

---

**fpgaemu**

*Release 0.1*

**Chadmond Wu**

**May 06, 2021**



# HARDWARE BASICS

<b>1</b>	<b>FPGA Review and Emulation Overview</b>	<b>3</b>
1.1	A One Minute Introduction to FPGAs . . . . .	3
1.2	What is an FPGA anyways? . . . . .	4
1.3	What's Inside an FPGA? . . . . .	7
1.4	The Basics of Hardware Emulation and HDLs . . . . .	11
1.5	Quick Definitions and Acronyms . . . . .	15
1.6	References . . . . .	15
<b>2</b>	<b>AXI Protocol Overview</b>	<b>17</b>
2.1	The AXI Protocol . . . . .	17
2.2	AXI Reads and Writes . . . . .	17
2.3	AXI4 Connections and Channels . . . . .	18
2.4	AXI Interconnect vs. SmartConnect . . . . .	23
2.5	AXI Verification IP . . . . .	27
2.6	References . . . . .	27
<b>3</b>	<b>Legacy PCI and PCI Express</b>	<b>29</b>
3.1	Peripheral Component Interconnect . . . . .	29
3.2	PCI Overview and Background . . . . .	30
3.3	The Legacy PCI Bus Cycle . . . . .	31
<b>4</b>	<b>DDR Memory and SDRAM</b>	<b>33</b>
4.1	What is RAM? . . . . .	33
4.2	Different Types of RAM . . . . .	35
4.3	SRAM . . . . .	35
4.4	DRAM . . . . .	36
4.5	The DDR SDRAM Protocol . . . . .	37
4.6	References . . . . .	39
<b>5</b>	<b>Clocks, Clocking Wizard, and Timing</b>	<b>41</b>
5.1	Clocks and Clock Conversion . . . . .	41
5.2	Clock Tree . . . . .	41
5.3	Multiple Clocks in an FPGA . . . . .	41
5.4	Propagation Delay . . . . .	42
5.5	Setup and Hold FF Time . . . . .	42
5.6	Metastability Prevention . . . . .	42
5.7	Clock Domain Crossing (CDC) . . . . .	42
<b>6</b>	<b>Linux Drivers, Kernel Programming, and You</b>	<b>43</b>
6.1	What is a Driver? . . . . .	43
6.2	Linux Driver Architecture and APIs . . . . .	44

6.3	Linux Kernel Modules . . . . .	45
6.4	Transferring Data Within the Kernel . . . . .	46
6.5	References . . . . .	46
<b>7</b>	<b>DMA/Bridge for PCIe Drivers Overview</b>	<b>47</b>
7.1	The PCIe DMA Driver . . . . .	47
7.2	Accessing and Building the Xilinx Driver . . . . .	47
<b>8</b>	<b>Creating a User-Friendly DUT GUI</b>	<b>49</b>
8.1	What is a GUI? . . . . .	49
8.2	System Overview . . . . .	50
8.3	Environment Setup . . . . .	51
<b>9</b>	<b>MIG 7 Series IP Overview</b>	<b>55</b>
9.1	Customizing the IP . . . . .	55
9.2	Simulating the Example Design . . . . .	57
9.3	Simulating Read/Writes with AXI VIP . . . . .	59
9.4	Connecting the MIG to a Custom Design . . . . .	66
9.5	Connecting the MIG to Two AXI Master VIPs using AXI SmartConnect . . . . .	68
<b>10</b>	<b>AXI MM to PCIe IP Overview</b>	<b>77</b>
10.1	Customizing the IP . . . . .	77
10.2	Simulating the Example Design . . . . .	77
10.3	Example IP Block Diagram . . . . .	79
10.4	Replacing the BRAM with DDR MIG in Example Design . . . . .	80
10.5	Simulating the AXI MM PCIe MIG Example Design . . . . .	91
<b>11</b>	<b>DMA/Bridge for PCIe IP Overview</b>	<b>95</b>
11.1	DMA IP Overview . . . . .	95
11.2	The DMA Protocol . . . . .	95
11.3	Configuring the DMA IP . . . . .	97
11.4	Additional Resources . . . . .	103
<b>12</b>	<b>Creating a Custom AXI IP Core</b>	<b>105</b>
12.1	Packaging Custom IP . . . . .	105
12.2	A Simple 8-Bit Counter . . . . .	105
12.3	Packaging a Custom AXI4Lite IP . . . . .	109
12.4	Adding a Custom AXI IP to a Design . . . . .	114
12.5	Creating a Testbench for a Custom DUT . . . . .	116
12.6	Interpreting Simulation Waveforms For a Custom DUT . . . . .	120
<b>13</b>	<b>Creating an Advanced Custom AXI Descriptor IP</b>	<b>123</b>
13.1	Features of Advanced DUT . . . . .	123
13.2	Editing the Descriptor Counter IP . . . . .	125
13.3	Creating the Master DUT Simulation Environment . . . . .	140
13.4	Testbench for a Master Custom DUT . . . . .	140
13.5	Simulating the Master Custom DUT . . . . .	141
<b>14</b>	<b>Building a Basic Simulation Environment (VC707)</b>	<b>143</b>
14.1	Generating PCIe and MIG Example Designs . . . . .	143
14.2	Creating the Block Diagram . . . . .	146
14.3	Connecting it All Together . . . . .	149
14.4	Modifying and Running the Simulation . . . . .	153
14.5	Checking Timing, Viewing Power Reports, Monitoring I/O Placement: . . . . .	158



<b>15 Building an Emulation Environment (without a Board)</b>	<b>167</b>
15.1 Project Start (Building the Block Diagram) . . . . .	167
<b>16 Project Summary</b>	<b>171</b>
16.1 Project Abstract . . . . .	171
16.2 Project Background . . . . .	171
<b>17 Additional Resources</b>	<b>173</b>
17.1 Tutorials . . . . .	173
<b>18 Indices and tables</b>	<b>175</b>
18.1 Acknowledgements . . . . .	175



**Note:** You can find a brief summary of this project [here](#) and don't forget to check out our GitHub [repo](#).

**Important:** New to FPGAs or just need a refresher? Jump [here](#) first!

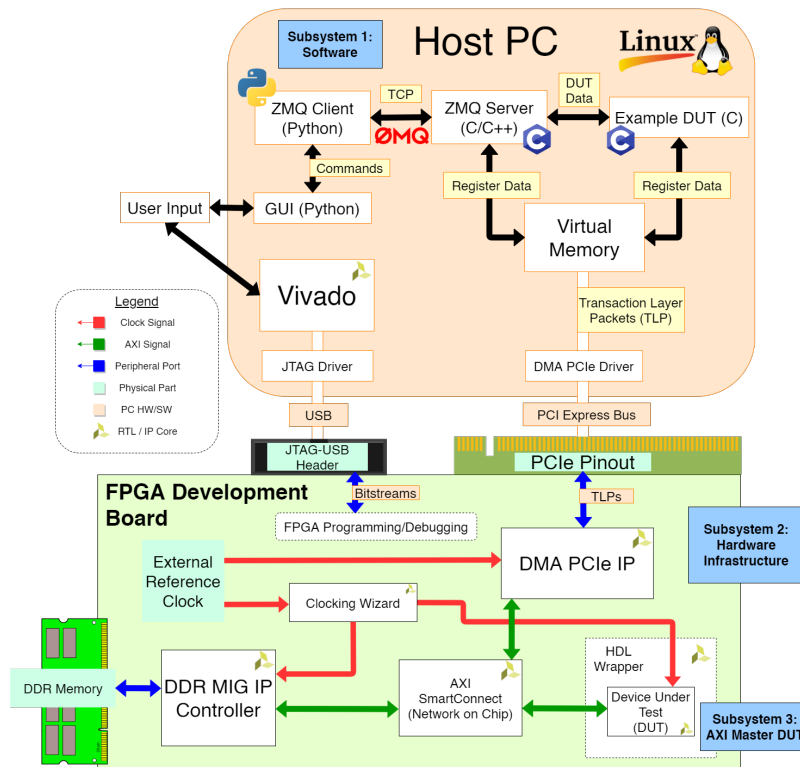


Fig. 1: Complete Block Diagram of FPGA Board



## FPGA REVIEW AND EMULATION OVERVIEW

---

**Important:** This section is intended for FPGA and digital design beginners, although some intermediate concepts are briefly discussed.

---

### 1.1 A One Minute Introduction to FPGAs

This fantastic quick video from Charles Clayton outlines the very basics of an FPGA. If you do not know what an FPGA is, start here before moving on.

Still confused? Here is a more approachable example that may help:

Imagine you had a box of USB sticks to sell, where each stick performs some task, whether it is counting from one to ten, transferring some songs to the hard drive, or even running another smaller computer. Because we hope to sell them, we should test every single stick (what we will refer to as a *device under test*) as much as possible for maximum compatibility to avoid any future errors or refunds. But, considering how many computers there are in the world, it is almost impossible to verify that every stick will work on every laptop or desktop aside from buying and testing on each one individually. The price for failure is high too, as a broken USB stick could potentially damage a user's computer, leading to an expensive safety recall.



Now imagine if you had access to a special computer called an emulation evaluation board. From the outside, it is like any other computer, as the board also has USB, Ethernet, and other standard peripherals. However, at the heart of the board is what is known as an *FPGA*, a special kind of chip that can replicate/emulate every other computer in existence. Although a little clunky and hard to use, this FPGA means that you can now rest assured knowing that your USB sticks can be fully tested on every possible configuration before being sold. Not to mention, the board only needs to be set up once as an *emulation environment* before you're able to easily swap in and out DUTs for testing. Not bad, right?

Ready for a (slightly) technical deep dive? Read on for more details.

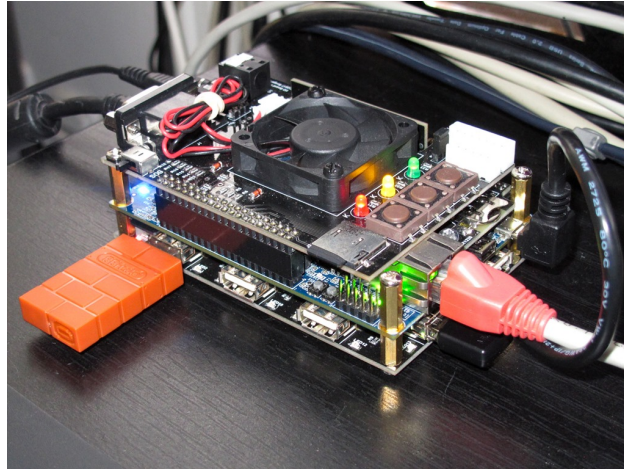
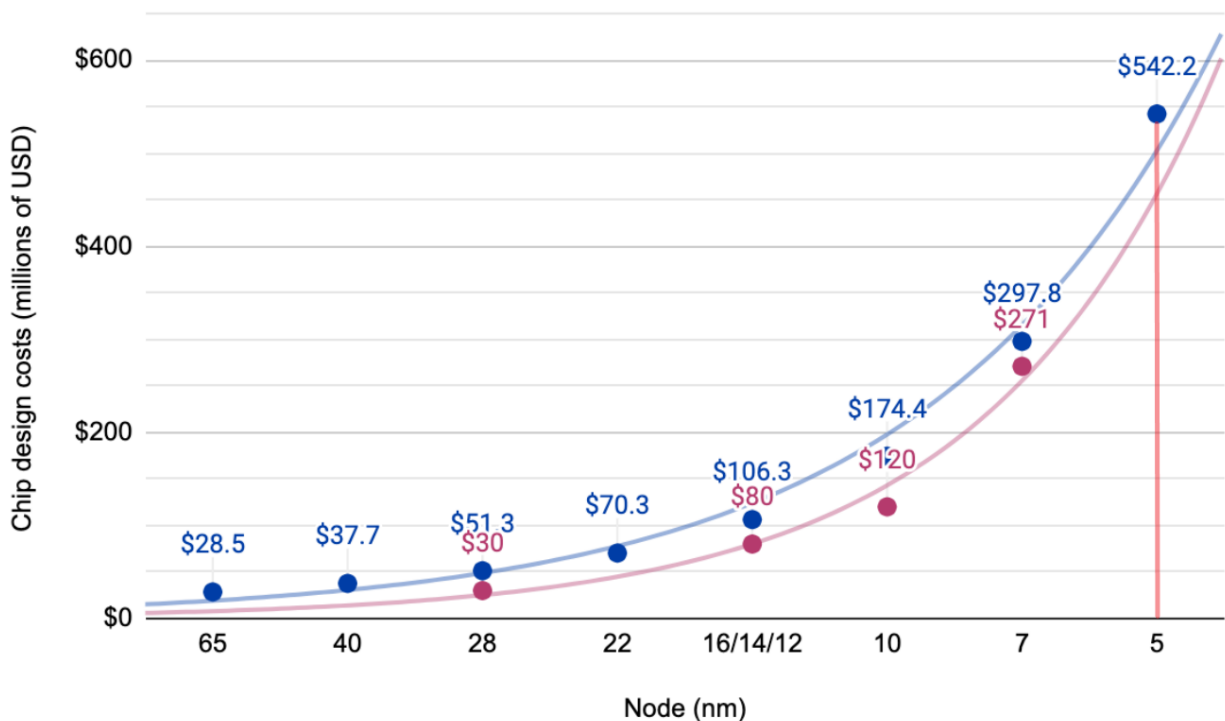


Fig. 1: A MiSTer board with Cyclone V FPGA

## 1.2 What is an FPGA anyways?

In 2020, Apple unveiled their newest computer chip, the M1, as the first 5nm processor for their range of Macintosh computers. As their very first in-house designed system on a chip, or **SoC**, the process of creating the M1 and other competitors like the Qualcomm Snapdragon 888 was very expensive, with costs reaching into the hundreds of millions of US dollars.



As an SoC iterates through each design and manufacturing step, the harder (and more expensive) it becomes to correct any errors. Think of a nightmare scenario where a manufacturer is forced to recall every single smartphone and computer due to a fatal bug with the CPU that was never caught (this happened to Intel and their Pentium processors in 1994!) This is where an FPGA becomes extremely useful.

Field programmable gate arrays or **FPGAs** are integrated circuits (a set of circuits layered within semiconductor material like silicon, also known as chips) designed to be configured by a customer or designer after manufacturing. With FPGAs, a designer can program features, adapt to environment and regulatory changes, and reconfigure hardware even in the field - hence the term *field programmable*. An FPGA contains programmable logic elements (LEs) that either act as basic logic gates or connect to perform complex actions as logic blocks. From a top-level perspective, FPGAs consist mostly of configurable memory, high speed I/O, logic blocks, and routing.



Fig. 2: A Virtex-7 FPGA on an ADC/DAC signal board<sup>1</sup>

While SoCs like the Snapdragon 888 and other ICs are not intended to be physically changed after manufacturing, FPGAs allow for design flexibility and provides the opportunity to change how sections in a system work without introducing subsequent cost, delays, or design risk. For example, because FPGAs excel in processing digital signals quickly, one common application is machine vision. An FPGA projecting a back-up camera onto the rear-view mirror of a self-driving car can be modified to reduce latency and comply with new government standards simply through a software update. Conversely, this flexibility is almost impossible with a microprocessor, as any drastic changes would ultimately result in a complete redesign.

This optimized behavior is possible due to one key point – FPGAs operate in parallel. Normally, a processor must load in instructions in a linear fashion, even for simple tasks such as multiplying or shifting bits. Each instruction must be evaluated in order before the CPU can move onto the next one. This is fine under normal use, but in real-time applications where latency must be as low as possible, having to wait for the AC to activate before the brake pedal can be used is unacceptable. In contrast, an FPGA can execute multiple complex operations simultaneously — with a 10-element matrix, a designer can implement 10 signal/data pipelines to use in parallel. While a microprocessor has sequential processing, an FPGA's concurrent processing allows it to achieve better optimization and a more deterministic latency than even a processor running an RTOS.

In more nuanced terms, the flexibility from an FPGA allows a designer to decide which operations occur at any given clock cycle. Even though FPGAs are clocked much slower than CPUs (100 MHz vs. up to 4 to 5 GHz), given the right design, FPGAs can become much more optimal than even the fastest CPUs. For example, instead of only being able

<sup>1</sup> The example FPGA ADC/DAC board used.

<sup>2</sup> Xilinx's automotive system is discussed in this [press release](#).

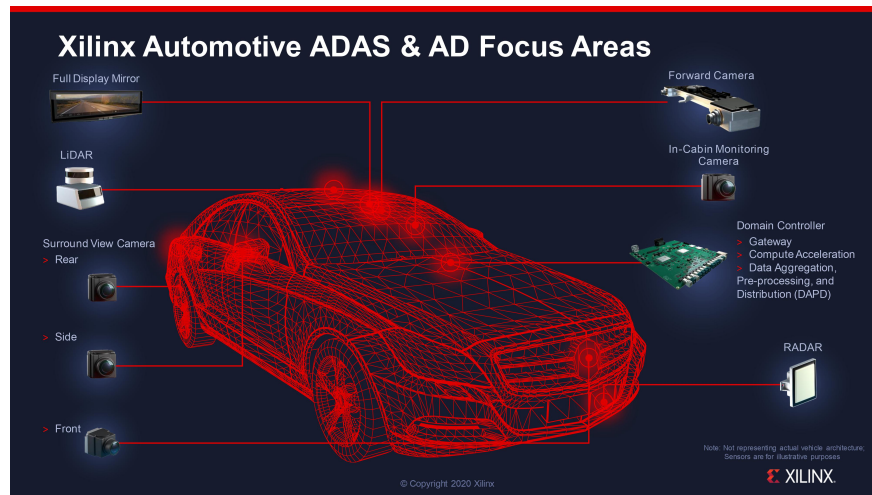


Fig. 3: Example of how an FPGA can excel as a DSP for self-driving cars<sup>2</sup>

to multiply two numbers at every cycle with a CPU, a designer can use all 10 pipelines to multiply 20 numbers for every single cycle - 1/10th the time it would take with a traditional CPU, assuming that the CPU is never interrupted by another instruction. Because an FPGA has very high determinism (in that we know exactly when each instruction will execute), we can rest easy knowing that the FPGA will never be unintentionally interrupted and continue to perform at a consistently high threshold.

**Important:** If you are coming from a traditional programming background, this is an especially crucial point. You are most likely accustomed to coding in sequential order, so always consider concurrency when working with FPGAs!

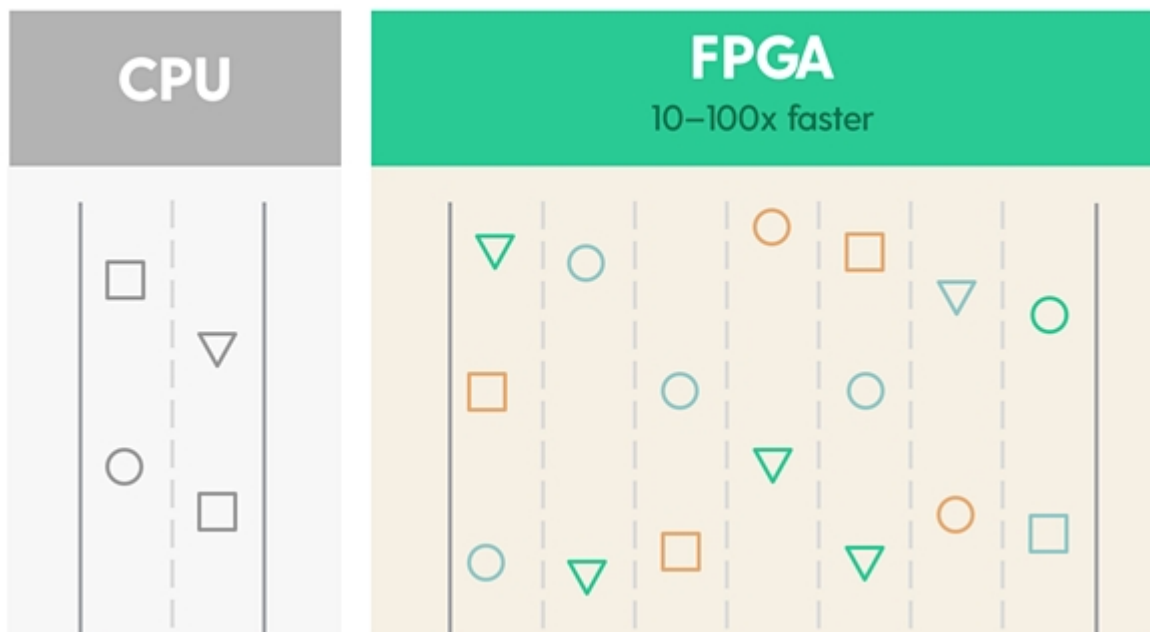


Fig. 4: A dual-core CPU vs. multi-channel FPGA<sup>3</sup>

<sup>3</sup> More about FPGA computational performance can be found in this [article](#).



## 1.3 What's Inside an FPGA?

While a software engineer writes linear high-level code to become compiled into low-level assembly instructions, a hardware designer does not have a compiler; instead, they manipulate much lower-level discrete digital components like LUTs, registers, etc. Again, this is important to remember moving forward — while programmers write software code, FPGA designers ‘code’ physical hardware present on the FPGA itself.

The smallest component on an FPGA is the simple logic gate. Of course, discrete logic gates do not physically exist inside an FPGA — instead, FPGAs calculate Boolean algebra using look up tables (**LUTs**) as truth tables, where each LUT can calculate any Boolean algebra equation based on the number of inputs. A typical FPGA will have thousands of three, four, and five-input LUTs. More about Boolean algebra and logic gates [here](#).

By combining multiple logic gates together, we can create truth tables that achieve more complicated functions. One of the most common examples is a **D flip-flop**, a logic component that changes the output Q based on the input D. The FF stores the current value on the D data line, essentially acting as a basic memory cell. FFs use sequential or registered logic, meaning that it operates based on the regular transitions of a clock, driven by the clock input line (>). FFs register data from D to Q on a clock's rising (or falling) edge, or when the clock transitions from 0 to 1 (or 1 to 0). With multiple flip-flops (or registers in this case) all acting as data storage elements, they collectively store the current state of the entire FPGA, including counters, state machines, and evaluations of other components. If an FPGA only had LUTs without any FFs, the FPGA would have no memory, forced to immediately evaluate all changes on any inputs and preventing any saved programs from working (meaning that our multiplication example from before would no longer function).

In a similar manner, a **gated D latch** is a simpler non-clocked flip-flop that is also used to store state information. A D flip-flop uses a clock signal to transfer data, while a latch simply checks an E enable input line. As before, input D is the data input line, containing the value to transfer to output Q (or Q bar, the opposite of Q). Q only receives the value on D when Enable is HIGH or 1 — when E is 0, output Q is considered ‘latched’ and will not change regardless of input D.

---

**Note:** Latches are often created unintentionally from incomplete assignments, so beginners are not advised to use them until they have significant FPGA debugging experience.

---

From an overarching perspective, an n-bit LUT is implemented as a  $2^n \times 1$  memory component. In other words,  $2^n$  SRAM latches hold the value of each LUT input combination, creating a larger general purpose truth table. Each latch is controlled by a  $2^n \times 1$  multiplexer, or **mux**, which is a simple logic component that chooses which of its multiple inputs to transfer to its output line. LUT inputs into the mux control determine which latch values are passed to output Q. For example, 16 latches store multiple values in memory and feed into a 16 by 1 mux. The mux also receives control inputs that determine which latch is pushed to the output.

For example, given a 4-input truth table with 16 rows, for the input ABCD = 0101, the output Y will be 1<sup>7</sup>.

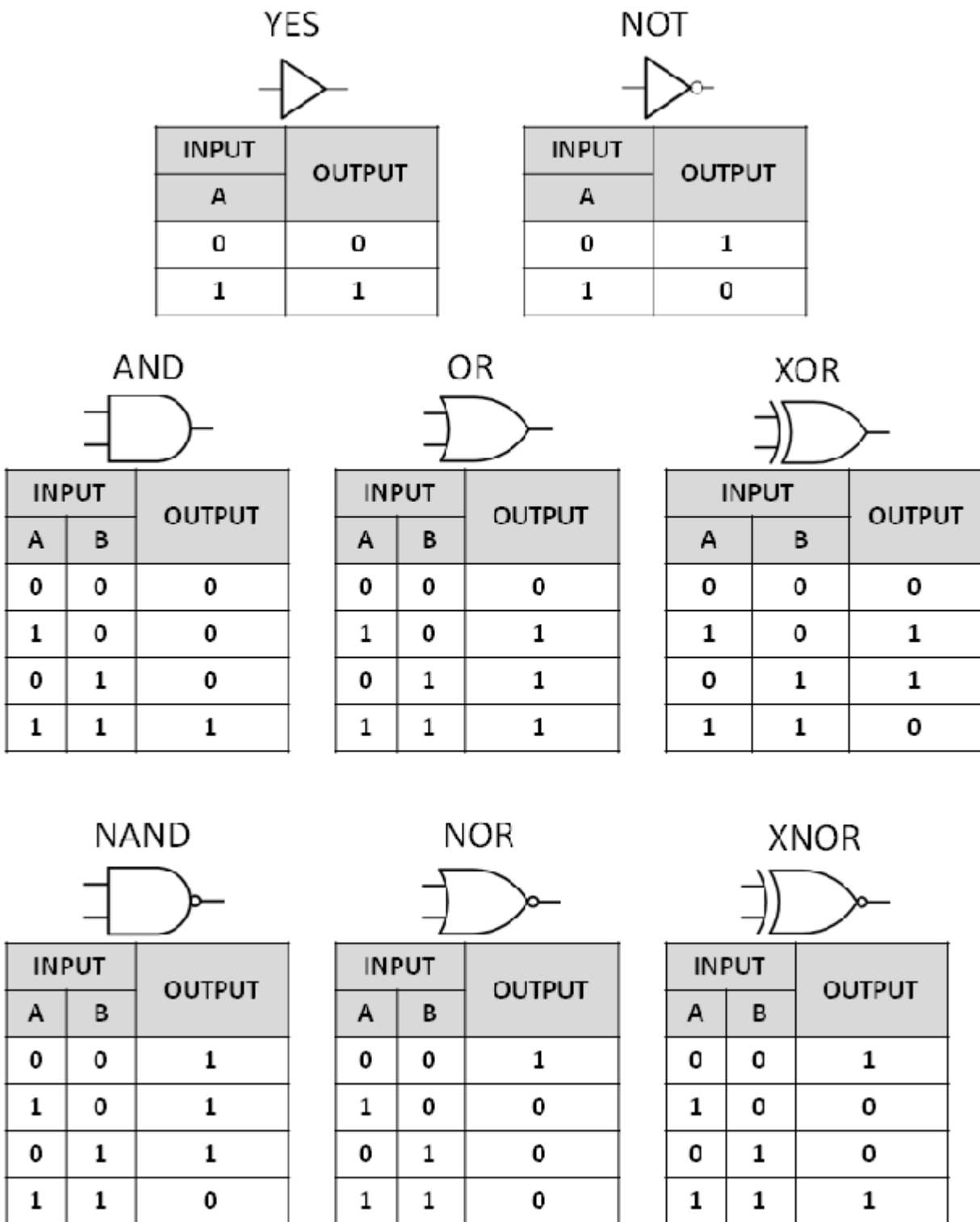
---

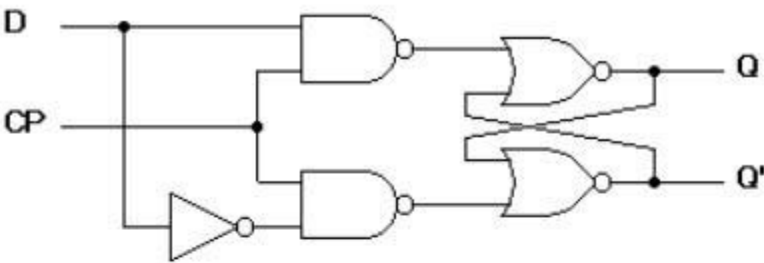
<sup>4</sup> From Abels, S. G., & Khisamutdinov, E. F. (2015). Nucleic Acid Computing and its Potential to Transform Silicon-Based Technology. DNA and RNA Nanotechnology, 1(open-issue), 13-22.

<sup>5</sup> More about flip flops and their diagrams are [here](#).

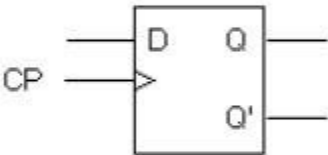
<sup>6</sup> From Abdel-Lattif, G. Y., Rehan, S. E., & Abdel-Fattah, A. F. I. (2012). OPTIMIZED SINGLE-ELECTRON NAND-BASED D-LATCH/FLIP-FLOP. The Mediterranean Journal of Electronics and Communications, 8(4).

<sup>7</sup> More about LUTs [here](#).

Fig. 5: Summary of common logic gates/truth tables<sup>4</sup>



(a) Logic diagram with NAND gates

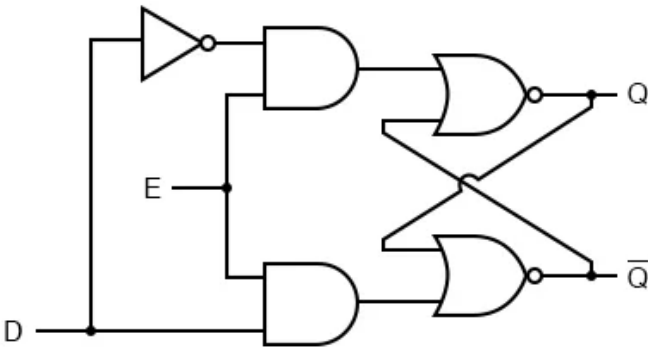


(b) Graphical symbol

Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

(c) Transition table

Fig. 6: Summary of a D flip flop<sup>5</sup>



E	D	Q	$\bar{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

Fig. 7: Summary of a D latch<sup>6</sup>

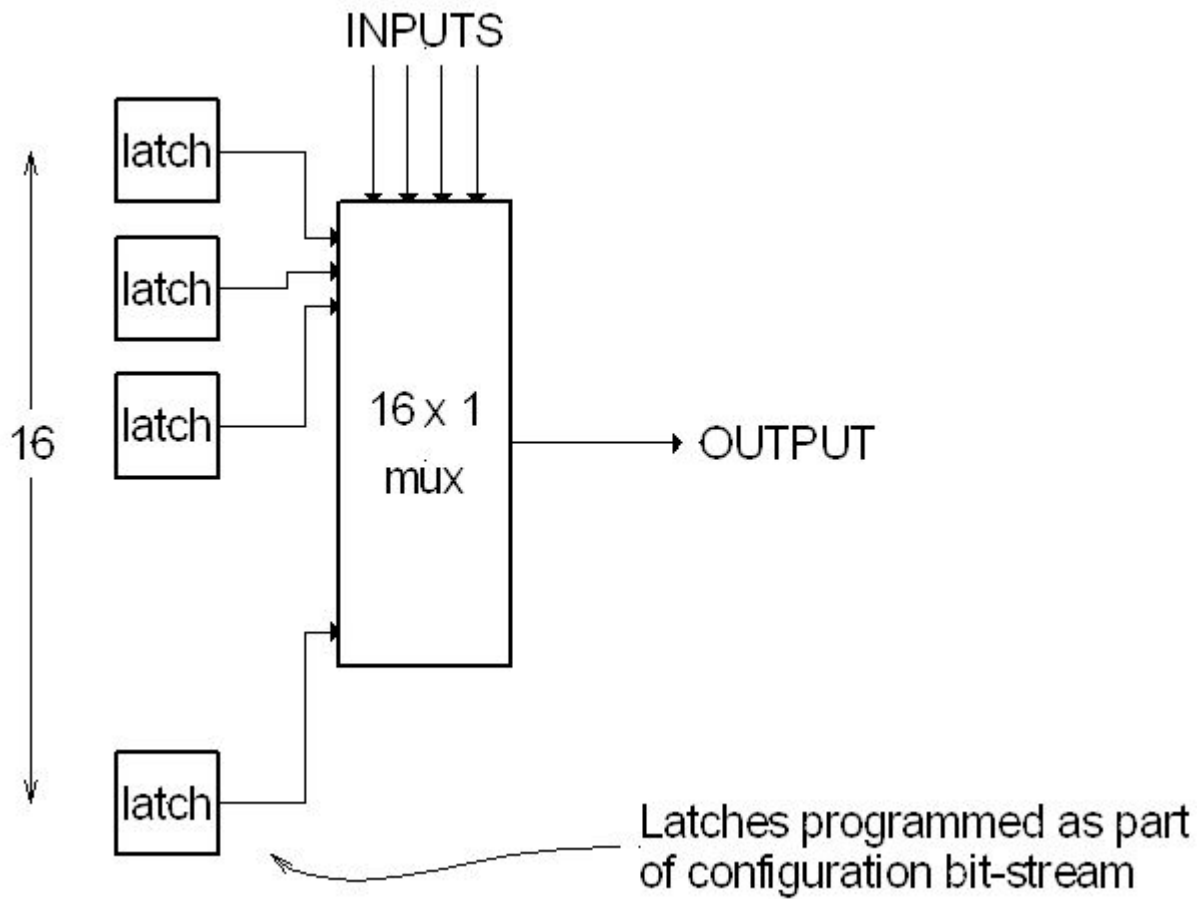
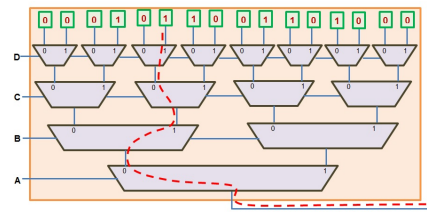
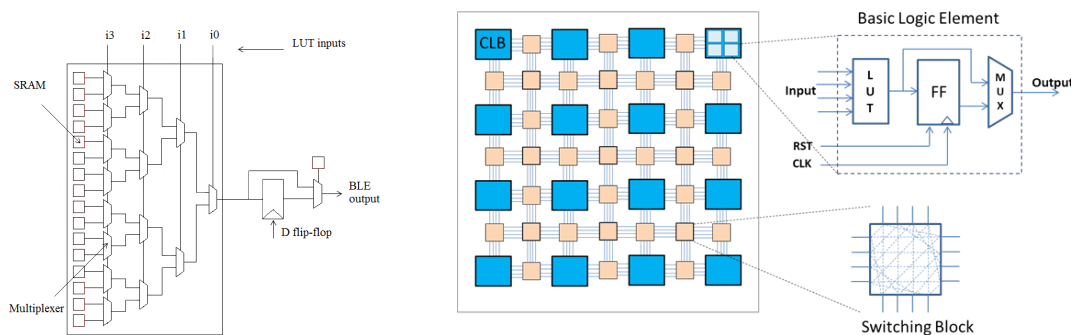


Fig. 8: Abstracted block diagram of a look-up table

Truth Table				
Inputs				Output
A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



A flip-flop then stores the LUT's output. One last multiplexer decides, based on the given configuration, whether the output receives the value from the flip-flop or directly from the LUT itself. As a whole, all components make up a single **configurable logic block (CLB)**. Like the introductory video, these logic blocks are routed together using switching blocks to finally make up the entire FPGA floorplan<sup>8</sup>.



## 1.4 The Basics of Hardware Emulation and HDLs

As we have discussed at length, FPGAs provide an unparalleled combination of performance and flexibility that rivals even the most expensive processors (of course, industry-grade FPGAs cost quite a bit too). This reprogrammability allows FPGAs to excel at one of their most interesting applications — **hardware emulation** or the method of copying the behavior of another hardware sample. Referring to our Apple M1/Qualcomm 888 example, instead of manufacturing new designs on the assumption that they will work 100% of the time, most if not all semiconductor companies first use FPGAs to emulate their chips. Hardware emulation allows these manufacturers to debug their designs in simulated but realistic conditions before undertaking the extreme cost of mass fabrication. By chaining multiple FPGAs together (sometimes up to the scale of entire rooms for one chip alone), these companies are able to logically simulate even the most complex integrated circuits in real time, testing both hardware performance and software compatibility. Modern GPUs and CPUs have billions of transistors, so ultimately hardware emulation is and will continue to be an essential part in the semiconductor industry.

Hardware emulation is achieved through a number of steps. First, the design is created from **HDL** code, also known as a hardware description language. Similar to traditional programming languages like C or Python, an HDL like Verilog or VHDL instantiates the FPGA's physical hardware using digital code. HDLs execute instructions in parallel, while software languages operate in sequential order. Designs are then **synthesized**, wherein the human-understandable code is converted in a **netlist** of connected logic gates or flip-flops. Essentially, synthesis acts the same as compiling software code to machine assembly code.

Place and route (**P&R**), or implementation, is a set of multiple procedures in which the list of nets is physically placed and mapped to the FPGA's resources. Implementation creates a roadmap where each element can be placed onto the

<sup>8</sup> FPGA floorplan from this [info page](#).

<sup>9</sup> More about NVIDIA's emulation lab in this [blog post](#).

<sup>10</sup> Details about the FPGA design flow [here](#).



Fig. 9: An entire room-scale Cadence Tigris emulator<sup>9</sup>

```
module and2 (c, b, a);  
output c; input a,b;  
assign c = a&b;  
endmodule
```

AND gate using Verilog



Synthesized AND gate

Fig. 10: Simple example of HDL synthesis<sup>10</sup>

FPGA chip. At the end, the software will output a **bitstream** that designers can program onto the FPGA for further testing. Both synthesis and implementation are typically done with first-party software, although synthesis can be completed with third-party alternatives.

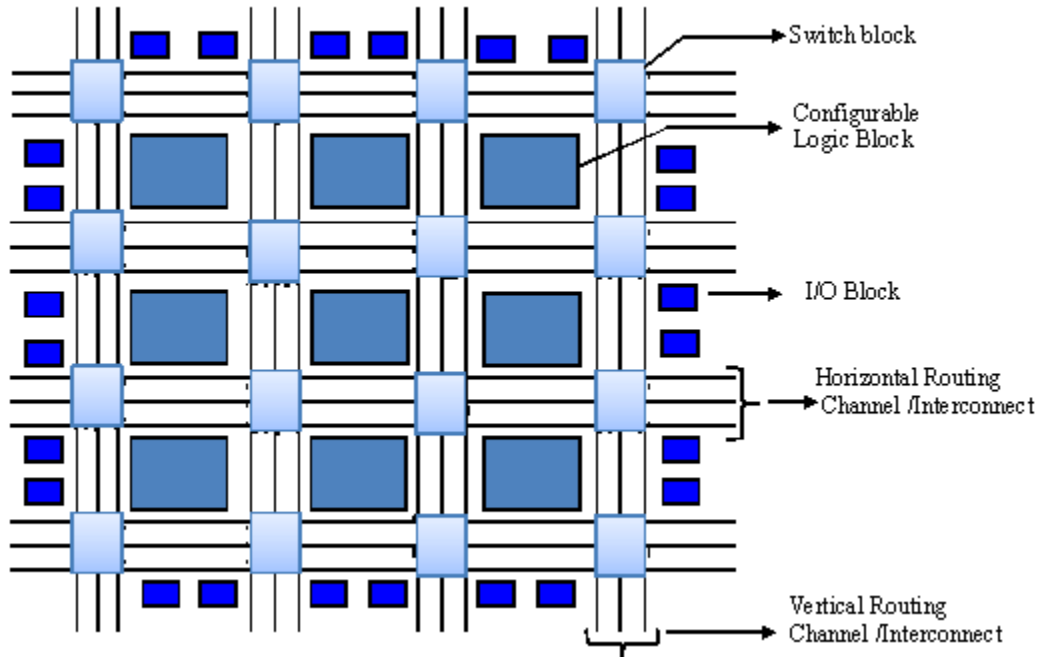


Fig. 11: Simple example of P&R<sup>11</sup>

Finally, the tasks of **simulation** and **verification** provide useful debugging methods along the entire development process. Verification is a multi-stage procedure from writing a testbench or set of tests in HDL code that checks the design against a given specification to testing for timing concerns. Behavioral simulation is one aspect of verification that simulates an environment based on the testbench and outputs relevant signal waveforms. Do not worry if you have little experience with reading waveforms or using an oscilloscope, as we will be explaining our simulation tests in every example project we provide using Vivado's ModelSim. This article will not go into SystemVerilog and UVM, as that is outside the scope of this entire project.

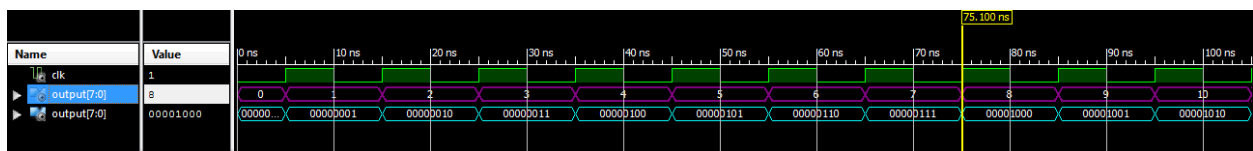


Fig. 12: Simulating an 8-bit binary counter

Of course, going through the entire process from synthesis to bitstream for every single hardware component is time-consuming, especially if you are repeatedly working with the same FPGA. It would more efficient and easier to first create the infrastructure first in the form of an **emulation environment**. This would include peripherals like the board memory or PCIe pinout, which never change between designs. After all, if you end up working with the same board, why start from scratch for every project? A premade environment allows us to get a running start for every future project onwards, which is why setting up such an environment is our first example project.

<sup>11</sup> Venugopal, N., Shobana, V., & Manimegalai, R. (2014, January). Analysis of optimization techniques in FPGA placement. In 2014 International Conference on Computer Communication and Informatics (pp. 1-5). IEEE.



**Note:** Don't worry if you don't have access to a physical FPGA board — 90% of design work is done in simulation anyways!

---

After the environment is completed, we will continue to guide you through creating and simulating a device under test (DUT), letting you emulate everything from a simple counter to a complex SoC.

---

**Important:** Jump [here](#) to get started with your environment. Otherwise, continue to the next page for a deeper introduction into the hardware.

---

What is a real-world example of hardware emulation? One interesting application that has evolved in the last few years is the [MiSTer project](#), an open-source design that emulates old video game consoles using nothing more than a small FPGA board. Using the same principles as software emulation, the MiSTer project emulates multiple reversed-engineered consoles on a single Altera Cyclone FPGA, opening the opportunity for a home arcade at a fraction of the price. Instead of paying hundreds of dollars for a new and working Nintendo Famicom Disk System, which was never released in the West, or use software emulation to run code in a similar fashion, the FPGA board can instead emulate the console hardware itself and play every game that was ever released with the same level of performance and compatibility. Of course, since FPGAs are flexible, an FPGA can reconfigure itself through LUTs to emulate other hardware. This means that different console cores from Atari to Pac-Man can be swapped out at any time, again illustrating the versatility of FPGAs and serving as a good example for our emulation environment project. By building up the proper infrastructure, it would become easy in the future to swap in different DUTs like the MiSTer cores for testing and debugging, similar to standard industry practices in the semiconductor field. All without even touching the original hardware.

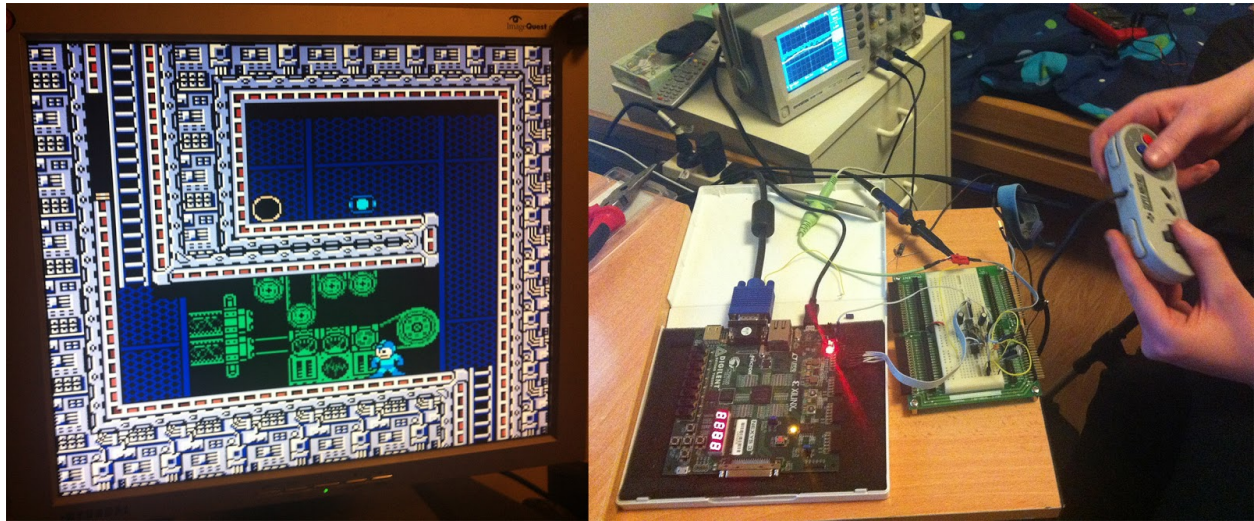


Fig. 13: Emulating an NES game console on an FPGA<sup>12</sup>

---

<sup>12</sup> More about the NES FPGA project [here](#).



## 1.5 Quick Definitions and Acronyms

**IC** [Integrated Circuit] Collection of electronic components on a single unit, typically made from silicon, also known as a chip.

**FPGA** [Field Programmable Gate Array] ICs designed to be configurable by engineer after manufacturing.

**ASIC** [Application Specific Integrated Circuit] Highly specialized ICs dedicated to one specific application.

**SoC** [System on a Chip] IC that hosts an entire computer system by itself.

**P&R** [Place and Route] Process by which logic components are placed onto an FPGA and connected/routed together.

**DUT** [Device Under Test] Any electronic part currently being tested, through emulation in our case.

**IP** [Intellectual Property] Commonly used electronic parts abstracted as logic blocks, provided by external companies (not the same as a patent).

**AXI** [Advanced eXtensible Interface] Communication standard that allows chip components to send signals to each other.

**MIG** [Memory Interface Generator] Xilinx IP that allows an FPGA to read/write into DDR memory.

**DDR SDRAM** [Double Data Rate Synchronous Dynamic Random-Access Memory] Volatile memory IC typically used to store information that is lost when power is lost, common interfaces are DDR3 and DDR4.

**PCIe** [Peripheral Component Interconnect Express] Communication network that allows an FPGA to control peripherals/communicate with a host PC.

**TLP** [Transaction Layer Packets] Data payloads that peripherals send through the PCIe bus.

**DMA** [Direct Memory Access] Xilinx IP that allows AXI peripherals to directly access memory without the help of the processor.

**ROM** [Read Only Memory] Flash memory that cannot be modified afterwards.

## 1.6 References



## AXI PROTOCOL OVERVIEW

### 2.1 The AXI Protocol

When building your first block diagram or reading the documentation of Xilinx's IP cores, you may notice one thing in common – they all use the AXI protocol. This article will provide a brief explanation about what AXI is and how it functions.

The *Advanced eXtensible Interface*, or **AXI**, protocol is a royalty-free communication standard developed by ARM, a prolific system-on-chip (SoC) company, as part of the AMBA (Advanced Microcontroller Bus Architecture) standard. You can find more information about AMBA and its other protocols (such as AHB or APB) [here](#).

Essentially, the AXI protocol outlines the process by which on-chip components can communicate with each other using signals, usually involving a master and slave device. By standardizing this protocol, we can ensure every peripheral and IP core present on an FPGA will be able to talk to each other, creating a cohesive system (rather than a scattered collection of cores).

There are three types of AXI4 interfaces (defined by AMBA 4.0):

- Full AXI4 - High-performance communication, using memory-mapped addresses ([more here](#)).
- AXI-Lite - Lightweight and simple memory-mapped interface, used for single transaction communication.
- AXI4-Stream - 'Direct' device communication, removing the need for addresses and allowing for maximum data transfer.

For the remainder of this article and throughout our projects, we will mainly focus on Full AXI4 for the best performance-cost ratio.

### 2.2 AXI Reads and Writes

#### Memory Addresses

*Example of how AXI can control devices using addresses.*

Both pure data and commands (like toggling an LED) can be sent on the data bus.

Address	Purpose
0x00000	Config
0x10000	LED1
0x20000	DDR Reg

AXI4 allows for multiple data transfers over a single request, allowing for greater data bandwidth in the scenario where large amounts of data must be transferred to/from specific addresses. This multi-transfer request is also known as a *burst*.

All AXI communication is with respect to memory addresses, which each have a specific purpose defined by the RTL and top module.

Three burst types are supported - **FIXED**, **INCR**, and **WRAP**. Each one alters the transfer address in a specific way, allowing for optimal transfers in different situations. For example, a **FIXED** burst sets every beat to have the same address, which is useful for memory transfers from the same repeated location.

In general, burst addressing specifies where each read or write should be performed in which addresses. Each burst type is as follows<sup>1</sup>:

Fig. 1: AXI Bursts

AXI4-Lite has no burst protocol (only sending one piece of data at a time) while AXI4-Stream acts as a single unidirectional channel for unlimited data flow between a master and slave, removing the need for addresses.

## 2.3 AXI4 Connections and Channels

In its most basic configuration, the AXI protocol connects and facilitates communication between one master and one slave device. As expected, the master initiates and drives data requests, while the slave responds accordingly. This communication, or transactions as we will now refer to, occurs over multiple channels, each one dedicated to a specific purpose.

The sender must always assert a **VALID** signal before the receiver and keep it **HIGH** until the handshake is completed. By using handshakes, the speed and regularity of any data transfer can be controlled.

There are five channels, each one transmitting a data payload in one direction. Each channel implements a handshake mechanism, wherein the sender drives a **VALID** signal when it has prepared the payload for delivery and the receiver drives a **READY** signal in response when it is ready to receive the data. The data transfer is also known as a *beat*.

Fig. 2: AXI Handshake Protocol

The five AXI4 channels are as follows:

- Write Address channel (AW): Provides address where data should be written (**AWADDR**)
- Can also specify burst size (**AWSIZE**), beats per burst (**AWLEN + 1**), burst type (**AWBURST**), etc.
- **AWVALID** (Master to Slave) and **AWREADY** (Slave to Master)
- Write Data channel (W): The actual data sent (**WDATA**)
- Can also specify data and beat ID
- Sender will always assert a finished transfer when done (**WLAST**)
- **WVALID** (Master to Slave) and **WREADY** (Slave to Master)

• Write Response channel (B): Status of write (**BRESP**)

• **BVALID** (Slave to Master) and **BREADY** (Master to Slave)

• Read Address channel (AR): Provides address where data should be read from (**ARADDR**)

• Can also specify burst size (**ARSIZE**), beats per burst (**ARLEN + 1**), burst type (**ARBURST**), etc. **ARVALID** (Master to Slave) and **ARREADY** (Slave to Master)

• Read Data channel (R): The actual data sent back

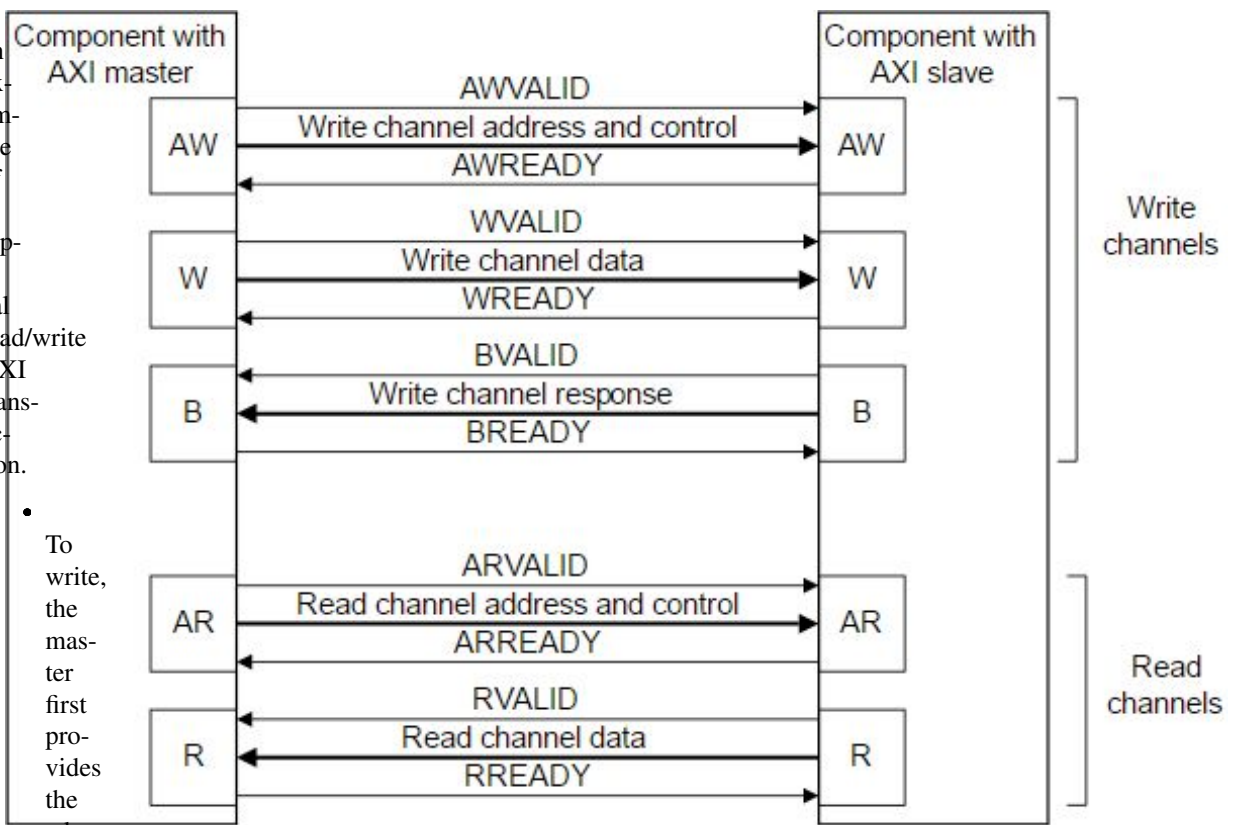
---

<sup>1</sup> AXI example images used from Wikimedia Commons and the [AXI Article](#).

Can also send back status (RRESP), data ID, etc. Sender will always assert a finished transfer when done (RLAST)  
RVALID (Slave to Master) and RREADY (Master to Slave)

Here

is an example of a typical read/write AXI transaction.



To write, the master first provides the address (0x0)

to write to, as well as the data specifications (4 beats of 4 bytes each, data type of INCR). Both the master and slave then exchange a handshake for verification.

The master then prepares and writes the actual data payload to send over the

chan-  
nel  
(0x10,  
0x11  
0x12,  
and  
0x13),  
again  
us-  
ing  
a  
hand-  
shake  
to  
ver-  
ify  
the  
trans-  
fer.  
The  
mas-  
ter  
will  
sig-  
nal  
the  
end  
of  
the  
pay-  
load  
to  
the  
slave  
us-  
ing  
WLAST.

- The  
slave  
re-  
sponds  
with  
a  
sta-  
tus  
of  
the  
write  
and  
whether  
it  
was

suc-  
cess-  
ful  
or  
a  
fail-  
ure  
(all  
OKAY  
in  
this  
case)  
and  
fin-  
ishes  
the  
en-  
tire  
trans-  
ac-  
tion  
with  
an-  
other  
hand-  
shake.

Fig. 3: A typical AXI Write transaction

- To read, the master first provides the first address to read from (0x0), as well as the data specifications (4 beats of 4 bytes each, data type of INCR). The usual handshake occurs.
- The slave then provides the actual data payload, as well as the status of each beat (all beats are OKAY). The slave will signal the end of the payload to the master using `RLAST`. As we can see, what was written to the specified addresses was the same as what was read back.

Fig. 4: A typical AXI Read transaction

We can also get an idea about what an AXI read and write cycle would look like in simulation through the 7 Series MIG documentation ([UG586](#)). As we can see, an AXI write consists of a command cycle (define address and burst length), data cycle (putting the data payload over the channel), and a response cycle (checking if the data was received). The master defines the payload specifications and writes the actual data payload (5a5aa5a5 at address 00000000). The slave toggles `s_axi_bvalid`, exchanging a handshake that signifies the transfer was successful.

Subsequently, an AXI read consists of a read command cycle (again, defining the address to read from, burst length, etc.) and a read data cycle (receiving the data from the requested address). The master specifies the address (00000000) and other payload specs, receives the data payload from the slave (5a5aa5a5), and exchanges a final handshake by toggling `s_axi_rlast` to complete the transfer.

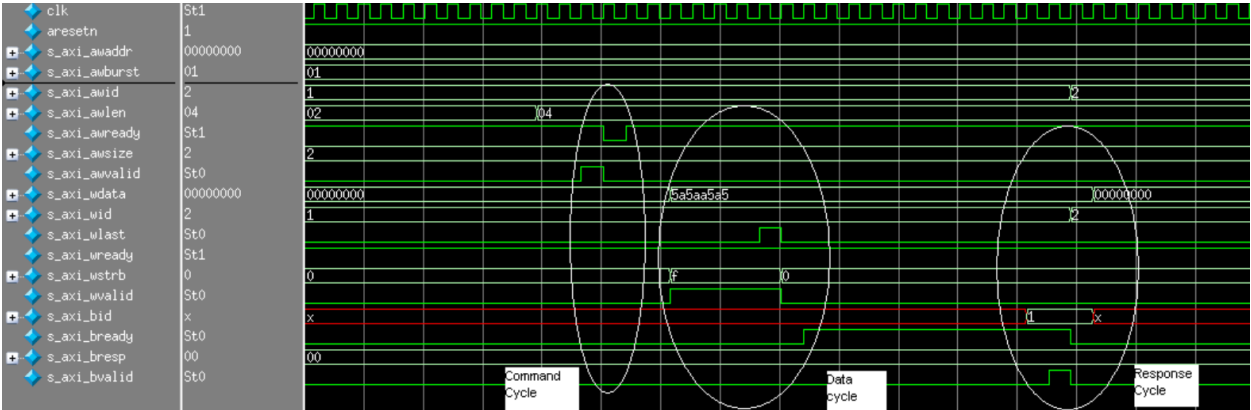


Fig. 5: AXI Write Cycle in Simulation

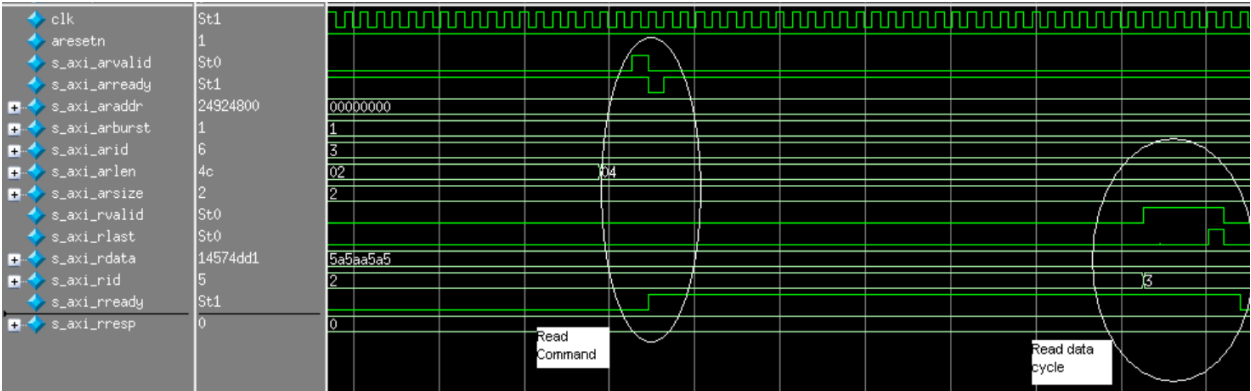


Fig. 6: AXI Read Cycle in Simulation



## 2.4 AXI Interconnect vs. SmartConnect

AXI is a very flexible standard in that it only outlines the interface itself, never requiring a designer to memorize multiple configurations for every scenario. This characteristic is exemplified with the introduction of the AXI Interconnect IP.

