clock enable—must be asserted High for one cycle of clk at the sample rate. For example, if clk is 6.144 MHz and the sample rate is 48 kHz, ce_word must be asserted once every 128 cycles of clk. The audio, valid, user, and channel status data are all loaded into the transmitter when ce_word is High.</td>
</tr>
<tr>
<td>ce_bit</td>
<td>In</td>
<td>Bit rate clock enable—must be asserted High for once cycle of clk at the bit rate. For example, if clk is 6.144 MHz, ce_bit must be High every other cycle of clk.</td>
</tr>
<tr>
<td>ce_bp</td>
<td>In</td>
<td>Symbol rate clock enable—must be asserted at 2X the bit rate. For example, if clk is running at 6.144 MHz, ce_bp must always be High.</td>
</tr>
<tr>
<td>audio1[23:0]</td>
<td>In</td>
<td>Channel 1 audio sample word.</td>
</tr>
<tr>
<td>valid1</td>
<td>In</td>
<td>Channel 1 valid bit. If the channel is valid, this input must be Low.</td>
</tr>
<tr>
<td>user1</td>
<td>In</td>
<td>Channel 1 user data bit.</td>
</tr>
<tr>
<td>cs1</td>
<td>In</td>
<td>Channel 1 channel status bit.</td>
</tr>
<tr>
<td>audio2[23:0]</td>
<td>In</td>
<td>Channel 2 audio sample word.</td>
</tr>
<tr>
<td>valid2</td>
<td>In</td>
<td>Channel 2 valid bit. If the channel is valid, this input must be Low.</td>
</tr>
<tr>
<td>user2</td>
<td>In</td>
<td>Channel 2 user data bit.</td>
</tr>
<tr>
<td>cs2</td>
<td>In</td>
<td>Channel 2 channel status bit.</td>
</tr>
</tbody>
</table>
The channel status inputs (cs1 and cs2) and the user data inputs (user1 and user2) of the transmitter are serial in nature instead of each being 192 bits wide. The transmitter must be supplied with the 192-bit channel status and user data serially through these ports, one channel status bit and one user bit with each audio sample. As with the receiver section, this can be done by different methods, brute force using 192-bit shift registers or other more efficient ways. If the transmitter is retransmitting the data from an aes_rx module, then the channel status and user data inputs can be connected directly to the corresponding serial outputs of the aes_rx module.

The frame0 input port is used to tell the transmitter when the first bit (LSB) of the 192-bit channel status and user data words are present on the inputs. The frame0 input, when High, causes the transmitter module to insert a Z preamble for the first subframe of the current frame, marking this frame as the first frame of a 192-frame block. The frame0 input must be asserted once every 192 frames.

### Channel Status CRC

The AES3 document defines the last byte (byte 23) of the channel status data as an 8-bit CRC value calculated from the other 23 bytes of channel status data. The AES3 document also states that the channel status CRC value is optional and when not used, the value of byte 23 must be set to zero.

The reference design includes a module for generating and checking channel status CRC values. This module, called aes_crc, can be used in an AES3 receiver for checking the channel status CRC value. It can also be used to generate the channel status CRC value to be inserted into channel status byte 23 in an AES3 transmitter.

The aes_crc module works serially, accepting a channel status bit on its input once per frame and generating the AES3 channel status CRC value using the polynomial:

$$G(x) = x^8 + x^4 + x^3 + x^2 + 1$$  \hspace{1cm} Eq. 1

In a receiver, the aes_crc module compares the 8 bits of the CRC value in byte 23 of the input data against the CRC value it calculates. This comparison is done one bit at a time during frames 184 through 191. The module asserts the crc_err output high if the CRC values are not the same. Figure 19-7 shows how to use two aes_crc modules to check the two channel status outputs of an aes_rx block.
In a transmitter, the dout port provides the complete channel status serial bitstream with the CRC bits inserted. During bytes 0 through 22, the aes_crc module simply passes the din signal through to dout. But during byte 23, the calculated CRC bits are output on dout. Thus, the dout port of the module can be directly connected to the cs1 or cs2 input of the aes_tx module. The aes_crc module is used to calculate and insert the channel status CRC bits for an AES transmitter (Figure 19-8). The frame191 output port of the aes_crc module is used to control when an 8-bit frame counter rolls over to zero after reaching a count of 191. The frame counter is only required if a frame count is not supplied to the transmitter from some other source. The frame191 output can be delayed by one frame to become the frame0 input required by the aes_tx module.

CRC generation and checking can be done by other methods, also. If a processor is being used to create or read the CRC data, the CRC generation or checking can easily be done in software. For example, if the channel status data is captured into dual-port RAM, the processor reading the data from the dual-port RAM after it has been captured can calculate the CRC and compare it to the captured CRC value (Figure 19-5, page 423).

The descriptions of the ports of the aes_crc module are listed in Table 19-3.
## Table 19-3: aes_crc Module Ports

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>In</td>
<td>Clock input. The module must be clocked at the audio sample rate.</td>
</tr>
<tr>
<td>ce</td>
<td>In</td>
<td>Word (sample) rate clock enable.</td>
</tr>
<tr>
<td>frames[7:0]</td>
<td>In</td>
<td>Frame count input. This 8-bit port must indicate the current frame number so that the module knows when to calculate the CRC and when to insert it.</td>
</tr>
<tr>
<td>din</td>
<td>In</td>
<td>The serial channel status bitstream enters the module here. One channel status bit is clocked into the module every frame – every time the ce signal is asserted.</td>
</tr>
<tr>
<td>dout</td>
<td>Out</td>
<td>The serial channel status bitstream with the CRC bits inserted is output from the module on this port.</td>
</tr>
<tr>
<td>crc_out_en</td>
<td>Out</td>
<td>This signal is High during frames 184 through 191, indicating that the CRC bits are being output from the dout port.</td>
</tr>
<tr>
<td>frame191</td>
<td>Out</td>
<td>This output is High when the frames input port equals 191.</td>
</tr>
<tr>
<td>crc_err</td>
<td>Out</td>
<td>This output is High if a CRC error is detected. During frames 184 through 191, the din port is compared to the CRC output bit. If they do not match the crc_err output is asserted High.</td>
</tr>
</tbody>
</table>

### Spartan-3E SDV Board AES3 Demonstration

The Spartan-3E SDV demo board has two AES3 outputs and two AES3 inputs. The Verilog and VHDL files called aes_demo_top provide an AES3 demonstration for the Spartan-3E board. A block diagram of this demonstration is shown in Figure 19-9.

In this demonstration, both AES3 receivers are active and all four audio channels can be viewed using ChipScope™ Pro analyzer. The ChipScope Bus Plot window provides a very convenient way to view the audio as analog waveforms (Figure 19-10). For demonstration purposes, one aes_rx module is connected in multiplexed mode and the other in demultiplexed mode.

The reference clock for the receivers comes from a DCM. The DCM takes a 33 MHz reference clock and synthesizes a 100 MHz clock.

The channel status information from channel 1 of the first AES3 receiver is captured in a dual-port memory. The other port of the memory is connected to a ChipScope Pro VIO module. Through the VIO module, the contents of memory array can be examined, one byte at a time.

The first AES3 transmitter always retransmits the data received by the first AES3 receiver. The second AES3 transmitter can retransmit the data from the first AES3 receiver or it can transmit audio tones generated internally by the FPGA. When transmitting the internally generated tones, channel 1 carries a 100 Hz tone and channel 2 a 500 Hz tone. The frequency of these tones varies with the sample rate. They are 100 Hz and 500 Hz only when the audio sample rate is 48 kHz. Both transmitters always transmit at the same sample rate as the input audio on AES Rx 1 unless this sample rate is 44.1 kHz, 88.2 kHz, or 176.4 kHz.
Figure 19-9: Spartan-3E SDV Board AES3 Demonstration Block Diagram

Figure 19-10: ChipScope Bus Plot Window Showing Audio Data from aes_rx Module
The reference clock for both transmitters comes from a 24.576 MHz VCXO. The VCXO is part of a PLL that recovers a clock from the first AES3 receiver’s input bitstream. The data recovery technique used by the AES3 receiver module does not produce a recovered clock, only recovered data. To retransmit the data from the receiver, a clock is recovered by a PLL. The PLL is constructed using the VCXO with a phase detector and loop filter built in the programmable logic of the FPGA. The control voltage for the VCXO comes from a DAC controlled by the phase detector in the FPGA. There is only one 24.576 MHz VCXO on the Spartan-3E SDV demo board, and both AES3 transmitters are driven from this VCXO.

A shallow FIFO is placed between the first AES3 receiver and the transmitters. This FIFO compensates for short term phase and frequency differences between the VCXO and the rate at which data is recovered by the AES3 receiver.

The sample rate of the audio input to the first AES3 receiver (AES Rx 1) is displayed using general-purpose LEDs marked C, D, E, and F. These four LEDs illuminate to indicate the sample rate of the audio as shown in Table 19-4. LED G is green if the rate detector is successful in determining the sample rate, otherwise it is red.

### Table 19-4: LED Displays for Various Audio Sample Rates

<table>
<thead>
<tr>
<th>Audio Sample Rate (kHz)</th>
<th>LEDs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td>16</td>
<td>OFF</td>
</tr>
<tr>
<td>32</td>
<td>OFF</td>
</tr>
<tr>
<td>48</td>
<td>OFF</td>
</tr>
<tr>
<td>96</td>
<td>OFF</td>
</tr>
<tr>
<td>192</td>
<td>OFF</td>
</tr>
<tr>
<td>44.1</td>
<td>ON</td>
</tr>
<tr>
<td>88.2</td>
<td>ON</td>
</tr>
<tr>
<td>176.4</td>
<td>ON</td>
</tr>
</tbody>
</table>

When the sample rate of the audio received on AES Rx 1 is 44.1 kHz, 88.2 kHz, or 176.4 kHz, it cannot be retransmitted by AES Tx 1 because the 24.576 MHz VCXO cannot be locked to the incoming bitstream to provide a recovered clock. In this case, and when the rate detector cannot determine the audio sample rate, the audio from AES Tx 1 and AES Tx 2 defaults to 48 kHz.
Figure 19-11 and Figure 19-12 show how the general purpose status LEDs, pushbutton switches, and DIP switches on the Spartan-3E SDV demo board are used for the AES3 audio demo.

Figure 19-11: Status LEDs and Pushbuttons in Spartan-3E SDV AES3 Audio Demo

Figure 19-12: DIP Switches in Spartan-3E SDV AES3 Audio Demo

Electrical Interface

The AES3 document describes a 110Ω balanced electrical signal carried on shielded twisted pair (STP) cable with XLR connectors. The 110Ω balanced interface is typically implemented with RS-422 drivers and receivers with transformers to provide impedance matching. Rise-time limiting components are often added to the transmitter interface to meet the rise-time requirements of the AES3 specification. The RS-422 drivers and receivers can easily be interfaced to the input and output buffers of Xilinx FPGAs.

The AES-3id-2001 document describes how to use unbalanced 75Ω coaxial cable with BNC connectors to transport AES3 audio [Ref 2]. This unbalanced AES-3id variant is heavily used in the video broadcast industry. AES-3id is identical to AES3 and AES/EBU in data...
format and bit rate, differing only in the electrical interface details. Refer to the AES-3id document for details of building AES-3id complaint electrical interfaces.

If S/PDIF compliant electrical interfaces are used, the AES modules in this reference design can be used to implement S/PDIF interfaces.

### Jitter

The aes_rx module was tested with a dScope Series III audio analyzer from Prism Sound and found to exceed the AES3 receiver input jitter tolerance requirements. The jitter tolerance of the multi-rate AES DRU is dependent upon the frequency of the oversampling clock and the jitter on this clock. When using oversampling clocks of 66 MHz and 100 MHz produced by the DCM from a 33 MHz clock source, the jitter tolerance of the receiver exceeded 20 UI to at least 4 kHz for all audio sample rates up to 96 kHz (the maximum sample rate provided by the dScope III) and exceeded 10 UI to at least 9 kHz. Above 10 kHz, the dScope III can add only up to 0.5 UI of jitter (except at 96 kHz, where the maximum is 0.375 UI). The AES receiver tolerated these maximum jitter levels without error.

Transmitter output jitter is primarily a function of the jitter on the transmitter clock. Because the bit rates of AES3 are relatively low, the absolute jitter budget is quite large compared to most serial standards implemented with FPGAs these days. Thus, it is very easy to meet the AES3 transmitter output jitter requirements with Xilinx FPGAs.

### Module FPGA Resource Usage

Table 19-5 shows the amount of FPGA resources required to implement the AES3 receiver and transmitter modules. The results for aes_rx include all the submodules that make up the receiver including aes_dru, aes_framer, and aes_rx_formatter. These results do not include any logic external to these modules such as transmitter clock enable generation or logic to collect and process channel status or user data bits. The aes_rx and aes_tx module also do not include any channel status CRC generation or checking. However, Table 19-5 does list the resource usage for one aes_crc module. One aes_crc module is required for each channel status value to be checked or generated.

These results were obtained using ISE 8.1 SP3 and targeting a Spartan-3E device.

### Conclusion

AES3-compliant digital audio receivers and transmitters are easily implemented in Xilinx FPGAs. The reference designs provided with this chapter use very few FPGA resources and can be used with any Xilinx FPGA family. This allows multiple AES3 transmitters and receivers to be implemented in one FPGA with plenty of logic resources left over for other functions.
Chapter 20

Asynchronous Sample Rate Converter

Summary

The asynchronous sample rate converter reference design converts stereo audio from one sample frequency to another. The input and output sample frequencies can be an arbitrary fraction of one another or the same frequency, but based on different clocks. The output is a band-limited version of the input, resampled to match the output sample timing. The reference design has the following features:

- Fully Asynchronous
- –130 dB THD +N Typical (Range: –125 dB to –139 dB)
- 24-bit Audio Word Width In and Out
- Automatic Ratio Detection
- Up-Conversion, Down-Conversion, and 1:1 Asynchronous Conversion
- Rate Change Tracking
- Sample Clock Jitter Rejection
  - Retains full performance over AES3-2003 jitter tolerance curve [Ref 1]
- Input Rates 8 kHz to 192 kHz, Continuous
- Output Rates 8 kHz to 192 kHz, Continuous
- Conversion Ratio 1:7.5 (Down) to 8:1 (Up), Continuous
- Low, Deterministic Latency
- Lock Status Outputs Provided for External Muting

The design is implemented in the Virtex™-4 FPGA architecture. It uses a DSP48 slice as the main math element, and block RAM for input sample buffers and storage of the prototype filter.

