Spring 5-2016

NEUROSim: Naturally Extensible, Unique RISC Operation Simulator

David Eric McNeil
Rose-Hulman Institute of Technology, mcneilde@rose-hulman.edu

NEUROSim:
Naturally Extensible, Unique RISC Operation Simulator

A Thesis

Submitted to the Faculty

of

Rose-Hulman Institute of Technology

by

David Eric McNeil

In Partial Fulfillment of the Requirements for the Degree

of

Master of Science in Electrical Engineering

May 2016

ROSE-HULMAN INSTITUTE OF TECHNOLOGY

Final Examination Report

David E. McNeil — Name

Electrical Engineering — Graduate Major

Thesis Title **NEUROSim: Naturally Extensible, Unique RISC Operation Simulator**

DATE OF EXAM: April 22, 2016

EXAMINATION COMMITTEE:

	Thesis Advisory Committee	Department
Thesis Advisor:	Daniel Chang	ECE
	Yong Jin Kim	ECE
	Mario Simoni	ECE
	Claude Anderson	CSSE

PASSED X **FAILED** _____

ABSTRACT

McNeil, David Eric

M.S.E.E

Rose-Hulman Institute of Technology

May 2016

NEUROSim: Naturally Extensible, Unique RISC Operation Simulator

Thesis Advisor: Dr. Daniel Chang

The NEUROSim framework consists of a compiler, assembler, and cycle-accurate processor simulator to facilitate computer architecture research. This framework provides a core instruction set common to many applications and a simulated datapath capable of executing these instructions. However, the core contribution of NEUROSim is its flexible and extensible design allowing for the addition of instructions and architecture changes which target a specific application. The NEUROSim framework is presented through the analysis of many system design decisions including execution forwarding, control change detection, FPU configuration, loop unrolling, recursive functions, self modifying code, branch predictors, and cache architectures. To demonstrate its flexible nature, the NEUROSim framework is applied to specific domains including a modulo instruction intended for use in encryption applications, a multiply accumulate instruction analyzed in the context of digital signal processing, Taylor series expansion and lookup table instructions applied to mathematical expression approximation, and an atomic compare and swap instruction used for sorting.

Keywords: electrical engineering, computer architecture, RISC, compiler, assembler, simulator

DEDICATION

To my wife, Meg, for her constant encouragement and love. I am truly blessed to have you in my life. And to my parents, David and Dorothy, for their support, love, and wonderful examples.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Daniel Chang, for his time, expertise, and dedication. I appreciate the many hours you spent meeting with me, reviewing this document, and practicing with me for the defense. I would also like to thank the members of my defense committee Dr. Claude Anderson, Dr. Yong Jin Daniel Kim, and Dr. Mario Simoni for their time and valuable suggestions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODE EXCERPTS

LIST OF ABBREVIATIONS

ALU ..... Arithmetic Logic Unit

CAS..... Compare And Swap

CISC..... Complex Instruction Set Computer

CPU ..... Central Processing Unit

DSP..... Digital Signal Processing

EX..... Execute Stage

FPU ..... Floating Point Unit

ID..... Instructions Decode Stage

IF..... Instructions Fetch Stage

IPC..... Instructions Per Cycle

IR..... Intermediate Representation

ISA ..... Instructions Set Architecture

L1..... Level 1 Cache

MAC..... Multiply Accumulate

MEM..... Memory Stage

MOD..... Modulo

NOP ..... No Operation

RISC..... Reduced Instruction Set Computer

SPU..... Special Purpose Unit

WB ..... Write Back Stage

XOR ..... Exclusive Or

1 Introduction

Computers are everywhere and range in complexity from multi-core supercomputers to simple microprocessors. Behind each of these computers is a processor which has undergone a series of design decisions intended to optimize the computer for a specific purpose or a target domain. These design decisions are intended to strike a balance between a variety of factors including speed, power, area, and cost. During the design process, a simulator capable of quantifying the consequences of a given design decision becomes invaluable. They provide a method to quickly determine the merits of a design without requiring the resources of an actual implementation.

The NEUROSim framework provides a cycle-accurate simulator targeting a reduced instruction set computer (RISC) architecture and the surrounding resources for targeting this architecture including an assembler and compiler. The simulator is capable of executing a core instruction set common to many applications, but the primary contribution of NEUROSim is its flexible and extensible design allowing this core instruction set to be augmented by instructions optimized for a specific domain.

1.1 Background Knowledge

The NEUROSim framework is intended to encompass the design process from implementing an algorithm in software to executing that software on a hardware platform. The interaction of software with hardware has multiple layers of abstraction intended to ease human computer interaction as well as generalize this interaction over a variety of platforms. A look at the various components in this stack is necessary in order to fully understand the breadth of NEUROSim.

At a high level, a computer program is written in a programming language. In general this program is intended to be capable of running on multiple hardware platforms. Assuming this language is a compiled language, this program will be passed through an architecture specific compiler which converts the high level language into assembly code for the targeted

platform. The assembly code describes the program in terms of instructions in the hardware's instruction set architecture (ISA). After being compiled, the resulting assembly program is passed through an assembler which converts the assembly program into machine code. The machine code is a binary representation of the assembly program. This binary representation can then be loaded into the memory of a processor and executed on hardware or passed to a simulator capable of simulating the execution. Executing a program in a simulator has the advantage of allowing one to easily record statistics on the underlying hardware beyond what the program has been programmed to output. Simulators allow one to easily change the configuration of the underlying hardware and quickly obtain statistics on the corresponding performance changes. This tight feedback loop of being able to make a design decision and quickly view its results is what makes simulators an indispensable tool in the design process. This is in contrast to requiring design decisions be implemented in actual hardware before their merits can be determined.

1.2 NEUROSim Implementation

The NEUROSim framework includes specific implementations of each of the components in the aforementioned stack as seen in Figure 1. The components of NEUROSim are named after components of a neuron because NEUROSim was originally developed with the intent of optimizing an architecture for simulating neurons. The programming language used by the NEUROSim framework is a subset of C. The NEUROSim compiler is named Axon, the assembler is named Synapse, and the simulator is named Neurosim (not to be confused with NEUROSim which represents the entire framework).

This document begins by examining related work in computer architecture simulators, motivates the contribution of NEUROSim, and analyzes the individual components of NEUROSim. Then this document inspects system design decisions in the context of NERUOSim including execute forwarding, control change detection, floating point unit (FPU) configuration, loop unrolling, recursive functions, self modifying code, branch predictors, and cache

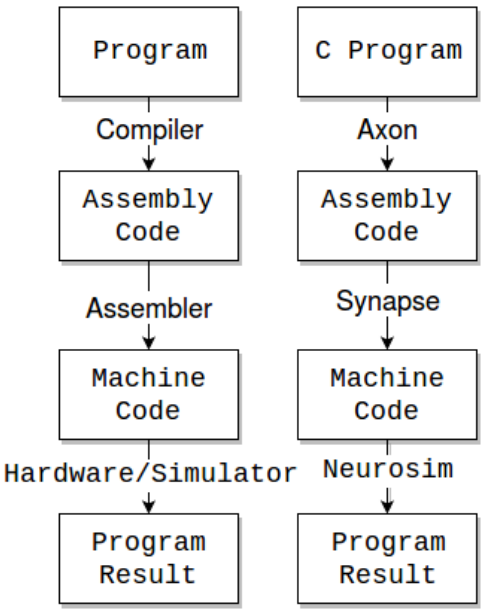

Figure 1: The NEUROSim technology stack.

architectures. To demonstrate its flexible nature, the NEUROSim framework is applied to specific domains namely implementing a modulo instruction intended for use in encryption applications, a multiply accumulate instruction analyzed in the context of digital signal processing, Taylor series expansion and lookup table instructions applied to mathematical expression approximation, and an atomic compare swap instruction used for sorting. Finally, this document concludes by proposing potential future work. The relevant code excerpts can be found at the end of this document in the appendix.

2 Related Work

There exists a plethora of computer architecture simulators, SimpleScalar [4], gem5 [5], and MARSSx86 (Micro-ARchitectural and System Simulator for x86-based Systems) [6], to name a few. These simulators are primarily concerned with supporting cutting edge research in computer architecture. As such they focus on supporting existing ISAs and platforms. For example, gem5 focuses on supporting interchangeable CPU (central processing unit) models for the Alpha, ARM, SPARC, MIPS, POWER, and x86 ISAs [5], and the MARSSx86

provides full system simulation of the x86-64 architecture [6]. These simulators use existing tool chains for handling the compiling and assembling of the programs they run. The ability to use existing tools is excellent for comparing the impact of computer architecture research as it provides a common benchmark by which to compare changes. However, because these simulators use existing tools and implement existing ISAs, these simulators are somewhat limited to analyzing changes which advance existing technologies. For instance, it would be difficult to develop a custom ISA for a target application or optimize a compiler for a specific algorithm using these simulators. These simulators are intended to be used for computer architecture design and analysis, but are not intended to be used as full system design and analysis tools. NEUROSim aims to differentiate itself from these other simulators because it is intended to be used as a design tool from compiler optimizations to ISA design to hardware configuration.

3 Motivation

NEUROSim aims to provide the design tools for custom RISC architectures, ISAs, and software development tools. NEUROSim's target use case is as follows. There is a given domain which could potentially see a performance increase by using a custom instruction. NEUROSim provides a way to quickly add this instruction and evaluate its benefit. First, the logic to execute the instruction is added to the simulator. Then, the ability to translate the instruction is added to the assembler. Finally, the compiler is edited to generate the new instruction. The properties of the new instruction can now be configured to strike a balance between the performance of the algorithm given the new instruction and design parameters from other sources such as cost, area, and power. The results of the simulation can potentially follow two paths. It could be deemed that adding the new instruction is not worth the resources. In which case, the simulator was still successful because it avoided the need to implement the proposed change to come to this conclusion. The other outcome is

that a good balance of the relevant parameters was determined and the designer now has a set of target specifications and an approximation of the corresponding performance benefits.

Essentially, NEUROSim is a tool intended to aid in striking a balance between hardware and software components of a system. Hardware represents the spatial component of the design and consumes space and power. Whereas, software represents the temporal portion of design taking a given amount of time to run. Software layers abstract away components of the hardware layer easing software development. However, the price of these abstractions is a loss of control over the underlying hardware. The purpose of NEUROSim is to give direct control over these abstractions allowing the user to determine what portion of the algorithm should be handled by hardware and what portion should be handled by software. NEUROSim is intended to provide exactly what the standard computer architecture simulators provide, a tight feedback loop between design and results, but with a broader scope. The ability to simulate all components of a system during the design process significantly reduces the resources needed to try a new design. Simulation allows the design process to be easily modeled as an optimization problem where the optimization parameters are the various parameters of the simulator. These parameters are adjusted to find an optimal configuration in regard to a specific output parameter such as latency, area, or power. Simulation even allows for the automation of this optimization process where an algorithm is used to fine-tune parameters until a desired outcome is reached.

4 Axon (Compiler)

Axon is responsible for converting a program written in the supported subset of C to the NEUROSim ISA. Axon is implemented using the LLVM toolchain. "The LLVM Project is a collection of modular and reusable compiler and toolchain technologies [7]."

Basic compiler architecture is composed of a front end, an optimizer, and a back end as seen in Figure 2. The front end converts the source code of a programming language into an

intermediate representation (IR). This intermediate representation is then used to perform optimizations. Finally, this intermediate language is converted to the target assembly. This architecture segments the development of the front end, optimizer, and back end allowing for individuals to tap into any layer of this stack without having to reinvent the other components. Figure 3 illustrates how this architecture can support multiple front end languages which all compile down to the common IR code. This IR code can then undergo a set of common optimizations and finally be converted by the back end to the desired hardware platform.

Figure 2: Compiler architecture [1].

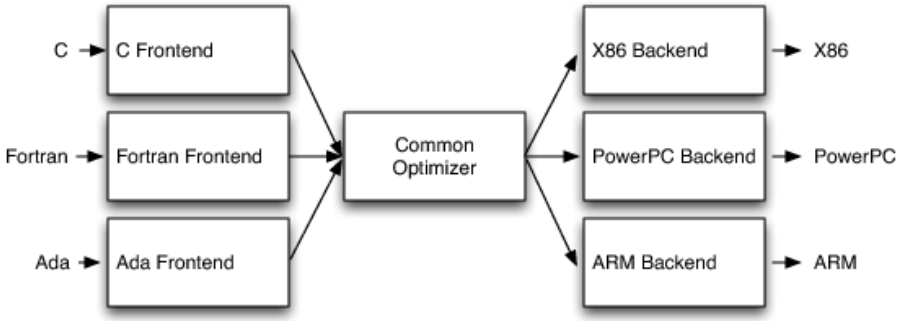

Figure 3: LLVM retargetable architecture [1].

The segmented design of LLVM allows Axon to use an existing C front end known as Clang [8] and a common set of optimizations. However, a new back end was written capable of converting the LLVM IR code to the NEUROSim ISA.

4.1 Supported Syntax

Currently, the Axon compiler supports a subset of C. The available syntax constructs are illustrated in Code 1. The next section will examine NEUROSim's instruction set and how the compiler converts C code into equivalent assembly language expressions.

5 Synapse (Assembler)

Synapse is responsible for converting the assembly program output of Axon to machine code. The tools Flex [9] and Bison [10] were used to construct the assembler. Flex is a lexical analysis generator and Bison is a parser generator. The lexical analyzer produces a token stream given an input file and a specified grammar. Bison generates a parser which given a token stream and grammar applies meaning, or semantics, to the lexical structure. In the case of an assembler, the tokens are instructions, labels, and assembler directives, and the meaning is the corresponding machine code. Intrinsic to the assembler's function are the ISA and register set used by the underlying hardware.

5.1 Register Sets

NEUROSim's ISA works on two sets of thirty-two element 32-bit register files. One register file holds integer values while the other holds single precision floating point values. It is important to realize that the registers in both register files are simply 32-bit values. It is the way these values are interpreted which dictates the type. The integer registers are interpreted as two's complement while the floating point registers are interpreted according to the IEEE floating point standards, shown in Table 1 and Equation 1. The range of integer values is calculated with Equation 2 and the floating point max value is calculated with Equation 3. Floating point values guarantee six digits of precision. The register files were split, instead of using a single sixty-four register register file, because it has been found that partitioning the register file decreases access latency as well as providing the possibility to read from both register files simultaneously [11].

