Project 2: LC-3 Datapath

Prabhav Gupta, Gregory Gould, Jonathan Marto, Athanasios Taprantzis

Fall 2023

Contents

1 Introduction

Welcome everyone to Project 2! **Please read through the entire document before starting**. Oftentimes things are elaborated on below where they are introduced, so reading the entire document can give you a better grasp on things. **Start early** and if you get stuck, there's always Piazza or office hours.

In this project, you'll be doing the following:

- Implementing some basic sequential logic elements **(3.1)**.
- Building three missing pieces of the LC-3 hardware **(3.2)**.
- Writing the microcode for the bulk of the LC-3 instructions **(3.4)**.

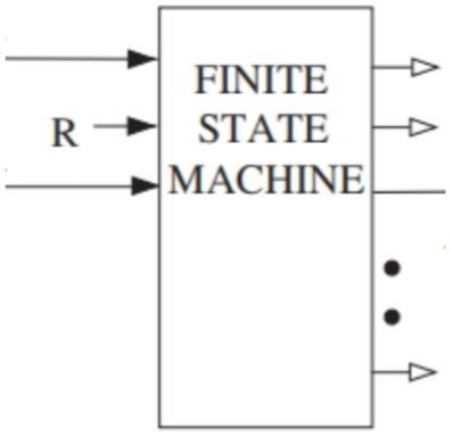
1.1 Latches

The first thing you'll implement are sequential logic elements. These allow us to retain data in a circuit, in other words "remember" a previous value. This allows us to have logic that depends on the last event that happened in a sequence of events, in other words sequential logic.

The goal of this section is to produce a register. A register is an edge-triggered sequential logic circuit element that can store a binary value. For more on this section, see Implementation: Latches

1.2 The LC-3

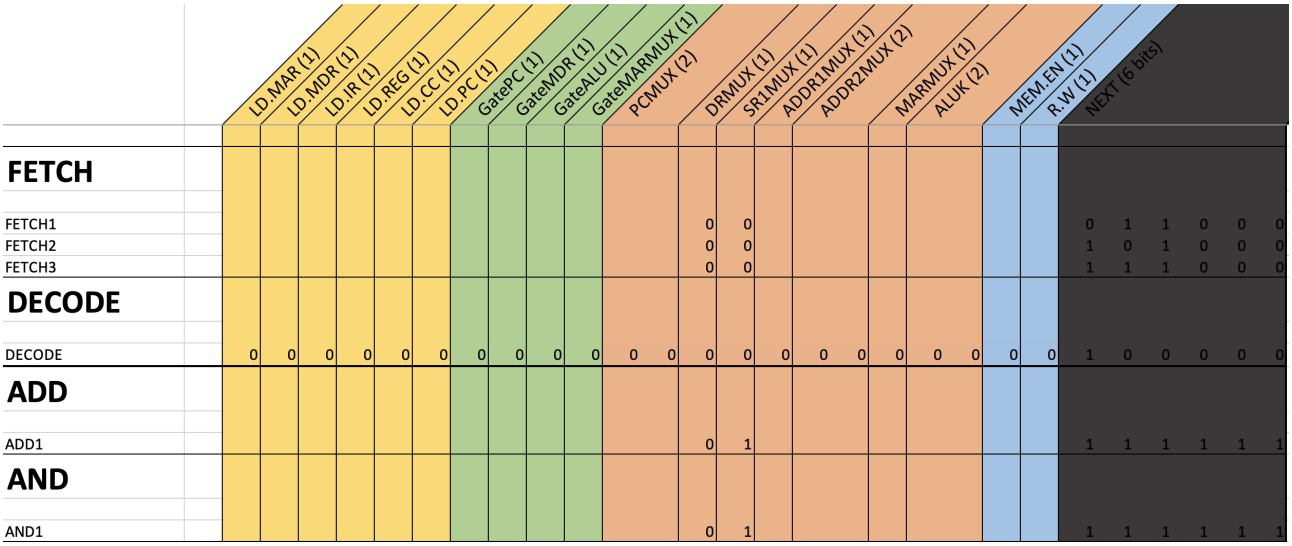
The LC-3 datapath we've discussed in class contains a lot of pieces very similar to circuits we've seen or even made before (e.g. an ALU, a register file with 8 edge-triggered general purpose registers, a RAM unit, etc.). One piece that we've mostly referred to as a "black-box" in the past is the micro-controller. It's responsible for controlling the entire datapath and getting it to properly execute the instructions that we give it. That's a big task! So, how does the micro-controller actually work? In this project, we'll build a few datapath components to develop some familiarity with the LC-3, and then we'll actually write the "microcode" which allows the micro-controller to function.



