

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/instructor at about 3 week intervals (see Syllabus).

The four components are:

Component	Description
0: Assembler	Write an assembler that converts each instruction into its 16 bit representation. And test the assembler.
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 Implement all instructions except for • CHK and • Floating Point/Vector operations • Trap 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 Demonstrate 2 nd program running 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 1.8 or later.

You will deliver:

- a JAR file that the grader can run to test your simulator.
- two programs in machine code which you have written based on specifications I will give you.
- the source code of your simulator
- YOUR TEST FILES for the simulator
- a written report describing how to operate your simulator
- a basic design document consisting of the application object models and interfaces, with brief description of the functions of each module.

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, Visual Studio. This will facilitate documentation of the object model as well as increase development productivity.

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 must be turned in with each segment. Refer to the project rubric provided by your professor.**

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. **Be sure that the JAR file will work on both Windows and Apple platforms so that it is readily graded.**

You may also implement a field engineer's console to help you debug the software. This would be a panel in the GUI which would allow input and output.

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!! **As long as they do not simply repeat the code.**

More importantly, part of the evaluation of your simulator is how well your code is commented so that we can understand what you are doing.

C6461 Computer Architecture

The C6461 Computer Architecture is described in an associated document, “C6461 Computer Design and Development Project – Instruction Set Architecture”. It should be closely followed.

Assembler

The assembler that you write will convert the instructions to octal and output into a text file with the address of the instruction and the numeric value of the instruction in octal.

As an example, consider the instruction: LDR 3,0,15 (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 15 is the EA.

This instruction would be encoded as:

Opcode	R	IX	I	Address
000001	11	00	0	01111

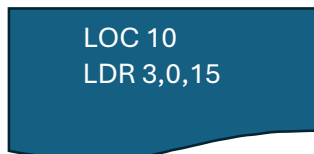
The octal (base 8) representation would be

0000011100001111
0 0 3 4 1 7

If this instruction is intended to be loaded at location 10 decimal, the output file of the assembler would be:

000012 003417 LDR 3,0,15

The Source File Might have been as below. Note I used and LOC 10 command just to set the address. Note* You have to read and translate all inputs. Note a full example is given below the figures.



LOC 10
LDR 3,0,15

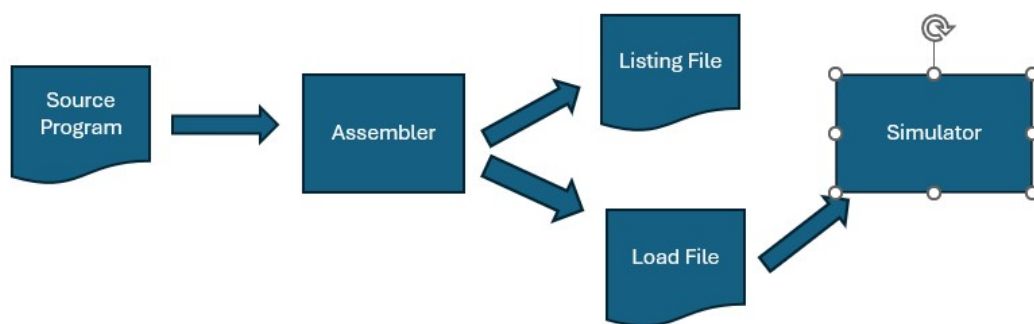
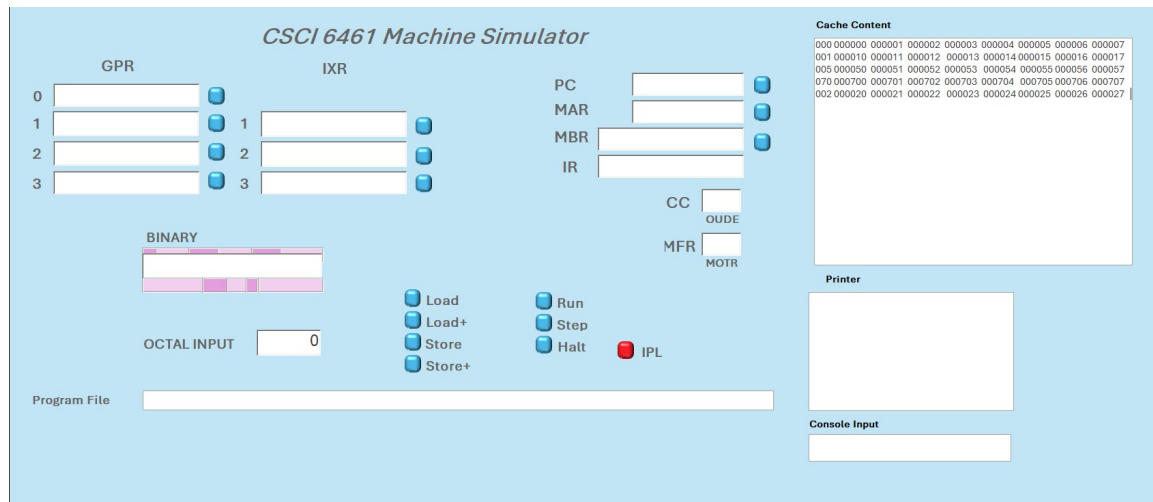


Figure – Overall assemble data flow

The Simulator

The following sections describe the instructions that you must simulate.



A Sample Front Panel for Programs 1 and 2 (Note that the cache contents are not required until program 2)

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 possible instruction numbers 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.

We will simulate a ROM load. This is a simplification using files as original machines may have used a card reader (not implemented) You can also use a fixed file name as long as you include instructions on how to read the file.. In other words the program could load from a fixed file in the same directory. The program to be loaded will be in a file on the machine. When the init button is pressed, your program should bring up a dialog that asks for the location of the file.

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.

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:

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 -32768 ... 32767. Therefore, as you read a character in, you need to check it is a digit, convert it to a number, and assemble the integer. You should be able to handle negative numbers.

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.

Part 0 Deliverable (See syllabus and Project Grading Rubric) (5 points)

Your assembler, packaged as a JAR file
Simple documentation on how to unload and run your assembler
Basic Design Notes
A test case for the assembler
A copy of the listing file for the test case
GitHub submit logs

Part I Deliverable (See syllabus and Project Grading Rubric) (10 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.
Test Cases
Your team's design notes
Source code – well documented.
GitHub submit logs

You should be able to enter data into any of R0 – R3; enter data into memory via switches; enter the various Load and Store instructions into memory; enter address into PC and press Single Step switch to execute the instruction at that address.

For this deliverable, when the IPL button is pushed the machine should pre load a program that shows that the machine works and stop at the beginning, ready for the user to hit run or single step. This program will be short but show all addressing modes for

the load and store instructions, allowing single step through. The program. Memory contents at the address given in the MAR should be on the user console.

Part II Deliverables (See syllabus and Project Grading Rubric) (15 points)

1. Cache Design and Implementation
2. Have all instructions working except Part IV
3. Have your cache design at least coded out if not working

Demonstrate that individual instructions work.

Your user interface, e.g., operator's console should be used to test instructions, etc.

Include source code for Program 1.

Your simulator, packaged as a JAR file, running program 1.

File containing program 1 as machine code.

Demonstration that Program 1 works.

Test Case

GitHub submit logs

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 (See syllabus and Project Grading Rubric) (15 points)

Your simulator, packaged as a JAR file, running program 2

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 2.

File containing program 2 as machine code.

Your team's design notes

Source code – well documented.

Test Data

GitHub submit logs

Demonstration that Program 2 works.

Part IV Deliverable TBD points - TBD

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.

Test Data

GitHub submit logs

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.