Contents

About the Video and Vision Processing Suite ................................................................. 4
  Device Family Support ......................................................................................... 4

Getting Started with the Video and Vision Processing IPs .............................................. 6
  Video and Vision Processing IP Time-out Behavior ................................................. 6
  Generating a Video and Vision Processing IP ....................................................... 6
  Simulating the Video and Vision Processing IPs ..................................................... 7

Video and Vision Processing IP Interfaces ..................................................................... 8

Video and Vision Processing IP Registers ....................................................................... 9

Protocol Converter Intel FPGA IP ................................................................................ 10
  About the Protocol Converter ............................................................................... 10
  Protocol Converter Release Information ................................................................ 10
  Protocol Converter Intel FPGA IP Parameters ....................................................... 11
  Protocol Converter Intel FPGA IP Functional Description ..................................... 12
  Protocol Converter Intel FPGA IP Interfaces ......................................................... 15
  Protocol Converter Intel FPGA IP Registers .......................................................... 16

3D LUT Intel FPGA IP ................................................................................................. 19
  About the 3D LUT Intel FPGA IP .......................................................................... 19
  3D LUT IP Features .............................................................................................. 19
  3D LUT IP Release Information ............................................................................. 19
  3D LUT IP Performance IP and Resource Information ............................................. 20
  3D LUT IP Parameters .......................................................................................... 20
  3D LUT IP Block Description ................................................................................. 22
  3D LUT IP Interfaces ............................................................................................. 23
  3D LUT IP Latency .................................................................................................. 24
  3D LUT IP Registers .............................................................................................. 24
  3D LUT IP Software API ....................................................................................... 27

Tone Mapping Operator Intel FPGA IP ............................................................................ 33
  About the Tone Mapping Operator IP .................................................................... 33
  TMO IP Features .................................................................................................... 34
  TMO IP Release Information .................................................................................. 34
  TMO IP Performance and Resource Utilization .................................................... 35
  TMO IP Parameters ............................................................................................... 36
  TMO IP Block Description ...................................................................................... 37
  TMO IP Interfaces .................................................................................................. 39
  TMO IP Latency ....................................................................................................... 43
  TMO IP Registers .................................................................................................. 43
  TMO IP Software API ......................................................................................... 45

Warp Intel FPGA IP .................................................................................................... 53
  About the Warp IP .................................................................................................. 53
  Warp IP Features .................................................................................................... 53
  Warp IP Release Information ................................................................................... 53
  Warp IP Performance and Resource Utilization ...................................................... 54
About the Video and Vision Processing Suite

The Video and Vision Processing Suite is the next-generation suite of video IPs. The suite includes three video processing IPs that transport video using a the Intel FPGA Streaming Video Protocol with a protocol convertor IP that allows interoperability with the Avalon Streaming Video standard and existing video and image IP-based systems.

The IPs support Intel Quartus Prime Pro Edition only.

These features are common to all video and vision processing IPs.

- Intel FPGA video streaming data interfaces for video I/O
- Avalon memory-mapped CPU interface for control
- Avalon memory-mapped interfaces

Related Information

- 3D LUT IP Features on page 19
- TMO IP Features on page 34
- Warp IP Features on page 53

Device Family Support

The device support is the same for all video and vision processing IPs.

Intel offers the following device support levels for Intel FPGA IP:

- **Advance support**—the IP is available for simulation and compilation for this device family. FPGA programming file (.pof) support is not available for Quartus Prime Pro Stratix 10 Edition Beta software and as such IP timing closure cannot be guaranteed. Timing models include initial engineering estimates of delays based on early post-layout information. The timing models are subject to change as silicon testing improves the correlation between the actual silicon and the timing models. You can use this IP for system architecture and resource utilization studies, simulation, pinout, system latency assessments, basic timing assessments (pipeline budgeting), and I/O transfer strategy (data-path width, burst depth, I/O standards tradeoffs).

- **Preliminary support**—Intel verifies the IP with preliminary timing models for this device family. The IP core meets all functional requirements, but might still be undergoing timing analysis for the device family. You can use it in production designs with caution.

- **Final support**—Intel verifies the IP with final timing models for this device family. The IP meets all functional and timing requirements for the device family. You can use it in production designs.
<table>
<thead>
<tr>
<th>Device Family</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Agilex™</td>
<td>Preliminary</td>
</tr>
<tr>
<td>Intel Arria® 10</td>
<td>Final</td>
</tr>
<tr>
<td>Intel Cyclone® 10</td>
<td>Final</td>
</tr>
<tr>
<td>Intel Stratix® 10</td>
<td>Final</td>
</tr>
</tbody>
</table>

**Related Information**
Timing Model, Power Model, and Device Status
Getting Started with the Video and Vision Processing IPs

Video and Vision Processing IP Time-out Behavior

All IPs in a device time out simultaneously when the most restrictive evaluation time is reached. If a design has more than one IP, the time-out behavior of the other IP may mask the time-out behavior of a specific IP.

For IP, the untethered time-out is 1 hour; the tethered time-out value is indefinite. Your design stops working after the hardware evaluation time expires. The Quartus Prime software uses Intel FPGA IP Evaluation Mode Files (.ocp) in your project directory to identify your use of the Intel FPGA IP Evaluation Mode evaluation program. After you activate the feature, do not delete these files.

When the evaluation time expires, the video and vision processing IP stops working.

Related Information
AN 320: OpenCore Plus Evaluation of Megafuntions

Generating a Video and Vision Processing IP

To include the IP in a design, generate the IP in Platform Designer.
1. Create a New Intel Quartus® Prime project
2. Open Platform Designer and create a project.
   The video and vision processing IPs are only available in Platform Designer.
3. Select DSP ➤ Vision and Video Processing ➤ <IP name>Intel FPGA IP and click Add
   The name is for both the top-level RTL module and the corresponding .ip file.
   The parameter editor for this IP appears.
4. Enter a name for your IP variant and click Create.
5. Choose your parameters.
6. Click Generate HDL.

Intel Quartus Prime generates the RTL and the files necessary to instantiate the IP in your design and synthesize it.

Related Information
- Warp IP Parameters on page 55
- TMO IP Parameters on page 36
- Protocol Converter Intel FPGA IP Parameters on page 11
- 3D LUT IP Parameters on page 20

*Other names and brands may be claimed as the property of others.
Simulating the Video and Vision Processing IPs

Related Information
Simulating Intel FPGA IP
Video and Vision Processing IP Interfaces

Table 2. Intel FPGA video stream input interface

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Direction</th>
<th>AXI4-Stream Wire Signal</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>axi4s_vid_in_tdata</td>
<td>Input</td>
<td>TDATA</td>
<td>Number of data bytes * 8</td>
</tr>
<tr>
<td>axi4s_vid_in_tlast</td>
<td>Input</td>
<td>TLAST</td>
<td>1</td>
</tr>
<tr>
<td>axi4s_vid_in_tuser</td>
<td>Input</td>
<td>TUSER</td>
<td>Number of data bytes</td>
</tr>
<tr>
<td>axi4s_vid_in_tvalid</td>
<td>Input</td>
<td>TVALID</td>
<td>1</td>
</tr>
<tr>
<td>axi4s_vid_in_tready</td>
<td>Output</td>
<td>TREADY</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3. Intel FPGA video stream output interface

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Direction</th>
<th>AXI4-Stream Wire Signal</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>axi4s_vid_out_tdata</td>
<td>Output</td>
<td>TDATA</td>
<td>Number of data bytes * 8</td>
</tr>
<tr>
<td>axi4s_vid_out_tlast</td>
<td>Output</td>
<td>TLAST</td>
<td>1</td>
</tr>
<tr>
<td>axi4s_vid_out_tuser</td>
<td>Output</td>
<td>TUSER</td>
<td>Number of data bytes</td>
</tr>
<tr>
<td>axi4s_vid_out_tvalid</td>
<td>Output</td>
<td>TVALID</td>
<td>1</td>
</tr>
<tr>
<td>axi4s_vid_out_tready</td>
<td>Input</td>
<td>TREADY</td>
<td>1</td>
</tr>
</tbody>
</table>

Number of data bytes = \( \text{max}(2, \text{ceil}(\text{bits per sample} / 8) \times \text{pixels in parallel}) \)

Related Information

- Warp IP Interfaces on page 59
- TMO IP Interfaces on page 39
- 3D LUT IP Interfaces on page 23
- Protocol Converter Intel FPGA IP Interfaces on page 15
- Intel FPGA Streaming Video Protocol Specification
The IPs have compatible register maps. The register maps contain parameterization information.

In general, video and vision processing IP register maps have two distinctives areas:

- A common area, which contains parameterization information. You can read to and write from components to determine the configuration, which allows portability of software and binaries between different video and vision processing platforms.
- An IP-specific video and vision processing IP area, which contains functional configuration information for the specific IP.

Control interfaces use the Avalon memory-mapped interfaces. AXI4-Stream protocols are natively supported in Platform Design and can be automatically adapted to and from Avalon memory-mapped interfaces. Memory interfaces also use Avalon memory-mapped interfaces. You may also adapt them to AXI4-Lite as required in Platform Designer.

### Table 4. Register Map for Video and Vision Processing IPs

<table>
<thead>
<tr>
<th>Register</th>
<th>Word Address</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>VID, PID</td>
<td>0x0</td>
<td>RO</td>
</tr>
<tr>
<td>Version number</td>
<td>0x1</td>
<td>RO</td>
</tr>
<tr>
<td>IP parameterization registers</td>
<td>0x2:0x3F</td>
<td>RO</td>
</tr>
<tr>
<td>IP control registers</td>
<td>0x40:0xFFFF</td>
<td>RW</td>
</tr>
</tbody>
</table>

#### Related Information
- 3D LUT IP Registers on page 24
- Protocol Converter Intel FPGA IP Registers on page 16
- TMO IP Registers on page 43
- Warp IP Registers on page 67
Protocol Converter Intel FPGA IP

About the Protocol Converter

The IP converts from Avalon Streaming Video to Intel FPGA Streaming Video and vice versa. The IP enables you to create systems with IPs from both Intel video IP libraries. The Video and Vision Processing IPs use the AXI4-Stream based Intel FPGA Streaming Video to receive and transmit streaming video data at their interfaces. The Video and Image Processing IPs use the Avalon Streaming Video protocol. The Video and Vision Processing IPs do not directly connect to IP from the Video and Image Processing library.

Figure 1. Protocol Converter

Protocol Converter Release Information


The Intel FPGA IP version (X.Y.Z) number can change with each Intel Quartus Prime software version. A change in:

- X indicates a major revision of the IP. If you update the Intel Quartus Prime software, you must regenerate the IP.
- Y indicates the IP includes new features. Regenerate your IP to include these new features.
- Z indicates the IP includes minor changes. Regenerate your IP to include these changes.
### Protocol Converter IP Release Information

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>21.2</td>
</tr>
<tr>
<td>Release date</td>
<td>June 2021</td>
</tr>
<tr>
<td>Ordering code</td>
<td>-</td>
</tr>
</tbody>
</table>

### Protocol Converter Intel FPGA IP Parameters

The IP offers compile-time parameters

### Table 6. Parameters

The table lists the parameters that are available to configure the IP in Platform Designer.

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits per color sample</td>
<td>8 to 16</td>
<td>Number of bits that represent each color sample</td>
</tr>
<tr>
<td>Number of color planes</td>
<td>1 to 4</td>
<td>Number of colors per pixel</td>
</tr>
<tr>
<td>Number of pixels in parallel</td>
<td>1, 2, 4 or 8</td>
<td>Number of pixels transmitted per clock cycle</td>
</tr>
<tr>
<td>YCbCr 444 colour swap</td>
<td>On or off</td>
<td>Turn on to automatically correct for the color plane ordering differences between Avalon Streaming Video and Intel FPGA Streaming Video when transmitting YCbCr data with 4:4:4 chroma</td>
</tr>
<tr>
<td>Memory mapped control interface</td>
<td>On or off</td>
<td>Turn on to allow the Avalon memory-mapped control agent interface to update settings at runtime</td>
</tr>
<tr>
<td>Separate clock for control interface</td>
<td>On or off</td>
<td>Turn on for a separate clock domain for the Avalon memory-mapped control agent interface</td>
</tr>
<tr>
<td>Debug features</td>
<td>On or off</td>
<td>Turn on for the debugging features of the Avalon memory-mapped control agent interface</td>
</tr>
<tr>
<td>Pipeline ready signals</td>
<td>On or off</td>
<td>Turn on to add extra pipeline registers to the AXI4-Stream or Avalon Streaming ready signals. Turning on this option may make it easier to close timing for the protocol converter and achieve a higher operation clock frequency, but may contribute to additional ALM usage.</td>
</tr>
<tr>
<td>Input protocol variant</td>
<td>Avalon Streaming Video or Intel FPGA Streaming Video Lite</td>
<td>Select the protocol for the input interface</td>
</tr>
<tr>
<td>Output protocol variant</td>
<td>Avalon Streaming Video or Intel FPGA Streaming Video</td>
<td>Select the protocol for the output interface</td>
</tr>
<tr>
<td>How Avalon-ST Video user packets are handled</td>
<td>No user packets expected at the input or Discard all user packets received</td>
<td>Select how user packets are handled if the input protocol is Avalon Streaming Video. If you do not expect the input stream to contain any user packets, you can select No user packets expected at the input and save the ALM resources required to discard these packets</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video color space</td>
<td>RGB or YCbCr</td>
<td>If the input protocol is Avalon Streaming Video and you do not turn on Avalon memory-mapped control agent interface, you must specify the color space of the incoming video.</td>
</tr>
<tr>
<td>Video chroma sampling</td>
<td>444, 422 or 420</td>
<td>If the input protocol is Avalon Streaming Video and you turn off Avalon memory-mapped control agent interface, you must specify the chroma sampling of the incoming video.</td>
</tr>
<tr>
<td>Enable low latency</td>
<td>On or off</td>
<td>If the input protocol is Intel FPGA Streaming Video this parameter determines the behaviour of the Protocol Converter at the end of each video frame.</td>
</tr>
</tbody>
</table>

**Protocol Converter Intel FPGA IP Functional Description**

**Protocol Converter - Avalon Streaming Video to Intel FPGA Streaming Video**

The IP converts the protocol in steps:

1. Changes the Avalon Streaming ready latency from 1 to 0.
Avalon Streaming Video specifies a ready latency of 1. The IP converts the ready latency to 0 to match the ready-valid handshake mechanism specified for AXI4-Stream.

2. Removes all non-video data packets from the stream.
   Avalon Streaming Video specifies a mechanism to assign a type identifier (a number between 0 and 15) to each packet in the stream. Type-0 packets are frames of pixel data and all other packet types indicate non-video data. Packets of type-15 (referred to as control packets) contain metadata to specify the width, height, and interlacing properties of subsequent type-0 video packets. Intel FPGA Streaming Video does not allow for nonvideo packets in the stream. The IP discards all packets with a type that is greater than 0. The IP does not propagate type-15 control packets that specify the properties of the video. However, it parses them during the discard process to extract the expected width of the video frames that follow. The IP uses this information in the next step of the conversion.

3. Splits frame packets into line packets.
   Avalon Streaming Video specifies that each video packet contains one cycle of header data followed by all of the pixels required for an interlaced or progressive frame of video. The header data specifies the packet type. Intel FPGA Streaming Video requires that each packet is one line of video data (with no header information). The IP strips out incoming Avalon Streaming Video frame packets and then splits them into multiple packets. Each packet contains a single video line. The IP extracts the expected width of the video frame from the discarded control packet that precedes the frame. The IP uses this value to determine where the incoming frame packet should be split to make each output line packet. The IP replaces the Avalon Streaming startofpacket and endofpacket signals by the AXI4-Stream tlast signal and creates the tuser signal. The IP reformats the incoming Avalon Streaming data to create byte aligned AXI4-Stream tdata.

Protocol Converter - Intel FPGA Streaming Video to Avalon Streaming Video

The IP converts the protocol in 3 steps.

1. Combines line packets into a single frame packet.
   Intel FPGA Streaming Video specifies that you transmit video data with one video line per packet. Avalon Streaming Video requires you transmit all the pixels in a frame in a single packet. The incoming line packets must merge to form one frame packet. If you transmit a single pixel per clock cycle, bit 0 of the incoming tuser signal marks the first pixel of each frame. The IP concatenates packets until bit 0 of tuser is asserted. If the number of pixels per line is not a multiple of the pixels per clock, the extra pixels in the final clock cycle of data are effectively empty and you must ignore them. You must specify the width of the incoming video frame via the register map. The IP uses this width information to determine which (if any) pixels it should ignore at the end of each line when concatenating the packets.

2. Adds the frame packet header and the control packet
   Avalon Streaming Video requires that each frame packet begins with a one cycle header specifying a packet type of 0. The IP adds this header to the frame packet created previously. Avalon Streaming Video also recommends that each frame packet is preceded by a control packet (of type-15) that specifies the width, height, and interlacing scheme of the following frame. You supply the width and height via the register map, as the initial value for interlacing specifier. The IP uses these values to control Avalon Streaming control packets that it adds to the
If you select an interlacing specifier for progressive video, the IP uses this value for all control packets. If the interlacing specifier identifies an interlaced scheme, the IP toggles the f0/f1 bit automatically in the outgoing control packets.

3. Converts Avalon Streaming ready latency 0 to 1

The IP replaces the AXI4-Stream \texttt{tlast} signal with the Avalon Streaming \texttt{startofpacket} and \texttt{endofpacket} signals. The IP creates the \texttt{empty} signal (if the number of pixels per clock cycle is greater than 1). The IP reformats the AXI4-Stream byte aligned \texttt{tdata} to non-byte aligned Avalon Streaming data. The interface is now compliant to the Avalon Streaming protocol, but with a ready latency of 0. Avalon Streaming Video requires that the ready latency is 1, so the IP converts the ready latency from 0 to 1.

**Pixel data format**

The pixel data format for Avalon Streaming Video and Intel FPGA Streaming Video is almost identical. Intel FPGA Streaming Video requires that the width of each pixel is rounded up to the next whole number of bytes. Any extra bits required can be filled with zeros, ones, or any random data. Avalon Streaming Video has no such requirement and uses only the required bits for each pixel. The Protocol Converter adds the required extra bits when converting from Avalon Streaming Video to Intel FPGA Streaming Video. It removes them when converting from Intel FPGA Streaming Video to Avalon Streaming Video.

Avalon Streaming Video and Intel FPGA Streaming Video both specify how the color planes in each pixel should be arranged for RGB and YCbCr formatted data. For YCbCr data, the protocols specify the color plane ordering for 4:4:4, 4:2:2 and 4:2:0 chroma sampling. The color plane ordering is almost identical between the two protocols, apart from the Y and Cr planes are swapped in the case of YCbCr 4:4:4. The Protocol Converter IP can implement the swap, but you must specify the color space and chroma sampling for each frame. You can specify either via the parameters or the register map accessed through an Avalon memory-mapped agent interface.