The micro-controller, shown above, is a finite state machine. It has 59 possible states (holy cow!) and, thus, needs 6 bits to store all its possible states. It also has 49 output bits composed of output flags. 10 of these bits are used to determine the next state and the remaining 39 extend throughout the datapath to control other pieces of the LC-3. That would be a lot of very complex hardware—**if it were built entirely in hardware**.

As it turns out, there is an easier way. We can actually use a ROM (Read-Only Memory) in order to specify the behavior of each distinct state in the state machine (e.g. each instruction will map to a series of entries in the ROM. Each entry in the ROM represents a micro-state, which is an individual state of the finite state machine and a *component* of a larger sequence of states known as a macro-state. There are 3 macro-states in the LC-3: FETCH, DECODE, and EXECUTE. Each macro-state requires between 1-5 micro-states to complete, depending on the complexity of the instruction.

What does a ROM entry look like? We encourage you to go ahead and open up `microcode.xlsx`, on the microcode sheet, to follow along.



	LD.MAR (1)	LD.MDR (1)	LD.IR (1)	LD.REG (1)	LD.CC (1)	LD.PC (1)	GatePC (1)	GateMDR (1)	GateALU (1)	GateMARMUX (1)	PCMUX (2)	DRMUX (1)	SR1MUX (1)	ADDR1MUX (1)	ADDR2MUX (2)	MARMUX (1)	ALUK (2)	MEM.EN (1)	R.W (1)	NEXT (6 bits)
FETCH																				
FETCH1												0	0							0 1 1 0 0 0
FETCH2												0	0							1 0 1 0 0 0
FETCH3												0	0							1 1 1 0 0 0
DECODE																				
DECODE	0	0	0	0	0	0	0	0	0	0	0 0	0	0	0	0 0	0	0 0	0	0	1 0 0 0 0 0
ADD																				
ADD1												0	1							1 1 1 1 1 1
AND																				
AND1												0	1							1 1 1 1 1 1

A ROM entry is basically a long binary string. The last few bits of it handle the transition to the next state—you don't need to worry about this at all during this project, so we've covered it in dark grey on the right. **Do NOT modify the NEXT bits.** Each of the other bits corresponds to a signal asserted onto the datapath during that clock cycle—this is what you will fill out!.

We've simplified and removed a number of micro-states and control signals which aren't directly a part of the LC-3's main instructions. You are given the microcode for some of these micro-states, along with entries for the DRMUX and SR1MUX—**do NOT change these**. Your task is to fill in the rest and finish the LC-3 micro-controller!

1.3 Assignment Files

- `latches.sim` - a CircuitSim file in which you will build sequential logic components.
- `LC3.sim` - a large CircuitSim file containing the LC-3 **AND** a "Manual LC-3" which does not need to be modified in any way but is simply present as a tool for you while writing microcode. You should only modify the PC, CC-logic, and ALU subcircuits in this file.
- `microcode.xlsx` - an Excel document in which you will write your microcode. **Do not touch cells that have been blacked out.**
- `ROM.dat` - a text file which you can paste your microcode into and then import into the LC-3.
- `tests/` - a subdirectory which contains a number of test cases you can use to verify the functionality of your circuit and microcode.
- `proj2-tester.jar` - a local tester that you can use to test all of the components of the project. This official tester is available on Gradescope (where it will be return your grade).

- `LC-3InstructionsDetail.pdf` - a PDF with descriptions and pseudocode for each instruction.

1.4 Elements to complete

1. Implement the RS Latch, Gated D Latch, D Flip-Flop, and Register subcircuits in `latches.sim`.
2. Implement the ALU, PC, and CC-Logic subcircuits in `LC3.sim`.
3. Complete the LC-3 microcode in `microcode.xlsx`.

2 Setup

The software you will be using for this project and all future circuit based assignments is called CircuitSim - an interactive circuit simulation package.

CircuitSim is a powerful simulation tool designed for educational use. This gives it the advantage of being a little more forgiving than some of the more commercial simulators. However, it still requires some time and effort to be able to use the program efficiently.

Please follow the instructions in the file "CircuitSim Installation Guide.pdf" in Canvas, under "Files > Course Software > CircuitSim" to install CircuitSim.

All .sim files should be opened in CircuitSim.

Please do not move, rename, or remove any of the provided inputs/outputs.

3 Implementation

3.1 Latches

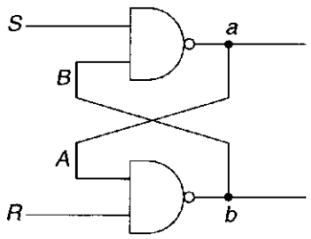
For this part of the assignment, you will build your own register from the ground up. For more information about each subcircuit, refer to your textbook. All work for this portion of the assignment will be done in the `latches.sim` file.