Table 1: Single precision floating point format.

sign{32}	exponent{30-23}	fraction{22-0}

$$FP_{value} = (-1)^{sign} * (1 + \sum_{i=1}^{23} fraction_{23-i} 2^{-i}) * 2^{exponent-127} \tag{1}$$

$$\begin{aligned} min &= -1 \times 2^{32-1} = -2,147,483,648 \\ max &= 2^{32-1} - 1 = 2,147,483,647 \end{aligned} \tag{2}$$

$$(1 - 2^{-24}) * 2^{128} \approx 3.402823466 \times 10^{38} \tag{3}$$

Table 2 lists the integer registers and Table 3 lists the floating point registers. The first column indicates the unique token by which the register is identified in the assembly language. The assembler converts this token to the index of the register in the register file. In the third column, the table indicates whether or not the register is preserved following a call to a function. Common to both register files is a hardwired zero register, a reserved register for use by the assembler and compiler, and sets of both temporary and saved registers. The integer register file also contains special registers which could potentially be used by an operating system kernel as well as registers to track the global data pointer, stack pointer, frame pointer, and return address from a function call.

Table 2: Integer registers.

Register	**Description**	**Preserved**
r0	Hardwired to zero.	NA
at	Reserved for assembler and compiler.	NA
v0, v1	Return values from functions.	N
a0 - a3	Arguments to functions.	N
t0 - t9	Temporary registers.	N
s0 - s7	Saved registers.	Y
k0, k1	Reserved for kernel.	NA
gp	Global data pointer.	NA
sp	Stack pointer.	NA
fp	Frame pointer.	NA
ra	Return address.	N

Table 3: Floating point registers.

Register	**Description**	**Preserved**
fr0	Hardwired to zero.	NA
fat	Reserved for assembler and compiler.	NA
fv0, fv1	Return values from functions.	N
fa0 - fa3	Arguments to functions.	N
ft0 - ft11	Temporary registers.	N
fs0 - fs11	Saved registers.	Y

5.2 Instruction Set Architecture

The instruction set provides the programmer a set of commands which can be used to modify the underlying hardware, specifically the registers and memory, to produce meaningful and useful results. Table 4 shows the three instruction formats present in the ISA. “op”

denotes the opcode of the instruction. An opcode is a unique identifier that the hardware can use to determine the type of the instruction. The opcode is six bits allowing for a total of sixty-four unique instructions. "reg" denotes a register identifier. "imm" indicates an immediate and "address" is an immediate which should be treated as an address to a location in instruction memory.

Table 4: Instruction formats.

Register Format				
op{31:26}	reg0{25:21}	reg1{20:16}	reg2{15:11}	unused{10:0}
Immediate Format				
op{31:26}	reg0{25:21}	reg1{20:16}	imm{15:0}	
Address Format				
op{31:26}	address{25:0}			

Tables 5 – 13 represent the various types of instructions included in the ISA. The assembler assumes the instructions are written with the opcode preceding the operands. For example, "addi s6 t7 -6" would add -6 to the value of register "t7" and then store the result in register "s6". In the instruction type tables, lowercase register names represent the unique register identifier while capitalized register names represent the value of a register. An "f" preceding the identifier or value of a register indicates that it is a floating point register. For some instructions, a register is explicitly defined. For example, the "no operation" instruction (nop) uses implied argument "r0". The use of explicit registers like this indicates that when writing the instruction, specifying the operand is optional. If the register is not included the assembler will assume the indicated operand. All immediate values are sign extended before being used as operands. "MEM" represents a byte addressable memory block. and "PC" represents the program counter which is used to store the address of the current instruction.

Tables 5 – 9 display the integer instructions. These are divided into five instruction types. "R" type instructions are mathematical or logical operations. They also include the

special “brk” and “halt” instructions which indicate to the simulator either a breakpoint or the end of simulation. “I” type instructions are arithmetic and logic operators which take an immediate as the last operand. This type also includes instructions for loading an immediate into the lower and upper sixteen bits of a register. “M” type instructions deal with memory and can either be used to load memory data to a register or store register data to a memory address. “B” type instructions are used to handle conditional jumps to addresses in program memory. It is important to note that these jumps are relative to the current PC and not to an absolute address. This allows the branch instructions to handle a larger range of addresses. The “J” type instructions perform absolute jumps and provide the option to link. A jump and link stores the address of what would have been the next instruction in the return address register. This allows the called function to return to the calling location in program memory after it has completed execution.

Table 5: R type instructions.

Opcode	**Operands**			**Result**
nop	r0	r0	r0	
add	rd	rs	rt	$RD := RS + RT$
sub	rd	rs	rt	$RD := RS - RT$
mul	rd	rs	rt	$RD := RS * RT$
div	rd	rs	rt	$RD := RS \div RT$
and	rd	rs	rt	$RD := RS \ \& \ RT$
or	rd	rs	rt	$RD := RS \mid RT$
xor	rd	rs	rt	$RD := RS \oplus RT$
shl	rd	rs	rt	$RD := RS << RT$
shr	rd	rs	rt	$RD := RS >> RT$
slt	rd	rs	rt	$RD := RS < RT$
brk	r0	r0	r0	Indicates breakpoint
halt	r0	r0	r0	Indicates end of program

Table 6: I type instructions.

Opcode	**Operands**			**Result**
addi	rd	rs	im	$RD := RS + im$
andi	rd	rs	im	$RD := RS \ \& \ im$
ori	rd	rs	im	$RD := RS \mid im$
slti	rd	rs	im	$RD := RS < im$
loi	rd	r0	im	$RD :=$ 0x0000ffff $\& \ im$
hii	rd	r0	im	$RD := (im << 16) \mid RD$

Table 7: M type instructions.

Opcode	**Operands**			**Result**
lw	rd	rs	im	$RD := MEM[RS + im]$
sw	rd	rs	im	$MEM[RS + im] := RD$

Table 8: B type instructions.

Opcode	**Operands**			**Result**
beq	rs	rt	im	$IF(RS = RT) : PC := PC + im$
bne	rs	rt	im	$IF(RS \neq RT) : PC := PC + im$

Table 9: J type instructions.

Opcode	**Operands**			**Result**
j	address			$PC := address$
jal	address			$RA := PC + 4; \ PC := address$
jr	r0	rs	r0	$PC := RS$
jrl	r0	rs	r0	$RA := PC + 4; PC := RS$

Tables 10 – 13 display the floating point instructions. These are divided into four instruction types. "FR" type instructions are general mathematical or logical operations. "FI"

type instructions are used to load an immediate floating point value into a register. "FM" type instructions deal with memory and can either be used to load memory data to a register or store register data to a memory address. It is important to note that these instructions require an integer register and not a floating point register to represent the address. "FB" type instructions are used to handle conditional jumps in program memory.

Table 10: FR type instructions.

Opcode	**Operands**			**Result**
addf	frd	frs	frt	$fRD := fRS + fRT$
subf	frd	frs	frt	$fRD := fRS - fRT$
mulf	frd	frs	frt	$fRD := fRS * fRT$
divf	frd	frs	frt	$fRD := fRS \div fRT$
sltf	frd	frs	frt	$fRD := fRS < fRT$

Table 11: FI type instructions.

Opcode	**Operands**			**Result**
lof	frd	fr0	im	$fRD :=$ 0x0000ffff & im
hif	frd	fr0	im	$fRD := (im << 16) \mid fRD$

Table 12: FM type instructions.

Opcode	**Operands**			**Result**
lwf	frd	rs	im	$fRD := MEM[RS + im]$
swf	frd	rs	im	$MEM[RS + im] := fRD$

Table 13: FB type instructions.

Opcode	**Operands**			**Result**
beqf	frs	frt	im	$IF(fRS = fRT) : PC := PC + im$
bnef	frs	frt	im	$IF(fRS \neq fRT) : PC := PC + im$

Synapse is also capable of handling three assembly directives listed in Table 14 as well as labels which end in a colon. Labels provide a simple way for one instruction to refer to the memory address of a specific instruction or datum.

Table 14: Assembly directives.

Directive	**Meaning**
.text	Start of the text segment (program instructions)
.data	Start of the static data segment
.datum	32 bits of data in the data segment

5.3 Sample Compiled Code

Code 2 – 6 are three simple examples of C code and the corresponding compiled assembly instructions. Comments have been included in the resulting assembly code to make it explicit what lines of assembly correspond to which C constructs.

6 Neurosim (Simulator)

The Neurosim simulator is the core of the NEUROSim framework. Neurosim provides cycle-accurate execution of a program on a RISC datapath. Cycle-accurate indicates that the simulator is executing the program cycle-by-cycle, as opposed to simply using heuristics to determine the run time of the program. This is significant because in general the results of a cycle-accurate simulator will better represent the results of actual hardware.

The general idea behind a simulator is that the user can specify and change settings of the target datapath, run the program, and then see the resulting statistics. In Neurosim, the simulator settings are specified using a config file and the resulting statistics written to an output file. Some example settings would be the number of cycles an instruction takes in each stage of the datapath, memory size, cache configuration, and branch predictor algorithm. Examples of some of the statistics Neurosim reports are the count of retired instructions of

a given type, the total number of cycles executed, the accuracy of the branch predictor, the hit rate of the cache, and the size of different memory segments.

6.1 The Datapath

Processor design has two competing design philosophies RISC (reduced instruction set computer) versus CISC (complex instruction set computer) [12]. The tenants of a RISC architecture involve using a relatively small instruction set, consistent instruction length and format, register mapped operands, and relatively shallow pipelines. A CISC architecture generally embraces the converse of these philosophies having many instructions with different lengths and formats which make use of memory mapped operands. The pipeline of CISC architectures are often very deep and complex to facilitate the fetching and decoding of these complex instructions. In fact, one can think of CISC architectures as converting the complex instructions into a sequence of RISC like instructions which are then executed further along in the pipeline.

Neurosim simulates a pipeline very similar to the classic RISC pipeline illustrated in Figure 4. A RISC pipeline was chosen because it simplifies the primary purpose of NEUROSim, which is to provide for the addition of domain specific instructions. RISC processors are also more common in embedded systems, and embedded systems often involve developing a system optimized for a specific task. As such a RISC datapath was a natural choice for Neurosim. The software representations of the datapath was constructed with the intent of representing the hardware equivalent as faithfully possible. This is important because it allows users to make changes which are more likely to reflect the capabilities of actual hardware.

The simulated datapath is broken up into five distinct stages. The first stage is instruction fetch (IF) which simply retrieves the instruction at the address of the PC from instruction memory. The next stage, instruction decode (ID), decodes the instruction, determining the opcode of the instruction and producing the correct operands for subsequent stages. The

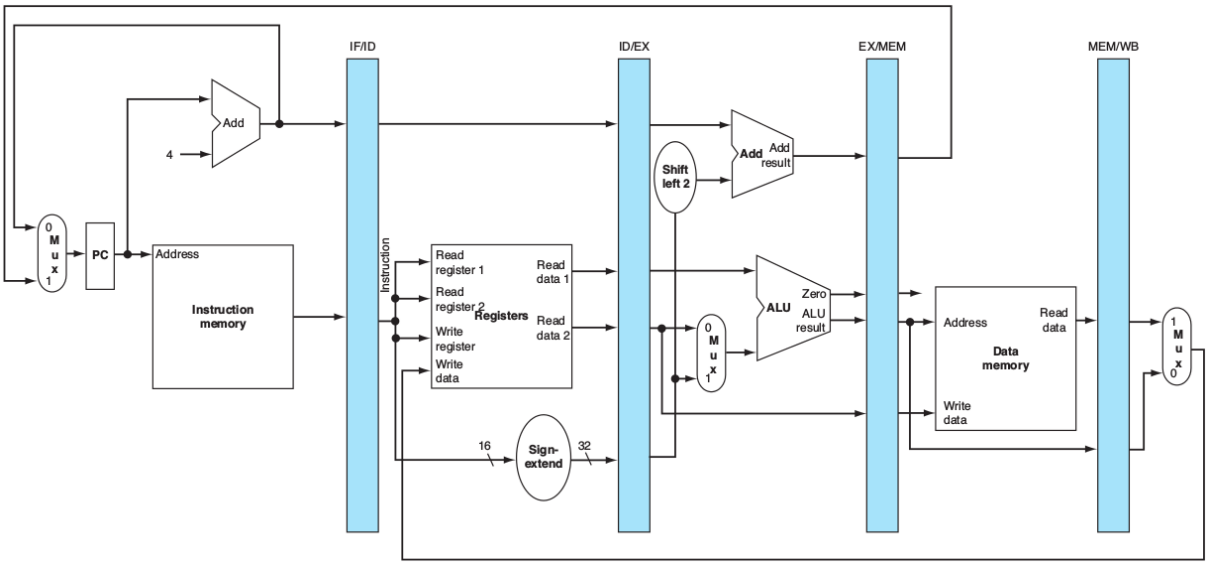

Figure 4: RISC datapath [2].

register file in Figure 4 is a black box representation of the two register files Neurosim uses. The datapath chooses the correct register file to read from either integer registers or floating point registers. After the instruction has been decoded, the operands enter into the execute stage (EX). Here the arithmetic logic unit (ALU) performs the correct operation on the operands. In the case of the Neurosim pipeline there are actually three potential logic units: the ALU, floating point unit (FPU), and the special purpose unit (SPU). The ALU is used for integer operations, the FPU operates on floating point values, and the SPU is used for special instructions which have been added to the ISA. The next stage is the memory stage (MEM) where the instruction can read from or write to data memory. Finally, the instruction enters the write back (WB) stage where the value computed by the instruction is written back to the appropriate register.

Neurosim implements a pipelined datapath meaning that the datapath does not wait to finish execution of an instruction to begin executing the next instruction. Instead, after one instruction leaves the IF stage the next instruction is fetched and execution begun. This significantly increases the instruction throughput of the datapath.

This concludes an overview of the three components of NEUROSim. To summarize,

Axon converts a source code file written in C to the supported assembly language. Synapse converts this textual assembly language to its binary equivalent. This binary file can then be passed to Neurosim for simulation.

7 Hardware Design Decisions

Neurosim has been structured in such a way as to allow for modifications to the base datapath. This allows users to analyze the value of datapath enhancements in the context of a specific algorithm. Three such modifications are built into Neurosim. These optimizations will be analyzed with very simple examples, but these examples could easily be extrapolated to larger more complex algorithms.

7.1 Execute Forwarding

A data hazard is one of the most common hazards in a standard RISC pipeline. This situation arises when an instruction requires the data of a previous instruction which has not written the data back to the register file. Figure 5 illustrates this scenario. The simple fix is to simply stall the pipeline until the instruction has had a chance to write the necessary data. A more efficient method is to implement an execute forwarding unit. The unit is responsible for looking at the register operands of the incoming instruction and determining if the data should be retrieved from the register file or the output of the execute stage. Code 8 presents a very simple example program used to demonstrate the merits of execute forwarding.

