1. Design Example Overview

The following table summarizes the configurations to be supported by the P-Tile Avalon®-MM design examples:

<table>
<thead>
<tr>
<th></th>
<th>Gen3/Gen4 x16</th>
<th>Gen3/Gen4 x8</th>
<th>Gen3/Gen4 x4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endpoint (EP)</td>
<td>Yes (1)</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Root Port (RP)</td>
<td>(2)</td>
<td>N/A</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note: Gen1/Gen2 x1/x2 configurations are supported via link down-training.

Note: N/A = configuration not supported.

(1) In the available design example, the only active blocks within the P-Tile Avalon-MM IP for PCIe are the Data Movers.

(2) A design example supporting these configurations may be available in a future release of Intel® Quartus® Prime.
2. Endpoint Design Example

This DMA design example includes a DMA Controller and an on-chip memory to exercise the Data Movers.

The design example also connects the Bursting Master (in non-bursting mode) to the on-chip memory to allow high-throughput transfers should the host or some other component of the PCIe system be capable of initiating such transfers (e.g. a Root Complex with a DMA engine).

The on-chip memory that the Data Movers and the Bursting Master connect to is a dual-port memory to allow full-duplex data movement.

The Bursting Master connects to a BAR Interpreter module, which combines the address and BAR number and allows the Bursting Master to control the DMA Controller. The BAR Interpreter also connects the Bursting Master to the dual-port memory.

The following table shows the address and BAR mapping that the BAR Interpreter in this design example uses:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Address Range</th>
<th>BAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA</td>
<td>0x0 - 0xFFFF</td>
<td>0</td>
</tr>
<tr>
<td>MEM0.s1</td>
<td>0x20000 - 0x27FFF</td>
<td>2</td>
</tr>
<tr>
<td>MEM0.s2</td>
<td>0x28000 - 0x2FFFF</td>
<td>4</td>
</tr>
</tbody>
</table>

The design example is generated dynamically based on the selected variation of the P-Tile Avalon-MM IP for PCIe. However, some of the user’s parameter selections may need to be overwritten to ensure proper functionality. A warning appears when such a need arises.

*Other names and brands may be claimed as the property of others.*
In the 20.2 release of Intel Quartus Prime, the only variation supported is the DMA variation. This variation instantiates the Bursting Master (in non-bursting mode), Read Data Mover and Write Data Mover. Software sends instructions via the Bursting Master to the Read or Write Data Movers to initiate DMA Reads or Writes to the system memory. The BAR Interpreter, on-chip memory and DMA Controller are also included.

Note: Beginning with the 17.1 release, the Intel Quartus Prime Pro Edition software dynamically generates design examples for the parameters you specify in the parameter editor. Consequently, the Intel Quartus Prime Pro Edition installation directory no longer provides static design examples for Intel Stratix® 10 devices. Static design examples are available for earlier device families, including Intel Arria® 10 and Intel Cyclone® 10 devices.

2.1. Block Descriptions

The DMA design example for the P-Tile Avalon-MM IP for PCIe includes the following components:

- DUT: The P-Tile Avalon-MM IP for PCIe Endpoint.
- MEM0: An on-chip dual-port memory that connects to the Read Data Mover and Write Data Mover interfaces of the DUT.
- DMA_CONTROLLER: A DMA Controller that interfaces with the normal and priority descriptor queues of the DUT's Read Data Mover and Write Data Mover.
- BAR_INTERPRETER: A BAR Interpreter that combines the address and BAR number to form a wider address that Platform Designer can use to route memory transactions to the various slaves. The BAR Interpreter connects the Bursting Master of the DUT to the dual-port memory.
- Reset Release IP: This IP holds the control circuit in reset until the device has fully entered user mode. The FPGA asserts the INIT_DONE output to signal that the device is in user mode. The Reset Release IP generates an inverted version of the internal INIT_DONE signal to create the nINIT_DONE output that you can use for your design.

The nINIT_DONE signal is high until the entire device enters user mode. After nINIT_DONE asserts (low), all logic is in user mode and operates normally. You can use the nINIT_DONE signal in one of the following ways:

- To gate an external or internal reset.
- To gate the reset input to the transceiver and I/O PLLs.
- To gate the write enable of design blocks such as embedded memory blocks, state machine, and shift registers.
- To synchronously drive register reset input ports in your design.
2. Endpoint Design Example

Figure 2. Platform Designer View of the x16 Endpoint DMA Design Example for the P-Tile Avalon-MM IP for PCIe

Figure 3. Platform Designer View of the x8 Endpoint DMA Design Example for the P-Tile Avalon-MM IP for PCIe

Note: Only Port 0 is used in the x8 design example.

