CS 2200 Systems and Networking

Prof. Ramachandran & Prof. Daglis

Project 1 - LC 5500 Datapath

Due: **February 10th 2023**

1 Requirements

- Download the proper version of CircuitSim. The proper version is version 1.9.0. Please follow the installation guide under Modules on Canvas.
- 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-5500 assembler is written in Python. If you do not have Python3 installed on your system, you will need to install it before you continue.

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. **Before you start, please read through the whole document, especially the Autograder section. It will be hard to fix some problems if you want to use the autograder to help you debug later on.**

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. The TAs are always here if you need help.

3 Phase 1 - Implement the Datapath

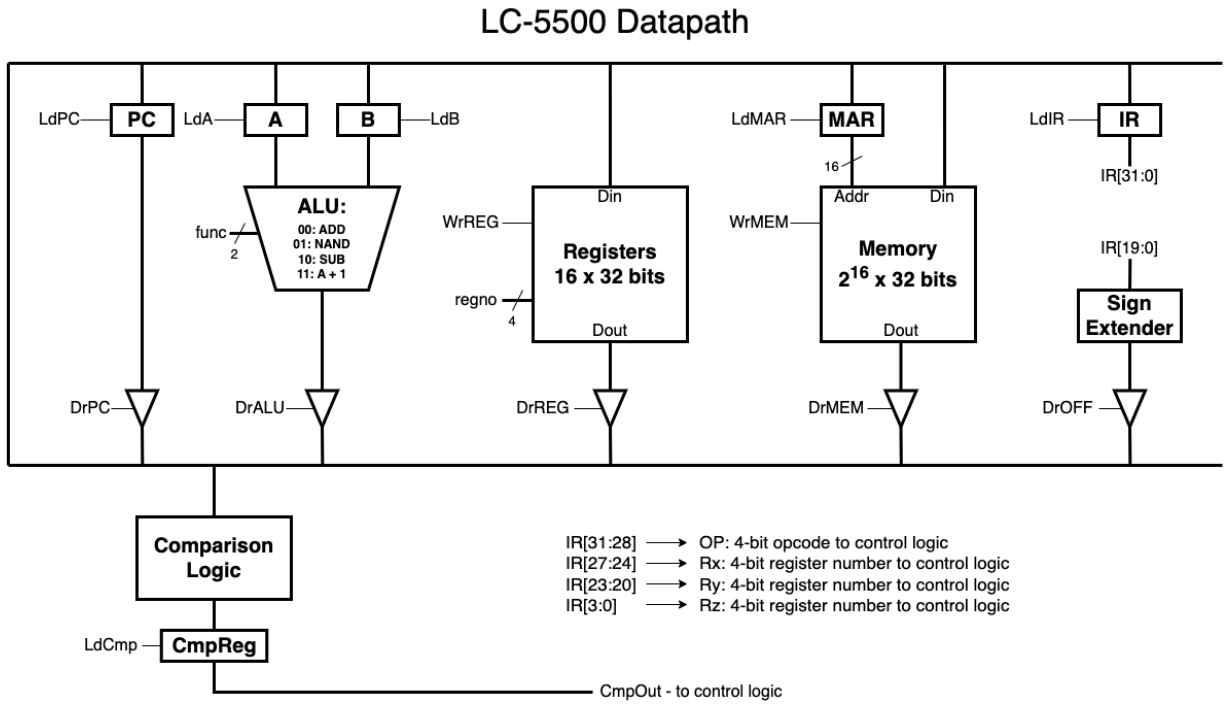


Figure 1: Datapath for the LC-5500 Processor

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-5500 datapath in CircuitSim using what you have just learned.

You must do the following:

1. Learn and understand the LC-5500 ISA. The ISA is fully specified and defined in Appendix A: LC-5500 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-5500 datapath. To do this, you will need to use the details of the LC-5500 datapath defined in Appendix A: LC-5500 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.

3.1 Hints

3.1.1 Subcircuits

CircuitSim enables you to 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. The control unit you implement in Phase 2 is another prime candidate for a subcircuit. Though tempting, placing key processor registers in a subcircuit such as the PC or IR will make

it more difficult to debug as you will not be able to see the value, so it is recommended that these are not placed in subcircuits.

3.1.2 Debugging Hardware

As you build the datapath, you should consider adding functionality that will allow you to operate the whole datapath by hand. This will make testing individual operations quite simple. We suggest that your datapath includes devices that will allow you to put arbitrary values on the bus and to view the current value of the bus. Feel free to add any additional hardware that will help you understand what is going on.

3.1.3 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). We would like you to implement this (i.e. truncate the most significant bits) before feeding the address value from the MAR (Memory Address Register) to the RAM.

3.1.4 Comparison Logic

For this project, you are only required to implement BLT, since all other branch can be derived from different uses of the branch less-than instruction.

When constructing your comparison logic (see 1), you must compare the SR1 and SR2 values and output the result to a "Comparison Register" (or CmpReg) for later use. This register's value corresponds to 1 if the branch should be taken, and 0 if it should not – it is up to you to determine how exactly you make this decision. We recommend using CircuitSim's built-in comparator circuit and comparing the difference between the two registers against 0.

