1 Requirements

- Download the proper version of Logisim. Be sure to use the most updated version. **DO NOT USE BRANDONSIM FROM CS 2110!**

  You may download Logisim from http://www.cburch.com/logisim/download.html.

- Logisim is not perfect and does have small bugs. In certain scenarios, files have been corrupted and students have had to re-do the entire project. Please back up your work using some form of version control, such as a local git repository. **Do not use public git repositories, it is against the Georgia Tech Honor Code**.

- The LC3-2200a 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 Logisim 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 (AKA control-unit) to control your computer and actually run programs on it.

Note: You will need to have a working knowledge of Logisim. 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

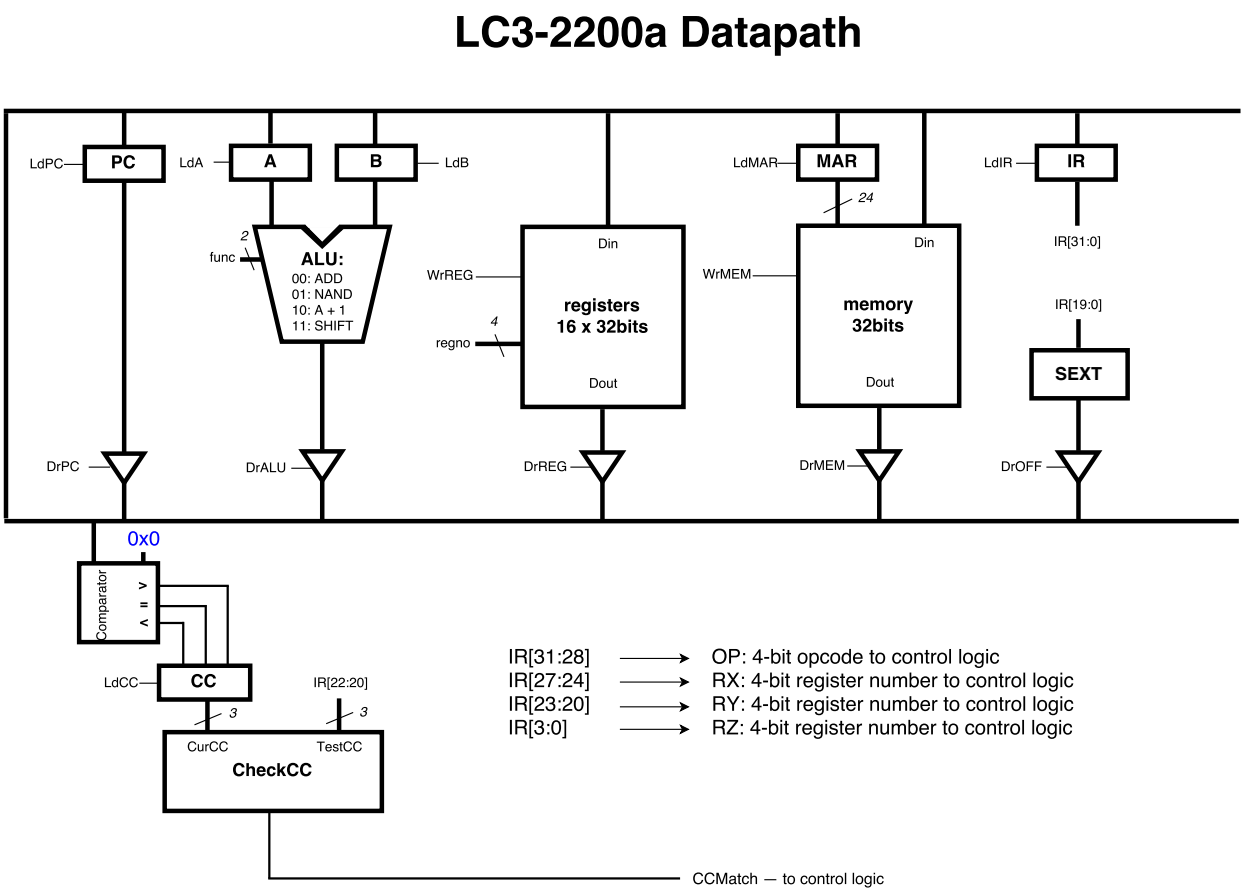


Figure 1: Datapath for the LC3-2200a Processor

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 LC3-2200a datapath in Logisim using what you have just learned.