Note: For hardware testing purpose, plug the x8 design example into a x8 slot.
Table 3. System Address Map

<table>
<thead>
<tr>
<th>Slave</th>
<th>BAR_INTERPRETER.bri_master</th>
<th>dut.p0_wrdm_master</th>
<th>dut.p0_rddm_master</th>
<th>dut.p0_bam_master</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAR_INTERPRETER.bri_slave</td>
<td>0x0000_0000_0000_0000 - 0x0000_0000_0000_0000</td>
<td>0x0000_0000_ff</td>
<td>0x0000_0000_0000_0000_ff</td>
<td></td>
</tr>
<tr>
<td>DMA_CONTROLLER dma_slave</td>
<td>0x0000_0000_0000_0000 - 0x0000_0000_0000_0000</td>
<td>0x0000_0000_0000_0000</td>
<td>0x0000_0000_0000_0000</td>
<td></td>
</tr>
<tr>
<td>MEM0.s1</td>
<td>0x0002_0000_0000_0000 - 0x0002_0000_0000_0000</td>
<td>0x0000_0000_0000_0000</td>
<td>0x0000_0000_0000_0000</td>
<td></td>
</tr>
<tr>
<td>MEM0.s2</td>
<td>0x0002_8000_0000_0000 - 0x0002_8000_0000_0000</td>
<td>0x0000_0000_0000_0000</td>
<td>0x0000_0000_0000_0000</td>
<td></td>
</tr>
</tbody>
</table>

2.1.1. DMA Controller

The DMA Controller in this example design consists of six addressable queues: two write-only queues and one read-only queue each for the Read Data Mover and the Write Data Mover. In addition, the DMA Controller has two MSI control registers for each Data Mover module.

The write-only queues directly feed into the Data Movers’ normal and priority descriptor queues. The read-only queues read directly from the Data Movers’ status queues.

The MSI control registers control whether MSI generation is enabled and defines the address and data to be used for the MSI.

The example design uses p<n>_app_clk generated from the coreclkout_hip clock.

Note: The P-Tile Avalon-MM IP core does not include an internal DMA Controller. You can use the DMA Controller included in the example design that you can generate, or provide your own DMA Controller.

2.1.1.1. Register Set

The registers in the DMA Controller are 512-bit wide to match the data path width of the Bursting Master’s and Read Data Mover’s Avalon-MM Master. This allows the Read Data Mover to write a descriptor in a single cycle if desired.

Table 4. Register Set of the DMA Controller

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>WDN</td>
<td>R/W</td>
<td>write: descriptor for the Write Data Mover normal descriptor queue</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x200</td>
<td>WDP</td>
<td>R/W</td>
<td>read: readiness and fill level of the Write Data Mover normal descriptor queue</td>
</tr>
<tr>
<td>0x400</td>
<td>WS</td>
<td>RO</td>
<td>Write Data Mover status queue</td>
</tr>
<tr>
<td>0x600</td>
<td>WI</td>
<td>R/W</td>
<td>Write Data Mover interrupt control register</td>
</tr>
<tr>
<td>0x800</td>
<td>RDN</td>
<td>R/W</td>
<td>write: descriptor for the Read Data Mover normal descriptor queue</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read: readiness and fill level of the Read Data Mover normal descriptor queue</td>
</tr>
<tr>
<td>0xA00</td>
<td>RDP</td>
<td>R/W</td>
<td>write: descriptor for the Read Data Mover priority descriptor queue</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read: readiness and fill level of the Read Data Mover priority descriptor queue</td>
</tr>
<tr>
<td>0xC00</td>
<td>RS</td>
<td>RO</td>
<td>Read Data Mover status queue</td>
</tr>
<tr>
<td>0xE00</td>
<td>RI</td>
<td>R/W</td>
<td>Read Data Mover interrupt control register</td>
</tr>
</tbody>
</table>

For the data written to the descriptor queue registers, use the same format and content as the data on the corresponding Avalon-ST interfaces of the Data Movers. The least significant of the application specific bits indicates whether an interrupt should be issued when processing of that descriptor completes. The data is written to the least significant 174 bits of the registers because the descriptors are 174-bit wide (refer to Table 6 on page 11 for the descriptor format).

The DMA Controller double buffers the write-only queues so that the descriptors can be built one DWORD at a time if required, for example by a 32-bit host controller. The content of the register is transferred to the Data Movers’ Avalon-ST input when the most significant DWORD is written.

Attempting to write to a descriptor queue when the corresponding Data Mover’s ready signal is not asserted causes the DMA Controller to assert its waitrequest signal until ready is asserted. You must make sure the Read Data Mover does not attempt to write to the same queue that it is processing while the queue is full, as that would lead to a deadlock. For more details on deadlocks, refer to the section Deadlock Risk and Avoidance.

You can find the status of the ready signal of a descriptor queue interface by checking the ready bit (bit [31]) of the queue registers. In addition, bits [7:0] of the queue registers indicate the approximate fill level of the queues. The other bits of the queue registers are set to 0.

Only the least significant DWORD of the WS and RS registers contains significant information. The other bits are set to 0.

The format and content of the status queues are identical to the corresponding Avalon-ST interfaces of the Data Movers with the addition of bit 31 indicating that the queue is empty. Reading from one of the status queues when it is empty returns 512’h8000_0000.
The format of the WI and RI interrupt control registers is as follows: \{enable, priority, reserved[414:0], msi_msg_data[15:0], reserved[15:0], msi_address[63:0]\}.

The \textit{enable} bit controls whether or not an MSI is sent. The \textit{priority} bit specifies whether to use the priority queue to send the MSI. The MSI memory write TLP also uses the contents of the \textit{msi_msg_data} and \textit{msi_address} fields.

\subsection{Deadlock Risk and Avoidance}

Under certain circumstances, it is possible for the DMA engine in the design example hardware to get into a deadlock. This section describes the conditions that may lead to a deadlock, and how to avoid them.

When you program the DMA Controller to use the Read Data Mover to fetch too many descriptors for the Read Data Mover descriptor queue, the following loop of backpressure that leads to a deadlock can occur.

