

IMPROVING STUDENT COMPREHENSION THROUGH INTERACTIVE
MICROARCHITECTURE SIMULATION AND VISUALIZATION

A Thesis
by
ANDREW BROWNFIELD

Submitted to the Graduate School
at Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

December 2013
Department of Computer Science

IMPROVING STUDENT COMPREHENSION THROUGH INTERACTIVE
MICROARCHITECTURE SIMULATION AND VISUALIZATION

A Thesis
by
ANDREW BROWNFIELD
December 2013

APPROVED BY:

Dr. Cindy Norris
Chairperson, Thesis Committee

Professor Frank Barry
Member, Thesis Committee

Dr. Alice McRae
Member, Thesis Committee

Dr. James T. Wilkes
Chairperson, Computer Science

Edelma D. Huntley
Dean, Research and Graduate Studies

Copyright by Andrew Brownfield 2013
All Rights Reserved

Abstract

IMPROVING STUDENT COMPREHENSION THROUGH INTERACTIVE MICROARCHITECTURE SIMULATION AND VISUALIZATION.

Andrew Brownfield
B.S., Appalachian State University
M.S., Appalachian State University

Chairperson: Dr. Cindy Norris

The curricula of most Computer Science departments include at least one course on computer organization and assembly language. The seminal concepts covered by such courses bridge the gap between hardware and software by introducing multiple layers of abstraction. Appalachian State University introduces this material in the course “Introduction to Computer Systems.” The current structure of the course teaches the concepts using the hypothetical LC-3 processor, as presented in Patt and Patel’s textbook “Introduction to Computing Systems: From Bits & Gates to C & Beyond (2nd edition).” Prior to the completion of the work presented in this thesis, tools existed for the assembly of LC-3 programs and simulation of the assembled code; however, no simulator existed to demonstrate the function of the microarchitectural level.

In this thesis, research on educational simulators is presented, with an emphasis on microarchitectural and graphical style simulators. Multiple simulators were

reviewed to determine which elements are pedagogically effective and which elements detract from the educational value of the simulator. Based on these findings, a graphical microarchitecture simulator named lc3uarch was implemented. The simulator targets the microarchitectural level of the LC-3 processor. To determine its effectiveness as an educational tool, student surveys were conducted. The responses indicated that the use of lc3uarch can help students comprehend the logic components of the LC-3 microarchitecture and provided ideas for making the tool more effective.

Acknowledgments

If it weren't for the support of many people, this thesis would not have come to fruition. Foremost, many thanks to my advisor, Dr. Norris. In spite of my lethargic pace, she did not lose hope that I would eventually finish. Many thanks also go to the thesis committee members Professor Barry and Dr. McRae.

As the graduate school liaison, Dr. Fenwick has been a great help in enrollment, scheduling issues, and ensuring that the deadline had not passed.

My deepest gratitude goes to my family. Their words of encouragement and eagerness to provide childcare have been a boon in finishing. My amazing wife Erin has been my biggest cheerleader, particularly during times of self doubt and in the final weeks of writing.

Dedication

This thesis is dedicated to my son, Dayton Robert Brownfield. On many occasions he attempted to help write the paper. In spite of his best efforts, the following is the only contribution that will make the final cut: “vnuhn7k7ufjk..g...v..g..gg/f.vbv/gb///t,,g,g,ggggggg.”

Table of Contents

Abstract	iv
Acknowledgments	vi
Dedication	vii
List of Figures	xi
List of Tables	xii
Foreword	xiii
1 Introduction	1
2 Uses of Simulation and Visualization	4
2.1 Advantages of Simulators	5
2.2 Use of Simulators for Education	8
2.2.1 Targeted Courses	8
2.2.1.1 Introduction to Computer Science	9
2.2.1.2 Introduction to Computer Organization	9
2.2.1.3 Advanced Computer Organization	12
2.2.2 Pedagogical Styles and the Use of Simulators	13
2.3 Simulation Visualization and Interfaces	14
2.4 Taxonomy and Categories of Educational Simulators	16
2.4.1 Categories of Simulators	16
2.4.1.1 Historical	17
2.4.1.2 Digital Logic	17
2.4.1.3 Simple Hypothetical Simulators	17
2.4.1.4 Intermediate Instruction Set	18
2.4.1.5 Advanced Microarchitecture Simulators	18
2.4.1.6 Multiprocessor	19
2.4.1.7 Memory Simulators	20
2.4.2 Taxonomy of Program Visualizations	20
2.4.2.1 Scope	20
2.4.2.2 Abstraction	21

2.4.2.3	Specification	21
2.4.2.4	Interface	22
2.4.2.5	Presentation	22
2.4.3	Taxonomy of Architecture Visualization	23
2.4.3.1	Scope of Operation	23
2.4.3.2	Content Modeling	23
2.4.3.3	Presentation Methods	24
2.4.3.4	Activity Style	24
3	Example Simulators	26
3.1	Introductory Simulators	26
3.1.1	Knob & Switch Computer	27
3.1.2	Little Man Computer (LMC)	29
3.1.3	EasyCPU	30
3.2	Digital Logic Simulators	32
3.2.1	Logisim	33
3.2.2	DLSim 3	35
3.2.3	JLS	36
3.3	Microarchitecture Simulators	37
3.3.1	MicroTiger	37
3.3.2	Pep8CPU	40
3.3.3	MarieSim	42
3.3.4	RTLsim	43
4	LC-3	45
4.1	Instruction Set Architecture	45
4.1.1	Memory	46
4.1.2	Registers	46
4.1.3	Interrupts and Exceptions	48
4.1.4	Instructions	48
4.2	Microarchitecture	51
4.2.1	Datapath	51
4.2.2	Control	55
4.3	Existing Projects	57
5	Design and Implementation	59
5.1	Designing the Simulator	59
5.1.1	Portability and Language Choice	60
5.2	Relating to Taxonomy and Categories	61
5.3	Design Goals	64
5.4	Graphical Implementation	65

5.4.1	Animated Datapath	65
5.4.2	Controls and Components	68
6	Evaluation	73
6.1	Survey Content	74
6.2	Result of Surveys	75
7	Conclusions and Future Work	78
7.1	Conclusions	78
7.2	Future Work	80
7.2.1	Graphical and Usability Changes	81
7.2.2	Pedagogical Enhancements	82
7.2.3	Extended Scope	83
	References	85
	Vita	88

List of Figures

3.1	First Stage of Knob & Switch Simulator	28
3.2	Final Stage of Knob & Switch Simulator	29
3.3	Little Man Computer	31
3.4	EasyCPU Basic-Mode	32
3.5	EasyCPU Advanced-Mode	33
3.6	Logisim Editor	34
3.7	DLSim 3	36
3.8	JLS Circuit Editor	37
3.9	JLS FSM Editor	38
3.10	MicroTiger Datapath Editor	39
3.11	MicroTiger Microcode Editor	40
3.12	Pep8CPU Datapath	41
3.13	Pep8CPU Microcode Editor	42
3.14	MarieSim Datapath	43
3.15	RTLsim Datapath	44
4.1	LC-3 Memory Layout	47
4.2	LC-3 Instructions	49
4.3	LC-3 Basic Datapath	53
4.4	LC-3 Datapath with Interrupts	54
4.5	LC-3 Finite State Machine	57
5.1	Full Window of lc3uarch	66
5.2	Animated Datapath	67
5.3	Animation Speed Menu	68
5.4	Opacity Menu	68
5.5	Controls	69
5.6	Registers	70
5.7	Changing Register Value	70
5.8	Memory	71
5.9	File Menu	71
5.10	Finite State Machine Information	72
5.11	Console Output	72
5.12	Toggle Animations	72
6.1	Histogram of Responses	76

List of Tables

4.1	Tabular View of Control Structure	58
6.1	Totals of Survey Responses	76

Foreword

This thesis has been formatted according to the style guide of the Department of Computer Science at Appalachian State University.

Chapter 1 - Introduction

In the field of computer science, students study a broad range of topics, from abstract theoretical concepts to the logic gates printed on silicon. Unlike most courses, which focus primarily on either end of the spectrum, the topic of computer organization bridges the gap between high level computation and the hardware that makes the computations happen. Appalachian State University introduces this material in the sophomore level course “Introduction to Computer Systems.” In this course, multiple layers of abstraction are presented, taking the student from the digital circuits to higher level programming languages. In its current form, the course uses Patt and Patel’s textbook “Introduction to Computing Systems: From Bits & Gates to C & Beyond (2nd edition).” The textbook presents a hypothetical processor called LC-3. As the title’s subtext suggests, the topics covered range from bit-wise operations to higher level languages. Since the described processor has never been built, execution of code written for the LC-3 must be done via simulation. Until the project presented by this thesis, no simulation existed for the microarchitectural level of the processor.

This thesis documents the design, implementation, and evaluation of a graphical simulator targeted at the microarchitectural level of the LC-3 processor. The simulator is dubbed lc3uarch, short for “LC-3 Microarchitecture”. The letter ‘u’ was chosen as a more easily typed version of the Greek letter μ . This symbol is used as short-hand for the term “micro.”

In Chapter 2, research is presented on the use of simulators, particularly in an educational setting. The literature covers why simulators should be used in education, the ways in which they can be used in a course, and the cognitive models created by experts in the field of educational simulation. The emphasis of the research is on visualization techniques and design of microarchitecture simulation.

Multiple simulators are reviewed in Chapter 3, detailing the features they offer and the basics of their structure and operation. These simulators target several topics covered in computer organization courses. Not all simulators reviewed share the same goals as lc3uarch, but the features and concepts are relevant and provide insight into designing a pedagogically sound tool.

A detailed view of the LC-3 processor is provided in Chapter 4. The definition of the Instruction Set Architecture is described, followed by the microarchitecture presented by Patt and Patel. The existing LC-3 related tools are briefly described as well.

A more in-depth look at the design and implementation of lc3uarch is provided in Chapter 5. In particular, the design is presented in the context of the taxonomies revealed by the literature. More general elements are presented as well.

In Chapter 6, evaluation of lc3uarch's effectiveness is assessed through student surveys. Finally, Chapter 7 draws conclusions from the surveys, showing that the simulation is helpful in aiding student comprehension. The feedback also provides ways to improve the tool for future students.

Chapter 2 - Uses of Simulation and Visualization

Simulation is broadly defined as an imitation of a process or system from the context of another process or system. The term has become commonplace, with computers serving as the platform on which other systems can be represented. Software simulators can be computationally taxing, originally requiring extensive resources to model anything of value. With advances in hardware capabilities and reduction of hardware prices, simulators have become increasingly common tools for solving problems and demonstrating otherwise inaccessible systems. The emphasis of this chapter is exploring the use of simulation and visualization within the computing domain, particularly computer architecture, with an emphasis on pedagogical needs.

In order to increase clarity, ambiguous terms are first defined. The term *simulation* in the context of computer science refers to the technique of representing the real world with a computer program. This generally applies to applications such as economic forecasting or weather modeling; however, the emphasis of this research is modeling computer architecture, digital logic, and computer code. The term *visualization* refers to a method of displaying the state and processes of the simulation.

Thus, it is common that the visualization is often layered upon simulation software. The term *interface* is defined as the methods used to interact with the simulator. In most cases, the visualization method and interface are tightly coupled.

The contents of this chapter cover the use and general concepts of simulation. In some areas, brief references are made that will be expanded upon in later parts of the thesis. First, the advantages of simulation over hardware systems are discussed. This is followed by describing the use of simulation in university-level education. Further, the addition of visualization with simulation is described. Finally, the taxonomy and categorization of simulators and visualization is explored.

2.1 Advantages of Simulators

In recent years, Computer Science and Engineering departments have been moving away from hardware systems and towards simulation for course related systems. Of the many benefits cited, two major reasons stand out in nearly all reviewed texts: cost and accessibility [24, 12, 19, 1, 8, 4].

At the onset, the initial cost of hardware does not necessarily appear prohibitive, but several factors make long term costs significantly higher. First, the faculty must consider what system or systems are needed. For a single course, there may be need for examples of several different architectures, immediately multiplying the cost of the initial purchase. For most labs, the use of these systems must be shared between many students. Depending on the size of the department, this could

imply a need for multiples of each system to allow students the necessary time to use the hardware. It is also necessary to consider the configuration and administration costs of the initial setup. This only addresses the initial purchase and configuration.

Over the life of the hardware, more costs are presented. As hardware ages, physical components break and must be replaced. Under optimal conditions with no repairs needed, power costs are continually incurred and routine maintenance is needed by trained administrator. As courses continue to evolve, new hardware needs must be addressed and the older systems must vie for lab space and administrative time. Thus, the costs continue beyond the initial purchase.

Another large advantage of using simulators instead of hardware is the accessibility of software. In a typical lab setting, students vie for time in front of a hardware system. At a minimum, this requires some administrative work to allot time to users. The users, in this context, include administrators maintaining the system, teachers creating coursework, and students doing the coursework. If a student cannot complete his or her work during allocated class time, it is the student's responsibility to vie for the remaining bit of time available on the system. These problems are compounded for students and faculty who commute long distances.

The use of simulation often resolves or largely mitigates the cost and accessibility problems of hardware. A simulator may run on multiple platforms, and each system may run multiple simulation instances. Further, a single physical system can generally run multiple simulated systems. This immediately reduces the total

hardware needs drastically. If the system running the simulation software become antiquated or administrative costs become too high, simulation software can often be ported with little to no effort. This reduces the physical hosting costs in terms of space needed to keep old architectures around. The hardware costs already minimized, it is also worth noting that many of the simulators used for educational purposes are free to download, incurring no software costs beyond the operating system and supporting software.

The issue of accessibility is addressed by allowing students to access the simulator from multiple locations. No longer does the student need to sit in front of a bare-metal system to execute his or her work. At a minimum, the student should be able to access the simulator from any computer within the department's network. In most cases, that access is extended to any system with a network connection and a terminal client. Many educational simulators are also available to students for direct download, allowing the student to use the simulator at his or her own convenience [10].

For the needs of education, there are accessibility benefits beyond the logistics of where the simulation runs. For example, a simulation is easily slowed down or paused. This allows the user to review the state of all the components at a given time. On hardware systems this is a non-trivial task, if possible at all.

Overall, the case for using simulation versus hardware for education heavily leans towards simulation. Some situations may arise where hardware is still needed, but these would be exceedingly rare exceptions for required Computer Science courses.

2.2 Use of Simulators for Education

Many courses within a Computer Science curriculum can benefit from the use of simulation software. With the breadth of software packages that can be considered simulators, it is necessary to focus on a small subset for this thesis. The targeted subset is simulation of computer hardware. From university to university, the exact scope, name, and topics for the relevant courses vary. In spite of this variability, the reviewed literature commonly refers to three types of classes where simulation has been tried and generally found beneficial: Introduction to Computer Science, Introduction to Computer Organization and Assembly Language, and Advanced Computer Organization. Additionally, several authors have researched pedagogical approaches for integrating simulators with traditional teaching methods.

2.2.1 Targeted Courses

Three levels of courses in computer science curricula often use simulators. At the freshman level, Introduction to Computer Science can benefit from simulation of digital circuits. At the sophomore level, Introduction to Computer Organization and Assembly Language covers varying levels of abstraction of a CPU and supporting hardware. At the junior through graduate levels, Advanced Computer Organization expands on those topics and introduces new ideas relying on those fundamentals.

2.2.1.1 Introduction to Computer Science

In most departments, introduction to computer science covers the mathematical roots of computing. Among the topics covered, the tenets of Boolean and digital logic are particularly foundational. While this logic can be expressed solely through mathematical formulas, digital logic simulators can aid in demonstrating how the concepts apply in practice. The concepts of digital logic have a basal overlap with electrical engineering and computer engineering coursework. This has led some researching to borrow from the tools and pedagogical techniques of courses in these fields [6].

Introductory level courses also cover the basic ideas of computer architecture. A simulator demonstrating an over-simplified Von Neuman architecture can make the concepts clearer for an introductory student. These simulations are highly simplified, since the complexities of computer architecture quickly outgrow the scope of an introductory computer science course.

2.2.1.2 Introduction to Computer Organization

Simulators are frequently used for courses covering the basics of computer organization and assembly language programming. A course of this type is taught to sophomore level students of computer science and is seminal in bridging the knowledge gap between what a computer can do and how it is actually done. Educators have found that both top-down and bottom-up approaches can be effective in exposing

the relevant concepts [14, 24]. Since the top-down approach is accessible to a larger audience, it is used in the following paragraphs to extrapolate the ways simulation can be used for computer organizational courses, starting from a problem statement through multiple layers to execution of basic tasks.

