# Computation II - 5EIB0
# Lab 1: Creating a Counter v3.0

**This assignment assumes that you have read the Appendix section called "The Basics of Logic Design," in the textbook as directed by the reading list.**

The goal of this lab is to create a hardware counter that can connect to an AXI4lite bus, so that it can be configured and used from a processor. Hardware counters are common system building blocks that can be used for many purposes. Timekeeping is one common use. By counting the number of clock cycles that have passed and knowing the frequency of the clock signal, the count can be translated into the passage of wall time (i.e. time in hours, minutes and seconds as we usually understand it). The exercises will guide you through the process of creating a project, simulating it and finally using it on your Pynq board.

*Please note that this is the first assignment in which you really get hands on with hardware implementation and the Vivado tools. You might feel a little out of your depth. This is normal. Read the exercises carefully and do not try to rush. The most important outcome is that you understand what you have been doing and not that you just mindlessly follow the steps to finish the exercises. The Vivado tools are professional tools with lots of options and these exercises will guide you through using a small but useful subset of them.*

## Contents

## 1 Learning Verilog

Hardware Description Languages (HDLs) are languages that enable a means of representing electronic circuits using an abstract textual representation. Software programming languages, such as C, are good at describing algorithms (the application's behaviour) at a relatively high-level of abstraction. The code is compiled and linked to form a binary executable made up of machine code instructions for a processor of some sort. HDLs, such as VHDL and Verilog, are not only able to describe the behaviour of electronic circuits, but also the circuit structure at various levels of abstraction.

You are probably used to thinking of electronic circuits using a graphical representation with symbols for components connected by lines representing wires. A basic function of HDLs is to enable electronic system designers to describe these structures in text. However, HDLs are not simply a textual means to describe a graphical representation, such as XML used in SVG. HDLs are Domain Specific Languages (DSLs) that have been specifically designed to also enable high-level behavioural descriptions, like you are used to in C, that resolve to a functioning circuit design. As you will no doubt learn while completing these labs, not everything that you can describe and simulate successfully in an HDL results in a well-defined functioning implementation.

In Computation II, you will apply the Verilog HDL to describe electronic circuits that are synthesisable, i.e. are realisable as functioning electronic circuits. To achieve this, you will learn the basics of the Verilog HDL. We will focus on the Verilog-2001 subset of the language, which is standardised as IEEE 1364-2001. We chose this version of Verilog because it is 100% supported by many Verilog synthesis tools. Later versions of the Verilog standard are often only partially supported making code portability problematic. Through practical application in the labs, you will learn how to write Verilog code in an unambiguous manner, so that it can be synthesised to a well-defined[1] functioning implementation.

The overall objective of the labs in Computation II is to learn how to create and modify functional electronic system descriptions, and some techniques that can be applied to achieve this. Learning Verilog is primarily a means to this end, i.e. you will learn sufficient Verilog to achieve this goal, but Computation II is not a comprehensive Verilog language course.

The Xilinx Vivado tool suite is used to create hardware project designs for Xilinx FPGA chips, such as the one on the Pynq board. We will use the Vivado tools throughout this lab to create our counter design from a Verilog description.

[1]Well-defined in this instance simply means that your code will not leave important structural or behavioural circuit description open to interpretation by the synthesis tool.

## 2 Creating a Vivado Project

Following the Vivado workflow the first thing that you need to do is create a new project. Follow these instructions using the Virtual Machine.

**Task 1**. Create a Vivado project.

1. First create a new folder in the file explorer. It is recommended that you place this in the SharedWork folder. **Do not use spaces in the folder name.**
2. Open Vivado, you can use the menu in the top left of the screen.
3. To create a project select "File" → "Project" → "New". A pop-up will be presented on the screen. Click "Next".
4. You can now choose a project name and location. By default Vivado will create a project folder in your home folder. Change the location to the folder that you just made. Make sure you have selected "Create project subdirectory". Click "Next" to continue.
5. Vivado has several different types of projects that you can create. Select "RTL Project" with "Do not specify sources at this time". You will add sources to your project later. Once again click "Next" to proceed.
6. Lastly, configure the project to target the PYNQ-Z2 board. In the "Boards" tab, select the "pynq-z2" board. Click "Next" and then "Finish".

## 3 Combinational Logic

Ultimately, Verilog code is translated into a hardware implementation. Verilog code describes an electronic circuit that commonly consists of sequential state elements (registers) with combinational logic used to derive the next state. This terminology should not by confused with the Verilog language itself in which sequential code blocks (e.g. "always" blocks with blocking assignments) are often used to describe combinational logic.

Parallel execution of code that is compiled to execute on a processor (like the C-code you wrote in Computation I) is limited by the number of physical resources (e.g. cores, threads) that the processor has available. Combinational logic is inherently parallel whenever there are no shared dependencies. This means that you can theoretically create infinite parallelism. However, parallel combinational logic means multiple copies of that logic, using physical area on a silicon wafer or physical resources on an FPGA, which are both ultimately limited. These limits on parallelism are generally much less restrictive than for code executing on a processor.

Our counter requires that a stored value is incremented whenever an event to be counted occurs. Incrementing a value requires an addition operation to be performed. This is a common operation in HDLs and can be achieved in the Verilog language with a high-level of abstraction by simply using the "+" operator. In the following tasks you will not use the "+" operator, but implement the addition operation in combinational logic from low-level logic operators. You will do this to gain experience in creating combinational logic and to understand that operators with high-levels of abstraction conceal complexity that you should be aware of.

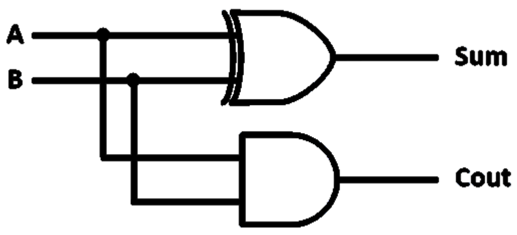


Figure 1: Logic circuit of half adder.

## 3.1 Logic design

```
module half_adder(
    // all IO signals of this module
    // have a wire width of 1
    input a,
    input b,
    output s,
    output c
    );

// continuous signal assignments
// defined using the 'assign' keyword
assign  s = a ^ b; // bitwise XOR ^
assign c = a & b; // bitwise AND &

// module definition must be terminated
// with an 'endmodule' keyword
endmodule
```

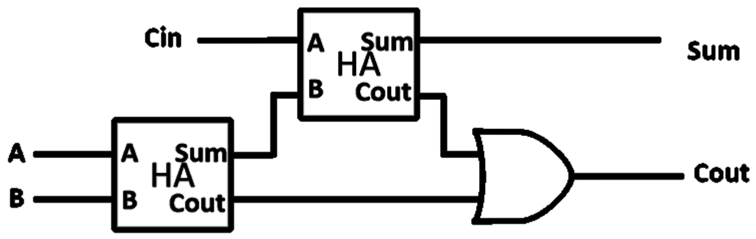


Figure 2: Logic circuit of full adder.

```
module full_adder(
    input a,
    input b,
    input cin,
    output s,
    output cout
    );

// wires declared using the 'wire' keyword
wire h1sum, h1Cout, h2Cout;

// module instantiation of the custom half adder module.
// the module name 'half_adder' is followed by a user chosen
// unique identifier , 'HA1' in this instance.
// signals are connected by name using the '.' notation
half_adder HA1 (
.a(a),
.b(b),
.s(h1sum),
.c(h1Cout)
);

// another example of module instantiation
half_adder HA2 (
.a(cin),
.b(h1sum),
.s(s),
.c(h2Cout)
);

// another example of continuous assignment
assign cout = h1Cout | h2Cout; // bitwise OR |

endmodule
```

**Task 2.** Create a new design source file called `adder.v` (Add Sources, Add or create design sources, Create File, give a name ending with .v for the file, Finish, Ok and use the default values in the I/O Port Definitions screen). Create the following modules: `half_adder`, `full_adder` and `adder` (you will design the adder in the task 4.1) in this file. First make a half adder (input: `a`,`b`; output: `s`, `cout`) using only combinational logic (no always block), see Fig.1. Then combine two 1 bit half adders to make a 1 bit full adder (input: `cin`,`a`,`b`; output: `s`, `cout`). See Fig. 2.

