George Washington University – Dept. of Computer Science
CS6461: Computer Architectures

Class 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.

Components:

The class project is structured into four segments of increasing difficulty that build towards a detailed understanding of the internal design of computer systems and a fairly complex simulation of a computer system.

Each segment is due to the grader at about 3 week intervals (see Syllabus).

The four components are:

Component	Description
I: Basic Machine 	Design and implement the basic machine architecture. Implement a simple memory Execute Load and Store instructions 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 your simulator.
III: Execute All Instructions	Make sure all instructions (as specified below) execute on your simulator
IV: ***DO 1 OF***: A. Floating Point and Vector Operations B: Enhanced Scheduling	Design and implement the modules for floating point and vector operations and simple pipelining; extend the user interface Design and implement simple branch prediction and speculative execution, trap if an error occurs to an error handling routine.

Programming Language:

You will program this simulator in Java. Use JDK 1.7 or later.
You will deliver a JAR file that I can run to test your simulator.
You will also deliver two programs in machine code which you have written based on specifications I will give you.
You will deliver the source code of your simulator

You will deliver a written report describing how to operate your simulator

There are NO exceptions to the language!

All of the above are to be submitted through Blackboard via the group that you belong to.

Tools:

I strongly recommend that you use an IDE as a development medium for your Java programs. Some examples: NetBeans, Eclipse, Bluej.

You may obtain BlueJ, a simple IDE, from the following web site:

http://www.bluej.org/download/download.html

You should download and read the documentation:

Tutorial
http://www.bluej.org/tutorial/tutorial-201.pdf
Unit Testing Tutorial
http://www.bluej.org/tutorial/testing-tutorial.pdf
Reference Manual
http://www.bluej.org/download/files/bluej-ref-manual.pdf
Installation Instructions
http://www.bluej.org/download/install.html

You may also want to purchase the book, but this is NOT required. If you do, get the 5th edition. In general, use of Bluej is intuitive. I recommend the book if you have not done object-oriented programming before.

Documentation:

Good documentation is absolutely essential to any project. During your design process for your simulator, you should keep good design notes. A compilation of design notes from each team member **must** be turned in with each segment.

You should also write a brief description (1-3 pages) of how to operate your simulator and explain features and operation of your operator's console. This will be updated for each segment and submitted with that segment of the project.

You may also implement a field engineer's console for extra credit, but your simulator must be completely working to receive extra credit.

Documentation extends to the software you write for the simulator.

COMMENTS are GOOD in CODE!!
LOTS of COMMENTS are BETTER!!
LOTS AND LOTS of COMMENTS are the BEST of ALL!!

Therefore, I expect to see lots of them in your code. More importantly, part of the evaluation of your simulator is *how well your code is commented* so that I can understand what you are doing.

CS6461 Computer

Our class 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 20 bits in length
- 3 Index Registers – 13 bits in length
- 20-bit words
- Memory of 2048 words, expandable to 8 KWords
- Word addressable

Instructions must be 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	13 bits	Program Counter: address of next instruction to be executed
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(1), cc(2), cc(3), cc(4). Or by the names OVERFLOW, UNDERFLOW, DIVZERO, EQUALORNOT
IR	20 bits	Instruction Register: holds the instruction to be executed
MAR	13 bits	Memory Address Register: holds the address of the word to be fetched from memory
MBR	20 bits	Memory Buffer Register: holds the word just fetched from or stored into memory
MSR	20 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	13 bits	Index Register: contains a 13-bit base address that supports base register addressing of memory.

You will need some other registers in your simulator. I am not going to tell you what they are; you must decide what they are, name them, and specify their function in your documentation.

Assume characters are represented in ASCII.

Reserved Locations:

Memory Address	Usage
0	Reserved for the Trap instruction for Part II.
1	Reserved for a machine fault (see below).
2,3	Store PC, MSR for a Trap instruction
4,5	Store PC, MSR for a Machine Fault

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

Machine Fault (Part II):
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. The ID of the fault is stored in register MFR. Your simulator must check for faults.

