

The UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Comp 541 Digital Logic and Computer Design

Prof. Montek Singh

Fall 2021

Lab #8: A Full Single-Cycle MIPS Processor

Issued Fri 10/15/21; Due Mon 10/25/21 (11:55pm)

You will learn the following in this lab:

- Integrating ALU, registers, etc., to form a full datapath
- Designing the control unit for a processor
- Integrating the control unit with the datapath
- Integrating memory units with a processor
- Encoding instructions
- More practice with test fixtures for testing a processor

Part 0: Carefully review the lecture on single-cycle MIPS processor design.

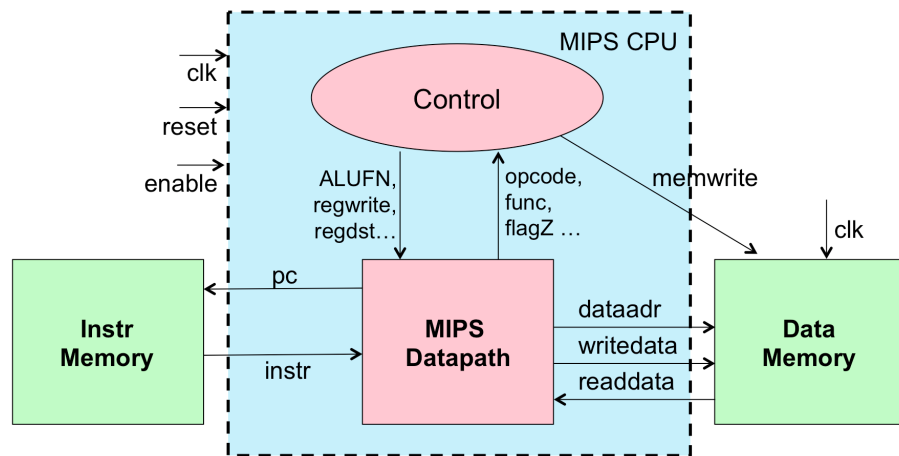
Study the lecture slides carefully, and review Chapter 7.1-7.3 of textbook. Also review the Patterson Hennessy front inside green flap for summary of the MIPS instruction set; a PDF version of this (“green card”) is posted on the class website. Wherever there are differences between the Patterson Hennessy textbook and the Harris and Harris textbook, we will adhere to the former. We will do so because the MARS simulator, which you will use to assemble your MIPS assembly code follows the former.

Please study Slides 36-41 especially carefully to note three specific design choices for our lab version of MIPS:

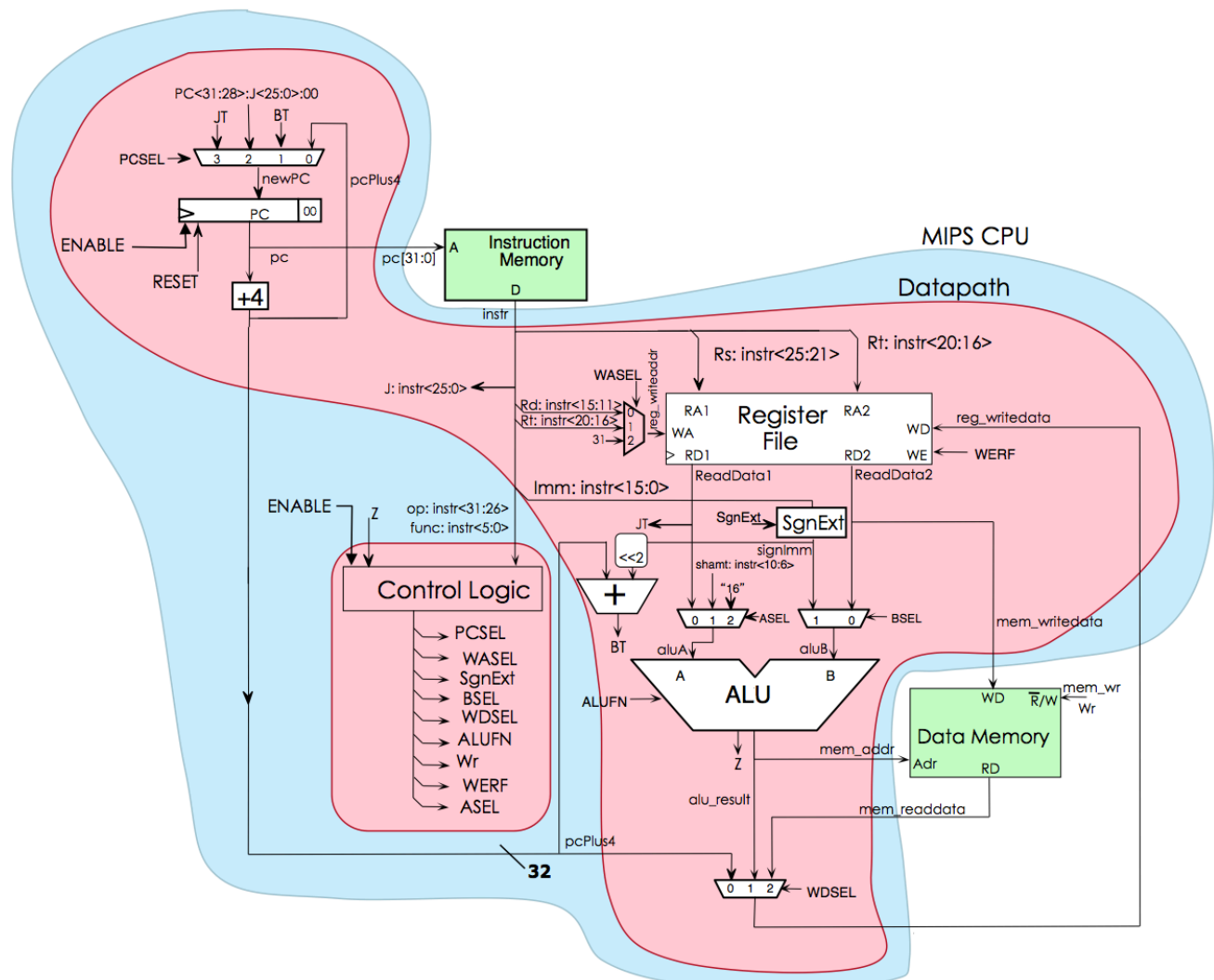
- *Exceptions*: Our lab version does not support exceptions.
- *Reset*: It does support handling resets though. In particular, if the *Reset* input is asserted, the program counter is re-initialized to the address **0x0040_0000**. This address was chosen for compatibility with the MARS assembler. **Accordingly, the PC register in the datapath should be initialized to this address, and re-initialized whenever reset is asserted.**
- *Enable*: To help with debugging, we will also incorporate an *Enable* input. When enabled, the processor runs normally. When *Enable* = 0, however, the processor freezes. This is accomplished by disabling writes to: PC, register file and data memory. This modification will allow us later to single-step through execution as a debugging aid.

Part 1: The Control Unit

In preparation for designing a full MIPS CPU, we will develop the Control Unit in this exercise. Below are two diagrams of our single-cycle MIPS CPU (from Comp411), first a top-level overview, then a detailed one.



Here is the picture showing the details:



Let us first develop the control unit. It should handle **ALL of the following 31 instructions**:

- `lw` and `sw`
- `addi`, `addiu`, `slti`, `sltiu`, `ori`, `lui`, `andi`, `xori`
 - **NOTE 1:** Contrary to what you may have learned earlier (e.g., in Comp411 from other instructors), `addiu` actually does not perform unsigned addition. In fact, it *sign-extends the immediate*. This instruction (along with `addu`) was actually misnamed! The only difference between `addiu` and `addi` is that `addiu` does not cause an exception on overflow, whereas `addi` does. Since we are not implementing exceptions, `addiu` and `addi` are identical for our purposes.
 - **NOTE 2:** Also, contrary to what you may have learned earlier, `sltiu` actually *sign-extends the immediate, but performs unsigned comparison*, i.e. ALUFN is “LTU”.
 - Also note that `ori` should zero-extend the immediate because it is a logical operation! Finally, sign-extension for `lui` is a *don't-care* because the 16-bit immediate is placed in the upper half of the register without any need for padding.
- R-type: `add`, `addu`, `sub`, `and`, `or`, `xor`, `nor`, `slt`, `sltu`, `sll`, `sllv`, `srl`, `srlv`, `sra`, `srav`
 - NOTE: The `addu` instruction, for our purposes, is *identical* to the `add` instruction. They only differ in how they handle overflow, which we ignore in our design. The reason for supporting the `addu` instruction is that the MARS assembler often automatically inserts `addu` instructions in our code, especially for the range of memory addresses we will be using.
- `beq`, `bne`, `j`, `jal`, `jr` and `jalr`

