September 2017

Improving Microcontroller and Computer Architecture Education through Software Simulation

Kevin Brightwell  
*The University of Western Ontario*

Supervisor  
McIsaac, Kenneth  
*The University of Western Ontario*

Graduate Program in Electrical and Computer Engineering

A thesis submitted in partial fulfillment of the requirements for the degree in Master of Engineering Science

© Kevin Brightwell 2017

Follow this and additional works at: [https://ir.lib.uwo.ca/etd](https://ir.lib.uwo.ca/etd)

教えは学びの一部として、コンピューターサーバー、デジタル回路、および、エンジニアリング教育の共通項目を追跡します。

Recommended Citation
[https://ir.lib.uwo.ca/etd/4886](https://ir.lib.uwo.ca/etd/4886)
Abstract

In this thesis, we aim to improve the outcomes of students learning Computer Architecture and Embedded Systems topics within Software and Computer Engineering programs. We develop a simulation of processors that attempts to improve the visibility of hardware within the simulation environment and replace existing solutions in use within the classroom. We designate a series of requirements of a successful simulation suite based on current state-of-the-art simulations within literature. Provided these requirements, we build a quantitative rating of the same set of simulations. Additionally, we rate our previously implemented tool, hc12sim, with current solutions. Using the gaps in implementations from our state-of-the-art survey, we develop two solutions. First, we developed a web-based solution using the Scala.js compiler for Scala with an event-driven simulation engine through Akka. This Scala model implements a VHDL-like DSL for instruction control definition. Next we propose tools for developing cross-platform native applications through a project-based build system within CMake and a continuous integration pipeline using Vagrant, Oracle VirtualBox and Jenkins. Lastly, we propose a configuration-driven processor simulation built from the original hc12sim project that utilizes a Lua-based scripting interface for processor configuration. While we considered other high-level languages, Lua best fit our requirements allowing students to use a modern high-level programming language for processor configuration. Instruction controls are defined through Lua functions using high-level constructs that implicitly trigger low-level simulation events. Lastly, we conclude with suggestions for building a new solution that would better meet requirements set forth in our research question building from successful aspects from this work.

Keywords: Embedded systems, computer architecture, computer architecture simulation, pedagogy, cross-platform application development
Acknowledgements

I would like to acknowledge my advisor Dr. Kenneth McIsaac for his continued support and mentorship throughout my undergraduate and graduate works. Thank you for taking the time to debug, construct, and destruct ideas from all angles. Your direction has allowed me to grow as a research, engineer and academic.

I would like to thank my peers in Dr. McIsaac’s lab for providing laughs and support over the course of this work. I would like to extend the same thanks to those in Dr. Xianbin Wang and Dr. Roy Eagleson’s labs for being supportive peers through forced geographic association.

I would like to thank my father and step mother, John and Amy Brightwell, and my mother and step father, Kyra Hanson and Wilf Wityshyn, for listening to my complaints, failures and successes throughout this work and never batting an eye of support. I would like to thank Mr. Brad de Vlugt, Mr. Kyle Fricke, and Ms. Trinette Wright for their continued support over the course of this work providing friendship and a talking post repeatedly. Lastly, I would like to thank my boyfriend Eric Pattara for his love, continuous support and delightful immaturity when I needed that most.

To any and all of my colleagues, peers, family and friends thank you for your support.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract</strong></td>
<td>i</td>
</tr>
<tr>
<td><strong>Acknowledgements</strong></td>
<td>ii</td>
</tr>
<tr>
<td><strong>List of Figures</strong></td>
<td>vi</td>
</tr>
<tr>
<td><strong>List of Tables</strong></td>
<td>viii</td>
</tr>
<tr>
<td><strong>List of Listings</strong></td>
<td>ix</td>
</tr>
<tr>
<td><strong>List of Definitions</strong></td>
<td>xi</td>
</tr>
<tr>
<td><strong>List of Appendices</strong></td>
<td>xiii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Research Question</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Problem Statement</td>
<td>9</td>
</tr>
<tr>
<td>1.3.1 Software versus Hardware</td>
<td>14</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>14</td>
</tr>
<tr>
<td><strong>2 Survey of other and previously completed works</strong></td>
<td>17</td>
</tr>
<tr>
<td>2.1 Previous Work: hc12sim</td>
<td>17</td>
</tr>
<tr>
<td>2.1.1 Software Design</td>
<td>18</td>
</tr>
<tr>
<td>Instructions</td>
<td>19</td>
</tr>
<tr>
<td>Hardware Simulation</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Evaluation of existing simulation technologies</td>
<td>23</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>ECE 3375 - Course Sentiment Analysis Results [24], [25]</td>
</tr>
<tr>
<td>1.2</td>
<td>WinIDE12Z Integrated Development Environment [26]</td>
</tr>
<tr>
<td>1.3</td>
<td>Freescale M68HC12 development board built by Western Engineering Electronics Shop for use in ECE 3375</td>
</tr>
<tr>
<td>1.4</td>
<td>StackOverflow 2016 developer survey results for desktop operating systems [36, Sec. VIII. Desktop Operating System]</td>
</tr>
<tr>
<td>2.1</td>
<td>hc12ide Integrated Development Environment showing a student error in an incorrect specification of an instruction mnemonic</td>
</tr>
<tr>
<td>2.2</td>
<td>hc12sim Instruction Generator showing the meta information for addressing modes, execution, and output</td>
</tr>
<tr>
<td>2.3</td>
<td>Block diagram of TinyCPU showing the internal architecture [29, p. 869]</td>
</tr>
<tr>
<td>2.4</td>
<td>State chart of TinyCPU execution path [29, p. 869]</td>
</tr>
<tr>
<td>2.5</td>
<td>TinyCSE execution state diagram modified from Figure 2.4 to include interrupts (broken lines show changes from TinyCPU [29]) [28, p. 640]</td>
</tr>
<tr>
<td>2.6</td>
<td>CPU Sim 4.0.0 IDE during debugging with Assembly Editor open [52]</td>
</tr>
<tr>
<td>2.7</td>
<td>CPU Sim Machine Instruction configuration interface showing combination of fields to create a microcode and an assembly instruction</td>
</tr>
<tr>
<td>2.8</td>
<td>CPU Sim 4.0.0 IDE module specification dialog showing modification of Transfer Register to Register Array microinstruction [52]</td>
</tr>
<tr>
<td>2.9</td>
<td>Emubuilder86 Control Builder: State machine definition [56, p. 325]</td>
</tr>
<tr>
<td>3.1</td>
<td>Graphical representation of three Actors sending bi-directional messages [70]</td>
</tr>
<tr>
<td>5.1</td>
<td>URISC hardware architecture with microcodes [6]</td>
</tr>
</tbody>
</table>
5.2 Control unit for the 16-bit URISC architecture from [6, p. 331]... 111
# List of Tables

1.1 ECE 3375 Course “Level of enthusiasm before taking the course” as reported by undergraduate students following course completion [24], [25] . . . . . . . . 5

2.1 Qualitative requirement rating of hc12sim [40] . . . . . . . . . . . . . . . . . . 25

2.2 Qualitative requirement rating of ShelbySim [35], [45] . . . . . . . . . . . . . 27

2.3 Qualitative requirement rating of EDCOMP [30] . . . . . . . . . . . . . . . . . 29

2.4 Qualitative requirement rating of p88110 [31] . . . . . . . . . . . . . . . . . 31

2.5 Qualitative requirement rating of EASE [34] . . . . . . . . . . . . . . . . . . 32

2.6 Qualitative requirement rating of TinyCPU/TinyCSE [28], [29] . . . . . . . . 35

2.7 Qualitative requirement rating of CPU Sim [39] . . . . . . . . . . . . . . . . . 40

2.8 Qualitative requirement rating of Emumaker86 [56] . . . . . . . . . . . . . . 42

2.9 Summary of Evaluation for Simulators . . . . . . . . . . . . . . . . . . . . . . 43

3.1 Summary of requirement matching for procsim.scala . . . . . . . . . . . . . . . 58

5.1 Summary of requirement matching for procsim . . . . . . . . . . . . . . . . . . 113
List of Listings

2.1 Sample of the JSON output used to generate code within hc12sim. BEQ performs a branch if two values were equal. ........................................... 21

3.1 An example DSL source written in Scala to emulate BASIC called “Baysick” [1] 47
3.2 Normal definition of Instruction. ..................................................... 52
3.3 Reified definition of Instruction. .................................................... 52
3.4 Clock pulse message for Akka ....................................................... 54
3.5 Actor that listens for ClockPulse messages and prints the time. .......... 55
4.1 Vagrant file that describes a Ubuntu 14.04 box with modern GCC 5.X and LLVM Clang 3.7 compilers installed. ............................................. 65
4.2 CMake script showing how two test suites, time and encoding, are defined as part of a project, core ................................................................. 71
4.3 Test output from CTest [2] from Microsoft Visual Studio Community 2015 for procsim showing organization provided by CMake configuration. ......... 75

5.1 Example of exposing a C++ class to Python [3]. ............................... 79
5.2 Table used to describe project information for the Lua.org site [4] .......... 84
5.3 class player that holds two fields, one for hitpoints (hp) and one for bullets a player has (adapted from [5]). ....................................................... 87
5.4 Utilize the class player within a Lua script (adapted from [5]). ............ 88
5.5 Required bindings to allow utilization of class player from Lua (adapted from [5]). ................................................................. 89
5.6 Configuration of Figure 5.1 for the URISC processor [6]. .................. 93
5.7 (Continued) Configuration of Figure 5.1 for the URISC processor [6]. .... 94
5.8 Clock configuration for the processors main clock (cut from Listing 5.6). . . 96
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9</td>
<td>Register configuration for the program counter (cut from Listing 5.6)</td>
</tr>
<tr>
<td>5.10</td>
<td>Memory unit configuration for the program counter (cut from Listing 5.6)</td>
</tr>
<tr>
<td>5.11</td>
<td>Processor utilizing memory mapping functionality.</td>
</tr>
<tr>
<td>5.12</td>
<td>Encoding of the opcode and operands of the SUBLEQ Instruction (cut from Listing 5.6)</td>
</tr>
<tr>
<td>5.13</td>
<td>Execution definition for the SUBLEQ Instruction (cut from Listing 5.6)</td>
</tr>
<tr>
<td>5.14</td>
<td>Local variable definitions within an Instruction::exec() method.</td>
</tr>
<tr>
<td>A.1</td>
<td>Source for procsim/memory/MemoryUnit.hpp</td>
</tr>
<tr>
<td>A.2</td>
<td>Source for procsim/memory/Register.hpp</td>
</tr>
<tr>
<td>A.3</td>
<td>Source for procsim/time/Clock.hpp</td>
</tr>
<tr>
<td>A.4</td>
<td>Source for procsim/time/AsyncTimerReceiver.hpp</td>
</tr>
<tr>
<td>A.5</td>
<td>Source for procsim/time/Timer.hpp</td>
</tr>
<tr>
<td>A.6</td>
<td>Source for procsim/encoding/Algorithm.hpp</td>
</tr>
<tr>
<td>A.7</td>
<td>Source for procsim/encoding/Code.hpp</td>
</tr>
<tr>
<td>A.8</td>
<td>Source for procsim/encoding/Operand.hpp</td>
</tr>
<tr>
<td>A.9</td>
<td>Source for procsim/core/Instruction.hpp</td>
</tr>
<tr>
<td>A.10</td>
<td>Source for procsim/core/Proc.hpp</td>
</tr>
</tbody>
</table>
List of Definitions

**Application Programming Interface (API)** A software layer between a developer that utilizes a library or program and the library or program itself. APIs are designed to hide implementation details and ease developer usage.

**Arithmetic Logic Unit (ALU)** In a computer system, ALUs are responsible for performing arithmetic operations on binary input such as addition, shifts, multiplication, division.

**Complex Instruction Set Computer (CISC)** A type of microprocessor architecture that utilizes larger, instructions aiming to reduce the software instructions required to complete a task [7], [8]. CISC architectures are used in most general applications based on Intel® x86 ISA [9].

**Domain-Specific Language (DSL)** A computer programming language that is specialized to a particular application space (domain). For example, HTML is a DSL used to layout websites but could not be used to program an embedded device. Similarly, C++ is an excellent language for creating applications but is very poor at describing layout for a webpage.

**Embedded Systems** Topics involving characteristics of embedded systems, techniques for embedded applications, parallel input and output, synchronous and asynchronous serial communication, interrupt handling, applications involving data acquisition, control, sensors, and actuators, implementation strategies for complex embedded systems [10] p. 118

**Computer Architectures** Topics involving computer organization and architecture. Including, but not limited to processor organization, instruction set architecture, memory system organization, performance, and interfacing fundamentals [10] p. 118

**Graphical User Interface (GUI)** Any interface based on rendering graphically, for example a window with buttons.
**Hardware Description Language (HDL)** A programming language used to describe the structure and behaviour of electronic circuits. These languages are used to synthesize simulated and physical circuit layout from the gate and register transfer level [11].

**Instruction Set Architecture (ISA)** The design of an interface between software and hardware designs in a computing environment. Including but not limited to assembly language and microcode design. Once realized, an ISA is referred to as an Instruction Set Implementation.

**Integrated Development Environment (IDE)** A tool used for developing hardware and software components that provides integrated tools for the current development context. For example, Xilinix ISE [12], JetBrains IntelliJ IDEA[1] or Eclipse[2].

**Just-in-Time Compiler (JIT)** A traditional compiler that operates at runtime to perform optimization on running code. JITs are typically found in interpreted languages such as on the JVM or in JavaScript. JITs vastly improve the runtime of the code they compile as they often utilize semantics such as code path tracing and known constant analysis that is not necessarily available at compile-time.

**Reduced Instruction Set Computer (RISC)** A type of microprocessor architecture that utilizes small, highly-optimized set of instructions used to reduce power and improve instruction performance [7], [8]. These architectures are commonly used in embedded, mobile (ARM), and high-performance server applications (IBM POWER).

**Reification** The process by which an abstraction (usually a String representation) of software is converted into an explicit object. For example, JavaScript's eval(script) compiles and executes the content of the script passed and any state changes within the script are applied to the global scope.

**Transpiler** A compiler that compiles one language to another. Typical transpilers convert one language to JavaScript to allow use in web applications. For example, Scala.js and TypeScript both compile their respective language to JavaScript.

---

List of Appendices

Appendix A Procsim headers ........................................ 132
Chapter 1

Introduction

Embedded systems and computer architectures are a critical part of both computer and software engineering undergraduate programs [10], [13]–[16]. Over time, it is expected that knowledge of embedded systems and computer architectures will be required given projections of rapid growth of positions in both systems software developers (+13%) and computer occupations (+12.5%) for 2020 [17]. At our institution, Western University, four courses cover embedded systems and computer architectures, ECE 3375 - Microprocessors and Microcomputers, ECE 3389 - Computer System Design, ECE 3390 - Hardware/Software Co-Design and ECE 4460 - Real-time and Embedded Systems [18]. These topics are taught through the use of industry focused software such as Intel Quartus Prime (formerly Altera Quartus II) [19], Xilinx ISE WebPACK [12] or WinIDE [20]. These established industry tools are complex and powerful for industry but we have anecdotally found students often feel overwhelmed during use. Additionally with the prevalence of high-level programming languages, embedded systems work has become increasingly difficult for students. Students are troubled connecting high-level concepts to low-level details in computer architectures largely due to theoretical discontinu-
ities between lecture materials and the “black box” of hardware.

### 1.1 Research Question

This thesis attempts to answer the following research question:

Can improvements and increased usage of software simulation technologies within laboratory exercises in undergraduate embedded system and computer architecture courses improve student engagement in laboratory and assignment exercises?

Given the overall research question, there exist sub-questions:

1. Are simulation software beneficial to improvements in student outcomes in computer and software engineering programs?

2. What elements of simulation software should be required for successful implementation within an undergraduate course?

Each of these questions are elaborated in further sections.

### 1.2 Motivation

Students in computer engineering have a growing requirement to have strong knowledge of both embedded systems and computer architectures due to increases in computer engineering careers [10], [17]. Unfortunately, many students do not show enthusiasm for subjects surrounding computer architecture and embedded systems [16], [21]. This lowered enthusiasm reduces student achievements and creates implicit barriers for learning new material [15], [21].
1.2. Motivation

At Western University, students in all students in Electrical & Computer Engineering degree programs must take the course ECE 3375 - Microprocessors and Microcomputers [22], [23]. We utilized the feedback students provide through end-of-semester course evaluations to gauge student enthusiasm and sentiment regarding ECE 3375. In order to extract sentiment, we applied a simplistic approach. If a comment was overtly positive or negative, we stated it was positive or negative respectively. If a comment was indistinguishable, we erred to neutral to try and remove our own biases. Unfortunately due to confidentiality, we are unable to provide these comments directly. As seen in Table 1.1, student enthusiasm for ECE 3375 at Western University is mixed. Unfortunately, our collected data is not broken down between computer and software engineering – both are required to take this course. Additionally, with primitive sentiment analysis applied to results from 2013 and 2014 [24], [25], shown in Figure 1.1a we can see that after completion of the course, students are pleased overall with the course content itself as approximately 73% of students found the experience positive. However, by filtering comments regarding the laboratory sessions and applying the same technique for sentiment as the course, shown in Figure 1.1b we find that students are likely very unhappy with the lab components as approximately 33% stated negative reviews and only 24% stated positive reviews. Due to time constraints, we were unable to compile a formal survey due to time required for ethics approvals and program duration.

Researchers have found that there are concerns with low-level hardware constructs being taught to high-level Computer Science, Software Engineering and Computer Engineering graduates [15], [16]. Additionally, we can corroborate the following claims by Ristov et. al. [15] regarding student’s opinions on microprocessor courses:

Observation 1. Disjointed material between lectures, theoretical and practical ex-
Figure 1.1: ECE 3375 - Course Sentiment Analysis Results [24], [25]
1.2. **Motivation**

Observation 2. Inappropriate programming and simulation environment. Students faced installation problems [...] leading to aversion to presented material and exercises.

Observation 3. No “real world” application. [...] Students [raise] the main question: Why we are learning this, and how and where shall I use it?

(15, p. 340)

Given our first-hand experiences in both taking and assisting in the instruction of ECE 3375 and ECE 3390, these claims are repeated by students during laboratory exercises and review sessions creating hostility towards the subject matter. In a direct example supporting observation 2, students in ECE 3375 use the WinIDE12Z Integrated Development Environment [20], shown in Figure 1.2, for completing laboratory work ranging from simple “hello world” assembly programs to hardware interfacing. In ECE 3375, students found WinIDE complicated, frustrating and “outdated” [24], [25] directly supporting both observation 2 and observation 3. Additionally students found it difficult to move from lecture material that presented embedded systems as “translucent” devices composed of inner components to the literal “black box” of the physical hardware shown in Figure 1.3.

![Table 1.1: ECE 3375 Course “Level of enthusiasm before taking the course” as reported by undergraduate students following course completion](image)

<table>
<thead>
<tr>
<th>Initial Level of Enthusiasm</th>
<th>Year</th>
<th>2013</th>
<th>2014</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>High %</td>
<td>45.6</td>
<td>57.8</td>
<td>53.0</td>
<td></td>
</tr>
<tr>
<td>Medium %</td>
<td>54.2</td>
<td>33.3</td>
<td>41.6</td>
<td></td>
</tr>
<tr>
<td>Low %</td>
<td>0.0</td>
<td>8.9</td>
<td>5.4</td>
<td></td>
</tr>
<tr>
<td>Total #</td>
<td>59</td>
<td>90</td>
<td>149</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: ECE 3375 Course “Level of enthusiasm before taking the course” as reported by undergraduate students following course completion [24], [25].
Figure 1.2: WinIDE12Z Integrated Development Environment [26]

Figure 1.3: Freescale M68HC12 development board built by Western Engineering Electronics Shop for use in ECE 3375
back [24], [25], we extracted several common propositions for improving laboratory sessions:

Proposition 1. Increase the number of laboratory sessions

Proposition 2. Add tutorial sessions to course schedule

Proposition 3. Change course material to use modern processors

Proposition 4. Replace laboratory microcontrollers with different hardware

Proposition 5. Improve and/or replace software within laboratory

All of these considerations were brought to the course instructor and limitations were noted for each. For Propositions 1 and 2, while the addition of more hands-on experience has been shown to improve student outcomes [15], [16], due to the large amount of financial resources required to add more laboratory sessions, this option is likely unavailable for many programs. In addition, due to the layout of the current course laboratory sessions, there does not exist enough weeks in the 12-week course to add more sessions given current staffing resources. It is reasonable to assume that these issues are prevalent at other institutions. Thus, both Propositions 1 and 2 can not be implemented given financial and temporal restrictions.

Proposition 3 asks the instructor to change the current course material to use more modern hardware and software architecture. There are several reasons why this is not beneficial to the improvement of learning outcomes. The current architecture used is the Freescale M68HC12. This architecture provides a simple Complex Instruction Set Computer (CISC) Instruction Set Architecture (ISA) with only four 16-bit addressable registers (SP, PC, X, Y); two eight-bit accumulators (A, B) and multiple addressing modes [27]. The software abstractions of the Freescale
M68HC12 assembly is similar to current Intel® x86 processors [9]. Parallels between architectures allows instructors to use more simple architectures that still allow students to apply concepts to newer architectures while simultaneously reducing the cognitive “base knowledge” required to succeed. Using simpler architectures like the Ultimate Reduced Instruction Set Computer (URISC, [6]) has been shown to be very beneficial to improving student outcomes [6], [28]–[31]. Additionally, by changing course materials, it leads into changing the laboratory hardware which Proposition 4 discusses. Unfortunately, replacing laboratory equipment requires significant investment and with justifications for Proposition 3 showcasing that there is not noticeable improvements in learning outcomes, the cost-benefit analysis does not prove worthwhile to adjust both Propositions 3 and 4.

The last proposition, Proposition 5 is the most attainable. Researchers have found that improving software technology in the classroom and laboratory can vastly improve the learning outcomes of students while improving engagement [10], [15], [16], [21], [30], [32]–[35]. Students within ECE3375 are immediately presented with the interfaces shown in Figure 1.2. Students have expressed immense frustrations with the software when used in the laboratory exercises and often encounter hardware issues that are not caused by their own work [24], [25]. Other researchers have found hardware solutions problematic with hardware-specific problems common such as broken peripherals and unconnected pins [21]. These errors are hard to debug, if not impossible for students with no prior knowledge. Teaching assistants and instructors are often unable to debug these issues without the aid of technicians. These hard-to-diagnose hardware issues lead to strained teaching resources and frustrations, supporting a large potential gain by replacing this ageing software with new tooling. Given these justifications, this thesis looks to investigate the requirements surrounding replacing the current software in use.
within the ECE 3375 course and propose a configurable solution for others to utilize in other courses.

1.3 Problem Statement

Our analysis of the current state of the ECE 3375 course showcased the pervasive dislike and discomfort with the current laboratory exercises [24], [25]. Given previous analysis, we chose to improve the current software used within the laboratory exercises and the classroom. To do this, a current survey of known technologies is completed to ascertain whether a new solution should be created or an existing solution will provide the requirements. Therein the formal problem becomes:

Can a solution be provided to students that is configurable for hardware and software interfaces and showcases the connection between hardware and software while focusing on pedagogy rather than industry performance?