The AXI Interconnect IP is similar to an operating system in that both mediate data and resource transactions between two independent entities. The Interconnect IP is made up of a combination of arbiters, decoders/routers, multiplexers, and other logic elements that seamlessly adapts to any AXI system, whether it be a multi-master system connected to one slave, a multi-slave system connected to one master, or multiple masters connected to multiple slaves (up to 16 each).

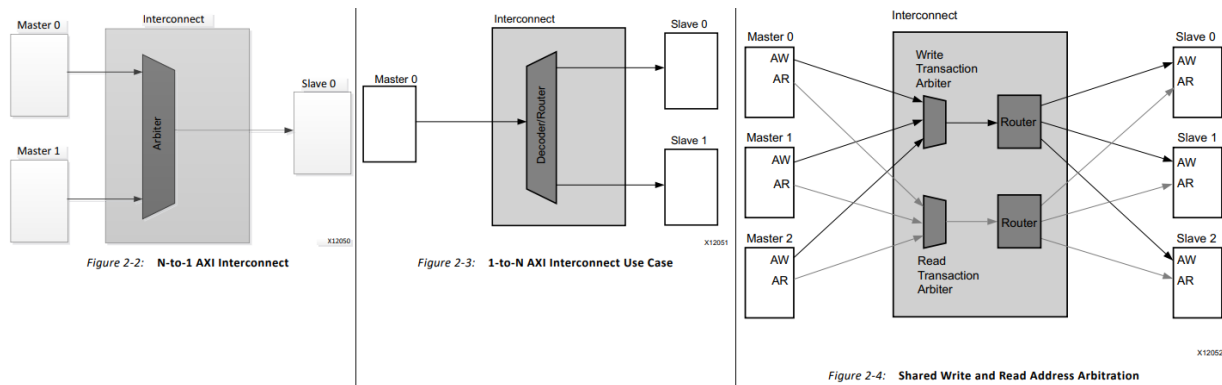


Fig. 7: AXI Interconnect Configurations<sup>2</sup>

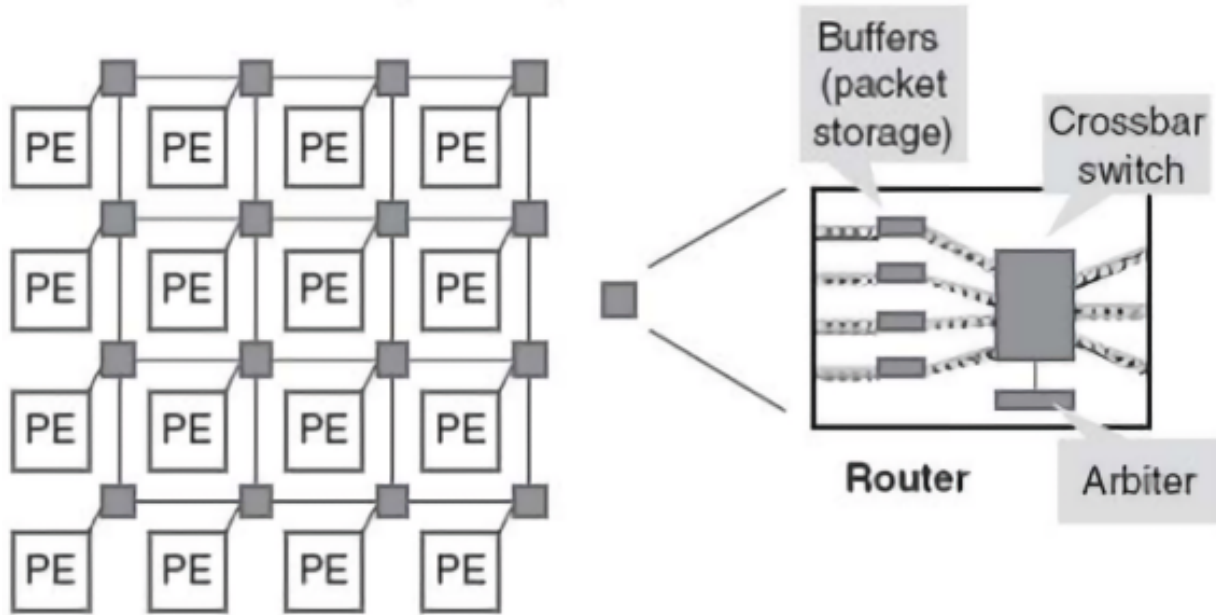
The AXI Interconnect is also known as a network-on-chip (NoC). There are many technical details about NoCs that we will not go into here, such as topology and routing strategies, but the only characteristic to keep in mind is that NoCs use packets, not wires, to route data from the source to the destination. While traditional Verilog instantiates connections between modules as wires and sends electrical signals as communication, NoCs like the Interconnect or SmartConnect utilize the AXI protocol to route signals and data payloads from the master to the appropriate slave device. The NoC architecture has multiple routers connected by wires or links with an array of processing elements or PEs built on a mesh topology. This creates a scalable architecture that has a higher bandwidth than connecting each module individually.

For a multi-master/slave system, the Interconnect will contain multiple arbiters and routers so that each write and read channel has a dedicated connection between masters and slaves — by doing this, both reads and writes can occur simultaneously. This is also known as an **AXI Crossbar** core. A typical Interconnect transaction would occur in this manner:

- As always, the master first provides the address for a write transfer onto the AW channel. The write transaction arbiter decides which master can monopolize the Interconnect Write channel and sends the master's address to the router.
- Using a preallocated address decoding table, the router then decodes the given address and selects the proper slave to write the address to. The typical AXI write transaction then commences, with an Interconnect multiplexer mediating a data transfer between master and slave.
- At the same time, another master can provide a different address to read from on the AR channel. The read transaction arbiter can also decide which master monopolizes the Interconnect Read channel, sending that master's address to a different router.

<sup>2</sup> AXI Interconnect documentation from Xilinx [here](#).

<sup>3</sup> From Sudeep Pasricha (Colorado State), Nikil Dutt (UC Irvine) "On-Chip Communication Architectures", Morgan Kaufmann, 2008

Fig. 8: AXI Interconnect NoC topology<sup>3</sup>

- This second router also decodes the given address and selects either the same or a different slave to read from. An AXI read transaction then starts with another Interconnect multiplexer as a mediator.

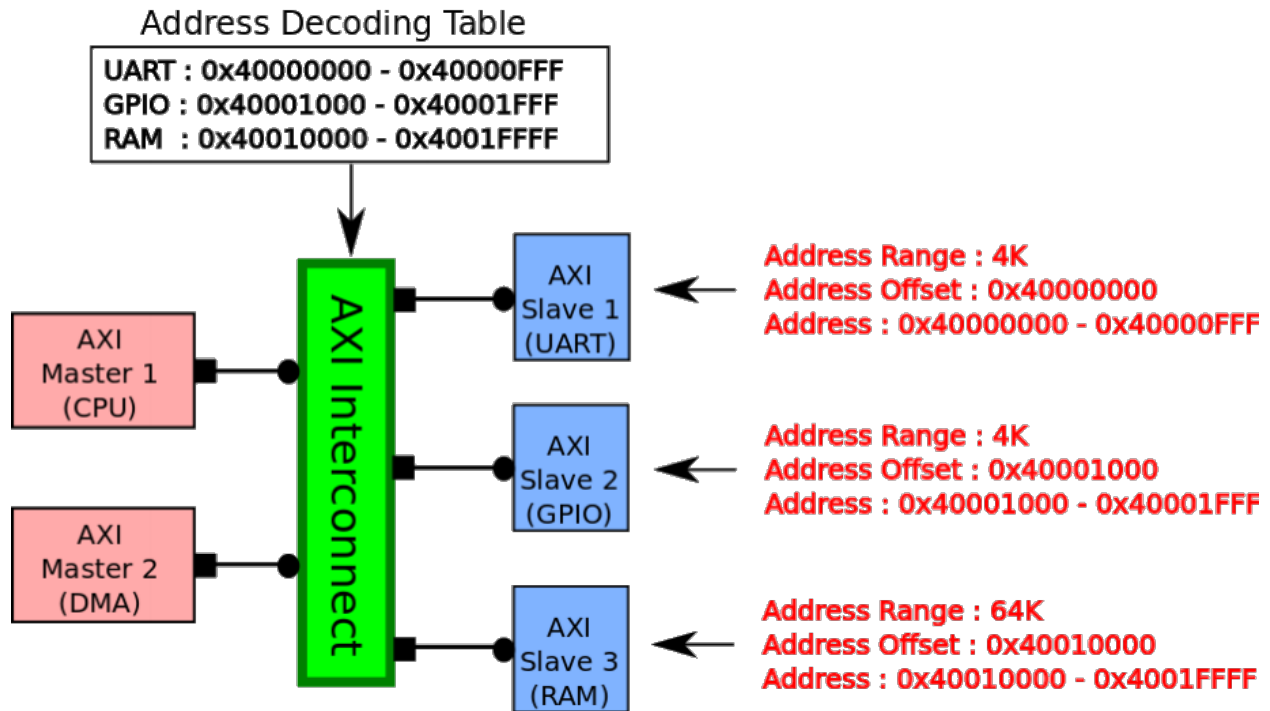
The Interconnect IP works on a round-robin basis, wherein the read and write channel will alternate for access if multiple masters are trying to write to/read from the same slave. Since the DDR protocol allocates a few clock cycles between reads and writes, the IP cannot immediately switch channels. Because slaves can queue multiple transactions and this round-robin schedule allows for out of order transfers (a slave device can respond to a master whenever), a deadlock risk emerges. Deadlock is a scenario where a transfer #1 cannot fully complete while transfer #2 is running. To finish, #2 requires transfer #1 to complete first, creating an endless loop that is never resolved. You can read more about the Dining Philosophers Problem [here](#).

From the Xilinx's Interconnect Documentation<sup>2</sup>, we can see how a deadlock situation can form:

1. Master *M1* reads from Slave device *S1* using *ID0*.
2. Master *M1* then reads from Slave device *S2* using the same ID thread *ID0*.
3. Master *M2* then reads from Slave device *S2* with a different ID *ID1*.
4. Master *M2* then reads from Slave device *S1* using the same ID thread *ID1*.
5. Slave *S1* responds to Master *M2* first. It is allowed to respond to *M2* before *M1* first, since the two Masters have different IDs. However, the AXI Crossbar cannot pass the response to *M2* because Master *M2* must first receive its response from Slave *S2*.
6. Slave *S2* responds to Master *M1* first without re-ordering. But the AXI Crossbar cannot pass the response to Master *M1* because *M1* must first receive its response from Slave *S1*, resulting in a deadlock situation.

Helpfully, the AXI Interconnect IP already resolves this concern by mandating the “Single Slave per ID” rule, where generally only one master device can talk to any slave at any given time. With this in-order rule, the Read transaction in step 2 from *M1* to *S2* is stalled until *S1* completes its response to *M1*. Similarly, the transaction between *M2* and *S1* in step 4 is stalled until *S2* completes its response to *M2*. This is important to keep in mind as the AXI protocol itself

<sup>4</sup> The example of Interconnect Addressing from Mohammadsadegh Sadri, PhD, can be found in this [post](#).

Fig. 9: AXI Interconnect Address Decoding Table<sup>4</sup>

has no in-order check between Read and Write transactions, meaning that deadlock can occur elsewhere, especially when combining multiple Interconnects and SmartConnects together.

The Interconnect also can update AXI3 interfaces to AXI4, perform bus-width conversion, use input/output FIFOs and register slices to break down timing paths, and convert between different clock domains. Simply put, the Interconnect IP is a versatile core that allows a designer to utilize the AXI protocol to its fullest extent without diving deep into the technical minutiae.

However, at the time of writing, the AXI Interconnect v2.1 core has been obsoleted by the new AXI SmartConnect IP. The SmartConnect operates on the same AXI4 principles of the Interconnect IP, providing similar performance with better optimization and a more streamlined experience. The AXI SmartConnect supports wider addressing and multi-threaded traffic along with a myriad of other benefits, so while Xilinx notes that pre-existing designs with the Interconnect v2.1 core do not need to upgrade, new designs are recommended to use the SmartConnect core moving forward. As such, our example designs will (almost) always use the SmartConnect IP as opposed to the older Interconnect. For more information, read the SmartConnect v1.0 documentation (PG247).

<sup>5</sup> From Chou, H. M., Chen, Y. C., Yang, K. H., Tsao, J., Chang, S. C., Jone, W. B., & Chen, T. F. (2015). High-performance deadlock-free id assignment for advanced interconnect protocols. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(3), 1169-1173.

<sup>6</sup> Read more about the SmartConnect IP in this [white paper](#).

Issued Order:  $T1 \Rightarrow T2 \Rightarrow T3 \Rightarrow T4$

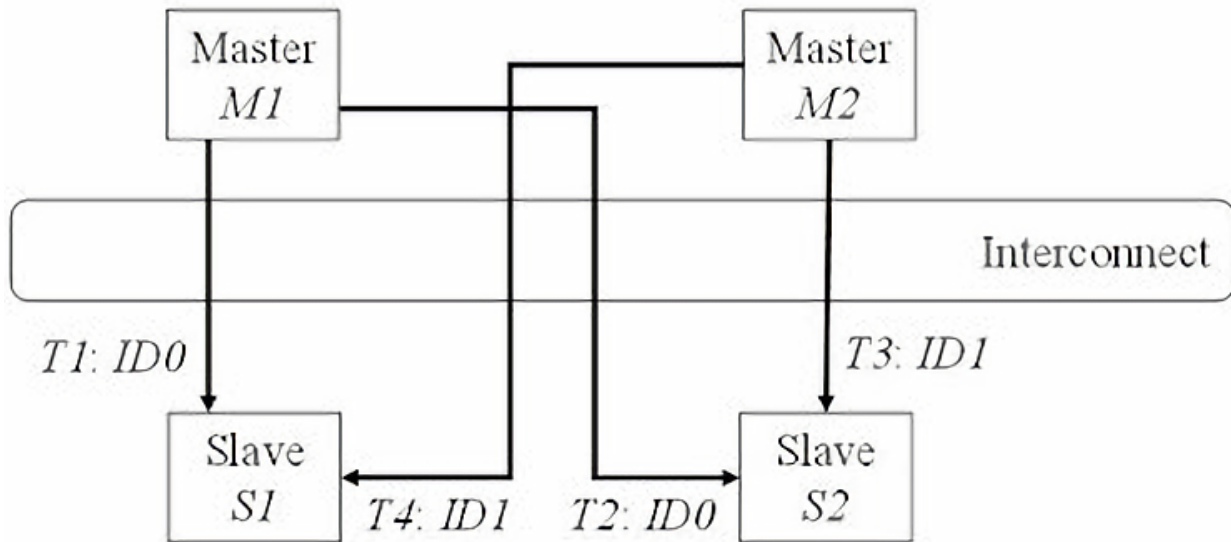


Fig. 10: An Interconnect deadlock situation<sup>5</sup>

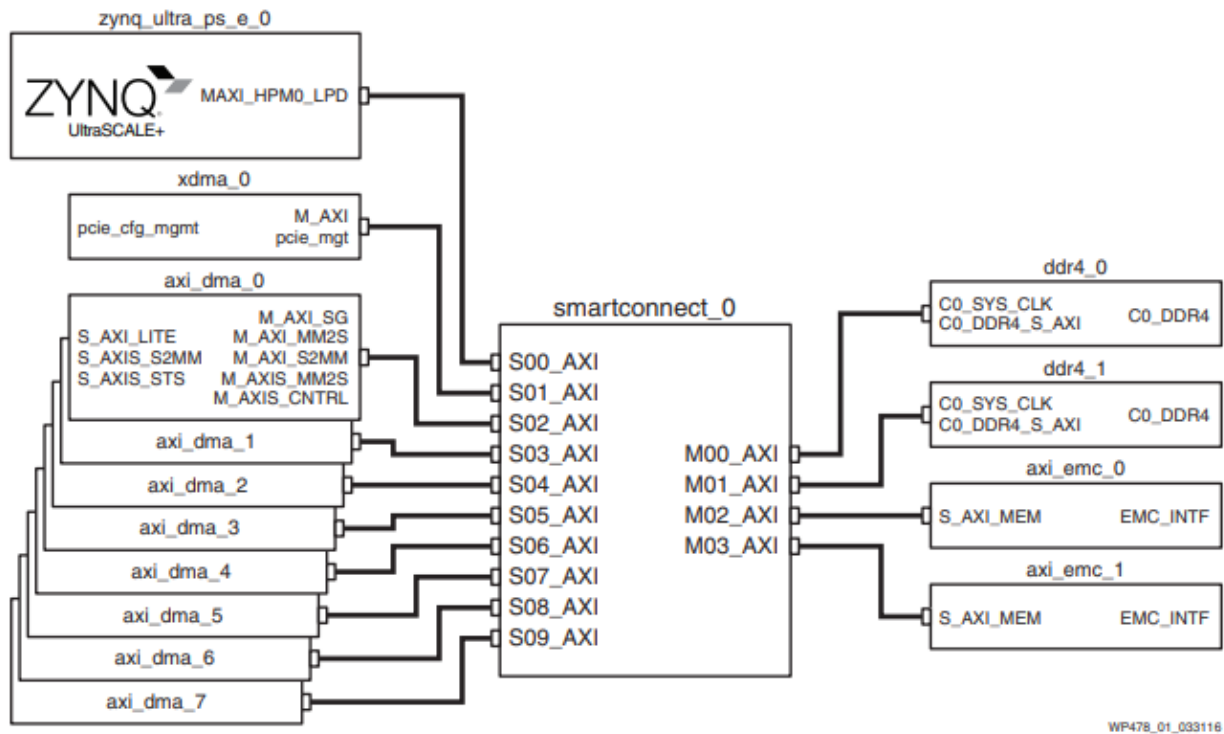


Fig. 11: Example SmartConnect IP system<sup>6</sup>

## 2.5 AXI Verification IP

With some of our example designs, we have chosen to use the AXI Verification IP or **AXI VIP** as a test DUT. The VIP, which is provided by Xilinx, is a useful AXI4 core that allows us to debug our block designs and verify for expected behavior. It is the successor to the now obsolete AXI Bus Functional Model or BFM and all new designs will use the VIP moving forward, as the BFM is no longer available. The VIP can be dropped into any design and simulate a master, slave, and pass-through device (connecting a Slave to Master). It has one (optional) active LOW reset `aresetn` which is synchronous to `aclk`. This IP is mainly for simulation and is not synthesized. We will be using the VIP to verify data transactions in simulation and overall it is a good introductory method for catching errors in any custom AXI IPs (although the VIP suite is prone to missing some background transfer errors). While setting up the emulation environment and custom DUTs, we will be using the VIP to monitor and generate AXI transactions, as well as check for protocol compliance.

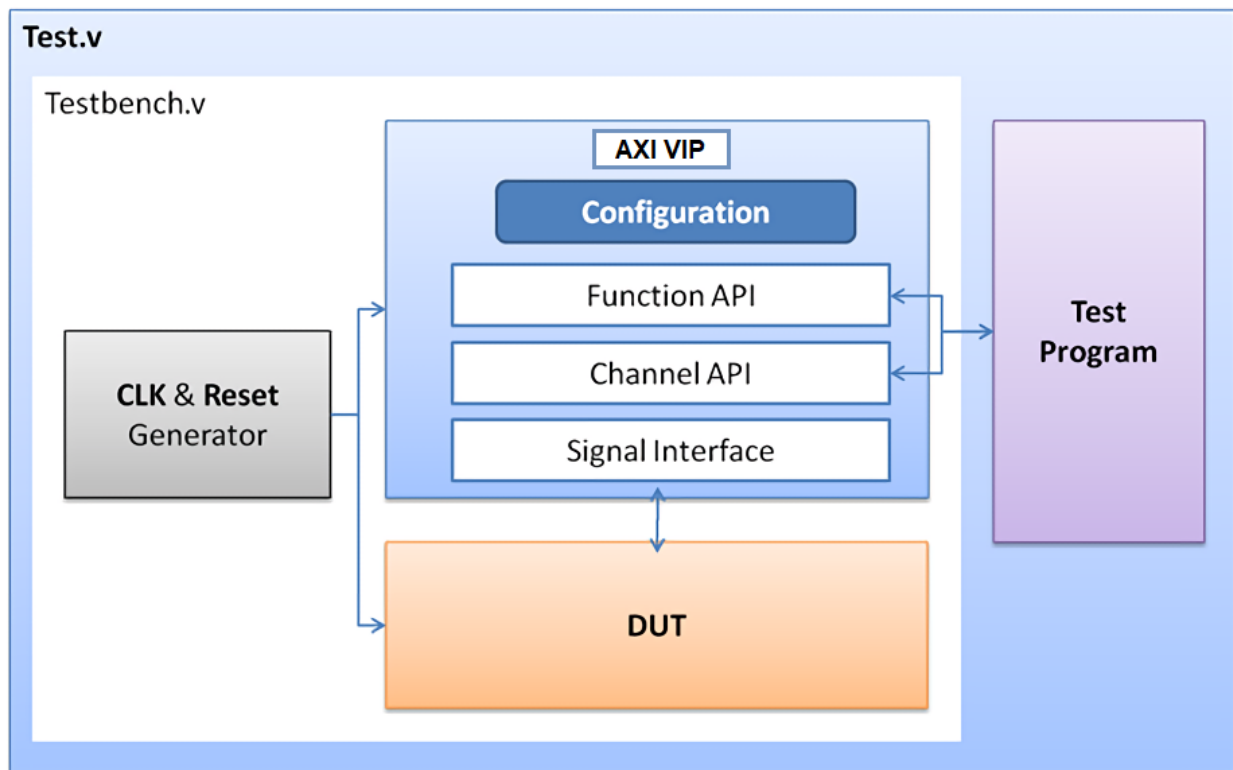


Fig. 12: Example AXI system with VIP<sup>7</sup>

## 2.6 References

<sup>7</sup> More about AXI BFM architecture [here](#) (modified image).



## LEGACY PCI AND PCI EXPRESS

---

**Important:** This article covers the PCI and PCIe buses. You can read about the specific AXI PCIe MM IP [here](#).

---

### 3.1 Peripheral Component Interconnect

If you ever have built or looked inside a desktop, you may have noticed that a few boards connected to the motherboard all use the same interface. Since the 1990's, computers have utilized the *Peripheral Component Interconnect* or **PCI** bus to attach additional components and hardware devices via expansion slots. Modern examples include graphics processing unit (GPU) boards and enthusiast-grade sound cards.



Fig. 1: An example PCIe expansion card with x16 width

The PCI format allowed computer manufacturers to build and sell peripherals with a standardized format that was compatible with all motherboards and independent of any processor's requirements, essentially creating a plug-and-play system. The legacy PCI bus and its upgraded version **PCI-X** were eventually made obsolete by their successor

**PCI Express**, which we currently use in 2021.

Although many PCI peripherals are now integrated into the motherboard itself or available as external USB devices, the PCI bus remains relevant today due to its widespread usage among industrial and enthusiast fields. Since PCIe is backwards compatible with its older standards and builds upon PCI concepts, we will first examine the fundamentals of the legacy PCI standard.

## 3.2 PCI Overview and Background

As a component bus (a bus is essentially communication between multiple components), all PCI concepts revolve around maintaining and processing transactions between peripherals and the processor (CPU). The PCI bus can be thought of as a traffic light, controlling the flow of data transactions between devices while other peripherals must wait for their turn to use the PCI bus. The external PCI bus works in tandem with the system bus, which allows internal computer components like the CPU or RAM to communicate with each other, and other external buses like the Universal Serial Bus, such as connecting a printer using an external USB cable.

Legacy PCI is synchronous, in that all events occur on the edge of the computer's internal clock. A device would begin a data transaction and specify a start memory address, taking one clock cycle. Sending the data itself would take multiple cycles until the transaction finished, at which point the connection was ended.

The  
PCI  
bus  
op-  
er-  
ates  
on  
a  
mas-  
ter-  
slave  
re-  
la-  
tio-  
ship,  
where  
the  
Bus  
Mas-  
ter  
is  
the  
agent  
that  
ini-  
ti-  
ates  
the  
trans-  
ac-

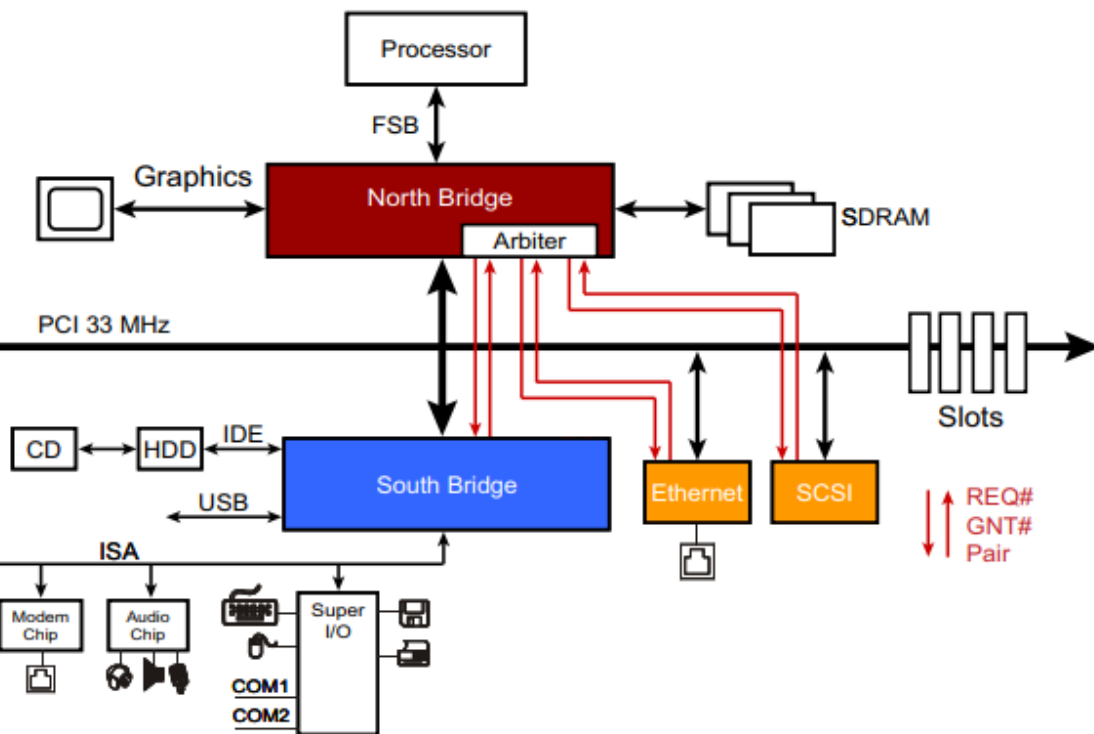


Fig. 2: Abstraction of a computer's core architecture

tion (either the CPU or PCI boards) and the slave is the Target Device. Many modern devices have Bus Mastering capabilities, so an example transaction could involve a keyboard acting as a Bus Master to write data into main memory (the Target Device). This relationship can be further elaborated upon by looking at a motherboard's core architecture.



The northbridge or host bridge represents the core logic chip on the PC's motherboard that is 'north' of the central PCI bus. The northbridge interfaces between the CPU, system memory, the AGP (graphics) bus, and high-speed PCI bus. The northbridge's Bus Arbiter receives requests from all initiators (Bus Masters), decides which requester should monopolize the PCI bus, creates the transaction channel between the initiator and Target Device, and assigns a ready pin (GNT#).

The southbridge serves as the I/O controller hub, hosting all system I/O and connecting the PCI bus to external peripherals. It also provides system signals like reset, clock, and error reporting. From the southbridge, Bus Masters representing a peripheral can submit a request to use and monopolize the PCI bus. Each Bus Master has a pair of pins that they can request with (REQ#) or know when the bus is available to use (GNT#).

### 3.3 The Legacy PCI Bus Cycle

As mentioned before, Legacy PCI is synchronous as in events occur on clock edges. To illustrate this bus cycle better, we will examine a simple example of a typical bus transaction. Rising clock edges are marked with dotted lines whenever signals are driven or sampled.

---

**Note:** Signals with # are active LOW, signals without are active HIGH.

---

- CLK Edge 1 - FRAME# (Bus Access) and IRDY# (Initiator Ready for data) are inactive, so PCI bus is idle.
- GNT# is active, showing that bus arbiter has selected this device to be the next initiator / Bus Master
- CLK Edge 2 - FRAME# is asserted by the initiator, indicating that a new transaction has started.
- Initiator drives address and command for the transaction, other devices on bus will decode address to determine if they are being requested
- CLK Edge 3 - Initiator indicates that it is ready for data transfer by asserting IRDY# to active low
- Arrow on AD bus shows that bus is undergoing turn-around cycle as ownership of signals changes (initiator drives address but also reads data on same pins)
- TRDY# is not driven low on the same edge as AD changing to avoid possibility of both buffers trying to drive a signal simultaneously, which can cause damage from shared signals
- CLK Edge 4 - Device on bus has recognized requested address and has asserted DEVSEL# (device select) to proceed with transaction
- Also asserts TRDY# (target ready) to drive first part of read data onto AD bus
- Since both IRDY# and TRDY# are active at the same time, data begins transferring on that clock edge
- Initiator knows how many bytes will eventually be transferred, but target does not, so the target must check FRAME# to see if it is still asserted or not (will become inactive when done)
- CLK Edge 5 - Target is not ready to deliver next set, so it de-asserts TRDY# for one clock cycle and enters a Wait State
- CLK Edge 6 - Second data item is transferred, and since FRAME# is still asserted, the target knows that the initiator is still requesting for more data
- CLK Edge 7 - Initiator forces a Wait State, allowing device to pause a transaction and either quickly fill or empty a buffer without stopping the request
- Often very inefficient as they will both stall their current transaction and prevent bus access to other devices
- CLK Edge 8 - Third data set is transferred, FRAME# is de-asserted so transaction is finished, at CLK edge 9 all control lines are turned off and bus becomes idle again

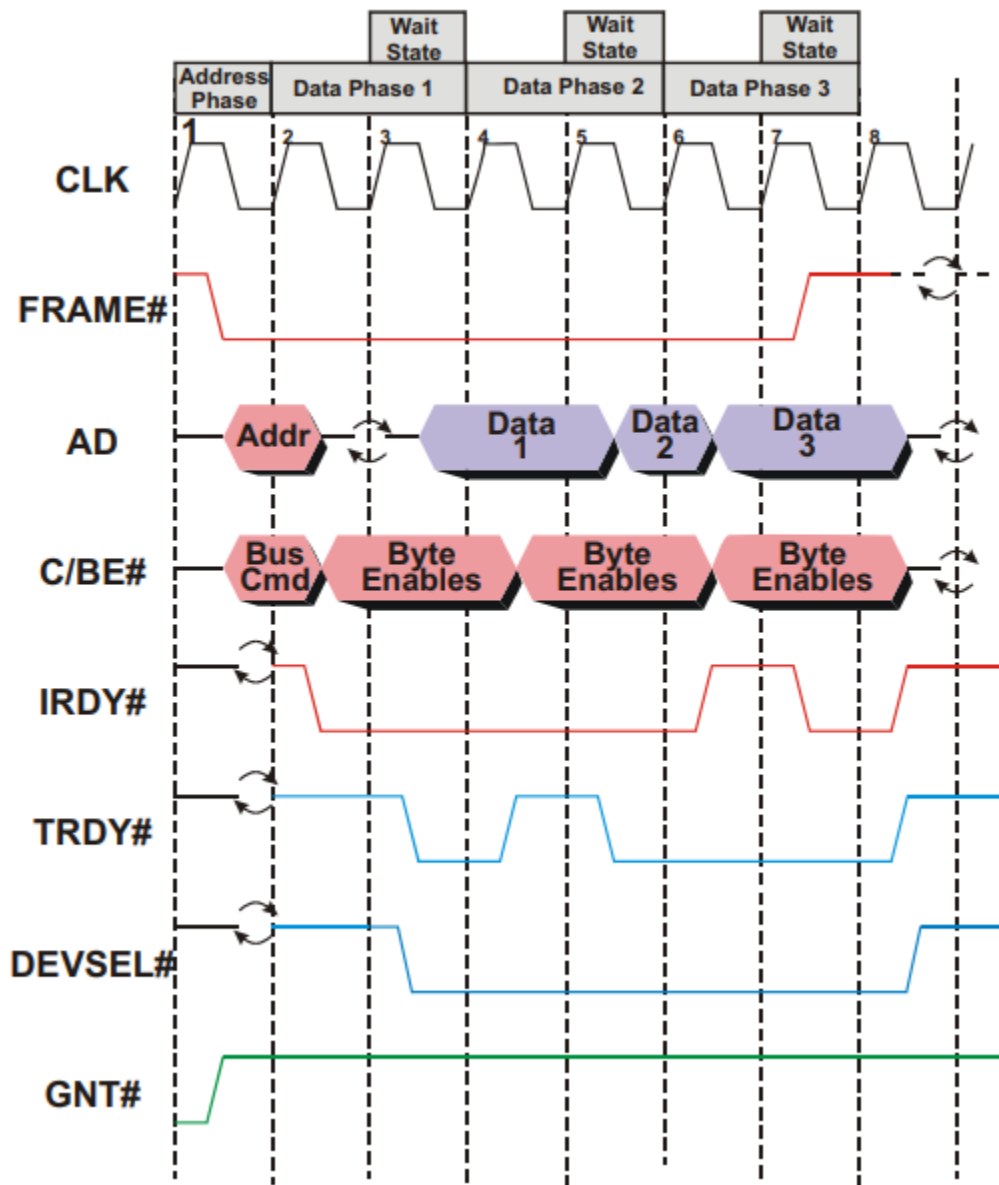


Fig. 3: Example Legacy PCI bus cycle

## DDR MEMORY AND SDRAM

### 4.1 What is RAM?

Inside any computer, phone, tablet, or other electronic device, it is almost guaranteed that there will be some sort of RAM inside that device. In fact, if you have ever looked inside a PC computer before, then you have most likely seen what a physical piece of RAM looks like. Below is an example of a DDR3 SDRAM component, and these can be purchased at most electronics stores (see Figure 1). The DDR stands for “Double-Data Rate”, and the SDRAM stands for “Synchronous Dynamic RAM”. These are just terms used to describe the process of how the RAM stores data, and they will be explained in more detail further down.

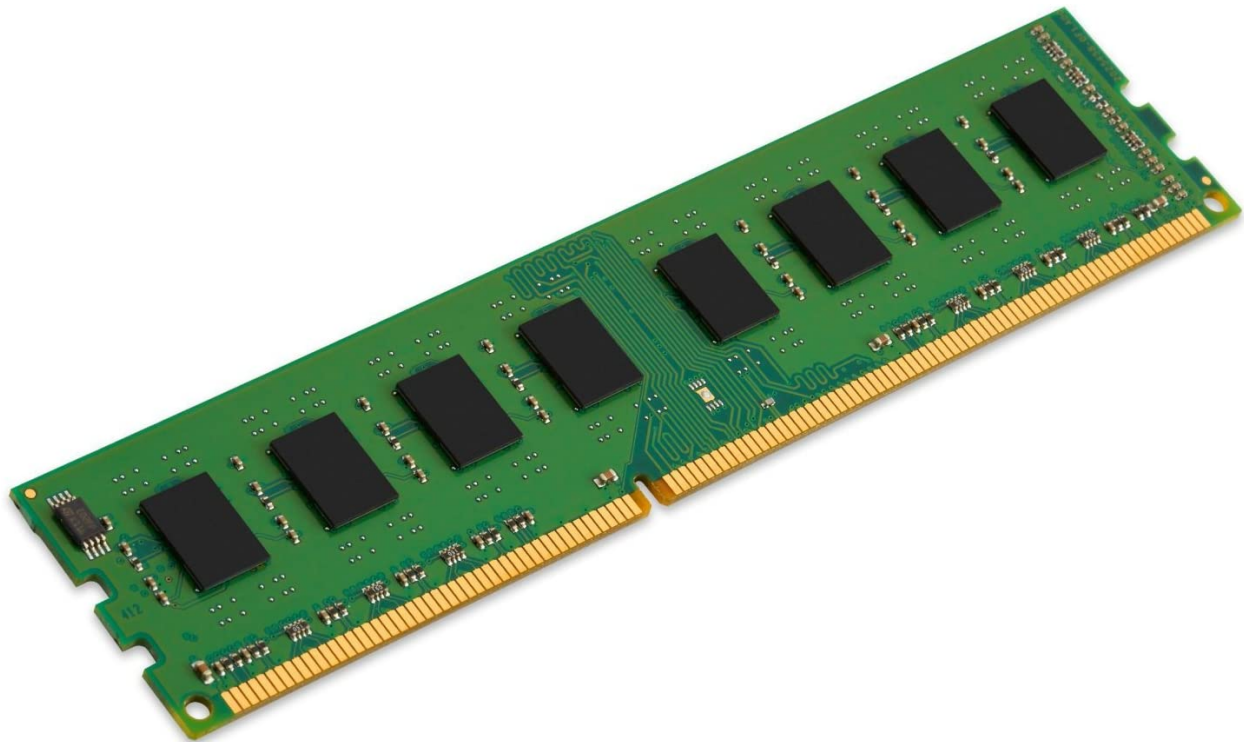


Fig. 1: Figure 1: Example DDR3 SDRAM Component

RAM stands for “Random-Access Memory”, and like all memories, it’s main purpose is to store information for future use. In your computer, for example, you should be able to check how much RAM is currently installed by going to your device settings. The more RAM that your computer has, the faster that it will be able to load programs and applications.

Almost all RAM components are “volatile” memory devices, which means that any stored data will be lost once the RAM loses power. Because of this, RAM is often used to store temporary data, such as program or application data. For more permanent storage of information, another type of memory such as “Read-Only Memory” (ROM) can be used instead. A good example of ROM is a CD disk, which is initially programmed with data (music, video, games, etc.) and is not meant to be overwritten (see Figure 2).



Fig. 2: Figure 2: Example of CD ROM

Most computers have both RAM and ROM components inside them, as RAM is needed for storing program/application data and ROM is needed for storing permanent instructions (i.e: boot-up instructions). They both have their own advantages and disadvantages, and so there is always a tradeoff between speed and volatility. While RAM components are generally much faster than ROM components, ROM has the ability to retain information even after power has been removed from the device.

## 4.2 Different Types of RAM

RAM can come in all shapes and sizes, but the two most common types of RAM are “Static RAM (SRAM)” and “Dynamic RAM (DRAM)”. While both SRAM and DRAM are effective at storing temporary data, the main difference between them lies in how each of them store this data. SRAM is referred to as “static” because it is made up solely of transistors. DRAM, on the other hand, uses capacitors to store the data. Both have their own advantages and disadvantages, and so let’s take a closer look at each type of RAM.

### 4.3 SRAM

Looking at the figure below (figure 3), we can see what a typical SRAM cell configuration looks like. The parts labeled M1 through M6 are MOSFET transistors, the line labeled WL corresponds to the “Write Line”, and the line labeled BL corresponds to the “Bit Line”. The write line and the bit line are used simultaneously to control the read and write operations of the SRAM. For example, if we wanted to write a new bit value into this SRAM cell, we would simply place the desired bit value (1 or 0), and then we would place a 1 on the write line to enable the write transaction.

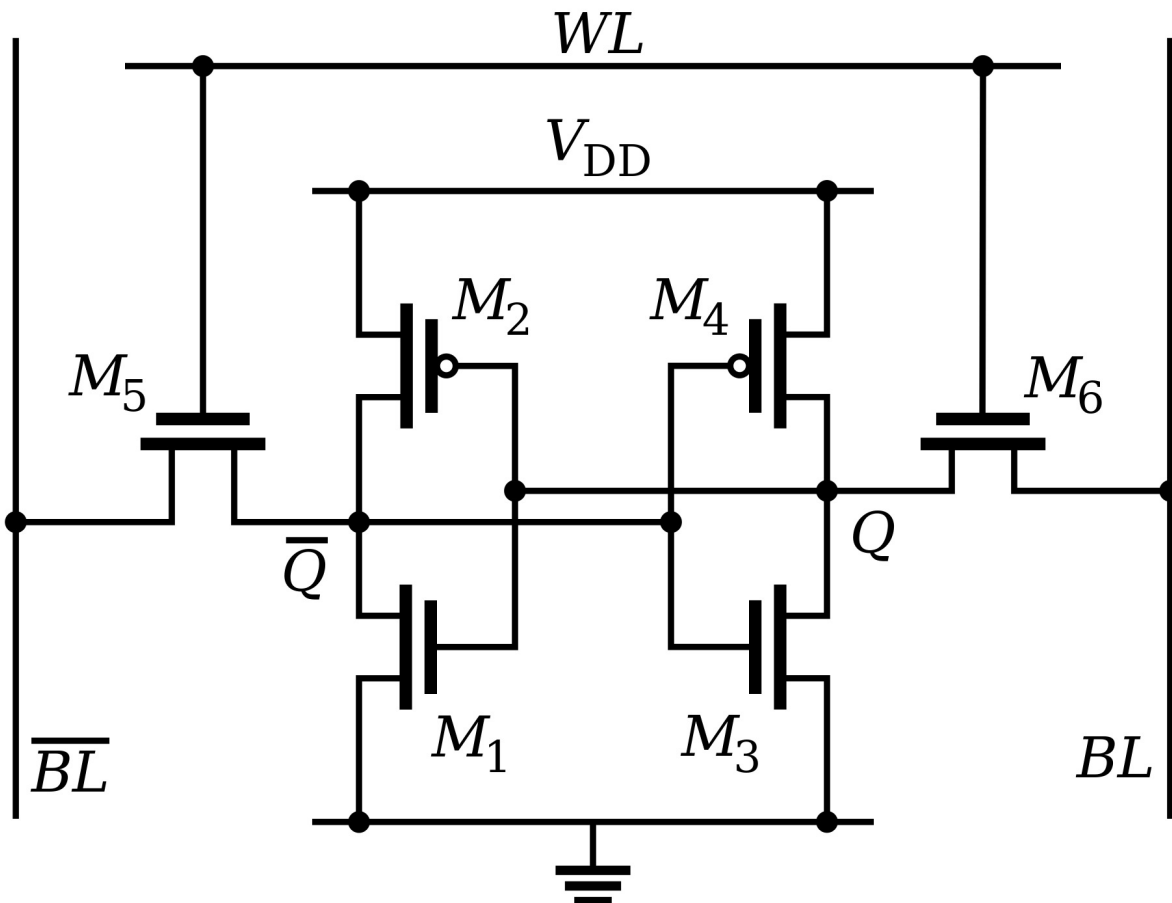


Fig. 3: Figure 3: Typical SRAM Cell Configuration

The primary advantage of using SRAM components is that they are much faster than DRAM components. However, with the advantage of being faster, they are also much more expensive to manufacture. Therefore, SRAMs are typically

only used for small amounts of memory that really need that extra speed. For example, a standard use of SRAM is for your computer's cache memory, which stores frequently-used instructions and data for faster fetching by the CPU. Have you ever noticed that after you restart your computer, it takes slightly longer to load any given website? This is because your computer has information stored inside its cache that allows the website to load faster, and when you restart your computer, you are also clearing that cache memory.

## 4.4 DRAM

In comparison to SRAM components, DRAM utilizes capacitors in order to store memory. The typical configuration of a DRAM cell can be seen below in figure 4, and as you can see, the configuration appears to be much simpler than the SRAM cell. The bitline and the wordline are still present, and they are utilized in the same way as the SRAM cell. However, there are not nearly as many transistors required for the DRAM cell, which means that the cost to manufacture a DRAM component is far less than that of an SRAM component.

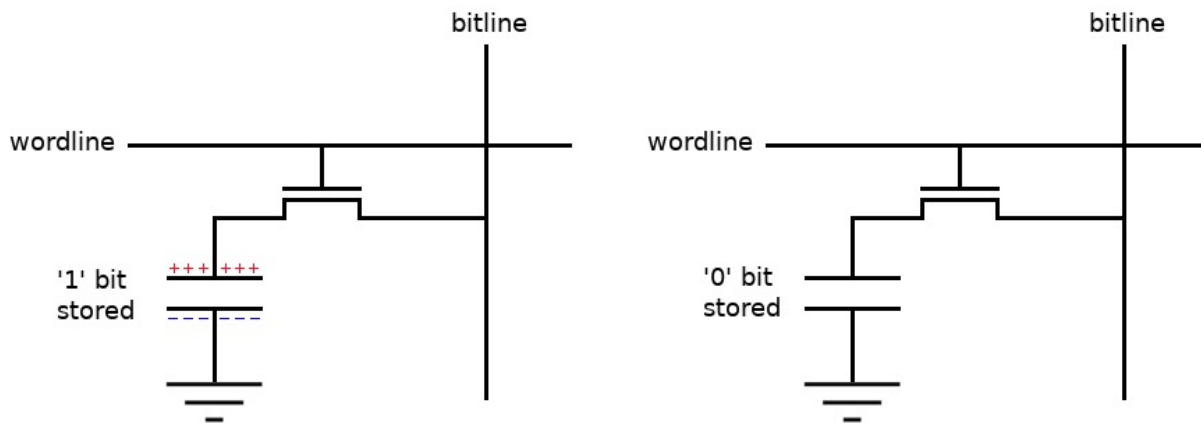


Fig. 4: Figure 4: Typical DRAM Cell Configuration

The fact that DRAM components are much cheaper than SRAM components make them a great choice for storing large amounts of data. For example, when you purchase a large piece of RAM like the illustration in figure 1, you are purchasing DRAM (the term “SDRAM” refers to Synchronous DRAM). While SRAM components like cache memory are typically in the kilobyte to low megabyte range, DRAM components can go all the way up to gigabyte range. The primary disadvantage of DRAM is that it is usually slower than SRAM, and this is due to the effects of using a capacitor. Over time, capacitors can begin to slowly discharge, and this can be very bad if it causes the stored data to be lost. In order to prevent this, DRAM components have to be constantly “refreshed”, which essentially just means that the current data values have to be re-written into the capacitors.

## 4.5 The DDR SDRAM Protocol

If you look back to our example in figure 1, you'll notice the specific memory part is called "DDR3 SDRAM". Well, now that we know what DRAM is, let's take a look at the rest of this name. First, let's look at the term "SDRAM". While this name may look similar to the Static RAM (SRAM) discussed earlier, it is actually referring to something very different. In a typical DRAM cell, there is no clock associated with the read and write transactions. However, digital electronics engineers are very fond of using clocks to keep everything synchronous in their designs, and so Synchronous DRAM (SDRAM) was created. This essentially means that read and write operations will only be processed on the rising edge of an associated clock.

Now that the read and write operations have been synchronized, a lot of the messiness has been cleaned up that could potentially occur from doing multiple asynchronous transactions. However, only sending data on one edge of a clock is rather slow, and it wastes time that could potentially be used for processing other transactions. Therefore, the Double Data Rate (DDR) protocol was created, and this process allows data to be sent on both the positive and the negative edge of an associated clock. This process can be seen below in figure 5.

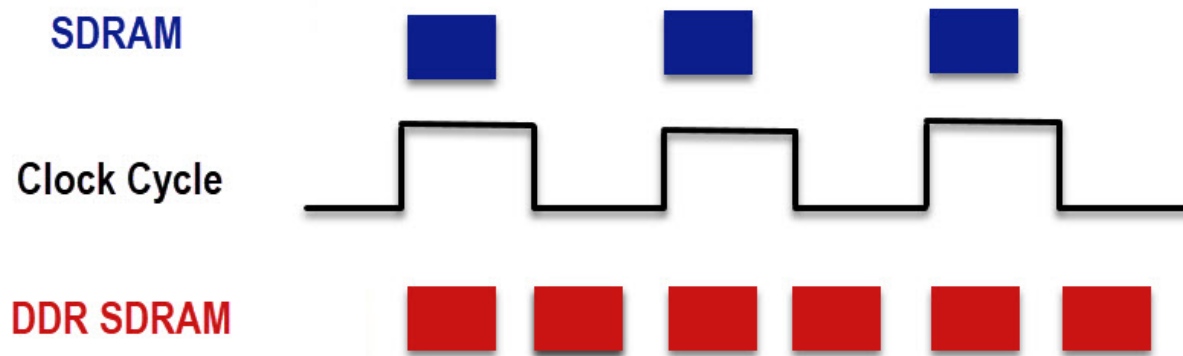


Fig. 5: Figure 5: DDR SDRAM vs SDRAM Protocols

There have been many variations of the DDR SDRAM protocol, and the term "DDR3" refers to the third generation of this protocol. At this current time in 2021, DDR5 is the most current and up-to-date DDR protocol, and it was released in July of 2020. However, DDR3 and DDR4 are both still used widely in electronics today.

Specific information about timing, signals, resets, and more can all be found in the [DDR3 SDRAM High-Performance Controller User Guide](#) from Intel. For example, let's take a look at this DDR3 Timing Diagram seen below in figure 6.

At the top of this diagram is the memory clock, which is what the DDR memory device uses to clock its transactions. Then, right below the clock are the memory chip select signal, the row-address strobe signal, the column-address strobe signal, and the write enable signal. All four of these signals are used to set up or initialize the desired read or write transaction. After these signals come the memory bank bus and the memory address bus, and these point specifically to the memory location that you would like to read or write from. Finally, the memory strobe signal indicates when the data is being transferred, the data bus contains the specific data, and the memory data mask signal indicates which bytes of data should actually be transferred. For a greater description of these signals, see the attached "Table 4-6 DDR3 SDRAM Interface Signals" from the DDR3 SDRAM High-Performance Controller User Guide.



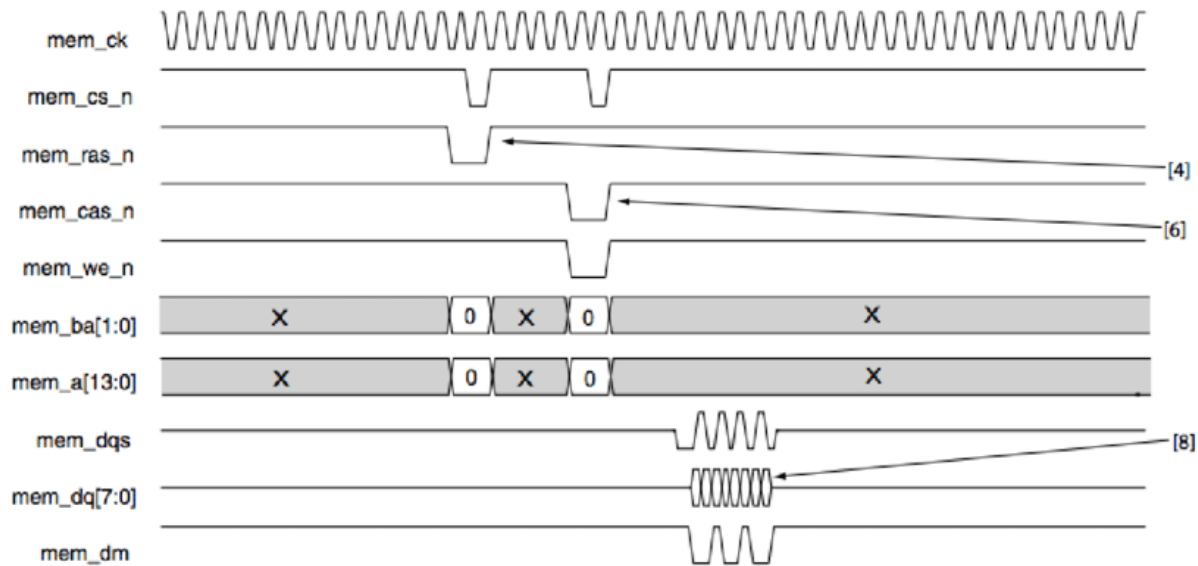


Fig. 6: Figure 6: DDR3 Timing Diagram

**Table 4–6. DDR3 SDRAM Interface Signals (Part 1 of 2)**

Signal Name	Direction	Description
mem_dq[]	Bidirectional	Memory data bus. This bus is half the width of the local read and write data busses.
mem_dqs[]	Bidirectional	Memory data strobe signal, which writes data into the DDR3 SDRAM and captures read data into the Altera device.
mem_dqs_n[]	Bidirectional	Memory data strobe signal, which writes data into the DDR3 SDRAM and captures read data into the Altera device.

**Table 4–6. DDR3 SDRAM Interface Signals (Part 2 of 2)**

Signal Name	Direction	Description
mem_clk (1)	Bidirectional	Clock for the memory device.
mem_clk_n (1)	Bidirectional	Inverted clock for the memory device.
mem_a[]	Output	Memory address bus.
mem_ba[]	Output	Memory bank address bus.
mem_cas_n	Output	Memory column address strobe signal.
mem_cke[]	Output	Memory clock enable signals.
mem_cs_n[]	Output	Memory chip select signals.
mem_dm[]	Output	Memory data mask signal, which masks individual bytes during writes.
mem_odt[]	Output	Memory on-die termination control signal.
mem_ras_n	Output	Memory row address strobe signal.
mem_reset_n	Output	Memory reset signal.
mem_we_n	Output	Memory write enable signal.



## 4.6 References



## CLOCKS, CLOCKING WIZARD, AND TIMING

---

**Note:** This page is currently under construction.

---

### 5.1 Clocks and Clock Conversion

Square wave with 50% duty cycle, can be 0 or 1 Determines how fast the design will run, drives all sequential logic (flip-flops, RAMs, FIFOs) Can have multiple clock domains in a single FPGA

### 5.2 Clock Tree

A dedicated input pin is used for clock signal, dedicated routing, logic used to minimize skew

Skew is difference in time between when it arrives at one FF to when it arrives at another FF, skew should be as small as possible

Clock tree network distributes the clock via dedicated routing signals to all FFs within the design

### 5.3 Multiple Clocks in an FPGA

Best to avoid multiple clocks for beginners

Usually only need different clocks if interfacing to some external component that requires it Ex. SDRAM, camera, special sensors Can use phase-locked loop (PLL), takes in reference clock to branch off into a different frequency

Typically use only one clock and Clock Enable signal Ex. UART has 19200 baud rate but don't need dedicated 19.2 kHz clock, just run a 50 MHz clock in intervals with counter

Never drive the clock of a FF off the output of another FF Use one central clock and parse through data with Clock Enable pulses Ex. input clock of 40 MHz, ADC runs of 10 Mhz, divide input signal by 4 and pulse once during output

## 5.4 Propagation Delay

Amount of time it takes for a signal to travel from a source to a destination Rule of thumb: signals can travel one foot of wire in one nanosecond Physical length of wires on board can be over a foot long, meaning that every portion of logic will take some finite delay time Propagation delay directly relates to sequential logic driven by a clock

Amount of time it takes from the output of one FF to travel to the second FF is the propagation delay The further apart or the more logic between the two FFs, the longer the delay, and the slower the clock is able to run Both FFs use the same clock, output of first FF at clock edge 1 should drive the second at clock edge 2 2 FFs that are 10 ns apart, a 50 MHz clock (20 ns period) will be fine while a 200 MHz clock (5 ns period) is not

FPGA timing analyzer will spot any timing errors Fix high propagation delay Slow down clock frequency Break up logic into stages through pipelining

Breaking up the logic between 3 FFs allows only half of the logic to be done between 2 FFs at a time Tools will have almost twice as much time to execute in a single clock cycle, also known as pipelining

## 5.5 Setup and Hold FF Time

Setup time - amount of time required for the input to a FF to be stable before a clock edge Hold time - minimum amount of time required for the input to a FF to be stable after a clock edge

Setup time, hold time, and propagation delay all affect FPGA design timing Minimum period of FPGA clock (and frequency where  $F = 1/T$ ) can be calculated through  $t_{clk}(\min) = t_{su} + t_h + t_p$  Generally, setup and hold time are fixed for FFs, so propagation delay is variable The more logic, the longer the propagation delay will be and the higher the clock period will be, leading to a slower frequency

If there are setup or hold time violations, the FF output is not guaranteed to be stable (could be 0, 1, or something else), also known as metastability Can check for metastability through placing and routing and timing analysis

## 5.6 Metastability Prevention

## 5.7 Clock Domain Crossing (CDC)

## LINUX DRIVERS, KERNEL PROGRAMMING, AND YOU

### 6.1 What is a Driver?

Software drivers play a critical role in how we use computers and electronics on a daily basis, yet most users never consider the complexity of driver development. As a very brief introduction, we will conceptualize a driver as any software component that lets the operating system or OS (such as Linux or Windows) communicate with an external device, like a keyboard. These devices can represent either physical hardware or other software tools. A driver allows user applications to interact and exchange data with other devices through the OS. For a more in-depth introduction to drivers and driver development, read this quick article from Microsoft [here](#).

Device drivers are parts of the operating system that facilitate the usage of hardware devices via certain programming interfaces so that software applications can control and operate the devices. As each driver is specific to a particular operating system, you need separate Linux, Windows, or Unix device drivers to enable the use of your device on different computers (this is why a career in driver development and embedded systems is often lucrative).

The first step in driver development is to understand the differences in the way each operating system handles its drivers, underlying driver model, and architecture it uses, as well as available development tools. For example, the Linux driver model is very different from the Windows one. While Windows emphasizes abstraction and separation between drivers and the host OS, Linux device drivers are often embedded within the OS kernel itself, as they are not built off a stable API. Each of these models has its own set of advantages and drawbacks, which is important to keep in mind while writing and analyzing drivers for each major OS.

Xilinx's DMA PCIe Drivers are available for both Windows and Linux. However, throughout this article and subsequent software tutorials, we will focus on the Linux OS and similar Unix distributions for its versatility and open-source nature.

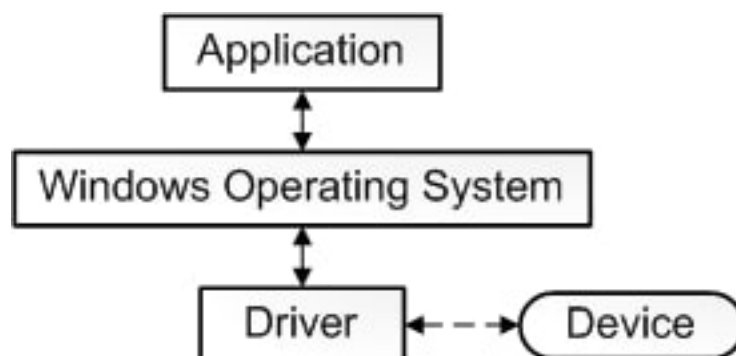


Fig. 1: Simple Block Diagram of a driver<sup>1</sup>

---

<sup>1</sup> Driver introduction from Microsoft [here](#).