First study the Powerpoint slides on Single-Cycle MIPS processor (including Slides 36-41). Next, fill out the table below with the values of all the control signals for the MIPS instructions listed here. If a control signal's value does not matter for a particular instruction, please make it a don't-care (i.e., 1'bX, 2'bX, etc., depending on the number of bits).

Using the values in this table, complete the Verilog description of the control unit in the file `controller.sv` available on the website.

Use the Verilog test fixture provided on the website to simulate and test your design. The text fixture is self-checking, so any errors will be flagged automatically. Please use exactly the same names for the top-level inputs and outputs as used in the tester.

MIPS instruction decoding table

	<i>Instr</i>	werf	wdsel	wasel	asel	bsel	sgnext	wr	alufn	pcsel
I-Type	LW	1	10	01	00	1	1	0	0XX01	00
	SW	0	10	01	00	1	1	1	0xx01	00
	ADDI	1	01	01	00	1	1	0	0xx01	00
	ADDIU	1	01	01	00	1	1	0	0xx01	00
	SLTI	1	01	01	00	1	1	0	1x011	00
	SLTIU	1	01	01	00	1	1	0	1x111	00
	ORI	1	01	01	00	1	0	0	x0100	00
	LUI	1	01	01	10	1	x	0	x0010	00
	ANDI	1	01	01	00	1	0	0	x0000	00
	XORI	1	01	01	00	1	0	0	x1000	00
J-Type	BEQ	0	xx	xx	00	0	1	0	1xx01	Z=1 01 Z=0 00
	BNE	0	xx	xx	00	0	1	0	1xx01	Z=1 00 Z=0 01
	J	0	xx	xx	xx	x	0	0	xxxxx	10
R-Type	JAL	1	00	10	xx	0	0	0	^x Xxxxxx	10
	ADD	1	01	00	00	0	0	0	0xx01	00
	ADDU	1	01	00	00	0	0	0	0xx01	00
	SUB	1	01	00	00	0	0	0	1xx01	00
	AND	1	01	00	00	0	0	0	x0000	00
	OR	1	01	00	00	0	0	0	x0100	00
	XOR	1	01	00	00	0	0	0	x1000	00
	NOR	1	01	00	00	0	0	0	x1100	00
	SLT	1	01	00	00	0	0	0	1x011	00
	SLTU	1	01	00	00	0	0	0	1x111	00
	SLL	1	01	00	01	0	0	0	x0010	00
	SLLV	1	01	00	00	0	0	0	x0010	00
	SRL	1	01	00	01	0	0	0	x1010	00
	SRLV	1	01	00	00	0	x	0	x1010	00
	SRA	1	01	00	01	1	0	0	x1110	00
	SRAV	1	01	00	00	0	x	0	x1110	00
	JR	0	xx	xx	00	1	0	0	xxxxxx	11
	JALR	1	00	00	00	1	0	0	xxxxxx	11

These values
will depend on
the Z flag.

Part 2: Design a full single-cycle MIPS.

Put all the pieces together to create a full single-cycle MIPS CPU as discussed in class. Verilog templates for some of the higher-level modules are on the course website. In particular, do the following:

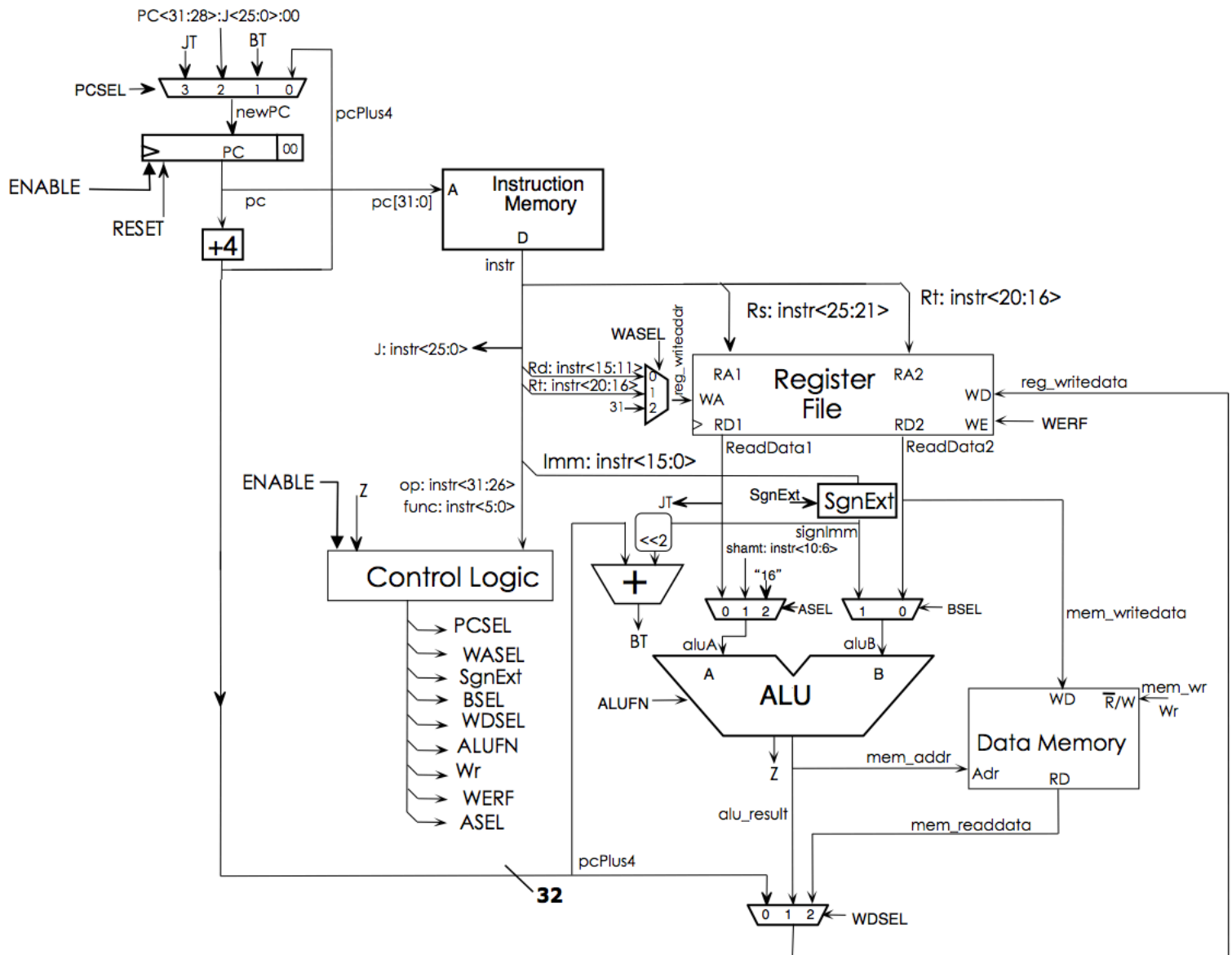
- **Design the full single-cycle MIPS CPU, along with instruction and data memories, as shown on Slides #39-41** of the lecture slides. Start with the file provided, `top.sv`, which already contains the top-level Verilog module. Understand how it conforms to the top-level block diagram on Page 2, and carefully make note of all of its parameters.

The size of the instruction and data memories **must be parameterized**, so that it is easy to change those sizes depending on the actual needs of the program (e.g., your final demo) we want to run on our processor. (The tester provided appropriately sets the sizes to be 128 memory locations for instruction memory and 64 for data memory.) *Note:* The addresses generated by the CPU to access these memories should be the full 32 bits long, even though fewer address bits will actually be used inside these memories. Use the ROM and RAM modules from the previous lab (`rom.sv` / `ram.sv`), and note the following:

- The instruction memory will be a ROM, while the data memory will be a RAM. We will instantiate the ROM and RAM modules (without changing their Verilog descriptions), and provide the appropriate parameters, which are in turn passed down to the top module from the tester. (See `top.sv`.)
 - We will send the full 32-bit PC out of the CPU, but strip the last two bits before connecting to the outside of the instruction memory. That is, **only a 30-bit word address is connected to the outside of the instruction memory**. This converts byte addresses into word addresses. (See `top.sv`.) However, the actual number of address bits inside the instruction memory is actually determined by the number of memory locations (automatically determined by our memory template). Only those many bits from the PC will actually be connected to the address inside the instruction memory; the remaining upper PC bits will automatically be discarded. As a result, even though the default address for start of program code is `0x0040_0000`, because only the lowest bits are actually connected through, this will correspond to address 0 in the actual ROM module that implements instruction memory.
 - Similarly, we will send the full 32-bit data memory address out of the CPU, but strip the last two bits before connecting to the data memory, so **only a 30-bit word address** is actually connected to the outside of the data memory. (See `top.sv`.) Once again, the actual number of bits in the address inside the data memory will automatically be determined by the number of locations.
 - Both of these memories should return a full 32-bit word (i.e., `Dbits = 32`). Their initial values are read from the `initfile`, which is the file name for memory initialization file.
- **Initialize the instruction and data memories**, using the method explained in Lab 7. The file that contains the initial values for the instruction memory will contain a 32-bit assembly coded instruction per line (in hex). The file that contains the initial values for the data memory will contain some 32-bit data values, again one per line. **Note: You can create these files either within Vivado, or outside of Vivado and then add these to the project. Make sure to set their type to “Memory File.”**
 - **The processor module, containing controller and datapath modules**, is provided (`mips.sv`). Understand how it conforms to the block diagram below. *Note:* There are differences between the MIPS design in the Harris and Harris textbook and what we are designing in the lab. Our lab version has a much more sophisticated ALU. Therefore, do not blindly follow the information in the book; instead, follow the lecture slides and the lab writeups. The datapath should be 32-bit wide (i.e., registers, ALU, data memory and instruction memory, all use 32-bit words).

- **Complete all the little pieces**, so the design looks like that in the lecture slides, also reproduced below. Small amounts of logic can be “inlined” (instead of written as a separate module), e.g., muxes, sign extension, adders, shift-by-2, etc.

The figures on Page 2 show the top-level hierarchical decomposition. Your design is required to follow that hierarchy.



Other helpful tips and testing aids:

- **Remember to use the directive ``default_nettype none`** to make it easier to catch missing declarations or name mismatches due to typos, etc.
- **It is highly recommended you connect inputs/outputs by name**, especially for modules that contain more than a couple of inputs/outputs. Otherwise, it is easy to misalign the input/output ports, causing debugging headaches. You can follow the style in `mips.sv` for the controller and datapath connections.
- **Use the test fixture to test your CPU via simulation.** A self-checking tester is provided on the class website. It was initially written in MIPS assembly, then compiled using MARS and converted to hex machine code, which should be used to initialize your instruction memory. Store the machine code into the file that is used to initialize the instruction memory. Be sure to initialize the program counter (PC) inside your MIPS design to `0x0040_0000`, so that it starts executing from the beginning of the instruction memory. Similarly, to initialize your data memory, put the initial values in the corresponding file.
 - The tester is called **full** (`tester_full.sv`): The assembly program (`full.asm`) executes each of the 31 instructions we have implemented, including procedure calls/returns, and recursion using a stack. You don't have to run this program in MARS, but you may. The instruction memory should be sized to have at least as many locations as the lines of machine code, and typically a power of 2. Accordingly, the tester sets the instruction memory size to 128. Also, in the tester, be sure to specify the correct names of the files that have initialization values ("`full_imem.mem`" and "`full_dmem.mem`", respectively).
- **For now, you will not be implementing your design on the board.** You will do so next lab.
- **Start thinking about what you would like to build for your final project!** Every project must use a VGA display as output. The Nexys boards have audio output and accelerometer input built-in. Keyboard inputs will be available to every project. Start thinking!

Okay, good luck!

What to submit:

- **Part 1:** The file for the (`controller.sv`), a picture of the instruction decoding table (phone picture is fine), and a screenshot of the simulation waveform window using the self-checking tester.
- **Part 2:** ALL of the Verilog files, but skip the ALU and its submodules. Also, a screenshot of the simulation waveforms for the "full" self-checking tester provided.

How to submit:

- Submit via Sakai ("Lab 8" under Assignments).
 - Include the Verilog sources and pictures as specified above
-