

1 Introduction

We have spent the last few weeks implementing our 32-bit datapath. The simple 32-bit RAMA-2200 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 RAMA-2200a 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.

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 Canvas. 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 RAMA-2200a 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.
- The MCGen program provided for generating microcode uses JavaFX, which is not included by default as part of Java 11. It will run easily on Java 8, but if you are running Java 11 on your computer, you may need to do some work to get it working. There is information about this online.

3 What We Have Provided

- A reference guide to the RAMA-2200a is located in *Appendix A: RAMA-2200a 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 CircuitSim circuit (`int-devices.sim`) containing a timer device and keyboard device subcircuit that you will use for this project. **You should copy and paste the contents of the new devices into subcircuits in your main circuit file.**
- A new MCGen configuration file `config.txt` 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 keyboard device that will generate an interrupt signal at regular intervals, and provides key press data. The pinout and functionality of this device are described in *Adding a Keyboard Device*.
- 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.
- An *incomplete* RAMA-2200a datapath circuit (`RAMA-2200a.sim`) that you may add the basic interrupt support onto will be provided **Tuesday, September 24**, after the one-time forgiveness period for

Project 1 has passed. You are also free to build off of your own Project 1 datapath, **but you must rename the file to RAMA-2200a.sim**. Most of the work can be easily carried over from one datapath to another.

- We will also release a `microcode.mc` microcode file that meets the requirements of Project 1, but feel free to supply your own.

4 Phase 1 - Implementing a Basic Interrupt

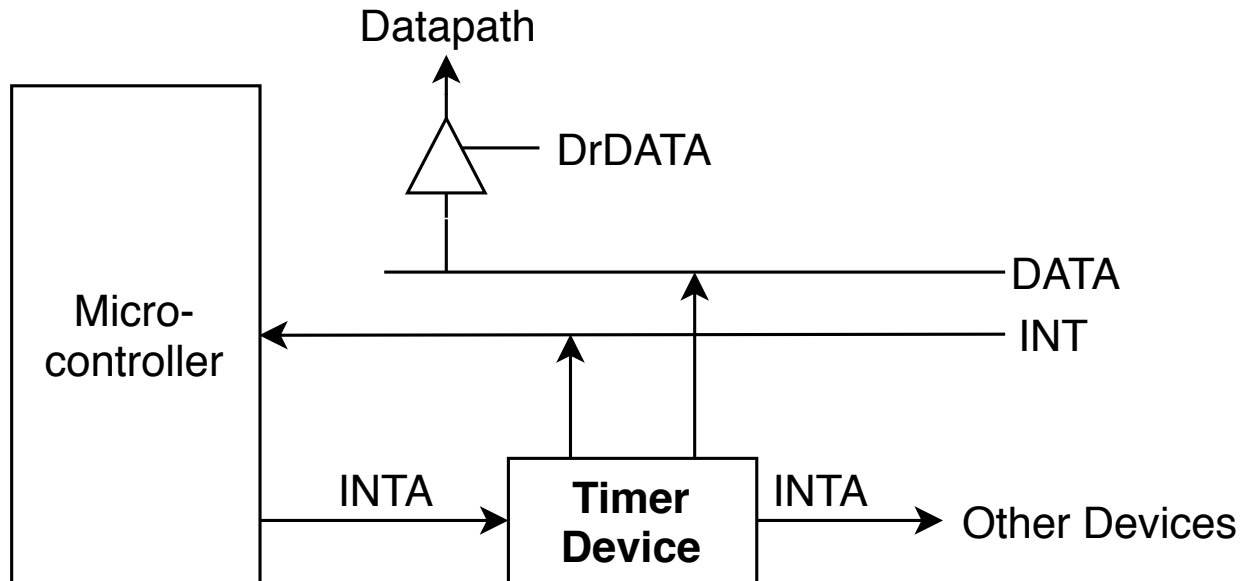


Figure 1: Basic Interrupt Hardware for the RAMA-2200a Processor

For this assignment, you will add interrupt support to the RAMA-2200a datapath. Then, you will test your new capabilities to handle interrupts using an external timer device.