You must do the following:

1. Learn and understand the LC3-2200a ISA. The ISA is fully specified and defined in Appendix A: LC3-2200a 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 Logisim, implement the LC3-2200a datapath. To do this, you will need to use the details of the LC3-2200a datapath defined in Appendix A: LC3-2200a Instruction Set Architecture. You should model your datapath on Figure 1.

3.1 Hints

3.1.1 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.

3.1.2 Memory Addresses

Because of Logisim limitations, the RAM module is limited to no more than 24 address bits. Therefore, per our ISA, any 32-bit values used as memory addresses will be truncated to 24 bits (with the 8 most significant bits disregarded). We recommend you implement this (i.e. truncate the MSB bits) before feeding the address value from the MAR (Memory Address Register) to the RAM.

3.1.3 Implementing Condition Codes

The following instructions modify the Condition Codes (CC) register: ADD, ADDI, NAND, LDR, LEA, SHF. The N, Z, and P bits can all be stored in a single register, or you may use multiple registers if you desire. **We recommend using the Logisim comparator to produce the N, Z, and P bits**.

To implement the BR instruction, you need to design and implement the CheckCC module indicated in Figure 1. The CheckCC module takes as input the current condition codes from the CC register, as well as the N, Z, and P bits from the instruction. The module then uses combinational logic to produce the CCMatch bit for your control logic.

Carefully examine the specification of the BR instruction in Section 7.4.4 and design your combinational logic to match the logic specified in "Operation".

3.1.4 SHF instruction

Logisim provides a shifter component that we highly recommend you use. The shifter is configured for a particular shift type (logical left, logical right, or arithmetic right), therefore you may have to use more than one. We recommend you use the same sign extend / drive offset (DrOFF) mechanism used in ADDI to pass the shift vector to the ALU, and then split out the A, D, and distance bits within your ALU.

3.1.5 Zero Register

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. **Do not forget to do this or you will lose points!**

4 Phase 2 - Implement the Microcontrol Unit

In this phase of the project, you will use Logisim to implement the microcontrol unit for the LC3-2200a 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 Logisim. 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 LC3-2200a 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. Fill out the microcode.xlsx (Excel spreadsheet) file that we have provided. You will need to mark which control signal is high (that is 1) for each of the states. We have given you a starting template.
2. After you have completed all the states, convert the binary strings you just computed into hex and move them to the main ROM (the Excel sheet will do this for you; just copy-and-paste the hex column into the Logisim ROM). Do the same for the sequencer and CCMatch 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.

   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 **bitcount.s** and **misctest.s** programs as a more comprehensive test case.

6 Deliverables

Please submit all of the following files in a **.tar.gz** archive. You must turn in:

- Logisim Datapath File (LC3-2200a.circ)
- Microcode file (microcode.xlsx)

Don't forget to sign up for a demo slot! We will announce when these are available. Failure to demo results in a 0.

We do not accept late submissions. If your assignment is not submitted on T-Square by the deadline, you will receive a 0.

Precaution: You should always re-download your assignment from T-Square 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.

7 Appendix A: LC3-2200a Instruction Set Architecture

The LC3-2200a (Little Computer 3-2200 Type A) is a simple, yet capable computer architecture. The LC3-2200a combines attributes of both the LC3 architecture, and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC3-2200a 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 24 bits on access, discarding the 8 most significant bits if the address was stored in a 32-bit register. This provides roughly 67 MB of addressable memory.

7.1 Registers

The LC3-2200a 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	Reserved for the Assembler	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 a general purpose register. You should not use it because the assembler will use it in processing pseudo-instructions.
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 your anchor on the stack. It keeps track of the top of the activation record for a subroutine.
9. **Register 14** is used to point to the first address on the activation record for the currently executing process. Don't worry about using this register.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing. It is automatically used for this purpose by the subroutine jump instruction.

7.2 Instruction Overview

The LC3-2200a 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: LC3-2200a 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
+ADDI	0001	DR	SR1	immval20	
+NAND	0010	DR	SR1	unused	SR2
BR	0011	0000	0 n z p	PCoffset20	
JALR	0100	RA	AT	unused	
+LDR	0101	DR	BaseR	offset20	
+LEA	0110	DR	unused	PCoffset20	
STR	0111	SR	BaseR	offset20	
+SHF	1000	DR	SR1	shiftvec20[1]	
HALT	1111	unused			

NOTE: Instructions marked with + modify condition codes.
[1] See Section 7.4.9 for the format of the shift vector.

7.3 Condition Codes

In addition to the general-purpose registers, the LC3-2200a also keeps state in the form of the **condition codes** register. These flags are updated by any instruction that writes to a destination register (except JALR). If a signed interpretation of the value is negative, the **N - negative** flag is set. If zero, the **Z - zero** flag is set. If positive, the **P - positive** flag is set.

