

Homework 4 - LC-3 Datapath

Jason Ng, Junchen Lu, Jayanee Venkat, Justin Hinckley, Srishti Grover, Udit Subramanya

Spring 2022

Contents

1 Overview	3
1.1 Purpose	3
1.2 Task	3
1.3 Criteria	3
2 Introduction	4
2.1 The LC-3's Microcontroller	4
2.2 The ROM	5
2.3 Files Provided	6
2.4 Tasks You Must Complete	6
3 Using the CircuitSim File	6
4 Writing the Microcode	6
4.1 General Instructions	6
4.2 Filling Out The Microcode Spreadsheet	7
4.3 LDSR	7
4.4 Tips, Tricks, and Recommendations	8
4.5 How to Test your Microcode	8
5 Checking Your Work	9
5.1 Using the Manual LC-3	9
5.1.1 Resources	10
6 Setup	11
7 Deliverables	11
8 Demos	11
9 Rules and Regulations	12

9.1	General Rules	12
9.2	Submission Conventions	12
9.3	Submission Guidelines	12
9.4	Syllabus Excerpt on Academic Misconduct	12
9.5	Is collaboration allowed?	13
10	Appendix: Datapath Control Signals	14
10.1	MUX Values	17

1 Overview

1.1 Purpose

The purpose of this assignment is for you to understand the datapath and the state machine of the LC-3 processor (as presented in the Patt textbook, and Lecture and Lab). You will implement components on the LC-3 datapath and complete the microcode for multiple LC-3 machine instructions.

Note: The LC-3 implementation in this assignment is simplified, and does not include all the features or components of the full LC-3 from the Patt textbook or Lecture or Lab.

1.2 Task

Please read this entire document before starting. It includes many details of the assignment, as well as tips, tricks, and references that will help you.

- You will complete the microcode for many of the LC-3 machine code instructions as well as a new instruction LDSR. You will enter the control signals into the microcode.xlsx spreadsheet we have provided. You may test your microcode, using the ROM.dat file and importing data into FSM subcircuit of the LC3.sim circuitsim file.

We have also provided a local checker, and the autograder on gradescope.

1.3 Criteria

Your grade on this assignment has two parts: the submissions (LC3.sim and microcode.xlsx), and a demo (demonstration of your solution and knowledge to a TA).

- The submissions are worth a total of 50% of your homework grade. As with prior assignments, the autograder will give you an approximation of your final grade on this portion of the assignment.
- You will also demonstrate your solution, and understanding of the concepts, to one of your TAs for 50% of your homework grade. The demos will happen after the assignment is due, and we will announce details closer to that time. **You are responsible for signing up for and then attending a demo slot at that time. Logistics and details of demos will be posted on Canvas before they are held.**

