We have just spent the last few weeks implementing our 16-bit datapath. The simple 16-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. We have invested a great deal of time and money into developing a more powerful LC-2200 computer. This one is 32-bit and supports interrupts.The only trouble is that the interrupts support does not appear to be completed. We are quite disappointed by this, and so it is your assignment to fully implement and test interrupts using the provided datapath and Logisim.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 acknowledge lines to the input devices, modify the datapath to support interrupt operations, and write an interrupt handler to increment the 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.
- Extra Credit - Implement some bonus instructions!

2 What We Have Provided

- A reference guide to the LC 2200-32 located in Appendix B: LC 2200-32 Processor Reference Manual. **PLEASE READ THIS FIRST BEFORE YOU MOVE ON!**
- An *INCOMPLETE* LC 2200-32 datapath circuit. You'll need to add interrupt support for this.
- An intergenerator subcircuit which will generate an interrupt signal every so ofter (this is also called the "clock").
- The complete microcode from project 1, to which you will have to add interrupt support.
- The compiler for the microcode.
- An *INCOMPLETE* assembly program (prj2.s) to run on the datapath to test interrupt support.
- An assembler to assemble you interrupt handling program.
- An example assembly program to demonstrate the use of the optional bonus instructions (example.s).

3 Initial Interrupt Hardware Support

For this part of the assignment, you need to add hardware support for interrupts to the LC2200-32 datapath. You have been provided with a completed LC2200-32 datapath from project 1. Keep in mind this is a 32-bit implementation; the LC2200 in project 1 was a 16-bit implementation.

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. Connect the intergenerator (the clock) to the datapath and make the following modifications:
   (a) Signal the microcontroller when an interrupt is recieved.
   (b) Continue to signal the interrupt until it recieves the **IntAck** signal.

(c) Once IntAck is recieved, it should drive its index on to the bus in either the same or the next clock cycle. Use 0x1 for the device index.

(d) At the very least, you will need to build hardware to store the interrupt signal, in case it is not immediately acknowledged and hardware that will drive the device index on to the bus once IntAck is received (in the same or next clock cycle).

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.

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 B: LC 2200-32 Processor Reference Manual 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 2200-32 to support interrupts. You will need to to the following:

1. Be sure to read the appendix on the microcontroller and look at its implementation in the LC-2200-32.circ file as there are instructions and guidance for what needs to be supported.

2. Modify the microcontroller to support asserting three new signals:

   (a) LdEnInt & DrEnInt to control whether interrupts are enabled/disabled and,

   (b) IntAck to send an interrupt acknowledge to the device.

3. Extend the size of the ROM accordingly.

4. Fully hook up the microcontroller to the datapath before continuing (our version is not 100% hooked-up).

5. Add the fourth ROM described in the appendix to handle onInt

6. 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: To do this new conditional in the FETCH macrostate, we have supplied you with a new attribute of the state tag called `onInt`. `onInt` works in the same manner that onZ did in project 1. The processor should banch 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.

7. 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 0x000001.
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) 0xFFFFC for seconds
   (b) 0xFFFFD for minutes
   (c) 0xFFFFE 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 Extra Credit - Implement Bouns Instructions

For extra credit, you may implement one or more non-trivial instructions. The assembler will accept BONI, BONJ, BONR, and BONO as valid instructions. You need to implement at least one of them and demonstrate its use during your demo. See the LC 2200-32 Processor Reference Manual for details about the instructions. Rememeber that example.s shows some usage of examples. Your tools for creating the microcode will support these new instructions, although **you will have to define some microcode for the bonus instructions even if you don't implement them!!!**

7 Deliverables

Please submit all of the following files in a **.zip** or **.tar.gz** archive. You must turn in:

- prj2.s
- LC-2200-32.circ
- microcode_*lastName*.xml (please replace *lastName* with you actual last name...)

Don't forget to sign up for a demo slot! We will announce when these are available. Failure to demo results in a 0!

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

8 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	LdZ	23	OPTest
3	NextState[3]	10	DrOFF	17	WrREG	24	chkZ
4	NextState[4]	11	LdPC	18	WrMEM	25	LdEnInt
5	NextState[5]	12	LdIR	19	RegSelLo	26	DrEnInt
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

Table 3: ALU Function Map

ALUHi	ALUlLo	Function
0	0	ADD
0	1	NAND
1	0	A - B
1	1	A + 1

Reminder: Logisim implements the typical edge-triggered logic used in modern digital circuits. This means that stateful devices only change state when the clock makes a 0 to 1 transition.

