Video and Image Processing Suite

User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

UG-VIPSUITE-12.0

Feedback Subscribe

Contents

1. About This MegaCore Function Suite

This document describes the Altera® Video and Image Processing Suite collection of IP cores that ease the development of video and image processing designs. You can use the following IP cores in a wide variety of image processing and display applications.

The Video and Image Processing Suite contains the following MegaCore® functions:

- "2D FIR Filter MegaCore Function" on page 4–1
- "2D Median Filter MegaCore Function" on page 5–1
- "Alpha Blending MegaCore Function" on page 6–1
- "Avalon-ST Video Monitor MegaCore Function" on page 7–1
- "Chroma Resampler MegaCore Function" on page 8–1
- "Clipper MegaCore Function" on page 9–1
- "Clocked Video Input MegaCore Function" on page 10–1
- "Clocked Video Output MegaCore Function" on page 11–1
- "Color Plane Sequencer MegaCore Function" on page 12–1
- "Color Space Converter MegaCore Function" on page 13–1
- "Control Synchronizer MegaCore Function" on page 14–1
- "Deinterlacer MegaCore Function" on page 15–1
- "Deinterlacer II MegaCore Function" on page 16–1
- "Frame Reader MegaCore Function" on page 17–1
- "Frame Buffer MegaCore Function" on page 18–1
- "Gamma Corrector MegaCore Function" on page 19–1
- "Interlacer MegaCore Function" on page 20–1
- "Scaler MegaCore Function" on page 21–1
- "Scaler II MegaCore Function" on page 22–1
- "Switch MegaCore Function" on page 23–1
- "Test Pattern Generator MegaCore Function" on page 24–1
- "Trace System MegaCore Function" on page 25–1

Release Information

Table 1–1 provides information about this release of the Altera Video and Image Processing Suite MegaCore functions.

Table 1–1. Video and Image Processing Suite Release Information

Item	Description		
Version	12.0 (All MegaCore functions)		
Release Date	July 2012		
Ordering Code	IPS-VIDEO (Video and Image Processing Suite)		
Product IDs	00B3 (2D FIR Filter) 00B4 (2D Median Filter) 00B5 (Alpha Blending Mixer) 00D1 (Avalon-ST Video Monitor) 00B1 (Chroma Resampler) 00C8 (Clipper)	00C4 (Clocked Video Input) 00C5 (Clocked Video Output) 00C9 (Color Plane Sequencer) 0003 (Color Space Converter) 00D0 (Control Synchronizer) 00B6 (Deinterlacer) 00EE (Deinterlacer II)	00B2 (Gamma Corrector) 00DC (Interlacer) 00B7 (Scaler) 00E9 (Scaler II) 00CF (Switch) 00CA (Test Pattern Generator) 00ED (Trace System)
Vendor ID(s)	6AF7		

For more information about this release, refer to the *MegaCore IP Library Release Notes and Errata*.

Device Family Support

Table 1–2 defines the device support levels for Altera IP cores.

Table 1–2. Altera IP Core Device Support Levels

FPGA Device Families	HardCopy Device Families
Preliminary support—The IP core is verified with preliminary timing models for this device family. The IP core meets all functional requirements, but might still be undergoing timing analysis for the device family. It can be used in production designs with caution.	**HardCopy Companion**—The IP core is verified with preliminary timing models for the HardCopy companion device. The IP core meets all functional requirements, but might still be undergoing timing analysis for the HardCopy device family. It can be used in production designs with caution.
Final support—The IP core is verified with final timing models for this device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.	**HardCopy Compilation**—The IP core is verified with final timing models for the HardCopy device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.

Table 1–3 lists the level of support offered by the SDI MegaCore function for each Altera device family.

Table 1–3. Device Family Support (Part 1 of 2)

Device Family	Support
Arria® GX	Final
Arria II GX	Final
Arria II GZ	Final

Table 1–3. Device Family Support (Part 2 of 2)

Device Family	Support
Arria V	Refer to the What's New in Altera IP page of the Altera website.
Cyclone® II	Final
Cyclone III	Final
Cyclone III LS	Final
Cyclone IV E	Final
Cyclone IV GX	Final
Cyclone V	Refer to the What's New in Altera IP page of the Altera website.
HardCopy II	HardCopy Compilation
HardCopy III	HardCopy Compilation
HardCopy IV E/GX	HardCopy Compilation
Stratix®	Final
Stratix II	Final
Stratix III	Final
Stratix IV	Final
Stratix V	Preliminary
Other device families	No support

Features

The following features are common to all of the Video and Image Processing Suite MegaCore functions:

- Common Avalon® Streaming (Avalon-ST) interface and Avalon-ST Video protocol
- Avalon Memory-Mapped (Avalon-MM) interfaces for run-time control input and connections to external memory blocks
- Easy-to-use parameter editor for parameterization and hardware generation
- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators
- Support for OpenCore Plus evaluation
- Qsys ready
- Extensively tested and verified using comprehensive regression tests to ensure quality and correctness

Stall Behavior and Error Recovery

The Video and Image Processing Suite MegaCore functions do not continuously process data. Instead, they use flow-controlled Avalon-ST interfaces, which allow them to stall the data while they perform internal calculations.

During control packet processing, the MegaCore functions might stall frequently and read/write less than once per clock cycle. During data processing, the MegaCore functions generally process one input/output per clock cycle. There are, however, some stalling cycles. Typically, these are for internal calculations between rows of image data and between frames/fields.

When stalled, the MegaCore function signals that it is not ready to receive or produce data. The time spent in the stalled state varies between MegaCore functions and their parameterizations. In general, it is a few cycles between rows and a few more between frames.

If data is not available at the input when required, all of the MegaCore functions stall, and thus do not output data. With the exceptions of the Deinterlacer and Frame Buffer in double or triple-buffering mode, none of the MegaCore functions ever overlap the processing of consecutive frames. The first sample of frame $F + 1$ is not input until after the last sample of frame F has been output.

When an `endofpacket` signal is received unexpectedly (early or late), the MegaCore function recovers from the error and prepares itself for the next valid packet (control or data).

For more information about the stalling, throughput, and error recovery of each MegaCore function, refer to the “Stall Behavior and Error Recovery” section of the respective MegaCore Function chapter in this user guide.

Design Example

A provided design example offers a starting point to quickly understand the Altera video design methodology, enabling you to build full video processing systems on an FPGA.

For more information about this design example, refer to *AN427: Video and Image Processing Up Conversion Example Design*.

Performance and Resource Utilization

This section shows typical expected performance for the Video and Image Processing Suite MegaCore functions with the Quartus® II software targeting Cyclone IV GX and Stratix V devices.

Cyclone IV GX devices use combinational look-up tables (LUTs) and logic registers; Stratix V devices use combinational adaptive look-up tables (ALUTs) and logic registers.

2D FIR Filter

Table 1–4 lists the performance figures for the 2D FIR Filter.

Table 1–4. 2D FIR Filter Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Edge detecting 3×3 asymmetric filter, working on 352×288 8-bit R’G’B’, using 3 bit coefficients.								
Cyclone IV GX (1)	984	1,341	16,896	4	—	9	—	207.9
Stratix V (2)	777	987	16,896	—	2	—	9	302.48
Smoothing 3×3 symmetric filter, working on 640×480 8-bit R’G’B’, using 9 bit coefficients.								
Cyclone IV GX (1)	986	1,313	30,720	4	—	6	—	205
Stratix V (2)	771	958	30,720	—	2	—	3	326.9
Sharpening 5×5 symmetric filter, working on 640×480 in 8-bit R’G’B’, using 9 bit coefficients.								
Cyclone IV GX (1)	1,894	2,412	61,440	8	—	12	—	197.36
Stratix V (2)	1,424	1,804	61,440	—	4	—	6	290.36
Smoothing 7×7 symmetric filter, working on 1,280×720 in 10-bit R’G’B’, using 15 bit coefficients.								
Cyclone IV GX (1)	3,725	4,681	230,400	30	—	20	—	178.25
Stratix V (2)	2,648	3,612	230,400	—	12	—	10	239.58

Notes to Table 1–4:

(1) EP4CGX22BF14C6 devices.

(2) EP4CGX22BF14C6 devices.

2D Median Filter

Table 1–5 lists the performance figures for the 2D Median Filter.

Table 1–5. 2D Median Filter Performance (Part 1 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
3×3 median filtering HDTV 720 pixel monochrome video.								
Cyclone IV GX (1)	1,567	1,724	25,600	6	—	—	—	245.64
Stratix V (2)	1,011	1,200	25,600	—	2	—	—	353.61
Median filtering 64×64 pixel R’G’B frames using a 3×3 kernel of pixels.								
Cyclone IV GX (1)	1,529	1,674	3,072	2	—	—	—	272.78

Table 1–5. 2D Median Filter Performance (Part 2 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Stratix V (2)	984	1,154	3,072	—	2	—	—	364.7
Median filtering 352×288 pixel two color frames using a 5×5 kernel of pixels.								
Cyclone IV GX (1)	5,402	5,667	28,160	8	—	—	—	235.07
Stratix V (2)	2,698	3,832	28,160	—	4	—	—	274.35
7×7 median filtering 352×288 pixel monochrome video.								
Cyclone IV GX (1)	10,801	11,192	16,896	6	—	—	—	216.59
Stratix V (2)	4,863	7,296	16,896	—	6	—	—	262.61

Notes to Table 1–5:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Alpha Blending Mixer

Table 1–6 lists the performance figures for the Alpha Blending Mixer.

Table 1–6. Alpha Blending Mixer Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Alpha blending an on-screen display within a region of 1,024×768 pixel 10-bit Y'CbCr 4:4:4 video. Alpha blending is performed using 16 levels of opacity from fully opaque to fully translucent.								
Cyclone IV GX (1)	1,068	1,236	752	1	—	4	—	200.72
Stratix V (2)	748	732	752	—	1	—	2	324.36
Drawing a picture-in-picture window over the top of a 128×128 pixel background image in 8-bit R'G'B' color.								
Cyclone IV GX (1)	1,814	2,143	752	1	—	—	—	180.51
Stratix V (2)	1,368	1,283	752	—	1	—	—	294.2
Rendering two images over 352×240 pixel background 8-bit R'G'B' video.								
Cyclone IV GX (1)	842	941	752	1	—	—	—	217.91
Stratix V (2)	597	529	752	—	1	—	—	309.98
Using alpha blending to composite three layers over the top of PAL resolution background video in 8-bit monochrome. Alpha blending is performed using 256 levels of opacity from fully opaque to fully translucent.								
Cyclone IV GX (1)	1,162	1,291	752	1	—	6	—	219.88
Stratix V (2)	824	709	752	—	1	—	6	317.86

Notes to Table 1–6:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Avalon-ST Video Monitor

Table 1–7 lists the performance figures for the Avalon-ST Video Monitor.

Table 1–7. Avalon-ST Video Monitor Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Cyclone IV GX (1)	885	870	5,856	11	—	—	—	237.87
Stratix V (2)	468	880	5,536	—	9	—	—	363.50

Notes to Table 1–7:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Chroma Resampler

Table 1–8 lists the performance figures for the Chroma Resampler.

Table 1–8. Chroma Resampler Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Upsampling from 4:2:0 to 4:4:4 with a parallel data interface and run time control of resolutions up to extended graphics array format (XGA - 1024x768). This parameterization uses luma-adaptive filtering on the horizontal resampling and nearest-neighbor on the vertical resampling.								
Cyclone IV GX (1)	1,778	2,353	16,384	4	—	—	—	158.76
Stratix V (2)	1,309	1,783	16,384	—	4	—	—	294.81
Upsamping from 4:2:2 to 4:4:4 with a sequential data interface at quarter common intermediate format (QCIF - 176x144) using luma-adaptive filtering.								
Cyclone IV GX (1)	956	1,120	—	0	—	—	—	231.27
Stratix V (2)	653	818	—	—	0	—	—	366.43
Downsampling from 4:4:4 to 4:2:0 with a parallel data interface and run-time control of resolutions up to XGA (1024x768). The parameterization uses anti-aliasing filtering on the horizontal resampling and nearest-neighbor on the vertical.								
Cyclone IV GX (1)	1,340	1,785	4,096	1	—	—	—	176.03
Stratix V (2)	840	1,371	4,096	—	1	—	—	311.82
Downsamping from 4:4:4 to 4:2:2 with a sequential data interface at quarter common intermediate format (QCIF - 176x144) using an anti-aliasing filter.								
Cyclone IV GX (1)	785	872	—	0	—	—	—	210.13
Stratix V (2)	406	560	—	—	0	—	—	323.31

Notes to Table 1–8:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Clipper

Table 1–9 lists the performance figures for the Clipper.

Table 1–9. Clipper Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		fMAX (MHz)
			Bits	M9K	M20K	(9x9)	(18x18)	
A 1080p60-compatible clipper with a clipping window that has fixed offsets from the size of the input frames.								
Cyclone IV GX (1)	596	664	—	0	—	—	—	191.28
Stratix V (2)	452	453	—	—	0	—	—	313.77
A 100×100 pixel clipper with a clipping window that is a rectangle from the input frames.								
Cyclone IV GX (1)	430	509	—	0	—	—	—	217.72
Stratix V (2)	355	275	—	—	0	—	—	321.13
A 1080p60-compatible clipper with a run-time interface which uses offsets to set the clipping window.								
Cyclone IV GX (1)	661	817	—	0	—	—	—	194.33
Stratix V (2)	522	599	—	—	0	—	—	298.78
A 100×100 pixel clipper with a run-time interface which uses a rectangle to set the clipping window.								
Cyclone IV GX (1)	577	697	—	0	—	—	—	207.04
Stratix V (2)	470	446	—	—	0	—	—	334.56

Notes to Table 1–9:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Clocked Video Input

Table 1–10 lists the performance figures for the Clocked Video Input.

Table 1–10. Clocked Video Input Performance (Part 1 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory				fMAX (MHz)
			Bits	M9K	M20K	MLAB Bits	
Converts DVI 1080p60 clocked video to Avalon-ST Video.							
Cyclone IV GX (1)	377	483	51,200	7	—	—	133.24
Stratix V (2)	296	376	51,200	—	3	—	206.57
Converts PAL clocked video to Avalon-ST Video.							
Cyclone IV GX (1)	361	461	18,432	3	—	—	134.88
Stratix V (2)	297	353	18,432	—	1	—	225.17
Converts SDI 1080i60 clocked video to Avalon-ST Video.							
Cyclone IV GX (1)	403	552	43,008	6	—	—	116.36
Stratix V (2)	322	426	43,008	—	3	40	194.36
Converts SDI 1080p60 clocked video to Avalon-ST Video.							
Cyclone IV GX (1)	395	549	43,008	6	—	—	116.36

Table 1–10. Clocked Video Input Performance (Part 2 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory				f_{MAX} (MHz)
			Bits	M9K	M20K	MLAB Bits	
Stratix V (2)	310	426	43,008	—	3	40	198.61

Notes to Table 1–10:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Clocked Video Output

Table 1–11 lists the performance figures for the Clocked Video Output.

Table 1–11. Clocked Video Output Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory				f_{MAX} (MHz)
			Bits	M9K	M20K	MLAB Bits	
Converts Avalon-ST Video to DVI 1080p60 clocked video.							
Cyclone IV GX (1)	276	292	51,200	7	—	—	138.81
Stratix V (2)	188	148	51,200	—	3	—	199.24
Converts Avalon-ST Video to PAL clocked video.							
Cyclone IV GX (1)	297	307	18,432	3	—	—	134.46
Stratix V (2)	240	144	18,432	—	1	—	213.68
Converts Avalon-ST Video to SDI 1080i60 clocked video.							
Cyclone IV GX (1)	320	325	43,008	6	—	—	138.10
Stratix V (2)	250	152	43,008	—	3	—	220.17
Converts Avalon-ST Video to SDI 1080p60 clocked video.							
Cyclone IV GX (1)	316	326	43,008	6	—	—	146.93
Stratix V (2)	241	152	43,008	—	3	—	216.54

Notes to Table 1–11:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Color Plane Sequencer

Table 1–12 lists the performance figures for the Color Plane Sequencer.

Table 1–12. Color Plane Sequencer Performance (Part 1 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Rearranging a channels in sequence 4:2:2 stream, from Cb,Y,Cr,Y to Y,Cb,Y,Cr. 8 bit data.								
Cyclone IV GX (1)	284	339	—	0	—	—	—	271.08
Stratix V (2)	213	240	—	—	0	—	—	397.3
Joining a single channel luminance stream and a channels in sequence horizontally half-subsampled chrominance stream to a single 4:2:2 channels in sequence output stream. 8 bit data.								
Cyclone IV GX (1)	388	464	—	0	—	—	—	230.20

Table 1–12. Color Plane Sequencer Performance (Part 2 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Stratix V [(2)]	272	313	—	—	0	—	—	385.21
Splitting a 4:2:2 stream from 2 channels in parallel to a single channel luminance output stream and a channels in sequence horizontally half-subsampled chrominance output stream. 8 bit data.								
Cyclone IV GX [(1)]	439	516	—	0	—	—	—	223.56
Stratix V [(2)]	325	342	—	—	0	—	—	353.98
Rearranging 3 channels in sequence to 3 channels in parallel. 8 bit data.								
Cyclone IV GX [(1)]	231	315	—	0	—	—	—	270.64
Stratix V [(2)]	174	249	—	—	0	—	—	387.90

Notes to Table 1–12:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Color Space Converter

Table 1–13 lists the performance figures for the Color Space Converter.

Table 1–13. Color Space Converter Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Converting 1,080 pixel 10-bit Studio R’G’B’ to HDTV Y’CbCr using 18-bit coefficients and 27-bit summands.								
Cyclone IV GX [(1)]	383	557	—	0	—	6	—	244.56
Stratix V [(2)]	311	467	—	—	0	—	3	351.25
Converting 1024×768 14-bit Y’UV to Computer R’G’B’ using 18-bit coefficients and 15-bit summands.								
Cyclone IV GX [(1)]	445	667	—	0	—	6	—	255.69
Stratix V [(2)]	360	564	—	—	0	—	3	360.62
Converting 640×480 8-bit SDTV Y’CbCr to Computer R’G’B’ using 9-bit coefficients and 16-bit summands, color planes in parallel.								
Cyclone IV GX [(1)]	549	899	—	0	—	9	—	247.71
Stratix V [(2)]	473	818	—	—	0	—	9	372.3
Converting 720×576 8-bit Computer R’G’B’ to Y’UV using 9-bit coefficients and 8-bit summands.								
Cyclone IV GX [(1)]	322	447	—	0	—	3	—	280.11
Stratix V [(2)]	259	359	—	—	0	—	3	400

Notes to Table 1–13:

(1) EP4CGX22BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Control Synchronizer

Table 1–14 lists the performance figures for the Control Synchronizer.

Table 1–14. Control Synchronizer Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_MAX (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Synchronizing the configuration of other MegaCore functions with 2 channels in parallel, and the maximum number of control data entries that can be written to other cores is 3.								
Cyclone IV GX (1)	609	805	—	0	—	—	—	209.69
Stratix V (2)	408	574	—	—	0	—	—	380.37
Synchronizing the configuration of other MegaCore functions with 3 channels in parallel, and the maximum number of control data entries that can be written to other cores is 3.								
Cyclone IV GX (1)	624	839	—	0	—	—	—	212.27
Stratix V (2)	418	604	—	—	0	—	—	378.79
Synchronizing the configuration of other MegaCore functions with 3 channels in parallel, and the maximum number of control data entries that can be written to other cores is 10.								
Cyclone IV GX (1)	1,256	1,582	—	0	—	—	—	211.77
Stratix V (2)	697	1,052	—	—	0	—	—	364.03
Synchronizing the configuration of other MegaCore functions with 3 channels in sequence, and the maximum number of control data entries that can be written to other cores is 3.								
Cyclone IV GX (1)	594	750	—	0	—	—	—	212.18
Stratix V (2)	398	398	—	—	0	—	—	377.93

Notes to Table 1–23:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Deinterlacer

Table 1–15 lists the performance figures for the Deinterlacer.

Table 1–15. Deinterlacer Performance (Part 1 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_MAX (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Deinterlacing 64×64 pixel 8-bit R’G’B’ frames using the bob algorithm with scanline duplication.								
Cyclone IV GX (1)	525	582	17,280	4	—	—	—	204.83
Stratix V (2)	389	332	17,280	—	2	—	—	294.55
Deinterlacing with scanline interpolation using the bob algorithm working on 352×288 pixel 12-bit Y’CbCr 4:2:2 frames.								
Cyclone IV GX (1)	632	704	14,400	3	—	—	—	202.18
Stratix V (2)	454	398	14,400	—	1	—	—	303.58
Deinterlacing PAL (720×576) with 8-bit Y'CbCr 4:4:4 color using the motion-adaptive algorithm.								
Cyclone IV GX (1)	6,992	9,697	157,372	37	—	4	—	135.15
Stratix V (2)	5,188	7,879	157,372	—	24	—	2	219.68
Deinterlacing HDTV 1080i resolution with 12-bit Y’CbCr 4:4:4 color using the weave algorithm.								

Table 1–15. Deinterlacer Performance (Part 2 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		fMAX (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Cyclone IV GX [(1)]	2,790	3,313	2,566	14	—	—	—	176.03
Stratix V [(2)]	2,144	2,299	2,566	—	14	—	—	283.61

Notes to Table 1–15:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Deinterlacer II

Table 1–16 lists the performance figures for the Deinterlacer II.

Table 1–16. Deinterlacer II Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		fMAX (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Deinterlacing PAL (720×576) with 8-bit Y'CbCr 4:4:4 color using the motion-adaptive algorithm.								
Cyclone IV GX [(1)]	4,990	4,821	48,398	72	—	4	—	153.23
Stratix V [(2)]	3,696	4,036	48,244	—	59	—	2	203.46
Deinterlacing PAL (720×576) with 8-bit Y'CbCr 4:4:4 color using the motion-adaptive high quality algorithm.								
Cyclone IV GX [(1)]	10,766	7,869	50,356	83	—	8	—	153.59
Stratix V [(2)]	8,252	7,010	50,594	—	70	—	4	203.67

Notes to Table 1–16:

(1) EP4CGX22CF19C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Frame Buffer

Table 1–17 lists the performance figures for the Frame Buffer.

Table 1–17. Frame Buffer Performance (Part 1 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		fMAX (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Double-buffering XGA (1024×768) 8-bit RGB with a sequential data interface.								
Cyclone IV GX [(1)]	1,489	1,942	7,936	4	—	—	—	175.59
Stratix V [(2)]	1,100	1,487	7,936	—	4	—	—	281.69
Triple-buffering VGA (640×480) 8-bit RGB with a parallel data interface.								
Cyclone IV GX [(1)]	1,287	1,663	7,168	4	—	—	—	170.94
Stratix V [(2)]	891	1,354	7,168	—	4	—	—	321.65
Triple-buffering VGA (640×480) 8-bit RGB buffering up to 32 large Avalon-ST Video packets into RAM.								
Cyclone IV GX [(1)]	2,292	3,881	11,168	4	—	—	—	166.67
Stratix V [(2)]	1,285	3,291	11,168	—	4	—	—	301.11
Triple-buffering 720×576 8-bit RGB with sequential data interface and run-time control interface.								

Table 1–17. Frame Buffer Performance (Part 2 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Cyclone IV GX *(1)*	1,286	1,684	8,192	5	—	—	—	179.21
Stratix V *(2)*	932	1,314	8,192	—	5	—	—	329.92

Notes to Table 1–17:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Gamma Corrector

Table 1–18 lists the performance figures for the Gamma Corrector.

Table 1–18. Gamma Corrector Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Gamma correcting 1,080 pixel one color 10-bit data.								
Cyclone IV GX *(1)*	244	271	10,260	3	—	—	—	229.89
Stratix V *(2)*	166	153	10,260	—	1	—	—	369.69
Gamma correcting 720×576 one color 10-bit data.								
Cyclone IV GX *(1)*	244	271	10,260	3	—	—	—	229.89
Stratix V *(2)*	166	153	10,260	—	13	—	—	369.69
Gamma correcting 128×128 three color 8-bit data.								
Cyclone IV GX *(1)*	225	236	2,064	1	—	—	—	242.01
Stratix V *(2)*	157	137	2,064	—	1	—	—	352.11
Gamma correcting 64×64 three color 8-bit data.								
Cyclone IV GX *(1)*	225	236	2,064	1	—	—	—	242.01
Stratix V *(2)*	157	137	2,064	—	1	—	—	352.11

Notes to Table 1–18:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Interlacer

Table 1–15 lists the performance figures for the Interlacer.

Table 1–19. Interlacer Performance (Part 1 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Interlacing 720p 8-bit video, 3 channels over a parallel interface.								
Cyclone IV GX *(1)*	424	515	2,944	2	—	—	—	133.65
Stratix V *(2)*	310	428	—	—	0	—	—	368.46
Interlacing 720p 10-bit video, 2 channels over a sequential interface.								

Table 1–19. Interlacer Performance (Part 2 of 2)

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Cyclone IV GX (1)	431	501	—	0	—	—	—	246.67
Stratix V (2)	280	347	—	—	0	—	—	330.14
Interlacing 1080p 10-bit video, 2 channels over a parallel interface.								
Cyclone IV GX (1)	461	548	—	0	—	—	—	242.42
Stratix V (2)	302	400	—	—	0	—	—	333.78
Interlacing 1080p 10-bit video, 2 channels over a parallel interface, with run-time interlacing control.								
Cyclone IV GX (1)	528	613	—	0	—	—	—	231.75
Stratix V (2)	356	444	—	—	0	—	—	338.41

Notes to Table 1–19:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Scaler

Table 1–20 lists the performance figures for the Scaler.

Table 1–20. Scaler Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		f_{MAX} (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Scaling 640×480, 8-bit, three color data up to 1,024×768 with linear interpolation. This can be used to convert video graphics array format (VGA - 640×480) to video electronics standards association format (VESA - 1024×768).								
Cyclone IV GX (1)	866	1,061	30,720	6	—	4	—	203.67
Stratix V (2)	634	681	30,720	—	6	—	4	336.47
Scaling R’G’B’ QCIF to common intermediate format (CIF) with no interpolation.								
Cyclone IV GX (1)	411	485	4,224	3	—	—	—	248.2
Stratix V (2)	295	297	4,224	—	3	—	—	354.99
Scaling up or down between NTSC standard definition and 1080 pixel high definition using 10 taps horizontally and 9 vertically. Resolution and coefficients are set by a run-time control interface.								
Cyclone IV GX (1)	4,048	5,243	417,456	—	—	19	—	182.95
Stratix V (2)	2,317	3,418	417,152	—	—	—	19	227.63
Scaling NTSC standard definition (720x480) RGB to high definition 1080p using a bicubic algorithm.								
Cyclone IV GX (1)	1,728	2,078	69,444	14	—	8	8	203.46
Stratix V (2)	1,030	1,225	69,408	—	8	—	8	309.98

Notes to Table 1–20:

(1) EP4CGX22BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Scaler II

Table 1–21 lists the performance figures for the Scaler II.

Table 1–21. Scaler II Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		fMAX (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Scaling 640×480, 8-bit, three color data up to 1,024×768 with linear interpolation. This can be used to convert video graphics array format (VGA - 640×480) to video electronics standards association format (VESA - 1024×768).								
Cyclone IV GX (1)	977	1,178	30,816	5	—	4	—	181.52
Stratix V (2)	780	805	30,816	—	3	—	5	279.96
Scaling up or down between NTSC standard definition and 1080 pixel high definition using 10 taps horizontally and 9 vertically. Resolution and coefficients are set by a run-time control interface.								
Cyclone IV GX (1)	2,839	4,016	417,936	76	—	29	—	156.37
Stratix V (2)	1,698	3,101	417,936	—	40	—	10	326.37
Scaling NTSC standard definition (720x480) RGB to high definition 1080p using a bicubic algorithm.								
Cyclone IV GX (1)	1,397	1,909	70,512	13	—	12	—	167.34
Stratix V (2)	964	1,407	70,512	—	7	—	4	349.53

Notes to Table 1–21:

(1) EP4CGX22CF19C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Switch

Table 1–22 lists the performance figures for the Switch.

Table 1–22. Switch Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		fMAX (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
2 input, 2 output switch with alpha channels disabled and doing three colors in sequence.								
Cyclone IV GX (1)	122	155	—	0	—	—	—	328.95
Stratix V (2)	80	127	—	—	0	—	—	527.43
12 input, 12 output switch with alpha channels enabled and doing three colors in parallel.								
Cyclone IV GX (1)	6,177	6,884	—	0	—	—	—	165.34
Stratix V (2)	4,553	2,547	—	—	0	—	—	231.70

Notes to Table 1–22:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Test Pattern Generator

Table 1–23 lists the performance figures for the Test Pattern Generator.

Table 1–23. Test Pattern Generator Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory			DSP Blocks		fMAX (MHz)
			Bits	M9K	M20K	(9×9)	(18×18)	
Producing a 400××200, 8-bit 4:2:0 Y'Cb'Cr' stream with a parallel data interface.								
Cyclone IV GX (1)	159	168	192	2	—	—	—	315.06
Stratix V (2)	152	115	192	—	2	—	—	500.00
Producing a 640×480, 8-bit R'G'B' stream with a sequential data interface.								
Cyclone IV GX (1)	214	217	192	3	—	—	—	315.06
Stratix V (2)	161	117	192	—	3	—	—	490.44
Producing a 720×480, 10-bit 4:2:2 Y'Cb'Cr' interlaced stream with a sequential data interface.								
Cyclone IV GX (1)	261	263	240	3	—	—	—	252.33
Stratix V (2)	240	135	240	—	3	—	—	482.39
Producing a 1920×1080, 10-bit 4:2:2 Y'Cb'Cr' interlaced stream with a parallel data interface. The resolution of the pattern can be changed using the run-time control interface.								
Cyclone IV GX (1)	338	370	304	4	—	—	—	262.12
Stratix V (2)	261	209	304	—	4	—	—	374.25

Notes to Table 1–23:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

Trace System

Table 1–24 lists the performance figures for the Trace System.

Table 1–24. Trace System Performance

Device Family	Combinational LUTs/ALUTs	Logic Registers	Memory Bits	DSP Blocks	fMAX (MHz)
Cyclone IV GX (1) JTAG mode	2,236	2,104	269,312	—	246
Cyclone IV GX (1) USB mode	2,185	2,159	272,384	—	173
Stratix V (2) JTAG mode	1,569	2,106	269,312	—	387
Stratix V (2) USB mode	1,503	2,157	272,384	—	316

Notes to Table 1–24:

(1) EP4CGX15BF14C6 devices.

(2) 5SGXEA7H3F35C3 devices.

2. Getting Started with Altera IP Cores

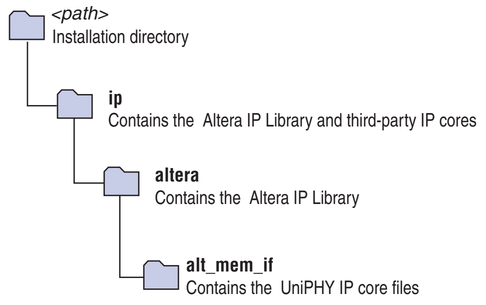
This chapter provides a general overview of the Altera IP core design flow to help you quickly get started with any Altera IP core. The Altera IP Library is installed as part of the Quartus II installation process. You can select and parameterize any Altera IP core from the library. Altera provides an integrated parameter editor that allows you to customize IP cores to support a wide variety of applications. The parameter editor guides you through the setting of parameter values and selection of optional ports. The following sections describe the general design flow and use of Altera IP cores.

Installation and Licensing

The Altera IP Library is distributed with the Quartus II software and downloadable from the Altera website (www.altera.com).

Figure 2–1 shows the directory structure after you install an Altera IP core, where <*path*> is the installation directory. The default installation directory on Windows is **C:\altera\**<*version number*>; on Linux it is **/opt/altera**<*version number*>**.**

Figure 2–1. IP core Directory Structure



<path>
Installation directory
ip
Contains the Altera IP Library and third-party IP cores
altera
Contains the Altera IP Library
alt_mem_if
Contains the UniPHY IP core files

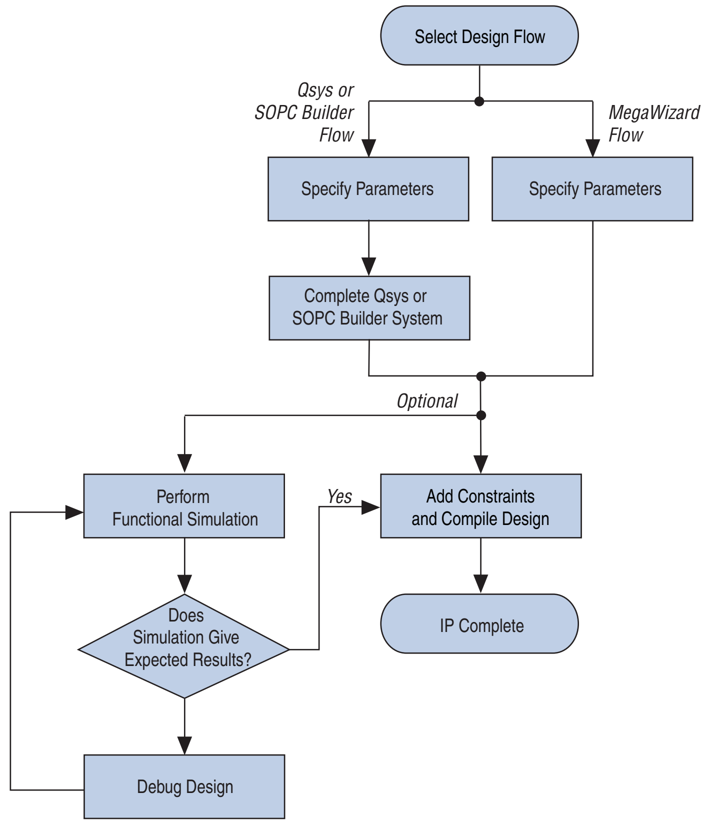
You can evaluate an IP core in simulation and in hardware until you are satisfied with its functionality and performance. Some IP cores require that you purchase a license for the IP core when you want to take your design to production. After you purchase a license for an Altera IP core, you can request a license file from the Altera Licensing page of the Altera website and install the license on your computer. For additional information, refer to *Altera Software Installation and Licensing*.

Design Flows

You can use the following flow(s) to parameterize Altera IP cores:

- MegaWizard Plug-In Manager Flow

Figure 2–2. Design Flows *(1)*


Select Design Flow
Qsys or SOPC Builder Flow
MegaWizard Flow
Specify Parameters
Specify Parameters
Complete Qsys or SOPC Builder System
Optional
Perform Functional Simulation
Yes
Add Constraints and Compile Design
Does Simulation Give Expected Results?
IP Complete
Debug Design

Note to Figure 2–2:

(1) Altera IP cores may or may not support the Qsys and SOPC Builder design flows.

The MegaWizard Plug-In Manager flow offers the following advantages:

- Allows you to parameterize an IP core variant and instantiate into an existing design
- For some IP cores, this flow generates a complete example design and testbench.

MegaWizard Plug-In Manager Flow

The MegaWizard Plug-In Manager flow allows you to customize your IP core and manually integrate the function into your design.

Specifying Parameters

To specify IP core parameters with the MegaWizard Plug-In Manager, follow these steps:

1. Create a Quartus II project using the **New Project Wizard** available from the File menu.
2. In the Quartus II software, launch the **MegaWizard Plug-in Manager** from the Tools menu, and follow the prompts in the MegaWizard Plug-In Manager interface to create or edit a custom IP core variation.
3. To select a specific Altera IP core, click the IP core in the **Installed Plug-Ins** list in the MegaWizard Plug-In Manager.
4. Specify the parameters on the **Parameter Settings** pages. For detailed explanations of these parameters, refer to the "Parameter Settings" section of the respective MegaCore Function chapter in this user guide or the "*Documentation*" button in the MegaWizard parameter editor.

   Some IP cores provide preset parameters for specific applications. If you wish to use preset parameters, click the arrow to expand the **Presets** list, select the desired preset, and then click **Apply**.

5. If the IP core provides a simulation model, specify appropriate options in the wizard to generate a simulation model.

   Altera IP supports a variety of simulation models, including simulation-specific IP functional simulation models and encrypted RTL models, and plain text RTL models. These are all cycle-accurate models. The models allow for fast functional simulation of your IP core instance using industry-standard VHDL or Verilog HDL simulators. For some cores, only the plain text RTL model is generated, and you can simulate that model.

   For more information about functional simulation models for Altera IP cores, refer to *Simulating Altera Designs* in volume 3 of the *Quartus II Handbook*.

   CAUTION Use the simulation models only for simulation and not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.

6. If the parameter editor includes **EDA** and **Summary** tabs, follow these steps:
   a. Some third-party synthesis tools can use a netlist that contains the structure of an IP core but no detailed logic to optimize timing and performance of the design containing it. To use this feature if your synthesis tool and IP core support it, turn on **Generate netlist**.
   b. On the **Summary** tab, if available, select the files you want to generate. A gray checkmark indicates a file that is automatically generated. All other files are optional.

   If file selection is supported for your IP core, after you generate the core, a generation report (<*variation name*>**.html)** appears in your project directory. This file contains information about the generated files.

7. Click the **Finish** button, the parameter editor generates the top-level HDL code for your IP core, and a simulation directory which includes files for simulation.

The **Finish** button may be unavailable until all parameterization errors listed in the messages window are corrected.

8. Click **Yes** if you are prompted to add the Quartus II IP File (**.qip**) to the current Quartus II project. You can also turn on **Automatically add Quartus II IP Files to all projects**.

You can now integrate your custom IP core instance in your design, simulate, and compile. While integrating your IP core instance into your design, you must make appropriate pin assignments. You can create a virtual pin to avoid making specific pin assignments for top-level signals while you are simulating and not ready to map the design to hardware.

For some IP cores, the generation process also creates complete example designs. An example design for hardware testing is located in the **<***variation_name***>_example_design/example_project/** directory. An example design for RTL simulation is located in the **<***variation_name***>_example_design/simulation/** directory.

For information about the Quartus II software, including virtual pins and the MegaWizard Plug-In Manager, refer to Quartus II Help.

Simulate the IP Core

You can simulate your IP core variation with the functional simulation model and the testbench or example design generated with your IP core. The functional simulation model and testbench files are generated in a project subdirectory. This directory may also include scripts to compile and run the testbench.

For a complete list of models or libraries required to simulate your IP core, refer to the scripts provided with the testbench.

For more information about simulating Altera IP cores, refer to *Simulating Altera Designs* in volume 3 of the *Quartus II Handbook*.

section of the respective MegaCore Function chapter in this user guide

section of the respective MegaCore Function chapter in this user guide**Generated Files**

Table 2–1 describes the generated files and other files that may be in your project directory.

The names and types of files vary depending on the variation name and HDL type you specify during parameterization For example, a different set of files are created based on whether you create your design in Verilog HDL or VHDL.

For a description of the signals that the MegaCore function variation supports, refer to the "Signals" section of the respective MegaCore Function chapter in this user guide.

Table 2–1. Generated Files *(1)*

File Name	Description
<*variation name*>**.bsf**	Quartus II block symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<*variation name*>**.cmp**	A VHDL component declaration file for the MegaCore function variation. Add the contents of this file to any VHDL architecture that instantiates the MegaCore function.
<*variation name*>**.qip**	A single Quartus IP file is generated that contains all of the assignments and other information required to process your MegaCore function variation in the Quartus II compiler. In the SOPC Builder flow, this file is automatically included in your project. In the MegaWizard™ Plug-In Manager flow, you are prompted to add the **.qip** file to the current Quartus II project when you exit from the wizard. In SOPC Builder, a **.qip** file is generated for each MegaCore function and SOPC Builder component. Each of these **.qip** files are referenced by the system level **.qip** file and together include all the information required to process the system.
<*variation name*>**.vhd**, or **.v**	A VHDL or Verilog HDL file that defines the top-level description of the custom MegaCore function variation. Instantiate the entity defined by this file inside your design. Include this file when compiling your design in the Quartus II software.
<*variation name*>**.vho** or **.vo**	VHDL or Verilog HDL output files that defines an IP functional simulation model.
<*variation name*>**_bb.v**	A Verilog HDL black-box file for the MegaCore function variation. Use this file when using a third-party EDA tool to synthesize your design.
<*variation name*>**_syn.v**	A timing and resource estimation netlist for use in some third-party synthesis tools.

Note to Table 2–1:

(1) The <*variation name*> prefix is added automatically using the base output file name you specified in the parameter editor.

3. Interfaces

Interface Types

The MegaCore functions in the Video and Image Processing Suite use standard interfaces for data input and output, control input, and access to external memory. These standard interfaces ensure that video systems can be quickly and easily assembled by connecting MegaCore functions together.

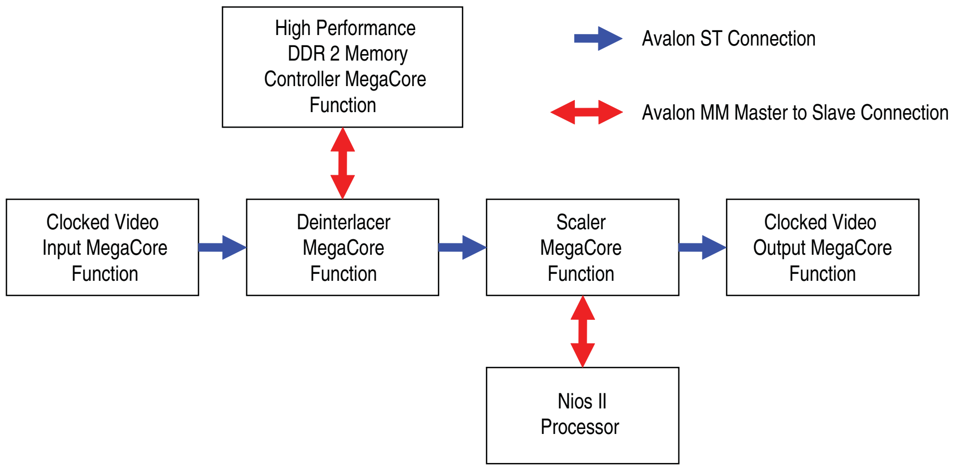
The functions use the following types of interfaces:

- Avalon-ST interface—a streaming interface that supports backpressure. The Avalon-ST Video protocol transmits video and configuration data. This interface type allows the simple creation of video processing data paths, where MegaCore functions can be connected together to perform a series of video processing functions.
- Avalon-MM slave interface—provides a means to monitor and control the properties of the MegaCore functions.
- Avalon-MM master interface—when the MegaCore functions require access to a slave interface, for example an external memory controller.

For more information about these interface types, refer to the *Avalon Interface Specifications*.

Figure 3–1 shows an example of video processing data paths using the Avalon-ST and Avalon-MM interfaces. This abstracted view is similar to that provided in the Qsys tool, where interface wires are grouped together as single connections.

Figure 3–1. Abstracted Block Diagram Showing Avalon-ST and Avalon-MM Connections


High Performance DDR 2 Memory Controller MegaCore Function
Avalon ST Connection
Avalon MM Master to Slave Connection
Clocked Video Input MegaCore Function
Deinterlacer MegaCore Function
Scaler MegaCore Function
Clocked Video Output MegaCore Function
Nios II Processor

The Clocked Video Input and Clocked Video Output MegaCore functions in Figure 3–1 also have external interfaces that support clocked video standards. These MegaCore functions can connect between the function's Avalon-ST interfaces and functions using clocked video standards such as BT.656.

☞ For information about the supported clocked video interfaces, refer to the "Functional Description" sections of "Clocked Video Input MegaCore Function" on page 10–1 and "Clocked Video Output MegaCore Function" on page 11–1.

Avalon-ST Video Protocol

The MegaCore functions in the Video and Image Processing Suite use the Avalon-ST Video protocol. The Avalon-ST Video protocol is a packet-oriented way to send video and control data over Avalon-ST connections. Using the Avalon-ST Video protocol allows the construction of image processing data paths which automatically configure to changes in the format of the video being processed. This minimizes the external control logic required to configure a video system.

Packets

The packets of the Avalon-ST Video protocol are split into symbols, where each symbol represents a single piece of data (see the *Avalon Interface Specifications*). For all packet types on a particular Avalon-ST interface the number of symbols sent in parallel (that is, on one clock cycle) and the bit width of all symbols is fixed. The symbol bit width and number of symbols sent in parallel defines the structure of the packets.

The functions predefine the following three types of packet:

- Video data packets containing only uncompressed video data
- Control data packets containing the control data configure the cores for incoming video data packets
- Ancillary (non-video) data packets containing ancillary packets from the vertical blanking period of a video frame.

There are also seven packet types reserved for users, and five packet types reserved for future definition by Altera.

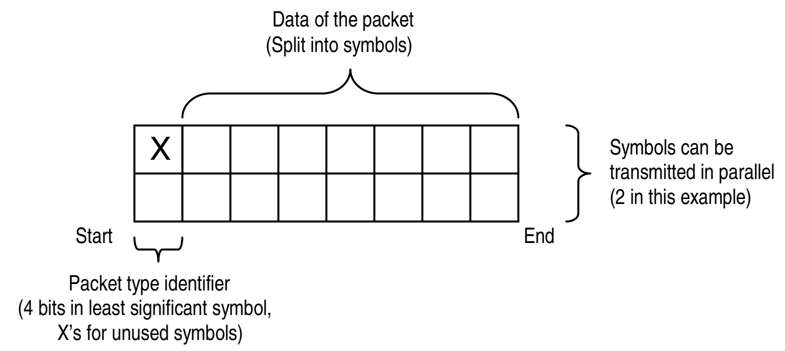
The packet type is defined by a 4-bit packet type identifier. This type identifier is the first value of any packet. It is the symbol in the least significant bits of the interface. Functions do not use any symbols in parallel with the type identifier (assigned X as shown in Figure 3–2).

Table 3–1 lists the packet types and Figure 3–2 shows the structure of a packet.

Table 3–1. Avalon-ST Video Packet Types

Type Identifier	Description
0	Video data packet
1–8	User packet types
9–12	Reserved for future Altera use
13	Ancillary data packet
14	Reserved for future Altera use
15	Control data packet

Figure 3–2. Packet Structure


Data of the packet
(Split into symbols)
X
Symbols can be transmitted in parallel (2 in this example)
Start
End
Packet type identifier
(4 bits in least significant symbol,
X's for unused symbols)

The Avalon-ST Video protocol is designed to be most efficient for transferring video data, therefore the symbol bit width and the number of symbols transferred in parallel (that is, in one clock cycle) are defined by the parameters of the video data packet types (refer to "Static Parameters of Video Data Packets" on page 3–4).

Video Data Packets

Video data packets transmit video data between the MegaCore functions. A video data packet contains the color plane values of the pixels for an entire progressive frame or an entire interlaced field.

The video data is sent per pixel in a raster scan order. The pixel order is as follows:

1. From the top left of the image right wards along the horizontal line.
2. At the end of the current line, jump to the left most pixel of the next horizontal line down.
3. Go right wards along the horizontal line.
4. Repeat 2 and 3 until the bottom right pixel is reached and the frame has been sent.

Static Parameters of Video Data Packets

The following two static parameters specify the Avalon-ST interface that video systems use:

Bits Per Pixel Per Color Plane

The maximum number of bits that represent each color plane value within each pixel. For example R′G′B′ data of eight bits per sample (24 bits per pixel) would use eight bits per pixel per color plane.

☞ This parameter also defines the bit width of symbols for all packet types on a particular Avalon-ST interface. An Avalon-ST interface must be at least four bits wide to fully support the Avalon-ST Video protocol.

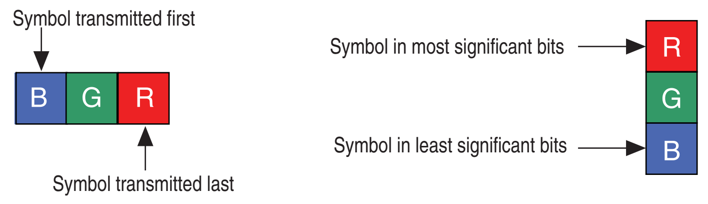
Color Pattern

The organization of the color plane samples within a video data packet is referred to as the color pattern. This color pattern cannot change within a video data packet.

A color pattern is represented as a matrix which defines a repeating pattern of color plane samples that make up a pixel (or multiple pixels). The height of the matrix indicates the number of color plane samples transmitted in parallel, the width determines how many cycles of data are transmitted before the pattern repeats.

Each color plane sample in the color pattern maps to an Avalon-ST symbol. The mapping is such that color plane samples on the left of the color pattern matrix are the symbols transmitted first. Color plane samples on the top are assigned to the symbols occupying the most significant bits of the Avalon-ST data signal as shown in Figure 3–3.

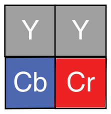
Figure 3–3. Symbol Transmission Order


Symbol transmitted first
B
G
R
Symbol transmitted last
Symbol in most significant bits
R
G
B
Symbol in least significant bits

☞ The number of color plane samples transmitted in parallel (that is, in one clock cycle) defines the number of symbols transmitted in parallel for all packet types on a particular Avalon-ST interface.

A color pattern can represent more than one pixel. This is the case when consecutive pixels contain samples from different color planes—There must always be at least one common color plane between all pixels in the same color pattern. Color patterns representing more than one pixel are identifiable by a repeated color plane name. The number of times a color plane name is repeated is the number of pixels represented. Figure 3–4 shows two pixels of horizontally subsampled Y' CbCr (4:2:2) where Cb and Cr alternate between consecutive pixels.

Figure 3–4. Horizontally Subsampled Y'CbCr



In the common case, each element of the matrix contains the name of a color plane from which a sample must be taken. The exception is for vertically sub sampled color planes. These are indicated by writing the names of two color planes in a single element, one above the other. Samples from the upper color plane are transmitted on even rows and samples from the lower plane transmitted on odd rows as shown in Figure 3–5.

Figure 3–5. Vertically Subsampled Y'CbCr

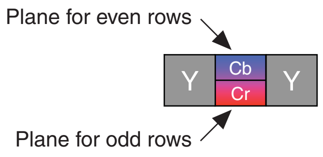
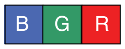
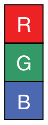
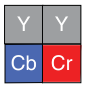


Table 3–2 lists the static parameters and gives some examples of how you can use them.

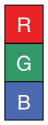
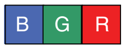
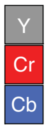
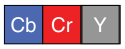
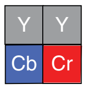
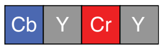
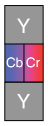
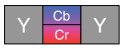
Table 3–2. Examples of Static Avalon-ST Video Data Packet Parameters

Parameters		Description
Bits per Color Sample	Color Pattern	
8		Three color planes, B', G', and R' are transmitted in alternating sequence and each B', G', or R' sample is represented using 8 bits of data.
10		Three color planes are transmitted in parallel, leading to higher throughput than when transmitted in sequence, usually at higher cost. Each R', G', or B' sample is represented using 10 bits of data, so that, in total, 30 bits of data are transmitted in parallel.
10		4:2:2 video in the Y'CbCr color space, where there are twice as many Y' samples as Cb or Cr samples. One Y' sample and one of either a Cb or a Cr sample is transmitted in parallel. Each sample is represented using 10 bits of data.

The Avalon-ST Video protocol does not force the use of specific color patterns, however a few MegaCore functions of the Video and Image Processing Suite only process video data packets correctly if they use a certain set of color patterns. Chapter 4, Functional Descriptions describes the set of color patterns that the MegaCore functions use.

Table 3–3 lists the recommended color patterns for common combinations of color spaces and color planes in parallel and sequence.

Table 3–3. Recommended Color Patterns

Color Space	Recommended Color Patterns	
	Parallel	Sequence
R'G'B		
Y'CbCr		
4:2:2 Y'CbCr		
4:2:0 Y'CbCr		

Following these recommendations, ensures compatibility minimizing the need for color pattern rearranging. These color patterns are designed to be compatible with common clocked video standards where possible.

If you must rearrange color patterns, you can use the Color Plane Sequencer MegaCore function.

Specifying Color Pattern Options

You can specify parameters in the parameter editor that allow you to describe a color pattern that has its color planes entirely in sequence (one per cycle) or entirely in parallel (all in one cycle). You can select the number of color planes per pixel, and whether the planes of the color pattern transmit in sequence or in parallel.

Some of the MegaCore functions' user interfaces provide controls allowing you to describe a color pattern that has color plane samples in parallel with each other and in sequence such that it extends over multiple clock cycles. You can select the number of color planes of the color pattern in parallel (number of rows of the color pattern) and the number of color planes in sequence (number of columns of the color pattern).

Structure of Video Data Packets

Figure 3–6 shows the structure of a video data packet using a set parallel color pattern and bits per pixel per color plane.

Figure 3–6. Parallel Color Pattern

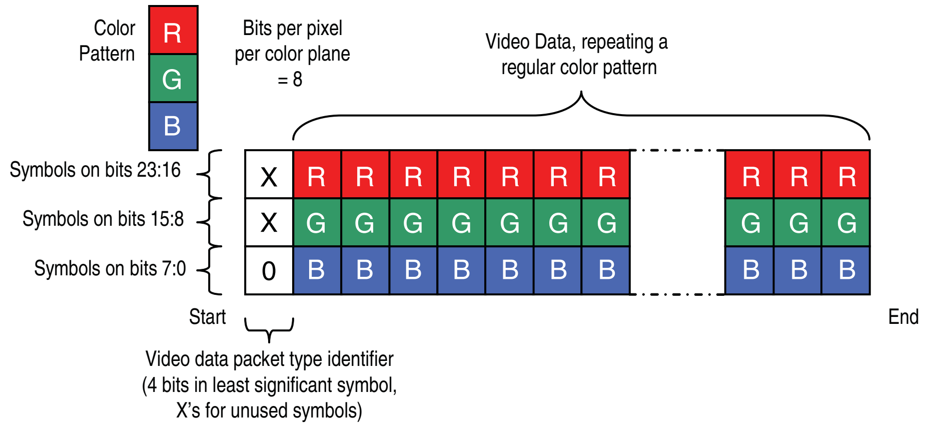
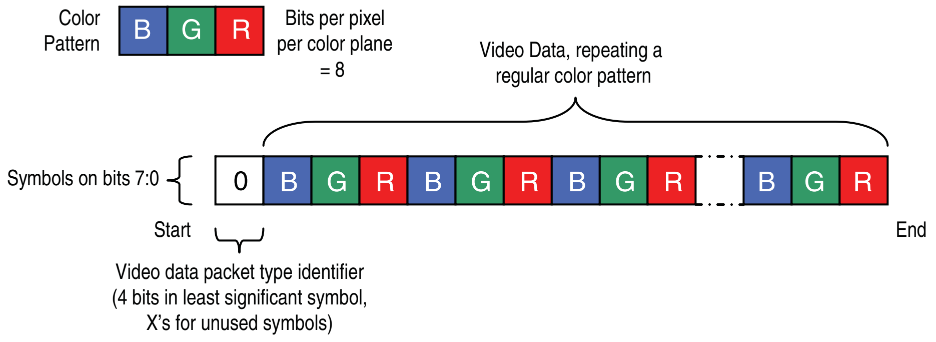

Color Pattern
R
G
B
Bits per pixel per color plane = 8
Video Data, repeating a regular color pattern
Symbols on bits 23:16
Symbols on bits 15:8
Symbols on bits 7:0
X R R R R R R R R R R
X G G G G G G G G G G
0 B B B B B B B B B B
Start
End
Video data packet type identifier (4 bits in least significant symbol, X's for unused symbols)

Figure 3–7 shows the structure of a video data packet using a set sequential color pattern and bits per pixel per color plane.

Figure 3–7. Sequence Color Pattern


Color Pattern
B G R
Bits per pixel per color plane = 8
Video Data, repeating a regular color pattern
Symbols on bits 7:0
0 B G R B G R B G R B G R
Start
End
Video data packet type identifier (4 bits in least significant symbol, X's for unused symbols)

Control Data Packets

Control data packets configure the MegaCore functions so that they correctly process the video data packets that follow.

In addition to a packet type identifier of value 15, control data packets contain the following data:

Width (16 bit), Height (16 bit), Interlacing (4 bit)

The width and height values are the dimensions of the video data packets that follow. The width refers to the width in pixels of the lines of a frame. The height refers the number of lines in a frame or field, such that a field of interlaced 1920×1080 (1080i) would have a width of 1920 and a height of 540, and a frame of 1920×1080 (1080p) would have a width of 1920 and a height of 1080.

When a video data packet uses a subsampled color pattern, the individual color planes of the video data packet have different dimensions. For example, 4:2:2 has one full width, full height plane and two half width, full height planes. For 4:2:0 there are one full width, full height plane and two half width, half height planes. In these cases you must configure the width and height fields of the control data packet for the fully sampled, full width, and full height plane.

The interlacing value in the control packet indicates whether the video data packets that follow contain progressing or interlaced video. The most significant two bits of the interlacing nibble describe whether the next video data packet is either progressive, interlaced field 0 (F0) containing lines 0, 2, 4.... or interlaced field 1 (F1) containing lines 1, 3, 5... 00 means progressive, 10 means interlaced F0, and 11 means interlaced F1.

The meaning of the second two bits is dependent on the first two bits. If the first two bits are set to 10 (F0) or 11 (F1), the second two bits describe the synchronization of interlaced data. Use the synchronization bits for progressive segmented frame (PsF) content, where progressive frames are transmitted as two interlaced fields.

Synchronizing on F0 means that a video frame should be constructed from an F1 followed by an F0. Similarly, synchronizing on F1 means that a video frame should be constructed from an F0 followed by an F1. The other synchronization options are *don't care* when there is no difference in combining an F1 then F0, or an F0 then F1. The final option is *don't know* to indicate that the synchronization of the interlaced fields is unknown. The encoding for these options are 00 for synchronize on F0, 01 for synchronize on F1, 11 for *don't care*, and 10 for *don't know*.

The synchronization bits do not affect the behavior of the Deinterlacer because the synchronization field is fixed at compile time. However, they do affect the behavior of the Frame Buffer when dropping and repeating pairs of fields.

If the first two bits indicate a progressive frame, the second two bits indicate the type of the last field that the progressive frame was deinterlaced from. The encoding for this is 10 for *unknown* or 11 for *not deinterlaced*, 00 for F0 last, and 01 for F1 last. Table 3–4 gives some examples of the control parameters.

Table 3–4. Examples of Control Data Packet Parameters

	Parameters			**Description**
Type	Width	Height	Interlacing	
15	1920	1080	0011	The frames that follow are progressive with a resolution of 1920×1080.
15	640	480	0011	The frames that follow are progressive with a resolution of 640×480.
15	640	480	0000	The frames that follow are progressive with a resolution of 640×480. The frames were deinterlaced using F0 as the last field.
15	640	480	0001	The frames that follow are progressive with a resolution of 640×480. The frames were deinterlaced using F1 as the last field.
15	640	240	1000	The fields that follow are 640 pixels wide and 240 pixels high. The next field is F0 (even lines) and it is paired with the F1 field that precedes it.
15	1920	540	1100	The fields that follow are 1920 pixels wide and 540 pixels high. The next field is F1 (odd lines) and it is paired with the F0 field that follows it.
15	1920	540	1101	The fields that follow are 1920 pixels wide and 540 pixels high. The next field is F1 (odd lines) and it is paired with the F0 field that precedes it.

Table 3–4. Examples of Control Data Packet Parameters

	Parameters			Description
Type	Width	Height	Interlacing	
15	1920	540	1011	The fields that follow are 1920 pixels wide and 540 pixels high. The next field is F0 (even lines) and you must handle the stream as genuine interlaced video material where the fields are all temporally disjoint.
15	1920	540	1010	The fields that follow are 1920 pixels wide and 540 pixels high. The next field is F0 (even lines) and you must handle the stream as genuine interlaced video content although it may originate from a progressive source converted with a pull-down.

Use of Control Data Packets

A control data packet must immediately precede every video data packet. To facilitate this, any IP function that generates control data packets must do so once before each video data packet. Additionally all other MegaCore functions in the processing pipeline must either pass on a control data packet or generate a new one before each video data packet. If the function receives more than one control data packet before a video data packet, it uses the parameters from the last received control data packet. If the function receives a video data packet with no preceding control data packet, the current functions keep the settings from the last control data packet received, with the exception of the next interlaced field type—toggling between F0 and F1 for each new video data packet that it receives.

This behavior may not be supported in future releases. Altera recommends for forward compatibility that functions implementing the protocol ensure there is a control data packet immediately preceding each video data packet.

Structure of a Control Data Packet

A control data packet complies with the standard of a packet type identifier followed by a data payload. The data payload is split into nibbles of 4 bits, each data nibble is part of a symbol. If the width of a symbol is greater than 4 bits, the function does not use the most significant bits of the symbol.

Table 3–5 lists the order of the nibbles and associated symbols.

Table 3–5. Order of Nibbles and Associated Symbols

Order	Symbol	Order	Symbol
1	width[15..12]	6	height[11..8]
2	width[11..8]	7	height[7..4]
3	width[7..4]	8	height[3..0]
4	width[3..0]	9	interlacing[3..0]
5	height[15..12]	—	—

If the number of symbols transmitted in one cycle of the Avalon-ST interface is more than one, then the nibbles are distributed such that the symbols occupying the least significant bits are populated first.

Figure 3–8 to Figure 3–10 show examples of control data packets, and how they are split into symbols.

Figure 3–8. Three Symbols in Parallel

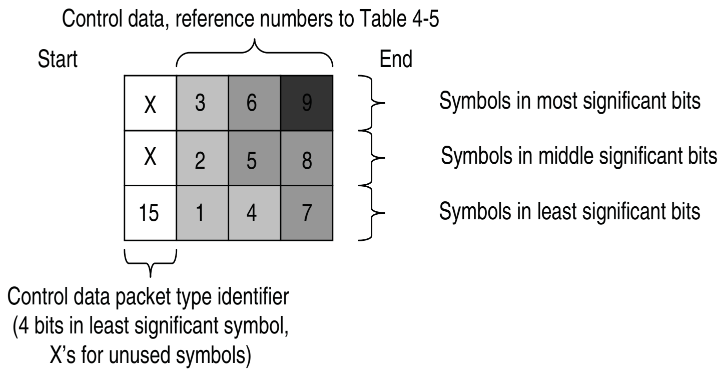

Control data, reference numbers to Table 4-5
Start
End
X 3 6 9
X 2 5 8
15 1 4 7
Symbols in most significant bits
Symbols in middle significant bits
Symbols in least significant bits
Control data packet type identifier
(4 bits in least significant symbol,
X's for unused symbols)

Figure 3–9. Two Symbols in Parallel

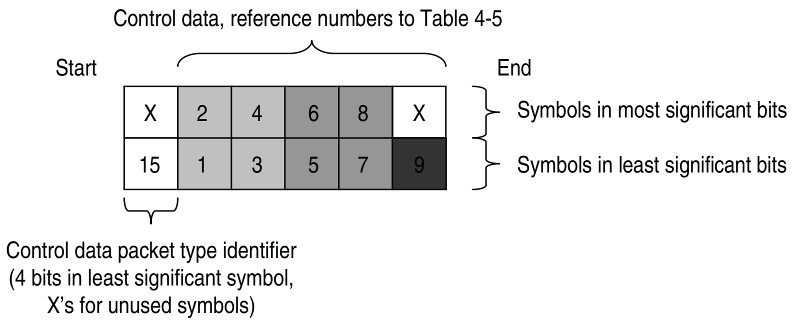
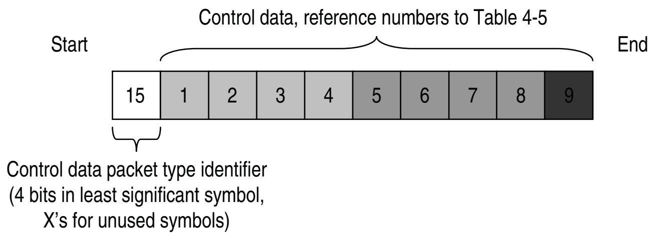

Control data, reference numbers to Table 4-5
Start
End
X 2 4 6 8 X
15 1 3 5 7 9
Symbols in most significant bits
Symbols in least significant bits
Control data packet type identifier
(4 bits in least significant symbol,
X's for unused symbols)

Figure 3–10. One Symbol in Parallel


Control data, reference numbers to Table 4-5
Start
End
15 1 2 3 4 5 6 7 8 9
Control data packet type identifier
(4 bits in least significant symbol,
X's for unused symbols)

Ancillary Data Packets

Ancillary data packets send ancillary packets between MegaCore functions. Ancillary data packets are typically placed between a control data packet and a video data packet and contain information that describes the video data packet, for example active format description codes.

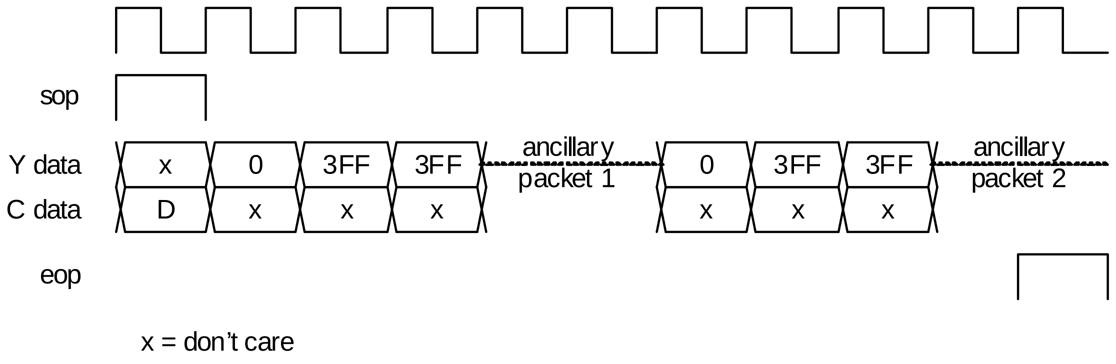
An ancillary data packet can contain one or more ancillary packets, each ancillary packet starts with the hexadecimal code 0, 3FF, 3FF.

The format of ancillary packets is defined in the SMPTE S291M standard.

MegaCore functions are not required to understand or process ancillary data packets, but must forward them on, as is done with user-defined and Altera-reserved packets.

Figure 3–11 shows an example of an Avalon-ST Video Ancillary Data Packet containing two ancillary packets.

Figure 3–11. Avalon-ST Video Ancillary Data Packet



User-Defined and Altera-Reserved Packets

The Avalon-ST Video protocol specifies that there are seven packet types reserved for use by users and five packet types reserved for future use by Altera. The data content of all of these packets is undefined. However the structure must follow the rule that the packets are split into symbols as defined by the number color plane samples sent in one cycle of the color pattern.

Unlike control data packets, user packets are not restricted to four bits of data per symbol. However when a core reduces the bits per pixel per color plane (and thus the bit width of the symbols) to less than the number of bits in use per symbol, data is lost.