You can turn on or turn off the Avalon memory-mapped agent interface via a parameter. If the Avalon memory-mapped agent interface is turned on, specify the color space and chroma sampling at runtime via the register map. If the Avalon memory-mapped agent interface is not turned on, specify the color space and chroma sampling in the \texttt{Video color space} and \texttt{Video chroma sampling} parameters respectively.

If the Protocol Convert IP converts from Intel FPGA Streaming Video to Avalon Streaming Video, turn on the Avalon memory-mapped agent interface and do not use the parameters. If the Protocol Convert IP converts from Avalon Streaming Video to Intel FPGA Streaming Video, the Avalon memory-mapped agent interface is optional. If you know the color space and chroma sampling are fixed for the system, you can opt to turn off the agent interface and specify the color space and chroma sampling via the parameters. If the color space and chroma sampling may vary at runtime, turn on the agent interface and specify the values in the register map.

For conversions both ways, the IP gates the color plane swap for YCbCr 4:4:4 formatted data by the \texttt{YCbCr 444 colour swap} parameter. You must turn on this option for the IP to apply the color plane swap.
Control packet width for 4:2:0 chroma sampled video

When you transmit 4:2:0 chroma sampled data across an Avalon Streaming Video interface, the frame width that the control packet reports is always half the actual frame width. Each section of the bus contains two luma samples that Avalon Streaming Video regards as a pixel. The value that the control packet reports is relative to a single pixel in each of these sections of the bus. If you specify the chroma sampling of the incoming video, the control panel reports the double-packing of luma samples in 4:2:0 chroma sampling.

End of frame detection

Intel FPGA Streaming Video does not explicitly mark the end of each frame. For Avalon Streaming Video, the endofpacket signal marks the end of each video frame that is asserted on the final pixel of the video data packet.

In Avalon Streaming Video, you cannot transmit the final pixel of each frame until you are certain that it is the final pixel, otherwise you risk driving the endofpacket signal incorrectly.

For Intel FPGA Streaming Video the end of each frame is inferred by receiving the start-of-frame marker for the next frame, which is explicitly indicated in the protocol. You might see potential latency issues when converting from Intel FPGA Streaming Video to Avalon Streaming Video. The IP cannot transmit the final pixel of each frame at the output until it receives the first pixel of the next frame at the input.

If the video application has no significant blanking (delay) between the last pixel of one frame and the first pixel of the next frame, the IP gives little or no delay in sending out the final pixel of each frame. If the application does have significant blanking, the delay to transmit the final pixel may be too long. The Protocol Converter IP includes an option to remove this delay.

If you turn on Low latency mode, the Protocol Converter IP transmits the Avalon Streaming Video frame endofpacket according to the number of lines it expects in each frame, as you specify in the register map. The Intel FPGA Streaming Video protocol transmits each line of video data as a packet, so the IP terminates the output frame at the end of the input packet for the specified number of lines. If the IP receives any additional lines, the IP discards them and does not transmit them at the output.

Protocol Converter Intel FPGA IP Interfaces

Table 7. Protocol Converter Interfaces
The table lists the interfaces used by the IP. The IP does not enable all interfaces in all parameterizations. The table shows the parameter settings for which the IP enables each interface.

<table>
<thead>
<tr>
<th>Interface name</th>
<th>Clock domain</th>
<th>Signals</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main_clock</td>
<td>n/a</td>
<td>main_clock_clk</td>
<td>Main clock used to drive the IP logic and streaming interfaces</td>
</tr>
<tr>
<td>main_reset</td>
<td>main_clock</td>
<td>main_reset_reset</td>
<td>Main reset that initializes the IP logic and streaming interfaces</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Interface name</th>
<th>Clock domain</th>
<th>Signals</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent_clock</td>
<td>n/a</td>
<td>agent_clock_clk</td>
<td>Only enabled if you turn on <strong>Separate clock for control interface</strong>. The clock that drives the <strong>av_mm_control_agent</strong> interface.</td>
</tr>
<tr>
<td>agent_reset</td>
<td>agent_clock</td>
<td>agent_reset_reset</td>
<td>Only enabled if you turn on <strong>Separate clock for control interface</strong>. The reset that initializes the <strong>av_mm_control_agent</strong> interface.</td>
</tr>
<tr>
<td>av_mm_control_agent</td>
<td>main_clock</td>
<td>av_mm_control_agent_read</td>
<td>Only enabled if you turn on <strong>Memory mapped control interface</strong>. The Avalon memory-mapped agent interface that you use to edit settings in the register map at runtime. Clocks on the <strong>agent_clock</strong> domain if you turn on <strong>Separate clock for control interface</strong>, otherwise clocks on the <strong>main_clock</strong> domain.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_mm_control_agent_write</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_mm_control_agent_address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_mm_control_agent_byteenable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_mm_control_agent_waitrequest</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_mm_control_agent_readdatavalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_mm_control_agent_writedata</td>
<td></td>
</tr>
<tr>
<td>axi4s_vid_in</td>
<td>main_clock</td>
<td>axi4s_vid_in_tvalid</td>
<td>Only enabled if you select <strong>Intel FPGA Streaming Video</strong> for the <strong>Input protocol variant</strong>. Intel FPGA Streaming Video compliant streaming sink.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>axi4s_vid_in_tready</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>axi4s_vid_in_tlast</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>axi4s_vid_in_tuser</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>axi4s_vid_in_tdata</td>
<td></td>
</tr>
<tr>
<td>av_st_vid_in</td>
<td>main_clock</td>
<td>av_st_vid_in_valid</td>
<td>Only enabled if the <strong>Input protocol variant</strong> parameter is set to Avalon Streaming Video. Avalon Streaming Video compliant streaming sink.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_in_ready</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_in_startofpacket</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_in_endofpacket</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_in_data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_in_empty</td>
<td></td>
</tr>
<tr>
<td>axi4s_vid_out</td>
<td>main_clock</td>
<td>axi4s_vid_out_tvalid</td>
<td>Only enabled if the <strong>Output protocol variant</strong> parameter is set to Intel FPGA Streaming Video Lite. Intel FPGA Streaming Video compliant streaming source.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>axi4s_vid_out_tready</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>axi4s_vid_out_tlast</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>axi4s_vid_out_tuser</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>axi4s_vid_out_tdata</td>
<td></td>
</tr>
<tr>
<td>av_st_vid_out</td>
<td>main_clock</td>
<td>av_st_vid_out_valid</td>
<td>Only enabled if you select <strong>Avalon Streaming Video</strong> in the <strong>Input protocol variant</strong>. Avalon Streaming Video compliant streaming source.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_out_ready</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_out_startofpacket</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_out_endofpacket</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_out_data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>av_st_vid_out_empty</td>
<td></td>
</tr>
</tbody>
</table>

**Related Information**

Video and Vision Processing IP Interfaces on page 8

**Protocol Converter Intel FPGA IP Registers**

Read and write access to the register map is via the Avalon memory-mapped compliant **av_mm_control_agent** interface. Turn on **Enable memory mapped control interface**, for access to this interface and access to the register map.
The `av_mm_control_agent` interface uses word addressing to access each register. The value the IP applies to the `av_mm_control_agent_address` signal should be the word address of the register to read or write to. Intel shows byte address of each register because Avalon memory-mapped host interfaces typically use byte addressing. Platform Designer applies any byte address to word address conversion if required.

Table 8. Protocol Converter Registers

<table>
<thead>
<tr>
<th>Register name</th>
<th>Byte Address</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product ID</td>
<td>0x0</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>Version number</td>
<td>0x4</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>Conversion mode</td>
<td>0x8</td>
<td>RO</td>
<td>A read to this register returns a value that specifies the input and output protocols for this instance of the Protocol Converter IP. A return value of 0 indicates that it converts from Avalon Streaming Video to Intel FPGA Streaming Video. A return value of 1 indicates that it converts from Intel FPGA Streaming Video to Avalon Streaming Video.</td>
</tr>
<tr>
<td>Enable debug</td>
<td>0xC</td>
<td>RO</td>
<td>A read to this register returns the value you select for the Enable debug parameter in this instance of the Protocol converter. The host software can read this value to determine which registers you can read.</td>
</tr>
<tr>
<td>Field width</td>
<td>0x120</td>
<td>WO</td>
<td>If you select Intel FPGA Streaming Video Lite for Input protocol, this value specifies the frame width (in pixels) that the IP uses to create the output Avalon Streaming Video control packet. For 4:2:0 chroma sampling, this width represents the total number of luma samples per line, and you do not need to divide the image width in half.</td>
</tr>
<tr>
<td>Field height</td>
<td>0x124</td>
<td>WO</td>
<td>If you select Intel FPGA Streaming Video Lite for Input protocol, this value specifies the frame height (in lines) that the IP uses to create the output Avalon Streaming Video control packet.</td>
</tr>
<tr>
<td>Field interlace</td>
<td>0x128</td>
<td>WO</td>
<td>If you select Intel FPGA Streaming Video Lite for Input protocol, this value specifies the frame interlace nibble that the IP uses to create the output Avalon Streaming Video control packet. Specify the value according to the 4-bit interlace nibble codes in the Avalon Streaming Video protocol. If 4-bit code specifies an interlaced, you should specify the interlace code that the IP should use for the first output frame. The f0/f1 indicator bit toggles automatically for subsequent frames.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x12C – 0x11C</td>
<td>RO</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>Color space</td>
<td>0x130</td>
<td>WO</td>
<td>The value you write to this register specifies the color space of the incoming video. Write 0 for RGB, 1 for YCbCr and 2 for monochrome.</td>
</tr>
<tr>
<td>Chroma sampling</td>
<td>0x134</td>
<td>WO</td>
<td>The value you write to this register specifies the chroma sampling of the incoming video. 0 for 420, 2 for 422, and 3 for 444.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x138</td>
<td>RO</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x13C</td>
<td>RO</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>Status</td>
<td>0x140</td>
<td>RO</td>
<td>The value you read from this register indicates the processing status of the IP.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x144</td>
<td>RO</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>

continued...
### Register names

<table>
<thead>
<tr>
<th>Register name</th>
<th>Byte Address</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIP control width</td>
<td>0x148</td>
<td>RO</td>
<td>If you select <strong>Avalon Streaming Video</strong>, for <strong>Input protocol variant</strong>, a read to this register returns the frame width specified in the most recently received control packet. The width reported is a literal decode of the information in the control packet. If the data the IP processes is 4:2:0 chroma sampled, the width reported is half the actual frame or frame width.</td>
</tr>
<tr>
<td>VIP control height</td>
<td>0x14C</td>
<td>RO</td>
<td>If you select <strong>Avalon Streaming Video</strong> for <strong>Input protocol variant</strong>, a read to this register returns the frame height specified in the most recently received control packet.</td>
</tr>
<tr>
<td>VIP control interlaced</td>
<td>0x150</td>
<td>RO</td>
<td>If you select <strong>Avalon Streaming Video</strong> for <strong>Input protocol variant</strong>, a read to this register returns the interlace nibble specified in the most recently received control packet.</td>
</tr>
<tr>
<td>Control</td>
<td>0x154</td>
<td>WO</td>
<td>Writes to this register instruct the IP to start processing video frames, or to stop processing at the next frame boundary. Write a 1 to bit[0] of this register to start the IP. Write a 0 to bit[0] to stop at the next frame boundary. If the IP is already at a frame boundary or is between frames when the write to stop occurs, it stops immediately and does not begin the next frame. The value of this register resets to 0, so if the av_mm_control_agent interface is turned on, the IP resets into the stopped state and you must write a 1 to bit[0] to begin processing.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x158</td>
<td>WO</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x15C</td>
<td>WO</td>
<td>Reserved for future use</td>
</tr>
</tbody>
</table>

### Table 9. Status register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>This bit indicates the IP is currently processing a frame. A value of 1 indicates that the IP is busy processing, a value of 0 indicates that it is idle. When converting from Avalon Streaming Video to Intel FPGA Streaming Video, bit 0 is set to 1 at the start of the first packet belonging to each video frame. This packet can be a user packet, a control packet, or the frame packet. Bit 0 is then set back to 0 when the final cycle of data in the video frame packet is received. When converting from Intel FPGA Streaming Video to Avalon Streaming Video the interpretation of bit 0 depends on if you turn on <strong>Enable low latency mode</strong>, the IP sets bit 0 to 1 when it receives the first pixel of the frame is received and sets to 0 when it receives the number of lines specified in register map address 73 (0x124). The IP holds bit 0 at 0 while it flushes any additional lines. If you do not turn on <strong>Enable low latency mode</strong>, the IP sets bit 0 to 1 at the start of the first frame received and it remains high until you reset the IP.</td>
</tr>
<tr>
<td>1</td>
<td>This bit indicates if the IP has fully processed at least one frame since the last reset. A 1 indicates that the IP has processed at least one, a 0 indicates that the IP has processed no frames.</td>
</tr>
<tr>
<td>2</td>
<td>This bit indicates if the last frame the IP receives has the expected number of pixels. A 0 indicates that the frame matched the width and height specified in the Avalon Streaming Video control packet or register map settings. A 1 indicates that the frame had too many or too few pixels according to these settings.</td>
</tr>
<tr>
<td>31:3</td>
<td>Unused.</td>
</tr>
</tbody>
</table>
3D LUT Intel FPGA IP

About the 3D LUT Intel FPGA IP

The IP maps a video stream’s color space to another using interpolated values from a lookup table where you can build-in or dynamically program the values.

You can configure the required number of bits per symbols, symbols per pixel, pixels in parallel. Also you can configure the LUT size and built-in initialization from a file.

Typical applications include:
- Color space conversion
- Chroma keying
- Dynamic range conversion
- Artistic effects (e.g. sepia, hue rotation, color volume adjustment)

3D LUT IP Features

- Avalon memory-mapped CPU interface for control and LUT upload
- LUT sizes of $17^3$, $33^3$, and $65^3$
- Tetrahedral interpolation
- Range of 8 to 16 bits per color
- Independent parameters for input, output, and LUT bits per color
- Up to 8 pixels in parallel
- Dynamic update of LUT via CPU interface
- Double buffered LUT option allows for seamless run-time switching
- Subframe fixed latency
- Very small ALM footprint (~ 2K ALMs @ 2 pixels in parallel)

3D LUT IP Release Information

The Intel FPGA IP version (X.Y.Z) number can change with each Intel Quartus Prime software version. A change in:

- X indicates a major revision of the IP. If you update the Intel Quartus Prime software, you must regenerate the IP.
- Y indicates the IP includes new features. Regenerate your IP to include these new features.
- Z indicates the IP includes minor changes. Regenerate your IP to include these changes.

### Table 10. 3D LUT IP Release Information

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>21.2</td>
</tr>
<tr>
<td>Release date</td>
<td>June 2021</td>
</tr>
<tr>
<td>Ordering code</td>
<td>IP-OM-3D-LUT</td>
</tr>
</tbody>
</table>

### 3D LUT IP Performance IP and Resource Information

Intel provides resource and utilization data for guidance.

### Table 11. 3D LUT Performance and Resource Usage

The numbers are for a design targeting an Intel Arria 10 device with a design $f_{\text{MAX}}$ of 300 MHz.

<table>
<thead>
<tr>
<th>Pixel in parallel</th>
<th>Bits per color sample</th>
<th>LUT Size</th>
<th>Double buffer</th>
<th>ALMs</th>
<th>Memory (M20K)</th>
<th>DSP Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>17</td>
<td>no</td>
<td>810</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>17</td>
<td>yes</td>
<td>937</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>33</td>
<td>no</td>
<td>1,640</td>
<td>33</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>33</td>
<td>yes</td>
<td>1,681</td>
<td>49</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>65</td>
<td>no</td>
<td>2,575</td>
<td>830</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>65</td>
<td>yes</td>
<td>4,035</td>
<td>1,622</td>
<td>12</td>
</tr>
</tbody>
</table>

### 3D LUT IP Parameters

The 3D LUT IP offers compile-time parameters.

### Table 12. 3D LUT IP compile-time parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video data format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of pixels in parallel</td>
<td>1 to 8</td>
<td>Number of pixels transmitted every clock cycle</td>
</tr>
<tr>
<td>Input bits per color sample</td>
<td>8 to 16</td>
<td>Number of bits per color sample at the input</td>
</tr>
<tr>
<td>Output bits per color sample</td>
<td>8 to 16</td>
<td>Number of bits per color sample at the output</td>
</tr>
<tr>
<td>Control settings</td>
<td></td>
<td>continued...</td>
</tr>
<tr>
<td>Name</td>
<td>Values</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Separate clock for control interface</td>
<td>On or off</td>
<td>Turn on to run the run-time control interface on a different clock domain</td>
</tr>
<tr>
<td>LUT read interface</td>
<td>On or off</td>
<td>Allows you to read LUT contents via the CPU interface</td>
</tr>
<tr>
<td>LUT settings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>17, 33, 65</td>
<td>Size of each LUT dimension</td>
</tr>
<tr>
<td>Bits per color</td>
<td>8 to 16</td>
<td>The number of bits per color in the LUT (LUT_DEPTH)</td>
</tr>
<tr>
<td>Output alpha channel</td>
<td>On or off</td>
<td>Turn on to add alpha channel to LUT (RGBA)</td>
</tr>
<tr>
<td>Double buffered</td>
<td>On or off</td>
<td>Double the memory for seamless LUT programming and switching</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On to instantiate the second buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Off for single buffer only</td>
</tr>
<tr>
<td>Buffer 0 and Buffer 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialize from file</td>
<td>On or off</td>
<td>• On to initialize the LUT from a file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Off to uninitialized LUT on reset</td>
</tr>
<tr>
<td>Init file</td>
<td>user file</td>
<td>Optional initialization file.</td>
</tr>
<tr>
<td>Init file type</td>
<td>normalized, integer</td>
<td>Type of coefficients in the initialization file:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Normalized floating or fixed point numbers between 0.0 and 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Integer numbers between 0 and $2^{LUT_\text{DEPTH}-1}$</td>
</tr>
</tbody>
</table>

**Figure 3. 3D LUT GUI**

![3D LUT GUI](image)
LUT Initialization File

You can initialize each buffer of the LUT from reset by providing a compatible 3D LUT file to **Init file** in the GUI. The IP generation process converts the LUT file into RAM initialization .hex files that get built into the firmware during compilation. The script can read .cube format files, or any 3D LUT files that follow these conventions:

- RGB component order (must match the video stream’s order)
- Components change first from left to right, i.e. R first, G second, B third
- If you turn on alpha, you append the alpha value as a fourth component (RGBA)
- The data type must match the IP GUI parameter and may either be:
  - normalized fixed- or floating-point numbers between 0.0 to 1.0
  - integers between 0 and $2^{LUT\_DEPTH-1}$ (e.g. 10-bit: 0 to 1023)
- The data type must be the same for the whole file
- Lines starting with # or any letter are ignored

3D LUT IP Block Description

The 3D LUT IP accepts RGB-format video input from its Intel FPGA video streaming interface. It uses the most significant bits (MSBs) of the 3 color component inputs to retrieve data values from the contents of the LUT and the least significant bits (LSBs) to interpolate the final output value. An Avalon Memory-Mapped compatible CPU interface handles the run-time control and LUT programming.