While the formalization is accurate, “striking the right balance between teaching sufficient details of hardware components and their working principles, and important theoretical concepts useful for programming the computer is always a challenge.” ([34]) From reviewing many existing software solutions to this problem, we propose the following requirements for a successful solution:

**Requirement 1.** Must be available for use outside of the laboratory on personal computers including all major desktop platforms (e.g. Windows, macOS and Linux)

**Requirement 2.** Provide a student configurable system for different ISAs including configuration of:
2.a. microcode and assembly instructions

2.b. execution semantics (i.e. how a processor executes code)

2.c. internal hardware components and connections

2.d. external peripherals

**Requirement 3.** Focus on pedagogy over simulation accuracy as best as reasonable

**Requirement 4.** Simulations must allow:

4.a. viewing of the current state of hardware components

4.b. stepping at the assembly and microcode level

4.c. setting of breakpoints to ease debugging

4.d. connecting “external” peripheral components

**Requirement 5.** Provide a modern user interface that is similar to current high-level programming IDEs

**Requirement 1:** Personal computer usage. [Requirement 1] outlines that any software must be cross-platform. StackOverflow’s Developer survey [36] found that approximately 47.9% of those surveyed used a non-Microsoft Windows environment (macOS or Linux). The metrics recorded for 2016 desktop operating systems are shown in [Figure 1.4](#). While other metrics show that over 80% of computers are Microsoft Windows [37], we believe the StackOverflow survey is better given students in computer and software engineering programs likely go on to become hardware or software developers [36, Sec. II. Developer Profiles]. Further, StackOverflow has found the year-over-year change to show a decrease in Microsoft Windows-based op-
1.3. Problem Statement

Operating systems between 2013 and 2016 \[36\]. StackOverflow’s results imply that this project’s solution must run on all three major operating systems with minimal effort to allow students to perform out-of-classroom exercises.

**Requirement 2: Expanded hardware/software configuration.** As stated within the Computer Engineering Curricula 2016:

One area of concern to the computer engineer is the software/hardware interface, where difficult trade-off decisions often provide engineering challenges. Considerations on this interface or boundary lead to an appreciation of and insights into computer architecture and the importance of a computer’s machine code. At this boundary, difficult decisions regarding hardware/software trade-offs can occur, and they lead naturally to the design of special-purpose computers and systems. ([10, p. 32])

**Requirement 2** enables students to design and configure all of the hardware/software component interactions. This enables students to have a stronger understanding of ISA design. While many tools have this flexibility (discussed later in [Section 2.2]), many of the currently used, industry-grade solutions provide over-complicated and powerful options that are overwhelming for students. It is expected that disorientation by complication of user interfaces for users may have an impact on performance, production, motivation and morale of users \[38\]. This “user hostility” must be reduced for students as they are less likely to have the motivation that an industry professional may have \[30\]. As the focus of this thesis is to improve student learning outcomes, the amount of irrelevant options for learning must be reduced as a trade-off in favour of pedagogical improvement instead of accuracy as listed in **Requirement 3**.
Figure 1.4: StackOverflow 2016 developer survey results for desktop operating systems [36, Sec. VIII. Desktop Operating System]
1.3. **Problem Statement**

**Requirement 3: Pedagogical versus accuracy trade-offs.** While pedagogy is the most important aspect of a teaching simulation, we also acknowledge that it is not a binary scenario. Any such solution must be accurate enough for students to understand underlying concepts while being high-level enough to not inhibit their learning outcomes by overwhelming them. **Requirement 4** outlines the software requirements that simulations must meet while still respecting **Requirement 3**. Simulations are an important practical tool for students to see the inner workings of hardware components and attempts to increase the transparency into the long-standing “black-box” of hardware for embedded systems courses. For computer architecture courses, simulation tools are required for debugging and implementing ISAs.

**Requirement 4: Run-time simulation features.** We believe students best learn from fully understanding concepts from a bottom-up approach. With the ubiquitous nature of debugging facilities in modern software environments, we assert any successful software simulation must have similar capabilities for the hardware it is simulating. This includes breakpoints and viewing of registers as “variables” within the context. These features give students a familiarity and a view inside the workings of a simulation.

**Requirement 5: Modern interface.** **Requirement 5** attempts to reduce the overhead gap of working with new tools by reducing differences in features for students. Decreasing the overall cognitive load by integrating similar styles and controls to existing tools within interfaces will lower the cognitive load for students which should in turn improve learning outcomes [6], [38].
1.3.1 Software versus Hardware

While providing a solution in software that emulates hardware we would like to discuss how the hardware itself comes into teaching contexts. Some would argue that a “simulator may perform the same actions as a hardware device, therein deprecating the use of hardware.” We believe this is false. Students require the use of hardware to understand the physical nature of the device being used. Many defects found in embedded systems projects are only found by probing and testing hardware directly. By providing a better software simulation, we can provide students with a home experience that is similar but not equivalent to working directly with hardware. This new experience allows students to spend less time learning in the moment on the hardware and more time understanding the physical nature of the devices [32], [39].

1.4 Contributions

This thesis is presented in an additional five chapters. This thesis discusses technical topics regarding: existing and future pedagogical simulation technologies; development of web applications on modern platforms; building cross-platform native applications; and integration of scripting languages as configuration engines within a native application. We summarize the contributions to each in the following sections.

Survey of other and previously completed works

Within this chapter, we provide a quantitative survey of existing simulator technologies in use today analysed against the requirements outlined in Section 1.3.
1.4. Contributions

**procsim.scala: Scala-based event-driven processor simulation for the web**  We investigate utilizing Scala and its compiler back-end Scala.js to build a parallel component-based processor simulator for the modern web. Within this investigation, we showcase the following contributions:

- A design for an asynchronous actor-based model for simulation of a custom processor architecture using distributed computing framework Akka

- A runtime, compiled VHDL-like DSL for specification of instructions within a custom ISA on top of Scala in the Java Virtual Machine and Scala.js

**Developing cross-platform C++ applications**  We discuss an anecdotal account of refactoring the hc12sim project into a modern C++ environment. We discuss topics involving: developing a multi-platform project and providing adequate quality assurance, improving CMake through custom build scripts, and improving application build times through optimization of build artefact selection. The contributions outlined within this section are:

- An automated platform-specific testing infrastructure using Virtual Machines provisioned with Vagrant and Oracle VirtualBox automated with Jenkins’ Pipeline architecture

- A collection of project-based CMake scripts to isolate build targets while improving build times, reducing manual configuration and modularizing test specification

**Lua-based configuration-driven processor simulation**  We discuss the use of scripting engines in a native C++ application and their integration points; utilizing Lua as a scripting language within an application; implementation of an application while developing integration...
tion tools in parallel; and the pedagogical gains of a high-level language utilized for teaching low-level concepts. We provide the following contributions:

- A design for a runtime configuration specification for custom processor architectures through Lua scripts

- A state-machine-like representation of a processor instruction execution specified through “compiling” a Lua function to microinstruction events
Chapter 2

Survey of other and previously completed works

2.1 Previous Work: hc12sim

In 2013, we worked collaboratively with colleague and fellow student Ramesh Raj to build a behaviourally accurate simulator for the Freescale M68HC12 processor. This work was completed as part of our capstone project for a Computer Engineering degree from Western University [40]. This software was produced over 2013-2014 and had the following features:

1. Included memory element simulation for the Freescale M68HC12 architecture (e.g. registers, RAM, ROM)

2. Compile-time configurable microinstructions (addressing modes were not configurable)

3. Real-time simulation speeds for the Freescale M68HC12

4. Debugging of assembly code with runtime viewing of internal hardware components
5. Accurate CISC addressing modes as specified by Freescale M68HC12

6. Feature complete assembler creating exact machine code

7. Simple IDE for writing Freescale M68HC12 assembly code, compiling and simulating it (shown in Figure 2.1)

![hc12ide Integrated Development Environment showing a student error in an incorrect specification of an instruction mnemonic.](image)

**Figure 2.1:** hc12ide Integrated Development Environment showing a student error in an incorrect specification of an instruction mnemonic.

### 2.1.1 Software Design

This project was written in C++ utilizing modern C++11 features such as lambdas, closures and cross-platform libraries (e.g. Qt and Boost)

1. Available: Qt ([https://www.qt.io](https://www.qt.io)) and Boost ([https://boost.org/](https://boost.org/)). The primary design goals for the hc12sim project were:

1. Support all three major platforms: Microsoft Windows, macOS, and GNU/Linux
2.1. Previous Work: hc12sim

2. Separate compiler and simulation components

3. Completely decouple any IDE from the simulation engine

4. Heavily unit-test all software components for validity confirmation

The goals set for the hc12sim project closely mimic the requirements set in Section 1.3. Given these design goals, supporting technologies were assessed against the following requirements (in order of priority):

1. User interface library available

2. Access to low-level types

3. Fast enough for simulation

While initially writing the software in Java utilizing Java’s Swing Toolkit and the Java Standard Library, it was rewritten in C++ due to access of low-level unsigned types, method references and multithreading capabilities. Java’s current Java 8 release added these features, however the Java 8 release was not available at the time of creation outside of “beta” environments.

Instructions

Due to the Freescale M68HC12 having over 200 instructions with multiple addressing modes for each [27], the simulation and assembler libraries put heavy emphasis on generating code to simplify writing and adding instructions. This allowed for generating multiple classes that had all of the meta-information required for all stages of a program (writing, assembly, programming, execution). In order to configure and ensure the quality of the large amount of
generation required, a custom user interface was created to easily load and store data and generate C++ code. Figure 2.2 showcases the interface created for code generation. The primitive interface supported choosing an instructions supported addressing modes and writing the transformations required within the small code window (bottom right of Figure 2.2). The generator interface did not support any syntax highlighting for the C++ code required and performed “magic” code injection upon generation [27, Table 3-1, p. 30] In addition to the user interface, the generator back-end was built into a command-line tool that allowed the generation to be run at compile time when hooked into build scripts (in this case, CMake [41]). The user interface generated JSON entries similar to that shown in Listing 2.1. The “body” field is injected into a method header for the execution engine and the other information is used in either compiling or execution as meta-information.

Figure 2.2: hc12sim Instruction Generator showing the meta information for addressing modes, execution, and output.
2.1. Previous Work: hc12sim

Listing 2.1: Sample of the JSON output used to generate code within hc12sim. BEQ performs a branch if two values were equal.

```
{  
  "BEQ": {  
    "body": "
      auto ccr = cpu->CCR();
      bool _Z = ccr->get(Z);
      if ( _Z ) {
        int8_t rel = math::toSys<int8_t>(resbytes[0]);
        cpu->PC()->inc(rel);
      }
    
  "cycles": 1,
  "handlePC": false,
  "types": {
    "REL8": "27"
  }
  },
}
```

Hardware Simulation

Physical modules within the simulation model were all individual custom classes. For example, the A and B registers were of class `Register<size_t width>` where the `width` template parameter defined the bit width of the Register. Regrettably, any special behaviours were “decorated” on top of the base type creating many different types depending on how the underlying component was used. Using the pattern of “compile-time constant templates” created issues when trying to use these types in polymorphic aggregate containers. For example, if a `Register<8>` A is defined and a `Register<16>` X typeof A is not the equal to typeof X as templated types are created at instantiation time making the type of a variable a “new” type (reuse happens when available)\[42\]. This created confusing class hierarchies and directly contradicts advice of the Decorator pattern\[43\] p. 175 creating large concerns for maintainabil-
ity. Of particular interest was the design of the class Timer and class TimerDependant types which allowed for completely concurrent, event-based timing. While these concurrent timing modules were fast and well designed, it created problems with discrete execution as each component operates concurrently in separate thread contexts and can not be “stepped” through without executing all of the threaded operations simultaneously. In addition, each TimerDependant added a thread of execution adding runtime overhead to simulation while reducing functionality of the simulation. Due to our ignorance at the time, neither thread nor fork-join pools were used to reduce the thread “spin-up” performance penalty. Further, the synchronization properties of this scheme reduced to near-synchronous execution with the time taken to check locks and mutexes likely costing more time than the savings of parallelism afforded – not to mention the required knowledge of maintaining a highly parallel code base.

**Execution**

The simulator implemented execution through an class Executor that asynchronously executes the execution scheme specified by the Freescale M68HC12 [27, Sec. 4, p. 47] which was implemented in terms of [44, p. 59]. The Executor utilized a fetch, decode, fetch (if needed), execute loop for executing all instructions. The actual execution of an instruction was completed directly by static code written in the instruction generation step (see Section 2.1.1 for more information). However, the generated code is run within a separate thread context to keep the “clock” of the system running at a consistent speed. Of particular note, this simulation could not handle interrupts or pipe-lining. Due to the reliance on the timing mechanisms discussed in Section 2.1.1 the execution engine suffered from the same “synchronous” parallelism. Further, if the clock speed was too fast and execution actions moved too slowly, the
result was an non-determinant clock skew over time.

2.2 Evaluation of existing simulation technologies

Presented in the following sections are a series of similar projects attempting to tackle the same solution space as the work of this thesis. Each software tool is evaluated against the requirements from Section 1.3. The intention of this evaluation is dual:

1. Find previous works that could be expanded upon and improved to match this thesis’ requirements rather than building a new solution in a “green field”

2. Extract features from other simulation technologies that are beneficial to future technology

Each simulation project is rated based on the requirements out of five and totalled to create a quantitative matrix of requirements. Each rating contains justifications for the rating assigned following Section 2.2.2.

2.2.1 Methodology

Holistically, we attempted to find the best solution for each requirement outlined in Section 1.3. To do this we created a ranking of each of our considered software and applied a rating of zero to five with zero being the worst, and five being the best matching of each requirement. Within this specification, we believe some requirements were not met by any of the software evaluated. For example, there is no “best” software in our opinion that met Requirement 5 perfectly with
having a modern user interface. Our methodology allows for a relative, though opinion-based, approach to contrasting existing solutions.

2.2.2 hc12sim

The hc12sim project was very immature and not well implemented in the presented state. hc12sim managed to provide a solution for all of the required platforms when distributed as individual platform specific binaries. The hc12ide was a desktop application and bound to the technology choice of C++, there is no possibility of using the software outside the desktop without large software porting efforts. Unfortunately, building C++ applications such that they can be distributed to cross-platform targets is labour intensive and very difficult without the correct amount of knowledge – knowledge that was lacking at the time of authorship. Binaries worked for some and often only worked with specific unknown environmental state requirements (particularly with Windows environments). Regarding Requirement 1, the project meets the requirement adequately, but does not do enough. Configuration of the ISA is provided through the JSON-driven mechanism for instruction generation but this mechanism only worked as a compile-time configuration. If students wanted to change the ISA for their processor, it required rebuilding the entire project and testing to see that it worked manually. This was aided by the use of the “Instruction Generator” described in Section 2.1.1 but does not remove the requirement of building requiring a large compiler tool-chain. Further, without any support for hardware configuration the project did not meet Requirement 2. hc12sim heavily favoured Requirement 3 as it was intended to create a simulator that was behaviourally accurate to the Freescale M68HC12, but only if it allowed students to better understand how the processor
2.2. Evaluation of existing simulation technologies

The simulation interface allowed for debugging at an instruction level and setting break points within execution contexts. During an execution, all component states could be viewed – except for the memory. The project also assembled fully compliant machine code that could potentially be programmed to a real machine. Unfortunately, peripherals and interrupts were not supported. While including these flaws, the hc12sim project provides a poor match for Requirement 4. Lastly, while the user interface does not have all of the features of a full IDE, it did provide syntax highlighting, and assembler plus simulation controls in an easy to use interface. Requirement 5 is adequately met, however it requires updates to the editor interface to improve modern facilities.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>hc12sim [40]</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 2.1: Qualitative requirement rating of hc12sim [40]

2.2.3 ShelbySim

ShelbySim is an education-oriented software system for designing, simulating, and evaluating computer-engineering based applications [35], [45]. ShelbySim was designed surrounding a new Java-like programming language including a compiler explicitly built around providing extensive diagnostic information such as logging, tracing, and inspection capabilities. These tools provide students with raw data for quantitative analysis, evaluation and reporting of their designs. The software is open-source, though unavailable, and is written using Java allowing full operating system independent support. Additionally, graphical visualizations of results are provided for viewing developed components. ShelbySim is broken down into three subcomp-
CHAPTER 2. SURVEY OF OTHER AND PREVIOUSLY COMPLETED WORKS

nents:

1. Software component - a custom programming language (Shelby), a compiler, and an interpretation runtime

2. Hardware component - filling a similar niche to MultiSim, but with tight integration with Shelby and its underlying tracing. Additional support exists for external component integration

3. Simulation component - providing a deterministic and stochastic approach for recording inputs and outputs to custom hardware simulations

ShelbySim provides evaluation criterion for students’ components and their underlying systems to aid in marking. The simulated components have parameters that are modifiable through switches and sliders (e.g. \{on, off\} or a range from 0 - 100%). This gives students metrics to evaluate their designs. Additionally, outputs are exported at runtime to Comma Separated Value (CSV) files allowing for more in-depth analysis with external programs such as Microsoft Excel or MATLAB. This gives a flexible and realistic testing environment for student learning.

Analysis of Requirements

For [Requirement 1], ShelbySim is open-source and claims to run on all major platforms that the Java Virtual Machine runs on, thus it should work on Windows, macOS and GNU/Linux. ShelbySim is built as a custom programming language which is similar to Java but is used as an HDL. Given the inherent flexibility of HDL languages, [Requirement 2] is met, however these languages fall under a more complicated model than required for this thesis’ use case,
therein hurting Requirement 3. Further, ShelbySim focuses on compiler semantics and hardware configuration over ISA and embedded systems knowledge. ShelbySim works well to provide quality simulations as required by Requirement 4. Additionally, ShelbySim produces comma-separated values of events which is useful for later analysis. Lastly, there does not appear to descriptions of a user interface component outside of graphical drawings. This does not concretely fail the modern interface component for Requirement 5 but it does call into question how students may react to a lack of a proper programming environment. Additionally, there does not appear to be compiler features in place to utilize the custom ISA that a student creates (e.g. assembler).

<table>
<thead>
<tr>
<th>Requirements</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShelbySim</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 2.2: Qualitative requirement rating of ShelbySim [35], [45]

### 2.2.4 EDCOMP: Flexible Web-Based Educational System

EDCOMP is an educational computer system and a web-based simulator that can be used to cover computer architecture fundamentals; computer organization; computer arithmetic; memory hierarchies and organization; and simple input/output relations [30]. EDCOMP is written in Java and runs inside of a Java applet within a web browser. The simulator supports animation of instruction execution and allows students to write their own assembly programs, compile them and view the status of the machine components. All parts have visual components at multiple levels (i.e. module, combinational and sequential circuits). The simulation can be run at several levels per clock cycle, per instruction and for the entire program. Further, the simulation supports advanced topics such as interrupts and parallel I/O units. Lastly, tim-
ing diagrams allow students to see how components are interacting at a lower, discrete level. This software aims to create a system that focuses on pedagogical learning rather than on industrial accuracy and lower complexity of the system. EDCOMP focuses heavily on graphical representation to aid in student learning.

Analysis of Requirements

For [Requirement 1] EDCOMP is written in Java and was previously run within a web browser inside of a Java applet intending to open access to students with a low barrier to entry. However, due to the deprecation of NPAPI[2] in Google Chrome, Chromium-based browsers, and Firefox, this means that likely over 80% of students would not be able to use this software [46]–[51]. In addition to being unusable with today’s browsers, the system is also dated by the use of applets and does not have modern editor features creating a gap between modern technology and students therein failing to meet [Requirement 5] The EDCOMP system is not completely configurable as it is bound by purposeful reductions in features for pedagogical reasons. It features a CISC-based architecture with configurable components, however the internal connections can not be configured—only the size and “shape” of components. Thus, EDCOMP does not meet [Requirement 2]. EDCOMP heavily focuses on learning and teaching and showcases the use of a user interface to improve simulation triggers in a pedagogically focused way. The authors showcased the software by utilizing it within classrooms meeting [Requirement 3]. Finally, the simulations described are extremely useful and thorough. These simulations showcase powerful techniques in stepping and debugging and also triggering signals in meaningful ways to show students how different components can interact. The quality of these simulations

---

2.2. Evaluation of existing simulation technologies

2.2.5 p88110: A Graphical Simulator for Computer Architecture and Organization Courses

The p88110 attempts to cover many core topics in Computer Architecture and Organization [31]. The authors reflected and have created a “one-size fits all” solution to try and replace many heterogeneous simulators used in practice at the time. This software emulates the ISA for the MC88110 microprocessor specifically – a RISC architecture. The authors intentionally removed components to alleviate the amount of information for students to learn. The user interface includes a view for the current simulation of components, and a simplistic, built-in editor and simulation stepping at the instruction level and adding debugging breakpoints. The simulator can be modified to execute in serial or parallel (super-scalar) modes. In parallel mode, the four-stage pipeline of the MC88110 is implemented with static branch prediction and delays. A unique feature is the use of built-in caches for instructions and memory. These include configuration for time-to-read and write. During simulation, cache and branch prediction hit and miss statistics are recorded for student analysis purposes. The authors have implemented this simulator in class assignments and created an automated evaluation tool for reducing overhead for instructors.
Analysis of Requirements

The authors claim that the p88110 simulator may be run on personal computers, however given changing technologies and a lack of software description, we could not validate the software is still able to run on current operating systems implying the system does not meet Requirement 1. p88110 directly emulates the MC88110 system which is a similar system to the Freescale M68HC12 and has been shown to be an excellent candidate for pedagogical purposes. Requirement 2 states that a software must be able to be configured for multiple architectures and while the MC88110 is a great candidate architecture specifically, p88110 is too limited to meet this requirement. However, the engine configuration features presented by p88110 are extremely powerful and useful. p88110 was developed for use within a classroom meeting Requirement 3. Further, p88110 reduced realism to create a simpler architecture for students to learn from – favouring pedagogy over accuracy. Additionally, the use of an automatic marking tool would reduce the overhead for teaching students and give students instant feedback without requiring instructor interaction. The simulations provided by p81100 are thorough and provide insight into statistical modelling of pipelined and cached architectures. This information is extremely relevant to computer architecture courses. However, the p88110 does not support the use of peripherals or interrupts. Requirement 4 is not met due to lacking features, though the statistical feedback provides an interesting insight for students to learn from. Lastly, the interface is not modern and supports only simple features found in most editors. The amount of dialogs created shows concern for focus when working with p88110 and does not reduce students cognitive workload therein failing to meet Requirement 5.
2.2. Evaluation of existing simulation technologies

<table>
<thead>
<tr>
<th>Requirements p88110 [31]</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2.4: Qualitative requirement rating of p88110 [31]

2.2.6 EASE - An Extensible Architecture Simulation Engine

Extensible Architecture Simulation Engine (EASE) is a simulation engine focused on custom simulations for classroom use [34]. EASE attempts to provide the following features:

- Support for multiple architecture types: RISC, CISC and URISC
- Provide a modular mechanism for simple extension
- Open-source software
- Portable to different platforms

The authors [34] reviewed the survey of simulation tools found in [33] and found the suggested tools were inadequate for teaching simulation architecture courses based on their requirements. Many put too much emphasis on RTL descriptions of the hardware and too little on ISA. EASE provides three ISAs for use, a CISC, RISC and URISC within the project. EASE is written in Java making it cross-platform. It comes with a very simple user interface written in Java Swing.

