UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUSTAVO GARCIA VALDEZ

A Generic Description and Simulation of Architectures Based on Microarchitectures

Monograph presented in partial fulfillment
of the requirements for the degree of
Bachelor of Computer Science

Prof. Dr. Raul Fernando Weber
Advisor

Porto Alegre, December 5th, 2013

CIP – CATALOGING-IN-PUBLICATION

Gustavo Garcia Valdez,

A Generic Description and Simulation of Architectures Based on Microarchitectures /

Gustavo Garcia Valdez. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2013.

64 f.: il.

Monograph – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2013. Advisor: Raul Fernando Weber.

1. Computer Architectures. 2. Machine Simulation. 3. Microarchitectures. 4. Computer Organization. I. Weber, Raul Fernando. II. Título.

Instruction tables will have to be made up by mathematicians with computing experience and perhaps a certain puzzle-solving ability. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself. (TURING, 1946)

ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor Prof. Raul Fernando Weber for the support in my bachelor thesis, for everything that I learned from him in many classes during the bachelor and for envisioning the system that I would them use as conceptual basis for my project idea.

Besides my advisor, I would like to thank all my Professors and TA's throughout my 6 years of study, both at Federal University of Rio Grande do Sul (UFRGS) as at the Berlin Institute of Technology (TU-Berlin).

My sincere thanks also to Prof. Aline Villavicencio, who advised me during almost my whole graduation when I was working in the field of Computer Linguistics.

Last but not least, I want to thank my parents Heraldo da Silva Valdez and Isabel Cristina Garcia Valdez for all the effort and guidance that I so much needed in those 23 years of life.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ALU Arithmetic Logic Unit. 33, 34, 36, 37, 39, 41, 45, 49

CPU Central Processing Unit. 23, 28

DLL Dynamic-Link Library. 40

DSO Dynamic Shared Object. 40

GUI Graphical User Interface. 25, 33, 34, 39, 40, 42, 43, 47, 50, 53

I/O Input/Output. 34, 36

ISS Instruction Set Simulator. 27–29, 34, 53

JVM Java Virtual Machine. 23

OO Object Oriented. 23, 33, 39, 40

PC Program Counter. 23, 26, 27

ABSTRACT

In teaching computer architecture, didactic simulator machines have been the rule to teach basic aspects of assembly. This work searches for another kind of simulator to teach the microcoding/microarchitectures and, once found, conceptualizes and implements one of those. The work documents the steps and problems encountered while achieving that objective. It begins by studying which kinds of simulators and emulators there are available today, then proceeds to presenting a conceptualization of a framework that allows for creation of simulators using microinstruction listings. It then shows the implementation choices, the implementation and concludes by showing which lessons were taken from the whole enterprise.

Keywords: Computer Architectures, Machine Simulation, Microarchitectures, Computer Organization.

Uma Descrição e Simulação Genérica de Arquiteturas Baseada em Microarquiteturas

RESUMO

No ensino de arquiteturas de computadores, simuladores didáticos têm sido a regra em ensinar os aspectos básicos de programação de máquina. Esse trabalho busca um outro tipo de simulador para o ensino de microcódigo/microarquiteturas e, quando o encontra, conceptualiza e implementa um destes. O trabalho documenta os passos e problemas encontrados durante a conquista desse objetivo. Começa estudando quais tipos de simuladores e emuladores estão disponíveis hoje, então apresenta uma conceptualização de um framework que permite a crição de simuladores usando listagens de microinstruções. Mostra as escolhas de implementação, a própria implementação e conclui apresentando quais lições foram aprendidas durante esta empreitada.

Palavras-chave: Arquiteturas de Computadores, Simulação de Máquinas, Microarquiteturas, Organização de Computadores.

1 INTRODUCTION

In this chapter, motivation and goals for the present bachelor thesis will be presented. A brief description of the work's structure is also shown.

1.1 Motivation

Teaching computer architectures is a very interesting endeavour and making the learning more dynamic helps the students a lot in retaining contents. Prof. Weber's didactic machines (WEBER, 2000) helped me a lot and many of my colleagues learning machine language, but I've also noted some of them had problems grasping the concepts of microarchitectures. Besides that, a lot of 'less useful' didactic machines from this set were not converted to Windows 32, since it would be some effort to convert them all.

Connecting those two, came the idea to create a framework for designing almost any machine that, at the same time, could teach students microarchitectures.

1.2 Goals

This bachelor thesis intends to achieve two basic goals by conceptualizing and implementing a a basic framework for simulating architectures, based on microinstructions.

The first goal is for this framework to be usable in classes, where the professor prepares some (more complex and/or outside the classes scope) configurations, but leaves for the student to make other configurations in order to learn specific topics in architecture and organization by doing instead of being lectured. As an example, section 3.3 shows a case study for teaching how different addressing modes work by implementing them on the system.

The second goal is to allow a professor to define a wide range of machines, in order to teach basics of assembly for the students in those machines. Maybe even configure a real processor in it, in order for the students to study it.

Some extra nice features would be that the student would not have to know any programming language, besides what is being learned to use it, since these topics are usually taught in very early introductory courses. Other desired feature would be for the professor to have powerful configuration tools, that allow him to code the widest possible range of machines.

1.3 Structure of this work

In chapter 2 (State-of-the-Art), many simulation and emulation systems that already exist are shown. Followed by chapter 3 (Conceptualization), which presents the conceptual idea for this work and a use case. Chapter 4 (The 'Tanuki' implementation) then presents a concrete implementation of this idea. After showing the implementation, chapter 5 (Description Languages) describes the description laguages for the input files the system uses. The work is then concluded in chapter 6 (Conclusions and Future Work).

2 STATE-OF-THE-ART

In this chapter, some simulation systems will be presented, with some of its main features, motivations and goals. Simulation / emulation of architectures is a vast area and in no way this list is intended to be extensive. The chapter's intent is only to show some of what can be achieved with those concepts and to put this work in perspective.

2.1 p-code Machines

A p-code machine is virtual machine that runs a instruction set for a hypothetical Central Processing Unit (CPU) that is run by multiple interpreters in different platforms. As examples, we could cite the Java Virtual Machine (JVM)'s p-code, Pascal-P's p-machine, MATLAB's precompiled code and many others.

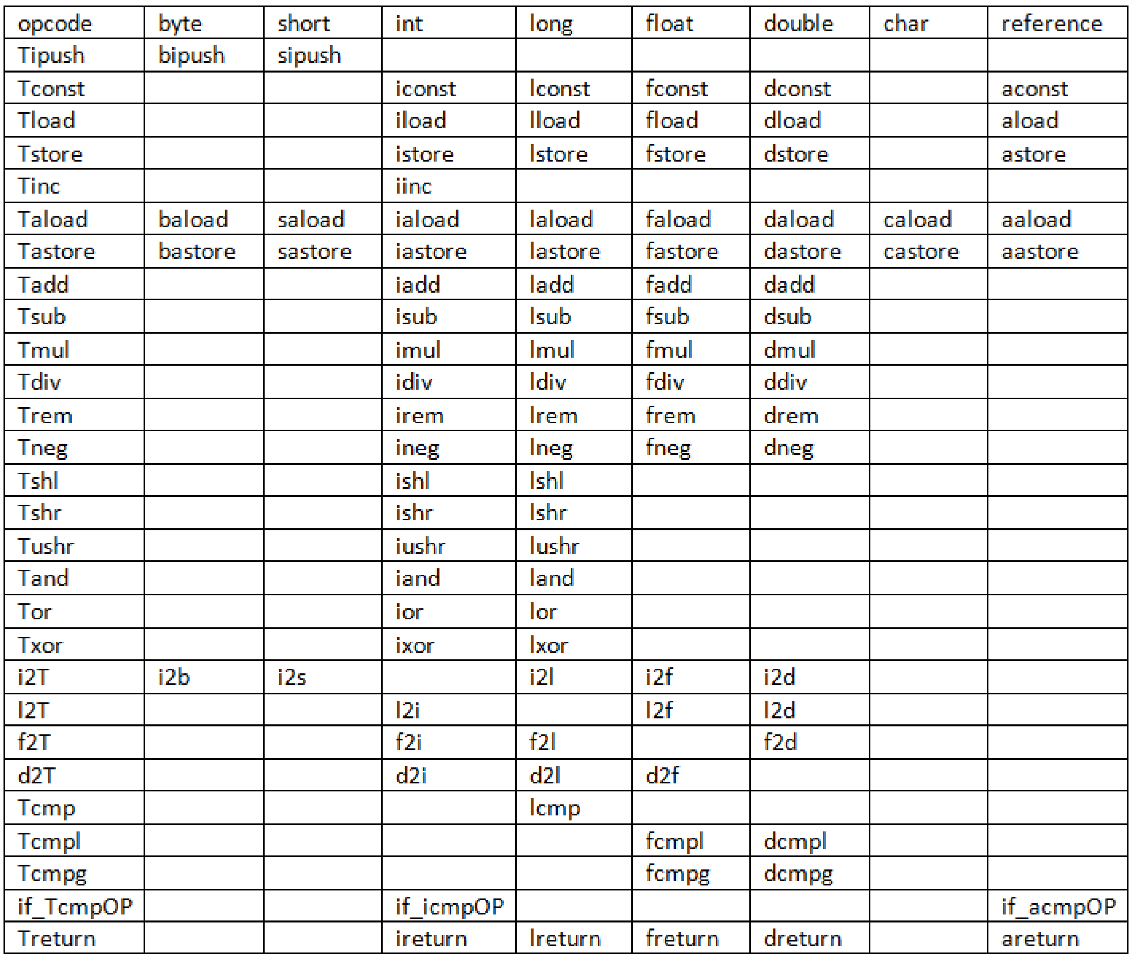
The term was p-code was first used in implementation notes for Pascal-P (AMMAN et al., 1974). The main motivation to use p-code instead of compiling directly to each machine is that you can "compile once and run anywhere". Thus, the goals of a p-code architecture is to be easily implementable in a big range of machines to achieve portability.