Work in the RAMA-2200a.circ 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 accomodate 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).

3. When a device receives an **IntAck** signal, it will drive a 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 0x10 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 0x10. Please make sure that your solution ensures that datapath can never execute from below 0x10 - or in other words, force the PC to drive the value 0x10 if the PC is pointing in the range of the 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.

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 0x1** and a **2000-cycle tick timer interval**.

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 (0x1) 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: RAMA-2200a 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 RAMA-2200a 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 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 an interrupt.
 - (b) If not, continue with FETCH normally.
 - (c) If the CPU should be interrupted, then perform the following:

Interrupt ROM -> int 1

 - 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 ChkCmp 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 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 controlling the datapath 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 fully support interrupts from devices, BUT we must now implement the interrupt handler `t1_handler` within the `prj2.s` file to support interrupts from the timer device while also not interfering with the correct operation of any user programs.

In `prj2.s`, we provide you with a program that runs in the background. For this part of the project, you need to write interrupt handler for the timer device (device ID 0x1). 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 program.
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 program and return using `RETI`.

The handler you have written for the timer device should run every time the device's interrupt is triggered. Make sure to write the handler such that interrupts can be nested. With that in mind, interrupts should be enabled for **as long 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 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 a location in memory increment as the program runs.

5 Phase 2 - Implementing Interrupts from Input Devices

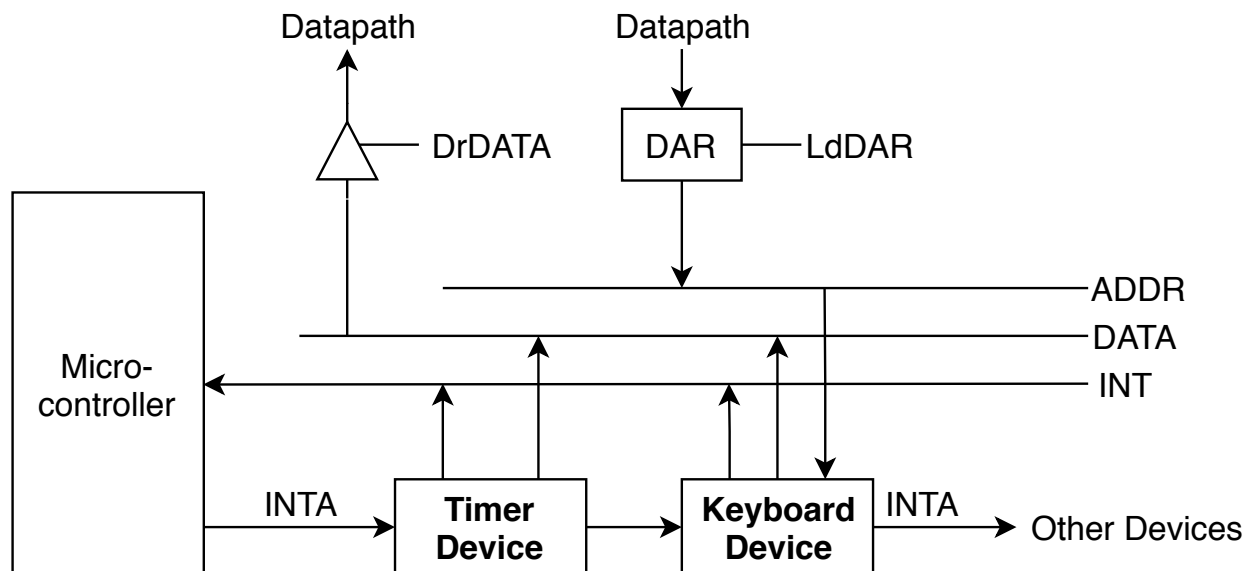


Figure 2: Interrupt Hardware for the RAMA-2200a with Basic I/O Support

Your datapath can now detect when external devices are ready, but cannot get data from input devices. In this part 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 keyboard and writing a simple handler for the device.

