AXI4-Stream Video IP and System Design Guide

UG934 October 5, 2016

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/05/2016	2.2	Updated Dynamic TDATA Configuration section with additional examples (RAW14 and RGB888).
04/06/2016	2.2	Updated for Encoding section.
09/30/2015	2.2	Updated pixel alignment in the Dynamic TDATA Configuration section.
04/02/2014	2.1	Updated Introduction chapter for the expansion of the video protocol to multiple pixels transferred over one AXI4-Stream DATA beat.
06/19/2013	2.0	Added Video Subsystem Software Guidelines and Video Subsystem Bandwidth Requirements sections. Removed Core Generator support.
07/25/2012	1.0	Initial Xilinx release.

Table of Contents

Appendix A: Additional Resources and Legal Notices

Introduction

This section summarizes the AXI4-Stream interface Video protocol as fully defined in the *Video IP: AXI Feature Adoption* section of the *AXI Reference Guide* (UG1037).

AXI4-Stream Signaling Interface

The AXI4-Stream carries active video data, driven by both the master and slave interfaces as seen in Figure 1-1.

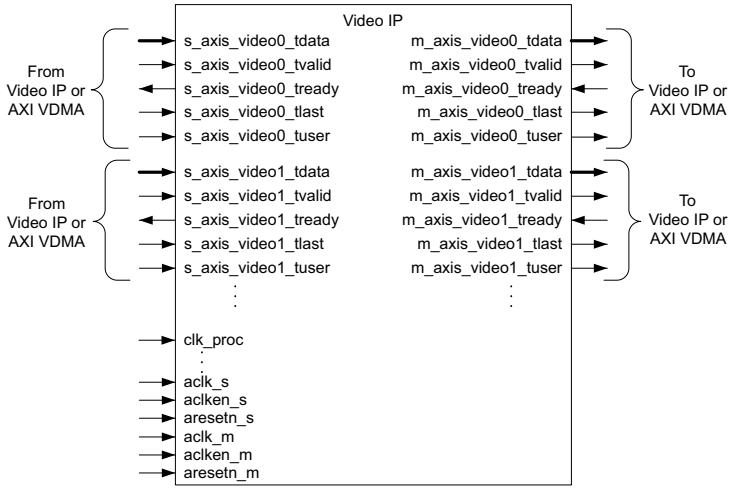

Figure 1-1: **Video IP with Multiple AXI4-Stream Slave (Input) and Master (Output) Interfaces**

Blank periods, audio data, and ancillary data packets are not transferred through the video protocol over AXI4-Stream. All signals listed in Table 1-1 and Table 1-2 are required for video over AXI4-Stream interfaces.

Table 1-1 shows the interface signal names and functions for the input (slave) side connectors. To avoid naming collisions, the signal prefix `s_axis_video` should be appended to `s_axis_video`*k*, for IP with multiple AXI4-Stream input interfaces, where *k* is the index of the respective input AXI4-Stream; for example, `axis_video_tvalid` becomes `s_axis_video0_tvalid` for stream 0 and `s_axis_video1_tvalid` for stream 1.

Table 1-1: **AXI4-Stream Video Protocol Input (Slave) Interface Signals**

Function	Width	Direction	AXI4-Stream Signal Name	Video Specific Name
Video Data	Any number of bytes	In	s_axis_video_tdata	DATA
Valid	1	In	s_axis_video_tvalid	VALID
Ready	1	Out	s_axis_video_tready	READY
Start Of Frame	1	In	s_axis_video_tuser	SOF
End Of Line	1	In	s_axis_video_tlast	EOL

1. InterfaceX Name mandates the top-level IP port names.
2. Video Specific Name should be short, descriptive signal names referring to AXI4-Stream ports that are to be used in HDL code, timing diagrams, and test benches.

Table 1-2 shows the interface signal names and functions for the output (master) side connectors. Similarly, for IP with multiple AXI4-Stream output interfaces, the signal prefix `m_axis_video` should be appended to `m_axis_video`*k*`_`, where *k* is the index of the respective output AXI4-Stream; for example, `axis_video_tvalid` becomes `m_axis_video0_tvalid` for stream 0 and `m_axis_video1_tvalid` for stream 1.

Table 1-2: **AXI4-Stream Video Protocol Output (Master) Interface Signals**

Function	Width	Direction	AXI4-Stream Signal Name	Video Specific Name
Video Data	Any number of bytes	Out	m_axis_video_tdata	DATA
Valid	1	Out	m_axis_video_tvalid	VALID
Ready	1	In	m_axis_video_tready	READY
Start Of Frame	1	Out	m_axis_video_tuser	SOF
End Of Line	1	Out	m_axis_video_tlast	EOL

READY/VALID Handshake

A valid transfer occurs whenever READY, VALID, ACLKEN, and ARESETn signals are High at the rising edge of ACLK, as shown in Figure 1-2.

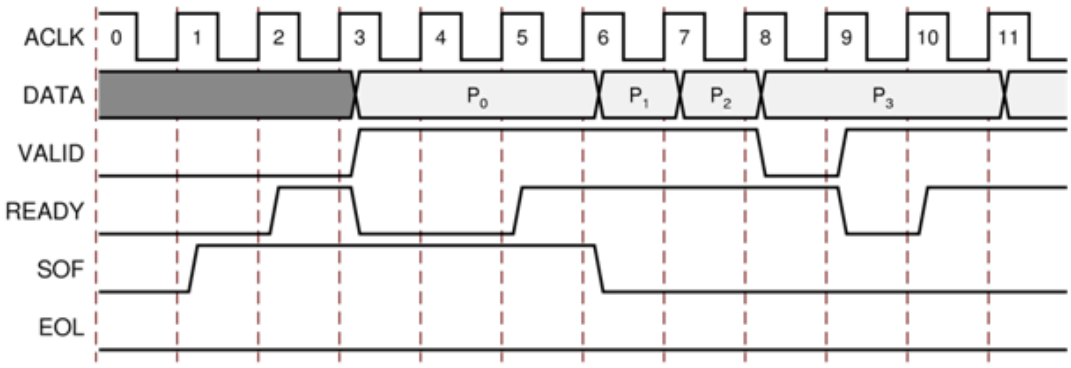

Figure 1-2: **Example of READY/VALID Handshake, Start of a New Frame**

During valid transfers, DATA only carries active video data. Blank periods and ancillary data packets are not transferred by video over AXI4-Stream.

Start of Frame Signal

The start of frame (SOF) signal is physically transmitted over the AXI4-Stream TUSER0 signal, and signifies the first pixel of a video field or frame. The SOF pulse is one valid transaction wide, and must coincide with the first pixel of the field or frame (Figure 1-2). SOF functions as a frame synchronization signal, allowing downstream cores to reinitialize, and detect the first pixel of a field or frame.

End of Line Signal

The end of line (EOL) signal is physically transmitted over the AXI4-Stream TLAST signal, and signifies the last pixel of a line. The EOL pulse is one valid transaction wide, and must coincide with the last pixel of a scan-line (Figure 1-3).

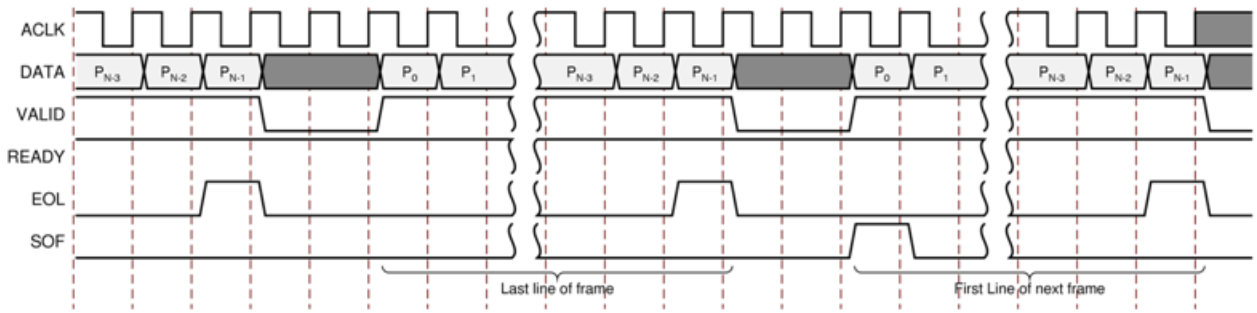

Figure 1-3: **Use of EOL and SOF Signals**

Data Format

To transport video data, the DATA vector encodes logical channel subsets of the physical DATA signals. AXI4-Stream interfaces between video modules can facilitate the transfer of video using different precision (e.g., 8, 10, or 12 bits per color channel), and/or different formats (e.g., RGB or YUV 420) and different number of pixels per data beat.

AXI4-Stream Specific Parameterization

Video IP configuration parameters are described in IP Parameterization in Chapter 3. The specific parameters for the AXI4-Stream interface video protocol are listed in Table 1-3.

Table 1-3: **Video over AXI4-Stream Specific IP Parameters**

Parameter Name	Function
C_*tk*_DATA_WIDTH	Width of color/component data
C_*tk*_VIDEO_FORMAT	Video format code
C_*tk*_AXIS_TDATA_WIDTH	Width of interface signal TDATA
C_*tk*_MAX_SAMPLES_PER_CLOCK	Maximum number of samples/pixels per data beat

The C_*tk*_AXIS_TDATA_WIDTH parameter determines the width of variable-width interface signal TDATA on AXI4-Stream interface *tk*, where interface type *t* can have the values [m,s] designating a master or slave interface, while optional integer *k* specifies the interface ID. Typically, C_*tk*_AXIS_TDATA_WIDTH is a function of the component data width, the number of pixels/samples per data beat, and the number of components the actual video format is using.

The recommended parameter names for component data width is C_*tk*_DATA_WIDTH. The optional format parameter C_*tk*_VIDEO_FORMAT can help the IP determine the number of color components present on DATA using a HDL function. Video IP typically requires specific formats on the input interfaces and can have the number of color component channels hard coded in the IP. However, when C_*tk*_VIDEO_FORMAT (set by a default value on the master interface) is propagated in HDL designs to slave interfaces, the IP source code can perform DRC using assertions to ensure that AXI4-Stream video interfaces are driven by video that was encoded in the expected format.

Encoding

The DATA bits are represented using the [N-1:0] bit numbering convention (N-1 through 0). The components of implicit subfields of DATA should be packed tightly together; for example, a DW=10 bit RGB data packed together to 30 bits. If necessary, the packed data word should be zero padded with most significant bits (MSBs) so the width of the resulting word is an integer that is a multiple of eight as shown in Figure 1-4.

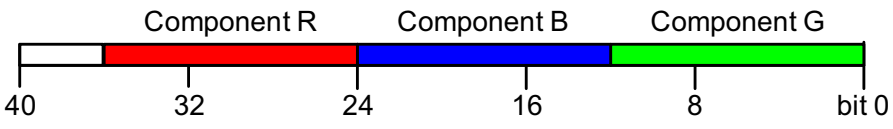

Figure 1-4: **Video Data Padding for TDATA for Multiple Pixels**

The detailed representation of different formats is listed in Table 1-4, with DW = C_DATA_WIDTH and VF = C_VIDEO_FORMAT.

Table 1-4: **Video Format Codes and Data Representation for C_*tk*_MAX_SAMPLES_PER_CLOCK =1**

VF Code	Video Format	[4DW-1: 3DW]	[3DW-1: 2DW]	[2DW-1: DW]	[DW-1:0]
0	YUV 4:2:2			V/U, Cr/Cb	Y
1	YUV 4:4:4		V, Cr	U, Cb	Y
2	RGB		R	B	G
3	YUV 4:2:0			V/U, Cr/Cb	Y
4	YUVA 4:2:2		α	V/U, Cr/Cb	Y
5	YUVA 4:4:4	α	V, Cr	U, Cb	Y
6	RGBA	α	R	B	G
7	YUVA 4:2:0			α, V/U, Cr/Cb	Y
8	YUVD 4:2:2		D	V/U, Cr/Cb	Y
9	YUVD 4:4:4	D	V, Cr	U, Cb	Y
10	RGBD	D	R	B	G
11	YUV 4:2:0		D	V/U, Cr/Cb	Y
12	Mono/Sensor				Y, RGB, CMY
13	Custom2			2 Components – No DRC	
14	Custom3		3 Components – No DRC		
15	Custom4	4 Components – No DRC			

Note: For any of the 4:2:2 and 4:2:0 formats, Cb (or U) and Cr (or V) samples are split over two data beats but only in a one sample per clock mode. The first data beat holds Cb (or U); the second data beat holds Cr (or V). In other words, the first active pixel of the frame contains [Cb0:Y0] and the next pixel contains [Cr0:Y1]. The 4:2:0 format adds vertical subsampling to the 4:2:2 format, which is implemented in Video over AXI4-Stream by omitting the chroma data on every other line.

Encoding Multiple Pixels

When multiple samples/pixels are carried by AXI4-Stream, pixels should be packed from least significant bit (LSB) to MSB, e.g., the least significant pixel should correspond to the left-most pixel in a scanline, or to the pixel captured earliest in time. For example, if 4 samples/pixels are sent per data beat, the first sample sits in the least significant, the 4th sample sits in the most significant bit positions.

When multiple pixels or samples are transferred using the video protocol over AXI4-Stream, color components pertinent to the individual pixels are arranged according to Table 1-5, presenting examples for transferring two pixels for video modes 0, 1, 2, 3, 12. Pixel data is packed continuously without any padding between pixels. When N*DW is not an integer multiple of 8, video data is zero padded on the MSBs, as presented on Figure 1-5. If the line size is not divisible by the number pixels/samples per data beat, then the last beat of the

line should use the LSBs. Then, the unused pixel in the MSBs of the last data beat of the line should be padded with zeros.

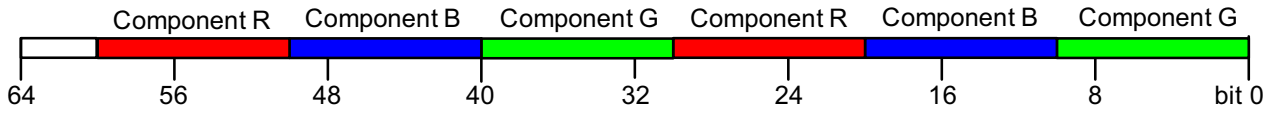

Figure 1-5: **Video Data Padding for TDATA**

Table 1-5: **Video Format Codes and Data Representation**

VF Code	Video Format	[6DW-1: 5DW]	[5DW-1: 4DW]	[4DW-1: 3DW]	[3DW-1: 2DW]	[2DW-1: DW]	[DW-1:0]
0	YUV 4:2:2			V0, Cr0	Y1	U0, Cb0	Y0
1	YUV 4:4:4	V1, Cr1	U1, Cb1	Y1	V0, Cr0	U0, Cb0	Y0
2	RGB	R1	B1	G1	R0	B0	G0
3	YUV 4:2:0			V0, Cr0	Y1	U0, Cb0	Y0
12	Bayer Sensor					RGB1, CMY1	RBGB0, CMY0

Dynamic TDATA Configuration

For applications where video IP can dynamically change color-component width, video format, or the number of pixels/samples per data beat, pixels and components should remain at the static locations determined by the generic parameters for instantiation. For example, if only one pixel is transmitted over an interface supporting at most two pixels per data beat, the sample/pixel should be aligned to the least significant pixel position. Similarly, if only 8 bits per component are transmitted over an interface generated for 10 bits per component, the active bits should be MSB aligned and LSB padded with zeros. Three examples are shown in Figure 1-6 through Figure 1-9.

IMPORTANT: *Although this specification supports dynamically changing the number of pixels/samples per data beat, this is not recommended because not all IPs support this feature.*

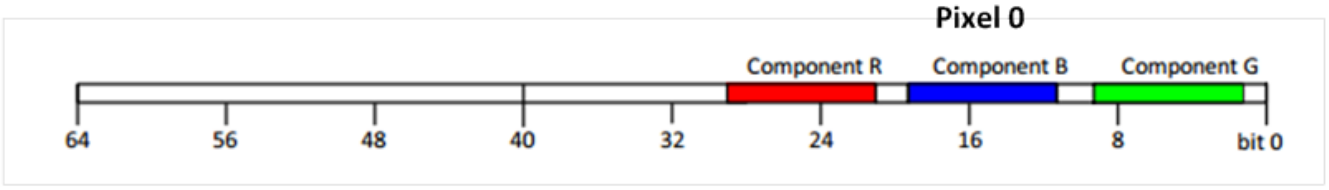

Figure 1-6: **One Pixel per Data Beat, Eight Bits per Component over a Two-Pixel per Data Beat, 10-Bits per Component Bus**

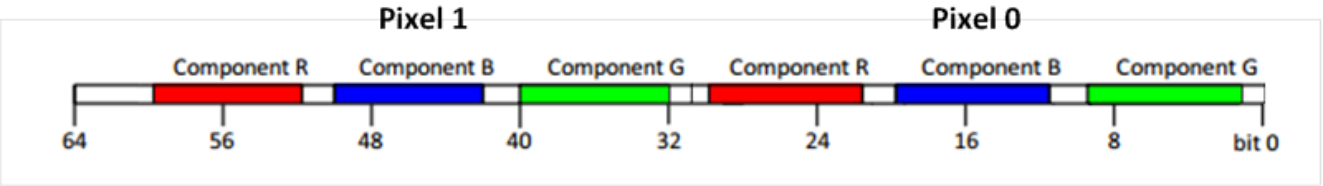

Figure 1-7: **Two Pixels per Data Beat, Eight Bits per Component over a Two-Pixel per Data Beat, 10-Bits per Component Bus**

Figure 1-8. captures RGB888 (pixel with three components, component width of 8).

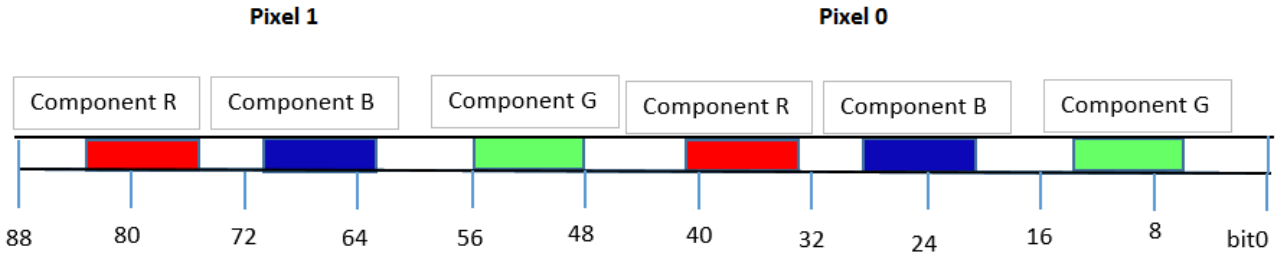

Figure 1-8: **Two Pixels per Data Beat, Eight bits per Component (RGB888) over a Two-Pixel per Data Beat, 14-bits per Component Bus**

Notes:

1. Each G,B,R component sits in 14-bit component space with MSB alignment.

Figure 1-9. captures RAW14 (pixel with single component, component width of 14).

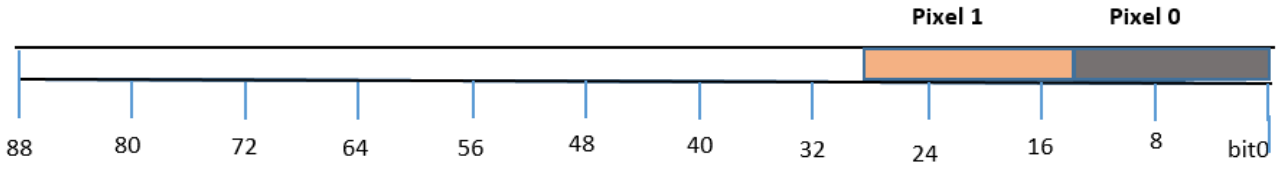

Figure 1-9: **Two Pixels per Data Beat, 14 Bits per Component (RAW14) over a Two-Pixel per Data Beat, 14-bits per Component Bus**

Notes:

1. Although RAW14 may only use the lower 28 bits, the full AXI4-Stream interface remains 88-bits because it must accommodate the possibility of switching to RGB at full 14-bits per color if requested when dealing with dynamic TDATA. Down stream logic must be aware of this and should provide the appropriate bus interface and then internally discard bits if it does not use them.