Structure

The asynchronous sample rate converter reference design consists of two main functions: the ratio control, and the resampler. These main functions are further divided into smaller functional units, as shown in Figure 20-1. The ratio control function has two main sub units: ratio detection and input sample storage. The resampler has two main parts, interpolation of the correct phase of the filter, and the FIR filter operation that applies the calculated filter coefficients to the set of input samples to form an output sample. The HDL breaks the operations down into modules with functional boundaries. This chapter explains what the modules are, how they fit together, and the details of the functional blocks.
Table 20-1 gives a list and description of the modules. Figure 20-2 shows the hierarchy of the modules in the ASRC reference design, and their relation to the functional blocks.

**Table 20-1: Reference Design Module Descriptions**

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asrc_gold</td>
<td>This is the top level wrapper that instantiates and connects the lower level modules and provides the I/O interface.</td>
</tr>
<tr>
<td>timing_control</td>
<td>Contains the master state machine that controls the creation of each output sample. Instatiates the divider that is used for ratio calculation and normalizing the output samples.</td>
</tr>
</tbody>
</table>
| shared_divider    | 27 x 27 bit signed serial divider. Quotient has 27 integer and 26 fractional bits. This divider is used for:  
|                   | • Calculating the ratio of output sample rate to input sample rate  
|                   | • Normalizing output samples based on the sum of input coefficients |
| ring_buffer_gold  | Pointers and control for the ring buffer memory. The ring buffer stores incoming samples and provides the sample stream to fir_gold. |
| buffer_mem_gold   | 512 x 48 Dual port RAM memory for the ring buffer. |
| ratio_calc        | This module contains the counters for determining input and output sample rates. These rates are sent to the shared divider and the calculated ratio is returned. It also determines the feedback error term based on FIFO level and regulates the ratio accordingly. |
| ratio_filt        | Instantiates the moving_ave_26 and determines when a new ratio has been calculated, and when the filter should be bypassed. |
| moving_ave_26     | Performs a 16-tap moving average filter on the calculated ratio. |
| filt_interp_gold  | Performs the Lagrange interpolation on the prototype filter. Interpolates a filter coefficient for every input sample in the FIR filter operation. |
Table 20-1: Reference Design Module Descriptions (Continued)

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filt_mem_gold</td>
<td>2048 x 24 single port ROM containing the prototype filter. The prototype filter is symmetrical with 4097 coefficients. The middle coefficient is stored separately.</td>
</tr>
<tr>
<td>MULT35X35_SEQUENTIAL_PIPE</td>
<td>35 x 35 multiplier using four sequential states in a DSP48 slice.</td>
</tr>
<tr>
<td>mult_one_third</td>
<td>Fixed multiplier implements a divide-by-3 on a 24-bit number.</td>
</tr>
<tr>
<td>fir_gold</td>
<td>Performs a 64-tap FIR filter for each output sample. Data comes from ring_buffer_gold. Coefficients come from filt_interp_gold.</td>
</tr>
</tbody>
</table>

Figure 20-2: Reference Design Module Hierarchy and Relation to Functional Blocks
Functional Description

Ratio Control Functional Block

The ratio control uses one of two algorithms depending on whether the input rate is changing. At startup, and whenever the input or output rate changes, rate-change tracking mode is used to quickly adjust to the correct ratio, and to adjust the level of the input FIFO to the proper level. In this mode, the ratio correction term grows exponentially with error in order to quickly track large rate changes and reduce the error to low levels. This is illustrated in Figure 20-3.

Once a small error has been achieved and the ratio is stable, an automatic switch is made to locked mode. This mode limits the amount and rate of change of the ratio in order to achieve maximum audio quality. The locked mode can track small drifts in the clock frequencies. However, if a large rate change occurs, the error term exceeds the locked range, and the mode automatically shifts to rate change tracking. When small error is achieved and the ratio is stable, the switch to locked mode again occurs. In this manner, changes in the sample frequencies, large and small, are continuously and smoothly tracked.

The input samples are buffered by the ring_buffer_gold module in the Ratio Control section. When a new output sample is required, the set of input samples required for the FIR filter convolution are sent to the resampler.

Ratio Detection

The ratio detection block is implemented in the ratio_calc and ratio_filt modules of the reference design. A ratio is computed by measuring the period of the input and output clock with a high frequency clock that, in general, is not related to either the input or output clocks. This is shown in the top section of Figure 20-4. To improve the accuracy of this calculation and mitigate the effects of jitter, the input and output clocks are measured over 1024 cycles. The input period is divided by the output period to obtain a calculated ratio.

To further attenuate sample clock jitter, the calculated ratio passes through a moving-average figure contained in the ratio_filt module. The moving average filter is only applied during locked mode, when the input frequency is stable. In frequency tracking mode, the moving-average filter is bypassed; the most current calculated ratio is used for ratio regulation.

![Figure 20-3: Error Correction Curves](image-url)
To regulate the level of the input FIFO, and thus the latency, the FIFO fill level is compared to a reference level in the regulation section. The difference is used as an error signal to adjust the ratio. Since the ratio determines the position of each new output sample relative to the input samples, it effectively controls the speed at which input samples are processed.

Figure 20-4: Ratio Detection Block Diagram
Ratio Calculation

Figure 20-5 is a detailed block diagram of the ratio calculation portion.

Figure 20-6 illustrates how the input period measurement is made. The input_count block counts input clocks on each rising edge. Max_count specifies how many input clocks to count before resetting the counter. It is a parameter in the reference design and is nominally set to 1024. This signal resets the in_period counter as well as the input_count. The in_period counter counts the number of mclk cycles (mclk is the high-frequency processing clock) that occur during max_count + 1 input clocks. At every pulse of term_cnt_in, the in_period_sync register stores the latest in_period count, and the counting begins again. The in_period count resets to 1 so that the resulting count is the actual number of mclk cycles over the specified period, not number of cycles – 1. The in_period_sync value is shifted right by four and sent to sample storage section as in_period_div16.

The output clock period is measured in the same fashion. The ratio is calculated based on the timing of the output clock. To obtain the calculated ratio, calc_ratio, in_period_sync is divided by out_period_sync. This is done by the shared_divider, a multi-cycle pipelined divider in the timing_control module. The calc_ratio signal is sent to the ratio regulation section. The calc_ratio allows for a range of 0 to 15 with 22 fractional bits.
Ratio Filtering for Jitter Tolerance

The AES3-2003 standard [Ref 1] specifies jitter tolerance for AES receivers. The AES receiver must recover the data correctly in the presence of jitter. This jitter in the timing of the audio data is propagated to the sample rate converter. Therefore, the SRC should also have jitter tolerance equivalent to that of the AES receiver. In the reference design, the ratio is filtered for added jitter tolerance. During locked mode operation, the calculated ratio is filtered using a moving-average FIR filter to prevent short-term variations in sampling frequency from causing harmonic distortion in the output sample stream. In other words, it attenuates the effects of input sample clock jitter. The result is no increase in distortion in the presence of jitter. Full performance is retained over the entire range of the AES3-2003 Jitter Tolerance Curve.

Figure 20-6: Timing Diagram of Input Period Measurement with max_count = 4

Figure 20-7 is a block diagram of the ratio filter. It is a recursive implementation requiring only one add and one subtract. As each new data point enters, it is added into the average and the oldest data is subtracted. A shift register is used for the storage element. A 16-location shift register is implemented very efficiently in SRL16 elements requiring only one LUT per input data bit. The calculated ratio has 4 integer and 22 fractional bits. An additional bit of storage is used as a data valid to track data through the shift register. This bit is required for the preload function, which simultaneously bypasses the filter operation and preloads every location in the shift register with the current value of the input data.

When the input signal ptr_reset goes High, the preload_out flag is set high, indicating the block is in preload mode. The preload mode holds the data_in_reg, and the current data in this register is propagated directly to the moving_sum register and on to ave_out. At the same time, the shift register enable is forced active, and the shift register begins shifting in the value data_in_reg. The preload bit also shifts through the shift register each time in parallel with the data. When the shift register has shifted the input value through every location, the preload bit is at the output of the shift register in the form of preload_out. This indicates that the preload cycle is complete. The contents of the shift register and the output register all equal the current input value in data_in_reg. This forces a reset of the preload register, which returns the module to normal filtering mode.
Figure 20-7: Ratio Filter
This preload functionality is used to bypass the moving-average filter when the ratio section is not locked (frequency tracking mode). This allows for better tracking and faster lock. When locked mode is entered, each new ratio calculated is averaged with the values preloaded in the shift register. The term_cnt_in input signals that a new input clock count has completed. The take_calc_ratio signal means a new output clock count has completed, as well as a divide operation to obtain calc_ratio. The combination of term_cnt_in and take_calc_ratio is used to form take_new_ratio, meaning a new data value can be taken into the moving average filter.

Ratio Regulation

The ratio regulation section adjusts the calculated ratio to regulate the input FIFO level. Figure 20-8 shows how this is done. The current fifo_level is compared with the target level, fifo_setpoint. The difference is used as an error_term that adds an offset to calc_ratio. The error term is conditioned separately for locked mode and rate change mode. Parameters in the HDL specify the gain in the error term, the error deadzone, and restrictions in the ratio slew rate, if any. These parameters establish the tradeoff between tight sample rate tracking and harmonic distortion performance. The tradeoff for extremely tight rate-change tracking is the presence of harmonic distortion components caused by the frequent, though minute, rate adjustments.

Figure 20-8: Detailed Block Diagram of Ratio Regulation Section

A small dead zone in the error term—that is, an error threshold below which no adjustment is made to the ratio—makes rate adjustments happen less often, thereby
reducing the distortion component of the rate change. The cost is that rate-change tracking is slower and less accurate. For locked mode, the default settings in the reference design allow a small dead zone (1/4 input-sample time), add no additional gain, and allow one lsb step of rate change per output sample. The frequency-rate tracking mode has no dead zone and no additional gain. This balance allows good rate-change slew and quick, reliable recovery from loss of input. At the same time, it provides good jitter rejection.

The amount and rate of the offset from the calculated ratio depends on whether or not locked mode is engaged. To enter locked mode, the `error_term` must be less than four input samples for five consecutive `term_count_out` times. Unlocked mode, also called rate-change tracking mode, is entered any time the error is more than six samples.

In the rate-change tracking mode, the error term is added directly to `calc_ratio`. It also has an exponential gain such that the correction factor is multiplied at higher error ratios. This facilitates faster locking at startup and after frequency changes without dropping or repeating samples.

In the locked mode, the error term is used to increment or decrement the ratio at a maximum rate of 1 lsb per output sample. The current `reg_ratio` is fed back and compared with the target ratio, `sum_lock`. If the target is different from `reg_ratio`, 1 is either added to or subtracted from the current ratio. The `reg_ratio` value is updated when the set of input samples used for the convolution changes, indicated by a pulse on `move_start_ptr` with a non-zero value of `start_ptr_step`. This limits the slew rate of the ratio to 0.24 ppm per output sample for optimal audio performance. This mode can track slow frequency variations because `calc_ratio` is periodically updated, and the fifo level is updated every output sample with sub-sample accuracy. This is discussed in the description of the ratio control functional block. This high degree of accuracy of `fifo_level` also enables the ratio detection circuit to maintain a deterministic latency when the clocks are stable. That is, for given input and output sample rates, the latency varies by only a fraction of a sample time, and the latency for any two instances of the SRC is the same to within a fraction of a sample time.

**Lock Status Indicators**

There are two top-level output signals that indicate the status of the ratio control section: `locked` and `fifo_overflow`. These two signals can be used to mute the audio when the sample rate converter is outside its bounds of normal operation, and/or when the input sample rate is changing. When `locked` is High, it means that ratio control is in locked mode, with minimal FIFO level error and maximum audio quality. When locked is Low, rate-change mode is active, meaning a more aggressive rate change tracking and correspondingly lowered THD + N performance. The `fifo_overflow` signal indicates the input sample FIFO has overflowed or underflowed, and therefore the output audio is corrupted. This could occur at the application or removal of the input audio stream or during extremely sharp sample rate changes.

Audio quality is severely compromised when `fifo_overflow` is asserted. Depending on the application, rate-change tracking mode audio might or might not be acceptable. These two status bits are provided so that muting can be performed externally when instability in the sample rates could cause unacceptable distortions.

**Input Sample Storage**

The input samples are stored in a ring buffer as shown in Figure 20-9, implemented in block RAM. Two data words of 24 bits each (one for each channel of the channel pair) can be stored per memory location. The ring buffer is 512 wide x 48 deep, enough to accommodate spreading the prototype filter by a factor of 7.5 plus room to act as an input.
FIFO. Two pointers, the write pointer and the start pointer, move through the addresses in the buffer in circular fashion, designating the locations into which input samples are to be written, and out of which input samples are to be read for the filtering operation.

Input samples are stored as they are received at the location indicated by write_ptr. The pointer is incremented each time a new sample is received. The first sample to be used in the FIR filter is indicated by start_ptr. For each output sample, a set of input samples is sent to the FIR filter, starting with the newest (at start_ptr) to the oldest (at end). The start_ptr is updated each time a new output sample is created.

The locations between write_ptr and start_ptr serve as an input FIFO. The difference between the two pointers is used as the fifo_level value. This is used as a feedback mechanism for the ratio. The net effect is to change slightly the rate at which input samples are used in order to keep the FIFO at a predetermined level. The level is set by parameter, and is nominally 16 in the reference design.

Figure 20-9 is a block diagram of the input storage section. The ring buffer consists of the buffer memory (using dual-port block memory) and control logic for reading and writing. For the write port, a write-enable pulse write_en is produced on the rising edge of clkin, the input sample clock. This pulse is used to write sample data into memory and increment the address counter to the next address.

For the read port, go_fir signals the start of a new FIR operation to produce an output sample. This resets the read_addr to the start_ptr value. The fetch_sample signal pulses once for each input sample used in the convolution. Each time it pulses, the read_addr is decremented. This sends sample data to the FIR filter in the newest-to-oldest order shown in Figure 20-9. The read control also generates an integer, start_ptr. The movement of this pointer is controlled by move_start_ptr and start_ptr_step. When move_start_ptr pulses, start_ptr is increased by start_ptr_step. The outputs of this section are input samples, read_data and a data_valid flag.

