1 Requirements

- Download VirtualBox (click here), and the latest version of Linux Mint. Click here to download.
- Alternatively, use 32-bit or 64-bit Ubuntu or Debian. Click here to download.
- Download build essentials for linux as needed. If you are using Mint/Ubuntu/Debian type in `sudo apt-get install build-essential`.
- Download the proper version of Logisim. Be sure to use the most updated version. **DO NOT USE BRANDONSIM FROM CS 2110!** Click here to go to the download page.
- 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**

2 Project Overview and Description

Project 1 is designed to give you a good feel for exactly how a processor works. In Phase I, 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 II and III 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. Always remember to see the TAs if you need help.

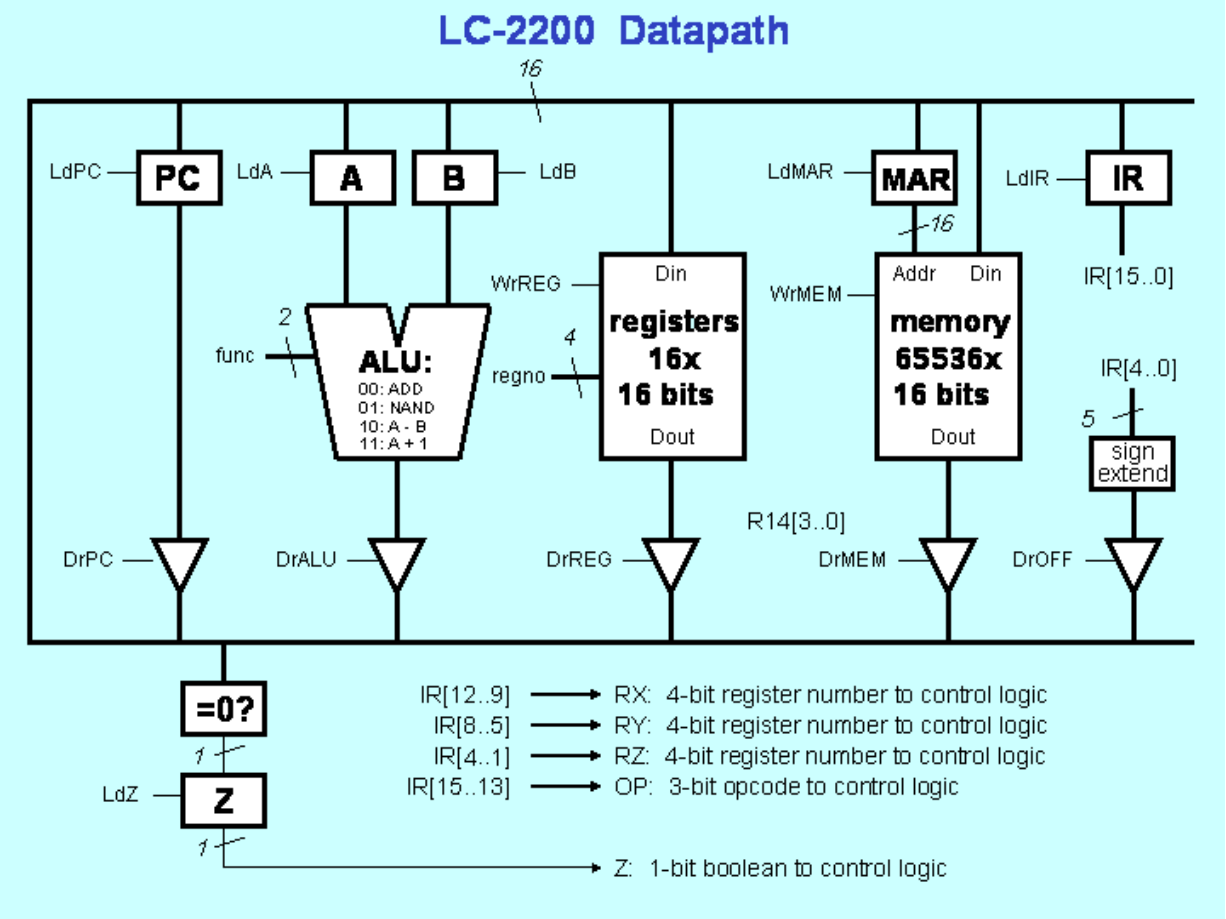

Figure 1: Datapath for the LC 2200 Processor

3 Phase 1 - Implement the Datapath

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 2200 16 bit datapath in Logisim using what you have just learned.

You must do the following:

1. Learn and understand the LC 2200 16 ISA. Simply read and understand Appendix B: LC 2200 Instruction Set Architecture. **DO NOT MOVE ON UNTIL YOU HAVE FULLY READ AND UNDERSTOOD THE ISA AS IT WILL BE IMPOSSIBLE TO IMPLEMENT THE DATAPATH IF YOU DO NOT!**

2. Using Logisim, implement the LC 2200 datapath. The basic operation of the datapath will be discussed in the class and is delineated in the book. 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.

