Computer Systems and Networks

Prof. Forsyth

Extra Credit Project

Due: **December 3rd 2023**

1 Requirements

2 Introduction

The datapath design that we implemented for Project 1 (LC-2222) was, in fact, grossly inefficient. This is your chance to show you can optimize the performance of LC-2222 using the concepts you learnt in this class. Ranging from optimizing the datapath to building a pipeline, it is up to you how you want your "Little Computer" to look like! (Hint: It must be better than LC-2222.) In the real world, that means higher performance, lower power draw, and most importantly, happy customers!

3 Requirements

Before you begin, please ensure you have done the following:

- Download the proper version of CircuitSim. A copy of CircuitSim is available under Files on Gradescope. You may also download it from the CircuitSim website (https://ra4king.github.io/CircuitSim/). In order to run CircuitSim, Java must be installed. If you are a Mac user, you may need to right-click on the JAR file and select "Open" in the menu to bypass Gatekeeper restrictions.
- CircuitSim is still under development and may have unknown bugs. Please back up your work using some form of version control, such as a local/private git repository or Dropbox. **Do not use public git repositories; it is against the Georgia Tech Honor Code**.
- The LC-2222 assembler is written in Python. If you do not have Python 2.6 or newer installed on your system, you will need to install it before you continue.

4 What We Have Provided

- A completed LC-2222 datapath circuit (`LC-2222.sim`) from Project 1 is provided. In this extra credit project, you will optimize the version of the LC-2222 provided.
  - This version of the LC-2222 will not need to have support for interrupts.
  - This version of the LC-2222 will have a counter attached to it that increments every clock cycle. The counter stops once the datapath reaches the HALT instruction. If you make significant modifications to the provided datapath circuit, make sure that the inputs to the counter (the clock and the current instruction's op code) remain accurate.
- The ISA as detailed in Appendix A: LC-2222 Instruction Set Architecture is the same as in project 1, but is attached again for your reference. It is recommended to review the ISA, as understanding the instructions supported by your ISA will make designing your optimizations much easier.
- A microcode configuration spreadsheet `microcode.xslx` has been provided.
- A complete `pow.s` file you will use to test your design with has been provided.
- We have provided you with an assembler with support for the instructions in our ISA to test your program.

5 What You Will Do

Optimize the design in whatever creative ways possible to reduce the number of clock cycles necessary to run `pow.s`. This means that it needs to be able to work for any BASE and EXP. Then, you will write a **short report** on the optimizations you made and why it works.

Here are some ideas to begin optimizing your processor:

- Use a Dual Ported Register File (DPRF).
- Use a Two-Bus Design.
- Create a five stage pipeline that accepts the given ISA for the LC-2222.
- Create a shorter pipeline (less than five stages):
  - You could potentially combine IF and ID/RR, or combine MEM and WB, or any other combination that you find apt.
- If you are creating a pipeline, you can implement data forwarding.
- There are potentially more ways to optimize as well. When in doubt, ask the TA's!

You **MAY NOT** add new instructions to the Instruction Set Architecture (ISA).
You **MAY NOT** edit the `pow.s` file.
After completing your optimizations, you will need to write a **report** on each change you made and why it makes the processor faster. Talk about the improvement in terms of the metrics you learnt in class.

Submission without a report will receive 0 points.

6 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 if you are pursuing a pipeline approach. A modular design will make it easier to debug and test your circuit during development.

DPRF

If you decide to use the approach of a DPRF, some considerations will need to be made. Since there will be a provided register file to be used in the project, it will need to be extended functionally to allow two reads and one write to occur simultaneously. Additionally, you will need to consider how this will impact the microcontroller, as it will be possible to specify two register indices in one clock cycle for each instruction.

Two-Bus Design

For a dual-bus design, you may need to restructure the entire datapath from scratch to allow it to allow it to carry multiple data at each cycle. This will introduce multiple possible connections for how each input and output ports for the components on the datapath will be connected. For example, we can connect the PC to input from Bus A but output to both Bus A and B; these decisions will need to be made for each of the components. Additionally, this will also bring about more signals which will need to be added to the microcontroller

Pipeline Buffers (Pipeline Approach)

This advice is for you if you decide to pursue the pipeline approach. 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 reuse space in the buffer for different purposes depending on the instruction, but this is not required.

Control Signals (Pipeline Approach)

This advice is for you if you decide to pursue the pipeline approach. 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 (Pipeline Approach)

This advice is for you if you decide to pursue the pipeline approach. 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 (Pipeline Approach)

This advice is for you if you decide to pursue the pipeline approach. Since branch instructions 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, the simplest method is 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 (Pipeline Approach)

This advice is for you if you decide to pursue the pipeline approach. For the branching 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.

7 Report

Alongside the project, you will be required to submit a written report, rough 1-2 pages in length. The report should be presentable, with appropriate formatting.

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

- Explanation of architectural changes that you have implemented.
- Challenges that were faced during implementation.
- Data relating to cycle count when running the program, and any associated metrics that were taught in class such as speedup.
- Potential areas of improvement.

The minimum cycle count achieved by the implementation must be clearly present in the report.

8 Testing

Note: The simulator does not test your processor, it simply runs the assembly so that you can see what the correct output should be. To test your processor, you must load the assembled HEX into CircuitSim.

Use the provided assembler (found in the "assembly" folder) to convert a test program from assembly to hex. For instructions on how to use the assembler and simulator, see README.txt in the "assembly" folder. We recommend using test programs that contain a single instruction since you are bound to have a few bugs at this stage of the project. Once you have built confidence, test your processor with the provided `pow.s` program as a more comprehensive test case.

