The aim of this lab is to introduce a design flow which allows you to create your own custom Intellectual Property (Custom IP) using Xilinx’s Vivado. Following this this lab introduces how to modify the generated component, by focusing on how the AXI-LITE protocol works and how it can be utilised to establish a circular data flow between the processor and the hardware component. This lab concludes on methods of maintaining and integrating this IP as part of a larger design.

The protocol which will be implemented by this lab is as follows:



# Configure Highlevel design

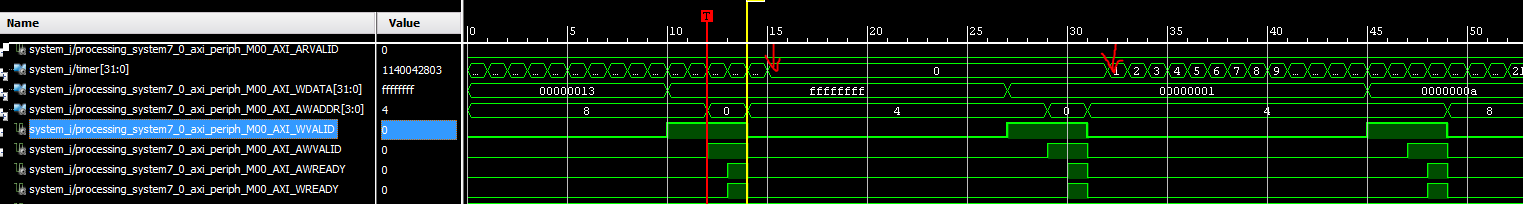
For this please refer to lab1 of advanced embedded design so you can create a Vivado design file with an ARM processor alongside the GPIO AXI component, which will be replaced by the steps in this lab.

# Create custom IP

… pictures and steps to follow …

# AXI tutorial

Once the custom IP has been generated please take about 10 minutes to read through the implemented “<WHATEVER\_YOUR\_IP\_IS\_CALLED>\_S00\_AXI.vhd” file, to see how the AXI-LITE interface is implemented on the SLAVE. We will now walk through this implementation and how the AXI-LITE process in general works.



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

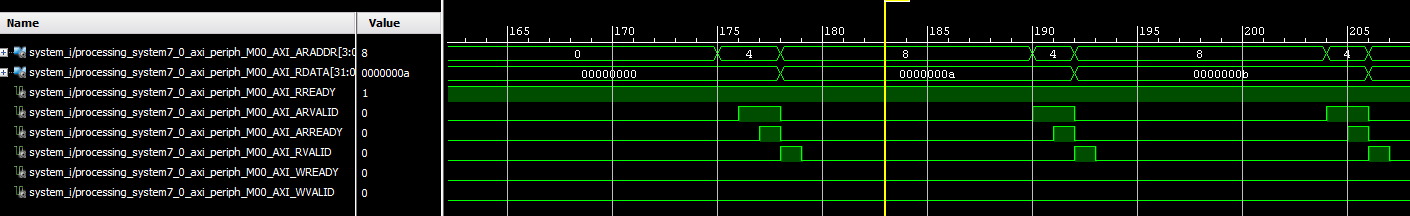
How AXI WRITES work:

* Master\* asserts WVALID (write data is valid, we can see that it is asserted the clock cycle 0xffffffff the data to be written is seen on WDATA)
* Master\* asserts AWVALID (master asserting that it has placed the valid address on the address bus, seen on AWADDR becoming 0)

\*By Master Asserts – strictly speaking it is the AXI interconnect which acts as the master for this slave AXI component, the difference between the interconnect and the processor being the interconnect multiplexes the WADDR/RADDR and only asserts the valid signals for the relevant slave (since there are often multiple slaves connected to the same interconnect)

Slave then does the following:

* Asserts AWREADY (write address can be accepted by the slave, determined by WVALID && AWVALID)
* Asserts WREADY (write data can be accepted by the slave, determined by WVALID && AWVALID)
* Also the write address is also latched (stored)
* 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
  + Because our timer implementation reads the slave register value on the rising edge the next clock cycle (15th clock cycle) it realises the value is 0xFFFFFFFF the trigger value to reset the timer, and we can see the timer being reset here



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.

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

* Master pretty much always has the signal RREADY asserted, signalling it is always able to receive requested data back from the slave
* Master then places the address that it wants to read from onto the ARADDR bus and asserts ARVALID

The slave then performs the following:

* Slave the firstly asserts ARREADY (to signal that the address can be accepted by the slave, determined by ARVALID only) and then is upon this signal that the
* Slave then asserts RVALID upon which ARREADY is de-asserted and the read is completed by the master, at this point 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.

