ECE 1110 Introduction to Computer Architecture
Electrical and Computer Engineering

Spring 2022

Assigned: 2/1/2022 Due: 3/17/2022

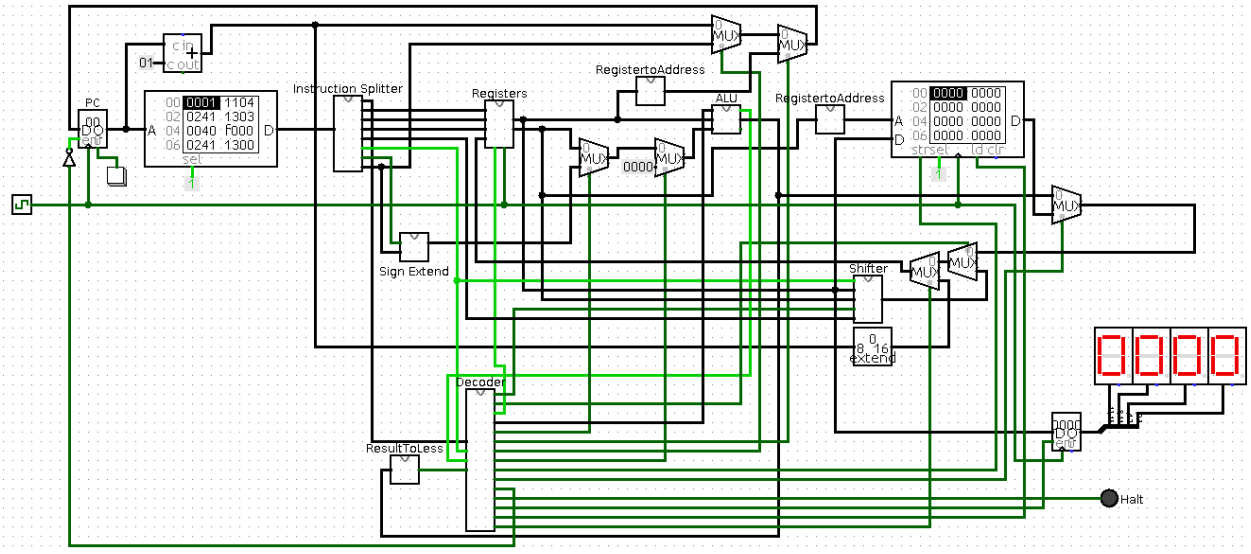
1. Introduction

In this project, you will implement a five-stage pipeline in Logisim for an ISA that resembles RISC-V. You are supposed to implement the corresponding function units, control units, registers etc. in processor. You also need to deal with the data hazards and control hazards in pipeline design.

You will form a team of 2 people, and complete the project together. Please form your team quickly, and plan how to divide the work between you.

2. Project Requirements

Your job is to implement a five-stage pipelined processor, which includes IF, ID, EX, MEM, and WB. Each stage takes 1 cycle to finish, which means the longest instruction needs 5 cycles to process. Your implementation will need several components: 1) a program counter and fetch adder; 2) an instruction memory; 3) a register file; 4) an instruction decoder; 5) a sign extender; 6) an ALU; 7) a data memory; 8) 4 pipeline latches; 9) a forwarding unit; 10) a hazard unit 11) an LED hexadecimal display; and, 12) an LED to indicate the processor has halted. You'll also need muxes as appropriate. For the most part, these components are quite similar to the single-cycle datapath shown below, which will be uploaded into Canvas. You will start from this datapath, and revise it into a pipeline datapath. You will find it helpful to consult the book and lecture slides, particularly the diagram of the learnt datapath with control signals, the decoder and other elements. For the project, you may use any component (e.g., an adder) from Logisim's built-in libraries. This makes the project much simpler. All other components must be implemented from scratch.