Comparing the two data type component widths in Figure 1-8 and Figure 1-9, the RAW14 data type has 14-bit component and RGB888 8-bit component. Therefore, the RGB888 components are placed with MSB aligned and LSB padded with zeros on 14-bit component bus. Additionally, the RAW14 pixels are packed tightly together.

System Design Guide

Video Timing Information

AXI4-Stream carries only video pixel data, SOF, and EOL signals between component interfaces. Blanking or sync signals are not carried by the signaling interface, and strict signal periodicity is not required.

In addition to extracting video pixel data from the input stream and sending it to subsequent modules using video over AXI4-Stream, the interface modules must measure timing information (including the number of pixels per scan-line, number of active rows per frame, and so on) when receiving video from a standard periodic video source such as DVI, HDMI, SDI, or an image sensor. Input interface modules make this information available to video processing and output interface modules, which then recreate periodic timing signals and embed output video pixel data that was processed by the video system to recreate a periodic output stream such as DVI (Figure 2-1).

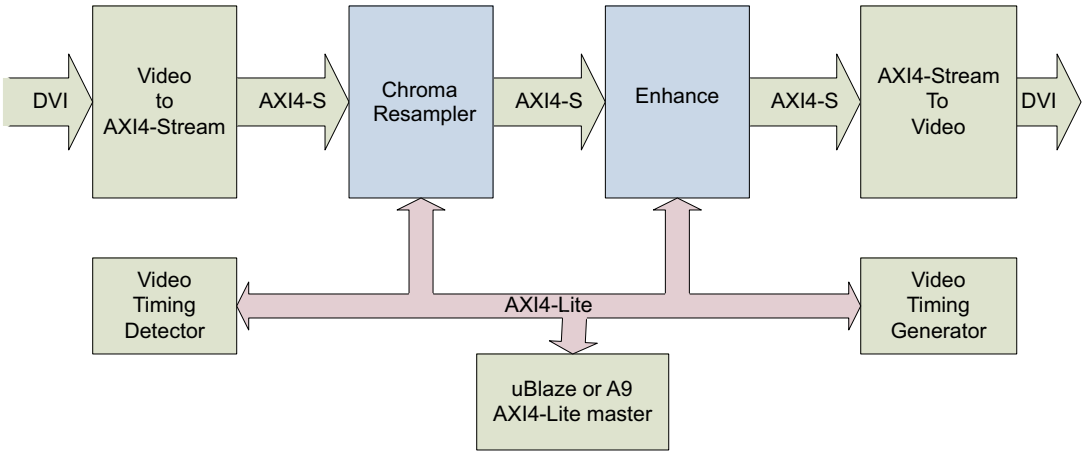

Figure 2-1: **Timing Information Extraction and Propagation Example**

Figure 2-1 illustrates the extraction and propagation of timing information. The Video In to AXI4-Stream input interface and Video Timing Detector cores measure timing information, and extract video pixel data. It then transmit the data using the AXI4-Stream (represented by the AXI4-S arrows in Figure 2-1). Timing information is propagated through optional AXI4-Lite interfaces. When present, the system processor (AXI4-Lite master) reads out measured timing information from the timing detector, and programs subsequent

processing cores and the timing generator using the AXI4-Lite control register interfaces. When instantiated without an AXI4-Lite control interface, video cores can only process a fixed video format / resolution, specified in the core GUI. In Figure 2-1, the Chroma Resampler and Enhance cores process the video stream. The processing cores might contain line buffers for which the number of active pixels per scan line is necessary. The processing cores receive active size (number of pixels per scan line, number of scan lines per frame) measurement values, among other timing parameters from the Video Timing Detector module, which is used with the DVI input interface IP. Processing cores also verify the data by employing pixel counters between subsequent EOL pulses. The AXI4-Stream to Video output interface core generates Standard Sync, Blank and Active Video timing signals, as defined by the timing information received, and embeds the video pixel data as received over the AXI4-Stream input interface.

Propagating Video Timing Information

Input and Output interface IP should provide two interface options to make measured timing information available for subsequent cores. For embedded systems either using a processor or dedicated IP acting as the AXI4-Lite master, an AXI4-Lite interface should be provided with a standardized register API to present timing information. For standalone video systems without an embedded processor, timing parameters for a fixed format/resolution should be provided through the IP parameters and/or GUI. The Video Timing Controller (VTC) core contains the Timing Detector and Timing Generator cores for use with custom AXI4-Stream interfaces.

The Video to AXI4-Stream and AXI4-Stream to Video cores are delivered as HDL source code and provided as examples to expedite custom interface development. For embedded systems using a processor acting as an AXI4-Lite master or dedicated IP acting as the AXI4-Lite master, an AXI4-Lite pCore interface should be provided with a standardized register API to present timing information. For more information, see AXI4-Lite Interface in Chapter 3.

Using the TUSER signal to transmit periodic sync information, such as hsync or vsync along with the video data is prohibited as there are no guarantees on IP delay consistency (aperiodicity), and delay matching between DATA and TUSER bits through IP cores. Furthermore, when video data is written and retrieved from frame buffers, sync information from TUSER is not recovered.

Transferring timing information or ancillary data embedded in the AXI4-Stream video stream is also prohibited, either in the form of a header or as a watermark. No method is provided for, or expected from processing cores to distinguish timing information or ancillary data packets from valid pixel data. When video data is re-formatted, for example video scaling changes the active frame dimensions, no mechanism is provided or expected to change timing or stream information embedded in video data.

Reset Requirements

Hardware Reset

Each AXI interface must be designed to accommodate entering or exiting a reset on a different (preceding or subsequent) cycle than the interface to which it is connected. Specifically, an IP core must not rely on another connected IP being reset simultaneously during the same cycle. Video IP should be designed so that any reset of the AXI4-Stream interfaces re-initializes the IP to reduce disruption on the output video stream.

Although Xilinx IP can generally have multiple AXI interfaces connected to isolated interconnection networks to support the localized reset of some interfaces, it is not recommended. As a practical system design guideline, the reset source(s) should be held active internally for some minimum number of cycles (of the slowest clock in the system) to ensure that all IP is properly reinitialized and all AXI interfaces go into the quiescent state prior to releasing the reset. If internal extension of the reset pulse is not throughble, video IP data sheets specify the required reset pulse-width, if greater than one cycle.

As stated in the Xilinx AXI Reference Guide guidelines, it is recommended that all AXI interfaces in a system be globally reset together. When resetting multiple video cores in a system, all interfaces must be reset before any interface comes out of reset. Video IP should accept and drop (not propogate) valid samples until the SOF signal is received.

AXI4-Stream interfaces must deassert their VALID and READY outputs while in reset. This does not need to commence immediately upon sampling the reset input active, but in time to allow the network of connected IP to reach a quiescent reset state prior to the deassertion of reset at any IP. This allows for arbitrary (but reasonable) internal pipe-lining of reset inputs, including resynchronization to a different clock domain, if necessary.

Software Reset

When resetting multiple video cores within a system, all interfaces must be reset before any interface comes out of reset. When reset is performed in the software (which asserts/deasserts software reset flags sequentially), the IP cores should be reset from the output towards the input. The software reset pin of video IP closest to the system output should be asserted first. Subsequent cores near the signal source should then be reset. Software reset pins should be deasserted in the same sequence.

If permitted by the application, provide a soft software reset option (SSR) for the video IP, where reset is synchronized with video frame boundaries. If sufficient time is available between video frames, (for example, a vertical blanking period is present), a soft reset after the predicted end-of-frame can facilitate the reset of individual cores without negatively impacting system performance.

Input/Output Interfaces - Automatic Delay Matching

The handshaking mechanism of AXI4-Stream provides a framework that allows building video systems that align data and timing signals without having to manually calculate propagation delay through processing blocks, as well as creating frame sync signals to trigger certain blocks. For data and output sync signal alignment, consider the following design constraints:

- Is it possible to hold up the input video stream? Is there a back pressure signal?
- Must the output stream be phase-locked to an external Frame Sync signal?
- Are the input and output video clocks the same or phase-locked to each other?

Based on the above consideration, typical use cases include:

- Timed video input, such as DVI, that cannot be delayed. Timed video output using the same video clock. For automatic delay matching, synchronization is necessary.
- Input and output are in unrelated clock domains (scaled video), and a frame buffer is necessary.

No delay matching is necessary in a hardware accelerator scenario where input is coming from memory or from a processor. Processing and output blocks can generate output when the input is available. If input and output are in unrelated clock domains, a frame buffer is necessary. The following sections contain recommendations for implementing protocol-based delay matching for scenarios with or without frame buffers.

In all cases, the input interface module is expected to have a "locked" output, originating from the VTC timing detector. The VTC timing detector issues a signal when the input timing measurements are stabilized. The input interface module is expected to drop pixel data until input timing has locked.

Periodic Input Stream, Unconstrained Output Stream, No Frame Buffer

This section provides an algorithm (Figure 2-2) describing how automatic data-sync signal alignment can be achieved at the output interface for a video system that contains

- no frame buffer
- a periodic input stream that cannot be held off
- an output video pixel clock that is either the same, or a derivative of the input pixel clock, and

- the output video stream does not have to be phase locked to an external Frame Sync signal.

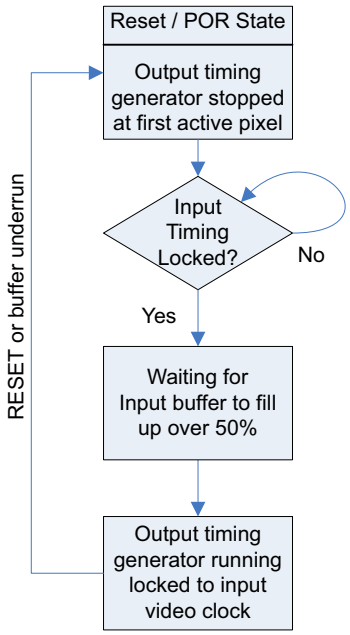

Figure 2-2: **Output Timing Generator Control Flowchart for Unconstrained Output Video Stream**

This scenario applies to a sensor image pre-processing pipeline, where input and output pixel rates are identical, and the output timing generator does not have to be locked to an external frame sync source. After power on or reset, the output AXI4-Stream interface deasserts READY, and the output timing signal generator state machine is initialized to wait in the state just before the start of active video.

Note: In this case, the function of READY is limited to what the internal buffers allow if the input stream cannot be held back.

The output timing generator waits for the input interface to signal that timing information has stabilized (locked). Now, the output AXI4-Stream interface should assert READY, which propagates backward towards the input of the pipeline. As a result, pixel data is propagated down the pipeline. Processed pixel data reaches the output interface module when its VALID input is sampled high. When the input data buffer of the output video interface gets 50% full, the output timing generator can start generating periodic output sync/blank signals, and pixel data can be fed forward to the output.

Output Stream Generation for Pixel Data from Frame Buffer

This section provides an algorithm for automatic data–sync signal alignment at the output interface for a video system that contains a frame buffer, and the output video stream might be in a separate clock domain or might have to be phase-locked to an external Frame Sync signal (Figure 2-3).

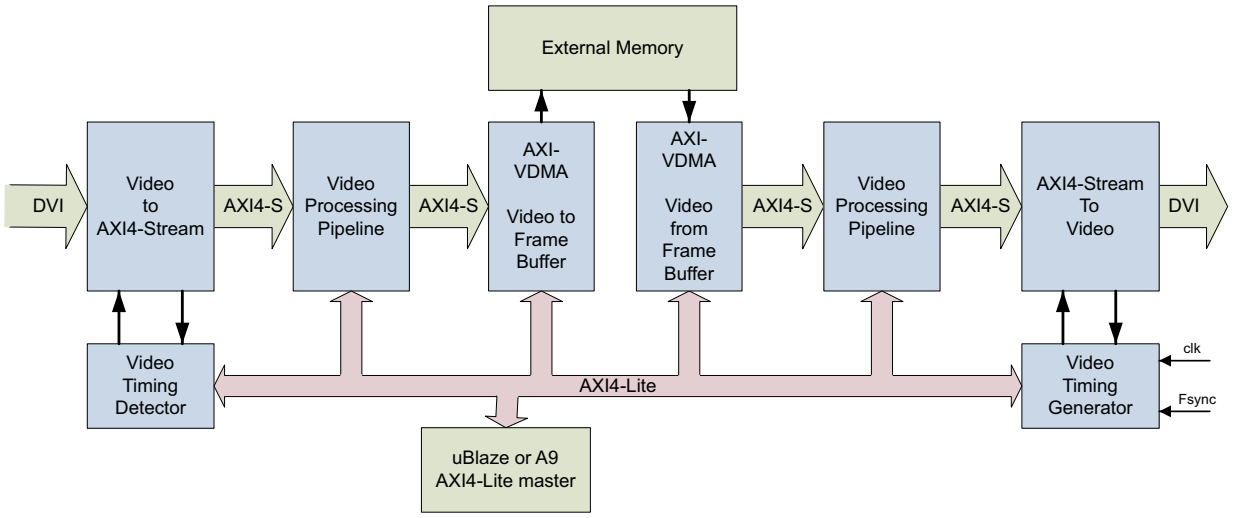

Figure 2-3: **Example System with Output Sync Tied to an External Frame Sync Signal**

The portion of this system relevant to output stream synchronization is the leg from the frame buffer to the output interface core, which can contain processing cores. These processing cores can change the effective pixel rate. The example presented in Figure 2-4 uses the video scaler core, which typically changes the pixel rate, and can operate in three different clock domains:

- its input interface running at the memory system clock rate
- the core processing data at a processing clock rate
- its output interface running at the same clock as the output interface, which can come from an external clock source

The choice of external frame buffer for AXI4-Stream based IP video systems is the AXI-VDMA core, which must be configured to the desired frame size using an AXI4-Lite interface. Figure 2-4 illustrates timing information (from an input interface core, or from software) distributed using this interface.

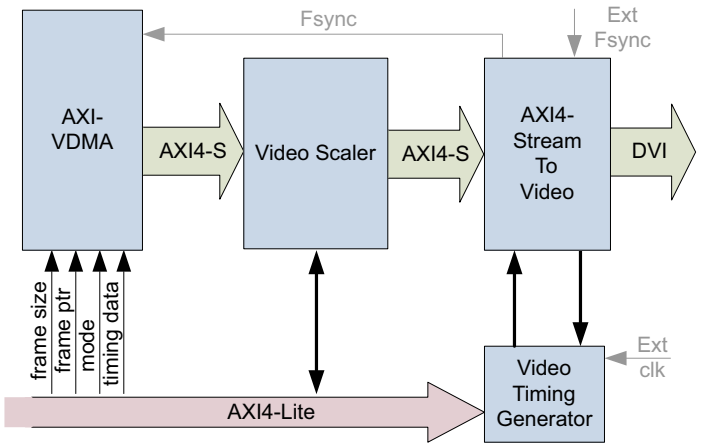

Figure 2-4: **Example System with Video Scaler**

Three possible scenarios are addressed in this setup:

1. External output clock (`Ext clk`) is different from the input clock, but there is no external `Fsync` signal.
2. Output Timing Generator needs to be locked to an external `Fsync`.
3. External `Fsync` driving the AXI-VDMA readouts.

For scenario 1, the data – sync signal alignment algorithm is as follows:

After power up or reset, the output interface core should deassert READY and set all outputs to defaults until timing information is locked (Figure 2-7). The AXI-VDMA should be configured with the write side being `Fsync` and Genlock master. When the input buffer of the Video output core is 50% filled with data from the AXI-VDMA, the output timing signal generation should commence. When the timing generator gets to the phase where active video needs to be sent, but pixels are not present yet, blank frames should be generated. If the output interface data buffer gets full, the output interface core should deassert TREADY.

For scenario two, the setup and protocol are identical, but the video timing generator should be configured to sync with the external Fsync.

For scenario 3, a frame sync signal originating from the output timing generator or an external fsync is used to trigger AXI-VDMA frame reads. If an external frame sync signal is present, ensure that the phase relationship between the external `Fsync` pulse and the VTC generator `Fsync` allows pixel data to be fetched from the AXI-VDMA and propagated through subsequent cores between the AXI-VDMA and the output interface module. This allows data and timing signals on the output interface to be synchronized.

A good example of this is when the external frame sync is in phase with the start of vertical blanking. If output pixels are needed immediately, this sync is too late to trigger readout from the AXI-VDMA.

The timing generator core contains logic which can generate frame sync pulses at arbitrary phases after the generator is generating periodic timing signals. For scenarios when the external frame sync is too late to trigger data readout, an earlier, regenerated frame sync pulse should be used. This ensures that pixel data gets to the output interface core before it needs to be sent in phase with the periodic output timing signals.

For video systems with a Frame Buffer but no external output frame sync source, the AXI-VDMA core can automatically fetch the last frame finished on the write-side to be picked up immediately when the read size is in idle (reading a frame has completed).

When pixel data propagates to the output interface core, the output interface core should deassert its READY output and start driving pixel data using READY to maintain synchrony between the input pixel flow and output sync signals.

When Sync is Lost

When output interface cores are used in conjunction with a frame buffer (see Periodic Input Stream, Unconstrained Output Stream, No Frame Buffer), output timing signal generation should start immediately after timing has locked, regardless of whether an external frame sync pulse is present.

When an out-of-sync external frame sync pulse is received, output timing generation should re-initialize. A new fsync pulse should be generated for the AXI-VDMA, and input pixels from the existing frame should be dropped until the arrival of the SOF pulse. If necessary, a blank frame should be sent on the output until sync is reestablished.

If the external frame sync pulse is not present when expected, output timing generation should continue freewheeling.

Input Interface cores should not start sending incomplete frames. If the timed video source is disconnected or reconnected, or when the system recovers from reset or power-up, the input AXI4-Stream interface core should wait until the start of the first frame after timing is locked before sending data over the AXI4-Stream master interface.

When Timing Information Is Incorrect

This situation can arise if any of the AXI-VDMA frame dimensions, the scaler frame dimensions or ratios, or the output interface timing parameters are programmed incorrectly (Figure 2-4).

There could be a discrepancy between measured frame dimensions based on EOL and SOF locations and the frame dimensions provided to the VTC generator side and the processing cores through the core GUI or the AXI4-Lite register interface.

If the SOF and EOL framing signals occur early, processing cores should immediately start processing the new line or new frame. If the framing signals are late, processing cores

should purge partial frames by dropping pixels until the expected SOF or EOL signal is received.

Streaming Video Input Connection

As illustrated in Figure 2-3, the Video In to AXI4-Stream core (VID-IN) is provided to interface periodic video, such as HDMI or DVI to AXI4-Stream, and is intended for use with the Video Timing Controller (VTC). Together, the VTC processes timing signals and the VID-IN core buffers input video data (as necessary) before transmission over AXI4-Stream. The VTC core can process one of the following sets of timing signals:

- Vsync, Hsync, and DE
- Vblank, Hblank, and DE
- Vsync, Hsync, Vbank, Hblank, and DE

The choice of timing signal sets should be specified when generating the VTC core.

Figure 2-5 shows a typical example of connecting the VID-IN and VTC cores to downstream video processing cores ("Video IP Sink") through AXI4-Stream interfaces.