The highest level of abstraction is expressed in terms of assembly language, a series of computational and control instructions. Individually, each instruction has a very minimal, but precise scope. For example, an assembly instruction may add two numbers. Assembly language code may also make decisions and navigate to other parts of the assembly code. These decisions are simple, such as “if this value is zero, go to this other instruction.” While each instruction is minimal in scope, they can be aggregated to perform complex tasks.

The assembly language used by programmers is largely defined by the instruction set architecture. The *instruction set architecture* is defined as the complete specification of the interface between programs that have been written and the underlying computer hardware that must carry out the work of those programs [11]. This specification includes the data types allowed by the assembly language, what operations can be done, where the values can be stored, how many values can be stored, and how those values can be accessed. The ISA is a template for processor designers to follow. In current industry, the most common ISA is referred to x86. While processing power has advanced significantly, this same ISA has been at the root of processors created for several decades by multiple companies.

The physical implementation of the ISA specification is called the microarchitecture. This layer of abstraction is the primary focus of most computer organization courses. How the components are physically and logically arranged determines how quickly each operation can be completed. Patt and Patel compare the microarchitecture to an automobile [11]. The ISA equivalent of an automobile would consist of the placement and function of the pedals, the ability to turn, ability to move forward and backward, and the other common factors that define a car; however, the microarchitecture constitutes the way the components are arranged to make these things happen: the size of the engine, the transmission gear ratios, the braking mechanisms, the exhaust system. As an extension to this analogy, there are many trade-offs that take place when designing and arranging these components. A car with a larger engine will go faster, but it will consume more fuel. Similarly, a microarchitecture with a higher clock-speed will generally consume more electricity and may run at a higher temperature. A particular gear ratio in an automotive transmission will allow faster acceleration at the cost of top speed. Similarly, a microarchitecture with pipelining features will execute sequential code more quickly, but it may take a penalty when executing branches. The term datapath is often used interchangeably with microarchitecture, though it is actually a subset. The datapath refers to the collection and arrangement of components, without regarding the controls that “drive” execution through those components. For the purposes of this thesis, the terms are used synonymously.

The components used in the microarchitecture are composed of logical gates such as AND, OR, and NOT. While this topic is covered by most introductory computer science courses, computer organization courses often expand upon the concepts and demonstrate a more realistic application. The very specific details of digital logic are left to electrical engineering and computer engineering courses.

2.2.1.3 Advanced Computer Organization

Most universities offer courses covering the more advanced topics of computer organization. No consensus was found for how these topics are covered in terms of which courses contain what content. In most cases, the concepts are divided and covered as a small part of multiple courses, each with broader scope. One such example is the inclusion of memory caching concepts within Appalachian State University's courses called Systems I and Systems II. In larger departments, a concentration in computer system engineering is sometimes offered, with multiple courses expanding greatly on one particular subtopic such as parallelization [21]. The availability and structure of these courses varies greatly between universities and between undergraduate and graduate programs.

While simulators are often used in these courses, the focus is much narrower. The general purpose simulators for introductory courses focus on usability and accessibility, whereas the advanced simulators are driven by specific features. It is possible for simulators to be both intuitive and feature-rich, but the limited development time

in an academic setting prevents most projects from achieving both goals simultaneously [22, 12, 24].

2.2.2 Pedagogical Styles and the Use of Simulators

In addition to the courses that can benefit from the use of simulators, particular pedagogical styles may benefit from the use of simulators. The styles may also determine the particular feature set required of a simulator [14, 21]. In addition to the variability of course needs, instructors may need to adopt a new style of teaching for a particular course.

In an inter-university study, it was found that Computer Organization courses have three main approaches [7]. The first approach is from an electrical engineering or computer engineering perspective, focusing on digital logic. The second approach is from a computer science perspective, focusing on assembly language and programming concepts. The last approach is to implement the ISA for a given architecture. Many simulators readily exist to assist with the first two pedagogical approaches, depending on the texts used in the course. Implementation of the ISA would generally require the student to create a simulator of his or her own. If the students do not implement their own simulators, they may use an advanced circuit design tool to implement the ISA.

According to Stenstrom and Dahlgren, computer organization courses increasingly focus on trade-off analysis [21]. With the aid of a simulator, students can quickly

go through cycles of making changes, testing, and quantitatively measuring the effects. Without simulation, the cycles' effects would have to be calculated by hand or executed on hardware, slowing down the evaluation of the changes.

Regardless of the pedagogical style used, all sources agreed that simulators aid in student comprehension of the material. In several instances, the added comprehension was attributed to allowing “hands-on” learning [8, 10, 9]. While this seems intuitive, this theme being explicitly stated by multiple authors bolsters the anecdotal evidence.

2.3 Simulation Visualization and Interfaces

Thus far, simulators have been discussed in abstract terms, in what they can accomplish and where they can be used. These ideas neglect the relationship to the user. Equally important is how information is displayed and how the user interacts with them.

At a bare minimum, the user must see the final result of the simulation. This outcome assumes that the use of the simulation is goal-oriented not pedagogically oriented. Simulators based on assembly code often allow intermittent results to be displayed, based on setting breakpoints at particular lines of code. The code is then executed up until that point, and the simulator waits for further instructions from the user. The user can then display the state of the various components and proceed with the simulation. The method of continuing varies between simulators, some allowing

the user to step a single instruction at a time, running until the next breakpoint is reached.

Simulators of the microarchitecture or hardware level often advance in terms of clock cycles instead of assembly instructions. The processing of a single instruction takes multiple clock cycles to load the appropriate values in registers, to interpret the instruction, to gather data, to execute the instruction, and finally to store the result. An assembly level simulator will commonly do all this in one step, glossing over details that are only relevant on an organizational level.

How the data is displayed varies between simulators. A simple method of displaying data is via a *pretty printer* which represents the data as formatted text. This technique is commonly used for assembly level simulators focused on the debugging and results of a program. For digital logic and microarchitecture simulators, it is more common to see graphical displays. In a graphical display, the screen is divided into multiple sections, each representing a different facet of the underlying simulation. What components are displayed and how they are arranged is largely dependent on the focus of the simulation.

The user interface and the visualization method are often codependent and highly dependent on the goals of the simulation. For example, a digital logic simulator designed for exploratory learning will generally allow the user to click through the interface and change the virtual wiring during a simulation. In a similar digital logic simulator designed for demonstrative purposes, the user would not be able to

click through the interface to change wiring configuration. Another simulator may allow both approaches, each in a separate mode. The introduction of such modality increases the complexity of developing and using the simulator. For this reason, designers often opt to avoid particular features, even if they have pedagogical value.

2.4 Taxonomy and Categories of Educational Simulators

Before using and reviewing an existing simulator, it is helpful to have an understanding of how it can be categorized and classified. This allows a level field when comparing simulators by feature and overall goal. Several authors offer taxonomies for informed simulator selection and/or design [24, 15, 25].

2.4.1 Categories of Simulators

Wolfe, Rucik, Osborne, and Holliday have undertaken the task of reviewing and aggregating existing simulators [24], with the goal of listing all educationally relevant simulators in a single place. In the process of reviewing simulators, seven categories were defined, based on what is being simulated. The primary motivation for compiling this list is to provide educators a simpler method for finding simulators for particular topics. According to the authors, these seven categories cover all current simulators used for university level education.

2.4.1.1 Historical

Historical simulators represent older architectures that are either scarce or non-existent in the present day. It is common for a historical architecture to be used to provide a demonstration of a particular concept for which it became famous. No specific historical simulators were reviewed for this thesis, but Wolffe et al. provide a list including PDP-8, PDP-11, Turing Machine, Babbage Analytical Engine, and the Enigma machine [24]. Individuals familiar with the history of computing will note that each of these is an important milestone in the advancement of CPU design.

2.4.1.2 Digital Logic

The focus of digital logic simulators is on basic elements, circuit analysis, timing systems, and low level storage structures. These simulators are used with a bottom-up pedagogical approaches, commonly for computer engineering and electrical engineering courses. DLSim 3, JLS, and Logisim are all educational digital logic simulators and are reviewed in more depth in Chapter 3 [16, 12, 6].

2.4.1.3 Simple Hypothetical Simulators

With the increasing complexity in real machines, it is often necessary to study a simplified counterpart. Simulators in the hypothetical category vary in complexity, depending on the target audience. At the simplest level, Knob & Switch Computer and Little Man Computer are designed for high school or freshman level students

who have little experience with computer hardware [4, 27, 26]. Both of these are described in Chapter 3. Targeting students with slightly more background knowledge, TRISC, EasyCPU, MarieSim, and Micro provide more details about the internal components of the architecture [3, 26, 10, 9]. While these simulators target the same level of student, their approaches for conveying information vary. Both EasyCPU and MarieSim are covered in more depth in Chapter 3. An advantage of using hypothetical simulators is that fine-grain details can be omitted, thus preventing possible confusion for students.

2.4.1.4 Intermediate Instruction Set

Intermediate instruction set simulators extend the functionality of simple hypothetical simulators. In these simulators, additional concepts are introduced, such as memory addressing modes, additional instructions, and memory modeling. There is no hard delineation between hypothetical and intermediate simulators, so there is potential for a particular simulator to belong in a gray area in between. Some intermediate simulators target assembly level programming such as GSPIM and MARS [2]. Others target elements of microarchitecture, such as SimICS and EASE [21, 18].

2.4.1.5 Advanced Microarchitecture Simulators

Advanced microarchitecture simulators add even more complexity, often exposing some method of modifying the datapath or control store. The ability to modify

the datapath allows the user to perform trade-off analysis, comparing how the implementations perform under different configurations. MicroTiger allows complete datapath reconfiguration, including rearranging components, changing connections, and modifying the microcode [22]. Pep8CPU, Mic-1, and CPU Sim do not allow rearranging components but allow the user to modify the control store by editing the microcode [23, 17, 19, 20]. RTLsim takes a more modest approach, allowing the user to set the control signals during each clock cycle [26]. The SLEEP simulator does not directly use the control store, but it allows the user to create components with behavior specified by Java code [13]. Once created, the components can be reused and combined into the datapath. Chapter 3 covers Pep8CPU and MicroTiger in more depth.

2.4.1.6 Multiprocessor

Multiprocessor simulators address the unique challenges of executing a program in a multiple processor environment. To demonstrate these unique elements, the simulators require an enhanced set of features, such as shared memory and simultaneous execution of instructions. Due to the complexity of multiprocessor programming, this category of simulators can be difficult to learn and use. No multiprocessor simulators were reviewed for this thesis, but Wolffe et al. list ABSS, MINT, Proteus, RSIM, and SimOS as members of this category [24].

2.4.1.7 Memory Simulators

Nearly all simulators reviewed contain a memory array as an integral component of the architecture; however, memory simulators focus solely on memory related concepts, often to the exclusion of instruction execution. The concepts covered include hierarchical caching level, cache sizes, and memory associativity. Since this specialized category does not relate to the topic of this thesis, no simulators were reviewed. Wolffe et al. mention Cacheprof, Cache Simulator, CACTI, Dinero IV, RPIMA, and Xcache as examples of memory simulators [24].

2.4.2 Taxonomy of Program Visualizations

Roman and Cox present a general taxonomy for program visualization [15]. Their taxonomy is broad, covering far more than simulations and computer architecture. The taxonomy applies reasonably well to the needs of computer architecture simulation, although the breadth is too generic in some places. The authors present five components in their taxonomy.

2.4.2.1 Scope

The scope defines which conceptual elements of a program are intended to be displayed. In architecture simulation, the scope usually targets the assembly layer, ISA layer, microarchitecture layer, or some combination of these. Generically, the authors present several sub-categories of scope including the code, the data state, the

control state, and the behavior. The selection of the scope determines how the other parts of the visualization will form.

2.4.2.2 Abstraction

The abstraction defines what data is to be displayed. Applied to architecture simulation, this aspect of the taxonomy determines which components will be displayed. For example, the abstraction can include simple components such as logic gates or more complex components such as multiplexers. The choice of which components to include is influenced by the targeted scope and granularity of the simulation. Depending on which components are used, the level of abstraction will determine the behavior of the simulation.

2.4.2.3 Specification

The specification aspect determines what visualization techniques can be employed in representing the abstractions. Generically, an increased range of techniques is often beneficial, allowing the user to view execution from different perspectives. Assembly language programs may similarly benefit from more specification methods; however, additional specification methods may be distracting for digital logic or microarchitecture simulators.

2.4.2.4 Interface

The interface aspect covers how the data will be displayed. This includes whether the data is displayed graphically or in text and what visual components are available to convey this information. An example from architecture simulation is how a memory location is displayed. A text level simulation would only display relevant memory values, whereas a graphical simulator could have a scrollable window containing the full contents of memory. Also defined within the interface aspect is how the user can interact with the visualization. This type of interaction refers explicitly to the visual elements and should not be confused with the interaction with a simulator as a whole. For example, it may be possible to click and drag a three dimensional model to rotate the view, leaving the underlying state of the program the same.

2.4.2.5 Presentation

The presentation aspect involves how the data is translated into information. The semantics of presentation includes how the interface elements are combined to convey the overall state and purpose of the program. For instance, when demonstrating the fetch of an instruction, it is necessary to display at least the program counter, the contents of key memory locations, and the instruction register contents. In simulators where all data is always present, highlighting the relevant parts would be considered a presentation technique.

2.4.3 Taxonomy of Architecture Visualization

Another taxonomy is presented by Yehezkel, tailored specifically for computer architecture visualization [25]. This taxonomy is partially based upon the work of Roman and Cox, described in the previous section. In addition to the architecture focus, the taxonomy applies more specifically to pedagogical purposes. This taxonomy is divided into four primary concepts with subdivision within those concepts.

2.4.3.1 Scope of Operation

The scope of operation addresses the environment in which the simulator will run. This includes the technical specifications required to run the simulation. Further, it defines the limitations of the simulator. These specifications greatly affect the portability and what types of systems can run the software.

2.4.3.2 Content Modeling

Content modeling addresses how literal the simulator is in representing the underlying architecture. At the most literal, the model has a one-to-one correspondence to the components of the architecture. Since all elements are taken into account, this approach is quite complex and presents challenges in visualizing all components. As a consequence, one-to-one modeling is reserved for professional purposes, rather than educational ones. The simplified model allows the designer leeway to reduce the complexity while still addressing the overall operation of the architecture. Since

not all sub-components are represented, the complexity of the tool is significantly reduced, making it more suitable for educational purposes. Even further removed is a hypothetical model wherein general concepts are presented without being tied to a real architecture. Because of this simplicity, these hypothetical models are good for introductory classes, since important details may be excluded without hindering comprehension. The models may not be completely accurate representations of the architecture being simulated, however.

2.4.3.3 Presentation Methods

The presentation methods include the visual arrangement of the components, how data is represented (decimal, binary, hexadecimal), and the methods used to emphasize certain elements over others. When simulating a complex architecture, it can be advantageous to retain a complex content model but simplify the presentation of that model. By doing so, it is feasible to provide an alternate presentation for users who have a greater knowledge of the material.

2.4.3.4 Activity Style

The activity style determines how the user interacts with the simulation. The role of the user is considered in determining how interactive the simulation must be. Simpler roles make it easier for the users to absorb individual concepts. More complex roles allow the student to understand more concepts and how those concepts relate

to one-another. Multiple roles can be defined and implemented by a single simulator. The activity style also defines how the user interacts with the simulator. This includes role-based interaction and control-based interaction. If the role is to handle the input and output, the role-based interaction would involve creating the input and storing the output. The same scenario would need control-based interactions to step through the code that processes the input.

Chapter 3 - Example Simulators

There are already quite a few simulators available for educational purposes, geared towards differing levels of education and exhibiting varied complexity. In reviewing what is available, ideas can be gleaned for the design of a new simulator and to find areas that are currently lacking in existing simulators. For the purposes of brevity, only the basics of each simulator are described, highlighting key elements that distinguish them from other simulators. The reader is encouraged to refer to the original literature for more detailed information.

3.1 Introductory Simulators

Simulators used for introductory level courses are generally the simplest to use and have the fewest available features. They demonstrate the concepts of computing, rather than an implementation of existing hardware. Three of these simulators are reviewed below.

3.1.1 Knob & Switch Computer

The Knob and Switch Computer, also referred to as K&S, was created to emphasize the basic concepts of computer organization [4]. The design focused on two concepts that differ from previously implemented simulators. First it was designed to employ “cognitive hooks,” allowing students to leverage their existing knowledge of interacting with a digital system such as a stereo. Second, it was designed to introduce a single component at a time, gradually adding to the complexity of the processor.

As the name implies, the simulator is created as a series of knobs and switches. This paradigm was chosen to leverage student’s intuitive understanding of how knobs and switches work in other contexts, such as on a stereo. An image of a knob is used to select a single value out of several possibilities. A switch is used to toggle a value on or off. The simulator is hosted as a web application, allowing students to access the simulator from any system that has a web browser. Within the web interface, knobs can be rotated between values and switches can be toggled. The use of these components as cognitive hooks makes the simulator intuitive even for novice users.