3.1.1 RS Latch

In the **"RS Latch" subcircuit in the `latches.sim` file**, you will start by building a RS latch using NAND gates, as described in your textbook. RS Latch is the basic circuit for sequential logic. It stores one bit of information and has 3 important states:

1. S=1, R=1: This is called the **Quiescent State**. In this state the latch is storing a value, and nothing is trying to change that value.
2. S=0, R=1: By changing momentarily from the Quiescent State to this state, the value of the latch is changed so that the output Q now stores a 1.
3. S=1, R=0: By changing momentarily from the Quiescent State to this state, the value of the latch is changed so that the output Q now stores a 0.

Once you set the bit you wish to store, change back to the Quiescent State to keep that value stored.
Notice that the circuit has two output pins; one is the bit the latch is currently storing, and the other is the opposite of that bit.
Note: In order for the RS Latch to work properly (for testing purposes, not for building), both R and S cannot be 0 at the same time (illegal state).



3.1.2 Gated D Latch

In the **"Gated D Latch" subcircuit in the `latches.sim` file**, using your RS latch subcircuit (in the Circuits tab), implement a Gated D Latch as described in the textbook. You are **not** allowed to use the built-in SR Latch in CircuitSim to build this circuit.

The Gated D Latch is made up of an RS Latch as well as two additional gates that control when a value can be stored.

The value of the output can only be changed when Write Enable is set to 1. Notice that the Gated D Latch subcircuit only has one output pin, so you should disregard the inverse output Q' of your RS Latch.

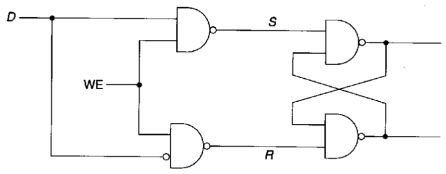


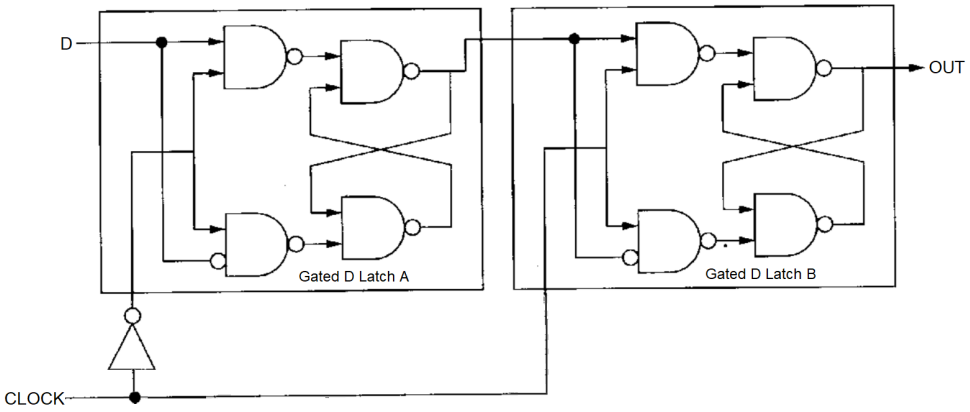
Figure 3.19 A gated D latch

3.1.3 D Flip-Flop

In the **"D Flip-Flop" subcircuit in the `latches.sim` file**, using the Gated D Latch circuit you built (in the Circuits tab), create a D Flip-Flop.

A D Flip-Flop is composed of two Gated D Latches back to back, and it implements edge triggered logic.

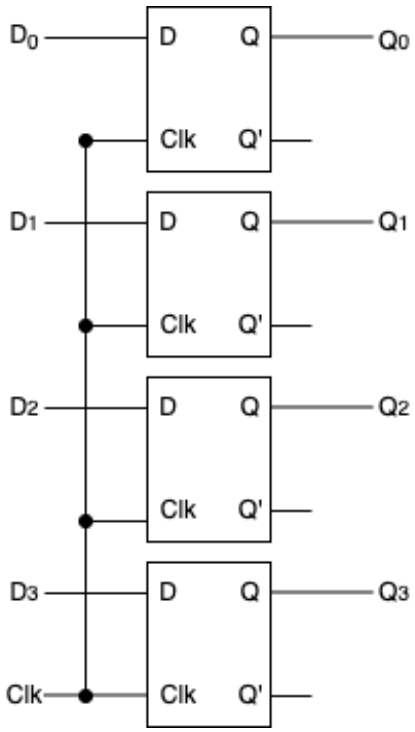
Your D Flip-Flop output should be able to change on the **rising edge**, which means that the state of the D Flip-Flop should only be able to change at the exact instant the clock goes from 0 to 1.



3.1.4 Register

In the **"Register" subcircuit in the `latches.sim` file**, using the D Flip-Flop you just created (in the Circuits tab), build a 4-bit Register.

Your register should also use edge-triggered logic. The value of the register should change on the rising edge.



3.2 Completing the LC3 SubCircuits

All work for this section of the assignment will be done in the `LC3.sim` file.

