Computer Systems and Networks

Prof. Forsyth

Project 1 - LC-4000 Datapath

Due: **September 15th 2024** @ 11:59 PM EST

1 Requirements

- Download the proper version of CircuitSim. The proper version is version 1.9.1 or later. You download it from the CircuitSim website (https://ra4king.github.io/CircuitSim/). 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-4000 assembler is written in Python. If you do not have Python 3 or newer installed on your system, you must install it before you continue. You may also use the Docker container for this.

2 Project Overview and Description

Project 1 is designed to give you a good feel for exactly how a processor works. In Phase 1, you will design a datapath in CircuitSim to implement a supplied instruction set architecture. You will use the datapath as a tool to determine the control signals needed to execute each instruction. In Phases 2 and 3, you are required to build a simple finite state machine (the "control unit") to control your computer and actually run programs on it.

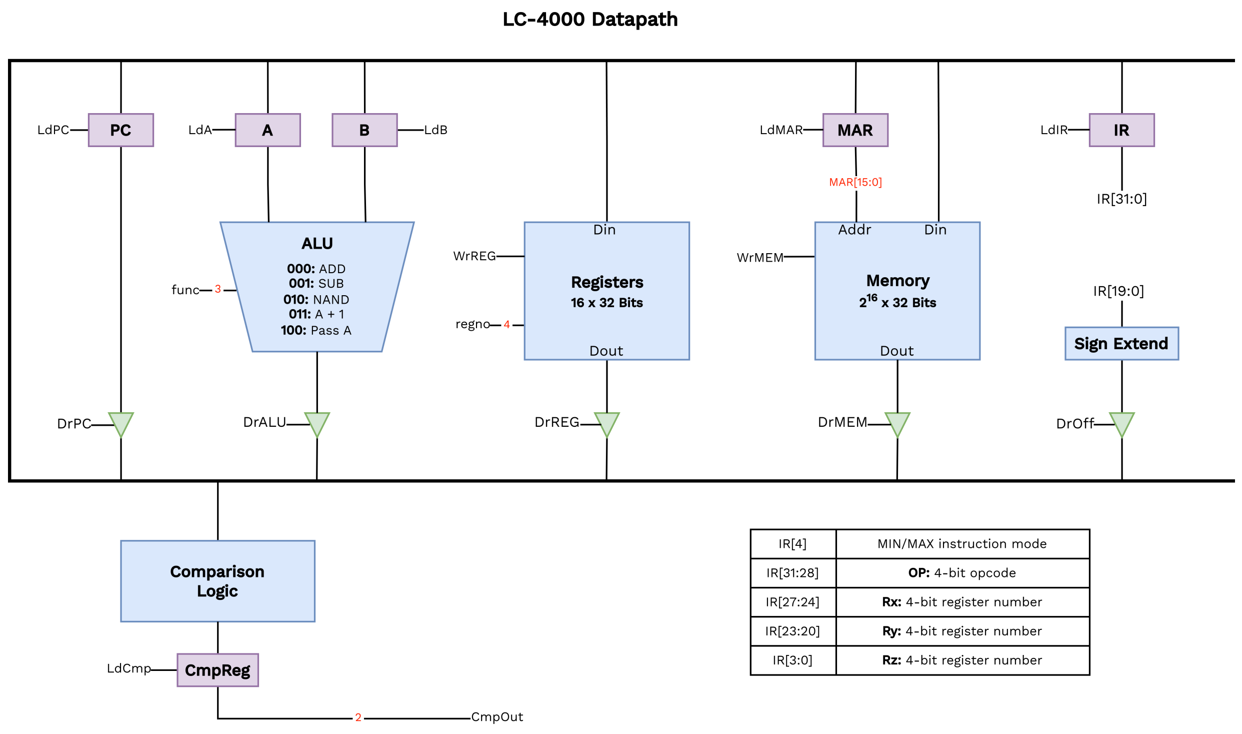
Note: You will need to have a working knowledge of CircuitSim. Make sure that you know how to make basic circuits as well as subcircuits before proceeding. You are free to use any of CircuitSim's provided components i.e. registers. Refer to the lab 1 slides for some key tips. The TAs are always here if you need further assistance.