The simulator is presented in stages, each stage adding concepts and complexity over the last. As shown in Figure 3.1, the first stage illustrates a simple data path with several registers, an ALU, and buses connecting them. The user selects two inputs for the ALU via two separate knobs. A third knob selects the operation to

perform within the ALU. Finally, a fourth knob selects where the result will be stored. The second stage adds memory and switches to control if values come from/go to registers or memory. The third stage introduces the concept of a control unit where the user can forego clicking the knobs and switches by writing simple binary microcode. In its final stage, shown in Figure 3.2, the concept of an assembly language is presented, building upon the microprograms and including some additional components to the data path.

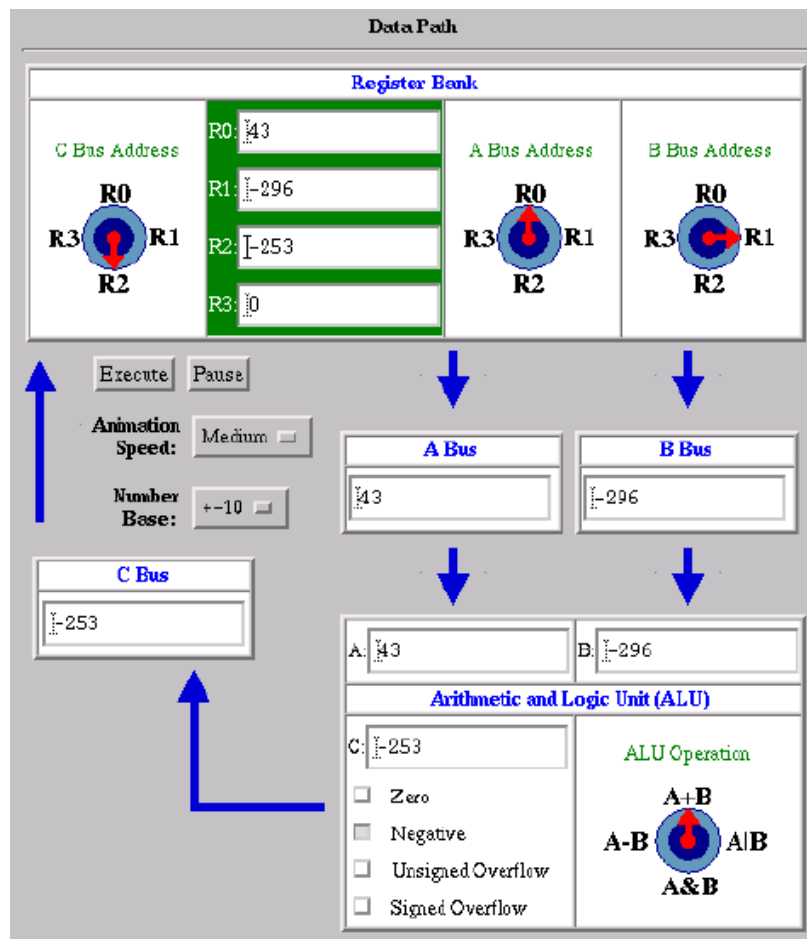


Figure 3.1: First Stage of Knob & Switch Simulator

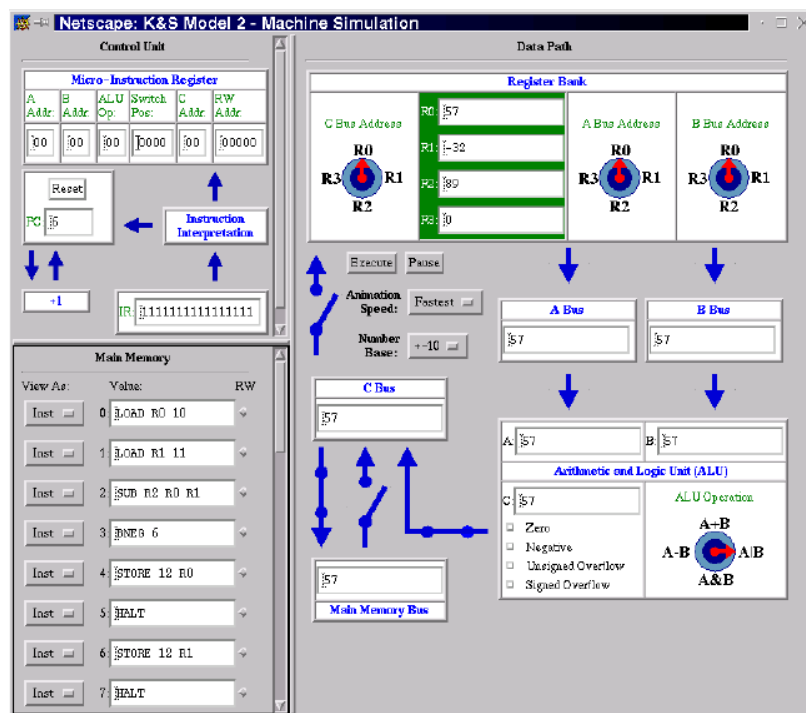


Figure 3.2: Final Stage of Knob & Switch Simulator

The staged approach is reported to work well for introductory courses; however, by the author’s admission, the final stage is not sophisticated enough for a computer organization course and does not sufficiently address the differences between assembly language and the machine language of the microcode.

3.1.2 Little Man Computer (LMC)

The Little Man Computer, shortened to LMC, is also designed for use in introductory computer science courses [27, 26]. The emphasis is on teaching a basic Von Neumann architecture through a hypothetical scenario. The scenario intentionally does not involve real computer components.

LMC follows the actions of a little man who is walled into a room. The room contains a series of mailboxes, a calculator, a 2-digit counter, and baskets for input and output. The only communication with the world outside his room is through the two baskets. Those familiar with basic architecture will recognize the mailboxes as a simple memory array and the calculator as an ALU. The user can only interact with the little man via written instructions dropped into the input basket.

As seen in Figure 3.3, the simulator contains all the elements presented by the scenario, with the exception of the little man. In spite of the minimal information being presented and processed, the analogy conveys rudimentary, yet important, concepts of computer architecture. In spite of its simplicity, LMC does cover the slightly more advanced topic of addressing modes. This inclusion was found to expand student understanding from previous courses, such as the use of pointers and basic data structures like linked lists.

3.1.3 EasyCPU

EasyCPU was also designed for introductory level computer science courses. The emphasis is the execution of a single instruction [26]. Unlike other simulators targeted for introductory students, EasyCPU is based on the real-world x86 architecture, albeit with a reduced instruction set. The visualizations for EasyCPU show data flow through a bus, but they lack finer-grain details such as control signals.

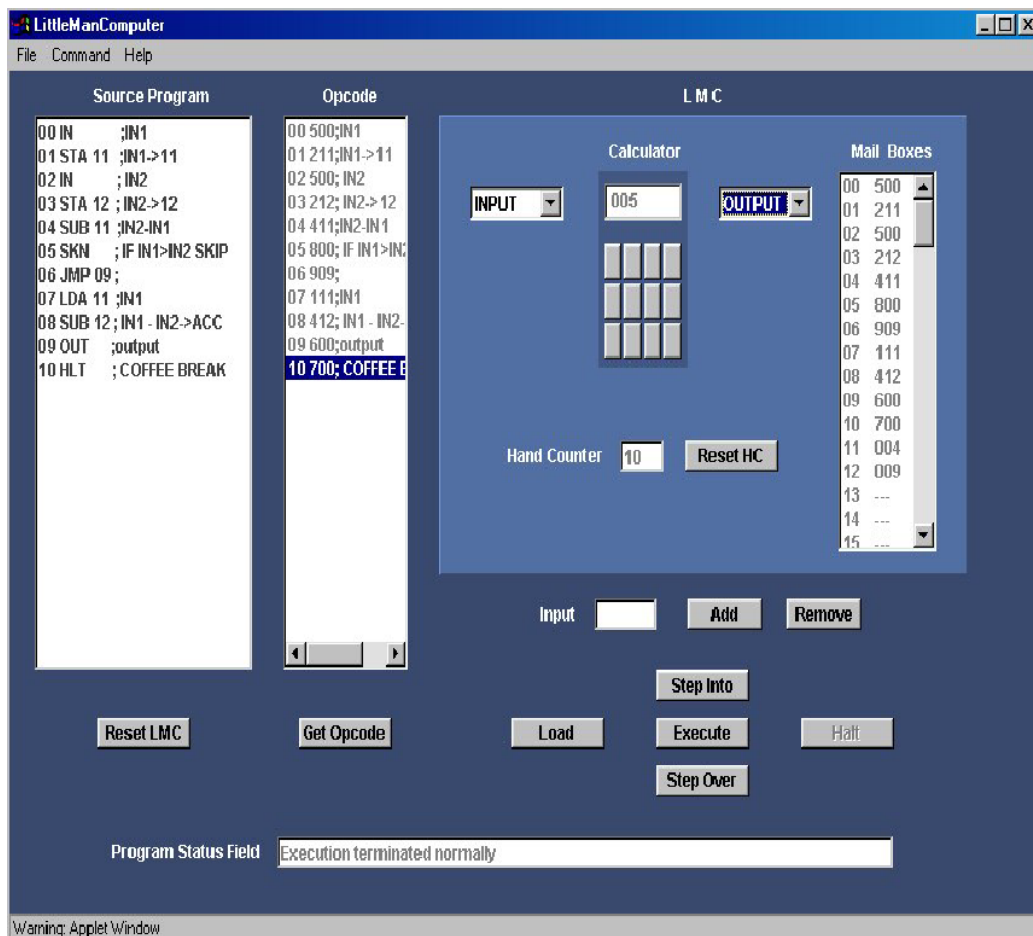


Figure 3.3: Little Man Computer

EasyCPU has two forms. The basic form focuses on the execution of a single instruction. As Figure 3.4 shows, the data, control, and address buses are central to tying together the components. In the advanced form, shown in Figure 3.5, the student can write multiple assembly instructions, execute them, and watch how the instructions modify the component values.

Having two modes was found advantageous, allowing students to use the same tool before and after learning x86 assembly language. One drawback found was that

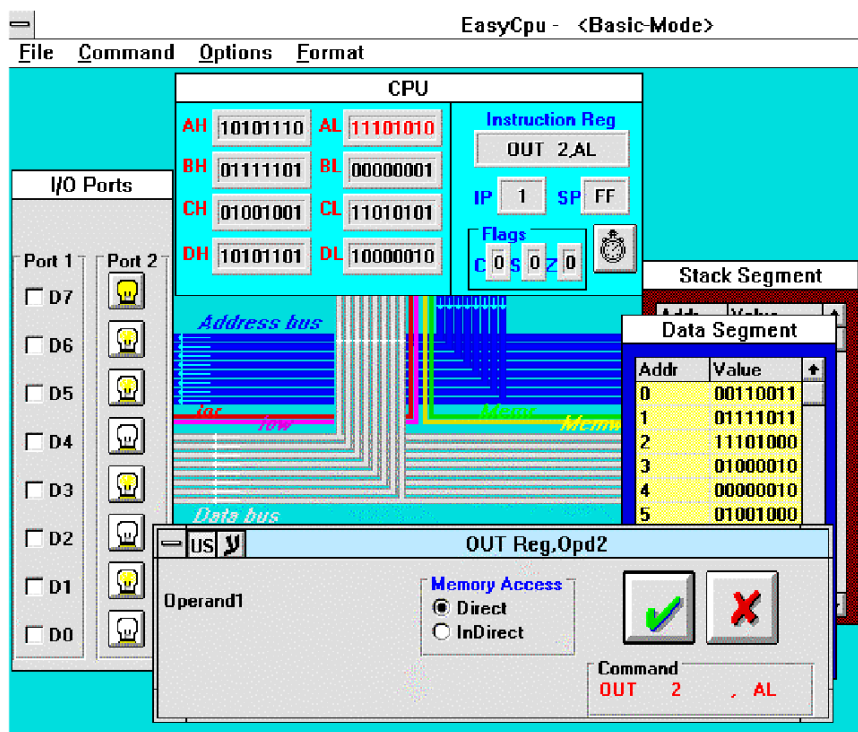


Figure 3.4: EasyCPU Basic-Mode

the EasyCPU was compiled only for Windows systems. This may prevent some students from using the simulator on their home machines.

3.2 Digital Logic Simulators

The next category of simulators reviewed are those designed for teaching digital logic. For some simulators, the scope stops at the digital logic level; in other cases, the simulator is able to expand these concepts towards the building of a CPU. While a CPU consists of only digital logic gates, the size and complexity makes it a challenging target for digital logic simulators. The large number of circuits needed

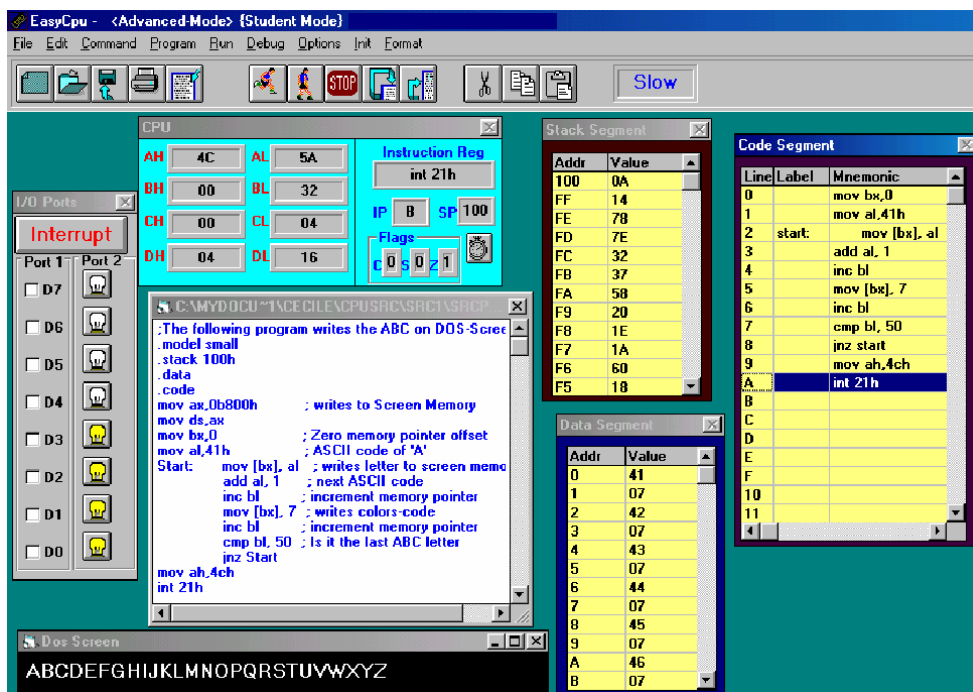


Figure 3.5: EasyCPU Advanced-Mode

makes execution of the simulation computationally inefficient for an architecture of moderate complexity. It can also be difficult to display the circuits in a navigable and meaningful way.

3.2.1 Logisim

Logisim is the only simulator reviewed that was explicitly built for digital logic simulation [6]. The simulator was designed for introductory level computer science classes. It was also designed to be portable, selecting Java as the implementation language. The emphasis of the design is to make the creation, editing, and execution of digital circuits as intuitive and easy as possible. For example, the simulator supports

custom wire paths, allowing intricate layouts beyond a straight line between components. The visualization is done via color coding, so there is no need for text-based labels for wire values. It also allows arbitrary input count for some components, such as AND and OR gates, instead of having to aggregate multiple 2-input components. Figure 3.6 shows the editor designing a simple circuit.

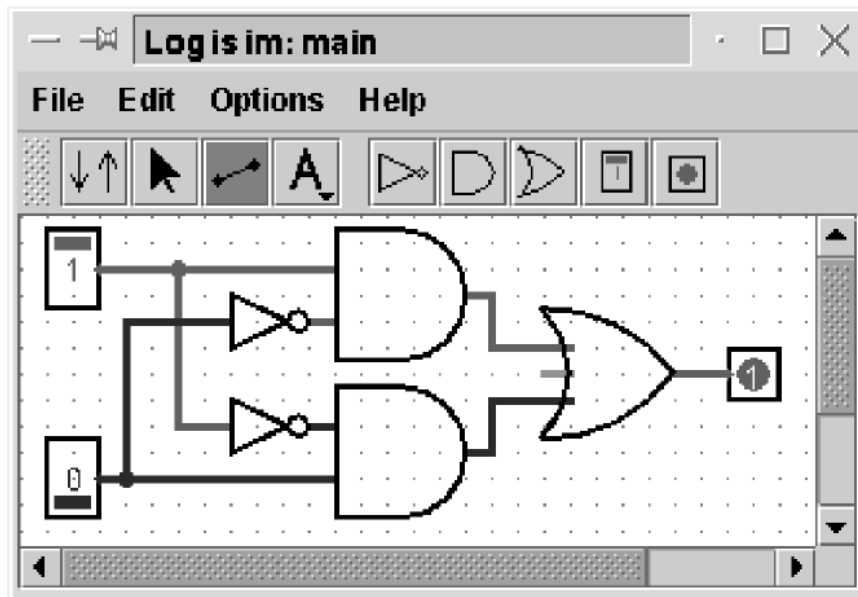


Figure 3.6: Logisim Editor

By the author's admission, a few short-comings make this simulator limited in its applicability. In terms of features, the inability to bundle wires prevents it from scaling to more complex circuit designs. The display of subcircuits is fixed and prevents readability when subcircuits are included. In addition, the authors mention Java as a limitation, making the simulator harder to install and launch than natively running programs.

