Graphical microcode simulator with a reconfigurable datapath

Brian VanBuren

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Graphical Microcode Simulator with a Reconfigurable Datapath

by

Brian G VanBuren

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Associate Professor Dr. Muhammad Shaaban
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2006

Approved By:

Dr. Muhammad Shaaban
Associate Professor
Primary Adviser

Dr. Roy Czernikowski
Professor, Department of Computer Engineering

Dr. Roy Melton
Visiting Assistant Professor, Department of Computer Engineering
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Graphical Microcode Simulator with a Reconfigurable Datapath

I, Brian G VanBuren, hereby grant permission to the Wallace Memorial Library reproduce my thesis in whole or part.

________________________________________________________________________
Brian G VanBuren

________________________________________________________________________
Date
Dedication

To my son.
Acknowledgments

I would like to thank Dr. Shaaban for all his input and desire to have an update microcode simulator. I would like to thank Dr. Czernikowski for his support and methodical approach to everything. I would like to thank Dr. Semeraro for discussing and helping resolve his concerns with the proposal, simulator and other thesis work. I would also like to thank Dr. Roy Melton for being able to quickly read and provide feedback on my thesis.

I would like to thank Mark Seidman for numerous proof reads and helping me with aspects of the user interface to the simulator. I would also like to thank Matt Leese for his input about displaying the multiplexer and demultiplexer control lines and critiquing other user interface aspects.
Abstract

Microcode is a symbolic way to simplify control design that allows changing, testing and updating the control unit of processors. By changing the microcode, the same datapath can be used for an entirely different application, such as supporting a completely different instruction set. For these reasons, a majority of control units in modern day processors are microcoded. The object was to investigate and implement a graphical microcode simulator with a reconfigurable datapath and microcode format. By allowing a wide configuration of the datapath, many types of logical processors can be designed and simulated. The resulting implemented simulator is able to fill the void in microprogramming tools since there are no graphical microcode simulators that allow such customization of the datapath. The customization of the datapath goes beyond allowing different files specifying the datapath, it allows the datapath to be created and modified using the graphical interface. This tool is able to be used to design and simulate general-purpose processors and application specific processors through datapath and microcode configurations. In the academic setting, this tool provides easier microcode testing through verification on the instruction level for instructors and provide simulation debugging through code tracing and breakpoints for students.
Contents

Dedication ..................................................................... iii
Acknowledgments ................................................................. iv
Abstract ........................................................................... v
Glossary .......................................................................... xiv

1 Introduction ................................................................. 1
  1.1 Objective ................................................................. 1
  1.2 Motivation ............................................................... 1
  1.3 Performed Work ......................................................... 2
  1.4 Summary of Chapters .................................................. 3

2 Microcode Overview ..................................................... 5
  2.1 Instruction Set Architecture ......................................... 6
    2.1.1 Exceptions .......................................................... 9
  2.2 Datapath ................................................................. 10
    2.2.1 Datapath Elements ............................................... 10
    2.2.2 Pipelining .......................................................... 13
  2.3 Control Unit .............................................................. 14
    2.3.1 Control Fields ......................................................... 14
    2.3.2 Microprogramming ............................................... 15
    2.3.3 Microinstructions .................................................. 17
    2.3.4 Microprogram Sequencing ...................................... 18
    2.3.5 Horizontal Microcode and Vertical Microcode ................. 19
    2.3.6 Different Types of Microprogram Implementation ............. 20
  2.4 Conclusion ............................................................... 20
### 3 Microprogramming Tools

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Command Line Microcode Tools</td>
<td>21</td>
</tr>
<tr>
<td>3.1.1 Micro8 Microcode Assembler</td>
<td>21</td>
</tr>
<tr>
<td>3.1.2 AMISS</td>
<td>22</td>
</tr>
<tr>
<td>3.1.3 STEP Development Tools</td>
<td>23</td>
</tr>
<tr>
<td>3.1.4 Structured Microcode Development and Simulation System</td>
<td>25</td>
</tr>
<tr>
<td>3.1.5 MX Simulator</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Graphical User Interface Microcode Tools</td>
<td>26</td>
</tr>
<tr>
<td>3.2.1 μCS</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2 CPU Sim</td>
<td>27</td>
</tr>
<tr>
<td>3.2.3 Little Man Microcode Simulator</td>
<td>27</td>
</tr>
<tr>
<td>3.2.4 MythSim</td>
<td>28</td>
</tr>
<tr>
<td>3.2.5 Motivation</td>
<td>29</td>
</tr>
<tr>
<td>3.3 Implemented Microcode Simulator</td>
<td>30</td>
</tr>
<tr>
<td>3.3.1 Implementation Choices</td>
<td>32</td>
</tr>
<tr>
<td>3.4 Conclusion</td>
<td>33</td>
</tr>
</tbody>
</table>

### 4 Microprogram Assembler

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Microprogramming Language</td>
<td>34</td>
</tr>
<tr>
<td>4.1.1 Complex Microinstruction Format</td>
<td>35</td>
</tr>
<tr>
<td>4.1.2 Control Fields</td>
<td>36</td>
</tr>
<tr>
<td>4.1.3 Simple Microinstruction Format</td>
<td>39</td>
</tr>
<tr>
<td>4.2 Microprogram Parsing</td>
<td>40</td>
</tr>
<tr>
<td>4.3 Microprogram Verification</td>
<td>41</td>
</tr>
<tr>
<td>4.4 Configuration of the Control</td>
<td>42</td>
</tr>
<tr>
<td>4.5 Implementation Details</td>
<td>44</td>
</tr>
<tr>
<td>4.6 Conclusion</td>
<td>46</td>
</tr>
</tbody>
</table>

### 5 Datapath Depiction and Configuration

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Visual Depiction of a Datapath</td>
<td>47</td>
</tr>
<tr>
<td>5.1.1 Graph Theory</td>
<td>51</td>
</tr>
<tr>
<td>5.2 Reconfiguration of Datapath Components</td>
<td>52</td>
</tr>
<tr>
<td>5.2.1 Placing and Moving Wires</td>
<td>52</td>
</tr>
<tr>
<td>5.2.2 Signal Name Configuration</td>
<td>54</td>
</tr>
<tr>
<td>5.2.3 Component Configuration</td>
<td>55</td>
</tr>
<tr>
<td>5.3 Datapath Editor</td>
<td>59</td>
</tr>
</tbody>
</table>
5.4 Student Version’s Datapath Editor ........................................ 60
5.5 Grammars Used .............................................................. 61
  5.5.1 Comments .............................................................. 61
  5.5.2 Datapath Grammar ................................................. 61
  5.5.3 Memory Grammar .................................................. 65
  5.5.4 Expression Grammar ............................................... 65
5.6 Implementation Details .................................................... 67
5.7 Conclusion ..................................................................... 71

6 Simulation ......................................................................... 72
  6.1 Signals ...................................................................... 72
  6.2 Datapath Components .................................................. 73
    6.2.1 Ordering Of Datapath Components ............................ 74
    6.2.2 Expression Language Evaluation ............................... 76
  6.3 Control ..................................................................... 77
    6.3.1 Microinstructions ................................................... 78
    6.3.2 Next Microprogram Counter .................................... 78
    6.3.3 Calculating CPI ..................................................... 79
  6.4 Execution .................................................................. 79
    6.4.1 Backwards Execution .............................................. 80
    6.4.2 Displaying Simulation Results ................................. 80
    6.4.3 Microprogram Highlighting and Breakpoints ............. 81
    6.4.4 Simulation Animation ............................................. 82
  6.5 Implementation Details .................................................. 82
    6.5.1 Testing and Validation ............................................. 84
  6.6 Implemented Simulator Comparison and Results ............... 85
    6.6.1 Implementation Size Comparison with Other Microprogramming Tools .................................................. 85
    6.6.2 Performance Comparison of Various Datapaths .......... 86
    6.6.3 Performance Comparison with MythSim .................... 88
    6.6.4 Performance Comparison with AMISS ...................... 89
  6.7 Conclusion .................................................................. 90

7 Conclusions ...................................................................... 91
  7.1 Salient Features of the Implemented Simulator ................... 91
  7.2 Contributions ................................................................ 92
7.3 Areas for Future Work .................................................. 92
7.4 Closing Remarks ......................................................... 94

Bibliography ........................................................................ 95

A Subtract And Branch On Negative Example ....................... 98
   A.1 Datapath ................................................................. 99
       A.1.1 Control Fields ..................................................... 100
   A.2 Microprogram To Implement ISA ................................. 102
   A.3 ISA Code To Implement Multiply ............................... 104
   A.4 Microprogram To Implement Multiply .......................... 107
   A.5 Datapath File ......................................................... 109

B Various Datapaths ............................................................ 111
   B.1 Tiny Datapath .......................................................... 111
   B.2 LMMS Datapath ........................................................ 112
   B.3 Mic-1 Datapath ....................................................... 113
   B.4 Primer Datapath ....................................................... 114
   B.5 MythSim with One Write-Back Datapath ...................... 116
   B.6 MythSim Datapath .................................................... 117
   B.7 AMISS Datapath ........................................................ 118

C User Documentation .......................................................... 119
   C.1 Overview ............................................................... 119
   C.2 Creating Datapaths .................................................... 119
   C.3 Creating MicroPrograms .......................................... 121
       C.3.1 Control Fields Specification ................................. 122
       C.3.2 Datapath Macros ............................................... 122
   C.4 Simulating It All ..................................................... 122
   C.5 ISA Language .......................................................... 123
       C.5.1 Comments ......................................................... 123
       C.5.2 Simple Numbers ............................................... 123
       C.5.3 Intel Hex Format ............................................... 124
       C.5.4 SRecord Hex Format ......................................... 124
   C.6 Microprogram Language ............................................ 125
       C.6.1 Comments ......................................................... 125
C.6.2 Microinstruction Format . . . . . . . . . . . . . . . . . . . . . . . 125
C.6.3 Simple Microinstruction Format . . . . . . . . . . . . . . . . . . 126
C.6.4 Macro Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
C.7 Expression Language . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
  C.7.1 Comments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
  C.7.2 Defining Constants . . . . . . . . . . . . . . . . . . . . . . . . . 127
  C.7.3 Signal Names . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
  C.7.4 Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
  C.7.5 Programming Language Statements . . . . . . . . . . . . . . . . 131
C.8 About . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133

D CD Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Sample Screenshot of the Implemented Simulator</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>A Simple Model of Computers and Applications</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Memory Snapshot For Sample SBN Instruction</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>SBN Datapath with Datapath Elements Named</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Pipelining Example</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Microprogrammed Control Unit and Datapath Overview</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Hierarchy of Instruction and Microprogram Code</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Sample AMISS Output for a Single Cycle</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Little Man Microcode Simulator Interface [3]</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>MythSim Interface [26]</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Another Sample Screenshot of the Implemented Simulator</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Datapath Macros Dialog</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Control Configuration Dialog</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Microprogram Editor</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>Collaboration Diagram For The Control Implementation</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Adder, ALU, Demux and Mux Components</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Constant, RAM, Register File, and Signal Viewer Components</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Register and Pipeline Register Components</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>SBN Datapath with Datapath Elements Named</td>
<td>50</td>
</tr>
<tr>
<td>5.5</td>
<td>A Datapath With All The Vertices Numbered</td>
<td>52</td>
</tr>
<tr>
<td>5.6</td>
<td>Wire Drawing Mode State Diagram</td>
<td>53</td>
</tr>
<tr>
<td>5.7</td>
<td>Signal List Window</td>
<td>54</td>
</tr>
<tr>
<td>5.8</td>
<td>Custom Component Configuration Dialog</td>
<td>56</td>
</tr>
<tr>
<td>5.9</td>
<td>Register Component Configuration Dialog</td>
<td>57</td>
</tr>
<tr>
<td>5.10</td>
<td>Memory Component Configuration Dialog</td>
<td>58</td>
</tr>
<tr>
<td>5.11</td>
<td>Screenshot With The SBN Datapath Loaded</td>
<td>59</td>
</tr>
<tr>
<td>5.12</td>
<td>Student Version Main Application Window</td>
<td>60</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.13</td>
<td>Collaboration Diagram For The GUI Implementation</td>
<td>68</td>
</tr>
<tr>
<td>5.14</td>
<td>Collaboration Diagram For The Component Inheritance Usage</td>
<td>69</td>
</tr>
<tr>
<td>5.15</td>
<td>Collaboration Diagram For The Expression Inheritance Usage</td>
<td>70</td>
</tr>
<tr>
<td>6.1</td>
<td>Directed Acyclic Graph for SBN Datapath’s Non-clocked Components</td>
<td>75</td>
</tr>
<tr>
<td>6.2</td>
<td>Expression Tree Example</td>
<td>77</td>
</tr>
<tr>
<td>6.3</td>
<td>Collaboration Diagram For The Simulator Implementation</td>
<td>83</td>
</tr>
<tr>
<td>6.4</td>
<td>Datapath Properties Dialog For The SBN Datapath</td>
<td>86</td>
</tr>
<tr>
<td>A.1</td>
<td>Subtract And Branch On Negative Datapath</td>
<td>99</td>
</tr>
<tr>
<td>A.2</td>
<td>Microinstruction Format for Example SBN Datapath</td>
<td>100</td>
</tr>
<tr>
<td>B.1</td>
<td>Sample Datapath Screenshot: tiny.dp</td>
<td>111</td>
</tr>
<tr>
<td>B.2</td>
<td>Sample Datapath Screenshot: lmms.dp</td>
<td>112</td>
</tr>
<tr>
<td>B.3</td>
<td>Sample Datapath Screenshot: mic1.dp</td>
<td>113</td>
</tr>
<tr>
<td>B.4</td>
<td>Sample Datapath Screenshot: primer.dp</td>
<td>114</td>
</tr>
<tr>
<td>B.5</td>
<td>Sample Datapath Screenshot: mythsim_1write.dp</td>
<td>116</td>
</tr>
<tr>
<td>B.6</td>
<td>Sample Datapath Screenshot: mythsim.dp</td>
<td>117</td>
</tr>
<tr>
<td>B.7</td>
<td>Sample Datapath Screenshot: amiss.dp</td>
<td>118</td>
</tr>
<tr>
<td>C.1</td>
<td>ISA Code Example Using Intel Hex Format</td>
<td>124</td>
</tr>
<tr>
<td>C.2</td>
<td>ISA Code Example Using SRecord Format</td>
<td>124</td>
</tr>
</tbody>
</table>
# List of Tables

2.1 Control Fields Mnemonics for SBN Datapath’s zdest Control Field . . . . . 15

6.1 Implementation Line Count Comparison To Other Microprogramming Tools 85
6.2 Simulation Performance For Various Datapaths . . . . . . . . . . . . . . 87
6.3 Simulation Performance For Various MythSim Datapaths . . . . . . . . . . 88
6.4 Simulation Performance Comparison To AMISS . . . . . . . . . . . . . . 89

A.1 Control Fields and Their Mnemonics for SBN Datapath . . . . . . . . . . 101

C.1 Order of Operations With Higher Precedence At The Top . . . . . . . . . . 130
Glossary

A

**arithmetic logic unit (ALU)** a datapath element that performs specified arithmetic operations, p. 11.

B

**Backus Naur Form (BNF)** a formal notation to describe the syntax of a language, p. 40.

**branching** Modification of the program flow to use an instruction address to one that does not follow the current instruction when a specific condition is satisfied, p. 7.

C

**central processing unit (CPU)** see processor, p. 10.

**clock cycle** a measure of the length of time for one clock cycle for the processor, p. 10.

**computer organization** the overview of computer design, p. 3.

**cycles per instruction (CPI)** a measure of performance of a system that indicates the number of clock cycles per instruction for a given set of instructions, p. 14.
E

edge  used to connect vertices together indicating relationships between the vertices, p. 51.

exception  any unscheduled procedure call, also called interrupt, p. 9.

F

firmware  is the microprogram as it is implemented into a memory store, also known as microcode store, p. 20.

G

graph  a set of vertices connected to each other through edges to show the relationships between the data the vertices represent, p. 51.

graphical user interface (GUI)  a way for a software program to display information to the user using elements such as windows and dialogs, p. 1.

grid snap  a constraint technique that forces elements that users drag to snap to a grid [13], p. 52.

I

input and output (I/O)  the input and output to a system unit such as the processor, p. 6.

instruction set architecture (ISA)  the architecture as it is used for a software program, p. 5.

integrated development environment (IDE)  a development environment that provides editors and other tools like compiling, building, running and simulation, p. 81.
J

jumping  Modifying the program flow to use an instruction address to one that does not follow the current instruction, p. 7.

M

Manhattan routing  a constraint technique that forces all wires to be orthogonal, p. 52.

microinstuction  a single line of microcode that sets the control signals for a single cycle of execution, p. 18.

microprogram  a collection of microinstructions that include microinstruction branching logic, p. 20.

microprogram counter (MPC)  used to indicate the next microinstruction address in the microprogram to execute, p. 18.

microword  the actual value used by a microinstruction to store in firmware, p. 22.

minimal instruction set computer (MISC)  an ISA that uses only a few instructions and is RISC philosophy taken to an extreme, p. 99.

P

portability  a measure of how many different machine platforms a piece of software can run on, p. 22.

processor  the unit that contains a datapath and control unit to perform operations of the system, p. 6.

program counter (PC)  used to indicate the next instruction address to execute, p. 7.
random access memory (RAM)  a typical type of memory that allows non sequential access for reading and writing of data, p. 12.

read only memory (ROM)  a type of memory that allows reads but does not allow writes, p. 12.

rubberbanding  a technique to draw lines that displays the line being drawn as the user keeps the mouse button held down to specify the ending location of the line [8, page 382], p. 53.

subtract and branch if negative (SBN)  an instruction set architecture that has only one instruction, the subtract and branch if negative, described in detail in Appendix A, p. 7.

vertex  a piece of data represented within a graph, p. 51.
Chapter 1

Introduction

1.1 Objective

Microcode is beneficial since its simplicity can reduce the possible errors and the time to develop part of the processor. In the 1980s and 1990s there were several microcode simulators that were text based but could adapt to be used for many different processors. The recent trend in microcode simulation is to use a graphical user interface (GUI). However, there is no available microcode simulator that provides a graphical user interface and high adaptability to different processors. By adding a graphical user interface to a reconfigurable architecture simulator, modifications can be performed on the processor diagram rather than using a text editor.

1.2 Motivation

The main motivation for this microcode simulator was to update the microcode simulator used for Computer Organization classes. However, after researching existing microcode simulators, there was an obvious lack of graphical simulators that had modifiable processors. Another motivation was to create a graphical microcode simulator that could simulate a wide variety of processors to fill this void in microcode simulators. By using ideas from circuit design packages and software development environments, such a microcode simulator could meet these goals.
1.3 Performed Work

Figure 1.1: A Sample Screenshot of the Implemented Simulator

The implemented simulator completed the main goals and resulted in many features including the ones shown in the sample screenshot in Figure 2.1. The salient features of the implemented simulator include:

- Provides a datapath editor similar to circuit editor which is not included by other microcode tools.
- Provides many components including an adder, an ALU, a constant, a demultiplexer, a multiplexer, a pipeline register, a RAM, a register, and a register file.
- Provides a customizable component using a C based programming language with
several added constructs and a namespace to decouple the expression from the datapath
diagram.

- Microprogram format with symbolic names, macros and line labels, including a special isa line label to indicate when an ISA level instruction starts.

- Provides several simulation options including running for a specific iteration count (1, 10, 100, 1000, 1000000), run to a specific iteration, run to a specific microinstruction, and simulation animation.

- Provides easy access to simulation results including updating the datapath editor with signal values, iteration number, and microprogram counter, and updating the integrated microprogram editor by highlighting the current microinstruction.