3 Grading

Grading will be done via submission to the project 1 autograder on gradescope. Each operation and certain parts of the datapath you implement will be worth an equivalent number of points. There will also be points for fully completing the project via completing a successful run of **pow.s**, which is provided to you. TAs will review your submission and add/remove points due to circumstances not reflected by the autograder.

Note: The autograder will have a limit to the daily number of submissions until one day before the due date, as it is not meant to be a debugger. See **Section 7** for more details about gradescope and how to debug locally.

4 Phase 1 - Implement the Datapath



IR[4]	MIN/MAX instruction mode
IR[31:28]	**OP:** 4-bit opcode
IR[27:24]	**Rx:** 4-bit register number
IR[23:20]	**Ry:** 4-bit register number
IR[3:0]	**Rz:** 4-bit register number

Figure 1: Datapath for the LC-4000 Processor

In this phase of the project, you must learn the Instruction Set Architecture (ISA) for the processor we will be implementing. Afterwards, we will implement a complete LC-4000 datapath in CircuitSim using what you have just learned.

You must do the following:

1. Learn and understand the LC-4000 ISA. The ISA is fully specified and defined in Appendix A: LC-4000 Instruction Set Architecture. **Do not move on until you have fully read and understood the ISA specification**. *Every single detail* will be relevant to implementing your datapath in the next step.
2. Using CircuitSim, implement the LC-4000 datapath. To do this, you will need to use the details of the LC-4000 datapath defined in Appendix A: LC-4000 Instruction Set Architecture. You should model your datapath on Figure 1.
3. Put your name on your CircuitSim data path in a comment box so we know it is your work.

4.1 Hints

4.1.1 Subcircuits

CircuitSim enables you to break create reusable components in the form of subcircuits. **We highly recommend that you break parts of your design up into subcircuits.** At a minimum, you will want to implement your ALU in a subcircuit, but we highly recommend using more. The control unit you implement in Phase 2 is another prime candidate for a subcircuit.

4.1.2 Debugging

As you build the datapath, you should consider adding functionality to allow you to operate the whole datapath by hand. This will make testing individual operations quite simple. We suggest your datapath include devices (such as probes) to view the values at different parts of your datapath as it runs a program. Feel free to add any additional hardware that will help you understand what is going on. Using breakpoints to pause the program at a specified point can be very helpful.

4.1.3 Memory Addresses

Because of CircuitSim limitations, the RAM module is limited to no more than 16 address bits. Therefore, per our ISA, any 32-bit values used as memory addresses will be truncated to 16 bits (with the 16 most significant bits disregarded). If you use the RAM subcircuit we provide, this truncation has already been handled, and you can simply attach the 32-bit value from the MAR (Memory Address Register) to our custom RAM circuit. Otherwise, you will need to truncate the most significant bits of the the address value from the MAR before feeding it into the RAM.

4.1.4 Comparison Logic

The "comparison logic" box in Figure 1 is responsible for performing the comparison logic associated with the BEQ, BLT, MIN and MAX instructions. The comparison logic should read the current value on the bus. When executing BEQ, you should compute $A - B$ using the ALU. While this result of the ALU is being driven on the bus, you should wire the value of $A - B$ into a comparator, and use the resulting outputs as necessary to handle all cases of the comparison logic across all instructions.

Your comparison logic should be purely combinational (not sequential). Feel free to use any CircuitSim components you wish to aid in your implementation.

4.1.5 Register File

You must implement your register file. That is to say, you cannot use CircuitSim's built-in RAM to create the register file. Consider what logic components you may want to use to implement addressing functionality (multiplexers, demultiplexers, decoders, etc). Your zero register must be implemented such that writes to it are ineffective, i.e., attempting to write a non-zero value to the zero register will do nothing. There are multiple ways to do this. **Do not forget to do this or you will lose points!**

4.1.6 Register Select

From the lecture and the textbook, you should be familiar with the "register select" (RegSel) multiplexer. The mux is responsible for feeding the register number from the correct field in the instruction into the register file. See Table 4 for a list of inputs your mux should have.
Note: There is one unused combination of bits. This is intentional and will be used in the next project.

