Computer Systems and Networks

Prof. Ramachandran, Prof. Daglis, Prof. Sarma

Extra Credit Project: Pipelining

Due: **April 22nd 2025**

1 Introduction

The datapath design that we implemented for Project 1 (LC-3200) was, in fact, grossly inefficient. This is your chance to show that you can optimize the performance of LC-3200 using the concepts you learned in this class. 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 consumption, and most importantly happy customers!

2 Project Requirements

In this extra credit project, you will make a pipelined processor that implements the LC-3200b 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-3200b 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.

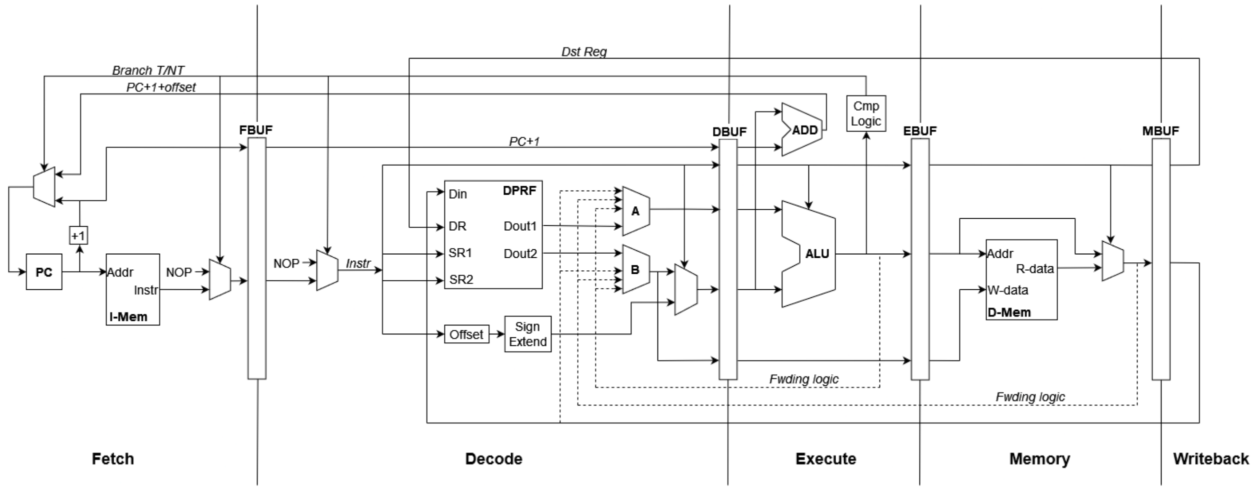

Figure 1: Pipeline Schematic

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.

3.1 IF Stage

Functionality:

- **PC Update**: In the IF stage, we need to update the PC's value in order to fetch the proper next instruction. For normal sequential execution, the IF stage should update the PC by incrementing it by 1. However, in the case of branches such as with JALR and BEQ, the PC's value should be set to whatever new address was calculated during the EX Stage.
- **Fetch from I-MEM**: We will then use the PC value to index into I-MEM and retrieve an instruction. Note that I-MEM has 16 address bits, so you will need some circuit to reduce the PC's bits from 32 bits to 16 bits.
- **Selecting the next PC**: Choosing whether the PC should be updated to PC+1 or to the address calculated in EX can be accomplished with a multiplexer. Consider what the selector bits for this MUX must be; they come from the EX Stage.

3.2 ID/RR Stage

Functionality:

- **Obtaining Data from the Instruction**: The first thing to accomplish in the ID/RR stage is to obtain the opcode of the 32 bit instruction that was fetched in the IF stage. Consider all other values that need to be obtained directly from the instruction, such as register numbers and the immediate/offset value.
- **Decoding the Instruction**: Once we have the opcode for our instruction, we need to determine the specific control signals that need to be asserted. Like our LC-3300 uniprocessor that we implemented in projects 1 and 2, we can use ROMs that store microcode for each instruction's control signals. Given the opcode, the ROMs should output the necessary control signals required for a specific instruction.
- **Reading Registers**: Our instruction might require that we read from one or two registers. If reading from a single register, we use only 1 port in our dual-ported register file (DPRF), using the register number specified in the instruction to read a value. If given two register numbers, then we utilize both ports in the DPRF to read from both simultaneously.
- **Selecting "A" and "B"**: Like the ALU in projects 1 and 2, the EX Stage utilizes two operands, "A" and "B". You must select the proper value for A and the proper value for B based on the instruction being executed (for example, select PCOffset20, immval20, a register value, etc.).