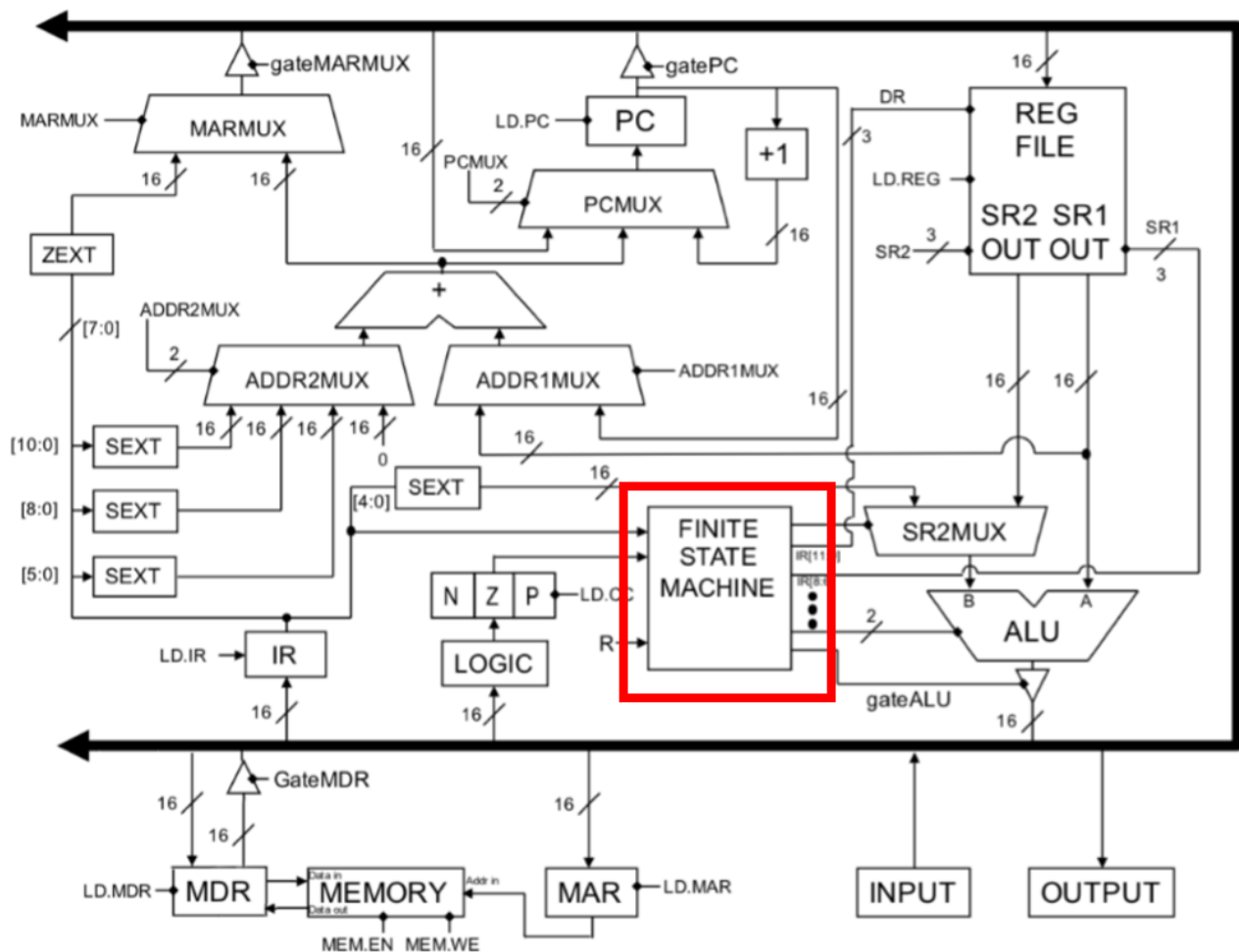
2 Introduction

Hello and welcome to Homework 4! In this homework, you'll develop familiarity with the functionality of the microcontroller and components on the LC-3 Datapath that we've covered in class.

Please read through the entire document before starting and acquaint yourself with the rules and submission policies stated in the syllabus. Often times 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 the Ed Discussion page or come visit us in office hours.

2.1 The LC-3's Microcontroller

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 we've mostly referred to as a "black-box" in the past is the microcontroller. 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 microcontroller actually work? The microcontroller uses the idea of states to keep track of which datapath components to use and when. In lecture, we introduced the notion of macro-states and

micro-states. A micro-state is an individual state of the finite state machine that specifies control signals that should be asserted in a single clock cycle. The micro-states are *components* of a slightly larger sequence of states known as macro-states. FETCH is a macro-state, and each of the execute stages of LC-3 instructions is also a macro-state. Each of the macro-states will require between 1-5 micro-states to complete, depending on the complexity of the instruction. The microcontroller, shown above, is a finite state machine. It has 59 possible states (holy dancing crab!), and because it is implemented in the “binary reduced” style, it needs 6 bits to store all its possible states. It also has 49 output bits of output flags, including 10 which are used to determine the next state and 39 which 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 with combinational logic.

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.

2.2 The ROM

ROM stands for Read-only Memory and as its name suggests it is a piece of memory that can only be read. Each memory address will hold an entry that represents a micro-state. This entry holds two crucial pieces of information: 1) what signals to assert on the datapath for a specific micro-state and 2) what the next state should be. With the ROM, we can hardcode these information into a binary string. We can now read from the ROM with the appropriate memory address to obtain the binary string for a specific micro-state.