This note pertains to the microsequencer implementation of the control logic. NOTE: Logisim has a minimum of two address bits for a ROM, even though only one address bit is needed for the OnZ ROM and the new interrupt ROM. You may want to do something so that the high address bit for these two ROMs are permanently set to zero.

9 Appendix B: LC 2200-32 Processor Reference Manual

The LC-2200-32 is a 32-bit computer with 16 general registers plus a separate program counter (PC) register. All addresses are word addresses. Register 0 is wired to zero: it always reads as zero and writes to it are ignored. There are four types of instructions: R-Type (Register Type), I-Type (Immediate value Type), J-Type (Jump Type), and O-Type (Other Type).

Here is the instruction format for R-Type instructions (ADD, NAND):

Bits	31 - 28	27 - 24	23 - 20	19 - 4	3 - 0
Purpose	opcode	RX	RY	Unused	RZ

Here is the instruction format for I-Type instructions (ADDI, LW, SW, BEQ):

Bits	31 - 28	27 - 24	23 - 20	19 - 0
Purpose	opcode	RX	RY	2's Complement Offset

Here is the instruction format for J-Type instructions (JALR):

Bits	31 - 28	27 - 24	23 - 20	19 - 0
Purpose	opcode	RX	RY	Unused (all 0s)

Here is the instruction format for S-Type instructions (HALT, EI, DI, RETI):

Bits	31 - 28	27-0
Purpose	opcode	Unused (all 0s)

Table 4: 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 temporary values. **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 calee (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 used to handle interrupts.
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 finsihed executing. It is only supposed to be used by the JALR (Jump And Link Register) instruction.

Table 5: Assembly Language Instruction Descriptions

Name	Type	Example	Opcode	Action
add	R	add $v0, $a0, $a2	0000	Add contents of RY with the contents of RZ and store the result in RX.
nand	R	nand $v0, $a0, $a2	0001	NAND contents of RY with the contents of RZ and store the result in RX.
addi	I	addi $v0, $a0, 7	0010	Add contents of RY to the contents of the offset field and store the result in RX.
lw	I	lw $v0, 0x07($sp)	0011	Load RX from memory. The memory address is formed by adding the offset to the contents of RY.
sw	I	sw $a0, 0x07($sp)	0100	Store RX into memory. The memory address is formed by adding the offset to the contents of RY.
beq	I	beq $a0, $a1, done	0101	Compare the contents of RX and RY. If they are the same, then branch to address PC + 1 + Offset, where PC is the address of the beq instruction. **Memory is word addressed**.
jalr	J	jalr $at, $ra	0110	First store PC + 1 in RY, where PC is the address of the jalr instruciton. Then branch to the address in RX. If RX = RY, then the processor will store PC + 1 into RY and end up branching to PC + 1.
halt	O	halt	0111	Tells the processor to halt.
bonr	R	bonr $a0, $a1, $a2	1000	Optional bonus R-Type instruction
bono	O	bono	1001	Optional bonus O-Type instruction
ei	O	ei	1010	Enable Interrupts
di	O	di	1011	Disable Interrupts
reti	O	reti	1100	Return from interrupt by loading address stored in $k0 into the PC and then enabling interrupts.
boni	I	$a0, (0x01)$a1	1101	Optional bonus I-Type instruction.
bonj	J	$a1, $a2	1110	Optional bonus J-Type instruction.

Finally, the assembler supports pseudo-operations. These operations aren't actually supported by the ISA, but the assembler will produce the appropriate instructions to get the desired instructions.

Table 6: Assembly Language Pseudo-Instructions

Name	Type	Example	Opcode	Action
noop	Pseudo-Op	noop	N/A	No operation, this does nothing. It actually just spits out "add $zero, $zero, $zero".
.word	Pseudo-Op	.word 32	N/A	Fill the word at the current location with some value.
la	Pseudo-Op	la $sp, stack	N/A	Loads the address of the label into the register. It actually spits out "addi $sp, $zero, label"

The provided datapath follows the following diagram:
Here is a description of each datapath component:

1. **PC** is the program counter register.
2. **Z** is the zero detection register (used for deciding if a branch should occur).
3. **ALU** performs four functions: ADD, NAND, SUBTRACT, and INCREMENT.
4. **Register Fiile** stores the 16 32-bit registers.
5. **IE** is the interrupts enabled register (interrupts on = 1, interrupts off = 0).
6. **MAR** memory address register (address to read/write from memory).
7. **IR** instruction register, which contains the 32-bit instruction currently being executed.
8. **Sign Extender** extends the 20 bit offset in the IR to 32-bit values suitable for driving on to the bus.