Once the Read Data Mover has transferred enough descriptors through the DMA Controller to its own descriptor queue to fill up the queue, it deasserts its ready output. The DMA Controller in turn asserts its waitrequest output, thus preventing the Read Data Mover from writing any remaining descriptor to its own queue. After this situation occurs, the Read Data Mover continues to issue MRd read requests, but because the completions can no longer be written to the DMA Controller, the tags associated with these MRd TLPs are not released. The Read Data Mover eventually runs out of tags and stops, having gotten into a deadlock situation.

To avoid this deadlock situation, you can limit the number of descriptors that are fetched at a time. Doing so ensures that the Read Data Mover’s descriptor queue never fills up when it is trying to write to its own descriptor queue.

\textit{Note:} Due to this risk, the design example has a limit of 128 descriptors to avoid this deadlock.

\subsection{Interrupts}

Two application specific bits (bits [13:12]) of the status words from the Write Data Mover and Read Data Mover Status Avalon-ST Source interfaces control when interrupts are generated.

\textbf{Table 5. Interrupts Control}

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Interrupt always</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Interrupt if error</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>No interrupt</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>No interrupt and drop status word (i.e, do not even write it to the WS or RS status queues)</td>
</tr>
</tbody>
</table>

The DMA Controller makes the decision whether to drop the status word and whether to generate an interrupt as soon as it receives the status word from the Data Mover. When generation of an interrupt is requested, and the corresponding RI or WI register does enable interrupts, the DMA Controller generates the interrupt. It does so by queuing an immediate write to the Write Data Mover’s descriptor queue specified in
the corresponding interrupt control register using the MSI address and message data provided in that register. You need to make sure that space is always available in the targeted Write Data Mover descriptor queue at any time when an interrupt may get generated. You can do so most easily by using the priority queue only for MSIs.

Setting the interrupt control bits in the immediate write descriptors that the DMA Controller creates to generate MSI interrupts to "No interrupt and drop status word" can avoid an infinite loop of interrupts.

2.1.1.4. Using the DMA Controller

To initiate a single DMA transfer, you only need to write a well-formed descriptor to one of the DMA Controller's descriptor queues (WDN, WDP, RDN or RDP).

To initiate a series of DMA transfers, you can prepare a table of descriptors padded to 512 bits each in a memory location accessible to the Read Data Mover. You can then write a single descriptor to the DMA Controller's priority descriptor queue (RDP) register to initiate the DMA transfers. These transfers move the descriptors from the source location in PCIe memory to the desired descriptor queue register.

To transmit an MSI interrupt upon completion of the processing of a descriptor, you must program the DMA Controller's WI or RI register with the desired MSI address and message before writing the descriptor.

2.1.2. Avalon-MM Address to PCIe Address Mapping

The Bursting Slave module transforms read and write transactions on its Avalon-MM interface into PCIe memory read (MRd) and memory write (MWr) request packets. The Bursting Slave uses the Avalon-MM address provided on its 64-bit wide address bus directly as the PCIe address in the TLPs that it creates.

The Bursting Slave, with its 64-bit address bus, uses up the whole Avalon-MM address space and prevents other slaves from being connected to the same bus. In many cases, the user application only needs to access a few relatively small regions of the PCIe address space, and would prefer to dedicate a smaller address space to the Bursting Slave to be able to connect to other slaves.

2.1.3. BAR Interpreter

The Bursting Master module transforms PCIe memory read and write request packets received from the PCIe system into Avalon-MM read and write transactions. The offset from the matching BAR is provided as the Avalon-MM address, and the number of the matching BAR is provided in a conduit synchronously with the address.

Although these signals are in a conduit separate from the Avalon-MM master interface, they are synchronous to it and can be treated as extensions of the address bus.

The BAR Interpreter simply concatenates the BAR number to the address bus to form a wider address bus that Platform Designer can now treat as a normal address bus and route to the various slaves connected to the BAR Interpreter.

2.2. Programming Model for the Design Example

The programming model for the DMA example design performs the following steps:
1. In system memory, prepare a contiguous set of descriptors. The last of these descriptors is an immediate write descriptor, with the destination address set to some special system memory status location. The descriptor table must start on a 64-byte aligned address. Even though each descriptor is only about 174-bit long, 512 bits are reserved for each descriptor. The descriptors are LSB-aligned in that 512-bit field.

2. In system memory, prepare one more descriptor which reads from the beginning of the descriptors from Step 1 and writes them to a special FIFO Avalon-MM address in FPGA.

3. Write the descriptor in Step 2 to the same special FIFO Avalon-MM address by:
   a. Writing one dword at a time, ending with the most significant dword.
   b. Writing three dwords of padding and the entire descriptor for a total of eight dwords (the descriptor takes up only five dwords, but CPUs do not typically support single-TLP, five-dword writes).

4. Poll the special status location in system memory to see if the final immediate write has occurred, indicating the DMA completion.