3.1.5 Register File

You must implement your own 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. Please also label your registers with their proper names (found in Appendix A: LC-5500 Instruction Set Architecture – **$ included!**) using CircuitSim's built-in label feature. **Do not forget to do this or you will lose points!**

3.1.6 Register Select

From 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.

4 Phase 2 - Implement the Microcontrol Unit

In this phase of the project, you will use CircuitSim to implement the microcontrol unit for the LC-5500 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-5500 datapath schematic (see Figure 1).

5 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 (that is 1) for each of the states.
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 README.txt in the "assembly" folder. **Note: The simulator does not test your project, it simply provides a model. To test your design, you must load the assembled HEX into the RAM. !!Important!! If you use the RAM subcircuit that we provide, always double-click into the subcircuit component instead of clicking the tabs in the tab menu. Otherwise, the RAM you see might not be connected to the main datapath.** 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.

Tips:

1. You can use the "Clock Enabled" feature to start the clock, and you can also determine the clock frequency. Both are under "Simulation" at the top.
2. There is a component in Debugging called Breakpoint. You can set a value and connect any wire to it. Then, once you start the clock, it will stop when the wire turns to the value you just set. You can connect it to your PC to stop at any instruction.

6 Autograder

The autograder for Project 1 mainly serves as a debugging tool, so there is no point assigned to it, and your final grade will not be determined by whether you pass the autograder or not. The autograder will execute the test program **pow.s** using your datapath and tell you which instruction goes wrong, but it won't evaluate anything else. Feel free to use it as a tool to help you debug your circuit, but you won't be able to rely on it entirely. As the autograder only tells you which instruction leads to an error, you must still figure out which part of your datapath is not functioning as expected. However, it should be easier to reproduce the

error knowing the address of the failing machine instruction! If you want to use the autograder, you must follow a few rules:

- 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 regfile in lower case with prefixed '$' sign ($zero, $at, $v0...), according to Appendix A: LC-5500 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 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. If you still don't know how to approach it, come to office hours or make a private post with a detailed explanation of your attempts of debugging, instead of just a post with the error message and a screenshot of your datapath. TAs won't be able to help you solve the problem with that little amount of information.

7 Deliverables

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

- CircuitSim datapath file (LC-5500.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. In order to receive full credit, you must sign up for a demo slot and complete the demo. We will announce when demo times are released.

8 Appendix A: LC-5500 Instruction Set Architecture

The LC-5500 is a simple, yet capable computer architecture. The LC-5500 combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC-5500 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.

8.1 Registers

The LC-5500 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 stack; it keeps track of the top of the activation record for a subroutine.
9. **Register 14** is the anchor point of the activation frame. It is used to point to the first address on the activation record for the currently executing process.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing.

8.2 Instruction Overview

The LC-5500 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-5500 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	SR	immval20
LW	0011	DR	BaseR	offset20
SW	0100	SR	BaseR	offset20
BR	0101	unused		offset20
JALR	0110	RA	AT	unused
HALT	0111	unused		
BLT	1000	SR1	SR2	PCoffset20
LEA	1001	DR	unused	PCoffset20
PUSH	1010	SR	$SP	offset20 (-1)
POP	1011	DR	$SP	offset20 (+1)

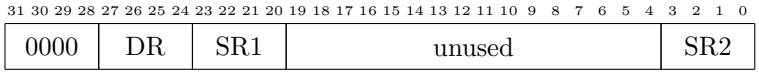
8.3 Detailed Instruction Reference

8.3.1 ADD

Assembler Syntax

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

Encoding



Operation

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

Description

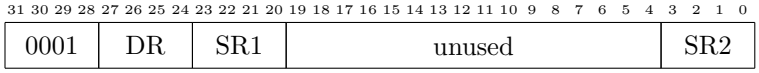
The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

8.3.2 NAND

Assembler Syntax

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

Encoding



Operation

```
DR = ~(SR1 & SR2);
```

Description

The NAND instruction performs a logical NAND (AND NOT) on the source operands obtained from SR1 and SR2. The result is stored in DR.

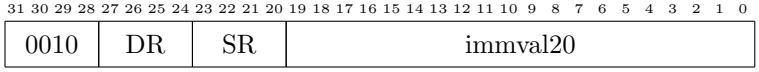
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

8.3.3 ADDI

Assembler Syntax

```
ADDI   DR, SR, immval20
```

Encoding



Operation

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

Description

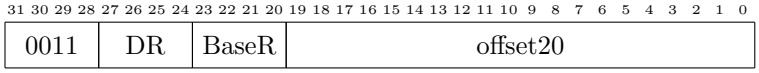
The ADDI instruction obtains the first source operand from the SR register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

8.3.4 LW

Assembler Syntax

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

Encoding



Operation

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

Description

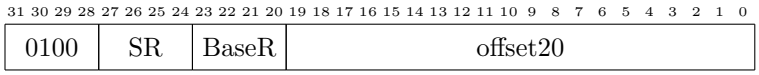
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