## 3.2 Testing on the board

```
set_property –dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { s }];
set_property –dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { cout }];

set_property –dict { PACKAGE_PIN D20   IOSTANDARD LVCMOS33 } [get_ports { cin }];
set_property –dict { PACKAGE_PIN L20   IOSTANDARD LVCMOS33 } [get_ports { a }];
set_property –dict { PACKAGE_PIN L19   IOSTANDARD LVCMOS33 } [get_ports { b }];
```

**Task 3.** Now that you have created the combinational logic for the full adder you can already test its functionality on the Pynq board. To do that you are going to connect ports `a` and `b` to Pynq board buttons BTN2 and BTN3, `cin` to BTN1, and the output of the full adder, `s` and `cout` to Pynq board LEDs LD2 and LD3.

1. Create a new **constraint** source ("file" → "Add sources"), select "Add or create constraints". Name this file `constraints.xdc` and place it local to the project.
2. Add the set_property constraints from the previous box to the file. This file tells the Vivado tools which physical IO pin (PACKAGE_PIN) on the FPGA to link each port to on the full adder design.

Each physical IO pin on the Zynq FPGA on the Pynq board has a static unique address that is represented by a letter followed by a one or two digit number, e.g. N16.

**Task 4.** Generate the bitstream. You will likely see some warnings on the Vivado console. You should not ignore these completely, but in general they are just advising you about potential problems in your design and do not indicate that your design will not function. If however your design does not function these warnings are a useful place to start looking.

1. Click on "Generate Bitstream".
2. If you get a pop-up message asking to run synthesis and implementation click on "Yes".
3. In the next pop-up keep all the default settings and click "OK". This will take a few minutes.

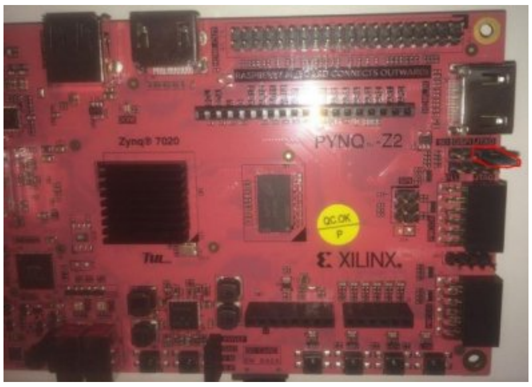


Figure 3: Jumper position on the board for JTAG mode

**Task 5**. Now that you have your bitstream you will now load it onto the FPGA using the JTAG interface.

1. Ensure your Pynq board is switched of and the USB cable is not connected to your computer.
2. Make sure the jumper on your board is set in the correct position for JTAG programming, see Figure 3.
3. Connect the USB cable to your computer.
4. Power on your board.
5. Connect the board to your virtual machine.
6. Click on "Open Hardware Manager".
7. Connect to the board by clicking on "Open target" and then "Auto Connect" in the menu that pops up.
8. Click on "Program device". A dialogue box should pop up.
9. The "Bitstream file" field should already point to the location of your generated bitfile. Click the "Program" button.
10. Check that the DONE LED on your Pynq board is on.
11. Use buttons BTN1-3 to verify the functionality of your full adder.

# 4 Implement a Counter

## 4.1 Logic Design

**Task 6.** Having verified that your full adder is functional, define a 32 bit adder called adder (in adder.v ) by using a for loop. This module has 2 inputs `a` and `b` and an output `c` such that $c = a + b$.

*You might be wondering where the description is of how to do this in Verilog. Use the Verilog resources on the course page or a search engine to work out how to achieve this. This is part of the exercise. As with any programming language there will be times when you know what you want to do but are unsure how to do it efficiently, or even do it at all. Finding and using reliable resources that you understand is just part of the process of learning a new language. If you are really stuck, you can ask for some pointers from the supervisors in the lab.*

**Task 7.** Write the logic for the counter. The counter should count the amount of seconds that have passed and write this into `count`. The counter should only be counting if `enable` is high and `reset` is low, if `reset` is set high then the counter should reset itself at the positive edge of the clock signal (active high synchronous reset). Use a parameter called `number_of_clk_cycles` for the number of clock cycles in a second, this will allow you to speed up simulation later on. You will set the correct value for this parameter to count the correct number of clock cycles in a second, depending on the clock frequency, later on. Use the 32 bit full adder you wrote in the previous task.

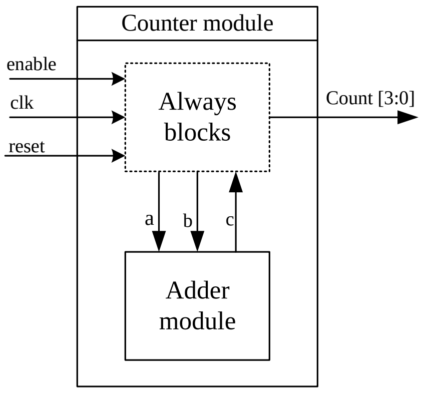


Figure 4: Counter module Block Diagram.

## 4.2 Testing on the board

```
// The next line enables the Vivado tools to permit a clock provided from an input pin
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk_IBUF]

// SW0−1
set_property −dict { PACKAGE_PIN M20  IOSTANDARD LVCMOS33 } [get_ports { enable }];
set_property −dict { PACKAGE_PIN M19  IOSTANDARD LVCMOS33 } [get_ports { reset }];

// LD0−3
set_property −dict { PACKAGE_PIN R14  IOSTANDARD LVCMOS33 } [get_ports { count[0] }];
set_property −dict { PACKAGE_PIN P14  IOSTANDARD LVCMOS33 } [get_ports { count[1] }];
set_property −dict { PACKAGE_PIN N16  IOSTANDARD LVCMOS33 } [get_ports { count[2] }];
set_property −dict { PACKAGE_PIN M14  IOSTANDARD LVCMOS33 } [get_ports { count[3] }];

// BTN3
set_property −dict { PACKAGE_PIN L19  IOSTANDARD LVCMOS33 } [get_ports { clk }];
```

**Task 8**. Just like with the full adder, you can now check the functionality of your design on the Pynq board.

1. Replace the contents of the constraints file that you made earlier with the set_property constraints from the previous box to the file. This file tells the Vivado tools which physical IO pin (PACKAGE_PIN) on the FPGA to link each port to on the full adder design.
2. In the Vivado tool, in the "Project Manager", right click on your counter instance in the "Sources" panel. Then select "Set as Top" in the menu. *This sets your counter as the top-level design for implementation.*
3. set the `number_of_clk_cycles` parameter of your counter to 1. The counter should increment every clock cycle.
4. Generate the bitstream.
5. Program the board and check the functionality of your design.

*Does your counter always only count one at a time when you press the clk button BTN3? Are you able to reset the counter value? Are you able to explain what you see?*

## 4.3 Logic Testbench

As this design is more complicated than the full_adder you have tested on the FPGA board for task 3.2 , it is possible that your counter does not work properly on the FPGA. Therefore, to make sure that the logic works as intended, you are need to create a testbench. A testbench generates values for the inputs of the module which you are testing. A complete testbench shall go through all the possible situations your module can experience to test its correct functionality. A testbench is only a simulation file and is not part of the final implementation.

**Task 9**. Create a testbench file.

1. Create a new **simulation** source ("file" → "Add sources"), select "Add or create simulation sources". Name this file `counter_tb.v`. Specify no input or output ports. Your testbench includes other modules and is not instantiated in other modules.
2. Set the testbench in the simulation sources as top module.
3. Create an instance of your custom counter module in the testbench. You can specify a number for your parameter `number_of_clk_cycles`. If this parameter is left to its correct value then you will have to simulate several seconds to completely verify that your logic works correctly. This would take a very long time (real time is different from simulation time!). Thus, to speed up the simulation, change the value of the parameter to 10 cycles.

If you have done everything correctly you should have a file structure that looks like Fig. 5.

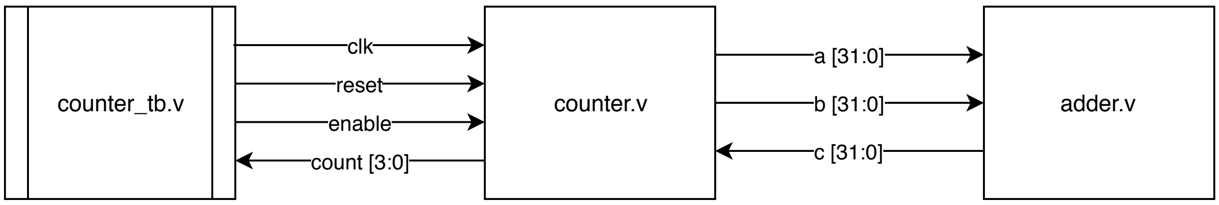


Figure 5: Structure overview of logic with all signals and modules (except parameters).

