1 Project Requirements

In this project, you will make a pipelined processor that implements the LC3-2200b 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: LC3-2200b Instruction Set Architecture to understand the ISA that you will be implementing. Please note, this is not the same ISA that you implemented in Project 1, and there is no concept of condition codes. We have provide you with a Logisim file with the some of the structure laid out.

BEFORE WE GO INTO THE DETAILS, WE WANT TO STRESS THAT THIS IS PROBABLY THE HARDEST PROJECT OF THE SEMESTER. START NOW!

2 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, a BNE 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 four 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.

Here is a hint for deciding what to pass through buffers in general: Think of what each instructions needs, and pass a union of those. (By union we mean the mathematical union, for example say I1 needs PC and Rx, while I2 needs Rx and Ry, then you should pass PC, Rx and Ry through the buffer)

INSTRUCTION DECODE/REGISTER READ

In this stage, you will have to decode the instruction and read the appropriate registers. Please look at Appendix A: LC3-2200b Instruction Set Architecture 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 LDR, STR, or some other instruction that does not effect memory. 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 consecutive ADD instructions and the destination register of the first ADD is used as a source register of the second ADD. However, this kind of stall is taken care of by data forwarding. The next section will describe data forwarding in more detail. There is one small edge case in which you must stall–this is described in the data forwarding section.

For branch instructions, you **will not stall**. Rather, the assembler will automatically populate a *delay slot* when assembling programs. **For this project, you can assume the instruction immediately following a BEQ or BNE instruction can be safely executed regardless of whether the branch is taken.**

To stall the pipeline, the stages preceding the stalled stage should disable writes into their buffers, i.e. they should continue to output the previous value into the next stage. The stalled stage itself will output NOOP (example, ADD $zero, $zero, $zero) instructions down the pipeline until the cause of the stall finishes.

Flushing the Pipeline

The JALR instruction sets the PC in the DECODE stage of the pipeline, however, this means that an instruction has been fetched that must not be executed. You must hence squash the instruction that is in the FETCH stage. This can be done by invalidating the FBUF or simply clearing all the fields of FBUF to zero in the next clock cycle.

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.

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 LDR 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 stall rules.

3 Testing

An assembler for the LC3-2200b ISA will be released by Monday.

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.

4 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.

5 Appendix A: LC3-2200b Instruction Set Architecture

The LC3-2200b (Little Computer 3-2200 Type B) is a simple, yet capable pipelined computer architecture. The LC3-2200b combines attributes of both the LC3 architecture, and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC3-2200b is a **word-addressable**, **32-bit** computer. **All addresses refer to words**, i.e. the first word (four bytes) in memory occupies address 0x0, the second word, 0x1, etc.

All memory addresses are truncated to 24 bits on access, discarding the 8 most significant bits if the address was stored in a 32-bit register. This provides roughly 67 MB of addressable memory.

5.1 Registers

The LC3-2200b has 16 general-purpose registers. While there are no hardware-enforced restraints on the uses of these registers, your code is expected to follow the conventions outlined below.

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	Argument 1	No
4	$a1	Argument 2	No
5	$a2	Argument 3	No
6	$t0	Temporary Variable	No
7	$t1	Temporary Variable	No
8	$t2	Temporary Variable	No
9	$s0	Saved Register	Yes
10	$s1	Saved Register	Yes
11	$s2	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

1. **Register 0** is always read as zero. Any values written to it are discarded. **Note:** for the purposes of this project, you must implement the zero register. Regardless of what is written to this register, it should always output zero.
2. **Register 1** is a general purpose register. You should not use it because the assembler will use it in processing pseudo-instructions.
3. **Register 2** is where you should store any returned value from a subroutine call.
4. **Registers 3 - 5** are used to store function/subroutine arguments. **Note:** registers 2 through 8 should be placed on the stack if the caller wants to retain those values. These registers are fair game for the callee (subroutine) to trash.
5. **Registers 6 - 8** are designated for temporary variables. The caller must save these registers if they want these values to be retained.
6. **Registers 9 - 11** are saved registers. The caller may assume that these registers are never tampered with by the subroutine. If the subroutine needs these registers, then it should place them on the stack and restore them before they jump back to the caller.

