Extra Credit Project: Pipeline

1 Why Pipelining?

The datapath design that we implemented for Project 1 was, in fact, grossly inefficient. By focusing on increasing throughput, a pipelined processor can get more instructions done per clock cycle. In the real world, that means higher performance, lower power draw, and most importantly, happy customers!

2 Project Requirements

In this extra credit project, you will make a pipelined processor that implements the LC-22 ISA. There will be five stages in your pipeline:

1. **IF** - Instruction Fetch
2. **ID/RR** - Instruction Decode/Register Read
3. **EX** - Execute (ALU operations)
4. **MEM** - Memory (both reads and writes with memory)
5. **WB** - Writeback (writing to registers)

Before you move on, read Appendix A: LC-22 Instruction Set Architecture to understand the ISA that you will be implementing. Understanding the instructions supported by your ISA will make designing your pipeline much easier. We provide you with a CircuitSim file with the some of the structure laid out.

3 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 (Ch. 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.

1. IF Stage

The IF stage is responsible for:

- Getting the instruction from I-MEM at location PC
- Updating the PC

For normal sequential execution, we would update the PC by incrementing it by 1. Notice, however, that this may not be the case when executing a branch or JALR instruction. Hence, you will likely need to multiplex which value is used to update the PC. I-MEM has 16 address bits.

2. ID/RR Stage

The ID/RR stage is responsible for:

- Decoding the instruction

- Reading the appropriate registers

Please look at Appendix A: LC-22 Instruction Set Architecture in order to understand the instruction formats! You will be provided a dual ported register file (DPRF), which allows you to read from two registers and write one register all at the same time.

Some of the instructions require both inputs into to the ALU to be values pulled from the DPRF. However, other instructions contain a value within the instruction, such as an immval20, offset20, or PCoffset20 field. You may either pass all of these possible values to the next stage (requires bigger buffer registers), or condense them into just the values needed to execute the instruction in the following cycles (requires more logic, but buffer size can be optimized).

3. EX Stage

The EX stage is responsible for:

- Performing all necessary arithmetic and logic calculations
- Resolving any branch or JALR instructions

In the Execute (EX) stage, you will perform any arithmetic computations required by the instruction. This stage should host a complete ALU to perform the actual adding or NANDing as required by the instruction. For memory access instructions, this stage will perform the Base + Offset computation required to determine the memory address to access.

4. MEM Stage

The MEM stage is responsible for:

- Reading from or writing a result to memory

All you need to do is to use the value calculated in the EX stage as the address for the RAM. **Note that you must use the maximum address length for the RAM block - this is 16 bits.** To accomplish this, simply take the lower 16 bits of the calculated address. Depending on the instruction, this stage will need to pass either the value read from memory or the value computed in EX to the WB stage.

5. WB Stage

The WB stage is responsible for:

- Writing results back to the DPRF (dual-ported register file)

Depending on the instruction, you may need to write a value back to a register. To do this, your WB stage will attach to the data in and write enable inputs of the DPRF in ID/RR. Remember that the DPRF can write **and** read different registers **in the same clock cycle**, which is why WB and ID/RR can share the same register file. For instructions that do not write a register, your WB stage may not do anything at all.

4 General Advice

Subcircuits

For this project, we highly encourage using modular design and creating subcircuits when necessary. We **strongly** recommend using subcircuits when building your pipeline buffers, stages, and forwarding unit. A modular design will make it easier to debug and test your circuit during development.

Pipeline Buffers

For deciding what to pass through buffers, remember that we need to support the requirements of every possible instruction. Think of what each instruction needs to fulfill its duty, and pass a union of all those requirements. (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). You can also feel free to implement your hardware such that you re-use space in the buffer for different purposes depending on the instruction, but this is not required.

Control Signals

In the Project 1 datapath, recall that we had one main ROM that was the single source of all the control signals on the datapath. Now that we are spreading out our work across different stages of the pipeline, you have a choice of how to implement your signals!

The first thing to note is that in a pipelined processor, each stage is like a simple one-cycle processor that can do exactly ONE thing intended for that stage in a single cycle. In this sense, there is really no need for a control ROM anymore! Therefore in real processors, each stage of the pipeline is implemented using hardwired control as discussed in Chapter 3 of the textbook. However, to keep your design simple for debugging and getting it working, we are going to suggest using a control ROM to generate the needed control signals for the different independent stages of the pipelined processor.