**Hint:** Make use of an initial block and time delays in your testbench.

**Task 10**. Write a testbench that generates the inputs for your logic module. The testbench should be able to verify that the `reset` and `enable` inputs both work correctly. The testbench should also generate the clock signal for your module.

**Task 11**. On the left side, click on "Run Simulation" → "Run Behavioural Simulation". A new window will appear, in which you will see the signals over time. Verify that the values of `count` are correct. Make sure that the zoom level allows you to see the signals and that the simulated time is long enough.

**Hint:** You can add and remove signals displayed in the wave window.

## 5 The Vivado IP-block

Reusability is very important in digital electronics. A hardware block is commonly used multiple times in the same, or in different projects. Vivado uses an IP-block abstraction that allows multiple IP-blocks to be integrated graphically (or using TCL command line scripting) into a larger system. IP-blocks are commonly written in a Hardware Description Language (HDL) such as Verilog or VHDL. In this course, we will use Verilog.

Vivado's IP-block abstraction allows you to view each IP-block as a stand-alone Vivado project. When you create a project that has an input/output interface, you can use Vivado to package it into a compact form that can be instantiated in a bigger project.

In this exercise, the counter IP will be split into two parts:

1. The logic that handles the counting
2. The AXI4Lite interface (see section 5.2) which handles communication from other parts of the hardware.

By the end of this exercise you should have an IP with the same structure as Figure 6. The following sections will instruct you on how to build your counter IP-module step-by-step, starting with how basic logic is used to create higher-level functionality that is required by a counter, such as an adder.

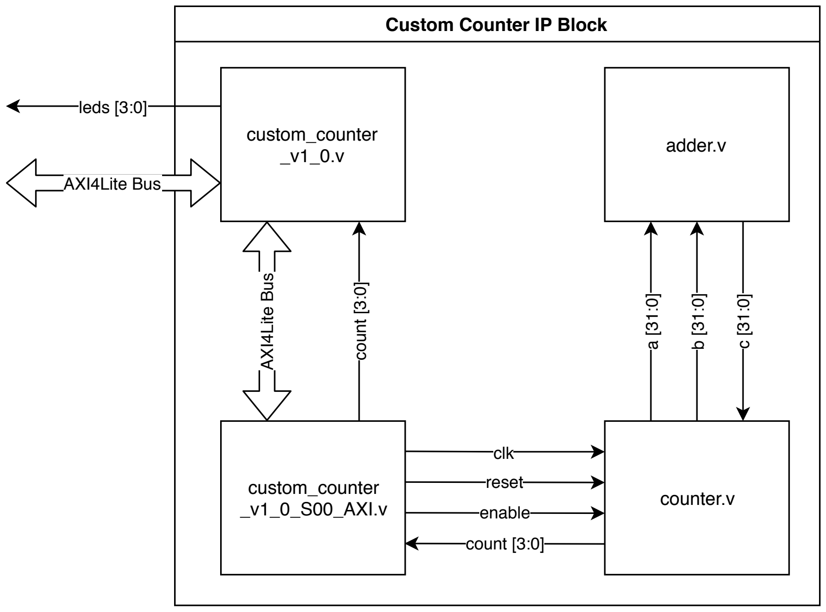


Figure 6: Structure overview of the final Vivado IP-block with all signals and modules (except parameters).

## 5.1 The Counter Logic and preparation

A simple counter increments a stored value whenever an event to be counted occurs. We now proceed to describe how to implement this in hardware using the Vivado tools.

**Task 12.** Create a Vivado IP-block

1. Select "Tools" → "Create and package New IP...".
2. A new window is opened and click "Next". Select "Create a new AXI4 peripheral" and click "Next" again.
3. The next step is to pick the name, location and version of your IP. Since you will be making a counter, change the name to `custom_counter`. Do not change the IP location. Click "next" to proceed.
4. You can now add and remove AXI interfaces on your IP block. You will only need one AXI4Lite slave interface with four registers. Make sure you have one interface with type "Lite", mode "slave" and at least four registers. Leave the rest of the settings on default. Click on "next" to proceed.
5. You will be greeted with an overview of your settings select "Edit IP" and click "Finish".
6. A new Vivado window will be opened. The source of the project already contains some verilog files (`custom_counter_v1_0.v` and `custom_counter_v1_0_S00_AXI.v`), which contain code for the AXI4Lite interface. You have to modify these two files, but before that, you need to import the counter that you defined and tested for the section 4.
7. Therefore, create a new design source ("File" → "Add sources"), select "Add or create design sources". Click "Create File", Name this file `counter.v` and place it in `../ip_repo/custom_counter_1.0/hdl`. Click "Finish" to proceed.
8. In the next window, it is possible to specify the input and output ports of your custom module. You are required to have the following inputs: 1 bit wide `clk`, 1 bit wide `reset` and 1 bit wide `enable`. Specify the following output: 4 bit wide `count`. This is done by enabling bus, specify 3 in the *MSB* column and 0 in the *LSB* column. **The names of the ports are very important, they will be used to connect your module to the other elements. Use the provided names for your input and output ports in the Labs.**
9. A file is generated, put the definition of the counter that you implemented in task 4.1 into this file. Please notice that you might need to create `adder.v` file and put the definition of adder into it like what you did for `counter.v`.

## 5.2 The AXI4Lite interface

The Counter is not intended to be a standalone block, you want it to be able to communicate with the ARM core of the PYNQ board. The ARM processors use an AXI4 bus to transfer data to and from memory and the peripherals. For your IP, you use the AXI4Lite interface, this interface has several channels and uses handshakes to transfer data. A detailed explanation of this process can be found at `https://www.realdigital.org/doc/32101c99686fe25ec47bedd94e76dc96`.

Vivado will generate the files for the bus slave interface when you select "Create a new AXI4 peripheral". The pre-generated slave interface has four registers (or more if you increased the number of registers when creating the IP) in which it stores data. When it receives a read request it sends the data from the requested register. Similarly, write requests are written to these registers. You are going to use these registers to receive instructions from the ARM core and send data back.

**Note:** You can go back to the "Project Manager" layout where you were in before by clicking on "Project Manager" on the left.

**Hint:** When instantiating a module, the input and output ports of the module can be of type `reg` or `wire`.

**Task 13**. Vivado has already generated a skeleton of the AXI4-Lite interface, you simply need to connect the AXI registers to the inputs and outputs of your module.

1. At the bottom of the file `custom_counter_v1_0_S00_AXI.v` there is a line stating "// Add user logic here". After this line make an instance of your module `counter.v`. Connect the `clk` input of your module to the clock of the AXI bus, on default the wire is called `S_AXI_ACLK`.
2. The remaining input and output should be connected to the registers of the AXI bus. The AXI4Lite slave interface reads and writes to and from the register `slv_reg0`, `slv_reg1`, `slv_reg2` and `slv_reg3`. Connect directly the the lowest bit of `slv_reg0` to input `enable`, the lowest bit of `slv_reg1` to input `reset`. Add a 4 bit wide "output wire" called `count`, then connect `count` (the outport port of `counter`) to the new output port(`count`) of `custom_counter_v1_0_S00_AXI` module. You do not need to connect anything to `slv_reg3`. In an `always` block, assign the value of `count` to `slv_reg2` using a blocking assignment (i.e. =). Keep in mind that the register is 32 bits and the count is 4 bits.
3. Modify the rest of the code in a way that there is no multiple drivers for a signal. **Hint:** you need to comment out the automatically generated parts that write a value to `slv_reg2` in `custom_counter_v1_0_S00_AXI` module.

**Task 14**. You also want to display the seconds on the PYNQ board leds. To do this the IP needs an external port that can be connected to the four leds on the board. You also want the parameter `number_of_clk_cycles` to be accessible from the top level module so you can change the value later without having to edit the code of IP (`counter.v`).

1. Add a 4 bit wire called `leds` to `custom_counter_v1_0.v` after the line "// Users to add ports here". Add the parameter `number_of_clk_cycles` under the line "// Users to add parameters here"
2. Connect the output wire `leds` you just created to output of your ip , which is `count`.
3. Pass the `number_of_clk_cycles` parameter from the top module `custom_counter_v1_0` through the `custom_counter_v1_0_S00_AXI` instance and to the `number_of_clk_cycles` parameter of the `counter` instance. In this way, later you will be able to directly modify the parameter from the top module in the schematic diagram without having to modify the code in the `counter.v` file.
4. Make `custom_counter_v1_0.v` as the top module if it is not.