An abbreviated version of the statistics from running the program in Neuosim are presented in Table 15. Each column represents a unique run of the simulator given different settings. In this case without and with execute forwarding. Statistic tables such as these are very common throughout this document and are the means by which various algorithms or hardware configurations are analyzed. In this case the first four rows represent the count of total retired instructions of the indicated type. Clearly, using execute forwarding results in a

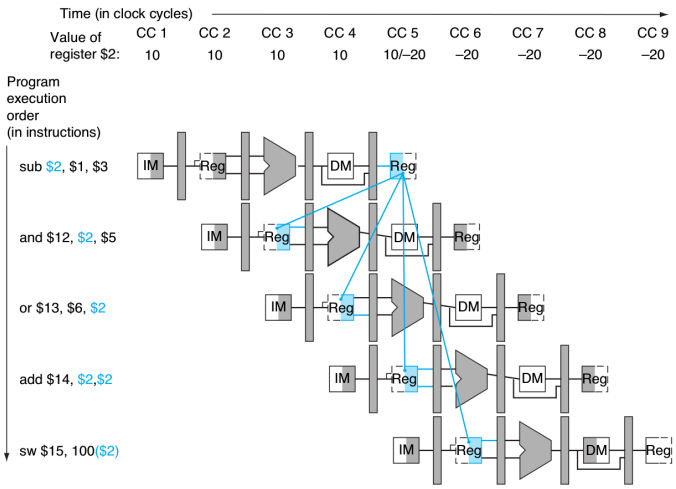

Figure 5: Data hazard [2].

decrease in the number of no operations, "nops." The "NonNop Instructions" entry povides a count of the total number of retired instructions without counting "nops." This a measure of the amount of useful work done by the processor. In this case the datapath executed one addition instruction, one subtraction instruction, and two load immediate instructions, but the simulator reports five "NonNop Instructions" this is because the "halt" instruction, which is used to indicate the end of execution to the simulator, is also counted. The "Cycles" row is simply a count of the total number of cycles, including cycles spent on "nops," executed by the simulator before the simulator encountered the halt instruction. In this example, the "Cycles" row clearly illustrates the merits of using execute forwarding decreasing the total number of cycles from 13 to 9. The instructions per cycle (IPC) row provides a measure of the throughput of the datapath. Ideally this number would be one, indicating that every cycle an instruction is retired. There are many more statistics Neurosim reports which will be discussed as they arise.

Table 15: Execute forwarding statistics.

	No Forwarding	**With Forwarding**
nop	8	4
add	1	1
sub	1	1
loi	2	2
NonNop Instructions	5	5
Cycles	13	9
IPC	0.384615	0.555556

7.2 Control Change Detection

The pipelined structure of a RISC datapath works well until it encounters an instruction which changes the PC. Such instructions include jumps and taken branches. The datapath must now flush all of the previous instructions it has begun executing and fetch the correct instruction. This can severely reduce the IPC of the processor. The solution to this problem is to try and detect a control change as soon as possible. The traditional pipeline handles control change in the execute stage. However, hardware can be added to detect a jump or branch instruction in the decode stage or even as early as the fetch stage. Code 9 presents an example program used to demonstrate the merits of early control change detection with the statistics summarized in Table 16. The "Cycles" row can again be used to clearly demonstrate that early control change detection can improve pipeline performance.

Table 16: Control change detection statistics.

	EX Detection	**ID Detection**	**IF Detection**
nop	1,014	913	812
slt	101	101	101
addi	102	102	102
loi	3	3	3
lw	201	201	201
sw	102	102	102
bne	101	101	101
j	102	102	102
NonNop Instructions	713	713	713
Cycles	1,727	1,626	1,525
IPC	0.41285	0.43849	0.46754

7.3 FPU Configuration

A fundamental detail of an instruction in a RISC style processor is how many cycles it takes to complete a given stage. For most instructions, it is assumed that it only takes one cycle. However, it is typical that floating point instructions can take orders of magnitude longer in the execute stage than integer instructions. Neurosim provides for the easy customization of instruction cycle counts. A potential use case for this would be determining how fast to make the FPU. Neurosim quickly allows one to see the performance increase of a floating point unit with various cycle times. Code 10 presents an example program used to analyze different FPU execute cycle counts with the statistics summarized in Table 17. These statistics clearly demonstrate what one would expect. Increasing the cycle count of a stage significantly decreases the IPC slowing down the entire pipeline. The manual configuration of stage cycle counts will be used extensively as instructions are added to the core ISA.

Table 17: FPU configuration statistics.

	1 Cycle	**10 Cycles**	**40 Cycles**
nop	10	55	205
addf	1	1	1
subf	1	1	1
mulf	1	1	1
divf	1	1	1
sltf	1	1	1
lof	2	2	2
hif	2	2	2
NonNop Instructions	10	10	10
Cycles	20	65	215
IPC	0.50000	0.15384	0.046512

7.4 Conclusion

These examples clearly demonstrate that implementing an execute forwarding unit, early control change detection, or decreasing the number of cycles an FPU needs to execute increases performance. However, the fact that these design decisions improve performance is not the point. Instead it is to show that these design decisions can be implemented in NEUROSim allowing users to determine if the benefits of implementing a datapath enhancement are worth the increase in other potential variables, such as area and power, for a given algorithm.

8 Software Design Decisions

Neurosim also provides a testbed for quickly analyzing different algorithm implementations and software design techniques. The following sections analyzes three design decisions which are particularly relevant in developing algorithms for embedded applications with limited resources.

8.1 Loop Unrolling

Loop unrolling is a technique used to optimize the execution time of a loop by attempting to decrease the number of instructions spent on loop overhead. Code 11 presents a simple program which adds all of the elements in an array together. The code also provides a macro "LOOP" which allows the number of times the loop is unrolled to be changed. The statistics from running the program given different settings of "LOOP" are presented in Table 18 and in Figure 6.

Table 18: Loop unrolling statistics.

	LOOP=1	**LOOP=5**	**LOOP=10**
nop	18,014	13,214	12,614
add	2,000	2,000	2,000
shl	1,000	1,000	1,000
slt	1,001	201	101
addi	1,002	1,002	1,002
loi	4	4	4
hii	1	1	1
lw	5,002	2,602	2,302
sw	2,004	1,204	1,104
beq	0	0	0
bne	1,001	201	101
j	1,002	202	102
NonNop Instructions	14,018	8,418	7,718
Cycles	32,032	21,632	20,332
IPC	0.43763	0.38915	0.37960
TextSize	29	57	92

Clearly unrolling the loop decreases the number of cycles needed to execute the program. The statistics reported by Neurosim clearly illustrate where this performance increase is coming from. Fewer "j" and "bne" instructions are executed because the loop is iterated fewer times. However, the unrolling produces diminishing returns and comes at the cost of substantially increasing the number of instructions stored in instruction memory. This is expressed in the "TextSize" row of the statistics. This statistic represents the number of words stored in the text, or instruction, portion of memory. Because each instruction is one

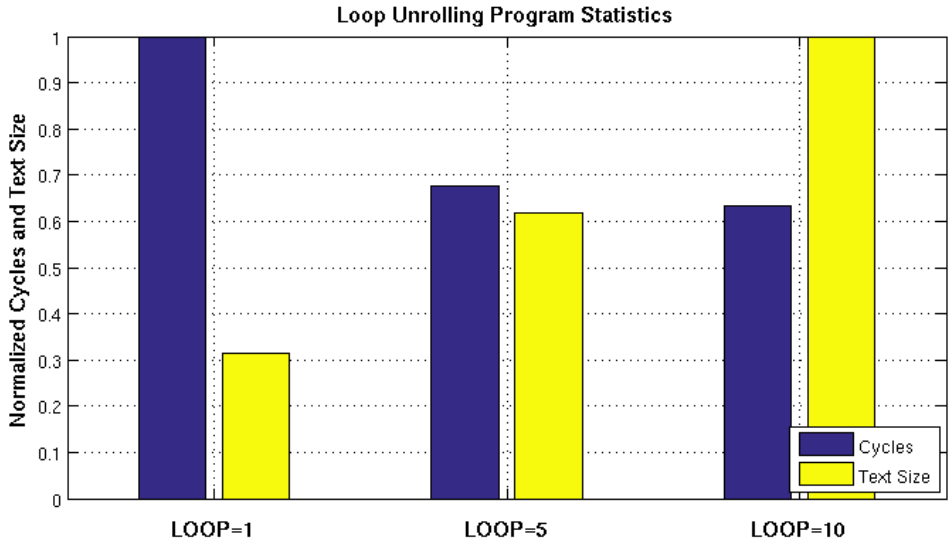

Figure 6: Loop unrolling statistics graph.

word, this statistic provides a count of the total number of instructions. This illustrates an example where Neurosim provides the user the ability to strike a balance between the loop unrolling performance improvements and the increased memory needed for the text segment.

8.2 Recursive Function Calls

The factorial function is a canonical example of a function which can be more intuitive when implemented recursively as seen in Code 12. This implementation will be compared with a loop version of the algorithm as seen in Code 13. The statistics from running the program given the factorial implementations are presented in Table 19.

Given the NEUROSim environment the loop implementation runs in approximately 40% fewer cycles compared to the recursive implementation. This is primarily due to the overhead required to call a function illustrated by the considerable decrease in the number of "jr" and "jrl" instructions. The simulator statistics also provide information on the stack size and shows a potential danger of recursive functions. The used stack space for the recursive implementation is significantly greater than that of the loop implementation as with each new function call more memory is allocated on the stack. As such there is a potential

danger of encountering a stack overflow especially in embedded applications where stack space is limited. It should be noted that this problem can be easily reconciled with tail call optimization and in general is not a reason to avoid recursive functions.

Table 19: Factorial algorithm statistics.

	Recursive	**Loop**
nop	453	265
add	20	1
mul	19	19
slt	21	21
addi	44	25
loi	65	6
hii	39	1
lw	83	104
sw	68	49
bne	21	21
j	22	22
jr	21	2
jrl	21	2
NonNop Instructions	445	274
Cycles	898	539
IPC	0.49555	0.50835
TextSize	43	40
StackSize	44	8

8.3 Self Modifying Code

Self modifying code is one of the lowest level manifestations of the concept that code is simply data and as such can be manipulated by a program like other data. This is a very common concept in higher level languages that implement powerful macro systems. At the assembly language level, self modifying code involves writing a value to an address in instruction memory. Now when the processor reads this address the new instruction will be executed. This is potentially a very dangerous tool because the instruction could potentially be invalid. However, this technique can often be used to pack considerable functionality into a few number of instructions.

RISC datapaths are often implemented as Harvard architectures, where the instruction and data memory are separated. This allows the datapath to access both blocks of memory in a single clock cycle. However, this makes writing self modifying code more difficult because the instructions and data are not in the same address space. The converse is a Von Neumann architecture which uses a single memory block and a unified address space. Neurosim allows for either architecture allowing algorithms which use self modifying code to be easily implemented. Code 14 shows an example of a self modifying code algorithm and Table 20 shows the corresponding statistics proving that Neurosim actually executes the modified instructions.

Table 20: Self modifying code statistics.

nop	add	sub	mul	div	addi	loi	lw	sw	bne	j
30	1	1	1	1	4	5	5	4	4	1

9 Branch Prediction

It is estimated that in an average program, one in three instructions are branch instructions. Therefore, being able to correctly predict which branches will be taken is of extreme importance. If the datapath is unable to successfully predict the correct outcome of the branch valuable cycles will be wasted fetching and decoding instructions that will not be used.

Code 15 presents an example test case for evaluating the performance of various branch predictors. This program takes an array of 1,000 values which have a range from 0 to 100 and counts how many values are less than 10, less than 15, less than 20, and so on up to less than 100. The intuition behind how this tests branch predictors is that the tests against smaller values will generally not be taken whereas tests against greater values are more likely to be taken. A good branch predictor should be able to pick up on this pattern. The last line of the main test loop repeatedly divides the current value by two until the value reaches zero; this is to test the branch predictor's ability to find patterns in loops.

9.1 Static Branch Predictors

The predictions of static branch predictors can be deterministically resolved before run time and will always select the same prediction for a given branch instruction. Four examples of static predictors are always taken, always not taken, forwards taken, and backwards taken. What each of these predictors do is fairly evident from their names. Always taken predicts the branch is always taken. Always not taken always predicts the branch is not taken. This is essentially the same as having no branch predictor. Forward taken predicts taken if the branch offset is greater than the address of the current branch instruction and not taken otherwise. Backwards taken predicts taken if the branch offset is less than the address of the current branch instruction and not taken otherwise. Forward and backward taken are particularly well suited for loops in which the majority of the time the program will simply branch back to the beginning of the loop. Which one is more effective depends on whether a branch is used to go back to the start of the loop or a branch is used to break out of the loop. The results of using each static branch predictor while running the test code is presented in Table 21. This table shows that Neurosim is capable of presenting a complete breakdown of what branches were taken and the predictions of the predictor.

Table 21: Static branch predictor statistics.

	Always Taken	**Always Not Taken**	**Forward Taken**	**Backward Taken**
nop	108,182	111,870	98,684	121,368
beq	6,749	6,749	6,749	6,749
bne	10,565	10,565	10,565	10,565
NonNop Instructions	85,302	85,302	85,302	85,302
Cycles	193,484	197,172	183,986	206,670
IPC	0.44087	0.43263	0.46363	0.41275
Correct Taken	9,579	0	8,579	1,000
Predicted Taken	17,314	0	10,565	6,749
Actual Taken	9,579	9,579	9,579	9,579
Correct Not Taken	0	7,735	5,749	1,986
Predicted Not Taken	0	17,314	6,749	10,565
Actual Not Taken	7,735	7,735	7,735	7,735
Accuracy	0.55325	0.44675	0.82754	0.17246

9.2 Dynamic Branch Predictors

The predictions of dynamic branch predictors are unknown until runtime and have the potential to predict differently for the same branch instruction at different times in the execution of the program. A simple dynamic branch predictor randomly chooses taken or not taken. A more complex branch predictor is a local 2-bit saturating counter, Figure 7. In this type of branch predictor, the datapath keeps a history, represented by 2-bits, for each unique branch in the program. The prediction is made based on the current state of the history and the state is updated after the actual branch address is known.