Throughout this article, we will constantly reference the Linux kernel, kernel mode, and virtual memory. More information about these topics can be found in this [article](#).

## 6.2 Linux Driver Architecture and APIs

Linux is an open-source operating system, thus the entire source code of Linux is the SDK for driver development. There is no formal framework for device drivers, but the Linux kernel includes numerous subsystems that provide common services like driver registration. The interfaces to these subsystems are described in kernel header files.

While Linux does have defined interfaces, these interfaces are not stable by design. Linux does not provide any guarantees about forward or backward compatibility. Device drivers are required to be recompiled to work with different kernel versions. No stability guarantees allow rapid development of the Linux kernel as developers do not have to support older interfaces and can use the best approach to solve the problems at hand.

Such an ever-changing environment does not pose any problems when writing *in-tree* drivers for Linux, as they are a part of the kernel source because they are updated along with the kernel itself. However, closed-source drivers must be developed separately, *out-of-tree*, and they must be maintained to support different kernel versions. Thus Linux encourages device driver developers to maintain their drivers in-tree.

Linux does not provide designated samples of device drivers, but the source code of existing production drivers is available and can be used as a reference for developing new device drivers.

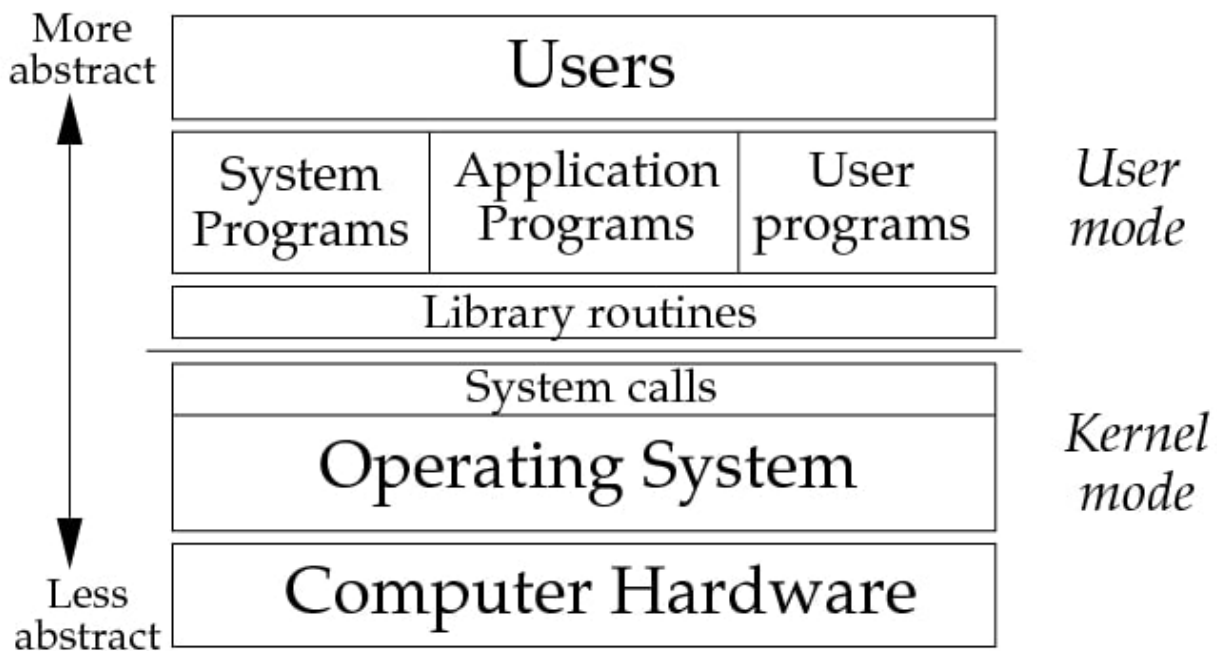


Fig. 2: Abstracted BD of a typical OS<sup>2</sup>

The core difference in the Linux device driver architecture as compared to Windows is that Linux does not have a standard driver model or a clean separation into layers. Each device driver is usually implemented as a module that can be loaded and unloaded into the kernel dynamically. Linux provides means for plug-and-play support and power management so that drivers can use them to manage devices correctly, but this is not a requirement. It is worth keeping in mind that Linux also has a Hardware Abstraction Layer or HAL above the hardware layer that acts as an interface between the actual hardware and the OS's device drivers.

<sup>2</sup> More about operating systems from this computer architecture [lecture](#).

Modules export functions they provide and communicate by calling these functions and passing around arbitrary data structures. Requests from user applications come from the filesystem or networking level and are converted into data structures as necessary. Modules can be stacked into layers, processing requests one after another, with some modules providing a common interface to a device family such as USB devices.

Linux device drivers support three kinds of devices:

- Character devices that implement a byte stream interface
- Block devices that host filesystems and perform IO with multibyte blocks of data
- Network interfaces are used for transferring data packets through the network

An important aspect of a driver is the API or Application Programming Interface it is built upon. An API is a software intermediary that allows two applications to communication with each other. Essentially, an API is a messenger that both delivers requests and subsequent responses between two applications. In the block diagram above, each layer provides an API as a set of functions/commands that the layer itself provides. We are mentioning APIs to create the distinction between drivers and APIs: drivers are low-level sections of code that run within the OS kernel itself and allow us to talk to hardware directly, while APIs are higher-level abstraction that allow us to utilize drivers within a human-understandable programming environment. APIs are often used in applications that are outside the scope of this article, so an overview of APIs can be found [here](#).

## 6.3 Linux Kernel Modules

At a module's initialization, the Linux device driver lifetime is managed by the kernel module's `module_init` and `module_exit` functions, which are called when the module is loaded or unloaded. They are responsible for registering the module to handle device requests using the internal kernel interfaces. The module has to create a device file (or a network interface), specify a numerical identifier of the device it wishes to manage, and register a number of callbacks to be called when the user interacts with the device file.

On Linux, user applications access the devices via file system entries, usually located in the `/dev` directory. The module creates all necessary entries during module initialization by calling kernel functions like `register_chrdev`. An application issues an open system call to obtain a file descriptor, which is then used to interact with the device. This call (and further system calls with the returned descriptor like `read`, `write`, or `close`) are then dispatched to callback functions installed by the module into structures like `file_operations` or `block_device_operations`.

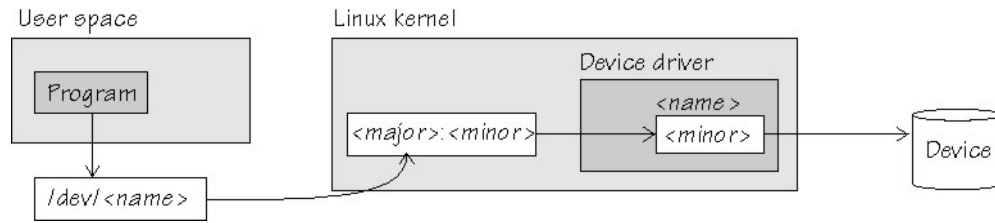
The device driver module is responsible for allocating and maintaining any data structures necessary for its operation. A `file` structure passed into the file system callbacks has a `private_data` field, which can be used to store a pointer to driver-specific data. The block device and network interface APIs also provide similar fields.

While applications use file system nodes to locate devices, Linux uses a concept of *major* and *minor* numbers to identify devices and their drivers internally. A major number is used to identify device drivers, while a minor number is used by the driver to identify devices managed by it. The driver has to register itself in order to manage one or more fixed major numbers or ask the system to allocate some unused number for it.

Currently, Linux uses 32-bit values for major-minor pairs, with 12 bits allocated for the major number allowing up to 4096 distinct drivers. The major-minor pairs are distinct for character and block devices, so a character device and a block device can use the same pair without conflicts. Network interfaces are identified by symbolic names like `eth0`, which are again distinct from major-minor numbers of both character and block devices.

---

<sup>3</sup> More about device nodes in this [IBM article](#).

Fig. 3: Major-minor Devices in Linux<sup>3</sup>

## 6.4 Transferring Data Within the Kernel

Both Linux and Windows support three ways of transferring data between user-level applications and kernel-level drivers:

- **Buffered Input-Output** which uses buffers managed by the kernel. For write operations, the kernel copies data from the user-space buffer into a kernel-allocated buffer and passes it to the device driver. Reads are the same, with kernel copying data from a kernel buffer into the buffer provided by the application.
- **Direct Input-Output** which does not involve copying. Instead, the kernel pins a user-allocated buffer in a physical memory so that it remains there without being swapped out while data is in progress.
- **Memory Mapping** can also be arranged by the kernel so that the kernel and user-space applications can access the same pages of memory using distinct addresses.

Linux provides a number of functions like `clear_user`, `copy_to_user`, `strncpy_from_user`, and some others to perform buffered data transfers between the kernel and user memory. These functions validate pointers to data buffers and handle all details of the data transfer by safely copying the data buffer between memory regions.

However, drivers for block devices operate on entire data blocks of known size, which can be simply moved between the kernel and user address spaces without copying them. This case is automatically handled by the Linux kernel for all block device drivers. The block request queue takes care of transferring data blocks without excess copying, and the Linux system call interface takes care of converting file system requests into block requests.

Finally, the device driver can allocate some memory pages from kernel address space (which is non-swappable) and then use the `remap_pfn_range` function to map the pages directly into the address space of the user process. The application can then obtain the virtual address of this buffer and use it to communicate with the device driver.

## 6.5 References



## **DMA/BRIDGE FOR PCIE DRIVERS OVERVIEW**

### **7.1 The PCIe DMA Driver**

The Xilinx PCI Express DMA IP provides high-performance direct memory access (DMA) via PCI Express. The PCIe DMA can be implemented in Xilinx 7-series XT and UltraScale devices. Xilinx Support Answer [65444](#) provides drivers and software that can be run on a PCI Express root port host PC to interact with the DMA endpoint IP via PCI Express. The drivers and software provided with the answer record are designed for Linux and Windows operating systems and can be used for lab testing or as a reference for driver and software development. Through the use of the PCIe DMA IP and the associated drivers and software you will be able to generate high-throughput PCIe memory transactions between a host PC and a Xilinx FPGA.

### **7.2 Accessing and Building the Xilinx Driver**

These steps are derived from Xilinx Support Answer [65444](#), with our suggestions added.

The current driver implementation uses the following Kernel functions and must be included in your OS kernel version. The following Linux distributions have been tested:

- Red Hat (RHEL 7)
- Fedora
- CentOS
- Ubuntu

Navigate to Xilinx's GitHub repo [here](#) and clone the repo from either the Linux CLI or by downloading the repo directly from the GitHub website. A helpful website is [DownGit](#), which will allow you to download the XDMA folder without pulling the entire repository.

On your host computer, make a temporary directory using `mkdir dma_driver` and navigate to this directory. Copy the downloaded zip file to the current directory with `cp ../Downloads/linux-kernel.zip .` (do not forget the period), and unzip the driver zip file. Navigating into `linux-kernel`, we can open the README to find out how to install the driver. Be aware that the Usage instructions are not exact and there are some additional steps required in between.

During our testing, after attempting to run the Makefile by using `sudo make install` in the `xdma` folder, we found a few errors while compiling. These errors may be fixed in the future, but as of the current XDMA driver version of v2020.1.8, these may prevent you from properly creating the driver:

- Make sure you install the dependencies `kernel-devel` and `elfutils-libelf-devel`.

```

Makefile:10: XVC FLAGS: .
make -C /lib/modules/4.18.0-240.15.1.el8_3.x86_64/build M=/home/gpello/dma_driver/linux-kernel/xdma modules
make[1]: Entering directory '/usr/src/kernels/4.18.0-240.15.1.el8_3.x86_64'
/home/gpello/dma_driver/linux-kernel/xdma/Makefile:10: XVC FLAGS: .
CC [M] /home/gpello/dma_driver/linux-kernel/xdma/xdma_mod.o
/home/gpello/dma_driver/linux-kernel/xdma/xdma_mod.c: In function 'xdma_error_resume':
/home/gpello/dma_driver/linux-kernel/xdma/xdma_mod.c:299:2: error: implicit declaration of function 'pci_cleanup_aer_uncorrect_error_status' [-Werror=implicit-function-declaration]
  pci_cleanup_aer_uncorrect_error_status(pdev);
  ~~~~~
cc1: some warnings being treated as errors
make[2]: *** [scripts/Makefile.build:316: /home/gpello/dma_driver/linux-kernel/xdma/xdma_mod.o] Error 1
make[1]: *** [Makefile:1544: _module_/home/gpello/dma_driver/linux-kernel/xdma] Error 2
make[1]: Leaving directory '/usr/src/kernels/4.18.0-240.15.1.el8_3.x86_64'
make: *** [Makefile:27: all] Error 2

```

Fig. 1: Possible Makefile error

- If you encounter this error (implicit declaration of function `pci_cleanup_aer_uncorrect_error_status`), open `xdma_mod.c` in your editor of choice and replace `pci_cleanup_aer_uncorrect_error_status(pdev)` for `pci_aer_clear_nonfatal_status(pdev)`. Save the file.
- There are other errors present in `libxdma.c` and `xdma_mod.c`. These files can be updated from this pull request [here](#).

Once all errors have been fixed (there may be more or less depending on the version, check the Github repo), make the driver with `sudo make install` and run `sudo make` in the `tools` folder. Load the driver by navigating to the `tests` folder, making the tests executables (we opted to test `load_driver` with `chmod +x 'load_driver.sh'`) and running `sudo ./load_driver.sh`. Check that the driver is loaded into the kernel with `lsmod`.

Module	Size	Used by
xdma	94208	0
binfmt_misc	20480	1
xt_CHECKSUM	16384	1
ipt_MASQUERADE	16384	3

Fig. 2: Checking the kernel modules

We can test the driver using the same process by running `sudo ./run_test.sh`. When we ran the test, we encountered an error on line 28. To fix this, open the `run_test.sh` file, and on line 28, change `if [ $channelId == "1fc" ]`; then to `if [ "$channelId" == "1fc" ]`; then. From here, you will be able to connect your physical PCIe device to the host machine and run each test to check that the host can identify the PCIe Endpoint.

```

Pushed a login request to your device...
Success. Logging you in...
Error at line 94, file reg_rw.c (2) [No such file or directory]
./run_test.sh: line 28: [: ==: unary operator expected
Error at line 94, file reg_rw.c (2) [No such file or directory]
./run_test.sh: line 28: [: ==: unary operator expected
Error at line 94, file reg_rw.c (2) [No such file or directory]
./run_test.sh: line 28: [: ==: unary operator expected
Error at line 94, file reg_rw.c (2) [No such file or directory]
./run_test.sh: line 28: [: ==: unary operator expected
Info: Number of enabled h2c channels = 0
Error at line 94, file reg_rw.c (2) [No such file or directory]
Error at line 94, file reg_rw.c (2) [No such file or directory]
Error at line 94, file reg_rw.c (2) [No such file or directory]
Error at line 94, file reg_rw.c (2) [No such file or directory]
Info: Number of enabled c2h channels = 0
Info: The PCIe DMA core is memory mapped.
Error: No PCIe DMA channels were identified.

```

Fig. 3: Error in `run_test.sh`

## CREATING A USER-FRIENDLY DUT GUI

This section focuses on creating a messaging system between a Python GUI, a C++ server, and an example C++ DUT replicating the functionality of our *FPGA AXI counter* in software instead.

### 8.1 What is a GUI?

A graphical user interface (GUI) allows a user to interact with a computer program using a mouse cursor or other pointing devices only. A GUI allows a user to interact with programs without prerequisite knowledge of the underlying system architecture. This is very useful for simplifying functions for use by a general audience.

A GUI program is very different from a program that uses a command line interface which receives user input from typed characters on a keyboard. Typically programs that use a command line interface perform a series of tasks in a predetermined order and then terminate, utilizing the Windows or Linux terminal and relying on the user's knowledge of standard Unix syntax. In contrast, a GUI program creates the icons that are displayed to a user and must wait for the user to interact with them. The order that tasks are performed by the program is under the user's control – not the program's control! This means a GUI program must keep track of its own internal state and respond correctly to user commands that are given in any order the user chooses. For example, after setting an initial value for a counter, the GUI must keep track of any subsequent changes to this value if the user decides to increment or decrement the counter.

An GUI program has the following structure:

- Create the icons and widgets that are displayed to a user and organize them inside a screen window.
- Define functions that will process user and application events.
- Associate specific user events with specific functions.
- Start an infinite event-loop that processes user events. When a user event happens, the event-loop calls the function associated with that event.

Usually, GUIs are created with higher-level programming languages like Java for their rich graphical libraries and ease of use. As such, we will use Python and the simple graphical library Tkinter to create the GUI for our test program. You can read more about GUIs and Tkinter [here](#).

## 8.2 System Overview

The purpose of this system was to build a framework that allowed for versatile communication between the end user and FPGA DUT. Xilinx's DMA/Bridge for PCIe *driver* arbitrates the mapping of physical memory to virtual memory. This allows for the FPGA board to communicate with the host machine through PCIe by writing values from physical kernel addresses into the PC's virtual memory. This framework bridges the gap between the FPGA's stored data in virtual userspace and the user themselves. There are many ways to display the FPGA's data to the user, but we chose to utilize a TCP messaging library called *ZeroMQ*. By using TCP sockets, the user will be able to write and read to/from virtual memory any amount of data, including PCIe TLPs, binary data, strings, etc.

However, one goal was to test this system without the presence of a board, as the software infrastructure should be able to operate independently without a PCIe device. To accomplish this, we also created an example DUT in C++ mimicking our simple AXI counter to simulate data transactions within virtual memory.

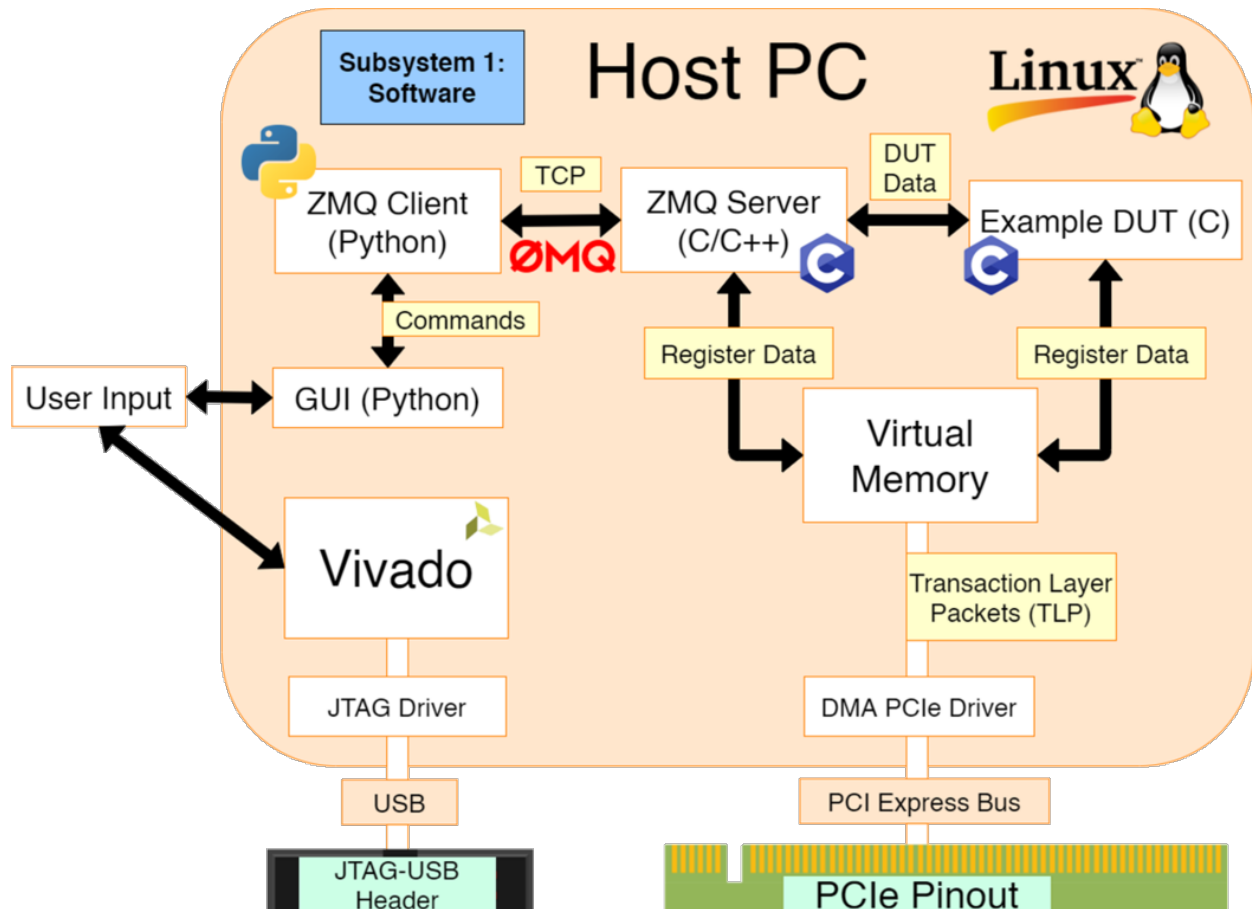


Fig. 1: Software system BD

## 8.3 Environment Setup

This setup has been tested on both Ubuntu/Debian-based distributions and CentOS/Red Hat. Steps may vary based on the Linux distribution used. Windows and MacOS not tested, although the system should function properly with all dependencies installed.

### 8.3.1 Dependencies

---

**Important:** The GUI folder can be downloaded [here](#).

---

For this system to function properly, we need to install multiple dependencies. This includes Python, ZeroMQ, CMake, and other libraries. Depending on your current environment, some steps may be skipped. Assume that all installation commands need `sudo` permission.

#### 1. Install CMake.

- a) For Debian, run `sudo apt-get install cmake g++ make`. Run `apt show cmake` to check if it has installed properly.
- b) For CentOS, run `sudo yum install cmake`. For all subsequent CentOS commands, `yum` can be substituted for the `dnf` package manager if it is already installed.

#### 2. Install Libsodium.

- a) For Debian, run `apt-get install libsodium-dev`. If this does not work, you may have to install `libsodium23` or `libsodium18`. Otherwise, download the stable tarball [here](#), unzip it, run `sudo ./configure`, `sudo make`, and `sudo make install` to install Libsodium manually.
- b) For CentOS, run `yum install libsodium`. You should enable EPEL first (check this [article](#)).
- c) For CentOS, if `yum` does not work, first run `uname -m` to check if the machine is `x86_64` or `aarch64`. Download the latest release [here](#) (our host machine was `x86_64`), run `rpm -Uvh libsodium-1.0.18-1.el7.remi.src.rpm` (this is the package filename), and then run `sudo yum install libsodium`.

#### 3. Download the rest of the necessary dependencies. For brevity, we will combine multiple packages together.

- a) For Debian, run `apt-get install libtool pkg-config build-essential autoconf automake`. Then run `apt-get install libzmq5 libzmq3-dev python3 python3-zmq python3-tk` to install ZeroMQ, Python 3, and Tkinter (if they are not already installed). You can also run `pip install pyzmq` but this is optional.
- b) For CentOS, run `yum install libtool pkg-config autoconf automake` and then `yum install python3 python3-zmq python3-devel python3-tkinter` if you do not already have Python. Also run `yum install gcc-c++` and `yum install -y ncurses-devel`. Finally, run `yum install zeromq-devel`, which should install *libsodium-devel*, *libunwind-devel*, *openpgm-devel*, and *zeromq-devel*.

**Danger:** The source file's `CMakeLists.txt` is currently configured for Debian.

- For CentOS, `libzmq.so` is found in `/usr/lib64` (different from Ubuntu), so after installing all dependencies for CentOS, open `CMakeLists.txt` and edit the line that finds the `libzmq.so` file to `FIND_FILE(ZMQLIB libzmq.so /usr/lib64)`.

- The location of `libzmq.so` will vary, so be sure to use the `find` command in the Terminal (`find /usr -name libzmq.so`).

### 8.3.2 Running the Program

1. Download and install all dependencies.
2. Download and decompress `gui.zip` into a folder.
3. Within the folder itself, make another temporary folder (this is where your C++ executable will go).
4. Navigate to this temporary folder using the terminal.
5. Compile `main.cpp` using the command `cmake ..`
  - a) If using **CentOS**, go into another folder on top (like `CMakeFiles`), and copy `zmq.hpp` into the same folder as `CMakeCache.txt` and `cmake_install.cmake`.
  - b) After running `cmake ..`, run `cmake --build ..` in the aforementioned top folder (in this case, `CMakeFiles`) and `./ZmqProject` will be generated in the previous folder (if there is trouble compiling, read this [post](#)).
  - c) If you do not do this, you may get an error about not compiling due to having no cache.
6. Run the C++ server using `./ZmqProject`.
7. In a separate terminal window, run the python script using `python3 client_tk.py`.
8. You should now see a simple blue GUI pop up.
  - a) Type an initial value into the textbox and click Start. You should see the value be set in the C++ server terminal and a reply back to the python client.
  - b) You can click the ++ button to increment the counter by 10, – to decrement the counter by 10, or Stop to close out of the program. With each command, both the server and client should respond to each other (for example, the command “increment” should be sent to the server and the client should receive a reply back that “The counter is at <num>”).

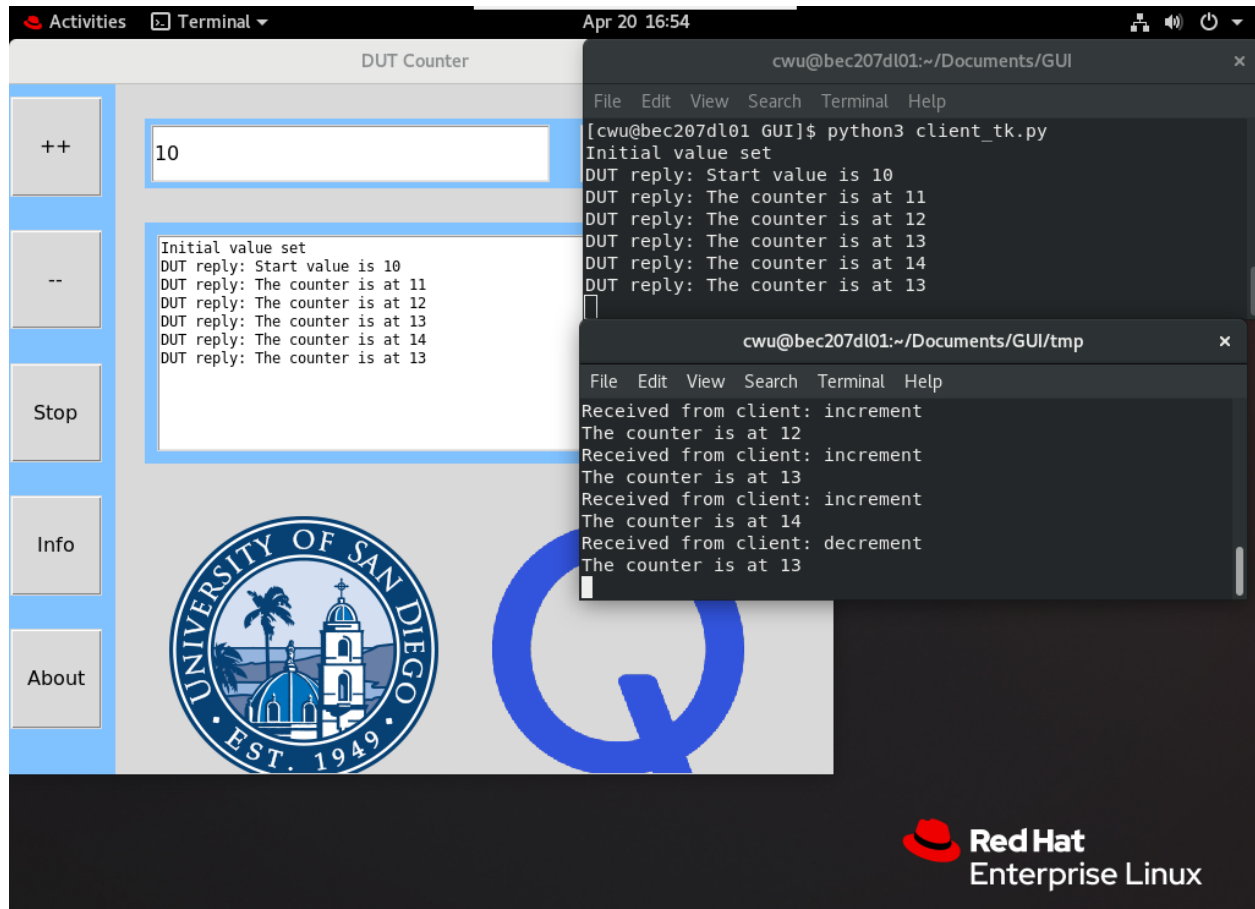


Fig. 2: Example counter using Python, C++, and ZeroMQ





## MIG 7 SERIES IP OVERVIEW

The MIG 7 Series IP is a ubiquitous core that is compatible with all 7 Series FPGAs, adding easy memory management into any design. For this article, we will discuss using the MIG with both a Kintex-7 and Virtex-7 board, such as the KC705 and VC707 respectively.

### 9.1 Customizing the IP

If using a board, be sure to select it as the project's default part before moving on. A good board to start with is the VC707, as it has ample computational power, DDR3 memory, and a PCIe interface, as well as other peripherals.





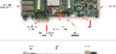
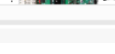
**Default Part**  
Choose a default Xilinx part or board for your project.

Parts | **Boards**

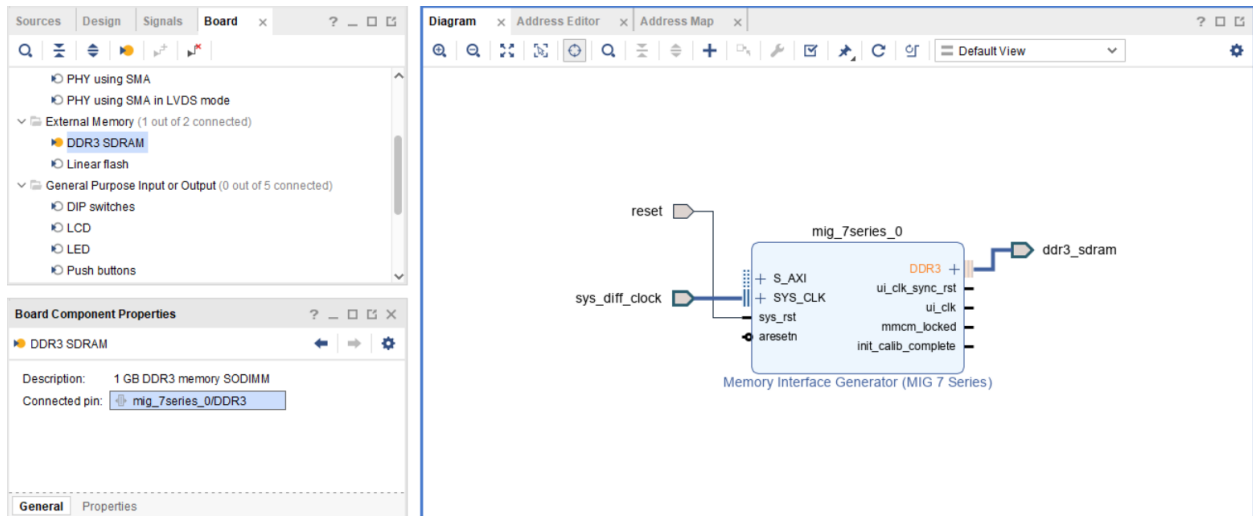
[Reset All Filters](#) Install/Update Boards

Vendor:  Name:  Board Rev:

Search:


Display Name	Preview	Vendor	File Version	Part	I/O Pin
<a href="#">Kintex UltraScale+ KCU116 Evaluation Platform</a> <a href="#">Add Companion Card</a> <a href="#">Connections</a>		xilinx.com	1.5	xcku5p-flvb676-2-e	676
<a href="#">Kintex UltraScale KCU1500 Acceleration Development Board</a>		xilinx.com	1.2	xcku115-flvb2104-2-e	2104
<a href="#">Spartan-7 SP701 Evaluation Platform</a>		xilinx.com	1.0	xc7s100fgga676-2	676
<a href="#">Virtex-7 VC707 Evaluation Platform</a>		xilinx.com	1.4	xc7vx485tffg1761-2	1761
<a href="#">Virtex-7 VC709 Evaluation Platform</a>		xilinx.com	1.8	xc7vx690tffg1761-2	1761
<a href="#">Versal VCK190 Evaluation Platform</a>		xilinx.com	2.0	xcvc1902-vsva2197-2MP-e-S	2197

Create a new block diagram (BD) and use the IP catalog to add a new IP to the BD - in this case, the “Memory Interface Generator (MIG 7 Series)” core. If using a board, a prepackaged MIG may be available. We can customize it by double clicking it.



**Important:** Unless mentioned otherwise, leave all values default.

- Make sure the AXI4 interface is enabled and select DDR3 SDRAM.
- The desired clock period must be between 2500 and 3300ps. For now, use 2500ps (400 MHz), as this is the speed of the actual physical DDR3 RAM transactions.
- Make sure the PHY to Controller Clock Ratio is 4:1 (ensuring that the physical DDR RAM will operate at 400 MHz, but the controller stays at 100 MHz i.e.,  $ui\_clk = 100\text{ MHz}$ ).
- For Kintex-7, select 8 bits as the Data Width for each address in memory. Also check that the number of Bank Machines used for managing DDR banks is set to 4. For Virtex-7, set the Data Width to 64 bits to account for the larger data bandwidth.
- At the bottom of the Controller 0 screen, make sure the memory details match 1 GB, x8, row:14, col:10, bank:3, data bits per strobe:8, with data mask, single rank, 1.5V
- The AXI Address Width is equal to the bank + row + column width =  $3 + 14 + 10 = 27$  bits wide.
- Leave the AXI ID Width at 4, as we will not use this.
- Select the Input Clock Period for PLL input clock (CLKIN) at 5000ps for 200 MHz, so that we can use the input clock as the reference clock, which must be 200 MHz. Deselect any additional clocks.
- Make sure the Memory Address Mapping Selection is set to the default configuration of Bank/Row/Column.
- Use *No Buffer* for the System Clock and *Use System Clock* for the Reference Clock, allowing us to use the system clock to also drive the reference clock at 200 MHz.
- Choose active HIGH for the System Reset Polarity.
- Import and validate the Pin Configuration file. Boards will come with preset constraints. For example, the VC707's pins are as such:
- After specifying the pinout, leave all system signals to their default values on the final page.
- Accept the T&C, generate, and save.



Pin Selection For Controller 0 - DDR3 SDRAM

	Signal Name	Bank Number	Byte Number	Pin Number	IO Standard
1	ddr3_dq[0]	39	T3	N14	SSTL15_T_DCI
2	ddr3_dq[1]	39	T3	N13	SSTL15_T_DCI
3	ddr3_dq[2]	39	T3	L14	SSTL15_T_DCI
4	ddr3_dq[3]	39	T3	M14	SSTL15_T_DCI
5	ddr3_dq[4]	39	T3	M12	SSTL15_T_DCI
6	ddr3_dq[5]	39	T3	N15	SSTL15_T_DCI
7	ddr3_dq[6]	39	T3	M11	SSTL15_T_DCI
8	ddr3_dq[7]	39	T3	L12	SSTL15_T_DCI
9	ddr3_dq[8]	39	T2	K14	SSTL15_T_DCI
10	ddr3_dq[9]	39	T2	K13	SSTL15_T_DCI
11	ddr3_dq[10]	39	T2	H13	SSTL15_T_DCI
12	ddr3_dq[11]	39	T2	J13	SSTL15_T_DCI
13	ddr3_dq[12]	39	T2	L16	SSTL15_T_DCI
14	ddr3_dq[13]	39	T2	L15	SSTL15_T_DCI
15	ddr3_dq[14]	39	T2	H14	SSTL15_T_DCI
16	ddr3_dq[15]	39	T2	J15	SSTL15_T_DCI
17	ddr3_dq[16]	39	T1	E15	SSTL15_T_DCI
18	ddr3_dq[17]	39	T1	E13	SSTL15_T_DCI
19	ddr3_dq[18]	39	T1	F15	SSTL15_T_DCI
20	ddr3_dq[19]	39	T1	E14	SSTL15_T_DCI
21	ddr3_dq[20]	39	T1	G13	SSTL15_T_DCI
22	ddr3_dq[21]	39	T1	G12	SSTL15_T_DCI
23	ddr3_dq[22]	39	T1	F14	SSTL15_T_DCI
24	ddr3_dq[23]	39	T1	G14	SSTL15_T_DCI
25	ddr3_dq[24]	39	T0	B14	SSTL15_T_DCI
26	ddr3_dq[25]	39	T0	C13	SSTL15_T_DCI
27	ddr3_dq[26]	39	T0	B16	SSTL15_T_DCI

INFO: Press **Validate** to proceed.

Validate Read XDC/UCF Save Pin Out

## 9.2 Simulating the Example Design

Right click the IP core under the Sources menu and click *Open IP Example Design*, which will create a new example Vivado project, connecting the generated MIG to a Traffic Generator IP using AXI4.

After running synthesis and implementation, your schematic should look similar to this:

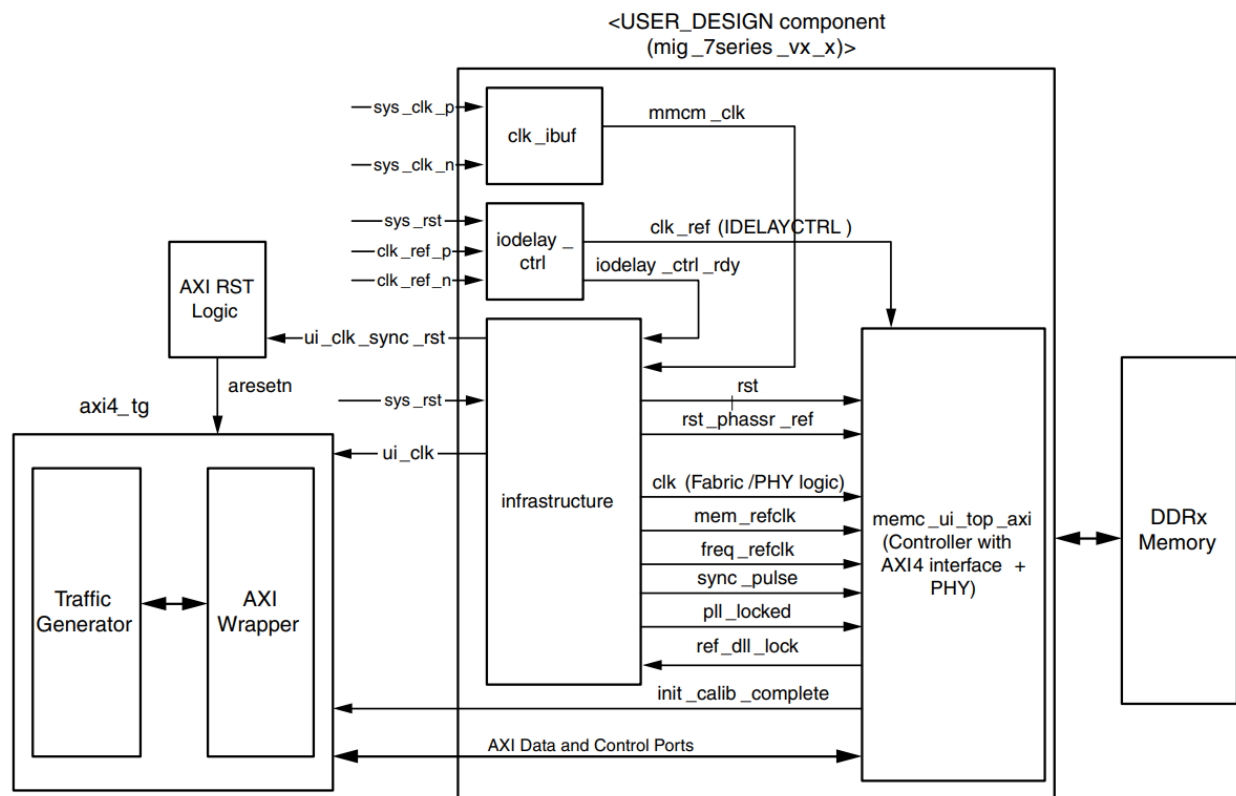
Looking at the 7 Series MIG documentation ([UG586](#)), we can see an abstracted BD of the example MIG design.

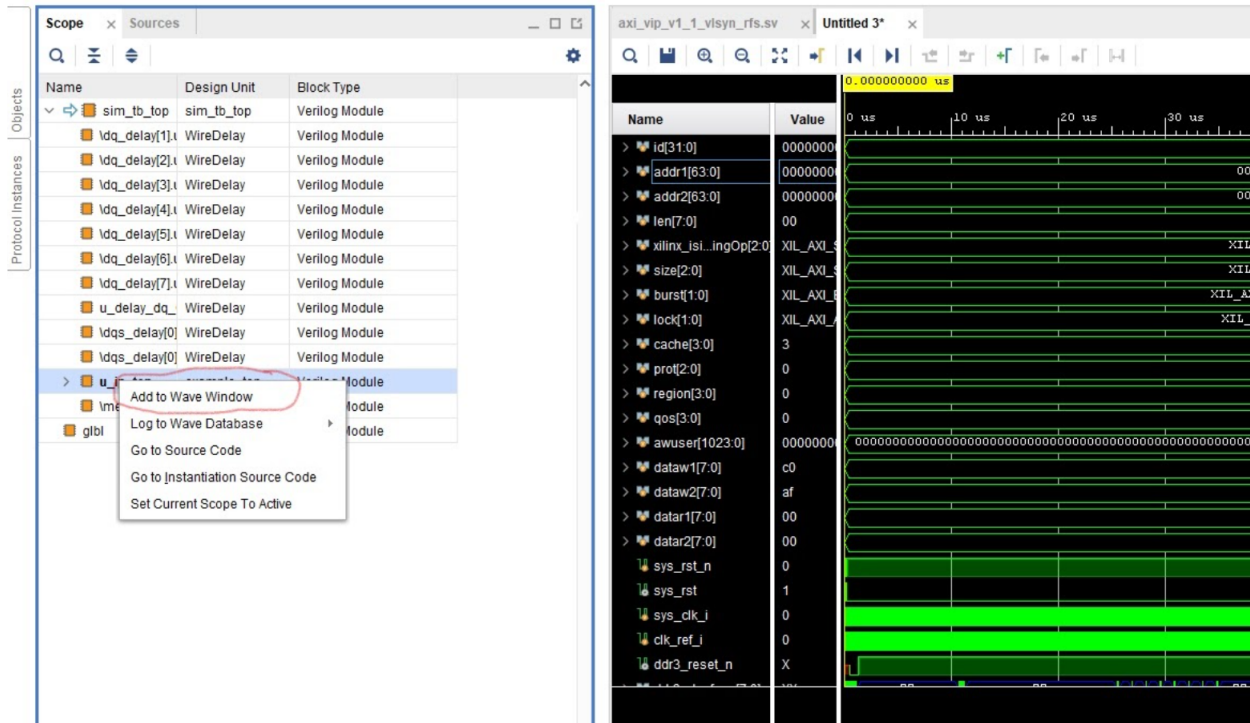
The example design uses a traffic generator to simulate the host PC reading/writing data from/to the MIG core. While useful, Xilinx's implementation is slightly obfuscated, so we will replace it with our own *VIP traffic generator* moving forward.

The MIG's reset scheme is as follows:

1. Raise the `sys_rst` port HIGH, since we defined it as active HIGH.
2. By doing this, the `ui_sync_rst` port also goes HIGH.
3. This port is passed into an inverter (the LUT1 in the schematic).
4. A LOW signal is sent from `aresetn_reg` (also known as `aresetn`), which resets all AXI components (including both the traffic generator and MIG).

We can observe this behavior by running a Behavioral Simulation in Vivado. Make sure to add the correct AXI signals by clicking the *Scope* heading, right clicking on the `u_ip_top` module and selecting *Add to Wave Window*. This will allow us to see the AXI read and write transactions.





**Note:** If you need a refresher on the AXI protocol or interpreting the simulation's waveforms, check [here](#).

Since the MIG needs time to calibrate and set up, no AXI reads/writes will occur until after the `init_calib_complete` pin goes HIGH after 100us.

After running the MIG's Behavioral Simulation, you should observe that the AXI Address Width is 27 bits and the AXI Data Width is 32 bits, which is expected.

**Tip:** To find AXI parameter values such as Address or Data Width and Base Address, look for the comment *AXI4 Shim parameters* in the `u_mig_7series_4_mig` module.

## 9.3 Simulating Read/Writes with AXI VIP

**Note:** All further examples are implemented using a Kintex-7 FPGA. However, the most pertinent portions apply to all other FPGAs (e.g., the VIP implementation can also be used in the VC707's provided testbench).

As mentioned before, Xilinx's implementation of their Traffic Generator is difficult to break down into understandable chunks. Luckily, Xilinx also provides an alternative — the AXI Verification IP (or AXI VIP), which can simulate an AXI master, slave, or pass-through device. You can find more information about this IP through its product guide (PG267), but for our purposes, we will instead manually instantiate the IP using the example design.

Open the `example_top` module and comment out the entire traffic generator instantiation. It will have a comment above it stating *The traffic generation module instantiated below drives traffic (patterns) on the application interface of the memory controller*.

Make sure to also disable all the traffic generator-related source files: `mig_7series_v4_2_axi4_tg.v`, `mig_7series_v4_2_axi4_wrapper.v`, `mig_7series_v4_2cmd_prbs_gen_axi.v`, `mig_7series_v4_2_data_gen_chk.v`, and `mig_7series_v4_2_tg.v`. In the abstracted block diagram (BD) above, we are essentially replacing the entire `axi4_tg` module.

Using the IP Catalog, customize the AXI VIP as such:

AXI Verification IP (1.1)

Documentation IP Location Switch to Defaults

☐ Show disabled ports

Component Name `axi_vip_0`

**Basic Settings**

INTERFACE MODE: MASTER

PROTOCOL: AXI4

READ\_WRITE MODE: READ WRITE

ADDRESS WIDTH: 27 [12 - 64]

DATA WIDTH: 32

ID WIDTH: 4

**User signal widths**

AWUSER WIDTH: 0 [0 - 1024]

ARUSER WIDTH: 0 [0 - 1024]

HAS\_USER\_BITS\_PER\_BYTE: NO

WUSER BITS PER BYTE: 0 [0 - 32]

RUSER BITS PER BYTE: 0 [0 - 32]

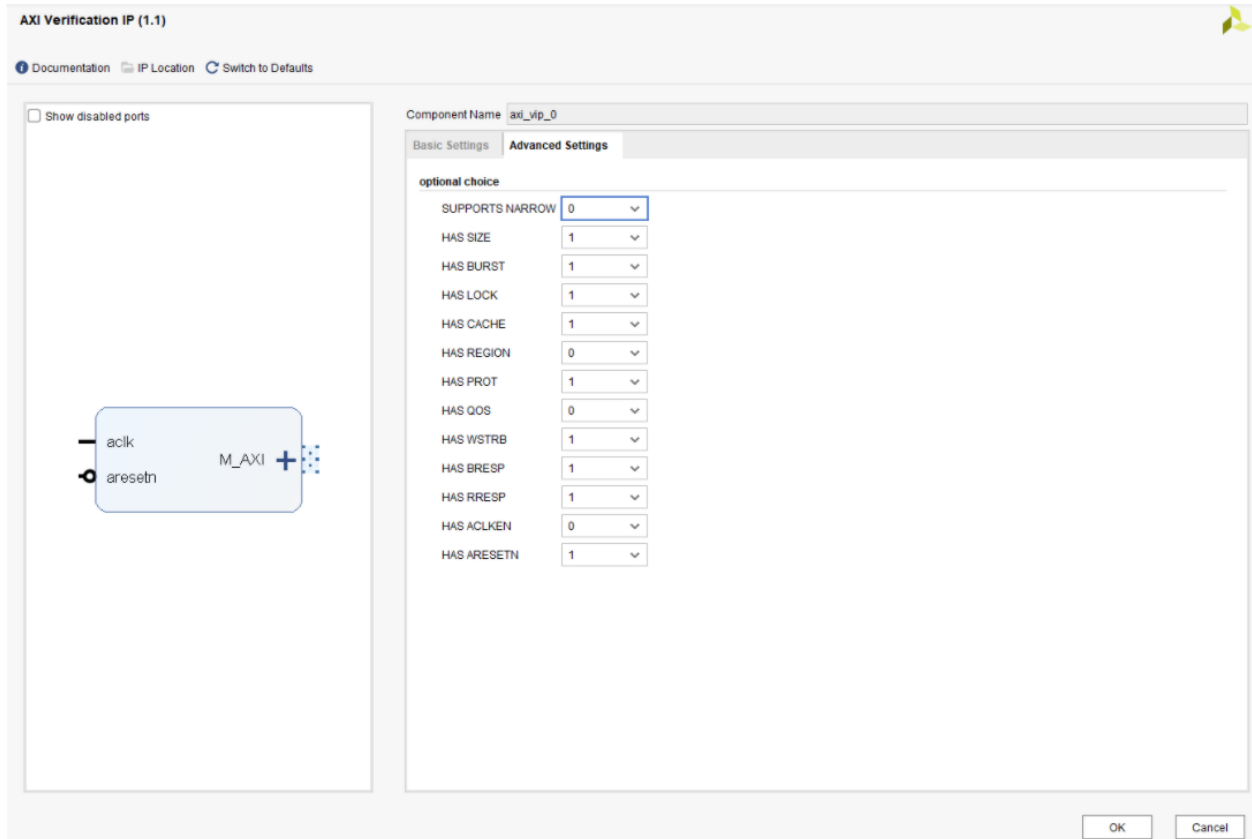
WUSER WIDTH: 0 [0 - 1024]

RUSER WIDTH: 0 [0 - 1024]

BUSER WIDTH: 0 [0 - 1024]

The AXI Verification IP can only act as a protocol checker when contained within a VHDL hierarchy.  
In order to use the virtual part of the AXI Verification IP, it must be in Verilog hierarchy.  
In order to use the virtual part of this IP, please refer PG267 section about "Useful Coding Guidelines and Examples "

The API for the virtual part of this IP can be found: <https://www.xilinx.com/support/answers/68234.html>



Open the top module of the AXI VIP (`axi_vip_0`), copy all input/output signals (listed underneath *module* `axi_vip_0`), and paste these signals back into the `example_top.v` file in place of the commented-out TG instantiation.

**Important:** If you want to download the top file instead, go [here](#). Just be sure to rename `example_top_axi.v` to `example_top.v`!

