Lab 4: 32-bit MIPS CPU

In this Lab, we are going to implement a 32-bit CPU, based on MIPS32 instruction set architecture (ISA). If you are not familiar with RISC-type instruction set, you should refer to the MIPS references provided with this Lab.

CPU Description

Our CPU design itself will be based on the figure below, which is taken from the Hennessy and Patterson text. It's based on Multi-Cycle CPU, as will be discussed in the lecture. The CPU will be composed of a Control Unit (highlighted in blue) and a Datapath (everything else, except for the memory block highlighted in red). This similar to what we did in Lab 3, except the datapath and the control unit are much complex now.

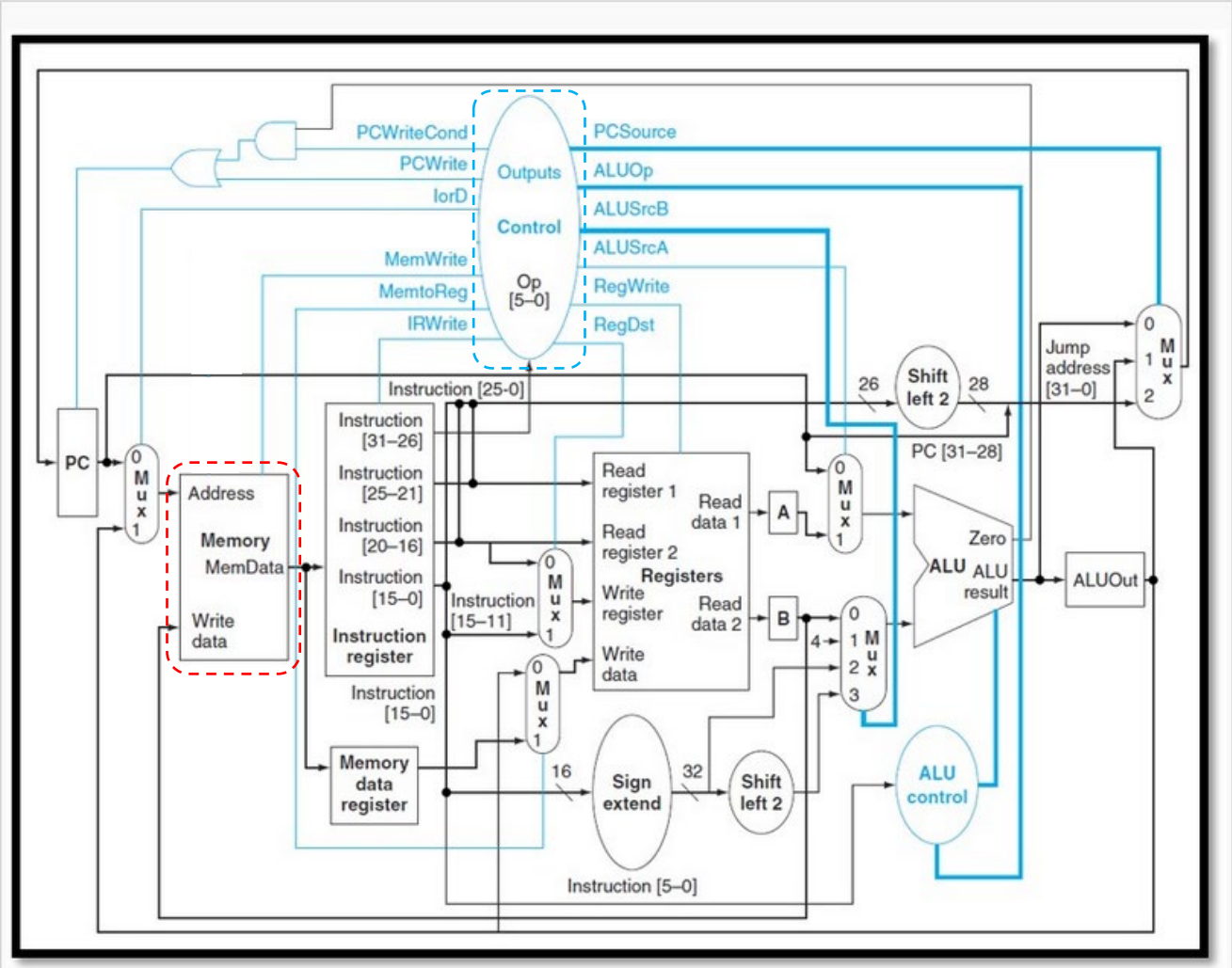


Figure 1: Entire MIPS Computer block diagram (Hennesy and Patterson)