Note: **DO NOT move the position of the inputs and outputs** in the LC3 subcircuits; this could break the rest of the LC3 simulator.

3.2.1 ALU

You will need to build the ALU (Arithmetic Logic Unit) subcircuit in `LC3.sim`.

Note: You may use the built-in adder under the Arithmetic tab!

The ALU will perform one of 4 functions and Output it depending on the ALUK signal:

1. **ALUK = 0b00:** A + B
2. **ALUK = 0b01:** A & B
3. **ALUK = 0b10:** NOT A
4. **ALUK = 0b11:** PASS A

You should be able to use what you learned from Project 1 to populate this subcircuit easily.

3.2.2 PC

You will need to build the PC (Program Counter) subcircuit in `LC3.sim`.

Note: You may use the built-in multiplexer under the Plexer tab!

The PC is a 16 bit register that holds the address of the next instruction to be executed. During the **FETCH** stage, the contents of the PC are loaded into the memory address register (MAR), and the PC is updated with the address of the next instruction. There are three scenarios for updating the PC:

1. The contents of the PC are incremented by 1.
   Selected when PCMUX = 0b00.
2. The result of the ADDR (an address-adding unit) is the address of the next instruction. The output from the ADDR should be stored in the PC. This occurs if we use the branching instruction (BR).
   Selected when PCMUX = 0b01.
3. The value on the BUS is the address of the next instruction. The value on the BUS should be stored into the PC. For example, this is used during the JMP instruction.
   Selected when PCMUX = 0b10.

The PC should only be loaded on a rising clock edge and when the LD.PC signal is on.

Ensure that you don't reach the unused case (PCMUX = 0b11) of the circuit, or else spooky stuff might happen (undefined behavior in LC-3).

3.2.3 CC-Logic

You will implement the CC-Logic subcircuit in `LC3.sim`.

The LC-3 has three condition codes: N (negative), Z (zero), and P (positive). These codes are saved on the next rising edge after the LC-3 executes Operate or Load instructions that include loading a result into a general purpose register, such as ADD and LDR.

For example, if ADD R2, R0, R1 results in a positive number, then NZP = 001.

The CC subcircuit should set N to 1 if the input is negative, set Z to 1 if the input is zero, and set P to 1 if the input is positive. Only 1 bit in NZP should be set at any given time. Zero is not considered a positive number.

Bit 2 (the MSB) is N, Bit 1 is Z, Bit 0 is P.

With that in mind, set the correct bit and implement this circuit in the CC-Logic subcircuit.

Hint: you can use a comparator for this subcircuit! They are found in the Arithmetic tab in CircuitSim.

3.3 Using Manual LC-3 (for testing only)

- Once you have your PC, CC-Logic, and ALU complete, you can begin playing around with the instructions and control signals to understand how the LC-3 works.
- In lecture and in lab, we have covered the control signals in the datapath and how to use them when tracing an instruction.
- The first thing you will want to do is use the Custom-Bus, GateBUS, and LD.IR signals to set a custom IR value on the next rising edge. Until you figure out the micro-states for FETCH, you can use these steps to set your IR with any instruction you want to work on.
- Once you have an idea of how FETCH works, you can load some instruction(s) in the RAM. In order to do that:
  1. right-click the RAM near the bottom of the Manual LC-3 circuit
  2. select "edit contents"
  3. click "Load from file"

4. locate and select one of the provided test files in the project (ex: addand.dat)
5. close the edit contents menu

- Now that you have loaded the RAM with a program, you can fetch instructions into the IR.
- Now that you have an instruction in the IR, you can start executing it. In order to do that you can turn on the different signal pins on the right in order to control the datapath and move data around like we did in lecture/lab. Once you think you know how an instruction is executed, you can enter it into the microcode spreadsheet, the process for which is outlined below.
  - Tests inside the `tests/` directory have a comment at the end of the .asm file which explains the system state after the end of the program's execution. You must ensure that this is the system state after you have run every instruction sequentially through the simulator.
  - To test whether the program acted correctly, go to the 'LC-3' circuit and double-click into the 'REG FILE' element **that is placed in the datapath**. Note: you **should not** just click into the 'REG FILE' subcircuit, as this will not properly load the state of the specific 'REG FILE' element that's built into the LC-3, just some generic REG FILE.
  - Bonus tip: Use Ctrl-R to reset the simulator state and easily clear RAM and registers to 0 in order to test again.
- If you are familiar with LC-3 assembly (you will learn it throughout the next few weeks), you are welcome to write your own test programs to verify your code. Make sure that your programs **do not** start at x3000, as in this project ***only*** we will start execution at x0000 for simplicity's sake. You can compile LC-3 assembly projects to machine code by following the LC-3 ISA, and then create your own RAM.dat files. However, we can't guarantee that we'll be able to help with these test cases in office hours.

