|  |
| --- |
| Creating AXI-LITE ‘Custom IP’ in Vivado |
| Lab for COMP4601 |
| Developed by: Shivam Garg |

Contents

[Introduction 2](#_Toc395394524)

[0. Highlevel design configuration 3](#_Toc395394525)

[1. Create custom IP 4](#_Toc395394526)

[2 Customising the ‘Custom IP’ 9](#_Toc395394527)

[2.a AXI Tutorial 9](#_Toc395394528)

[2.a.i AXI Writes 10](#_Toc395394529)

[2.a.ii AXI Reads 11](#_Toc395394530)

[2.b Customising the custom IP 12](#_Toc395394531)

[2.b.i Changes to Slave\_AXI 13](#_Toc395394532)

[2.b.ii Changes to Toplevel 14](#_Toc395394533)

[3 Packaging and testing your IP 16](#_Toc395394534)

[3.a IP Packager (Within the Custom IP’s Vivado project) 16](#_Toc395394535)

[3.b IP upgrade in high-level design (Within the highlevel Vivado project) 19](#_Toc395394536)

[3.c Interfacing with the Custom IP 20](#_Toc395394537)

[4 Implementation Exercises 22](#_Toc395394538)

[4.a Timer implementation (32 bits) 22](#_Toc395394539)

[4.b FIFO implementation 23](#_Toc395394540)

[4.c GPIO implementation 25](#_Toc395394541)

[4.d Block ram implementation 26](#_Toc395394542)

[5. Conclusion 27](#_Toc395394543)

# 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 student 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 of this document will cover the following:

1. Setting up your Vivado high level design, focussing on the configuration of the Processing System.
2. Using Vivado’s built in tools to generate your own ‘Custom IP’, and showing you how to modify the custom IP on an ongoing basis.
3. Starts off with a tutorial on the AXI protocol, and then goes into critical modifications which allow for abstraction of the Slave AXI implementation. Which leaves you to concentrate on implementing hardware based solutions, and making the AXI communication process as simple as possible.
4. Teaches you how to package and upgrade your IP on an ongoing basis, and concludes with software implementations to test some basic modifications to your hardware.
5. The final section of this documentation consists of a series of implementation exercises designed to get you comfortable with using the custom IP, and through the exercises introduce ways of interfacing with AXI.

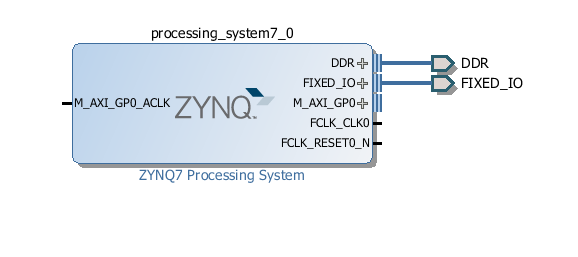
By the end of this document you should be able to generate your own components quickly, implement hardware solutions, and effectively utilise the AXI bus to get data to and from the PS to the PS. You’ll also become proficient in the flow of developing hardware within the Vivado framework, and learn methods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

# Highlevel design configuration

Firstly you’ll need to configure a high level design which features a ZynQ7 processing core along with its default pins assigned as well as some automation. For detailed instructions on the Vivado design flow please refer to lab1 of the Xilinx Advanced Embedded Design course, so you know how to work with Vivado, and design hardware on a high level basis (using IP blocks). In this lab we will replace the Xilinx AXI component by our own custom IP component, which will be programmed on a hardware level to provide some common hardware implementations including a timer, FIFO and GPIO, the details of which can be seen in section 4 of this document.

The starting point for this lab is the following high level configuration, (represented diagrammatically in figure 0.1):

* Instantiated a Zynq7 processing system (which has UART1 enabled)
* Applied block automation (without board pre-sets applied) to auto connect the DDR/FIXED\_IO pins to external pins
* Opened up this design on the “Open block diagram” to show the following