3.2.2 DLSim 3

The objective of DLSim 3 is to build all the fundamental components of a CPU [16]. The simulator graphically displays basic logic gates and allows the user to add and remove components. As a language choice, the designers chose Java, allowing portability between environments. This simulator is unique in how it handles extensibility and scalability. DLSim 3 allows three methods for creating more complex structures. After creating a circuit, the user can export it as a *card*, which is allowed to be imported later as the same circuit. This export allows the user to clone pieces of functionality but not lose the details of the unit's inner workings. The user can also export a circuit as a *chip*, which maintains the functionality but is imported later as a single unit. This allows the user to preserve the overall function of the circuit without exposing the underlying complexity. Finally, the simulator allows developers to create *plug-ins* that can be significantly more powerful than would be allowed by the circuit design capabilities of the existing tool. These allow for significantly greater expansion, though they require knowledge of Java programming to implement.

Figure 3.7 shows the circuit design for a sequential multiplier. It is worth noting that the figure shows basic logical gates such as OR along-side aggregate components such as the adder.

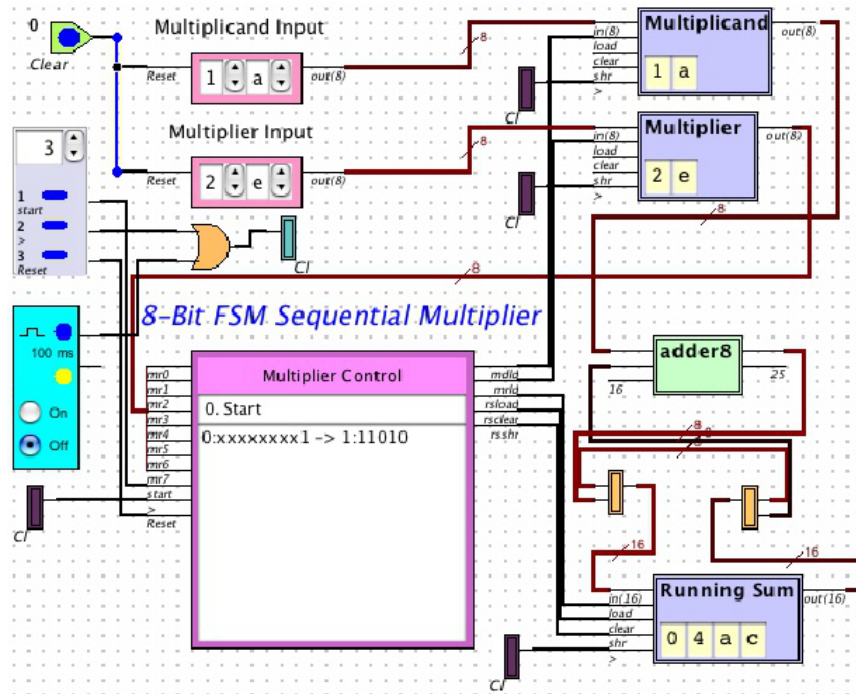


Figure 3.7: DLSim 3

3.2.3 JLS

To combat the non-portable, text-based simulators found in the industry, JLS was designed for educational use [12]. As shown in Figure 3.8, the simulator has a comprehensive selection of components to add to the circuit. Circuits can be built, saved, duplicated, and aggregated into more complex circuits. Additionally, JLS contains a teacher-specific mode that allows rapid grading of student work.

JLS contains two unique mechanisms for building more complex components. The user is allowed to specify the behavior of a component via a truth table or a finite state machine. Figure 3.9 shows an example of the FSM editor.

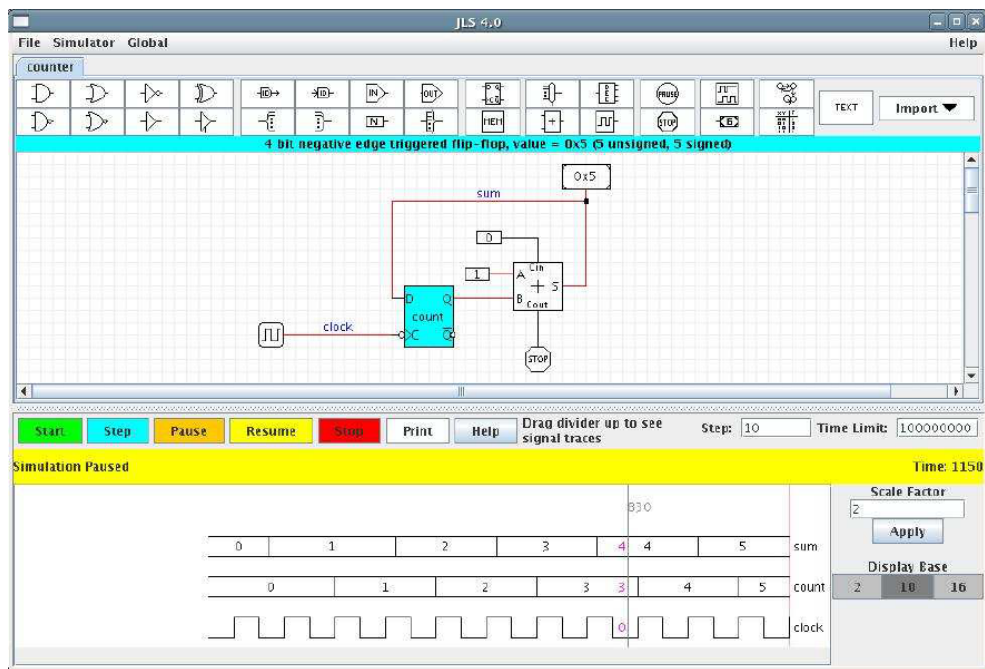


Figure 3.8: JLS Circuit Editor

3.3 Microarchitecture Simulators

The following section focuses on simulators that target the microarchitecture level. In some cases the simulator also covers digital logic or assembly language programming, but the emphasis is on microarchitecture simulation. Of the reviewed simulators, these most closely match the goals and features of lc3uarch.

3.3.1 MicroTiger

The MicroTiger simulator is a full-featured simulator for sophomore level students [22]. The designer cites the lack of graphical simulators with configurable datapath as a main motivation for creating MicroTiger. The simulator allows creation and

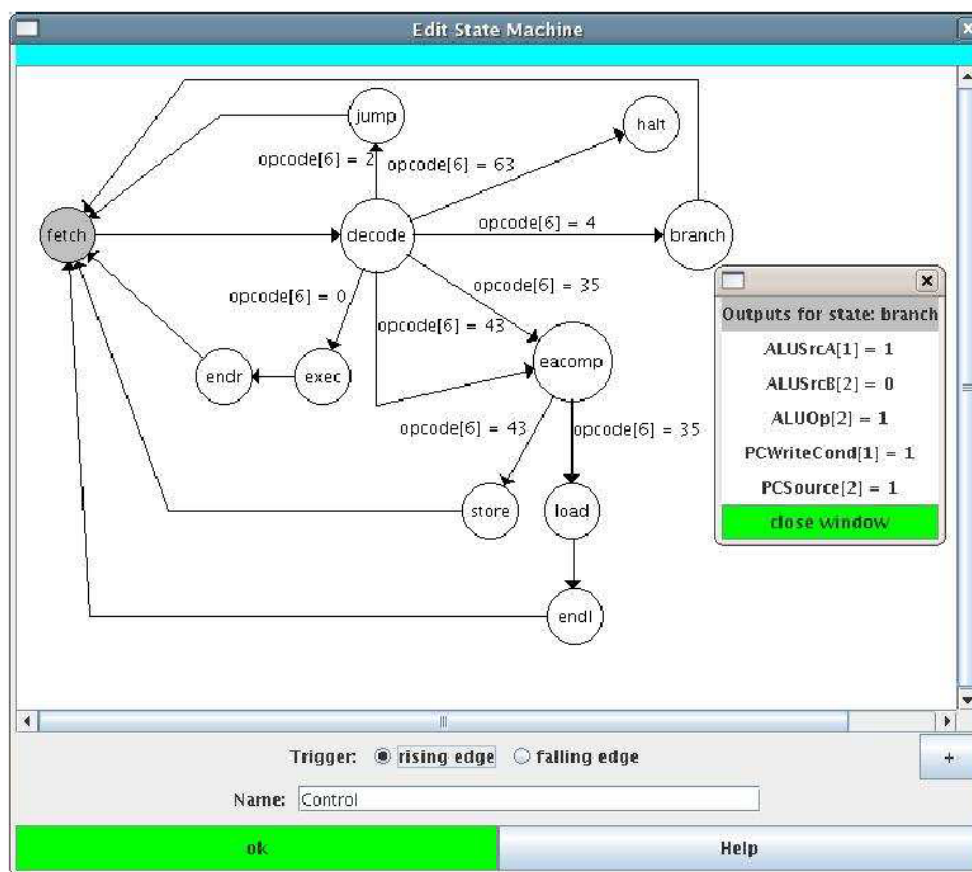


Figure 3.9: JLS FSM Editor

modification of a datapath in a manner similar to most circuit editors, using registers, RAM, and other high level components, instead of digital logic gates. Figure 3.10 shows an example of the editor with registers, multiplexers, demultiplexers, memory, and an ALU.

MicroTiger provides a text based microcode editor, as shown in Figure 3.11. The custom microcode language allows the user to determine the assertion of control signals, thus creating a complete control store.

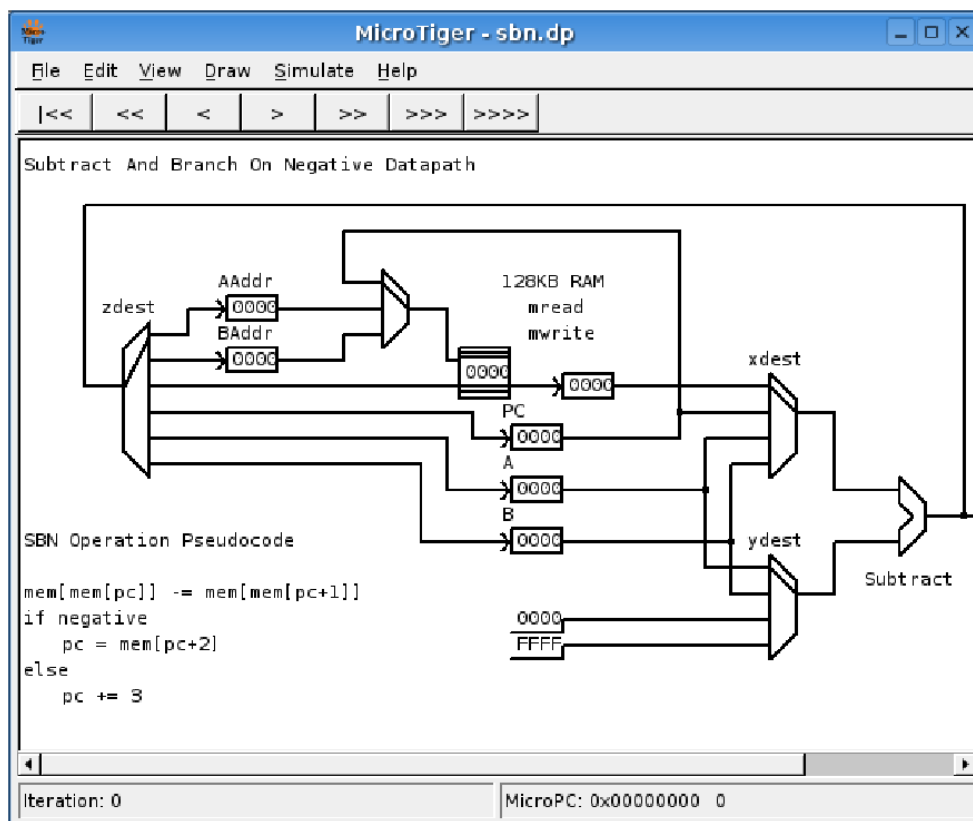
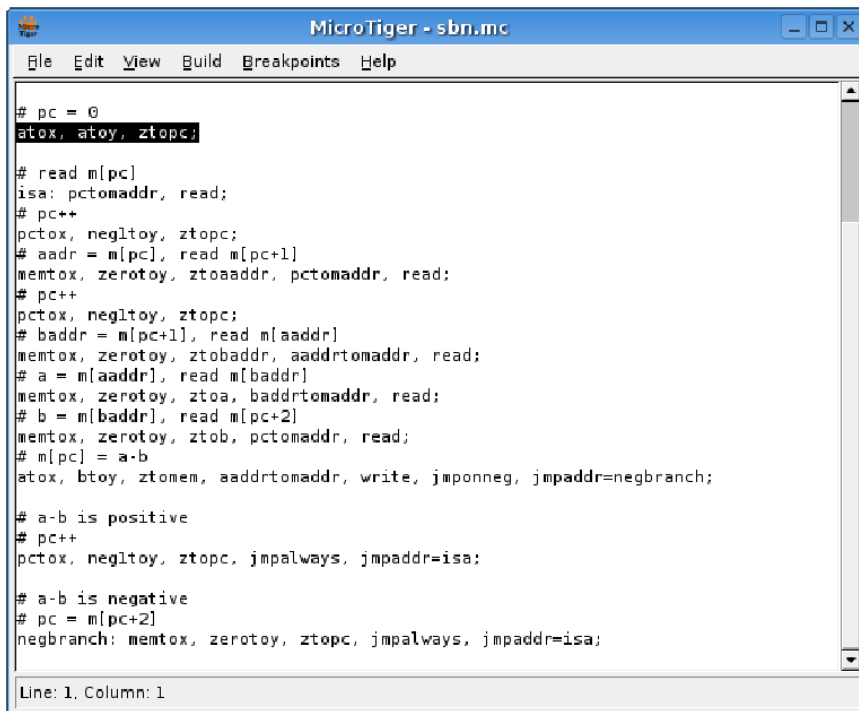


Figure 3.10: MicroTiger Datapath Editor

By allowing the user to edit both the physical configuration of components and the microcode being executed, MicroTiger is extremely flexible compared to most architectural simulators. Being unrestrained makes MicroTiger very useful for advanced study in advanced topics such as trade-off analysis; however, this flexibility comes with the cost of operational slowness and interface complexity. To create or modify a circuit, many pop-up dialogs are used, which may confuse and deter novice users.



```
MicroTiger - sbn.mc
File Edit View Build Breakpoints Help
# pc = 0
atox, atoy, ztopc;
# read m[pc]
isa: pctomaddr, read;
# pc++
pctox, negltoy, ztopc;
# aadr = m[pc], read m[pc+1]
memtox, zerotoy, ztoaaddr, pctomaddr, read;
# pc++
pctox, negltoy, ztopc;
# baddr = m[pc+1], read m[aaddr]
memtox, zerotoy, ztobaddr, aaddrtomaddr, read;
# a = m[aaddr], read m[baddr]
memtox, zerotoy, ztoa, baddrtomaddr, read;
# b = m[baddr], read m[pc+2]
memtox, zerotoy, ztob, pctomaddr, read;
# m[pc] = a-b
atox, btoy, ztomen, aaddrtomaddr, write, jmponneg, jmpaddr=negbranch;
# a-b is positive
# pc++
pctox, negltoy, ztopc, jmpalways, jmpaddr=isa;
# a-b is negative
# pc = m[pc+2]
negbranch: memtox, zerotoy, ztopc, jmpalways, jmpaddr=isa;
Line: 1, Column: 1
```

Figure 3.11: MicroTiger Microcode Editor

MicroTiger is written in C++ using WxWidgets for graphical visualization. This combination allows the simulator to be easily ported to many platforms. While the porting process is simple, this necessitates recompilation for each platform, resulting in a binary for each platform.

