Project Description

Purpose:

To give you a deeper understanding of the design, structure and operations of a computer system, principally focusing on the ISA and how it is executed. In addition, we will also focus on memory structure and operations, and simple I/O capabilities.

Component	Description
I: Basic Machine	Design and implement the basic machine architecture. Implement a simple memory Build initial user interface to simulator
II: Memory and Cache Design	Design and implement the modules for enhanced memory and cache operations Extend the user interface. Demonstrate 1st program running on the simulator.
III: Execute All Instructions	Make sure all instructions (as specified below) execute on the simulator Demonstrate 2nd program running on the simulator
IV: Floating Point and Vector Operations	Design and implement the modules for floating point and vector operations and simple pipelining; extend the user interface

Our Computer

Our computer is a small classical CISC computer. This does not match any real computer. Rather it is a contrived example to get you to think about how to execute certain kinds of instructions. In doing so, it will make you think about the macro-structure of the CPU, e.g., what the programmer sees and the micro-structure, e.g., those components the programmer does not see. Note that some of the components the programmer (operator) might be able to see are not accessible by instructions.

It has the following characteristics – for Phase I:

- 4 General Purpose Registers (GPRs) – each 16 bits in length
- 3 Index Registers – 16 bits in length
- 16-bit words
- Memory of 2048 words, expandable to 4096 words
- Word addressable

Instructions are aligned on a word boundary. Words must be fetched on a word boundary.

The four GPRs are numbered 0-3 and can be mnemonically referred to as R0 – R3. They may be used as accumulators. The index registers are mnemonically referred to as X1 or X2 or X3.

The CPU has other registers:

Mnemonic	Size	Name
PC	12 bits	Program Counter: address of next instruction to be executed. Note that $2^{12} = 4096$.
CC	4 bits	Condition Code: set when arithmetic/logical operations are executed; it has four 1-bit elements: overflow, underflow, division by zero, equal-or-not. They may be referenced as cc(0), cc(1), cc(2), cc(3). Or by the names OVERFLOW, UNDERFLOW, DIVZERO, EQUALORNOT
IR	16 bits	Instruction Register: holds the instruction to be executed
FR	16 bits	Floating Point Register
MAR	16 bits	Memory Address Register: holds the address of the word to be fetched from memory
MBR	16 bits	Memory Buffer Register: holds the word just fetched from or the word to be /last stored into memory
MSR	16 bits	Machine Status Register: certain bits record the status of the health of the machine
MFR	4 bits	Machine Fault Register: contains the ID code if a machine fault after it occurs
X1…X3	16 bits	Index Register: contains a base address that supports base register addressing of memory.

Assume characters are represented in ASCII.

Reserved Locations:

Memory Address	Usage
0	Reserved for the Trap instruction for Part III.
1	Reserved for a machine fault (see below).
2	Store PC for Trap
3	Not Used
4	Store PC for Machine Fault
5	Not Used

Interrupts:
There are no interrupts in this machine. We will not be simulating interrupts or complex I/O devices in this project.

Machine Fault: ***(How to set the MFR)***
An erroneous condition in the machine will cause a machine fault. The machine traps to memory address 1, which contains the address of a routine to handle machine faults. Our simulator checks for faults.

The possible machine faults that are predefined are:

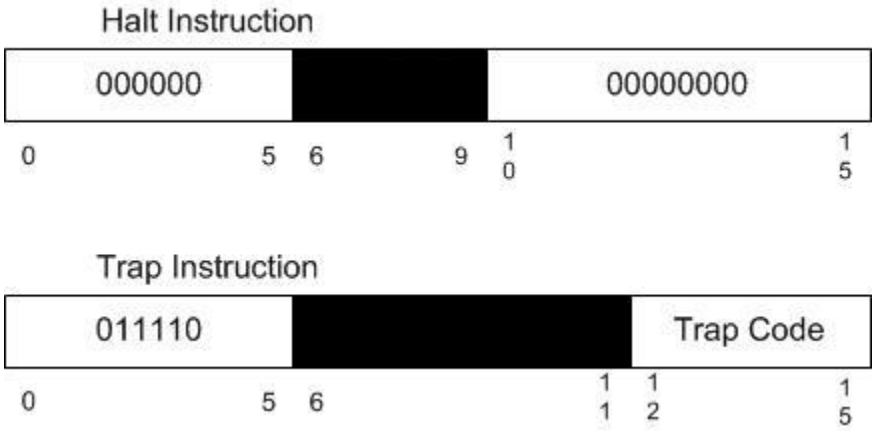
ID	Fault
0	Illegal Memory Address to Reserved Locations ***MFR set to binary 0001***

1 Illegal TRAP code ***MFR set to binary 0010***
2 Illegal Operation Code ***MFR set to 0100***
3 Illegal Memory Address beyond 2048 (memory installed) ***MFR set to binary 1000***

When a Trap instruction or a machine fault occurs, the processor saves the current PC and MSR saves (stored with MFR register) contents to the locations specified above, then fetches the address from Location 0 (Trap) or 1 (Machine Fault) into the PC which becomes the next instruction to be executed. This address will be the first instruction of a routine which handles the trap or machine fault.

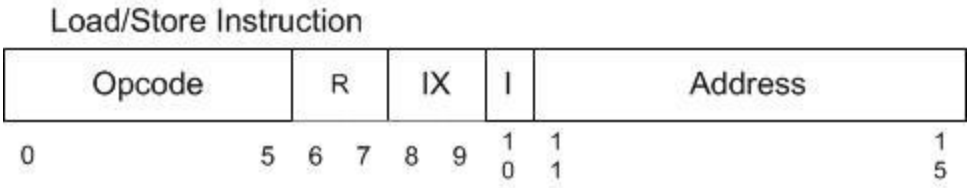
The following sections describe the instructions that we simulate.

Miscellaneous Instructions:
Miscellaneous instructions do not fit into another category (given the size of the machine). The formats are:



OpCode$_8$	Instruction	Description
000	HLT	Stops the machine.
036	TRAP code	Traps to memory address 0, which contains the address of a table in memory. Stores the PC+1 in memory location 2. The table can have a maximum of 16 entries representing 16 routines for user-specified instructions stored elsewhere in memory. Trap code contains an index into the table, e.g. it takes values 0 – 15. When a TRAP instruction is executed, it goes to the routine whose address is in memory location 0, executes those instructions, and returns to the instruction stored in memory location 2. The PC+1 of the TRAP instruction is stored in memory location 2.

Load/Store Instructions:
The basic instruction format is shown below:



Field	#Bits	Description
Opcode	6	Specifies one of 64 possible instructions; Not all may be defined in this project
IX	2	Specifies one of three index registers; may be referred to by X1 – X3. O value indicates no indexing.
R	2	Specifies one of four general purpose registers; may be referred to by R0 – R3
I	1	If I =1, specifies indirect addressing; otherwise, no indirect addressing.
Address	5	Specifies one of 32 locations

To address all of memory, indexing will be required. We will use a base address indexing scheme.

The value of IX is used to select an index register and to specify indirect addressing:

00	No Indexing
01	Index Register 1
10	Index Register 2
11	Index Register 3

Computing the Effective Address:

Effective Address (EA) =
if I field = 0:
// NO indirect addressing
If IX = 00, EA = contents of the Address field
// just indexing
if IX = 1..3, c(Xj) + contents of the Address field, where j = c(IX)
// that is, the IX field has an index register number, the contents of which are added to the contents of the address field
if I field = 1:
// indirect addressing, but NO indexing
if IX = 00, c(Address)
// both indirect addressing and indexing
if IX = 1..3, c(c(Xj) + Address), where j = c(IX)

Load/Store instructions move data from/to memory and a register. The access to memory may be indirect (by setting the I bit).