2.3. Descriptor Format for the Design Example

The Read and Write Data Movers uses descriptors to transfer data. The descriptor format is fixed and specified below:

Table 6. Descriptor Format for Data Movers

<table>
<thead>
<tr>
<th>Signals Description (for rddm_desc_data_i or wrdm_desc_data_i)</th>
<th>Read Data Mover</th>
<th>Write Data Mover</th>
</tr>
</thead>
<tbody>
<tr>
<td>[173:160]: reserved</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>[159:152]: descriptor ID</td>
<td>ID of the descriptor</td>
<td>ID of the descriptor</td>
</tr>
<tr>
<td>[151:149]: application-specific</td>
<td>Application-specific bits. Example of an Intel application is provided below.</td>
<td>Application-specific bits. Example of an Intel application is provided below.</td>
</tr>
<tr>
<td>[148]: single destination</td>
<td>When the single destination bit is set, the same destination address is used for all the transfers. If the bit is not set, the address increments for each transfer.</td>
<td>N/A</td>
</tr>
<tr>
<td>[147]: single source</td>
<td>N/A</td>
<td>When the single source bit is set, the same source address is used for all the transfers. If the bit is not set, the address increments for each transfer. Note that in single source mode, the PCIe address and Avalon-MM address must be 64-byte aligned.</td>
</tr>
<tr>
<td>[146]: immediate</td>
<td>N/A</td>
<td>When set, the immediate bit indicates immediate writes. Immediate writes of one or two dwords are supported. For immediate transfers, bits [31:0] or [63:0] contain the payload for one- or two-dword transfers respectively. The two-dword immediate writes cannot cross a 4k boundary. This can be used for MSI/MSI-X for example.</td>
</tr>
</tbody>
</table>

continued...
### Application-Specific Bits

Three application-specific bits (bits [151:149]) from the Write Data Mover and Read Data Mover Status Avalon-ST Source interfaces control when interrupts are generated.

#### Table 7. Encodings for Application-Specific Bits

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Interrupt always</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Interrupt if error</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>No interrupt</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No interrupt and drop status word</td>
</tr>
</tbody>
</table>

The External DMA Controller makes the decision whether to drop the status word and whether to generate an interrupt as soon as it receives the status word from the Data Mover. When the generation of an interrupt is requested, and the corresponding RI or WI register does enable interrupts, the DMA Controller generates the interrupt. It does so by queuing an immediate write to the Write Data Mover’s descriptor queue (specified in the corresponding interrupt control register) using the MSI address and message data provided in that register.

### 2.4. DMA Operations Using the Design Example

#### 2.4.1. Read DMA Example

A Read DMA transfers data from the PCIe address space (system memory) to the Avalon-MM address space. It sends Memory Read TLPs upstream, and writes the completion data to local memory in the Avalon-MM address space using the Read Data Mover’s Avalon-MM write master interface.

The sequence of steps the example design follows to do a Read DMA is:

1. Prepare a table of descriptors (padded to 512-bit each) to perform the Read operation and put the table into the system memory.
2. Using the BAM, send one descriptor from software containing the address of the descriptor table to the DMA Controller, which forwards it to the Read Data Mover.
3. The Read Data Mover fetches the descriptor table and puts it in a FIFO inside the DMA Controller.
4. The DMA Controller outputs these descriptors to the Read Data Mover based on the readiness of the Read Data Mover (indicated by an asserted rddm_desc_ready_o or rddm_prio_ready_o signal).
5. The Read Data Mover processes the descriptors by fetching data from the system memory, and writing it to the appropriate Avalon-MM memory.

6. The last descriptor processed by the Read Data Mover points to an immediate write descriptor (i.e., a descriptor where the data to be written is inside the descriptor itself) in the system memory. This descriptor’s destination address is the Avalon memory address of the DMA Controller’s Write Data Mover port. The Read Data Mover fetches this descriptor from system memory and transfers it to the DMA Controller’s Write Data Mover Avalon address.

7. The Write Data Mover uses the descriptor from Step 6 to perform an immediate write to the system memory indicating the completion of the Read Data Mover’s data processing.

2.4.2. Write DMA Example

A Write DMA transfers data from the Avalon-MM address space to the PCIe address space (system memory). It uses the Write Data Mover’s Avalon-MM read master to read data from the Avalon-MM address space and sends it upstream using Memory Write TLPs.

The sequence of steps the example design follows to do a Write DMA is:

1. Prepare a table of descriptors (padded to 512-bit each) to perform the Write operation and put the table into the system memory.

2. Using the BAM, send one descriptor from software containing the address of the descriptor table to the DMA Controller, which forwards it to the Read Data Mover.

3. The Read Data Mover fetches the descriptor table and puts it in a FIFO inside the DMA Controller.

4. The DMA Controller outputs these descriptors to the Write Data Mover based on the readiness of the Write Data Mover (indicated by an asserted wrdm_desc_ready_o or wrdm_prio_ready_o signal).

5. The Write Data Mover processes the descriptors by fetching data from the Avalon-MM memory, and writing it to the appropriate system memory.

6. The Write Data Mover uses the last descriptor in the descriptor table to indicate the completion of the Write Data Mover’s data processing. This descriptor is an Immediate Write (the data is inside the descriptor itself) to the system memory indicating the Write Data Mover’s operations are done.
3. Root Port Design Example

The P-Tile Avalon-MM IP for PCIe Root Port (RP) design example supports the following configurations:

- Gen3 x4 with x4 bifurcation Root Port
- Gen4 x4 with x4 bifurcation Root Port

The RP design example performs enumeration and configuration of the PCI Express hardware by using the RP Master system script running on System Console. The RP Master system script configures the RP itself through the P-Tile Hard IP Reconfiguration interface and discovers the PCI Express hardware by performing CfgRd and CfgWr transactions and BAR initialization via the Control Register Access (CRA) Avalon-MM Slave interface.