9 Grading

We will award up to 2.5% Extra Credit (EC).

- Performance Points:
  - 0.5% EC awarded to anyone with a speedup of greater than 1.4 and less than 1.7
  - 1% EC awarded to anyone with a speedup of greater than 1.7 and less than 4
  - 2.5% EC awarded to anyone with a speedup of greater than 4
  - Note: the speedup of your design will be evaluated using the provided `pow.s` program with the same `BASE` of 3 and `EXP` of 8. The base design we provided takes **0x1B9B** (or 7067 in decimal) cycles to halt.
- Report Points: **Not including a report is an automatic 0.**

Note: Due to the late deadline, we cannot accept regrade requests for this project.

10 Deliverables

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

- CircuitSim datapath file (`LC-2222.sim`)
- Microcode file (microcode.xlsx)
- Report file (as a PDF)

If you are missing any of those files, you will get a 0 so make sure that you have uploaded both of them.

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.

11 Appendix A: LC-2222 Instruction Set Architecture

The LC-2222 is a simple, yet capable computer architecture. The LC-2222 combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC-2222 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.

11.1 Registers

The LC-2222 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.

11.2 Instruction Overview

The LC-2222 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-2222 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	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	RA	AT	unused		
HALT	0111	unused				
BLT	1000	SR1	SR2	offset20		
LEA	1001	DR	unused	PCoffset20		
BGT	1010	SR1	SR2	offset20		
OR	1011	DR	SR1	unused	0	SR2
XOR	1011	DR	SR1	unused	1	SR2

11.2.1 Conditional Branching

Branching in the LC-2222 ISA is a bit different than usual. We have a set of branching instructions including BEQ, BLT, 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.

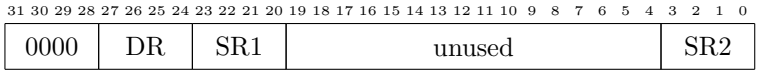
11.3 Detailed Instruction Reference

11.3.1 ADD

Assembler Syntax

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

Encoding



Operation

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

Description

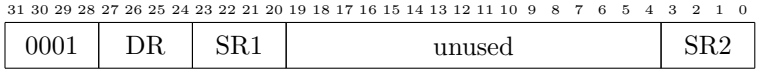
The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

11.3.2 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.

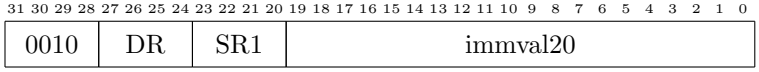
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

11.3.3 ADDI

Assembler Syntax

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

Encoding



Operation

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

Description

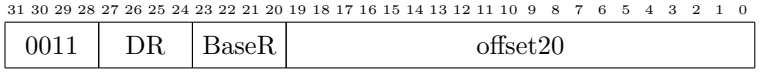
The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

11.3.4 LW

Assembler Syntax

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

Encoding



Operation

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

Description

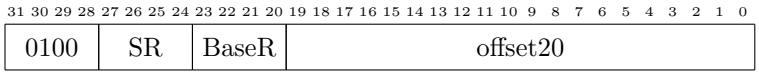
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

11.3.5 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.

11.3.6 BEQ

Assembler Syntax

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

Encoding

Operation

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

Description

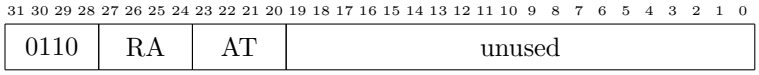
A branch is taken if SR1 is equal to SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

11.3.7 JALR

Assembler Syntax

```
JALR   RA, AT
```

Encoding



Operation

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

Description

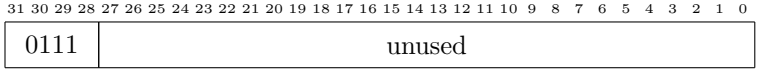
First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

11.3.8 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

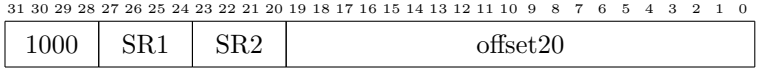
The machine is brought to a halt and executes no further instructions.

11.3.9 BLT

Assembler Syntax

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

Encoding



Operation

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

Description

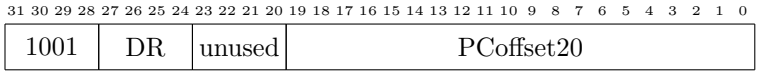
A branch is taken if SR1 is less than SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

11.3.10 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.

11.3.11 BGT

Assembler Syntax

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

Encoding

Operation

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

Description

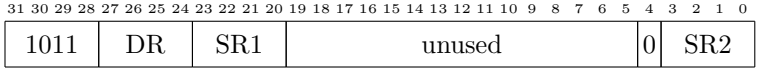
A branch is taken if SR1 is greater than SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

11.3.12 OR

Assembler Syntax

```
OR  DR, SR1, SR2
```

Encoding



Operation

```
DR = SR1 | SR2;
```

Description

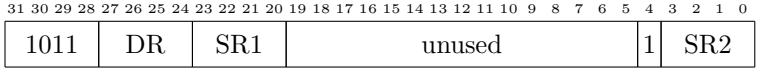
The OR instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register.The bitwise OR of operands SR1 and SR2 is calculated, and stored into DR.

11.3.13 XOR

Assembler Syntax

```
XOR  DR, SR1, SR2
```

Encoding



Operation

```
DR = SR1 ^ SR2;
```

Description

The XOR instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register.The bitwise XOR of operands SR1 and SR2 is calculated, and stored into DR.