CENG 351: Computer Architecture I

Lab 6: Adding new instructions

Labs should be completed in pairs.

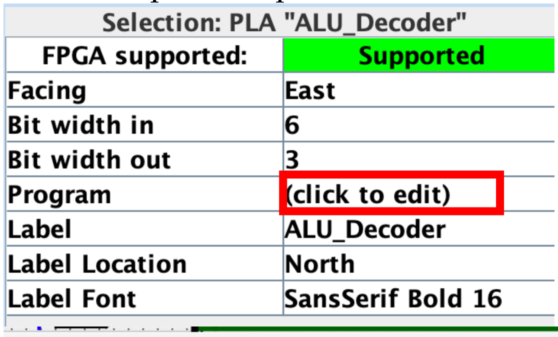
Designers will continually update their designs as they add functionality for all instructions present in the instruction set. This process involves adjusting the control unit and modifying the datapath if necessary. Modifying the datapath may require changes at several places in the hierarchy.

In this lab, you will modify your datapath to add functionality for j, bne, and ori instructions. We have covered modifying the datapath for j and bne, but ori is a new instruction to add. This I-type instruction performs an OR operation on a register value and an immediate, and stores the results back to a register. You should build off of the processor design in previous labs. Once completed, we will use a new, more complicated assembly program to test your processor design.

Instructions to add

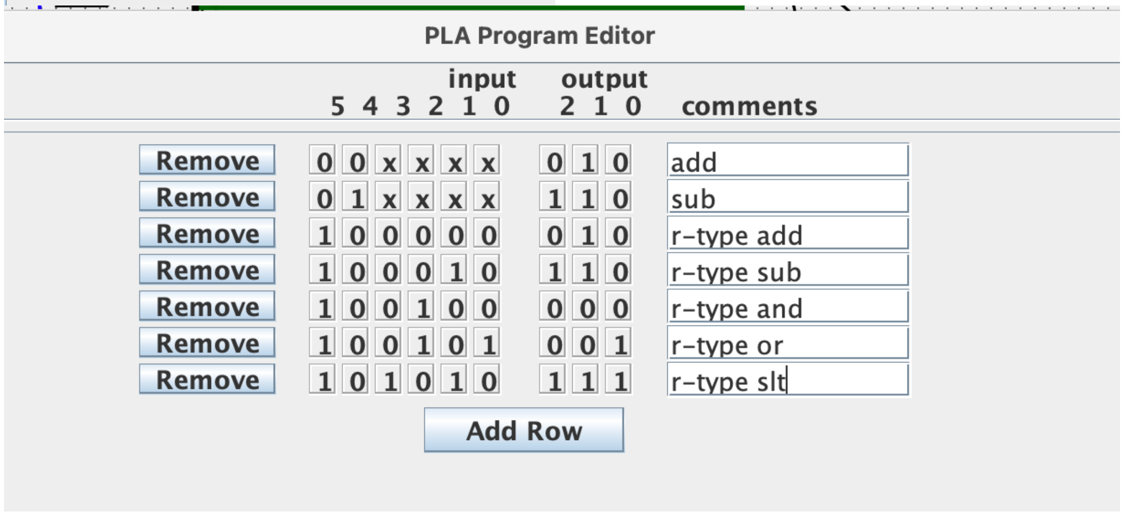
1. The j instruction performs a jump. We covered adding this instruction to the single-cycle processor in class. The opcode is 000010 and the rest of the instruction is a pseudo-direct address. Note that the shift left by 2 for this instruction adds bits to the bus, so you may need to reconsider how you use the shift left module in Logisim.
2. The bne instruction performs a branch if the operands are not equal. We covered adding this instruction to the single-cycle processor as an example in class. The opcode is 000101 and the format is: bne rs, rt, label.
3. The ori instruction performs an OR operation on a register value and an immediate value, and stores the result back to a register. The opcode is 001101 and the format is: ori rt, rs, imm, where rs is the source register and rt is the destination register. There are two important notes about this instruction. First, by definition, the instruction uses a *zero-extended immediate* instead of a sign-extended immediate. Second, our ALU has the functionality to perform an OR operation, but the ALU controller can only issue an OR operation based on the funct field for an R-type instruction.
4. In order to implement the ori instruction, I recommend you replace the ALU decoder logic with a programmable logic array (PLA) module available under the “Gates” section of the design tree. Using this module, you can enter in the truth table of the behavior you want. Making this change will simplify changes you need to make for the ori instruction. You will need to click the “click to edit” field under program in the properties tab to adjust the truth table (in image

below). Below is an image of the original ALU decoder where input bits 5 and 4 are ALUOp, and input bits 3-0 are bits 3-0 of the funct field.