Considerations for Implementation:

- **ROMs**: Unlike the processor implemented in projects 1 and 2, there is no need for next state bits. Implement a component using one or more ROMs that takes in an opcode and returns the control signals associated with that opcode.
- **Dual Ported Register File**: When implementing the DPRF, consider that you only need to read from two registers at once. You will not have to write to more than one register simultaneously.
- **Data Forwarding**: If you decide to implement data forwarding, note that your selector for A and B should also be able to select any forwarded values.

3.3 EX Stage

Functionality:

- **Calculation**: The EX Stage must compute calculations using A and B. For example, it will calculate the sum for an ADD instruction, or the Base + Offset computation for LW and SW. This should be done with a complete ALU, capable of performing adding, nanding, and any other required operation.
- **Branch Evaluation**: The EX Stage must evaluate a branch condition, and then determine if the branch is taken or not taken. This can be performed by installing a comparison logic unit.
- **Branching**: As previously mentioned, the IF stage will require information about the branch evaluation performed in the EX Stage. Consider implementing forwarding lines that can forward this information between stages.

3.4 MEM Stage

Functionality:

- **Address Calculation**: The effective address for memory operations is derived from the Execute (EX) stage, typically involving arithmetic operations or immediate values combined with base register values. In systems with a 16-bit address space, it is crucial to use only the lower 16 bits of the calculated address, requiring a masking operation to discard any higher-order bits.
- **Read Operation**: Load instructions necessitate reading data from the memory address calculated in the MEM stage. This involves initiating a memory access with the calculated address and retrieving the corresponding data.
- **Write Operation**: Store instructions require writing data from a source register to the memory address determined in the MEM stage. The operation must ensure data is correctly written to the intended memory location.

3.5 WB Stage

Functionality:

- The WB stage selectively writes values back to the registers. This involves interfacing directly with the data in and write enable inputs of the DPRF, ensuring that results of computations or memory operations are correctly stored.
- The dual-ported nature of the DPRF allows simultaneous read and write operations on different registers within the same clock cycle. This design facilitates sharing of the DPRF between the ID/RR and WB stages.
- Control Logic: Implementing control logic within the WB stage is crucial for determining whether a write-back operation is required based on the instruction type.
- Data Selection: The WB stage must select the correct data source for write-back operations. This is typically between data fetched from memory or computation results generated in previous stages. Multiplexers, guided by control signals, can be a good design choice.

4 General Advice

4.1 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.

4.2 Pipeline Buffers

- Identify and support the requirements of all possible instructions by analyzing their needs.
- Pass a union of all requirements through the buffers to ensure functionality across diverse instructions.
- Optionally, implement dynamic buffer space utilization to optimize for instruction-specific requirements.

4.3 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 a few options for how to implement control signals:

1. Implement a single large main ROM in ID/RR which calculates all the control signals for every stage.
2. Implement small(er) ROMs in each stage that takes in the opcode and assert the proper signals for that operation.
3. Use hardwired combinational logic to obtain control signals without ROMs altogether.

4.4 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 that forwarding cannot save you from one situation: when the destination register of an 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.

4.4.1 Stalling

- Initiate stalling when data hazards prevent instruction progression to maintain data integrity.
- Implement stalling by disabling buffer writes in the preceding stages and issuing NOP instructions until the hazard is resolved.

4.4.2 Data Forwarding

- Use data forwarding to minimize stalls by allowing early access to values computed at later stages.
- Design a forwarding unit that evaluates the necessity for forwarding based on register comparisons.
- Address limitations, acknowledging scenarios like 'load-to-use' hazards that still require stalling.
- The priority encoder is a hardware component in CircuitSim that can choose priority between stages, resolving all WAW hazards without additional bubbles.

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

4.5 Branch Prediction

Since branch instructions (BEQ/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.