3.3.2 Pep8CPU

The Pep8CPU simulator was designed with an emphasis on microcode [23]. The simulator implements two modes. The first mode requires the user to enter control signals manually. On the right side of Figure 3.12, are a series of check boxes and numerical inputs that control the components during a clock cycle. The second

mode uses custom microcode to set control signal values during each clock cycle. The microcode editor, shown in Figure 3.13, features syntax highlighting for the user's convenience.

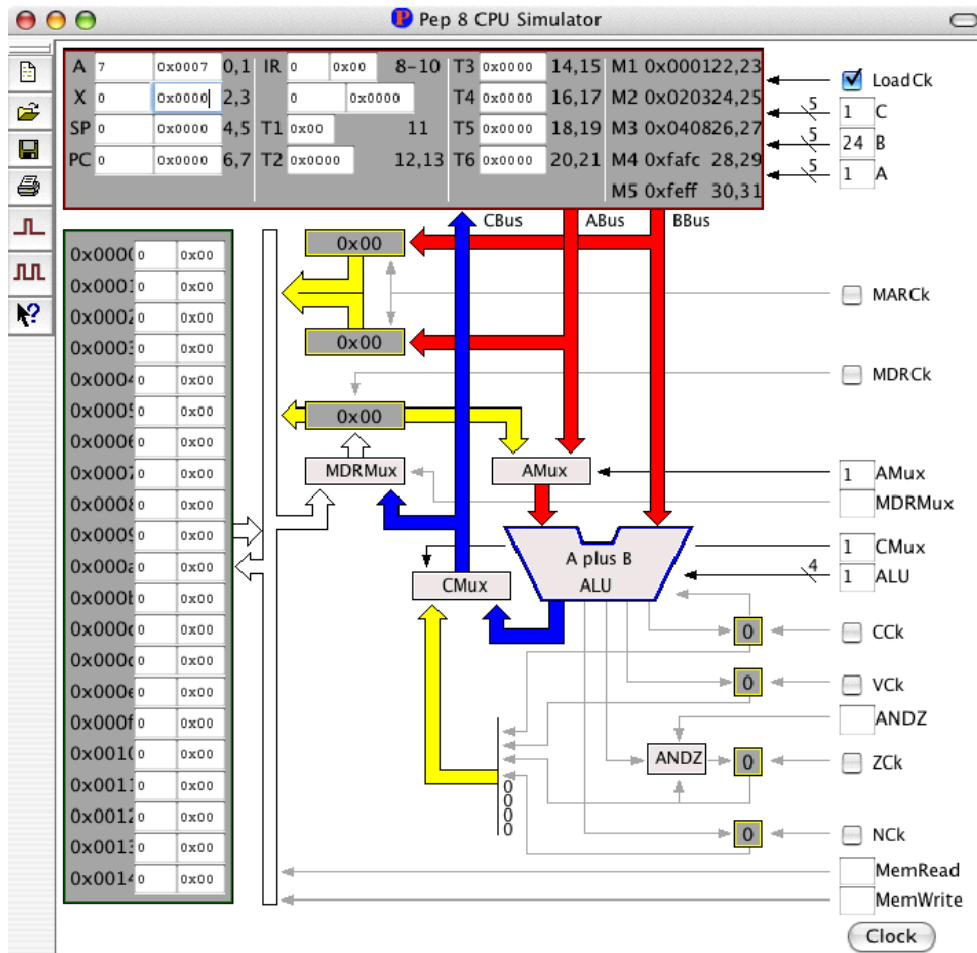


Figure 3.12: Pep8CPU Datapath

Not allowing modification of the physical datapath layout, Pep8CPU has a clean interface that can quickly be understood for either mode. The key drawback is that the relationship between assembly instructions and microcode is not emphasized.

```
PCS Editor - '/Users/warford/Desktop/...
// File: fig1205.pcs
// Figure 12.5
// Fetch the instruction specifier and increment PC by 1

// Save the status bits in T1
1. CMux=0, C=11; LoadCk

// MAR < PC, fetch instruction specifier.
2. A=6, B=7; MARCk
3. MemRead
4. MemRead, MDRMux=0; MDRCk
5. AMux=0, ALU=0, CMux=1, C=8; LoadCk

// PC < PC + 1, low-order byte first.
6. A=7, B=23, AMux=1, ALU=1, CMux=1, C=7; CCK, LoadCk
7. A=6, B=22, AMux=1, ALU=2, CMux=1, C=6; LoadCk

Line: 12, Col: 32
```

Figure 3.13: Pep8CPU Microcode Editor

Warford and Okelberry leave it to the instructor to make this connection for the students.

3.3.3 MarieSim

MarieSim is targeted at introductory level computer architecture courses [10]. As the full name (Machine Architecture that is Really Intuitive and Easy) implies, the simulator has the primary objective of being easy to use. The simulator displays the datapath for a 16-bit accumulator based CPU. Accumulator based CPUs have lost favor in modern real-world CPUs, but the concepts are still relevant for educational purposes. An additional benefit is a reduced number of components, allowing

everything to be displayed on the same screen. As Figure 3.14 shows, the simulation can be executed to completion or one clock cycle at a time. MarieSim also provides an environment for executing code with no datapath visualizations.

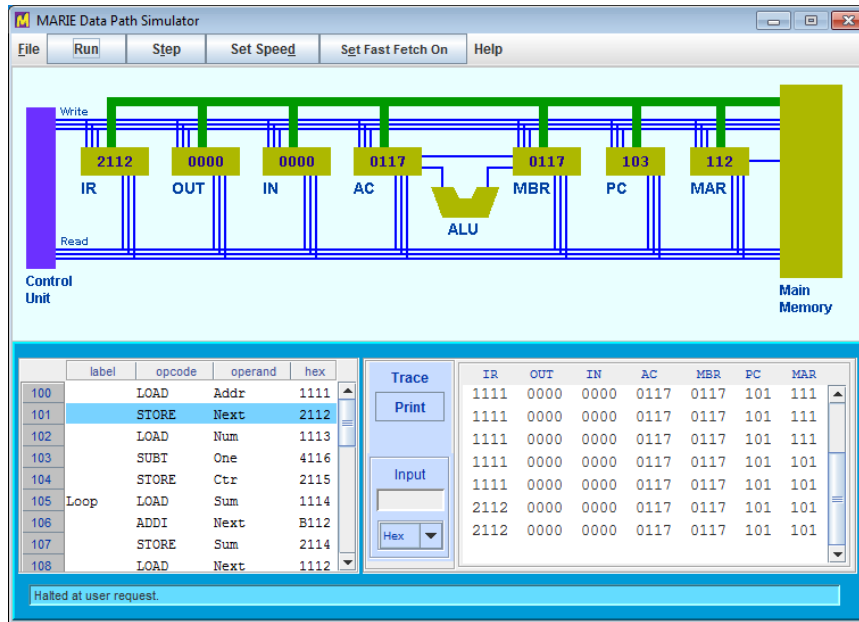


Figure 3.14: MarieSim Datapath

3.3.4 RTLSim

RTLSim is designed to have the user act as the control store for the microarchitecture [26]. The main display contains a graphical representation of the datapath and a series of checkboxes, as shown in Figure 3.15. Optional windows for memory, register values, and a traceback of previous control signals are also shown. As the user sets control signals, the relevant components on the datapath are highlighted. In the case that conflicting control signals are selected, the components will be highlighted

in red, indicating an error. Since the user is the control store, execution proceeds only in single clock cycle increments, with no mechanism to execute an entire instruction.

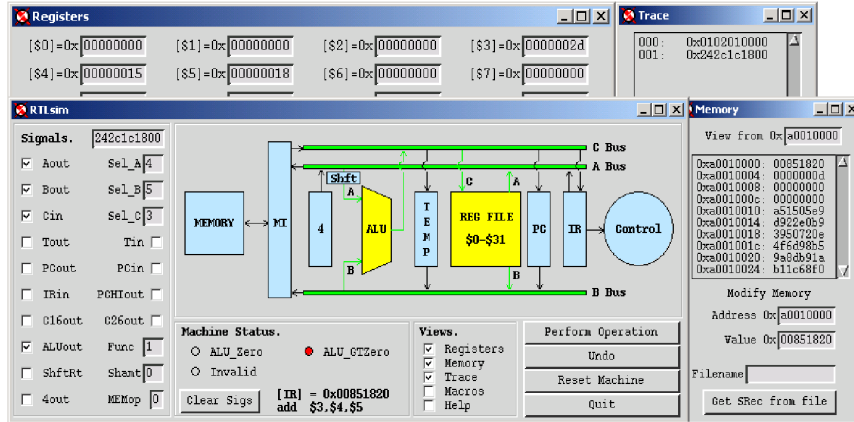


Figure 3.15: RTLsim Datapath

Chapter 4 - LC-3

The LC-3 architecture is a hypothetical processor used solely for educational purposes. While lacking features of many real-world architectures, the chosen features form a complete architecture that could be built into a physical product. Focusing on a subset of common real-world features allows the appropriate depth for an introductory level course.

4.1 Instruction Set Architecture

Patt and Patel define an Instruction Set Architecture (ISA) as “the complete specification of the interface between programs that have been written and the underlying computer hardware that must carry out the work of those programs” [11]. The specification for the LC-3 processor includes the memory structure, the types and number of registers, provisions for interrupt handling, and the available instructions. As described in Chapter 5, only a subset of the ISA is implemented by `lc3uarch`.

4.1.1 Memory

LC-3 registers are 16 bits, thus supporting a memory size of 2^{16} (65,536) locations. The memory addresses start at 0, generally expressed with four hexadecimal digits from 0x0000 to 0xFFFF. The array is partitioned into several areas, each dedicated to a specific purpose. The lowest portion is allocated to the trap vector table, associated with the TRAP instruction. TRAP instructions serve as system calls, executing tasks such as halting a program, reading characters from the keyboard, or printing characters to the screen. Above the trap vector table is the interrupt vector table, for handling external signals such as receiving a key press from a keyboard. Higher in the memory array is a portion reserved for the operating system and the supervisor stack. The responsibilities and capabilities of an operating system are beyond the scope of introductory architecture course. The supervisor stack is covered briefly in a future section. Directly above is the largest memory segment, available to user programs. The highest and smallest section of memory is reserved for device registers. These addresses are used for memory-mapped input and output. The full memory layout is provided in Figure 4.1.

4.1.2 Registers

The ISA defines 8 general purpose registers, a program counter, and the condition code registers. The program counter (PC) contains the address of the next instruction to be executed. By default, the PC is incremented with each instruction

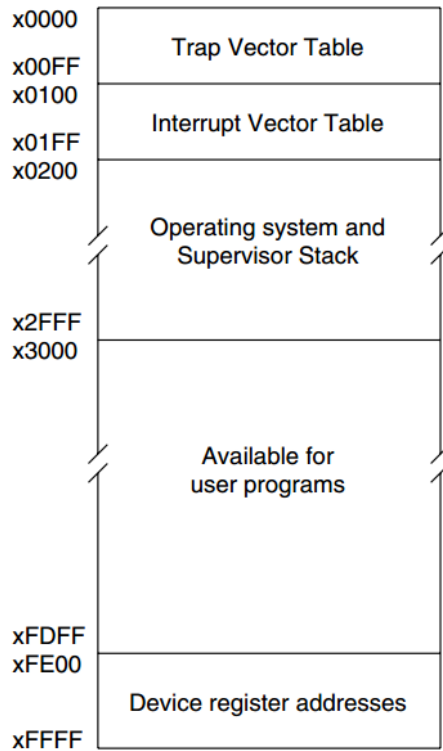


Figure 4.1: LC-3 Memory Layout

fetch. Other methods can update the program counter depending on the nature of the current instruction being executed. For example, the jump instruction will change the PC to a specific memory address. There are 8 general purpose registers (GPR) each holding 16 bits of data. As the name “general purpose” implies, these registers are not associated with a specific task. The purpose varies depending on the instruction using them. Additionally, there are three 1-bit condition code registers that are set when a load or operating instruction is executed. All other instructions leave the condition codes unchanged. The condition registers indicate if the recently calculated 2’s complement value is negative, zero, or positive (NZP).

4.1.3 Interrupts and Exceptions

The Instruction Set Architecture defines several elements related to privilege levels, the handling of exceptions, and the handling of interrupts. Privilege levels allow a distinction between system level and user level execution. In the instruction set, only one operation is reserved for higher level privileges: RTI. An interrupt is triggered by an external event, often I/O. When detected, the CPU stops execution of the current process, responds to the external event, and resumes the process. Only a single interrupt method is currently defined on the LC-3, keyboard input, but room is left for additional interrupts in future revisions. Exceptions are caused by an illegal sequence of events. Two exceptions are specified: an illegal opcode and privilege mode violation. Of the 16 available opcodes (4 bits are used to specify the opcode), one is reserved for future expansion and currently raises the illegal opcode exception when used. If the RTI instruction is used while the processor is not in a higher privilege level, the privilege mode violation exception occurs. The control logic for the LC-3 allows the handling of interrupts, exceptions, and privileged mode with the aid of several specially purposed registers to track the processor state.

4.1.4 Instructions

The instructions for the LC-3 are 16 bits with the first four bits, called the opcode, identifying the instruction. In some cases, a single opcode yields multiple mnemonic assembly instructions, based on other bits within the instruction. Detailed

information on the bit-by-bit format for each instruction is provided in Figure 4.2 as reference.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD*	0001			DR			SR1			0	00		SR2			
ADD*	0001			DR			SR1			1	imm5					
AND*	0101			DR			SR1			0	00		SR2			
AND*	0101			DR			SR1			1	imm5					
BR	0000			n	z	p	PCOffset9									
JMP	1100			000			BaseR			000000						
JSR	0100			1	PCOffset11											
JSRR	0100			0	00		BaseR			000000						
LD*	0010			DR			PCOffset9									
LDI*	1010			DR			PCOffset9									
LDR*	0110			DR			BaseR			offset6						
LEA*	1110			DR			PCOffset9									
NOT*	1001			DR			SR			111111						
RET	1100			000			111			000000						
RTI	1000			000000000000												
ST	0011			SR			PCOffset9									
STI	1011			SR			PCOffset9									
STR	0111			SR			BaseR			offset6						
TRAP	1111			0000			trapvect8									
reserved	1101															

Figure 4.2: LC-3 Instructions

For mathematical and logical operations, AND, ADD, and NOT are provided. Both AND and ADD can use either two source registers or a single source register paired with an immediate value. The NOT operation inverts the bits of the specified source register and stores the result in the destination register.

For navigating between the instructions within a process, four instructions are specified: BR, JMP, JSR, JSRR, and RET. These allow the code to “branch” depending on condition codes, “jump” unconditionally to a specific instruction, “jump to subroutine” based on the specified location, “jump to subroutine” based on a location in a register, and “return” from a subroutine.

To load values from memory into a register LD, LDI, LDR, and LEA are used. LD loads a value from memory located at the address “offset + program counter.” A value can be “loaded indirectly” with LDI, using the value in “offset + program counter” as the address of the target value. Similar to LD, LDR uses an offset but adds it to the address specified in a register instead of the program counter: “offset + register value.” Instead of loading a value from memory, LEA “loads an effective address” by calculating “offset + program counter” and storing that value in the destination register. To store values ST, STI, and STR work similarly to the mnemonically relevant load instructions, in reverse. Instead of memory being the source of the value, it is the destination.

The final three instructions are RTI, TRAP, and “reserved.” As the mnemonic “return from interrupt” implies, RTI is used to return from an interrupt. During the execution of this instruction, the privilege mode and stack pointers are updated, possibly resulting in a privilege mode violation. The TRAP instruction triggers the execution of a system call. The available TRAP routines provide several operations at a higher level of abstraction: read a character from the keyboard, print a character

to the console, print a string to the console, or halt the execution of a program. The final “reserved” instruction is set aside for possible future expansion. If the reserved instruction is executed within any program, the illegal opcode exception will occur.

4.2 Microarchitecture

Patt and Patel provide a detailed microarchitecture that implements the Instruction Set Architecture. The required components are arranged and supplemented with additional components to form the datapath. The authors also specify a finite state machine to set the control signal values during each clock cycle. The design and documentation are thorough enough to implement a functional CPU.

4.2.1 Datapath

In addition to the required registers and memory array, the datapath contains supplemental registers for controlling the memory array, a bus for communication between components, multiplexers (MUX), sign extenders (SEXT), zero extenders (ZEXT), adders, an arithmetic logic unit (ALU), and gates to control access to the data bus. The datapath is expressed in two main forms in the text: with and without interrupt handling, exception handling, and privilege mode.

In both forms, several conventions are used. On a line connecting two components, a slash and a number specifies the width of the connection, measured in bits. Unless otherwise specified, the width is assumed to be the same as that of the

most recent connection. At certain points, a set of lines is forked, propagating the connection to two components. In these cases, a solid dot is used to denote the fork. In some locations, the format “[X:Y]” is used, indicating a subset of the lines. The possible values for X and Y range from 0 for the rightmost bit to 15 for the left-most bit. For example, “[4:0]” indicates that the 5 rightmost bits are used.