Other signals which can be utilised but are not included in your own designs are:

* BRESP/RRESP – Write response/Read response: AXI allows for a return of value by the slave upon every write/read.
* BVALID & BREADY – used in the same nature the AXI read valid/ready signals are, with the master asserting the ready signal when it can accept a response (pretty much all the time) and the valid signal being asserted by the slave when there is valid data on the BRESP bus.

## 2.b Customising the custom IP

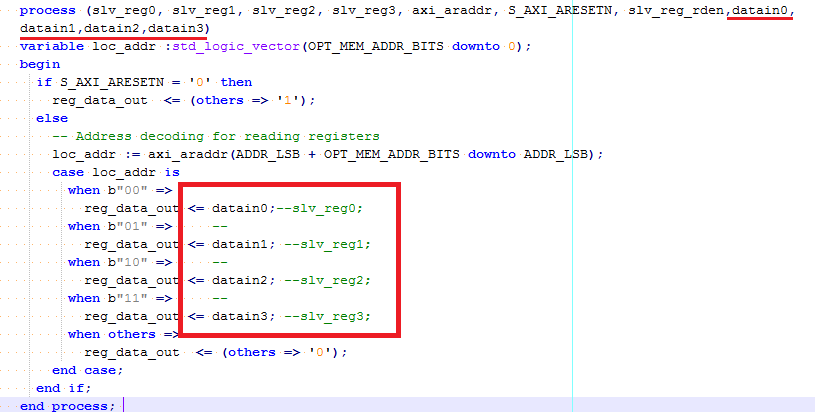
Based on the guide above about the AXI protocol above it should be clear that the signals can effectively be snooped upon by the programmer to determine whether or not a read/write has gone down and to determine which action to undertake as a result. There are a number of ways to which this effect can 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.

Firstly however 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 things to take a note of before starting:

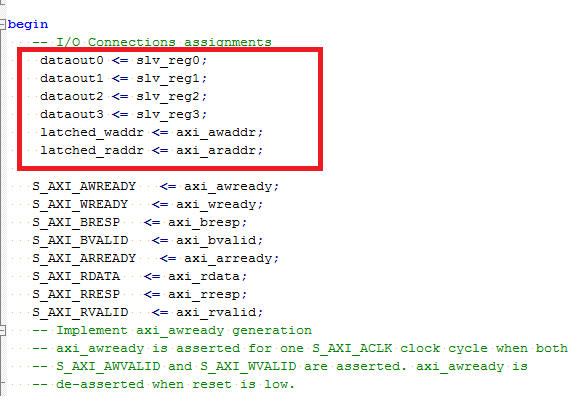
* “<your\_ip\_name>\_S00\_AXI.vhd” – generated file which implements the AXI-LITE handshaking process and stores all writes into registers, and uses those same registers as read response values.
* Toplevel refers to the toplevel VHDL file which encapsulates the AXI implementation file described above, you’ll notice that is largely empty, and is where we will be programming most of the code here. 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 tutorial we will code within this file for convenience.

### 2.b.i Changes to “……\_S00\_AXI.vhd”

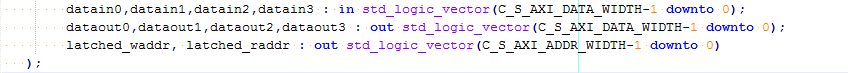
Firstly you should notice that the output values for the AXI reads are driven by reg\_data\_out, but these signals are originally being driven by the slave registers (which are the registers the AXI write data is stored into), since this is pointless we’ll replace it with the following signals (datain0/1/2/3).



Now that we have taken care of the output values, we next need to worry about getting the AXI written values out of this component as well; hence we can simply pipe these values (since they are all implemented as registers) out of this generated AXI component onto the toplevel where they can be used.



As a consequence the following signals have to be added to the definition of the S00\_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 get 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, 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 dynamically accordingly. This is quite useful when you want to implement a method whereby your IP can be in a data transfer state, then switch to a processing the data and transferring back data; as is a common implementation pattern in hardware.

# Implementation

I’m thinking of walking through the implementations of the following components…

## 3.a Timer implementation

## 3.b FIFO implementation

## 3.c GPIO implementation

## 3.d Block ram implementation

Leave this as exercise, however provide the protocol to implement (and solution is provided).

# Saving your IP

## 4.a IP Packager

## 4.b IP upgrade in highlevel design

References

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>