3.4 Writing the Microcode

ONLY open this Excel Sheet in Microsoft Excel. Do NOT use other applications.
If need help accessing Excel: https://oit.gatech.edu/email

Note: We recommend you read this entire section before starting to fill out the microcode sheet.

- **DO NOT CHANGE:**
  - NEXT bits (greyed out on the right side)
  - Values for DRMUX and SR1MUX (aleady filled in)
  - Any microstates that are already filled out for you

  As a general rule of thumb, try not to break the spreadsheet :).
- Throughout this assignment, you will find the `CS2110-Reference-Sheet.pdf` to be very helpful. It can be found in the Canvas Files.
- For information on filling out **LDSR** and **STII**, see below.
- For more information on each of the control signals, see the Appendix at the end of this PDF.
- For the **Store** instructions, ensure that you prioritize loading the address we want to write to into the MAR over loading the value we want to write with into the MDR. The autograder is expecting that you load the address first and then the value.
- Now that you've developed some familiarity with the datapath, gotten a chance to "act as the micro-controller", and run the execute stage of instructions on the Manual LC-3, you can complete the final part of the project: writing the microcode for a number of micro-instructions on the LC-3!

- In the `microcode.xlsx` Excel spreadsheet, there exists a number of macro-states. Among them are FETCH, DECODE, and the EXECUTE stages for most instructions supported by the LC-3 (we've removed several macro-states related to trap and interrupt handling). For each macro-state, we've provided space for the micro-states which will make up that macro-state. For each micro-state, we've handled all of the logic related to transitioning to the next micro-state. We've also implemented a small subset of the macro-states in order to provide inspiration to you, our students (you're welcome).
- You should complete all the remaining macro-states by filling in their micro-states.
  - We recommend that you fill out the micro-states for FETCH first, so that you don't run into any problems with the autograders (several tests may fail).
  - If you notice that the output column to the right of the blacked-out columns (column AH) is showing artifacts like `#NAME?`, try opening the Excel spreadsheet in your Georgia Tech Office 365 Online Excel workspace. Sign in to Office 365 with Georgia Tech credentials, select 'Excel', and then choose 'Upload and open' to edit the excel sheet online.
- **LDSR**
  - Your fellow TAs are trying to create a new LC-3 instruction! We're calling it - **LDSR** - Load Summed Registers.
  - This instruction should access memory at the value specified by the sum of the two source registers (SR1, SR2) and save the result on DR.
  - DR = Mem[SR1 + SR2]
  - The 16-bit instruction is formatted in the following:
    [OPCODE | DR (3 bits) | SR1 (3 bits) | 000 (3 bits) | SR2 (3 bits)]
  - Note: You need to set the condition codes (CC) for this instruction.
- **STII**
  - The TAs got together again (yikes) and decided to make another function: **STII** - Store Indirect Indirect.
  - This instruction works similarly to STI but with one more trip to memory. To be more specific, we calculate an initial address by using the value of the PC and adding it to the offset. We access memory at this address and we use value we retrieve as an address to memory again. Then we use the new value we obtained as an address once again to access memory one last time. This final value we get from memory is the memory address where we will store the value of SR. The instruction is very similar to STI but with one more indirect address lookup on memory.
  - Mem[Mem[Mem[PC* + PCOffset9]]] = SR
  - The 16-bit instruction is formatted in the following:
    [OPCODE | SR1 (3 bits) | PCOffset9 (9 bits)]
  - Note: You do **NOT** need to set the condition codes (CC) for this instruction.

  We could really use your help! Fill out the micro-states for LDSR and STII in the microcode.xlsx excel sheet.

3.4.1 How to Test your Micro-code

At any time that you want to test your micro-code, you can export it from the .xlsx file and apply it to LC-3 hardware by following these steps. **IMPORTANT NOTE: Passing all of the tests provided does not guarantee that you have a functional datapath, any number of coincidences could cause you to get the correct output with incorrect functionality. As always, we reserve the right to grade with additional test cases.**

1. At the bottom of the `microcode.xlsx` file select the **output** tab.
2. Copy all of column D from row 1 through row 64
3. Paste the result into ROM.dat
4. Ensure that ROM.dat is in the same directory as ./cs2110docker.sh (or some subdirectory)
5. Ensure that ROM.dat is populated with the correct values for the autograder to work
6. You may run the autograder with the java -jar code, or follow the rest of the steps for manual testing
7. In Docker CircuitSim, open LC3.sim. Navigate to the 'Fsm' subcircuit. This circuit contains the micro-controller.
8. Right-click on the ROM and select "Edit contents."
9. Select "Load from file"
10. navigate to and select "ROM.dat." This will load the ROM.
11. Navigate to the 'LC-3' subcircuit.
12. You can now load a program into the RAM, following the instructions above in the Manual LC-3 sections.
13. To run the LC-3, you can manually click through the CLK signal or use 'Ctrl-K' to start or stop the automatic clock. After your program has stopped executing (you can tell when it's finished running because it will HALT and the datapath will stop changing).
14. Tests inside the `tests/` directory have a comment at the end of the .asm file which explains the system state after the end of the program's execution. To test whether the program acted correctly, go to the 'LC-3' circuit and double-click into the 'REG FILE' element **that is placed in the datapath**. Note: you **should not** just click into the 'REG FILE' subcircuit, as this will not properly load the state of the specific 'REG FILE' element that's built into the LC-3, just some generic REG FILE.

