

ECE532S Digital Systems Design

Tutorial 6 - Creating and Packaging a Custom IP

Last Updated: July, 2020

In previous tutorials we've used IP cores provided by Xilinx to implement various functionalities. In this tutorial, we'll go through the project flow for creating your own IP cores. In particular, we will be creating an IP core that uses an AXI interface, such that our core can be controlled by a MicroBlaze system.

1 Creating the AXI IP

Launch Vivado and select **Manage IP...** → **New IP Location** under the *Tasks* section of the Vivado Launch screen. Press **Next** and then select the part number that corresponds with the FPGA on the Nexsys FPGA, the **xc7a100tcsg324-1**. In the *IP Location* field, select the folder to create a new IP repository. From the window toolbar, select **Tools** → **Create and Package IP...**, which will bring up the *Create New IP Wizard*. Press **Next** and then select the **Create a new AXI4 peripheral** option, as shown in Figure 1. Press **Next**.

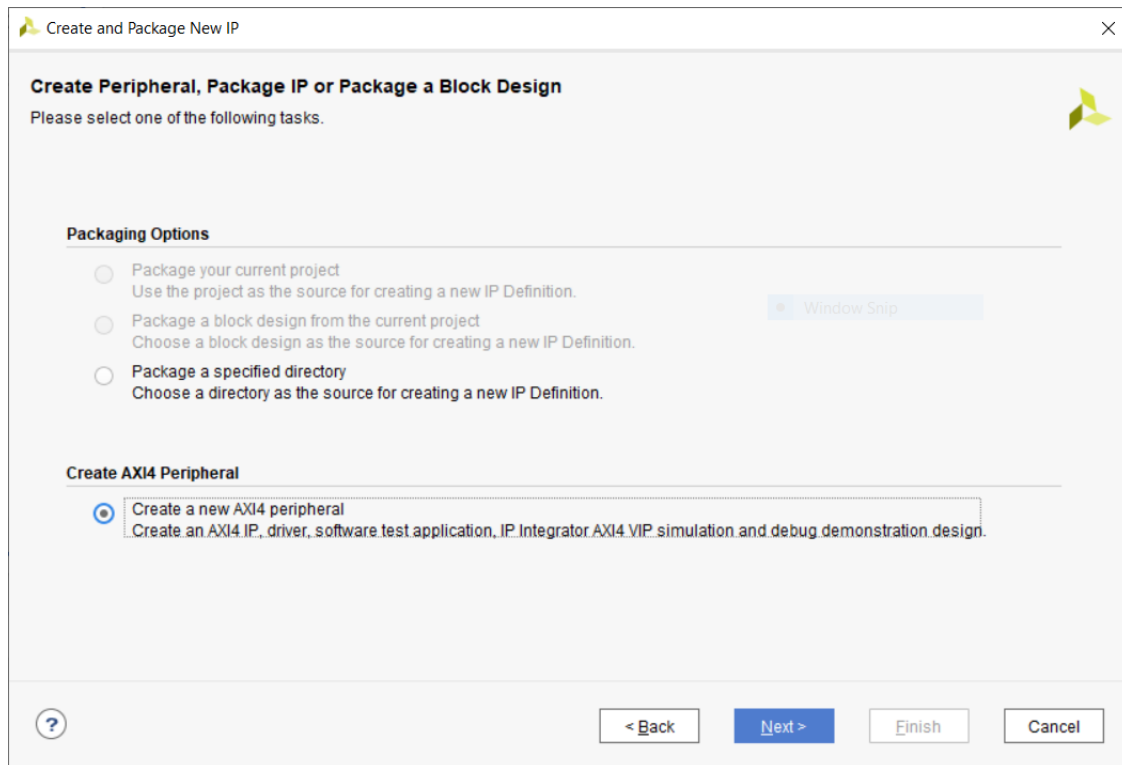


Figure 1: Create new IP Wizard, selecting AXI4 peripheral

In the next window, we can select a name and version number for our new IP, we'll leave these settings at the default value. The *IP Location* field determines where the newly created IP will be stored, and by default it's location will be within a subfolder titled *ip_repo* in the directory selected at the beginning of this section. Click **Next** to select the parameters of the new AXI IP. This *AXI Interfaces* pane, shown in Figure 2, allows you to choose which and the number of AXI interfaces to include in the AXI IP. The middle section of this window lists all of the interfaces currently added, and allows for more interfaces to be added. The right section of this window shows the configuration for the selected AXI Interfaces.