```
//*****
// The traffic generation module instantiated below drives traffic (patterns)
// on the application interface of the memory controller
//*****
always @(posedge clk) begin
    aresetn <= ~rst;
end

//INSTANTIATE AXI VIP INSTEAD OF TRAFFIC GENERATOR

axi_vip_0 u_axi_vip_0 (
    .aclk(clk),
    .aresetn(aresetn),
    .m_axi_awid(s_axi_awid),
    .m_axi_awaddr(s_axi_awaddr),
    .m_axi_awlen(s_axi_awlen),
    .m_axi_awsz(s_axi_awsz),
    .m_axi_awburst(s_axi_awburst),
    .m_axi_awlock(s_axi_awlock),
    .m_axi_awcache(s_axi_awcache),
```

(continues on next page)

(continued from previous page)

```

.m_axi_awprot(s_axi_awprot),
.m_axi_awvalid(s_axi_awvalid),
.m_axi_awready(s_axi_awready),
.m_axi_wdata(s_axi_wdata),
.m_axi_wstrb(s_axi_wstrb),
.m_axi_wlast(s_axi_wlast),
.m_axi_wvalid(s_axi_wvalid),
.m_axi_wready(s_axi_wready),
.m_axi_bid(s_axi_bid),
.m_axi_bresp(s_axi_bresp),
.m_axi_bvalid(s_axi_bvalid),
.m_axi_bready(s_axi_bready),
.m_axi_arid(s_axi_arid),
.m_axi_araddr(s_axi_araddr),
.m_axi_arlen(s_axi_arlen),
.m_axi_arsize(s_axi_arsize),
.m_axi_arburst(s_axi_arburst),
.m_axi_arlock(s_axi_arlock),
.m_axi_arcache(s_axi_arcache),
.m_axi_arprot(s_axi_arprot),
.m_axi_arvalid(s_axi_arvalid),
.m_axi_arready(s_axi_arready),
.m_axi_rid(s_axi_rid),
.m_axi_rdata(s_axi_rdata),
.m_axi_rresp(s_axi_rresp),
.m_axi_rlast(s_axi_rlast),
.m_axi_rvalid(s_axi_rvalid),
.m_axi_rready(s_axi_rready)
);

// COMMENT THIS PART OUT BELOW
mig_7series_v4_2_axi4_tg #(

```

If synthesis completes, the AXI VIP has been successfully instantiated into the design in place of the traffic generator. The file hierarchy should be like this:

We can now add our AXI VIP testbench into the simulation top file `sim_tb_top`. We will use SystemVerilog to implement this testbench, so right click on the file, select *Set File Type*, and change the simulation language to SystemVerilog.

The objective of this testbench is to write some data to the DDR memory and read back from the addresses we specified to compare the data. To achieve this, first initialize the AXI VIP in `sim_tb_top` like so:

```

import axi_vip_pkg::*; //import packages for the AXI VIP
import axi_vip_0_pkg::*;

module sim_tb_top;

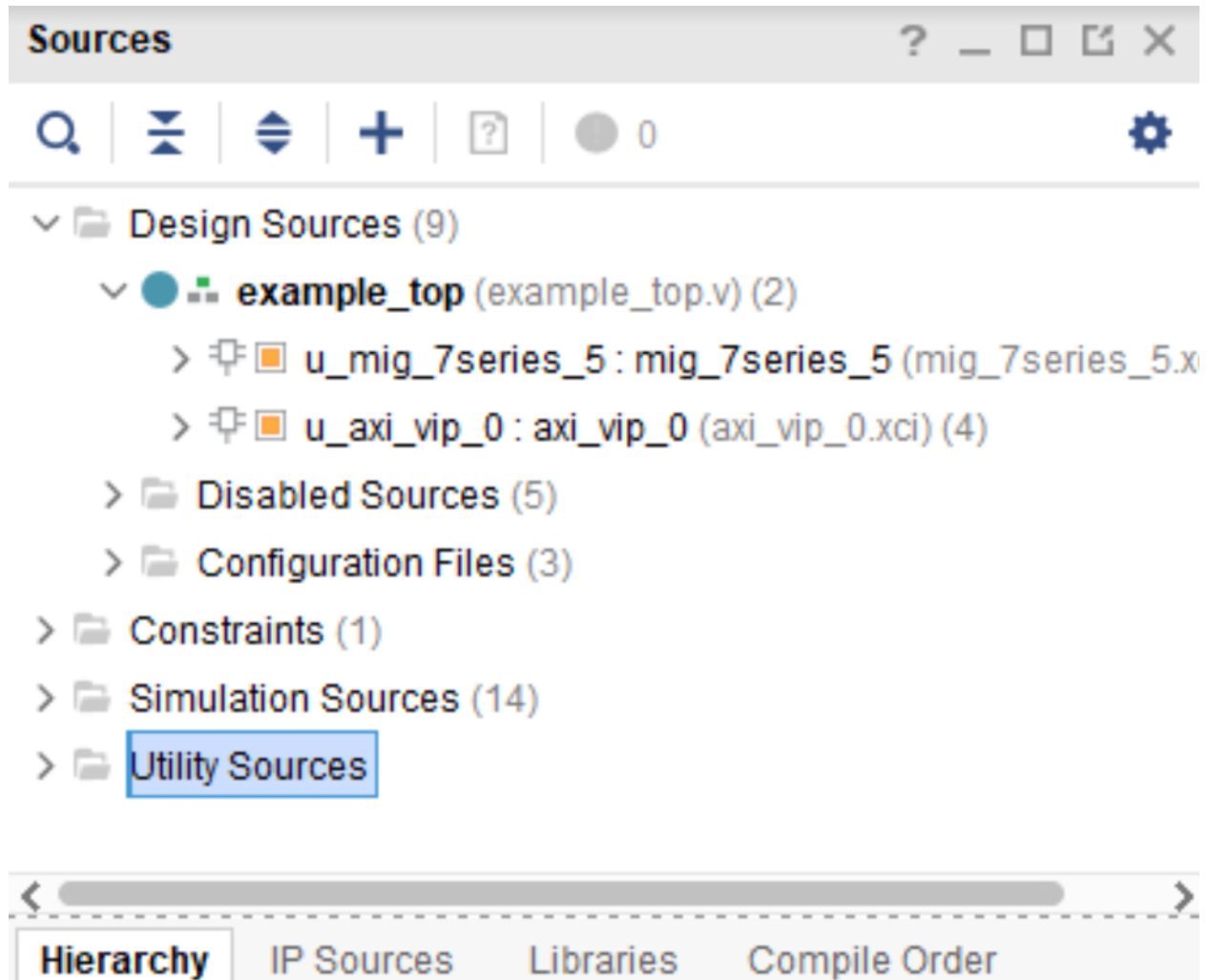
//declare AXI agent as master
axi_vip_0_mst_t      agent;

//define parameters for AXI VIP
axi_transaction      wr_trans1, wr_trans2; //two AXI write transactions
axi_transaction      rd_trans1, rd_trans2; //two AXI read transactions
xil_axi_uint          id =0; //default
xil_axi_ulong         addr1 =32'h0000, addr2 = 32'h0004; //define two test_
↪addresses

```

(continues on next page)





(continued from previous page)

```

xil_axi_len_t          len =0; //only one burst
xil_axi_size_t         size =xil_axi_size_t'(xil_clog2((32)/8)); //default,
↳maximum of 4 words per transaction (4 bytes for 32 bit AXI bus)
xil_axi_burst_t        burst =XIL_AXI_BURST_TYPE_INCR; //default,incremental
↳burst type
xil_axi_lock_t         lock = XIL_AXI_ALOCK_NOLOCK; //default
xil_axi_cache_t        cache =3; //default
xil_axi_prot_t         prot =0; //default
xil_axi_region_t       region =0; //default
xil_axi_qos_t          qos =0; //default
xil_axi_data_beat [255:0] wuser =0; //default
xil_axi_data_beat      awuser =0; //default
bit [7:0]              dataw1 = 8'hC0, dataw2 = 8'hAF; //define two data words
↳for AXI writes
bit [7:0]              datar1, datar2; //if successful, these should match dataw1
↳and dataw2

```

Then, we perform two writes into DDR, one to address 0x0000 of data 0xC0 and the other to address 0x0004 of data 0xAF, and two reads from the same addresses, through:

```

//*****
// Reporting the test case status
// Status reporting logic exists both in simulation test bench (sim_tb_top)
// and sim.do file for ModelSim. Any update in simulation run time or time out
// in this file need to be updated in sim.do file as well.
//*****
initial
begin : Logging

    fork
        begin : calibration_done
            wait (init_calib_complete); //wait until init_calib_complete is done
            $display("Calibration Done");

            #100000; //100 ns delay

            agent = new("master vip agent",u_ip_top.u_axi_vip_0.inst.IF); //pass correct
↳IF path
            agent.start_master(); //start master agent

            //begin write transactions to address 1 and address 2
            wr_trans1 = agent.wr_driver.create_transaction("single_write"); //initialize
↳first transaction
            wr_trans1.set_write_cmd(addr1,burst,id,len,size); //declare address 1, as
↳well as burst length and size
            wr_trans1.set_prot(prot); //set all other default parameters
            wr_trans1.set_lock(lock);
            wr_trans1.set_cache(cache);
            wr_trans1.set_region(region);
            wr_trans1.set_qos(qos);
            wr_trans1.set_data_block(dataw1); //put data1 on the AXI data bus
            agent.wr_driver.send(wr_trans1); //send write transaction

            #100000; //100 ns delay

            wr_trans2 = agent.wr_driver.create_transaction("single_write"); //initialize
↳second transaction

```

(continues on next page)

(continued from previous page)

```

        wr_trans2.set_write_cmd(addr2,burst,id,len,size); //declare address 2, as_
↪well as burst length and size
        wr_trans2.set_prot(prot); //set all other default parameters
        wr_trans2.set_lock(lock);
        wr_trans2.set_cache(cache);
        wr_trans2.set_region(region);
        wr_trans2.set_qos(qos);
        wr_trans2.set_data_block(dataw2); //put data2 on the AXI data bus
        agent.wr_driver.send(wr_trans2); //send write transaction

        #100000; //100 ns delay

        //begin read transaction to address 1 and address 2
        rd_trans1 = agent.rd_driver.create_transaction("single_read"); //initialize_
↪read transaction
        rd_trans1.set_read_cmd(addr1,burst,id,len,size); //set the correct parameters
        rd_trans1.set_prot(prot);
        rd_trans1.set_lock(lock);
        rd_trans1.set_cache(cache);
        rd_trans1.set_region(region);
        rd_trans1.set_qos(qos);
        rd_trans1.set_driver_return_item_policy(XIL_AXI_PAYLOAD_RETURN); //default,_
↪set driver return policy
        agent.rd_driver.send(rd_trans1); //send read transaction
        agent.rd_driver.wait_rsp(rd_trans1); //wait for response signal
        datar1 = rd_trans1.get_data_block(); //obtain read data

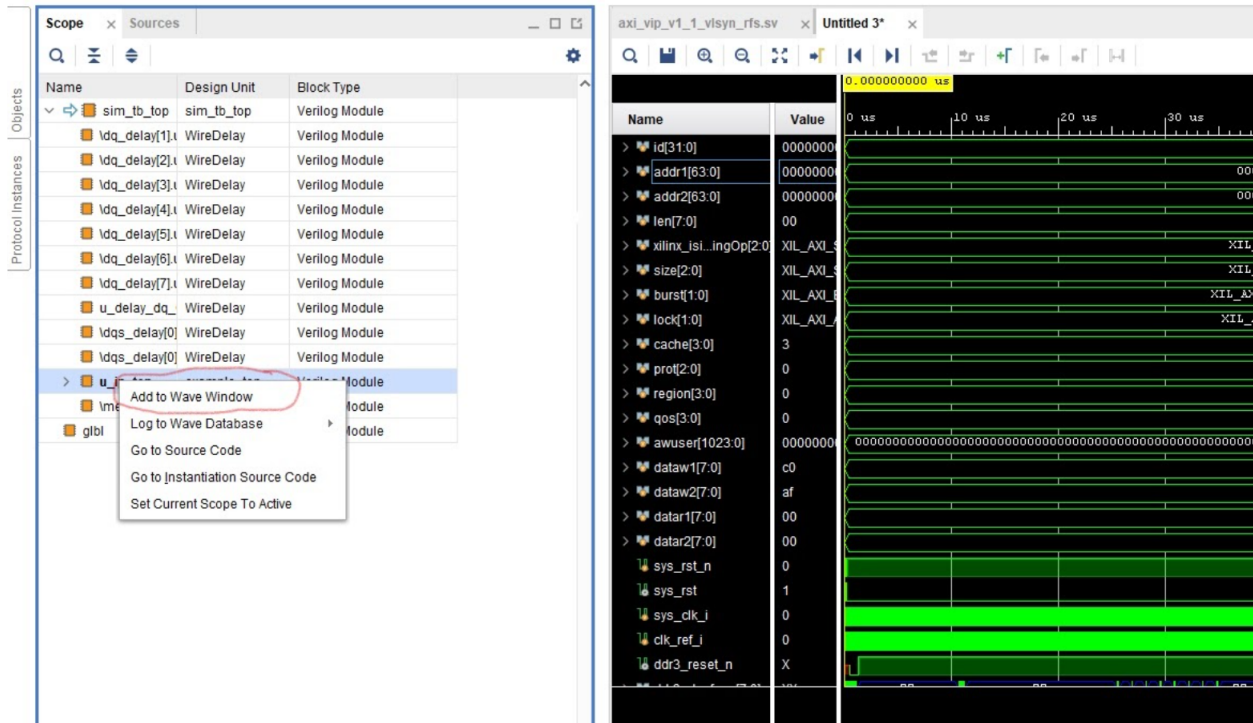
        #100000; //100 ns delay

        rd_trans2 = agent.rd_driver.create_transaction("single_read"); //initialize_
↪read transaction
        rd_trans2.set_read_cmd(addr2,burst,id,len,size); //set correct parameters
        rd_trans2.set_prot(prot);
        rd_trans2.set_lock(lock);
        rd_trans2.set_cache(cache);
        rd_trans2.set_region(region);
        rd_trans2.set_qos(qos);
        rd_trans2.set_driver_return_item_policy(XIL_AXI_PAYLOAD_RETURN); //default,_
↪set driver return policy
        agent.rd_driver.send(rd_trans2); //send read transaction
        agent.rd_driver.wait_rsp(rd_trans2); //wait for response signal
        datar2 = rd_trans2.get_data_block(); //obtain read data

        #100000; //100 ns delay
        if (datar1 == dataw1 && datar2 == dataw2) begin //test successful if this_
↪condition is true
            $display("TEST PASSED");
        end
        else begin
            $display("TEST FAILED: DATA ERROR");
        end
        disable calib_not_done;
        $finish;
    end
end

```

We can now run our Behavioral Simulation, but make sure to add the AXI signals by opening the Scope menu, right clicking on the ui\_top file, and selecting *Add to Wave Window*.



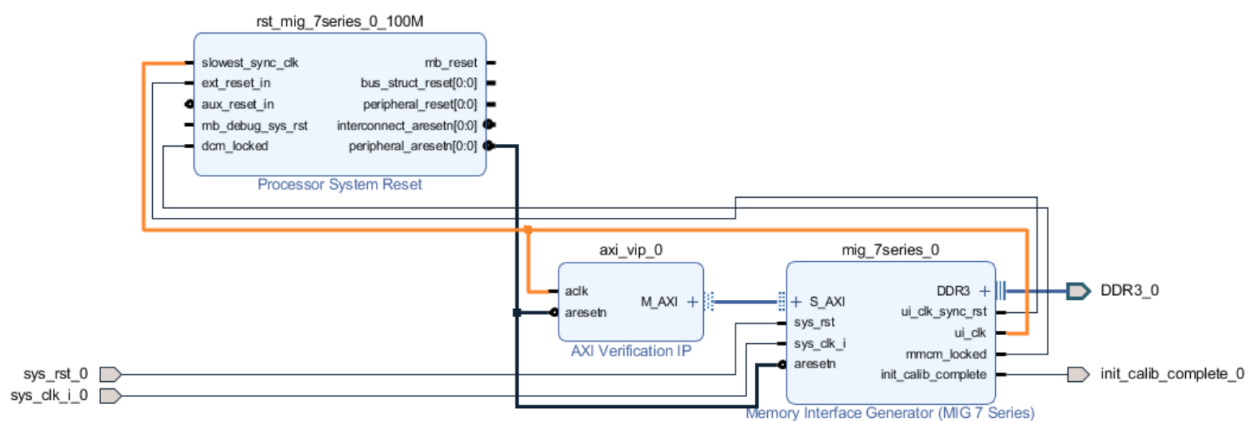
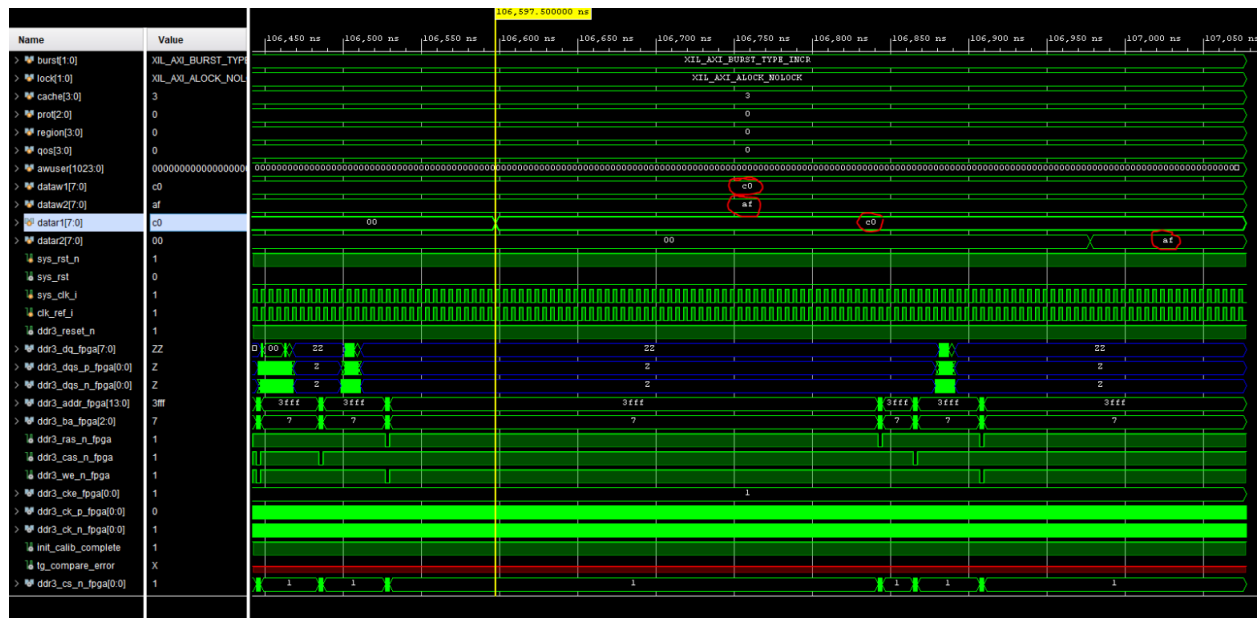
During the simulation, `init_calib_complete` will go HIGH after about 100us, after which the reads and writes will begin. `sys_reset` will be held HIGH for the first 200ns, causing the other resets to initiate and begin calibration. Here is what a successful simulation will look like:

As we can see, the two bytes that were read from memory (`c0` and `af` from `datar1` and `datar2`, respectively) matched the two bytes that were initially written to those memory addresses (`dataw1` and `dataw2`). If your simulation matches this, good job! The simulation was a success.

## 9.4 Connecting the MIG to a Custom Design

Perhaps you wish to connect the generated MIG to any AXI master, not just the AXI VIP. Using the VIP as another example, using the IP Integrator (making a BD) makes this process very straightforward.

- The `ui_clk` must be driving the AXI read/write transactions to the MIG (i.e., the `aclk` on the AXI VIP).
- The `ui_clk_sync_rst` must be driving the `aresetn` pin on the AXI master (since `ui_clk_sync_rst` is active HIGH and `aresetn` is active LOW, we use a Processor System Reset IP for easy conversion)
- The `sys_clk_i` is the 200 MHz input clock that we defined in our MIG customization (which is also tied to the reference clock).
- `sys_rst` is the active HIGH reset that we defined in our MIG customization; bringing this pin HIGH will trigger the `ui_clk_sync_rst`, which will in turn trigger the `aresetn` pin on the AXI master.
- `init_calib_complete` tells us when the MIG calibration is complete, so that we can begin using the DDR memory (will take about 100us to go HIGH in simulation).
- Finally, the external DDR bus connects to the physical RAM on the emulation board (bus outputs need to be assigned correctly using a XDC constraints file).



## 9.5 Connecting the MIG to Two AXI Master VIPs using AXI SmartConnect

After connecting one AXI VIP to the MIG, naturally we should also test dual reads/writes from two AXI masters simultaneously by connecting two AXI VIPs to a singular MIG. Later, we will use this principle to replace the AXI masters with a PCIe core and a DUT, moving closer to a full emulation environment. To achieve this, we will use an AXI SmartConnect IP.

**Error:** Xilinx now recommends that all new AXI designs use the SmartConnect v1.0 core. It is not recommended to use the AXI Interconnect v2.1 core.

**Note:** You can read more about the SmartConnect IP [here](#).

Beginning with our modified MIG example design with one AXI VIP, create a new block diagram (BD). Add a SmartConnect IP and customize it as shown:

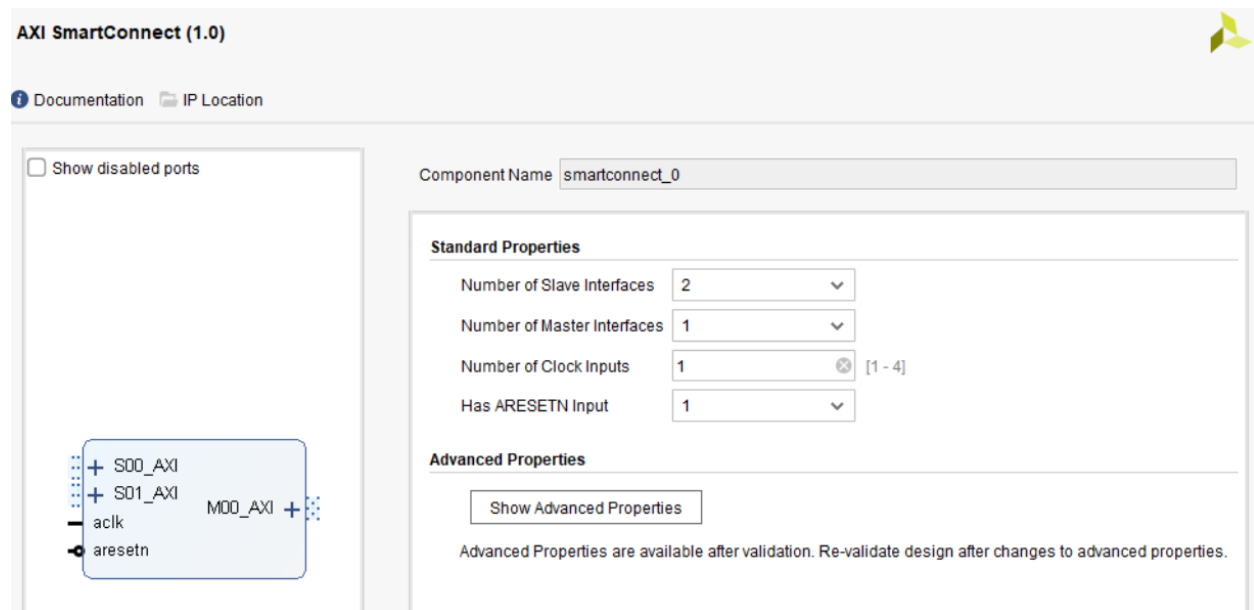


Fig. 1: SmartConnect customization

Add two Master AXI VIP IPs to the BD and customize them:

Connect them together in the BD (make `aclk`, `aresetn`, and `M00_AXI` external to instantiate them later):

If you try to Validate the BD now, a warning message about an unmapped slave will appear. To fix this, go to the **Address Editor** tab and right click on the two AXI Master VIPs to map the `M00_AXI_0` port to Offset Address `0x0000_0000` for both AXI VIPs.

Make sure your design fully validates by right clicking the BD and selecting *Validate Design*.

Right click your BD in the Sources directory and *Create a HDL Wrapper*, which will generate the RTL needed to instantiate our BD. When it is done generating, open the top file (default name is similar to `design_1_wrapper`) and copy all inputs/outputs in the module.

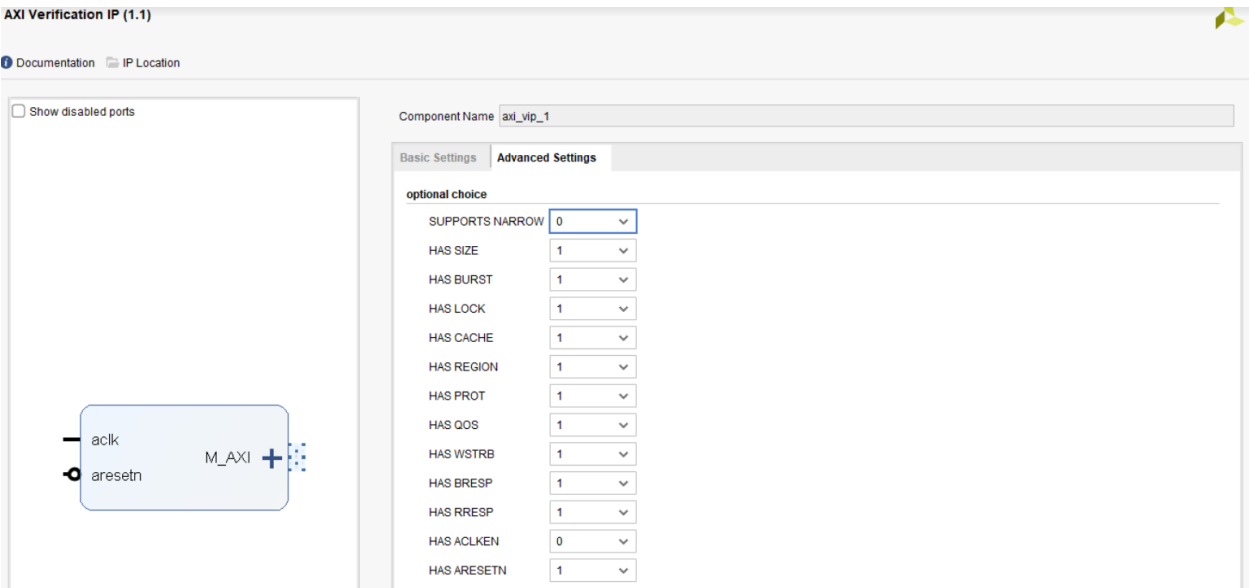
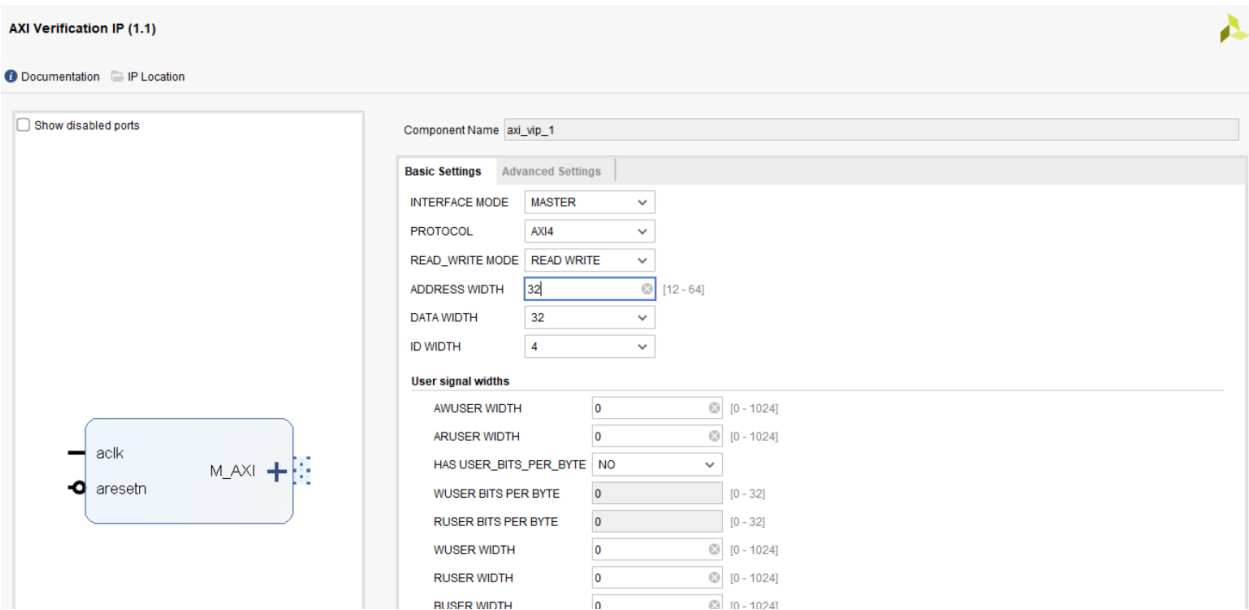


Fig. 2: MIG AXI VIP customization

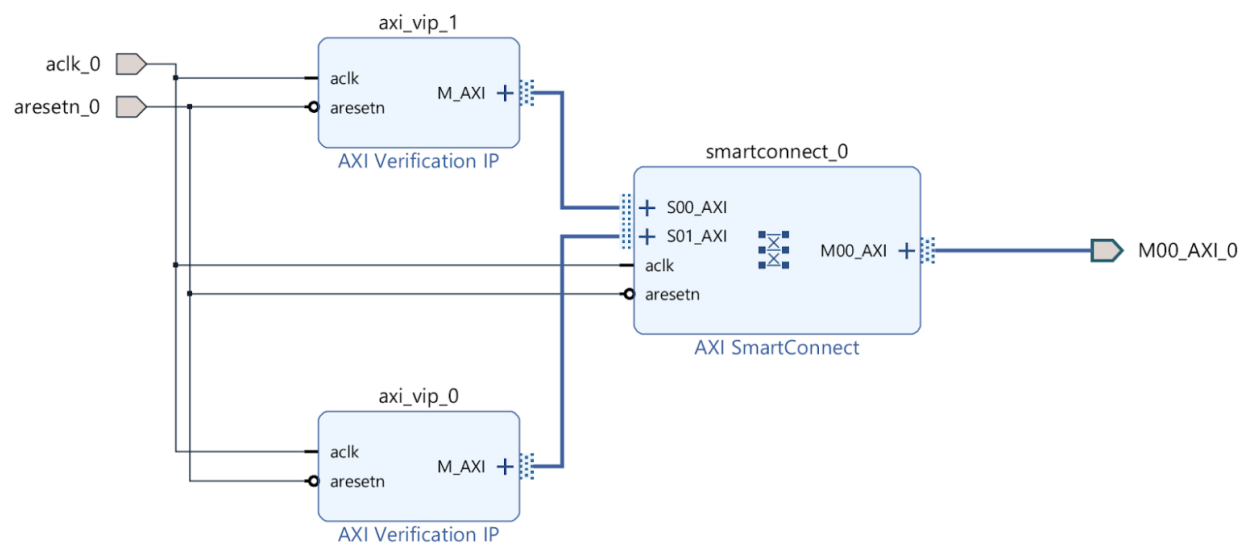


Fig. 3: AXI VIP Block Diagram

Diagram x Address Editor x readme.txt x xsim\_options.tcl x ddr3\_model.sv x

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_vip_0					
Master_AXI (32 address bits : 4G)					
M00_AXI_0	M00_AXI_0	Reg	0x0000_0000	64K	0x0000_FFFF
axi_vip_1					
Master_AXI (32 address bits : 4G)					
M00_AXI_0	M00_AXI_0	Reg	0x0000_0000	64K	0x0000_FFFF

Fig. 4: MIG AXI Address Editor



Moving back to our MIG example\_top file, remove the previous example instantiation of the AXI VIP and insert the new instantiation with the ports from design\_1\_wrapper. It will look like this:

```
//*****
// The traffic generation module instantiated below drives traffic (patterns)
// on the application interface of the memory controller
//*****
always @(posedge clk) begin
aresetn <= ~rst;
end

//INSTANTIATE Block Diagram with 2 AXI VIPs and an AXI Interconnect

design_1_wrapper u_axi_vip_interconnect_bd (

    //M00_AXI_arid(s_axi_arid), //no port on AXI Smartconnect
    .M00_AXI_0_araddr(s_axi_araddr),
    .M00_AXI_0_arburst(s_axi_arburst),
    .M00_AXI_0_arcache(s_axi_arcache),
    .M00_AXI_0_arlen(s_axi_arlen),
    .M00_AXI_0_arlock(s_axi_arlock),
    .M00_AXI_0_arprot(s_axi_arprot),
    //M00_AXI_0_arqos(s_axi_arqos), //no port on AXI Smartconnect
    .M00_AXI_0_arready(s_axi_arready),
    .M00_AXI_0_arsize(s_axi_arsize),
    .M00_AXI_0_arvalid(s_axi_arvalid),
    //M00_AXI_awid(s_axi_awid), //no port on AXI Smartconnect
    .M00_AXI_0_awaddr(s_axi_awaddr),
    .M00_AXI_0_awburst(s_axi_awburst),
    .M00_AXI_0_awcache(s_axi_awcache),
    .M00_AXI_0_awlen(s_axi_awlen),
    .M00_AXI_0_awlock(s_axi_awlock),
    .M00_AXI_0_awprot(s_axi_awprot),
    //M00_AXI_0_awqos(s_axi_awqos), //no port on AXI Smartconnect
    .M00_AXI_0_awready(s_axi_awready),
    .M00_AXI_0_awsized(s_axi_awsized),
    .M00_AXI_0_awvalid(s_axi_awvalid),
    //M00_AXI_0_bid(s_axi_bid), //no port on AXI Smartconnect
    .M00_AXI_0_bready(s_axi_bready),
    .M00_AXI_0_bresp(s_axi_bresp),
    .M00_AXI_0_bvalid(s_axi_bvalid),
    .M00_AXI_0_rdata(s_axi_rdata),
    //M00_AXI_0_rid(s_axi_rid), //no port on AXI Smartconnect
    .M00_AXI_0_rlast(s_axi_rlast),
    .M00_AXI_0_rready(s_axi_rready),
    .M00_AXI_0_rresp(s_axi_rresp),
    .M00_AXI_0_rvalid(s_axi_rvalid),
    .M00_AXI_0_wdata(s_axi_wdata),
    .M00_AXI_0_wlast(s_axi_wlast),
    .M00_AXI_0_wready(s_axi_wready),
    .M00_AXI_0_wstrb(s_axi_wstrb),
    .M00_AXI_0_wvalid(s_axi_wvalid),
    .aclk_0(clk),
    .aresetn_0(aresetn)
);

// COMMENT OUT THIS PART BELOW
mig_7series_v4_2_axi4_tg #(
```

Now we can run synthesis to verify that the top file compiles. There may be a small syntax error, which we can ignore.

Now that we have successfully instantiated our new design, our two AXI Masters should be able to perform read/write requests to the MIG through the AXI SmartConnect IP. We can verify this through a behavioral simulation that performs two simultaneous write/read requests to two different addresses.

---

**Important:** The simulation top file can be found here. Just be sure to rename `example_top_2axi.v` to `example_top.v`!

---



---

**Note:** This testbench will only work if you named your BD instantiation as `u_axi_vip_interconnect_bd` and left the component names of the AXI VIPs as default.

---

As before, make sure to instantiate the two AXI VIPs and their ports within the example testbench:

```
import axi_vip_pkg::*; //import packages for the AXI VIP
import design_1_axi_vip_0_0_pkg::*;
import design_1_axi_vip_0_1_pkg::*;

module sim_tb_top;

    //declare AXI agent as master
    design_1_axi_vip_0_0_mst_t    agent0;
    design_1_axi_vip_0_1_mst_t    agent1;

    //define parameters for AXI VIP
    axi_transaction              wr_trans1, wr_trans2; //two AXI write transactions
    axi_transaction              rd_trans1, rd_trans2; //two AXI read transactions
    xil_axi_uint                 id = 0; //default
    xil_axi_ulong                addr1 = 32'h0000, addr2 = 32'h0004; //define two test_
    ↪addresses
    xil_axi_len_t                len = 0; //only one burst
    xil_axi_size_t               size = xil_axi_size_t'(xil_clog2((32)/8)); //default,
    ↪maximum of 4 words per transaction (4 bytes for 32 bit AXI bus)
    xil_axi_burst_t              burst = XIL_AXI_BURST_TYPE_INCR; //default,incremental
    ↪burst type
    xil_axi_lock_t               lock = XIL_AXI_ALOCK_NOLOCK; //default
    xil_axi_cache_t              cache = 3; //default
    xil_axi_prot_t               prot = 0; //default
    xil_axi_region_t             region = 0; //default
    xil_axi_qos_t                qos = 0; //default
    xil_axi_data_beat [255:0]    wuser = 0; //default
    xil_axi_data_beat            awuser = 0; //default
    bit [7:0]                    dataw1 = 8'hC0, dataw2 = 8'hAF; //define two data words
    ↪for AXI writes
    bit [7:0]                    datar1, datar2; //if successful, these should match dataw1
    ↪and dataw2
```

Then we set up two write and read requests using both AXI VIPs to two specified addresses, using the same procedure as our last testbench with one AXI VIP.

```
//*****
// Reporting the test case status
// Status reporting logic exists both in simulation test bench (sim_tb_top)
// and sim.do file for ModelSim. Any update in simulation run time or time out
// in this file need to be updated in sim.do file as well.
```

(continues on next page)

(continued from previous page)

```

//*****
initial
begin : Logging

    fork
        begin : calibration_done
            wait (init_calib_complete); //wait until init_calib_complete is done
            $display("Calibration Done");

            #100000; //100 ns delay

            agent0 = new("master vip agent",u_ip_top.u_axi_vip_interconnect_bd.design_1_
↪i.axi_vip_0.inst.IF); //pass correct IF path
            agent0.start_master(); //start master agent
            agent1 = new("master vip agent",u_ip_top.u_axi_vip_interconnect_bd.design_1_
↪i.axi_vip_1.inst.IF); //pass correct IF path
            agent1.start_master(); //start master agent

            //write using AXI VIP 1
            wr_trans1 = agent1.wr_driver.create_transaction("single_write"); //
↪initialize first transaction
            wr_trans1.set_write_cmd(addr1,burst,id,len,size); //declare address 1, as_
↪well as burst length and size
            wr_trans1.set_prot(prot); //set all other default parameters
            wr_trans1.set_lock(lock);
            wr_trans1.set_cache(cache);
            wr_trans1.set_region(region);
            wr_trans1.set_qos(qos);
            wr_trans1.set_data_block(dataw1); //put data1 on the AXI data bus
            agent1.wr_driver.send(wr_trans1); //send write transaction

            //write using AXI VIP 0
            wr_trans2 = agent0.wr_driver.create_transaction("single_write"); //
↪initialize second transaction
            wr_trans2.set_write_cmd(addr2,burst,id,len,size); //declare address 2, as_
↪well as burst length and size
            wr_trans2.set_prot(prot); //set all other default parameters
            wr_trans2.set_lock(lock);
            wr_trans2.set_cache(cache);
            wr_trans2.set_region(region);
            wr_trans2.set_qos(qos);
            wr_trans2.set_data_block(dataw2); //put data2 on the AXI data bus
            agent0.wr_driver.send(wr_trans2); //send write transaction

            #100000; //100 ns delay

            //read using AXI VIP 0
            rd_trans1 = agent0.rd_driver.create_transaction("single_read"); //initialize_
↪read transaction
            rd_trans1.set_read_cmd(addr1,burst,id,len,size); //set the correct parameters
            rd_trans1.set_prot(prot);
            rd_trans1.set_lock(lock);
            rd_trans1.set_cache(cache);
            rd_trans1.set_region(region);
            rd_trans1.set_qos(qos);
            rd_trans1.set_driver_return_item_policy(XIL_AXI_PAYLOAD_RETURN); //default,_
↪set driver return policy

```

(continues on next page)

(continued from previous page)

```

agent0.rd_driver.send(rd_trans1); //send read transaction
agent0.rd_driver.wait_rsp(rd_trans1); //wait for response signal
datar1 = rd_trans1.get_data_block(); //obtain read data

//read using AXI VIP 1
rd_trans2 = agent1.rd_driver.create_transaction("single_read"); //initialize,
↪read transaction
rd_trans2.set_read_cmd(addr2,burst,id,len,size); //set correct parameters
rd_trans2.set_prot(prot);
rd_trans2.set_lock(lock);
rd_trans2.set_cache(cache);
rd_trans2.set_region(region);
rd_trans2.set_qos(qos);
rd_trans2.set_driver_return_item_policy(XIL_AXI_PAYLOAD_RETURN); //default,
↪set driver return policy
agent1.rd_driver.send(rd_trans2); //send read transaction
agent1.rd_driver.wait_rsp(rd_trans2); //wait for response signal
datar2 = rd_trans2.get_data_block(); //obtain read data

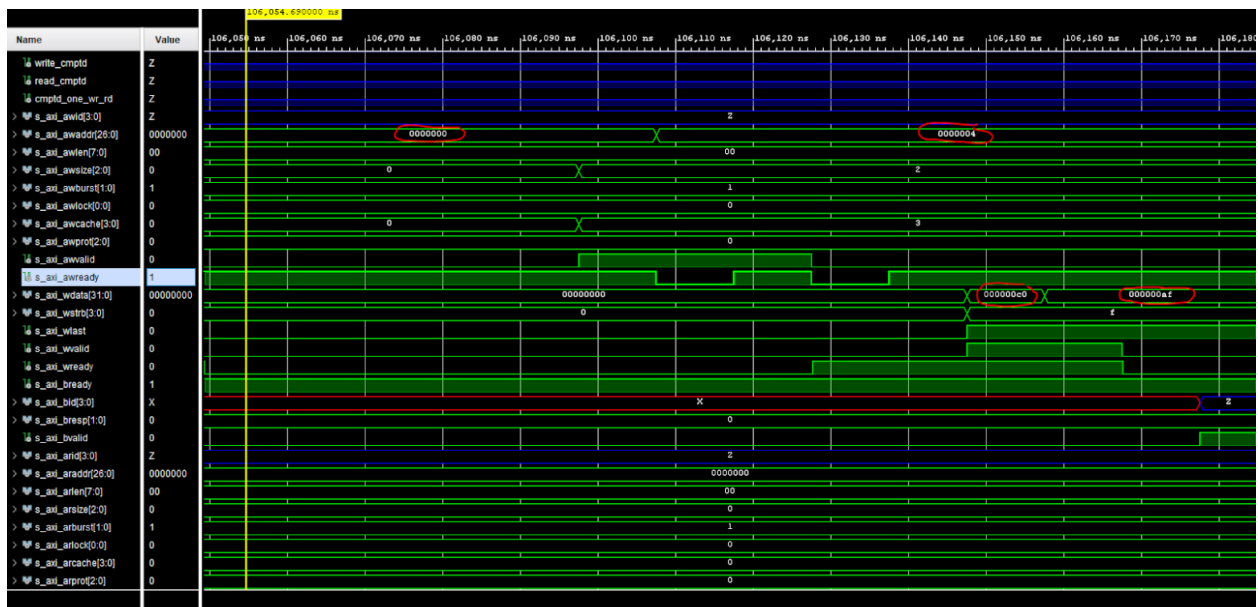
#1000000; //1000 ns delay

if (datar1 == dataw1 && datar2==dataw2) begin //test successful if this
↪condition is true
    $display("TEST PASSED");
end
else begin
    $display("TEST FAILED: DATA ERROR");
end
disable calib_not_done;
$finish;
end

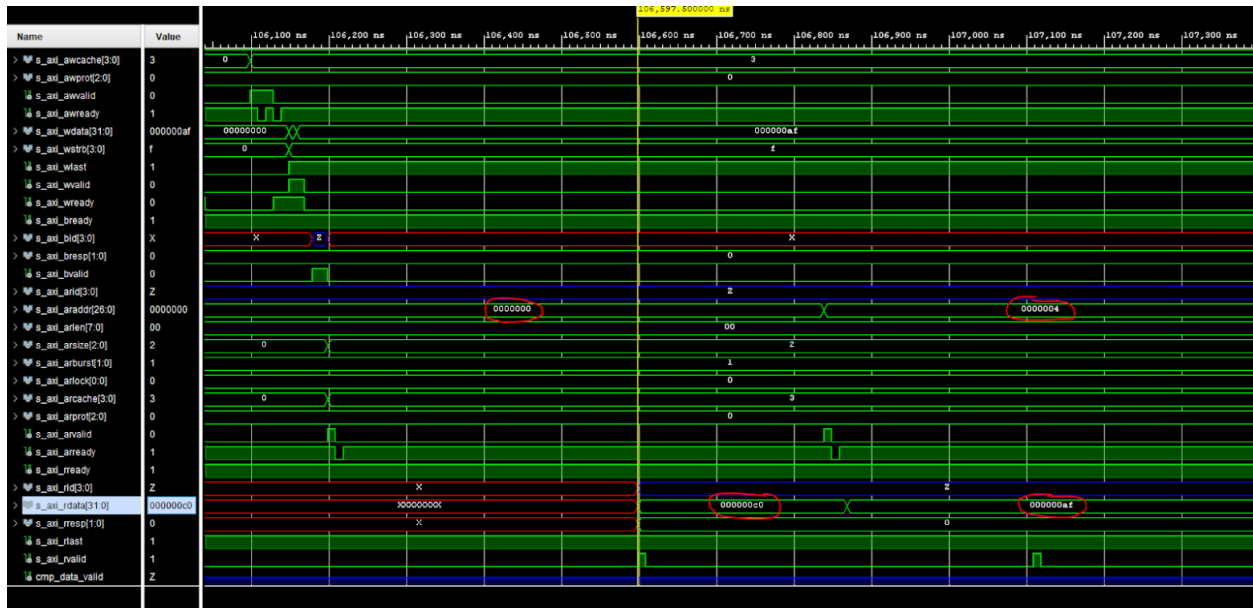
```

We can observe the simulation's intended behavior by running a Behavioral Simulation.

Here we can see two AXI Write transactions — one writing data C0 to address 0x0000 and one writing data AF to address 0x0004.



We can also observe two AXI Read transactions, one from address 0x0000 reading data C0 and one from address 0x0004 reading data AF.



If the TCL console prints a **Test Passed** message, congratulations! The test worked and you have successfully implemented two AXI VIPs with a MIG.

```
sim_tb_top.mem_rnk[0].gen_mem[0].u_comp_ddr3.data_task: at time 107027064.0 ps INFO:
↳READ @ DQS= bank = 0 row = 0000 col = 00000006 data = 00
sim_tb_top.mem_rnk[0].gen_mem[0].u_comp_ddr3.data_task: at time 107028314.0 ps INFO:
↳READ @ DQS= bank = 0 row = 0000 col = 00000007 data = 00
sim_tb_top.mem_rnk[0].gen_mem[0].u_comp_ddr3.cmd_task: at time 107048314.0 ps INFO:
↳Precharge bank 0
```

TEST PASSED

Executing Axi4 End of Simulation checks

Executing Axi4 End of Simulation checks

\$finish called at time : 108227500 ps : File "..."



## AXI MM TO PCIE IP OVERVIEW

The AXI Memory Mapped to PCI Express IP is a useful core that is compatible with only some FPGAs, offering a different implementation than that offered by the 7 Series Integrated Block for PCIe IP. More information can be found in the IP's documentation ([PG055](#)).

### 10.1 Customizing the IP

Create a new block diagram (BD) and use the IP catalog to add a new IP to the BD - in this case, the “AXI Memory Mapped to PCIe” core. We can customize it by double clicking it.

---

**Important:** Unless mentioned otherwise, leave all values default.

---

- In the PCIe:Basics tab, set the Reference Clock frequency to 100 MHz.
- Make sure we are customizing the device as an Endpoint.
- Depending on the application, in the PCIe: Link Config tab, change the Link Width. Most boards support at least X4, although newer boards will support X8. In addition, select the highest possible Link Speed for maximum performance.
- Again, depending on the application, you may change the Vendor ID and Class Code to something else. For example, if you plan to use this IP in conjunction with a soft CPU like MicroBlaze, you will change the Class Code to 0x060400 accordingly. If not, leave the entire tab default.
- In the PCIe:BARs tab, set BAR 0 at a size of 64 kB at offset address 0x00000000 and BAR 1 at size 64 kB at address 0x40000000.
- Every other tab can be left default.

### 10.2 Simulating the Example Design

After customizing, right click the IP block and open the IP Example Design.

The Example Design consists of the AXI MM to PCIe IP block connected to both a Block RAM (BRAM) Controller through the PCIe's AXI Master port and a Root Complex simulation on the PCIe's physical serial ports. Essentially, the example design simulates a host PC generating and sending traffic into the FPGA through the PCIe interface. The AXI MM to PCIe IP processes the incoming traffic and writes into BRAM using the AXI protocol.

---

**Note:** If you need a refresher on the PCIe protocol, check [here](#).

---

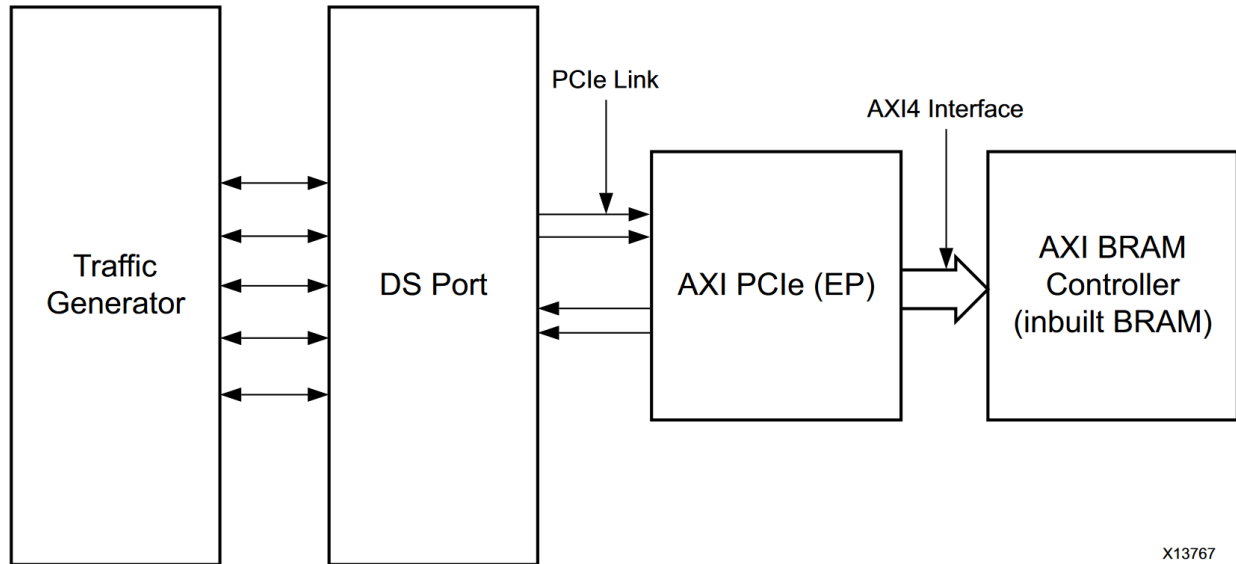


Fig. 1: Abstracted PCIe Example Design BD

Like the MIG design, the PCIe example design must first spend 175 us calibrating and initializing its serial ports. It then performs a simple write to address 0x10 in BRAM and subsequently reads it back, verifying that the read data matches what was written.

The PCIe serial ports take much longer to initialize than the MIG's. If you run a Behavioral Simulation, do not be surprised if nothing happens at first. The simulation may stall after returning the message *Built simulation snapshot board\_behav*. Since, by default, the simulation only runs for less than 1 us, simply use the command `run -all` in the TCL console to allow the serial lines to start toggling and the simulation to fully complete.

After around 30 minutes to 1 hour, you should receive a message in the console stating that the testbench has timed out. If the simulation is successful, the TCL console will also show that the test passed.

**Note:** The simulation will only work for one registered BAR in the IP. If you customized your IP to have multiple BARs, like we did, you will receive a message that the second BAR was disabled for this simulation.

```

[ 207869316] : TSK_PARSE_FRAME on Receive
[ 209749220] : Transmitting TLPs to Memory 32 Space BAR 0
[ 209767229] : TSK_PARSE_FRAME on Transmit
[ 209805308] : TSK_PARSE_FRAME on Transmit
[ 212125269] : TSK_PARSE_FRAME on Receive
[ 213797220] : Test PASSED --- Write Data: 01020304 successfully received
[ 213837220] : Finished transmission of PCI-Express TLPs
Test Completed Successfully
$finish called at time 213837220 ps : File "...
  
```



## 10.3 Example IP Block Diagram

After running Block and Connection Automation, the AXI MM to PCIe IP example BD will look similar to this:

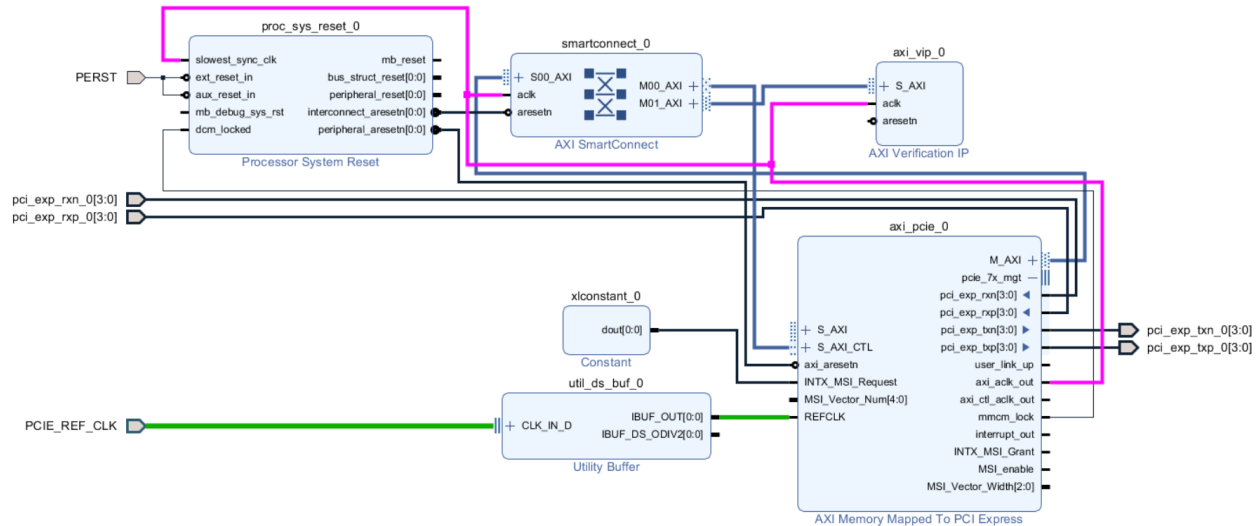


Fig. 2: PCIe Block Diagram

- The PCIe Reference Clock (REFCLK) at 100 MHz will go through an IBUFDSGTE Utility Buffer.
- The physical PERST (PCIe reset) pin is connected to a Processor System Reset IP, with the output going into the axi\_aresetn port.
- The INTX\_MSI\_Request port is connected to a Constant block tied active LOW (0) to prevent unwanted MSI interrupts.
- The M\_AXI port feeds into an AXI SmartConnect, where our AXI Slave devices are connected. The other AXI slave at this moment is a basic AXI Verification IP (VIP).
- The S\_AXI\_CTL port can be used as an AXI Slave port to perform reads and writes to the PCIe Configuration Space.
- The axi\_ack\_out port outputs a clock frequency of 125 MHz, which is the frequency that the AXI MM to PCIe Core operates at. It is currently clocking the SmartConnect and AXI VIP, which is typically not recommended (should use a Clocking Wizard for all other peripherals).
- The pci\_exp\_7x\_mgt ports are all external ports that connect to the physical PCIe port. They control the serial transactions between the root complex and PCIe endpoint.

To ensure that our customized BARs are accurately reflected in our AXI Slave devices, we must assign the correct addresses using the Address Editor. Map the S\_AXI\_CTL slave to address 0x00000000 and the AXI VIP slave to address 0x40000000.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_pcie_0					
M_AXI (32 address bits : 4G)					
axi_pcie_0	S_AXI_CTL	CTL0	0x0000_0000	16t	0x0000_3FFF
axi_vip_0	S_AXI	Reg	0x4000_0000	16t	0x4000_3FFF
Unconnected Slaves					

## 10.4 Replacing the BRAM with DDR MIG in Example Design

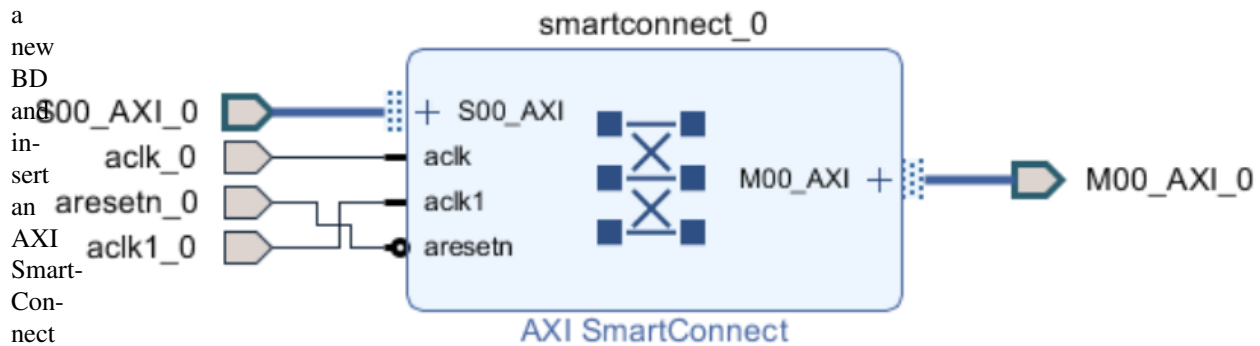
Create

a  
new  
BD  
and  
in-  
sert  
an  
AXI  
Smart-  
Con-  
nect  
with  
one

AXI Master input, one AXI Slave output, 64-bit Data Width, and two clock inputs. This SmartConnect will resolve the different clock domains that the PCIe IP and MIG run at.

Once the SmartConnect wrapper has been added to the project, open the IP catalog, and select the MIG 7 Series IP, customizing it like [this](#).

After the MIG has been generated, we will instantiate the MIG and SmartConnect into the example design top file. Open `xilinx_axi_pcie_ep.v` and remove the instantiation of the BRAM Controller, replacing it with instantiations of both the MIG and SmartConnect.



**Important:** If you want to download the top file instead, go [here](#).

```
//INSTANTIATE MIG CORE
mig_7series_5 u_mig_7series_5(

// Memory interface ports
    .ddr3_addr      (ddr3_addr),
    .ddr3_ba        (ddr3_ba),
    .ddr3_cas_n     (ddr3_cas_n),
    .ddr3_ck_n      (ddr3_ck_n),
    .ddr3_ck_p      (ddr3_ck_p),
    .ddr3_cke       (ddr3_cke),
    .ddr3_ras_n     (ddr3_ras_n),
    .ddr3_we_n      (ddr3_we_n),
```

(continues on next page)

(continued from previous page)

```

.ddd3_dq                (ddd3_dq),
.ddd3_dqs_n             (ddd3_dqs_n),
.ddd3_dqs_p             (ddd3_dqs_p),
.ddd3_reset_n           (ddd3_reset_n),
.init_calib_complete    (init_calib_complete),
.ddd3_cs_n              (ddd3_cs_n),
.ddd3_dm                (ddd3_dm),
.ddd3_odt               (ddd3_odt),

// Application interface ports
.ui_clk                 (clk),
.ui_clk_sync_rst        (rst),
.mmcm_locked            (mmcm_locked),
.aresetn                (aresetn),

.app_sr_req(app_sr_req),
.app_ref_req(app_ref_req), //HAD TO ADD THESE MANUALLY
.app_zq_req(app_zq_req),

.app_sr_active          (app_sr_active),
.app_ref_ack            (app_ref_ack),
.app_zq_ack             (app_zq_ack),

// Slave Interface Write Address Ports
.s_axi_awid             (s_axi_awid),
.s_axi_awaddr           (s_axi_awaddr),
.s_axi_awlen            (s_axi_awlen),
.s_axi_awsz            (s_axi_awsz),
.s_axi_awburst          (s_axi_awburst),
.s_axi_awlock           (s_axi_awlock),
.s_axi_awcache          (s_axi_awcache),
.s_axi_awprot           (s_axi_awprot),
.s_axi_awqos            (4'h0),
.s_axi_awvalid          (s_axi_awvalid),
.s_axi_awready          (s_axi_awready),

// Slave Interface Write Data Ports
.s_axi_wdata            (s_axi_wdata),
.s_axi_wstrb            (s_axi_wstrb),
.s_axi_wlast            (s_axi_wlast),
.s_axi_wvalid           (s_axi_wvalid),
.s_axi_wready           (s_axi_wready),

// Slave Interface Write Response Ports
.s_axi_bid              (s_axi_bid),
.s_axi_bresp            (s_axi_bresp),
.s_axi_bvalid           (s_axi_bvalid),
.s_axi_bready           (s_axi_bready),

// Slave Interface Read Address Ports
.s_axi_arid             (s_axi_arid),
.s_axi_araddr           (s_axi_araddr),
.s_axi_arlen            (s_axi_arlen),
.s_axi_arsz            (s_axi_arsz),
.s_axi_arburst          (s_axi_arburst),
.s_axi_arlock           (s_axi_arlock),
.s_axi_arcache          (s_axi_arcache),

```

(continues on next page)

(continued from previous page)

```

.s_axi_arprot      (s_axi_arprot),
.s_axi_arqos       (4'h0),
.s_axi_arvalid     (s_axi_arvalid),
.s_axi_arready     (s_axi_arready),

// Slave Interface Read Data Ports
.s_axi_rid         (s_axi_rid),
.s_axi_rdata       (s_axi_rdata),
.s_axi_rresp       (s_axi_rresp),
.s_axi_rlast       (s_axi_rlast),
.s_axi_rvalid      (s_axi_rvalid),
.s_axi_rready      (s_axi_rready),

// System Clock Ports
.sys_clk_i         (sys_clk_i),
.device_temp       (device_temp),

`ifdef SKIP_CALIB
.calib_tap_req      (calib_tap_req),
.calib_tap_load     (calib_tap_load),
.calib_tap_addr     (calib_tap_addr),
.calib_tap_val      (calib_tap_val),
.calib_tap_load_done (calib_tap_load_done),
`endif

.sys_rst           (sys_rst)
);

assign s_axi_awid = 4'h0; //tie off unneeded ports to 0
assign s_axi_arid = 4'h0;
assign app_sr_req = 1'h0;
assign app_ref_req = 1'h0;
assign app_zq_req = 1'h0;

always @(posedge clk) begin
    aresetn <= ~rst;
end

//INSTANTIATE AXI SMARTCONNECT MODULE
design_1_wrapper u_axi_smartconnect(

    //Master ports going into MIG
    .M00_AXI_0_araddr(s_axi_araddr),
    .M00_AXI_0_arburst(s_axi_arburst),
    .M00_AXI_0_arcache(s_axi_arcache),
    .M00_AXI_0_arlen(s_axi_arlen),
    .M00_AXI_0_arlock(s_axi_arlock),
    .M00_AXI_0_arprot(s_axi_arprot),
    // .M00_AXI_0_arqos(s_axi_arqos),
    .M00_AXI_0_arready(s_axi_arready),
    .M00_AXI_0_arsize(s_axi_arsize),
    .M00_AXI_0_arvalid(s_axi_arvalid),
    .M00_AXI_0_awaddr(s_axi_awaddr),
    .M00_AXI_0_awburst(s_axi_awburst),
    .M00_AXI_0_awcache(s_axi_awcache),
    .M00_AXI_0_awlen(s_axi_awlen),
    .M00_AXI_0_awlock(s_axi_awlock),

```

(continues on next page)

(continued from previous page)

```

.M00_AXI_0_awprot(s_axi_awprot),
//.M00_AXI_0_awqos(s_axi_awqos),
.M00_AXI_0_awready(s_axi_awready),
.M00_AXI_0_awsiz(e)s_axi_awsiz(e),
.M00_AXI_0_awvalid(s_axi_awvalid),
.M00_AXI_0_bready(s_axi_bready),
.M00_AXI_0_bresp(s_axi_bresp),
.M00_AXI_0_bvalid(s_axi_bvalid),
.M00_AXI_0_rdata(s_axi_rdata),
.M00_AXI_0_rlast(s_axi_rlast),
.M00_AXI_0_rready(s_axi_rready),
.M00_AXI_0_rresp(s_axi_rresp),
.M00_AXI_0_rvalid(s_axi_rvalid),
.M00_AXI_0_wdata(s_axi_wdata),
.M00_AXI_0_wlast(s_axi_wlast),
.M00_AXI_0_wready(s_axi_wready),
.M00_AXI_0_wstrb(s_axi_wstrb),
.M00_AXI_0_wvalid(s_axi_wvalid),

//Slave ports coming from the PCIE
.S00_AXI_0_araddr(m_axi_araddr),
.S00_AXI_0_arburst(m_axi_arburst),
.S00_AXI_0_arcache(m_axi_arcache),
.S00_AXI_0_arlen(m_axi_arlen),
.S00_AXI_0_arlock(m_axi_arlock),
.S00_AXI_0_arprot(m_axi_arprot),
//.S00_AXI_0_arqos(m_axi_arqos),
.S00_AXI_0_arready(m_axi_arready),
.S00_AXI_0_arsize(m_axi_arsize),
.S00_AXI_0_arvalid(m_axi_arvalid),
.S00_AXI_0_awaddr(m_axi_awaddr),
.S00_AXI_0_awburst(m_axi_awburst),
.S00_AXI_0_awcache(m_axi_awcache),
.S00_AXI_0_awlen(m_axi_awlen),
.S00_AXI_0_awlock(m_axi_awlock),
.S00_AXI_0_awprot(m_axi_awprot),
//.S00_AXI_0_awqos(m_axi_awqos),
.S00_AXI_0_awready(m_axi_awready),
.S00_AXI_0_awsiz(e)m_axi_awsiz(e),
.S00_AXI_0_awvalid(m_axi_awvalid),
.S00_AXI_0_bready(m_axi_bready),
.S00_AXI_0_bresp(m_axi_bresp),
.S00_AXI_0_bvalid(m_axi_bvalid),
.S00_AXI_0_rdata(m_axi_rdata),
.S00_AXI_0_rlast(m_axi_rlast),
.S00_AXI_0_rready(m_axi_rready),
.S00_AXI_0_rresp(m_axi_rresp),
.S00_AXI_0_rvalid(m_axi_rvalid),
.S00_AXI_0_wdata(m_axi_wdata),
.S00_AXI_0_wlast(m_axi_wlast),
.S00_AXI_0_wready(m_axi_wready),
.S00_AXI_0_wstrb(m_axi_wstrb),
.S00_AXI_0_wvalid(m_axi_wvalid),

//Clocks and Resets
.aclk1_0(clk), //MIG clock (100MHz)
.aclk_0(axi_aclk_out), //PCIE clock (125MHz)

```

(continues on next page)

(continued from previous page)

```
.aresetn_0 (aresetn) //use MIG reset signal
);
```

We will also add in the necessary MIG ports and parameters that was present in the MIG example design.