Analysis of Requirements

EASE is a very immature project without a lot of proven use--this makes it difficult to gauge how effective it is. EASE focuses on pedagogy as it’s major requirement meeting Requirement 3. While the authors mention new architectures may be added through implementation of their interface Arch, students have no way to do this themselves and forces a recompiling
of the application to create a new architecture therein failing to meet Requirement 2. Given that EASE is written with Java, all that is required to run the software is a Java runtime making it cross-platform. However, stated the software was available under the GNU Public License v3, but the software is not currently available for consumption. Theoretically EASE meets Requirement 1 through using the Java Virtual Machine for it’s environment but we could not verify this. EASE does support step-based execution, however it has no support for debug breakpoints. Further, there is no reference to stepping at the microcode level. Runtime viewing of registers is available, however there is no way to view the memory. These outlined features do not meet Requirement 4. Lastly, the user interface provided is very simple providing syntax highlighting for a single file and simple debugging controls. The interface does not provide modern editing features that students are accustomed to with most editors – failing to meet Requirement 5.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>EASE [34]</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 2.5: Qualitative requirement rating of EASE [34]

### 2.2.7 TinyCPU and TinyCSE: Hardware simulations for education

TinyCPU is a teaching assignment designed to be used in a Masters of Science program in embedded systems to teach students about computer architectures by having them incrementally design a CPU (TinyCPU) [29]. TinyCPU has been used in a program described to be very similar to Western University’s computer engineering program. TinyCPU runs in Verilog and is a simple stack-based machine. The full block diagram for TinyCPU is shown in Figure 2.3. The architecture is meant to be extremely simple to mitigate extra information for students to
2.2. Evaluation of existing simulation technologies

Figure 2.3: Block diagram of TinyCPU showing the internal architecture [29, p. 869]

Figure 2.4: State chart of TinyCPU execution path [29, p. 869]

consume. The entire CPU configuration is rigidly bound with:

- 16-bit word size
- 12-bit address space
- A single ALU
- Dual bus structure with a data bus (dbus) and address bus (abus)

The CPU is compiled with a Verilog tool chain (e.g. Xilinx ISE) and loaded onto either a hardware or software-simulated FPGA to run. The execution model is intentionally simplistic for improving ease of learning (shown in Figure 2.4). Given that TinyCPU is written in Verilog, it works well with existing industry tools such as ModelSim. Lastly, there exists a C compiler and assembler for TinyCPU making a very strong case for modern programming interactions.
TinyCSE is an extension to TinyCPU that provides a full system supporting hardware interrupts that can interact with peripheral components like mice and keyboards [28]. TinyCSE adds an I/O space memory mapping controller for interacting with peripherals. This allows I/O components to “register” themselves into a position and programs in TinyCSE can read/write to I/O components through memory mapping. Further, TinyCSE adds an interrupt flag that the state machine utilizes to allow for hardware interrupts. These interrupts are extremely important as this is how most hardware devices interact with a CPU. The modified state diagram is shown in Figure 2.5. At an architectural level, interrupts are supported via CALL and RETURN instructions which branch to an interrupt routine and return from an interrupt routine respectively while maintaining the stack.

Analysis of Requirements

For the purposes of analysis, this section will only consider TinyCSE as it is a super-set of TinyCPU functionality. TinyCSE focuses on pedagogy first and foremost, but also focuses on a hardware descriptor language, Verilog. While Verilog is a very powerful tool, it requires a complicated tool-chain creating complications for the purpose of teaching introductory courses.

Figure 2.5: TinyCSE execution state diagram modified from Figure 2.4 to include interrupts (broken lines show changes from TinyCPU [29]) [28, p. 640]
– we do acknowledge the authors intended on utilizing TinyCSE for Masters of Science students. Additionally, the software required for typical Verilog environments is very expensive and not accessible to the average student. This does not fully meet the requirements of Requirement 1 and 3. The simulations provided by TinyCSE provide more information than any other simulator evaluated at the expense of complexity for the system and a high barrier of entry to configure the simulator. This barrier creates a large enough gap that TinyCSE does not meet Requirement 2 (see the discussion in Section 1.2 for a discussion on existing hardware IDEs). While failing to meet Requirement 2 due to the level of simulation granularity Requirement 4 is easily met given existing industry tools. Lastly, the software provides no user interface of it's own and requires the use of a Verilog IDE and FPGA programming software. These tools are modern and updated regularly, however they often fail to have the editor/IDE features students are used to not meeting Requirement 5.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyCPU/TinyCSE</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 2.6: Qualitative requirement rating of TinyCPU/TinyCSE [28], [29]

2.2.8 CPU Sim

CPU Sim is a Java CPU simulator written for use within a classroom environment [39]. CPU Sim allows students to design, modify, and compare various computer architectures at the register-transfer level and higher. Additionally, students may write and debug assembly code for custom architectures. The simulator and IDE are written in Java, with the latter utilizing Oracle’s JavaFX user interface. The software is currently available from [52] as “freeware.” In conversations with Dr. Skrien, he expressed the want to open source CPU Sim in the future.
to increase availability [53]. The user interface is feature rich allowing students to configure a processor with:

- Hardware specifications (e.g. RAM word size, length),
- Hardware inter-connections,
- Processor microcodes,
- Assembly instructions,

These specifications gives students the ability to configure a nearly custom processor. However, using the current 4.0.10 release, configuration options are missing: clock speed changes, CISC-like addressing modes, and peripheral support [52]. Using Java technologies allows CPU Sim to work with the three major platforms as a desktop application. CPU Sim’s IDE provides a modern assembly editor for custom architectures providing syntax highlighting, tabbed editor panes and simple editor commands like find-and-replace. During a simulation, CPU Sim allows students to view memory and register states. [Figure 2.6] shows the IDE during a simple simulation with a program written in a custom ISA open for editing. CPU Sim fully supports debug points and step-based execution in it’s engine. Unfortunately the execution engine is very simplistic and does not provide any configuration points outside of changing the instruction fetch microcode sequence.

CPU Sim supports creating custom assembly languages to “compile” to the a custom ISA. Assembly or “microinstructions” in the context of CPU Sim are specified as sequences of “machine instructions.” Each “microinstruction” is a single transfer or operation within the CPU. For example, “transfer between registers,” “read from memory” or “branch if condition bit.”
These microinstructions are configured based on hardware specifications. When designing an assembly instruction, the microcode is laid out with a drag-and-drop interface placing “fields.” The drag and drop interface is shown in Figure 2.7. The top layout shows the microcode, the bottom is the “assembly” layout which designates how the instruction is written in an assembly program. These values can be rearranged independent of each other, though relative ordering of repeated values is maintained. These interfaces allow for adding punctuation and non-encoded values to the assembly layout. This is a very powerful feature as the assembler provided will parse the custom assembly layouts to produce compliant microcode.

Lastly, CPU Sim contains excellent documentation within it’s internal help program. The help documentation is very deep and has many images and “tutorial style” documentation embedded internally. The author used this documentation exclusively to learn how to use CPU Sim and found it very useful. This documentation in particular makes CPU Sim attractive to
Chapter 2. Survey of other and previously completed works

Figure 2.7: CPU Sim Machine Instruction configuration interface showing combination of fields to create a microcode and an assembly instruction.

prospective instructors due to the reduction in student inquiries as most questions may be found in the program itself.

Analysis of Requirements

CPU Sim is readily available for use by downloading it from [52]. The software runs well on Microsoft Windows and Linux. macOS support is available, however it has some non-critical user interface bugs that make it feel slightly “foreign” on macOS (e.g. the menu bar does not reside in the operating system menu bar). This strongly meets Requirement 1 as it runs well on the three major platforms. The assembly editor has many modern features which leads well to having students feel as though they are using a modern software. However when configuring hardware modules, the user interface is difficult to navigate and does not always use suitable user interface components to better utilize the available space. Figure 2.8 shows the module modification interface for the Transfer Register to Register Array interface. The interface presented does not scale and for different modules, there exists many disjointed
user interface components where the same “expected action” between two different dialogs is completed a different way. In our experience, this led to confusing amongst those using the software. Unfortunately, these usability concerns hinder both Requirements 3 and 5 severely.

![Figure 2.8: CPU Sim 4.0.0 IDE module specification dialog showing modification of Transfer Register to Register Array microinstruction [52]](image)

While usability is impacted, the simplicity of these interfaces improves the pedagogical outcomes by removing non-essential information which supports Requirement 3. Dr. Skrien’s intention to open source CPU Sim should improve student learning by allowing students to investigate the source code within [53]–[55]. The large amount of configuration available strongly supports Requirement 2; however, due to missing features such as changing how the simulator executes code and the addition of external peripherals, CPU Sim is not a perfect match for Requirement 2. Lastly, CPU Sim’s simulations are very accurate and the user interface makes it simple to watch modules within the simulated engine partially meeting Requirement 4.
– though external peripherals are missing.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Sim [39]</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 2.7: Qualitative requirement rating of CPU Sim [39]

### 2.2.9 Emumaker86: A Hardware Simulator for Teaching CPU Design

Emumaker86 is an entirely GUI-based hardware simulator for designing CPUs in a Computer Organization course [56]. Emumaker86 is written in Java and is a desktop-based application. Emumaker86 is integrated on top of an existing personal computer simulator giving Emumaker86 full access to PC-compatible peripherals (e.g. video controller, CD drive). Emumaker86 attempts to keep students at a higher level than digital circuit simulators, abstracting away flip-flops and logic gates into registers and ALUs respectively. Emumaker86 breaks CPU design into two main components, 1. Datapath and 2. Control unit. These two are built independently from each other in custom user interfaces.

The Datapath builder works by students specifying RTL-like structures with digital components. Each of these components are laid out and connected through wires and buses. Any clocked component is connected to the “system” clock by design to simplify requirements for students. There are interrupt and I/O ports available to enable peripheral connections. These components can be combined to create sub-components as an abstraction. Each component runs in parallel within the device. All of these components and datapath entities are serialized as XML so they can be saved, distributed and reloaded.

The control builder is an interface used to specify a state machine with transitions and control unit states. The interface is shown in Figure 2.9. Each colour corresponds to a different
Analysis of Requirements

Enumaker86 meets most of the requirements outlined in Section 1.3. However, it is not without issue. For Requirement 1, a minor concern is that the software runs on Java requiring a desktop environment. Similar to [28], [29], Emumaker86 uses state machines to represent the control unit. The largest difference is that Emumaker86 allows for the state machine to be changed by users. Allowing students to change how the control unit functions is essential to give students experience designing a controller. Additionally, the “Datapath” creation mechanism uses a graphical way of designing the hardware controls while still hiding the internals of supplied components. Emumaker86 meets Requirement 3. There does not appear to be any assembler
mechanism requiring code to be hand-compiled to work on any architecture created (barring an existing architectural tool chain). Given the lack of an assembler, there is no assembly-level debugging – failing to meet Requirement 4. Lastly, the largest concern is the need for modern features. The interface provided for working with the state machine is extremely hard to read as it utilizes a table-like structure relying on colour coding over modern user interface design paradigms.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emumaker86</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 2.8: Qualitative requirement rating of Emumaker86

2.3 Summary

Given the results accumulated through our best effort objective comparison in Section 2.2, the results were tabulated into Table 2.9. Naively, out of the simulations, CPU Sim [39], [52] and Emumaker86 [56] have the largest total match for the requirements. This information was considered when selecting a trajectory for the thesis’ project, however the raw numbers do not extract enough information. Each simulator project has design ideas that can be combined into a better software to meet the requirements from Section 1.3.

The hc12sim project provided a very fast, cross-platform system, but similar to EASE, it was bound to compile-time configuration. Work completed on the hc12sim project lead us to develop the idea of runtime-configurable simulations (a known entity in other projects, e.g. CPU Sim [39] and Emumaker86 [56]). This runtime-configurable feature is paramount in any project moving forward. Further, several other projects had distinguishing features that should
2.3. Summary

<table>
<thead>
<tr>
<th>Requirements</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>hc12sim [40]</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>ShelbySim [35, 45]</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>EDCOMP [30]</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>p88110 [31]</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>EASE [34]</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>TinyCPU/TinyCSE [28, 29]</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>CPU Sim [39, 52]</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>Emumaker86 [56]</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 2.9: Summary of Evaluation for Simulators

be included future projects.

The first significant feature is the use of “event-driven” simulation that TinyCSE [28, 29] and p88110 [31] suggested. Event-driven simulation allows for a simpler approach to maintaining state within a simulation. Event-driven simulation models are further discussed in both Chapters 3 and 5. Another feature found to be extremely useful is the use of stochastic model outputs that can be integrated with existing tools. All of EDCOMP [30], ShelbySim [35, 45], TinyCSE [28, 29] and p88110 [31] provide tooling to integrate with existing tools such as ModelSim, Excel or MATLAB. While dependence on industry tools can create problems for pedagogical reasons, allowing for integration points with these tools gives students more experience with industry prevalent tools – particularly ModelSim and Excel.

A feature only provided by TinyCSE [28, 29] and Emumaker86 [56] is the ability to configure the entire execution loop. TinyCSE [28, 29] does this by utilizing Verilog and having students adjust the state machine directly. Emumaker86 [56] takes an entirely different approach having students specify a state machine based on input signals and state of components. Emumaker86 [56]’s approach gives student’s the most power in configuring a machine and uses a familiar concept of finite state machines [10]. The implementation falls flat due to the
hard-to-use user interface (shown in Figure 2.9). Students are used to seeing machines as Figure 2.5 displays. A system that gives the power of the “table-based” approach of Emumaker86 with the visual queues from the state chart would give both familiarity and power to students.

Lastly, CPU Sim [39] provides two features that appear to be distinct: 1. Strong accompanying documentation, 2. Assembler configuration. Documentation is important for any software to be heavily utilized, particularly if it is to be used by novice users. CPU Sim’s assembler configuration allows students to create their own assembly-level language tied into their custom ISA.
Chapter 3

procsim.scala: Scala-based event-driven processor simulation for the web

In the following chapter we present an event-driven processor simulation written in Scala utilizing the actor model with Akka. We aim to provide a configurable digital processor simulation that could be utilized as a web application within a classroom environment such as ECE 3375. In addition, we aim to provide a configurable enough solution through a Domain Specific Language (DSL) for specification that this project could be utilized within other courses such as ECE 3390.

3.1 Introduction to Scala and Akka

procsim.scala attempted to port work of hc12sim to Scala and Akka while providing incremental feature improvements surfaced by reviewing other projects in Section 2.21 In recent years,  

1See Section 2.1 for an extended discussion on the original hc12sim
Scala has emerged as a powerful language used in big data and scalable web applications thanks to Apache Spark\(^2\) and Lightbend Akka\(^3\)\[57\]–[59]. Scala is a multi-paradigm programming language supporting functional and object-oriented style programming on both the Java Virtual Machine (JVM) and within a JavaScript environment such as a web browser \[60\]. On the JVM, Scala can interoperate with Java programs without any code changes and in a JavaScript environment most Java functionality is available \[61\], \[62\]. One of Scala’s original use-cases was as a less verbose and more functional version of Java that runs within the same Java infrastructure. Thanks to Scala’s implicit parameters \[63\], custom operator overloading \[64\] and optional post-fix notation \[65\] developers can create Domain-Specific Languages within Scala with ease. For example, Listing 3.1 is a DSL that was written to emulate BASIC (a 50 year-old language) showcasing how flexible Scala’s syntax can be. In regards to processor simulation, all of the simulators surveyed provide at least one mechanism for loading and saving machine state to a persistent storage (e.g. XML files for \[39\], \[56\]). A DSL-based configuration allows specification of a system without extraneous syntax designed specifically for this purpose. Utilizing a DSL could allow students to specify their machines programmatically in addition to a graphical user interface improving the amount of configuration, reuse and flexibility available. This provides an excellent approach to improving Requirement 2’s configuration requirements.

Akka is a Scala framework that provides an implementation of the Actor model \[66\] that was originally written by Haller and Odersky in \[67\] for the Scala Library. Haller’s \[67\] original implementation makes use of an event- and message-driven programming model to lower thread counts and alleviate lock contentions compared to traditional threading models. Fig-

---

\(^2\)Apache Spark is available at: https://spark.apache.org/

\(^3\)Lightbend was formerly known as Typesafe. Akka is available at: https://akka.io
Figure 3.1 shows a simple diagram of actors sending messages bi-directionally between each other.

Each actor has a conceptual “mailbox” that they can store and process messages from. The initial actor implementation by Haller ([67]) was expanded to become Akka which is now sponsored by Lightbend. Akka’s major revisions to Haller ([67]) includes additional abstractions to remove locational knowledge such that actors “live” wherever they are best suited. Additionally, Akka’s scheduling mechanisms control Actor execution patterns in an idealistic and a deterministic way [68]. For example, two Actor instances can communicate with each other regardless of their physical location on the same node of a distributed system or completely geographically separated by a network – an individual actor neither knows nor cares where a message comes from or is sent to. Akka provides fault tolerance through supervisor semantics utilizing the “let-it-crash” paradigm for faults popularized by the Erlang programming language in telecommunication systems since 1985 [69]. These features on top of the actor model create a very scalable technology that maps simply to microservice-driven architectures. While Akka’s main goal is to create microservice architectures and scalable applications, we intended to utilize this model to instead implement an event-driven simulation model.

```scala
object SquareRoot extends Baysick {
  def main(args: Array[String]) = {
    10 PRINT "Enter a number"
    20 INPUT 'n
    30 PRINT "Square root of " % "'n is " % SQRT('n)
    40 END
  }
}
```

Listing 3.1: An example DSL source written in Scala to emulate BASIC called “Baysick” [1]
3.2 Event- and Message-driven Simulation

Several projects surveyed utilized event-driven models of simulation [28], [29], [31]. As discussed in Section 2.3, event-driven models provide a simplification to maintenance of state within an application. In a traditional object-oriented application, each object maintains its own state performing create, read, update and delete operations on itself and acts on other objects to trigger their state changes. By contrast in a traditional event-driven model, the “owner” of state still remains with components, but program flow is dictated by utilization of an event loop that processes events and asks components to “react” to these events further creating more events. This model is traditionally found in graphical user interface applications as it simplifies flow of a user-driven system. Lightbend coined the phrase “reactive” in regards to software by creating a manifesto known as the “Reactive Manifesto” that states the qualities of a “reactive” application [71]. The “Reactive Manifesto” details message-driven applications as:

Reactive [systems] rely on asynchronous message-passing to establish a bound-
ary between components that ensures loose coupling, isolation and location transparency. This boundary also provides the means to delegate failures as messages.

[...] Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

While the majority of this definition deals with distributed systems, the same benefits are gained with local applications. A processor simulator does not require elasticity or load management; it does however require the ability to process messages quickly and in such a way that modules do not block each other as electronics all operate in parallel. These ideals were transcribed into a model of a processor in which all components interact through “signal” messages between each other rather than direct interactions. This opens up the opportunity for much more accurate simulations while simultaneously reducing the effort to interact between modules. These relationships allow simulations to have higher fidelity and ease the time required to implement features lending well to both Requirements 3 and 4.

3.3 Scala.js - Scala transpiler for JavaScript

Scala.js is a “compiler back-end” for the Scala compiler that transpiles Scala code to JavaScript so that it can execute within a web browser. At the time of creating the procsim.scala project, the most recent release of the Scala.js transpiler was version 0.6.5 [61]. Version 0.6.5 allowed for most Scala applications to be cross-compiled to the Java Virtual Machine and JavaScript
applications with little-to-no code changes. Through Scala.js, a simulation written in Scala could be run easily on both a desktop and web platform opening up the opportunity to utilize Scala for both a front-end client and a web service. Requirements 1 and 5 support the use of a web-based solution due to the prevalence of web browsers’ compatibility in a cross-platform environment and current student’s relative comfort with web-based applications. Given our strong Java and Scala background, utilizing Scala.js to build a fully-featured web application on top of the Scala and Akka platforms was an excellent candidate for this thesis project.

### 3.4 Implementation of procsim.scala

#### 3.4.1 VHDL-like DSL configuration and runtime-based instructions

The first goal of procsim.scala was to introduce a DSL for defining modules and instructions within Scala. VHSIC Hardware Descriptor Language (VHDL) is a commonly used HDL language at Western University. By creating a DSL that mimics VHDL, students may define instruction states within their processors using a known language. The intention was to have all modules pre-built and configurable but instructions are specified by students to give a transparent look into microcoding for a processor architecture. Given the compile-time configuration presented by the original hc12sim project in Section 2.1.1 it was our intention to allow students to have the same level of reconfigurability at runtime instead. The largest benefit to runtime over compile-time configuration is removal of the requirement to install a large compiler tool-chain and wait for the rest of a large application to compile – in the order of 10 to 40 minutes
3.4. Implementation of procsim.scala

depending on hardware utilized\(^4\).

The author achieved runtime-configuration of a system by utilizing Scala’s built-in reflection toolbox [72]. Scala defines reflection as “the ability of a program to inspect, and possibly even modify itself.” ([72]) Self modification of running code is the primary feature required for runtime configuration of a model without utilizing separate compiler tool-chains and processes. Scala’s reflection tools allow for a developer to write code as a String and execute the result, storing any and all state produced by the “sub-program.” This feature allows students to specify code in a configuration editor interface and have the software provide Scala-based compilation feedback and JVM-level performance. Two listings are provided: Listings 3.2 and 3.3 each showing the definition of the same simple instruction. The syntax used for assignment is meant to mimic the syntax used in VHDL to “connect” two modules. Listing 3.2 utilizes a normal Scala class to define these operations, but Listing 3.3 uses runtime-reification to parse a String, producing a result with the same functionality as Listing 3.2. In small micro-benchmarks, an initial overhead for parsing the String and compiling it to run within the JVM at runtime was found, however there appears to be no performance penalty to execute code found within a reified construct. These test benches utilized primitive timing mechanisms and accounted for the JVM’s JIT warm-up time. Micro-benchmarks proved that it is possible to have complete runtime-based, and compiled configurations within a Scala application without utilizing a secondary compiler tool-chain. The examples shown have the ability to be expanded to mimic the capabilities found in simulators like TinyCSE [28], [29], reducing the overall knowledge required but still giving students the feeling of “hands-on” instruction

\(^4\)On an Intel® i7-3770k, a clean build of the hc12sim project takes approximately 10 minutes, incremental builds were faster depending on the files modified.
import net.navatwo._

class Concrete extends Instruction {
  def exec(cpu: CPU) = {
    import cpu._
    A <= 5
    B <= A + 1
  }
}

val cIns = new Concrete

import scala.reflect.runtime.
  universe => u }
val tb = mirror.mkToolBox()
val tree = tb.parse(
  "import net.navatwo._;" +
  "new Instruction {" +
  "def exec(cpu: CPU) = {" +
  "import cpu._;" +
  "A <= 5;" +
  "B <= A + 1;" +
  "}" +
  "}"
)

val sIns = tb.eval(tree).asInstanceOf[Instruction]

Listing 3.2: Normal definition of Instruction.  Listing 3.3: Reified definition of Instruction.

