# Creating AXI-LITE ‘Custom IP’ in Vivado

Lab for COMP4601

Developed by: Shivam Garg

# Contents

# 1. Introduction

The aim of this lab is to introduce a design flow that allows you to create your own custom Intellectual Property (Custom IP) targeted at a Zynq device using Xilinx's Vivado 2013.4. The lab has been created for senior undergraduates using the ZedBoard. We assume the reader is familiar with the use of VHDL for specifying hardware. The lab explains how to modify the generated component, by focusing on how the AXI-LITE protocol works and how it can be utilised to establish a two-way data flow between the Processing System (PS) and the hardware component implemented in programmable logic (PL). This lab concludes on methods for maintaining and integrating this IP as part of a larger design.

As a high-level overview the sections (numbered) of this document will cover the following:

2. Setting up your Vivado high-level design, focussing on the configuration of the Processing System.
3. Using Vivado's built-in tools to generate your own 'Custom IP', and showing you how to start modifying the IP.
4. A tutorial on the AXI protocol that explains the critical modifications that allows the Slave AXI implementation to be abstracted, thereby leaves you to concentrate on implementing hardware-based solutions, and making the AXI communication process as simple as possible.
5. Shows you how to package and upgrade your IP, presenting you with software implementations to test some basic modifications to your hardware.
6. The penultimate section of this documentation consists of a series of implementation exercises designed to get you comfortable with using the Custom IP, and to familiarise you with alternative ways of interfacing with AXI.
7. Concluding remarks for this lab, listing methods of going forward and developing your own hardware based solutions.

By the end of this lab you should be able to generate your own components quickly, implement hardware solutions, and effectively utilise the AXI bus to get data to and from the PS to the PL. You'll also become proficient in the flow of developing hardware within the Vivado framework, and learn methods of debugging and turning out that hardware solutions as soon as possible.

## 2. High-level design configuration

Firstly you'll need to configure a high-level design that features a ZynQ7 processing core. For detailed instructions on the Vivado design flow please refer to lab1 of the Xilinx Advanced Embedded Design course, which teaches you how to work with Vivado, and design hardware on a high-level basis (using IP blocks). In this lab we will be replacing standard Xilinx AXI based IP with our own Custom IP components, which will be configured to provide some common hardware implementations including a timer, FIFO and GPIO (See Section 6).

The starting point for this lab is the following high-level configuration, (represented diagrammatically in Figure 2.1):

- Instantiated a Zynq7 processing system (which has UART1 enabled)
- Applied block automation (without board pre-sets applied) to auto connect DDR and FIXED_IO to external pins
- Opened up this design on the "Open block diagram" to show the following

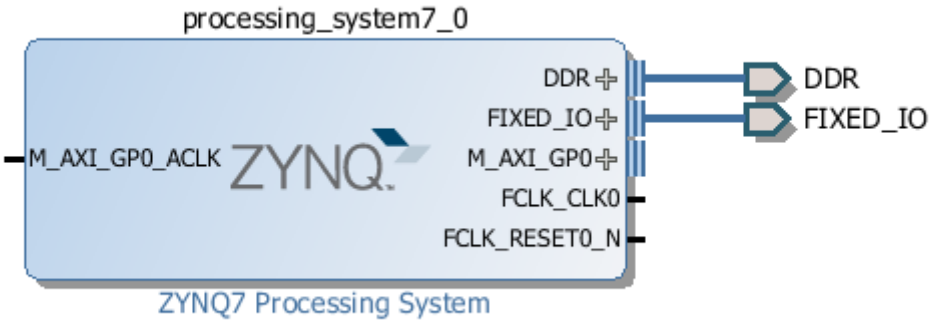



*Figure 2.1: Initial design*

# 3. Creating Custom IP

In this section we will be creating our own Custom IP which features the AXI-LITE interface, so that we can bridge between the Processing System (PS) and the Programmable Logic (PL). We will then connect this IP to the PS and prepare a project file so that you can readily modify the design later.

## 3.a Generating a Custom IP component

3.1. Click on **Project Settings**, then ensure that the Target Language is set to **VHDL** (else the generated IP will be in Verilog), click OK when done.

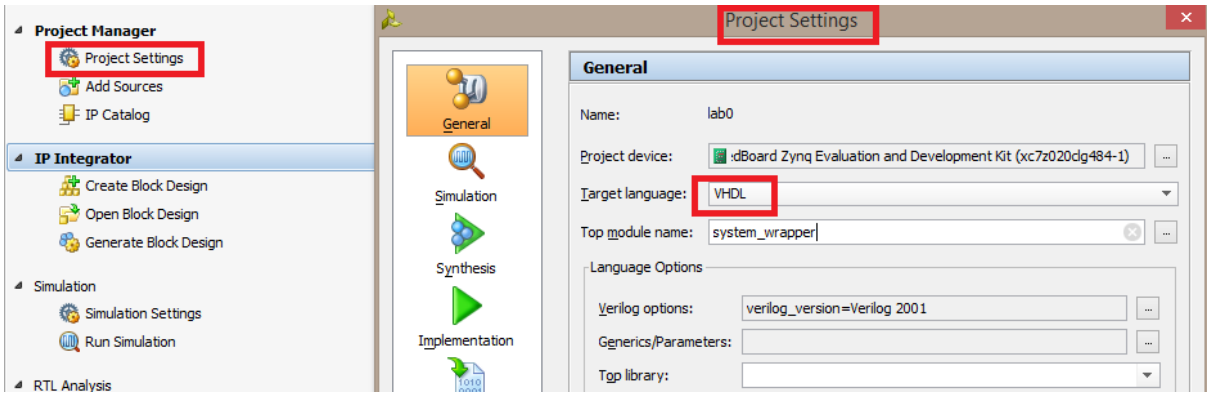



*Figure 3.1: Step 3.1*

3.2. Go to the **tools menu** > "Create and Package IP"

3.3. On the introductory screen, select the **next** option.

3.4. Select "**Create new AXI4 peripheral**" and then in the IP location, go up one level in the directory hierarchy from where your high-level project file is located, so your high-level project directory and the IP that we will create will be located in the same directory (e.g. C:/…./XX/high-level & C:/…/XX/IP)

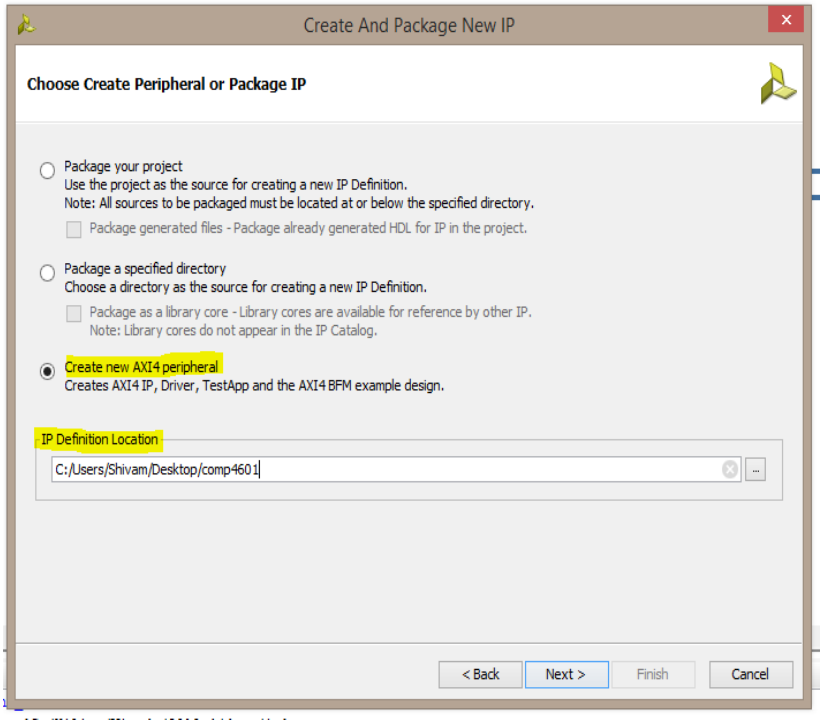



*Figure 3.2: Step 3.4, selecting IP Type/location*

3.5. Now name the IP as "**lab0_ip**", updating the display name accordingly, as well as adding a more relevant description.

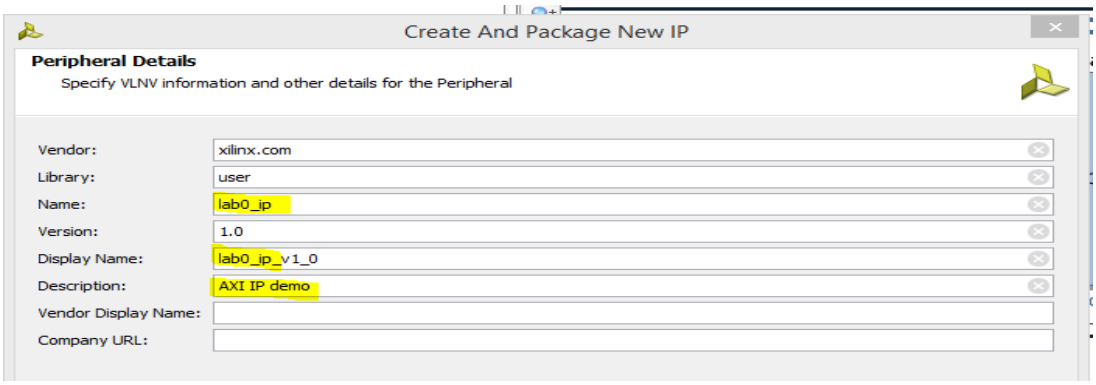



*Figure 3.3: Step 3.5, naming your IP*

3.6. For the next menu we'll keep the **default** options selected, here is an explanation as to why those options were selected:

- **Interface Type (LITE)** – LITE is the simplest of the three AXI protocols to program for, Full AXI allows for burst (4 packets at a time) transfer, while Stream offers continual data transfer. However both Full & Stream are not easily programmed for on the PS, both featuring complex software designs as precursors to interfacing with the IP, hence we will stick to the simpler LITE interface type.
- **Interface Mode (SLAVE)** – Since this IP is going to be issued commands by the processor this IP will act as a Slave.
- **Data Width (32)** – Again for simplicity, we'll keep the bus at the default width.
- **Number of registers (4)** – This option will affect the generated Slave AXI code, with four registers the data transferred from Master to Slave will be stored in 4 unique registers, with the 4 lower address bits act as a multiplexing address (b0000 first register, b0100 second register, b1000 third register and b1100 fourth register, the last two bits are "00" for byte alignment). Section 5.c will show the effect of only having 4 bits available to the Slave, but for now just note that the Slave will only be able to see the lower 4 bits of the AXI address for each transaction.