```
//INSERT PARAMETERS FOR MIG

//*****
// Traffic Gen related parameters
//*****
parameter BEGIN_ADDRESS      = 32'h00000000,
parameter END_ADDRESS        = 32'h00ffffff,
parameter PRBS_EADDR_MASK_POS = 32'hff000000,
parameter ENFORCE_RD_WR      = 0,
parameter ENFORCE_RD_WR_CMD  = 8'h11,
parameter ENFORCE_RD_WR_PATTERN = 3'b000,
parameter C_EN_WRAP_TRANS    = 0,
parameter C_AXI_NBURST_TEST  = 0,

//*****
// The following parameters refer to width of various ports
//*****
parameter CK_WIDTH           = 1, // # of CK/CK# outputs to memory.
parameter nCS_PER_RANK      = 1, // # of unique CS outputs per rank for phy
parameter CKE_WIDTH         = 1, // # of CKE outputs to memory.
parameter DM_WIDTH          = 1, // # of DM (data mask)
parameter ODT_WIDTH         = 1, // # of ODT outputs to memory.
parameter BANK_WIDTH        = 3, // # of memory Bank Address bits.
parameter COL_WIDTH         = 10, // # of memory Column Address bits.
parameter CS_WIDTH          = 1, // # of unique CS outputs to memory.
parameter DQ_WIDTH          = 8, // # of DQ (data)
parameter DQS_WIDTH         = 1,
parameter DQS_CNT_WIDTH     = 1, // = ceil(log2(DQS_WIDTH))
parameter DRAM_WIDTH        = 8, // # of DQ per DQS
parameter ECC               = "OFF",
parameter ECC_TEST          = "OFF",
parameter nBANK_MACHS       = 4,
parameter RANKS             = 1, // # of Ranks.
parameter ROW_WIDTH         = 14, // # of memory Row Address bits.
parameter ADDR_WIDTH        = 28, // # = RANK_WIDTH + BANK_WIDTH + ROW_WIDTH + _
↳COL_WIDTH;

// Chip Select is always tied to low for_
↳single rank devices

//*****
// The following parameters are mode register settings
//*****
parameter BURST_MODE        = "8", // DDR3 SDRAM:
// Burst Length (Mode Register 0).
// # = "8", "4", "OTF".

//*****
// The following parameters are multiplier and divisor factors for PLLE2.
// Based on the selected design frequency these parameters vary.
//*****
parameter CLKIN_PERIOD      = 5000, // Input Clock Period
parameter CLKFBOUT_MULT     = 4, // write PLL VCO multiplier
```

(continues on next page)

(continued from previous page)

```

parameter DIVCLK_DIVIDE          = 1, // write PLL VCO divisor
parameter CLKOUT0_PHASE          = 315.0, // Phase for PLL output clock (CLKOUT0)
parameter CLKOUT0_DIVIDE         = 1, // VCO output divisor for PLL output clock
↳ (CLKOUT0)
parameter CLKOUT1_DIVIDE         = 2, // VCO output divisor for PLL output clock
↳ (CLKOUT1)
parameter CLKOUT2_DIVIDE         = 32, // VCO output divisor for PLL output clock
↳ (CLKOUT2)
parameter CLKOUT3_DIVIDE         = 8, // VCO output divisor for PLL output clock
↳ (CLKOUT3)
parameter MMCM_VCO               = 800, // Max Freq (MHz) of MMCM VCO
parameter MMCM_MULT_F            = 8, // write MMCM VCO multiplier
parameter MMCM_DIVCLK_DIVIDE     = 1, // write MMCM VCO divisor

//*****
// Simulation parameters
//*****
parameter SIMULATION              = "FALSE",
                                   // Should be TRUE during design simulations and
                                   // FALSE during implementations

//*****
// IODELAY and PHY related parameters
//*****

parameter TCQ_MIG                 = 0.1, //100 ps for MIG

parameter DRAM_TYPE               = "DDR3",
//*****
// System clock frequency parameters
//*****
parameter nCK_PER_CLK             = 4, // # of memory CKs per fabric CLK

//*****
// AXI4 Shim parameters
//*****
parameter C_S_AXI_ID_WIDTH        = 4, // Width of all master and slave ID
↳ signals. # = >= 1.
parameter C_S_AXI_ADDR_WIDTH      = 27, // Width of S_AXI_AWADDR, S_AXI_ARADDR,
↳ M_AXI_AWADDR and M_AXI_ARADDR for all SI/MI slots. # = 32.
parameter C_S_AXI_DATA_WIDTH      = 32, // Width of WDATA and RDATA on SI slot.
↳ Must be <= APP_DATA_WIDTH. # = 32, 64, 128, 256.
parameter C_S_AXI_SUPPORTS_NARROW_BURST = 0, // Indicates whether to instantiate
↳ upsizer. Range: 0, 1

//*****
// Debug parameters
//*****
parameter DEBUG_PORT              = "OFF", // # = "ON" Enable debug signals/controls.
↳ "OFF" Disable debug signals/controls.
parameter RST_ACT_LOW             = 0 // =1 for active low reset, =0 for active high.

) (

output [3:0] pci_exp_txp,

```

(continues on next page)

(continued from previous page)

```

output [3:0] pci_exp_txn,
input [3:0] pci_exp_rxp,
input [3:0] pci_exp_rxn,

input sys_clk_p,
input sys_clk_n,
input sys_rst_n, //ACTIVE LOW

//INSERT INPUTS/OUTPUTS FOR MIG
// Inouts

inout [7:0] ddr3_dq,
inout [0:0] ddr3_dqs_n,
inout [0:0] ddr3_dqs_p,

// Outputs
output [13:0] ddr3_addr,
output [2:0] ddr3_ba,
output ddr3_ras_n,
output ddr3_cas_n,
output ddr3_we_n,
output ddr3_reset_n,
output [0:0] ddr3_ck_p,
output [0:0] ddr3_ck_n,
output [0:0] ddr3_cke,
output [0:0] ddr3_cs_n,
output [0:0] ddr3_dm,
output [0:0] ddr3_odt,

// Single-ended system clock
input sys_clk_i,
output tg_compare_error,
output init_calib_complete,

// System reset - Default polarity of sys_rst pin is Active Low.
// System reset polarity will change based on the option
// selected in GUI.
input sys_rst //Active HIGH
);

////////////////////////////////////

//INSERT FUNCTIONS FROM MIG TOP FILE

function integer clogb2 (input integer size);
begin
size = size - 1;
for (clogb2=1; size>1; clogb2=clogb2+1)
size = size >> 1;
end
endfunction // clogb2

function integer STR_TO_INT;
input [7:0] in;
begin
if(in == "8")

```

(continues on next page)



(continued from previous page)

```

        STR_TO_INT = 8;
    else if(in == "4")
        STR_TO_INT = 4;
    else
        STR_TO_INT = 0;
    end
endfunction

//INSERT LOCALPARAMS FROM MIG TOP FILE

localparam DATA_WIDTH          = 8;
localparam RANK_WIDTH = clogb2(RANKS);
localparam PAYLOAD_WIDTH        = (ECC_TEST == "OFF") ? DATA_WIDTH : DQ_WIDTH;
localparam BURST_LENGTH         = STR_TO_INT(BURST_MODE);
localparam APP_DATA_WIDTH       = 2 * nCK_PER_CLK * PAYLOAD_WIDTH;
localparam APP_MASK_WIDTH       = APP_DATA_WIDTH / 8;
//*****
// Traffic Gen related parameters (derived)
//*****
localparam TG_ADDR_WIDTH = ((CS_WIDTH == 1) ? 0 : RANK_WIDTH) + BANK_WIDTH + ROW_
↳ WIDTH + COL_WIDTH;
localparam MASK_SIZE      = DATA_WIDTH/8;
localparam DBG_WR_STS_WIDTH = 40;
localparam DBG_RD_STS_WIDTH = 40;

//INSERT MIG WIRE DECLARATIONS

wire          clk;
wire          rst;
wire          mmcm_locked;
reg           aresetn;
wire          app_sr_active;
wire          app_ref_ack;
wire          app_zq_ack;
wire          app_rd_data_valid;
wire [APP_DATA_WIDTH-1:0] app_rd_data;
wire          mem_pattern_init_done;
wire          cmd_err;
wire          data_mismatch_err;
wire          write_err;
wire          read_err;
wire          test_cmptd;
wire          write_cmptd;
wire          read_cmptd;
wire          cmptd_one_wr_rd;

//ADDITIONAL WIRES NEEDED TO ADD
wire          app_sr_req;

wire          app_ref_req;

wire          app_zq_req;

// Slave Interface Write Address Ports
wire [C_S_AXI_ID_WIDTH-1:0] s_axi_awid;
wire [C_S_AXI_ADDR_WIDTH-1:0] s_axi_awaddr;
wire [7:0] s_axi_awlen;

```

(continues on next page)

(continued from previous page)

```

wire [2:0]          s_axi_awsiz;
wire [1:0]          s_axi_awburst;
wire [0:0]          s_axi_awlock;
wire [3:0]          s_axi_awcache;
wire [2:0]          s_axi_awprot;
wire               s_axi_awvalid;
wire               s_axi_awready;

// Slave Interface Write Data Ports
wire [C_S_AXI_DATA_WIDTH-1:0] s_axi_wdata;
wire [(C_S_AXI_DATA_WIDTH/8)-1:0] s_axi_wstrb;
wire               s_axi_wlast;
wire               s_axi_wvalid;
wire               s_axi_wready;

// Slave Interface Write Response Ports
wire               s_axi_bready;
wire [C_S_AXI_ID_WIDTH-1:0] s_axi_bid;
wire [1:0]          s_axi_bresp;
wire               s_axi_bvalid;

// Slave Interface Read Address Ports
wire [C_S_AXI_ID_WIDTH-1:0] s_axi_arid;
wire [C_S_AXI_ADDR_WIDTH-1:0] s_axi_araddr;
wire [7:0]          s_axi_arlen;
wire [2:0]          s_axi_arsize;
wire [1:0]          s_axi_arburst;
wire [0:0]          s_axi_arlock;
wire [3:0]          s_axi_arcache;
wire [2:0]          s_axi_arprot;
wire               s_axi_arvalid;
wire               s_axi_arready;

// Slave Interface Read Data Ports
wire               s_axi_rready;
wire [C_S_AXI_ID_WIDTH-1:0] s_axi_rid;
wire [C_S_AXI_DATA_WIDTH-1:0] s_axi_rdata;
wire [1:0]          s_axi_rresp;
wire               s_axi_rlast;
wire               s_axi_rvalid;
wire               cmp_data_valid;
wire [C_S_AXI_DATA_WIDTH-1:0] cmp_data; // Compare data
wire [C_S_AXI_DATA_WIDTH-1:0] rdata_cmp; // Read data
wire               dbg_wr_sts_vld;
wire [DBG_WR_STS_WIDTH-1:0] dbg_wr_sts;
wire               dbg_rd_sts_vld;
wire [DBG_RD_STS_WIDTH-1:0] dbg_rd_sts;
wire [11:0]         device_temp;

`ifndef SKIP_CALIB // skip calibration wires
wire               calib_tap_req;
reg               calib_tap_load;
reg [6:0]          calib_tap_addr;
reg [7:0]          calib_tap_val;
reg               calib_tap_load_done;
`endif
assign tg_compare_error = cmd_err | data_mismatch_err | write_err | read_err;

```

In addition, we need to tie some of the MIG input wires to ground, since the SmartConnect itself does not have every connection, as well as initialize the debug ports and calibration logic.

```
//INSERT REMAINING RTL FROM MIG TOP FILE
//*****
// Default values are assigned to the debug inputs
//*****
assign dbg_sel_pi_incdec      = 'b0;
assign dbg_sel_po_incdec      = 'b0;
assign dbg_pi_f_inc           = 'b0;
assign dbg_pi_f_dec           = 'b0;
assign dbg_po_f_inc           = 'b0;
assign dbg_po_f_dec           = 'b0;
assign dbg_po_f_stg23_sel     = 'b0;
assign po_win_tg_rst          = 'b0;
assign vio_tg_rst             = 'b0;

`ifndef SKIP_CALIB

//*****
// Skip calib test logic
//*****
reg[3*DQS_WIDTH-1:0]          po_coarse_tap;
reg[6*DQS_WIDTH-1:0]          po_stg3_taps;
reg[6*DQS_WIDTH-1:0]          po_stg2_taps;
reg[6*DQS_WIDTH-1:0]          pi_stg2_taps;
reg[5*DQS_WIDTH-1:0]          idelay_taps;
reg[11:0]                     cal_device_temp;

always @(posedge clk) begin
    // tap values from golden run (factory)
    po_coarse_tap    <= #TCQ_MIG 'h2;
    po_stg3_taps     <= #TCQ_MIG 'h0D;
    po_stg2_taps     <= #TCQ_MIG 'h1D;
    pi_stg2_taps     <= #TCQ_MIG 'h1E;
    idelay_taps      <= #TCQ_MIG 'h08;
    cal_device_temp  <= #TCQ_MIG 'h000;
end

always @(posedge clk) begin
    if (rst)
        calib_tap_load <= #TCQ_MIG 1'b0;
    else if (calib_tap_req)
        calib_tap_load <= #TCQ_MIG 1'b1;
end

always @(posedge clk) begin
    if (rst) begin
        calib_tap_addr    <= #TCQ_MIG 'd0;
        calib_tap_val     <= #TCQ_MIG po_coarse_tap[3*calib_tap_addr[6:3]+:3]; //'d1;
        calib_tap_load_done <= #TCQ_MIG 1'b0;
    end else if (calib_tap_load) begin
        case (calib_tap_addr[2:0])
            3'b000: begin
                calib_tap_addr[2:0] <= #TCQ_MIG 3'b001;
                calib_tap_val       <= #TCQ_MIG po_stg3_taps[6*calib_tap_addr[6:3]+:6]; //'d19;
            end
        end
    end
end
```

(continues on next page)

(continued from previous page)

```

3'b001: begin
calib_tap_addr[2:0] <= #TCQ_MIG 3'b010;
calib_tap_val      <= #TCQ_MIG po_stg2_taps[6*calib_tap_addr[6:3]+:6]; //
↪ 'd45;
end

3'b010: begin
calib_tap_addr[2:0] <= #TCQ_MIG 3'b011;
calib_tap_val      <= #TCQ_MIG pi_stg2_taps[6*calib_tap_addr[6:3]+:6]; //
↪ 'd20;
end

3'b011: begin
calib_tap_addr[2:0] <= #TCQ_MIG 3'b100;
calib_tap_val      <= #TCQ_MIG idelay_taps[5*calib_tap_addr[6:3]+:5]; // 'd1;
end

3'b100: begin
if (calib_tap_addr[6:3] < DQS_WIDTH-1) begin
calib_tap_addr[2:0] <= #TCQ_MIG 3'b000;
calib_tap_val      <= #TCQ_MIG po_coarse_tap[3*(calib_tap_
↪ addr[6:3]+1)+:3]; // 'd1;
calib_tap_addr[6:3] <= #TCQ_MIG calib_tap_addr[6:3] + 1;
end else begin
calib_tap_addr[2:0] <= #TCQ_MIG 3'b110;
calib_tap_val      <= #TCQ_MIG cal_device_temp[7:0];
calib_tap_addr[6:3] <= #TCQ_MIG 4'b1111;
end
end
end

3'b110: begin
calib_tap_addr[2:0] <= #TCQ_MIG 3'b111;
calib_tap_val      <= #TCQ_MIG {4'h0,cal_device_temp[11:8]};
calib_tap_addr[6:3] <= #TCQ_MIG 4'b1111;
end

3'b111: begin
calib_tap_load_done <= #TCQ_MIG 1'b1;
end
endcase
end
end

//*****skip calib test logic end*****
`endif

```

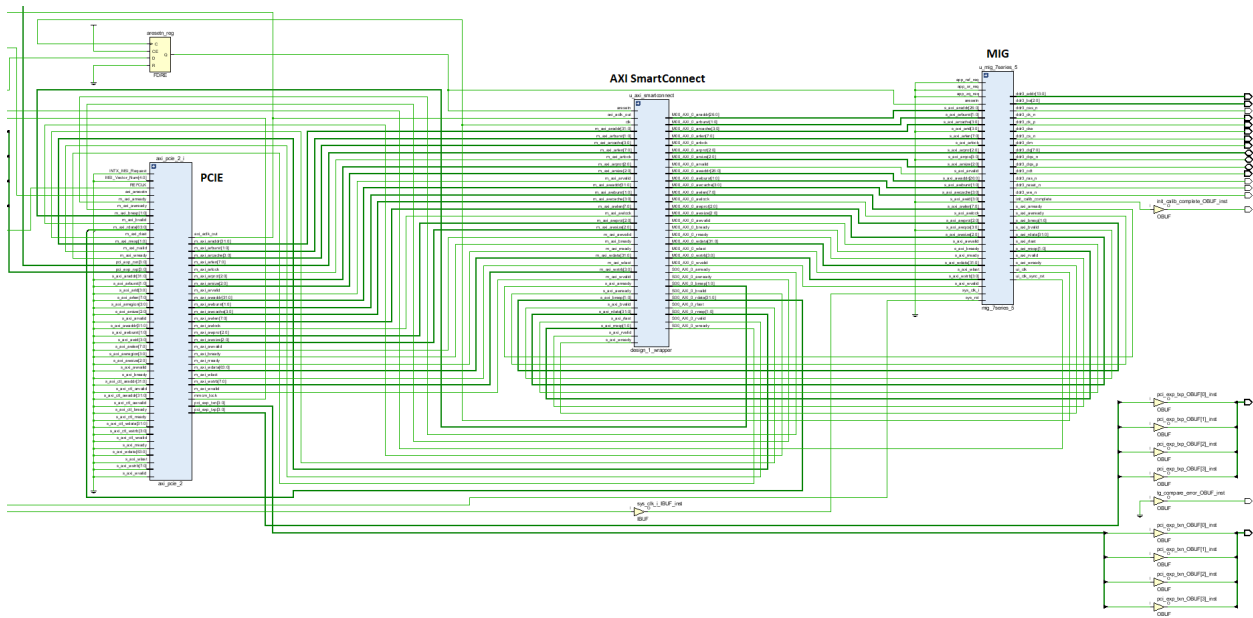


Fig. 3: The complete PCIe and MIG schematic

## 10.5 Simulating the AXI MM PCIe MIG Example Design

After instantiating the MIG into the PCIe's example design, we also need to copy over some modules from the MIG's generated design for the PCIe MIG simulation to run properly. We need to import the relevant DDR3 Memory Model and Wire Delay modules.

**Note:** The MIG 7 Series IP Example Design will output these modules, so generate the design if you have not done so already.

In  
the

Source Traffic\_Generator > AXI\_VIP\_Traffic\_Generator.srcs > sim\_1 > imports > imports

Name	Date modified	Type	Size
ddr3_model.sv	12/9/2020 9:59 AM	SV File	167 KB
ddr3_model_parameters.vh	12/9/2020 9:59 AM	VH File	273 KB
ies_run	12/9/2020 9:59 AM	SH Source File	6 KB
readme	12/9/2020 9:59 AM	Text Document	10 KB
sim.do	12/9/2020 9:59 AM	DO File	7 KB
sim_tb_top.v	12/15/2020 5:58 PM	V File	28 KB
vcs_run	12/9/2020 9:59 AM	SH Source File	6 KB
wiredly.v	12/9/2020 9:59 AM	V File	6 KB
xsim_files.prj	12/9/2020 9:59 AM	PRJ File	18 KB
xsim_options.tcl	12/9/2020 9:59 AM	TCL File	4 KB
xsim_run	12/9/2020 9:59 AM	Windows Batch File	4 KB

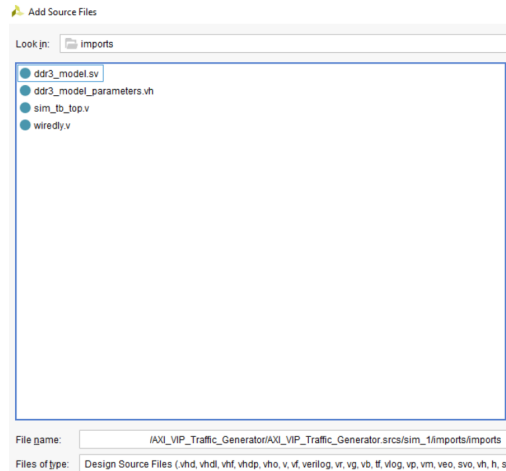
then  
point  
it

to the modules in the MIG Example Design folder located in the user directory. On Windows (or Linux), navigate to the directory where your MIG project is saved.

From there, locate the project's imported directory. An example directory would be similar to `<project name>\srcs\sim_1\imports\imports`.

The directory should look like this:

Select the `ddr3_model sv`, `ddr3_model_parameters.vh`, and `wiredly.v` files to add them to the project.



Modify the simulation top file to properly instantiate these new modules, including all MIG parameters. The example simulation top file can be found [here](#). Make sure to rename `axi_pcie_board.v` to `board.v`!

Run a Behavioral Simulation, making sure to add the property AXI signals for the DUT in the Scope Window (such as the `u_ip_top` module).

---

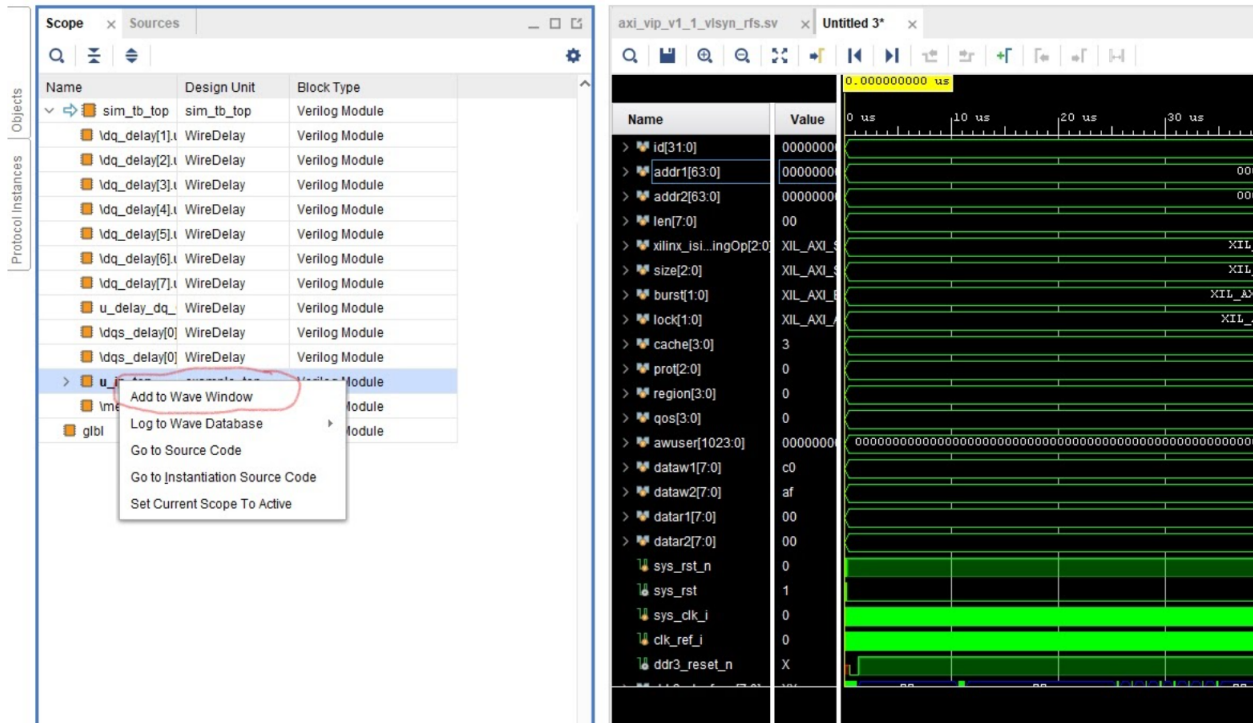
**Important:** Remember to run the command `run -all` in the TCL console to allow the simulation to fully complete!

---

- The MIG will take about 120 us to fully calibrate. Afterwards, the `init_calib_complete` pin will go HIGH, outputting this message in the TCL console.

```
board.mem_rnk[0].gen_mem[0].u_comp_ddr3.cmd_task: at time 120768564.0 ps INFO: Refresh
board.mem_rnk[0].gen_mem[0].u_comp_ddr3.cmd_task: at time 122328564.0 ps INFO:
↪Activate bank 0 row 0000
PHY_INIT: Write Calibration completed at 124203100.0 ps
board.mem_rnk[0].gen_mem[0].u_comp_ddr3.cmd_task: at time 125424564.0 ps INFO:
↪Precharge All
board.mem_rnk[0].gen_mem[0].u_comp_ddr3.cmd_task: at time 125424564.0 ps INFO:
↪Precharge bank 0
MIG Calibration Done
```

- Around 200 us, the PCIe Endpoint will also fully calibrate. The simulated Root Port Complex will then begin to send Transaction Layer Packets (TLPs) to the PCIe Endpoint signaling for a read and a write to the DDR3 memory.
- The Endpoint will then convert these TLPs to the correct AXI Memory Mapped read/write signals and send these through the SmartConnect into the MIG. Eventually, the MIG will receive these AXI requests on its AXI Slave port and subsequently perform the desired reads/writes to the simulated DDR3 memory.



- If successful, the TCL console will output this message:

```
[187477264.0 ps] : TSK_PARSE_FRAME on Receive
[187781296.0 ps] : Test PASSED --- Write Data: 01020304 successfully received
[187821329.0 ps] : Finished transmission of PCI-Express TLPs
Test Completed Successfully
$finish called at time : 187821329 ps : File "..."
```

Like the original PCIe example design simulation, this test writes the data 0x01020304 to address 0x00000010. It then reads the data back from the same address, verifying that it is the same value. If your simulation looks like this, congratulations! You have successfully implemented a PCIe Endpoint with a MIG Controller.



Fig. 4: Successful PCIe Simulation



## DMA/BRIDGE FOR PCIE IP OVERVIEW

### 11.1 DMA IP Overview

Xilinx's DMA/Bridge Subsystem for PCI Express IP is an alternative to the AXI Memory Mapped to PCI Express IP, which was used previously in the "AXI Memory Mapped to PCI Express" section. It still provides a customizable PCIe interface to the FPGA, but this IP also utilizes the DMA (Direct Memory Access) protocol.

Xilinx's user guide for this IP can be found [here](#) and Xilinx also provides an XDMA driver that can be used to interface with this IP over Windows 10 or Linux OS. This particular driver can be found [here](#), and a helpful guide to using this driver can be found [here](#). Essentially, instead of manually performing read and write operations to specific addresses (this is called Programmable Input/Output or PIO), DMA provides a much more efficient way to handle data transactions. The DMA PCIe IP core also provides an AXILite interface and an AXI Memory Mapped "Bypass" interface for simple PIO operations.

### 11.2 The DMA Protocol

The entire goal of DMA is to make data transactions more efficient and less work for the CPU. Instead of having to specify each transaction with the corresponding data and address, DMA allows for transfer of large batches of data, all independent of the CPU. In general, the DMA protocol works like this:

1. User specifies whether the transaction will be a "Host to Card" (H2C) or "Card to Host". Data moving from the host machine to the FPGA will be considered H2C in this case.
2. User specifies the starting source address and starting destination address of the data, as well as the size of the transfer in bytes. If the transaction is H2C, then the source address will be in PCIe memory space and the destination address will be in AXI memory space.
3. The CPU programs the DMA engine with this information, and then the DMA begins transferring all of the data. Once the data has finished transferring, the DMA will send an interrupt back to the CPU.

This specific type of DMA engine is known as a "Scatter Gather" DMA, which means that the target data, source address, destination address, and transfer length are all configured in registers known as "Descriptors". Each descriptor is stored in host memory, and they act as pointers to designated buffers within host memory based on the source/destination address and the transfer length specified. Here is an image from the Xilinx XDMA Driver Debugging Guide, which shows the exact configuration of the Descriptor registers:

Once a descriptor register has been filled out with the correct information, then it is ready to perform the data transaction. In order to start the transaction, the user will need to write to a DMA control register (register 0x04 specifically) in order to enable the transaction. When the data is finished transmitting, the DMA will send an interrupt to the CPU, acknowledging the end of the transfer. More information about the DMA control registers can be found in Xilinx's User Guide for the DMA PCIe IP.

### Descriptor Format

Table 1 shows descriptor formats. Descriptors reside in the host memory. Each descriptor has a source address, destination address, length, and a pointer to the next descriptor on the list unless the STOP bit is set. The Next\_adjacent field indicates how many contiguous descriptors are in the next descriptor address.

Offset	Field	Bit Index	Sub Field	Description
0x0	Magic	15:0		16'had4b. Code to verify that descriptor is valid.
	Nxt_adj	5:0		The number of additional adjacent descriptors after the descriptor located at the next descriptor address field. A block of adjacent descriptor cannot cross a 4K boundary.
	Control	4	EOP	End of packet (AXI ST C2H only)
		1	Completed	Set to (1) to interrupt after the engine has completed this descriptor. This requires global IE_DESCRIPTOR_COMPLETED control flag set in the SGDMA control register.
		0	Stop	Set to (1) to stop the engine when it completes this descriptor.
0x04	Len	[27:0]		Descriptor Data length
0x08 & 0x0C	Source Address	63:0		Source address for the DMA transfer
0x10 & 0x14	Destination Address	63:0		Destination address for the DMA transfer
0x18 & 0x1C	Next Descriptor Address	63:0		Address of the next descriptor in the list

Fig. 1: Figure 1: Xilinx DMA Descriptor Format

Below is an image from the “DMA for PCI Express” Youtube video from Xilinx, which outlines the DMA process using the Descriptor registers. Each of the descriptors correspond to an allocated buffer within System Memory, and then that buffer is either filled by data from the FPGA (C2H transfer), or the data from that buffer is transferred to the FPGA (H2C transfer).

## Typical DMA Operation for DMA

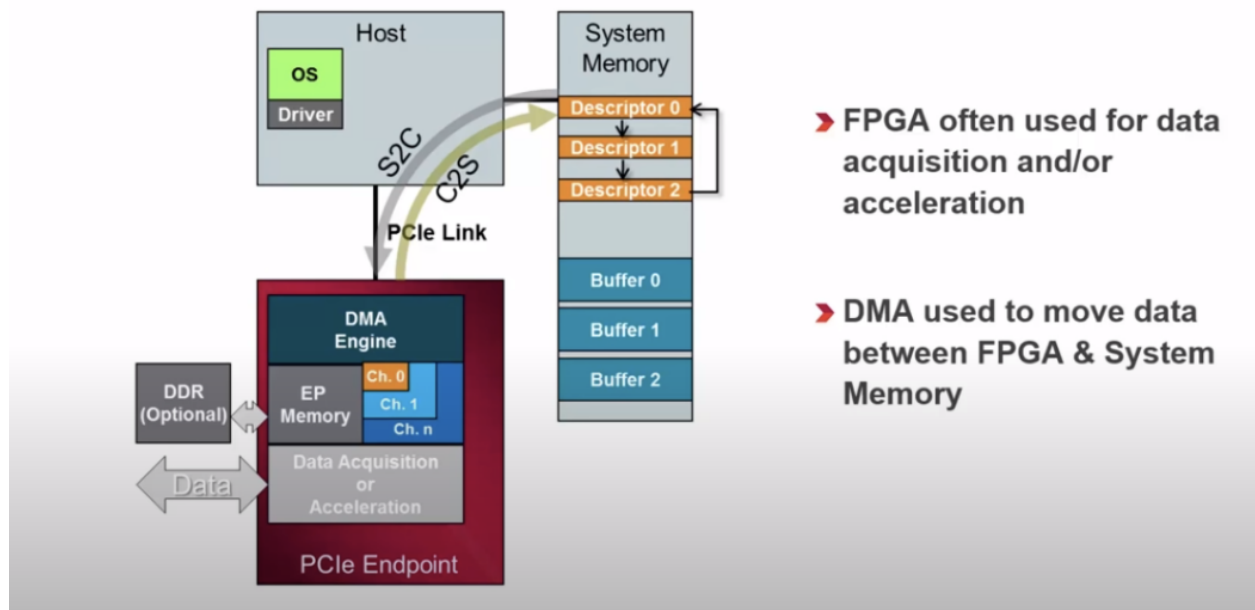


Fig. 2: Figure 2: Typical DMA Operation Diagram

The XDMA Driver is what allows us to be able to read and write to these configuration registers, and Xilinx’s XDMA Driver Debugging guide is a great resource to understand exactly how it works. In brief, here is a short summary from the DMA PCIe User Guide that explains how the driver works to create a H2C transaction:

The process for a C2H Transfer is very similar to these steps, except that the data is being transferred from the user side to the host machine. These steps can be seen below:

## 11.3 Configuring the DMA IP

The user configuration of the DMA/Bridge Subsystem for PCI Express IP is very similar to the AXI Memory Mapped to PCI Express IP. It can be customized by either selecting the IP in the IP Integrator tool or by inserting it into a block diagram. Here are some of the new customization options presented with this IP:

### Basic tab:

- You can select whether you would like to use AXI Memory Mapped or AXI Stream for the DMA interface.
- Just like the AXI Memory Mapped PCIe IP, you can also specify the lane width, link speed, AXI data width, AXI clock frequency, reference clock frequency, and external PIPE interface for faster simulations.

### BARs tab:

---

## Example H2C Flow

In the example H2C flow, `loaddriver.sh` loads devices for all available channels. The `dma_to_device` user program transfers data from host to Card.

The example H2C flow sequence is as follows:

1. Open the H2C device and initialize the DMA.
2. The user program reads the data file, allocates a buffer pointer, and passes the pointer to write function with the specific device (H2C) and data size.
3. The driver creates a descriptor based on input data/size and initializes the DMA with descriptor start address, and if there are any adjacent descriptor.
4. The driver writes a control register to start the DMA transfer.
5. The DMA reads descriptor from the host and starts processing each descriptor.
6. The DMA fetches data from the host and sends the data to the user side. After all data is transferred based on the settings, the DMA generates an interrupt to the host.
7. The ISR driver processes the interrupt to find out which engine is sending the interrupt and checks the status to see if there are any errors. It also checks how many descriptors are processed.
8. After the status is good, the drive returns transfer byte length to user side so it can check for the same.

Fig. 3: Figure 3: H2C Transaction

## Example C2H Flow

In the example C2H flow, `loaddriver.sh` loads the devices for all available channels. The `dma_from_device` user program transfers data from Card to host.

The example C2H flow sequence is as follow:

1. Open device C2H and initialize the DMA.
2. The user program allocates buffer pointer (based on size), passes pointer to read function with specific device (C2H) and data size.
3. The driver creates descriptor based on size and initializes the DMA with descriptor start address. Also if there are any adjacent descriptor.
4. The driver writes control register to start the DMA transfer.
5. The DMA reads descriptor from host and starts processing each descriptor.
6. The DMA fetches data from Card and sends data to host. After all data is transferred based on the settings, the DMA generates an interrupt to host.
7. The ISR driver processes the interrupt to find out which engine is sending the interrupt and checks the status to see if there are any errors and also checks how many descriptors are processed.
8. After the status is good, the drive returns transfer byte length to user side so it can check for the same.

Fig. 4: Figure 4: C2H Transaction

Basic	PCIe ID	PCIe : BARs	PCIe : MISC	PCIe : DMA
<b>Functional Mode</b> <span>DMA</span>				
<b>Mode</b> <span>Basic</span>				
<b>Device / Port Type</b> <span>PCI Express Endpoint device</span>				
<b>PCIe Block Location</b> <span>X1Y0</span>				
<b>PCIe Interface</b>				
<b>Lane Width</b> <span>X1</span>				
<b>Maximum Link Speed</b>				
<input checked="" type="radio"/> 2.5 GT/s <input type="radio"/> 5.0 GT/s				
<b>Reference Clock Frequency (MHz)</b> <span>100 MHz</span>				
<b>AXI Interface</b>				
<b>AXI Address Width</b> <span>64</span> [32 - 64]				
<b>AXI Data Width</b> <span>64 bit</span>				
<b>AXI Clock Frequency</b>				
<input type="radio"/> 62.5 <input checked="" type="radio"/> 125 <input type="radio"/> 250				
<b>DMA Interface option</b>				
<input checked="" type="radio"/> AXI Memory Mapped <input type="radio"/> AXI Stream				
<input type="checkbox"/> AXI-Lite Slave Interface				
<input checked="" type="checkbox"/> Enable PIPE Simulation				
<input type="checkbox"/> Enable GT Channel DRP Ports				
<input type="checkbox"/> Enable PCIe DRP Ports				
<input type="checkbox"/> Additional Transceiver Control and Status Ports				
<input type="checkbox"/> Enable Lane Reversal				

Fig. 5: Figure 5: IP Configuration - “Basic” Tab

Basic	PCIe ID	PCIe : BARs	PCIe : MISC	PCIe : DMA
<input checked="" type="checkbox"/> PCIe to AXI Lite Master Interface				
<input type="checkbox"/> 64bit Enable <input type="checkbox"/> Prefetchable				
Size		64	Scale	Kilobytes
Value		FFFF0000		
PCIe to AXI Translation		0x40000000		
 <input checked="" type="checkbox"/> PCIe to DMA Interface				
<input type="checkbox"/> 64bit Enable <input type="checkbox"/> Prefetchable				
 <input checked="" type="checkbox"/> PCIe to DMA Bypass Interface				
<input type="checkbox"/> 64bit Enable <input type="checkbox"/> Prefetchable				
Size		1	Scale	Gigabytes
Value		C0000000		
PCIe to AXI Translation		0x0000000000000000		

Fig. 6: Figure 6: IP Configuration - “BARs” Tab

- Here, you can choose to enable or disable the AXILite and AXI Bypass Base Address Registers (BARs), as well as specify the desired offset address and size.

**Note:** Based on your selections in this window, the BARs will be created according to this table from the Xilinx IP User Guide:

**Table 3: 32-Bit BARs**

PCIe BARs Selection During IP Customization	BAR0 (32-bit)	BAR1 (32-bit)	BAR2 (32-bit)
Default	DMA		
PCIe to AXI Lite Master enabled	PCIe to AXI4-Lite Master	DMA	
PCIe to AXI Lite Master and PCIe to DMA Bypass enabled	PCIe to AXI4-Lite Master	DMA	PCIe to DMA Bypass
PCIe to DMA Bypass enabled	DMA	PCIe to DMA Bypass	

Fig. 7: Figure 7: BAR Configurations

DMA tab:

Basic	PCIe ID	PCIe : BARs	PCIe : MISC	PCIe : DMA	
Number of DMA Read Channel (H2C)				1	▼
Number of DMA Write Channel (C2H)				1	▼
Number of Request IDs for Read channel (2,4,8,16,32,64)				32	⊗ [2 - 64]
Number of Request IDs for Write channel (2,4,8,16,32)				16	⊗ [2 - 32]
Descriptor Bypass for Read (H2C)				0000	▼
Descriptor Bypass for Write (C2H)				0000	▼
AXI ID Width				4	▼
<input type="checkbox"/> DMA Status Ports					

Fig. 8: Figure 8: IP Configuration - “DMA” Tab

- Here, you can select the number of DMA read and write channels, as well as specify other parameters related to DMA operation.

Seen below is an example configuration of this IP in a typical block diagram. This particular design was generated by the Xilinx “Run Block Automation” tool, and can be easily recreated by following these steps:



1. Open up a new block diagram and place the DMA /Bridge Subsystem for PCI Express IP into the page.
2. Click on the green banner at the top of the screen that says “Run Block Automation”, and then change the “Automation Level” to “Subsystem Level”.
3. Optionally, replace the AXI Interconnect with an AXI Smartconnect for more up-to-date designs.

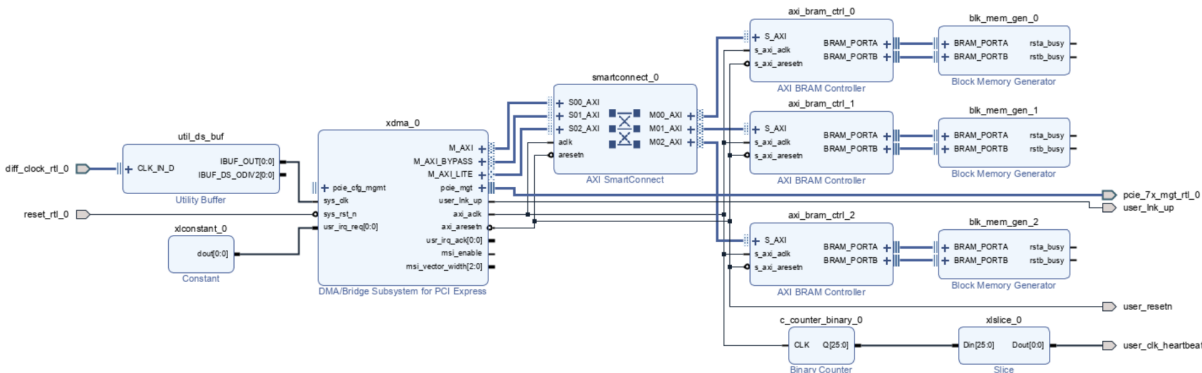


Fig. 9: Figure 9: Example Block Diagram

As we can see from this design, our 100MHz differential reference clock needs to be connected through a IBUFDSGTE utility buffer before it can be connected to the sys\_clk input of the IP. This was also the case for the AXI Memory Mapped to PCI Express IP. We also inserted a constant value of 0 to the usr\_irq\_req port in order to ensure that we are not accidentally sending any unwanted interrupts.

Unlike the AXI Memory Mapped to PCI Express IP, the sys\_rst\_n pin can be directly connected to the PERST (PCIe Reset) pin for an Active Low reset. Then, the axi\_aresetn output should be used to reset all other components driven by the DMA PCIe IP (Smartconnects, BRAM, etc.).

The axi\_aclk output port contains a 125MHz clock, which is the operating frequency of the DMA PCIe core. The external pins coming from the pcie\_mgt output are what physically connect to the PCIe header to allow for the communication of Transaction Layer Packets (TLPs) from the host machine to the PCIe endpoint (FPGA). The user\_link\_up output port is a status port that goes high once a connection has been made between the host machine and the PCIe endpoint.

Lastly, the M\_AXI port is what connects to the device(s) that you would like to interface using the DMA protocol, the M\_AXI\_BYPASS port is what connects to the device(s) that you would like to interface using standard AXI-full PIO protocol, and the M\_AXI\_LITE port is what connects to the device(s) that you would like to interface using AXILITE PIO protocol. In this specific case, we have an AXI BRAM controller connected to each of the three interfaces, and these are mapped into the AXI space as seen in the Address Editor image below:

## 11.4 Additional Resources

Diagram

Address Editor

Address Map

top\_constraints.xdc

design\_1\_wrapper.v

</

Fig. 10: Figure 9: Example Block Diagram (Address Editor)

## CREATING A CUSTOM AXI IP CORE

### 12.1 Packaging Custom IP

Xilinx provides a large library of premade IP cores that cover a multitude of applications. However, sometimes it is best to create and modify our own cores to suit specific needs, such as creating an emulation environment. Instead of instantiating peripherals in Verilog, we can instead take advantage of the plug-and-play nature of the AXI protocol to easily connect them to an AXI master through a SmartConnect. This article will step through the process of packaging a custom peripheral using the IP Integrator.

---

**Note:** If you need a refresher on the AXI protocol, check [here](#).

---

### 12.2 A Simple 8-Bit Counter

As the FPGA ‘Hello World’, a simple 8-bit counter is the perfect introductory example for a custom IP block without the need for a development board.

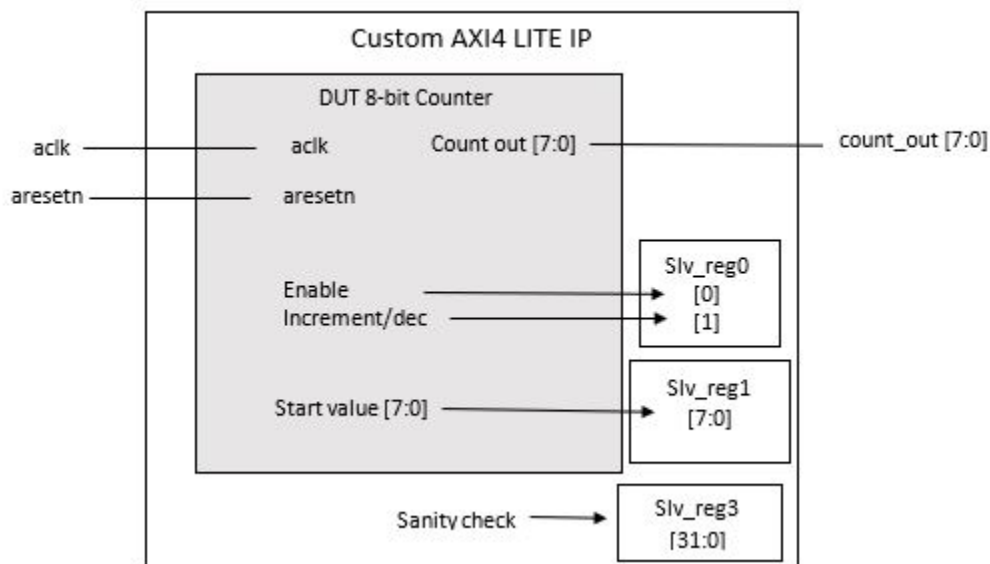
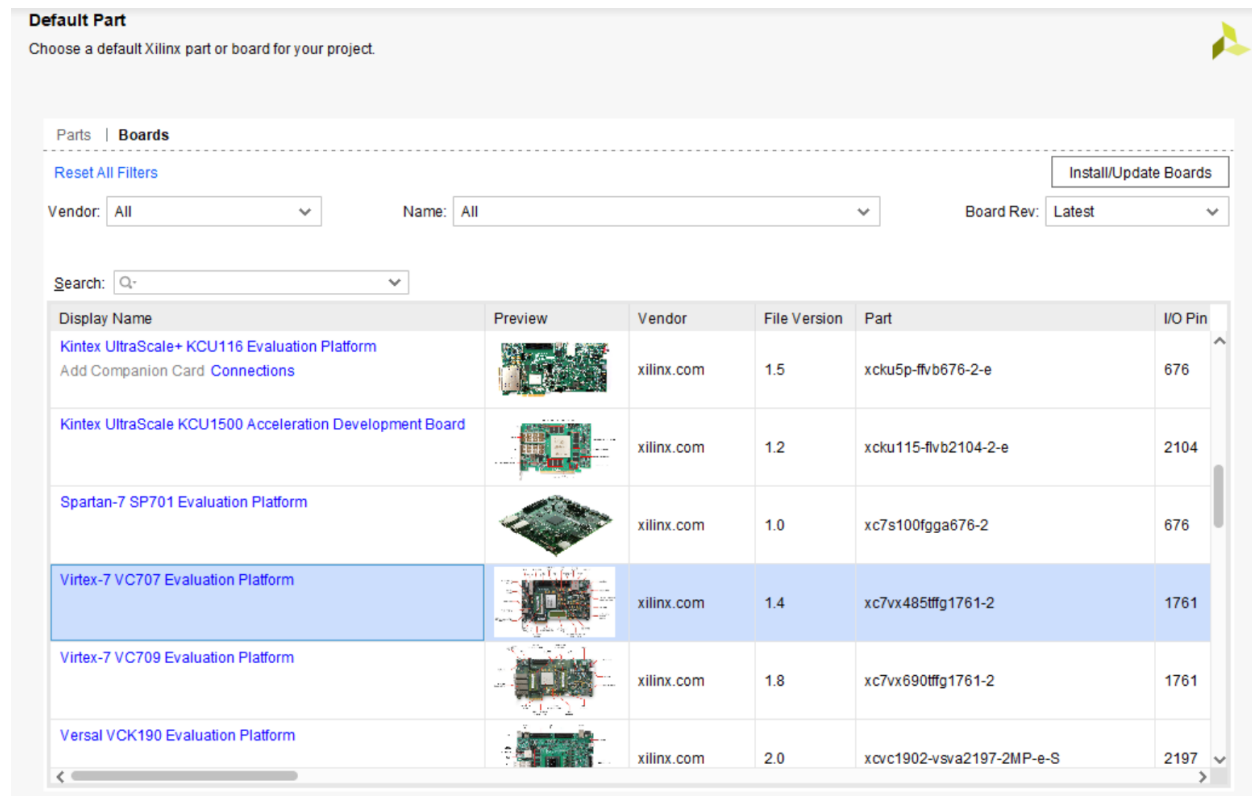


Fig. 1: Block Diagram of our counter

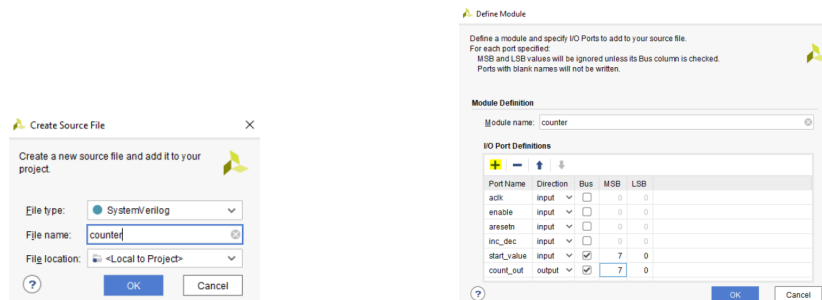
Our first device under test (DUT) will be an 8-bit counter with five inputs: `clock`, `enable`, `reset`, `increment`/`decrement`, and a `start` value, as well as one output - the current count value. The counter will follow these conditions:

- The `enable` flag, when HIGH, will allow the count to change; otherwise, it will keep the same value.
- When a new start value is entered, the counter will automatically start incrementing or decrementing from that new value.
- There is no default start value, so an initial start value must be given.
- The `reset` flag will reset the count to the given start value.

To create this counter, first create a new RTL project and define its directory. For this example, we will use the VC707 evaluation board, but other FPGA boards can be used, like the KC705.



After the project opens, go to *Add Sources* and select *Add or Create Design Sources*. Create a new file, select the desired HDL (we will use SystemVerilog here), and name the file as `counter`. Our new DUT `counter.sv` will be created. A pop-up window will appear, prompting to define a module and specify I/O ports. Customize the counter as so and accept.



Make counter\_out a register and add in the counter logic.

```
//timescale 1ns/1ps

module counter(
  input aclk,
  input enable, //will enable or disable count
  input aresetn, //will reset count back to start value
  input inc_dec, //will indicate wheather increment count or decrement count. inc_
  count is 0, dec count is 1
  input [7:0] start_value, //value to start counting from
  output reg [7:0] count_out //count value
);

//local registers
reg [7:0] count_next; //next count value
reg [7:0] prev_start_value=start_value;

always @(posedge aclk)
begin
  if(aresetn ==0 || prev_start_value!=start_value) //reset mode or new start value
  begin
    count_out =start_value; //reset count out to start value
    prev_start_value=start_value; //set prev start value to start value
  end
  else //reset=1, no reset
  begin
    if(enable==1) //enable is high a
    begin
      if(inc_dec==0) begin//and incdec is low
        count_next=count_out+1; //increment next value
      end
      else begin //inc_dec is high
        count_next=count_out-1; // decrement next value
      end
      count_out=count_next; // set output equal to next value
    end
    else count_out=count_out; //same value if no enable
  end
end
endmodule
```

### Counter Testbench

Now that we have instantiated our design, we will simulate it using a simple testbench.

As a refresher, a test bench is HDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators.

After the project opens, go to *Add Sources* and select *Add or Create Simulation Sources*. Create a new file, select the desired HDL (we will use SystemVerilog here), and name the file as counter\_tb. Our new testbench counter\_tb.sv will be created.

Create a testbench that practices all functions of your custom DUT. For this simple counter example, we will create a testbench that exercises the enable/disable, reset, increment/decrement, start value, and lastly ensure that the counter rolls over correctly.

