CS 2110 Timed Lab 2: LC-3 Datapath Tracing

Richard Zhang, Dayne Bergman, Gal Ovadia, Ava Gavin

v1.1

Contents

Please take the time to read the entire document before starting the assignment. It is your responsibility to follow the instructions and rules.

1 Timed Lab Rules - Please Read

1. You are allowed to submit this timed lab starting from the moment the assignment is released, until **75 minutes** after your lab period—no more and no less (unless you have accommodations or special circumstances that have already been discussed with your professor). Gradescope submissions will remain open but **you are not allowed to submit after the 75 minute period is over.**
   - You may submit to Gradescope as many times as you wish within your allotted test time period. We will grade your last submission.
   - **Submitting or resubmitting the assignment after this is a violation of the honor code—doing so will automatically incur a zero on the assignment and might be referred to the Office of Student Integrity.**
2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. **The information provided in this Timed Lab pdf takes precedence.** If in doubt, please make sure to indicate any conflicting information to your TAs.
3. Resources you are allowed to use during the timed lab:
   - Assignment files
   - Previous homework and lab submissions (this includes homework PDFs)
   - Class Notes (Open Net, Open Book)
   - Other resources you find on the internet.
   - Your mind!
4. Resources you are **NOT** allowed to use:
   - Email/messaging
   - Contact in any form with any other person besides TAs
   - Anything written with the use of AI-based code completion software such as GitHub Copilot or ChatGPT.
   - Any assignment or submission from this course from another student.

2 Overview

You will complete the microcode for seven brand-new LC-3 instructions. For each instruction you should fill out the corresponding rows in the `tl02microcode.xlsx` sheet. You need to implement **INC, LDPC, LJSR, MULT8, LDAR, STPCI, and CCH**, as described below. Note, when talking about registers, DR, SR1, and SR2, they can either refer to the 3 bits that indicate WHICH register (we have 8 total), or refer to the 16 bit value contained within a register. In the 16 bit instruction layouts, SR1 would refer to the 3 bits (as seen by the bit range 9:11 for example). In a line like SR1 <= SR1 + sext(imm9), this in plain English means: SR1 takes the value of SR1's previous value plus a 9 bit offset number.

3 Hints

- Fill in all unused control signals with 0.
- If you are lost, try seeing how each instruction compares to those that you completed in the HW4, and transform your HW4 instructions into the ones listed above. This does not work for all instructions, but will work for a few.
- Do not forget to set the condition codes for the instructions that require it!

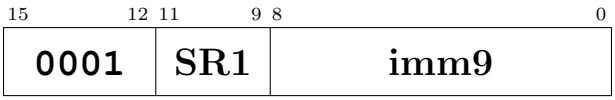
4 Instructions

4.1 Writing the Microcode

Use the provided `tl02microcode.xlsx` file and fill in the control signals for each instruction. **Do not edit any of the values that have already been provided.**

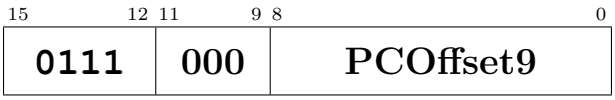
Also, please refer to Section 8.1 if you need to know which MUX selector value corresponds to which input, or reference the LC3.sim file. Do NOT use the LC3 reference sheet to guess the order of pins.

INC (Increment) 13 points



- This instruction increments the value in `SR1` by an `imm9` (`PCOffset9`) value. This instruction sets the condition codes.
- `SR1 <= SR1 + sext(imm9)`
- Examples:
  `INC R0, 15 ;` `R0 <- R0 + 15`
  `INC R3, -23 ;` `R3 <- R3 + (-23)`

LDPC (Load PC) 15 points

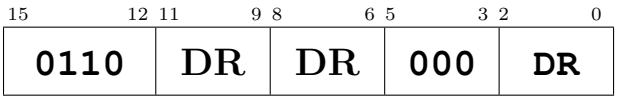


- This instruction sets the value of the PC to the value in memory at PC + PCOffset9. This instruction does **not** set the condition codes.
- `PC = MEM[PC + PCOffset9]`
- Examples:
  `LDPC 15 ;` `PC <- MEM[PC + 15]`
  `LDPC -3 ;` `PC <- MEM[PC - 3]`

LJSR (Lazy Jump to Subroutine) 13 points

- This instruction sets the program counter (`PC`) to the address designated by `PC + PCOffset11`. This instruction does **not** set the condition codes.
- `PC <= PC + PCOffset11`
- Examples:
  `LJSR 200 ;` `PC <- PC + 200`
  `LJSR -157 ;` `PC <- PC + (-157)`