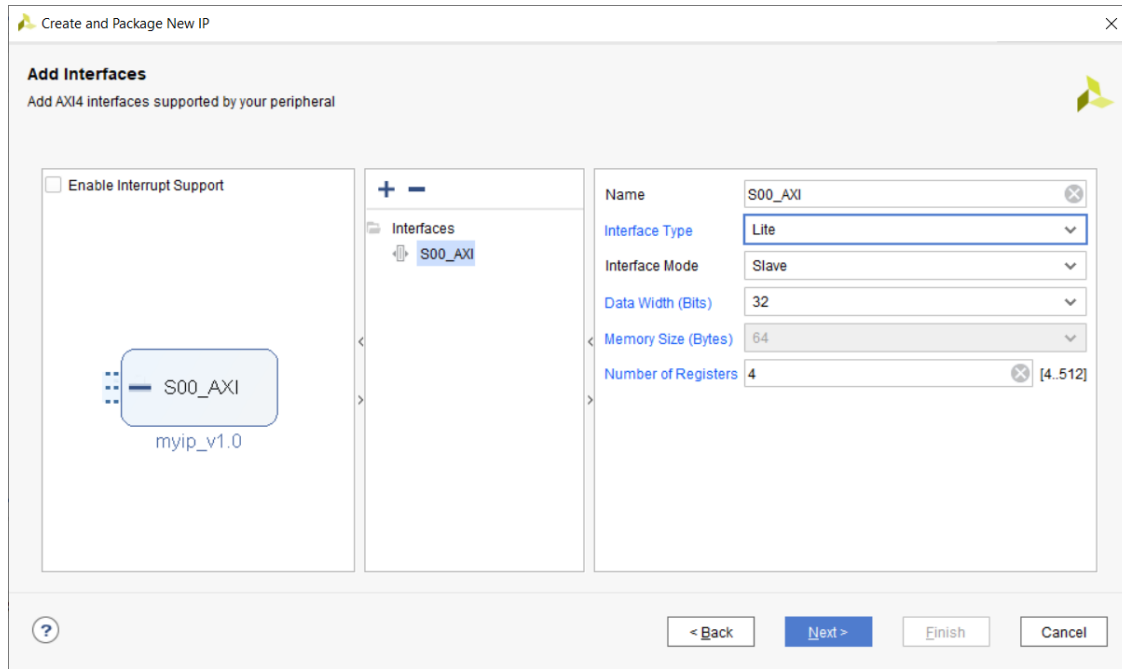


Figure 2: Create new IP Wizard, AXI interfaces selection

The first option of note is the *Interface Type* field. The AXI4 standard defines 3 main types of interfaces, the AXI4-full, the AXI4-lite, and the AXI-Stream. AXI4-full and AXI4-Lite are both memory mapped interfaces that can be accessed directly from a MicroBlaze. The AXI4-full interface supports burst accesses, and as such is generally used for high bandwidth interfaces. The AXI4-lite interface is simpler and generally used for peripherals that have a small number of registers to be accessed from the MicroBlaze. The AXI-Stream interface is generally used to encapsulate streaming data, such as network streams of video streams. For the purposes of this tutorial, we'll be creating an AXI-lite interface, so select **Lite** for the *Interface Type*, select **Slave** for the *Interface Mode* (i.e. the IP core is read/written to, rather than initiating reads and writes itself), and set the *Number of Registers* to **4**. Press **Next** and then select **Edit IP** under *Next Steps* and press **Finish**.

This will open a new Vivado project for the IP. Under sources you will see both the Verilog source and a component.xml that describes your IP in IP-XACT format that the IP Integrator can understand. There are two sources, a top level *myip_v1_0.v* and an interface module *my_ip_v1_0_S00_AXI*. For this design the top-level module just wraps the *my_ip_v1_0_S00_AXI* module (see Figure 3).



Figure 3: AXI IP file hierarchy

2 Understanding the AXI Lite Protocol

The template core implements a simple AXI lite slave interface and here we will describe a few basics of the AXI interface. The AXI protocol consists of five channels (the Verilog signals are prefixed with the parenthesized characters):

- Read Address (AR)
- Read Data (R)
- Write Address (AW)
- Write Data (W)
- Write Response (B)

In general all channels communicate using READY/VALID handshaking. Only when the sender asserts a valid signal and the receiver asserts a ready signal can data flow.

2.1 Write Operations

Write operations use the write address, write data and write response channels. Since this core is a slave, the address and data are inputs given by the master telling the slave where and what to write. The response channel is an output for acknowledging the transfer. Look for the assignment of the AWREADY, WREADY and AWADDR (write address data) signals. This implementation sets the ready signals high and gets the address when both address and data channels are valid. In the next cycle, the data to be written (WDATA) is stored in the addressed slave register and BVALID is asserted.

2.2 Read Operations

Read operations only use an address and data channel. Like the write channel, the address is registered one cycle before the data is transferred. However, in this case the data is an output from

our slave containing the value of the addressed register. The logic for read operations is below the write operations in the template Verilog implementation. You are encouraged to examine the code more closely to more fully understand the functionality.