Since write_ptr and start_ptr are updating at different times and possibly in different increments, the difference between them can vary widely, even if the ratio is correct and the input and output rates are perfectly stable. For example, if the ASRC is performing a down-conversion by a factor of 4, the write pointer increments four times during the time the start_ptr moves just once. Thus, fifo_level varies by 3 over the period of a single output cycle. To reduce such fluctuations, fifo_level is updated only after start_ptr is updated.
Chapter 20: Asynchronous Sample Rate Converter

Fifo_level is calculated to an accuracy of \(1/16\) of a sample by creating sub-sample accurate write and start pointers. As shown in Figure 20-11, the fractional bits for start_ptr come from the inverse of delta_in_ctr_i. This signal indicates the position of the current output sample with respect to input samples. Thus, it represents a fractional start position. These bits are appended to start_ptr to form start_ptr_subsamp.

Figure 20-10: Input Buffer Storage Block Diagram

Figure 20-11: FIFO Level Calculation with Fractional Bits
The circuit in Figure 20-11 shows how the fractional bits for the write_ptr are created. The in_period_sync register of the ratio calculation section contains an accurate count of the number of mclk periods in 1024 input clocks (the nominal max_count). This number is right-shifted so as to obtain the number of mclk periods in $1/16$ of an input period. This number, in_period_div16, is subject to truncation errors, but it is accurate enough to use to create fractional write_ptr bits. The value in_period_div16 is used as the terminal count for the fract_period_cnt counter. The fract_period_cnt, then, pulses every $1/16$ input sample. It is resynchronized to the updating of the write pointer by write_en, the write enable to the ring buffer. The delta_in_count counter counts each $1/16$ of an input sample time. This count saturates at 15, waiting for write_en to provide a reset. Thus, subsample bits are created for the write_ptr and appended to form write_ptr_subsamp.

The difference between write_ptr_subsamp and start_ptr_subsamp determines fifo_level with 4 fractional bits. This fifo_level changes only when the start_ptr is updated, and is fed back to the ratio regulation section.

Clock Domain Considerations

There are several places where the rising edge of the input and output clock must be detected. Since the processing clock mclk is asynchronous to both the input and output clocks, the handling of control signals and of data crossing these clock boundaries must be done with care.

First, the pulse width of the input clock and output clock must be wide enough that they are reliably sampled by mclk. Also, even though Virtex-4 family silicon is hardened against metastability, it does occur on occasion at clock boundaries and should be properly handled. For example, the rising edge of the input clock is detected in the mclk clock domain. Rarely, the first register in the mclk domain experiences metastability. Except for extremely rare cases, measured in decades per event, the output of the first register settles and meets the setup requirements of the second register, so the output of the second register can be assumed to be reliable, and likewise for the third register.

Thus, these can be used to detect the rising edge of the asynchronous input. Figure 20-12 shows a simple circuit that synchronizes the inputs into a new clock domain through reg_1 and reg_2, then detects the rising edge when the input to reg_3 is high, but the output is still low. This circuit is used in the reference design as the interface to the input and output sample clocks. The timing requirements for these signals are shown in Figure 20-23 and Figure 20-24.

![Figure 20-12: Rising Edge Detect](image)
Resampler Functional Block

The resampler creates a set of samples from the input samples based on the output-to-input ratio produced by the ratio detection section. There are two major computational tasks required to produce each output sample:

1. Interpolate filter coefficients for the convolution based on the prototype filter
2. Perform the convolution of the interpolated coefficients with the corresponding set of input samples

Prototype Filter

The prototype filter was designed using the Filter Design and Analysis tool of Matlab. It is a low-pass equiripple filter consisting of 64 phases of 64 taps each. This is done with a filter order of 4096 and frequency specs of $1/64$ of the desired response of each phase. The resulting coefficients are scaled by a factor of 64 in order to fully utilize the coefficient bit width and to maintain the signal amplitude. The prototype filter is symmetric, so only half of the coefficients are stored. Since the filter is of order 4096, there are actually 4097 coefficients. The center coefficient is stored in a 24-bit register, and the rest are stored in block RAM-based memory of size 2048 x 24.

The transition band of the filter is symmetric about the Nyquist frequency: $\text{wpass} = \text{Nyquist} - 9.3\%$, $\text{wstop} = \text{Nyquist} + 9.3\%$. The resulting filter has a passband of 0.4535 times the sampling rate, and a stop band of 0.5465 times the sampling rate. This yields a passband of 20 kHz for a sampling rate of 44.1 kHz, for example. The parameters used for the prototype filter in the reference design are shown in Table 20-2.

Table 20-2: Prototype Filter Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response Type</td>
<td>Low-Pass</td>
<td></td>
</tr>
<tr>
<td>Design Method</td>
<td>Equiripple</td>
<td></td>
</tr>
<tr>
<td>Filter Order</td>
<td>4096</td>
<td></td>
</tr>
</tbody>
</table>
Table 20-2: Prototype Filter Parameters (Continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency Spec Wpass</td>
<td>$\frac{1}{64} \times 2 \times 0.4535$</td>
<td>Normalized</td>
</tr>
<tr>
<td>Frequency Spec Wstop</td>
<td>$\frac{1}{64} \times 2 \times 0.5465$</td>
<td>Normalized</td>
</tr>
<tr>
<td>Magnitude Spec Wpass</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Magnitude Spec Wstop</td>
<td>50000</td>
<td></td>
</tr>
<tr>
<td>Density Factor</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Passband Ripple</td>
<td>$\pm0.016$ dB</td>
<td></td>
</tr>
<tr>
<td>Stopband Attenuation</td>
<td>149 dB</td>
<td></td>
</tr>
</tbody>
</table>

Figure 20-14 shows the frequency response of the resulting filter.

Figure 20-14: Prototype Filter Frequency Response

Figure 20-15 (a and b) show details of the transition band, and Figure 20-16 (a and b) show details of the passband. The (a) figures show the calculated filter response, and the (b) figures show the response based on measurements through the sample rate converter performing a 48 kHz-to-48 kHz asynchronous conversion.

Figure 20-15(a): Prototype Filter Transition Band (Calculated)
Figure 20-15(b): Prototype Filter Transition Band (Measured)

Figure 20-16(a): Prototype Filter Passband (Calculated)

Figure 20-16(b): Prototype Filter Passband (Measured)
Coefficient Interpolation

Equation 1. Third Order Lagrange Interpolation

The filter coefficients are interpolated with a third order Lagrange interpolation according to the equation:

\[ y = \frac{-\Delta(\Delta-1)(\Delta-2)(\Delta-3)}{6}h_0 + \frac{\Delta(\Delta-2)(\Delta-3)}{2}h_1 + \frac{-\Delta(\Delta-1)(\Delta-3)}{2}h_2 + \frac{\Delta(\Delta-1)(\Delta-2)}{6}h_3 \]

where \( h_0, h_1, h_2, \) and \( h_3 \) are four adjacent stored coefficients. \( \Delta \) represents the difference between the location of the coefficients to be calculated (marked with an X) and \( h_0 \), as shown in Figure 20-17.

Equation 2. Third Order Lagrange Interpolation, Factored

To minimize the number of multiplications the equation is factored to:

\[ y = (\Delta - 2) \cdot (\Delta - 3)/2 \cdot [-(-\Delta - 1) \cdot h_0/3 + \Delta \cdot h_1] + \Delta \cdot (\Delta - 3)/2 \cdot [-(-\Delta - 2) \cdot h_2 + (\Delta - 2) \cdot (h_3)/3] \]

The multiplications are performed using DSP 48 blocks configured as 4-stage, 35 x 35 multipliers. The multiplier block follows the “Single Slice 35 x 35 Multiplier Use Model” as described in the "Xtreme DSP Design Considerations" section of the "Virtex 4 Handbook". Each 35 x 35 multiply makes use of 4 passes through the 18 x 18 DSP48 slice. Input multiplexers and output registers for storage of intermediate results are used in conjunction with the multipliers to form multiply/adder units. Two multiply/adder units are used in parallel for the coefficient interpolation. Each unit operates in a 16 state sequence consisting of four, four-state multiplies.

Figure 20-17: Third-Order Lagrange Interpolation

Figure 20-18: Coefficient Interpolator
Figure 20-18 is a block diagram of the coefficient interpolator. The input \( cf \_if \) is the conversion factor from input samples to filter coefficients. This tells how many coefficients correspond to the distance between input samples. For up-conversion, \( cf \_if \) is 16. For down-conversion, \( cf \_if \) is 16 times the ratio of output rate to input rate. The input, \( first \_sample \_f \), is the location of the first input sample to be used in the convolution relative to the stored filter coefficients. The \( curr \_pos \_accum \) register keeps track of the location of each coefficient to be accumulated relative to the stored coefficients.

When a new output sample calculation is started, \( curr \_pos \_accum \) is initialized with \( first \_sample \_f \). As a new filter coefficient is interpolated, \( curr \_pos \_accum \) is incremented by \( cf \_if \). This continues until the end of the stored prototype filter is reached, indicating that all the required coefficients have been interpolated and, consequently, the convolution is complete.

The output of \( curr \_pos \_accum \) is the current position of the filter coefficient in filter space. The integer portion of this quantity is the address of the leftmost stored filter coefficient to be used in the Lagrange interpolation. The fractional portion is the delta value to be used in the interpolation.

The four coefficients used in the interpolation, \( h0, h1, h2, \) and \( h3 \), are retrieved serially and stored in registers during the 16-state interpolation. Likewise, several factors are calculated from \( \Delta \) and stored in registers during interpolation. The \( \Delta \)-related values, along with the associated stored coefficients, are sent to the two 35 x 35 multiply/add units. The multiply/add units operate in parallel, multiplying and summing the various terms of the Lagrange interpolation. There is one final addition to produce the interpolated coefficient that is used in the FIR operation, \( coef \_fir \), as given by Equation 2.

As each FIR coefficient is calculated, it is accumulated in the coefficient accumulator to form \( coef \_accum \), which is subsequently used to normalize the result of the convolution. In the case of down-sampling, the final \( coef \_accum \) of the convolution is virtually equal to the inverse of the scaling factor required to compensate for the increased length of the convolution. In the case of upsampling, the final \( coef \_accum \) is virtually equal to 1 and serves to compensate for small amplitude distortions that might otherwise occur.

**FIR Filter**

The FIR Filter section (Figure 20-19) operates in parallel with coefficient interpolation. After each coefficient is interpolated, it is multiplied by the corresponding input sample and the result is accumulated. A third 35 x 35 multiplier unit performs the multiply operations.
It operates on the same 16-cycle sequence as the multipliers in the coefficient interpolator. Figure 20-20 shows the sequence of operations in this multiplier. Each of the labeled “cycles” consists of four clocks to perform a single 35 x 35 multiply.

<table>
<thead>
<tr>
<th>cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mult 3</td>
<td>Isamp A * coef</td>
<td>Isamp B * coef</td>
<td>Isamp C * coef</td>
<td>Isamp D * coef</td>
</tr>
</tbody>
</table>

Figure 20-20: Multiplier Cycles for FIR Filter

Up to four audio channels (A,B,C, and D) are accommodated; however, the reference design utilizes only the A and B channels. There is also an auxiliary set of inputs and outputs so this unit can perform other multiplications besides those for the FIR filter. These auxiliary multiplications are used by the control section for such functions as converting locations from input sample space to filter coefficient space.

There are separate accumulators for each channel. At the end of the convolution, each accumulator holds the result of the convolution. These results are normalized by dividing the accumulated output sample values, osamp_accum by the coef_sum.

Shared Divider

A module called shared_divider is a signed 27 x 27 multi-cycle pipelined divider with 25 fractional output bits. It has a latency of 56 states, and a new quotient is produced every 8 states. This divider is used in two ways:

- Produce the ratio and 1/ratio values
- Normalize the output sample accumulation values by dividing osamp_accum by coef_accum

The accuracy of these calculations directly affects the quality of the sample rate conversion; thus, a high degree of precision is required. Since the divide operations are done very infrequently compared to other operations, the divider is optimized for minimum area and thus low throughput.

Control

The high-level control is contained in the timing_control module. The output sample clock starts the sample calculation sequence. Each time an output sample is taken, as indicated by output_clk, a new one is calculated. Whereas ratio detection operates in a more or less continuous fashion, the resampler operates in a burst fashion, interpolating a filter phase and performing the convolution each time an output sample is taken, then idling until the next sample is taken and a new one can be calculated. The idle time can be very short (for example, when the output rate is very high), or it can be the majority of the time (for example, when the output rate is low, and the ratio is near 1:1). In any case, the computation starts at the rising edge of the output sample clock and terminates when the end of the prototype filter is reached.

The timing control state machine controls the creation of an output sample. Figure 20-21 is a basic diagram of the state machine. The state register holds the current state. There is a count associated with each state that determines the minimum time each state lasts. State changes occur only when this delay is met, as determined by when the state delay count times out, indicated by state_dly_tc. The status inputs and the current state determine if a state change occurs.
The main purposes of the state machine are to control access to shared multiply and divide resources, sequence the data, and load the results into registers at the proper time. The current state, control bits, and state counter all contribute to the control and timing of these calculations.

Figure 20-22 shows the state diagram of the top level control state machine in the module timing_control. The initial state is RATIO. In this state, the shared divider is used to calculate the ratio from the most recent in_period_sync and out_period_sync values. The state machine remains in this state until the get_new_sample signal asserts in response to the rising edge of clkout.

In the POSITIONS state, a new position for the output sample relative to the input samples is calculated based on the regulated ratio, ratio_reg. The start position in the prototype filter is also calculated. The auxiliary functionality of multiplier unit 3 is used in some of these calculations. The INIT_FIR state simply allows the initial positions to propagate.

The bulk of the calculations happen during the FIR state. In this state, the go_fir signal pulses to indicate the start of a new FIR filter sequence. The resampler is reset and enabled to interpolate and apply filter coefficients to the input samples. The result is a single output sample. The state machine remains in this state until the osamp_done signal is asserted by the resampler indicating that prototype filter has been traversed, and the FIR filter operation is completed.

The SCALE state uses the shared_divider to normalize the accumulated results of the FIR filter by dividing the accumulated results of the FIR by the accumulated coefficients,
coef_sum. Each audio channel is normalized and clamped independently to produce the final output sample value.