- Provides the CPI for a simulation run.

- Provides an integrated number converter for different bases.

## 1.4 Summary of Chapters

Chapter 2 overviews the basics of computer organization and microprogramming. Chapter 3 overviews a variety of microcode assemblers and simulators and provides a list of features desired in a new microcode simulator. In Chapter 4, Chapter 5, and Chapter 6, various aspects of the implemented simulator are detailed. Chapter 4 overviews the control and microprogram portion of the developed microcode simulator. Chapter 5 overviews the graphical user interface of the developed microcode simulator. Chapter 6 overviews the details of the simulation engine for the developed microcode simulator. Chapter 7 summarizes the developed microcode simulator and possible future capabilities that could enhance the simulator.

Appendix A overviews the example processor, instruction level code and microprograms for the simple example used throughout the chapters. Appendix B overviews several
of the other datapaths implemented for the designed microcode simulator. Appendix C is the user documentation for using the simulator. Appendix D overviews the contents of the CD.
Chapter 2

Microcode Overview

The object of any computer system is to run applications. Applications are developed and compiled using specific instructions for a instruction set architecture (ISA). The instruction set architecture allows different computer systems that conform to the ISA specifications to run the same compiled applications.

Figure 2.1: A Simple Model of Computers and Applications

Figure 2.1 shows how the application compiled for the instruction set architecture interacts with the processor only using the instruction set architecture interface. The computer system provides the specified interface required to meet the instruction set architecture
The processor is the central processing unit of the system that performs operations of the system. As shown in Figure 2.1, the processor interacts with memory, input and output (I/O), and other subsystems. The processor is comprised of a datapath and control unit as shown in Figure 2.1. The control dictates the operations for the datapath to perform and uses the state of the system to determine the next set of controls. The datapath performs the bulk of the operations of the system. The memory component stores both the application instructions and the application data for all the applications running on the system. The I/O channels are used to connect the system to other devices such as disks, keyboards, mice, monitors and other processor systems. The culmination of the processor, memory and I/O are a simple computer system and provide the architecture for the ISA as shown in Figure 2.1.

The rest of this details different aspects of computer organization as they relate to microprogramming. In particular, instruction set architectures, datapaths, and control units will be overviewed.

### 2.1 Instruction Set Architecture

Every instruction set architecture has locations to read and store data. These locations can be accumulators, registers, or memory. Accumulators are registers used to read data for every instruction and store data for every instruction. Registers are used in groups and can be indexed to indicate which registers are being used for inputs and which registers are being used for outputs so that different registers can be used for different instructions. Memory is used in the same manner as registers. However, since memory provides a huge store of data, it has slower access time for loading and storing data compared to registers.

An ISA provides arithmetic and logical instructions that do certain operations on the data. Other instructions that are more specific include I/O instructions, loading and storing instructions, and instructions that work on specific data types like strings and arrays.
A majority of instructions are in a sequential order, meaning that one instruction follows another instruction and they are stored sequentially. The program counter (PC) points to the current instruction address. During a sequential instruction, the program counter is incremented by the length of the instruction so that the program counter points to the following instruction to indicate that it is the next instruction to execute.

However, there are additional instructions that modify the sequence of instructions by changing the program counter to point to an instruction other than the next sequential instruction. When an instruction always modifies the program counter it is called jumping. When an instruction modifies the program counter when a certain condition is satisfied it is called branching. The jump type instruction modifies the program counter to a specified instruction address. The branch type instruction jumps if a certain condition is satisfied, such as the result of an arithmetic operation being zero. Otherwise the branch type instruction uses the next sequential instruction. The call subroutine and return from subroutine instructions allow the program to jump to another sequence of instructions and return to the current sequence of instructions. This is accomplished by storing the next program counter value in some location (such as memory) when the call subroutine instruction is specified. Then when the return from subroutine instruction is specified, the program counter is restored from the same location to return to the sequence of instructions following the call subroutine instruction.

Each instruction usually consists of an opcode and operands. The opcode specifies the instruction and the operands specify how to perform the instruction. Since opcode values are difficult to remember, instruction mnemonics are assigned for each opcode. Sample instruction mnemonics are ADD, LOAD, and JUMP. Typical instruction operands are register indexes, memory addresses, branch addresses, and constant values.

An example instruction set architecture is specified in Appendix A. The subtract and branch if negative (SBN) ISA defines a single instruction, the subtract and branch if negative instruction. This ISA uses only memory and a single register for the program counter. However, the implementation of the ISA may use registers to store intermediate values.
within a instruction’s execution. These intermediate registers are not visible to the application since they are not specified by the ISA. The instruction performs a subtraction and stores the result in the minuend’s memory address. If the result of the subtraction is negative, it branches to a specified instruction address. Otherwise, the next sequential instruction is executed. While this architecture is relatively simple, it is fully capable of performing any computational task if given an infinite amount of memory.

The instruction mnemonic for the subtract and branch if negative instruction is \textit{sbn}. The operands follow the mnemonic as in the following example:

\begin{verbatim}
sbn negone, zero, loop
\end{verbatim}

The above instruction is part of the ISA code example that implements multiply in Section A.3. This instruction subtracts the value at memory location specified by \textit{zero} from the value at memory location specified by \textit{negone} and stores the results in the memory location specified by \textit{negone}. If the values of \textit{negone} and \textit{zero} are -1 and 0 as their names imply, \textit{negone} will be unaffected by subtracting \textit{zero}. The subtraction will be negative so that the program counter will be set to instruction address specified by \textit{loop}. If \textit{zero} remains zero and \textit{negone} remains -1, this instruction will always jump to the instruction address specified.

Figure 2.2 shows the memory address and value pairs from the ISA code in Section A.3. At memory location 0x0018 are the three memory addresses of the operands. The first address is specified as \textit{negone} which translates to 0x0024 which has a value of 0xFFFF. The second address is specified as \textit{zero} which translates to 0x0022 which has a value of 0x0000. The third address is specified as \textit{loop} which translates to 0x0006. So the instruction will subtract 0x0000 from 0xFFFF and store it at the memory location specified by \textit{negone}, which does not affect the memory location since it is storing the same value that is already stored there. Since the subtraction is negative, instruction address specified by \textit{loop} is branched to. This means that the program counter will be set to 0x0006 and the instruction execution will continue using the updated program counter. Using the information from Table 2.2, the instruction \textit{sbn negone, zero, loop} may also be written as:
Figure 2.2: Memory Snapshot For Sample SBN Instruction

```
sbn 0x0024, 0x0022, 0x0006
```

This means that it uses memory addresses 0x0024 and 0x0022 for the subtraction and 0x0006 for the branch address. It is just more convenient for a ISA programmer to use labels for memory locations since code modifications can easily move these locations to different locations.

### 2.1.1 Exceptions

In addition to the regular sequence of a program, an unscheduled procedure call may interrupt the program. These unscheduled procedure calls are called exceptions. An exception can happen for a variety reasons such as an external I/O device request, an application invoking the operating system, arithmetic overflows, using an undefined instruction, or a hardware malfunction [17]. Once an exception occurs, the program control is transferred to the exception handling procedure to handle the particular exception. Once the exception
is handled, program control is returned to the program that was interrupted.

For each exception, an exception procedure address is saved in an exception lookup table. This allows the control to know where to jump when an exception happens. To avoid affecting the interrupted program, the state of the program is saved so that the state can be restored at the end of the exception handler. This at least requires storing the program counter so that the program counter can be restored after the exception procedure is completed.

2.2 Datapath

Hennessy and Patterson define a datapath as “the component of the processor that performs arithmetic operations” [17]. The control unit dictates to the datapath what operations to perform according to the instructions in the program. Together, the datapath and control unit are called the processor, or central processing unit (CPU). The processor coordinates the system resources (such as I/O, memory and other processors) and provides the interface specified by the ISA.

At the logic design level, a clock cycle is the unit of time for the datapath to perform a set of the control’s commands. The clock cycle can be the time to execute one ISA instruction. However, many modern systems only do a portion of the instruction in a single clock cycle in an effort to improve the speed and performance of the processor.

Next, the datapath elements that comprise a datapath will be detailed. Afterwards, one of the most important datapath optimizations, pipelining, will be reviewed.

2.2.1 Datapath Elements

The datapath is composed of many different types of function performing elements. Many common datapath elements are adders, arithmetic logic units, comparators, demultiplexers, memories, multiplexers, and registers.

 Figure 2.3 shows the datapath for the SBN datapath with all the datapath elements
named. The datapath contains a demultiplexer (labeled as Demux), a few multiplexers (labeled as Mux), a memory, an ALU, a custom element, two constants, and several registers. The same datapath is presented in Appendix A in Figure A.1. However Figure A.1 is labeled with the control field names and a few comments.

Adders perform the single operation of adding operands together. Adders typically add two operands together and are useful for incrementing the program counter or determining the next instruction address for a relative branch.

Comparators are used to compare operands (such as equal to and less than). They are mainly used to help control signals to determine what operation to perform on one or more of the operands being compared.

An arithmetic logic unit (ALU) is a generic term for any datapath element that performs an arithmetic operation. Even though technically adders, comparators and other single functioned arithmetic datapath elements are ALUs, they are usually named after their single operation. Rather, a typical ALU performs a variety of arithmetic operations based on a control signal given to it. Some ALUs perform a few operations for instruction sequencing while others help perform all the essential arithmetic operations specified by
the instructions in the ISA.

Multiplexers take several inputs and output a selected input. Demultiplexers perform the opposite operation; demultiplexers assign a single input to only one of the outputs. These signal selection elements are vital to a datapath in modifying the signal flow from cycle to cycle.

Memory is a large storage of data. It provides instructions for the datapath to execute, the location to load inputs for the instructions, and the location to store results of the instructions. There are two majors type of memories, random access memory (RAM) and read only memory (ROM). RAM allows non-sequential accesses for reading and writing whereas ROM allows data only to be read and does not allow data to be written.

Memory performance has not improved as well as processor performance, and as a result a small memory is placed on many modern processors to improve the performance of memory. This memory is called cache and holds a temporary copy of a small portion of memory. Since cache is within the processor, cache is much faster to use.

Registers are used for holding temporary values of operations currently being performed. Since registers operate at the same speed as the processor’s clock cycles, they have the highest performance. However, there is only a small limited number of registers (usually between ten and one hundred). Sometimes groups of registers are called a register file. A register file can have any number of inputs and outputs and uses indexes to indicate which register in the register file to use for each input and output.

The two right most multiplexers (Muxes) in Figure 2.3 are used to determine the operands for the ALU. The ALU in Figure 2.3 is used only for subtraction of its two operands (which is also called a subtracter). The only demultiplexer (Demux) in Figure 2.3 is used to indicate where to store the results of the ALU. The left most multiplexer (Mux) in Figure 2.3 is used to determine which signal to use for the memory address. The memory in Figure 2.3 is used to store the instructions for ISA code and the data used by the ISA code. The registers in Figure 2.3 are used to store the program counter, the instruction operand addresses and values. The constants in Figure 2.3 are the constant values 0 and -1 (shown as
Subtraction of 0 assigns the ALU output to its first input which is used to assign one register value to another location. Subtraction of -1 is the same as adding 1 and is used for incrementing the program counter. The custom datapath element in Figure 2.3 is used to determine the next microprogram counter.

Next, pipelining is overviewed as to how these datapath elements can be utilized to work on different sets of data at the same to increase the throughput of the datapath.

2.2.2 Pipelining

Pipelining is a technique used to design processors such that the actual execution of several instructions is done in parallel to maximize the use of the components of the processor as much as possible. This is possible since a single instruction can be broken into several sequential stages. Then each sequential stage can be performed one cycle at a time allowing other instructions to be executing different stages at the same time. Therefore all the stages can pass their results into their next stage at the end of the cycle to allow the processor to perform the next stages on all the instructions being pipelined.

![Pipelining Example](image)

Figure 2.4: Pipelining Example

Figure 2.4 shows the fundamentals of pipelining. The figure uses a pipeline that breaks
the execution into five stages: Fetch, Decode, Load, Exec, and Store. The figure shows instructions in the rows and the instruction cycles in the columns. The first four cycles are devoted to filling the pipeline. Once the pipeline is filled (cycle 4 in the figure), every cycle has an instruction completing and four other instructions in progress. The cycles per instruction (CPI) of a pipelined processor is theoretically one since one instruction completes every cycle.

However, sometimes instructions cannot be parallelized completely such as when an instruction requires the previous instruction’s results and are called pipeline stalls. Some pipeline stalls can be shortened or avoided by using forwarding techniques. In some cases, compilers can reorder the instructions to try to avoid pipeline stalls (but still maintain the same execution results). A pipeline stall will increase the CPI for a given program since an instruction will not complete every cycle.

2.3 Control Unit

The job of the control unit is to set the control fields for the datapath to execute the program written for a given ISA.

Next, the fundamental output of the control unit, control fields, will be described. Then microprogramming, microinstructions, microprogramming sequence, different types of microprogram formats, and different types of control implementation will be detailed.

2.3.1 Control Fields

The control fields are the control fields needed by the datapath to determine the operation of all the datapath elements. Sometimes multiple datapath elements use the same control field. Sometimes a datapath element uses multiple control fields. In addition, there are control fields that are used by the control unit to determine its next state.

What makes a control field special is how it specifies a particular action or desired data and these can be better described using control field mnemonics rather than though control
field values.

**Control Field Mnemonics**

Each control field performs different operations within the datapath based on the control field’s value. These different operations can be given names that are called control field mnemonics. These mnemonics describe the operation the control field value specifies. For example, Table 2.1 shows the control field mnemonics and their respective control field values for the \( zdest \) control field in the SBN datapath. Using the table, the control field mnemonic \( ztopc \) would assign the value 3 to the control field \( zdest \). The datapath for the SBN (as shown in Figure 2.3 and Figure A.1) will use the control field value as a select for the demultiplexer to send the output of the ALU into the \( PC \) register. The entire listing of all the control fields and their respective mnemonics for the SBN datapath is in Table A.1 in Appendix A.

<table>
<thead>
<tr>
<th>Control Field Mnemonic</th>
<th>Control Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ztoaaddr</td>
<td>0</td>
</tr>
<tr>
<td>ztobaddr</td>
<td>1</td>
</tr>
<tr>
<td>ztomem</td>
<td>2</td>
</tr>
<tr>
<td>ztopc</td>
<td>3</td>
</tr>
<tr>
<td>ztoa</td>
<td>4</td>
</tr>
<tr>
<td>ztob</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2.1: Control Fields Mnemonics for SBN Datapath’s \( zdest \) Control Field

Microprogramming makes extensive use of control field values and control field mnemonics to provide the control field values during execution.

**2.3.2 Microprogramming**

Microprogramming is a form of low-level programming to represent the control lines for the datapath. A microprogram consists of microinstructions. Each microinstruction consists of control fields for the control signals to the datapath and control fields to determine
the next microinstruction. Microprogramming is beneficial since its simplicity can reduce the possible errors in the control unit and reduce the time to develop the control unit.

Figure 2.5: Microprogrammed Control Unit and Datapath Overview

Figure 2.5 shows the connectivity between the microprogrammed control unit basic components and the datapath. The control fields are represented by the arrows from the microprogram store to the datapath and are used to control the datapath’s operation. The control fields are also used along with the status of the datapath in the microprogram sequencing logic to determine the next microprogram counter. The microprogram counter is used to lookup the next microinstruction.

Within the processor view of the software, a microprogrammed processor has two levels of code being executed: the instruction level code and the microprogram level code. Each instruction is stored in memory, and its execution involves the execution of one or more microinstructions. Each microinstruction specifies to the datapath the small steps to perform that result in performing the ISA level instruction. Many simple instructions use only a few microinstructions. More complex instructions may use many microinstructions including loops of microinstructions. The more microinstructions an ISA instruction has associated with it, the longer the execution time of the ISA instruction is.

Figure 2.6 shows the correlation between the instruction level code and the microprogram level code for the sbn ISA. The figure uses the same “sbn negone, zero, loop”
instruction previously analyzed in Section 2.1. The figure uses the microprogram detailed in Section A.2 of Appendix A. The figure shows how the one instruction in the ISA corresponds to nine microinstructions (a majority of which are fetching the instruction and its operands). The microinstructions are executed sequentially by the processor. Since every ISA level instruction using this microprogram uses nine microinstructions, the resulting cycles per instruction (CPI) is 9. Each microinstruction has a set of control field values that it sends to the datapath to execute the steps necessary to fetch, decode and execute the ISA instruction.

### 2.3.3 Microinstructions

A microinstruction contains all the control fields for the datapath’s controls. Each different microinstruction controls the datapath to perform a different operation. Additionally, the microinstruction specifies how to determine the next microinstruction.

Typically, the ISA instructions are implemented as one or more microinstructions. Since many instructions perform similar tasks, such as reading data values from memory, the same microinstruction may be used by several instructions to reduce the microprogram length.

In addition to writing microinstructions to perform the ISA instruction’s operation,
there are other tasks performed by the microprogram. These include decoding an instruction’s opcode and performing exception handling. Both of these typically require advanced microprogram sequencing to perform the task optimally.

2.3.4 Microprogram Sequencing

The current microinstruction is addressed by the microprogram counter (MPC). Many microinstructions flow sequentially, and the MPC is incremented by 1 to indicate the following microinstruction. However, the power of microprogramming comes from its ability to provide more sophisticated sequencing techniques.

One such sequencing technique is to give the address for the next instruction to allow jumps in the microprogramming sequencing. This allows a microinstruction to jump back to the beginning of the ISA instruction decode to evaluate the next ISA instruction.

Another sequencing technique is to allow branches in the microinstruction. This can be done several ways. One way is to specify the branch address to use if the branch condition is met and if the branch condition is not met then continue with the next microinstruction. Another way is to specify both addresses for the branch taken and the branch not taken cases. And yet another way is to specify an address that is modified using the branch value (0 or 1) which results in two different addresses.

One of the more advanced microsequencing techniques involve specifying an address that is modified by a value from the datapath to provide many microinstruction addresses and is useful for decoding microinstructions.

Another technique is have a control unit that separates the microcode store into a control store, a sequence store and a condition store as done in the SMDSS tool explained in Section 3.1.4 [22]. This allows the sequencing to use advanced high-level type sequencing such as the conditional statement (if), multi-way statement (case), several types of looping (while, repeat, for), function calling, and jumping [22].
Just Another Microsequencer

Many of the advanced microsequencing techniques are implemented by Just Another Microsequencer (JAM) [24]. The microsequencer is a VLSI design that implements very flexible microprogram sequencing. It supports advanced microinstruction sequencing used in different commercial processors including N-Way branch, N-Way call, N-Way return, conditional branch, conditional call, conditional return, and loop control.

The microinstruction sequencer has a branch address input, a branch condition input, a branch mode input, and a program counter output. The sequencing type is specified by the branch mode input [24]. Also, the branch mode can be set to 6 to indicate the branch address is the N-Way mask. This mask is used to indicate which bits of the branch address to combine the current program counter register to calculate the next program counter. An internal stack (with 8 levels) provides the mechanism for storing and retrieving program addresses for calls, returns, and loops.

2.3.5 Horizontal Microcode and Vertical Microcode

In additional to microprogram sequencing, another software technique applied to microprogramming to try to reduce the microprogram size though vertical microcode.

Horizontal microcode uses many of the control fields within the microinstruction and requires little decoding [17]. As a result, horizontal microcode is very wide (or horizontal).

