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 two 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.

The three components are:

Component	Description
I: Basic Machine	Design and implement the basic machine architecture (see segment I description); build user interface to simulator; demonstrate 1st program running on your simulator.
II: Memory and Cache Design	Design and implement the modules for enhanced memory and cache operations; extend the user interface. Demonstrate 2nd program running on your simulator.
III: Floating Point and Vector Operations	Design and implement the modules for floating point and vector operations and simple pipelining; extend the user interface; 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 1.6.0_23 or later. You will deliver a JAR file that I can run to test your simulator.

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 4th 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 of how to operate your simulator and explain features and operation of your operators console and field engineers console.

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 16 bits in length
- 1 Index Register – 16 bits in length
- 16-bit words/2 8-bit bytes
- Memory of 16,384 words/32,768 bytes
- Byte addressable

Instructions must be aligned on a word boundary (even byte number). Words must be fetched on a word boundary.

The two GPRs are numbered 0-3 and can be mnemonically referred to as R0 – R3. They may be used as accumulators. The index register is mnemonically referred to as X0.

The CPU has other registers:

Mnemonic	Size	Name
PC	16 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	16 bits	Instruction Register: holds the instruction to be executed
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 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

Assume characters are represented in ASCII.

Reserved Locations:

Memory Address 0: Reserved for the Trap instruction for Part II.
Memory Address 1: Reserved for a machine fault (see below).

Interrupts:

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

Machine Fault:

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 MSR.

The possible machine faults that are predefined are:

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

The following sections describe the instructions that you must simulate.

Miscellaneous Instructions:

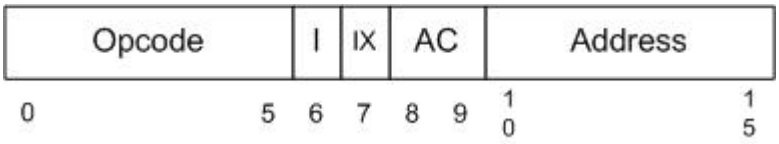
Miscellaneous instructions do not fit into another category (given the size of the machine).

$OpCode_8$	Instruction	Description
000	HLT	Stops the machine.
031	TRAP r	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. Register r 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 in r is greater than 15, a machine fault occurs.

Do not implement instruction 031 until Part II 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:



There are 64 basic opcodes, ranging from 000 to 063 octal. The I-bit stands for indirect addressing. The AC bits select one of four registers. The IX bit is 1 for indexing; otherwise, 0. The address is 6 bits and can address immediately 64 words. Therefore, to address all of memory, indexing will be required. We will use a base address indexing scheme.

Computing the Effective Address:

Effective Address (EA) = if I = 0: c(IX) + Address
If I = 1: c(c(IX) + Address)

If IX = 0, then no indexing! Hardware detects this!

To address 16K words, how many bits do you need?

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

$OpCode_8$	Instruction	Description
01	LDR r, x, address[,I]	Load Register From Memory r <– c(EA)
02	STR r, x, address[,I]	Store Register To Memory EA <– c(r)
03	LDA r, x, address[,I]	Load Register with Address r <– EA; r <– c(EA), 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:

000001 0 0 11 101001

Transfer Instructions:

The Transfer instructions change control of program execution. Conditional transfer instructions test the value of a register.

$OpCode_8$	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
015	RFS Immed	Return From Subroutine w/ return code in Immed portion (optional) stored in R0. R0 <– Immed; PC <– c(R3)
016	SOB r, x, address[,I]	Subtract One And Branch: r <– c(r) – 1 If c(r) > 0, PC <- EA; PC <– c(EA), if I bit set; Else PC <- PC + 1

For the RFS instruction, the Immediate portion will be the Address field.

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

Note: There is no register to register transfer! How strange! So, how do I transfer a value from one register to another?

Arithmetic and Logical Instructions:

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

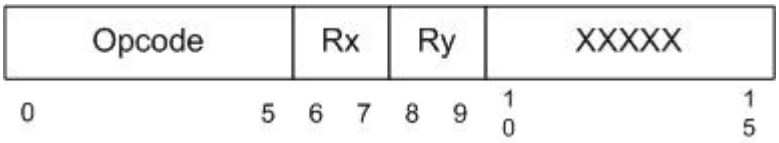
For immediate instructions, the Address portion is considered to be the Immediate value. The condition codes are set for the arithmetic operations.

$OpCode_8$	Instruction	Description
004	AMR r,x, address[,I]	Add Memory To Register r<– c(r) + c(EA)
005	SMR r,x,address[,I]	Subtract Memory From Register r1<– c(r1) – c(EA)
006	IAR r, immed	Immediate Add to Register 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	ISR r, immed	Immediate Subtract from Register 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.