3.4.2 Akka-based Components

The second challenge was utilizing the Akka framework to produce hardware modules that
based on Akka’s Actor model. Within the original description of the Actor model, Agha states:

A processor is a physical machine while a process is an abstract computation.

From operating systems, we know that we may improve over-all performance of a
processor by executing several processes concurrently instead of sequentially. [66]

This quote is both ironic and informative as when considering hardware simulation, each mod-
ule should be developed as an individual asynchronous entity. Each has input and output con-
nections and only cares about signals on input connections. Instead of considering the system
sequentially, we consider the system as a massively parallel system in which all modules processes signals concurrently. In Akka, all messages are passed to all Actor instances and it becomes a design problem of filtering and producing data for other Actor instances within an Actor’s system. The author considered all hardware modules as Actor instances and in doing so, all modules can process information asynchronously within an Akka scheduler provided. Akka’s schedulers may run sequentially or concurrently, it is decided by the scheduling algorithm but Akka forces developers to treat everything as asynchronous to improve performance and reduce changes required should an Actor move to a distributed environment later \[68\]. While there is no intention of ever moving a hardware simulation to multiple distributed nodes, the concepts of immutable messaging and decoupling mutability and behaviour into Actor instances reduces the development choices in a positive way.

Akka’s Actor works by sending messages between Actor instances. In the context of Akka, a message is an envelope around state. Most often, this is a case class in Scala. As stated previously, a message has only immutable state. In the context of hardware simulation, a clock pulse might be represented as a message shown in Listing 3.4. This message only contains the time that the message occurred, represented as a Instant. In order for another module to listen for this pulse, it must implement the Actor.receive method which is a PartialFunction[\(\text{Any, Unit}\)]. The def receive method is usually implemented as a PartialFunction[\([]\)] using pattern matching \[73\] on message classes like Listing 3.4. Listing 3.5 shows a Actor that listens for our previously defined ClockPulse message. In order to send a message to an Actor, once an Actor is created the ! operator sends the message:

\[5\]Instant is part of the java.time library: \url{https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html}
actor ! ClockPulse(Instant.now()). With no other interaction required, the message is sent to actor regardless of where actor is found within the actor system. The simplicity of these interactions allowed for very straightforward communication protocols between modules and the paradigms enforced by Akka eased development efforts for new modules directly improving development agility.

### 3.5 Technology Challenges

This project progressed well while utilizing Akka on a local machine. However, in order to utilize Akka in the browser, much of Akka needed to be rewritten to utilize the browser concurrency semantics. A primary concern is that web browsers do not provide threading in the traditional operating system context, but they provide threading through the Web Workers specification [74]. Web Workers are not compatible with JVM threading as it is impossible to accomplish the full specification of JVM threads which include shared memory and sleeping under the current specification standard [62], [74]. As such, Scala.js has no means to implement Java’s threading model natively within a JavaScript environment. Without a supported JVM threading library, Scala.js does not have the ability to compile Akka to run within the browser directly. Conversely JavaScript’s programming model is heavily focused on asynchronous single-threaded programming paradigm through callbacks and surprisingly its model lends well to the actor model from [66]. As [75] eloquently explains:

```scala
case class ClockPulse(time: Instant)
```

Listing 3.4: Clock pulse message for Akka
class ClockedActor extends Actor {
    override def receive = {
        case ClockPulse(time) =>
            println(s"Pulsed at $time")

        case _ => // do nothing
    }
}

Listing 3.5: Actor that listens for ClockPulse messages and prints the time.

It may seem contradictory to implement an actor model, which is inherently concurrent and asynchronous, on a purely single-threaded platform like Scala.js. However, concurrency and asynchrony must not be confused with parallelism. While parallelism involves physically executing different tasks at the same time, e.g., on multiple processors or multiple machines, concurrency is a form of modularity which allows to model software components as independent units of execution and behavior which can communicate between each other. [75]

[75] implemented Scala Actors on top of the JavaScript VM utilizing a single-threaded “asynchronous model” as part of an undergraduate project to port the original Scala actors implemented in [67] to run on a JavaScript VM under the supervision of the authors of [67]. Through further development, [75] was extended to implement the Akka interfaces and became what is now known as Akka.js [76], [77]. The Akka.js project implements nearly all of the features of Akka encompassed entirely within a web browser JavaScript engine. The project has grown quickly over time, but at the time of our work it had large issues that we needed to manually patch in order to utilize Akka.js in a proper software stack. These complications created massive work flow problems and a majority of time was spent debugging Akka.js rather than
working on the procsim.scala project itself.

In addition to its immaturity, Akka.js could not keep up with simulation requirements (Requirement 4) as it did not support small discrete times due to relying on JavaScript’s Date implementation’s millisecond clock resolution and network latency [78]. Naively at this time, we intended accomplish real-time simulation speeds. Given that we achieved message passing with approximately 50ms of latency between the sender and receiver, the latency and lack of precise clocks meant that real-time simulation was completely unattainable. Additionally, this resolution was not acceptable in the intended parallel simulation environment as multiple operations happened concurrently at any given moment of time.

Another fault of ecosystem immaturity was that Scala.js created applications that were too heavy by default. At the time, the Scala.js compiler did not perform dead code elimination to an acceptable level leaving large applications to be distributed from a web server[6]. This meant that even small applications were in the order of mebibytes of minified JavaScript data to serve as part of an application compared to the kibibytes of information from other modern web frameworks. This size was unacceptable from an application distribution and usage stance (not adequately meeting Requirement 1).

### 3.6 Analysis of Requirements

Unfortunately, due to the immaturity of Scala.js, Akka.js and the ecosystem at the time, the procsim.scala project was ultimately discarded. The procsim.scala project had two contributions that should be addressed by future projects. First, using an “always asynchronous”

---

<sup>⁶ As of July 2017, the Scala.js compiler has *significantly* improved since the procsim.scala project and contains a powerful optimization tool-chain on top of Google’s Closure Compiler [79].</sup>
approach to modelling components was very simple to reason about and eased implementation
details significantly. Within Akka, all actors within an actor system are “connected” imply-
ing communication algorithms changed from a problem of “where to send data?” to “which
data does each module care about?” This filtering was non-trivial and often meant that Actor
instances were spending more time filtering messages than performing actions. This author
believes the question of “which” provides a simpler model than the direct connection model
with the same power implying a need for future investigation.

The second contribution is using a runtime DSL to configure the system gives students the
ability to modify components behaviour similar to the hc12sim’s instruction generation scheme
discussed in Section 2.1.1. The massive benefit over the previous project was in runtime con-
figuration instead of compile-time significantly reducing the required effort for students and
providing a foundation for both pedagogical gains and configuration capabilities (Requirements
2 and 3). As a qualification for any future DSL proposals, the DSL must feel familiar
to students so that they do not need to learn entirely new syntaxes or paradigms (in the case of
procsim.scala, this was using a VHDL-like syntax over Scala’s typical syntax).

Looking objectively at procsim.scala, it does not adequately meet many of the requirements
outlined by Section 1.3. Many of the following justifications are based on a theoretical prod-
uct as it was not implemented to completion due to concerns raised in Section 3.5. Utilizing
procsim.scala within a web browser allows students significant ease of use on their personal
computers meeting Requirement 1. Though, regrettably due to the speed at which Akka.js
performed in the JavaScript environment, its simulations were not fast enough to provide rea-
sonable experiences making the simulations feel unresponsive. It is difficult to say how flexible
the exposed interface for interacting with simulations would be without a full implementation.
Most modern web application frameworks work heavily on asynchronous paradigms, thus they should be able to handle a slower model but may integrate poorly (Requirement 5). Requirement 2 is aided by the use of the VHDL-like DSL language. For a pedagogical simulator procsim.scala is built to be simpler than a traditional fully behaviourally accurate simulator system. Lastly, procsim.scala was not completed far enough to showcase the functionality for simulation of modules for Requirement 4. One can assume that utilizing Akka actors, listening for signal messages would allow for collection of large amounts of state information.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>procsim.scala</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td></td>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of requirement matching for procsim.scala.
Chapter 4

Developing cross-platform C++ applications

After concluding work on procsim.scala, we applied lessons learned from procsim.scala and applied them to the original hc12sim project discussed in Section 2.1. Unfortunately, the hc12sim simulator was dated, poorly designed and poorly organized due to inexperienced student work. A large effort was placed in modernizing and correcting the hc12sim software to be well tested, build in a cross-platform environment, reduce feature development time and reduce the amount of time required for maintenance. The hc12sim project was renamed procsim as it was intended to be a general purpose simulation framework rather than a single purpose Freescale M68HC12 simulator. In an effort to improve modernize procsim, we designed and set up continuous integration infrastructure to effectively develop a new procsim suite. This chapter discusses changes required to an aged code base to bring support the C++14 standard, upgrading existing CMake [41] build script infrastructure and developing multi-platform environments for automated platform testing.
4.1 Modernizing hc12sim

The hc12sim project that was showcased in [40] was modern at the time of development. Since 2013, the C++ community has progressed thoroughly and many of the libraries in use within the project were superseded by the Standard Template Library (STL) on most compiler platforms. The hc12sim project utilized many poor practices such as excessive use of `typedef` statements, over use of threads or abuse of operator overloading facilities. hc12sim heavily relied on Boost’s non-standard implementations of tr1 structures such as `boost::shared_ptr<T>` [80] (now superseded by C++11’s STL `std::shared_ptr<T>` [81]) and the `boost::thread` with its associated tooling [82] (superseded by C++11’s STL `<thread>` [83]). The insidious nature of threading and smart pointer usage throughout the hc12sim software made this effort a large task. When writing C++ software, distribution of binary files is a common concern when working in cross-platform environments. Thus a conscientious effort was made to remove all large binary-based libraries in favour of simple-to-build or header-only libraries to remove concerns of finding pre-installed development packages on multiple platforms. After replacing existing Boost implementations with STL equivalents, these changes needed to be cross-platform tested to validate that the implementations by different compiler vendors on varied operating system configurations behaved as expected.

4.2 Cross-platform development

The C++ language has unofficially always tried to follow the motto: “write once, compile anywhere.” With any compiled language, when moving between platforms it is often non-trivial to assure compiled code runs between any two platforms. Given that Requirement 1 requires...
any solution must run on student’s personal computers, we required procsim run on modern\footnote{“Modern” in this context implies most recent stable long-term release.} versions of the three major platforms, Microsoft Windows, macOS and GNU/Linux. Within the C++ ecosystem, the ubiquitous way to maintain cross-platform support is to utilize only commonly used tools and depend on the Standard Template Library (STL) as much as possible. That is not to say there are not powerful cross platform libraries (e.g. Boost \cite{boost}, Qt \cite{qt}), these libraries provide excellent tools but often introduce complex build requirements and add large binary files into any distributed application. An artefact of 2013, Microsoft’s Visual C++ (MSVC) 2011 compiler and accompanying STL were not completely standards compliant with C++11 and thus to use modern C++11 STL features hc12sim required the use of Boost to get non-standard but similar implementations \cite{boost}. Easing refactoring efforts was many of the C++11 STL additions were modelled after Boost’s implementations \cite{boost}. For procsim Boost’s STL implementation dependency were mostly removed by upgrading the lowest supported compiler to MSVC 2015. MSVC 2015 fully supports C++11 in both compiler semantics and STL compatibility \cite{msvc}. This allowed us to fully remove the main Boost dependencies on Boost.Memory and Boost.Thread by effectively renaming the boost:: namespace to use std:: namespace for the components in `<memory>` and `<thread>`. By removing these library dependencies, it eased cross-platform development as only the Boost C++ headers were required to compile procsim’s core library. However, the unit test platform for hc12sim was based on Boost.Test \cite{boosttest} – a problem addressed later.
4.2.1 Access to multiple platforms within a single host platform

Once procsim utilized the C++11 STL over Boost, the next task was configuring mechanisms to ensure each platform performed correctly. It was found multiple times that different compiler’s STL implementations behaved differently depending on different versions and the compilation target in use. For example, we found that the GNU Compiler Collection (GCC) 5.X C++ STL shipped with a broken version of the C++11 <regex> library. But the GCC front-end found on macOS had no such issues as it uses libc++ provided by LLVM with a GCC compliant compiler command-line interface that runs Xcode’s version of LLVM clang. This bug was found well after feature implementation due to testing on Xcode 8.3 and MSVC 2015 passing successfully (our two main platforms). Building on Linux was tested and the issue was found through unit tests checking a feature that utilized regex searching functionality. This small but frustrating issue opened up a question: How can the software be verified working on three major platforms while not having native access to each platform?

The original approach was to change physical devices to access the different major operating systems. For each machine, we downloaded the software from the development repository and ran tests manually. It became difficult to maintain build environment changes made over time to successfully build the project as little records were maintained and compiling system projects heavily depend on environment variables. This meant sharing the software development environment became difficult and near impossible to replicate. Further, if a developer did not have access to Apple hardware, macOS software could not be tested. In previous works, we utilized VMWare vSphere technologies to create common developer environments through Virtual Machines that could be used for consistent development and testing.
Given that vSphere is a proprietary and expensive software, we researched other tools that provided virtualized environments. Oracle’s VirtualBox [94] provides virtual machine provisioning through open source tools. Building on the idea of “cloning” common configurations between development environments through VirtualBox, we eventually found HashiCorp’s Vagrant [95]. Vagrant provides a consistent declarative configuration for defining a virtual machine environments through Ruby scripts. Vagrant’s parent company, HashiCorp, hosts “boxes” that are preconfigured virtual machines ranging from Ubuntu Linux to FreeBSD to macOS El Capitain [96]. Boxes provide a base VM image that users configure to match the environment requirements for a project. When a configuration is settled, developers commit the configuration file, known as a Vagrantfile, to their repository to live directly beside the software being developed. The intention of storing environment configuration within software repositories is to attempt to merge “build” teams and development teams as these are no longer distinct processes. For example, Listing 4.1 show cases a simple configuration that installs both GCC 5.X and LLVM Clang [97] 3.7 – two fully C++11 compliant compiler tool-chains into a Ubuntu Trusty 14.04 box. Due to Vagrantfiles being Ruby programs, they have full access to Ruby’s gem ecosystem. We used Vagrant and VirtualBox to create three environments for Microsoft Windows 10, macOS El Capitain, Ubuntu 14.04 Precise and Ubuntu 16.04 Xenial. These configurations allowed instantiating virtual machines quickly and consistently on any platform to build and run test configurations. Vagrant allows users to create their own boxes and we created a box for each base configuration for distribution that would save time for setting up a machine. Our Vagrant box configurations are available at https://github.com/Nava2/procsim-bld available under the MIT Open-source License. With the three major platforms easily accessible, it became necessary to improve the build
infrastructure to be consistent across these platforms.

### 4.2.2 Building with cross-platform tools

CMake [41] is a cross-platform meta-build system. A meta-build system provides a “front-end” language for defining a project’s build and generates build scripts for other build systems to consume. CMake provides support for many different build systems, each is supported through a CMake “generator.” The generators we chose to build on the Windows, macOS and Linux were “Unix Makefiles” for GNU make [98] on Linux and macOS, “Xcode” for macOS’s IDE Xcode [99], and “Visual Studio 14 2015 [Win64]” for MSVC 2015 [100]. On Unix environments, make is generally available however and a newer build tool, ninja [101], is preferred as builds are faster and provides the same familiar behaviours as make. As stated, CMake provides a cross-platform meta-build system, CMake can not abstract direct interactions with compilers due to tool-chain and operating system specific options. For example, turning on optimization flags on GCC/clang differs from the MSVC platform. The original hc12sim project utilized CMake, but it was poorly organized and added large build crippling file dependencies. Changing a single file likely caused the entire build to trigger as if running a clean build rather than running an incremental build. At first, this was thought to be caused by CMake not properly structuring inter-file dependencies, however it was soon found to be due to poor header isolation and deep nested coupling. We separated all headers to try and isolate concerns of what a header defines – often breaking headers into multiple smaller single purpose headers. These efforts attempted to remove inter-dependencies as much as possible. Reorganization of the procsim library coupled with improved build tools and the addition of OBJECT
library files [102] significantly reduced build time and allowed for less time spent compiling procsim.

As previously discussed, compilers come with consequential differences in support for new and current standards. In order to try and utilize features appropriately it is often helpful to “test” compilers support of features. CMake provides this functionality through cmake-compiler-features [103] but CMake’s built-in support lags heavily behind current C++ standards (including C++14 and 17 at the time of writing). Fortunately, compatibility [104] is a library used to enhance compiler and STL feature detection within CMake projects. We worked with Müller to improve compatibility features and generation of header files that will include a “shim” on top of the current STL to add features where possible. “Shims” can only be added for STL features as syntax elements are not extensible in C++. To counter a lack of syntax tokens in some compilers, compatibility scripts also automatically generated C-style macros for keywords that are only available in certain C++ tool-chains (e.g. constexpr or override). These macros were used to add extra meta-information for the compiler where available to improve code generation. The use of compatibility allowed procsim to build more easily on the heterogeneous compiler platforms required to work on most personal computers for Requirement 1 while simultaneously improving the software’s performance where possible.

4.3 Testing Infrastructure

Testing a multi-platform compiled systems project is very difficult and relies on many tools to be successful. Through utilization of Vagrant and better build infrastructure within CMake scripts, procsim was able to be automatically tested on multiple platforms through continuous
integration software. Continuous integration describes a process of building, testing and deploying software as rapidly as possible to reduce time-to-market for software projects [105]. Idealistically, by utilizing continuous integration, developers receive near-instantaneous feedback on whether or not a particular change set breaks the product in unforeseen ways – this heavily depends on the level of testing and environments available. For procsim, we configured a small “micro-PC” with an Intel® i5-4590T and 4 GB of RAM to run a Jenkins [106] continuous integration server running Jenkins. The device was chosen due having CPU accelerated virtualization to aid in running Oracle VirtualBox through Intel VT-X.

Within Jenkins, Vagrant was used to create a new virtual machine for each test platform which registered as “slaves” to the “master” Jenkins service provided by Jenkins’ distributed build infrastructure [107]. Each virtual machine spawned from a separate base box to test procsim’s software test suites consistently. For example, one machine spawned a Ubuntu 14.04 machine with manually installed compiler tool-chains and a second machine spawned a macOS instance with Xcode 8. Each machine ran the exact same unit tests after configuration of the virtual machine and selection of a CMake generator. This was achieved through a Jenkins Pipeline [108] specified in a Jenkinsfile script [109]. Jenkins Pipelines allow for parallel execution of “tasks,” dependent and independent, for a continuous integration task graph. For procsim, due to its immaturity, the pipeline only included stages for building the software and running unit tests within the multiple operating system platforms required by Requirement 1 – deployment tasks were unused due to project maturity. While not strictly related to simulation Requirement 4, the addition of a continuous integration solution allowed us to maintain a rapid pace of development and provide confidence that changes made worked across all platforms without time consuming manual confirmation.
Over the course of developing procsim, the build process became more and more complicated requiring improvements in automating build script writing. When writing unit tests, the original hc12sim project utilized Boost.Test [89] for creating and running unit tests. Unfortunately, Boost.Test required library distribution which made working on platforms without official software distribution systems difficult (e.g. Windows or macOS). In order to remove the final thread of Boost binary dependence, we replaced Boost.Test with Catch [110]. Catch is a header-only C++ testing framework built on top of modern C++. Catch provides a Behaviour-Driven Development (BDD) interface that allows developers to describe the behaviour of their system within their tests creating easily understandable and clearly organized tests [111]. Catch improved the mechanisms used to test and provided an easier to maintain testing suite. Catch’s test runner requires that all tests are defined through C-style macros and the executable must include Catch’s main once per test suite through the CATCH_CONFIG_MAIN macro [112]. When defining many tests, it is often useful to organize them in meaningful “suites” of test executables. With any testing framework, when creating multiple executables it is advantageous to share object resources to reduce compiling time – particularly in template or constexpr heavy software which procsim rapidly became.

4.4 Improving developer work flows within CMake

In Section 4.2 CMake was discussed as the build tool for procsim. CMake utilizes OBJECT libraries to allow multiple “targets” to share compiled object files and statically link against the same objects as required [102]. These OBJECT libraries are not the same as shared objects or DLLs and should not be treated as such. An OBJECT library is a CMake abstraction on top of
4.4. Improving developer work flows within CMake

compilation units that must be statically linked into some other object. These shared OBJECT libraries were very useful in unit test files as a common object used between each executable “suite” is the main test runner. They provide a faster compiling but unportable binary object similar to a static library. Manually specifying all of these suite instances to get individual suites and manually maintain a global “test all” suite created significant time investment and manual adjustments whenever files were added or removed from the project. We developed a series of CMake macros and functions to reduce the required commands to define tests and library files in a “project-based” work flow [113], [114]. Within CMake, any action that has a side-effect or result is considered a target. When CMake executes a build script, it creates a graph of inter-target dependencies. From the target graph, CMake calculates a topological sort for a dependency resolution scheme used to build the project. Listing 4.2 shows a simple shared library defined utilizing the build infrastructure described. Three functions are used to define four distinct targets:

1. **new_project**() defines a “project” that logically groups libraries, executables and tests;

2. A library, `procsim` with headers, sources and external dependencies;

3. Two test “groups” of logically packaged tests for the library from (1) and adds a dependency on (1) and a common test harness;

4. An aggregate PROJECT suite that combines all the tests from each group into a suite for the current project;

Within these functions, each performs many tasks to try and simplify definition of build components to form a completed program build. This could include setting common compiler
flags, introduce dependencies on vendor libraries or generating build-time code required for the target.

The function, `new_project()` defines a set of “global” variables within the build that are named after the directory that the current defined CMake script is found. For example, if [Listing 4.2] was found in “procsim/core” the project would be called “core.” This behaviour is meant to be consistent and simplistic between project definitions. This project model is based on the idea of “convention over configuration” which was popularized by tools and frameworks like Apache Maven and Ruby on Rails [115], [116]. For example, `new_project()` defines the following configurable global variables:

- **SOURCE_DIR**: Implementation files directory, defaulting to “core/src”
- **INCLUDE_DIR**: Header include directory, defaulting to “core/include”
- **TEST_DIR**: Test implementation directory, defaulting to “core/test”

The *DIR properties allow for inner-project commands to use relative paths to simplify declaration of their file dependencies. These properties are used to generate “namespaced” build targets and variables allowing for use in specification within CMake’s dependency graph. To simplify, if you define two projects, a and b, you can set b to depend on a being built first. In [Listing 4.2] all of the files are specified relative to the “project directory” removing repeated values required in standard CMake scripts. Once the `new_project()` function is called within a cmake script hierarchy, all further functions from the build tools will utilize the state set by the project.

