CS 2200: Computer Systems and Networks

Prof. Forsyth

Project 2 - Interrupts

Due: **October 4^{th} 2024**

1 Introduction

We have spent the last few weeks implementing our 32-bit datapath. The simple 32-bit LC-4000 is capable of performing advanced computational tasks and logical decision making. Now it is time for us to move on to something more advanced—the upgraded LC-4000a enables the ability for programs to be interrupted. Your assignment is to fully implement and test interrupts using the provided datapath and CircuitSim. You will hook up the interrupt and data lines to the new timer device, modify the datapath and microcontroller to support interrupt operations, and write an interrupt handler to operate this new device. You will also use the tiny, inexpensive LC-4000a as an embedded system to monitor a kitchen appliance.

2 Requirements

Before you begin, please ensure you have done the following:

- Download the proper version of CircuitSim. A copy of CircuitSim is available under Files on Gradescope. You may also download it from the CircuitSim website (https://ra4king.github.io/CircuitSim/). In order to run CircuitSim, Java must be installed. If you are a Mac user, you may need to right-click on the JAR file and select "Open" in the menu to bypass Gatekeeper restrictions.
- CircuitSim is still under development and may have unknown bugs. Please back up your work using some form of version control, such as a local/private git repository or Dropbox. **Do not use public git repositories; it is against the Georgia Tech Honor Code**.
- The LC-4000a assembler is written in Python. If you do not have Python 2.6 or newer installed on your system, you will need to install it before you continue.

3 What We Have Provided

- A reference guide to the LC-4000a is located in *Appendix A: LC-4000a Instruction Set Architecture*. **Please read this first before you move on!** The reference introduces several new instructions that you will implement for this project.
- A microcode file (`microcode.xlsx`) that meets the requirements of Project 1; however, feel free to supply your own. This microcode file has a new configuration with additional bits for the new signals that will be added in this project.
- A timer device that will generate an interrupt signal at regular intervals. The pinout and functionality of this device are described in *Adding an External Timer Device*.
- A distance tracker that will generate an interrupt signal at regular intervals, and provides distance tracker readings. The pinout and functionality of this device are described in *Adding a Distance Tracker*.
- An *incomplete* assembly program `prj2.s` that you will complete and use to test your interrupt capabilities.
- An assembler with support for the new instructions to assemble the test program.
- A completed LC-4000 datapath circuit (`LC-4000a.sim`) from Project 1 is provided. Use this as a base to add the basic interrupt support for the LC-4000a or build off of your own Project 1 datapath, **but you must make sure the file is named** `LC-4000a.sim`. Most of the work can be easily carried over from one datapath to another.
- **Note:** We are releasing a prerelease version of this project for students that want to start early. The respective `LC-4000a.sim` and `microcode.xlsx` will be blank until our full release on September 19th, 2024. Feel free to use your own implementations from Project 1 to aid you until our full release.

4 Phase 1 - Implementing a Basic Interrupt

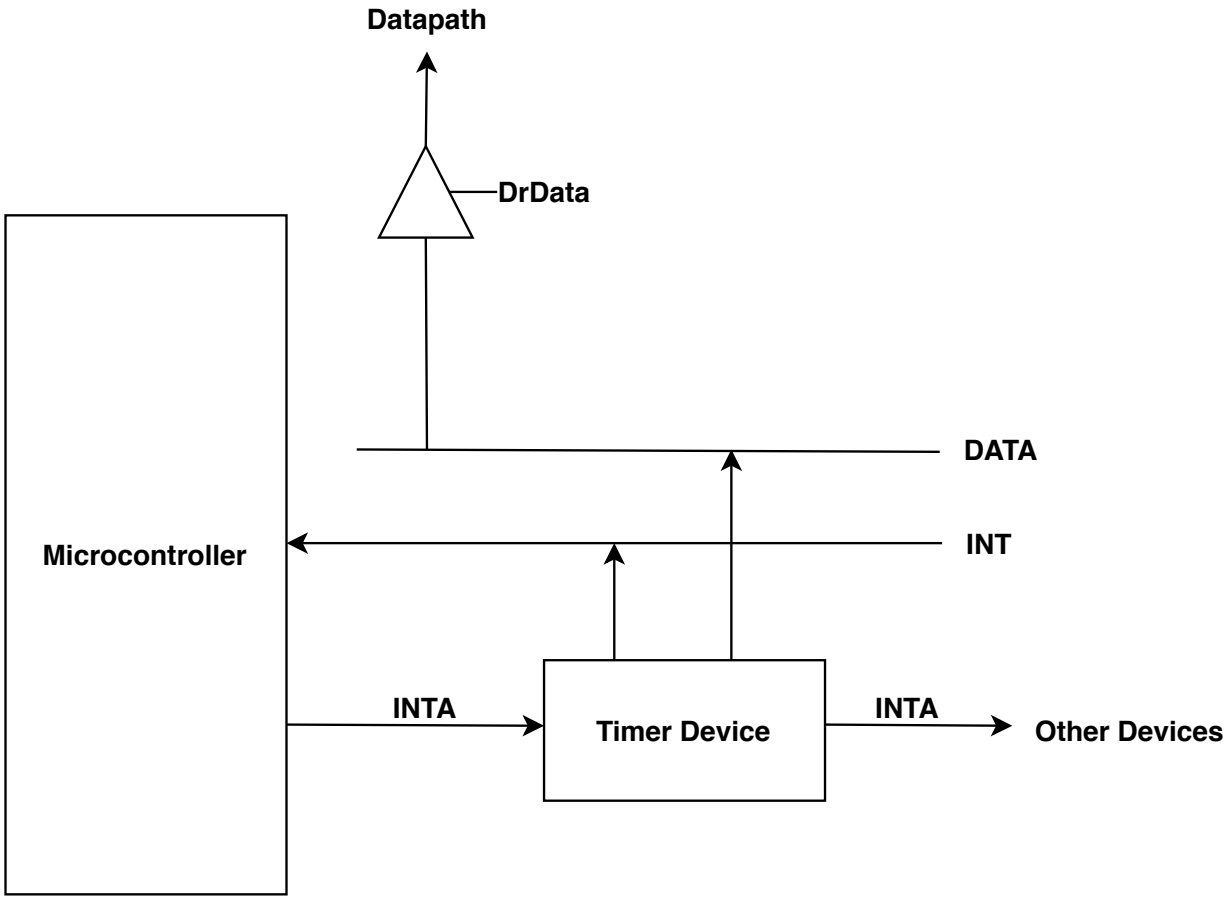


Figure 1: Basic Interrupt Hardware for the LC-4000a Processor

For this assignment, you will add interrupt support to the LC-4000a datapath. Then, you will test your new capabilities to handle interrupts using an external timer device.

