In this project, you will make a pipelined processor that implements the LC-2200 ISA. There will be five stages in your pipeline: **FETCH**, **INSTRUCTION DECODE/REGISTER READ**, **EXECUTE**, **MEMORY**, and **WRITE BACK**. You will first build the pipeline and then test it with a simple program. Before you move on, read Appendix A: LC 2200-32 Processor Reference Manual to understand the ISA that you will be implementing. We provide you with a logisim file with the some of the structure laid out. Please do not modify the Register File or the ALU.

Building the Pipeline

First, you will have to build the hardware to support all of your instructions. You will have to make each stage such that it can accommodate the actions of all instructions passing through it. Use the book (Chapter 5) to get an idea of what the pipeline looks like and to understand the function of each stage before you start building your circuits.

FETCH

Fetch is arguably the simplest of the five stages to complete. All you need to do in fetch is use the PC to get the instruction out of the instruction memory and increment the PC with the appropriate value. First, notice that the PC can be changed by going to the next instruction, a BEQ instruction, or a JALR instruction. You do not know where the updated PC is supposed to come from. Hence, for now, simply have a multiplexer with at least three input ports, and tie the port that corresponds to selection bit 0 to an adder that adds the current PC value to 1. We will make use of the other ports at a later time. You will need to construct a register buffer between FETCH and INSTRUCTION DECODE/ REGISTER READ. I will leave it up to you to decide what should go in the register buffer.

Hint: Think of what each instructions needs, and pass a union of those.

INSTRUCTION DECODE/REGISTER READ

In this stage, you will have to decode the instruction and read the appropriate registers. Please look at Appendix A: LC 2200-32 Processor Reference Manual in order to understand the instruction formats! To read the registers, you will have to get the register numbers out of the instruction word and use them as inputs to the register file. You will have to make the register file dual ported, which means that you are able to read from two registers at the same time. You will still have only one write port, however. As you will notice, the TA's have been very kind in making the DPRF and providing it to you.

Notice that we need some way to control the hardware in the next three stages. The control signals we have to supply to these stages is determined by the instruction we are trying to execute. The easiest way to get the appropriate control signals is to program a ROM with the all of the control signals, and use the opcode to index into the ROM. As you add more hardware, it will become clear to you what control signals you need. You can come back to this at a later point in time. You can use a spreadsheet to help you make the contents of the ROM. I recommend making a column for each control signal and a row for each operation. Then, you can convert each row into a hex string and copy and paste this into logisim.

Here's a hint: you have to pass the control signals using the pipeline register buffers between the stages. There will be a lot of signals and data that you will need to pass between the ID/RR stage and the EX stage, be careful to accommodate them all.

EXECUTE

In this stage, you will perform all necessary arithmetic calculations, determine if a branch or a jump should be taken, and the target address for the PC. First, you should have an ALU that takes in a function code and two inputs. For one of the inputs, you will need to choose between the immediate value and the second source register value (this is to accommodate R-type and I-type instructions). You will also need hardware

to determine if you should take the branch or not (this can be done using a comparator). You should also have a dedicated adder unit separate from your ALU that determines the branch target address. Note that for JALR, you may simply pass the appropriate source register value back to the PC selection logic in the FETCH stage (ie. just feed it into the multiplexer). In the pipeline buffer between the EXECUTE and the MEMORY stages, you should place the control signals needed for the MEMORY and the WRITE BACK stages, as well as the appropriate values you calculated in this (the EXECUTE) stage.

MEMORY

This is a relatively simple stage. All you need to do is to use the value calculated in the EXECUTE stage as the address for the RAM. **Note that you must use the maximum address length for the RAM block - this is 24 bits.** To accomplish this, simply take the lower 24 bits of the calculated address. You should supply additional control signals to the RAM block in order to control reading and writing to it. These bits will be set based on if the instruction is a LW, SW, or otherwise. The buffer between the MEMORY and the WRITE BACK stage should contain the appropriate control signals for the WRITE BACK stage, the results from the EXECUTE stage, and the results from the MEMORY stage.

WRITE BACK

This is yet another simple pipeline stage. We have two sources from which we may write the registers from, the ALU and the RAM block. We will have a multiplexer that selects between these two; the output of the selection will be hooked up to the input of the register file. It would also be wise to have a signal that enables the register file for writing - as we may not always want to write to a register.

Stalling the Pipeline

One must stall the pipeline when an instruction cannot proceed to the next stage. This usually happens because of a data or a control flow hazard. For example, I may have two simultaneous add instructions and the destination register of the first add is used as a source register of the second add. If we do not have forwarding, we must stall the second add instruction until the first add instruction has finished writing to the register file. The only stages we need to stall will be the FETCH and ID/RR stages. This is because these stages are the ones affected by control and data hazards. One might stall the FETCH and ID/RR stages until a register that an instruction in the ID/RR stage needs has been updated. One might also stall the FETCH and ID/RR stage when a BEQ is in the EXECUTE stage - you may decide to either keep the instructions in the FETCH or ID/RR stage or squash them depending on the outcome of the BEQ. One may stall the FETCH stage by simply latching on the same PC value and by not updating the pipeline buffer. One may stall the ID/RR stage by latching on the appropriate signals for a no-operation into the pipeline buffer. Please note that you will be implementing data forwarding in the next few sections. However, pipeline stalls may still be needed for BEQ instrutions.