7. **Register 12** is reserved for handling interrupts. While it should be implemented, it otherwise will not have any special use on this assignment.
8. **Register 13** is your anchor on the stack. It keeps track of the top of the activation record for a subroutine.
9. **Register 14** is used to point to the first address on the activation record for the currently executing process. Don't worry about using this register.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing. It is automatically used for this purpose by the subroutine jump instruction.

5.2 Instruction Overview

The LC3-2200b supports a variety of instruction forms, only a few of which we will use for this project. The instructions we will implement in this project are summarized below.

Table 2: LC3-2200b Instruction Set

	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4	3 2 1 0
ADD	0000	DR	SR1	unused	SR2
ADDI	0001	DR	SR1	immval20	
NAND	0010	DR	SR1	unused	SR2
BEQ	0011	SR1	SR2	PCoffset20	
JALR	0100	RA	AT	unused	
LDR	0101	DR	BaseR	offset20	
LEA	0110	DR	unused	PCoffset20	
STR	0111	SR	BaseR	offset20	
SHF	1000	DR	SR1	shiftvec20[1]	
BNE	1001	SR1	SR2	PCoffset20	
HALT	1111	unused			

[1] See Section SHF for the format of the shift vector.

Note: A Noop can be simulated by doing ADD $zero, $zero, $zero. The assembler supports the NOOP pseudocode, and outputs the ADD instruction as shown in the above line.

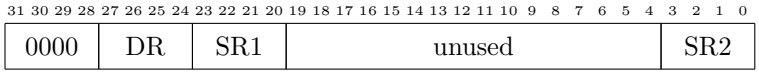
5.3 Detailed Instruction Reference

5.3.1 ADD

Assembler Syntax

```
ADD    DR, SR1, SR2
```

Encoding



Operation

```
DR = SR1 + SR2;
```

Description

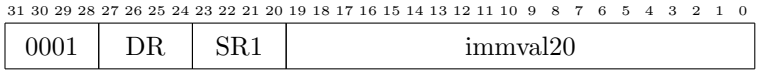
The ADD instruction adds the source operand obtained from SR2 to the source operand obtained from SR1. The result is stored in DR.

5.3.2 ADDI

Assembler Syntax

```
ADDI   DR, SR1, immval20
```

Encoding



Operation

```
DR = SR1 + SEXT(immval20);
```

Description

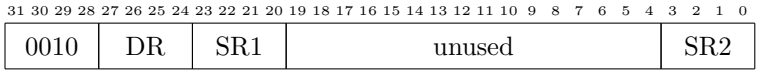
The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The second source operand is added to the first source operand, and the result is stored in DR.

5.3.3 NAND

Assembler Syntax

```
NAND   DR, SR1, SR2
```

Encoding



Operation

```
DR = ~(SR1 & SR2);
```

Description

The NAND instruction performs a logical NAND (AND NOT) on the source operands obtained from SR1 and SR2. The result is stored in DR.

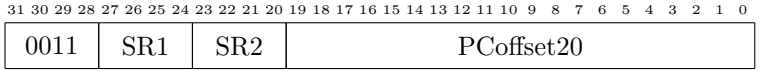
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND  DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

5.3.4 BEQ

Assembler Syntax

```
BEQ   SR1, SR2, immval20
```

Encoding



Operation

```
if (SR1 == SR2) {
    PC = PC + SEXT(PCoffset20);
}
```

Description

The BEQ instructions compares source operands SR1 and SR2, and if they are equal the program branches to the location specified by adding the sign-extended PCoffset20 field to the incremented PC (address of instruction + 1). **In other words, the PCoffset20 field specifies the number of instructions, forwards or backwards, to branch over.**

5.3.5 JALR

Assembler Syntax

```
JALR   AT, RA
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0100	RA	AT	unused

Operation