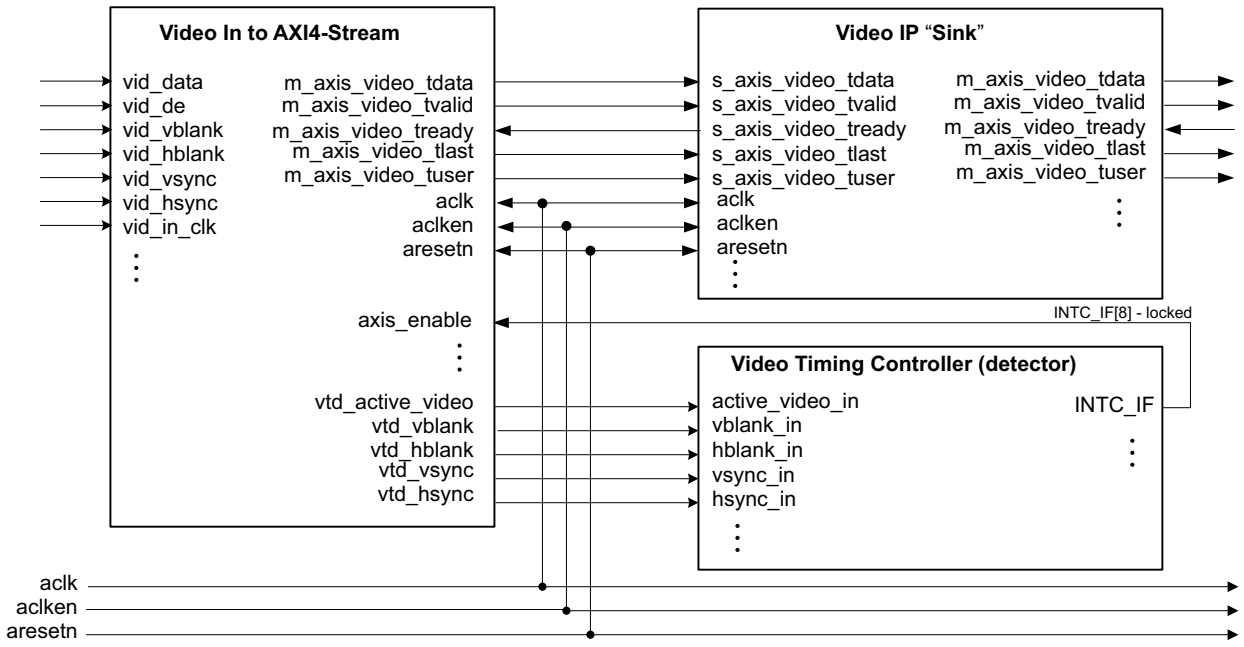

Figure 2-5: **Connecting the Video to AXI4-Stream Core to the Video Timing Controller**

At startup, the following points should be considered:

- The VID-IN core should not start sending data to downstream core(s) until they are enabled and initialized.
- The VID-IN core should not start sending data to downstream cores until the VTC cores is enabled, initialized, and locked.

After the start of streaming video, bootup, or resetting the system, the VTC core can take more than a full frame of data to accurately measure all timing parameters. During this time the locked status bit of the VTC, available through bit 8 of the optional INTC_IF interface, is 0. It is recommended to connect INTC_IF[8] to the `axis_enable` input of VID-IN core. This hardware configuration ensures that no video is sent before the VTC is locked.

Xilinx® recommends that the VTC detector be enabled only after the rest of the downstream processing cores are all initialized and enabled. Otherwise, the output FIFO within the VID-In core can become full while downstream cores initialize in the pipe, ultimately resulting to lost pixels, lines, and/or frames of video.

If the downstream IP core need to know the input resolution before it can be configured, the you should:

1. SW Reset and SW disable all processing cores and the VTC
2. Enable the VTC to detect input resolution.
3. Once the VTC is locked, read measured resolution.
4. Reset the VTC
5. Configure the downstream IP.
6. Enable the downstream IP.
7. Enable the VTC

External Frame Buffers

The choice of an external-frame-buffer solution for AXI4-Stream based video systems is the AXI-VDMA core. The AXI-VDMA core supports the AXI4-Stream video interfaces natively, meaning S0F and E0L signals are properly interpreted and generated by the AXI-VDMA core.

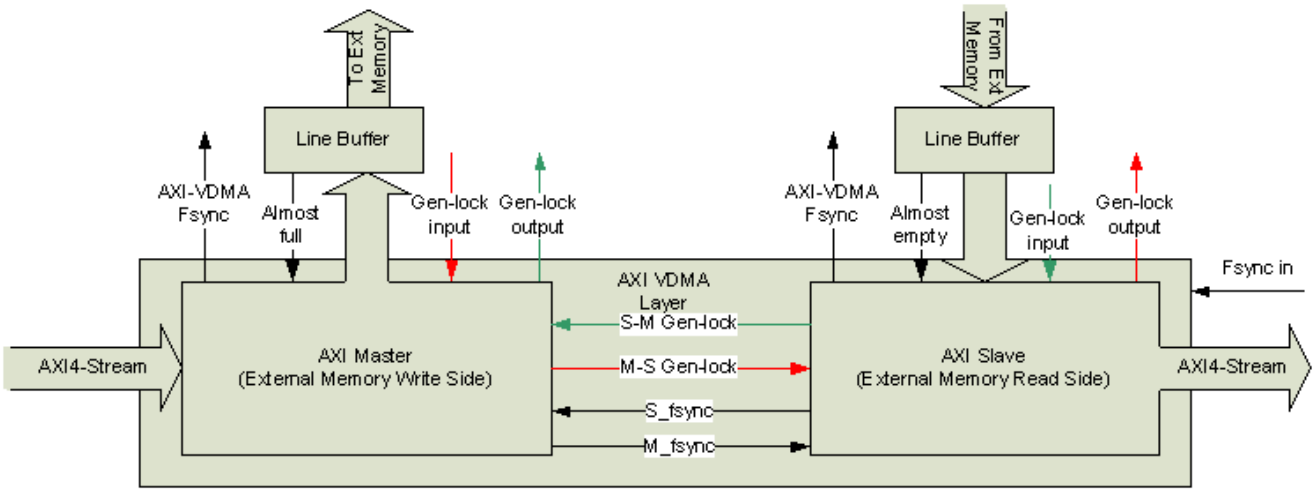

Figure 2-6: **AXI-VDMA Layer**

As illustrated in Figure 2-6, the AXI-VDMA core supports one master and one slave interface. Slave/Master interfaces can:

- Use any input SOF signals, or an external Frame Sync input as a source to initiate Frame transfers (AXI-VDMA Frame sync crossbar).
- AXI Master interfaces to use any AXI Slave interfaces to be the Gen-lock master.
- AXI Slave interfaces to use any AXI Master interfaces to be the Gen-lock master (Genlock crossbar).

Using a Frame sync crossbar enables video systems with a Frame Buffer, but without external output Frame sync source, to automatically retrieve the last frame finished on the write-side. This is picked up immediately after reading a frame has completed on the read side.

Some IP cores, such as the Video On-Screen Display, can have multiple read channels (slave interfaces) which must be synchronized. You might need multiple instances of a slower core running in parallel to achieve sufficient throughput. These parallel core instances can use multiple write channels (master interfaces), which must be synchronized. Operating modes for single write - multiple read ports:

- Genlock mode: Write side and read side individually freewheels.
- Same Frame Readout mode: Write side freewheels, but all read sides need to read out the same frame.
- Synchronizer mode: All frames written need to be read out on all ports.

Incorrect Timing Information

There can be some discrepancy between the measured frame dimensions based on EOL and SOF locations and the frame dimensions the AXI-VDMA programmed through the AXI4-Lite interface. This is often due to programming or communication errors.

When the number of pixels between subsequent EOL pulses is less than the line-length programmed into the AXI-VDMA core, the core triggers an interrupt indicating the error. The AXI-VDMA line pointer moves forward to the next line. Data received after received EOL is written to the start of a new line. No padding data is written to the frame buffer to complete the line as programmed to the core.

When the number of pixels between subsequent EOL pulses is more than the line-length programmed into the AXI-VDMA, the core triggers an interrupt indicating the error and drops extraneous pixels until EOL is received.

When the number of lines between subsequent SOF pulses is less than the line-length programmed into the AXI-VDMA, the core triggers an interrupt indicating the error and the frame pointer moves forward to the next line. Data received after received SOF is written to the next frame in the buffer. No padding data is written to the buffer to complete the frame as programmed to the AXI-VDMA core.

When the number of lines between subsequent SOF pulses is more than the line-length programmed into the AXI-VDMA, the core triggers an interrupt indicating the error and drops extraneous lines until SOF is received.

Multipoint Interfaces

Some applications require a single AXI4-Stream master interface connected to multiple slaves, such as a stream splitter, or multiple master interfaces to be connected to a single slave, such as a stream combiner.

For video applications, the use of stream combiners is discouraged. Without the TID and TDEST fields, pixel sources are ambiguous. The recommended solution is to create separate slave component interfaces on the receiver IP to the IP to distinguish data received from different sources, if necessary. No explicit video IP is provided to split AXI4-Streams. HDL and EDK users can easily implement the video splitter with AND gates.

Example: 1-to-2 splitter implemented in VHDL

```
source_READY   <= target1_READY and target2_READY;
target1_VALID  <= source_VALID  and target2_READY;
target2_VALID  <= source_VALID  and target1_READY;
```

The example above assumes downstream target interfaces asserting READY as soon as the target is ready to receive data, independent from VALID. Otherwise, a small, distributed

memory based FIFO must be inserted between the splitter and the target to avoid deadlocks.

Ancillary Data

Ancillary data (which includes audio, teletext, captions, or metadata) is digital data embedded in a video stream. Because video over an AXI4-Stream interface is not packetized to carry video and non-video data, ancillary data must be de-embedded or discarded by the input interface and transmitted from front-to-end using a separate (AXI or non-AXI) auxiliary channel, as seen in Figure 2-7.

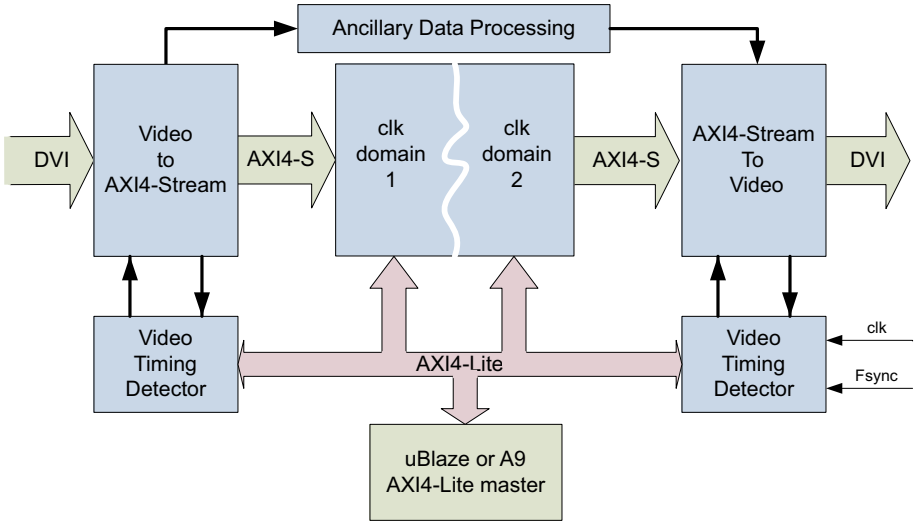

Figure 2-7: **Ancillary Data Management**

When video frame rates change, buffering, re-sampling, and other processing may be required on ancillary data. This must be done separately from the Video over AXI4-Stream interface by deembedding the ancillary data before the frame rate change, processing it, and reembedding it into the video stream after the frame rate change.

Video Subsystem Software Guidelines

Each video subsystem comprises one or more video pipelines. A video pipeline is any chain of video IP cores that starts from a Video-In or AXI VDMA (MM2S Channel) core and terminates on a Video-Out or AXI VDMA (S2MM channel) core.

Each pipeline must be reset, configured, reconfigured, enabled, or disabled starting from the output (back-end) moving toward the input (front-end). The following is a list of typical video pipeline operations that must be performed from back-end to front-end:

- Video pipeline reset: Resetting all cores within a pipeline
- Video pipeline configuration: Configuring all cores after reset. Do not Enable the cores during this step
- Video pipeline dynamic reconfiguration: Configuring all cores without resetting, such as a frame size change
- Video pipeline enable: Enabling all cores within a pipeline
- Video pipeline disable: Disabling all cores within a pipeline

In general to initialize a video pipe, the following operations should be performed in this order:

1. Initialize all video IP drivers.
2. Reset all cores starting from the back-end first, moving forward in the pipe.
3. Configure without enabling all cores starting from the back-end first, moving forward in the pipe.
4. Enable all cores starting from the back-end first, moving forward in the pipe.

Note: Step one only needs to be done once after boot time. Drivers do not need to be reinitialized if the video pipeline needs to be reconfigured.

If a video subsystem contains more than one video pipeline, then each pipeline can be operated upon individually. However, in most applications the input (front-end) pipelines should be operated upon first, before back-end pipelines to avoid invalid data to be processed and/or displayed.

Note: Pipelines are operated upon from front-end to back-end. Cores within a pipeline are operated upon from back-end to front-end.

Video Pipeline Example

Refer to the video subsystem depicted in Figure 2-8 in the following example operations and C code snippets. This video subsystem contains three video pipelines. The three pipelines consist of the following cores:

- Pipeline 1:
  - Video to AXI4-Stream
  - Video IP 1
  - AXI VDMA 1 (S2MM Channel)
- Pipeline 2:
  - AXI VDMA 1 (MM2S Channel)
  - Video Processing Subsystem

- AXI VDMA 2 (S2MM Channel)

- Pipeline 3:
  - AXI VDMA 2 (MM2S Channel)
  - Video IP 2
  - AXI4-Stream to Video

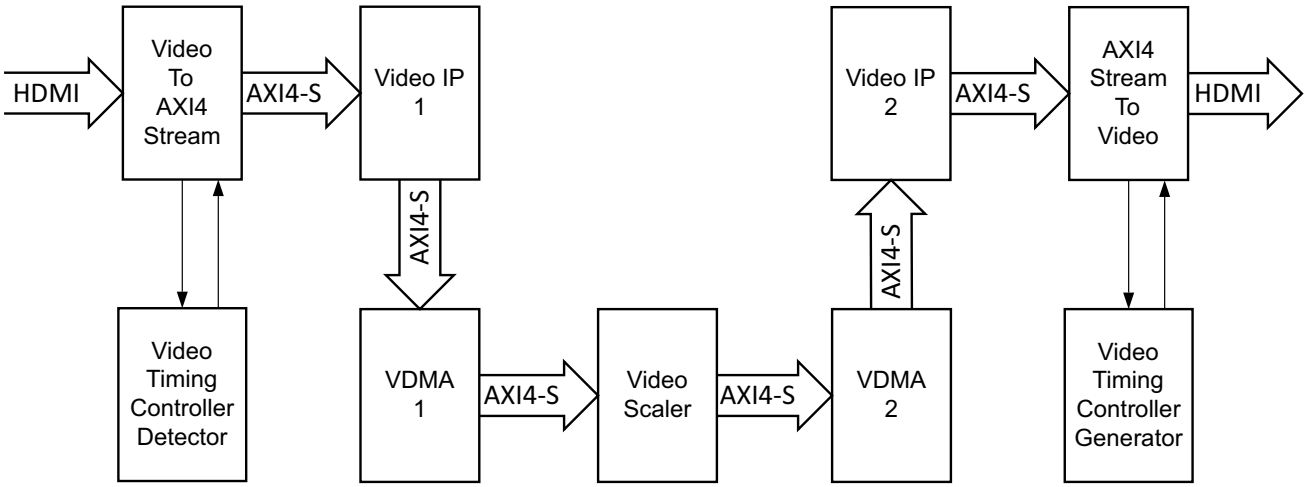

Figure 2-8: **Example Video Subsystem with Three Video Pipelines**

To bring up this system in software, the following operations should be performed in the following order:

1. Initialize core drivers (Perform One time only) using the `<core>_CfgInitialize()` functions.
2. Bring up Pipeline 1 (Input Video Pipeline)
   a. SW Reset AXI VDMA 1 (S2MM Channel)
   b. SW Reset Video IP 1
   c. SW Reset VTC detector
   d. Configure AXI VDMA 1 (S2MM Channel)
   e. Configure Video IP 1
   f. Configure VTC detector
   g. Enable AXI VDMA 1 (S2MM Channel)
   h. Enable Video IP 1
   i. Enable VTC detector
3. Bring up Pipeline 2 (Scaler Pipeline)
   a. SW Reset AXI VDMA 2 (S2MM Channel)

b. SW Reset Scaler
c. SW Reset AXI VDMA 1 (MM2S Channel)
d. Configure AXI VDMA 2 (MM2S Channel)
e. Configure Scaler
f. Configure AXI VDMA 1 (MM2S Channel)
g. Enable AXI VDMA 2 (MM2S Channel)
h. Enable Scaler
i. Enable AXI VDMA 1 (MM2S Channel)

4. Bringup Pipeline 3 (Output Video Pipeline)
   a. SW Reset VTC generator
   b. SW Reset Video IP 2
   c. SW Reset AXI VDMA 2 (MM2S Channel)
   d. Configure VTC generator
   e. Configure Video IP 2
   f. Configure AXI VDMA 2 (MM2S Channel)
   g. Enable VTC generator
   h. Enable Video IP 2
   i. Enable AXI VDMA 2 (MM2S Channel)

To reconfigure this system, perform the above operations except step 1 (Initialize core drivers).

Note: VDMA S2MM and MM2S channels should be reset, configured, reconfigured and enabled separately. Each VDMA channel should be treated as individual cores belonging to separate video pipelines. Avoid operating on both channels at the same time. The channel operations should by synchronized to the pipeline in which the channel belongs.