3 Modifying the AXI IP

The skeleton Verilog code handles reading and writing the four slave registers. We will now insert an adder to produce a sum output of our four slave registers. There are two main areas that we generally modify within this skeleton code, and each is annotated as such with comments. First, the section that appears as follows:

```
// Users to add ports here  
  
// User ports ends  
// Do not modify the ports beyond this line
```

We want to add a single output port that carries the sum of all of our registers within our AXI-Lite IP. Modify this section of code to appear as follows:

```
// Users to add ports here  
output [7:0] sum,  
// User ports ends  
// Do not modify the ports beyond this line
```

In future custom IP designs, you could add any number of signals here that you would like to appear on the interface of the IP core. The next section of that we can modify appears as follows later in the code:

```
// Add user logic here  
  
// User logic ends
```

We will add logic here that determines the sum from our register values. Modify this section of code to appear as follows:

```
// Add user logic here  
assign sum = slv_reg0[7:0] + slv_reg1[7:0] + slv_reg2[7:0] + slv_reg3[7:0];  
// User logic ends
```

Now that we have the port added and the logic to implement our required functionality, we need to modify the wrapper verilog file to carry this signal to the top level of our design. The top level file, titled *myip_v1_0.v*, also contains an area to add ports:

```

// Users to add ports here

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


```


Modify it to add a sum port just as we did with the *S00_AXI* module. Next, we have to connect this signal in the top level wrapper to the instance of the *S00_AXI* module. Change the instantiation of the inner module such that it appears as follows:

```

myip_v1_0_S00_AXI # (
    .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
    .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
) myip_v1_0_S00_AXI_inst (
    .sum(sum), // We added this connection
    .S_AXI_ACLK(s00_axi_aclk),
    .S_AXI_ARESETN(s00_axi_aresetn),
    .S_AXI_AWADDR(s00_axi_awaddr),
    .S_AXI_AWPROT(s00_axi_awprot),
    .S_AXI_AWVALID(s00_axi_awvalid),
    .S_AXI_AWREADY(s00_axi_awready),
    .S_AXI_WDATA(s00_axi_wdata),
    .S_AXI_WSTRB(s00_axi_wstrb),
    .S_AXI_WVALID(s00_axi_wvalid),
    .S_AXI_WREADY(s00_axi_wready),
    .S_AXI_BRESP(s00_axi_bresp),
    .S_AXI_BVALID(s00_axi_bvalid),
    .S_AXI_BREADY(s00_axi_bready),
    .S_AXI_ARADDR(s00_axi_araddr),
    .S_AXI_ARPROT(s00_axi_arprot),
    .S_AXI_ARVALID(s00_axi_arvalid),
    .S_AXI_ARREADY(s00_axi_arready),
    .S_AXI_RDATA(s00_axi_rdata),
    .S_AXI_RRESP(s00_axi_rresp),
    .S_AXI_RVALID(s00_axi_rvalid),
    .S_AXI_RREADY(s00_axi_rready)
);

```

Note that the only modification we've made is that we've added the connection to the *sum* port. Now that we've implemented the functionality of our custom core within the skeleton code, we can proceed to package our IP. **Save** all your sources and open the *Package IP* tab. When we started creating this IP, all of the steps in the *Package IP* tab displayed a  icon to indicate that the step

is complete. Now that we've changed some of the source files, a  icon indicates that the step needs to be updated based on the changes. Open the first packaging step that needs to be updated (it should be *File Groups*, and you should notice a *Merge changes from File Groups Wizard* prompt, as in Figure 4. Click **Merge changes from File Groups Wizard** and the step should transition to complete. Repeat this for each of the *Packaging Steps*. Note, the *Merge changes* wizard may complete multiple steps at a times.