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)	Gate PC (1)	Gate MDR (1)	Gate ALU (1)	Gate MARMUX (1)	PCMUX (2)	DRMUX (1)	SRI MUX (1)	ADDR1 MUX (1)	ADDR2 MUX (2)	MARMUX (1)	ALUX (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	1 0 0 0 0 0
ADD																				
ADD1												0	1							0 0 1 0 0 0
AND																				
AND1												0	1							0 0 1 0 0 0
BR																				

A ROM entry is basically a long binary string. The last few bits of it cover the transition to the next state (these are essentially the address of the binary string for the next state in the ROM)—you don’t need to worry about this at all during this homework, so we’ve covered it in dark grey on the right. Do NOT modify the NEXT bits, or stuff will break. Each of the other bits corresponds to a signal asserted onto the datapath during that clock cycle. For this homework, we only require that you implement 19 of the signals asserted onto the datapath (notice that three of these are 2-bit signals).

In this homework, we’ll actually write the “microcode” which allows the microcontroller to function. We’ve simplified and removed a number of micro-states which aren’t directly a part of the LC-3’s main instructions. There are only 36 micro-states in this homework. We’ve also given you the microcode for JSR, JSRR, BR, and HALT. Your task is to fill in the rest and finish the LC-3 microcode!

2.3 Files Provided

- `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. **Once again, you do not need to modify any 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 and do not edit the output sheet directly.**
- `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 microcode.
- `hw4-tester.jar` - a local tester you can use to verify your microcode. This tester is also available on Gradescope (where it will be a part of your grade).
- `LC-3InstructionsDetail.pdf` - a PDF with descriptions and pseudocode for each instruction.

2.4 Tasks You Must Complete

1. Complete the microcode in `microcode.xlsx`, in the microcode sheet.

3 Using the CircuitSim File

After you finish Part 1, you will have a fully functional LC-3 datapath. The LC-3 datapath you will be working with for this homework, as contained in the `LC3.sim` file, is a complete but simpler version of the LC-3 datapath you might encounter in your textbook’s appendix C. This is due to the lack of IO related components in the LC-3 datapath that you will be working on as these components and how they are utilized are beyond the scope of this particular homework, though they will be encountered in a later part of this course. As such, you can ignore the components not present in the `LC3.sim` file as they appear in the textbook. We have provided you with this built LC-3 datapath for your own understanding, but **you do not need to modify anything in this file.**

4 Writing the Microcode

4.1 General Instructions

- Now that you’ve developed some familiarity with the datapath (and perhaps tried to “act as the microcontroller” to execute instructions on the Manual LC-3 subcircuit - see the “checking your work” section for information on this), you can complete Part 2: writing the microcode for a number of micro-instructions on the LC-3!
- In `microcode.xlsx` Excel document, **microcode** sheet, there exist a number of macro-states. Among them are `FETCH` 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, and for each micro-state, we’ve handled all of the logic related to transitioning to the next micro-state. We’ve also implemented a pretty small subset of the macro-states in order to provide inspiration to you, our students.
- You should complete all the remaining macro-states by filling in their micro-states.

- 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.

4.2 Filling Out The Microcode Spreadsheet

	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)	ALUX (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	1 0 0 0 0 0
ADD																				
ADD1												0	1							0 0 1 0 0 0
AND																				
AND1												0	1							0 0 1 0 0 0
BR																				

The microcontroller is the "brain" of the LC-3 and it performs operations by manipulating the control signals within the datapath. The `microcode.xlsx` spreadsheet is an abstraction of what the microcontroller does in each micro-state.

In the `microcode.xlsx` spreadsheet, the rows correspond to micro-states and the columns correspond to control signals on the LC-3 datapath. What you need to do is fill in the spreadsheet based on the control signals that are set for each micro-state. **You should check out the Appendix and subcircuits in LC3.sim for specifications on each control signal.**

For example, for the micro-state BR1, the control signals involved are:

- LD.PC
- PCMUX = 01
- ADDR2MUX = 10

You do not need to worry about the control signals of DRMUX and SR1MUX. We have already filled them in for you. You should not touch the NEXT state bits which are covered in dark grey and changing them will result in failure of autograder tests.

4.3 LDSR

Your fellow TAs are trying to create a new LC-3 instruction! We're calling it - **LDSR** - Load Summed Registers. This instruction should add the values of two source registers together, go to memory and use this sum as an address, look up the data that is contained within this address, and finally load it into a destination register.