Work in the LC-4000a.sim file. If you wish to use your existing datapath, make a copy with this name, and add the devices we provided.

4.1 Interrupt Hardware Support

First, you will need to add the hardware support for interrupts.

You must do the following:

1. Our processor needs a way to turn interrupts on and off. Create a new one-bit "Interrupt Enable" (IE) register. You'll connect this register to your microcontroller in a later step.
2. Create the INT line. The external device you will create in 4.2 will pull this line high (assert a '1') when they wish to interrupt the processor. Because multiple devices can share a single INT line, only one device can write to it at once. When a device does not have an interrupt, it neither pulls the line high nor low. You must accommodate this in your hardware by making sure that the final value going to the microcontroller always has a value (i.e. not a blue wire in CircuitSim). This can be done

by using a specific gate to act like a pull-down resistor so that there is always a value asserted (See Appendix C for more information)..

3. When a device receives an **IntAck** signal, it will drive its 32-bit device ID onto the I/O Data Bus. To prevent devices from interfering with the processor, the I/O Data Bus is attached to the Main Bus with a tri-state driver. Create this driver and the bus, and attach the microcontroller's **DrDATA** signal to the driver.
4. Modify the datapath so that the PC starts at 0x08 when the processor is reset. Normally the PC starts at 0x00, however we need to make space for the interrupt vector table (IVT). Therefore, when you actually load in the test code that you will write, it needs to start at 0x08. Please make sure that your solution ensures that datapath can never execute from below 0x08 - or in other words, force the PC to drive the value 0x08 if the PC is pointing in the range of the interrupt vector table.
5. Create hardware to support selecting the register $k0 within the microcode. This is needed by some interrupt related instructions. Because we need to access $k0 outside of regular instructions, we cannot use the Rx / Ry / Rz bits. **HINT:** Use only the register selection bits that the main ROM already outputs to select $k0. Notice that there is an unused input to the RegSel multiplexer.

4.2 Adding an External Timer Device

Hardware timers are an essential device in any CPU design. They allow the CPU to monitor the passing of various time intervals, without dedicating CPU instructions to the cause.

The ability of timers to raise interrupts also enables preemptive multitasking, where the operating system periodically interrupts a running process to let another process take a turn. Timers are also essential to ensuring a single misbehaving program cannot freeze up your entire computer.

You will connect an external timer device to the datapath. It is internally configured to have a **device ID of 0x0** and **interrupt every 2000 clock ticks**.

The pinout of the timer device is described below. If you like, you may also examine the internals of the device in CircuitSim.

- **CLK**: The clock input to the device. Make sure you connect this to the same clock as the rest of your circuit.
- **INT**: The device will begin to assert this line when its time interval has elapsed. It will not be lowered until the cycle after it receives an INTA signal.
- **INTA_IN**: When the INTA_IN line is asserted while the device has asserted the INT line, it will drive its device ID to the DATA line and lower its INT line **on the next clock cycle**.
- **INTA_OUT**: When the INTA_IN line is asserted while the device does not have an interrupt pending, its value will be propagated to INTA_OUT. This allows for daisy chaining of devices.
- **DATA**: The device will drive its ID (0x0) to this line after receiving an INTA.

The INT and DATA lines from the timer should be connected to the appropriate buses that you added in the previous section.

4.3 Microcontroller Interrupt Support

Before beginning this part, be sure you have read through *Appendix A: LC-4000a Instruction Set Architecture* and *Appendix B: Microcontrol Unit* and pay special attention to the new instructions. However, for this part of the project, you do not need to worry about the LdDAR signal or the IN instruction.

In this part of the assignment you will modify the microcontroller and the microcode of the LC-4000a to support interrupts. You will need to do the following:

1. Be sure to read the appendix on the microcontroller before starting this section.
2. Modify the microcontroller to support asserting four new control signals:
   (a) **LdEnInt** & **EnInt** to control whether interrupts are enabled/disabled. You will use these 2 signals to control the value of your interrupts enabled register.
   (b) **IntAck** to send an interrupt acknowledge to the device.
   (c) **DrDATA** to drive the value on the I/O Data Bus to the Main Bus.
3. Extend the size of the ROM accordingly.
4. Add the fourth ROM described in *Appendix B: Microcontrol Unit* to handle onInt.
5. Modify the FETCH macrostate microcode so that we actively check for interrupts. Normally this is done within the INT macrostate (as described in Chapter 4 of the book and in the lectures) but we are rolling this functionality in the FETCH macrostate for the sake of simplicity. You can accomplish this by doing the following:
   (a) First check to see if the CPU should be interrupted. To be interrupted, two conditions must be true: (1) interrupts are enabled (i.e., the IE register must hold a '1'), and (2) a device must be asserting a '1' on the INT signal line.
   (b) If not, continue with FETCH normally.
   (c) If the CPU should be interrupted, then we enter the INT macrostate and perform the following:
      i. Save the current PC to the register $k0.
      ii. Disable interrupts.
      iii. Assert the interrupt acknowledge signal (IntAck). Next, drive the device ID from the I/O Data Bus and use it to index into the interrupt vector table to retrieve the new PC value. The device will drive its device ID onto the I/O Data Bus one clock cycle **after** it receives the IntAck signal.
      iv. This new PC value should then be loaded into the PC.

   Note: `onInt` works in the same manner that CmpOut did in Project 1. The processor should branch to the appropriate microstate depending on the value of `onInt`. `onInt` should be true when interrupts are enabled AND when there is an interrupt to be acknowledged. Note: The mode bit mechanism and user/kernel stack separation discussed in the textbook has been omitted for simplicity.
6. Implement the microcode for three new instructions for supporting interrupts as described in Chapter 4. These are the EI, DI, and RETI instructions. You need to write the microcode in the main ROM as well as the SEQ ROM for these three new instructions. Keep in mind that:
   (a) EI sets the IE register to 1.
   (b) DI sets the IE register to 0.
   (c) RETI loads $k0 into the PC, and enables interrupts.