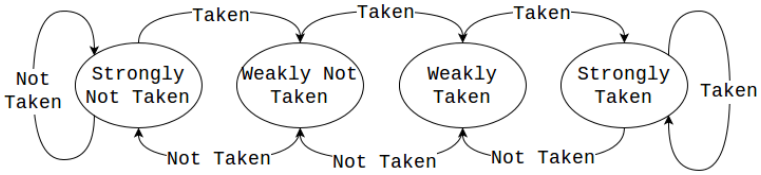

Figure 7: 2-bit saturating counter state diagram.

Another dynamic predictor is the global predictor which uses one 2-bit saturating counter history for all branches in the program. This predictor has the benefit of using less hardware because only one table is needed, but the branch history will be corrupted by different branches. The results of using each dynamic branch predictor while running the test code is presented in Table 22. Figure 8 shows the accuracy and normalized cycle counts of both the static and dynamic branch predictors.

Table 22: Dynamic branch predictor statistics.

	Random	**Local History**	**Global History**
nop	110,170	96,814	101,572
beq	6,749	6,749	6,749
bne	10,565	10,565	10,565
NonNop Instructions	85,302	85,302	85,302
Cycles	195,472	182,116	186,874
IPC	0.43639	0.46839	0.45647
Correct Taken	4,781	8,436	6,813
Predicted Taken	8,712	9,344	8,477
Actual Taken	9,579	9,579	9,579
Correct Not Taken	3,804	6,827	6,071
Predicted Not Taken	8,602	7,970	8,837
Actual Not Taken	7,735	7,735	7,735
Accuracy	0.49584	0.88154	0.74414

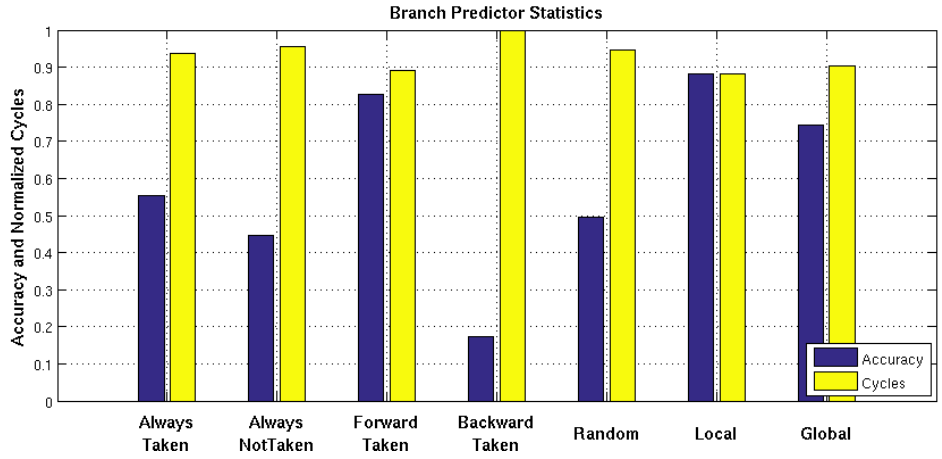

Figure 8: Branch predictor statistics graph.

The statistics from the the branch predictor test program follows intuition. As branch predictor accuracy increases, the total number of cycles needed to run the program decreases. This is because the pipeline wastes fewer cycles fetching and then flushing the wrong instructions. Choosing a branch predictor presents a design decision with distinct trade-offs. A local branch predictor certainly has the best performance with an accuracy of 88%. However, this comes with the cost of expensive hardware. Whereas a simple static forward taken predictor performed almost as well with an accuracy of 82%. Obviously, these performance numbers

are heavily dependent on the algorithm. NEUROSim provides a framework to implement an algorithm and compare the performance given different branch predictors. This prevents overengineering at the microarchitecture level when developing an application, for example implementing a complex local predictor when a simple static predictor would have worked comparably well.

10 Cache Architecture

Memory operations can easily take several orders of magnitude longer than normal datapath operations, necessitating the ability to cache results from memory. The general theory behind memory architecture is that ideally memory would be large and fast. However, memory is slow and large, but caches are fast and small. By using the two in tandem and having effective cache maintenance a memory module that appears to be large and fast can be achieved as seen in Figure 9.

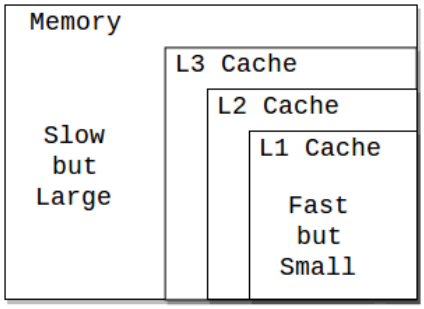

Figure 9: Basic cache structure.

Neurosim is currently equipped with a level 1 (L1) cache and allows for the custom setting of cache hit latency and cache miss latency. Neurosim allows for the custom configuration of the L1 cache size, associativity, and cache line size. An example configuration is illustrated in Table 23. The cache is 64 words in size with an associativity of two and line size of four. An incoming memory address is broken into an index used to select a set, a tag used to verify it is the correct line of data, and an offset used to select the correct data block from the line. All cache lines have a valid bit which confirms that the cache line is up to date.

Table 23: Cache example configuration: Size=64, Associativity=2, LineSize=4.

	Way 0	Way 1
set 0	v tag [block0 block1 block2 block3]	v tag [block0 block1 block2 block3]
set 1	v tag [block0 block1 block2 block3]	v tag [block0 block1 block2 block3]
set 2	v tag [block0 block1 block2 block3]	v tag [block0 block1 block2 block3]
set 3	v tag [block0 block1 block2 block3]	v tag [block0 block1 block2 block3]
set 4	v tag [block0 block1 block2 block3]	v tag [block0 block1 block2 block3]
set 5	v tag [block0 block1 block2 block3]	v tag [block0 block1 block2 block3]
set 6	v tag [block0 block1 block2 block3]	v tag [block0 block1 block2 block3]
set 7	v tag [block0 block1 block2 block3]	v tag [block0 block1 block2 block3]

Six different 64 word cache configurations are analyzed under three different access conditions: random indices, a randomized few indices, and linear indices. All tests involve indexing into a 2,048 element array. The random test simply uses 1,024 random values between 0 and 2,047 as indices. Because the size of the array is so much larger than the cache, it is nearly impossible for the cache to perform well. The randomized few indices uses 122 indices randomly selected 1,024 times. Caches with a higher associativity will perform better on this test because the ability to map multiple pieces of data to different cache lines prevents the eviction of data which still needs to be used. The linear test selects 512 random indices and then reads a random number, from 1 to 100, of elements from the array. Caches with a larger line size should perform well under these test conditions as the subsequent pieces of data in the array are prefetched.

Table 24 and Figure 10 summarize the results of running these tests. What should be evident from these tests is that there is no absolute correct cache configuration; it largely depends on the type of memory accesses. This illustrates the value of NEUROSim; it provides a quick testing environment to compare different configurations for a specific application.

Table 24: Cache hit rates.

	Associtivity x Line Size					
Access Type	**1x1**	**1x4**	**2x1**	**2x4**	**4x1**	**4x4**
Random	0.0501	0.0501	0.058116	0.046092	0.056112	0.0501
Random Few	0.527132	0.251938	0.589147	0.27907	0.600775	0.275194
Linear	0.051736	0.749587	0.048595	0.748512	0.044463	0.74719
Average	0.20966	0.35054	0.23195	0.35789	0.23378	0.35749

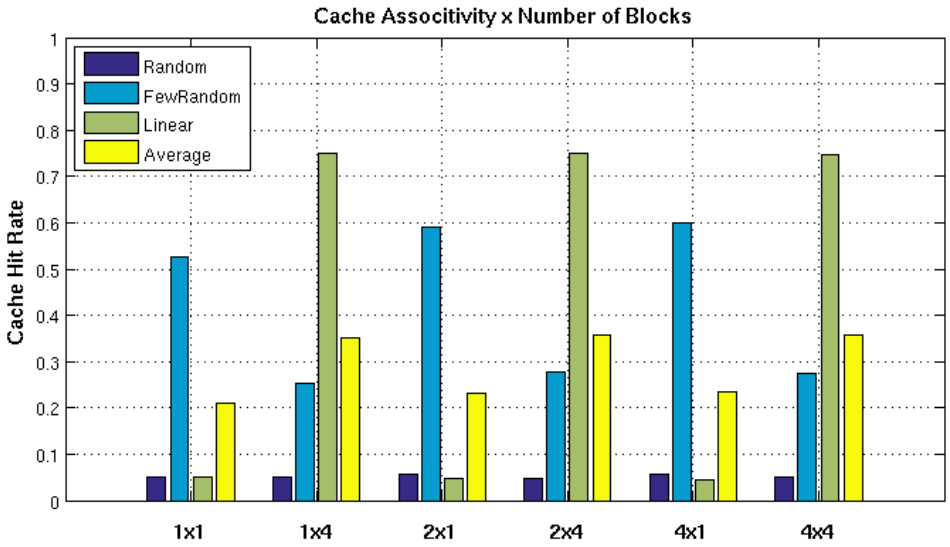

Figure 10: Cache hit rates graph.

11 Setup and Configuration of Examples

In the following sections, instructions will be added to the core ISA and the impact of those instructions evaluated. This is where NEUROSim begins to truly differentiate itself from other computer architecture simulators because it was designed with the purpose of allowing for the addition of new instructions and provides the surrounding tooling to easily accomplish this.

A question presents itself: how to compare the benefits of adding an instruction. The following sections all use the same basic testing environment. First, the desired functionality of the instruction is factored out from the program into a separate function. Then the

algorithm is run with the function called normally and with the function inlined in the program. The inlining of the function removes the overhead associated with the function call giving a more accurate representation of the potential benefits of adding an instruction. Code 19 gives a simple example of factoring out the "add" functionality and then calling the normal and inlined version of the function. After these two tests are run, the inlined portion of the code is then removed and replaced with the new instruction. The program is then executed with different cycle counts for the instruction to evaluate how efficient the hardware would need to be to get the desired performance increase.

The configuration of the simulator, barring the experimentation with the new instruction's cycle counts, is the same for all trials: local branch predictor with control change detection handled in the fetch stage, a zero cycle cache miss penalty, and an FPU execute stage latency of 10 cycles. This configuration was chosen to try and represent a basic system while preventing the statistics from being skewed in favor of the implementation with the custom instruction. The software implementation of the functionality intrinsically executes more instructions providing the possibility for a greater number of branch mispredictions and more cache miss penalties to be incurred. By using the best branch predictor and no cache miss penalties this inequality is mitigated. This configuration can be thought of providing a nearly ideal run of the program, allowing for the sole analysis of the new instruction's performance.

12 Modulo Instruction

Several basic design decisions have been examined in the context of NEUROSim. However, NEUROSim's true strength is seen in the context of adding new instructions to its core ISA. The simplest type of instructions to add are ones that follow the R type format. An example of such an instruction would be adding hardware support for the modulo operator. "Modular reduction, also known as the modulo or mod operation, is a value within Y, such

that it is the remainder after Euclidean division of X by Y. This operation is heavily used in encryption algorithms, since it can 'hide' values within large prime numbers, often called keys [13]." Due to the computational needs of encryption algorithms, the mod operation is often optimized to improve performance. Table 25 presents the proposed instruction. Code 20 provides an example program to test the performance benefits of using a hardware optimized mod instruction. Essentially, this program computes the mod between every element in array "X". It should be noted that the software implementation of mod is very inefficient and better algorithms exist. However, for the sake of showing an example design process in NEUROSim it is sufficient.

Table 25: MOD instruction.

Opcode	**Operands**			**Result**
mod	rd	rs	rt	$RD := RS \bmod RT$

Table 26 and Figure 11 show the statistics of running the program with five different configurations. The first trial is with the modulo functionality called as a function, then the functionality is inlined, and finally three different runs with the new "mod" instruction are run with a 1 cycle, 10 cycle, and 100 cycle execute stage latencies. Clearly, adding a "mod" instruction can significantly improve performance. Even if the instruction takes ten cycles to compute, the program still undergoes a 55% decrease in the number of cycles taken to execute in comparison to the inlined version.

Table 26: Modulo test program statistics.

	Function Call	**Inlined Function**	**mod 1 cycle EX**	**mod 10 cycle EX**	**mod 100 cycle EX**
nop	2,308,686	2,268,682	962,818	1,322,818	4,922,818
add	80,000	80,000	80,000	80,000	80,000
sub	134,766	134,766	0	0	0
shl	80,000	80,000	80,000	80,000	80,000
slt	215,167	215,167	40,401	40,401	40,401
addi	120,202	40,202	40,202	40,202	40,202
lw	939,671	939,665	240,601	240,601	240,601
sw	335,174	335,168	80,402	80,402	80,402
bne	215,167	215,167	40,401	40,401	40,401
j	214,968	174,968	40,202	40,202	40,202
jr	40,000	0	0	0	0
jrl	40,000	0	0	0	0
mod	0	0	40,000	40,000	40,000
NonNop Instructions	2,415,124	2,215,110	682,216	682,216	682,216
Cycles	4,723,810	4,483,792	1,645,034	2,005,034	5,605,034
IPC	0.51127	0.49403	0.41471	0.34025	0.12172

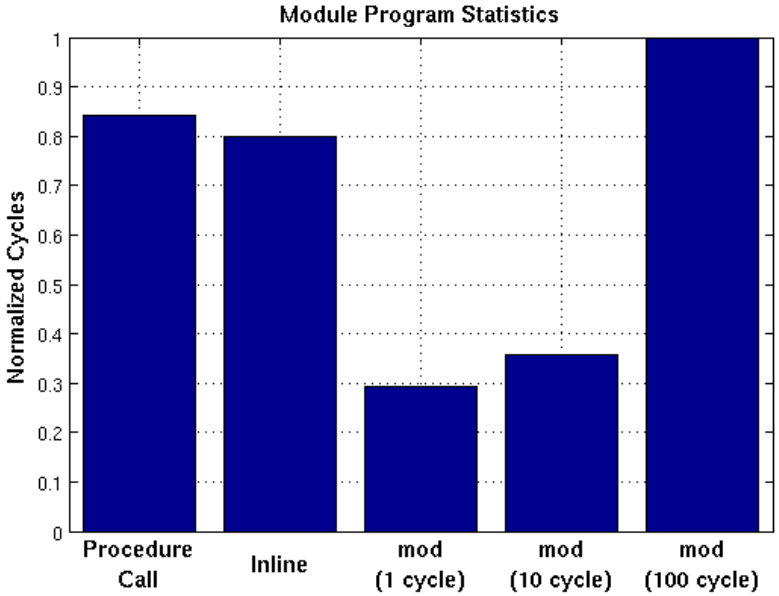

Figure 11: Modulo test program statistics graph.