The ATTENUATE state is not currently used in the reference design, but is included as a state where attenuation of the output can be performed for fade out or fade in, or overall magnitude control. The auxiliary multiplier functionality can be used to accomplish this.

**Figure 20-22: Top-Level Control State Machine**

The ATTENUATE state is not currently used in the reference design, but is included as a state where attenuation of the output can be performed for fade out or fade in, or overall magnitude control. The auxiliary multiplier functionality can be used to accomplish this.

**Interface Timing**

All data processing in the asynchronous sample rate converter is done in the mclk (high speed processing clock) domain which, in general, has no relationship to either the input or the output clock. The input and output sample clocks are treated as enables that specify when input data is valid, and, on the output, when output data is taken. The period of the sample clocks is used to determine the conversion ratio.

**Figure 20-23** shows the timing requirements for input samples. The clkin signal is resampled in the mclk domain with the circuit of **Figure 20-12**, and the rising edge is used to determine when data is valid. Therefore, setup and hold times are specified in terms of mclk periods. Relative to the rising edge of clkin, there is a setup requirement of 0 and a hold requirement of 5 mclk periods. The clkin signal must be high for a minimum of 5 mclk periods and low for a minimum of 5 mclk periods for accurate edge detection.
Chapter 20: Asynchronous Sample Rate Converter

Figure 20-24 shows the timing characteristics of output samples. Since they are created in the mclk domain, the timing specifications are also given in terms of mclk periods. Relative to the rising edge of clkout, output sample data is valid a minimum of 3 mclk periods before and remain valid until the next sample is presented. 100 mclk periods is given as a conservative minimum time the output data is valid after the rising edge of clkout. Like clkin, clkout (an input to the sample rate converter) is resampled in the mclk domain to determine the output sampling rate. Therefore it has a minimum high time requirement of 5 mclk periods, and a minimum low time of 5 mclk periods.

![Input Timing](image1)

![Output Timing](image2)

Performance

THD + N

Typical performance overall is ~130 dB THD + N. The performance varies somewhat according to the input and output sample rates and the frequency content of the signal. The range is from ~125 dB to ~139 dB.

Table 20-3 lists performance measurements taken over a variety of common ratios. Measurements were taken with a Prism Sound dScopeIII audio analyzer in the default THD + N measurement mode scanned from 20 Hz to 20 kHz. These are examples of possible conversions only. The ASRC reference design allows for virtually infinite combinations of input and output frequencies within the maximum frequency and maximum ratio constraints.
**Table 20-3: THD + N Performance vs. Conversion Frequency**

<table>
<thead>
<tr>
<th>Input Sample Frequency (kHz)</th>
<th>Output Sample Frequency (kHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>–125</td>
</tr>
<tr>
<td>max</td>
<td>–131</td>
</tr>
<tr>
<td>1 kHz</td>
<td>–128</td>
</tr>
<tr>
<td>44.1</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>–129</td>
</tr>
<tr>
<td>max</td>
<td>–131</td>
</tr>
<tr>
<td>1 kHz</td>
<td>–130</td>
</tr>
<tr>
<td>48</td>
<td></td>
</tr>
<tr>
<td>88.2</td>
<td></td>
</tr>
<tr>
<td>1 kHz</td>
<td>–130</td>
</tr>
<tr>
<td>96</td>
<td></td>
</tr>
<tr>
<td>typ</td>
<td>–133</td>
</tr>
<tr>
<td>1 kHz</td>
<td>–132</td>
</tr>
</tbody>
</table>

Maximum Conversion Ratios

- Up-conversion: 8:1
- Down-conversion: 1:7.5

The range of the up-conversion is limited by the number of integer bits in the ratio calculation. The down-conversion ratio is limited by the amount of input sample storage memory, and how much of this memory is allocated to the input FIFO.

Sample Frequency Range

- Input: 8 kHz to 192 kHz
- Output: 8 kHz to 192 kHz

The stated sample frequencies are for 250 MHz master processing clock (mclk). A slower or faster mclk reduces or increases both minimum and maximum sample frequency in approximate proportion to the change in mclk. The upper limit is a factor of processing clock frequency, the lower limit is a function of the width of the period counters and the processing clock frequency.
Chapter 20: Asynchronous Sample Rate Converter

Latency

For any given conversion ratio, the latency is fixed. It is determined by the phase delay of the FIR filter and the fill level of the input FIFO. The FIFO level is fixed at 16, but the size of the FIR filter, and consequently its phase delay vary in the case of down-conversion. Therefore, formula for latency is different depending on whether the sample rate converted is performing up-conversion or down-conversion.

Equation 3: Up-Conversion Latency

For up-conversion, the filter length is 64, therefore the phase delay is 32. The latency, then, is:

\[
\text{Latency} = \text{phase delay} + \text{FIFO delay} = 32 + 16 = 48 \text{ input sample periods}
\]

The time in milliseconds depends on the input sample frequency.

Equation 4: Down-Conversion Latency

For down-conversion, the filter is spread across more samples, so the phase delay and subsequently the latency are longer in terms of the number of input samples:

\[
\text{Latency} = \text{phase delay} + \text{FIFO delay} = (32 \cdot \frac{f_{\text{out}}}{f_{\text{in}}}) + 16
\]

Examples:

48 kHz : 48 kHz conversion:
Latency = 32 + 16 = 48 input sample periods = 1 ms

48 kHz : 96 kHz up-conversion:
Latency = 32 + 16 = 48 input sample periods = 1 ms

32 kHz : 48 kHz up-conversion:
Latency = 32 + 16 = 48 input sample periods = 1.5 ms

96 kHz : 48 kHz down-conversion:
Latency = 32 \cdot 2 + 16 = 80 input sample periods = 0.83 ms

48 kHz : 44.1 kHz down-conversion:
Latency = 32 \cdot 48/44.1 + 16 = 50.83 input samples = 1.06 ms

In cases of changing frequency, the latency changes smoothly as specified in Equation 3 and Equation 4. For up-conversion, changes in input sample frequency result in changes in latency; changes in output sample frequency do not. For down-conversion changes in input or output sample frequency result in changes in latency.
FPGA Resource Utilization and Performance

The reference design has been implemented and hardware verified in Virtex-4 family devices with the results shown in Table 20-4 and Table 20-5:

Table 20-4: Reference Design Resource Utilization

<table>
<thead>
<tr>
<th>LUTS</th>
<th>Registers</th>
<th>DSP48 Slices</th>
<th>RAM Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,750</td>
<td>3,235</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 20-5: Reference Design Frequency Performance

<table>
<thead>
<tr>
<th>Speed Grade</th>
<th>Processing Clock Freq. (mclk)</th>
<th>General Max Sample Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-12</td>
<td>255 MHz</td>
<td>192 kHz</td>
</tr>
<tr>
<td>-11</td>
<td>225 MHz</td>
<td>170 kHz</td>
</tr>
<tr>
<td>-10</td>
<td>200 MHz</td>
<td>150 kHz</td>
</tr>
</tbody>
</table>

To calculate minimum required processing clock frequencies for specific conversions, use the following equations:

- Up-conversion: \( F_{mclk} = F_{sout} \cdot 1325 \)
- Down-conversion: \( F_{mclk} = F_{sinput} \cdot 1030 + F_{sout} \cdot 270 \)

For example, a down-conversion from 192 kHz to 48 kHz requires a clock frequency of \( 192 \text{ kHz} \cdot 1030 + 48 \text{ kHz} \cdot 270 = 210.72 \text{ MHz} \), so a -11 part works for this particular conversion.

Additional Channels

The bulk of the FPGA resources in the reference design are required for the functions that distinguish an asynchronous sample rate converter from a regular sample rate converter: tracking the ratio and interpolating the coefficient set. This functionality is common to all channels, and it need not be replicated for additional channels that share the input clock and output clock. The architecture is such that ratio detection and filter interpolation can be easily shared with additional channels. Therefore, the incremental resource usage for adding channels is small. An additional benefit from sharing these resources is the outputs are inherently phase matched.

For example, to add a second stereo pair to the reference design, the FIR filter already has time slots for 4 channels, so only the input buffer needs to be replicated, along with a pair of output registers to hold the computed output samples, as shown in Figure 20-25.
Table 20-6 shows the approximate resource utilization for a 4-channel (2-pair) ASRC configuration.

Table 20-6: 4-Channel Resource Utilization

<table>
<thead>
<tr>
<th></th>
<th>LUTS</th>
<th>Registers</th>
<th>DSP48 Slices</th>
<th>RAM Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>2900</td>
<td>3300</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Per Stereo Pair</td>
<td>1450</td>
<td>1650</td>
<td>1.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

For an 8 channel (4 pair) configuration, the per-pair cost goes down even more dramatically. Two additional instances of the input sample storage element are required, but only one additional instance of the FIR filter is needed. Additional time slots for output sample normalization must be allotted, reducing the maximum frequency, but only by a few percent.

Table 20-7 shows the approximate resource utilization for an 8-channel (4-pair) ASRC.

Table 20-7: 8-Channel Resource Utilization

<table>
<thead>
<tr>
<th></th>
<th>LUTS</th>
<th>Registers</th>
<th>DSP48 Slices</th>
<th>RAM Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>3600</td>
<td>4000</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>Per Stereo Pair</td>
<td>900</td>
<td>1000</td>
<td>1</td>
<td>2.75</td>
</tr>
</tbody>
</table>
Data Flow Spreadsheet

The reference design includes a data flow spreadsheet, `asrc_dataflow.xls`, that graphically illustrates the timing of data as it flows and gets processed from register to register through the pipeline. The registers are listed across the top of the spreadsheet and each row represents one clock cycle, progressing from top to bottom. There are three sheets corresponding to three stages of processing: filter phase interpolation, FIR filter, and output sample normalization.

The dataflow spreadsheet is an aid understanding the intent of the RTL, as well as a powerful tool for debugging, and making and documenting modifications. Mult-cycle paths are apparent from inspection of the dataflow spreadsheet. Comments are used on some of the cells to give cycle-specific explanations. There are enough passes to show the entire start up sequence and the process end sequence for an output sample calculation. Obviously there are many more passes required in order to perform the filter interpolation and FIR for and output sample.
Chapter 20: Asynchronous Sample Rate Converter

Filter Phase Interpolation and FIR Filter

The filter interpolation and FIR filter blocks operate on a 16-state cycle matching the 16-state cycle of the multipliers (4 multiplications of 4 states each). The left columns contain the cycle and sub-cycle counts and a column labeled pass. The pass field is for correlation of the startup sequence between the filter phase interpolation page and the FIR filter page.

The cells in the interior of the spreadsheet indicate the data present in a particular register at a particular time. The registers are arranged so that the flow of data is generally upper left to lower right. In most cases, there are only entries for new, valid data. Blank cells indicate, depending on the context, either invalid data or data that remains the same as previous entries or not relevant at that time. To the right of the register list is a list of the state machine output bits. These illustrate the timing relationship of state machine bits to data path registers. One cycle of the state machine is boxed. The boxed area contains a complete state machine cycle and is corresponds directly to the RTL state machine definition. There are unused control bits in the state machine. These are to facilitate the addition of control outputs to the state machine, and are removed by synthesis.

Output Sample Normalization

The output sample normalization page shows data flow in the timing_control module, particularly the last stage of output sample processing done during the SCALE state of the timing control state machine (see Figure 20-22). This sheet details the data sequencing through the divider for normalization, and the timing for the steps of rounding, clamping, and loading the output registers. Register names are shown in the top of each column. Data for each audio channel (a,b,c,d) passes through these stages in sequence. Although time slots for four audio channels are shown, only two (a,b) are used in the reference design. This page of the spreadsheet reflects the latency (56) and throughput (one result per 8 states) of the divider. Since this is a shared divider, the timing also applies to the other operations it performs.

Design Files

The Xilinx reference designs for the AES3 receiver and transmitter are available at www.xilinx.com/bvdocs/appnotes/xapp514.zip. Open the ZIP archive and extract file xapp514_samp-rate-conv.zip.

Conclusion

The asynchronous sample rate converter reference design is useful for converting between source and destination audio streams with independent clocking sources. It has excellent THD + N performance, and converts over a wide range of ratios and frequencies, automatically detecting the conversion settings and tracking changes in frequency. The reference design is implemented in Virtex 4 using dual-port block RAM and DSP48 Slices, in addition to general logic resources. The implementation uses interpolated-coefficient FIR filtering. It precisely interpolates millions of filter phases from a small set of stored phases and applies the interpolated filter to the corresponding input samples to obtain an output sample at a particular instant. The latency is low and deterministic due to the use of a single FIR filter stage. The source code and dataflow spreadsheet, along with the functional description and filter parameters are provided to facilitate integration of the design “as-is” or adaptation of the design for specific application requirements.
AES3 Audio Demultiplexer for Standard-Definition Digital Audio

In many applications, embedded audio must be extracted, or demultiplexed, from the video stream so that the audio and video can be processed separately. This chapter describes an audio demultiplexer for standard-definition (SD) digital video. This small, efficient reference design can be implemented in the Xilinx Virtex-5, Virtex-4, Virtex-II Pro, Virtex-II, and Spartan-3E FPGA families. One module can demultiplex audio from up to sixteen separate video streams.

Specifications and Features

- **Standards:** The demultiplexer detects and extracts embedded audio conforming to SMPTE 272M-2004 [Ref 1] from standard-definition 4:2:2 component digital video streams conforming to SMPTE 125M and ITU.R BT-656. The demultiplexer supports non-PCM data conforming to SMPTE 337M-2000. The demultiplexer does not distinguish between non-PCM data and regular PCM embedded audio, treating both types of data identically. It is up to logic external to the demultiplexer to determine whether the data is PCM or non-PCM and treat it appropriately.

- **Multiple input streams:** A single audio demultiplexer module can demultiplex the audio from multiple video streams. Running at the standard SD video clock rate of 27 MHz, the demultiplexer supports up to six NTSC video streams (five for PAL). Running at 72.5 MHz, the demultiplexer supports up to 16 input video streams. The demultiplexer has a maximum capacity of 16 input video streams. The input video streams can either be synchronous or asynchronous relative to the demultiplexer’s clock.

- **24-bit audio support:** The demultiplexer supports both 20-bit and 24-bit audio samples. It automatically detects and processes the extended data packets containing the extra four bits of audio data for each sample. An output from the module indicates whether each audio sample is 20 or 24 bits wide.