3. Assembly Language

3.1 Instruction Set

Opcode	Subop	Format	Instruction	Definition
0000	0	R	add $rs, $rt	$rs ← $rs + $rt
0001	X	I	addi $rs, imm	$rs ← $rs + sxt (imm)
0001	X	I	addui $rs, imm	$rs ← $rs + zxt (imm)
0010	0	R	sub $rs, $rt	$rs ← $rs - $rt
0110	0	R	lw $rs,$rt	$rs ← MEM[$rt]
0110	1	R	sw $rs,$rt	MEM[$rt] ← $rs
0111	0	R	halt	stop fetching and set halt LED to red
1000	X	I	bz $rs,imm	PC ← ($rs = 0 ? imm : PC + 1)
1010	X	I	bp $rs,imm	PC ← ($rs > 0 ? imm : PC + 1)
1111	X	I	put $rs	output $rs to Hex LED display

3.2 Execution of instruction

In this table, "X" indicates the Subop field (see below) is not applicable for I-format instructions. There are some differences between this project and the regular RISC-V instructions. First, in this project we use a "two-operand" instruction set. An instruction has at most two operands, including source and destination operands. In this instruction format, one of the source operands (registers) is also the destination. However, most instructions still behave like their RISC-V counterparts.

3.3 Instruction Format

There are two instruction formats in this project: R and I. R is used for instructions that have only registers and I is used for instructions with an immediate. The formats are:

R Format Instruction					
15-12	11-9	8-6	5	4-1	0
Opcode	*Rs*	*Rt*	*unused*	*Shamt*	*Subop*

I Format Instruction			
15-12	11-9	8	7-0
Opcode	*Rs*	*Unsigned*	*Imm*

Rs is the first source register and Rt is the second source register. Rs is also the destination register.

Imm is an 8-bit immediate. The immediate is signed in addi and unsigned in addui, bn, bx, bp, bz, jal, and j. In addi, the bit Unsigned controls whether the immediate is sign or zero extended. When Unsigned is 1, then Imm is zero extended for the addui instruction. Otherwise, Imm is sign extended for addi. Imm is zero extended for branches and jump (j). In branches instructions, Imm specifies the target address. Both branches and jumps use absolute addressing for the target address. For example, if a branch is taken and Imm is 0x16, then the target address for the branch is 0x16.

3.4 Implementation Breakdown

3.4.1 Registers

There are 8 registers, labeled $r0 to $r7. In this project, $r0 is not 0. It's a general register and can be used like any other register. The registers are 16 bits wide. An R-format instruction can read 2 source registers and write 1 destination register. Thus, the register file has 2 read ports and 1 write port. You need to implement the register file with the register model in Logisim by yourself.

3.4.2 Instruction Memory

This component is a ROM configured to hold 256 16-bit instructions. You should use the ROM in Logisim's Memory library. In your implementation, the ROM must be visible in the main circuit. The ROM's contents will hold the set of instructions for a program. You can set the contents with the Poke tool or load the contents from a txt file.

3.4.3 Data Memory

This component is a RAM configured to hold 256 16-bit words. You should use the RAM in Logisim's Memory library. In your implementation, the RAM must be visible in the main circuit. Be sure to read Logisim's documentation carefully for this component! To simplify the implementation, configure the RAM's Data Interface as Separate Load and Store Ports. This configuration is similar to what was described in lecture and the book. Hint: You'll need to set the RAM's Sel signal.

3.4.4 Arithmetic Logic Unit (ALU)

The ALU is used in the arithmetic instructions, memory instructions and branch instructions. It can do addition, subtraction and comparison. You are given a module that already does addition and comparison. But you would need to add subtraction to this unit. Review the material (mostly from your prerequisite course) how subtraction is done in 2's complement form. Build the ALU as a subcircuit and be sure to use Logisim's multi-bit Arithmetic library subcircuits. A mux is also needed.

3.4.6 Sign Extender

Logisim has a built-in sign-extender component, which you can use.

3.4.7 Decoder

This component takes the instruction opcode as an input and generates control signals for the data path in each stage as outputs. It's easy to make a decoder subcircuit with Logisim's Combinational Analysis tool (Window→Combinational Analysis). This tool will automatically build a subcircuit from a truth table. To make the decoder, list the opcode bits (from the instruction) as table inputs and the control signals as outputs. For each opcode, specify the output values of the control signals. Once you've filled in the table, click Build to create the circuit. To make your main circuit prettier, the opcode inputs and ALU operation outputs can be combined into multi-bit input and output pins using Splitters. Hint: Logisim has a limit on the number of fields in a truth table. So, you may need two (or more!) decoders.