Selection: PLA "ALU_Decoder"

FPGA supported:	Supported
Facing	East
Bit width in	6
Bit width out	3
Program	(click to edit)
Label	ALU_Decoder
Label Location	North
Label Font	SansSerif Bold 16



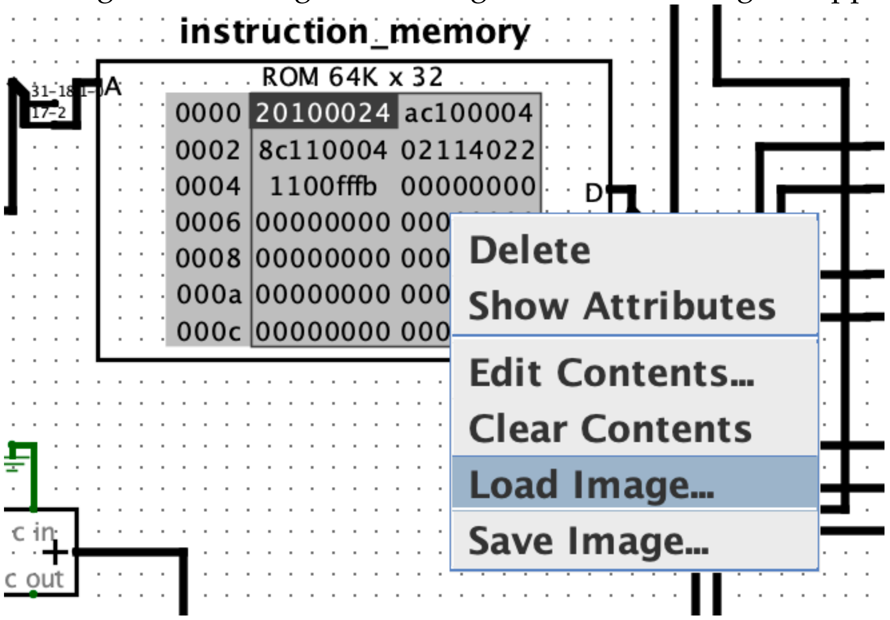
PLA Program Editor

	input 5	4	3	2	1	0	output 2	1	0	comments
Remove	0	0	x	x	x	x	0	1	0	add
Remove	0	1	x	x	x	x	1	1	0	sub
Remove	1	0	0	0	0	0	0	1	0	r–type add
Remove	1	0	0	0	1	0	1	1	0	r–type sub
Remove	1	0	0	1	0	0	0	0	0	r–type and
Remove	1	0	0	1	0	1	0	0	1	r–type or
Remove	1	0	1	0	1	0	1	1	1	r–type slt

5. I recommend you plan out your changes for each instruction before you make any modifications. Add the instructions to the processor in a systematic manner to make sure you don't forget on anything. For each instruction, focus on adding to the control unit, then update the datapath (if necessary), then move on to the next instruction. For all of the instructions, you will need to make an update to the control unit, but you won't need to update the datapath for every instruction. *A copy of the datapath and control unit truth table is provided at the end of this lab document.*
6. *If you add signals to the main decoder, you will also need to add outputs for those signals in the "control unit" module.*

Simulation

1. You will test the processor design using a new assembly program. You will need to load this program into the instruction memory. This can be done by right-clicking and choosing "Load image…" then selecting the appropriate hex file.



```
0000 20100024 ac100004
0002 8c110004 02114022
0004 1100fffb 00000000
0006 00000000 000
0008 00000000 000
000a 00000000 000
000c 00000000 000
```

2. An assembly program which tests all the instructions is below. This program should only produce the correct result if all of the instructions are functioning properly. The end result is a specific value written to a specific memory location. If there are bugs in your hardware, you are unlikely to get the correct result, an example of *ad hoc* testing.

```
main:     ori  $t0, $0,   0x8000
          addi $t1, $0,   -32768
          ori  $t2, $t0, 0x8001
          beq  $t0, $t1, there
          slt  $t3, $t1, $t0
          bne  $t3, $0,   here
          j    there
here:     sub  $t2, $t2, $t0
          ori  $t0, $t0, 0xFF
          j    where
there:    sub  $t3, $t0, $t2
          add  $t0, $t3, $t3
where:    add  $t3, $t3, $t2
          sub  $t0, $t2, $t0
          sw   $t0, 82($t3)
```

3. The machine language for this assembly program is provided in a file called "**memfile2.hex**"

4. Once you have loaded the simulation, you will step through the assembly one clock cycle at a time. Before you step forward, make sure that the control unit is setting appropriate control signals for your instruction.
5. After each instruction is verified, click on the "Tick clock one full cycle" button. You should see the program counter increment and the instruction memory move on to the next instruction.
6. Repeat this process for all instructions until the program is complete. **Take a screenshot of the specific value stored in memory and include it in your lab report**. You should be able to figure out what this value should be and where it should be written by analyzing the assembly code. You can view the contents of memory by right-clicking and selecting "edit contents".

