LC2200 Simulator

Sean Hoyt, Nina Chepovska, Tyler Braden

TCSS 372 Computer Architecture

Fall 2015

# Contents

[Contents 2](#_Toc437966070)

[Abstract 3](#_Toc437966071)

[Tools 4](#_Toc437966072)

[LC2200 Program Flow 5](#_Toc437966073)

[Design 6](#_Toc437966074)

[Unique Features 8](#_Toc437966075)

[Improvements 10](#_Toc437966076)

[Final Thoughts 11](#_Toc437966077)

[Additional Documentation 12](#_Toc437966078)

[Test Program Output 13](#_Toc437966079)

[Test Program Output (continued) 14](#_Toc437966080)

[Test Program Output (continued) 15](#_Toc437966081)

[Test Program Final Output 16](#_Toc437966082)

# Abstract

A LC2200 simulator program which models the functionality of an LC2200 computer. The program reads an input file containing LC2200 assembly code, tokenizes and parses the code instructions, sends each instruction through the Finite State Machine where it is fetched, decoded, and executed. After the all instructions are processed, the final result is output. The program is written in the C programming language and Git was used for version control.

# Tools

* C programming language
* Linux terminal
* Glade3
* GTK+
* Git
* GitHub

# 

# LC2200 Program Flow

GUI

Assembler functions

Input file

MEM{MAR[PC]]->IR Updates GUI

Processor

MAR

PC+1->PC Sends instruction to table

Transition Table

(FSM )

# Design

Firstly let's talk about design of a single instruction in our ISA . Each instruction is stored in a special "instruction format" union, where the various R,I,J are instruction types of the LC2200 ISA. The registers and MAR also have fields that are structures of this type, allowing for easy transfer of instructions or data through the data path. Each instruction is merely an unsigned 32-bit integer that has bit-fields that contain information about the instruction i.e. op-code is the 4 left-most bits, register1 is the next 4 bits and so on.

This 32bit unsigned integer is coupled with a tag for the type of instruction it is, as well as a string name representing the instruction. Due to the ease of creating and adding instructions, our current ISA could easily be expanded upon to facilitate a variety of different utility instructions if the need should arise. Detailed diagrams showing the various relationships between instructions and other structures in our program can be viewed in the included file "class\_diagrams.pdf". Alternatively navigate to folder "html " and file "annotated.html" to view html web-page version of diagrams.

The design of our program largely follows the flow diagram provided on the previous page. An input file, ("input.txt") is provided in the included zip, and is also the input used for the sample output featured later in the paper. This input file is then passed through the first stage of the assembly process where each line is tokenized and stripped of invalid characters. It is then sent the "build instruction" function that will create an instruction format structure from an input string. At this point, the new instruction is added into the MAR, and the program continues to build instructions until it no longer has any input strings.

Now that all the instructions we want to execute are loaded into the MAR, next the program loads instruction in MAR at PC, into the IR, which then passes instruction to our transition table. The transition table, is a 2D-Array of structures that contain a state, and a function pointer that executes a micro instruction. In our table nearly all are implemented exactly in the same manner is those in textbook, except some minor variations, which we will get to shortly. The table functions as a look-up and since many instructions function similarly it simplifies the decoding stage of executing an instruction significantly. After looking up the function in the table program executes corresponding micro instruction, and continues to execute until all states have been reached.

At this point in the fetch-decode-execute cycle, the GUI is updated, then the next instruction is loaded, and the process continues until PC no longer points to an instruction in memory. The GUI for our program ended up being a text based one due to certain time constraints and various unfortunate circumstances.

Utilizing the ncurses library we were able to keep a constant terminal window that we could step through to display our processor during execution. After stepping through all instructions, register window closes and then displays the MAR. Though for demonstration purposes we opted for the text based GUI, we also do have a gtk2 based GUI included but ran out of time to get all the various fields updating on the execution of our program. This is included in the various files in the zip under name "test.c" and executable "test".

