We have just spent the last few weeks implementing our 32-bit datapath. The simple 32-bit LC-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. Your assignment is to fully implement and test interrupts using the provided datapath and Brandonsim. In this project, you will add the additional support needed to make all this work properly. Your job will be to hook up the interrupt acknowledgment lines to the input devices, modify the datapath to support interrupt operations, and write an interrupt handler to increment a clock value at a designated memory address.

1 Requirements

- Part 1 - Add hardware support for handling interrupts.
- Part 2 - Add microcontroller support for handling interrupts.
- Part 3 - Code an interrupt handler for the clock.
- Download the proper version of Brandonsim. A copy of Brandonsim is available under Resources on T-Square.
- 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 git repository.

2 What We Have Provided

- A reference guide to the LC 2200-32a located in Appendix A: LC3-2200a Instruction Set Architecture. **PLEASE READ THIS FIRST BEFORE YOU MOVE ON!**
- An *INCOMPLETE* LC 2200-32a datapath circuit will be released February 14th for you to add the interrupt support onto. We are not releasing this until then due to the one-time-forgiveness 3 day extra submission period for project 1. You are free to use your own project 1 to build off of. Most of the work can be easily carried over from one datapath to another if you feel more comfortable with the provided circuit.
- An intergenerator subcircuit which will generate an interrupt signal every so often (this is also called the "clock", not to be confused with the clock circuit element).
- The complete microcode from project 1 will also be released February 14th for the same reasoning as above. You will have to add interrupt support to this. Once again, you are free to use your own project 1 microcode.
- An *INCOMPLETE* assembly program (prj2.s) to run on the datapath to test interrupt support.
- An assembler to assemble your interrupt handling program.

3 Initial Interrupt Hardware Support

For this part of the assignment, you need to add hardware support for interrupts to the LC2200-32a datapath from project 1.

You must do the following:

1. Create an Interrupt Enabled Register (**IE**) so we can keep track of whether or not interrupts are currently enabled or disabled.

2. Sometimes the processor might be busy when an interrupt is raised. *(Think of some cases that might cause this behavior for our simple processor.)* So, we need to keep track of whether or not an interrupt is pending. Additionally, we need to continue asserting it to the microcontroller until the processor responds with an **IntAck** signal to let the device know it is ready to handle the interrupt. *(What happens if we don't catch the interrupt? **Hint:** Consider how long (in clock cycles) the device is raising its interrupt. Analyze the intergenerator circuit if you need to.)* Once the INTA signal is received, the device should drive its device index onto the bus. Use 0x1 as the device index.

3. 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. Therefore, when you actually load in the code you wrote in part 2, 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.

4. Create hardware to support selecting the register $k0 within the microcode. This is needed by some interrupt related instructions. **HINT:** Use only the register selection bits that the main ROM already outputs to select $k0.

4 Microcontroller Interrupt Support

Before beginning this part, be sure you have read through Appendix A: LC3-2200a Instruction Set Architecture and Appendix A: Microcontroller Unit and pay special attention to the new instruction set.

In this part of the assignment you will modify the microcontroller and the microcode of the LC 2200-32a to support interrupts. You will need to to the following:

1. Be sure to read the appendix on the microcontroller before starting this section.

2. Modify the microcontroller to support asserting three 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.

3. Extend the size of the ROM accordingly.

4. Add the fourth ROM described in the appendix 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 an interrupt was raised.

   (b) If not, continue with FETCH normally.

   (c) If an interrupt was raised, then perform the following:

      i. Save the current PC to the register $k0.

      ii. Disable interrupts.

      iii. Assert the interrupt acknowledge signal (IntAck). Next, take the device index on the bus and use it to index into the interrupt vector table at (0x01) and retrive the new PC value. This new PC value should the be loaded into the PC. This second step can either be done during the same clock cycle as the IntAck assertion or the next clock cycle (depending on how you choose to implement the hardware support for this in part 1).

Note: `onInt` works in the same manner that ccMatch 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.

