Computer Simulation using Object-oriented design in C

Henry N. Lawerence, Nicholas A. Hays, Benjamin J. Pasero

University of Washington – Tacoma

February 21, 2016

Table of Contents

[Abstract 3](#_Toc445067823)

[Overview 4](#_Toc445067824)

[ISA 4](#_Toc445067825)

[Instruction Set 5](#_Toc445067826)

[R-Type (add, nand) 5](#_Toc445067827)

[I-Type (addi, lw, sw, beq) 5](#_Toc445067828)

[J-Type (jalr) 5](#_Toc445067829)

[O-Type (halt) 5](#_Toc445067830)

[DataPath Design 6](#_Toc445067831)

[CPU 6](#_Toc445067832)

[ALU 6](#_Toc445067833)

[Memory 6](#_Toc445067834)

[Register File 7](#_Toc445067835)

[Assembler 7](#_Toc445067836)

[Parser 7](#_Toc445067837)

[View 7](#_Toc445067838)

[Signals 7](#_Toc445067839)

[Simulator 8](#_Toc445067840)

[Model 8](#_Toc445067841)

[View 8](#_Toc445067842)

[Signals 8](#_Toc445067843)

[Appendix 10](#_Toc445067844)

# Abstract

This project is an abstraction of a computer’s data path design using object-oriented programming in C with supporting GIMP Toolkit (GTK) libraries to support front-end display. The model is built in accordance with the 32-bit, fixed-length LC-2200 Instruction Set Architecture (ISA) which supports four types of addressing modes (R-type, I-type, J-type, and O-type) and a total of 8 different instructions: ADD, NAND, ADDI, LW, SW, BEQ, JALR, and NOP. The core program files consist of four basic structures that represent the combinational and sequential-logic hardware elements of the data path: Central Processing Unit (CPU), Memory, Arithmetic Logic Unit (ALU), and Register set. The CPU—in conjunction with its supporting structures— will mimic the Finite State Machine (FSM) to orchestrate the fetch, decode and execution cycles of a modern processor.

*Keywords:* gtk, c, object-oriented, PC, ALU, register, memory, assembler

# Overview

There are several layers involved when simulating the execution of a computer program. Before the process begins, however, the instruction set must be defined in order to establish the interface between the software and the hardware. The LC-2200 will serve as the fundamental instruction set design for this simulation. To initiate the process, a program must be created using the instruction format specified in the ISA (refer to Appendix A to see a comprehensive list set of the LC-2200 ISA). Once the program is complete, it is then assembled into its binary representation. This binary file will contain the bytes representing the instructions needed by the CPU to execute the program. The simulator will then load the binary file into memory so the CPU can begin the fetch, decode, and execution cycle. After each instruction execution cycle, the simulator will update the changes made to the register values along with the Program Counter (PC) and Instruction Register (IR). This process will continue until all instructions in memory have been executed.

# ISA

(Heading 1 centered and bolded)

-describe our ISA. i.e 32bit, fixed length, refer to our assignment 1 papers for content.

-what instructions do we have? Addressing modes?

How have we incorporated OOD into our ISA?

Example in text figure..

Our model is similar to the LC-2200 datapath design. (see Figure 1 below)

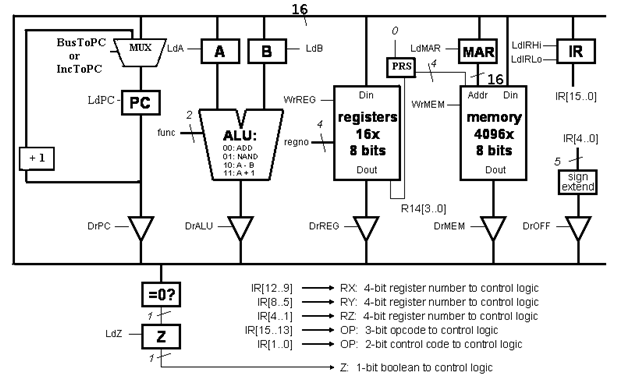


Figure 1: LC-2200 Data Path Design.

## Instruction Set

What is an instruction set? What purpose does it serve?

Describe what instructions we support

### R-Type (add, nand)

Describe mode

Give graphic outline of format..i.e 31-28 opcode, 27-24 Reg X…

### I-Type (addi, lw, sw, beq)

Describe mode

Give graphic outline of format..i.e 31-28 opcode, 27-24 Reg X…

### J-Type (jalr)

Describe mode

Give graphic outline of format..i.e 31-28 opcode, 27-24 Reg X…

### O-Type (halt)

Describe mode

Give graphic outline of format..i.e 31-28 opcode…

-

# DataPath Design

-describe the data path design

-how does the data path design influence out project?

How have we incorporated OOP principles into the design?

Grab information from c docs or gtk docs and pull directly and cite in text apa

## CPU

-What is the CPU? What are the components in the CPU structure, describe their purpose

-how does our cpu work? fetch, decode, execute (PC IR…)

-How have we implemented the CPU?

-describe main functions of fetch, decode and execute.

Screen shots of source code

FSM

Screen shot of ROM put in text

Screen shot of source code, put in appendix apa format, reference appropriately in text

Describe all main functions purpose.

## ALU

-what is the ALU?

-what type of operations does the ALU perform in our application?

- Screen shot of source code, put in appendix apa format, reference appropriately in text

## Memory

-what functions does it have? How does it do them?

Refer to source code in appendix using correct apa in text reference

Screen shot of source code, put in appendix apa format, reference appropriately in text

## Register File

how many?

Describe function.

Screen shot from lc2200 online and put in text with caption

Screen shot of source code, put in appendix apa format, reference appropriately in text

# Assembler

Discuss purpose

What its used for.