The function `new_library()` creates a new library target that compiles all of the specified files to create a library. The function accepts SHARED, STATIC, OBJECT, etc. to mimic
new_project()

# Define a new library called `procsim`
new_library(procsim
    HEADERS procsim/encoding/Algorithm.hpp
    procsim/encoding/Code.hpp
    procsim/encoding/Operand.hpp
    procsim/encoding/Primitives.hpp
    procsim/encoding/Utility.hpp
    procsim/time/Clock.hpp
    procsim/time/Timer.hpp
    procsim/time/AsyncTimerReceiver.hpp

    SOURCES encoding/Algorithm.cpp
    encoding/Code.cpp
    encoding/Operand.cpp
    encoding/Primitives.cpp
    time/Clock.cpp
    time/Timer.cpp
    time/AsyncTimerReceiver.cpp

    GENERATED_HEADERS ${PROCSIM_EXPORT_HEADER}
    ${PROCSIM_COMPILER_HEADER}
    VENDORS cpp17_libs fmtlib # non-builtins
    INCLUDE_DIRECTORIES ${Boost_INCLUDE_DIRS}
    LIBS ${CMAKE_THREAD_LIBS_INIT}) # builtin libs

# Create two test suites utilizing the new library
create_test(time SOURCES time/ClockTest.cpp
            time/TimerTests.cpp
            LIBS procsim test-harness)

create_test(encoding SOURCES encoding/CodeTest.cpp
             encoding/OperandTest.cpp
             encoding/PrimitivesTest.cpp
             encoding/UtilityTest.cpp
             LIBS procsim test-harness)

# Create a test for the whole project
create_test(core PROJECT)

Listing 4.2: CMake script showing how two test suites, `time` and `encoding`, are defined as part of a project, `core`
add_library()’s syntax [102] – an attempt to remove some discontinuity between the two functions. As shown in Listing 4.2, there are several other parameters used:

- **HEADERS** Header files
- **SOURCES** Implementation files
- **GENERATED_SOURCES** Source files that are generated at build time by another target
- **GENERATED_HEADERS** Same as GENERATED_SOURCES except for header files
- **VENDORS** These are external dependencies that are built within the build chain after downloading from an external source. This forces the library to depend on their download and build
- **LIBS** These are internal CMake libraries or other targets that are built in a multi-project configuration

Utilizing these parameters, the function creates an appropriate CMake target using add_library() and adds the required dependencies to the new library target. In addition, it performs platform-specific compiler adjustments to try and reduce the amount of configurations required per target. For example, many flags for MSVC to not apply to Clang which do not apply to GCC. This torrent of compiler-specific option configuration is eased by the use of CMake’s generator expressions [117] but many compilers do not properly identify to CMake’s internal infrastructure so manual adjustments must be made for newer tool-chains. By utilizing our own mechanism, the targets created have “localised” properties such as include directories and target_properties() added in reproducible ways that do not accidentally pollute
other targets or the global variable space. For example, if two targets require different versions of the same files, the built-in `include_directories()` command could cause these targets to “collide” and fail to compile [118]. `new_library()` appropriately applies the SYSTEM option to all includes that are outside of the multi-project build so any warnings generated are not applied. Lastly, `install` targets are created that install headers and compiled binaries to appropriate locations depending on the type of library and the platform in use. For example, on Microsoft Windows requires libraries be beside their executables, where UNIX utilizes PATH resolution mechanics.

The last function used in Listing 4.2 is `create_test()` which has three distinct incantations. The first creates a single test suite from Catch-based test sources [110]. These sources are aggregated and compiled to an OBJECT library [102]. The test objects are aggregated for future linking into the current PROJECT’s test executable. By generating OBJECT libraries, these smaller units of compilation can be combined and reused intelligently saving large amounts of compile time as previously discussed. The second pattern as `create_test(test PROJECT)` generates a PROJECT-based test executable and “check” target that runs the PROJECT test executable. This test executable includes all previously defined OBJECT libraries in the current “project.” This executable is used to debug and execute tests for the given project, it is currently not possible to run a smaller test individually without utilizing the selected test runner’s (Catch) command-line interface manually. Additionally, the generated executable is registered with CMake’s CTest which creates build-script targets to run test executables providing dashboards and other useful tools [2].

The last use case is `create_test(target ALL)`, not included in Listing 4.2. The ALL request generates a test executable that includes every test suite created in a multi-project build.
into a single executable – a convenience executable. As multi-project programs develop, it is useful to be more or less granular with the scope of tests executed depending on the context required at the time of use. While intuitively it appears as though large amounts of executables create large amounts of compilation units creating a very slow build time, due to the use of OBJECT libraries created at definition, adding more test executables only adds incremental link time due to linking static objects – a marginal increase of time for the convenience provided. Summarized test output from procsim’s generated ctest command is shown in Listing 4.3. The output is easy to read and, should errors occur, Catch provides assertion errors and suites will continue summarizing all errors after a suite completes.

All of these build improvements worked towards improving software maintainability of the simulation suite. Whilst these changes do not directly contribute to the Problem Statement in Section 1.3, it is patently obvious that improvements in developer work flow have a direct consequence in improvements to developer time utilization and reduces the likelihood of poor software being produced. Given improvements to developer productivity, one may argue simply that this improves the validity of producing modern software (Requirement 5) and any software produced will more accurately meet requirements specified [111]. As such, these improvements allowed us to develop procsim more rapidly and reduced time spent fighting against compilers and cross-platform development issues.
4.4. Improving developer work flows within CMake

--- Build started: Project: RUN_TESTS, Configuration: Debug x64 ---

Test project S:/research/procsim/build-windows

Start 1: core-components
1/11 Test #1: core-components ............... Passed 0.09 sec

Start 2: core-encoding
2/11 Test #2: core-encoding ................. Passed 0.06 sec

Start 3: core-time
3/11 Test #3: core-time .................... Passed 4.46 sec

Start 4: core-misc
4/11 Test #4: core-misc .................... Passed 0.09 sec

Start 5: conf-encoding
5/11 Test #5: conf-encoding ................. Passed 0.20 sec

Start 6: conf-loader
6/11 Test #6: conf-loader ................... Passed 0.06 sec

Start 7: conf-arch
7/11 Test #7: conf-arch .................... Passed 0.07 sec

Start 8: conf-components
8/11 Test #8: conf-components .............. Passed 0.13 sec

Start 9: conf-proc
9/11 Test #9: conf-proc .................... Passed 0.18 sec

Start 10: conf-time
10/11 Test #10: conf-time ................. Passed 0.07 sec

Start 11: conf-env
11/11 Test #11: conf-env ................... Passed 0.04 sec

100% tests passed, 0 tests failed out of 11

Total Test time (real) = 5.50 sec

====== Build: 1 succeeded, 0 failed, 1 up-to-date, 0 skipped ======

Listing 4.3: Test output from CTest [2] from Microsoft Visual Studio Community 2015 for procsim showing organization provided by CMake configuration.
Chapter 5

Lua-based configuration-driven processor simulation

From working with Scala-based configurations in Section 3.4.1, we believed that runtime-configuration was extremely important in encouraging students to try and manipulate processor designs. The author chose to expand the hc12sim project’s the JSON-based, compile-time configuration capabilities replacing the mechanism with runtime-based configurations utilizing an embedded scripting language. Within the C++ programming ecosystem, there exists many different scripting environments that can be embedded with varying degrees of difficulty, feature capabilities and execution speeds. The author investigated utilization of several scripting environments before settling on utilizing Lua [119] with bindings provided through sol2, a wrapper between C++ and Lua API calls [120]. Once Lua was selected, the author designed configuration schemas built to execute in a Lua sandbox with timing and other hardware-level considerations abstracted away from the configuration definitions as much as possible.
5.1 Utilizing runtime configurations through scripting

procsim.scala had a VHDL-like syntax for defining instructions within a processor simulation. Providing the same facilities within a non-managed language like C++ is not possible without the use of a scripting engine. When researching possible solutions to this problem, several scripting languages stood out: Python [121], JavaScript through Google’s V8 [122] or Mozilla’s SpiderMonkey [123], Lua [119], or reusing Scala through the Java Native Interface (JNI) [124].

5.1.1 Scripting language selection

Python

Several scripting languages were evaluated against each other for candidacy as the configuration specification language. First, Python was considered [121]. Python is a mature, stable, dynamically typed language with wide use in industry, scientific and academic communities [36]. Python was considered due to its exposed foreign function interface (FFI) which allows for C functions to be called from Python or C functions to call into Python. Python’s syntax supports overriding operators [125], object-oriented programming [126] and low-level bitwise operations [127] – features excellent for implementing low-level hardware simulations and configurations. For working with the Python VM, the Python community provides a library called CFFI that wraps python’s FFI interface to call into and from Python. CFFI’s purpose is to provide a bridge layer between Python and C easing the effort required. Any library wishing to utilize CFFI for interacting with the Python engine requires all exposed functions in the foreign library are externalized as C functions. procsim heavily employed C++11 syntax
and does easily extend to a C-based API through its use of `template, inline` functions and `constexpr` values or functions. Thus porting procsim’s API as a C API to utilize CFFI would become tedious.

An alternate to the CFFI library is Boost.Python which provides binding mechanisms for working with complex classes in C++ and having them work within the Python virtual machine [3]. Boost.Python provides a very simple and easy to use syntax for defining types. Further, Boost.Python respects constructor and destructor semantics of any types passed into Python. **Listing 5.1** showcases a simple type exposed into Python. Given the simplicity of exposing data into Python, and the ubiquitous nature of the language, it lends itself well to a pedagogical application ([Requirement 3]) and will feel modern to students coming from Software Engineering contexts ([Requirement 5]). Regrettably, the largest compelling argument against Python is that to run Python scripts, one must install a Python virtual machine adding an extra dependency that is external to the project. This makes distribution more difficult for personal computers, though not all as some operating systems come with python pre-installed (e.g. most Linux distributions and macOS).

**JavaScript**

JavaScript is by far the most popular language in use in modern applications [36]. JavaScript provides many of the same features as Python, but does not have as “strong” of a type system. JavaScript provides many coercions of types into other types making it difficult to reason at times compared to Python (e.g. `{ } + [] === 0` for reasons outside the scope of this document). While providing a modern interface for students ([Requirement 5]), it also creates a regrettable pedagogical experience due to the extremely high-level nature of the language for
5.1. Utilizing runtime configurations through scripting

Listing 5.1: Example of exposing a C++ class to Python [3].
low-level implementations (Requirement 3). In order to embed JavaScript within an application, it requires a JavaScript Virtual Machine to be embedded. The two current leading JavaScript engines are Google’s V8 [122] powering Google Chrome and Mozilla’s SpiderMonkey [123] for Mozilla Firefox. SpiderMonkey provides a C++ API to embed functions and values into the engine. This C++ API is very similar to the API provided by Python’s CFFI, however it provides slightly stronger type safety than a traditional C API utilizing `void*` arguments. When investigating to integrate the SpiderMonkey VM into procsim, an adverse design of SpiderMonkey is that the VM’s entire state is bound to a single thread of execution [128]. JavaScript itself is a single-threaded language which is largely inconsequential for configuration declaration. Though due to SpiderMonkey itself being bound to a single thread, it may become too difficult to efficiently produce multi-threaded software when accessing SpiderMonkey JavaScript code at runtime.

Google’s V8 JavaScript engine is the most commonly used JavaScript engine as it powers Node.js, all Chromium-based browsers and Electron-based applications [122]. Embedding V8 is a difficult task requiring large amounts of “glue” code to bind components [129]. Unlike SpiderMonkey, Simplified Wrapper Interface Generator (SWIG) provides a V8 wrapper [130]. However, direct utilization of the V8 API is recommended. To use V8 in an application, it must be built and the library contains non-trivial build process per platform. Removal of non-trivial built libraries were removed as part of the work for Section 4.1 thus adding a new complicated library proves contradictory to efforts previously made. To avoid the complicated build steps, for users to utilize V8 within procsim the entire engine would need to be shipped with the application or installed by users. Providing a working “drop-in” source for V8 or SpiderMonkey is unfortunately not feasible. As such, utilizing V8 or SpiderMonkey is not
advantageous for procsim as an JavaScript is too high-level and the available engines are too large with large build dependencies.

**Java and Scala**

Java and Scala both require the Java Virtual Machine to execute software. Any application that has a JVM requirement involves either 1) packaging and shipping the entire JVM with the application or 2) expecting the user has a correctly installed JVM in default location. Both of these dependency resolution schemes for the JVM are difficult to complete with absolute certainty, but are not impossible. The JVM itself is also several hundred mebibytes in size with a large runtime overhead. We considered integration with the JVM because we had the existing DSL created within the procsim.scala project with a VHDL-like syntax that we could reuse. In order to access Scala or Java libraries outside the JVM, software wrappers must be written to utilize the Java Native Interface (JNI) [124]. The JNI is notoriously frustrating to write software for because C/C++ is not a managed language and the JVM is a managed VM making memory guarantees frustrating to correctly implement. Additionally, the JNI was developed to be efficient for software to communicate between two environments, it was not written to allow the process to be easy. Java developers are now discouraged from writing native libraries that interact with Java where possible as the JVM has significantly improved performance characteristics eliminating the largest use case for native libraries. SWIG provides an excellent wrapper definition tool to generate bindings for C++ classes into the JNI [130] allowing developers to write generator files using a C++-like syntax which includes extra meta-information that is added to appease the Java Virtual Machine [131]. SWIG uses “directives” to annotate existing C/C++ code to generate efficient JNI code that can then be compiled into a
C/C++ application as any other software. SWIG does not support method references or lambda expressions which proved problematic when developing a communication layer. In addition, working with SWIG wrappers it non-trivial as classes become more complicated. SWIG has not updated over time to keep up with C++ standards and does not directly support many semantics such as `std::unique_ptr` and lambda expressions. With the additional overhead of utilizing the JVM, reusing the older procsim.scala DSL is not feasible as an embedded scripting language. Java is not designed to be embedded. Java was designed to be it's own managed runtime and have foreign native code be given selective access to run within the JVM. Due to large overhead costs at both build and runtime, we discarded Java as a viable scripting language.

**Lua**

Lastly, the Lua programming language was considered as it is by design an embedded scripting language [119]. The Lua Programming Language home page directly states:

> Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description. [119]

These features fully encompass those features previously provided by Python while adding additional simplicity. Lua places data description as a first order citizen allowing for extremely descriptive dictionaries of heterogeneous data [4]. Similar to JavaScript’s Object, Lua provides traditional object-oriented programming through an associative arrays known as tables [4], [132]. Listing 5.2 shows a simple dictionary-like description of a “project entry” that the Lua
community uses to display known projects that use Lua \[133\]. Creating tables in Lua are a trivial task and through external libraries, such as tableshape, the schema or “shape” of a table can be quickly validated \[134\].

Lua’s design is first and foremost an embedded scripting language. Interaction with Lua’s VM is through direct calls to Lua’s C API. Many wrapper generation tools exist to allow higher-level binding than hand-written C API calls. SWIG provides a wrapper for Lua, but it has the same concerns as the Java generator discussed previously. For wrapping the procsim library, we considered sol \[135\] and luacppinterface \[136\] as they provided the best integration with C++ at the time. Both sol and luacppinterface provided tested interfaces for working with the Lua VM from C++, however sol provided more modern bindings for emerging C++ standards. Lua’s syntax is very similar to other C-style languages and is commonly utilized in games world-wide showcasing the modern nature and performance characteristics of Lua (Requirement 5). In addition to the traditional Lua engine, there is a hand-written JIT version of Lua known as LuaJIT. LuaJIT is extremely fast and provides machine-compiled Lua scripts at execution. Further, LuaJIT’s C API is a superset of Lua’s meaning it is fully compatible.

Lua’s syntax is extremely simple but provides flexibility with the ability to override all operators. This meant that the style of configuration intended for use with procsim.scala could be roughly ported to use within a Lua environment. When defining configurations, Lua is still a full programming environment with all of the power of a programming language adding a new layer to configuration capabilities for Requirement 2. Lastly, through LuaJIT, Lua is the near best performing scripting language after JavaScript. Lua provides a very simple interaction point thanks to wrapping libraries and Lua’s embedding first design philosophy leaves it small, unbloated and easy to compile and package.
Listing 5.2: Table used to describe project information for the Lua.org site [4]

Scripting language decision

After consideration of the four different scripting languages, it was narrowed to a choice between Lua and Python. Boost.Python’s long history of use coupled with existing Boost usage in procsim made Python an enticing option. In addition, Python’s feature set is very complete for most of the known use cases for procsim’s configuration design requirements. A large issue with Python was that Boost.Python proved to be non-trivial to cross-compile on multiple platforms and imposed a direct coupling to the sizeable Python runtime. Further, to have simple configurations `dict` literals are required which provides less type safety guarantees than Lua’s equivalent through tables validated with `tableshape` [4], [127], [134]. Lastly, Python’s garbage collection can become difficult to work with in a non-managed language as it was not built with embedding in mind. Lua’s focus on embedded scripting interfaces; its wide-spread use in games and industrial software as a scripting interface; the simplicity to build the C-based JIT or interpreted VM; and data description capabilities drove the decision to utilize Lua for configuration within procsim.
5.1. Utilizing runtime configurations through scripting

5.1.2 Lua integration

Once Lua was chosen as a scripting language, the task of integrating the language into proc-sim’s existing object model arose. As stated previously, there existed two main projects under consideration for wrapping C++ API into a scripting engine: luacppinterface and sol \[135\], \[136\]. When first attempting to implement a scripting interface, luacppinterface was chosen because it was simpler than sol. However due to the simple features available and the API design of luacppinterface, binding large class instances became overly complicated and runtime performance severely degratted. Thus, we migrated the software to sol. sol provided an elegant and still relatively simple C++11-based interface to wrap classes and functions to expose them to a Lua environment without the use of problematic C-style macros at zero runtime overhead cost \[135\].

When working with sol, several bugs were found in the implementation of the wrapper. Of significant importance was in sol’s current state, it could not pass a table to a C++ constructor \[137\] – a feature we planned on utilizing heavily. This author raised the issue and after several weeks of remaining open, the author of sol never responded. However, another GitHub user by the name of “ThePhD” revived the project under a new name, sol2 (herein referred to as sol) \[120\]. sol was updated to include support for C++14 features and heavily relied on inline variadic template functions and SFINAE structures \[138\] to provide extremely efficient zero-cost bindings to Lua in a clean elegant syntax \[5\], \[139\]. In Listings 5.3, 5.4 and 5.5 sol2’s capabilities are shown to simply bind the class player into Lua and utilize it within a simple toy script. Binding class properties is provided by utilizing sol::property(...) and sol::readonly(...) allowing a pattern very similar to Java Bean properties \[140\]. The
flexibility of overriding the operations allows for the creation of new “syntax-like” changes that alter the traditional behaviour of Lua operations as done with procsim.scala. While overloading operators is something heavily debated, the intention of changing “typical” behaviours is to reduce the amount of effort required by students. If the reading context remains consistent with traditional behaviour, a transparent side-effect should not increase the amount of knowledge required to use the system. The Java-bean style “getter” and “setters” let developers add or change side-effects of how an action works transparently to users. Without the wrappers provided by sol, this task is very difficult as it involves manually modifying Lua’s meta-tables [141].

Working with sol was not entirely without problems. Because sol was actively developing, we worked closely with the new author of sol, “ThePhD,” to implement features and verify behaviours across multiple platforms (a requirement deeply discussed within Section 4.2). We implemented tests and benchmarks on the macOS platform as “ThePhD” did not have access to the platform. We worked to add continuous integration support for sol through Travis-CI [142] to confirm the libraries support on platforms required for procsim [143], [144]. The contributions surrounding continuous integration improvements were re-purposed configurations from procsim provided as open-source contribution to sol. Within procsim, the development heavily tracked the cutting-edge of sol development. By tracking sol closely, performance fixes and improvements were rapidly integrated into our implementation. Problematic ergonomics were discussed and often implemented and a symbiotic relationship was built that benefited implementation of procsim and sol.
5.1. Utilizing runtime configurations through scripting

```cpp
class player {
public:
    int bullets;

    player(): player(3) {} 

    player(int ammo) : bullets(ammo), hp(10) {} 

    bool shoot () {
        if (bullets < 1)
            return false;

        --bullets;
        return true;
    }

    void set_hp(int value) {
        hp = value;
    }

    int get_hp() const {
        return hp;
    }

private:
    int hp;
};
```

Listing 5.3: class player that holds two fields, one for hitpoints (hp) and one for bullets a player has (adapted from [5]).
-- player_script.lua

-- call single argument integer constructor
p1 = player.new(2)

-- p2 is still here from being
-- set with lua["p2"] = std::make_shared<player>(0);
-- in cpp file
local p2shoots = p2:shoot()
assert(not p2shoots) -- had 0 ammo

-- set variable property setter
p1.hp = 545;
-- get variable through property getter
print(p1.hp);

local did Shoot_1 = p1:shoot()
print(did Shoot_1)
print(p1.bullets)
local did Shoot_2 = p1:shoot()
print(did Shoot_2)
print(p1.bullets)
local did Shoot_3 = p1:shoot()
print(did Shoot_3)

-- can read
print(p1.bullets)
-- would error: is a readonly variable, cannot write
-- p1.bullets = 20

Listing 5.4: Utilize the class player within a Lua script (adapted from [5]).
#include <sol.hpp>

int main () {
    sol::state lua;

    // Register usertype metatable
    lua.new_usertype<player>( "player",
        // 2 constructors
        sol::constructors<player(), player(int)>(),

        // member function that returns a variable
        "shoot", &player::shoot,

        // gets or set the value using member variable syntax
        "hp", sol::property(&player::get_hp, &player::set_hp),

        // can only read from, not write to
        "bullets", sol::readonly( &player::bullets )
    );

    // set a variable "p2" with a new "player" with 0 ammo
    // using an std::shared_ptr<
    lua["p2"] = std::make_shared<player>(0);

    // run the example script
    lua.script_file("player_script.lua");

    const std::shared_ptr<player> p1 = lua["p1"];
    std::cout << p1->hp << std::endl; // prints 545
}

Listing 5.5: Required bindings to allow utilization of class player from Lua (adapted from [5]).
5.1.3 Configuration versus Simulation entities

The original hc12sim project shared all modules between both configuration and simulation. This meant less lines of code to understand and generally made the software “easier” to write for novice developers. However as modules grow in size, they become more complicated to maintain. Something found in work for Chapter 3 was that immutable objects produce easier to reason and better performing software. In developing configuration entities to specify simulation entities, it was found that a lot of meta-information required to make configurations easier to specify could be removed once the system was fully specified. The meta-information was provided to students to reduce their cognitive load but does not provide value to a simulation. Most values within the configuration entity need to be mutable within a configuration file, but within the simulation these values are immutable allowing for simpler testing of a simulation engine.

For example, when defining a Register component, there are a several values that are required when configuring an instance:

- name - A logical name
- clock - The clock that this Register is bound to (Registers are sequential)
- readCount - Number of clock ticks this takes to read a value (required to “slow” down execution)
- writeCount - Number of clock ticks to write a value to the Register
- access - What type of access this register supports, i.e. read or write
When writing a configuration, it is easier to define a clock by its name, a literal definition or by utilizing a local value. By contrast, at simulation time the clock must be a reference to an existing instance from somewhere. This idea of “higher-level configuration” that is transformed into a “lower-level” representation directly mimics a compiler’s approach when compiling high-level code to the compiler’s intermediate representation. The intermediate representation removes superfluous information in exchange for a smaller but easier to optimize representation of the program’s state which is how simulation entities are represented. We have included most of the class definitions for simulation entities in Appendix A.