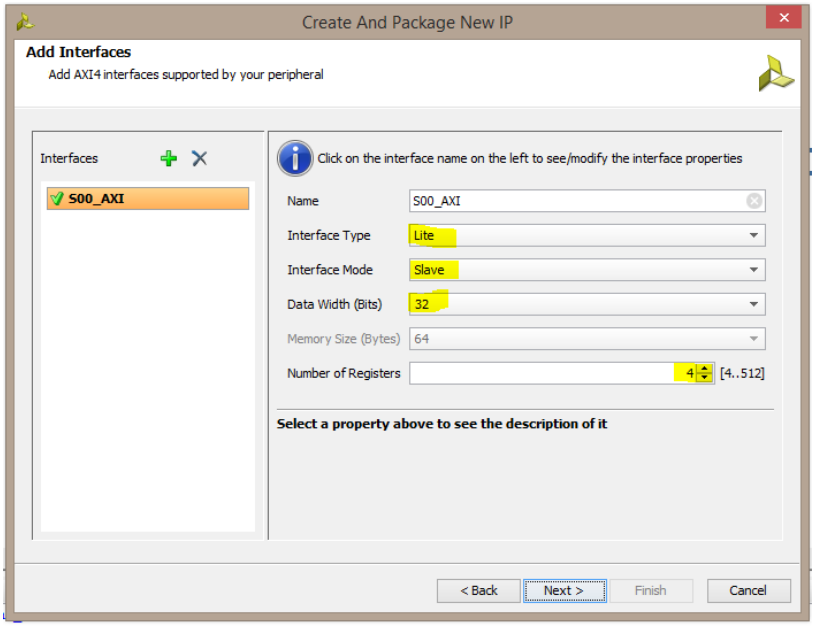



*Figure 3.4: Step 3.6, configuring AXI protocol for IP*

3.7. Click the **Next** button, when you are happy with this configuration.
3.8. On the "**Generation Options**" screen, leave the options unchecked and click next.
3.9. Select "**Add IP to catalog**" and hit **finish.**

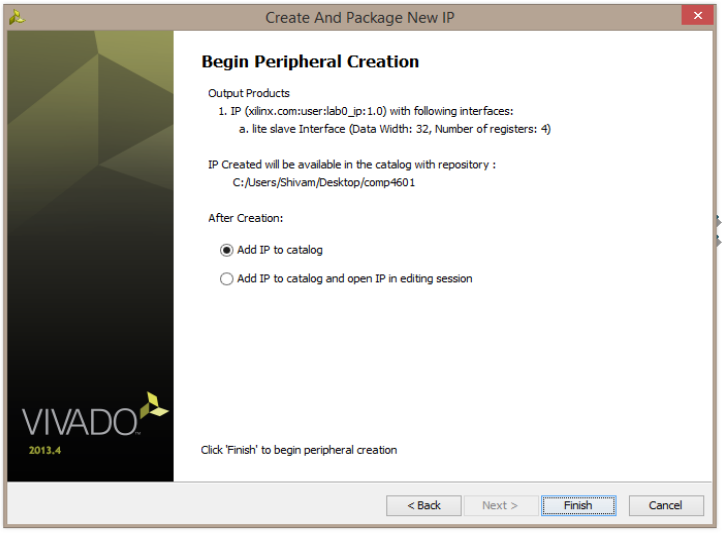



*Figure 3.5: Step 3.9, Finishing off the creation of Custom IP*

You have now generated your own Custom IP component and it can now be integrated into our high-level design. To then modify this IP we will create a Vivado project file for the Custom IP so that we can continually modify the Custom IP, abstracting it from the high-level.

## 3.b Creating a Project file for Custom IP

3.10. On the block diagram (with **only** the ZynQ7 IP instantiated), select "**Add IP**" and find the lab0_ip that you just created.

3.11. Select the "**Run connection automation**" to the s00_AXI of the Custom IP just created, the end result should be as follows:

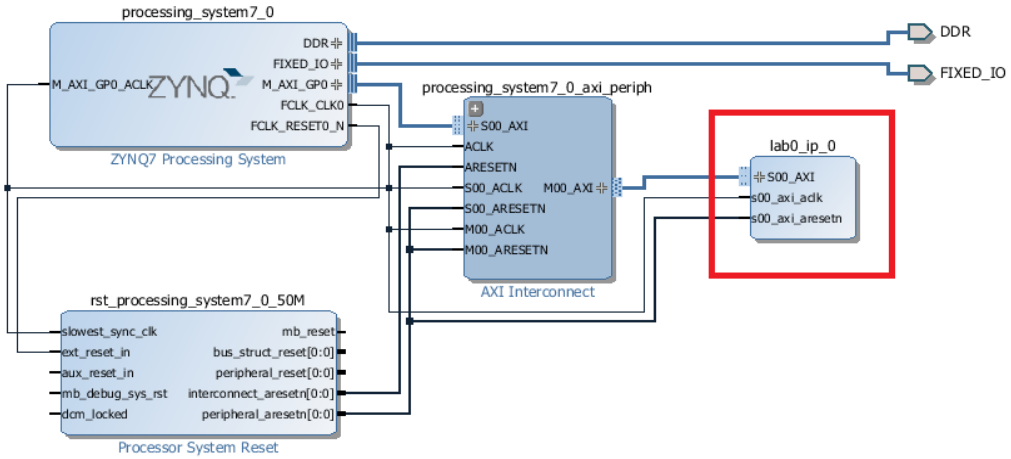



*Figure 3.6: Step 3.11, Adding Custom IP to your high level design*

3.12. **Save** your block design and/or project file.

3.13. Right click the lab0_ip_v1_0 (Custom IP) in your design and select "**Edit in IP Packager**"

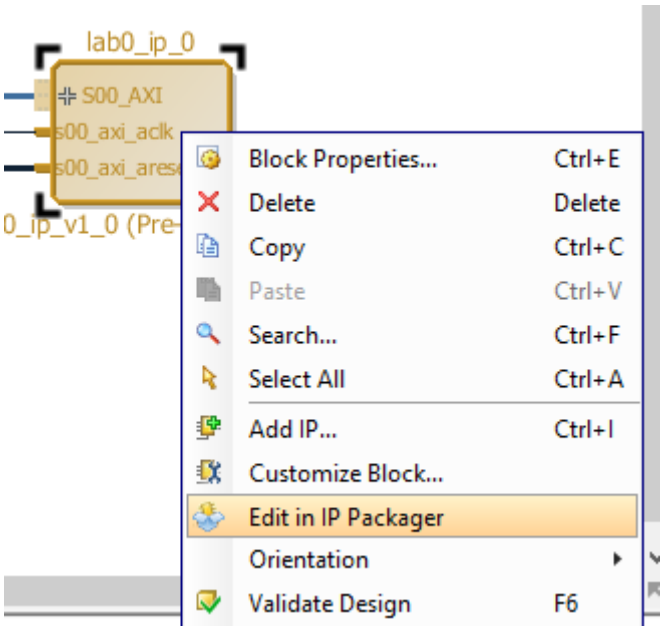



*Figure 3.7: Step 3.13*

3.14. Select "**Ok**" in the project location screen

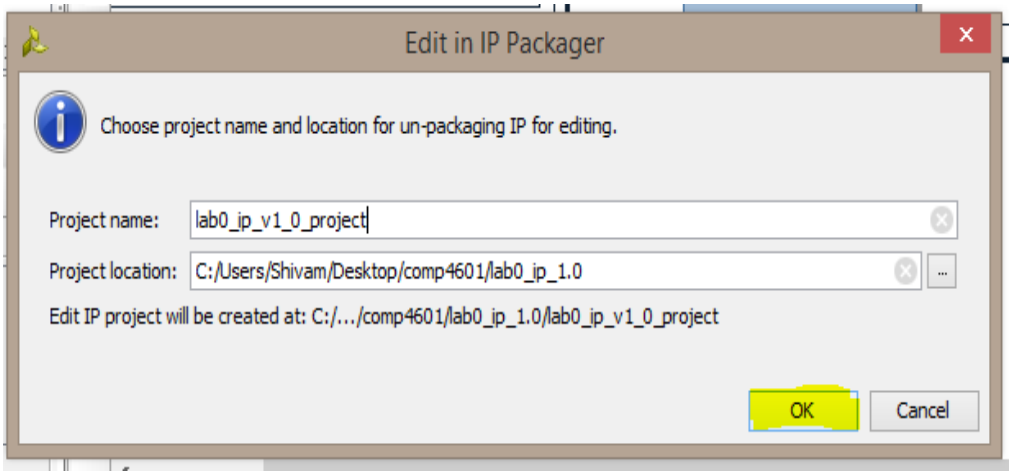



*Figure 3.7: Step 3.14*

3.15. When the new instance of Vivado shows up the **first thing to do is to close it**. The reason for this is so that a permanent project file will form, such that we may more easily edit the IP in future without having to keep generating temporary project files, and protect against files being lost if Vivado crashes.

3.16. Here is a breakdown of what your directory structure should look like, where you now have two project files one for the high-level module (lab0) and the second which contains just the IP file (lab0_ip_1.0).

