1 Requirements

- Download the proper version of CircuitSim. The proper version is version 1.8.2. A copy of CircuitSim is available under Files on Canvas. 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-2199 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.

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. The TAs are always here if you need help.

3 Phase 1 - Implement the Datapath

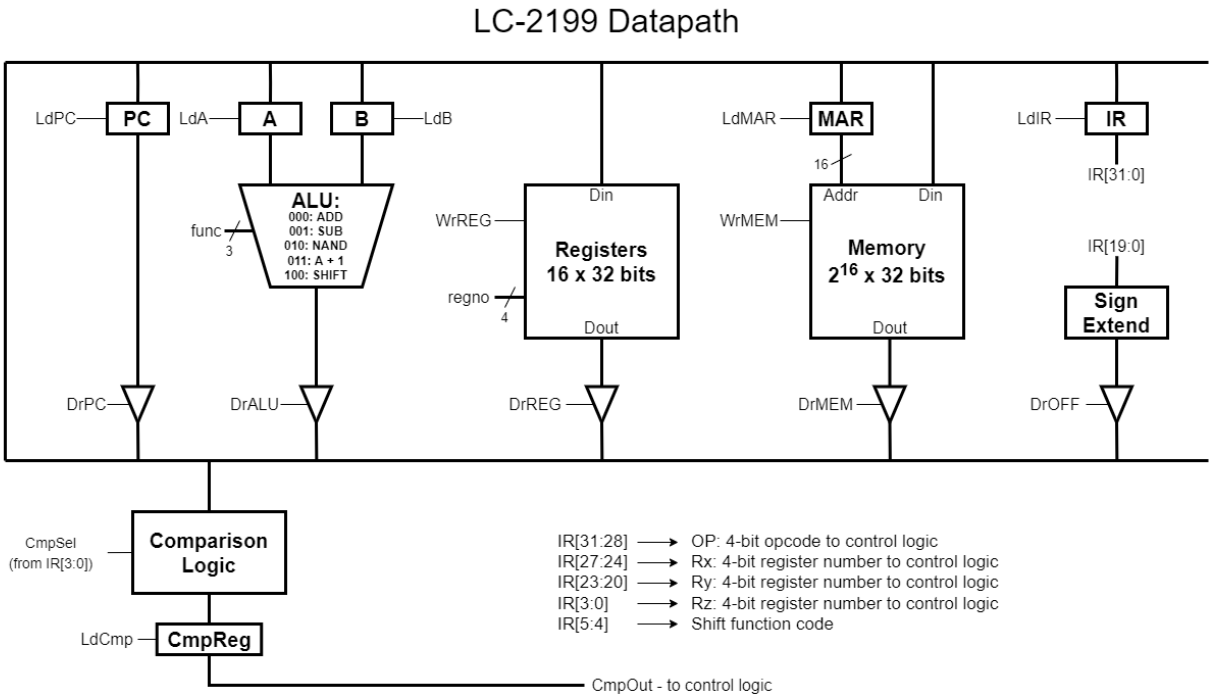


Figure 1: Datapath for the LC-2199 Processor

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-2199 datapath in CircuitSim using what you have just learned.

You must do the following:

1. Learn and understand the LC-2199 ISA. The ISA is fully specified and defined in Appendix A: LC-2199 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-2199 datapath. To do this, you will need to use the details of the LC-2199 datapath defined in Appendix A: LC-2199 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 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. The control unit you implement in Phase 2 is another prime candidate for a subcircuit.

3.1.2 Debugging

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 your datapath include 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 Shifting

The LC-2199 implements 4 kinds of shifts: logical left, logical right, arithmetic right, and right rotation. CircuitSim already contains a sub-circuit that handles shifting; make sure to take a look at it. All four of these instructions share an opcode, with the difference lying in bits [5:4]. We expect these shift functions to be part of your ALU design, so you will need to control the ALU function select to implement the correct shift.

You have options for implementation. One possibility is to use the ALU2 microcode bit as an indicator that a shifting operation is desired, and send {ALU2, IR[5], IR[4]} to the 3-bit ALU function select (instead of sending {ALU2, ALU1, ALU0} as you'll do with AND and NAND). The other possibility is to connect a 2-input multiplexer to the ALU function select letting the "zero" input be {0, ALU1, ALU0} and the "one" input be {1, IR[5], IR[4]}. Then connect ALU2 to the multiplexer control. These are not the only methods of correct implementation – any setup that results in ISA-adherent hardware will be accepted. These complexities are necessary to allow these functions to share an opcode while performing different operations. For more details, see the ALU function map in 8.