- **Audio group support:** The demultiplexer supports all four audio groups and supports both channel pairs in each audio group, for a total of 16 audio channels per video stream. Inputs to the module allow control over which audio groups and channel pairs are demultiplexed and which are not. This selection is independent for each input video stream when multiple video streams are processed by one demultiplexer module.

- **Indication of channel pairs present in the video stream:** The demultiplexer has "sticky" status outputs indicating which channel pairs in each audio group are present in each input video stream. The time period over which these sticky status bits are accumulated is user defined and controlled by inputs to the module. There are independent clear inputs for the status bits of each input video stream.
• **Full AES3 data support:** The demultiplexer outputs the AES3 channel status bit, user data bit, valid bit, and Z frame bit with each audio sample.

• **Audio control packet support:** The demultiplexer supports audio control packets. The data words from the payload portion of audio control packets are output in their raw format from the module on a separate port along with several status signals, allowing the application to capture all of the data from the control packet or only specific words that are of interest to a particular application.

• **Audio packet deletion in video stream:** The demultiplexer can optionally modify the video stream to mark audio packets as deleted. This feature is only available when processing a single video stream with the demultiplexer.

• **Error detection and handling:** The demultiplexer gracefully handles and reports a variety of error conditions:
  
  ♦ Each output audio sample is accompanied by a parity error signal generated by checking the parity of the audio sample.
  
  ♦ The demultiplexer rejects audio packets that have an invalid data count (DC). The data count of audio packets, even non-PCM data packets, must be a multiple of 6 to be valid. The module generates an error code when it rejects an audio packet for this reason.
  
  ♦ The demultiplexer rejects extended data packets that have an invalid DC. Extended data packets must be of the proper length to match the preceding audio packet. The module generates an error code when it rejects an extended data packet for this reason. All audio samples associated with the rejected extended data packet are output as 20-bit samples.
  
  ♦ Audio packets that have a checksum error are processed and the audio samples output as normal, but the last sample of the packet is accompanied by a checksum error flag. The module has an output that indicates the first sample of each audio packet. By tracking the start of packet signal and the checksum error signal, it is possible to determine which audio samples are potentially corrupted, allowing the implementation of techniques like soft muting to avoid potentially objectionable audio noise. Optionally, audio packets with checksum errors can be rejected by the demultiplexer. An input pin determines which policy is observed for audio packets with checksum errors.
  
  ♦ Extended data packets with checksum errors are processed as normal and the audio samples are output as 24-bit audio samples.
  
  ♦ Incomplete audio packets, those that are prematurely terminated by another ancillary data packet or by the end of the HANC interval, are rejected. The module generates an error code when it rejects a packet for this reason.
  
  ♦ Incomplete extended data packets, those that are prematurely terminated by another ancillary data packet or by the end of the HANC interval, are rejected. The audio samples from the associated audio packet are output as 20-bit audio samples.
  
  ♦ The data block number (DBN) from each audio data packet is output with the audio sample. This makes it possible to check the DBN values on the output of the demultiplexer for discontinuities. DBN discontinuities indicate missing audio packets. These discontinuities can be caused by the video stream switching between video sources or by the demultiplexer dropping packet for various errors as indicated above.
Usage Models

The demultiplexer module is very flexible and can be used in a variety of ways. Described here are some typical usage models.

One Video Stream with Video Rate Clock

In Figure 21-1, a single video stream is connected to the input of the demultiplexer. The clock used by the demultiplexer is the standard 27 MHz video clock, and the video is synchronous with this clock. The module processes each and every word of the video stream. The TRS and ADF detection module indicates the presence of these special sequences to the demultiplexer. This mode of operation supports the audio packet deletion feature. Audio packet deletion from the video stream is optional and can easily be disabled.

Figure 21-1: One-Input Video Stream with 27 MHz Video Clock

Some power savings can be achieved if the clock enable of the demultiplexer is controlled so that the demultiplexer is only enabled during the HANC portion of each active video line. This can be done by asserting the demultiplexer’s in_ce port when the EAV is detected and negating it when SAV is detected. The video packet deletion feature cannot be used if this power saving scheme is implemented.

One Video Stream with Faster Clock

When using oversampling data recovery techniques for receiving SD-SDI with MGTs, it is common for the frequency of the clock from the SD-SDI receiver to be a multiple of 27 MHz, such as 108 MHz (4X), 148.5 MHz (5.5X), or 270 MHz (10X). The SD-SDI oversampling data recovery unit (DRU) produces a clock enable that is asserted whenever a data word is recovered, throttling the data rate down to 27 MHz. The demultiplexer can be interfaced to the output of such a DRU by connecting the in_ce port to the clock enable signal produced by the DRU (Figure 21-2).

Figure 21-2: One-Input Video Stream with High-Speed Clock & Input Clock Enable
This mode supports the audio packet deletion feature.

The demultiplexer has an independent output clock enable for the output data. The input and output sides of the demultiplexer use the same clock, but have separate clock enables, allowing the output data rate to be independent of the input data rate.

Multiple Synchronous Video Streams

The demultiplexer can process up to six NTSC or five PAL input video streams when running from a standard 27 MHz video clock, or up to 16 input video streams with a 72.5 MHz clock. Processing multiple input streams requires FIFOs and control logic on the input of the demultiplexer so that the demultiplexer only has to process the HANC data of each video stream. In Figure 21-3, the TRS and ADF detection module and the FIFO write control logic work together to write only the HANC data from each input video stream to the associated FIFO.

The input stream control module selects which FIFO is read by the demultiplexer. The demultiplexer can not switch from one input stream to another arbitrarily. It is designed to only switch after all of the HANC data for one stream has been processed. When the last word of an SAV is read from a FIFO, the input stream controller selects another video stream—one with a non-empty FIFO. If all of the FIFOs are empty, the demultiplexer is disabled until one of the FIFOs contains data. If the FIFO for the currently selected input stream becomes empty before the end of the HANC (i.e., before the SAV is detected), the demultiplexer stalls until that FIFO has more data available. The input stream controller has a timeout mechanism that prevents a stalled video stream from stopping the processing of data for other streams. The input stream controller can force the demultiplexer to switch to another input stream. However, if the demultiplexer is in the middle of processing an audio packet when it stalls, the loss of some data could result. The demultiplexer treats this as a prematurely terminated packet. In the reference design, the timeout period is selected by a Verilog parameter or VHDL generic. It should be set appropriately, taking several things into account:

- How long can the application allow the demultiplexer to stall, waiting for a video stream that has stopped? This is determined primarily by how much overhead the input clock frequency allows. For example, with six-input NTSC video streams and the demultiplexer running at 27 MHz, the clock is almost 1 MHz faster than is needed to process the six video streams. Thus, there is plenty of allowance for overhead and the timeout period can be generous. Timeout periods that are too short can cause the demultiplexer to lose audio data.

- The timeout counter is not enabled by \text{in\_ce} because a stalled video stream could possibly cause the clock enable to remain Low. The timeout period must be adjusted to take into consideration the overclocking factor. For example, if a 108 MHz clock is used and \text{in\_ce} is normally asserted once every four clock cycles, the timeout period should be four times longer than if the input clock is 27 MHz.

In Figure 21-3, all of the input video streams are synchronous, running from the same video clock. Each input video stream can, however, have an independent clock enable signal to control writes to the input FIFO.

The input clock also clocks the output side of the FIFOs, the stream control module, and the core of the demultiplexer. Synchronous FIFOs are used in this application because the input and output side of the FIFOs use the same clock.
Figure 21-3: Multiple Synchronous Video Streams
Multiple Asynchronous Video Streams

The demultiplexer can also be used with multiple asynchronous video streams—up to sixteen, depending on the frequency of the clock provided to the demultiplexer. This application is very similar to the multiple synchronous video stream application. In this case, however, asynchronous FIFOs are used. As shown in Figure 21-4, each FIFO has an independent video clock that can be a 27 MHz clock or an oversampling clock with a clock enable. The output side of each FIFO is in the same clock domain as the input stream controller and the demultiplexer. The asynchronous FIFOs handle the synchronization of the data between the input and output clock domains.

Clock Requirements for Processing Multiple Video Streams

For both the synchronous and asynchronous cases, the minimum clock frequency required for the demultiplexer when processing multiple input video streams is determined by the number of words in the horizontal blanking interval of the video formats to be supported, the number of video lines per frame, and the video frame rate.

NTSC has 276 words in the blanking interval, including the EAV and SAV. With 525 lines per frame and a frame rate of 29.97 Hz, the required demultiplexer clock frequency is calculated as:

$$\text{clock frequency} = \frac{276 \text{ words/line} \times 525 \text{ lines/frame} \times 29.97 \text{ frames/sec}}{\text{number of input video streams}}$$

This works out to about 4.34 MHz per input video stream. Thus, a 27 MHz clock is sufficient to support six input streams of NTSC video.

PAL has 288 words in the horizontal blanking interval, 625 lines per frame, and 25 frames per second. This works out to exactly 4.5 MHz per input video stream or exactly 27 MHz for six PAL streams. However, there are a few clock cycles of overhead used by the demultiplexer when switching between input video streams. When clocked at 27 MHz, the demultiplexer does not have extra clock cycles available for this overhead. Thus, a 27 MHz clock only supports five PAL video streams. Some overhead should also be included to compensate for shortened lines that might occur during synchronous switching events. Even though there are usually no audio packets in the HANC on the lines surrounding the synchronous switching interval, the demultiplexer still processes all of the HANC words on these lines.

If other video standards, such as wide-screen NTSC and PAL, are supported by the application, the required clock frequency must calculated using the size of the HANC interval of these additional video standards.

Modules

The basic audio demultiplexer module is called sd_aes_demux. This module has all the ports necessary to support 16 video streams, but does not have the support modules such as the input stream controller, FIFOs, and TRS/ADF detector.

Because this module has so many ports to support 16 video streams, it is unwieldy when used with a single input stream. The module called sd_aes_demux_1 is a wrapper around the sd_aes_demux module, exposing only those I/O ports required to support a single video stream.

There are several wrapper modules optimized to support different numbers of video streams. The module sd_aes_demux_4 supports four video streams and includes the input stream controller. For convenience, additional wrapper modules are provided that also include the input FIFOs, FIFO input control logic, and TRS/ADF detection.
The sd_aes_demux_4_sync module provides everything necessary to support four synchronous input streams. In addition, the sd_aes_demux_4_async module provides everything necessary to support four asynchronous input streams. These two modules are wrappers around the sd_aes_demux_4 module, which in turn is a wrapper around sd_aes_demux.
There are equivalent wrapper modules supporting eight and sixteen input streams. For applications with other than 1, 4, 8, or 16 input streams, one of the wrapper modules can be modified to support the required number of video streams. Alternatively, a wrapper module supporting more streams than required can be used with the unused input ports hard-wired appropriately.

Table 21-1 lists all of the main modules included in the reference design.

### Table 21-1: List of Modules

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Module Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sd_aes_demux</td>
<td>This is the main demultiplexer module with support for up to 16 input streams.</td>
</tr>
<tr>
<td>sd_aes_demux_1</td>
<td>This module provides the ports necessary to support one input stream. The TRS/ADF detection module is not included.</td>
</tr>
<tr>
<td>sd_aes_demux_4</td>
<td>This module provides the ports necessary to support four input streams. Input FIFOs and TRS/ADF detection are not included.</td>
</tr>
<tr>
<td>sd_aes_demux_4_async</td>
<td>This module supports four asynchronous input streams. It includes the asynchronous input FIFOs and TRS/ADF detection module.</td>
</tr>
<tr>
<td>sd_aes_demux_4_sync</td>
<td>This module supports four synchronous input streams. It includes the synchronous input FIFOs and TRS/ADF detection module.</td>
</tr>
<tr>
<td>sd_aes_demux_8</td>
<td>This module provides the ports necessary to support eight input streams. Input FIFOs and TRS/ADF detection are not included.</td>
</tr>
<tr>
<td>sd_aes_demux_8_async</td>
<td>This module supports eight synchronous input streams. It includes the synchronous input FIFOs and TRS/ADF detection module.</td>
</tr>
<tr>
<td>sd_aes_demux_8_sync</td>
<td>This module supports eight synchronous input streams. It includes the synchronous input FIFOs and TRS/ADF detection module.</td>
</tr>
<tr>
<td>sd_aes_demux_16</td>
<td>This module provides the ports necessary to support sixteen input streams. Input FIFOs and TRS/ADF detection are not included.</td>
</tr>
<tr>
<td>sd_aes_demux_16_async</td>
<td>This module supports sixteen synchronous input streams. It includes the synchronous input FIFOs and TRS/ADF detection module.</td>
</tr>
<tr>
<td>sd_aes_demux_16_sync</td>
<td>This module supports sixteen synchronous input streams. It includes the synchronous input FIFOs and TRS/ADF detection module.</td>
</tr>
<tr>
<td>sd_aes_demux_infifo_ctrl</td>
<td>This module controls writing of video data to an input FIFO so that only HANC data is written. One instance of this module is included for each FIFO in the demultiplexer modules that include input FIFOs.</td>
</tr>
</tbody>
</table>
Table 21-1: List of Modules (Continued)

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Module Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sd_aes_demux_infsm</td>
<td>This module implements the input side state machine in sd_aes_demux.</td>
</tr>
<tr>
<td>sd_aes_demux_instream_ctrl</td>
<td>This is the input stream controller module. It is included in all multi-stream demultiplexer modules.</td>
</tr>
<tr>
<td>sd_aes_demux_outfsm</td>
<td>This module implements the output side state machine in sd_aes_demux.</td>
</tr>
<tr>
<td>sd_aes_pkt_del</td>
<td>This module implements the audio packet deletion feature. It is included in the sd_aes_demux_1 module.</td>
</tr>
<tr>
<td>sd_aes_present_flags</td>
<td>This module is used by sd_aes_demux to implement the channel pair detection flags.</td>
</tr>
<tr>
<td>sd_aes_pri_encoder_16</td>
<td>This module is part of the arbiter in the sd_aes_demux_instream_ctrl module.</td>
</tr>
<tr>
<td>sd_aes_trs_adf_detect</td>
<td>This module implements TRS (EAV and SAV) and ADF detection. An instance of this module is included for each video stream in the multi-stream demultiplexer modules listed above as including TRS/ADF detection.</td>
</tr>
<tr>
<td>sd_aes_demux_buffer_512</td>
<td>This module is instantiated in sd_aes_demux. It provides the block RAMs required for the 512-deep sample buffer. This is the standard buffer size.</td>
</tr>
<tr>
<td>sd_aes_demux_buffer_2K</td>
<td>This is an example of how to implement a larger sample buffer. This one is 2048 samples deep. This module can optionally replace the 512-deep sample buffer.</td>
</tr>
</tbody>
</table>

I/O Port Description

This section describes the I/O ports of the audio demultiplexer wrapper modules. Only those ports for the sd_aes_demux_1, sd_aes_demux_x_async, and sd_aes_demux_x_sync are described.