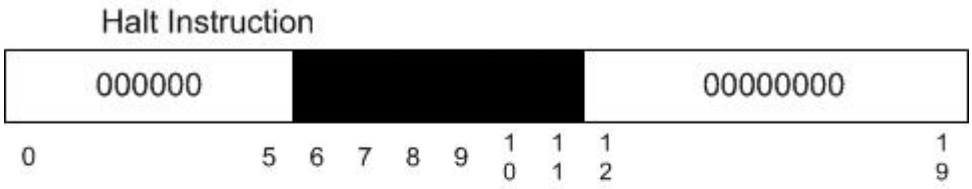
The possible machine faults that are predefined are:

ID	Fault
0	Illegal Memory Address
1	Illegal TRAP code
2	Illegal Operation Code

When a Trap instruction or a machine fault occurs, the processor saves the current PC and MSR 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 you must simulate.

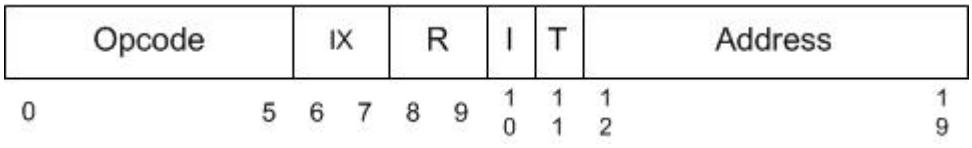
Miscellaneous Instructions:
Miscellaneous instructions do not fit into another category (given the size of the machine). The formats are:



$OpCode_8$	Instruction	Description
000	HLT	Stops the machine.
030	TRAP code	Traps to memory address 0, which contains the address of a table in memory. The table can have a maximum of 16 entries representing 16 routines for user-specified instructions stored elsewhere in memory. Trap code (in Address field) contains an index into the table, e.g. it takes values 0 – 15. When a TRAP instruction is executed, it goes to the routine, executes those instructions, and returns to the instruction after the TRAP instruction. If the value of code is greater than 15, a machine fault occurs.

Do not implement instruction 030 until Part III of the project!! This will allow you to create a few instructions in case I missed any.

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
R	2	Specifies one of four general purpose registers; may be referred to by R0 – R3
I	1	If I =1, specifies indirect addressing
T	1	If T = 1; specifies that you trace this instruction
Address	8	Specifies one of 256 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:

The values of IX are:

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

Computing the Effective Address:

Effective Address (EA) = if I = 0:
    if IX = 00, Address
    if IX = 1..3, c(Xj) + Address, where j = c(IX)
if I = 1:
    if IX = 00, c(Address) // indirect addressing
    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 … 8192.
[,I] means “the indirect bit should be set.”
[,T] means “the trace bit should be set.”

What happens if the EA is greater than 8192??

$OpCode_8$	**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 EA <– c(r) c(c(EA)) <- c(r), if I-bit set
03	LDA r, x, address[,I]	Load Register with Address, r = 0..3

		r <– EA r <– c(EA), if I bit set
41	LDX x, address[,I]	Load Index Register from Memory, x = 1..3 X0 <- c(EA)
42	STX x, address[,I]	Store Index Register to Memory. X = 1..3 EA <- c(X0) C(EA) <- c(X0), if I-bit set

As an example, consider the instruction: LDR 3,0,54. This would be read as: Load register 3 with the contents of the memory location 54. Since IX = 0, there is no indexing, so 54 is the EA.

This instruction would be encoded as:

Opcode	**I**	**IX**	**AC**	**Address**
000001	**0**	**0**	**11**	**101100**

Transfer Instructions:
The Transfer instructions change control of program execution. Conditional transfer instructions test the value of a register. Note r = 0…3.

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

$OpCode_8$	**Instruction**	**Description**
010	JZ r, x, address[,I]	Jump If Zero: If c(r) = 0, then PC <– EA or c(EA), if I bit set; Else PC <- PC+1
011	JNE r, x, address[,I]	Jump If Not Equal: If c(r) != 0, then PC <–- EA or c(EA) , if I bit set; Else PC <- PC + 1
012	JCC cc, x, address[,I]	Jump If Condition Code cc replaces r for this instruction cc takes values 0, 1, 2, 3 as above If cc bit = 1, PC <– EA or c(EA), if I bit set; Else PC <- PC + 1
013	JMP x, address[,I]	Unconditional Jump To Address PC <- EA, if I bit not set; PC <– c(EA), if I bit set Note: r is ignored in this instruction
014	JSR x, address[,I]	Jump and Save Return Address: R3 <– PC+1; PC <– EA; PC <- c(EA), if I bit set R0 should contain pointer to arguments Argument list should end with –17777 value
015	RFS Immed	Return From Subroutine w/ return code as Immed portion (optional) stored in R0's address field. R0 <– Immed; PC <– c(R3) IX, I fields are ignored.
016	SOB r, x, address[,I]	Subtract One And Branch. R = 0..3