8.3.5 SW

Assembler Syntax

```
SW    SR, offset20(BaseR)
```

Encoding



Operation

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

Description

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

8.3.6 BR

Assembler Syntax

```
BR  offset20
```

Encoding

Operation

```
PC = incrementedPC + offset20
```

Description

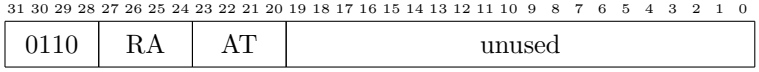
A branch is unconditionally taken. The PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

8.3.7 JALR

Assembler Syntax

```
JALR   RA, AT
```

Encoding



Operation

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

Description

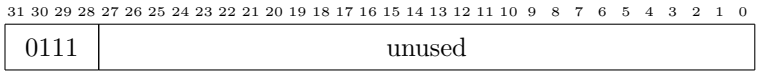
First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

8.3.8 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

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

8.3.9 BLT

Assembler Syntax

```
BLT  SR1, SR2
```

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	PCoffset20

Operation

```
if (SR1 < SR2) {
    PC = incrementedPC + SEXT(PCoffset20)
}
```

Description

This is a conditional branch that will be taken only if the value of the register SR1 is less that then value of the SR2 register. The PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

8.3.10 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 into register DR.

8.3.11 PUSH

Assembler Syntax

```
PUSH    SR
```

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	SR	$SP	offset20 (-1)

Operation

```
X = SR
$SP = $SP - 1
MEM[$SP] = X
```

Description

Decrements the stack pointer and stores the value of the source register SR at the memory location of the stack pointer. Note: the operation of the decrement must happen before the store in order to ensure that the stack pointer always points to the top of the stack, but be sure to take special care when pushing the stack pointer itself.

8.3.12 POP

Assembler Syntax

```
POP     DR
```

Encoding

Operation

```
X = MEM[$SP]
$SP = $SP + 1
DR = X
```

Description

Loads the value from memory pointed to by the stack pointer into DR and increments the value of the stack pointer. Note: the operation of the increment must happen after the load in order to ensure that the stack pointer always points to the top of the stack.

9 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 skip during the SKP instructions.

Examine the following:

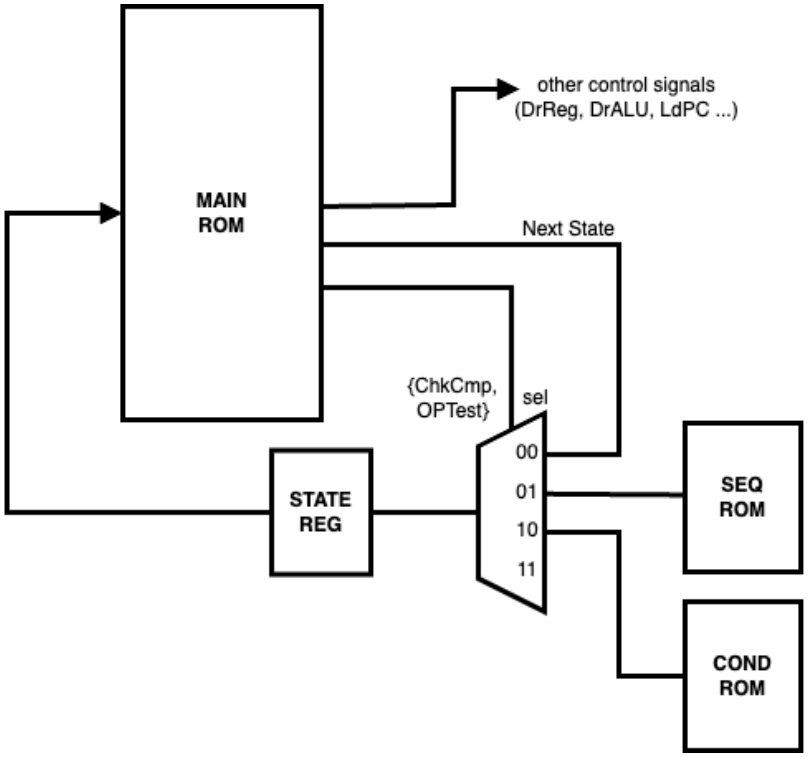


Figure 2: Three ROM Microcontrol Unit

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 SKP instruction), 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) 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	ChkCmp
1	NextState[1]	7	DrMEM	13	LdMAR	19	RegSel[0]		
2	NextState[2]	8	DrALU	14	LdA	20	RegSel[1]		
3	NextState[3]	9	DrPC	15	LdB	21	Func[0]		
4	NextState[4]	10	DrOFF	16	LdCmp	22	Func[1]		
5	NextState[5]	11	LdPC	17	WrREG	23	OPTest		

Table 4: Register Selection Map

RegSel[1]	RegSel[0]	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

Func[1]	Func[0]	Function
0	0	ADD
0	1	NAND
1	0	SUB
1	1	A + 1