Vertical microcode uses few control fields within the microinstruction and requires more decoding [17]. As a result, vertical microcode is very narrow but can require more microinstructions and increases the microprogram length (make the microprogram more vertical). Vertical microcode typically has an opcode field that indicates the microinstruction structure. This can allow different fields to be specified based on the microinstruction opcode. This microinstruction opcode is very similar to an instruction opcode and how an instruction opcode indicates the format of the rest of the microinstruction. The main reasoning behind vertical microinstructions is that not every control field is used during every
Microprograms can be written initially using a horizontal microinstruction format and later converted to a vertical microinstruction format to reduce the overall number of bits used to store the microprogram. As a result, many microprogramming tools do not specify a vertical microinstruction format.

2.3.6 Different Types of Microprogram Implementation

Microprogramming is just a method to develop the control unit and can be implemented in several different ways. As seen previously in Figure 2.5, the microprogram can be stored in a customized memory store within the control unit called firmware, or microprogram store. Some microprocessors allow the microprogram to be reprogrammed to allow support of another ISA. The reprogrammable firmware allows instructions to be added, instructions to be optimized, and instructions to be fixed for previous errors.

Additionally, the microcode can be implemented as a hard-wired finite state machine [17]. This hard-wired finite state machine gains all the benefits of the microcode design process without the delays and physical area associated with a hardware microprogram sequencer and microstore.

2.4 Conclusion

Instruction set architectures are definitions of the architecture as seen by the software. Multiple processors can implement the ISA differently but provide the same standard interface specified by the ISA. The datapath consists of many datapath elements such as ALUs, memories, multiplexers, demultiplexers, and registers. The datapath performs different operations based on the control values specified by the control unit. Microprogramming is a technique to use programming techniques to specify the control values for the datapath to implement the instructions in the ISA. Since microprogramming is a form of software
programming, microprogramming tools can be used to aid in developing and testing microprograms. The following chapter overviews several existing microprogramming tools and introduces the microcode simulator implemented.
Chapter 3

Microprogramming Tools

Over the last three decades, various microcode assemblers and simulators have been developed to meet the changing needs of microprogram writers. The following assemblers and simulators are a small portion of the available microprogramming tools and represent many of the features that are possible.

First, the command line tools will be summarized in chronological order: Micro8 Microcode Assembler (1981), STEP Development Tools (1985), AMISS (1985), Structured Microcode Development and Simulation System (1990), and MX Simulator (1995). Then the graphical user interface tools will be summarized in chronological order: μCS (1992), CPU Sim (2001), Little Man Microcode Simulator (2002), and MythSim (2004). Finally, the implemented simulator will be overviewed.

3.1 Command Line Microcode Tools

3.1.1 Micro8 Microcode Assembler

The Micro8 Microcode Assembler [12] is a commercially-available tool that would allow microcode to be developed for various “real world” logic designs. The tool was developed to fill the demand for a microprogramming tool that had several criteria. The criteria are portability, flexibility in defining the microinstruction, adaptability to existing hardware designs, aid in debug during firmware and hardware integration, and minimal restrictions
on the naming conventions.

Since this is only a microcode assembler, only the control unit structure of the logic design needs to be defined instead of the entire logic design of the processor. Therefore only two input files are needed, a control unit structure and a microprogram.

The adaptability and flexibility of the microword definition is provided though the use of a very flexible control field definition. Each control field can be any size and be located anywhere in the microword to allow overlapping control fields for vertical microcode. Each control field defines mnemonics for the control field values that simplify writing a microprogram. Each control field can also specify an overflow control field used to assign any overflow to when a microinstruction assigns a value too large for a control field.

Micro8 provides extensive error checking. It provides checks for assignments that are too large (including any overflow control fields). It also checks if overlapping control fields specify conflicting default values or if a part of the microword is specified more than once within a single microinstruction. And Micro8 checks for the reuse of a microinstruction address.

3.1.2 AMISS

The AMISS tool [19] was developed through Robert J. Pesar’s Master’s Thesis, A Microprogramming Simulation System for Educational Use. It provides a flexible simulation environment that allows the datapath to be described behaviorally through a high-level language. The datapath definition is fairly complicated with different sections for different aspects of the processor definition. As a result, the datapath is difficult to modify for the simulator. Consequently, the datapath used at the Rochester Institute of Technology has not been modified in nine years.

In the AMISS tool, the microinstruction format is very simple as a list of all the values for the control fields. This simple format is difficult to learn since both the control field order and the control field values need to understood and memorized. The actual simulation is performed using an interactive command line interface that allows for simple tracing of
the microprogram.

<table>
<thead>
<tr>
<th>CYCLE 42</th>
<th>opc0</th>
<th>cond</th>
<th>addr</th>
<th>intr</th>
<th>test</th>
<th>load</th>
<th>cons</th>
<th>alud</th>
<th>xsou</th>
<th>ysou</th>
<th>aluc</th>
<th>shift</th>
<th>lcnt</th>
<th>mend</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUBCYCLE 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>pch</td>
</tr>
<tr>
<td>spl</td>
</tr>
<tr>
<td>xlo</td>
</tr>
<tr>
<td>mw</td>
</tr>
<tr>
<td>mbus</td>
</tr>
<tr>
<td>shift</td>
</tr>
<tr>
<td>mpc</td>
</tr>
<tr>
<td>vbit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUBCYCLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>pch</td>
</tr>
<tr>
<td>spl</td>
</tr>
<tr>
<td>xlo</td>
</tr>
<tr>
<td>mw</td>
</tr>
<tr>
<td>mbus</td>
</tr>
<tr>
<td>shift</td>
</tr>
<tr>
<td>mpc</td>
</tr>
<tr>
<td>vbit</td>
</tr>
</tbody>
</table>

Figure 3.1: Sample AMISS Output for a Single Cycle

Figure 3.1 shows the output of a sample cycle from the AMISS tool. However, usability was not a primary concern of the simulator. The microinstruction shown at the beginning of the output does not order the control field values in the same order as the microinstruction format. Additionally, the order of the registers does not seem to have any apparent order and makes it very difficult to determine a register’s value quickly.

### 3.1.3 STEP Development Tools

The STEP Development Tools [27] is a commercial microprogram development system. It provides high level language constructs for both the control structure definition and the microprogram definition. It is highly flexible in many ways that allow many vastly different architectures to use the tool.

The language for both the control structure and the microprogram is called the METASTEP
The language uses many common high order language-like constructs such as flow control, source inclusion and macros.

Each control field has many attributes that can be set. Some of these attributes include the position of the control field in the microinstruction, the length of the field, and a parity field for the microinstruction. The field can store signed numbers as either one’s complement or two’s complement and can reverse the bit order. Also, the field can have mnemonics specified and the legal values for the field specified. To facilitate vertical microcoding, the simulator allows fields to be defined based on the value of a specified field. This allows better vertical microcode verification since fields can be checked against another field’s particular value as to whether the field is valid for this microinstruction.

When writing microcode for STEP, the control field is specified and assigned either a numeric value or one of the mnemonics associated with the control field. It was done this way rather than have global mnemonics that set all the control fields using the mnemonic since it shows the microprogrammer’s intent and allows for better validation. The most impressive feature of the STEP Development Tools is the macro expression flexibility. In addition to macros accepting multiple arguments, one macro argument can appear before macro name to provide easier to read microprograms. The following examples show the $IFT$ macro and the $< -$ macro:

```
IFT aReg < bReg goto label;
aReg < - aReg - bReg;
```

These macros might expand to the following:

```
aluOp=sub, rand1=aReg, rand2=bReg, branchTest=negative, branchAddr=label;
```

```
aluOp=sub, rand1=aReg, rand2=bReg, aluDest=aReg;
```
This macro facility is one of the most impressive of any the microcode assemblers and allows for a high level language influenced constructs within the microprogram.

### 3.1.4 Structured Microcode Development and Simulation System

A Structured Microcode Development and Simulation System (SMDSS) [22] a limited re-configurable tool. The limitation is that the tool focuses on advanced microcode sequencing that uses a fixed method for microprogramming sequence.

The tool separates the microcode store into a control store, a sequence store and a condition store. The control store contains the control values for the datapath. The condition store and sequencing store are coordinated to determine the appropriate control store address to run next based on the given sequencing. Many of the high level sequencing constructs are available for use such as the conditional statement (if), the multi-way statement (case), several types of loop statements (while, repeat, for), function calling statements, and the jump statement.

The tool provides a simulator for testing the microprogram sequencing but does not simulate the datapath of the processor. This tool provides excellent support for the advanced microprogramming sequencing but is very limited in scope of simulation and forces the microprogram sequencing into the defined control structure.

### 3.1.5 MX Simulator

The MX Simulator [5] implements the Mic-1 microprogrammable architecture that Tanenbaum uses in his *Structured Computer Organization* book. The simulator is targeted for the Mic-1 architecture and an extended architecture that adds a carry flag and multi-way branching support. The microprogram compiler accepts the Pascal based language that Tanenbaum defines for his example microprogram. The compiler compiles to a low-level format that uses microinstruction address and control field numeric values that can also be used to input a microprogram.
The user interface defines three different interfaces. The full screen mode presents all the registers to the user with two lines for the user to input commands. An interactive command line mode uses commands for explicit displays and execution. Finally, a batch mode allows the compiler and simulator to be scripted to aid teachers in grading student’s microcode.

The Pascal-based microprogramming language allows operations to be specified to simplify the job of coding the microprogram and increases the readability of the microprogram. Since the operations and architecture are simple enough, it is relatively easy to understand how the Pascal-like operations are converted into the microinstruction control fields. Donaldson, the author of the simulator, says that an appropriate extension of the simulator would be to allow the architecture to be redefined.

### 3.2 Graphical User Interface Microcode Tools

#### 3.2.1 μCS

A microcode simulator for the Apple Macintosh called μCS (for microcode simulator) [10] is one of the earliest microcode simulators utilizing a graphical user interface (GUI). The microcode simulator targets a hypothetical machine architecture strongly based on the PDP-11 instruction set and influenced by useful attributes from the DEC LSI-11. This simulator was developed to be used for an undergraduate computer organization class to teach the PDP-11 instruction set.

One of the most unique features of the μCS simulator is the way it allows a microprogram to be written. The simulator uses a sequence of dialogs to allow the user to select the attributes of the microinstruction quickly. For "power users", the key presses are buffered so the input can go through a sequence of dialogs without too much slow down (supposedly a microinstruction with 6 control fields can be entered in about 4 seconds). This unique method is definitely faster than typing in text strings that a majority of microprogram tools use but can require additional specifications to be used for a flexible microinstruction definition.
3.2.2 CPU Sim

CPU Sim [21] is a processor simulator that involves both the instruction and the micro-
programming levels. Since the simulator targets Computer Science students, the simulator
focuses on the instruction set architecture. As a result the microprogramming aspect is only
oversimplified macros chosen through dialogs. It allows the hardware to be described on a
register transfer level and allows for a variety of processor types to be implemented. The
simulator hides a majority of the hardware and microprogramming details so that only
parts that affect the instruction set architecture are seen. CPU Sim is a unique simulator in
how it takes and presents the processor details to the user. Since the simulator is instruction
set architecture centered, it is a poor tool to use to teach microprogramming and computer
organization.

3.2.3 Little Man Microcode Simulator

The Little Man Microcode Simulator [3] is a simple microcode simulator designed to pro-
vide an introduction to computer organization concepts. It differs from the other micro-
code simulators in that the user cannot modify the actual microcode but only the instruc-
tions. Figure 3.2 shows the interface for the Little Man Microcode Simulator. Since the
microcode is fixed, the simulator only runs one instruction at a time and shows the register
trace for the microprogram. This is useful for introducing students to the ideas of micropro-
gramming while reinforcing the concepts of running a program at the assembly level.

3.2.4 MythSim

MythSim, or The Mythical Simulator for Real Students [26], is designed from the ground
up to be an easy to use simulator to teach students the fundamentals of microprogram-
ning. MythSim is cross platform and uses a very intuitive graphical user interface. The sim-
ulator uses a very simple datapath (consisting of an ALU, a register file, and memory). It has
many integrated features that aid in microprogram development and debugging. The microprogram viewer highlights the current microinstruction, the memory viewer highlights the current memory address and memory value, and the register file viewer highlights the register in use. These windows and the datapath window are shown in Figure 3.3. Each of these viewers use a different color to highlight. The same color as the highlight colors in the viewer windows are used on the datapath to correspond to the datapath elements. The simulator aspect allows various simulation controls such as simulating microinstruction cycles by 1, 10, and 100 in both directions. Overall, the simulator is very usable and simple which allows students to focus more on learning microprogramming rather than learning the tool.
3.2.5 Motivation

The initial motivation was to update the microcode simulator used for Computer Organization classes at the Rochester Institute of Technology. After researching existing microcode simulators, there was an obvious lack of graphical simulators that had modifiable architectures. The motivation was to create a graphical microcode simulator with a highly reconfigurable datapath. While CPU Sim does technically fit the requirements, its focus was on the instruction set architecture rather than the computer organization of the datapath and microprogram correlation. By using ideas from circuit design packages like PSpice and integrated development environments like Microsoft Visual Studio, a microcode simulator can be made easy to use and have a reconfigurable datapath.
3.3 Implemented Microcode Simulator

In the 1980s and 1990s, there were several microcode simulators that were text based with reconfigurable datapaths. The recent trend in microcode simulation is to use a graphical user interface (GUI). However, there is no available microcode simulator that uses both a visual interface with a reconfigurable datapath. After developing Micro, the author, David Magagnosc, wanted to develop another versatile graphical simulator by allowing modification to the underlying datapath [16]. Also, John L. Donaldson indicated the next logical step to improve his MX simulator tool would be to provide a means to modify the base architecture [5]. By combining a graphical user interface with a reconfigurable architecture simulator, modifications can be performed on the architecture diagram rather than using a hardware description language to generate the datapath. However, a hardware description language was still needed to load and save the configured datapaths even though it is not exposed to the user.

The implemented simulator completed the main goals and resulted in many features including the ones shown in the sample screenshot in Figure 3.4. The salient features of the implemented simulator include:

- Provides a datapath editor similar to circuit editor which is not included by other microcode tools.

- Provides many components including an adder, an ALU, a constant, a demultiplexer, a multiplexer, a pipeline register, a RAM, a register, and a register file.

- Provides a customizable component using a C based programming language with several added constructs and a namespace to decouple the expression from the datapath diagram.

- Microprogram format with symbolic names, macros and line labels, including a special isa line label to indicate when an ISA level instruction starts.
Figure 3.4: Another Sample Screenshot of the Implemented Simulator

- Provides several simulation options including running for a specific iteration count (1, 10, 100, 1000, 1000000), run to a specific iteration, run to a specific micro-instruction, and simulation animation.

- Provides easy access to simulation results including updating the datapath editor with signal values, iteration number, and microprogram counter, and updating the integrated microprogram editor by highlighting the current microinstruction.

- Provides the CPI for a simulation run.
• Provides an integrated number converter for different bases.

The interface for datapath modification resembles a circuit design environment that has standard datapath structures such as ALUs, multiplexers, demultiplexers, memories, buses, and registers to enable an efficient mechanism to datapath reconfiguration. The datapath editor also allows generic datapath modules that use a built-in expression evaluation to set the outputs based on internal signals and inputs. These generic datapath modules allow for greater configuration to the datapath.

3.3.1 Implementation Choices

The C++ language was chosen as the implementation language due to its high portability and proficient speed. Java was considered since it has higher platform independence, but disregarded since it lacks the necessary speed for the simulation environment.

The wxWidgets toolkit was chosen for its high portability [9]. The GUI is a datapath editor and an observer of the simulation environment. Since the simulator is written in C++ the GUI should also be written in C++. The three major portable graphical toolkits for C++ are QT, GTK and wxWidgets. The QT toolkit is a commercially developed toolkit with Windows, Unix/Linux and Macintosh support. However, to generate a Windows executable using the QT toolkit has to be licensed. GTK is a toolkit heavily developed for Linux and ported to Windows with poor support. The wxWidgets toolkit was developed to be platform independent and supports many platforms, including Windows and Linux. For developing open source platform independent GUIs, wxWidgets seems to be the best choice.

There are many inputs to the microcode simulator that require parsing. These parsed inputs include the microprogram, the datapath, the ISA code, and the expression language for the datapath elements. Lex and Yacc are common tools for input parsing in C and C++ and are used by the microprogramming tools AMISS [19] and SMDSS [22]. Flex is the freely available tool that is completely upwardly compatible with Lex and was utilized to provide the pattern matching for the implemented simulator [18]. Bison is the freely
available tool that is completely upwardly compatible with Yacc and was utilized to provide
the general parsing capability for the implemented simulator [6].

3.4 Conclusion

While there are many existing microcode simulators that target different aspects, there
is no tool that combines both a graphical user interface with a reconfigurable datapath.
There are some highly reconfigurable microprogramming tools. Also, there are varying
degrees of user interfaces associated with different microprogramming tools. By using
favorable aspects of several of the existing microprogramming tools, a microcode simulator
was implemented to bring together a simulator that uses both a graphical interface and a
reconfigurable datapath. In the next chapter, the microcode aspect of the implemented
microcode simulator is analyzed.
Chapter 4

Microprogram Assembler

The main use of the microcode simulator is to aid with microprogram development and testing. The microprogram language needs to be easy to read and easy to write to aid in both development and testing. The microprogramming interface needs to be easy to use and aid in the more complex aspects of the microprogram development and testing.

This chapter overviews the control and microprogram portion of the developed microcode simulator. First the microprogramming language will be detailed. Then the parsing and verification of the microprogramming language is overviewed. Next, the control configuration is detailed including how the control field mnemonics can be described to the simulator. Finally, the microprogramming implementation design is described at its high-level code design.

4.1 Microprogramming Language

Ideally, the microprogram could be written at a very high level without much knowledge of the underlying hardware. A high-level microprogramming language could then allow the same microprogram to be utilized for various related datapaths. For example the MX simulator uses goto and boolean OR operations within the microcode to allow for a multi-way branch [5]. The MythSim project uses if, else and goto keywords for microprogram sequencing [26]. However, it is easier for these simulators to specify these microprogram languages at a high level since the datapath and microprogram sequencing are fixed.
Trying to implement a high level microprogramming language that corresponds to a reconfigurable datapath increases both the software implementation and the microprogram design complexity. It increases the complexity of the microprogram design since the high level constructs need specifications in how to convert the constructs into the control field values for the microinstruction. However, a lower level microprogramming language is easier for one to understand what exactly is performed by the microinstruction.

In addition, the microprogram sequencing intent would need to be fully specified for a reconfigurable datapath. Allowing the sequencing to be fully specified would require a complex interface that would be very difficult to understand. This is why a majority of reconfigurable microcode simulators do not use high level constructs. The Micro8 microcode assembler provides an incredibly flexible control signal definition that allows a microinstruction to be labeled and that label to be assigned to control field values [12]. For example, this allows microprogram branching by setting a microinstruction address to a control field. Couple this with a reconfigurable datapath that controls the microprogram logic and the resulting microinstruction sequencing has the same potential power as the high level constructs.

As a result, the microprogramming language designed for the implemented simulator is a fairly low level microprogramming language that uses control field mnemonics, microinstruction labels and macros. These are the same type of constructs used within an assembly language.

The microprogramming language is a series of microinstructions. Each microinstruction represents the controls for a single cycle of the datapath. There are two formats of microinstruction formats: a complex one that uses symbols to avoid memorization of control field values and a simple format consisting of a simple list of control field values.