2.1.1 JVM Architecture

To better understand p-code, the JVM architecture will be taken as an example. Its instructions are similar to what you would expect from any architecture, whilst also keeping some features quite useful for an Object Oriented (OO) language like Java. That is done intentionally, so that programmers are able to write efficient implementations for every processor and, at the same time, making the compiler's work a bit simpler.

There are 8 data types and many operation codes, but not every operation code operates every data type (See examples in table 2.1). The instructions can be divided in 9 main types: 'Load and Store', 'Arithmetic', 'Type Conversion', 'Object Creation and Manipulation', 'Stack Management Instructions', 'Control Transfer Instructions' (jump instructions), 'Method Invocation and Return Instructions', 'Throwing Exceptions' and 'Synchronization'.

The instruction cycle follows the algorithm presented in Figure 2.1, which is: calculate Program Counter (PC) and fetch next instruction atomically; fetch any operands if needed; and execute code for the instruction.



opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

Table 2.1: Type support in the Java Virtual Machine instruction set
Table 2.2 from chapter 2.11 of the specification (LINDHOLM et al., 2013)

```
do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

Figure 2.1: JVM instruction cycle pseudo-code description
Description of the instruction cycle as in chapter 2.11 of the specification (LINDHOLM et al., 2013)

2.2 Historical and Legacy Code Emulators

Emulation nowadays is a very common way to execute legacy code that can't run on a given software/hardware platform. Those emulators can work in many ways, from simulating the whole hardware circuits, to simulating at organization or architectural levels, or even using clues from how the code was made to make 'tweaks' in order to get performance improvements. Those give different grades of fidelity and, inversely, performance.

2.2.1 The SIMH Simulator set

One important example of a historical simulator is SIMH, created and maintained by the The Computer History Simulation Project (BURNET; SUPNIK, 1996), which is a collective of people that are interested in restoring by simulation computers that are historically significant. With that goal, they decided to create and freely publish a simulator for those systems, alongside with their most important software.

As of the writing of this text, it supports 29 machines (not including beta versions). Each one is its own executable program, but they all follow the same rules, documentation and use the same command-line commands. They have no Graphical User Interface (GUI), but are very well documented and with some effort can be easily used. For most machines there's even specific documentation. A key point is that, being able to have I/O with files in the host machine, it's possible to connect physical devices to the simulators (even easier in *NIX systems).

2.3 Gaming Console Platform Emulators

From ZSNES (ZSNES, 2001) to Dolphin (DOLPHIN, 2013), there are many systems that intend to emulate the experience from a given console platform.

Their basic goal is to provide performance and experience as good (or even better) than the original platform. This is done by emphasizing on performance and experience much more than an emulation of the same instructions run in the same order with the same outcome, sometimes using tricks to improve computing speed in order to run in a speed that is consistent with playing the games. At some cases, they even use the host's computational power to have improved graphics compared to what is being emulated, usually by running at higher resolution or applying image filters on the final screens.

In structure, they are very similar to legacy code emulators, but since the systems are closed and almost no documentation is available, some documentation has to be built from scratch by reverse engineering and trial-and-error methods before even starting the emulators construction. Sometimes, legal issues make it impossible to distribute the console's BIOS with the emulator, which is solved by either rewriting it or asking for the user to take it from his physical video game system.

Since it's usually very hard to code the peripherals (video, sound, input), they're usually programmed as plugins, as to allow for improvement and customization of those systems directly.

2.4 Didactic Machine Simulators

A didactic machine is a computer architecture designed to teach students some basic concepts of computer architectures. It can also embed what a professor thinks would have been the correct way for computers to evolve in a pure academical view, without the interference of market and economic forces.

In a way, the didactic machines created by Professor Raul Fernando Weber (WEBER, 2000) do exactly that, each one introducing a new concept for the student and, at the same time, projecting the author's view of how computers should have evolved. That means students can experience decades of computer history in a controlled environment and learn by programming machine level code. Most of the machine names are of historical marks, in order to give a sense of progress, starting with an 8-bit very simple Neander and ending in a much more complex 16-bit 'Cesar'.

2.4.1 Neander Machine

Inspired by Von Neumanns IAS, it aims to teach the basics of binary code programming. Being so basic it has no practical uses, yet it's simplicity allows students to, in no time, program it (or even design it's architecture and organization in class).

Features: 8-bit-sized data and address, 1 register accessible to the programmer, 8-bit PC, 2 state registers (N and Z)

2.4.2 Ahmes Machine

Completely compatible with neander code, it adds some instructions to allow the execution of the four basic arithmetic operations, with addition and subtraction being instructions. Binary Multiplication and Binary Division being learned by the student by coding it as software, as it was done in the old age of computing.

Features: 8-bit-sized data and address, 1 register accessible to the programmer, 8-bit PC, 5 state registers (N, C, Z, B, V)

2.4.3 Ramses Machine

Ramses is a third architecture that adds even more features to the neander base. For example, there are many registers (instead of just one). This allows the student to work with more complex data structures like arrays, matrixes, lists and pointers at assembler level. This allows for 'more efficient' running, while still keeping compatibility with the Neander architecture.

Features: 8-bit-sized data and address, 2 general purpose registers, 1 index register, 8-bit PC, 4 addressing modes (Direct, Indirect, Imediate and Indexed), 3 state registers (N, C, Z)

2.4.4 Cesar Machine

Inspired by the PDP-11 family from Digital Equipments Corporation (DEC) from the 70's, this hypothetical machine has all the basic features of modern computers: Eight general purpose registers, eight address modes and a complex instruction set. It also has the ability to manage a Memory Stack for sub-routines and parameter (passing?). It also includes some basic I/O system.

Features: 16-bit-sized data and address, 6 general purpose registers, 1 stack register, 16-bit PC, 8 addressing modes, 4 state registers (N, C, Z, V)

2.4.5 Other Machines

There are also other machines in the set that are variations of the ones presented before: Cromag, Queops, Pitagoras, Pericles, REG and Volta.

Cromag

8-bit-sized data and address, 1 register accessible to the programmer, 8-bit PC, 2 addressing modes (Direct, Indirect), 3 state registers (N, Z, C).

Queops

8-bit-sized data and address, 1 register accessible to the programmer, 8-bit PC, 4 addressing modes (Direct, Indirect, Imediate and PC Relative), 3 state registers (N, Z, C).

Pitagoras

8-bit-sized data and address, 1 register accessible to the programmer, 8-bit PC, 3 state registers (N, Z, C).

Pericles

8-bit-sized data and 12-bit-sized addresses, 2 general purpose registers, 1 index register, 8-bit PC, 4 addressing modes (Direct, Indirect, Imediate and Indexed), 3 state registers (N, C, Z).

REG

8-bit-sized data, 64 general purpose registers, 8-bit PC, no explicit condition codes.

Volta

8-bit-sized data, no general purpose registers, 1 stack register, 8-bit PC, no explicit condition codes.

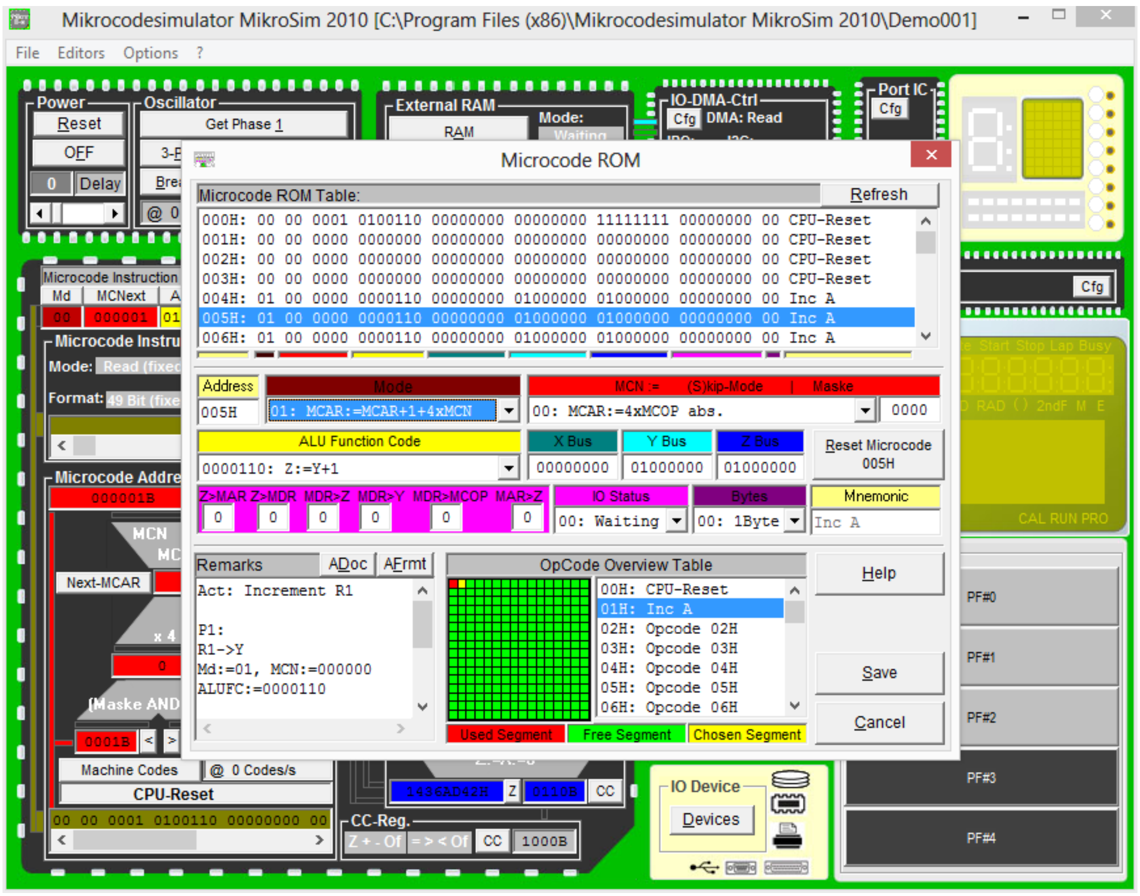
2.5 Instruction Set Simulators

An Instruction Set Simulator (ISS) is a computer model written in a high level programming language that interprets instructions and keeps registers and other internal information in variables. In this work, whenever the term ISS appears, it will refer to a subset of those that also have configurable instruction sets written in any sort of microcode language.

Two main examples would be MikroSim (CZERWINSKI; PUTTKAMER, 1979) and CPU Sim (SKRIEN, 2001), which are presented below.

2.5.1 MikroSim

MikroSim (CZERWINSKI; PUTTKAMER, 1979) is a paid closed-source ISS. It should be noted that some of the information comes from the attached documentation,



Microcode ROM Table:

```
000H: 00 00 0001 0100110 00000000 00000000 11111111 00000000 00 CPU-Reset
001H: 00 00 0000 0000000 00000000 00000000 00000000 00000000 00 CPU-Reset
002H: 00 00 0000 0000000 00000000 00000000 00000000 00000000 00 CPU-Reset
003H: 00 00 0000 0000000 00000000 00000000 00000000 00000000 00 CPU-Reset
004H: 01 00 0000 0000110 00000000 01000000 01000000 00000000 00 Inc A
005H: 01 00 0000 0000110 00000000 01000000 01000000 00000000 00 Inc A
006H: 01 00 0000 0000110 00000000 01000000 01000000 00000000 00 Inc A
```

Address	Mode	MCN := (S)kip-Mode | Maske	
005H	01: MCAR:=MCAR+1+4xMCN	00: MCAR:=4xMCOP abs.	0000

ALU Function Code	X Bus	Y Bus	Z Bus
0000110: Z:=Y+1	00000000	01000000	01000000

Z>MAR	Z>MDR	MDR>Z	MDR>Y	MDR>MCOP	MAR>Z	IO Status	Bytes	Mnemonic
0	0	0	0	0	0	00: Waiting	00: 1Byte	Inc A

Remarks

```
Act: Increment R1

P1:
R1->Y
Md:=01, MCN:=000000
ALUFC:=0000110
```

OpCode Overview Table

```
00H: CPU-Reset
01H: Inc A
02H: Opcode 02H
03H: Opcode 03H
04H: Opcode 04H
05H: Opcode 05H
06H: Opcode 06H
```

Figure 2.2: MikroSim Microcode ROM Editor
Allows for a graphic ROM coding, that is, defining the signals that activate each part of the processor

since the demonstration version does not allow for full functionality.

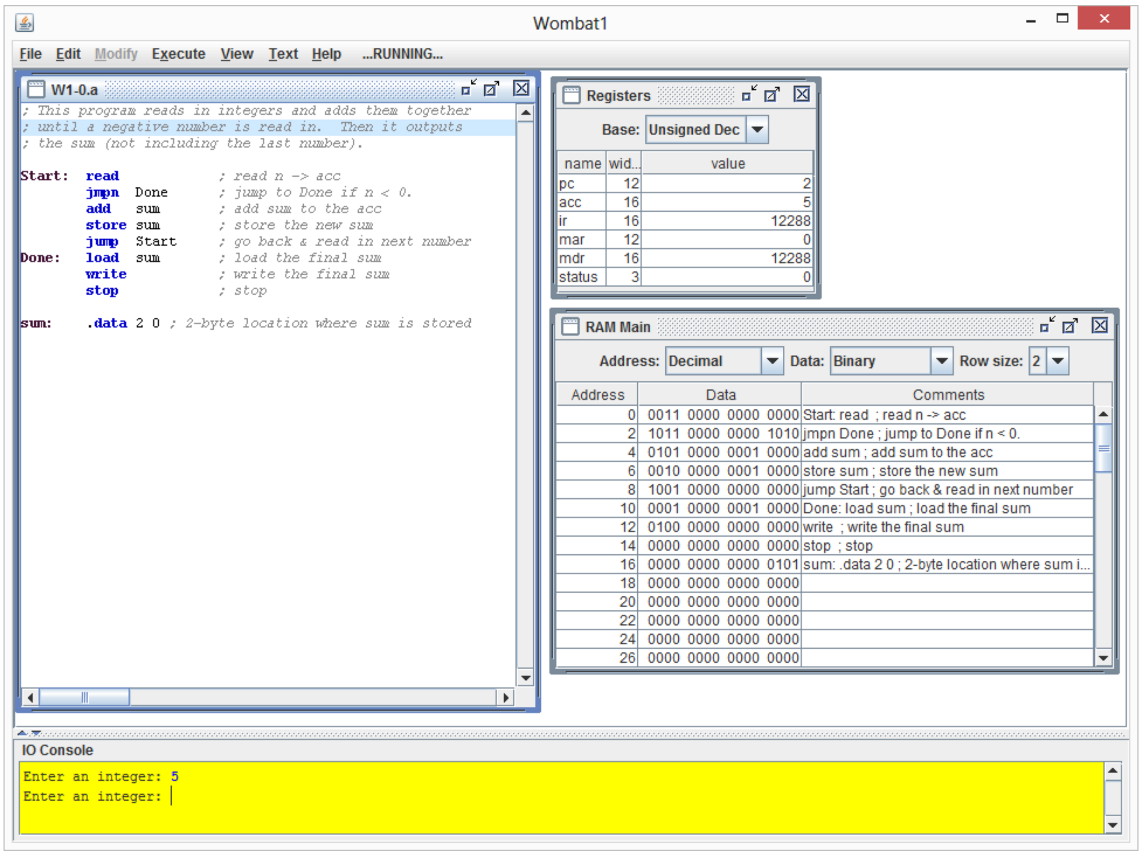
Instead of allowing you to build any architecture/organization, it gives you a CPU that can be configured by ROM. This means that you can't really change what is available, but you can define the internal signalling for the CPU. All the registers come predefined, so it's not possible to implement a machine with more than 8 user registers directly. The microcode ROM coding is very low-level and consists of defining which signal bits are activated in a specific microcode cycle, as show in Figure 2.2.

The simulated I/O Systems are also very complete and allow for DMA, IRQ, I2C, Ports, Event, Clock, Sensor, Keyboard, IO-RAM and some on-screen displays. It doesn't seem to be possible to write any kind of plugins to improve the simulator.

2.5.2 CPU Sim

CPU Sim (SKRIEN, 2001) is a cost-free closed-source ISS. It has the look-and-fell of an IDE, integrated with an assembler, views for registers, RAM and a simple I/O console, which can be seen in Figure 2.3.

To build an architecture, you have as tools a couple of editors: registers and condition codes can be added or removed freely, microinstructions can be created and removed in a very simple, albeit limited, system (Figure 2.4), the fetch cycle can be changed to any list



```
; This program reads in integers and adds them together
; until a negative number is read in.  Then it outputs
; the sum (not including the last number).

Start:  read            ; read n -> acc
        jmpn  Done      ; jump to Done if n < 0.
        add   sum       ; add sum to the acc
        store sum       ; store the new sum
        jump  Start     ; go back & read in next number
Done:   load  sum       ; load the final sum
        write           ; write the final sum
        stop            ; stop

sum:    .data 2 0 ; 2-byte location where sum is stored
```

Base: Unsigned Dec

name	wid...	value
pc	12	2
acc	16	5
ir	16	12288
mar	12	0
mdr	16	12288
status	3	0

Address: Decimal Data: Binary Row size: 2

Address	Data	Comments
0	0011 0000 0000 0000	Start: read ; read n -> acc
2	1011 0000 0000 1010	jmpn Done ; jump to Done if n < 0.
4	0101 0000 0001 0000	add sum ; add sum to the acc
6	0010 0000 0001 0000	store sum ; store the new sum
8	1001 0000 0000 0000	jump Start ; go back & read in next number
10	0001 0000 0001 0000	Done: load sum ; load the final sum
12	0100 0000 0000 0000	write ; write the final sum
14	0000 0000 0000 0000	stop ; stop
16	0000 0000 0000 0101	sum: .data 2 0 ; 2-byte location where sum i...
18	0000 0000 0000 0000	
20	0000 0000 0000 0000	
22	0000 0000 0000 0000	
24	0000 0000 0000 0000	
26	0000 0000 0000 0000	