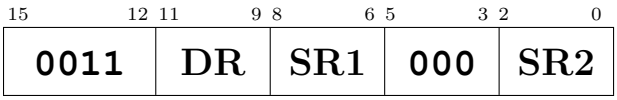
MULT8 (Multiply by 8) 15 points



- This instruction assigns `DR` the value of `DR` (itself) multiplied by 8. Thinking about how we accomplish this with bitwise operations, multiplying by 8 is equivalent in binary to bitshifting left by 3. This instruction sets the condition codes.
- `DR <= DR * 8`
- `DR = DR << 3`

- Examples:
  MULT8 R0 ; `R0 <- R0 * 8`
  MULT8 R7 ; `R7 <- R7 << 3`

LDAR (Load Added Registers) 16 points

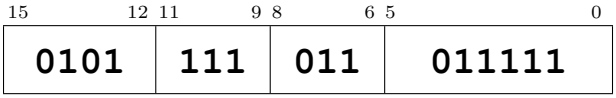


- This instruction assigns `DR` the value in memory at the address designated by `SR1 + SR2`. This instruction sets the condition codes.
- `DR <= mem[SR1 + SR2]`
- Examples:
  LDAR R1, R2, R3 ; `R1 <- mem[R2 + R3]`
  LDAR R4, R4, R4 ; `R4 <- mem[R4 + R4]`

STPCI (Store PC Indirect) 12 points

- This instruction stores the program counter (`PC`) to the address designated by `mem[PC + PCOffset11]`. This instruction does **not** set the condition codes.
- `mem[mem[PC + PCOffset11]] <= PC`
- Examples:
  STPCI 23 ; `mem[mem[PC + 23] <- PC`
  STPCI -14 ; `mem[mem[PC + (-14)] <- PC`

CCH (Capitalize Character) 15 points



- This instruction takes an ASCII character in `R3` (as specified by the `SR1` bits) capitalizes this character by and-ing it with the provided bit mask, and stores the new character in `R7` (as specified by the `DR` bits). This instruction takes in no arguments. This instruction does **not** set the condition codes.
- **IMPORTANT CLARIFICATION/HINT: We have provided you with the mask needed to implement this instruction in the bits [7:0] of the instruction. Consider which component(s) use this specific range of bits.**
- `R7 <= R3 (capitalized)`
- Examples:
  CCH ; `R7 <- 'A', R3 = 'a' initially`
  CCH ; `R7 <- 'D', R3 = 'd' initially`

5 Common Errors

Use the autograder's output to determine where you have gone wrong. The names of the tests you fail should usually (but not always) point you in the right direction.

Some common errors and their remedies:

1. Make sure that your MUX control signals are correct. For reference, read section 8.1.
2. If you think you have accidentally edited one of the given values in `tl02microcode.xlsx`, please raise your hand and a TA will come fix it for you.
3. If passing all tests except the condition codes test, make sure you are setting the condition codes properly for that instruction. The description for each instruction will tell you whether or not it sets the condition codes.

5.1 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 `tl02microcode.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 .dat file for one of the tests into the RAM, following the instructions in the Manual LC-3 sections (Section 9).
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.

6 Autograder/Grading

To run the autograder locally, navigate to the directory containing your timed lab and the tester and run the following command:

```
java -jar tl02-tester.jar
```

Make sure to run the local autograder out of your docker container and have your most recent microcode in the `ROM.dat` file. **You must update `ROM.dat` for the local autograder to work.** The local autograder cannot read your Excel file directly. For instructions on how to put your microcode into the `ROM.dat` file, see Section 5.1

The output of the autograder is an approximation of your score on this timed lab, so that you can evaluate how much of the assignment expectations your submission fulfills. We reserve the right to change the autograder test cases before finalizing grades. Timed labs are not manually graded or reviewed by the TAs, and there will not be opportunities for partial credit beyond the autograder. Submissions that are not runnable will receive a 0.

7 Deliverables

Please upload the following files to **Gradescope**:

1. `tl02microcode.xlsx`

Note: if you were granted an extension, you will still turn in the assignment over Gradescope.

Download and test your submission to make sure you submitted the right files.

8 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 timed lab, 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).