Packet Propagation

The Avalon-ST Video protocol is optimized for the transfer of video data while still providing a flexible way to transfer control data and other information. To make the protocol flexible and extensible, the Video and Image Processing MegaCore functions obey the following rules about propagating non-video packets:

- MegaCore functions must propagate user packets until their end of packet signal is received. Nevertheless, MegaCore functions that buffer packets into external memory might introduce a maximum size due to limited storage space.
- MegaCore functions can propagate control packets or modify them on the fly. MegaCore functions can also cancel a control packet by following it with a new control packet.
- When the bits per color sample change from the input to the output side of a MegaCore function, the non-video packets are truncated or padded. Otherwise, the full bit width is transferred.

- MegaCore functions that can change the color pattern of a video data packets may also pad non-video data packets with extra data. When defining a packet type where the length is variable and meaningful, it is recommended to send the length at the start of the packet.

Transmission of Avalon-ST Video Over Avalon-ST Interfaces

Avalon-ST Video is a protocol transmitted over Avalon-ST interfaces. The *Avalon Interface Specifications* define parameters that you can use to specify the types of Avalon-ST interface.

Table 3–6 lists the values of these parameters that are defined for transmission of the Avalon-ST Video protocol. All parameters not explicitly listed in the table have undefined values.

Table 3–6. Avalon-ST Interface Parameters

Parameter Name	Value
BITS_PER_SYMBOL	Variable. Always equal to the Bits per Color Sample parameter value of the stream of pixel data being transferred.
SYMBOLS_PER_BEAT	Variable. Always equal to the number of color samples being transferred in parallel. This is equivalent to the number of rows in the color pattern parameter value of the stream of pixel data being transferred.
READY_LATENCY	1

The *Avalon Interface Specifications* defines signal types of which many are optional. Table 3–7 lists the signals for transmitting Avalon-ST Video. Table 3–7 does not list unused signals.

Table 3–7. Avalon-ST Interface Signal Types

Signal	Width	Direction
`ready`	1	Sink to Source
`valid`	1	Source to Sink
`data`	`bits_per_symbol` × `symbols_per_beat`	Source to Sink
`startofpacket`	1	Source to Sink
`endofpacket`	1	Source to Sink

Packet Transfer Examples

All packets are transferred using the Avalon-ST signals in the same way. Three examples are given here, two showing video data packets, and one showing a control data packet. Each is an example of generic packet transmission.

Example 1 (Data Transferred in Parallel)

This example shows the transfer of a video data packet in to and then out of a generic MegaCore function that supports the Avalon-ST Video protocol.

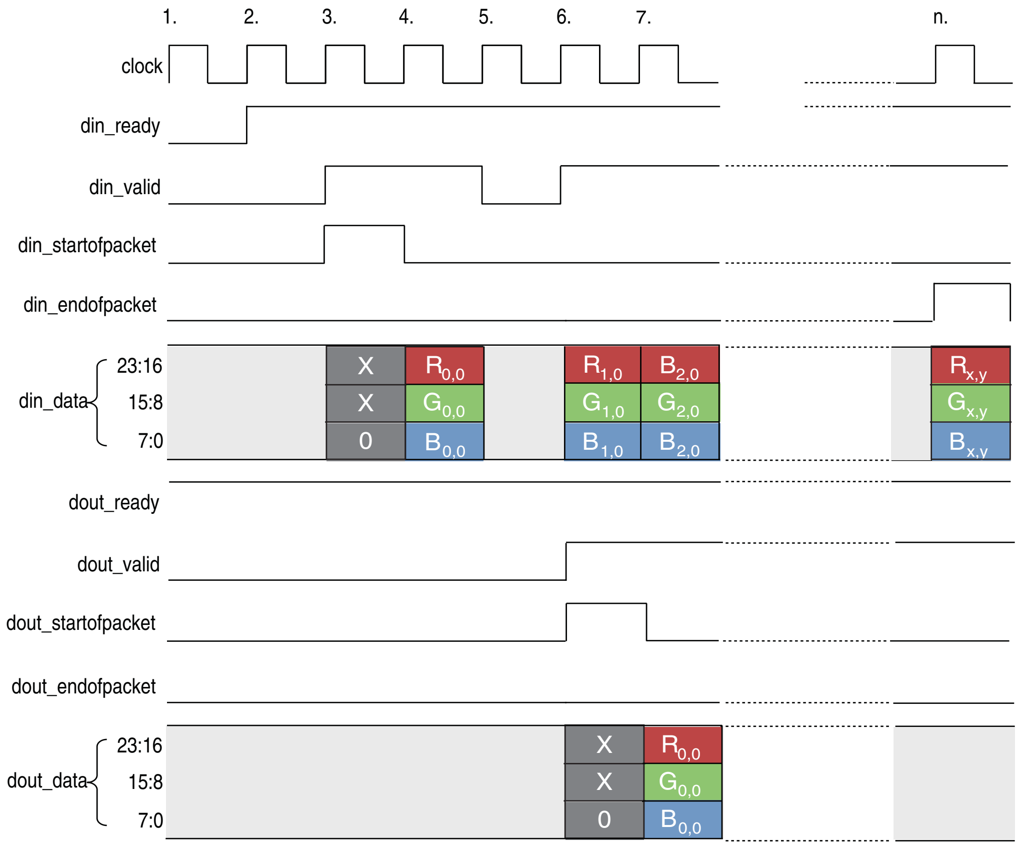
In this case, both the input and output video data packets have a parallel color pattern and eight bits per pixel per color plane as listed in Table 3–8.

Table 3–8. Parameters for Example of Data Transferred in Parallel

Parameter	Value
Bits per Pixel per Color Plane	8
Color Pattern	R G B

Figure 3–12 shows how the first few pixels of a frame are processed.

Figure 3–12. Timing Diagram Showing R'G'B' Transferred in Parallel



This example has one Avalon-ST port named `din` and one Avalon-ST port named `dout`. Data flows into the MegaCore function through `din`, is processed and flows out of the MegaCore function through `dout`.

There are five signals types (ready, valid, data, startofpacket, and endofpacket) associated with each port. The `din_ready` signal is an **output** from the MegaCore function and indicates when the input port is ready to receive data. The `din_valid` and `din_data` signals are both inputs. The source connected to the input port sets `din_valid` to logic '1' when `din_data` has useful information that must be sampled. `din_startofpacket` is an input signal that is raised to indicate the start of a packet, with `din_endofpacket` signaling the end of a packet.

The five output port signals have equivalent but opposite semantics.

The sequence of events shown in Figure 3–12 is:

1. Initially, `din_ready` is logic '0', indicating that the MegaCore function is not ready to receive data on the next cycle. Many of the Video and Image Processing Suite MegaCore functions are not ready for a few clock cycles in between rows of image data or in between video frames.

   For further details of each MegaCore function, refer to the "Functional Description" section of the respective MegaCore Function chapter in this user guide.

2. The MegaCore function sets `din_ready` to logic '1', indicating that the input port is ready to receive data one clock cycle later. The number of clock cycles of delay which must be applied to a ready signal is referred to as ready latency in the *Avalon Interface Specifications*. All of the Avalon-ST interfaces that the Video and Image Processing Suite uses have a ready latency of one clock cycle.
3. The source feeding the input port sets `din_valid` to logic '1' indicating that it is sending data on the data port and sets `din_startofpacket` to logic '1' indicating that the data is the first value of a new packet. The data is 0, indicating that the packet is video data.
4. The source feeding the input port holds `din_valid` at logic '1' and drops `din_startofpacket` indicating that it is now sending the body of the packet. It puts all three color values of the top left pixel of the frame on to `din_data`.
5. No data is transmitted for a cycle even though `din_ready` was logic '1' during the previous clock cycle and therefore the input port is still asserting that it is ready for data. This could be because the source has no data to transfer. For example, if the source is a FIFO, it could have become empty.
6. Data transmission resumes on the input port: `din_valid` transitions to logic '1' and the second pixel is transferred on `din_data`. Simultaneously, the MegaCore function begins transferring data on the output port. The example MegaCore function has an internal latency of three clock cycles so the first output is transferred three cycles after being received. This output is the type identifier for a video packet being passed along the datapath.

   For guidelines about the latencies of each Video and Image Processing MegaCore function, refer to refer to the "Latency" section of the respective MegaCore Function chapter in this user guide.

7. The third pixel is input and the first processed pixel is output.
8. For the final sample of a frame, the source sets `din_endofpacket` to logic '1', `din_valid` to '1', and puts the bottom-right pixel of the frame on to `din_data`.

Example 2 (Data Transferred in Sequence)

This example shows how a number of pixels from the middle of a frame could be processed by another MegaCore function. This time handling a color pattern that has planes B'G'R' in sequence. This example does not show the start of packet and end of packet signals because these are always low during the middle of a packet.

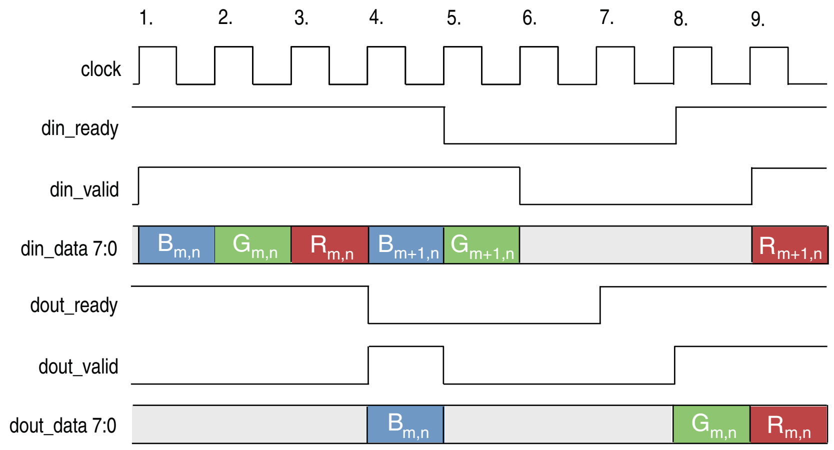
The bits per pixel per color plane and color pattern are listed in Table 3–9.

Table 3–9. Parameters for Example of Data Transferred in Sequence

Parameter	Value
Bits per Color Sample	8
Color Pattern	B G R

Figure 3–13 shows how a number of pixels from the middle of a frame are processed.

Figure 3–13. Timing Diagram Showing R'G'B' Transferred in Sequence



Note to Figure 3–13:

(1) The `startofpacket` and `endofpacket` signals are not shown but are always low during the sequence shown in this figure.

This example is similar to Figure 3–12 on page 3–13 except that it is configured to accept data in sequence rather than parallel. The signals shown in the timing diagram are therefore the same but with the exception that the two data ports are only 8 bits wide.

The sequence of events shown in Figure 3–13 is:

1. Initially, `din_ready` is logic '1'. The source driving the input port sets `din_valid` to logic '1' and puts the blue color value $B_{m,n}$ on the `din_data` port.
2. The source holds `din_valid` at logic '1' and the green color value $G_{m,n}$ is input.
3. The corresponding red color value $R_{m,n}$ is input.
4. The MegaCore function sets `dout_valid` to logic '1' and outputs the blue color value of the first processed color sample on the `dout_data` port. Simultaneously the sink connected to the output port sets `dout_ready` to logic '0'. The *Avalon Interface Specifications* state that sinks may set ready to logic '0' at any time, for example because the sink is a FIFO and it has become full.

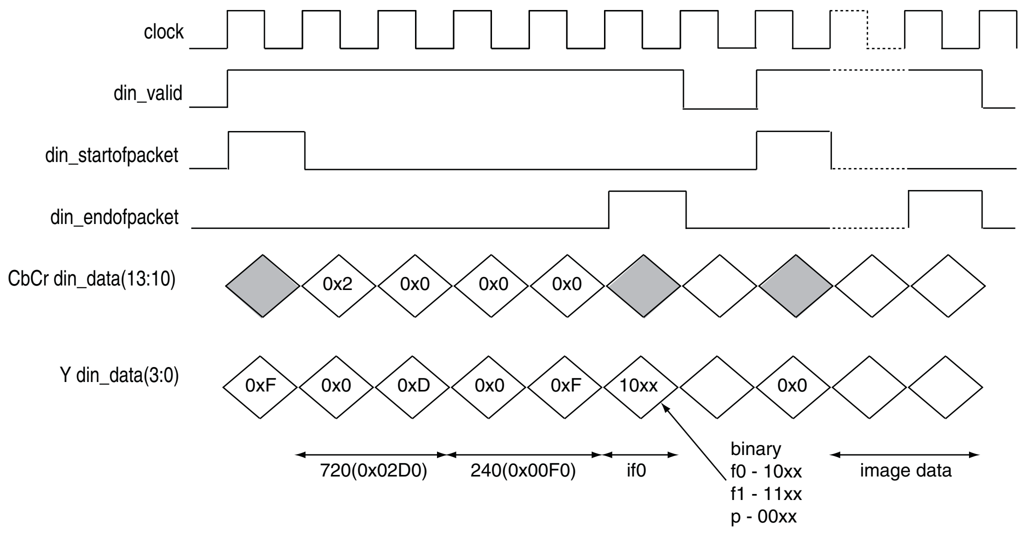
5. The MegaCore function sets `dout_valid` to logic '0' and stops putting data on the `dout_data` port because the sink is not ready for data. The MegaCore function also sets `din_ready` to logic '0' because there is no way to output data and the MegaCore function must stop the source from sending more data before it uses all internal buffer space. The sink holds `din_valid` at logic '1' and transmits one more color sample $G_{m+1,n}$, which is legal because the ready latency of the interface means that the change in the MegaCore function's readiness does not take effect for one clock cycle.
6. Both the input and output interfaces transfer no data: the MegaCore function is stalled waiting for the sink.
7. The sink sets `dout_ready` to logic '1'. This could be because space has been cleared in a FIFO.
8. The MegaCore function sets `dout_valid` to logic '1' and resumes transmitting data. Now that the flow of data is again unimpeded, it sets `din_ready` to logic '1'.
9. The source responds to `din_ready` by setting `din_valid` to logic '1' and resuming data transfer.

Example 3 (Control Data Transfer)

Figure 3–14 shows the transfer of a control packet for a field of 720×480 video (with field height 240). It is transferred over an interface configured for 10-bit data with two color planes in parallel. Each word of the control packet is transferred in the lowest four bits of a color plane, starting with bits 3:0, then 13:10.

Example 1 uses the start of packet and end of packet lines in exactly the same way.

Figure 3–14. Example of Control Packet Transfer



Avalon-MM Slave Interfaces

The Video and Image Processing Suite MegaCore functions that permit run-time control of some aspects of their behavior, use a common type of Avalon-MM slave interface for this purpose.

Each slave interface provides access to a set of control registers which must be set by external hardware. You must assume that these registers power up in an undefined state. The set of available control registers and the width in binary bits of each register varies with each control interface.

For a description of the control registers for each MegaCore function, refer to the "Control Register Map" section of the respective MegaCore Function chapter in this user guide.

The first two registers of every control interface perform the following two functions (the others vary with each control interface):

- Register 0 is the `Go` register. Bit zero of this register is the `Go` bit. A few cycles after the function comes out of reset, it writes a zero in the `Go` bit (remember that all registers in Avalon-MM control slaves power up in an undefined state).

  Although there are a few exceptions, most Video and Image Processing Suite MegaCore functions stop at the beginning of an image data packet if the `Go` bit is set to 0. This allows you to stop the MegaCore function and to program run-time control data before the processing of the image data begins. A few cycles after the `Go` bit is set by external logic connected to the control port, the MegaCore function begins processing image data. If the `Go` bit is unset while data is being processed, then the MegaCore function stops processing data again at the beginning of the next image data packet and waits until the *Go* bit is set by external logic.
- Register 1 is the `Status` register. Bit zero of this register is the `Status` bit, the function does not use all other bits. The function sets the `Status` bit to 1 when it is running, and zero otherwise. External logic attached to the control port must not attempt to write to the `Status` register.

The following pseudo-code illustrates the design of functions that double-buffer their control (that is, all MegaCore functions except the Gamma Corrector and some Scaler parameterizations):

```
go = 0;
while (true)
{
    read_non_image_data_packets();
    status = 0;
    while (go != 1)
        wait;
    read_control(); // Copies control to internal registers
    status = 1;
    send_image_data_header();
    process_frame();
}
```

For MegaCore functions that do not double buffer their control data, the algorithm described in the previous paragraph is still largely applicable but the changes to the control register will affect the current frame.

Most Video and Image Processing Suite MegaCore functions with a slave interface read and propagate non-image data packets from the input stream until the image data header (0) of an image data packet has been received. The status bit is then set to 0 and the MegaCore function waits until the `Go` bit is set to 1 if it is not already. Once the `Go` bit is set to 1, the MegaCore function buffers control data, sets its status bit back to 1 and starts processing image data.

☞ There is a small amount of buffering at the input of each Video and Image Processing Suite MegaCore function and you must expect that a few samples are read and stored past the image data header even if the function is stalled.

You can use the `Go` and `Status` registers in combination to synchronize changes in control data to the start and end of frames. For example, suppose you want to build a system with a Gamma Corrector MegaCore function where the gamma look-up table is updated between each video frame.

You can build logic (or program a Nios® II processor) to control the gamma corrector as follows:

1. Set the `Go` bit to zero. This causes the MegaCore function to stop processing at the end of the current frame.
2. Poll the `Status` bit until the MegaCore function sets it to zero. This occurs at the end of the current frame, after the MegaCore function has stopped processing data.
3. Update the gamma look-up table.
4. Set the `Go` bit to one. This causes the MegaCore function to start processing the next frame.
5. Poll the `Status` bit until the MegaCore function sets it to one. This occurs when the MegaCore function has started processing the next frame (and therefore setting the `Go` bit to zero causes it to stop processing at the end of the next frame).
6. Repeat steps 1 to 5 until all frames are processed.

This procedure ensures that the update is performed exactly once per frame and that the MegaCore function is not processing data while the update is performed. When using MegaCore functions which double-buffer control data, such as the Alpha Blending Mixer, a more simple process may be sufficient:

1. Set the `Go` bit to zero. This causes the MegaCore function to stop if it gets to the end of a frame while the update is in progress.
2. Update the control data.
3. Set the `Go` bit to one.

   The next time a new frame is started after the `Go` bit is set to one, the new control data is loaded into the MegaCore function.

The reading on non-video packets is performed by handling any packet until one arrives with type 0. This means that when the `Go` bit is checked, the non-video type has been taken out of the stream but the video is retained.

Specification of the Type of Avalon-MM Slave Interfaces

The *Avalon Interface Specifications* define many signal types, many of which are optional.

Table 3–10 lists the signals that the Avalon-MM slave interfaces use in the Video and Image Processing Suite. Table 3–10 does not list unused signals.

Table 3–10. Avalon-MM Slave Interface Signal Types

Signal	Width	Direction
`chipselect` *(1)*	1	Input
`read` *(1)*	1	input
`address`	Variable	Input
`readdata`	Variable	Output
`write`	1	Input
`writedata`	Variable	Input
`waitrequest` *(2)*	1	Output
`irq` *(3)*	1	Output

Notes to Table 3–10:

(1) The Slave interfaces of the Video and Image Processing MegaCore functions may use either `chipselect` or `read`.

(2) For slave interfaces that do not have a predefined number of wait cycles to service a read or a write request.

(3) For slave interfaces with an interrupt request line.

Clock and reset signal types are not included. The Video and Image Processing Suite does not support Avalon-MM interfaces in multiple clock domains. Instead, the Avalon-MM slave interfaces must operate synchronously to the main clock and reset signals of the MegaCore function. The Avalon-MM slave interfaces must operate synchronously to this clock.

The *Avalon Interface Specifications* define a set of transfer properties which may or may not be exhibited by any Avalon-MM interface. Together with the list of supported signals, these properties fully define an interface type.

The control interfaces of the Video and Image Processing Suite MegaCore functions that do not use a `waitrequest` signal, exhibit the following transfer properties:

- Zero wait states on write operations
- Two wait states on read operations

Avalon-MM Master Interfaces

The Video and Image Processing Suite MegaCore functions use a common type of Avalon-MM master interface for access to external memory. Connect these master interfaces to external memory resources through arbitration logic such as that provided by the system interconnect fabric.

Specification of the Type of Avalon-MM Master Interfaces

The *Avalon Interface Specifications* define many signal types, many of which are optional.

Table 3–11 lists the signals for the Avalon-MM master interfaces in the Video and Image Processing Suite. Table 3–11 does not list unused signals.

Table 3–11. Avalon-MM Master Interface Signal Types

Signal	Width	Direction	Usage
clock	1	Input	Read-Write (optional)
`readdata`	variable	Input	Read-only
`readdatavalid`	1	Input	Read-only
reset	1	Input	Read-Write (optional)
`waitrequest`	1	Input	Read-write
`address`	32	Output	Read-write
`burstcount`	variable	Output	Read-write
`read`	1	Output	Read-only
`write`	1	Output	Write-only
`writedata`	variable	Output	Write-only

The clock and reset signal types are optional. The Avalon-MM master interfaces can operate on a different clock from the MegaCore function and its other interfaces by selecting the relevant option in the parameter editor when and if it is available.

Some of the signals in Table 3–11 are read-only and not required by a master interface which only performs write transactions.

Some other signals are write-only and not required by a master interface which only performs read transactions. To simplify the Avalon-MM master interfaces and improve efficiency, read-only ports are not present in write-only masters, and write-only ports are not present in read-only masters.

Read-write ports are present in all Avalon-MM master interfaces. Refer to the description of each MegaCore function for information about whether the master interface is read-only, write-only or read-write.

The *Avalon Interface Specifications* define a set of transfer properties which may or may not be exhibited by any Avalon-MM interface. Together with the list of supported signals, these properties fully define an interface type.

The external memory access interfaces of the Video and Image Processing Suite MegaCore functions exhibit the following transfer property:

- Pipeline with variable latency

Buffering of Non-Image Data Packets in Memory

The Frame Buffer and the Deinterlacer (when buffering is enabled) route the video stream through an external memory. Non-image data packets must be buffered and delayed along with the frame or field they relate to and extra memory space has to be allocated. You must specify the maximum number of packets per field and the maximum size of each packet to cover this requirement.

The maximum size of a packet is given as a number of symbols, header included. For instance, the size of an Avalon-ST Video control packet is 10. This size does not depend on the number of channels transmitted in parallel. Packets larger than this maximum limit may be truncated as extra data is discarded.

The maximum number of packets is the number of packets that can be stored with each field or frame. Older packets are discarded first in case of overflow. When frame dropping is enabled, the packets associated with a field that has been dropped are automatically transferred to the next field and count towards this limit.

The Frame Buffer and the Deinterlacer handle Avalon-ST Video control packets differently. The Frame Buffer processes and discards incoming control packets whereas the Deinterlacer processes and buffers incoming control packets in memory before propagating them. Because both MegaCore functions generate a new updated control packet before outputting an image data packet, this difference must be of little consequence as the last control packet always takes precedence

☞ Altera recommends that you keep the default values for **Number of packets buffered per frame** and **Maximum packet length**, unless you intend to extend the Avalon-ST Video protocol with custom packets.

4. 2D FIR Filter MegaCore Function

Core Overview

The 2D FIR Filter MegaCore function performs 2D convolution using matrices of 3×3, 5×5, or 7×7 coefficients. The 2D FIR Filter retains full precision throughout the calculation while making efficient use of FPGA resources. With suitable coefficients, the 2D FIR Filter can perform operations such as sharpening, smoothing, and edge detection. You can configure the 2D FIR Filter to change coefficient values at run time with an Avalon-MM slave interface.

Functional Description

The 2D FIR Filter MegaCore function calculates an output pixel from the multiplication of input pixels in a filter size grid (kernel) by their corresponding coefficient in the filter.

These values are summed together. Prior to output, this result is scaled, has its fractional bits removed, is converted to the desired output data type, and is constrained to a specified range. The position of the output pixel corresponds to the mid-point of the kernel. If the kernel runs over the edge of an image, the function uses zeros for the out of range pixels.

The 2D FIR Filter allows its input, output and coefficient data types to be fully defined. Constraints are 4 to 20 bits per pixel per color plane for input and output, and up to 35 bits for coefficients.

The 2D FIR Filter supports symmetric coefficients. This reduces the number of multipliers, resulting in smaller hardware. Coefficients can be set at compile time, or changed at run time using an Avalon-MM slave interface.

Calculation Precision

The 2D FIR Filter does not lose calculation precision during the FIR calculation. The calculation and result data types are derived from the range of input values (as specified by the input data type, or input guard bands if provided), the coefficient fixed point type and the coefficient values. If scaling is selected, then the result data type is scaled up appropriately such that precision is not lost.

Coefficient Precision

The 2D FIR Filter requires a fixed point type to be defined for the coefficients. The user-entered coefficients (shown as white boxes in the parameter editor) are rounded to fit in the chosen coefficient fixed point type (shown as purple boxes in the parameter editor).

Result to Output Data Type Conversion

After the calculation, the fixed point type of the results must be converted to the integer data type of the output.

The conversion is performed in four stages, in the following order:

1. **Result Scaling**. You can choose to scale up the results, increasing their range. This is useful to quickly increase the color depth of the output. The available options are a shift of the binary point right –16 to +16 places. This is implemented as a simple shift operation so it does not require multipliers.
2. **Removal of Fractional Bits**. If any fractional bits exist, you can choose to remove them.

   There are three methods:

   - Truncate to integer. Fractional bits are removed from the data. This is equivalent to rounding towards negative infinity.
   - Round - Half up. Round up to the nearest integer. If the fractional bits equal 0.5, rounding is towards positive infinity.
   - Round - Half even. Round to the nearest integer. If the fractional bits equal 0.5, rounding is towards the nearest even integer.
3. **Conversion from Signed to Unsigned**. If any negative numbers can exist in the results and the output type is unsigned, you can choose how they are converted. There are two methods:
   - Saturate to the minimum output value (constraining to range).
   - Replace negative numbers with their absolute positive value.
4. **Constrain to Range**. If any of the results are beyond the range specified by the output data type (output guard bands, or if unspecified the minimum and maximum values allowed by the output bits per pixel), logic to saturate the results to the minimum and maximum output values is automatically added.

Avalon-ST Video Protocol Parameters

The 2D FIR Filter MegaCore function can process streams of pixel data of the types listed in Table 4–1.

Table 4–1. 2D FIR Filter Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	As selected in the parameter editor.
Frame Height	As selected in the parameter editor.
Interlaced / Progressive	Progressive.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	One, two or three channels in sequence. For example, if three channels in sequence is selected, where α, β, and γ can be any color plane: α β γ

Stall Behavior and Error Recovery

The 2D FIR Filter has a delay of a little more than N–1 lines between data input and output in the case of a $N \times N$ 2D FIR Filter. This is due to line buffering internal to the MegaCore function.

Error Recovery

The 2D FIR Filter MegaCore function resolution is not configurable at run time. This MegaCore function does not read the control packets passed through it.

An error condition occurs if an `endofpacket` signal is received too early or too late for the compile time configured frame size. In either case, the 2D FIR Filter always creates output video packets of the configured size. If an input video packet has a late `endofpacket` signal, then the extra data is discarded. If an input video packet has an early `endofpacket` signal, then the video frame is padded with an undefined combination of the last input pixels.

☞ For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 4–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the of 2D FIR Filter MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 4–2. 2D FIR Filter Latency

Mode	Latency
Filter size: $N \times N$	(N–1) lines +O (cycles)

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 4–3 and Table 4–4 list the 2D FIR Filter MegaCore function parameters.

Table 4–3. 2D FIR Filter Parameter Settings Tab, General Page (Part 1 of 2)

Parameter	Value	Description
Maximum image width	32–2600, Default = **640**	Choose the maximum image width in pixels.
Number of color planes in sequence	1–**3**	The number of color planes that are sent in sequence over one data connection. For example, a value of 3 for R'G'B' R'G'B' R'G'B'.

Table 4–3. 2D FIR Filter Parameter Settings Tab, General Page (Part 2 of 2)

Parameter	Value	Description
Input Data Type: Bits per pixel per color plane (3)	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Input Data Type: Data type:	**Unsigned**, Signed	Choose whether input is unsigned or signed 2's complement.
Input Data Type: Guard bands	On or **Off**	Turn on to enable a defined input range.
Input Data Type: Max	1,048,575 to -524,288, Default = **255**	Set input range maximum value. (1)
Input Data Type: Min	1,048,575 to -524,288, Default = **0**	Set input range minimum value. (1)
Output Data Type: Bits per pixel per color plane (3)	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Output Data Type: Data type	**Unsigned**, Signed	Choose whether output is unsigned or signed 2's complement.
Output Data Type: Guard bands	On or **Off**	Turn on to enable a defined output range.
Output Data Type: Max	1,048,575 to -524,288, Default = **255**	Set output range maximum value. (2)
Output Data Type: Min	1048575 to -524288, Default = **0**	Set output range minimum value. (2)
Move binary point right (3)	–16 to +16, Default = **0**	Specify the number of places to move the binary point. This can be useful if you require a wider range output on an existing coefficient set.
Remove fraction bits by	**Round values - Half up**, Round values - Half even, Truncate values to integer	Choose the method for discarding fractional bits resulting from the FIR calculation.
Convert from signed to unsigned by	**Saturating to minimum value at stage 4**, Replacing negative with absolute value	Choose the method for signed to unsigned conversion of the FIR results.

Notes to Table 4–3

(1) The maximum and minimum guard bands values specify a range in which the input must always fall. The 2D FIR filter behaves unexpectedly for values outside this range.

(2) The output is constrained to fall in the specified range of maximum and minimum guard band values.

(3) You can specify a higher precision output by increasing **Bits per pixel per color plane** and **Move binary point right**.

Table 4–4. 2D FIR Filter Parameter Settings Tab, Coefficients Page (Part 1 of 2)

Parameter	Value	Description
Filter size (1)	**3x3**, 5x5, 7x7	Choose the size in pixels of the convolution kernel used in the filtering.
Run-time controlled	**On** or Off	Turn on to enable run-time control of the coefficient values.
Coefficient set (2)	**Simple Smoothing**, Simple Sharpening, Custom	You can choose a predefined set of simple smoothing or simple sharpening coefficients which are used for color model convolution at compile time. Alternatively, you can create your own custom set of coefficients by modifying the coefficients in the matrix.

Table 4–4. 2D FIR Filter Parameter Settings Tab, Coefficients Page (Part 2 of 2)

Parameter	Value	Description
Enable symmetric mode	**On** or Off	When on, the 3×3 coefficient matrix must be symmetrical, which enables optimization in the hardware reducing the number of multiplications required. In this mode a limited number of matrix cells are editable and the remaining values are automatically inferred. Symmetric mode is enabled for the predefined coefficient sets but can be disabled when setting custom coefficients. If you turn off this option while one of the predefined coefficient sets is selected, its values are used as the defaults for a new custom set.
Coefficients *(2)*	9, 25, or 49 fixed-point values	Each coefficient is represented by a white box with a purple box underneath. The value in the white box is the desired coefficient value, and is editable. The value in the purple box is the actual coefficient value as determined by the coefficient fixed point type specified. The purple boxes are not editable. You can create a custom set of coefficients by specifying one fixed-point value for each entry in the convolution kernel. The matrix size depends on the selected filter size.
Coefficient Precision: Signed *(3)*	On or **Off**	Turn on if you want the fixed-point type that stores the coefficients to have a sign bit.
Coefficient Precision: Integer bits *(3)*	0–35, Default = **0**	Specifies the number of integer bits for the fixed-point type used to store the coefficients.
Coefficient Precision: Fraction bits *(3)*	0–35, Default = **6**	Specifies the number of fractional bits for the fixed point type used to store the coefficients.

Notes to Table 4–4:

(1) The size of the coefficient grid changes to match the filter size when this option is changed.

(2) The values in the coefficient grid change when you select a different coefficient set.

(3) Editing these values change the actual coefficients and summands and the results values on the **General** page. Signed coefficients allow negative values; increasing the integer bits increases the magnitude range; and increasing the fraction bits increases the precision.

Signals

Table 4–5 lists the input and output signals for the 2D FIR Filter MegaCore function.

Table 4–5. 2D FIR Filter Signals (Part 1 of 2)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.

Table 4–5. 2D FIR Filter Signals (Part 2 of 2)

Signal	Direction	Description
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

Control Register Map

Table 4–6 lists the control register map for the 2D FIR Filter MegaCore function.

The width of each register in the 2D FIR Filter control register map is 32 bits. The coefficient registers use integer, signed 2's complement numbers. To convert from fractional values, simply move the binary point right by the number of fractional bits specified in the user interface.

The control data is read once at the start of each frame and is buffered inside the MegaCore function, so the registers can be safely updated during the processing of a frame.

Table 4–6. 2D FIR Filter Control Register Map

Address	Register Name	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the 2D FIR Filter MegaCore function to stop the next time control information is read. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Coefficient 0`	The coefficient at the top left (origin) of the filter kernel.
3	`Coefficient 1`	The coefficient at the origin across to the right by one.
4	`Coefficient 2`	The coefficient at the origin across to the right by two.
n	`Coefficient` *n*	The coefficient at position: ■ Row (where 0 is the top row of the kernel) is the integer value via the truncation of (*n*–2) / (filter kernel width) ■ Column (where 0 is the far left row of the kernel) is the remainder of (*n*–2) / (filter kernel width)

5. 2D Median Filter MegaCore Function

Core Overview

The 2D Median Filter MegaCore function applies 3×3 or 5×5 pixel median filters to video images. Median filtering removes speckle noise and salt-and-pepper noise while preserving the sharpness of edges in video images.

Functional Description

The 2D Median Filter MegaCore function provides a means to perform 2D median filtering operations using matrices of 3×3 or 5×5 kernels.

Each output pixel is the median of the input pixels found in a 3x3, 5x5, or 7×7 kernel centered on the corresponding input pixel. Where this kernel runs over the edge of the input image, zeros are filled in.

Larger kernel sizes require many more comparisons to perform the median filtering function and therefore require correspondingly large increases in the number of logic elements. Larger sizes have a stronger effect, removing more noise but also potentially removing more detail.

All input data samples must be in unsigned format. If the number of bits per pixel per color plane is *N*, this means that each sample consists of *N* bits of data which are interpreted as an unsigned binary number in the range $[0, 2^N - 1]$. All output data samples produced by the 2D Median Filter MegaCore function are also in the same unsigned format.

Avalon-ST Video Protocol Parameters

The 2D Median Filter MegaCore function can process streams of pixel data of the types listed in Table 5–1.

Table 5–1. 2D Median Filter Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	As selected in the parameter editor.
Frame Height	As selected in the parameter editor.
Interlaced / Progressive	Progressive.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	One, two or three channels in sequence. For example, if three channels in sequence is selected where α, β, and γ can be any color plane: α β γ

Stall Behavior and Error Recovery

The 2D Median Filter MegaCore function has a delay of a little more than *N*–1 lines between data input and output in the case of a *N*×*N* 2D Median Filter. This is due to line buffering internal to the MegaCore function.

Error Recovery

The 2D Median Filter MegaCore function resolution is not configurable at run time. This MegaCore function does not read the control packets passed through it.

An error condition occurs if an `endofpacket` signal is received too early or too late for the compile-time-configured frame size. In either case, the 2D FIR Filter always creates output video packets of the configured size.

If an input video packet has a late `endofpacket` signal, then the extra data is discarded. If an input video packet has an early `endofpacket` signal, the video frame is padded with an undefined combination of the last input pixels.

☞ For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 5–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the 2D Median Filter MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 5–2. 2D Median Filter Latency

Mode	Latency
Filter size: $N \times N$	$(N–1)$ lines $+O$ (cycles)

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 5–3 lists the 2D Median Filter MegaCore function parameters.

Table 5–3. 2D Median Filter Filter Parameter Settings (Part 1 of 2)

Parameter	Value	Description
Image width	32–2600, Default = **640**	Choose the required image width in pixels.
Image height	32–2600, Default = **480**	Choose the required image height in pixels.
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).

Table 5–3. 2D Median Filter Filter Parameter Settings (Part 2 of 2)

Parameter	Value	Description
Number of color planes in sequence	1–**3**	The number of color planes that are sent in sequence over one data connection. For example, a value of 3 for R'G'B' R'G'B' R'G'B'.
Filter size	**3x3**, 5x5	Choose the size of kernel in pixels to take the median from.

Signals

Table 5–4 lists the input and output signals for the 2D Median Filter MegaCore function.

Table 5–4. 2D Median Filter Signals

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

6. Alpha Blending MegaCore Function

Core Overview

The Alpha Blending Mixer MegaCore function mixes together up to 12 image layers. The Alpha Blending Mixer supports both picture-in-picture mixing and image blending. Each foreground layer can be independently activated and moved at run time using an Avalon-MM slave interface.

Functional Description

The Alpha Blending Mixer MegaCore function provides an efficient means to mix together up to 12 image layers. The Alpha Blending Mixer provides support for both picture-in-picture mixing and image blending with per pixel alpha support.

The location and size of each layer can be changed dynamically while the MegaCore function is running, and individual layers can be switched on and off. This run-time control is partly provided by an Avalon-MM slave port with registers for the location, and on or off status of each foreground layer. The dimensions of each layer are then specified by Avalon-ST Video control packets.

☞ It is expected that each foreground layer fits in the boundaries of the background layer.

Control data is read in two steps at the start of each frame and is buffered inside the MegaCore function so that the control data can be updated during the frame processing without unexpected side effects.

The first step occurs after all the non-image data packets of the background layer have been processed and transmitted, and the core has received the header of an image data packet of type 0 for the background. At this stage, the on/off status of each layer is read. A layer can be disabled (0), active and displayed (1) or consumed but not displayed (2). The maximum number of image layers mixed cannot be changed dynamically and must be set in the parameter editor for the Alpha Blending Mixer.

Non-image data packets of each active foreground layer, displayed or consumed, are processed in a sequential order, layer 1 first. Non-image data packets from the background layer are integrally transmitted whereas non-image data packets from the foreground layers are treated differently depending on their type. Control packets, of type 15, are processed by the core to extract the width and height of each layer and are discarded on the fly. Other packets, of type 1 to type 14, are propagated unchanged.

The second step corresponds to the usual behavior of other Video and Image Processing MegaCore functions that have an Avalon-MM slave interface.

After the non-image data packets from the background layer and the foreground layers have been processed and/or propagated, the MegaCore function waits for the `Go` bit to be set to 1 before reading the top left position of each layer.

Consequently, the behavior of the Alpha Blending Mixer differs slightly from the other Video and Image Processing MegaCore functions.

This behavior is illustrated by the following pseudo-code:

```
go = 0;
while (true)
{
    status = 0;
    read_non_image_data_packet_from background_layer();
    read_control_first_pass(); // Check layer status
                                    (disable/displayed/consumed)
    for_each_layer layer_id
    {
        // process non-image data packets for displayed or consumed
                                                    layers
        if (layer_id is not disabled)
        {

        handle_non_image_packet_from_foreground_layer(layer_id);
        }
    }
    while (go != 1)
        wait;
    status = 1;
    read_control_second_pass(); // Copies top-left coordinates to
                                            internal registers
    send_image_data_header();
    process_frame();
}
```

For information about using Avalon-MM Slave interfaces for run-time control, refer to "Avalon-MM Slave Interfaces" on page 3–17. For details of the control register maps, refer to Table 6–5 on page 6–7. For information about the Avalon-MM interface signals, refer to Table 6–4 on page 6–6.

Alpha Blending

When you turn on **Alpha blending**, the Avalon-ST input ports for the alpha channels expect a video stream compliant with the Avalon-ST Video protocol. Alpha frames contain a single color plane and are transmitted in video data packets. The first value in each packet, transmitted while the `startofpacket` signal is high, contains the packet type identifier 0. This condition holds true even when the width of the alpha channels data ports is less than 4 bits wide. The last alpha value for the bottom-right pixel is transmitted while the `endofpacket` signal is high.

It is not necessary to send control packets to the ports of the alpha channels. The width and height of each alpha layer are assumed to match with the dimensions of the corresponding foreground layer. The Alpha Blending Mixer MegaCore function must recover cleanly if there is a mismatch although there may be throughput issues at the system-level if erroneous pixels have to be discarded. All non-image data packets (including control packets) are ignored and discarded just before the processing of a frame starts.

The valid range of alpha coefficients is 0 to 1, where 1 represents full translucence, and 0 represents fully opaque.

For n-bit alpha values (RGBAn) coefficients range from 0 to 2^n–1. The model interprets (2^n–1) as 1, and all other values as (Alpha value)/2^n. For example, 8-bit alpha value 255 => 1, 254 => 254/256, 253 => 253/256 and so on.

The value of an output pixel O_N, where N is the maximum number of layers, is deduced from the following recursive formula:

$$O_N = (1 - a_N)p_N + a_N O_{N-1}$$

$$O_0 = p_0$$

where p_N is the input pixel for layer N and a_N is the alpha pixel for layer N. Consumed and disabled layers are skipped. The function does not use alpha values for the background layer (a_0) and you must tie the alpha0 port off to 0 when the core is instantiated in SOPC Builder or the parameter editor.

All input data samples must be in unsigned format. If the number of bits per pixel per color plane is N, then each sample consists of N bits of data which are interpreted as an unsigned binary number in the range $[0, 2^N - 1]$. All output data samples produced by the Alpha Blending Mixer MegaCore function are also in the same unsigned format.

Avalon-ST Video Protocol Parameters

The Alpha Blending Mixer MegaCore function can process streams of pixel data of the types listed in Table 6–1.

Table 6–1. Alpha Blending Mixer Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Run time controlled. (Maximum value specified in the parameter editor.)
Frame Height	Run time controlled. (Maximum value specified in the parameter editor.)
Interlaced / Progressive	Progressive. Interlaced input streams are accepted but they are treated as progressive inputs. Consequently, external logic is required to synchronize the input fields and prevent the mixing of F0 fields with F1 fields.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor (specified separately for image data and alpha blending).
Color Pattern (din and dout)	One, two or three channels in sequence or in parallel as selected in the parameter editor. For example, if three channels in sequence is selected where α, β, and γ can be any color plane: α β γ
Color Pattern (alpha_in)	A single color plane representing the alpha value for each pixel: A

Stall Behavior and Error Recovery

All modes at the Alpha Blending Mixer stall for a few cycles after each output frame and between output lines.

Between frames, the Alpha Blending Mixer processes non-image data packets from its input layers in sequential order and may exert backpressure during the process until the image data header has been received for all its input.

During the mixing of a frame, the Alpha Blending Mixer reads from the background input for each non-stalled cycle. The Alpha Blending Mixer also reads from the input ports associated with layers that currently cover the background image. Because of pipelining, the foreground pixel of layer *N* is read approximately *N* active cycles after the corresponding background pixel has been read. If the output is applying backpressure or if one input is stalling, the pipeline stalls and the backpressure propagates to all active inputs. When alpha blending is enabled, one data sample is read from each alpha port once each time that a whole pixel of data is read from the corresponding input port.

There is no internal buffering in the Alpha Blending Mixer MegaCore function, so the delay from input to output is just a few clock cycles and increases linearly with the number of inputs.

Error Recovery

The Alpha Blending Mixer MegaCore function processes video packets from the background layer until the end of packet is received. If an `endofpacket` signal is received too early for the background layer, the Alpha Blending Mixer enters error mode and continues writing data until it has reached the end of the current line. The `endofpacket` signal is then set with the last pixel sent. If an `endofpacket` signal is received early for one of the foreground layers or for one of the alpha layers, the Alpha Blending Mixer stops pulling data out of the corresponding input and pads the incomplete frame with undefined samples. If an `endofpacket` signal is received late for the background layer, one or more foreground layers, or one or more alpha layers, the Alpha Blending Mixer enters error mode.

When the Alpha Blending Mixer MegaCore function enters error mode (because of an early `endofpacket` for the background layer or a late `endofpacket` for any layer), it has to discard data until the `endofpacket` has been reached for all input layers.

This error recovery process maintains the synchronization between all the inputs and is started once the output frame is completed. A large number of samples may have to be discarded during the operation and backpressure can be applied for a long time on most input layers. Consequently, this error recovery mechanism could trigger an overflow at the input of the system.

☞ For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 6–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the Alpha Blending Mixer MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 6–2. Alpha Blending Mixer Latency

Mode	**Latency**
All modes	*O* (cycles)

The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 6–3 lists the Alpha Blending Mixer MegaCore function parameters.

Table 6–3. Alpha Blending Mixer Parameter Settings

Parameter	**Value**	**Description**
Maximum layer width	32–2600, Default = **1024**	Choose the maximum image width for the layer background in pixels. No layer width can be greater than the background layer width. The maximum image width is the default width for all layers at start-up.
Maximum layer height	32–2600, Default = **768**	Choose the maximum image height for the layer background in pixels. No layer height can be greater than the background layer height. The maximum image height is the default height for all layers at start-up.
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color planes in sequence	1–**3**	Choose the number of color planes that are sent in sequence over one data connection. For example, a value of 3 for R'G'B' R'G'B' R'G'B'.
Number of color planes in parallel	**1**–3	Choose the number of color planes in parallel.
Number of layers being mixed	**2**–12	Choose the number of image layers to overlay. Higher number layers are mixed on top of lower layer numbers. The background layer is always layer 0.
Alpha blending	On or **Off**	When on, alpha data sink ports are generated for each layer (including an unused port `alpha_in_0` for the background layer). This requires a stream of alpha values; one value for each pixel. When off, no alpha data sink ports are generated, and the image layers are fully opaque.
Alpha bits per pixel	2, 4, **8**	Choose the number of bits used to represent the alpha coefficient.

Signals

Table 6–4 lists the input and output signals for the Alpha Blending Mixer MegaCore function.

Table 6–4. Alpha Blending Mixer Signals (Part 1 of 2)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`alpha_in_N_data`	In	`alpha_in_N` port Avalon-ST `data` bus for layer *N*. This bus enables the transfer of pixel data into the MegaCore function. (1)
`alpha_in_N_endofpacket`	In	`alpha_in_N` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet. (1)
`alpha_in_N_ready`	Out	`alpha_in_N` port Avalon-ST alpha `ready` signal. This signal indicates when the MegaCore function is ready to receive data. (1)
`alpha_in_N_startofpacket`	In	`alpha_in_N` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet. (1)
`alpha_in_N_valid`	In	`alpha_in_N` port Avalon-ST alpha `valid` signal. This signal identifies the cycles when the port must input data. (1)
`control_av_address`	In	`control` slave port Avalon-MM `address` bus. Specifies a word offset into the slave address space.
`control_av_chipselect`	In	`control` slave port Avalon-MM `chipselect` signal. The `control` port ignores all other signals unless you assert this signal.
`control_av_readdata`	Out	`control` slave port Avalon-MM `readdata` bus. These output lines are used for read transfers.
`control_av_write`	In	`control` slave port Avalon-MM `write` signal. When you assert this signal, the `control` port accepts new data from the `writedata` bus.
`control_av_writedata`	In	`control` slave port Avalon-MM `writedata` bus. These input lines are used for write transfers.
`din_N_data`	In	`din_N` port Avalon-ST `data` bus for port `din` for layer *N*. This bus enables the transfer of pixel data into the MegaCore function.
`din_N_endofpacket`	In	`din_N` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_N_ready`	Out	`din_N` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_N_startofpacket`	In	`din_N` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_N_valid`	In	`din_N` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.

Table 6–4. Alpha Blending Mixer Signals (Part 2 of 2)

Signal	Direction	Description
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

Note to Table 6–4:

(1) These ports are present only if you turn on **Alpha blending**. Note that alpha channel ports are created for layer zero even though no alpha mixing is possible for layer zero (the background layer). These ports are ignored and can safely be left unconnected or tied to 0.

Control Register Maps

Table 6–5 describes the Alpha Blending Mixer MegaCore function control register map.

The width of each register in the Alpha Blending Mixer control register map is 16 bits. The control data is read once at the start of each frame and is buffered inside the MegaCore function, so the registers may be safely updated during the processing of a frame.

Table 6–5. Alpha Blending Mixer Control Register Map

Address	Register(s)	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the Alpha Blending Mixer MegaCore function to stop the next time control information is read. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Layer 1 X`	Offset in pixels from the left edge of the background layer to the left edge of layer 1. (1)
3	`Layer 1 Y`	Offset in pixels from the top edge of the background layer to the top edge of layer 1. (1)
4	`Layer 1 Active`	Layer 1 is displayed if this control register is set to 1. Data in the input stream is consumed but not displayed if this control register is set to 2, Avalon-ST packets of type 2 to 14 are still propagated as usual. Data from the input stream is not pulled out if this control register is set to 0. (1), (2).
5	`Layer 2 X`	…. (3)

Notes to Table 6–5:

(1) The value of this register is checked at the start of each frame. If the register is changed during the processing of a video frame, the change does not take effect until the start of the next frame.

(2) For efficiency reasons, the Video and Image Processing Suite MegaCore functions buffer a few samples from the input stream even if they are not immediately processed. This implies that the Avalon-ST inputs for foreground layers assert ready high and buffer a few samples even if the corresponding layer has been deactivated.

(3) The rows in the table are repeated in ascending order for each layer from 1 to the foreground layer.

7. Avalon-ST Video Monitor MegaCore Function

Core Overview

The Avalon-ST Video Monitor MegaCore function is a debugging and monitoring component. The monitor together with the associated software in the System Console allows you to capture and visualize the flow of video data in a system. You can inspect the video data flow at multiple levels of abstraction from the Avalon-ST video protocol level down to raw packet data level.

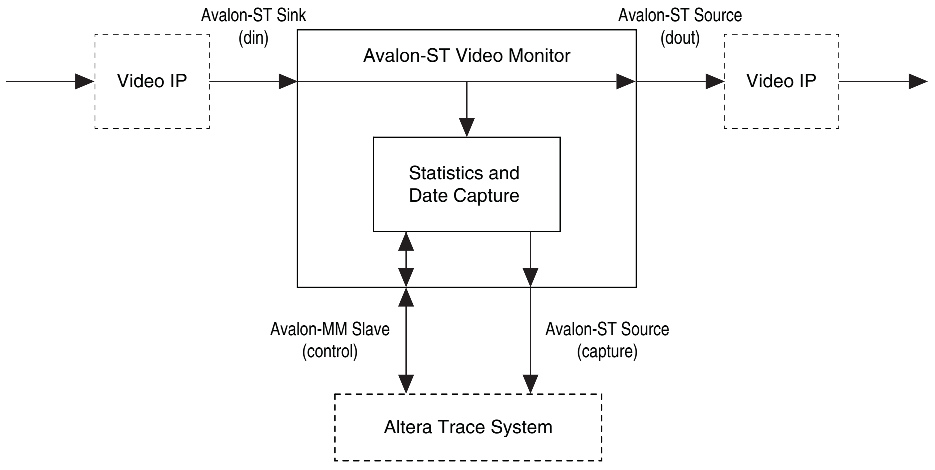
To know more about debugging designs with System Console, refer to the *Analyzing and Debugging Designs with the System Console* chapter in the *Quartus II Handbook*.

Functional Description

The Avalon-ST Video Monitor enables the visibility of the Avalon-ST video control and data packets streaming between video IP components. To monitor the video control and data packets, you must insert the monitor components into a system.

Figure 7–1 shows the monitor components in a system.

Figure 7–1. Avalon-ST Video MonitorFunctional Block Diagram



The monitored Avalon-ST video stream enters the monitor through the `din` Avalon-ST sink port and leaves the monitor through the `dout` Avalon-ST source port. The monitor does not modify, delay, or stall the video stream in any way. Inside the monitor, the stream is tapped for you to gather statistics and sample data. The statistics and sampled data are then transmitted through the `capture` Avalon-ST source port to the trace system component. The trace system component then transmits the received information to the host. You may connect multiple monitors to the Trace System.

The System Console uses the **sopcinfo** file written by Qsys to discover the connections between the trace system and the monitors. If you instantiate and manually connect the trace system and the monitors using HDL, the System Console will not detect them.

For more information on the Trace System, refer to Chapter 25, Trace System MegaCore Function.

Packet Visualization

The System Console contains a tabular view for displaying the information the monitors send out. Selecting a row in the trace table allows you to inspect a video packet in more detail. The following detailed information is available for a packet:

- Statistics—Data flow statistics such as backpressure. Refer to Table 7–1 for more information on the available statistics.
- Data—The sampled values for up to first 6 beats on the Avalon-ST data bus. `[n]` is the nth beat on the bus.
- Video control—Information about Avalon-ST video control packet.
- Video data—Packet size, the number of beats of the packet.

Table 7–1 lists the description of the available statistics.

Table 7–1. Statistics

Statistic	Description
Data transfer cycles (beats)	The number of cycles transferring data.
Not ready and valid cycles (backpressure)	The number of cycles between start of packet and end of packet during which the sink is not ready to receive data but the source has data to send.
Ready and not valid cycles (sink waiting)	The number of cycles between start of packet and end of packet during which the sink is ready to receive data but the source has no data to send.
Not ready and not valid cycles	The number of cycles between start of packet and end of packet during which the sink is not ready to receive data and the source has no data to send.
Inter packet valid cycles (backpressure)	The number of cycles before start of packet during which the sink is not ready to receive data but the source has data to send.
Inter packet ready cycles	The number of cycles before start of packet during which the sink is ready to receive data but the source has no data to send.

Table 7–1. Statistics

Statistic	Description
Backpressure	[(Not ready and valid cycles + Inter packet valid cycles) / (Data transfer cycles + Not ready and valid cycles + Ready and not valid cycles + Not ready and not valid cycles + Inter packet valid cycles)] × 100 Inter packet ready cycles are not included in the packet duration. A packet begins when a source is ready to send data.
Utilization	[Data transfer cycles / (Data transfer cycles + Not ready and valid cycles + Ready and not valid cycles + Not ready and not valid cycles + Inter packet valid cycles)] × 100 Inter packet ready cycles are not included in the packet duration. A packet begins when a source is ready to send data.

Monitor Settings

The capture settings panel of the trace table allows convenient access to the settings of the monitor. You can also change the monitor settings with the `trace_write_monitor` and `trace_read_monitor` TCL commands. At the hardware level, you can access the register map through the `control` Avalon-MM slave port of the monitor component.

Selecting the **Enable** settings available in the capture setting panel enables the sending of statistics and sampled data. If you do not select, the monitor is disabled.

Parameter Settings

Table 7–2 lists the Avalon-ST Video Monitor MegaCore function parameters.

Table 7–2. Avalon-ST Video Monitor Parameter Settings

Parameter	Value	Description
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color plane in parallel	1–3, Default = **3**	Choose the number of color planes that are sent in sequence over one data connection. For example, you set a value of 3 for R'G'B'R'G'B'R'G'B'.
Number of color planes in sequence	1–3, Default = **1**	Choose the number of color planes in parallel.
Bit width of capture interface(s)	8, 16, 32, 64, or 128, Default = **32**	Choose the data bus width of the Avalon-ST interface sending the captured information.

Signals

Table 7–3 lists the input and output signals for the Avalon-ST Video Monitor MegaCore function.

Table 7–3. Avalon-ST Video Monitor Signals (Part 1 of 2)

Signal	Direction	Description
clock_clk	In	All signals on the monitor are synchronous to this clock. Drive this signal from the clock which drives the video components that are being monitored. Do not insert clock crossing between the monitor and the trace system component. You must drive the trace system's clock from the same source which drives this signal.
reset_reset	In	This signal only resets the debugging parts of the monitor. It does not affect the system being monitored. Drive this signal directly from the reset_reset output of the trace system component.
din_data	In	din port Avalon-ST data bus.
din_endofpacket	In	din port Avalon-ST endofpacket signal.
din_ready	Out	din port Avalon-ST ready signal.
din_startdofpacket	In	din port Avalon-ST startofpacket signal.
din_valid	In	din port Avalon-ST valid signal.
dout_data	Out	dout port Avalon-ST data bus. This bus enables the transfer of pixel data out of the MegaCore function.
dout_endofpacket	Out	dout port Avalon-ST endofpacket signal. This signal marks the end of an Avalon-ST packet.
dout_ready	In	dout port Avalon-ST ready signal. The downstream device asserts this signal when it is able to receive data.
dout_startofpacket	Out	dout port Avalon-ST startofpacket signal. This signal marks the start of an Avalon-ST packet.
dout_valid	Out	dout port Avalon-ST valid signal. This signal is asserted when the MegaCore function outputs data.
capture_data	Out	capture port Avalon-ST data bus.
capture_endofpacket	Out	capture port Avalon-ST endofpacket signal.
capture_empty	Out	capture port Avalon-ST empty signal.
capture_ready	In	capture port Avalon-ST ready signal.
capture_startofpacket	Out	capture port Avalon-ST startofpacket signal.
capture_valid	Out	capture port Avalon-ST valid signal.
control_address	In	control slave port Avalon-MM address bus.
control_burstcount	In	control slave port Avalon-MM burstcount bus.
control_byteenable	In	control slave port Avalon-MM byteenable bus.
control_debugaccess	In	control slave port Avalon-MM debugaccess signal.
control_read	In	control slave port Avalon-MM read signal.
control_readdata	Out	control slave port Avalon-MM readdata bus.
control_readdatavalid	Out	control slave port Avalon-MM readdatavalid signal.
control_write	In	control slave port Avalon-MM write signal.

Table 7–3. Avalon-ST Video Monitor Signals (Part 2 of 2)

Signal	Direction	Description
`control_writedata`	In	`control` slave port Avalon-MM `writedata` bus.
`control_waitrequest`	Out	`control` slave port Avalon-MM `waitrequest` signal.

Control Register Map

Table 7–4 describes the Avalon-ST Video Monitor MegaCore function control register map.

Table 7–4. Avalon-ST Video Monitor Control Register Map

Address	Register(s)	Description
0	`Identity`	Read only register—manufacturer and monitor identities. Bits 11:0 are identities for the manufacturer, Altera = 0×6E Bits 27:12 are identities for the monitor, Avalon-ST video monitor = 0×110
1	`Configuration Information`	For use of the System Console only.
2	`Configuration Information`	For use of the System Console only.
3	`Configuration Information`	For use of the System Console only.
4	`Control`	Writing a 1 to bit 0 and bit 8 sends statistic counters. Writing a 1 to bit 0 and bit 9 sends up to first 6 beats on the Avalon-ST data bus. Writing a 0 to bit 0 disables both the statistics and beats.

8. Chroma Resampler MegaCore Function

Chroma Resampler

The Chroma Resampler MegaCore function resamples video data to and from common sampling formats. The human eye is more sensitive to brightness than tone. Taking advantage of this characteristic, video transmitted in the Y’CbCr color space often subsamples the color components (Cb and Cr) to save on data bandwidth.

Functional Description

The Chroma Resampler MegaCore function allows you to change between 4:4:4, 4:2:2 and 4:2:0 sampling rates where:

- 4:4:4 specifies full resolution in planes 1, 2, and 3
- 4:2:2 specifies full resolution in plane 1; half width resolution in planes 2 and 3
- 4:2:0 specifies full resolution in plane 1; half width and height resolution in planes 2 and 3

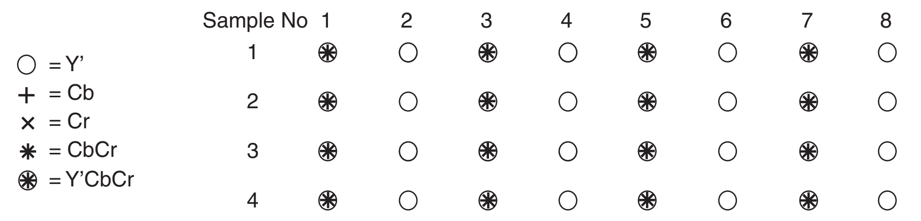
All modes of the Chroma Resampler assume the chrominance (chroma) and luminance (luma) samples are co-sited (that is, their values are sampled at the same time). The horizontal resampling process supports nearest-neighbor and filtered algorithms. The vertical resampling process only supports the nearest-neighbor algorithm.

The Chroma Resampler MegaCore function can be configured to change image size at run time using control packets.

Horizontal Resampling (4:2:2)

Figure 8–1 shows the location of samples in a co-sited 4:2:2 image.

Figure 8–1. Resampling 4.4.4 to a 4.2.2 Image



Conversion from sampling rate 4:4:4 to 4:2:2 and back are scaling operations on the chroma channels. This means that these operations are affected by some of the same issues as the Scaler MegaCore function. However, because the scaling ratio is fixed as 2× up or 2× down, the Chroma Resampler MegaCore function is highly optimized for these cases.

The Chroma Resampler MegaCore Function only supports the cosited form of horizontal resampling—the form for 4:2:2 data in *ITU Recommendation BT.601*, MPEG-2, and other standards.

For more information about the ITU standard, refer to *Recommendation ITU-R BT.601, Encoding Parameters of Digital Television for Studios, 1992, International Telecommunications Union, Geneva*.

4:4:4 to 4:2:2

The nearest-neighbor algorithm is the simplest way to down-scale the chroma channels. It works by simply discarding the Cb and Cr samples that occur on even columns (assuming the first column is numbered 1). This algorithm is very fast and cheap but, due to aliasing effects, it does not produce the best image quality.

To get the best results when down-scaling, you can apply a filter to remove high-frequency data and thus avoid possible aliasing. The filtered algorithm for horizontal subsampling uses a 9-tap filter with a fixed set of coefficients.

The coefficients are based on a Lanczos-2 function ("Choosing and Loading Coefficients" on page 21–6) that the Scaler MegaCore function uses. Their quantized form is known as the Turkowski Decimator.

For more information about the Turkowski Decimator, refer to *Ken Turkowski. Graphics Gems, chapter Filters for common resampling tasks, pages 147–165. Academic Press Professional, Inc., San Diego, CA, USA, 1990.*

The coefficients are fixed and approximate to powers of two, therefore they can be implemented by bit-shifts and additions. This algorithm efficiently eliminates aliasing in the chroma channels, and uses no memory or multipliers. However, it does use more logic area than the nearest-neighbor algorithm.

4:2:2 to 4:4:4

The nearest-neighbor algorithm is the simplest way to up-scale the chroma channels. It works by simply duplicating each incoming Cb and Cr sample to fill in the missing data. This algorithm is very fast and cheap but it tends to produce sharp jagged edges in the chroma channels.

The filtered algorithm uses the same method as the Scaler MegaCore function would use for upscaling, that is a four-tap filter with Lanczos-2 coefficients. Use this filter with a phase offset of 0 for the odd output columns (those with existing data) and an offset of one-half for the even columns (those without direct input data). A filter with phase offset 0 has no effect, so the function implements it as a pass-through filter. A filter with phase offset of one-half interpolates the missing values and has fixed coefficients that bit-shifts and additions implement.

This algorithm performs suitable upsampling and uses no memory or multipliers. It uses more logic elements than the nearest-neighbor algorithm and is not the highest quality available.

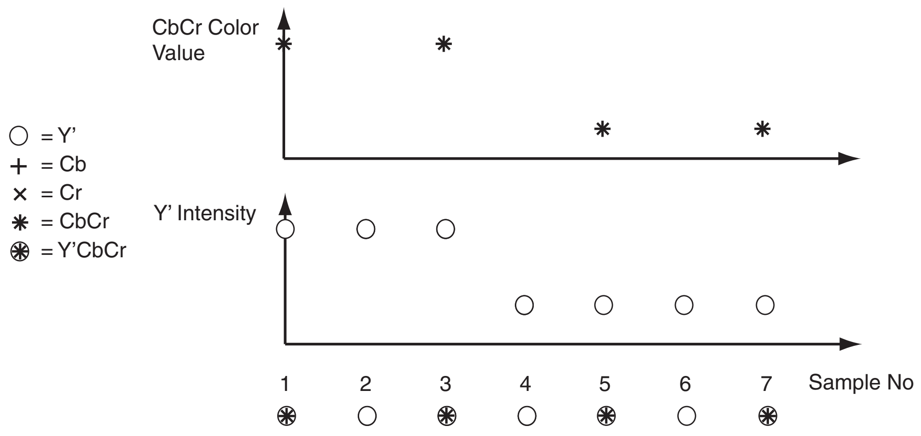
The best image quality for upsampling is obtained by using the filtered algorithm with luma-adaptive mode enabled. This mode looks at the luma channel during interpolation and uses this to detect edges. Edges in the luma channel make appropriate phase-shifts in the interpolation coefficients for the chroma channels.

Figure 8–2 shows 4:2:2 data at an edge transition. Without taking any account of the luma, the interpolation to produce chroma values for sample 4 would weight samples 3 and 5 equally. From the luma, you can see that sample 4 falls on an the low side of an edge, so sample 5 is more significant than sample 3.

The luma-adaptive mode looks for such situations and chooses how to adjust the interpolation filter. From phase 0, it can shift to -1/4, 0, or 1/4; from phase 1/2, it can shift to 1/4, 1/2, or 3/4. This makes the interpolated chroma samples line up better with edges in the luma channel and is particularly noticeable for bold synthetic edges such as text.

The luma-adaptive mode uses no memory or multipliers, but requires more logic elements than the straightforward filtered algorithm.

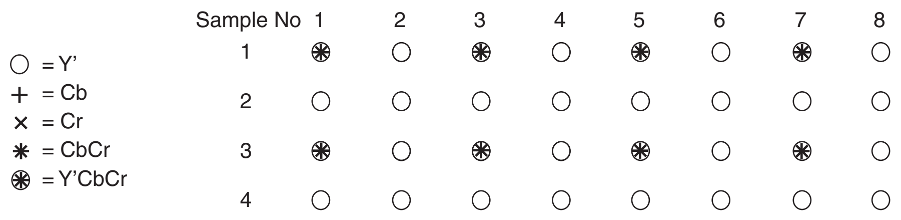
Figure 8–2. 4:2:2 Data at an Edge Transition



Vertical Resampling (4:2:0)

The Chroma Resampler MegaCore function does not distinguish interlaced data with its vertical resampling mode. It only supports the co-sited form of vertical resampling shown in Figure 8–3.

Figure 8–3. Resampling 4.4.4 to a 4.2.0 Image



For both upsampling and downsampling, the vertical resampling algorithm is fixed at nearest-neighbor.

Vertical resampling does not use any multipliers. For upsampling, it uses four line buffers, each buffer being half the width of the image. For downsampling it uses one line buffer which is half the width of the image.

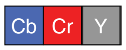
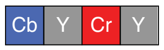
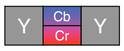
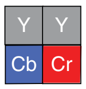
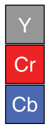
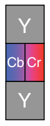
All input data samples must be in unsigned format. If the number of bits per pixel per color plane is *N*, this means that each sample consists of *N* bits of data which are interpreted as an unsigned binary number in the range [0, $2^N - 1$]. All output data samples are also in the same unsigned format.

For more information about how non-video packets are transferred, refer to "Packet Propagation" on page 3–11.

Avalon-ST Video Protocol Parameters

The Chroma Resampler MegaCore function can process streams of pixel data of the types listed in Table 8–1.

Table 8–1. Chroma Resampler Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Maximum frame width is specified in the parameter editor, the actual value is read from control packets.
Frame Height	Maximum frame height is specified in the parameter editor, the actual value is read from control packets.
Interlaced / Progressive	Progressive.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	For 4:4:4 sequential data:  For 4:2:2 sequential data:  For 4:2:0 sequential data:  For 4:2:2 parallel data:  For 4:4:4 parallel data:  For 4:2:0 parallel data: 

Stall Behavior and Error Recovery

All modes of the Chroma Resampler MegaCore function stall for a few cycles between frames and between lines. Latency from input to output varies depending on the operation mode of the Chroma Resampler MegaCore function. The only modes with latency of more than a few cycles are 4:2:0 to 4:2:2 and 4:2:0 to 4:4:4. These modes have a latency corresponding to one line of 4:2:0 data.