**4.1.1 Complex Microinstruction Format**

The Micro8 assembler [12] and the MythSim tool [26] both use a comma separated list of control fields labels and control field assignments with an optional line label. This format
is simple to write, is easy to read, and still has high flexibility. The control field mnemonics indicate a value for a control field. However, it would be tedious to make a mnemonic for every possible value. For example, assigning a mnemonic for every register file index would be pointless since the registers in a register file are usually referred to by their index anyway. Therefore the control field assignments are still very necessary and useful. The line labels allow the microprogram sequencing to specify a line of the microprogram and let the compiler determine the address of the specified microinstruction.

**Microprogram Comments**

As with any programming language, comments are necessary for designers to pass information to others about the details of the microprogram. The comments use both C++ style comments (“//”) and Perl style comments (“#”). Everything after the comment delimiters until the end of the line is part of the comment. It is useful to provide several standard commenting styles since they are recognizable as comments and assist new users using the microprogramming language.

**4.1.2 Control Fields**

There are many ways control signals are used. They can be used as enables, addresses, operation control logic, sequence control logic and more. Several of the microprogram tools reviewed in Chapter 3, including the Micro8 microcode assembler, allow every field to have user defined control field mnemonics for the control field values which can be referenced within the microinstruction [12]. These control field mnemonics can give descriptive names to the enables, operation controls and sequence controls. This useful feature was included in the implemented simulator to allow the user to specify the operations for a microinstruction without the need to remember specific control values.

Additionally, the Micro8 assembler provided a special control field mnemonic called “DEFAULT” to specify the default value of the control field if the microinstruction does not specify the value [12]. Finally, multiple mnemonics can indicate the same control field
value. This is useful when a control means different things depending on the context of other signals.

However, one useful constraint on the control field mnemonics is that no control field mnemonic can be used by more than one control field. The exception to this constraint is that any number of control fields may use the special “DEFAULT” control field mnemonic. This improves clarity of the microcode since a control field mnemonic means only one assignment and does not contain assignments hidden in other control fields. This differs from the Micro8 assembler since the Micro8 assembler allows the control field mnemonics to be used in multiple control fields to assign a value to each of those control fields using one mnemonic [12]. This also differs from the STEP development tools which forced the control field name to be used in conjunction with the mnemonic requiring longer keywords to set the control value [27]. This constraint was chosen to use ideas from both: to allow global control field mnemonics without the worry about what fields may also be using the same control field mnemonic.

Additionally, Micro8 allows an overflow to be defined for each control field. When a value too large for the control field is assigned, it assigns the unassigned part of the value into the overflow control field [12]. This allows unique uses of control fields and was included in the implemented microcode simulator.

**Microinstruction Labels**

Labels in a microprogram really represent the microinstruction address of the line with which they are associated. When labels are used in the microprogram, the labels then correspond to the the value of the microinstruction address where the microinstruction label was defined. As with any programming language, it is useful to allow multiple labels for one microinstruction since the microinstruction may be the branch target of many microinstructions and the branching may be for different purposes. For instance, a microinstruction may indicate the target of a branch instruction and may also indicate the beginning of a loop.
In addition, microinstruction labels can be numeric addresses specifying the address of the microinstruction. These are useful for microprograms that use branching techniques that require branch addresses to align against certain microinstruction addresses. These numeric labels were inspired by the required numeric labels used in Tanenbaum’s Pascal based microprogramming language used in the MX Simulator [5]. A sample microinstruction with microinstruction labels declared and used is:

```
  negbranch: memtox, zerotoy, ztopc, jmpalways, jmpaddr=isa;
```

The microinstruction has a microinstruction label `negbranch` which can be used in other microinstructions to assign the microinstruction address to a control field. The microinstruction also uses the control field mnemonics `memtox`, `zerotoy`, `ztopc`, and `jmpalways`. Also, the microinstruction uses the `isa` microinstruction label to assign it to the control field named `jmpaddr`.

**Macros**

Often there are groups of control signals that need to be set in a particular manner to perform a certain higher level operation. The Micro8 assembler provided macros to meet the need to simplify the assignment of a group of control signals to perform this higher level operation [12]. Macros are just like microinstructions in that control field labels can be used, control fields can be assigned and microprogram line labels can be used. The difference is that a macro is not a part of the microprogram, it is used within a microinstruction similar to a control field label to easily assign all the things assigned by the macro. Macros cannot be assigned microinstruction labels but are given macro names that are used within a microinstruction to utilize the macro.

Macro8 only allows the microcode designer to write macros since the designer is the most logical user of the microprogram macros [12]. Both the datapath designer and the microcode designed can specify macros in the implemented simulator which allows for
greater flexibility. Through the use of macros, the datapath designer can simplify the micro-

code by making certain universal operations for the datapath into macros.

The following is a simple example of making the previous instruction into a macro:

```plaintext
# defines a macro as the same microinstruction as before
@macro mymicro memtox, zerotoy, ztopc,
   jmpalways, jmpaddr=isa;

# use the macro to implement a microinstruction
negbranch: mymicro;
```

The example defined a macro name `mymicro` that uses a few control field mnemonics
and a control assign to a microinstruction label. The microinstruction below the macro
definition uses the macro name and also uses a microinstruction label.

### 4.1.3 Simple Microinstruction Format

In the AMISS tool, the microinstruction format is very simple. It is a list of all the values
for the controls [19]. This format is not the easiest to read since it requires knowing the
order of the control field values in the format and what the values mean for each of the
control fields. However, when writing massive amounts of microcode, this format can
be easier to use after using it for a while for the same datapath. However, this simple
microinstruction format cannot be mixed with the other complex microinstruction format
that uses field labels and field assignment within a single microinstruction. But within a
microprogram, different microinstructions can use either format and the user is not forced
to use only one microinstruction format for the entire microprogram. Additionally, the
control fields are sorted alphabetically for the order of the control fields values rather than
having them explicitly ordered by the user.

This simple format provides additional flexibility in the microinstruction input. For ex-
ample this format is also easier for another program to generate since it does not require
knowledge of the control field mnemonics or control field names. This simple micro-
instruction format allows this program to operate with other tools by using making a small
parsing utility to convert formats to this simple microinstruction format.

The following example uses the instruction in the last example but with the simple
microinstruction format instead:

```
# negbranch: memtox, zerotoy, ztopc, jmpalways, jmpaddr=isa;
# the same microinstruction as the above line
# just uses the simple microinstruction format instead
negbranch: 1 2 0 0 0 0 2 3;
```

### 4.2 Microprogram Parsing

The parsing for the microprogram and datapath macros uses Flex [18] and Bison [6] (the
GNU versions of Lex and Yacc). Backus Naur Form (BNF) is a formal syntax to easily
specify the grammar of a language [7]. Using the original BNF, the microprogramming
language can be formally be described as the following:

```plaintext
<microprogram> ::= |
    <microprogram> <microinstruction>
<microinstruction> ::= <microlabels> <valueslist> ; |
    <simpleformat> ; |
    @macro <identifier> <valueslist> ;
<microlabels> ::= |
    <microlabels> <identifier> : |
    <microlabels> <number> : |
<valueslist> ::= |
    <valueslist> <identifier> , |
    <valueslist> <identifier> = <identifier> , |
    <valueslist> <identifier> = <number> , |
<simpleformat> ::= <number> |
```
The BNF is implemented using Bison with additional error handling to report errors in microprograms that do not conform to the microprogram syntax. The tokens of the language are “@macro”, “:”, “,”, “;”, identifiers, and numbers. Flex is responsible for analyzing the microprogram and converting the microprogram into these tokens to be used by Bison. Additionally, Flex also removes comments from the microprogram as it converts the microprogram into tokens.

4.3 Microprogram Verification

It is better to inform the user of his or her mistakes as early as possible. When writing a microprogram, as with any programming language, there are many possible errors that can exist. The most obvious errors are due to lexical analysis where the microcode does not conform to the microinstruction format. These are easy to detect and report information to the user such as line number, the exact text that caused the failure and what the microprogram parser expected as text instead of what it received. Since the lexical analysis and parsing are done by Flex and Bison respectively, these tools provide an easy method to provide all this information to the user.

However, there are more errors than just parsing the microinstructions. Each micro-instruction line label should only be used for one microinstruction to avoid ambiguity. Each label should be either a valid control field mnemonic or macro name. Also, each control field assigned should use a valid control field name and use a valid number or microinstruction line label.

Also, when a value too large is assigned to a control field, an error should be presented to the user since those extra bits that should not be ignored. This validation is more complicated than expected since the control fields can specify overflow fields and need to be considered.
The final verification of the microinstruction ensures that each control field is not being forced to more than one value by the microinstruction. This is necessary since the control fields mnemonics and macros may hide the control fields being used in the microinstruction.

Since the datapath macros do not have any knowledge of the microinstructions, the datapath macros cannot use any microinstruction line labels. This ensures the datapath macros are not forcing the microprogram to have certain defined microinstruction line labels.

To aid in microprogramming, the control field mnemonics along with other useful control field attributes need to be configured before writing any microinstructions. These are configured through the control configuration dialog.

### 4.4 Configuration of the Control

The aim of this work is to develop a graphically reconfigurable microcode simulator. That includes configuring the control unit graphically. The result is a configuration dialog that shows all the control fields. When a control field is selected, all the control field mnemonics for the selected control field are listed. Also, the control field length can be modified since control fields tend to use many different lengths that are not necessarily the same as the datapath bus width. The control field overflow can also be specified using the control configuration dialog. The control configuration dialog is shown in Figure 4.2.

![Figure 4.1: Datapath Macros Dialog](image-url)
Since having the datapath macros in the control configuration dialog would make the
dialog seem bulky and inconsistent, the datapath macros are configured in a separate dialog.
The datapath macros dialog is shown in Figure 4.1.

![Control Properties Dialog](image)

**Figure 4.2: Control Configuration Dialog**

Both the control configuration dialog and the datapath macros dialog are accessible
from both the datapath editor (described in Chapter 5) and the microprogram editor (de-
scribed next). The datapath editor needs to be able to modify the control and datapath
macros. The microprogram editor needs to be able to review the control features that are
fundamental to writing microcode.

The integrated microcode assembler allows coordination of datapath and the microcode
format specification. The microprogram editor is the user interface to the microcode assem-
bler. Figure 4.3 shows the microprogram editor with sample microcode from the SBN ISA
described in Appendix A. It allows microprograms to be easily loaded and saved, has basic editing controls, and provides access to the control configuration and datapath macros. Additionally, the datapath editor shows highlighting for the current microinstruction being simulated and provides breakpoint control. These simulation features are discussed furthered in Chapter 6.

4.5 Implementation Details

This microprogramming design was used to develop a software design. Figure 4.4 shows the collaboration diagram for the control implementation. The main classes are the Field
The Field class is derived from the Signal class. The Signal class encapsulates the signal data. The signal data can contain any number of bits, can be represented in different formats and has arithmetic functions that allow many operations to be performed on the signal data. The Field class stores the mnemonic names and values within a SignalHash class object. The SignalHash class provides a hash object for storing Signal class objects using a name. The Field class also stores the Field overflow information.

The MicroInstruction class uses the SignalHash class to store all the control field mnemonics, control field assignments, and macro name usages for a single microinstruction. The class also stores the line number where the microinstruction starts and ends (for when a microinstruction is used on multiple lines) so that the line can be highlighted by the microprogram editor. The microinstruction also stores the breakpoint information. The microinstruction also stores the evaluation information about how many times it has been evaluated, the first iteration it was evaluated, and last iteration it was evaluated. This evaluation information is used for determining the CPI of the isa microinstruction using the
following formula:

\[
CPI = \frac{\text{last isa iteration} - \text{first isa iteration}}{\text{number of isa iterations} - 1}
\]  

(4.1)

The Control class has the microprogram counter (mpc) as a field signal (since it could be a control field). The Control class also stores the fields using the SignalHash class since the Field class is derived from the Signal class. The Control class also stores the microprogram and macros.

The ControlConfig, MacroConfig and MicroProgramEditor classes all use the control class object to view and modify various aspects of the control unit. The ControlConfig also uses the currentField variable for the control field being viewed and modified.

The MicrocodeParser is used to parse the macros and microprograms. It stores the control it is parsing for, the microinstruction currently being parsed, and the unresolved address labels. The unresolved address labels allow microinstruction labels to be used before they are declared. At the end of parsing a microprogram, unresolved address labels are reevaluated to ensure that they were declared and assign the proper values to the control fields using the unresolved address labels.

### 4.6 Conclusion

The microprogramming language is one of the core design components of a microcode simulator and is one of the most widely used parts of the simulator. Therefore the microprogram language needs to be easy to specify and simple to use to allow microprogram writers to focus more on writing microinstructions rather than the semantics of the microprogramming language. Also, the configuration dialogs and microprogram editor assist in specifying the control unit and microprogram. In the next chapter, the datapath editor and supporting configuration dialogs are detailed.
Chapter 5

Datapath Depiction and Configuration

The datapath consists of many datapath components and signals. Components perform operations on the signals such as a multiplexer (mux) component selecting a signal, an arithmetic logic unit (ALU) component performing arithmetic operations, or a memory component reading and writing signals into a large store of signals. Signals transfer data from component to component and represent the wires of a real processor.

This chapter overviews the graphical user interface of the developed microcode simulator. First, the visual depiction of the datapath is described. Then, the reconfiguration of the datapath using the graphical interface is detailed. After the datapath depiction and reconfiguration are detailed, the datapath editor as a whole and the student version of the datapath editor are described. Then all the grammars for the datapath are described followed by a description of the overall graphical datapath software implementation is described.

5.1 Visual Depiction of a Datapath

There are many different methods to represent a datapath. Many representations use similar symbols and techniques. In particular, many datapaths have a standard symbol for multiplexers, demultiplexers, and ALU components. Wires are used to show connections between datapath components but do not show the wires for signals from the control unit (and therefore do not show the control unit).

Each component has a specific associated operation such as selecting a signal from a
group of signals or performing an arithmetic operation. Each component also has three
types of signals associated with it. A control signal is used to determine the operation the
component should perform and typically originates from the control unit. An input signal is
an operand to the operation such as a mathematical operand or a signal to write to memory.
An output signal is the location to store an operation’s result such as the value selected by
a mux or a value read from memory.

To ease familiarity of datapath components, standard symbols are used such as an
isosceles trapezoids for the multiplexers and demultiplexers and the standard ALU sym-
bol. Figure 5.1 shows these standard components.

![Figure 5.1: Adder, ALU, Demux and Mux Components](image)

Other standard components such as memory and registers have a distinct appearance to
quickly identify the component’s type visually. Figure 5.2 shows these other components.

![Figure 5.2: Constant, RAM, Register File, and Signal Viewer Components](image)

To make the datapath quicker to read, control signals are not shown on components to
reduce the clutter of wires shown on the datapath. However it may be beneficial to use
labels to indicate the control signal name so that there is a visual correspondence between
the datapath and the microprogram.

Another way to make the datapath easier to read, a signal viewer component is provided
to allow the important signal values to be displayed. Additionally, a constant component is
provided to assign a constant and displays the value of the constant on the datapath.

Several datapath diagrams have signals propagating from left to right or top to bottom to show the signal propagation quickly. For example, the datapaths used in Patterson and Hennessy’s Computer Organization book have the datapath signals propagating from the left to the right with only a few feedback signals going to the left. To further ease the reading of a datapath, all the components are oriented in the same direction, from left to right. This means the input signals are on a component’s left side and the output signals are on a component’s right side. While it may be useful in some situations to be able to rotate and flip a component, it would make the user second guess the direction the component’s inputs and outputs. For example if a multiplexer was flipped horizontally, it would look identical to a demultiplexer and make the datapath more difficult to read.

Additionally, there is a default signal size for the signals. This is useful since many datapaths use many signals that typically are all the same size. By having a default signal size, even intermediate signals such as those used during expression evaluation can use the default signal size.

Each component has one of two modes of evaluation. One mode is whenever its inputs change such as a multiplexer or ALU. The other mode is whenever the component is clocked such as a register or memory. To visually discriminate between these two types of components, at least one of the inputs to a clocked component has an arrow on it to indicate it is clocked. This way a register looks different from a custom component.

Additionally, a pipeline register is offered to show a datapath designer’s intent. According to the simulation it is the same as the regular register, but the pipeline register looks like a pipe to show that this register is meant to propagate a signal through to a later pipeline stage. This allows pipelined datapaths to discriminate the pipeline stages quickly with these very distinctive looking components. The appearance of registers and pipeline registers are shown in Figure 5.3.

Since the basic types are not always enough to easily cover the nuances of a datapath’s design, a custom type is provided that provides a variable number of control signals, input
signals, and output signals. The custom component is a simple rectangle which no other components use to let the user know that it is a custom component.

In addition to wires and components, the datapath can contain text messages that allow further explanation of the datapath giving insight into what sections and components of the datapath are doing. Similarly, a signal viewer component is provided to quickly show the signal value in correspondence with the datapath signal location.

![Figure 5.4: SBN Datapath with Datapath Elements Named](image)

Figure 5.4 shows the major components, wires and text messages of a possible datapath. Next, the actual theory for the wire and component interactions are described using graph theory.
5.1.1 Graph Theory

The datapath is a set of components that have various connections to other components within the datapath. The datapath can be represented as a graph. One possible graph representation of the datapath is that each of the datapath components is represented by a vertex and each wire is represented by an edge.

The graph representation used specifies each component input and output port as a vertex and each wire segment as connecting two vertices. Each component then has a list of its vertices and each edge knows its two vertices. Each vertex knows all the components and edges connected to it, so a component can determine which other components are connected to it.

Since the components are treated differently from the wire segments during activities such as moving, the restriction to allow only one component to be connected to a vertex is specified. If two or more component vertices overlap, they are connected by an edge with a length of zero. This allows one of the components to be moved away from the overlapping component vertices without moving the other component.

Figure 5.5 shows the SBN datapath with all the vertices numbered consecutively. Even with this relatively small datapath there are 73 vertices. Considering there are only 21 signals, this results in each signal being represented by 3.5 vertices on average.

The datapath and its vertices shown in Figure 5.5 shows many of the typical vertex usages. Vertices 1 and 2 show how vertices are connected to connect vertical and horizontal edges to avoid having diagonal edges. Vertices 3 through 9 show how edges and components connect using vertices. Vertices 45, 48, 51, and 71 show how when multiple edges meet at one vertex, the common dot notation is used to show that they are connected together. This allows distinction from when edges cross each other without a common vertex such as the crossing of edges between vertices 44 and 45. Now that the datapath depiction has been described, the modification of the datapath using the interface is another very important task of the datapath editor.
5.2 Reconfiguration of Datapath Components

The easy part of making a graphical depiction of a datapath is displaying the datapath, the hard part is in allowing the user to use the interface to modify the datapath. The user needs to be able to add, move and delete components and wires, to change the properties of the components, and to change the signals represented by the wires. First, the wire placement and movement algorithms are described at a user level. Then the signal names configuration and component configuration are described.

5.2.1 Placing and Moving Wires

Moving components and wires uses a grid snap. Grid snap forces the components to align along a grid that limits the possible locations components and wires can be placed [13]. While components are moving, they are moved using the grid snap so the user is informed of the grid snap locations while moving the components. This makes it definitively clear to the user when components and wires are connected.
Many CAD programs for circuit and architecture design use Manhattan routing. Manhattan routing only uses orthogonal lines to draw wires [14]. By putting this constraint on the wires, the resulting datapaths are easier to read since the wires run along the background grid.

When a new wire is added, a technique called rubberbanding is used to make the wire. Rubberbanding shows the start and current point of the wire to be drawn before the user selects the end point of the wire. The GUI draws the diagonal between the two points to make this draw wire stand out against the existing orthogonal wires and makes it conform to the Manhattan routing constraints when the wire is finished. Figure 5.6 shows the state diagram for the wire drawing mode.