The following C code snippet shows the code needed to bring up the VDMA 1, Scaler, VDMA 2 pipeline:

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xscaler.h"
#include "xaxivdma.h"

////////////////////////////////////////////////////////////////
// Global Defines
////////////////////////////////////////////////////////////////
#define VIDIN_FBADDR     0x31800000
#define SCALEROUT_FBADDR 0x33000000
```

```
#define FRAME_STORE_WIDTH      2048
#define FRAME_STORE_HEIGHT     2048
#define FRAME_STORE_DATA_BYTES 2

#define VDMA_CIRC        1
#define VDMA_NOCIRC      0
#define VDMA_EXT_GENLOCK 0
#define VDMA_INT_GENLOCK 2
#define VDMA_S2MM_FSYNC  8
#define COEFF_SET_INDEX  0

////////////////////////////////////////////////////////////////////
// Function Prototypes
////////////////////////////////////////////////////////////////////
void vdma_init(XAxiVdma *VDMAPtr, int device_id);
int vdma_reset(XAxiVdma *VDMAPtr, int direction);
int vdma_setup(XAxiVdma *VDMAPtr,
  int direction,
  int width,
  int height,
  int frame_stores,
  int start_address,
  int mode
);
void scaler_init(XScaler *ScalerPtr, int device_id);
int scaler_setup(XScaler *ScalerInstPtr,
  int ScalerInWidth,
  int ScalerInHeight,
  int ScalerOutWidth,
  int ScalerOutHeight);

////////////////////////////////////////////////////////////////////
// Global Core Driver Structures
////////////////////////////////////////////////////////////////////
XAxiVdma VDMA1;
XAxiVdma VDMA2;
XScaler  Scaler;

XScalerAperture Aperture;/* Aperture setting */
XScalerStartFraction StartFraction;/* Luma/Chroma Start Fraction setting*/
XScalerCoeffBank CoeffBank;/* Coefficient bank */



////////////////////////////////////////////////////////////////////
// Function: configure_scaler_pipeline()
// Configure Scaler Pipeline (Pipeline 2)
////////////////////////////////////////////////////////////////////
int configure_scaler_pipeline(
  int input_x,
  int input_y,
  int output_x,
  int output_y)
{
  int Status;
  ////////////////////////////////////////////////////////
  // Initialize Drivers – Order not important
  // Do after clocks are setup
```

```
    ////////////////////////////////////////////////////////
    vdma_init  (&VDMA1,  0);
    vdma_init  (&VDMA2,  1);
    scaler_init(&Scaler, 0);

    ////////////////////////////////////////////////////////////////////////
    // Pipeline 2: Reset Cores
    ////////////////////////////////////////////////////////////////////////
    vdma_reset  (&VDMA2, XAXIVDMA_WRITE);
    scaler_reset(&Scaler);
    vdma_reset  (&VDMA1, XAXIVDMA_READ);

    ////////////////////////////////////////////////////////////////////////
    // Pipeline 2: Configure Cores
    ////////////////////////////////////////////////////////////////////////
    printf("Setting up VDMA Writer...\n");
    vdma_setup(&VDMA2,
        XAXIVDMA_WRITE,
        output_x,
        output_y,
        3,
        SCALEROUT_FBADDR,
        VDMA_NOCIRC|VDMA_INT_GENLOCK);

    printf("Setting up Scaler...\n");
    scaler_setup(&Scaler, input_x, input_y, output_x, output_y);

    printf("Setting up VDMA Reader...\n");
    vdma_setup(&VDMA1,
               XAXIVDMA_READ,
               input_x,
               input_y,
               3,
               VIDIN_FBADDR,
               VDMA_NOCIRC|VDMA_INT_GENLOCK|VDMA_S2MM_FSYNC);

    ////////////////////////////////////////////////////////////////////////
    // Pipeline 2: Enable cores
    ////////////////////////////////////////////////////////////////////////

    //Enable write VDMA, VDMA2 (S2MM Channel)
    Status = XAxiVdma_DmaStart(&VDMA2, XAXIVDMA_WRITE);
    if (Status != XST_SUCCESS)
    {
      printf("ERROR: VDMA2 Start write transfer failed %d\r\n", Status);
      return XST_FAILURE;
    }

    XScaler_Enable(&Scaler);

    Status = XAxiVdma_DmaStart(&VDMA1, XAXIVDMA_READ);
    if (Status != XST_SUCCESS)
    {
      printf("ERROR: VDMA1 Start read transfer failed %d\r\n", Status);
      return XST_FAILURE;
    }

    return 1;
}
```

```
////////////////////////////////////////////////////////////////////
// Function: vdma_init()
// Initialize VDMA Driver
////////////////////////////////////////////////////////////////////
void vdma_init(XAxiVdma *VDMAPtr, int device_id)
{
  int Status;
  XAxiVdma_Config *VDMACfgPtr;

  VDMACfgPtr = XAxiVdma_LookupConfig(device_id);
  if (!VDMACfgPtr)
  {

    printf("ERROR: No VDMA found for ID %d\r\n", device_id);
  }

  Status = XAxiVdma_CfgInitialize(VDMAPtr,
    VDMACfgPtr,
    VDMACfgPtr->BaseAddress
  );
  if (Status != XST_SUCCESS) {
    printf( "ERROR: VDMA Configuration Initialization failed %d\r\n",
      Status);
  }

}
////////////////////////////////////////////////////////////////////
// VDMA Channel Reset
////////////////////////////////////////////////////////////////////
int vdma_reset(XAxiVdma *VDMAPtr, int direction)
{

  int Polls;

  printf("Resetting VDMA ...\n");
  XAxiVdma_Reset(VDMAPtr, direction);
    Polls = 100000;

  while (Polls && XAxiVdma_ResetNotDone(VDMAPtr, direction)) {
    Polls -= 1;
  }

  if (!Polls) {
    printf( "ERROR: VDMA %s channel reset failed %x\n\r",
     (direction==XAXIVDMA_READ)?"Read":"Write", 0);

    return XST_FAILURE;
  }

  return 1;
}


////////////////////////////////////////////////////////////////////
//  VDMA Channel Configure/Setup
////////////////////////////////////////////////////////////////////
```

```
int vdma_setup(XAxiVdma *VDMAPtr, int direction, int width, int height, int
frame_stores, int start_address, int mode)
{
  int Status, i, Addr;

  XAxiVdma_DmaSetup DmaSetup;

  //printf("Setting up VDMA Read Config...\n");
  DmaSetup.VertSizeInput = height;
  DmaSetup.HoriSizeInput = width * FRAME_STORE_DATA_BYTES ;

  DmaSetup.Stride = FRAME_STORE_WIDTH * FRAME_STORE_DATA_BYTES ;
    DmaSetup.FrameDelay = 0;

  DmaSetup.EnableCircularBuf = mode&1;
  DmaSetup.EnableSync = mode&1;

  DmaSetup.PointNum = (mode>>2) & 1;
  DmaSetup.EnableFrameCounter = 0; /* Endless transfers */

  DmaSetup.FixedFrameStoreAddr = 0; /* We are not doing parking */


  //Only set the number of frames if the VDMA can support more that we need
  //NOTE: the VDMA debug features for write to the frame store
  //      num reg must be enabled.
  if(VDMAPtr->MaxNumFrames > frame_stores)
  {
    Status = XAxiVdma_SetFrmStore(VDMAPtr, frame_stores, direction);
    if (Status != XST_SUCCESS) {

      printf("WARNING %d: VDMA - Setting Frame Store Number to %d Failed for %s
Channel. Exiting config.\r\n",
          Status, frame_stores,
          (direction==XAXIVDMA_READ)?"Read":"Write");

      return XST_FAILURE;
    }
  }



  Status = XAxiVdma_DmaConfig(VDMAPtr, direction, &DmaSetup);
  if (Status != XST_SUCCESS) {
    printf("ERROR: VDMA - %s channel config failed. (%d)\r\n",
     (direction==XAXIVDMA_READ)?"Read":"Write", Status);

    return XST_FAILURE;
  }

  /* Initialize buffer addresses
   *
   * These addresses are physical addresses
   */
  Addr = start_address;
  for(i=0; i < frame_stores; i++) {
    printf("  vdma_setup: Address %d = 0x%08x.\n\r", i, Addr);
    DmaSetup.FrameStoreStartAddr[i] = Addr;
```

```
    Addr += FRAME_STORE_WIDTH * FRAME_STORE_HEIGHT * FRAME_STORE_DATA_BYTES;
  }

  /* Set the buffer addresses for transfer in the DMA engine
   * The buffer addresses are physical addresses
   */
  Status = XAxiVdma_DmaSetBufferAddr(VDMAPtr, direction,
      DmaSetup.FrameStoreStartAddr);
  if (Status != XST_SUCCESS) {
    printf("ERROR: VDMA - %s channel set buffer address failed %d\r\n",
      (direction==XAXIVDMA_READ)?"Read":"Write",Status);

    return XST_FAILURE;
  }

  if(direction==XAXIVDMA_WRITE)
  {
    // use the TUSER bit for the frame sync for the write (S2MM side)
    XAxiVdma_FsyncSrcSelect(VDMAPtr,
      XAXIVDMA_S2MM_TUSER_FSYNC,
      XAXIVDMA_WRITE);
  }
  else
  {
    if(mode&0x08)
    {
      // VDMA Read (MM2S side) for the scaler input must be synced
      // to the S2MM frame Sync
      XAxiVdma_FsyncSrcSelect(VDMAPtr,
        XAXIVDMA_CHAN_OTHER_FSYNC,
        XAXIVDMA_READ); // DMA_CR[6:5] = 0b01

    }
    else
    {
      // VDMA 2 Read (MM2S side) must be not by synced and in free run
      // Its timing is governed by the output VTC generator
      // and AXI4-Stream to Video Out
      XAxiVdma_FsyncSrcSelect(VDMAPtr, XAXIVDMA_CHAN_FSYNC, XAXIVDMA_READ);
      // DMA_CR[6:5] = 0b00
    }
  }

  Status = XAxiVdma_GenLockSourceSelect(VDMAPtr, (mode>>1)&1, direction);
  if (Status != XST_SUCCESS) {
    printf("ERROR: VDMA - %s channel set gen-lock %s src failed %d\r\n",
        (direction==XAXIVDMA_READ)?"Read":"Write",
        (((mode>>1)&1)==XAXIVDMA_INTERNAL_GENLOCK)?"Internal":"External",

        Status);

    return XST_FAILURE;
  }

  return 1;

}
////////////////////////////////////////////////////////////////
// Initialize Scaler Driver
```

```
////////////////////////////////////////////////////////////////////
void scaler_init(XScaler *ScalerPtr, int device_id)
{
  int Status;
  XScaler_Config *ScalerCfgPtr;

  ScalerCfgPtr = XScaler_LookupConfig(device_id);
  if (!ScalerCfgPtr)
  {
    printf("ERROR: No Scaler found for ID %d\r\n", device_id);
  }
  Status = XScaler_CfgInitialize(ScalerPtr,
     ScalerCfgPtr,
ScalerCfgPtr->BaseAddress);
  if (Status != XST_SUCCESS) {
    printf( "ERROR: Scaler Configuration Initialization failed %d\r\n",
     Status);
  }

}
////////////////////////////////////////////////////////////////////
// Scaler Configure/Setup
////////////////////////////////////////////////////////////////////
int scaler_setup(XScaler *ScalerInstPtr,
     int ScalerInWidth, int ScalerInHeight,
     int ScalerOutWidth, int ScalerOutHeight)

{
  u8 ChromaFormat;
  u8 ChromaLumaShareCoeffBank;
  u8 HoriVertShareCoeffBank;

  /*
   * Disable the scaler before setup and tell the device not to pick up
   * the register updates until all are done
   */
  XScaler_DisableRegUpdate(ScalerInstPtr);
  XScaler_Disable(ScalerInstPtr);

  /*
   * Load a set of Coefficient values
   */

  /* Fetch Chroma Format and Coefficient sharing info */
  XScaler_GetCoeffBankSharingInfo(ScalerInstPtr,
      &ChromaFormat,
      &ChromaLumaShareCoeffBank,
      &HoriVertShareCoeffBank);

  CoeffBank.SetIndex = COEFF_SET_INDEX;
  CoeffBank.PhaseNum = ScalerInstPtr->Config.MaxPhaseNum;
  CoeffBank.TapNum = ScalerInstPtr->Config.VertTapNum;

  /* Locate coefficients for Horizontal scaling */
  CoeffBank.CoeffValueBuf = (s16 *)
    XScaler_CoefValueLookup(ScalerInWidth,
        ScalerOutWidth,
        CoeffBank.TapNum,
        CoeffBank.PhaseNum);
```

```
/* Load coefficient bank for Horizontal Luma */
XScaler_LoadCoeffBank(ScalerInstPtr, &CoeffBank);

/* Horizontal Chroma bank is loaded only if chroma/luma sharing flag
 * is not set */
if (!ChromaLumaShareCoeffBank)
  XScaler_LoadCoeffBank(ScalerInstPtr, &CoeffBank);

/* Vertical coeff banks are loaded only if horizontal/vertical sharing
 * flag is not set
 */
if (!HoriVertShareCoeffBank) {

  /* Locate coefficients for Vertical scaling */
  CoeffBank.CoeffValueBuf = (s16 *)
    XScaler_CoefValueLookup(ScalerInHeight,
        ScalerOutHeight,
        CoeffBank.TapNum,
        CoeffBank.PhaseNum);

  /* Load coefficient bank for Vertical Luma */
  XScaler_LoadCoeffBank(ScalerInstPtr, &CoeffBank);

  /* Vertical Chroma coeff bank is loaded only if chroma/luma
   * sharing flag is not set
   */
  if (!ChromaLumaShareCoeffBank)
    XScaler_LoadCoeffBank(ScalerInstPtr, &CoeffBank);
}

/*
 * Load phase-offsets into scaler
 */
StartFraction.LumaLeftHori = 0;
StartFraction.LumaTopVert = 0;
StartFraction.ChromaLeftHori = 0;
StartFraction.ChromaTopVert = 0;
XScaler_SetStartFraction(ScalerInstPtr, &StartFraction);

/*
 * Set up Aperture.
 */
Aperture.InFirstLine = 0;
Aperture.InLastLine = ScalerInHeight - 1;

Aperture.InFirstPixel = 0;
Aperture.InLastPixel = ScalerInWidth - 1;

Aperture.OutVertSize = ScalerOutHeight;
Aperture.OutHoriSize = ScalerOutWidth;


// Added by Xilinx 2012.12.10
Aperture.SrcVertSize = ScalerInHeight;
Aperture.SrcHoriSize = ScalerInWidth;

XScaler_SetAperture(ScalerInstPtr, &Aperture);
```

```
    /*
     * Set up phases
     */
    XScaler_SetPhaseNum(ScalerInstPtr, ScalerInstPtr->Config.MaxPhaseNum,
        ScalerInstPtr->Config.MaxPhaseNum);

    /*
     * Choose active set indexes for both vertical and horizontal directions
     */
    XScaler_SetActiveCoeffSet(ScalerInstPtr, COEFF_SET_INDEX,
        COEFF_SET_INDEX);

    /*
     * Enable the scaling operation
     */
    XScaler_EnableRegUpdate(ScalerInstPtr);