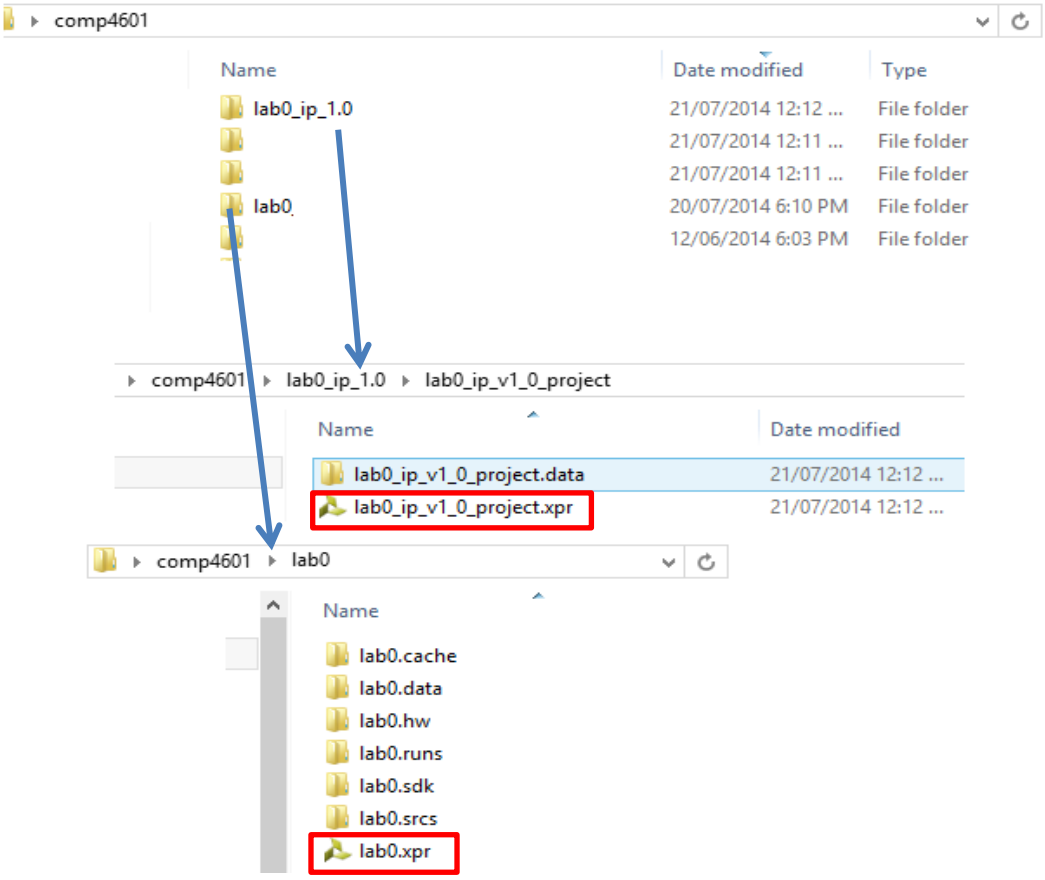



*Figure 3.7: Step 3.16, Expected directory structure*

3.17. Now go to the **lab0_ip_1.0/ lab0_ip_v1_0_project** and **open up the .xpr** file shown above. (i.e. open up the Vivado project file for the Custom IP). This should be close to an identical view of "edit in IP packager" which we temporarily saw before.

3.18. **Open** the VHDL file called "lab0_ip_v_1_0_S00_AXI.vhd", from the project manager view.

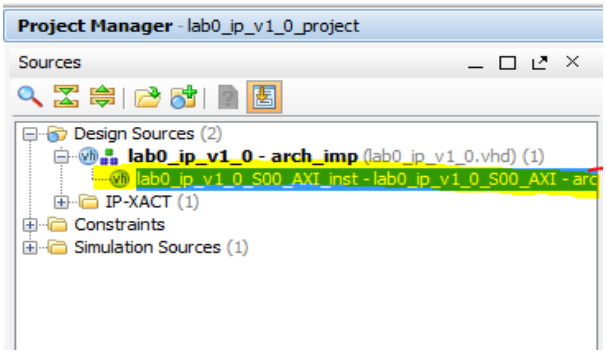



*Figure 3.8: Step 3.18, Opening the Slave AXI file*

# 4. Customising the Custom IP

In this section we will first go through the generated Slave AXI file to explain how the AXI protocol works, via a tutorial on the AXI-LITE interface, and followed up by simple modifications to the Custom IP component to set the stage for extending the generated functionality provided by Vivado.

## 4.a AXI Tutorial

Advanced eXtensible Interface (AXI) is a protocol developed by ARM which is a mechanism for controlling shared bus access. Some of the key features of this protocol are as follows:

- Separates address, control and data lines
- Incredibly simple handshaking, due to the separate control lines
- Burst mode transfer supported with the provision of only a starting address
- Uses a Master Slave model, with the Master being solely responsible for the arbitration of the bus, directing writes and requesting reads from the Slave
- See the “AXI Reference Guide” [2] for extensible documentation of the AXI protocol

The Master accesses the Slaves by loading the address bus with an address that is within the Slave’s assigned address range. Because the address bus is shared between all Slaves it is generally the responsibility of the Slave to ignore any request to an address if the address is not within its assigned range, before acting upon a request to transfer data. However when connection automation is run on your Custom AXI IP, Vivado inserts an AXI Interconnect between the real Master (the Zynq Processor) and the Slave IP (See the “Xilinx AXI Interconnect documentation” [3] for the implementation of the interconnect). Essentially, it does most of the heavy lifting on arbitration and implements a bus like interface via multiplexers and internally embedded routing data.

The implication of this is that the Slave and Master AXI components can be significantly simplified such that they do not have to check addresses on the bus, and once the ready/valid signals are asserted for a particular Slave it does not have to re-check addresses. This further builds abstraction allowing simpler and more generalised Slaves (variable addresses) to be connected to Master components, the downside of this is that this introduces some delay (which will be seen in the timing diagrams in the following subsections). A detailed explanation for the delays can be found in the “Xilinx AXI Interconnect documentation” [3].

### 4.a.i AXI Writes

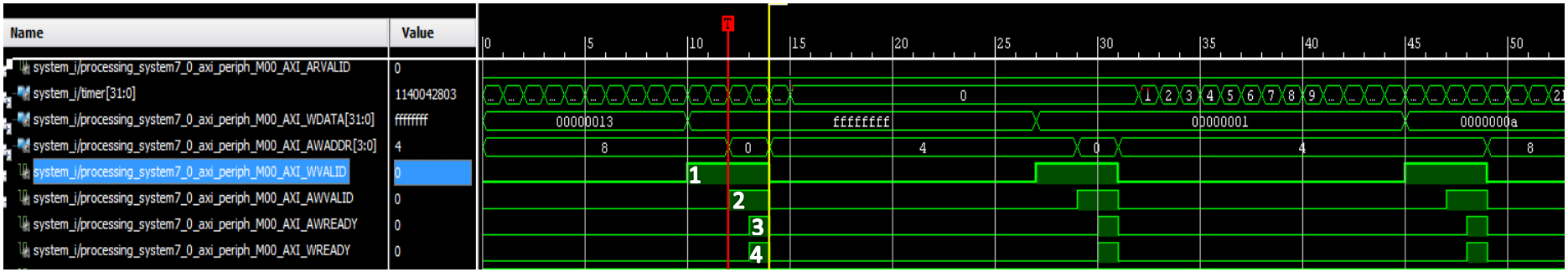



*Figure 4.1: Debug output for AXI write transactions*

The waveforms in Figure 4.1 show the Master writing 0xFFFFFFFF @ BASE_ADDR (0x0) then 0x00000001 @ BASE_ADDR (0x0), and finally 0x0000000a @ BASE_ADDR+4 (0x4). Note how the signals are prefixed with "M00_AXI" (Master AXI) instead of "S00_AXI" (Slave AXI), this occurs due to the fact that when debugging the AXI bus, one end if connected to the Master and the other end to the Slave.

How AXI writes are initiated by the Master (numbers refer to the labelled signals between clock cycles 10-12):

1. Master* sets up **WDATA** (with 0xFFFFFFFF) and asserts **WVALID** (write data is valid)
2. Master* sets up **AWADDR** (with 0x0) and asserts **AWVALID** (Master asserting that it has placed the valid address on the address bus)

*Master – strictly speaking it is the AXI interconnect which acts as the Master for this Slave AXI component, note the PS.

The Slave then responds as follows (numbers refer to the labelled signals between clock cycles 13-15):

3. Asserts **AWREADY** (write address can be accepted by the Slave, determined by **WVALID && AWVALID**)
4. Asserts **WREADY** (write data can be accepted by the Slave, determined by **WVALID && AWVALID**), at this point the **WADDR** address is also latched (stored address so the Master may perform some other operation, yet Slave knows which address the data relates to)

Once **WVALID & AWVALID & AWREADY & WREADY** are all asserted

- Slave register write is enabled
- Next clock cycle (14$^{th}$ clock cycle in the figure, yellow line) the Slave register (slv_reg0 - since address was 0x0) has data on **WDATA** bus written into it.

### 4.a.ii AXI Reads

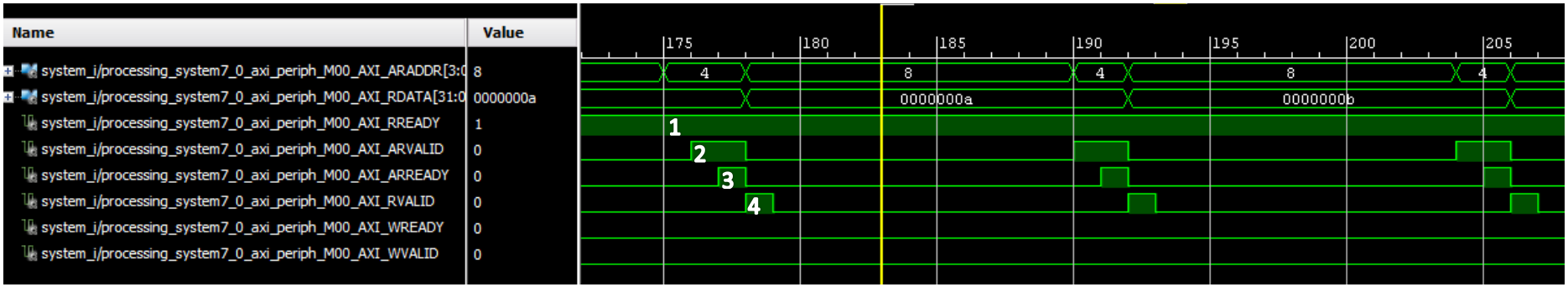



*Figure 4.2: Debug output for AXI read transactions*

The waveforms in Figure 4.2 show the processor reads from a FIFO which contains the data {0x0a, 0x0b, 0x0c, …..} from the Custom IP via AXI-LITE at the address of BASEADDR+4 (0x4).