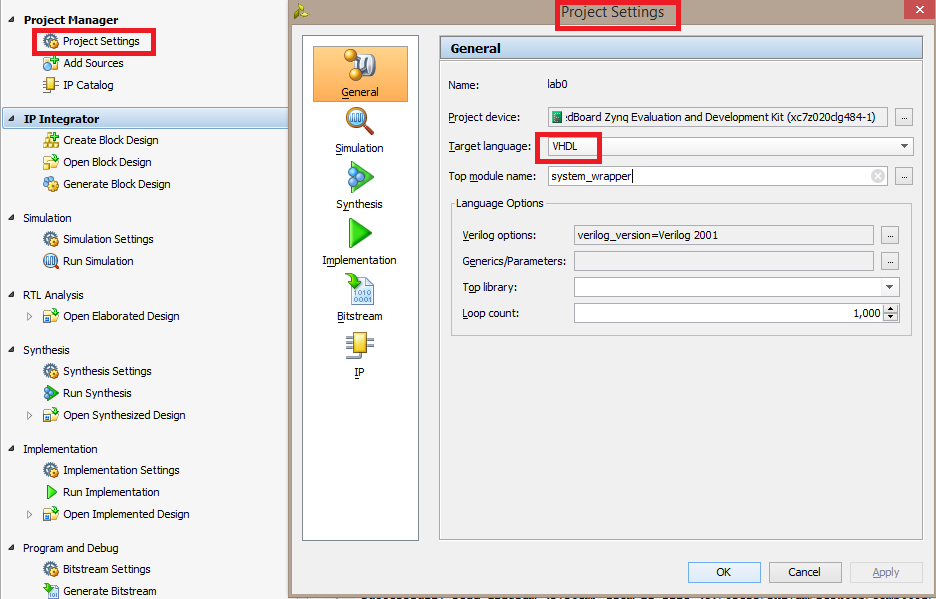


*Figure 0.1: Initial design*

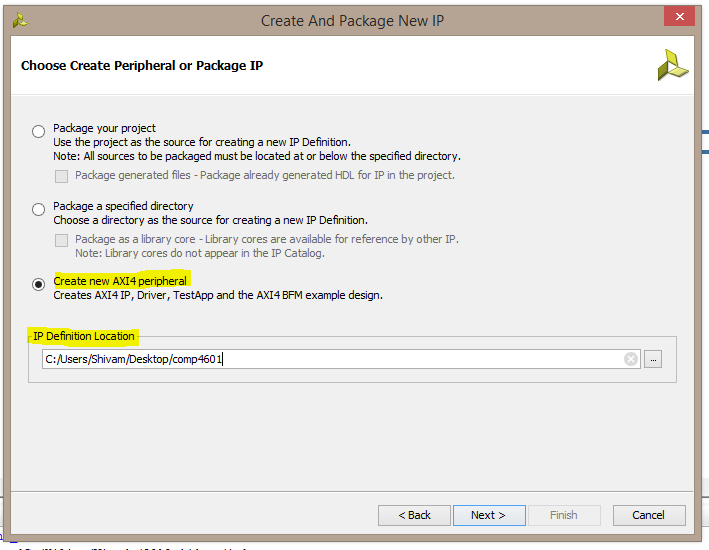
# Create custom IP

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

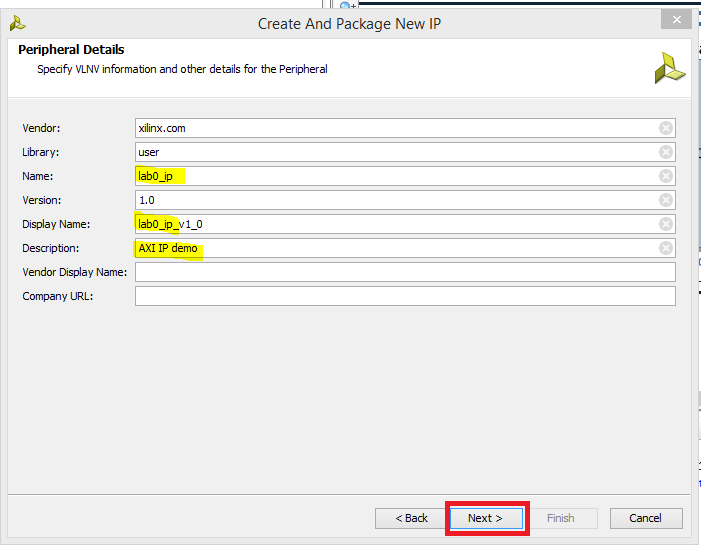
1. Click **project settings**, then ensure that the Target Language is set as **VHDL** (else the generated IP will be in Verilog), click OK when done.



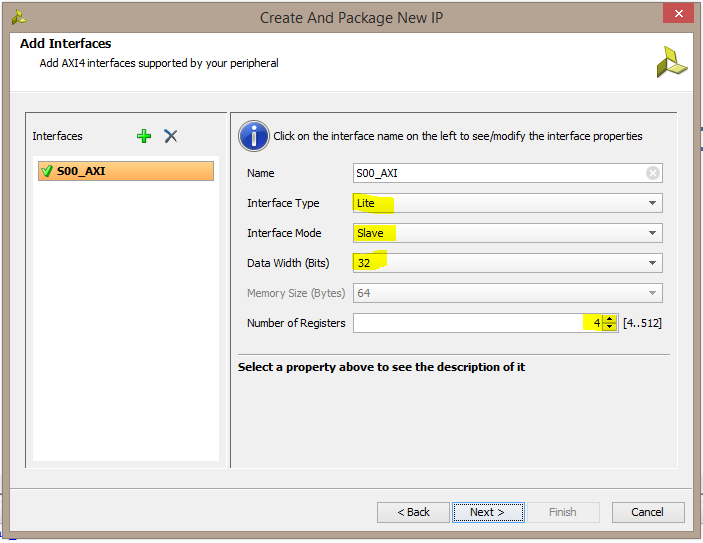
1. Go to the **tools menu** > “Create and Package IP”
2. On the introductory screen, select the **next** option.  
   Select “**Create new AXI4 peripheral**” and then in the IP location, go up one directory from where your high level project file is located; so your high level project directory and IP that we are creating will sit in the same directory (e.g. C:/…./XX/high\_level & C:/…/XX/IP)



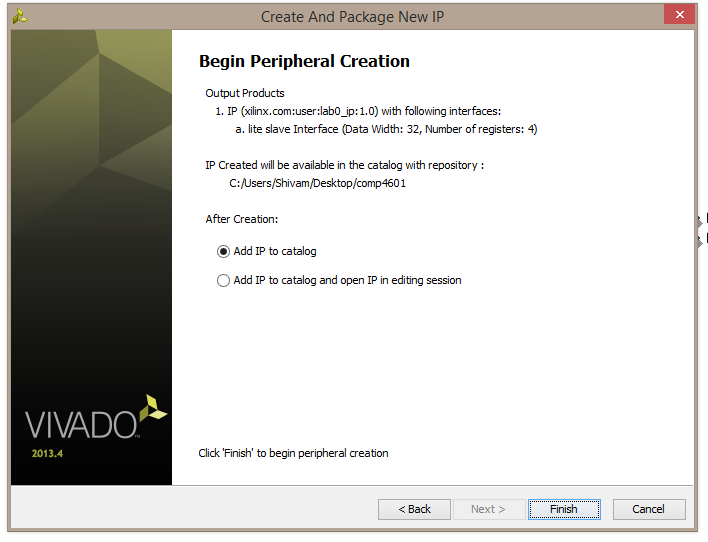
1. Now name the IP as “**lab0\_ip**”, updating the display name accordingly, as well as adding a more relevant description.