6. Implement the microcode for the 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.

5 Implementing the Interrupt Handler

Our datapath and microcontroller now fully support interrupts BUT we still need to implement an interrupt handler within prj2.s to support interrupts without interfering with the correct operation of any user programs. In prj2.s we provide you with a program that runs in the background. For part 3 of this project, you have to write an interrupt handler for the clock device (Intergenerator). 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), and the state of the interrupted program.
2. Enable interrupts (which should have been disabled implicitly by the processor in the FETCH macrostate).
3. Implement the actual work to be done in the handler. In the case of this project, we want you to **increment a clock variable in memory**.
4. Restore the state of the original program and return using RETI.

The handler you have written should run every time the clock interrupt is triggered. Even though there is only one interrupt for this project, the handler should be written such that interrupts can be nested (higher priority interrupts should be allowed while running handler). With that in mind, interrupts should be enabled for as long as possible within the handler. Furthermore, you will need to do the following:

1. Load the starting address of the handler into the interrupt vector table at address 0x00000001.
2. 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 some predetermined locations:
   (a) 0xFFFFFC for seconds
   (b) 0xFFFFFD for minutes
   (c) 0xFFFFFE for hours

   Assume that the clock interrupt fires every second.
3. Complete the two FIXMEs located in prj2.s. You should read through this file as it contains more information about this part of the project.

6 Deliverables

Please submit all of the following files in a *firstNamelastName***.tar.gz** archive (please replace *firstNamelastName* with you actual name...). You must turn in:

- prj2.s
- LC-2200-32a.circ
- microcode.xlsx

Don't forget to sign up for a demo slot! We will announce when these are available.

Precaution: You should always re-download your assignment from T-Square after submitting to ensure that all necessary files were properly uploaded.

7 Appendix A: Microcontroller 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 clock interrupt from the part 1. 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 1: ROM Output Signals

Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose
0	NextState[0]	7	DrMEM	14	LdA	21	ALULo
1	NextState[1]	8	DrALU	15	LdB	22	ALUHi
2	NextState[2]	9	DrPC	16	LdCC	23	OPTest
3	NextState[3]	10	DrOFF	17	WrREG	24	chkCC
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 2: Register Selection Map

RegSelHi	RegSelLo	Register
0	0	RX
0	1	RY
1	0	RZ
1	1	$k0

8 Appendix A: LC3-2200a Instruction Set Architecture

The LC3-2200a (Little Computer 3-2200 Type A) is a simple, yet capable computer architecture. The LC3-2200a combines attributes of both the LC3 architecture, and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.
The LC3-2200a 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.