The statistics from Neurosim explicitly indicate the source of the performance improvements. First, it can be seen that by inlining the function call, the program goes from

executing 40,000 "jr" and "jrl" instructions down to zero. There is also some savings in the number of "addi" and memory operations which are used for maintaining the stack. However, the true performance benefits come when the new "mod" instruction is used, which cuts the number of executed instructions in almost every category. As one would expect, the only instruction count that goes up is for the "mod" instruction itself. The statistics also demonstrate the considerable decrease in IPC as the "mod" instruction execute stage latency is increased. Overall, these statistics provide exactly the information needed for a systems designer to determine which portions of an algorithm should be handled in hardware and which should be handled in software.

13 Digital Signal Processing Example

Digital signal processing (DSP) algorithms involve the manipulation of signals in the digital domain and are commonly done using dedicated embedded systems. With the great popularity of cellphones, DSP specific hardware has become ubiquitous. The following section looks at implementing a dedicated multiply accumulate instruction intended to improve the performance of DSP filter implementations.

13.1 Lowpass Filter

One of the most common algorithms in DSP is low pass filtering. Filtering in general involves shaping an input signal $x[n]$ to produce an output signal $y[n]$. This is accomplished by scaling previous inputs with coefficients, b_k, and scaling previous outputs with coefficients, a_k. Equation 4 presents the the formula for computing the output signal given an input signal and filter coefficients. M and N represent the number of coefficients and the greater of the two is equal to the filter order.

$$y[n] = -\sum_{k=1}^{M} a_k y[n-k] + \sum_{k=0}^{N} b_k x[n-k] \tag{4}$$

For this example, a sixth order elliptic filter will be used. The filter has a passband ripple of 5dB and a stopband attenuation of 40dB. The passband edge frequency is set to be one eighth the sampling frequency of the original signal. The magnitude and phase responses of the filter are illustrated in Figure 12.

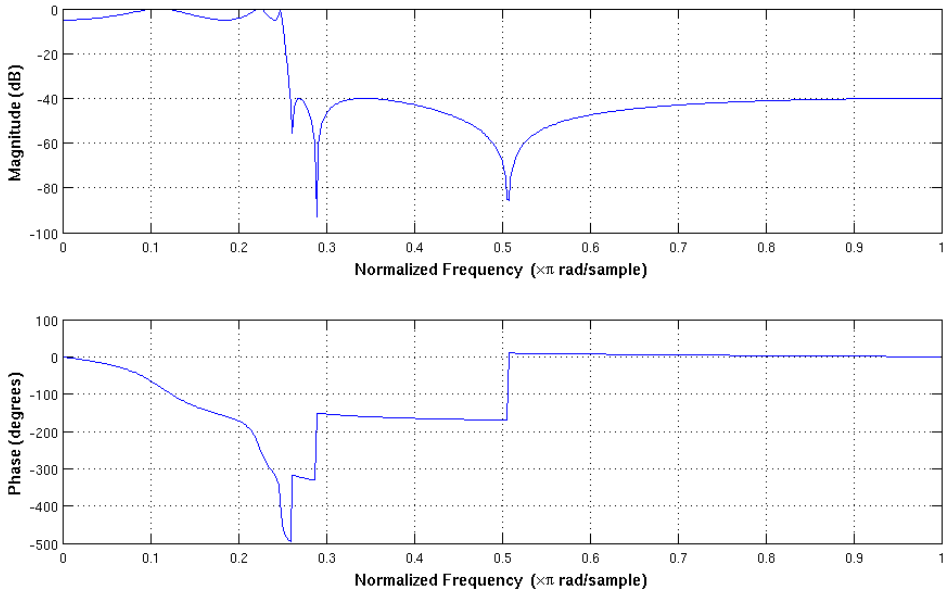

Figure 12: Lowpass filter magnitude and frequency response.

In DSP there are different flow graph representations of the general filter formula. Figure 13 shows one such flow graph, the direct-form-one flow graph, of a sixth order filter. In the flowgraph z^{-1} represents a one sample delay The filter coefficients are listed in Equation 5.

$$\begin{aligned} a_k &= [1.000000,\ -4.544816,\ 9.579983,\ -11.695033,\ 8.686814,\ -3.721021,\ 0.727505] \\ b_k &= [0.018789,\ -0.047933,\ 0.085798,\ -0.094508,\ \ 0.085798,\ -0.047933,\ 0.018789] \end{aligned} \tag{5}$$

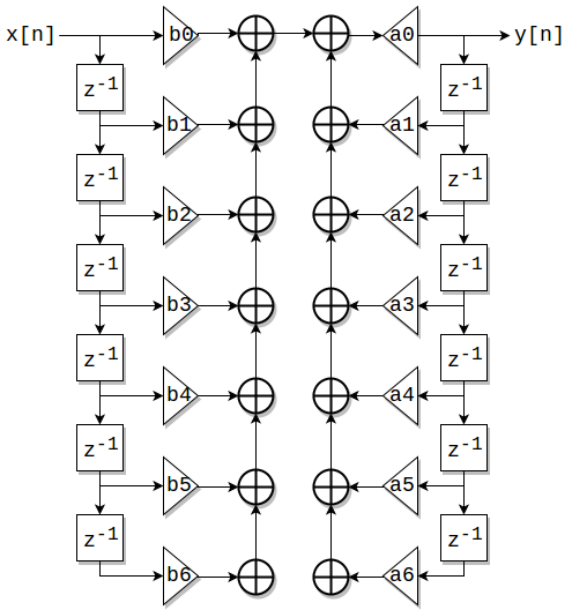

Figure 13: Flowgraph representation of filter.

13.2 Multiply Accumulate (MAC) Instruction

As demonstrated by Equation 4, a filter implementation essentially involves repeatedly multiplying two numbers together and then adding the product to an accumulated result. This has given rise to a special multiply accumulate (MAC) instruction, Table 27.

Table 27: MAC instruction.

Opcode	**Operands**			**Result**
mac	rd	rs	rt	$RD := RD + (RS * RT)$

The need to introduce such an instruction perfectly illustrates the power of a NEUROSim. This is actually a somewhat complicated instruction to implement in hardware because it requires being able to read three operands from the register file. This is a substantial change and presents a series of design problems. The simulator will allow the user to determine if the performance benefits warrant adding the instruction.

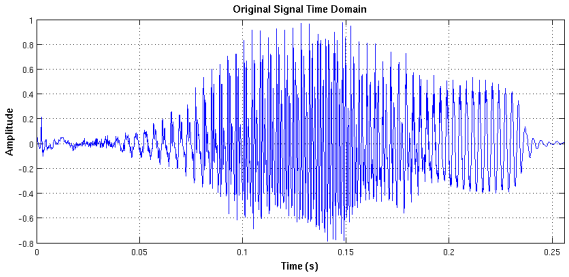

Figure 14: Original signal time domain.

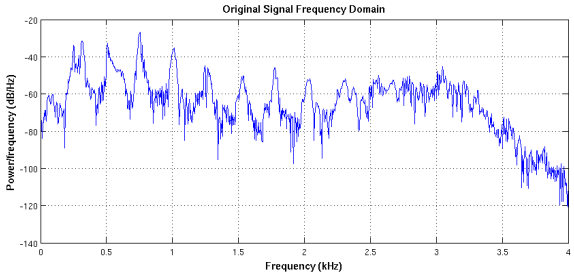

Figure 15: Original signal frequency domain.

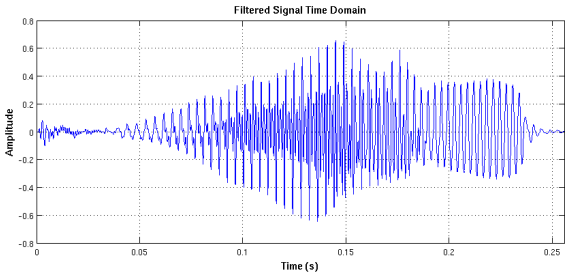

Figure 16: Filtered signal time domain.

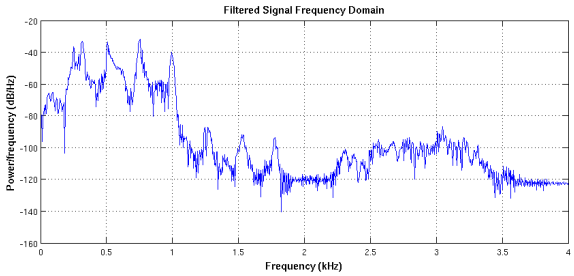

Figure 17: Filtered signal frequency domain.

13.3 Lowpass Filter Implementation

Code 21 shows the implementation of the lowpass filter in software. The implementation uses a fixed-point number system allowing for the use of integers despite the fact that the algorithm deals with floating point values. The fixed point base was chosen to be a power of two allowing for the downscaling and upscaling to be accomplished by a simple shift.

The input signal, Figure 14 and Figure 15, to our filter is 2048 samples of a speech signal sampled at 8 kHz. As such the output signal, Figure 16 and Figure 17, should have a cutoff frequency of approximately 1 kHz. The output plots are plots of the actual output data from running the algorithm on the simulator. The data points were acquired by performing a memory dump of the "Y" array. This demonstrates another potential use of the simulator to confirm that an algorithm actually performs correctly given hardware constraints. It is very common for fixed point DSP algorithms to be susceptible to catastrophic quantization. The simulator proves that the algorithm is possible under the given constraints.

Table 28 and Figure 18 show the statistics of implementing a dedicated MAC instruction. If the MAC instruction could be implemented in a single cycle, the algorithm would be

capable of running in 5.8% fewer cycles than the inline version of the algorithm.

Table 28: Lowpass filter statistics.

	Function Call	**Inline Function**	**mac 1 cycle EX**	**mac 2 cycle EX**	**mac 4 cycle EX**
nop	833,266	776,078	718,902	747,490	804,666
add	91,896	91,896	63,308	63,308	63,308
sub	42,882	42,882	42,882	42,882	42,882
mul	28,588	28,588	0	0	0
shl	59,224	59,224	59,224	59,224	59,224
shr	6,126	6,126	6,126	6,126	6,126
slt	18,386	18,386	18,386	18,386	18,386
addi	73,520	16,344	16,344	16,344	16,344
loi	14	13	13	13	13
hii	5	4	4	4	4
lw	249,154	240,976	240,976	240,976	240,976
sw	138,888	138,875	138,875	138,875	138,875
bne	18,386	18,386	18,386	18,386	18,386
j	16,345	16,345	16,345	16,345	16,345
jr	28,588	0	0	0	0
jrl	28,588	0	0	0	0
mac	0	0	28,588	28,588	28,588
NonNop Instructions	800,591	678,046	649,458	649,458	649,458
Cycles	1,633,857	1,454,124	1,368,360	1,396,948	1,454,124
IPC	0.49000	0.46629	0.47463	0.46491	0.44663

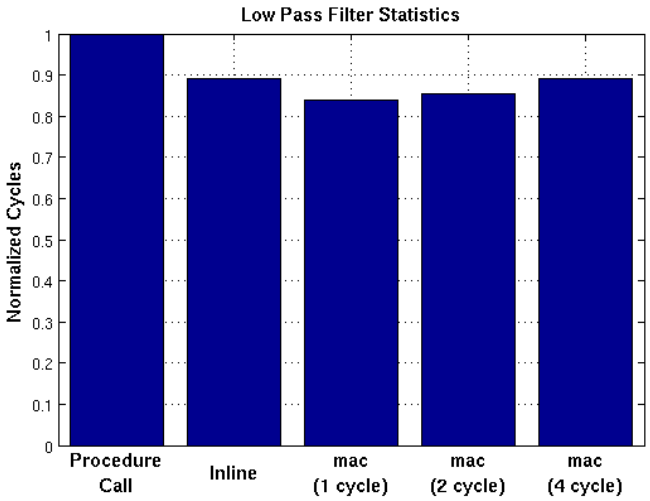

Figure 18: Low pass filter statistics graph.

The increase in performance due to adding a dedicated instruction seems minimal and for this example it is probably not worth the trouble. However, in general a DSP algorithm will be run on much longer data samples than the approximately quarter second sample used in this example. This 5.8% increase in performance will add up as longer samples are run. DSP systems are often real-time systems and as such any performance benefit possible is often needed. For a system dedicated to doing this type of computation, the performance increase could be worth the extra hardware overhead. This demonstrates the primary intent of NEUROSim. To provide the necessary information on performance change to allow the designer to make informed decisions.

14 Mathematical Expression Approximation

Due to its digital nature, a standard processor does a poor job of representing and computing complex, continuous mathematical functions. If these functions use a large dynamic range, floating point becomes a necessity, and the algorithms used to estimate these functions are often computationally intensive. As a result, computing complex mathematical functions often become bottlenecks in a program. Bottlenecks often present a point where hardware optimizations can be used to improve performance. As an example, two methods for approximating the exponential function will be examined in the following section. These methods can easily be extended to other mathematical functions.

14.1 Lookup Table Approximation

A common method of approximating a mathematical function is to use a lookup table to store input values and the corresponding output value of the function over a given range. For example, this is commonly done in DSP processors to represent sinusoidal functions. Neurosim supports a curve lookup instruction as shown in Table 29 for the express purpose of lookup table approximation. This instruction takes an x value in as operand frs and a

curve selection value in as operand rt. The "crv" instruction can potentially support many curves. The curve selection value determines which curve to use. Once the curve is selected, the instruction will perform linear interpolation as shown in Equation 6 to determine the approximated output where x lies on the interval $x_0 < x < x_1$. Figure 19 shows a lookup table approximation of the exponential function on the interval -1.5 to 1.5. Eight equally spaced points were used to form the lookup table. More complex algorithms exist for obtaining a better estimate of the curve by changing the spacing of the points. However, for this example this configuration adequately approximates the curve.

$$y = y_0 + (y_1 - y_0)\frac{x - x_0}{x_1 - x_0} \tag{6}$$

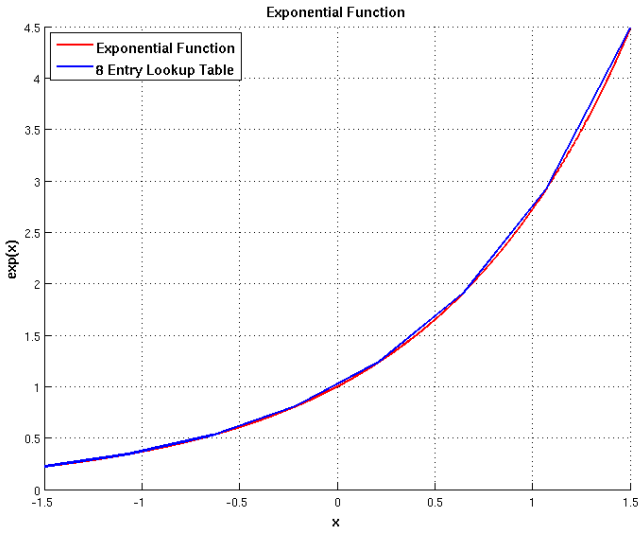

Figure 19: Exponential function with lookup table approximation.