![3D LUT IP block diagram](image)

The address decoder converts the MSBs of the three input color components into read addresses for the LUT. If you turn on **Double buffered**, the IP adds a page offset to the address when selecting the second buffer via the CPU interface. Page-flip double buffering allows for instantaneous switching between LUTs.
The LUT RAM instantiates the on-chip memory containing the LUT. The 3D LUT cube vertices are divided across eight sub-RAMs in order to output the target sub-cube vertices in parallel. Enabling the second buffer doubles the memory depth of the LUT. Both buffers contents are programmable via the CPU interface and can also be pre-initialized in the firmware via the 3D LUT IP GUI.

The tetrahedral Interpolator uses a DSP efficient method to interpolate four of the LUT subcube vertices using the input LSBs. Part of the input MSBs determines which of the six tetrahedra in the target sub-cube contains the pixel.

Turn on or off the LUT processing switch between the interpolated output and the bypass output with the control register in the run-time register map.

Consider these points when integrating in a streaming video pipeline:

- The IP controls buffer selection and output enable and only updates them at the start of each new frame.
- The internal pipeline forwards control signals and is unaffected by changes to video resolution.

**Figure 5. 3D LUT color transform examples**

From top left: original, saturation, brightness increase, colorize (purple), colorize (green), desaturation

**3D LUT IP Interfaces**

The IP has three functional interfaces:

- Intel FPGA video stream input interface
- Intel FPGA video stream output interface
- Avalon Memory-Mapped compatible CPU interface

The 3D LUT IP control interface uses Avalon Memory-Mapped protocol to access control and RAM interface registers.

**Clocks**

The 3D LUT IP has two clock domains, each with a corresponding reset signal.
Table 13. Clock domains

<table>
<thead>
<tr>
<th>Clock name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_clock</td>
<td>CPU interface clock domain</td>
</tr>
<tr>
<td>vid_clock</td>
<td>Video processing clock domain</td>
</tr>
</tbody>
</table>

The CPU interface uses little bandwidth and therefore does not impose a minimum clock frequency. The video clock frequency depends on the video resolution, frame rate, and the 3D LUT IP’s number of pixels in parallel. For example, a 300 MHz clock at 2 pixels in parallel supports active video resolutions up to 4096x2160 at 60 Hz.

All RTL-based blocks that transfer or receive data from a different clock domain include clock domain crossing (CDC) circuits for both, single bit and data bus signal cases. The CDC safely allow exchange of data between the two asynchronous clock domains. This principle applies to the control signals from the CPU interface to the main video datapath. The 3D LUT IP includes an .sdc file to constrain this CDC.

Table 14. Resets associated to clock domains

<table>
<thead>
<tr>
<th>Reset name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_reset</td>
<td>CPU interface clock domain reset.</td>
</tr>
<tr>
<td>vid_reset</td>
<td>Video processing clock domain reset.</td>
</tr>
</tbody>
</table>

3D LUT IP Latency

The latency information can predict the approximate latency between the input and the output of your video processing pipeline.

Table 15. 3D LUT IP operation mode latency

<table>
<thead>
<tr>
<th>Device</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Arria 10</td>
<td>21</td>
</tr>
<tr>
<td>Intel Cyclone 10 GX</td>
<td>21</td>
</tr>
</tbody>
</table>

3D LUT IP Registers

The 3D LUT IP allows run-time control and LUT programming via the CPU interface. The register map allows access to the:
• Build parameters that expose compile-time parameters.
• Control interface that enables switching between bypass and operational modes. Also toggle buffers when you turn onDouble buffered for the LUT.
• RAM interface that allows programming of the LUT’s 8 sub-RAMs during runtime and reading their contents if you turn on LUT read interface.

Table

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Byte Address Offset</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vid_pid</td>
<td>0x000</td>
<td>RO</td>
<td>Vendor ID and Product ID</td>
</tr>
<tr>
<td>version_number</td>
<td>0x004</td>
<td>RO</td>
<td>Version number</td>
</tr>
<tr>
<td></td>
<td>0x008</td>
<td>RO</td>
<td>Reserved</td>
</tr>
<tr>
<td>pixels_in_parallel</td>
<td>0x00C</td>
<td>RO</td>
<td>Video data format Number of pixels in parallel parameter</td>
</tr>
<tr>
<td>input_bps</td>
<td>0x010</td>
<td>RO</td>
<td>Video data format Input bits per color sample parameter</td>
</tr>
<tr>
<td>output_bps</td>
<td>0x014</td>
<td>RO</td>
<td>Video data format Output bits per color sample parameter</td>
</tr>
<tr>
<td>lut_alpha</td>
<td>0x018</td>
<td>RO</td>
<td>LUT settings Output alpha channel parameter</td>
</tr>
<tr>
<td>lut_depth</td>
<td>0x01C</td>
<td>RO</td>
<td>LUT settings Bits per color parameter</td>
</tr>
<tr>
<td>lut_dimension</td>
<td>0x020</td>
<td>RO</td>
<td>LUT settings Size parameter</td>
</tr>
<tr>
<td>lut_double_buffered</td>
<td>0x024</td>
<td>RO</td>
<td>LUT settings Double buffered parameter</td>
</tr>
<tr>
<td>lut_cpu_readable</td>
<td>0x028</td>
<td>RO</td>
<td>Control settings LUT read interface parameter</td>
</tr>
<tr>
<td>-</td>
<td>0x02C – 0x147</td>
<td>RO</td>
<td>Reserved</td>
</tr>
<tr>
<td>Control</td>
<td>0x148</td>
<td>RW</td>
<td>Control interface: enable and buffer select</td>
</tr>
<tr>
<td></td>
<td>0x14C – 0x17F</td>
<td>RO</td>
<td>Reserved</td>
</tr>
<tr>
<td>RAM n Control</td>
<td>0x180 + 0x10*n</td>
<td>RW</td>
<td>RAM n interface: address and write enable</td>
</tr>
<tr>
<td></td>
<td>0x184 + 0x10*n</td>
<td>RW</td>
<td>Reserved</td>
</tr>
<tr>
<td>RAM n Data Lower</td>
<td>0x188 + 0x10*n</td>
<td>RW</td>
<td>RAM n interface: data, lower 32 bits</td>
</tr>
<tr>
<td>RAM n Data Upper</td>
<td>0x18C + 0x10*n</td>
<td>RW</td>
<td>RAM n interface: data, upper 32 bits (if applicable)</td>
</tr>
</tbody>
</table>

Table 17. vid_pid Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>15:0</td>
<td>3D LUT Product ID: 0x0165</td>
</tr>
<tr>
<td>VID</td>
<td>31:16</td>
<td>Intel FPGA Vendor ID: 0x6AF7</td>
</tr>
</tbody>
</table>

Table 18. version_number Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minor</td>
<td>15:0</td>
<td>Minor version number for this release of the 3D LUT IP</td>
</tr>
<tr>
<td>Major</td>
<td>31:16</td>
<td>Major version number for this release of the 3D LUT IP</td>
</tr>
</tbody>
</table>
### Table 19. `pixels_in_parallel` Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixels in Parallel</td>
<td>31:0</td>
<td>Video data format <strong>Number of pixels in parallel</strong> parameter</td>
</tr>
</tbody>
</table>

### Table 20. `input_bps` Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input BPS</td>
<td>31:0</td>
<td>Video data format <strong>Input bits per color sample</strong> parameter</td>
</tr>
</tbody>
</table>

### Table 21. `output_bps` Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output BPS</td>
<td>31:0</td>
<td>Video data format <strong>Output bits per color sample</strong> parameter</td>
</tr>
</tbody>
</table>

### Table 22. `lut_alpha` Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT alpha</td>
<td>31:0</td>
<td>LUT settings <strong>Output alpha channel</strong> parameter</td>
</tr>
</tbody>
</table>

### Table 23. `lut_depth` Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT depth</td>
<td>31:0</td>
<td>LUT settings <strong>Bits per color</strong> parameter</td>
</tr>
</tbody>
</table>

### Table 24. `lut_double_buffered` Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT double buffered</td>
<td>31:0</td>
<td>LUT settings <strong>Double buffered</strong> parameter</td>
</tr>
</tbody>
</table>

### Table 25. `lut_cpu_readable` Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT CPU readable</td>
<td>31:0</td>
<td>Control settings <strong>LUT read interface</strong> parameter</td>
</tr>
</tbody>
</table>

### Table 26. Control Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>0</td>
<td>• 0: bypass</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1: enable LUT</td>
</tr>
<tr>
<td>Buffer select</td>
<td>1</td>
<td>• 0: buffer 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1: buffer 1 (if <strong>Double buffered</strong> is enabled)</td>
</tr>
<tr>
<td></td>
<td>31:2</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Table 28. RAM \( n \) Control Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>16:0</td>
<td>RAM ( n ) address to write data to or read data from</td>
</tr>
<tr>
<td>27:17</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>Write enable</td>
<td>28</td>
<td>Write enable (clears to 0 automatically)</td>
</tr>
<tr>
<td>31:29</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 29. RAM \( n \) Data Lower Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>31:0</td>
<td>LUT data, lower 32 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write access: first write the new LUT entry data, then set the target address with the write enable asserted in RAM ( n ) Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read access: if you turn on LUT read interface, retrieve the data from RAM ( n ) at the address set in RAM ( n ) Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only present when the LUT data width is greater than 32, i.e.:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((\text{lut_alpha} + 3) \times \text{lut_depth} &gt; 32)</td>
</tr>
</tbody>
</table>

Table 30. RAM \( n \) Data Upper Register

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>31:0</td>
<td>LUT data, upper 32 bits</td>
</tr>
</tbody>
</table>

3D LUT IP Software API

The IP includes a software driver that configures and controls all the necessary parameters of the IP.

Figure 6. Software driver usage example

```c
intel_vvp_core_base base = INTEL_VVP_3D_LUT_0_BASE;
intel_vvp_3d_lut_instance_t intel_vvp_3d_lut_instance;
ret = intel_vvp_3dlut_init (&intel_vvp_3d_lut_instance, base);
if (ret == 0) {
  /* Load LUT if being configured by software */
  if (load_3d_lut(&intel_vvp_3d_lut_instance) == 0) {
    /* Enable LUT processing */
    intel_vvp_3dlut_enable(&intel_vvp_3d_lut_instance, 1);
  } else {
    printf("Error loading LUT data: %d\n", ret);
  }
} else {
  printf("Error initializing intel_vvp_3d_lut_instance: %d\n", ret);
}
```
The driver does not include the load function. You have alternative ways to source the LUT entry data. The simplest is from a precompiled structure in the software source code. An example of using this method is:

```c
int load_3d_lut_(intel_vvp_3d_lut_instance* instance)
{
    uint16_t r_idx = 0, g_idx = 0, b_idx = 0;
    uint32_t table_idx = 0;

    while (table_idx < ((sizeof(udx_lut_table)/sizeof(uint16_t)) - 4))
    {
        int result = intel_vvp_3dlut_load(instance, r_idx, g_idx, b_idx, 0,
                                            udx_lut_table[table_idx],
                                            udx_lut_table[table_idx + 1],
                                            udx_lut_table[table_idx + 2],
                                            udx_lut_table[table_idx + 3]);
        if (result != 0)
        {
            return result;
        }
        table_idx += 4;
        if (++r_idx == UDX_3D_LUT_TABLE_DIMENSION)
        {
            r_idx = 0;
            if (++g_idx == UDX_3D_LUT_TABLE_DIMENSION)
            {
                g_idx = 0;
                if (++b_idx == UDX_3D_LUT_TABLE_DIMENSION)
                {
                    break;
                }
            }
        }
    }
    return 0;
}
```

In this example, the data is a flat structure containing four elements per LUT entry (4 * lut_dimension³).

The definition for the dimension and data table is:

```c
#define UDX_3D_LUT_TABLE_DIMENSION 17
const uint16_t udx_lut_table[] = {
0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,
0x0050, 0x0050, 0x0050, 0x0000,
0x0079, 0x0079, 0x0079, 0x0000,
0x0092, 0x0092, 0x0092, 0x0000,
0x00A6, 0x00A6, 0x00A6, 0x0000,
0x00B9, 0x00B9, 0x00B9, 0x0000,
/* continues... 4913 lines total */
```
Table 31. 3D LUT IP API reference

The software driver for 3D LUT IP provides the following API functions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>intel_vvp_3d_lut_init</code></td>
<td>Initialize the LUT instance</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_enable</code></td>
<td>Enable LUT processing</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_buffer_select</code></td>
<td>Select between LUT buffers</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_load</code></td>
<td>Load a LUT entry</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_get_double_buffered</code></td>
<td>Get double buffered configuration parameter</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_get_input_depth</code></td>
<td>Get bit resolution of input streams</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_get_lut_alpha_channel</code></td>
<td>Get alpha channel support configuration parameter</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_get_lut_depth</code></td>
<td>Get bit resolution of LUT streams</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_get_dimension</code></td>
<td>Get size of LUT</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_get_output_depth</code></td>
<td>Get bit resolution of output streams</td>
</tr>
<tr>
<td><code>intel_vvp_3d_lut_get_pixels_per_clock</code></td>
<td>Get number of pixels processed per clock cycle</td>
</tr>
</tbody>
</table>

**intel_vvp_3d_lut_init**

```c
void intel_vvp_3d_lut_init( intel_vvp_3d_lut_instance* instance, intel_vvp_core_base base);
```

**Description**

Initialize a 3D LUT instance

**Arguments**

- instance – pointer to the 3D LUT software driver instance structure
- base – pointer to base address of 3D LUT IP

**Return Value**

Zero on success, negative integer otherwise

**intel_vvp_3d_lut_enable**

```c
void intel_vvp_3d_lut_enable( intel_vvp_3d_lut_instance* instance, int enable);
```

**Description**

Enable LUT processing

**Arguments**

- instance – pointer to the 3D LUT software driver instance structure
- enable – enable LUT operation:
  - 0 – Passthrough input stream unchanged
  - 1 – Enable LUT processing

**Return Value**

None
**intel_vvp_3d_lut_buffer_select**

```c
int
t
intel_vvp_3d_lut_buffer_select( intel_vvp_3d_lut_instance* instance, uint8_t buffer);
```

**Description**
Select between LUT procession buffers (double buffering must be enabled)

**Arguments**
- `instance` – pointer to the 3D LUT software driver instance structure
- `buffer` – buffer to be selected (0 or 1)

**Return Value**
- 0 – operation is successful
- -1 – buffer parameter is out of range or double buffering is not configured

**intel_vvp_3d_lut_load**

```c
int intel_vvp_3d_lut_load( intel_vvp_3d_lut_instance* instance, uint16_t r_idx, uint16_t g_idx, uint16_t b_idx, uint8_t buffer, uint16_t r_val, uint16_t g_val, uint16_t b_val, uint16_t a_val);
```

**Description**
Load an entry into the LUT table. Parameters specify the indices for the table, and the R/G/B/A value for the table entry.

**Arguments**
- `instance` – pointer to the 3D LUT software driver instance structure
- `r_idx` - red index. Range 0 to (LUT dimension - 1)
- `g_idx` - green index. Range 0 to (LUT dimension - 1)
- `b_idx` - blue index. Range 0 to (LUT dimension - 1)
- `buffer` - range 0 to 1 (for double buffered configuration)
- `r_val` - red value. Range 0 to (2^{lut_depth - 1})
- `g_val` - green value. Range 0 to (2^{lut_depth - 1})
- `b_val` - blue value. Range 0 to (2^{lut_depth - 1})
- `a_val` - alpha value. LUT alpha must be enabled. If not, value must be set to 0.

**Return Value**
- 0 - successful
- -1 if r_idx/g_idx/b_idx is out of range
- -2 if buffer parameter is out of range, or double buffering not configured
- -3 if alpha value is set and not supported
- -4 if r_val/g_val/b_val is out of range
**intel_vvp_3d_lut_get_double_buffered**

```c
uint8_t
intel_vvp_3d_lut_get_double_buffered( intel_vvp_3d_lut_instance* instance);
```

*Description*  Get double-buffered IP configuration

*Arguments*  instance – pointer to the 3D LUT software driver instance structure

*Return Value*  
- 0 if double buffer option is not configured
- 1 if double buffer option is configured

**intel_vvp_3d_lut_get_input_depth**

```c
uint8_t
intel_vvp_3d_lut_get_input_depth( intel_vvp_3d_lut_instance* instance);
```

*Description*  Get bit resolution of input streams

*Arguments*  instance – pointer to the 3D LUT software driver instance structure

*Return Value*  
- Range is 8 to 16. Value is number of bits per input color plane

**intel_vvp_3d_lut_get_lut_alpha_channel**

```c
uint8_t
intel_vvp_3d_lut_get_lut_alpha_channel( intel_vvp_3d_lut_instance* instance);
```

*Description*  Get alpha channel support configuration parameter

*Arguments*  instance – pointer to the 3D LUT software driver instance structure

*Return Value*  
- 0 if alpha channel is not supported
- 1 if alpha channel is supported

**intel_vvp_3d_lut_get_lut_depth**

```c
uint8_t
intel_vvp_3d_lut_get_lut_depth( intel_vvp_3d_lut_instance* instance);
```

*Description*  Get configured bit resolution of LUT processing streams

*Arguments*  instance – pointer to the 3D LUT software driver instance structure
Return Value

**intel_vvp_3d_lut_get_dimension**

```c
uint8_t
intel_vvp_3d_lut_get_dimension( intel_vvp_3d_lut_instance* instance);
```

**Description**
Get configured LUT size. Value is single dimension size. A dimension size of A will result in a LUT size of (A x A x A) entries

**Arguments**
instance – pointer to the 3D LUT software driver instance structure

**Return Value**
Valid values are \{17, 33, 65\}

**intel_vvp_3d_lut_get_output_depth**

```c
uint8_t
intel_vvp_3d_lut_get_output_depth( intel_vvp_3d_lut_instance* instance);
```

**Description**
Get bit resolution of LUT output streams

**Arguments**
instance – pointer to the 3D LUT software driver instance structure

**Return Value**
Range is 8 to 16. Value is number of bits per output color plane

**intel_vvp_3d_lut_get_pixels_per_clock**

```c
uint8_t
intel_vvp_3d_lut_get_pixels_per_clock( intel_vvp_3d_lut_instance* instance);
```

**Description**
Number of input pixels processed for each video clock cycle

**Arguments**
instance – pointer to the 3D LUT software driver instance structure

**Return Value**
Number of pixels. Range is 1 to 8.
The tone mapping operator (TMO) Intel FPGA IP dynamically adapts the processing of an image based on a regional (i.e. tile based) approach. It improves the visibility of latent image detail and enhances the overall viewing experience.