Notation:
c(EA) means "fetch the contents of the memory location specified by EA", where EA = 0 … maximum memory size, or
c(IX) means "fetch the contents of the field IX in the instruction".

[,I] means "the indirect bit should be set."

What happens if the EA is greater than maximum memory size (as specified above)??

Note that the [,I] is optional at the end. So the instruction might not have the ,I at the end.

$OpCode_8$	Instruction	Description
01	LDR r, x, address[,I]	Load Register From Memory, r = 0..3 r <– c(EA) r <- c(c(EA)), if I bit set
02	STR r, x, address[,I]	Store Register To Memory, r = 0..3 Memory(EA) <– c(r)
03	LDA r, x, address[,I]	Load Register with Address, r = 0..3 r <– EA
41	LDX x, address[,I]	Load Index Register from Memory, x = 1..3 Xx <- c(EA)
42	STX x, address[,I]	Store Index Register to Memory. X = 1..3 Memory(EA) <- c(Xx)

As an example, consider the instruction: LDR 3,0,31 (Symbolic Form)
This would be read as: Load register 3 with the contents of the memory location 31. Since IX = 00, there is no indexing, so 31 is the EA.

This instruction would be encoded as:

Opcode	R	IX	I	Address
000001	11	00	0	11111

Transfer Instructions:
The Transfer instructions change control of program execution. Conditional transfer instructions test the value of a register. Note R = 0…3. They have the same format as the Load/Store instructions.

Notation: c(r) means "the contents of register r", r = 0..3

$OpCode_8$	Instruction	Description
10	JZ r, x, address[,I]	Jump If Zero: If c(r) = 0, then PC <– EA Else PC <- PC+1

11	JNE r, x, address[,I]	Jump If Not Equal: If c(r) != 0, then PC <–- EA Else PC <- PC + 1
12	JCC cc, x, address[,I]	Jump If Condition Code cc replaces r for this instruction cc takes values 0, 1, 2, 3 as above and specifies the bit in the Condition Code Register to check; If cc bit = 1, PC <– EA Else PC <- PC + 1
13	JMA x, address[,I]	Unconditional Jump To Address PC <- EA, Note: r is ignored in this instruction
14	JSR x, address[,I]	Jump and Save Return Address: R3 <– PC+1; PC <– EA R0 should contain pointer to arguments Argument list should end with –1 (all 1s) value
15	RFS Immed	Return From Subroutine w/ return code as Immed portion (optional) stored in the instruction's address field. R0 <– Immed; PC <– c(R3) IX, I fields are ignored.
16	SOB r, x, address[,I]	Subtract One and Branch. R = 0..3 r <– c(r) – 1 If c(r) > 0, PC <- EA; Else PC <- PC + 1
17	JGE r,x, address[,I]	Jump Greater Than or Equal To: If c(r) >= 0, then PC <- EA Else PC <- PC + 1

OpCode 016 allows you to support simple loops. I like this instruction. I first encountered it on the Data General Eclipse S/200.

Arithmetic and Logical Instructions:
Arithmetical and Logical instructions perform most of the computational work in the machine.

For immediate instructions, the Address portion is considered to be the Immediate value. The condition codes are set for the arithmetic operations. The maximum absolute value of the Immediate value is 32 (5 bits).

$OpCode_8$	**Instruction**	**Description**
04	AMR r, x, address[,I]	Add Memory To Register, r = 0..3 r<– c(r) + c(EA)
05	SMR r, x, address[,I]	Subtract Memory From Register, r = 0..3 r<– c(r) – c(EA)
06	AIR r, immed	Add Immediate to Register, r = 0..3 r <– c(r) + Immed

		Note: 1. if Immed = 0, does nothing 2. if c(r) = 0, loads r with Immed IX and I are ignored in this instruction
07	SIR r, immed	Subtract Immediate from Register, r = 0..3 r <– c(r) - Immed Note: 1. if Immed = 0, does nothing 2. if c(r) = 0, loads r1 with –(Immed) IX and I are ignored in this instruction

As an example, add to r2 the contents of memory location 523.

