1 Introduction

We have spent the last few weeks implementing our 32-bit datapath. The simple 32-bit BOB-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 BOB-2200a enables the ability for programs to be interrupted. Your assignment is to fully implement and test interrupts using the provided datapath and Brandonsim. 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 Brandonsim. A copy of Brandonsim is available under Files on Canvas. In order to run Brandonsim, 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.
- Brandonsim is not perfect and does have small bugs. In certain scenarios, files have been corrupted and students have had to re-do the entire project. 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 BOB-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.

3 What We Have Provided

- A reference guide to the BOB-2200a is located in *Appendix A: BOB-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 Brandonsim library (`project2-devices.circ`) containing the timer device subcircuit you will use for this project. **To load the library into an existing Brandonsim file, use Project → Load Library → Logisim Library.**
- A new microcode spreadsheet template `microcode.xlsx` with additional columns for the new signals that will be added in this project. We've provided you a complete microcode that meets the requirements of Project 1, but feel free to supply your own.
- A timer device that will generate an interupt signal at specified intervals. The pinout and functionality of this device are described in *Adding External Timer 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* BOB-2200a datapath circuit (`BOB-2200a.circ`) that you may add the basic interrupt support onto will be provided **Tuesday, September 18**, after the one-time forgiveness period for Project 1 has passed. You are also free to build off of your own Project 1 datapath. Most of the work can be easily carried over from one datapath to another.

4 Implementing a Basic Interrupt

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

Work in the BOB-2200a.circ file. If you wish to use your existing datapath, make a copy with this name, and include the Timer subcircuit from the file 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. To ensure your INT line reads as low (i.e., '0') when no devices are requesting an interrupt, connect a pull-down resistor to the INT line (Brandonsim contains a component to do this).
3. When a device receives an **IntAck** signal, it will drive a 32-bit device ID onto the I/O data bus. To prevent misbehaving 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 **DrIO** 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.

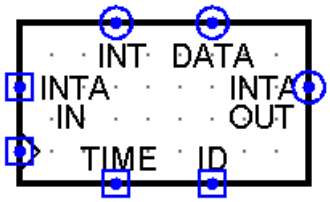
4.2 Adding External Timer Device

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 a external timer device to the datapath. And it should have a device ID of 0x1 and a 1000-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 Brandonsim.



- **INT**: The device will begin to assert this line when its time interval has elapsed. It will not be lowered until the device receives an INTA signal.
- **INTA IN** and **INTA OUT**: 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 in the same clock cycle, and lower its INT line in the next clock cycle. If the device receives an INTA signal while it has not asserted INT, it will pass the signal onto the next device through INTA OUT. This functionality can be used to daisy-chain devices.
- **ID** and **DATA**: The user may configure the device's ID through the ID pin. The device ID is passed to the DATA pin when the device receives an INTA signal after asserting the INT line.
- **TIME**: The user may configure the device's timer interval through this pin. The interval is specified in number of clock cycles. When the interval has elapsed, the device will raise the INT line.

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: BOB-2200a Instruction Set Architecture* and *Appendix B: Microcontrol Unit* and pay special attention to the new instructions.

In this part of the assignment you will modify the microcontroller and the microcode of the BOB-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) **DrIO** to drive the value on the I/O 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:

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 bus and use it to index into the interrupt vector table to retrieve the new PC value. The should be done in the same clock cycle as the IntAck assertion.

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: 0xFFFFFD
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. If it works correctly, you should see a location in memory increment as the program runs.

5 Deliverables

Please submit all of the following files in a **.tar.gz** archive generated by our 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. The generated archive should contain at a minimum the following files:

- BOB-2200a.circ
- microcode.xlsx
- assembly/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.

6 Appendix A: BOB-2200a Instruction Set Architecture

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

The BOB-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 24 bits on access, discarding the 8 most significant bits if the address was stored in a 32-bit register. This provides roughly 67 MB of addressable memory.

6.1 Registers

The BOB-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.

6.2 Instruction Overview

The BOB-2200a supports a variety of instruction forms. The instructions we will implement in this project are summarized below.