The file structure you have now will be similar the one given in Fig. 7 except for the testbench.

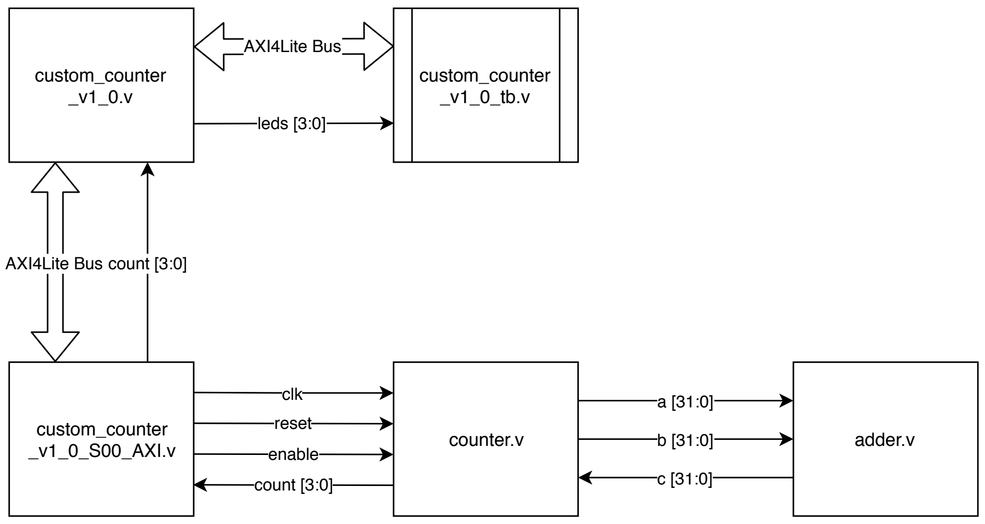


Figure 7: Structure overview of AXI4Lite testbench setup with all signals and modules (except parameters).

## 5.3 AXI4Lite Testbench

To verify that the AXI interface works properly and as intended you are going to create a testbench. This testbench does not have to fully test the logic of the counter, you have already done that, but you do want to test whether or not the AXI bus can read and write to the registers.

**Task 15**. Vivado allows you to have multiple simulation sets, this allows you to easily switch between testbenches.

1. Go to "File" → "Add Sources".
2. Select "Add or create simulation sources" and click "Next"
3. Click on "Create File" and name it `custom_counter_v1_0_tb.v`.

You should now have a folder structure such as in figure 8.

**Hint:** Do not forget to set the reset line of the AXI bus in your testbench. There are two types of reset line, one type is active low reset, the other active high. Think about how you reset the AXI4-lite bus and your module. Is it active high or active low ? The active low reset line is most of the time named something like `...resetn...` and the active high reset line is most of the time something like `...reset...`.

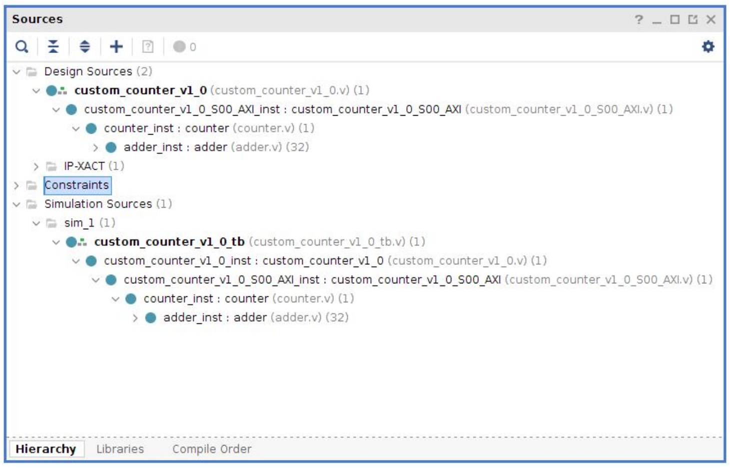
**Hint:** For now (in the testbench) your counter registers start at address 0 and go up from there. They are each 4 bytes wide so `slv_reg0` is at address 0, `slv_reg1` at 4 and `slv_reg2` at 8.

**Task 16**. Write the testbench `custom_counter_v1_0_tb.v` that verifies your counter. Enable and disable your counter at regular intervals. Read the register in which `count` is stored, from the AXI bus and display it in the terminal. You can use `\$display(...)` to print to the terminal. Compare the `count` value from the AXI-bus to the output pins of the module, that you can see in the waveforms screen. Also verify that you can properly reset your counter. You can use the testbench template given in appendix A. Use the read and write tasks to interact with the AXI bus. you will need to include `custom_counter_v1_0.v` in your testbench.

**Hint:** Do the displayed values always correspond to the ones you see in the waveforms? If not, why?

**Task 17**. Simulate your testbench and verify whether your counter works. Use the wave window and the terminal.

You now should have a working IP, to be able to use it you need to package it. Vivado will create a compact package that can be included in other projects, like the one created in section 2.



```
Sources
Design Sources (2)
  custom_counter_v1_0 (custom_counter_v1_0.v) (1)
    custom_counter_v1_0_S00_AXI_inst : custom_counter_v1_0_S00_AXI (custom_counter_v1_0_S00_AXI.v) (1)
      counter_inst : counter (counter.v) (1)
        adder_inst : adder (adder.v) (32)
  IP-XACT (1)
Constraints
Simulation Sources (1)
  sim_1 (1)
    custom_counter_v1_0_tb (custom_counter_v1_0_tb.v) (1)
      custom_counter_v1_0_inst : custom_counter_v1_0 (custom_counter_v1_0.v) (1)
        custom_counter_v1_0_S00_AXI_inst : custom_counter_v1_0_S00_AXI (custom_counter_v1_0_S00_AXI.v) (1)
          counter_inst : counter (counter.v) (1)
            adder_inst : adder (adder.v) (32)
Hierarchy   Libraries   Compile Order
```

Figure 8: Expected Vivado sources overview.

**Note:** After packaging the counter IP Vivado will delete the project for it, if you want go back and edit it later you will have to click “Edit packaging settings” and uncheck “Delete project after packaging” before you click on “Re-Package IP”.

**Task 18**. Package the IP

1. Change the value of `number_of_clk_cycles` back to the correct value such that the counter counts up by 1 every second (the frequency of the clock is 100 MHz). You only have to do this in the top file `custom_counter_v1_0.v`. **Do not change the parameters in the other files!**.
2. Click in the left side of the menu under “Project Manager” on “Package IP”.
3. In the “Package IP” screen that you just opened select “Customization Parameters”. You can edit the parameters of your IP here. Using the right mouse button click and select “Import IP Parameters”, click “OK”.
4. Now go to “Review and Package”. In the now open tab click on “Merge changes” for each item in the list.
5. Click on “Re-Package IP” and continue.

## 6 Simulating with the MicroBlaze

Vivado is unable to simulate the ZYNQ processor that is on the board. To be able to simulate with code running on a processor you will have to use a MicroBlaze. The MicroBlaze is a really small processor that Vivado can simulate. Later when implementing the hardware on the board you will replace the MicroBlaze with the ZYNQ core.

**Task 19**. Connect your IP to a MicroBlaze block.

1. Click on the left side of the screen under the top module "IP Integrator" on "Create block design". This will make a block design that you can use to connect the counter IP to the rest of the system. A new window is opened. Name the design `design_1`, and keep the rest of the settings untouched. Click "OK" to generate the block design. An empty window should be opened.
2. Click on the + to add IP blocks to the block design. Add your own custom counter (`custom_counter_v1_0`) as well as a "Simulation Clock Generator" and a "MicroBlaze".
3. Vivado can do the handwork of setting up and connecting the blocks for you. Click on "Run Block Automation" in the green bar. Select the simulation clock for the **"Clock Connection"** of the MicroBlaze, on default the block automation will create a new clocking wizard which is unnecessary.
4. Now click on "Run Connection Automation", This will connect the slave bus of the counter IP to the MicroBlaze. You will have to do this twice.
5. The last step is to make the connection port to the leds on the counter IP external. Select the wire and click on "Make External", make sure you have only selected the port and not the whole IP block.