```
//timescale 1ns/1ps

module counter_tb();
    //create necessary variables
    reg aclk;
    reg enable;
    reg aresetn;
    reg inc_dec;
    reg [7:0]start_value;
    wire[7:0] count_out;

    //create DUT
    counter DUT(
        .aclk(aclk),
        .enable(enable),
        .aresetn(aresetn),
        .inc_dec(inc_dec),
        .start_value (start_value),
        .count_out (count_out)
    );

    //define clk
    always begin
        #5 //delay 5ns
        aclk=~aclk;//should be a 100MHz clk
    end

    initial begin
        //will turn in after 100ns and start inc from af for 100ns
        //then reset and new start value at c0 will increment for 50
        //disbale for 50ns
        //enable again and then decrement
        aresetn=0;//turn on reset
        enable=0;//not enabled
        aclk=0;
        start_value=8'haf;//set a start value
        inc_dec=0;//will increment

        #100 //100ns delay
        aresetn=1;//turn off reset
        #20
        enable=1;//turn on enable

        #100
        aresetn=0;
        start_value=8'hc0;//new start value
        aresetn=1; //lift the reset
        #50
        enable=0;
        #50ns
        enable=1;
        inc_dec=1;

    end
endmodule
```

To run this very simple testbench, follow the instructions below:

1. On the left sidebar, right click on *Run Simulation* and select *Run Behavioral Simulation*.

2. The waveform should have automatically opened. It may be hard to see what you are looking at because the simulation might be very zoomed in. If this is the case, ensure to zoom out so you can see several clock cycles of `aclk`.

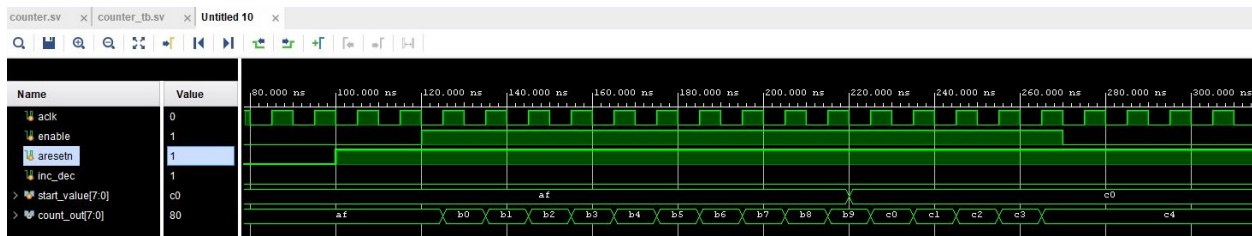


Fig. 2: Working Start Value, Increment, Decrement, and Enable

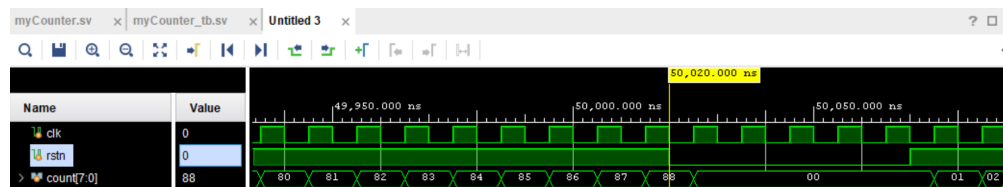


Fig. 3: Working Reset

Once the simple counter slave DUT is working as expected, ensure you know where to find the project files in your computer. we recommend closing the project to avoid confusion while following the future instructions.

## 12.3 Packaging a Custom AXI4Lite IP

This section focuses on how to create a custom AXI4Lite IP and how to correctly instantiate the DUT.

This specific example will focus on a simple counter. Referring to our previous counter block diagram, this is where the `enable`, `increment/decrement`, and `start value` will be mapped to the AXI4Lite slave registers. In addition, the inputs of `aclk` and `aresetn` will be connected to the AXI clock and reset. Afterwards, we will add a sanity check for slave register 3 along with an output port `count_out`. It is important to note that because the reset and registers will be tied to the AXI4Lite slave registers and that `aresetn` will reset all the slave register values. This means that once a reset is performed, the `enable`, `inc/dec`, `start value`, and `count_out` will all be set back to 0. An AXI Write is required to change any values from 0 after a reset has occurred.

Because our custom IP is very simple and does not generate any commands to send to another peripheral, we will be creating only a Slave IP. An example of when we would want to use both a slave and a master interface in the same IP would be if we wanted to have this same counter, but with the additional features of being able to read and write to an external memory, as we will guide you through in this section [here](#)

Create a new project and name as desired. This is the project that will be the main project for this simple counter IP. Select the same board and settings as selected previously. Once the new project is opened, go to *Tools* and select *Create and Package New IP...* A new window will open and explain the features. Click *Next* and select *Create AXI4Peripheral*. Name the IP as desired and select *Next*. Adding and subtracting new interfaces can be done with the `+` and `-` buttons. Remember to select the desired *Interface Type*, *Interface Mode*, *Data Width*, and *Number of Registers*. For a counter, modify the parameters as shown in the image below. These parameters mean that there will be only one interface - an AXI4Lite slave with 4 registers, each with a data width 32 bits. When finished, click *Next*.

Select *Edit IP* and click *Finish*. A new design source and IP Packager will open. The next step is to add our DUT (counter) into this project. In order to do this select *Add Sources* and select *Add or Create Design Sources*. Select *next*

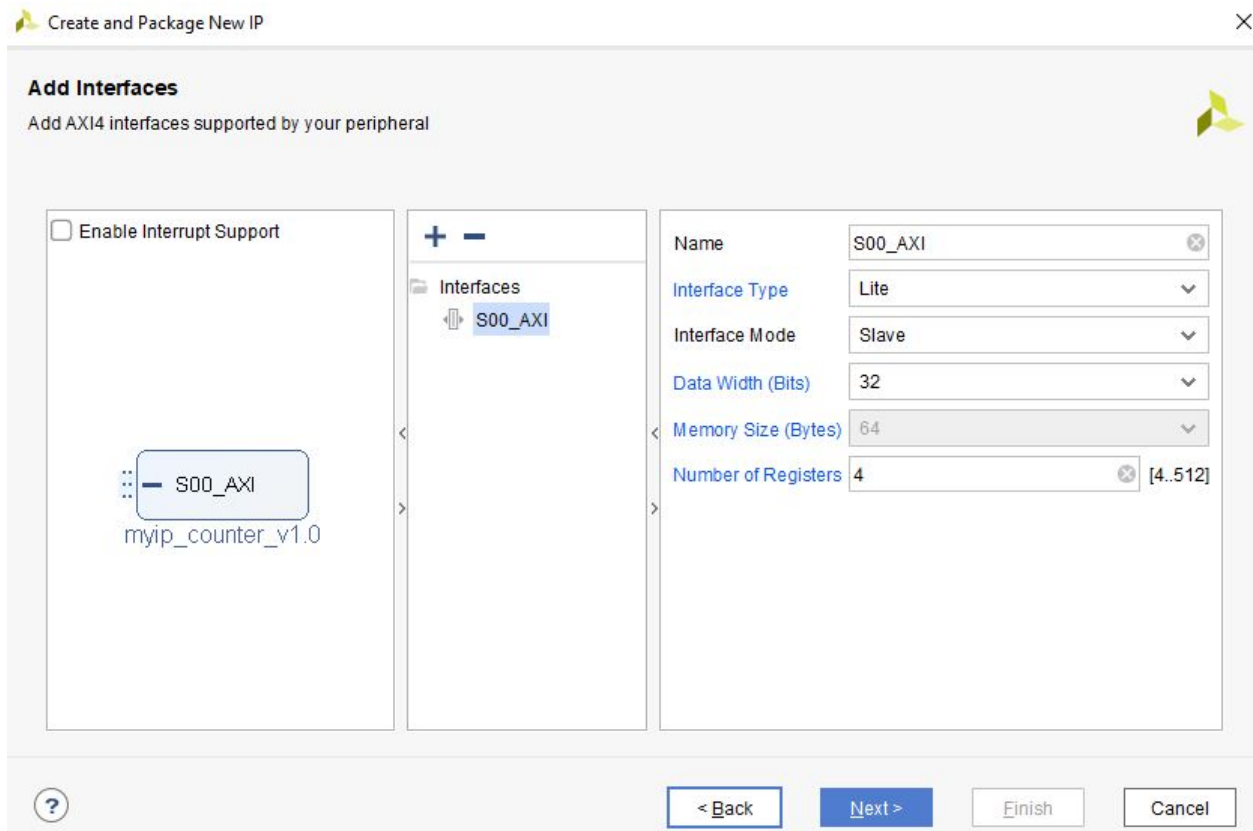


Fig. 4: Add Interfaces to AXI4 peripheral

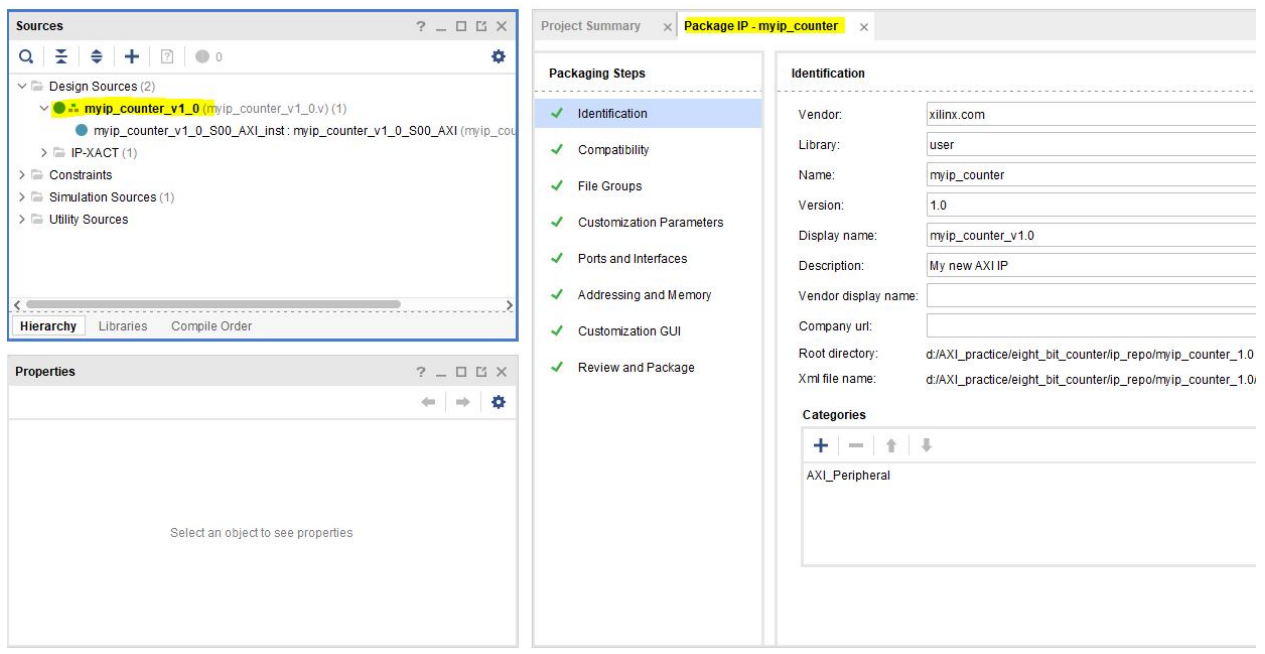


Fig. 5: DUT successfully added to Design Sources



and *Add Files* to add your DUT. Once you have correctly selected your DUT, it will appear in the window. When you select *Finish*, the file will be successfully added to your Sources window.

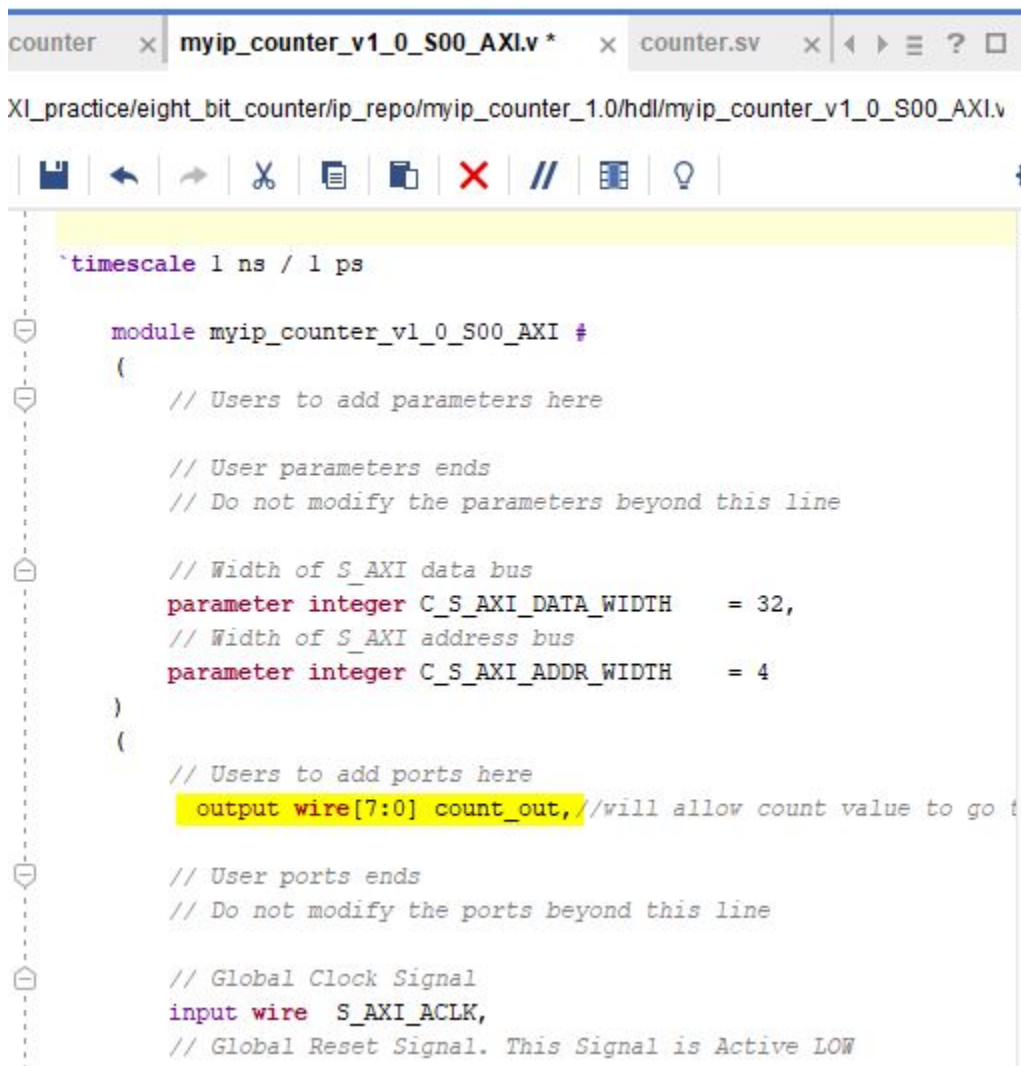
The DUT needs to be correctly instantiated into the custom AXI IP. In order to do this, open the file ending with `_S00_AXI` as it's name. When we created this AXI Peripheral, we selected 4 registers with 32 bits of data. Those registers are shown in this file as `slv_reg0-3`. These are where we are going to store the necessary data for our instantiated DUT. The changes that should be made are listed below:

---

**Important:** If you wish to download the `_S00_AXI` file directly, go [here](#).

---

- Create a User Defined output port, which will be a wire and 8 bits wide, called `count_out`.



```

counter x myip_counter_v1_0_S00_AXI.v * x counter.v x
XI_practice/eight_bit_counter/ip_repo/myip_counter_1.0/hdl/myip_counter_v1_0_S00_AXI.v

`timescale 1 ns / 1 ps

module myip_counter_v1_0_S00_AXI #
(
    // Users to add parameters here

    // User parameters ends
    // Do not modify the parameters beyond this line

    // Width of S_AXI data bus
    parameter integer C_S_AXI_DATA_WIDTH    = 32,
    // Width of S_AXI address bus
    parameter integer C_S_AXI_ADDR_WIDTH    = 4
)
(
    // Users to add ports here
    output wire[7:0] count_out, //will allow count value to go t

    // User ports ends
    // Do not modify the ports beyond this line

    // Global Clock Signal
    input wire  S_AXI_ACLK,
    // Global Reset Signal. This Signal is Active LOW

```

Fig. 6: Adding Output Port to the Slave File

- Scroll down and instantiate the DUT under where it says *Add User Logic Here*. Include the necessary parameters. In this counter, tie `slv_reg0` bit 0 to enable, increment/decrement to `slv_reg0` bit 1 and start value to `slv_reg1` bits 7-0. Tie the clock and reset to the slave AXI clock and reset.
- Use `slv_reg3` as a sanity check. Set `slv_reg3` to `abcd1234`. This means that everytime `slv_reg3` is read it will always be `abcd1234`.

```

401      // Add user logic here
402  counter DUT (
403      .aclk (S_AXI_ACLK),
404      .enable (slv_reg0[0]), //set bit 0 of slv_reg0 to enable
405      .aresetn (S_AXI_ARESETN), //reset as axi slave reset
406      .inc_dec (slv_reg0[1]), //set bit 1 of slv_reg0 as inc/dec setting
407      .start_value (slv_reg1[7:0]), //slv_reg1 bits 7-0 to store start value
408      .count_out (count_out) //count value
409  );
410      // User logic ends
411
412  endmodule
413

```

Fig. 7: Adding User Logic DUT to the Slave File

```

// Implement memory mapped register select and read logic generated by the core
// Slave register read enable is asserted when valid address is decoded
// and the slave is ready to accept the read address.
assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
always @(*)
begin
    // Address decoding for reading registers
    case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        2'h0 : reg_data_out <= slv_reg0;
        2'h1 : reg_data_out <= slv_reg1;
        2'h2 : reg_data_out <= slv_reg2;
        2'h3 : reg_data_out <= 32'habcd1234; //sanity check
        default : reg_data_out <= 0;
    endcase
end

```

Fig. 8: Adding a Sanity Check to the Slave File

The custom counter is correctly instantiated into the file ending in `_S00_AXI`. Now it is necessary to instantiate this counter into the top file. There are two steps necessary to do this:

- We must add our output port `count_out` in the ports of the AXI slave bus interface `S00_AXI`.

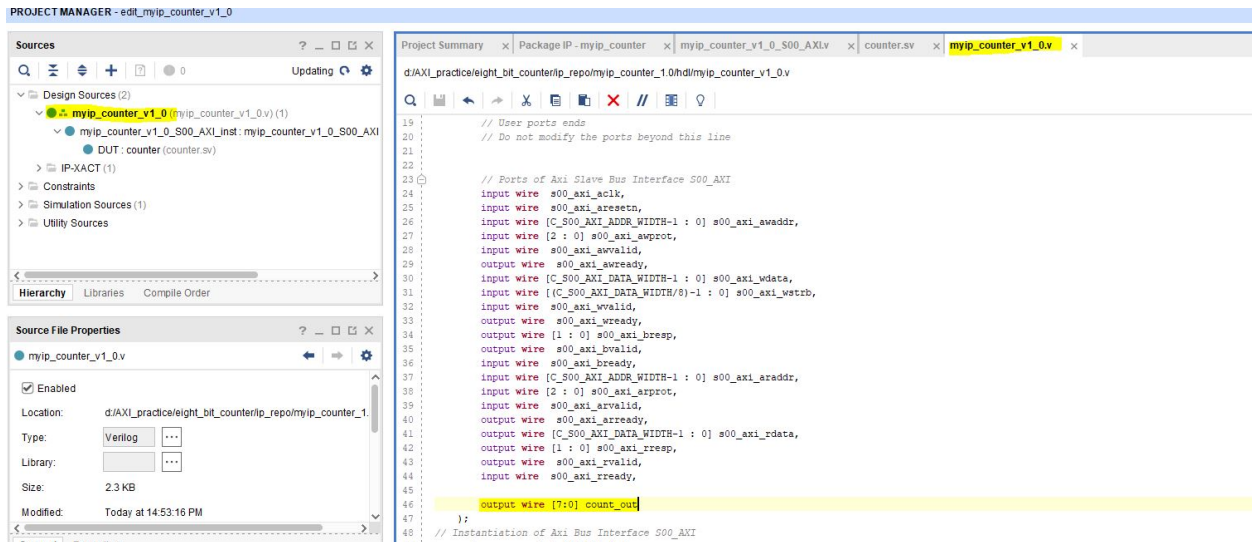


Fig. 9: Adding Output Port to Top File

- Add the ports to the instantiation of the AXI bus interface `S00_AXI`.

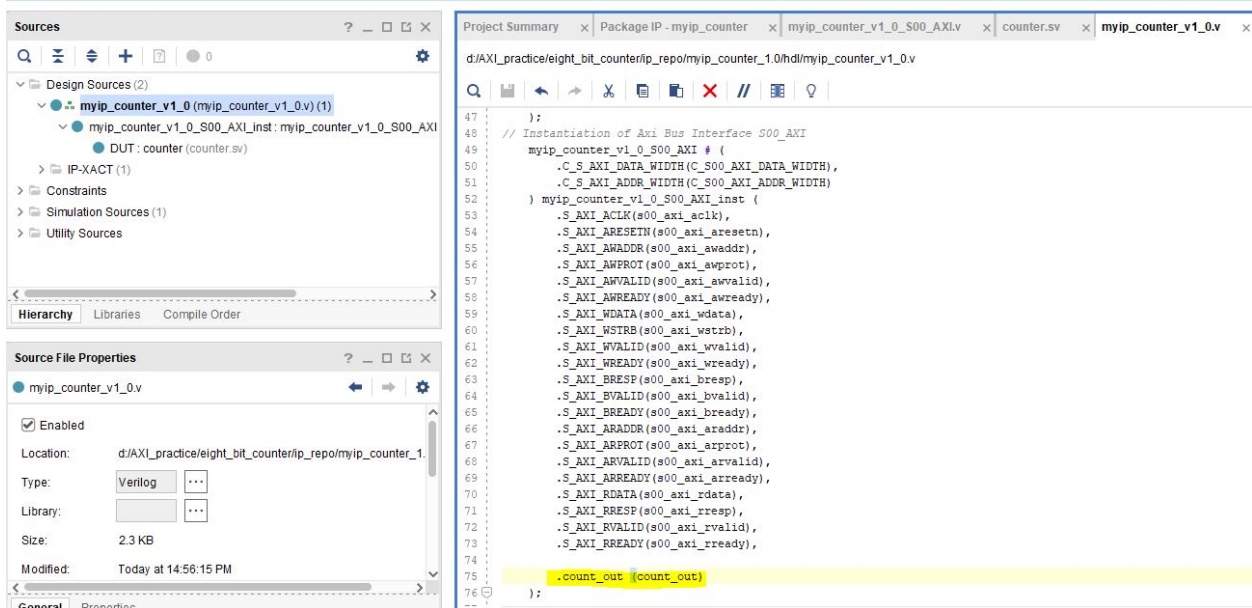


Fig. 10: Adding Ports to AXI Bus Interface in Top

**Important:** If you want to download the top file instead, go [here](#).

Now that the DUT is correctly instantiated, the next step is to open the Package IP tab. Under *Packaging Steps*, verify that every category has a green check mark next to it. In order to achieve this, click on a category such as *File Groups* and select *Merge changes from File Groups Wizard*. This will automatically merge the changes made. Continue this with all of the necessary categories.

Once at the *Review and Package* category, click *IP has been modified* and then click *Re-Package IP* at the bottom of the window. A new window will pop up and tell you the directory of your IP. Keep note of this directory in case you might need to add the repository to a new project.

Once the IP has been correctly packaged, you will be prompted *Do you want to close the project*. Select *Yes* and the project that you were editing the IP will be closed, however, the “main project” that we created at the beginning of this section will still be open.

## 12.4 Adding a Custom AXI IP to a Design

This section will walk through how to add the packaged custom IP to a block diagram and test its functionality with AXI VIP.

If you have been following along with us, congratulations! The custom IP is now correctly packaged! The project made earlier in this tutorial should still be open. The repository for our IP was automatically added to this project, so integrating it into a block design is very straightforward.

- Select *Create New Block Design* and name it as desired.
- A new window will open. Select the + to add IP into the block design. Look for the custom IP that was just created and add it to the block design.

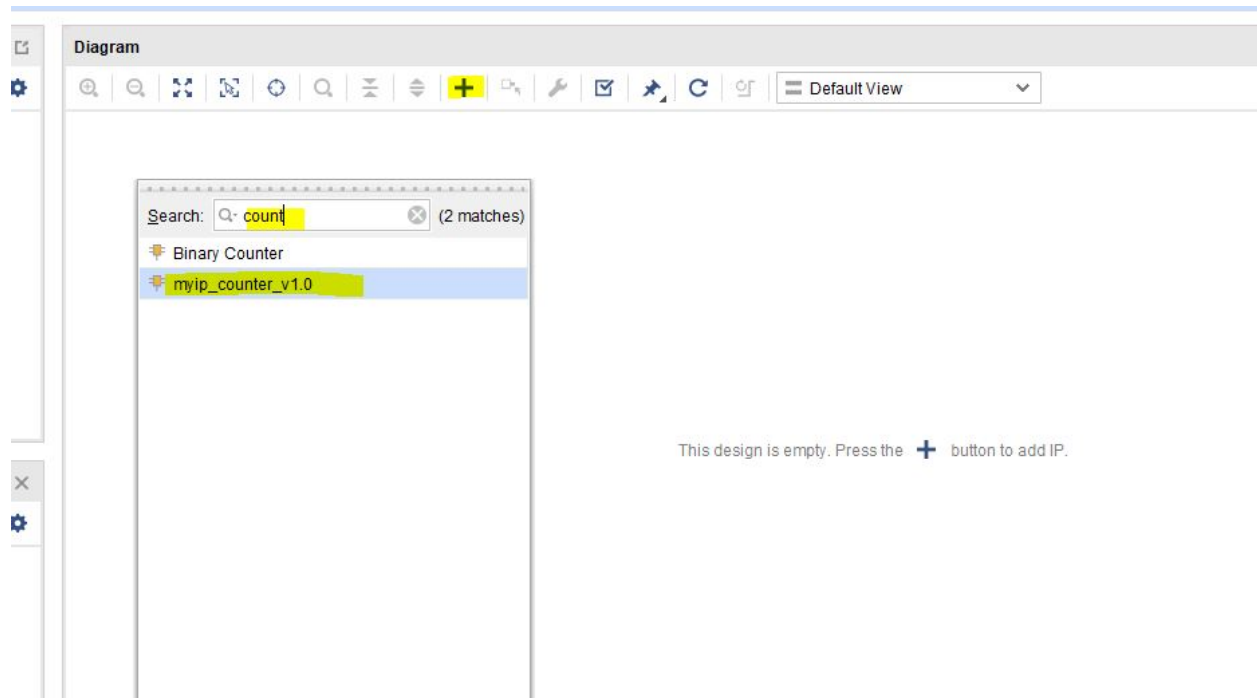


Fig. 11: Add Custom IP to Block Diagram

- Add the AXI VIP from the IP catalog. Double click on the AXI VIP and make it a **Master** and change the interface mode to *manual* for protocol, and change it to **AXI4LITE**. Select *OK*.
- Connect the Master port of the AXI VIP to the slave of the counter.

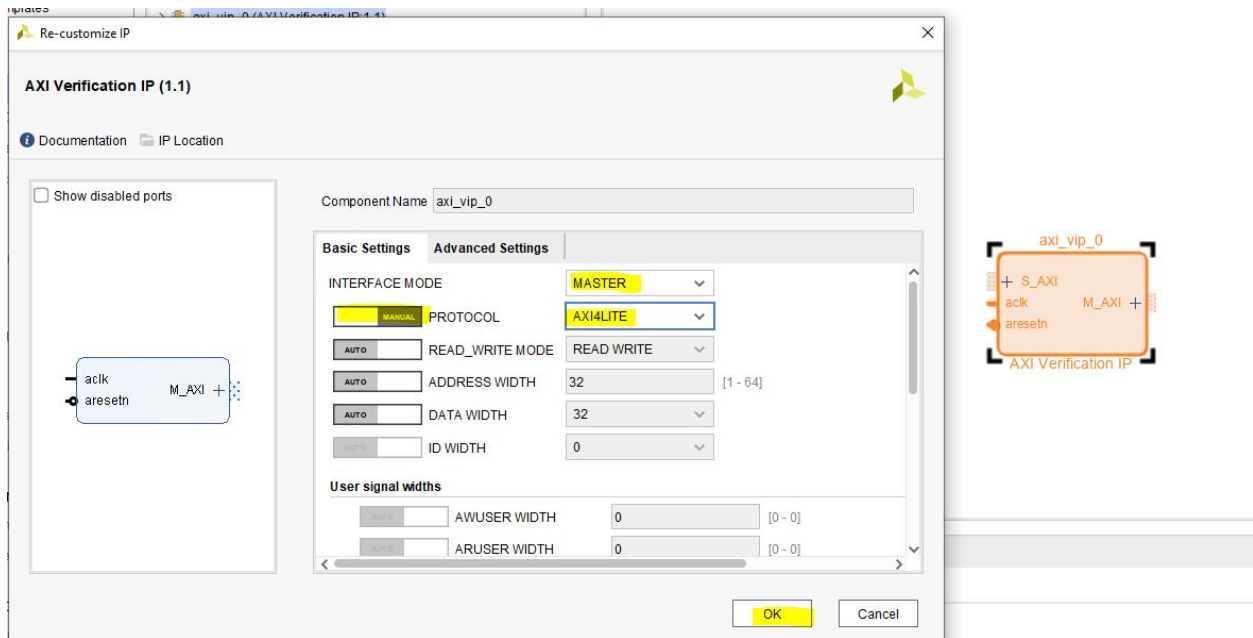


Fig. 12: Add AXI VIP Parameters

- Make the clock and reset ports of the AXI VIP external. In order to do this, right click on the signal such as `aclk` of the AXI VIP and select *Make External*. Once the clock and reset of the AXI VIP are external, drag the clock and reset of the counter IP to connect with the appropriate external signal.
- On the counter IP, make the `count_out` ports external.

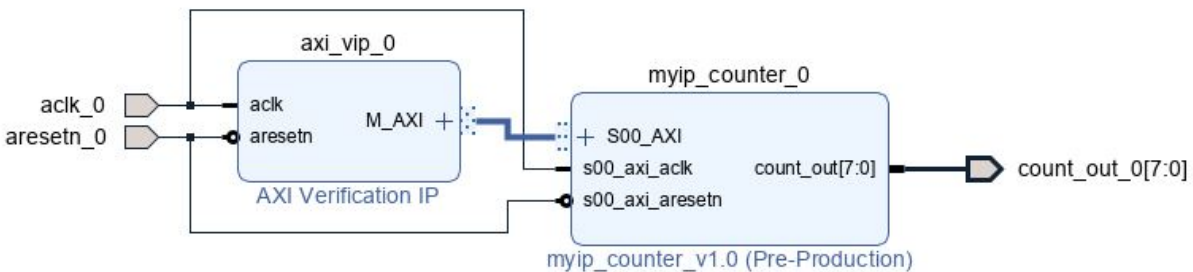


Fig. 13: Simple Counter Block Diagram

- Go to the *Address Editor* tab and right-click on the custom AXI IP. Click *Assign*. This will automatically assign the address range for this IP. Keep note of it for the test bench; for example, the assigned base address may be a hex value like `0x44A0_0000`. If the address editor is not apparent on your screen, complete the next step and you will receive an error to assign addresses, and the address editor will appear.
- Go back to the block diagram and right-click on a blank spot in the design. Select *Validate Design*.
- The next step is to create a wrapper file which turns the block diagram into HDL. To do this go to the *Sources* and right-click on the source for your block diagram (the default name is `design_1` or something similar).



Select *Create HDL Wrapper* and then *Let Vivado manage wrapper and auto-update*.

- The next step is to create a testbench to ensure the custom AXI IP works as intended.

## 12.5 Creating a Testbench for a Custom DUT

This section will walk through the necessary parts to make a testbench.

After the project opens, go to *Add Sources* and select *Add or Create Simulation Sources*. Create a new file, select the desired HDL (we will use SystemVerilog here), and name the file as desired (this example is called `counter_ip_tb`). Our new testbench `counter_ip_tb.sv` will be created.

When using the AXI VIP, there are two packages that you must import.

**Note:** The packages will be underlined in red and appear as a syntax error. This is a Vivado bug and can be safely ignored.

The first package `axi_vip_pkg::*` needs to be copied directly. The second package is a hierarchy path that may be different for you. The file hierarchy should be found from the sources tab.

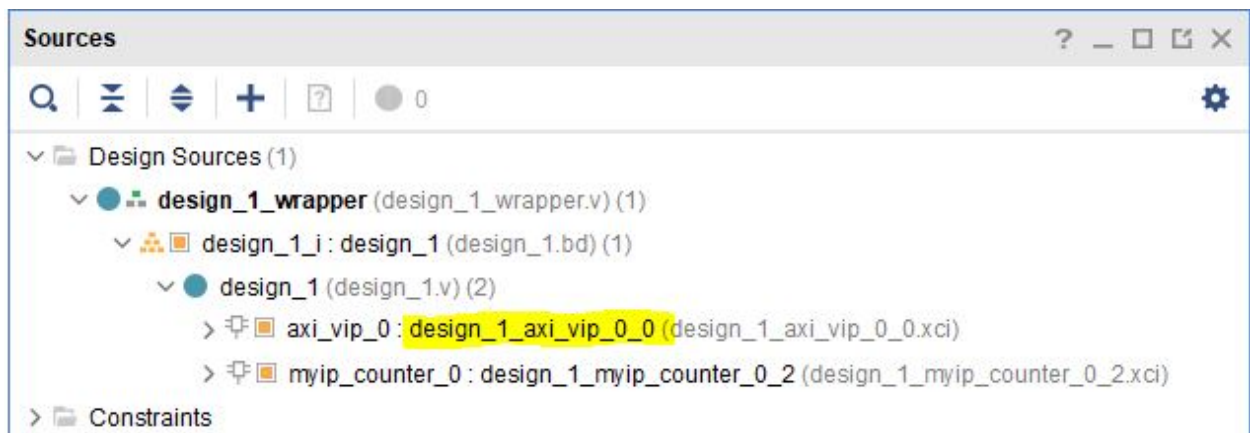


Fig. 14: AXI VIP Component Hierarchy

An example of the two imported packages for this hierarchy are shown below:

```
//import necessary packages
import axi_vip_pkg::*;
import design_1_axi_vip_0_0_pkg::*;

module counter_ip_tb();
```

Fig. 15: Import Packages Example

Be sure to import these packages before the module `your_testbench_name`.

Next, after the autogenerated module `counter_ip_tb()`; (the `counter_ip_tb` will be replaced with what you named your testbench), make sure to add a `clk` and `reset` bit and initialize them both to zero.

After this, create names for both the addresses and data that will be sent (this is optional, you can instead insert the addresses and data directly into the commands).

Next, instantiate the block design from the wrapper file.

From there, it is necessary to create a master agent vip, create an agent, and start the agent (using appropriate hierarchy as well).

```
//initialize AXI Master Agent
//create master agent vip
    design_1_axi_vip_0_0_mst_t    master_agent;

    always begin
        #5
        aclk=~aclk;//100mhz clk
    end
//create agent and start
    initial begin
        master_agent=new("master vip agent",DUT.design_1_i.axi_vip_0.inst.IF);
        master_agent.start_master();
    end
```

Fig. 16: Create and Start Master Agent for VIP

This master agent is for the AXI Verification IP and allows for you to simulate the custom IP receiving read and write transactions. It is important to recognize that the AXI VIP is just for simulation purposes and allows us to test our custom DUT without building the entire infrastructure around it.

From here, create the necessary logic to test all aspects of the custom DUT. It is important to note that this logic will be executed sequentially, so ensure you have delays large enough to allow the necessary transactions enough time to complete. For this simple counter example, the code is provided and also available for direct download below.

Because this simple DUT is an 8-bit counter, with an enable, increment/decrement, start value, and sanity check register the testbench below exercises all of these features. Here is a quick outline of the testbench logic:

- Enables the counter in increment mode
- Write a start value to the counter and read it back to ensure it worked
- Write a new value AF to the start value
- Disable the counter
- Enable it again in increment mode
- Change to decrement mode
- Write new start value 11 to counter
- Read sanity check register (should always be abcd1234 even if write to it)
- Exercise the reset
- Enable counter and read value of start register

---

**Important:** If you want to download the testbench file directly, go [here](#).

---

```

//timescale 1ns / 1ps

//import necessary packages
import axi_vip_pkg::*;
import design_1_axi_vip_0_0_pkg::*;

module counter_ip_tb();

bit aclk = 0;
bit aresetn = 0;
logic[7:0] count_out;

xil_axi_ulong base_reg=32'h44A00000; //slv_reg0 is base reg enable bit is LSB, reg_
↳increment/decrement setting is bit 1
xil_axi_ulong start_value_reg = 32'h44A00004; //reg for start value. slv_reg1 is 4_
↳away from base
xil_axi_ulong sanity_check_reg = 32'h44A0000C; //sanity check reg. slv_reg3 which is_
↳12 away from base reg
xil_axi_prot_t prot = 0;
xil_axi_resp_t resp;
//data to set settings
bit[31:0] enable_data = 32'h00000001; //bit 0 is tied to enable. high will enable._
↳this data will also set inc/dec to increment (0001)
bit[31:0] disable_data=32'h00000000; //disable the enable and inc/dec set back to_
↳increment (0010)
bit[31:0] inc_dec =32'h00000003; //bit 1 is tied to inc/dec. high is decrement. this_
↳will set decrement and enable (0011)
//test data
bit[31:0] test_data1 = 32'h000000C0;
bit[31:0] test_data2 = 32'h000000AF;
bit[31:0] test_data3 = 32'h00000011;
bit[31:0] sanity_data;

//instantiate block design//
design_1_wrapper DUT(
.aclk_0(aclk),
.aresetn_0(aresetn),
.count_out_0(count_out)
);

//initialize AXI Master Agent
//create master agent vip
design_1_axi_vip_0_0_mst_t master_agent;

always begin
#5
aclk=~aclk; //100mhz clk
end
//create agent and start
initial begin
master_agent=new("master vip agent",DUT.design_1_i.axi_vip_0.inst.IF);
master_agent.start_master();

#100
aresetn = 1; //turn off reset

//enable

```

(continues on next page)



(continued from previous page)

```

#50
    master_agent.AXI4LITE_WRITE_BURST(base_reg, prot, enable_data, resp); //write to_
↪enable. increment mode

    //test read and write
    #100
    master_agent.AXI4LITE_WRITE_BURST(start_value_reg, prot, test_data1, resp); //
↪write data c0 into start value register
    #50
    master_agent.AXI4LITE_READ_BURST(start_value_reg, prot, sanity_data, resp); //
↪read start value reg
    #100
    master_agent.AXI4LITE_WRITE_BURST(start_value_reg, prot, test_data2, resp); //
↪write data2 AF into start reg. still increment mode

//test enable/disable
    #50
    master_agent.AXI4LITE_WRITE_BURST(base_reg, prot, disable_data, resp); //disable.
↪increment mode
    #50
    master_agent.AXI4LITE_WRITE_BURST(base_reg, prot, enable_data, resp); //write to_
↪enable. increment mode

    //test decrement
    #100
    master_agent.AXI4LITE_WRITE_BURST(base_reg, prot, inc_dec, resp); //write to_
↪change to decrement mode
    #100
    master_agent.AXI4LITE_WRITE_BURST(start_value_reg, prot, test_data3, resp); //
↪write data3 11 into start value

    //sanity check
    #100
    master_agent.AXI4LITE_READ_BURST(sanity_check_reg, prot, sanity_data, resp); //
↪read sanity check register. should be abcd1234

    //test reset
    #100
    aresetn=0;//enable reset
    #100
    aresetn=1;//turn off reset

    //enable and read start reg, should be 0 after reset
    #100
    master_agent.AXI4LITE_WRITE_BURST(base_reg, prot, enable_data, resp); //write to_
↪enable. increment mode
    #100
    master_agent.AXI4LITE_READ_BURST(start_value_reg, prot, sanity_data, resp); //
↪read start value register

    end
endmodule

```

## 12.6 Interpreting Simulation Waveforms For a Custom DUT

This section will walk through how to understand the waveforms created from running your testbench for your custom DUT.

1. On the left sidebar, right click on *Run Simulation* and select *Run Behavioral Simulation*.

**Note:** If you receive an error, the message will often tell you a file you can locate on your computer that will have more information. I highly recommend looking at this text document because it is very helpful when debugging!

2. Ensure you are running the simulation long enough to see all actions performed in the testbench by using the TCL command `run -all!`
3. The waveform should have automatically opened. Add the desired signals that you would like to analyze to the waveform. For the simple counter simulation, there are some signals we want to add to the waveform. These are found in the left column under *Scope*. The first signal is `axi_vip_0`, this will show the reads and writes that we initiate from the `axi_vip` in our testbench. In order to add a signal to the waveform, right click on the desired signal and choose *Add to Wave Window*. The next group of signals necessary to add to the waveform are for our custom DUT, in this example labeled `myip_counter_0`.

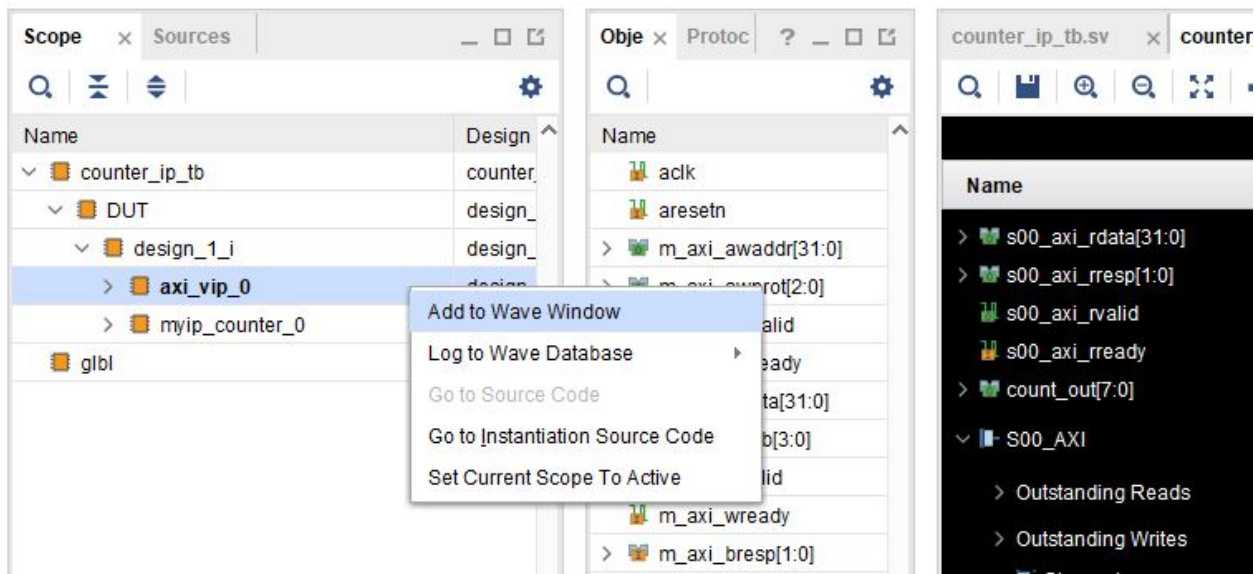


Fig. 17: Add Desired Signals to Waveform

4. Once these signals are added to the waveform, zoom out of the waveform so you can see several clock cycles on the screen.
5. On the waveform, if you hover over the `M_AXI` it will tell you what master axi it is referring to (this will be important once you create more advanced DUTs). The `M_AXI` in this case is referring to the `axi_vip`. This means that in the testbench whenever you use the master agent to perform write or a read it will show up here in the waveform.
6. For this simple counter DUT, the first command we had the `axi_vip` perform was to enable the counter. You can see in the that the `axi_vip` initiated a write to the address of `44A0_0000` and the data it sent was a 1, as shown in the figure below outlined in red.
7. From there, you can scroll down on the waveform to see the `S_00` signals. These are the signals for the slave simple counter. It shows that in the slave, the write address is 0 and the data is 1, which is completed at about

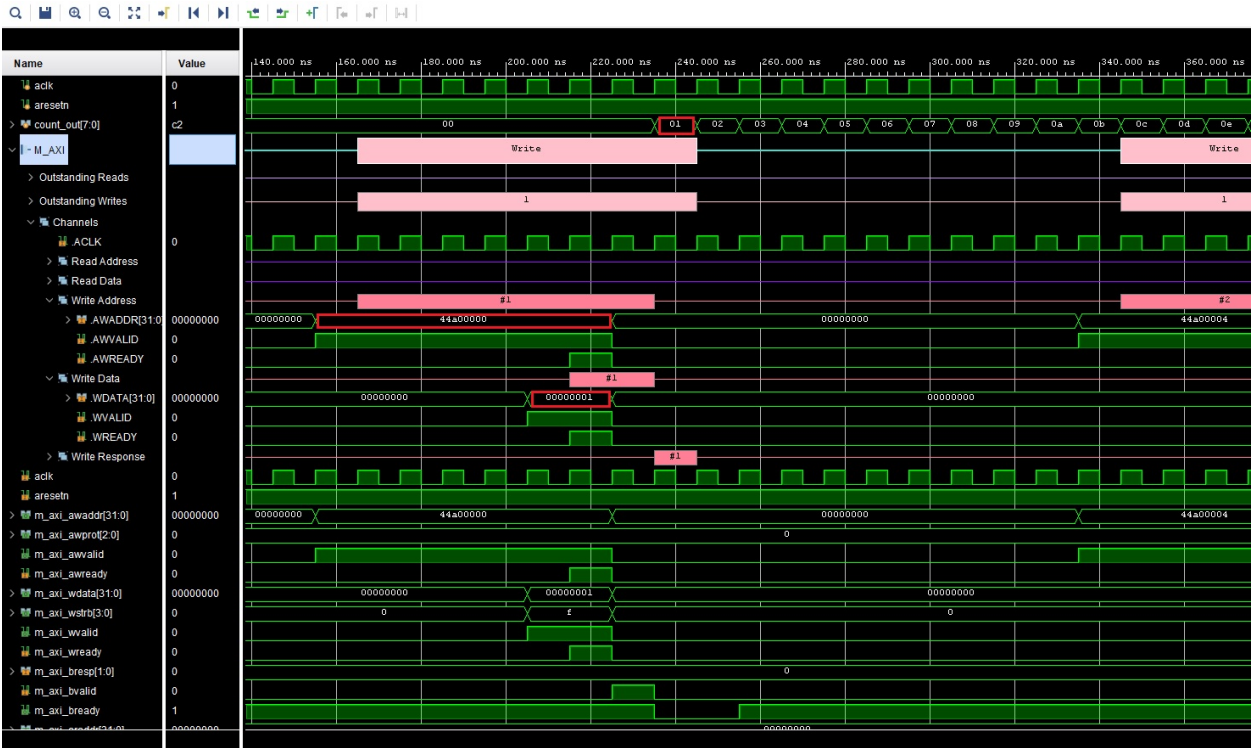


Fig. 18: Simple Counter Waveform of Enable

220ns. This is what we expect because that is what we need to do to start this simple counter IP. If you return to the previous image you can see that the counter began counting at about 240ns. You can continue to read the waveforms in this manner.

**Note:** Ensure you are running the simulation long enough to see all actions performed in the testbench by using the TCL command `run -all!`

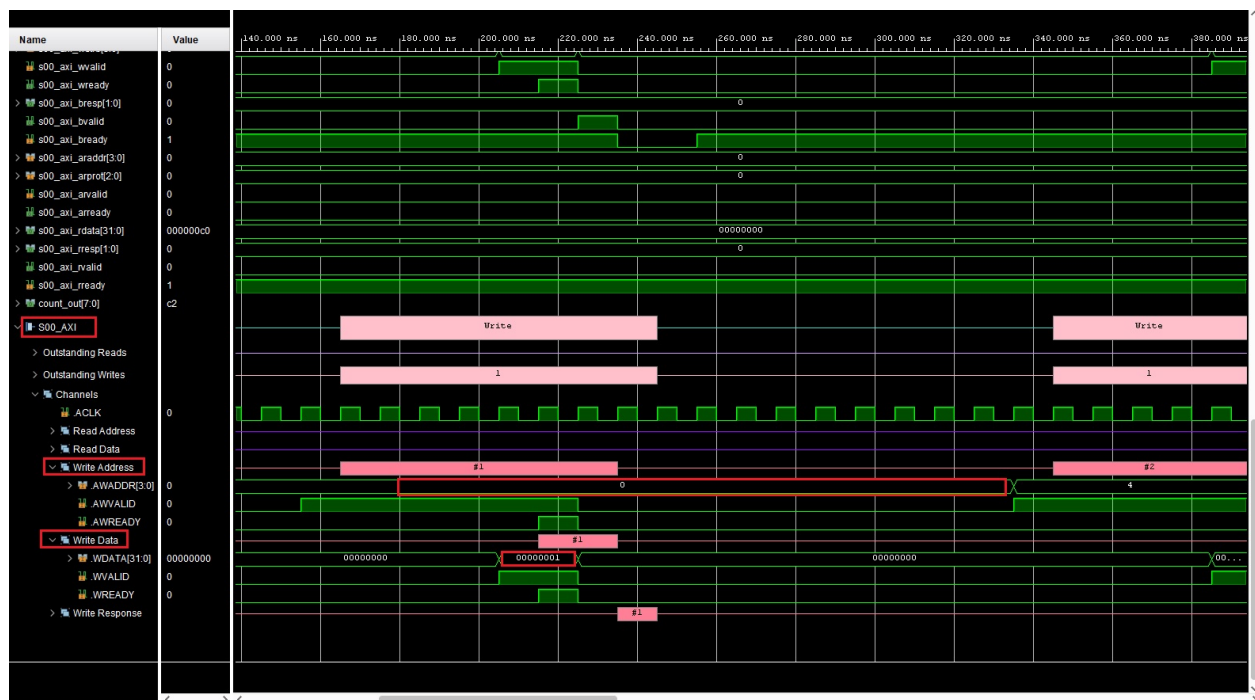


Fig. 19: Simple Counter Waveform of Enable pt.2

## CREATING AN ADVANCED CUSTOM AXI DESCRIPTOR IP

This documentation is a continuation of *Creating a Custom AXI IP Core*. The purpose of this documentation is to take the counter previously created and make it more advanced by allowing it to communicate with memory. In this documentation the DUT will utilize BRAM through the standard IP catalog provided through Xilinx.

### 13.1 Features of Advanced DUT

#### Features of the Descriptor DUT:

- Write the current count value to memory (BRAM addr: 200)
- Read in an initial count value from memory (BRAM addr: 204 [7:0])
- Read in increment or decrement mode from memory (BRAM addr: 204 [8])
- Prompt the DUT to initiate a write or read transaction to memory
- External `count_out` bus port

Here is a brief description of the counter's operation:

The initial counter we previously made takes in a clock and reset signal. When the counter is enabled, it will increment the 8-bit count value on every clock cycle. When `reset` is activated, the counter value will reset to the initial count value.

This more advanced counter will have both a AXI master and a AXI slave when packaged. The slave will take in values/commands and pass them to the master. The master will then take these commands and instantiate them into memory.

In order for the DUT to obtain and send values to/from memory, it is necessary to create some intermediate variables. For example, when the initial count value is read in from memory, we will refer to this as `initial_count_value_i` in the top file, `initial_count_value_in` in the slave file, and `initial_count_value_out` in the master file.

When reading in a value from memory, it obtains data from the `rdata` bus. This means that the data must already be on the `rdata` bus for it to be implemented. For example, when you first want to use the counter, you need to send a 9-bit value directly to BRAM address 204. You would then write to initiate the DUT to perform a read transaction so it will read from BRAM and the data will be loaded onto `rdata`, then you can enable the counter.

The value that is read will be 32 bits in this example. However the only bits that are important to this counter are the lower 9 bits ([8:0]). The last 8 bits ([7:0]) are the initial count value and the 9th bit ([8]) is the increment or decrement mode. Increment is low active.

In this specific example, I made the address editor BRAM set to addr 0 and the DUT to addr 4000\_0000. In addition when I edited the custom IP, I set a slave address offset to address 200. This means that when I initiate the counter to write to BRAM, even though the BRAM is at 0000\_0000, the value will be written to addr 200 because of this offset.

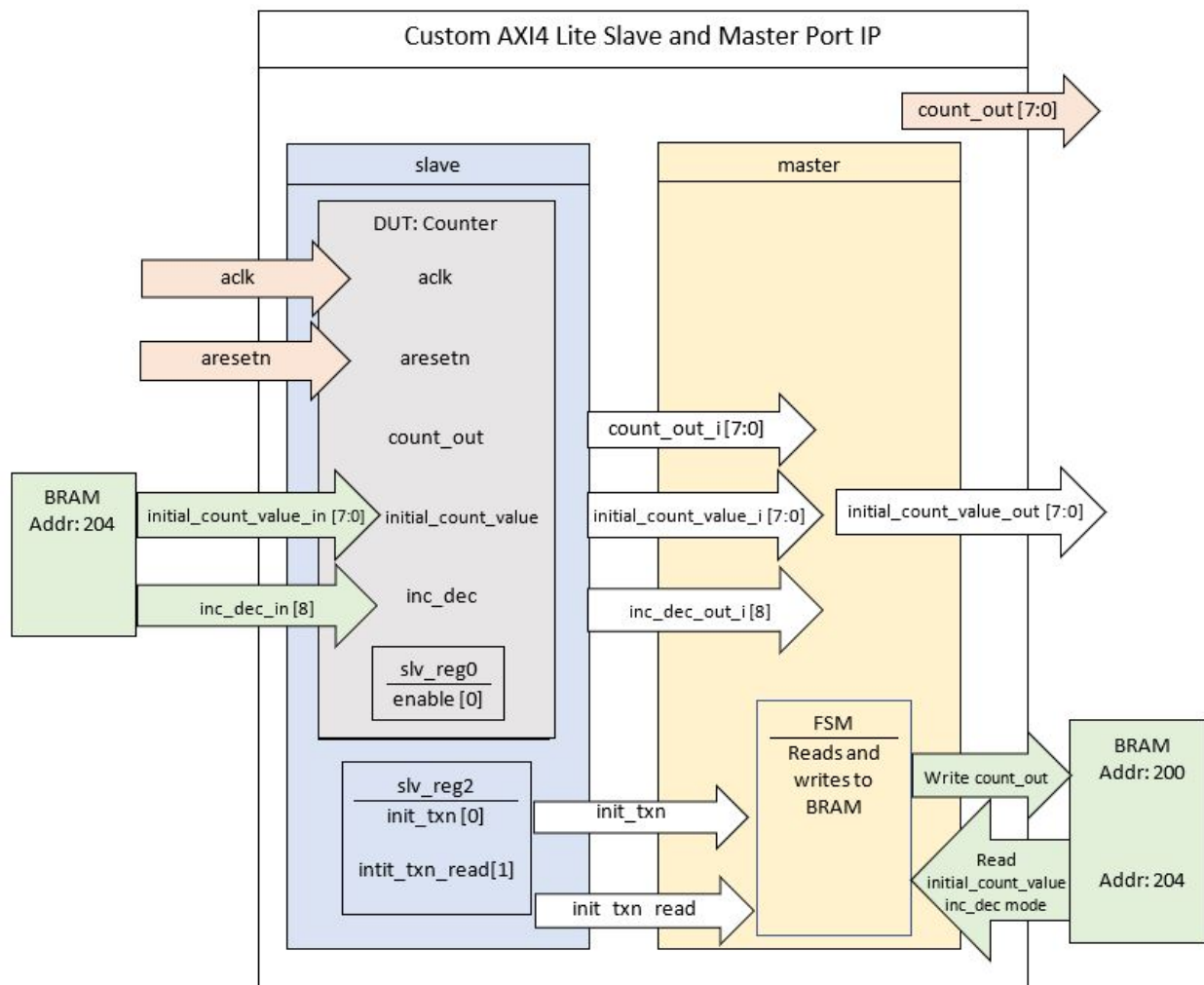


Fig. 1: Block Diagram of the Descriptor Counter

## 13.2 Editing the Descriptor Counter IP

To edit the counter IP, we will directly edit the top file.

---

**Important:** If you wish to download the top file directly, go [here](#).

---

1. Add port output port [7:0] count\_out.
2. Comment out line port for master input wire m00\_axi\_init\_axi\_txn because we want this to be controlled manually from an external port by user.
3. Before the slave instantiation, add the following wires. These are necessary because they are used in the instantiation below so we need to create them before using them.

```
//create wires for instantiation below
wire m00_axi_init_axi_txn;
wire [7:0] count_out_i;
wire [7:0] initial_count_value_i;
wire m00_axi_init_read_txn;
```

4. In the slave instantiation insert the following:

```
.init_txn_read(m00_axi_init_read_txn),
.init_txn(m00_axi_init_axi_txn),
.count_out_i(count_out_i),
.initial_count_value_in(initial_count_value_i) //connect initial_count_value from_
↳Master output to Slave input
```

5. In the master instantiation, add the following lines:

```
.init_axi_txn_read(m00_axi_init_read_txn),
.count_out_i(count_out_i),
.count_out(count_out),
.initial_count_value_out(initial_count_value_i) //connect initial_count_value from_
↳Master output to Slave input
```

Slave File:

---

**Important:** If you wish to download the top file directly, go [here](#).

---

1. Add the following user ports:

```
output wire[7:0] count_out_i,
output wire init_txn,
output wire init_txn_read, //make it an external port
input wire[7:0] initial_count_value_in, //initial count value sent from rdata
```

2. Add the user logic at the bottom of this file. In this example we are instantiating a counter as follows:

```
counter DUT(
    .aclk (S_AXI_ACLK),
    .enable (slv_reg0[0]), //set bit 0 of slv_reg0 to enable
    .aresetn (S_AXI_ARESETN), //reset as axi slave reset
    .inc_dec (slv_reg0[1]), //set bit 1 of slv_reg0 as inc/dec setting
    .start_value (initial_count_value_in), //slv_reg1 bits 7-0 to store start_
    ↳value
```

(continues on next page)

(continued from previous page)

```

        .count_out (count_out_i) //count value
    );
assign init_txn = slv_reg2[0];
assign init_txn_read =slv_reg2[1];

```

Master File:

**Important:** If you wish to download the top file directly, go [here](#).

1. Insert the following ports:

```

input wire [7:0] count_out_i, //intermediate count value
output wire [7:0] count_out,
input wire init_axi_txn_read, //signal to initiate a read
output wire [7:0] initial_count_value_out, // output signal for initial counter value

```

2. Customize the master file to work as desired. In this case we changed the finite state machine and created an initiate read txn that will operate separate from initiating a write txn. The code is below and the changes made are highlighted:

```

`timescale 1 ns / 1 ps
module myip_counter_master_read_v1_0_M00_AXI #
(
    // Users to add parameters here

    // User parameters ends
    // Do not modify the parameters beyond this line

    // The master will start generating data from the C_M_START_DATA_VALUE value
    parameter C_M_START_DATA_VALUE = 32'h00000000,
    // The master requires a target slave base address.
    // The master will initiate read and write transactions on the slave with base_
    ↪address specified here as a parameter.
    parameter C_M_TARGET_SLAVE_BASE_ADDR = :guilabel:`32'h00000200`,

    // Width of M_AXI address bus.
    // The master generates the read and write addresses of width specified as C_M_
    ↪AXI_ADDR_WIDTH.
    parameter integer C_M_AXI_ADDR_WIDTH = 32,
    // Width of M_AXI data bus.
    // The master issues write data and accept read data where the width of the data_
    ↪bus is C_M_AXI_DATA_WIDTH
    parameter integer C_M_AXI_DATA_WIDTH = 32,
    // Transaction number is the number of write
    // and read transactions the master will perform as a part of this example memory_
    ↪test.
    parameter integer C_M_TRANSACTIONS_NUM = 4
)
(
    // Users to add ports here
    :guilabel:`input wire [7:0] count_out_i, //intermediate count value
    output wire [7:0] count_out,
    input wire init_axi_txn_read, //signal to initiate a read
    output wire [7:0] initial_count_value_out, // output signal for initial counter_
    ↪value`

```

(continues on next page)



(continued from previous page)

```

// User ports ends
// Do not modify the ports beyond this line

// Initiate AXI transactions
:guilabel:`input wire  INIT_AXI_TXN,`
// Asserts when ERROR is detected
output reg  ERROR,
// Asserts when AXI transactions is complete
output wire  TXN_DONE,
// AXI clock signal
input wire  M_AXI_ACLK,
// AXI active low reset signal
input wire  M_AXI_ARESETN,
// Master Interface Write Address Channel ports. Write address (issued by master)
output wire [C_M_AXI_ADDR_WIDTH-1 : 0] M_AXI_AWADDR,
// Write channel Protection type.
// This signal indicates the privilege and security level of the transaction,
// and whether the transaction is a data access or an instruction access.
output wire [2 : 0] M_AXI_AWPROT,
// Write address valid.
// This signal indicates that the master signaling valid write address and
↳control information.
output wire  M_AXI_AWVALID,
// Write address ready.
// This signal indicates that the slave is ready to accept an address and
↳associated control signals.
input wire  M_AXI_AWREADY,
// Master Interface Write Data Channel ports. Write data (issued by master)
output wire [C_M_AXI_DATA_WIDTH-1 : 0] M_AXI_WDATA,
// Write strobes.
// This signal indicates which byte lanes hold valid data.
// There is one write strobe bit for each eight bits of the write data bus.
output wire [C_M_AXI_DATA_WIDTH/8-1 : 0] M_AXI_WSTRB,
// Write valid. This signal indicates that valid write data and strobes are
↳available.
output wire  M_AXI_WVALID,
// Write ready. This signal indicates that the slave can accept the write data.
input wire  M_AXI_WREADY,
// Master Interface Write Response Channel ports.
// This signal indicates the status of the write transaction.
input wire [1 : 0] M_AXI_BRESP,
// Write response valid.
// This signal indicates that the channel is signaling a valid write response
input wire  M_AXI_BVALID,
// Response ready. This signal indicates that the master can accept a write
↳response.
output wire  M_AXI_BREADY,
// Master Interface Read Address Channel ports. Read address (issued by master)
output wire [C_M_AXI_ADDR_WIDTH-1 : 0] M_AXI_ARADDR,
// Protection type.
// This signal indicates the privilege and security level of the transaction,
// and whether the transaction is a data access or an instruction access.
output wire [2 : 0] M_AXI_ARPROT,
// Read address valid.
// This signal indicates that the channel is signaling valid read address and
↳control information.
output wire  M_AXI_ARVALID,

```

(continues on next page)

(continued from previous page)

```

    // Read address ready.
    // This signal indicates that the slave is ready to accept an address and
    ↪associated control signals.
    input wire M_AXI_ARREADY,
    // Master Interface Read Data Channel ports. Read data (issued by slave)
    input wire [C_M_AXI_DATA_WIDTH-1 : 0] M_AXI_RDATA,
    // Read response. This signal indicates the status of the read transfer.
    input wire [1 : 0] M_AXI_RRESP,
    // Read valid. This signal indicates that the channel is signaling the required
    ↪read data.
    input wire M_AXI_RVALID,
    // Read ready. This signal indicates that the master can accept the read data and
    ↪response information.
    output wire M_AXI_RREADY

);
:guilabel:`assign initial_count_value_out = M_AXI_RDATA[7:0];`

// function called clogb2 that returns an integer which has the
// value of the ceiling of the log base 2

// function called clogb2 that returns an integer which has the
// value of the ceiling of the log base 2

function integer clogb2 (input integer bit_depth);
begin
    for(clogb2=0; bit_depth>0; clogb2=clogb2+1)
        bit_depth = bit_depth >> 1;
    end
endfunction

// TRANS_NUM_BITS is the width of the index counter for
// number of write or read transaction.
localparam integer TRANS_NUM_BITS = clogb2(C_M_TRANSACTIONS_NUM-1);

// Example State machine to initialize counter, initialize write transactions,
// initialize read transactions and comparison of read data with the
// written data words.
parameter [1:0] IDLE = 2'b00, // This state initiates AXI4Lite transaction
    // after the state machine changes state to INIT_WRITE
    // when there is 0 to 1 transition on INIT_AXI_TXN
    INIT_WRITE = 2'b01, // This state initializes write transaction,
    // once writes are done, the state machine
    // changes state to INIT_READ
    INIT_READ = 2'b10, // This state initializes read transaction
    // once reads are done, the state machine
    // changes state to INIT_COMPARE
    INIT_COMPARE = 2'b11; // This state issues the status of comparison
    // of the written data with the read data

reg [1:0] mst_exec_state;

// AXI4LITE signals
//write address valid
reg axi_awvalid;

```

(continues on next page)

(continued from previous page)

```

//write data valid
reg      axi_wvalid;
//read address valid
reg      axi_arvalid;
//read data acceptance
reg      axi_rready;
//write response acceptance
reg      axi_bready;
//write address
reg [C_M_AXI_ADDR_WIDTH-1 : 0]    axi_awaddr;
//write data
reg [C_M_AXI_DATA_WIDTH-1 : 0]    axi_wdata;
//read addresss
reg [C_M_AXI_ADDR_WIDTH-1 : 0]    axi_araddr;
//Asserts when there is a write response error
wire     write_resp_error;
//Asserts when there is a read response error
wire     read_resp_error;
//A pulse to initiate a write transaction
reg      start_single_write;
//A pulse to initiate a read transaction
reg      start_single_read;
//Asserts when a single beat write transaction is issued and remains asserted_
↪till the completion of write trasaction.
reg      write_issued;
//Asserts when a single beat read transaction is issued and remains asserted till_
↪the completion of read trasaction.
reg      read_issued;
//flag that marks the completion of write trasactions. The number of write_
↪transaction is user selected by the parameter C_M_TRANSACTIONS_NUM.
reg      writes_done;
//flag that marks the completion of read trasactions. The number of read_
↪transaction is user selected by the parameter C_M_TRANSACTIONS_NUM
reg      reads_done;
//The error register is asserted when any of the write response error, read_
↪response error or the data mismatch flags are asserted.
reg      error_reg;
//index counter to track the number of write transaction issued
reg [TRANS_NUM_BITS : 0]    write_index;
//index counter to track the number of read transaction issued
reg [TRANS_NUM_BITS : 0]    read_index;
//Expected read data used to compare with the read data.
reg [C_M_AXI_DATA_WIDTH-1 : 0]    expected_rdata;
//Flag marks the completion of comparison of the read data with the expected read_
↪data
reg      compare_done;
//This flag is asserted when there is a mismatch of the read data with the_
↪expected read data.
reg      read_mismatch;
//Flag is asserted when the write index reaches the last write transction number
reg      last_write;
//Flag is asserted when the read index reaches the last read transction number
reg      last_read;
reg      init_txn_ff;
reg      init_txn_ff2;
reg      init_txn_edge;
wire     init_txn_pulse;

```

(continues on next page)

(continued from previous page)

```

//added registers for init_txn_read
:guilabel:`reg init_txn_ff_read;
reg init_txn_ff2_read;`

//set count out as count out i
:guilabel:`assign count_out=count_out_i;`

// I/O Connections assignments

//Adding the offset address to the base addr of the slave
assign M_AXI_AWADDR = C_M_TARGET_SLAVE_BASE_ADDR + axi_awaddr;
//AXI 4 write data
assign M_AXI_WDATA = axi_wdata;
assign M_AXI_AWPROT = 3'b000;
assign M_AXI_AWVALID = axi_awvalid;
//Write Data(W)
assign M_AXI_WVALID = axi_wvalid;
//Set all byte strobes in this example
assign M_AXI_WSTRB = 4'b1111;
//Write Response (B)
assign M_AXI_BREADY = axi_bready;
//Read Address (AR)
assign M_AXI_ARADDR = C_M_TARGET_SLAVE_BASE_ADDR + axi_araddr;
assign M_AXI_ARVALID = axi_arvalid;
assign M_AXI_ARPROT = 3'b001;
//Read and Read Response (R)
assign M_AXI_RREADY = axi_rready;
//Example design I/O
assign TXN_DONE = compare_done;
assign init_txn_pulse = (!init_txn_ff2) && init_txn_ff;

:guilabel:`assign init_txn_pulse_read = (!init_txn_ff2_read) && init_txn_ff_read;`

//Generate a pulse to initiate AXI transaction.
always @(posedge M_AXI_ACLK)
begin
    // Initiates AXI transaction delay
    if (M_AXI_ARESETN == 0 )
    begin
        init_txn_ff <= 1'b0;
        init_txn_ff2 <= 1'b0;
        :guilabel:`init_txn_ff_read <= 1'b0; //do the same thing for read txn
        init_txn_ff2_read<=1'b0;`
    end
    else
    begin
        init_txn_ff <= INIT_AXI_TXN;
        init_txn_ff2 <= init_txn_ff;
        :guilabel:`init_txn_ff_read <= init_axi_txn_read;
        init_txn_ff2_read <= init_txn_ff_read;`
    end
end
end

```

(continues on next page)

(continued from previous page)

```

//-----
//Write Address Channel
//-----

// The purpose of the write address channel is to request the address and
// command information for the entire transaction. It is a single beat
// of information.