To initiate an AXI Reads the Master performs the following (numbers refer to the labelled signals between the clock cycles 175 – 177):

1. For AXI-LITE, the Master generally always has the signal **RREADY** asserted, signalling that it is able to receive data from the Slave
2. Master then places the address (0x4) that it wants to read from onto the **ARADDR** bus and asserts **ARVALID**

The Slave then performs the following (numbers refer to the labelled signals between the clock cycles 177 – 180):

3. The Slave asserts **ARREADY** to signal that the address has been accepted by the Slave
4. Slave then sets **RDATA** to reflect the appropriate data (0x0000000a), and asserts **RVALID** upon which **ARREADY** is de-asserted. At this point (178$^{th}$ clock cycle) the correct read data is placed onto the bus, where it has one clock cycle to be read by the Master, after which **RVALID** will be de-asserted.

   Since the Master (AXI Interconnect) and the Slave are clocked at the same rate (FCLK_CLK0) the **RVALID** signal can be viewed as a latch signal for the AXI Interconnect to store this data into its own internal register and later forward it to the real Master (the Zynq Processor).

## 4.b Customising the Custom IP

Based on the tutorial on the AXI protocol in the previous subsection, it should be clear that these signals can used by the hardware designer to determine whether or not a read/write has been placed by the Master and to determine if certain actions on the Slave's end should be undertaken. There are a number of ways in which this effect can be achieved, and we will introduce these via a sequence of simple projects in described in Section 6 of this report.

Before we begin there are some general modifications that will greatly speed up the design process. They are as listed in the following sections; there are also a few of files and naming conventions to take note of before starting:

- **Slave_AXI** (lab0_ip_v1_0_S00_AXI.vhd) – generated file which implements the AXI-LITE handshaking process and stores all writes into registers, and uses those same registers as read response values.
- **Toplevel** (lab0_ip_v1_0.vhd) - refers to the VHDL file that encapsulates the AXI implementation file described above. You'll notice that is largely empty, and it is where we will be focussing our implementation efforts. When coding your own designs it is recommended that you use this file as a connection point for your main VHDL components. However given this is a relatively small lab we will code entirely within this file for convenience.

Figure 4.3 below shows the overall organization of to our design. The high-level project file contains the IP blocks (Zynq, AXI Interconnect and Custom IP) as well as the lab project file, which consists of the files mentioned above. The green arrows in Figure 4.3 denote the changes we will be making to expose some of the internal signals of the Slave_AXI file into the Toplevel where we can then code our implementation or instantiate our components. The reason for this is so that we preserve the protocol implemented by the Slave AXI file and concentrate our efforts on the production of the device logic (Implementation code, orange box in Figure 4.3).

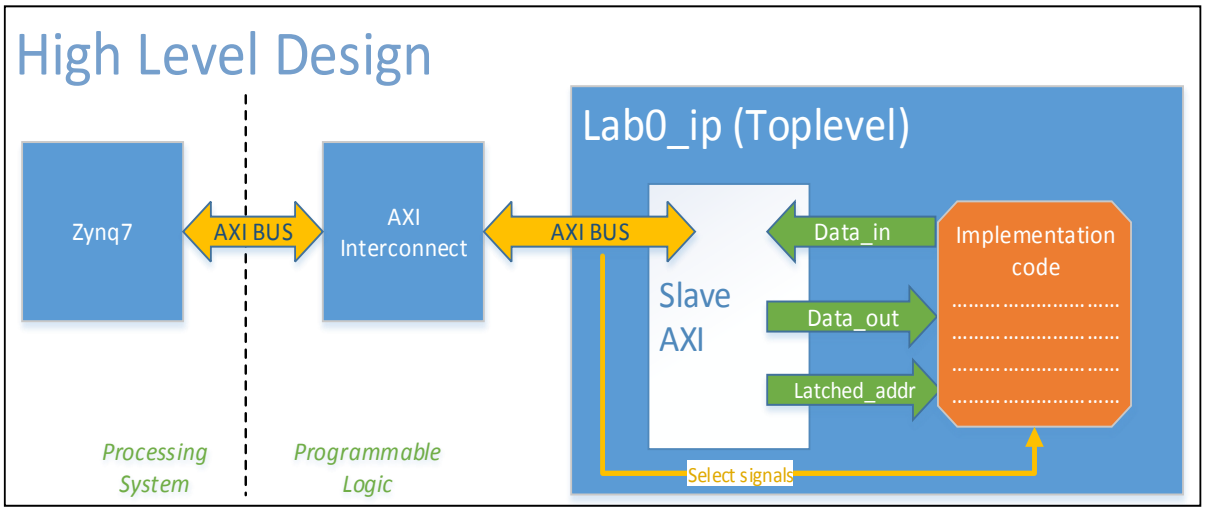



*Figure 4.3: System diagram denoting the changes we are about to make (green arrows) and a high level overview of how all the components are connected.*

### 4.b.i Changes to Slave_AXI

In Figure 4.4 and the Slave_AXI code you should notice that the RDATA bus for the AXI reads is driven by reg_data_out. This signal is originally driven by the Slave registers (which are the registers the AXI write data is stored into). Since this is trivial functionality, with there being very little that we can do with such an implementation, we'll replace it with our own signals (datain0/1/2/3, see Figure 4.4) which will later be declared as inputs to the Slave_AXI entity.

```
353     process (~~slv_reg0, slv_reg1, slv_reg2, slv_reg3~~, axi_araddr, S_AXI_ARESETN,
354     slv_reg_rden,datain0,datain1,datain2,datain3)
355     variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
356     begin
357       if S_AXI_ARESETN = '0' then
358         reg_data_out  <= (others => '1');
359       else
360         -- Address decoding for reading registers
361         loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
362         case loc_addr is
363           when b"00" =>
364             reg_data_out <= datain0;--slv_reg0;
365           when b"01" =>
366             reg_data_out <= datain1; --slv_reg1;
367           when b"10" =>
368             reg_data_out <= datain2; --slv_reg2;
369           when b"11" =>
370             reg_data_out <= datain3; --slv_reg3;
371           when others =>
372             reg_data_out  <= (others => '0');
373         end case;
374       end if;
375     end process;
```

*Figure 4.4: Changes to AXI read data*

Now that we have taken care of the output values, we next need to get the AXI written values out of this component as well; hence we can simply pipe out these values (since they are all implemented as registers) out of this generated AXI component onto the Toplevel where their values can be used (See Figure 4.5). Notice from the timing figures in Section 4.a.(i,ii) that the address bus data is valid for a very short amount of time, hence the **axi_awaddr/axi_araddr** are used by the implementation as latches for the address, storing it for a single transaction (write or read respectively), hence we will need these values as well.

```
124   begin
125     -- I/O Connections assignments
126     dataout0 <= slv_reg0;
127     dataout1 <= slv_reg1;
128     dataout2 <= slv_reg2;
129     dataout3 <= slv_reg3;
130     latched_waddr <= axi_awaddr;
131     latched_raddr <= axi_araddr;
```

*Figure 4.5: Forwarding out the written values/latched addresses*

As a consequence of the changes above the following signals (Figure 4.6) have to be added to the port definition of the Slave_AXI component, so that the toplevel component can pipe in AXI read values, and the written values can be read by the toplevel.

```
80      S_AXI_RVALID    : out std_logic;
81      -- Read ready. This signal indicates that the master can
82    -- accept the read data and response information.
83      S_AXI_RREADY    : in std_logic;
84      datain0,datain1,datain2,datain3     : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
85      dataout0,dataout1,dataout2,dataout3 : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
86      latched_waddr, latched_raddr        : out std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0)
87    );
88  end lab0_ip_v1_0_S00_AXI;
```

*Figure 4.6: Signals to add to the Entity declaration*

We are now done with the Slave_AXI implementation file, and will now move onto making changes which correspond to these on the Toplevel file.

### 4.b.ii Changes to Toplevel

There are two ways in order to utilise the values from the processor:

- Firstly you can simply use the existing register implementation in the generated AXI protocol and read from the registers. This will work if your implementation is not dependent on user (Master Processor) actions and simply runs on its own based on the values the user has sent to you. While for AXI-reads static values can simply be inserted and read by the Master on an as needed basis.
- The second (much more useful approach) is to do this in real time while the reads/writes are taking place on the AXI bus so that you can effectively 'snoop' the AXI bus lines, thus being able to realise when the Master has invoked an action and react to it accordingly. Of course the actual Slave_AXI implementation can be modified to achieve this goal as well. However, to keep the complexity down to a minimum and to avoid difficult debugging problems we'll stick to the snooping-based approach.

We'll come back to these ideas in the Section 6, but for now we'll stick to some simple modifications so that we can determine if the changes that we have made to the source of the IP have carried through to our high-level design.