The RP design example automatically creates the files necessary to compile in the Intel Quartus Prime software. It also generates the RP Master system script (.tcl) which can run on System Console.

![Design Example for Root Port](image)

**Note:** The P-Tile Avalon-MM PCIe IP Root Port variant only uses Port 0. Therefore, the three interfaces enabled in the IP (shown in the figure above) are for Port 0 (P0) only.

3.1. Block Descriptions

The Root Port design example consists of two main blocks:
• DUT: This is the generated Intel P-Tile Avalon-MM PCIe IP Root Port variant configured with the parameters you specified. The DUT supports x4 bifurcation mode in the RP configuration. It has the following interfaces enabled (note that only Port 0 (P0) is utilized):
  — P0 Control Register Access (CRA) Avalon-MM Slave port.
  — P0 Avalon-MM Slave interface with individual byte access (TXS).
  — P0 Hard IP (HIP) dynamic reconfiguration interface of PCIe registers.

• RP_MASTER: This is the JTAG to Avalon Master Bridge Intel FPGA IP. It is a driver that converts the memory read/write commands from the RP Master system script running in System Console to Avalon-MM transactions that are sent to the DUT.

Here are the descriptions for the modules associated with the interfaces that are enabled for the DUT:

• Control Register Access (CRA) Avalon-MM Slave: This module is used to issue accesses to the Endpoint's configuration space registers. It supports a single transaction at a time. It converts single-cycle, 32-bit Avalon-MM read and write transactions into PCIe configuration read and write TLPs (CfgRd0, CfgRd1, CfgWr0 and CfgWr1) to be sent over the PCIe link.

• Non-bursting Avalon-MM (TXS) Slave: This module has a 32-bit wide data bus. It converts single-cycle, 32-bit Avalon-MM read and write transactions into PCIe memory read and write TLPs (MemRd, MemWr) to be sent over the PCIe link.

• Hard IP Reconfiguration Avalon-MM Slave: This interface has a 21-bit address bus and an 8-bit data bus. The application logic (RP Master) uses this interface to access its PCIe configuration space to perform link control functions (such as Hot Reset, link disable, or link retrain).
Table 8. Address Map for the Root Port Avalon-MM Components

<table>
<thead>
<tr>
<th>Components</th>
<th>Base Address</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard IP Reconfiguration interface</td>
<td>0x0000_0000</td>
<td>0x20_0000</td>
</tr>
<tr>
<td>CRA interface</td>
<td>0x2000_0000</td>
<td>0x8000</td>
</tr>
<tr>
<td>TXS interface</td>
<td>0x2100_0000</td>
<td>0x100_0000</td>
</tr>
</tbody>
</table>

For more details on the CRA interface register map, refer to the Programming Model for the Avalon-MM Root Port chapter of the Intel Stratix 10 H-Tile and L-Tile Avalon memory mapped Hard IP for PCI Express User Guide.

Related Information

Intel Stratix 10 H-Tile and L-Tile Avalon memory mapped Hard IP for PCI Express User Guide

In the 20.2 release of Intel Quartus Prime, the design examples for the P-Tile Avalon-MM IP for PCIe have the following limitations:

- The Endpoint DMA design example cannot handle 10-bit tags.
- To enable the Gen4 x16 Endpoint DMA design example to meet timing requirements at 350 MHz, you need to manually enable all pipelinable locations in the Platform Designer Interconnect fabric (`mm_interconnect`). Here are the steps to enable the `mm_interconnect` pipeline stages:
  1. Open the generated design example in Platform Designer.
  2. Click on System, then Show System with Platform Designer Interconnect.
  3. Click on Show Pipelinable Locations.
  4. Go through each `mm_interconnect_N` and enable all pipelinable registers for both Command and Response.
  5. Generate the HDL for the design example.
- Simulation is supported for the Endpoint design example in the 20.2 release of Intel Quartus Prime, but it is available for the VCS simulator only.
- Simulation is not supported for the Root Port design examples in the 20.2 release of Intel Quartus Prime, but will be supported in a future release.
5. Quick Start Guide

Using Intel Quartus Prime Pro Edition, you can generate a simple Endpoint (EP) DMA design example or a Root Port (RP) design example for the P-Tile Avalon-MM IP for PCI Express IP core.

The generated design example reflects the parameters that you specify. It automatically creates the files necessary to simulate and compile the design example in the Intel Quartus Prime Pro Edition software. You can download the compiled design example to the Intel Stratix 10 DX Development Board or Intel Agilex™ Development Board to do hardware testing. To download to custom hardware, update the Intel Quartus Prime Settings File (.qsf) with the correct pin assignments.

The RP design example transfers RP commands from the RP Master system script running in System Console to the Endpoint PCIe device. It supports link status checking between the RP and EP, RP configuration, EP enumeration and BAR initialization.

Note: Simulation is not supported for the RP design example in the 20.2 release of Intel Quartus Prime. It will be supported in a future release.

5.1. Design Components

The available design example is for an Endpoint with a single function. This DMA design example includes a DMA Controller and an on-chip memory to exercise the Data Movers in the P-Tile Avalon-MM IP for PCI Express.