As you all know, any CPU needs a memory block to be interfaced with. This is the memory block, highlighted in red, in the Fig. 1. This memory block will be divided to both Instruction memory (where the program instructions are stored), and Data memory (where any results will be saved). You will be provided with this memory block, and you **DON'T** have to implement it.

In order for the CPU to be correctly interfaced with the provided memory, the top-level CPU should be as below.

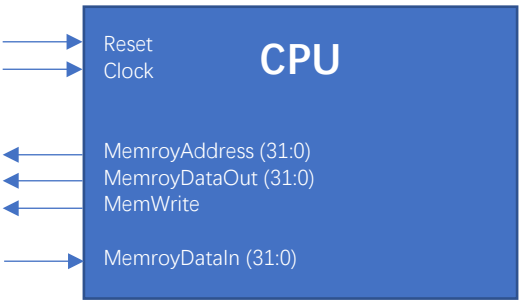


Figure 2: CPU top level structure

You may have noticed that the pin out of your CPU is almost the same as the memory components in Fig. 1. This is because that schematic shows an entire MIPS based computer system which only requires a clock input. In our design, you are implementing a processor that accesses an external memory, hence the memory access input and output pins on the symbol. A more detailed description of the ports is shown in Table 1.

Table 1: CPU ports description

Port	Direction	Description
Reset	In	Asynchronous active high reset for CPU (single bit)
Clock	In	Global clock signal (single bit)
MemoryDataIn(31:0)	In	32-bit input bus carrying the data from memory.
MemoryAdress(31:0)	Out	32-bit output bus carrying the memory address.
MemoryDataOut(31:0)	Out	32-bit output bus carrying the data to be written to memory.
MemWrite	Out	Write enable signal for the memory, active high (single bit)

1. CPU Components

This section will provide some guidelines on most of the components in Fig. 1. Note that the information mentioned here is just a guideline, you will likely need other components, depending on how you approach the design.

A. ALU:

This is the ALU designed in Lab 2, you are going to use this component.

B. ALU Control:

This is a simple asynchronous block that would map between the op-code coming from the Control Unit and the ALUOp bits of the ALU controlling which operation to choose. You have the option of implementing this as a separate block, or merge it with the Control Unit and control the ALUOp bits of the ALU directly.

C. Register File:

This is a dual-ported register file that you can read from the two read ports concurrently. Each read port is 32-bit wide and has an associated address port, which is 5-bit bus. Hence, the register file should have 32 registers. The write data and write address are used, along with the RegWrite signal to write data to a register file (see Fig. 1). The only pins that are not shown in Fig. 1 are the reset and clock pins.

D. Sign-extend, Shift Left:

These are asynchronous blocks to sign extend an operand, or shift left by certain number of bits. They come in different variations, depending on where they are used

E. Registers, Muxes:

Various synchronous registers and asynchronous muxes are used in the CPU. Both components are used in different sizes, so it's smart to design one of them as generic and instantiate them with different sizes. Examples of the registers in Fig. 1 would be: Instruction Register, PC, A, B, ALU Out, ….

F. Multiplier

This is the multiplier implemented in Lab 3, and it's not shown in Fig. 1. It should be added to the Datapath, along with any needed extra registers or control signals, as we are going to implement a multiplication operation.