Compile a short report that contains screenshots of the completed datapath and control unit (including underlying implementation) as well as the correct end result of the test program. Feel free to break up the datapath over multiple screenshots if that is helpful. Describe any problems you had during the lab and what you did to troubleshoot. Describe each partner's contribution to the lab. Include answers to the lab questions below

Lab questions:

1. Why does the ori instruction use a zero-extended immediate instead of a sign-extended immediate?
2. What is the correct value stored to memory at the end of the program? What memory address is this value written to?
3. What is the advantage of using our ad-hoc testing approach? What is another option for testing, and what are some pros or cons of this different approach?

Submission

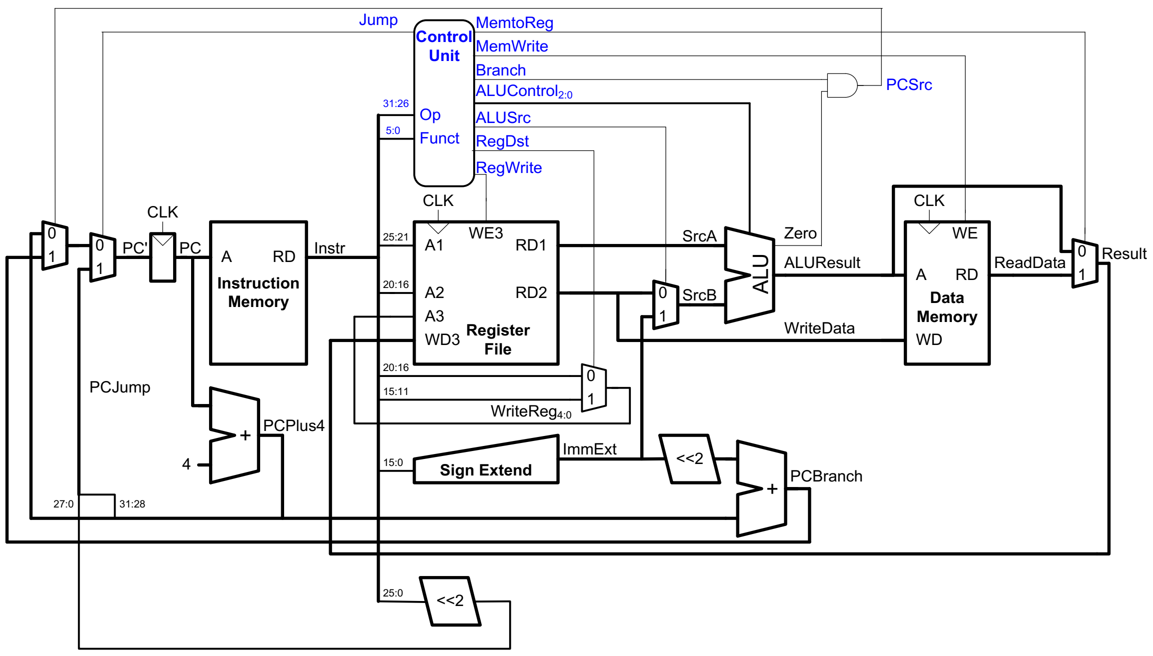
Please submit a digital copy of your lab report. If taking screenshots, please save them into the lab report document instead of as separate files. The lab should be submitted through canvas and only one report needs to be submitted per group. In the lab report, please indicate all lab partners and their contributions.

Due Date

This assignment is due on the date shown on Canvas by 11:59pm. Submit via Canvas in PDF format.

Grading

Grading will be based on correctness and completeness of the solution (i.e. show all your work).



Single-cycle MIPS processor

Extended functionality. Main Decoder:

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump			
R-type	000000	1	1	0	0	0	0	10	0			
`lw`	100011	1	0	1	0	0	1	00	0			
`sw`	101011	0	X	1	0	1	X	00	0			
`beq`	000100	0	X	0	1	0	X	01	0			
`addi`	001000	1	0	1	0	0	0	00	0			
`j`	000010	0	X	X	X	0	X	XX	1			
`ori`	001101											
`bne`	000101											

Extended functionality. ALU Decoder:

$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at `funct` field
11	