.

Table 29: Lookup table approximation instruction.

Opcode	**Operands**			**Result**
crv	frd	frs	rt	$fRD := curves[RT](fRS)$

14.2 Taylor Series Expansion Approximation

A Taylor series is a common method for approximating an arbitrary mathematical function with a polynomial. Equation 7 provides the general equation where $n!$ represents the factorial of n, $f^{(n)}(a)$ represents the n^{th} derivative of f computed at a. As M increases the approximation better matches the original function. The Taylor series expansion for an exponential is given in Equation 8 where $a = 0$ and $M = 10$. Figure 20 shows the exponential function and the Taylor series approximation for $M = 5$ and $M = 10$. Table 30 shows a new instruction which can be used to approximate the exponential function using the Taylor series expansion with $M = 10$.

$$f(x) \approx \sum_{n=0}^{M} \frac{f^{(n)}(a)}{n!}(x-a)^n \qquad (7)$$

$$e^x \approx \sum_{n=0}^{10} \frac{x^n}{n!} \qquad (8)$$

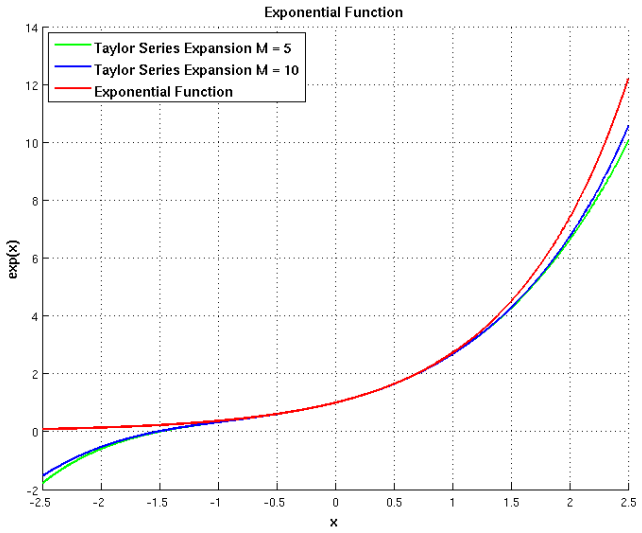

Figure 20: Exponential function with Taylor series approximation.

.

Table 30: Taylor series exponential approximation instruction.

Opcode	**Operands**		**Result**
exp	frd	frs	$fRD := \sum_{n=0}^{M} \frac{fRS^n}{n!}$

14.3 Artificial Neural Network Example

Machine learning is a general term for an algorithm capable of categorizing data or making predictions from data in ways that it was not explicitly programmed to do. Machine learning is becoming more and more prominent because vast amounts of data are gathered and robust methods for finding patterns in the data is needed. A common method for learning complex patterns is to use artificial neural networks. "Neural networks are one of the most beautiful programming paradigms ever invented. In the conventional approach to programming, we tell the computer what to do, breaking big problems up into many small, precisely defined tasks that the computer can easily perform. By contrast, in a neural network we don't tell the computer how to solve our problem. Instead, it learns from observational data, figuring out its own solution to the problem at hand [3]." Because of machine learning's tendency to deal with extremely large datasets, hardware optimizations are often employed to improve performance. This example will examine how using the "crv" and "exp" instructions can improve performance in application to neural networks.

The structure of a single artificial neuron is shown in Figure 21. A neuron is composed of n inputs, x_i, and an equal number of weights, w_i, and a bias b. From these parameters the network computes a value z according to Equation 9. The value of z is then given to an activation function A which is used to simulate the firing of a neuron. For this example, the sigmoid activation function will be used as shown in Equation 10. It is in this activation function that hardware optimizations will be introduced by using the "crv" and "exp" instructions to compute the exponential. The basic idea is that by chaining many of these neurons into a network any function can be estimated [3]. Figure 22 presents an example network of artificial neurons.

$$z = (\sum_{i=0}^{n} w_i * x_i) + b \qquad (9) \qquad A(z) = \frac{1}{1 + e^{-z}} \qquad (10)$$

This example will look at using a network for the most trivial non-linear classification problem, exclusive or (XOR) Table 31. Generally a network undergoes a training phase where the weights and biases are modified in order to minimize the error given a training set. For

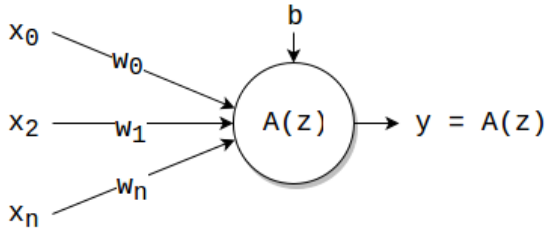

Figure 21: Single artificial neuron.

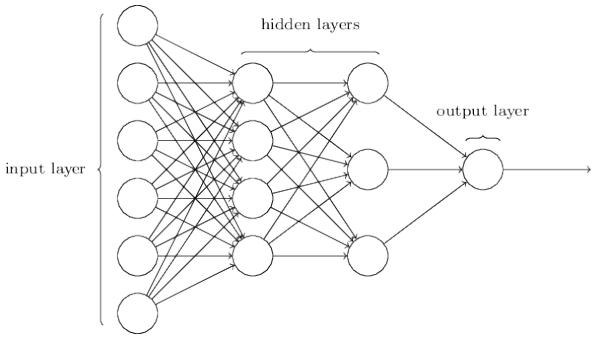

Figure 22: Artificial neural network [3].

Input 0	Input 1	Output
0	0	0
1	0	1
0	1	1
1	1	0

Table 31: XOR logic table.

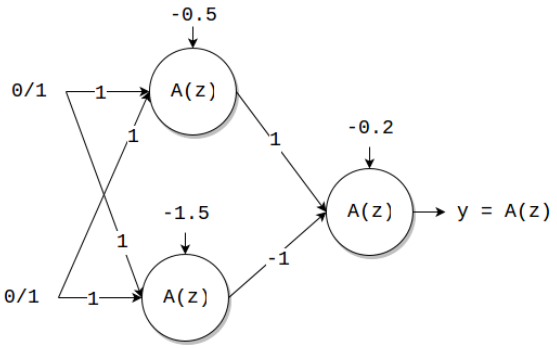

Figure 23: XOR neural network.

this example, predetermined weights and biases are used [14]. As such, this example focuses on the forward propagation step of a network where a given input "propagates" through the network to produce an output. Figure 23 displays the network which implements XOR and Code 22 and Code 23 present implementations of this network using the lookup table and the Taylor series approximation of the exponential function. These are software implementations of computing the exponential. The "expLookupTable" and "expTaylorSeries" will be replaced by the "crv" and "exp" instructions respectively.

Table 32 shows the network output using an ideal exponential function (using the C math library to compute the exponential), the lookup table approximation, and the Taylor series expansion approximation. It should be noted that in order to produce the correct binary output, the output of the network must be rounded. This is because neural networks can only approximate functions and in the case of discrete classification this rounding step is necessary. The simulator provides valuable information on which implementation produces

a result closest to the ideal value. For this specific example, the Taylor series approximation more closely matched the ideal value.

Table 32: XOR neural network output.

Input 0	**Input 1**	**Ideal Output**	**Lookup Table**	**Taylor Series**
0	0	0.498779	0.491996	0.498779
1	0	0.511228	0.505496	0.511228
0	1	0.511228	0.505496	0.511228
1	1	0.498779	0.494335	0.498778

Table 33 and Figure 24 present the statistics of running the algorithm under different circumstances. The lookup table approximation soundly outperforms the Taylor series approximation by almost ten times. However, as was demonstrated this comes at the cost of decreased accuracy in comparison to the ideal value. Clearly, this is an area for dramatic performance increase because a 10 cycle implementation of the "crv" or "exp" instructions provides a 72% reduction in the number of executed cycles compared to the lookup table implementation.

Table 33: Neural network statistics.

	Lookup Table	**Taylor Series**	**crv/exp 1 cycle EX**	**crv/exp 10 cycle EX**	**crv/exp 100 cycles EX**
nop	4,271	41,362	1,246	1,354	2,434
add	94	0	0	0	0
mul	11	0	0	0	0
shl	71	0	0	0	0
slt	59	0	0	0	0
addi	105	58	34	34	34
loi	124	54	18	18	18
hii	76	53	17	17	17
lw	223	58	22	22	22
sw	81	58	22	22	22
beq	12	0	0	0	0
bne	12	0	0	0	0
j	61	565	1	1	1
jr	28	160	16	16	16

jrl	28	160	16	16	16
addf	71	1,848	60	60	60
subf	45	24	12	12	12
mulf	35	588	24	24	24
divf	23	144	12	12	12
sltf	59	696	0	0	0
lof	14	302	14	14	14
hif	14	302	14	14	14
lwf	221	238	46	46	46
swf	47	83	23	23	23
beqf	59	828	0	0	0
crv/exp	0	0	12	12	12
NonNop Instructions	1,574	6,220	364	364	364
Cycles	5,845	47,582	1,610	1,718	2,798
IPC	0.26929	0.13072	0.22609	0.21187	0.13009

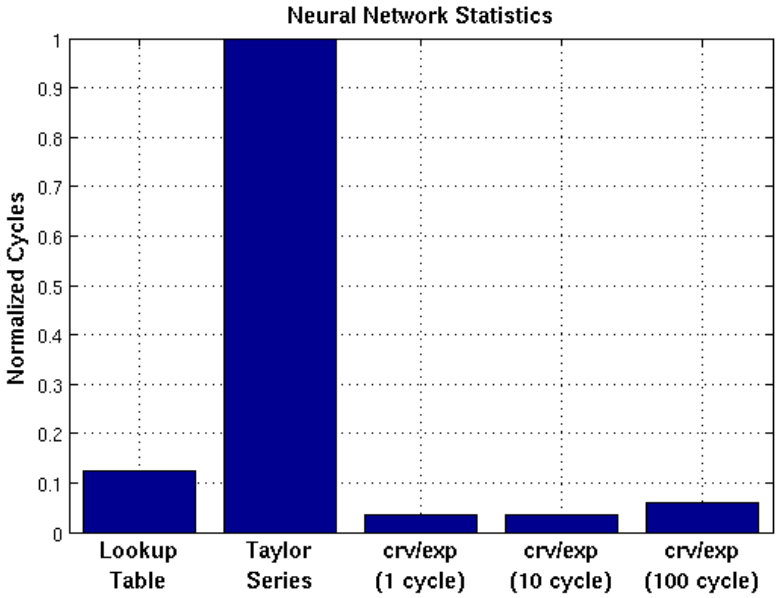

Figure 24: Neural network statistics graph.

While this specific example dealt with approximating the exponential function, NEUROSim is intended to easily allow the addition of new curves for the "crv" function and to develop new instructions which use a Taylor series approximation similar to that of the "exp" instruction.

15 Array Sorting Example

A common application of computers is sorting. A simple sorting algorithm is bubble sort, see Code 24. Bubble sort involves making a pass over each element of the array to be sorted. During the pass each element is compared to the next element and if they are in the wrong relative location the two elements are swapped. Bubble sort makes a number of passes equal to the length of the array ensuring that the resulting array will be sorted. The algorithm is so named because large values tend to "bubble" to the top of the array

15.1 Compare and Swap (CAS) Instruction

Intrinsic to bubble sort is the compare and swap (CAS) step. This is a very expensive operation because it requires two reads from memory, a comparison, and then potentially two writes to memory. If this computation could be done atomically there could be potential for substantial performance increase. The benefit of using a simulator is again displayed as adding the hardware to support such an instruction is non trivial whereas the logic can be quickly added to the simulator. The proposed CAS instruction is presented in Table 35.

Table 35: CAS instruction.

Opcode	**Operands**			**Result**
cas	r0	rs	rt	$IF(MEM[RS] > MEM[RT]):$ $AT := MEM[RS];$ $MEM[RS] := MEM[RT];$ $MEM[RT] := AT;$

Table 36 and Figure 25 show the results of implementing the CAS instruction. The six cycle CAS instruction cuts the total number of instructions by over 40% in comparison to the inlined version. Six cycles for the instruction is fairly reasonable because it allows two cycles for reading in the memory data, two cycles for performing the comparison, and finally two cycles for saving the results. Implementing this functionality in hardware would require a complex design. The simulator allows the designer to first confirm that the performance benefit is worth the effort.

Table 36: Bubble sort statistics.

	Function Call	**Inline Function**	**cas 1 cycle MEM**	**cas 6 cycle MEM**	**cas 20 cycle MEM**
nop	38,483,166	33,245,404	16,775,186	22,012,946	36,678,674
add	3,132,748	3,132,748	1,047,552	1,047,552	1,047,552
shl	3,132,748	3,132,748	1,047,552	1,047,552	1,047,552
slt	2,097,153	2,097,153	1,049,601	1,049,601	1,049,601
addi	4,191,234	2,096,130	2,096,130	2,096,130	2,096,130
loi	2,095,109	6	6	6	6
hii	1,047,553	1	1	1	1
lw	8,104,267	8,104,262	3,145,729	3,145,729	3,145,729
sw	3,922,944	3,922,939	1,049,602	1,049,602	1,049,602
beq	1,047,552	1,047,552	0	0	0
bne	1,049,601	1,049,601	1,049,601	1,049,601	1,049,601
j	1,048,578	260,437	1,048,578	1,048,578	1,048,578
jr	1,047,552	0	0	0	0
jrl	1,047,552	0	0	0	0
cas	0	0	1,047,552	1,047,552	1,047,552
NonNop Instructions	32,964,592	24,843,578	12,581,905	12,581,905	12,581,905
Cycles	71,447,758	58,088,982	29,357,091	34,594,851	49,260,579
IPC	0.46138	0.42768	0.42858	0.36369	0.25542

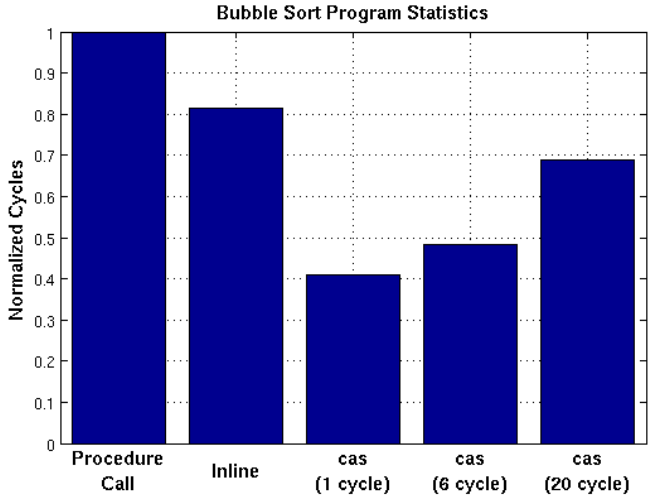

Figure 25: Bubble sort statistics graph.