4.4 Implementing the Timer Interrupt Handler

Our datapath and microcontroller now support receiving interrupts from devices, BUT we must now implement the interrupt handler `timer_handler` within the `prj2.s` file to handle interrupts from the timer device in a way that doesn't incorrectly interfere with any user programs.

In `prj2.s`, we provide you with a modified version of `pow.s` that will run while you are waiting for interrupts. For this part of the project, you need to write the interrupt handler for the timer device (device ID 0x0).

You should refer to Chapter 4 of the textbook to see how to write a correct interrupt handler. As detailed in that chapter, your handler will need to do the following:

1. First save the current value of $k0 (the return address to where you came from to the current handler)
2. Enable interrupts (which should have been disabled implicitly by the processor within the INT macrostate).
3. Save the state of the interrupted processor.
4. Implement the actual work to be done in the handler. In the case of this project, we want you to **increment a counter variable in memory**, which we have already provided.
5. Restore the state of the original processor and return using RETI.

The handler you have written for the timer device should run every time the device interrupts the processor. Make sure to write the handler such that interrupts can be nested. With that in mind, interrupts should be disabled for **as few instructions as possible** within the handlers.

You will need to do the following:

1. Write the interrupt handler (should follow the above instructions or simply refer to Chapter 4 in your book). In the case of this project, we want the interrupt handler to keep track of time in memory at the predetermined location: 0xFFFF
2. Load the starting address of the first handler you just implemented in `prj2.s` into the interrupt vector table at the appropriate addresses (the table is indexed using the device ID of the interrupting device).

Test your design before moving onto the next section. If it works correctly, you should see the value at address 0xFFFF in memory increment as the program runs.

5 Phase 2 - Implementing Interrupts from Input Devices

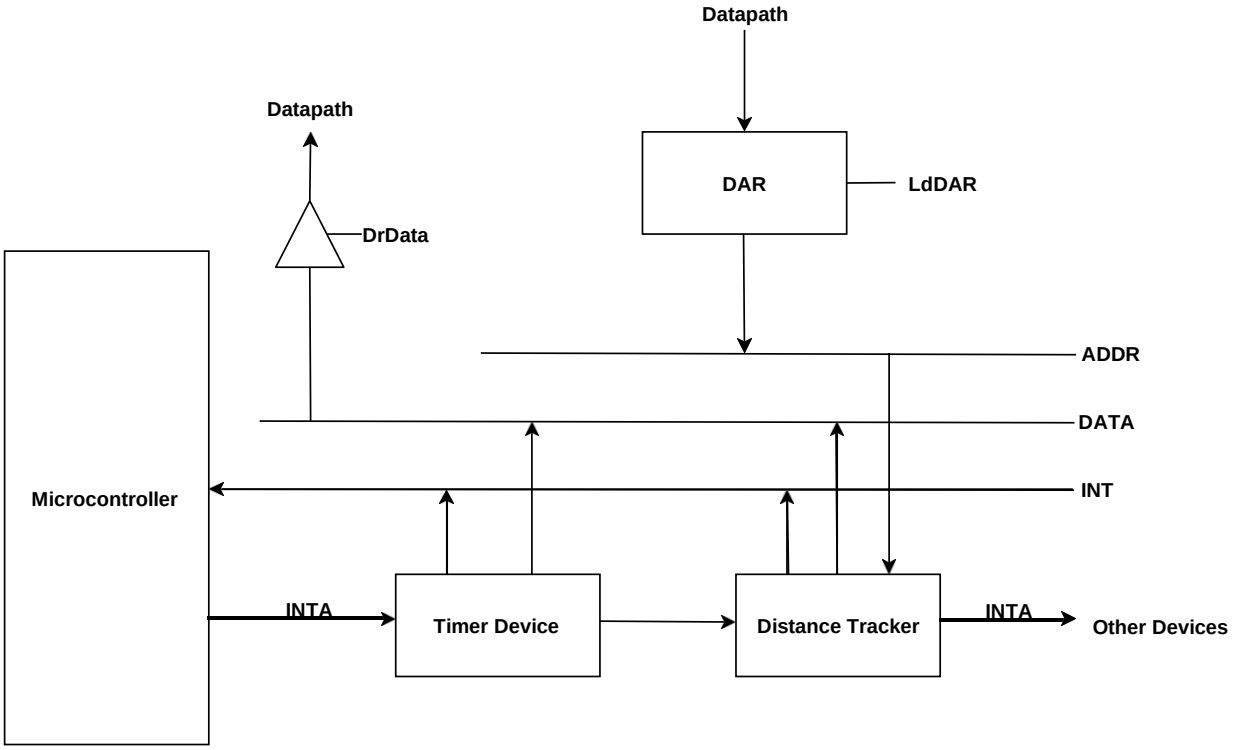


Figure 2: Interrupt Hardware for the LC-4000a with Basic I/O Support

Eager to put your newfound knowledge of device interrupts from CS2200 to good use, you decide to apply what you've learned to your engineering passion: a distance tracker! You are interested to know the maximum and minimum distance in that area.

You've rigged up a device that is able to report the current distance measured to an LC-4000a processor via an interrupt. There's only one issue: as of right now, your datapath can detect when an external device is ready to interrupt the processor, but it cannot retrieve data from external devices.

In this phase of the project, you will add functionality for device-addressed input. You will then make use of this functionality by adding a device simulating a distance tracker and writing a simple handler for the device.

5.1 Basic I/O Support

Before adding the distance tracker, you will first need to add support for device addressed I/O. In order to get input from a device such as a distance tracker, you will write a value to an Address Bus, which instructs the device with that address (which in this case is the same as the device ID) to write its output data to the I/O Data Bus.

You must do the following:

1. Create the device address register (DAR) and connect its enable to the LdDAR signal from your microcontroller. This register gets its input from the Main Bus, and its output will be directly connected to the Address Bus. It will allow us to assert a value on the Address Bus while using the Main Bus for other operations.

2. Modify the microcontroller to support a new control signal, **LdDAR**. This signal will be used in order to enable writing to the DAR.
3. Implement the IN instruction in your microcode. This instruction takes a device address an immediate offset (IR[19:0]), loads it into the DAR, and writes the value on the data bus into a register. When it is done, it **must clear the DAR** (since interrupts use the data bus to communicate device IDs). Examine the format of the IN instruction and consider what signals you might raise in order to write a constant zero into the DAR.