```
Enter an integer: 5
Enter an integer: 
```

Figure 2.3: CPU Sim Running an Assembly Program

of microinstructions, instructions can be also easily created based on those microinstruction lists and the memory size can also be changed in a simple way.

As can be also seen in Figure 2.3, the I/O system is quite limited, being just a text console. As with MikroSim, it isn't possible to write any kind of plugins to improve the simulator.

2.6 Feature Comparison

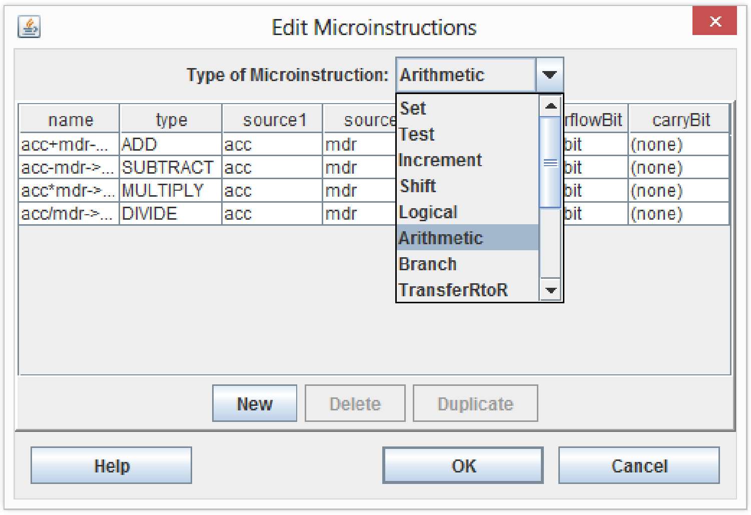
Many different systems were presented in this chapter, ranging from didactic to efficiency-driven ones. A direct comparison is a complicated endeavour, but their features can be seen side-by-side in table 2.2.

Considering that what is intended to be built is some kind of ISS, it makes sense to compare those in a deeper way. Table 2.3 shows that more specific feature comparison.

2.7 Conclusions

After considering all these systems, we can conclude that these many different systems have something we can take for the creation of a new ISS, a term that was learned through the research for this chapter.

From SIMH and MikroSIM, we took the lesson that too much complexity may be a



Type of Microinstruction: Arithmetic

name	type	source1	source	rflowBit	carryBit
acc+mdr-...	ADD	acc	mdr	bit	(none)
acc-mdr->...	SUBTRACT	acc	mdr	bit	(none)
acc*mdr->...	MULTIPLY	acc	mdr	bit	(none)
acc/mdr->...	DIVIDE	acc	mdr	bit	(none)

Figure 2.4: CPU Sim Microinstruction Editor
It’s basically about choosing a pre-existing operation, then selecting condition codes, registers, etc it operates with

	Concept	Main Features	Drawbacks
p-code	Simulated architecture with many VMs in different platforms.	Compile once, run anywhere; Portability.	Loses efficiency.
SIMH	Emulation of historically significant computers.	Many Systems; Console-operated; Versatile I/O.	Documentation Reading is necessary to use; Not user-friendly.
Console Emulators	Emulation of proprietary gaming console systems.	User-friendly; Pluggable I/O; Experience, not fidelity.	Legal issues; Usually needs high-end computers.
Weber Machines	Didactic simulators that teach assembly and architecture evolution.	User-friendly; Created as part of a teaching methodology.	Some were not redone with a GUI; Don’t teach organization.
MikroSim	Simulated architecture with a customizable operation ROM.	Teaches organization; Low-level coding; Many kinds of simulated I/O.	Registers cannot be changed; Not user friendly; No plugins.
CPU Sim	Architecture simulator with integrated assembly IDE.	Teaches organization; Easier organization coding, with mnemonics.	Limited I/O; No plugins; New microinstructions are defined in a very restrictive way.

Table 2.2: State-of-the-art Comparison Table
Compares the software seen in this chapter

Feature	MikroSIM	CPU Sim
Pluggable I/O	No	No
Customizable Registers	No	Yes
Teaches	Organization	Organization and Assembly
Assembler included	No	Yes
IDE included	No	Yes
I/O	Complex, many simulated systems	Simple, console-based
Instruction Coding	Signal-code microinstructions	Mnemonic microinstructions
Paid Software	Yes	No
Open Source	No	No

Table 2.3: Instruction Set Simulators Comparison Table
Compares the ISS seen in this chapter

bad thing, but that a versatile I/O system is a good idea. The gaming console emulators teach us that plugin system can be a good way to manage that I/O complexity and give some versatility. The Weber machines show us that teaching by doing is much better than by being lecture and were, probably, the biggest inspiration for this work. Last, but no least, CPU Sim gives us a pretty good system to compare with and, albeit with some minor issues, is probably the one that closest comes to the vision at the beginning of this work.

3 CONCEPTUALIZATION

Based on the conclusions from chapter 2 and the goals set on chapter 1, we can start specifying the system. As said before, a plugin system seems a good way to develop an I/O system, but it could also be used for defining an Arithmetic Logic Unit (ALU), giving freedom for the professor to include any operation he chooses by coding it. That considered, it seems like a good idea to separate in a basic system that manages the computer and an GUI system that manages the interface.

Two different definitions for the microinstructions appear in last chapter: one defined by which processor signals it activates and other based on mnemonics. Since on the architecture text book used for this text (WEBER, 2000), the microinstructions follow the mnemonic choice, that's also the one the proposed system will be using.

Being an educational tool, another very nice feature would be for it to be cross-platform.

Both plugins and machine definitions would have to be files with a defined structure, as shown in figure 3.1. The use of OO seems also like a good idea, the same figure also shows a proposed class diagram with 7 classes, where lines are communication paths and arrows mean reference in files or the fact that files are read by the 'Parser' class. The classes represent the CPU parts they emulate and the communication/control class is called 'Circuit'.

The applications goal would be to with some plugins developed by the professor (or coming with the application), any student with no programming language knowledge that is learning computer organization would be able, with microinstruction listings for every instructions, design a didactic simulator for a Von Neumann architecture in the style of the didactic machines presented in chapter 2. This is specially useful in a teaching context, where to teach the concepts of microinstructions, a professor could ask the students to implement simple computer architectures or to modify an existing architecture with some extra features.

Another feature that comes as a side-effect is that it is also easier for the professor to make small changes to the machines students have to work every semester, making sure every semester the challenge is a bit different. On the same page, it could be possible to update the machines with new concepts very easily for a specific course.

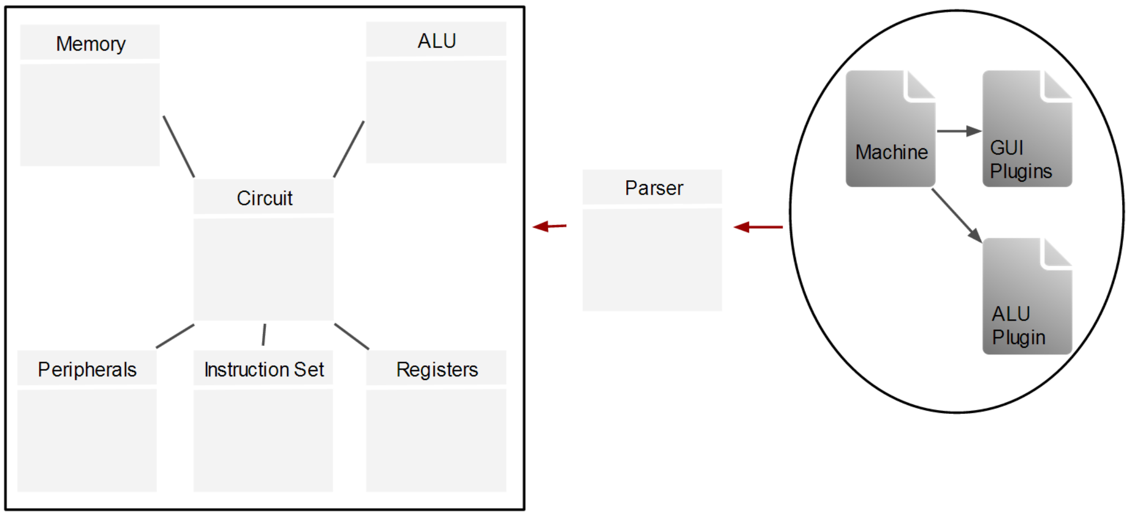


Figure 3.1: Class diagram of the simulator concept
Proposed class diagram, with 7 classes. Lines are communication paths, arrows mean file requirement/input

3.1 System

The basic system should be an ISS that receives a Machine Definition and it's necessary plugins. With those, it runs a didactic simulator for that machine, similar to those presented before in chapter 2.4.

It should have 7 classes, as shown in figure 3.1.

'Memory' class manages the computer's memory accesses, its reads and writes, keeps its data structure and counts the accesses.

'Register' class manages the computer's registers and condition codes (also called state registers and keeps their data structure.

'Instruction Set' class manages the translation of operation codes into references to runnable code.

'ALU' class manages the code that comes from the ALU plugin file and makes it available to 'Instruction Set' through 'Circuit'.

'Peripherals' class manages the GUI and all Input/Output (I/O) plugins available.

'Circuit' is the system's communication and control class, managing the other classes.

'Parser' class reads the input files and creates the object structures to run the system.

3.2 Machine and Plugin Files

Any machine that runs on the proposed ISS will consist of at least 3 files: a machine definition, an ALU plugin file and one or more I/O plugin files.