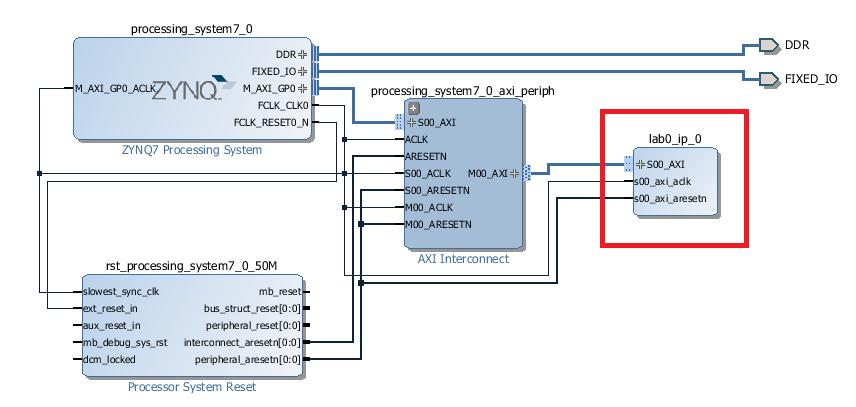
1. For the next menu we’ll keep the **default** options selected, here is an explanation as to why those options were selected:
   1. **Interface Type (LITE)** – LITE is the simplest of the three AXI protocols to program for, Full AXI allows for burst (4 packet at a time) transfer, while Stream offers continual data transfer, however both Full & Stream feature poor driver support on the PS, therefore we will stick to LITE.
   2. **Interface Mode (SLAVE)** – Since this IP is going to be issued commands by the processor this IP will act as a slave.
   3. **Data Width (32)** – Again for simplicity, we’ll keep the bus at the default width.
   4. **Number of registers** **(4)** – This option will affect the generated AXI code, with 4 registers the data transferred from Master to Slave will be stored in 4 unique registers, the lower 4 bits act as a multiplexing address (byte addressed, b0000 first register, b0100 second register, b1000 third register and b1100 fourth register, hence the MSB’s of the address are not utilised for a 4 register design)



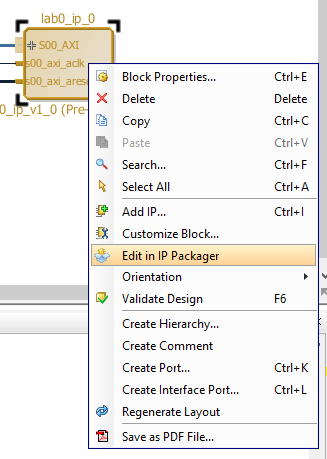
1. Click the **Next** button, when you are happy with this configuration
2. On the “**Generation Options**” screen, leave the options unchecked and click next.
3. Leave “Add IP to catalog selected” and hit **finish**



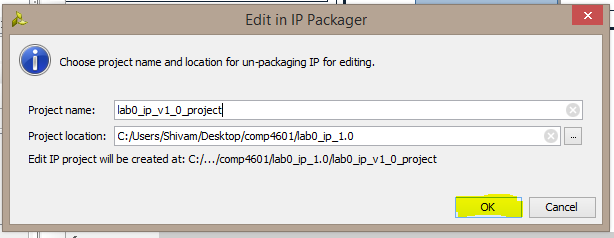
1. On the block diagram (with **only** the ZynQ7 IP instantiated), select “**Add IP**” and find the lab0\_ip that you just created.
2. Select the “**Run connection automation**” to the s00\_AXI of the custom IP just created, the end result should be as follows:



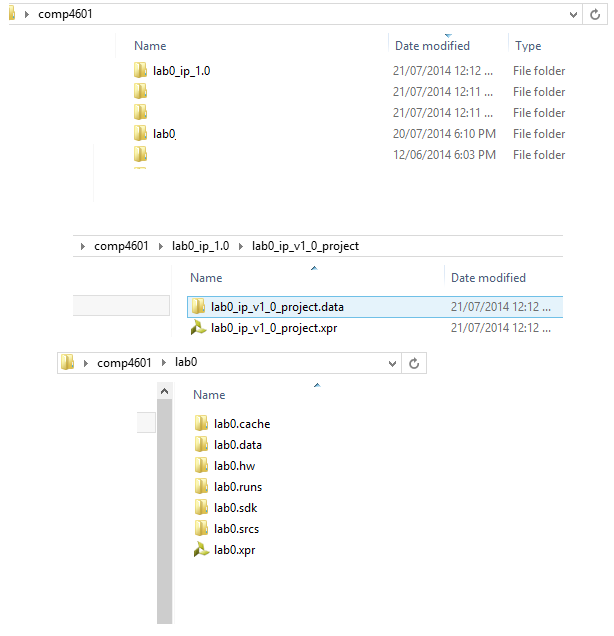
* 1. **Save** your block design and/or project file
  2. Right click the lab0\_ip\_v1\_0 (custom IP) in your design and select “**Edit in IP Packager**”



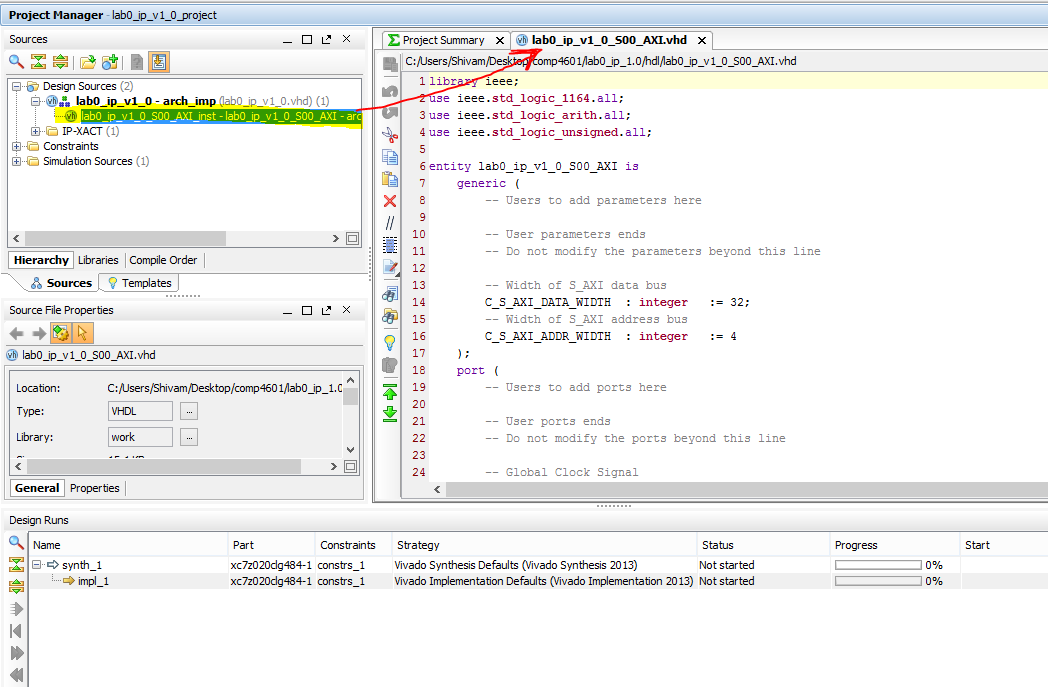
* 1. Select “**Ok**” in the project location screen



* 1. 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.
  2. 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).