- Always predict branch-not-taken, and flush the IF and ID/RR stages when a branch is taken.
- Address control hazards by predicting branch outcomes, with a default prediction that the branch is not taken.
- Implement hardware mechanisms to handle both correct predictions (continue normally) and incorrect predictions (flush incorrectly fetched instructions).
- Forward signals between stages (as needed) to alter the course of execution upon a branch.
- Develop a mechanism to flush the preceding stages upon a branch. **Hint:** What hardware component can we use to send a NOP into the FBUF and DBUF instead of the instruction that is squashed.
- Avoid using the "clear" port on registers to prevent timing/clock issues which operate asynchronously.

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 similar to how you did in Projects 1 and 2.

Note: There is currently NO autograder for this project, so the only way to test if your pipeline is working successfully is to test it locally and verify that the final state of the registers is as expected. You can use the simulator.py or the provided Datapath from Project 1 to verify its functionality. This is the same way TAs will grade your submissions.

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 Report

Alongside the project, you will be required to submit a written report, rough 2-3 pages in length. The report should be presentable, with appropriate formatting.

Contents of the report may include, but are not limited to:

- Explanation of how to load your ROM(s) with your microcode.
- Explanation of the pipeline implementation (in particular the design of each stage and the data forwarding mechanism).
- Challenges that were faced during development.
- Results relating to cycle count when running the pow.s file, and any associated pipelining metrics that were taught in class.
- Potential areas of improvement or further optimization.

Submissions without a report will result in no extra credit points.

7 Grading

You may receive **up to 4 points of extra credit on your final grade** by completing this project.

- Submissions without an adequate report will receive no credit for the project.
- Partial credits may or may not be awarded for effort given to a pipeline with minor errors.
- We will not accept regrades for the extra credit project. You cannot use late days on this project.

8 Deliverables

To submit your project, you need to upload the following files to Gradescope:

- LC-3300b.sim
- microcode.xlsx (if applicable, but not required)
- pipeline-report.pdf

Always re-download your assignment from Gradescope 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.

9 Appendix A: LC-3200b Instruction Set Architecture

The LC-3200b is a simple, yet capable computer architecture. The LC-3200b ISA is the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200 with a few additional instructions. We highly recommend reading the textbook to understand how an ISA works. It may also provide further hints on how to implement some of the instructions

The LC-3200b 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 roughly 64 KB of addressable memory.

9.1 Registers

The LC-3200b 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 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. Pseudo-instructions generated by the assembler may also be used.
3. **Register 2** is where you should store any returned value from a subroutine call. A quick way to check if **pow.s** runs correctly to verify if this register's value is correct.
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 the subroutine never tampered with these registers. 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 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.

9.2 Instruction Overview

The LC-3200b 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-3200b 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	
BEQ	0101	SR1	SR2	offset20	
JALR	0110	AT	RA	unused	
HALT	0111	unused			
BGT	1000	SR1	SR2	offset20	
LEA	1001	DR	unused	PCoffset20	

9.2.1 Conditional Branching

Branching in the LC-3200b ISA is slightly different than usual. We have a set of branching instructions including both BEQ and BGT, which offer the ability to branch upon a certain condition being met. These 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.

HINT: You can reuse microstates for different instructions for a more efficient microcode!

9.3 Detailed Instruction Reference

9.3.1 ADD

Assembler Syntax

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

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
0000	DR	SR1	unused	SR2

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.

9.3.2 NAND

Assembler Syntax

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

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
0001	DR	SR1	unused	SR2

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. The following assembly:

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

achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

9.3.3 ADDI

Assembler Syntax

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

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
0010	DR	SR1	immval20

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.

9.3.4 LW

Assembler Syntax

```
LW    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
0011	DR	BaseR	offset20

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.

9.3.5 SW

Assembler Syntax

```
SW    SR, 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
0100	SR	BaseR	offset20

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.

9.3.6 BEQ

Assembler Syntax

```
 BEQ  SR1, SR2, offset20
```

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	SR1	SR2	offset20

Operation

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

Description

A branch is taken if SR1 is equal to 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].

9.3.7 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

0110	AT	RA	unused

Operation

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

Description

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

9.3.8 HALT

Assembler Syntax

```
HALT
```

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

0111	unused

Description

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

9.3.9 BGT

Assembler Syntax

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

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

1000	SR1	SR2	offset20

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].

9.3.10 LEA

Assembler Syntax

```
LEA    DR, label
```

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

1001	DR	unused	PCoffset20

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 in the register DR.