3.4.8 LED Hexadecimal Display

This project has a four digit hexadecimal (16 bit) display. Instruction "put" outputs a register value to this display. The contents of a put's source register (16-bit value) is output on the display. A value that is "put" must remain until the next put is executed. To implement the LED Hexadecimal Display, you should use Logisim's Hex Digit Display library element (in the Input/Output library). You'll need four Hex Digit

Displays, where each one shows a hex digit in the 16-bit number. A way is also needed to make the display stay fixed until the next put is executed (don't simply wire the hex digits to the register file!). Hint: Use a separate register. The display should only be updated when the put instruction is executed.

3.4.9 Halting Processor Execution

When halt is executed, the processor should stop fetching instructions. But the instructions which are already in the pipeline should be finished one by one. After all the instructions are done, the main circuit must have an LED that turns red when the processor is halted. Hint: A simple way to stop the processor is to AND the instruction register write control with a halt control signal. An alternative way to halt is to set the program counter's enable signal to 0 when a halt instruction is executed.

3.4.10 Program Counter

The program counter is a register that holds an 8-bit instruction address. It specifies the instruction to fetch from the instruction memory. It is updated every clock cycle with PC + 1 or the target address of a taken branch (or jump).

3.5 Pipeline Implementation

3.5.1 Pipeline Latches/Register

To guarantee that portions of the datapath could be shared during instruction execution, we need to place registers between adjacent pipeline stages. Note there is no pipeline register at the end of write back stage. You are supposed to figure out what data in current stage will be useful in all later stages and also build the path to transmit the data between stages.

3.5.2 Forwarding Unit and Hazard Unit

Both data hazard and control hazard should be considered in this project. (There's no structure hazard since each instruction only takes one cycle in each stage).

Data hazard can be classified into EX hazard and MEM hazard, based on different kinds of in struction pairs which cause the hazard. In this project all EX hazards and part of MEM hazard can be solved by adding forwarding unit inside EX stage and also between MEM and EX stage. Hint: A simple way to realize this is using the pipeline registers to build the condition of doing forwarding and build your truth table based on the value of those registers. (Similar to the way in textbook)

For the other part of data hazard that cannot be solved by forwarding and the control hazard, we need to build Hazard Unit which can detect hazard and solve it by inserting bubbles into the pipeline.

4. Project Suggestions

This is essentially a programming project. Plan your design carefully before trying to implement anything. Once you have a good plan, implement the design in small parts. Test each part independently and thoroughly to make sure it behaves as expected. Once you have the different parts, put them together to implement various classes of instructions. You can find testcase programs on the project web site. I recommend that you write and try additional test cases as well.

Subcircuits will make the project easier. To define a subcircuit, use the Project→Add Circuit menu option. Next, draw the subcircuit. Finally, add input and output pins to the subcircuit. Be sure to label each pin (i.e., set the pin's Label attribute). Once you're done with the subcircuit, double click the main circuit in the design folder (on the left side of the window). This will switch to the main circuit. Now, the subcircuit will

appear in the design folder. It can be instantiated (added) in the main circuit by selecting and placing it in the main circuit. The subcircuit should be wired to the main circuit through its input and output pins. Logisim isn't smart about how it handles changes to instantiated subcircuits. If you make changes to a subcircuit that is already instantiated, I recommend that you delete it from the main circuit first. Then, make your changes and re-instantiate it. See Logisim's subcircuit tutorial.

When you build your main circuit, follow a few conventions to make it easier to understand. First, wire your data path in a way that data signals flow from left to right (west to east). Second, wire control inputs so they run bottom to top (south to north). As a consequence of these two guidelines, put data input pins on the left and data output pins on the right in a subcircuit. Control input pins should be on the bottom of the subcircuit. Third, leave plenty of space between different elements in the design. This will make it easier to add more elements and route wires cleanly. Finally, try to route wires in straight lines as much as possible.

5. **Project Demo (3/17/2022)**
   Please prepare to demo your project with our TA on 3/17/2022. We will post a schedule later for you to sign up your suitable demo slot. The demo will be done in-person temporarily. Please use your own laptop. We will test several testcases and check their results. You are encouraged to come up with your own testcases during the development for debugging.

   Schedule TBD

6. **Policy**

You may discuss with others on how to approach the project, but you are not allowed to show your design or discuss specific decisions (e.g., control signal settings) with anyone else other than the TA and instructor. Work that you turn in must represent your own work. Any violation of the course and project policies will subject to penalty.