```
001.  mem val1 << val2
002.  reg << mem val
003.  reg << val
004.  reg << op val
005.  reg << op val1 val2
006.  if statereg then reg << val
007.  if !statereg then reg << val
```

Figure 3.2: Supported Microinstructions

3.2.1 Machine Definitions

A Machine Definition is a set of information useful for the framework to create a machine, it consists basically in everything that can be configured in the machines the system is able to run. It includes base, word size, memory size, a list of registers available, microinstruction listings for every instruction and a microinstruction listing for the fetch cycle.

3.2.1.1 Base and Word Size

A computer has to have a numerical base and word size, and nothing says it must be 8-bit binary. Why not test a 7-sized base-3 or a 5-decimal base-10 computer? In this system, it should be as easy as changing two input values.

3.2.1.2 Memory Size

Once you know how many representations one "byte" can accept, it makes sense to define how many positions the memory has. For simplicity, you can't define a value that cannot be addressed by the "byte".

3.2.1.3 Registers

Any computer has to have a set of registers that will be used in the microinstructions and fetch phase.

3.2.1.4 Microinstruction Listings

A computer needs to have an instruction set and, in this system, every instruction is defined of microinstructions. In the following lines, all the line number references are from figure 3.2 and a value can be either an integer or a register.

The transfer microinstruction on line 001 copies the value `val2` to the memory position `val1`.

The transfer microinstruction on line 002 copies the value stored in memory `val` to the register `reg`.

The transfer microinstruction on line 003 copies the value `val` to the register `reg`.

The operation microinstruction on line 004 puts the result of unary operation `op` (de-

```
001.  MAR << PC
002.  MDR << mem MAR
003.  IR << MDR
004.  PC << add PC 1
```

Figure 3.3: Fetch code example

fined in the ALU) with value `val` as parameter in the register `reg`.

The operation microinstruction on line 005 puts the result of binary operation `op` (defined in the ALU) with values `val1` and `val2` as parameters in the register `reg`.

The test microinstruction on line 006, if state register `statereg` has the value true, copies the value `val` to the register `reg`.

The test microinstruction on line 007, if state register `statereg` has the value false, copies the value `val` to the register `reg`.

3.2.1.5 Fetch Code

The fetch code is constructed like an instruction, using the same microinstructions to create the fetch instruction code. Figure 3.3 shows an example, in which the lines do, in order: copy the Program Counter to the Memory Address Register, read memory position in Memory Address Register to Memory Data Register, copy Memory Data Register to Instruction Register and increment Program Counter.

3.2.2 ALU and I/O Plugin Files

The ALU plugin file should contain three things: a list of state registers it supports, some executable code that updates the state registers after the ALU operations and, for each operation available, executable code that does any calculation that is needed to perform that operation. For each of those ALU operations, a microinstruction mnemonic would be created for the activation of the ALU with that operation.

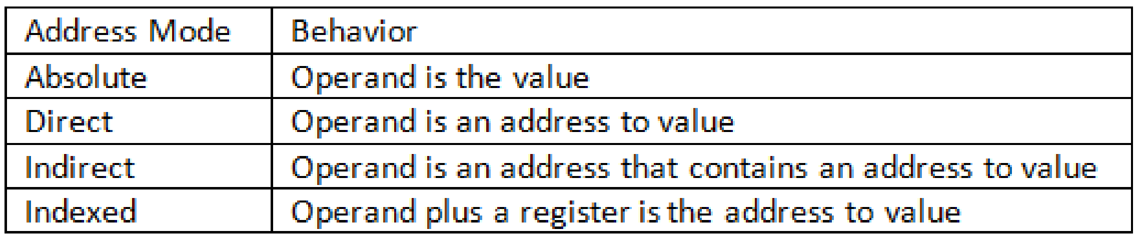
The I/O plugin files should define an element of the user interface, in order to allow versatility in choosing which plugins to use when configuring a machine file. They don't need to be restricted to graphical interfaces, as they may offer sound, read/write files or any other use needed by the users.

File format and what information must be in each plugin file is very hard to specify before choosing the programming language and beginning the implementation phase. Because of that, those details will be defined only in the next chapters.

3.3 Use Case Scenario

In this section a use case will be described where a professor wants to teach a class on addressing modes, using the proposed system. The addressing modes he wants to teach are in table 3.1. The use case steps are as follows:

1. The professor would define a machine with plugins, similar to the NEANDER



Address Mode	Behavior
Absolute	Operand is the value
Direct	Operand is an address to value
Indirect	Operand is an address that contains an address to value
Indexed	Operand plus a register is the address to value

Table 3.1: Address Modes

```
001.  MAR << PC
002.  PC << add PC 1
003.  MDR << mem MAR
004.  AC << add AC MDR
```

Figure 3.4: Absolute ADD operation

discussed in section 2.4. This machine has at least one operation defined. In this example, we will assume it is an absolute ADD instruction (shown in figure 3.4), and that the ALU has support for the 'add' operation.

2. Professor would explain to the students how the addressing modes work.

3. Students would be expected to create a new index register (here called 'IX') and produce three new instructions for the other addressing modes as shown in figures 3.5, 3.6 and 3.7.

4. The students could, then, test their instruction's behavior with the simulator.

```
001.  MAR << PC
002.  PC << add PC 1
003.  MDR << mem MAR
004.  MAR << MDR
005.  MDR << mem MAR
006.  AC << add AC MDR
```

Figure 3.5: Direct ADD operation

```
001.  MAR << PC
002.  PC << add PC 1
003.  MDR << mem MAR
004.  MAR << MDR
005.  MDR << mem MAR
006.  MAR << MDR
007.  MDR << mem MAR
008.  AC << add AC MDR
```

Figure 3.6: Indirect ADD operation