Flushing the Pipeline

When one performs a BEQ or a JALR, one cannot know the outcome of these instructions until we have reached the EXECUTE stage. Once the BEQ and the JALR instructions have reached the EXECUTE stage, we will have fetched two instructions that come after them. For the BEQ instruction, we may or may not execute the two instructions we have already fetched. If the BEQ is successful, then we must squash the two instructions in the FETCH and ID/RR stages. We can do this by clearing out the pipeline registers for these two stages. For JALR, we must always clear out the pipeline registers, since it is essentially an unconditional branch.

Forwarding

If you really liked the busy-bit/read-pending signal forwarding described in lecture and in your book, feel free to use that. We present an alternate way to do forwarding in this section. I will describe a different way to do forwarding here.

Forwarding is one way to increase the performance of the pipeline. This allows us to get values computed in stages beyond ID/RR back to ID/RR so that we do not have to stall the instruction. I would strongly recommend against using the busy bit/read pending bit strategy suggested in the book - this has some very nasty edge cases and requires much more logic than necessary. I would recommend that you make a forwarding unit that implements various stock rules. The forwarding unit should take in the two register values you are reading, the output value from the EXECUTE stage, the output value from the MEMORY stage, and the output value from the WRITE BACK stage. To forward a value from a future stage back to ID/RR, you must check to see if the destination register number from a particular stage is equal to your source register numbers in the ID/RR stage. If so, you must forward the value form that stage to your ID/RR stage. You shouldn't update the value of the register when you forward the value back - writes to the register file should only occur in the WRITE BACK stage. Of course, forwarding cannot save you from one situation: when the destination register of a LW instruction is the source register of an instruction immediately after it. In this case, you must stall the instruction in the ID/RR stage. I will leave it to you to flesh out all of the stock rules.

Testing

When you have constructed your pipeline, you should test it instruction by instruction to see if you have all the necessary components to ensure proper execution. Then you should make the following program: increment $a0 from 0 to 10 using a loop. Then, increment memory address 0x0 from 0 to 10 using a loop. Call this program `test.s`, and use the assembler to compile it. Be careful to only use the instructions listed in the appendix - there are some subtle points in having a separate instruction and data memory. Load the assembled program into the instruction memory and let your processor execute it.

Deliverables

You are to turn in the following:

- `lastname_firstname.circ`, the logisim file that contains all of your subcircuits and your processor. Replace lastname and firstname with you actual last name and first name.
- `test.s`, your assembly program

in a `.tar.gz` archive.

Appendix A: LC 2200-32 Processor Reference Manual

The LC-2200-32 is a 32-bit computer with 16 general registers plus a separate program counter (PC) register. All addresses are word addresses. Register 0 is wired to zero: it always reads as zero and writes to it are ignored.
There are four types of instructions: R-Type (Register Type), I-Type (Immediate value Type), J-Type (Jump Type), and O-Type (Other Type).

Here is the instruction format for R-Type instructions (ADD,NAND):

Bits	31 - 28	27 - 24	23 - 20	19 - 4	3 - 0
Purpose	opcode	RX	RY	Unused	RZ

Here is the instruction format for I-Type instructions (ADDI,LW,SW,BEQ):

Bits	31 - 28	27 - 24	23 - 20	19 - 0
Purpose	opcode	RX	RY	2's Complement Offset

Here is the instruction format for J-Type instructions (JALR):

Bits	31 - 28	27 - 24	23 - 20	19 - 0
Purpose	opcode	RX	RY	Unused (all 0s)

Table 1: Registers and their Uses

Register Number	Name	Use	Callee Save?
0	$zero	Always Zero	NA
1	$at	Reserved for the Assembler	NA
2	$v0	Return Value	No
3	$a0	Arg or Temp Register	No
4	$a1	Arg or Temp Register	No
5	$a2	Arg or Temp Register	No
6	$a3	Arg or Temp Register	No
7	$a4	Arg or Temp Register	No
8	$s0	Saved Register	Yes
9	$s1	Saved Register	Yes
10	$s2	Saved Register	Yes
11	$s3	Saved Register	Yes
12	$k0	Reserved for OS and Traps	NA
13	$sp	Stack Pointer	No
14	$fp	Frame Pointer	Yes
15	$ra	Return Address	No

Table 2: Assembly Language Instruction Descriptions

Name	Type	Example	Opcode	Action
add	R	add $v0, $a0, $a2	0000	Add contents of RY with the contents of RZ and store the result in RX.
nand	R	nand $v0, $a0, $a2	0001	NAND contents of RY with the contents of RZ and store the result in RX.
addi	I	addi $v0, $a0, 7	0010	Add contents of RY to the contents of the offset field and store the result in RX.
lw	I	lw $v0, 0x07($sp)	0011	Load RX from memory. The memory address is formed by adding the offset to the contents of RY.
sw	I	sw $a0, 0x07($sp)	0100	Store RX into memory. The memory address is formed by adding the offset to the contents of RY.
beq	I	beq $a0, $a1, done	0101	Compare the contents of RX and RY. If they are the same, then branch to address PC + 1 + Offset, where PC is the address of the beq instruction. **Memory is word addressable**.
jalr	J	jalr $at, $ra	0110	First store PC + 1 in RY, where PC is the address of the jalr instruciton. Then branch to the address in RX. If RX = RY, then the processor will store PC + 1 into RY and end up branching to PC + 1.