5.2 Lua-based Configuration

To limit the scope of a system’s configuration, procsim only exposes the following components: registers, memory banks (ROM, RAM), clocks, and arithmetic operations. Simplification of the model allows for reduction of complexity in an attempt to reduce the cognitive load for students when creating their own processors, similar to the approach taken by [39] and [31]. This trade-off removes excessive modules in favor of simplifying interfaces to improve pedagogical outcomes (Requirement 3). The minimal canonical example of a Turing-complete machine is URISC, popularized by [6] and Figure 5.1 shows the hardware architecture of URSIC which Listings 5.6 and 5.7 was based off of which we use as a referential base for configuration specifications.

Because procsim’s configurations are written in Lua, they are fully Lua compliant scripts in every aspect. In Listing 5.6 on line 4, a local variable is defined that is used throughout the processor definition. While defining a numerical value of four is trivial, this potential
Figure 5.1: URISC hardware architecture with microcodes [6].
5.2. Lua-based Configuration

```
-- variable defined for convenience
local wordWidth = 16

return {
    proc = Proc.new {
        name = "urisc",
        clock = Clock.new {
            name = "cpu",
            period = millis(1),
        },
    },

    -- Registers within the CPU
    registers = {
        PC = Register.new {
            width = wordWidth,
            access = Access.ReadWrite,
            clock = "cpu", -- Previously defined
            readCount = 1,
            writeCount = 1
        },
        -- Buffer for ALU results
        R = Register.new {
            width = wordWidth,
            access = Access.ReadWrite,
            clock = "cpu",
            readCount = 1,
            writeCount = 1
        },

        -- Define a memory mapping
        memory = {
            0x0000 = Memory.new {
                width = wordWidth,
                length = 16 * 1024, -- 16Ki
                access = Access.ReadWrite,
                clock = "cpu",
                readCount = 1,
                writeCount = 1
            },
        }
    }
}
```

Listing 5.6: Configuration of Figure 5.1 for the URISC processor [6].
-- Instruction table specification

instructions = {
    -- SUBLEQ
    SUBLEQ = Instruction.new {
        -- AAAA BBBB PPPP
        code = Code.new(0, -- no constant values
            {
                A = u16(15), -- A operand
            }),
        -- Word width is 16, so operands need to be
        -- defined outside the opcode
        op = {
            B = u16(31), -- B operand
            P = u16(15) -- JUMP address
        },
        -- Function to execute
        exec = function (proc, operand)
            local B = operand:B -- alias
            -- Store A into "rhs" of operation
            proc:R = operand:A
            proc:memory[B] = B - proc:R -- *B <- B - A
            if (proc:N) { -- Negative val from ALU
                proc:PC = operand:P -- Jump as result was < 0
            } else {
                proc:PC++ -- Next ins
            }
        end
    }
},
}

Listing 5.7: (Continued) Configuration of Figure 5.1 for the URISC processor [6].
5.2. **Lua-based Configuration**

expands to allow for definitions that include loops and other control flow statements to simplify
definitions of large or iteratively built module configurations. This mechanic defines why these
configurations are so powerful, the example of a variable declaration is a shamefully inadequate
description. The idea to utilize Lua as a means of “configuration as code” was inspired by
previous work with Google Guice through its `Module<T>` syntax [145] and annotation driven
serialization with Jackson Annotations [146].

We use the Lua scripting engine to directly execute a specified configuration as a Lua script. When
the script executes, it must create and return a `Proc` instance. While the script executes,
any instantiations of types are projected through Lua into the C++ configuration entities as
described in Section 5.1.3. These entities have properties that can be easily read and reused
within a configuration script. Once the full configuration is returned, the `Proc` and it’s children
must be converted into near-immutable simulation entities. We have included the definitions
of the simulation classes within Appendix A. We elaborate on how each of these configuration
entities are defined within configurations in further discussions.

Moving further into the URISC example in Listings 5.6 and 5.7, all objects use a static `new`
method that accepts a Lua table of configuration values. This approach was utilized to mimic
JavaScript’s colloquially named “option object” of parameters to make parameter values as
obvious as possible by using explicit keys. For each table-based function, the keys and values
are validated against an expected schema – invalid or missing keys or bad value types throw
informative exceptions for students to recover from. All instances within the configuration
schema inherit from `class ConfObj` which defines that every instance must have a name.
These names are utilized to lookup references in other referenced modules. From these names,
the “compilation” of the configuration utilizes a two-phase compilation pattern inspired by a
typical assembler pattern. The first phase runs across the entire structure and creates a list of required references attempting to verify any named reference has an associated configuration object. This table of references are sorted using a topological ordering such that every module is built before it is required. Each configuration value is converted through sol2 bindings to create non-configuration entities

**Clock.** Breaking down each component type, first is definition of a Clock, repeated in [Listing 5.8](#): Clock modules must have a :period or :frequency – defining one implies the other. These each have useful “helper” functions that allow for defining relative values. For :period, the helper functions: seconds(), millis(), micros() and nanos() define times relative to nanoseconds. For :frequency the style of functions exist based on Hertz (e.g. MHz(), or KHz()). These Clock modules generate instances that produce “tick” operations based on the :period and allow synchronous modules to listen for events.

```lua
10  name = "cpu",
11  period = millis(1),
12  }
```

Listing 5.8: Clock configuration for the processors main clock (cut from [Listing 5.6](#)).

**Register.** In procsim, a Register is a simple memory unit that stores a single value. The PC definition is shown in [Listing 5.9](#): The :width parameter represents the bit-width of the Register’s stored value. access is a set of enumeration values that are used to define “getter” and “getter” methods for the “value” of a Register. If the :access parameter is

---

1In Lua, the object method invocation syntax is: `object:method` which passes `object` as the `this` parameter. For the remainder of this document, the `this` parameter is omitted for brevity and is implied when prefaced by `:`.
Access.ReadWrite then the value may be set or read. Alternately if the access for an example Register R is Access.Write then the following instruction code will not compile in the Lua interpreter: `local r_plus_1 = proc.R + 1`. By setting the :access to Access.Write the value can not be read from the register – effectively disconnecting the Q signal in traditional Register diagrams. procsim achieves this behaviour by not binding a “setter” or any arithmetic operations for the Register type bound through sol to the property for proc.R parameter utilized when defining an Instruction execution (see Section 5.3.2).

The :readCount and :writeCount parameters define how many clock “ticks” it takes to read or write to the Register respectively. These values are utilized by the execution engine to schedule the triggering of events.

```lua
width = wordWidth,
access = Access.ReadWrite,
clock = "cpu", -- Previously defined
readCount = 1,
writeCount = 1
},
-- Buffer for ALU results
```

Listing 5.9: Register configuration for the program counter (cut from Listing 5.6).

**Memory.** The second last module type available are Memory units. Memory are either read-only or read-write like Register values. Listing 5.10 shows the RAM definition for the URISC architecture. Parameters are identical to a Register except that they additionally have a :length parameter that defines the length of a Memory in units of :width (the word width). In the sample, the memory is 16KiB “words” wide. With a RAM, depending on the type of memory being modelled, it is often useful to have the :clock parameter be slower
than the processor clock. This models the exponentially different time taken to read from a
Register rather than cache or off-chip memory.

```lua
width = wordWidth,
length = 16 * 1024, -- 16Ki
access = Access.ReadWrite,
clock = "cpu",
readCount = 1,
writeCount = 1
}
```

Listing 5.10: Memory unit configuration for the program counter (cut from Listing 5.6).

**Proc.** The second last module defined is the processor itself through the `Proc.new()` function. The `:clock` parameter defines the clock utilized by the processor itself. This value is not shared amongst internal modules unless explicitly named by another module. The `:registers` table defines Register values that live “on-chip” within the created Proc. The `:memory` table defines a sparse-matrix representation of the processor’s memory mapping. Any inline defined Memory or Register components are given the name of the key in the table they are specified in. This is a small convenience for students to not replicate information which risks typographic mistakes. Within the `:memory` map, any memory unit may be defined at an “address” specified.

In the URISC machine, there are no memory mapped registers, however [Listing 5.11] displays a small processor with three Register values mapped to the first three addresses and a RAM mapped starting at address $0x0004$. Specification of `:memory` mappings take into account the width of the memory elements specified and will do partial reads or writes depending on what is required. The current implementation does not support memory elements smaller than the word size. The `:instructions` table provides a look-up table of instruction information in-
5.3 Instruction definition

The Instruction definition is more complicated than any other module and where the best benefits are found from a programmatic configuration. Within the proc:instructions table, the :name of the Instruction is calculated based on the value of the key an Instruction is assigned to, mimicking the proc:registers table.

5.3.1 Opcode and Operand encoding

procsim allows for easily defining complex encodings for opcode and operand values. Instruction:code specifies the opcode of an Instruction. A Code definition describes values encoded within an opcode a traditional microarchitecture. Listing 5.12 shows a cut of the encoding definition for the URISC SUBLEQ Instruction. Within the :code parameter, the Code construct defines an opcode. The first parameter of Code’s constructor is a constant value with bits that remain constant in all encodings of the current instruction – both high and low values are. The second parameter is a table of “fields” that are encoded within the opcode when compiled. Any key passed into the table will be extracted from the opcode at decoding time and passed into the execution handler attached as an “operand field.” With both the constant opcode and the table of fields, Code.new{} calculates the constant bits of the opcode encoding. Once all Instruction values are defined for a Proc, the configuration engine computes a discriminant decoding schema for runtime. When defining a field, there are several
-- Memory mapped machine

-- Many irrelevant configuration parameters are
-- ommitted to reduce this listing size.

local A = Register.new {
  width = 8,
};

local B = Register.new {
  width = 8,
};

local X = Register.new {
  width = 16,
};

return {
  proc = Proc.new {
    name = "memory-mapped",
    clock = Clock.new {
      name = "cpu",
      period = millis(1),
    },
  },

  -- Registers within the CPU
  registers = {
    A = A,
    B = B,
    X = X
  },

  -- Memory map for addressable registers
  memory = {
    0x0000 = A,  
    0x0001 = B,  
    0x0002 = X,  -- two words wide
    0x0004 = Memory.new {
      width = 8,
      length = 16 * 1024,  -- 16k
    },
  }
}
helper functions in use. Line 51 of Listing 5.12 includes the helper function, \texttt{u16(15)} which creates an unsigned 16-bit field at index 15\footnote{While Lua uses a 1-based indexing scheme like MATLAB or FORTRAN, procsim uses zero-based indexes to match C.}—indexed from the least-significant bit to the most-significant bit. The naming scheme for the primitive field helper functions are named using the Rust programming language’s names for primitive types as they are short and unambiguous in their context \cite{147}.

```python
code = Code.new(0, -- no constant values
    {
        A = u16(15), -- A operand
    }),

-- Word width is 16, so operands need to be
-- defined outside the opcode
op = {
    B = u16(31), -- B operand
    P = u16(15) -- JUMP address
},
```

Listing 5.12: Encoding of the opcode and operands of the SUBLEQ Instruction (cut from Listing 5.6).

The decoding scheme for a processor specification is computed from the table of Instructions in \texttt{Proc:instructions} by computing individual discriminants per instruction and then cross-checking each discriminant against other instructions in a pair-wise fashion. To check if two discriminants are ambiguous, mask each “constant” opcode with another’s “constant mask” and vice versa and if the resultant values are equal, the constant values are equal implying there is an ambiguous opcode specified. Algorithm 1 details the full computation used to verify ambiguous opcode specifications. If an ambiguous opcode is found within the instruction table an exception is thrown alerting the user which instructions have ambiguous
conflicts and which constant bit values are shared. Unfortunately, this verification algorithm operates in a \( O(n^2) \) complexity as each operation must be compared to another. A solution utilizing a radix-based sort could reduce the search space for each constant operand to make the algorithm \( O(n \log n) \) \[148\] – though the complexity of each step within this computation is of trivial time due to consisting of two bitwise operations, making the trade-off between readable and fast code inconsequential. This specification scheme for encoding allows for extremely flexible ISA design but also provides a mechanism to allow students to find and recover from mistakes. By using this simplistic specification system and providing feedback for students, students will have an easier time learning why certain ISA design schemes do not work.

Operands for an Instruction are specified within the :op parameter table. :op accepts a table of key to field specifications that behaves identically to the second parameter of Code.new(). Unlike the :code parameter, there is no verification for operand values. Keys within the :code and :op tables are checked for uniqueness and Lua failed to compile if a table has non-unique keys. The table values are unioned and their widths are computed, storing the total size of the operand for encoding into machine code. As with :code, the keys are extracted and from the compiled encodings and passed to the execution function.

### 5.3.2 Instruction execution: Side-effect-based compilation

The intention of the Instruction::exec() function is to provide the schema for how an Instruction executes within the context of a simulated processor and operands passed. When students provide a :exec() method, they are not writing the function that will directly execute within the engine. procsim.scala discussed utilizing an event-based execution schema that
Algorithm 1 ISA opcode verification algorithm to find ambiguous opcode definitions

**Input:** $ins = $ Array of Instruction definitions  
**Output:** Array of Instruction definition pairs with ambiguous opcode values.

**Step 1:** *Compute a discriminant mask for each Instruction*

1. Define a buffer for computed discriminants and masks
2. $insBuffer ← []$
3. for all $i ∈ ins$ do
4.   Build a mask for field location in the opcode
5.   $fieldMask ← 0$
6.   for all $field ∈ i.fields$ do
7.     Compute the field’s bit mask
8.     $currentMask ← (mask (field.width) $\ll$ field.index)$
9.     Mask the current field’s mask with the overall field mask
10. $fieldMask ← field.mask | currentMask$
11. end for
12. Compute the constant bits of the opcode, bits that are not part of fields
13. $opcodeMask ← not fieldMask$
14. $discriminant ← ins.opcode$
15. Store the discriminant and mask
16. $insBuffer.push([ins, discriminant, opcodeMask])$
17. end for

**Step 2:** *Use the computed masks to search for ambiguities*

18. $ambiguous ← []$
19. Perform a cross-validation of Instruction values
20. for all $[computed1, index] ∈ insBuffer$ do
21.   for all $computed2 ∈ insBuffer[index + 1 :]$ do
22.     Unpack the computed values
23.     $[ins1, discriminant1, opcodeMask1] ← computed1$
24.     $[ins2, discriminant2, opcodeMask2] ← computed2$
25.     Compute a combined opcode mask
26.     $cMask ← opcodeMask1 & opcodeMask2$
27.     if $(discriminant1 & cMask) = (discriminant2 & cMask)$ then
28.       The two opcodes are ambiguous if equal
29.       $ambiguous.push([ins1, ins2, cMask])$
30.     end if
31.   end for
32. end for
33. return $ambiguous$
behaved through message passing; the :exec() method describes a sequence of events that will register within an event queue when the Instruction is called. When defining the syntax, we attempted to simplify the logic behind defining Instructions by converting normal Lua-operations (e.g. arithmetic) into sequences of events instead. [Listing 5.13] defines the execution of the very simple SUBLEQ instruction used in the URISC architecture. Executing the :exec() function performs a process akin to compilation using the state changes defined by the function itself and utilizing side-effect operations to “collect” the sequence of state changes within the function definition. This process allows the engine to compile the execution events through a sequential mechanism and create a state machine that behaves as a student describes it. This does not provide a fully-asynchronous behaviour like VHDL’s connection syntax, but does allow an execution engine to decide how state changes are scheduled. The following sub-sections describe the Lua statements found within [Listing 5.13] and how they compile to scheduled events within procsim.

```lua
exec = function (proc, operand)
    local B = operand:B -- alias

    -- Store A into "rhs" of operation
    proc:R = operand:A
    proc:memory[B] = B - proc:R -- *B <- B - A
    if (proc:N) { -- Negative val from ALU
        proc:PC = operand:P -- Jump as result was < 0
    } else {
        proc:PC++ -- Next ins
    }
end
```

Listing 5.13: Execution definition for the SUBLEQ Instruction (cut from [Listing 5.6]).
Assignment

An assignment statement, e.g. proc.R = operand.A is scheduled using Algorithm 2. The right-hand side arguments is known as the rval which is treated as an rvalue in C++ terminology [149]. The left-hand size argument is the value assigned to and is known as lval mimicking lvalue in C++ [149]. When working with assignments, this event-based architecture allows a simulation engine to reorder assignments depending on how the execution will progress. This flexibility allows for implementations of optimization patterns at a future time.

Algorithm 2 Assignment statement compilation

1. if rval is an operand then
2.  Read the bit pattern from the decoded operands
3.  bus ← operand
4. else
5.  Value is a module within the machine
6.  bus ← rval
7. end if
8. lval ← bus

Lua’s local variables are handled utilizing the result of any operations or assignments as a “BusValue” representing the state of the Bus in use rather than traditional integer values.

For example, after an assignment, the result of an operation is a “BusValue” that states what is currently on the data bus if and only if the assignment is not a direct module lookup. For example, consider the following local assignments:

Implicitly created “BusValue” semantics allow for code reuse within implementation definitions while remaining behaviourally accurate. local defined variables provide a simple way
CHAPTER 5. LUA-BASED CONFIGURATION-DRIVEN PROCESSOR SIMULATION

Listing 5.14: Local variable definitions within an Instruction:exec() method.

for students to write “high-level” code to simplify their implementations.

Memory Access

Memory accesses, proc.memory[B], read a memory location or write to it depending on whether the access is an lvalue or rvalue. In Listing 5.13 on line 65, the memory access is used as an lvalue, and we utilize Algorithm 3 to schedule the events.

Alternately, Algorithm 4 showcases the events triggered on writing a value to a location in memory such as proc.memory[0x04] = proc.B.

Once the memory access completes, the bus mechanics reuse the semantics outlined in the previous Assignment Section 5.3.2.

Arithmetic Operations

Arithmetic operations, e.g. operand.B - proc.R, happen through a single ALU within procsim’s model. In the context of Listing 5.13 line 65, when performing the subtraction ALU conditional bit flags based on the Freescale M68HC12 condition code flags are assigned by de-
5.3. **Instruction definition**

**Algorithm 3** Memory read compilation

**Input:** memory = Memory unit to read from or write to  
**Input:** address = address within memory  
**Input:** abus = Bus for address path  
**Input:** dbus = Bus for data path  
**Side-effect:** abus contains address  
**Side-effect:** dbus contains the value from memory[address]

Write the address to abus

1: \( abus \leftarrow address \)

   Set memory.readWrite to READ

2: \( \text{memory.readWrite} \leftarrow \text{READ} \)

3: \( \text{Memory buffer register gets the value at memory[address] on memory clock pulse} \)

4: \( \text{memory.buffer} \leftarrow \text{memory[address]} \)

5: \( \text{Write result of read to dbus on dbus clock pulse} \)

6: \( \text{dbus} \leftarrow \text{memory.buffer} \)

---

**Algorithm 4** Memory write compilation

**Input:** memory = Memory unit to read from or write to  
**Input:** address = address within memory  
**Input:** value = Value to write to memory  
**Input:** abus = Bus for address path  
**Input:** dbus = Bus for data path  
**Side-effect:** abus contains address  
**Side-effect:** dbus contains value  
**Side-effect:** memory[address] contains value

Write the address to abus

1: \( abus \leftarrow address \)

2: \( \text{Write value to dbus} \)

2: \( dbus \leftarrow value \)

   Set memory.readWrite to WRITE

3: \( \text{memory.readWrite} \leftarrow \text{WRITE} \)

4: \( \text{memory.buffer register gets the value from dbus on the dbus clock pulse} \)

5: \( \text{memory.buffer} \leftarrow \text{dbus} \)

6: \( \text{memory writes value to address on memory clock pulse} \)

7: \( \text{memory[address]} \leftarrow \text{memory.buffer} \)
fault to the proc:C, proc:N, and proc:Z. For multiplication or division operations, proc:O is also set. In the case of URISC, a manual “buffer” register of proc:R is utilized as the “right-hand side” of it’s adder. In the ALU for procsim, both left and right side of a binary operation have “buffer” inputs that can be assigned. Algorithm 5 shows the compilation steps for any ALU-based operation. The arithmetic application reuses the previously defined assignment semantics and memory access where required.

Algorithm 5 Arithmetic operation compilation

**Input:**
- `lhs` = Left-hand side argument
- `rhs` = Right-hand side argument
- `operation` = Arithmetic operation, e.g. `SUBTRACT`
- `result` = Result location
- `dbus` = Bus for data path
- `alu` = ALU module with condition bit flags

**Side-effect:**
- `dbus` and `result` contain result of arithmetic operation
- `alu` condition flags are set as appropriate

Using assignment semantics:

1: `alu.lhs ← lhs`
2: `alu.rhs ← lhs`
3: `alu.operation ← operation`
4: Start operation
5: `alu.start ← START`
6: Spin until operation completes, signalled via `alu.done`
7: `alu` condition bits are set to appropriate values per the operation
8: `result ← alu.result`
5.4 Project design and technical flaws

5.4.1 Compiler strain

While utilizing Lua gave a massive boost in power, speed and functionality of runtime configuration over the previous hc12sim project and procsim.scala, it has large costs to developer productivity. With the amount of software bugs found in the sol library and time spent fixing them, it was a slow process to implement features within the library. Given that sol utilizes high-level variadic template functions and SFINAE compile-time calculations and method resolution techniques [138] to implement most of its features, sol causes most compilers to begin to exponentially increase the time taken to compile [150]–[152]. sol’s inherent compilation growth caused compilation times for procsim to grow from under 10 minute for a clean build to over 40 minutes as more Lua wrapped functionality was added. More over, the use of templated and SFINAE mechanics caused compiler errors that were near impossible to decipher due to the raw amount of template instantiations present making small errors take massive amounts of time to decipher often resulting in a “try it and see” approach to fixing compiler errors. The work completed in Section 4.4 was started with the goal to try and appease these compiler constraints as reductions in objects to compile would reduce the over-all compiling times – linking times were inconsequentially affected by sol’s combinatorial explosion. In addition to changing how the software was built, as little information was placed into headers as possible when declaring Lua wrappers. This approach allowed for hiding implementations and forcing the compilers to reuse template instantiations as best as possible. However, the times were still unable to lowered below approximately 20 minutes for a clean build. “ThePhD” has since created a feature known as simple_usertype<T> which provides a runtime-binding
mechanism that severely reduces code size and memory contention in most compilers \[153\]. At the time, the `simple_usertype<T>` had significant performance penalties at runtime that have since been removed. The performance penalties of `simple_usertype<T>` were severe enough to impede use within the procsim library.