Clock and Control Signals

Table 21-2 shows the input signals that provide clocks, clock enables, resets, etc.

Table 21-2: Clock and Control Signals

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>In</td>
<td>1</td>
<td>Clock for the demultiplexer. For all modules except those with asynchronous FIFOs, all inputs are synchronous with this clock. All outputs from the demultiplexer are also synchronous with this clock.</td>
</tr>
<tr>
<td>rst</td>
<td>In</td>
<td>1</td>
<td>Reset input. This is an asynchronous reset. In order to insure that the state machines in the demultiplexer exit reset cleanly, it is recommended that the falling edge of the rst signal meet setup and hold times on all flip-flops clocked by the clk input. All critical control logic in the demultiplexer exits FPGA configuration in a known good state, so for most applications, the rst input can simply be hardwired Low.</td>
</tr>
</tbody>
</table>
Chapter 21: AES3 Audio Demultiplexer for Standard-Definition Digital Audio

The demultiplexer has two state machines, one on the input side and the other on the output side. These two state machines are independent of each other, although they share a common clock. The input state machine and the output state machine each have their own clock enable signals, called in_ce and out_ce. The input and output sides of the demultiplexer can have different clock rates by asserting these two clock enables at different rates. Most of the outputs from the demultiplexer are controlled by out_ce. However, some outputs are generated by the input state machine and are controlled by in_ce. These exceptions are noted in the descriptions of the ports.

Audio samples must be read from the output of the demultiplexer fast enough that the internal sample buffer does not overflow. Samples are read by asserting both out_ce and output_ack when the output_ready output is asserted. The output state machine takes several clock cycles to output each audio sample. In fact, it takes seven clock cycles to output one sample pair (two samples from the same channel pair), and out_ce must be asserted during all seven of the clock cycles. The output state machine discards audio samples from channel pairs that are not enabled for output. The output state machine requires three clock cycles with out_ce asserted to discard an audio sample from a channel pair that is not enabled. The out_ce signal should not be used to clock the output state machine at the audio sample rate because it must be asserted faster than the audio sample rate. The primary use of this signal is to throttle the output state machine clock rate down to a manageable frequency for timing purposes if the clk input is a high frequency clock.

Input Video Streams

The video input signals vary considerably between the single stream module, the multi-stream asynchronous modules, and the multi-stream synchronous modules. Table 21-3, Table 21-4, and Table 21-5) cover these three cases respectively.
Each asynchronous input video stream has the set of four ports shown in Table 21-4. Separate write clocks and write enables are provided. A valid video data word must be present on the strm_n_din port every clock cycle in which strm_n_we is asserted. A TRS/ADF detector is included in these modules for each video stream input.

### Table 21-5: Input Video Streams for Synchronous Multi-Stream Modules

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strm_n_we</td>
<td>In</td>
<td>1</td>
<td>For each input stream, a write enable input is provided. A video data word must be provided on the strm_n_din port every clock cycle in which the write enable is asserted.</td>
</tr>
<tr>
<td>strm_n_din</td>
<td>In</td>
<td>10</td>
<td>Video data input ports for each stream.</td>
</tr>
<tr>
<td>strm_n_full</td>
<td>Out</td>
<td>1</td>
<td>FIFO full outputs for each stream.</td>
</tr>
</tbody>
</table>

Each synchronous input video stream has the set of three ports shown in Table 21-5. All of these ports are synchronous to the clk input. A valid data word must be present on the strm_n_din port every rising edge of clk in which strm_n_we is asserted. A TRS/ADF detector is included in these modules for each video stream input.

### Audio Sample Output Ports

The audio samples are output from the demultiplexer one at a time as they are received and processed. Each audio sample is accompanied by additional signals, providing information about the audio sample.

---

**Table 21-3: Input Video Stream for Single Stream Module (sd_aes_demux_1)**

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>afd</td>
<td>In</td>
<td>1</td>
<td>This input must be asserted during the third word of every 3-word ADF sequence.</td>
</tr>
<tr>
<td>sav</td>
<td>In</td>
<td>1</td>
<td>This input must be asserted during the XYZ word (fourth word) of every 4-word SAV sequence.</td>
</tr>
<tr>
<td>din</td>
<td>In</td>
<td>10</td>
<td>Digital video word input. The sd_aes_demux_1 module does not contain a TRS/ADF detector. The sd_aes_trs_adf_detect module can be used to provide the afd and sav input signals for this module, or these inputs can be driven by the SD-SDI receiver if it provides these signals with the correct timing required by the demultiplexer. A video input word must be provided on din, along with valid afd and sav inputs, on every clock cycle when in_ce is asserted.</td>
</tr>
</tbody>
</table>

**Table 21-4: Input Video Streams for Asynchronous Multi-Stream Modules**

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strm_n_wclk</td>
<td>In</td>
<td>1</td>
<td>For each input stream, a write clock is provided. These clocks are considered to be asynchronous to each other and to main demultiplexer clock (clk).</td>
</tr>
<tr>
<td>strm_n_we</td>
<td>In</td>
<td>1</td>
<td>For each input stream, a write enable input is provided. A video data word must be provided on the strm_n_din port every clock cycle in which the write enable is asserted.</td>
</tr>
<tr>
<td>strm_n_din</td>
<td>In</td>
<td>10</td>
<td>Video data input ports for each stream.</td>
</tr>
<tr>
<td>strm_n_full</td>
<td>Out</td>
<td>1</td>
<td>FIFO full outputs for each stream.</td>
</tr>
</tbody>
</table>
Whenever an audio sample is present on the output, the output_ready signal is asserted High. The audio sample remains on the output port and the output_ready signal remains High until the next rising edge of clk when both out_ce and output_ack are asserted. The output_ack input serves as an output handshaking signal. If this handshaking mechanism is not required, output_ack must be tied High.

No matter how many input video streams the demultiplexer handles, there is just a single output port, and all audio samples from all input video streams are output on this port. When handling multiple video streams, logic and a buffer on the output of the demultiplexer module might be required for each video stream to send each audio sample to the appropriate audio data path.

Table 21-6: Audio Sample Output Ports

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_ready</td>
<td>Out</td>
<td>1</td>
<td>Asserted High when an audio sample is present on the output of the demultiplexer.</td>
</tr>
<tr>
<td>output_ack</td>
<td>In</td>
<td>1</td>
<td>The demultiplexer keeps an audio sample on the output port until both output_ack and out_ce are High during the rising edge of clk.</td>
</tr>
<tr>
<td>sf</td>
<td>Out</td>
<td>1</td>
<td>Indicates which audio subframe the sample came from in the embedded stream. Generally, the chan output is more useful than sf, but this output can provide some debugging information.</td>
</tr>
<tr>
<td>audio</td>
<td>Out</td>
<td>24</td>
<td>Output port for audio sample. For 20-bit audio, the sample is left justified and the LS 4 bits of this port are zeros.</td>
</tr>
<tr>
<td>audio_is_24b</td>
<td>Out</td>
<td>1</td>
<td>High = 24-bit audio sample, Low = 20-bit audio sample.</td>
</tr>
<tr>
<td>stream_out</td>
<td>Out</td>
<td>2/3/4</td>
<td>Indicates which input video stream produced the current audio output sample. This output port is not present on sd_aes_demux_1. The width of this port is dependent on how many input video streams are supported. The port is 2 bits wide for the 4-stream modules, 3 bits wide for the 8-stream modules, and 4 bits wide for the 16-stream modules.</td>
</tr>
<tr>
<td>audio_group</td>
<td>Out</td>
<td>2</td>
<td>Indicates the embedded audio group of the audio sample.</td>
</tr>
<tr>
<td>channel_pair</td>
<td>Out</td>
<td>1</td>
<td>Indicates the channel pair of the audio sample.</td>
</tr>
<tr>
<td>chan</td>
<td>Out</td>
<td>1</td>
<td>Indicates the audio channel (within the channel pair) of the audio sample.</td>
</tr>
<tr>
<td>z</td>
<td>Out</td>
<td>1</td>
<td>AES3 Z frame indicator. This bit is asserted High on the first audio sample of each 192-sample block.</td>
</tr>
<tr>
<td>c</td>
<td>Out</td>
<td>1</td>
<td>AES3 channel status bit.</td>
</tr>
<tr>
<td>u</td>
<td>Out</td>
<td>1</td>
<td>AES3 user data bit.</td>
</tr>
<tr>
<td>v</td>
<td>Out</td>
<td>1</td>
<td>AES3 valid bit. This bit is Low if the audio sample is valid, High if it is not. This bit is the V bit from the audio data packet and is not derived from the channel valid bits in the audio control packet.</td>
</tr>
<tr>
<td>dbn</td>
<td>Out</td>
<td>8</td>
<td>Data block number from the audio data packet that contained the audio sample</td>
</tr>
<tr>
<td>parity_err</td>
<td>Out</td>
<td>1</td>
<td>This output is High if a parity error is detected in the audio sample.</td>
</tr>
</tbody>
</table>

The chan, channel_pair, audio_group, and stream_out ports serve to identify the audio channel, audio channel pair, audio group, and input video stream of the audio sample. The audio_group, channel_pair, and chan bits combine to form a 4-bit audio channel number.
Each video stream can have up to 16 channels. For the multi-channel modules, the stream_out indicates which input video stream contained the audio sample.

All outputs listed in Table 21-6 are controlled by the out_ce clock enable.

**Channel Pair Present Flags**

The demultiplexer produces a set of output flags that indicate which audio channel pairs are detected in each input stream. These flags are "sticky"—that is, a flag is set when an audio sample is detected for the corresponding channel pair. It remains set until cleared by the associated clr_present input.

For each video stream, there is an 8-bit output port with one bit allocated to each of the eight channel pairs in the stream. Each video stream also has an individual clr_present input so the flags can be managed on a per-stream basis. See Table 21-7 and Table 21-8.

The flags are independent of the demux enable status of the channel pair (see the “Channel Pair Demux Control Ports” section). If a channel pair is detected in a video stream, its present flag is set, even if that channel pair is currently being rejected by the demux control mechanism.

These outputs are all controlled by the out_ce clock enable.

<table>
<thead>
<tr>
<th>Table 21-7: Channel Pair Present Flags for sd_aes_demux_1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Name</strong></td>
</tr>
<tr>
<td>clr_present</td>
</tr>
<tr>
<td>present</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 21-8: Channel Pair Present Flags (Multi-Stream Modules)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Name</strong></td>
</tr>
<tr>
<td>clr_present</td>
</tr>
<tr>
<td>strmN_present</td>
</tr>
</tbody>
</table>

Table 21-9 shows how the 8 bits of the present flag output port corresponds to audio groups and channel pairs.

<table>
<thead>
<tr>
<th>Table 21-9: Channel Pair Present Flag Vector Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>strmN_present Bit</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Chapter 21: AES3 Audio Demultiplexer for Standard-Definition Digital Audio

Table 21-9: Channel Pair Present Flag Vector Mapping (Continued)

<table>
<thead>
<tr>
<th>strm_present Bit</th>
<th>Audio Group</th>
<th>Channel Pair</th>
<th>Audio Channels</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>9 &amp; 10</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>11 &amp; 12</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
<td>13 &amp; 14</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>2</td>
<td>15 &amp; 16</td>
</tr>
</tbody>
</table>

Channel Pair Demux Control Ports

The demultiplexer outputs samples only for those audio channel pairs that are enabled. An 8-bit input port allows each audio channel pair to be individually enabled or disabled on a per-stream basis. Each bit in the chpair\_demux\_en port enables one channel pair in the video stream. (See Table 21-10.) Mapping of channel pairs to the bits of the chpair\_demux\_en port is identical to the mapping used for the channel pair present flags shown in Table 21-9.

There is only one 8-bit channel pair demux enable port, even for the multi-stream modules. However, it is possible to have different channel pairs enabled for each different video stream. The demultiplexer indicates which video stream is currently being processed on the current\_stream output port. By changing the data on the chpair\_demux\_en port whenever the current\_stream port changes, different channel pairs can be enabled for each video stream. The simple way to do this is with a mux, as shown in Figure 21-5. Alternatively, the filter bits can be stored in a small distributed dual-port RAM with the current\_stream port providing the read address to the RAM (see Figure 21-6). Of course, if the same filter selection is desired for all input video streams, the current\_stream port can be ignored. The chpair\_demux\_en input value must be updated during the same clock cycle in which the current\_stream port changes.

Table 21-10: Channel Pair Filtering Ports

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chpair_demux_en</td>
<td>In</td>
<td>8</td>
<td>This input port determines which channel pairs are to be demultiplexed and which are not. If the bit for a channel pair is High, the channel pair is demultiplexed and output if present in the video stream. If the bit for a channel pair is Low, the channel pair is not be demultiplexed and no samples for this channel pair are output by the demultiplexer.</td>
</tr>
<tr>
<td>current_stream</td>
<td>Out</td>
<td>2/3/4</td>
<td>This output port indicates which video stream is being processed by the output state machine in the demultiplexer. It is intended to be used to select the appropriate data for the chpair_demux_en input port so that different channel pairs can be enabled for each input video stream. This port is not available on the sd_aes_demux_1 module. The port is 2 bits wide on the 4-stream modules, 3 bits wide on the 8-stream modules, and 4 bits wide on the 16-stream modules.</td>
</tr>
</tbody>
</table>
Input Packet Error Ports

The demultiplexer provides several outputs that indicate the error status of detected audio packets, as shown in Table 21-11.