5.1 Basic I/O Support

Before adding the keyboard device, you will first need to add support for device addressed I/O. In order to get input from a device such as a keyboard, 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 send 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 from the immediate, 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 to zero** (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 Keyboard Device

You will connect a keyboard device to your datapath that simulates a real keyboard (you cannot actually type into it). Its internals are similar to the timer device, meaning it asserts interrupts and handles acknowledgements in the same way. Every 2500 cycles, it will assert an interrupt signaling that a key has been pressed. This key can be fetched as a 32-bit word by writing the device's address to the ADDR line.

The keyboard is internally configured to have a **device ID of 0x2**

Place the keyboard device 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 keyboard raise an interrupt at the same time, the timer will be acknowledged first, and the keyboard will be acknowledged after. **This is known as “daisy chaining” devices.**

5.3 Implementing the Keyboard Interrupt Handler

The handler for your keyboard will work similarly to the one you wrote for the timer device. However, instead of incrementing a timer at a memory location, you will write a character to the memory location pointed to by a tail pointer. You'll then increment the tail pointer to the next memory location.

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

1. Use the IN instruction to obtain the most recently pressed key from the keyboard. Note that although an ASCII character is only 7 bits, our ISA has only word addressable operations. Because of this, zeros are appended to the output of the keyboard.
2. Write the value obtained from the keyboard to the memory location pointed to by the tail pointer which starts at 0xFFE0.

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

The keyboard hardware is designed to emit **DEAD BEEF BOBA CAFE**. If your design is working properly, you should see each hex nibble in one memory location after the program finishes running.

6 Deliverables

Please submit all of the following files in a **.tar.gz** archive generated by one of the following:

- **On Linux/Mac:** Use the provided Makefile. The Makefile will work on any Unix or Linux-based machine (on Ubuntu, you may need to `sudo apt-get install build-essential` if you have never installed the build tools). Run `make submit` to automatically package your project into the correct archive format.
- **On Windows:** Use the provided submit.bat script. Submitting through this method will require 7zip (<https://www.7-zip.org/>) to be installed on your system. Run `submit.bat` to automatically package your project into the correct archive format. **Note:** Sometimes 7zip isn't added to your path when you install it, and you may get an error. If this happens, try running `set PATH=%PATH%;C:\Program Files\7-Zip\`.

The generated archive should contain at a minimum the following files:

- CircuitSim datapath file (RAMA-2200a.sim)
- Microcode file (microcode.mc)
- Assembly code (prj2.s)

Always re-download your assignment from Canvas 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.

7 Appendix A: RAMA-2200a Instruction Set Architecture

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

The RAMA-2200a 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.

7.1 Registers

The RAMA-2200a 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. **Note:** for the purposes of this project, you must implement the zero register. Regardless of what is written to this register, it should always output zero.
2. **Register 1** is 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. While it should be implemented, it otherwise will not have any special use on this assignment.
8. **Register 13** is your anchor on the stack. It keeps track of the top of the activation record for a subroutine.
9. **Register 14** is used to point to the first address on the activation record for the currently executing process.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing.

7.2 Instruction Overview

The RAMA-2200a 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: RAMA-2200a 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																								SR2
NAND	0001								DR																								SR2
ADDI	0010								DR																								immval20
LW	0011								DR																								offset20
SW	0100								SR																								offset20
GOTO	0101							0000				unused																					PCOffset20
JALR	0110							RA				AT																					unused
HALT	0111																																unused
SKP	1000							mode				SR1																					SR2
LEA	1001							DR				unused																					offset20
EI	1010																																unused
DI	1011																																unused
RETI	1100																																unused
IN	1101							DR				0000																					addr20

7.2.1 Conditional Branching

Conditional branching in the RAMA-2200a ISA is provided via two instructions: the SKP (“skip”) instruction and the GOTO (“unconditional branch”) instruction. The SKP instruction compares two registers and skips the immediately following instruction if the comparison evaluates to true. If the action to be conditionally executed is only a single instruction, it can be placed immediately following the SKP instruction. Otherwise a GOTO can be placed following the SKP instruction to branch over to a longer sequence of instructions to be conditionally executed.