Start with .orig followed by hex address

Ends with .end

Grab information from c docs or gtk docs and pull directly and cite in text apa

## Parser

How it works

Objectives

Describe bit shifts, bin to string, anything you did during process

Screen shot of source code, put in appendix apa format, reference appropriately in text

## View

describe the object hierarchy (i.e high level container (type), layout container(s), menu bar in first sub container, …)

You might also want to be sure you describe what the image is.

Refer to image in text with caption at the bottom of image as shown above in ISA

Screen shot of source code for view, put in appendix apa format, reference appropriately in text

## Signals

If any? Maybe to parser?

Screen shot of source code, put in appendix apa format, reference appropriately in text

# Simulator

Describe purpose of simulator

Describe its function

How it relates to this project

## Model

How the tree model works

What information it stores, how it retrieves its information, how it updates the view

Screen shot of source code, put in appendix apa format, reference appropriately in text

## View

Describe view layout

Describe view object hierarchy

Include screen shot in text

## Signals

What do the signals do?

Gtk signals

What signals are there (how many)?

How do they interact with control unit

How do they allow user to update register values

Open file chooser

# Appendix A

LC-2200 Instruction Set Architecture

-----------------------------------------------------------------

Instruction Formats

-----------------------------------------------------------------

There are five instruction formats. Bit 0 is the least-significant:

R-type instructions (add,nand):

bits 31-28: opcode

bits 27-24: reg A

bits 23-20: reg B

bits 19-4: unused (should be all 0s)

bits 3-0: reg DST

I-type instructions (addi, lw, sw, beq):

bits 31-28: opcode

bits 27-24: reg A

bits 23-20: reg B

bits 19-0: OFFSET (a 20-bit, 2s complement number with a range

of -524288 to +524287

J-type instructions (jalr):

bits 31-28: opcode

bits 27-24: reg A

bits 23-20: reg B

bits 19-0: unused (should be all 0s)

O-type instructions (halt, ei, di, reti):

bits 31-28: opcode

bits 27-0: unused (should be all 0s)

-----------------------------------------------------------------

Register Convention

-----------------------------------------------------------------

Registers indicated with a '$' sign. The register names in assembly

are according to their use in the assembly convention:

regno name use callee-save

----- ---- ------------------------- -----------

0 $zero always zero (by hardware) n.a.

1 $at reserved for assembler n.a.

2 $v0 return value no

3 $a0 argument or temporary no

4 $a1 argument or temporary no

5 $a2 argument or temporary no

6 $a3 argument or temporary no

7 $a4 argument or temporary no

8 $s0 saved register YES

9 $s1 saved register YES

10 $s2 saved register YES

11 $s3 saved register YES

12 $k0 reserved for OS/traps n.a.

13 $sp stack pointer YES

14 $fp frame pointer YES

15 $ra return address YES

-----------------------------------------------------------------

Instruction Semantics

-----------------------------------------------------------------

Assembly language Opcode in binary Action

name for instruction (bits 31/30/29/28)

-----------------------------------------------------------------

add (R-type format) 0000 add contents of A with

ex: add $v0, $a0, $a1 contents of B, store results in

DST. Ex: $v0 := $a0 + $a1

nand (R-type format) 0001 nand contents of A with

ex: nand $v0, $a0, $a1 contents of B, store results in

DST. Ex: $v0 := ~($a0 + $a1)

addi (I-type format) 0010 Add OFFSET to the contents of A

ex: addi $v0, $a0, 25 and store the result in B.

Ex: $v0 := $a0 + 25

lw (I-type format) 0011 load B from memory. The memory

ex: lw $v0, 0x42($fp) address is formed by adding

OFFSET to the contents of A.

Ex: $v0 := memory[$fp + 0x42]

sw (I-type format) 0100 store B into memory. The memory

ex: sw $a0, 0x42($fp) address is formed by adding

OFFSET to the contents of A.

Ex: memory[$fp + 0x42] := $a0

beq (I-type format) 0101 compare the contents of A and B.

ex: beq $a0, $a1, done If they are the same, then

branch to the address

PC+1+OFFSET, where PC is the

address of the beq instruction.

Ex: if ($a0 == $a1)

PC := (PC+1)+OFFSET

\*\*\* NOTE \*\*\*

For programmer convenience (and

implementor confusion), the

assembler \*computes\* the OFFSET

value from the number or symbol

given in the instruction and the

assemblers idea of the PC. In the

example, the assembler stores

done-(PC+1) in OFFSET so that

the machine will branch to label

"done" at run time.

jalr (J-type format) 0110 First store PC+1 into B,

ex: jalr $at, $ra where PC is the address of the

jalr instruction. Then branch to

the address now contained in A.

Note that if A is the same as B,

the processor will first store

PC+1 into that register, then end

up branching to PC+1.

Ex: $ra := PC+1; PC := $a0

halt (O-type format) 0111 halt the machine: i.e. do nothing

ex: halt and let the simulator notice that

the machine halted.

ei (O-type format) 1010 enable interrupts

ex: ei

di (O-type format) 1011 disable interrupts

ex: di

reti (O-type format) 1100 return from interrupt by loading address

ex: reti stored in $k0 into the PC and then

enabling interrupts

----Assembler Directives----

noop (pseudo-op) n.a. No operation: does nothing (actually

ex: noop Emits "add $zero, $zero, $zero")

.word (pseudo-op) n.a. fill word with a value.

ex: .word 32 Ex: fill the current location

with the 32-bit represenation of

the number "32".

Simulator Screen Shots:

GtkWindow

GtkBox -> Container1

MenuBar

FileMenu

FileLoadMenuItem

GtkBox->Container2

TreeView1

TreeModel

ScrollableContainer

TreeView2

TreeModel

TreeListStore

TreeList