8.1 MUX Values

We'll take a second to clarify which selection codes correspond to which inputs in the nine 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

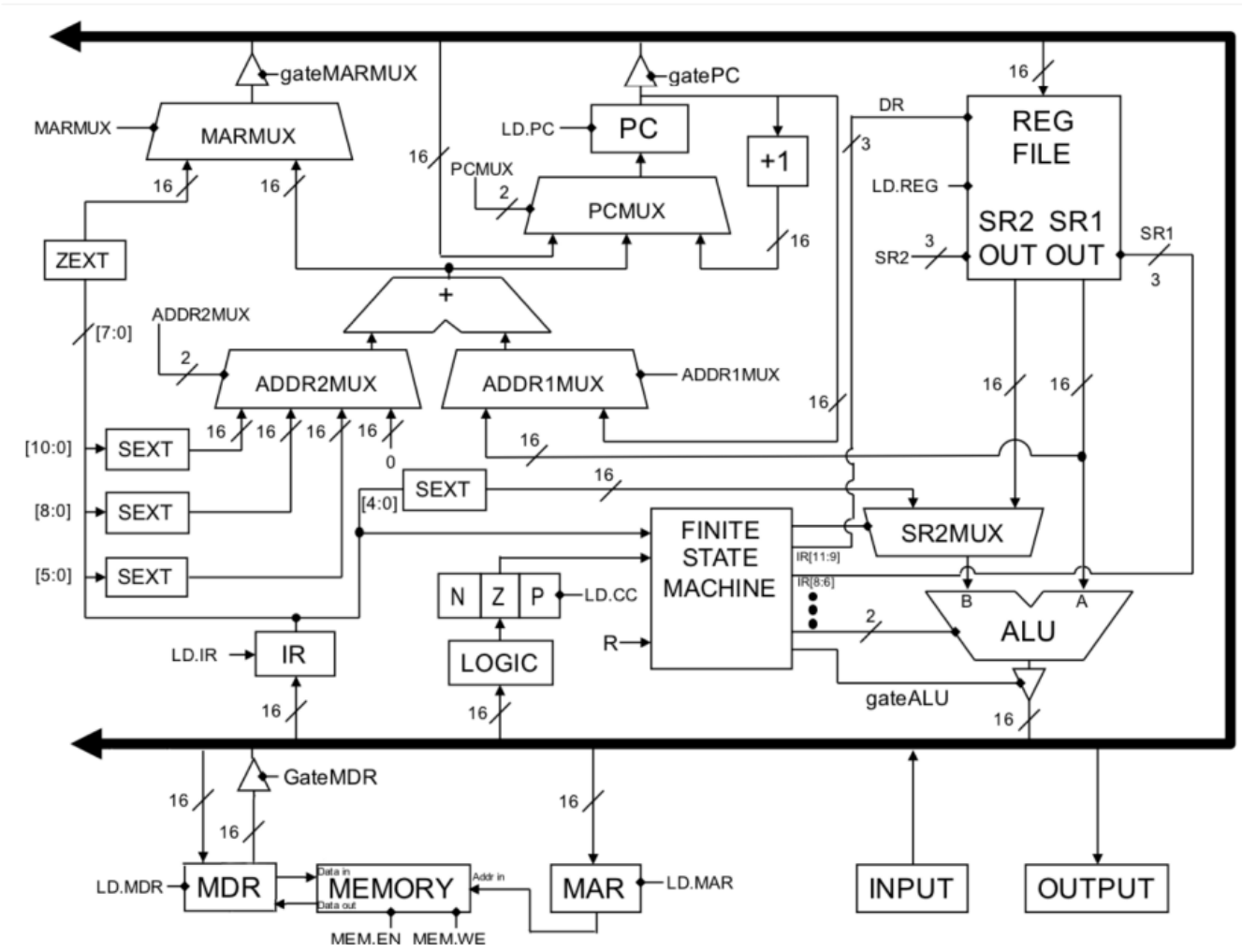
  Note: This is handled for you if you properly use MEM.EN. Detailed explanation: The selector bit for this mux should be the MIO.EN / MEM.EN signal, i.e. you don't need to worry about it, because whenever memory enable is on, it takes from the mem, and when it is off, it takes from the bus, which is expected behavior
  0. Bus.
  1. Memory data output.
- **ALUK - ALU Control (selector) MUX**
  00. ADD ($out = A + B$)
  01. AND ($out = A \,\&\, B$)
  10. NOT ($out = \sim A$)
  11. PASS ($out = A$)

9 Appendix: 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 timed lab (ex: mult8.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.

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

10 Appendix: LC-3 Datapath diagram



NOTE: As this is an open note assignment, you may find it useful to also simply use the LC3 Reference sheet. However, please follow the LC3.sim file and this pdf for pin numbering, and NOT the order of items listed in the LC3 reference sheet tables.