7.3 Detailed Instruction Reference

7.3.1 ADD

Assembler Syntax

ADD 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
0000				DR				SR1				unused																SR2			

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.

7.3.2 NAND

Assembler Syntax

NAND 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
0001				DR				SR1				unused																SR2			

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.

HINT: A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,

NAND DR, SR1, SR1

...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

7.3.3 ADDI

Assembler Syntax

ADDI DR, SR1, immval20

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
0010				DR				SR1				immval20																			

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.

7.3.4 LW

Assembler Syntax

LW DR, 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
0011				DR				BaseR				offset20																			

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.

7.3.5 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

$\text{MEM}[\text{BaseR} + \text{SEXT}(\text{offset20})] = \text{SR};$

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.

7.3.6 GOTO

Assembler Syntax

GOTO LABEL

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	0000	unused	PCoffset20																												

Operation

$\text{PC} = \text{PC} + \text{SEXT}(\text{PCoffset20});$

Description

The program unconditionally branches to the location specified by adding the sign-extended PCoffset20 field to the incremented PC (address of instruction + 1). **In other words, the PCoffset20 field specifies the number of instructions, forwards or backwards, to branch over.**

7.3.7 JALR

Assembler Syntax

JALR RA, AT

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
0110				RA				AT				unused																			

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.

7.3.8 HALT

Assembler Syntax

HALT

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
0111				unused																											

Description

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

7.3.9 SKP

Assembler Syntax

```
SKPE    SR1, SR2
SKPLT   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
1000				mode				SR1				unused																SR2			

mode is defined to be 0x0 for SKPE, and 0x1 for SKPLT.

Operation

```
if (MODE == 0x0) {
    if (SR1 == SR2) PC = PC + 1;
} else if (MODE == 0x1) {
    if (SR1 < SR2) PC = PC + 1;
}
```

Description

The SKP instruction compares the source operands SR1 and SR2 according to the rule specified by the mode field. For mode 0x0, the comparison succeeds if SR1 equals SR2. For mode 0x1, the comparison succeeds if SR1 is less than SR2.

If the comparison succeeds, the incremented PC (address of instruction + 1) is incremented again, for a resulting PC of (address of instruction + 2). **This effectively skips the immediately following instruction..** If the comparison fails, the program continues execution as normal.

7.3.10 LEA

Assembler Syntax