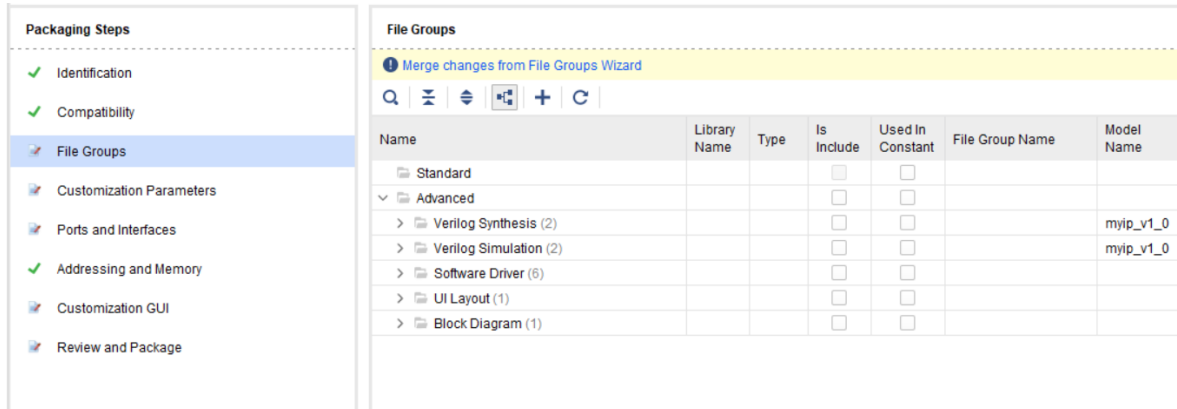


Figure 4: IP packaging steps, with merge changes wizard prompt

While we don't need to make any changes from the defaults selected by the wizard, it is still important to know what the purpose of each of these *Packaging Steps* is. The *Identification* step allow you to enter the name, version, and description for your core. Here you will generally update the version with every update. The *Compatibility* step allows you to select what FPGAs the IP is compatible with; you'll only need to modify this if your Verilog includes modules that in themselves are not compatible with all Xilinx FPGAs (usually these are device primitives). The *File Groups* step lists all of the files included in the IP, and is where you would select to add any newly created verilog files to the IP. The *Customization Parameters* lists all of the Verilog parameters within your top level module; you can change settings for them here. The *Ports and Interfaces* should list all of the top level ports of your design; check to make sure the *sum* port was added as an output. The *Addressing and Memory* step lists all memory regions of your IP Core. The *Customization GUI* step allows you to change what the GUI looks like for configuring your core. Finally, select the **Review and Package** step to finish the packaging process. Press **Re-Package IP** to package the IP with all of the changes we made. You can now close the IP project.

4 Using the Custom IP

Open or recreate the project from tutorial 3 (create a copy of the project directory if you want to preserve the version of the project for reference later). Open the settings window from **Tools** → **Settings...** and expand the *IP* heading in the left pane. Select **Repository** to bring up the *IP Repository* settings in the right pane. Press the **+** button to add a new IP repository to the project. Navigate to the *ip_repo* subfolder of the directory you selected in the first step of this tutorial, and press **Select**. The window that comes up lists all IP found in that repository; you should see the new custom IP we created listed. See Figure 5 for reference.

Open the block diagram in your project. In order to simplify the project, go ahead and **delete** the *MIG*, as well as the *Processor System Reset* connected to the *MIG*. If your project had an *ILA* that