3.1.5 Comparison Logic

For this project, you are only required to implement SKPEQ and SKPGT. These two instructions share an opcode, and are denoted by differences in the IR[3:0] bits. We denote IR[3:0] as CmpSel in 1, though that is not its only purpose.

When constructing your comparison logic (see 1), you must compare the SR1 and SR2 values according to the instruction being handled, and output the result to a "Comparison Register" (or CmpReg) for later use. This register's value corresponds to 1 if a skip should be taken, and 0 if it should not – it is up to the student 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.

As an example, if we want to skip when SR1 > SR2, we would set IR[3:0] = 0000 when assembling the instruction. We would then utilize this subset of bits to select which comparison in the comparison logic we actually care about (keep in mind that your comparison logic should be combinational, and so both comparisons will occur simultaneously). Note that, although IR[3:0] is used to denote this difference, not all bits may be relevant.

For a discussion on implementing comparison logic for all combinations of arithmetic comparisons (>, <, ≤, etc.), see 9.

3.1.6 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-2199 Instruction Set Architecture – **$ included!**) using CircuitSim's built-in label feature. **Do not forget to do this or you will lose points!**

3.1.7 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-2199 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-2199 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 CircuitSim.** 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.

6 Deliverables

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

- CircuitSim datapath file (LC-2199.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.

7 Appendix A: LC-2199 Instruction Set Architecture

The LC-2199 is a simple, yet capable computer architecture. The LC-2199 combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC-2199 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.

7.1 Registers

The LC-2199 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.

7.2 Instruction Overview

The LC-2199 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-2199 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	
BR	0101	unused		offset20	
JALR	0110	RA	AT	unused	
HALT	0111	unused			
SKPEQ	1000	SR1	SR2	unused	0000
SKPGT	1000	SR1	SR2	unused	0001
LEA	1001	DR	unused	PCoffset20	
SLL	1010	DR	SR1	unused 00	SR2
SRL	1010	DR	SR1	unused 01	SR2
SRA	1010	DR	SR1	unused 10	SR2
ROR	1010	DR	SR1	unused 11	SR2

7.2.1 Conditional Branching

Branching in the LC-2199 ISA is a bit different than usual. We have the series of isntructions known as the Skip Instructions, or SKP. These instructions use the comparison operators, comparing the values of two source registers. If the comparisons are true (for example, with the SKPGT instruction, if SR1 > SR2), then we skip over the next line of code – we increment PC by 1 (remember that at the time of execution, PC has already been incremented by 1, so this is an additional increment).

Note: These SKP instructions all have the same opcode and use IR[2:0] to determine the comparison type. Recall the following. Bit 0 is used to check equality between SR1 and SR2. Bit 1 is used to check if SR1 is less than SR2. Bit 2 is used to negate the condition. We have given you some examples in section 3.1.4, so try and work out the rest on your own.

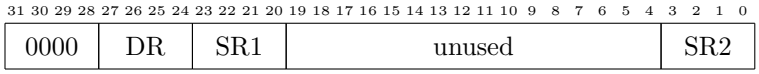
7.3 Detailed Instruction Reference

7.3.1 ADD

Assembler Syntax

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

Encoding



Operation

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

Description

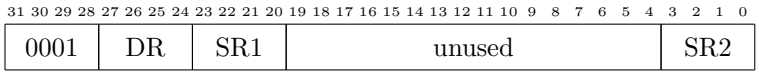
The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

7.3.2 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.

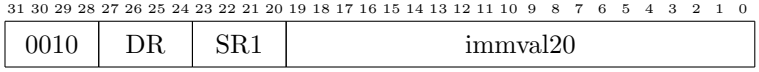
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

7.3.3 ADDI

Assembler Syntax

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

Encoding



Operation

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

Description

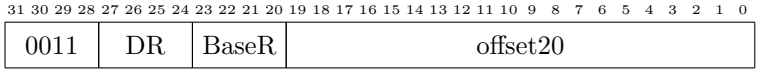
The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

7.3.4 LW

Assembler Syntax

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

Encoding



Operation

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

Description

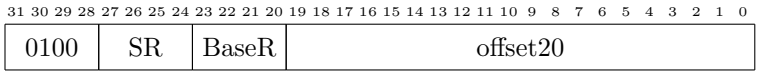
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

7.3.5 SW

Assembler Syntax

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

Encoding



Operation

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

Description

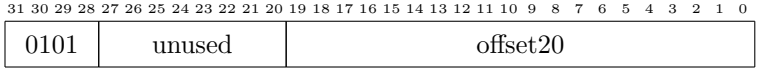
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word obtained from register SR is then stored at this address.