```
001.  MAR << PC
002.  PC << add PC 1
003.  MDR << mem MAR
004.  MDR << add IX MDR
004.  AC << add AC MDR
```

Figure 3.7: IX-indexed ADD operation

4 THE 'TANUKI' IMPLEMENTATION

Tanuki is an implementation of the concept presented in the previous chapter. The name comes from a Japanese species of racoon that, according to legend, has the ability to change into any form it wants. It follows the concepts defined before and tries to implements them.

It must be noted that plugins (both ALU and GUI) do execute code in you computer and are NOT secure (Running just a machine definition is). A Professor should never use a student's plugin without checking it's code beforehand.

4.1 Programming Language

The first step in implementing was to decide which language to use. Given time constraints, only three programming languages were considered, that is, the ones I already had previous knowledge of in bigger or smaller degrees. Those were Ruby, Java and C.

Important factors that needed to be taken on account were time needed, how hard it was to implement the plugin systems and the simplicity for parsing the machine files.

4.1.1 Java

Java in an OO programming language, compiled to a p-code (as seen in chapter 2), which guarantees a good portability. It's type safe, has garbage-collection and, at the cost of a highly verbose code, makes it harder to have run-time errors, as they are usually catch in compile time. For the plugins that our system needs, it offers an interesting solution, OSGi (HALL et al., 2011).

OSGi is a powerful modular service architecture, in which bundles of code are run separately and can be started or terminated in run-time. For this work's case, it's probably too complicated and not a time-effective solution to implement.

4.1.2 C

C is a widely-used imperative language, it is also the language that is taught at the first semester at our university and would be a nice option for the students to also be, by default, able to write the system's plugins. It's at the same time the lowest-level of the common high-level languages and brings some relationship to what the system intends to teach.

Plugins in C are usually implemented using Dynamic Shared Objects (DSOs) (in Windows systems, Dynamic-Link Librarys (DLLs)), which are not cross-platform and would defeat the portability that would be very interesting for an educational system like this. Besides that, having to care about every little implementation detail, as C usually does, would take more time than I had available.

4.1.3 Ruby

Ruby (RUBY, 2013) is an interpreted OO imperative and functional programming language, it's very simple to code and takes not much time and effort and, because of that, is one of the most used for rapid prototyping.

Being interpreted, it has support for what is called metaprogramming, which is the possibility of writing code that writes code and treating any string as code. This can be easily used to define plugins that include strings that are supposed to be run as code.

4.1.4 Choice

Giving the time restrictions and how much faster it would be to be implemented in Ruby, the obvious choice was Ruby. It also had an easy way to support the plugins and was portable, so it fit all the features I was looking for.

4.2 Graphical User Interface

The biggest project error was probably to start implementing in Ruby before getting to know the available GUIs. When the internal code was done and working and came time to decide which GUI to use, I discovered there were only bad choices, as they all had big drawbacks and most of the choices for Ruby either weren't native or were bad.

Some solutions were bindings for either C++ or Java code, so I decided to try something that could be written in Ruby. The only choice available was Ruby Shoes (SHOES, 2013), a lightweight and multi-platform framework that is able to generate executables that include the interpreter for easy deployment.

It seemed easy to use and was working well in first tests, so that was the choice taken. But when it came time to test the plugins working and updating, it became very slow. That was indeed because of the way Shoes worked, the only practical way to update was to make every object update some times per second and with a whole memory displaying, there were more than 200 objects to be updated and no event system was available to solve that issue.

To complicate the fact, the way plugins were developed made it impossible to implement any work-around, given the fact that a plugin could not access other plugin's code and objects. The end-result was, then, a graphical interface that worked, but took around 10 seconds to update its information.

Another problem with Shoes was that 'for' loops simply didn't operate correctly for updating the screen, because of the way Shoes was developed (using metaprogramming) and the way Ruby treats parameters (which is always call-by-value, but variables really store object names, which has some of the drawbacks of a call-by-reference language).

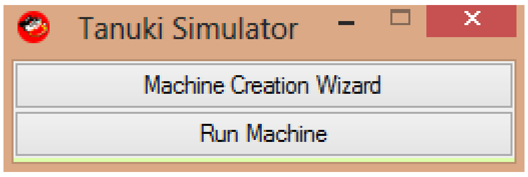


Figure 4.1: Tanuki Main Screen

The only way to solve that was repeating the same code many times in plugins like 'Memory Inspector'.

Figure 4.1 shows the main screen, after the system was complete.

4.3 How to Run a Machine

0. Copy the machine to Tanuki's machine folder
1. Open Tanuki
2. Click in 'Run a Machine'
3. Select the right machine
4. Done :)

4.4 Coding

The class structure projected in figure 3.1 ended up being a good project and suffered no major changes, except that some outside class code was needed to load Shoes and the main screen, everything else followed that diagram.

Some design choices clearly were not the best, specially in the way accessors for the registers were made. In many cases, instead of setting them up as Hash structures, they were added as fields (taking advantage of the metaprogramming) and that made the code much more complicated than it should. On the other hand, the translation of parsed microcode into instructions was made very easy by the advantages of using said metaprogramming and Ruby's excellent support for regular expressions.

Building a instruction's code consisted basically of concatenating strings that contained code, some of them originally from the ALU plugin, and then setting this resulting code as a method that was called whenever a call with the correct opcode was sent to the 'Instruction Set' class.

The 'Parser' class was a very simple parser, which indeed never needed to look back, because of the way the description languages were created (these languages are shown in chapter 5). It created the whole object structure when called and returned the control/access class 'Circuit'.

The 'Register' and 'Memory' classes were just data structures that kept those values and counted how many times each one was accessed, in case a plugin wanted to display that information.

The 'ALU' class was a repository for the code read from the ALU file. It was used in the construction of the 'Instruction Set' class, as its operations were microinstructions to be translated.

The 'Circuit' class contained the logic behind running the system, running the whole instruction cycle, including the translated fetch code. It also contained references for all the others and was referenced by all (the whole communication process went through it).

4.5 Implementation Scope

The system created can only simulate Von Neumann architectures, that is, the ones based on a Control Unit, an Arithmetic-Logic Unit, Registers and Memory. It also was defined for an instruction cycle that includes a fetch cycle. This means that a data flow computer would be very hard, if not impossible, to be implemented in this system.

In theory, with some operations in the ALU and some registers, the defined microinstruction system can perform any operation any Von Neumann architecture can, even if not in the most efficient way, as trivial corollary of the Church–Turing thesis. As exception to that would be operations depending on what would be considered external in a Classic Von Neumann model, like Clock, Interruptions or an extra processor. Instructions designed to gain efficiency may also make no sense to be implemented, since they probably would have no real effect (but they could, in theory, make sense if the implementation is done taking account of the Ruby interpreter's implementation).

Because of practical implementation reasons, no machine with more than 32 bits can be implemented, nor any machine in which $base^{wordsize}$ is greater than 2^{32}.

No memory stress test were performed, but at best case scenario, you can't use memory size bigger than half the host's available memory. (Since don't cares are treated internally by making multiple references to the same code, memory would end up at least twice as fast)

4.6 Comparison with other ISSs

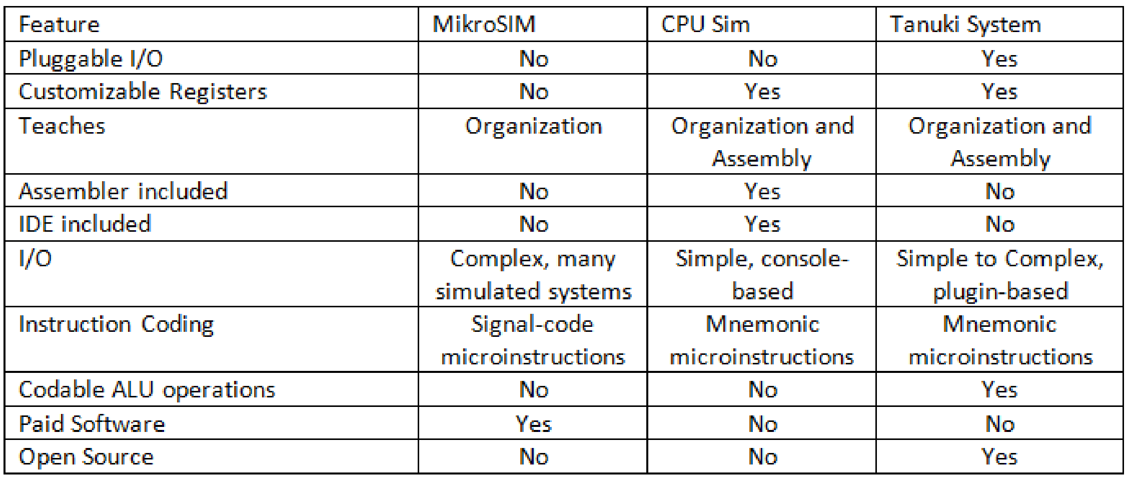
On chapter 2 there was a table (2.3) that compared both ISSs presented. Now, it makes sense to revisit that comparison table, including Tanuki, as we can see in table 4.1.

Tanuki is certainly not perfect, but it has some advantages in relation to the others, specially in the scope of machines that can be simulated by it. In that sense, we can compare it's register system with MikroSIM and see that it can have any number of registers. We can also compare it with CPU Sim and see that it's able to simulate any one or two-operand ALU operation, and not only those predefined.

CPU Sim still offers an IDE and Assembler that was out of scope for Tanuki, but could well be implemented in another version.

4.7 Implementation Conclusions and Lessons Learned

The biggest lesson taken from this implementation was to never assume things, since not checking if there was a good GUI in Ruby made it look like the best option, when



Feature	MikroSIM	CPU Sim	Tanuki System
Pluggable I/O	No	No	Yes
Customizable Registers	No	Yes	Yes
Teaches	Organization	Organization and Assembly	Organization and Assembly
Assembler included	No	Yes	No
IDE included	No	Yes	No
I/O	Complex, many simulated systems	Simple, console-based	Simple to Complex, plugin-based
Instruction Coding	Signal-code microinstructions	Mnemonic microinstructions	Mnemonic microinstructions
Codable ALU operations	No	No	Yes
Paid Software	Yes	No	No
Open Source	No	No	Yes

Table 4.1: Revisited Instruction Set Simulators Comparison Table
Compares Tanuki with the ISS seen in chapter 2

with that factored in, I would probably choose another language.

Other lesson was to not choose a not very tested framework: although there is a big Ruby community, the ones that used Shoes are not that many and those who tried building something more complicated with it are even less. The fact that the documentation not only wasn't complete, but there was almost no other source of information, except for the source code and a book (GILLETTE, 2013) that looks more like a modern art project than a documentation manual.

I don't really have anything to complain about Ruby, as it was an excellent language to work with and I recommend it for projects with no or simple GUIs. It's also very good for web, specially with the Ruby on Rails framework. It probably has better options for GUIs than the one I used, but there's not that much information about graphical ruby applications available and you have to search a lot for it in the web forums.

In essence, after comparing with the other ISSs, it seems that Tanuki achieved exactly the goal it wanted in concept and, except for some GUI problems that can be solved by rewriting that part, seems like a success.

5 DESCRIPTION LANGUAGES

The files conceptualized in section 3.2.2 need well-defined description languages in order to be implemented. This chapter describes such languages and some of its limitations. All the figures in this chapter are excerpts from the machine definition examples presented in Appendix A, with their original line numbers.

5.1 The Machine File

The machine file is a plain text file that contains all the definitions needed to run a machine in the Tanuki System.

5.1.1 Base, Wordsize, Memory and Registers

It contains lines starting with '`BASE: `', '`WORDSIZE: `' and '`MEM_SIZE: `', followed by numbers. Those lines define, respectively: the numerical base used by the computer (not bigger than 32), how many base-sized numerals form the minimal informational unit (as an example, the byte we are accustomed has 8 base-2 numerals) and the size of the memory (capped at base times wordsize, for addressing reasons). They are presented in figure 5.1.

Another line in the file starts with '`REGS: `' and is followed by all the registers the user wants in his machine separated by spaces. Figure 5.2 shows a definition of 5 registers (AC, IR, MDR, MAR and PC).

5.1.2 ALU Reference

The ALU file is referenced by a line starting with '`ALU: `' followed by the filename. The file is expected to be in the ALU folder of the program and its description is in section 5.2. Figure 5.3 shows the reference to a file named `neander.alu` in the ALU folder.

```
001.  BASE: 2
002.  WORDSIZE: 8
003.  MEM_SIZE: 256
```

Figure 5.1: Base, Wordsize and Memory Size Definitions

```
REGS: AC IR MDR MAR PC
```

Figure 5.2: Register Definition

```
ALU: neander.alu
```

Figure 5.3: ALU File Name Definition

5.1.3 Fetch Code

The fetch code, which is the code run before every instruction in order to prepare the Instruction Register, is defined using the microinstruction mnemonics conceptualized in section 3.2.1.4. The code begins with a starting word 'FETCH_START' and ends with the word 'FETCH_END'. Between those words, every line is read as a microinstruction mnemonic. Figure 5.4 has an example.

5.1.4 Instruction Codes

As with the fetch code, instructions are also defined using the microinstruction mnemonics conceptualized in section 3.2.1.4.

To start defining the instructions, the word 'INST_CODES_START' is used. Then, every instruction, starts with a line containing an operation code (that may or may not include so-called don't cares) followed or not by a mnemonic for that instruction (this is just intended for compatibility with any future assembler that could be created for the system) and ends with a line containing the word 'INST_END'. After the last instruction is defined, the word 'INST_CODES_END' must be used.

This operation code is a sequence of numbers lesser than the base defined before or 'X's meaning that any value is accepted (also known as don't cares). If the base is greater than 10, after the number 9, the letters of the alphabet are used (until 'v' since bases go up to 32).

Two additions were also made to the microinstruction mnemonics when writing instructions: they can have comment lines, if the first non-blank character is a '#', and they can have a mnemonic 'SIGNAL_HALT' that sends the machine a halt signal.

It's also interesting to point out that the values of the don't care bits can be obtained by operating with 'IR' (the computer‘s instruction register), making it possible to have partial operands or choosing addressing modes with those bits by using masks and/or shift

```
FETCH_START
  MAR << PC
  MDR << mem MAR
  IR << MDR
  PC << add PC 1
FETCH_END
```

Figure 5.4: Fetch Code Definition

```
INST_CODES_START
  0000XXXX NOP
    #DO NOTHING
  INST_END
(..)
  1111XXXX HLT
    SIGNAL_HALT
  INST_END
INST_CODES_END
```

Figure 5.5: Instruction Code Definitions