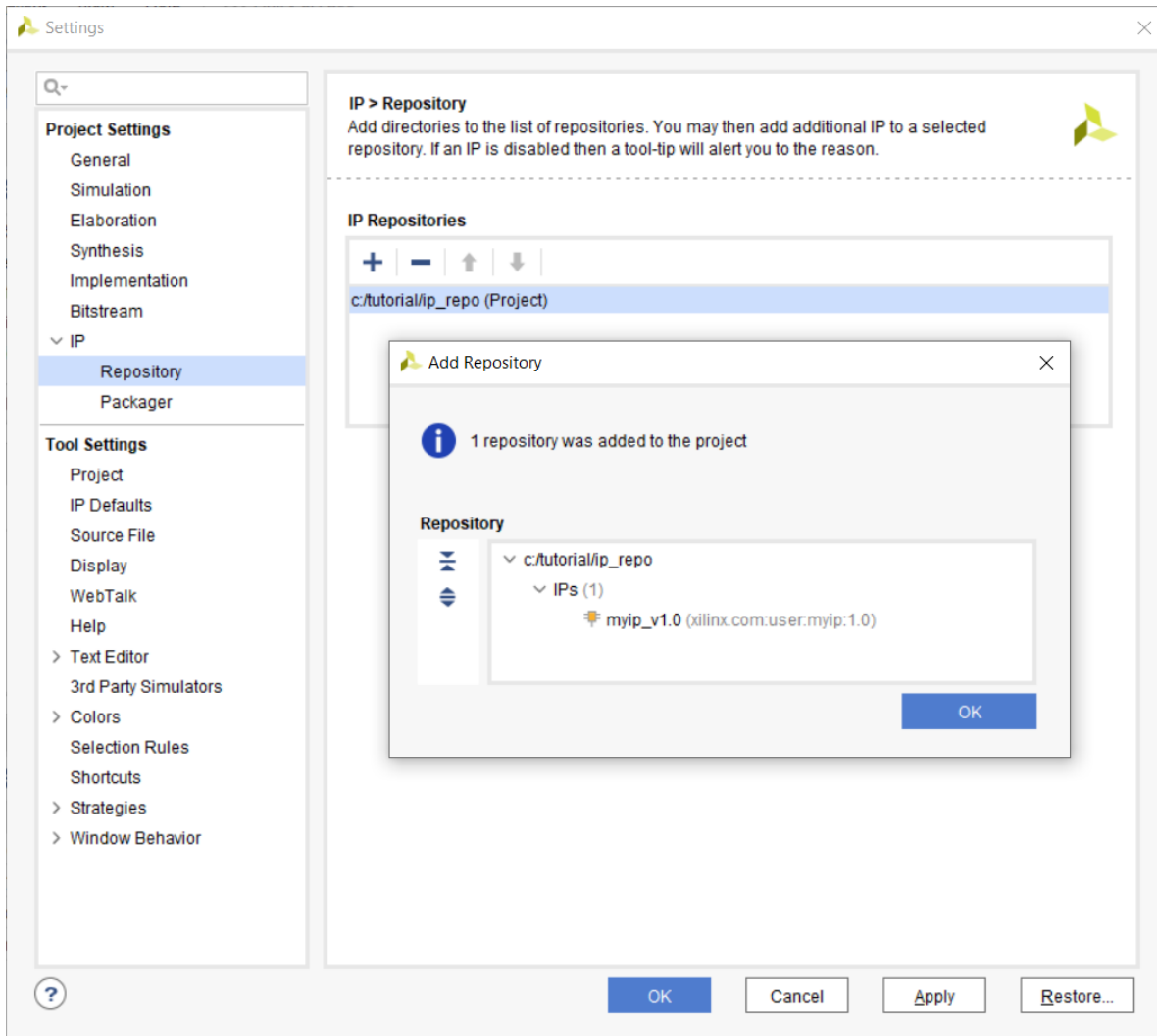


Figure 5: IPs listed after adding an IP repository

was connected to the *MIG*, you can **delete** that core as well. Now **right click** anywhere in the block diagram window and select **Add IP...** Search for the IP core we just created, titled *myip_v1.0*, and add it to your project. Click **Run Connection Automation** in order to automatically connect the AXI signal on our custom IP Core to the AXI interconnect. **Right click** anywhere in the editor window and select **Create Port**. Make a port called *led*, set as **Output**, as a vector of size 7 down to 0. Connect this newly created port to the sum port of our core. You should now have an IP core in the project connected as per Figure 6.

We need to add a constraint file to our project to connect this new output value to the leds. Close the block diagram view. **Right click** the block diagram in the source window and select **Create HDL Wrapper** to regenerate the block diagram wrapper. Open the verilog wrapper and verify that the *led* signal has been added as an output of the top level module. Note, we usually check the wrapper since the name of the signal in the wrapper doesn't always exactly match the name we gave to the signal in the block diagram view. Add the *tutorial.xdc* constraint file found in this tutorial's zip folder with the contents of Figure 7 to the project, making sure the signal names match those in the wrapper.

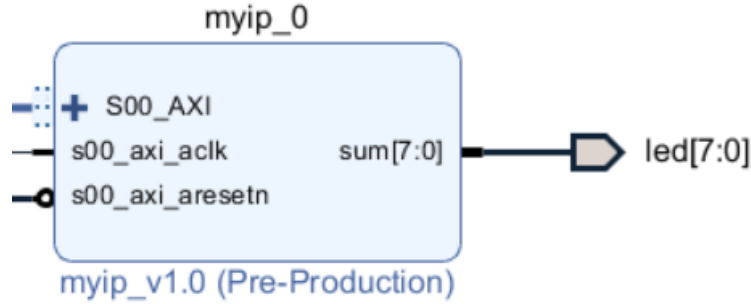


Figure 6: The custom IP core connected within IP Integrator

```

1  set_property PACKAGE_PIN H17 [get_ports led[0]]
2  set_property IOSTANDARD LVCMOS33 [get_ports led[0]]
3  set_property PACKAGE_PIN K15 [get_ports led[1]]
4  set_property IOSTANDARD LVCMOS33 [get_ports led[1]]
5  set_property PACKAGE_PIN J13 [get_ports led[2]]
6  set_property IOSTANDARD LVCMOS33 [get_ports led[2]]
7  set_property PACKAGE_PIN N14 [get_ports led[3]]
8  set_property IOSTANDARD LVCMOS33 [get_ports led[3]]
9  set_property PACKAGE_PIN R18 [get_ports led[4]]
10 set_property IOSTANDARD LVCMOS33 [get_ports led[4]]
11 set_property PACKAGE_PIN V17 [get_ports led[5]]
12 set_property IOSTANDARD LVCMOS33 [get_ports led[5]]
13 set_property PACKAGE_PIN U17 [get_ports led[6]]
14 set_property IOSTANDARD LVCMOS33 [get_ports led[6]]
15 set_property PACKAGE_PIN U16 [get_ports led[7]]
16 set_property IOSTANDARD LVCMOS33 [get_ports led[7]]

```

Figure 7: Contents of an XDC file to set pin constraints for the LEDs

Run **Synthesis**, **Implementation**, **Generate Bitstream**, and **Export Hardware**, and then launch **Vivado SDK**. Open the *helloworld* application and modify the contents of *helloworld.c* to the contents of Figure 8. This is a very simple application that just writes random values to the different registers of our core. The output value produced will be the sum of these values, which is 87, or 01010111. Before running our project, remember to open one of the board support package include files, such as *xil_printf.h*, and press **F5** to refresh them (as we did in tutorial 4). Open the *Run Configurations* windows and ensure the **Program FPGA** check box is checked and press **Apply** (we unchecked it in tutorial 4). Now press **Run** to run the application on the FPGA. Verify the functionality is as expected on the leds.

