CS 2200 Systems and Networking

Prof. Ramachandran & Prof. Daglis

Pipelining Extra Credit: LC-3300-pipe

Due: **April 22nd 2024**

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-3300-pipe 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-3300-pipe 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.

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.

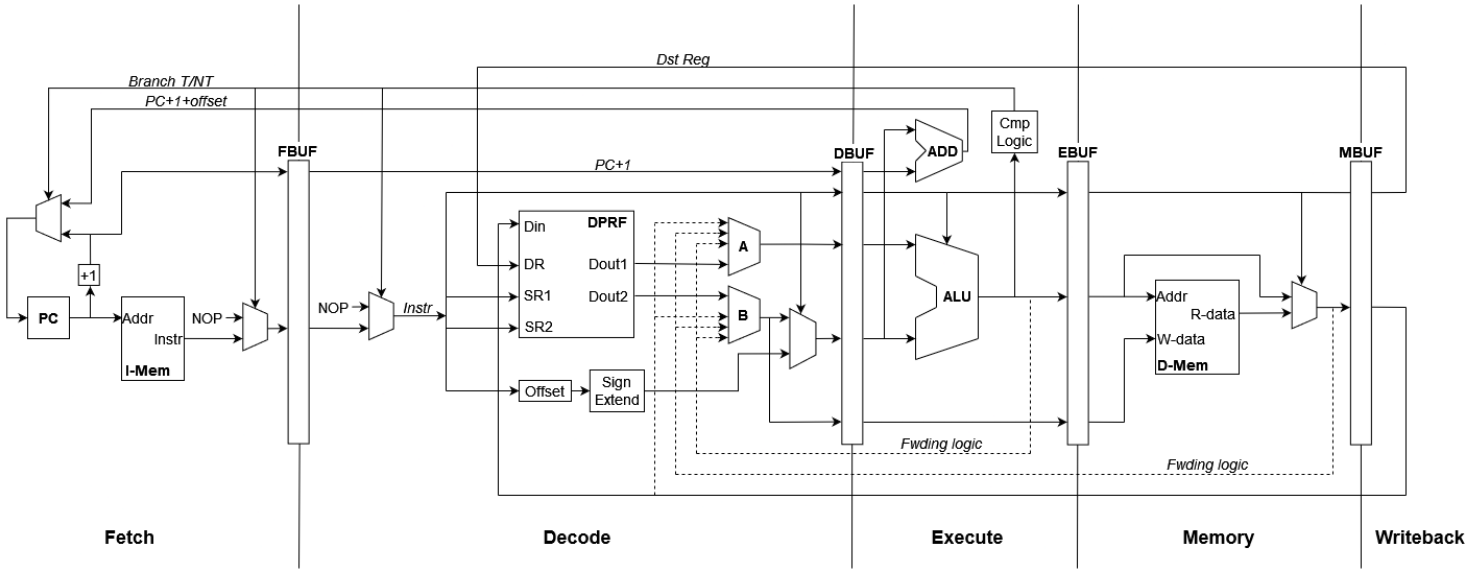


Figure 1: Pipeline Diagram

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.

Considerations for Implementation:

- **Selecting the proper 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.

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. 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, shifting, 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.

Considerations for Implementation:

- **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.

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.

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.

Considerations for Implementation:

- **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 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.

4.2 Control Signals

- Reflect on the shift from a single ROM source for control signals to more flexible implementation strategies.
- Consider each pipeline stage as a standalone processor that performs a specific task within one cycle, reducing the need for centralized control ROM.
- For simplicity and ease of debugging, a control ROM is suggested to generate control signals for each pipeline stage.

4.2.1 Control Signal Implementation Options

We have provided an **optional** microcode sheet that you can use to translate an opcode into signals. It is up to the programmer to decide what signals are needed; you will likely not need to use every column.

1. Opt for smaller, stage-specific ROMs that generate signals based on the opcode, and pass the opcode through buffers.
2. Use a single, large main ROM at the ID/RR stage for all control signals, and pass all necessary control signals through buffers.
3. Other implementations for control signals are also accepted if they work.

4.3 Stalling and Data Forwarding

4.3.1 Stalling the Pipeline

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

4.3.2 Implementing Data Forwarding

- Utilize data forwarding to minimize stalls by enabling early access to values computed in 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.
- Data forwarding with WAW hazards
  - The priority encoder is a hardware component in CircuitSim that can choose priority between stages, resolving all WAW hazards with no additional bubbles.
  - Busy bits, as detailed in the textbook, involve stalling the pipeline upon encountering a WAW hazard, until the first instruction exits the WB stage

4.3.3 Special Considerations for Forwarding and Stalling

- Exclude the zero register from forwarding and stalling logic due to its immutable value.
- Implement selective forwarding logic for instructions that do not perform register writes.

Keep in mind: the zero register can never change, therefore it should not be considered for forwarding and stalling situations. Additionally not all instructions will be writing back to a register, so blindly checking bits [27-24] does not work for a lot of instructions.

Forwarding however 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.

4.4 Branch Prediction

- Always predict branch-not-taken, and clear IF and IDRR 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).