Because this is a rate-changing function, the quantities of data input and output are not equal. The Chroma Resampler MegaCore function always outputs the same number of lines that it inputs. However the number of samples in each line varies according to the subsampling pattern used.

When not stalled, the Chroma Resampler always processes one sample from the more fully sampled side on each clock cycle. For example, the subsampled side pauses for one third of the clock cycles in the 4:2:2 case or half of the clock cycles in the 4:2:0 case.

Error Recovery

On receiving an early `endofpacket` signal, the Chroma Resampler stalls its input but continues writing data until it has sent an entire frame. If it does not receive an `endofpacket` signal at the end of a frame, the Chroma Resampler discards data until the end-of-packet is found.

☞ For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 8–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the Chroma Resampler MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 8–2. Chroma Resampler Latency

Mode	**Latency**
Input format: 4:2:2; Output format: 4:4:4	*O* (cycles)
Input format: 4:2:0; Output format: 4:4:4 or 4:2:2	1 line + *O* (cycles)

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 8–3 lists the Chroma Resampler MegaCore function parameters.

Table 8–3. Chroma Resampler Parameter Settings (Part 1 of 2)

Parameter	**Value**	**Description**
Maximum width	32–2600, Default = **256**	Choose the maximum image width in pixels.
Maximum height	32–2600, Default = **256**	Choose the maximum image height in pixels.
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Color plane configuration	**Sequence**, Parallel	There must always be three color planes for this function but you can choose whether the three color planes are transmitted in sequence or in parallel.

Table 8–3. Chroma Resampler Parameter Settings (Part 2 of 2)

Parameter	Value	Description
Input Format (1)	4:4:4, **4:2:2**, 4:2:0	Choose the format/sampling rate format for the input frames. Note that the input and output formats must be different.
Output Format (1)	**4:4:4**, 4:2:2, 4:2:0	Choose the format/sampling rate format for the output frames. Note that the input and output formats must be different.
Horizontal Filtering Algorithm	**Filtered**, Nearest Neighbor	Choose the algorithm to use in the horizontal direction when re-sampling data to or from 4:4:4.
Luma adaptive	**On** or Off	Turn on to enable luma-adaptive mode. This mode looks at the luma channel during interpolation and uses this to detect edges.

Note to Table 8–3:

(1) The input and output formats must be different. A warning is issued when the same values are selected for both.

Signals

Table 8–4 lists the input and output signals for the Chroma Resampler MegaCore function.

Table 8–4. Chroma Resampler Signals

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

9. Clipper MegaCore Function

Core Overview

The Clipper MegaCore function clips video streams. You can configure the Clipper at compile time or optionally at run time using an Avalon-MM slave interface.

Functional Description

The Clipper MegaCore function provides a means to select an active area from a video stream and discard the remainder.

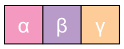
The active region can be specified by either providing the offsets from each border, or by providing a point to be the top-left corner of the active region along with the region's width and height.

The Clipper can deal with changing input resolutions by reading Avalon-ST Video control packets. An optional Avalon-MM interface allows the clipping settings to be changed at run time.

Avalon-ST Video Protocol Parameters

The Clipper MegaCore function can process streams of pixel data of the types listed in Table 9–1.

Table 9–1. Clipper Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Maximum frame width is specified in the parameter editor, the actual value is read from control packets.
Frame Height	Maximum frame height is specified in the parameter editor, the actual value is read from control packets.
Interlaced / Progressive	Either. Interlaced inputs are accepted but are treated as progressive inputs.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	Any combination of one, two, three, or four channels in each of sequence or parallel. For example, if three channels in sequence is selected where α, β, and γ can be any color plane: 

Stall Behavior and Error Recovery

The Clipper MegaCore function stalls for a few cycles between lines and between frames. Its internal latency is less than 10 cycles. During the processing of a line, it reads continuously but the Clipper only writes when inside the active picture area as defined by the clipping window.

Error Recovery

On receiving an early `endofpacket` signal, the Clipper stalls its input but continues writing data until it has sent an entire frame. If it does not receive an `endofpacket` signal at the end of a frame, the Clipper discards data until the end-of-packet is found.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 9–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the Clipper MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 9–2. Clipper Latency

Mode	Latency
All modes	*O* (cycles)

The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 9–3 lists the Clipper MegaCore function parameters.

Table 9–3. Clipper Parameter Settings (Part 1 of 2)

Parameter	Value	Description
Maximum width	32 to input image width, Default = **1024**	Specify the maximum width of the clipping rectangle for the input field (progressive or interlaced).
Maximum height	32 to input image height, Default = **768**	Specify the maximum height of the clipping rectangle for the input field (progressive or interlaced).
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color planes in sequence	1–**3**	Choose the number of color planes that are sent in sequence over one data connection. For example, a value of 3 for R'G'B' R'G'B' R'G'B'.
Number of color planes in parallel	**1**–3	Choose the number of color planes in parallel.

Table 9–3. Clipper Parameter Settings (Part 2 of 2)

Parameter	Value	Description
Include Avalon-MM interface	On or **Off**	Turn on if you want to specify clipping offsets using the Avalon-MM interface.
Clipping method	**Offsets**, Rectangle	Choose whether to specify the clipping area as offsets from the edge of the input area or as a fixed rectangle.
Left offset	positive integer, Default = **10**	Specify the x coordinate for the left edge of the clipping rectangle. 0 is the left edge of the input area. (1)
Right offset	positive integer, Default = **10**	Specify the x coordinate for the right edge of the clipping rectangle. 0 is the right edge of the input area. (1)
Width	positive integer, Default = **10**	Specify the width of the clipping rectangle.
Top offset	positive integer, Default = **10**	Specify the y coordinate for the top edge of the clipping rectangle. 0 is the top edge of the input area. (2)
Bottom offset	positive integer, Default = **10**	Specify the y coordinate for the bottom edge of the clipping rectangle. 0 is the bottom edge of the input area. (2)
Height	positive integer, Default = **10**	Specify the height of the clipping rectangle.

Notes to Table 9–3:

(1) The left and right offset values must be less than or equal to the input image width.

(2) The top and bottom offset values must be less than or equal to the input image height.

Signals

Table 9–4 lists the input and output signals for the Clipper MegaCore function.

Table 9–4. Clipper Signals (Part 1 of 2)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`control_av_address`	In	`control` slave port Avalon-MM `address` bus. Specifies a word offset into the slave address space. (1)
`control_av_chipselect`	In	`control` slave port Avalon-MM `chipselect` signal. The `control` port ignores all other signals unless you assert this signal. (1)
`control_av_readdata`	Out	`control` slave port Avalon-MM `readdata` bus. These output lines are used for read transfers. (1)
`control_av_waitrequest`	Out	`control` slave port Avalon-MM `waitrequest` signal. (1)
`control_av_write`	In	`control` slave port Avalon-MM `write` signal. When you assert this signal, the `control` port accepts new data from the `writedata` bus. (1)
`control_av_writedata`	In	`control` slave port Avalon-MM `writedata` bus. These input lines are used for write transfers. (1)
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.

Table 9–4. Clipper Signals (Part 2 of 2)

Signal	Direction	Description
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`dout_data`	Out	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

Note to Table 9–4:

(1) These ports are present only if you turn on **Include Avalon-MM interface**.

Control Register Maps

Table 9–5 lists the Clipper MegaCore function control register map.

The control data is read once at the start of each frame and is buffered inside the MegaCore function, so the registers can be safely updated during the processing of a frame. Note that all Clipper registers are write-only except at address 1.

Table 9–5. Clipper Control Register Map

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the Clipper MegaCore function to stop the next time control information is read. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. The Clipper MegaCore function sets this address to 0 between frames. It is set to 1 while the MegaCore function is processing data and cannot be stopped. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Left Offset`	The left offset, in pixels, of the clipping window/rectangle. (1)
3	`Right Offset or Width`	In clipping window mode, the right offset of the window. In clipping rectangle mode, the width of the rectangle. (1)
4	`Top Offset`	The top offset, in pixels, of the clipping window/rectangle. (2)
5	`Bottom Offset or Height`	In clipping window mode, the bottom offset of the window. In clipping rectangle mode, the height of the rectangle. (2)

Notes to Table 9–5:

(1) The left and right offset values must be less than or equal to the input image width.

(2) The top and bottom offset values must be less than or equal to the input image height.

10. Clocked Video Input MegaCore Function

Core Overview

The Clocked Video Input MegaCore function converts clocked video formats (such as BT656, BT1120, and DVI) to Avalon-ST Video. You can configure the Clocked Video Input at run time using an Avalon-MM slave interface.

Functional Description

The Clocked Video Input converts clocked video to the flow controlled Avalon-ST Video protocol. It also provides clock crossing capabilities to allow video formats running at different frequencies to enter the system.

The Clocked Video Input strips the incoming clocked video of horizontal and vertical blanking, leaving only active picture data, and using this data with the horizontal and vertical synchronization information creates the necessary Avalon-ST Video control and active picture packets. No conversion is done to the active picture data, the color plane information remains the same as in the clocked video format.

In addition, the Clocked Video Input provides a number of status registers that provide feedback on the format of video entering the system (resolution, and interlaced or progressive mode) and a status interrupt that can be used to determine when the video format changes or is disconnected.

Video Formats

The Clocked Video Input MegaCore function accepts the following clocked video formats:

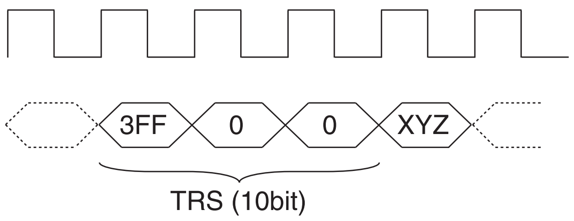
- Video with synchronization information embedded in the data (in BT656 or BT1120 format)
- Video with separate synchronization (H sync, Vsync) signals

Embedded Synchronization Format

The BT656 and BT1120 formats use time reference signal (TRS) codes in the video data to mark the places where synchronization information is inserted in the data.

These codes are made up of values that are not present in the video portion of the data and take the format shown in Figure 10–1.

Figure 10–1. Time Reference Signal Format



The Clocked Video Input MegaCore function supports both 8 and 10-bit TRS and XYZ words. When in 10-bit mode the bottom 2 bits of the TRS and XYZ words are ignored to allow easy transition from an 8-bit system.

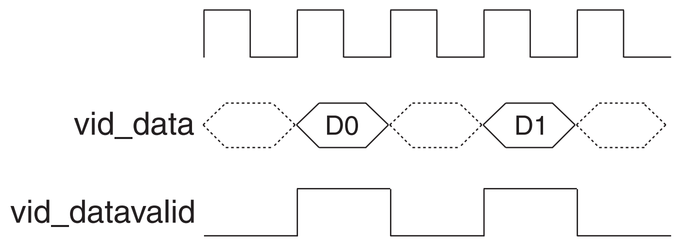
The XYZ word contains the synchronization information and the relevant bits of it's format are listed in Table 10–1.

Table 10–1. XYZ Word Format

	10-bit	8-bit	Description
Unused	[5:0]	[3:0]	These bits are not inspected by the Clocked Video Input MegaCore function.
H (sync)	6	4	When 1, the video is in a horizontal blanking period.
V (sync)	7	5	When 1, the video is in a vertical blanking period.
F (field)	8	6	When 1, the video is interlaced and in field 1. When 0, the video is either progressive or interlaced and in field 0.
Unused	9	7	These bits are not inspected by the Clocked Video Input MegaCore function.

For the embedded synchronization format, the `vid_datavalid` signal indicates a valid BT656 or BT1120 sample as shown in Figure 10–2. The Clocked Video Input MegaCore function only reads the `vid_data` signal when `vid_datavalid` is 1.

Figure 10–2. vid_datavalid Timing



The Clocked Video Input MegaCore function extracts any ancillary packets from the Y channel during the vertical blanking. Ancillary packets are not extracted from the horizontal blanking. The extracted packets are output via the Clocked Video Input's Avalon-ST output with a packet type of 13 (0xD). For information about Avalon-ST Video ancillary data packets, refer to "Ancillary Data Packets" on page 3–10.

Separate Synchronization Format

The separate synchronization format uses separate signals to indicate the blanking, sync, and field information. For this format, the `vid_datavalid` signal behaves slightly differently from in embedded synchronization format.

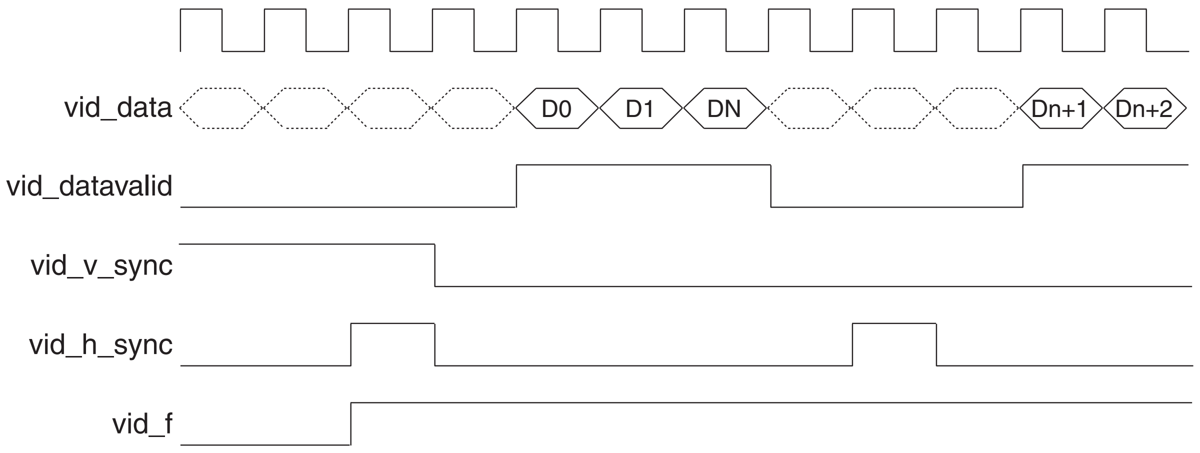
The Clocked Video Input MegaCore function only reads `vid_data` when `vid_datavalid` is high (as in the embedded synchronization format) but it treats each read sample as active picture data.

Table 10–2 lists the signals and Figure 10–3 shows the timing.

Table 10–2. Clocked Video Input Signals for Separate Synchronization Format Video

Signal Name	Description
`vid_datavalid`	When asserted the video is in an active picture period (not horizontal or vertical blanking).
`vid_h_sync`	When 1, the video is in a horizontal synchronization period.
`vid_v_sync`	When 1, the video is in a vertical synchronization period.
`vid_f`	When 1, the video is interlaced and in field 1. When 0, the video is either progressive or interlaced and in field 0.

Figure 10–3. Separate Synchronization Signals Timing



Video Locked Signal

The `vid_locked` signal indicates that the clocked video stream is active. When the signal has a value of 1, the Clocked Video Input MegaCore function takes the input clocked video signals as valid and reads and processes them as normal.

When the signal has a value of 0 (if for example the video cable is disconnected or the video interface is not receiving a signal) the Clocked Video Input MegaCore function takes the input clocked video signals as invalid and does not process them.

If the `vid_locked` signal goes invalid while a frame of video is being processed, the Clocked Video Input MegaCore function ends the frame of video early.

Control Port

If you turn on **Use control port** in the parameter editor for the Clocked Video Input, its Avalon-ST Video output can be controlled using the Avalon-MM slave control port.

Initially, the MegaCore function is disabled and does not output any data. However, it still detects the format of the clocked video input and raises interrupts.

The sequence for starting the output of the MegaCore function is as follows:

1. Write a 1 to `Control` register bit 0.

2. Read `Status` register bit 0. When this is a 1, the MegaCore function outputs data. This occurs on the next start of frame or field that matches the setting of the **Field order** in the parameter editor.

The sequence for stopping the output of the MegaCore function is as follows:

1. Write a 0 to `Control` register bit 0.
2. Read `Status` register bit 0. When this is a 0, the MegaCore function has stopped data output. This occurs on the next end of frame or field that matches the setting of the **Field order** in the parameter editor.

The starting and stopping of the MegaCore function is synchronized to a frame or field boundary.

Table 10–3 lists the output of the MegaCore function with the different **Field order** settings.

Table 10–3. Synchronization Settings

Video Format	Field Order	Output
Interlaced	F1 first	Start, F1, F0, ..., F1, F0, Stop
Interlaced	F0 first	Start, F0, F1, ..., F0, F1, Stop
Interlaced	Any field first	Start, F0 or F1, ... F0 or F1, Stop
Progressive	F1 first	No output
Progressive	F0 first	Start, F0, F0, ..., F0, F0, Stop
Progressive	Any field first	Start, F0, F0, ..., F0, F0, Stop

Format Detection

The Clocked Video Input MegaCore function detects the format of the incoming clocked video and uses it to create the Avalon-ST Video control packet. It also provides this information in a set of registers.

The MegaCore function can detect the following different aspects of the incoming video stream:

- Picture width (in samples)—The MegaCore function counts the total number of samples per line, and the number of samples in the active picture period. One full line of video is required before the MegaCore function can determine the width.
- Picture height (in lines)—The MegaCore function counts the total number of lines per frame or field, and the number of lines in the active picture period. One full frame or field of video is required before the MegaCore function can determine the height.
- Interlaced/Progressive—The MegaCore function detects whether the incoming video is interlaced or progressive. If it is interlaced, separate height values are stored for both fields. One full frame or field of video and a number of lines from a second frame or field are required before the MegaCore function can determine whether the source is interlaced or progressive.
- Standard—The MegaCore function provides the contents of the `vid_std` bus via the `Standard` register. When connected to the `rx_std` signal of a SDI MegaCore function, for example, these values can be used to report the standard (SD, HD, or 3G) of the incoming video.

If the MegaCore function has not yet determined the format of the incoming video, it uses the values specified under the **Avalon-ST Video Initial/Default Control Packet** section in the parameter editor.

After determining an aspect of the incoming videos format, the MegaCore function enters the value in the respective register, sets the registers valid bit in the `Status` register, and triggers the respective interrupts.

Table 10–4 lists the sequence for a 1080i incoming video stream.

Table 10–4. Resolution Detection Sequence for a 1080i Incoming Video Stream

Status	Interrupt	Active Sample Count	F0 Active Line Count	F1 Active Line Count	Total Sample Count	F0 Total Sample Count	F1 Total Sample Count	Description
00000000000	000	0	0	0	0	0	0	Start of incoming video.
00000101000	000	1,920	0	0	2,200	0	0	End of first line of video.
00100101000	100	1,920	0	0	2,200	0	0	Stable bit set and interrupt fired —Two of last three lines had the same sample count.
00100111000	100	1,920	540	0	2,200	563	0	End of first field of video.
00110111000	100	1,920	540	0	2,200	563	0	Interlaced bit set—Start of second field of video.
00111111000	100	1,920	540	540	2,200	563	562	End of second field of video.
10111111000	110	1,920	540	540	2,200	563	562	Resolution valid bit set and interrupt fired.

Interrupts

The Clocked Video Input MegaCore function outputs a single interrupt line which is the `OR` of the following internal interrupts:

- The status update interrupt—Triggers when a change of resolution in the incoming video is detected.
- Stable video interrupt—Triggers when the incoming video is detected as stable (has a consistent sample length in two of the last three lines) or unstable (if, for example, the video cable is removed). The incoming video is always detected as unstable when the `vid_locked` signal is low.

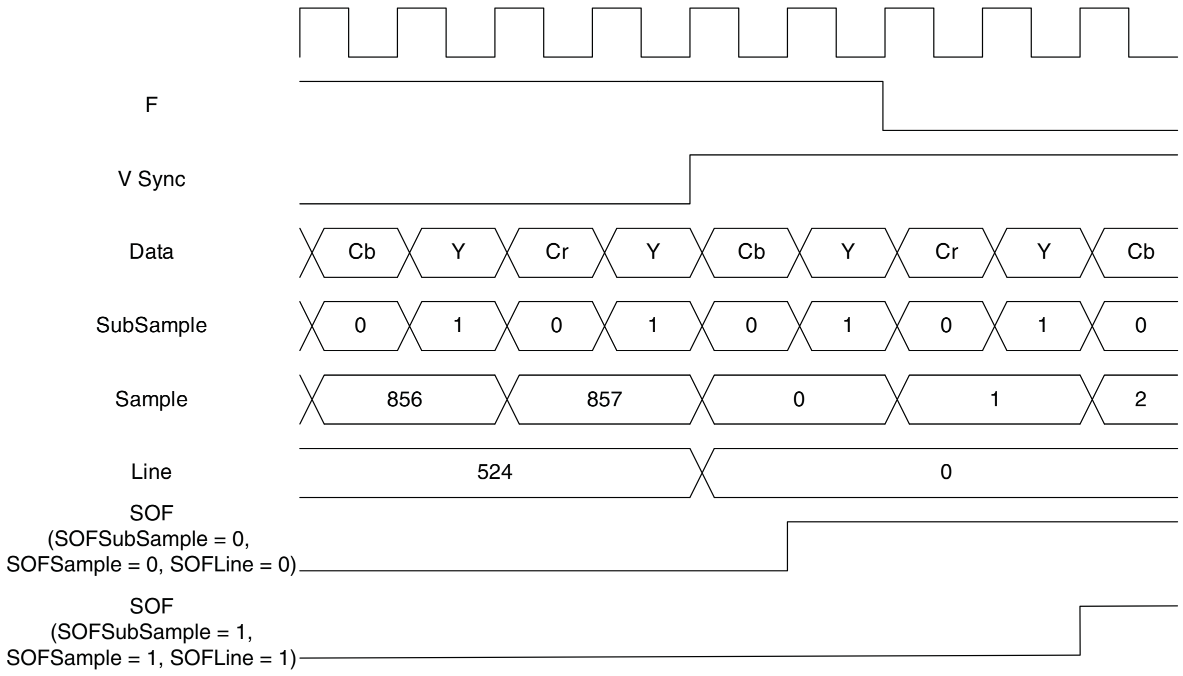
Both interrupts can be independently enabled using bits [2:1] of the `Control` register. Their values can be read using bits [2:1] of the `Interrupt` register and a write of 1 to either of those bits clears the respective interrupt.

Generator Lock

Generator lock (Genlock) is the technique for locking the timing of video outputs to a reference source. Sources that are locked to the same reference can be switched between cleanly, on a frame boundary. The Genlock functionality is enabled using the Control register.

The Clocked Video Input MegaCore function provides some functions to facilitate Genlock. The MegaCore function can be configured to output, via the `refclk_div` signal, a divided down version of its `vid_clk` (`refclk`) aligned to the start of frame (SOF). By setting the divide down value to the length in samples of a video line, the `refclk_div` signal can be configured to output a horizontal reference which a phase-locked loop (PLL) can align its output clock to. By tracking changes in the `refclk_div` signal, the PLL can then ensure that its output clock is locked to the incoming video clock. Figure 10–4 shows an example configuration.

Figure 10–4. Genlock Example Configuration


F
V Sync
Data
Cb
Y
Cr
Y
Cb
Y
Cr
Y
Cb
SubSample
0
1
0
1
0
1
0
1
0
Sample
856
857
0
1
2
Line
524
0
SOF (SOFSubSample = 0, SOFSample = 0, SOFLine = 0)
SOF (SOFSubSample = 1, SOFSample = 1, SOFLine = 1)

The SOF signal can be set to any position within the incoming video frame. The registers used to configure the SOF signal are measured from the rising edge of the F0 vertical sync. Due to registering inside the Clocked Video Input MegaCore function setting the SOF Sample and SOF Line registers to 0 results in a SOF signal rising edge six cycles after the rising edge of the vsync, in embedded synchronization mode, and three cycles after the rising edge of the vsync, in separate synchronization mode. A start of frame is indicated by a rising edge on the SOF signal (0 to 1).

An example of how to set up the Clocked Video Input to output an SOF signal aligned to the incoming video synchronization (in embedded synchronization mode) is listed in Table 10–5.

Table 10–5. Example of Clocked Video Input To Output an SOF Signal

Format	SOF Sample Register	SOF Line Register	Refclk Divider Register
720p60	1644 << 2	749	1649
1080i60	2194 << 2	1124	2199
NTSC	856 << 2	524	857

A Clocked Video Output MegaCore function can take in the locked PLL clock and the SOF signal and align the output video to these signals. This produces an output video frame that is synchronized to the incoming video frame. For more information, refer to "Clocked Video Output MegaCore Function" on page 11–1.

Overflow

Moving between the domain of clocked video and the flow controlled world of Avalon-ST Video can cause problems if the flow controlled world does not accept data at a rate fast enough to satisfy the demands of the incoming clocked video.

The Clocked Video Input MegaCore function contains a FIFO that, when set to a large enough value, can accommodate any bursts in the flow data, as long as the input rate of the upstream Avalon-ST Video components is equal to or higher than that of the incoming clocked video.

If this is not the case, the FIFO overflows. If overflow occurs, the MegaCore function outputs an early endofpacket signal to complete the current frame. It then waits for the next start of frame (or field) before re-synchronizing to the incoming clocked video and beginning to output data again.

The overflow is recorded in bit [9] of the `Status` register. This bit is sticky, and if an overflow occurs, stays at 1 until the bit is cleared by writing a 0 to it.

In addition to the overflow bit, the current level of the FIFO can be read from the `Used Words` register.

Timing Constraints

To constrain the Clocked Video Output MegaCore function correctly, add the following file to your Quartus II project:

<*install_dir*>**\ip\clocked_video_input\lib\alt_vip_cvi.sdc**

When you apply the SDC file, you may see some warning messages in a format as follows:

- Warning: At least one of the filters had some problems and could not be matched.
- Warning: * could not be matched with a keeper.

These warnings are expected, because in certain configurations the Quartus II software optimizes unused registers and they no longer remain in your design.

Active Format Description Extractor

The AFD Extractor is an example of how to write a core to handle ancillary packets. It is available in the following directory:

<*install_dir*>**\ip\clocked_video_output\lib\afd_example**

When the output of the Clocked Video Input MegaCore function is connected to the input of the AFD Extractor, the AFD Extractor removes any ancillary data packets from the stream and checks the DID and secondary DID (SDID) of the ancillary packets contained within each ancillary data packet. If the packet is an AFD packet (DID = 0x41, SDID = 0x5), the extractor places the contents of the ancillary packet into the AFD Extractor register map.

Refer to the SMPTE 2016-1-2007 standard for a more detailed description of the AFD codes.

Table 10–6 lists the AFD Extractor register map.

Table 10–6. AFD Extractor Register Map

Address	Register	Description
0	Control	When bit 0 is 0, the core discards all packets. When bit 0 is 1, the core passes through all non-ancillary packets.
1		Reserved.
2	Interrupt	When bit 1 is 1, a change to the AFD data has been detected and the interrupt has been set. Writing a 1 to bit 1 clears the interrupt.
3	AFD	Bits 0-3 contain the active format description code.
4	AR	Bit 0 contains the aspect ratio code.
5	Bar data flags	When AFD is 0000 or 0100, bits 0-3 describe the contents of bar data value 1 and bar data value 2. When AFD is 0011, bar data value 1 is the pixel number end of the left bar and bar data value 2 is the pixel number start of the right bar. When AFD is 1100, bar data value 1 is the line number end of top bar and bar data value 2 is the line number start of bottom bar.
6	Bar data value 1	Bits 0-15 contain bar data value 1
7	Bar data value 2	Bits 0-15 contain bar data value 2
8	AFD valid	When bit 0 is 0, an AFD packet is not present for each image packet. When bit 0 is 1, an AFD packet is present for each image packet.

Stall Behavior and Error Recovery

The stall behavior of the Clocked Video Input MegaCore function is dictated by the incoming video. If its output FIFO is empty, during horizontal and vertical blanking periods the Clocked Video Input does not output any video data.

Error Recovery

If an overflow is caused by a downstream core failing to receive data at the rate of the incoming video, the Clocked Video Input MegaCore function sends an early end of packet and restart sending video data at the start of the next frame or field.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 10–7 lists the approximate latency from the video data input to the video data output for typical usage modes of the Clocked Video Input MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 10–7. Clocked Video Input Latency

Mode	Latency (1)
Synchronization signals: Embedded in video Video in and out use the same clock: On	8 cycles
Synchronization signals: On separate wires Video in and out use the same clock: On	5 cycles

Note to Table 10–7:

(1) Add 1 cycle if **Allow color planes in sequence input** is turned on.

The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 10–8 lists the Clocked Video Input MegaCore function parameters.

Table 10–8. Clocked Video Input Parameter Settings (Part 1 of 2)

Parameter	Value	Description
Select preset to load	**DVI 1080p60**, SDI 1080p60, SDI 1080i60, PAL, NTSC	You can choose from a list of preset conversions or use the other fields in the dialog box to set up custom parameter values. If you click **Load values into controls** the dialog box is initialized with values for the selected preset conversion.
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color planes	1–4, Default = **3**	Choose the number of color planes.
Color plane transmission format	Sequence, **Parallel**	Choose whether the color planes are transmitted in sequence or in parallel.
Field order	**Field 0 first**, Field 1 first, Any field first,	Choose the field to synchronize to first when starting or stopping the output.

Table 10–8. Clocked Video Input Parameter Settings (Part 2 of 2)

Parameter	Value	Description
Interlaced or progressive	**Progressive**, Interlaced	Choose the format to be used when no format can be automatically detected.
Width	32–65,536, Default = **1920**	Choose the image width to be used when no format can be automatically detected.
Height, Frame / Field 0	32–65,536, Default = **1080**	Choose the image height to be used when no format can be automatically detected.
Height, Field 1	32–65,536, Default = **1080**	Choose the image height for interlaced field 1when no format can be automatically detected.
Sync Signals	Embedded in video, **On separate wires**	Choose whether the synchronization signal is embedded in the video stream or provided on a separate wire.
Allow color planes in sequence input	On or **Off**	Choose whether run-time switching is allowed between sequential and parallel color plane transmission formats. The format is controlled by the `vid_hd_sdn` signal.
Generate synchronization outputs	**No**, Yes, Only	Specifies whether the Avalon-ST output and synchronization outputs (`sof`, `sof_locked`, `refclk_div`) are generated: ■ No—Only Avalon-ST Video output ■ Yes—Avalon-ST Video output and synchronization outputs ■ Only—Only synchronization outputs
Width of bus "vid_std"	**1 - 16**	The width, in bits, of the vid_std bus.
Extract ancillary packets	On or **Off**	Specifies whether ancillary packets are extracted in embedded sync mode.
Pixel FIFO size	32–(memory limit), Default = **1920**	Choose the required FIFO depth in pixels (limited by the available on-chip memory).
Video in and out use the same clock	On or **Off**	Turn on if you want to use the same signal for the input and output video image stream clocks.
Use control port	On or **Off**	Turn on to use the optional stop/go control port.

Signals

Table 10–9 lists the input and output signals for the Clocked Video Input MegaCore function.

Table 10–9. Clocked Video Input Signals (Part 1 of 3)

Signal	Direction	Description
`rst`	In	The MegaCore function asynchronously resets when you assert `rst`. You must deassert `rst` synchronously to the rising edge of the `is_clk` signal.
`vid_clk`	In	Clocked video clock. All the video input signals are synchronous to this clock.
`av_address`	In	`control` slave port Avalon-MM address bus. Specifies a word offset into the slave address space. (1)
`av_read`	In	`control` slave port Avalon-MM read signal. When you assert this signal, the `control` port drives new data onto the read data bus. (1)
`av_readdata`	Out	`control` slave port Avalon-MM read data bus. These output lines are used for read transfers. (1)
`av_write`	In	`control` slave port Avalon-MM write signal. When you assert this signal, the `control` port accepts new data from the write data bus. (1)

Table 10–9. Clocked Video Input Signals (Part 2 of 3)

Signal	Direction	Description
av_writedata	In	control slave port Avalon-MM write data bus. These input lines are used for write transfers. (1)
is_clk	In	Clock signal for Avalon-ST ports dout and control. The MegaCore function operates on the rising edge of the is_clk signal.
is_data	Out	dout port Avalon-ST data bus. This bus enables the transfer of pixel data out of the MegaCore function.
is_eop	Out	dout port Avalon-ST endofpacket signal. This signal is asserted when the MegaCore function is ending a frame.
is_ready	In	dout port Avalon-ST ready signal. The downstream device asserts this signal when it is able to receive data.
is_sop	Out	dout port Avalon-ST startofpacket signal. This signal is asserted when the MegaCore function is starting a new frame.
is_valid	Out	dout port Avalon-ST valid signal. This signal is asserted when the MegaCore function outputs data.
overflow	Out	Clocked video overflow signal. A signal corresponding to the overflow sticky bit of the Status register synchronized to vid_clk. This signal is for information only and no action is required if it is asserted. (1)
refclk_div	Out	A divided down version of vid_clk (refclk). Setting the Refclk Divider register to be the number of samples in a line produces a horizontal reference on this signal that a PLL can use to synchronize its output clock.
sof	Out	Start of frame signal. A change of 0 to 1 indicates the start of the video frame as configured by the SOF registers. Connecting this signal to a Clocked Video Output MegaCore function allows the function to synchronize its output video to this signal.
sof_locked	Out	Start of frame locked signal. When high the sof signal is valid and can be used.
status_update_int	Out	control slave port Avalon-MM interrupt signal. When asserted the status registers of the MegaCore function have been updated and the master must read them to determine what has occurred. (1)
vid_data	In	Clocked video data bus. This bus enables the transfer of video data into the MegaCore function.
vid_datavalid	In	Clocked video data valid signal. Assert this signal when a valid sample of video data is present on vid_data.
vid_f	In	(Separate Synchronization Mode Only.) Clocked video field signal. For interlaced input, this signal distinguishes between field 0 and field 1. For progressive video, you must deassert this signal.
vid_h_sync	In	(Separate Synchronization Mode Only.) Clocked video horizontal synchronization signal. Assert this signal during the horizontal synchronization period of the video stream.
vid_hd_sdn	In	Clocked video color plane format selection signal (in run-time switching of color plane transmission formats mode only). This signal distinguishes between sequential (when low) and parallel (when high) color plane formats.
vid_locked	In	Clocked video locked signal. Assert this signal when a stable video stream is present on the input. Deassert this signal when the video stream is removed.
vid_std	In	Video Standard bus. Can be connected to the rx_std signal of the SDI MegaCore function (or any other interface) to read from the Standard register.

Table 10–9. Clocked Video Input Signals (Part 3 of 3)

Signal	Direction	Description
`vid_v_sync`	In	(Separate Synchronization Mode Only.) Clocked video vertical synchronization signal. Assert this signal during the vertical synchronization period of the video stream.

Note to Table 10–9:

(1) These ports are present only if you turn on **Use control port**.

Control Register Maps

Table 10–10 lists the Clocked Video Input MegaCore function control register map. The width of each register is 16 bits.

Table 10–10. Clocked Video Input Control Register Map (Part 1 of 2)

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit: ■ Setting this bit to 1 causes the Clocked Video Input MegaCore function to start data output on the next video frame boundary. For more information, refer to "Control Port" on page 10–3. Bits 3, 2, and 1 of the `Control` register are the interrupt enables: ■ Setting bit 1 to 1, enables the status update interrupt. ■ Setting bit 2 to 1, enables the stable video interrupt. ■ Setting bit 3 to 1, enables the synchronization outputs (`sof`, `sof_locked`, `refclk_div`).
1	Status	Bit 0 of this register is the `Status` bit: ■ Data is being output by the Clocked Video Input MegaCore function when this bit is asserted. For more information, refer to "Control Port" on page 10–3. Bits 2 and 1 of the `Status` register are not used. Bits 6, 5, 4, and 3 are the resolution valid bits: ■ When bit 3 is asserted, the `SampleCount` register is valid. ■ When bit 4 is asserted, the `F0LineCount` register is valid. ■ When bit 5 is asserted, the `SampleCount` register is valid. ■ When bit 6 is asserted, the `F1LineCount` register is valid. Bit 7 is the interlaced bit: ■ When asserted, the input video stream is interlaced. Bit 8 is the stable bit: ■ When asserted, the input video stream has had a consistent line length for two of the last three lines. Bit 9 is the overflow sticky bit: ■ When asserted, the input FIFO has overflowed. The overflow sticky bit stays asserted until a write of is performed to this bit. Bit 10 is the resolution bit: ■ When asserted, indicates a valid resolution in the sample and line count registers.

Table 10–10. Clocked Video Input Control Register Map (Part 2 of 2)

Address	Register	Description
2	`Interrupt`	Bits 2 and 1 are the interrupt status bits: ■ When bit 1 is asserted, the status update interrupt has triggered. ■ When bit 2 is asserted, the stable video interrupt has triggered. ■ The interrupts stay asserted until a write of 1 is performed to these bits.
3	`Used Words`	The used words level of the input FIFO.
4	`Active Sample Count`	The detected sample count of the video streams excluding blanking.
5	`F0 Active Line Count`	The detected line count of the video streams F0 field excluding blanking.
6	`F1 Active Line Count`	The detected line count of the video streams F1 field excluding blanking.
7	`Total Sample Count`	The detected sample count of the video streams including blanking.
8	`F0 Total Line Count`	The detected line count of the video streams F0 field including blanking.
9	`F1 Total Line Count`	The detected line count of the video streams F1 field including blanking.
10	`Standard`	The contents of the `vid_std` signal.
11	`SOF Sample`	Start of frame sample register. The sample and sub-sample upon which the SOF occurs (and the `sof` signal triggers): ■ Bits 0–1 are the subsample value. ■ Bits 2–15 are the sample value.
12	`SOF Line`	Start of frame line register. The line upon which the SOF occurs measured from the rising edge of the F0 vertical sync.
13	`Refclk Divider`	Number of cycles of `vid_clk` (`refclk`) before `refclk_div` signal triggers.

11. Clocked Video Output MegaCore Function

Core Overview

The Clocked Video Output MegaCore function converts Avalon-ST Video to clocked video formats (such as BT656, BT1120, and DVI). You can configure the Clocked Video Output at run time using an Avalon-MM slave interface.

Functional Description

The Clocked Video Output MegaCore function formats Avalon-ST Video into clocked video by inserting horizontal and vertical blanking and generating horizontal and vertical synchronization information using the Avalon-ST Video control and active picture packets.

No conversion is done to the active picture data, the color plane information remains the same as in the Avalon-ST Video format.

The Clocked Video Output MegaCore function converts data from the flow controlled Avalon-ST Video protocol to clocked video. It also provides clock crossing capabilities to allow video formats running at different frequencies to be output from the system.

In addition, this MegaCore function provides a number of configuration registers that control the format of video leaving the system (blanking period size, synchronization length, and interlaced or progressive mode) and a status interrupt that can be used to determine when the video format changes.

Video Formats

The Clocked Video Output MegaCore function creates the following clocked video formats:

- Video with synchronization information embedded in the data (in BT656 or BT1120 format)
- Video with separate synchronization (h sync, v sync) signals

The Clocked Video Output MegaCore function creates a video frame consisting of horizontal and vertical blanking (containing syncs) and areas of active picture (taken from the Avalon-ST Video input).

The format of the video frame is shown in Figure 11–1 for progressive and Figure 11–2 for interlaced.

Figure 11–1. Progressive Frame Format

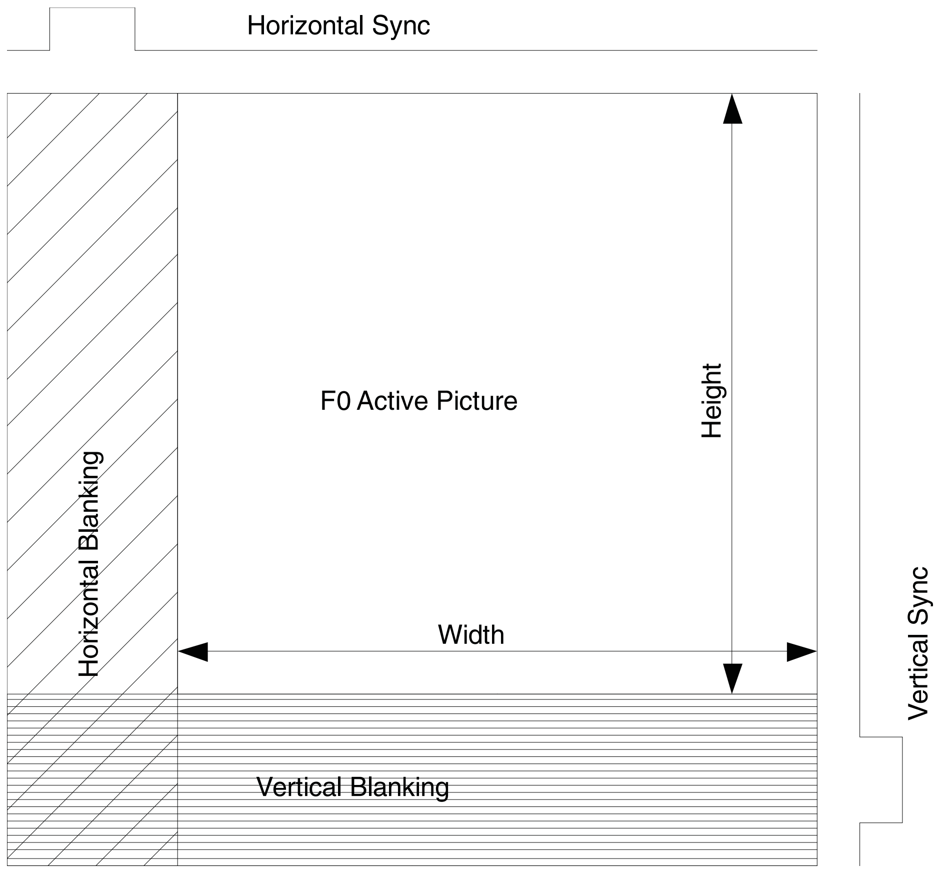
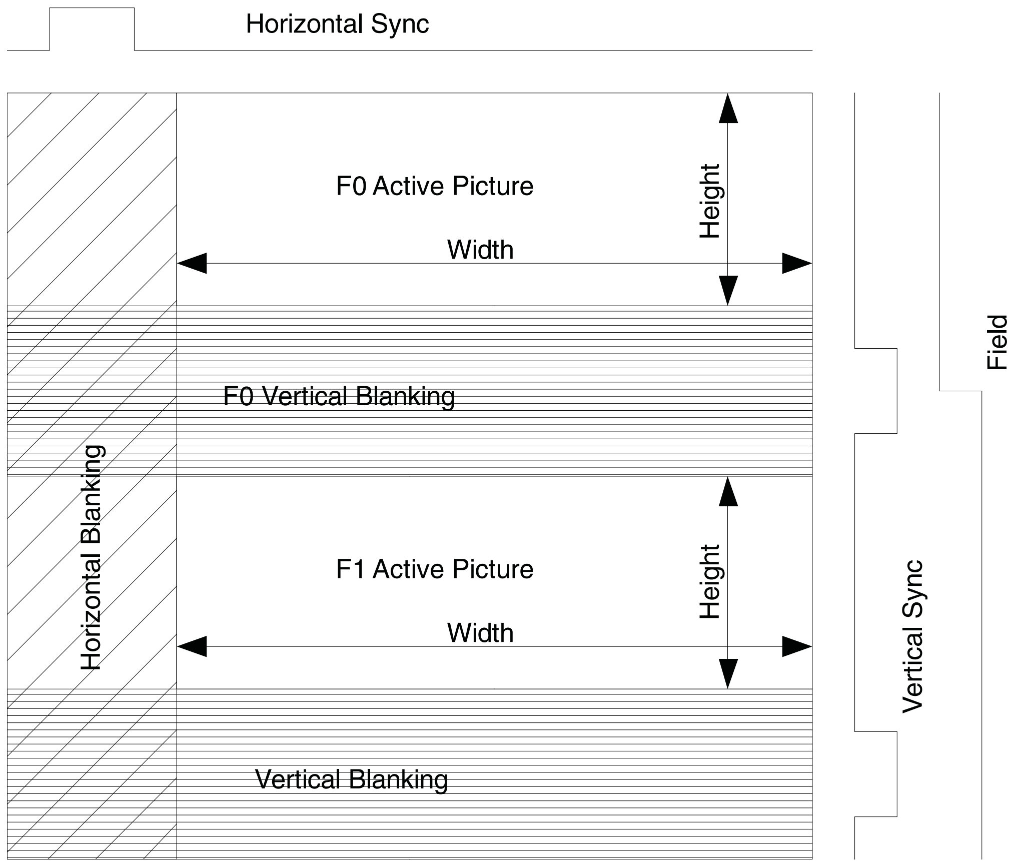


Figure 11–2. Interlaced Frame Format



Embedded Synchronization Format

For the embedded synchronization format, the MegaCore function inserts the horizontal and vertical syncs and field into the data stream during the horizontal blanking period (Table 10–1 on page 10–2).

A sample is output for each clock cycle on the `vid_data` bus.

There are two extra signals that are only used when connecting to the SDI MegaCore function. They are `vid_trs`, which is high during the `3FF` sample of the TRS, and `vid_ln`, which outputs the current SDI line number. These are used by the SDI MegaCore function to insert line numbers and cyclical redundancy checks (CRC) into the SDI stream as specified in the 1.5 Gbps HD SDI and 3 Gbps SDI standards.

The Clocked Video Output MegaCore inserts any ancillary packets (packets with a type of 13 or 0xD) into the output video during the vertical blanking. For information about Avalon-ST Video ancillary data packets, refer to "Ancillary Data Packets" on page 3–10. The Clocked Video Output MegaCore begins inserting the packets on the lines specified in its parameters or mode registers (ModeN Ancillary Line and ModeN F0 Ancillary Line). The Clocked Video Output MegaCore stops inserting the packets at the end of the vertical blanking.

The Clocked Video Input MegaCore function extracts any ancillary packets from the Y channel during the vertical blanking. Ancillary packets are not extracted from the horizontal blanking. The extracted packets are output via the Clocked Video Input's Avalon-ST output with a packet type of 13 (0xD).

Separate Synchronization Format

For the separate synchronization format, the MegaCore function outputs horizontal and vertical syncs and field information via their own signals.

A sample is output for each clock cycle on the `vid_data` bus. The `vid_datavalid` signal is used to indicate when the `vid_data` video output is in an active picture period of the frame.

Table 11–1 lists five extra signals for separate synchronization formats.

Table 11–1. Clocked Video Output Signals for Separate Synchronization Format Video

Signal Name	Description
`vid_h_sync`	1 during the horizontal synchronization period.
`vid_v_sync`	1 during the vertical synchronization period.
`vid_f`	When interlaced data is output, this is a 1 when F1 is being output and a 0 when F0 is being output. During progressive data it is always 0.
`vid_h`	1 during the horizontal blanking period.
`vid_v`	1 during the vertical blanking period.

Control Port

If you turn on **Use control port** in the parameter editor for the Clocked Video Output, it can be controlled using the Avalon-MM slave control port. Initially, the MegaCore function is disabled and does not output any video. However, it still accepts data on the Avalon-ST Video interface for as long as it has space in its input FIFO.

The sequence for starting the output of the MegaCore function is as follows:

1. Write a 1 to `Control` register bit 0.
2. Read `Status` register bit 0. When this is a 1, the function outputs video.

The sequence for stopping the output of the MegaCore function is as follows:

1. Write a 0 to `Control` register bit 0.
2. Read `Status` register bit 0. When this is a 0, the function has stopped video output. This occurs at the end of the next frame or field boundary.

The starting and stopping of the MegaCore function is synchronized to a frame or field boundary.

Video Modes

The video frame is described using the mode registers that are accessed via the Avalon-MM control port. If you turn off **Use control port** in the parameter editor for the Clocked Video Output, then the output video format always has the format specified in the parameter editor.

The MegaCore function can be configured to support between 1 to 14 different modes and each mode has a bank of registers that describe the output frame. When the MegaCore function receives a new control packet on the Avalon-ST Video input, it searches the mode registers for a mode that is valid and has a field width and height that matches the width and height in the control packet. The register `Video Mode Match` shows the selected mode. When found, it restarts the video output with those format settings. If a matching mode is not found, the video output format is unchanged and a restart does not occur.

Figure 11–3 shows how the register values map to the progressive frame format described in "Video Formats" on page 11–1.

Figure 11–3. Progressive Frame Parameters

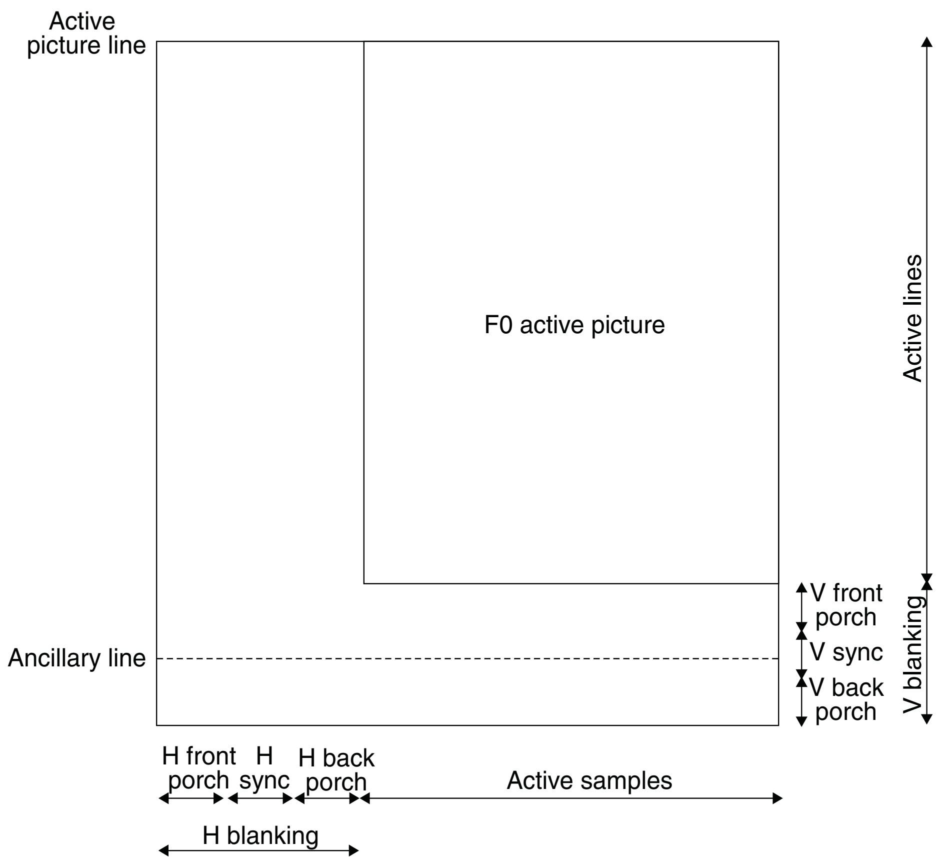


Table 11–2 lists how Figure 11–3 relates to the register map.

Table 11–2. Progressive Frame Parameter Descriptions

Register Name	Parameter	Description
ModeN Control	N/A	The zeroth bit of this register is the Interlaced bit: ■ Set to 0 for progressive. Bit 1 of this register is the sequential output control bit (only if the **Allow output of color planes in sequence** compile-time parameter is enabled). ■ Setting bit 1 to 1, enables sequential output from the Clocked Video Output, such as for NTSC. Setting bit 1 to a 0, enables parallel output from the Clocked Video Output, such as for 1080p.
ModeN Sample Count	Active samples	The width of the active picture region in samples/pixels.
ModeN F0 Line Count	Active lines	The height of the active picture region in lines.
ModeN Horizontal Front Porch	H front porch	(Separate synchronization mode only.) The front porch of the horizontal synchronization (the low period before the synchronization starts).
ModeN Horizontal Sync Length	H sync	(Separate synchronization mode only.) The synchronization length of the horizontal synchronization (the high period of the sync).
ModeN Horizontal Blanking	H blanking	The horizontal blanking period (non active picture portion of a line).
ModeN Vertical Front Porch	V front porch	(Separate synchronization mode only.) The front porch of the vertical synchronization (the low period before the synchronization starts).
ModeN Vertical Sync Length	V sync	(Separate synchronization mode only.) The synchronization length of the vertical synchronization (the high period of the sync).
ModeN Vertical Blanking	V blank	The vertical blanking period (non active picture portion of a frame).
ModeN Active Picture Line	Active picture line	The line number that the active picture starts on. For non SDI output this can be left at 0.
ModeN Valid	N/A	Set to enable the mode after the configuration is complete.
ModeN Ancillary Line	Ancillary line	(Embedded synchronization mode only.) The line to start inserting ancillary packets.

Figure 11–4 shows how the register values map to the interlaced frame format described in "Video Formats" on page 11–1.

Figure 11–4. Interlaced Frame Parameters

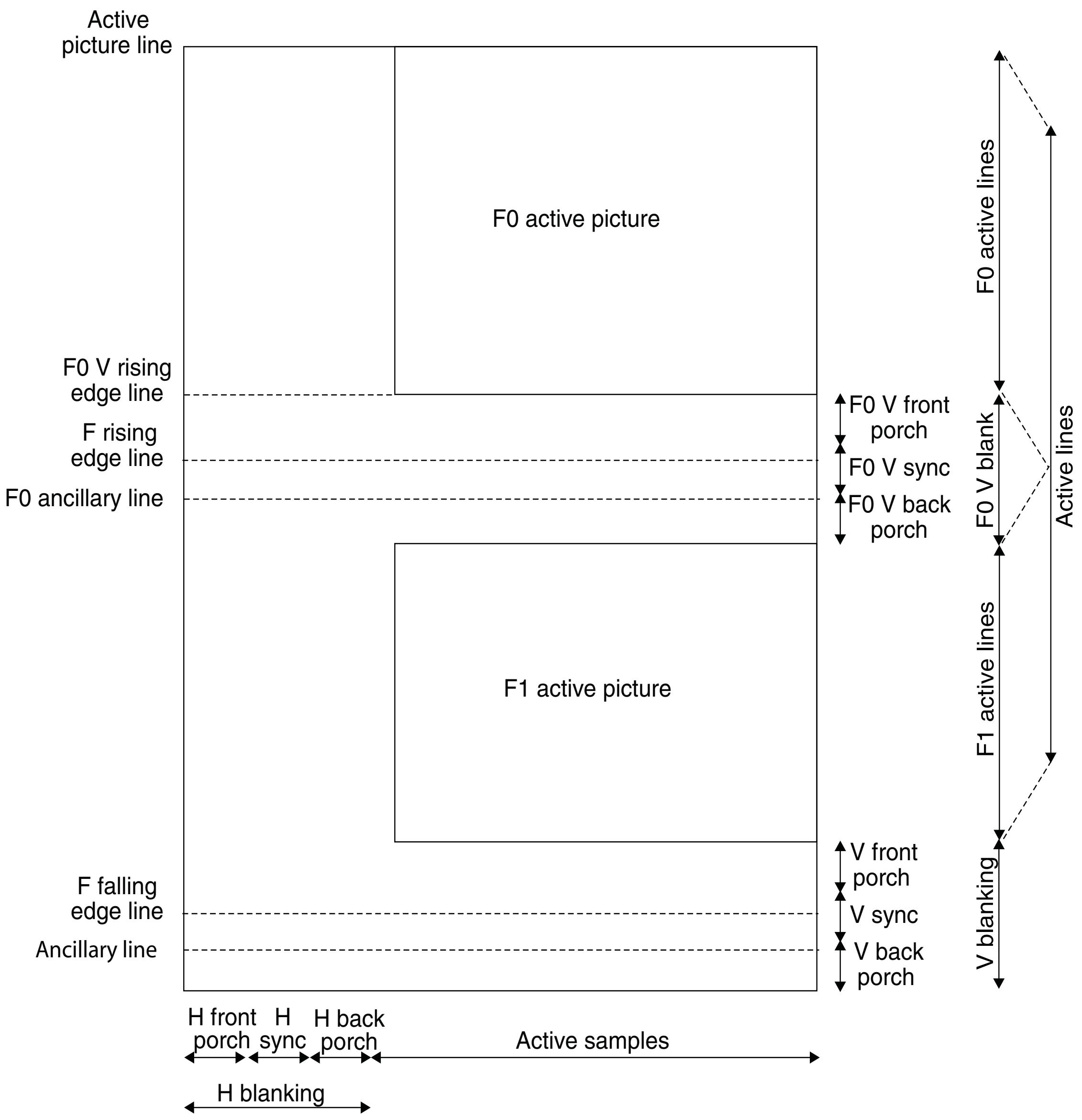


Table 11–3 lists how Figure 11–4 relates to the register map.

Table 11–3. Interlaced Frame Parameter Descriptions

Register Name	Parameter	Description
ModeN Control	N/A	The zeroth bit of this register is the Interlaced bit: ■ Set to 0 for interlaced. ■ Bit 1 of this register is the sequential output control bit (only if the **Allow output of color planes in sequence** compile-time parameter is enabled). ■ Setting bit 1 to 1, enables sequential output from the Clocked Video Output, such as for NTSC. Setting bit 1 to a 0, enables parallel output from the Clocked Video Output, such as for 1080p.
ModeN Sample Count	Active samples	The width of the active picture region in samples/pixels.
ModeN F0 Line Count	F0 active lines	The height of the active picture region for field F0 in lines.
ModeN F1 Line Count	F1 active lines	The height of the active picture region for field F1 in lines.
ModeN Horizontal Front Porch	H front porch	(Separate synchronization mode only.) The front porch of the horizontal synchronization (the low period before the synchronization starts).
ModeN Horizontal Sync Length	H sync	(Separate synchronization mode only.) The synchronization length of the horizontal synchronization (the high period of the sync).
ModeN Horizontal Blanking	H blanking	The horizontal blanking period (non active picture portion of a line).
ModeN Vertical Front Porch	V front porch	(Separate synchronization mode only.) The front porch of the vertical synchronization (the low period before the synchronization starts) for field F1.
ModeN Vertical Sync Length	V sync	(Separate synchronization mode only.) The synchronization length of the vertical synchronization (the high period of the sync) for field F1.
ModeN Vertical Blanking	V blanking	The vertical blanking period (non active picture portion of a frame) for field F1.
ModeNF0 Vertical Front Porch	F0 V front porch	(Separate synchronization mode only.) The front porch of the vertical synchronization (the low period before the synchronization starts) for field F0.
ModeN F0 Vertical Sync Length	F0 V sync	(Separate synchronization mode only.) The synchronization length of the vertical synchronization (the high period of the sync) for field F0.
ModeN F0 Vertical Blanking	F0 V blank	The vertical blanking period (non active picture portion of a frame) for field F0.
ModeN Active Picture Line	active picture line	The line number that the active picture starts on. For non SDI output this can be left at 0.
ModeN F0 Vertical Rising	F0 V rising edge line	The line number that the vertical blanking period for field F0 begins on.
ModeN Field Rising	F rising edge line	The line number that field F1 begins on.
ModeN Field Falling	F falling edge line	The line number that field F0 begins on.
ModeN Valid	N/A	Set to enable the mode after the configuration is complete.
ModeN Ancillary Line	Ancillary line	(Embedded synchronization mode only.) The line to start inserting ancillary packets.
ModeN F0 Ancillary Line	F0 ancillary line	(Embedded synchronization mode only.) The line in field F0 to start inserting ancillary packets.

The mode registers can only be written to if a mode is marked as invalid. For example, the following steps reconfigure mode 1:

1. Write 0 to the `Mode1 Valid` register.
2. Write to the mode 1 configuration registers.
3. Write 1 to the `Mode1 Valid` register. The mode is now valid and can be selected.

A currently-selected mode can be configured in this way without affecting the video output of the MegaCore function.

When searching for a matching mode and there are multiple modes that match the resolution, the function selects the lowest mode. For example, the function selects Mode1 over Mode2 if they both match. To allow the function to select Mode2, invalidate Mode1 by writing a 0 to its mode valid register. Invalidating a mode does not clear its configuration.

Interrupts

The Clocked Video Output MegaCore function outputs a single interrupt line which is the `OR` of the following internal interrupts:

- The status update interrupt— Triggers when the `Video Mode Match` register is updated by a new video mode being selected.
- Locked interrupt—Triggers when the outgoing video SOF is aligned to the incoming SOF.

Both interrupts can be independently enabled using bits [2:1] of the `Control` register. The `ir` values can be read using bits [2:1] of the `Interrupt` register and a write of 1 to either of these bits clears the respective interrupt.

Generator Lock

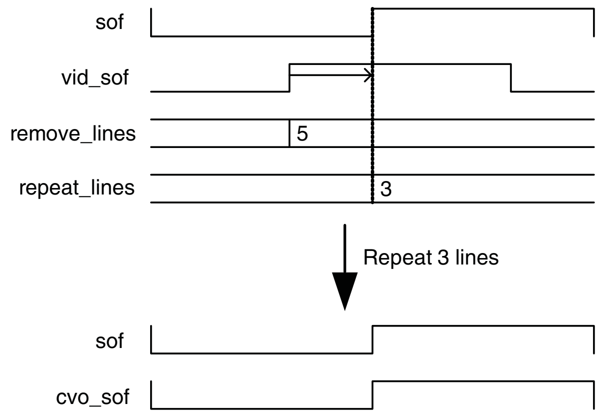
The Clocked Video Output MegaCore function provides some functions to facilitate Genlock. The MegaCore function can be configured to output, via the `vcoclk_div` signal, a divided down version of its `vid_clk` (`vcoclk`) signal aligned to the SOF. By setting the divided down value to be the length in samples of a video line, the `vcoclk_div` signal can be configured to output a horizontal reference. The Genlock functionality is enabled using the Control register. When Genlock functionality is enabled the Clocked Video Output MegaCore does not synchronize itself to the incoming Avalon-ST Video. Altera recommends that you disable Genlock functionality before changing output mode and then only enable it again when the status update interrupt has fired, indicating that the mode change has occurred.

The `vcoclk_div` signal can be compared to the `refclk_div` signal, output by a Clocked Video Input MegaCore function, using a phase frequency detector (PFD) that controls a voltage controlled oscillator (VCXO). By controlling the VCXO, the PFD can align its output clock (`vcoclk`) to the reference clock (`refclk`). By tracking changes in the `refclk_div` signal, the PFD can then ensure that the output clock is locked to the incoming video clk.

The Clocked Video Output MegaCore function can take in the SOF signal from a Clocked Video Input MegaCore function and align its own SOF to this signal. The Clocked Video Output SOF signal can be set to any position within the outgoing video frame. The registers used to configure the SOF signal are measured from the rising edge of the F0 vertical sync. A start of frame is indicated by a rising edge on the SOF signal (0 to 1). Figure 11–2 on page 11–3 shows an example configuration.

Figure 11–5 shows how the Clocked Video Output MegaCore function compares the two SOF signals to determine how far apart they are.

Figure 11–5. Aligning the Output Video to the Incoming SOF

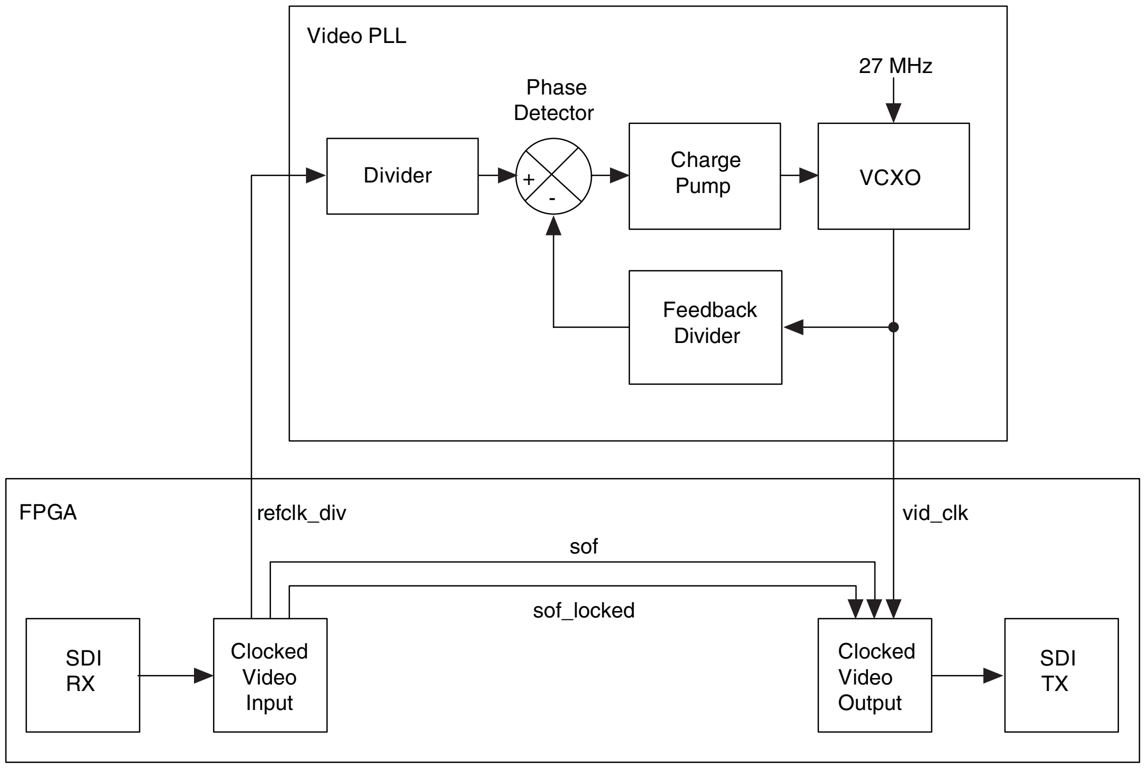


The Clocked Video Output MegaCore function then repeats or removes that number of samples and lines in the output video to align the two SOF signals. If the SOF signals are separated by less than a threshold number of samples (the value of the `Vcoclk Divider` register), the Clocked Video Output does not alter the output video. If your PFD clock tracking has a delay associated with it, Altera recommends that even if the `vcoclk_div` signal is not being used, you must set the `Vcoclk Divider` register to a threshold value (for example, 1). This stops the Clocked Video Output MegaCore function from resyncing every time a delay in clock tracking causes the SOF signals to drift out by a clock cycle.

The current distance between the SOF signals is stored internally and when either the repeat registers or the remove registers read 0 then the locked interrupt triggers.

Figure 11–6 shows an example of how to connect the Clocked Video Input and Clocked Video Output MegaCore functions to a video PLL.

Figure 11–6. Example System Connections



Underflow

Moving between flow controlled Avalon-ST Video and clocked video can cause problems if the flow controlled video does not provide data at a rate fast enough to satisfy the demands of the outgoing clocked video.

The Clocked Video Output MegaCore function contains a FIFO that, when set to a large enough value, can accommodate any "burstiness" in the flow data, as long as the output rate of the downstream Avalon-ST Video components is equal to or higher than that of the outgoing clocked video.

If this is not the case, the FIFO underflows. If underflow occurs, the MegaCore function continues to output video and re-syncs the `startofpacket`, for the next image packet, from the Avalon-ST Video interface with the start of the next frame.

The underflow can be detected by looking at bit 2 of the `Status` register. This bit is sticky and if an underflow occurs, stays at 1 until the bit is cleared by writing a 1 to it. In addition to the underflow bit, the current level of the FIFO can be read from the `Used Words` register.