Figure 6. Block Diagram for the Platform Designer Avalon-MM with DMA Design Example
5.2. Directory Structure

Figure 7. Directory Structure for the Generated Design Example

5.3. Generating the Design Example

Figure 8. Design Example Generation Procedure

1. In the Intel Quartus Prime Pro Edition software, create a new project (File → New Project Wizard).
2. Specify the Directory Name, and Top-Level Entity.
3. For Project Type, accept the default value, Empty project. Click Next.
4. For Add Files click Next.
5. For **Family, Device & Board Settings** under **Family**, select **Intel Agilex** or **Intel Stratix 10**.

6. If you select **Intel Stratix 10** in the last step, select **Stratix 10 DX** in the **Device** pull-down menu.

7. Select the **Target Device** for your design.

8. Click **Finish**.

9. In the IP Catalog locate and add the **Intel P-Tile Avalon-MM IP for PCI Express**.

10. In the **New IP Variant** dialog box, specify a name for your IP. Click **Create**.

11. On the **Top-Level Settings** and **PCIE* Settings** tabs, specify the parameters for your IP variation. For example, select **Root Port** for the RP variant.

12. On the **Example Designs** tab, make the following selections:
   a. For **Example Design Files**, turn on the **Synthesis** option. If you do not need these synthesis files, leaving the corresponding option turned off significantly reduces the example design generation time.

   **Note:** Simulation is not supported for the RP design example in the 20.2 release of Intel Quartus Prime. It will be supported in a future release. Therefore, if you are working with the RP design example, turn on the **Synthesis** option only. Do not turn on the **Simulation** option.

   b. For **Generated HDL Format**, only Verilog is available in the current release.

   c. For **Target Development Kit**, select the appropriate option. For the current release, the supported development kits are:
      - **Intel Stratix 10 DX P-Tile ES1 FPGA Development Kit**
      - **Intel Agilex F-Series P-Tile ES0 FPGA Development Kit**

13. Select **Generate Example Design** to create a design example that you can compile and download to hardware. If you select one of the P-Tile development boards, the device on that board overwrites the device previously selected in the Intel Quartus Prime project if the devices are different. When the prompt asks you to specify the directory for your example design, you can accept the default directory, `<project_dir>/intel_pcie_ptile_avmm_0_example_design`, or choose another directory.
14. Click **Finish**. You may save your .ip file when prompted, but it is not required to be able to use the example design.

15. Open the example design project.

16. Compile the example design project to generate the .sof file for the complete example design. This file is what you download to a board to perform hardware verification.

17. Close your example design project.

*Note:* You cannot change the PCIe pin allocations in the Intel Quartus Prime project. However, to ease PCB routing, you can take advantage of the lane reversal and polarity inversion features supported by this IP.

### 5.4. Simulating the Design Example

**Figure 10. Procedure**

1. Change to the testbench simulation directory, `intel_pcie_ptile_avmm_0_example_design\pcie_ed_tb`.

2. Run the simulation script for VCS. Refer to the table below.

3. Analyze the results.

*Note:* **P-Tile does not support parallel PIPE simulations.**
Note: Simulation is not supported for the Gen3/Gen4 x4 Root Port design example for Intel Agilex and Intel Stratix 10 DX in the 20.2 release of Intel Quartus Prime.

Table 9. Steps to Run Simulation

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Working Directory</th>
<th>Instructions</th>
</tr>
</thead>
</table>
| VCS*      | `<example_design>/pcie_ed_tb/pcie_ed_tb/sim/synopsys/vcs` | 1. sh vcs_setup.sh  
USER_DEFINED_COMPILE_OPTIONS=""
USER_DEFINED_ELAB_OPTIONS="-xlrm\uniq_prior_final"  
USER_DEFINED_SIM_OPTIONS=""
2. A successful simulation ends with the following message, "Simulation stopped due to successful completion!"

Note: To run a simulation in interactive mode, use the following steps: (if you already generated a simv executable in non-interactive mode, delete the simv and simv.dadiir)
   1. Open the vcs_setup.sh file and add a debug option to the VCS command: vcs -debug_access+r
   2. Compile the design example: sh vcs_setup.sh  
USER_DEFINED_ELAB_OPTIONS="-xlrm\uniq_prior_final"  
SKIP_SIM=1
3. Start the simulation in interactive mode: simv -gui & |

The DMA testbench completes the following tasks:
1. Writes to the Endpoint memory using the DUT Endpoint non-bursting Avalon-MM master interface.
2. Reads from Endpoint memory using the DUT Endpoint non-bursting Avalon-MM master interface.
3. Verifies the data using the `shmem_chk_ok` task.
4. Writes to the Endpoint DMA controller, instructing the DMA controller to perform a MRd request to the PCIe* address space in host memory.
5. Writes to the Endpoint DMA controller, instructing the DMA controller to perform a MWr request to PCIe address space in host memory. This MWr uses the data from the previous MRd.
6. Verifies the data using the `shmem_chk_ok` task.

The simulation reports, "Simulation stopped due to successful completion" if no errors occur.
Figure 11. Partial Transcript from Successful Simulation Testbench

The following figure shows the behavior of Data Mover interface signals during a read data transfer followed by a write data transfer.

Figure 12. Behavior of Read Data Mover and Write Data Mover Interface Signals During Data Transfers

As shown in the simulation waveforms, the Read Data Mover’s data transfer happened around 200 us, and the Write Data Mover’s data transfer happened around 204 us.