```
GUI_PLUGIN
  GRID_X: 2
  GRID_Y: 2
  PLUGIN simple_controller 0 0
  PLUGIN simple_register 1 0
  PLUGIN simple_memory 0 1
GUI_END
```

Figure 5.6: Peripheral Plugin References and GUI Box Definitions

operations.

The descriptions presented for instruction definition can all be seen in figure 5.5.

5.1.5 Peripheral/GUI Plugin References

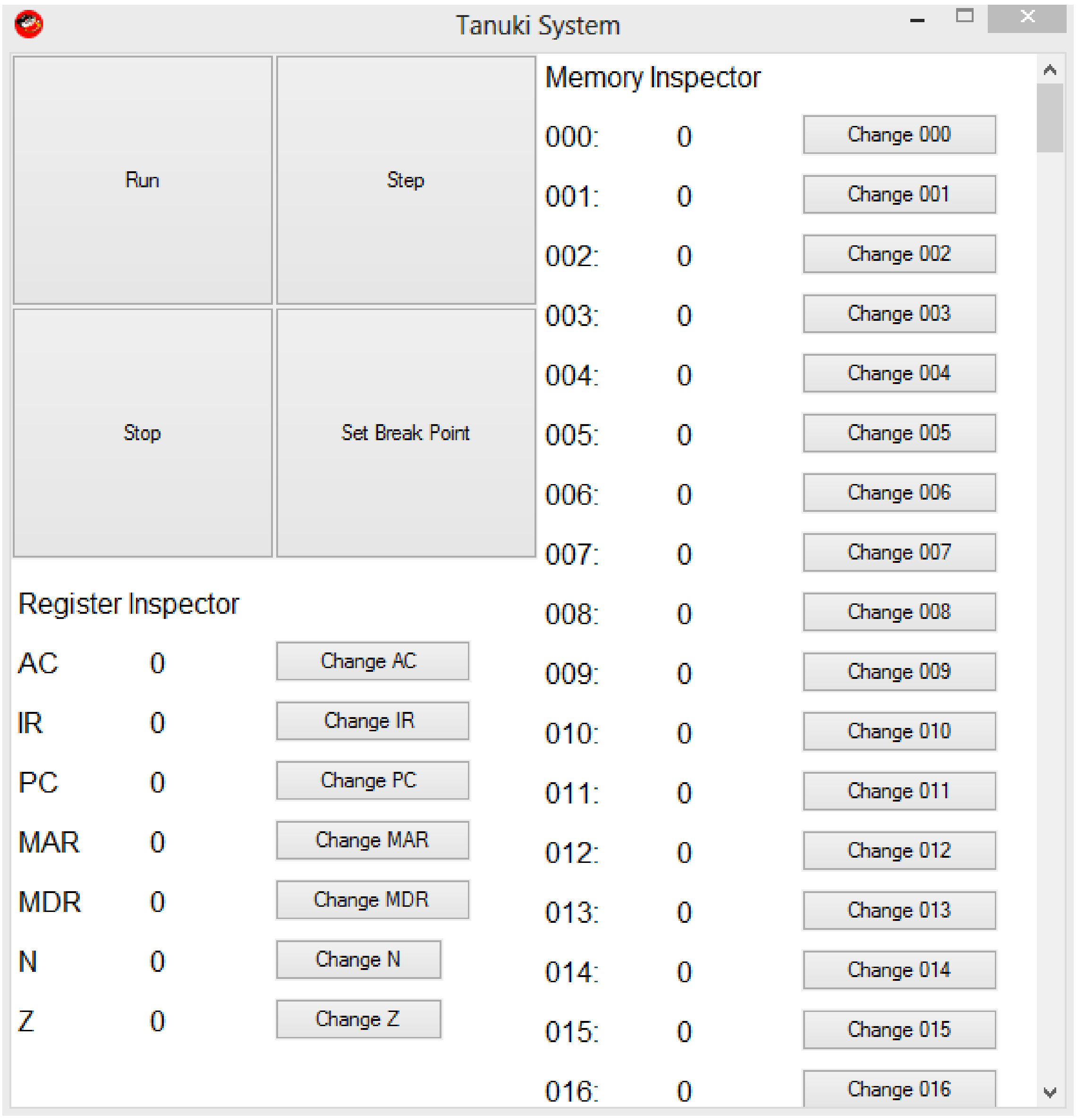
Before referencing GUI plugins, the word '`GUI_PLUGIN`' is used. Then, you have to define how many plugin boxes (of size 300 by 300 pixels) you want in X an Y directions, by using the words '`GRID_X: `' and '`GRID_Y: `' followed by numbers (you should make sure your computer fits the resolution of the hosts you want your machine to run on).

Then, for every plugin you want to load, use the word '`PLUGIN`' followed by the plugin's name (it should be in the '`Peripherals`' folder) and the position plugin boxes grid you defined before, where '`0 0`' is top left and '`GRID_X GRID_Y`' is bottom right. Remember that plugins may have sizes different than one by one, as can be seen in section 5.3.

To complete, you have to use the word '`GUI_END`'. All those definitions can be seen in figure 5.6

Figure 5.7 shows the machine said code generates, with the following plugins:

- A controller, in position (0,0) with size (1,1)
- A register inspector, in position (1,0) with size (1,1)
- A memory inspector, in position (0,1) with size (1,2)



Register Inspector

AC	0
IR	0
PC	0
MAR	0
MDR	0
N	0
Z	0

Memory Inspector

000:	0
001:	0
002:	0
003:	0
004:	0
005:	0
006:	0
007:	0
008:	0
009:	0
010:	0
011:	0
012:	0
013:	0
014:	0
015:	0
016:	0

Figure 5.7: Tanuki Neander Simple - A very basic NEANDER implementation with simple plugins

```
STATE_REGS: N Z
```

Figure 5.8: State Register Definition

```
STATE_UPDATE
  |val| val = val % @circuit.base_exp_word;
  if val == 0 then @circuit.reg.state_Z = 1
      else @circuit.reg.state_Z = 0 end ;
  if val >= ( @circuit.base_exp_word/
      @circuit.base )
      then @circuit.reg.state_N = 1
      else @circuit.reg.state_N = 0
      end;
END_SU
```

Figure 5.9: Code for Update Status

5.2 ALU file

The ALU file is a plain text file that contains the definitions of an ALU, which are: state registers it operates with, Ruby code for updates of those registers and Ruby code for every operation it supports.

5.2.1 State Register Definition

The file has a line that starts with '`STATE_REGS: `' and is followed by all the state registers the user wants his ALU to update separated by spaces. Figure 5.8 shows a definition of 2 state registers (N and Z).

5.2.2 Code for Updating State Registers

The ALU contains some code used to update the condition codes (also called state registers). This code begins with a starting word '`STATE_UPDATE`' and ends with the word '`END_SU`'. Between those words, a Ruby code is written that updates registers. This code receives a value that is the result from the ALU before any overflow (as Ruby works with more than 32 bit-sized numbers).

This code has access to internal variables and a state register can be accessed through '`@circuit.reg.state_X`', where '`X`' is that state register. It also has access to useful information like base ('`@circuit.base`'), wordsize ('`@circuit.wordsize`') and number of representations available ('`@circuit.base_exp_word`').

Figure 5.9 shows an example.

5.2.3 Operation Codes

For every operation, the ALU has to have a Ruby code for it. It should start with a line containing the operator's name and end with the word '`END_MI`'. Between those lines, a Ruby code block has to be inserted. This code can receive either one or two operands and

```
and
  |op1, op2| result = op1 & op2;
END_MI
not
  |op1| result = ~op1;
END_MI
```

Figure 5.10: Code for Operations

```
NAME: simple_controller
GRID_X: 1
GRID_Y: 1
```

Figure 5.11: Name and Grid Definitions

put the result in the variable '`result`'. Figure 5.10 has examples of both.

5.3 Creating a Peripheral Plugin

The peripheral (or GUI) plugin file is a plain text file that contains the definitions of a GUI plugin, which are: number of plugin boxes it uses both in X and Y directions and the plugins code.

5.3.1 Name and Grid Sizes

Optionally, you can define a plugin name, starting a line with '`NAME: `' and then writing the name.

Then, you have to define how many plugin boxes (of size 300 by 300 pixels) you want this plugin to occupy in X an Y directions, by using the words '`GRID_X: `' and '`GRID_Y: `', followed by a number. (Remember these sizes when using the plugin on a machine).

Those can be seen in 5.11

5.3.2 Plugin Code

To complete, you have to write the plugin's code, which begins with a starting word '`PLUGIN_CODE`' and ends with the word '`PLUGIN_END`'. Between those words, a Ruby Shoes code is written that displays the plugin and executes any logic needed. It has access to internal variables and the '`Circuit`' object can be accessed as the variable '`c`', for more information on the variables, the system's code should be checked.

The code for the controller shown in figure 5.7 is shown in figure 5.12