Timing Constraints

To constrain the Clocked Video Output MegaCore function correctly, add the following file to your Quartus II project:

<*install_dir*>**\ip\clocked_video_output\lib\alt_vip_cvo.sdc.**

When you apply the SDC file, you may see some warning messages in a format as follows:

- Warning: At least one of the filters had some problems and could not be matched.
- Warning: * could not be matched with a keeper.

These warnings are expected, because in certain configurations the Quartus II software optimizes unused registers and they no longer remain in your design.

Active Format Description Inserter

The AFD Inserter is an example of how to write a core to handle ancillary packets. It is available in the following directory:

<*install_dir*>**\ip\altera\clocked_video_output\afd_example**

When the output of the AFD Inserter is connected to the input of the Clocked Video Output MegaCore function, the AFD Inserter inserts an Avalon-ST Video ancillary data packet into the stream after each control packet. The AFD Inserter sets the DID and SDID of the ancillary packet to make it an AFD packet (DID = 0x41, SDID = 0x5). The contents of the ancillary packet are controlled by the AFD Inserter register map.

For more information about the AFD codes, refer to the SMPTE 2016-1-2007 standard.

Table 11–4 lists the AFD Inserter register map.

Table 11–4. AFD Inserter Register Map

Address	Register	Description
0	Control	When bit 0 is 0, the core discards all packets. When bit 0 is 1, the core passes through all non-ancillary packets.
1		Reserved.
2		Reserved.
3	AFD	Bits 0-3 contain the active format description code.
4	AR	Bit 0 contains the aspect ratio code.
5	Bar data flags	Bits 0-3 contain the bar data flags to insert
6	Bar data value 1	Bits 0-15 contain the bar data value 1 to insert
7	Bar data value 2	Bits 0-15 contain the bar data value 2 to insert
8	AFD valid	When bit 0 is 0, an AFD packet is not present for each image packet. When bit 0 is 1, an AFD packet is present for each image packet.

Stall Behavior and Error Recovery

Once its input FIFO is full, the stall behavior of the Clocked Video Output MegaCore function is dictated by the outgoing video. During horizontal and vertical blanking periods it stalls and does not take in any more video data.

Error Recovery

If the Clocked Video Output MegaCore receives an early end of packet it will re-synchronize the outgoing video to the incoming video data on the next start of packet it receives. If the Clocked Video Output MegaCore receives a late start of packet it will re-synchronize the outgoing video data to the incoming video immediately. Note that when Genlock functionality is enabled the Clocked Video Output MegaCore does not re-synchronize to the incoming video.

☞ For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 11–5 lists the approximate latency from the video data input to the video data output for typical usage modes of the Clocked Video Output MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 11–5. Clocked Video Output Latency

Mode	Latency (1)
All modes with Video in and out use the same clock: On	3 cycles (2)

Notes to Table 11–5:

(1) Add 1 cycle if **Allow color planes in sequence input** is turned on.

(2) Minimum latency case when video input and output rates are synchronized.

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 11–6 lists the Clocked Video Output MegaCore function parameters.

Table 11–6. Clocked Video Output Parameter Settings (Part 1 of 2)

Parameter	Value	Description
Select preset to load	**DVI 1080p60**, SDI 1080p60, SDI 1080i60, PAL, NTSC	You can choose from a list of preset conversions or use the other fields in the dialog box to set up custom parameter values. If you click **Load values into controls** the dialog box is initialized with values for the selected preset conversion.
Image width / Active pixels	32–65,536, Default = **1,920**	Specify the image width by choosing the number of active pixels.
Image height / Active lines	32–65,536, Default = **1,080**	Specify the image height by choosing the number of active lines.
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color planes	1–4, Default = **3**	Choose the number of color planes.
Color plane transmission format	Sequence, **Parallel**	Choose whether the color planes are transmitted in sequence or in parallel.
Allow output of color planes in sequence	On or **Off**	Choose whether run-time switching is allowed between sequential formats, such as NTSC, and parallel color plane transmission formats, such as 1080p. The format is controlled by the ModeXControl registers. See the Avalon-ST Video Protocol section under Interfaces for a description of the difference between sequential and parallel color plane transmission formats.
Interlaced video	On or **Off**	Turn on if you want to use interlaced video. If on, you can set the additional Interlaced and Field 0 Parameters.
Sync signals	Embedded in video, **On separate wires**	Choose whether the synchronization signal is embedded in the video stream or provided on a separate wire. If you choose **Embedded in video**, you can set the active picture line, horizontal blanking, and vertical blanking values. If you choose **On separate wires**, you can set horizontal and vertical values for sync, front porch, and back porch.
Active picture line	0–65,536, Default = **0**	Choose the start of active picture line for Frame.
Frame / Field 1: Ancillary packet insertion line	0–65,536, Default = **0**	Choose the line where ancillary packet insertion starts.
Frame / Field 1: Horizontal blanking	0–65,536, Default = **0**	Choose the size of the horizontal blanking period in pixels for Frame/Field 1.
Frame / Field 1: Vertical blanking	0–65,536, Default = **0**	Choose the size of the vertical blanking period in pixels for Frame/Field 1.
Frame / Field 1: Horizontal sync	1–65,536, Default = **60**	Choose the size of the horizontal synchronization period in pixels for Frame/Field 1.
Frame / Field 1: Horizontal front porch	1–65,536, Default = **20**	Choose the size of the horizontal front porch period in pixels for Frame/Field 1.
Frame / Field 1: Horizontal back porch	1–65,536, Default = **192**	Choose the size of the horizontal back porch period in pixels for Frame/Field 1.
Frame / Field 1: Vertical sync	0–65,536, Default = **5**	Choose the number of lines in the vertical synchronization period for Frame/Field 1.

Table 11–6. Clocked Video Output Parameter Settings (Part 2 of 2)

Parameter	Value	Description
Frame / Field 1: Vertical front porch	0–65,536, Default = **4**	Choose the number of lines in the vertical front porch period for Frame/Field 1.
Frame / Field 1: Vertical back porch	0–65,536, Default = **36**	Choose the number of lines in the vertical back porch period for Frame/Field 1.
Interlaced and Field 0: F rising edge line	0–65,536, Default = **0**	Choose the line when the rising edge of the field bit occurs for Interlaced and Field 0.
Interlaced and Field 0: F falling edge line	0–65,536, Default = **18**	Choose the line when the rising edge of the vertical blanking bit for Field 0 occurs for Interlaced and Field 0.
Interlaced and Field 0: Vertical blanking rising edge line	0–65,536, Default = **0**	Choose the line when the vertical blanking rising edge occurs for Interlaced and Field 0.
Interlaced and Field 0: Ancillary packet insertion line	0–65,536, Default = **0**	Choose the line where ancillary packet insertion starts.
Interlaced and Field 0: Vertical blanking	0–65,536, Default = **0**	Choose the number of lines in the vertical front porch period for Interlaced and Field 0.
Interlaced and Field 0: Vertical sync	0–65,536, Default = **0**	Choose the number of lines in the vertical back porch period for Interlaced and Field 0.
Interlaced and Field 0: Vertical front porch	0–65,536, Default = **0**	Choose the number of lines in the vertical front porch period for Interlaced and Field 0.
Interlaced and Field 0: Vertical back porch	0–65,536, Default = **0**	Choose the number of lines in the vertical back porch period for Interlaced and Field 0.
Pixel FIFO size	32–(memory limit), Default = **1,920**	Choose the required FIFO depth in pixels (limited by the available on-chip memory).
FIFO level at which to start output	0–(memory limit), Default = **0**	Choose the fill level that the FIFO must have reached before the output video starts.
Video in and out use the same clock	On or **Off**	Turn on if you want to use the same signal for the input and output video image stream clocks.
Use control port	On or **Off**	Turn on to use the optional Avalon-MM control port.
Run-time configurable video modes (1)	1–14, Default = **1**	Choose the number of run-time configurable video output modes that are required when you are using the Avalon-MM control port.
Accept synchronization outputs	**No**, Yes	Specifies whether the synchronization outputs are used: ■ No - Not used ■ Yes - Synchronization outputs, from the Clocked Video Input MegaCore function, (`sof`, `sof_locked`) are used
Width of “vid_std”	**0**–16	Specifies the width of the `vid_std` bus.

Note to Table 11–6:

(1) This parameter is available only when you turn on **Use control port**.

Signals

Table 11–7 lists the input and output signals for the Clocked Video Output MegaCore function.

Table 11–7. Clocked Video Output Signals (Part 1 of 2)

Signal	Direction	Description
`rst`	In	The MegaCore function asynchronously resets when you assert `rst`. You must deassert `rst` synchronously to the rising edge of the `is_clk` signal. When the video in and video out do not use the same clock, this signal is resynchronized to the output clock to be used in the output clock domain.
`vid_clk`	In	Clocked video clock. All the video input signals are synchronous to this clock.
`av_address`	In	`control` slave port Avalon-MM `address` bus. Specifies a word offset into the slave address space. (1)
`av_read`	In	`control` slave port Avalon-MM `read` signal. When you assert this signal, the `control` port drives new data onto the read data bus. (1)
`av_readdata`	Out	`control` slave port Avalon-MM `readdata` bus. These output lines are used for read transfers. (1)
`av_waitrequest`	Out	`control` slave port Avalon-MM `waitrequest` bus. When this signal is asserted, the `control` port cannot accept new transactions. (1)
`av_write`	In	`control` slave port Avalon-MM `write` signal. When you assert this signal, the `control` port accepts new data from the write data bus. (1)
`av_writedata`	In	`control` slave port Avalon-MM `writedata` bus. These input lines are used for write transfers. (1)
`is_clk`	In	Clock signal for Avalon-ST ports `dout` and `control`. The MegaCore function operates on the rising edge of the `is_clk` signal.
`is_data`	In	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`is_eop`	In	`dout` port Avalon-ST `endofpacket` signal. Assert this signal when the downstream device is ending a frame.
`is_ready`	Out	`dout` port Avalon-ST `ready` signal. This signal is asserted when the MegaCore function is able to receive data.
`is_sop`	In	`dout` port Avalon-ST `startofpacket` signal. Assert this signal when the downstream device is starting a new frame.
`is_valid`	In	`dout` port Avalon-ST `valid` signal. Assert this signal when the downstream device outputs data.
`sof`	In	Start of frame signal. A rising edge (0 to 1) indicates the start of the video frame as configured by the SOF registers. Connecting this signal to a Clocked Video Input MegaCore function allows the output video to be synchronized to this signal.
`sof_locked`	Out	Start of frame locked signal. When high the `sof` signal is valid and can be used.
`status_update_int`	Out	`control` slave port Avalon-MM interrupt signal. When asserted the status registers of the MegaCore function have been updated and the master must read them to determine what has occurred. (1)
`underflow`	Out	Clocked video underflow signal. A signal corresponding to the underflow sticky bit of the `Status` register synchronized to `vid_clk`. This signal is for information only and no action is required if it is asserted. (1)

Table 11–7. Clocked Video Output Signals (Part 2 of 2)

Signal	Direction	Description
`vcoclk_div`	Out	A divided down version of `vid_clk (vcoclk)`. Setting the `Vcoclk Divider` register to be the number of samples in a line produces a horizontal reference on this signal that a PLL can use to synchronize its output clock.
`vid_data`	Out	Clocked video data bus. This bus transfers video data into the MegaCore function.
`vid_datavalid`	Out	(Separate Synchronization mode Only.) Clocked video data valid signal. This signal is asserted when an active picture sample of video data is present on `vid_data`.
`vid_f`	Out	(Separate Synchronization Mode Only.) Clocked video field signal. For interlaced input, this signal distinguishes between field 0 and field 1. For progressive video, this signal is unused.
`vid_h`	Out	(Separate Synchronization Mode Only.) Clocked video horizontal blanking signal. This signal is asserted during the horizontal blanking period of the video stream.
`vid_h_sync`	Out	(Separate Synchronization Mode Only.) Clocked video horizontal synchronization signal. This signal is asserted during the horizontal synchronization period of the video stream.
`vid_ln`	Out	(Embedded Synchronization Mode Only.) Clocked video line number signal. Used with the SDI MegaCore function to indicate the current line number when the `vid_trs` signal is asserted.
`vid_mode_change`	Out	Clocked video mode change signal. This signal is asserted on the cycle before a mode change occurs.
`vid_sof`	Out	Start of frame signal. A rising edge (0 to 1) indicates the start of the video frame as configured by the SOF registers.
`vid_sof_locked`	Out	Start of frame locked signal. When high the `vid_sof` signal is valid and can be used.
`vid_std`	Out	Video standard bus. Can be connected to the `tx_std` signal of the SDI MegaCore function (or any other interface) to set the `Standard` register.
`vid_trs`	Out	(Embedded Synchronization Mode Only.) Clocked video time reference signal (TRS) signal. Used with the SDI MegaCore function to indicate a TRS, when asserted.
`vid_v`	Out	(Separate Synchronization Mode Only.) Clocked video vertical blanking signal. This signal is asserted during the vertical blanking period of the video stream.
`vid_v_sync`	Out	(Separate Synchronization Mode Only.) Clocked video vertical synchronization signal. This signal is asserted during the vertical synchronization period of the video stream.

Note to Table 11–7:

(1) These ports are present only if you turn on **Use control port**.

Control Register Maps

Table 11–8 lists the Clocked Video Output MegaCore function control register map. The width of each register is 16 bits.

Table 11–8. Clocked Video Output Control Register Map (Part 1 of 2)

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit: ■ Setting this bit to 1 causes the Clocked Video Output MegaCore function to start video data output. Refer to "Control Port" on page 11–4 for full details. Bits 3, 2, and 1 of the `Control` register are the interrupt enables: ■ Setting bit 1 to 1, enables the status update interrupt. ■ Setting bit 2 to 1, enables the locked interrupt. ■ Setting bit 3 to 1, enables the synchronization outputs (`vid_sof`, `vid_sof_locked`, `vcoclk_div`). ■ When bit 3 is set to 1, setting bit 4 to 1, enables frame locking. The Clock Video Output attempts to align its vid_sof signal to the sof signal from the Clocked Video Input MegaCore function.
1	`Status`	Bit 0 of this register is the `Status` bit: ■ Data is being output by the Clocked Video Output MegaCore function when this bit is asserted. Refer to "Control Port" on page 11–4 for full details. Bit 1 of the `Status` register is unused. Bit 2 is the underflow sticky bit: ■ When bit 2 is asserted, the output FIFO has underflowed. The underflow sticky bit stays asserted until a 1 is written to this bit. Bit 3 is the frame locked bit. ■ When bit 3 is asserted, the Clocked Video Output has aligned its start of frame to the incoming sof signal.
2	`Interrupt`	Bits 2 and 1 are the interrupt status bits: ■ When bit 1 is asserted, the status update interrupt has triggered. ■ When bit 2 is asserted, the locked interrupt has triggered. ■ The interrupts stay asserted until a write of 1 is performed to these bits.
3	`Used Words`	The used words level of the output FIFO.
4	`Video Mode Match`	One-hot register that indicates the video mode that is selected.
5	`ModeX Control`	Video Mode 1 Control. Bit 0 of this register is the Interlaced bit: ■ Set to 1 for interlaced. Set to a 0 for progressive. Bit 1 of this register is the sequential output control bit (only if the **Allow output of color planes in sequence** compile-time parameter is enabled). ■ Setting bit 1 to 1, enables sequential output from the Clocked Video Output e.g. for NTSC. Setting bit 1 to a 0, enables parallel output from the Clocked Video Output e.g. for 1080p.
6	`Mode1 Sample Count`	Video mode 1 sample count. Specifies the active picture width of the field.
7	`Mode1 F0 Line Count`	Video mode 1 field 0/progressive line count. Specifies the active picture height of the field.
8	`Mode1 F1 Line Count`	Video mode 1 field 1 line count (interlaced video only). Specifies the active picture height of the field.

Table 11–8. Clocked Video Output Control Register Map (Part 2 of 2)

Address	Register	Description
9	`Mode1 Horizontal Front Porch`	Video mode 1 horizontal front porch. Specifies the length of the horizontal front porch in samples.
10	`Mode1 Horizontal Sync Length`	Video mode 1 horizontal synchronization length. Specifies the length of the horizontal synchronization length in samples.
11	`Mode1 Horizontal Blanking`	Video mode 1 horizontal blanking period. Specifies the length of the horizontal blanking period in samples.
12	`Mode1 Vertical Front Porch`	Video mode 1 vertical front porch. Specifies the length of the vertical front porch in lines.
13	`Mode1 Vertical Sync Length`	Video mode 1 vertical synchronization length. Specifies the length of the vertical synchronization length in lines.
14	`Mode1 Vertical Blanking`	Video mode 1 vertical blanking period. Specifies the length of the vertical blanking period in lines.
15	`Mode1 F0 Vertical Front Porch`	Video mode 1 field 0 vertical front porch (interlaced video only). Specifies the length of the vertical front porch in lines.
16	`Mode1 F0 Vertical Sync Length`	Video mode 1 field 0 vertical synchronization length (interlaced video only). Specifies the length of the vertical synchronization length in lines.
17	`Mode1 F0 Vertical Blanking`	Video mode 1 field 0 vertical blanking period (interlaced video only). Specifies the length of the vertical blanking period in lines.
18	`Mode1 Active Picture Line`	Video mode 1 active picture line. Specifies the line number given to the first line of active picture.
19	`Mode1 F0 Vertical Rising`	Video mode 1 field 0 vertical blanking rising edge. Specifies the line number given to the start of field 0's vertical blanking.
20	`Mode1 Field Rising`	Video mode 1 field rising edge. Specifies the line number given to the end of Field 0 and the start of Field 1.
21	`Mode1 Field Falling`	Video mode 1 field falling edge. Specifies the line number given to the end of Field 0 and the start of Field 1.
22	`Mode1 Standard`	The value output on the `vid_std` signal.
23	`Mode1 SOF Sample`	Start of frame sample register. The sample and subsample upon which the SOF occurs (and the `vid_sof` signal triggers): ■ Bits 0–1 are the subsample value. ■ Bits 2–15 are the sample value.
24	`Mode1 SOF Line`	SOF line register. The line upon which the SOF occurs measured from the rising edge of the F0 vertical sync.
25	`Mode1 Vcoclk Divider`	Number of cycles of `vid_clk` (`vcoclk`) before `vcoclk_div` signal triggers.
26	`Mode1 Ancillary Line`	The line to start inserting ancillary data packets.
27	`Mode1 F0 Ancillary Line`	The line in field F0 to start inserting ancillary data packets.
28	`Mode1 Valid`	Video mode 1 valid. Set to indicate that this mode is valid and can be used for video output.
29	`Mode2 Control`	...
30	... *(1)*	...

Note to Table 11–8:

(1) The rows in the table are repeated in ascending order for each video mode. All of the Mode*N* registers are write only.

12. Color Plane Sequencer MegaCore Function

Core Overview

The Color Plane Sequencer MegaCore function changes how color plane samples are transmitted across the Avalon-ST interface.

You can configure the channel order in sequence or in parallel. In addition to reordering color plane samples, the Color Plane Sequencer splits and joins video streams, giving control over the routing of color plane samples.

Functional Description

The Color Plane Sequencer MegaCore function rearranges the color pattern used to transmit Avalon-ST Video data packets over an Avalon-ST connection (stream). The Color Plane Sequencer can also split or duplicate a single Avalon-ST Video stream into two or, conversely, combine two input streams into a single stream.

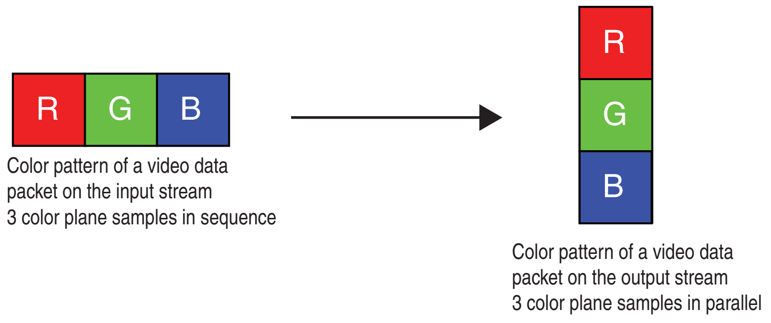
A color pattern is a matrix that defines a repeating pattern of color samples. For full details of the Avalon-ST Video protocol, refer to "Avalon-ST Video Protocol" on page 3–2.

Rearranging Color Patterns

The Color Plane Sequencer can rearrange the color pattern of a video data packet in any valid combination of channels in sequence and parallel. The Color Plane Sequencer can also drop color planes. Avalon-ST Video packets of types other than video data packets are forwarded unchanged.

Figure 12–1 shows an example that rearranges the color pattern of a video data packet which transmits color planes in sequence, to a color pattern that transmits color planes in parallel.

Figure 12–1. Example of Rearranging Color Patterns


R
G
B
Color pattern of a video data packet on the input stream
3 color plane samples in sequence
R
G
B
Color pattern of a video data packet on the output stream
3 color plane samples in parallel

When the color pattern of a video data packet changes from the input to the output side of a block, the color sequencer adds padding to the end of non-video data packets with extra data. Altera recommends that when you define a packet type where the length is variable and meaningful, you send the length at the start of the packet.

Combining Color Patterns

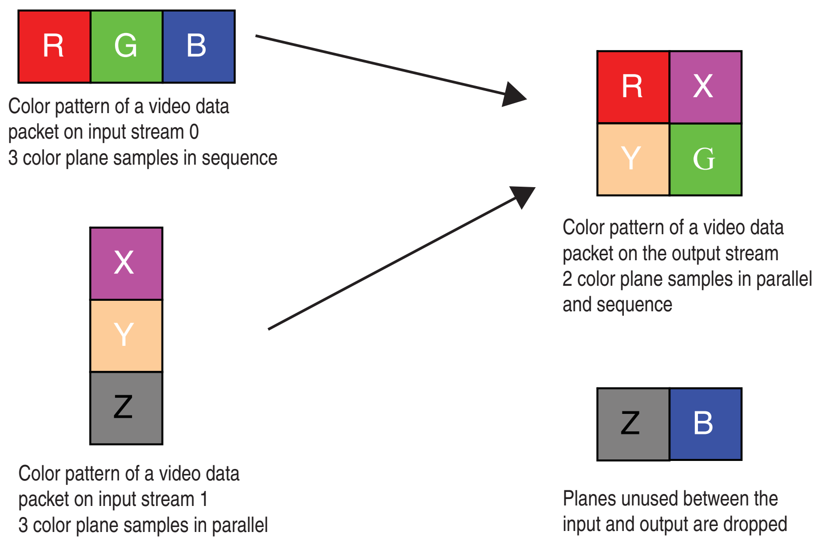
The Color Plane Sequencer also allows the combination of two Avalon-ST Video streams into a single stream. In this mode of operation, two input color patterns (one for each input stream) are combined and arranged to the output stream color pattern in a user defined way, so long as it contains a valid combination of channels in sequence and parallel.

In addition to this combination and arrangement, color planes can also be dropped. Avalon-ST Video packets other than video data packets can be forwarded to the single output stream with the following options:

- Packets from input stream 0 (port `din0`) and input stream 1 (port `din1`) forwarded, input stream 0 packets being transmitted last. (The last control packet received is the one an Avalon-ST Video compliant MegaCore function uses.)
- Packets from input stream 0 forwarded, packets from input stream 1 dropped.
- Packets from input stream 1 forwarded, packets from input stream 0 dropped.

Figure 12–2 shows an example of combining and rearranging two color patterns.

Figure 12–2. Example of Combining Color Patterns


R
G
B
Color pattern of a video data packet on input stream 0
3 color plane samples in sequence
R
X
Y
G
Color pattern of a video data packet on the output stream
2 color plane samples in parallel and sequence
X
Y
Z
Color pattern of a video data packet on input stream 1
3 color plane samples in parallel
Z
B
Planes unused between the input and output are dropped

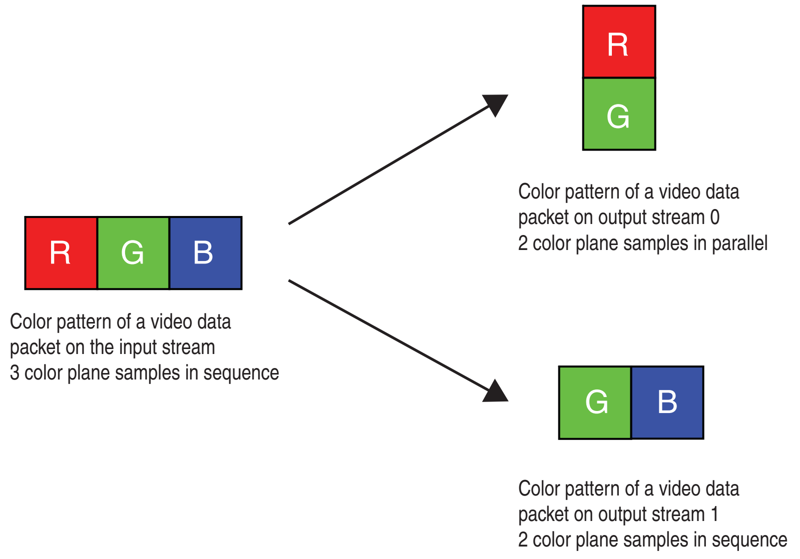
Splitting/Duplicating

The Color Plane Sequencer also allows the splitting of a single Avalon-ST Video input stream into two Avalon-ST Video output streams. In this mode of operation, the color patterns of video data packets on the output streams can be arranged in a user defined way using any of the color planes of the input color pattern.

The color planes of the input color pattern are available for use on either, both, or neither of the outputs. This allows for splitting of video data packets, duplication of video data packets, or a mix of splitting and duplication. The output color patterns are independent of each other, so the arrangement of one output stream's color pattern places no limitation on the arrangement of the other output stream's color pattern.

Avalon-ST Video packets other than video data packets are duplicated to both outputs. Figure 12–3 shows an example of partially splitting and duplicating an input color pattern.

Figure 12–3. Example of Splitting and Duplicating Color Patterns


R
G
B
Color pattern of a video data packet on the input stream
3 color plane samples in sequence
R
G
Color pattern of a video data packet on output stream 0
2 color plane samples in parallel
G
B
Color pattern of a video data packet on output stream 1
2 color plane samples in sequence

CAUTION A deadlock may happen when the sequencer splits, processes independently and then joins back the color planes, or when the sequencer splits the color planes in front of another Video IP core. To avoid this issue, add small FIFO buffers at the output of the Color Plane Sequencer that are configured as splitters.

Subsampled Data

In addition to fully sampled color patterns, the Color Plane Sequencer supports 4:2:2 subsampled data. To facilitate this support, you can configure the Color Plane Sequencer with two color patterns in sequence, so that subsampled planes can be specified individually.

When splitting subsampled planes from fully-sampled planes, the Avalon-ST Video control packet for the subsampled video data packet can have its width value halved, so that the subsampled planes can be processed by other MegaCore functions as if fully sampled. This halving can be applied to control packets on port `dout0` and port `dout1`, or control packets on port `dout0` only.

Avalon-ST Video Protocol Parameters

The only stream requirement imposed is that when two streams are being combined, the video data packets must contain the same total number of pixels, and to make a valid image, the packets must have the same dimensions. The Color Plane Sequencer can process streams of pixel data of the types listed in Table 12–1.

Table 12–1. Color Plane Sequencer Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Read from control packets at run time.
Frame Height	Read from control packets at run time.
Interlaced / Progressive	Either.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	The color pattern you select in the parameter editor.

Stall Behavior and Error Recovery

The Color Plane Sequencer MegaCore function stalls for approximately 10 cycles after processing each line of a video frame. Between frames the MegaCore function stalls for approximately 30 cycles.

Error Recovery

The Color Plane Sequencer MegaCore function processes video packets per line until an `endofpacket` signal is received on `din0`. (The line width is taken from the control packets on `din0`.) When an `endofpacket` signal is received on either `din0` or `din1` the Color Plane Sequencer ceases output. For the number of cycles left to finish the line, the MegaCore function continues to drain the inputs that have not indicated end-of-packet. The MegaCore function drains `din0` until it receives an `endofpacket` signal on this port (unless it has already indicated end-of-packet), and stalls for up to one line after this `endofpacket` signal. The MegaCore function then signals end-of-packet on its outputs and continue to drain its inputs that have not indicated end-of-packet.

☞ For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 12–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the Color Plane Sequencer MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 12–2. Color Plane Sequencer Latency

Mode	Latency
All modes	*O* (cycles)

The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 12–3 lists the Color Plane Sequencer MegaCore function parameters.

Table 12–3. Color Plane Sequencer Parameter Settings

Parameter	Value	Description
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Two pixels per port *(1)*	On or **Off**	Turn on to enable two pixels on each port.
Color planes in parallel (din0)	1–**3**	Choose the number of color planes in parallel for input port `din0`.
Color planes in sequence (din0)	**1**–4	Choose the number of color planes in sequence for input port `din0`.
Port enabled (din1)	On or **Off**	Turn on to enable input port `din0`.
Color planes in parallel (din1)	1–**3**	Choose the number of color planes in parallel for input port `din1`.
Color planes in sequence (din1)	**1**–4	Choose the number of color planes in sequence for input port `din1`.
Port enabled (dout0)	**On** or Off	Turn on to enable output port `dout0`.
Source non-image packets from port (dout0)	**din0**, din1, din0 and din1	Choose the source port(s) that are enabled for non-image packets for output port `dout0`.
Halve control packet width (dout0) *(2) (3)*	On or **Off**	Turn on to halve the Avalon-ST Video control packet width for output port `dout0`.
Color planes in parallel (dout0)	1–**3**	Choose the number of color planes in parallel for output port `dout0`.
Color planes in sequence (dout0)	**1**–4	Choose the number of color planes in sequence for output port `dout0`.
Port enabled (dout1)	**On** or Off	Turn on to enable output port `dout1`.
Source non-image packets from port (dout1)	**din0**, din1, din0 and din1	Choose the source port used for non-image packets for output port `dout1`.
Halve control packet width (dout1)	On or **Off**	Turn on to halve the Avalon-ST Video control packet width for output port `dout1`. *(1)*
Color planes in parallel (dout1)	1–**3**	Choose the number of color planes in parallel for output port `dout1`.
Color planes in sequence (dout1)	**1**–4	Choose the number of color planes in sequence for output port `dout1`.

Notes to Table 12–3:

(1) Turn on when treating Cb and Cr separately because two pixels worth of data is required. Alternatively, you can turn this parameter off and use channel names C, Y instead of Cb, Y, Cr, Y.

(2) This option can be useful if you want to split a subsampled color plane from a fully sampled color plane. The subsampled color plane can then be processed by other functions as if fully sampled.

(3) Turn on when stream contains two subsampled channels. For other MegaCore functions to be able to treat these channels as two fully sampled channels in sequence, the control packet width must be halved.

Signals

Table 12–4 lists the input and output signals for the Color Plane Sequencer MegaCore function.

Table 12–4. Color Plane Sequencer Signals

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`dinN_data`	In	`dinN` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`dinN_endofpacket`	In	`dinN` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dinN_ready`	Out	`dinN` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`dinN_startofpacket`	In	`dinN` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dinN_valid`	In	`dinN` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`doutN_data`	Out	`doutN` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`doutN_endofpacket`	Out	`doutN` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`doutN_ready`	In	`doutN` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`doutN_startofpacket`	Out	`doutN` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`doutN_valid`	Out	`doutN` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

13. Color Space Converter MegaCore Function

Core Overview

The Color Space Converter MegaCore function transforms video data between color spaces. The color spaces allow you to specify colors using three coordinate values. The Color Space Converter supports a number of predefined conversions between standard color spaces, and allows the entry of custom coefficients to translate between any two three-valued color spaces. You can configure the Color Space Converter to change conversion values at run time using an Avalon-MM slave interface.

Functional Description

The Color Space Converter MegaCore function provides a flexible and efficient means to convert image data from one color space to another.

A color space is a method for precisely specifying the display of color using a three-dimensional coordinate system. Different color spaces are best for different devices, such as R'G'B' (red-green-blue) for computer monitors or Y'CbCr (luminance-chrominance) for digital television.

Color space conversion is often necessary when transferring data between devices that use different color space models. For example, to transfer a television image to a computer monitor, you are required to convert the image from the Y'CbCr color space to the R'G'B' color space. Conversely, transferring an image from a computer display to a television may require a transformation from the R'G'B' color space to Y'CbCr.

Different conversions may be required for standard definition television (SDTV) and high definition television (HDTV). You may also want to convert to or from the Y'IQ (luminance-color) color model for National Television System Committee (NTSC) systems or the Y'UV (luminance-bandwidth-chrominance) color model for Phase Alternation Line (PAL) systems.

Input and Output Data Types

The Color Space Converter MegaCore function inputs and outputs support signed or unsigned data and 4 to 20 bits per pixel per color plane. Minimum and maximum guard bands are also supported. The guard bands specify ranges of values that must never be received by, or transmitted from the MegaCore function. Using output guard bands allows the output to be constrained, such that it does not enter the guard bands.

Color Space Conversion

Conversions between color spaces are achieved by providing an array of nine coefficients and three summands that relate the color spaces. These can be set at compile time, or at run time using the Avalon-MM slave interface.

Given a set of nine coefficients [$A0$, $A1$, $A2$, $B0$, $B1$, $B2$, $C0$, $C1$, $C2$] and a set of three summands [$S0$, $S1$, $S2$], the output values on channels 0, 1, and 2 (denoted *dout_0*, *dout_1*, and *dout_2*) are calculated as follows:

$$dout\_0 = (A0 \times din\_0) + (B0 \times din\_1) + (C0 \times din\_2) + S0$$
$$dout\_1 = (A1 \times din\_0) + (B1 \times din\_1) + (C1 \times din\_2) + S1$$
$$dout\_2 = (A2 \times din\_0) + (B2 \times din\_1) + (C2 \times din\_2) + S2$$

where *din_0*, *din_1*, and *din_2* are inputs read from channels 0, 1, and 2 respectively.

User-specified custom constants and the following predefined conversions are supported:

- Computer B′G′R′ to CbCrY′: SDTV
- CbCrY′: SDTV to Computer B′G′R′
- Computer B′G′R′ to CbCrY′: HDTV
- CbCrY′: HDTV to Computer B′G′R′
- Studio B′G′R′ to CbCrY′: SDTV
- CbCrY′: SDTV to Studio B′G′R′
- Studio B′G′R′ to CbCrY′: HDTV
- CbCrY′: HDTV to Studio B′G′R′
- IQY' to Computer B'G'R'
- Computer B'G'R' to IQY'
- UVY' to Computer B'G'R'
- Computer B'G'R' to UVY'

The values are assigned in the order indicated by the conversion name. For example, if you select Computer B′G′R′ to CbCrY′: SDTV, *din_0* = B′, *din_1* = G′, *din_2* = R′, *dout_0* = Cb′, *dout_1* = Cr, and *dout_2* = Y′.

If the channels are in sequence, *din_0* is first, then *din_1*, and *din_2*. If the channels are in parallel, *din_0* occupies the least significant bits of the word, *din_1* the middle bits and *din_2* the most significant bits. For example, if there are 8 bits per sample and one of the predefined conversions inputs B′G′R′, *din_0* carries B′ in bits 0–7, *din_1* carries G′ in bits 8–15, and *din_2* carries R′ in bits 16–23.

Predefined conversions only support unsigned input and output data. If signed input or output data is selected, the predefined conversion produces incorrect results. When using a predefined conversion, the precision of the constants must still be defined. Predefined conversions are based on the input bits per pixel per color plane. If using different input and output bits per pixel per color plane, you must scale the results by the correct number of binary places to compensate.

Constant Precision

The Color Space Converter MegaCore function requires fixed point types to be defined for the constant coefficients and constant summands. The user entered constants (in the white cells of the matrix in the parameter editor) are rounded to fit in the chosen fixed point type (these are shown in the purple cells of the matrix).

Calculation Precision

The Color Space Converter MegaCore function does not lose calculation precision during the conversion. The calculation and result data types are derived from the range of the input data type, the fixed point types of the constants, and the values of the constants. If scaling is selected, the result data type is scaled up appropriately such that precision is not lost.

Result of Output Data Type Conversion

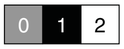
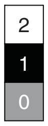
After the calculation, the fixed point type of the results must be converted to the integer data type of the output. This conversion is performed in four stages, in the following order:

1. **Result Scaling**. You can choose to scale up the results, increasing their range. This is useful to quickly increase the color depth of the output. The available options are a shift of the binary point right –16 to +16 places. This is implemented as a simple shift operation so it does not require multipliers.
2. **Removal of Fractional Bits**. If any fractional bits exist, you can choose to remove them. There are three methods:
   - Truncate to integer. Fractional bits are removed from the data. This is equivalent to rounding towards negative infinity.
   - Round - Half up. Round up to the nearest integer. If the fractional bits equal 0.5, rounding is towards positive infinity.
   - Round - Half even. Round to the nearest integer. If the fractional bits equal 0.5, rounding is towards the nearest even integer.
3. **Conversion from Signed to Unsigned**. If any negative numbers can exist in the results and the output type is unsigned, you can choose how they are converted. There are two methods:
   - Saturate to the minimum output value (constraining to range).
   - Replace negative numbers with their absolute positive value.
4. **Constrain to Range**. If any of the results are beyond the range specified by the output data type (output guard bands, or if unspecified the minimum and maximum values allowed by the output bits per pixel), logic that saturates the results to the minimum and maximum output values is automatically added.

Avalon-ST Video Protocol Parameters

The Color Space Converter MegaCore function can process streams of pixel data of the types listed inTable 13–1.

Table 13–1. Color Space Converter Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Read from control packets at run time.
Frame Height	Read from control packets at run time.
Interlaced / Progressive	Either.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern (1)	For color planes in sequence:  For color planes in parallel: 

Note to Table 13–1:

(1) For channels in parallel, the top of the color pattern matrix represents the MSB of data and the bottom represents the LSB. For details, refer to "Avalon-ST Video Protocol" on page 3–2.

Stall Behavior and Error Recovery

In all parameterizations, the Color Space Converter only stalls between frames and not between rows. It has no internal buffering apart from the registers of its processing pipeline so there are only a few clock cycles of latency.

Error Recovery

The Color Space Converter MegaCore function processes video packets until an endofpacket signal is received; the control packets are not used. For this MegaCore function, there is no such condition as an early or late endofpacket, any mismatch of the endofpacket signal and the frame size is propagated unchanged to the next MegaCore function.

☞ For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 13–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the Color Space Converter MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 13–2. Color Space Converter Latency

Mode	Latency
All modes	*O* (cycles)

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 13–3 lists the Color Space Converter MegaCore function parameters in the **General Page**.

Table 13–3. Color Space Converter Parameter Settings Tab, General Page (Part 1 of 2)

Parameter	Value	Description
Color Plane Configuration	**Three color planes in sequence**, or Three color planes in parallel	Specifies whether the three color planes are transmitted in sequence or in parallel.
Input Data Type: Bits per pixel per color plane	4–20, Default = **8**	Specifies the number of input bits per pixel (per color plane).
Input Data Type: Data type *(2)*	**Unsigned**, Signed	Specifies whether the input is unsigned or signed 2's complement.
Input Data Type: Guard bands *(1)*	On or **Off**	Enables using a defined input range.
Input Data Type: Max *(1)*	-524288–1048575, Default = **255**	Specifies the input range maximum value.
Input Data Type: Min *(1)*	-524288–1048575, Default = **0**	Specifies the input range minimum value.
Output Data Type: Bits per pixel per color plane *(2)*	4–20, Default = **8**	Choose the number of output bits per pixel (per color plane).
Output Data Type: Data type	**Unsigned**, Signed	Specify whether the output is unsigned or signed 2's complement.
Output Data Type: Guard bands *(1)*	On or **Off**	Turn on to enable a defined output range.
Output Data Type: Max *(1)*	-524288–1048575, Default = **255**	Specify the output range maximum value.
Output Data Type: Min *(1)*	-524288–1048575, Default = **0**	Specify the output range minimum value.
Move binary point right *(2)*	–16 to +16, Default = **0**	Specify the number of places to move the binary point.
Remove fraction bits by	**Round values - Half up**, Round values - Half even, Truncate values to integer	Choose the method of discarding fraction bits resulting from the calculation.

Table 13–3. Color Space Converter Parameter Settings Tab, General Page (Part 2 of 2)

Parameter	Value	Description
Convert from signed to unsigned by	**Saturating to minimum value at stage 4**, Replacing negative with absolute value	Choose the method of signed to unsigned conversion for the results.

Notes to Table 13–3:

(1) When **Guard bands** are on, the MegaCore function never receives or sends data outside of the range specified by **Min** and **Max**.

(2) You can specify a higher precision output by increasing **Bits per pixel per color plane** and **Move binary point right**.

Table 13–4 lists the Color Space Converter MegaCore function parameters in the **Operands Page**.

Table 13–4. Color Space Converter Parameter Settings Tab, Operands Page (Part 1 of 2)

Parameter	Value	Description
Color model conversion *(1)*	**Computer B’G’R’ to CbCrY’: SDTV**, CbCrY’: SDTV to Computer B’G’R’, Computer B’G’R’ to CbCrY’: HDTV, CbCrY’: HDTV to Computer B’G’R’, Studio B’G’R’ to CbCrY’: SDTV, CbCrY’: SDTV to Studio B’G’R’, Studio B’G’R’ to CbCrY’: HDTV, CbCrY’: HDTV to Studio B’G’R’, IQY' to Computer B'G'R', Computer B'G'R' to IQY', UVY' to Computer B'G'R' Computer B'G'R' to UVY', Custom	Specifies a predefined set of coefficients and summands to use for color model conversion at compile time. Alternatively, you can select **Custom** and create your own custom set by modifying the *din_0*, *din_1*, and *din_2* coefficients for *dout_0*, *dout_1*, and *dout_2* separately. The values are assigned in the order indicated by the conversion name. For example, if you select **Computer B’G’R’ to CbCrY’: SDTV**, then *din_0* = B’, *din_1* = G’, *din_2* = R’, *dout_0* = Cb, *dout_1* = Cr, and *dout_2* = Y’.
Run-time controlled	On or **Off**	Turn on to enable run-time control of the conversion values.
Coefficients and Summands A0, B0, C0, S0 A1, B1, C1, S1 A2, B2, C2, S2	12 fixed-point values	Each coefficient or summand is represented by a white cell with a purple cell underneath. The value in the white cell is the desired value, and is editable. The value in the purple cell is the actual value, determined by the fixed-point type specified. The purple cells are not editable. You can create a custom coefficient and summand set by specifying one fixed-point value for each entry. You can paste custom coefficients into the table from a spreadsheet (such as Microsoft Excel). Blank lines must be left in your input data for the non-editable cells.
Coefficients: Signed *(2)*	**On** or Off	Turn on to set the fixed point type used to store the constant coefficients as having a sign bit.
Coefficients: Integer bits *(2)*	0–16, Default = **0**	Specifies the number of integer bits for the fixed point type used to store the constant coefficients.
Summands: Signed *(2)*	On or **Off**	Turn on to set the fixed point type used to store the constant summands as having a sign bit.
Summands: Integer bits *(2)*	0–22, Default = **8**	Specifies the number of integer bits for the fixed point type used to store the constant summands.

Table 13–4. Color Space Converter Parameter Settings Tab, Operands Page (Part 2 of 2)

Parameter	Value	Description
Coefficient and summand fraction bits (2)	0–34, Default = **8**	Specifies the number of fraction bits for the fixed point type used to store the coefficients and summands.

Notes to Table 13–4:

(1) Editing the coefficient values automatically changes the **Color model conversion** value to **Custom**.

(2) Editing these values change the actual coefficients and summands and the results values on the **General** page. Signed coefficients allow negative values; increasing the integer bits increases the magnitude range; and increasing the fraction bits increases the precision.

Signals

Table 13–5 lists the input and output signals for the Color Space Converter MegaCore function.

Table 13–5. Color Space Converter Signals

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

Control Register Maps

Table 13–6 describes the control register map for the Color Space Converter MegaCore function.

The width of each register in the Color Space Converter control register map is 32 bits. The coefficient and summand registers use integer, signed 2's complement numbers. To convert from fractional values, simply move the binary point right by the number of fractional bits specified in the user interface.

The control data is read once at the start of each frame and is buffered inside the MegaCore function, so the registers can be safely updated during the processing of a frame.

Table 13–6. Color Space Converter Control Register Map

Address	Register Name	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the Color Space Converter MegaCore function to stop the next time control information is read. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Coefficient A0`	For more information, refer to "Color Space Conversion" on page 13–1.
3	`Coefficient B0`	
4	`Coefficient C0`	
5	`Coefficient A1`	
6	`Coefficient B1`	
7	`Coefficient C1`	
8	`Coefficient A2`	
9	`Coefficient B2`	
10	`Coefficient C2`	
11	`Summand S0`	
12	`Summand S1`	
13	`Summand S2`	

14. Control Synchronizer MegaCore Function

Core Overview

The Control Synchronizer MegaCore function synchronizes the configuration change of MegaCores with an event in a video stream. For example, the MegaCore function could synchronize the changing of a position of a video layer with the changing of the size of the layer.

Functional Description

The Control Synchronizer MegaCore function has an Avalon Video Streaming Input and Output port, which passes through Avalon-ST Video data, and monitors the data for trigger events. The events that can trigger the Control Synchronizer are the start of a video data packet, or a change in the width or height field of a control data packet that describes the next video data packet.

The Control Synchronizer MegaCore function also has an Avalon Master port. When the Control Synchronizer MegaCore function detects a trigger event the MegaCore writes data to the Avalon Slave control ports of other MegaCores. The Control Synchronizer MegaCore function also has an Avalon Slave port that sets the data to be written and the addresses that the data must be written to when the MegaCore function detects a trigger event.

When the Control Synchronizer MegaCore function detects a trigger event, it immediately stalls the Avalon-ST video data flowing through the MegaCore, which freezes the state of other MegaCore functions on the same video processing data path that do not have buffering in between. The Control Synchronizer then writes the data stored in its Avalon Slave register map to the addresses that are also specified in the register map. Once this writing is complete the Control Synchronizer resumes the Avalon-ST video data flowing through it. This function ensures that any cores after the Control Synchronizer have their control data updated before the start of the video data packet to which the control data applies. Once all the writes from a Control Synchronizer trigger are complete, an interrupt is triggered or is initiated, which is the “completion of writes” interrupt.

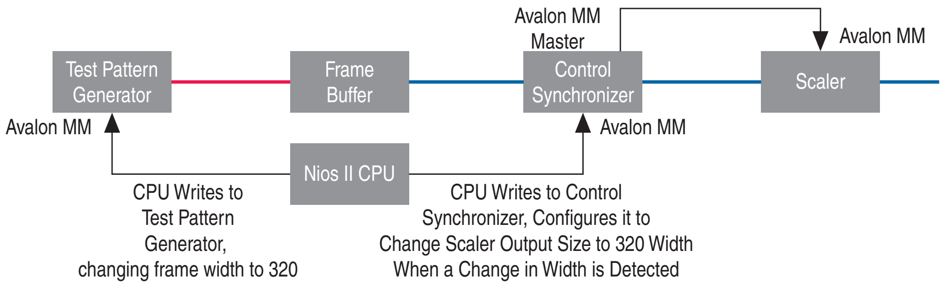
The control synchronizer has an address in its Avalon Slave Control port that you can use to disable or enable the trigger condition. The Control Synchronizer can optionally be configured before compilation to set this register to the “disabled” value after every trigger event, this is useful when using the control synchronizer to trigger only on a single event.

Using the Control Synchronizer

This section provides an example of how to use the Control Synchronizer MegaCore function. The Control Synchronizer is set to trigger on the changing of the width field of control data packets. In the following example, the Control Synchronizer is placed in a system containing a Test Pattern Generator, a Frame Buffer, and a Scaler. The Control Synchronizer must synchronize a change of the width of the generated video packets with a change to the Scaler output size, such that the Scaler maintains a scaling ratio of 1:1 (no scaling). The Frame Buffer is configured to drop and repeat; this makes it impossible to calculate when packets streamed into the Frame Buffer are streamed out to the Scaler, which means that the Scaler cannot be configured in advance of a certain video data packet. The Control Synchronizer solves this problem, as described in the following scenario.

1. Set up the change of video width as shown in Figure 14–1.

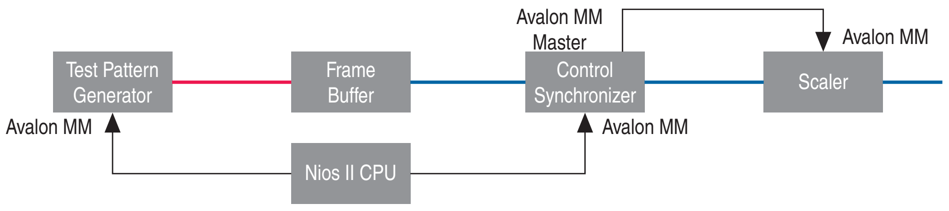
Figure 14–1. Change of Video Width


Avalon MM
Master
Avalon MM
Test Pattern Generator
Frame Buffer
Control Synchronizer
Scaler
Avalon MM
Avalon MM
Nios II CPU
CPU Writes to Test Pattern Generator, changing frame width to 320
CPU Writes to Control Synchronizer, Configures it to Change Scaler Output Size to 320 Width When a Change in Width is Detected

🟥	Red Line Indicates Control Data Packet and Video Data Packet Pair Number 4 (Width 640)
🟦	Blue Line Indicates Control Data Packet and Video Data Packet Pair Number 0 (Width 640)
	Control Data packet and Video Data Packet Pair Numbers 1, 2 and 3 are Stored in the Frame Buffer

2. The Test Pattern Generator has changed the size of its Video Data Packet and Control Data Packet pairs to 320 width. It is not known when this change will propagate through the Frame Buffer to the Scaler (Figure 14–2).

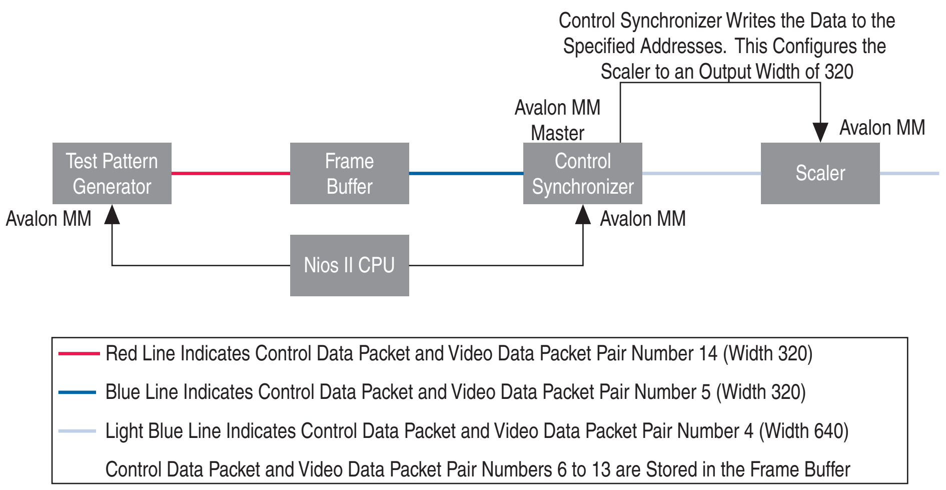
Figure 14–2. Changing Video Width



🟥	Red Line Indicates Control Data Packet and Video Data Packet Pair Number 5 (Width 320)
🟦	Blue Line Indicates Control Data Packet and Video Data Packet Pair Number 1 (Width 640)
	Control Data Packet and Video Data Packet Pair Numbers 2, 3, and 4 are Stored in the Frame Buffer

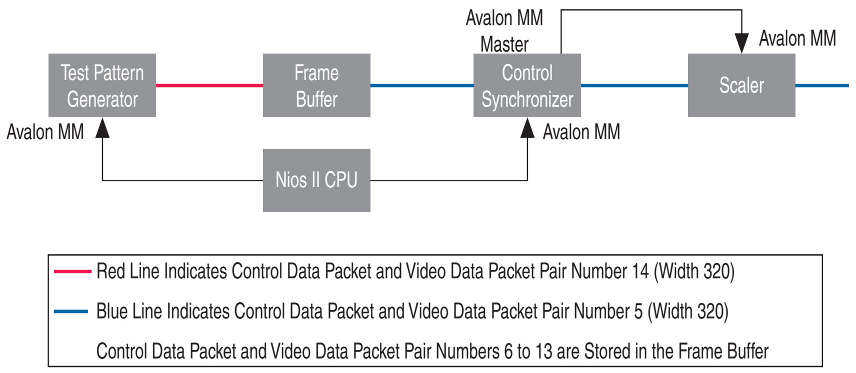
3. The Video Data Packet and Control Data Packet pair with changed width of 320 have propagated through the Frame Buffer. The Control Synchronizer has detected the change and triggered a write to the Scaler. The Control Synchronizer has stalled the video processing pipeline while it performs the write, as shown in Figure 14–3.

Figure 14–3. Test Pattern Generator Change


Control Synchronizer Writes the Data to the Specified Addresses. This Configures the Scaler to an Output Width of 320
Avalon MM Master
Avalon MM
Test Pattern Generator
Frame Buffer
Control Synchronizer
Scaler
Avalon MM
Avalon MM
Nios II CPU
Red Line Indicates Control Data Packet and Video Data Packet Pair Number 14 (Width 320)
Blue Line Indicates Control Data Packet and Video Data Packet Pair Number 5 (Width 320)
Light Blue Line Indicates Control Data Packet and Video Data Packet Pair Number 4 (Width 640)
Control Data Packet and Video Data Packet Pair Numbers 6 to 13 are Stored in the Frame Buffer

4. The Scaler has been reconfigured to output width 320 frames. The Control Synchronizer has resumed the video processing pipeline. At no point did the Scaling ratio change from 1:1, as shown in Figure Figure 14–4.

Figure 14–4. Reconfigured Scaler.


Avalon MM Master
Avalon MM
Test Pattern Generator
Frame Buffer
Control Synchronizer
Scaler
Avalon MM
Avalon MM
Nios II CPU
Red Line Indicates Control Data Packet and Video Data Packet Pair Number 14 (Width 320)
Blue Line Indicates Control Data Packet and Video Data Packet Pair Number 5 (Width 320)
Control Data Packet and Video Data Packet Pair Numbers 6 to 13 are Stored in the Frame Buffer

Avalon-ST Video Protocol Parameters

You can customize the Control Synchronizer according to the parameters listed in Table 14–1.

Table 14–1. Control Synchronizer Avalon-ST Video Protocol Parameters (Part 1 of 2)

Parameter	Value
Frame Width	Run-time controlled. Any valid value supported.
Frame Height	Run-time controlled. Any valid value supported.

Table 14–1. Control Synchronizer Avalon-ST Video Protocol Parameters (Part 2 of 2)

Parameter	Value
Interlaced / Progressive	Run-time controlled. Any valid value supported.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	Up to four color planes in parallel, with any number of color planes in sequence.

Stall Behavior and Error Recovery

The Control Synchronizer stalls for several cycles between packets. When the Control Synchronizer enters a triggered state it stalls while it writes to the Avalon-MM Slave ports of other MegaCore functions. If the slaves do not provide a "wait request" signal, the stall lasts for no more than 50 clock cycles. Otherwise the stall is of unknown length.

Clipper and scaler use the `wait_request` signal.

Error Recovery

The Control Synchronizer MegaCore function processes all packets until an `endofpacket` signal is received; the image width, height and interlaced fields of the control data packets are not compared against the following video data packet. Any mismatch of the `endofpacket` signal and the frame size of a video data packet is propagated unchanged to the next MegaCore function.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 14–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the Control Synchronizer MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 14–2. Control Synchronizer Latency

Mode	Latency
All modes	*O* (cycles)

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 14–3 lists the Control Synchronizer MegaCore function parameters.

Table 14–3. Control Synchronizer Parameter Settings

Parameter	Value	Description
Bits per pixel per color plane	4–16, Default = **8**	The number of bits used per pixel, per color plane.
Number of color planes	1–4, Default = **3**	The number of color planes that are sent over one data connection. For example, a value of **3** for R'G'B' R'G'B' R'G'B' in serial.
Color planes are in parallel	**On** or Off	Color planes are transmitted in parallel or in series.
Trigger on width change	**On** or Off	Trigger compares control packet width values.
Trigger on height change	**On** or Off	Trigger compares control packet height values.
Trigger on start of video data packet	On or **Off**	Trigger activates on each start of video data packet.
Require trigger reset via control port	On or **Off**	Once triggered, the trigger is disabled and must be re-enabled via the control port.
Maximum number of control data entries	1–10, Default = **3**	Maximum number of control data entries that can be written to other cores.

Signals

Table 14–4 lists the input and output signals for the Control Synchronizer MegaCore function.

Table 14–4. Control Synchronizer Signals (Part 1 of 2)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`din_data`	In	`din port Avalon-ST` data bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din port Avalon-ST endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din port Avalon-ST ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din port Avalon-ST startofpacket` signal. This signal marks the start of an Avalon-ST packet.

Table 14–4. Control Synchronizer Signals (Part 2 of 2)

Signal	Direction	Description
`din_valid`	In	`din port Avalon-ST valid` signal. This signal identifies the cycles when the port must input data.
`dout_data`	Out	`dout port Avalon-ST data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout port Avalon-ST endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	in	`dout port Avalon-ST ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout port Avalon-ST startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout port Avalon-ST valid` signal. This signal is asserted when the MegaCore function is outputs data.
`slave_av_address`	In	`slave port Avalon-MM` address. Specifies a word offset into the slave address space.
`slave_av_read`	In	`slave port Avalon-MM read` signal. When you assert this signal, the slave port drives new data onto the read data bus.
`slave_av_readdata`	Out	`slave port Avalon-MM readdata` bus. These output lines are used for read transfers.
`slave_av_write`	In	`slave port Avalon-MM write` signal. When you assert this signal, the `gamma_lut` port accepts new data from the `writedata` bus.
`slave_av_writedata`	In	`slave port Avalon-MM writedata` bus. These input lines are used for write transfers.
`status_update_int_w`	Out	`slave port Avalon-MM interrupt` signal. When asserted the interrupt registers of the MegaCore function have been updated and the master must read them to determine what has occurred.
`master_av_address`	Out	`master port Avalon-MM address` bus. Specifies a byte address in the Avalon-MM address space.
`master_av_writedata`	Out	`master port Avalon-MM writedata` bus. These output lines carry data for write transfers.
`master_av_write`	Out	`master port Avalon-MM write` signal. Asserted to indicate write requests from the master to the system interconnect fabric.
`master_av_waitrequest`	In	master port Avalon-MM waitrequest signal. The system interconnect fabric asserts this signal to cause the master port to wait.

Control Register Maps

The width of each register of the frame reader is 32 bits. The control data is read once at the start of each frame. The registers may be safely updated during the processing of a frame. Table 14–5 describes the Control Synchronizer MegaCore function control register map.

Table 14–5. Control Synchronizer Control Register Map

Address	Register(s)	Description
0	`Control`	Bit 0 of this register is the `Go` bit. Setting this bit to 1 causes the Control Synchronizer MegaCore function to start passing through data. Bit 1 of the `Control` register is the interrupt enable. Setting bit 1 to 1, enables the completion of writes interrupt.
1	`Status`	Bit 0 of this register is the `Status` bit. All other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Interrupt`	Bit 1 of this register is the completion of writes interrupt bit, all other bits are unused. Writing a 1 to bit 1 resets the completion of writes interrupt.
3	`Disable Trigger`	Setting this register to 1 disables the trigger condition of the control synchronizer. Setting this register to 0 enables the trigger condition of the control synchronizer. When the compile time option **Require trigger reset via control port** is enabled this register value is automatically set to 1 every time the Control Synchronizer triggers.
4	`Number of writes`	This register sets how many write operations, starting with address and word 0, are written when the control synchronizer triggers.
5	`Address 0`	Address where word 0 must be written on trigger condition.
6	`Word 0`	The word to write to address 0 on trigger condition.
7	`Address 1`	Address where word 1 must be written on trigger condition.
8	`Word 1`	The word to write to address 1 on trigger condition.
9	`Address 2`	Address where word 2 must be written on trigger condition.
10	`Word 2`	The word to write to address 2 on trigger condition.
11	`Address 3`	Address where word 3 must be written on trigger condition.
12	`Word 3`	The word to write to address 3 on trigger condition.
13	`Address 4e`	Address where word 4 must be written on trigger condition.
14	`Word 4`	The word to write to address 4 on trigger condition.
15	`Address 5`	Address where word 5 must be written on trigger condition.
16	`Word 5`	The word to write to address 5 on trigger condition.
17	`Address 6`	Address where word 6 must be written on trigger condition.
18	`Word 6`	The word to write to address 6 on trigger condition.
19	`Address 7`	Address where word 7 must be written on trigger condition.
20	`Word 7`	The word to write to address 7 on trigger condition.
21	`Address 8`	Address where word 8 must be written on trigger condition.
22	`Word 8`	The word to write to address 8 on trigger condition.
23	`Address 9`	Address where word 9 must be written on trigger condition.
24	`Word 9`	The word to write to address 9 on trigger condition.

15. Deinterlacer MegaCore Function

Core Overview

The Deinterlacer MegaCore function converts interlaced video to progressive video using a bob, weave, or simple motion-adaptive algorithm. Interlaced video is commonly used in television standards such as phase alternation line (PAL) and national television system committee (NTSC), but progressive video is required by LCD displays and is often more useful for subsequent image processing functions.

Additionally, the Deinterlacer can provide double -buffering or triple-buffering in external RAM. Double-buffering can help solve throughput problems (burstiness) in video systems. Triple-buffering can provide simple frame rate conversion.

Functional Description

You can configure the Deinterlacer to produce one output frame for each input field or to produce one output frame for each input frame (a pair of two fields). For example, if the input video stream is NTSC video with 60 interlaced fields per second, the former configuration outputs 60 frames per second but the latter outputs 30 frames per second.

☞ Producing one output frame for each input field must give smoother motion but may also introduce visual artefacts on scrolling text or slow moving objects when using the bob or motion adaptive algorithm.

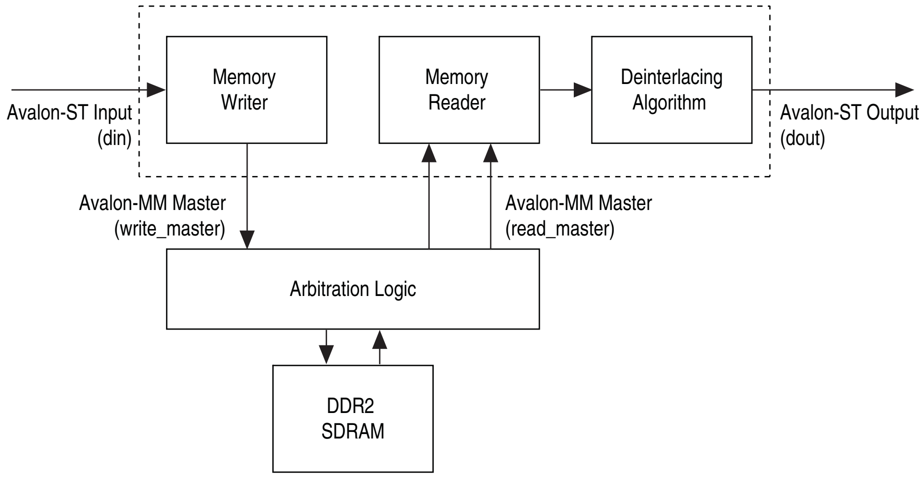
When you select a frame buffering mode, the Deinterlacer output is calculated in terms of the current field and possibly some preceding fields. For example, the bob algorithm uses the current field, whereas the weave algorithm uses both the current field and the one which was received immediately before it. When producing one output frame for every input field, each field in the input frame takes a turn at being the current field.

However, if one output frame is generated for each pair of interlaced fields then the current field moves two fields through the input stream for each output frame. This means that the current field is either always a F0 field (defined as a field which contains the top line of the frame) or always a F1 field.

☞ The Deinterlacer MegaCore function does not currently use the two synchronization bits of the interlace nibble. (Refer to "Control Data Packets" on page 3–7.) When input frame rate = output frame rate, the choice of F0 or F1 to be the current field has to be made at compile time. The deinterlacing algorithm does not adapt itself to handle PsF content.

Figure 15–1 shows a simple block diagram of the Deinterlacer MegaCore function with frame buffering.

Figure 15–1. Deinterlacer Block Diagram with Buffering in External RAM



Note to Figure 15–1:

(1) There can be one or two Avalon-MM masters connected to the Memory Reader.

Deinterlacing Methods

The Deinterlacer MegaCore function supports four deinterlacing methods:

- Bob with scanline duplication
- Bob with scanline interpolation
- Weave
- Motion-adaptive

The Deinterlacer does not support interlaced streams where F0 fields are one line higher than F1 fields in most of its parameterizations. (Bob with one output frame for each input frame is the only exception.) Altera recommends using the Clipper MegaCore function to feed the Deinterlacer with an interlaced video stream that it can support.

Bob with Scanline Duplication

The bob with scanline duplication algorithm is the simplest and cheapest in terms of logic. Output frames are produced by simply repeating every line in the current field twice. The function uses only the current field, therefore if the output frame rate is the same as the input frame rate, the function discards half of the input fields.

Bob with Scanline Interpolation

The bob with scanline interpolation algorithm has a slightly higher logic cost than bob with scanline duplication but offers significantly better quality.

Output frames are produced by filling in the missing lines from the current field with the linear interpolation of the lines above and below them. At the top of an F1 field or the bottom of an F0 field there is only one line available so it is just duplicated. The function only uses the current field, therefore if the output frame rate is the same as the input frame rate, the function discards half of the input fields.

Weave

Weave deinterlacing creates an output frame by filling all of the missing lines in the current field with lines from the previous field. This option gives good results for still parts of an image but unpleasant artefacts in moving parts.

The weave algorithm requires external memory, so either double or triple-buffering must be selected. This makes it significantly more expensive in logic elements and external RAM bandwidth than either of the bob algorithms, if external buffering is not otherwise required.

The results of the weave algorithm can sometimes be perfect, in the instance where pairs of interlaced fields have been created from original progressive frames. Weave simply stitches the frames back together and the results are the same as the original, as long as output frame rate equal to input frame rate is selected and the correct pairs of fields are put together. Usually progressive sources split each frame into a pair consisting of an F0 field followed by an F1 field, so selecting F1 to be the current field often yields the best results.

Motion-Adaptive

The Deinterlacer MegaCore function provides a simple motion-adaptive algorithm. This is the most sophisticated of the algorithms provided but also the most expensive, both in terms of logic area and external memory bandwidth requirement.

This algorithm avoids the weaknesses of bob and weave algorithms by using a form of bob deinterlacing for moving areas of the image and weave style deinterlacing for still areas.

If the input is 4:2:2 Y’CbCr subsampled data, the compatibility mode for 4:2:2 data must be enabled to prevent the motion adaptive algorithm from introducing chroma artefacts when using bob deinterlacing for moving regions.