There are two options:

1. You can either have a single large main ROM in ID/RR which calculates all the control signals for every stage.

   OR

2. you can have a small(er) ROM in each stage which takes in the opcode and assert the proper signals for that operation.

Note that if you choose the first method, you will need to pass all the signals needed for later stages through the earlier stages, and in the second method, you will need to pass the instruction opcode though all the stages so that you know which signals to assert during that stage.

Stalling and Data Forwarding

One must stall the pipeline when an instruction cannot proceed to the next stage because a value is not yet available to an instruction. This usually happens because of a data hazard. For example, consider two instructions in the following program:

1. `LW $t0, 5($t1)`
2. `ADDI $t0, $t0, 1`

Without stalling the ADDI instruction in the ID/RR stage, it will get an out of date value for $t0 from the regfile, as the correct value for $t0 isn't known the LW reaches the MEM stage! Therefore, we must stall. Consult the textbook (or your notes) for more information on data hazards.

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.

Note that you may eliminate a good deal of stalls by implementing data forwarding. This allows the ID/RR stage to retrieve values computed in later stages of the pipeline early so that stalling the instruction is not necessary. It is strongly recommended that you not use 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.

It is recommended 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 EX stage, the output value from the MEM stage, and the output value from the WB 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 from that stage to your ID/RR stage.

Note, 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, sometimes called "load-to-use", you must stall the instruction in the ID/RR stage. It is your job to flesh out all of the stall and forwarding rules.

Keep in mind: the zero register can never change, therefore it should not be considered for forwarding and stalling situations.

Branch Prediction

Since branch instructions (BLT/BGT/JALR) are resolved in EX, the pipeline may be unsure of which instructions are correct to fetch. We could stall fetching further instructions until resolution, but this is inefficient and naive. To better handle control hazards, we can "predict" which instruction could be correct.

For this project we will be predicting the **branch is not taken**, and so the pipeline will continue fetching sequentially. Upon resolving the branch, the pipeline should continue normally in the case of a correct prediction, or flush the incorrectly fetched instructions in the case of an incorrect prediction.

Flushing the Pipeline

For the BLT/BGT/JALR instructions, we calculate the target in the EX stage of the pipeline. However, the next two instructions the IF stage fetches while EX is computing the target may not be the next instructions we want to execute. When this happens, we must have a hardware mechanism to "cancel" or "flush" the incorrectly-fetched instructions after we realize they are incorrect.

In implementing your flushing mechanism, it is **highly recommended** that you avoid the asynchronous clear feature of registers in CircuitSim, as this may cause timing issues. Instead, we suggest using a multiplexer to selectively send a NOOP into the buffer input.

5 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.

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 both the instruction memory and the data memory and let your processor execute it. Any writes to memory will only affect the data memory.

6 Grading

You may receive **up to 3 points of extra credit on your final grade** by completing this project.

One point will be given for each of the following:

- Your pipeline produces the correct output when running our test program.
- All stages of the pipeline have been implemented with all required components and connections visible.
- Your pipeline correctly implements register forwarding. Note, you may design a pipeline without register forwarding, but it will not receive this point.

We will not accept regrades for the extra credit project.

7 Deliverables

Please submit all of the required files in a **.tar.gz** archive generated by one of the following:

- **On Linux/Mac:** Use the provided Makefile. The Makefile will work on any Unix or Linux-based machine (on Ubuntu, you may need to `sudo apt-get install build-essential` if you have never installed the build tools). Run `make submit` to automatically package your project into the correct archive format.
- **On Windows:** Use the provided submit.bat script. Submitting through this method will require 7zip (https://www.7-zip.org/) to be installed on your system. Run `submit.bat` to automatically package your project into the correct archive format. **Note:** Sometimes 7zip isn't added to your path when you install it, and you may get an error. If this happens, try running `set PATH=%PATH%;C:\Program Files\7-Zip\`.

The generated archive should contain at a minimum the following files:

- `LC22-Pipeline.sim`

Always re-download your assignment from Canvas after submitting to ensure that all necessary files were properly uploaded. If what we download does not work, you will get a 0 regardless of what is on your machine.

8 Appendix A: LC-22 Instruction Set Architecture

The LC-22 is a simple, yet capable computer architecture. The LC-22 combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC-22 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 16 bits on access, discarding the 16 most significant bits if the address was stored in a 32-bit register. This provides 256 KB of addressable memory.

8.1 Registers

The LC-22 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	Assembler/Target Address	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 used to hold the target address of a jump. It may also be used by pseudo-instructions generated by the assembler.
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 it jumps 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 the everchanging top of the stack; it keeps track of the top of the activation record for a subroutine.
9. **Register 14** is the anchor point of the activation frame. It is used to point to the first address on the activation record for the currently executing process.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing.