### 5.4.2 Loss of pedagogical gains for high- versus low-level constructs

As the features were developed, it became obvious that the “high-level” software approach produced a significant hiding of underlying hardware design – an original concern this approach was meant to combat. In revisiting other tools such as CPU Sim \[39\], we realized that instead of “lifting the black box,” the Lua configurations only shaved several layers off leaving a largely opaque sheet across the model. The model of side-effect-based compilation is extremely powerful and useful as a productivity tool, but it does not benefit students as it hides a lot of wiring details from students. The side-effect based approach to state machine definition provided a higher level approach than projects like \[56\] but also hid too much of the implementation details from students reducing their hands-on design experience. This “high-level” synthesis of a control unit causes students to ignore design considerations surrounding control signals or control unit algorithms. This is particularly evidenced by comparing the control unit for the URISC ISA in Figure 5.2 to the instruction definition from Listing 5.7 coupled with the implicit fetch-decode cycle for execution. While the control unit displayed by Figure 5.2 is not inherently complicated, but it provides a significant design challenge for students to implement it themselves. The procsim configuration does not have any of these signals included and implicitly provides most of them. We believed this to be a direct counter to the pedagog-
5.4. Project design and technical flaws

Figure 5.2: Control unit for the 16-bit URISC architecture from [6, p. 331].

...ical Requirement 3. The ease of use and agility provided by procsim’s configurations is very powerful but costs students learning opportunities. We believed there to be a dissection point between the Lua synthesis of control unit logic and [56]’s approach using a state table. Students require the ability to get their “hands dirty” and unfortunately procsim’s configurations could not provide enough dirt [21], [31], [39], [56].
5.5 Analysis of Requirements

procsim provided a powerful simulation framework design and a lot of extremely strong ideas on how to develop a proper cross-platform, high-performance solution to education of embedded systems and computer architectures. procsim was proven to run on most major operating systems as required by [Requirement 1] but required the installation of a program putting it a step behind a purely web-based application. The configurations available were not so deep as to provide complete circuit-level simulation like ShelbySim, CPU Sim or Emuemaker86, but it intentionally left out mechanics to try and alleviate the overwhelming nature of a full-blown circuit simulation. Thus procsim met most of the configuration requirements for [Requirement 2]. In trading off configuration for pedagogy, procsim provided a solid base for improving students outcomes. But due to the level of information implicitly provided by the side-effect based execution, or outright hidden, it encroached and passed the requirement into becoming detrimental to pedagogy compared to competing projects (Requirement 3).

Simulations within procsim were fast and provided hooks for all information within components. It did not however provide support for reading signals within the system or investigation of microcode execution. It had support for event-based execution implying different granularities of debugging could be implemented on top of the simulation engine. These features provide most of the requirements for [Requirement 4] but procsim had no capabilities for peripheral support at the time. The event-based execution engine would have allowed for granular debugging. Lastly, the IDE provided by the original hc12sim project was more modern than most projects surveyed, but it did not have the features required to keep up with major IDEs. It was our intention to integrate procsim into Eclipse and utilize the Lua Development Toolkit
to provide the procsim simulator as an Eclipse-based IDE [154]. This would make procsim extremely modern as a fully-featured IDE thanks to Eclipse’s plugin architecture. Additionally, procsim utilized cutting edge technologies through Lua and C++11 / C++14 to produce powerful and easy to use software familiar to students. This modern tool set combined with a current IDE would create a modern feeling experience for students alleviating concerns of “dated” technologies for Requirement 5.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>procsim</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of requirement matching for procsim.
Chapter 6

Conclusion

This thesis examined the topic of education for Computer and Software Engineering students on the subject of embedded systems and Instruction Set Architectures. While ultimately unsuccessful in developing a working solution, we have provided a series of known working and invalid approaches to solving the problem outlined in Section 1.3. We proposed a framework of requirements for a solution to work in the modern laboratory and investigated two major solutions. The first implementation used Scala and the distributed framework Akka to develop a massively asynchronous solution to representing an ISA. In addition, this project investigated the use of a Scala program as a configuration definition. We then implemented a set of novel test and build infrastructure components to better build cross-platform C++ applications. Lastly, we used the knowledge gained from the first and second projects to iterate on an existing project and create a Lua-based configuration system for the desktop. The last chapter included a “compiler”-like implementation of state-machine representations of a microinstruction and showcased a configuration language capable of defining a full Turing-complete machine. In this chapter, we will summarize the main contributions from each chapter and use the lessons
learned to propose future goals for this research question.

In the following sections, we reiterate our contributions and provide recommendations for future work based on post-mortem analysis of our projects discuss in previous chapters.

6.1 Contributions

Survey of other and previously completed works  We provided a quantitative application of the requirements outlined in the Problem Statement within Section 1.3 with previous works in use today. From these existing technologies, we also extracted key features that should be built into new solutions going forward. Many of the existing simulations technologies surveyed use dated software tools to build them and do not provide ample interfaces focused on human computer interaction concepts from the student’s perspective.

procsim.scala: Scala-based event-driven processor simulation for the web  Chapter 3 proposed a web application known as procsim.scala built on the Scala.js platform using Akka and Akka.js to simulate a custom, massively parallel processor in a synchronous JavaScript environment. We outlined a DSL language for specifying a microinstruction’s execution within a Scala script. We also pushed the envelope for state-of-the-art utilizing emerging tools in a novel environment. Regrettably, these emerging tools could not maintain a high performing application at the time.

Developing cross-platform C++ applications  We discussed an anecdotal account of refactoring the hc12sim project into a modern C++ environment. Through the recount, we proposed infrastructure improvements to test large native applications within multiple platforms while
We evaluated the use of four scripting languages within a C++ application as an embedded language runtime. Each of these languages were evaluated and Lua was selected as the optimal choice due to proved use in high-performance applications, ease of embedding and data-description capabilities. We implemented a processor description within Lua that allowed for fully Lua-compliant scripts to describe the modules of a simple architecture. Within this definition, we provided a design for instruction state change microcoding through unintrusive side-effect-based execution which was compiled to events within a simulation engine. This representation hid a lot of implementation details hoping to improve student outcomes, however we realized that through utilizing such high-level constructs to teach low-level hardware design we lost a lot of necessary context.

6.2 Recommendations

While our work in this thesis did not produce a fully working prototype, we developed many contributions towards future work in developing teaching simulation tools for embedded sys-
tems and hardware-software co-design. Through our work, we hope others can improve their tooling to close the open problem of improving teaching of a difficult subject for many students. We recommend the following improvements to our models and existing software.

**Remove the requirement for real-time simulations.** Naively, we intended to implement simulations at real-time speeds of older architectures. When implementing the hc12sim project in Section 2.1, we produced a real-time simulation of the Freescale M68HC12. This was an admirable goal, but the cost of this meant utilizing native tools and imposing design requirements that proved undesirable to the pedagogical outcome of the simulations. Maintaining a behaviourally accurate simulation at the cost of speed of execution is a valid if not beneficial trade-off. When implementing Akka.js, [75] showcased that asynchronous software does not necessitate parallelism. By removing parallelism, it opens technology choices we could not previously utilize and allows for a more simplistic simulation engine implementation.

**Investigate functional style component development through reactive frameworks.** Akka’s model for distributed systems is necessary when dealing with bidirectional data flow within an application. We have realized that for circuits and hardware simulation, data flow is generally unidirectional in that signals are passed onto a connection. This is evidenced by how HDL languages like VHDL often treat hardware specifications as input-output connected black boxes. Since developing the procsim.scala project, there has been a movement within the reactive community to produce a series of libraries known as ReactiveX that utilize unidirectional program flow to reduce software complexity within asynchronous applications [155]. With processor simulation being an asynchronous system, these libraries could simplify applica-
tion development significantly. Further, the ReactiveX community provides implementations for most major programming environments including Java [156], Scala [157] and JavaScript [158].

**Use a bisecting design between Lua-based control units and Enumaker86’s ([56]) state-table.** The process of compiling the Lua specifications of instruction microcode, we accidentally hid a lot of information to students. We allowed them to quickly define designs, but as others found [21], [28]–[30], [56] students benefit from getting into the deeper design of a hardware architecture. We believe the work completed by [56] to provide a distinction between hardware data paths and the control unit provides a “deep enough” philosophy for students to learn how these concepts work without holding them back. We suggest improvements on their model by utilizing state charts over state tables and developing mechanisms to better organize the information into “modules” of state for repeatability – something our Lua configurations allowed through variable declarations.

**Utilize modern web technologies over desktop applications.** While a large component of our work completed in Chapters 4 and 5 involved native applications, we believe the best environment for students in a modern classroom is through web applications. With the performance of modern JavaScript in the browser, we believe students are more likely to enjoy computer architecture discussions if the tools they utilize are within comfortable environments. Additionally, we found much higher productivity when working with Scala.js than with our C++ applications. While improving the developer work flow for C++, we continually fought with multiple platforms and massive tool-chain differentiation. We believe JavaScript and modern
web technologies give the best of developer productivity and modern applications to reduce the
time taken to develop and iterate on these tools.

6.3 Future Work

For any work going forwards, we believe the lessons learned in this thesis provide a through
starting point for a project. We expect future work would include a web-based solution that
takes advantage of reactive components to provide an accurate and easy to develop and iterate
solution. By utilizing these modern tools, paradigms, and testing knowledge we firmly believe
a solution could be created that caters to all of our requirements outlined in Section 1.3.
Bibliography


Appendix A

Procsim headers

The following listings contain headers describing the core model that is assembled from a Lua configuration such as Listings 5.6 and 5.7. These headers are supplied to showcase the state produced from Lua configurations. Implementation files are not provided. We have only included the simulation entities for brevity as outlined in Section 5.1.3.

A.1 Memory Components

Listing A.1: Source for procsim/memory/MemoryUnit.hpp.

```cpp
#pragma once
#ifndef PROCSIM_MEMORY_MEMORYUNIT_HPP_
#define PROCSIM_MEMORY_MEMORYUNIT_HPP_

/*!
 * The following is part of the Procsim Library written and developed
 * by Kevin Brightwell and Ramesh Raj under the supervision of
 * Dr. Kenneth McIssac at Western University, Canada.
 *
 * \author Kevin Brightwell
 *
 * This software is provided as is and all rights are maintained by
 * Kevin Brightwell
```
A.1. Memory Components

#include "procsim/Types.hpp"
#include "procsim/util/StringIO.hpp"
#include <memory>
#include <set>
#include <string>
#include <boost/variant.hpp>

PROCSIM_NAMESPACE_START

namespace time
{
    class Clock;
}

namespace memory
{
    /*!
     * \brief Simple marker class for how to access memory
     * Memory and register access is completed by using an `unordered_set`
     * of `Access::Value`'s.
     *!
    struct Access
    {
        public:
            /*!
             * Marker for Access Types
             *!
            enum Value
            {
                _MIN = 0,
                READ = 1, // Readable
                WRITE = 2, // Writable
                _MAX = 3
            };

            // Shortcut typedef for a set of `Access::Value`'s.
```cpp
typedef std::set<Value> Set;

// signifies Read ability
static const Access::Set Read;

// Write ability
static const Access::Set Write;

// Shortcut member for Read+Write
static const Access::Set ReadWrite;

private:
    Access() = delete;
    ~Access() = delete;
};  // end struct Access

/// Stream the Access::Value enumeration
inline std::ostream& operator<<(std::ostream& os, const Access::Value& v)
{
    switch (v)
    {
        case Access::READ:
            os << "Access::Read";
            break;
        case Access::WRITE:
            os << "Access::Write";
            break;
        default:
            os << "Access::UNKNOWN{" << static_cast<int32_t>(v) << "}";
            break;
    }

    return os;
}

/// Stream the Access::Value enumeration
inline std::ostream& operator<<(std::ostream& os, const Access::Set& v)
{
    if (v == Access::ReadWrite)
    {
        os << "Access::ReadWrite";
    }
```
else if (v.size() == 1)
{
    os << *v.begin();
}
else
{
    // this doesn't make much sense
    stringio::join(os, v, ", ", "{', '");
}

return os;

/*! * rief Describes a unit of memory, for example a ROM or RAM */
class PROCSIM_EXPORT MemoryUnit
{

public:

    /// Simple struct for memory parameters, there's too many for
    /// function calls.
    struct PROCSIM_EXPORT Parameters
    {
        /// Name for the parameters
        optional<std::string> name;

        /// A name of a \c conf::Clock that will be found outside or a
        /// definition
        boost::variant<std::shared_ptr<time::Clock>, std::string> clock;

        /// Cycle count before returning a read value
        size_t readCount;

        /// Cycle count before writing a value
        size_t writeCount;

        /// Offset into processor memory
        optional<size_t> offset;

        /// Size of this memory unit in bytes
        optional<size_t> size;

        /// Access pattern for this memory
        Access::Set access;
};

/// Name of `this` type, all subclasses of `ConfObj` must have this.
static PROCSIM_CONSTEXPR const char TYPE_NAME[] = "MemoryUnit";

/// Base constructor
MemoryUnit(const Parameters&& params);

MemoryUnit(MemoryUnit&&) = default;
MemoryUnit(const MemoryUnit&) = default;
MemoryUnit& operator=(const MemoryUnit&) = default;
virtual ~MemoryUnit() = default;

/// Check for equality
bool operator==(const MemoryUnit& other) const;

inline const std::string name() const { return *_name; }

inline const optional<std::string> name_opt() const
  PROCSIM_NOEXCEPT { return _name; }
inline void set_name(const std::string& name) { _name = name; }

/// Get the clock value this will crash if the clock value has not
/// been resolved
inline const std::shared_ptr<time::Clock> clock() const
  PROCSIM_NOEXCEPT { return boost::get<std::shared_ptr<time::Clock>>(_clock); }

/// Get the clock value, will either be a \c conf::Clock or a
/// path.
inline const boost::variant<std::shared_ptr<time::Clock>,
  std::string> clock_var() const PROCSIM_NOEXCEPT
{
  return _clock;
}

/// Get the read count for this instance.
inline const size_t readCount() const PROCSIM_NOEXCEPT { return _readCount; }

/// How many cycles to write a value
inline const size_t writeCount() const PROCSIM_NOEXCEPT { return _writeCount; }

/// The offset this memory lives at in processor memory (in bytes)
inline const optional<size_t> offset() const PROCSIM_NOEXCEPT {
    return _offset;
}

/// The size of this memory in bytes
inline const optional<size_t> size() const PROCSIM_NOEXCEPT {
    return _size;
}

/// The Access pattern for this memory
inline const Access::Set& access() const PROCSIM_NOEXCEPT {
    return _access;
}

/*! Set the clock instance.
 * \param pclock \c conf::Clock instance to set
 */
void set_clock(const std::shared_ptr<time::Clock>& pclock);

protected:
    /// Name of the memory unit
    optional<std::string> _name;

    /// How many cycles to read a value
    size_t _readCount;

    /// How many cycles to write a value
    size_t _writeCount;

    /// The offset this memory lives at in processor memory (in bytes)
    const optional<size_t> _offset;

    /// The size of this memory in bytes
    optional<size_t> _size;

    /// The Access pattern for this memory
    Access::Set _access;

private:
    /// Clock configuration
    boost::variant<std::shared_ptr<time::Clock>, std::string> _clock;

    friend std::ostream& operator<<(std::ostream& os, const
        MemoryUnit& mu);
};

export_smart_ptrs(MemoryUnit)
class PROCSIM_EXPORT RAM PROCSIM_FINAL : public MemoryUnit
{

public:
    /// Name of `this` type, all subclasses of `ConfObj` must have this.
    static PROCSIM_CONSTEXPR const char TYPE_NAME[] = "RAM";

    /// Base constructor
    RAM(const MemoryUnit::Parameters& params);

    RAM(RAM&&) = default;
    RAM(const RAM&) = default;
    RAM& operator=(const RAM&) = default;
    virtual ~RAM() = default;

    /// Check for equality
    inline bool operator==(const RAM& other) const
    {
        return _access == other.access() &&
               MemoryUnit::operator==(other);
    }

    inline bool check_eq(const MemoryUnit& other) const
    {
        return _access == other.access() &&
               MemoryUnit::operator==(other);
    }
};

export_smart_ptrs(RAM)

class PROCSIM_EXPORT ROM PROCSIM_FINAL : public MemoryUnit
{

public:
    /// Name of `this` type, all subclasses of `ConfObj` must have this.
    static PROCSIM_CONSTEXPR const char TYPE_NAME[] = "ROM";

    /// Base constructor
ROM(const MemoryUnit::Parameters& params);

ROM(ROM&&) = default;
ROM(const ROM&) = default;
ROM& operator=(const ROM&) = default;
virtual ~ROM() = default;

/// Check for equality
inline bool operator==(const ROM& other) const
{
  return _access == other.access() &&
  ~MemoryUnit::operator==(other);
}
inline bool check_eq(MemoryUnit& other) const
{
  return _access == other.access() &&
  ~MemoryUnit::operator==(other);
}

export_smart_ptrs(ROM)

// Out of line operators
bool operator==(const RAM& lhs, const MemoryUnit& rhs);
bool operator==(const MemoryUnit& lhs, const RAM& rhs);

bool operator==(const ROM& lhs, const MemoryUnit& rhs);
bool operator==(const MemoryUnit& lhs, const ROM& rhs);

} // end namespace memory

PROCSIM_NAMESPACE_END

#endif // PROCSIM_MEMORY_MEMORYUNIT_HPP_
Listing A.2: Source for procsim/memory/Register.hpp.

```cpp
#ifndef CONF_REGISTER_HPP_
#define CONF_REGISTER_HPP_

/*! 
 * The following is part of the Procsim Library written and developed 
 * by Kevin Brightwell and Ramesh Raj under the supervision of 
 * Dr. Kenneth McIssac at Western University, Canada. 
 * 
 * \author Kevin Brightwell 
 * 
 * This software is provided as is and all rights are maintained by 
 * Kevin Brightwell 
 */

#include "MemoryUnit.hpp"
#include "procsim/Types.hpp"
#include <map>
#include <set>
#include <string>
#include <vector>
#include <boost/variant.hpp>

PROCSIM_NAMESPACE_START

namespace memory
{

class Register : public MemoryUnit
{

public:
    // FIXME Make this actually work 
    // Location map for joined registers 
    typedef std::map<size_t, boost::variant<std::shared_ptr<Register>,
                                                  std::string>> joined_map_t;

    // Name of the class "Register"
    static const std::string TYPE_NAME;

PROCSIM_NAMESPACE_END

} // namespace memory

// End of memory/Register.hpp
```

/!
* \brief Base ctor for memory units, this ctor is a convenience function
* \param name Name of the Register
* \param params Basic \c MemoryUnit::Parameters for a MemoryUnit
* \param joined Optional joined parameter
*/
Register(const MemoryUnit::Parameters&& params, const
optional<joined_map_t>& joined = nullopt);

/*!*
* \brief Creates a mapped register based on offsets and underlying registers.
* Creates a Register configured to be overlapping or a combination. For example, in the PROCSIM,
* A + B = D where the map would be `{ 0 = A, 8 = B }`.
* The `readCount` and `writeCount` are calculated from the underlying `Register`s as the `max()`
* of them.
* If no `access` is set, the default setting is to calculate the gcd of the access patterns of the underlying registers. If the underlying set is empty, this is an `Error` as is an illegal specification.
* The `size` of the register is calculated by adding the first bit index + last index + last
* `Register`s `size` to get the minimum size required which is then mapped to a valid `uint`
* type.
* There is no way to memory map a joined register, if the underlying components are valid, it
* will "just work".
* \param joined Mapping of index to `conf::Register` where the underlying Registers are verified to be in valid positions. They may overlap or be separated.
* \param access The access pattern for this shared register. `Access::Write` must be
transitive, that is, you can not have underlying read-only registers and have write access. This applies for `Access::Read` but conversely.

```
static Register make_joined(const std::string& name, const joined_map_t& joined,  
const Access::Set& access = Access::Set());
```

Register(Register&&) = default;
Register(Register&) = default;

```
virtual ~Register() = default;
```

```
inline bool operator==(const Register& other) const 
{
    return (this == &other || (MemoryUnit::operator==(other) &&  
        _joined == other._joined));
}
```

```
/// Get the width of the \c Register in bits.
inline const size_t bitWidth() const { return (*_size) * 8; }
```

```
/// Get the joined mapping if it exists
const optional<joined_map_t>& joined() const { return _joined; }
```

```
protected:
    void joinedRegister(const std::shared_ptr<Register> pregister);
```

```
private:
    optional<joined_map_t> _joined;

    friend std::ostream& operator<<(std::ostream&, const Register&);
};

export_smart_ptrs(Register)

} // end namespace memory

PROCSIM_NAMESPACE_END

#endif // CONF_REGISTER_HPP_
A.2 Time

Listing A.3: Source for procsim/time/Clock.hpp.

```cpp
#pragma once
#ifndef PROCSIM_TIME_CLOCK_HPP_
define PROCSIM_TIME_CLOCK_HPP_

/*!
 * The following is part of the Procsim Library written and developed
 * by Kevin Brightwell and Ramesh Raj under the supervision of
 * Dr. Kenneth McIssac at Western University, Canada.
 *
 * \author Kevin Brightwell
 *
 * This software is provided as is and all rights are maintained by
 * Kevin Brightwell
 */

#include "procsim/Macros.hpp"
#include "procsim/Types.hpp"

#include <chrono>
#include <iostream>
#include <memory>
#include <string>
#include <tuple>

PROCSIM_NAMESPACE_START

namespace time {

class Clock PROCSIM_FINAL {

public:

    enum class PulseType {
        SQUARE  = 0,
        PULSE   = 1,
        INV_PULSE = 2
    };

    //!< Name of `this` type
```
```cpp
static PROCSIM_CONSTEXPR const char TYPE_NAME[] = "Clock";

//!< Minimum allowable clock speed
static const std::chrono::nanoseconds MINIMUM_CLOCK_TIME;

//!
/*! Constructs a new instance <i>outside</i> of Lua
/*! \param name Name of this instance (used with path)
/*! \param parent A pointer to the parent */
Clock(const optional<std::string>& name, const
  std::chrono::nanoseconds& period, const PulseType pulseType);

Clock(Clock&&) = default;
Clock(const Clock&) = default;

virtual ~Clock() = default;

inline bool operator==(const Clock& other) const
{
    return (this == &other || (_name == other._name && _pulseType
   == other._pulseType && _period == other._period));
}

inline const std::string& name() const { return *_name; }

inline const optional<std::string>& name_opt() const
PROCSIM_NOEXCEPT { return _name; }

inline void set_name(const std::string& name) { _name = name; }

//!< Name of the current node
template <class TimeUnit = std::chrono::nanoseconds>
inline const TimeUnit period() const PROCSIM_NOEXCEPT
{
    return _period;
}

inline const PulseType pulseType() const PROCSIM_NOEXCEPT { return
   _pulseType; }

private:
    optional<std::string> _name;

//!< Name of the current node
const std::chrono::nanoseconds _period;
```
const PulseType _pulseType;

friend std::ostream& operator<<(std::ostream&, const Clock&);
};

const std::shared_ptr<Clock>& inherit_clock();

template <>
inline const uint64_t Clock::period<uint64_t>() const PROCSIM_NOEXCEPT
{
    return _period.count();
}

inline std::ostream& operator<<(std::ostream& os, const Clock::PulseType pt)
{
    os << "PulseType::";
    switch (pt)
    {
    case Clock::PulseType::PULSE:
        os << "PULSE";
        break;

    case Clock::PulseType::INV_PULSE:
        os << "INV_PULSE";
        break;

    case Clock::PulseType::SQUARE:
        os << "SQUARE";
        break;

    default:
        break;
    }

    return os;
}

} // end namespace time

PROCSIM_NAMESPACE_END

#endif // PROCSIM_TIME_CLOCK_HPP_
Listing A.4: Source for procsim/time/AsyncTimerReceiver.hpp.