To start off modify the port map of the Slave_AXI component within the Toplevel, to add signals to the new ports that we have just added in. Figure 4.7 shows the modifications that were made to the port map. Also remember to declare the corresponding signals which have just been added to the port map of Slave_AXI.

```
103  lab0_ip_v1_0_S00_AXI_inst : lab0_ip_v1_0_S00_AXI
104      generic map (
105          C_S_AXI_DATA_WIDTH    => C_S00_AXI_DATA_WIDTH,
106          C_S_AXI_ADDR_WIDTH    => C_S00_AXI_ADDR_WIDTH
107      )
108      port map (
109          S_AXI_ACLK    => s00_axi_aclk,

129          S_AXI_RREADY    => s00_axi_rready,
130          datain0         => datain0,
131          datain1         => datain1,
132          datain2         => datain2,
133          datain3         => datain3,
134          dataout0        => dataout0,
135          dataout1        => dataout1,
136          dataout2        => dataout2,
137          dataout3        => dataout3,
138          latched_waddr   => lwaddr,
139          latched_raddr   => lraddr
140      );
```

*Figure 4.7: Signals which need to be added to the port map of the Slave_AXI, lines 110-128 are existing ports, requiring no changes.*

We will now implement some trivial logic (shown in Figure 4.8) for the purposes of testing the changes that we have made. Notice how we have altered the default generated operation of the IP now, since the registers which stores the write values by the Master (dataout0,dataout1) are now set to the AXI read values (datain1,datain0). However it is writes to register 1 (BASE_ADDR + 4) which set the read value of BASE_ADDR. Meanwhile for the reads from BASE_ADDR + 8 (or +12), the constant values of 3 and 4 will be read from these addresses regardless of the values have been written.

```
141
142      --Add user logic here
143      datain0 <= dataout1;
144      datain1 <= dataout0;
145      datain2 <= X"00000003";
146      datain3 <= X"00000004";
```

*Figure 4.8: Simple variation to the original functionality, to test the changes that we have made*

Within the IP project file, click on the "**Synthesise**" button in the left hand pane, so as to check for any compilation errors, and so that the high-level synthesis does not fail later down the track. Once this is successful all that remains is to save these changes within the IP, then implement it alongside the processor and start testing out the implementation (covered in the following section).

The importance of following the steps outlined in this section is that you no longer have to worry about the Slave_AXI implementation file, since you have exposed all the signals that will be useful to your implementation. From now on you are free to modify only the Toplevel and to instantiate the sub components in the Toplevel that you require.

# 5 Packaging and testing your IP

In this Section we will package the Custom IP (Toplevel and Slave_AXI files) that we have generated and modified. Once packaged, it can be used in the high-level design as an independent IP block, exactly like how you would treat other such IP blocks. Then all that remains is to write some C (driver-like) code to interface with the IP and ensure that the changes that we have made are working. This process will need to be repeated every time you change the VHDL, note however a lot of the steps below are listed as **conditional**.

## 5.a IP Packager (Within the Custom IP's Vivado project)

5.1. Select the "**Package IP"** in the project manager section in the left hand pane.

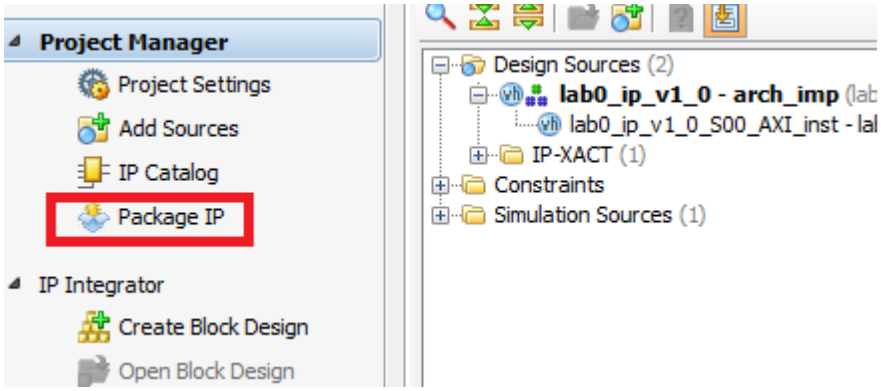



*Figure 5.1: Step 5.1, starting the packaging process*

5.2. At the start screen leave all the options the same except for the version number; ensure that you **INCREASE** the version number (e.g. 1.0 -> 2.0). The reason for this is so that Vivado will detect the version change, and prompts you for an upgrade. Also, alter the display name to reflect the version number change.

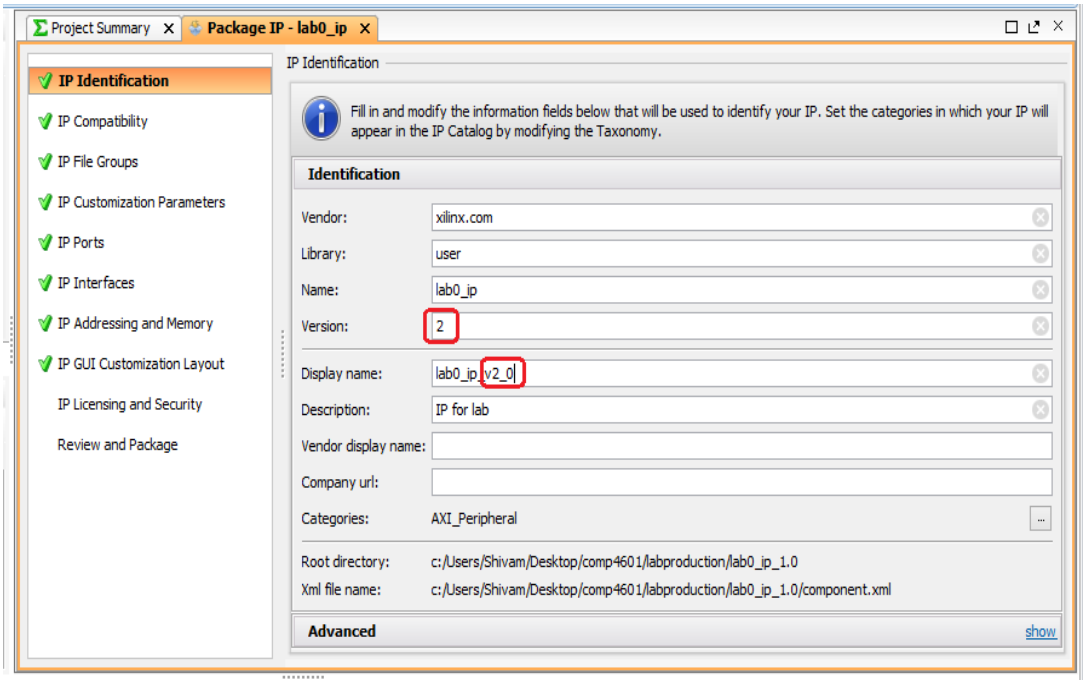



*Figure 5.2: Step 5.2, altering the version number for your IP*

5.3. **If** the changes to the file involved adding new VHDL files, they must be added in the **"IP File Groups"** to both the "VHDL synthesis" and "VHDL Simulation" folders, as shown by Figure 5.3.

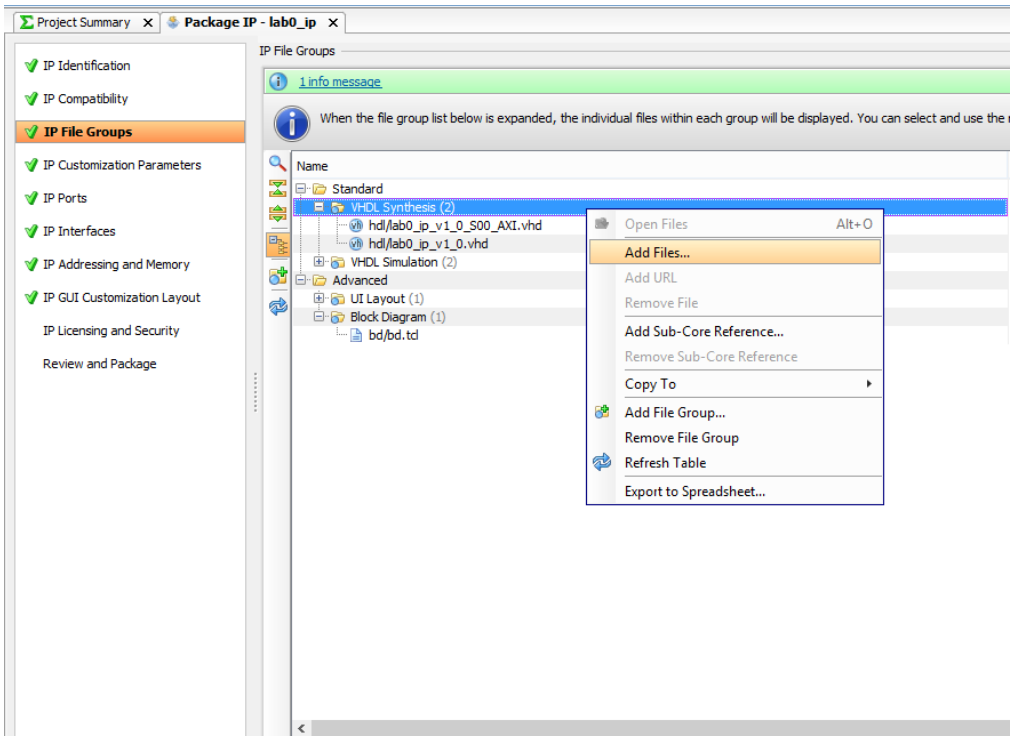



*Figure 5.3: Step 5.3, Add VHDL files to the IP definition if needed*

5.4. **If** the ports to the Toplevel have been changed, then use the "**IP Ports**" page by simply clicking on the Port import dialog and following the prompts.