G. Control Unit

This is the brain of the CPU, where you would decode an instruction and assert the corresponding control signals. This should be implemented as a Finite State Machine (FSM). As mentioned above, this is a multi-cycle CPU, which means that instructions execution can vary in the number of clock cycles. However, except for the multiplication operation, **all instructions must execute in 5 clock cycles or less.**

Implementation Notes:

a) Build and test your design incrementally.
b) First design and test all of the Datapath components separately.
c) Implement the Datapath first, then add control later on.
d) Start by implementing a simple R-Type instruction (ADDU for example), do the Control Unit for this one, and test it thoroughly. It's very likely that the rest of the R-Type instructions will need minimal change in your Control Unit or Datapath. So, go back, add those instructions, modifying your Datapath/Control Unit as necessary.
e) Test new instructions as you add them. Don't hold the testing towards the end. The design will get very complex for you to debug.
f) Test using Tcl files first, setting the 32-bit instruction that you are testing, and make sure all the internal signals, memory address bus, register files, and memory data bus are exactly as you want. Once you are done with all the instructions, you can test with the provided memory component.

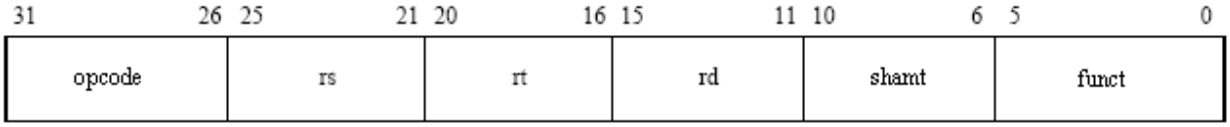
2. MIPS Instructions

This section will provide information about the instructions required to be implemented. We are not going to implement the whole ISA, just a subset of them in each category. All instructions in the MIPS ISA you are implementing are 32 bits in length.

A. MIPS Instruction Types

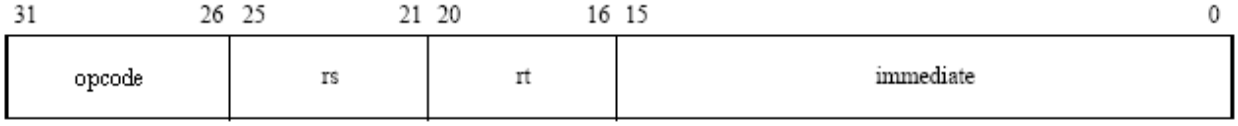
There are three different instruction formats: R-Type instructions, I-Type instructions, and J-Type instructions. R-Type instructions, or Register instructions are used for register based ALU operations. The two operands and the destination of the result are specified by locations in the general-purpose register (GPR) file. I-Type instructions, or Immediate instructions, can be either Load/Store operations, Branch operations, or Immediate ALU operations. In these instructions, register file locations are specified as well as a 16-bit immediate value which may be used as an operand or an address. The *rs* and *rt* register addresses, which are present in both R- and I-type instructions, specify the two addresses which the register file is to read. In R-Type instructions the destination (write) register for the register file is specifies by *rd* and in I-Type instructions the destination register is specified by *rt* (the second read from the register file is ignored). Finally, J-Type instructions, or Jump instructions, devote all of the non-opcode space to a 26-bit jump destination field.

R-Type Instruction Format



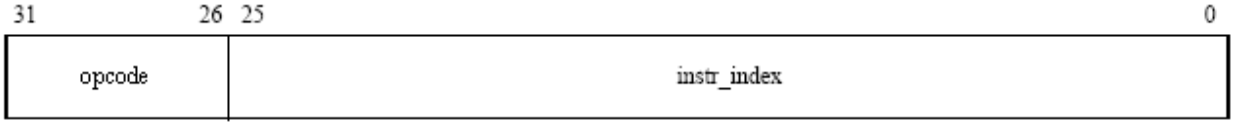
opcode: Instruction encoding constant (6 bits)
rs: Operand located in general purpose register file (5 bits)
rt: Operand located in general purpose register file (5 bits)
rd: Destination location of result in general purpose register file (5 bits)
shamt: Shift amount (5 bits)
funct: Function code (6 bits)

I-Type Instruction Format



opcode: Instruction encoding constant (6 bits)
rs: Operand located in general purpose register file (5 bits)
rt: Destination location of result in general purpose register file (5 bits)
immediate: Immediate operand value (16 bits)