    return 1;
}
```

Video Subsystem Bandwidth Requirements

Video data is typically transmitted in contiguous bursts. Each burst comprises active pixel data. This data is transmitted in contiguous clock cycles which can be followed by clock cycles of no active data. These cycles of "no data" are called blanking periods. There are horizontal blanking periods which occur during each between video lines, and vertical blanking periods that equate to full video lines with no active pixel data at all.

To a memory subsystem, this translates to periods of bursts of video data the size of the active video frame size followed by burst gaps the length of the video blanking period. Therefore, for a given video frame, there are periods that require a certain peak bandwidth, or BW_{peak}, followed by quiescent periods of no data transmittal. This equates to a peak bandwidth requirement, or BW_{peak}.

BW_{peak} is calculated from the data width, or bits-per-pixel (bpp), and from the video pixel clock frequency, Fvid. F_{vid} can be calculated from the video frame rate (F_{frame}) measured in frames-per-second, the number of lines-per-frame (including blanking lines) and the number of pixel clock-cycles-per-line (including blanking clock cycles), shown in Equation 2-1.

$$F_{vid} = F_{frame} * N_{full\ lines} * N_{pixels} \qquad \textit{Equation 2-1}$$

The BW_{peak} is calculated by multiplying the Video Pixel clock frequency by the number of bits-per pixel, shown in Equation 2-2.

$$BW_{peak} = F_{vid} * bpp \quad \textit{Equation 2-2}$$

The average bandwidth requirement is defined as the overall number of bits within a frame over a one entire times the frame rate video frame period (not just during the bursts). This is the average bandwidth and is always lower than the peak bandwidth requirement. For a given video frame period, the average bandwidth is B_{Wave}. This is shown in Equation 2-3.

$$F_{ave} = F_{frame} * N_{active\ lines} * N_{active\ pixels} \quad \textit{Equation 2-3}$$

The B_{Wave} is calculated the same as BW_{peak} by multiplying the Video Pixel clock frequency by the number of bits-per pixel. This is shown in Equation 2-4.

$$BW_{ave} = F_{ave} * bpp \quad \textit{Equation 2-4}$$

It is important to keep the BW_{peak} and B_{Wave} in mind when designing video subsystems, as these numbers define the clock frequencies and data width of the video IP core(s) and of the memory subsystem.

Bandwidth and Clocking

Live Video to/from Memory

If a memory subsystem is connected to a video subsystem that drives a live video output or is driven by a live video input, the memory subsystem must be able to accommodate the peak frame bandwidth requirements.

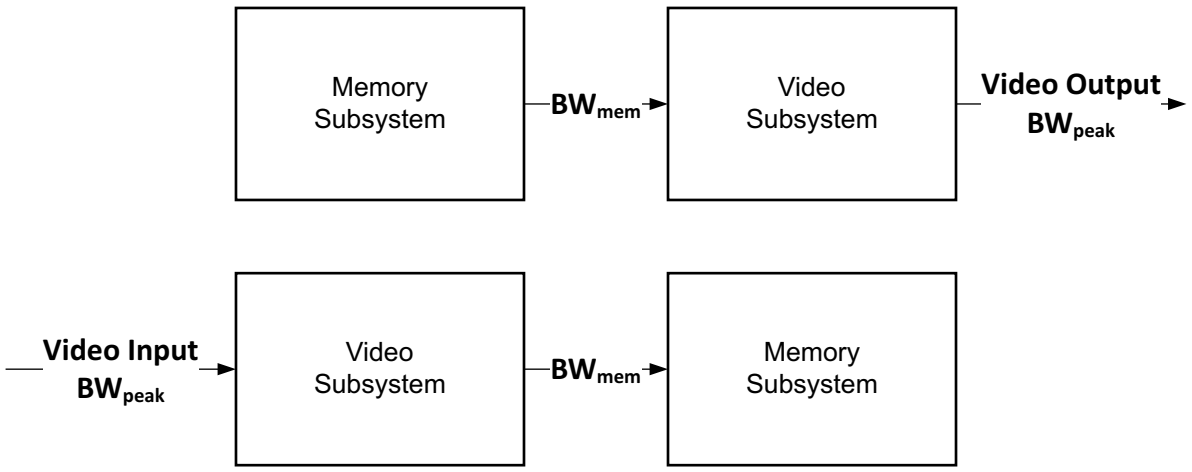

Figure 2-9: **Video Bandwidth and Live Video**

Memory to/from Memory

If a memory subsystem is connected to a video subsystem that writes to an external memory interface (to Frame Buffer) or reads from an external memory interface (from frame buffer) ONLY (thus, no live external video input/outputs), the memory subsystem must be only able to accommodate the average frame bandwidth requirements.

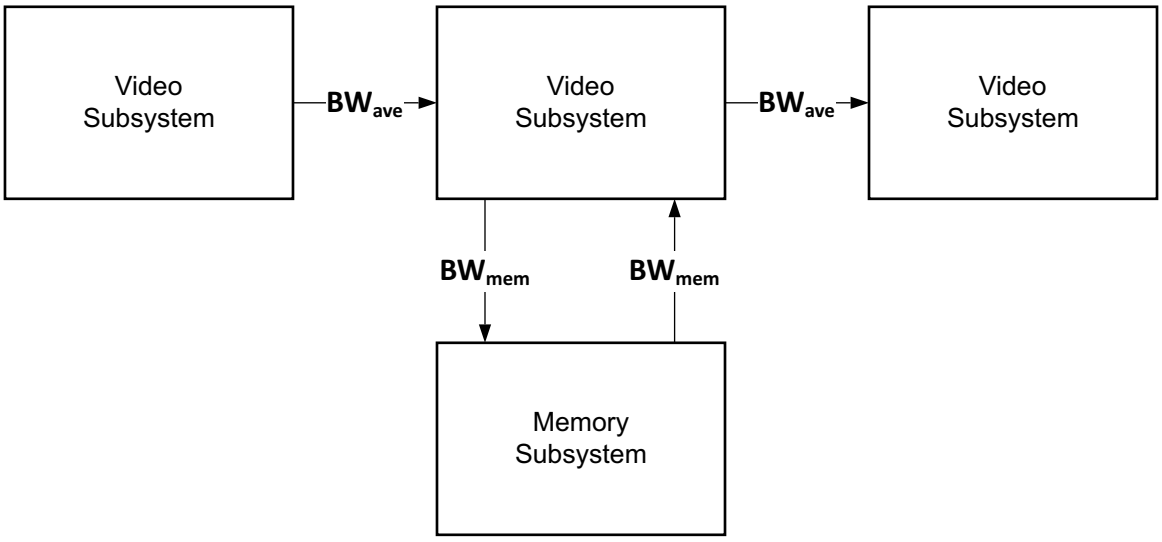

Figure 2-10: **Video Bandwidth and External Memory**

Bandwidth Examples

Scaling: Down-Scaling/Decimation

In the down-scaling system case, the input video frame is larger than the output video frame. The average bandwidth of the output is less than the input.

Down-scaling Memory-to-Memory

Figure 2-11 shows an example of down-scaling a video frame. It assumes that the video frame is read from external memory and written back to external memory. This allows for a slower minimum operating clock frequency and a lower bandwidth requirement.

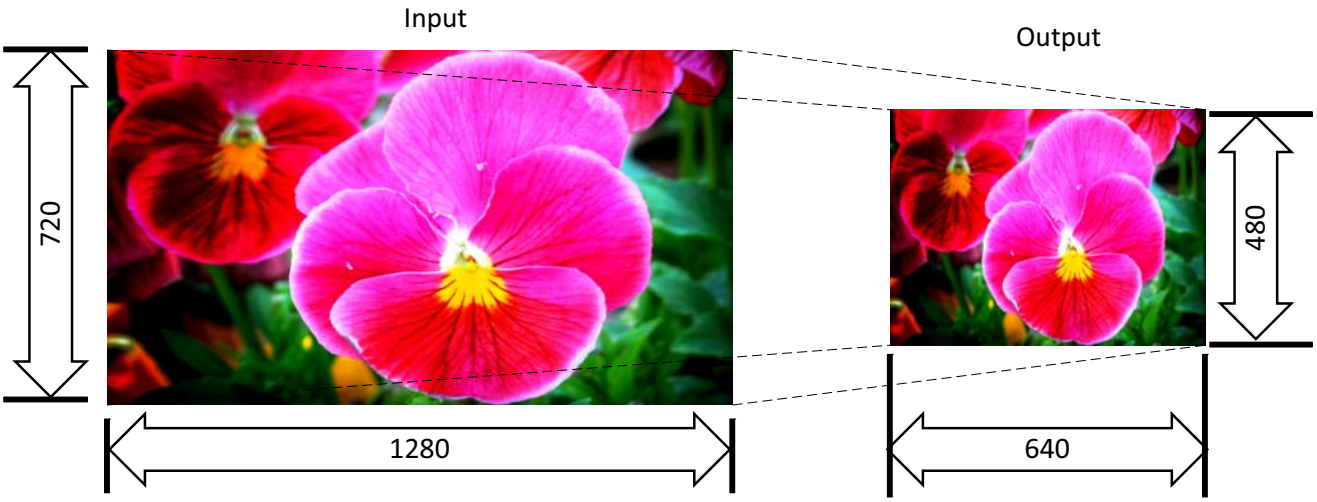

Figure 2-11: **Down-scaling Memory-to-Memory (720p@60 to 640x480p@60)**

For the example in Figure 2-11, Table 2-1 shows the input and output minimum frequency and minimum bandwidth requirements, assuming a data width of 16 and a 60 frames-per-second frame rate.

Table 2-1: **Down-scaling Mem-to-Mem Example Minimum Bandwidth and Frequency Requirements**

	Input	Output
Minimum Frequency	55.3 MHz	18.43 MHz
Minimum Bandwidth	0.88 Gb/s	0.29 Gb/s

Down-scaling Live External Video

Figure 2-12 shows an example of down-scaling a video frame. It assumes that the input video frame is from live external video and the output video frame is to live external video. The bandwidth requirement in this case is the peak bandwidth and has to take into account bursts of video at the higher frequency.

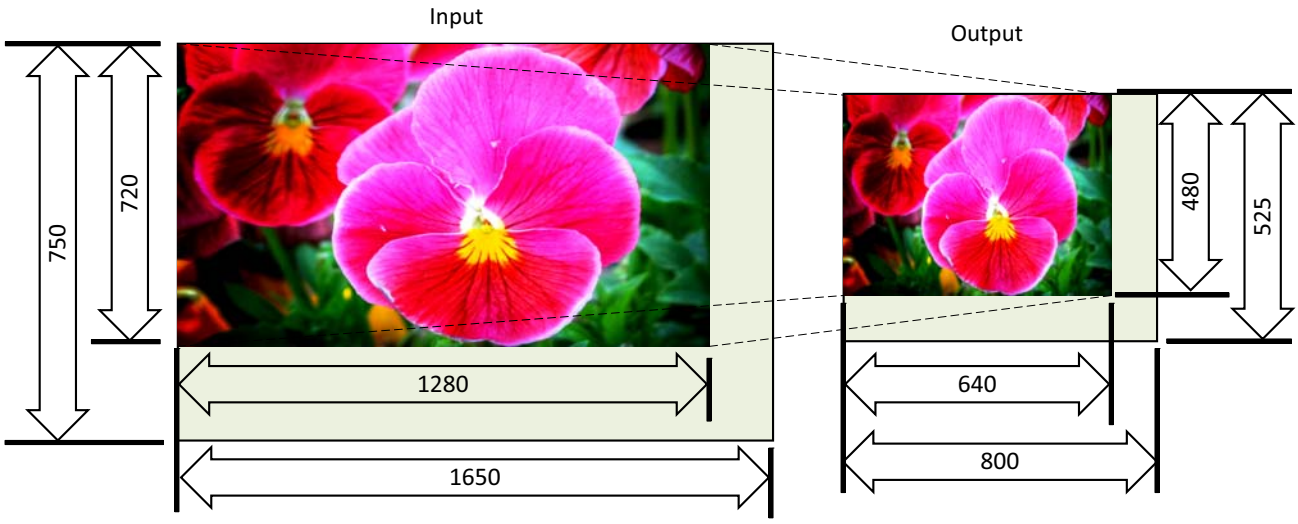

Figure 2-12: **Down-scaling Live Video (720p@60 to 640x480p@60)**

In the example in Figure 2-12, Table 2-2 shows the input and output minimum frequency and minimum bandwidth requirements, assuming a data width of 16 and a 60 frames-per-second frame rate.

Table 2-2: **Down-scaling Live-Video Example Minimum Bandwidth and Frequency Requirements**

	Input	Output
Minimum Frequency	74.25 MHz	25.20 MHz
Minimum Bandwidth	1.19 Gb/s	0.40 Gb/s

Down-scaling Example System

Figure 2-13 shows a video system that includes live-external video (peak) bandwidth and memory (average) bandwidth requirements.

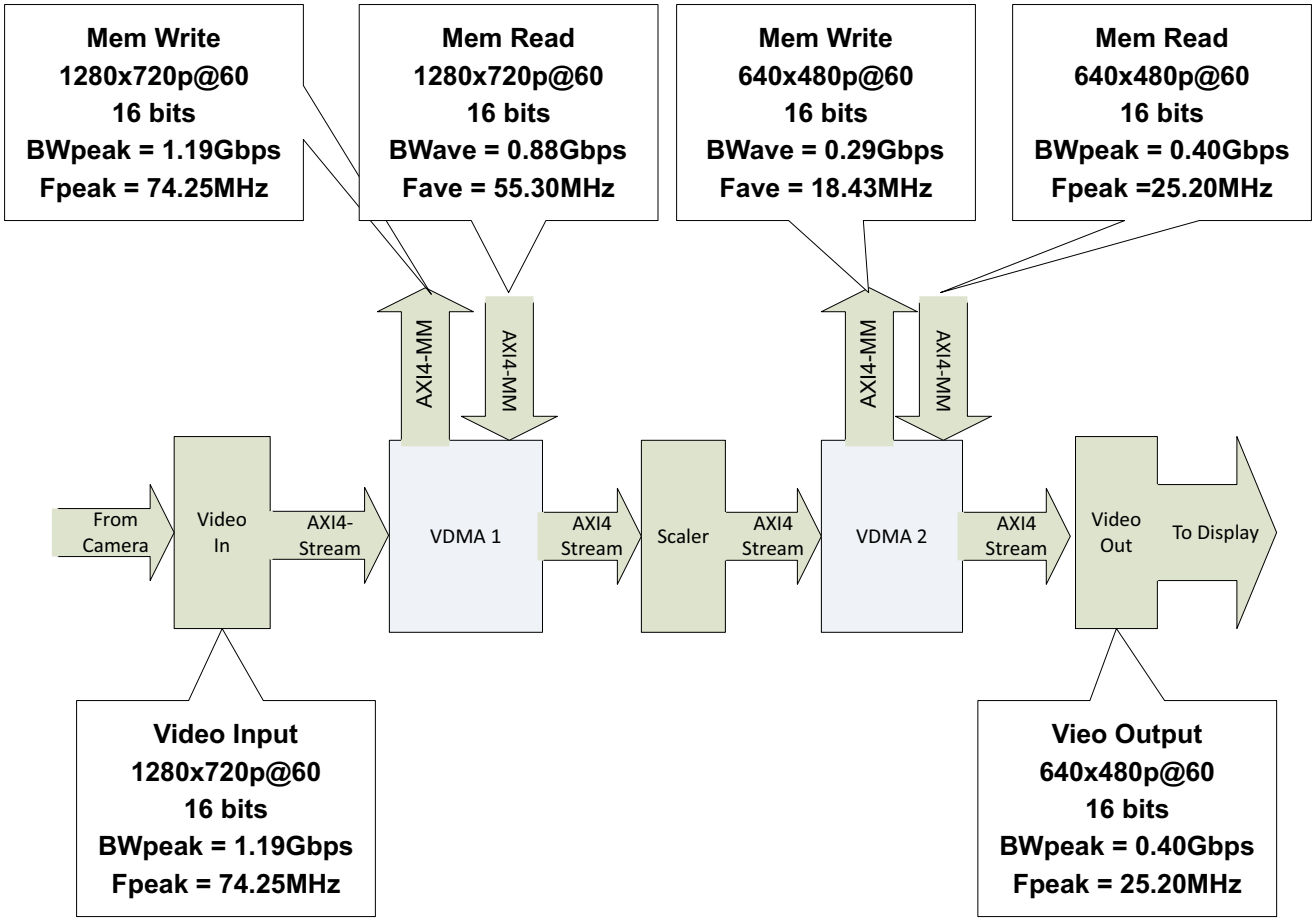

Figure 2-13: **Down-scaling (720p@60 to 640x480p@60) Subsystem Example**

In Figure 2-13, 720p live video frames are written to external memory with a bandwidth of 1.19 Gb/s. These frames can be read at an average bandwidth of 0.88 Gb/s. These frames are then downscaled and written at an average bandwidth of 0.29 Gb/s. The downscaled frames can then be read from external memory at a peak bandwidth of 0.40 Gb/s to display to external video.

Thus, video input bandwidth is BW_{peak} of input size. Video output bandwidth is BW_{peak} of output size. Intermediate memory read bandwidth is B_{Wave} of input size. Intermediate memory write bandwidth is B_{Wave} of output size.

Table 2-3: **Downscaling Subsystem Example Total Bandwidth and Minimum Frequency Requirements**

	Video Input (Memory Write)	Memory Read	Memory Write	Memory Read (Video Output)	Total/Maximum
Minimum Frequency	74.25 MHz	55.3 MHz	18.43 MHz	25.20 MHz	74.25 MHz (Max)
Minimum Bandwidth	1.19 Gb/s	0.88 Gb/s	0.29 Gb/s	0.40 Gb/s	2.76 Gb/s (Sum) 1.48 Gb/s (W) 1.28 Gb/s (R)

Up-Scaling

In the up-scaling case, the input video frame is smaller than the output video frame. The average bandwidth of the output is more than the input.

Up-scaling Memory-to-Memory

Figure 2-14 shows an example of up-scaling a video frame. It assumes that the video frame is read from external memory and written back to external memory. This allows for a slower minimum operating clock frequency and a lower bandwidth requirement.

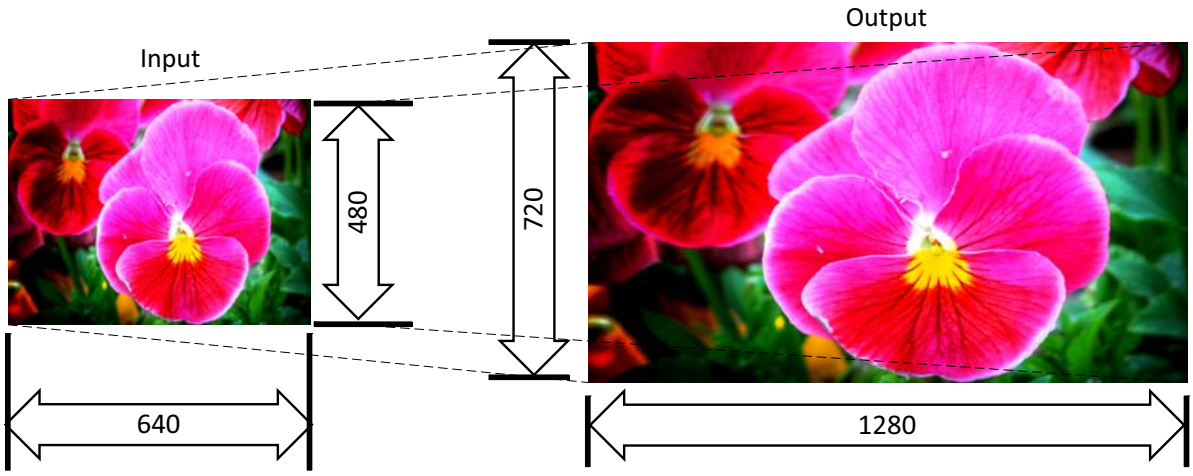

Figure 2-14: **Up-scaling Memory-to-Memory (640x480p@60 to 720p@60)**

In the example in Figure 2-14, the Table 2-4 shows the input and output minimum frequency and minimum bandwidth requirements, assuming a data width of 16 and a 60 frames-per-second frame rate.

Table 2-4: **Up-scaling Mem-to-Mem Example Minimum Bandwidth and Frequency Requirements**

	Input	Output
Minimum Frequency	18.43 MHz	55.3 MHz
Minimum Bandwidth	0.29 Gb/s	0.88 Gb/s

Up-scaling Live External Video

Figure 2-15 shows an example of up-scaling a video frame. It assumes that the input video frame is from live external video and the output video frame is to live external video. The bandwidth requirement in this case is the peak bandwidth and has to take into account bursts of video at the higher frequency.

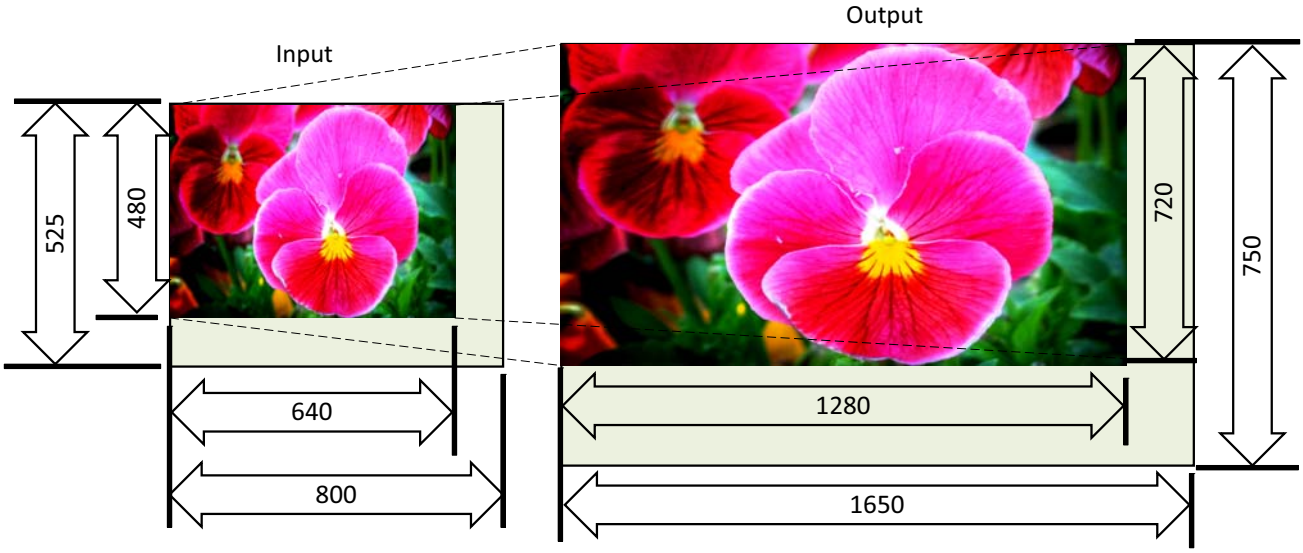

Figure 2-15: **Up-scaling Live Video (640x480p@60 to 720p@60)**

In the example in Figure 2-15 and Table 2-5 describe the input and output minimum frequency and minimum bandwidth requirements, assuming a data width of 16 and a 60 frames-per-second frame rate.

Table 2-5: **Up-scaling Live-Video Example Minimum Bandwidth and Frequency Requirements**

	Input	Output
Frequency	25.20 MHz	74.25 MHz
Bandwidth	0.40 Gb/s	1.19 Gb/s

Up-scaling Example System

Figure 2-16 shows a video system that includes live-external video (peak) bandwidth and memory (average) bandwidth requirements.

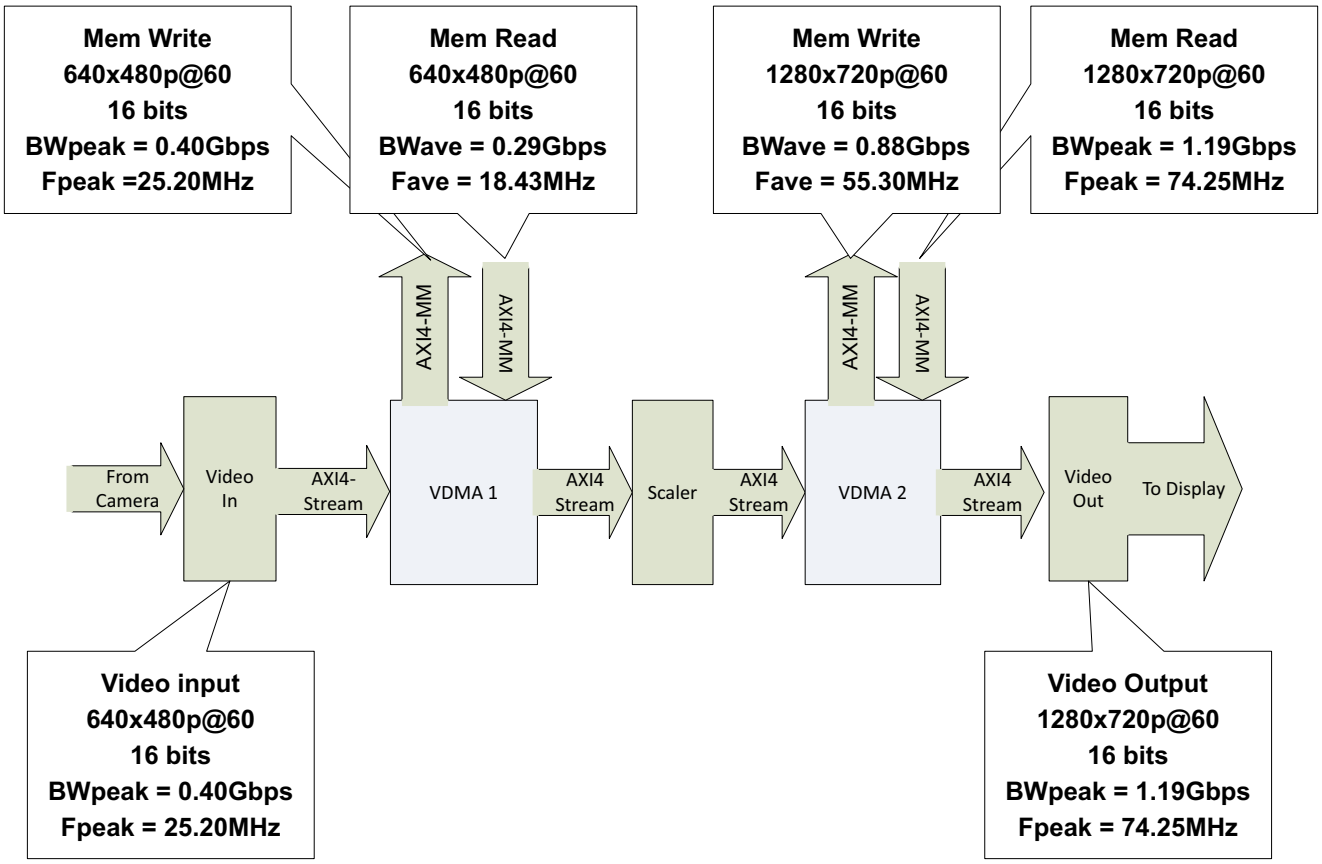

Figure 2-16: **Up-scaling (640x480p@60 to 720p@60) Subsystem Example**

In the Figure 2-16, 640x480p live video frames are written to external memory with a bandwidth of 0.40 Gb/s. These frames can be read at an average bandwidth of 0.29 Gb/s. These frames are then up-scaled and written at an average bandwidth of 0.88 Gb/s. The upscaled frames can then be read from external memory at a peak bandwidth of 1.19 Gb/s to display to external video.

Thus, video input bandwidth is BW_{peak} of input size. Video output bandwidth is BW_{peak} of output size. Intermediate memory read bandwidth is B_{Wave} of input size. Intermediate memory write bandwidth is B_{Wave} of output size.

Table 2-6: **Up-scaling Subsystem Example Total Bandwidth and Minimum Frequency Requirements**

	Video Input (Memory Write)	Memory Read	Memory Write	Memory Read (Video Output)	Total/Maximum
Minimum Frequency	25.20 MHz	18.43 MHz	55.3 MHz	74.25 MHz	74.25 MHz (Max)
Minimum Bandwidth	0.40 Gb/s	0.29 Gb/s	0.88 Gb/s	1.19 Gb/s	2.76 Gb/s (Sum) 1.28 Gb/s (W) 1.48 Gb/s (R)

Zoom

In the zoom system case, the input video frame is the same as the output video frame. The average bandwidth at the output is same as the input.

Zoom Memory-to-Memory

Figure 2-17 shows an example of zooming a video frame. It assumes that the video frame is read from external memory and written back to external memory. This allows for a slower minimum operating clock frequency and a lower bandwidth requirement.

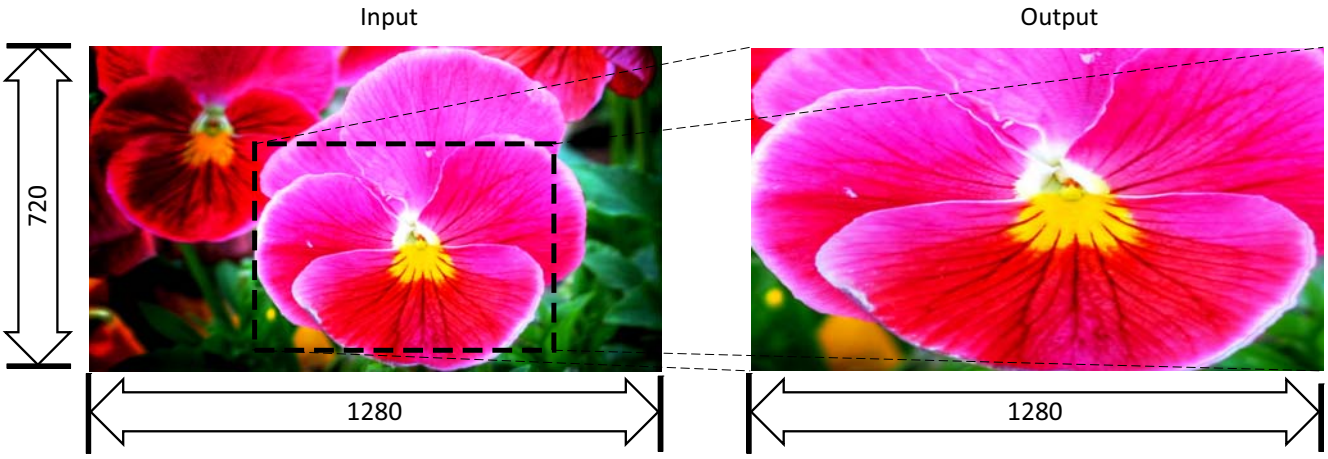

Figure 2-17: **Zoom Memory-to-Memory (to 720p@60 to 720p@60)**

In the example in Figure 2-17, Table 2-7 shows the input and output minimum frequency and minimum bandwidth requirements, assuming a data width of 16 and a 60 frames-per-second frame rate.

Table 2-7: **Zoom Mem-to-Mem Example Minimum Bandwidth and Frequency Requirements**

	Input	Output
Minimum Frequency	55.3 MHz	55.3 MHz
Minimum Bandwidth	0.88 Gb/s	0.88 Gb/s

Zoom Live External Video

Figure 2-18 shows an example of zooming a video frame. It assumes that the input video frame is from live external video and the output video frame is to live external video. The bandwidth requirement in this case is the peak bandwidth and has to take into account bursts of video at the higher frequency.

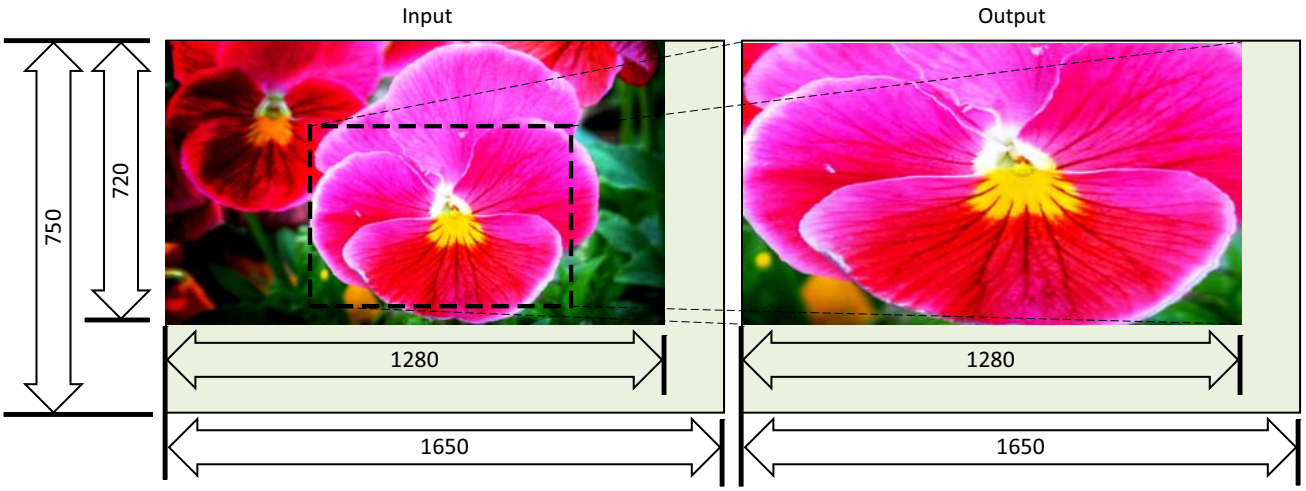

Figure 2-18: **Zoom Live Video (640x480p@60 to 720p@60)**

In the example in Figure 2-18 and Table 2-8 describe the input and output minimum frequency and minimum bandwidth requirements, assuming a data width of 16 and a 60 frames-per-second frame rate.

Table 2-8: **Zoom Live-Video Example Minimum Bandwidth and Frequency Requirements**

	Input	Output
Minimum Frequency	74.25 MHz	74.25 MHz
Minimum Bandwidth	1.19 Gb/s	1.19 Gb/s

Zoom Example System

Figure 2-19 shows a video system that includes live-external video (peak) bandwidth and memory (average) bandwidth requirements.

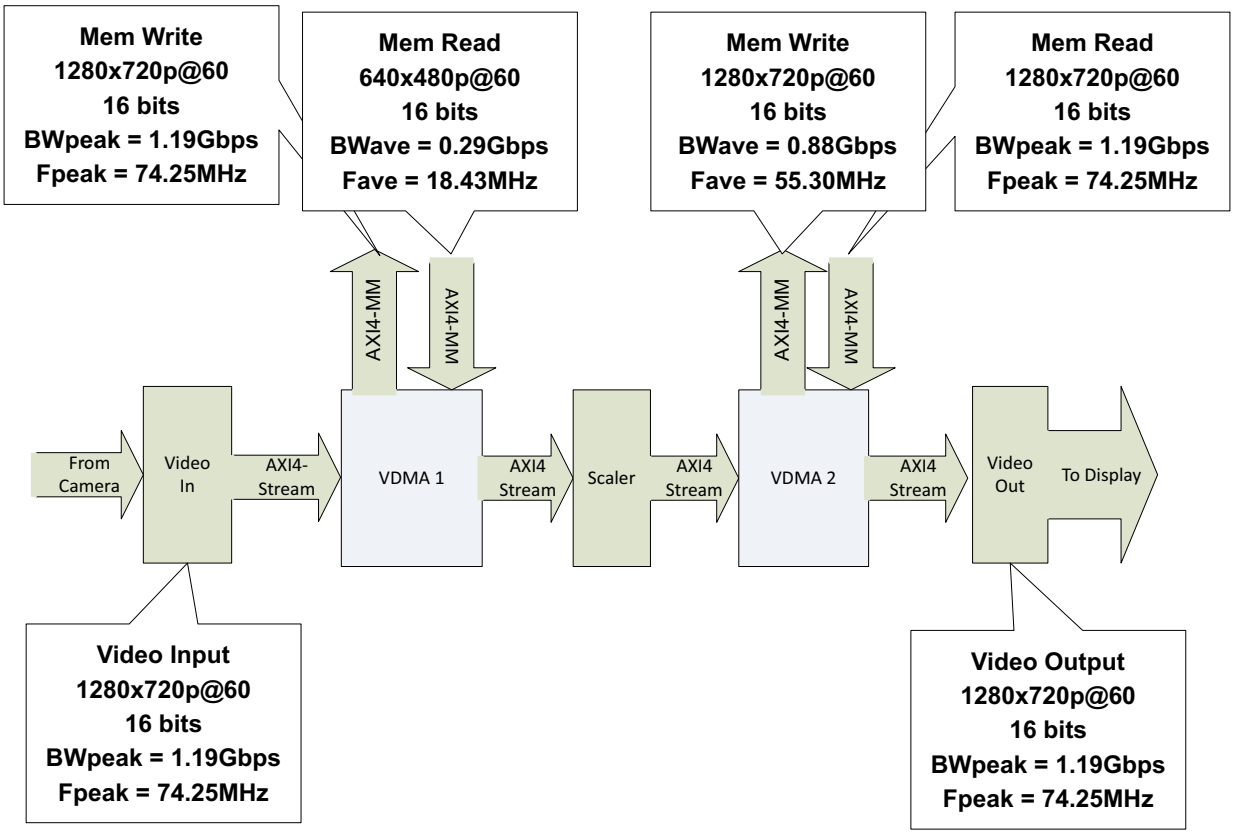

Figure 2-19: **Zoom (640x480p@60 to 720p@60) Subsystem Example**

In the Figure 2-19, 1280x720p live video frames are written to external memory with a bandwidth of 1.19 Gb/s. A 640x480 region in the video frame is read at an average bandwidth of 0.29 Gb/s. These frames are then up-scaled and written at an average bandwidth of 0.88 Gb/s. The upscaled frames can then be read from external memory at a peak bandwidth of 1.19 Gb/s to display to external video (Same as the input).

Thus, video input bandwidth is BW_{peak} of input size. Video output bandwidth is BW_{peak} of output size. Intermediate memory read bandwidth is B_{Wave} of input size. Intermediate memory write bandwidth is B_{Wave} of output size.

Table 2-9: **Zoom Subsystem Example Total Bandwidth and Minimum Frequency Requirements**

	Video Input (Memory Write)	Memory Read	Memory Write	Memory Read (Video Output)	Total/Maximum
Minimum Frequency	74.25 MHz	18.43 MHz	55.3 MHz	74.25 MHz	74.25 MHz Max
Minimum Bandwidth	1.19 Gb/s	0.29 Gb/s	0.88 Gb/s	1.19 Gb/s	3.55 Gb/s (Sum) 2.07 Gb/s (W) 1.48 Gb/s (R)

Typical Video Formats

Typical video formats and their operating frequency and bandwidth (average and peak) in Table 2-10.

Table 2-10: **Typical Video Format Sizes, Frequencies and Bandwidths**

Video Format						Frequency		Average Bandwidth BWave		Peak Burst Bandwidth BWpeak	
BPP	Active H	Active V	FPS	Full H	Full V	Min Frame/ Ave MHz	Min Line/Peak MHz	Gb/s	Gb/s2	Gb/s3	GB/s4
16	1920	1080	60	2200	1125	124.42	148.50	1.99	0.25	2.38	0.30
32	1920	1080	60	2200	1125	124.42	148.50	3.98	0.50	4.75	0.59
16	800	600	60	1056	628	28.80	39.79	0.46	0.06	0.64	0.08
32	800	600	60	1056	628	28.80	39.79	0.92	0.12	1.27	0.16
16	1280	720	60	1650	750	55.30	74.25	0.88	0.11	1.19	0.15
32	1280	720	60	1650	750	55.30	74.25	1.77	0.22	2.38	0.30
16	4096	2048	60	4300	2300	503.32	593.40	8.05	1.01	9.49	1.19
36	4096	2048	60	4300	2300	503.32	593.40	18.12	2.26	21.36	2.67
16	640	480	60	800	525	18.43	25.20	0.29	0.04	0.40	0.05
16	720	480	60	858	525	20.74	27.03	0.33	0.04	0.43	0.05
16	720	576	50	864	625	20.74	27.00	0.33	0.04	0.43	0.05