8.2 Instruction Overview

The LC-22 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: LC-22 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
NAND	0001	DR	SR1	unused SR2
ADDI	0010	DR	SR1	immval20
LW	0011	DR	BaseR	offset20
SW	0100	SR	BaseR	offset20
BR	0101	unused		offset20
JALR	0110	RA	AT	unused
HALT	0111	unused		
BLT	1000	SR1	SR2	offset20
BGT	1001	SR1	SR2	offset20
LEA	1010	DR	unused	PCoffset20

8.2.1 Conditional Branching

Branching in the LC-22 ISA is a bit different than usual. We have a set of branching instructions including BR, an unconditional branch, as well as BLT and BGT, which offer the ability to branch upon a certain condition being met. The BLT and BGT instructions use comparison operators, comparing the values of two source registers. If the comparisons are true (for example, with the BGT instruction, if SR1 > SR2), then we will branch to the target destination of incrementedPC + offset20.

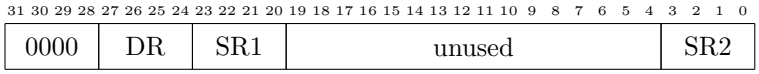
8.3 Detailed Instruction Reference

8.3.1 ADD

Assembler Syntax

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

Encoding



Operation

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

Description

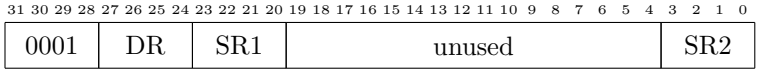
The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

8.3.2 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.

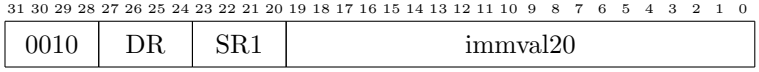
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

8.3.3 ADDI

Assembler Syntax

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

Encoding



Operation

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

Description

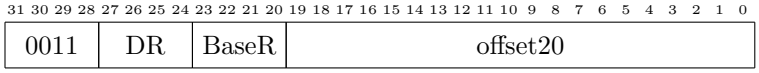
The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

8.3.4 LW

Assembler Syntax

```
LW    DR, offset20(BaseR)
```

Encoding



Operation

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

Description

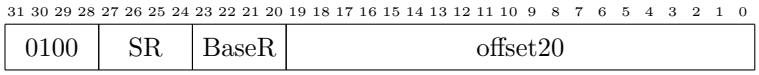
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

8.3.5 SW

Assembler Syntax

```
SW    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 then 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.

8.3.6 BR

Assembler Syntax

```
BR  offset20
```

Encoding

Operation

```
PC = incrementedPC + offset20
```

Description

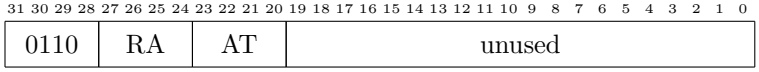
A branch is unconditionally taken. The PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

8.3.7 JALR

Assembler Syntax

```
JALR   RA, AT
```

Encoding



Operation

```
RA = PC;
PC = AT;
```

Description

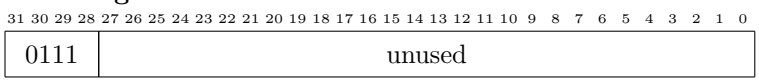
First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

8.3.8 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

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

8.3.9 BLT

Assembler Syntax

```
BLT  SR1, SR2, offset20
```

Encoding

Operation

```
if (SR1 < SR2) {
    PC = incrementedPC + offset20
}
```

Description

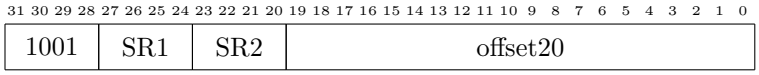
A branch is taken if SR1 is less than SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

8.3.10 BGT

Assembler Syntax

```
BGT  SR1, SR2, offset20
```

Encoding



Operation

```
if (SR1 > SR2) {
    PC = incrementedPC + offset20
}
```

Description

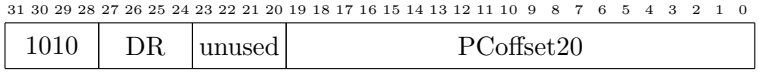
A branch is taken if SR1 is greater than SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

8.3.11 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). It then stores the computed address into register DR.