9.1 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 0x000000. It can store 16 interrupt vectors. Program memory starts at 0x000010.
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.

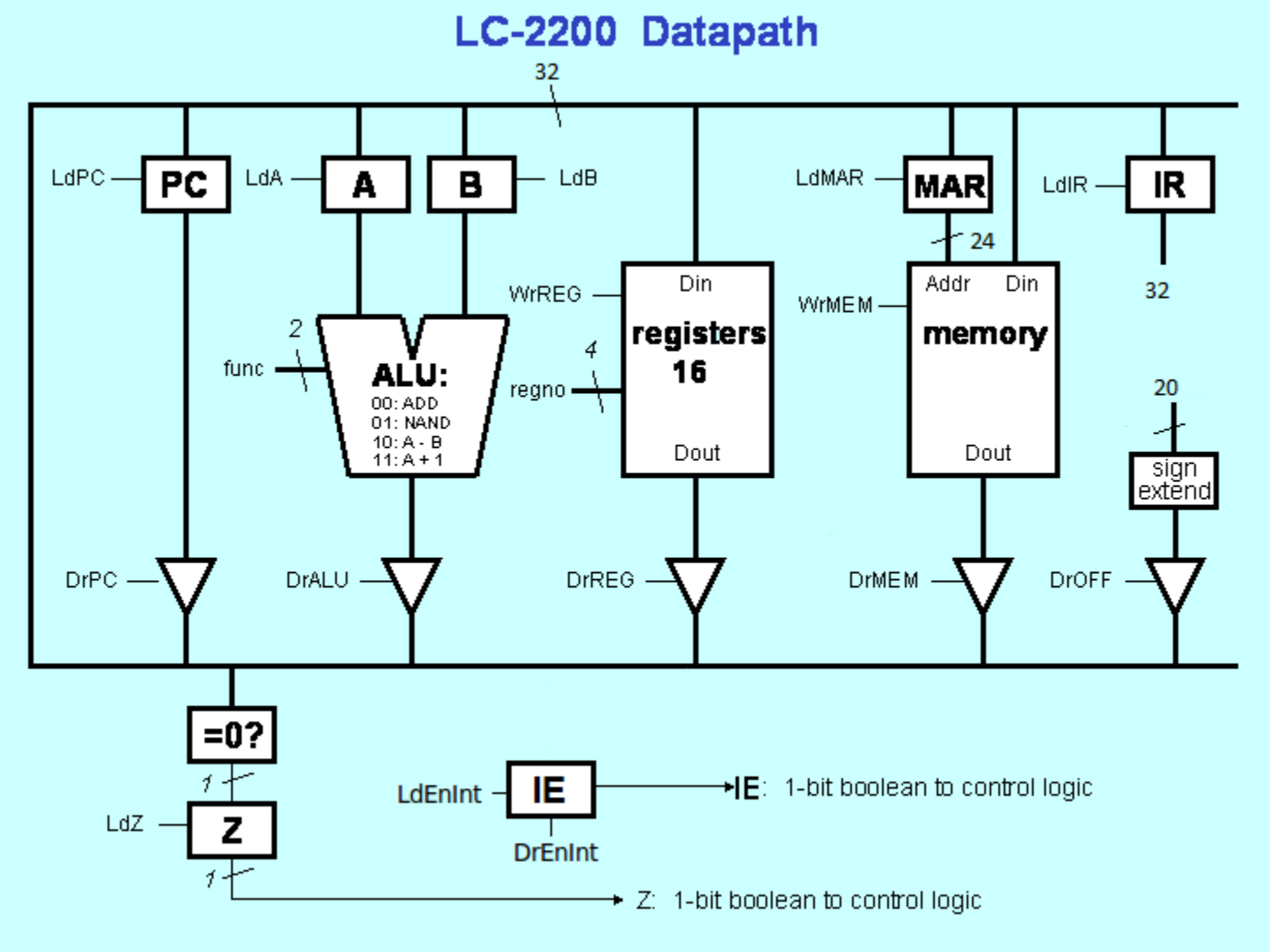

Figure 1: LC 2200-32 Datapath Diagram