The condition codes can be tested by the **BR - conditional branch** instruction. See the specification of the branch instruction in Section 7.4.4 for details.

When the system first starts up, the condition codes are indeterminate. Therefore, at least one instruction that modifies the condition codes should be executed before attempting a branch.

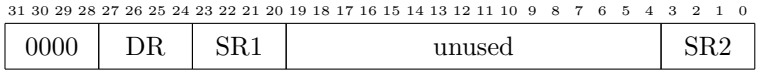
7.4 Detailed Instruction Reference

7.4.1 ADD

Assembler Syntax

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

Encoding



Operation

```
DR = SR1 + SR2;
SetConditionCodes();
```

Description

The ADD instruction adds the source operand obtained from SR2 to the source operand obtained from SR1. The result is stored in DR.

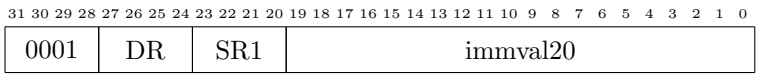
The condition codes are set, based on whether the result is negative, zero, or positive.

7.4.2 ADDI

Assembler Syntax

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

Encoding



Operation

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

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 second source operand is added to the first source operand, and the result is stored in DR.

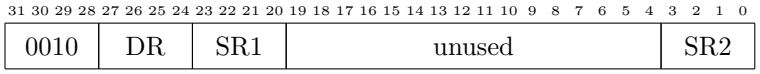
The condition codes are set, based on whether the result is negative, zero, or positive.

7.4.3 NAND

Assembler Syntax

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

Encoding



Operation

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

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.

The condition codes are set, based on whether the result is negative, zero, or positive.

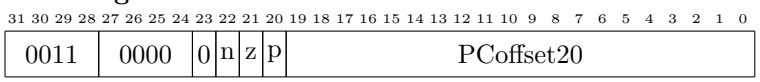
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

7.4.4 BR

Assembler Syntax

```
BRn    LABEL        BRnz   LABEL
BRz    LABEL        BRzp   LABEL
BRp    LABEL        BRnp   LABEL
BR     LABEL        BRnzp  LABEL      (BR is equivalent to BRnzp)
```

Encoding



Operation

```
if ((n AND N) OR (z AND Z) OR (p AND P)) {
    PC = PC + SEXT(PCoffset20);
}
```

Description

The condition codes specified by the state of bits [22:20] are tested. If bit [22] is set, N is tested; if bit [22] is clear, N is not tested. If bit [21] is set, Z is tested, etc. If any of the condition codes tested is set, the program branches to the location specified by adding the sign-extended PCoffset20 field to the incremented PC (address of instruction + 1). **In other words, the PCoffset20 field specifies the number of instructions, forwards or backwards, to branch over.**

7.4.5 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
0100	RA	AT	unused

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.

7.4.6 LDR

Assembler Syntax

```
LDR    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
0101	DR	BaseR	offset20

Operation

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

Description

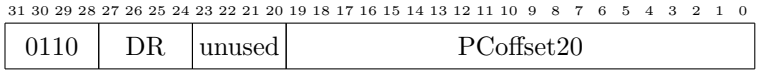
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

7.4.7 LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding



Operation

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

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). This instruction effectively performs the same computation as the BR instruction, but rather than performing a branch, merely stores the computed address into register DR.

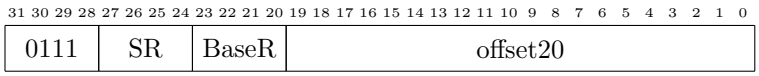
The condition codes are set, based on whether the result is negative, zero, or positive.

7.4.8 STR

Assembler Syntax

```
STR    SR, offset20(BaseR)
```

Encoding



Operation

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

Description

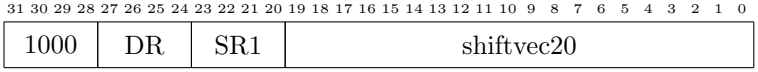
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the contents of the register specified by bits [23:20]. The 32-bit word obtained from register SR is then stored at this address.

7.4.9 SHF

Assembler Syntax

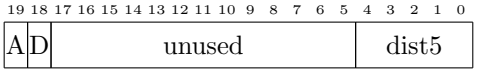
```
SHFLL  DR, SR1, dist5    ; left shift logical
SHFRL  DR, SR1, dist5    ; right shift logical
SHFRA  DR, SR1, dist5    ; right shift arithmetic
```

Encoding



Shift Vector Encoding

The embedded shift vector is a 20-bit value that defines both the distance of the shift and the type of shift to perform.



Operation