5 Phase 2 - Implement the Microcontrol Unit

In this phase of the project, you will use CircuitSim to implement the microcontrol unit for the LC-4000 processor. This component is referred to as the "Control Logic" in the images and schematics. The microcontroller will contain all of the signal lines to the various parts of the datapath.

You must do the following:

1. Read and understand the microcontroller logic:
    - Please refer to Appendix B: Microcontrol Unit for details.
    - **Note:** You will be required to generate the control signals for each state of the processor in the next phase, so make sure you understand the connections between the datapath and the microcontrol unit before moving on.

2. Implement the Microcontrol Unit using CircuitSim. The appendix contains all of the necessary information. Take note that the input and output signals on the schematics directly match the signals marked in the LC-4000 datapath schematic (see Figure 1).

6 Phase 3 - Microcode and Testing

In this final stage of the project, you will write the microcode control program that will be loaded into the microcontrol unit you implemented in Phase 2. Then, you will hook up the control unit you built in Phase 2 of the project to the datapath you implemented in Phase 1. Finally, you will test your completed computer using a simple test program and ensure that it properly executes.

You must do the following:

1. Open and fill out microcode.xlsx file we've provided you (note: the formulas in the provided file will **only** work with Excel). You will need to mark which control signal is high for each of the states.
(High = 1; Low = 0)
2. After you have completed all the microstates, convert the binary strings you just computed into hex and move them into the main ROM. You can just copy and paste the hex column (highlighted yellow) from the spreadsheet directly into the ROM component in Circuitsim. Do the same for the sequencer and condition ROMs.
3. Connect the completed control unit to the datapath you implemented in Phase 1. Using Figures 1 and 2, connect the control signals to their appropriate spots.
4. Finally, it is time to test your completed computer. 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 section 11.

   We highly 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.

Important: When copying and pasting hex into CircuitSim sub-circuits, be sure to *double-click* into the sub-circuit from the Datapath circuit page, and then paste the values into any ROMs, otherwise, the hex will not persist once you exit the sub-circuit.

Warning: The Excel sheet has multiple columns filled with formulas. Be sure not to change the equations, or the hex will not be computed accurately. Be especially careful about this when copying and pasting within the sheet.

7 Autograder

The gradescope autograder will execute a series of tests that will help determine your grade on this project:

1. A test that verifies you implemented the zero register correctly.
2. A set of unit tests that verify you implemented the instructions correctly.
3. A comprehensive test that runs **pow.s**.

Feel free to use it as a tool to assist you debug your circuit, but you should not rely on it entirely. The autograder message may not pinpoint your problem, and local debugging is the only way to find the issue. You must adhere to the following rules, or you will be penalized and the gradescope autograder will not run correctly:

- Don't rename the main subcircuit "Datapath"
- Name your PC register as "PC"

- Name your IR register as “IR”
- Name your state register as “State”
- Name your 3 ROMs as ”MAIN”, ”SEQ”, and ”CC” respectively
- Reference registers in the register file in lower case with prefixed ‘$’ sign ($zero, $at, $v0...), according to Appendix A: LC-4000 Instruction Set Architecture.
- Use only one clock globally, which means you should not use clock components in any subcircuits besides the main datapath. Instead, you should use an input pin and connect the clock signal to that subcircuit in the main datapath.
- Use only one RAM as memory
- In microcode, use the first row as your first state of FETCH macrostate.
- Don’t change the layout or formulas of the microcode Excel sheet

If the autograder fails you, please first double-check if you meet all the rules above. If the autograder points you to a line of code, try to load the assembled hex of the **pow.s** program into your RAM, clock it until PC turns to that line number, and check state by state to see if any component goes wrong when executing that instruction. Sometimes, you may not reproduce the error when you reach that instruction for the first time. This is because there are some loop structures and subroutine calls in the test program, and you will execute some instructions multiple times. The error occurs when you reach that instruction again and the condition changes (e.g. a conditional branch instruction). When you get an error message, please first try to reproduce it locally and think about what you observe.