You can configure the required number of bits per symbols, symbols per pixel, and pixels in parallel. Typical applications include:

- Medical imaging
- Machine vision
- Video conferencing
- Surveillance
- Automotive imaging

**Figure 7. Example of processing a real-life image using the TMO IP**

The figure shows example results obtained after applying the TMO IP dataflow on a real-life image: left is the original image; right is the output image after TMO IP processing.

You provide and receive video data to the TMO IP in RGB format via the AXI4-Stream video interfaces. The IP determines the size of the video busses from the number of pixels processed per clock cycle, the color bit depth, and the number of component streams parameters. The number of video component streams is fixed at 3. The IP supports:

- Component bit depths of 8, 10 and 12-bit.
- Pixels per clock of 1, 2 and 4.

You control the strength of the contrast enhancement for the output images provided by the TMO IP via an Avalon memory-mapped control interface. The data bus for the control interface is set to 32-bit to interface with an embedded CPU. During operation, you can configure the TMO IP using a software driver that controls all the IP parameters via a set of software APIs.
The TMO IP supports RGB sampling. The sampling method at the output is always the same as the input. You must provide details of the current standard video resolution via the CPU control interface to ensure correct behavior. The IP only supports 4:4:4 progressive sampling. You should prepare any deinterlacing and chroma up or down sampling externally to the TMO IP.

**TMO IP Features**

- Intel FPGA video streaming data interfaces for video IOs
- Avalon memory-mapped interface for CPU control interfaces
- User defined volume controls to dial up or down contrast enhancement strength
- RGB 8-bit, 10-bit, or 12-bit per color component
- 1, 2, or 4 parallel pixels per clock
- 16 tiles (arranged in a 4x4 grid) for local image statistics collection
- Video resolutions up to 4096x2160 at 60 fps
- Latency of less than 150 pixels
- FPGA footprint of approximately:
  - 7K ALMs
  - 56 DSP blocks
  - 60 M20Ks

**TMO IP Release Information**


The Intel FPGA IP version (X.Y.Z) number can change with each Intel Quartus Prime software version. A change in:

- X indicates a major revision of the IP. If you update the Intel Quartus Prime software, you must regenerate the IP.
- Y indicates the IP includes new features. Regenerate your IP to include these new features.
- Z indicates the IP includes minor changes. Regenerate your IP to include these changes.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>21.2</td>
</tr>
<tr>
<td>Release date</td>
<td>June 2021</td>
</tr>
<tr>
<td>Ordering code</td>
<td>IP-OM-TMO (Contrast enhancement engine)</td>
</tr>
</tbody>
</table>
TMO IP Performance and Resource Utilization

Intel provides resource and utilization data for guidance. TMO IP resource utilization depends on the device family and IP parameters, i.e. number of supported bits per sample and pixels in parallel.

**Table 33. Resource Utilization for Intel Agilex Devices**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Resource Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits per Sample</td>
<td>Pixels in Parallel</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 34. Resource Utilization for Intel Arria 10 Devices**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Resource Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits per Sample</td>
<td>Pixels in Parallel</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 35. Resource Utilization for Intel Cyclone 10 GX Devices**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Resource Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits per Sample</td>
<td>Pixels in Parallel</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

continued...
### Table 36. Resource Utilization for Intel Stratix 10 Devices

Targetting Intel Stratix 10 1SX280LN2F43E1VG device

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Resource Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits per Sample</td>
<td>Pixels in Parallel</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
</tbody>
</table>

### TMO IP Parameters

The IP offers compile-time and run-time parameters

#### Table 37. Compile-time Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video Configuration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of pixels in parallel</td>
<td>1, 2, 4</td>
<td>Number of pixels/samples in parallel</td>
</tr>
<tr>
<td>Number of color planes</td>
<td>3</td>
<td>Number of color planes (4:4:4 RGB video)</td>
</tr>
<tr>
<td>Bits per color sample</td>
<td>8, 10, 12</td>
<td>Number of bits per pixels/samples</td>
</tr>
</tbody>
</table>

For more information on the run-time parameters refer to *TMO IP Registers*. 
TMO IP Block Description

The IP accepts RGB-format video input as an Intel FPGA video streaming interface, statistically analyses image content (locally and globally), and dynamically enhances the luma range to improve overall image contrast. This IP enhances input video frame imagery into a well-lit and detailed image.

Figure 8. TMO IP GUI

Tone Mapping Operator (TMO) Intel FPGA IP

Figure 9. TMO IP High-level block diagram.
The TMO IP consists of several blocks for video processing, memory, and control. The video datapath includes a luma extractor, image statistics calculator, a soft-processor-based mapping LUT generator, CPU register interface, a contrast enhancement engine, and an image enhancer.

The luma extractor takes an RGB input frame, analyzes it, and extracts luminance. The image statistics calculator takes luma information contained in a video frame and provides a set of global and local statistic parameters regarding the contrast information on the input video frame.

The IP collects local information about the input images in different regions on a video frame, providing the necessary granularity to properly enhance contrast in areas within the video frame that need to be adjusted.

The soft-processor-based mapping LUT generator takes the data gathered from the image statistics calculator block and generates a set of mapping transfer functions. The IP temporarily stores the mapping transfer functions in LUTs to reduce resource utilization footprint.

The contrast enhancement engine applies different amounts of mapping transfer functions in different regions of a video frame, providing the necessary granularity to properly enhance contrast in areas within the frame that you need to adjust. The TMO IP does not use external video frame buffers. Consequently, the contrast enhancement process that the IP applies to the current frame uses statistic information it collects from the previous video frame.

The image enhancer takes the image statics information gathered from the input video frame and with the generated mapping transfer function, it enhances the luma range. The image enhancer calculates a set of weights that it applied to the input RGB data to generate contrast enhanced RGB output video streams.

The embedded Nios® II processor used as a mapping LUT generator, is packaged as part of the TMO IP, and customers do not have direct access to it. An external Avalon-MM CPU control interface is then provided for you to interact and configure TMO IP, giving them access to the control registers. Due to a higher level of abstraction, a set of software API is provided as part of the TMO IP delivery package, so you can easily configure and interact with the IP.
Figure 10. **Graphical description of a tile-based histogram generation**

The figure shows a graphical description regarding the tile-based approach, explicitly showing tiles boundaries. Tile boundaries are not visible when you operate the TMO IP. The figure shows them only to demonstrate the IP operation.

---

**TMO IP Interfaces**

The IP has three functional interfaces, three clock domains, and three resets.
Functional Interfaces

The TMO IP has three functional interfaces:

- Intel FPGA video stream input interface (axi4s_vid_in)
- Intel FPGA video stream output interface (axi4s_vid_out)
- External Avalon memory-mapped compatible CPU interface (av_mm_cpu_agent)

The Intel FPGA video streaming protocol is a standard interface to connect components that exchange data.

Avalon memory-mapped interface

The IP external CPU interface is an Avalon memory-mapped interface that accesses the control and status registers.

For CPU control interfaces, the TMO IP uses the Avalon memory-mapped protocol. AXI4 protocols are natively supported in Platform Designer. You can automatically adapt to and from Avalon memory mapped interfaces.

The Avalon memory-mapped interface is an address-based read and write interface typical of host and agent connections. A host is the interface that initiates a transfer request, and an agent is the interface that receives the transfer request. The Avalon memory-mapped interface allows you to dynamically control parameters within the TMO IP by connecting the TMO IP Avalon-MM interface (Avalon memory-mapped agent) to an embedded ARM processor or soft system processor such as a Nios™ II processor (Avalon memory-mapped host).

You can control the TMO IP through the Avalon memory-mapped interface with the IP drivers and API functions.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Avalon Specification Name</th>
<th>Direction</th>
<th>Width (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>av_mm_cpu_agent_address</td>
<td>address</td>
<td>Host to Agent</td>
<td>7</td>
<td>For a host, by default, the address signal represents a byte address. The value of the address must align to the data width. To write to specific bytes within a data word, the host must use the byteenable signal. For an agent, by default, the interconnect translates the byte address into a word address in the agent’s address space. From the perspective of the agent, each agent access is for a word of data.</td>
</tr>
<tr>
<td>av_mm_cpu_agent_byteenable</td>
<td>byteenable</td>
<td>Host to Agent</td>
<td>4</td>
<td>Enables one or more specific byte lanes during transfers on interfaces of width greater than 8 bits. Each bit in byteenable corresponds to a byte in writedata and readdata. The host bit &lt;n&gt; of byteenable indicates whether the IP is writing to byte &lt;n&gt;. During writes, byteenable specify which bytes the IP is writing to. Other bytes should be ignored by the agent. During reads, byteenable indicate which bytes the host is reading. Agents that return read data with no side</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Specification Name</th>
<th>Direction</th>
<th>Width (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>av_mm_cpu_agent_write</td>
<td>write</td>
<td>Host to Agent</td>
<td>1</td>
<td>Asserted to indicate a write transfer. If present, writedata is required. Required for interfaces that support writes.</td>
</tr>
<tr>
<td>av_mm_cpu_agent_writedata</td>
<td>writedata</td>
<td>Host to Agent</td>
<td>32</td>
<td>Data for write transfers. The width must be the same as the width of readdata if both are present. Required for interfaces that support writes.</td>
</tr>
<tr>
<td>av_mm_cpu_agent_read</td>
<td>read</td>
<td>Host to Agent</td>
<td>1</td>
<td>Asserted to indicate a read transfer. If present, readdata is required. Required for interfaces that support reads.</td>
</tr>
<tr>
<td>av_mm_cpu_agent_readdata</td>
<td>readdata</td>
<td>Agent to Host</td>
<td>32</td>
<td>You drive the readdata from the agent to the host in response to a read transfer. Required for interfaces that support reads.</td>
</tr>
<tr>
<td>av_mm_cpu_agent_readdatavalid</td>
<td>readdatavalid</td>
<td>Agent to Host</td>
<td>1</td>
<td>Use for variable-latency, pipelined read transfers. When asserted, indicates that the readdata signal contains valid data. For a read burst with burstcount value &lt;n&gt;, the readdatavalid signal must be asserted &lt;n&gt; times, once for each read data item. Ensure at least one cycle of latency between acceptance of the read and assertion of readdatavalid. An agent may assert read data valid to transfer data to the host independently of whether the agent is stalling a new command with waitrequest. Required if the host supports pipelined reads. Bursting hosts with read functionality must include the readdatavalid signal.</td>
</tr>
</tbody>
</table>
| av_mm_cpu_agent_waitrequest    | waitrequest        | Agent to Host | 1            | An agent asserts waitrequest when unable to respond to a read or write request. Forces the host to wait until the interconnect is ready to proceed with the transfer. At the start of all transfers, a host initiates the transfer and waits until waitrequest is deasserted. A host must make no assumption about the assertion state of waitrequest when the host is idle: wait request may be high or low, depending on system properties. When wait request is asserted, host control signals to the agent must remain constant except for begin burst transfer. An Avalon memory mapped agent may assert waitrequest during idle cycles. An Avalon memory mapped host may }
initiate a transaction when waitrequest is asserted and wait for that signal to be deasserted. To avoid system lockup, an agent device should assert waitrequest when in reset.

**Clocks**

**Table 38. TMO IP Clocks**

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Direction</th>
<th>Width (Bits)</th>
<th>Associated Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal_cpu_clock_clk</td>
<td>Input</td>
<td>1</td>
<td>N.A</td>
<td>Input clock for the soft-processor-based mapping LUT</td>
</tr>
<tr>
<td>external_cpu_clock_clk</td>
<td>Input</td>
<td>1</td>
<td>CPU control interface</td>
<td>Input clock for the external CPU control interface</td>
</tr>
<tr>
<td>video_clock_clk</td>
<td>Input</td>
<td>1</td>
<td>Video input and output interfaces</td>
<td>Input clock for the video and processing datapath</td>
</tr>
</tbody>
</table>

**Table 39. Video Clock Frequency Range Values**

<table>
<thead>
<tr>
<th>Device Family</th>
<th>Frequency range (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Cyclone 10 GX</td>
<td>150 to 300</td>
</tr>
<tr>
<td>Intel Arria 10</td>
<td>150 to 300</td>
</tr>
<tr>
<td>Intel Stratix 10</td>
<td>150 to 400</td>
</tr>
<tr>
<td>Intel Agilex</td>
<td>150 to 600</td>
</tr>
</tbody>
</table>

Frequency depends on:
- Number of pixels in parallel
- Maximum video resolution
- Device family

All three input clocks are asynchronous from each other. Internally, the TMO IP includes clock domain crossing (CDC) circuits for both single bit and data bus signal cases, which safely allows data exchange between any of the three asynchronous clock domains. The TMO IP also includes an embedded .sdc file, which provides all the necessary information to the Timing Analyzer. For system integration, when you instantiate the TMO IP in a design, the only constraints required are:
- Cock frequency constraints for the video clock (video_clock_clk)
- CPU clock (external_cpu_clock_clk)
- Soft-processor-based mapping LUT generator clock (internal_cpu_clock_clk)
Resets

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width (Bits)</th>
<th>Type</th>
<th>Associated Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal_cpu_reset_reset</td>
<td>Input</td>
<td>1</td>
<td>Active-high</td>
<td>N.A</td>
<td>Input reset for the soft-processor-based mapping LUT generator</td>
</tr>
<tr>
<td>external_cpu_reset_reset</td>
<td>Input</td>
<td>1</td>
<td>Active-high</td>
<td>CPU control</td>
<td>Input reset for the external CPU control interface</td>
</tr>
<tr>
<td>video_reset_reset</td>
<td>Input</td>
<td>1</td>
<td>Active-high</td>
<td>Video input and output</td>
<td>Input reset for the video and processing datapath</td>
</tr>
</tbody>
</table>

Ensure before connecting the reset signals to the TMO IP, they are synchronized with their respective associated clock domain. Platform Designer provides a Reset Bridge IP for this task.

**Related Information**
- Video and Vision Processing IP Interfaces on page 8
- Platform Designer: Reset Bridge
  
The Reset Bridge allows you to use a reset signal in two or more subsystems of your Platform Designer system.

**TMO IP Latency**

The latency information can predict the approximate latency between the input and the output of your video processing pipeline.

**Table 40. TMO Latency**

The table shows latency as a number of valid clock cycles. Intel measures the latency assuming that other functions are not stalling the IP on the datapath, i.e., the output ready signal is high.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing or bypass</td>
<td>106</td>
</tr>
</tbody>
</table>

**TMO IP Registers**

The TMO IP allows runtime configuration parameters via AXI4-Lite CPU register interface.

The IP offers the following categories of runtime configuration parameters:

- Flow control parameters that allow you to put the TMO IP into either reset, bypass, or operational mode.
- Status and debug parameters that provide information about compile-tile parameters and on-the-flight status of the TMO IP
- Video configuration parameters that allow you to configure the input video frame geometry
- Image statistics collection parameters that allow you to configure the tile’s dimension
Table 41. **Register Map**

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Byte Address Offset</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>vid_pid</td>
<td>0x000</td>
<td>RO</td>
</tr>
<tr>
<td>version_number</td>
<td>0x004</td>
<td>RO</td>
</tr>
<tr>
<td>reserved_area</td>
<td>0x140 – 0x147</td>
<td>Reserved</td>
</tr>
<tr>
<td>ip_information_0</td>
<td>0x148</td>
<td>RO</td>
</tr>
<tr>
<td>ip_information_1</td>
<td>0x14C</td>
<td>RO</td>
</tr>
<tr>
<td>ip_information_2</td>
<td>0x150</td>
<td>RO</td>
</tr>
<tr>
<td>vid_flow_control</td>
<td>0x154</td>
<td>RW</td>
</tr>
<tr>
<td>actv_vid_size</td>
<td>0x158</td>
<td>RW</td>
</tr>
<tr>
<td>volume_control</td>
<td>0x15C</td>
<td>RW</td>
</tr>
<tr>
<td>tmoDerivedParameters</td>
<td>0x160 – 0x17F</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Table 42. **vid_pid**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Product Identification Number.</td>
</tr>
</tbody>
</table>

Table 43. **version_number**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Version Number.</td>
</tr>
</tbody>
</table>

Table 44. **reserved_area**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Reserved register area</td>
</tr>
</tbody>
</table>

Table 45. **ip_information_0**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27:24</td>
<td>Number of tiles (C_TILE)</td>
</tr>
<tr>
<td>23:16</td>
<td>AXI4-Stream data width</td>
</tr>
<tr>
<td>11:8</td>
<td>Pixels in parallel (C_PIXELS)</td>
</tr>
<tr>
<td>7:4</td>
<td>Components per sample (C_STREAMS)</td>
</tr>
<tr>
<td>3:0</td>
<td>Bits per component (C_DEPTH)</td>
</tr>
</tbody>
</table>

Table 46. **ip_information_1**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>29:25</td>
<td>Fractional precision for luminance weights (C_FRAC_PREC_MLUT)</td>
</tr>
<tr>
<td>24:20</td>
<td>Fractional precision for TMO volume control (C_FRAC_PREC_VOLCNTR)</td>
</tr>
<tr>
<td>19:15</td>
<td>Fractional precision for RGB to luma conversion (C_FRAC_PREC_RGB2LUMA)</td>
</tr>
</tbody>
</table>

...continued...
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:10</td>
<td>Fractional precision for luma to RGB conversion (C_FRAC_PREC_LUMA2RGB)</td>
</tr>
<tr>
<td>9:5</td>
<td>Histogram address data width (C_HIST_ADDR_WIDTH)</td>
</tr>
<tr>
<td>4:0</td>
<td>Histogram data width (C_HIST_DATA_WIDTH)</td>
</tr>
</tbody>
</table>

**Table 47. ip_information_2**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21:17</td>
<td>Fractional precision for interpolation (C_FRAC_PREC_INTP)</td>
</tr>
<tr>
<td>16:12</td>
<td>Fractional precision for histogram equalization (C_FRAC_PREC_HEQ)</td>
</tr>
<tr>
<td>11:0</td>
<td>Fractional precision for histogram normalization factor (C_NORM_FACT)</td>
</tr>
</tbody>
</table>

**Table 48. vid_flow_control**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Soft-reset bit. When set to 1 the TMO IP is in reset.</td>
</tr>
<tr>
<td>0</td>
<td>Passthrough bit. When set to 1, the TMO IP is in bypass mode, i.e. it does not perform any tone mapping operation on the input images.</td>
</tr>
</tbody>
</table>

**Table 49. actv_vid_size**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>29:16</td>
<td>Total number of active pixels per video line (C_WIDTH)</td>
</tr>
<tr>
<td>13:0</td>
<td>Total number of active lines per video frame (C_HEIGHT)</td>
</tr>
</tbody>
</table>

**Table 50. volume_control**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22:16</td>
<td>Fine-level TMO volume control. Valid range [0:100] decimal</td>
</tr>
<tr>
<td>13:0</td>
<td>Coarse-level TMO strength threshold. Valid range [0:9000] decimal</td>
</tr>
</tbody>
</table>

**Table 51. tmo_derived_parameters**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>This area is reserved for all derived parameter registers. Do not write or read from it.</td>
</tr>
</tbody>
</table>

**TMO IP Software API**

The IP includes a software driver that configures and controls all the IP parameters.