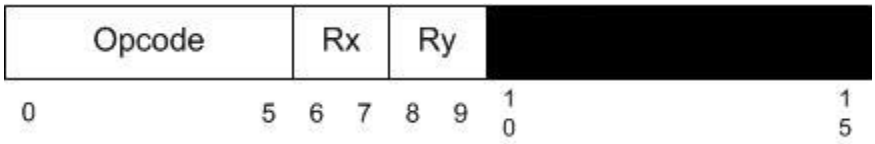
ADD 2,1,23 where c(X1) = 500

Transfer the immediate value 10 to register 3 so that the value of register 3 is 10.
But, register 3 may already have something in it!

```
STR 3,0,20        ; store register 3 contents in location 20
SMR 3,0,20        ; clear register 3!
AIR 3,10          ; load register 3 with 10
```

How do we test for overflow? Underflow? How do you know this occurs?
Hennessey and Patterson discuss this.

Certain arithmetic and logical instructions are register to register operations. The format of these instructions is:



The blacked out portion means that portion of the instruction is ignored. Rx and Ry refer to one of R0-R3.

OpCode$_8$	**Instruction**	**Description**
20	MLT rx,ry	Multiply Register by Register rx, rx+1 <- c(rx) * c(ry) rx must be 0 or 2 ry must be 0 or 2 rx contains the high order bits, rx+1 contains the low order bits of the result Set OVERFLOW flag, if overflow
21	DVD rx,ry	Divide Register by Register rx, rx+1 <- c(rx)/ c(ry) rx must be 0 or 2 rx contains the quotient; rx+1 contains the remainder

		ry must be 0 or 2 If c(ry) = 0, set cc(3) to 1 (set DIVZERO flag)
22	TRR rx, ry	Test the Equality of Register and Register If c(rx) = c(ry), set cc(4) <– 1; else, cc(4) <– 0
23	AND rx, ry	Logical And of Register and Register c(rx) <– c(rx) AND c(ry)
24	ORR rx, ry	Logical Or of Register and Register c(rx) <– c(rx) OR c(ry)
25	NOT rx	Logical Not of Register To Register C(rx) <– NOT c(rx)

The logical instructions perform bitwise operations.

TRR 0,2 where r0 = 0 000 000 000 000 001 and r2 = 0 000 000 000 000 001.
Then, cc(4) <- 1 indicating equality

NOT 3 where r3 = 1 000 000 000 110 110
Then r3 = 0 111 111 111 001 001

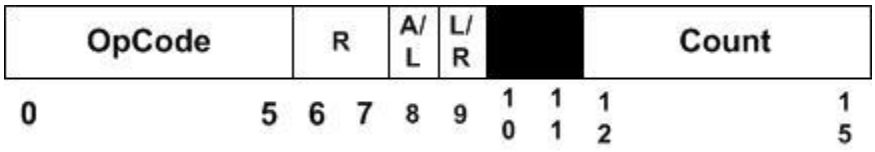
Shift/Rotate Operations
Shift and Rotate instructions manipulate a datum in a register.

Arithmetic Shift (A/L = 0) instructions move a bit string to the **right** or **left**, with excess bits discarded (although one or more bits might be preserved in flags). The sign bit is not shifted in this instruction.

Logical Shift (A/L = 1) instructions move a bit string **lef**t or **right**, with excess bits discarded and zero(es) inserted at the opposite end.

Logical Rotate (A/L = 1) instructions are similar to shift instructions, except that rotate instructions are circular, with the bits shifted out one end returning on the other end. Rotates can be to the left or right.