* 1. 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.
  2. **Open** the VHDL file called “lab0\_ip\_v\_1\_0\_S00\_AXI.vhd”, from the project manager view.



# Customising the ‘Custom IP’

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

## 2.a AXI Tutorial

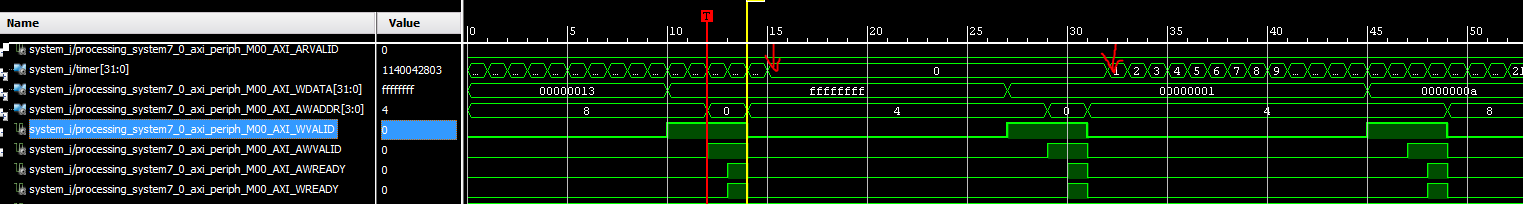
* Advanced eXtensible Interface (AXI) is a protocol developed by ARM which is a mechanism for controlling shared bus access. Some of the key features of this protocol are as follows:
* Separates address, control and data lines
* Incredibly simple handshaking, due to the separate control lines
* Burst mode transfer supported with the provision of only a starting address
* Uses a Master Slave model, with the Master is solely responsible for the arbitration of the bus, directing writes and requesting reads from the Slave
* See the “AXI Reference Guide” [2] for full documentation of the AXI protocol

The Master accesses the slaves by loading the address bus with an address that is within the slaves assigned address range. Because the address bus is shared between 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 the transfer of data.

However when you run connection automation on your custom AXI IP, Vivado inserts an AXI Interconnect in between the real master (Processor) and the Slave IP. See the “Xilinx AXI Interconnect documentation” [3] for the implementation of the interconnect, but essentially it does most of the heavy lifting on arbitration and implementing a bus like nature through 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 below) due to the internals of this interconnect the details of which can be seen in the “Xilinx AXI Interconnect documentation” [3].

## 2.a.i AXI Writes



4

3

2

1

The above waveforms show the master writing 0xFFFFFFFF @ BASE\_ADDR (0x0) then 0x00000001 @ BASE\_ADDR (0x0), and finally 0x0000000a @ BASE\_ADDR+4 (0x4).

How AXI WRITES work:

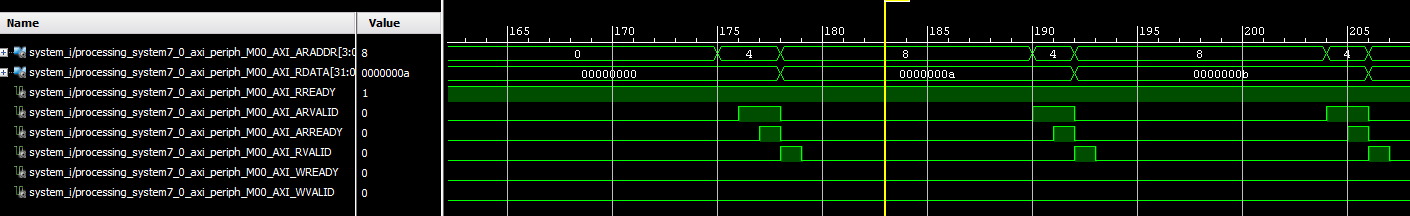
1. Master\* sets up WDATA asserts **WVALID** (write data is valid)
2. Master\* sets up AWADDR and asserts **AWVALID** (master asserting that it has placed the valid address on the address bus)

\*By Master Asserts – strictly speaking it is the AXI interconnect which acts as the master for this slave AXI component.

Slave then does the following:

1. Asserts **AWREADY** (write address can be accepted by the slave, determined by WVALID && AWVALID)
2. 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)
3. Once WVALID & AWVALID & AWREADY & WREADY are all asserted
   * Slave write is enabled
   * Next clock cycle (14cc in diagram, yellow line) the slave register (slv\_regX) has data on WDATA bus written into it

## 2.a.ii AXI Reads



4

3

2

1

In the above example the processor is reading from a FIFO which contains the data {0x0a, 0x0b, 0x0c, …..} from the custom IP via AXI-LITE at the address of BASEADDR+4 (0x4).

For AXI Reads there are 4 basic signals to worry about:

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

The slave then performs the following:

1. The Slave asserts ARREADY to signal that the address can be accepted by the slave
2. Slave then sets RDATA to reflect the appropriate data, and asserts RVALID upon which ARREADY is de-asserted and the read is completed by the master, at this point (178cc) the correct read data is placed onto the bus, where it has a limited amount of time to be read by the master. The RVALID signal is then de-asserted exactly one clock cycle later.