16	1024	768	60	1344	806	47.19	65.00	0.75	0.09	1.04	0.13
16	1280	768	60	1440	790	58.98	68.26	0.94	0.12	1.09	0.14
16	1280	800	60	1680	831	61.44	83.76	0.98	0.12	1.34	0.17
16	1280	960	60	1800	1000	73.73	108.00	1.18	0.15	1.73	0.22
16	1280	1024	60	1688	1066	78.64	107.96	1.26	0.16	1.73	0.22
16	1440	900	60	1904	934	77.76	106.70	1.24	0.16	1.71	0.21
16	1680	1050	60	2240	1089	105.84	146.36	1.69	0.21	2.34	0.29

IP Development Guide

IP Parameterization

General IP configuration parameters are not covered in this specification. However, commonly used video IP parameters generally are listed in Table 3-1.

Table 3-1: **Standard Video IP Parameters**

Parameter Name	Parameter Function
C_HAS_AXI4_LITE	0 or 1 determines whether the core has an AXI4-Lite control interface
C_ACTIVE_ROWS	Number of active (non-blank) scan lines per frame
C_ACTIVE_COLS	Number of active (non-blank) pixels per scan line
C_MAX_COLS	Maximum number of active (non-blank) pixels per scan line supported by a particular core instance

Only one video format can be supported in video IP core systems that use an AXI-4 interface without an embedded processor. For this configuration (`C_HAS_AXI4_LITE=0`), you can define the supported resolution through generic parameters `C_ACTIVE_ROWS` and `C_ACTIVE_COLS` defined in the core GUI. When `C_HAS_AXI4_LITE=0`, `C_ MAX_COLS` should be equal to `C_ACTIVE_COLS`.

When an embedded processor is present and the Video core is instantiated with an AXI4-Lite interface (`C_HAS_AXI4_LITE=1`), generic parameters `C_ACTIVE_ROWS` and `C_ACTIVE_COLS` assign default values to control registers to define the active resolution. As an upper bound on the active scanline length supported by the core instance, `C_MAX_COLS` is used to define line buffer depths, which have a direct effect on block RAM footprint. For example, a video core, instantiated to service 720p video (1650 total pixels, 1280 active pixels per line), needs to have `C_MAX_COLS` set to 1280. This core instance is not be able to service 1080p video, but works with 720p or any lower resolutions, such as 480p, when the `active_size` register in the AXI4-Lite control interface is set according to 720p or 480p.

`C_MAX_COLS` refers to the maximum number of non-blank pixels a core instance must service. This parameter is often used to allocate block RAMs for line buffers within the core. For example, a core instance targeting resolutions up to 720p must have this parameter set to 1280.

General IP Structure

Video IP cores should provide an AXI4-Lite interface option to allows dynamic read and write processing parameters, status and control data, and timing parameters. For embedded systems using either a processor or dedicated IP acting as the AXI4-Lite master, an AXI4-Lite interface should be provided with a standardized register API. For systems without an embedded processor, video cores should provide a way to be instantiated, supporting one fixed video resolution.

Figure 3-1 is a schematic for a typical video processing core with one AXI4-Stream slave input, one AXI4-Stream master output, and an AXI4-Lite interface. In this example, the IP core processing the input and the output AXI4-Stream interfaces are apart of the same clock domain (ACLK), but the AXI4-Lite processor interface input is in the AXI4-Lite processor clock domain. Typically the AXI4-Lite interface does not use the same clock as the AXI4-Stream video slave and master interfaces. Therefore, the IP should contain Clock-Domain Crossing (CDC) logic to facilitate re-sampling the AXI4-Lite register data to the processing core clock domain.

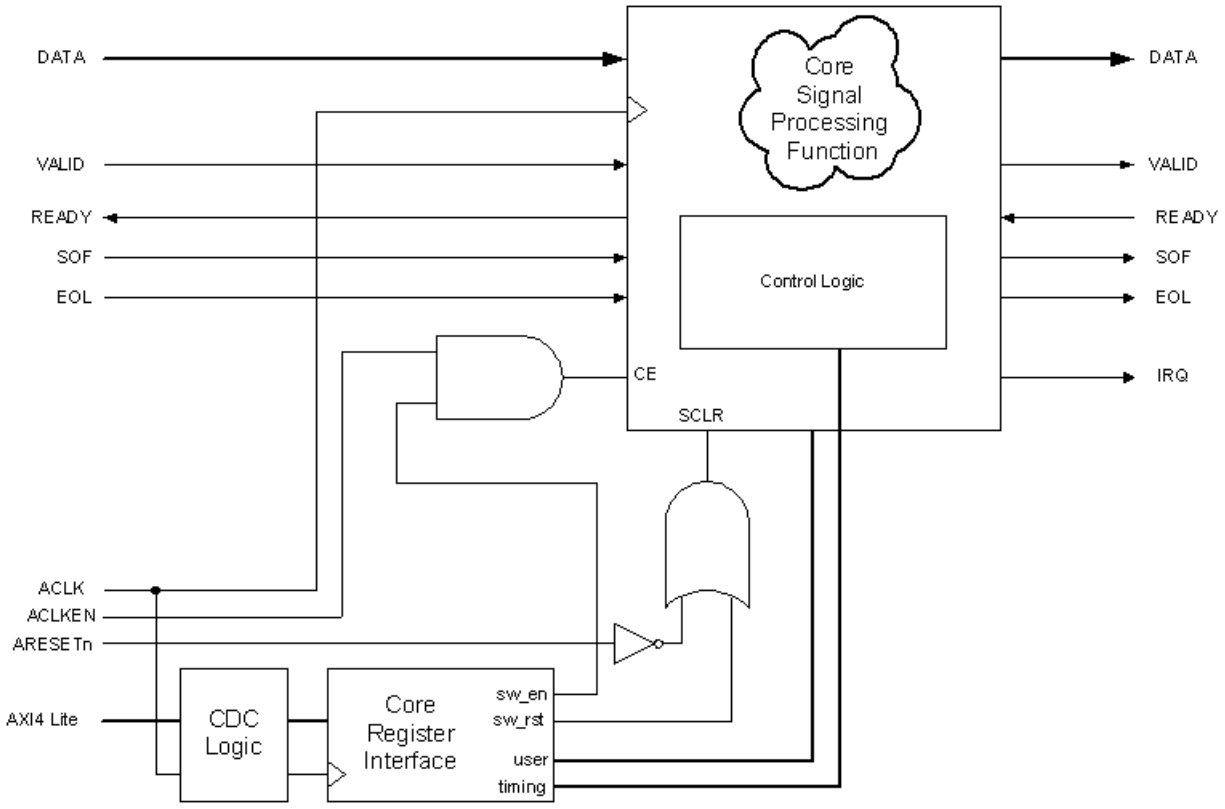

Figure 3-1: **General Video IP Structure with AXI4-Lite and AXI4-Stream Interfaces**

All video IP cores should contain control logic to govern the propagation of VALID and READY signals, enable/disable/initialize the core Signal Processing Function, manage

internal buffers, generate SOF and EOL signals, and monitor error conditions. See READY – VALID Propagation and Flushing Pipelined Cores for more information.

AXI4-Lite Interface

Many video applications have an embedded processor that can dynamically monitor and control processing parameters within IP cores. The AXI4-Lite interface provides a standardized API across which core functionality can be controlled and monitored. Layers of the API consist of a memory-mapped interface with programmable registers, a low level driver to identify physical memory locations, and higher level driver functions to control multiple registers or complex processes. The proposed standard set of memory mapped registers is described in Table 3-2.

Table 3-2: **Standard Video IP Registers**

Offset	Function	Default	Access	Bit-field Definitions
0x0000	CONTROL	0	R/W	Bit 0: SW_ENABLE Bit 1: REG_UPDATE Bit 4: BYPASS (Optional. See Core Bypass Option.) Bit 5: TEST_PATTERN (Optional. See Built in Test-Pattern Generator.) Bit 31: SW_RESET (1: reset)
0x0004	STATUS	0	R/W	Bit 0: Frame processing Started Bit 1: Frame Processing Complete Bits 2-15: Core specific Status Flags Bit 16: Slave0 error Bit 17: Slave1 error (Optional) Bit 18: Slave2 error (Optional) Bit 19: Slave3 error (Optional)

Table 3-2: **Standard Video IP Registers *(Cont'd)***

Offset	Function	Default	Access	Bit-field Definitions
0x0008	ERROR	0	R/W	Bit 0: Slave0 EOL early Bit 1: Slave0 EOL late Bit 2: Slave0 SOF early Bit 3: Slave0 SOF late Bit 4: Slave1 EOL early (Optional) Bit 5: Slave1 EOL late (Optional) Bit 6: Slave1 SOF early (Optional) Bit 7: Slave1 SOF late (Optional)
0x000C	IRQ_ENABLE	0	R/W	Bit 0-31: Interrupt enable bits corresponding to STATUS conditions
0x0010	VERSION		R	31-16: Core version in 4bits. 4bits format. 0-15: CRC generated by CORE Generator (Optional. See Version Register.)
0x0014	SYSDEBUG0	0	R	Frame Throughput monitor (Optional)
0x0018	SYSDEBUG1	0	R	Line Throughput monitor (Optional)
0x001C	SYSDEBUG2	0	R	Pixel Throughput monitor (Optional)
0x0020 0x005C	Timing Register Set 0		Application Dependent	See Timing Representation.
0x0060 0x009C	Timing Register Set 1		Application Dependent	Optional for IP using multiple interfaces with different Encoding or Timing.
0x00A0 - 0x00FC	Reserved			
0x0100 0x3FFC	Core Specific Registers		Application Dependent	Defined in Core Data Sheets

For more information on optional debugging, see Debugging Features.

Control Register

The SW_ENABLE flag, located on bit 0 of the CONTROL register, allows the core to be dynamically enabled or disabled. Disabling the core from software has similar effects as deasserting ACLKEN. When disabled, the core AXI4-Lite decoding units remain active to facilitate re-enabling the core. The default value of Software Enable is 1 (enabled).

Flags of the CONTROL register are not buffered, which means changes take effect immediately. The application or higher-level driver functions need to deassert these flags to re-enable status/error acquisition.

Status and Error Registers

When using the AXI4-Lite interface, it is recommended that processing events and errors assert STATUS and ERROR register flags. The event flags should remain set until the application clears the flags, or the core is reset. STATUS register flags should be able to trigger interrupts through an IRQ pin. Bits of the STATUS and ERROR registers should be individually toggled when the application writes a '1' to the appropriate bit position of the STATUS and ERROR registers.

If the core does not provide an AXI4-Lite interface, the IP should be configured to provide notification of critical status and error events through a dedicated set of pins. These pins can be connected to an external interrupt controller (INTC) core in an EDK system to facilitate interrupt requests, identification, and clearing of interrupt sources. For this application, it is recommended that the dedicated output signals remain asserted only as long as the status or error event persists.

IRQ_ENABLE (0x000C) Register

Any bits of the STATUS register can generate a host-processor interrupt request through the IRQ pin. The Interrupt Enable register facilitates selecting which bits of STATUS register asserts IRQ. Bits of the STATUS registers are masked by (AND) corresponding bits of the IRQ_ENABLE register and the resulting terms are combined (OR) together to generate IRQ. For more information, see Debugging Features.

Version (0x0010) Register

Bit fields of the Version register facilitate software identification of the exact version of the hardware peripheral incorporated into a system. The core driver can use this Read-Only value to verify that the software version is matched to the hardware. For more information, see Debugging Features.

SYSDEBUG0 (0x0014) Register

The SYSDEBUG0, or Frame Throughput Monitor, register indicates the number of frames processed because power-up or the last time the core was reset. The SYSDEBUG registers can be useful to identify external memory, Frame buffer, or throughput bottlenecks in a video system. For more information, see Debugging Features.

SYSDEBUG1 (0x0018) Register

The SYSDEBUG1, or Line Throughput Monitor, register indicates the number of lines processed because power-up or the last time the core was reset. The SYSDEBUG registers can be useful to identify external memory, Frame buffer, or throughput bottlenecks in a video system. For more information, see Debugging Features.

SYSDEBUG2 (0x001C) Register

The SYSDEBUG2, or Pixel Throughput Monitor, register indicates the number of pixels processed because power-up or the last time the core was reset. The SYSDEBUG registers can be useful to identify external memory, Frame buffer, or throughput bottlenecks in a video system. For more information, see Debugging Features.

Register Synchronization

Most control registers that provide frame-by-frame control over processing should be double-buffered to ensure no image tearing occurs if register values are modified while a frame is being processed. Exceptions are registers which command immediate actuation (CONTROL, STATUS, ERROR and IRQ_ENABLE registers) or need to be changed multiple times within a frame (a readout or coefficient address register). With double buffering, register writes are updating the first set of registers while the processing core uses values from a second set of registers. All writable registers are also readable. Any reads from writable registers return values that are stored in the first set of registers.

A semaphore mechanism allows you to update multiple registers without having all updates take place within a single frame or between frames.

Values from the first register set should be copied over (committed) to the second register set when processing cores receive the SOF signal and semaphore flag REG_UPDATE, located on bit 1 of register CONTROL, is set.

deasserting REG_UPDATE allows applications to modify multiple registers at any time without causing any artifacts with incomplete intra-frame updates. By asserting REG_UPDATE, congruently updated registers are being used for the subsequent frames starting at the next frame boundary.

Timing Representation

Timing information captures the phase/edge relationships between four periodic timing signals:

- Vertical Sync (VSync)
- Horizontal Sync (HSync)
- Vertical Blank (VBlank)
- Horizontal Blank (HBlank)

Timing detector/timing generator modules provided as part of the Xilinx Video Timing Controller core measure and regenerate timing signals. For an embedded processor with AXI4-Lite interface, measured timing information is accessible through a standardized register set, described inTable 3-3.

Blank/Sync Polarities

The input interface core automatically detects if timing signals (VSync, HSync, VBlank, HBlank) are inverted. Periodic sync pulses are defined as Active Low if the low portion of the signal is shorter than the high portion (signal pulses low). Bits 0 and 1 of timing variable POLARITY correspond to VSync and HSync respectively, and should be set to 1 when Active Low sync pulses are detected or to 0 when Active Low sync pulses are not detected

Periodic Blank signals are defined Active Low if the low portion of the signal is shorter than the high portion because an active area is expected to be longer than the blanked area. Bits 2 and 3 of timing variable POLARITY correspond to VBlank and HBlank respectively, and should be set to 1 when active low blank signals are detected or 0 when Active Low blank signals are not detected.

Description of Timing Variables

A frame period for progressive video is defined by the number of video clock cycles between Vsync pulses. Similarly, a field period for interlaced video is defined by the number of video clock cycles between Vertical Sync pulses.

The field periods for even (F0) and odd (F1) fields can differ. A frame period for interlaced video is defined by the sum of two subsequent (odd + even) field periods. The frame periods for both interlaced and progressive video is expected to be constant for any given video format.

The intervals when both HBlank and VBlank are inactive mark the active video area of the frame, where pixel data is considered valid and should be translated from a periodic standard such as DVI to AXI4-Stream.