		r <– c(r) – 1 If c(r) > 0, PC <- EA; but PC <– c(EA), if I bit set; Else PC <- PC + 1
017	JGE r,x, address[,I]	Jump Greater Than or Equal To: If c(r) >= 0, then PC <- EA or c(EA) , if I bit set; 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 value of the Immediate value is 256 (8 bits).

$OpCode_8$	**Instruction**	**Description**
004	AMR r, x, address[,I]	Add Memory To Register, r = 0..3 r<– c(r) + c(EA)
005	SMR r, x, address[,I]	Subtract Memory From Register, r = 0..3 r<– c(r) – c(EA)
006	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
007	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 563.

ADD 2,1,63 where c(X1) = 500

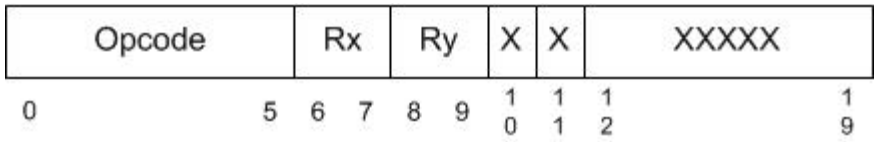
Transfer the immediate value 10 to register 3 so that the value of register 3 is 10.
But, register 3 may already have something in it!

```
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:



"XXXXX" means that portion of the instruction is ignored. Rx and Ry refer to one of R0-R3.

$OpCode_8$	Instruction	Description
020	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
021	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)
022	TRR rx, ry	Test the Equality of Register and Register If c(rx) = c(ry), set cc(4) <– 1; else, cc(4) <– 0
023	AND rx, ry	Logical And of Register and Register c(rx) <– c(rx) AND c(ry)
024	ORR rx, ry	Logical Or of Register and Register c(rx) <– c(rx) OR c(ry)
025	NOT rx	Logical Not of Register To Register C(rx) <– NOT c(rx)

The logical instructions perform bitwise operations.

TRR 0,2 where r0 = 00 000 000 000 000 000 001 and r2 = 00 000 000 000 000 000 001.
Then, cc(4) <- 1 indicating equality

NOT 3 where r3 = 10 000 000 000 000 110 110
Then r3 = 01 111 111 111 111 001 001

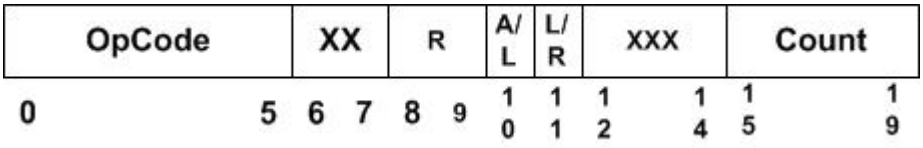
Shift/Rotate Operations
Shift and Rotate instructions manipulate a 20-bit datum in a register.

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:



OpCode	Instruction	Description
031	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…20 If Count = 0, no shift occurs
032	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…20 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.

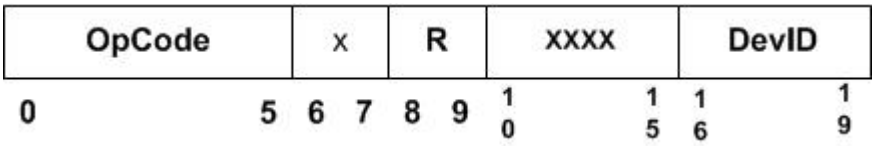
So, suppose r3 = 00 000 000 000 000 000 110
Then, SRC 3,3,1,1 would yield r0 = 00 000 000 000 000 110 000
e.g., shift left bits 17…19

So, suppose r1 = 10 000 000 000 000 000 110
Then, SRC 1,2,0,0 would yield r1 = 10 100 000 000 000 000 001
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
061	IN r, devid	Input Character To Register from Device, r = 0..3
062	OUT r, devid	Output Character to Device from Register, r = 0..3
063	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
3-16	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?*

You will simulate the card reader by reading from a file on disk.

Your simulator will have a GUI. It should have a pane that simulates the console printer and a pane that simulates the console keyboard.

Floating Point Instructions/Vector Operations:

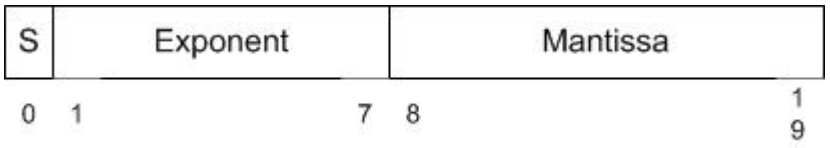
Do not implement floating point numbers until Part IV

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 20 bits.

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 20 bits in length. So, a floating point number has the representation:



There are six bits for the exponent and 13 bits for the mantissa. The exponent ranges from -15 to 16.

OpCode	Instruction	Description
033	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
034	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
035	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).
036	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).
037	CNVRT r, x, address[,I]	Convert to Fixed/FloatingPoint: If x = 0, convert c(EA) to a fixed point number and store in r. If x = 1, convert c(EA) to a floating point number and store in FR0. The x field is used to store the value of F.
40	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
41	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	I	IX	R	Address
011110	0	1	00	011001

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(R0) words long