```cpp
#pragma once
#ifndef PROCSIM_TIME_ASYNCTIMERRECEIVER_HPP_
define PROCSIM_TIME_ASYNCTIMERRECEIVER_HPP_

#include "procsim/time/Timer.hpp"
#include "procsim/Types.hpp"
#include <memory>
#include <thread>
#include <boost/signals2.hpp>

PROCSIM_NAMESPACE_START

/**
 * An object which relies on an external timer to execute. This is done to
 * better simulate and possibly underclock some operations within the simulator. Most objects will be bound to the main CPU, but may have other clocks associated.
 */

class PROCSIM_EXPORT AsyncTimerReceiver
{
public:

    /**
     * \brief Creates a new AsyncTimerReceiver object.
     * \warning The Timer object must be bound after constructing. Use `get_ptr()`.
     */
    AsyncTimerReceiver(const std::shared_ptr<PROCSIM_N::Timer> timer, const Timer::OnSignalChange& tickFunc);

    virtual ~AsyncTimerReceiver();

    void start();

    void stop();

    inline const bool running() const { return _active; }

PROCSIM_NAMESPACE_END
```

private:
/** \brief Executes a smart waiting loop while waiting for the
   next `tick` to arrive to execute the next action.
*/
void _thread_run();

std::mutex mtx_Tick;//!< Mutex for locking and unlocking a timer, locked while waiting for a tick

std::unique_ptr<std::thread> _thread;

const std::shared_ptr<Timer> _timer;  //!< Reference to the Timer

const Timer::OnSignalChange _onSignal;

std::atomic_bool _active;
};

PROCSIM_NAMESPACE_END

#endif // PROCSIM_TIMEASYNCTIMERRECEIVER_HPP_
#pragma once
#ifndef PROCSIM_TIMER_HPP_
#define PROCSIM_TIMER_HPP_

#include "procsim/Macros.hpp"
#include "procsim/Types.hpp"

#include <atomic>
#include <chrono>
#include <condition_variable>
#include <functional>
#include <mutex>
#include <set>
#include <shared_mutex>
#include <thread>
#include <vector>

#include <boost/signals2.hpp>

PROCSIM_NAMESPACE_START

// Forward dependancies

class AsyncTimerReceiver;

/**
 * A top level class which allows an object to maintain the time cycle of other
 * AsyncTimerReceiver objects. The Timer will asynchronously update the
 * AsyncTimerReceiver objects it has bound. These objects will then work in their
 * own thread space to execute then wait for the next "tick."
 */
class PROCSIM_EXPORT Timer : public std::enable_shared_from_this<Timer>
{

public:
    typedef std::chrono::nanoseconds res_type;
    typedef uint64_t tick_count_t;
enum class SignalState : int16_t
{
    X  = std::numeric_limits<int16_t>::min(), // lowest neg
    Z  = std::numeric_limits<int16_t>::max(), // highest pos
    LOW = 0,
    HIGH = 5
};

typedef std::function<void(const SignalState, const tick_count_t)> OnSignalChange;

/**
 * \brief Creates a Timer object with the specified resolution.
 * \param resolution The period of the `Timer`
 */
Timer(const res_type& resolution);

/** \brief Deallocates resources, forcibly stops all related threads if necessary
 */
virtual ~Timer();

void start();

/**!
 * \brief Pauses the timer after the current `tick` completes.
 */
void pause();

/**!
 * \brief Returns true if paused.
 */
bool isPaused() const;

void setStepFor(const int32_t steps = 1);

int32_t stepFor() const;

/**!
 * \brief Attempts to elegantly stop all attached objects by setting them

* inactive followed by std::thread::join. It then joins its
  inact ing
* thread.
*/

void stop();

// tick
void tick();

/**
 * \brief Returns the period of the `Timer` object.
 * \return The period of the `Timer`.
 */

res_type resolution() const;

/**!
 * \brief Get the number of ticks that have occurred.
 * \returns Count of ticks, total running time is `(tickCount() * 
  resolution())`
 */

const tick_count_t tickCount() const;

/**!
 * \brief Gets the current running time, accurate with pausing.
 * \returns `std::chrono::duration` based on the templated
 * parameter.
 * */

template <class DurType>
DurType runningTime() const
{
  // FIXME This only lets you pause once...
  if (!active)
  {
    auto now = std::chrono::system_clock::now();
    DurType dur;
    if (_paused)
    {
      dur = std::chrono::duration_cast<DurType>(_tpPauseMod
                                               - _tpRunning);
    }
    else
    {
      dur = std::chrono::duration_cast<DurType>(now
                                               - _tpRunning);
    }
}
return dur;
}
else {
    return DurType(0);
}
}

template <class TD>
std::shared_ptr<TD> makeDependant(const Timer::OnSignalChange&& tickFunc)
{
    auto ptr = std::make_shared<TD>(shared_from_this(), tickFunc);
    bindDep(ptr);
    return ptr;
}

template <class TD, class... U>
std::shared_ptr<TD> makeDependant(const Timer::OnSignalChange&& tickFunc, U&&... u)
{
    auto ptr = std::make_shared<TD>(shared_from_this(), tickFunc,
                                      std::forward<U>(u)...);
    bindDep(ptr);
    return ptr;
}

/*! rief Get the signal state of the timer. 
   
   \return Current signal state 
   */
const SignalState signalState() const;

/*! rief Causes the currently executing thread to wait for the next notification from the `std::condition_variable` stored within the Timer.
   
   \param lock The lock to wait on
   \param LockType The class of the lock, not necessary to specify
```cpp
// Get the current amount of ticks
uint64_t tc = tickCount();

while (_active && tc == tickCount())
{
    // it should *never* take longer than `resolution`
    _cvTick.wait_for(lock, resolution());
}

return std::make_tuple(signalState(), tickCount());
```

--

```cpp
/*!
 * \brief Get the number of dependants currently bound to this Timer.
 * \return Count of dependants.
 */
inline const size_t dependantCount()
{
    std::lock_guard<std::recursive_mutex> lock(mtx_depObjs);

    cleanDeps();
    return _depObjs.size();
}
```

--

```cpp
private:
/*!
 * \brief Binds a AsyncTimerReceiver object to this Timer. It will be called
 * to run when necessary.
 * \param dep AsyncTimerReceiver object to run on Tick
 */
inline void bindDep(const
    std::weak_ptr<PROCSIM_N::AsyncTimerReceiver> dep)
{
    std::lock_guard<std::recursive_mutex> lock(mtx_depObjs);

    if (!dep.expired())
    {
        // Don't worry about multiple inserts, it's a set
```
_depObjs.insert(dep);
}

cleanDeps();
}

/*! 
 * \brief Removes a AsyncTimerReceiver object from the timer. This stops its execution until re-added. Additionally, any weak_ptrs that are expired are removed.
 */
inline void removeDep(const std::weak_ptr<PROCSIM_N::AsyncTimerReceiver> dep)
{
    std::lock_guard<std::recursive_mutex> lock(mtx_depObjs);
    _depObjs.erase(dep);
    cleanDeps();
}

/*! 
 * \brief Remove any "dead" dependant weak_ptrs.
 */
inline void cleanDeps()
{
    std::lock_guard<std::recursive_mutex> lock(mtx_depObjs);
    for (auto it = _depObjs.begin(); it != _depObjs.end();)
    {
        if (!it->lock())
        {
            it = _depObjs.erase(it);
        }
        else
        {
            ++it;
        }
    }
    std::atomic_bool _active; //! If set to false, the thread will stop at its ideal point
std::unique_ptr<std::thread> _tickThread; //"Main" thread executing to "tick" on its objects

res_type _resolution; //"Time between consecutive ticks"

// TIME RUNNING INFORMATION
std::chrono::system_clock::time_point _tpRunning; //"Used for accurate time running metric"
std::chrono::system_clock::time_point _tpPauseMod; //"Used to compute accurate pausing"

// mutable keyword used because there is no reason to not be able to make waitForTick const.
mutable std::condition_variable _cvTick; // no need to have lock because it's already synchronized

std::atomic<tick_count_t> _tickCount; // The number of ticks since starting

mutable std::shared_mutex mtx_signalState;
SignalState _signalState;

mutable std::recursive_mutex mtx_depObjs; //"Used to lock the `depObjs` vector"
std::set<std::weak_ptr<AsyncTimerReceiver>, std::owner_less<std::weak_ptr<AsyncTimerReceiver>>> _depObjs; //"The timer dependant objects"

mutable std::mutex mtx_Step;
int32_t _step;

std::condition_variable _cvPause;
std::mutex mtx_Pause, mtx_PauseWait; //"Needed for condition_variable"
bool _paused; //"If true, the timer will wait"
### Encoding

Listing A.6: Source for procsim/encoding/Algorithm.hpp.

```cpp
#pragma once
#ifndef PROCSIM_ENCODING_ALGORITHM_HPP_
define PROCSIM_ENCODING_ALGORITHM_HPP_

/*!
* The following is part of the Procsim Library written and developed
* by Kevin Brightwell and Ramesh Raj under the supervision of
* Dr. Kenneth McIssac at Western University, Canada.
* 
* \author Kevin Brightwell
* 
* This software is provided as is and all rights are maintained by
* Kevin Brightwell
*/

#include "Primitives.hpp"
#include "procsim/Macros.hpp"
#include "procsim/Types.hpp"
#include "procsim/util/Math.hpp"

#include <cstdint>
#include <memory>
#include <unordered_map>

PROCSIM_NAMESPACE_START

namespace encoding
{

/// Definition of how to encode variables
typedef std::unordered_map<std::string, TypedIndexedNumber> Definition;

/// Extracted variables from an encoding
typedef std::unordered_map<std::string, TypedValue> Extraction;

class PROCSIM_EXPORT Algorithm
{

public:
```
```cpp
inline Algorithm(const Definition& definition) :
    _encodeDefinition(new Definition(definition)) {}

inline Algorithm() : _encodeDefinition(nullptr) {}

inline Algorithm(const Algorithm&) = default;

virtual ~Algorithm() {
    if (_encodeDefinition)
    {
        delete _encodeDefinition;
    }
}

inline bool operator==(const Algorithm& other) const
    PROCSIM_NOEXCEPT
{
    if (this == &other)
    {
        return true;
    }

    // Check that the pointers are valid or invalid
    return ptr_eq(this->_encodeDefinition, other._encodeDefinition);
}

/// Calculate the minimum width required to represent all of the
/// encoded values. This
/// calculation finds the minimum number of bytes
inline virtual const bytes minRep() const
{
    if (!this->_encodeDefinition)
    {
        return bytes{ 0 };  
    }

    // We use | because it is faster than |=
    uint64_t v = 0;
    for (const auto& p : *this)
    {
        v |= p.second.mask;
    }

    return math:min_rep_width<bytes>(v);
}
Compute a mask for this encoding that hits all of the encoded positions.

```cpp
inline virtual const uint64_t mask() const { return math::build_mask<uint64_t>(minRep()); }
```

Decodes a value into a map with all of the values by name with their types.

```cpp
virtual const Extraction decode(const uint64_t value) const;
```

Defines the encoded variables

```cpp
const Definition* const _encodeDefinition;
```

Commented block

```cpp
PROCSIM_NAMESPACE_END
```

```cpp
#endif // PROCSIM_ENCODING_ALGORITHM_HPP_
```
Listing A.7: Source for procsim/encoding/Code.hpp.

```cpp
#pragma once
#ifndef PROCSIM_ENCODING_CODE_HPP_
define PROCSIM_ENCODING_CODE_HPP_

/*! 
 * The following is part of the Procsim Library written and developed 
 * by Kevin Brightwell and Ramesh Raj under the supervision of 
 * Dr. Kenneth McIssac at Western University, Canada.
 */
#include "Algorithm.hpp"
#include "procsim/Macros.hpp"
#include "procsim/Types.hpp"
#include <cstdint>
#include <iostream>
#include <memory>
#include <unordered_map>
#include <unordered_set>

PROCSIM_NAMESPACE_START

namespace encoding
{

class Code;

namespace Code_details_
{
    // The following classes allow Encoding::Code to be a pimpl 
    // implementation, making the interface faster and 
    // much more clean

    class PROCSIM_EXPORT CodeImpl : public Algorithm
    {
```
public:
inline CodeImpl(const std::unordered_map<std::string, uint64_t>& values, const Definition& encodingDefinition) : Algorithm(encodingDefinition), _values(values) {
    _valueSet.reserve(_values.size());
    for (const auto& p : _values) {
        _valueSet.insert(p.second);
    }
}

virtual ~CodeImpl() = default;

inline virtual const bool operator==(const CodeImpl& other) const PROCSIM_NOEXCEPT {
    return (this == &other) || (Algorithm::operator==(other) && _values == other._values);
}

virtual const bool check(const uint64_t code) const PROCSIM_NOEXCEPT = 0;

inline const std::unordered_set<uint64_t> values() const {
    return _valueSet;
}

inline const std::unordered_map<std::string, uint64_t> pairs() const {
    return _values;
}

virtual std::ostream& stream(std::ostream& os) const = 0;

protected:
virtual const uint8_t tag() const = 0;

friend class Code;

private:
const std::unordered_map<std::string, uint64_t> _values;
std::unordered_set<uint64_t> _valueSet;
};
class PROCSIM_EXPORT Code PROCSIM_FINAL : public Algorithm
{

public:

static const std::string TYPE_NAME;

Code(const uint64_t constant);
Code(const std::unordered_map<std::string, uint64_t>& alternates,
const Definition& encodedVars = {});
Code(const uint64_t constant, const Definition& encodedVars);

Code(Code&&) = default;
Code(const Code&) = default;
Code& operator=(const Code&) = default;
virtual ~Code()
{
    if (_impl)
    {
        delete _impl;
    }
}

const bool operator==(const Code& other) const PROCSIM_NOEXCEPT;
virtual const bytes minRep() const override final;

/// Compute a mask for this encoding that hits all of the encoded positions.
virtual const uint64_t mask() const override final;

/// Decodes a value into a map with all of the values by name with their types
const Extraction decode(const uint64_t value) const override final;

const bool check(const uint64_t code) const PROCSIM_NOEXCEPT;
const std::unordered_set<uint64_t> values() const;
const std::unordered_map<std::string, uint64_t> pairs() const;

private:

// use pimpl to improve performance
const Code_details_::CodeImpl* const _impl;
friend std::ostream& operator<<(std::ostream& os, const Code& code);

export_smart_ptrs(Code)

} // end namespace Encoding

PROCSIM_NAMESPACE_END

#endif // PROCSIM_ENCODING_CODE_HPP_
/*!
 * The following is part of the Procsim Library written and developed
 * by Kevin Brightwell and Ramesh Raj under the supervision of
 * Dr. Kenneth McIssac at Western University, Canada.
 * 
 * \author Kevin Brightwell
 * 
 * This software is provided as is and all rights are maintained by
 * Kevin Brightwell
 */

#include "Algorithm.hpp"
#include "procsim/Macros.hpp"
#include "procsim/util/Math.hpp"
#include <cstdint>
#include <iostream>
#include <memory>

PROCSIM_NAMESPACE_START

namespace encoding
{

// Used as a marker for an Algorithm as an Operand
class PROCSIM_EXPORT Operand PROCSIM_FINAL : public Algorithm
{
public:
    static PROCSIM_CONSTEXPR const char TYPE_NAME[] = "Operand";

    inline Operand(const bits bit_width, const Definition& definition) :
        Algorithm(definition), _bit_width(bit_width) {}

    inline Operand(const Operand&) = default;
    virtual ~Operand() = default;

    const bool operator==(const Operand& other) const;

PROCSIM_NAMESPACE_END
/// Return the *bit* width of the operand
inline const bits width() const PROCSIM_NOEXCEPT { return _bit_width; }

/// Return the *byte* width of the operand
inline const bytes byteWidth() const PROCSIM_NOEXCEPT { return _bit_width; }

private:
friend std::ostream& operator<<(std::ostream& os, const Operand& enc);

/// Width of the encoded operand
const bits _bit_width;

export_smart_ptrs(Operand)

} // end namespace encoding

PROCSIM_NAMESPACE_END

#endif // PROCSIM_ENCODING_OPERAND_HPP_
A.4 Core

Listing A.9: Source for procsim/core/Instruction.hpp.

```cpp
#pragma once
#ifndef PROCSIM_INSTRUCTION_HPP_
#define PROCSIM_INSTRUCTION_HPP_

/*!
 * The following is part of the Procsim Library written and developed
 * by Kevin Brightwell and Ramesh Raj under the supervision of
 * Dr. Kenneth McIssac at Western University, Canada.
 *
 * \author Kevin Brightwell
 *
 * This software is provided as is and all rights are maintained by
 * Kevin Brightwell
 */

#include "procsim/Macros.hpp"
#include "procsim/Types.hpp"

#include "procsim/util/Units.hpp"

#include <functional>
#include <iostream>
#include <memory>
#include <string>
#include <unordered_map>

#include <boost/assert.hpp>
#include <boost/variant.hpp>

PROCSIM_NAMESPACE_START

// forwards:
namespace encoding
{
    class Code;
    class Operand;
}

class Proc;

PROCSIM_NAMESPACE_END
```
// end forwards

class PROCSIM_EXPORT Instruction PROCSIM_FINAL {

public:
    /// The type of function all Instructions use when executing
    typedef std::function<void(Proc&, encoding::Operand&)> exec_fn_t;

    struct Parameters {
        /// Byte codes for encoding, max size uint64
        const std::shared_ptr<encoding::Code> code;

        /// Number of cycles for function
        const uint8_t cycles;

        /// The encoding of the operand
        const std::shared_ptr<encoding::Operand> opEncoding;

        const exec_fn_t exec;
    };

    /// Name of `this` type, all subclasses of `ConfObj` must have this.
    static PROCSIM_CONSTEXPR const char TYPE_NAME[] = "Instruction";

    /*!
    * Constructs a new instance <i>outside</i> of Lua
    * \param name Name of this instance (used with path)
    * \param parent A pointer to the parent
    */
    Instruction(const optional<std::string>& name,
                const std::shared_ptr<encoding::Code>& code,
                const uint8_t& cycles,
                const std::shared_ptr<encoding::Operand>& opEncoding,
                const exec_fn_t& exec);

    Instruction(Parameters&& params, const optional<std::string>& name
                = nullopt);

    Instruction(Instruction&&) = default;
    Instruction(const Instruction&) = default;
    Instruction() = default;
bool operator==(const Instruction& other) const;

/// Name (mnemonic) for this instruction
inline const std::string& name() const { return *_name; }
inline void set_name(const std::string& name) { _name = name; }

inline const optional<std::string> name_opt() const { return _name; }
inline const std::shared_ptr<encoding::Code> code() const { return _code; }

/// Get the width of the code
const bytes codeWidth() const;

/// Get the cycles per instruction call
inline const uint8_t cycles() const { return _cycles; }

/// Get the encoding of the operand, if it exists
inline const std::shared_ptr<encoding::Operand> opEncoding() const { return _opEncoding; }

/// Get the execution function
inline exec_fn_t& exec() { return _exec; }

private:
  optional<std::string> _name;

  /// Byte codes for encoding, max size uint64
  std::shared_ptr<encoding::Code> _code;

  ///! Number of cycles for function
  const uint8_t _cycles;

  const std::shared_ptr<encoding::Operand> _opEncoding;

  exec_fn_t _exec;

friend std::ostream& operator<<(std::ostream& os, const Instruction& ins);
};
#endif // PROCSIM_INSTRUCTION_HPP_
Listing A.10: Source for procsim/core/Proc.hpp.

```cpp
#pragma once
#ifndef PROCSIM_CORE_PROC_HPP_
#define PROCSIM_CORE_PROC_HPP_

/*!
 * The following is part of the Procsim Library written and developed
 * by Kevin Brightwell and Ramesh Raj under the supervision of
 * Dr. Kenneth McIssac at Western University, Canada.
 */

#include "procsim/Macros.hpp"
#include "procsim/Types.hpp"

#include <memory>
#include <string>
#include <unordered_map>
#include <unordered_set>

PROCSIM_NAMESPACE_START

// Forwards:
namespace time {
    class Clock;
} // time

namespace memory {
    class MemoryUnit;
    class Register;
} // memory

class Instruction;

namespace conf {
    class ProcBuilder;

PROCSIM_NAMESPACE_END
```

class PROCSIM_EXPORT Proc PROCSIM_FINAL
{

public:
  //!< Name of `this` type, all subclasses of `ConfObj` must have
  static PROCSIM_CONSTEXPR const char TYPE_NAME[] = "Proc";

  */!
  * Constructs a new instance <i>outside</i> of Lua
  * \param name Name of this instance (used with path)
  * \param parent A pointer to the parent
  */
  Proc(const std::shared_ptr<time::Clock>& procClock,
       const std::string& name,
       const named_map<time::Clock>& clocks,
       const named_map<Instruction>& instructions,
       const named_map<memory::Register>& registers,
       const std::unordered_set<std::shared_ptr<memory::MemoryUnit>>& memory);

  Proc(Proc&&) = default;
  Proc(const Proc&) = default;
  ~Proc() = default;

  bool operator==(const Proc& other) const;

  const std::string& name() const PROCSIM_NOEXCEPT;

  const std::shared_ptr<time::Clock>& procClock() const PROCSIM_NOEXCEPT;

  const named_map<time::Clock>& clocks() const PROCSIM_NOEXCEPT;

  const named_map<Instruction>& instructions() const PROCSIM_NOEXCEPT;

  const named_map<memory::Register>& registers() const PROCSIM_NOEXCEPT;

```
const std::unordered_set<std::shared_ptr<memory::MemoryUnit>>&
memory() const PROCSIM_NOEXCEPT;

private:
    const std::string _name;
    const std::shared_ptr<time::Clock> _procClock;

    /// all known clocks
    const named_map<time::Clock> _clocks;
    const named_map<Instruction> _instructions;
    const named_map<memory::Register> _registers;
    const std::unordered_set<std::shared_ptr<memory::MemoryUnit>>
    _memory;

friend class conf::ProcBuilder;
friend std::ostream& operator<<(std::ostream& os, const Proc&
    proc);
};

PROCSIM_NAMESPACE_END

#endif // PROCSIM_CORE_PROC_HPP_
```
Curriculum Vitae

Name: Kevin Brightwell

Post-Secondary Education and Degrees:
The University of Western Ontario
London, ON
2009 - 2015 BESc. Computer Engineering
BSc. Computer Science
The University of Western Ontario
London, ON
2015 - 2017 MESc.

Honours and Awards:
Ontario Graduate Scholarship
2015-2016
Best Presentation - Software Engineering
Electrical and Computer Engineering Graduate Symposium
Spring 2016

Related Work Experience:
Limited Duties Instructor
Data Structures and Algorithms for Software Engineering
The University of Western Ontario
Winter 2017

Teaching Assistant
The University of Western Ontario
2015 - 2016