Table 21-11: Input Packet Status Ports

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>drop_pkt_err</td>
<td>Out</td>
<td>1</td>
<td>This signal is asserted High if an audio or extended data packet is rejected because of an error.</td>
</tr>
<tr>
<td>drop_pkt_code</td>
<td>Out</td>
<td>3</td>
<td>The code on this port is valid when drop_pkt_err is High and indicates the reason the packet was rejected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>000 = No error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>001 = Audio packet rejected due to invalid data count</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>010 = Extended data packet rejected due to invalid data count</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>011 = Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>100 = Audio packet ended prematurely by SAV</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>101 = Audio packet ended prematurely by ADF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>110 = Audio packet rejected due to checksum error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>111 = Reserved</td>
</tr>
<tr>
<td>pkt_cs_err</td>
<td>Out</td>
<td>1</td>
<td>This output is asserted on the last audio sample of an audio packet when a checksum error is detected in the packet.</td>
</tr>
<tr>
<td>pkt_start</td>
<td>Out</td>
<td>1</td>
<td>Asserted on the first audio sample of each audio packet.</td>
</tr>
</tbody>
</table>
The `drop_pkt_err` and `drop_pkt_code` outputs are generated by the input state machine and are controlled by `in_ce`. Once `drop_pkt_err` is asserted, it remains asserted until the next rising edge of `clk` with `in_ce` High.

The `pkt_cs_err` and `pkt_start` outputs are generated by the output state machine and are controlled by `out_ce` and `output_ack`. They are valid until the audio sample present on the output of the demultiplexer is acknowledged by `output_ack` asserted High with `out_ce` High.

Checksum errors in audio data packets are indicated in two different ways, depending on the checksum error policy selected by the `ignore_cs_errs` input.

If `ignore_cs_errs` is High, audio samples from audio data packets with checksum errors are output as normal and the checksum error is indicated by assertion of the `pkt_cs_err` signal during the last audio sample of the packet. It is possible to determine which samples are potentially affected by the checksum error by monitoring both the `pkt_start` and `pkt_cs_err` signals. The `pkt_start` output is asserted when the first audio sample of each packet is output from the demultiplexer. The demultiplexer does not interleave samples from different packets or different streams on the output so once `pkt_start` is asserted, all the audio samples output from the demultiplexer until the next assertion of `pkt_start` come from the same packet. When `pkt_cs_err` is asserted, all samples previously output back to when `pkt_start` was asserted are suspect. See Figure 21-7 for details.

If `ignore_cs_errs` is Low, the demultiplexer rejects any audio packet with a checksum error and the audio samples contained in the packet are not output. These rejected packets are indicated by assertion of `drop_pkt_err` and a code of 110 on `drop_pkt_code`.

![Figure 21-7: `pkt_start` and `pkt_cs_err` Timing](image-url)
Audio Control Packet Ports

The demultiplexer detects and outputs audio control packets. The demultiplexer does not interpret or format the audio control packet data in any way. The words of the audio control packet are output sequentially by the demultiplexer along with some status signals, including a data word count indicating which data word of the control packet is currently on the output port. Only the user data words in the payload portion of the audio control packet are output by the demultiplexer. These ports are all controlled by the input state machine and thus they are controlled by the in_ce clock enable. See Table 21-12.

Control packet output timing is shown in Figure 21-8.

Table 21-12: Audio Control Packet Ports

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctrl_pkt_strobe</td>
<td>Out</td>
<td>1</td>
<td>This output is High whenever a data word from an audio control packet is on</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the ctrl_pkt_data port.</td>
</tr>
<tr>
<td>ctrl_pkt_data</td>
<td>Out</td>
<td>10</td>
<td>The audio control packet data words are output on this port.</td>
</tr>
<tr>
<td>ctrl_pkt_group</td>
<td>Out</td>
<td>2</td>
<td>This output port indicates which audio group the audio control packet belongs to.</td>
</tr>
<tr>
<td>ctrl_pkt_stream</td>
<td>Out</td>
<td>2/3/4</td>
<td>For the multi-stream demultiplexers, this output port indicates which input</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>stream the audio control packet was found in. This port is 2 bits wide for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the 4-stream modules, 3 bits wide for the 8-stream modules, and 4 bits wide</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>for the 16-bit demultiplexer. This port is not present on the sd_aes_demux_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>module.</td>
</tr>
<tr>
<td>ctrl_pkt_word</td>
<td>Out</td>
<td>5</td>
<td>This port indicates the word number of the current user data word on the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ctrl_pkt_data port. This value on this port is zero when the first user data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>word of an audio control packet is on the ctrl_pkt_data port and increments</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>by one with each user data word from the control packet.</td>
</tr>
</tbody>
</table>

Figure 21-8: Audio Control Packet Output Timing
Audio Packet Deletion Ports

The sd_aes_demux_1 module has two ports not present on the multi-stream modules to support the audio packet deletion feature. An input port called del_groups enables audio packet deletion individually by audio group. The port has one input bit for each of the four audio groups. The LSB of this port controls packet deletion for audio group 1. The MSB controls packet deletion for audio group 4. A High on a bit in the del_groups port enables deletion of the audio data packet, audio control packets, and extended data packets for that audio group.

The audio packet deletion function is independent of the audio sample demux feature. It is possible to enable different audio groups for demultiplexing than for deletion because the chpair_demux_en port controls demultiplexing and the del_groups port controls deletion.

The modified video, after audio packet deletion, is output on the video_output port. This port is controlled by the in_ce clock enable so that a video word is output every clock cycle when in_ce is High. There is one clock cycle of latency (counting only those clock cycles with in_ce asserted) added to the video between the video input port (din) and the video_out port. See Table 21-13.

Audio data packets, audio control packets, and extended data packets are deleted by changing their DID words to hex 180. This is the DID value for a deleted packet. The checksum word is also modified so that checksum is correct for the deleted packet. If the video stream contains a SMPTE RP 165 EDH packet, the full-field CRC value in the EDH packet is not correct after audio packet deletion. The video stream should be reprocessed by an EDH processor to calculate and insert new EDH values after audio packet deletion.

Table 21-13: Audio Control Packet Ports

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>del_groups</td>
<td>In</td>
<td>4</td>
<td>A High input bit enables audio packet deletion for the corresponding audio group. Bit 0 controls audio group 1. Bit 3 controls audio group 4.</td>
</tr>
<tr>
<td>video_out</td>
<td>Out</td>
<td>10</td>
<td>The modified video, after audio packet deletion, is output on this port. A video word is output each clock cycle when in_ce is asserted.</td>
</tr>
</tbody>
</table>

Parameters

The reference design has several Verilog parameters or VHDL generics that affect the operation of the demultiplexer.

The parameter/generic called STRICT_EXT_PKT_MODE determines how strictly the demultiplexer enforces the rule that an extended data packet must immediately follow its associated ancillary data packet. If this parameter is 1, the demultiplexer requires that the extended data packet immediately follow the audio data packet with no HANC words in between. If this parameter is 0, the demultiplexer implements a more liberal policy and waits until the end of the HANC interval for a potential extended data packet. In both cases, the extended data packet must be the next ancillary data packet in the HANC interval after the associated audio data packet. The only difference is that extraneous data words can exist between the audio data packet and the extended data packet when the parameter is 0.

Setting this parameter to 0 does slightly affect the timing of the demultiplexer. If there are no extended data packets, then the samples from the last audio data packet in the HANC interval are not output until the end of the HANC interval is detected. If this parameter is
1, then the audio samples from an audio data packet are output immediately as 20-bit samples if an extended data packet does not immediately follow an audio data packet.

The multi-stream demultiplexer modules have another parameter/generic called TIMEOUT_CNTR_BITS that sets the stream timeout period. The timeout counter is a binary counter whose width is set by this parameter. The timeout period always elapses when the timeout counter reaches its terminal count with all bits High. The input stream timeout feature is discussed in further detail in the Multiple synchronous video streams section.

Sample Buffer Size

Audio samples embedded in a video stream arrive in bursts, but are often consumed on the output of a demultiplexer at a fixed rate equal to the audio sample rate. Buffers are required in any audio demultiplexer to smooth out this difference. The SMPTE 272M standard recommends a minimum sample buffer depth of 80 samples per active channel. A total buffer size of 1280 samples is required to support all 16 possible audio channels in one stream and 20K samples for all 16 audio channels in all 16 input video streams. However, few, if any, video streams have 16 active audio channels.

The sd_aes_demux module has an internal audio sample buffer, located between the input and output state machines. This buffer is not designed to meet the SMPTE 272M recommended sample buffer size. Its default size is 512 samples. This is sufficient to handle 16 channels in 16 input streams, if the samples are taken from the demultiplexer at a sufficiently high rate. For example, in NTSC video it is possible to have a maximum of about 82 samples in one HANC. So, in the case of a demultiplexer handling 16 video channels, it must be possible to read 1312 samples (82 * 16), if available, from the demultiplexer during every video line in order to guarantee that the buffer does not overflow. The demultiplexer does not check for overflows of the internal sample buffer, so data is lost if an overflow occurs.

The 512-sample internal buffer is sufficient to meet the SMPTE 272M buffer depth recommendations for six active audio channels. If more active channels are supported by an application then either the internal sample buffer in the demultiplexer must be increased or additional sample buffers outside of the demultiplexer must be used. In a multi-stream application, it is generally preferred to use individual audio sample buffers external to the demultiplexer for each input stream. This is usually more efficient than expanding the internal buffer because the internal buffer carries extra information used internally by the demultiplexer.

All audio samples present in an input video stream, even those that are not output from the demultiplexer because they are not enabled, are stored in the sample buffer.

It is possible to increase the size of the internal sample buffer. The default 512-sample buffer uses two 18-Kbit block RAMs, one for the audio data and one for the extended data. The audio data RAM is configured as 512x36. The extended data RAM is configured as 1024x18, but only one quarter of the extended data RAM is used. Doubling the size of the buffer to 1024 samples requires adding one additional block RAM for the audio samples, but an additional extended data RAM is not required.

A total sample buffer depth of 20K samples is required to meet the SMPTE 272M buffer depth requires for 6 active channels in 16 input video streams. Such a sample buffer requires a total of 60 block RAMs, 50 for the audio data and 10 for the extended data. Again, it might be more efficient to place the sample buffers on the output of the demultiplexer rather than expanding the internal sample buffer.
It is easy to change the width of the buffer address paths in the demultiplexer to accommodate larger buffers. A single Verilog local parameter or VHDL constant called ADR_WIDTH in the sd_aes_demux module specifies the number of address bits used for the sample buffer. The two block RAMs that make up the buffer are located in a separate module called sd_aes_demux_buffer_512. This module can be modified to create a larger sample buffer. A 2K-sample buffer module, sd_aes_demux_buffer_2K, is included as an example of how to implement a larger sample buffer. If this 2K-sample buffer is used, the ADR_WIDTH constant in sd_aes_demux must be set to 11.

FPGA Resource Requirements

Table 21-14 shows the FPGA resources required to implement the audio demultiplexer. The chart shows the single stream, 4-stream, 8-stream, and 16-stream requirements. These results were obtained using ISE 8.1 and XST for synthesis. All results are with the default sample buffer size of 512.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>sd_aes_demux_1</td>
<td>261</td>
<td>195</td>
<td>2</td>
</tr>
<tr>
<td>sd_aes_demux_4_sync</td>
<td>907</td>
<td>491</td>
<td>6</td>
</tr>
<tr>
<td>sd_aes_demux_4_async</td>
<td>1115</td>
<td>727</td>
<td>6</td>
</tr>
<tr>
<td>sd_aes_demux_8_sync</td>
<td>1558</td>
<td>782</td>
<td>10</td>
</tr>
<tr>
<td>sd_aes_demux_8_async</td>
<td>1974</td>
<td>1254</td>
<td>10</td>
</tr>
<tr>
<td>sd_aes_demux_16_sync</td>
<td>2783</td>
<td>1351</td>
<td>18</td>
</tr>
<tr>
<td>sd_aes_demux_16_async</td>
<td>3615</td>
<td>2295</td>
<td>18</td>
</tr>
</tbody>
</table>

Theory of Operation

This section provides the theory of operation for the SD audio demultiplexer reference design.

Overview

The core demultiplexer module, sd_aes_demux, has three main pieces: an input state machine, an output state machine, and an audio sample buffer located between the input and output state machines as shown in Figure 21-9. The sample buffer has data formatting logic on its input and output sides. The input state machine controls the input data formatting logic and writes to the sample buffer. The output state machine controls reads from the sample buffer and the output formatting logic.

The sample buffer is a circular buffer. When the read pointer from the output state machine equals the write pointer from the input state machine, the buffer is empty.

The input state machine processes audio data packets and extended data packets and writes the audio samples into the sample buffer. When an audio data packet and its associated extended data packet have been processed and all audio samples from the packet have been written to the sample buffer, the input state machine changes the write pointer to point to the next empty buffer location past the just written samples. The output state machine sees that the buffer is not empty and reads each audio sample from the sample buffer and outputs it from the demultiplexer.
The input state machine writes all audio samples to the sample buffer, regardless of whether the samples are from channel pairs enabled for output or not. As the output state machine reads each sample from the sample buffer, it checks to see if the sample is from a channel pair that is enabled for output. If the channel pair is enabled, the sample is output from the demultiplexer. If the channel pair is disabled, the sample is discarded.

The sample buffer has two separate RAMs, one holding the audio data and the other holding the extended data. Figure 21-10 shows the format of the data written into the two parts of the sample buffer.
Chapter 21: AES3 Audio Demultiplexer for Standard-Definition Digital Audio

Input State Machine

Figure 21-11, Figure 21-12, and Figure 21-13 show the input state machine’s state diagram.

The input state machine remains idle until the adf input is asserted, indicating the start of an ANC packet. After adf is asserted, the input state machine decodes the DID word of the ANC packet to determine if the packet is an embedded audio packet and, if it is, whether it is an audio data packet, an extended data packet, or an audio control packet, branching to different algorithms for each of these types of packets. If the ANC packet is not an embedded audio packet, the state machine goes back to waiting for another ANC packet.

As the input state machine processes an audio sample from an audio data packet, it formats the three data words that contain the audio sample, along with some additional information like the video stream number, into a 36-bit word that is written to the audio data portion of the sample buffer. At the same time, the input state machine also writes to the extended data portion of the sample buffer, clearing the Ext Data Valid bit to 0 indicating that valid extended data has not yet been written to the sample buffer for that audio sample.

Once the input state machine has finished processing an audio data packet, it waits to see if an extended data packet follows. If there is an extended data packet, the input state machine goes back and writes to only the extended data portion of the sample buffer, setting the Ext Data Valid bit to 1 as it writes the extended data word from the packet into the sample buffer. Each word in the extended data packet contains the extended data for both audio samples of a channel pair. The input state machine writes the extended data for both samples of the channel pair only to the sample buffer location of the first sample of the channel pair.