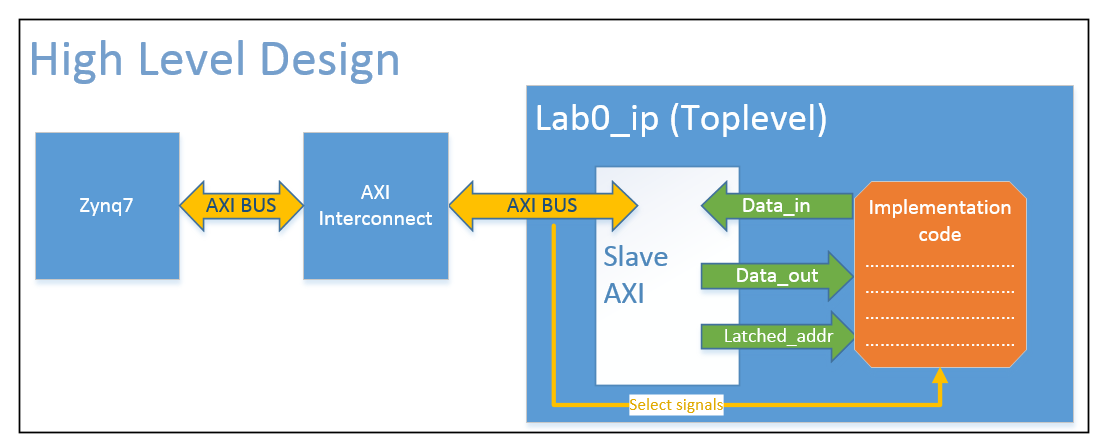
## 2.b Customising the custom IP

Based on the tutorial on the AXI protocol above it should be clear that these signals can used by the programmer to determine whether or not a read/write has been placed by the master and to determine if certain actions on the slaves end should be undertaken. There are a number of ways to which this effect can be achieved, and we will introduce these by simple projects involving the implementation of a hardware timer, hardware FIFO and hardware GPIO which allow communication to the physical buttons and LED’s (detailed in section 4).

Before we begin there are some general modifications which will greatly speed up the design process, they are as listed in the following section. Just a couple of files and naming conventions to take a note of before starting:

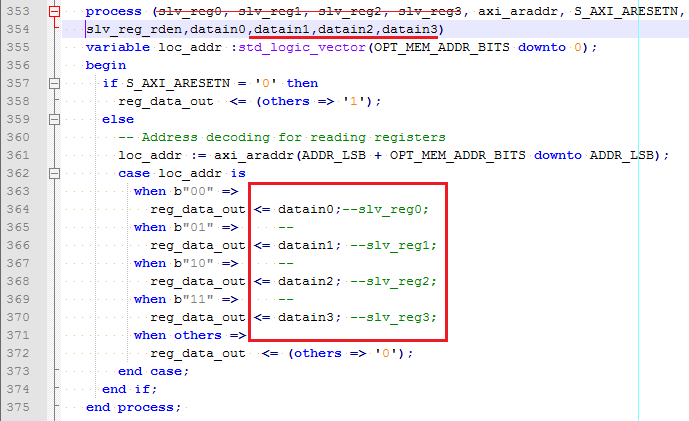
* **Slave\_AXI** (lab0\_ip\_v1\_0\_S00\_AXI.vhd) – generated file which implements the AXI-LITE handshaking process and stores all writes into registers, and uses those same registers as read response values.
* **Toplevel** (lab0\_ip\_v1\_0.vhd) - refers to the VHDL file which encapsulates the AXI implementation file described above, you’ll notice that is largely empty, and 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 major sub VHDL components, however given this is a relatively small lab we will code entirely within this file for convenience.

The diagram below shows the overall process to our design methodology, as we can see the high level project file that contains the IP blocks (Zynq, AXI Interconnect and custom IP) all the way down to the lab project file which consists of the files mentioned above. The green arrows in the diagram below denote the changes we will be making to bring out some of the signals from the slave AXI file into the toplevel where we can then code our implementations. The reason for this is so that we preserve the protocol implemented by the Slave AXI file, and concentrate our efforts towards the production of the non-AXI related hardware (implementation code).

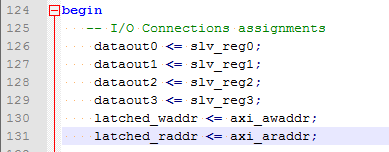


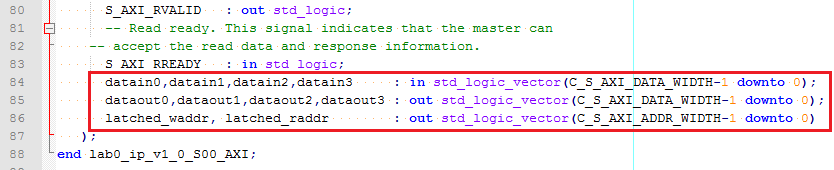
### 2.b.i Changes to Slave\_AXI

You should notice that the output values for the AXI reads are driven by reg\_data\_out. These signals are originally being driven by the slave registers (which are the registers the AXI write data is stored into). Since this is a pointless function we’ll replace it with our own signals (datain0/1/2/3) which will later be declared as inputs to the Slave\_AXI entity.



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. Notice from the timing diagrams in section 2.a/b 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.



As a consequence the following signals (image below) 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.

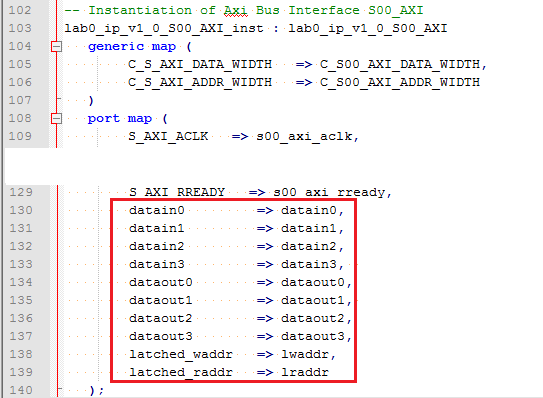
### 2.b.ii Changes to Toplevel

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

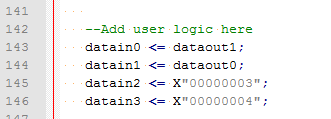
* Firstly you can simply use the existing register implementation in the generated AXI protocol and read from the registers. This will work if your implementation is not dependent on user actions and simply runs on its own based on the values the user has sent to you. While for AXI-reads static values can simply be inserted and read by the user on a needs 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 bus lines for data, thus being able to realise when the master has evoked an action and being able to react accordingly. Of course the actual AXI implementation can be modified to achieve this same goal, however to keep the complexity down to a minimum and prevent difficult to debug problems whereby the protocol has been implemented incorrectly we’ll stick to the snooping based approach.

We’ll come back to these ideas in the section 4 for implementation of the designs, but for now we’ll stick to some simple modifications so that we can easily determine if the changes that we have made to the source of the IP are carrying through to our high level designs.

Next modify the port map of the Slave\_AXI component within the Toplevel file, to add signals to the new ports which we have just added in. The following diagram shows the modifications that were made to the port map; also remember to declare the corresponding signals on the right hand side.