J-Type Instruction Format (Jump)



opcode: Instruction encoding constant (6 bits)
instr_index: Destination address for jump operation (26 bits)

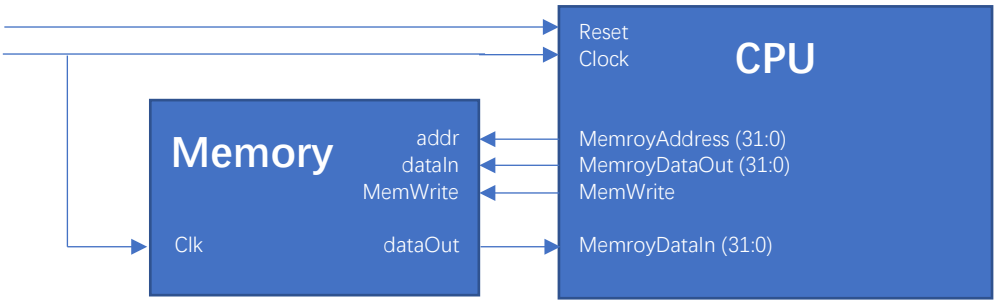
You are only required to implement a subset of the MIPS instruction set. Table 2 lists all required instructions. Please refer to the MIPS32 Instruction Set manual, provided with this Lab**,** to see descriptions of these instructions and their encodings.

Table 2: MIPS Instructions

ADDI	**ADDU**	**AND**
ORI	**SUB**	**SRA**
SLLV	**SLL**	**SLTI**
LUI	**SW**	**LW**
LH	**LB**	**CLO**
BNE	**BLTZAL**	**J**
JR	**MULTU**	**MFHI**
MFLO		

B. CPU Simulation

Once you are done with the all the instructions, you can use provided memory to sequentially feed the CPU with a sequence of instructions, that, for the sake of this course, would serve as a simple mini-program. The memory should be connected to the CPU as in the figure below. You can verify the functionality of your CPU by visually inspecting the output signals (MemWrite , MemoryAddress and MemoryDataOut) on the simulation waveforms, along with the internal state of the Memory block itself.



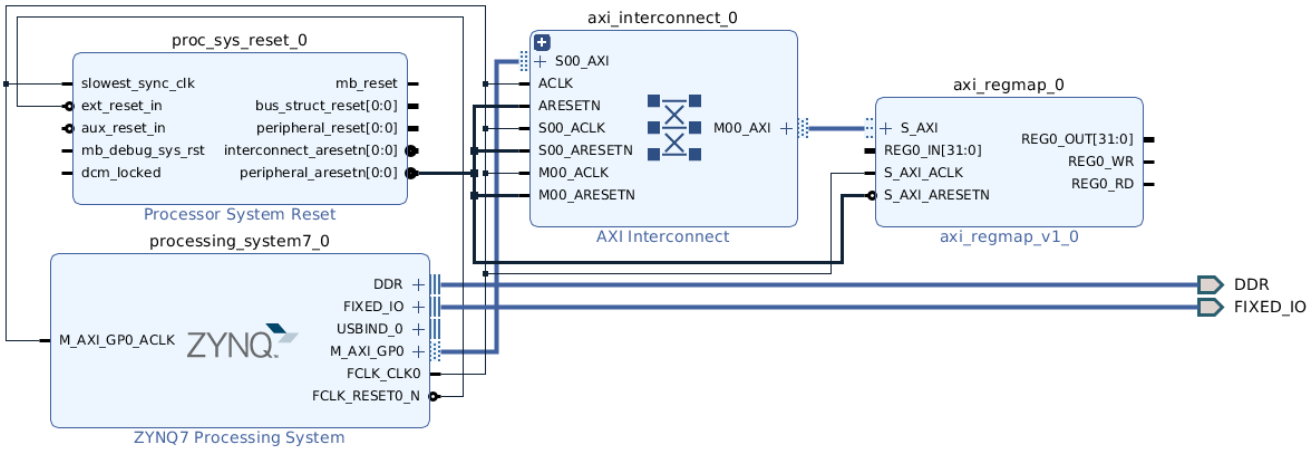
It is a good idea to use a set of test instructions such that you first initialize the register file operands and after the execution has completed, see the final result on the output bus. For example, in order to test the addi, sub and sw instructions, you could use the following program:

```
addi    $7, $0, 17     #   GPR[7]← 17
addi    $11, $0, -3    #   GPR[11]← -3
and     $11, $7, $11   #   GPR[11]← 17
sw      $11, 15($7)    #   memory[32]← 17
```