![Figure 5.6: Wire Drawing Mode State Diagram](image)

The wires in a datapath diagram represent the signal connections. So when wires are added, modified or removed, the datapath signal connections need to be updated to reflect those changes. For example, when an existing wire is connected to another wire, the signals being connected together need to be merged into one. Also, when a wire segment is removed, the signal needs to be made into two signals with the larger node maintaining the original signal and the smaller node assuming a newly generated signal.

When placing wires, the signal names are not that important and are given randomly generated names. These names can be changed by the user to make them easier to use in other aspects of the implemented simulator.
5.2.2 Signal Name Configuration

The signals in a datapath do not need to be named. This is because the wires drawn on the datapath implicitly make the signal connections according the wires.

However it is much easier for the user to understand the signals if the signals have a meaningful name. Therefore, when components and wires are drawn, they are given randomly generated names. The user then can specify a name so that this signal can be used in expressions and distinguished in the signal list window. When wires are being modified, real names given by the user are given preference over generated names in order to keep the user specified signal names intact.

Figure 5.7: Signal List Window

Figure 5.7 shows the signal list window for the SBN datapath. There are no signal names given besides the name for the microprogram counter (MPC) signal since all the pertinent information for the datapath is given by the register component, RAM component displays, and signal viewer component displays on the datapath itself. The signal list allows signal names, lengths, and values to be modified. The signal list also allows several signals
to have their lengths or values modified.

As shown in Figure 5.7, the generated names follow the format “_HHHHHHHHH” where H is a hexadecimal digit. This was chosen since it is a very unlikely as a descriptive signal name, follows a simple format, and provides over four billion possible names. When a generated name is needed, the first unused generated name is used. This allows predictibility in the generated signal names.

Wires do not necessarily do not need to connect two components to the same signal. This can be accomplished by assigning the same signal name to both the components and a more friendly signal name aids in this task.

5.2.3 Component Configuration

For components that have a variable number of signals, a configuration dialog allows modification of the number of control signals, number of input signals and number of output signals that are allowed to be modified for the component. For instance, a multiplexer can have the number of input signals changed, but always has one control signal and one output signal.

One can only go so far with standard components in making a datapath. Sometimes even the simplest datapaths have a peculiar feature that cannot be easily implemented by the standard components. Even the standard component ALU is varying since its operation is customizable. The customization allows each output to have an expression associated with it. The more flexible the programming language is for these expressions, the more reconfigurable the entire datapath is in allowing for a variety of datapath types.

Hardware description languages provide a solid basis to build such a programming language since they deal more with the hardware level than a software programming language. However a software programming language is better suited since an action is defined on a behavioral level rather than a hardware structure level. The C based programming language implemented for the expression language has added constructs for bit addressing, status flags like overflow and carry, and operations that are used on a hardware level such
as rotation.

Figure 5.8: Custom Component Configuration Dialog

The custom component configuration dialog for the MPC logic for the SBN datapath is shown in Figure 5.8. The dialog for the custom component allows controls, inputs, and outputs to be added and removed. For other components, the Insert and Remove buttons may not appear if the component cannot change the number of signals for that group of signals. Additionally, the expression of the selected output signal can be modified using the expression editor at the bottom of the window. The expression editor is only shown for the custom component and ALU component configuration dialogs.

Figure 5.9 shows the register configuration dialog. Since the only items that can be modified are the one input signal and the one output signal. All the components use the
same base configuration dialog code and only display the needed items to modify the control. As a result, the control signals are not shown, the insert and remove input signals are not shown, the insert and remove output signals are not shown, and the expression editor for the output signals is not shown.

The memory configuration dialog is shown in Figure 5.10. The memory configuration allows the signal names to be modified just like the other component configuration dialogs. Additionally the memory configuration dialog allows the memory size to be adjusted by specifying the bits per memory element and the number of memory elements. Also, the memory elements can be viewed and modified. The dialog shows four memory elements at a time, allowing the user to input the memory address to view for the first element shown. The address can be quickly changed to view the next four or previous four memory elements using the Next and Previous buttons respectively.

The memory configuration dialog also has a menu bar. The File menu allows the memory state to be loaded and saved to files using the memory file format. This allows the memory state to be saved using the three accepted memory formats: a simple format, the Intel Hex Format, and the Motorola S-Record format. The View menu provides access to open the ISA code for the datapath. The ISA viewer is a simple text viewer that shows the file that is loaded in the instruction memory. Now that the datapath depiction and configuration are detailed, the whole datapath editor will be described.
Figure 5.10: Memory Component Configuration Dialog
5.3 Datapath Editor

Figure 5.11: Screenshot With The SBN Datapath Loaded

Figure 5.11 shows the datapath editor window with the SBN datapath loaded. The datapath editor allows the datapath to be depicted and modified using the interface. It shows how the datapath can be scrolled both horizontally and vertically to allow the visible portion to be scrolled for large datapaths and small screen resolutions. Additionally, the zoom feature allows for very high and very low zoom levels for both small and large datapaths to be completely shown in the window. The datapath editor also provides access to all the other dialogs within the application since the datapath editor is the main application window. Next, the student version of the implemented simulator will be described.
5.4 Student Version’s Datapath Editor

The student version of the application does not allow the datapath to be edited. Therefore the student version’s datapath editor allows the datapath only to be depicted and not modified. This is useful when a professor makes a datapath for students to write microprograms to implement a specified instruction set. It is simpler for the students not to be encumbered by the datapath modification aspects of the interface.

The student version removes several menu components including saving the datapath, the entire Edit menu and the entire Draw menu as shown in Figure 5.12. Additionally, the components configuration dialogs allow the information only to be viewed and not edited.

The implementation uses the C preprocessor variable \textit{NOEDIT} to indicate the student version. It uses this C preprocessor variable to either strip or condition sections of code to create the student version.

Figure 5.12: Student Version Main Application Window

The student version of the application does not allow the datapath to be edited. Therefore the student version’s datapath editor allows the datapath only to be depicted and not modified. This is useful when a professor makes a datapath for students to write microprograms to implement a specified instruction set. It is simpler for the students not to be encumbered by the datapath modification aspects of the interface.

The student version removes several menu components including saving the datapath, the entire Edit menu and the entire Draw menu as shown in Figure 5.12. Additionally, the components configuration dialogs allow the information only to be viewed and not edited.

The implementation uses the C preprocessor variable \textit{NOEDIT} to indicate the student version. It uses this C preprocessor variable to either strip or condition sections of code to create the student version.
Now that the datapath editors have been described, the datapath depiction must be saved in a file format so that the datapath can be loaded. This is done through the datapath grammar.

5.5 Grammars Used

The datapath requires several different grammars to provide inputs for the datapath, the initial memory values, and the expressions for customizable components.

5.5.1 Comments

As used in the microprogramming language, all the grammars used for the datapath, the initial memory values and the expressions language use the “#” and “//” comments. Additionally, the memory grammar uses “;” as a comment prefix to allow additional memory file formats to be acceptable by the memory grammar. Using the same comments for all the grammars provides coherence between the grammars.

5.5.2 Datapath Grammar

The datapath grammar is used to store the datapath information. The datapath grammar is responsible for saving all the information about the default signal size, the signals, the control fields, and the datapath components. All the attributes begin at the beginning of a line (which is why every keyword starts with <newline>). The program uses only one line for each attribute even though it is mandated by the grammar (to avoid names that may have the same name as a keyword). The following BNF describes the datapath grammar at a high level:

```
<datapath> ::= <datapathitem>
     <datapath> <datapathitem>
<datapathitem> ::= <newline>dss <number> | ...
```

62
The default signal size is stored using "<newline>dss" keyword. The field overflows are stored using "<newline>ovf" keyword specifying the field name using the overflow and the overflow field name. It was specified using this method since it is easier to read the fields first then the overflow information so that the overflows fields have already been defined.

The datapath macros are stored using the "<newline>mac" keyword specifying a string that contains the datapath macros text. Since the double quote characters are not valid characters within the macro besides within comments, the double quote characters can be converted to a character sequence to store in the datapath file and reversed when displayed to the user. The double quote characters are converted to "\q" and backslash characters, "\", are converted to "\s". This can be passed to the macro parser since it strips out comments and yields the same functional parsing results. And these character conversions can be reversed easily when displaying the converted macro string back to the user.

The following BNF describes the <signal> and the <controlfield> used in the high level datapath BNF:

<signal> ::= <newline>sig <number> <identifier>
<controlfield> ::= 
   <newline>fld <number> <identifier> <number> <fieldinfo>
<fieldinfo> ::= | 
   <identifier> = <number>
The signal is stored using only the vital information which is the size of the signal and the name of the signal.

The control field also specifies the field size and the field name just like the signal. The control field in general was inspired by the Macro8 assembler which specified the size, name, and control field mnemonics and corresponding control field values. The implemented control field grammar also specifies the bit position in the microinstruction but the bit position is not used by the program. The control field mnemonics are specified at the end of the control field definition since the number of control field mnemonics is variable.

The following BNF describes the <vertices> used in the high level datapath BNF:

<vertices> ::= <vertex> | <vertices> <vertex>
<vertex> ::= <newline>vtx <number> <number>

The vertices section of the datapath defines all the used vertices in the datapath. Later on, the datapath components reference the vertices by using the index of their definition order. For example vertex index 2 indicates the 3rd defined vertex in the datapath file.

The following BNF describes the <datapathcomponent> used in the high level datapath BNF:

<datapathcomponent> ::= 
<newline>add <names> <xy> <vectorlist> | <newline>alu <names> <xy> <exprlist> <vectorlist> | <newline>con <number> <names> <xy> <vectorlist> | <newline>cst <number> <numbers> <names> <xy> <exprlist> <vectorlist> | <newline>dmx <number> <names> <xy> <vectorlist> | <newline>edg <names> <xy> <vectorlist> | <newline>msg "<non"chars">" <names> <xy> <vectorlist> | <newline>mux <number> <names> <xy> <vectorlist> |
All the components specify the signal names, x coordinate, y coordinate, and vector list. For the components that use expressions (the ALU and custom component) the expressions are also specified. At the beginning of each component definition, additional component specific information is specified. For the constant component, the number of outputs is specified (even though currently only one output is supported). For the demultiplexer component, the number of inputs is specified. For the message component, the message text is specified. For the multiplexer component, the number of outputs is specified. For the RAM and the register file components, the number of bits per elements, the number of elements in the memory, if the memory is the instruction memory, and the file name for the initial memory state are specified. For the signal viewer component, the number of inputs is specified (even though currently only one input is supported).
5.5.3 Memory Grammar

The memory grammar is very simple and accepts a simple format, the Intel hex format and the Motorola S-Record format. This allows for flexibility in assigning the initial memory values. Other tools can be used to assemble the instruction program without the need to convert it using another utility if it supports one of these three memory formats.

The simple memory format uses hexadecimal values to assign each element a value. The elements are stored sequentially starting at address 0.

The Intel hex format starts with a colon and is followed by a hexadecimal string. The actual representation of the hexadecimal is determined by the Intel hex format which is too detailed to describe here.

The Motorola S-record hex format starts with a “S” character and is followed by a hexadecimal string. The actual representation of the hexadecimal is determined by the Motorola S-Record format that is also too detailed to describe here.

The BNF for this simple grammar for memory initial values is described by the following:

\[
\begin{align*}
\text{<memory> ::= } & \text{ <memoryline> | } \\
& \text{ <memory> <memoryline>} \\
\text{<memoryline> ::= } & \text{ <hexnumber> | } \\
& \text{ :<hexnumber> | } \\
& \text{ S<hexnumber>} \\
\end{align*}
\]

5.5.4 Expression Grammar

The expression language developed for the implemented simulator allows for configuration of the ALU and the custom datapath component. This full language is based on the C programming language expressions with ideas from VHDL, a hardware description programming language.

Numbers can be specified just as they are specified in the C programming language
such as simple decimal numbers or numbers using prefixes ("0b", "0o", "0d", and "0x") to specify the base. Additionally, a number string can be specified using B, D, O, or H to specify the base to interpret the string as used in the STEP Development Tools [27]. Additionally the dollar sign prefix can used to specify a hexadecimal number as is used in many assembly languages.

The standard operations from C that are provided are ternary conditional operator, logical inclusive or, logical and, logical exclusive or, boolean inclusive or, boolean and, boolean exclusive or, equality testing, inequality testing, less than testing, greater than testing, less than and equality testing, greater than and equality testing, shift left, shift right, addition, subtraction, multiplication, modulus, division, negation, logical not, boolean not, and casting. Additionally the programming structure operations from C are the assignment statement, the if conditional statement and else conditional statement, the for looping statement, the while looping statement, and the do looping statement. Additional operations provided are rotate left "<<<", rotate right ">>>", bit catenation ".", bit indexing, range of bits indexing, and flag retrieval (for carry, overflow, zero and negative flags). Additionally, the case statement is a simplified version of the C programming language’s switch statement and is more based on the Pascal programming language’s case statement.

The following example utilizes several of the interesting aspects of the expression language:

```c
# combine the flags into one
a[carry].a[overflow].a[negative].a[zero];

# rotate left
rotate = alu <<< shift_value;

# programming structure for alu verification
case (instruction) {
    1: alu==0;
    2: alu==1;
```

67
The expression grammar is very versatile and allows many operations to be easily written. The expression parser converts the expression into an expression tree for evaluation.

The BNF for the expression grammar is very similar to the C programming language. Therefore, the BNF for the C programming language and the expression language implementation serves as a reference for the general BNF for the expression grammar.

### 5.6 Implementation Details

Finally, the software implementation of the datapath depiction, configuration, and grammars needs to be described. Figure 5.13 shows the collaboration diagram for the GUI implementation. The classes that represent the windows and dialogs are the DatapathEditor class, the MicroProgramEditor class, the ComponentConfig class, the ControlConfig class, the DatapathMacro class, the IsaEditor class and the SignalList class. The MicroTiger class is the class that starts the application and invokes an instance of the DatapathEditor.

The DatapathEditor in the main application window allows access to all the other windows and dialogs in the program. The main functions of the DatapathEditor class are to display and modify the datapath, and allow access to other aspects of the system such as the microprogram and ISA level code. It always has one instance of the ProcessorCanvas which is the display for the datapath. It also sets up the Error class to be used throughout the application.

The Error class allows for easy access to provide error handling in all the classes. The DatapathEditor class creates an instance of the Error class and the Error class stores this class statically. Therefore the other classes can use static function calls to the Error class to use the created Error class object without having to pass around the Error class object to all the objects of classes that have errors to present to the user. The Error class is implemented
by both the ErrorGui class and the ErrorConsole class. The ErrorGui class uses a log
window to provide a log of the errors that occurred and is shown whenever an error occurs.
The ErrorConsole class prints the errors to the standard output and is used within the test
benches and the MicroTiger console application.

The MicroProgramEditor, ControlConfig and MacroConfig classes were reviewed in
Chapter 4.

The ProcessorCanvas class extends the Processor class. The Processor class implements
all the simulation and component information. The ProcessorCanvas implements all the
GUI handling code such as drawing, handling the properties dialogs for the components,
and coordinating the mouse state information for moving and modifying the components
on the processor display.

![Collaboration Diagram For The Component Inheritance Usage](image)

Figure 5.14: Collaboration Diagram For The Component Inheritance Usage

The Component class implements the basics used for all the components. Then each
type of component is defined as its own class that extends the Component class. These
classes are shown in Figure 5.14. The classes are the Adder class, the Alu class, the Con-
stant class, the Custom class, the Demux class, the Edge Class, the Memory class, the
Message class, the Mux class, the Register class, and the Signal Viewer class. The spe-
cial Edge class could have been its own class, but since it uses a good portion of the same
properties as the Component class. The resulting Edge class extends the Component class,
changes a few functions, and ignores several functions. The special Message class makes it
easier to include using the comments in the drawing and moving algorithms even though it
has no signals or vertices associated with it. The special Constant class is used for providing constants for the datapath. It shows the constant value being used and is only evaluated when the datapath is reset. The special Signal Viewer class is used for displaying a signal’s value.

The Memory class itself is extended by the Ram class and the RegisterFile class. The Memory class is implemented as a hierarchy itself since it provides the memory file format that is useful for more than just simple RAMs. It could be used for Register Files, ROMs, caches, and other components that could be implemented later on.

As shown in Figure 5.14, the ProcessorParser class and the ProcessorCanvas class are aware of the individual component classes. All the other classes that use the Component class use the only the Component class and do not care about the details of the individual classes. Additionally the DatapathEditor class knows about the individual component classes since it creates them through the ProcessorCanvas.

Figure 5.15: Collaboration Diagram For The Expression Inheritance Usage

Figure 5.15 shows the collaboration diagram for the hierarchy of the Expressions classes. The Expression class sets up the basic data access structure for the Expression subclasses and each of the Expression subclasses implements the actual function to perform. The ExpressionParser converts the expression into an expression tree that consists of Expression class objects.

The Expression subclasses are the UnaryExpression class, the BinaryExpression class, the TernaryExpression class, the SignalExpression class, the ListExpression class, the ConditionExpression class, the IterationExpression class, and the CaseExpression class. These classes are shown in Figure 5.15. Unary operations such as logical NOT and negation result in a UnaryExpression class object. Binary operations such as addition and logical OR result
in a BinaryExpression class object. Ternary operations such as the the bit range operator, “x[y:z]”, and the bit range assignment, “x[y:z] = expression”, result in a TernaryExpression class object. Both constant values and variables result in a SignalExpression class object. The list of statements that are separated by commas or semi-colons result in a ListExpression class object. The if and else keywords result in a ConditionExpression class object. The iteration keywords for, while, and do result in an IterationExpression class object. The case keyword results in a CaseExpression class object. After these objects are created, they are treated only as Expression class objects by the rest of the code.

5.7 Conclusion

The datapath viewer and editor is a very complex system. It allows the datapath to be viewed and all the aspects of the datapath to be modified including the signals used, the expressions used for the customizable components, the number of signals a component has, and intuitive wires for connecting components. The datapath viewer also shows the simulation results. The simulation of the datapath with its microprogram and ISA code is detailed further in Chapter 6.
Chapter 6

Simulation

Simulation is the main aspect of the developed microcode simulator. The simulator developed models of various aspects of a microprogrammed processor such as signals, datapath components, the control unit, and microinstructions. Additionally, the microcode simulator needs to allow the results of the simulation to be quickly shown and easily understandable.

This chapter overviews the details of the simulation engine for the developed microcode simulator. First, the basic building blocks are described: the signals and the datapath components. Next, control simulation aspect is described. Then the execution of the entire processor is described along with how the simulation results get depicted. The simulation software implementation is then detailed. Finally, the results of the implemented microcode simulator are described including source code comparisons to other microprogramming tools, performance comparison to other microprogramming tools, and performance comparison within the implemented simulator for various datapaths.

6.1 Signals

The signals represent the fundamental data type for the processor since the signals are utilized in nearly every aspect of the simulator code. The signals need to be highly flexible and easy to use. The signals provide flexibility in being able to use any number of bits, provided the machine the program is running on has enough storage for all the bits being used. The signals have many common operators which were developed to support the
expression language. Additionally for any given operator, its operations can vary based on
the casting type applied to the signals given to the operation. The casting types allow the
bits be interpreted as unsigned integers, signed integers, floating point decimal numbers
and double precision floating point decimal numbers. The signals also retain information
about carry and overflow for various arithmetic operations to aid functions that use these
arithmetic results. The signals are mostly utilized by components that perform various
operations on the signals and drive the requirements of the signals.