Before packaging your IP you changed the value of the parameter `number_of_clk_cycles` to its correct value. Yet you still want to simulate the IP so you need to change this back to 10.

**Task 20**. Customize your IP block.

1. Right click on your IP and select "Customize Block".
2. Change the value of `number_of_clk_cycles` to 10.
3. Click "OK" to save the parameter.

You have now made all necessary connections, however you still need to supply the MicroBlaze with instructions. To do this you are going to write a simple C program in the SDK (Software Development Kit).

**Note:** The "Simulation Clock Generator" IP only works in simulations, you can not synthesize this part. Later on you will remove this part and use the clock signal generated by the ZYNQ processor instead. Any error stating that Vivado can not synthesize this block during the step "Generate Output Products" can be ignored.

**Task 21**. Launch the SDK and create a project.

1. To be able to launch the SDK you need to set a top module and to do this you need to create a wrapper for the block design. In the “Sources” tab right click on the block design and select “Create HDL Wrapper”, let Vivado automaticly update the wrapper and click “OK”
2. Set the wrapper as the top module.
3. In the sources overview right click on the block diagram file and select “Generate Output Products”.
4. For the SDK to be able to correctly compile your C code it needs to understand on what kind of hardware it will be running. Vivado needs to simulate the project once before it can export the hardware configuration. Go ahead and run the simulation. Then click on “File” → “Export” → “Export Hardware”. Click “OK”.
5. Open the SDK by going to “File” → “Launch SDK”.
6. In the newly opened SDK click on “File” → “New” → “Application Project”. Set the name of your project to `counter_code` and click on **“Finish”**, not “Next”.
7. You now need to add a C source file to your project. Right click the “src” folder and select “New” → “Source File”. Set the name to `main.c`.
8. Go back to the block diagram in Vivado, go to the Address editor. You need to know the base address of your counter block. You can find your counter under Data. Write down the “Offset Address”, you are going to need this later.

The register in your counter are addressable just like any other part of the memory. Your four register start at the “Offset Address” and go up from there. Writing to these registers is as easy as creating a pointer that points to the register and then writing and reading to/from the address stored by the pointer. This is only possible because you created a standalone project. This means that your code is the only code running on the processor so you do not have to worry about conflicts with other programs or the Linux kernel.

**Note:** The compiler might use caches to optimize your code. In this case anything you write to your registers are not actually written to the counter but saved in temporary memory. To prevent this you can make a `volatile int*` pointer to the address. This will tell the compiler to not use caches.

**Hint:** You can use hexadecimal values to set the value of a pointer.

**Task 22**. Write a C program that enables your counter and after a set interval resets it. You can use the value of `count` to time your reset. Use `int main(void){..}` as your starting point.

You now have a code that you can run on the MicroBlaze, however before you can simulate you have to tell Vivado that the MicroBlaze should run your code. You do this by associating the binary file you are about to create with the MicroBlaze. Vivado will load the instruction into the MicroBlaze memory for you.

**Task 23**. Associate your C code with the MicroBlaze.

1. In the SDK you first need to compile the code you just wrote. While you have `main.c` open, click on the hammer icon in the toolbar.
2. Go back to the block diagram. Right click the MicroBlaze and select “Associate ELF Files”.
3. Under “Simulation Sources” click on the three dots next to “microblaze_0”.
4. Click “Add Files” and goto `../<project name>/<project name>.sdk/counter_code/Debug/counter_code.elf`
5. With the newly added elf file selected, click “OK”.
6. You can now run the simulation. Check in the wave window whether or not your code works correctly.

## 7 Testing on the board

Now that you have verified that it all works using the simulations you will test the counter on your PYNQ board. To do this you will first need to replace the MicroBlaze with the ZYNQ core.

**Task 24**. Replace the MicroBlaze.

1. In the block diagram remove everything except the counter and the connection to the leds.
2. Add the "ZYNQ7 Processing System".
3. Now run the Block Automation and Connection Automation.

For the board you will need to restore your parameter `number_of_clk_cycles` back to the correct value. If you do not the leds will start blinking very fast.

**Task 25**. Change the value of `number_of_clk_cycles` to its original (correct) value, such that the counter adds one every second. Do this in the same way as you have done in section 6. Keep in mind that the default clock frequency is 100MHz.

Now the board design of the processor is ready, a few steps should be taken to program the FPGA. In the next task, you will connect the external ports of your block design, to the physical pins of the FPGA chip, which are connected to the external interfaces like leds, buttons and switches. For every FPGA chip, the physical connections are different.

**Task 26**. Connecting the external ports to physical pins.

1. Create a new **constraint** source ("file" → "Add sources"), select "Add or create constraints". Name this file `constraints.xdc` and place it local to the project.
2. A template constraint file of this model type of FPGA chip, is available on the internet. The template is also provided in appendix B. Uncomment the lines needed for the leds. Change the name of the wire to the names of the external ports. The names of your external ports can be found in `design_1_wrapper.v`, this is the HDL wrapper of the block diagram. You generated this earlier.

**Hint:** In the constraints file, find the correct section by looking at the names after the double hash. For example, in the line "set_property -dict { PACKAGE_PIN R14 IOSTANDARD LVCMOS33 } [get_ports { led[0] }]; #IO_L6N_T0_VRED_34 Sch=led[0]" the name of the external port in the block diagram should correspond to the name in the get_ports function. Change led[0] to the correct port name you used in the diagram and uncomment the first part of the line.

Your hardware model is now complete for this type of FPGA. You only need a bitstream to program your FGPA chip. In the next step, Vivado will first convert your files to a diagram of flip-flops, lookup tables and other gate-level components. This is called "synthesize". The next step is to generate a physical layout for this model type of FPGA chip. This is called "implementation". The latest step, is to generate a bitstream, which is used to program the physical layout to the FPGA stream.

**Note:** Generating the bitstream can take a while (15 to 25 minutes for this project). If you open the console and log file, information will be outputted by the program while running.

**Task 27**. Generate the bitstream. This will probably generate a lot of warnings, this is not unusual so can be ignored.

1. Click on "Generate Bitstream".
2. If you get a pop-up message asking to run synthesis and implementation click on "Yes".
3. In the next pop-up keep all the default settings and click "OK"

The bitstream is ready. Now the FPGA should be programmed from the Vivado environment. The PYNQ board will be used in JTAG mode.

**Hint:** You can make use of `sleep()` and `usleep()` of the `"sleep.h"` library to have some time between the lines of code.

**Task 28**. Upload the bitstream to the FPGA and run your code with the SDK.

1. Set the jumper of your PYNQ board to JTAG like shown in figure 3. The red circle shows the correct position.
2. Connect the PYNQ board to your PC with only the USB cable, so no Ethernet cable.
3. If the board is turned on a red led should turn on, if it does not then you need to switch the board on.
4. Export the hardware. If asked to, generate the output products of the block diagram.
5. Open the SDK.
6. Program C code for resetting and enabling your module. You can use the same code as for the MicroBlaze however you will need to create a new project and regenerate the binary elf file. Make sure your address pointers are still pointing to the correct location (the address will probably be different from the MicroBlaze).
7. Upload the bitstream to the FPGA by clicking on “Xilinx” $\rightarrow$ “Program FPGA” $\rightarrow$ “Program”.
8. Now run your program in debug mode by clicking on “Run” $\rightarrow$ “Debug”.
9. The program will automatically stop before the first line of code. Click on “Resume” to run your C code. If everything done correctly, the leds of your PYNQ board should now be a binary counter, counting every second.