3.4.2 Tips, Tricks, Resources

This is all pretty crazy to take in at first. But it's not the end of the world if you make sure to use the resources available to you. Here are some options:

- CS 2110 Reference Sheet: Canvas → Files → CS2110-Reference-Sheet.pdf
- LC-3 Instructions Detail Sheet: `LC-3InstructionsDetail.pdf`, in this project.zip
- The Manual LC-3!
- Your friendly TAs (Office Hours, Piazza, etc.)
- Textbook
- Appendix on Datapath Control Signals in **this PDF**.

4 Autograder

To run the autograder, run

```
java -jar proj2-tester.jar
```

at a command prompt in the same directory as all your deliverable files (see next section for a list). This is the same autograder that Gradescope uses, but is much easier and faster to use. **The autograder should be run from the terminal.**

5 Deliverables

Please submit the follow files:

1. `latches.sim`
2. `LC3.sim`
3. `microcode.xlsx`

to Gradescope under the assignment "Project 2 - LC-3 Datapath".

Note: The autograder may not reflect your final grade on this assignment. We reserve the right to update the autograder as we see fit when grading.

6 Rules and Regulations

6.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit.
2. Please read the assignment in its entirety before asking questions.
3. Please start assignments early, and ask for help early. Do not email us a few hours before the assignment is due with questions.
4. If you find any problems with the assignment, it would be greatly appreciated if you reported them to the authors (names can be found at the top of the assignment). Announcements will be posted if the assignment changes.

6.2 Submission Conventions

1. In order to submit your assignment, submit the files individually to the Gradescope assignment.
2. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends.

6.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.
2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test it to see if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Gradescope. Under no circumstances whatsoever will we accept any email submissions of an assignment. Note: if you were granted an extension you will still turn in the assignment over Gradescope.
3. Projects turned in late receive partial credit within the first 48 hours, as defined in the syllabus. Between 0 and 24 hours late, you can receive a maximum score of 70%. Between 24 and 48 hours late, you can receive a maximum score of 50%. We will not accept projects turned in over 48 hours late.

4. You alone are responsible for submitting your project before the assignment is due; neither Canvas/Gradescope, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until the deadline.

6.4 Is collaboration allowed?

From the syllabus:

- You must submit an assignment or project as your own work. **No collaboration on answers is permitted. Absolutely no code or answers may be copied from others. Such copying is Academic Misconduct.**
- Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "submission of material that is wholly or substantially identical to that created or published by another person or persons").
- Publishing your assignments on public repositories, Github, etc, that is accessible to other students is unauthorized collaboration and thus Academic Misconduct.

6.5 Syllabus Excerpt on Academic Misconduct

The goal of all assignments in CS 2110 is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives an unfair advantage over students who do the work and follow the rules.

1. **As a Georgia Tech student, you have read and agreed to the Georgia Tech Academic Honor Code.** The Academic Honor Code defines Academic Misconduct as "***any act that does or could improperly distort Student grades or other Student academic records.***"
2. You must submit an assignment or project as your own work. **No collaboration on answers is permitted. Absolutely no code or answers may be copied from others. Such copying is Academic Misconduct**.
3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "*submission of material that is wholly or substantially identical to that created or published by another person or persons*").
4. Publishing your assignments on public repositories, github, etc, that is accessible to other students is unauthorized collaboration and thus Academic Misconduct.
5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.
6. Students suspected of Academic Misconduct are informed **at the end of the semester**. Suspects receive an *Incomplete* final grade until the issue is resolved.
7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.
8. Submitting an assignment with code or text from an AI assistant (e.g., ChatGPT) is academic misconduct.
9. **If you are not sure about any aspect of this policy, please ask your lecturer**.

Using AI Assistants

Anything you did not write in your assignment will be treated as an academic misconduct case. If you are unsure where the line is between collaborating with AI and copying AI, we recommend the following heuristics:

Heuristic 1: Never hit "Copy" within your conversation with an AI assistant. You can copy your own work into your own conversation, but do not copy anything from the conversation back into your assignment. Instead, use your interaction with the AI assistant as a learning experience, then let your assignment reflect your improved understanding.