There is a lot for you to think about here! We will discuss in lectures 9 and 10.

Description of the CS6461 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 20-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?

3. Specification

In this project you will design, implement, and test a simulator to simulate a basic machine. There are 4 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.

Your ROM Loader should read a boot program from a *virtual card reader* and place them into memory in a location you designate. The ROM Loader then transfers control to the program which executes until completion or error. The virtual card reader is implemented as a file.

If your program completes normally, it returns to the boot program to read the next program (at this point your simulation should stop). 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.

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.

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.

For purposes of this project think of memory as a 2-D array consisting of 8 banks of 256 words each. So, you will need to decompose the contents of the MAR into a bank identifier and a specific word within the bank. But, remember your machine can have up to 8 KWords maximum! What considerations must you make?

3.3 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 of field engineer certainly would when he or she is debugging the machine.

Test Programs:

You need to write two programs using the instructions in the instruction set and demonstrate that they execute using your simulator.

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 1 … 32,767. 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 10 words from the console keyboard into memory. It prints the set of 10 words on the console printer. It then sorts the 10 words and prints out the sorted list. It then requests a sequence of characters from the user and searches the 10 words to see if they contain the sequence. For each word containing the sequence, it prints out the word containing it.

Part I Deliverable: (Part I – 5 points)

Your simulator, packaged as a JAR file.
Simple documentation describing how to use your simulator, what the console layout is and how to operate it.
Your team's design notes
Source code – well documented.

You should be able to enter data into any of R0 – R3, enter data into memory via switches, enter AMR, SMR, AIR, or SIR instructions into memory, enter address into PC and press Single Step switch to execute the instruction at that address.

Part II Deliverables (Week 7): (10 Points)

1. Memory Banking and Cache Design
2. Have the following instruction classes working: Load/Store, Transfer, Arithmetic and Logical
3. Memory Banking
4. Have your cache design at least coded out if not working

Demonstrate that individual instructions work. Demonstrate that memory banking is working.

Your user interface, e.g., operator's console should be used to test instructions, etc.
Your simulator, packaged as a JAR file.

Simple documentation describing how to use your simulator, what the console layout is and how to operate it.

Your team's design notes

Source code – well documented.

Part III Deliverables (Week 11): 15 points

Your simulator, packaged as a JAR file, running program 1
Load instructions from a file.
Simple documentation describing how to use your simulator, what the console layout is and how to operate it. Include source code for program I.
File containing program 1 as machine code.
Your team's design notes
Source code – well documented.
Demonstration that Program 1 works.

Part IV Deliverable (**End of Semester**): 20 points

Your simulator, packaged as a JAR file, running programs 1 and 2.
Updated documentation for your simulator. Include source code for programs 1, 2.
Files containing programs 1, 2 as machine code.
Additional design notes.
Source code – well documented.

If you choose IVa, you should write a simple program that demonstrates floating point add/subtract, vector add/subtract, and floating point conversion.