```
PLUGIN_CODE
 @controller = flow(:width => grid_size_x *
      $square_size, :height => grid_size_y *
      $square_size) do
   @stop = false
   @run_button = button "Run" do
     $halt = false
     @stop = false
     @run = true
   end
   @step_button = button "Step" do
     $halt = false
     c.step_machine
     if ($halt) then
       alert("Found HALT")
     end
   end
   @stop_button = button "Stop" do
     @stop = true
   end
   animate(30) do |frame|
     if (@run == true && $halt != true &&
          @stop != true) then
       c.step_machine
     end
     if (@run == true) then
       @run = false
       if ($halt) then
         alert("Found HALT")
       end
       if (@stop) then
         alert("Stopped by user!!")
       end
     end
      end
   @run_button.style(:width => grid_size_x *
        $square_size/2, :height =>
        grid_size_y*$square_size/2);
   @step_button.style(:width => grid_size_x *
        $square_size/2, :height =>
        grid_size_y*$square_size/2);
   @stop_button.style(:width => grid_size_x *
        $square_size/2, :height =>
        grid_size_y*$square_size/2);
 end
 @controller.move(pos_x*$square_size,
      pos_y*$square_size)
PLUGIN_END
```

Figure 5.12: Plugin Code

6 CONCLUSIONS AND FUTURE WORK

This work was, in essence, the specification and development of a simple, but interesting idea. The end result was a system that may have some practical functions, if bugs are fixed, another GUI is implemented and some improvements made. It served also as a good way to learn more about the Ruby language, including some quite complex metaprogramming structures. The best that came from it probably was the fact that it was a project developed, from idea to specification to implementation. This was one of the first opportunities in my bachelor's to do that.

6.1 Lessons

Besides the lessons already presented in chapter 4, some other lessons were also taken from the project as whole, the main being that documentation is best written during execution, since after it ends, ideas become so clear to us that it's more complicated to explain it in detail and we may even consider everything too trivial. This became a problem in earlier versions of this text and was quite hard to overcome.

Another good side-effect that could have come from having to explain my decisions in writing before implementing is that it could have been a better method to find out if my reasoning was sound and could have made me notice that I hadn't researched enough on the possible GUI frameworks available for Ruby.

6.2 Contribution

The biggest contribution this work gives is the full documentation of an implementation of an ISS using ruby and the problems that appeared. It also can be used as a cautionary tale for not starting to implement forgetting to analyse one detail, as even one minor detail can be the difference between success and failure.

6.3 Future Work

The most important thing would be to redo the whole GUI system. After that, an IDE and assembler would be nice features. Other features like interruptions and multi-threading that are not yet possible to implement using the framework would also be interesting. A lot of good plugins could then be developed in order to allow for emulation of more complete platforms. In my opinion, a quite interesting experience would be to try to implement a working complete-set 8086.

REFERENCES

AMMAN, U. et al. The Pascal'P'Compiler: implementation notes. **ETH Zurich**, [S.l.], 1974.

BURNET, M. M.; SUPNIK, R. M. Preserving computing's past: restoration and simulation. **Digital Technical Journal**, [S.l.], v.8, p.23–38, 1996.

CZERWINSKI, M.; PUTTKAMER, E. **MIKROSIM-SIMULATOR FOR MICROPROGRAM-ASSISTED DIGITAL SYSTEMS**. [S.l.]: FRIEDR VIEWEG SOHN VERLAG GMBH PO BOX 5829, W-6200 WIESBADEN 1, GERMANY, 1979. 513–514p. n.11.

DOLPHIN. **Dolphin**: a wii and gamecube emulator. Available at: http://www.dolphin-emulator.com/. Visited in November, 2013.

GILLETTE, J. **Nobody Knows Shoes**. [S.l.: s.n.], 2013. Available at: http://cloud.github.com/downloads/shoes/shoes/nks.pdf. Visited in December, 2013.

HALL, R. et al. **OSGi in action**: creating modular applications in java. [S.l.]: Manning Publications Co., 2011.

LINDHOLM, T. et al. **The Java virtual machine specification**. [S.l.]: Addison-Wesley, 2013.

RUBY. **Ruby Documentation**. Available at: https://www.ruby-lang.org/en/documentation/. Visited in October, 2013.

SHOES. **The Rules of Shoes**. Available at: http://shoesrb.com/manual/Rules.html. Visited in October, 2013.

SKRIEN, D. CPU Sim 3.1: a tool for simulating computer architectures for computer organization classes. **Journal on Educational Resources in Computing (JERIC)**, [S.l.], v.1, n.4, p.46–59, 2001.

TURING, A. Proposed electronic calculator. **Report, National**, [S.l.], 1946.

WEBER, R. F. **Fundamentos de arquitetura de computadores**. [S.l.: s.n.], 2000.

ZSNES. **ZSNES**: super nintendo entertainment system emulator. Available at: http://www.zsnes.com/. Visited in November, 2013.

AppendixA

A.1 NEANDER_SIMPLE_PLUGIN

The following code is the listing for a Machine File describing the NEANDER computer, seen in section 2.4, in the description language explained in section 5.1.

```
BASE: 2
WORDSIZE: 8
MEM_SIZE: 256
REGS: AC IR MDR MAR PC

FETCH_START
  MAR << PC
  MDR << mem MAR
  IR << MDR
  PC << add PC 1
FETCH_END
ALU: neander.alu
INST_CODES_START
  0000XXXX NOP
    #DO NOTHING
  INST_END
  0001XXXX #STO
    MAR << PC
    PC << add PC 1
    MDR << mem MAR
    MAR << MDR
    MDR << AC
    mem MAR << MDR
  INST_END
  0010XXXX LDA
    MAR << PC
    PC << add PC 1
    MDR << mem MAR
    MAR << MDR
    MDR << mem MAR
    AC << pass MDR
```

```
INST_END
0011XXXX ADD
  MAR << PC
  PC << add PC 1
  MDR << mem MAR
  MAR << MDR
  MDR << mem MAR
  AC << add AC MDR
INST_END
0100XXXX OR
  MAR << PC
  PC << add PC 1
  MDR << mem MAR
  MAR << MDR
  MDR << mem MAR
  AC << or AC MDR
INST_END
0101XXXX AND
  MAR << PC
  PC << add PC 1
  MDR << mem MAR
  MAR << MDR
  MDR << mem MAR
  AC << and AC MDR
INST_END
0110XXXX NOT
  AC << not AC
INST_END
1000XXXX JMP
  MAR << PC
  PC << add PC 1
  MDR << mem MAR
  PC << MDR
INST_END
1001XXXX JN
  MAR << PC
  PC << add PC 1
  MDR << mem MAR
  if N then PC << MDR
INST_END
1010XXXX JZ
  MAR << PC
  PC << add PC 1
  MDR << mem MAR
  if Z then PC << MDR
INST_END
1111XXXX HLT
  SIGNAL_HALT
  INST_END
INST_CODES_END

GUI_PLUGIN
  GRID_X: 2
  GRID_Y: 2
  PLUGIN simple_controller 0 0
  PLUGIN simple_register 1 0
  PLUGIN simple_memory 0 1
GUI_END
```

A.2 Simple GUI Plugin (Controller)

The following code is the listing for a GUI Plugin File (Specifically, a machine controller plugin), in the description language explained in section 5.1.5.

```
NAME: simple_controller
GRID_X: 1
GRID_Y: 1
PLUGIN_CODE
 @controller = flow(:width => grid_size_x *
      $square_size, :height => grid_size_y *
      $square_size) do
   @stop = false
   @run_button = button "Run" do
     $halt = false
     @stop = false
     @run = true
   end
   @step_button = button "Step" do
     $halt = false
     c.step_machine
     if ($halt) then
       alert("Found HALT")
     end
   end
   @stop_button = button "Stop" do
     @stop = true
   end
   animate(30) do |frame|
     if (@run == true && $halt != true &&
          @stop != true) then
       c.step_machine
     end
     if (@run == true) then
       @run = false
       if ($halt) then
         alert("Found HALT")
       end
       if (@stop) then
         alert("Stopped by user!!")
       end
     end
      end
   @run_button.style(:width => grid_size_x *
        $square_size/2, :height =>
        grid_size_y*$square_size/2);
   @step_button.style(:width => grid_size_x *
        $square_size/2, :height =>
        grid_size_y*$square_size/2);
   @stop_button.style(:width => grid_size_x *
        $square_size/2, :height =>
        grid_size_y*$square_size/2);
 end
 @controller.move(pos_x*$square_size,
      pos_y*$square_size)
PLUGIN_END
```

A.3 NEANDER ALU

The following code is the listing for an ALU File describing the ALU for the NEANDER computer, seen in section 2.4, in the description language explained in section 5.2.

```
STATE_REGS: N Z
STATE_UPDATE
  |val| val = val % @circuit.base_exp_word;
  if val == 0 then @circuit.reg.state_Z = 1
      else @circuit.reg.state_Z = 0 end ;
  if val >= ( @circuit.base_exp_word/
      @circuit.base )
      then @circuit.reg.state_N = 1
      else @circuit.reg.state_N = 0
      end;
END_SU
add
  |op1, op2| result = (op1 + op2);
END_MI
or
  |op1, op2| result = op1 | op2;
END_MI
and
  |op1, op2| result = op1 & op2;
END_MI
not
  |op1| result = ~op1;
```

```
END_MI
pass
  |op1| result = op1;
END_MI
```

GLOSSARY

architecture See computer architecture. 21, 26, 33

base The numerical base in which a computer operates, usually 2. 35

computer architecture Computer architecture is a computer definition by it's parts and their relationships. 21, 26, 33

condition code A condition code is a value that is set by the ALU, considering what happened in it's operation. 27, 28, 34

emulation Emulation is the process of simulating a real machine in order to run legacy code. 22, 23, 25, 53

fetch The act of gathering the next instruction or operand, updating the registers needed for that i.e. program counter. 23

fetch code A set of microinstructions that prepares the next instruction the processor has to run. 36, 42, 46

fetch cycle Is the first ste of the instruction cycle, where the instruction will be loaded from memory in order to continue. 28, 35, 42

host In simulation or emulation, a host is the computar that runs the simulator or emulator. 25

instruction A instruction is the minimal order a programmer can give to a processor. 27, 29, 33, 41, 46

microarchitecture Microarchitecture is a description of how to implement a given architecture on a processor. 21

microcode A code implemented using microinstructions. 27

microinstruction Microinstructions are internal instructions that a processor uses to build more complex instructions. 21, 28, 29, 33, 35, 36

platform A set of Software, Computer and Peripherals that works together. 23, 25, 40, 54

register A processor component that stores a value. 26–28, 34, 35, 49

simulation Simulation is the process of simulating a machine using another. 22, 23

state register A processor component that stores a boolean value, usually a condition code. 26, 27, 34, 36, 49

virtual machine A virtual machine is a software implementation of a computer. 23

word The size of the minimum information a computer can handle, usually 8-bits. 35