**Software Driver Example**

```c
int main()
{
    int ret = -1;
    /* Initialize datapath Video Timing Generator instance */
    /* This API function is not part of the TMO driver. */
    /* Hence, it is up to the user to implement it, to provide VIDEO_WIDTH, VIDEO_HEIGHT values */
    datapath_vtiming_config(VTIMING_TMO_BASE, 1);
}```
/* Initialize TMO IP instance*/
intel_vvp_base_t base = INTEL_VVP_TMO_BASE;
in tel_vvp_tmo_instance_t intel_vvp_tmo_instance;

/*Query TMO IP instance, to know whether is has been initialized correctly. */
/* A value == '0' indicates TMO IP has been initialized correctly */
ret = intel_vvp_tmo_init_instance(&intel_vvp_tmo_instance, base);

if (ret == 0) {
    /* TMO IP Bypass control */
    uint32_t bypass = 0;
    intel_vvp_tmo_set_bypass(&intel_vvp_tmo_instance, bypass);
    printf("Intel VVP TMO Bypass %s\n", bypass ? "ENABLED" : "DISABLED");

    /* Initialize TMO IP to specific video resolution */
    intel_vvp_tmo_set_resolution(&intel_vvp_tmo_instance, VIDEO_WIDTH,
        VIDEO_HEIGHT);

    /* Update TMO IP volume control values */
    intel_vvp_tmo_set_volume(&intel_vvp_tmo_instance, VOLUME_CTL_USER);
    intel_vvp_tmo_set_threshold(&intel_vvp_tmo_instance, INT_THR_USER);

    printf("TMO initialization done\n");
    fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);

    while (1) {
        /* This function checks if TMO IP bypass mode and/or volume control
            values need to be updated */
        /* to implement their own function according to their specific needs */
        /* but it is not part of the TMO IP SW driver package. Hence, customers
            are expected */
        /* This part of the code is for debug purpose only */
        /* Two conditions are checked: */

        /* Condition #1: Missing data collection per tile */
        static const uint32_t num_rows = 4;
        static const uint32_t row_mask = (~(~0u << num_rows);

        /* Read TMO IP debug register to check for error conditions */
        uint32_t reg_val = intel_vvp_tmo_get_debug(&intel_vvp_tmo_instance);

        if (reg_val & row_mask) {
            printf("Timeout rows: ");
            for (uint32_t i = 0; i < num_rows; ++ i) {
                if (reg_val & (0x1 << i))
                    printf("%"PRIu32" ", i);
            }
            printf("\n");
        }

        /* Condition #2: Missing entire video frame */
        if (reg_val & 0x100) {
            printf("FSYNC toggle error\n");
        }
    }
}
Tone Mapping Operator Intel FPGA IP

void process_user_input(intel_vvp_tmo_instance* instance)
{
    int c = getchar();
    static const int32_t VOL_STEP = 5;
    static const int32_t TS_STEP = 1000;
    int vol_delta = 0;
    int ts_delta = 0;

    if(c != EOF)
    {
        switch(c)
        {
            case 'w':
            case 'W':
                vol_delta = VOL_STEP;
                break;
            case 's':
            case 'S':
                vol_delta = -VOL_STEP;
                break;
            case 'd':
            case 'D':
                ts_delta = TS_STEP;
                break;
            case 'a':
            case 'A':
                ts_delta = -TS_STEP;
                break;
            case 'b':
            case 'B':
            {
                uint32_t bypass = intel_vvp_tmo_get_bypass(instance);
                bypass ^= 0x1;
                intel_vvp_tmo_set_bypass(instance, bypass);
                printf("TMO Bypass %s\n", bypass ? "ENABLED" : "DISABLED");
            }
                break;
            default:
                break;
        }
    }

    if(vol_delta || ts_delta)
    {
        int32_t vol = (int32_t)(intel_vvp_tmo_get_volume(instance));
        int32_t ts = (int32_t)(intel_vvp_tmo_get_threshold(instance));

        vol += vol_delta;
        ts += ts_delta;

        if(vol > 100)
        {
            vol = 100;
        }

        intel_vvp_tmo_set_volume(instance, vol);
        intel_vvp_tmo_set_threshold(instance, ts);
        intel_vvp_tmoأفر

    return ret;
}
All the API functions require a pointer to `intel_vvp_tmo_instance_t` structure as the first parameter. The structure represents an individual instance of TMO IP and defined as following:

```c
typedef struct intel_vvp_tmo_instance {
    intel_vvp_tmo_base_t base;
} intel_vvp_tmo_instance_t;
```

Where:
- `intel_vvp_tmo_base_t` base is a platform specific access handler that the driver uses to access configuration and control registers of the IP. Default definition for a bare metal environment is 32-bit unsigned integer representing base address of the TMO IP on the external CPU bus.

The internal driver uses the following macros to access individual registers of the IP:
- `tmoss_read_reg(x)` – read register
- `tmoss_write_reg(x, y)` – write register

Where:
- `x` is the byte offset of the register from the IP core base address
- `y` is the 32-bit register value to write.

Default bare metal implementation of the IP register access provided in the file `intel_vvp_tmo_io.h`. Provide alternative implementations through separate header files included conditionally from `intel_vvp_tmo_io.h`.

Byte offsets of all TMO IP registers are defined in the file `intel_vvp_tmo_regs.h`.

### Table 52. Software driver API reference
The software driver for TMO IP provides various API functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>intel_vvp_tmo_init_instance</code></td>
<td>Initialize TMO IP driver instance</td>
</tr>
<tr>
<td><code>intel_vvp_tmo_set_bypass</code></td>
<td>Set TMO IP into bypass mode</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intel_vvp_tmo_get_bypass</td>
<td>Return current setting of the bypass mode</td>
</tr>
<tr>
<td>intel_vvp_tmo_set_resolution</td>
<td>Set input video resolution for TMO IP</td>
</tr>
<tr>
<td>intel_vvp_tmo_set_volume</td>
<td>Adjust fine-level TMO volume as a percentage, which allows more granularity when setting TMO strength value</td>
</tr>
<tr>
<td>intel_vvp_tmo_get_volume</td>
<td>Return current level of TMO strength</td>
</tr>
<tr>
<td>intel_vvp_tmo_set_threshold</td>
<td>Adjust coarse level TMO volume, which allows setting TMO strength value</td>
</tr>
<tr>
<td>intel_vvp_tmo_get_threshold</td>
<td>Return current value of fine-level TMO volume adjustment</td>
</tr>
<tr>
<td>intel_vvp_tmo_get_debug</td>
<td>Return current value of debug information register</td>
</tr>
<tr>
<td>intel_vvp_tmo_set_debug</td>
<td>Clear individual bits in debug information register</td>
</tr>
</tbody>
</table>

**intel_vvp_tmo_init_instance**

```c
int intel_vvp_tmo_init_instance(intel_vvp_tmo_instance_t* instance, intel_vvp_tmo_base_t base)
```

*Description*  
Initialize TMO software driver instance

*Arguments*  
- `instance` – pointer to the TMO software driver instance structure
- `base` – platform specific IP access handle. In a bare metal environment this is typically defined as 32-bit unsigned integer representing base address of the IP on the external CPU bus

*Return Value*  
Zero on success, negative integer otherwise

**intel_vvp_tmo_set_bypass**

```c
void intel_vvp_tmo_set_bypass(intel_vvp_tmo_instance_t* instance, uint32_t bypass)
```

*Description*  
Enable/disable TMO bypass mode. With bypass mode the TMO IP does not process the incoming video stream and passes it as is.

*Arguments*  
- `instance` – pointer to the TMO software driver instance structure
- `bypass` - 1 – enable bypass mode; 0 – disable bypass mode

*Return Value*  
void
**intervp_tmo_get_bypass**

```c
uint32_t
intervp_tmo_get_bypass(intervp_tmo_instance_t* instance)
```

*Description*  
Get the current setting of the bypass mode

*Arguments*  
instance – pointer to the TMO software driver instance structure

*Return Value*  
1 – bypass mode is enabled; 0 – bypass mode is disabled

**intervp_tmo_set_resolution**

```c
void
intervp_tmo_set_resolution(intervp_tmo_instance_t* instance, const uint32_t width, const uint32_t height)
```

*Description*  
Set up TMO IP for the required video resolution

*Arguments*  
instance – pointer to the TMO software driver instance structure  
width - video width in pixels e.g. 1920  
height - video height in lines e.g. 1080

*Return Value*  
void

**intervp_tmo_set_volume**

```c
void
intervp_tmo_set_volume(intervp_tmo_instance_t* instance, const uint32_t volume)
```

*Description*  
Set desired tone enhancement strength

*Arguments*  
instance – pointer to the TMO software driver instance structure  
volume - tone enhancement strength in the range [0..100]

*Return Value*  
void

**intervp_tmo_get_volume**

```c
uint32_t
intervp_tmo_get_volume(intervp_tmo_instance_t* instance)
```
Get currently configured tone enhancement strength

**Arguments**

instance – pointer to the TMO software driver instance structure

**Return Value**

Current tone enhancement strength in the range [0..100]

### `intel_vvp_tmo_set_threshold`

```c
void intel_vvp_tmo_set_threshold(intel_vvp_tmo_instance_t* instance, const uint32_t threshold)
```

**Description**

Set fine-level tone enhancement strength

**Arguments**

instance – pointer to the TMO software driver instance structure

threshold – fine level tone enhancement strength in the range [0..10000]

**Return Value**

void

### `intel_vvp_tmo_get_threshold`

```c
uint32_t intel_vvp_tmo_get_threshold(intel_vvp_tmo_instance_t* instance)
```

**Description**

Get currently configured fine-level tone enhancement strength

**Arguments**

instance – pointer to the TMO software driver instance structure

**Return Value**

Current fine level tone enhancement strength in the range [0..10000]

### `intel_vvp_tmo_get_debug`