The format for shift and rotate instructions is:



Note: We have 16-bit words, but the maximum value for Count can be 16. So, what happens when the Count is specified to be 15?

OpCode	Instruction	Description
31	SRC r, count, L/R, A/L	Shift Register by Count c(r) is shifted left (L/R =1) or right (L/R = 0) either

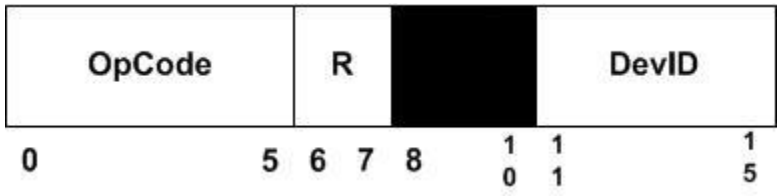
		logically (A/L = 1) or arithmetically (A/L = 0) XX, XXX are ignored Count = 0…15 If Count = 0, no shift occurs
32	RRC r, count, L/R, A/L	Rotate Register by Count c(r) is rotated left (L/R = 1) or right (L/R =0) either logically (A/L =1) XX, XXX is ignored Count = 0…15 If Count = 0, no rotate occurs

On arithmetic shifts to the right, the sign bit is replicated in the position 1 for each shift. There's a lot going on here with these instructions. These are exemplars of some early machines which packed a lot of functionality into a few instructions.

So, suppose r3 = 0 000 000 000 000 110
Then, SRC 3,3,1,1 would yield r0 = 0 000 000 000 110 000
e.g., shift left bits 13…15

So, suppose r1 = 1 000 000 000 000 110
Then, SRC 1,2,0,0 would yield r1 = 1 100 000 000 000 001
e.g., shift right 2 bits
And underflow would be set. Why?

I/O Operations
I/O operations communicate with the peripherals attached to the computer system. This is a really simple model of I/O meant to give you a flavor of how I/O works. For character I/O, the instruction format is:



OpCode	Instruction	Description
61	IN r, devid	Input Character To Register from Device, r = 0..3
62	OUT r, devid	Output Character to Device from Register, r = 0..3
63	CHK r, devid	Check Device Status to Register, r = 0..3 c(r) <- device status

We will assume the devices whose DEVIDs are:

DEVID	Device
0	Console Keyboard

1 Console Printer
2 Card Reader
3-31 Console Registers, switches, etc

Notes:
(1) You may only use the IN and CHK instructions with the console keyboard and the card reader.
(2) You may only use the OUT and CHK instruction with the console printer.
(3) Devices 3 – 31 are affected only by the IN and OUT opcodes. Some of these devices may be affected by only one of these opcodes. *Can you think of an example now?*

The console printer is implemented as a pop-up window to which you can only print. The console keyboard is implemented as a text field into which you can type characters and numbers.

Floating Point Instructions/Vector Operations:

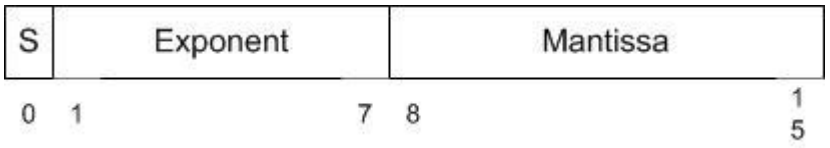
We have limited space in our instruction set, with only six bits for opcodes. So, we have to limit our floating point and vector operations. This will give you a chance to think about how to write a software routine to do multiplication and division for both floating point numbers.

There are two floating point registers: FR0 and FR1. Each is 16 bits in length.

The format of a floating point number is the same as that for a load/store instruction, except that the r field takes only 2 values: 0 or 1 to specify the two floating point registers.

Vector operations are performed memory to memory. This was used on several models of vector processors as opposed to using lots of expensive registers to hold vectors (unless you were Seymour Cray).