AMR 2,1,63 where c(IX) = 500.

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.

OpCode$_8$	Instruction	Description
020	MUL 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, ry contains the low order bits of the result
021	DIV 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
022	TST 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	OR 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.

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

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

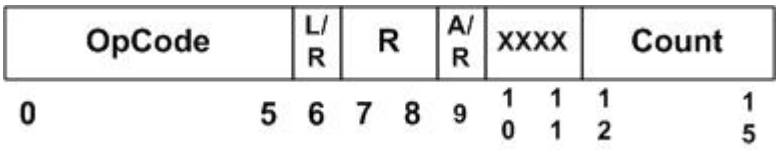
Shift/Rotate Operations

Shift and Rotate instructions manipulate a 16-bit datum in a register.

Arithmetic Shift (A/R = 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..

Rotate (**A/R=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) XXXX is ignored
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) or Arithmetically (A/L=0) XXXX is ignored

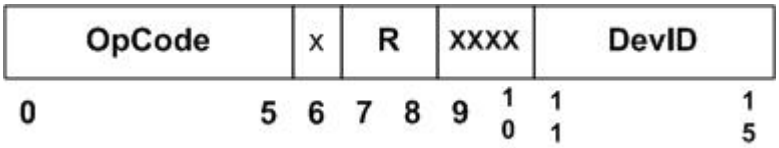
There's a lot going on here with these instructions. These are exemplars of some early machines which packed a lot of functionality into a few instructions.

So, suppose r0 = 0 000 000 000 000 110
Then, SRC 0,3,1,1 would yield r0 = 0 000 000 000 110 000

So, suppose r1 = 1 000 000 000 000 110
Then, SRC 1,2,1,0 would yield r1 = 1 000 000 000 011 000

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
062	OUT r, devid	Output Character to Device from Register
063	CHK r, devid	Check Device Status to Register c(r) <- device status

For Part I 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?*

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

Floating Point Instructions/Vector Operations:

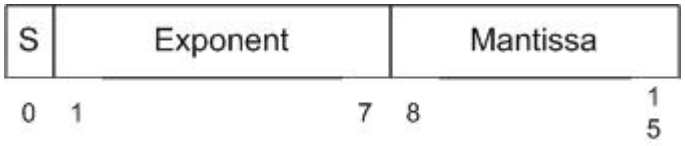
Do not implement floating point numbers until Part II!

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.

Note that we do not have special floating point registers in this machine. We use the GPRs for floating point operations as well.

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:



OpCode	Instruction	Description
033	FADD r, x, address[,I]	Floating Add Memory To Register c(r) <– c(r) + c(EA) c(r) <– c(r) + c(c(EA)), if I bit set R must be 0,2 OVERFLOW may be set
034	FSUB r, x, address[,I]	Floating Subtract Memory From Register c(r) <– c(r) - c(EA) c(r) <– c(r) - c(c(EA)), if I bit set R must be 0,2 UNDERFLOW may be set
035	VADD r, x, address[,I]	Vector Add c(r) = length of 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 V1 be vector at address; Let V2 be vector at address+1 Then, V1[i] = V1[i]+ V2[i], i = 1, c(r)
036	VSUB r, x, address[,I]	Vector Subtract c(r) = length of 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 V1 be vector at address; Let V2 be vector at address+1 Then, V1[i] = V1[i] - V2[i], i = 1, c(r)
037	CNVRT F, x, address[,I]	Convert to Fixed/FloatingPoint: If F = 0, convert c(EA) to a fixed point number and store in R0. If F = 1, convert c(EA) to a floating point number and store in R0 The R field is used to store the value of F

Note: Opcode 037 is a strange beast! It latches the result to R0 – 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:

011110 0 1 00 011001

At memory location c(IX) + 31: address of first vector
At memory location c(IX) + 32: address of second vector
Each of these vectors would c(r) 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 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.

3. Phase I Specification

In Phase I of 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.

For part I, 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

For Phase I of 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 2 banks of 8096 words each. So, you will need to decompose the contents of the MAR into a bank identifier and a specific word within the bank.

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. You can think of this latter component as a field engineer console as opposed to a user console. 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 operators console and field engineers console. So, plan accordingly and allow some growth space as you make your initial design.

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.

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

Part I 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 Deliverables:

Your simulator, packaged as a JAR file, running the two programs.
Simple documentation describing how to use your simulator, what the console layout is and how to operate it.
Source code – well documented.

Parts II will be handed out prior to the beginning of lecture 8.