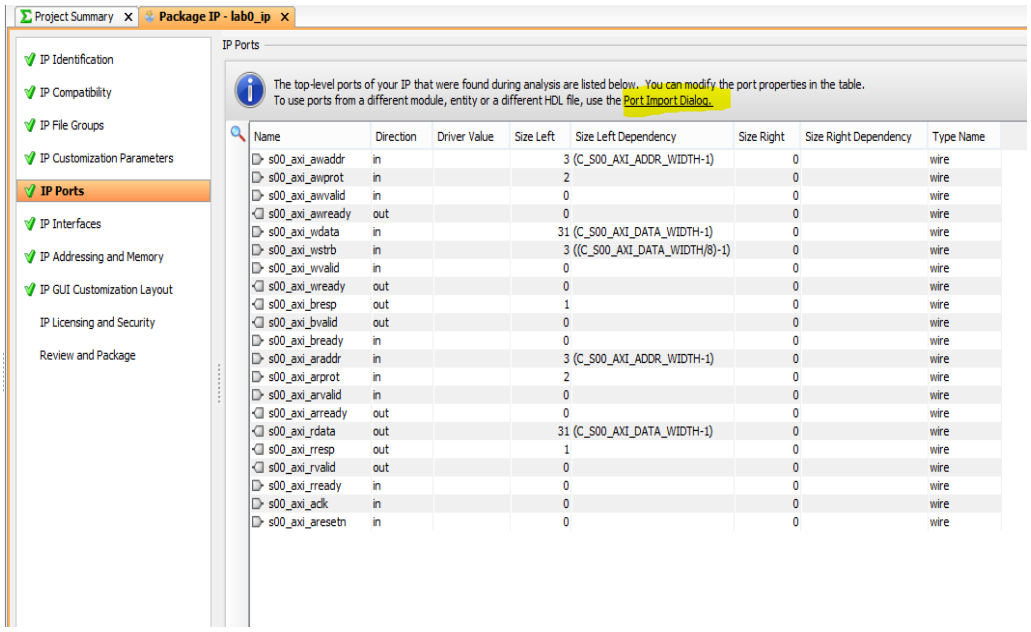



*Figure 5.4: Step 5.4, Modifying IP ports if needed*

5.5. If you used the "**IP ports**" page to add/remove ports, you should now go to the "**IP GUI Customization Layout**" and use the IP GUI customization layout wizard to regenerate the image of the IP component. Simply use the run the wizard link to regenerate the diagram of the IP.

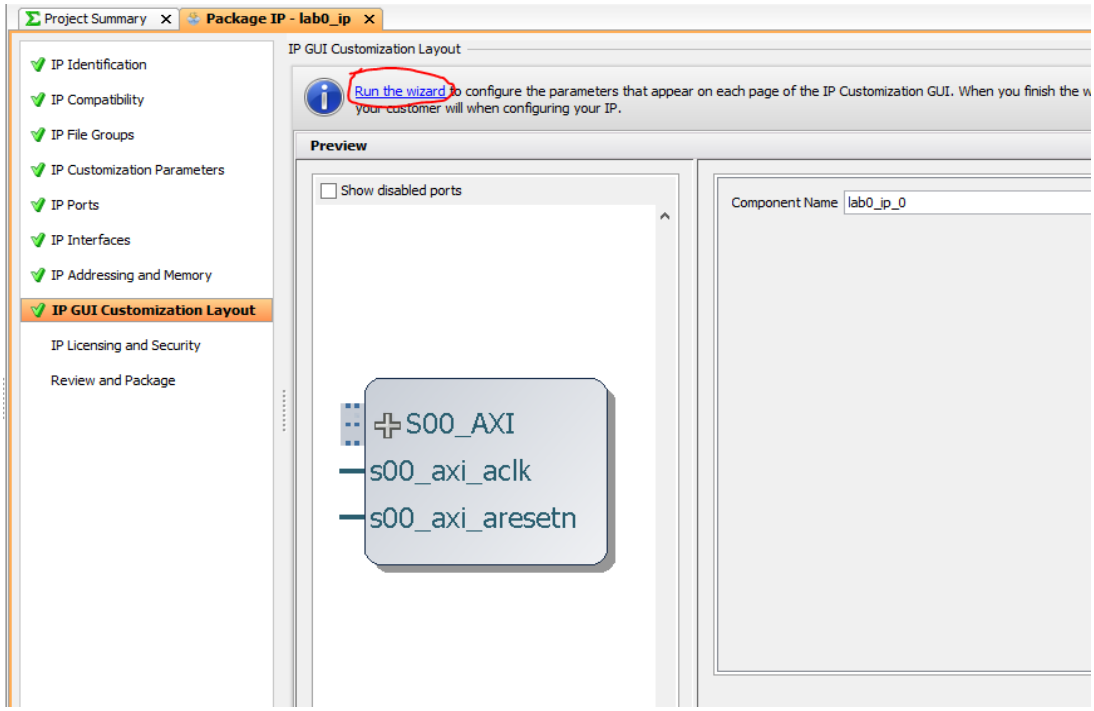



*Figure 5.5: Step 5.5, Regenerating the IP GUI*

5.6. To complete the process, in the "**Review and package"** screen select **Re-Package IP**.

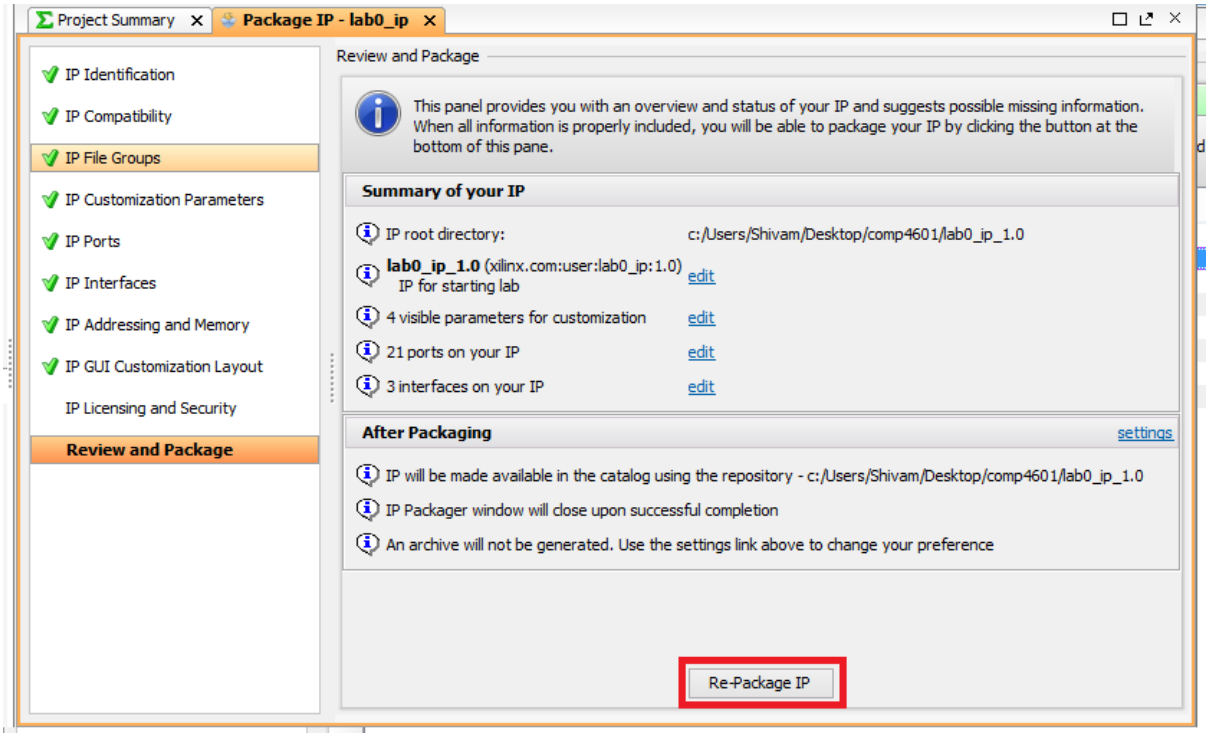



*Figure 5.6: Step 5.6, finalising the packaging process*

Screens which were skipped:

- **IP compatibility** - it is used to specify what boards the IP is valid for, which will always be the ZYNQ board for our designs.
- **IP Customisation Parameters** – should be used if the custom parameters (generic parameters) for the IP have been changed.
- **IP interfaces** – If the ports for the IP were changed and you wish to create a standardised port (e.g. you create a FIFO_WRITE port that you want to connect to a Xilinx FIFO) then you can group ports together to create an IP interface.
- **IP Addressing and Memory** – Informational only
- **IP Licencing and Security** – Informational only

## 5.b IP upgrade in high-level design (Within the high-level Vivado project)

5.7. Now **reopen the high-level design Vivado file** and open the Block Design

5.8. Select the **TCL console** window and run the following commands

a. `'update_ip_catalog -rebuild'`
   i. This refreshes the IP repositories specified in Project Settings > IP > IP Repositories (you can do this manually if you wish)

b. `'report_ip_status -name ip_status_1'`
   i. This generates an IP report, showing whether or not the IP in your design are up to date

5.9. This command should report that lab0_ip_0 has a “Major Version Change” (can also be minor, depending on the version number you selected)

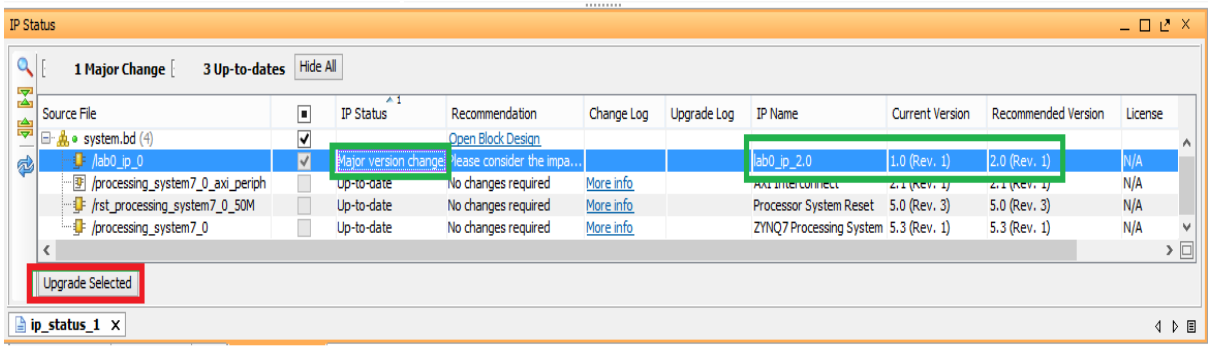



*Figure 5.7: Step 5.9, Vivado reporting changes to the IP in your high-level design*

5.10. Ensure it is selected and **hit upgrade selected** button (Vivado will now upgrade the IP, retaining all existing connections)

5.11. Regenerate the HDL wrapper for your high-level design

5.12. Hit the **Generate bitstream** button (which should also synthesise and implement your high-level design) and wait for this to finish

5.13. Open Implemented Design

5.14. File > Export > **Export Hardware for SDK**

## 5.c Interfacing with the Custom IP

On the software side of things, all that remains is to write some interface code and test the functionality of the hardware. When creating the **application project** it's generally best to use the "Hello World" example project as a template, since one of the first steps that it performs is to initialize the UART.