Use the **Motion bleed** algorithm to prevent the motion value from falling too fast at a specific pixel position. If the motion computed from the current and the previous pixels is higher than the stored motion value, the stored motion value is irrelevant and the function uses the computed motion in the blending algorithm, which becomes the next stored motion value. However, if the computed motion value is lower than the stored motion value, the following actions occur:

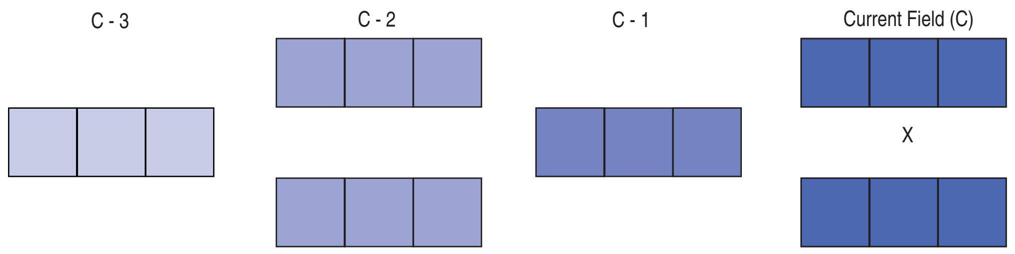
- The blending algorithm uses the stored motion value
- The next stored motion value is an average of the computed motion and of the stored motion

This computed motion means that the motion that the blending algorithm uses climbs up immediately, but takes about four or five frames to stabilize.

The motion-adaptive algorithm fills in the rows that are missing in the current field by calculating a function of other pixels in the current field and the three preceding fields as shown in the following sequence:

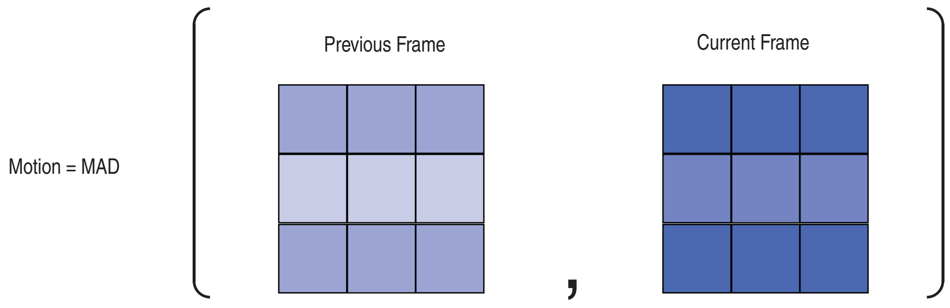
1. Pixels are collected from the current field and the three preceding it (the *X* denotes the location of the desired output pixel) (Figure 15–2).

Figure 15–2. Pixel Collection for the Motion-Adaptive Algorithm



2. These pixels are assembled into two 3×3 groups of pixels. Figure 15–3shows the minimum absolute difference of the two groups.

Figure 15–3. Pixel Assembly for the Motion-Adaptive Algorithm



3. The minimum absolute difference value is normalized into the same range as the input pixel data. If you select the **Motion bleed** algorithm, the function compares the motion value with a recorded motion value for the same location in the previous frame. If it is greater, the function keeps the new value; if the new value is less than the stored value, the function uses the motion value that is the mean of the two values. This action reduces unpleasant flickering artefacts but increases the memory usage and memory bandwidth requirements.
4. Two pixels are selected for interpolation by examining the 3×3 group of pixels from the more recent two fields for edges. If the function detects a diagonal edge, the function selects two pixels from the current field that lie on the diagonal, otherwise the function chooses the pixels directly above and below the output pixel.

   The 4:2:2 compatibility mode prevents incorrect interpolation of the chroma samples along the diagonal edges.

5. The function uses a weighted mean of the interpolation pixels to calculate the output pixel and the equivalent to the output pixel in the previous field with the following equation:

$$\text{Output Pixel} = \text{M} \,.\, \frac{\text{Upper Pixel} + \text{Lower Pixel}}{2} + (1 - \text{M}) \,.\, \text{Still Pixel}$$

The motion-adaptive algorithm requires the buffering of two frames of data before it can produce any output. The Deinterlacer always consumes the three first fields it receives at start up and after a change of resolution without producing any output.

☞ The weave and motion-adaptive algorithms cannot handle fields of different sizes (for example, 244 lines for F0 and 243 lines for F1). Both implementations discard input fields and do not produce an output frame until they receive a sufficient number of consecutive fields with matching sizes.

Pass-Through Mode for Progressive Frames

In its default configuration, the Deinterlacer discards progressive frames. Change this behavior if you want a datapath compatible with both progressive and interlaced inputs and where run-time switching between the two types of input is allowed. When the Deinterlacer lets progressive frames pass through, the deinterlacing algorithm in use (bob, weave or motion-adaptive) propagates progressive frames unchanged. The function maintains the double or triple-buffering function while propagating progressive frames.

☞ Enabling the propagation of progressive frames impacts memory usage in all the parameterizations of the bob algorithm that use buffering.

Frame Buffering

The Deinterlacer MegaCore function also allows frame buffering in external RAM, which you can configure at compile time. When using either of the two bob algorithm subtypes, you can select no buffering, double-buffering, or triple-buffering. The weave and motion-adaptive algorithms require some external frame buffering, and in those cases only select double-buffering or triple-buffering.

When you chose no buffering, input pixels flow into the Deinterlacer through its input port and, after some delay, calculated output pixels flow out through the output port. When you select double-buffering, external RAM uses two frame buffers. Input pixels flow through the input port and into one buffer while pixels are read from the other buffer, processed and output.

When both the input and output sides have finished processing a frame, the buffers swap roles so that the frame that the output can use the frame that you have just input. You can overwrite the frame that the function uses to create the output with a fresh input.

The motion-adaptive algorithm uses four fields to build a progressive output frame and the output side has to read pixels from two frame buffers rather than one. Consequently, the motion-adaptive algorithm actually uses three frame buffers in external RAM when you select double-buffering. When the input and output sides finish processing a frame, the output side exchanges its buffer containing the oldest frame, frame *n*-2, with the frame it receives at the input side, frame *n*. It keeps frame *n*-1 for one extra iteration because it uses it with frame *n* to produce the next output.

When triple-buffering is in use, external RAM usually uses three frame buffers. The function uses four frame buffers when you select the motion-adaptive algorithm. At any time, one buffer is in use by the input and one (two for the motion adaptive case) is (are) in use by the output in the same way as the double-buffering case. The last frame buffer is spare.

This configuration allows the input and output sides to swap asynchronously. When the input finishes, it swaps with the spare frame if the spare frame contains data that the output frame uses. Otherwise the function drops the frame which you have just wrote and the function writes a fresh frame over the dropped frame.

When the output finishes, it also swaps with the spare frame and continues if the spare frame contains fresh data from the input side. Otherwise it does not swap and just repeats the last frame.

Triple-buffering allows simple frame rate conversion. For example, suppose you connect the Deinterlacer's input to a HDTV video stream in 1080i60 format and connect its output i to a 1080p50 monitor. The input has 60 interlaced fields per second, but the output tries to pull 50 progressive frames per second.

If you configure the Deinterlacer to output one frame for each input field, it produces 60 frames of output per second. If you enable triple-buffering, on average the function drops one frame in six so that it produces 50 frames per second. If you chose one frame output for every pair of fields input, the Deinterlacer produces 30 frames per second output and triple-buffering leads to the function repeating two out of every three frames on average.

When you select double or triple-buffering the Deinterlacer has two or more Avalon-MM master ports. These must be connected to an external memory with enough space for all of the frame buffers required. The amount of space varies depending on the type of buffering and algorithm selected. An estimate of the required memory is shown in the Deinterlacer parameter editor.

If the external memory in your system runs at a different clock rate to the Deinterlacer MegaCore function, you can turn on an option to use a separate clock for the Avalon-MM master interfaces and use the memory clock to drive these interfaces.

To prevent memory read and write bursts from being spread across two adjacent memory rows, you can turn on an option to align the initial address of each read and write burst to a multiple of the burst target used for the read and write masters (or the maximum of the read and write burst targets if using different values). Turning on this option may have a negative impact on memory usage but increases memory efficiency.

Frame Rate Conversion

When you select triple-buffering, the decision to drop and repeat frames is based on the status of the spare buffer. Because the input and output sides are not tightly synchronized, the behavior of the Deinterlacer is not completely deterministic and can be affected by the burstiness of the data in the video system. This may cause undesirable glitches or jerky motion in the video output.

By using a double-buffer and controlling the dropping/repeating behavior, the input and output can be kept synchronized. For example, if the input has 60 interlaced fields per second, but the output requires 50 progressive frames per second (fps), setting the input frame rate to 30 fps and the output frame rate at 50 fps guarantees that exactly one frame in six is dropped.

To control the dropping/repeating behavior and to synchronize the input and output sides, you must select double-buffering mode and turn on **Run-time control for locked frame rate conversion** in the **Parameter Settings** tab of the parameter editor. The input and output rates can be selected and changed at run time. Table 15–5 on page 15–15 lists the control register map.

The rate conversion algorithm is fully compatible with a progressive input stream when the progressive passthrough mode is enabled but it cannot be enabled simultaneously with the run-time override of the motion-adaptive algorithm.

Behavior When Unexpected Fields are Received

So far, the behavior of the Deinterlacer has been described assuming an uninterrupted sequence of pairs of interlaced fields (F0, F1, F0, …) each having the same height. Some video streams might not follow this rule and the Deinterlacer adapts its behavior in such cases.

The dimensions and type of a field (progressive, interlaced F0, or interlaced F1) are identified using information contained in Avalon-ST Video control packets. When a field is received without control packets, its type is defined by the type of the previous field. A field following a progressive field is assumed to be a progressive field and a field following an interlaced F0 or F1 field is respectively assumed to be an interlaced F1 or F0 field. If the first field received after reset is not preceded by a control packet, it is assumed to be an interlaced field and the default initial field (F0 or F1) specified in the parameter editor is used.

When the weave or the motion-adaptive algorithms are used, a regular sequence of pairs of fields is expected. Subsequent F0 fields received after an initial F0 field or subsequent F1 fields received after an initial F1 field are immediately discarded.

When the bob algorithm is used and synchronization is done on a specific field (input frame rate = output frame rate), the field that is constantly unused is always discarded. The other field is used to build a progressive frame, unless it is dropped by the triple-buffering algorithm.

When the bob algorithm is used and synchronization is done on both fields (input field rate = output frame rate), the behavior is dependent on whether buffering is used. If double or triple-buffering is used, the bob algorithm behaves like the weave and motion-adaptive algorithms and a strict sequence of F0 and F1 fields is expected. If two or more fields of the same type are received successively, the extra fields are dropped. When buffering is not used, the bob algorithm always builds an output frame for each interlaced input field received regardless of its type.

If passthrough mode for progressive frames has not been selected, the Deinterlacer immediately discards progressive fields in all its parameterizations.

Handling of Avalon-ST Video Control Packets

When buffering is used, the Deinterlacer MegaCore function stores non-image data packets in memory as described in "Buffering of Non-Image Data Packets in Memory" on page 3–21.

Control packets and user packets are never repeated and they are not dropped or truncated as long as memory space is sufficient. This behavior also applies for the parameterizations that do not use buffering in external memory; incoming control and user packets are passed through without modification.

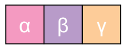
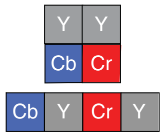
In all parameterizations, the Deinterlacer MegaCore function generates a new and updated control packet just before the processed image data packet. This packet contains the correct frame height and the proper interlace flag so that the following image data packet is interpreted correctly by following MegaCore functions.

☞ The Deinterlacer uses 0010 and 0011 to encode interlacing values into the Avalon-ST Video packets it generates. These flags mark the output as being progressive and record information about the deinterlacing process. (Refer to Table 3–4 on page 3–8.) The interlacing is encoded as 0000 when the Deinterlacer is passing a progressive frame through.

Avalon-ST Video Protocol Parameters

The Deinterlacer MegaCore function can process streams of pixel data of the types listed in Table 15–1.

Table 15–1. Deinterlacer Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Run time controlled. (Maximum value specified in the parameter editor.)
Frame Height	Run time controlled. (Maximum value specified in the parameter editor.)
Interlaced / Progressive	Interlaced input, Progressive output (plus optional passthrough mode for progressive input).
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	One, two or three channels in sequence or in parallel as selected in the parameter editor. For example, for three channels in sequence where α, β, and γ can be any color plane: 
	When the compatibility mode for subsampled 4:2:2 Y'CbCr data is turned on, the motion-adaptive deinterlacer expects the data as either 4:2:2 parallel data (two channels in parallel) or 4:2:2 sequential data (two channels in sequence): 

Stall Behavior and Error Recovery

While the bob algorithm (with no buffering) is producing an output frame it alternates between simultaneously receiving a row on the input port and producing a row of data on the output port, and just producing a row of data on the output port without reading any data from the input port.

The delay from input to output is just a few clock cycles. While a field is being discarded, input is read at the maximum rate and no output is generated.

When you select the weave algorithm, the MegaCore function may stall for longer than the usual periods between each output row of the image. Stalls of up to 45 clock cycles are possible due to the time taken for internal processing in between lines.

When you select the motion-adaptive algorithm, the Deinterlacer may stall up to 90 clock cycles.

Error Recovery

An error condition occurs if an `endofpacket` signal is received too early or too late relative to the field dimensions contained in the last control packet processed. In all its configurations, the Deinterlacer discards extra data if the `endofpacket` signal is received too late.

If an early `endofpacket` signal is received when the Deinterlacer is configured for no buffering, the MegaCore function interrupts its processing within one or two lines sending undefined pixels, before propagating the `endofpacket` signal.

If an early `endofpacket` signal is received when the Deinterlacer is configured to buffer data in external memory, the input side of the MegaCore function stops processing input pixels. It is then ready to process the next frame after writing undefined pixels for the remainder of the current line into external RAM. The output side of the Deinterlacer assumes that incomplete fields have been fully received and pads the incomplete fields to build a frame, using the undefined content of the memory.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 15–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the Deinterlacer MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 15–2. Deinterlacer Latency

Mode	Latency
Method: Bob Frame buffering: None	*O* (cycles)
Method: Motion-adaptive or Weave Frame buffering: Double or triple buffering with rate conversion Output frame rate: As input frame rate	1 frame +*O* (lines)
Method: Motion-adaptive or Weave Frame buffering: Double or triple buffering with rate conversion Output frame rate: As input field rate	1 field +*O* (lines)
Method: All Frame buffering: Double or triple buffering with rate conversion Passthrough mode (propagate progressive frames unchanged): On.	1 frame +*O* (lines)

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 15–3 lists the Deinterlacer MegaCore function parameters.

Table 15–3. Deinterlacer Parameter Settings (Part 1 of 3)

Parameter	Value	Description
Maximum image width	32–2600, Default = **640**	Choose the maximum frame width in pixels. The maximum frame width is the default width at start-up.
Maximum image height *(7)*	32–2600, Default = **480**	Choose the maximum progressive frame height in pixels. The maximum frame height is the default progressive height at start-up.
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color planes in sequence	1–**3**	Choose the number of color planes that are sent in sequence over one data connection. For example, a value of 3 for R'G'B' R'G'B' R'G'B'.
Number of color planes in parallel	**1**–3	Choose the number of color planes in parallel.
Default initial field	**F0**, F1	Choose a default type for the initial field. The default value is not used if the first field is preceded by an Avalon-ST Control packet.
Deinterlacing Method *(1) (8)*	**Bob - Scanline Duplication**, Bob - Scanline Interpolation, Weave, Motion Adaptive	For more information, refer to "Deinterlacing Methods" on page 15–2.

Table 15–3. Deinterlacer Parameter Settings (Part 2 of 3)

Parameter	Value	Description
Frame buffering mode [1], [3], [4], [5]	**No buffering**, Double buffering, Triple buffering with rate conversion	Specifies whether external frame buffers are used. In no buffering mode, data is piped directly from input to output without using external memory. This is possible only with the bob method. Double-buffering routes data via a pair of buffers in external memory. This is required by the weave and motion-adaptive methods, and can ease throughput issues for the bob method. Triple-buffering uses three buffers in external memory and has the advantage over double-buffering that the Deinterlacer can drop or repeat frames, to perform simple frame rate conversion.
Output frame rate [9]	**As input frame rate (F0 synchronized)**, As input frame rate (F1 synchronized), As input field rate	Specifies whether to produce a frame out for every field which is input, or a frame output for every frame (pair of fields) input. Each deinterlacing method is defined in terms of its processing of the current field and some number of preceding fields. In the case where a frame is produced only for every two input fields, the current field is either always an F1 field or always an F0 field.
Passthrough mode	On or **Off**	Turn on to propagate progressive frames unchanged. When off, the progressive frames are discarded.
Run-time control for locked frame rate conversion [2], [6]	On or **Off**	Turn on to add an Avalon-MM slave interface that synchronizes the input and output frame rates.
4:2:2 support for motion adaptive algorithm [2]	On or **Off**	Turn on to avoid color artefacts when processing 4:2:2 Y'CbCr data when the **Motion Adaptive** deinterlacing method is selected. This option cannot be turned on if you are not using either two channels in sequence or two channels in parallel.
Motion bleed	On or **Off**	Turn on to compare the motion value with the corresponding motion value for the same location in the previous frame. If it is greater, the new value is kept, but if the new value is less than the stored value, the motion value used is the mean of the two values. This reduces unpleasant flickering artefacts but increases the memory usage and memory bandwidth requirements. [2]
Run-time control of the motion-adaptive blending	On or **Off**	Turn on to add an Avalon-MM slave interface that controls the behavior of the motion adaptive algorithm at run time. The pixel-based motion value computed by the algorithm can be replaced by a user selected frame-based motion value that varies between the two extremes of being entirely bob or entirely weave. [4], [6]
Number of packets buffered per field	**1**–32	Specify the number of packets that can be buffered with each field. Older packets are discarded first in case of an overflow. [5]
Maximum packet length	**10**–1024	Choose the maximum packet length as a number of symbols. The minimum value is 10 because this is the size of an Avalon-ST control packet (header included). Extra samples are discarded if packets are larger than allowed. [5]
Use separate clocks for the Avalon-MM master interfaces	On or **Off**	Turn on to add a separate clock signal for the Avalon-MM master interfaces so that they can run at a different speed to the Avalon-ST processing. This decouples the memory speed from the speed of the data path and is sometimes necessary to reach performance target.
Avalon-MM master ports width [3]	16, 32, **64**,128, 256	Specifies the width of the Avalon-MM ports used to access external memory when double-buffering or triple-buffering is used.

Table 15–3. Deinterlacer Parameter Settings (Part 3 of 3)

Parameter	Value	Description
Read-only master(s) interface FIFO depth	16–1024, Default = **64**	Choose the FIFO depth of the read-only Avalon-MM interface.
Read-only master(s) interface burst target	2–256, Default = **32**	Choose the burst target for the read-only Avalon-MM interface.
Write-only master(s) interface FIFO depth	16–1024, Default = **64**	Choose the FIFO depth of the write-only Avalon-MM interface.
Write-only master(s) interface burst target	8–256, Default = **32**	Choose the burst target for the write-only Avalon-MM interface.
Base address of frame buffers (3) (10)	Any 32-bit value, Default = **0x00000000**	Hexadecimal address of the frame buffers in external memory when buffering is used.
Align read/write bursts with burst boundaries (3)	On or **Off**	Turn on to avoid initiating read and write bursts at a position that would cause the crossing of a memory row boundary.

Notes to Table 15–3:

(1) Either double or triple-buffering mode must be selected before you can select the weave or motion-adaptive deinterlacing methods.
(2) These options are available only when you select Motion Adaptive as the deinterlacing method.
(3) The options to align read/write bursts on burst boundaries, specify the Avalon-MM master ports width, and the base address for the frame buffers are available only when you select double or triple-buffering.
(4) The option to synchronize input and output frame rates is only available when double-buffering mode is selected.
(5) The options to control the buffering of non-image data packets are available when you select double or triple-buffering.
(6) You cannot enable both run-time control interfaces at the same time.
(7) This MegaCore function does not support interlaced streams where fields are not of the same size (eg, for NTSC, F0 has 244 lines and F1 has 243 lines). Altera recommends that you use the clipper MegaCore function to crop the extra line in F0.
(8) The weave and motion-adaptive algorithms stitch together F1 fields with the F0 fields that precede rather than follow them.
(9) NTSC video transmits 60 interlaced fields per second(30 frames per second). Selecting an **Output frame rate** of **As input frame rate** ensures that the output is 30 frames per second.
(10) The total memory required at the specified base address is displayed under the base address.

Signals

Table 15–4 lists the input and output signals for the Deinterlacer MegaCore function.

Table 15–4. Deinterlacer Signals (Part 1 of 4)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when `reset` is high. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.

Table 15–4. Deinterlacer Signals (Part 2 of 4)

Signal	Direction	Description
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. The MegaCore function asserts this signal when it outputs data.
`ker_writer_control_av_address`	In	`ker_writer_control` slave port Avalon-MM `address` bus. This bus specifies a word offset into the slave address space. *(6)*
`ker_writer_control_av_chipselect`	In	`ker_writer_control` slave port Avalon-MM `chipselect` signal. The `ker_writer_control` port ignores all other signals unless you assert this signal. *(6)*
`ker_writer_control_av_readdata`	Out	`ker_writer_control` slave port Avalon-MM `readdata` bus. The MegaCore function uses these output lines for read transfers. *(6)*
`ker_writer_control_av_waitrequest`	Out	`ker_writer_control` slave port Avalon-MM `waitrequest` signal. *(6)*
`ker_writer_control_av_write`	In	`ker_writer_control` slave port Avalon-MM `write` signal. When you assert this signal, the `ker_writer_control` port accepts new data from the `writedata` bus. *(6)*
`ker_writer_control_av_writedata`	In	`ker_writer_control` slave port Avalon-MM `writedata` bus. The MegaCore function uses these input lines for write transfers. *(6)*
`ma_control_av_address`	In	`ma_control` slave port Avalon-MM `address` bus. This bus specifies a word offset into the slave address space. *(5)*
`ma_control_av_chipselect`	In	`ma_control` slave port Avalon-MM `chipselect` signal. The `ma_control` port ignores all other signals unless you assert this signal. *(5)*
`ma_control_av_readdata`	Out	`ma_control` slave port Avalon-MM `readdata` bus. The MegaCore function uses these output lines for read transfers. *(5)*
`ma_control_av_waitrequest`	Out	`ma_control` slave port Avalon-MM `waitrequest` signal. *(5)*
`ma_control_av_write`	In	`ma_control` slave port Avalon-MM `write` signal. When you assert this signal, the `ma_control` port accepts new data from the `writedata` bus. *(5)*
`ma_control_av_writedata`	In	`ma_control` slave port Avalon-MM `writedata` bus. The MegaCore function uses these input lines for write transfers. *(5)*

Table 15–4. Deinterlacer Signals (Part 3 of 4)

Signal	Direction	Description
`read_master_N_av_address`	Out	`read_master_N` port Avalon-MM `address` bus. This bus specifies a byte address in the Avalon-MM address space. (1), (2), (3)
`read_master_N_av_burstcount`	Out	`read_master_N` port Avalon-MM `burstcount` signal. This signal specifies the number of transfers in each burst. (1), (2), (3)
`read_master_N_av_clock`	In	`read_master_N` port clock signal. The interface operates on the rising edge of the clock signal. (1), (2), (3), (4)
`read_master_N_av_read`	Out	`read_master_N` port Avalon-MM `read` signal. The MegaCore function asserts this signal to indicate read requests from the master to the system interconnect fabric. (1), (2), (3)
`read_master_N_av_readdata`	In	`read_master_N` port Avalon-MM `readdata` bus. These input lines carry data for read transfers. (1), (2), (3)
`read_master_N_av_readdatavalid`	In	`read_master_N` port Avalon-MM `readdatavalid` signal. The system interconnect fabric asserts this signal when the requested read data has arrived. (1), (2), (3)
`read_master_N_av_reset`	In	`read_master_N` port reset signal. The interface asynchronously resets when this signal is high. You must deassert this signal synchronously to the rising edge of the `clock` signal. (1), (2), (3), (4)
`read_master_N_av_waitrequest`	In	`read_master_N` port Avalon-MM `waitrequest` signal. The system interconnect fabric asserts this signal to cause the master port to wait. (1), (2), (3)
`write_master_av_address`	Out	`write_master` port Avalon-MM `address` bus. This bus specifies a byte address in the Avalon-MM address space. (1), (3)
`write_master_av_burstcount`	Out	`write_master` port Avalon-MM `burstcount` signal. This signal specifies the number of transfers in each burst. (1), (2), (3)
`write_master_av_clock`	In	`write_master` port clock signal. The interface operates on the rising edge of the clock signal. (1), (3), (4)
`write_master_av_reset`	In	`write_master` port reset signal. The interface asynchronously resets when this signal is high. You must deassert this signal synchronously to the rising edge of the `clock` signal. (1), (3), (4)
`write_master_av_waitrequest`	In	`write_master` port Avalon-MM `waitrequest` signal. The system interconnect fabric asserts this signal to cause the master port to wait. (1), (3)
`write_master_av_write`	Out	`write_master` port Avalon-MM `write` signal. The MegaCore function asserts this signal to indicate write requests from the master to the system interconnect fabric. (1), (3)

Table 15–4. Deinterlacer Signals (Part 4 of 4)

Signal	Direction	Description
`write_master_av_writedata`	Out	`write_master` port Avalon-MM `writedata` bus. These output lines carry data for write transfers. (1), (3)

Notes to Table 15–4:

(1) The signals associated with the `write_master` and `read_master` ports are present only when buffering is used.

(2) When you select **Motion Adaptive** algorithm, two read master interfaces are used.

(3) When you select **Motion Adaptive** algorithm and turn on **Motion bleed**, one additional read master (`motion_read_master`) and one additional write master (`motion_write_master`) port are used to read and update motion values.

(4) Additional clock and reset signals are available when you turn on **Use separate clocks for the Avalon-MM master interfaces**.

(5) The signals associated with the `ma_control` port are not present unless you turn on **Run-time control of the motion-adaptive blending**.

(6) The signals associated with the `ker_writer_control` port are not present unless you turn on **Run-time control for locked frame rate conversion**.

Control Register Maps

An run-time control interface can be attached to the Deinterlacer that you can use to override the default behavior of the motion-adaptive algorithm or to synchronize the input and output frame rates. However, it is not possible to enable both interfaces simultaneously.

Table 15–5 describes the control register map that controls the motion-adaptive algorithm at run time. The control data is read once and registered before outputting a frame. It can be safely updated during the processing of a frame.

Table 15–5. Deinterlacer Control Register Map for Run-Time Control of the Motion-Adaptive Algorithm

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the Deinterlacer MegaCore function to stop before control information is read and before outputting a frame. While stopped, the Deinterlacer may continue to receive and drop frames at its input if triple-buffering is enabled. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Motion value override`	Write-only register. Bit 0 of this register must be set to 1 to override the per-pixel motion value computed by the deinterlacing algorithm with a user specified value. This register cannot be read.
3	`Blending coefficient`	Write-only register. The 16-bit value that overrides the motion value computed by the deinterlacing algorithm. This value can vary between 0 (weaving) to 65535 (bobbing). The register cannot be read.

Table 15–5 describes the control register map that synchronizes the input and output frame rates. The control data is read and registered when receiving the image data header that signals new frame. It can be safely updated during the processing of a frame.

Table 15–6. Deinterlacer Control Register Map for Synchronizing the Input and Output Frame Rates

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the Deinterlacer MegaCore function to stop before control information is read and before receiving and buffering the next frame. While stopped, the Deinterlacer may freeze the output and repeat a static frame if triple-buffering is enabled. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Input frame rate`	Write-only register. An 8-bit integer value for the input frame rate This register cannot be read. *(1)*
3	`Output frame rate`	Write-only register. An 8-bit integer value for the output frame rate. The register cannot be read. *(1)*

Note to Table 15–6:

(1) The behavior of the rate conversion algorithm is not directly affected by a particular choice of input and output rates but only by their ratio. 23.976 -> 29.970 is equivalent to 24 -> 30.

16. Deinterlacer II MegaCore Function

Core Overview

The Deinterlacer II MegaCore function provides you the option of using high quality motion-adaptive deinterlacing algorithms that significantly enhances edge-adaptive reconstruction and improves image quality streams. The Deinterlacer II does not support lower quality bob and weave deinterlacing. The Deinterlacer II converts interlaced video to progressive video using higher quality motion-adaptive algorithm that enhances the edge-adaptive reconstruction and improves image quality streams.

The buffering behavior is significantly different than the Deinterlacer, and the Deinterlacer II does not support triple buffering. All deinterlacing algorithms in the Deinterlacer II require external RAM.

The Deinterlacer II provides you the option to detect both 3:2 and 2:2 cadences in the input video sequence and perform a reverse telecine operation for perfect restoration of the original progressive video.

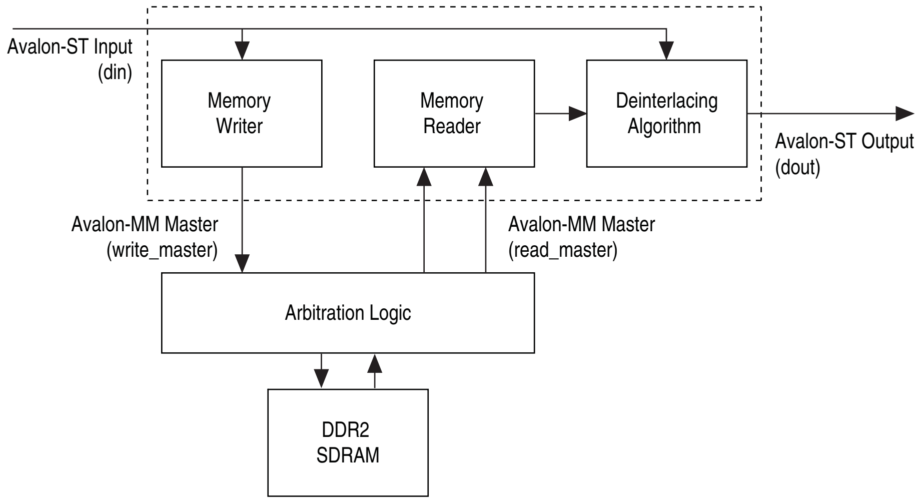
Functional Description

The features and functionality of the Deinterlacer II MegaCore function are largely similar to those of the Deinterlacer MegaCore Function. The Deinterlacer II does not support bob and weave methods but it can convert interlaced video to progressive video using two high quality motion-adaptive methods. The standard motion-adaptive algorithm is largely similar to the Deinterlacer MegaCore function motion-adaptive implementation. The high quality motion-adaptive algorithm uses a kernel of pixels and significantly enhances the edge-adaptive reconstruction to improve image quality.

The Deinterlacer II also uses a different frame buffering method. The Deinterlacer II stores the input video fields in the external memory and concurrently uses these input video fields to construct deinterlaced frames.

Figure 16–1 shows a top-level block diagram of the Deinterlacer II frame buffering.

Figure 16–1. Deinterlacer II Block Diagram



Note to Figure 16–1:

(1) There can be one or two Avalon-MM masters connected to the Memory Reader.

This buffering method provides the following features:

- The Deinterlacer II has a latency of only a few of lines, compared to the Deinterlacer that has a latency of a field.
- The Deinterlacer II requires less memory bandwidth. In normal operating mode, the Deinterlacer II writes incoming input fields into the memory and only fetches the three preceding fields to build the progressive output frame. The simple motion-adaptive algorithm in the Deinterlacer requires four fields to build the progressive output frame. Additionally, the Deinterlacer II does not use external memory when propagating progressive frames.
- The Deinterlacer II does not provide double and triple-buffering, and does not support the user-defined frame rate conversion feature offered in the Deinterlacer.

  You may face throughput issues when you swap the Deinterlacer with the Deinterlacer II in your designs. You can easily fix the throughput issues by instantiating the Frame Buffer MegaCore Function into the designs.

- The Deinterlacer II only allows one output frame for one input field. The Deinterlacer II uses each interlaced field to construct a deinterlaced frame, which effectively doubles the frame rate.
- The Deinterlacer II does not store the Avalon-ST video user packets in the memory, and directly propagates the user packets to the output. However, it does not propagate the receive control packets. The Deinterlacer II builds and sends a new control packet before each output frame.

The Deinterlacer II gives you the option to detect both 3:2 and 2:2 cadences in the input video sequence and perform a reverse telecine operation for perfect restoration of the original progressive content. You can switch off the cadence detector at run time when you enable the slave interface.

When the Deinterlacer II detects a cadence, it maintains the output frame rate at twice the input frame rate. For example, the Deinterlacer II reconstructs a 24 frames-per-second progressive film that is converted and transmitted at 60 fields-per-seconds to a 60 frames-per-second progressive output. The Deinterlacer II repeats the progressive frames of the original content to match the new frame rate.

Stall Behavior and Error Recovery

The Deinterlacer II stores the input video fields in the external memory and concurrently uses these input video fields to construct deinterlaced frames. The MegaCore function stalls up to 50 clock cycles for the first output frame.

For the second output frame, there is an additional delay of one line because the Deinterlacer II generates the last line of the output frame before accepting the first line of the next input field.

For the following output frames, there is a delay of two lines which includes the one line delay from the second output frame.

For all subsequent fields the delay alternates between one and two lines.

Error Recovery

The Deinterlacer II generates a line with the correct length if the MegaCore function receives an early `endofpacket` signal. The video data in the output frame is valid up to the point where the MegaCore function receives the `endofpacket` signal. The Deinterlacer II then stops generating output until it receives the next `startofpacket` signal.

If the Deinterlacer II receives a late `endofpacket` signal, the MegaCore function completes generating the current output frame with the correct number of lines as indicated by the last control packet. The MegaCore function discards the subsequent input lines. Once the MegaCore function receives a `startofpacket` signal, the Deinterlacer II performs a soft reset and it looses the stored cadence or motion values. The MegaCore function resumes deinterlacing anew when it receives the next `startofpacket` signal.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 16–1 lists the approximate latency from the video data input to the video data output for typical usage modes of the Deinterlacer II MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 16–1. Deinterlacer II Latency

Mode	Latency
Method: Motion-adaptive Frame buffering: None Output frame rate: As input field rate	*n* (lines)

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 16–2 lists the Deinterlacer II MegaCore function parameters.

Table 16–2. Deinterlacer II Parameter Settings (Part 1 of 2)

Parameter	Value	Description
Video Data Format		
Maximum frame width	20–2600, Default = **1920**	Choose the maximum frame width in pixels. The maximum frame width is the default width at start-up.
Maximum frame height	32–2600, Default = **480**	Choose the maximum progressive frame height in pixels. The maximum frame height is the default progressive height at start-up.
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Symbols in parallel	1–4, Default = **2**	Choose the number of color planes that are sent in parallel over one data connection. For example, a value of 3 for R'G'B' R'G'B' R'G'B'.
4:2:2 support	**On** or Off	Turn on to use the 4:2:2 data format. Turn off to use 4:4:4 video format.
Behavior		
Deinterlace algorithm	**Motion Adaptive High Quality** or Motion Adaptive	Choose the deinterlacing algorithm. For high quality progressive video sequence, choose the **Motion Adaptive High Quality** option.
Run-time control	On or **Off**	Turn on to enable run-time control for the cadence detection and reverse pulldown. When turned off, the Deinterlacer II always perform cadence detection and reverse pulldown if you turn on the **Cadence detection and reverse pulldown** option.
Cadence detection and reverse pulldown	On or **Off**	Turn on to enable automatic cadence detection and reverse pulldown.
Cadence detection algorithm	**3:2 detector**	Choose the cadence detection algorithm.
Memory		
Avalon-MM master(s) local ports width	16–**256**	Choose the width of the Avalon-MM ports that are used to access the external memory.

Table 16–2. Deinterlacer II Parameter Settings (Part 2 of 2)

Parameter	Value	Description
Use separate clock for the Avalon-MM master interface(s)	**On** or Off	Turn on to add a separate clock signal for the Avalon-MM master interface(s) so that they can run at a different speed to the Avalon-ST processing. This decouples the memory speed from the speed of the data path and is sometimes necessary to reach performance target.
Base address of storage space in memory	0–0x7FFFFFFF, Default = **0x00000000**	Choose a hexadecimal address for the frame buffers in the external memory.
Write Master FIFO depth	8–512, Default = **64**	Choose the FIFO depth of the Avalon-MM write master interface.
Write Master FIFO burst target	2–256, Default = **32**	Choose the burst target for the Avalon-MM write master interface.
EDI Read Master FIFO depth	8–512, Default = **64**	Choose the FIFO depth of the edge-dependent interpolation (EDI) Avalon-MM read master interface.
EDI Read Master FIFO burst target	2–256, Default = **32**	Choose the burst target for the EDI Avalon-MM read master interface.
MA Read Master FIFO depth	8–512, Default = **64**	Choose the FIFO depth of the motion-adaptive (MA) Avalon-MM read master interface.
MA Read Master FIFO burst target	2–256, Default = **32**	Choose the burst target for the MA Avalon-MM read master interface.
Motion Write Master FIFO depth	8–512, Default = **64**	Choose the FIFO depth of the motion Avalon-MM write master interface.
Motion Write Master FIFO burst target	2–256, Default = **32**	Choose the burst target for the motion Avalon-MM write master interface.
Motion Read Master FIFO depth	8–512, Default = **64**	Choose the FIFO depth of the motion Avalon-MM read master interface.
Motion Read Master FIFO burst target	2–256, Default = **32**	Choose the burst target for the motion Avalon-MM read master interface.

Signals

Table 16–3 lists the input and output signals for the Deinterlacer II MegaCore function.

Table 16–3. Deinterlacer II Signals (Part 1 of 4)

Signal	Direction	Description
`av_st_clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `av_st_clock` signal.
`av_st_reset`	In	The MegaCore function asynchronously resets when you assert `av_st_reset`. You must deassert this reset signal synchronously to the rising edge of the `av_st_clock` signal.
`av_mm_clock`	In	Clock for the Avalon-MM interfaces. The interfaces operate on the rising edge of the `av_mm_clock` signal. *(3)*
`av_mm_reset`	In	Reset for the Avalon-MM interfaces. The interfaces asynchronously resets when you assert `av_mm_reset`. You must deassert this reset signal synchronously to the rising edge of the `av_mm_clock` signal. *(3)*

Table 16–3. Deinterlacer II Signals (Part 2 of 4)

Signal	Direction	Description
din_data	In	din port Avalon-ST data bus. This bus enables the transfer of pixel data into the MegaCore function.
din_valid	In	din port Avalon-ST valid signal. This signal identifies the cycles when the port must input data.
din_startofpacket	In	din port Avalon-ST startofpacket signal. This signal marks the start of an Avalon-ST packet.
din_endofpacket	In	din port Avalon-ST endofpacket signal. This signal marks the end of an Avalon-ST packet.
din_ready	Out	din port Avalon-ST ready signal. This signal indicates when the MegaCore function is ready to receive data.
dout_data	Out	dout port Avalon-ST data bus. This bus enables the transfer of pixel data out of the MegaCore function.
dout_valid	Out	dout port Avalon-ST valid signal. The MegaCore function asserts this signal when it outputs data.
dout_startofpacket	Out	dout port Avalon-ST startofpacket signal. This signal marks the start of an Avalon-ST packet.
dout_endofpacket	Out	dout port Avalon-ST endofpacket signal. This signal marks the end of an Avalon-ST packet.
dout_ready	In	dout port Avalon-ST ready signal. The downstream device asserts this signal when it is able to receive data.
control_address	In	control slave port Avalon-MM address bus. This bus specifies a word offset into the slave address space. (4)
control_write	In	control slave port Avalon-MM write signal. When you assert this signal, the control port accepts new data from the writedata bus. (4)
control_writedata	In	control slave port Avalon-MM writedata bus. These input lines are used for write transfers. (4)
control_read	In	control slave port Avalon-MM read signal. When you assert this signal, the control port outputs new data at readdata. (4)
control_readdata	Out	control slave port Avalon-MM readdata bus. These output lines are used for read transfers. (4)
control_readdatavalid	Out	control slave port Avalon-MM readdatavalid bus. The MegaCore function asserts this signal when the readdata bus contains valid data in response to the read signal. (4)
control_waitrequest	Out	control slave port Avalon-MM waitrequest signal. (4)
control_byteenable	In	control slave port Avalon-MM byteenable bus. This bus enables specific byte lane or lanes during transfers. Each bit in byteenable corresponds to a byte in writedata and readdata. During writes, byteenable specifies which bytes are being written to; the slave ignores other bytes. During reads, byteenable indicates which bytes the master is reading. Slaves that simply return readdata with no side effects are free to ignore byteenable during reads. (4)

Table 16–3. Deinterlacer II Signals (Part 3 of 4)

Signal	Direction	Description
`edi_read_master_address`	Out	`edi_read_master` port Avalon-MM `address` bus. This bus specifies a byte address in the Avalon-MM address space. *(1)*
`edi_read_master_read`	Out	`edi_read_master` port Avalon-MM `read` signal. The MegaCore function asserts this signal to indicate read requests from the master to the system interconnect fabric. *(1)*
`edi_read_master_burstcount`	Out	`edi_read_master` port Avalon-MM `burstcount` signal. This signal specifies the number of transfers in each burst. *(1)*
`edi_read_master_readdata`	In	`edi_read_master` port Avalon-MM `readdata` bus. These input lines carry data for read transfers. *(1)*
`edi_read_master_readdatavalid`	In	`edi_read_master` port Avalon-MM `readdatavalid` signal. The system interconnect fabric asserts this signal when the requested read data has arrived. *(1)*
`edi_read_master_waitrequest`	In	`edi_read_master` port Avalon-MM `waitrequest` signal. The system interconnect fabric asserts this signal to cause the master port to wait. *(1)*
`ma_read_master_address`	Out	`ma_read_master` port Avalon-MM `address` bus. This bus specifies a byte address in the Avalon-MM address space. *(1)*
`ma_read_master_read`	Out	`ma_read_master` port Avalon-MM `read` signal. The MegaCore function asserts this signal to indicate read requests from the master to the system interconnect fabric. *(1)*
`ma_read_master_burstcount`	Out	`ma_read_master` port Avalon-MM `burstcount` signal. This signal specifies the number of transfers in each burst. *(1)*
`ma_read_master_readdata`	In	`ma_read_master` port Avalon-MM `readdata` bus. These input lines carry data for read transfers. *(1)*
`ma_read_master_readdatavalid`	In	`ma_read_master` port Avalon-MM `readdatavalid` signal. The system interconnect fabric asserts this signal when the requested read data has arrived. *(1)*
`ma_read_master_waitrequest`	In	`ma_read_master` port Avalon-MM `waitrequest` signal. The system interconnect fabric asserts this signal to cause the master port to wait. *(1)*
`motion_read_master_address`	Out	`motion_read_master` port Avalon-MM `address` bus. This bus specifies a byte address in the Avalon-MM address space. *(1) (2)*
`motion_read_master_read`	Out	`motion_read_master` port Avalon-MM `read` signal. The MegaCore function asserts this signal to indicate read requests from the master to the system interconnect fabric. *(1) (2)*
`motion_read_master_burstcount`	Out	`motion_read_master` port Avalon-MM `burstcount` signal. This signal specifies the number of transfers in each burst. *(1) (2)*
`motion_read_master_readdata`	In	`motion_read_master` port Avalon-MM `readdata` bus. These input lines carry data for read transfers. *(1) (2)*

Table 16–3. Deinterlacer II Signals (Part 4 of 4)

Signal	Direction	Description
`motion_read_master_readdatavalid`	In	`motion_read_master` port Avalon-MM `readdatavalid` signal. The system interconnect fabric asserts this signal when requested read data has arrived. (1) (2)
`motion_read_master_waitrequest`	In	`motion_read_master` port Avalon-MM `waitrequest` signal.The system interconnect fabric asserts this signal to cause the master port to wait. (1) (2)
`write_master_address`	Out	`write_master` port Avalon-MM `address` bus. This bus specifies a byte address in the Avalon-MM address space.
`write_master_write`	Out	`write_master` port Avalon-MM `write` signal. The MegaCore function asserts this signal to indicate write requests from the master to the system interconnect fabric.
`write_master_burstcount`	Out	`write_master` port Avalon-MM `burstcount` signal. This signal specifies the number of transfers in each burst.
`write_master_writedata`	Out	`write_master` port Avalon-MM `writedata` bus. These output lines carry data for write transfers.
`write_master_waitrequest`	In	`write_master` port Avalon-MM `waitrequest` signal. The system interconnect fabric asserts this signal to cause the master port to wait.
`motion_write_master_address`	Out	`motion_write_master` port Avalon-MM `address` bus. This bus specifies a byte address in the Avalon-MM address space. (2)
`motion_write_master_write`	Out	`motion_write_master` port Avalon-MM `write` signal. The MegaCore function asserts this signal to indicate write requests from the master to the system interconnect fabric. (2)
`motion_write_master_burstcount`	Out	`motion_write_master` port Avalon-MM `burstcount` signal. This signal specifies the number of transfers in each burst. (2)
`motion_write_master_writedata`	Out	`motion_write_master` port Avalon-MM `writedata` bus. These output lines carry data for write transfers. (2)
`motion_write_master_waitrequest`	In	`motion_write_master` port Avalon-MM `waitrequest` signal.The system interconnect fabric asserts this signal to cause the master port to wait. (2)

Notes to Table 16–3:

(1) Two read master interfaces are used: `edi_read_master` and `ma_read_master`.

(2) When you select **Motion Adaptive High Quality** or **Motion Adaptive**, one additional read master (`motion_read_master`) and one additional write master (`motion_write_master`) ports are used to read and update motion values.

(3) Additional `av_mm_clock` and `av_mm_reset` signals are available when you turn on **Use separate clocks for the Avalon-MM master interface(s)**.

(4) The signals associated with the `control` slave port are not present unless you enable **Run-time control**.

Control Map Registers

Table 16–4 describes the Deinterlacer II MegaCore function control register map. The Deinterlacer II reads the control data once at the start of each frame and buffers the data inside the MegaCore function. The registers may safely update during the processing of a frame.

Table 16–4. Deinterlacer II Control Register Map for Run-Time Control of the Motion-Adaptive Algorithm

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the Deinterlacer II to stop after generating the current output frame.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. When this bit is set to 0, the Deinterlacer II either gets disabled through the `Go` bit or waits to receive video data.
2	Reserved	This register is reserved for future use.
3	`Cadence detect on`	Setting bit 0 of this register to 1 enables cadence detection. Setting bit 0 of this register to 0 disables cadence detection. Cadence detection is disabled on reset.
4	`Cadence detected`	Reading a 1 from bit 0, indicates that the Deinterlacer II has detected a cadence and is performing reverse telecine. Reading a 0 indicates otherwise.

17. Frame Reader MegaCore Function

Core Overview

The Frame Reader MegaCore function reads video frames stored in external memory and outputs them as a video stream. You can configure the MegaCore function to read multiple video frames using an Avalon-MM slave interface.

Functional Description

The Frame Reader reads video frames stored in external memory and outputs them using the Avalon-ST Video protocol.

The Frame Reader has an Avalon-MM read master port that reads data from an external memory. The Frame Reader has an Avalon-ST source on which it streams video data using the Avalon-ST Video protocol. The Frame Reader also has an Avalon slave port, which provides the MegaCore function with configuration data.

Video frames are stored in external memory as raw video data (pixel values only). Immediately before the Frame Reader MegaCore function reads video data from external memory it generates a control packet and the header of a video data packet on its Avalon-ST source. The video data from external memory is then streamed as the payload of the video data packet. The content of the control data packet is set via the Avalon Slave port. This process is repeated for every video frame read from external memory.

The Frame Reader is configured during compilation to output a fixed number of color planes in parallel, and a fixed number of bits per pixel per color plane. In terms of Avalon-ST Video, these parameters describe the structure of one cycle of a color pattern, also known as the single-cycle color pattern.

☞ The Frame Reader is also configured with the number of channels in sequence, this parameter does not contribute to the definition of the single-cycle color pattern.

To configure the Frame Reader to read a frame from memory, the Frame Reader must know how many single-cycle color patterns make up the frame. If each single-cycle color pattern represents a pixel, then the quantity is simply the number of pixels in the frame. Otherwise, the quantity is the number of pixels in the frame, multiplied by the number of single-cycle color patterns required to represent a pixel.

You must also specify the number of words the Frame Reader must read from memory. The width of the word is the same as the Avalon-MM read **Master port width** parameter. This width is configured during compilation. Each word can only contain whole single-cycle color patterns. The words cannot contain partial single-cycle color patterns. Any bits of the word that cannot fit another whole single-cycle color pattern are not used.

Also, the Frame Reader must be configured with the starting address of the video frame in memory, and the width, height, and interlaced values of the control data packet to output before each video data packet.

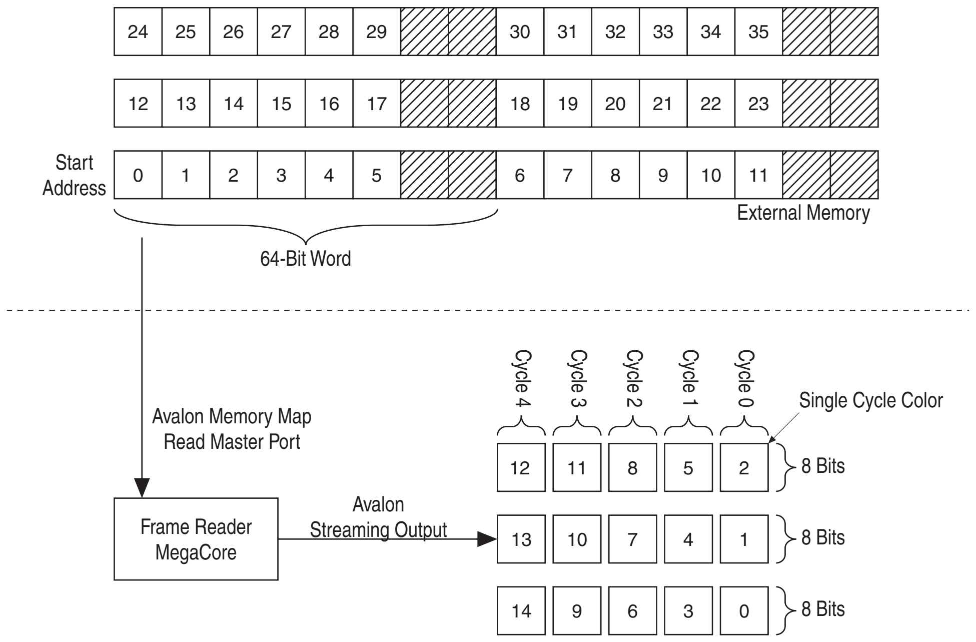
The raw data that comprises a video frame in external memory is stored as a set of single-cycle color patterns. In memory, the single-cycle color patterns must be organized into word-sized sections. Each of these word-sized sections must contain as many whole samples as possible, with no partial single-cycle color patterns. Unused bits are in the most significant portion of the word-sized sections. Single-cycle color patterns in the least significant bits are output first. The frame is read with words at the starting address first.

Figure 17–1 shows the output pattern and memory organization for a Frame Reader MegaCore, which is configured for:

- 8 bits per pixel per color plane
- 3 color planes in parallel
- Master port width 64

Other Frame Reader parameters affect only resources and performance, or both. For more information, refer to Table 17–1.

Figure 17–1. Frame Reader Output Pattern and Memory Organization



The Avalon Slave control port allows the specification of up to two memory locations, each containing a video frame. Switching between these memory locations is performed with a single register. This allows the Frame Reader MegaCore function to read a series of frames from different memory addresses without having to set multiple registers within the period of a single frame. This feature is useful when reading very small frames, and helps to simplify control timing. To aid the timing of control instructions and to monitor the core, the Frame Reader MegaCore function also has an interrupt that fires once per video data packet output, which is the “frame completed” interrupt.

Avalon-ST Video Protocol Parameters

The Avalon-ST Video parameters for the Frame Reader MegaCore function are listed in Table 17–1.

Table 17–1. Frame Reader Avalon-ST Video Parameters

Parameter	Value
Frame Width	Set via the Avalon-MM slave control port. Maximum value specified in parameter editor.
Frame Height	Set via the Avalon-MM slave control port. Maximum value specified in parameter editor.
Interlaced / Progressive	Set via the Avalon-MM slave control port, all values supported.
Bits per Color Sample	Specified in parameter editor.
Color Pattern	Up to four color planes in parallel, with up to three color planes in sequence.

Stall Behavior and Error Recovery

The Frame Reader MegaCore function stalls the output for several tens of cycles before outputting each video data packet, and stalls the output where there is contention for access to external memory. The Frame Reader MegaCore can be stalled due to backpressure, without consequences and it does not require error recovery.

☞ For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

The Frame Reader MegaCore function does not have latency issue because the MegaCore function is only an Avalon-ST Video source.

Parameter Settings

Table 17–2 lists the Frame Reader parameters.

Table 17–2. Frame Reader Parameter Settings (Part 1 of 2)

Parameter	Value	Description
Bits per pixel per color plane	4–16, Default = **8**	The number of bits used per pixel, per color plane
Number of color planes in parallel	1–4, Default = **3**	The number color planes transmitted in parallel
Number of color planes in sequence	1–3, Default = **3**	The maximum number of color planes transmitted in sequence
Maximum image width	32–2600, Default = **640**	The maximum width of images / video frames
Maximum image height	32–2600, Default = **480**	The maximum height of images / video frames
Master port width	16–256, Default = **256**	The width in bits of the master port
Read master FIFO depth	8–1024, Default = **64**	The depth of the read master FIFO

Table 17–2. Frame Reader Parameter Settings (Part 2 of 2)

Parameter	Value	Description
Read master FIFO burst target	2–256, Default = **32**	The target burst size of the read master
Use separate clock for the Avalon-MM master interface	**On** or Off	Use separate clock for the Avalon-MM master interface

Signals

Table 17–3 lists the input and output signals for the Frame Reader MegaCore function.

Table 17–3. Frame Reader Signals (Part 1 of 2)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`dout_data`	Out	`dout port Avalon-ST data bus`. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout port Avalon-ST endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout port Avalon-ST ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout port Avalon-ST startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout port Avalon-ST valid` signal. This signal is asserted when the MegaCore function outputs data.
`slave_av_address`	In	`slave port Avalon-MM` address. Specifies a word offset into the slave address space.
`slave_av_read`	In	`slave port Avalon-MM read` signal. When you assert this signal, the slave port drives new data onto the read data bus.
`slave_av_readdata`	Out	`slave port Avalon-MM readdata` bus. These output lines are used for read transfers.
`slave_av_write`	In	`slave port Avalon-MM write` signal. When you assert this signal, the `gamma_lut` port accepts new data from the `writedata` bus.
`slave_av_writedata`	In	`slave port Avalon-MM writedata` bus. These input lines are used for write transfers.
`slave_av_irq`	Out	`slave port Avalon-MM interrupt` signal. When asserted the interrupt registers of the MegaCore function have been updated and the master must read them to determine what has occurred.
`master_av_address`	Out	`master port Avalon-MM` address bus. Specifies a byte address in the Avalon-MM address space.
`master_av_burstcount`	Out	`master port Avalon-MM burstcount` signal. Specifies the number of transfers in each burst.

Table 17–3. Frame Reader Signals (Part 2 of 2)

Signal	Direction	Description
`master_av_read`	Out	`master port Avalon-MM read` signal. Asserted to indicate read requests from the master to the system interconnect fabric.
`master_av_readdata`	In	`master port Avalon-MM readdata` bus. These input lines carry data for read transfers.
`master_av_readdatavalid`	In	`master port Avalon-MM readdatavalid` signal. The system interconnect fabric asserts this signal when the requested read data has arrived.
`master_av_waitrequest`	In	`master port Avalon-MM waitrequest` signal. The system interconnect fabric asserts this signal to cause the master port to wait.
`master_av_reset`	In	`master port reset` signal. The interface asynchronously resets when you assert this signal. You must deassert this signal synchronously to the rising edge of the `clock` signal.
master_av_clock	In	`master port The clock` signal. The interface operates on the rising edge of the clock signal.

Control Register Maps

The width of each register of the frame reader is 32 bits. The control data is read once at the start of each frame. The registers may be safely updated during the processing of a frame. Table 17–4 describes the Frame Reader run-time control registers.

Table 17–4. Frame Reader Register Map for Run-Time Control (Part 1 of 2)

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit. Setting this bit to 1 causes the Frame Reader to start outputting data. Bit 1 of the `Control` register is the interrupt enable. Setting bit 1 to 1, enables the end of frame interrupt.
1	`Status`	Bit 0 of this register is the `Status` bit. All other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Interrupt`	Bit 1 of this register is the end of frame interrupt bit. All other bits are unused. Writing a 1 to bit 1 resets the end of frame interrupt.
3	`Frame Select`	This register selects between frame 0 and frame 1 for next output. Frame 0 is selected by writing a 0 here, frame is selected by writing a 1 here.
4	`Frame 0 Base Address`	The 32-bit base address of the frame.
5	`Frame 0 Words`	The number of words (reads from the master port) to read from memory for the frame.
6	`Frame 0 Single Cycle Color Patterns`	The number of single-cycle color patterns to read for the frame.
7	`Frame 0 Reserved`	Reserved for future use.
8	`Frame 0 Width`	The Width to be used for the control packet associated with frame 0.
9	`Frame 0 Height`	The Height to be used for the control packet associated with frame 0.
10	`Frame 0 Interlaced`	The interlace nibble to be used for the control packet associated with frame 0.

Table 17–4. Frame Reader Register Map for Run-Time Control (Part 2 of 2)

Address	Register	Description
11	`Frame 1 Base Address`	The 32-bit base address of the frame.
12	`Frame 1 Words`	The number of words (reads from the master port) to read from memory for the frame.
13	`Frame 1 Single Cycle Color Patterns`	The number of single-cycle color patterns to read for the frame.
14	`Frame 1 Reserved`	Reserved for future use.
15	`Frame 1 Width`	The Width to be used for the control packet associated with the frame.
16	`Frame 1 Height`	The Height to be used for the control packet associated with the frame.
17	`Frame 1 Interlaced`	The interlace nibble to be used for the control packet associated with the frame.

18. Frame Buffer MegaCore Function

Core Overview

The Frame Buffer MegaCore function buffers video frames into external RAM. The Frame Buffer supports double or triple buffering with a range of options for frame dropping and repeating.

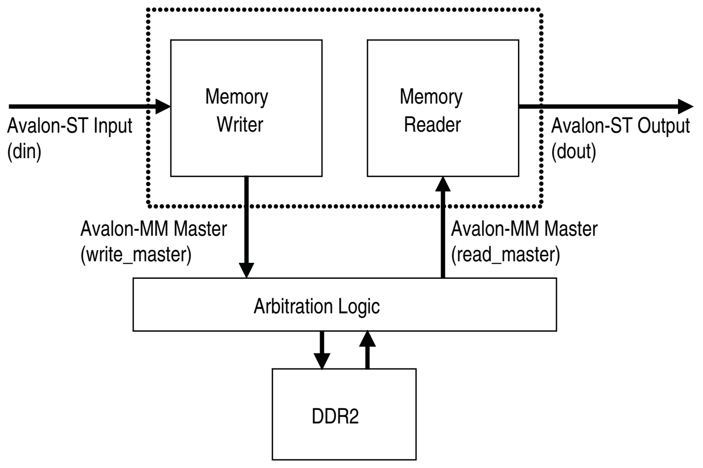
Functional Description

The Frame Buffer MegaCore function buffers progressive or interlaced video fields in external RAM. When frame dropping and frame repeating are not allowed, the Frame Buffer provides a double-buffering function that can be useful to solve throughput issues in the data path. When frame dropping and/or frame repeating are allowed, the Frame Buffer provides a triple-buffering function and can be used to perform simple frame rate conversion.

The Frame Buffer is built with two basic blocks: a writer which stores input pixels in memory and a reader which retrieves video frames from the memory and outputs them.

Figure 18–1 shows a simple block diagram of the Frame Buffer MegaCore function.

Figure 18–1. Frame Buffer Block Diagram



When double-buffering is in use, two frame buffers are used in external RAM. At any time, one buffer is used by the writer component to store input pixels, while the second buffer is locked by the reader component that reads the output pixels from the memory.

When both the writer and the reader components have finished processing a frame, the buffers are exchanged. The frame that has just been input can then be read back from the memory and sent to the output, while the buffer that has just been used to create the output can be overwritten with fresh input.

A double-buffer is typically used when the frame rate is the same both at the input and at the output sides but the pixel rate is highly irregular at one or both sides.

A double-buffer is often used when a frame has to be received or sent in a short period of time compared with the overall frame rate. For example, after the Clipper MegaCore function or before one of the foreground layers of the Alpha Blending Mixer MegaCore function.

When triple-buffering is in use, three frame buffers are used in external RAM. As was the case in double-buffering, the reader and the writer components are always locking one buffer to respectively store input pixels to memory and read output pixels from memory. The third frame buffer is a spare buffer that allows the input and the output sides to swap buffers asynchronously. The spare buffer is considered *clean* if it contains a fresh frame that has not been output, or *dirty* if it contains an old frame that has already been sent by the reader component.

When the writer has finished storing a frame in memory, it swaps its buffer with the spare buffer if the spare buffer is *dirty*. The buffer locked by the writer component becomes the new spare buffer and is *clean* because it contains a fresh frame. If the spare buffer is already *clean* when the writer has finished writing the current input frame and if dropping frames is allowed, then the writer drops the frame that has just been received and overwrites its buffer with the next incoming frame. If dropping frames is not allowed, the writer component stalls until the reader component has finished its frame and replaced the spare buffer with a *dirty* buffer.

Similarly, when the reader has finished reading and has output a frame from memory, it swaps its buffer with the spare buffer if the spare buffer is *clean*. The buffer locked by the reader component becomes the new spare buffer and is *dirty* because it contains an old frame that has been sent previously. If the spare buffer is already *dirty* when the reader has finished the current output frame and if repeating frames are allowed, the reader immediately repeats the frame that has just been sent. If repeating frames is not allowed, the reader component stalls until the writer component has finished its frame and replaced the spare buffer with a *clean* buffer.

Triple-buffering therefore allows simple frame rate conversion to be performed when the input and the output are pushing and pulling frames at different rates.

Locked Frame Rate Conversion

With the triple-buffering algorithm described previously, the decision to drop and repeat frames is based on the status of the spare buffer. Because the input and output sides are not tightly synchronized, the behavior of the Frame Buffer is not completely deterministic and can be affected by the burstiness of the data in the video system. This may cause undesirable glitches or jerky motion in the video output, especially if the data path contains more than one triple buffer.

By controlling the dropping/repeating behavior, the input and output can be kept synchronized. To control the dropping/repeating behavior and to synchronize the input and output sides, you must select triple-buffering mode and turn on **Run-time control for locked frame rate conversion** in the **Parameter Settings** tab of the parameter editor. The input and output rates can be selected and changed at run time. Using the slave interface, it is also possible to enable or disable synchronization at run time to switch between the user-controlled and flow-controlled triple-buffering algorithms as necessary.

Table 18–5 on page 18–8 lists the control register maps for the Frame Buffer writer component.

Interlaced Video Streams

In its default configuration the Frame Buffer MegaCore function does not differentiate between interlaced and progressive fields. When interlaced fields are received, the MegaCore function buffers, drops, or repeats fields independently. While this may be appropriate, and perhaps even desired, behavior when using a double-buffer, it is unlikely to provide the expected functionality when using a triple-buffer because using a triple-buffer would result in an output stream with consecutive F0 or F1 fields.

When you turn on **Support for interlaced streams**, the Frame Buffer manages the two interlaced fields of a frame as a single unit to drop and repeat fields in pairs. Using **Support for interlaced streams** does not prevent the Frame Buffer from handling progressive frames, and run-time switching between progressive and interlaced video is supported.

The Frame Buffer typically groups the first interlaced field it receives with the second one unless a synchronization is specified. If synchronizing on F1, the algorithm groups each F1 field with the F0 field that precedes it. If a F1 field is received first, the field is immediately discarded, even if dropping is not allowed.

For more information, refer to "Control Data Packets" on page 3–7.

Handling of Avalon-ST Video Control Packets

The Frame Buffer MegaCore function stores non-image data packets in memory as described in "Buffering of Non-Image Data Packets in Memory" on page 3–21. User packets are never repeated and they are not dropped as long as the memory space is sufficient. Control packets are not stored in memory. Input control packets are processed and discarded by the writer component and output control packets are regenerated by the reader component.

When a frame is dropped by the writer, the non-image data packets that preceded it are kept and sent with the next frame that is not dropped. When a frame is repeated by the reader, it is repeated without the packets that preceded it.

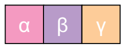
The behavior of the Frame Buffer MegaCore function is not determined by the field dimensions announced in Avalon-ST Video control packets and relies exclusively on the `startofpacket` and `endofpacket` signals to delimit the frame boundaries. The Frame Buffer can consequently handle and propagate mislabelled frames. This feature can be used in a system where dropping frame is not an acceptable option. The latency introduced during the buffering could provide enough time to correct the invalid control packet.

Buffering and propagation of image data packets incompatible with preceding control packets is an undesired behavior in most systems. Dropping invalid frames is often a convenient and acceptable way of dealing with glitches from the video input and the Frame Buffer can be parameterized to drop all mislabelled fields or frames at compile time. Enabling flow-controlled frame repetition and turning on this option can guarantee that the reader component keeps on repeating the last valid received frame, that is, freezes the output, when the input drops.

Avalon-ST Video Protocol Parameters

The Frame Buffer MegaCore function can process streams of pixel data of the type listed in Table 18–1.

Table 18–1. Frame Buffer Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Run time controlled. Maximum value selected in the parameter editor.
Frame Height	Run time controlled. Maximum value selected in the parameter editor.
Interlaced / Progressive	Progressive, although interlaced data can be accepted in some cases.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	Any combination of one, two, three, or four channels in each of sequence or parallel. For example, for three channels in sequence where α, β, and γ can be any color plane: 

Stall Behavior and Error Recovery

The Frame Buffer MegaCore function may stall frequently and read or write less than once per clock cycle during control packet processing. During data processing at the input or at the output, the stall behavior of the Frame Buffer is largely decided by contention on the memory bus.

Error Recovery

The Frame Buffer MegaCore function does not rely on the content of the control packets to determine the size of the image data packets. There is consequently no error condition such as early or late `endofpacket` signal and any mismatch between the size of the image data packet and the content of the control packet is propagated unchanged to the next MegaCore function. Nevertheless, the Frame Buffer does not write outside the memory allocated for each non-image and image Avalon-ST Video packet, and packets are truncated if they are larger than the maximum size defined at compile time.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 18–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 18–2. Latency

Mode	**Latency**
All modes	1 frame +*0* lines

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 18–3 lists the Frame Buffer parameters.