When we created and packaged the custom IP, Xilinx also packaged a simple *low level driver* for the IP. Open the *myip.h* file in the *include* folder of the board support package. Examine the contents to see how to use the driver. Now modify the *helloworld.c* file to use the *low level driver*, as shown in Figure 9. Run this modified application and verify that we get the same expected output.

```

1  #include <stdio.h>
2  #include "platform.h"
3  #include "xil_printf.h"
4  #include "xparameters.h"
5
6  volatile unsigned int* myip_base = (unsigned int*) XPAR_MYIP_0_S00_AXI_BASEADDR;
7
8  int main()
9  {
10     init_platform();
11
12     print("Writing to Registers\n\r");
13     *(myip_base+0) = 7;
14     *(myip_base+1) = 12;
15     *(myip_base+2) = 67;
16     *(myip_base+3) = 1;
17
18     cleanup_platform();
19     return 0;
20 }

```

Figure 8: Modified Hello World application using custom IP core

5 Simulating the System

We need to be able to simulate our new AXI IP in order to effectively debug it. AXI interfaces can be difficult to simulate since the AXI protocol contains a number of signals, all of which must be varied together to create the proper read and write transactions. There are two ways to ease this simulation: using specially built simulation cores, and simulating the MicroBlaze system as a whole. An AXI simulation core will be covered in a future tutorial, here we will simulate the entire MicroBlaze system we just generated. Note, simulating the entire MicroBlaze system is a crude way to run a simulation for our AXI core, since this introduces the overhead of simulation the MicroBlaze processor, but it is a quick way to create a simulation that's easy to understand. In general, we prefer the simulation methods using simulation cores presented in the later tutorials.

Close the Vivado SDK and return to the Vivado project window. To simulate our MicroBlaze with the application we just wrote in *Vivado SDK*, we need to associate the ELF file (the compiled executable) with the simulation sources. Open **Tools** → **Associate Elf Files...** to bring up the *Associate ELF Files* window. Under *simulation sources*, press the ☐ button beside the *microblaze* component. In the new window click **Add Files...** and browse to your SDK folder. The SDK folder should be located within your Vivado project folder, with a name generally appended with *.sdk*. The ELF file should be located at a path similar to:

```
<project name>.sdk/<application>/Debug/<application>.elf
```

where *<application>* is the name of the *helloworld* application you created in the *Vivado SDK*. **Select** the ELF file and press **OK**. Make sure the newly added ELF file is highlighted and press **OK** again. You should now see the new elf file associated with the microblaze under simulation sources, as in Figure 10. Press **OK**.

```

1  #include <stdio.h>
2  #include "platform.h"
3  #include "xil_printf.h"
4  #include "xparameters.h"
5  #include "xil_io.h"
6  #include "myip.h"
7
8  int main()
9  {
10     init_platform();
11
12     print("Writing to Registers\n\r");
13     MYIP_mWriteReg(XPAR_MYIP_0_S00_AXI_BASEADDR, MYIP_S00_AXI_SLV_REG0_OFFSET, 7);
14     MYIP_mWriteReg(XPAR_MYIP_0_S00_AXI_BASEADDR, MYIP_S00_AXI_SLV_REG1_OFFSET, 12);
15     MYIP_mWriteReg(XPAR_MYIP_0_S00_AXI_BASEADDR, MYIP_S00_AXI_SLV_REG2_OFFSET, 67);
16     MYIP_mWriteReg(XPAR_MYIP_0_S00_AXI_BASEADDR, MYIP_S00_AXI_SLV_REG3_OFFSET, 1);
17
18     cleanup_platform();
19     return 0;
20 }

```

Figure 9: Modified Hello World application using custom IP core, with low-level driver