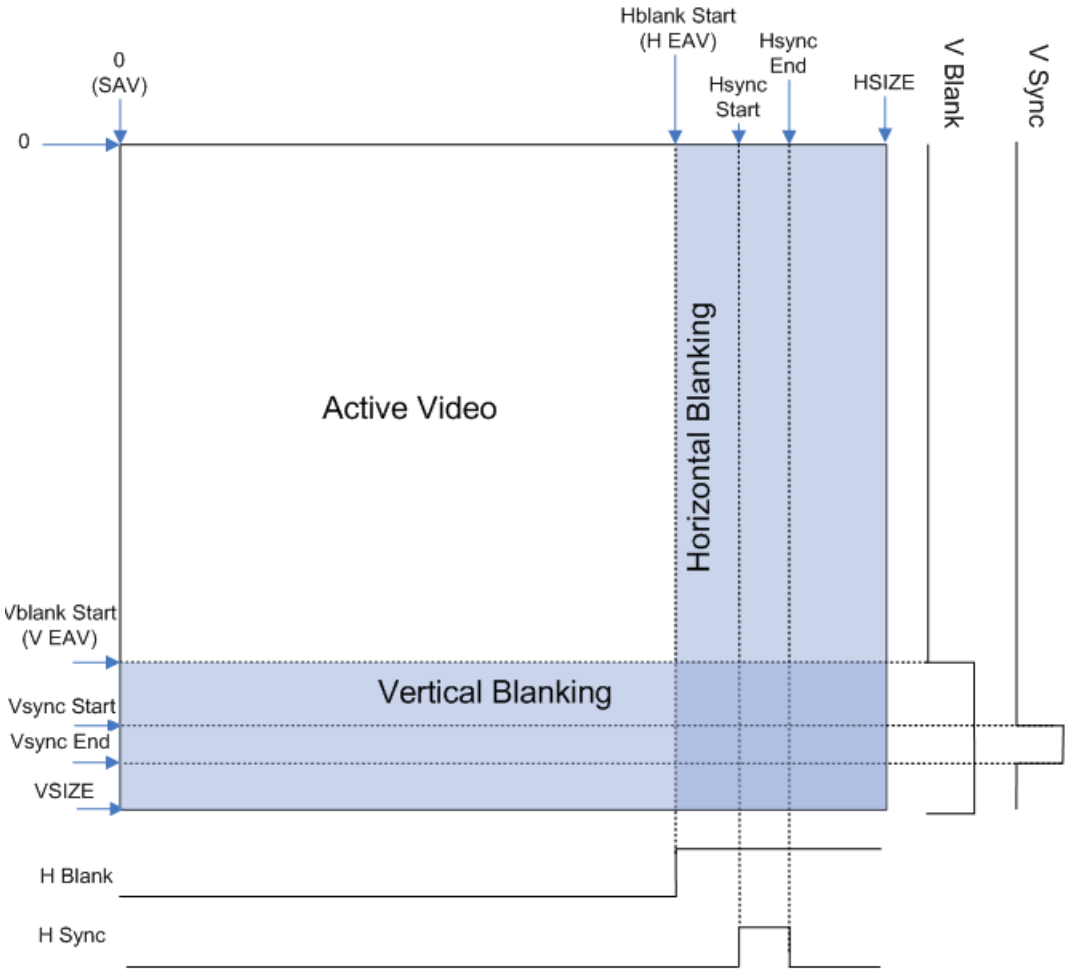

Figure 3-2: **Definition of Timing Variables – Falling Edge of Blanks**

The frame period contains blank and active areas and can be visualized as a set of rectangles, as seen in Figure 3-2. In the top-left corner of the frame, pixel index 0 (scan line index 0) is designated to be the first active pixel on the first complete active line.

The total number of scan lines per frame is defined as the number of scan-lines per frame, or VSIZE. The timing variable VSIZE reflects the total number of active and blank lines per frame. The index of the last scan line in a frame is VSIZE-1.

The number of video clock cycles between the HBlank pulses is expected to be equal to the number of video clock cycles between the HSync pulses in each field. The timing variable HSIZE reflects the total number of active and blank pixels per scan line. The index of the last pixel in scan lines is HSIZE-1.

The Xilinx Video Timing Controller IP works with complete scan lines, so the total number of video clock cycles in a frame period is expected to be an integer multiple of the total number of pixels per scan line (HSIZE * VSIZE).

For progressive video, the period between the VBlank pulses is expected to have the same number of video clock cycles as the period between the VSync pulses. For interlaced video, the number of total scan lines in even and odd fields can differ. Therefore, two sets of timing registers (F0 for even fields and F1 for odd fields) keep track of timing variables for interlaced video fields.

For progressive video, only the F0 bank of timing registers are used.

The falling and rising edges of VBlank might not coincide with the falling edge of HBlank, which could be visualized as VBlank falling on a pixel position other than 0 in a scan line (Figure 3-2). Also, the phase difference between VBlank and HBlank can change between even and odd fields. This phase difference between the falling and rising edges of VBlank is captured in the nibbles of the registers F0_VBLANK_H and F1_VBLANK_H.

The phase relationships of the VSync and HSync signals can be arbitrary in relationship to the first active pixel, the origin of the V/H coordinate system (Figure 3-2), and might be different between even and odd fields. Nibbles in registers F0_VSYNC_V and F0_VSYNC_H capture the horizontal and vertical positions of falling and rising edges of VSYNC for even fields. Similarly, nibbles in registers F1_VSYNC_V and F1_VSYNC_H capture the horizontal and vertical positions of falling and rising edges of VSYNC for odd fields.

The scan line index where VBlank transitions high1 (VBlank start) marks the vertical end of the active area and the start of the vertical blank area. The pixel index where HBlank transitions high1 (HBlank start) marks the horizontal end of the active area, and the start of the horizontal blank area.

Nibbles of timing registers ACTIVE_SIZE denote the vertical (number of scan lines), and horizontal sizes (number of pixels) in the active area.

Table 3-3: **Standardized Timing Registers**

Offset	Name	Function	Bit fields
0x0020	ACTIVE_SIZE	Horizontal and Vertical Frame Size (without blanking)	15-0: Horizontal active frame size 31-16: Vertical active frame size
0x0024	TIMING_STATUS	Timing Measurement Status	0: LOCKED 1: VBLANK_START_DETECT 2: VBLANK_END_DETECT
0x0028	ENCODING	Frame encoding	0-3: VIDEO_FORMAT 4-5: NBITS 6: INTERLACED/Progressive(0) 7: FIELD_PARITY 8: CHROMA_PARITY
0x0032	POLARITY	Blank, Sync polarities	0: Vertical Blank pulse polarity 1: Horizontal Blank pulse polarity 2: Vertical Sync polarity 3: Horizontal Sync polarity

Table 3-3: **Standardized Timing Registers *(Cont'd)***

Offset	Name	Function	Bit fields
0x0030	HSIZE	Horizontal Frame Size (with blanking)	15-0: Horizontal frame size
0x0034	VSIZE	Vertical Frame Size (with blanking)	15-0: Vertical frame size for field 0 31-16: Vertical frame size for field 1
0x0038	HSYNC	Start and end cycle index of HSync	15:0: Start cycle index of HSync. 31-16: End cycle index of HSync.
0x003C	F0_VBLANK_H	Start and end cycle index of VBlank for field 0.	15:0: Start cycle index of VBlank 31-16: End cycle index of VBlank
0x0040	F0_VSYNC_V	Start and end line index of VSync for field 0.	15:0: Start line index of VSync 31-16: End line index of VSync
0x0044	F0_VSYNC_H	Start and end cycle index of VSync for field 0.	15:0: Start cycle index of VSync 31-16: End cycle index of VSync
0x0048	F1_VBLANK_H	Start and end cycle index of VBlank for field 1.	15:0: Start cycle index of VBlank 31-16: End cycle index of VBlank
0x004C	F1_VSYNC_V	Start and end line index of VSync for field 1.	15:0: Start line index of VSync 31-16: End line index of VSync
0x0050	F1_VSYNC_H	Start and end cycle index of VSync for field 1.	15:0: Start cycle index of VSync 31-16: End cycle index of VSync
0x0058	Reserved		
0x005C	Reserved		

ACTIVE_SIZE (0x0020) Register

The ACTIVE_SIZE register encodes the number of active pixels per scan line and the number of active scan lines per frame. The lower half-word (bits 12:0) encodes the number of active pixels per scan line. Supported values should be between 32 and the value provided in the **Maximum number of pixels per scan line** field in the GUI. The upper half-word (bits 28:16) encodes the number of active pixels per scan line. Supported values should be 32 to 7680. To avoid processing errors, restrict values written to ACTIVE_SIZE to the range supported by the core instance.

Frame Encoding

Bits 0 to 3 (VIDEO_FORMAT) define the sampling structure of video using the video format codes (VF) defined in Table 1-4. Bits 4-5 define the data representation, the number of bits per component channel, as defined in Table 3-4.

Table 3-4: **Data Representation Codes**

ENCODING[5:4]	Bits per Component Channel
00	8
01	10
10	12
11	16

Bit 6 (INTERLACED) should be set if the video processed is interlaced (1). For progressive video, this bit should be set to 0. Corresponding Bit 7, indicates field polarity (0: even field, 1: odd field) if interlaced video is used. Processing cores should not expect the host processor to update this register value on a frame-by-frame basis. Instead, the IP is expected to toggle automatically after processing fields, using the programmed value as the initial value for the first field after the value is committed.

Bit 8 (CHROMA_PARITY) of the ENCODING register specifies whether the first line of video contains chroma information (1) or not (0) when YUV 420 encoded video is being processed. Processing cores should not expect the host processor to update this register value on a line-by-line basis to reflect whether the current line contains chroma or not. Instead, the IP is expected to toggle automatically after each line was processed, using the programmed value as the initial value for the first line of the first frame after the value is committed. Table 3-5 provides example values for timing variable assignments for typical video standards using 8 bit data.

Table 3-5: **Typical Values for Timing Variables**

Name	720p@59.94/60 RGB	1080p@59.94/60 YUV422	1080i@59.94/60 YUV420
ENCODING	0x0000_0002	0x0000_0000	0x0000_0043
POLARITY	0x0000_000F 0: VB Active-High 1: HB Active-High 2: VS Active-High 3: HS Active-High	0x0000_000F 0: VB Active-High 1: HB Active-High 2: VS Active-High 3: HS Active-High	0x0000_000F 0: VB Active-High 1: HB Active-High 2: VS Active-High 3: HS Active-High
ACTIVE_SIZE	0x02D0_0500 15-0: HSIZE = 1280 31-16: VSIZE = 720	0x0438_0780 15-0: HSIZE = 1920 31-16: VSIZE = 1080	0x021C_0780 15-0: HSIZE = 1920 31-16: VSIZE = 540
HSIZE	0x0000_0672 15-0: HSIZE_F0= 1650 31-16: Reserved	0x0000_0898 15-0: HSIZE_F0 = 2200 31-16: Reserved	0x0000_0898 15-0: HSIZE_F0= 2200 31-16: Reserved
VSIZE	0x0000_02EE VSIZE_F0 = 750 VSIZE_F1 = 0	0x0000_0465 VSIZE_F0 = 1125 VSIZE_F1 = 0	0x0233_0232 VSIZE_F0 = 562 VSIZE_F1 = 563
HSYNC	0x0596_056E 15-0: START = 1390 31-16: END = 1430	0x0804_07D8 15-0: START = 2008 31-16: END = 2052	0x0804_07D8 15-0: START = 2008 31-16: END = 2052

Table 3-5: **Typical Values for Timing Variables**

Name	720p@59.94/60 RGB	1080p@59.94/60 YUV422	1080i@59.94/60 YUV420
F0_VBLANK_H	0x0000_0000	0x0000_0000	0x0000_0000 15-0: H_START = 0 31-16: H_END = 0
F0_VSYNC_V	0x02DA_02D5 15-0: START = 725 31-16: END = 730	0x0441_043C 15-0: START = 1084 31-16: END = 1089	0x0223_021E 15-0: START = 542 31-16: END = 547
F0_VSYNC_H	0x0000_0000	0x0000_0000	0x0000_0000 15-0: H_START = 0 31-16: H_END = 0
F1_VBLANK_H	0x0000_0000	0x0000_0000	0x0000_0000 15-0: H_START = 0 31-16: H_END = 0
F1_VSYNC_V	0x0000_0000	0x0000_0000	0x0223_021E 15-0: START = 542 31-16: END = 547
F1_VSYNC_H	0x0000_0000	0x0000_0000	0x044C_044C 15-0: H_START = 1100 31-16: H_END = 1100

Input/Output Timing

The recommended design convention for AXI4-Stream component interfaces suggests that outputs should be registered or driven directly by flip-flops or FIFO/block RAM primitives. Ideally, inputs are also registered but can be combinatorial. Combinatorial inputs can limit Fmax so the amount of combinatorial logic present on inputs should be limited.

There must be no combinatorial paths between input and output signals on either master or slave interfaces. Combinatorial paths between input and output signals are not permitted across separate AXI4-Stream interfaces. In some cases, outputs driven by combinatorial logic are a suitable design choice or a reasonable design trade-off, such as when latency is critical. The IP core data sheet describes AXI4-Stream output signals that are not registered.

Buffering Requirements

The output interface module does not start generating valid output frames until it receives valid data on its input AXI4-Stream interface. However, after periodic output frame generation starts, all cores in the processing pipeline should be able to provide data at the rate required by the output standard.

For most output standards three different data rates should be defined. As an example, 720p30 video over DVI rates are used. Table 3-6 describes the three data rates.

Table 3-6: **Output Data Rates**

Pixel Rate	Description
Active	Within the active portion of each row, pixels are sent back to back on each clock cycle, at 37.125 MHz.
Line	Active video lines typically contain active and non-active (horizontally blanked) periods. As no pixels need to be transmitted in the non-active period, the average data rate within an active line is less than the active pixel rate. For 720p30 over DVI, this rate is 28.8 Msps.
Frame	Video frames typically contain active, and non-active (vertically blanked) periods. As no pixels need to be transmitted in the non-active period, the average data rate within a frame is less than line pixel rate. For 720p30 over DVI, this rate is 27.648 Msps.

Identifying the above rates helps determine what type of buffering is necessary, if any, within or between processing cores. If a processing core can maintain the active pixel rate indefinitely, such as a test-pattern generator core, no buffering is necessary.

- If a processing core cannot maintain the active pixel rate but can maintain the line pixel rate, a line buffer is necessary on the processing core output.
- If a processing core cannot maintain the line pixel rate but can maintain the frame pixel rate, a frame buffer is necessary on the processing core output. It is assumed that the frame buffer IP also contains line buffers to smooth access bursts.
- If a processing core cannot maintain the frame pixel rate due to insufficient throughput, no amount of buffering is sufficient to produce uninterrupted output video for the desired output standard.

Line Buffer Placement

All cores that cannot process pixels fast enough to sustain one pixel per output clock need output line-buffer(s) to avoid stalling the pipeline. Although combining line buffers at the end of a processing pipeline (by taking advantage of an output interface core with programmable line-buffer depth) might seem like an attractive option to save resources, it can also defeat the purpose of buffering.

In this example, (Figure 3-3) a hypothetical output interface needs to generate frames with 320 clock cycles per line, with 200 active pixels per line. The external memory interface retrieves pixels in 64 pixel bursts after which it is unavailable for 16 clock cycles. Core A takes 3 clock cycles to generate 2 output pixels. Core B takes three line periods to generate two active lines (no output for the 960 pixels, then 400 pixels consecutively).

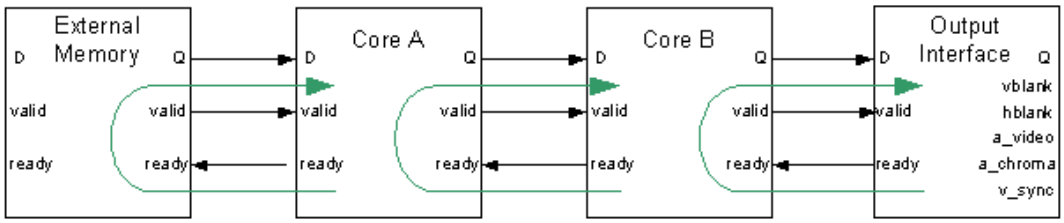

Figure 3-3: **Simple Pipeline with Internal Line Buffers**

Although all cores (external memory, Core A, Core B) have the throughput necessary to generate 200 pixels per 320 clock cycles on the average, the throughput degrades unless there are line buffers on each core output when connected as a system. For example, if the external memory provides data in 64 cycle bursts, Core A produces 42 output samples per burst or 170 pixels per line. Core A requires the whole line period to produce the active pixels, but it is forced to idle during the 4x16 cycles when the external memory is not available.