Table 18–3. Frame Buffer Parameter Settings (Part 1 of 2)

Parameter	**Value**	**Description**
Maximum image width	32–2600, Default = **640**	Specify the maximum frame width.
Maximum image height	32–2600, Default = **480**	Specify the maximum frame height. In general, this value should be set to the full height of a progressive frame. However, it can be set to the height of an interlaced field for double-buffering on a field-by-field basis when the support for interlaced inputs has been turned off.
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color planes in sequence	1–**3**	Choose the number of color planes in sequence.
Number of color planes in parallel	**1**–3	Choose the number of color planes in parallel.
Frame dropping	**On** or Off	Turn on to allow frame dropping.
Frame repetition	**On** or Off	Turn on to allow frame repetition.
Drop invalid fields/frames	On or **Off**	Turn on to drop image data packets whose length is not compatible with the dimensions declared in the last control packet.
Run-time control for the writer thread	On or **Off**	Turn on to enable run-time control for the write interfaces.
Run-time control for the reader thread	On or **Off**	Turn on to enable run-time control for the read interfaces.
Support for locked frame rate conversion *(1)*, *(2)*	On or **Off**	Turn on to synchronize the input and output frame rates through an Avalon-MM slave interface.
Support for interlaced streams	On or **Off**	Turn on to support consistent dropping and repeating of fields in an interlaced video stream. This option must not be turned on for double-buffering of an interlaced input stream on a field-by-field basis.
Number of packets buffered per frame *(3)*	**0**–32	Specify the maximum number of non-image, non-control, Avalon-ST Video packets that can be buffered with each frame. Older packets are discarded first in case of an overflow.
Maximum packet length	**10**–1024	Specify the maximum packet length as a number of symbols. The minimum value is 10 because this is the size of an Avalon-ST control packet (header included). Extra samples are discarded if packets are larger than allowed.

Table 18–3. Frame Buffer Parameter Settings (Part 2 of 2)

Parameter	Value	Description
Use separate clocks for the Avalon-MM master interfaces	On or **Off**	Turn on to add a separate clock signal for the Avalon-MM master interfaces so that they can run at a different speed to the Avalon-ST processing. This decouples the memory speed from the speed of the data path and is sometimes necessary to reach performance target.
External memory port width	16–256, Default = **64**	Choose the width of the external memory port.
Write-only master interface FIFO depth	16–1024, Default = **64**	Choose the FIFO depth of the write-only Avalon-MM interface.
Write-only master interface burst target	2–256, Default = **32**	Choose the burst target for the write-only Avalon-MM interface.
Read-only master interface FIFO depth	16–1024, Default = **64**	Choose the FIFO depth of the read-only Avalon-MM interface.
Read-only master interface burst target	2–256, Default = **32**	Choose the burst target for the read-only Avalon-MM interface.
Base address of frame buffers (4)	Any 32-bit value, Default = **0x00000000**	Choose a hexadecimal address for the frame buffers in external memory.
Align read/write bursts with burst boundaries	On or **Off**	Turn on to avoid initiating read and write bursts at a position that would cause the crossing of a memory row boundary.

Notes to Table 18–3:

(1) Locked frame rate conversion cannot be turned on until dropping and repeating are allowed.

(2) Locked frame rate conversion cannot be turned on if the run-time control interface for the writer component has not been enabled.

(3) The **Maximum packet length** option is not available when the **Number of packets buffered per frame** is set to 0.

(4) The number of frame buffers and the total memory required at the specified base address is displayed under the base address.

Signals

Table 18–4 lists the input and output signals for the Frame Buffer MegaCore function.

Table 18–4. Frame Buffer Signals (Part 1 of 3)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.

Table 18–4. Frame Buffer Signals (Part 2 of 3)

Signal	Direction	Description
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function is outputs data.
`read_master_av_address`	Out	`read_master` port Avalon-MM `address` bus. Specifies a byte address in the Avalon-MM address space.
`read_master_av_burstcount`	Out	`read_master` port Avalon-MM `burstcount` signal. Specifies the number of transfers in each burst.
`read_master_av_clock`	In	`read_master` port The clock signal. The interface operates on the rising edge of the clock signal. *(1)*
`read_master_av_read`	Out	`read_master` port Avalon-MM `read` signal. Asserted to indicate read requests from the master to the system interconnect fabric.
`read_master_av_readdata`	In	`read_master` port Avalon-MM `readdata` bus. These input lines carry data for read transfers.
`read_master_av_readdatavalid`	In	`read_master` port Avalon-MM `readdatavalid` signal. The system interconnect fabric asserts this signal when the requested read data has arrived.
`read_master_av_reset`	In	`read_master` port reset signal. The interface resets asynchronously when you assert this signal. You must deassert this signal synchronously to the rising edge of the clock signal. *(1)*
`read_master_av_waitrequest`	In	`read_master` port Avalon-MM `waitrequest` signal. The system interconnect fabric asserts this signal to cause the master port to wait. *(2)*
`reader_control_av_chipselect`	In	`reader_control` slave port Avalon-MM `chipselect` signal. The `reader_control` port ignores all other signals unless you assert this signal. *(2)*
`reader_control_av_readdata`	Out	`reader_control` slave port Avalon-MM `readdata` bus. These output lines are used for read transfers. *(2)*
`reader_control_av_write`	In	`reader_control` slave port Avalon-MM `write` signal. When you assert this signal, the `reader_control` port accepts new data from the `writedata` bus. *(2)*
`reader_control_av_writedata`	In	`reader_control` slave port Avalon-MM `writedata` bus. These input lines are used for write transfers. *(2)*
`write_master_av_address`	Out	`write_master` port Avalon-MM `address` bus. Specifies a byte address in the Avalon-MM address space.
`write_master_av_burstcount`	Out	`write_master` port Avalon-MM `burstcount` signal. Specifies the number of transfers in each burst.
`write_master_av_clock`	In	`write_master` port clock signal. The interface operates on the rising edge of the clock signal. *(1)*

Table 18–4. Frame Buffer Signals (Part 3 of 3)

Signal	Direction	Description
`write_master_av_reset`	In	`write_master` port reset signal. The interface resets asynchronously when you assert this signal. You must deassert this signal synchronously to the rising edge of the clock signal. (1)
`write_master_av_waitrequest`	In	`write_master` port Avalon-MM `waitrequest` signal. The system interconnect fabric asserts this signal to cause the master port to wait.
`write_master_av_write`	Out	`write_master` port Avalon-MM `write` signal. Asserted to indicate write requests from the master to the system interconnect fabric.
`write_master_av_writedata`	Out	`write_master` port Avalon-MM `writedata` bus. These output lines carry data for write transfers.
`writer_control_av_chipselect`	In	`writer_control` slave port Avalon-MM `chipselect` signal. The `writer_control` port ignores all other signals unless you assert this signal. (3)
`writer_control_av_readdata`	Out	`writer_control` slave port Avalon-MM `readdata` bus. These output lines are used for read transfers. (3)
`writer_control_av_write`	In	`writer_control` slave port Avalon-MM `write` signal. When you assert this signal, the `writer_control` port accepts new data from the `writedata` bus. (3)
`writer_control_av_writedata`	In	`writer_control` slave port Avalon-MM `writedata` bus. These input lines are used for write transfers. (3)

Notes to Table 18–4:

(1) Additional clock and reset signals are available when you turn on **Use separate clocks for the Avalon-MM master interfaces**.

(2) These ports are present only if the control interface for the reader component has been enabled.

(3) These ports are present only if the control interface for the writer component has been enabled

Control Register Maps

A run-time control can be attached either to the writer component or to the reader component of the Frame Buffer MegaCore function but not to both. The width of each register is 16 bits.

Table 18–5 describes the Frame Buffer MegaCore function control register map for the writer component.

Table 18–5. Frame Buffer Control Register Map for the Writer Component (Part 1 of 2)

Address	Register(s)	Description
0	`Control`	Bit 0 of this register is the `Go` bit. Setting this bit to 1 causes the Frame Buffer MegaCore function to stop the next time control information is read to start outputting data. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Frame Counter`	Read-only register updated at the end of each frame processed by the writer. The counter is incremented if the frame is not dropped and passed to the reader component.
3	`Drop Counter`	Read-only register updated at the end of each frame processed by the writer. The counter is incremented if the frame is dropped.

Table 18–5. Frame Buffer Control Register Map for the Writer Component (Part 2 of 2)

Address	Register(s)	Description
4	`Controlled Rate Conversion`	Bit 0 of this register determines whether dropping and repeating of frames or fields is tightly controlled by the specified input and output frame rates. Setting this bit to 0, switches off the controlled rate conversion and returns the triple-buffering algorithm to a free regime where dropping and repeating is only determined by the status of the spare buffer.
5	`Input Frame Rate`	Write-only register. A 16-bit integer value for the input frame rate. This register cannot be read.
6	`Output Frame Rate`	Write-only register. A 16-bit integer value for the output frame rate. This register cannot be read.

Table 18–6 describes the Frame Buffer MegaCore function control register map for the reader component.

Table 18–6. Frame Buffer Control Register Map for the Reader Component

Address	Register(s)	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the reader component to stop the next time control information is updated. While stopped, the Frame Buffer may continue to receive and drop frame at its input if frame dropping is enabled. Refer to “Avalon-MM Slave Interfaces” on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. Refer to “Avalon-MM Slave Interfaces” on page 3–17 for full details.
2	`Frame Counter`	Read-only register updated at the end of each frame processed by the reader. The counter is incremented if the frame is not repeated.
3	`Repeat Counter`	Read-only register updated at the end of each frame processed by the reader. The counter is incremented if the frame is about to be repeated.

19. Gamma Corrector MegaCore Function

Core Overview

The Gamma Corrector MegaCore function corrects video streams for the physical properties of display devices. For example, the brightness displayed by a cathode-ray tube monitor has a nonlinear response to the voltage of a video signal. You can configure the Gamma Corrector with a look-up table that models the nonlinear function to compensate for the non linearity. The look-up table can then transform the video data and give the best image on the display.

Functional Description

The Gamma Corrector MegaCore function provides a look-up table (LUT) accessed through an Avalon-MM slave port. The gamma values can be entered in the LUT by external hardware using this interface.

For information about using Avalon-MM slave interfaces for run-time control in the Video and Image Processing Suite, refer to "Avalon-MM Slave Interfaces" on page 3–17.

When dealing with image data with N bits per pixel per color plane, the address space of the Avalon-MM slave port spans $2^N + 2$ registers where each register is N bits wide.

Registers 2 to $2^N + 1$ are the look-up values for the gamma correction function. Image data with a value x will be mapped to whatever value is in the LUT at address $x + 2$.

Avalon-ST Video Protocol Parameters

The Gamma Corrector MegaCore function can process streams of pixel data of the types listed in Table 19–1.

Table 19–1. Gamma Corrector Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Read from control packets at run time.
Frame Height	Read from control packets at run time.
Interlaced / Progressive	Either.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	One, two or three channels in sequence or parallel. For example, if three channels in sequence is selected where α, β, and γ can be any color plane: α β γ

Stall Behavior and Error Recovery

In all parameterizations, the Gamma Corrector stalls only between frames and not between rows. It has no internal buffering aside from the registers of its processing pipeline so there are only a few clock cycles of latency.

Error Recovery

The Gamma Corrector MegaCore function processes video packets until an `endofpacket` signal is received. Non-image packets are propagated but the content of control packets is ignored. For this MegaCore function there is no such condition as an early or late endofpacket. Any mismatch of the `endofpacket` signal and the frame size is propagated unchanged to the next MegaCore function.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 19–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the Gamma Corrector MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 19–2. Gamma Corrector Latency

Mode	Latency
All modes	*O* (cycles)

The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 19–3 lists the Gamma Corrector MegaCore function parameters.

Table 19–3. Gamma Corrector Parameter Settings

Parameter	Value	Description
Bits per pixel per color plane	4–16, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color planes	1–3, Default = **3**	The number of color planes that are sent in sequence or parallel over one data connection.
Color plane transmission format	**Color planes in sequence**, Color planes in parallel	Specifies whether the specified number of color planes are transmitted in sequence or in parallel. For example, a value of 3 planes in sequence for R'G'B' R'G'B' R'G'B'.

You program the actual gamma corrected intensity values at run time using the Avalon-MM slave interface.

Signals

Table 19–4 lists the input and output signals for the Gamma Corrector MegaCore function.

Table 19–4. Gamma Corrector Signals

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.
`gamma_lut_av_address`	In	`gamma_lut` slave port Avalon-MM `address`. Specifies a word offset into the slave address space.
`gamma_lut_av_chipselect`	In	`gamma_lut` slave port Avalon-MM `chipselect` signal. The `gamma_lut` port ignores all other signals unless you assert this signal.
`gamma_lut_av_readdata`	Out	`gamma_lut` slave port Avalon-MM `readdata` bus. These output lines are used for read transfers.
`gamma_lut_av_write`	In	`gamma_lut` slave port Avalon-MM `write` signal. When you assert this signal, the `gamma_lut` port accepts new data from the `writedata` bus.
`gamma_lut_av_writedata`	In	`gamma_lut` slave port Avalon-MM `writedata` bus. These input lines are used for write transfers.

Control Register Maps

The Gamma Corrector can have up to three Avalon-MM slave interfaces. There is a separate slave interface for each channel in parallel. Table 19–5 to Table 19–7 describe the control register maps for these interfaces.

The control registers are read continuously during the operation of the MegaCore function, so making a change to part of the Gamma look-up table during the processing of a frame always has immediate effect. To synchronize changes to frame boundaries, follow the procedure which is described in "Avalon-MM Slave Interfaces" on page 3–17.

The width of each register in the Gamma Corrector control register map is always equal to the value of the *Bits per pixel per color plane* parameter selected in the parameter editor.

Table 19–5. Gamma Corrector Control Register Map: Interface 0

Address	Register Name	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the Gamma Corrector MegaCore function to stop the next time control information is read. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2 to 2^N+1 where *N* is the number of bits per color plane.	`Gamma Look-Up Table`	These registers contain a look-up table that is used to apply gamma correction to video data. An input intensity value of *x* is gamma corrected by replacing it with the contents of the (*x*+1)th entry in the look-up table. Changing the values of these registers has an immediate effect on the behavior of the MegaCore function. To ensure that gamma look-up values do not change during processing of a video frame, use the `Go` bit to stop the MegaCore function while the table is changed.

Table 19–6. Gamma Corrector Control Register Map: Interface 1

Address	Register Name	Description
0	`Unused`	This register is not used
1	`Unused`	This register is not used
2 to 2^N+1 where *N* is the number of bits per color plane.	`Gamma Look-Up Table`	These registers contain a look-up table that is used to apply gamma correction to video data. An input intensity value of *x* is gamma corrected by replacing it with the contents of the (*x*+1)th entry in the look-up table. Changing the values of these registers has an immediate effect on the behavior of the MegaCore function. To ensure that gamma look-up values do not change during processing of a video frame, use the `Go` bit in Interface 0 to stop the MegaCore function while the table is changed.

Table 19–7. Gamma Corrector Control Register Map: Interface 2

Address	Register Name	Description
0	`Unused`	This register is not used
1	`Unused`	This register is not used
2 to 2^N +1 where *N* is the number of bits per color plane.	`Gamma Look-Up Table`	These registers contain a look-up table that is used to apply gamma correction to video data. An input intensity value of *x* is gamma corrected by replacing it with the contents of the (*x*+1)th entry in the look-up table. Changing the values of these registers has an immediate effect on the behavior of the MegaCore function. To ensure that gamma look-up values do not change during processing of a video frame, use the `Go` bit in Interface 0 to stop the MegaCore function while the table is changed.

20. Interlacer MegaCore Function

Core Overview

The Interlacer MegaCore function converts progressive video to interlaced video by dropping half the lines of incoming progressive frames. You can configure the MegaCore function to discard or propagate already-interlaced input. You can also disable the interlacer at run time to propagate progressive frames unchanged.

Functional Description

The Interlacer MegaCore function generates an interlaced stream by dropping half the lines of each progressive input frame. The Interlacer drops odd and even lines in successive order to produce an alternating sequence of F0 and F1 fields. The output field rate is consequently equal to the input frame rate.

The Interlacer MegaCore function handles changing input resolutions by reading the content of Avalon-ST Video control packets. The Interlacer supports incoming streams where the height of the progressive input frames is an odd value. In such a case, the height of the output F0 fields are one line higher than the height of the output F1 fields.

When the input stream is already interlaced, the Interlacer either discards the incoming interlaced fields or propagates the fields without modification, based on the compile time parameters you specify. When you turn on **Run-time control**, you also can deactivate the Interlacer at run-time to prevent the interlacing and propagate a progressive video stream without modification.

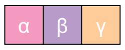
At start up or after a change of input resolution, the Interlacer begins the interlaced output stream by dropping odd lines to construct a F0 field or by dropping even lines to construct a F1 field, based on the compile time parameters you specify. Alternatively, when you turn on **Control packets override field selection** and the interlace nibble indicates that the progressive input previously went through a deinterlacer (0000 or 0001), the Interlacer produces a F0 field if the interlace nibble is 0000 and a F1 field if the interlace nibble is 0001. For more information, refer to Table 3–4 on page 3–8.

For most systems, turn off **Control packets override field selection** to guarantee the Interlacer function produces a valid interlaced video output stream where F0 and F1 fields alternate in regular succession.

Avalon-ST Video Protocol Parameters

The Interlacer MegaCore function can process streams of pixel data of the types listed in Table 20–1. The Interlacer does not support vertically subsampled video streams. For example, 4:2:2 is supported but 4:2:0 is not.

Table 20–1. Interlacer Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Run time controlled. (Maximum value specified in the parameter editor.)
Frame Height	Run time controlled. (Maximum value specified in the parameter editor.)
Interlaced / Progressive	Progressive, interlaced data is either discarded or propagated without change as selected in the parameter editor.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	One, two or three channels in sequence or in parallel as selected in the parameter editor. For example, for three channels in sequence where α, β, and γ can be any color plane: 

Stall Behavior and Error Recovery

While producing an interlaced output field, the Interlacer MegaCore function alternates between propagating and discarding a row from the input port. Consequently, the output port is inactive every other row. The delay from input to output is a few clock cycles when pixels are propagated.

Error Recovery

The Interlacer MegaCore function discards extra data when the `endofpacket` signal is received later than expected. When an early `endofpacket` signal is received, the current output field is interrupted as soon as possible and may be padded with a single undefined pixel.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 20–2 lists the approximate latency from the video data input to the video data output for typical usage modes of the Interlacer MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 20–2. Interlacer Latency

Mode	Latency
All modes	*O* (cycles)

The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 20–3 shows the Interlacer MegaCore function parameters.

Table 20–3. Interlacer Parameter Settings

Parameter	Value	Description
Maximum image width	32–2600, Default = **640**	Specifies the maximum frame width in pixels. The maximum frame width is the default width at start up.
Maximum image height	32–2600, Default = **480**	Specifies the maximum progressive frame height in pixels. The maximum frame height is the default progressive height at start up.
Bits per pixel per color plane	4–20, Default = **8**	Specifies the number of bits per color plane.
Number of color planes in sequence	1–**3**	Specifies the number of color planes that are sent in sequence over one data connection. For example, a value of 3 for R'G'B' R'G'B' R'G'B'.
Number of color planes in parallel	**1**–3	Specifies the number of color planes sent in parallel.
initial field	**F0**, F1	Specifies the type for the first field output after reset or after a resolution change.
Pass-through mode	On or **Off**	Turn on to propagate interlaced fields unchanged. Turn off to discard interlaced input.
Run-time control	On or **Off**	Turn on to enable run-time control.
Control packets override field selection	On or **Off**	Turn on when the content of the control packet specifies which lines to drop when converting a progressive frame into an interlaced field.

Signals

Table 20–4 shows the input and output signals for the Interlacer MegaCore function.

Table 20–4. Interlacer Signals (Part 1 of 2)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.

Table 20–4. Interlacer Signals (Part 2 of 2)

Signal	Direction	Description
`control_av_address`	In	`control` slave port Avalon-MM `address` bus. Specifies a word offset into the slave address space. *(1)*
`control_av_chipselect`	In	`control` slave port Avalon-MM `chipselect` signal. The `control` port ignores all other signals unless you assert this signal. *(1)*
`control_av_readdata`	Out	`control` slave port Avalon-MM `readdata` bus. These output lines are used for read transfers. *(1)*
`control_av_waitrequest`	Out	`control` slave port Avalon-MM `waitrequest` signal. *(1)*
`control_av_write`	In	`control` slave port Avalon-MM `write` signal. When you assert this signal, the `control` port accepts new data from the `writedata` bus. *(1)*
`control_av_writedata`	In	`control` slave port Avalon-MM `writedata` bus. These input lines are used for write transfers. *(1)*
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port should input data.
`dout_data`	Out	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

Note to Table 20–4:

(1) These ports are present only if you turn on **Pass-through mode**.

Control Register Maps

Table 20–5 describes the control register map for the Interlacer. The control interface is 8 bits wide but the Interlacer only uses bit 0 of each addressable register.

Table 20–5. Interlacer Control Register Map

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit. All other bits are unused. Setting this bit to 1 causes the Interlacer MegaCore function to pass data through without modification.

Table 20–5. Interlacer Control Register Map

Address	Register	Description
1	`Status`	Bit 0 of this register is the `Status` bit. All other bits are unused. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Progressive pass-through`	Setting bit 0 to 1 disables the Interlacer. When disabled, progressive inputs are propagated without modification.

21. Scaler MegaCore Function

Core Overview

The Scaler MegaCore function resizes video streams. The Scaler supports nearest-neighbor, bilinear, bicubic, and polyphase scaling algorithms. You can configure the Scaler to change resolutions or filter coefficients, or both, at run time using an Avalon-MM slave interface.

Functional Description

The Scaler MegaCore function provides a means to resize video streams. It supports nearest neighbor, bilinear, bicubic, and polyphase scaling algorithms.

The Scaler MegaCore function can be configured to change the input resolution using control packets. It can also be configured to change the output resolution and/or filter coefficients at run time using an Avalon-MM slave interface.

For information about using Avalon-MM slave interfaces for run-time control in the Video and Image Processing Suite, refer to "Avalon-MM Slave Interfaces" on page 3–17.

In the formal definitions of the scaling algorithms, the width and height of the input image are denoted w_{in} and h_{in} respectively. The width and height of the output image are denoted w_{out} and h_{out}. F is the function that returns an intensity value for a given point on the input image and O is the function that returns an intensity value on the output image.

Nearest Neighbor Algorithm

The nearest-neighbor algorithm that the scaler uses is the lowest quality method, and uses the fewest resources. Jagged edges may be visible in the output image as no blending takes place. However, this algorithm requires no DSP blocks, and uses fewer logic elements than the other methods.

Scaling down requires no on-chip memory; scaling up requires one line buffer of the same size as one line from the clipped input image, taking account of the number of color planes being processed. For example, up scaling an image which is 100 pixels wide and uses 8-bit data with 3 colors in sequence but is clipped at 80 pixels wide, requires $8 \times 3 \times 80 = 1920$ bits of memory. Similarly, if the 3 color planes are in parallel, the memory requirement is still 1920 bits.

For each output pixel, the nearest-neighbor method picks the value of the nearest input pixel to the correct input position. Formally, to find a value for an output pixel located at (i, j), the nearest-neighbor method picks the value of the nearest input pixel to $((i+0.5)\, w_{in}/w_{out},\, (j+0.5)\, h_{in}/h_{out})$.

The 0.5 values in this equation come from considering the coordinates of an image array to be on the lines of a 2D grid, but the pixels to be equally spaced between the grid lines that is, at half values.

This equation gives an answer relative to the mid-point of the input pixel. You must subtract 0.5 to translate from pixel positions to grid positions. However, this 0.5 would then be added again so that later truncation performs rounding to the nearest integer. Therefore no change is required. The calculation performed by the scaler is equivalent to the following integer calculation:

$$O(i, j) = F((2 \times w_{in} \times i + w_{in})/(2 \times w_{out}), (2 \times h_{in} \times j + h_{in})/(2 \times h_{out}))$$

Bilinear Algorithm

The bilinear algorithm that the scaler uses is higher quality and more expensive than the nearest-neighbor algorithm. The jaggedness of the nearest-neighbor method is smoothed out, but at the expense of losing some sharpness on edges.

Resource Usage

The bilinear algorithm uses four multipliers per channel in parallel. The size of each multiplier is either the sum of the horizontal and vertical fraction bits plus two, or the input data bit width, whichever is greater. For example, with four horizontal fraction bits, three vertical fraction bits, and eight-bit input data, the multipliers are nine-bit.

With the same configuration but 10-bit input data, the multipliers are 10-bit. The function uses two line buffers. As in nearest-neighbor mode, each of line buffers is the size of a clipped line from the input image. The logic area is more than the nearest-neighbor method.

Algorithmic Description

This section describes how the algorithmic operations of the bilinear scaler can be modeled using a frame-based method. This does not reflect the implementation, but allows the calculations to be presented concisely. To find a value for an output pixel located at (i, j), we first calculate the corresponding location on the input:

$$in_i = (i \times w_{in})/w_{out}$$

$$in_j = (j \times h_{in})/h_{out}$$

The integer solutions,($\lfloor in_i$û, $\lfloor in_j$û) to these equations provide the location of the top-left corner of the four input pixels to be summed. The differences between in_i, in_j and ($\lfloor in_i$û, $\lfloor in_j$û) are a measure of the error in how far the top-left input pixel is from the real-valued position that we want to read from. Call these errors err_i and err_j. The precision of each error variable is determined by the number of fraction bits chosen by the user, B_{fh} and B_{fv}, respectively.

Their values can be calculated as:

$$err_i = \frac{((i \times w_{in})\% w_{out}) \times 2^{B_{fh}}}{max(w_{in}, w_{out})}$$

$$err_j = \frac{((j \times h_{in})\% h_{out}) \times 2^{B_{fv}}}{max(h_{in}, h_{out})}$$

where % is the modulus operator and $max(a, b)$ is a function that returns the maximum of two values.

The sum is then weighted proportionally to these errors. Note that because the values are measured from the top-left pixel, the weights for this pixel are one minus the error.

That is, in fixed-point precision: $2^{B_{fh}} - err_i$ and $2^{B_{fv}} - err_j$

The sum is then:

$$O(i, j) \times 2^{B_{fv} + B_{fh}} = F(in_i, in_j) \times (2^{B_{fh}} - err_i) \times (2^{B_{fv}} - err_j) + F(in_i + 1, in_j) \times err_i \times (2^{B_{fv}} - err_j)$$
$$+ F(in_i, in_j + 1) \times (2^{B_{fh}} - err_i) \times err_j + F(in_i + 1, in_j + 1) \times err_i \times err_j$$

Polyphase and Bicubic Algorithms

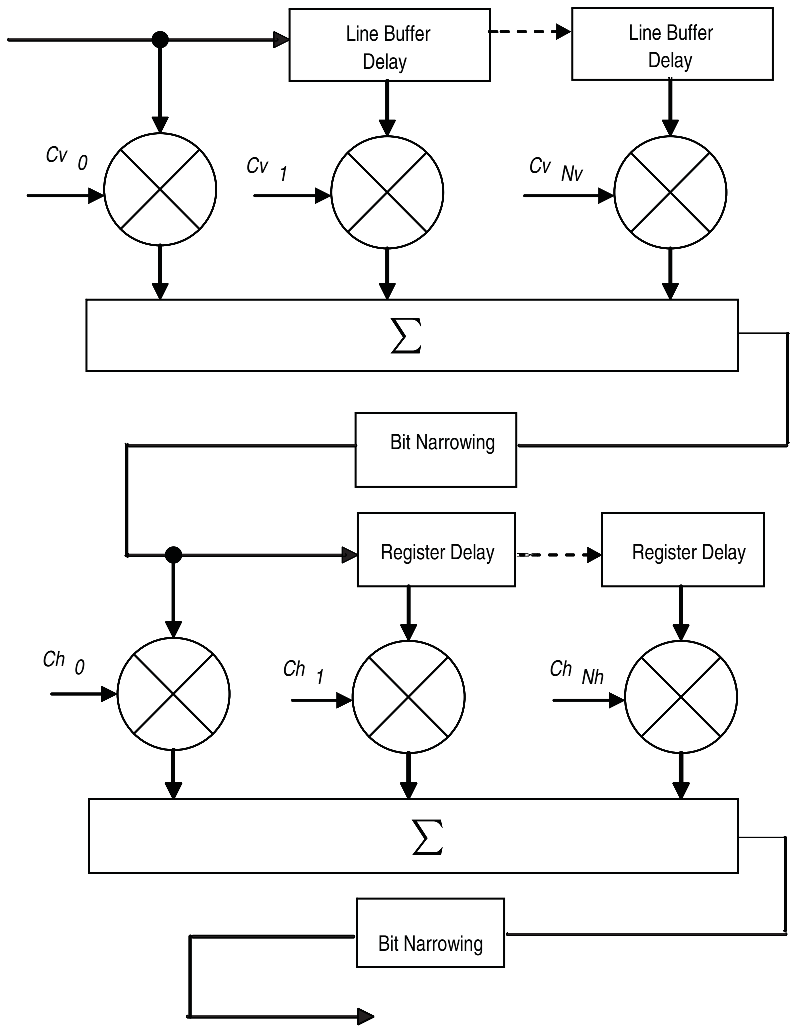
The polyphase and bicubic algorithms offer the best image quality, but use more resources than the other modes of the scaler. They allow up scaling to be performed in such a way as to preserve sharp edges, but without losing the smooth interpolation effect on graduated areas.

For down scaling, a long polyphase filter can reduce aliasing effects.

The bicubic and polyphase algorithms use different mathematics to derive their filter coefficients, but the implementation of the bicubic algorithm is just the polyphase algorithm with four vertical and four horizontal taps. In the following discussion, all comments relating to the polyphase algorithm are applicable to the bicubic algorithm assuming 4×4 taps.

Figure 21–1 shows the flow of data through an instance of the scaler in polyphase mode.this.

Figure 21–1. Polyphase Mode Scaler Block Diagram



Data from multiple lines of the input image are assembled into line buffers–one for each vertical tap. These data are then fed into parallel multipliers, before summation and possible loss of precision. The results are gathered into registers–one for each horizontal tap. These are again multiplied and summed before precision loss down to the output data bit width.

The progress of data through the taps (line buffer and register delays) and the coefficient values in the multiplication are controlled by logic that is not present in the diagram. Refer to "Algorithmic Description" on page 21–6.

Resource Usage

Consider an instance of the polyphase scaler with N_v vertical taps and N_h horizontal taps. B_{data} is the bit width of the data samples.

B_v is the bit width of the vertical coefficients and is derived from the user parameters for the vertical coefficients. It is equal to the sum of integer bits and fraction bits for the vertical coefficients, plus one if coefficients are signed.

B_h is defined similarly for horizontal coefficients. P_v and P_h are the user-defined number of vertical and horizontal phases for each coefficient set.

C_v is the number of vertical coefficient banks and C_h the number of horizontal coefficient banks.

The total number of multipliers is $N_v + N_h$ per channel in parallel. The width of each vertical multiplier is $max(B_{data},B_v)$. The width of each horizontal multiplier is the maximum of the horizontal coefficient width, B_h, and the bit width of the horizontal kernel, B_{kh}.

The bit width of the horizontal kernel determines the precision of the results of vertical filtering and is user-configurable. Refer to the **Number of bits to preserve between vertical and horizontal filtering** parameter in Table 19–3 on page 19–2.

The memory requirement is N_v line-buffers plus vertical and horizontal coefficient banks. As in the nearest-neighbor and bilinear methods, each line buffer is the same size as one line from the clipped input image.

The vertical coefficient banks are stored in memory that is B_v bits wide and $P_v \times N_v \times C_v$ words deep. The horizontal coefficient banks are stored in memory that is $B_h \times N_h$ bits wide and $P_h \times C_h$ words deep. For each coefficient type, the Quartus II software maps these appropriately to physical on-chip RAM or logic elements as constrained by the width and depth requirements.

If the horizontal and vertical coefficients are identical, they are stored in the horizontal memory (as defined above). If you turn on **Share horizontal /vertical coefficients** in the parameter editor this setting is forced even when the coefficients are loaded at run time.

Using multiple coefficient banks allows double-buffering, fast swapping, or direct writing to the Scaler’s coefficient memories. The coefficient bank to be read during video data processing and the bank to be written by the Avalon-MM interface are specified separately at run time, (refer to the control register map in Table 21–8 on page 21–15). This means that you can accomplish double-buffering by performing the following steps:

1. Select two memory banks at compile time.
2. At start-up run time, select a bank to write into (for example 0) and write the coefficients.
3. Set the chosen bank (0) to be the read bank for the Scaler, and start processing.
4. For subsequent changes, write to the unused bank (1) and swap the read and write banks between frames.

Choosing to have more memory banks allows for each bank to contain coefficients for a specific scaling ratio and for coefficient changes to be accomplished very quickly by changing the read bank. Alternatively, for memory-sensitive applications, use a single bank and coefficient writes have an immediate effect on data processing.

Algorithmic Description

This section describes how the algorithmic operations of the polyphase scaler can be modelled using a frame-based method. This description shows how the filter kernel is applied and how coefficients are loaded, but is not intended to indicate how the hardware of the scaler is designed.

The filtering part of the polyphase scaler works by passing a windowed sinc function over the input data. For up scaling, this function performs interpolation. For down scaling, it acts as a low-pass filter to remove high-frequency data that would cause aliasing in the smaller output image.

During the filtering process, the mid-point of the sinc function must be at the mid-point of the pixel to output. This is achieved be applying a phase shift to the filtering function.

If a polyphase filter has N_v vertical taps and N_h horizontal taps, the filter is a $N_v \times N_h$ square filter.

Counting the coordinate space of the filter from the top-left corner, (0, 0), the mid-point of the filter lies at $((N_v - 1)/2, (N_h - 1)/2)$. As in the bilinear case, to produce an output pixel at (i, j), the mid-point of the kernel is placed at ($\lfloor in_i$û, $\lfloor in_j$û) where in_i and in_j are calculated using the algorithmic description equations on page 21–2. The difference between the real and integer solutions to these equations determines the position of the filter function during scaling.

The filter function is positioned over the real solution by adjusting the function's phase:

$$phase_i = \frac{((i \times w_{in})\% w_{out}) \times P_h}{max(w_{in}, w_{out})}$$

$$phase_j = \frac{((j \times h_{in})\% h_{out}) \times P_v}{max(h_{in}, h_{out})}$$

The results of the vertical filtering are then found by taking the set of coefficients from $phase_j$ and applying them to each column in the square filter. Each of these N_h results is then divided down to fit in the number of bits chosen for the horizontal kernel. The horizontal kernel is applied to the coefficients from $phase_i$, to produce a single value. This value is then divided down to the output bit width before being written out as a result.

Choosing and Loading Coefficients

The filter coefficients, which the polyphase mode of the scaler uses, may be specified at compile time or at run time. At compile time, the coefficients can be either selected from a set of Lanczos-windowed sinc functions, or loaded from a comma-separated variable (CSV) file.

At run time they are specified by writing to the Avalon-MM slave control port (Table 21–8 on page 21–15).

When the coefficients are read at run time, they are checked once per frame and double-buffered so that they can be updated as the MegaCore function processes active data without causing corruption.

Figure 21–2 shows how a 2-lobe Lanczos-windowed sinc function (usually referred to as Lanczos 2) would be sampled for a 4-tap vertical filter.

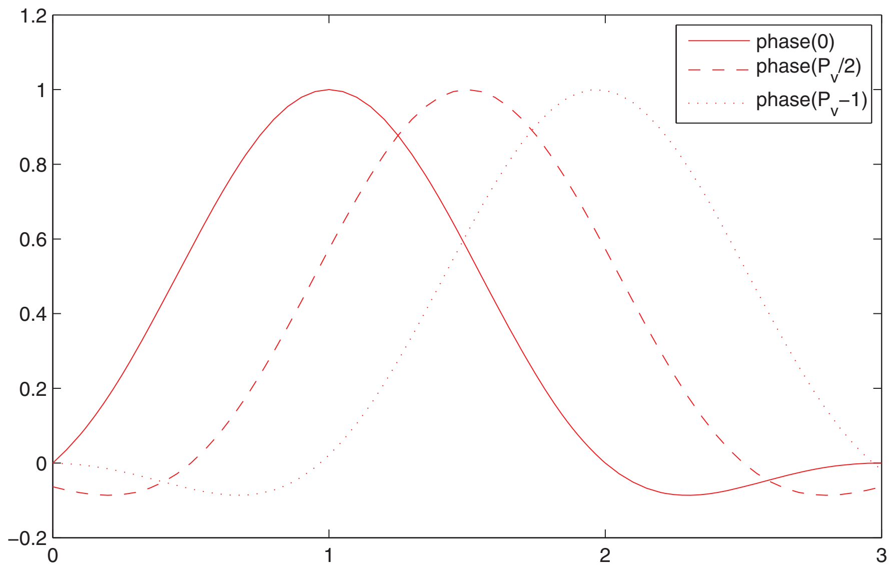
The two lobes refer to the number of times the function changes direction on each side of the central maxima, including the maxima itself.

The class of Lanczos N functions is defined as:

$$LanczosN(x) = \begin{cases} 1 & x = 0 \\ \dfrac{\sin(\pi x)}{\pi x}\dfrac{\sin(\pi x/N)}{\pi x/N} & x \neq 0 \wedge |x| < N \\ 0 & |x| \geq N \end{cases}$$

As can be seen in the figure, phase 0 centers the function over tap 1 on the x-axis. By the equation above, this is the central tap of the filter. Further phases move the mid-point of the function in $1/P_v$ increments towards tap 2. The filtering coefficients applied in a 4-tap scaler for a particular phase are samples of where the function with that phase crosses 0, 1, 2, 3 on the x-axis. The preset filtering functions are always spread over the number of taps given. For example, Lanczos 2 is defined over the range –2 to +2, but with 8 taps the coefficients are shifted and spread to cover 0 to 7.

Figure 21–2. Lanczos 2 Function at Various Phases



Compile-time custom coefficients are loaded from a CSV file. One CSV file is specified for vertical coefficients and one for horizontal coefficients. For N taps and P phases, the file must contain $N \times P$ values. The values must be listed as N taps in order for phase 0, N taps for phase 1, up to the Nth tap of the Pth phase. You are not required to present these values with each phase on a separate line.

The values must be pre-quantized in the range implied by the number of integer, fraction and sign bits specified in the parameter editor, and have their fraction part multiplied out. The sum of any two coefficients in the same phase must also be in the declared range. For example, if there is 1 integer bit, 7 fraction bits, and a sign bit, each value and the sum of any two values must be in the range [–256, 255] representing the range [-2, 1.9921875].

In summary, you can generate a set of coefficients for an N-tap, P-phase instance of the Scaler as follows:

1. Define a function, $f(x)$ over the domain $[0, N-1]$ under the assumption that $(N-1)/2$ is the mid-point of the filter.
2. For each tap t Î $\{0, 1, \ldots, N-1\}$ and for each phase $p \in \{0, 1/P, \ldots, (P-1/P)\}$, sample $f(t-p)$.
3. Quantize each sample. Ideally, the sum of the quantized values for all phases must be equal.
4. Either store these in a CSV file and copy them into the parameter editor, or load them at run time using the control interface.

Coefficients for the bicubic algorithm are calculated using Catmull-Rom splines to interpolate between values in tap 1 and tap 2.

Altera recommends that you use the Scaler II MegaCore function if your designs require custom coefficients.

For more information about the mathematics for Catmull-Rom splines refer to *E Catmull and R Rom. A class of local interpolating splines. Computer Aided Geometric Design, pages 317–326, 1974.*

The bicubic method does not use the preceding steps, but instead obtains weights for each of the four taps to sample a cubic function that runs between tap 1 and tap 2 at a position equal to the phase variable described previously. Consequently, the bicubic coefficients are good for up scaling, but not for down scaling.

If the coefficients are symmetric and provided at compile time, then only half the number of phases are stored. For N taps and P phases, an array, $C[P][N]$, of quantized coefficients is symmetric if:

for all p Œ $[1, P-1]$ and for all t Œ $[0, N-1]$, $C[p][t] = C[P-p][N-1-t]$

That is, phase 1 is phase $P-1$ with the taps in reverse order, phase 2 is phase $P-2$ reversed and so on. The predefined Lanczos and bicubic coefficient sets satisfy this property. Selecting **Symmetric** for a coefficients set on the **Coefficients** page in the parameter editor, forces the coefficients to be symmetric.

Recommended Parameters

In polyphase mode, you must choose parameters for the Scaler MegaCore function carefully to get the best image quality.

Incorrect parameters can cause a decrease in image quality even as the resource usage increases. The parameters which have the largest effect are the number of taps and the filter function chosen to provide the coefficients. The number of phases and number of bits of precision are less important to the image quality.

Table 21–1 summarizes some recommended values for parameters when using the Scaler in polyphase mode.

Table 21–1. Recommended Parameters for the Scaler MegaCore Function

Scaling Problem	Taps	Phases	Precision	Coefficients
Scaling up with any input/output resolution	4	16	Signed, 1 integer bit, 7 fraction bits	Lanczos-2, or Bicubic
Scaling down from *M* pixels to *N* pixels	$\frac{M \times 4}{N}$	16	Signed, 1 integer bit, 7 fraction bits	Lanczos-2
Scaling down from *M* pixels to *N* pixels (lower quality)	$\frac{M \times 2}{N}$	16	Signed, 1 integer bit, 7 fraction bits	Lanczos-1

Avalon-ST Video Protocol Parameters

The Scaler MegaCore function can process streams of pixel data of the types listed in Table 21–2.

Table 21–2. Scaler Avalon-ST Video Protocol Parameters

Parameter	Value
Frame Width	Maximum frame width is specified in the parameter editor, the actual value is read from control packets.
Frame Height	Maximum frame height is specified in the parameter editor, the actual value is read from control packets.
Interlaced / Progressive	Progressive.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Pattern	One, two or three channels in sequence or in parallel as selected in the parameter editor. For example, if three channels in sequence is selected where α, β and, γ can be any color plane: α β γ

Stall Behavior and Error Recovery

In the Scaler MegaCore function, the ratio of reads to writes is proportional to the scaling ratio and occurs on both a per-pixel and a per-line basis. The frequency of lines where reads and writes occur is proportional to the vertical scaling ratio. For example, scaling up vertically by a factor of 2 results in the input being stalled every other line for the length of time it takes to write one line of output; scaling down vertically by a factor of 2 results in the output being stalled every other line for the length of time it takes to read one line of input.

In a line that has both input and output active, the ratio of reads and writes is proportional to the horizontal scaling ratio. For example, scaling from 64×64 to 128×128 causes 128 lines of output, where only 64 of these lines have any reads in them. For each of these 64 lines, there are two writes to every read.

The internal latency of the Scaler depends on the scaling algorithm and whether any run time control is enabled. The scaling algorithm impacts stalling as follows:

- In nearest-neighbor mode, the delay from input to output is just a few clock cycles.

- In bilinear mode, a complete line of input is read into a buffer before any output is produced. At the end of a frame there are no reads as this buffer is drained. Exactly how many writes are possible during this time depends on the scaling ratio.
- In bicubic mode, three lines of input are read into line buffers before any output is ready. As with linear interpolation, there is a scaling ratio dependent time at the end of a frame where no reads are required as the buffers are drained.
- In polyphase mode with N_v vertical taps, $N_v - 1$ lines of input are read into line buffers before any output is ready. As with bilinear mode, there is a scaling ratio dependent time at the end of a frame where no reads are required as the buffers are drained.

Enabling run-time control of coefficients and/or resolutions affects stalling between frames:

- With no run-time control, there is only a few cycles of delay before the behavior described in the previous list begins.
- Enabling run-time control of resolutions in nearest-neighbor mode adds about 20 clock cycles of delay between frames. In other modes, it adds a maximum of 60 cycles delay.
- Enabling run-time control of coefficients adds a constant delay of about 20 cycles plus the total number of coefficients to be read. For example, 16 taps and 32 phases in each direction would add a delay of $20 + 2(16 \times 32) = 1024$ cycles.

Error Recovery

On receiving an early `endofpacket` signal, the Scaler stalls its input but continues writing data until it has sent an entire frame. If it does not receive an `endofpacket` signal at the end of a frame, the Scaler discards data until the end-of-packet is found.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 21–3 lists the approximate latency from the video data input to the video data output for typical usage modes of the Scaler MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 21–3. Scaler Latency

Mode	Latency
Scaling algorithm: Polyphase Number of vertical taps: *N*	(*N*–1) lines +*O* (cycles)

☞ The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 21–4 to Table 21–6 list the Scaler MegaCore function parameters.

Table 21–4. Scaler Parameter Settings Tab, Resolution Page

Parameter	Value	Description
Run-time control of image size	On or **Off**	Turn on to enable run-time control of the image size. When on, the input and output size parameters control the maximum values. When off, the Scaler does not respond to changes of resolution in control packets.
Input image width	32–2600, Default = **1,024**	Choose the required input width in pixels.
Input image height	32–2600, Default = **768**	Choose the required input height in pixels.
Output image width	32–2600, Default = **640**	Choose the required output width in pixels.
Output image height	32–2600, Default = **480**	Choose the required output height in pixels.
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color planes	1–3, Default = **3**	The number of color planes that are sent over one data connection. For example, a value of 3 for R'G'B' R'G'B' R'G'B' in serial.
Color planes transmission format	**Sequence**, Parallel	The transmission mode used for the specified number of color planes.

Table 21–5. Scaler Parameter Settings Tab, Algorithm and Precision Page (Part 1 of 2)

Parameter	Value	Description
Scaling Algorithm	Nearest Neighbor, Bilinear, Bicubic, **Polyphase**	Choose the scaling algorithm. For more information about these options, refer to "Nearest Neighbor Algorithm" on page 21–1, "Bilinear Algorithm" on page 21–2, and "Polyphase and Bicubic Algorithms" on page 21–3.
Number of vertical taps *(1)*	3–16, Default = **4**	Specify the number of vertical taps.
Number of vertical phases	2, 4, 8, **16**, 32, 64, 128, 256	Specify the number of vertical phases.
Number of horizontal taps *(1)*	3–16, Default = **4**	Specify the number of horizontal taps.
Number of horizontal phases	2, 4, 8, **16**, 32, 64, 128, 256	Specify the number of horizontal phases.

Table 21–5. Scaler Parameter Settings Tab, Algorithm and Precision Page (Part 2 of 2)

Parameter	Value	Description
Vertical Coefficient Precision: Signed	**On** or Off	Turn on if you want the fixed-point type that stores the vertical coefficients to have a sign bit.
Vertical Coefficient Precision: Integer bits:	0–15, Default = **1**	Specifies the number of integer bits for the fixed-point type used to store the vertical coefficients.
Vertical Coefficient Precision: Fraction bits:	3–15, Default = **7**	Specifies the number of fractional bits for the fixed point type used to store the vertical coefficients.
Number of bits to preserve between vertical and horizontal filtering *(1)*	3–32, Default = **9**	Specifies the number of bits to preserve between vertical and horizontal filtering.
Horizontal Coefficient Precision: Signed	**On** or Off	Turn on if you want the fixed-point type that stores the horizontal coefficients to have a sign bit.
Horizontal Coefficient Precision: Integer bits:	0–15, Default = **1**	Specifies the number of integer bits for the fixed-point type used to store the horizontal coefficients.
Horizontal Coefficient Precision: Fraction bits:	0–15, Default = **7**	Specifies the number of fractional bits for the fixed point type used to store the horizontal coefficients.

Note to Table 21–5:

(1) These parameters determine the number and size of the DSP blocks. For example, with four vertical and four horizontal taps and nine bits preserved between vertical and horizontal filtering, the scaler uses a total of eight 9×9 DSP blocks.

Table 21–6. Scaler Parameter Settings Tab, Coefficients Page (Part 1 of 2)

Parameter	Value	Description
Load coefficient data at run time	On or **Off**	Turn on to load the coefficient data at run time.
Share horizontal / vertical coefficients	On or **Off**	Turn on to map horizontal and vertical coefficients to the same memory. When on and **Load coefficient data at run time** is also on, writes to the vertical coefficients are ignored. (The choice of read bank remains independent for horizontal and vertical coefficients.)
Vertical Coefficient Data: Memory banks	1–6, Default = **2**	Choose the number of coefficient banks to enable double-buffering, fast coefficient swapping or direct writes.
Vertical Coefficient Data: Filter function	Lanczos 1–12, or Custom, Default = **Lanczos 2**	You can choose from 12 pre-defined Lanczos functions or use the coefficients saved in a custom coefficients file.
Vertical Coefficient Data: Custom coefficient file	User specified	When a custom function is selected, you can browse for a comma-separated value file containing custom coefficients. Key in the path for the file that contains these custom coefficients. Use the **Preview coefficients** button to view the current coefficients in a preview window.
Vertical Coefficient Data: Symmetric	On or **Off**	Turn on to save coefficient memory by using symmetric coefficients. When on and **Load coefficient data at run time** is also on, coefficient writes beyond phases 2 and 1 are ignored.
Horizontal Coefficient Data: Memory banks	1–6, Default = **2**	Choose the number of coefficient banks to enable double-buffering, fast coefficient swapping or direct writes.
Horizontal Coefficient Data: Filter function	Lanczos 1–12, or Custom, Default = **Lanczos 2**	You can choose from 12 pre-defined Lanczos functions or use the coefficients saved in a custom coefficients file.

Table 21–6. Scaler Parameter Settings Tab, Coefficients Page (Part 2 of 2)

Parameter	Value	Description
Horizontal Coefficient Data: Custom coefficient file	User specified	When a custom function is selected, you can browse for a comma-separated value file containing custom coefficients. Key in the path for the file that contains these custom coefficients. Use the **Preview coefficients** button to view the current coefficients in a preview window.
Horizontal Coefficient Data: Symmetric	On or **Off**	Turn on to save coefficient memory by using symmetric coefficients. When on and **Load coefficient data at run time** is also on, coefficient writes beyond phases 2 and 1 are ignored.

You can create custom coefficient data using third-party tools such as Microsoft Excel or the MATLAB Array Editor. To do so, click **Preview coefficients** under **Vertical Coefficient Data** and **Horizontal Coefficient Data**, copy the data from the predefined coefficient spreadsheet, edit the data with your third-party tool, delete the **Phase** column, and store the data in the Coeff columns as a **.csv** file. Then in the parameter editor, select **Custom** from the **Filter function** list, click **Browse**, load the **.csv** file, and click **Preview coefficients** to verify the data.

☞ When editing the data, each row of coefficients must sum to the same value. Refer to "Choosing and Loading Coefficients" on page 21–6.

Signals

Table 21–7 lists the input and output signals for the Scaler MegaCore function.

Table 21–7. Scaler Signals

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`control_av_address`	In	`control` slave port Avalon-MM `address` bus. Specifies a word offset into the slave address space. (1)
`control_av_chipselect`	In	`control` slave port Avalon-MM `chipselect` signal. The `control` port ignores all other signals unless you assert this signal. (1)
`control_av_readdata`	Out	`control` slave port Avalon-MM `readdata` bus. These output lines are used for read transfers. (1)
`control_av_waitrequest`	Out	`control` slave port Avalon-MM `waitrequest` signal. (1)
`control_av_write`	In	`control` slave port Avalon-MM `write` signal. When you assert this signal, the `control` port accepts new data from the `writedata` bus. (1)
`control_av_writedata`	In	`control` slave port Avalon-MM `writedata` bus. These input lines are used for write transfers. (1)
`din_data`	In	`din` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_endofpacket`	In	`din` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_ready`	Out	`din` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_startofpacket`	In	`din` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_valid`	In	`din` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

Note to Table 21–7:

(1) These ports are present only if you turn on **Run-time control of image size**.

Control Register Maps

Table 21–8 describes the Scaler MegaCore function control register map.

The Scaler reads the control data once at the start of each frame and buffers the data inside the MegaCore function. The registers may be safely updated during the processing of a frame, unless the frame is a coefficient bank.

The coefficient bank that is being read by the Scaler must not be written to unless the core is in a stopped state. To change the contents of the coefficient bank while the Scaler is in a running state, you must use multiple coefficient banks to allow an inactive bank to be changed without affecting the frame currently being processed.

The Scaler control interface allows the programming of 1 to 6 banks of coefficients and their phases. You can preprogram these coefficients and phases before any video is processed. The preprogramming is useful for rapid switching of scaling ratios as you only required to update 2 bank select registers plus any resolution changes.

If you require more than 6 bank configurations, then you can change the bank data externally. Using 2 banks allows one to be used by the Scaler while the other is being configured, and reduces the extra time required in-between frames to very few additional cycles.

Note that all Scaler registers are write-only except at address 1.

Table 21–8. Scaler Control Register Map (Part 1 of 2)

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0, causes the Scaler to stop the next time that control information is read. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. The Scaler MegaCore function sets this address to 0 between frames. It is set to 1 while the MegaCore function is processing data and cannot be stopped. Refer to "Avalon-MM Slave Interfaces" on page 3–17 for full details.
2	`Output Width`	The width of the output frames in pixels. (1)
3	`Output Height`	The height of the output frames in pixels. (1)
4	`Horizontal Coefficient Bank Write Address`	Specifies which memory bank horizontal coefficient writes from the Avalon-MM interface are made into.
5	`Horizontal Coefficient Bank Read Address`	Specifies which memory bank is used for horizontal coefficient reads during data processing.
6	`Vertical Coefficient Bank Write Address`	Specifies which memory bank vertical coefficient writes from the Avalon-MM interface are made into. (2)
7	`Vertical Coefficient Bank Read Address`	Specifies which memory bank is used for vertical coefficient reads during data processing
8 to 7+N_h	`Horizontal Tap Data`	Specifies values for the horizontal coefficients at a particular phase. Write these values first, then the `Horizontal Phase` to commit the write.
8+N_h	`Horizontal Phase`	Specifies which phase the `Horizontal Tap Data` applies to. Writing to this location, commits the writing of tap data. This write must be made even if the phase value does not change between successive sets of tap data.

Table 21–8. Scaler Control Register Map (Part 2 of 2)

Address	Register	Description
9+N_h to 8+N_h+N_h+N_v	`Vertical Tap Data`	Specifies values for the vertical coefficients at a particular phase. Write these values first, then the `Vertical Phase` to commit the write. (2)
9+N_h+N_v	`Vertical Phase`	Specifies which phase the `Vertical Tap Data` applies to. Writing to this location, commits the writing of tap data. This write must be made even if the phase value does not change between successive sets of tap data. (2)

Notes to Table 21–8:

(1) Value can be from 32 to the maximum specified in the parameter editor.

(2) If **Share horizontal/vertical coefficients** is selected in the parameter editor, this location is not used.

Table 21–9 lists an example of the sequence of writes to the horizontal coefficient data for an instance of the Scaler MegaCore function with four taps and eight phases.

Table 21–9. Example of Using the Scaler Control Registers

Address	Value	Purpose
8	0	Setting up Tap 0 for Phase 0.
9	128	Setting up Tap 1 for Phase 0.
10	0	Setting up Tap 2 for Phase 0.
11	0	Setting up Tap 3 for Phase 0.
12	0	Commit the writes to Phase 0.
8	–8	Setting up Tap 0 for Phase 1.
9	124	Setting up Tap 1 for Phase 1.
10	13	Setting up Tap 2 for Phase 1.
11	–1	Setting up Tap 3 for Phase 1.
12	1	Commit the writes to Phase 1.
...	...	...
8	–1	Setting up Tap 0 for Phase 7.
9	13	Setting up Tap 1 for Phase 7.
10	124	Setting up Tap 2 for Phase 7.
11	–8	Setting up Tap 3 for Phase 7.
12	7	Commit the writes to Phase 7.

22. Scaler II MegaCore Function

Core Overview

The Scaler II MegaCore function resizes video streams and offers improved functionality compared to the Scaler MegaCore Function. The Scaler II supports nearest neighbor, bilinear, bicubic and polyphase scaling algorithms, but extends beyond the functionality of the Scaler by supporting a simple edge-adaptive scaling algorithm and 4:2:2 sampled video data.

Functional Description

The features and functionality of the Scaler II MegaCore function are largely the same as those of the Scaler MegaCore function. However, the Scaler II potentially uses less area while delivering higher performance.

The Scaler II features the following different algorithmic changes:

- In bilinear mode, the Scaler II uses a different method to calculate the horizontal and vertical position error values, `err`$_i$ and `err`$_j$. In the Scaler II, these values are calculated using the following equation:

$$err_i = \frac{((i \times w_{in}) \% w_{out}) \times 2^{B_{fh}}}{w_{out}}$$

$$err_j = \frac{((j \times h_{in}) \% h_{out}) \times 2^{B_{fv}}}{h_{out}}$$

- In bicubic and polyphase modes, the Scaler II uses a different calculation method to calculate the horizontal and vertical phase values, `phase`$_i$ and `phase`$_j$. The Scaler II uses the following equation:

$$phase_i = \frac{((i \times w_{in}) \% w_{out}) \times 2^{P_h}}{w_{out}}$$

$$phase_j = \frac{((j \times h_{in}) \% h_{out}) \times 2^{P_h}}{h_{out}}$$

- The Scaler II offers an additional edge-adaptive scaling algorithm option. This algorithm is almost identical to the polyphase algorithm, with extensions to detect edges in the input video and use a different set of scaling coefficients to generate output pixels that lie on detected edges. You may set a value for the edge threshold either at compile or runtime. The edge threshold is the minimum difference between the values of neighboring pixels at which an edge is deemed to exist. The edge threshold value is specified per color plane; where internally the Scaler II multiplies the edge threshold by the number of color planes per pixel to create an internal edge threshold value. When the edge detection is done, the differences across all color planes are summed before comparing to the internal edge threshold.

Edge-Adaptive Scaling Algorithm

In the edge-adaptive mode, each bank of scaling coefficients inside the Scaler II consists of the following two full coefficient sets:

- A set for pixels that do not lie on the edge—allows you to select a coefficient set with a softer frequency response for the non-edge pixels
- A set for pixels that lie on the edges—allows you to select a coefficient set with a harsher frequency response for the edge pixels

These options potentially offer you a more favorable trade-off between blurring and ringing around edges. The current version of the Scaler II requires you to select the option to load coefficients at runtime to use the edge-adaptive mode; the core does not support fixed coefficients set at compile time. Altera recommends that you use Lanczos-2 coefficients for the non-edge coefficients and Lanczos-3 or Lanczos-4 for the edge coefficients.

The Scaler II offers the option to add a post-scaling edge-adaptive sharpening filter to the datapath. You may select this option in any scaling mode, with the exception of the nearest neighbor mode. Altera does not recommend you to use this option with nearest neighbor because it is of little or no benefit.

The sharpening filter is primarily for downscale, but you may also use it with upscale. You may enable or disable this functionality at runtime. The edge-adaptive sharpening filter attempts to detect blurred edge in the scaling video stream, and applies a sharpening filter where blurred edges are detected. The areas that are not detected as blurred edges are left unaltered. The blurred edge detection uses upper and lower blur thresholds, which you may alter at compile time or runtime. The sharpening engine detects a blurred edge where the difference between neighboring pixels falls between the upper and lower blur thresholds. As with the edge threshold for the edge-adaptive scaling, the upper and lower blur threshold values are specified per color plane.

Stall Behavior and Error Recovery

In the Scaler II MegaCore function, the ratio of reads to writes is proportional to the scaling ratio and occurs on both a per-pixel and a per-line basis. The frequency of lines where reads and writes occur is proportional to the vertical scaling ratio. For example, scaling up vertically by a factor of 2 results in the input being stalled every other line for the length of time it takes to write one line of output; scaling down vertically by a factor of 2 results in the output being stalled every other line for the length of time it takes to read one line of input.

In a line that has both input and output active, the ratio of reads and writes is proportional to the horizontal scaling ratio. For example, scaling from 64×64 to 128×128 causes 128 lines of output, where only 64 of these lines have any reads in them. For each of these 64 lines, there are two writes to every read.

The internal latency of the Scaler II depends on the scaling algorithm and whether any run time control is enabled. The scaling algorithm impacts stalling as follows:

- In bilinear mode, a complete line of input is read into a buffer before any output is produced. At the end of a frame there are no reads as this buffer is drained. Exactly how many writes are possible during this time depends on the scaling ratio.

- In polyphase mode with N_v vertical taps, $N_v - 1$ lines of input are read into line buffers before any output is ready. As with bilinear mode, there is a scaling ratio dependent time at the end of a frame where no reads are required as the buffers are drained.

Enabling run-time control of resolutions affects stalling between frames:

- With no run-time control, there are about 10 cycles of delay before the behavior described in the previous list begins, and about 20 cycles of further stalling between each output line.
- Enabling run-time control of resolutions adds about 25 cycles of delay between frames.

Error Recovery

On receiving an early `endofpacket` signal, the Scaler stalls its input but continues writing data until it has sent an entire frame. If it does not receive an `endofpacket` signal at the end of a frame, the Scaler discards data until the end-of-packet is found.

On receiving an early `endofpacket` signal at the end of an input line, the Scaler II stalls its input but continues writing data until it has sent on further output line. On receiving an early `endofpacket` signal part way through an input line, the Scaler II stalls its input for as long as it would take for the open input line to complete, completing any output line that may accompany that input line. It then continues to stall the input, and writes one further output line. If it does not receive an `endofpacket` signal at the end of a frame, the Scaler II discards data until the end-of-packet is found.

☞ For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 22–1 lists the approximate latency from the video data input to the video data output for typical usage modes of the Scaler II MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles O (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 22–1. Scaler II Latency

Mode	Latency
Scaling algorithm: Polyphase Number of vertical taps: N	(N–1) lines +O (cycles)

The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

This section describe the parameters for the Scaler II MegaCore function.

The Scaler II MegaCore function currently does not offer the nearest neighbor and bicubic scaling modes, and the symmetric coefficient functionality is disabled for both horizontal and vertical coefficients. You must specify the coefficients at run time for version 10.1. However, you can achieve bicubic scaling by setting the horizontal and vertical taps to 4 and load bicubic coefficients through the Avalon-MM control port.

Table 22–2 lists the Scaler II MegaCore function parameters.

Table 22–2. Scaler II Parameter Settings Tab

Parameter	Value	Description
Video Data Format		
Bits per symbol	4–20, Default = **10**	Choose the number of bits per color plane.
Symbols in parallel	1–4, Default = **2**	Choose the number of color planes sent in parallel.
Symbols in sequence	1–4, Default = **1**	Choose the number of color planes sent in sequence.
Enable run-time control of input/output frame size	**On** or Off	Turn on to enable run-time control of the image size. If you do not turn on this option, the Scaler II IP core does not respond to the resolution changes in control packets; and the input and output resolutions are set to the maximum values you specify.
Maximum input frame width	32–2600, Default = **1920**	Choose the maximum width for the input frames (in pixels).
Maximum input frame height	32–2600, Default = **1080**	Choose the maximum height for the input frames (in pixels).
Maximum output frame width	32–2600, Default = **1920**	Choose the maximum width for the output frames (in pixels).
Maximum output frame height	32–2600, Default = **1080**	Choose the maximum height for the output frames (in pixels).
4:2:2 video data	**On** or Off	Turn on to use the 4:2:2 data format. Turn off to use the 4:4:4 video format.
No blanking in video	On or **Off**	Turn on if the input video does not contain vertical blanking at its point of conversion to the Avalon-ST video protocol.
Algorithm Settings		
Scaling algorithm	Nearest Neighbor, Bilinear, Bicubic, **Polyphase**, or Edge Adaptive	Choose the scaling algorithm. For more information about these options, refer to , "Nearest Neighbor Algorithm" on page 21–1, "Bilinear Algorithm" on page 21–2, "Polyphase and Bicubic Algorithms" on page 21–3, and "Edge-Adaptive Scaling Algorithm" on page 22–2.
Enable post scaling sharpen	On or **Off**	Turn on to include a post-scaling edge-adaptive sharpening filter.

Table 22–2. Scaler II Parameter Settings Tab

Parameter	Value	Description
Always downscale or pass-through	On or **Off**	Turn on if you want the output frame height to be less than or equal to the input frame height, which reduces the size of the line buffer by one line. If the input height becomes less than the specified output height for any reason, the output video is undefined but it must still have the correct dimensions and the core must not crash.
Share horizontal and vertical coefficients	On or **Off**	Turn on to force the bicubic and polyphase algorithms to share the same horizontal and vertical scaling coefficient data.
Vertical filter taps	4–64, Default = **8**	Choose the number of vertical filter taps for the bicubic and polyphase algorithms.
Vertical filter phases	1–256, Default = **16**	Choose the number of vertical filter phases for the bicubic and polyphase algorithms.
Horizontal filter taps	4–64, Default = **8**	Choose the number of horizontal filter taps for the bicubic and polyphase algorithms.
Horizontal filter phases	1–256, Default = **16**	Choose the number of horizontal filter phases for the bicubic and polyphase algorithms.
Default edge threshold	0 to $2^{\text{bits per symbol}}$–1, Default = **7**	Choose the default value for the edge-adaptive scaling mode. This value will be the fixed edge threshold value if you do not turn on **Enable run-time control of input/output frame size** and edge/blur thresholds.
Default upper blur limit	0 to $2^{\text{bits per symbol}}$–1, Default = **15**	Choose the default value for the blurred-edge upper threshold in edge-adaptive sharpening. This value will be the fixed threshold value if you do not turn on **Enable run-time control of input/output frame size** and edge/blur thresholds.
Default lower blur limit	0 to $2^{\text{bits per symbol}}$–1, Default = **0**	Choose the default value for the blurred-edge lower threshold in edge-adaptive sharpening. This value will be the fixed threshold value if you do not turn on **Enable run-time control of input/output frame size** and edge/blur thresholds.
Precision Settings		
Vertical coefficients signed	**On** or Off	Turn on to force the algorithm to use signed vertical coefficient data.
Vertical coefficient integer bits	0–32, Default = **1**	Choose the number of integer bits for each vertical coefficient.
Vertical coefficient fraction bits	1–32, Default = **7**	Choose the number of fraction bits for each vertical coefficient.
Horizontal coefficients signed	**On** or Off	Turn on to force the algorithm to use signed horizontal coefficient data.
Horizontal coefficient integer bits	0–32, Default = **1**	Choose the number of integer bits for each horizontal coefficient.
Horizontal coefficient fraction bits	1–32, Default = **7**	Choose the number of fraction bits for each horizontal coefficient.
Number of fractional bits to preserve between horizontal and vertical filtering	Default = **0**	Choose the number of fractional bits you want to preserve between the horizontal and vertical filtering.

Table 22–2. Scaler II Parameter Settings Tab

Parameter	Value	Description
Coefficient Settings		
Load scaler coefficients at run time	On or **Off**	Turn on to update the scaler coefficient data at run time.
Vertical coefficient banks	1–32, Default = **1**	Choose the number of banks of vertical filter coefficients for polyphase algorithms.
Vertical coefficient function	Lanczos 2 and 3, or Custom, Default = **Lanczos 2**	Choose the function used to generate the vertical scaling coefficients. Choose either one for the pre-defined Lanczos functions or choose **Custom** to use the coefficients saved in a custom coefficients file.
Vertical coefficients file	User specified	When a custom function is selected, you can browse for a comma-separated value file containing custom coefficients. Key in the path for the file that contains these custom coefficients.
Horizontal coefficient banks	1–32, Default = **1**	Choose the number of banks of horizontal filter coefficients for polyphase algorithms.
Horizontal coefficient function	Lanczos 2 and 3, or Custom, Default = **Lanczos 2**	Choose the function used to generate the horizontal scaling coefficients. Choose either one for the pre-defined Lanczos functions or choose **Custom** to use the coefficients saved in a custom coefficients file.
Horizontal coefficients file	User specified	When a custom function is selected, you can browse for a comma-separated value file containing custom coefficients. Key in the path for the file that contains these custom coefficients.
Pipelining		
Add extra pipelining registers	On or **Off**	Turn on to add extra pipeline stage registers to the data path. You must to turn on this option to achieve a frequency of 150 MHz for Cyclone III or Cyclone IV devices, and frequencies above 250 MHz for Arria II, Stratix IV, and Stratix V devices.