Test Case for the Endpoint Gen4 x16 Design Example
The test case for this design example is in the file:
`intel_pcie_ptile_avmm_0_example_design\pcie_ed_tb\ip\pcie_ed_tb\dut_pcie_tb_ip\intel_pcie_ptile_tbed_100\sim\altpcietb_bfm_rp_gen4_x16.sv`.

The task to run the test is called `avmmdma_rdwr_512IP_test`.

The test case consists of:

For the Read Data Mover:
1. Create a Ramp in Buffer_0.
2. Create an associated descriptor table targeting Buffer_0.
3. Create a descriptor for the read descriptor table (created in step 2) and append a descriptor to write to the host memory when DMA is done.
4. Send the descriptor (created in step 3) to the DMA controller through the BAM Interface.
5. The DMA Controller processes the descriptor then reads the descriptor tables. At this time, all descriptors are fetched and placed inside the DMA Controller FIFO.
6. The Read Data Mover issues a read command to read data from Buffer_0 and send it to the internal RAM.
7. The Read Data Mover issues a MWr for the Status bit.

For the Write Data Mover:
1. Create an empty Buffer_1.
2. Create an associated descriptor table targeting Buffer_1.
3. Create a descriptor for the read descriptor table (created in step 2) and append a descriptor to write to the host memory when DMA is done.
4. Send the descriptor (created in step 3) to the DMA controller through the BAM Interface.
5. The DMA Controller processes the descriptor then reads the descriptor tables. At this time, all descriptors are fetched and placed inside the DMA Controller FIFO.
6. The Write Data Mover issues a read command to read data from internal RAM and send it to the PCIe host memory.
7. The Write Data Mover issues a MWr for the Status bit.

The test case then compares the contents of Buffer_0 and Buffer_1. If they match, the test passes.
5.5. Compiling the Design Example

1. Navigate to `<project_dir>/intel_pcie_ptile_avmm_0_example_design/` and open `pcie_ed.qpf`.

2. If you select one of the supported development kits mentioned in the Generating the Design Example section, the necessary VID-related settings are included in the `.qsf` file of the generated design example.

3. If you are using another Intel Stratix 10 DX development kit, check that appropriate VID-related assignments have been included in the `.qsf` file of your project.

4. If you are using another Intel Agilex development kit, check that appropriate VID-related assignments have been included in the `.qsf` file of your project.

5. On the Processing menu, select **Start Compilation**.

5.6. Installing the Linux Kernel Driver

Before you can test the design example in hardware, you must install the Linux kernel driver. You can use this driver to perform the following tests:

- A PCIe link test that performs 100 writes and reads
- Memory space DWORD(3) reads and writes
- Configuration Space DWORD reads and writes

In addition, you can use the driver to change the value of the following parameters:

- The BAR being used
- The selected device by specifying the bus, device and function (BDF) numbers for that device

Complete the following steps to install the kernel driver:

1. Navigate to `./software/kernel/linux` under the example design generation directory.

2. Change the permissions on the `install`, `load`, and `unload` files:
   
   ```
   $ chmod 777 install load unload
   ```

3. Install the driver:
   
   ```
   $ sudo ./install
   ```

4. Verify the driver installation:
   
   ```
   $ lsmod | grep intel_fpga_pcie_drv
   Expected result:
   intel_fpga_pcie_drv 17792 0
   ```

5. Verify that Linux recognizes the PCIe design example:
   
   ```
   $ lspci -d 1172:000 -v | grep intel_fpga_pcie_drv
   ```

---

(3) Throughout this user guide, the terms word, DWORD and QWORD have the same meaning that they have in the PCI Express Base Specification. A word is 16 bits, a DWORD is 32 bits, and a QWORD is 64 bits.
**5.7. Running the Endpoint Design Example Application**

Table 10. **Test Operations Supported by the P-Tile Avalon-MM PCIe Design Example**

<table>
<thead>
<tr>
<th>Operations</th>
<th>Required BAR</th>
<th>Supported by P-Tile Avalon-MM PCIe Design Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Link test - 100 writes and reads</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>1: Write memory space</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>2: Read memory space</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>3: Write configuration space</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>4: Read configuration space</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>5: Change BAR</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>6: Change device</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>7: Enable SR-IOV</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>8: Do a link test for every enabled virtual function belonging to the current device</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>9: Perform DMA</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>10: Quit program</td>
<td>N/A</td>
<td>Yes</td>
</tr>
</tbody>
</table>

To run a DMA test:
1. Navigate to `./software/user/example` under the design example directory.
2. Compile the design example application:
   
   `$ make`
3. Run the test:
   
   `$ sudo ./intel_fpga_pcie_link_test`

You can run the Intel FPGA IP PCIe link test in manual or automatic mode.
- In automatic mode, the application automatically selects the device. The test selects the Intel Stratix 10 DX or Intel Agilex PCIe device with the lowest BDF by matching the Vendor ID. The test also selects the lowest available BAR.
- In manual mode, the test queries you for the bus, device, and function number and BAR.