# Unique Features

One of the key features of the design of our instructions is that all instructions take the same basic form: ARG1 ARG2 ARG3, this differs slightly from the ISA in the book where the memory address of various I-type instructions requires 2 registers and memory address attached to 2nd register, rather than having 3 arguments. Input strings are written in all caps, and appended with a semi colon, for example:

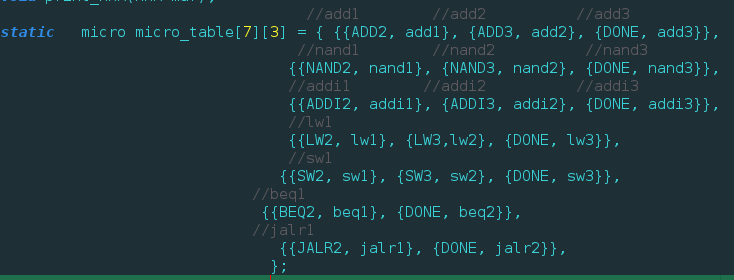
LW $A1, $ZERO, 42;

Since these strings are from file and of this form, this has the consequence of simplifying the structure of our transition table, as all function pointers need to be of the same form. As well as giving some sense of unity between our ISA and C by having a ';' at the end of each line. It has the added benefit of making each line exceptionally easier to parse, since every line must contain the semi-colon delimiter.

Since our emulator is hosted on an operating system that takes advantage of pipelining and other built-in instructions, we were able to reduce a few of our instructions down a couple micro-states. This fact allowed us to not have to worry about different hardware elements that would be needed to actually shorten the steps the way we have.

For example, BEQ was able to be turned two different states rather than the extended 6 from book. This is due to being able to test conditionals like if((A-B) > 0) without the need for more instructions, as the C programming language provides them. Instructions LW and SW were also able to be shortened by performing an extra step in the third micro state, eliminating the need of a fourth.

The following image indicates the micro-states present for each instruction, and where they are stored in the transition table.



As mentioned above, we were able to implement both the BEQ and JALR instructions, though we do not currently have the ability to parse labels. So you can, for example say:

BEQ $A1, $A2, -3;

to have the program counter set back three places, or really any arbitrary number positive or negative. This same semantic scheme goes for JALR as well, allowing the user to jump around anywhere in memory.

Another key feature our program is the ability to step through each instruction being executed, allowing inspection of the registers, and the various values they may contain. In this way, we are able to see how overall state of the LC2200 simulator at any given point, and ensure instructions are executing properly.

# Improvements

Currently our program has a few improvements that must be made in order for it to operate on any scale other than a small test. The first being, the size of our MAR; currently it is set to hold a max 1000 instructions, and if we wanted to run a rather large program, or a program that wanted to store more than a thousand variables, then we would need to make the MAR adjustable to whatever a user specifies. Continuing on this notion, we also need to be able to have our assembler parse a starting address, since currently we use fixed size, we disabled this option for user programs. This could be done in much the same way a label would be parsed, so we would be able to get two major features accomplished if the assembler could identify non-executable lines of code.

Although not a major deal and more a stylistic choice, it may be necessary to allow the user to enter instructions in lowercase as well as the current all caps input schematic. It would also be nice to allow and be able parse some of the comments and be able to display them next to the MAR.

For our text based GUI it would have been nice to have two separate windows displaying and running simultaneously. This would have allowed for a more complete picture of what was happening to the entire LC2200 during each step of the program.

Though we did complete a gtk2 GUI we had some last minute issues that prevented it from being correctly hooked up to the back-end. Maybe given a few extra days we could have had everything working and displaying perfectly in the GUI, but that's generally how those type of things go.

# Final Thoughts

In conclusion, this project provided an interesting and in depth look into the inner workings of a small computer. It required extensive research into parts of the C programming language many of us never encountered, as this was our first medium size extensive project done in C.

These new areas such as: OOP in C, unions, function pointers, and bit-fields. Object Oriented programming in C was especially exciting considering the language by design hadn't considered objects. The use of unions allowed us to implement structures that could handle various types of data, in this case instructions. While function pointers allowed us to treat all of our micro instructions as the same instruction.

Not only that but the use of glade3 to generate layouts was an exceptionally useful as now, we are much more comfortable attempting to build a layout whilst programming in C. Like Java's AWT or whatever it they use now, it merely requires time to learn the library. After that you can generate very appealing GUI's in a relatively short period of time. The decoupling of the GUI and back-end is very appealing, and allows the independent development of each. Since glade3 generates XML documents it allowed us to delve into what much of the world is using to generate layout in a variety of languages and platforms, including Android.

All in all, this project provided an avenue to better our skills and become significantly more proficient in C than we were going into the project. Not only that, but now we actually have an understanding of how each MACRO instruction we use in our real-life ISA like Intel, is actually breaking each instruction into various micro states and executing them. Now when programming it will be much harder to not consider the amount of states a particular MACRO instruction like '\*' in C might be taking. In general this project provided genuine insight into how one might build an emulator for any target platform.

# Additional Documentation

We also have class diagrams in:

class\_diagram.pdf

as well as an explorable html offline webpage include in folder "html"

"annotated.html"

Will display visualization of relationships between various structures in program.