Note: The autograder is not meant to be used as a primary debugging tool. For that reason, submissions are limited to **10 per day**. Please utilize local debugging to find and fix any problems, and then submit to the autograder when you believe everything is working.

If you come to office hours without having debugged the issue locally, you will not receive assistance.

Disclaimer: We have heavily modified the auto grader this semester, and we might have introduced a couple of bugs. If you can demonstrate that your circuit works via local debugging, and it fails the autograder, please let us know on EdStem.

In addition, the autograder will **not work** if `HALT` is not implemented correctly.

8 Deliverables

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

- CircuitSim Datapath file (`LC-4000.sim`)
- Microcode file (`microcode.xlsx`)

If you are missing one or both 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.

This project will be demoed. To receive full credit, you must sign up for a demo slot and complete the demo. Demo information will be released on Canvas near the due date.

9 Appendix A: LC-4000 Instruction Set Architecture

The LC-4000 is a simple, yet capable computer architecture. The LC-4000 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-4000 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-4000 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-4000 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-4000 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				
BLT	1000	SR1	SR2	offset20		
LEA	1001	DR	unused	PCoffset20		
MIN	1010	DR	SR1	unused	0	SR2
MAX	1010	DR	SR1	unused	1	SR2

9.2.1 Conditional Branching

Branching in the LC-4000 ISA is slightly different than usual. We have a set of branching instructions including both BEQ and BLT, 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 BLT instruction, if SR1 $<$ SR2), then we will branch to the target destination of incrementedPC + offset20.

For MIN, if SR1 $<$ SR2, we branch to the series of microstates that stores the value in SR1 to DR. Otherwise, we branch to a series of microstates that stores the value in SR2 to DR. For MAX, we do the opposite.

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.

Note: The autograder will not run if this instruction is not implemented correctly.

9.3.9 BLT

Assembler Syntax

```
BLT  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.

9.3.11 MIN

Assembler Syntax

```
MIN    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	0	SR2

Operation

```
if (SR1 < SR2) {
    DR = SR1
} else {
    DR = SR2
}
```

Description

The minimum is computed between the values in both source registers. It then stores the minimum value in the register DR.

Note: MIN and MAX have the same opcode. Bit 4 being 0 indicates the MIN instruction.
The control flow for MIN must go through the CC ROM. Not doing so will cause you to lose points!

9.3.12 MAX

Assembler Syntax

```
MAX    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	1	SR2

Operation