Floating Point numbers are 16 bits in length. So, a floating point number has the representation:



There are 7 bits for the exponent and 8 bits for the mantissa. The first bit of the exponent is its sign bit. The S bit (bit 0) is the sign of the entire floating point number. The exponent ranges from –63 to 64, e.g., -2^6-1 to 2^6.

OpCode	Instruction	Description
33	FADD fr, x, address[,I]	Floating Add Memory To Register c(fr) <– c(fr) + c(EA)

		c(fr) <– c(fr) + c(c(EA)), if I bit set fr must be 0 or 1. OVERFLOW may be set
34	FSUB fr, x, address[,I]	Floating Subtract Memory From Register c(fr) <– c(fr) - c(EA) c(fr) <– c(fr) - c(c(EA)), if I bit set fr must be 0 or 1 UNDERFLOW may be set
35	VADD fr, x, address[,I]	Vector Add fr contains the length of the vectors c(EA) or c(c(EA)), if I bit set, is address of first vector c(EA+1) or c(c(EA+1)), if I bit set, is address of the second vector Let V_1 be vector at address; Let V_2 be vector at address+1 Then, $V_1[i] = V_1[i] + V_2[i]$, i = 1, c(fr).
36	VSUB fr, x, address[,I]	Vector Subtract fr contains the length of the vectors c(EA) or c(c(EA)), if I bit set is address of first vector c(EA+1) or c(c(EA+1)), if I bit set is address of the second vector Let V_1 be vector at address; Let V_2 be vector at address+1 Then, $V_1[i] = V_1[i] - V_2[i]$, i = 1, c(fr).
37	CNVRT r, x, address[,I]	Convert to Fixed/FloatingPoint: If F = 0, convert c(EA) to a fixed point number and store in r. If F = 1, convert c(EA) to a floating point number and store in FR0. The r register contains the value of F before the instruction is executed.
50	LDFR fr, x, address [,i]	Load Floating Register From Memory, fr = 0..1 fr <– c(EA), c(EA+1) fr <- c(c(EA), c(EA)+1), if I bit set
51	STFR fr, x, address [,i]	Store Floating Register To Memory, fr = 0..1 EA, EA+1 <– c(fr) c(EA), c(EA)+1 <- c(fr), if I-bit set

Note: Opcode 037 is a strange beast! It latches the result to FR0 when converting from integer to floating point – no other choices allowed!

So, the vector add instruction might be encoded as:

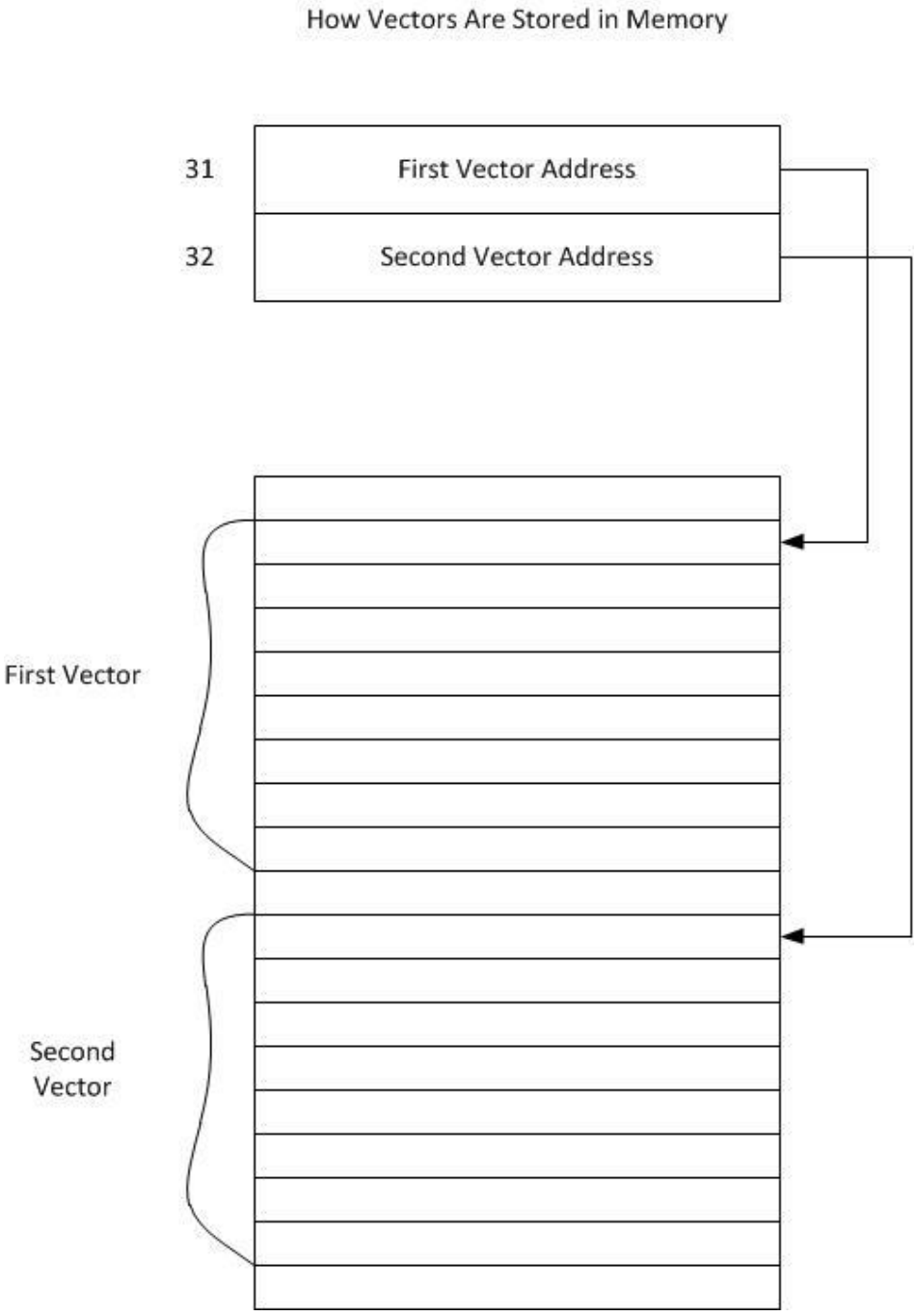
VADD 0, 1, 31 w/ I = 0

In memory this would look like:

Opcode	fr	I	IX	Address
011110	00	0	01	11111

R field designates either FR0 or FR1.

At memory location c(X0) + 31: address of first vector
At memory location c(X0) + 32: address of second vector
Each of these vectors would be c(fr) words long



Description of Our Computer

The computer system that you will develop a simulator for is a classic instruction set architecture modeled after, but not equivalent to any known CISC machines. We are examining a CISC machine so that you get to experience some of the tradeoff decisions that computer designers make.

1. General Properties

Our computer is a 16-bit processor that will eventually accommodate both fixed point and floating point arithmetic operations.

2. Instruction Set Architecture

The instruction set architecture (ISA) consists of 64 possible instructions. There are several instruction formats as depicted above. However, not all instructions are defined. What happens if you try to execute an opcode for an undefined instruction? A machine fault!

3. Specification

In this project you will design, implement, and test a simulator to simulate a basic machine. There are five elements to this process.

3.1 Central Processor

Your CPU simulator should implement the basic registers, the basic instruction set, a simple ROM Loader, and the elements necessary to execute the basic instruction set.

You will need a ROM that contains the simple loader. When you press the IPL button on the console, the ROM contents are read into memory and control is transferred to the first instruction of the ROM Loader program. The ROM can be either a file on your computer or just an array of instructions in your program.