5.2 Adding a Distance Tracker

You will connect a distance tracker to your datapath that simulates a distance tracker by returning the current distance. Its internals are similar to the timer device, meaning it asserts interrupts and handles acknowledgments in the same way. Every 1500 cycles, it will assert an interrupt signaling that a distance value has been captured. This distance can be fetched as a 32-bit word by writing the device's address to the ADDR line.

The distance tracker is internally configured to have a **device ID of 0x1**.

Place the distance tracker in your datapath circuit. This device will share the INT and DATA lines with the timer you added previously. However, it should receive its INTA signal from the INTA_OUT pin on the timer device. This ensures that if both the timer and distance tracker raise an interrupt at the same time, the timer will be acknowledged first, and the distance tracker will be acknowledged after. **This is known as "daisy chaining" devices.**

5.3 Implementing the Distance Tracker Handler

Now that your LC-4000a datapath can accept data from your distance tracker, we need to decide what to do with the data. In this case, we want to keep track of the *maximum* and *minimum* distance we have seen so far in two particular memory locations, 0xFFFD and 0xFFFC. You'll have to implement this logic in your handler, which will work similarly to the one you wrote for the timer device. However, instead of incrementing a timer at a memory location, **you will be keeping track of the maximum and minimum distance we have seen so far.** You will also keep track of the **range** between the maximum and minimum distances we have seen so far.

Note: Your code should be able to update the minimum and maximum in the same iteration if necessary.

In addition to the usual overhead of an interrupt handler, your distance tracker handler must do the following:

1. Use the IN instruction to obtain the most recently captured distance value from the distance tracker.
2. Write the value obtained from the distance tracker to the memory location with the address 0xFFFD **only if the value is greater than the current maximum** or the address 0xFFFC **only if the value is less than the current minimum**.
3. Calculate the range and store it in address 0xFFFE.

Make sure that you properly install the location of the new handler into the IVT.

The distance tracker hardware is designed to emit a sequence of numbers representing distance readings. If your design is working properly, you should see the value stored in the memory location 0xFFFD increase and 0xFFFC decrease after a few thousand clock cycles as it updates when a new distance value is pushed onto the datapath.

To validate you're updating the distance expended correctly, you can check the values that the distance tracker will emit by inspecting the internals of the circuit and checking the values in the ROM labeled 'Key Buffer'.

6 Autograder

6.1 Testcases

Similar to the autograder used in Project 1, the autograder for Project 2 will run your `prj2.s` file using your datapath and simulate the interrupt handling process. It will evaluate whether your handler codes perform their intended tasks correctly. The Project 2 autograder includes four different test cases:

- **Verification of hardware and microcode**: Confirms that both your hardware and microcode are implemented correctly.
- **Device 1 interrupt handler check**: Ensures that the assembly code for the Device 1 interrupt handler is correctly implemented.
- **Device 2 interrupt handler check**: Validates the correct implementation of the assembly code for the Device 2 interrupt handler.
- **Comprehensive integration test**: Combines all the components mentioned above into a single test to evaluate their combined functionality.

You are encouraged to use the autograder as a tool to help debug your circuit and assembly code, but it shouldn't be your primary method. You'll still need to identify which parts of your datapath, microcode, or handler code aren't functioning as expected. Once you are ready to submit to the autograder, you must adhere to the following guidelines:

- Do not rename any existing components.
- Name your IE register as "IE."
- Name your Interrupt ROM as "INT."
- Use a single global clock.
- Use only one RAM for memory.
- Do not alter the layout of the microcode Excel sheet.
- If you've modified any constants in devices for debugging, revert them to their original values.

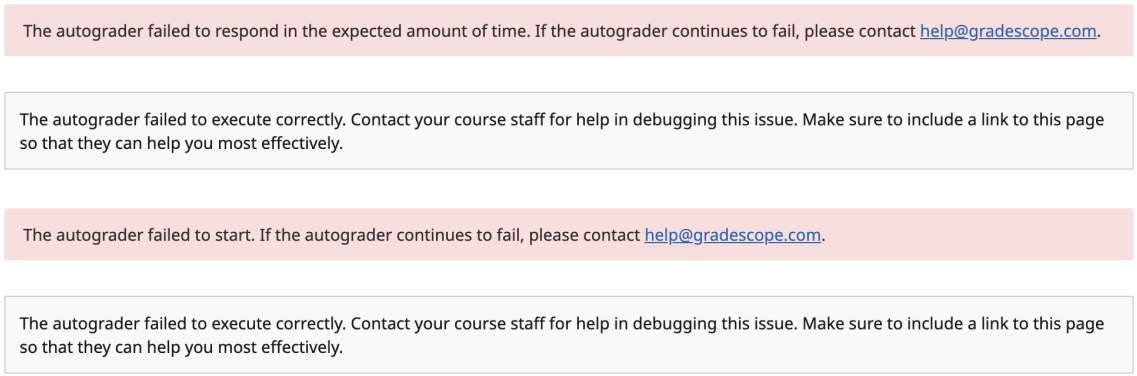
If the autograder fails, first check that you have followed all the rules above and those from Project 1. If the autograder flags an issue with a device handler, load the assembled HEX of your prj2.s program into RAM, clock it until the device interrupts, and monitor the state changes to identify where a component may be malfunctioning. Occasionally, the error might not reproduce during the first interrupt of the device; in such cases, observe all subsequent interrupts of that device to pinpoint the problem.

We **strongly recommend** testing your project locally before submitting it to Gradescope. For instructions on local testing, refer to Appendix D.

When you encounter an error message, first try to reproduce it locally and analyze what you observe. If you still need guidance, visit office hours or make a private post detailing your debugging attempts, rather than just sharing the error message and a screenshot of your datapath. The TAs will need more information to effectively help you resolve the issue.

6.2 Unexpected Autograder Errors

Sometimes, the autograder does not run as expected, and you will see outputs similar to those below.



If so, make a private post on Ed and link the gradescope submission for which the error occurred. A TA will try to rerun your submission for you, and the correct autograder output should be displayed on the same submission. Note, this will not count towards your submission count for the day.

7 Deliverables

To submit your project, you need to upload the following files to Gradescope:

- CircuitSim datapath file (LC-4000a.sim)
- Microcode file (microcode.xlsx)
- Assembly code (prj2.s)

If you are missing any of those files, you will get a 0, so make sure that you have uploaded all three of them.

Always re-download your assignment from Gradescope after submitting to ensure that all necessary files were properly uploaded. If what we download does not work, you will get a 0 regardless of what is on your machine.