```c
uint32_t intel_vvp_tmo_get_debug(intel_vvp_tmo_instance_t* instance)
```

**Description**

Get current value of the Debug information register Ext_Reg_0xA0

**Arguments**

instance – pointer to the TMO software driver instance structure

**Return Value**

Current value of Ext_Reg_0xA0 as 32-bit unsigned integer
int intel_vvp_tmo_set_debug

Description
Clear individual bits of the Debug information register Ext_Reg_0xA0

Arguments
instance – pointer to the TMO software driver instance structure
val - bit mask of the bits within the debug register to clear

Return Value
void
Warp Intel FPGA IP

About the Warp IP

The Warp IP applies an arbitrary warp (image transform) to a video stream. It processes RGB or YUV video streams at resolutions of up to 3840x2160 at 60 fps. It can process either one or two pixels in parallel.

The Warp IP can process arbitrary warps up to a limit of 2:1 for the effective downscale ratio. If, in any region, the warp produces an output image that is downscaled by more than 2:1 from the input image, an error occurs in the IP software.

Typical applications include:
- Camera lens distortion correction
- Projector system distortion correction

Warp IP Features

- Avalon memory-mapped interface for memory access
- Fixed 10 bits per color RGB
- One or two pixels in parallel
- Two to three frame latency
- Maximum image size of 3840x2160
- Minimum image size of 128x64
- Output image width must be multiple of 16
- Output image height must be multiple of 8

Warp IP Release Information

The Intel FPGA IP version (X.Y.Z) number can change with each Intel Quartus Prime software version. A change in:

- X indicates a major revision of the IP. If you update the Intel Quartus Prime software, you must regenerate the IP.
- Y indicates the IP includes new features. Regenerate your IP to include these new features.
- Z indicates the IP includes minor changes. Regenerate your IP to include these changes.

Table 53.  **Warp IP Release Information**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>21.2</td>
</tr>
<tr>
<td>Release date</td>
<td>June 2021</td>
</tr>
<tr>
<td>Ordering code</td>
<td>-</td>
</tr>
</tbody>
</table>

**Warp IP Performance and Resource Utilization**

Intel provides resource and utilization data for guidance. The designs target an Intel Arria 10 10AX115N2F40I2LG device.

Table 54.  **Resource Usage for HD frame processing**

<table>
<thead>
<tr>
<th>Pixel in parallel</th>
<th>Bits per Color Sample</th>
<th>Number of Engines</th>
<th>Maximum Video Width (1)</th>
<th>Memory Buffer Size</th>
<th>ALMs</th>
<th>Memory Blocks (M20K)</th>
<th>DSP Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>HD</td>
<td>~7,000</td>
<td>253</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

Processing frames of up to 1920x1080 resolution. Intel set the video related clocks `axi4s_vid_in_0_clock`, `axi4s_vid_out_0_clock`, and `core_clock` to a minimum of 150 MHz to allow the IP to process 60 fps. Set these clocks to 300 MHz for frame rates of 120 fps.

This example use case will

Table 55.  **UHD Frames at 30 fps**

<table>
<thead>
<tr>
<th>Pixel in parallel</th>
<th>Bits per Color Sample</th>
<th>Number of Engines</th>
<th>Max Video Width (2)</th>
<th>Memory Buffer Size</th>
<th>ALMs</th>
<th>Memory Blocks (M20K)</th>
<th>DSP Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>UHD</td>
<td>~7,000</td>
<td>389</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

Processing frames of up to 3840x2160 resolution at 30 fps. Intel set the video related clocks `axi4s_vid_in_0_clock`, `axi4s_vid_out_0_clock`, and `core_clock` to 300 MHz.

Table 2-9. Parameters and resource figures for 30FPS UHD frame processing

(1) Same maximum video width for input and output.

(2) Same maximum video width for input and output.
Table 56.  UHD Frames at 60 fps
Processing frames of up to 3840x2160 resolution at 60 fps. Intel set the video related clocks `axi4s_vid_in_0_clock`, `axi4s_vid_out_0_clock`, and `core_clock` to 300 MHz.

![Table 56](image)

Warp IP Parameters

The IP offers various compile-time parameters.

Table 57.  Warp IP Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video data format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of pixels in parallel</td>
<td>1 or 2</td>
<td>Number of pixels processed in parallel</td>
</tr>
<tr>
<td>Number of color planes</td>
<td>3</td>
<td>Number of color planes per pixel</td>
</tr>
<tr>
<td>Bits per color sample</td>
<td>10</td>
<td>Number of bits per color sample</td>
</tr>
<tr>
<td>Maximum input video width</td>
<td>2048 or 3840</td>
<td>Maximum number of pixels per input line. Configures the depth of line buffers in the video input block. The IP can process image widths of up to 3840. However, it can process only horizontal resolutions that are a multiple of 4 pixels. For example, the IP can process image widths of 720 or 724 correctly but not widths of 721, 722 or 723.</td>
</tr>
<tr>
<td>Maximum output video width</td>
<td>2048 or 3840</td>
<td>Maximum number of pixels per output line. Configures the depth of line buffers in the video output block.</td>
</tr>
<tr>
<td>Configuration Settings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of engines</td>
<td>1 or 2</td>
<td>Number of processing engines to use. <strong>Number of engines</strong> must match the <strong>Number of pixels in parallel</strong>.</td>
</tr>
<tr>
<td>Memory frame buffer size</td>
<td>SD, HD or UHD</td>
<td>The amount of memory space the IP allocates to each frame buffer. SD is 1024x1024 pixels HD is 2048x2048 pixels UHD is 4096x4096 pixels.</td>
</tr>
</tbody>
</table>

(3) Same maximum video width for input and output.
**Table 58.** Warp IP Throughput for different parameters

<table>
<thead>
<tr>
<th>Number of pixels in parallel</th>
<th>The number of processing engines to use</th>
<th>f_{MAX} (MHz)</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>150</td>
<td>Image resolutions of up to 1920x1080 at 60 fps</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>300</td>
<td>Image resolutions of up to 3840x2160 at 30 fps</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>300</td>
<td>Image resolutions of up to 3840x2160 at 60 fps</td>
</tr>
</tbody>
</table>

**Warp IP Block Description**

The Warp IP accepts RGB or YUV format video input from its Intel FPGA streaming video interface and directly stores the video data in external memory. The video input process has a pool of four frame buffers that it uses to store the incoming video data. The IP accesses the buffers in a cyclic order.

The IP processes the buffered input video by the configured number of warp engines to apply the required warp. Three coefficient tables control the warp engines to define the warp that the IP applies. External memory stores the three coefficient tables that are generated using the Warp IP software API.

The IP defines the required warp with a backward mapping from the output to the input pixel positions. It represents the warp as a subsampled mesh that defines the mapping in 8x8 regions. For output pixel mappings within the 8x8 positions, the warp engine applies bilinear interpolation.

The IP writes back the resultant warped image to external memory in to one of two output video buffers. The IP writes to these dual output buffers alternately.

The video output process generates an RGB or YUV format Intel FPGA streaming video by reading the warped image data from external memory.
Figure 12.  **Warp IP block diagram**

The figure shows a high-level block diagram for the Warp IP with its connection to external memory.

![Warp IP block diagram](image)

**Figure 13.  **Warp image transform examples**

From left: arbitrary warp, four corner warp

![Warp image transform examples](image)
Coefficient Tables

Each engine within the Warp IP has read access to its own set of three coefficient tables that define and control the image transform that the IP applies. The three different tables are:

- Mesh coefficients that define the output to input pixel transform
- Fetch coefficients that control the loading of the input image into the cache memory within the engine(s).
- Filter coefficients that control the mapping from the cache memory as the IP generates the interpolated or filtered output pixels.

The format of the mesh coefficients is different to the mesh data that the IP provides to the software API. The Software API uses 32-bit signed integers for the mesh values; the Warp IP uses a 16-bit offset binary format.

The IP needs just the mesh data to define the warp. The software API uses this mesh data to generate the required coefficient tables.

**Warp Mesh Interpolation**

The IP defines the warp transform using an 8x8 subsampled mesh. This mesh defines the mapping from the output pixel positions to the corresponding input pixel positions. The 8x8 subsampled mesh requires that only the mappings for the following output pixel positions are defined:

$(0,0), (8,0), (16,0) ... (W, 0)$

$(0,8), (8,8), (16,8) ... (W, 8)$

$(0,H), (8,H), (16,H) ... (W,H)$

where $W=8*\text{ceil}(\text{image width}/8)$ and $H=8*\text{ceil}(\text{image height}/8)$

To generate the output pixel positions that lie in between these 8x8 positions, the Warp IP uses bilinear interpolation.

**Output Pixel Interpolation and Filtering**

The IP generates output pixels with the pixel data from the associated input pixel positions as defined by the warp that the IP applies. The IP generates output pixel values with a bicubic interpolation calculation using a 4x4 kernel of the associated input pixel values.

The weightings for the interpolation over the 4x4 kernel are a bicubic function and a variable low pass filtering function. The software API automatically applies the degree of low pass filtering, which it bases on the amount of downscaling that results for that particular region of the warp.

**Blank Skip Regions**

When you configure the Warp IP to substantially downscale regions of an image, large areas of the output image can map to points outside the input image. These unmapped regions result in the IP producing black.
Because these regions in the output image do not require any processing of the input image by the Warp IP, for efficiency the IP skips the processing associated with these regions. This skipping process is setup automatically by the software API which determines, from the desired warp mapping, which regions you may program to skip.

**Warp IP Interfaces**

The Warp IP has four functional interfaces.

The functional interfaces are:
- Intel FPGA video stream input interface
- Intel FPGA video stream output interface
- Avalon Memory-Mapped compatible CPU interface
- Avalon Memory-Mapped compatible memory interface

**Avalon Memory-Mapped CPU interface**

The Warp IP control interface uses a 32bit Avalon Memory-Mapped interface to access control registers.

**Table 59. Avalon Memory-Mapped CPU interface Signals**

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>av_mm_control_agent_address</td>
<td>Input</td>
<td>13</td>
<td>The byte address of the register being accessed.</td>
</tr>
<tr>
<td>av_mm_control_agent_write</td>
<td>Input</td>
<td>1</td>
<td>Assert to indicate a write transfer.</td>
</tr>
<tr>
<td>av_mm_control_agentbyterenable</td>
<td>Input</td>
<td>4</td>
<td>Enables one or more byte lanes during a write transfer.</td>
</tr>
<tr>
<td>av_mm_control_agent_writedata</td>
<td>Input</td>
<td>32</td>
<td>Data for write transfers.</td>
</tr>
<tr>
<td>av_mm_control_agent_read</td>
<td>Input</td>
<td>1</td>
<td>Assert to indicate a read transfer.</td>
</tr>
<tr>
<td>av_mm_control_agent_readdata</td>
<td>Output</td>
<td>32</td>
<td>Data for read transfers.</td>
</tr>
<tr>
<td>av_mm_control_agent_readdatavalid</td>
<td>Output</td>
<td>1</td>
<td>Asserted by the IP to indicate valid read data.</td>
</tr>
<tr>
<td>av_mm_control_agent_waitrequest</td>
<td>Output</td>
<td>1</td>
<td>Asserted by the IP to indicate that the host must wait to complete the transfer.</td>
</tr>
</tbody>
</table>

**Avalon Memory-Mapped Memory interface**

The Warp IP memory interface uses a 512-bit Avalon Memory-Mapped interface to access external memory.

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>av_mm_memory_host_waitrequest</td>
<td>Input</td>
<td>1</td>
<td>Asserted by the agent to indicate that the Warp IP must wait to complete the transfer.</td>
</tr>
<tr>
<td>av_mm_memory_host_readdata</td>
<td>Input</td>
<td>512</td>
<td>Data for read transfers.</td>
</tr>
<tr>
<td>av_mm_memory_host_readdatavalid</td>
<td>Input</td>
<td>1</td>
<td>Assert to indicate valid read data.</td>
</tr>
<tr>
<td>av_mm_memory_host_response</td>
<td>Input</td>
<td>2</td>
<td>The response status of the agent.</td>
</tr>
<tr>
<td>av_mm_memory_host_burstcount</td>
<td>Output</td>
<td>4</td>
<td>Indicates the number of transfers in each burst.</td>
</tr>
<tr>
<td>av_mm_memory_host_writedata</td>
<td>Output</td>
<td>512</td>
<td>Data for write transfers.</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Signal name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>av_mm_memory_host_address</code></td>
<td>Output</td>
<td>32</td>
<td>The byte address of the memory location being accessed.</td>
</tr>
<tr>
<td><code>av_mm_memory_host_write</code></td>
<td>Output</td>
<td>1</td>
<td>Asserted to indicate a write transfer.</td>
</tr>
<tr>
<td><code>av_mm_memory_host_read</code></td>
<td>Output</td>
<td>1</td>
<td>Asserted to indicate a read transfer.</td>
</tr>
<tr>
<td><code>av_mm_memory_host_byteenable</code></td>
<td>Output</td>
<td>64</td>
<td>Enables one or more byte lanes during a write transfer.</td>
</tr>
<tr>
<td><code>av_mm_memory_host_debugaccess</code></td>
<td>Output</td>
<td>1</td>
<td>Not used by the Warp IP.</td>
</tr>
</tbody>
</table>

**Clocking**

The Warp IP has five clock domains, each with a corresponding reset. All clock domains run up to 300 MHz.

**Table 60. Clock Domains**

<table>
<thead>
<tr>
<th>Clock name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>av_mm_control_agent_clock</code></td>
<td>CPU interface clock domain</td>
</tr>
<tr>
<td><code>av_mm_memory_host_clock</code></td>
<td>Memory interface clock domain</td>
</tr>
<tr>
<td><code>axi4s_vid_in_0_clock</code></td>
<td>Input video stream clock domain</td>
</tr>
<tr>
<td><code>axi4s_vid_out_0_clock</code></td>
<td>Output video stream clock domain</td>
</tr>
<tr>
<td><code>core_clock</code></td>
<td>Processing engine clock domain</td>
</tr>
</tbody>
</table>

The CPU interface uses little bandwidth and does not impose a minimum clock frequency.

The video clock frequency depends on the video resolution and frame rate and the Warp IP’s number of pixels in parallel. For example, a 300 MHz clock at 2 pixels in parallel supports active video resolutions up to 3840x2160 at 60 fps. A 150 MHz clock at 1 pixel in parallel supports resolutions up to 1920x1080 at 60 fps.

All RTL-based blocks that transfer or receive data from a different clock domain include clock domain crossing (CDC) circuits for both, single bit and data bus signal cases. The CDC circuits safely allow exchange of data between the two asynchronous clock domains. The Warp IP includes an `.sdc` file to constrain these CDC paths.

**Resets**

**Table 61. Resets associated to clock domains**

All resets are synchronous active-high

<table>
<thead>
<tr>
<th>Reset name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>av_mm_control_agent_reset</code></td>
<td>CPU interface clock domain reset.</td>
</tr>
<tr>
<td><code>av_mm_memory_host_reset</code></td>
<td>Memory interface clock domain reset.</td>
</tr>
<tr>
<td><code>axi4s_vid_in_0_reset</code></td>
<td>Input video stream clock domain reset.</td>
</tr>
<tr>
<td><code>axi4s_vid_out_0_reset</code></td>
<td>Output video stream clock domain reset.</td>
</tr>
<tr>
<td><code>core_reset</code></td>
<td>Processing engine(s) clock domain reset.</td>
</tr>
</tbody>
</table>
All the resets in the Warp IP are synchronous. Ensure that, when resetting the Warp IP, all clocks are active at the same time while you apply the resets. In a typical system, an EMIF IP block drives and controls these signals. The relationship between the various resets and clocks is not always obvious.

**Interrupts**

Table 62. Interrupt Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt_irq</td>
<td>Active high interrupt triggered at the start of each output frame sent from the axi4s_vid_out_0 interface. The signal is synchronous to the av_mm_control_agent_clock domain. Enable interrupt_irq using the interrupt control register and clear using the interrupt status register.</td>
</tr>
</tbody>
</table>

**Warp IP Latency**

The warp IP includes buffers for full frames of video data at the video input and at the output of its processing engines. This buffer introduces a two-frame latency between the input and output video. This two-frame buffering latency together with any delays in resynchronising between the input and output frames produces between two and three frames latency in total.

Table 63. Operation mode latency

<table>
<thead>
<tr>
<th>Mode</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active warp</td>
<td>Two to three video frames.</td>
</tr>
</tbody>
</table>

**External Memory for Warp IP**

The IP requires access to two separate areas of external memory: one for its input and output video buffers and one for its coefficient tables. The processor system running the Warp Software API must be able to access the coefficient tables but does not need access to the buffer area.

**Memory Space Allocation in External Memory**

Table 64. Warp IP Video Buffer Memory Region

The table defines how much space is required in external memory by the Warp IP for the video buffer region. This space depends on the size of the images to be processed in a system. It is defined by the **Space allocated for each frame buffer in memory** parameter. Six buffers require space in total: four input and two output.

<table>
<thead>
<tr>
<th>Buffer Space Configuration</th>
<th>Region Size (MB)</th>
<th>Memory Region Required</th>
<th>Alignment (multiples of)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD buffer size (1024x1024)</td>
<td>24</td>
<td>0x0180_0000</td>
<td>0x00200_0000</td>
</tr>
<tr>
<td>HD buffer size (2048x2048)</td>
<td>96</td>
<td>0x0600_0000</td>
<td>0x00800_0000</td>
</tr>
<tr>
<td>UHD buffer size (4096x4096)</td>
<td>384</td>
<td>0x1800_0000</td>
<td>0x002000_0000</td>
</tr>
</tbody>
</table>

The IP passes the base address of the memory region allocated to the frame buffers to the software API using the ram_addr element in the structure.
The memory region that the coefficient tables require is related to the number of warp engines, the resolution of the images, and the type of warp.

Table 65.  **Warp IP Coefficient Tables Memory Region**  
The table shows the maximum size of the coefficient table memory region, per engine.

<table>
<thead>
<tr>
<th>Warp Engines</th>
<th>Region Size (MB)</th>
<th>Memory Region Required</th>
<th>Alignment (multiples of)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>0x0100_0000</td>
<td>0x0100_0000</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>0x0200_0000</td>
<td>0x0200_0000</td>
</tr>
</tbody>
</table>

**Bandwidth to External Memory**

The performance of the interface from the Warp IP to the external memory is important for the correct operation of a system using the Warp IP.

The Warp IP generates a substantial amount of memory traffic. It has four video streams passing to and from external memory. In addition, each engine has three read streams to access the coefficient tables. All these streams combine to make Warp IP memory accesses complex. The streams affect how much efficiency you can obtain when accessing DDR4 memory.

The Warp IP memory controller mitigates potential inefficiencies caused by these complex access patterns. It uses burst lengths of 8 beats for all its read and write accesses to improve the burst performance to DDR4 memory. It also attempts to cluster individual read and write bursts together to eliminate some of the issues with read and write turnaround dead time at the DDR4 interface.

These memory access patterns depend on applying the image transform. Some complex image transforms may reduce memory traffic because of the skip region functionality. One of the worst transforms for generated memory traffic is a unity warp that gives a 1:1 mapping between input and output pixels.

The operation of the Warp IP is easier to predict when it is the only user of the DDR4 memory in a system. When other high bandwidth accesses are made to the memory at the same time as the Warp IP, ensure that any interactions don’t adversely affect performance.

**Example system sharing access to memory**

In this example system the Warp IP shares the DDR4 interface with a frame buffer in a system that processes UHD frames at 60 fps. The system runs on an Intel Arria 10 GX Development Kit with the DDR4 EMIF running a 2,133 MHz interface to a DDR4 memory.
Figure 14. Warp and Video Frame Buffer Platform Designer

The figure shows the Platform Designer connectivity where the Frame Buffer II component is sharing access to the DDR4 EMIF with the Warp IP. The Frame Buffer is part of the same video processing pipeline as the Warp IP.

For this system to work:

- Configure Frame Buffer to use bursts of 32 beats for read and write.
- Configure Frame Buffer to use read and write FIFO depths of 128
- Set the arbitration weighting at the front end of the DDR4 EMIF to 16:1 in favor of the Warp IP (versus the Frame Buffer’s read and write interfaces connected through the \texttt{mm\_bridge\_vfb} component).
- Set the Maximum pending read transactions parameter in the pipelined transfers section of the Avalon Memory Mapped agent port to be at 8.
- Set Limit interconnect pipeline stages to for the domain at the front end of the DDR4 EMIF to 4.
Figure 15. **Video Frame Buffer Parameterization**

<table>
<thead>
<tr>
<th>Video Data Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum frame width:</td>
<td>3840</td>
</tr>
<tr>
<td>Maximum frame height:</td>
<td>2160</td>
</tr>
<tr>
<td>Bits per color sample:</td>
<td>10</td>
</tr>
<tr>
<td>Number of color planes:</td>
<td>3</td>
</tr>
<tr>
<td>Color planes transmitted in parallel</td>
<td></td>
</tr>
<tr>
<td>Number of pixels in parallel:</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use separate clock for the Avalon-MM master interface(s)</td>
</tr>
<tr>
<td>Avalon-MM master(s) local ports width:</td>
</tr>
<tr>
<td>FIFO depth Write:</td>
</tr>
<tr>
<td>AV-MM burst target Write:</td>
</tr>
<tr>
<td>FIFO depth Read:</td>
</tr>
<tr>
<td>AV-MM burst target Read:</td>
</tr>
</tbody>
</table>

[Click here to view the full page of the document.]
Figure 16. Maximum Pending Read Transactions
Figure 17. Limit interconnect pipeline stages to

<table>
<thead>
<tr>
<th>Parameters</th>
<th>System Info</th>
<th>Component Instantiation</th>
<th>Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Show System with Interconnect</td>
</tr>
</tbody>
</table>

### Memory Mapped Domains
- cpu:master
- vlp:mem_master_r0
- intel:ip:memory

---

### Interconnect Parameters
- **Limit interconnect pipeline stages to**: 4
- **Clock crossing adapter type**: FIFO
- **Automate default slave insertion**: FALSE
- **Enable instrumentation**: FALSE
- **Interconnect reset source**: Default
- **Burst adapter implementation**: Generic converter (slower, lower area)
- **Width adapter implementation**: Generic converter (slower, lower area)
- **Enable ECC protection**: FALSE
- **Use synchronous resets**: FALSE

---

Configure interconnect parameters for selected domains.
# Multiple Warp IPs sharing access to memory

The figure shows an example with two Warp IPs that share a DDR4 interface. To match the burst access patterns of the Warp IP, set the arbitration values at the combining interface to 8.

## Warp IP Registers

As the software API allows you to program and control the warp IP, it only has a limited set of registers.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>Interrupt enable</td>
</tr>
<tr>
<td>axi4s_vid_in_0</td>
<td>Video data input</td>
</tr>
<tr>
<td>axi4s_vid_out_0</td>
<td>Video data output</td>
</tr>
<tr>
<td>core_clock</td>
<td>Clock for core</td>
</tr>
<tr>
<td>core_reset</td>
<td>Reset for core</td>
</tr>
<tr>
<td>av_mm_memory_host_clock</td>
<td>Clock for memory host</td>
</tr>
<tr>
<td>av_mm_memory_host_reset</td>
<td>Reset for memory host</td>
</tr>
<tr>
<td>av_mm_control_agent_clock</td>
<td>Clock for control agent</td>
</tr>
<tr>
<td>av_mm_control_agent_reset</td>
<td>Reset for control agent</td>
</tr>
<tr>
<td>axi4s_vid_in_0</td>
<td>Video data input</td>
</tr>
<tr>
<td>axi4s_vid_out_0</td>
<td>Video data output</td>
</tr>
<tr>
<td>av_mm_control_agent</td>
<td>Control logic</td>
</tr>
<tr>
<td>av_mm_memory_host</td>
<td>Memory host logic</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_reset_n</td>
<td>Reset for global module</td>
</tr>
<tr>
<td>pll_ref_clk</td>
<td>Reference clock for PLL</td>
</tr>
<tr>
<td>cct</td>
<td>Control logic for timing</td>
</tr>
<tr>
<td>mem</td>
<td>Memory interface</td>
</tr>
<tr>
<td>status</td>
<td>Status register</td>
</tr>
<tr>
<td>emif_usr_reset_n</td>
<td>Reset for EMIF user interface</td>
</tr>
<tr>
<td>emif_usr_clk</td>
<td>Clock for EMIF user interface</td>
</tr>
<tr>
<td>ctrl_amrm</td>
<td>Control register for AMRM</td>
</tr>
</tbody>
</table>

---

**Figure 18.** Multiple Warp IPs sharing access to memory

---

**Warp Intel FPGA IP**

UG-20344 | 2021.09.10

Video and Vision Processing Suite Intel® FPGA IP User Guide

67
Table 66. **Warp IP Registers**

These registers are all read-only and allow interrogation of the Warp IP’s parameter settings. All the registers are 32-bit wide.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Offset Address</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vid_pid</td>
<td>0x000</td>
<td>RO</td>
<td>Warp IP product and vendor ID</td>
</tr>
<tr>
<td>version_number</td>
<td>0x004</td>
<td>RO</td>
<td>The version for this release of the Warp IP</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x008</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>0x00C</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>pip</td>
<td>0x010</td>
<td>RO</td>
<td>Indicates value of pixels in parallel parameter</td>
</tr>
<tr>
<td>color_planes</td>
<td>0x014</td>
<td>RO</td>
<td>Indicates value of number of color planes parameter</td>
</tr>
<tr>
<td>cps</td>
<td>0x018</td>
<td>RO</td>
<td>Indicates value of bits per color sample parameter</td>
</tr>
<tr>
<td>num_engines</td>
<td>0x01C</td>
<td>RO</td>
<td>Indicates value of number of engines parameter</td>
</tr>
<tr>
<td>max_input_width</td>
<td>0x020</td>
<td>RO</td>
<td>Indicates value of maximum input video width parameter</td>
</tr>
<tr>
<td>max_output_width</td>
<td>0x024</td>
<td>RO</td>
<td>Indicates value of maximum output video width parameter</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x028-0x16C</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>int_control</td>
<td>0x170</td>
<td>RW</td>
<td>Enables the interrupt</td>
</tr>
<tr>
<td>int_status</td>
<td>0x174</td>
<td>RW1C</td>
<td>Read interrupt status and clear interrupt</td>
</tr>
</tbody>
</table>

Table 67. **vid_pid Register**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>VID</td>
<td>Vendor ID that returns a value of 0x6AF7</td>
</tr>
<tr>
<td>15:0</td>
<td>PID</td>
<td>Warp product ID that returns a value of 0x016F</td>
</tr>
</tbody>
</table>

Table 68. **version_number Register**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Version Number</td>
<td>The version number of the Warp IP</td>
</tr>
</tbody>
</table>

Table 69. **pip Register**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Pixels in Parallel</td>
<td>The pixel in parallel parameter. Returns a value of 1 or 2.</td>
</tr>
</tbody>
</table>

Table 70. **color_planes Register**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Number of Color Planes</td>
<td>The number of color planes parameter. Returns a value of 3.</td>
</tr>
</tbody>
</table>

Table 71. **bps Register**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Bits per Color Sample</td>
<td>The bits per color sample parameter. Returns a value of 10.</td>
</tr>
</tbody>
</table>
Table 72. `num_engines` Register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Number of Engines</td>
<td>The number of engines parameter. Returns a value of 1 or 2.</td>
</tr>
</tbody>
</table>

Table 73. `max_input_width` Register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Maximum input video width</td>
<td>The maximum input video width parameter. Returns a value of 2048 or 3840.</td>
</tr>
</tbody>
</table>

Table 74. `max_output_width` Register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Maximum output video width</td>
<td>The maximum output video width parameter. Returns a value of 2048 or 3840.</td>
</tr>
</tbody>
</table>

Table 75. `int_control` Register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Interrupt Enable</td>
<td>Setting this bit to 1 will enable the interrupt. Setting to 0 will disable the interrupt.</td>
</tr>
</tbody>
</table>

Table 76. `int_status` Register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Interrupt Status</td>
<td>Reading from this bit returns the status of the interrupt. Writing a 1 to this bit will clear the interrupt. Once triggered, the interrupt will remain set until it is cleared by writing a 1 to this bit.</td>
</tr>
</tbody>
</table>

Warp IP Software API

The Warp IP software is implemented in C and C++. This topic describes the API functions in the code that you may use.

Figure 19. Warp software architecture

The software includes:
- `intel_vvp_warp_driver`
- `intel_vvp_warp_data`
- `intel_vvp_warp_mesh`

The `intel_vvp_warp_driver` is the warp IP driver. It provides API for initializing, configuring, and controlling the Warp IP, warp video channel management, and using debug features.
The `intel_vvp_warp_data` is the software component that generates mesh, cache, and filter coefficient data required by the warp IP. Place generated data in a region of RAM accessed by the warp IP. This location and other necessary parameters is passed to the warp driver.

This component requires transformation mesh in a predefined format as the main input data. It also requires some IP parameters e.g. number of available engines, processing block dimensions, block cache size.

The `intel_vvp_warp_mesh` is the software component that generates warp meshes for common affine transformations such as translation, rotation, zoom; perspective transformations e.g. keystone and radial distortion (fisheye) and arbitrary warps defined by a set of curves. Meshes are generated in the format required by the `intel_vvp_warp_data` interface.

You can use this reference software in a number of real-world applications. However, in some more complex cases you might want to use external software to generate transformation mesh. For example, when projecting image onto a complex surface.

You can deploy the software on a Nios II based system. However, because of computational intensity of the mesh and data generating components, Intel recommends an SoC with a dedicated CPU such as an Intel Arria 10 SX device.

**Software Examples**

**intel_vvp_warp_driver**

**Table 77. intel_vvp_warp_driver API reference**

The software driver for Warp IP provides the following set of API functions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>intel_vvp_warp_init_instance</code></td>
<td>Initialize driver instance</td>
</tr>
<tr>
<td><code>intel_vvp_warp_create_channel</code></td>
<td>Create single engine video processing channel</td>
</tr>
<tr>
<td><code>intel_vvp_warp_create_double_channel</code></td>
<td>Create dual engine video processing channel</td>
</tr>
<tr>
<td><code>intel_vvp_warp_add_engine_to_channel</code></td>
<td>Add a warp engine to existing channel</td>
</tr>
<tr>
<td><code>intel_vvp_warp_free_channel</code></td>
<td>Free video processing channel</td>
</tr>
<tr>
<td><code>intel_vvp_warp_configure_channel</code></td>
<td>Configure video processing channel</td>
</tr>
<tr>
<td><code>intel_vvp_warp_apply_transform</code></td>
<td>Apply video transformation</td>
</tr>
<tr>
<td><code>intel_vvp_warp_bypass</code></td>
<td>Bypass video processing</td>
</tr>
<tr>
<td><code>intel_vvp_warp_reset_skip_ram</code></td>
<td>Reset skip RAM page</td>
</tr>
</tbody>
</table>