Now we’ll implement the following, so that we can test 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 + 8/12, the constant values of 3 and 4 will be read from these addresses regardless of the values have been written.



Within the IP project file, click on the synthesise button in the left hand pane, the reason for this is to pick up on any silly compilation errors, 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 section 3).

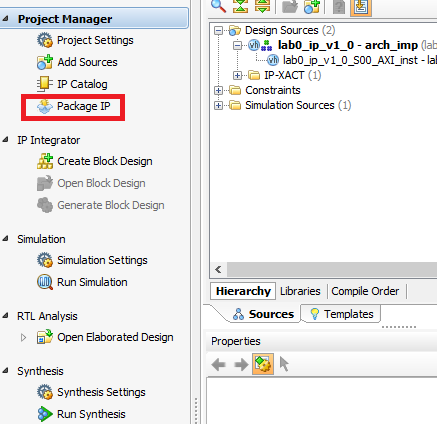
The importance of following the steps outlined in section 2 is that you no longer have to worry about the slave\_AXI implementation file, since you have catered for all signals that will be useful to your implementation. From now on you are free to modify only the toplevel and add sub components in the toplevel that you instantiate.

# Packaging and testing your IP

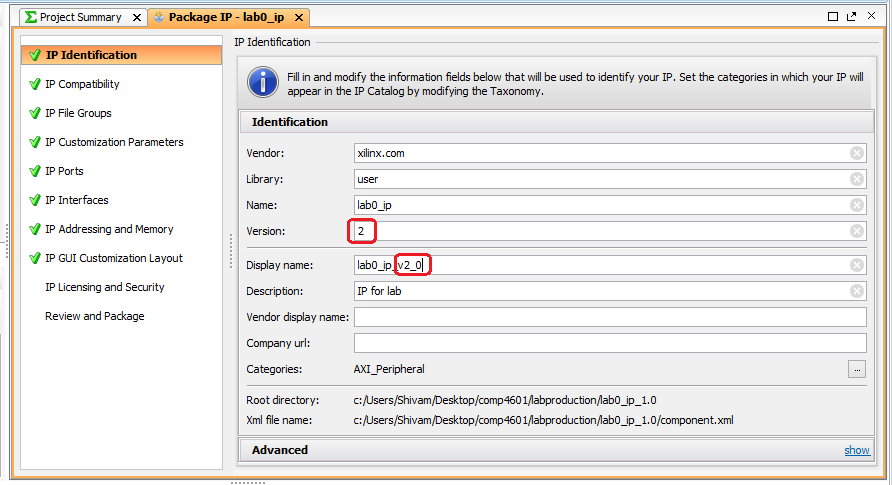
In this section we will now package the IP that we have generated and modified, once packaged it can be used in the high level design as an independent IP block, exactly like you would treat other such IP blocks. Then all that remains is to write some C driver like code to interface with the IP and ensure that the changes that we have made are working. Note 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**.

## 3.a IP Packager (Within the Custom IP’s Vivado project)

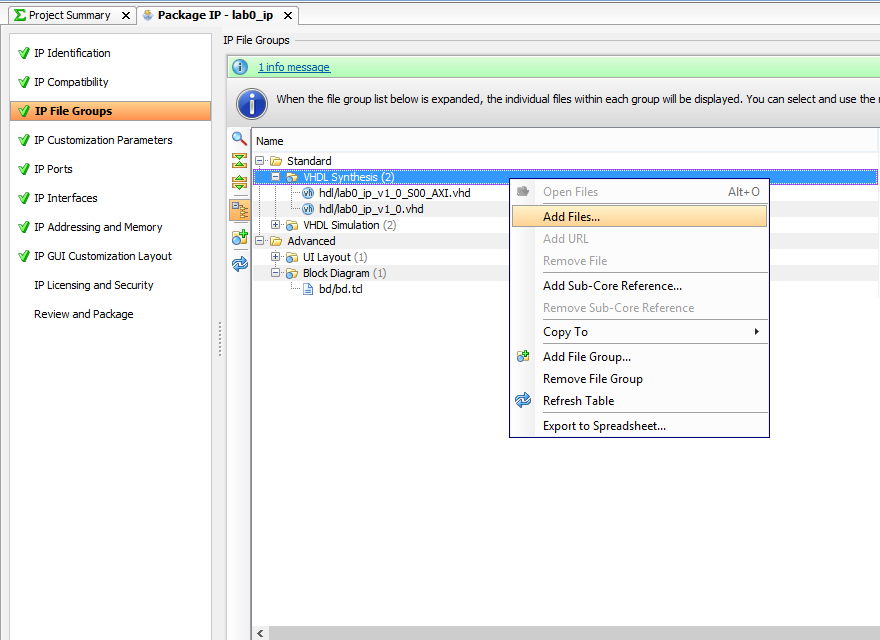
1. Select the **Package IP** in the project manager section in the left hand pane.



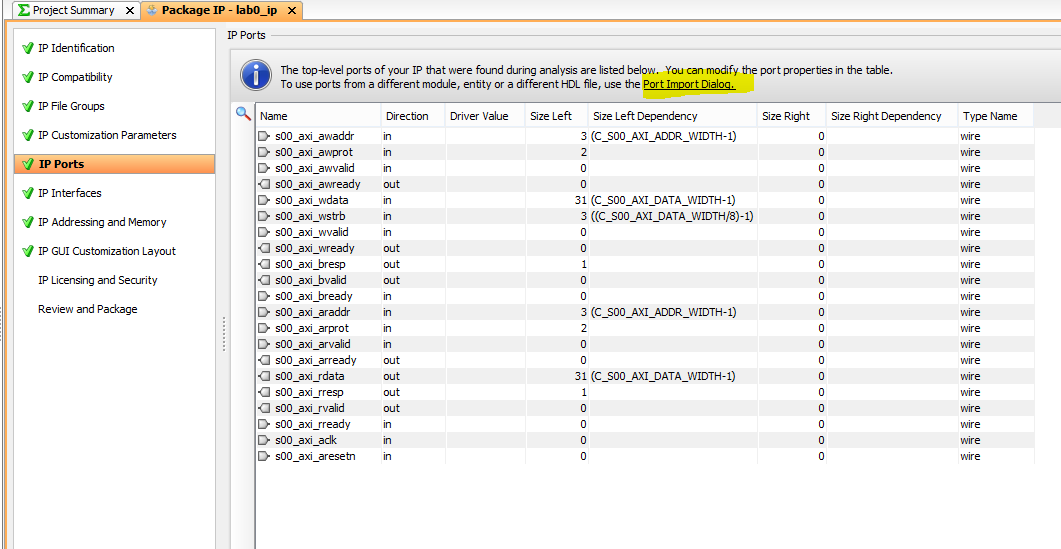
1. At the starting screen leave all the options the same except the version number; ensure that you **INCREASE** the version number (e.g. 1.0 -> 2.0). The reason for this is so that Vivado will detect the version change, and prompts you for an upgrade. Also alter the display name to reflect the version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