The 16-bit instruction is formatted in the following:

[OPCODE | DR (3 bits) | SR1 (3 bits) | 000 | SR2 (3 bits)]

Note: Since this is a load instruction, you **need** to set the condition code (CC) for this macro-state. We could really use your help! Fill out the micro-states for LDSR in the microcode.xlsx excel sheet.

4.4 Tips, Tricks, and Recommendations

Firstly, note that the LC-3 in this homework is not exactly the same as the LC-3 presented in the book. The processor in Homework 4 is an alternate implementation of a somewhat simplified subset of the LC-3 ISA. Here are some notable differences between the two:

- The instruction set used by the LC-3 in Homework 4 does not have a TRAP instruction.
- In the regular LC-3, halting is handled by a trap. In Homework 4, this opcode (1111) is taken up by a single "HALT" instruction that just loops infinitely. The LC-3 in Homework 4 does not have circuitry for handling interrupts/traps/exceptions/etc.
- The state machine used by the LC-3 in Homework 4 numbers its states differently from those in the book, and the microcontroller signals are a little different. The LC-3 in Homework 4 assumes that memory reads/writes take a single clock cycle like in CircuitSim, while in practice they take much longer.
- The LC-3 in Homework 4 does not include I/O.

Secondly, the microcode is given in a somewhat confusing order. We recommend completing the microcode in the following order to make testing go smoothly:

- FETCH (you can't execute any instructions if you can't fetch any instructions!)
- ADD
- AND
- NOT
- LEA
- JMP
- LD
- ST
- LDR
- STR
- LDI
- STI
- LDSR

4.5 How to Test your Microcode

At any time that you want to test your microcode, 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. Go to the output sheet of `microcode.xlsx`
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. In Docker CircuitSim, open `LC3.sim`. Navigate to the ‘Fsm’ subcircuit. This circuit contains the microcontroller.
6. Right-click on the ROM and select “Edit contents.”
7. Select “Load from file”
8. navigate to and select “`ROM.dat`.” This will load the ROM.
9. Navigate to the ‘LC-3’ subcircuit.
10. You can now load a program into the RAM, following the instructions above in the Manual LC-3 sections.
11. 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).
12. 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.

5 Checking Your Work

Once complete, update your `ROM.dat` as described above and run the autograder

```
java -jar hw04-tester.jar
```

inside the Docker container in the command prompt in the same directory as your `LC3.sim` file. **NOTE: the autograder uses `LC3.sim` to grade, so please don’t make any changes to it as it may break the local autograder.**

If all of the relevant tests pass, you’ve completed this part of the homework. Congrats!

If all of the relevant tests do not pass, and you would like to double check your own understanding of how your microcode interacts with the LC-3 Datapath, you can use the “Manual LC-3” subcircuit provided in the homework file. **NOTE: This is an ungraded part of the homework that you do not need to use to get full points on this homework. This is supposed to be used as a debugging tool to help you understand how to complete the homework.**

5.1 Using the Manual LC-3

- In lecture and lab we have covered the signals in the datapath and how they are used 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 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 homework (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 over the course of the next two 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 homework *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. We can’t guarantee that we’ll be able to help with these test cases in office hours, though.

NOTE: The above section on the manual LC-3 is not needed to get full points on this homework. This is supposed to be used as only a debugging tool to help you understand how to complete the homework.

5.1.1 Resources

This can all be pretty daunting to read and understand at first. But it’s not the end of the world so do not panic, carry a towel and make sure to use the resources available to you. Here are some options:

- LC-3 Datapath Diagram and ISA Quick Sheet: Canvas → Files → LC-3 Resources → LC3ReferenceSheet.pdf
- LC-3 Instructions Detail Sheet: LC-3InstructionsDetail.pdf, in this homework.zip
- The Manual LC-3!
- Your friendly TAs (Office Hours, Ed Discussion, etc.)
- Textbook
- Appendix on Datapath Control Signals in **this PDF**.

6 Setup

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

In order to use CircuitSim, you must have Docker up and running. If you do not have Docker, follow the instructions laid out in the installation guide found under Canvas → Files → Docker. **Make sure you are using the updated docker container!! The docker-script should automatically do this when you run it. Check by ensuring that the version of CircuitSim in the docker container is version 1.8.2**

CircuitSim comes pre-installed on the Docker image, and should be accessible via the desktop. Please only use the CircuitSim through Docker to build your circuits as it is the correct version. **CircuitSim downloaded from elsewhere may not be compatible with our grader. You have been warned.**

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 do not move or rename any of the provided circuits/elements. The only things you need to modify is RAM/ROM contents and using the pins/buttons provided while debugging using the provided LC3.sim file.