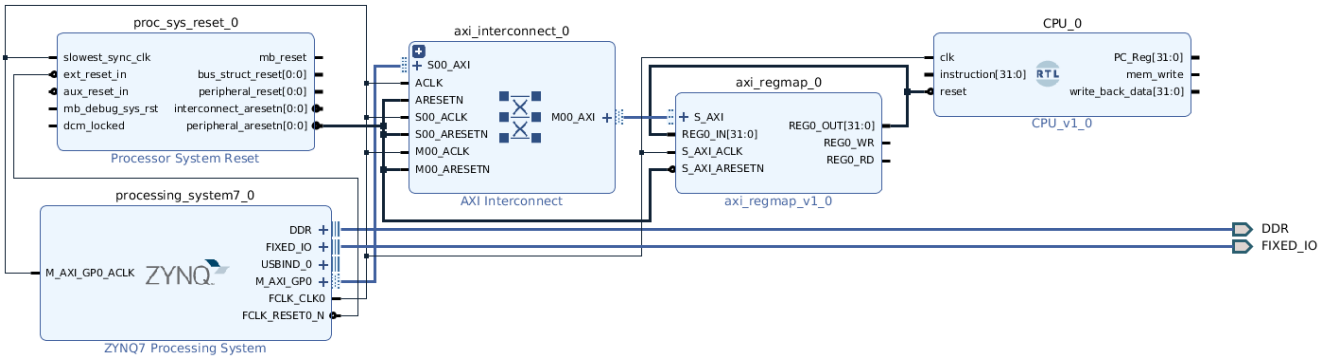
A sample tcl script initializing the memory with the instructions corresponding to the above program is provided with this Lab. If everything is working correctly, then the four instructions would have been fetched at addresses 0x00000000, 0x00000004, 0x00000008 and 0x0000000C. You can verify that the **addi**, **and** and **sw** instructions were executed properly by observing the output bus during the execution of the fourth instruction. If the value of 0x00000011 was stored at address 0x00000020, then all of the instructions executed correctly.

Creating the Block Design for C/C++ Testbench:

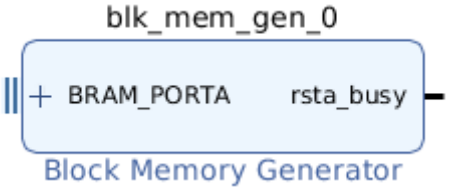
To be able to test your CPU using the ZYNQ PS, you will need to add a few additional components, other than your CPU, for the instruction/data memory. To start, have a block design with the ZYNQ PS (along with its requirements) and a RegMap IP, similar to previous labs:



Then, add your CPU and connect the **Clock** port to **FCLK_CLK0** and **Reset** port to RegMap as shown below (be sure to adjust the width of the RegMap registers):



Now, we will add the component that will serve as our instruction/data memory for the CPU. We will use an IP from the Xilinx library named "**Block Memory Generator**". Using the Block Memory Generator, you can easily create a RAM component that uses dedicated block RAM (BRAM) resources in the FPGA instead of fabric. It should look like this (search for Block Memory Generator):



Now, let's customize it. Open up its configuration window and make the following changes:
Under the *Basic* tab:

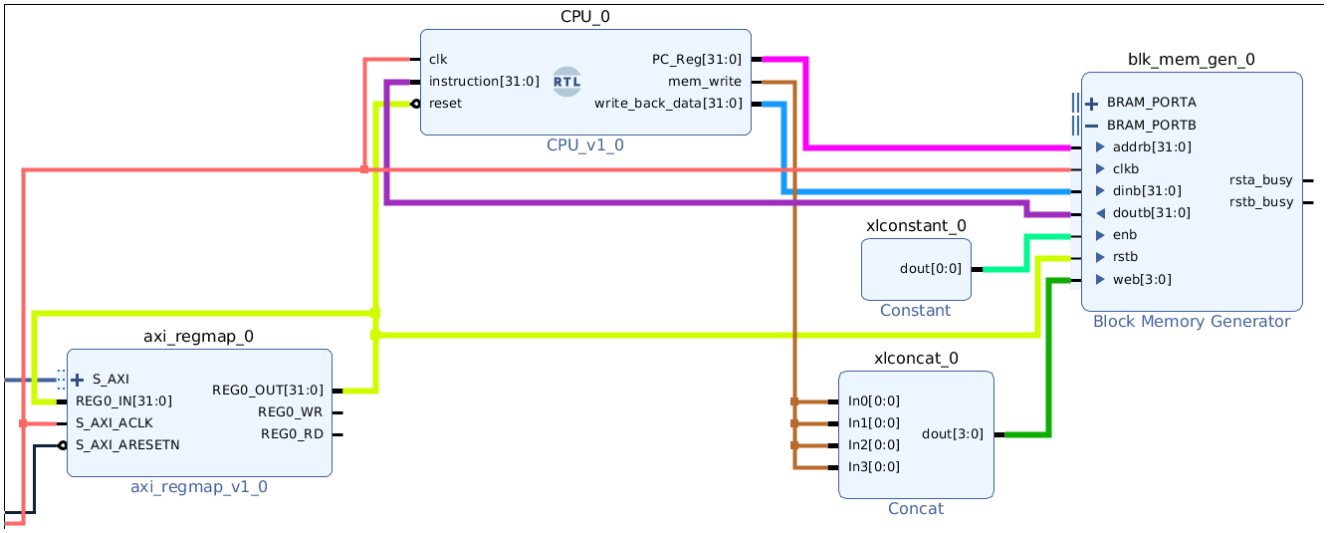
- Make sure that "Mode" is set to "BRAM Controller"
- Change "Memory Type" to "True Dual Port RAM" (to have two read/write ports)
- Check the box that says "Common Clock"

Looking under the *Summary* tab, ensure the following settings match:

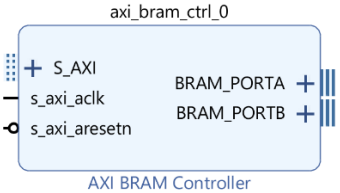
- Memory Type: **True Dual Port RAM**
- Total Port A Read Latency: **1 Clock Cycle(s)**
- Total Port B Read Latency (From Rising Edge of Read Clock): **1 Clock Cycle(s)**
- Address Width A: **32**
- Address Width B: **32**

If it all looks correct, click OK to close the customization window. Your component should have two BRAM interface ports (*BRAM_PORTA* and *BRAM_PORTB*) as well as two additional ports (*rsta_busy* and *rstb_busy*). Now, let's connect one of the BRAM ports to your CPU. Expand BRAM_PORTB of the block memory (by pressing the + icon) and make the following wire connections.

You will also need to add a Constant IP to keep the *enb* port as a '1', and a Concat IP to replicate the value of the *mem_write* signal of your CPU and match the width of the *web* (write enable B) port of the BRAM. Your design should now look like this (highlighting the nets is only done here to help you see the port connections):