Your ROM Loader should read the boot program from the ROM and place it into memory in a location you designate. The ROM Loader then transfers control to the program which executes until completion or error.

If your program completes normally, it returns to the boot program to read the next program (at this point your simulation should stop with PC having the value of the first address of the boot program). Returning to the boot program means that it prompts the user to either run the currently loaded program again or to load a new program and run it.

NOTE: Thus, the first address of your program should be greater than the length of your program.

I suggest you load the boot program beginning at location octal 10 and the first address of your program at octal address = octal 10 + length of boot program + octal 10.

If the program encounters an error, your program should display an error message on the console printer and stop.

If an internal error is detected, display an error code in the console lights and stop. But, you should consider handling the error in your system by generating a machine fault.

3.2 Simple Memory

In this project, you should design and implement a single port memory in Part I

Upon powering up your system, all elements of memory should be set to zero.

Your memory simulation should accept an address from the MAR on one cycle. It should then accept a value in the MBR to be stored in memory on the next cycle or place a value in the MBR that is read from memory on the next cycle.

But, remember your machine can have up to 2048 words maximum! What considerations must you make?

3.3 Simple Cache

You will, in part II, implement a simple cache, which sits between memory and the rest of the processor.

The cache is just a vector having a format similar to that described in the lecture notes. It should be a fully associative, unified cache.

What do you need to do?
a. What are the fields of the cache line?
b. Use a simple FIFO algorithm to replace cache lines.
c. How many cache lines? With 2048 words, probably 16 cache lines is enough.
d. Need to think about how to demonstrate caching works.

HINT: When you run your simulator, you may want to write a trace file of things that are happening inside your simulator. You can use this to help debug your simulator.

REMEMBER: More trace data is always useful!!

3. 4 User Interface

You should design a user interface that simulates the console of the CS6461 Computer. The UI should include both the console plus some additional capabilities to support the debugging of your simulator. I will give you some examples of consoles in a later lecture.

Remember that later phases will add more instructions and more complexity to the computer system and will result in additional displays and switches on your operator console. So, plan accordingly and allow some growth space as you make your initial design.

You should consider displaying registers which are not programmer-accessible, but are required for correct operation of the computer (and your simulator).

3.3.1 Operators Console

Your operators console should include:

- Display for all registers
- Display for machine status and condition registers
- An IPL button (to start the simulation)
- Switches (simulated as buttons) to load data into registers, to select displays, and to initiate certain conditions in the machine.

We will assume that when you start up the simulator that your computer is powered on. If you want to simulate a "Power" light, that is OK.

Some suggestions for switches and displays that you might want to consider are:

Displays:
Current Memory Address
Various Registers (as mentioned above)
You may wish to think about some sense switches that the user can inform the program. You have ample device IDs to accommodate these. One DEVID accesses one sense switch.

Switches:
Run, Halt, Single Step, the IPL button, switches to load the registers,

3.3.2 Field Engineers Console

Your field engineer's console design and contents are left up to you. As the simulator designer, you will understand the structure of your machine best, so you will know what additional data and switches you will need to diagnose your simulator.

For example, you may want to display the contents of internal registers within your simulated CPU. The operator doesn't need to see these, but the operator or field engineer certainly would when he or she is debugging the machine.

Test Programs:

Program 1: A program that reads 20 numbers (integers) from the keyboard, prints the numbers to the console printer, requests a number from the user, and searches the 20 numbers read in for the number closest to the number entered by the user. Print the number entered by the user and the number closest to that number. Your numbers should not be 1…10, but distributed over the range of 0 … 65,535. Therefore, as you read a character in, you need to check it is a digit, convert it to a number, and assemble the integer.

Program 2: A program that reads a set of a paragraph of 6 sentences from a file into memory. It prints the sentences on the console printer. It then asks the user for a word. It searches the paragraph to see if it contains the word. If so, it prints out the word, the sentence number, and the word number in the sentence.