Table 2: BOB-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	SR1	unused SR2
NAND	0001	DR	SR1	unused SR2
ADDI	0010	DR	SR1	immval20
LW	0011	DR	BaseR	offset20
SW	0100	SR	BaseR	offset20
BNE	0101	SR1	SR2	offset20
JALR	0110	RA	AT	unused
HALT	0111	unused		
SLT	1000	DR	SR1	unused SR2
LEA	1001	DR	unused	offset20
EI	1010	unused		
DI	1011	unused		
RETI	1100	unused		

6.2.1 Conditional Branching

Conditional branching in the BOB-2200a ISA is provided via the BNE ("branch if not equal") instruction. BNE will branch to address "incrementedPC + offset20" only if SR1 and SR2 are not equal

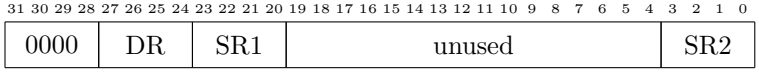
6.3 Detailed Instruction Reference

6.3.1 ADD

Assembler Syntax

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

Encoding



Operation

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

Description

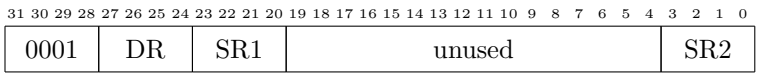
The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

6.3.2 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.

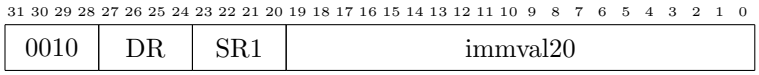
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

6.3.3 ADDI

Assembler Syntax

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

Encoding



Operation

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

Description

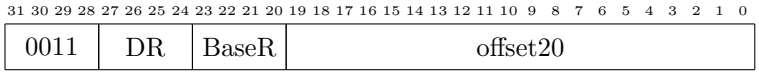
The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

6.3.4 LW

Assembler Syntax

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

Encoding



Operation

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

Description

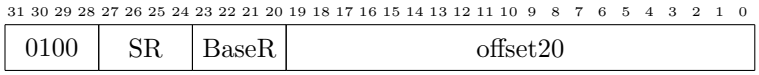
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

6.3.5 SW

Assembler Syntax

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

Encoding



Operation

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

Description

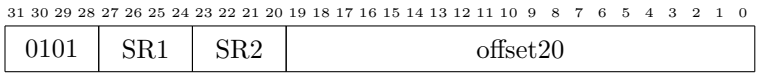
An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word obtained from register SR is then stored at this address.

6.3.6 BNE

Assembler Syntax

```
BNE    SR1, SR2, offset20
```

Encoding



Operation

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

Description

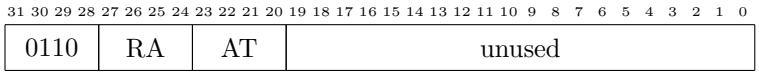
A branch is taken if SR1 and SR2 are not equal. If this is the case, the PC will be set to the sum of the incremented PC (since we have already undergone fetch) and the sign-extended offset[19:0].

6.3.7 JALR

Assembler Syntax

```
JALR   RA, AT
```

Encoding



Operation

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

Description

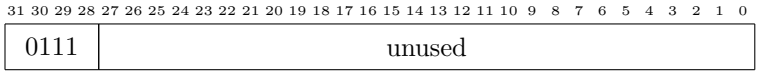
First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

6.3.8 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

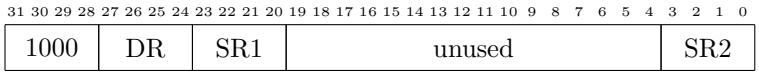
The machine is brought to a halt and executes no further instructions.

6.3.9 SLT

Assembler Syntax

```
SW    DR, SR1, SR2
```

Encoding



Operation

```
if (SR1 < SR2) {
    DR = 1
} else {
    DR = 0
}
```

Description

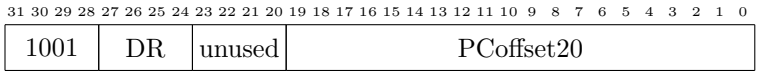
If SR1 is less than SR2, a 1 should be stored into DR. Otherwise a 0 should be stored in DR.

6.3.10 LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding



Operation

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

Description

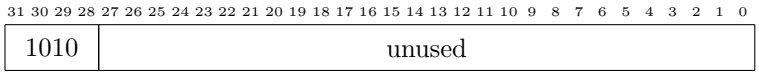
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the incremented PC (address of instruction + 1). It then stores the computed address into register DR.

6.3.11 EI

Assembler Syntax

```
EI
```

Encoding



Operation

```
IE = 1;
```

Description

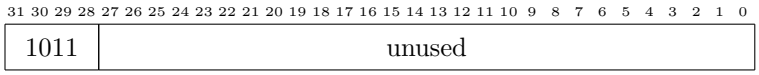
The Interrupts Enabled register is set to 1, enabling interrupts.

6.3.12 DI

Assembler Syntax

```
DI
```

Encoding



Operation

```
IE = 0;
```

Description

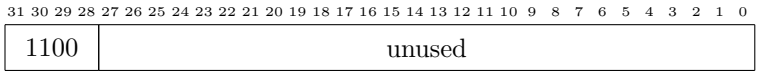
The Interrupts Enabled register is set to 0, disabling interrupts.