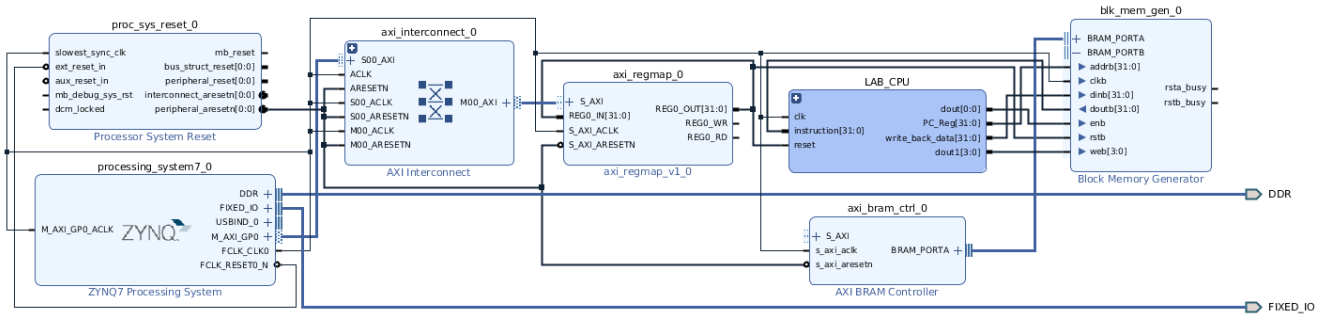
Your CPU will now be able to use this block RAM. But how to connect it to the ZYNQ PS? For this, you will need to add another component from the Xilinx library called "**AXI BRAM Controller**". It looks like this (search for AXI BRAM Controller):



This component will convert between AXI (or AXI-Lite) and BRAM interfaces. Open its configuration window and change the following:

- Change AXI Protocol to "AXI4LITE"
- Change Number of BRAM interfaces to "1"

Next, connect the BRAM_PORTA port of the BRAM controller to the other remaining port of the block memory, and wire up the clock and reset signals accordingly. Your design should look like this:



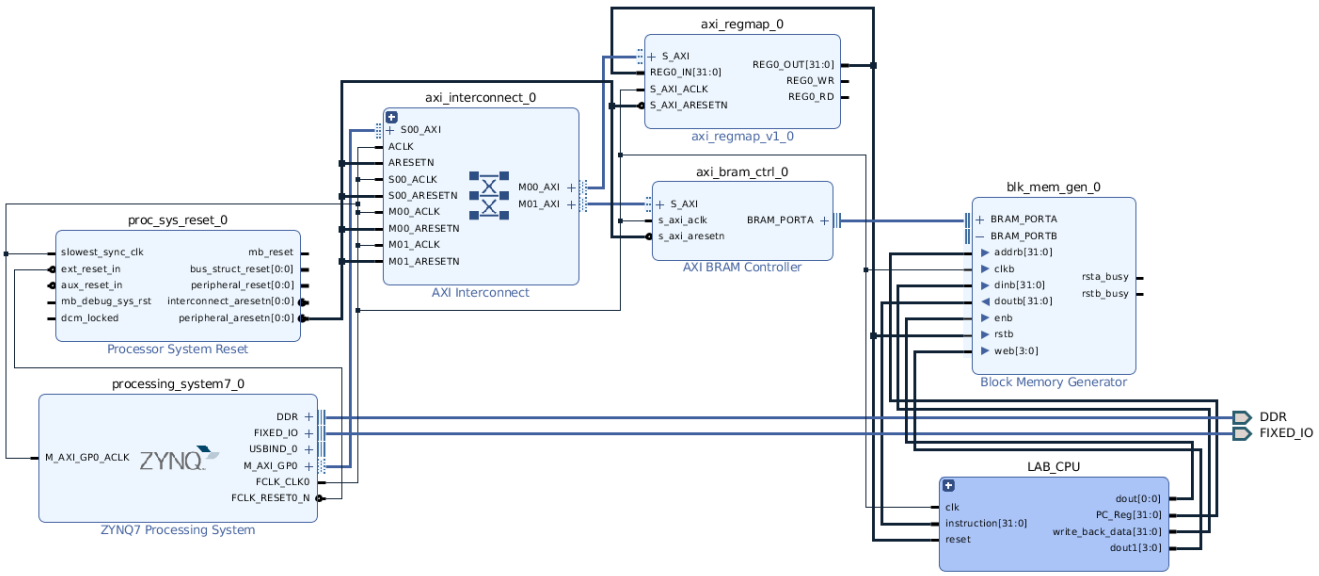
As you can see, the blocks related to your CPU (Constant and Concat IPs) have been put in a hierarchy (LAB_CPU) just to reduce clutter. To do so, you can select all the component that you want to add in a hierarchy, then right click on any of them and choose “Create Hierarchy”.