The following in red is for compiling programs from source otherwise just

run: ./micro\_processor

Specs:

input file: input.txt

compile command : gcc -Wall -g -lncurses processor.c micro\_table.c -o lc2200

run command : ./lc2200

must have ncurses library to compile, otherwise just run included executable "micro\_processor"

run command : ./micro\_processor

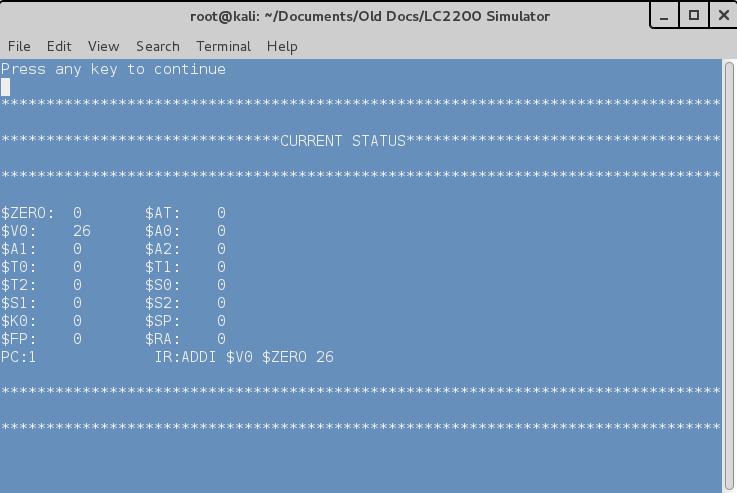
We also have a gtk2 gui that you can test, but doesn't run any commands,

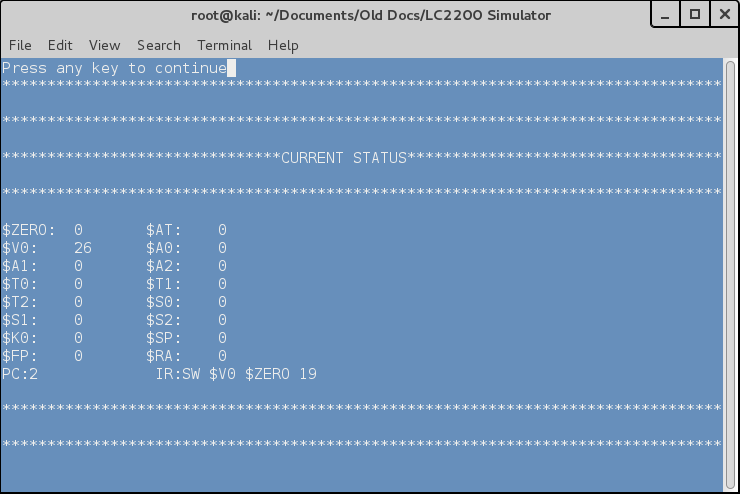
we've included the executable as "test"

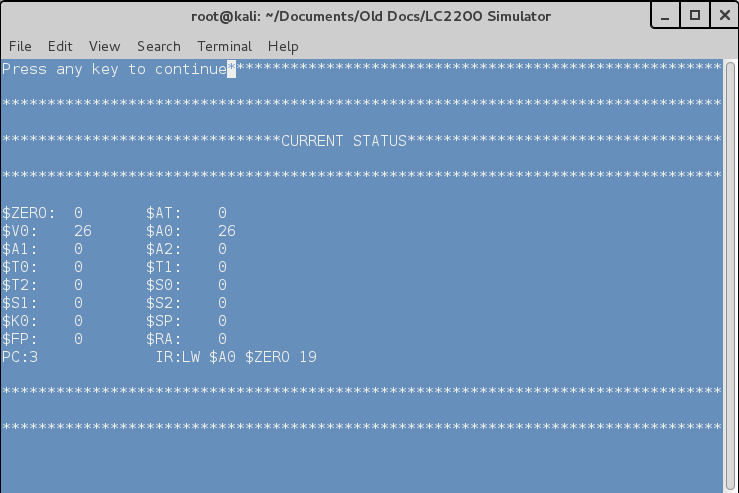
run command: . /test

We've also included PNGs depicting each step of the program executing file input.txt in the remaining pages of the document. Just in case for some reason you cannot run or compile our program.