```
PC = AT;
RA = PC(of instruction) + 1;
```

Description

The PC is loaded with the contents of register AT and the RA register is simultaneously populated with the current PC (address of instruction + 1). The computer resumes execution at the new PC.

5.3.6 LDR

Assembler Syntax

```
LDR    DR, offset20(BaseR)
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0101	DR	BaseR	offset20

Operation

```
DR = MEM[BaseR + SEXT(offset20)];
```

Description

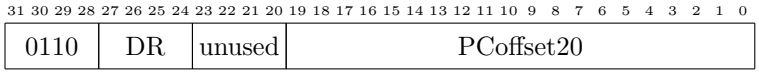
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

5.3.7 LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding



Operation

```
DR = PC + SEXT(PCoffset20);
```

Description

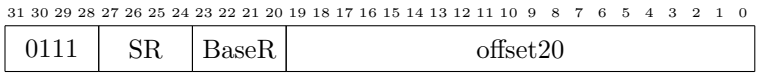
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the incremented PC (address of instruction + 1). This instruction effectively performs the same computation as the BR instruction, but rather than performing a branch, merely stores the computed address into register DR.

5.3.8 STR

Assembler Syntax

```
STR    SR, offset20(BaseR)
```

Encoding



Operation

```
MEM[BaseR + SEXT(offset20)] = SR;
```

Description

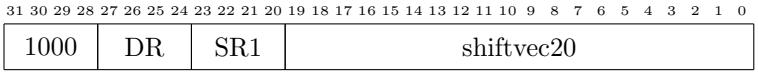
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the contents of the register specified by bits [23:20]. The 32-bit word obtained from register SR is then stored at this address.

5.3.9 SHF

Assembler Syntax

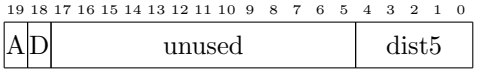
```
SHFLL  DR, SR1, dist5    ; left shift logical
SHFRL  DR, SR1, dist5    ; right shift logical
SHFRA  DR, SR1, dist5    ; right shift arithmetic
```

Encoding



Shift Vector Encoding

The embedded shift vector is a 20-bit value that defines both the distance of the shift and the type of shift to perform.



Operation

```
if (shiftvec[D] == 0) {
    DR = SR1 << shiftvec[dist5];
} else {
    if (shiftvec[A] == 0) {
        DR = SR1 >> shiftvec[dist5],0;
    } else {
        DR = SR1 >> shiftvec[dist5],SR1[31];
    }
}
```

Description

If the D bit of the shift vector is 0, the source operand in SR1 is shifted left by the number of bit positions indicated by the dist5 field. If D is 1, the source operand is shifted to the right by dist5 bits.

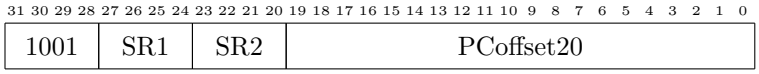
When shifting to the right, the A bit of the shift vector indicates whether the sign bit of the original source operand is preserved. When A is set to 1, the right shift is an arithmetic shift and the original SR1[31] is shifted into the vacated bit positions. The result stored in DR. Otherwise the shift is a logical shift and zeroes are shifted in.

5.3.10 BNE

Assembler Syntax

```
BNE   SR1, SR2, immval20
```

Encoding



Operation

```
if (SR1 != SR2) {
    PC = PC + SEXT(PCoffset20);
}
```

Description

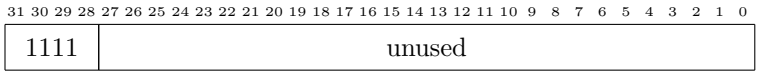
The BNE instructions compares source operands SR1 and SR2, and if they are **NOT** equal the program branches to the location specified by adding the sign-extended PCoffset20 field to the incremented PC (address of instruction + 1). **In other words, the PCoffset20 field specifies the number of instructions, forwards or backwards, to branch over.**

5.3.11 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

The machine is brought to a halt and executes no further instructions.