To avoid visual clutter, control signals are specified with a hollow arrow-head with no indication of the source. The implied source is the control unit which is described in the next section. Similarly, the tristate gates, such as GateALU, do not indicate a control source. Again, the implied source is the control unit. When a gate’s control line is asserted, the input at the base of the triangle flows to the output at the point of the triangle. If the gate is not active, the input does not flow through. All connections between components are uni-directional, with the exception of the gray bus with arrow heads on both ends. A shaded arrow head indicates the directional flow of the circuit.

Figure 4.3 shows the basic layout of the datapath. The thick band around the top, right, and bottom of the datapath is the bus. The arrows at each end indicate that the bus is bidirectional. The bus serves as primary path of communication between components. As an example of the circuit width, take note of the control lines to the register file in the upper right. The width of 3 allows the control to specify 2^3 (8) values, referencing general purpose registers 0 through 7. The input from the bus to the register file is 16 bits wide, allowing the data or memory address

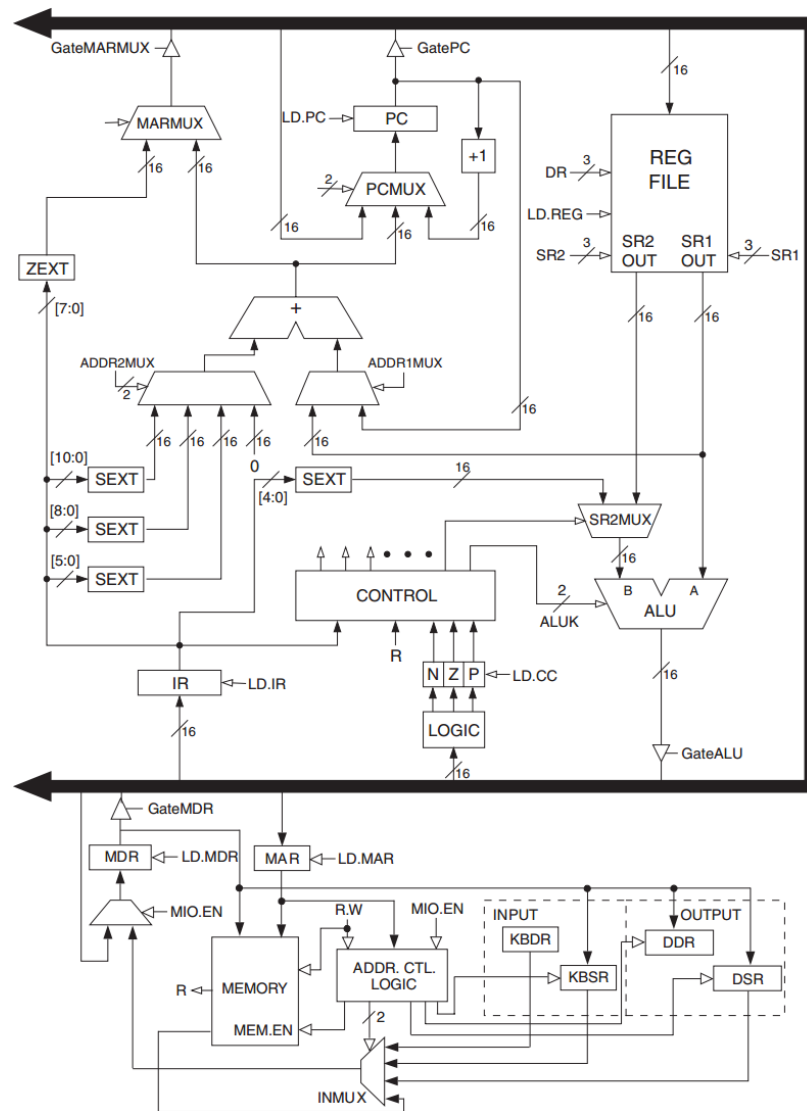


Figure 4.3: LC-3 Basic Datapath

to be stored within a single clock cycle. Similarly, the outputs of the register file are 16 bits wide. In the lower left of the datapath, the memory array is accessed with the assistance of the memory address register (MAR) and the memory data register (MDR). These specially purposed registers allow loading values onto the bus. The

input and output section, in the lower right, allows interaction with the console and the keyboard.

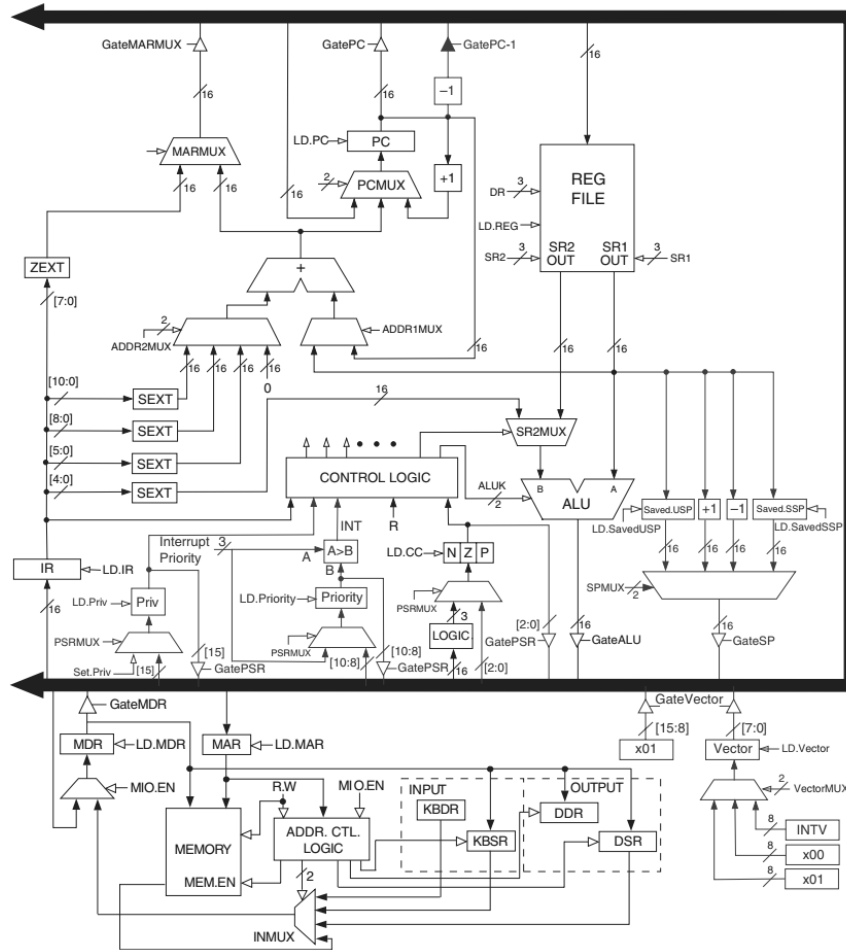


Figure 4.4: LC-3 Datapath with Interrupts

In a second form, the datapath is expanded further, as shown in Figure 4.4. The expanded datapath adds the ability to handle interrupts, throw exceptions, and enter privileged user mode. The bottom-right contains additional registers, control signals, gates, and a multiplexer to denote an interrupt from hardware external to

the CPU. The portion immediately below the control unit contains additional logic for monitoring the privilege levels and presence of interrupts. The additional logic enables the LC-3 to throw exceptions when violations occur. To the right of the ALU are additional static values and a multiplexer used to manipulate the user stack and supervisor stack when exceptions and interrupts are detected. This second form adds complexity, but allows the ISA to function as a more realistic CPU.

4.2.2 Control

The control structure for the LC-3 sets control signal values depending on the current state of the execution. The signals to be set during a clock cycle are determined by the finite state machine, laid out in Figure 4.5. Bubbles in the directed graph represent the conceptual work done on the datapath, with each transition representing a single clock cycle. Numbers at the top right of each bubble are unique identifiers, useful as an index when portraying the information in other formats. Text within the bubbles describes the datapath-level operations that takes place: moving values between registers, setting conditional flags, and in some cases basic logical and arithmetic operations. In most cases the exiting arrow shows a condition that must be met for the transition to the subsequent state during the next iteration of the clock.

The diagram does not fully describe how the logical flow of the Finite State Machine relates to the components on the datapath. Each state requires a set of asso-

ciated control signals needed to execute the logical operation. For example, state 18 needs to accomplish two tasks. The first task moves the Program Counter (PC) value to the Memory Address Register (MAR). In order to accomplish this, the GatePC must be asserted, allowing the PC value to flow to the bus. The LD.MAR must be asserted in order to allow the value on the bus to overwrite the contents of the MAR. The second task updates the PC with an incremented value of the PC. For this task, the PCMUX must accept a control signal, selecting the incremented input. Additionally the LD.PC signal is asserted, loading the output of the PCMUX to the PC. At the end of the clock cycle, the control evaluates whether or not an interrupt has been raised. Depending on the result of this evaluation of the interrupt status, the state will change to 49 or 33. Additional signals are used when including more complexity, such as the interrupt and exception handling.

Another method used to express the control structure is a simple spreadsheet; each column represents a state with the various control signals as rows. This tabular representation of the state offers a more literal translation of the physical architecture. As an example, the first few states from Figure 4.5 are transcribed in Table 4.1. This table is heavily truncated with only a few states and a few control signals represented. In state 18, the first pseudo-instruction “MAR \leftarrow PC” requires asserting GatePC to allow the value to be written to the bus and GateMAR to propagate the bus value to the MAR. The second pseudo-instruction “PC \leftarrow PC+1” requires asserting LD.PC

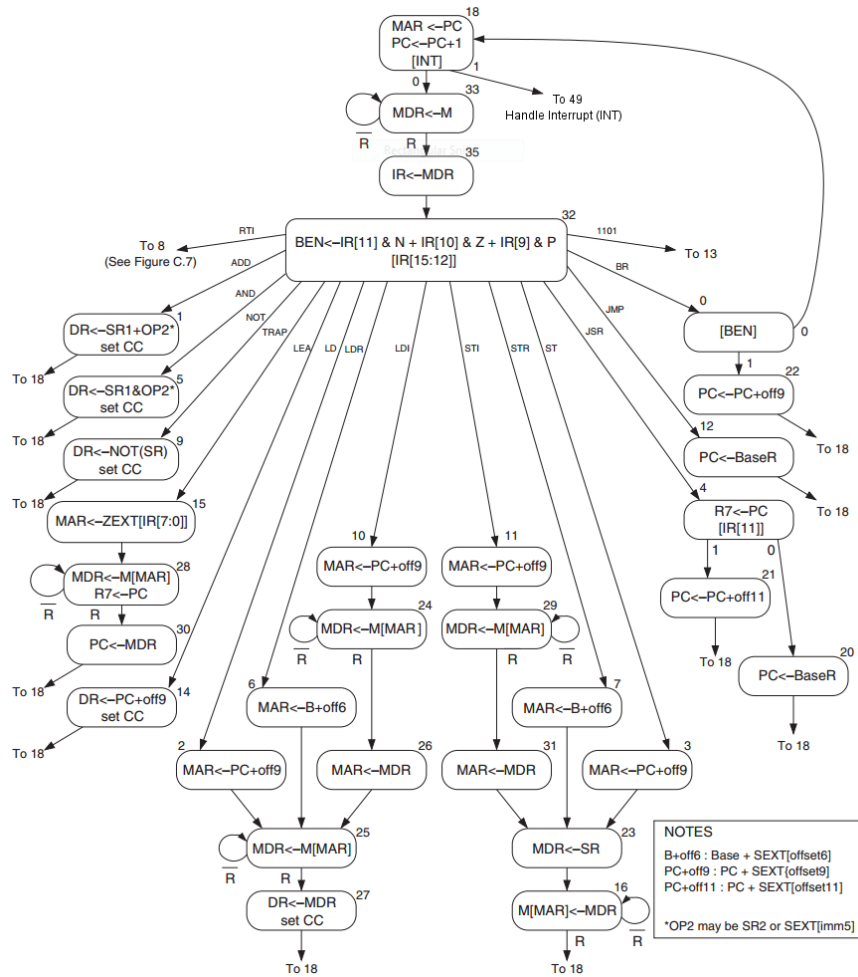


Figure 4.5: LC-3 Finite State Machine

and setting two bits for the PCMUX selection. The value “x” in the table indicates that the value is irrelevant and will not effect the execution.

4.3 Existing Projects

At the onset of creating the research and implementation, two LC-3 related projects were found. The first was an assembler. This tool allows the user to write

Table 4.1: Tabular View of Control Structure

Control Signal	States				
	010010(18)	100001(33)	100011(35)	100000(32)	000001(1)
LD.MAR	1	0	0	0	0
LD.MDR	0	1	0	0	0
LD.IR	0	0	1	0	0
LD.BEN	0	0	0	1	0
LD.REG	0	0	0	0	1
LD.CC	0	0	0	0	1
LD.PC	1	0	0	0	0
GatePC	1	0	0	0	0
GateMDR	0	0	1	0	0
GateALU	0	0	0	0	1
GateMARMUX	0	0	0	0	0
PCMUX1	0	x	x	x	x
PCMUX0	0	x	x	x	x
ALUK1	x	x	x	x	0
ALUK0	x	x	x	x	0

LC-3 assembly code and to compile it into a binary format, ready for execution on any simulator implementing the Instruction Set Architecture. A command line based simulator also exists, allowing the user to execute assembly code or otherwise encoded LC-3 instructions. These tools are useful for learning the assembly level of the LC-3 architecture and basic debugging skills; however, these projects do not contain any information or visualization of the datapath. Inclusion of datapath visualization would impart understanding of the physical implementation of the ISA.

Chapter 5 - Design and Implementation

The primary motivation for this research is to supplement Appalachian State University's sophomore level course "Introduction to Computer Systems" with an effective microarchitecture simulator. For several years, the course has been taught using the textbook *Introduction to Computing System: From Bits & Gates to C & Beyond* (2nd edition) [11]. The text uses the hypothetical architecture called LC-3, explained in the previous chapter. The existing tools assemble and simulate execution of LC-3 machine code, but no simulation addresses the microarchitectural aspects. Based on an examination of other simulators, design elements and visualization techniques were selected to provide the most user-friendly and effective tool possible.

5.1 Designing the Simulator

A review of other simulators and related texts allowed the determination of useful features as well as features that provide little benefit. Portability and extensibility problems, found to hinder adoption and usefulness, were also identified and avoided.

The research on multiple taxonomies allowed the design of a simulator that adhered to principles and models that have proven effective. The research guided implementation, adoption of the tool, and usability by students. A carefully designed simulator should be modifiable and extendable for future research and use.

5.1.1 Portability and Language Choice

As a general topic, one primary motivation for simulation is portability. It stands to reason that all other factors being equal, a portable simulator is more desirable; the application should not be tied to a particular system or platform. Many applications turn to a browser-based interface, though this suffers the distinct disadvantage of needing network access or a complex configuration to run locally.

For the purposes of portability, the Java programming language was selected. Java programs can be run on nearly all modern platforms. For the graphical elements, the Java Swing toolkit provides common graphical elements such as windows, panels, tabs, and buttons. The Swing elements can be further customized using the AWT toolkit, if the graphical needs are beyond those provided. Where possible, XML was used to configure parts of the simulator. The XML language is easily interpreted by most programming languages and environments. Ideally, these configurations could be used for completely separate implementations of the simulator or supplemental tools, without needing to use the source code for the original simulator. While no

plan exists for such additional tools, leaving these configurations open and accessible may provide quicker development of other projects in the future.

5.2 Relating to Taxonomy and Categories

Research on simulator categories and taxonomies provided insight on which features are desirable, how the data should be modeled, how the data should be visualized, and how to interact with the controls. This knowledge offers a critical lens for balancing the number of features against ease of use, realism, cohesion of elements, and potential for future changes.

Of the categories set forth by Wolffe, Rucik, Osborne, and Holliday, lc3uarch falls into the “intermediate instruction set” category [24]. The simulator adheres to the majority of the ISA set forth by Patt and Patel. Part of the implementation is the inclusion of memory modes, which are considered an advanced feature; however, lc3uarch does not include other portions of the ISA, such as exception and interrupt handling, which prevent it from being a completely accurate implementation of the LC-3 ISA. The code for lc3uarch is intended to be left open to future expansion and addition of new features, so this categorization may change in the future.

The taxonomy set forth by Yehezkel, as a more specific revision of Roman and Cox’s work, covers the considerations in designing a graphical architecture simulator [25, 15]. The goals for lc3uarch are briefly discussed here in terms of the taxonomy

elements. Further details are covered in the subsequent sections, independent of the taxonomy.