```
if (shiftvec[D] == 0) {
    DR = SR1 << shiftvec[dist5];
} else {
    if (shiftvec[A] == 0) {
        DR = SR1 >> shiftvec[dist5],0;
    } else {
        DR = SR1 >> shiftvec[dist5],SR1[31];
    }
}
SetConditionCodes();
```

Description

If the D bit of the shift vector is 0, the source operand in SR1 is shifted left by the number of bit positions indicated by the dist5 field. If D is 1, the source operand is shifted to the right by dist5 bits.

When shifting to the right, the A bit of the shift vector indicates whether the sign bit of the original source operand is preserved. When A is set to 1, the right shift is an arithmetic shift and the original SR1[31] is shifted into the vacated bit positions. The result stored in DR. Otherwise the shift is a logical shift and zeroes are shifted in.

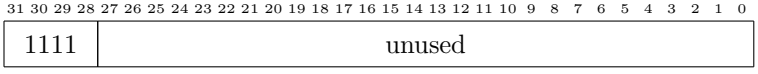
The condition codes are set, based on whether the result is negative, zero, or positive.

7.4.10 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

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

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 CC register to determine which signals to assert. For example, since the CCMatch line is an input for the address, every microstate would have to have an address for CCMatch = 0 as well as CCMatch = 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 CCMatch ROM, which helps determine whether or not to branch during the BR instruction.

Examine the following:

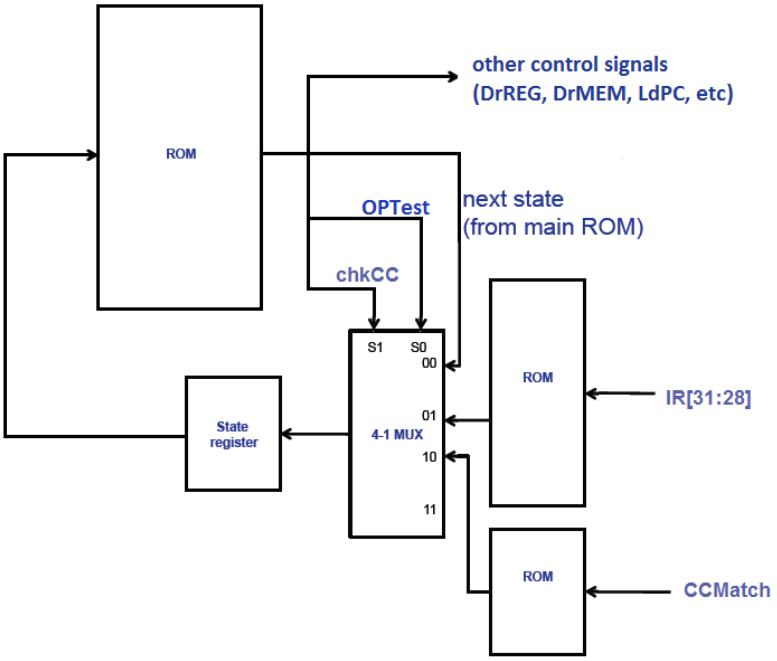


Figure 2: Three ROM Microcontrol Unit

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 CCMatch 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 ChkCC 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 condition code circuitry determines the next state (such as in the BR instruction), the ChkCC 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 CCMatch ROM should have one address for taking the branch and one for not taking the branch.

Note: Logisim has a minimum of two address bits for a ROM (i.e. four addresses), even though only one address bit (two addresses) is needed for the CCMatch ROM. Just ignore the other two addresses. You should design it so that the high address bit for this ROM is permanently set to zero.

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 part 1 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 with a tool we are providing. Then you must design and build (in Logisim) 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 insure that it is working properly. Finally, you will load the Control Unit's ROMs with the output of the tool.

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]	5	NextState[5]	10	DrOFF	15	LdB	20	RegSelHi
1	NextState[1]	6	DrREG	11	LdPC	16	LdCC	21	ALULo
2	NextState[2]	7	DrMEM	12	LdIR	17	WrREG	22	ALUHi
3	NextState[3]	8	DrALU	13	LdMAR	18	WrMEM	23	OPTest
4	NextState[4]	9	DrPC	14	LdA	19	RegSelLo	24	ChkCC

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

ALUHi	ALUlLo	Function
0	0	ADD
0	1	NAND
1	0	A + 1
1	1	SHIFT