6.2 Datapath Components

The datapath components are the operation performers of the datapath. They perform an
operation specified by a set of control signals on a set of inputs signals to produce a set of
output signals. Each component has a set of control signals, a set of input signals, and a set
output signals. While the input signals and control signals can be grouped together under
several circumstances, it is beneficial to maintain the sets as two separate sets of signals.
Additionally, each component has a set of vertices for the inputs and outputs. These vertices
provide the connection points on the datapath canvas to connect one component to another
component. The custom component, the ALU component, and the constant component all
have a set of expressions to produce the set of output signals.

In order to make evaluation of the components easier, there is a restriction to allow only
one output to be connected to a signal. This only is not a good practice to ensure the results
written to the signal are the results expected, but also makes ordering the components
for evaluation easier. Additionally, it is easy to work around this restriction by using a
multiplexer to select which signal should be assigned to a particular signal at the cost of
adding one control signal and one component.

There are two main tasks for the datapath components for simulation. One is how to
order the datapath components for simulation. The other is how to evaluate the expression
language to provide the custom component and ALU reconfigurability.
6.2.1 Ordering Of Datapath Components

The datapath components should be ordered so that during simulation each component can be evaluated once per simulated clock cycle. First, the non-clocked components are evaluated followed by evaluating the clocked components. Certain components are not be evaluated: edges, text messages, constants and signal viewers. The clocked components are the standard registers and pipeline registers. All the other components are considered non-clocked components.

For non-clocked components, the components that do not rely on any other clocked components’ output(s) are evaluated. For clocked components, the reverse is true. Clocked components that rely on other clocked components’ output(s) are evaluated once those components are evaluated. Any component that uses another components’ output(s) should be evaluated first to avoid propagating a signal though several clocked components in one cycle. The actual algorithms used to sort these components is the topological sort from the field of graph theory.

Graph Theory and Topological Sort

Non-clocked components and clocked components can be treated as two separate directed graphs. A directed graph means that information flows in a specific direction. The direction is from the component output signals to the connected component input signals. Components that output a signal need to be evaluated before the components that use that output signal as an input signal. The restriction that only one output can drive a signal simplifies knowing when the signal is set.

An additional restriction is that no component can use feedback for its inputs. This means that there should be no cycles within the directed graph. If any cycles exist, the user should be presented with an error about the components involved in the cycle. Therefore when the components comply with this restriction the components form a valid directed acyclic graph (DAG). These directed acyclic graphs are found very often with datapaths and are used by layout optimization algorithms such as the Optimal Integer Delay Budgeting
on Directed Acyclic Graphs algorithm developed by Elaheh Bozorgzadeh, Soheil Ghias, Atsushi Takahashi, and Majid Sarrafzad [1].

Figure 6.1: Directed Acyclic Graph for SBN Datapath’s Non-clocked Components

Figure 6.1 shows the Subtract and Branch if Negative datapath with the non-clocked components numbered and a directed acyclic graph for the evaluation order of those components using the assigned numbers. As the graph shows, either component 2, 4, or 5 must be the first non-clocked components evaluated and either component 3 or 7 must be the last non-clocked component. A topological sort goes beyond this and orders all the components in a list so that their directed order is maintained. Since there are typically many possible lists when such a sort is performed, the exact order does not necessarily matter as long as the directed acyclic graph order is maintained. When no topological ordering
exists, it indicates that there is a cycle in the directed graph and an error is reported to the user indicating which components cannot be ordered. In the example, there are 34 possible orderings of the seven components.

For the clocked components of the SBN datapath, the ordering does not matter since none of the clocked components share any of the same signals. However, if they did share any signals, the clocked component that has shared inputs is evaluated before the clocked component that outputs the signal. The topological sort for the clocked components is very similar algorithmically to the topological sort for the non-clocked components.

All the components can be stored in a single list, containing the sorted lists of non-clocked components, clocked components, and non-evaluated components (in that order). Then when simulation occurs, the evaluated components can be evaluated using the one list and the non-evaluated components at the end of the list can be ignored.

6.2.2 Expression Language Evaluation

The expression grammar is used to parse an expression into an expression tree ready for execution. The following expression is the expression used in the MPC logic custom component for the Subtract and Branch If Negative example:

```c
if ((jmplogic == 1 && input0[negative]) ||
    jmplogic == 2) {
  jmpaddr;
} else {
  mpc+1;
}
```

The expression returns `jmpaddr` only when the `jmplogic` is 1 and the subtraction resulting in a negative number or when `jmplogic` is 2. Otherwise the expression returns the incremented `mpc`. 77
The expression tree for this expression is shown in Figure 6.2. The root node is the if statement, which is the same as the C programming language’s ternary conditional operator, “x?y:z”. It then evaluates the condition to see if the condition is true or not. This involves evaluating each of the nodes recursively.

![Expression Tree Example](image)

Figure 6.2: Expression Tree Example

The leaf nodes in the diagram (the nodes with only arrows pointing in and no arrows pointing out) are the signals and the constant values. The other nodes represent functions to perform on the values returned from its children nodes. Sometimes nodes can be reduced during parsing when all the nodes used by a function are constant, the function can then be pre-evaluated to be the constant value it represents. Now that the datapath component’s simulation behavior has been described, the control’s simulation aspects need to be described.

### 6.3 Control

The control allows the control fields and the microprogram to be specified. During evaluation the control assigns the control fields using the microinstruction indicated by the
microprogram counter. The simulation of the control involves using the microinstruction to assign the control field values, determining the next microprogram counter, and retaining information about the microinstruction execution to aid in calculating the CPI.

### 6.3.1 Microinstructions

The microinstruction utilizes the difficult algorithm of creating the microinstruction structure to use during evaluation. This is done in three parts. When the microinstruction is created, each of the control field mnemonics, control field assignments, and macro uses are recorded. The microinstruction is verified at the end of its creation to verify no control field is being set more than once. Then during the first evaluation, the control fields are assigned using the microinstruction’s store control field mnemonics, control field assignments, and macros uses. These control field assignments are then stored using the control field names and control field values for this microinstruction. Then any subsequent evaluations of the microinstruction just copy the signals from the stored values to the control fields. This control field values cache for each microinstruction tremendously improves the performance of the simulation. For example, small datapaths had a overall simulation performance increase by a factor of 2. The other aspect of a microprogram control simulation is the microprogram counter determination that is described next.

### 6.3.2 Next Microprogram Counter

The next microprogram counter is determined by allowing the datapath to assign the \( mpc \) signal. This signal is used by the control unit as the microprogram counter. Therefore each datapath should have at least one component dedicated for the microprogram counter logic since this microprogram logic is typically very different from processor to processor. By having the datapath update the microprogram counter rather than within the control, the datapath can use the calculated microprogram counter within the datapath and expands the possible datapaths possible.
6.3.3 Calculating CPI

The useful feature of calculating the cycles per instruction (CPI) requires additional information about the simulation to be retained. Results from the simulator are more than just register values and other signal values. One important microprogramming simulator result is the CPI for the microprogram and instruction program. This allows the microprogram to be benchmarked against a particular instruction program.

The CPI is calculated using the following equation:

\[ \text{CPI} = \frac{\text{last isa iteration} - \text{first isa iteration}}{\text{number of isa iterations} - 1} \]  

(6.1)

This requires the microinstruction to store the first iteration it was executed during, the last iteration it was executed, and how many times was the microinstruction executed. Since these results are not as important for each microinstruction, only the CPI is presented to the user. Now that the control and datapath components simulation aspects are covered, the execution of the entire processor is detailed next.

6.4 Execution

A datapath and microprogram can be executed only when there is a valid datapath and corresponding microprogram. Additionally, no microprogram is needed when there are no control fields defined. This allows the simulator to be used as a simple datapath simulator. The datapath is only valid when it has at least one signal and is sorted. If it is not sorted, a sort will be attempted to see if it becomes sorted.

The actual simulation is performed in a loop for as many iterations need to be simulated to meet the specified simulation condition. These include running for a specified number of iterations, running to a specific iteration number, and running until the microprogram counter is a specific value. For the case when microprogram counter is specified, only 10,000 iterations will be simulated and if the condition is not met, the execution will
return to the original iteration number by executing backwards. 10,000 was arbitrarily cho-
sen since it is relatively long and can be performed on many simple to moderate datapath
designs in under one second.

6.4.1 Backwards Execution

Backwards execution is important since when debugging a datapath or microprogram it is
convenient to step backwards to try and understand what is going on. To enable backwards
execution, the simulation is reseted and simulated until the specified iteration. This is the
same behavior as the MythSim simulator [26]. Additionally, this feature will mostly be
used on small iterations which makes the execution from the beginning not as costly.

The other backwards execution option would be to provide state information. To save
all the state information on a cycle by cycle basis would quickly use memory resources
available on the machine, especially considering the memory components store all the pos-
sible bits of the memory. The other option would be save the difference between the states
and would require complex algorithms to implement efficiently. Also, state saving would
impact the performance of the forward execution since the state saving would need to be
processed and possibly memory would need to be allocated. Since the performance of
simulator is very adequate, the reset and execute option is sufficient.

6.4.2 Displaying Simulation Results

Simulation within a GUI allows information about the simulation progress and results in
more intuitive manners than console based microcode simulators. The simulation results
can be shown on the datapath diagram, lines of code being executed can be highlighted,
and simulation options like breakpoints can be more easily entered.

The main application window shows the iteration number and the microprogram counter.
This information is very useful and is displayed on the main application window. The
MythSim tool also displays the iteration number and the microprogram counter on the simulator window [26].

One of the main ways to represent datapath results is by showing the register values. One of the output modes of the text based MX Simulator is to display all the register values since these are the most useful values [5]. Similar to the register, the constant component shows the constant value represented by the constant component. Additionally, the signal viewer component allows any signal to be viewed on the datapath. This allows other important signals to be viewed.

The signal list dialog allows all the datapath signals to be viewed for complete analysis of all the signal values. The memory configuration dialog also incorporates a memory value viewer. This allows four consecutive memory addresses to be viewed. The addresses can be changed using either a text box or buttons that allow the previous and the next four addresses to be viewed.

### 6.4.3 Microprogram Highlighting and Breakpoints

During simulation the microprogram editor window highlights the microinstruction being simulated. The MythSim tool also uses microinstruction highlighting [26]. Additionally, this is similar to an integrated development environment (IDE) for normal software development. This technique is very useful and quickly identifies the microinstruction being evaluated.

Additionally several software integrated development environments provide breakpoint selection and breakpoint control. By adding these features to the microcode simulator, the user can see the correlation between the microprogram source for the simulation and the results of the simulation represented by the graphical datapath. These features should improve debugging time for testing new microcode since it offers greater interface to the simulation.
6.4.4 Simulation Animation

Simulation animation allows the simulator to simulate, display the results and wait for a specified delay. One of the uses for simulation animation is for demos. The animation delay is changeable and the animation can be quickly started and stopped by pressing the “A” key. Animation is only used by a small number of microcode simulators. Now that all the aspects of the simulation are detailed, the following section describes how the simulation was designed at a software level.

6.5 Implementation Details

The overall implementation for the simulation aspect of the microcode simulator is shown in Figure 6.3. The Processor class is comprised of both the datapath and the control unit used for simulation. The Processor class has a list of components, a list of vertices, an instruction memory reference to one of the components, an expression parser, a reference to the control and a list of signals. The list of components needs to be sorted for simulation using the topological sort. It contains all the components displayed on the datapath canvas including messages, signal viewers and the edges for the wire segments. The list of vertices is determined when needed since it is only used during wire modification functions. The expression parser is passed to the components to generate the expression trees from an expression string. The control unit is contained within a Control class object. The signals are stored with a SignalHash class object and are a store of the signals used associated with their signal names.

The ProcessorParser class is used to read the datapath file and create the components, control fields, signals, and vertices for the Processor class object.

The Node class is used by the wire moving algorithms. The Node uses a vertex to determine all the vertices and signals connected to a node. Therefore when wires are connected, the signals can be determined to determine what signal should represent the modified node.
Figure 6.3: Collaboration Diagram For The Simulator Implementation
The Signal class is the data type for the processor. It provides the initial data modification and access. The SignalMath class is a friend class to the Signal class. This means the SignalMath class can access all the private data within the Signal class. The SignalMath class provides all the arithmetic operations on the Signal class objects.

The Component class implementation was examined in Chapter 5. The Control class, the Field class, and the MicroInstruction class implementations were examined in Chapter 4.

### 6.5.1 Testing and Validation

When working on any large scale software project, testing is vital to generating a more accurate implementation of the software design. The main testing methodologies used were assertions, unit testing, and overall testing.

The assertions were used mainly to ensure inputs to functions are valid. Assertions were also used to test if a function produced valid results, and occasionally within complex functions to test intermediate values for validity. Mainly these assertions were used to test that pointers were not NULL and that arguments were in valid ranges.

Unit testing performs several tests on the simulator aspects of the implementation. There is unit testing for the Signal class and the SignalHash class which includes approximately 270 tests of various aspects of the two classes. There is unit testing for the Expression class (which also tests the Signal and SignalHash classes) that has approximately 60 tests. Additionally, the Expression class testing uses around 200 different expressions to test the validity of the expression parsing and expression evaluation. The control unit testing uses approximately 450 tests on the Field class, the MicroInstruction class, and the Control class. The full application automated unit testing uses approximately 480 tests to test the full processor and microprogram and ensures the simulation has the correct simulation results.

Additional testing is done by using both the graphical simulator and the console simulator. Several datapaths, microprograms, and ISA code files were developed to test various
architectures that stress various aspects of the simulator.

6.6 Implemented Simulator Comparison and Results

This section describes the results of the implemented microcode simulator. This section compares the simulator to other microprogramming tools for implementation size and performance. Additionally, the simulator performance comparison for various datapaths is detailed.

6.6.1 Implementation Size Comparison with Other Microprogramming Tools

Table 6.1: Implementation Line Count Comparison To Other Microprogramming Tools

<table>
<thead>
<tr>
<th>Application</th>
<th>Total Line Count</th>
<th>Source Lines Of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td>30244</td>
<td>18691</td>
</tr>
<tr>
<td>MythSim 3.0 [26]</td>
<td>8966</td>
<td>6012</td>
</tr>
<tr>
<td>SMDSS [22]</td>
<td>5699</td>
<td>Not Available</td>
</tr>
<tr>
<td>AMISS [19]</td>
<td>6018</td>
<td>3385</td>
</tr>
</tbody>
</table>

Table 6.1 shows the line count comparison between the implementation with MythSim 3.0 and SMDSS. The line count comparison allows an approximate complexity comparison between the implemented microcode simulator and other microcode simulators. The implementation has nearly 30000 lines of code. Using the source lines of code methodology, where comments and other trivial lines are not counted, the implementation has nearly 20000 source lines of code. This is far more than the other microprogramming tools compared. The MythSim is the only tool thoroughly examined with the source still accessible.

Together, SMDSS and AMISS represent the text based reconfigurable tools and MythSim represents the graphical microprogramming tools. The implemented simulator more
than doubles the line count of the line count totals for the SMDSS and MythSim tools combined or the AMISS and MythSim tools combined. This is because the added complexity of combining the two techniques requires more overhead. For example the component configuration dialog is code that neither previous microprogramming tool required but it is required when both techniques are implemented together.

6.6.2 Performance Comparison of Various Datapaths

The complexity of a datapath is dependent on many factors. Several of these factors can be accessed through the Datapath Properties dialog. This dialog shows the total number of components (including edges, constants, signal viewers, and text messages), the number of evaluated components, the total number of signals, the number of datapath signals, the number of control fields, and the number of vertices. The dialog for the SBN datapath is shown in Figure 6.4.

![Datapath Properties Dialog For The SBN Datapath](image)

Figure 6.4: Datapath Properties Dialog For The SBN Datapath

The overall simulation of the implemented simulator as shown in Table 6.2. The datapath properties dialog was used to report the complexity for the Table 6.2. Several datapaths were implemented for the microcode simulator and used configurations of microprograms and ISA code that resulted in an infinite loop. Therefore one million iterations could be performed without using an invalid ISA instruction or invalid microinstruction.
The one million iterations were timed on an old computer \(^1\) in various runs to provide the simulation results. The Datapath Editor reports the duration of the simulation when one million iterations are requested. The SBN datapath is fully described in Appendix A. All the other datapaths are briefly described along with a datapath screenshot in Appendix B.

Table 6.2: Simulation Performance For Various Datapaths

<table>
<thead>
<tr>
<th>Datapath</th>
<th>Components</th>
<th>Vertices</th>
<th>Total Signals</th>
<th>Controls</th>
<th>Simulated Cycles/Sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>tiny.dp</td>
<td>8</td>
<td>29</td>
<td>16</td>
<td>4</td>
<td>202675</td>
</tr>
<tr>
<td>lmms.dp</td>
<td>13</td>
<td>53</td>
<td>18</td>
<td>8</td>
<td>192975</td>
</tr>
<tr>
<td>sbn.dp</td>
<td>13</td>
<td>73</td>
<td>29</td>
<td>8</td>
<td>146627</td>
</tr>
<tr>
<td>mic1.dp</td>
<td>30</td>
<td>133</td>
<td>59</td>
<td>13</td>
<td>84076</td>
</tr>
<tr>
<td>primer.dp</td>
<td>4</td>
<td>13</td>
<td>8</td>
<td>0</td>
<td>63783</td>
</tr>
<tr>
<td>mythsim_write.dp</td>
<td>21</td>
<td>90</td>
<td>49</td>
<td>20</td>
<td>57663</td>
</tr>
<tr>
<td>mythsim.dp</td>
<td>39</td>
<td>181</td>
<td>83</td>
<td>26</td>
<td>42115</td>
</tr>
<tr>
<td>amiss.dp</td>
<td>35</td>
<td>176</td>
<td>66</td>
<td>14</td>
<td>39071</td>
</tr>
</tbody>
</table>

The simplest datapath (“tiny.dp”) has only a few registers and a 16 function ALU and produced 200,000 simulated cycles per second. The “lmms.dp” datapath is designed to be compatible with the datapath in the Little Man Microcode Simulator “lmms.dp” \([3]\). The implemented “lmms.dp” yielded nearly 193,000 simulated cycles. The SBN datapath (“sbn.dp”) produced over 146,000 simulated cycles per second. The “mic1.dp” datapath is an implementation of Tanenbaum’s Mic-1 microprogrammable architecture and yields 84,000 simulated cycles per second. The “primer.dp” datapath is a simple datapath with a complex custom component expression that calculates whether a given number is prime or not for numbers 0 through 255. Due to this complex expression, even though the datapath is small in terms of component count, vertex count and signal count, the expression evaluation is very time consuming and only produced approximately 63,000 simulated cycles per second. The “mythsim.dp” and “mythsim_write.dp” datapaths are based on the datapath

---

\(^1\)Simulations ran on Ubuntu Linux running the simulator that was compiled using gcc version 4.0.2 with the flags “-g0 -O2 -DNDEBUG” and processing on a 598 MHz Intel Celeron (Coppermine) w/ 128 KB Cache processor
used in the MythSim tool [26]. The performance of these datapaths are reviewed in the
next section and will be compared to the MythSim tool. The “amiss.dp” datapath is an
implementation of the datapath used by the Rochester Institute of Technology’s Computer
Organization class that uses the AMISS tool. For intermediate complexity datapaths as
shown by the “amiss.dp” datapath, the performance is very reasonable with over 39,000
simulated cycles per second. So even for complex datapaths, the expected performance
should still be very reasonable since the performance is fairly linear with added complex-
ity.