6.3.13 RETI

Assembler Syntax

```
RETI
```

Encoding



Operation

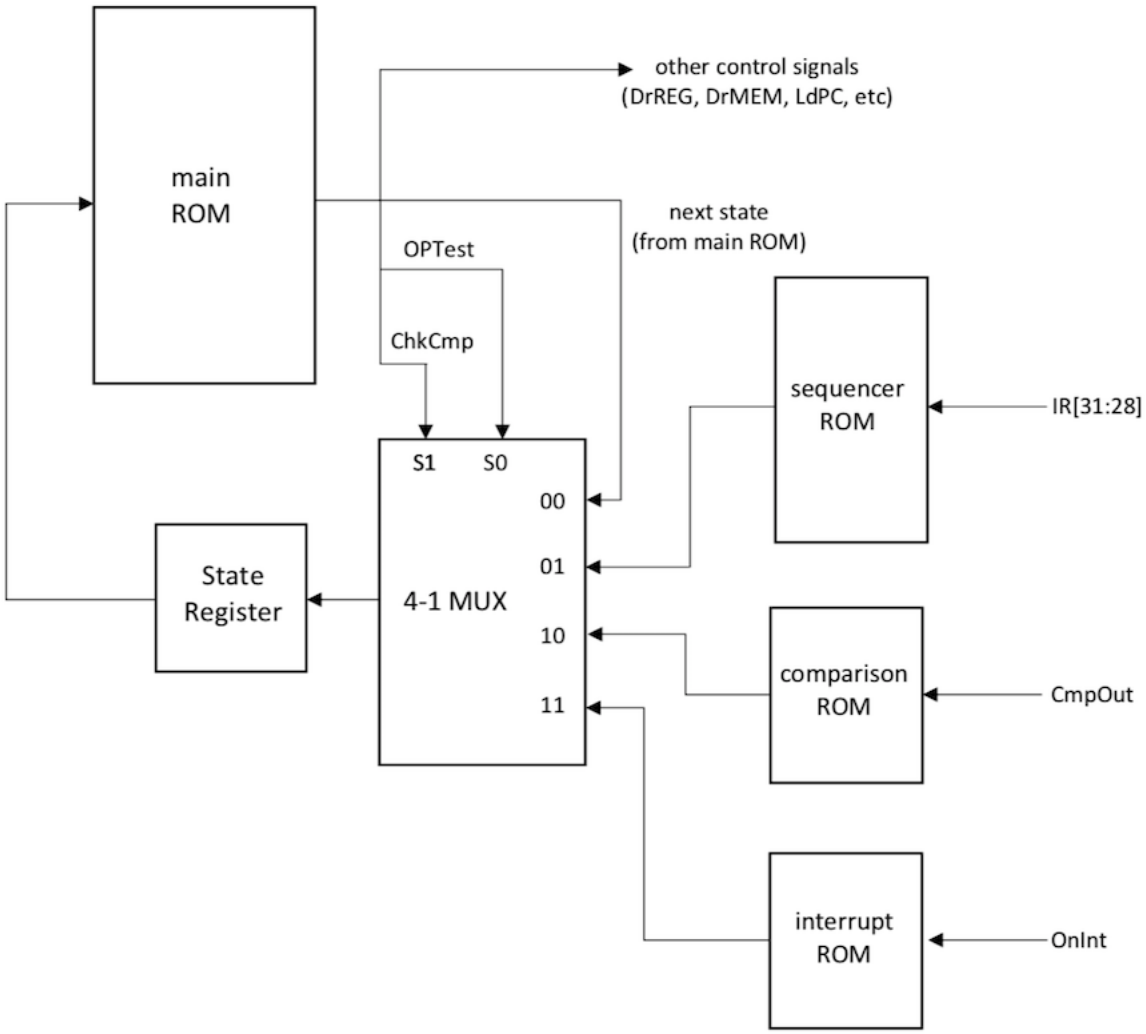
```
PC = $k0;
IE = 1;
```

Description

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

7 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 2-bit input and 6-bit output. The most significant input bit of the ROM should be set to 0.



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	EnInt
1	NextState[1]	8	DrALU	15	LdB	22	ALUHi	29	IntAck
2	NextState[2]	9	DrPC	16	LdCmp	23	OPTest	30	DrIO
3	NextState[3]	10	DrOFF	17	WrREG	24	ChkCmp		
4	NextState[4]	11	LdPC	18	WrMEM	25	DrCmp		
5	NextState[5]	12	LdIR	19	RegSelLo	26	CondType		
6	DrReg	13	LdMAR	20	RegSelHi	27	LdEnInt		

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