```c
int intel_vvp_warp_init_instance(intel_vvp_warp_instance_t* instance, intel_vvp_warp_base_t base)
```

**Description**  Initialize `intel_vvp_warp` driver instance
**Arguments**

- `instance` – pointer to the `intel_vvp_warp` software driver instance structure
- `base` – hardware access handler. In a bare metal environments such as Nios II it is defined as 32-bit unsigned integer representing physical address of the `intel_vvp_warp` IP on the CPU bus.

**Return value**

- Zero on success, negative integer otherwise

**intel_vvp_warp_create_channel**

```c
intel_vvp_warp_create_channel(intel_vvp_warp_instance_t *instance, uint32_t input_idx, uint32_t engine_idx, uint32_t output_idx)
```

**intel_vvp_warp_create_double_channel**

```c
intel_vvp_warp_create_double_channel(intel_vvp_warp_instance_t *instance, uint32_t input_idx0, uint32_t engine1_idx, uint32_t engine2_idx, uint32_t output_idx)
```

**Description**

Create a video processing channel. The functions allocate hardware resources and initialize driver data structures necessary for processing a video stream up to 3840x2160 pixels.

**Arguments**

- `instance` – pointer to initialized `intel_vvp_warp_instance_t` structure;
- `input_idx`, `engine_idx`, `engine2_idx`, `output_idx` – indexes of the warp IP input, engines and output to use in this channel.

You can configure a single warp IP to provide multiple input, output, and processing engine blocks. These blocks are grouped into video processing channels arbitrarily by using index of the block.

The indexes must be in the range 0..num_inputs(num_outputs or num_engines respectively), where you can obtain num_inputs, num_outputs and num_engines through the `intel_vvp_warp_instance_t` structure after you initialize the driver instance.

**Return value**

- Valid pointer to initialized `intel_vvp_warp_channel_t` structure on success, null pointer otherwise.

**intel_vvp_warp_add_engine_to_channel**

```c
int intel_vvp_warp_add_engine_to_channel(intel_vvp_warp_channel_t *channel, uint32_t engine_idx)
```

**Description**

Add a warp engine to existing channel.
Warp IP allows you to split video processing between multiple engines. This function adds additional engines to already allocated channels and updates driver and channel data structures accordingly.

**Arguments**
- `instance` – pointer to the `intel_vvp_warp` software driver instance structure
- `engine_idx` – indexes of the warp IP engine to add to the channel

**Return value**
- Zero on success, negative integer otherwise

```c
void intel_vvp_warp_free_channel(intel_vvp_warp_channel_t* channel)
```

**Description**
Delete video processing channel and release hardware resources and driver data structures allocated for it.

**Arguments**
- `channel` – a valid pointer to initialized `intel_vvp_warp_channel_t` structure

**Return value**
- None

```c
int intel_vvp_warp_configure_channel(intel_vvp_warp_channel_t* channel, intel_vvp_warp_channel_config_t* cfg)
```

**Description**
Configure video processing channel data structures and allocated hardware by providing necessary parameters such as input/output resolutions, frame buffers location etc.

**Arguments**
- `channel` – a valid pointer to initialized video processing channel;
- `cfg` – a valid pointer to initialized `intel_vvp_warp_channel_config_t` structure.

**Return value**
- Zero on success, negative integer otherwise

```c
int intel_vvp_warp_apply_transform(intel_vvp_warp_channel_t* channel, intel_vvp_warp_data_t* data)
```
**Description**
Start warping video by configuring the channel to use provided warp data coefficients

**Arguments**
- channel – a valid pointer to initialized video processing channel
- data – a valid pointer to initialized intel_vvp_warp_data_t structure.

**Return value**
Zero on success, negative integer otherwise

**intel_vvp_warp_bypass**

```c
int intel_vvp_warp_bypass(intel_vvp_warp_channel_t* channel, uint32_t bypass, uint32_t skip_ram_page)
```

**Description**
Enable/disable video bypass where by input video is displayed as is without any processing

**Arguments**
- channel – a valid pointer to initialized video processing channel
- bypass – a non zero value to enable video bypass, zero to disable
- skip_ram_page – number of skip ram page to use after enabling/disabling bypass

**Return value**
Zero on success, negative integer otherwise

**intel_vvp_warp_reset_skip_ram**

```c
void intel_vvp_warp_reset_skipram(intel_vvp_warp_channel_t* channel, uint32_t skip_ram_page)
```

**Description**
Reset skip RAM page setting all values to zero. Skip RAM pages are programmed as part of intel_vvp_warp_apply_transform(). Some use scenarios may require skip RAM to reset explicitly. This function allows for that

**Arguments**
- channel – a valid pointer to initialized video processing channel
- ram_page – skip RAM page to reset

**Return value**
None
Data structures and types

intel_vvp_warp_base_t

Description
Platform specific type to perform register read and write operations on the warp IP. In bare metal environments such as a Nios II processor, it is defined as 32-bit unsigned integer representing physical address of the warp IP on the CPU bus

intel_vvp_warp_instance

typedef struct
intel_vvp_warp_instance{ intel_vvp_warp_base_t base;
uint32_t num_inputs; uint32_t num_engines; uint32_t num_outputs; } intel_vvp_warp_instance_t

Description
Main data structure of the driver

Members
- base – warp IP access handler
- num_inputs – number of available warp inputs
- num_engines – number of available warp engines
- num_outputs – number of available warp outputs

intel_vvp_warp_channel

typedef struct intel_vvp_warp_channel
{ uint32_t in_use;
uint32_t idx;
struct intel_vvp_warp_instance* instance;
uint32_t num_engines;
uint32_t ram_addr;
} intel_vvp_warp_channel_t

Description
Video processing channel resources and parameters structure

Arguments
- in_use – non-zero value if the channel is allocated and initialized. Zero otherwise.
- idx – index number of the channel
- instance – pointer to the parent driver instance the channel belongs to
- num_engines – number of warp engines for the channel
- ram_addr - base address of RAM region allocated for the channel frame buffers
intel_vvp_warp_channel_config

typedef struct intel_vvp_warp_channel_config
{
    uint32_t ram_addr;
    intel_vvp_warp_cs_t cs;
    intel_vvp_warp_scan_t scan;
    uint32_t width_input;
    uint32_t height_input;
    uint32_t width_output;
    uint32_t height_output;
    uint8_t bypass;
    uint8_t lfr;
} intel_vvp_warp_channel_config_t

Description
Video processing channel configuration

Members
ram_addr - base address of RAM region allocated for the channel frame buffers

cs – video color space. Must be initialized to
    • 00b = YUV
    • 01b = RGB reduced range
    • 10b = RFG full range.

scan – video processing scan. Must be initialized to 0x1

width_input, height_input, width_output, height_output – input and output video dimensions

bypass – 1 to configure the channel into video processing bypass mode

lfr – low frame rate fallback. Must be initialized to 0x0

intel_vvp_warp_data

typedef struct intel_vvp_warp_data
{
    uint32_t num_engines;
    uint8_t* _skip_megablock_data;
    uint32_t _skip_ram_page;
    intel_vvp_warp_engine_data_t engine_data[0];
} intel_vvp_warp_data_t

Description
Warp data descriptor

Members
num_engines – data for how many engines is contained in this set.

_skip_megablock_data – data for the blank skip logic. Generated by the intel_vvp_warp_data library. See WarpData structure.

_skip_ram_page – blank skip page to use for this transform. Warp IP provides 8 blank skip pages. Use these can to keep skip blank data for different transforms and switch between the pages as necessary.
engine_data – array of intel_vvp_warp_engine_data_t structures containing warp coefficients for each individual engine

```c
typedef struct intel_vvp_warp_engine_data {
    uint32_t mesh_addr;
    uint32_t filter_addr;
    uint32_t fetch_addr;
    uint32_t start_h;
    uint32_t start_v;
    uint32_t end_h;
    uint32_t end_v;
    uint32_t mesh_stride;
} intel_vvp_warp_engine_data_t;
```

**Description**

Warp coefficient descriptor for an individual warp engine

**Members**

- mesh_addr – address of the mesh coefficients in memory
- filter_addr – address of the filter coefficients in memory
- fetch_addr – address of the fetch coefficients in memory
- start_h, start_v, end_h, end_v – engine processing region. Internally warp engine processes video in 16x8 pixel blocks with the top-left corner being block [0, 0]. The processing region is defined by the start and end block indexes in horizontal and vertical dimensions.
- mesh_stride – number of warp mesh nodes per mesh data row. (in sets of 4 minus 1). Required mesh stride is calculated by the intel_vvp_warp_data library.

**intel_vvp_warp_data**

This software component uses transformation mesh in a predefined format to generate mesh, cache and filter coefficient data required by the warp IP.

**Table 78. intel_vvp_warp_data API reference**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenerateData</td>
<td>Generate warp data</td>
</tr>
</tbody>
</table>

**GenerateData**

WarpDataPtr GenerateData(const WarpDataContext& ctx, const WarpMeshSet& mesh_set)

**Description**

Generate warp data coefficients using provided transformation mesh, video resolution and necessary IP parameters. If two warp engines are available, the workload is split evenly such that each engine processes either left or right half of the image accordingly.
Arguments

tctx – reference to an instance of WarpDataContext object containing information about video resolution and IP parameters required to generate data

mesh_set – reference to a set of transformation meshes

Return value

On success the function returns a valid smart pointer to a WarpData object. On error nullptr object is returned. If the IP detects a downscale ratio higher than 2:1 in any region of the frame, nullptr is returned.

Data structures and types

WarpMesh

class WarpMesh
{
public:
    WarpMesh(uint32_t width, uint32_t height, uint32_t step);
    uint32_t GetStep();
    uint32_t GetVNodes();
    uint32_t GetHNodes();
    mesh_node_t* GetRow(uint32_t v);
}

Description

Class containing user defined transformation mesh.

The mesh is essentially a look up table that defines a mapping between the output and input images. Each entry of the mesh (node) represents a position in the output image and contains the relevant x and y coordinates of the input image. The step between mesh nodes is defined by the IP. The class automatically manages internal storage for the mesh data.

Members

WarpMesh(uint32_t width, uint32_t height, uint32_t step) - Constructor

Arguments: width, height – output image dimensions

step – distance in pixels between mesh nodes. The IP uses the same distance horizontally and vertically. The IP only supports the step of 8 pixels and ignores all other values.

uint32_t GetStep() - Returns distance between mesh nodes

uint32_t GetVNodes() - Returns number of vertical/horizontal nodes in the mesh respectively

mesh_node_t* GetRow(uint32_t v) - Returns pointer to the mesh row. Access individual nodes in the row using pointer arithmetic or indexing

Arguments: v – index number of the mesh row
typedef struct mesh_node {
    int32_t _x;
    int32_t _y;
} mesh_node_t

**Description** Structure representing individual entry in the mesh

**Members** _x, _y – coordinates of the input image this node maps to. The IP represents each value by a 32-bit signed integer with the four least significant bits containing fractional part for subpixel precision.

**WarpMeshPtr**

using WarpMeshPtr = std::shared_ptr<WarpMesh>

**Description** Type alias for a smart pointer wrapper around the mesh class

**WarpMeshSet**

using WarpMeshSet = std::vector<WarpMeshPtr>

**Description** Type alias for a container allowing to group multiple meshes together

**WarpDataContext**

```c
struct WarpDataContext {
    WarpDataContext(const WarpHwContextPtr hw, uint32_t wi, uint32_t hi, uint32_t wo, uint32_t ho);
};
```

**Description** Structure containing image dimensions and IP parameters required to generate data. If two warp engines are available, the IP splits the workload evenly such that each engine processes either left or right half of the image accordingly.

**Members**

WarpDataContext(const WarpHwContextPtr hw, uint32_t wi, uint32_t hi, uint32_t wo, uint32_t ho) - Constructor. The constructor initializes remaining data members using provided arguments that you should not modify.

Arguments: hw – smart pointer to an instance of WarpHwContext object containing relevant IP parameters

wi, hi – input image width and height

wo, ho – output image width and height
_engine_hblocks – number 16x8 pixel blocks each engine processes horizontally

_engine_mesh_stride – number of mesh nodes per mesh data row each engine processes

_h_blocks_out – total horizontal number of 16x8 blocks in the frame for the given output width

_v_blocks_out – total vertical number of 16x8 blocks in the frame for the given output height

**WarpData**

```c
struct WarpData
{
    uint32_t _engines;
    void* _skip_megablock_data;
    WarpEngineData* _engine_data[MAX_ENGINES];
}
```

**Description**
Structure containing warp data coefficients

**Members**
- _engines – number warp engine data objects contained in this structure (number of valid entries in the _engine_data array member). The value is always less or equal to MAX_ENGINES;
- _skip_megablock_data – data for the skip logic;
- _engine_data – array of pointers to WarpEngineData objects containing data for individual warp engines; The array has maximum fixed size of MAX_ENGINES.

**WarpEngineData**

```c
struct WarpEngineData
{
    uint32_t _mesh_entries;
    uint32_t _filter_entries;
    uint32_t _fetch_entries;
    void* _mesh_data;
    void* _filter_data;
    void* _fetch_data;
}
```

**Description**
Structure containing data coefficients for individual warp engine

**Members**
- _mesh_entries,
- _fetch_entries – Number of mesh, filter and fetch coefficient entries in the data block. These values along with the corresponding sizeof() operator calculate the size of individual data block in bytes
- _filter_entries,
- _mesh_data,
_filter_data,
_fetch_data – Pointers to the blocks of raw mesh, filter and fetch data

mesh_entry_t, filter_entry_t, fetch_entry_t

Description Type aliases to use when calculating sizes of individual data block in bytes.

Helper functions

GetHwContext

WarpHwContextPtr
GetHwContext(intel_vvp_warp_channel_t* ch)

Description Generate object containing warp IP parameters necessary to generate data. Function uses warp channel driver structure to extract all necessary IP parameters and create object of the type WarpHwContext, required by the GenerateData() call

Arguments ch – valid pointer to initialized intel_vvp_warp_channel_t structure
object mesh_set – reference to a set of transformation meshes

Return value On success the function returns a valid smart pointer to a WarpHwContext object. On failure a nullptr object is returned

intel_vvp_warp_mesh

This reference software component allows you to generate sample warp meshes for common affine transformations, perspective transformations, radial distortion compensation and arbitrary transforms defined by a set of curves.
The `intel_vvp_warp_mesh` software instance provides three types of warp configurations which you can use independently by the host application:

**Figure 20. Fixed warp**

In the fixed warp configuration a set of affine transformations such as translation, rotation, scaling, perspective correction as well as radial distortion (fisheye) compensation are configured individually using corresponding API calls.

![Fixed warp diagram](image)

**Figure 21. 4x corner warp**

In the 4x corner configuration affine and perspective transformations are defined using four points positioned in or around the screen area; This configuration also allows to apply radial distortion compensation.

![4x corner warp diagram](image)
Figure 22. **Arbitrary warp**

Arbitrary transformation is defined by a set of curves controlled by a fixed number of control points (knots).

Warp meshes are tied to a specific input and output video resolution. The user must set these accordingly before generating a mesh.

Offset, sizes and positions are accepted as absolute values in pixels. To allow for sub-pixel precision floating point type is used. Internally these parameters normalized to the [0..1] range using current output resolution and kept in this form. The warp automatically fits into the new output resolution if it changes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetInputResolution, GetInputResolution</td>
<td>Set or get input image resolution</td>
</tr>
<tr>
<td>SetOutputResolution, GetOutputResolution</td>
<td>Set or get output image resolution</td>
</tr>
<tr>
<td>GenerateMeshFromFixed</td>
<td>Generate mesh using fixed warp configuration</td>
</tr>
<tr>
<td>GenerateMeshFromCorners</td>
<td>Generate mesh using 4x corner warp configuration</td>
</tr>
<tr>
<td>GenerateMeshFromArbitrary</td>
<td>Generate mesh using arbitrary warp configuration</td>
</tr>
<tr>
<td>SetHSize, GetHSize</td>
<td>Set or get horizontal size of the image</td>
</tr>
<tr>
<td>SetVSize, GetVSize</td>
<td>Set or get vertical size of the image</td>
</tr>
<tr>
<td>SetHOffset, GetHOffset</td>
<td>Set or get horizontal offset of the image</td>
</tr>
<tr>
<td>SetVOffset, GetVOffset</td>
<td>Set or get vertical offset of the image</td>
</tr>
<tr>
<td>SetRotate, GetRotate</td>
<td>Set or get rotation angle</td>
</tr>
<tr>
<td>SetHMirror, GetHMirror</td>
<td>Set or get horizontal mirroring</td>
</tr>
<tr>
<td>SetVMirror, GetVMirror</td>
<td>Set or get vertical mirroring</td>
</tr>
</tbody>
</table>

Table 79. **intel_vvp_warp_mesh API reference**

*continued...*
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetZoom, GetZoom</td>
<td>Set or get zoom factor</td>
</tr>
<tr>
<td>SetHKeystone, GetHKeystone</td>
<td>Set or get horizontal keystone angle</td>
</tr>
<tr>
<td>SetVKeystone, GetVKeystone</td>
<td>Set or get vertical keystone angle</td>
</tr>
<tr>
<td>SetFOV, GetFOV</td>
<td>Set or get field of view</td>
</tr>
<tr>
<td>SetVAxisOffset, GetVAxisOffset</td>
<td>Set or get vertical axis offset</td>
</tr>
</tbody>
</table>
| SetPreRadial, GetPreRadial              | Set or get radial distortion
|                                            | compensation parameters           |
| SetCorner, GetCorner                     | Set or get corner position         |
| SetArbitraryKnotsNum, GetArbitraryKnotsNum | Set or get size of the arbitrary control grid |
| SetArbitraryKnot, GetArbitraryKnot       | Set or get position of arbitrary control knot |
| SetMaintainRatio, GetMaintainRatio       | Set or get maintain aspect ratio setting |
| SetShrinkToFit, GetShrinkToFit           | Set or get shrink to fit setting   |

### SetInputResolution, GetInputResolution, SetOutputResolution, GetOutputResolution