Signals

Table 22–3 lists the input and output signals for the Scaler II MegaCore function.

Table 22–3. Scaler II Signals (Part 1 of 2)

Signal	Direction	Description
`main_clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `main_clock` signal.
`main_reset`	In	The MegaCore function asynchronously resets when you assert `main_reset`. You must deassert `main_reset` synchronously to the rising edge of the `main_clock_clk` signal.
`control_address`	In	`control` slave port Avalon-MM `address` bus. Specifies a word offset into the slave address space. (1)

Table 22–3. Scaler II Signals (Part 2 of 2)

Signal	Direction	Description
control_byteenable	In	control slave port Avalon-MM byteenable bus. Enables specific byte lane or lanes during transfers. Each bit in byteenable corresponds to a byte in writedata and readdata. During writes, byteenable specifies which bytes are being written to; other bytes are ignored by the slave. During reads, byteenable indicates which bytes the master is reading. Slaves that simply return readdata with no side effects are free to ignore byteenable during reads. (1)
control_read	In	control slave port Avalon-MM read signal. When you assert this signal, the control port outputs new data at readdata. (1)
control_readdata	Out	control slave port Avalon-MM readdata bus. Output lines for read transfers. (1)
control_readdatavalid	Out	control slave port Avalon-MM control_readdata bus. When you assert this signal, the control port outputs new data at control_readdata.
control_waitrequest	Out	control slave port Avalon-MM waitrequest signal. (1)
control_write	In	control slave port Avalon-MM write signal. When you assert this signal, the control port accepts new data from the writedata bus. (1)
control_writedata	In	control slave port Avalon-MM writedata bus. Input lines for write transfers. (1)
din_data	In	din port Avalon-ST data bus. This bus enables the transfer of pixel data into the MegaCore function.
din_endofpacket	In	din port Avalon-ST endofpacket signal. This signal marks the end of an Avalon-ST packet.
din_ready	Out	din port Avalon-ST ready signal. This signal indicates when the MegaCore function is ready to receive data.
din_startofpacket	In	din port Avalon-ST startofpacket signal. This signal marks the start of an Avalon-ST packet.
din_valid	In	din port Avalon-ST valid signal. This signal identifies the cycles when the port must input data.
dout_data	Out	dout port Avalon-ST data bus. This bus enables the transfer of pixel data out of the MegaCore function.
dout_endofpacket	Out	dout port Avalon-ST endofpacket signal. This signal marks the end of an Avalon-ST packet.
dout_ready	In	dout port Avalon-ST ready signal. The downstream device asserts this signal when it is able to receive data.
dout_startofpacket	Out	dout port Avalon-ST startofpacket signal. This signal marks the start of an Avalon-ST packet.
dout_valid	Out	dout port Avalon-ST valid signal. This signal is asserted when the MegaCore function outputs data.

Note to Table 22–3:

(1) These ports are not present if you turn off **Enable run-time control of input/output frame size** and select **Bilinear** for **Scaling algorithm** in the parameter editor.

Control Register Maps

Table 22–4 describes the Scaler II MegaCore function control register map. The run-time control register map for the Scaler II MegaCore function is altered and does not match the register map of the Scaler MegaCore function.

The N_{taps} is the number of horizontal or vertical filter taps, whichever is larger.

The Scaler II reads the control data once at the start of each frame and buffers the data inside the MegaCore function. The registers may be safely updated during the processing of a frame, unless the frame is a coefficient bank.

The coefficient bank that is being read by the Scaler II must not be written to unless the core is in a stopped state. To change the contents of the coefficient bank while the Scaler II is in a running state, you must use multiple coefficient banks to allow an inactive bank to be changed without affecting the frame currently being processed.

The Scaler II allows for dynamic bus sizing on the slave interface. The slave interface includes a 4-bit byte enable signal, and the width of the data on the slave interface is 32 bits.

For more information about dynamic bus sizing, refer to the "Avalon-MM Slave Addressing" section in *Avalon Interface Specifications*.

Table 22–4. Scaler II Control Register Map (Part 1 of 2)

Address	Register	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0, causes the Scaler II to stop the next time that control information is read.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. When this bit is set to 0, the Scaler II sets this address to 0 between frames. It is set to 1 while the MegaCore function is processing data and cannot be stopped.
2	`Interrupt`	This bit is not used because the Scaler II does not generate any interrupts.
3	`Output Width`	The width of the output frames in pixels.
4	`Output Height`	The height of the output frames in pixels.
5	`Edge Threshold`	Specifies the minimum difference between neighboring pixels beyond which the edge-adaptive algorithm switches to using the edge coefficient set. To get the threshold used internally, this value is multiplied by the number of color planes per pixel.
6	`Lower Blur Threshold`	Specifies the minimum difference between two pixels for a blurred edge to be detected between the pixels during post scaling edge-adaptive sharpening. To get the threshold used internally, this value is multiplied by the number of color planes per pixel.
7	`Upper Blur Threshold`	Specifies the maximum difference between two pixels for a blurred edge to be detected between the pixels during post scaling edge-adaptive sharpening. To get the threshold used internally, this value is multiplied by the number of color planes per pixel.
8	`Horizontal Coefficient Write Bank`	Specifies which memory bank horizontal coefficient writes from the Avalon-MM interface are made into.

Table 22–4. Scaler II Control Register Map (Part 2 of 2)

Address	Register	Description
9	`Horizontal Coefficient Read Bank`	Specifies which memory bank is used for horizontal coefficient reads during data processing.
10	`Vertical Coefficient Write Bank`	Specifies which memory bank vertical coefficient writes from the Avalon-MM interface are made into.
11	`Vertical Coefficient Read Bank`	Specifies which memory bank is used for vertical coefficient reads during data processing.
12	`Horizontal Phase`	Specifies which horizontal phase the coefficient tap data in the `Coefficient Data` register applies to. Writing to this location, commits the writing of coefficient tap data. This write must be made even if the phase value does not change between successive sets of coefficient tap data. To commit to an edge phase, write the horizontal phase number +32768. For example, set bit 15 of the register to 1.
13	`Vertical Phase`	Specifies which vertical phase the coefficient tap data in the `Coefficient Data` register applies to. Writing to this location, commits the writing of coefficient tap data. This write must be made even if the phase value does not change between successive sets of coefficient tap data. To commit to an edge phase, write the vertical phase number +32768. For example, set bit 15 of the register to 1.
14 to 14+N_{taps}	`Coefficient Data`	Specifies values for the coefficients at each tap of a particular horizontal or vertical phase. Write these values first, then the `Horizontal Phase` or `Vertical Phase`, to commit the write.

23. Switch MegaCore Function

Core Overview

The Switch MegaCore function allows the connection of up to twelve input video streams to twelve output video streams and the run-time reconfiguration of those connections via a control input.

Functional Description

The Switch MegaCore function allows the connection of up to twelve input video streams to twelve output video streams. For example, 1 to 2, 4 to 1, 6 to 6, and so on. The connections can be reconfigured at run time via a control input. Figure 23–1 shows an example 3 to 2 Switch with the possible connections for each input and output.

The Switch MegaCore function does not support duplication or combining of streams. (If these functions are required, use the Color Plane Sequencer MegaCore function.) Each output from the Switch can be driven by only one input and each input to the Switch can drive only one output. Any input can be disabled that is not routed to an output, which stalls the input by pulling it's `ready` signal low.

The routing configuration of the Switch MegaCore function is run time configurable through the use of an Avalon-MM slave control port. The registers of the control port can be written to at anytime but the Switch loads the new values only when it is stopped. Stopping the Switch MegaCore function causes all the input streams to be synchronized at the end of an Avalon-ST Video image packet.

There are two ways to load a new configuration:

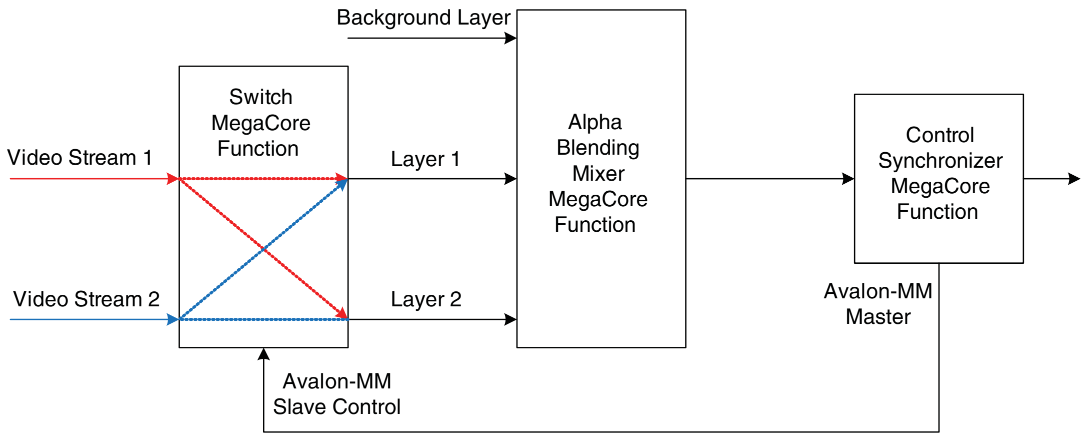
- Writing a 0 to the `Go` register, waiting for the `Status` register to read 0 and then writing a 1 to the `Go` register.
- Writing a 1 to the `Output Switch` register performs the same sequence but without the need for user intervention. This is the recommended way to load a new configuration.

Mixer Layer Switching

You can use the Switch MegaCore function in conjunction with the Alpha Blending Mixer MegaCore function and Control Synchronizer MegaCore function to perform run time configurable layer switching in the Alpha Blending Mixer. Layer switching is the ability to change the layer that a video stream is on, moving it in front of or behind the other video streams being mixed.

Figure 23–1 shows the system configuration used to achieve this.

Figure 23–1. Example of a layer Switching System



The Control Synchronizer MegaCore function ensures that the switch of the video streams is performed at a safe place in the streams. Performing the switch when the Alpha Blending Mixer MegaCore function is outputting the start of an image packet, ensures that the video streams entering the Switch MegaCore function are all on the same frame. They can then be switched on the next image end-of-packet without causing a deadlock situation between the Switch and Alpha Blending Mixer.

The following sequence shows an example for layer switching:

1. Switch MegaCore function—Write to the `DoutN Output Control` registers setting up the outputs. For example:
   a. Write 1 to address 3
   b. Write 2 to address 4
2. Switch MegaCore function—Enable the function by writing 1 to address 0
3. Switch MegaCore function—Write to the `DoutN Output Control` registers to switch the outputs. For example:
   a. Write 2 to address 3
   b. Write 1 to address 4
4. Control Synchronizer MegaCore function—Set up the Control Synchronizer to write a 1 to the Switch MegaCore function's `Output Switch` register on the next start of an image packet.

Stall Behavior and Error Recovery

The Switch MegaCore function only stalls its inputs when performing an output switch. Before switching its outputs it synchronize all its inputs and during this synchronization the inputs may be stalled.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

Table 23–1 lists the approximate latency from the video data input to the video data output for typical usage modes of the Switch MegaCore function. You can use this table to predict the approximate latency between the input and the output of your video processing pipeline.

The latency is described using one or more of the following measures:

- the number of progressive frames
- the number of interlaced fields
- the number of lines when less than a field of latency
- a small number of cycles *O* (cycles)

The latency is measured with the assumption that the MegaCore function is not being stalled by other functions on the data path (the output ready signal is high).

Table 23–1. Switch Latency

Mode	Latency
All modes	2 cycles

The latency associated with the initial buffering phase, when a MegaCore function first receives video data, is not included.

Parameter Settings

Table 23–2 lists the Switch MegaCore function parameters.

Table 23–2. Switch Parameter Settings

Parameter	Value	Description
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Number of color planes	1–3, Default = **3**	Choose the number of color planes.
Color planes are in parallel	**On** or Off	Turn on to set colors planes in parallel, turn off to set colors planes in sequence.
Number of input ports	1–12, Default = **2**	Number of input ports (`din` and `alpha_in`).
Number of output ports	1–12, Default = **2**	Number of output ports (`dout` and `alpha_out`).
Alpha enabled	On or **Off**	Turn on to enable the alpha ports.
Bits per pixel representing the alpha coefficient	2, 4, or **8**	Choose the number of bits used to represent the alpha coefficient.

Signals

Table 23–3 lists the input and output signals for the Switch MegaCore function.

Table 23–3. Switch Signals (Part 1 of 2)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`alpha_in_N_data`	In	`alpha_in_N` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function. (1)
`alpha_in_N_endofpacket`	In	`alpha_in_N` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet. (1)
`alpha_in_N_ready`	Out	`alpha_in_N` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data. (1)
`alpha_in_N_startofpacket`	In	`alpha_in_N` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet. (1)
`alpha_in_N_valid`	In	`alpha_in_N` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data. (1)
`alpha_out_N_data`	Out	`alpha_out_N` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function. (1)
`alpha_out_N_endofpacket`	Out	`alpha_out_N` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet. (1)
`alpha_out_N_ready`	In	`alpha_out_N` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data. (1)
`alpha_out_N_startofpacket`	Out	`alpha_out_N` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet. (1)
`alpha_out_N_valid`	Out	`alpha_out_N` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data. (1)
`din_N_data`	In	`din_N` port Avalon-ST `data` bus. This bus enables the transfer of pixel data into the MegaCore function.
`din_N_endofpacket`	In	`din_N` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`din_N_ready`	Out	`din_N` port Avalon-ST `ready` signal. This signal indicates when the MegaCore function is ready to receive data.
`din_N_startofpacket`	In	`din_N` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`din_N_valid`	In	`din_N` port Avalon-ST `valid` signal. This signal identifies the cycles when the port must input data.
`dout_N_data`	Out	`dout_N` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_N_endofpacket`	Out	`dout_N` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_N_ready`	In	`dout_N` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.

Table 23–3. Switch Signals (Part 2 of 2)

Signal	Direction	Description
`dout_N_startofpacket`	Out	`dout_N` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.
`dout_N_valid`	Out	`dout_N` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

Note to Table 23–3:

(1) These ports are present only when Alpha Enabled is turned on in the parameter editor.

Control Register Map

Table 23–4 describes the Switch MegaCore function control register map.

Table 23–4. Switch Control Register Map

Address	Register(s)	Description
0	`Control`	Writing a 1 to bit 0, starts the MegaCore function, writing a 0 to bit 0 stops the MegaCore function.
1	`Status`	Reading a 1 from bit 0, indicates that the MegaCore function is running (video is flowing through it), reading a 0 indicates that it is stopped.
2	`Output Switch`	Writing a 1 to bit 0, indicates that the video output streams must be synchronized and then the new values in the output control registers must be loaded.
3	`Dout0 Output Control`	A one-hot value that selects which video input stream must propagate to this output. For example, for a 3 input switch: ■ 3'b000 = no output ■ 3'b001 = din_0 ■ 3'b010 = din_1 ■ 3'b100 = din_2
4	`Dout1 Output Control`	As `Dout0 Output Control` but for output `dout1`.
...	...	...
15	`Dout12 Output Control`	As `Dout0 Output Control` but for output `dout12`.

24. Test Pattern Generator MegaCore Function

Core Overview

The Test Pattern Generator MegaCore function generates a video stream that displays either still color bars for use as a test pattern or a constant color for use as a uniform background. You can use this MegaCore function during the design cycle to validate a video system without the possible throughput issues associated with a real video input.

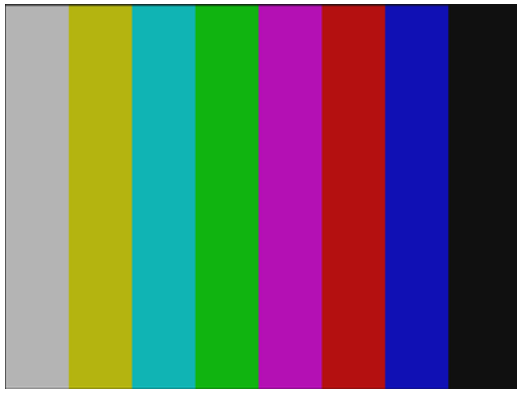
Functional Description

The Test Pattern Generator MegaCore function can be used to produce a video stream compliant with the Avalon-ST Video protocol that feeds a video system during its design cycle. The Test Pattern Generator MegaCore function produces data on request and consequently permits easier debugging of a video data path without the risks of overflow or misconfiguration associated with the use of the Clocked Video Input MegaCore function or of a custom component using a genuine video input.

Test Pattern

The Test Pattern Generator MegaCore function can generate either a uniform image using a constant color specified by the user at compile time or a set of predefined color bars. Both patterns are delimited by a black rectangular border. The color bar pattern (Figure 24–1) is a still image composed with a set of eight vertical color bars of 75% intensity (white, yellow, cyan, green, magenta, red, blue, black).

Figure 24–1. Color Bar Pattern



The sequence runs through the eight possible on/off combinations of the three color components of the RGB color space starting with a 75% amplitude white. Green is on for the first four bars and off for the last four bars, red cycles on and off every two bars, and blue cycles on and off every bar.

The actual numerical values are given in Table 24–1 (assuming 8 bits per color samples). If the output is requested in a different number of bits per color sample these values are converted by truncation or promotion.

Table 24–1. Test Pattern Color Values

Color	R'G'B'	Y'CbCr
White/Grey	(180,180,180)	(180,128,128)
Yellow	(180,180,16)	(162,44,142)
Cyan	(16,180,180)	(131,156,44)
Green	(16,180,16)	(112,72,58)
Magenta	(180,16,180)	(84,184,198)
Red	(180,16,16)	(65,100,212)
Blue	(16,16,180)	(35,212,114)
Black	(16,16,16)	(16,128,128)

The choice of a specific resolution and subsampling for the output leads to natural constraints on the test pattern. If the format has a horizontal subsampling period of two for the Cb and Cr components when the output is in the Y'CbCr color space, the black borders at the left and right are two pixels wide. Similarly, the top and bottom borders are two pixels wide when the output is vertically subsampled.

The width and the horizontal subsampling might also have an effect on the width of each color bar. When the output is horizontally subsampled, the pixel-width of each color bar is a multiple of two. When the width of the image (excluding the left and right borders) cannot be exactly divided by eight, then the last black bar is larger than the others. For example, when producing a 640×480 frame in the Y'CbCr color space with 4:2:2 subsampling, the left and right black borders are two pixels wide each, the seven initial color bars are 78 pixels wide ((640–4)/8 truncated down to the nearest multiple of 2) and the final black color bar is 90 pixels wide (640–7×78–4).

Generation of Avalon-ST Video Control Packets and Run-Time Control

The Test Pattern Generator MegaCore function outputs a valid Avalon-ST Video control packet before each image data packet it generates, whether it is a progressive frame or an interlaced field. When the output is interlaced, the Test Pattern Generator MegaCore function produces a sequence of pairs of field, starting with F0 if the output is F1 synchronized of with F1 if the output is F0 synchronized.

When the Avalon Slave run-time controller is enabled, the resolution of the output can be changed at run-time at a frame boundary, that is, before the first field of a pair when the output is interlaced.

Because the Test Pattern Generator does not accept an input stream, the pseudo-code in "Avalon-MM Slave Interfaces" on page 3–17 is slightly modified:

```
go = 0;
while (true)
{
    status = 0;
    while (go != 1 )
        wait();
    read_control(); //Copies control to internal register
    status = 1;
do once for progressive output or twice for interlaced output
{
   send_control_packet();
   send_image_data_header();
   output_test_pattern ();
}
}
```

Avalon-ST Video Protocol Parameters

The Test Pattern Generator MegaCore function supports a wide range of resolutions and color spaces with either a sequential or parallel data interface.

In all combinations of color space and subsampling that are allowed, the stream of pixel data is of a type consistent with the conventions adopted by the other MegaCore functions in the Video and Image Processing Suite.

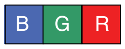
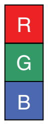
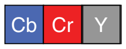
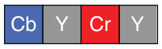
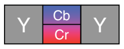
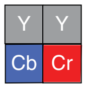
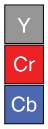
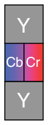
The Test Pattern Generator MegaCore function can output streams of pixel data of the types listed in Table 24–2.

☞ The Test Pattern Generator cannot produce interlaced streams of pixel data with an odd frame height. To create interlaced video streams where F0 fields are one line higher than F1 fields, Altera recommends feeding Test Pattern Generator progressive video output into the Interlacer MegaCore function.

Table 24–2. Test Pattern Generator Avalon-ST Video Protocol Parameters (Part 1 of 2)

Parameter	Value
Frame Width	Width selected in the parameter editor. Can be run-time controlled in which case, the value specified in the GUI is the maximum allowed value.
Frame Height	Height selected in the parameter editor. Can be run-time controlled in which case, the value specified in the GUI is the maximum allowed value.
Interlaced / Progressive	Mode selected in the parameter editor.
Bits per Color Sample	Number of bits per color sample selected in the parameter editor.
Color Space	As selected in the parameter editor. RGB (4:4:4 subsampling only) or YCbCr.

Table 24–2. Test Pattern Generator Avalon-ST Video Protocol Parameters (Part 2 of 2)

Parameter	Value			
Color Pattern	For RGB sequential data:		For RGB parallel data:	
	For 4:4:4 sequential data:		For 4:2:2 sequential data:	
	For 4:2:0 sequential data:		For 4:2:2 parallel data:	
	For 4:4:4 parallel data:		For 4:2:0 parallel data:	

Notes to Table 24–2:

(1) 4:2:2 and 4:2:0 subsampling are not available for the RGB color space.

(2) Vertical subsampling and interlacing cannot be used when the height of the output is not even. The GUI does not allow such a parameterization and the behavior of the MegaCore function is undefined if the height is subsequently set to an odd value through the run-time control.

(3) Vertical subsampling and interlacing are incompatible with each other and cannot be selected simultaneously in the GUI.

Stall Behavior and Error Recovery

All modes of the Test Pattern Generator stall for a few cycles after a field, after a control packet, and between lines. When producing a line of image data, the Test Pattern Generator outputs one sample on every clock cycle, but it can be stalled without consequences if other functions down the data path are not ready and exert backpressure.

For more information about the stall behavior and error recovery, refer to "Stall Behavior and Error Recovery" on page 1–3.

Latency

The Test Pattern Generator MegaCore function does not have latency issue because the MegaCore function is only an Avalon-ST Video source.

Parameter Settings

Table 24–3 lists the Test Pattern Generator MegaCore function parameters.

Table 24–3. Test Pattern Generator Parameter Settings (Part 1 of 2)

Parameter	Value	Description
Run-time control of image size	On or **Off**	Turn on to enable run-time control of the image size. When on, the output size parameters control the maximum values.
Maximum image width	32–2600, Default = **640**	Choose the required output width in pixels.
Maximum image height	32–2600, Default = **480**	Choose the required output height in pixels. This value must be the height of the full progressive frame when outputting interlaced data.

Table 24–3. Test Pattern Generator Parameter Settings (Part 2 of 2)

Parameter	Value	Description
Bits per pixel per color plane	4–20, Default = **8**	Choose the number of bits per pixel (per color plane).
Color space	**RGB** or YCbCr	Choose whether to use an R'G'B' or Y'CbCr color space.
Output format	**4:4:4**, 4:2:2, 4:2:0	Choose the format/sampling rate format for the output frames.
Color planes transmission format	**Sequence**, Parallel	This function always outputs three color planes but you can choose whether they are transmitted in sequence or in parallel.
Interlacing	**Progressive output**, Interlaced output (F0 first), Interlaced output (F1 first)	Specifies whether to produce a progressive or an interlaced output stream.
Pattern	**Color bars**, Uniform background	Choose the standard color bar or a uniform background.
Uniform values	0–255, Default = **128**	When pattern is uniform background, you can specify the individual R'G'B' or Y' Cb Cr values depending on the currently selected color space.

Signals

Table 24–4 lists the input and output signals for the Test Pattern Generator MegaCore function.

Table 24–4. Test Pattern Generator Signals (Part 1 of 2)

Signal	Direction	Description
`clock`	In	The main system clock. The MegaCore function operates on the rising edge of the `clock` signal.
`reset`	In	The MegaCore function asynchronously resets when you assert `reset`. You must deassert `reset` synchronously to the rising edge of the `clock` signal.
`control_av_address`	In	`control` slave port Avalon-MM `address` bus. Specifies a word offset into the slave address space. (1)
`control_av_chipselect`	In	`control` slave port Avalon-MM `chipselect` signal. The `control` port ignores all other signals unless you assert this signal. (1)
`control_av_readdata`	Out	`control` slave port Avalon-MM `readdata` bus. These output lines are used for read transfers. (1)
`control_av_write`	In	`control` slave port Avalon-MM `write` signal. When you assert this signal, the `control` port accepts new data from the `writedata` bus. (1)
`control_av_writedata`	In	`control` slave port Avalon-MM `writedata` bus. These input lines are used for write transfers. (1)
`dout_data`	Out	`dout` port Avalon-ST `data` bus. This bus enables the transfer of pixel data out of the MegaCore function.
`dout_endofpacket`	Out	`dout` port Avalon-ST `endofpacket` signal. This signal marks the end of an Avalon-ST packet.
`dout_ready`	In	`dout` port Avalon-ST `ready` signal. The downstream device asserts this signal when it is able to receive data.
`dout_startofpacket`	Out	`dout` port Avalon-ST `startofpacket` signal. This signal marks the start of an Avalon-ST packet.

Table 24–4. Test Pattern Generator Signals (Part 2 of 2)

Signal	Direction	Description
`dout_valid`	Out	`dout` port Avalon-ST `valid` signal. This signal is asserted when the MegaCore function outputs data.

Note to Table 24–4:

(1) These ports are present only if you turn on **Run-time control of image size**.

Control Register Map

The width of each register in the Test Pattern Generator control register map is 16 bits. The control data is read once at the start of each frame and is buffered inside the MegaCore function, so that the registers can be safely updated during the processing of a frame or pair of interlaced fields.

After control data has been read, the Test Pattern Generator MegaCore function outputs a control packet that describes the following image data packet. When the output is interlaced, the control data is processed only before the first field of a frame, although a control packet is sent before each field.

Table 24–5 describes the Test Pattern Generator MegaCore function control register map.

Table 24–5. Test Pattern Generator Control Register Map

Address	Register(s)	Description
0	`Control`	Bit 0 of this register is the `Go` bit, all other bits are unused. Setting this bit to 0 causes the Test Pattern Generator MegaCore function to stop before control information is read. Refer to “Generation of Avalon-ST Video Control Packets and Run-Time Control” on page 24–2 for full details.
1	`Status`	Bit 0 of this register is the `Status` bit, all other bits are unused. The Test Pattern Generator MegaCore function sets this address to 0 between frames. It is set to 1 while the MegaCore function is producing data and cannot be stopped. Refer to “Generation of Avalon-ST Video Control Packets and Run-Time Control” on page 24–2 for full details.
2	`Output Width`	The width of the output frames or fields in pixels. (1)
3	`Output Height`	The progressive height of the output frames or fields in pixels. (1)
4	`R/Y`	The value of the R (or Y) color sample when the test pattern is a uniform color background. (2)
5	`G/Cb`	The value of the G (or Cb) color sample when the test pattern is a uniform color background. (2)
6	`B/Cr`	The value of the B (or Cr) color sample when the test pattern is a uniform color background. (2)

Notes to Table 24–5:

(1) Value can be from 32 to the maximum specified in the parameter editor.

(2) These control registers are only available when the test pattern generator MegaCore function is configured to output a uniform color background and when the run-time control interface has been enabled.

25. Trace System MegaCore Function

Core Overview

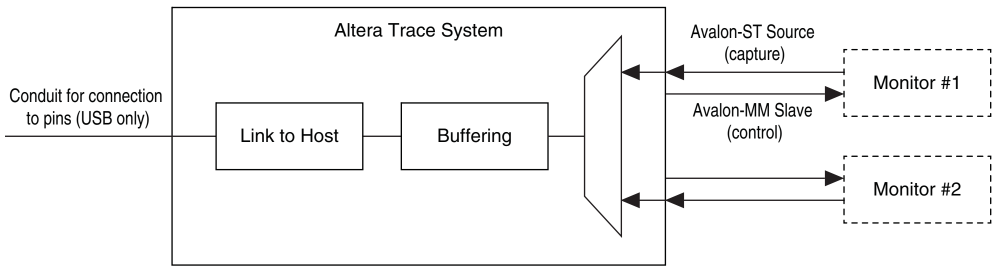
The Trace System MegaCore function is a debugging and monitoring component. The trace system collects data from various monitors, such as the Avalon-ST monitor, and passes it to the System Console software on the attached debugging host. The System Console software allows you to capture and visualize the behavior of the attached system. You can transfer data to the host over a JTAG connection or over a direct USB connection with a higher bandwidth, such as the one provided by On-Board USB-Blaster™ II.

Functional Description

The Trace System transports messages describing the captured events from the trace monitor components, such as the Avalon-ST Video Monitor, to a host computer running the System Console software.

Figure 25–1 shows the instantiation and connection of the Trace System.

Figure 25–1. Trace System Functional Block Diagram


Altera Trace System
Conduit for connection to pins (USB only)
Link to Host
Buffering
Avalon-ST Source (capture)
Avalon-MM Slave (control)
Monitor #1
Monitor #2

When you instantiate the Trace System, turn on the option to select the number of monitors required. The trace system exposes a set of interfaces, `capturen` and `controln`. You must connect each pair of the interfaces to the appropriate trace monitor component.

If you select the USB connection to the host, the trace system exposes the `usb_if` interface. Export this interface from the Qsys system and connect to the pins on the device that connects to the On-Board USB-Blaster II. If you select the JTAG connection to the host, then the Quartus II software automatically makes this connection during synthesis.

The trace system provides access to the `control` interfaces on the monitors. You can use these control ports to change the capture settings on the monitors; for example, to control the type of information captured by the monitors or to control the maximum data rate sent by the monitor.

Each type of monitor is different. Refer to the relevant documentation of the monitors for more information.

Each trace monitor sends information about interesting events through its capture interface. The trace system multiplexes these data streams together and, if the trace systems is running, stores them into a FIFO buffer. The contents of this buffer are streamed to the host using as much as the available trace bandwidth.

The amount of buffering required depends on the amount of jitter inserted by the link, in most cases, the default value of 32Kbytes is sufficient.

The System Console uses the **sopcinfo** file written by Qsys to discover the connections between the trace system and the monitors. If you instantiate and manually connect the trace system and the monitors using HDL, the System Console will not detect them.

Parameter Settings

Table 25–1 lists the Trace System MegaCore function parameters.

Table 25–1. Trace System Parameter Settings

Parameter	Value	Description
Connection to host	JTAG or USB, Default = **JTAG**	Choose the type of connection to the host running the System Console.
Bit width of capture interface(s)	8, 16, 32, 64, Or 128, Default = **32**	Choose the data bus width of the Avalon-ST interface sending the captured information.
Number of inputs	1+	Choose the number of trace monitors which will be connected to this trace system.
Buffer size	8k, 16k, 32k, or 64k Default = **32k**	Choose the size of the jitter buffer in bytes.
Insert pipeline stages	Boolean	Enable this option to insert the pipeline stages within the trace system. Enabling this option gives a higher f_{max} but uses more logic.

Signals

Table 25–2 lists the input and output signals for the Trace System MegaCore function.

Table 25–2. Trace System Signals (Part 1 of 2)

Signal	Direction	Description
`clk_clk`	In	All signals on the trace system are synchronous to this clock. Do not insert clock crossing between the monitor and the trace system components. You must drive the trace monitors' clocks from the same source which drives this signal.
`reset_reset`	Out	This signal is asserted when the trace system is being reset by the debugging host. Connect this signal to the reset inputs on the trace monitors. Do not reset parts of the system being monitored with this signal because this will interfere with functionality of the system.
`usb_if_clk`	In	Clock provided by On-Board USB-Blaster II. All `usb_if` signals are synchronous to this clock; the trace system provides clock crossing internally.

Table 25–2. Trace System Signals (Part 2 of 2)

Signal	Direction	Description
`usb_if_reset_n`	In	Reset driven by On-Board USB-Blaster II.
`usb_if_full`	Out	Host to the target full signal.
`usb_if_empty`	Out	Target to the host empty signal.
`usb_if_wr_n`	In	Write enable to the host to target FIFO.
`usb_if_rd_n`	In	Read enable to the target to host FIFO.
`usb_if_oe_n`	In	Output enable for data signals.
`usb_if_data`	Bidir	Shared data bus
`usb_if_scl`	In	Management interface clock.
`usb_if_sda`	In	Management interface data.
`capture`*n*`_data`	In	`capture`*n* port Avalon-ST `data` bus.
`capture`*n*`_endofpacket`	In	`capture`*n* port Avalon-ST `endofpacket` signal.
`capture`*n*`_empty`	In	`capture`*n* port Avalon-ST `empty` signal.
`capture`*n*`_ready`	Out	`capture`*n* port Avalon-ST `ready` signal.
`capture`*n*`_startofpacket`	In	`capture`*n* port Avalon-ST `startofpacket` signal.
`capture`*n*`_valid`	In	`capture`*n* port Avalon-ST `valid` signal.
`control`*n*`_address`	Out	`control`*n* slave port Avalon-MM `address` bus.
`control`*n*`_burstcount`	Out	`control`*n* slave port Avalon-MM `burstcount` bus.
`control`*n*`_byteenable`	Out	`control`*n* slave port Avalon-MM `byteenable` bus.
`control`*n*`_debugaccess`	Out	`control`*n* slave port Avalon-MM `debugaccess` signal.
`control`*n*`_read`	Out	`control`*n* slave port Avalon-MM `read` signal.
`control`*n*`_readdata`	In	`control`*n* slave port Avalon-MM `readdata` bus.
`control`*n*`_readdatavalid`	In	`control`*n* slave port Avalon-MM `readdatavalid` signal.
`control`*n*`_write`	Out	`control`*n* slave port Avalon-MM `write` signal.
`control`*n*`_writedata`	Out	`control`*n* slave port Avalon-MM `writedata` bus.
`control`*n*`_waitrequest`	In	`control`*n* slave port Avalon-MM `waitrequest` signal.

Operating the Trace System from System Console

The System Console provides a GUI and a TCL-scripting API that you can use to control the trace system.

To start System Console, do one of the following steps:

- Run `system-console` from the command line.
- In the Qsys, on the Tools menu, select **System Console**.
- In the Quartus II software, on the Tools menu, select **Transceiver Toolkit**.

☞ Close the transceiver toolkit panes within System Console.

Loading the Project and Connecting to the Hardware

To connect to the Trace System, the System Console needs access to the hardware and to the information about what the board does. To enable access for System Console, follow these steps:

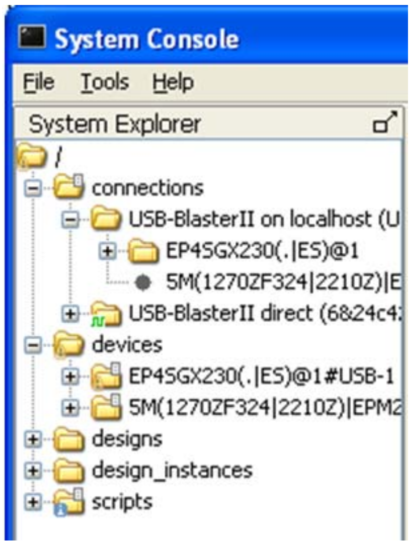
1. Connect to the host by doing one of the following:
   - Connect the On-Board USB-Blaster II to the host with the USB cable.
   - Connect the JTAG pins to the host with a USB-Blaster, Ethernet Blaster, or a similar cable.
2. Start the System Console and make sure that it detects your device.

   Figure 25–2 shows the System Explorer pane with the **connections** and **devices** folders expanded, with an On-Board USB-Blaster II cable connected.

   The individual connections appear in the **connections** folder, in this case the JTAG connection and the direct USB connections provided by the USB-Blaster II.

   The System Console discovers which connections go to the same device and creates a node in the **devices** folder for each unique device which visible at any time. If both connections go to the same device, then the device only appears once.

Figure 25–2. Connections and Devices Within the System Console



3. Load your design into the System Console. Do one of the following:
   - In the System Console window, on the File menu, select **Load Design**. Open the Quartus II Project File (**.qpf**) for your design.
   - From the System Console TCL shell, type the following command:

   ```
   [design_load </full/path/to/project.qpf>]
   ```

   You will get a full list of loaded designs by opening the designs' node within the System Explorer pane on the System Console window, or by typing the following command on the System Console TCL shell:

   ```
   [get_service_paths design]
   ```

4. After loading your design, link it to the devices detected by System Console. Do one of the following:
   - In the System Console window, right click on the device folder, and click **Link device to**. Then select your uploaded design.

     If your design has a JTAG USERCODE, the System Console is able to match it to the device and automatically links it after design is loaded.

   ☞ To set a JTAG USERCODE, in the Quartus II software, under Device Assignments menu, click **Device and Pin Options**>**General Category**, and turn on **Auto Usercode**.

   - From the System Console TCL shell, type the following command to manually link the design:

```
[design_link <design> <device>]
```

   ☞ Both `<design>` and `<device>` are System Console paths as returned by, for example, `[lindex [get_service_paths design] 0]`.

When the design is loaded and linked, the nodes representing the Trace System and the monitors are visible.

Trace Within The System Console

When the System Console detects a trace system, the Tools menu shows **Trace Table View**. Select this option to display the trace table view configuration dialogue box.

Each detected trace system contains an entry at the **Select hardware** drop down menu. Select one of them and the available settings for its monitors will display. Each type of monitor provides a different set of settings, which can be changed by clicking on the **Value** column.

Figure 25–3 shows the trace control bar, which lets you control the acquisition of data through the trace system.

Figure 25–3. Trace Control Bar Icons

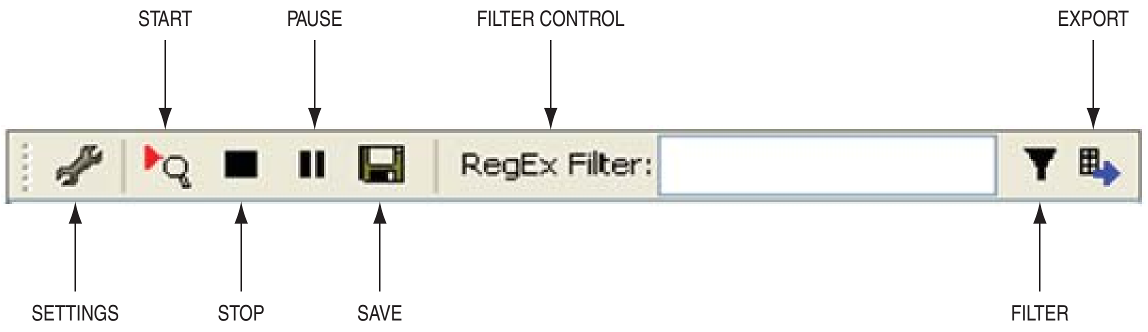


Table 25–3 lists the function for each trace bar icon.

Table 25–3. Functions for Trace Control Bar Icons

Icon	Function
Settings	Displays the configuration dialogue again.
Start	Tells the trace system to start acquiring data. Data is displayed in the table view as soon as possible after it is acquired.
Stop	Stops acquiring data.
Pause	Stops the display from updating data, but does not affect data acquisition. If you want to examine some data for a length of time, it good to pause so that your data is not aged out of the underlying database.
Save	Saves the raw captured data as a trace database file. You can reload this file using the Open file icon in the configuration dialogue.
Filter Control	Lets you filter the captured items to be displayed, but it does not affect acquisition. The filter accepts standard regular expression syntax—you can use filters such as blue/white to select either color.
Filter	Opens the filter settings dialogue, that allows you to select the parts of the captured data you want to display.
Export	Exports a text file containing the current contents of the trace table view. Filters affect the contents of this file.

TCL Shell Commands

You can control the Trace System components from the TCL scripting interface using `trace` service. Table 25–4 lists the commands and their functions.

Table 25–4. Trace System Commands (Part 1 of 2)

Command	Arguments	Function
`get_service_paths`	`trace`	Returns the System Console names for all the Trace System components which are currently visible.
`claim_service`	`trace` `<service_path>` `<library_name>`	Opens a connection to the specified trace service so it can be used. Returns a new path to the opened service.
`close_service`	`trace` `<open_service>`	Closes the service so that its resources can be reused.
`trace_get_monitors`	`<open_service>`	Returns a list of monitor IDs—one for each monitor that is available on this trace system.
`trace_get_monitor_info`	`<open_service>` `<monitor_id>`	Returns a serialized array containing information about the specified monitor. You can use the array set command to convert this into a TCL array.

Table 25–4. Trace System Commands (Part 2 of 2)

Command	Arguments	Function
`trace_read_monitor`	`<open_service>` `<monitor_id>` `<index>`	Reads a 32-bit value from configuration space within the specified monitor. Refer to the trace monitor documentation for the register maps.
`trace_write_monitor`	`<open_service>` `<monitor_id>` `<index><value>`	Writes a 32-bit value from configuration space within the specified monitor. Refer to the trace monitor documentation for the register maps.
`trace_get_max_db_size`	`<open_service>`	Gets the maximum (in memory) trace database size set for this trace system. If the trace database size exceeds this value, then the oldest values are discarded.
`trace_set_max_db_size`	`<open_service>` `<size>`	Returns the current maximum trace database size. Trace database sizes are approximate but can be used to prevent a high data rate monitor from using up all available memory.
`trace_start`	`<open_service>fifo`	Starts capturing with the specified trace system in real time (fifo) mode.
`trace_stop`	`<open_service>`	Stops capturing with the specified trace system.
`trace_get_status`	`<open_service>`	Returns the current status (idle or running) of the trace system. In future, new status values may be added.
`trace_get_db_size`	`<open_service>`	Returns the approximate size of the database for the specified trace system.
`trace_save`	`<open_service>` `<filename>`	Saves the trace database to disk.
`trace_load`	`<filename>`	Loads a trace database from disk. This returns a new service path, which can be viewed as if it is a trace system. However, at this point, the start, stop and other commands will obviously not work on a file-based node. If you load a new trace database with the `trace_load` command, the trace GUI becomes visible if it was previously hidden.

A. Avalon-ST Video Verification IP Suite

This chapter describes the Avalon-ST Video Verification IP Suite. The Avalon-ST Video Verification IP Suite provides a set of SystemVerilog classes (the class library) that you can use to ensure that a Video IP conforms to the Avalon-ST Video standard.

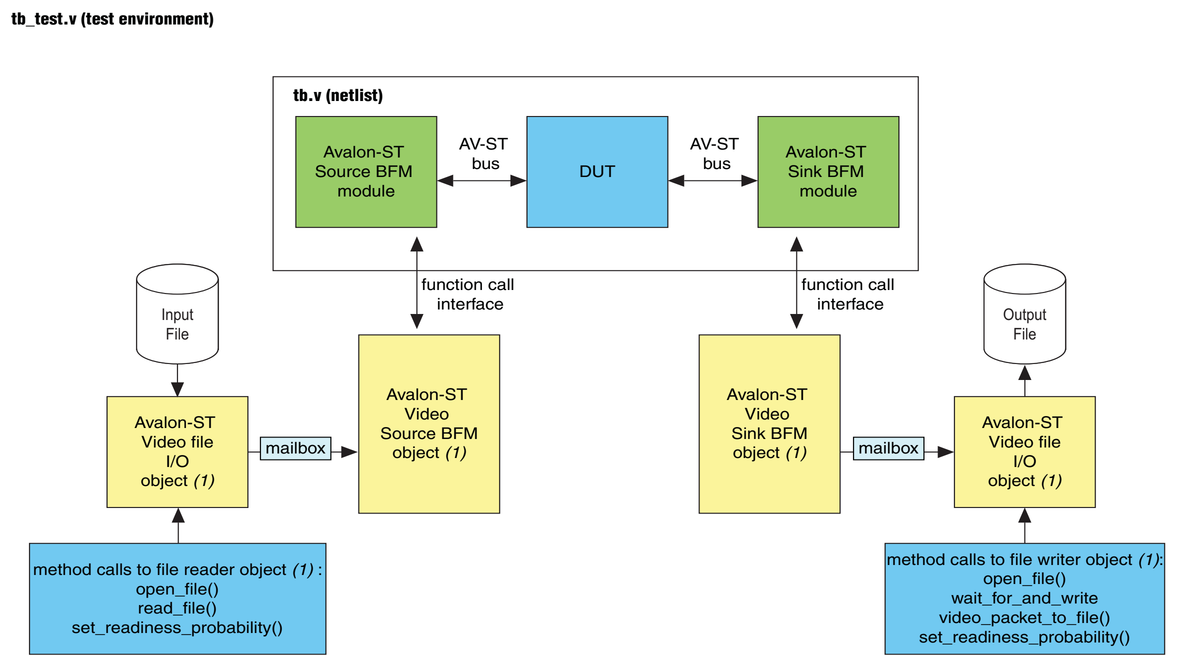
This chapter contains the following section:

- Avalon-ST Video Verification IP Suite Overview
- Avalon-ST Video Class Library
- Video File Reader Test
- Constrained Random Test
- Complete Class Reference

Avalon-ST Video Verification IP Suite Overview

Figure A–1 shows the elements in the Avalon-ST Video Verification IP Suite. Yellow indicates the class library components of the test environment, green indicates the Avalon-ST bus functional model (BFM), and blue indicates the device under test (DUT) and the test method calls themselves.

Figure A–1. Test Environment for the Avalon-ST Video Class Library



tb_test.v (test environment)
tb.v (netlist)
Avalon-ST Source BFM module
AV-ST bus
DUT
AV-ST bus
Avalon-ST Sink BFM module
function call interface
function call interface
Input File
Output File
Avalon-ST Video file I/O object (1)
mailbox
Avalon-ST Video Source BFM object (1)
Avalon-ST Video Sink BFM object (1)
mailbox
Avalon-ST Video file I/O object (1)
method calls to file reader object (1) : open_file() read_file() set_readiness_probability()
method calls to file writer object (1): open_file() wait_for_and_write video_packet_to_file() set_readiness_probability()

Note to Figure A–1:

(1) Objects are built using the Avalon-ST Video Class Library.

In Figure A–1, DUT is fed with Avalon-ST Video-compliant video packets and control packets. The responses from the DUT are collected, analyzed, and the resultant video written to an output file.

Although the test environment in Figure A–1 shows a simple example of using the class library, other test environments can conform to this test structure (with respect to the Verilog module-level connectivity and object/class-level connectivity.)

The class library uses the Avalon-ST source and sink `BFM [1]` and provides the following functionality:

- Embodies the Avalon-ST Video standard to facilitate compliance testing.
- Implements a host of common Avalon-ST Video protocol failures that the DUT can be tested against. You can configure these using simple method calls to the class library.
- Implements file reader or file writer functionality to facilitate DUT testing with real video sequences.
- The class library is built from a fresh code-base, designed from the ground-up from newly-defined objects such as 'pixels' and 'video packets', which means:
  - The library code is easily understandable for new users.
  - The library code has all the properties of good object-oriented code design, so it is easily extensible to meet any further requirements.
- Uses SystemVerilog's powerful verification features such as mailboxes and randomization of objects, allowing you to construct complex and noisy bus environments easily for rigourous stress-testing of DUTs.

Avalon-ST Video Class Library

This section describes the class library.

Figure A–2 shows a unified modeling language (UML)-styled diagram of the class structure of the library and how these break down into individual files and packages.

☞ The method call arguments are not shown in Figure A–2 but are detailed in "Complete Class Reference" on page A–22.

Figure A–2. UML-Style Class Diagram

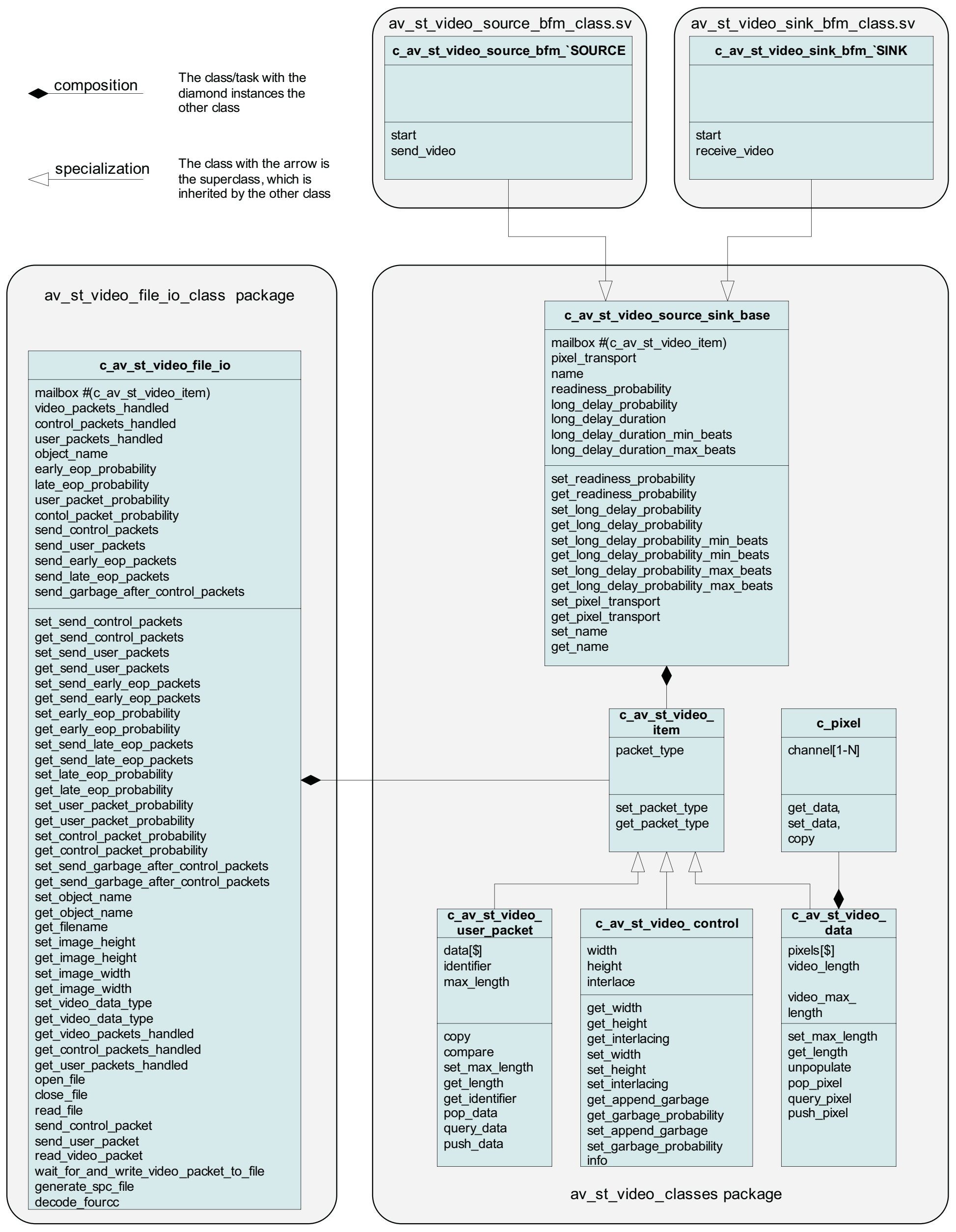


Table A–1 describes each of the classes in the *av_st_video_classes* package shown in Figure A–2.

Table A–1. Class Descriptions

Class	Description	Parameter
class c_av_st_video_item	The most fundamental of all the classes. Represents any item that is sent over the Avalon-ST bus and contains a `packet_type` field. You can set the field to `video_packet`, `control_packet`, or `user_packet` types. These three packet types are represented by classes which extend this base class. Structuring the classes in this way allows you to define the mailboxes and queues of *c_av_st_video_items*. Then, you can send any type of packet in the order that they appear on the bus. For more information about the method call and members of this class library, refer to "c_av_st_video_item" on page A–28.	—
class c_pixel	Fundamental and parameterized class. Comprised of an array of channels that contains pixel data. For example, a pixel from an RGB24 video system comprises an array of three channels (8 bits per channel). A pixel for a YcbCr system is comprised of two channels. An individual channel either represents a luminence or chroma-type component of video data, one RGB component, or one alpha component. The class provides "getters", "setters", and "copy" methods. For more information about the method call and members of this class library, refer to "c_pixel" on page A–33.	■ **BITS_PER_CHANNEL** ■ **CHANNELS_PER_PIXEL**
class c_av_st_video_data	Parameterized class. Contains a queue of pixel elements. This class library is used by other classes to represent fields of video and line (or smaller) units of video. It extends *c_av_video_item*. The class provides methods to push and pop pixels on and off the queue. For more information about the method call and members of this class library, refer to "c_av_st_video_data" on page A–24.	■ **BITS_PER_CHANNEL** ■ **CHANNEL_PER_PIXEL**
class c_av_st_video_control	Parameterized class. Extends *c_av_video_item*. Comprised of width, height, and interlaced bits (the fields found in an Avalon-ST video control packet). It also contains data types and methods that control the addition of garbage beats that are used by other classes. The class provides methods to get and set the individual fields. For more information about the method call and members of this class library, refer to "c_av_st_video_control" on page A–23.	■ **BITS_PER_CHANNEL** ■ **CHANNELS_PER_PIXEL**
class c_av_st_user_packet	Parameterized class. Contains a queue of data and is used by the other classes to represent packets of user data. It extends *c_av_video_item*. The class provides methods to push and pop data on and off the queue. For more information about the method call and members of this class library, refer to "c_av_st_video_user_packet" on page A–32.	■ **BITS_PER_CHANNEL** ■ **CHANNEL_PER_PIXEL**

The classes listed in Table A–1 do not contain information about the physical transport mechanism and the Avalon-ST Video protocol. To foster advanced verification techniques, Altera uses a high-level abstract view.

Table A–2 describes the classes included in the *av_st_video_file_io_class* package and the source and sink class packages.

Table A–2. Additional Class Descriptions (Part 1 of 2)

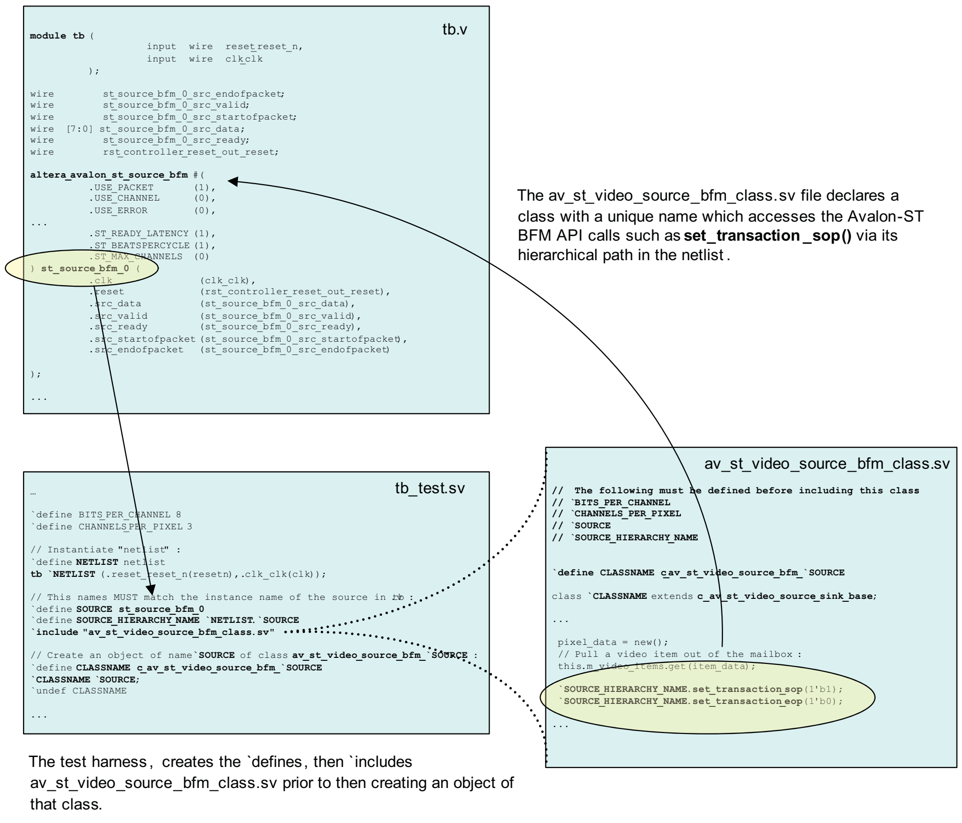
Class	Description
class c_av_st_video_source_sink_base	Designed to be extended by source and sink BFM classes. Contains a mailbox of *c_av_st_video_item*, together with various fields that define the transport mechanism (serial or parallel), record the numbers of packets sent, and define the service quality (readiness) of the source or sink. For more information about the method call and members of this class library, refer to "c_av_st_video_source_sink_base" on page A–30.
class c_av_st_video_source_bfm_'SOURCE	Extends *c_av_st_video_source_sink_base*. Named according to the instance names of the Avalon-ST source and sink BFMs in the SystemVerilog netlist. This is because you must access the API functions in the Avalon-ST BFMs by directly calling them through the design hierarchy. Therefore, this hierarchy information is required in the Avalon-ST video source and sink classes. This means that a unique class with the correct design hierarchy information for target source or sink is required for every object created of that class type. To overcome this limitation, create the source and sink class files (**av_st_video_bfm_class.sv** and **av_st_video_sink_bfm_class.sv**) which are designed to be **'included** into the test environment with **'defines** set to point to the correct hierarchy, as shown in figure Figure A–3. The source class comprises of a simple `start()` task and a `send_video` task (called by the start task). The `send_video` task continually polls its mailbox. When a `video_item` arrives, the `video_item` is assembled into a set of transactions according to its type and the transport mechanism specified. Then, the `video_item` is sent to the Avalon-ST BFM. One Avalon-ST BFM transaction is considered as one beat on the Avalon-ST bus, comprised of the logic levels on the `SOP`, `EOP`, `READY`, `VALID` signals, as well as the data on the bus in a given clock cycle. For example, a video packet is sent to the BFM preceded by a 0x0 on the LSB of the first transaction, as per the Avalon-ST video protocol. A control packet is preceded by a 0xf on the LSB. Then, the height, width and interlacing fields are sent in subsequent transaction in accordance to the Avalon-ST Video protocol. The class *c_av_st_video_source_bfm_`SOURCE* requires you to create an object from it and to call the `start()` task as it automatically handles any `video_items` sent to its mailbox. No other interaction is required. For more information about the method calls and members of this class library, refer to "c_av_st_video_source_bfm_'SOURCE" on page A–32. For more information, refer to *Section IV. Avalon-ST BFMs* section in the *Avalon Verification IP Suite User Guide*.

Table A–2. Additional Class Descriptions (Part 2 of 2)

Class	Description
class c_av_st_video_sink_bfm_'SINK	Operates in the same way as the source class except it contains a `receive_video()` task and performs the opposite function to the source. This class receives incoming transactions from the Avalon-ST sink BFM, decoding their type, assembling them into the relevant objects (control, video, or user packets), and pushing them out of its mailbox. No further interaction is required from the user. For more information about the method call and members of this class library, refer to "c_av_st_video_sink_bfm_'SINK" on page A–31.
class c_av_st_video_file_io	This parameterized class is defined in a separate file (**av_st_video_file_io_class.sv**) because some test environments do not use video data from a file, using constrained random data generated by the other classes instead. This class provides the following methods: ■ to read and write video files (in **.raw** format) ■ to send or receive videos and control packet objects to or from the mailbox. Variables that govern the file I/O details include the ability to artificially lengthen and shorten video packets and to introduce garbage beats into control packets by various get and set method calls. Typical usage of the file I/O class is to construct two objects—a reader and a writer, call the open file methods for both, call the `read_file` method for the reader, and repeatedly call the `wait_for_and_write_video_packet_to_file` method in the writer. For more information about the method call and members of this class library, refer to "c_av_st_video_file_io" on page A–25. The parameters for the class are **BITS_PER_CHANNEL** and **CHANNELS_PER_PIXEL**.

Figure A–3 shows the solution to the Avalon-ST BFM API function.

Figure A–3. How the Class Library interfaces to the Avalon-ST BFM



```
module tb (
                input  wire  reset_reset_n,
                input  wire  clk_clk
        );

wire        st_source_bfm_0_src_endofpacket;
wire        st_source_bfm_0_src_valid;
wire        st_source_bfm_0_src_startofpacket;
wire  [7:0] st_source_bfm_0_src_data;
wire        st_source_bfm_0_src_ready;
wire        rst_controller_reset_out_reset;

altera_avalon_st_source_bfm #(
        .USE_PACKET       (1),
        .USE_CHANNEL      (0),
        .USE_ERROR        (0),
...
        .ST_READY_LATENCY (1),
        .ST_BEATSPERCYCLE (1),
        .ST_MAX_CHANNELS  (0)
) st_source_bfm_0 (
        .clk              (clk_clk),
        .reset            (rst_controller_reset_out_reset),
        .src_data         (st_source_bfm_0_src_data),
        .src_valid        (st_source_bfm_0_src_valid),
        .src_ready        (st_source_bfm_0_src_ready),
        .src_startofpacket (st_source_bfm_0_src_startofpacket),
        .src_endofpacket  (st_source_bfm_0_src_endofpacket)

);

...
```

The av_st_video_source_bfm_class.sv file declares a class with a unique name which accesses the Avalon-ST BFM API calls such as **set_transaction_sop()** via its hierarchical path in the netlist.

```
…

`define BITS_PER_CHANNEL 8
`define CHANNELS_PER_PIXEL 3

// Instantiate "netlist" :
`define NETLIST netlist
tb `NETLIST (.reset_reset_n(resetn),.clk_clk(clk));

// This names MUST match the instance name of the source in tb :
`define SOURCE st_source_bfm_0
`define SOURCE_HIERARCHY_NAME `NETLIST.`SOURCE
`include "av_st_video_source_bfm_class.sv"

// Create an object of name`SOURCE of class av_st_video_source_bfm_`SOURCE :
`define CLASSNAME c_av_st_video_source_bfm_`SOURCE
`CLASSNAME `SOURCE;
`undef CLASSNAME

...
```

```
//  The following must be defined before including this class
// `BITS_PER_CHANNEL
// `CHANNELS_PER_PIXEL
// `SOURCE
// `SOURCE_HIERARCHY_NAME


`define CLASSNAME c_av_st_video_source_bfm_`SOURCE

class `CLASSNAME extends c_av_st_video_source_sink_base;

...

 pixel_data = new();
 // Pull a video item out of the mailbox :
 this.m_video_items.get(item_data);

 `SOURCE_HIERARCHY_NAME.set_transaction_sop(1'b1);
 `SOURCE_HIERARCHY_NAME.set_transaction_eop(1'b0);

...
```

The test harness, creates the `defines, then `includes av_st_video_source_bfm_class.sv prior to then creating an object of that class.

Types of Example Test Environment

There are two types of example test environment:

- Video File Reader Test
- Constrained Random Test

The video file reader test is useful for checking the video functionality of the DUT for any video types. However, this test is not suitable to test the DUT with a variety of differently-sized and formatted video fields. Altera recommends a constrained random approach that is easily accomplished using the class library. For more information, refer to "Constrained Random Test" on page A–18.

This section describes how you can run the test and the prerequisites before running the test. You can run both tests in a similar way, unless specified otherwise.

Running the Tests

This example system is available in the Quartus II install directory at:

- For example video files test:
  `$(QUARTUS_ROOTDIR)/../ip/altera/vip/verification/example_video_files`
- For constrained random test:
  `$(QUARTUS_ROOTDIR)/../ip/altera/vip/verification/example_constrained_random`

☞ The actual commands used in this section are for a Linux example. However, the same flow applies for Windows users.

Requirement

Before you can run the test, you must automatically generate the **tb.v** netlist from the Qsys system integration tool in the Quartus II software by following these steps:

1. Copy the verification files to a local directory and cd to the testbench directory below (where `$ALTERA_VIDEO_VERIFICATION` is a local directory of your choice):

```
>cp $(QUARTUS_ROOTDIR)/../ip/altera/vip/verification
$ALTERA_VIDEO_VERIFICATION
```

```
>cd $ALTERA_VIDEO_VERIFICATION/testbench
```

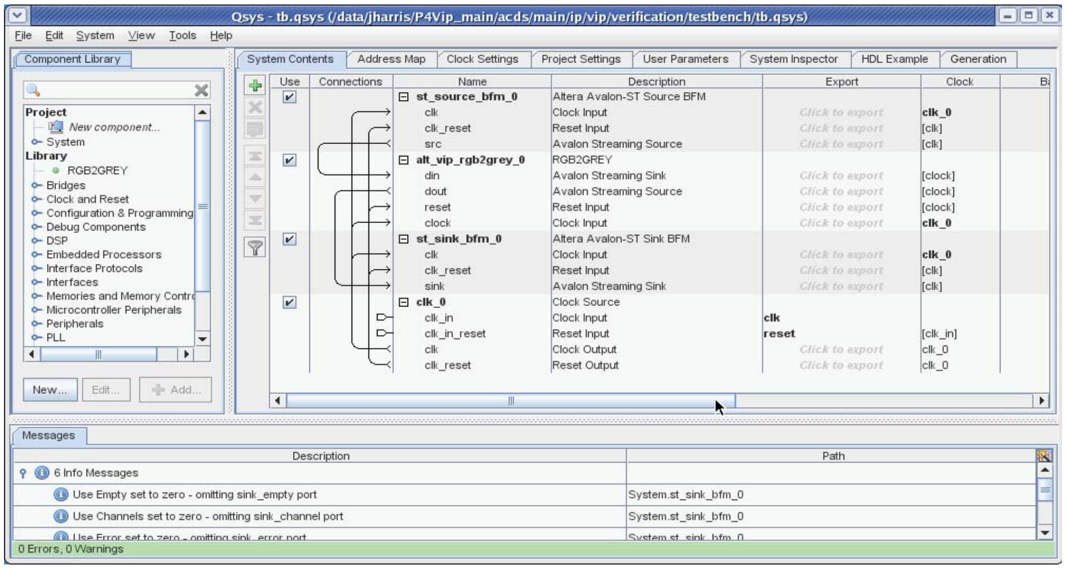
2. Start the Qsys system integration tool from the Quartus II software (**tools > Qsys**) or through command line:

```
eg. G:\altera\11.1\quartus\sopc_builder\bin>qsys-edit
```

3. Load the Qsys project by opening **tb.qsys**
4. Update the IP search path by selecting from the **Tools menu > Options > Add**. Navigate to one directory higher and into the **dut** directory.
5. Click **Open**, then **Finish**.

   The system refreshes and shows the RGBtogreyscaleconvertor (in between Avalon-ST source and sink BFMs), which is our example DUT. You can easily replaced this by any other user IP function, as shown in Figure A–4.