4.5 Flushing the Pipeline

- Develop a flushing mechanism for when branch instructions in the EX stage render previously fetched instructions incorrect.
- Avoid the asynchronous clear feature of registers to prevent timing issues, a multiplexer-based approach to selectively send NOOP instructions could be helpful.

5 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.

6 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. We will not provide an autograder due to varying implementations of the pipeline.

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.

7 Grading

You may receive **up to 4 points of extra credit on your final grade** by completing this project and submitting a fully functioning 5-stage pipeline.

Submissions without an adequate report will receive no credit for the project. Partial credits may 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-3300-pipe.sim`
- Microcode file (microcode.xlsx) if applicable
- Report file as a 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 not get credit regardless of what is on your machine.

9 Appendix A: LC-3300-pipe Instruction Set Architecture

The LC-3300-pipe is a simple, yet capable computer architecture. The LC-3300-pipe combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC-3300-pipe 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-3300-pipe 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 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-3300-pipe 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-3300-pipe 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		
SLL	1010	DR	SR1	unused	00	SR2
SRL	1010	DR	SR1	unused	01	SR2
ROL	1010	DR	SR1	unused	10	SR2
ROR	1010	DR	SR1	unused	11	SR2

9.2.1 Conditional Branching

Branching in the LC-3300-pipe ISA is a bit different than usual. We have a set of branching instructions including BEQ, BGT, and FABS 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. For FABS, if SR < 0 then we will branch to the series of microstates for negation.

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.

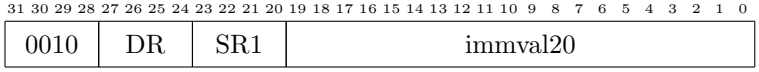
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

9.3.3 ADDI

Assembler Syntax

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

Encoding



Operation

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

Description

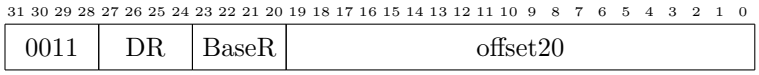
The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

9.3.4 LW

Assembler Syntax

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

Encoding



Operation

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

Description

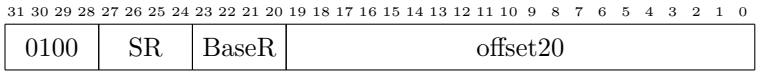
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

9.3.5 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.

9.3.6 BEQ

Assembler Syntax

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

Encoding

Operation

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

Description

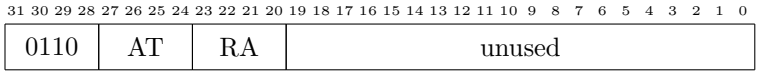
A branch is taken if SR1 is equal to SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

9.3.7 JALR

Assembler Syntax

```
JALR   AT, RA
```

Encoding



Operation

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

Description

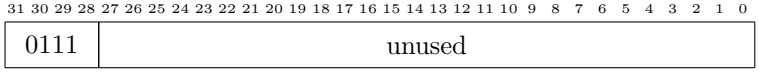
First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

9.3.8 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

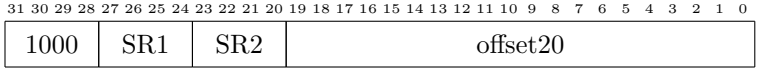
The machine is brought to a halt and executes no further instructions.

9.3.9 BGT

Assembler Syntax

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

Encoding



Operation

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

Description

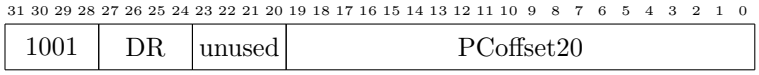
A branch is taken if SR1 is greater than SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

9.3.10 LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding



Operation

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

Description

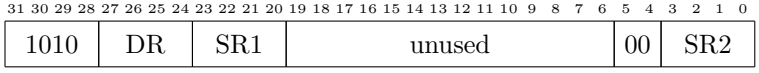
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the incremented PC (address of instruction + 1). It then stores the computed address into register DR.

9.3.11 SLL

Assembler Syntax

```
SLL    DR, SR1, SR2
```

Encoding



Operation

```
DR = SR1 << SR2;
```

Description

The value stored in SR1 is logically left shifted by the value stored in SR2, and the result is stored in DR.

9.3.12 SRL

Assembler Syntax

```
SRL    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
1010	DR	SR1	unused	01	SR2

Operation

```
DR = SR1 >> SR2;
```

Description

The value stored in SR1 is logically right shifted by the value stored in SR2, and the result is stored in DR.

9.3.13 ROL

Assembler Syntax

```
ROL    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
1010	DR	SR1	unused	10	SR2

Operation

```
DR = (SR1 << SR2) | (SR1 >> (32 - SR2));
```

Description

Bits in SR1 are "rotated" left by SR2 number of bits using circular shifting. During a left rotation, the bits that are shifted out from the left are brought back in on the right side.

9.3.14 ROR

Assembler Syntax

```
ROR    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
1010	DR	SR1	unused	11	SR2

Operation

```
DR = (SR1 >> SR2) | (SR1 << (32 - SR2));
```

Description

Bits in SR1 are "rotated" right by SR2 number of bits using circular shifting. During a right rotation, the bits that are shifted out from the right are brought back in on the left side.