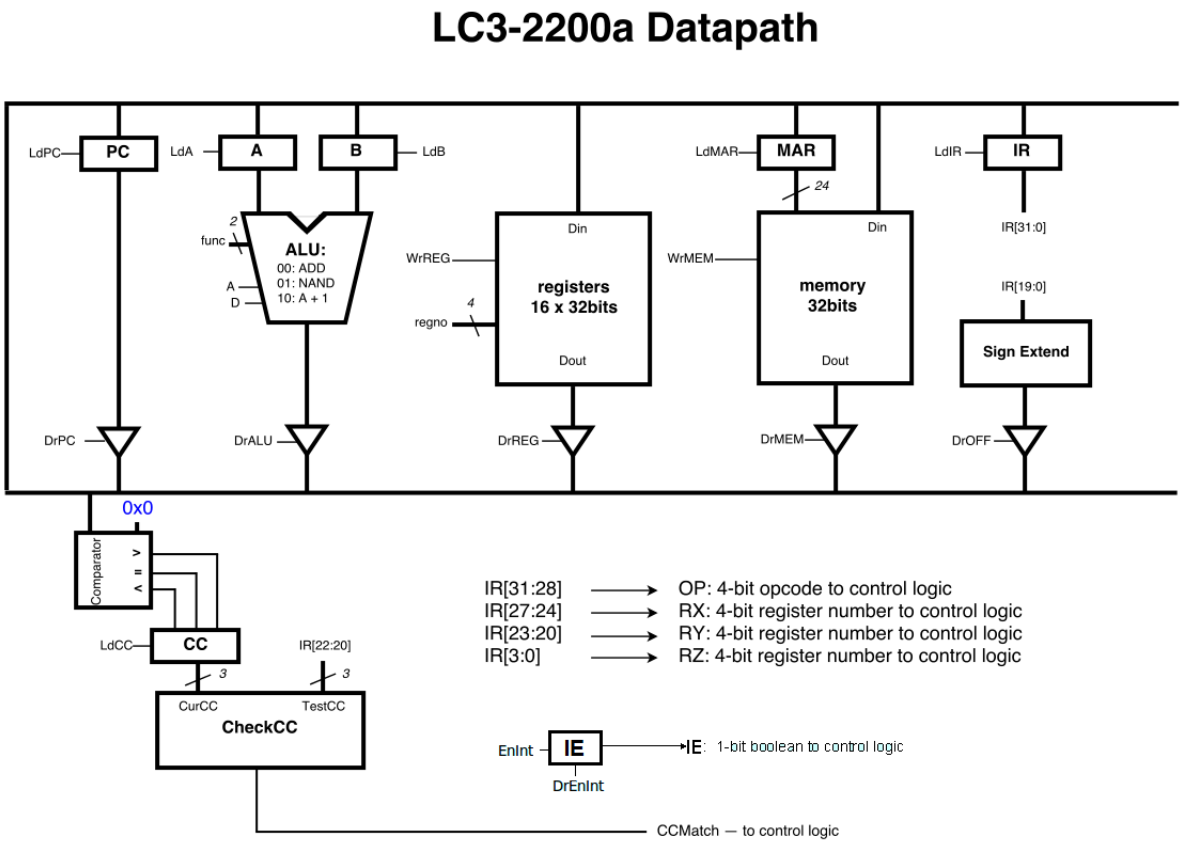


Figure 1: LC 2200-32 Datapath Diagram

8.1 Registers

The LC3-2200a 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 3: Registers and their Uses

Register Number	Name	Use	Callee Save?
0	$zero	Always Zero	NA
1	$at	Reserved for the Assembler	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 a general purpose register. You should not use it because the assembler will use it in processing pseudo-instructions.
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. You should use this to store the return address when an interrupt occurs
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. Don't worry about using this register.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing. It is automatically used for this purpose by the subroutine jump instruction.

8.2 Instruction Overview

The LC3-2200a supports a variety of instruction forms. The instructions we will implement in this project are summarized below.

Table 4: LC3-2200a 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
+ADDI	0001	DR	SR1	immval20
+NAND	0010	DR	SR1	unused SR2
BR	0011	0000	0 n z p	PCoffset20
JALR	0100	RA	AT	unused
+LW	0101	DR	BaseR	offset20
+LEA	0110	DR	unused	PCoffset20
SW	0111	SR	BaseR	offset20
EI	1000	unused		
DI	1001	unused		
RETI	1010	unused		
HALT	1111	unused		

NOTE: Instructions marked with + modify condition codes.

8.3 Condition Codes

In addition to the general-purpose registers, the LC3-2200a also keeps state in the form of the **condition codes** register. These flags are updated by any instruction that writes to a destination register (except JALR). If a signed interpretation of the value is negative, the **N - negative** flag is set. If zero, the **Z - zero** flag is set. If positive, the **P - positive** flag is set.

The condition codes can be tested by the **BR - conditional branch** instruction. See the specification of the branch instruction in Section 8.4.4 for details.

When the system first starts up, the condition codes are indeterminate. Therefore, at least one instruction that modifies the condition codes should be executed before attempting a branch.

8.4 Detailed Instruction Reference

8.4.1 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;
SetConditionCodes();
```

Description
The ADD instruction adds the source operand obtained from SR2 to the source operand obtained from SR1. The result is stored in DR.
The condition codes are set, based on whether the result is negative, zero, or positive.

8.4.2 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
0001	DR	SR1	immval20

Operation

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

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 second source operand is added to the first source operand, and the result is stored in DR.
The condition codes are set, based on whether the result is negative, zero, or positive.

8.4.3 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
0010	DR	SR1	unused	SR2

Operation

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

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.
The condition codes are set, based on whether the result is negative, zero, or positive.

> **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: $DR \leftarrow \overline{SR1}$.

8.4.4 BR

Assembler Syntax