To avoid processing bubbles, all cores should be appropriately buffered on the output of the core as if the core was driving the output interface directly. Figure 3-3 illustrates the scenario when processing cores can maintain the line-pixel rate, but cores need output buffers to avoid processing bubbles. Green arrows represent subsequent cores reading from the output buffers of preceding cores.

Buffer Management

Even if sufficiently deep line buffers (FIFOs) are present on the output of processing cores, bubbles can form if buffers under-run. This can happen when a core master interface asserts its VALID output immediately when the core output FIFO is not empty. In this case, data percolates through a processing pipeline rapidly and trigger the output interface to start output timing generation, after which output pixels have to be supplied consistently. Now, if any of the cores cannot sustain the uninterrupted data rate and have to deassert its VALID output, processing bubbles form, which eventually cause a buffer under-run at the output interface core and break the output data–sync alignment.

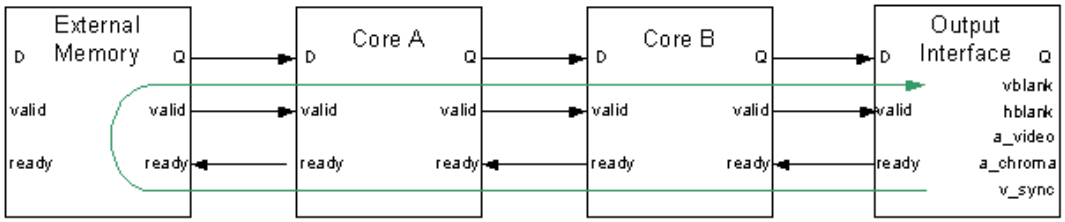

Figure 3-4: **Processing Bubble Example**

1. Core A and Core B ran out of valid samples.

Figure 3-4 presents an example scenario when processing cores A and B run out of valid samples mid-frame, so when the output interface asserts its ready output to start a new

line, samples must be retrieved from external memory and must be processed by Core A and Core B, causing significant delay, which can break the sync - data alignment at the output interface.

To avoid processing bubbles, cores should not assert the VALID signal on the output interfaces until internal FIFOs are almost full and keep VALID asserted until output FIFOs and internal pipeline stages are empty.

The READY output should be driven in a greedy fashion; asserted unless all pipeline stages are full, internal FIFOs are almost full, and the master interface READY is sampled low, as described in READY – VALID Propagation, or internal pipelines need to be flushed as described in Flushing Pipelined Cores. This behavior ensures processing efficiency and proper flushing of pipelines and processing systems at line and frame ends.

READY – VALID Propagation

For very simple IP cores, propagating VALID from master to slave and propagating READY from slave to master seems straight-forward. However, when the IP core has pipeline registers and/or FIFOs, the internal state of pipelines and FIFOs must be factored in to the READY/VALID output assignments. See Buffer Management for more information.

As stated in Input/Output Timing, the READY output on the slave interface and VALID output on the master interface must be registered. This requirement inserts a propagation delay of at least one clock cycle between the deasserted READY signal on the IP core slave interface input and the master interface READY output. The logic controlling these outputs, as well as the latching in of new pixels from the slave interface to internal FIFOs or pipeline registers, must consider the scenario when all internal buffers (pipeline registers and FIFOs) are full, the downstream slave interface just deasserted READY, but the upstream master interface sends one more pixel due to the core master interface READY signal lagging behind the slave interface.

To avoid pixel drops in the above situation, pipelined cores without internal FIFOs should contain one (or more) additional pipeline stage(s) to accept one (or more) pixel(s). These cores should keep the master interface READY output deasserted until the extra pipeline stage is processed.

To mitigate the pixel drop condition for cores with internal FIFOs the master interface READY output should be asserted unless:

- all pipeline stages are full, internal FIFOs are almost full, and the master interface READY is sampled low.
- internal pipelines need to be flushed.

Flushing Pipelined Cores

Pipelined IP cores must maintain the consistent validity of data in pipeline stages from beginning to end of video lines. For example, if horizontal FIR filtering is performed to generate valid output samples, all taps of the FIR filter delay line should only contain valid pixels. If valid data is not always present on the input (slave) interface of the filtering core, the clock-enable pins of the delay-line and the filter arithmetic should be pulsed to latch in and process only valid input samples. This implies that data in the processing pipeline of the IP core only moves forward when new, valid samples are available to process. Take for example a Color-Space Converter processing streaming video with horizontal and vertical blanking periods where no valid samples are transferred over the AXI4-Stream video interfaces for a large number of ACLK cycles. This behavior would imply that the results corresponding to the end of scan line are only available when the samples from the beginning of the next line clock them out. Similarly, the last samples from the end of a frame only become available at the beginning of the next frame. Both behaviors are prohibited because they introduce processing bubbles that break the output interface data-sync alignment.

Instead, processing pipelines must be flushed at the end of each scan-line. If samples for the next line (and next frame) are available immediately, processing cores can use these samples. If samples are not available, processing cores can flush pipelines by repeating the last valid pixel or applying a more sophisticated edge padding solution. If padding by zeros or repeated samples from the next line are needed in preparation for the next line or next frame, a processing core might deassert its READY input for as many clock cycles as it takes to empty valid data samples from the pipeline or to pad and re-initialize for a new line.

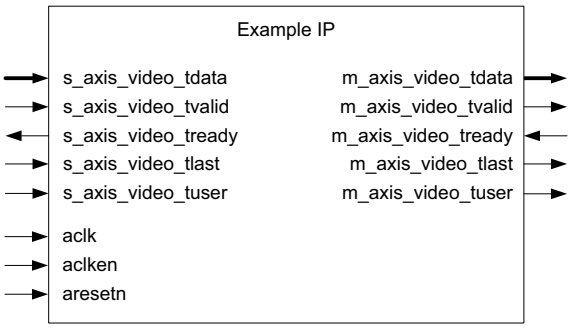

Figure 3-5: **Simple Video IP with One Slave and One Master AXI4-Stream Interfaces**

When flushing is completed and the pipeline is empty, processing cores should assert the READY output signals on the slave interfaces irrespective of the READY inputs of the master

interfaces, as seen in the READY_out and READY_in signals of Figure 3-5 and described in READY – VALID Propagation.

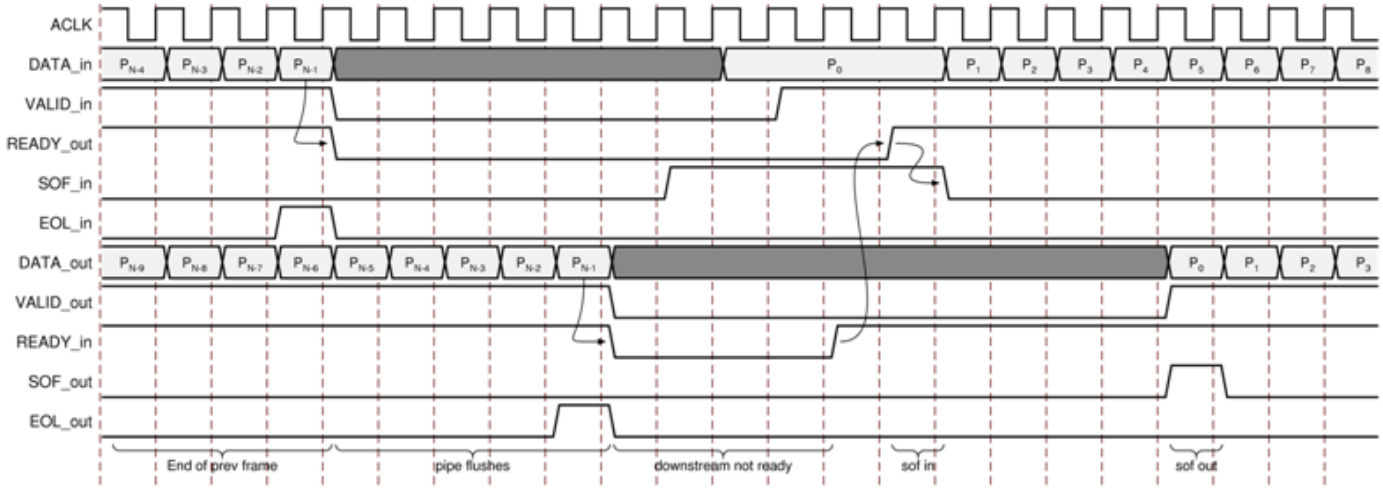

Figure 3-6: **Inefficient Flushing Growing a Processing Bubble at the End of Frame**

If the READY output signal (READY_out) assertion is delayed until the slave interface READY_in is asserted, subsequent cores would keep inserting longer breaks between lines/frames, as illustrated on Figure 3-6. In this example, the gap between frames/lines of the input stream grows because the flushing periods of subsequent cores accumulate if the IP core holds off re-asserting its READY_out output until its READY_in is asserted.

Propagating SOF and EOL Signals

Video processing IP cores either delay or re-generate the SOF and EOL pulses. No recommendations are given for which method to use when generating output SOF and EOL pulses. However, for simple pipelined IP cores without line buffers, such as a Color Space Converter, delay lines matching pipeline latency is recommended. For complex IP with line buffers, generating SOF and EOL pulses is recommended.

In accordance with AXI4-Stream Signaling Interface in Chapter 1, complex video IP can detect a discrepancy between expected number of active lines (as programmed by timing variables) and the actual number of EOL pulses received between consecutive SOF pulses.

When SOF is detected early, the output SOF signal should be generated early as well, meaning the previous frame is not padded to match programmed frame dimensions. When SOF is detected late, extra lines/pixels from the previous frame should be dropped and the output SOF signal should be generated according to the programmed values.

In accordance with End of Line Signal in Chapter 1, complex video IP can detect a discrepancy between expected number of active pixels, as programmed by timing variables, and the actual number of valid pixels received between consecutive EOL pulses.

When EOL is detected early, the output EOL signal should be generated early as well, meaning the previous frame is not padded to match programmed frame dimensions. When EOL is detected late, the output EOL signal should be according to programmed values and extra pixels from the previous line should be dropped.

Interframe Reinitialization

Some video IP cores, such as the Image Statistics and Image Characterization, take thousands of clock cycles to initialize between frames because block RAMs holding statistical data must be cleared or large sets of metadata must be written to external memory.

As a general recommendation, video IP cores should re-initialize at the end of the frame, instead of at the beginning of the frame when the SOF pulse is received.

Interrupt Subsystem

Video processing cores should provide optional interrupt pin (IRQ). Timing and core function related STATUS and ERROR flags, described in Table 3-2, should be individually selectable to generate an interrupt.

In EDK, the interrupt controller (INTC) IP can be used to integrate IRQ pins for the system processor. For Vivado tools, you might need to create a custom built priority interrupt controller to aggregate interrupt requests and identify interrupt sources.

Video IP core APIs, including registers and driver functions, should enable application software developers to identify and clear interrupt sources within the IP.

Debugging Features

The following sections recommend video IP core features which ease and accelerate system design, starting up and debug.

Version Register

Bit fields of the Version Register facilitate identification of the exact version of the hardware peripheral incorporated into a system. The core driver uses this Read-Only value to verify that the software is matched to the correct version of the hardware.

Recommended bit assignments of the version register are:

- Bits 7-0: REVISION_NUMBER
- Bits 11-8: PATCH_ID
- Bits 15-12: VERSION_REVISION
- Bits 23-16: VERSION_MINOR
- Bits 31-24: VERSION_MAJOR

Core Bypass Option

If conceptually possible, video processing IP cores should facilitate an optional straight through connection between input (AXI4-Stream slave) and output (AXI4-Stream master) by-passing any processing functionality.

Use `Flag BYPASS`, located on bit 4 of the `CONTROL` register, to turn bypassing on (1) or off. For single-clock-domain IP cores, this switch can control multiplexers in the AXI4-Stream path. For applications where the input and output AXI4-Stream interfaces are in different clock domains, the bypass multiplexers select between a clock-domain crossing FIFO implemented using distributed memory and the actual video processing core.

Built in Test-Pattern Generator

If conceptually possible, video processing IP should offer an optional built-in test-pattern generator to temporarily feed the output AXI4-Stream master interface with a predefined pattern.

Use `Flag TEST_PATTERN`, located on bit 5 of the `CONTROL` register to turn test-pattern generation on (1) or off. This switch can control multiplexers driving the AXI4-Stream master output and switch between the regular core processing output and the test-pattern generator. When enabled, a set of counters should generate 256 scan-lines of color-bars, each color bar 64 pixels wide, repetitively cycling through the colors Black, Red, Green, Yellow, Blue, Magenta, Cyan, and White until the end of each scan line. After the Color-Bars segment is processed, the remainder of the frame should be filled with a monochrome horizontal + vertical ramp.

Throughput Monitors

To debug frame-buffer bandwidth limitation issues, and if possible allow video application software to balance memory pathways, video IP cores should offer frame, line, and pixel counter registers.

The recommended name and location of these registers are `SYSDEBUG0`, `SYSDEBUG1` and `SYSDEBUG2`, as indicated in Table 3-2. The registers should initialize to 0 after reset, but the core might implement other, additional mechanisms to clear the counters.

Tool Support

Core Generator and Vivado Compatibility

For video-IP to show up in the Core Generator and Vivado repositories, CORE Generator and/or Vivado GUI files must be present in the core `/gui /xgui` directories, product guide documentation must be in PDF format in the `/doc` directory of the IP, and VHDL or Verilog simulation models, if present, must reside in the `/simulation` directory.

The IP is also recommended to include a C model (`/lib` directory), test-fixtures (`/verification`) and hardware validation projects or designs in the `/validation` directory.

For more information on designing and delivering IP using Vivado tools, see the Vivado tools documentation at:

http://www.xilinx.com/cgi-bin/docs/rdoc?v=2012.2;t=vivado+userguides

EDK Compatibility

For native Xilinx EDK support, video IP must have a peripheral descriptor file (.**mpd** file), a user interface file (.**mui** file), and driver files. The MPD file lists IP parameters and ports, and identifies clock, reset, and interrupt pins.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Please Read: Important Legal Notices