For the Intel Stratix 10 DX or Intel Agilex Development Kit, you can determine the BDF by typing the following command:

   `$ lspci -d 1172`

4. For example, enter 0 to select **Automatically select a device.**
5. Enter 9 to perform the DMA test.
6. Enter 0 to run DMA. Then enter 0 again to run an infinite DMA loop test.
7. The DMA test can be stopped by hitting the ESC key. Here is a sample transcript of the results:

```
> 0
Opened a handle to BAR 0 of a device with BDF 0x4b00

> 9
```

```
Current DMA configurations
Run Read (card->system) ? 1
Run Write (system->card) ? 1
Run Simultaneous ? 1
Number of dwords/desc : 2048
Number of descriptors : 128
Total length of transfer : 1024 KiB
```

```
> 0
```

```
> 1
```

```
> 2
```

```
> 3
```

```
> 4
```

```
> 5
```

```
> 6
```

```
```
Current DMA configurations
- Run Read (card->system) : 1
- Run Write (system->card) : 1
- Run Simultaneous : 1
- Number of dwords/desc : 8192
- Number of descriptors : 128
- Total length of transfer : 4.1e+03 KiB

Current run #: 11007
Current time : Mon Jun 8 23:46:45 2020

DMA throughputs, in GB/s (10^9B/s)
- Current Read Throughput : 14.82
- Average Read Throughput : 14.81
- Current Write Throughput : 13.93
- Average Write Throughput : 13.34
- Current Simul Throughput : 24.82
- Average Simul Throughput : 24.73

Note: The throughput numbers above are for a Gen3 x16 design example.

8. Enter 6 to return to the main menu.
9. Enter 5 to change the BAR. Then enter 2 to select BAR2.
10. Enter 0 to start the Link test - 100 writes and reads.
11. The results below show zero write and read error as well as zero Dword mismatch.
12. Enter 5 to change the BAR. Then enter 4 to select BAR4.

13. Enter 4 to Read the configuration space.

14. Enter 0x0 to read out the Vendor ID.

> 5
Changing BAR...
Enter BAR number (-1 for none):
> 2
Successfully changed BAR!

> 0
Doing 100 writes and 100 reads..
Number of write errors: 0
Number of read errors: 0
Number of dword mismatches: 0
5.8. Running the Root Port Design Example Application

5.8.1. Root Port Master System Script

The Root Port design example automatically generates the system script .tcl file. You can find this script in the design example directory `<Project_dir>/software/user/example/intel_fpga_pcie_link_rp.tcl`.

5.8.2. Running the Root Port Master System Script on System Console

The tcl-based Root Port Master system script can be run on System Console by following these steps:

```
0: Link test - 100 writes and reads
1: Write memory space
2: Read memory space
3: Write configuration space
4: Read configuration space
5: Change BAR
6: Change device
7: Enable SRIOV
8: Do a link test for every enabled virtual function belonging to the current device
9: Perform DMA
10: Quit program
```

> 5
Changing BAR...
Enter BAR number (-1 for none):
> 4
Successfully changed BAR!

```
0: Link test - 100 writes and reads
1: Write memory space
2: Read memory space
3: Write configuration space
4: Read configuration space
5: Change BAR
6: Change device
7: Enable SRIOV
8: Do a link test for every enabled virtual function belonging to the current device
9: Perform DMA
10: Quit program
```

> 4
Enter address to read, in hex:
> 0x6
Reading from BDF 0x4b00 config space offset 0..
Read 0x1172
```
1. Launch System Console.
2. Go to File and select Execute Script.
3. Select `<Project_dir>/software/user/example/intel_fpga_pcie_link_rp.tcl`.
4. Run the RP command:
   
   ```
   $ sys.init
   ```
5. Here is a sample transcript showing the results of enumeration and BAR initialization:

```plaintext
$ sys.init
  RF Vendor ID : 0x00001172
  RF bus master/mem/io enable : 0x00100006
  RF Link Status : 0x30430048
  Link Speed : 8.05
  Link Width : x4
    in for loop, bus=1, device=0, func=0
  EF 100 Device_Vendor ID : 0x00001172
  Header Type: 0
  Found a single-function PCIe EF device at b/d/f: 1/0/0, DID/VID: 0x00001172
  Starting EF Initialization for 1/0/0
  Error Reporting Disabled: 0
  Maximum Payload Size: 128
  Extended Tag Field Enabled: 0
  BAR0: Request Type: Memory Request
  BAR0: Address Width: 64-bit
  BAR0: Size: 64.0x8
  BAR0: Assigning pref. Range: 0x0 - 0xffffffff
  BAR1: type: None
  BAR2: type: None
  BAR3: type: None
  BAR4: type: None
  BAR5: type: None
  Ends EF Initialization for 1/0/0
  RP/EF direct connection
  Writing final subordinate bus for RP: 1
  RP ENF : 0x000010160
  pcf base : 0x00000000
  pcf limit : 0x60000000
  pcf num : 2
  nonPref : 0x00000000
  pcf : 0x60000000
  io : 0x00000000
```
### A. Document Revision History for the Intel FPGA P-Tile Avalon Memory Mapped Design Example for PCI Express User Guide

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>IP Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020.07.10</td>
<td>20.2</td>
<td>3.0.0</td>
<td>Added the Platform Designer view of the x8 Endpoint DMA design example. Added the Root Port design example description.</td>
</tr>
<tr>
<td>2020.04.20</td>
<td>20.1</td>
<td>2.0.0</td>
<td>Updated the configurations supported by the Endpoint design example of the P-Tile Avalon Memory Mapped IP for PCIe.</td>
</tr>
<tr>
<td>2019.12.16</td>
<td>19.4</td>
<td>1.1.0</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>