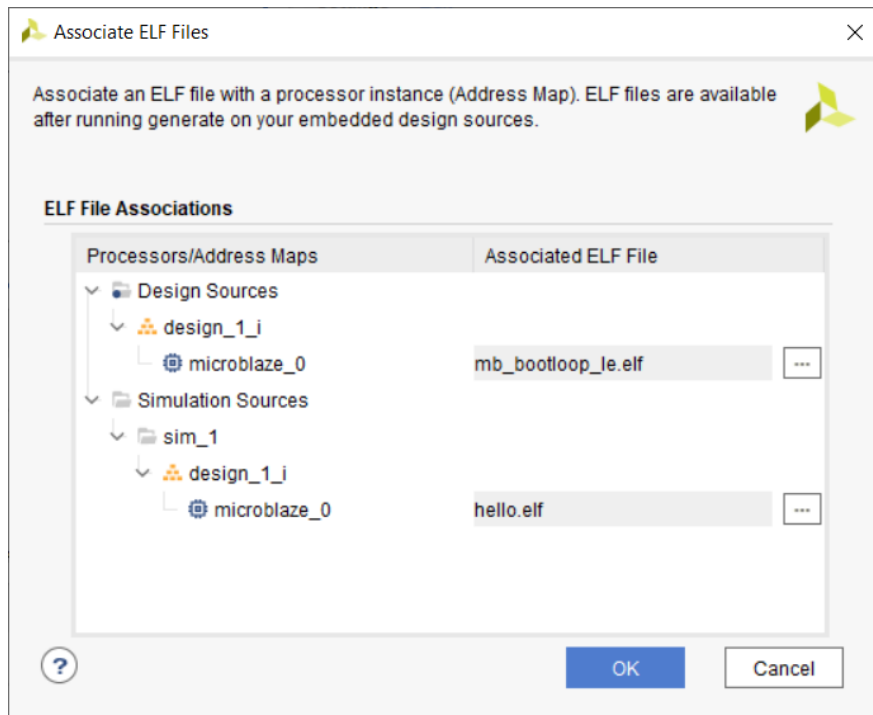


Figure 10: ELF file associated with the simulation microblaze

Now we simply need to create a testbench that can drive the inputs of our top level module, the block diagram wrapper. Add the *tutorial_tb.v* file (the contents of which are shown in Figure 11) found in this tutorial's zip folder as a simulation source to your project. Make sure the instantiation of the HDL wrapper matches the one from your project. This testbench simply connects wires to each of the top level ports of the HDL wrapper, and create a valid reset and clock signal. The reset signal is asserted for the first 40ns and the clock pulses with a period of 4ns. Note, we hardwired the *usb_uart_rxd* to zero as we don't need to test the uart signals but we cannot leave input signals unconnected.

```

1  `timescale 1ns / 1ps
2
3  module tb();
4
5      //Wires to connect to DUT
6      reg clock, reset;
7      wire usb_uart_rxd, usb_uart_txd;
8      wire [7:0] led;
9
10     //DUT
11     design_1_wrapper dut(
12         .sys_clock(clock),
13         .reset(reset),
14         .usb_uart_rxd(usb_uart_rxd),
15         .usb_uart_txd(usb_uart_txd),
16         .led(led)
17     );
18
19     assign usb_uart_rxd = 0;
20
21     //Reset signal
22     initial begin
23         reset = 0;
24         #40 reset = 1;
25     end
26
27     //Clock signal
28     initial clock = 0;
29     always begin
30         #2 clock = ~clock;
31     end
32
33 endmodule

```

Figure 11: Testbench to simulate the system

Run the behavioural simulation, select **Run Simulation** → **Run Behavioural Simulation** from the *Simulation* heading in the *Flow Navigator*. Once the behavioural simulation window opens, it will have simulated the design for some default amount of time (likely 500ns). This amount of

time is not nearly enough time for the MicroBlaze to do some useful work, so we will have to run the simulation for a longer time. In any case, we want to add some signals from our custom IP first before proceeding. In the *Scope* tab, expand the hierarchy until you find the lowest level module of the *myip* core. Now, in the *Objects* tab, find the four slave registers and drag them to your waveform window. See Figure 12 for reference. Also add the *sum* signals from the *Objects* tab to the waveform viewer.