This project will be demoed. In order to receive full credit, you must sign up for a demo slot and complete the demo. We will announce when demo times are released, which will be after the project is due.

8 Appendix A: LC-4000a Instruction Set Architecture

The LC-4000a is a simple, yet capable computer architecture. The LC-4000a combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The LC-4000a is a **word-addressable**, **32-bit** computer. **All addresses refer to words**, i.e. the first word (four bytes) in memory occupies address 0x0, the second word, 0x1, etc.

All memory addresses are truncated to 16 bits on access, discarding the 16 most significant bits if the address was stored in a 32-bit register. This provides roughly 64 KB of addressable memory.

8.1 Registers

The LC-4000a has 16 general-purpose registers. While there are no hardware-enforced restraints on the uses of these registers, your code is expected to follow the conventions outlined below.

Table 1: Registers and their Uses

Register Number	Name	Use	Callee Save?
0	$zero	Always Zero	NA
1	$at	Assembler/Target Address	NA
2	$v0	Return Value	No
3	$a0	Argument 1	No
4	$a1	Argument 2	No
5	$a2	Argument 3	No
6	$t0	Temporary Variable	No
7	$t1	Temporary Variable	No
8	$t2	Temporary Variable	No
9	$s0	Saved Register	Yes
10	$s1	Saved Register	Yes
11	$s2	Saved Register	Yes
12	$k0	Reserved for OS and Traps	NA
13	$sp	Stack Pointer	No
14	$fp	Frame Pointer	Yes
15	$ra	Return Address	No

1. **Register 0** is always read as zero. Any values written to it are discarded. Regardless of what is written to this register, it should always output zero.
2. **Register 1** is used to hold the target address of a jump. It may also be used by pseudo-instructions generated by the assembler.
3. **Register 2** is where you should store any returned value from a subroutine call.
4. **Registers 3 - 5** are used to store function/subroutine arguments. **Note:** registers 2 through 8 should be placed on the stack if the caller wants to retain those values. These registers are fair game for the callee (subroutine) to trash.
5. **Registers 6 - 8** are designated for temporary variables. The caller must save these registers if they want these values to be retained.
6. **Registers 9 - 11** are saved registers. The caller may assume that these registers are never tampered with by the subroutine. If the subroutine needs these registers, then it should place them on the stack and restore them before they jump back to the caller.
7. **Register 12** is reserved for handling interrupts.
8. **Register 13** is the everchanging top of the stack; it keeps track of the top of the activation record for a subroutine.

9. **Register 14** is the anchor point of the activation frame. It is used to point to the first address on the activation record for the currently executing process.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing.

8.2 Instruction Overview

The LC-4000a supports a variety of instruction forms, only a few of which we will use for this project. The instructions we will implement in this project are summarized below.

Table 2: LC-4000a Instruction Set

	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
ADD	0000	DR	SR1	unused	SR2
NAND	0001	DR	SR1	unused	SR2
ADDI	0010	DR	SR1	immval20	
LW	0011	DR	BaseR	offset20	
SW	0100	SR	BaseR	offset20	
BEQ	0101	SR1	SR2	offset20	
JALR	0110	AT	RA	unused	
HALT	0111	unused			
BLT	1000	SR1	SR2	offset20	
LEA	1001	DR	unused	PCoffset20	
MIN	1010	DR	SR1	unused 0	SR2
MAX	1010	DR	SR1	unused 1	SR2
EI	1011	unused			
DI	1100	unused			
RETI	1101	unused			
IN	1110	DR	0000	addr20	

8.2.1 Conditional Branching

Branching in the LC-4000a ISA is slightly different than usual. We have a set of branching instructions including both BEQ and BLT, which offer the ability to branch upon a certain condition being met. These instructions use comparison operators, comparing the values of two source registers. If the comparisons are true (for example, with the BLT instruction, if SR1 $<$ SR2), then we will branch to the target destination of incrementedPC + offset20. For MIN, if SR1 $<$ SR2, we branch to the series of microstates that stores the value in SR1 to DR. Otherwise, we branch to a series of microstates that stores the value in SR2 to DR. For MAX, we do the opposite

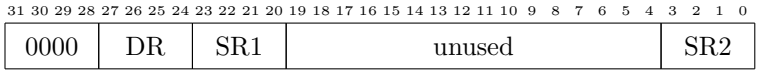
8.3 Detailed Instruction Reference

8.3.1 ADD

Assembler Syntax

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

Encoding



Operation

```
DR = SR1 + SR2;
```

Description

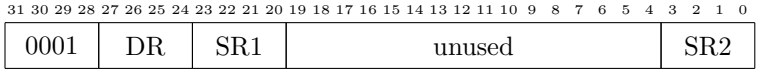
The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

8.3.2 NAND

Assembler Syntax

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

Encoding



Operation

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

Description

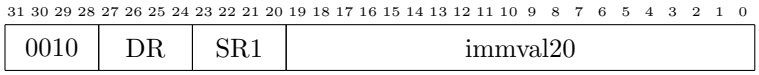
The NAND instruction performs a logical NAND (AND NOT) on the source operands obtained from SR1 and SR2. The result is stored in DR.

8.3.3 ADDI

Assembler Syntax

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

Encoding



Operation

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

Description

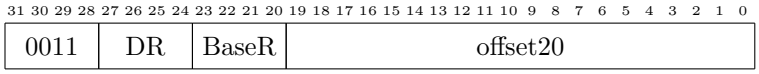
The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

8.3.4 LW

Assembler Syntax

```
LW    DR, offset20(BaseR)
```

Encoding



Operation

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

Description

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

8.3.5 SW

Assembler Syntax

```
SW    SR, offset20(BaseR)
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0100	SR	BaseR	offset20

Operation

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

Description

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

8.3.6 BEQ

Assembler Syntax

```
BEQ  SR1, SR2, offset20
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0101	SR1	SR2	offset20

Operation

```
if (SR1 == SR2) {
    PC = incrementedPC + offset20
}
```

Description

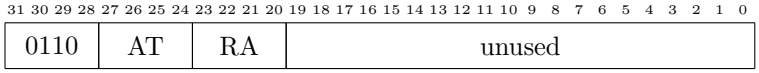
A branch is taken if SR1 is equal to SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

8.3.7 JALR

Assembler Syntax

```
JALR   AT, RA
```

Encoding



Operation

```
RA = PC;
PC = AT;
```

Description

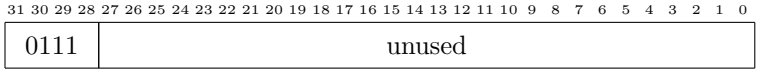
First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

8.3.8 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

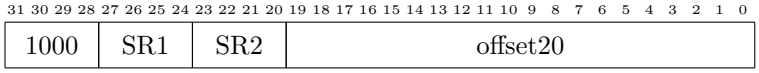
The machine is brought to a halt and executes no further instructions.