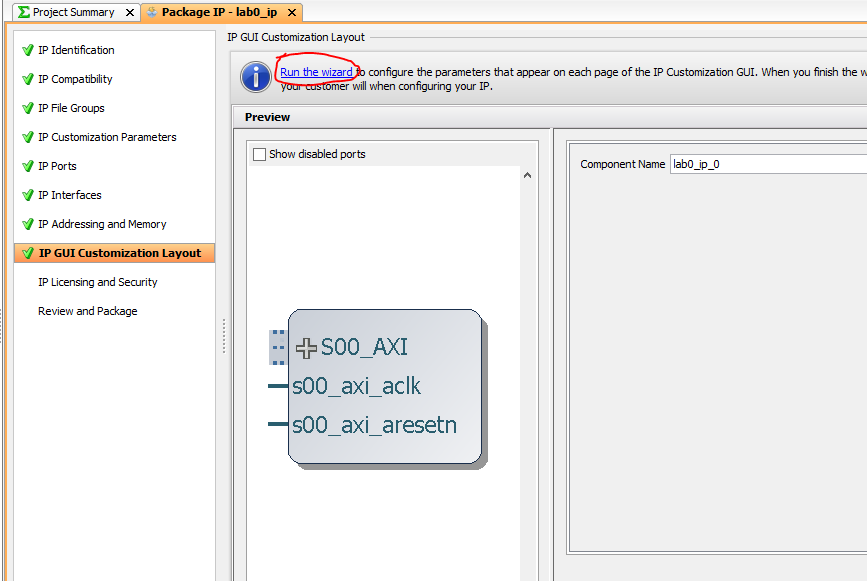
1. **If** the changes to the file involved adding new VHDL files, they must be added in the **“IP File Groups”** to both the “VHDL synthesis” and “VHDL Simulation” folders, as shown by the diagram below.



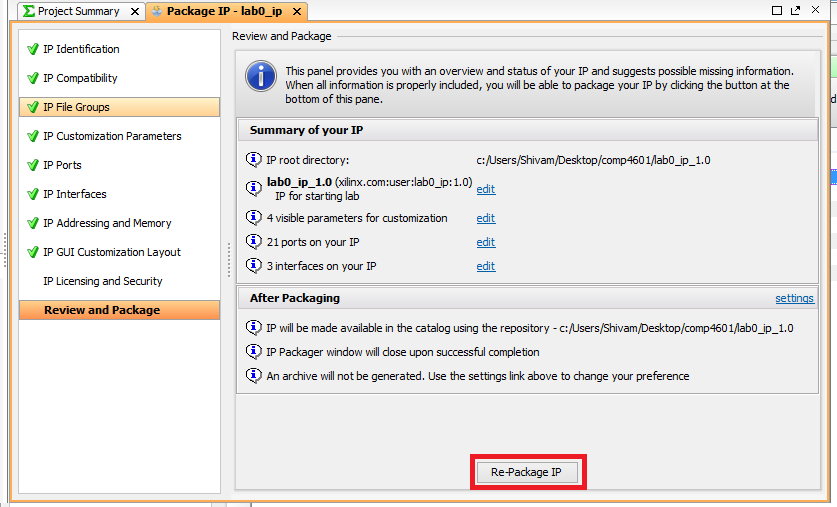
1. **If** the ports have been changed, then use the “**IP Ports**” page, simply clicking on the Port import dialog and follow the prompts.



1. **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.



1. To complete the process, in the “**Review and package”** screen select Re-Package IP

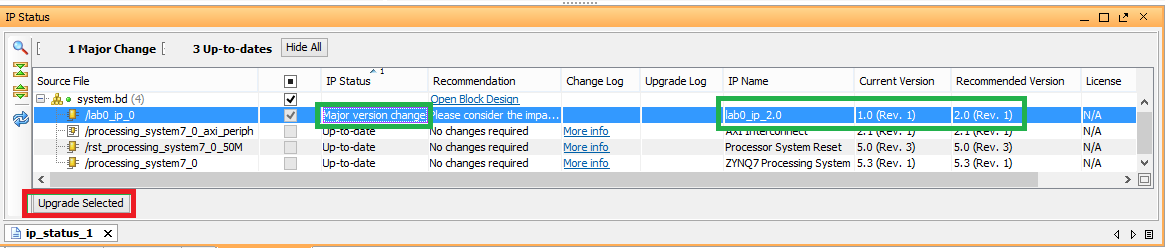


Screens which were skipped:

* **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

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

1. Now **reopen the high-level design Vivado file** and open the Block Design
2. Select the **TCL console** window and run the following commands
   1. **update\_ip\_catalog –rebuild**
      1. This refreshes the IP repositories specified in Project Settings > IP > IP Repositories (you can do this manually if you wish)
   2. **report\_ip\_status -name ip\_status\_1**
      1. This generates an IP report, showing whether or not the IP in your design are up to date
3. This command should report that lab0\_ip\_0 has a “Major Version Change” (can also be minor, depending on the version number you selected)



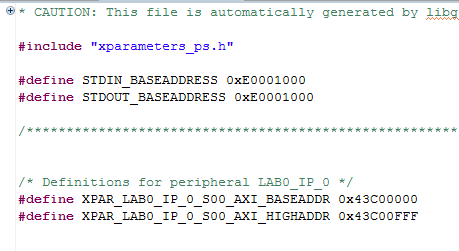
1. Ensure it is selected and **hit upgrade selected** button (Vivado will now upgrade the IP, retaining all existing connections)
2. Regenerate the HDL wrapper for your high level design
3. Hit the **Generate bitstream** button (which should also synthesise and implement your high-level design) and wait for this to finish
4. Open Implemented Design
5. File > Export > **Export Hardware for SDK**