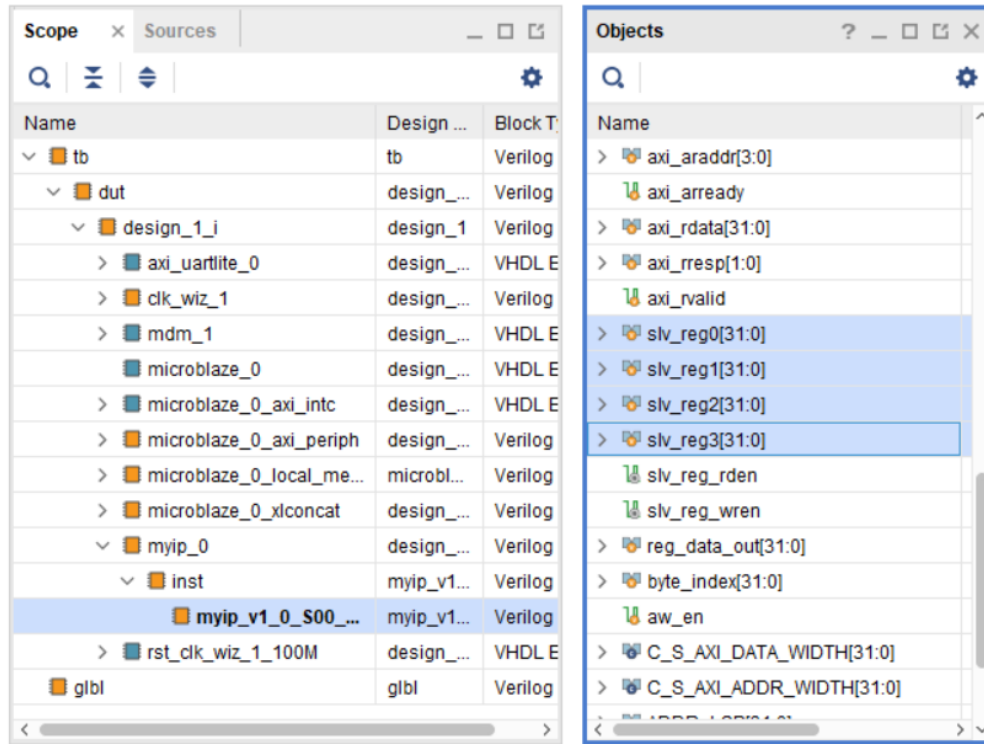


Figure 12: The register signals within the AXI core

Once you've added the signals to the waveform viewer, proceed to run the simulation for 3 ms (note, milliseconds, not nanoseconds, use the drop down menu to select ms). Here you will notice the major downside of performing full system simulation, the simulation time can be incredibly long. This simulation may take up to a couple minutes to complete, but yet more complex systems or applications could take even longer to complete the simulation. This is because the processor is simulated at the RTL level. Once the simulation is complete, you should see an output like Figure 13. Note, the radix for the registers and sum signal is set to *Unsigned Decimal* in the figure.

We notice that the write to our AXI registers are not made until around the 2.5ms point, which is over 600,000 cycles of the clock. In the meantime, the system seems to be trying to transmit something over the UART signal, likely the print statement from the application. Zoom in on the section where the register values are written to, you should now notice the gap between the successive write. This larger gap is because AXI-Lite doesn't support burst transfers, and as such each individual transfer takes longer to initiate. Here we also note that the sum value changes with each write to a register, which is something we wouldn't be able to see on the LEDs when testing on the physical board.

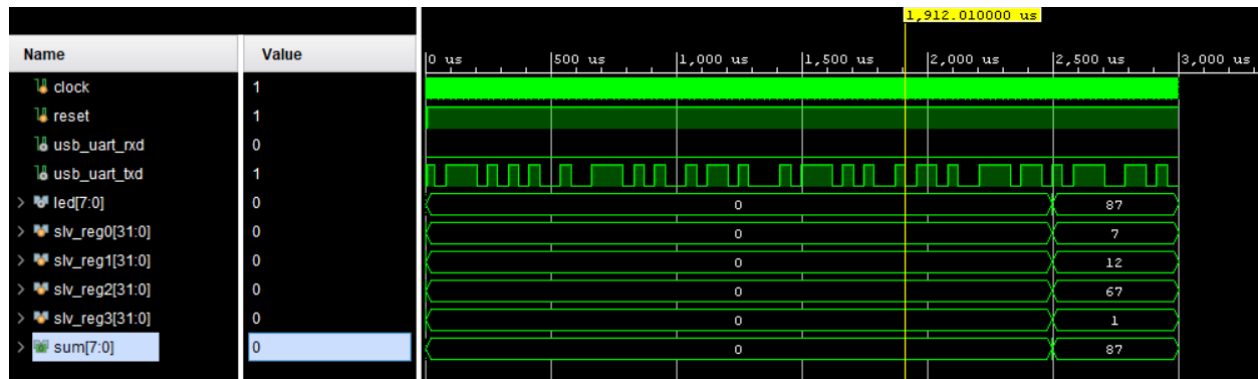


Figure 13: Simulation waveform of the system simulation

6 Summary

The Vivado flow for creating a custom IP allows us to wrap our own HDL cores such that they can be included in Vivado projects through the block diagram editor. This flow even creates skeleton code with all of the functionality for any of the AXI protocols you'd like to include implemented for you. Note, the IP packaging flow demonstrated in this lab created a core with AXI interfaces and corresponding generated skeleton code, though the IP packager can even be used with cores that have no AXI functionality. The IP core we created was instantiated in the block diagram from a previous tutorial, and we interfaced with it through a MicroBlaze program and system simulation.