# Appendices

## A custom_counter_v1_0_tb.v

```
'timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
// Company: Eindhoven University of Technology
// Author: Diederik Markus
//
// Create Date: 04/12/2018
// Module Name: custom_counter_v1_0_tb
// Description: 5EIB0 lab 1 IP block testbench. This version should be given to
//  the students before they start with the assignments.
//
// Version: 1.1
//////////////////////////////////////////////////////////////////////////////////

module custom_counter_v1_0_tb();
   //clock and reset_n signals
   reg aclk =1'b1;
   reg arstn = 1'b0;

   //Write Address channel (AW)
   reg [3:0] write_addr =4'd0;     //Master write address
   reg [2:0] write_prot = 3'd0;    //type of write(leave at 0)
   reg write_addr_valid = 1'b0;    //master indicating address is valid
   wire write_addr_ready;          //slave ready to receive address

   //Write Data Channel (W)
   reg [31:0] write_data = 4'd0;   //Master write data
   reg [3:0] write_strb = 4'd0;    //Master byte-wise write strobe
   reg write_data_valid = 1'b0;    //Master indicating write data is valid
   wire write_data_ready;          //slave ready to receive data

   //Write Response Channel (WR)
   reg write_resp_ready = 1'b0;    //Master ready to receive write response
   wire [1:0] write_resp;          //slave write response
   wire write_resp_valid;          //slave response valid

   //Read Address channel (AR)
   reg [3:0] read_addr = 4'd0;     //Master read address
   reg [2:0] read_prot =3'd0;      //type of read(leave at 0)
   reg read_addr_valid = 1'b0;     //Master indicating address is valid
   wire read_addr_ready;           //slave ready to receive address

   //Read Data Channel (R)
   reg read_data_ready = 1'b0;     //Master indicating ready to receive data
   wire [31:0] read_data;          //slave read data
   wire [1:0] read_resp;           //slave read response
   wire read_data_valid;           //slave indicating data in channel is valid

   reg [31:0] output_data = 32'b0;

   wire [3:0] leds;

   custom_counter_v1_0 #(
         .C_S00_AXI_DATA_WIDTH(32),
         .C_S00_AXI_ADDR_WIDTH(4),
         .number_of_clk_cycles(10)
   ) custom_counter_v1_0_inst (
         .leds(leds),

         .s00_axi_aclk(aclk),
         .s00_axi_aresetn(arstn),

         .s00_axi_awaddr(write_addr),
         .s00_axi_awprot(write_prot),
         .s00_axi_awvalid(write_addr_valid),
         .s00_axi_awready(write_addr_ready),

         .s00_axi_wdata(write_data),
         .s00_axi_wstrb(write_strb),
         .s00_axi_wvalid(write_data_valid),
         .s00_axi_wready(write_data_ready),

         .s00_axi_bresp(write_resp),
         .s00_axi_bvalid(write_resp_valid),
         .s00_axi_bready(write_resp_ready),

         .s00_axi_araddr(read_addr),
         .s00_axi_arprot(read_prot),
         .s00_axi_arvalid(read_addr_valid),
         .s00_axi_arready(read_addr_ready),

         .s00_axi_rdata(read_data),
         .s00_axi_rresp(read_resp),
         .s00_axi_rvalid(read_data_valid),
         .s00_axi_rready(read_data_ready)
   );

   //
   // ADD YOUR OWN LOGIC HERE.
   //

   task axi_read;
   input [31:0] addr;
   output reg [31:0] data;
   begin
         #3 read_addr <= addr; //puts adress on the bus
         read_addr_valid <= 1'b1; //indicating address is valid
         read_data_ready <= 1'b1; //indicting ready for a response

         wait(read_addr_ready); //wait till slave is ready to read the address

         @(posedge aclk); //handshake occurs
         read_addr_valid <= 1'b0;

         wait(read_data_valid);
         @(posedge aclk);
         data <=read_data;
         @(posedge aclk);
         read_data_ready <= 1'b0;
   end
   endtask

   task axi_write;
   input [31:0] addr;
   input [31:0] data;
   begin
         #3 write_addr <= addr;    //Put write address on bus
         write_data <= data;    //put write data on bus
         write_addr_valid <= 1'b1;    //indicate address is valid
         write_data_valid <= 1'b1;    //indicate data is valid
         write_resp_ready <= 1'b1;    //indicate ready for a response
         write_strb <= 4'hF;         //writing all 4 bytes

         //wait for one slave ready signal or the other
         wait(write_data_ready || write_addr_ready);

         @(posedge aclk); //one or both signals and a positive edge
         if(write_data_ready&&write_addr_ready)//received both ready signals
         begin
             write_addr_valid<=0;
             write_data_valid<=0;
         end
         else    //wait for the other signal and a positive edge
         begin
             if(write_data_ready)    //case data handshake completed
             begin
                 write_data_valid<=0;
                 wait(write_addr_ready); //wait for address address ready
             end
             else if(write_addr_ready)   //case address handshake completed
             begin
                 write_addr_valid<=0;
                 wait(write_data_ready); //wait for data ready
             end
             @ (posedge aclk);// complete the second handshake
             write_addr_valid<=0; //make sure both valid signals are deasserted
             write_data_valid<=0;
         end

         //both handshakes have occured
         //deassert strobe
         write_strb<=0;

         //wait for valid response
         wait(write_resp_valid);

         //both handshake signals and rising edge
         @(posedge aclk);

         //deassert ready for response
         write_resp_ready<=0;


         //end of write transaction
   end
   endtask
endmodule
```
appendix/custom_counter_v1_0_tb.v