The statistics Neurosim reports on the performance improvements match what one might expect by looking at the logic that was replaced with the CAS dedicated instruction. The

greatest decrease in the number of instructions is in memory operations, "lw" and "sw." Recall that the test assumed a zero cycle cache miss penalty. In reality the performance improvements of the CAS instruction could be much larger when cache miss penalties are introduced.

15.2 Bubble Sort vs Merge Sort

Neurosim can not only be used to evaluate hardware design decision but can also be used to compare different implementations of an algorithm. This has already been demonstrated in the context of loop unrolling and recursive functions, but a comparison of sorting algorithms provides a more complete example.

Often, sorting algorithms are analyzed by their run times and memory usage. For example, bubble sort is an $O(n^2)$ algorithm. This means that the runtime scales quadratically with the number of inputs. Another sorting algorithm, merge sort, has a runtime of $O(n * log(n))$ which scales much more favorably with array size. An implementation of merge sort is presented in Code 25. This is a fairly complex program and shows the power of NEUROSim. Merge sort works by dividing the array into two subarrays. The subarrays are then divided again and this is continued recursively until there are only two elements in each subarray. These two element arrays are sorted and then the subarrays are repeatedly "merged" back together resulting in a full sorted array. It should be noted that this algorithm requires an extra array to be used as working space, doubling the space needs of that of bubble sort.

Table 37 and Figures 26–28 illustrate the results of running the two algorithms on arrays sized from 2 to 1024. Some interesting conclusions can be drawn from this data. The two sorting algorithms follow their predicted curves and clearly merge sort is superior for sorting arrays larger than 8 elements. Bubble sort wins on small arrays because it has less overhead than mergesort. These results provide a way to optimize a general sorting algorithm by combining mergesort and bubble sort. Essentially, merge sort should only subdivide arrays until they are eight elements in size. Then bubble sort should be run on these arrays and

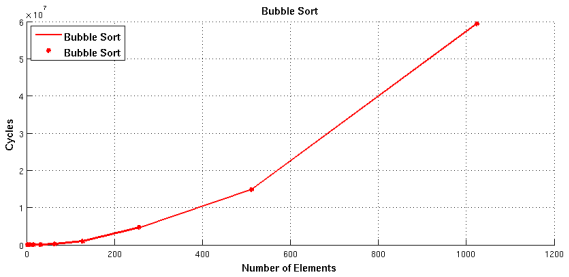

Figure 26: Bubble sort run time.

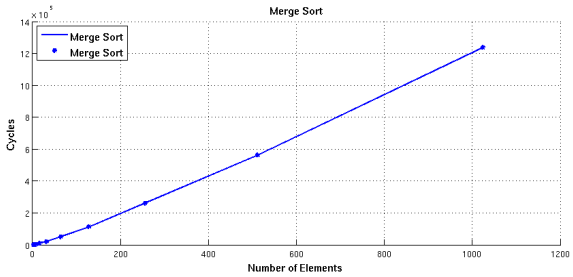

Figure 27: Merge sort run time.

merge sort used to merge the arrays back together.

Table 37: Bubble Sort vs Merge Sort statistics.

Number of Elements	**Bubble Sort Cycles**	**Merge Sort Cycles**
2	215	404
4	768	1,338
8	3,237	3,586
16	13,965	9,010
32	56,869	21,410
64	222,105	49,900
128	942,787	113,510
256	4,670,708	261,254
512	14,876,572	56,3758
1024	59,367,696	1,237,892

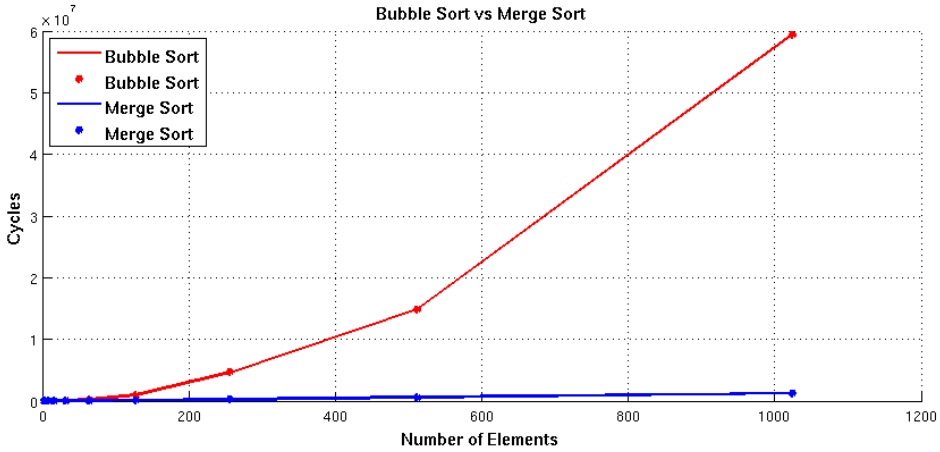

Figure 28: Bubble Sort vs Merge Sort.

The use of a simulator to analyze these sorting algorithms has the key advantage of using

cycle count information opposed to timing information. This is in general a much more accurate and repeatable method of comparison. For example, determining the threshold at which to use merge sort over bubble sort would be very difficult if only timing information was available. The ability to directly configure the hardware also allows the user to experiment with how hardware changes such as cache configuration or branch predictor algorithm impact the details of the algorithms general runtime curve. For example, maybe due to the linear nature of accesses in the bubble sort algorithm it incurs less penalties due to cache misses and actually performs better on array sizes much larger than eight.

16 Conclusion

NEUROSim provides a framework similar to that of a traditional computer architecture simulator allowing for a tight feedback loop in the evaluation of design. However, NEUROSim is also intended to be used as a systems design tool because it provides a straightforward method by which hardware and software components of a system can be analyzed. Numerous examples of how performance can be increased using various design decisions have been examined including execution forwarding, control change detection, FPU configuration, loop unrolling, recursive functions, and self modifying code. It was also demonstrated how two key components of a datapath, branch predictors and cache architectures, can be configured and evaluated. NEUROSim demonstrated its true differentiation from classic simulators as five instructions ("mod", "mac", "crv", "exp", and "cas") were easily added to supplement the core ISA. However, this is by no means an exhaustive list of the domains or problems to which NEUROSim can be applied. NEUROSim aims to be a design tool in any domain which involves optimization or specialization which crosses the hardware, software divide.

17 Future Work

The NEUROSim framework is intended to be constantly improved and extended as unique design decisions or algorithms arise providing for the possibility of nearly endless future work and use. However, there are a few areas in particular which need improvement.

NEUROSim is currently focused on hardware design decisions. However, there are also incredible benefits to tailoring an assembler and compiler to a target domain. Axon began to do this. For instance, the compiler will automatically recognize when a "mac" instruction can be used to gain performance benefits. Support for recognizing arbitrary instructions needs to be added. Axon also needs to provide full support for floating point data. Currently, some of the resulting assembly from the "Mathematical Expression Approximation" section must be curated. It would also be of incredible benefit to develop an implementation of the datapath Neurosim simulates on an FPGA (field programmable gata array) board. This would expand the scope of NEUROSim to right before an ASIC (application specific integrated circuit) is fabricated because a typical hardware design process involves simulation, FPGA implementation, and then ASIC design and fabrication. On the software side, it would be of value to develop a light weight real time operating system (RTOS) to run on top of the simulator. This would provide the designer of a new embedded system a starting point.

LIST OF REFERENCES

[1] "The architecture of open source applications." `http://aosabook.org/en/index.html`, 2011.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 4th ed., 2008.

[3] M. Nielsen, *Neural Networks and Deep Learning*. 2016.

[4] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, pp. 13–25, June 1997.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[6] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: A full system simulator for multicore x86 cpus," in *Proceedings of the 48th Design Automation Conference*, DAC '11, (New York, NY, USA), pp. 1050–1055, ACM, 2011.

[7] "The llvm compiler infrastructure." `http://llvm.org/`.

[8] "clang: a c language family frontend for llvm." `http://clang.llvm.org/`.

[9] "flex: The fast lexical analyzer." `http://flex.sourceforge.net/`.

[10] "Gnu bison." `https://www.gnu.org/software/bison/`.

[11] H.-Y. Kim, J. Rosser, K. Bryson, and S. Majumder, "Partitioning register file to reduce access time." http://www.owlnet.rice.edu/~elec525/projects/prf_report.pdf.

[12] D. Bhandarkar, "Risc versus cisc: A tale of two chips," *SIGARCH Comput. Archit. News*, vol. 25, pp. 1–12, Mar. 1997.

[13] M. A. Will and R. K. L. Ko, "Computing mod without mod," *IACR Cryptology ePrint Archive*, vol. 2014, p. 755, 2014.

[14] T. Kocak, "Sigmoid functions and their usage in artificial neural networks." https://excel.ucf.edu/classes/2007/Spring/appsII/Chapter1.pdf.

APPENDIX

Code 1: Axon supported syntax.

```
/* Macro. */
#define LENGTH_X (10)

/* Global array and variable. */
int X[LENGTH_X] = {1, 2, 3, 4, 5, 5, 4, 3, 2, 1};
int G = 12;

/* Enumeration type. */
typedef enum Test {
  Test_a,
  Test_b,
  Test_c,
} Test_t;

/* Function declaration. */
int f(int a, int b) {
  return a + b;
}

int main() {
  /* Integer variables. */
  int x = 32;
  int y = 8;
  int z = 5;
  int i = 0;
  /* Pointer to global variables. */
  int* g = &G;
  Test_t t = Test_a;
  /* Arithmetic and logic operators. */
  z = x + y;
  z = x - y;
  z = x * y;
  z = x / y;
  z = x | y;
  z = x & y;
  z = x ^ y;
  /* Compound expressions. */
  z = z * (x + 10) - y;
  /* While loop. */
  while (i < z) {
    i++;
  }
  /* For loop. */
  for (i = 0; i < (LENGTH_X - 2); i++) {
    /* Function call and array indexing. */
    z = z + f(X[i], X[i + 1]);
  }
  /* If else statement. */
  if (z > 100) {
    z = z * 2;
  } else {
    z = z + 2;
  }
  /* Switch statement. */
  switch(t) {
```

```
    case Test_a:
      z = z + 21;
      break;
    case Test_b:
      z = f(z, x);
      break;
    case Test_c:
    default:
      break;
  }
  /* Dereferencing global pointers. */
  z = *g + z;
  return z; // Z = 3487
}
```

Code 2: Addition in C.

```
int main() {
  int x = 64;
  int y = 27;
  int z = x + y;
  return z;
}
```

Code 3: Addition in assembly.

```
  .text
  j main:
main:
  # Allocate 3 words on stack
  addi sp sp -12
  loi  t0 64      # x
  sw   t0 sp 8
  loi  t0 27      # y
  sw   t0 sp 4
  lw   t1 sp 8    # Reload x
  add  v0 t1 t0   # z in return register
  sw   v0 sp 0
  addi sp sp 12   # Restore stack
  halt
```

Code 4: For loop in C.

```c
int main() {
  int i ;
  for(i = 0; i < 10; i++);
  return i;
}
```

Code 5: For loop in assembly.

```
  .text
  j main:
main:
  # Allocate 1 word on stack
  addi sp sp -4
  loi  t0 0     # i
  sw   t0 sp 0
  loi  t0 9     # Loop upper bound
  j    LBB0_1:
LBB0_2:
  lw   t1 sp 0  # Load i
  addi t1 t1 1  # Increment i
  sw   t1 sp 0  # Save i
LBB0_1:
  lw   t1 sp 0  # Load i
  slt  at t0 t1 # Check loop condition
  bne  at r0 LBB0_3: # Break out of loop
  j    LBB0_2:  # Continue loop
LBB0_3:
  lw   v0 sp 0  # Load i as return value
  addi sp sp 4  # Restore stack
  halt
```

Code 6: Function call in C.

```c
int X[2] = {42,-89};

int f(int x, int y) {
  return x + y;
}

int main() {
  int r = f(X[0], X[1]);
  return r;
}
```

Code 7: Function call in assembly.

```
    .text
    j main:
f:
  # Allocate 2 words on stack
    addi sp sp -8
    sw   a0 sp 4
    sw   a1 sp 0
    lw   t0 sp 4
    add  v0 t0 a1
    addi sp sp 8 # Restore stack
    jr   ra
main:
    # Allocate 3 words on stack
    addi sp sp -12
    sw   ra sp 8  # Save return address
    loi  t0 0
    sw   t0 sp 4
    loi  t0 X:
    hii  t0 X:
    lw   a0 t0 0  # Load X[0]
    addi t0 t0 4
    lw   a1 t0 0  # Load X[1]
    loi  t0 f:
    hii  t0 f:
    jrl  t0       # Call f
    sw   v0 sp 0
    lw   ra sp 8  # Load return address
    addi sp sp 12 # Restore stack
    halt

    .data
X:
    .datum  42
    .datum  4294967207
```

Code 8: Execute forwarding example.

```
  .text
main:
  loi t0 17
  loi t1 31
  add t2 t1 t0
  sub t3 t2 t1
  halt
```

Code 9: Control change detection example.

```
int main() {
  int i;
  for (i = 0; i < 100; i++);
  return 0;
}
```

Code 10: FPU configuration example.

```
  .text
main:
  lof  ft0 0x1eb8
  hif  ft0 0x41db # 27.39
  lof  ft1 0x0000
  hif  ft1 0xc18c # -17.5
  addf fs0 ft0 ft1
  subf fs1 ft0 ft1
  mulf fs2 ft0 ft1
  divf fs3 ft0 ft1
  sltf fs4 ft0 ft1
  halt
```