Figure A–4. QSys Dialog Box


Qsys - tb.qsys (/data/jharris/P4Vip_main/acds/main/ip/vip/verification/testbench/tb.qsys)
File Edit System View Tools Help
Component Library
Project
New component...
System
Library
RGB2GREY
Bridges
Clock and Reset
Configuration & Programming
Debug Components
DSP
Embedded Processors
Interface Protocols
Interfaces
Memories and Memory Contr
Microcontroller Peripherals
Peripherals
PLL
New... Edit... Add...
System Contents Address Map Clock Settings Project Settings User Parameters System Inspector HDL Example Generation
Messages

Use	Connections	Name	Description	Export	Clock
✓		**st_source_bfm_0**	Altera Avalon-ST Source BFM		
		clk	Clock Input	Click to export	**clk_0**
		clk_reset	Reset Input	Click to export	[clk]
		src	Avalon Streaming Source	Click to export	[clk]
✓		**alt_vip_rgb2grey_0**	RGB2GREY		
		din	Avalon Streaming Sink	Click to export	[clock]
		dout	Avalon Streaming Source	Click to export	[clock]
		reset	Reset Input	Click to export	[clock]
		clock	Clock Input	Click to export	**clk_0**
✓		**st_sink_bfm_0**	Altera Avalon-ST Sink BFM		
		clk	Clock Input	Click to export	**clk_0**
		clk_reset	Reset Input	Click to export	[clk]
		sink	Avalon Streaming Sink	Click to export	[clk]
✓		**clk_0**	Clock Source		
		clk_in	Clock Input	**clk**	
		clk_in_reset	Reset Input	**reset**	[clk_in]
		clk	Clock Output	Click to export	clk_0
		clk_reset	Reset Output	Click to export	clk_0

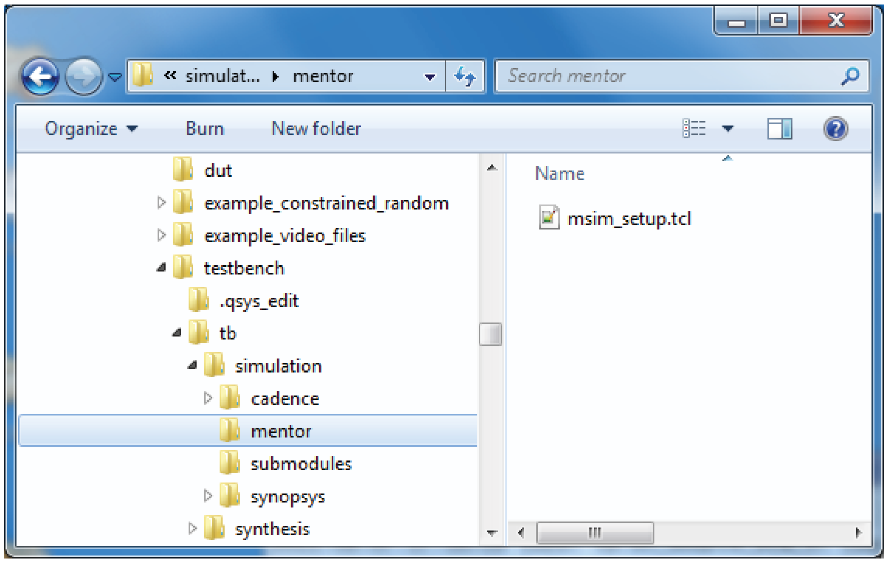
Description	Path
6 Info Messages	
Use Empty set to zero - omitting sink_empty port	System.st_sink_bfm_0
Use Channels set to zero - omitting sink_channel port	System.st_sink_bfm_0
Use Error set to zero - omitting sink_error port	System.st_sink_bfm_0

0 Errors, 0 Warnings

6. Create the **tb.v** netlist from the Qsys project by selecting **Generation**, set **Create simulation model** to **Verilog.** Select **Generate**. Close the **generate completed** dialog box, and exit Qsys.

   Qsys has now generated the **tb.v** netlist and all the required simulation files, as shown in Figure A–5.

Figure A–5. tb.v Netlist



7. Run the test by changing to the example video files test or example constrained random test directory and start the QuestaSim™ software:

- For example video files test:

```
>cd $ALTERA_VIDEO_VERIFICATION/example_video_files

 >vsim -do run.tcl
```

- For constrained random test:

```
>cd $ALTERA_VIDEO_VERIFICATION/example_constrained_random

 >vsim -do run.tcl
```

To run with other simulators, edit the **run.tcl** file as provided by Qsys (including the appropriate TCL script for that vendor’s simulator). The test runs and completes with the following message:

```
“Simulation complete. To view resultant video, now run the windows
raw2avi application.”
```

The example video files test produces a raw output video file (**vip_car_out.raw**). Together with an **.spc** file (**vip_car_out.spc**), you can use the **vip_car_out.raw** to generate an **.avi** file for viewing. To generate the **.avi** file, open a DOS command prompt from a Windows machine and run the following convertor utility:

```
C:>raw2avi.exe vip_car_out.raw video.avi
```

You can view the **video.avi** file with a media player. The media player shows a greyscale version of the source video file (**vip_car_0.avi**) that you can also play to see the full color video. You can view the full sequence from which the clip is taken in the **vip_car.avi** file.

Video File Reader Test

This section describes the example test environment shown in Figure A–1 on page A–1—the simplest way of using the class library. A video file is read, translated into `video_item` objects, streamed to the DUT by the BFM, and then retranslated back into video, and written to file again.

This section focuses on the non-standard elements of the environment that requires input shown in Figure A–1 on page A–1.

Test Environment

The test environment itself is set up through the **tb_test.v** file (found in the example_video_files directory), which instantiates the Qsys generated netlist, creates the necessary classes, and sets the test running. The main feature of the codes are described in the following sections:

- "tb_test.sv—Section 1"
- "tb_test.sv—Section 2"
- "tb_test.sv—Section 3"
- "tb_test.sv—Section 4"

tb_test.sv—Section 1

Example A–1 shows the first section of the code.

Example A–1. tb_test.sv—Section 1

```
`timescale 1ns / 1ns

module tb_test;

`define CHANNELS_PER_PIXEL  3
`define BITS_PER_CHANNEL    8

import av_st_video_classes::*;
import av_st_video_file_io_class::*;

// Create clock and reset:
logic clk, reset;

initial
    clk <= 1'b0;

always
    #2.5 clk <= ~clk; //200 MHz

initial
begin
    reset <= 1'b1;
    #10 @(posedge clk) reset <= 1'b0;
end

// Instantiate "netlist" :
`define NETLIST netlist
tb `NETLIST (.reset(reset),.clk(clk));

// Create some useful objects from our defined classes :
c_av_st_video_data        #(`BITS_PER_CHANNEL, `CHANNELS_PER_PIXEL)    video_data_pkt1;
c_av_st_video_control     #(`BITS_PER_CHANNEL, `CHANNELS_PER_PIXEL) video_control_pkt1;
c_av_st_video_user_packet #(`BITS_PER_CHANNEL, `CHANNELS_PER_PIXEL)          user_pkt1;
```

First, the test must define the numbers of bits per channel and channels per pixel, because most of the classes require this information. Next, the class packages are imported, the clock and reset defined, and the netlist itself instantiated with connections for clock and reset in place.

The BFM resets are all active high. If an active low reset is required, it may be necessary to invert the reset at the DUT input.

The final part of the this section of the code creates some objects from the class library which are used later in the code. The parameterization is standard across all object instances of the classes as the bits per channel and channels per pixel is constant in any given system.

tb_test.sv—Section 2

Example A–2 shows the second section of the code.

Example A–2. tb_test_sv (Section 2)

```
// This creates a class with a names specific to `SOURCE0, which is needed
// because the class calls functions for that specific `SOURCE0.  A class
// is used so that individual mailboxes can be easily associated with
// individual sources/sinks :

// This names MUST match the instance name of the source in tb.v :
`define SOURCE st_source_bfm_0
`define SOURCE_STR "st_source_bfm_0"
`define SOURCE_HIERARCHY_NAME `NETLIST.`SOURCE
`include "av_st_video_source_bfm_class.sv"

// Create an object of name `SOURCE of class av_st_video_source_bfm_`SOURCE :
`define CLASSNAME c_av_st_video_source_bfm_`SOURCE
`CLASSNAME `SOURCE;
`undef CLASSNAME

// This names MUST match the instance name of the sink in tb.v :
`define SINK st_sink_bfm_0
`define SINK_STR "st_sink_bfm_0"
`define SINK_HIERARCHY_NAME `NETLIST.`SINK
`include "av_st_video_sink_bfm_class.sv"

// Create an object of name `SINK of class av_st_video_sink_bfm_`SINK :
`define CLASSNAME c_av_st_video_sink_bfm_`SINK
`CLASSNAME `SINK;
`undef CLASSNAME

// Create mailboxes to transfer video packets and control packets :
mailbox #(c_av_st_video_item) m_video_items_for_src_bfm  = new(0);
mailbox #(c_av_st_video_item) m_video_items_for_sink_bfm = new(0);

// Now create file I/O objects to read and write :
c_av_st_video_file_io #(`BITS_PER_CHANNEL, `CHANNELS_PER_PIXEL) video_file_reader;
c_av_st_video_file_io #(`BITS_PER_CHANNEL, `CHANNELS_PER_PIXEL) video_file_writer;

int r;
int fields_read;
```

In this section, the video source and sink BFMs are declared as previously described in Figure A–3 on page A–7. When creating your own tests, ensure that the correct **`` `defines ``** are in place and the **av_st_video_source_bfm_class.sv** and **av_st_video_sink_bfm_class.sv** files are in the correct directory as required by the **`` `include ``**. After the source and sink BFMs are declared, two mailboxes are declared—`m_video_items_for_src_bfm` and `m_video_items_for_sink_bfm`, each of type `c_av_st_video_item`. These shall be used to pass video items from the file reader into the source BFM and from the sink BFM to the file writer.

At the end of this section, the file I/O class is used to declare the file reader and file writer objects.

tb_test.sv—Section 3

Example A–3 shows the third section of the code.

Example A–3. tb_test.sv (Section 3)

```
initial
begin

    wait (resetn == 1'b1)
    repeat (4) @ (posedge (clk));

    // Constructors associate the mailboxes with the source and sink classes
    `SOURCE = new(m_video_items_for_src_bfm);
    `SINK   = new(m_video_items_for_sink_bfm);

    `SOURCE.set_pixel_transport(`TRANSPORT);
      `SINK.set_pixel_transport(`TRANSPORT);

    `SOURCE.set_name(`SOURCE_STR);
      `SINK.set_name(  `SINK_STR);

    `SOURCE.set_readiness_probability(90);
      `SINK.set_readiness_probability(90);

    `SOURCE.set_long_delay_probability(0.01);
      `SINK.set_long_delay_probability(0.01);
```

In this code, after reset has gone high, the video source and sink BFM objects are **constructed** with the previously declared mailboxes. Then, some method calls are made to configure the transport mechanism, name the objects (for reporting purposes), and set some attributes regarding readiness and probability of long delays.

tb_test.sv—Section 4

Example A–4 shows the final section of the code.

Example A–4. tb_test.sv (Section 4)

```
fork

`SOURCE.start();
`SINK.start();

begin

    // File reader :

    // Associate the source BFM's video in mailbox with the video
    // file reader object via the file reader's constructor :
    video_file_reader = new(m_video_items_for_src_bfm);
    video_file_reader.set_object_name("file_reader_0");

    video_file_reader.open_file("flag_i_crop.raw",read, 60, 100, 4'b1100);
    video_file_reader.read_file();
    video_file_reader.close_file();

    fields_read = video_file_reader.get_video_packets_handled();

    // File writer :

    video_file_writer = new(m_video_items_for_sink_bfm);
    video_file_writer.set_object_name("file_writer_0");
    video_file_writer.open_file("data_out.raw", write, 60, 100, 4'b1100);

    // Write the video output packets to a file :
    do
    begin
        video_file_writer.wait_for_and_write_video_packet_to_file();
    end
    while ( video_file_writer.get_video_packets_handled() < fields_read );

    video_file_writer.close_file();

    $finish;

end
```

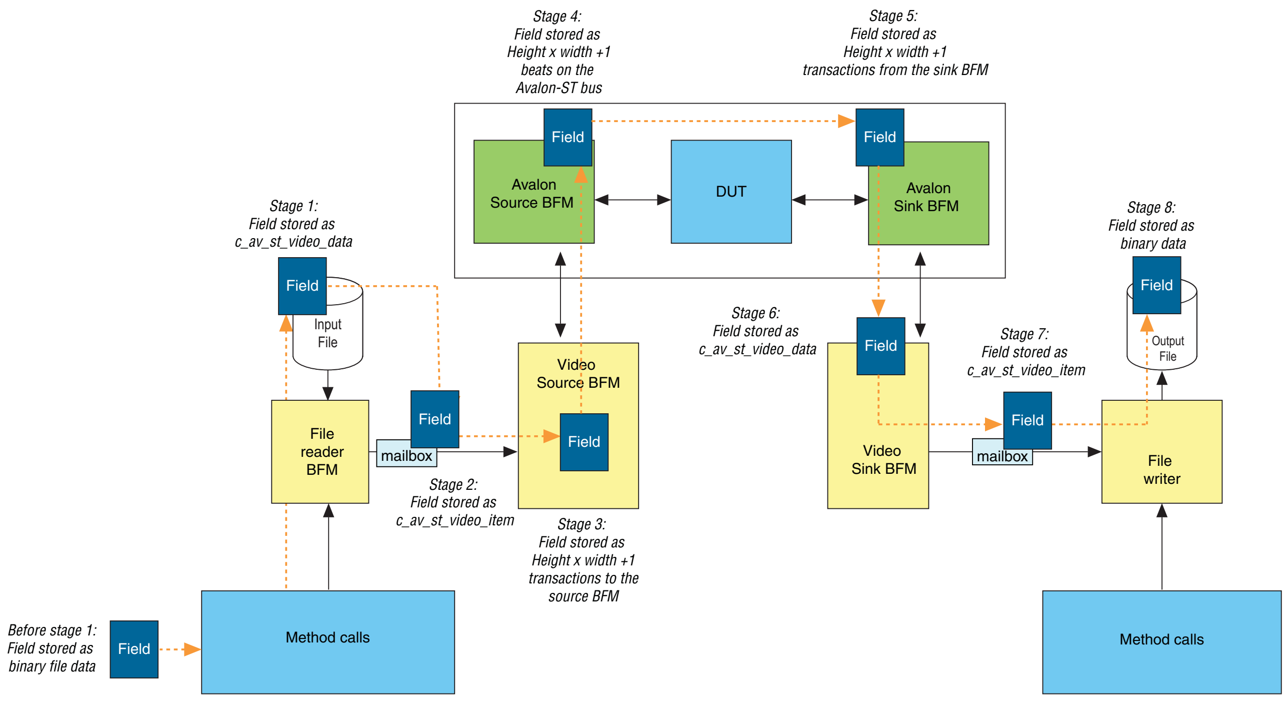
The final section of the code is a three parallel blocks. The first and second blocks call the start methods for the source and sink video BFMs. After started, the source waits for an entry into its mailbox to appear, and the sink waits for a transaction from the Avalon-ST sink BFM to appear. The third block **constructs** the file reader object, which is connected to the video source BFM's mailbox by its constructor. Then, method calls are made to name the reader, open a video file, read the file, and close the file. After the file has been read and closed, a final method call returns the number of fields read.

The use of mailboxes allows you to set up events without concern as to whether a particular source or sink is ready. This allows you to issue all the commands to the file reader before constructing the file writer (refer to the final part of Example A–4). The writer is constructed, named, and an output file specified. `wait_for_and_write_video_packets_to_file` method is then called. This method call handles one video packet at a time. Therefore, this method is called once for every field of video that the reader reads. After every field has been read, the output file is closed and the test finishes.

Video Field Life Cycle

To aid understanding, this section describes how a video field is represented in the system. Figure A–6 shows the life cycle of the video field.

Figure A–6. Video Field Life Cycle


Stage 4: Field stored as Height x width +1 beats on the Avalon-ST bus
Stage 5: Field stored as Height x width +1 transactions from the sink BFM
Field
Avalon Source BFM
DUT
Field
Avalon Sink BFM
Stage 1: Field stored as c_av_st_video_data
Field
Input File
Stage 8: Field stored as binary data
Field
Output File
Stage 6: Field stored as c_av_st_video_data
Stage 7: Field stored as c_av_st_video_item
Video Source BFM
Field
File reader BFM
Field
mailbox
Field
Video Sink BFM
Field
mailbox
File writer
Stage 2: Field stored as c_av_st_video_item
Stage 3: Field stored as Height x width +1 transactions to the source BFM
Before stage 1: Field stored as binary file data
Field
Method calls
Method calls

In Stage 1, a method call to the reader initiates the field to be read by the file reader BFM. The reader streams binary pixel data from the file, packs each pixel into an object of type `c_pixel`, and pushes the pixels into a `c_av_st_video_data` video object.

In Stage 2, after the reader assembles a complete video object, the reader casts the video object into the base class (`c_av_st_video_item`). The following is the code for this base class:

```
typedef c_av_st_video_data #(BITS_PER_CHANNEL, CHANNELS_PER_PIXEL)
video_t;
```

```
item_data = video_t'(video_data);
```

After the reader casts the video object into the base class, it sends the video object to the mailbox.

In Stage 3, the video source BFM retrieves the data from its mailbox, recasts the data back into a `c_av_st_video_data` video object, and begins translating it into transactions for the Avalon-ST source BFM. To indicate that a video packet is being sent, there is one transaction per pixel and an initial transaction with LSBs of 0×0 when using RGB24 data, 24-bit data buses, and parallel transmission.

In Stage 4, the Avalon-ST source BFM turns each transaction into beats of data on the Avalon-ST bus, which are received by the DUT.

In Stage 5, the DUT processes the data and presents the output data on the Avalon-ST bus. The Avalon-ST Sink BFM receives these and triggers a `signal_transaction_received` event for each beat.

In Stage 6, after the video sink BFM detects the `signal_transaction_received` event, the video sink BFM starts collecting transaction data from the BFM. When a start of packet (SOP) beat is detected, the type of the packet is verified, and transaction data is pushed into a pixel object, which is in turn pushed into a video_data object.

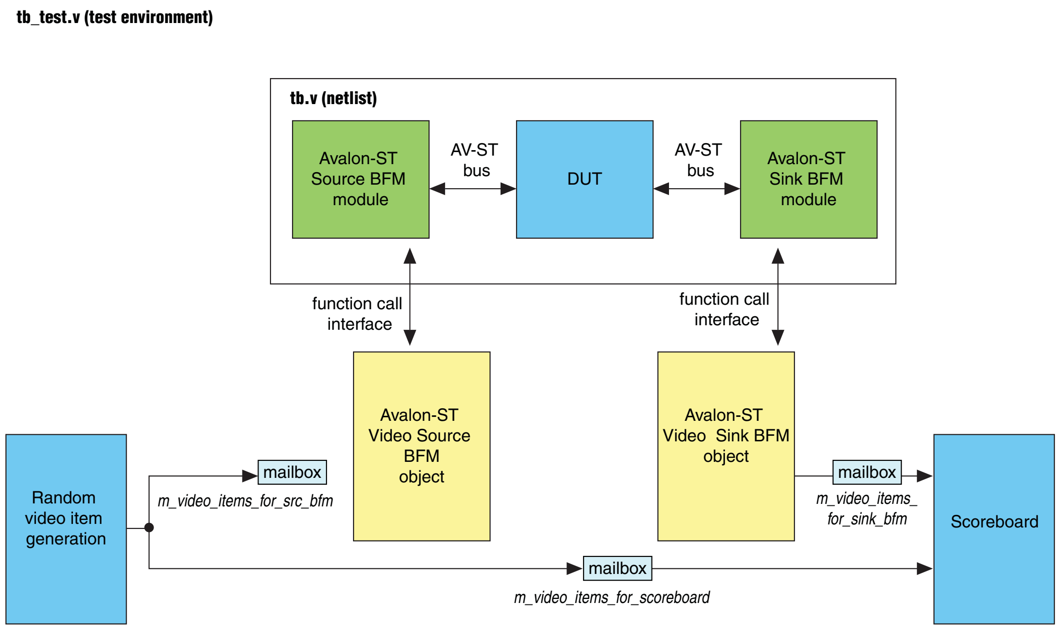
In Stage 7, an end of packet (EOP) is seen in the incoming transactions, the video sink BFM casts the video data into a `c_av_st_video_item` object, and transfers the data into its mailbox. The file writer then receives the video item.

Lastly, in Stage 8, the file writer recasts the video item to a video data packet, pops off the pixel, and writes the data to the output file as binary data.

Constrained Random Test

This section describes the function of a constrained random test. The test is easily assembled using the class library. Figure A–7 shows the constrained random test environment structure.

Figure A–7. Example of a Constrained Random Test Environment


tb_test.v (test environment)
tb.v (netlist)
Avalon-ST Source BFM module
AV-ST bus
DUT
AV-ST bus
Avalon-ST Sink BFM module
function call interface
function call interface
Avalon-ST Video Source BFM object
Avalon-ST Video Sink BFM object
Random video item generation
mailbox
m_video_items_for_src_bfm
mailbox
m_video_items_for_sink_bfm
Scoreboard
mailbox
m_video_items_for_scoreboard

In Figure A–7, randomized video as well as control and user packets are generated using the SystemVerilog's built-in constrained random features. The DUT processes the video packets and the scoreboard determines a test pass or fail result.

Test Environment

Example A–5 shows the code for constrained random generation.

Example A–5. Constrained Random Generation

```
fork

`SOURCE.start();
`SINK.start();

    forever
    begin

        // Randomly determine which packet type to send :
        r = $urandom_range(100, 0);

        if (r>67)
        begin
            video_data_pkt1.set_max_length(100);
            video_data_pkt1.randomize();
            video_data_pkt1.populate();

            // Send it to the source BFM :
            m_video_items_for_src_bfm.put(video_data_pkt1);

            // Copy and send to scoreboard :
            video_data_pkt2 = new();
            video_data_pkt2.copy(video_data_pkt1);
            m_video_items_for_scoreboard.put(video_data_pkt2);
        end

        else if (r>34)
        begin
            video_control_pkt1.randomize();
            m_video_items_for_src_bfm.put(video_control_pkt1);

            // Copy and send to scoreboard :
            video_control_pkt2 = new();
            video_control_pkt2.copy(video_control_pkt1);
            m_video_items_for_scoreboard.put(video_control_pkt2);
        end

        else
        begin
            user_pkt1.set_max_length(33);
            user_pkt1.randomize() ;
            m_video_items_for_src_bfm.put(user_pkt1);

            // Copy and send to scoreboard :
            user_pkt2 = new();
            user_pkt2.copy(user_pkt1);
            m_video_items_for_scoreboard.put(user_pkt2);
        end

        // Video items have been sent to the DUT and the scoreboard,
        //wait for the analysis :
        -> event_constrained_random_generation;
        wait(event_dut_output_analyzed);

    end

join
```

The code in Example A–5 starts the source and sink, then randomly generates either a video data, control or user packet. Generation is achieved by simply calling randomize() on the objects previously created at the end of the code in Example A–1, putting the objects in the source BFM's mailbox (`m_video_items_for_src_bfm`), making a copy of the objects, and putting that in a reference mailbox used by the scoreboard (`m_video_items_for_scoreboard`).

Finally, the code signals to the scoreboard that a video item has been sent and waits for the output of the DUT to be analyzed, also signalled by an event from the scoreboard.

All that remains now is to create the scoreboard, which retrieves the video item objects from the two scoreboard mailboxes and compares the ones from the DUT with the reference objects.

☞ The scoreboard expects to see the DUT returning greyscale video data. You must customize the data to mirror the behavior of individual DUTs exactly.

Example A–6 shows the codes for the scoreboards.

Example A–6. Scoreboard

```
c_av_st_video_item ref_pkt;
c_av_st_video_item dut_pkt;

initial
begin

    forever
    begin


        @event_constrained_random_generation
        begin

            // Get the reference item from the scoreboard mailbox :
            m_video_items_for_scoreboard.get(ref_pkt);

            // If the reference item is a video packet, then check
            // for the control & video packet response :
            if (ref_pkt.get_packet_type() == video_packet)
            begin

                m_video_items_for_sink_bfm.get(dut_pkt);
                if (dut_pkt.get_packet_type() != control_packet)
                    $fatal(1,"SCOREBOARD ERROR”);

                m_video_items_for_sink_bfm.get(dut_pkt);
                if (dut_pkt.get_packet_type() != video_packet)
                    $fatal(1, "SCOREBOARD ERROR”);

                // A video packet has been received, as expected.
                // Now compare the video data itself :

                dut_video_pkt = c_av_st_video_data'(dut_pkt);

                if (dut_video_pkt.compare (to_grey(c_av_st_video_data'(ref_pkt))))
                    $display("%t Scoreboard match”);
                else
                    $fatal(1, "SCOREBOARD ERROR : Incorrect video packet.\n");

            end

            -> event_dut_output_analyzed;

        end

    end

end

initial
#1000000 $finish;
```

If the reference video item is a `video_packet` type, the scoreboard code shown in Example A–6 receives the reference video item from the scoreboard mailbox. The code then receives two consecutive items from the DUT and checks whether or not these items are a control and video packet. To check that greyscale video is generated the code calls the to_grey function (refer to Example A–7) on the reference video item and calls the `compare()` method. If the items matched, the code returns a **1**. If the items does not matched, the code returns an **0**. Then, the result is output to the display. You can run the test for as long as you have to. In this example, it is 1 us.

Example A–7 shows the code for the `to_grey` function.

Example A–7. Scoreboard Behavioral Model of the DUT

```
// The scoreboard calls a function which models the behaviour of the video algorithm

function c_av_st_video_data to_grey (c_av_st_video_data rgb) ;

    const bit [7:0]   red_factor =  76; // 255 * 0.299
    const bit [7:0] green_factor = 150; // 255 * 0.587;
    const bit [7:0]  blue_factor =  29; // 255 * 0.114;

    c_av_st_video_data            grey;
    c_pixel                   rgb_pixel;
    c_pixel                  grey_pixel;
    int                      grey_value;

    grey = new ();
    grey.packet_type = video_packet;

    do
    begin
        grey_pixel = new();
        rgb_pixel = rgb.pop_pixel();

        // Turn RGB into greyscale :
        grey_value = (  red_factor * rgb_pixel.get_data(2) +
                      green_factor * rgb_pixel.get_data(1) +
                       blue_factor * rgb_pixel.get_data(0));

        grey_pixel.set_data(2, grey_value[15:8]);
        grey_pixel.set_data(1, grey_value[15:8]);
        grey_pixel.set_data(0, grey_value[15:8]);
        grey.push_pixel(grey_pixel);
    end
    while (rgb.get_length()>0);

    return grey;

endfunction
```

`to_grey` function reads each pixel in turn from the RGB `video_packet` object, calculates the greyscale value, writes the value to each channel of the outgoing pixel, and pushes that on to the returned `video_packet` object, grey.

A complete test would set up functional coverpoints in the DUT code and use the SystemVerilog's `get_coverage()` call to run the test until the required amount of coverage has been seen. The Avalon-ST BFM monitors (available in Qsys) include a comprehensive set of functional coverage points (in the **altera_avalon_st_monitor_coverage.sv** file) that can add an Avalon-ST protocol coverage to any user DUT coverage, greatly facilitating powerful constrained random test.

Complete Class Reference

This section provides a detailed description of the method calls implemented by each class:

- "c_av_st_video_control"
- "c_av_st_video_data"
- "c_av_st_video_file_io"
- "c_av_st_video_item"
- "c_av_st_video_source_sink_base"
- "c_av_st_video_sink_bfm_'SINK"
- "c_av_st_video_source_bfm_'SOURCE"
- "c_av_st_video_user_packet"
- "c_pixel"

c_av_st_video_control

The following is the declaration for the *c_av_st_video_control* class:

```
class c_av_st_video_control #(parameter BITS_PER_CHANNEL = 8,
CHANNELS_PER_PIXEL = 3) extends c_av_st_video_item;
```

Table A–3 lists the method calls for the *c_av_st_video_control* class.

Table A–3. Method Calls for the c_av_st_video_control Class

Method Call	Description
`function new();`	Constructor
`function bit compare (c_av_st_video_control r);`	Compares this instance to object r. Returns **1** if identical, **0** if otherwise.
`function bit [15:0] get_width ();`	—
`function bit [15:0] get_height ();`	—
`function bit [3:0] get_interlacing ();`	—
`function t_packet_control get_append_garbage ();`	—
`function int get_garbage_probability ();`	—
`function void set_width (bit [15:0] w);`	—
`function void set_height (bit [15:0] h);`	—
`function void set_interlacing (bit [3:0] i);`	—
`function void set_append_garbage (t_packet_control i);`	Refer to `append_garbage` member.
`function void set_garbage_probability (int i);`	—
`function string info();`	Returns a formatted string containing the width, height and interlacing members.

Table A–4 lists the members of the method calls.

Table A–4. Members of the c_av_st_video_control Class

Members Call	Description
`rand bit[15:0] width;`	—
`rand bit[15:0] height;`	—
`rand bit[3:0] interlace;`	—

Table A–4. Members of the c_av_st_video_control Class

Members Call	Description
`rand t_packet_control append_garbage = off;`	The `append_garbage` control is of type `t_packet_control`, defined as: `typedef enum{on,off,random} t_packet_control;`
`rand int garbage_probability = 50;`	The source BFM uses `garbage_probability` and `append_garbage` to determine whether or not to append garbage beats to the end of the control packets. Garbage beats are generated with a probability of **Garbage_probability%**. When a stream of garbage is being generated, the probability that the stream terminates is fixed in the source BFM at 10%. When garbage is produced, this typically produces around 1 to 30 beats of garbage per control packet.

c_av_st_video_data

The following is the declaration for the *c_av_st_video_data* class:

```
class c_av_st_video_data#(parameter BITS_PER_CHANNEL = 8,
CHANNELS_PER_PIXEL = 3) extends c_av_st_video_item;
```

Table A–5 lists the method calls for the *c_av_st_video_data* class.

Table A–5. Method Calls for the c_av_st_video_data Class

Method Call	Description
`function new();`	Constructor
`function void copy (c_av_st_video_data c);`	Copies object c into this object.
`function bit compare (c_av_st_video_data r);`	Compares this instance to object r. Returns **1** if identical, **0** if otherwise.
`function void set_max_length(int length);`	—
`function int get_length();`	—
`function c_pixel #(BITS_PER_CHANNEL,CHANNELS_PER_PIXEL) pop_pixel();`	Returns a pixel object from the packet in first in first out (FIFO) order.
`function c_pixel #(BITS_PER_CHANNEL,CHANNELS_PER_PIXEL) query_pixel(int i);`	Returns a pixel object from the packet at index i, without removing the pixel.
`function void unpopulate(bit display);`	Pops all pixels from the packet, displaying them if display = 1.
`function void push_pixel(c_pixel #(BITS_PER_CHANNEL, CHANNELS_PER_PEXEL)pixel);`	Pushes a pixel into the packet.

Table A–6 lists the members of the method calls.

Table A–6. Members of the c_av_st_video_data Class

Members	Description
`c_pixel #(BITS_PER_CHANNEL,CHANNELS_PER_PIXEL)pixels [$];`	The video data is held in a queue of pixel objects.
`c_pixel #(BITS_PER_CHANNEL,CHANNELS_PER_PIXEL) pixel, new_pixel, r_pixel;`	Pixel objects used for storing intermediate data.
`rand int video_length;`	The length of the video packet (used for constrained random generation only).
`int video_max_length = 10;`	Maximum length of video packet (used for constrained random generation only).

c_av_st_video_file_io

The following is the declaration for the *c_av_st_video_file_io* class:

```
class c_av_st_video_file_io#(parameter BITS_PER_CHANNEL = 8,
CHANNELS_PER_PIXEL = 3);
```

Table A–7 lists the method calls for the *c_av_st_video_file_io* class.

Table A–7. Method Calls for the c_av_st_video_file_io Class (Part 1 of 3)

Method Call	Description
`function void set_send_control_packets(t_packet_controls);`	If this method is used to set the `send_control_packet` control to **off**, then one control packet is sent at the beginning of video data, but no further control packets are sent.
`function t_packet_control get_send_control_packets();`	—
`function void set_send_user_packets(t_packet_control s);`	If the `send_user_packets` control is **off**, no user packets at all are sent. Otherwise, user packets are sent before and after any control packets.
`function t_packet_control get_send_user_packets();`	—
`function void set_send_early_eop_packets(t_packet_control s);`	If the `send_eop_packets` control is **off**, all packets are of the correct length (or longer). Otherwise, early EOP are sent of a length determined by the constraints on `early_eop_packet_length`.
`function t_packet_control get_send_early_eop_packets();`	—
`function void set_early_eop_probability(int s);`	If the `send_early_eop_packets` control is set to **random**, the `early_eop_probability` control determines what proportion of video packets are terminated early.
`function int get_early_eop_probability();`	—
`function void set_send_late_eop_packets(t_packet_controls);`	If the `send_late_eop_packets` control is **off**, all packets are of the correct length (or longer). Otherwise, late EOP are sent of a length determined by the constraints on `late_eop_packet_length`.

Table A–7. Method Calls for the c_av_st_video_file_io Class (Part 2 of 3)

Method Call	Description
`function t_packet_control get_send_late_eop_packets();`	—
`function void set_late_eop_probability (int s);`	If the `send_late_eop_packets` control is set to **random,** the `late_eop_probability` control determines what proportion of video packets are terminated late.
`function int get_late_eop_probability ();`	—
`function void set_user_packet_probability (int s);`	If the `send_user_packets` is set to **random,** the `user_packet_probability` control determines the probability that a user packet being sent before a control packet. It also determines the probability that a user packet will be sent after a control packet.
`function int get_user_packet_probability ();`	—
`function void set_control_packet_probability(int s);`	If the `send_control_packets` control is set to **random**, the `control_packet_probability` control determines the probability of a control packet being sent before a video packet.
`function int get_control_packet_probability();`	—
`function void set_send_garbage_after_control_packets (t_packet_control s);`	When the `send_control_packet()` method puts a control packet into the `m_video_item_out` mailbox, the `append_garbage` member of the control packet object is set to the value of `send_garbage_after_control_packets`.
`function t_packet_control get_send_garbage_after_control_packets();`	—
`function void set_object_name(string s);`	You can use **object_name** to name a given object instance of a class to ensure any reporting that the class generates is labelled with the originating object's name.
`function string get_object_name();`	—
`function string get_filename();`	This returns the filename associated with the object, by the `open_file` call.
`function void set_image_height(bit[15:0]height);`	—
`function bit[15:0]get_image_height();`	—
`function void set_image_width(bit[15:0] width);`	—
`function bit[15:] get_image_width();`	—
`function void set_video_data_type(string s);`	Sets the `fourcc[3]` code associated with the raw video data. The following are the supported four character code (FOURCC) codes: ■ `RGB32` ■ `IYU2` ■ `YUY2` ■ `Y410` ■ `A2R10GB10` ■ `Y210`
`function string get_video_data_type();`	Returns the FOURCC code (for example, `RGB32`) being used for the raw video data.

Table A–7. Method Calls for the c_av_st_video_file_io Class (Part 3 of 3)

Method Call	Description
`function int get_video_packets_handled();`	—
`function int get_control_packets_handled();`	—
`function int get_user_packets_handled();`	—
`function new(mailbox #(c_av_st_video_item)m_vid_out);`	Constructor. The mailbox is used to pass all packets in and out of the file I/O object.
`function void open_file(string fname, t_rwrw);`	Files are opened using this method. For example: `video_file_reader.open_file(`"`vip_car_0.bin`", read); `t_rw` is an enumerated type with values read or write. NB. The read fails if there is no associated **.spc** file, for example, **vip_car_o.spc**).
`function void close_file();`	For example, `video_file_reader.close_file();`
`task read_file();`	`Read_file()` optionally calls `send_user_packet()` and `send_control_packet()`, then calls `read_video_packet()`.
`task send_control_packet();`	The control packet sent is derived from the image height, width, and interlace fields as provided by `open_file()`.
`task send_user_packet();`	The user packet sent is always comprised of random data and had a maximum length hard-coded to 33 data items.
`task_generate_spc_file();`	When writing a file, this call creates the necessary associated **.spc** file.
`task read_video_packet();`	The main file reading method call. Binary data is read from the file and packed into pixel objects according to the settings of `ycbr_pixel_order` and endianism. Pixel objects are packed into a video data object, with some pixels optionally added or discarded if late/early EOP is being applied. When one complete field of video has been read (as determined by the height and width controls), the `video_data` object is put in the mailbox.
`task wait_for_and_write_video_packet_to_file();`	When called, this method waits for an object to be put in the mailbox (usually from a sink BFM). When a control or a user packet object arrives, this call is reported and ignored. When a video packet arrives, the video data is written to the open file in little endianism format.

Table A–8 lists the members of the method calls.

Table A–8. Members of the c_av_st_video_file_io Class (Part 1 of 2)

Members	Description
`local int video_packets_handled = 0;`	`Video_packets_handled` is added whenever a packet is read or written to or from the file.
`local int control_packets_handled = 0;`	`control_packets_handled` is added whenever a control packet is put in the object's mailbox.
`local int user_packets_handled = 0;`	`user_packets_handled` is added whenever a user packet is put in the object's mailbox.
`local reg[15:0] image_height;`	—

Table A–8. Members of the c_av_st_video_file_io Class (Part 2 of 2)

Members	Description
`local reg[15:0] image_width;`	—
`local reg[3:0] image_interlaced;`	—
`string image_fourcc;`	—
`local string object_name = "file_io";`	—
`local string filename;`	—
`local string spc_filename;`	—
`local int fourcc_channels_per_pixel;`	Set when the associate **.spc** file is read.
`local int fourcc_bits_per_channel;`	Set when the associate **.spc** file is read.
`local int fourcc_pixels_per_word;`	Set when the associate **.spc** file is read.
`local int fourcc_channel_lsb;`	Set when the associate **.spc** file is read.
`int early_eop_probability = 20;`	—
`Int late_eop_probability = 20;`	—
`int user_packet_probability = 20;`	—
`int control_packet_probability = 20;`	—
`mailbox #(c_av_st_video_item) m_video_item_out = new(0);`	The mailbox is used to pass all packets in/out of the file i/o object.
`rand t_packet_control send_control_packets = on;`	—
`rand t_packet_control send_user_packets = off;`	—
`rand t_packet_control send_early_eop_packets = off;`	—
`rand t_packet_control send_late_eop_packets = off;`	If both `send_late_eop_packets` and `send_early_eop_packets` are set to random, a late EOP will only be generated if an early EOP has not been.
`rand t_packet_control send_garbage_after_control_packets = off;`	—
`rand int early_eop_packet_length = 20;`	`constraint early_eop_length {` `early_eop_packet_length dist {1:= 10, [2:image_height*image_width-1]:/90};` `early_eop_packet_length inside {[1:image_height*image_width]};` `}`
`rand int late_eop_packet_length = 20;`	`constraint late_eop_length {` `late_eop_packet_length inside {[1:100]};` `}`

c_av_st_video_item

The following is the declaration for the *c_av_st_video_item* class:

```
class c_av_st_video_item;
```

Table A–9 lists the method calls for the *c_av_st_video_item* class.

Table A–9. Method Calls for the c_av_st_video_item Class

Method Call	Description
`function new ();`	Constructor
`function void copy (c_av_st_video_item c);`	Sets **this.packet_type** to match that of c.
`function void set_packet_type (t_packet_types ptype);`	—
`function t_packet_typesget_packet_type();`	—

Table A–10 lists the members of the method calls for the *c_av_st_video_item* class.

Table A–10. Members of the c_av_st_video_item Class

Method	Description
`t_packet_types packet_type;`	`Packet_type` must be one of the following: ■ `video_packet` ■ `control_packet` ■ `user_packet` ■ `generic_packet` ■ `undefined`

c_av_st_video_source_sink_base

The following is the declaration for the *c_av_st_video_source_sink_base* class:

```
class c_av_st_video_source_sink_base;
```

Table A–11 lists the method calls for the *c_av_st_video_source_sink_base* class.

Table A–11. Method Calls for the c_av_st_video_source_sink_base Class

Method Call	Description
`function new(mailbox #(c_av_st_video_item)m_vid);`	Constructor. The video source and sink classes transfer video objects through their mailboxes.
`function void set_readiness_probability(int percentage);`	—
`function int get_readiness_probability();`	—
`function void set_long_delay_probability(real percentage);`	—
`function real get_long_delay_probability();`	—
`function void set_long_delay_duration_min_beats(int percentage);`	—
`function int get_long_delay_duration_min_beats();`	—
`function void set_long_delay_duration_max_beats(int percentage);`	—
`function int get_long_delay_duration_max_beats();`	—
`function void set_pixel_transport(t_pixel_format in_parallel);`	—
`function t_pixel_format get_pixel_transport();`	—
`function void set_name(string s);`	—
`function string get_name();`	—

Table A–12 lists the members of the method calls for the *c_av_st_video_source_sink_base* class.

Table A–12. Members of the c_av_st_video_source_sink_base Class

Member	Description
`mailbox # (c_av_st_video_item) m_video_items= new(0);`	—
`t_pixel_format pixel_transport = parallel;`	The Avalon-ST video standard allows you to send symbols in serial or parallel format. You can set this control to either format.
`string name = "undefined";`	—
`int video_packets_sent =    0;`	—
`int control_packets_sent =    0;`	—
`int user_packets_sent =    0;`	—
`int readiness_probability =   80;`	Determines the probability of when a sink or source is ready to receive or send data in any given clock cycle, as manifested on the bus by the `READY` and `VALID` signals, respectively.

Table A–12. Members of the c_av_st_video_source_sink_base Class

Member	Description
`real long_delay_probability = 0.01;`	The `readiness_probability` control provides a 'steady state' readiness condition. The `long_delay_probability` allows for the possibility of a much rarer and longer period of unreadiness, of durations of the order of the raster line period or even field period.
`rand int long_delay_duration_min_beats= 100;`	This control sets the minimum duration (as measured in data beats (1)) of a long delay.
`rand int long_delay_duration_max_beats = 1000;`	This control sets the maximum duration (as measured in data beats) of a long delay.
`rand int long_delay_duration =   80;`	`constraint c1 {long_delay_duration inside [long_delay_duration_min_beats: long_delay_duration_max_beats]};}`

Note to Table A–12:

(1) If `pixel_transport` = parallel than one data beats = one pixel = one clock cycle.

c_av_st_video_sink_bfm_'SINK

The following is the declaration for the *c_av_st_video_sink_bfm_'SINK* class:

```
'define CLASSNAME c_av_st_video_sink_bfm_'SINK;
 class 'CLASSNAME extends c_av_st_video_source_sink_base;
```

Table A–13 lists the method calls for the *c_av_st_video_sink_bfm_'SINK* class.

☞ This class has no additional members to those of the base class.

Table A–13. Method Calls for the c_av_st_video_source_sink_base Class

Method Call	Description
`function new(mailbox#(c_av_st_video_item)m_vid);`	Constructor.
`task start;`	The start method simply waits until the reset of the Avalon-ST sink BFM goes inactive, then calls `receive_video()`.
`task receive_video;`	The `receive_video` task continually drives the Avalon-ST sink BFM's `ready` signal in accordance with the probability settings in the base class. It also continually captures `signal_received_transaction` events from the Avalon-ST sink BFM and uses the Avalon-ST sink BFM API to read bus data. Bus data is decoded according to the Avalon-ST video specification and data is packed into an object of the appropriate type (video, control or, user). The object is then put into the mailbox.

c_av_st_video_source_bfm_'SOURCE

The following is the declaration for the *c_av_st_video_source_bfm_'SOURCE* class:

```
`define CLASSNAME c_av_st_video_source_bfm_'SOURCE
 class `CLASSNAME extends c_av_st_video_source_sink_base;
```

Table A–14 lists the method calls for the *c_av_st_video_source_bfm_'SOURCE* class.

This class has no additional members to those of the base class.

Table A–14. Method Calls for the c_av_st_video_source_bfm_'SOURCE Class

Method Call	Description
`function new(mailbox#(c_av_st_video_item)m_vid);`	Constructor.
`task start;`	The start method simply waits until the reset of the Avalon-ST source BFM goes inactive, then continually calls the `send_video()` task.
`task send_video;`	The `send_video()` task waits until a video item is put into the mailbox, then it drives the Avalon-ST sink BFM's API accordingly. The `set_transaction_idles()` call is used to set the valid signal in accordance with the probability settings in the base class. The mailbox object is categorized according to object type. Each object is presented to the bus according to the Avalon-ST Video specification and the setting of the `pixel_transport` control.

c_av_st_video_user_packet

The following is the declaration for the *c_av_st_video_user_packet* class:

```
class c_av_st_video_user_packet#(parameters BITS_PER_CHANNEL=8,
CHANNELS_PER_PIXEL=3) extends c_av_st_video_item;
```

Table A–15 lists the method calls for the *c_av_st_video_user_packet* class.

Table A–15. Method Calls for the c_av_st_video_user_packet Class (Part 1 of 2)

Method Call	Description
`function new();`	Constructor.
`function void copy (c_av_st_video_user_packet c);`	Copies object c into this object.
`function bit compare (c_av_st_video_user_packet r);`	Compares this instance to object r. Returns **1** if identical, **0** for otherwise.
`function void set_max_length(int l);`	For constrained random generation, this method is used to apply a maximum length to the user packets.
`function int get_length();`	—
`function bit[3:0] get_identifier();`	The identifier is the Avalon-ST video packet identifier. 0x0 indicates video, 0xf indicates a control packet and the user packets take random values from 0x4 to 0xe.
`function bit [BITS_PER_CHANNEL*CHANNELS_PER_PIXEL-1:0] pop_data();`	Returns the next beat of user data.

Table A–15. Method Calls for the c_av_st_video_user_packet Class (Part 2 of 2)

Method Call	Description
`function bit [BITS_PER_CHANNEL*CHANNELS_PER_PIXEL-1:0] query_data(int i);`	Returns the next beat of user data without removing it from the object.
`function void push_data(bit [BITS_PER_CHANNEL*CHANNELS_PER_PIXEL-1:0] d);`	—

Table A–16 lists the members of the method calls for the *c_av_st_video_user_packet* class.

Table A–16. Members of the c_av_st_video_user_packet Class

Member	Description
`rand bit[BITS_PER_CHANNEL*CHANNELS_PER_PIXEL-1:0]data[$]`	User data is stored as a queue of words.
`rand bit[3:0] identifier;`	constraint c2 {identifier inside {[4:14]};}
`int max_length = 10;`	constraint c1 {data.size() inside {[1:max_length]};}

c_pixel

The following is the declaration for the *c_pixel* class:

```
class c_pixel#(parameters BITS_PER_CHANNEL=8, CHANNELS_PER_PIXEL=3);
```

Table A–17 lists the method calls for the *c_pixel* class.

Table A–17. Method Calls for the c_pixel Class

Method	Description
`function new();`	Constructor.
`function void copy(c_pixel #(BITS_PER_CHANNEL, CHANNELS_PER_PIXEL) pix);`	Copies object pixel into this object.
`function bit[BITS_PER_CHANNEL-1:0] get_data(int id);`	Returns pixel data for channel id.
`function void set_data(int id, bit [BITS_PER_CHANNEL-1:0] data);`	Sets pixel data for channel id.

Raw Video Data Format

Altera provides and recommends two Microsoft Windows utilities for translating between **.avi** and **.raw** file formats. Example A–8 shows ways to convert an **.avi** file.

Example A–8. Convert .raw to .avi file

```
>avi2raw.exe vip_car_0.avi vip_car_0.raw

"dshow" decoder created
Information on the input file:
filename: vip_car_0.avi
fourcc = ARGB32
width = 160
height = 120
stride = 640
inversed = yes
endianness = little_endian
frame_rate = 25
Choose output dimensions, 0 to keep original values (this will apply a crop/pad,
 not a scaling):
width (160) = 0
height (120) = 0
Choose output colorspace:
1.RGB
2.YCbCr
3.MONO
1
Choose output bps (8, 10)
8
Choose output interlacing:
1.progressive
2.interlaced F0 first
3.interlaced F1 first
1
Choose a number of frames to skip at the start of the sequence (use 0 to start t
he decoding from the beginning) :
0
Choose the maximum number of frames to decode (use 0 to decode up to the end of
the video) :
0
"raw" encoder created
Decoding in progress.......
7 fields extracted
deleting decoding chain
deleting encoder
press a key to exit
1
```

Example A–9 shows ways to produce a **.raw** file together with a **.spc** file that contains the FOURCC information:

Example A–9. .raw and .spc Files

```
> more vip_car_0.spc

fourcc = RGB32
width = 160
height = 120
stride = 640
frame_rate = 25
```

To decode the data, the file I/O class reader must see both the **.raw** and **.spc** files. The file I/O class reader writes a **.raw**/**.spc** file pair that you can view using the **.avi** encoder utility, as shown in Example A–10.

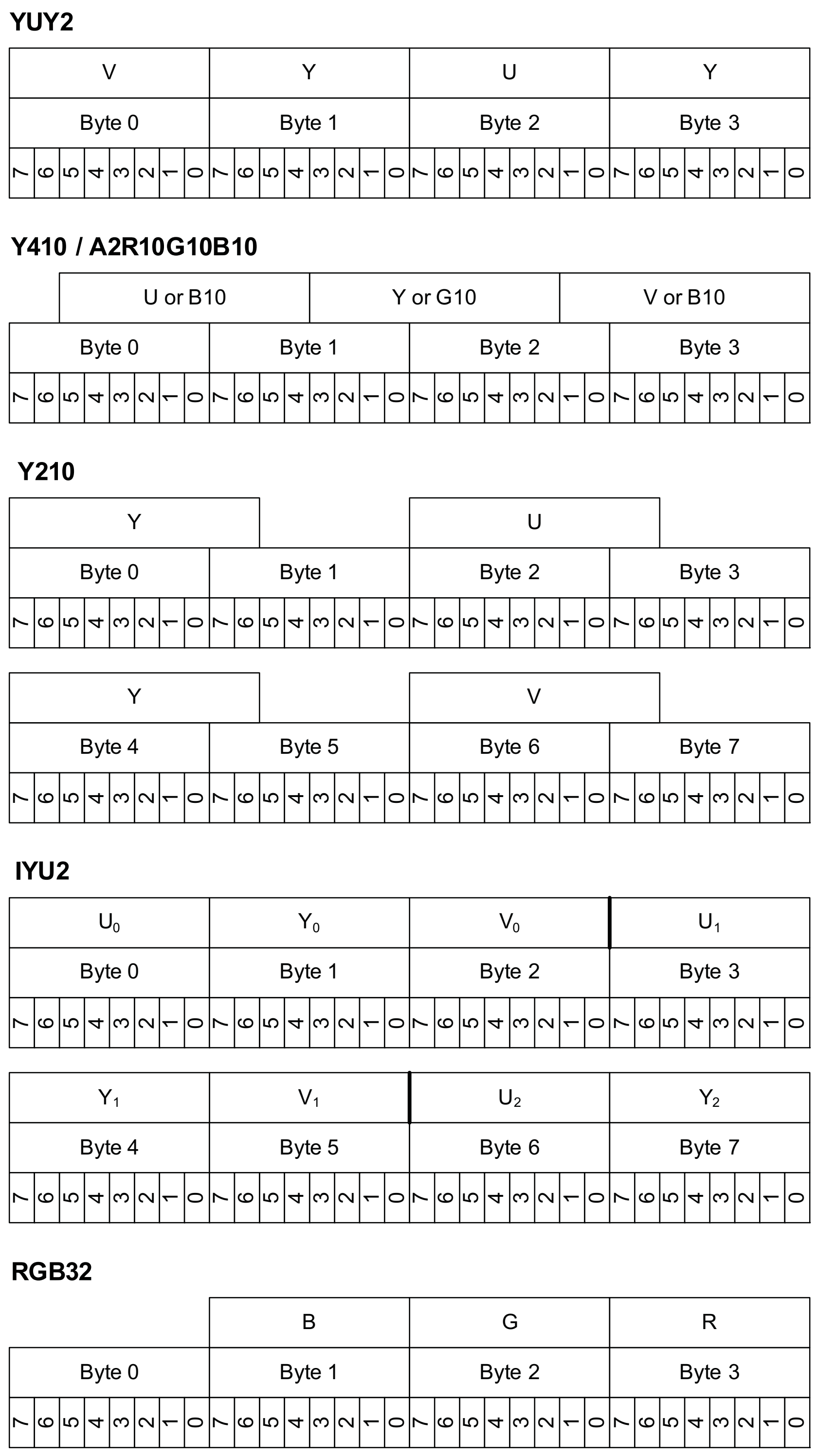
Example A–10. .raw and .spc Files Pairing

```
> raw2avi.exe vip_car_1.raw vip_car_1.avi
"raw" decoder created
vip_car_1.raw:
RGB32 160*120, progressive, 25fps
"dshow" encoder created
AVI encoder created
Encoding in progress.......
7 frames encoded
deleting conversion chain
deleting encoder
press a key to exit
1
```

If there is no Windows machine available to run the utilities, you must then provide the video data in the correct format by some other means.

Example A–11 shows and describes the data format required by the file I/O class for each of the supported FOURCC codes.

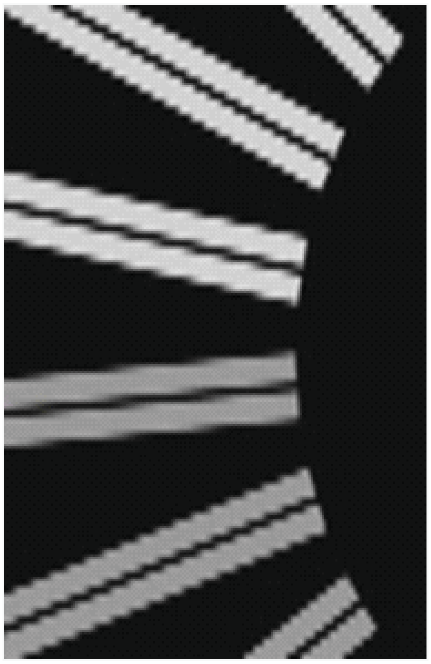
Example A–11. Supported FOURCC Codes and Data Format



B. Choosing the Correct Deinterlacer

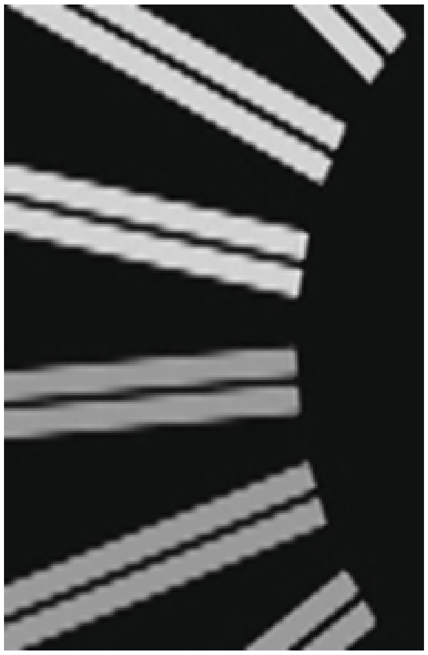
Figure B–1 through Figure B–3 show some example output from the various deinterlacer options available. The lowest quality is a simple Bob deinterlacer. The highest quality is motion adaptive high quality (HQ) setting in the Deinterlacer II MegaCore function.

Figure B–1. Bob Deinterlacing Option



To enable this option, open the Deinterlacer MegaWizard Plug-In Manager. In the Parameter Settings tab, under the **Behavior** parameter, select **Bob Scanline Interpolation** from the **Deinterlacing method** drop-down option. With this moving "Dial" test sequence, a Bob deinterlace produces the characteristic staircasing effect on the edges of these diagonal lines. The area is 454 look-up tables (LUTs).

Figure B–2. Deinterlacer I and Deinterlacer II Motion Adaptive

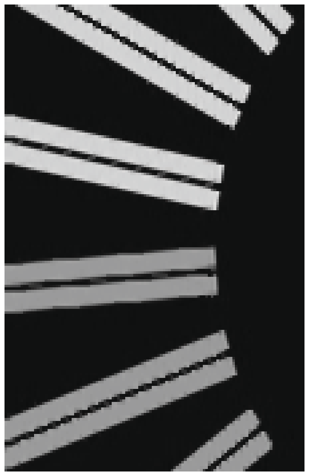


To enable this option in the Deinterlacer MegaCore function, open the Deinterlacer MegaWizard Plug-In Manager. In the Parameter Settings tab, under the **Behavior** parameter, select **Motion Adaptive** from the **Deinterlacing method** drop-down option and select **Triple buffering with rate conversion** from the **Frame buffering mode** drop-down option.

If you are using the Deinterlacer II MegaCore function, in the Parameter Settings tab, under the **Behavior** parameter, select **Motion Adaptive** from the **Deinterlace Algorithm** drop-down option.

With the moving “Dial” test sequence, a motion-adaptive interlacer detects the motion, preventing incoming artifacts that would otherwise appear. The motion-adaptive interlacer performs a Bob interpolation and operates on a 3x3 kernel of pixels, therefore has the ability to interpolate along the diagonal and reduce the staircasing effect. The area is 5,188 LUTs for the Deinterlacer MegaCore function and 3, 696 LUTs for the Deinterlacer II MegaCore function.

Figure B–3. Deinterlacer II Motion Adaptive High Quality



To enable this option, open the Deinterlacer II MegaWizard Plug-In Manager. In the **Parameter Settings** tab, under the **Behavior** parameter, select **Motion Adaptive High Quality** from the **Deinterlacing method** drop-down option.

With the moving "Dial" test sequence, the motion adaptive HQ mode of the Deinterlacer II MegaCore function improves the Bob interpolation further by operating on a 17x3 kernel of pixels. This allows much lower angles to be detected and interpolated, eliminating the staircasing effect almost completely. The area is 8,252 LUTs.

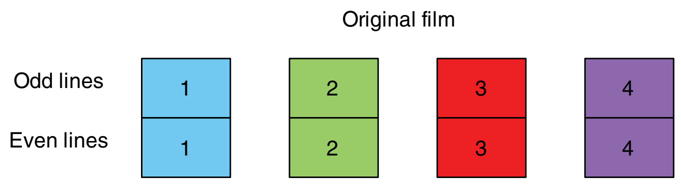
Cadence Detection and Reverse Pulldown in the Deinterlacer II MegaCore Function—In Depth

The deinterlacer is compile-time configurable to implement a 3:2 or 2:2 cadence detect mode.

Cadence detection and reverse pulldown only applies to the Deinterlacer II MegaCore function.

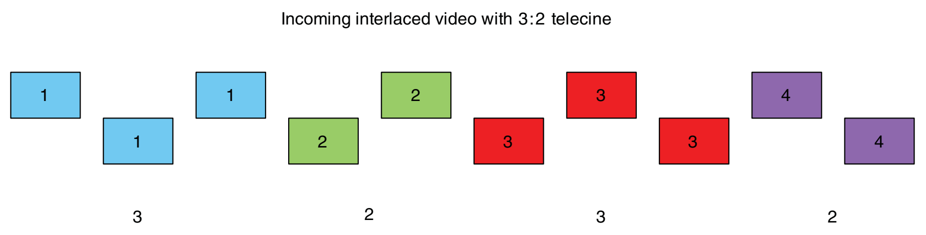
When images from film are transferred to NTSC or PAL for broadcast (which requires frame-rate conversion), a cadence is introduced into the interlaced video. Consider four frames from a film where each frame is shown split into odd and even fields, as shown in Figure B–4:

Figure B–4. Odd and Even Fields



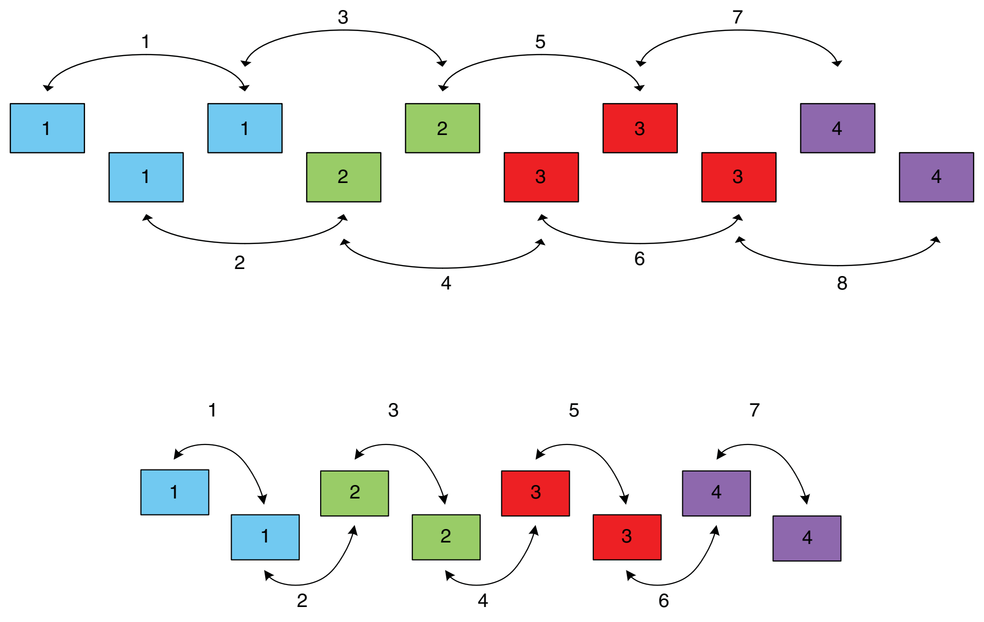
For broadcast, a telecine is applied to minimize artifacts due to the rate conversion, which introduces a 3:2 cadence to the fields, as shown in Figure B–5:

Figure B–5. Incoming Interlaced Video



A video sequence such as Figure B–5 may be correctly handled by detecting the cadence and reconstructing (reverse pulldown) the original film. You can achieve this by comparing each field with the preceding field of the same type (3:2 detection) or detecting possible comb artifacts that occurs if weaving two consecutive fields (2:2 detection), as shown in Figure B–6:

Figure B–6. 3:2 Detection and 2:2 Detection Comparison

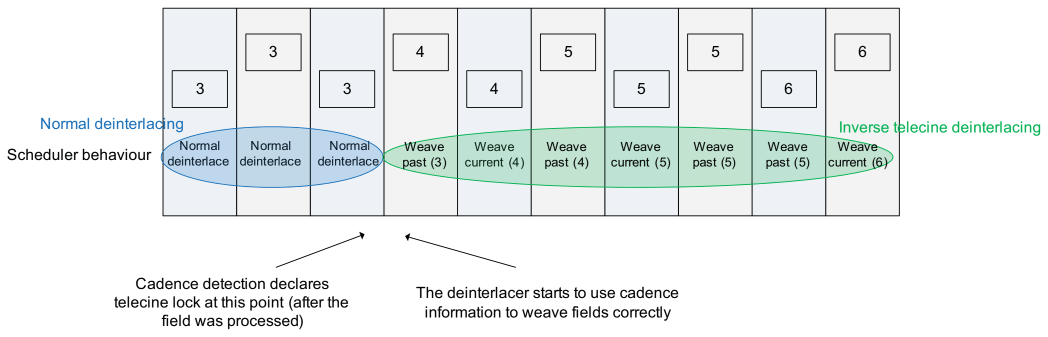


The 3:2 cadence detector tries to detect matches separated by four mismatches. When this pattern is seen a couple of times, the 3:2 cadence detector locks. The 3:2 cadence detector unlocks after 11 successive mismatches.

After six fields of cadenced video is presented, the 2:2 cadence detector will lock. After three fields of uncadenced data is presented, the 2:2 cadence detector will unlock.

When the cadence detect component enters a lock state, the deinterlacer continuously assembles a coherent frame from the incoming fields, by either weaving the current incoming field with the previous one (shown as weave current in Figure B–7) or by weaving the two past fields together (weave past), as shown in Figure B–7:

Figure B–7. Weave Current and Weave Past



If the incoming video contains any cadenced video, you must enable the **Cadence detection and reverse pulldown** option from the **Behavior** parameter in the Parameter Settings tab. Then, select **the cadence detection algorithm** according to the type of content you are expecting. If the incoming video will contain both 3:2 and 2:2 cadences, select **3:2 & 2:2** detector.

The cadence detection algorithms are also designed to be robust to false-lock scenarios—for example, features on adjacent fields may trick other detection schemes into detecting a cadence where there is none.

Additional Information

This chapter provides additional information about the document and Altera.

Document Revision History

The following table lists the revision history for this document.

Date	Version	Changes
July 2012	12.0	■ Added "Avalon-ST Video Monitor MegaCore Function"and "Trace System MegaCore Function" chapters. ■ Added information on the edge-adaptive scaling algorithm feature in the "Scaler II MegaCore Function" chapter.
February 2012	11.1	■ Reorganized the user guide. ■ Added "Avalon-ST Video Verification IP Suite" and "Choosing the Correct Deinterlacer" appendixes. ■ Updated Table 1–1 and Table 1–3.
May 2011	11.0	■ Added Deinterlacer II MegaCore function. ■ Added new polyphase calculation method for Scaler II MegaCore function. ■ Final support for Arria II GX, Arria II GZ, and Stratix V devices.
January 2011	10.1	■ Added Scaler II MegaCore function. ■ Updated the performance figures for Cyclone IV GX and Stratix V devices.
July 2010	10.0	■ Preliminary support for Stratix V devices. ■ Added Interlacer MegaCore function. ■ Updated Clocked Video Output and Clocked Video Input MegaCore functions to insert and extract ancillary packets.
November 2009	9.1	■ Added new Frame Reader, Control Synchronizer, and Switch MegaCore functions. ■ The Frame Buffer MegaCore function supports controlled frame dropping or repeating to keep the input and output frame rates locked together. The Frame Buffer also supports buffering of interlaced video streams. ■ The Clipper, Frame Buffer, and Color Plane Sequencer MegaCore functions now support four channels in parallel. ■ The Deinterlacer MegaCore function supports a new 4:2:2 motion-adaptive mode and an option to align read/write bursts on burst boundaries. ■ The Line Buffer Compiler MegaCore function has been obsoleted. ■ The Interfaces chapter has been re-written.
March 2009	8.0	■ The Deinterlacer MegaCore function supports controlled frame dropping or repeating to keep the input and output frame rates locked together. ■ The Test Pattern Generator MegaCore function can generate a user-specified constant color that can be used as a uniform background. ■ Preliminary support for Arria II GX devices.

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact [(1)]	**Contact Method**	**Address**
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Nontechnical support (general)	Email	nacomp@altera.com
(software licensing)	Email	authorization@altera.com

Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table lists the typographic conventions this document uses.

Visual Cue	**Meaning**
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, <*file name*> and <*project name*>**.pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections in a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix `n` denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
↵	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
☞	The hand points to information that requires special attention.

Visual Cue	Meaning
?	The question mark directs you to a software help system with related information.
👣	The feet direct you to another document or website with related information.
🎥	The multimedia icon directs you to a related multimedia presentation.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
⚡ WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
✉	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.
💬	The feedback icon allows you to submit feedback to Altera about the document. Methods for collecting feedback vary as appropriate for each document.