# B constraints.xdc

```

################################################################################
## Company: Eindhoven University of Technology
## Author: Diederik Markus
##
## Create Date: 04/12/2018
## Module Name: constraints.xdc
## Description: 5EIB0 lab 1 constraints file . Should be given to
## the students before they start with the assignments.
##
## Version: 1.0
################################################################################

## This file is a general .xdc for the PYNQ-Z2 board
## To use it in a project :
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

## Clock signal 125 MHz

#set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports { sysclk }]; #IO_L13P_T2_MRCC_35 Sch=sysclk
#create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { sysclk }];

##Switches

#set_property -dict { PACKAGE_PIN M20   IOSTANDARD LVCMOS33 } [get_ports { sw[0] }]; #IO_L7N_T1_AD2N_35 Sch=sw[0]
#set_property -dict { PACKAGE_PIN M19   IOSTANDARD LVCMOS33 } [get_ports { sw[1] }]; #IO_L7P_T1_AD2P_35 Sch=sw[1]

##RGB LEDs

#set_property -dict { PACKAGE_PIN L15   IOSTANDARD LVCMOS33 } [get_ports { led4_b }]; #IO_L22N_T3_AD7N_35 Sch=led4_b
#set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports { led4_g }]; #IO_L16P_T2_35 Sch=led4_g
#set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 } [get_ports { led4_r }]; #IO_L21P_T3_DQS_AD14P_35 Sch=led4_r
#set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { led5_b }]; #IO_0_35 Sch=led5_b
#set_property -dict { PACKAGE_PIN L14   IOSTANDARD LVCMOS33 } [get_ports { led5_g }]; #IO_L22P_T3_AD7P_35 Sch=led5_g
#set_property -dict { PACKAGE_PIN M15   IOSTANDARD LVCMOS33 } [get_ports { led5_r }]; #IO_L23N_T3_35 Sch=led5_r

##LEDs

#set_property -dict { PACKAGE_PIN R14   IOSTANDARD LVCMOS33 } [get_ports { led[0] }]; #IO_L6N_T0_VREF_34 Sch=led[0]
#set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { led[1] }]; #IO_L6P_T0_34 Sch=led[1]
#set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { led[2] }]; #IO_L21N_T3_DQS_AD14N_35 Sch=led[2]
#set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { led[3] }]; #IO_L23P_T3_35 Sch=led[3]

##Buttons

#set_property -dict { PACKAGE_PIN D19   IOSTANDARD LVCMOS33 } [get_ports { btn[0] }]; #IO_L4P_T0_35 Sch=btn[0]
#set_property -dict { PACKAGE_PIN D20   IOSTANDARD LVCMOS33 } [get_ports { btn[1] }]; #IO_L4N_T0_35 Sch=btn[1]
#set_property -dict { PACKAGE_PIN L20   IOSTANDARD LVCMOS33 } [get_ports { btn[2] }]; #IO_L9N_T1_DQS_AD3N_35 Sch=btn[2]
#set_property -dict { PACKAGE_PIN L19   IOSTANDARD LVCMOS33 } [get_ports { btn[3] }]; #IO_L9P_T1_DQS_AD3P_35 Sch=btn[3]

##PmodA

#set_property -dict { PACKAGE_PIN Y18   IOSTANDARD LVCMOS33 } [get_ports { ja[0] }]; #IO_L17P_T2_34 Sch=ja_p[1]
#set_property -dict { PACKAGE_PIN Y19   IOSTANDARD LVCMOS33 } [get_ports { ja[1] }]; #IO_L17N_T2_34 Sch=ja_n[1]
#set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports { ja[2] }]; #IO_L7P_T1_34 Sch=ja_p[2]
#set_property -dict { PACKAGE_PIN Y17   IOSTANDARD LVCMOS33 } [get_ports { ja[3] }]; #IO_L7N_T1_34 Sch=ja_n[2]
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { ja[4] }]; #IO_L12P_T1_MRCC_34 Sch=ja_p[3]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports { ja[5] }]; #IO_L12N_T1_MRCC_34 Sch=ja_n[3]
#set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports { ja[6] }]; #IO_L22P_T3_34 Sch=ja_p[4]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports { ja[7] }]; #IO_L22N_T3_34 Sch=ja_n[4]

##PmodB

#set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports { jb[0] }]; #IO_L8P_T1_34 Sch=jb_p[1]
#set_property -dict { PACKAGE_PIN Y14   IOSTANDARD LVCMOS33 } [get_ports { jb[1] }]; #IO_L8N_T1_34 Sch=jb_n[1]
#set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { jb[2] }]; #IO_L1P_T0_34 Sch=jb_p[2]
#set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { jb[3] }]; #IO_L1N_T0_34 Sch=jb_n[2]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { jb[4] }]; #IO_L18P_T2_34 Sch=jb_p[3]
#set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports { jb[5] }]; #IO_L18N_T2_34 Sch=jb_n[3]
#set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { jb[6] }]; #IO_L4P_T0_34 Sch=jb_p[4]
#set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports { jb[7] }]; #IO_L4N_T0_34 Sch=jb_n[4]

##Audio

#set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { adr0 }]; #IO_L8P_T1_AD10P_35 Sch=adr0
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { adr1 }]; #IO_L8N_T1_AD10N_35 Sch=adr1

#set_property -dict { PACKAGE_PIN U5    IOSTANDARD LVCMOS33 } [get_ports { au_mclk_r }]; #IO_L19N_T3_VREF_13 Sch=au_mclk_r
#set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { au_sda_r  }]; #IO_L12P_T1_MRCC_13 Sch=au_sda_r
#set_property -dict { PACKAGE_PIN U9    IOSTANDARD LVCMOS33 } [get_ports { au_scl_r  }]; #IO_L17P_T2_13 Sch= au_scl_r
#set_property -dict { PACKAGE_PIN F17   IOSTANDARD LVCMOS33 } [get_ports { au_dout_r }]; #IO_L6N_T0_VREF_35 Sch=au_dout_r
#set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports { au_din_r  }]; #IO_L16N_T2_35 Sch=au_din_r
#set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports { au_wclk_r }]; #IO_L20P_T3_34 Sch=au_wclk_r
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { au_bclk_r }]; #IO_L20N_T3_34 Sch=au_bclk_r


## Single Ended Analog Inputs
##NOTE: The ar_an_p pins can be used as single ended analog inputs with voltages from 0-3.3V (Arduino Analog pins a[0]-a[5]).
##      These signals should only be connected to the XADC core. When using these pins as digital I/O, use pins a[0]-a[5].

#set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports { ar_an0_p }]; #IO_L3P_T0_DQS_AD1P_35 Sch=ar_an0_p
#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { ar_an0_n }]; #IO_L3P_T0_DQS_AD1P_35 Sch=ar_an0_n
#set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports { ar_an1_p }]; #IO_L5N_T0_AD9P_35 Sch=ar_an1_p
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports { ar_an1_n }]; #IO_L5N_T0_AD9N_35 Sch=ar_an1_n
#set_property -dict { PACKAGE_PIN K14   IOSTANDARD LVCMOS33 } [get_ports { ar_an2_p }]; #IO_L20P_T3_AD6P_35 Sch=ar_an2_p
#set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { ar_an2_n }]; #IO_L20P_T3_AD6N_35 Sch=ar_an2_n
#set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { ar_an3_p }]; #IO_L24P_T3_AD15P_35 Sch=ar_an3_p
#set_property -dict { PACKAGE_PIN J16   IOSTANDARD LVCMOS33 } [get_ports { ar_an3_n }]; #IO_L24P_T3_AD15N_35 Sch=ar_an3_n
#set_property -dict { PACKAGE_PIN J20   IOSTANDARD LVCMOS33 } [get_ports { ar_an4_p }]; #IO_L17P_T2_AD5P_35 Sch=ar_an4_p
#set_property -dict { PACKAGE_PIN H20   IOSTANDARD LVCMOS33 } [get_ports { ar_an4_n }]; #IO_L17P_T2_AD5P_35 Sch=ar_an4_n
#set_property -dict { PACKAGE_PIN G19   IOSTANDARD LVCMOS33 } [get_ports { ar_an5_p }]; #IO_L18P_T2_AD13P_35 Sch=ar_an5_p
#set_property -dict { PACKAGE_PIN G20   IOSTANDARD LVCMOS33 } [get_ports { ar_an5_n }]; #IO_L18P_T2_AD13P_35 Sch=ar_an5_n

##Arduino Digital I/O

#set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { ar[0] }]; #IO_L5P_T0_34 Sch=ar[0]
#set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { ar[1] }]; #IO_L2N_T0_34 Sch=ar[1]
#set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { ar[2] }]; #IO_L3P_T0_DQS_PUDC_B_34 Sch=ar[2]
#set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports { ar[3] }]; #IO_L3N_T0_DQS_34 Sch=ar[3]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { ar[4] }]; #IO_L10P_T1_34 Sch=ar[4]
#set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { ar[5] }]; #IO_L5N_T0_34 Sch=ar[5]
```

```
#set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { ar[6] }]; #IO_L19P_T3_34 Sch=ar[6]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { ar[7] }]; #IO_L9N_T1_DQS_34 Sch=ar[7]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { ar[8] }]; #IO_L21P_T3_DQS_34 Sch=ar[8]
#set_property -dict { PACKAGE_PIN V18   IOSTANDARD LVCMOS33 } [get_ports { ar[9] }]; #IO_L21N_T3_DQS_34 Sch=ar[9]
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { ar[10] }]; #IO_L9P_T1_DQS_34 Sch=ar[10]
#set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { ar[11] }]; #IO_L19N_T3_VREF_34 Sch=ar[11]