```
BRn    LABEL        BRnz   LABEL
BRz    LABEL        BRzp   LABEL
BRp    LABEL        BRnp   LABEL
BR     LABEL        BRnzp  LABEL      (BR is equivalent to BRnzp)
```

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	0000	0	n	z	p	PCoffset20

Operation

```
if ((n AND N) OR (z AND Z) OR (p AND P)) {
    PC = PC + SEXT(PCoffset20);
}
```

Description
The condition codes specified by the state of bits [22:20] are tested. If bit [22] is set, N is tested; if bit [22] is clear, N is not tested. If bit [21] is set, Z is tested, etc. If any of the condition codes tested is set, the program 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.**

8.4.5 JALR

Assembler Syntax

```
JALR   AT, RA
```

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	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.

8.4.6 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
0101	DR	BaseR	offset20

Operation

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

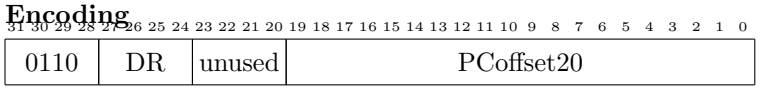
Description
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

8.4.7 LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding



Operation

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

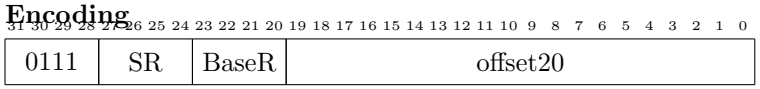
Description
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the incremented PC (address of instruction + 1). This instruction effectively performs the same computation as the BR instruction, but rather than performing a branch, merely stores the computed address into register DR. The condition codes are set, based on whether the result is negative, zero, or positive.

8.4.8 SW

Assembler Syntax

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

Encoding



Operation

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

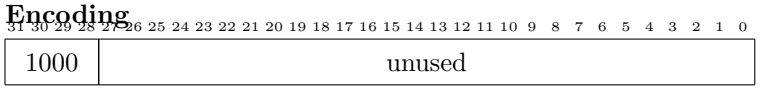
Description
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the contents of the register specified by bits [23:20]. The 32-bit word obtained from register SR is then stored at this address.

8.4.9 EI

Assembler Syntax

```
EI
```

Encoding



Operation

```
IE = 1;
```

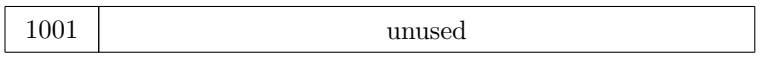
Description
The Interrupts Enabled register is set to 1, enabling interrupts.

8.4.10 DI

Assembler Syntax

```
DI
```

Encoding



Operation

```
IE = 0;
```

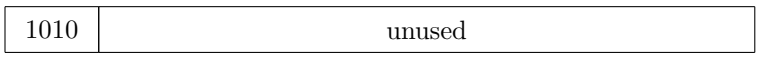
Description
The Interrupts Enabled register is set to 0, disabling interrupts.

8.4.11 RETI

Assembler Syntax

```
RETI
```

Encoding



Operation

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

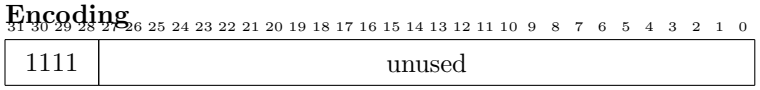
Description
PC is restored to the return address stored in $k0. The Interrupts Enabled register is set to 1, enabling interrupts.

8.4.12 HALT

Assembler Syntax

```
HALT
```

Encoding



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

8.5 Interrupts Support

Note that some items mentioned in this section are *not* implemented yet. The implementation is part of your assignment for this project. You must handle the following:

1. Memory Mappings
   (a) For the purposes of this assignment, we have chosen to keep the interrupt vector table to be located at address 0x00000000. It can store 16 interrupt vectors. Program memory starts at 0x00000010.
2. Hardware Timer
   (a) The hardware timer will fire every so often. You should configure it as device 1 - it should place the assigned index (its driver is located on the vector table) onto the bus when it receives an IntAck signal from the processor.