7.3.6 BR

Assembler Syntax

```
BR  offset20
```

Encoding



Operation

```
PC = incrementedPC + offset20
```

Description

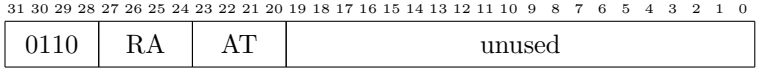
A branch is unconditionally taken. The PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

7.3.7 JALR

Assembler Syntax

```
JALR   RA, AT
```

Encoding



Operation

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

Description

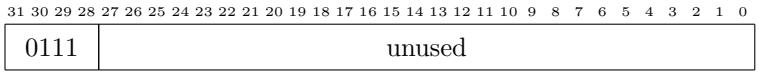
First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

7.3.8 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

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

7.3.9 SKPEQ

Assembler Syntax

```
SKPEQ  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	unused	0000

Operation

```
if (SR1 == SR2) {
    PC = incrementedPC + 1
}
```

Description

The incremented PC is further incremented by 1 if SR1 is equal to SR2.

7.3.10 SKPGT

Assembler Syntax

```
SKPGT  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	unused	0001

Operation

```
if (SR1 > SR2) {
    PC = incrementedPC + 1
}
```

Description

The incremented PC is further incremented by 1 if SR1 is greater than SR2.

7.3.11 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.

7.3.12 SLL

Assembler Syntax

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

Encoding

Operation

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

Description

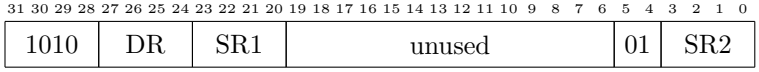
The value stored in SR1 is logically left shifted by the value stored in SR2, and the result is stored in DR.

7.3.13 SRL

Assembler Syntax

```
SRL    DR, SR1, SR2
```

Encoding



Operation

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

Description

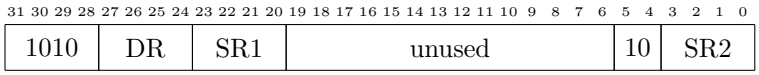
The value stored in SR1 is logically right shifted by the value stored in SR2, and the result is stored in DR.

7.3.14 SRA

Assembler Syntax

```
SRA    DR, SR1, SR2
```

Encoding



Operation

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

Description

The value stored in SR1 is arithmetically right shifted by the value stored in SR2, and the result is stored in DR. **NOTE THE DIFFERENCE BETWEEN SHIFTS: Logical right shift will fill the resulting space with 0s, while arithmetic right shift sign-extends the MSB**.

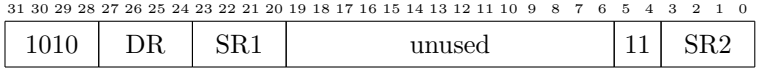
> **HINT:** Like there is a component to add or subtract, there is also a component that performs shifts. Try to find and play around with this component to see how it can be helpful in your project.

7.3.15 ROR

Assembler Syntax

```
ROR    DR, SR1, SR2
```

Encoding



Operation

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

Description

Bits in SR1 are "rotated" by SR2 times as if the left and right ends of the register were joined. The values that are shifted into the left will be the values that are shifted off from the right during the right-shift operation.

8 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:

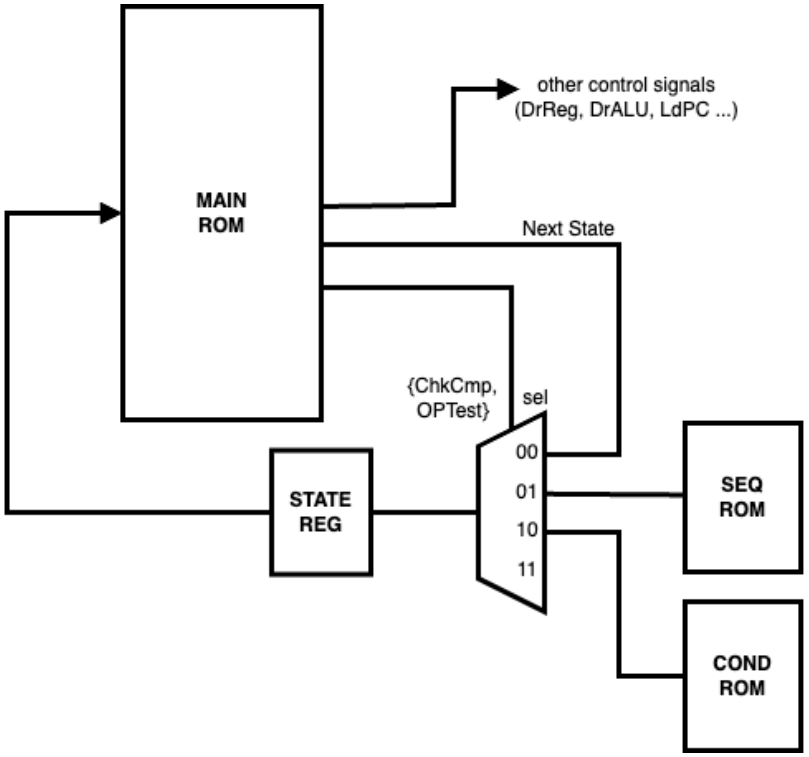


Figure 2: Three ROM Microcontrol Unit

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 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 (Option 1)

ALU2	ALU1	ALU0	IR[5]	IR[4]	Function
0	0	0	N/A	N/A	ADD
0	0	1	N/A	N/A	SUB
0	1	0	N/A	N/A	NAND
0	1	1	N/A	N/A	A + 1
1	IR[5]	IR[4]	0	0	SLL
1	IR[5]	IR[4]	0	1	SRL
1	IR[5]	IR[4]	1	0	SRA
1	IR[5]	IR[4]	1	1	ROR

Table 6: ALU Function Map (Option 2)

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

IR[5]	IR[4]	Function
0	0	SLL
0	1	SRL
1	0	SRA
1	1	ROR

9 Appendix C: Control Logic, Expanded

For this project, we only ask you to implement SKPEQ and SKPGT. The particulars of that implementation are up to you, so long as you keep in line with the spec as given in the rest of the document. In the past, however, we asked students to implement all possible arithmetic comparisons. For those interested in making a more generalized comparison mechanism, the details are here.

NOTE: NO INFORMATION HERE IS REQUIRED TO COMPLETE PROJECT 1. REFER TO 3.1.5 FOR THE MOST CONCISE INFORMATION ON PROJECT REQUIREMENTS.

The "comparison logic" in Figure 1 is responsible for performing the comparison calculations associated with the SKP instructions. There are two separate comparisons: `(C0, C1)`. We calculate C0 by checking if $SR1 > SR2$, and we calculate C1 by checking if $SR1 < SR2$.

We recommend computing the difference between the two registers and comparing it against 0. You must calculate both C0 and C1 and store this into the "Comparison Register" (or CmpReg). Note: CircuitSim has a built-in comparator circuit that is very useful here.

After storing the result in CmpReg, the LC-2199 uses unique comparison logic to determine whether to skip or not. That equation is $!IR[2] == ((IR[1] == C1)\&\&(IR[0] == C0))$. This equation uses bits in the IR to decide what comparison to make. This may seem like gibberish, so here's a breakdown:

IR[0] tells us whether or not we should skip if $SR1 > SR2$ (C0).

IR[1] tells us whether or not we should skip if $SR1 < SR2$ (C1).

IR[2] tells us whether or not we should negate the equation.

For example, if we want to skip when SR1 is less than SR2, IR[2:0] should be 010 because we don't want to negate C0 or C1, we do want to check C1 (the calculation of $SR1 < SR2$), and we ignore C0 (the calculation of $SR1 == SR2$).

Another example: if we want to skip when $SR1 > SR2$, then we would set IR[2:0] = 000. This is because we don't want to negate the conditions in IR[1:0], which are checking if $SR1 > SR2$ and $SR1 < SR2$ respectively. If something not less than nor equal to another, it must be greater than. Note that everything except for the CmpReg (which stores C0 and C1) should be purely combinational circuitry.

Determining the combinations of IR[2:0], C0, and C1 that result in each arithmetic comparison is left as an exercise to the reader.