Code 11: Loop unrolling.

```
#define LENGTH_X (1000)
#define LOOP_1  (1)
#define LOOP_5  (5)
#define LOOP_10 (10)
#define LOOP    LOOP_10

int X[LENGTH_X] = { -36,-12,2,-84,-16,95,75,90,-92,-25,
```

⋮

```
-90,65,-59,52,-44,1,27,15,-72,-13,51,43,76,20 };

int main() {
  int i = 0;
  int x = 0;
  for (i = 0; i < LENGTH_X; i = i + LOOP) {
    x = x + X[i];
#if (LOOP >= LOOP_5)
    x = x + X[i + 1];
    x = x + X[i + 2];
    x = x + X[i + 3];
    x = x + X[i + 4];
#if (LOOP >= LOOP_10)
    x = x + X[i + 5];
    x = x + X[i + 6];
    x = x + X[i + 7];
    x = x + X[i + 8];
    x = x + X[i + 9];
#endif
#endif
  }
  return x;
}
```

Code 12: Recursive implementation of factorial.

```
/* Recursive function to compute n! */
int factorial(int n) {
  if (n < 1) {
    return 1;
  } else {
    return factorial(n - 1) * n;
  }
}

int main() {
  int x = factorial(7);
  int y = factorial(12);
  return x + y;
}
```

Code 13: Loop implementation of factorial.

```c
/* Loop implementation of n! */
int factorial(int n) {
  int i;
  int fact = 1;
  for (i = 1; i <= n; i++) {
    fact = fact * i;
  }
  return fact;
}

int main() {
  int x = factorial(7);
  int y = factorial(12);
  return x + y;
}
```

Code 14: Self modifying code.

```
  .text
  j main:

main:
  # The registers to perform operations on
  loi t0 786
  loi t1 512
  # Get the starting address of the
  # instructions held in data
  loi t2 INSTRUCTIONS:
  # Get instruction address to change
  loi t3 InstructionToChange:
  loi t4 END:
  lw  t4 t4 0
Loop:
  lw t5 t2 0
  sw t5 t3 0
  nop # Delay till the memory is saved
  nop
InstructionToChange:
  nop # Instruction that gets modified
  addi t2 t2 4 # Address of next instruction
  bne t5 t4 Loop: # Check if we have reached the end
  halt

  .data
INSTRUCTIONS:
  .datum 0x5091000 # add v0 t0 t1
  .datum 0x9091000 # sub v0 t0 t1
  .datum 0xD091000 # mul v0 t0 t1
END:
  .datum 0x11091000 # div v0 t0 t1
```

Code 15: Branch prediction test.

```
#define LENGTH_X (1000)

int X[LENGTH_X] = {
36,12,2,84,16,95,75,90,92,25,99,74,0,16,4,40,
```

⋮

```
90,65,59,52,44,1,27,15,72,13,51,43,76,20 };

int main() {
  int x = 0;
  int two = 2;
  for (int i = 0; i < LENGTH_X; i++) {
    x = X[i];
         if (x < 10)  { }
    else if (x < 15)  { }
    else if (x < 20)  { }
    else if (x < 25)  { }
    else if (x < 30)  { }
    else if (x < 35)  { }
    else if (x < 40)  { }
    else if (x < 45)  { }
    else if (x < 50)  { }
    else if (x < 55)  { }
    else if (x < 60)  { }
    else if (x < 65)  { }
    else if (x < 70)  { }
    else if (x < 75)  { }
    else if (x < 80)  { }
    else if (x < 85)  { }
    else if (x < 90)  { }
    else if (x < 95)  { }
    else if (x < 100) { }
    while (x != 0) { x = x / two; }
  }
  return 0;
}
```

Code 16: Random cache access.

```
#define LENGTH_I (1024)
#define LENGTH_X (2048)

/* The block of memory on which to test the cache */
int X[LENGTH_X] = {1};
/* The random indexes into X */
int I[LENGTH_I] = { 14,1463,694,106,361,1719,181,2034,
```

⋮

```
1290,1247,110,1776,1911,1490,1332,1022,1328,802 };

int main() {
  int i = 0;
  int x = 0;
  for(i = 0; i < LENGTH_I; i++) {
    x = X[I[i]];
  }
  return i;
}
```

Code 17: Few random cache access.

```
#define LENGTH_I (1024)
#define LENGTH_X (2048)

/* The block of memory on which to test the cache */
int X[LENGTH_X] = {1};
/* 128 indexes randomly placed. */
int I[LENGTH_I] = { 1932,1333,1177,87,872,1016,1743,
```

⋮

```
1863,1490,927,1529,549,622,692,199,1288,1579 };

int main() {
  int i = 0;
  int j = 0;
  int x = 0;
  for(i = 0; i < LENGTH_I; i = i + 2) {
    x = X[I[i]];
  }
  return i;
}
```

Code 18: Linear cache access.

```
#define LENGTH_I (1024)
#define LENGTH_X (2048)

/* The block of memory on which to test the cache */
int X[LENGTH_X] = {1};
/* The indexes and linear ranges into X */
int I[LENGTH_I] = { 506,51,1530,100,1362,22,1365,76,
```

⋮

```
int main() {
  int i = 0;
  int j = 0;
  int x = 0;
  for(i = 0; i < LENGTH_I; i = i + 2) {
    for (j = 0; j < I[i + 1]; j++) {
      x = x + X[I[i] + j];
    }
  }
  return i;
}
```

Code 19: Inlined function example.

```
#define INLINE __attribute__((always_inline)) inline

int add(int x, int y) {
  return x + y;
}

INLINE int addInlined(int x, int y) {
  return x + y;
}

int main() {
  int x = 17;
  int y = 31;
  int z = add(x, y);
  int z = addInlined(x, y);
  return z;
}
```

Code 20: Modulo test program

```
#define INLINE __attribute__((always_inline)) inline
#define LENGTH_X (200)

int X[LENGTH_X] = {
442,144,943,14,943,326,833,386,546,652,319,516,783,13,
```

⋮

```
373,837,950,654,567,588,455};

/* Compute r = x % y */
INLINE int mod(int x, int y) {
    int r = x;
    while(r >= y) {
      r = r - y;
    }
    return r;
}

int main() {
  int i;
  int k;
  int r;
  /* Compute the modulo between each pair of numbers in X */
  for (i = 0; i < LENGTH_X; i++) {
    for (k = 0; k < LENGTH_X; k++) {
      r = mod(X[i], X[k]);
    }
  }
  return 0;
}
```

Code 21: Mac Instruction

```
#define INLINE __attribute__((always_inline)) inline
#define SCALING_FACTOR    (32768)
#define LENGTH_X          (2048)
#define ORDER             (6)
#define NUM_COEFFICIENTS (ORDER + 1)

/* The input array */
int X[LENGTH_X] = {144,544,1312,1600,1120,576,256,0,
```

⋮

```
-175,-113,-64,-32,32,81,95,160,224,256,321,351,368 };
/* The output array */
int Y[LENGTH_X] = {1};
/* Reverse coefficients */
int A[NUM_COEFFICIENTS] = {32768,-148924,313916,-383222,
                284649,-121930,23838};
/* Forward coefficients */
int B[NUM_COEFFICIENTS] = {615,-1570,2811,-3096,2811,-1570,615};

/* Multiply accumulate */
INLINE int mac(int a, int b, int c) {
    return a + (b * c);
}

int main() {
  int n = 0; /* Index into X */
  int k = 0; /* Index into A or B */
  int sumA = 0;
  int sumB = 0;
  /* Fill the delay lines */
  for (n = 0; n < ORDER; n++) {
    Y[n] = 0;
  }
  for (n = ORDER; n < LENGTH_X; n++) {
    sumA = 0;
    sumB = 0;
    for (k = 0; k < NUM_COEFFICIENTS; k++) {
      sumA = mac(sumA, -A[k], Y[n - k]);
      sumB = mac(sumB,  B[k], X[n - k]);
    }
    /* Scale Y[n] back down by one scaling factor */
    Y[n] = (sumA + sumB) / SCALING_FACTOR;
  }
  return 0;
}
```

Code 22: Lookup table exponential approximation.

```
#define LOOKUP_LENGTH (8u)

/* Hidden neurons 0 and 1 [weight1, weight2, bias]. */
float H0[3] = {1.0, 1.0, -0.5};
float H1[3] = {1.0, 1.0, -1.5};
/* Output neuron [weight1, weight2, bias]. */
float O[3] =  {1.0, -1.0, -0.2};
/* x values of exponential lookup table. */
float X[LOOKUP_LENGTH] = { -1.5, -1.07142, -0.64285, -0.21428,
                           0.214285, 0.642857, 1.071428, 1.500000 };
/* y values of exponential lookup table. */
float Y[LOOKUP_LENGTH] = { 0.223130, 0.342518, 0.525788, 0.807117,
                1.238976, 1.901907, 2.919547, 4.481689 };

/* Lookup table aproximation of exponential. */
float expLookupTable(float x) {
  int i;
  for ( i = 0; i < LOOKUP_LENGTH; i++) {
      if (x <= X[i]) {
          break;
      }
  }
  i--;
  if (i == 0 || i == (LOOKUP_LENGTH - 1)) {
    return Y[i];
  } else {
    return Y[i] + (x - X[i]) * ((Y[i + 1] - Y[i]) / (X[i + 1] - X[i]));
  }
}

/* The sigmoid activation function. */
float activation(float x) {
  return 1 / (1 + expLookupTable(-x));
}

/* Forward propagation of the inputs through the network. */
float forwardPropagation(float In0, float In1) {
  float aH0 = activation((In0 * H0[0]) + (In1 * H0[1]) + H0[2]);
  float aH1 = activation((In0 * H1[0]) + (In1 * H1[1]) + H1[2]);
  float aOutput = activation((aH0 * O[0]) + (aH1 * O[1]) + O[2]);
  return aOutput;
}

int main() {
  forwardPropagation(0, 0);
  forwardPropagation(1.0, 0);
  forwardPropagation(0, 1.0);
  forwardPropagation(1.0, 1.0);
  return 0;
}
```

Code 23: Taylor series exponential approximation.

```
/* Hidden neurons 0 and 1 [weight1, weight2, bias]. */
float H0[3] = {1.0, 1.0, -0.5};
float H1[3] = {1.0, 1.0, -1.5};
/* Output neuron [weight1, weight2, bias]. */
float O[3] =  {1.0, -1.0, -0.2};
/* Precomputed factorial values. */
float FACTORIAL[11] = {1, 1, 2, 6, 24, 120, 720,
           5040, 40320, 362880, 3628800};

/* Compute x^n */
float power(float x, int n) {
  int i;
  float result = x;
  if (n == 0) {
    return 1;
  }
  for (i = 1; i < n; i++) {
    result = result * x;
  }
  return result;
}

/* Taylor series aproximation of exponential. */
float expTaylorSeries(float x) {
  int i;
  float result = 0;
  for (i = 0; i <= 10; i++) {
      result = result + power(x, i) / FACTORIAL[i];
  }
  return result;
}

/* The sigmoid activation function. */
float activation(float x) {
  return 1 / (1 + expTaylorSeries(-x));
}

/* Forward propagation of the inputs through the network. */
float forwardPropagation(float In0, float In1) {
  float aH0 = activation((In0 * H0[0]) + (In1 * H0[1]) + H0[2]);
  float aH1 = activation((In0 * H1[0]) + (In1 * H1[1]) + H1[2]);
  float aOutput = activation((aH0 * O[0]) + (aH1 * O[1]) + O[2]);
  return aOutput;
}

int main() {
  forwardPropagation(0, 0);
  forwardPropagation(1.0, 0);
  forwardPropagation(0, 1.0);
  forwardPropagation(1.0, 1.0);
  return 0;
}
```

Code 24: Bubble sort.

```
#define INLINE __attribute__((always_inline)) inline
#define LENGTH_X (1024)

/* The array to sort. */
int X[LENGTH_X] = { 4219,2029,3501,7547,6025,1505,9131,
```

⋮

```
3973,1869,7253,4063,8731,8147,8018,9794,8569,4502,4315 };

/* Compare and swap */
INLINE void cas(int idx1, int idx2) {
    int temp;
    if (X[idx1] > X[idx2]) {
        temp = X[idx2];
        X[idx2] = X[idx1];
        X[idx1] = temp;
    }
}

int main() {
    int i;
    int j;
    /* Bubble sort X */
    for (i = 0; i < LENGTH_X; i++) {
        for (j = 0; j < (LENGTH_X - 1); j++) {
            cas(j, j + 1);
        }
    }
    return 0;
}
```

Code 25: Merge sort.

```
#define LENGTH_X (128)

/* The array to sort. */
int X[LENGTH_X] = {4219,2029,3501,7547,6025,1505,9131,
```

⋮

```
3973,1869,7253,4063,8731,8147,8018,9794,8569,4502,4315 };

/* The working area used when sorting X. */
int Y[LENGTH_X] = {1};

void merge(int low, int mid, int high) {
    int k;
    int l = low;
    int i = low;
    int m = mid + 1;
    /* Copy elements from the two subarrays to the
       working area sorting the elements as we go. */
    while (1 /*(l <= mid) && (m <= high)*/) {
```

```
45          if (m > high) break;
46          if (l > mid) break;
47
48          if (X[l] <= X[m]) {
49              Y[i] = X[l];
50              l++;
51          }
52          else {
53             Y[i] = X[m];
54             m++;
55          }
56          i++;
57      }
58      /* Copy potential remaining elements from a subarray. */
59      if (l <= mid) {
60          for (k = l; k <= mid; k++) {
61             Y[i] = X[k];
62             i++;
63          }
64      }
65      else { /* m <= high */
66          for (k = m; k <= high; k++) {
67             Y[i] = X[k];
68             i++;
69          }
70      }
71      /* Copy results from Y back to X. */
72      for (k = low; k <= high; k++) {
73          X[k] = Y[k];
74      }
75  }
76
77  void mergeSort(int low,int high) {
78      int mid;
79      int two = 2;
80      if (low < high) {
81          mid = (low + high) / two;
82          /* Recursivly sort the two subarrays. */
83          mergeSort(low, mid);
84          mergeSort(mid + 1, high);
85          /* Merge the resulting sorted subarrays. */
86          merge(low, mid, high);
87      }
88  }
89
90  int main() {
91      mergeSort(0, LENGTH_X-1);
92      return 0;
93  }
```