When the extended data packet has been fully processed or when no extended data packet is found after an audio data packet, the input state machine advances the write pointer, causing the output state machine to begin processing the audio samples. This write pointer is not exactly the same as the write address that the input state machine uses to write audio and extended data to the sample buffer. The write address is manipulated by the input state machine to write all the audio samples from an audio packet and all the extended data from an associated extended data packet to the sample buffer without changing the write pointer. Only when the input state machine is finished processing the audio packet and the extended data packet, does it advance the write pointer, thereby handing the completed audio samples to the output state machine.

Audio control packets are not written to the audio sample buffer. The data words of the audio control packet are output directly to the ctrl_pkt_data port without going through the audio sample buffer.

When the input state machine is processing a packet, it can be interrupted by the assertion of the adf or sav signals. Assertion of the sav signal indicates the end of the horizontal blanking interval. If the state machine is processing a packet and the end of that packet does not occur before the sav signal is asserted, the state machine jumps to the SAV state and aborts the packet. Likewise, if the adf signal is asserted, indicating the start of a new packet, before the end of the current packet is reached, the state machine aborts the packet and jumps to the DIDX state to begin processing the new packet.

If an audio data packet is aborted by the sav or adf signals, the audio samples from the aborted packet are lost and are not output by the demultiplexer. If an extended data packet is aborted, the audio samples from the associated audio data packet are output as 20-bit samples. If an audio control packet is aborted, all the data words received prior to the assertion of the sav or adf signals are output on the ctrl_pkt_data port.
Figure 21-11: Input FSM Audio Data Packet Processing
Chapter 21: AES3 Audio Demultiplexer for Standard-Definition Digital Audio

Figure 21-12: Input FSM Extended Data Packet Processing

Figure 21-13: Input FSM Audio Control Packet Processing
The input state machine has two flags that it uses to keep track of the status of packets. The "inpkt" flag indicates that the state machine is currently processing a packet. This flag is checked in the interrupt states (SAV and DIDX) to see if a packet was interrupted so that the state machine can properly abort the packet by resetting its internal sample buffer write address to the first sample of the aborted packet. The "pend" flag is set when the input state machine has finished processing an audio data packet. It indicates that there are audio samples pending in the sample buffer, waiting for extended data. If the next ANC packet is not an extended data packet or if there are no more ANC packets found in the HANC, the input state machine checks the pend flag to see if there are audio samples pending in the sample buffer. If the pend flag is set, the input state machine updates the write pointer, sending these samples to the output state machine as 20-bit samples without extended data.

Output State Machine

The state diagram for the output state machine is shown in Figure 21-14, page 486.

The output state machine is designed to process audio samples from the sample buffer in pairs, with both samples belonging to the same channel pair. The output state machine waits in the WAIT1 state for the availability of the audio samples in the sample buffer. When the empty signal is deasserted, the state machine moves to the READ1 state and reads the first sample of the channel pair from the sample buffer. It checks to make sure that this is the first sample of the sample pair by examining the sf bit. If the sf bit indicates that this is not the first sample of the sample pair, the output state machine considers itself to be out of sync with the data and it returns to the WAIT1 state, discarding the audio sample. If the sample is the first sample of the channel pair, the state machine advances to the CHECK state.

In the CHECK state the state machine determines whether the sample is from a channel pair that is enabled for output. If the channel pair is not enabled, the state machine returns to the WAIT1 state, discarding the sample. If the channel pair is enabled, the sample is output in the OUT1 state.

This process is repeated for the second sample of the pair in the WAIT2, READ2, and OUT2 states. The state machine does not check to see if the channel pair is enabled for the second sample because it has already determined that the channel pair is enabled.

Output handshaking is implemented in the OUT1 and OUT2 states. In these states, the output_ready signal is asserted and the state machine waits until the output_ack signal is asserted until it proceeds.

Audio Packet Deletion

The audio packet deletion feature found in the single stream demultiplexer is implemented by the sd_aes_pkt_del module. This module is entirely controlled by timing signals from the input state machine. The input state machine tells it when the current input video word contains an audio packet DID. The module checks to see if the audio group is to be deleted, and if so, it replaces the DID with the deleted packet DID value, 180 hex.

The module must also calculate a new checksum value for the packet that it is deleting. The new checksum calculation begins with the new DID value. The module accumulates each video word into the checksum value until the input state machine indicates that the current word is the packet's checksum word. At that point, the module replaces the old checksum value with the newly calculated value.
Input Stream Control Module

The multi-stream wrapper modules include a module to control the multiple input streams called sd_aes_demux_instream_ctrl. This module monitors the empty signals from all of the input FIFOs and selects a FIFO as the input stream source by asserting the FIFO's read enable signal. The module informs the demultiplexer module which stream is currently selected by driving the stream_in port of the demultiplexer.

The input stream controller normally only switches streams when a word is read from the selected FIFO with the SAV flag set indicating the end of the HANC data. When this occurs, the controller selects another stream whose FIFO is not empty to be the new source stream. If a FIFO becomes empty before the SAV, the controller begins a timeout period. If the selected stream does not begin supplying data before the end of the timeout period, the controller switches to another stream. However, the timeout period does not expire if all of the FIFOs are empty. The length of the timeout period is controlled by the parameter/generic TIMEOUT_CNTR_BITS. This value specifies the number of bits used in the timeout counter. The timeout period expires when the timeout counter reaches the terminal count. The terminal count occurs when all of the bits of the counter are '1'. The timeout counter is not enabled by in_ce and so it counts every clock cycle.

Figure 21-14: Output FSM State Diagram
The controller generates a clock enable signal to the demultiplexer. This clock enable is asserted when in_ce is asserted and a non-empty FIFO has been selected. To insure that the demultiplexer sees SAV events and timeout events, the clock enable to the demultiplexer is asserted for one clock cycle when the SAV is read from a FIFO (which usually coincides with the FIFO becoming empty) and when a timeout event occurs.

The controller uses a priority encoder as an arbiter to select the next input stream when switching streams. This priority encoder gives priority to lower numbered input streams. However, because the FIFOs for the streams are written only during the HANC interval, they fill up only once per video line. The demultiplexer must be clocked fast enough that it can process all the HANC data for every input stream every line. So, under these conditions, the priority encoder arbiter allows each and every input stream to be processed every video line. The priority encoder is a separate module and can be replaced with a different arbitration scheme if desired.

**Design Files**

The Xilinx reference designs for the AES3 receiver and transmitter are available at [www.xilinx.com/bvdocs/appnotes/xapp514.zip](http://www.xilinx.com/bvdocs/appnotes/xapp514.zip). Open the ZIP archive and extract file xapp514_aes3-audio-demux.zip.

**Conclusions**

FPGAs are now commonly used to implement SD-SDI receivers. Embedded audio is often included with the video in an SD-SDI stream. The embedded audio must often be demultiplexed from the video so that it can be processed separately.

The reference design described here can detect and demultiplex audio from the SD-SDI video stream. The module can support up to 16 input video streams. This allows audio and video to be processed by the FPGA, increasing the level of integration and decreasing the costs of video applications.
Section VII: Appendixes

Audio/Video Connectivity Solutions for the Broadcast Industry
Appendix A

References

Chapter 2

1. All the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE web site: http://www.smpte.org.

2. The ITU-R BT.601-5 standard can be purchased from the International Telecommunication Union at: http://www.itu.int/itudoc/itu-r/rec/bt/.


5. Xilinx Application Note XAPP250: Clock and Data Recovery with Coded Data Streams by Leonard Dieguez.

6. Xilinx Application Note XAPP224: Data Recovery by Nick Sawyer.


Chapter 3

1. All the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE web site: http://www.smpte.org.

2. The ITU-R BT.601-5 standard can be purchased from the International Telecommunication Union at http://www.itu.int/itudoc/itu-r/rec/bt/.

3. The IEC 1179 standard is now called the IEC 61179 standard and can be purchased from the International Electrotechnical Commission at http://www.iec.ch/webstore.

Chapter 4

1. All the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE web site: http://www.smpte.org.

2. The ITU-R BT.601-5 standard can be purchased from the International Telecommunication Union at http://www.itu.int/itudoc/itu-r/rec/bt/.

3. The IEC 1179 standard is now called the IEC 61179 standard and can be purchased from the International Electrotechnical Commission at http://www.iec.ch/webstore.
Appendix A:


Chapter 5

6. IEC 61179, Helical-scan digital composite video cassette recording system using 19mm magnetic tape, format D2 (International Electrotechnical Commission). This standard can be purchased from the International Electrotechnical Commission at: http://www.iec.ch/webstore.

Chapter 6

2. ITU-R BT.1364, Format of Ancillary Data Signals Carried in Digital Component Studio Interfaces (International Telecommunication Union). The ITU-R BT.1364 standard can be purchased from the International Telecommunication Union at: http://www.itu.int/itudoc/itu-r/rec/bt/.
3. SMPTE 272M-1994, SMPTE Standard for Television - Formatting AES/EBU Audio and Auxiliary Data into Digital Video Ancillary Data Space (The Society of Motion Picture and Television Engineers).

4. RP 168-1993, SMPTE Recommended Practice - Definition of Vertical Interval Switching Point for Synchronous Video Switching (The Society of Motion Picture and Television Engineers).

5. RP 165-1994, SMPTE Recommended Practice - Error Detection Checkwords and Status Flags for Use in Bit-Serial Digital Interfaces for Television (The Society of Motion Picture and Television Engineers).


Chapter 7


2. All SMPTE standards mentioned in this document are published by The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE website at http://www.smpte.org.

3. The ITU-R BT.1304 standard can be purchased from the International Telecommunication Union at http://www.itu.int/itu-t/rec/bt.

Chapter 8

1. The Xilinx SDV demonstration board is available from Cook Technologies (part number CTXIL103). Further information is available at http://www.cook-tech.com.

2. All SMPTE standards mentioned in this document are published by The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE website: http://www.smpte.org.

3. The ITU-R BT.656 standard can be purchased from the International Telecommunication Union at http://www.itu.int/itu-t/rec/bt.

Chapter 9

1. All the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE website: http://www.smpte.org.

2. IEC 60169-8 (1978-01), Radio Frequency Connectors, Part 8: R.F. Coaxial Connectors with Inner Diameter of Outer Conductor 6.5mm (0.256 in) with Bayonet Lock – Characteristic Impedance 50 Ohms (Type BNC)


4. Xilinx publication UG024: RocketIO Transceiver User Guide.

5. The Xilinx SDV Demo board is available from Cook Technologies (part number CTXIL103). Further information is available at http://www.cook-tech.com.
Chapter 10

1. All the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE website: http://www.smpte.org.
2. The Xilinx SDV Demo board is available from Cook Technologies (part number CTXIL103). Further information is available at http://www.cook-tech.com.
4. Xilinx publication UG024: RocketIO Transceiver User Guide.

Chapter 11

1. The Xilinx SDV Demonstration board is available from Cook Technologies (part number CTXIL103). Further information is available at http://www.cook-tech.com.
2. All SMPTE standards mentioned in this document are published by The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE website at http://www.smpte.org.

Chapter 12

1. All the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE web site: http://www.smpte.org.
2. The ITU-R BT.601-5 standard can be purchased from the International Telecommunication Union at http://www.itu.int/itudoc/itu-r/rec/bt/.
3. The Xilinx SDV Demo board is available from Cook Technologies (part number CTXIL103). Further information is available at http://www.cook-tech.com.
4. Xilinx publication UG024: RocketIO Transceiver User Guide.

Chapter 13

1. All the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE web site: http://www.smpte.org.
2. The ITU-R BT.601-5 standard can be purchased from the International Telecommunication Union at http://www.itu.int/itudoc/itu-r/rec/bt/.
3. The Xilinx SDV Demo board is available from Cook Technologies (part number CTXIL103). Further information is available at http://www.cook-tech.com.
5. Xilinx publication UG024: RocketIO Transceiver User Guide.
6. Xilinx Application Note XAPP660: Partial Reconfiguration of RocketIO Pre-emphasis and Differential Swing Control Attributes by Derek R. Curd.
Chapter 14

1. The Xilinx SDV demonstration board is available from Cook Technologies (part number CTXIL103). Further information is available at http://www.cook-tech.com.

2. All SMPTE standards mentioned in this document are published by The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE website: http://www.smpte.org.

3. The ITU-R BT.656 standard can be purchased from the International Telecommunication Union at http://www.itu.int/itudoc/itu-r/rec/bt

4. Data sheets for the ICS664-01 and ICS664-02 parts are available at http://www.icst.com

Chapter 15

1. Xilinx Application Note XAPP671: High Speed Data Recovery Using Asynchronous Data Capture Techniques

Chapter 16

1. All of the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers and can be purchased at the SMPTE web site: http://www.smpte.org

2. The ITU-R BT.601-5 standard can be purchased from the International Telecommunication Union at: http://www.itu.int/itudoc/itu-r/rec/bt/

3. The EIA-189-A standard can be purchased from the Electronic Industries Alliance at: http://www.eia.org

Chapter 17

1. All the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE web site: http://www.smpte.org.

2. The Xilinx SDV Demo board is available from Cook Technologies (part number CTXIL103). Further information is available at http://www.cook-tech.com.

Chapter 18

1. AES3-2003, AES Recommended Practice for Digital Audio Engineering—Serial transmission format for two-channel linearly represented digital audio data, Audio Engineering Society, Inc., http://www.aes.org


3. AES18-1996 (r2002), AES Recommended Practice for Digital Audio Engineering - Format for the user data channel of the AES3 digital audio interface, Audio Engineering Society, Inc.

4. AES5-2003, AES Recommended Practice for Professional Digital Audio—Preferred sampling frequencies for applications employing pulse-code modulation, Audio Engineering Society, Inc.
Chapter 19

1. XAPP224: Data Recovery

Chapter 20

1. AES3-2003, AES Recommended Practice for Digital Audio Engineering—Serial transmission format for two-channel linearly represented digital audio data, Audio Engineering Society, Inc., http://www.aes.org
2. Xilinx publication UG073: XtremeDSP for Virtex-4 FPGAs User Guide.

Chapter 21

1. All of the SMPTE standards referenced in this chapter are available from The Society of Motion Picture and Television Engineers and can be purchased at the SMPTE web site: http://www.smpte.org