8.3.9 BLT

Assembler Syntax

```
BLT  SR1, SR2, offset20
```

Encoding



Operation

```
if (SR1 < SR2) {
    PC = incrementedPC + offset20
}
```

Description

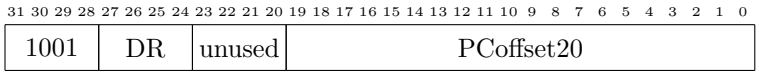
A branch is taken if SR1 is less than SR2. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

8.3.10 LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding



Operation

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

Description

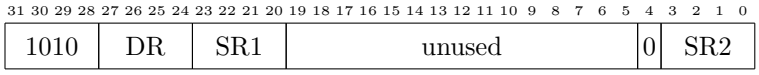
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the incremented PC (address of instruction + 1). It then stores the computed address into register DR.

8.3.11 MIN

Assembler Syntax

```
MIN    DR, SR1, SR2
```

Encoding



Operation

```
if (SR1 < SR2) {
    DR = SR1
} else {
    DR = SR2
}
```

Description

The minimum is computed between the values in both source registers. It then stores the minimum value in the register DR.

> **Note:** MIN and MAX have the same opcode. Bit 4 being 0 indicates the MIN instruction.
> **The control flow for MIN must go through the CC ROM. Not doing so will cause you to lose points!**

8.3.12 MAX

Assembler Syntax

```
MAX    DR, SR1, SR2
```

Encoding

31 30 29 28	27 26 25 24 23	22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
1010	DR	SR1	unused	1	SR2

Operation

```
if (SR1 > SR2) {
    DR = SR1
} else {
    DR = SR2
}
```

Description

The maximum is computed between the values in both source registers. It then stores the maximum value in the register DR.

> **Note:** MIN and MAX have the same opcode. Bit 4 being 1 indicates the MAX instruction.
> **The control flow for MAX must go through the CC ROM. Not doing so will cause you to lose points!**

8.3.13 EI

Assembler Syntax

```
EI
```

Encoding

31 30 29 28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1011	unused

Operation

```
IE = 1;
```

Description

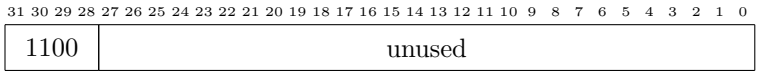
The Interrupts Enabled register is set to 1, enabling interrupts.

8.3.14 DI

Assembler Syntax

```
DI
```

Encoding



Operation

```
IE = 0;
```

Description

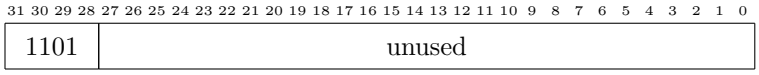
The Interrupts Enabled register is set to 0, disabling interrupts.

8.3.15 RETI

Assembler Syntax

```
RETI
```

Encoding



Operation

```
PC = $k0;
IE = 1;
```

Description

The PC is restored to the return address stored in $k0. The Interrupts Enabled register is set to 1, enabling interrupts.

8.3.16 IN

Assembler Syntax

```
IN DR, DeviceADDR
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1110	DR	0000	addr20

Operation