```
if (SR1 > SR2) {
    DR = SR1
} else {
    DR = SR2
}
```

Description

The maximum is computed between the values in both source registers. It then stores the maximum value in the register DR.

Note: MIN and MAX have the same opcode. Bit 4 being 1 indicates the MAX instruction.
The control flow for MAX must go through the CC ROM. Not doing so will cause you to lose points!

10 Appendix B: Microcontrol Unit

You will make a microcontrol unit which will drive all of the control signals to various items on the datapath. This Finite State Machine (FSM) can be constructed in a variety of ways. You could implement it with combinational logic and flip flops, or you could hard-wire signals using a single ROM. The single ROM solution will waste a tremendous amount of space since most of the microstates do not depend on the opcode or the conditional test to determine which signals to assert. For example, since the condition line is an input for the address, every microstate would have to have an address for condition = 0 as well as condition = 1, even though this only matters for one particular microstate.

To solve this problem, we will use a three ROM microcontroller. In this arrangement, we will have three ROMs:

- the main ROM, which outputs the control signals,
- the sequencer ROM, which helps to determine which microstate to go at the end of the FETCH state,
- and the condition ROM, which helps determine whether or not to branch during the BEQ and BLT instructions. It also helps determine which source value to store during the MIN/MAX instructions.

Examine the following:

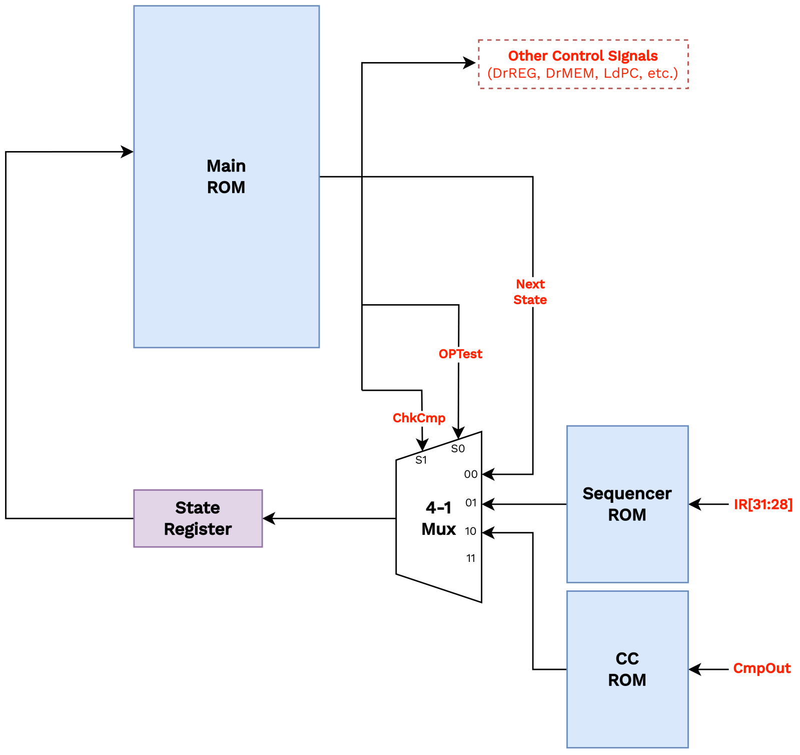


Figure 2: Three ROM Microcontrol Unit

As you can see, there are three different locations that the next state can come from: part of the output from the previous state (main ROM), the sequencer ROM, and the condition ROM. The mux controls which of these sources gets through to the state register. If the previous state's "next state" field determines where to go, neither the OPTest nor ChkCmp signals will be asserted. If the opcode from the IR determines the next state (such as at the end of the FETCH state), the OPTest signal will be asserted. If the comparison circuitry determines the next state (such as in the BEQ instructions), the ChkCmp signal will be asserted.

Note that these two signals should never be asserted at the same time since nothing is input into the "11" pin on the MUX.

The sequencer ROM should have one address per instruction, and the condition ROM should have one address for condition true and one for condition false.

Before getting down to specifics you need to determine the control scheme for the datapath. To do this examine each instruction, one by one, and construct a finite state bubble diagram showing exactly what control signals will be set in each state. Also determine what are the conditions necessary to pass from one state to the next. You can experiment by manually controlling your control signals on the bus you've created in **Phase 1 - Implement the Datapath** to make sure that your logic is sound.

Once the finite state bubble diagram is produced, the next step is to encode the contents of the Control Unit ROM. Then you must design and build (in CircuitSim) the Control Unit circuit which will contain the three ROMs, a MUX, and a state register. Your design will be better if it allows you to single step and ensure that it is working properly. Finally, you will load the Control Unit's ROMs with the hexadecimal generated by your filled out microcode.xlsx.

Note that the input address to the ROM uses bit 0 for the lowest bit of the current state and 5 for the highest bit for the current state.

Table 3: ROM Output Signals

Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose
0	NextState[0]	6	DrREG	12	LdIR	18	WrMEM	24	OPTest
1	NextState[1]	7	DrMEM	13	LdMAR	19	RegSelLo	25	ChkCmp
2	NextState[2]	8	DrALU	14	LdA	20	RegSelHi		
3	NextState[3]	9	DrPC	15	LdB	21	ALU0		
4	NextState[4]	10	DrOFF	16	LdCmp	22	ALU1		
5	NextState[5]	11	LdPC	17	WrREG	23	ALU2		

Table 4: Register Selection Map

RegSelHi	RegSelLo	Register
0	0	RX (IR[27:24])
0	1	RY (IR[23:20])
1	0	RZ (IR[3:0])
1	1	unused

Table 5: ALU Function Map

ALU2	ALU1	ALU0	Function
0	0	0	ADD
0	0	1	SUB
0	1	0	NAND
0	1	1	A + 1
1	0	0	Pass A

Note: The ordering of RegSelHi/RegSelLo and ALU2/ALU1/ALU0 are in the **opposite order** in the microcode excel spreadsheet! Please make sure you are enabling the correct signals in your microcode!

11 Appendix C: Assembler and Simulator

11.1 LC-4000 Assembler and Simulator

To aid in testing your processor, we have provided an assembler and simulator for the LC-4000 architecture. The assembler supports converting text **.s** files into either binary (for the simulator) or hexadecimal (for pasting into CircuitSim) formats.

The assembler and simulator run on any version of Python 3+. An instruction set architecture definition file is required along with the assembler. The LC-4000 assembler definition is included. If you have issues with Python, run the commands in the Docker container as it has all the dependencies for using these files.

- **assembler.py**: the main assembler program
- **lc4000.py**: the LC-4000 assembler definition
- **lc4000-sim.py**: the LC-4000 simulator program

Note: At any point, if using the `python3` command does not work for you, try replacing it with the `python` command.

We highly recommend reading through these files, and understanding the tools available to help you debug and complete this project.

11.2 Using the Assembler

Example usage to assemble **pow.s**:

```
python3 assembler.py -i lc4000 pow.s --sym pow.sym
python3 assembler.py -i lc4000 pow.s --bin pow.bin
```

This should create **pow.sym** and **pow.bin** files. These files can then be loaded into the simulator.

To use assembled code in CircuitSim, we need it in hexadecimal. This command will create a hex file from a **.s** file:

```
python3 assembler.py -i lc4000 pow.s --hex pow.hex
```

You can then open the resulting **pow.hex** file in your favorite text editor. In CircuitSim, double-click into the RAM subcircuit from your datapath. Then right-click on your RAM, select "Edit Contents", and copy-and-paste the contents of **pow.hex** into this window. Make sure to click into the subcircuit from your datapath, **not the tabs at the top of the window**, otherwise the changes will not persist once you leave the subcircuit.

Do not use the Open or Save buttons in CircuitSim, as it will not recognize the format.

11.3 Using the Simulator

You can use the simulator to explore how your processor is expected to behave, step through code instruction-by-instruction, and examine the values of registers and memory at each stage of the program.

Example usage to simulate **pow.bin** (generated by the assembler):

```
python3 lc4000-sim.py pow.bin
```

Then type “help” at the prompt for a list of available commands:

```
(sim) help

Welcome to the simulator text interface. The available commands are:

r[un] or c[ontinue]             resume execution until next breakpoint
s[tep]                          execute one instruction
b[reak] <addr or label>         view and set breakpoints
                                ex) 'break'
                                ex) 'break 0x3'
                                ex) 'break main'
print <lowaddr>-<highaddr>      print the values in memory between <lowaddr> and <highaddr>
                                ex) 'print 0x20-0x30'
q[uit]                          exit the simulator
```

This can help examine register values at different points of a program. You can also do this by uploading the hex into CircuitSim and running the program.

11.4 Assembler Pseudo-Ops

In addition to the syntax described in the LC-4000 ISA reference, the assembler supports the following pseudo-instructions:

- `.fill`: fills a word of memory with the literal value provided
- `.word`: an alias for `.fill`

For example: `mynumber:  .fill 0x500` places `0x500` at the memory location labeled `mynumber`.