Heuristic 2: Do not have your assignment and the AI agent open at the same time. Similar to the above, use your conversation with the AI as a learning experience, then close the interaction down, open your assignment, and let your assignment reflect your revised knowledge. This heuristic includes avoiding using AI directly integrated into your composition environment: just as you should not let a classmate write content or code directly into your submission, so also you should avoid using tools that directly add content to your submission.

Deviating from these heuristics does not automatically qualify as academic misconduct; however, following these heuristics essentially guarantees your collaboration will not cross the line into misconduct.

7 Appendix: Datapath Control Signals

The microcontroller of the LC-3 has 52 bits of output signals to control program execution on the datapath. In this assignment, we will focus on 20 of them. There are four categories of signals we need to worry about:

1. **Load Signals** Each register has a load signal associated with it. When the load signal of a register is high (1), the value of the register will update to its input at the uptick of the clock.

   - **LD.MAR** The MAR (Memory Address Register) register holds the address of data to be read from, or written to, memory. This signal loads the MAR with the value from the bus, which should be the address of data to be read in a load signal (LD, LDR, LDI), or data to be written to in a store signal (ST, STR, STI). This address should be come from either the PC (For FETCH) or from the MARMUX (for all other memory access instructions).
   - **LD.MDR** The MDR (Memory Data Register) holds the data either read from or to be written to memory. The MDRMUX that selects between the bus (For store instructions - when data from a register is to be written to memory) and memory out (For load instructions - when data read from the memory is to be written to a register). When LD.MAR is high the MDR loads whichever the MDRMUX outputs.
   - **LD.IR** The IR (Instruction Register) holds the currently executing instruction (Contrast this to the PC, which holds the *address* of the next instruction to be executed, the IR holds the literal 16-bit assembled instruction which is fetched from memory at the address in the PC). The IR is only written to during the FETCH stage, so that is the only time LD.IR should be used.
   - **LD.REG** LD.REG is used for writing to the general purpose registers. When LD.REG is high (1), the DR register will load the value on the bus. In general, this signal should be active in the last state of any instruction that writes to a destination register.
   - **LD.CC** The CC (Condition Code) register is used for conditional (branching) statements. The CC itself is a three bit register, with one bit for each of (negative, zero, positive). Branching instructions (BR) use the value of the CC to determine if a branch should be taken (i.e. BRn means 'branch if cc == negative'). Because of this, the CC should always reflect the result of the previous instruction. The 'result' of an instruction is generally whatever is written to a register in the last cycle. This means that LD.CC should be closely related to LD.REG as those loads are done in the same cycle (Because the result is already on the bus to load into the register file, we can also load it into the CC for free.) **Note that not all instructions should set the condition codes.** Generally, things like load instructions (LD, LDR, LDI) and all arithmetic instructions (ADD, AND, NOT) should set the CC, while things like LEA, branching, and store instructions don't really have a 'result' so they do not set the CC. LEA does not set the condition codes.
   - **LD.PC** The PC holds the address of the next instruction to be executed. Therefore, the value in the PC defines the control flow of the program. By default, the PC should be incremented by 1 during every FETCH stage. Branching and Jumping instructions work by setting the PC to some other value which causes the execution to jump to another point in the program. This signal should be high whenever the value of the PC should be changed, namely, in the FETCH stage and all branching and jumping instructions (There is a PCMUX which chooses the input of the PC to either increment the PC for fetch, read from the bus, or the ADDR calculation circuit – ADDR1MUX + ADDR2MUX).

2. **Gate Signals** All of the components on the datapath are connected by the **bus.** The bus is a single wire which any component can read from, and any component can assert to. However, we already know what happens when we try to assert two different signals to the same wire (Short circuits, fire, ensuing chaos and certain doom). Enter the **Tri-State Buffer.** The tri-state buffer works similarly to a transistor. It has an **input**, **output**, and **enable** bit, analogous to the source, drain and gate of the transistor. If the enable bit is high (1), then the output of the tri-state buffer will be whatever is connected to its input. If the enable bit is low (0), then the output will have no value, so it won't ever cause a short circuit. So, we use tri-state buffers to connect each component to the datapath. That way, as long as only one tri-state buffer is enabled per clock cycle, we can move anything on the datapath and don't have to worry about short circuits! However, this also means that we can only move one thing on the bus at a time. **This is very important. It also means it is your responsibility to make sure that only one tri-state buffer on the bus is ever enabled in a given clock cycle.**

    - **GatePC** This signal asserts the value of the PC to the bus. This should be used any time you want to load the PC into another register. Namely, this could be the MAR (for fetch), or R7 for saving the PC as a return address in branching and jumping instructions.
    - **GateMDR** This signal asserts the value of the MDR to the bus. In this case, the MDR should hold data read from memory, so it is being asserted to the bus to be saved to another register. Namely, this should be used to load the value of the MDR into the IR for FETCH, into a general purpose register for load instructions (LD, LDR, LDI), or back into the MAR for indirect memory access instructions (LDI, STI).
    - **GateALU** This signal asserts the output of the ALU to the bus. Remember, the ALU can output 8 different operations: A + B, A & B, ¬A, PASS A. Clearly, this signal should be active for the arithmetic instructions that use any operation that alters the value of A or B. The GateALU should also be active any time the value of a general purpose register should be written somewhere else (i.e. for storing instructions), which is when the PASS A option would be used (PASS A directly asserts the value of SR1 onto the bus).
    - **GateMARMUX** This signal asserts the output of the MARMUX onto the bus. Almost always, the value asserted onto the bus represents an address to be loaded into the MAR for loading and storing instructions (or directly into a destination register for LEA).