```
DAR = SEXT(addr20);
DR = DeviceData;
DAR = 0;
```

Description

The value in addr20 is sign-extended to determine the 32-bit device address. This address is then loaded into the Device Address Register (DAR). The processor then reads a single word value off the device data bus, and writes this value to the DR register. The DAR is then reset to zero, ending the device bus cycle.

9 Appendix B: Microcontrol Unit

The microcontrol unit drives all of the control signals to various items on the datapath. This Finite State Machine (FSM) can be constructed in a variety of ways. You could implement it with combinational logic and flip flops, or you could hard-wire signals using a single ROM. The single ROM solution will waste a tremendous amount of space since most of the microstates do not depend on the opcode or the conditional test to determine which signals to assert. For example, since the condition line is an input for the address, every microstate would have to have an address for condition = 0 as well as condition = 1, even though this only matters for one particular microstate.

To solve this problem, we will use a four ROM microcontroller which will also handle interrupts. In this arrangement, we will have 4 ROMs:

- the main ROM, which outputs the control signals,
- the sequencer ROM, which helps determine which microstate to go to at the end of the FETCH state,
- the condition ROM, which helps determine which microstate to go to after a branching instruction.
- and the interrupt ROM, which helps determine whether the next state is fetch2 or the start of the INT macrostate.

Examine the following:

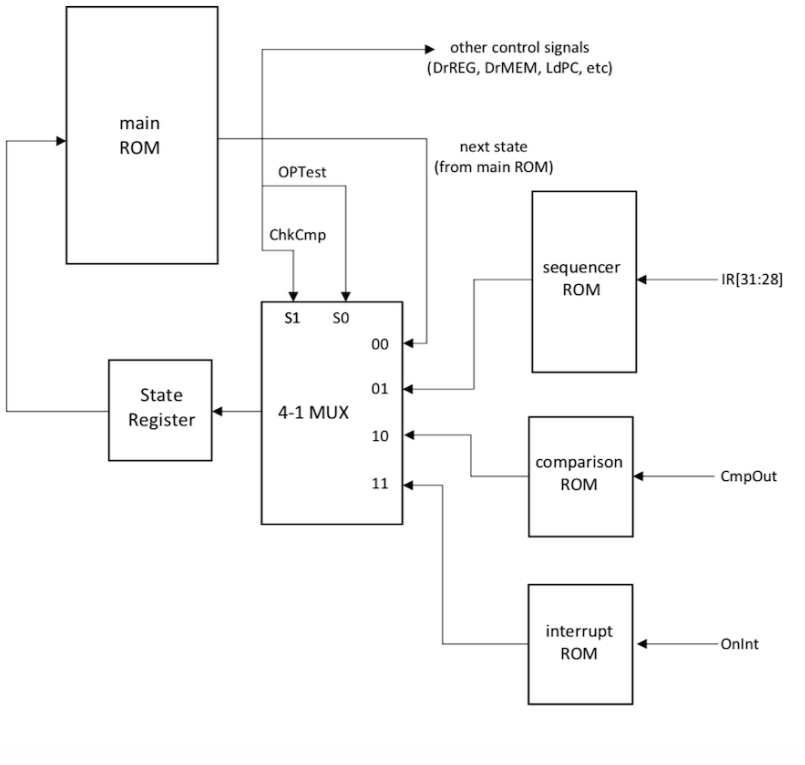


Figure 3: Four ROM Microcontrol Unit

As you can see, there are four different locations that the next state can come from: part of the output from the previous state (main ROM), the sequencer ROM, the condition ROM, and the interrupt ROM. The MUX controls which of these sources gets through to the state register. If the previous state's "next state" field determines where to go, neither the OPTest nor ChkCmp signals will be asserted. If the opcode from the IR determines the next state (such as at the end of the FETCH state), the OPTest signal will be asserted. If the comparison circuitry determines the next state (such as in the BGT instruction), the ChkCmp signal will be asserted. If an interrupt needs to be dealt with (entering the INT macrostate), both the OPTest and ChkCmp signals will be asserted.

The sequencer ROM should have one address per instruction, and the condition ROM should have three addresses for when the branching condition is true, when it is false, and when we are executing FABS microcode.

Before getting down to specifics you need to determine the control scheme for the datapath. To do this examine each instruction, one by one, and construct a finite state bubble diagram showing exactly what control signals will be set in each state. Also determine what are the conditions necessary to pass from one state to the next. You can experiment by manually controlling your control signals on the bus you've created in **Phase 1 - Implementing a Basic Interrupt** to make sure that your logic is sound.

Once the finite state bubble diagram is produced, the next step is to encode the contents of the Control Unit ROM. Then you must design and build (in CircuitSim) the Control Unit circuit which will contain the four ROMs, a MUX, and a state register. Your design will be better if it allows you to single step and ensure that it is working properly. Finally, you will load the Control Unit's ROMs with the hexadecimal generated by your filled out microcode.xlsx.

Note that the input address to the ROM uses bit 0 for the lowest bit of the current state and 5 for the highest bit for the current state.

Table 3: ROM Output Signals

Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose
0	NextState[0]	6	DrREG	12	LdIR	18	WrMEM	24	OPTest	30	LdDAR
1	NextState[1]	7	DrMEM	13	LdMAR	19	RegSel[0]	25	ChkCmp		
2	NextState[2]	8	DrALU	14	LdA	20	RegSel[1]	26	LdEnInt		
3	NextState[3]	9	DrPC	15	LdB	21	ALU[0]	27	EnInt		
4	NextState[4]	10	DrOFF	16	LdCmp	22	ALU[1]	28	IntAck		
5	NextState[5]	11	LdPC	17	WrREG	23	ALU[2]	29	DrData		

Table 4: Register Selection Map

RegSel[1]	RegSel[0]	Register
0	0	RX (IR[27:24])
0	1	RY (IR[23:20])
1	0	RZ (IR[3:0])
1	1	$k0 (1100)

Table 5: ALU Function Map

ALU2	ALU1	ALU0	Function
0	0	0	ADD
0	0	1	SUB
0	1	0	NAND
0	1	1	A + 1
1	0	0	Pass A

10 Appendix C: Pull Down and Pull Up Resistors

10.1 Intro

In CS 2200 and CS 2110, you've built functional logic gates that don't need to consider the physical variables of real circuit logic. There is another way that only needs half the number of transistors. This method works best with discrete components (but our two-transistor style is better with integrated circuits as the transistors takes less chip space overall).

10.2 How we build our logic gates

We build our gates so that the output is either connected to Vcc or Ground depending on the calculated output value. What we could do is build the gate so that the output is connected to one side of the power bus (either Vcc or Ground) when the output should be that value. When the output is the opposite value, we leave the output floating. This by itself would give us an unpredictable output value, but not if we connect a small resistor (known as a pull-up or pull down resistor depending on whether it is connected to Vcc or Ground, respectively) between the gate's output line and the opposite side of the power bus.

This resistor allows a tiny bit of current to flow; that current is overwhelmed when the output is connected to Vcc or Ground, but when the output is floating the resistor provides just enough charge to "pull" the output to the opposite potential. The downstream device is none the wiser, but this saves us about half the transistors in each gate. If you look at most other introductory digital logic textbooks, you may see that the authors build their gates with pull-up and pull-down resistors instead of the "two- transistor both sides switched" approach that CircuitSim, Patt et al., and Ramachandran et al. use.

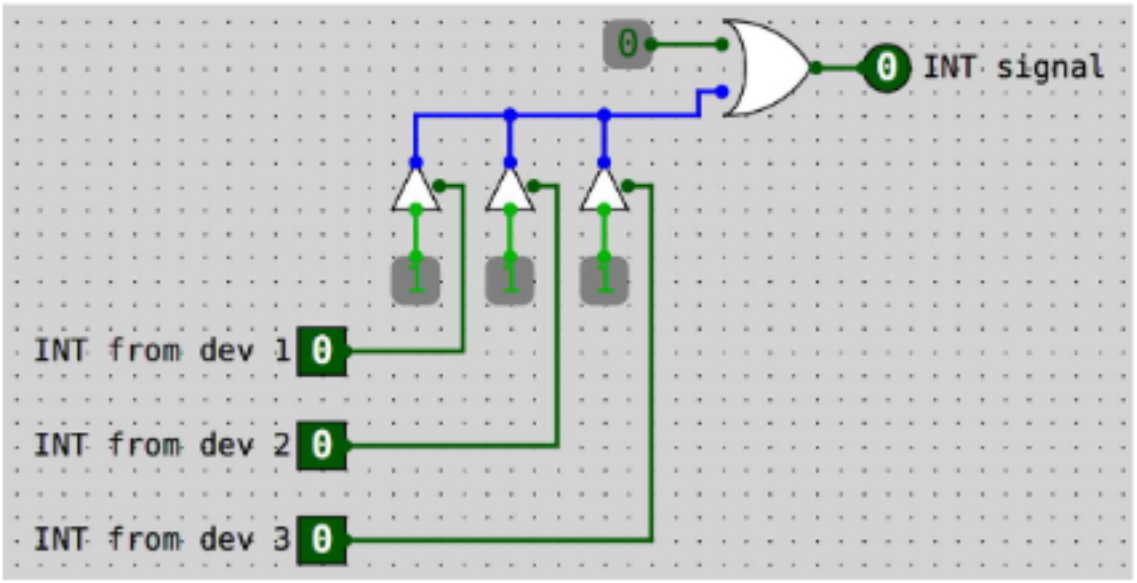
See Canvas → Files → Course Resources → Additional Resources → "Building Logic Gates from Transistors.pdf" for a more thorough explanation of the different methods for building gates.

10.3 Why are we telling you this?

Because the INT line is one of the very few situations that we can't handle with our two-transistor logic style. We need to combine several different signals on the INT line (which are connected by tri-state buffers) that are either 1 or floating (we can't allow devices to assert a 0 on the INT line because it will cause a short circuit when one of our devices asserts a 1 signal to signal an interrupt).

The way we would handle this in a real circuit would be to simply add a pull-down resistor from the INT line to Ground so that if none of the tri- state buffers are open to assert an interrupt 1 signal, the INT line will reliably read as 0. The immediate issue is that CircuitSim doesn't understand pull-up and pull-down resistors, so it instead has a reality-distorting "feature" that allows us to implement the interrupt line without pull-down resistors. The feature is this: A CircuitSim OR gate will consider a floating input line to be a 0.

This is not guaranteed in real life, but it is in CircuitSim. To implement the INT line, you can connect a signal line that is either 1 or floating to one side of an OR gate and connect a 0 to the other input. The output of the OR gate will be 1 if the first input is 1. If the first input is floating (or 0), the output will definitely be 0. The diagram below shows an implementation of the INT line circuit for 3 devices:


0
0
INT signal
1
1
1
INT from dev 1
0
INT from dev 2
0
INT from dev 3
0

11 Appendix D: Debugging Guide

11.1 Intro

Although it may seem confusing at first, **testing locally is the easiest and quickest way to find bugs in your project** (believe us: it will save you hours of work). To test locally, you can use the provided LC-4000a assembler (**assembler.py**) in the "assembly" folder. Using the assembler, you can assemble the **prj2.s** file (or any programs that you write), and then paste the assembled instructions into your CircuitSim datapath to run them.

11.2 How to Assemble a Program

Within the "assembly" folder, you will find **assembler.py**, along with your **prj2.s** file. To assemble an assembly file, **prj2.s** for example, simply navigate to the "assembly" folder in your terminal, and run the following command:

```
python assembler.py -i lc4000a prj2.s --hex prj2.hex
```

If everything is correct, you should see the following output:

```
Assembling for lc4000a architecture...
Writing to prj2.hex...done!
```

Check the "assembly" folder again. You will notice that there is a new file: **prj2.hex**. This file contains the assembled 32-bit instructions from your file in hexadecimal. If you are encountering errors when trying to assemble, it is likely from one of the following reasons:

- If you are getting the error "failed to assemble", then one or more of the instructions in your program is invalid. This is almost always due to a syntax error (such as an additional comma or a misspelled instruction).
- If you are getting a message regarding "unknown file" or "file not found", then make sure that A) you are in the "assembly" folder, and B) that you have properly spelled the file names correctly.
- If you are getting an error regarding "Python not found", then this means that your computer does not have Python on it. You can try to re-install Python, or simply just use the CS-2200 Docker Container, where Python is pre-installed.

11.3 Running an Assembled Program in CircuitSim

If you open the .hex file you assembler just generated, it will look similar to this (but probably with many more values).

```
00000000
9D000042
3DD00000
260FFFFF
B6000000
```

Select all of the values in the .hex file, copy them to your clipboard, and open your datapath in CircuitSim. Right click on the "Memory" component and select "View Internal State" from the pop-up menu. This will bring you into the Memory component. Then, right click on the "RAM" and select "Edit Contents". You should see a big CSV pop-up on your screen, displaying the contents of your RAM. Paste your assembled hex values into the RAM.

Your datapath is now ready to run your program! If you start clicking on the clock, you should see that instructions will start executing. To speed things up, you can click on the "Simulation" tab at the top of the screen. Here is a review of the basic functionalities in the "Simulation" tab:

- **To make the clock run continuously**, click *Clock Enabled*, or you can simply use Ctrl + K (Command + K on Mac).
- **To pause the clock**, simply do Ctrl + K (or Command + K) again.
- **To change the frequency of the clock**, click *Frequency*, where you can select how fast you want the clock to run.
- **To restart your program and reset your datapath**, click *Restart Simulation*, or use Ctrl + R (Command + R on Mac). Note that you will need to re-paste your hex values into RAM if you do this.

While your datapath is running the program, you can pause it at anytime and right click on components to view their internal state. For example, you can see if a particular register is being properly accessed or if a memory address is being updated.

11.4 Debugging Tips

By running a program in CircuitSim, you can easily identify bugs and errors. Here are a list of tips from the TAs to help you debug your code:

- **Routinely check if values in memory and registers are correctly updated.** This is a simple way to see if your datapath, microcode, and program are behaving correctly. For example, a memory value like *ticks* should increment by 1 over time, while *maxval* should strictly increase as the program executes.
- **Ensure that your handler implementations are correct.** If a register value or memory value isn't properly updating, it might not be a hardware error, but an error with your code in the **prj2.s** file. Make sure that you are following proper procedure in your handlers to ensure that $k0 is properly saved and restored. **Additionally, make sure that your handler implementations cover any edge cases!**
- **Use the CircuitSim breakpoint circuit.** This circuit will automatically pause the clock when it detects a specified input. This is particularly useful when you want to test the state of your datapath after an interrupt signal has been asserted.
- **If you are encountering a short circuit**, then immediately try to re-create it. Try to see where the short-circuit is occurring in your datapath by viewing the internal states of each component. Remember: short circuits often happen when 2 values are forced onto the same wire simultaneously. Manually stepping through each clock cycle can help determine this.
- **Write your own test programs**. The given **prj2.s** file is rather complex. If you want to isolate and test a specific issue, such as an individual instruction, then it's easier to write your own assembly program. It only needs to be a couple of lines long, but it can help tremendously to verify that a specific feature behaves properly.

Debugging can get frustrating at times, especially with CircuitSim and assembly. The best tip is to always try to narrow down your options for potential bug sources. Blind testing is almost always futile.