# Test Program Output

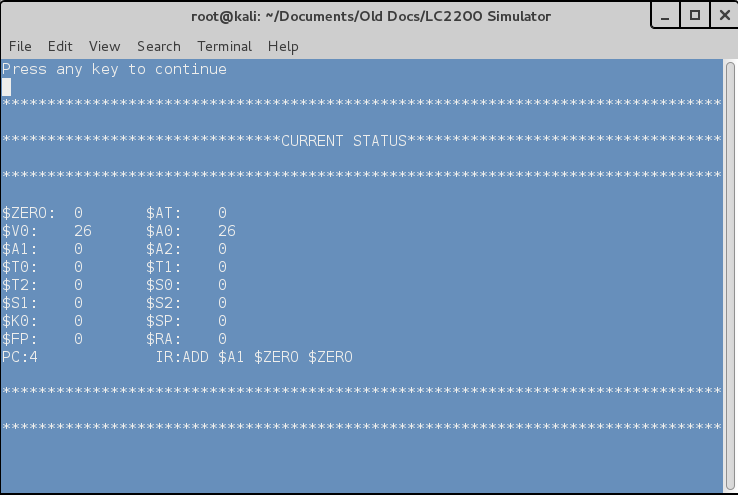
 PC=1: $V0+26 -> $V0



PC=2: $V0->MEM[MAR[19]]

PC=3:

$A0<-MEM[MAR[19]]



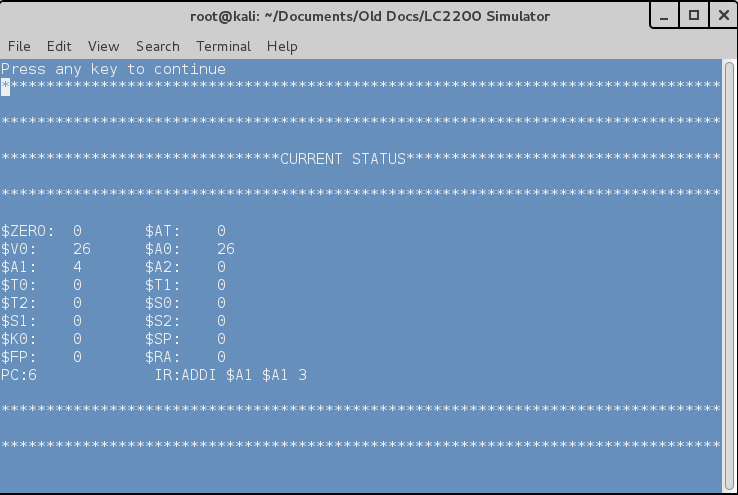
PC=4:

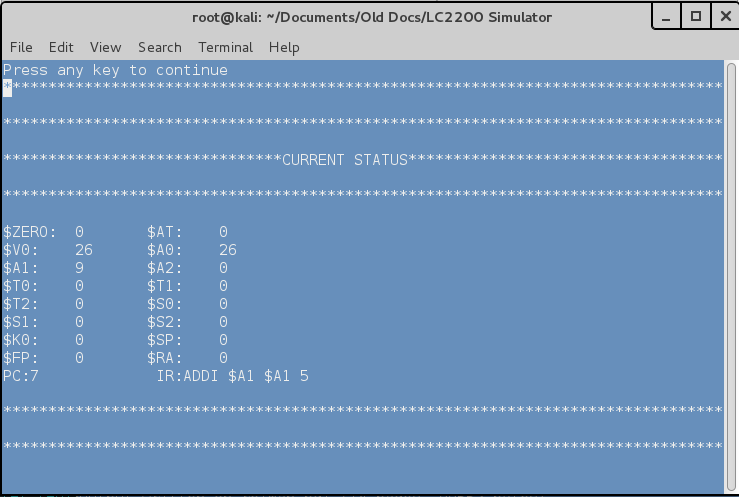
$ZERO+$ZERO->$A1

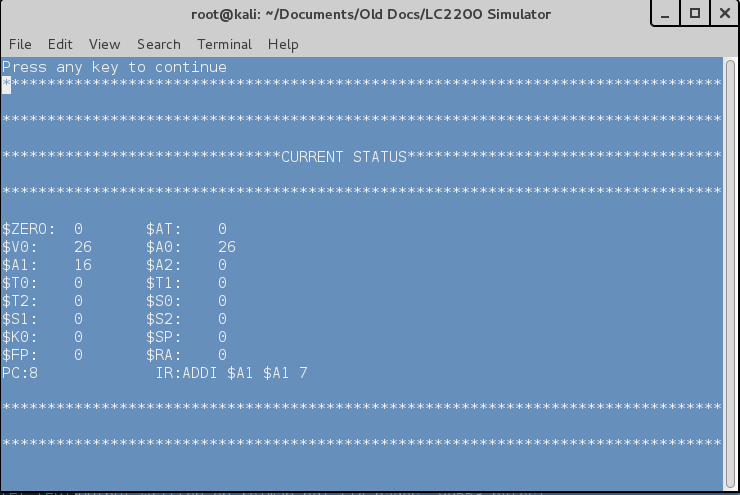
# Test Program Output (continued)