3. **MUX Signals** These signals have a range of possible values, and this range of values can differ based on the number of inputs to a given MUX (some have 2 inputs, others have 3 or 4).

    - **PCMUX** The PC has 3 options every time it is updated. During every FETCH, the PC is incremented by 1, and during branching and jumping instructions, the PC can be loaded either from the ADDR calculation circuit (ADDR1MUX + ADDR2MUX, for most branching/jumping), or read from the bus (rarely).
    - **ADDR1MUX** For memory address calculation (For data or instructions), all addresses take the form of a base register (which is usually the PC, but sometimes a general purpose register from the register file) which is added to the sign extension of some number of bits from the IR. The ADDR1MUX chooses what the base register should be, either the PC (for most instructions), or a general purpose base register (for LDR, STR, JSRR, and JMP).
    - **ADDR2MUX** For memory address calculation (For data or instructions), all addresses take the form of a base register (which is usually the PC, but sometimes a general purpose register from the register file) which is added to the sign extension of some number of bits from the IR. The ADDR2MUX chooses what that offset should be. Different instructions can allocate different numbers of bits for their immediate offset, with some having 6, 9, or 11. Each of these, IR[5:0], IR[8:0], IR[10:0], as well as a hardcoded 0 option, is sign extended to 16 bits to be added to the base register. ADDR2MUX chooses which of these is passed through.
    - **MARMUX** Most memory calculations will come through the MARMUX. The MARMUX has 2 inputs, one for the ADDR calculation circuit (ADDR1MUX + ADDR2MUX), and one to zero

extend the lower eight bits of the instruction register (ZEXT(IR[7:0])). The later option is only used for TRAP instructions, which are outside the scope of this project, so you only need to worry about the former.

- **ALUK** The ALUK selects which operation the ALU should output, from A + B, A & B, ¬A, PASS A. The first three are used for the arithmetic instruction (ADD, AND, NOT), and the PASS A operation directly outputs SR1 to the bus. This last option is used whenever the value of a general purpose register needs to be written somewhere else (Like store instructions.)

4. **Memory Signals** There are two signals that are used for memory access, MEM.EN and R.W. These control the behavior of the memory for read and write operations.

   - **MEM.EN** The MEM.EN signal will be high whenever the memory is accessed in any way, whether it is for reading or writing.
   - **R.W** R.W, or Read.Write is used to distinguish between memory operations that *read from* the memory and memory operations that *write to* the memory. Clearly, operations that are writing to memory (store instructions) should have R.W set to Write (1), while memory operations that read data already in memory should have R.W set to Read (0).

7.1 MUX Values

We'll take a second to clarify which selection codes correspond to which inputs in the 8 MUXes we've implemented on the LC-3 that you need to worry about.

- **MARMUX** - Memory Address Register Mux
  0. ZEXT (Zero-extend) input.
  1. ADDR (address adder) input.
- **PCMUX** - Program Counter Mux
  00. PC+1 input.
  01. ADDR (address adder) input.
  10. BUS input.
- **DRMUX** - Destination Register Mux (values given for you)
  0. IR[11:9] input.
  1. Constant 0b111 input.
- **SR1MUX** - Source Register 1 Mux (values given for you)
  0. IR[11:9] input.
  1. IR[8:6] input.
- **SR2MUX** - Source Register 2 Mux (determined by IR[5]) - don't worry about this
  0. SR2 input.
  1. SEXT[4:0] (sign extend) input.
- **ADDR1MUX** - Address Adder Input 1 MUX
  0. PC input.
  1. SR1 input.
- **ADDR2MUX** - Address Adder Input 2 MUX
  00. Constant 0x0000 input.
  01. SEXT[5:0] input.
  10. SEXT[8:0] input.
  11. SEXT[10:0] input.
- **MDRMUX** - MDR Input MUX - don't worry about this
  Note: The selector bit for this mux should be the MIO.EN / MEM.EN signal
  0. Bus.
  1. Memory data output.
- **ALU** - Bonus Entry (not exactly a MUX)
  00. ADD (A + B)
  01. AND (A & B)
  10. NOT (A)
  11. PASS (just returns A)