6.6.3 Performance Comparison with MythSim

Table 6.3: Simulation Performance For Various MythSim Datapaths

<table>
<thead>
<tr>
<th>Application</th>
<th>Datapath</th>
<th>Simulated Cycles/Sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implemented Simulator</td>
<td>mythsim.dp</td>
<td>42115</td>
</tr>
<tr>
<td>Implemented Simulator</td>
<td>mythsim_1write.dp</td>
<td>57663</td>
</tr>
<tr>
<td>MythSim 3.0 (Console)</td>
<td></td>
<td>74718</td>
</tr>
</tbody>
</table>

Table 6.3 shows the number of simulated cycles per second for both the implemented
simulator and the MythSim tool. The implemented simulator had two versions of the
MythSim datapath. One datapath (“mythsim.dp”) had strict conformance to the MythSim
datapath. The other datapath (“mythsim_1write.dp”) only allowed one register to be written
back in a single cycle since the examples provided with MythSim did not use more than one
write-back in a single cycle. The implemented simulator was able to simulate a little over
42,000 cycles per second with the strict conformance to the mythsim datapath. The single
write-back datapath was able to simulate over 57,000 cycles per second. The MythSim 3.0
tool was slightly modified to allow the console simulator to run one million iterations and
quit. As a result, the MythSim tool was shown to provide nearly 75,000 simulated cycles
per second.
The MythSim application differs from the implemented simulator in that the MythSim application was written in Java and does not allow for a reconfigurable datapath. Since the MythSim application is not reconfigurable, it should be much faster since the data structures and execution are much simpler than a reconfigurable datapath simulator. But conversely, since the MythSim application was written in Java, it would be expected to be slower than an equivalent C++ program. According to Lutz Prechelt, the Java version an application is over 3 times slower than the C++ equivalent, but discrepancies in developers and the application itself make this factor widely variable [20]. According to this factor of 3, a C++ version of MythSim should approximately simulate 225,000 cycles per second. Therefore the reconfigurable implementation has a very approximate performance cost factor of nearly 5.5. Given the fact the performance for both is very acceptable (considering that most users of the simulator will simulate less than a thousand iterations), the reconfiguration aspect was worth the performance cost.

6.6.4 Performance Comparison with AMISS

Table 6.4: Simulation Performance Comparison To AMISS

<table>
<thead>
<tr>
<th>Application</th>
<th>Datapath</th>
<th>Simulated Cycles/Sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implemented Simulator</td>
<td>amiss.dp</td>
<td>39071</td>
</tr>
<tr>
<td>AMISS</td>
<td></td>
<td>415627</td>
</tr>
</tbody>
</table>

AMISS is a reconfigurable microcode simulator, so the datapath used for comparison is the datapath used for the Rochester Institute of Technology’s Computer Organization class. Additionally, the same microprogram and memory file were used by both simulators. The implemented simulator was able to perform a little over 39,000 simulated cycles per second. The AMISS simulator, as expected was able to perform better with over 415,000 simulated cycles per second. AMISS outperformed the implemented simulator by a factor of around 10.5. This is to be expected, since AMISS compiles the datapath directly into C which
is then compiled with the simulator base code to produce a resulting executable specific for the datapath. As a result, the AMISS simulator is highly optimized for the defined datapaths. This approach would be difficult to take with a graphical microcode simulator since the datapath needs to be able to be reconfigured interactively and be able to be ready for simulation after the configuration is completed.

6.7 Conclusion

The simulation aspect of the microcode simulator resulted in a high performance simulator. This was accomplished through a good software design, software principles such as caching, and testing to ensure the optimizations were accurate to the software design. The next Chapter draws conclusions about the implemented simulator along with possible improvements on the simulator.
Chapter 7

Conclusions

This chapter summarizes the developed microcode simulator and possible future capabilities that could enhance the simulator. This chapter overviews the salient features implemented in the simulator, the contributions of the simulator to the field, and the areas for future work on the simulator.

7.1 Salient Features of the Implemented Simulator

The implemented simulator accomplished the main goals and resulted in many features to coordinate the combination of the graphical interface and reconfigurable datapath including:

- Provides a datapath editor similar to circuit editor which is not included by other microcode tools.

- Provides many components including an adder, an ALU, a constant, a demultiplexer, a multiplexer, a pipeline register, a RAM, a register, and a register file.

- Provides a customizable component using a C based programming language with several added constructs and a namespace to decouple the expression from the datapath diagram.
• Microprogram format with symbolic names, macros and line labels, including a special \textit{isa} line label to indicate when an ISA level instruction starts.

• Provides several simulation options including running for a specific iteration count (1, 10, 100, 1000, 1000000), run to a specific iteration, run to a specific microinstruction, and simulation animation.

• Provides easy access to simulation results including updating the datapath editor with signal values, iteration number, and microprogram counter, and updating the integrated microprogram editor by highlighting the current microinstruction.

• Provides the CPI for a simulation run.

• Provides an integrated number converter for different bases.

7.2 Contributions

The major contribution of this work is a microcode simulator that is both reconfigurable and graphical. No previous microcode simulator has previously had both of these important features. The main reason this was not done before is that it is much more difficult to design and implement a graphical and reconfigurable microcode simulator. Not only is the reconfigurable datapath displayed using a graphical user interface, but the interface is also used to create and modify the entire datapath.

7.3 Areas for Future Work

While the simulator does perform simple performance evaluation in terms of calculating the CPI for microprograms, this performance measuring could be expanded to other microinstructions and displayed in a coherent manner.

The simulator uses only one clock. However some processors use a dual phase clock and that would be very difficult to implement using the current simulator design. Therefore
adding a multiple clock enhancement to the simulator would greatly improve its flexibility in allowing these datapaths to be used with the simulator.

The simulator’s microcode format could be expanded to allow more high-level constructs. This would require designing an interface to allow the specification of how these high-level constructs are interpreted into control fields and modifying the microcode parser to accept the high-level constructs.

The signals could be expanded to allow different bit orderings and byte orderings (little endian, middle endian, or big endian). The system currently uses the Intel x86 definitions for bit orderings and byte orderings, but other datapaths may use other combinations of these orderings. Additionally, it would be nice to allow the datapath to specify which base is used for all the displayed values since the current is only hexadecimal.

Another possible extension of the simulator would be to provide more datapath components that assist in creating caches, Tomasulo cluster of functional units, exception handling, and other specialized components.

Another useful feature is to write out the compiled microprogram in some sort of memory file format so that other tools can interact with the microprogram such as performing optimizations or converting the microprogram to a vertical microinstruction format. Similarly, it would be useful to be able to write out the datapath in other common file formats such as VHDL or Verilog to use the datapath in other simulation environments.

To aid others to expand upon the simulator, comments were used liberally to help pass on the code design concepts within the code. Additionally, a single reference guide was created to show the inheritance diagrams and collaboration diagrams for each class, describe the documented function description and parameter descriptions, detail the functions that each function calls, and detail the functions that call each function. The reference guide is 361 pages which shows the complexity of the implemented simulator.
7.4 Closing Remarks

Even though implementing the simulator in Java may have eased the platform compatibility, the performance of the resulting C++ implementation is quite impressive.

The complexity of creating a graphical reconfigurable microcode simulator was underestimated. Most of the additional complexity comes from interaction between the graphical configuration of the datapath and the complexity of creating the grammars for configurations. Developing a tested Signal class may have been unnecessary since possible suitable alternatives such as CBigInt [2] and GNU MP [11] were discovered after implementing the Signal class.

The graphical aspect of the simulator allows the simulator to be easily used to quickly develop datapaths, to develop microprograms, and most of all, easily simulate and debug the processor. This made testing the application easier since datapaths and microprograms could be quickly developed to test various aspects of the application.

The graphical microcode simulator with a reconfigurable datapath was able implement the datapaths used by several microprogram simulators. This allows the microprogram simulator to fill the same context as these simulators and allow additional datapaths to be created and simulated without the need of developing a new simulator. This allows the simulator to have extended usefulness since it has the modern user interface that simplifies the user experience and has the configurability that allows it to be used in a variety of situations and environments.
Bibliography


Appendix A

Subtract And Branch On Negative Example

Appendix A overviews the example processor, instruction level code and microprograms for the simple example used throughout the chapters.

The subtract and branch on negative (SBN) ISA was first described by W.L. van der Poel in 1956 [25]. The ISA contains a single instruction, the subtract and branch on negative instruction [25]. Therefore the instruction does not have an opcode and only consists of operands specifying memory addresses. The format for the instruction is:

\[
\text{sbn addr0, addr1, branchAddr}
\]

The value at memory address addr1 is subtracted from the value at memory addr0 and is stored at memory address addr0. If the subtraction is negative, the program branches to branchAddr otherwise the program goes to the following instruction. The address of the current instruction being executed is stored in the program counter (PC). This process is described by the following pseudo-code:

\[
\text{mem[addr0] = mem[addr0] - mem[addr1]}
\]

\[
\text{if ( mem[addr0] < 0 ) } \{
\text{pc = branchAddr}
\} \text{ else } \{
\]
This ISA is an example of a minimal instruction set computer (MISC) ISA. This instruction is capable of performing any operation given enough memory.

A.1 Datapath

The datapath needed for this architecture is very simple. The datapath must have memory, a program counter register, and a subtracter. The datapath in Figure A.1 has these basic requirements. In addition it names the first two operands A and B, and uses register for A, B, the address of A and the address of B. It uses the subtracter for subtracting, passing values through (by subtracting zero), and incrementing a value (by subtracting negative one).

Figure A.1: Subtract And Branch On Negative Datapath
A.1.1 Control Fields

The control fields for the example SBN datapath are jumpAddr, jumpLogic, maddr, mread, mwrite, xsel, ysel, and zdest. These control fields are shown in Table A.1 along with their control field mnemonics. Figure A.2 shows the microinstruction format. The jmpaddr control field is the microinstruction jump address if the microprogram sequencing determines the program should jump. The jmplogic control field is used to determine the microinstruction jumping behavior, whether to never jump, to jump on a negative subtraction, or to always jump. The maddr control field is used to determine what address to use for memory address, either the aAddr, the bAddr, or the PC. The mread control field determines if memory should be read from. The mwrite control field determines if memory should be written to. The xsel control field determines what input to assign to the first input of the subtracter. The ysel control field determines what input to assign to the second input of the subtracter. The zdest control field determines what register to assign the subtracter output to.

Figure A.2: Microinstruction Format for Example SBN Datapath
<table>
<thead>
<tr>
<th>Control Field</th>
<th>Control Field Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmpaddr</td>
<td>DEFAULT=0</td>
</tr>
<tr>
<td></td>
<td>jmpalways=2</td>
</tr>
<tr>
<td></td>
<td>jmpnext=0</td>
</tr>
<tr>
<td></td>
<td>jmponneg=1</td>
</tr>
<tr>
<td>jmplogic</td>
<td>DEFAULT=0</td>
</tr>
<tr>
<td></td>
<td>aaddrtomaddr=1</td>
</tr>
<tr>
<td></td>
<td>baddrtomaddr=2</td>
</tr>
<tr>
<td></td>
<td>pctomaddr=0</td>
</tr>
<tr>
<td>maddr</td>
<td>DEFAULT=0</td>
</tr>
<tr>
<td></td>
<td>aaddrtomaddr=1</td>
</tr>
<tr>
<td></td>
<td>baddrtomaddr=2</td>
</tr>
<tr>
<td></td>
<td>pctomaddr=0</td>
</tr>
<tr>
<td>mread</td>
<td>DEFAULT=0</td>
</tr>
<tr>
<td></td>
<td>noread=0</td>
</tr>
<tr>
<td></td>
<td>read=1</td>
</tr>
<tr>
<td>mwrite</td>
<td>DEFAULT=0</td>
</tr>
<tr>
<td></td>
<td>nowrite=0</td>
</tr>
<tr>
<td></td>
<td>write=1</td>
</tr>
<tr>
<td>xsel</td>
<td>DEFAULT=0</td>
</tr>
<tr>
<td></td>
<td>atox=2</td>
</tr>
<tr>
<td></td>
<td>btox=3</td>
</tr>
<tr>
<td></td>
<td>memtox=0</td>
</tr>
<tr>
<td></td>
<td>pctox=1</td>
</tr>
<tr>
<td>ysel</td>
<td>DEFAULT=0</td>
</tr>
<tr>
<td></td>
<td>atoy=0</td>
</tr>
<tr>
<td></td>
<td>btoy=1</td>
</tr>
<tr>
<td></td>
<td>neg1toy=3</td>
</tr>
<tr>
<td></td>
<td>zerotoy=2</td>
</tr>
<tr>
<td>zdest</td>
<td>DEFAULT=2</td>
</tr>
<tr>
<td></td>
<td>ztoa=4</td>
</tr>
<tr>
<td></td>
<td>ztoaaddr=0</td>
</tr>
<tr>
<td></td>
<td>ztob=5</td>
</tr>
<tr>
<td></td>
<td>ztobaddr=1</td>
</tr>
<tr>
<td></td>
<td>ztmem=2</td>
</tr>
<tr>
<td></td>
<td>ztopc=3</td>
</tr>
</tbody>
</table>

Table A.1: Control Fields and Their Mnemonics for SBN Datapath
A.2 Microprogram To Implement ISA

The following microprogram implements the SBN ISA for the datapath shown in Figure A.1:

```
# implements single instruction: sbn addr0 addr1 addrBranch

# mem[addr0] = mem[addr0] - mem[addr1]
# if negative
#    pc = addrBranch
# else
#    pc = pc + 3

# 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, ztomem, aaddrtomaddr, write,
    jmponneg, jmpaddr=negbranch;
```
# a-b is positive
# pc++
pctox, neg1toy, ztopc, jmpalways, jmpaddr=isa;

# a-b is negative
# pc = m[pc+2]

negbranch: memtox, zerooy, ztopc, jmpalways, jmpaddr=isa;
A.3 ISA Code To Implement Multiply

Now that we have a datapath and microcode, a program must be specified to do something useful. The multiply algorithm is a simple operation that one might use but requires several instructions in a loop to implement. The following memory file was generated by a simple assembler. The original sbn instructions are commented followed by the three memory addresses representing the assembled instruction. At the end of the program is the data section that has constants and memory mapped registers used by the program.

```plaintext
# sbn computer multiply
# const a;
# c = 0;
# while ( --b > 0 ) {
#   c += a;
# }

# sbn negone, one, .+1 # make negone actually negone
# addr:0
24 21 3
# sbn c, c, .+1 # c = 0
# addr:3
20 20 6
# loop: sbn b, one, pool # b--, jmp to pool if b<0
# addr:6
1f 21 1b
# sbn temp, temp, .+1 # temp = 0
# addr:9
23 23 c
# sbn temp, c, .+1 # temp = 0-c
# addr:c
23 20 f
# sbn temp, a, .+1 # temp = 0-c-a
# addr:f
```
23 1e 12
# sbn c, c, .+1 # c = 0
# addr:12
20 20 15
# sbn c, temp, .+1 # c = 0-(0-c-a) => c+a
# addr:15
20 23 18
# sbn negone, zero, loop # jmp loop
# addr:18
24 22 6
# pool: sbn negone, zero, .-2 # loop forever
# addr:1b
24 22 1b

# 23*17 == 0x187

# a: def 23
# addr:1e
17
# b: def 17
# addr:1f
11
# c: def 13
# addr:20
d
# one: def 1
# addr:21
1
# zero: def 0
# addr:22
0
# temp: def 7
# addr:23
7
# will make this -1 in intialization
In Chapter 2, the memory resulting from the above ISA code is shown in Figure 2.2 in Section 2.1. This shows the memory addresses used for the instruction at memory addresses 0x0018.
A.4 Microprogram To Implement Multiply

By looking at the microcode, the cycles per instruction (CPI) can be calculated to be 9. Then looking at the ISA code for the multiply algorithm, the number of instructions per loop iteration is 7. This means the clock cycles of this datapath per loop iteration is 63. However, if the multiply algorithm is implemented in microcode instead, the number of clock cycles per loop iteration becomes 2 as shown in the following microprogram. This shows how custom microcode can dramatically increase the performance of operations that are not already directly supported by the microprogram.

```
# memory:
# a: addr 0
# b: addr 1
# c: addr 2

# aaddr = 0
atox, atoy, ztoaaddr;
# pc = 0, read mem[aaddr]
atox, atoy, ztopc, aaddrtomaddr, read;
# a = mem[aaddr] = m[0] = (a)
memtox, zerotoy, ztoa;
# a = pc-a = 0-(a)
pctox, atoy, ztoa;
# baddr = 1;
btox, negltoy, ztobaddr;
# b = 1, read mem[baddr]
pctox, negltoy, ztob, baddrtomaddr, read;
# pc = mem[baddr] = m[1] = (b)
memtox, zerotoy, ztopc;
# aaddr = 2
btox, negltoy, ztoaaddr;
# mem[aaddr] = 0
atox, atoy, ztOMEM, aaddrtomaddr, write;
```
# pc-- = (b)--, read m[aaddr] = m[2] = (c)

loop: pctox, btoy, ztopc, aaddrtomaddr, read,
    jmponneg, jmpaddr=done;
# (c) = (c)-(0-(a)) = (c)+(-a)

memtox, atoy, ztomem, aaddrtomaddr, write,
    jmpalways, jmpaddr=loop;

done: jmpalways, jmpaddr=done;
A.5 Datapath File

The following code is the datapath file for the SBN architecture. Since the vertices, signals, edges and messages section are very long and repetitive, only a few of the first and last items of each section are displayed with “...” in between to indicate that there was more items.

```
# MicroTiger Datapath File
# Version: 0.90.0017
# $

dss16
vtx$131$69
vtx$11D$5F
...
vtx$13B$69
vtx$109$A5
fld4 jmpaddr$0
fld2 jmplogic$0 DEFAULT=0 jmpalways=2 jmpnext=0 jmpnonnull=1
fld2 maddr$0 DEFAULT=0 aaddrtomaddr=1 baddrtomaddr=2 pctomaddr=0
fld1 mread$0 DEFAULT=0 noread=0 read=1
fld1 mwrite$0 DEFAULT=0 nowrite=0 write=1
fld2 xsel$0 DEFAULT=0 atox=2 btox=3 memtox=0 pctox=1
fld2 ysel$0 DEFAULT=0 atoi=0 btoy=1 neg1toy=3 zerotoy=2
fld3 zdest$0 DEFAULT=2 ztoa=4 ztoaddr=0 zto=5 ztoaddr=1 ztop=2 ztopaddr=3
sig16 __00000000
sig16 __00000001
...
sig4 mpc
sig16 whatever2
sig16 whatever4
edg whatever2 whatever2$7D$37$34$35
edg whatever2 whatever2$7D$23$35$36
...
```
"if ((jmplogic == 1 && input0[negative]) || jmplogic == 2) {
    jmpaddr;
} else {
    mpc+1;
}
"$181$8C$1A$1B
ram16$10000$1"factorial.isa" __00000006 whatever4
 __00000008 mread mwrite$B9$50$1C$1D$1E
reg0 __00000004 __00000005$B9$96$1F$20
reg0 whatever2 __00000001$B9$6E$21$22
reg0 __00000002 __00000003$B9$82$23$24
reg0 __0000007C __00000090$4B$50$25$26
reg0 __0000007E __0000008F$4B$3C$27$28
Appendix B

Various Datapaths

This appendix reviews various datapaths designed using the implemented simulator. They are ordered by their performance starting with the simpler and faster datapaths.