```
LEA     DR, label
```

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
1001				DR				unused				PCOffset20																			

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.

7.3.11 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
1010				unused																											

Operation

IE = 1;

Description

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

7.3.12 DI

Assembler Syntax

DI

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 = 0;

Description

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

7.3.13 RETI

Assembler Syntax

RETI

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																													
1100				unused																									

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.

7.3.14 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																													
1101				DR				0000				addr20																	

Operation

DAR = addr20;

DR = DeviceData;

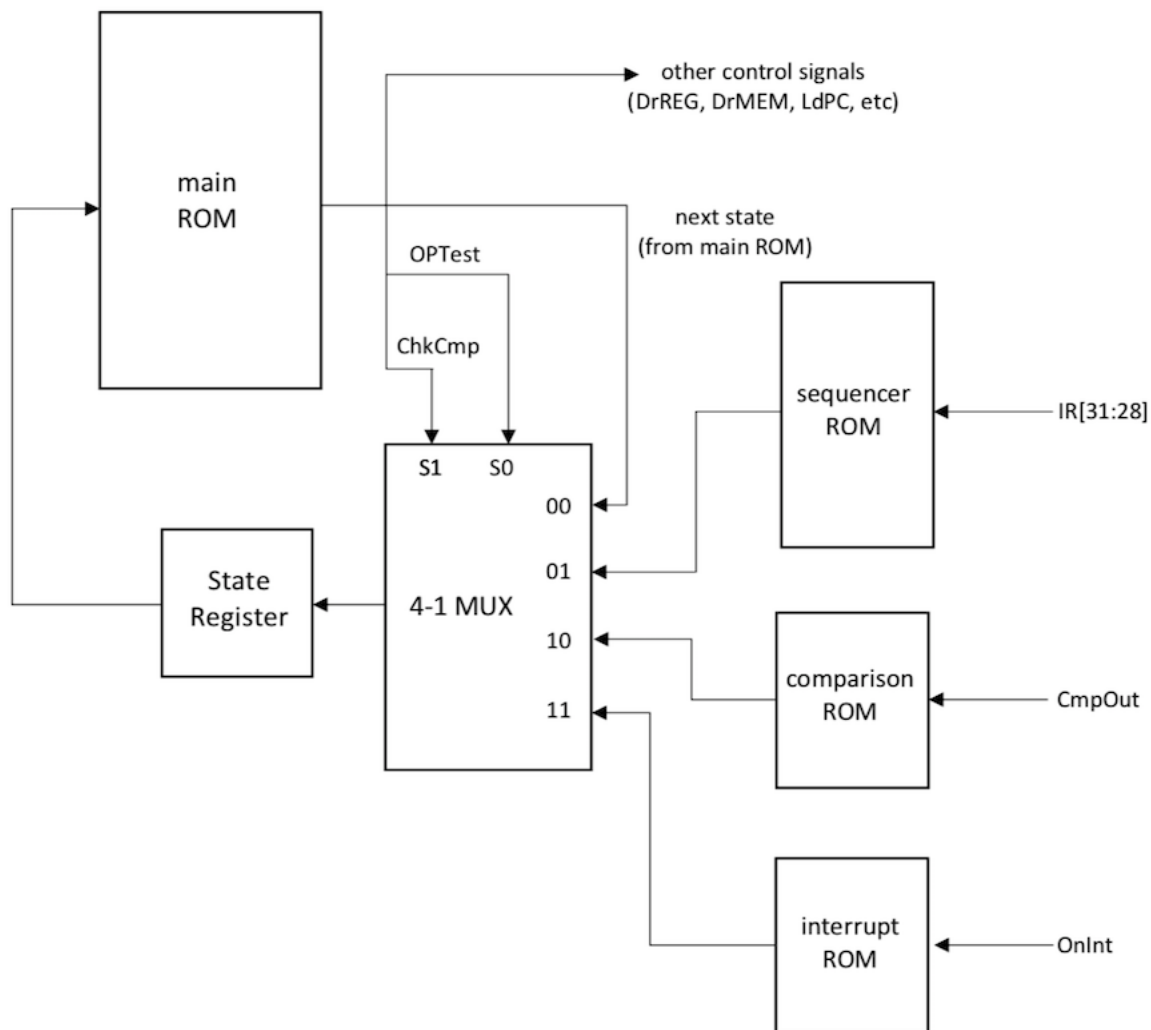
DAR = 0;

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.

8 Appendix B: Microcontrol Unit

As you may have noticed, we currently have an unused input on our multiplexer. This gives us room to add another ROM to control the next microstate upon an interrupt. You need to use this fourth ROM to generate the microstate address when an interrupt is signaled. The input to this ROM will be controlled by your interrupt enabled register and the interrupt signal asserted by the timer interrupt. This fourth ROM should have a 1-bit input and 6-bit output.



The outputs of the FSM control which signals on the datapath are raised (asserted). Here is more detail about the meaning of the output bits for the microcontroller:

Table 3: ROM Output Signals

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

Table 4: Register Selection Map

RegSelHi	RegSelLo	Register
0	0	RX (IR[27:24])
0	1	RY (IR[23:20])
1	0	RZ (IR[3:0])
1	1	\$k0