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { ar[12] }]; #IO_L23N_T3_34 Sch=ar[12]
#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { ar[13] }]; #IO_L23P_T3_34 Sch=ar[13]
#set_property -dict { PACKAGE_PIN Y13   IOSTANDARD LVCMOS33 } [get_ports { a }]; #IO_L20N_T3_13 Sch=a

##Arduino Digital I/O On Outer Analog Header
##NOTE: These pins should be used when using the analog header signals A0-A5 as digital I/O

#set_property -dict { PACKAGE_PIN Y11   IOSTANDARD LVCMOS33 } [get_ports { a[0] }]; #IO_L18N_T2_13 Sch=a[0]
#set_property -dict { PACKAGE_PIN Y12   IOSTANDARD LVCMOS33 } [get_ports { a[1] }]; #IO_L20P_T3_13 Sch=a[1]
#set_property -dict { PACKAGE_PIN W11   IOSTANDARD LVCMOS33 } [get_ports { a[2] }]; #IO_L18P_T2_13 Sch=a[2]
#set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { a[3] }]; #IO_L21P_T3_DQS_13 Sch=a[3]
#set_property -dict { PACKAGE_PIN T5    IOSTANDARD LVCMOS33 } [get_ports { a[4] }]; #IO_L19P_T3_13 Sch=a[4]
#set_property -dict { PACKAGE_PIN U10   IOSTANDARD LVCMOS33 } [get_ports { a[5] }]; #IO_L12N_T1_MRCC_13 Sch=a[5]

## Arduino SPI

#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports { ck_miso }]; #IO_L10N_T1_34 Sch=miso
#set_property -dict { PACKAGE_PIN T12   IOSTANDARD LVCMOS33 } [get_ports { ck_mosi }]; #IO_L2P_T0_34 Sch=ar_mosi_r
#set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { ck_sck }]; #IO_L19P_T3_35 Sch=sck
#set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports { ck_ss }]; #IO_L6P_T0_35 Sch=ss

## Arduino I2C

#set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports { ar_scl }]; #IO_L24N_T3_34 Sch=ar_scl
#set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { ar_sda }]; #IO_L24P_T3_34 Sch=ar_sda

##Raspberry Digital I/O

#set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports { rpio_02_r }]; #IO_L22P_T3_34 Sch=rpio_02_r
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports { rpio_03_r }]; #IO_L22N_T3_34 Sch=rpio_03_r
#set_property -dict { PACKAGE_PIN Y18   IOSTANDARD LVCMOS33 } [get_ports { rpio_04_r }]; #IO_L17P_T2_34 Sch=rpio_04_r
#set_property -dict { PACKAGE_PIN Y19   IOSTANDARD LVCMOS33 } [get_ports { rpio_05_r }]; #IO_L17N_T2_34 Sch=rpio_05_r
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { rpio_06_r }]; #IO_L22P_T3_13 Sch=rpio_06_r
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports { rpio_07_r }]; #IO_L12P_T1_MRCC_34 Sch=rpio_07_r
#set_property -dict { PACKAGE_PIN F19   IOSTANDARD LVCMOS33 } [get_ports { rpio_08_r }]; #IO_L12N_T1_MRCC_34 Sch=rpio_08_r
#set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { rpio_09_r }]; #IO_L21N_T3_DQS_13 Sch=rpio_09_r
#set_property -dict { PACKAGE_PIN V8    IOSTANDARD LVCMOS33 } [get_ports { rpio_10_r }]; #IO_L15P_T2_DQS_13 Sch=rpio_10_r
#set_property -dict { PACKAGE_PIN W10   IOSTANDARD LVCMOS33 } [get_ports { rpio_11_r }]; #IO_L16P_T2_13 Sch=rpio_11_r
#set_property -dict { PACKAGE_PIN B20   IOSTANDARD LVCMOS33 } [get_ports { rpio_12_r }]; #IO_L1N_T0_AD0N_35 Sch=rpio_12_r
#set_property -dict { PACKAGE_PIN W8    IOSTANDARD LVCMOS33 } [get_ports { rpio_13_r }]; #IO_L15N_T2_DQS_13 Sch=rpio_13_r
#set_property -dict { PACKAGE_PIN V6    IOSTANDARD LVCMOS33 } [get_ports { rpio_14_r }]; #IO_L22P_T3_13 Sch=rpio_14_r
#set_property -dict { PACKAGE_PIN Y6    IOSTANDARD LVCMOS33 } [get_ports { rpio_15_r }]; #IO_L13N_T2_MRCC_13 Sch=rpio_15_r
#set_property -dict { PACKAGE_PIN B19   IOSTANDARD LVCMOS33 } [get_ports { rpio_16_r }]; #IO_L2P_T0_AD8P_35 Sch=rpio_16_r
#set_property -dict { PACKAGE_PIN U7    IOSTANDARD LVCMOS33 } [get_ports { rpio_17_r }]; #IO_L11P_T1_SRCC_13 Sch=rpio_17_r
#set_property -dict { PACKAGE_PIN C20   IOSTANDARD LVCMOS33 } [get_ports { rpio_18_r }]; #IO_L1P_T0_AD0P_35 Sch=rpio_18_r
#set_property -dict { PACKAGE_PIN Y8    IOSTANDARD LVCMOS33 } [get_ports { rpio_19_r }]; #IO_L14N_T2_SRCC_13 Sch=rpio_19_r
#set_property -dict { PACKAGE_PIN A20   IOSTANDARD LVCMOS33 } [get_ports { rpio_20_r }]; #IO_L2N_T0_AD8N_35 Sch=rpio_20_r
#set_property -dict { PACKAGE_PIN Y9    IOSTANDARD LVCMOS33 } [get_ports { rpio_21_r }]; #IO_L14P_T2_SRCC_13 Sch=rpio_21_r
#set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS33 } [get_ports { rpio_22_r }]; #IO_L17N_T2_13 Sch=rpio_22_r
#set_property -dict { PACKAGE_PIN W6    IOSTANDARD LVCMOS33 } [get_ports { rpio_23_r }]; #IO_IO_L22N_T3_13 Sch=rpio_23_r
#set_property -dict { PACKAGE_PIN Y7    IOSTANDARD LVCMOS33 } [get_ports { rpio_24_r }]; #IO_L13P_T2_MRCC_13 Sch=rpio_24_r
#set_property -dict { PACKAGE_PIN F20   IOSTANDARD LVCMOS33 } [get_ports { rpio_25_r }]; #IO_L15N_T2_DQS_AD12N_35 Sch=rpio_25_r
#set_property -dict { PACKAGE_PIN W9    IOSTANDARD LVCMOS33 } [get_ports { rpio_26_r }]; #IO_L16N_T2_13 Sch=rpio_26_r
#set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports { rpio_sd_r }]; #IO_L7P_T1_34 Sch=rpio_sd_r
#set_property -dict { PACKAGE_PIN Y17   IOSTANDARD LVCMOS33 } [get_ports { rpio_sc_r }]; #IO_L7N_T1_34 Sch=rpio_sc_r

##HDMI Rx

#set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { hdmi_rx_cec }]; #IO_L13N_T2_MRCC_35 Sch=hdmi_rx_cec
#set_property -dict { PACKAGE_PIN P19   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_clk_n }]; #IO_L13N_T2_MRCC_34 Sch=hdmi_rx_clk_n
#set_property -dict { PACKAGE_PIN N18   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_clk_p }]; #IO_L13P_T2_MRCC_34 Sch=hdmi_rx_clk_p
#set_property -dict { PACKAGE_PIN W20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_n[0] }]; #IO_L16N_T2_34 Sch=hdmi_rx_d_n[0]
#set_property -dict { PACKAGE_PIN V20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_p[0] }]; #IO_L16P_T2_34 Sch=hdmi_rx_d_p[0]
#set_property -dict { PACKAGE_PIN U20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_n[1] }]; #IO_L15N_T2_DQS_34 Sch=hdmi_rx_d_n[1]
#set_property -dict { PACKAGE_PIN T20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_p[1] }]; #IO_L15P_T2_DQS_34 Sch=hdmi_rx_d_p[1]
#set_property -dict { PACKAGE_PIN P20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_n[2] }]; #IO_L14N_T2_SRCC_34 Sch=hdmi_rx_d_n[2]
#set_property -dict { PACKAGE_PIN N20   IOSTANDARD TMDS_33  } [get_ports { hdmi_rx_d_p[2] }]; #IO_L14P_T2_SRCC_34 Sch=hdmi_rx_d_p[2]
#set_property -dict { PACKAGE_PIN T19   IOSTANDARD LVCMOS33 } [get_ports { hdmi_rx_hpd }]; #IO_25_34 Sch=hdmi_rx_hpd
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { hdmi_rx_scl }]; #IO_L11P_T1_SRCC_34 Sch=hdmi_rx_scl
#set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports { hdmi_rx_sda }]; #IO_L11N_T1_SRCC_34 Sch=hdmi_rx_sda

##HDMI Tx

#set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [get_ports { hdmi_tx_cec }]; #IO_L19N_T3_VREF_35 Sch=hdmi_tx_cec
#set_property -dict { PACKAGE_PIN L17   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_clk_n }]; #IO_L11N_T1_SRCC_35 Sch=hdmi_tx_clk_n
#set_property -dict { PACKAGE_PIN L16   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_clk_p }]; #IO_L11P_T1_SRCC_35 Sch=hdmi_tx_clk_p
#set_property -dict { PACKAGE_PIN K18   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_n[0] }]; #IO_L12N_T1_MRCC_35 Sch=hdmi_tx_d_n[0]
#set_property -dict { PACKAGE_PIN K17   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_p[0] }]; #IO_L12P_T1_MRCC_35 Sch=hdmi_tx_d_p[0]
#set_property -dict { PACKAGE_PIN J19   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_n[1] }]; #IO_L10N_T1_AD11N_35 Sch=hdmi_tx_d_n[1]
#set_property -dict { PACKAGE_PIN K19   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_p[1] }]; #IO_L10P_T1_AD11P_35 Sch=hdmi_tx_d_p[1]
#set_property -dict { PACKAGE_PIN H18   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_n[2] }]; #IO_L14N_T2_AD4N_SRCC_35 Sch=hdmi_tx_d_n[2]
#set_property -dict { PACKAGE_PIN J18   IOSTANDARD TMDS_33  } [get_ports { hdmi_tx_d_p[2] }]; #IO_L14P_T2_AD4P_SRCC_35 Sch=hdmi_tx_d_p[2]
#set_property -dict { PACKAGE_PIN R19   IOSTANDARD LVCMOS33 } [get_ports { hdmi_tx_hpdn }]; #IO_0_34 Sch=hdmi_tx_hpdn


##Crypto SDA

#set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { crypto_sda }]; #IO_25_35 Sch=crypto_sda
```

appendix/constraints.xdc