You will need to "**#include xparameters.h"**; if you read near the top of the file you should find the definition of the LAB0_IP_0_S00_AXI_BASEADDR and HIGHADDR. The addresses should correspond to those listed in Vivado's "**Address Editor**", and since the generated IP has a 4 register implementation, only the bottom 4 bits of the address sent will be seen on the Slave's side (byte addressing, 32 bit data bus). You should also "**#include <xil_io.h>**" to get the Xil_OUT32/Xil_IN32 function definitions.

```
/* Definitions for peripheral LAB0_IP_0 */
#define XPAR_LAB0_IP_0_S00_AXI_BASEADDR 0x43C00000
#define XPAR_LAB0_IP_0_S00_AXI_HIGHADDR 0x43C00FFF
```

*Figure 5.8: AXI address range of the AXI peripheral, inside "xparameters.h"*

Once you've verified this, go back to "**helloworld.c**" and add the code shown in Figure 5.9 below. This code writes 4 values and reads 4 values back from the IP. The expected output should be "Values read = 00000002, 00000001, 00000003, 00000004" if you followed the steps in Section 4.b of this lab correctly.

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include <xil_io.h>

#define TIMER_STOP 0
#define TIMER_START 1
#define TIMER_RESET 2

int main() {
    init_platform();
    Xil_Out32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR, 0x1);
    Xil_Out32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR+0x4, 0x2);
    Xil_Out32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR+0x8, 0x990);
    Xil_Out32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR+0xC, 0x100);

    u32 r0,r1,r2,r3;
    r0 = Xil_In32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR);
    r1 = Xil_In32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR+0x4);
    r2 = Xil_In32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR+0x8);
    r3 = Xil_In32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR+0xC);

    xil_printf("Values read = %0X, %0X, %0X, %0X\r\n",r0,r1,r2,r3);
```

*Figure 5.9: C code which is used to test our Custom IP*

If you were to write to the address: XPAR_LAB0_IP_0_S00_AXI_BASEADDR + 16, it would mimic the effect of writing to XPAR_LAB0_IP_0_S00_AXI_BASEADDR, since the Slave only sees a 4 bit address and 0x43C00000 => (0b0000) and 0x43C00010 => (0b0000) while S00_AXI_BASEADDR + 20 = 0x43C00014 => (0b0100), and so on.

Also note that the xil_io.h file contains references to functions like Xil_in8, Xil_out16 etc. you may have considered using the following functions given that we are reading/writing such small data sizes. However when using these functions the data may be stored at the MSB segment of the data bus (similarly for reads) which will cause misinterpretation by our IP, therefore it is recommended that you self-manage this by always using the in32/out32 such that the performance remains consistent. Similarly, note the use the u32 data type when dealing with the read/write across AXI. Of course you may cast these numbers to whatever you like after the read. However prior to the operation it is best to use these exactly sized data types such that any ambiguously sized data types do not cause unexpected behaviour.

# 6 Implementation Exercises

Now that you have gathered an overview of the process of modifying the Custom IP component, repackaging the IP and then integrating it back into your high-level design, what remains is to develop some more useful implementations on the hardware side. All exercises are intended to be implemented in the Toplevel file of the Custom IP. The AXI protocol for this lab is shown in Figure 6.1.

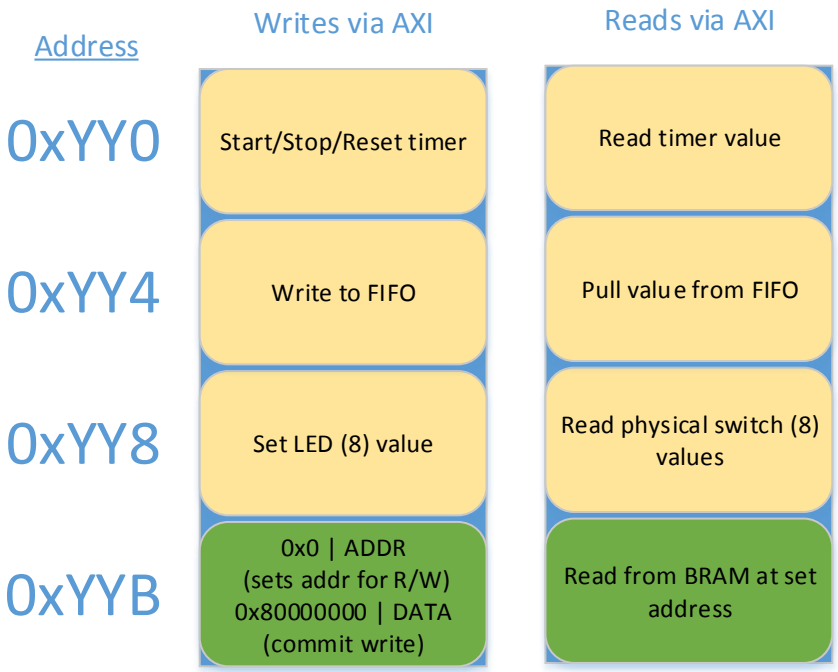



*Figure 6.1: AXI Protocol for the Implementations to follow*

## 6.a Timer implementation (32 bits)

For the timer we will implement a simple register based implementation of a PL timer, which runs at FCLK_CLK0 and counts the number of clock cycles between elapsed since the timer was last reset by the user (Zynq Processor, AXI Master). The data word written by the Master acts as a controller for the state of the timer:

| Bit | 31 | 30 | ..... | ..... | ..... | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Function | n/a | n/a | ..... | ..... | ..... | Reset | Enable |

*Table 6.1: Timer control register*

Implementing this will be quite simple, since as we have previously noted any AXI writes from the Master to the Slave are stored in the Slave registers (which we have piped out to the Toplevel). Since the timer will generally check the enable bit every clock cycle and continue operating as long as this bit is set, the timer can be considered to be a counter which has a reset signal controlled by bit 1 of the **dataout0** and bit 0 of this same register being the enable counter signal.

After implementing the internals of the timer, all that remains is getting the value of the timer back to the Master (PS). Since the AXI data bus width we selected is **32 bits**, it's recommended that you implement a 32bit timer, and since this value needs to be provided every time the user reads from 0xYY0 (where YY is any number) we simply need to set the **datain0** signal we added earlier to be the

timer's current value. Appendix A contains the solution to the timer, in case you wish to verify your code prior to compiling it.

## 6.b FIFO implementation

Specification:

- FIFO of data width **16 bits**, **1024 words** in size
  - This should be implemented as block ram in your toplevel with an address width of 16 bits and a data width of also 16 bits. If you need a refresher as to how to go about this, please refer to "Distributed and Block ram on Xilinx FPGA's" guide [5]
- If we reach the end of FIFO addressing, writes/reads should simply wrap around the BRAM
- Similarly reading beyond data in the FIFO should not cause a loss of 'place' the user is up to in the FIFO. For a detailed walkthrough on the operation of the FIFO see Figure 6.3.
- If the user tries to read beyond the data in the FIFO, bit 31 (Most Significant Bit) should be set, to indicate that the data is invalid, and it is expected that the user checks this bit for data validity.

| Bit | 31 | 30 | ….. | 15 | …… | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Function | Read Invalid | n/a | ….. | Data | | | |

*Table 6.2: FIFO read interpretation (write is identical with bits 16 to 31 unused)*

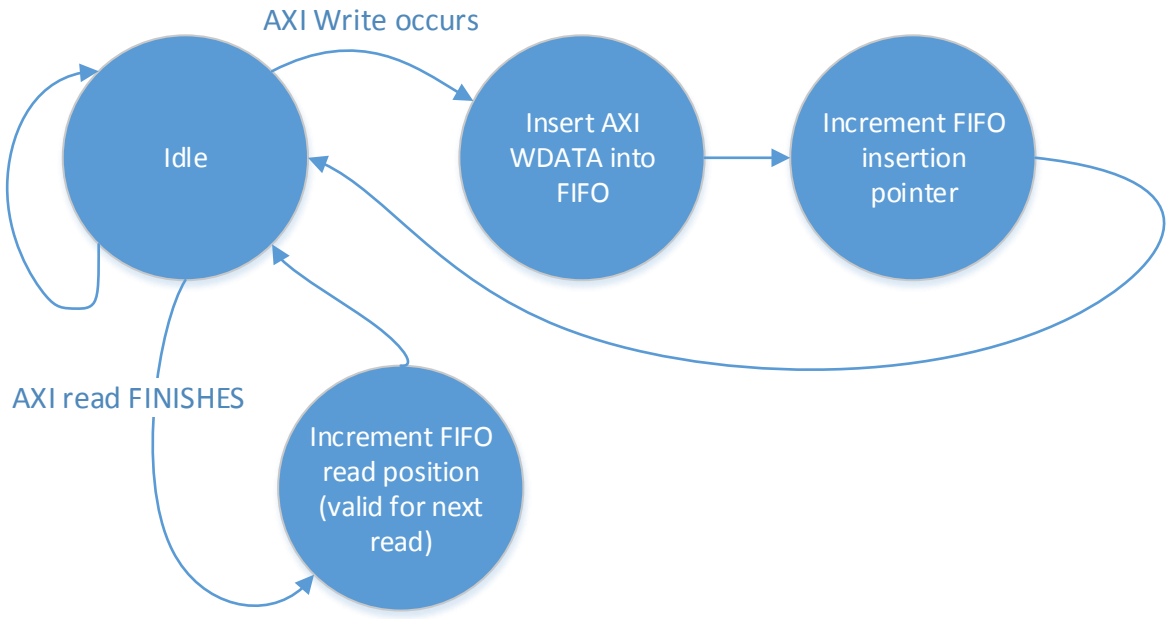



*Figure 6.2: FIFO Finite State Machine (FSM)*

Figure 6.2 details the FSM that you will be implementing, it should be noted that for reads, you should not be doing anything during the read process (so that the data is stable during the read) and instead focus your efforts towards ensuring that at every stage valid data is available to be read form **datain1**, more details to follow.

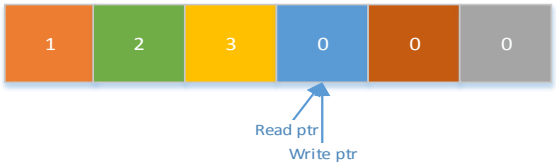
Implementing a FIFO will be more challenging than the timer, since we can no longer process every data every clock cycle depending on the value written, instead the solution to this problem is one

which will involve listening on the AXI bus lines to figure out when a write/read has taken place and perform the following:

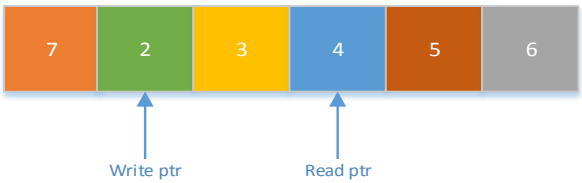
- **Write –** When we know that a write is taking place, we should read the data bus (**WDATA**) and set this as the FIFO data input, as well as enabling the FIFO write for exactly one clock cycle. Referring back to Section 4.a.i and the original source code for Slave_AXI, it should be noted that the **S_AXI_WREADY** is asserted by the Slave for exactly one clock cycle once the write was successful. We can probe this signal as high and once so, enable a write to the FIFO. Hence we will be performing our FIFO insertion operation at the start of the 14th clock cycle in Figure 4.1, and as you can see WDATA is valid at this point in time.
- **Read –** From the timing diagrams (in section 4) it should be apparent that there is only a couple of clock cycles between the Master issuing a read and it actually being performed, so instead of trying to provide a read result at the exact instance it is required, we shall set up the **next read** value after the previous read has taken place. Referring back to Section 4.a.ii and the original source code it should be noted that when **S_AXI_RVALID** is asserted the channel has valid read data, furthermore it too is asserted for exactly 1 clock cycle, so if we were to wait for this signal to be asserted on the rising edge of the clock, the read will have taken place (by the time we view the signal) and we can safely replace the value of **datain1** to point to the next value in the FIFO. This position in time is denoted by the start of the 179th clock cycle in Figure 4.2.

The last point to note is that you also have to check the address of the write/read operation to ensure it is a FIFO operation (denoted by the addressing corresponding to 0xYY4). However if you refer back to the timing diagrams in Figures 4.1/4.2 you'll notice that the write/read address is only valid for a very small amount of time. Therefore we will need to make use of the **latched write and read addresses** and check the [3..2] bits are equal to "01".


1. Initial FIFO, after 3 values have been written




2. After 3 values read by user, note next read should not move read ptr. Set bit 31 to indicate current read is invalid




3. After 4 more values have been written by the user, note read ptr is now valid


4. More values written, note the write ptr has moved beyond read ptr. Undefined behaviour.

*Figure 6.3: Run through of FIFO behaviour*

The FIFO diagrams in Figure 6.3 denote the functioning of the FIFO; they should all be fairly easy to follow. The one which causes us some concern is the 4th diagram, where the user has written over data that has not been read by the user yet. Since this is an exercise, we'll leave you to decide how to handle this:

a. Just ignore and assume the user knows not to overfill the FIFO
b. Keep a counter of writes to FIFO (decrement with reads) and if it hits 1024, stop writing to the FIFO, note it will not be possible to provide user **immediate** feedback when this occurs, only when a read occurs you may set bit 30 to indicate FIFO full.

At this stage you should be able to write some C code which push 1000 sequential values onto the HW FIFO and then read them back in the same order, to verify the workings of your hardware FIFO.

## 6.c GPIO implementation

For the GPIO implementation we will mimic the functions of the advanced embedded design Lab 1. However, we will do it entirely through our Custom IP component, so that we have a better idea as to what is going on. This process will involve adding ports to your IP as well as external pins and constraints to your high-level design.

Essentially we will be implementing two registers:

- LED reg – This register's value is written by the Master, and can therefore be read directly from **dataout2** and the output of this register will be directly connected to the LED pins.
- Switch reg – This register's value is set by the SWITCH pins and its value out should be the value for **datain2**, i.e. the value the Master receives when it reads from BASE_ADDR + 12.

Once you have implemented this very simple hardware solution, you will now need to repackage the IP. However since you have added two ports to the IP, you'll need to run the "**IP ports**" and "**GUI customisation**" of the IP packager (instructions above).

Finally when in the high-level design you need to declare the LED's and SW's and connect them up to external pins (see AED lab1 for how to declare in xdc file and refer to the "Zedboard user manual" [4] for the Switch pin numbers). Figure 6.4 shows what your high-level design should resemble. For software it is recommended to write an infinite loop which reads the switch values and sets the LED via your Custom IP. You may also wish to introduce a loop counting up to a couple of million, in between these two operations to introduce a small delay between the switches and LEDs.

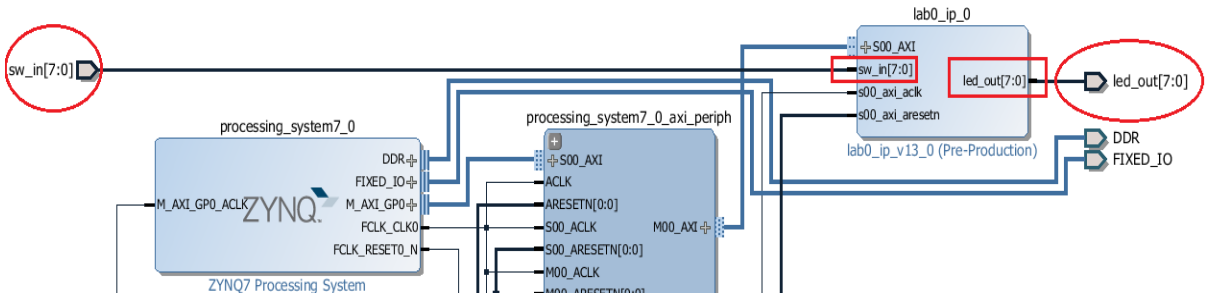



*Figure 6.4: High-Level Vivado project file, denoting the relevant pin to port connections you will need to make.*

## 6.d Block ram implementation

The following is left as a challenge exercise, with only the protocol to implement detailed.

Assumptions:

- BRAM which consists of shorts (16 bits)
- Has 2^16 addresses to write to
- Master has control over the whole BRAM, can read and write to any address in the BRAM

| Bit | 31 | 30 | ..... | 15 | ..... | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Function | A/D select | n/a | ..... | Address OR Data | | | |

*Table 6.3: Block RAM control register, stored by the signal **dataout1**, at our Toplevel*

**Bit 31** - selects what you are storing, either the address to read/write at or the data at the set address.

**Bits 15…0** - Is the Address/Data to correspond to bit 31.

So to perform a write/read operation at a particular address, you would need to write a driver which does the following:

```
void writeToBRAM(u16 addr, s16 data){
    Xil_Out32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR + 0xC, addr);
    Xil_Out32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR + 0xC, (1u << 31) | data);
}

s16 readFromBRAM(u16 addr){
    Xil_Out32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR + 0xC, addr);
    u32 read = Xil_In32(XPAR_LAB0_IP_0_S00_AXI_BASEADDR + 0xC);
    s16 readData = (s16) (0xFFFF & read);
    return readData;
}
```

*Figure 6.5: C functions to interface with the BRAM within the Custom IP*

# 7. Conclusion

Now that you are comfortable with utilising Vivado's built-in tools to generate and modify Custom IP, and the design flow related to the process; it's time to go out and design full-fledged hardware solutions with the knowledge that you have gained in this lab. While designing your own solutions we have a few recommendations:

- **Simulation –** In terms of compilation time and quality of debugging output, Simulation provides the fastest way to test hardware. So you should first make sure that your modules are flawless and then spend a bit of time at the end to integrate the VHDL components into the AXI data flow. One important point to keep mind of while simulating is that on an FPGA with clock speeds in in the MegaHertz range, the amount of clock cycles which occur within a second are more than what you could possibly view within simulation. Consequently ensure that your FSM's within the custom IP are initiated by the some AXI message, and STOP when the data is processed, rather than keeping these FSM's spinning waiting for the Master to read back the data. Whereby they may overwrite correct data simply due to the speed of the hardware. This is particularly evident if you decide to print out some data in between writing and reading data from the Custom IP, whereby a large amount of clock cycles will be used up writing out to the UART, thus distorting perception of time between the two operations.
- **Debug –** If your hardware is not working the way you envisioned (despite simulations telling you otherwise) one method of identifying the problem is to set all relevant signals as outputs to the Toplevel of the Custom IP. Then, utilising knowledge the gained in lab2 of Advanced Embedded Design, set all of these ports as debug. Once you have assigned the debug cores, set the waveform to trigger on a change in one of the AXI signals and run through your software, and finally analyse the waveform to work out where the issues lie. This approach has been tried and found to be much faster at identifying problems than trying to simulate your toplevel/Custom IP individually, since that approach involves having to "simulate" Master AXI behaviour!

# References

[1] Xilinx Custom IP guide, slightly outdated but quite comprehensive guide to Custom IP

http://www.xilinx.com/support/documentation/application_notes/xapp1168-axi-ip-integrator.pdf

[2] AXI reference guide

http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[3] Xillinx AXI Interconnect

http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf

[4] Zedboard user manual

http://www.zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf

[5] Block and distributed RAM's on Xilinx

http://vhdlguru.blogspot.com.au/2011/01/block-and-distributed-rams-on-xilinx.html

# Appendix

## *Appendix A (Timer Solution)*

```
--Timer implementation: uses the dataout0 signal to represent
--the current value which has been written to the timer's control
--register. And datain0 signal to output the timer value.
process(clk,dataout0)
begin
if (dataout0(1) = '1') then
     --"asynchronous" reset
     timer32 <= (others=>'0');
else
     if (rising_edge(clk)) then
          if (dataout0(0) = '1') then
               timer32 <= timer32 + X"00000001";
          end if;
     end if;
end if;
end process;

datain0 <= timer32;
```