In order to connect the *S_AXI* port of the BRAM controller the AXI Interconnect, we need to add another Master Interface first. Open the configuration window for the AXI Interconnect and change the “Number of Master Interfaces” to 2). This will add another inter face, *M01_AXI*, which can now be connect to *S_AXI* of the BRAM Controller.

If you now look on the Address Editor, you should see that you can map a memory address to the AXI BRAM Controller. Now, you will be able to read/write to the instruction/data memory component from the ZYNQ PS using a mapped memory address in the same way as you have for the RegMap IP. Make sure the value are the same as in the picture below.

Cell	Slave Interface	Base Name	Offset Ad...	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_regmap_0	S_AXI	reg0	0x4000_0000	8K	0x4000_1FFF
axi_bram_ctrl_0	S_AXI	Mem0	0x4100_0000	8K	0x4100_1FFF

Your block design is now complete! Here’s a final reference showing all the connections:



You can now refer to the attached C code template to start implementing your testbenches.

What to do:

1. Design the 32-bit MIPS CPU, per the description above. Your final top-level CPU should have the following IO ports. Make sure that the ports' name in your entity matches the letter case and naming in the Table.

Port	Direction	Description
Reset	In	Asynchronous active high reset for CPU (single bit)
Clock	In	Global clock signal (single bit)
MemoryDataIn(31:0)	In	32-bit input bus carrying the data from memory.
MemoryAdress(31:0)	Out	32-bit output bus carrying the memory address.
MemoryDataOut(31:0)	Out	32-bit output bus carrying the data to be written to memory.
MemWrite	Out	Write enable signal for the memory, active high (single bit)

2. Write multiple Tcl scripts implementing different programs. Your programs should at least try each instruction once. The scripts should follow the example script given to instantiate the memory component.
   your final CPU testbench top-level entity (the one that includes the memory) should be named *"cpu_tb"* and it should only have two components, the CPU and the provided memory block. It should also have two ports only:

Port	Direction	Description
reset	In	Asynchronous active high reset (single bit)
clock	In	Global clock signal (single bit)

   When you are instantiating the components, make sure to label the CPU component *U_0* and the memory component *U_1*. This will make sure the tcl scripts we are going to use for testing your design can work right out of the box, without the need to change anything on the fly.

3. Design an assembly program that would test your multiplication commands and write a Tcl script for it. Remember that we can only inspect memory registers, so make sure your program and script are complete and thorough in order to be able to check if the product is correct.
   You need to write a short report (1 page or less) about this test, including the following:
   a. Your program in assembly language
   b. Your Tcl script
   c. Waveform showing the simulation results for the Tcl
   d. A paragraph discussing the results you got. This paragraph should describe whether you got the multiplication commands working correctly or not and how did you find out that. If they weren't working correctly, or you got a weird result, write what you think went wrong, why do you think that, and how you might solve it if you have time.

4. Write multiple C/C++ testbenches implementing different programs. Your programs should at least try each instruction once. Follow the provided C template.

Deliverables:

1. You will be delivering all the VHDL and C codes that you have written, including the design files, testbenches, and the C codes. The submission will be on Canvas Assignment for Lab 4. You can just Zip the top Vivado project folder that has all the files that was mentioned above. (please make sure that it has all the files required).
2. Make sure to submit the reflection you wrote for the multiplication program as a separate document file (pdf or docx) in your submission. Don't include it in your compressed project folder, as this will make it very hard to grade them. Failing to do so may result in points loss.
3. Check-offs will be done on the due date for this assignment. The rubric for the check-offs will be provided later.

Bonus:

Design an assembly program to divide two integers, using only the instructions you implemented. A maximum of 2% bonus towards your final grade will be given, depending on how you write the code (handles positive only or mix of operands). You can assume you have enough memory and expand the size of the given memory component as needed.