## 3.c Interfacing with the Custom IP

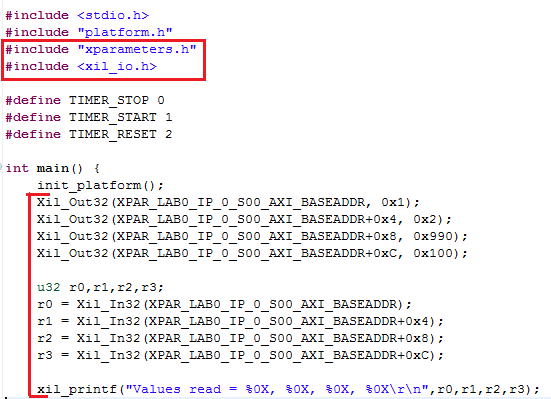
Now that we are in the software side of things, all that remains is to write some interface code and test the functionality of the hardware.

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). You should also “**#include <xil\_io.h>**” to get the Xil\_OUT32/Xil\_IN32 function definitions.



Once you’ve verified this, go back to “**helloworld.c**” and write some simple source code to the effect shown 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 this lab correctly.



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 includes 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 exact size defined data types such that any ambiguously sized data typed do not cause unnecessary overflows.

# Implementation Exercises

Now that you are across 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 now work out some more useful implementations on the hardware side. All exercises are intended to be implemented in the toplevel file of the custom IP. The protocol for this lab is shown by the following diagram:



## 4.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 the read requests for the value from the user.

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

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 slv\_reg0 (**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 bus 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 value.

Appendix A contains the solution to the timer, in case you wish to verify your code prior to compiling.

## 4.b FIFO implementation

Specification:

* FIFO of data width **16 bits**, Address width of **1024 words (16 bit)**
  + This should be implemented as block ram in your toplevel. If you need a refresher as to how to go about this, please refer to this excellent guide titled “Distributed and Block ram on Xilinx FPGA’s” [5]
* If we reach the end of FIFO addressing, writes/reads should simply wrap around
* Similarly the read pointer should not be advanced when we don’t have any data for it
* 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 |  | n/a | ….. | Data | | | |



The diagram above 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 **slv\_reg1**, more details to follow.

Implementing a FIFO will be a little 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 and (or slv\_reg1) and set that to be the value for FIFO input as well as enabling the FIFO write for exactly one clock cycle. Referring back to section 2.a.i and the original source code 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 read the data bus at this point) and once so, enable a write to the FIFO.
* **Read –** From the timing diagrams it should be apparent that there is only one clock cycle 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, instead set up the **next read** value after the read has taken place. Referring back to section 2.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 to be asserted on the rising edge of the clock (this point will be the falling edge of the RVALID signal), the read will have taken place and we can safely replace the value of **datain1** to point to the next value in the FIFO.

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 you’ll notice that the address is only valid for a very small amount of time. 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 note move read ptr.* |



|  |  |
| --- | --- |
| *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.* |

The FIFO diagrams above 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 over data which 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 over fill 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 C drivers 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.

## 4.c GPIO implementation

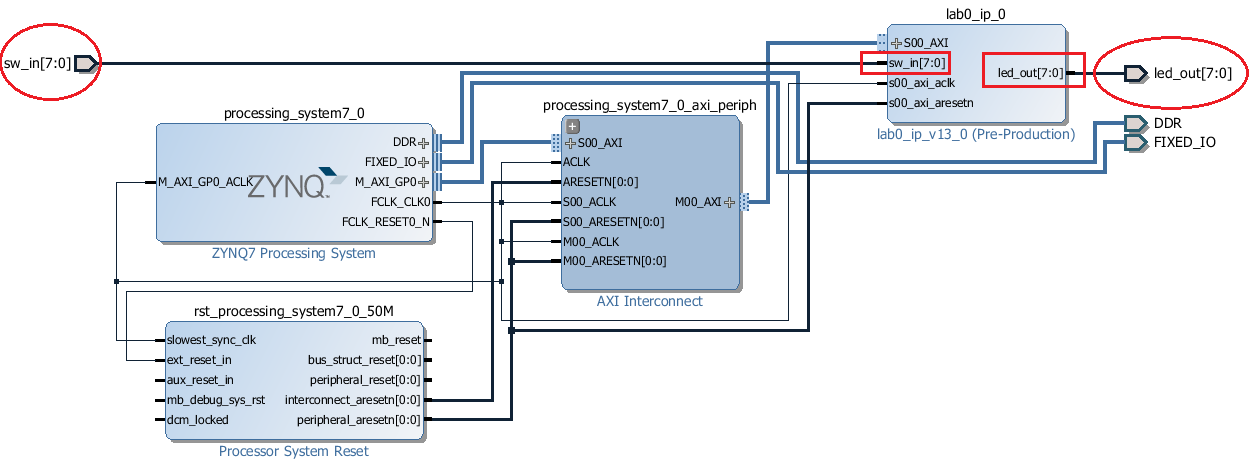
For the GPIO implementation we will mimic the functions of the advanced embedded systems Lab 1 however we will do it entirely through our custom IP component, so that we have a better idea as to what is going on. This process will involve the adding of ports to your IP as well as external pins and constraints 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 slv\_reg2 (**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 he 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 lab1 for how to declare in xdc file and “Zedboard user manual” [4] for the Switch pin numbers). The following diagram 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.



## 4.d Block ram implementation

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

Assumptions:

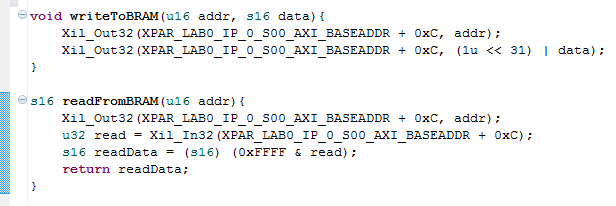
* BRAM which consists of shorts (16 bits)
* Has 2^16 addresses to write to
* Master has control over the whole BRAM, can read and write to any address in the BRAM

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

**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:



# 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 now 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.
* **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 the 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 it 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**;**