Please use figure 1 while constructing your datapath in Logisim.

4 Phase 2 - Implement the Microcontrol Unit

In this phase of the project, you will use Logisim to implement the microcontrol unit for the LC 2200 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 A: 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 connection between the datapath and the microcontrol unit before moving on.
2. Implement the Microcontrol Unit using Logisim. The above link 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 2200 datapath schematic (see figure 1).

5 Phase 3 - Microcode and Testing

In this final stage of the project, you will write the microcode control signals 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.ods (open office spread sheet) file that we have provided. You will need to mark which control signal is high (that is 1) for each of the states. Please read the **README** file for more details. Feel free to modify microcode.ods as you deem necessary. We have just given you a starting template.
2. After you have completed all the states, covert the binary strings you just computed into hex and move them to the main ROM. Appendix A describes what goes in the sequencer and Z ROMs.
3. Connect the completed control unit to the datapath you implemented in Phase 1. Using the figure 1 and the microcontrol unit schematic, connect the control signals to their appropriate spots.
4. It is now time to test your completed computer. Use the provided assembler (found in the "Assembly" folder) to convert a test program from LCC2200 assembly to hex (run python lc2200-16as.py –help for help). 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.
5. Finally write a program that combines multiple instructions, and try running that on the processor you just implemented, as an "ultimate test." Name this file "utest.s." Note: Please write the expected output at the end of execution as a comment on the top of the "ultimate test" file.

6 Deliverables

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

- Logisim Datapath File (LC-2200-16.circ)
- Microcontrol Unit Logisim File (you can implement this in the same file as the datapath if you wish)
- Microcode file (microcode.ods) - Make sure you have a column with your final hex values for each MACRO state.

- The assembly source file you wrote as the "ultimate test." (Name this file "utest.s")

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: 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 hardwire signals using a single ROM. The signle ROM solution will waste a tremendous amount of space since most of the microstates do not depend on the opcode or the Z register to determine which signals to assert. For example, since Z register is an input for the address, every microstate would have to have an address for Z = 0 as well as Z = 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 microsotate to go at the end of the FETCH state,
- and the OnZ ROM, which helps determine whether or not to branch during the BEQ instruction.

Examine the following:

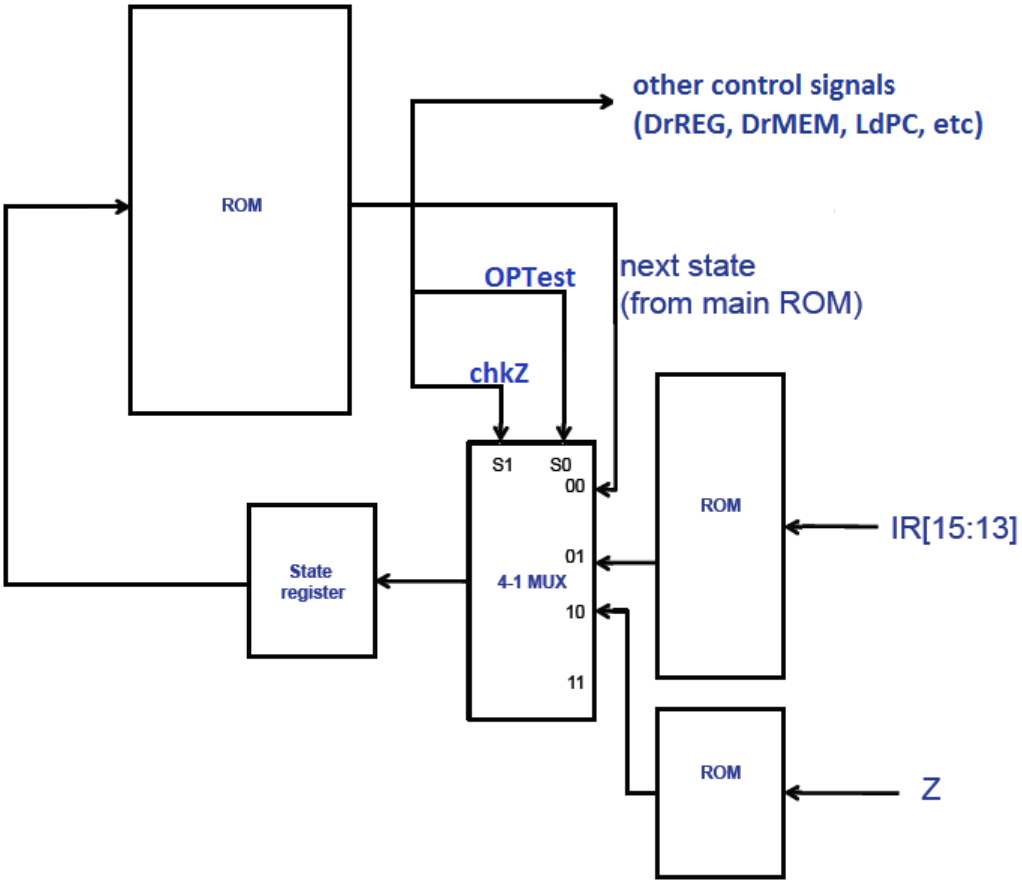

Figure 2: Three ROM Microcontrol Unit

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 OnZ 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 chkZ signals will be asserted. If the Op Code from the IR determines the next state (such as at the end of the Fetch state), the OpTest signal will be asserted. If the zero-detection circuitry determines the next state (such as in the BEQ instruction), the TestZ 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 OpCheck ROM should have one address per instruction, and the OnZ 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 OnZ 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 ROMs. 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 hex control signals you just generated.

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 1: ROM Output Signals

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

Table 2: Register Selection Map

RegSelHi	RegSelLo	Register
0	0	RX
0	1	RY
1	0	RZ
1	1	Unused

Table 3: ALU Function Map

ALUHi	ALUlLo	Function
0	0	ADD
0	1	NAND
1	0	A - B
1	1	A + 1

8 Appendix B: LC 2200 Instruction Set Architecture

The LC-2200-16 (Little Computer 2200-16 bits) is very simple, but it is general enough to solve complex problems. (Note: This is a 16-bit version of the ISA specification you will find in the Ramachandran & Leahy textbook for CS 2200.) This section describes the instruction set and instruction format of the LC-2200. The LC-2200-16 is a 16-register, 16-bit computer. All addresses are word-addresses. Although the 16 registers are for general purpose use, we will assign then special duties for convention (and for all that is good on this Earth).

Table 4: 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 temporary values. **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 used to handle interrupts.

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 finsihed executing. It is only supposed to be used by the JALR (Jump And Link Register) instruction.

There are four types of instructions: R-Type (Register Type), I-Type (Immediate value Type), J-Type (Jump Type), and S-Type (Stack Type).

Here is the instruction format for R-Type instructions (ADD, NAND):

Bits	15 - 13	12 - 9	8 - 5	4 - 1	0
Purpose	opcode	RX	RY	RZ	Unused

Here is the instruction format for I-Type instructions (ADDI, LW, SW, BEQ):

Bits	15 - 13	12 - 9	8 - 5	4 - 0
Purpose	opcode	RX	RY	2's Complement Offset

Here is the instruction format for J-Type instructions (JALR):

Bits	15 - 13	12 - 9	8 - 5	4 - 0
Purpose	opcode	RX	RY	Unused (all 0s)

Here is the instruction format for S-Type instructions (SPOP):

Bits	15 - 13	12 - 2	1 - 0
Purpose	opcode	Unused (all 0s)	Control Code

Table 5: Assembly Language Instruction Descriptions

Name	Type	Example	Opcode	Action
add	R	add $v0, $a0, $a2	000	Add contents of RY with the contents of RZ and store the result in RX.
nand	R	nand $v0, $a0, $a2	001	NAND contents of RY with the contents of RZ and store the result in RX.
addi	I	addi $v0, $a0, 7	010	Add contents of RY to the contents of the offset field and store the result in RX.
lw	I	lw $v0, 0x07($sp)	011	Load RX from memory. The memory address is formed by adding the offset to the contents of RY.
sw	I	sw $a0, 0x07($sp)	100	Store RX into memory. The memory address is formed by adding the offset to the contents of RY.
beq	I	beq $a0, $a1, done	101	Compare the contents of RX and RY. If they are the same, then branch to address PC + 1 + Offset, where PC is the address of the beq instruction. **Memory is word addressed**.
jalr	J	jalr $at, $ra	110	First store PC + 1 in RY, where PC is the address of the jalr instruciton. Then branch to the address in RX. If RX = RY, then the processor will store PC + 1 into RY and end up branching to PC + 1.
halt	S	halt	111	Tells the processor to halt. No further instructions should be executed.

Finally, the assembler supports labels which represent the address of the line it is on. If a label is used in a BEQ instruction, it will evaluate to some relative offset.

For example:

```
(address 0):          add $s0, $zero, $zero
(address 1):  loop:   addi $s0, $s0, -1
(address 2):          beq $s0, $zero, end
(address 3):          beq $zero, $zero, loop
(address 4):  end:    halt
```

Becomes:

```
(address 0):  000 1001 0000 00000 or 0x1200
(address 1):  010 1001 1001 11111 or 0x533F
(address 2):  101 1001 0000 00001 or 0xB201
(address 3):  101 0000 0000 11101 or 0xA01D
(address 4):  111 0000 0000 00000 or oxE000
```