// Note for this example the axi_awvalid/axi_wvalid are asserted at the same
// time, and then each is deasserted independent from each other.
// This is a lower-performance, but simpler control scheme.

// AXI VALID signals must be held active until accepted by the partner.

// A data transfer is accepted by the slave when a master has
// VALID data and the slave acknowledges it is also READY. While the master
// is allowed to generated multiple, back-to-back requests by not
// deasserting VALID, this design will add rest cycle for
// simplicity.

// Since only one outstanding transaction is issued by the user design,
// there will not be a collision between a new request and an accepted
// request on the same clock cycle.

always @(posedge M_AXI_ACLK)
begin
    //Only VALID signals must be deasserted during reset per AXI spec
    //Consider inverting then registering active-low reset for higher fmax
    if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
begin
    axi_awvalid <= 1'b0;
end
    //Signal a new address/data command is available by user logic
    else
begin
    if (start_single_write)
begin
    axi_awvalid <= 1'b1;
end
    //Address accepted by interconnect/slave (issue of M_AXI_AWREADY by slave)
    else if (M_AXI_AWREADY && axi_awvalid)
begin
    axi_awvalid <= 1'b0;
end
end
end
end

// start_single_write triggers a new write
// transaction. write_index is a counter to
// keep track with number of write transaction
// issued/initiated
always @(posedge M_AXI_ACLK)
begin
    if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
begin
        write_index <= 0;

```

(continues on next page)

(continued from previous page)

```

        end
        // Signals a new write address/ write data is
        // available by user logic
        else if (start_single_write)
        begin
            write_index <= write_index + 1;
        end
    end
end

//-----
//Write Data Channel
//-----

//The write data channel is for transferring the actual data.
//The data generation is specific to the example design, and
//so only the WVALID/WREADY handshake is shown here

always @(posedge M_AXI_ACLK)
begin
if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
begin
    axi_wvalid <= 1'b0;
    end
    //Signal a new address/data command is available by user logic
    else if (start_single_write)
    begin
        axi_wvalid <= 1'b1;
    end
    //Data accepted by interconnect/slave (issue of M_AXI_WREADY by slave)
    else if (M_AXI_WREADY && axi_wvalid)
    begin
        axi_wvalid <= 1'b0;
    end
    end
end

//-----
//Write Response (B) Channel
//-----

//The write response channel provides feedback that the write has committed
//to memory. BREADY will occur after both the data and the write address
//has arrived and been accepted by the slave, and can guarantee that no
//other accesses launched afterwards will be able to be reordered before it.

//The BRESP bit [1] is used indicate any errors from the interconnect or
//slave for the entire write burst. This example will capture the error.

//While not necessary per spec, it is advisable to reset READY signals in
//case of differing reset latencies between master/slave.

always @(posedge M_AXI_ACLK)
begin
if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
begin
    axi_bready <= 1'b0;

```

(continues on next page)

(continued from previous page)

```

        end
        // accept/acknowledge bresp with axi_bready by the master
        // when M_AXI_BVALID is asserted by slave
        else if (M_AXI_BVALID && ~axi_bready)
        begin
            axi_bready <= 1'b1;
        end
        // deassert after one clock cycle
        else if (axi_bready)
        begin
            axi_bready <= 1'b0;
        end
        // retain the previous value
        else
            axi_bready <= axi_bready;
        end

        //Flag write errors
        assign write_resp_error = (axi_bready & M_AXI_BVALID & M_AXI_BRESP[1]);

        //-----
        //Read Address Channel
        //-----

        //start_single_read triggers a new read transaction. read_index is a counter to
        //keep track with number of read transaction issued/initiated

        always @(posedge M_AXI_ACLK)
        begin
            if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
            begin
                read_index <= 0;
            end
            // Signals a new read address is
            // available by user logic
            else if (start_single_read)
            begin
                read_index <= read_index + 1;
            end
        end

        // A new axi_arvalid is asserted when there is a valid read address
        // available by the master. start_single_read triggers a new read
        // transaction
        always @(posedge M_AXI_ACLK)
        begin
            if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
            begin
                axi_arvalid <= 1'b0;
            end
            //Signal a new read address command is available by user logic
            else if (start_single_read)
            begin
                axi_arvalid <= 1'b1;
            end
            //Address accepted by interconnect/slave (issue of M_AXI_ARREADY by slave)

```

(continues on next page)

(continued from previous page)

```

        else if (M_AXI_ARREADY && axi_arvalid)
        begin
            axi_arvalid <= 1'b0;
        end
        // retain the previous value
    end

    //-----
    //Read Data (and Response) Channel
    //-----

    //The Read Data channel returns the results of the read request
    //The master will accept the read data by asserting axi_rready
    //when there is a valid read data available.
    //While not necessary per spec, it is advisable to reset READY signals in
    //case of differing reset latencies between master/slave.

    always @(posedge M_AXI_ACLK)
    begin
if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
begin
        axi_rready <= 1'b0;
    end
    // accept/acknowledge rdata/rresp with axi_rready by the master
    // when M_AXI_RVALID is asserted by slave
    else if (M_AXI_RVALID && ~axi_rready)
    begin
        axi_rready <= 1'b1;
    end
    // deassert after one clock cycle
    else if (axi_rready)
    begin
        axi_rready <= 1'b0;
    end
    // retain the previous value
    end

    //Flag write errors
    assign read_resp_error = (axi_rready & M_AXI_RVALID & M_AXI_RRESP[1]);

    //-----
    //User Logic
    //-----

    //Address/Data Stimulus

    //Address/data pairs for this example. The read and write values should
    //match.
    //Modify these as desired for different address patterns.

    //Write Addresses
    always @(posedge M_AXI_ACLK)
    begin
if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
begin

```

(continues on next page)



(continued from previous page)

```

        axi_awaddr <= 0;
    end
    // Signals a new write address/ write data is
    // available by user logic
    else if (M_AXI_AWREADY && axi_awvalid)
    begin
        :guilabel:`axi_awaddr <= axi_awaddr; //dont increment write address +
↪32'h00000004; `

    end
end

// Write data generation
always @(posedge M_AXI_ACLK)
begin
    if (:guilabel:`M_AXI_ARESETN == 0`)
    begin
        axi_wdata <= C_M_START_DATA_VALUE;
    end
    // Signals a new write address/ write data is
    // available by user logic
else if (:guilabel:`init_txn_pulse == 1'b1`) //ORIGINALLY WAS M_AXI_WREADY && axi_
↪wvalid
    begin
        :guilabel:`axi_wdata <= count_out_i`; //send count out intermediate
↪value
    end
end

//Read Addresses
always @(posedge M_AXI_ACLK)
begin
    if (:guilabel:`M_AXI_ARESETN == 0`) //|| init_txn_pulse == 1'b1) //
↪put one clk cycle ahead
    begin
        :guilabel:`axi_araddr <= 32'h0000_0000;` //always reading from
↪address 200
    end
    // Signals a new write address/ write data is
    // available by user logic
    else if (:guilabel:`init_txn_pulse_read==1'b1`) //originally was: (M_AXI_
↪ARREADY && axi_arvalid)
    begin
        :guilabel:`axi_araddr <= axi_araddr;` //do not increment the read
↪address + 32'h00000004;
    end
end

always @(posedge M_AXI_ACLK)
begin
    if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
    begin
        expected_rdata <= C_M_START_DATA_VALUE;
    end
    // Signals a new write address/ write data is

```

(continues on next page)

(continued from previous page)

```

        // available by user logic
        else if (M_AXI_RVALID && axi_rready)
        begin
            expected_rdata <= C_M_START_DATA_VALUE + read_index;
        end
    end
//implement master command interface state machine
always @ ( posedge M_AXI_ACLK)
begin
    if (M_AXI_ARESETN == 1'b0)
    begin
        // reset condition
        // All the signals are assigned default values under reset condition
        mst_exec_state <= IDLE;
        start_single_write <= 1'b0;
        write_issued <= 1'b0;
        start_single_read <= 1'b0;
        read_issued <= 1'b0;
        compare_done <= 1'b0;
        ERROR <= 1'b0;
    end
    else
    begin
        // state transition
        case (mst_exec_state)
IDLE:
            // This state is responsible to initiate
            // AXI transaction when init_txn_pulse is asserted
            :guilabel:` if ( init_txn_pulse == 1'b1 )
            begin
                mst_exec_state <= INIT_WRITE;
                ERROR <= 1'b0;
                compare_done <= 1'b0;`
            end
            :guilabel:`else if (init_txn_pulse_read ==1'b1 )
            begin
                mst_exec_state <=INIT_READ;
            end `
            else
            begin
                mst_exec_state <= IDLE;
            end
INIT_WRITE:
            // This state is responsible to issue start_single_write pulse to
            // initiate a write transaction. Write transactions will be
            // issued until last_write signal is asserted.
            // write controller
            if (writes_done)
            begin
                mst_exec_state <= :guilabel:`IDLE;`//
            end
            else
            begin
                mst_exec_state <= INIT_WRITE;
            end
        end case
    end
end

```

(continues on next page)

(continued from previous page)

```

if (~axi_awvalid && ~axi_wvalid && ~M_AXI_BVALID && ~last_write && ~start_single_
→write && ~write_issued)
    begin
        start_single_write <= 1'b1;
        write_issued <= 1'b1;
    end
else if (axi_bready)
    begin
        write_issued <= 1'b0;
    end
else
    begin
        start_single_write <= 1'b0; //Negate to generate a pulse
    end
end

INIT_READ:
    // This state is responsible to issue start_single_read pulse to
    // initiate a read transaction. Read transactions will be
    // issued until last_read signal is asserted.
    // read controller
    if (reads_done)
    begin
        mst_exec_state <= :guilabel:`IDLE`;
    end
    else
    begin
        mst_exec_state <= INIT_READ;

        if (~axi_arvalid && ~M_AXI_RVALID && ~last_read && ~start_single_
→read && ~read_issued)
            begin
                start_single_read <= 1'b1;
                read_issued <= 1'b1;
            end
            else if (axi_rready)
            begin
                read_issued <= 1'b0;
            end
            else
            begin
                start_single_read <= 1'b0; //Negate to generate a pulse
            end
        end
    end

INIT_COMPARE:
    begin
        // This state is responsible to issue the state of comparison
        // of written data with the read data. If no error flags are set,
        // compare_done signal will be asseted to indicate success.
        ERROR <= error_reg;
        mst_exec_state <= IDLE;
        compare_done <= 1'b1;
    end
default :
    begin

```

(continues on next page)

(continued from previous page)

```

                mst_exec_state <= IDLE;
            end
        endcase
    end
end //MASTER_EXECUTION_PROC

//Terminal write count

always @(posedge M_AXI_ACLK)
begin
    if (:guilabel:`M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1`)
        last_write <= 1'b0;

        //The last write should be associated with a write address ready response
    else if ((write_index == C_M_TRANSACTIONS_NUM) && M_AXI_AWREADY)
        last_write <= 1'b1;
    else
        last_write <= last_write;
end

//Check for last write completion.

//This logic is to qualify the last write count with the final write
//response. This demonstrates how to confirm that a write has been
//committed.

always @(posedge M_AXI_ACLK)
begin
    if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
        writes_done <= 1'b0;

        //The writes_done should be associated with a bready response
    else if (last_write && M_AXI_BVALID && axi_bready)
        writes_done <= 1'b1;
    else
        writes_done <= writes_done;
end

//-----
//Read example
//-----

//Terminal Read Count

always @(posedge M_AXI_ACLK)
begin
    if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
        last_read <= 1'b0;

        //The last read should be associated with a read address ready response
    else if ((read_index == C_M_TRANSACTIONS_NUM) && (M_AXI_ARREADY) )
        last_read <= 1'b1;
    else
        last_read <= last_read;
end
end

```

(continues on next page)

(continued from previous page)

```

/*
Check for last read completion.
This logic is to qualify the last read count with the final read
response/data.
*/
always @(posedge M_AXI_ACLK)
begin
    if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
        reads_done <= 1'b0;

    //The reads_done should be associated with a read ready response
    else if (last_read && M_AXI_RVALID && axi_rready)
        reads_done <= 1'b1;
    else
        reads_done <= reads_done;
    end

//-----
//Example design error register
//-----

//Data Comparison
always @(posedge M_AXI_ACLK)
begin
    if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
        read_mismatch <= 1'b0;

    //The read data when available (on axi_rready) is compared with the expected_
↪data
    else if ((M_AXI_RVALID && axi_rready) && (M_AXI_RDATA != expected_rdata))
        read_mismatch <= 1'b1;
    else
        read_mismatch <= read_mismatch;
    end

    // Register and hold any data mismatches, or read/write interface errors
    always @(posedge M_AXI_ACLK)
    begin
        if (M_AXI_ARESETN == 0 || :guilabel:`init_txn_pulse == 1'b1`)
            error_reg <= 1'b0;

        //Capture any error types
        else if (read_mismatch || write_resp_error || read_resp_error)
            error_reg <= 1'b1;
        else
            error_reg <= error_reg;
    end
    // Add user logic here

    // User logic ends

endmodule

```

## 13.3 Creating the Master DUT Simulation Environment

1. Package the custom IP and import it into the project. This was previously explained with the simple counter, but for a refresher refer to adding a custom IP to a design.
2. Create a block diagram with an AXI VIP, two AXI Smart Connects, AXI BRAM Controller, and Clock Memory Generator connected as shown.

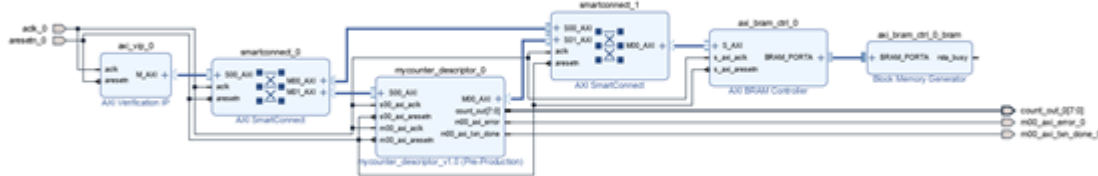


Fig. 2: Block Diagram Setup

3. Navigate to the address editor and assign addresses to the custom DUT and the BRAM. In this example we assigned the BRAM to address 0 and the DUT to 0x4000\_0000.

Diagram x Address Editor x Address Map x descriptor_tb.sv x						
<input checked="" type="checkbox"/> Assigned (3) <input checked="" type="checkbox"/> Unassigned (0) <input checked="" type="checkbox"/> Excluded (0) <input type="button" value="Hide All"/>						
Name	Interface	Slave Segment	Master Base Address	Range	Master High Address	
Network 0						
/axi_vip_0						
/axi_vip_0Master_AXI (32 address bits : 4G)						
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_0000	8K	0x0000_1FFF	
/mycounter_descriptor_0/S00_AXI	S00_AXI	S00_AXI_reg	0x4000_0000	64K	0x4000_FFFF	
/mycounter_descriptor_0						
/mycounter_descriptor_0/M00_AXI (32 address bits : 4G)						
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_0000	8K	0x0000_1FFF	

Fig. 3: Address Editor

4. Go back to the block diagram and right-click on a blank spot in the design. Select *Validate Design*.
5. The next step is to create a wrapper file which turns the block diagram into HDL. To do this go to the *Sources* and right-click on the source for your block diagram (the default name is `design_1` or something similar). Select *Create HDL Wrapper* and then *Let Vivado manage wrapper and auto-update*.
6. The next step is to create a testbench to ensure the custom AXI IP works as intended.

## 13.4 Testbench for a Master Custom DUT

The testbench for this advanced master counter DUT is similar to the testbench of the simpler DUT we previously created and follows all of the core concepts. The difference is that this advanced master DUT reads in the start value and counting mode from memory. It is important to remember this so you can first place these values directly into memory, and then send the DUT the command to read these values in before enabling the counter. Another thing important to keep straight are the addresses for writing to the memory directly (0000\_0000 in this example with an offset of 200), and the address for writing to the DUT directly (4000\_0000 in this example).

Follow the steps stated for creating a testbench for a simple counter. Make the appropriate address changes and update the logic to test all aspects of the advanced descriptor DUT.

A brief description of my testbench logic is stated below, the parentheses include the address that the command is sent to:

- Write the start value and counting mode directly into memory (addr:0000\_0204)
- Initiate the counter to read the start value into the DUT (addr: 4000\_0008)
- Enable the counter (addr:4000\_0000)
- After a delay, initiate the DUT to send the current count out value to memory(4000\_0008)
- Disable counter (4000\_0000)
- Read count value that was sent previously directly from memory (0000\_0200)
- Write a new start value into memory, this time decrement mode (0000\_0204)
- Initiate the counter to read in the start value into the DUT (4000\_0008)
- Enable the counter
- After a delay, disable the counter

---

**Important:** If you want to download the testbench file directly, go [here](#).

---

## 13.5 Simulating the Master Custom DUT

This section is based on the Interpreting Simulation Waveforms For a Custom DUT earlier section. Please refer to that documentation for details.

1. Run the Behavioral Simulation
2. The waveform should have automatically opened. In the left column, there are some signals we want to add to the waveform. The first signal is `axi_vip_0`, this will show the reads and writes that we initiate from the `axi_vip` in our testbench. In order to add a signal to the waveform, right click on the desired signal and choose *Add to waveform*. The next group of signals necessary to add to the waveform are for our custom DUT, in this example labeled `mycounter_descriptor`. This will show the writes written to the counter from the AXI VIP, as well as the commands the DUT performs to memory. And the last group of signals to add to the waveform is `axi_bram_ctrl_0`. This will allow you to see the data stored in memory.
3. Now that we have added the necessary waveforms, in order to see the simulation run through our testbench properly we need to simulate for 3ms. To do this, make sure that the top toolbar is set to at least 3ms and then click the button highlighted in the photo below.

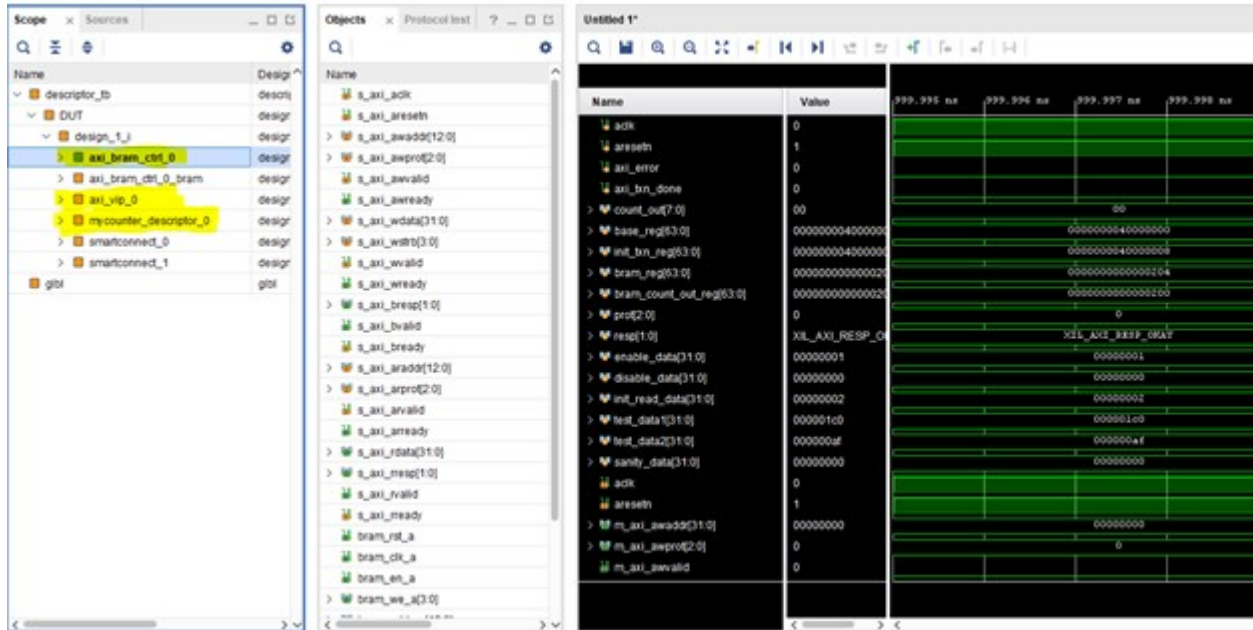


Fig. 4: Add Desired Signals to Waveform

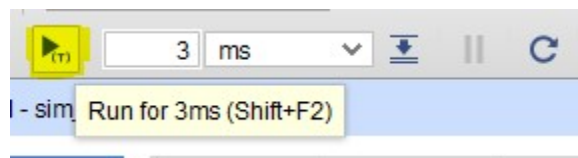


Fig. 5: 3ms Simulation time



## BUILDING A BASIC SIMULATION ENVIRONMENT (VC707)

### 14.1 Generating PCIe and MIG Example Designs

Now that we have experience generating and manipulating the PCIe and MIG example designs, we can start putting the pieces together - that is, building the basic infrastructure behind our FPGA emulation environment. The infrastructure will begin modifying Xilinx's PCIe example design, as this will allow us to perform reads and writes to both DDR memory and a replaceable Device Under Test (DUT), as well as other on-board peripherals. This can be accomplished through the use of an AXI SmartConnect, or what is known as a "NoC" in industry. You can read more about the SmartConnect IP and the AXI protocol [here](#). We will give the DDR memory and the Device Under Test different offset addresses in the AXI memory space, and then we can decide which device the PCIe will read or write to by specifying the address of the transaction.

---

**Note:** For a refresher on generating the MIG example design or targeting the VC707 board, please see this [MIG overview](#).

---

First, we will want to create a new Vivado project and select your preferred FPGA or board. For this article, we will be using the Xilinx VC707 board as our target. Then, open up a new block diagram. Under the *Board* tab, select the *DDR3 SDRAM* option.

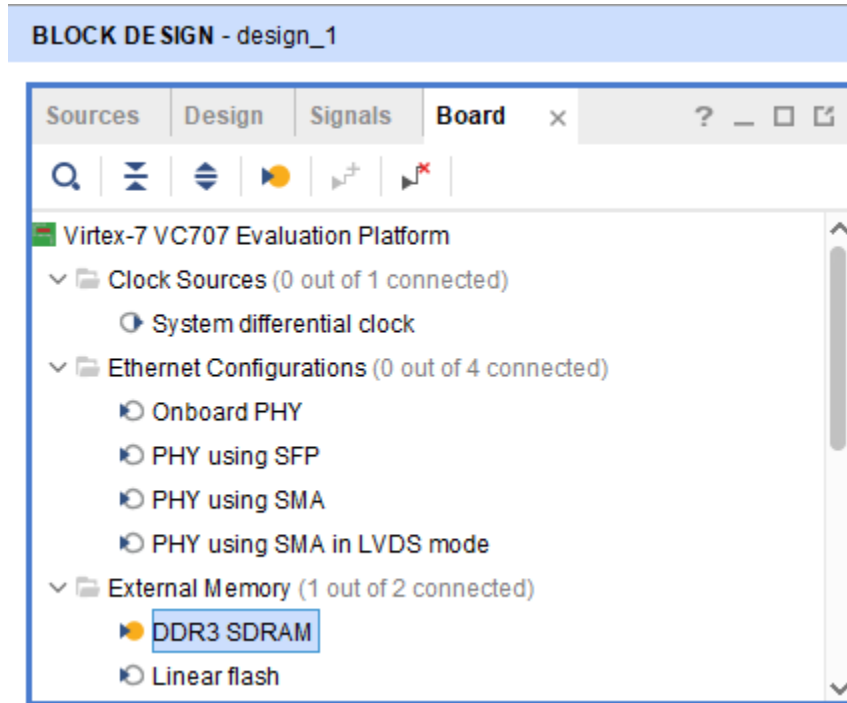
This will insert a MIG into the block diagram, which we can edit by double clicking on the IP. If you are not using a board, generate a MIG 7 Series or equivalent IP using Xilinx's IP integrator. For the MIG 7 Series, modify the following fields:

---

**Important:** Unless mentioned otherwise, leave all values default.

---

- Desired Clock Period → 2500ps (400MHz)
- Data Width → 64 bit (default)
- AXI Data Width → 64 bit
- Input Clock Period → 5000ps (200MHz)
- Deselect any Additional Clocks
- Addressing → Bank/Row/Column
- System Clock → Differential
- Reference Clock → Use System Clock
- Reset → ACTIVE LOW
- Uncheck the Box for DCI Cascade



- Select Fixed Pinout, then select Validate for the given pinout
- In the System Signals section:
  - Leave `sys_clk_p` and `sys_clk_n` to their default pins
  - Assign `sys_rst` to *AR40* (push button)
  - Assign `init_calib_complete` to *AM39* (LED)

Of course, the pinout will differ depending on the board or FPGA chosen. For more information on the VC707 board pinout, see this documentation from Xilinx here: [UG885](#).

Once these modifications have been made, the MIG IP will regenerate. Then, generate the IP example design by right-clicking on the IP block and selecting *Generate IP Example Design*. As before, this will open up a project in Vivado with the MIG IP example design, which we can set aside for the moment.

Now, we will also need to generate the IP example design for the AXI Memory Mapped to PCI Express core.

---

**Note:** For a refresher on generating the AXI Memory Mapped to PCI Express example design, please see this [PCIe overview](#).

---

Click on the + icon to add IP to the block design, then select *AXI Memory Mapped to PCI Express*. Make the following changes to the core:

---

**Important:** Unless specified, please leave everything as default.

---

- Reference Clock Frequency → 100MHz
- Check the box to enable External PIPE Interface (this helps to speed up the simulation time)
- Lane Width → X8

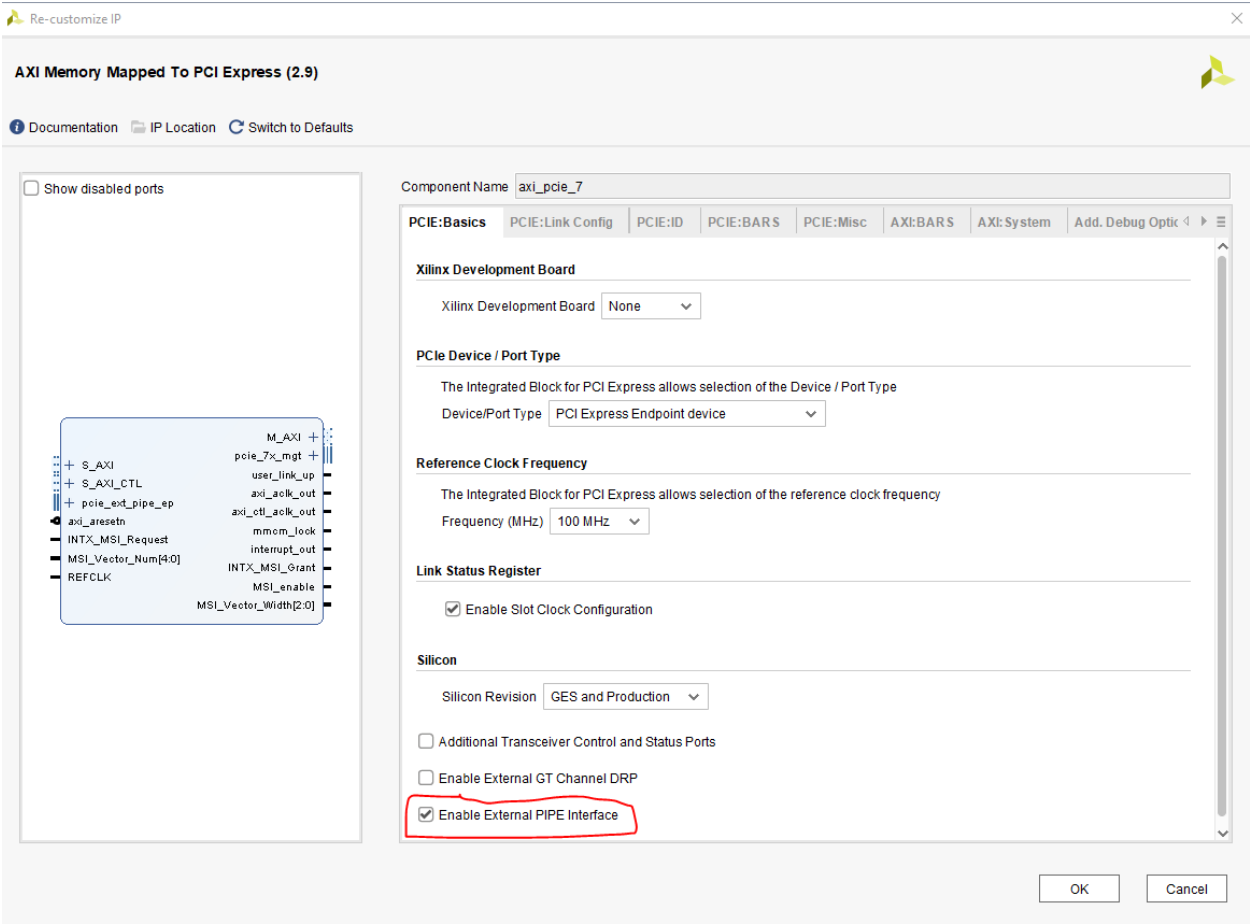


Fig. 1: PCIE:Basics Customization

- Link Speed → 2.5GT/s
- In the PCIe BARs section, ensure only 1 BAR is enabled and that it is 16KB in size with offset at address 0x00000000.

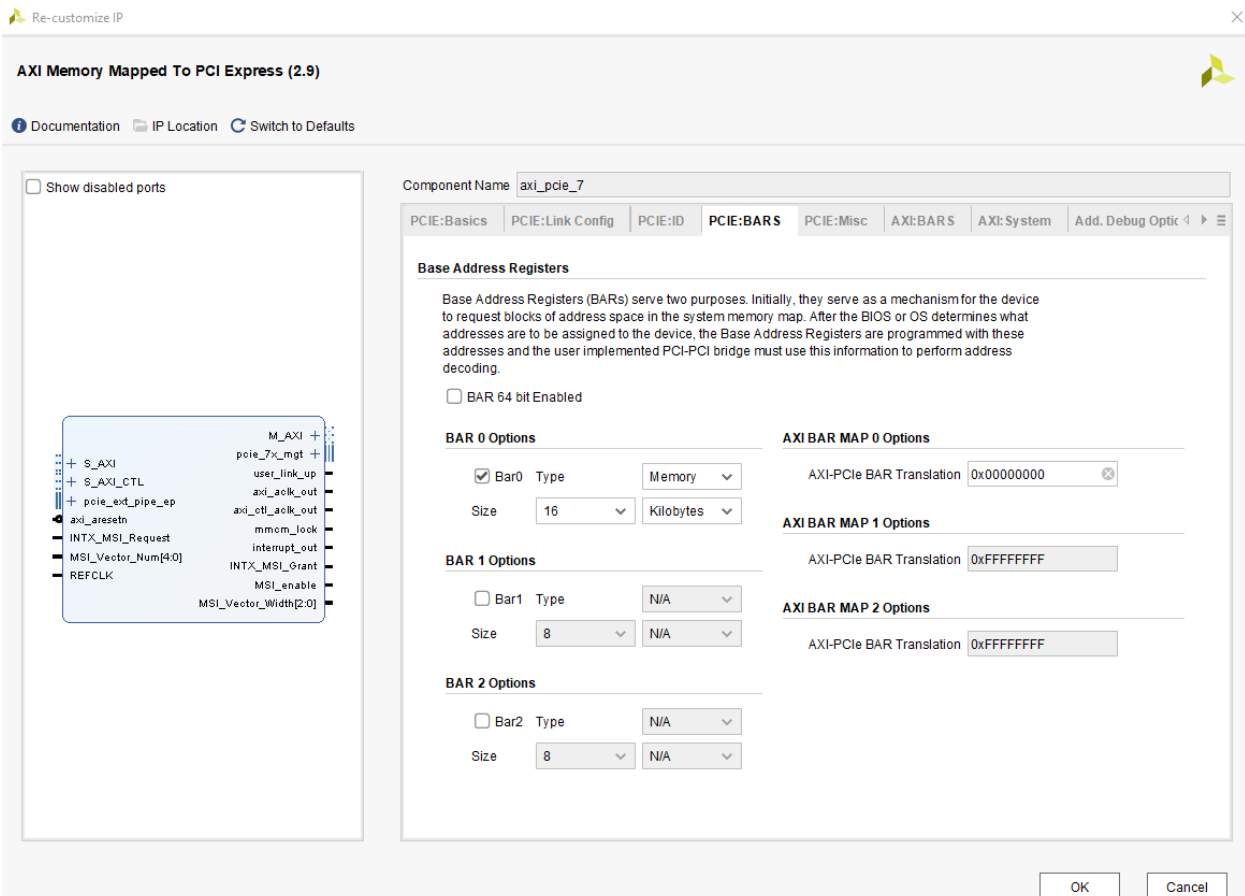


Fig. 2: PCIe:BARS Customization

Once this core has been generated, generate an example design for this IP as well. Now that the example designs have been generated for both the MIG and the PCIe IPs, we are ready to move onto the next section.

## 14.2 Creating the Block Diagram

Like we did in the [section 2.4](#) of the AXI MM to PCIe IP Overview, the first step that we will do is comment out the BRAM instantiation from the top file of the PCIe example design (`xilinx_axi_pcie_ep.v`). However, instead of inserting a MIG into its place, we are instead going to create a new block diagram. In the end, this is what we want the block diagram to look like:

In order to create this block diagram, follow these instructions:

1. Add an AXI Smartconnect IP to the block design with two AXI Master outputs and one AXI Slave input. Make sure that the data width is set to at least 32 bits, and make sure that there are two clock inputs.
2. Make the S00\_AXI, aclk, and aresetn ports external, as these will connect back into our PCIe core.

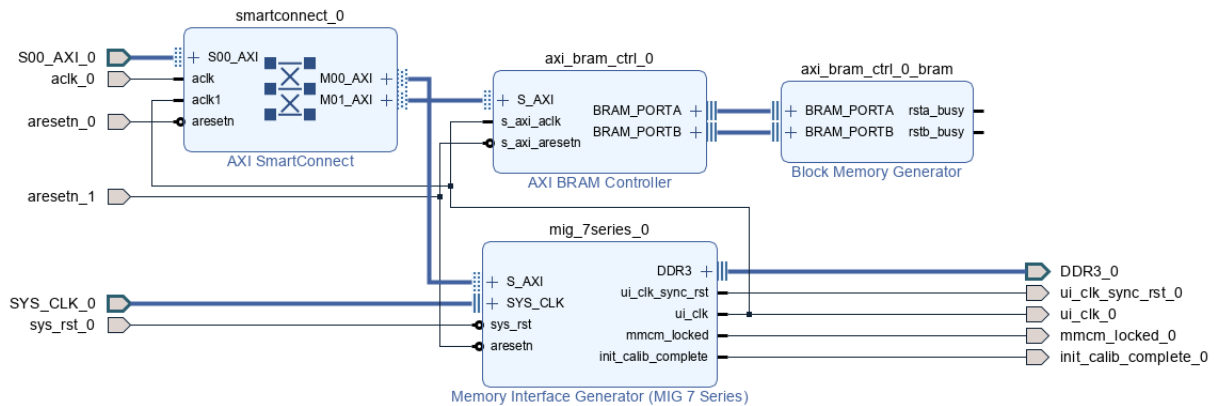


Fig. 3: Combined block diagram

3. Add a MIG 7 Series IP to the block design from the **Board** tab, and make sure to customize it in the EXACT SAME way as the MIG you customized in the previous section. This will ensure that the example design we generated will have the correct parameters associated with it.
4. Make the `SYS_CLK`, `sys_rst`, `aresetn`, `DDR3`, `ui_clk_sync_rst`, `ui_clk`, `mmcm_locked`, and `init_calib_complete` pins external, as these will be handled by our MIG example design. The `SYS_CLK` and `DDR3` pins should already be external, but to keep the same naming convention, delete the previous external connections, and then right-click to make them external again.
5. Add an AXI BRAM controller IP to the block design, and make sure to set the interface type to AXILite and Data Width to 32 bits. This BRAM represents the replaceable DUT that we should be able to exchange with a custom design later.
6. Connect the `M00_AXI` port from the Smartconnect to the `S_AXI` port on the MIG, and connect the `M01_AXI` port from the Smartconnect to the `S_AXI` port on the BRAM controller.
7. Connect the `ui_clk` from the MIG to the `ack1` port on the Smartconnect and the `s_axi_aclk` port on the BRAM controller. This way, the example DUT will be in the same clock domain as the MIG.
8. Connect the `s_axi_aresetn` port on the BRAM controller to the external `aresetn` signal going into the MIG. This way, the example DUT reset will be synchronous with the MIG reset.
9. Finally, there should be an option at the top of the screen to *Run Connection Automation*, and doing this should insert the Block Memory Generator, which will be attached to the BRAM controller.

Now that the block diagram has been created, we will need to use the address editor to assign the MIG and BRAM locations in the AXI memory space. Click on the *Address Editor* tab, and edit the offset addresses as follows:

- MIG: size 8KB, range: 0x0000\_0000 to 0x0000\_1FFF
- BRAM: size 8KB, range: 0x2000\_3FFF

If we click on the *Address Map* tab, then we can even see a layout of the memory mapping:

Since we configured the PCIe to have a 16KB BAR from address 0x0000\_0000 to 0x0000\_3FFF, we should now be able to access both of our AXI slaves from within the PCIe memory space.

Finally, we can go ahead and right-click on our block diagram and select *validate design*. There might be a warning that the resets are not synchronous - this is because we have not connected the PCIe IP to the design yet, so we can ignore this for now. Once Validation is successful, we will need to right-click on the block design under the *Sources*

Diagram x board.v x sample\_tests.vh x Address Editor x Address Map x

Assigned (2)

Unassigned (0)

Excluded (0)

Hide All

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
External Masters (1)					
/S00_AXI_0 (32 address bits : 4G)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_2000	8K	0x0000_3FFF
/mig_7series_0/memmap	S_AXI	memaddr	0x0000_0000	8K	0x0000_1FFF

Fig. 4: Address Editor for MIG and BRAM

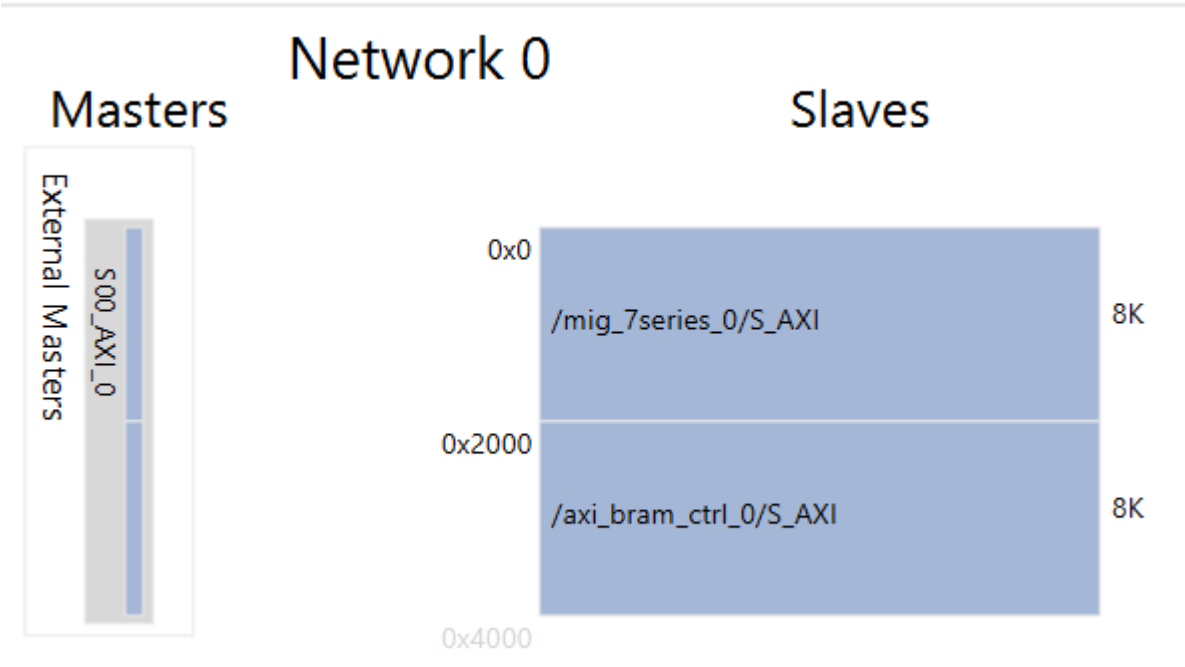


Fig. 5: Address Map for MIG and BRAM

menu, and select *Create HDL Wrapper*. Just like before, this will generate an RTL wrapper file for this block diagram, which we can instantiate into our PCIe example design in the next section.

## 14.3 Connecting it All Together

Similar to [section 2.4](#), we will now need to instantiate our block diagram into the PCIe example design. Since this process has several steps involved with it, we will include the design, constraints, and simulation top file here. This next section will be a brief overview of the steps needed to combine the PCIe example design, the MIG example design, and the block diagram. This has already been done for you in this case (just download the files), but it is highly recommended that you follow along and try to understand what modifications were made in each step.

---

**Important:** You can download our design top file [here](#).

---

---

**Important:** You can download our constraints file [here](#).

---

---

**Important:** You can download our simulation top file [here](#).

---

First, we will need to correctly instantiate the block design wrapper file into the PCIe example top file. In order to do this, we can locate where we commented out the old BRAM instantiation, and instead instantiate the block design.

Then, we will need to copy all of the relevant parameters, wires, functions, inputs, and outputs from the MIG example design top file into the PCIe example design top file. For more a deeper explanation on this, see [section 2.4](#) on the AXI MM to PCIe IP Overview tab.

---

**Note:** The following fields had to be changed because of already existing fields in the PCIe example design.

---

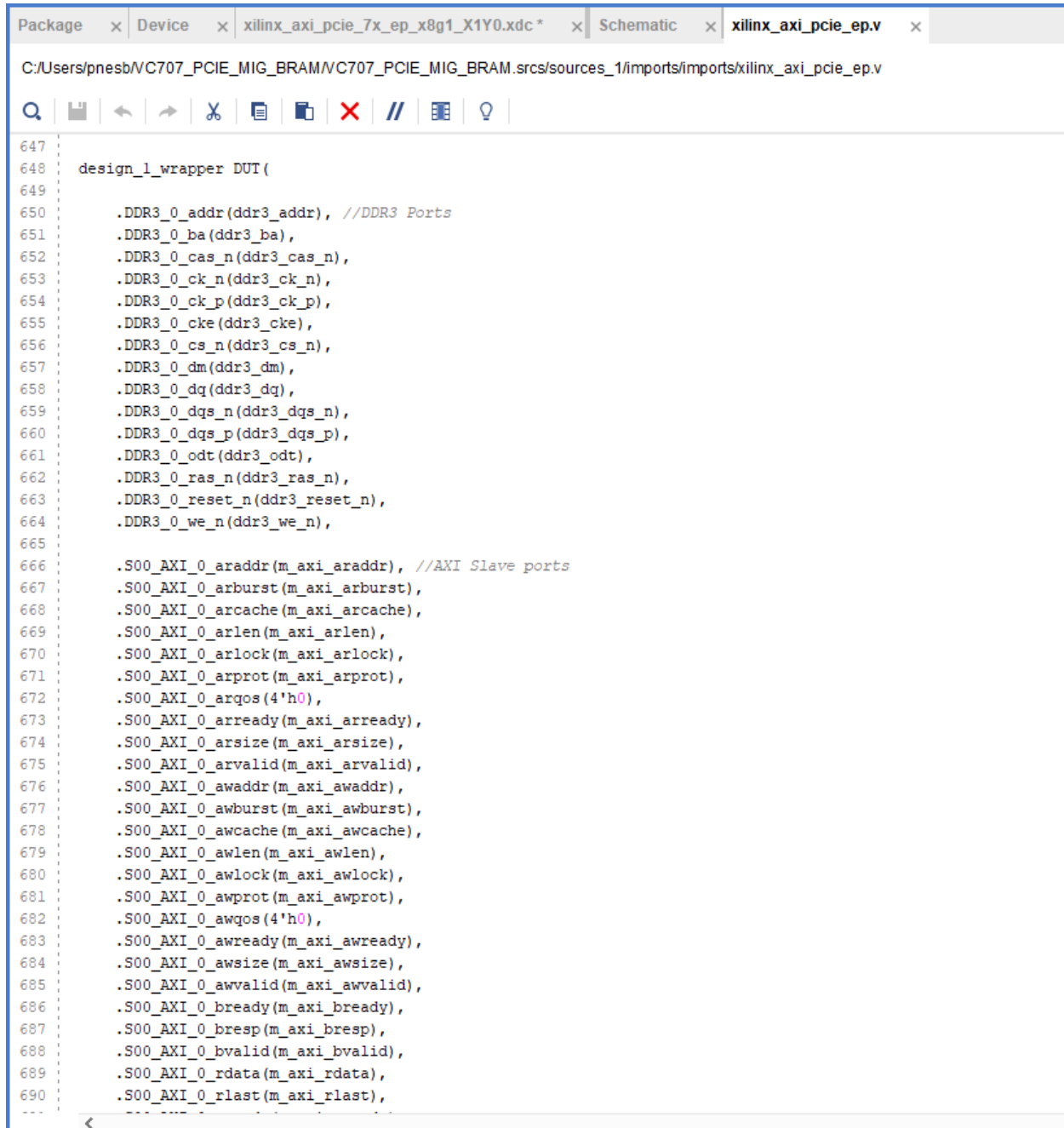
- Parameters: TCQ → TCQ\_MIG
- Inputs: sys\_clk\_n → sys\_clk\_n\_mig
- Outputs: sys\_clk\_p → sys\_clk\_p\_mig

Make sure to copy over the statement that synchronizes the MIG reset:

Then, we will need to copy over the top-level constraints from the MIG example design and paste them into the top-level constraints file for the PCIe example design. The top level constraints for each project can be found under the *Constraints* tab in the *Sources* menu.

Once the top file and the constraints file have been modified, then we can run synthesis and implementation to ensure that there are no errors in our design. Refer to the TCL console and the Xilinx forums for help with debugging, as every board/FPGA has different parameters, or cross reference your design and constraints top file with the provided example files above.

Once synthesis and implementation are complete, your schematic should look something like this. Once synthesis and implementation are complete, we can now move on to the next section.



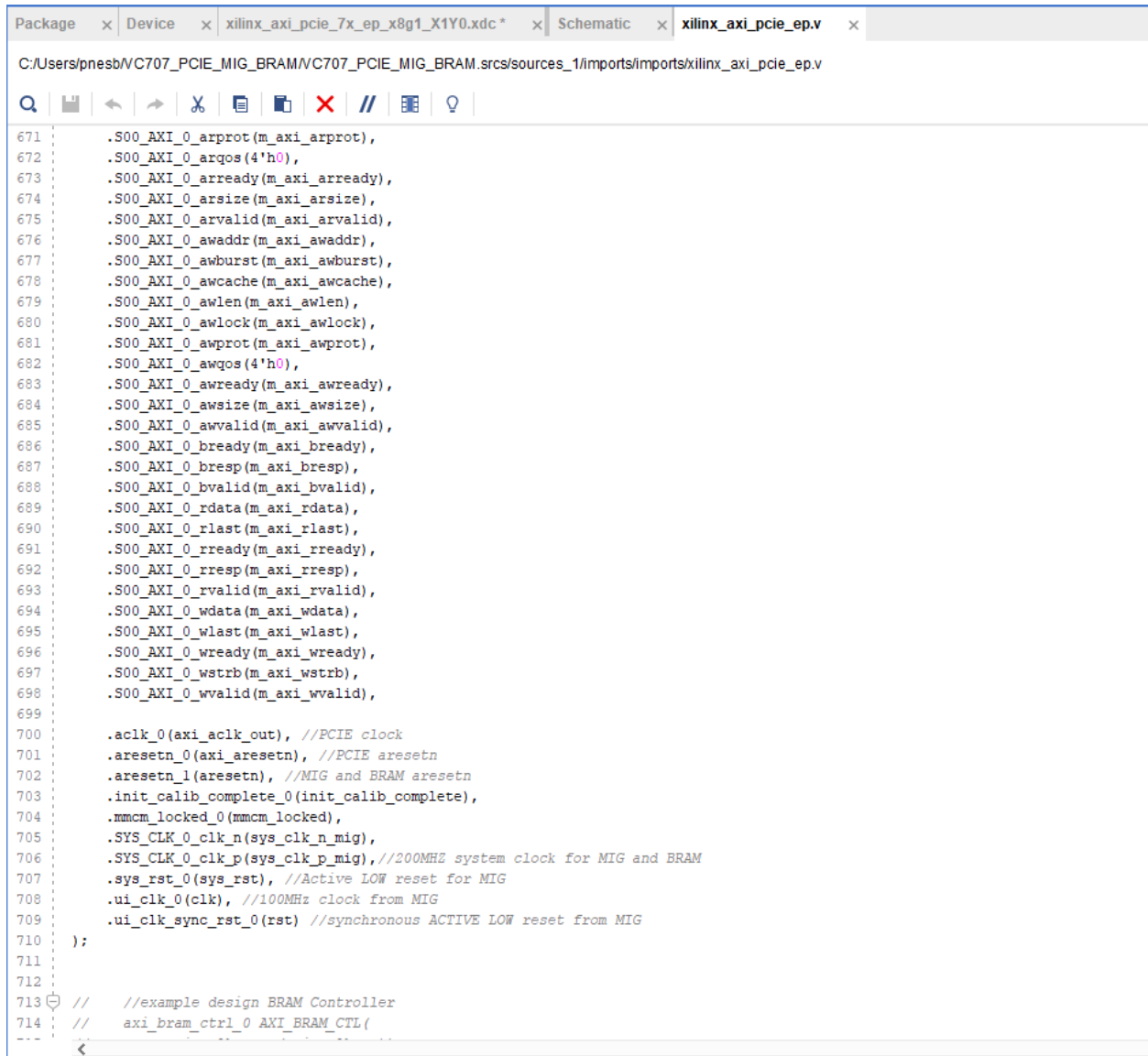
```

647
648 design_1_wrapper DUT(
649
650     .DDR3_0_addr(DDR3_addr), //DDR3 Ports
651     .DDR3_0_ba(DDR3_ba),
652     .DDR3_0_cas_n(DDR3_cas_n),
653     .DDR3_0_ck_n(DDR3_ck_n),
654     .DDR3_0_ck_p(DDR3_ck_p),
655     .DDR3_0_cke(DDR3_cke),
656     .DDR3_0_cs_n(DDR3_cs_n),
657     .DDR3_0_dm(DDR3_dm),
658     .DDR3_0_dq(DDR3_dq),
659     .DDR3_0_dqs_n(DDR3_dqs_n),
660     .DDR3_0_dqs_p(DDR3_dqs_p),
661     .DDR3_0_odt(DDR3_odt),
662     .DDR3_0_ras_n(DDR3_ras_n),
663     .DDR3_0_reset_n(DDR3_reset_n),
664     .DDR3_0_we_n(DDR3_we_n),
665
666     .S00_AXI_0_araddr(m_axi_araddr), //AXI Slave ports
667     .S00_AXI_0_arburst(m_axi_arburst),
668     .S00_AXI_0_arcache(m_axi_arcache),
669     .S00_AXI_0_arlen(m_axi_arlen),
670     .S00_AXI_0_arlock(m_axi_arlock),
671     .S00_AXI_0_arprot(m_axi_arprot),
672     .S00_AXI_0_arqos(4'h0),
673     .S00_AXI_0_arready(m_axi_arready),
674     .S00_AXI_0_arsize(m_axi_arsize),
675     .S00_AXI_0_arvalid(m_axi_arvalid),
676     .S00_AXI_0_awaddr(m_axi_awaddr),
677     .S00_AXI_0_awburst(m_axi_awburst),
678     .S00_AXI_0_awcache(m_axi_awcache),
679     .S00_AXI_0_awlen(m_axi_awlen),
680     .S00_AXI_0_awlock(m_axi_awlock),
681     .S00_AXI_0_awprot(m_axi_awprot),
682     .S00_AXI_0_awqos(4'h0),
683     .S00_AXI_0_awready(m_axi_awready),
684     .S00_AXI_0_awsz(m_axi_awsz),
685     .S00_AXI_0_awvalid(m_axi_awvalid),
686     .S00_AXI_0_bready(m_axi_bready),
687     .S00_AXI_0_bresp(m_axi_bresp),
688     .S00_AXI_0_bvalid(m_axi_bvalid),
689     .S00_AXI_0_rdata(m_axi_rdata),
690     .S00_AXI_0_rlast(m_axi_rlast),
691

```

Fig. 6: Instantiating the Block Diagram (1)





```

Package x Device x xilinx_axi_pcie_7x_ep_x8g1_X1Y0.xdc x Schematic x xilinx_axi_pcie_ep.v x
C:/Users/pnesb/VC707/PCIE_MIG_BRAM/VC707/PCIE_MIG_BRAM/srcs/sources_1/imports/imports/xilinx_axi_pcie_ep.v

671 .S00_AXI_0_arprot(m_axi_arprot),
672 .S00_AXI_0_arqos(4'h0),
673 .S00_AXI_0_arready(m_axi_arready),
674 .S00_AXI_0_arsize(m_axi_arsize),
675 .S00_AXI_0_arvalid(m_axi_arvalid),
676 .S00_AXI_0_awaddr(m_axi_awaddr),
677 .S00_AXI_0_awburst(m_axi_awburst),
678 .S00_AXI_0_awcache(m_axi_awcache),
679 .S00_AXI_0_awlen(m_axi_awlen),
680 .S00_AXI_0_awlock(m_axi_awlock),
681 .S00_AXI_0_awprot(m_axi_awprot),
682 .S00_AXI_0_awqos(4'h0),
683 .S00_AXI_0_awready(m_axi_awready),
684 .S00_AXI_0_awsz(m_axi_awsz),
685 .S00_AXI_0_awvalid(m_axi_awvalid),
686 .S00_AXI_0_bready(m_axi_bready),
687 .S00_AXI_0_bresp(m_axi_bresp),
688 .S00_AXI_0_bvalid(m_axi_bvalid),
689 .S00_AXI_0_rdata(m_axi_rdata),
690 .S00_AXI_0_rlast(m_axi_rlast),
691 .S00_AXI_0_rready(m_axi_rready),
692 .S00_AXI_0_rresp(m_axi_rresp),
693 .S00_AXI_0_rvalid(m_axi_rvalid),
694 .S00_AXI_0_wdata(m_axi_wdata),
695 .S00_AXI_0_wlast(m_axi_wlast),
696 .S00_AXI_0_wready(m_axi_wready),
697 .S00_AXI_0_wstrb(m_axi_wstrb),
698 .S00_AXI_0_wvalid(m_axi_wvalid),
699
700 .aclk_0(axi_aclk_out), //PCIE clock
701 .aresetn_0(axi_aresetn), //PCIE aresetn
702 .aresetn_1(aresetn), //MIG and BRAM aresetn
703 .init_calib_complete_0(init_calib_complete),
704 .mmcm_locked_0(mmcm_locked),
705 .SYS_CLK_0_clk_n(sys_clk_n_mig),
706 .SYS_CLK_0_clk_p(sys_clk_p_mig), //200MHZ system clock for MIG and BRAM
707 .sys_rst_0(sys_rst), //Active LOW reset for MIG
708 .ui_clk_0(clk), //100MHz clock from MIG
709 .ui_clk_sync_rst_0(rst) //synchronous ACTIVE LOW reset from MIG
710 );
711
712
713 // //example design BRAM Controller
714 // axi_bram_ctrl_0 AXI_BRAM_CTL(

```

Fig. 7: Instantiating the Block Diagram (2)

```

always @(posedge clk) begin
    aresetn <= ~rst;
end

```

Fig. 8: Copy over the MIG Reset Statement

```

Package x Device x xilinx_axi_pcie_7x_ep_x8g1_X1Y0.xdc * x Schematic x
C:/Users/pnesb/VC707_PCIE_MIG_BRAM/VC707_PCIE_MIG_BRAM.srsc/constrs_1/imports/imports/xilinx_axi_pcie_7x_ep_x8g1_X1Y0.xdc