“Scope of operation” determines the environment in which the simulator is meant to be run. For lc3uarch, the aim is to have the simulator available in as many operating systems as possible. The user should be able to run the simulator on their personal computer or on a lab system. The selection of the Java programming language and the Swing toolkit make achieving this goal possible. The simulator has successfully run on OS X, Linux, and Windows with a single executable.

“Content modeling” is concerned with how literal the internal data-model is, when compared with the architecture specification. For pedagogical purposes, it is not necessary for the content model to be complete; higher levels of realism and complexity can serve to confuse students. Patt and Patel’s textbook demonstrates this by exposing two forms of the microarchitecture. To reduce the number of components to be displayed, lc3uarch does not include interrupt and exception handling. This obviates the need for priority levels, processor status changes, and privilege modes. An additional benefit is the reduction in the number of states in the finite state machine. An argument can be made for inclusion of the relevant components in the content model but not in the presentation; however, lc3uarch does not currently take this approach.

“Presentation methods” establish the visual arrangement of components selected in the content model, though not necessarily all elements. The presentation

oriented goal of lc3uarch is to provide as much information as possible on a single screen. Some of the reviewed simulators require toggling between views to access the full gamut of information. This can detract from seeing the overall state and may reduce comprehension of how the components are interrelated; however, too much information in a single view can cause “information overload” and may draw attention away from the currently relevant details. Since the simplified content model was chosen, the presentation will include all elements.

The single screen contains simulation controls, control registers, the general purpose register, state machine information, the memory array, console output, and the datapath. To reduce distraction caused by the information-dense display, the memory array will automatically scroll to the currently relevant memory address. The datapath display shows a color-coded animation, with red highlighting on active control signals and the flow of the circuit in blue. The color-coding shows what components are relevant during a particular clock cycle, with all other components remaining gray.

“Activity style” defines how users should interact with the simulation. The target style for lc3uarch is observational learning by animating the flow of circuits and components within the datapath. Unlike many microarchitecture simulators, there is no provision for changing the datapath for free-form exploration. The user can modify the contents of registers by clicking the informational display and using the resulting dialog. This does not have an intended pedagogical application, but it

may allow curious users to modify the execution or debug a program without having to re-assemble the code. To control execution of the simulation, the user can advance through a single clock cycle, advance to the next instruction fetch, or repeat the last clock cycle.

5.3 Design Goals

The design of lc3uarch is aimed at being as easy to use as possible without sacrificing accuracy of the datapath. This goal is not easily quantifiable and can only be evaluated through user feedback. To work towards this goal, as few controls as possible are presented to the user. Non-essential configuration options are not presented on the main screen. Reducing the interactions should reduce the complexity of use.

Concrete goals were also chosen. These goals can be evaluated as a success or failure while implementing the design. Even though achieving the goals can be immediately evaluated, measuring the effectiveness still requires user feedback. Aside from accurate execution of instructions, at clock cycle granularity, a concrete goal was to animate the flow of the datapath. This feature was seen in only one reviewed simulator and was found to add value in student comprehension. To ensure that the animations were correct and not misleading, the animation corresponding to each state in the finite state machine was evaluated individually.

One of the distinguishing features of lc3uarch is the ability to repeat the most recent clock cycle. No other simulator was found that supported this feature. Without this feature, repeating a clock cycle would require resetting the simulator, reloading the program, advancing to the previous state, and advancing a clock cycle. Being able to repeat the clock cycle allows the user to review the steps of the previous clock cycle, such as the assertion of control signals. The addition of this feature does depart from the realism of processor execution, but it adds another technique to support student learning.

5.4 Graphical Implementation

It is not possible to demonstrate the full feature set of any software through only the written word; however, screenshots and descriptions aid the reader with understanding the basic visual design. Where necessary, details of the interactions and animations are given. For context, Figure 5.1 shows the full simulator window. As indicated earlier, the chosen presentation method for the simulator is a single screen. Further details are in the subsequent sections.

5.4.1 Animated Datapath

Since lc3uarch is a datapath simulator, the majority of the window is dedicated to the datapath. Figure 5.2 shows an example of the datapath during animation. The screenshot was taken in the midst of fetching an instruction. As shown in red, the

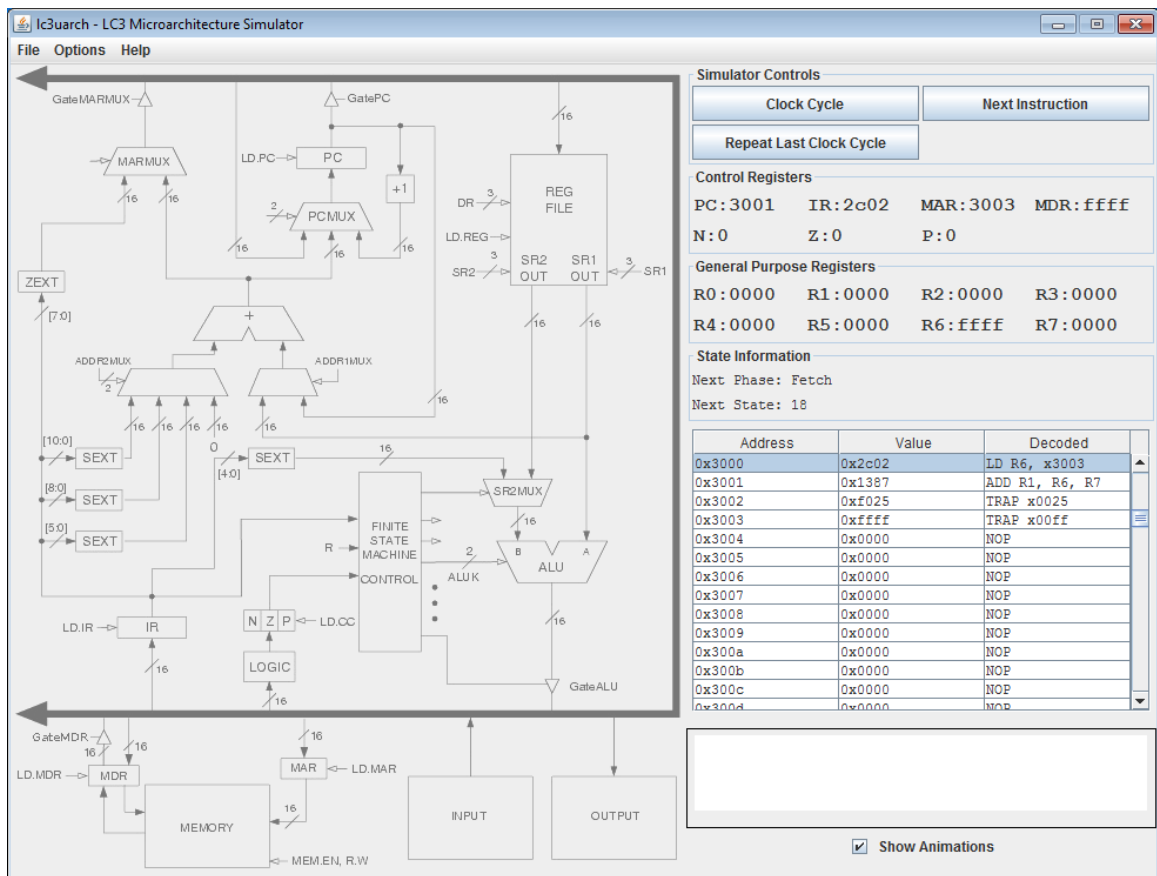


Figure 5.1: Full Window of lc3uarch

GatePC, LD.PC, LD.MAR, and the input selection for the PCMUX are asserted. The blue represents the flow of the circuitry, with the value of the PC propagating to the main bus.

Two menu items allow modification of this display. As Figure 5.3 shows, the user can speed up or slow down the speed of animation by use of a menu item. Figure 5.4 shows how the opacity of the main diagram can be changed. By making the background lighter, the animations are more pronounced, though the non-animated

elements may be more difficult to read. Other than these two configurations, the datapath is non-modifiable.

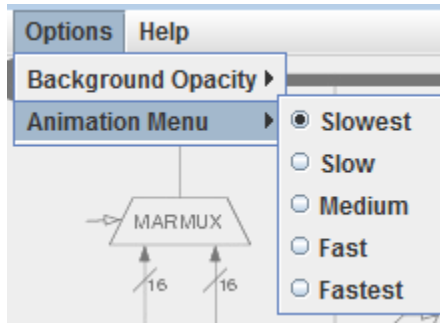


Figure 5.3: Animation Speed Menu

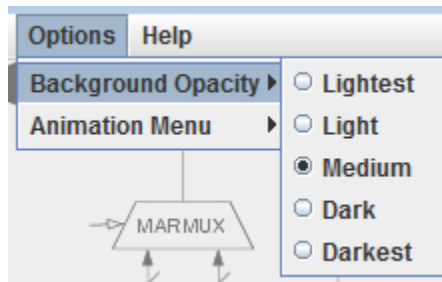


Figure 5.4: Opacity Menu

5.4.2 Controls and Components

The right side of the screen is dedicated to components and controls. The three controls are shown in Figure 5.5: Clock Cycle, Next Instruction, and Repeat Last Clock Cycle. The controls act as the names suggest. The Clock Cycle button advances the simulation by a single clock cycle. If hit before the current animation is complete, the animation resets and begins the next clock cycle immediately. The assumption is

that the user is not interested in watching the current animation complete. When the Repeat Last Clock cycle button is clicked, the previous register values are loaded and the clock cycle is restarted. When the user clicks Next Instruction, the simulation advances to the next instruction. In this case, the animation continues through all clock cycles needed to execute the current instruction without the user having to repeatedly click the Clock Cycle button. This will span a variable number of clock cycles, depending on the current phase of the fetch-execute cycle and how many clock cycles are needed to complete the particular instruction.

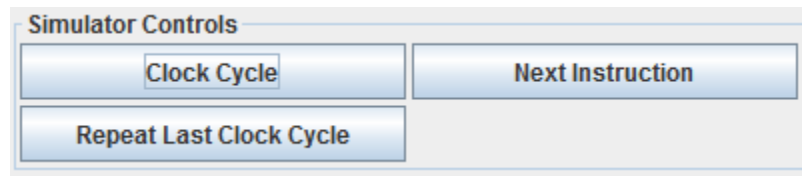


Figure 5.5: Controls

The registers shown in Figure 5.6 are divided into the specialized registers and the General Purpose Registers. The values are updated in tandem with the animation, only changing when the flow of the circuit arrives at the register. When the label for the register or its value is clicked, the dialog in Figure 5.7 is displayed, allowing the user to alter the value. This does not have a specific educational purpose, but it was included to allow experimentation by the user.

The memory array is displayed as a scrollable table, listing the address, the value at the address, and the assembly equivalent of the value. As Figure 5.8 shows,

Control Registers			
PC: 3000	IR: 0000	MAR: 0000	MDR: 0000
N: 0	Z: 0	P: 0	
General Purpose Registers			
R0: 0000	R1: 0000	R2: 0000	R3: 0000
R4: 0000	R5: 0000	R6: 0000	R7: 0000

Figure 5.6: Registers

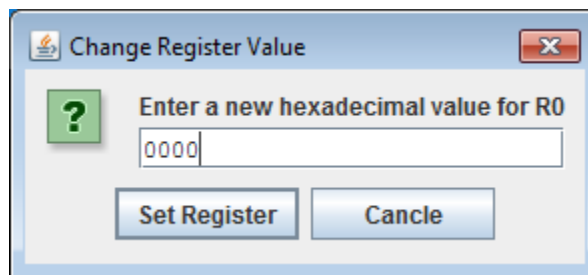


Figure 5.7: Changing Register Value

the decoded value does not differentiate between instructions and data. In the figure, the first instruction loads the value at address 0x3003 to register R6. The value 0xffff is data, not a TRAP instruction. It was desired to prevent this decoding of data, but no method was found that didn't require extensive work. To prevent the user from needing to scroll through the table, the relevant row is highlighted and scrolled to when a new value is loaded into the IR.

The contents of memory can only be altered by loading a new program via the Open menu item or through execution of the program. When opening a program or resetting the simulator through the menu items in Figure 5.9, the memory is populated with the trap vector table, interrupt vector table, and device registers.

Address	Value	Decoded
0x3000	0x2c02	LD R6, x3003
0x3001	0x1387	ADD R1, R6, R7
0x3002	0xf025	TRAP x0025
0x3003	0xffff	TRAP x00ff
0x3004	0x0000	NOP
0x3005	0x0000	NOP
0x3006	0x0000	NOP
0x3007	0x0000	NOP
0x3008	0x0000	NOP
0x3009	0x0000	NOP
0x300a	0x0000	NOP
0x300b	0x0000	NOP
0x300c	0x0000	NOP
0x300d	0x0000	NOP

Figure 5.8: Memory

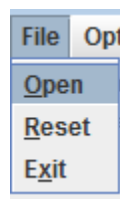


Figure 5.9: File Menu

The state information shown in Figure 5.10 shows the status of the finite state machine. The Next Phase demonstrates the steps in the fetch-execute cycle. The Next State correlates to the state machine provided by Patt and Patel. These values can be indexed against a print-out or the text book to emphasize how the control unit's logic relates to the datapath components

The final two portions of the right panel are the console output in Figure 5.11 and the Show Animations toggle in Figure 5.12. The console output is only used for specific TRAP instructions to print characters and strings. The Show Animations

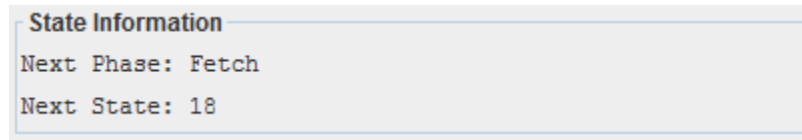


Figure 5.10: Finite State Machine Information

checkbox allows the user to bypass the animation and simply advance the clock cycles. Disabling animation can speed up the execution of instructions that are already understood. By default, animations are enabled since the primary purpose of lc3uarch is to animate the flow of the datapath.

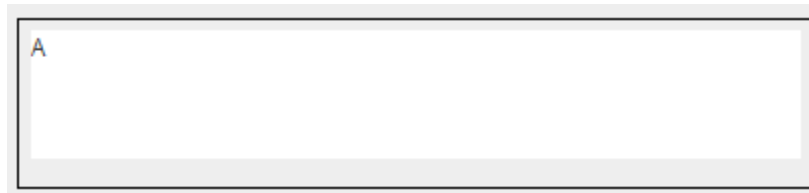


Figure 5.11: Console Output

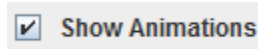


Figure 5.12: Toggle Animations

Chapter 6 - Evaluation

A common complaint in the reviewed literature is the difficulty in quantitatively evaluating simulator effectiveness. A primary factor in this difficulty is that most universities lack the student population to teach two simultaneous sections of a computer organization course. This limits the ability to use one section as a control group, comparing one class that uses the simulator to another class that does not. Even if two sections were available, natural variability of enrolled students and the teaching style of instructors can influence comprehension without the introduction of a simulator.

By extension, smaller class sizes are preferable for most departments, so splitting a single section in half for comparative analysis is not reasonable. These factors render quantitative evaluation statistically challenging. Searching for standardized, generic methods for qualitative evaluation of simulators did not provide any insight. Yehezkel eloquently summarizes the lack of standardized evaluation methods and a need for case-by-case evaluation.

The diversity of available simulators due to different pedagogical goals hinders the design of uniform evaluation methods for large populations or comparative research. The effectiveness of each simulator has to be evaluated in its own context according to well-defined individualized criteria. Qualitative research tools may yet answer these specific field research questions in the future [26].

Following Yehezkel's cue, student comprehension was evaluated via student surveys. Students were allowed to use the simulator for approximately 30 minutes in a lab setting, before being presented with an evaluation survey.

6.1 Survey Content

The survey consisted of two parts. The first part listed four statements to be ranked on a scale of Strongly Agree, Agree, Neutral, Disagree, or Strongly Disagree. Following these statements, three open-ended questions allowed for more general feedback.

The statements aimed at evaluating the overall usefulness of the simulator, the clarity of datapath visualization, ease of control, and use of colors in the visualization.

The statements were worded as follows:

- Overall, this simulator would be helpful in understanding the LC-3 microarchitecture.
- I can see the activities that occur during the fetch/execute cycle.
- Controls for the simulator were easily understood.

- The architectural portion of the simulator displayed colors/graphics effectively.

The open-ended questions sought more detailed information about elements that were noticeably distracting and what could be improved. Concentrating on the shortcomings, rather than the strengths, was chosen to get more critical feedback. The final question was open to any additional feedback.

- What parts of the simulator, if any, are distracting and/or useless?
- What improvement, if any, would help make the simulator better?
- Do you have any additional comments or suggestions?