B.1 Tiny Datapath

The tiny datapath utilizes only a few components and was the first datapath implemented in the simulator.
B.2 LMMS Datapath

Figure B.2: Sample Datapath Screenshot: lmms.dp

The LMMS datapath is the same datapath as the one implemented in the Little Man Microcode Simulator [3].
B.3 Mic-1 Datapath

Figure B.3: Sample Datapath Screenshot: mic1.dp

The Mic-1 datapath is the same datapath as the Mic-1 microprogrammable architecture that Tanenbaum uses in his *Structured Computer Organization* book. This datapath is implemented by many simulators since the book is used by many computer organization classes including the MX Simulator [5].
The primer datapath uses a few components and a very intensive custom component expression to determine if numbers are prime or not. The following expression is used within the custom component to determine if input signal is a prime number:

```c
if ( input0[0] == 0 ) {
    input0 == 2;
} else if ( input0 < 6 ) {
    input0 == 3 || input0 == 5;
} else {
    # determine max 1 bit
    y = 2;
    for ( x = 4; x < 8; x = x+1 ) {
        if ( input0[x] == 1 ) {
            y = x;
        }
    }
    # determine number slightly above square root
    y = input0 >> (y>>1);
    x = 3;
```
z = (input0 % x != 0);
while ( z && x < y ) {
    x = x + 2;
    z = (input0 % x != 0);
}
z;
}
B.5 MythSim with One Write-Back Datapath

The MythSim with one write-back is a simplified version of the datapath developed for the MythSim tool [26]. The simplification is that only write back is allowed per cycle which allows the register file to be used. This simplification does not interfere with the microprogram examples provided with the MythSim tool since they do not utilize the multiple write-back feature.
B.6 MythSim Datapath

The MythSim datapath is the same as the datapath developed for the MythSim tool [26].
B.7 AMISS Datapath

The AMISS datapath is the datapath used by the Rochester Institute of Technology Computer Organization that utilizes the AMISS tool [19].

Figure B.7: Sample Datapath Screenshot: amiss.dp
Appendix C

User Documentation

C.1 Overview

MicroTiger is a graphical microcode simulator with a reconfigurable datapath. The datapath editor window, the main application window, allows for datapath creation and simulation. In addition, several other dialog windows are provided to aid in creating the datapath. The microprogram editor window allows for microprogram and aids in simulation. The ISA code is externally edited and can be viewed in the ISA code viewer window. The simulation controls are available in the main application window. The expression language is available for ALUs and custom components to customize the evaluation of these components.

C.2 Creating Datapaths

The datapath editor window is the main application window and is always available. The editor resembles other circuit editors in that the user can move components and wires, and configure the properties of components. The editor provides a set of common datapath elements that are easy to configure for many different purposes.

The File menu allows the datapath to be loaded and saved. The File menu also allows the microprogram and ISA code to be loaded. The File menu also allows the application to be exited.
The *Edit* menu allows highlighted components to be cleared, all the components to be selected, the default signal size for the datapath to be edited, and the RAM to use for the ISA code to be specified.

The *View* menu allows the microprogram editor window, ISA code viewer window, signal list window, control properties window, and datapath macros window to be opened. The *View* menu also allows the toolbar, statusbar, and datapath grid to toggled. By default, the toolbar and statusbar are enabled and the grid is disabled. The *View* menu also allows the datapath to be zoomed between 30% and 150% with increments of 10%. The default zoom level is 100%. The zoom can also be changed using the keyboard keys. The '-' key is used to decrement the zoom level and '+' key is used to increment the zoom level. The *View* menu also allows access to the datapath properties window that reports information about the number of components, signals, control fields, and vertices.

The *Draw* menu allows several components to be added including Adders, Adder, Alus, Comparators, Custom components, Demultiplexers (Demuxes), Multiplexers (Muxes), Pipeline Registers, RAMs, Registers, Register Files, Sign Extenders, and Subtracters. The *Draw* menu also provides a message component to comment the datapath. The datapath components and wires can be moved by using the left mouse button to drag the components to new locations and tries to intelligently move any attached wires with the component. Components can be deleted by using the delete key or *Edit* menu’s *Clear* item. Both datapath components and messages can be configured by right clicking on the component. For messages, the message text can be configured. For other components, the signals can be modified (added and removed if there are variable number of signals for the component) and the expression for configurable components can be modified. The expression language for ALUs and custom components is described later in the *Expression Language* section.

The *Draw* menu also provides a wire drawing mode. The wire drawing mode waits for a first left mouse click which specifies the first end point of the wire. Until the second left mouse click is encountered, a dotted line is drawn between the first wire end point and the current mouse location. The wire can be drawn when the second left mouse click is
encountered. The wire mode then continues waiting for another first left mouse click to specify another wire.

The Simulate menu provides several simulation controls and are described in the Simulating It All section.

The Help menu provides this user documentation, a number converter for different numeric bases, and information about the application.

C.3 Creating MicroPrograms

The microprogram editor provides a integrated text editor for writing and compiling the microprogram for the simulator. When simulating, the microprogram editor highlights the current operation in the microprogram making debug easier. Also, breakpoints can be defined to further aid in debugging. The format of the microprogram language is describe in the MicroProgram Language section.

The File menu allows the microprogram to loaded and saved. The File menu also allows the microprogram editor to be closed. Also, the microprogram can be loaded from the main application’s File menu.

The Edit menu allows control field mnemonics to be inserted from the control properties window and allows for standard text editing features.

The View menu allows the control properties window and the datapath macros window to be brought up. The View menu can also toggle the statusbar and the syntax highlighting during simulation.

The Build menu allows the microprogram to be compiled and present an error dialog when the microprogram has parsing problems.

The Breakpoints menu allows breakpoints to be set, breakpoints to be cleared, and all breakpoints to be cleared.

The Help menu provides this user documentation and information about the application.
C.3.1 Control Fields Specification

The control fields are created and modified using the control properties window. The control properties window is accessible through both the datapath editor’s (main window) and microprogram editor’s View menu. The control properties dialog allows control fields to be added, edited, and removed. Once a control field is selected, its properties can be edited. The editable properties include the signal length, the overflow control field to store the overflow of this control field (for more advanced microprogram formats), and the assigning of control field mnemonics. Also, there is a button to insert a selected control field mnemonic into the microprogram editor window.

C.3.2 Datapath Macros

The datapath macros window allows the macros to be defined in the datapath file and therefore are accessible by any microprogram using the datapath to reduce redundancy in microprogram macros. It provides a simple text editor to write the datapath macros. When either OK or Apply buttons are pressed, the datapath macros are compiled and any parsing errors are presented to the user and saved if compiled correctly. When either the OK or Cancel buttons are pressed, the window closes.

C.4 Simulating It All

The simulator is accessed through the Simulation menu of the datapath editor window and the optional toolbar in the datapath editor window. It provides different running options like stepping on different code levels, running to halt and resetting.

The Simulation menu items are only available to use when a valid datapath exists and a microprogram is compiled. The menu provides functions to reset, to step backwards, to step forward, to step to the last or next ISA instruction (using the isa label in the microprogram), to step to the next or last iteration that has the same MPC, to step to the next or last iteration with a user specified MPC, a facility to provide an animation of the simulation.
Also, the *Simulation* menu provides the ability to calculate the cycles per instruction (CPI) of the current simulation run. The CPI calculation uses the microinstruction labeled *isa* to determine the first ISA iteration, last ISA iteration and the number of ISA iterations executed. The CPI is calculated using the following equation:

\[
CPI = \frac{\text{last isa iteration} - \text{first isa iteration}}{\text{number of isa iterations} - 1}
\]  

(C.1)

### C.5 ISA Language

The ISA language is very simple. The ISA language is used to input ISA files for the RAM and Register File and is used to save the state of memory. When saving the memory state, the simple, Intel hex and S-Record formats can be used for the output file.

#### C.5.1 Comments

Comments can start with either (C++ style) or # (Perl style). Comments end at the end of line. These are the same commenting style as the microprogram language described in the *Microprogram Language* section and the expression language described in *Expression Language*. Additionally a semi-colon can specify a comment (this was to provide compatibility with the AMISS simulator memory file format). For example:

```markdown
# this is a comment just like those in Perl
// this is a comment just like those in C++
; this is comment just like those in AMISS
```

#### C.5.2 Simple Numbers

The simple format accepts only hex numbers and each number corresponds to a single memory location regardless of the size of the hex number. When using this format, memory values start at memory address zero and increment the memory address for each memory value. For example:
# this store in the first and second memory addresses
4D A0
# next value stores in third memory address
FF
# this long number will be truncated to the element size
74BA897894CAB347892A80001A

C.5.3 Intel Hex Format

The Intel hex format is also an acceptable input for ISA language. See other documentation on the actual Intel hex format specification. An example Intel hex format input is:

```
# sbn primes.isa => intel hex format
:20000006D006B0003006606600660606606060009006606D000C06A006A000F006A002B
:2000200660012006700670015067006A00180067006B001B00670068001E0067006B00CD
:200040004E006A006A024006A006700270068006800200068006A002D0068006D003006D4
:200060068006D0033006A006A0036006A0066003900690069003C0069006A003F006900DC
:200080068004B0069006B0090069006D0048006D006C003F006D006C001E006A006A00CF
:2000A00651006A00660054006E0050006E006A0054006D005D0055006D00600026
:2000C0057006D00630066006C000900170011000D00590007000100000000000FF02000080
:0000001FF
```

Figure C.1: ISA Code Example Using Intel Hex Format

C.5.4 SRecord Hex Format

The SRecord format is also an acceptable input for ISA language. See other documentation on the actual SRecord format specification. An example SRecord format input is:

```
# sbn primes.isa => srecord hex format
S32500000006D006B000300660660066006606060009006606D000C06A006A000F006A0025
S32500000200660012006700670015067006A00180067006B001B00670068001E0067006B00C7
S325000000404E006A006A0024006A0067002700680068002B0068006A002D0068006D0030005E
S3250000006C8006D0033006A006A0036006A0066003900690069003C0069006A003F006900DG
S3250000008068004B006B0006006D000D0048006D006C003F006D006C001E006A006A00C9
S3250000006A51006A00660054006E0050006E006A0054006D005D0055006D00600020
S325000000C57006D0063006D006C000900170011000D00590007000100000000000FF0200007A
S9030000FC
```

Figure C.2: ISA Code Example Using SRecord Format
C.6  Microprogram Language

C.6.1  Comments

Comments can start with either (C++ style) or # (Perl style). Comments end at the end of line. These are the same commenting style as the expression language described in the Expression Language section. For example:

```
#  this is a comment just like those in Perl
//  this is a comment just like those in C++
```

C.6.2  Microinstruction Format

To specify the control field values to assign, either the control field mnemonic can be used, or the control field can be named and be assigned a numeric value or microinstruction address. The values are separated by commas and each microinstruction ends with a semi-colon.

Microinstructions can optionally be labeled by prepending the microinstruction with the label name and a colon. Additionally a microinstruction can be labeled with a number to indicate the microinstruction address of the instruction. If the address is larger than the current microinstruction address, empty microinstructions are inserted between the previous microinstruction and the current microinstruction to place the current microinstruction at the specified microinstruction address.

An example of this microinstruction format is:

```
# negbranch label
# uses mnemonics: memtox, zerotoy, ztopc, jmpalways
# assigns microinstruction address isa to jmpaddr
negbranch: memtox, zerotoy, ztopc,
            jmpalways, jmpaddr=isa;
```
C.6.3 Simple Microinstruction Format

A simpler microinstruction format is one that is a sequence of numeric values that are ordered by the control field names sorted in alphabetic order. The values are only separated by whitespace and the microinstruction ends with a semi-colon. These microinstructions use the same microinstruction labels as the regular microinstruction format (described above). This allows both the simple microinstruction format and the regular microinstruction format to be used within the same microprogram.

```plaintext
# the same microinstruction as the last example
# just uses the simple microinstruction format instead
negbranch: 1 2 0 0 0 0 2 3;
```

C.6.4 Macro Format

The macro format is the same format as the microinstruction (without the microinstruction label) and are appended with `@macro` and the macro name. Macros can then be used in exactly the same manner as a control field mnemonic but provide several control fields assignation in a single phrase.

```plaintext
# defines a macro as the same microinstruction as before
@macro mymicro memtox, zerotoy, ztopc, jmpalways, jmpaddr=isa;

# use the macro to implement a microinstruction
negbranch: mymicro;
```

C.7 Expression Language

The expression language is mainly based on the C++ programming language with modifications to simplify certain bit level operations such as bit indexing and bit rotation.
C.7.1 Comments

Comments can start with either (C++ style) or # (Perl style). Comments end at the end of line. These are the same commenting style as the expression language described in the MicroProgram Language section. For example:

```
# this is a comment just like those in Perl
// this is a comment just like those in C++
```

C.7.2 Defining Constants

By default, constants use the default signal size of the architecture. If the constant requires more bits than the default signal size, then the signal is created using as many bits is required to contain the unsigned integer being represented. Also, the number of bits can be specified by appending the size in bits in parenthesis. For example:

```
# this defines a 20 bit constant with the value 27
27(20)

# this defines a 1 bit constant
# so the value becomes 1 rather than 255
255(1)
```

Different bases can be specified based on how different bases can be specified in C++ and some assemblers. The binary base is specified inserting “0b” or “0B” before the constant. The decimal base can also be specified by inserting “0d” or “0D” before the constant. The octal base is specified inserting “0o” or “0O” before the constant. The hexadecimal base is specified inserting “0x”, “0X”, or “$” before the constant. Additionally, a number string can be specified using B, D, O, or H to specify the base to interpret the string as. For example:
C.7.3 Signal Names

Signal names must start with a letter or underscore and can contain any combination of letters, numbers, and underscores. If a name is not defined when used in an expression, it is cast to the default signal size. For example:

```vhdl
# this uses the signal aluOutput
aluOutput

# even really long names are acceptable
reallyLongSignalNameWithNumbersandUnderscoresLike_123

# make a local variable with a certain size
tempIndex(8)
```
C.7.4 Operations

All expressions use a syntax similar to that used by many languages like C++, Java and Perl.

All signals are by default treated as unsigned integers and can be represented by as many bits (such as using a single signal to represent an all the values for memory). Additionally signals can be treated as a signed integer, floating point number or a double-precision floating point number (using the machine’s float and double C++ types) by casting like in C++.

Most operations treat the signal as unsigned array of bits, except the comparison operators and mathematical operators. If operation has parameters of unequal types, the double type has highest priority, then float, signed and finally unsigned.

The order of operations are shown in Table (C.1). All the operators that are in the C++ language use the same order that the C++ language uses for the operators. Added operators go in the most obvious locations relative to the similarity to C++ operators. The operators located near the top of the table have a higher precedence, meaning that they are evaluated first. For example in the expression “b = 1 + 2”, the assignment operator is evaluated first so that we have “b” being assigned “1+2” which needs to be evaluated using the order of operations.

Examples of various operations:

```plaintext
# simple addition
12 + 28;

# lots of operations
1 + 3 < 5 && 2 * 6 == 12 - 0 || (23 % 6) << 5 > 37;

# casting
(signed)89 == (unsigned)--89;

# bit and flag retrieval
aluOutput[carry] . aluOutput[negative] . aluOutput[1:0];
```
Table C.1: Order of Operations With Higher Precedence At The Top

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>combine expression together</td>
</tr>
<tr>
<td>=</td>
<td>assignment operator</td>
</tr>
<tr>
<td>?:</td>
<td>ternary operator for conditions</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^^</td>
<td>logical exclusive or</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>boolean exclusive or</td>
</tr>
<tr>
<td>&amp;</td>
<td>boolean and</td>
</tr>
<tr>
<td>==</td>
<td>equality</td>
</tr>
<tr>
<td>!=</td>
<td>inequality</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
</tr>
<tr>
<td>&lt;&lt;&lt;</td>
<td>rotate left</td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td>rotate right</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>.</td>
<td>catenation</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td>-</td>
<td>negation</td>
</tr>
<tr>
<td>!</td>
<td>logical not</td>
</tr>
<tr>
<td>~</td>
<td>boolean not</td>
</tr>
<tr>
<td>(unsigned)</td>
<td>cast to unsigned</td>
</tr>
<tr>
<td>(signed)</td>
<td>cast to signed</td>
</tr>
<tr>
<td>(float)</td>
<td>cast to float</td>
</tr>
<tr>
<td>(double)</td>
<td>cast to double</td>
</tr>
<tr>
<td>[x]</td>
<td>get the x bit</td>
</tr>
<tr>
<td>[x:y]</td>
<td>get the bits from x to y</td>
</tr>
<tr>
<td>[carry]</td>
<td>return 1 if the signal has a carry</td>
</tr>
<tr>
<td>[overflow]</td>
<td>return 1 if the signal has an overflow</td>
</tr>
<tr>
<td>[negative]</td>
<td>return 1 if the signal is negative</td>
</tr>
<tr>
<td>[zero]</td>
<td>return 1 if the signal is zero</td>
</tr>
</tbody>
</table>
Assignment

On the left side of the assignment operator, “=”, must either me a name, a name[x] or name[x:y]. The name must be a valid signal name. The name[x] assignment assigns a particular bit in the signal. The name[x:y] assignment assigns a range of bits in the signal.

```
# simple assignment
output = 32;

# assign a bit
setMy3rdBit[2] = 1;

# assign a range of bits
setMy2ndByte[8:15] = 0x72;
```

C.7.5 Programming Language Statements

For, While, and Do

Provides the three loop types with the same syntax as C++. The **for** keyword provides a starting expression, a looping condition (which is also tested before the loop begins), and an expression to evaluate at the end of each loop. The **while** keyword provides a looping condition which is also tested before entering the loop. The **do** keyword provides a looping condition which is not tested until one iteration of the loop is completed. Examples:

```
# for loop
for ( index = 0; index < 5; index++ ) {
    output[index*5] = input[index];
}

# while loop
while ( counter > 0 ) {
```
output[counter] = counter % 2;
counter--;
}

# do loop
do {
    output = output >> 1;
} while ( output != 1 );

If and Else

Provides a conditional statement with the same syntax as C++ using the if and else keywords.

    # if example
    if ( output[carry] ) {
        error = 1;
    } else {
        error = 0;
    }

Case

The case keyword is provided to make it easier to execute several different blocks of code depending on a certain value. Any valid arithmetic expression can be tested. Each expression will have one or more numbers associated with it that will be executed if any of those numbers are equal to the expression specified. The numbers are followed by colons. Expressions can be any valid expression (including block statements, lists of expressions with semi colons or just simple expressions). At the end of the case statement is an optional default expression that gets executed if there are no matching numbers for the expression.

It is similar in function to the switch statement in C++ except there are no break statements and execution does not go into the next block. Additionally, the case statement
returns the value of the last expression evaluated allowing the case statement to assign a value to a component output.

```c
# case example
# any valid arithmetic expression can be used
case ( aluOutput-lastAluOutput ) {
  0:
    # case are specified by numbers followed by colons
    0;
  1: 2: 3:
    # multiple values can be specified
    1;
  default:
    # complex statements can be used,
    # and case statements can be nested
case ( aluOutput ) {
  0:
    2;
  1: 2: 3:
    3;
  default:
    4;
}
}
```

## C.8 About

This tool was developed by Brian G VanBuren under a thesis at the Rochester Institute of Technology entitled Graphical Microcode Simulator with a Reconfigurable Datapath. Dr. Muhammad Shaaban was the primary thesis advisor with Dr. Roy Czernikowski and Dr. Greg P. Semeraro as secondary advisors. The goal was to create a versatile microcode simulator with a side goal of replacing the existing microcode simulator used for RIT’s
Computer Organization class.
Appendix D

CD Contents

The following items can be found on the included CD:

- Thesis Proposal
- Thesis Document
- Executables
- Example Datapaths, Microprograms and ISA Code
  - Register File Example
  - Tiny Example
  - Little Man Example
  - Subtract And Branch If Negative
  - Prime Number Determiner Example
  - Tannenbaum’s Mic-1 Architecture
  - MythSim
  - Original RIT EECC-550 Microprogramming Project
- User documentation
- Developer documentation
- Source code