7 Deliverables

Please submit the follow files:

1. microcode.xlsx

to Gradescope under the assignment “Homework 4”. **The Gradescope autograder will check your work on the microcode file against a working and correct LC3.sim file, not your own submission of the LC3.sim file.**

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

8 Demos

This homework will be demoed. The demos will be ten minutes long and will occur **IN PERSON**. Stay tuned for details as the due date approaches.

- Sign up for a demo time slot via Canvas **before** the beginning of the first demo slot. This is the only way you can ensure you will have a slot.
- If you cannot attend any of the predetermined demo time slots, e-mail the head TA Shawn Wahi (shawn.wahi@gatech.edu) **before** the week of demos.
- If you know you are going to miss your demo, you may cancel your slot on Canvas with no penalty, as long as you cancel 24 hours in advance. However, you are **not** guaranteed another time slot. If you cancel within 24 hours of your demo, it will be counted as a missed demo.
- Your overall homework score will be $((\text{homework_score} * 0.5) + (\text{demo_score} * 0.5))$, meaning if you received a 90% on your homework, but a 30% on the demo, you would receive an overall score of 60%. **If you miss your demo you will not receive any of these points, and the maximum you can receive on the homework is 50%.**
- You will be able to make up one of your missed demos at the end of the semester for half credit.

9 Rules and Regulations

9.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. As such, please start assignments early, and ask for help early. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.
2. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

9.2 Submission Conventions

1. 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. You must submit all files listed in the **Deliverables** section individually to Gradescope as separate files.

9.3 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 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 Canvas/Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas/Gradescope.

9.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

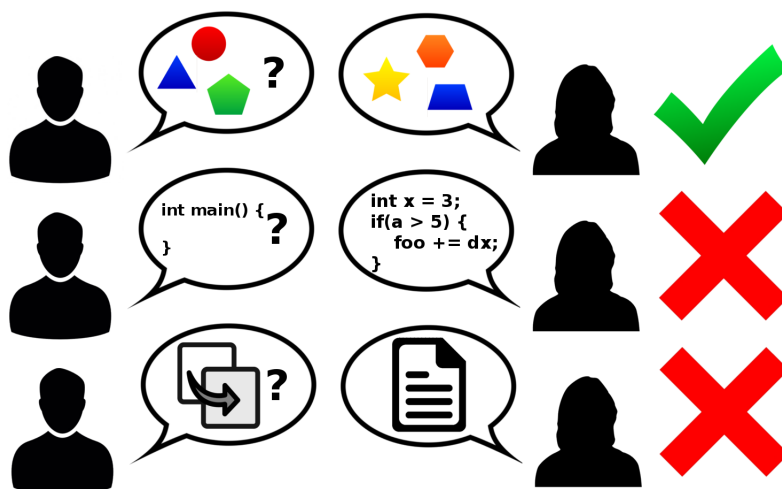
Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use [github.gatech.edu](https://github.com)

9.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own.



10 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 branching and store instructions don't really have a 'result' so they do not set the CC.
 - **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 4 different operations: $A + B$, $A \& B$, A , PASS A . Clearly, this signal should be active for the arithmetic instructions that use the first 3 operations (ADD, AND, NOT). 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).
 - **DRMUX** For most instructions, the DR (Destination Register) is explicitly defined in the instruction. Sometimes, however, the DR is implicitly set to R7 (as R7 is always used as the return address for branching instructions.) The DRMUX can set the DR to either IR[11:9] (the 3 bits of the instruction register used to encode the DR for most instructions), or hardcoded R7 for the branching instructions that save a return address (the PC) in R7.
 - **SR1MUX** For most instructions, the first Source Register (SR1) is encoded at IR[8:6] (bits 6, 7, and 8 of the instruction). However, some instructions have their source/base register located higher in the instruction at IR[11:9] (Namely, storing instructions that need space lower in the instruction for an immediate offset.)
 - **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 homework, 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).

10.1 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
 - Note: The selector bit for this mux should be the MIO.EN / MEM.EN signal**
 - 0. Bus.
 - 1. Memory data output.