6.2 Result of Surveys

The survey was administered to an Introduction of Computer System class containing 20 students. The results of the survey were positive, overall. Assuming a score of 5 for Strongly Agree down to a score of 1 for Strongly Disagree, the overall helpfulness of the simulator was averaged at 4.35. As Table 6.1 shows, the net score for each question was around the level of “Agree.” The relevance of these scores could be improved by surveying an additional section and providing a larger set of questions.

Examined another way, the same numeric scoring system is used to determine a score from each student, averaging the four questions. As the histogram in Figure 6.1 shows, three particular students had a less favorable opinion of the simulator.

Table 6.1: Totals of Survey Responses

	Str. Agree	Agree	Neutral	Disagree	Str. Disagree	Avg.
Overall	10	7	3	0	0	4.35
Activities	9	10	1	0	0	4.4
Controls	4	12	3	1	0	3.95
Colors	6	10	2	2	0	4

These surveys were noted, so particular attention could be paid to their feedback.

Addressing the concerns of the most critical students will likely benefit all students.

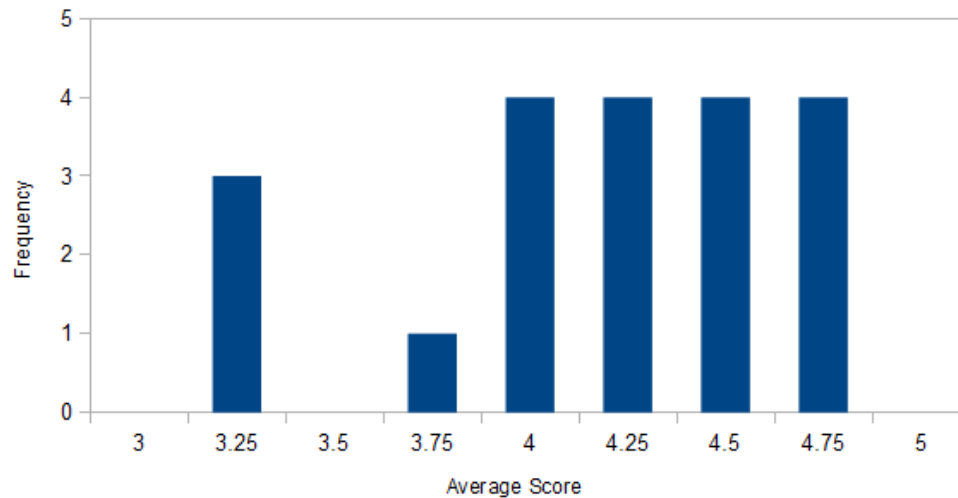


Figure 6.1: Histogram of Responses

From the open ended feedback, two trends were found. First, multiple students complained that the animation proceeded too quickly. Some cited the animation as too “jumpy,” particularly when getting to the bus. Among the three students who evaluated lc3uarch with the lowest score, the jumpiness and animation speed

were primary complaints. Another repeated response was problems interpreting the memory display. Students found it difficult to keep track of which memory location corresponded to the current instruction. This address can often be deduced by the value of the program counter, but the visualization of the memory array did not make this immediately apparent.

Another piece of feedback suggested that the original assembly code be displayed while the simulation is running. One student wished for a way to select a particular line in memory as the next instruction for execution. Another student explicitly mentioned that the tool would not provide much benefit for self-learning, but would be useful for an instructor demonstrating the concepts to a full class.

The concerns about animation speed and jumpiness have already been addressed in the `lc3uarch` source code. The slowest speed is now about half the previous rate. In addition, the jumpiness with regard to the bus has been repaired. Previously, the entire bus would highlight during the span of a single frame. The animation now flows from the point of origin towards both ends of the bus. Small changes to the memory array have also been made. The highlighting of the pertinent row has been made more obvious by changing the color. Additionally, periodically erratic behavior in the memory arrays scrolling was found and corrected. Due to the anonymity of those surveyed, it was not possible to re-evaluate with the same group of students.

Chapter 7 - Conclusions and Future Work

7.1 Conclusions

The graphical simulator `lc3uarch` addresses the lack of simulation tools for teaching microarchitectural concepts in the current structure of Appalachian State University's Introduction to Computer Systems course. According to student feedback, the simulator is helpful for learning parts of the course material. Several contributions were made by the research, project implementation, and writing of this thesis.

An overview was provided for the use of simulation in a pedagogical setting. First, research was presented on how simulators are beneficial in computer architecture courses when compared to hardware, primarily due to fiscal and accessibility constraints. Additionally, an overview was provided on the types of courses and styles of teaching that benefit from the use of simulation. The categories presented by Wolffe et al. serve as a starting point for selecting the appropriate simulator for a specific course and assist in narrowing the search criteria of available simulators [24]. If an appropriate simulator cannot be found, a new simulator may be designed and

implemented, as was done with lc3uarch. For the design process, Roman and Cox's taxonomy of program visualizations serves as a bottom-up framework [15]. This thesis also presents Yehezkel's taxonomy of architecture visualization, as an architecture-focused version of Roman and Cox's framework [25].

A survey of existing simulators was also provided by this thesis. While it was only possible to cover a subset of available simulators, strengths and weaknesses of each were highlighted. Understanding these strengths and weakness may guide the design and implementation of new introductory, digital logic, and microarchitecture simulators. These simulators targeted several courses with students of differing education levels. One underlying theme of the strengths found was the benefit of portability and accessibility, particularly through language choice.

An overview of the LC-3 ISA and microarchitecture was described in Chapter 4. This synopsis of Patt and Patel's architecture established the basis for designing a microarchitecture simulator and may also serve as a brief introduction for instructors interested in adopting the textbook for a course. The material also laid a foundation for the implementation of lc3uarch.

Guided by the taxonomies and best practices found in other simulators, lc3uarch was designed and implemented. An intentionally limited scope was chosen, focusing on the operation of the datapath and the interaction of microarchitectural components. The choice of Java and Swing was deliberate based on the portability issue

found in some of the simulators reviewed. A unique approach of a “flowing” datapath visualization was implemented to better convey the movement of data.

The implementation of lc3uarch was evaluated in the classroom through student surveys. Analysis of the feedback reinforced a common theme of the researched literature; quantitative analysis on the effectiveness of simulators is difficult, if not impossible. In spite of this difficulty, the comments provided anecdotal evidence that the tool was beneficial in comprehending the operation of the data path for instruction execution. The responses pointed to shortcomings of lc3uarch that have already been addressed. The feedback also provides ideas for future work as covered in the following section.

In addition to the direct contributions of this thesis, lc3uarch served as the topic of an academic article in SIGCSE [5]. The article discusses the instructor’s use of lc3uarch in the classroom and covers several of the topics in this thesis. The wide distribution of SIGCSE may alert other instructors of lc3uarch’s existence, providing another simulator to supplement courses using Patt and Patel’s textbook.

7.2 Future Work

As with any project, the number of features is limited by the time allowed to complete the implementation. Some additional features could be added fairly easily, provided ample time would be allowed to implement them. Other features could expand the pedagogical scope to emphasize different parts of the LC-3 architecture.

7.2.1 Graphical and Usability Changes

For the graphical display, several changes could be made relatively easily that may enhance comprehension of the animation. The first change would be to highlight the relevant registers in the display panel. This would work in tandem with the asserted control signals to show which registers have changed or will change. The color being used to highlight the datapath and registers should also be configurable. It has not been expressed in the surveys, but those with visual impairments such as color-blindness may have issues differentiating the colors. Additionally, the color scheme may be difficult to see on some displays. This also has not been reported, but it is a conceivable issue. The scroll bar for the memory array can be imprecise when trying to get to a particular address. The ability to jump to a particular memory address, instead of scrolling to it could make this operation easier.

One of the features unique to lc3uarch is the ability to repeat the last clock cycle. With some additional work, it could be possible to step back farther than a single clock cycle. This would require a moderate amount of additional code, but it could be significantly helpful to students who would like to review all the steps for a particular instruction's execution, not just the most recent clock cycle.

Not specific to the simulation, but as a general application, a few changes would improve lc3uarch. Migrating the look and feel of the particular platform would make the simulator appear to be a native application. The tool currently uses the Java

look and feel, which does not detract from the functionality, but it has an unpolished quality to the application. It would also be advantageous for the application to remember recently opened files or paths. Currently, every time the user opens a compiled assembly program, the file dialog starts in the home directory, not the directory of the most recently opened file.

7.2.2 Pedagogical Enhancements

Some changes could aid in the pedagogical value of lc3uarch. As Yehezkel suggests, pairing a simulator with educational materials improves the value [25]. One difficulty with this change is that different instructors might prefer a different set of augmented materials; however, it would be a simple task to provide some small example programs and the wherewithal to quickly load them into the simulator for demonstrative purposes.

Based on student feedback, it would also be helpful to integrate lc3uarch with the assembler. If done carefully, this could allow the students to write assembly code, assemble it, and have it immediately loaded into the simulator. This would negate the need for multiple tools. Additionally, the memory display could link to the original assembly instruction, allowing a more cohesive link between the code written and the instructions executed. A non-obvious advantage would be that labels in the assembly code could be preserved and displayed in the memory array. Using these labels, it

would be possible to prevent the data portion of memory from showing the decoded instruction, since the value is not intended to be executed.

7.2.3 Extended Scope

Aside from these finitely scoped changes, the simulator could expand in several directions. These changes would require significant amounts of work and alter or extend the purpose of the simulator. It would be necessary to design these alterations carefully to prevent “feature creep,” resulting in an unusable tool.

A common theme among several of the reviewed simulators is a configurable datapath. Several tools of a more circuit-oriented nature did this by allowing the user to change how individual components are connected. The software design of lc3uarch would need to be overhauled to provide this type of functionality. Providing this functionality would allow students a more experimental approach to understanding the concepts and could provide instructors an opportunity to guide the classes in a different direction, based on their teaching style. It is important to note that following this avenue would require the simulator to diverge from the LC-3 datapath described by Patt and Patel’s textbook. Ultimately, it could still implement the Instruction Set Architecture, but the details of the datapath would change and may serve as a point of confusion for some students.

Several of the reviewed simulators had this as a primary target and their approaches could be used for guidance on implementation. MicroTiger approached

this goal by introducing a custom microinstruction language [22]. Components could be re-arranged as would be done with a circuit editor and their functionality could be altered through a custom C-like microinstruction language.

RTLsim approaches this goal by allowing students to change the control signals issued during each clock cycle [26]. This approach avoids the microinstruction complexity introduced by Microtiger, but breaks from lc3uarch's current goal of observational learning. Without the microinstructions, change in the user interaction model would be necessary, requiring the student to set the proper control signals with each state in the control unit.

The JLS circuit editor approaches this problem by providing a Finite State Machine editor [12]. Providing an editor prevents the RTLsim's requirement of setting control signals with each clock cycle. Since the Finite State Machine provides the same functionality as microinstructions, it may be feasible to present both as alternate abstractions of the same control structure.

References

- [1] E. Z. Bem and L. Petelczyc. Minimips: a simulation project for the computer architecture laboratory. *SIGCSE Bull.*, 35(1):64–68, Jan. 2003.
- [2] P. Borunda, C. Brewer, and C. Erten. Gspim: graphical visualization tool for mips assembly programming and simulation. *SIGCSE Bull.*, 38(1):244–248, Mar. 2006.
- [3] P. Bose. Use of architectural simulation tools in education. In *Proceedings of the 1995 workshop on Computer architecture education*, WCAE '95, New York, NY, USA, 1995. ACM.
- [4] G. Braught and D. Reed. The knob & switch computer: A computer architecture simulator for introductory computer science. *J. Educ. Resour. Comput.*, 1(4):31–45, Dec. 2001.
- [5] A. Brownfield and C. Norris. Lc3uarch: a graphical simulator of the lc-3 microarchitecture. *SIGCSE Bull.*, 41(1):413–417, Mar. 2009.
- [6] C. Burch. Logisim: a graphical system for logic circuit design and simulation. *J. Educ. Resour. Comput.*, 2(1):5–16, Mar. 2002.
- [7] L. B. Cassel, M. Holliday, D. Kumar, J. Impagliazzo, K. Bolding, M. Pearson, J. Davies, G. S. Wolffe, and W. Yurcik. Distributed expertise for teaching computer organization & architecture. *SIGCSE Bull.*, 33(2):111–126, June 2001.
- [8] D. L. Knox. Integrating design and simulation into a computer architecture course. *SIGCSE Bull.*, 29(3):42–44, June 1997.
- [9] D. Magagnosc. Simulation in computer organization: a goals based study. *SIGCSE Bull.*, 26(1):178–182, Mar. 1994.
- [10] L. Null and J. Lobur. Mariesim: The marie computer simulator. *J. Educ. Resour. Comput.*, 3(2), June 2003.
- [11] Y. N. Patt and S. J. Patel. *Introduction to Computing Systems - From Bits & Gates to C & Beyond (2nd edition)*. McGraw Hill, 2004.

- [12] D. A. Poplawski. A pedagogically targeted logic design and simulation tool. In *Proceedings of the 2007 workshop on Computer architecture education*, WCAE '07, pages 1–7, New York, NY, USA, 2007. ACM.
- [13] Z. Radivojevic, M. Cvetanovic, and J. Dordevic. Design of the simulator for teaching computer architecture and organization. *Engineering of Computer Based Systems, IEEE Eastern European Conference on the*, 0:124–130, 2011.
- [14] U. Ramachandran and W. D. Leahy, Jr. An integrated approach to teaching computer systems architecture. In *Proceedings of the 2007 workshop on Computer architecture education*, WCAE '07, pages 38–43, New York, NY, USA, 2007. ACM.
- [15] G.-C. Roman and K. C. Cox. A taxonomy of program visualization systems. *Computer*, 26(12):11–24, 1993.
- [16] R. M. Salter and J. L. Donaldson. Abstraction and extensibility in digital logic simulation software. *SIGCSE Bull.*, 41(1):418–422, Mar. 2009.
- [17] J. E. Sayers and D. E. Martin. A hypothetical computer to simulate micro-programming and conventional machine language. *SIGMICRO Newsl.*, 19-20(4-1):4–10, Mar. 1989.
- [18] N. Skillen, V. Manickam, and A. Aravind. Ease: an extensible architecture simulation engine. In *Proceedings of the 16th Western Canadian Conference on Computing Education*, WCCCE '11, pages 23–27, New York, NY, USA, 2011. ACM.
- [19] D. Skrien. CPU Sim 3.1: A tool for simulating computer architectures for computer organization classes. *J. Educ. Resour. Comput.*, 1(4):46–59, Dec. 2001.
- [20] D. Skrien and J. Hosack. A multilevel simulator at the register transfer level for use in an introductory machine organization class. *SIGCSE Bull.*, 23(1):347–351, Mar. 1991.
- [21] P. Stenström and F. Dahlgren. A holistic approach to computer system design education based on system simulation techniques. In *Proceedings of the 1998 workshop on Computer architecture education*, WCAE '98, New York, NY, USA, 1998. ACM.
- [22] B. G. VanBuren and M. Shaaban. Microtiger: a graphical microcode simulator with a reconfigurable datapath. *SIGCSE Bull.*, 39(1):283–287, Mar. 2007.
- [23] J. S. Warford and R. Okelberry. Pep8cpu: a programmable simulator for a central processing unit. *SIGCSE Bull.*, 39(1):288–292, Mar. 2007.

- [24] G. S. Wolffe, W. Yurcik, H. Osborne, and M. A. Holliday. Teaching computer organization/architecture with limited resources using simulators. *SIGCSE Bull.*, 34(1):176–180, Feb. 2002.
- [25] C. Yehezkel. A taxonomy of computer architecture visualizations. *SIGCSE Bull.*, 34(3):101–105, June 2002.
- [26] C. Yehezkel, W. Yurcik, M. Pearson, and D. Armstrong. Three simulator tools for teaching computer architecture: Little man computer, and RTLSim. *J. Educ. Resour. Comput.*, 1(4):60–80, Dec. 2001.
- [27] W. Yurcik and L. Brumbaugh. A web-based little man computer simulator. *SIGCSE Bull.*, 33(1):204–208, Feb. 2001.

Vita

Andrew Brownfield was raised in Wilmington, NC, since the age of 3. After graduating from John T. Hoggard High School in 2001, he entered Appalachian State University. In 2005, he graduated with a Bachelor of Science in Computer Science. He entered the Computer Science graduate program at Appalachian State University in 2006. After completing two years of coursework, he left Boone to start his career with back-end web development in Raleigh. Unsatisfied with this field of work, he went back to his passion for the command line as an enterprise technical support engineer at Red Hat. During his final semester, before receiving his Master of Science degree in December 2013, he started a new job as an enterprise storage support engineer at NetApp.