PC=5:

$A1 + 1 -> $A1



PC=6: $A1 + 3 -> $A1

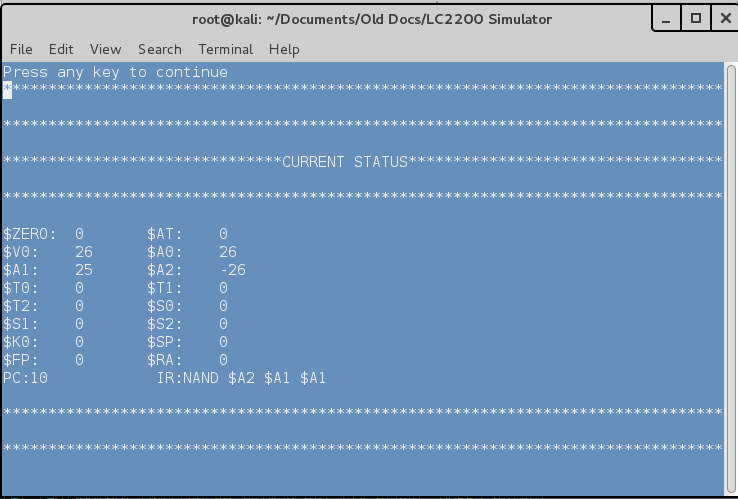
 PC=7: $A1 + 5 -> $A1

PC=8:

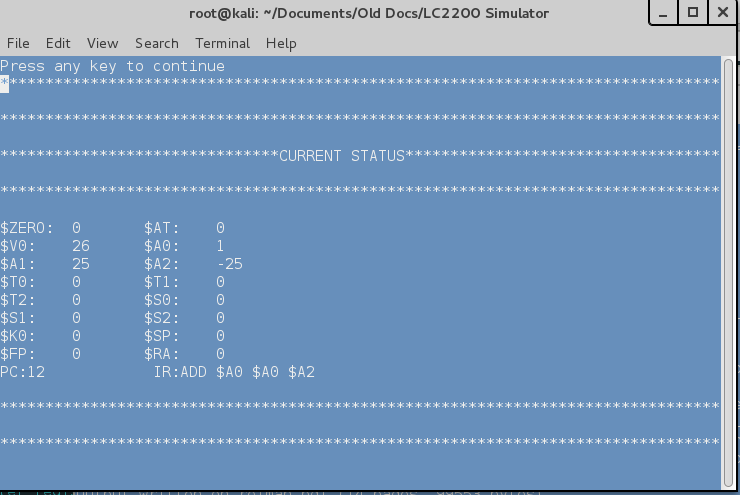
$A1 + 7 -> $A1

# Test Program Output (continued)

PC=9: $A1 + 9 -> $A1



PC=10: $A1 NAND $A1 -> $A2

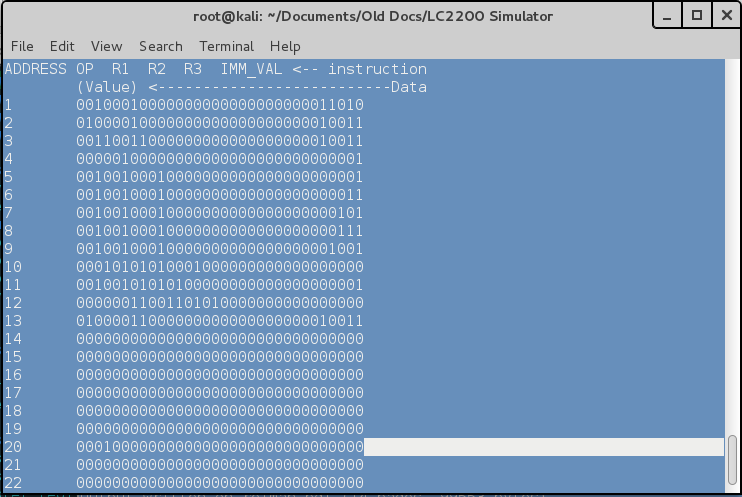
 PC=11: $A2 + 1 -> $A2

PC=12: $A0 + $A2 -> $A0

# Test Program Final Output

PC=13:

$A0 + $ZERO -> MEM[MAR[19]]

MAR

(note value = 1 stored at mem[MAR[19]] aka ("20") on print-out