```cpp
def SetInputResolution(const uint32_t width, const uint32_t height):
    def GetInputResolution(uint32_t& width, uint32_t& height):
    def SetOutputResolution(const uint32_t width, const uint32_t height):
    def GetOutputResolution(uint32_t& width, uint32_t& height):
```

**Description**

Set or get input and output image resolution

**Arguments**

- `width` – image width
- `height` – image height

**Return value**

None

### GenerateMeshFromFixed, GenerateMeshFromCorners, GenerateMeshFromArbitrary

```cpp
WarpMeshPtr GenerateMeshFromFixed()
WarpMeshPtr GenerateMeshFromCorners()
WarpMeshPtr GenerateMeshFromArbitrary()
```

**Description**

Generate transformation mesh using fixed, 4x corners or arbitrary configurations respectively

**Arguments**

None

**Return value**

Smart pointer to the object of the type WarpMesh containing generated transformation mesh (please see intel_vvp_data interface section for the description of WarpMesh)
SetHSize, GetHSize, SetVSize, GetVSize

```c
void SetHSize(float pixels) float GetHSize()
void SetVSize(float pixels) float GetVSize()
```

**Description**
Set or get horizontal and vertical size of the image, which is equivalent to scaling the image along the horizontal/vertical axis

**Arguments**
pixels – horizontal or vertical size of the image accordingly

**Return value**
horizontal or vertical size of the image accordingly

SetHOffset, GetHOffset, SetVOffset, GetVOffset

```c
void SetHOffset(float pixels) float GetHOffset()
void SetVOffset(float pixels) float GetVOffset()
```

**Description**
Set or get horizontal and vertical offsets of the image

**Arguments**
pixels – horizontal or vertical offset of the image accordingly

**Return value**
horizontal or vertical offset of the image accordingly

SetRotate, GetRotate

```c
void SetRotate(float angle) float GetRotate()
```

**Description**
Set or get rotation angle of the image in degrees. Image is rotated around the center

**Arguments**
angle – rotation angle in degrees

**Return value**
rotation angle in degrees

SetHMirror, GetHMirror, SetVMirror, GetVMirror

```c
void SetHMirror(bool v) bool GetHMirror()
void SetVMirror(bool v) bool GetVMirror()
```

**Description**
Set or get image mirroring along horizontal and vertical axis accordingly

**Arguments**
v – true to enable mirroring, false to disable

**Return value**
ture if mirroring enabled, false otherwise

SetZoom, GetZoom

```c
void SetZoom(float zoom) float GetZoom()
```
**Description**  
Set or get image zoom value

**Arguments**  
zoom – value > 1 to zoom in, < 1 to zoom out

**Return value**  
Configured zoom value

**SetHKeystone, GetHKeystone, SetVKeystone, GetVKeystone**

void SetHKeystone(float angle) float GetHKeystone()  
void SetVKeystone(float angle) float GetVKeystone()

**Description**  
Set or get horizontal and vertical keystone compensation angle accordingly

**Arguments**  
angle - horizontal or vertical angle in degrees by which the projector is tilting relative to the normal to the projection surface.

**Return value**  
Horizontal or vertical projector tilting angle accordingly

**SetFOV, GetFOV**

void SetFOV(float angle) float GetFOV()

**Description**  
Set or get field of View. Field of view represents projector’s beam angle. This value is necessary for correct keystone correction. The default value is 30°

**Arguments**  
angle - projector’s beam angle in degrees

**Return value**  
Projector’s beam angle in degrees

**SetVAxisOffset, GetVAxisOffset**

void SetVAxisOffset(float v) float GetVAxisOffset()

**Description**  
Set or get vertical offset of the optical axis from the normal to the projection plane. This value is required for correct keystone correction. Some projectors, particularly short throw models have their beam already at an angle to the wall. Take account of this initial angle when calculating keystone correction. The provided value is the absolute offset divided by the image height. The default value is 0

**Arguments**  
v - value of vertical axis offset

**Return value**  
value of vertical axis offset
**SetPreRadial, GetPreRadial**

```c
void SetPreRadial(float x, float y, float k1, float k2) void GetPreRadial(float& x, float& y, float& k1, float& k2)
```

**Description**
Set or get radial distortion compensation parameters. The software uses Brown-Conrady distortion model

**Arguments**
- `x, y` – distortion center
- `k1, k2` – 1st and 2nd distortion coefficients

**Return value**
None

**SetCorner, GetCorner**

```c
void SetCorner(const ECornerRadius& corner, float pixel_x, float pixel_y) void GetCorner(const ECornerRadius& corner, float& pixel_x, float& pixel_y)
```

**Description**
Set or get position of an individual corner point of the 4x corner warp configuration

**Arguments**
- `corner` – corner identifier
- `pixel_x, pixel_y` – horizontal and vertical coordinate of the corner

**Return value**
None

**SetArbitraryKnotsNum, GetArbitraryKnotsNum**

```c
void SetArbitraryKnotsNum(uint32_t num) uint32_t GetArbitraryKnotsNum()
```

**Description**
Set or get number of control points (knots) for the arbitrary warp. Arbitrary warp is defined by a set of curves and control points arranged into NxN grid. This parameter sets the size of this grid. For simplicity the number of control points in both dimensions is the same

**Arguments**
- `num` – number of control knots in each dimension

**Return value**
number of control knots in each dimension

**SetArbitraryKnot, GetArbitraryKnot**

```c
void SetArbitraryKnot(const uint32_t idx, const float x, const float y) void GetArbitraryKnot(const uint32_t idx, float& x, float& y)
```


Set or get absolute position of individual control points. Points are numbered left to right, top to bottom

**Arguments**
idx – zero based point index number
x, y – horizontal and vertical coordinate of the point

**Return value** None

**SetMaintainRatio, GetMaintainRatio**

```c
void SetMaintainRatio(bool val) bool GetMaintainRatio()
```

**Description**
Set or get maintain aspect ratio flag. If set the aspect ratio of the input resolution is preserved. As a result the output image may have blank areas along either horizontal or vertical axis

**Arguments**
val – true to maintain input aspect ratio, false otherwise

**Return value** Current maintain aspect ratio setting

**SetShrinkToFit, GetShrinkToFit**

```c
void SetShrinkToFit(bool val) bool GetShrinkToFit()
```

**Description**
Set or get shrink to fit setting. If set the resulting image is scaled down after applying all transforms to make sure it fits into the screen. Otherwise the resulting image is cropped to fit output dimensions. This flag is only relevant for the Fixed warp configuration

**Arguments**
val – true to shrink resulting image to fit output dimensions, false otherwise

**Return value** Current shrink to fit setting

**Data structures and types**

**ECornerId**

```c
enum ECornerId { ETopLeft = 0, ETopRight, EBottomLeft, EBottomRight, vfc32ETotalCorners, };
```

**Description**
Enum identifies individual corners in the 4x corner warp configuration
Using Warp IP Software

1. Initialize `intel_vvp_warp` driver instance
2. Allocate warp video channel
3. Configure the channel by providing video resolution, color space, base address of the framebuffer region in RAM and other required parameters
4. Instantiate `WarpConfigurer` object (`intel_vvp_warp_mesh`), set input, output video resolutions and desired transforms
5. Generate transformation mesh
6. Instantiate `WarpDataGenerator` object (`intel_vvp_warp_data`)
7. Instantiate `WarpDataContext` object (`intel_vvp_warp_data`) and fill it in using input, output video resolution and required IP parameters available through the warp channel structure
8. Generate mesh, data and filter coefficients by calling `WarpDataGenerator::GenerateData()` method and passing the data context object and the transformation mesh
9. Transfer generated coefficients to the designated RAM region accessible by the warp IP
10. Instantiate and fill in an `intel_vvp_warp_data_t` structure object using the addresses of the coefficient data in RAM and other required parameter
11. Apply new warp by calling `intel_vvp_warp_apply_transform()` method of the driver passing the warp channel and warp data structures as parameters.

Warp IP Software Code Examples

### UHD 60 Hz Workflow example

This example shows the workflow and basic warp software usage of the C++ source code to generate and apply 15 degree rotation warp. The example is for 3840x2160@60Hz video, which requires the processing to be split between two warp engines. The framebuffer and warp coefficient base addresses in the example are arbitrary. Actual values depend on your particular system design.

```c
const uint32_t FRAMEBUF_BASE_ADDR = 0x80000000;
const uint32_t COEF_BASE_ADDR = 0xa0000000;
intel_vvp_warp_base_t base = INTEL_VVP_WARP_BASE;
intel_vvp_warp_instance_t wrp0;

// Warp data sizes should be multiples of 256kb
auto align_256k = [](const uint32_t addr)->uint32_t { 
    static const uint32_t DATA_SIZE_256KB = (256 * 1024);
    return ((addr + DATA_SIZE_256KB - 1) & ~(DATA_SIZE_256KB - 1));
};

// Initialize driver instance
intel_vvp_warp_init_instance(&wrp0, base);
assert(wrp0.num_engines > 1);
intel_vvp_warp_channel_t* ch0 = intel_vvp_warp_create_double_channel(&wrp0, 0, 0, 1, 0);
```
// Fill in warp channel configuration structure
intel_vvp_warp_channel_config_t cfg;

// Configure warp channel using the parameters above
intel_vvp_warp_configure_channel(ch0, &cfg);

// Instantiate and initialize mesh generator
WarpConfigurator configurator;
configurator.SetInputResolution(3840, 2160);
configurator.SetOutputResolution(3840, 2160);
configurator.Reset();
configurator.SetRotate(15.0f);

// Instantiate data generator
WarpDataGenerator data_generator;

// Obtain required hardware information
WarpHwContextPtr hw = WarpDataHelper::GetHwContext(ch0);

WarpDataContext ctx{
    hw,
    3840, 2160,
    3840, 2160
};

// Generate warp data using provided hardware configuration and mesh
WarpDataPtr user_data = data_generator.GetData(ctx, mesh_set);

assert(user_data->_engines > 1);

const uint32_t warp_data_size = sizeof(intel_vvp_warp_data_t) + user_data->_engines * sizeof(intel_vvp_warp_engine_data_t);
const intel_vvp_warp_data_t* warp_data = (intel_vvp_warp_data_t*)malloc(warp_data_size);

// processing is split between two engines
// 1st engine processes left half of the frame
{
    engine_data[0].start_h = 0;
    engine_data[0].start_v = 0;
    engine_data[0].end_h = ctx._engine_hblocks - 1;
    engine_data[0].end_v = ctx._vblocks_out - 1;
    engine_data[0].mesh_stride = mesh_stride;

    const uint32_t mesh_data_size = user_data->_engine_data[0]->_mesh_entries * sizeof(mesh_entry_t);
    const uint32_t filter_data_size = user_data->_engine_data[0]->_filter_entries * sizeof(filter_entry_t);
    const uint32_t fetch_data_size = user_data->_engine_data[0]->_fetch_entries * sizeof(fetch_entry_t);
// Point engine to the location of the mesh, filter and fetch data
engine_data[0].mesh_addr = COEF_BASE_ADDR;
engine_data[0].filter_addr = engine_data[0].mesh_addr +
align_256k(mesh_data_size);
engine_data[0].fetch_addr = engine_data[0].filter_addr +
align_256k(filter_data_size);

// Transfer generated warp data to the calculated destination
memcpy((void*) (engine_data[0].mesh_addr), user_data->engine_data[0]->_mesh_data,
(mesh_data_size);
memcpy((void*) (engine_data[0].filter_addr), user_data->engine_data[0]->_filter_data,
(filter_data_size);
memcpy((void*) (engine_data[0].fetch_addr), user_data->engine_data[0]->_fetch_data,
(fetch_data_size);
}

// 2nd engine - right half of the frame
{
    engine_data[1].start_h = ctx._engine_hblocks;
    engine_data[1].start_v = 0;
    engine_data[1].end_h = ctx._hblocks_out - 1;
    engine_data[1].end_v = ctx._vblocks_out - 1;
    engine_data[1].mesh_stride = mesh_stride;

    const uint32_t mesh_data_size = user_data->engine_data[1]->_mesh_entries *
    sizeof(mesh_entry_t);
    const uint32_t filter_data_size = user_data->engine_data[1]->_filter_entries *
    sizeof(filter_entry_t);
    const uint32_t fetch_data_size = user_data->engine_data[1]->_fetch_entries *
    sizeof(fetch_entry_t);

    engine_data[1].mesh_addr = engine_data[0].fetch_addr + align_256k(user_data->
    engine_data[0]->_fetch_entries * sizeof(fetch_entry_t));
    engine_data[1].filter_addr = engine_data[1].mesh_addr +
align_256k(mesh_data_size);
    engine_data[1].fetch_addr = engine_data[1].filter_addr +
align_256k(filter_data_size);

    // Transfer generated warp data to the calculated destination
    memcpy((void*) (engine_data[1].mesh_addr), user_data->engine_data[1]->_mesh_data,
    mesh_data_size);
    memcpy((void*) (engine_data[1].filter_addr), user_data->engine_data[1]->_filter_data,
    filter_data_size);
    memcpy((void*) (engine_data[1].fetch_addr), user_data->engine_data[1]->_fetch_data,
    fetch_data_size);
}

// 2nd engine - right half of the frame
{
    engine_data[1].start_h = ctx._engine_hblocks;
    engine_data[1].start_v = 0;
    engine_data[1].end_h = ctx._hblocks_out - 1;
    engine_data[1].end_v = ctx._vblocks_out - 1;
    engine_data[1].mesh_stride = mesh_stride;

    const uint32_t mesh_data_size = user_data->engine_data[1]->_mesh_entries *
    sizeof(mesh_entry_t);
    const uint32_t filter_data_size = user_data->engine_data[1]->_filter_entries *
    sizeof(filter_entry_t);
    const uint32_t fetch_data_size = user_data->engine_data[1]->_fetch_entries *
    sizeof(fetch_entry_t);

    engine_data[1].mesh_addr = engine_data[0].fetch_addr + align_256k(user_data->
    engine_data[0]->_fetch_entries * sizeof(fetch_entry_t));
    engine_data[1].filter_addr = engine_data[1].mesh_addr +
align_256k(mesh_data_size);
    engine_data[1].fetch_addr = engine_data[1].filter_addr +
align_256k(filter_data_size);

    // Transfer generated warp data to the calculated destination
    memcpy((void*) (engine_data[1].mesh_addr), user_data->engine_data[1]->_mesh_data,
    mesh_data_size);
    memcpy((void*) (engine_data[1].filter_addr), user_data->engine_data[1]->_filter_data,
    filter_data_size);
    memcpy((void*) (engine_data[1].fetch_addr), user_data->engine_data[1]->_fetch_data,
    fetch_data_size);
}

warp_data->_skip_megablock_data = user_data->_skip_megablock_data;
warp_data->_skip_ram_page = 0;

// Apply warp by passing new warp data set to the driver
intel_vvp_warp_apply_transform(ch0, warp_data);

// Release allocated resources
free(warp_data);
intel_vvp_warp_free_channel(ch0);

Full HD up to UHD@30 Hz warp channel allocation

Input video streams in full HD and up to UHD@30Hz formats require a single warp
engine for processing. The example shows how to allocate a warp channel for such use
cases:

intel_vvp_warp_base_t base = INTEL_VVP_WARP_BASE;
intel_vvp_warp_instance_t wrp0;

// Initialize driver instance
intel_vvp_warp_init_instance(&wrp0, base);
assert(wrp0.num_engines > 0);

uint32_t input = 0, output = 0;
uint32_t engine = 0;

intel_vvp_warp_channel_t* ch0 = intel_vvp_warp_create_channel(&wrp0, input, engine, output);

// Application code here
//
intel_vvp_warp_free_channel(ch0);

**Warp mesh usage**

Define required warp using the WarpMesh object. The example shows the simplest case of 1:1 (unity) warp for a 3840x2160 video.

```cpp
intel_vvp_warp::WarpMesh mesh{3840, 2160};
for(uint32_t v = 0; v < mesh.GetVNodes(); ++v)
{
    mesh_node_t* node = mesh.GetRow(v);
    for(uint32_t h = 0; h < mesh.GetHNodes(); ++h)
    {
        node->_x = (h * mesh.GetStep()) << 4;
        node->_y = (v * mesh.GetStep()) << 4;
    }
}
```

Mesh coordinates use the least significant four bits as fractional part for subpixel precision. In the example above the fractional part is always 0. Store subpixel positions in the following way:

```cpp
mesh_node_t* node = mesh.GetRow(v);
...
float pos_x = 10.6f;
node->_x = static_cast<int32_t>(roundf(pos_x * 16.0f));
```
# Document Revision History for Video and Vision Processing Suite User Guide

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2021.09.10</td>
<td>21.2</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.