102 # To use these pins an IBUFDS primitive (refclk_ibuf) is
103 # instantiated in user's design.
104 # Please refer to the Virtex-7 GT Transceiver User Guide
105 # (UG) for guidelines regarding clock resource selection.
106 #
107
108 set_property LOC IBUFDS_GTE2_X1Y5 [get_cells refclk_ibuf]
109
110 set_false_path -from [get_ports sys_rst_n]
111
112 #####
113 # Timing Constraints
114 #####
115 #
116 create_clock -name sys_clk -period 10 [get_ports sys_clk_p]
117 #
118 #
119 #####
120 # Physical Constraints
121
122 # INSERT CONSTRAINTS FROM MIG EXAMPLE DESIGN
123 # PadFunction: IO_L6P_T0_15
124 set_property VCCAUX_IO DONTCARE [get_ports {init_calib_complete}]
125 set_property IOSTANDARD LVCMOS18 [get_ports {init_calib_complete}]
126 set_property PACKAGE_PIN AM39 [get_ports {init_calib_complete}]
127
128
129 set_property INTERNAL_VREF 0.900 [get_iobanks 15]
130 set_property INTERNAL_VREF 0.750 [get_iobanks 37]
131 set_property INTERNAL_VREF 0.750 [get_iobanks 39]
132 #####
133
134 # End

```

Fig. 9: Copy over top-level constraints from MIG Example Design

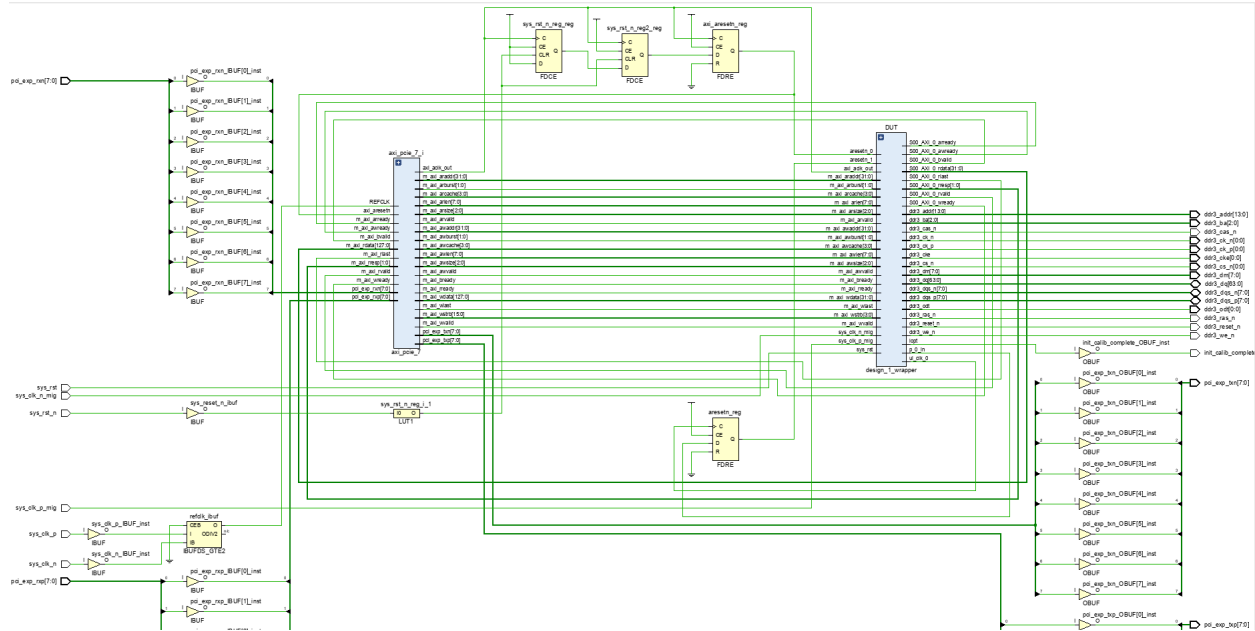


Fig. 10: Example schematic of infrastructure Block Diagram (BD)

## 14.4 Modifying and Running the Simulation

Just like the example in [section 2.5](#) of the AXI MM to PCIe IP Overview, the first step to running our simulation is to import the correct simulation files from the MIG example project (`ddr3_model.sv`, `ddr3_model_parameters.vh`, and `wiredly.v`). For more information on how to import these files, please reference that section. As an additional reference, these files have also been attached below.

---

**Important:** `ddr3_model.sv` file available [here](#).

---



---

**Important:** `ddr3_model_parameters.vh` file available [here](#).

---



---

**Important:** `wiredly.v` file available [here](#).

---

Now, we will need to edit our simulation top file to accommodate the MIG and DDR3 memory model, as well as include our block diagram from earlier. In this case, you can simply download the above files and import them into your design, but it is again recommended that you read through and try to understand the modifications made below.

Some notes about the modifications made to the PCIe example design top file:

- Parameters changed:
  - `TCQ` → `TCQ_MIG` (duplicate name)
  - `ADDR_WIDTH` → `ADDR_WIDTH_MIG` (duplicate name)
  - `RESET_PERIOD` = 100 (convert to nanoseconds)
- Wires/Regs changed:

- sys\_rst\_n → sys\_rst\_n\_mig (duplicate name)

- Variables changed:

- In the memory model instantiation, the variable *i* had to be changed to *s* due to a duplicate name

```
genvar r,s;
generate
  for (r = 0; r < CS_WIDTH; r = r + 1) begin: mem_rnk
    for (s = 0; s < NUM_COMP; s = s + 1) begin: gen_mem //change i to s
      ddr3_model u_comp_ddr3
      (
        .rst_n    (ddr3_reset_n),
        .ck       (ddr3_ck_p_sdram[(s*MEMORY_WIDTH)/72]),
        .ck_n     (ddr3_ck_n_sdram[(s*MEMORY_WIDTH)/72]),
        .cke      (ddr3_cke_sdram[((s*MEMORY_WIDTH)/72)+(1*r)]),
        .cs_n     (ddr3_cs_n_sdram[((s*MEMORY_WIDTH)/72)+(1*r)]),
        .ras_n    (ddr3_ras_n_sdram),
        .cas_n    (ddr3_cas_n_sdram),
        .we_n     (ddr3_we_n_sdram),
        .dm_tdq   (ddr3_dm_sdram[s]),
        .ba       (ddr3_ba_sdram[r]),
        .addr     (ddr3_addr_sdram[r]),
        .dq       (ddr3_dq_sdram[MEMORY_WIDTH*(s+1)-1:MEMORY_WIDTH*(s)]),
        .dqs      (ddr3_dqs_p_sdram[s]),
        .dqs_n    (ddr3_dqs_n_sdram[s]),
        .tdqs_n   (),
        .odt      (ddr3_odt_sdram[((s*MEMORY_WIDTH)/72)+(1*r)])
      );
    end
  end
endgenerate
```

Fig. 11: Changing variable ‘i’ to ‘s’ due to duplicate name

- MIG input system and reference clocks: - Due to timescale issue (MIG simulation top file is in picoseconds, PCIe simulation top file is in nanoseconds),

We were forced to change the system and reference clocks to run at 250MHz instead of 200MHz (4ns period instead of 5ns period). This in turn causes the MIG ui\_clk to run at 125MHz instead of 100MHz. However, everything in the simulation should still run fine.

- Instantiations included:

- Top file from design sources
- DDR3 memory model
- Wire delay modules

- In order to determine when init\_calib\_complete goes HIGH for the MIG, a simple check that displays “MIG Calibration Done” when this event occurs was added.

Now, if we were to click *Run Behavioral Simulation*, the standard PCIe example simulation would run, which would simply perform a read and a write to address 0x0000\_0010. For debugging purposes, it may be smart to try and

```

//*****//
// Clock Generation
//*****//

initial
    sys_clk_i = 1'b0;
always
    sys_clk_i = #2 ~sys_clk_i; //use 250MHz input clock instead of 200MHz due to timescale issue

assign sys_clk_p = sys_clk_i;
assign sys_clk_n = ~sys_clk_i;

initial
    clk_ref_i = 1'b0;
always
    clk_ref_i = #2 ~clk_ref_i; //250MHz reference clock input

```

Fig. 12: Change system and reference clock to 250MHz

```

initial //insert this to check for MIG calibration
begin : Logging
    wait (init_calib_complete);
    $display("MIG Calibration Done");
end

```

Fig. 13: Finished MIG calibration

run this simulation to make sure that everything is set up properly. However, we want to be able to read and write our own data to our own specific addresses. In order to do this, we will need to edit the simulation header file called `sample_tests1.vh`. This file can be located in the *Verilog Header* folder within *Simulation Sources*. As a reference, we have also attached our own `sample_tests1.vh` file below for you to download.

---

**Important:** You can download our custom simulation header file [here](#).

---

Under the comment that says “MEM 32 SPACE” in the BAR Testing section, a 60us delay is included to allow for the MIG to finish calibrating before attempting to read and write from it. The predefined tasks `TSK_TX_BAR_WRITE` and `TSK_TX_BAR_READ` perform the custom reads and writes. The definitions of these tasks can be found in the `pci_exp_usrapp_tx.v` file contained within the Root Port simulation model.

To test the MIG, the sample data `0xABCD_BEEF` was written to address `0x0000_0010`, which corresponds to address `0x0000_00010` on the MIG. If the read data equals the written data, then the message *MIG Test Passed* will appear in the TCL console.

```
2'b10 : // MEM 32 SPACE
begin
    #60000; //delay to allow for init calib complete

    $display("[%t] : Transmitting TLPs to Memory 32 Space BAR %x", $realtime,
        board.RP.tx_usrapp.ii);

    //PERFORM CUSTOM WRITE TO BAR MIG at BAR0 (0x0000_0000 -> 0x0000_1FFF)
    #1000;
    board.RP.tx_usrapp.TSK_TX_BAR_WRITE(3'b0,32'h10,8'd0,3'b0,32'habcd_beeef);
    #1000;

    // Event : Memory Read 32 bit TLP
    //-----

    // make sure P_READ_DATA has known initial value
    board.RP.tx_usrapp.P_READ_DATA = 32'hffff_ffff;

    //PERFORM CUSTOM READ TO MIG at BAR 0
    board.RP.tx_usrapp.TSK_TX_BAR_READ(3'b0,32'h10,8'd0,3'b0);
    board.RP.tx_usrapp.TSK_WAIT_FOR_READ_DATA;

    //CHECK IF DATA IS A MATCH
    if (board.RP.tx_usrapp.P_READ_DATA != {board.RP.tx_usrapp.DATA_STORE[3],
        board.RP.tx_usrapp.DATA_STORE[2], board.RP.tx_usrapp.DATA_STORE[1],
        board.RP.tx_usrapp.DATA_STORE[0] })
    begin
        $display("[%t] : MIG Test FAILED --- Data Error Mismatch, Write Data %x != Read Data %x",
            $realtime, {board.RP.tx_usrapp.DATA_STORE[3],board.RP.tx_usrapp.DATA_STORE[2],
            board.RP.tx_usrapp.DATA_STORE[1],board.RP.tx_usrapp.DATA_STORE[0]},
            board.RP.tx_usrapp.P_READ_DATA);
        test_failed_flag = 1;
    end
    else
    begin
        $display("[%t] : MIG Test PASSED --- Write Data: %x successfully received",
            $realtime, board.RP.tx_usrapp.P_READ_DATA);
    end
end
```

Fig. 14: MIG Test Passed

In order to test the BRAM controller (aka the DUT), I sent the data `0x1234_4321` to address `0x0000_2000`, which should correspond to address `0x0000_0000` on the BRAM controller. If the read data equals the written data, then the message “BRAM Test Passed” will appear in the TCL Console.

Now that we have built our simulation environment, we can go ahead and Run Behavioral Simulation.

---

**Note:** If the simulation fails to launch, the TCL console will direct you to the location of a log file that will provide

---

```

//PERFORM CUSTOM WRITE TO BRAM at BAR 0 (0x0000_2000 -> 0x0000_3FFF)
#1000;
board.RP.tx_usrapp.TSK_TX_BAR_WRITE(3'b0,32'h2000,8'd0,3'b0,32'h1234_4321);
#1000;

//PERFORM CUSTOM READ TO BAR 0
board.RP.tx_usrapp.TSK_TX_BAR_READ(3'b0,32'h2000,8'd0,3'b0);
board.RP.tx_usrapp.TSK_WAIT_FOR_READ_DATA;

//CHECK IF DATA IS A MATCH
if (board.RP.tx_usrapp.P_READ_DATA != {board.RP.tx_usrapp.DATA_STORE[3],
board.RP.tx_usrapp.DATA_STORE[2], board.RP.tx_usrapp.DATA_STORE[1],
board.RP.tx_usrapp.DATA_STORE[0] })
begin
    $display("[%t] : BRAM Test FAILED --- Data Error Mismatch, Write Data %x != Read Data %x",
    $realtime, {board.RP.tx_usrapp.DATA_STORE[3],board.RP.tx_usrapp.DATA_STORE[2],
    board.RP.tx_usrapp.DATA_STORE[1],board.RP.tx_usrapp.DATA_STORE[0]},
    board.RP.tx_usrapp.P_READ_DATA);
    test_failed_flag = 1;
end
else
begin
    $display("[%t] : BRAM Test PASSED --- Write Data: %x successfully received",
    $realtime, board.RP.tx_usrapp.P_READ_DATA);
end
end
end

```

Fig. 15: BRAM Custom Test

more specific error-related information for debugging.

---

The simulation should automatically pause itself after 1 nanosecond, and this is a good time to add the desired waveform signals into the simulation window. This can be done by navigating to the *Scope* window, right clicking on the signals you would like to see, and then clicking *Add to Wave Window*. I would personally recommend adding the signals from the *XILINX\_AXIPCIE\_EP* file, the *axi\_bram\_ctrl\_0* file, and the *mig\_7series\_0* file as shown in the image below.

Once we've added the correct signals, we can click on the green play button at the top left corner of the screen to resume the simulation.

---

**Note:** If the simulation stops early (before 100us) due to a timeout error from one of the PCIE root port files, we can go ahead and just click the green play button to force the simulation to resume anyways. If this becomes bothersome, we can comment out the timeout error from occurring like this:

---

Finally, the simulation should conclude around 110 us, and if you see the following messages in the TCL console, then the simulation was a success!

Additionally, we can view the AXI transactions in the simulation window. One important thing to notice is that the PCIE sent a write transaction to address `0x0000_2000` for the BRAM test, but because of the address offset that we specified for the BRAM controller back in the block diagram stage, the BRAM received this write request at address `0x0000_0000`. This is how we will be able to use the PCIE to read and write to multiple slave devices simultaneously.

## 14.5 Checking Timing, Viewing Power Reports, Monitoring I/O Placement:

After running through synthesis and implementation, Vivado provides us with several tools that we can use to monitor important factors of our design such as timing, power, and I/O placement.

The first category that we can take a look at is the Timing section. In this Design Timing Summary, we can see several aspects of our timing report, such as the total number of endpoints, worst negative slack, and most importantly, whether our device meets timing or not. In this example, we can see that our device successfully meets all of the timing requirements as shown in the figure below.

If we click on the *Check Timing* tab on the left side of the screen, it will show us a more detailed layout of the timing summary

In this case, we can see that there are 4 total errors with our timing: 2 `no_input_delays` and 2 `no_output_delays`. If we click on those respective sections on the left side of the screen, we can see which exact ports are afflicted by these errors. However, since all of the timing constraints are still met within the design, it is alright to ignore these errors.

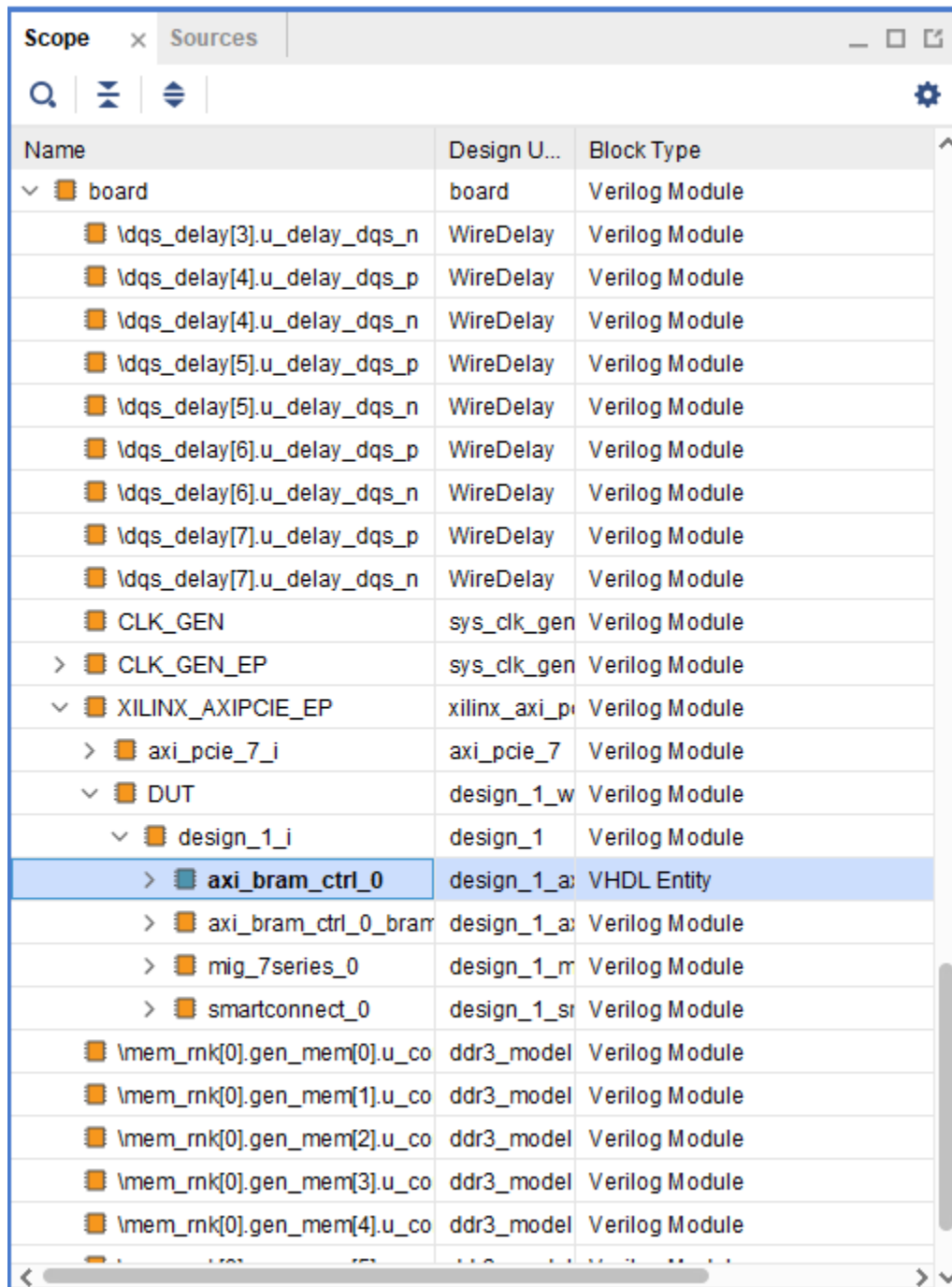
This is also the place where we would see if any clocks were not properly constrained. If this were the case, we would usually see a large amount of errors under the `no_clock` category.

If any of these errors were preventing our design from meeting timing, we can use the *Vivado Timing Constraints Wizard* to help us write clock constraints to fix these errors. In order to access the wizard, open up the implemented design, click on the *Tools* menu at the very top of the screen, and then click on *Timing* → *Constraints Wizard*.

---

**Note:** If you do decide to use the timing constraints wizard, it will automatically write the constraints for you based on the clocks you need to define, and it will **OVERWRITE** any constraints that you already have in your





Name	Design U...	Block Type
board	board	Verilog Module
\dqs_delay[3].u_delay_dqs_n	WireDelay	Verilog Module
\dqs_delay[4].u_delay_dqs_p	WireDelay	Verilog Module
\dqs_delay[4].u_delay_dqs_n	WireDelay	Verilog Module
\dqs_delay[5].u_delay_dqs_p	WireDelay	Verilog Module
\dqs_delay[5].u_delay_dqs_n	WireDelay	Verilog Module
\dqs_delay[6].u_delay_dqs_p	WireDelay	Verilog Module
\dqs_delay[6].u_delay_dqs_n	WireDelay	Verilog Module
\dqs_delay[7].u_delay_dqs_p	WireDelay	Verilog Module
\dqs_delay[7].u_delay_dqs_n	WireDelay	Verilog Module
CLK_GEN	sys_clk_gen	Verilog Module
> CLK_GEN_EP	sys_clk_gen	Verilog Module
XILINX_AXIPCIE_EP	xilinx_axi_pc	Verilog Module
> axi_pcie_7_i	axi_pcie_7	Verilog Module
DUT	design_1_w	Verilog Module
design_1_i	design_1	Verilog Module
> <b>axi_bram_ctrl_0</b>	design_1_axi	VHDL Entity
> axi_bram_ctrl_0_bram	design_1_axi	Verilog Module
> mig_7series_0	design_1_m	Verilog Module
> smartconnect_0	design_1_sr	Verilog Module
\mem_rnk[0].gen_mem[0].u_co	ddr3_model	Verilog Module
\mem_rnk[0].gen_mem[1].u_co	ddr3_model	Verilog Module
\mem_rnk[0].gen_mem[2].u_co	ddr3_model	Verilog Module
\mem_rnk[0].gen_mem[3].u_co	ddr3_model	Verilog Module
\mem_rnk[0].gen_mem[4].u_co	ddr3_model	Verilog Module

Fig. 16: BRAM Scope

```

438
439
440
441 //trn_rdst_rdy_toggle_count <= trn_rdst_rdy_toggle_count - 1;
442
443
444
445
446
447
448 reg [31:0] sim_timeout;
449 initial
450 begin
451   sim_timeout = `TRN_RX_TIMEOUT;
452 end
453
454 /* Transaction Receive Timeout */
455
456 always @(trn_clk or trn_rst_n or trn_rsrc_rdy_n) begin
457
458   // if (next_trn_rx_timeout == 0) begin
459   //   if (!EXPECT_FINISH_CHECK)
460   //     $display("%t: TEST FAILED --- Haven't Received All Expected TLEs", $realtime);
461   //   $finish(2);
462   // end
463
464   if ((trn_rst_n == 1'b0) && (trn_rsrc_rdy_n == 1'b0)) begin
465
466     next_trn_rx_timeout = sim_timeout;
467
468   end else begin
469
470     if (trn_lnk_up_n == 1'b0)
471
472       next_trn_rx_timeout = next_trn_rx_timeout - 1'b1;
473
474   end
475
476 end
477
478 endmodule // pci_exp_usrapp_rx
479
480

```

Fig. 17: Comment out timeout error

```

Tcl Console x Messages Log
[106412000.0 ps] : TSK_PARSE_FRAME on Receive
[107416000.0 ps] : MIG Test PASSED -- Write Data: abcdbeef successfully received
[108420000.0 ps] : TSK_PARSE_FRAME on Transmit
[109424000.0 ps] : TSK_PARSE_FRAME on Transmit
[110228000.0 ps] : TSK_PARSE_FRAME on Receive
board.\mem_rnk[0].gen_mem[5].u_comp_ddr3.cmd_task: at time 105975064.0 ps INFO: Precharge bank 0
board.\mem_rnk[0].gen_mem[6].u_comp_ddr3.cmd_task: at time 105975064.0 ps INFO: Precharge bank 0
board.\mem_rnk[0].gen_mem[7].u_comp_ddr3.cmd_task: at time 105975064.0 ps INFO: Precharge bank 0
board.\mem_rnk[0].gen_mem[0].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[1].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[2].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[3].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[4].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh

```

Fig. 18: MIG Test Passed

```

Tcl Console x Messages Log
[Icons]

board.\mem_rnk[0].gen_mem[1].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[2].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[3].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[4].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[5].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[6].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
board.\mem_rnk[0].gen_mem[7].u_comp_ddr3.cmd_task: at time 111115064.0 ps INFO: Refresh
[111424000.0 ps] : BRAM Test PASSED -- Write Data: 12344321 successfully received
[111424000.0 ps] : Finished transmission of PCI-Express TLPs
Test Completed Successfully
$finish called at time : 111424 ns : File "C:/Users/pnesb/VC707_PCIE_MIG_BRAM/VC707_PCIE_MIG_BRAM.srcs/sim_1/imports/imports/sample_tests1.vh" Line 394
run: Time (s): cpu = 00:16:13 ; elapsed = 00:23:06 . Memory (MB): peak = 4082.293 ; gain = 0.000

```

Fig. 19: BRAM Test Passed

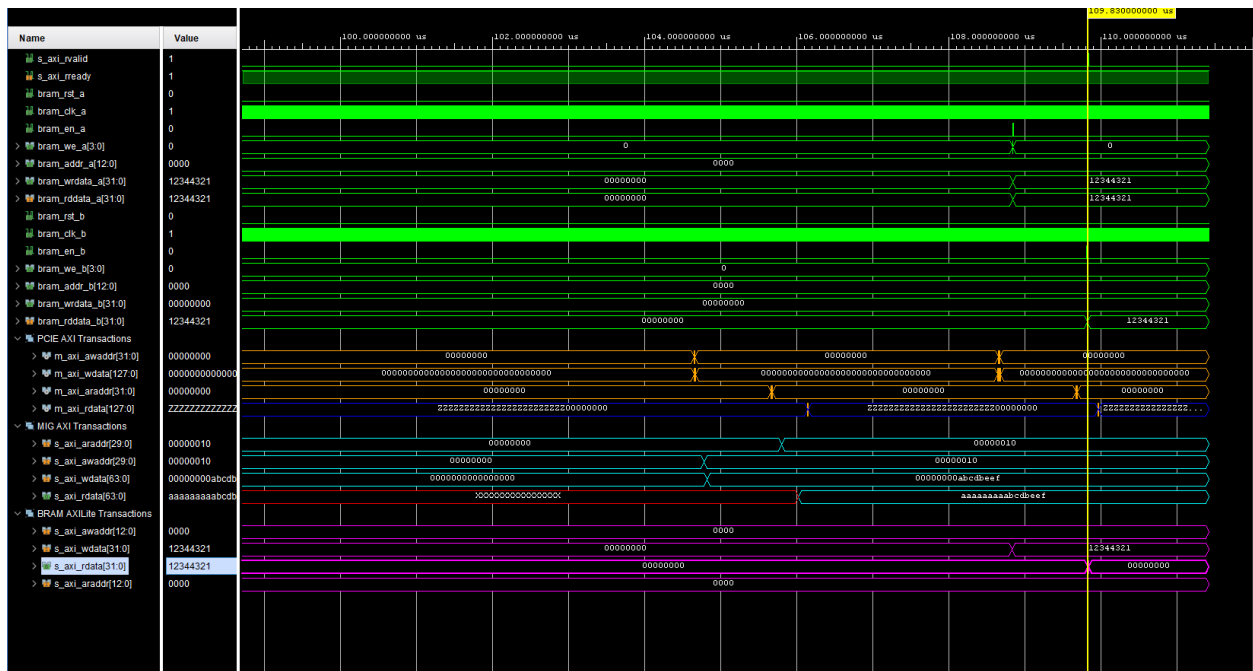


Fig. 20: BRAM MIG Waveform

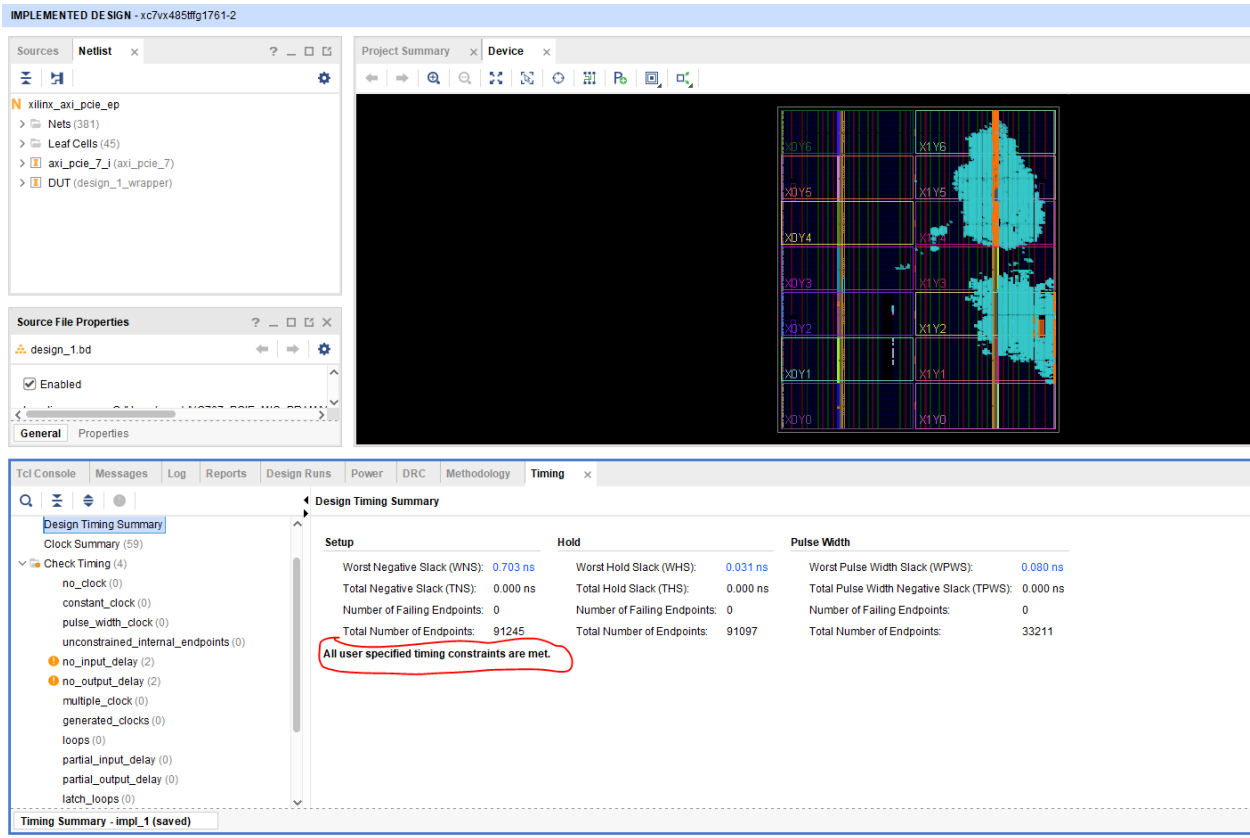


Fig. 21: Timing Summary Met

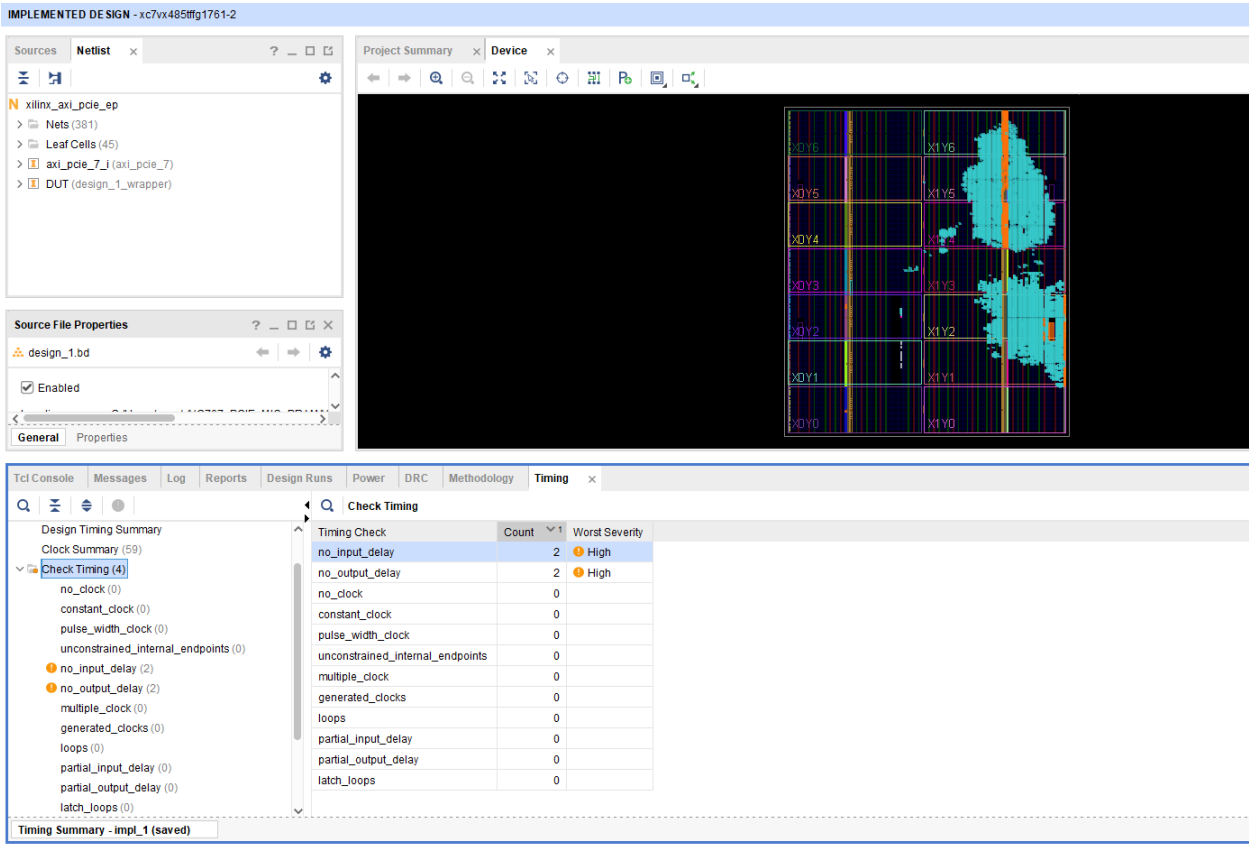


Fig. 22: Check Timing Summary

target constraints file. Personally, I would recommend copying and pasting the text from your target constraints file somewhere safe before running the wizard.

To check the *Vivado Power Report* for our design, click on the **Power** tab within the implemented design.

From here, we can see additional information relevant to the on-chip power required for implementation, as well as the power distribution for each FPGA primitive used in order to build the design (clocks, PLLs, I/O, BRAM, etc.)

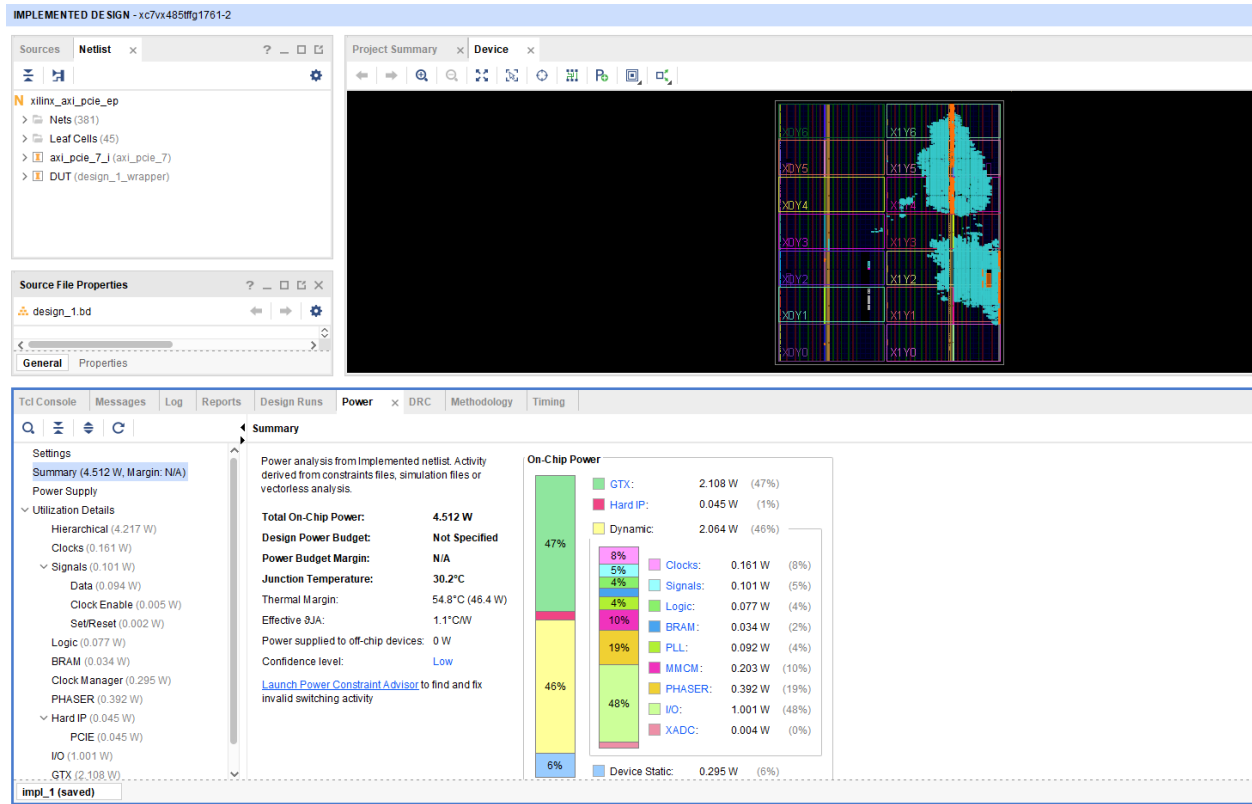


Fig. 23: Check Power Summary

In this case, we can see that the total on-chip power required is 4.512 Watts, which is broken down into the individual FPGA components in the diagram to the right.

One other very handy tool that Vivado provides for us is the ability to view and modify the I/O planning of the design. In order to access the I/O planning page, open up the implemented design, select the *Layout* menu at the very top of the screen, and then select *I/O Planning*.

This should open up a new tab on the Implemented design called **I/O Ports**, and navigating through this tab allows you to view all of the pin locations defined within your constraints, as well as their respective location within the FPGA.

Similar to the *Timing Constraints Wizard*, we can manually assign the input/output ports of our designs to any respective package pin port, and the Vivado tool will write the constraints for us. However, it will also overwrite any previously written constraints, so always make sure to copy and paste your top level constraints somewhere safe before saving any edits.

Other things that we can do within this window include setting the I/O Std type and enabling/disabling pullup resistors.

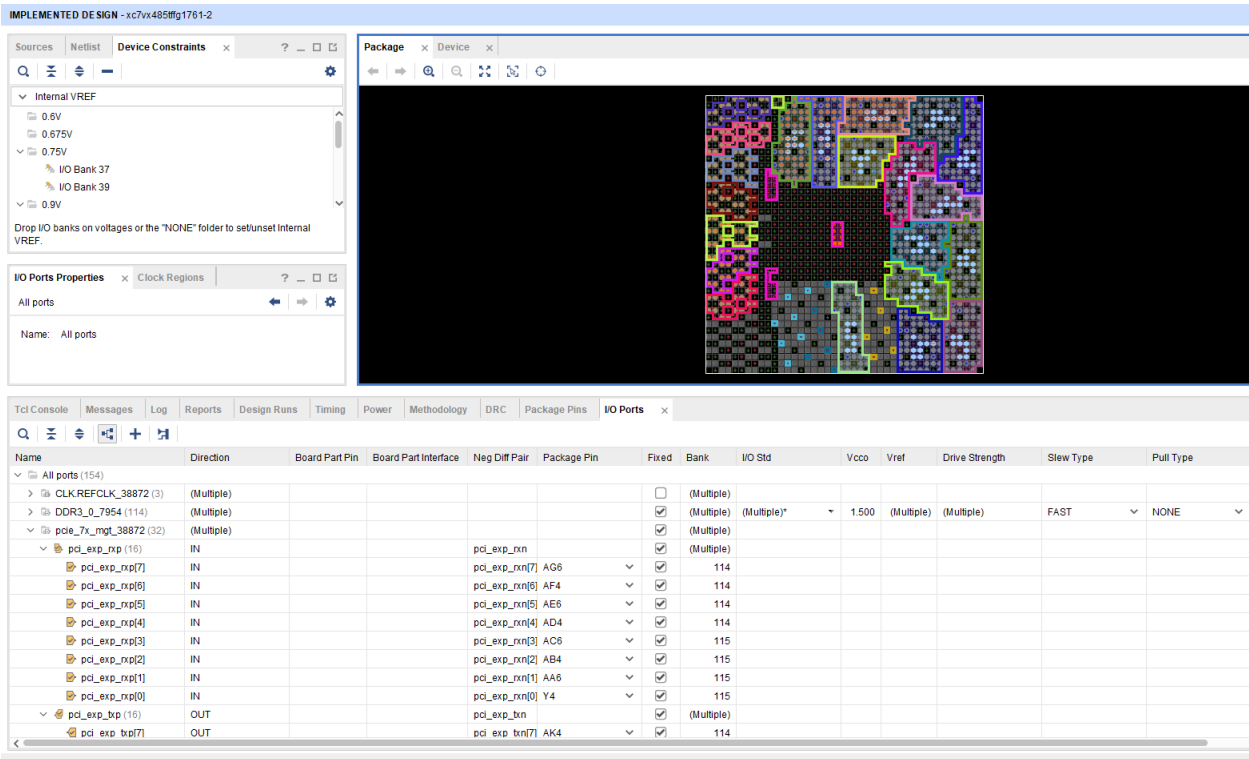


Fig. 24: IO Pin Planning





## BUILDING AN EMULATION ENVIRONMENT (WITHOUT A BOARD)

---

**Note:** All pages are under construction as we work to finalize this project. Please be patient!

---

---

**Important:** This guide uses Vivado 2020.2, so while the software and IPs will change in the future, underlying principles will remain the same.

---

Create a new project and select your project device. For this article, we will assume that no pre-existing board files will be used, except for the MIG's UCF constraint file. For example, we will use the Virtex-7 2000T FPGA for this design (similar to the HTG-700 development board from HTG which both has DDR3 SoDIMM memory and 8x Gen.3 PCIe) which at the moment does not have prepackaged board files in a standard Vivado installation. This design should also work with Vivado's WebPACK version (eg. you can use the Kintex-7 XCKU025 which is compatible with all IPs), given you have access or are willing to write UCF and XDC constraints.

As such, we will not include any source files within this article, except our example AXI counter. We encourage you to step through and generate the design yourself, as parameters will vary between FPGAs. We will, however, give samples of our source code so that you can step through the process yourself.

Here is a companion video with the Virtex-7 2000T that can be used alongside this article for further clarification.

### 15.1 Project Start (Building the Block Diagram)

If you need a refresher on digital design and FPGAs, read this [article](#) first.

If this is your first Vivado project, make sure you download the Vivado Design Suite [here](#). If you wish to follow this tutorial with a Virtex-7 or equivalent FPGA and have access to a license, use Vivado's License Manager to install the license. Otherwise, download the WebPACK version for access to the Kintex-7 FPGAs.

---

**Note:** As of Vivado 2020.2, there is a bug where Vivado cannot create a project if there is a space in your Windows/Linux username. Be careful if you try to use this or earlier versions.

---

Create a new RTL project and choose your project device. For this article, we will choose the xc7v000tflg1925-2 FPGA. After the project initializes, create a new block diagram (also referred to as a BD).

In your new BD, generate a DMA/Bridge Subsystem for PCI Express IP using the plus sign at the top of the BD window. Each FPGA will require a different customization, but you can refer to our companion video for our example configuration. After the IP has been generated, right click the IP block and select *Open IP Example Design...*, allowing Vivado to generate and manage everything. A new example design project will automatically open.

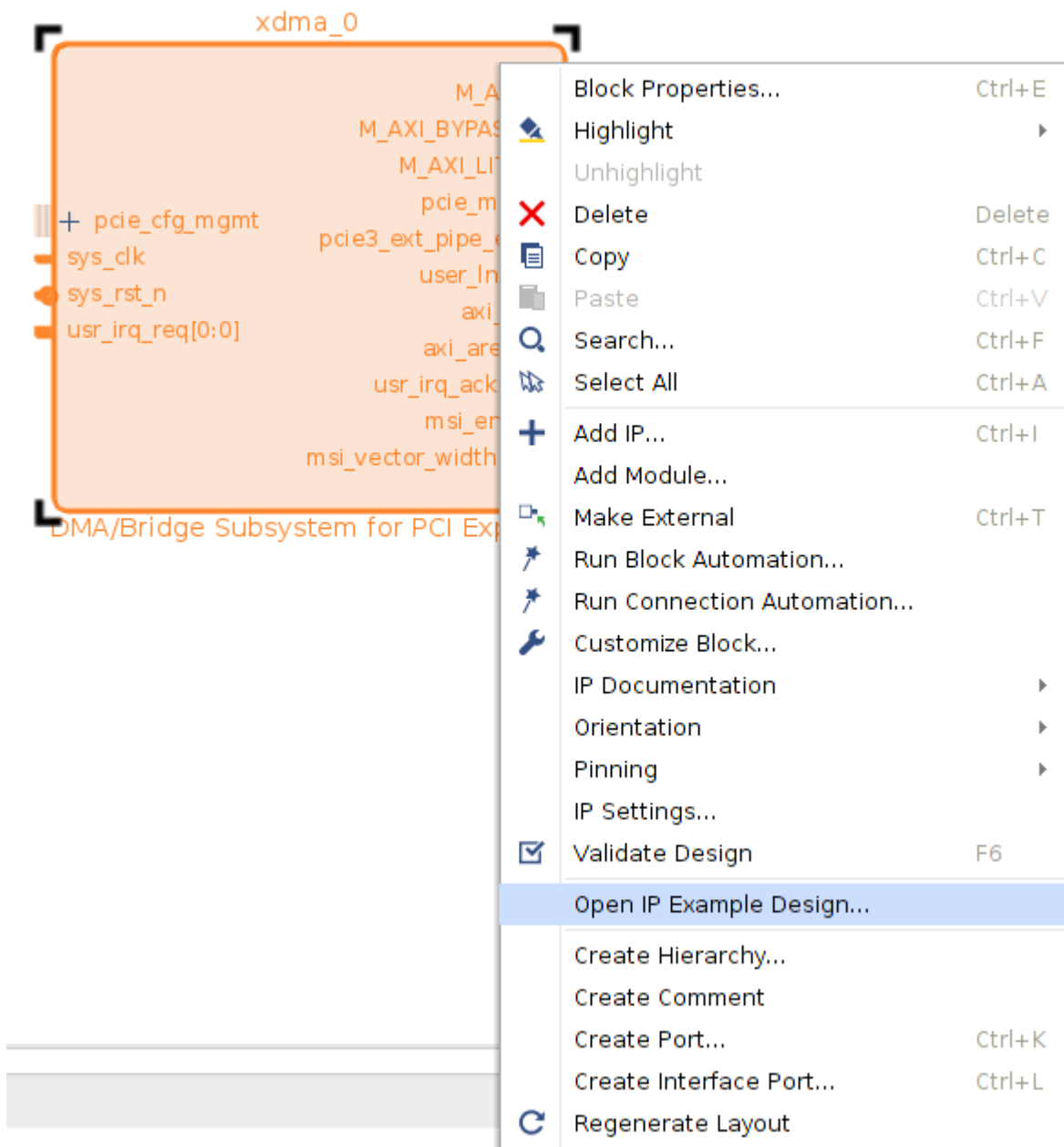


Fig. 1: Opening the example design for XDMA IP

After the example XDMA project opens, create a new block design in this new project (you can close out of the first project, as we will not use it going forward). The current source directory should look similar to this:

In this BD, generate a AXI SmartConnect, which we will use to control each IP. You can read more about AXI and the SmartConnect [here](#). We generated our SmartConnect with 2 Master and 3 Slave interfaces, 2 Clock Inputs, and an ARESETN input. The PCIe Master has 3 BARs, so there are 3 slave interfaces. We also have 2 AXI masters for both the DUT and MIG, and 2 Clocks for the PCIe and MIG. `aclk_0` is the PCIe reference clock (in this case is 125 MHz) and must be clocked separately from the entire system. We will use the MIG's clock for the rest of the design. Also generate a MIG 7 Series IP for our onboard DDR3 memory. You can read more about the MIG [here](#). For our MIG customization, refer to our companion video as a guide, but make sure you have access to a UCF pin constraints file (examples can be found online).

With both IPs in the BD generated, make the `S00_AXI`, `S01_AXI`, `S02_AXI`, `aclk_0`, and `aresetn_0` ports of `smartconnect_0` and the `SYS_CLK_0`, `sys_rst_0`, `DDR3_0`, `ui_clk_sync_rst_0`, `ui_clk_0`, `mmcm_locked_0`, and `init_calib_complete_0` ports of `mig_7series_0` external by right clicking each port and selecting *Make External*. We are making these pins external to utilize them elsewhere in the design and to monitor them during simulation, so we will instantiate these pins in Verilog later.

Double click on the `S00_AXI_0`, `S01_AXI_0`, and `S02_AXI_0` pins to modify their parameters. `S00_AXI_0` and `S01_AXI_0` will represent the two data signals from our DMA PCIe top module (DMA and bypass). You can read more about the DMA PCIe IP [here](#). As such, to properly connect both signals, `S00` and `S01` should be set to AXI4 and the ID Width for both `S00` and `S01` needs to be set to 4 (as the PCIe AXI data signals will throw an error when not set to 4 or unconnected). The AXI ID width determines how many IDs we can allocate ( $2^4 = 16$ ), which in turn determines how many transactions the AXI Master can track and reorder with the AXI transaction ID (side note - packet reordering allows an AXI Master to correctly order transactions to ensure data integrity, akin to TCP networking). The ID width itself is determined by the number of bits between `cap_max_link_width - 1:0` (eg.  $X4 = C\_M\_AXI\_ID\_WIDTH - 1 = 3:0$  which is 4 bits). We will use an AXI data width of 64 bits and an address width of 32 bits.

The external `S02` pin corresponds to the DUT. The parameters will depend based on which DUT is being tested. In our example, we will test an AXI4Lite DUT, which only has a data width of 32 bits and an ID width of 0 (as there are no ID signals in the AXI4Lite protocol). Accordingly, we will set the protocol of the `S02` pin to AXI4Lite.



Fig. 2: Source Directory of our new XDMA project

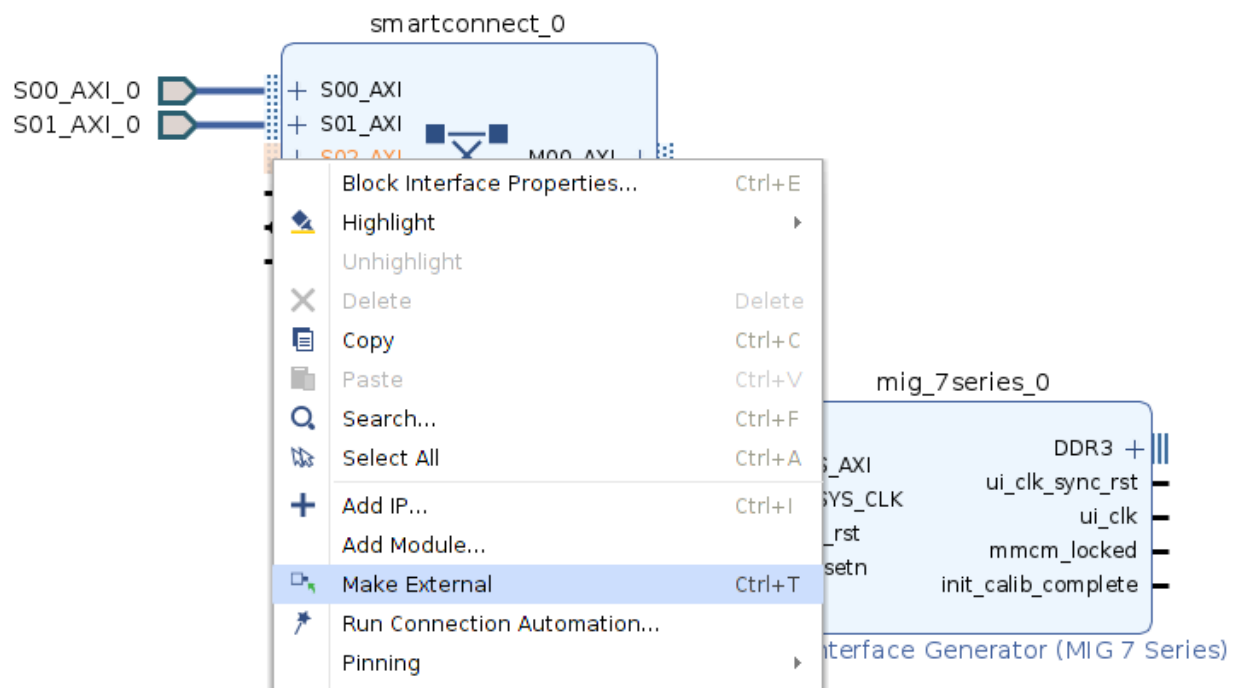


Fig. 3: Making MIG and SmartConnect pins external

## PROJECT SUMMARY

### 16.1 Project Abstract

It can be challenging for universities to provide applicable, real-world experience to undergraduate students. Utilizing development boards provided by Qualcomm, students will be able to explore fundamental HDL-based concepts not normally covered in an academic setting. Project effectiveness and student engagement is studied for further improvements.

The project is separated into three distinct subsystems: software, infrastructure, and DUT. The relationship between each subsystem is outlined according to the block diagrams here and on the introductory page. Each subsystem and its complementary documentation is tested for cohesiveness according to student participation and subsequent feedback. The subsystems make up a comprehensive prototyping environment that will provide valuable FPGA experience for undergraduate students in the future and act as a valuable asset for the University of San Diego's computer engineering department moving forward.

### 16.2 Project Background

Our primary goal was to provide an industry-grade development kit to foster advanced digital design and hardware platform competencies. This included focusing on building a development kit for the given Qualcomm hardware, setting up the infrastructure of the board, instantiating the design components, and utilizing software to directly interface with the board.

The project objectives concentrated on having clear and concise enough information for an undergraduate engineering student to utilize and provide illustrations/examples in order to deepen comprehension with benchmark tests for the completion of each task and providing additional resources for further research.

This capstone project was part of the 2021 University of San Diego Shiley-Marcos School of Engineering & Computing Showcase. The landing page can be found [here](#).

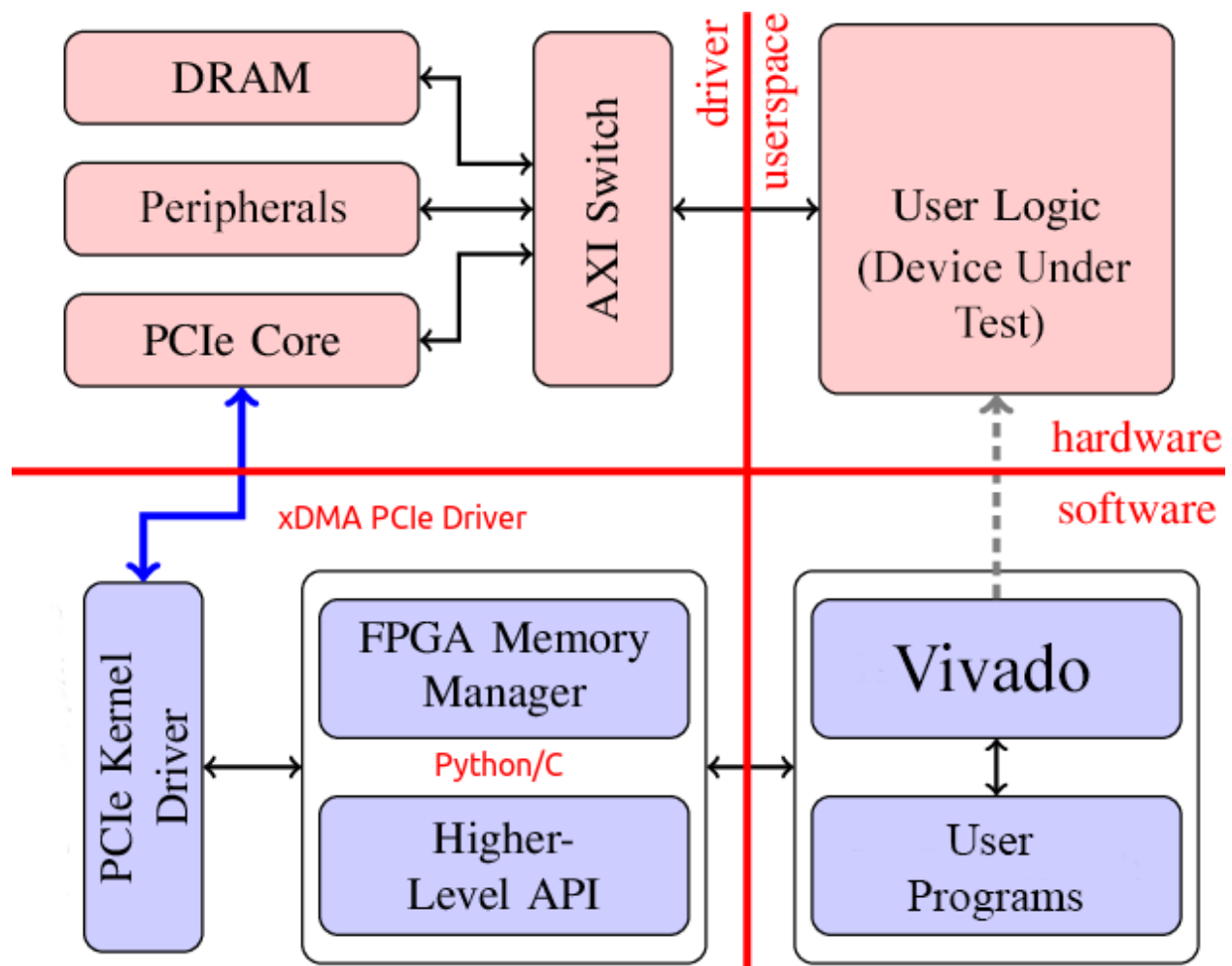


Fig. 1: Top-level block diagram of system

## ADDITIONAL RESOURCES

### 17.1 Tutorials

The following websites are external resources that we highly recommend taking a look at:

- [HDLBits](#) - Great for learning Verilog
- [ZipCPU](#) - Interesting FPGA verification tutorials
- [Nandland](#) - FPGA and digital design fundamentals
- [Xilinx Forums](#) - Support for Xilinx FPGAs





## INDICES AND TABLES

- [genindex](#)
- [search](#)

### 18.1 Acknowledgements

FPGAEmu was created and is maintained by a small group of collaborators. Thanks to everyone that has contributed, including:

- [Chadmond Wu](#)
- [Perry Nesbet](#)
- [Haley Guastaferrro](#)
- [Gabriel Pellot](#)
- [Jose Zamudio Gomez](#)
- [Ruiwen Wang](#)
- [Venkat Shastri, PhD](#)
- [Jeffrey Teza](#)

Special thanks to our sponsors that made this all possible: