FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Tool to Support Computer Architecture Teaching and Learning

Bruno Miguel Barroso da Nova

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: António José Duarte Araújo

Second Supervisor: João Paulo de Castro Canas Ferreira

July 17, 2013

Tool to Support Computer Architecture Teaching and Learning

Bruno Miguel Barroso da Nova

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Doctor Alexandre Miguel Barbosa Valle de Carvalho

External Examiner: Doctor Iouliia Skliarova

Supervisor: Doctor António José Duarte Araújo

July 17, 2013

Abstract

Computer architecture is an important subject for informatics and electrical engineering courses, where students get to know how a CPU works internally. However, the students exhibit some difficulties in this subject. This is due to the lack of versatile educational tools that simulate the operation of a processor in an intuitive, integrated, graphical and configurable way.

One of the most used processor architectures for teaching computer architecture is MIPS. The architecture has a few different versions, but the most used for teaching are the unicycle and the 5-stage pipeline.

In this dissertation, an educational MIPS simulator, DrMIPS, is described. This tool simulates the execution of an assembly program on the CPU, step-by-step, and displays the status of the datapath graphically. Registers, data memory and assembled code are also displayed, and a “performance mode” for latencies and critical path analysis is also provided. Both unicycle and pipeline implementations are supported and the CPUs and their instruction sets are configurable. The pipeline implementation includes complete hazard detection and resolution.

The tool seeks to help students to understand topics like the composition and operation of a datapath, pipelining, instruction encoding and processor measuring. It is available not only for PCs but also for Android tablets. None of the other existing tools have a version for Android and this is a platform that is becoming very popular. The tool supports multiple languages and is fairly intuitive and versatile on both platforms.

Resumo

A arquitectura de computadores é uma disciplina importante dos cursos de engenharia informática e electrotécnica, onde os estudantes ficam a conhecer como um CPU funciona internamente. No entanto, os estudantes demonstram algumas dificuldades nesta disciplina. Isto deve-se à ausência de ferramentas educativas versáteis que simulem o funcionamento de um processador de forma intuitiva, integrada, gráfica e configurável.

Uma das arquitecturas de processadores mais usadas para o ensino de arquitectura de computadores é o MIPS. A arquitectura tem algumas versões diferentes, mas as mais usadas no ensino são as versões uniciclo e *pipeline* de 5 etapas.

Nesta dissertação, um simulador educativo do MIPS, DrMIPS, é descrito. Esta ferramenta simula a execução de um programa em *assembly* no CPU, passo-a-passo, e mostra o estado do caminho de dados graficamente. Registos, memória de dados e código assemblado também são mostrados, e um "modo de desempenho" para análise de latências e caminho crítico é fornecido. Ambas as implementações uniciclo e *pipeline* são suportadas e os CPUs e seus conjuntos de instruções são configuráveis. A implementação *pipeline* inclui detecção e resolução completa de conflitos.

A ferramenta pretende ajudar os estudantes a entender tópicos como a composição e funcionamento de um caminho de dados, *pipelining*, codificação de instruções e desempenho de processadores. Está disponível não só para PCs mas também para tablets Android. Nenhuma das outras ferramentas existentes tem uma versão para Android e esta é uma plataforma que está a ganhar bastante popularidade. A ferramenta suporta vários idiomas e é bastante intuitiva e versátil em ambas as plataformas.

Acknowledgements

This dissertation is the result of not only my personal work but also the contribution and support of other people. I want to thank these people for helping during this work.

I start by thanking my family for encouraging and supporting me, both affectionately and economically.

I also want to give my thanks to my supervisors, Professor António José Duarte Araújo and Professor João Paulo de Castro Canas Ferreira, for advising and orienting me during this dissertation, and also for proposing this dissertation.

Finally, I would like to thank all the teachers and people of the Faculty of Engineering of the University of Porto, especially of the Department of Informatics Engineering, for all the knowledge and services provided.

Bruno Miguel Barroso da Nova

“There are two ways of constructing a software design:
One way is to make it so simple that there are obviously no deficiencies,
and the other way is to make it so complicated that there are no obvious deficiencies.
The first method is far more difficult.”

Charles Antony Richard Hoare

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

ALU	Arithmetic and Logic Unit
ASP	Active Server Pages
CISC	Complex Instruction Set Computer
CPI	Cycles Per Instruction
CPU	Central Processing Unit
FEUP	Faculty of Engineering of the University of Porto
GUI	Graphical User Interface
IDE	Integrated Development Environment
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
MARS	MIPS Assembler and Runtime Simulator
MIEEC	Mestrado Integrado em Engenharia Electrotécnica e de Computadores
MIEIC	Mestrado Integrado em Engenharia Informática e Computação
MIPS	Microprocessor without Interlocked Pipeline Stages
MIT	Massachusetts Institute of Technology
PC	Personal Computer
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
UML	Unified Modeling Language
XML	eXtensible Markup Language

Chapter 1

Introduction

This thesis was written for the dissertation of the MIEIC (*Mestrado Integrado em Engenharia Informática e Computação*) of the Faculty of Engineering of the University of Porto (FEUP) and its aim is to present and describe the work done during the dissertation and the methodology used.

The work described in this document refers to a software tool that was developed to help students to learn about processor architectures, using the Microprocessor without Interlocked Pipeline Stages (MIPS) architecture as an example.

This chapter is an introduction to the document, presenting the motives that led to this work and the objectives that it aimed to achieve. It ends with the outline of the rest of the document.

1.1 Context and Motivation

Computer architecture is an important subject in the syllabus of Informatics and Electrical Engineering courses, such as the MIEIC and MIEEC (*Mestrado Integrado em Engenharia Electrotécnica e de Computadores*) of FEUP. Here, students get to know the basics of how processors and computers work, learning topics like data representation on the computer, digital circuits, conceptual composition of a Central Processing Unit (CPU), assembly programming and processor performance.

However, many students exhibit difficulties understanding various topics on this subject, such as pipelined processors and calculating processor performance. Teachers and researchers from FEUP have concluded that these difficulties exist mostly due to the absence of tools on an integrated environment that are geared towards education. More specifically, tools that allow students to view the composition of a CPU's datapath[1] graphically and consult detailed information about the data in each functional block, data being transmitted on the buses and control signals for each instruction on a set of instructions executed by the CPU.

[1]Datapath: the component of the processor that performs arithmetic and logical operations. [PH05]

There are already many tools created to simulate the operation of a CPU, and even some of them have graphical interfaces to show the CPU's datapath. However, most of them are not very adequate for educational purposes, are difficult to use and understand or are too specific for some problem and not very flexible.

Due to this lack of educational processor simulators, a versatile and intuitive MIPS simulator was developed. It aggregates several features that are found scattered through the other existing tools, aiming to aid both students in learning and teacher in teaching this important subject that is computer architecture. The simulator was named DrMIPS, a name that was inspired by DrScheme/DrRacket [PLT13], the programming environment used in the first semester of the first year of the MIEIC course at FEUP.

1.2 Objectives

The main objective of this dissertation was to create a tool to support computer architecture teaching and learning. This educational tool is, more specifically, a simulator of a processor, the MIPS processor, which is a well-known processor in the computer architecture academic community and also one of the most used processors for teaching computer architecture courses in universities [Per09].

The development of this tool was initially based on the following requirements:

- Allow the configuration of the datapath by allowing the configuration and parametrization of each individual block, defining their interface, functionality and latency. With this, anomalies can be introduced, simulating faulty processors and showing the consequences.
- Execute the simulation either step-by-step[2] or resorting to animations, or both. On each step the tool should display visually detailed information about the datapath, more specifically: values of the registers, values of the signals on the buses and at the inputs and outputs of each block, and values of each memory variable in the Random-Access Memory (RAM). The values on the datapath are represented in binary, hexadecimal and decimal.
- Simulate both the unicycle and the pipelined versions of the processor. Simulating the pipelined version also involves simulating the hazards that can occur and the methods used by the processor to resolve them, like shortcuts and stalls.
- Allow the execution of not only an assembly program but also a single individual instruction. The execution of a single instruction allows the students to watch and understand the execution steps of each instruction.
- Identify the critical path and calculate the processor performance though the simulated latency of the datapath components.

[2]Each execution step is a clock cycle.

- Be simple to use and easy to understand. This is crucial since the simulator's objective is to aid students to learn about computer architecture and not serve as an obstacle for that. The students shouldn't need to spend too much time learning how to use the simulator.

The goal of the tool is to help students to learn about computer architectures. By using the tool the students are expected to better understand:

- The composition and operation of a "simple" datapath.
- How instructions are encoded into machine code and how they are executed.
- The values of the signals in the datapath.
- Relevant blocks and control signals for each instruction.
- Pipelining, hazards, forwarding and stalls.
- Performance measuring and critical path identification.

The tool was developed mainly for personal computers, but a version for touch screen Android devices, especially tablets, was also created.

1.3 Work Summary

In this dissertation an educational MIPS simulator was developed. The simulator, DrMIPS, can be executed on most Personal Computer (PC) operating systems and also on Android devices. The user can load or write an assembly program using the built-in code editor and then see and understand, step-by-step, how it is executed in the CPU. In each step the simulator presents detailed information about the state of the processor, namely: the assembled code, the instructions being executed, the registers, the data memory and the CPU's datapath and values flowing through it. The performance of the processor is also simulated and the user can view the components' latencies and the CPU's critical path. DrMIPS can simulate several different MIPS unicycle and pipeline datapaths and these can be completely configured, including their instruction sets, or new ones can be created.

1.4 Document Structure

Besides this introduction, this document is composed by six more chapters. In Chapter 2 the State-of-the-Art is discussed and related work and tools are presented, indicating their strengths and weaknesses. Chapter 3 presents a brief overview of the MIPS architecture and describes the methodology used for the development of this work. Chapter 4 discusses how the internal simulation logic was implemented and how the CPU and the instruction set is represented. Chapter 5 presents and details the implementation of the graphical user interface of both PC and Android versions. Chapter 6 discusses some examples of how the tool can be used by students to better understand several topics of computer architecture. Finally, in Chapter 7 a review of what was accomplished in this dissertation is made and some suggestions for future work are presented.

Chapter 2

Related Work

As mentioned in the previous chapter, various tools to simulate the operation of a CPU already exist and some of them even display the composition of the datapath visually. Most of them, however, are not very suitable to teach students of computer architecture courses. They are too difficult for a student to use, or have a very specific objective and focus only on the unicycle or the pipeline version of the CPU, among other problems. This chapter presents and describes some of the most relevant educational simulators of the MIPS processor, mentioning how adequate they are for educational purposes, if they are open-source, and their problems and strong points.

2.1 SPIM

SPIM [Lar] is an open-source simulator, written in C++ by James Larus, that runs MIPS32 programs [Lar]. It was widely used, both for education and for the industry [VS06], and supports a large number of MIPS instructions, including syscalls[1] and some floating point[2] operations [Lar90, Lar]. Recently, a new version of the tool using Qt [Dig13] was released with the name *Qt-Spim*, allowing the tool to run in all major operating systems with the same code and user interface [Lar].

When SPIM is executed, a window like the one shown in Figure 2.1 is displayed. The window can display the registers, including the floating point ones, the text segment[3] and the data segment[4]. The text segment initially contains initialization code. Besides the main window, another window representing the console, for input and output, is displayed. SPIM doesn't provide a code editor, so

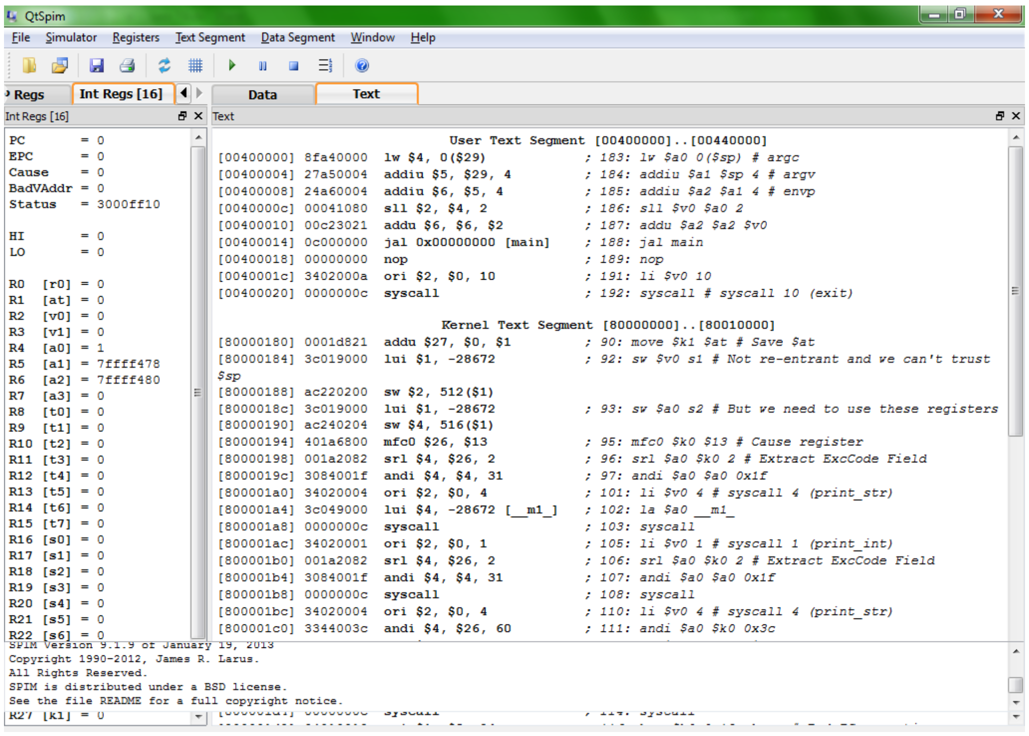
[1]Syscall (or system call): A special instruction that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process. [PH05]

[2]Floating point: Computer arithmetic that represents numbers in which the binary point is not fixed. [PH05]

[3]Text segment: The segment of a UNIX object file that contains the machine language code for routines in the source file. [PH05]

[4]Data segment: the segment of a UNIX object or executable file that contains a binary representation of the initialized data used by the program. [PH05]

the code must be written by an external tool and then loaded. After that, the code can be executed completely or step-by-step. The user can also set breakpoints, edit the values of the registers and data memory during the simulation and choose to display these values in either binary, decimal or hexadecimal formats.


QtSpim
File Simulator Registers Text Segment Data Segment Window Help
Regs
Int Regs [16]
Data
Text
Int Regs [16]
Text

```
PC       = 0
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 1
R5  [a1] = 7ffff478
R6  [a2] = 7ffff480
R7  [a3] = 0
R8  [t0] = 0
R9  [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R27 [k1] = 0
```

```
                          User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c000000  jal 0x00000000 [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10           ; 191: li $v0 10
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)

                         Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1         ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672           ; 92: sw $v0 s1 # Not re-entrant and we can't trust
$sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672           ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13            ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2           ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31          ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4            ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]   ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                  ; 103: syscall
[800001ac] 34020001  ori $2, $0, 1            ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082  srl $4, $26, 2           ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f  andi $4, $4, 31          ; 107: andi $a0 $a0 0x1f
[800001b8] 0000000c  syscall                  ; 108: syscall
[800001bc] 34020004  ori $2, $0, 4            ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c  andi $4, $26, 60         ; 111: andi $a0 $k0 0x3c
```

```
SPIM Version 9.1.9 of January 19, 2013
Copyright 1990-2012, James R. Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
```

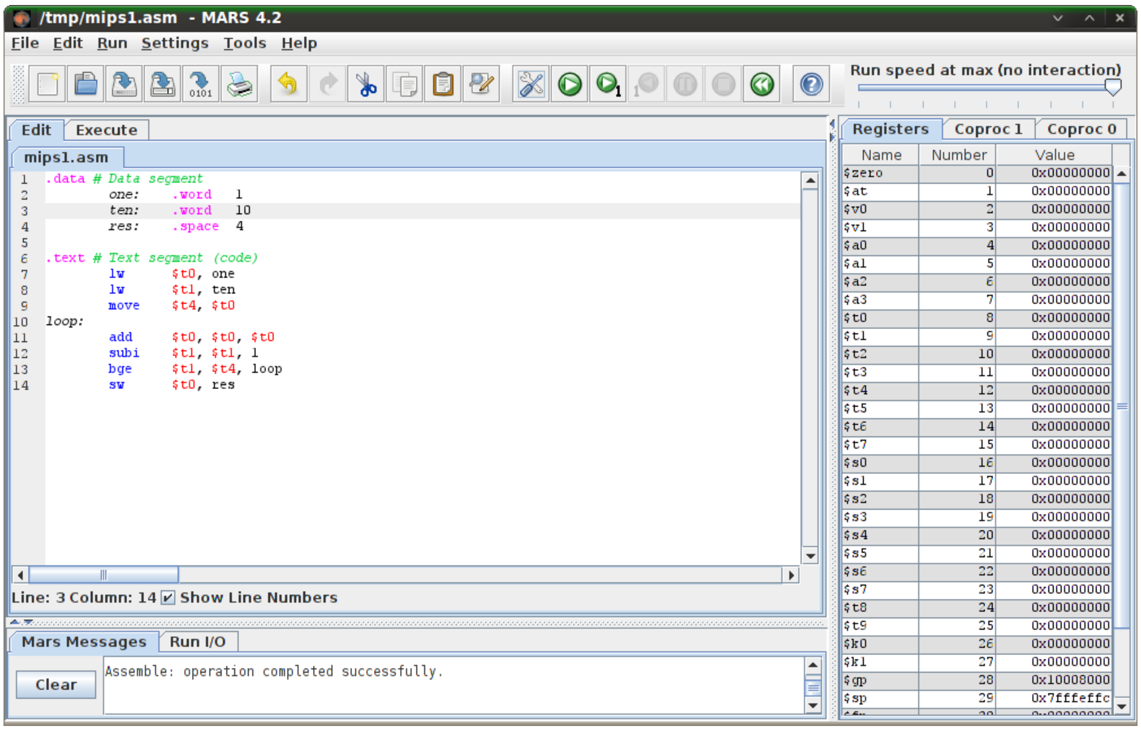
Figure 2.1: QtSpim (SPIM) user interface

The tool is good for debugging MIPS assembly programs and is reasonably intuitive, although the initialization code shown can confuse the user at first, and it would benefit from an integrated code editor. SPIM simulates the unicycle version of the processor and doesn't display any visual representation of the CPU's datapath and, thus, only covers computer architecture topics related to assembly programming and general CPU operation.

2.2 MARS

The MIPS Assembler and Runtime Simulator (MARS) [Uni12] was created by Dr. Pete Sanderson and Dr. Kenneth Vollmar [VS06] and is used in computer architecture courses in many faculties all over the world. It simulates the execution of a MIPS assembly program and shows the results on the screen. It was developed in Java and, therefore, can be executed in most operating systems, provided they have a Java Runtime Environment (JRE) installed.

The simulation can be executed at once or step-by-step, one instruction at a time. During the simulation, the tool displays not only the resulting program outputs and inputs but also the values of each register, the compiled code and the data segment. The values can be displayed in hexadecimal or decimal bases. The user can edit the values of the registers and the data memory



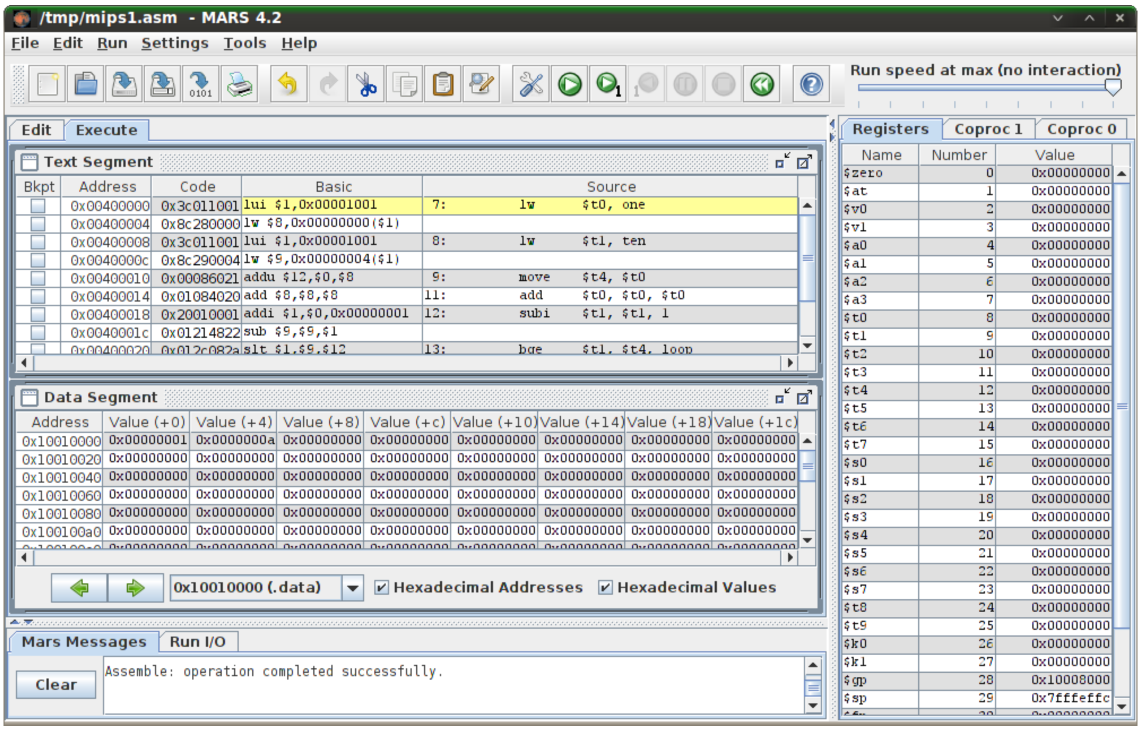
```
.data # Data segment
        one:    .word   1
        ten:    .word   10
        res:    .space  4

.text # Text segment (code)
        lw      $t0, one
        lw      $t1, ten
        move    $t4, $t0
loop:
        add     $t0, $t0, $t0
        subi    $t1, $t1, 1
        bge     $t1, $t4, loop
        sw      $t0, res
```

Line: 3 Column: 14 ☑ Show Line Numbers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc

```
Assemble: operation completed successfully.
```

Figure 2.2: MARS edit window



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	7: lw $t0, one
☐	0x00400004	0x8c280000	lw $8,0x00000000($1)	
☐	0x00400008	0x3c011001	lui $1,0x00001001	8: lw $t1, ten
☐	0x0040000c	0x8c290004	lw $9,0x00000004($1)	
☐	0x00400010	0x00086021	addu $12,$0,$8	9: move $t4, $t0
☐	0x00400014	0x01084020	add $8,$8,$8	11: add $t0, $t0, $t0
☐	0x00400018	0x20010001	addi $1,$0,0x00000001	12: subi $t1, $t1, 1
☐	0x0040001c	0x01214822	sub $9,$9,$1	
☐	0x00400020	0x012c082a	slt $1,$9,$12	13: bge $t1, $t4, loop

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000001	0x0000000a	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc

```
Assemble: operation completed successfully.
```

Figure 2.3: MARS execution window

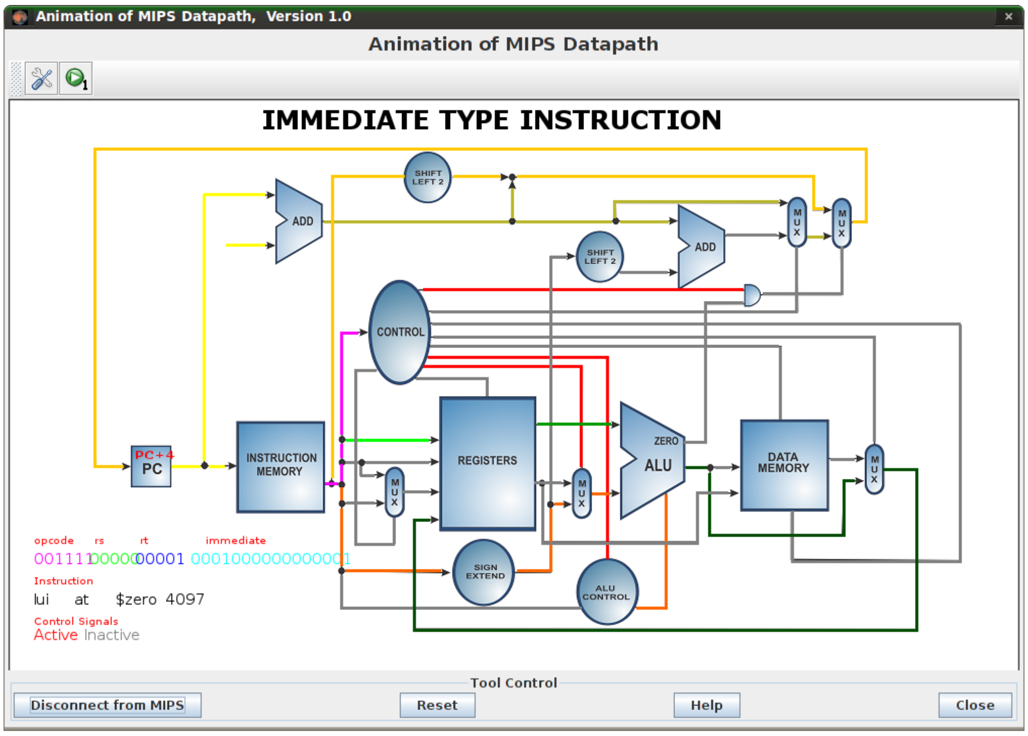
easily during the simulation. Breakpoints can be set in the code and the step-by-step simulation even allows to undo steps. MARS also simulates the floating-point co-processor and implements various pseudo-instructions and syscalls.

Besides a simulation tool, MARS is also an Integrated Development Environment (IDE) that includes an editor with syntax highlighting, help tooltips in the editor and a help window with a list of MIPS instructions, syscalls, etc. Two screenshots of the MARS edit and execution window are shown in figures 2.2 and 2.3 respectively.

The tool is very good for simulating and debugging MIPS assembly programs. However, computer architecture courses place a strong emphasis on the conceptual composition of processors and pipelined architectures. MARS doesn't display the CPU visually neither allows the simulation of the pipeline version of the MIPS, executing the instructions like the unicycle version.

MARS is an open-source software licensed under the Massachusetts Institute of Technology (MIT) license[5]. Plugins for MARS can be developed by extending the *AbstractMarsToolAndApplication* class inside the *mars.tools* package [SAPJ10]. The plugins can then be started from the *Tools* menu of the simulator.

Several plugins have been developed for the MARS simulator. One of these plugins is the MIPS X-Ray.



opcode rs rt immediate
001111 00000 00001 0001000000000001
Instruction
lui at $zero 4097
Control Signals
Active Inactive

Figure 2.4: MIPS X-Ray plugin window during a simulation

MIPS X-Ray displays a window with the datapath of a unicycle MIPS processor [SAPJ10]. During the execution of an assembly program by the MARS simulator an animation is shown by

[5]MIT license: http://courses.missouristate.edu/kenvollmar/mars/license.htm

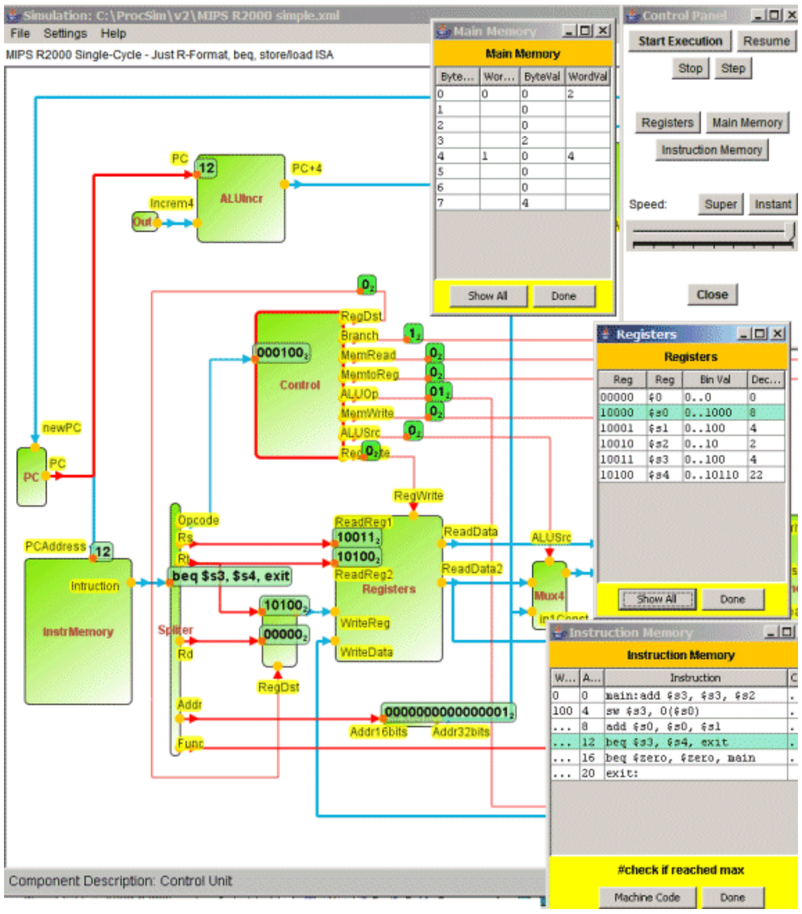
the plugin, displaying what buses and blocks each instruction uses. The type of the instruction and its machine code are also displayed.

The source code of the plugin was found on a recent Google Code repository at `https://code.google.com/p/plugin-mips-xray`. Figure 2.4 displays the plugin running, compiled from the code of that repository.

The plugin only highlights the buses used by each instruction, through an animation with a fixed speed, without displaying neither the data transmitted in those buses nor the components' inputs and outputs. Also, the plugin is very CPU intensive.

2.3 ProcSim

ProcSim, or ProcessorSim, is a tool developed by James Garton in 2005 for his Master's degree in Software Engineering. The tool simulates a processor's internal circuits executing a piece of assembly code as an animation. The simulator is based on the MIPS R2000 unicycle processor, is aimed for people who want to learn how processors work and is developed in Java, so it should work on most Operating Systems, if they have a JRE installed. [Gar05]



Main Memory

Byte...	Wor...	ByteVal	WordVal
0	0	0	2
1		0	
2		0	
3		2	
4	1	0	4
5		0	
6		0	
7		4	

Registers

Reg	Reg	Bin Val	Dec...
00000	$0	0..0	0
10000	$s0	0..1000	8
10001	$s1	0..100	4
10010	$s2	0..10	2
10011	$s3	0..100	4
10100	$s4	0..10110	22

Instruction Memory

W...	A...	Instruction	C
0	0	main:add $s3, $s3, $s2	.
100	4	sw $s3, 0($s0)	.
...	8	add $s0, $s0, $s1	.
...	12	beq $s3, $s4, exit	.
...	16	beq $zero, $zero, main	.
...	20	exit:	

#check if reached max

Figure 2.5: ProcSim simulation window [Gar05]

The tool includes several different datapaths and the user can create more, either by writing an eXtensible Markup Language (XML) file or by using the graphical interface provided by the tool [Gar05]. The simulator executes assembly code from a file and also provides a very simple code editor to create assembly programs. The MIPS instructions that the tool can simulate are limited to: *add, sub, and, or, slt, lw, sw, beq, addi, andi, ori, j* [Gar05].

After selecting a datapath (or processor architecture) and the assembly code the simulation can be started, being displayed on the simulation window (see Figure 2.5). The window displays the selected datapath and also other small windows to control the simulation and view the processor registers and memories. The simulation is done using animations. Each instruction is executed automatically or step-by-step, displaying for each instruction the buses and components used and data passed through them, in binary or decimal format.

ProcSim provides a good visualization of the datapath. However, it supports only a small set of MIPS instructions and only one component can send messages at a time during the simulation, displaying the animations sequentially by component, whereas in a real processor the components work concurrently [SCB08]. Furthermore, it doesn't support pipelined datapaths and the source code isn't provided.

2.4 MIPS-Datapath

MIPS-Datapath [GC] was developed by Andrew Gascoyne-Cecil in C++. It is an open-source software under the GNU General Public License[6]. The tool simulates a set of MIPS instructions and displays graphically how the processor datapath executes them. Registers and memory are also shown. The tool webpage [GC] provides executables for Linux and Windows.

The tool can simulate not only a unicycle datapath but also a pipelined one, with or without data forwarding (see Figure 2.6). The instructions are executed step-by-step and the buses used by the selected instruction up to that step are highlighted. The inputs and outputs of each component can be displayed as a tooltip by hovering the mouse cursor over the desired component. The tool provides a simple code editor to create the assembly program and the program's initial data can be set using a table.

MIPS-Datapath allows a person to see how each instruction is executed by the processor but not very well how pipelining in processors work, since the simulator highlights the used buses only for one instruction at a time and not by the whole processor at that time, as the processor can be executing several instructions in different execution steps at the same time. Although the tool supports data forwarding, it doesn't support stalls and can't detect and solve branch hazards. The tool supports only 10 MIPS instructions: *lw, sw, add, addi, and, or, sub, stl, beq, nop*. Also, the inputs and outputs of each component are not clearly shown and one must hover the cursor over the component to show its data. Finally, it's not possible to configure the datapath without editing the code.

[6]GNU General Public License `http://opensource.org/licenses/GPL-3.0`

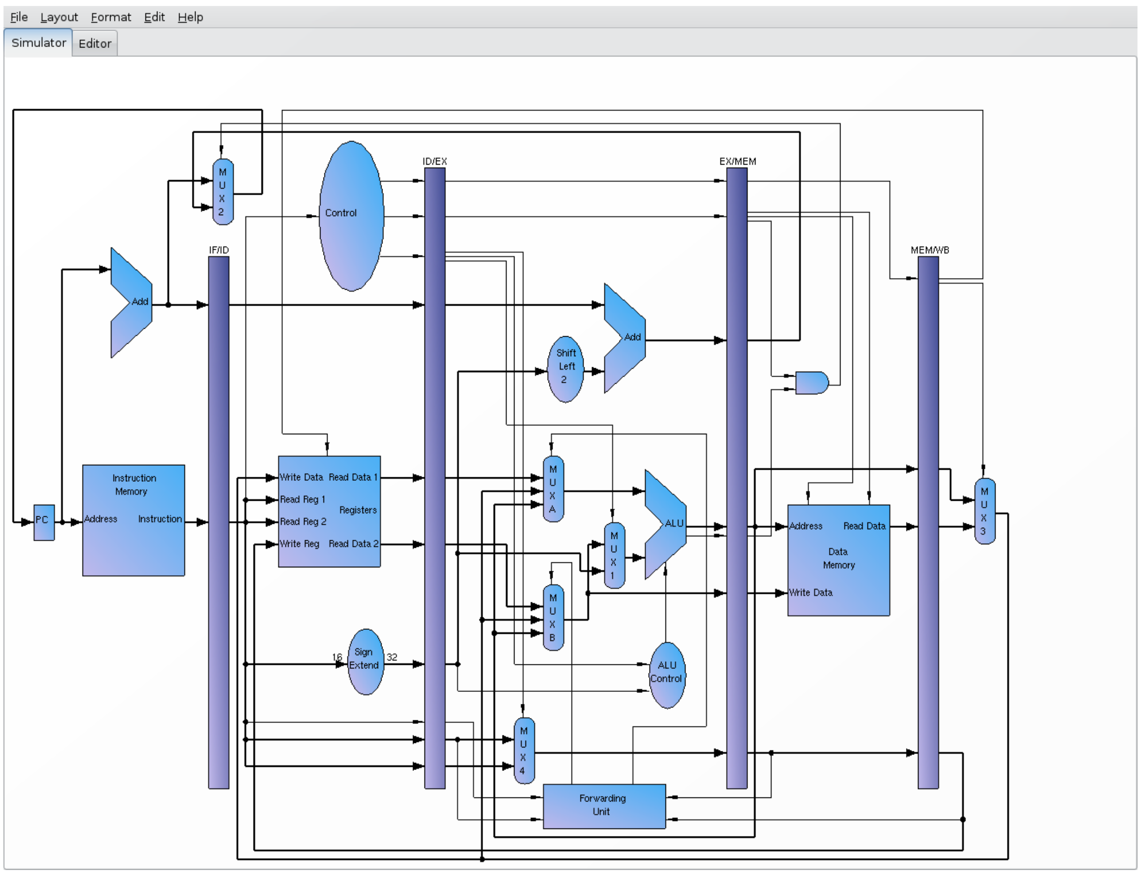


Figure 2.6: MIPS-Datapath simulator [GC]

2.5 WebMIPS

WebMIPS is an educational MIPS simulator that can be executed from a Web browser. It was written in the Active Server Pages (ASP) language and simulates a five-stage pipeline, having been used in an introductory computer architecture course of the Faculty of Information Engineering in Italy [BGM04]. A running version of WebMIPS is available here: `http://www.maiconsoft.com.br/webmips/index.asp`. The source code is also available online.

Being a web application, WebMIPS can be executed from almost any system, provided it has a web browser and is connected to the internet, without requiring the download or installation of any program. The application provides a simple editor to load an assembly program. After loaded, the program can be executed step-by-step or all at once, displaying the number of clock cycles used to execute the program. Pipeline hazards are detected and resolved using data forwarding and stalls, and branch decision is in Instruction Decode phase [BGM04]. Figure 2.7 displays what the WebMIPS window looks like during the execution of a program.

On the left side of the window the list of instructions, registers and data memory addresses are displayed. For each instruction, the resulting machine code, instruction type and instruction fields are shown, and the values of the fields, registers and memory are displayed both in binary and decimal formats. This part of the window also displays what parts of the pipeline are stalled,

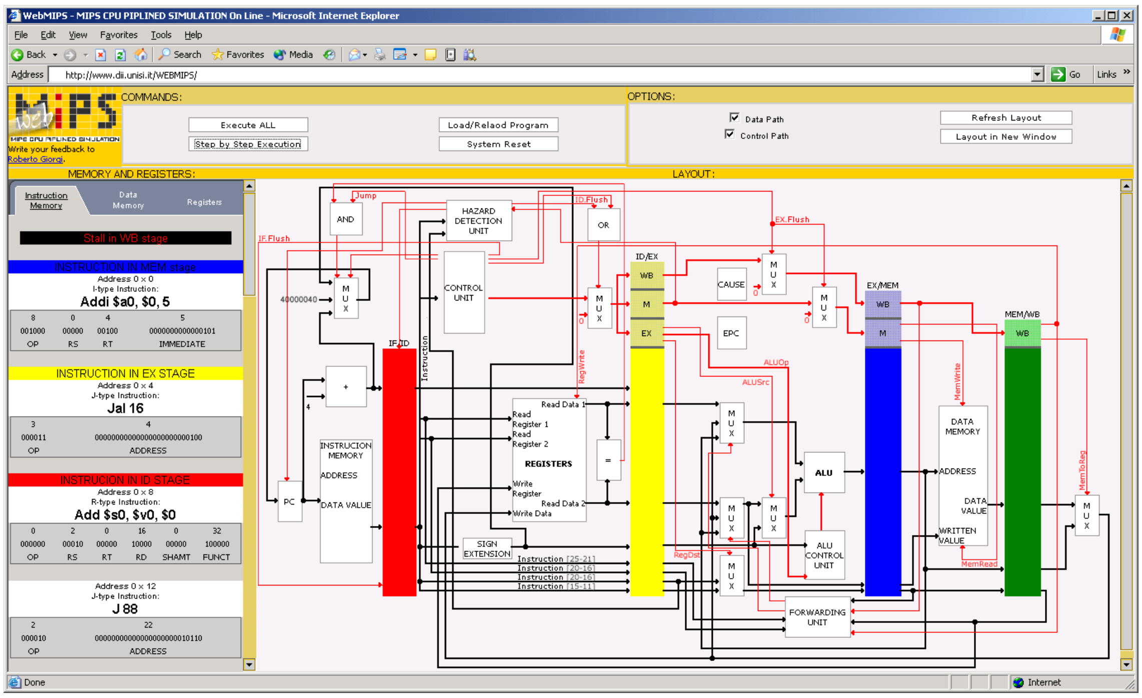


Figure 2.7: WebMIPS window during execution [BGM04]

if any.

The central part of the window displays the MIPS pipeline. The control and data wires can be hidden and the user can click on each individual component of the pipeline, including multiplexers, to view a brief description of the component and its inputs and outputs at that moment. Besides that, the displayed pipeline is static, not highlighting the wires used by each instruction.

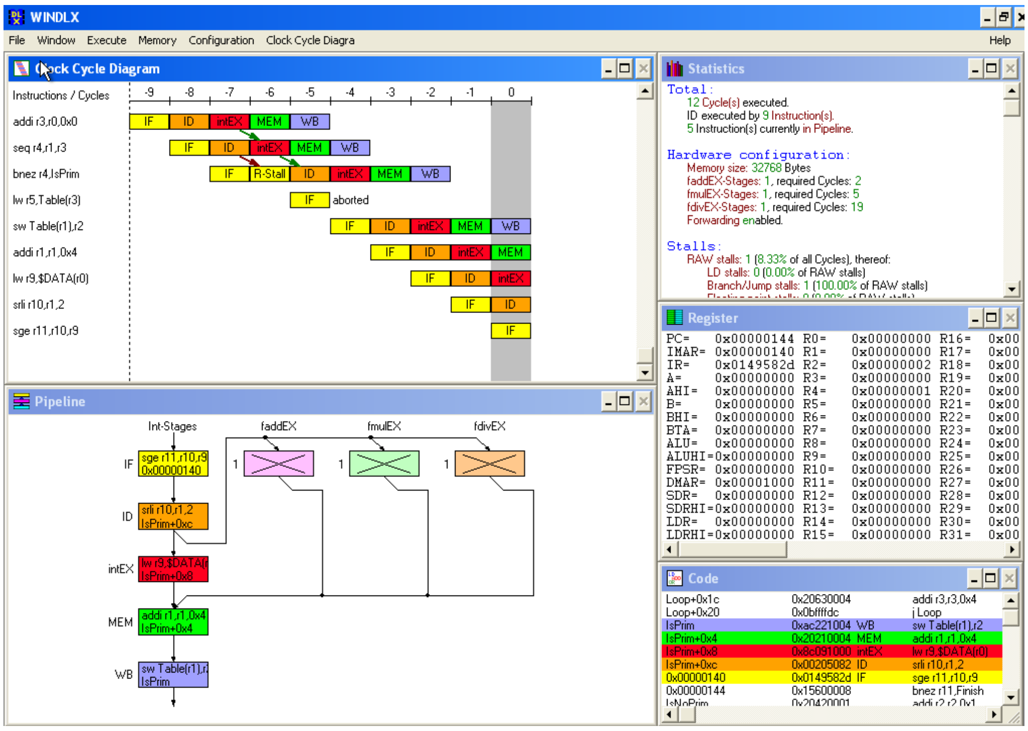
WebMIPS is a good educational MIPS simulator but has some important shortcomings. The application simulates only the pipeline version of the MIPS processor. The representation of the pipeline shown is mostly static and, while the user can click on the components to check their input and output data, it doesn't display the wires used by the instruction nor the data that passes through them without requiring a click. It only distinguishes by color the wires that belong to the data path or to the control path. It also doesn't support floating point instructions and syscalls.

2.6 EduMIPS64

EduMIPS64 is an educational simulator, targeted for Computer Architecture students and developed at the University of Catania in Italy [PSP+12], that runs MIPS64 programs [Tea]. The tool was used in some undergraduate courses to evaluate it and the results were positive, both in terms of percentage of success [PSP+12] and in terms of appreciation from the students [edu].

EduMIPS64 was initially a port of the WinMIPS64 simulator to Java [Tea, SCB08] and has, in fact, a very similar interface. WinMIPS64 [Sco12] is, in turn, a replacement for the WinDLX simulator. Both WinMIPS64 and WinDLX are Windows-only programs, but WinDLX simulates the DLX architecture. As for source code availability, only EduMIPS64 has the code available online.

WinMIPS64's source code, in C++, is available too but only on request [Sco12]. Two screenshots of WinDLX and WinMIPS64 are presented in figures 2.8 and 2.9 respectively. EduMIPS64 is described in more detail below since it is more recent, cross-platform and has the source code available online.



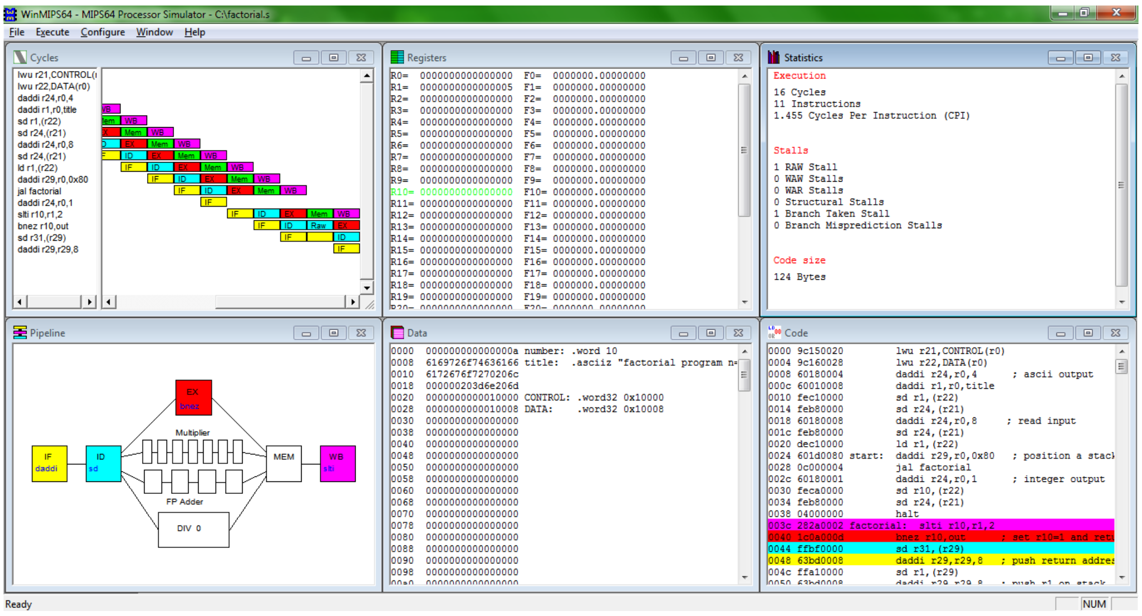
```
Instructions / Cycles   -9    -8    -7      -6     -5    -4    -3    -2    -1    0
addi r3,r0,0x0          IF    ID    intEX   MEM    WB
seq r4,r1,r3                  IF    ID      intEX  MEM   WB
bnez r4,IsPrim                      IF      R-Stall ID   intEX MEM   WB
lw r5,Table(r3)                             IF     aborted
sw Table(r1),r2                                          IF    ID    intEX MEM   WB
addi r1,r1,0x4                                                 IF    ID    intEX MEM
lw r9,$DATA(r0)                                                      IF    ID    intEX
srli r10,r1,2                                                              IF    ID
sge r11,r10,r9                                                                   IF
```

```
Total:
    12 Cycle(s) executed.
    ID executed by 9 Instruction(s).
    5 Instruction(s) currently in Pipeline.

Hardware configuration:
    Memory size: 32768 Bytes
    faddEX-Stages: 1, required Cycles: 2
    fmulEX-Stages: 1, required Cycles: 5
    fdivEX-Stages: 1, required Cycles: 19
    Forwarding enabled.

Stalls:
    RAW stalls: 1 (8.33% of all Cycles), thereof:
        LD stalls: 0 (0.00% of RAW stalls)
        Branch/Jump stalls: 1 (100.00% of RAW stalls)
```

```
PC=    0x00000144 R0=   0x00000000 R16=  0x00
IMAR=  0x00000140 R1=   0x00000000 R17=  0x00
IR=    0x0149582d R2=   0x00000002 R18=  0x00
A=     0x00000000 R3=   0x00000000 R19=  0x00
AHI=   0x00000000 R4=   0x00000001 R20=  0x00
B=     0x00000000 R5=   0x00000000 R21=  0x00
BHI=   0x00000000 R6=   0x00000000 R22=  0x00
BTA=   0x00000000 R7=   0x00000000 R23=  0x00
ALU=   0x00000000 R8=   0x00000000 R24=  0x00
ALUHI=0x00000000 R9=   0x00000000 R25=  0x00
FPSR=  0x00000000 R10=  0x00000000 R26=  0x00
DMAR=  0x00001000 R11=  0x00000000 R27=  0x00
SDR=   0x00000000 R12=  0x00000000 R28=  0x00
SDRHI=0x00000000 R13=  0x00000000 R29=  0x00
LDR=   0x00000000 R14=  0x00000000 R30=  0x00
LDRHI=0x00000000 R15=  0x00000000 R31=  0x00
```

```
Loop+0x1c     0x20630004         addi r3,r3,0x4
Loop+0x20     0x0bffffdc         j Loop
IsPrim        0xac221004  WB     sw Table(r1),r2
IsPrim+0x4    0x20210004  MEM    addi r1,r1,0x4
IsPrim+0x8    0x8c091000  intEX  lw r9,$DATA(r0)
IsPrim+0xc    0x00205082  ID     srli r10,r1,2
0x00000140    0x0149582d  IF     sge r11,r10,r9
0x00000144    0x15600008         bnez r11,Finish
```

Figure 2.8: WinDLX user interface



```
lwu r21,CONTROL(r
lwu r22,DATA(r0)
daddi r24,r0,4
daddi r1,r0,title      WB
sd r1,(r22)            Mem WB
sd r24,(r21)           EX  Mem WB
daddi r24,r0,8         ID  EX  Mem WB
sd r24,(r21)           IF  ID  EX  Mem WB
ld r1,(r22)                IF  ID  EX  Mem WB
daddi r29,r0,0x80              IF  ID  EX  Mem WB
jal factorial                      IF  ID  EX  Mem WB
daddi r24,r0,1                         IF
slti r10,r1,2                              IF  ID  EX  Mem WB
bnez r10,out                                   IF  ID  Raw EX
sd r31,(r29)                                       IF      ID
daddi r29,r29,8                                            IF
```

```
R0=  0000000000000000  F0=  0000000.00000000
R1=  0000000000000005  F1=  0000000.00000000
R2=  0000000000000000  F2=  0000000.00000000
R3=  0000000000000000  F3=  0000000.00000000
R4=  0000000000000000  F4=  0000000.00000000
R5=  0000000000000000  F5=  0000000.00000000
R6=  0000000000000000  F6=  0000000.00000000
R7=  0000000000000000  F7=  0000000.00000000
R8=  0000000000000000  F8=  0000000.00000000
R9=  0000000000000000  F9=  0000000.00000000
R10= 0000000000000000  F10= 0000000.00000000
R11= 0000000000000000  F11= 0000000.00000000
R12= 0000000000000000  F12= 0000000.00000000
R13= 0000000000000000  F13= 0000000.00000000
R14= 0000000000000000  F14= 0000000.00000000
R15= 0000000000000000  F15= 0000000.00000000
R16= 0000000000000000  F16= 0000000.00000000
R17= 0000000000000000  F17= 0000000.00000000
R18= 0000000000000000  F18= 0000000.00000000
R19= 0000000000000000  F19= 0000000.00000000
```

```
Execution

16 Cycles
11 Instructions
1.455 Cycles Per Instruction (CPI)


Stalls

1 RAW Stall
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
1 Branch Taken Stall
0 Branch Misprediction Stalls


Code size

124 Bytes
```

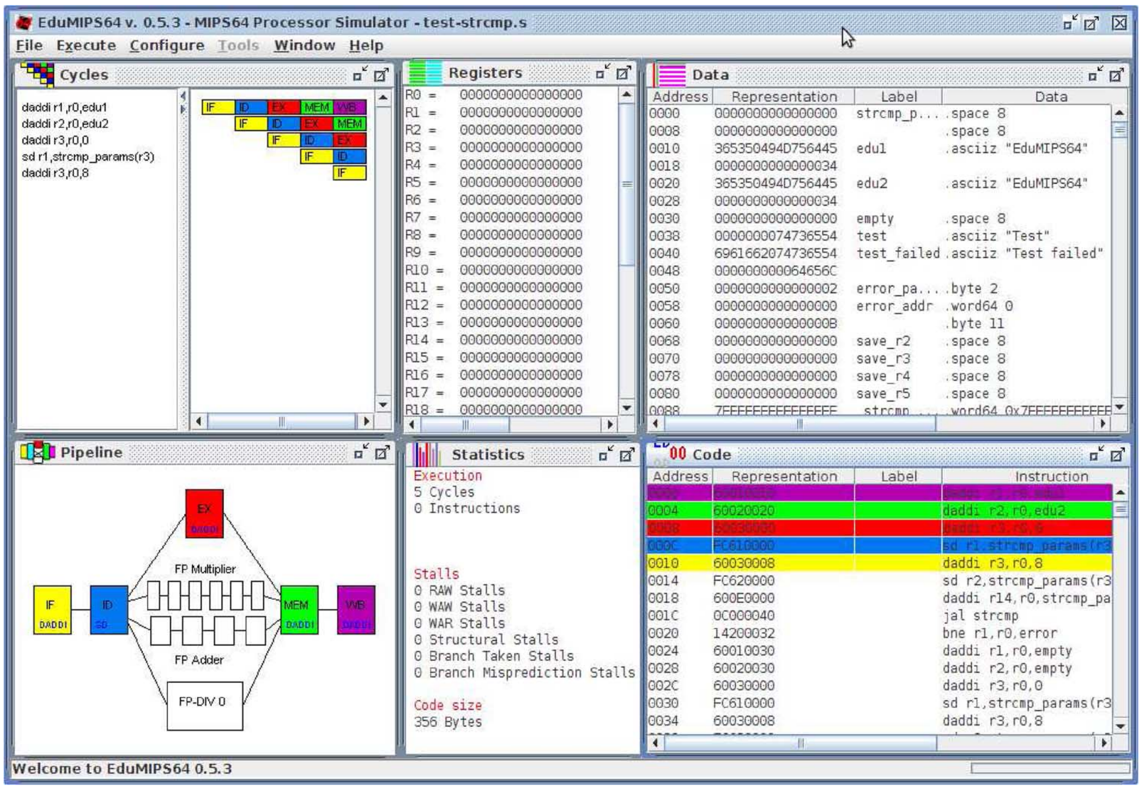
```
0000  000000000000000a number: .word 10
0008  6169726f74636166 title:  .asciiz "factorial program n=
0010  6172676f7270206c
0018  000000203d6e206d
0020  0000000000010000 CONTROL: .word32 0x10000
0028  0000000000010008 DATA:    .word32 0x10008
0030  0000000000000000
0038  0000000000000000
0040  0000000000000000
0048  0000000000000000
0050  0000000000000000
0058  0000000000000000
0060  0000000000000000
0068  0000000000000000
0070  0000000000000000
0078  0000000000000000
0080  0000000000000000
0088  0000000000000000
0090  0000000000000000
0098  0000000000000000
```

```
0000 9c150020            lwu r21,CONTROL(r0)
0004 9c160028            lwu r22,DATA(r0)
0008 60180004            daddi r24,r0,4      ; ascii output
000c 60010008            daddi r1,r0,title
0010 fec10000            sd r1,(r22)
0014 feb80000            sd r24,(r21)
0018 60180008            daddi r24,r0,8      ; read input
001c feb80000            sd r24,(r21)
0020 dec10000            ld r1,(r22)
0024 601d0080 start:     daddi r29,r0,0x80   ; position a stack
0028 0c000004            jal factorial
002c 60180001            daddi r24,r0,1      ; integer output
0030 feca0000            sd r10,(r22)
0034 feb80000            sd r24,(r21)
0038 04000000            halt
003c 282a0002 factorial:  slti r10,r1,2
0040 1c0a000d            bnez r10,out       ; set r10=1 and retu
0044 ffbf0000            sd r31,(r29)
0048 63bd0008            daddi r29,r29,8    ; push return addres
004c ffa10000            sd r1,(r29)
```

Figure 2.9: WinMIPS64 user interface

EduMIPS64 was, as mentioned before, developed in Java and, thus, can be run in almost any operating system. It simulates a 5-stage MIPS64 processor [PSP+12] that includes a floating point unit, and supports a quite reasonable number of assembly instructions, including some syscalls. It also detects hazards, allows the simulation to run with or without data forwarding and allows registers and memory to be edited during the execution.

The tool has an intuitive user interface, as shown in Figure 2.10. No editor is provided, so the program to simulate is loaded from a file created by an external editor. After loaded, the program can be executed step-by-step and the CPU state is shown in several internal windows. These windows display the registers, the memory, the instructions, performance statistics including stalls and Cycles Per Instruction (CPI), a timing diagram of the blocks used by each instruction in each clock cycle and a simple pipeline diagram composed by the CPU's main blocks, including the ones relative to the floating point unit, and displaying the instructions using them. Additionally, each block is represented by a different color.



```
daddi r1,r0,edu1
daddi r2,r0,edu2
daddi r3,r0,0
sd r1,strcmp_params(r3)
daddi r3,r0,8
```

```
R0  =  0000000000000000
R1  =  0000000000000000
R2  =  0000000000000000
R3  =  0000000000000000
R4  =  0000000000000000
R5  =  0000000000000000
R6  =  0000000000000000
R7  =  0000000000000000
R8  =  0000000000000000
R9  =  0000000000000000
R10 =  0000000000000000
R11 =  0000000000000000
R12 =  0000000000000000
R13 =  0000000000000000
R14 =  0000000000000000
R15 =  0000000000000000
R16 =  0000000000000000
R17 =  0000000000000000
R18 =  0000000000000000
```

```
Address  Representation    Label        Data
0000     0000000000000000  strcmp_p.... .space 8
0008     0000000000000000               .space 8
0010     365350494D756445  edu1         .asciiz "EduMIPS64"
0018     0000000000000034
0020     365350494D756445  edu2         .asciiz "EduMIPS64"
0028     0000000000000034
0030     0000000000000000  empty        .space 8
0038     0000000074736554  test         .asciiz "Test"
0040     6961662074736554  test_failed  .asciiz "Test failed"
0048     000000000064656C
0050     0000000000000002  error_pa.... .byte 2
0058     0000000000000000  error_addr   .word64 0
0060     000000000000000B               .byte 11
0068     0000000000000000  save_r2      .space 8
0070     0000000000000000  save_r3      .space 8
0078     0000000000000000  save_r4      .space 8
0080     0000000000000000  save_r5      .space 8
0088     7FFFFFFFFFFFFFFF  strcmp_....  .word64 0x7FFFFFFFFFFF
```

```
Execution
5 Cycles
0 Instructions


Stalls
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
0 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
356 Bytes
```

```
Address  Representation  Label  Instruction
0000     60010010               daddi r1,r0,edu1
0004     60020020               daddi r2,r0,edu2
0008     60030000               daddi r3,r0,0
000C     FC610000               sd r1,strcmp_params(r3
0010     60030008               daddi r3,r0,8
0014     FC620000               sd r2,strcmp_params(r3
0018     600E0000               daddi r14,r0,strcmp_pa
001C     0C000040               jal strcmp
0020     14200032               bne r1,r0,error
0024     60010030               daddi r1,r0,empty
0028     60020030               daddi r2,r0,empty
002C     60030000               daddi r3,r0,0
0030     FC610000               sd r1,strcmp_params(r3
0034     60030008               daddi r3,r0,8
```

Figure 2.10: EduMIPS64 user interface [PSP+12]

EduMIPS64 has a very friendly user interface and, apparently, was successful [edu]. It could be improved, however. The tool doesn't provide a code editor and only supports the pipeline version of the processor. It also doesn't show a detailed datapath representation.

2.7 For Android

For the Android platform, only one MIPS simulator was found in the Google Play store: the Assembly Emulator [Jim13]. This application is recent and very basic, being in its early development stages. A MIPS assembly program can be assembled, executed and, after that, the application displays the last values of the registers. The applications supports a very limited number of instructions. Currently there is only support for MIPS but more assembly languages will be implemented [Jim13].

Another simulator was found but for the M32 processor. This simulator, the M32 Assembly [UG13], allows an M32 assembly program to be executed step-by-step or automatically with a delay between instructions. During the execution, the user can see the state of the processor (registers and output) and what it is doing.

2.8 Concluding Remarks

In this chapter some existing relevant educational simulators, related to the solution proposed in the previous chapter, were discussed, making it clear that, although some of them are great simulators, none are complete or versatile enough to support alone teaching and learning of many subjects in computer architecture courses.

	SPIM	MARS	ProcSim	MIPS-Datapath	WebMIPS	EduMIPS64
Open-source	Yes	Yes	No	Yes	Yes	Yes
Code editor	No	Yes	Yes	Yes	Yes	No
Editor syntax-highlighting	No	Yes	No	No	No	No
Unicycle simulation	Yes	Yes	Yes	Yes	No	No
Pipeline simulation	No	No	No	Partial	Yes	Yes
Floating point support	Yes	Yes	No	No	No	Yes
Syscall support	Yes	Yes	No	No	No	Yes
Edit data during execution	Yes	Yes	No	No	No	Yes
Datapath visualization	No	No	Yes	Yes	Yes	Simple
Datapath configuration	No	No	Yes	No	No	No
Timing diagrams	No	No	No	No	No	Yes
Latencies & critical path	No	No	No	No	No	No
Native Android version	No	No	No	No	No	No
Written in	C++,Qt	Java	Java	C++	ASP	Java

Table 2.1: Comparison of the presented tools

Table 2.1 summarizes the tools presented in this chapter, comparing the features amongst them. The table shows that the simulators usually cannot simulate both the unicycle and the pipeline version of the CPU and that the simulated processor datapath is usually not very configurable. We can also see that the code editors provided by most of them, when provided, are usually very simple without displaying any sort of syntax highlighting. Furthermore, none of them has an Android version and none of them simulate the latencies of the CPU's components and display its critical path. On the other hand, many of these simulators have their source code available online and, additionally, many of them can be executed in all major computer operating systems. One could use more than one of these simulators together to aggregate more features and cover more computer architecture topics, but it would be somewhat unintuitive and cumbersome, requiring students to learn and adapt to several tools. Joining these features in an integrated environment would solve these problems.

A lot more simulators exist, but only a few were presented in this chapter, namely some of the most relevant simulators oriented for computer architecture teaching and learning. Some other simulators not described here are: DIMIPSS [FPC06], EKS-MIPS [Ara10, SAA10], MIPSim [Koc08], MipsIt [Bro02b, Bro02a], WebSimple-MIPS [web08], PS - CAS MIPS [MFN09, MVP09], R10k [JdSGM07], ViSiMIPS [KBH11] and UCO.MIPSIM [dC06, GLPHB08].

As for Android, only one very basic and recent MIPS simulator was found, and none of the presented tools has an Android version. As such, and seeing how this tablet is becoming very popular (see Section 3.2), a version of the developed simulator for Android is something innovative.

The next chapter describes the methodology used in the development of the proposed tool and presents a very brief introduction about the MIPS processor architecture. It also provides an introduction to the following chapters, where the implementation is detailed.

Chapter 3

Requirements and Development Methodology

This chapter presents the methodology used in the development of this work. First, a very brief overview of the MIPS processor architecture is presented. Then, the technologies used are discussed. Next, the development process is detailed. Finally, an overview of the developed tool's requirements and features is given, and the general structure of the code is explained.

3.1 MIPS

In this section a small overview of the MIPS processor architecture is presented, focusing on the details that are important for this dissertation. For a more in-depth description and explanation of the MIPS architecture it is recommended to read the referenced book "Computer Organization and Design – The Hardware/Software Interface" by David Patterson and John Hennessy [PH05] or any other similar book.

MIPS Computer Systems, founded in 1984, was one of the pioneers in Reduced Instruction Set Computer (RISC) CPU development with the creation of the MIPS microprocessor architecture. The company was bought by Silicon Graphics Inc. in 1992 and is now known as MIPS Technologies Inc. [Per09].

RISC processors, as opposed to the Complex Instruction Set Computer (CISC) processors, contain fewer and simpler instructions, all with the same size and, ideally, with a number of CPI of 1 and a lower amount of addressing modes and formats [Jam95]. They also include a larger number of registers to avoid frequent accesses to the memory [Per09].

The MIPS processor architecture is widely used for computer architecture teaching by faculties and universities around the world [Per09], like FEUP. The architecture has several versions. The most used ones for teaching are the unicycle and the pipelined versions. In the unicycle version, each instruction is executed in a single clock cycle. This means that, as the clock frequency is fixed, the clock period must be no shorter that the time of the longest instruction, which usually is the load instruction [Per09]. To solve that performance problem a pipeline was introduced. In the pipelined version, each instruction is divided into five stages [PH05]:

Instruction fetch (IF) Fetch the instruction from memory and compute the address of the next sequential instruction.

Instruction decode (ID) Determine which instruction it is and read the registers indicated in the instruction.

Execution (EX) Necessary calculations executed by the Arithmetic and Logic Unit (ALU).

Memory (MEM) The memory can be read from or written to in this step.

Write back (WB) Complete the instruction by writing the result on a register.

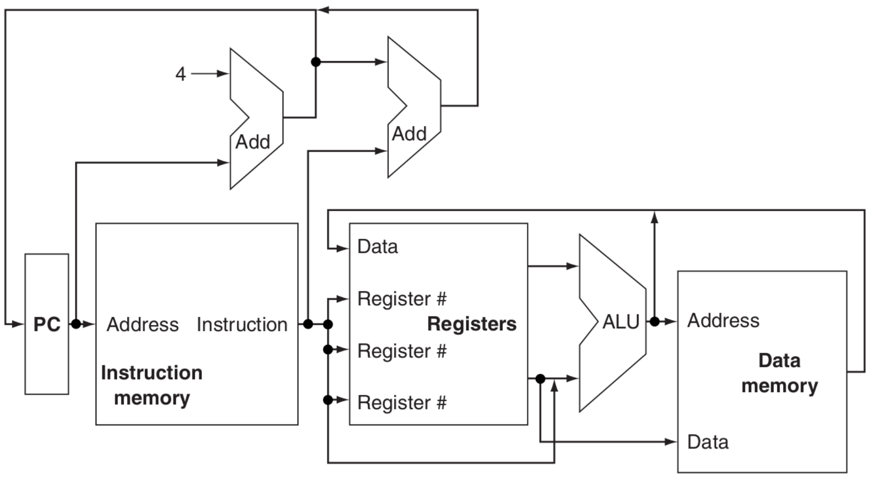


Figure 3.1: MIPS basic datapath [PH05, p. 287]

The datapaths of the unicycle and pipeline versions of MIPS are shown in figures 3.1 and 3.2 respectively. These are very simple representations of them with very little detail, omitting many things like, for example, the control path. A datapath is composed by several components connected together by buses. The major components, or blocks, of the MIPS CPU can be identified in both versions of the datapath: the program counter (PC), the instruction memory, the registers, the ALU and the data memory. The representation of the pipeline datapath (Figure 3.2) also indicates how the blocks are divided by the five stages mentioned before. The pipelining technique is briefly explained next.

Pipelining is an implementation technique in which multiple instructions are overlapped in execution, like an assembly line [PH05]. In the MIPS case, the five execution steps are executed in different clock cycles and the clock period is set to at least the duration of the longest step, which usually is the memory access. The sequential instructions are then overlapped when executed: in the first cycle the first instruction is loaded in the IF stage, in the second cycle the first instruction advances to the ID stage while the second instruction is loaded in the IF stage, and so on (see

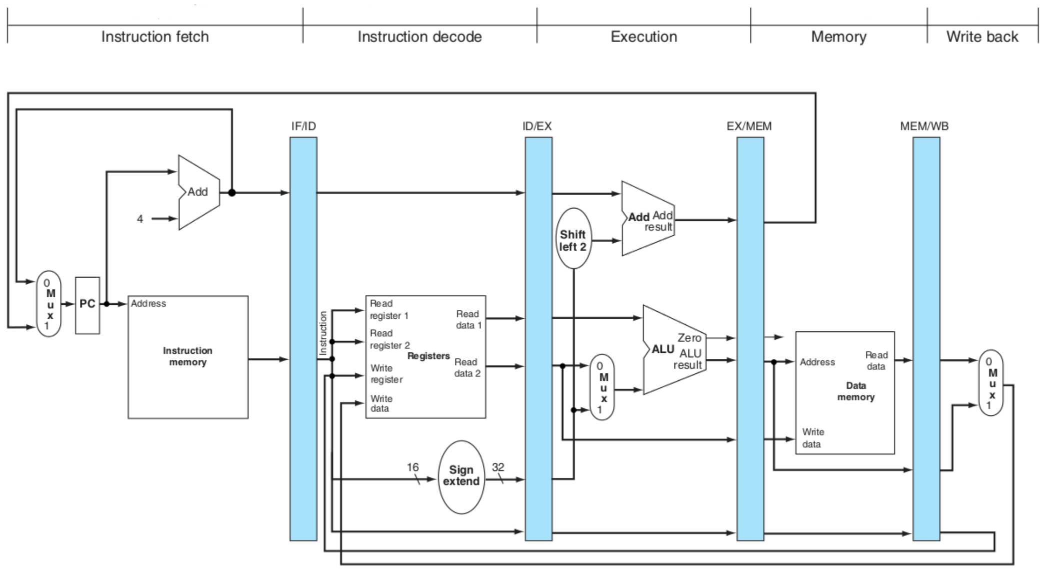
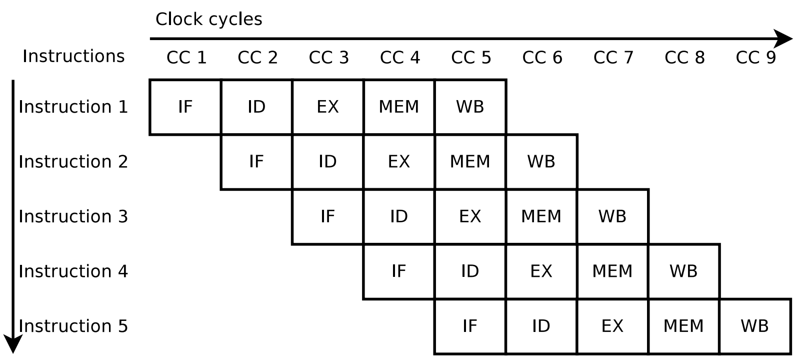


Figure 3.2: MIPS simplified pipelined datapath [PH05, p. 398]

Figure 3.3). This greatly increases the number of instructions executed per second. However, some problems arise with the use of a pipeline, like data hazards[1] and control hazards[2]. These problems are solved by the use of methods like forwarding[3] and stalls[4].



Instructions	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Instruction 1	IF	ID	EX	MEM	WB				
Instruction 2		IF	ID	EX	MEM	WB			
Instruction 3			IF	ID	EX	MEM	WB		
Instruction 4				IF	ID	EX	MEM	WB	
Instruction 5					IF	ID	EX	MEM	WB

Clock cycles

Figure 3.3: Timing diagram of 5 instructions in an ideal MIPS pipeline (based on [PH05])

[1]Data hazard: Also called pipeline data hazard. An occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available. [PH05]

[2]Control hazard: Also called branch hazard. An occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected. [PH05]

[3]Forwarding: also called bypassing. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory. [PH05]

[4]Stall: also called bubble. A stall initiated in order to resolve a hazard. [PH05]

3.2 Technology and Tools

As mentioned in the previous chapters of this document, the objective of this dissertation was to develop a simulator of the MIPS processor to help students to learn about computer architecture. A considerable knowledge about this architecture was therefore needed.

The tool was developed for the PC using the Java language. Java [Ora], being a cross-platform language, allows the tool to run on most operating systems, like Windows, Linux and Mac OS. The only requirement to run Java applications is the presence of a JRE installed in the computer. The Graphical User Interface (GUI) was created using Swing and the chosen IDE was NetBeans [Ora13], as it simplifies the creation of Java graphical interfaces. Some external libraries were also used for the user interface, namely: RSyntaxTextArea [Fif13b] and AutoComplete [Fif13a] for the code editor and JTattoo [Hag] for the light and dark themes.

A similar version of the tool was also developed for touch screen Android devices, especially tablets. The tablet is a device that is gaining a lot of popularity among consumers [Shi12], and Android [Gooa] is one of the most popular operating systems for not only tablets but also smartphones [But11]. Even though the Android version was targeted especially for tablets, it also runs fairly well on smartphones. It is important to note that Android applications are developed in Java, so the choice of using Java for the PC version made the creation of the Android version a lot easier. The IDE used to develop the Android version was Eclipse [Fou13], as it is the one recommended by Google [Goob]. The Android version was developed because the system is popular and, as discussed in Section 2.7, almost no MIPS simulators exist for Android platforms. Moreover, Android applications can be developed on any of the major computer operating systems without any cost.

The different CPUs and their instructions are configurable and stored in JavaScript Object Notation (JSON) files. JSON [jsoa] is a file format that can be parsed easily in Java through a library. It is also easy for humans to read and write, and creates smaller files that other formats like XML. The chosen library to parse the JSON files was the standard *json.org* library [jsob], which is already built-in in Android.

As for version control, git [Con13] was used, as it is decentralized, allowing the versioning to be done offline. The repository containing the project's source code was hosted in a private Bitbucket [Atl] repository, as academic users have access to all of Bitbucket features.

3.3 Development Process

The initial work plan was very general and merely indicative. The development would start by the requirements specifications followed by the interface and interaction specification, then the tool would be implemented and finally the documentation and the final report would be written. The implementation part was further divided into smaller tasks: representation of the CPU, execution simulation, graphical interface for the PC version, CPU graphical editor and graphical interface for the Android version. This plan was also very ambitious. In reality the followed process was very different.

The real development process was very informal and agile. New versions of the simulator were released for the supervisors to see every week and meetings were made with them every two or three weeks. No formal requirements or interface specifications were made, thus maximizing the time available for the implementation of the tool. The progress was tracked using a list of tasks and their individual progress.

The development started by creating the foundations to represent the CPU, the components and their inputs and outputs. Not long after that, the GUI for the PC version was started, serving as a working prototype. By the time the unicycle MIPS CPU was finished, the development of the Android version had started. The development of the user interfaces was done in parallel with the development of the simulation logic. However, the development of the simulation logic was given a higher priority than the development of the GUI until the final weeks, when the user interface was given more attention and made more user-friendly.

3.4 Requirements and Code Structure

The developed simulator, DrMIPS, lets the user create or load an existing assembly program and then simulate its execution on the 32 bit MIPS CPU while visualizing what happens inside the processor. The simulation can be executed step-by-step or the full program at once, and the user can also undo steps (i.e. back step). In each step, the user can see the contents of the registers, the data memory and, more importantly, the composition and state of the unicycle or 5-stage pipeline datapath. It also shows the values at each input and output of each component, which wires are relevant for the execution of the current instruction and, in the case of the pipelined datapath, what instructions are in each stage. Besides seeing how the data flows in the datapath, the user can also view the latencies of the components and the critical path of the CPU using the "performance mode" and experiment how changes in the latencies can alter the critical path as well.

DrMIPS provides several different MIPS CPU configurations, based on [PH05], including the unicycle datapath, with some simplified variants, and the pipelined datapath, with or without hazard detection and resolution. These CPUs can be created and configured by specifying in a file all the components and their properties and the wires connecting them. But, besides the datapath, the instruction sets used by them can also be configured and new instructions can be created, by specifying the properties of the different instruction types, instructions and pseudo-instructions and what they do. The initial requirements and objectives of this work were presented in Section 1.2.

The simulator was implemented for the PC and for Android devices. For that reason, and to ease the development of both versions, the code was divided in two parts: the simulation logic and the user interface. With this division, only the user interface part is dependent on the platform (PC or Android), while the simulation logic part is exactly the same on both platforms. Figure 3.4 shows a simplified Unified Modeling Language (UML) class diagram of the simulator.

The code consists of the following Java packages:

- `mips`: contains the simulation logic.
  - `components`: inner package that defines all the types of components.
- `gui`: contains the platform dependent user interface.
- `util`: contains some utility classes.
- `exceptions`: contains the exception classes.

The `gui` package, where the user interface is defined, is the only package that differs from both platforms. The other packages are all platform independent and are shared by both platforms. The implementation of the simulation logic and the user interfaces is detailed in chapters 4 and 5.

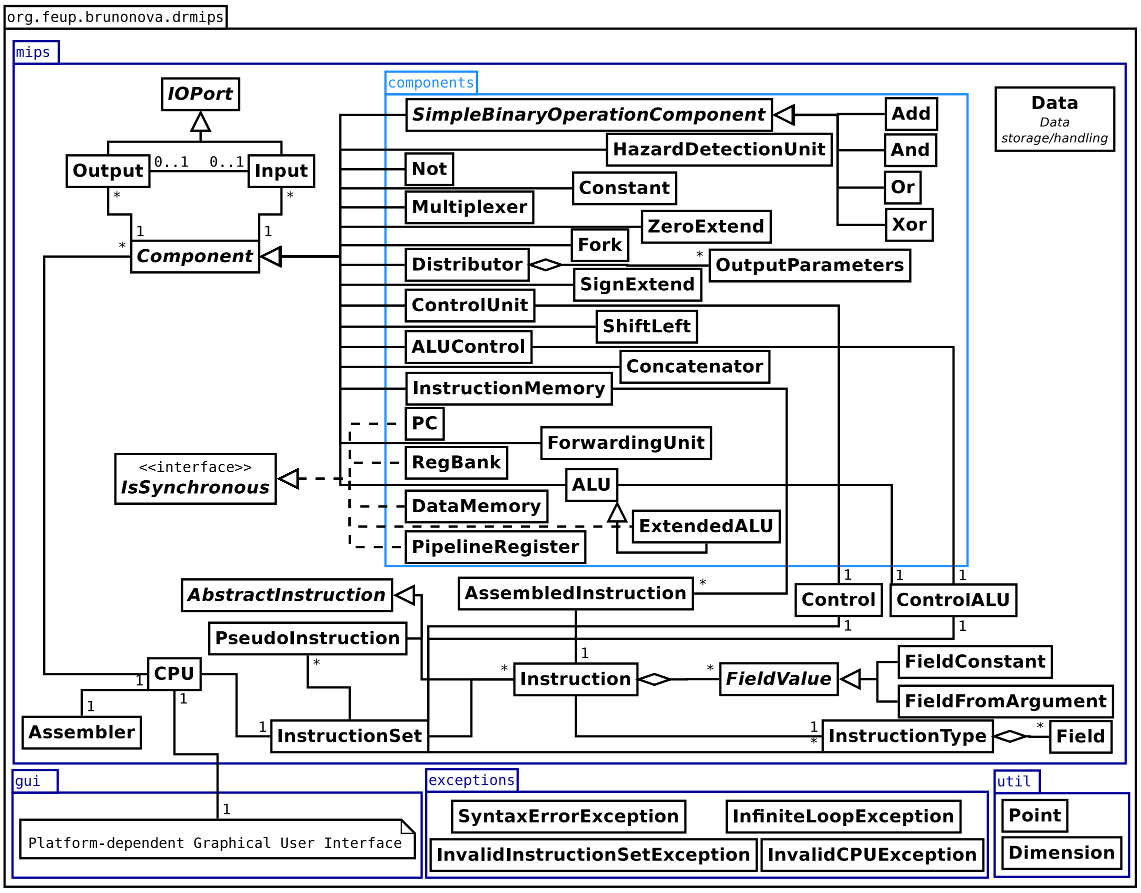


Figure 3.4: Simulator UML class diagram

3.5 Concluding Remarks

This chapter presented an overview of the MIPS architecture, the requirements and features of the tool and the methodology used in its development. Next chapter describes how the internal simulation logic was implemented, while Chapter 5 describes how the graphical interfaces for both PC and Android were implemented.

Chapter 4

Simulation Logic Implementation

This chapter describes how the MIPS processor was represented and how the simulation logic was implemented. As discussed in the previous chapters, the simulator, DrMIPS, was developed for both computers and Android devices and, to ease the development of both versions, the internal simulation logic was coded in a way that didn't make it dependent on the platform.

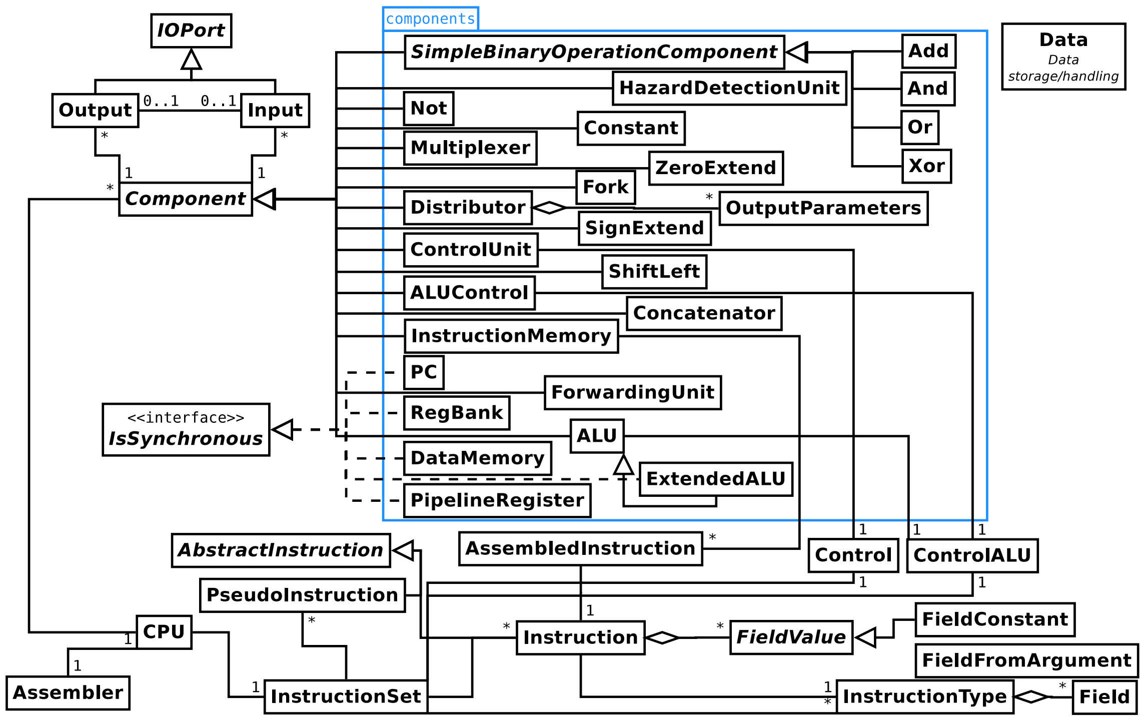


Figure 4.1: UML class diagram of the simulation logic

Figure 4.1 shows a simplified UML class diagram of the simulation logic code. This corresponds to a portion of the UML diagram shown in Figure 3.4 and is in the `mips` Java package. As can be seen in the diagram, this package defines the CPU and all its components, inputs, outputs, supported instructions and pseudo-instructions and assembler. It also defines the `Data` class, which is an abstraction on top of the integer data type where the values have a size in bits. When a value is assigned to an instance of this class, all the bits that don't fit in the specified size are zeroed. It also provides easy access to the value's binary, octal and hexadecimal representations. The bits in the binary representation are grouped in groups of 4 bits to facilitate reading. This class is used almost everywhere so, to avoid complicating the UML diagram, it appears isolated.

4.1 CPU Definition

Each CPU is defined in a JSON file. A CPU file lists the components and their properties, the wires connecting the components, the "friendly" names of each register and the used instruction set (described in the next section). A snippet of the definition of a CPU file is shown in Listing 4.1. Most components and wires were omitted for brevity and replaced by ellipsis.

```
{
  "components": {
    "MUX_DST": {"type": "mux", "x": 205, "y": 260, "size": 5, "sel": "RegDst", "out": "OUT", "in": ["0", "1"], "desc": {"default": "Selects rt or rd as the destination register.", "pt": "Seleciona o rt ou rd como registo de destino."}},
    ...
  },
  "wires": {
    {"from": "DIST_INST", "out": "15-11", "to": "MUX_DST", "in": "1", "start": {"x": 185, "y": 270}, "points": [{"x": 195, "y": 270}, {"x": 195, "y": 282}]},
    {"from": "MUX_DST", "out": "OUT", "to": "REG", "in": "WriteReg", "end": {"x": 250, "y": 277}},
    ...
  },
  "reg_names": ["zero", "at", "v0", "v1", "a0", "a1", "a2", "a3", "t0", "t1", "t2", "t3", "t4", "t5", "t6", "t7", "s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "t8", "t9", "k0", "k1", "gp", "sp", "fp", "ra"],
  "instructions": "default.set"
}
```

Listing 4.1: Snippet of the definition of a CPU file

The different sections that compose the CPU file are:

components Defines the properties of each component, such as the type, the graphical position, the identifiers of the inputs and outputs and possibly a custom description for the various languages. Most of these properties are different for each type of component and each component is identified by a unique identifier.
The component in this example (Listing 4.1) is a multiplexer. The properties `sel`, `in` and `out` define the identifiers of the inputs and outputs of the multiplexer and `size` defines its data size in bits. The other properties are common to all components.

wires Defines all the wires. Each wire connects an output of a component to an input of another component. The position of the input or output on the component is calculated automatically but, as it is not always the most adequate, it can be set manually in the definition of the wire. Intermediate points can also be specified.

reg_names Is optional and defines the "friendly" names of all registers, without including the dollar sign (`$`).

instructions Specifies the file that defines the supported instruction set (see Section 4.2.1).

The CPU JSON files are loaded and parsed by the `CPU` class. The `CPU` class is the central part of the simulator (see Figure 4.1), serving as the interface between the MIPS CPU and the GUI. A `CPU` contains all its components, instruction set and assembler. Furthermore, it controls the simulation of assembly programs, calculates the accumulated latencies of the components and determines the CPU's critical path and control path.

A CPU is comprised of several components connected together by wires. The `Component` abstract class is the base class for all the components and declares all the properties that are common to all components, like the name, position and size. It also provides methods to add and access the inputs and outputs of the component. Each component must extend from this class, set the required properties and implement `execute()`, where its behaviour is defined, using the values of the inputs to set the correct values of the outputs. The CPU's specific components are stored in a map, more specifically a `TreeMap`, indexing each component by its identifier. The inputs and outputs of each component are also stored in a `TreeMap`. Every type of component was defined in the `components` package, as show in Figure 4.1.

The components' inputs and outputs are represented by the `Input` and `Output` classes, respectively. These classes are very similar so, to avoid repeating code, the `IOPort` superclass that defines this similar behaviour was created. The sizes and values of the inputs and outputs are defined using the `Data` class explained earlier. Each output can be, and usually is, connected to an input, and vice-versa, but only if they both have the same size. This is handled by these classes and represents a wire. Each input/output holds a reference to the component it belongs to and also holds a reference to the input/output it is connected to. In other words, each component has access to every component it is connected to and, therefore, the CPU can be represented as a graph.

The inputs and outputs are "attached" to one of the sides of the component it belongs to. Inputs are usually shown on the left side of the component while the outputs are usually on the right, although each type of component can assign each input/output to any of the four sides. Graphically they must have a position with `x` and `y` coordinates. These coordinates can be specified in the CPU JSON file but, if not specified, the default ones are used. When the default coordinates are used, the inputs and outputs in each side are spaced evenly and ordered alphabetically by their identifiers[1]. The distance between the corner of the component to the default position of an input/output along the side it is assigned to is shown in Equation 4.1, where L corresponds to the length of that side

[1]The inputs and outputs are ordered alphabetically because they are stored in a `TreeMap`, which is a data structure that keeps its elements ordered.

of the component, N to the number of inputs and outputs in that side and i to the zero-based index of the input/output relative to the ordered list of inputs and outputs in that side.

$$d = \frac{L}{(N+1) \times (i+1)} \tag{4.1}$$

Some of the components are synchronous, like the program counter and the register bank. These components have an internal state that can be changed only during the clock transition. In terms of code, these components implement the `IsSynchronous` interface, and must then implement the `executeSynchronous()` method, where the synchronous behaviour that changes the component's internal state is defined. Furthermore, to allow the user to return to previous states during execution (i.e. back step), the internal states must be saved in each clock cycle in, for example, a stack and some additional methods to save and restore these states must be implemented.

Concerning accumulated latencies and critical path, they are calculated when the CPU is loaded and when the latency of a component is changed by the user. Each component has its individual latency, which can be zero, and an accumulated latency[2]. Each input also stores its accumulated latency. The accumulated latency of the component corresponds to the highest accumulated latency of the component's inputs plus the component's own latency. The calculation of the accumulated latencies starts on the synchronous components and ends on the inputs that are only used by synchronous behaviours (like the `WriteData` input of the register bank). This is done by propagating forward the accumulated latencies from each component to its connected components recursively. The critical path is, finally, calculated backwards from the input or inputs with the highest accumulated latency. It is worth noting that a CPU can have multiple critical paths.

Regarding the pipeline version of the MIPS CPU, it must have exactly five stages and, thus, four pipeline registers that separate them. These registers, and the program counter, are used to determine what instructions are in each stage of the pipeline by checking their `Write` and `Flush` control signals. This is necessary due to the hazards that can occur. Both an hazard detection unit and a forwarding unit were implemented, and their behaviours were based on [PH05].

The implemented datapaths are presented in the next subsections. Note that the datapaths are configurable and more can be created without changing the code, unless more features are required.

4.1.1 Implemented Unicycle Datapaths

The implemented unicycle version of the MIPS processor is shown in Figure 4.2 and was based on the reference book [PH05, p. 314]. The datapath is very similar to the one in the book and supports the instructions that are indicated on the page 285 of the book, plus a few more instructions and pseudo-instructions. The instructions and pseudo-instructions supported by this datapath's instruction set, *default*, are listed in Subsection 4.2.1.

[2]The necessary time for the component to produce the correct outputs since the start of a clock cycle.

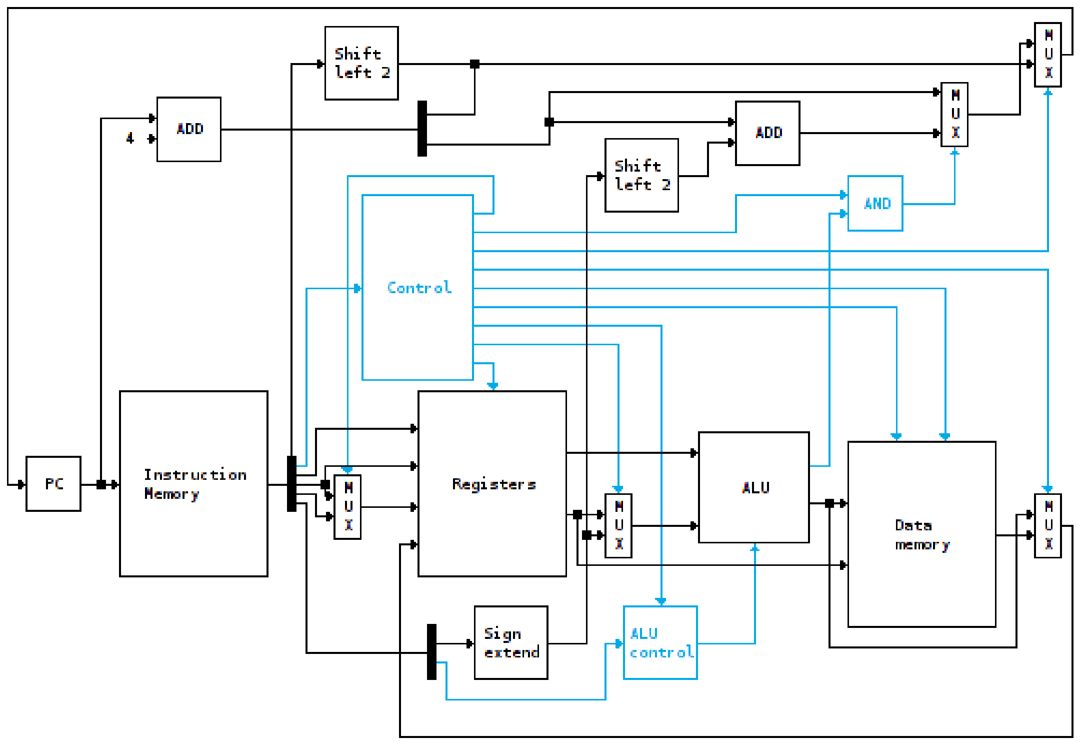


Figure 4.2: Most complete unicycle datapath

Two simpler variations of the datapath were also created. One, shown in Figure 4.3 and based on [PH05, p. 307], which is a version that doesn't support jumps, and the other, shown in Figure 4.4, that is even simpler by also not supporting branches. These variations were created so that the students can start learning with a simpler datapath.

An "extended" datapath was also created. This datapath is very similar to the one in Figure 4.2 but the ALU was replaced by an "extended" ALU[3] and the supported instruction set was extended. This extended instruction set is explained in Subsection 4.2.1. The extended ALU and instruction set could have been added to the default unicycle datapath (Figure 4.2), but that datapath was made to be as similar as possible to the one detailed in the book [PH05].

4.1.2 Implemented Pipeline Datapaths

The implemented pipeline version of the MIPS processor is shown in Figure 4.5 and was based on the reference book [PH05, p. 427]. It has both a forwarding unit and an hazard detection unit. The datapath is similar to the one in the book but has some differences in the branches and stalls. The branch decision is in the `MEM` stage, unlike the final pipeline datapath of the book where it is in the `ID` stage. This is because the book doesn't explain in detail how the implementation of the branch decision in the `ID` stage is done and only gives hints. Additional forwarding logic and

[3]The extended ALU is a synchronous ALU that supports multiplications and divisions and stores the `hi` and `lo` "registers". In reality, multiplications and divisions are not performed by the ALU but here, for simplicity, they were aggregated in this "extended" ALU.

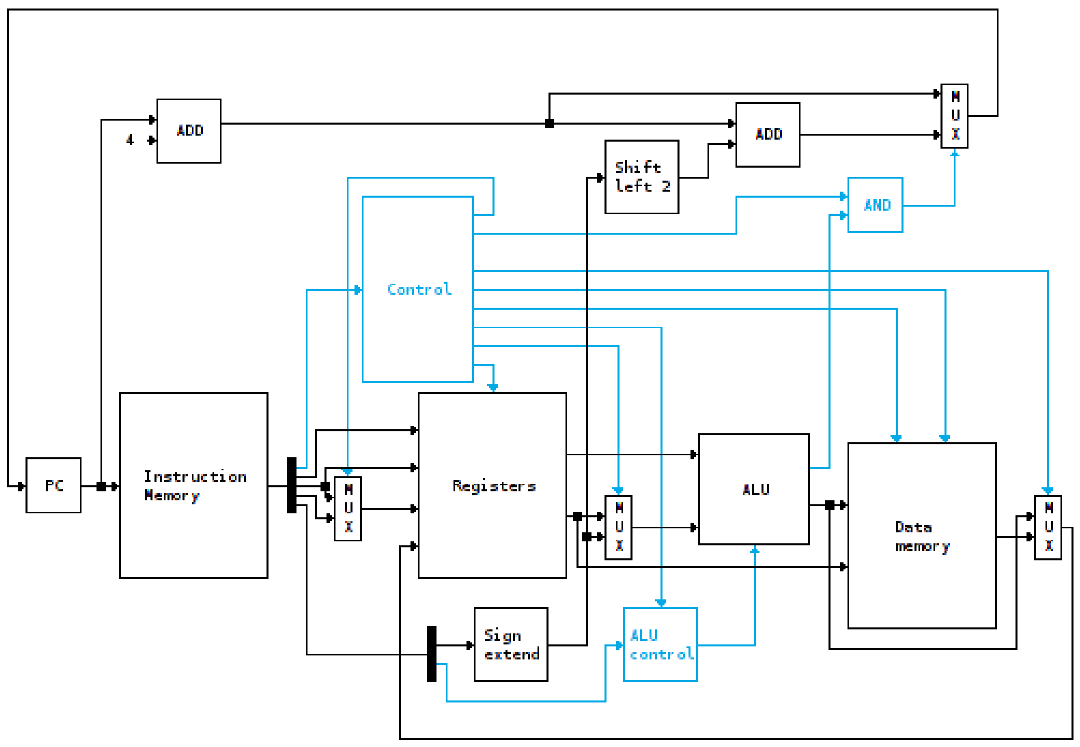


Figure 4.3: Unicycle datapath variant that doesn't support jumps

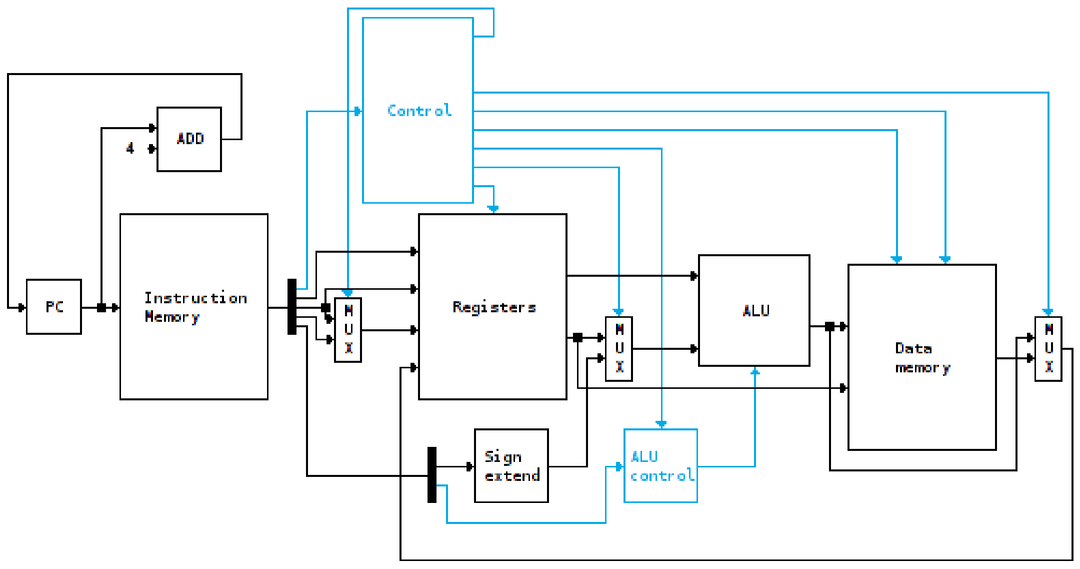


Figure 4.4: Unicycle datapath variant that doesn't support jumps nor branches

stalls would also be required [PH05, p. 419], but they are also not explained. The hazard detection unit has only one output and "bubbles" are introduced in the EX stage by setting the Flush input of the ID/EX pipeline register to 1 instead of zeroing all control signals in the ID stage with a multiplexer. In the pipeline version of the CPU the register bank includes internal forwarding to avoid a data hazard when reading and writing to the same register[4] as indicated in [PH05, p. 410]. These pipelined datapaths don't support the j instruction.

Two variations of the pipeline datapath were also created. One, shown in Figure 4.6 and based on [PH05, p. 416], supports data forwarding but not stalls, while the other, shown in Figure 4.7 and based on [PH05, p. 404], doesn't detect nor solve any pipeline hazard. These variations allow students to understand why forwarding and stalls are necessary.

Finally, just like there is an extended unicycle datapath, as explained in the previous subsection, there is also an extended pipeline datapath that supports a larger number of instructions.

4.2 Instruction Set Definition

The main class that defines the instruction set is `InstructionSet`, and is accessible from the `CPU` (see Figure 4.1). This class loads and parses the instruction set from the file specified in the CPU file, and grants access to all of the instruction types, instructions, pseudo-instructions and control definitions.

An instruction set has several different instructions. In the MIPS case, which is a RISC CPU, all of the instructions have the same size. Each instruction belongs to a type and, on the MIPS, that means one of R, I or J. The 32 bits that comprise the instruction code are split in fields. The fields are different for each type, except the *op code* field, which is always the first field and has always the same size. `InstructionType` is the class that represents this information. The instruction types and their fields are presented in Table 4.1.

Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address/immediate		
J-type	op	target address				

Table 4.1: The MIPS instruction types (based on [PH05, p. 104])

The individual instructions are represented by the `Instruction` class. Additionally, the instruction set can also define pseudo-instructions[5], represented by the `PseudoInstruction` class. Both instructions and pseudo-instructions have mnemonics, arguments and short symbolic descriptions that can be view by the user. As both classes are similar in those aspects, they both

[4] A data hazard can occur in the register bank when the register being read in the ID stage is the same register being written to in the WB stage and RegWrite is 1. When this happens, and if the register bank is configured to use internal forwarding in the CPU file, the value of the WriteData input is forwarded to the one of the outputs.

[5] Pseudo-instructions: These instructions are accepted by the MIPS assembler, even though they are not real instructions within the MIPS instruction set. Instead, the assembler translates them into sequences of real instructions.

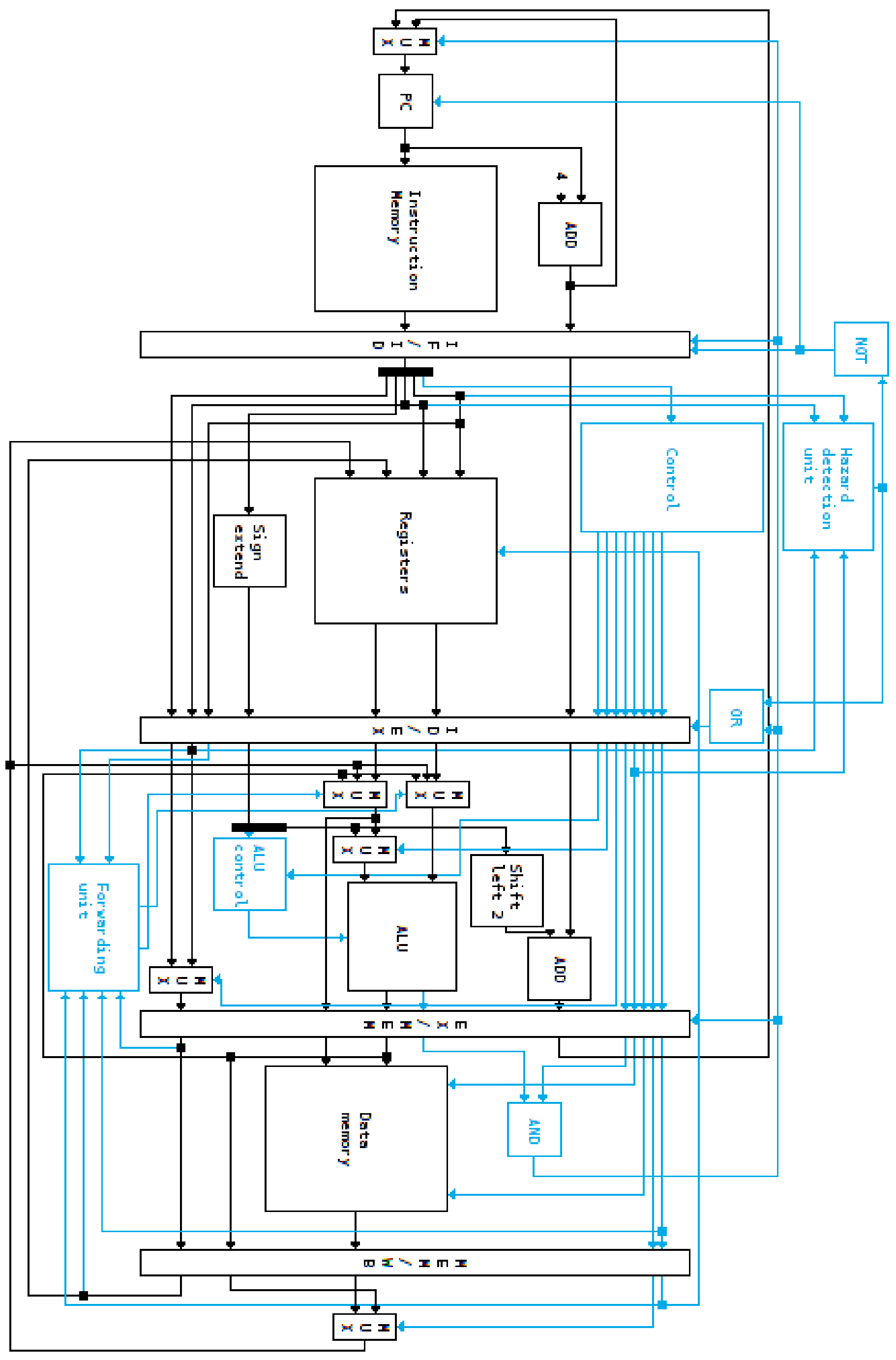


Figure 4.5: Most complete pipeline datapath

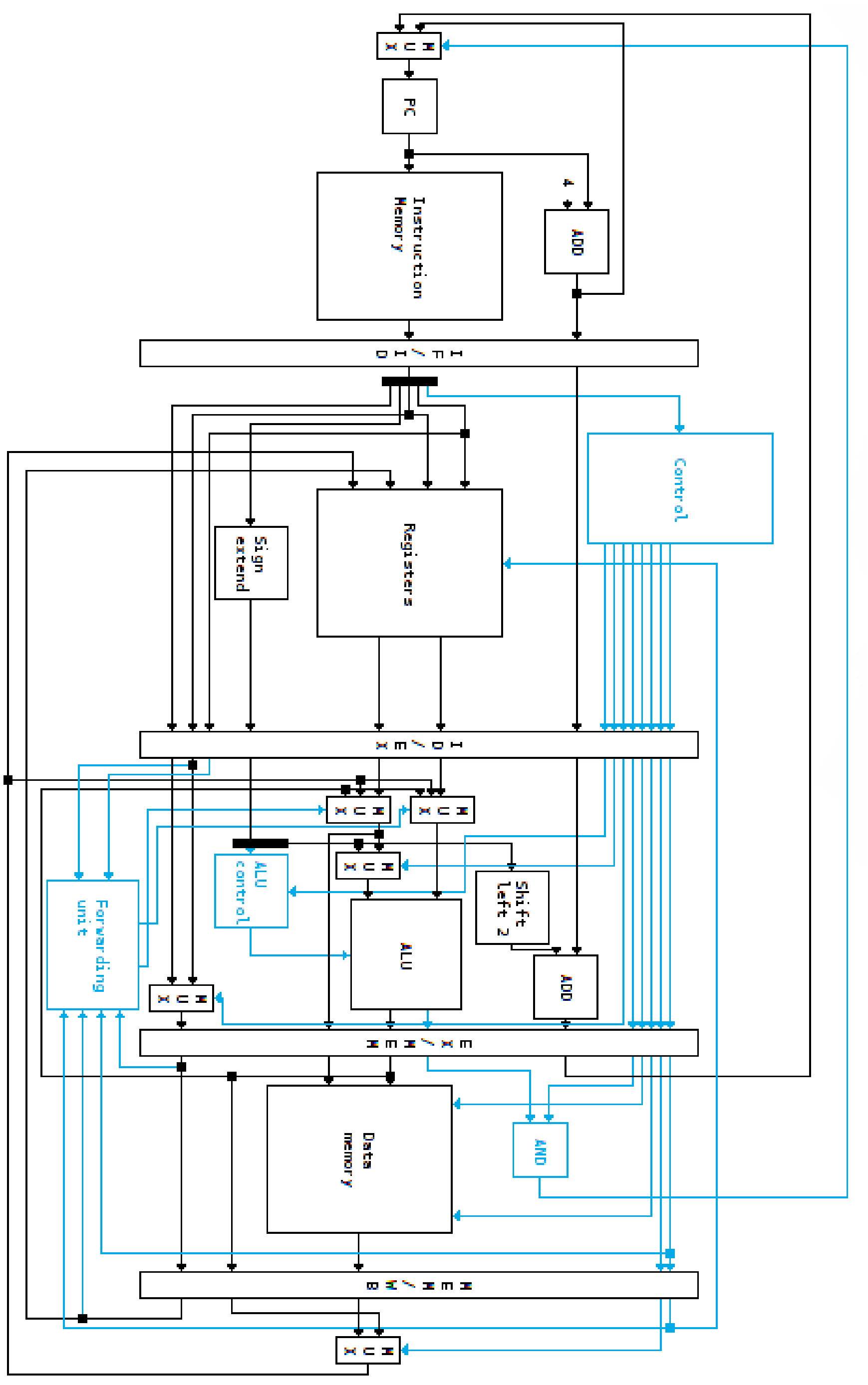


Figure 4.6: Pipeline datapath variant that implements forwarding but not stalls

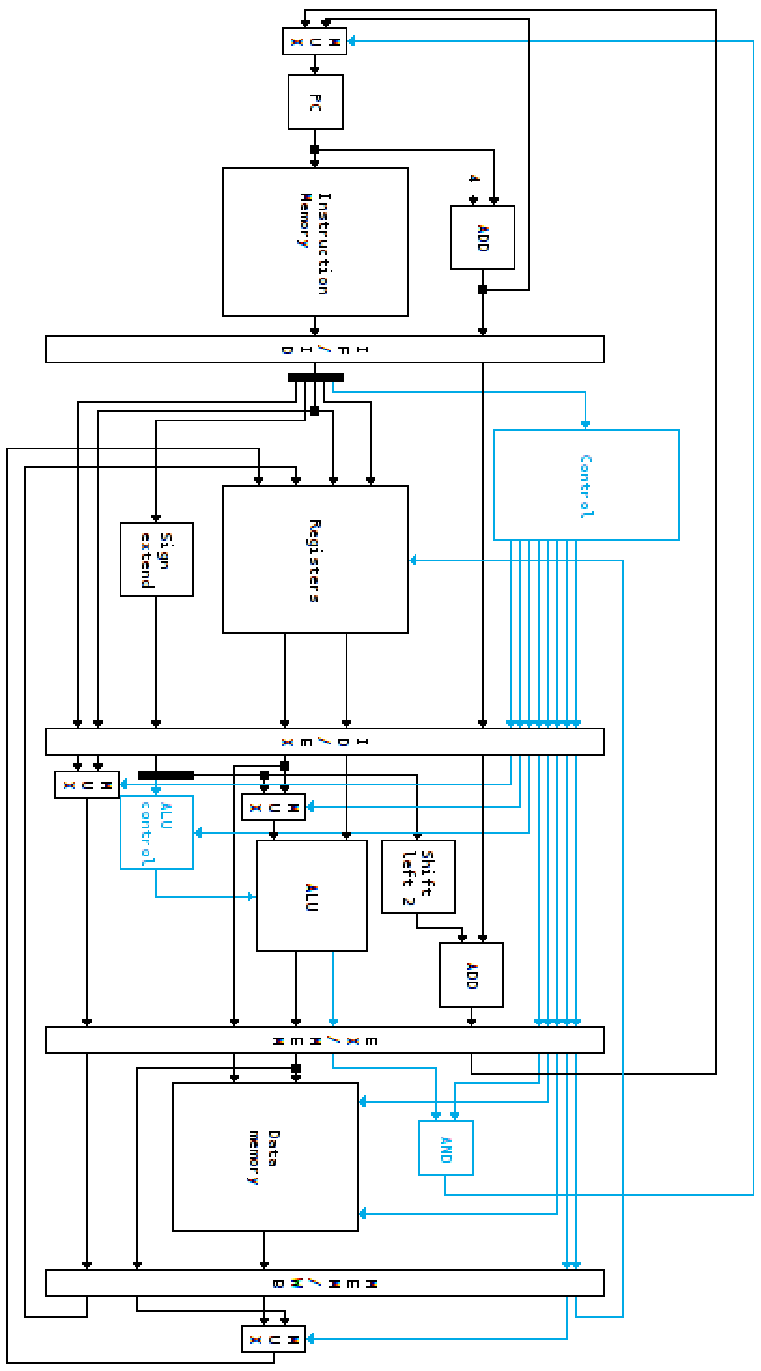


Figure 4.7: Pipeline datapath variant that doesn't implement hazard detection

extend from the more general class `AbstractInstruction`. Each instruction also defines the type it belongs to and the values of each field, that may either be constant or come from an argument. Each pseudo-instruction additionally defines the actual instructions it is converted to when assembled. In the developed simulator, each mnemonic can only belong to one instruction or pseudo-instruction.

Each argument of an instruction or pseudo-instruction can be of one of several types. These types are used to validate the arguments of the instructions in the code. They are also used show a description to the user with what can be used as arguments. The different types are:

reg The argument is a register and will be encoded as the index of the register.

int The argument is an immediate value (an integer) and is encoded as such.

Although not informed in the user interface, the argument can also be a label, which is encoded as the address of the label. This is allowed so that the `la`[6] pseudo-instruction can be supported. This pseudo-instruction, in the implemented datapaths, is converted to the `addi` instruction, meaning that the integer argument of that instruction can also be a label.

target The argument is a code label or an integer address divided by 4, usually used for the target of a jump. When the argument is an integer it is encoded as such. When it is a label it is encoded as the address of the label divided by 4.

offset The argument is an integer offset or a code label, usually used by branches. When the argument is an integer it is encoded as such. When it is a label it is encoded as the difference between the address of the label and the address of the next instruction, divided by 4.

data The argument is a reference to a value in the data memory, usually used by loads and stores. The reference is composed by the base address of the value and an optional offset. The base of the reference is either an integer address of a data label, while the offset is a register.

label The argument is a label. Technically it is parsed like a **int** argument, but the description shown to the user is different.

Another thing that the instruction set defines is how the CPU's control unit works. This is defined in the `Control` class and is associated to the control unit component. The class specifies the values of all the control signals for each instruction *op code*. This information is stored in a `TreeMap`. Additionally, the sizes of the control signals are determined automatically based on their possible values.

The instruction set also defines how the ALU and the ALU control unit work. This is defined in the `ControlALU` class and is associated to the ALU and ALU control components. An ALU control unit has two inputs: the *ALUOp* control signal produced in the control unit and the *funct* field from the instruction. The values of these two inputs are used to determine the values of the output(s). The ALU control has at least one output, which should be connected to the control

[6]The `la` pseudo-instructions loads a register with the address of a label.

input of the ALU, but can have more. This correspondence between input and output values is specified in this class. The class also specifies the correspondence between each possible ALU control input and operation to perform. These correspondences are stored in `TreeMaps`. The sizes of the outputs are determined automatically, like in the `Control` class. The `ControlALU` class is also responsible for calculating the results of the ALU operations.

The supported ALU functions are shown in Table 4.2 for the standard ALU and the "extended" ALU mentioned in the previous section. The names of the functions in the 1st column are the actual names used to reference the functions in the instruction set file.

Function	Meaning	ALU	Extended ALU
`and`	AND	✓	✓
`or`	OR	✓	✓
`xor`	XOR	✓	✓
`nor`	NOR	✓	✓
`add`	add	✓	✓
`sub`	subtract	✓	✓
`slt`	set on less than	✓	✓
`sll`	shift left logical	✓	✓
`srl`	shift right logical	✓	✓
`sra`	shift right arithmetic	✓	✓
`mult`	multiply	✗	✓
`div`	divide	✗	✓
`mfhi`	move from `hi`	✗	✓
`mflo`	move from `lo`	✗	✓

Table 4.2: Functions supported by the ALU and the extended ALU

Like the CPU, each instruction set is defined in a JSON file. An instruction set file lists the instruction types, the instructions, the pseudo-instructions and their properties and also specify how the control unit, the ALU and ALU control work. A snippet of the definition of an instruction set file is shown in Listing 4.2. Most instructions, pseudo-instructions and control informations were omitted for brevity and replaced by ellipsis.

```
{
  "types": {
    "R": [{"id": "op", "size": 6}, {"id": "rs", "size": 5}, {"id": "rt", "size":
        5}, {"id": "rd", "size": 5}, {"id": "shamt", "size": 5}, {"id": "func", "
        size": 6}],
    "I": [{"id": "op", "size": 6}, {"id": "rs", "size": 5}, {"id": "rt", "size":
        5}, {"id": "imm", "size": 16}],
    "J": [{"id": "op", "size": 6}, {"id": "target", "size": 26}]
  },
  "instructions": {
    "add":  {"type": "R", "args": ["reg", "reg", "reg"], "fields": {"op": 0, "rs": "#2", "rt": "#3", "rd": "#1", "shamt": 0, "func": 32}, "desc": "$t1 = $t2 + $t3"},
    ...
  },
  "pseudo": {
    "move": {"args": ["reg", "reg"], "to": ["add #1, #2, $0"], "desc": "$t1 = $t2"},
    ...
  },
  "control": {
    "0": {"RegDst": 1, "RegWrite": 1, "ALUOp": 2, "ALUSrc": 0, "MemToReg": 0},
    ...
  },
  "alu": {
    "aluop_size": 2,
    "func_size": 6,
    "control_size": 3,
    "control": [
      {"aluop": 0, "out": {"control": 2}},
      {"aluop": 2, "func": 32, "out": {"control": 2}},
      ...
    ],
    "operations": {
      "2": "add",
      ...
    }
  }
}
```

Listing 4.2: Snippet of the definition of an instruction set file

The different sections that compose the instruction set file are:

types Lists the types of instructions that exist and defines the identifiers and sizes of the fields of each type. The first field is considered to be the *op code* field and must have the same size in all types.

instructions Defines the supported instructions and their properties. These properties are:

- The type of the instruction.
- The types of the arguments.
- The values that each field will have when assembled.
- A short symbolic description of the operation it performs.

pseudo Defines the supported pseudo-instructions and their properties. These properties are:

- The types of the arguments.

- The list of instructions the pseudo-instruction is converted to when assembled.
- A short symbolic description of the operation it performs.

control Defines the behaviour of the processor's control unit.

The implemented instruction sets are presented in the next subsection. Note that the instruction sets are configurable and more can be created without changing the code, unless more features are required. Floating point operations and syscalls are currently not supported.

4.2.1 Provided Instruction Sets

The instructions and pseudo-instructions supported by the different implemented instruction sets are shown in tables 4.3 and 4.4, respectively. The encoding and operation of the instructions were based mostly on the reference book [PH05]. The translation of pseudo-instructions to actual instructions was based on the MARS simulator [Uni12, VS06].

As shown in subsections 4.1.1 and 4.1.2, several variations of the unicycle and pipeline datapaths were implemented. As such, several variations of the instruction set had to be defined. The reference instruction set is `default`, and is used by the default unicycle datapath. The first variation of this instruction set is `default-no-jump`, shortened to *DNJ* in the tables, and doesn't support jumps. The second variation `default-no-jump-branch`, shortened to *DNJB* in the tables, additionally doesn't support branches.

An extended instruction set was also defined. This instruction set supports some additional instructions, including multiplications and divisions. These instructions are not detailed in the reference book [PH05] and, for this reason, the extended instruction set was created. The name of this is instruction set is `default-extended` and is shortened to *DE* in the tables. The extended instruction set also has a variation, called `default-extended-no-jump` and shortened to *DENJ* in the tables, that doesn't support jumps. This variation is used by the extended pipeline datapath.

4.3 The Assembler

The assembler is represented by the `Assembler` class, and is accessible from the `CPU`, as visible in Figure 3.4. When the code is to be assembled, the user interface uses the `CPU`'s `Assembler`. To parse the code, the assembler consults the CPU's `InstructionSet` and converts the instructions and pseudo-instructions in the text segment to assembled instructions, represented by the `AssembledInstruction` class, which are then loaded into the CPU's instruction memory. The assembling process also involves parsing the data segment in the code to initialize the CPU's data memory with the specified values.

The syntax of the assembler is based on the MARS simulator [Uni12, VS06]. Comments can be added to any line and are started by the '`#`' character. A label can be added to any line in the text and data segments, but only one label is allowed per line. Integer values can be written in

	Default	DNJ	DNJB	DENJ	DE
`nop`	✓	✓	✓	✓	✓
`add`	✓	✓	✓	✓	✓
`sub`	✓	✓	✓	✓	✓
`and`	✓	✓	✓	✓	✓
`or`	✓	✓	✓	✓	✓
`nor`	✓	✓	✓	✓	✓
`slt`	✓	✓	✓	✓	✓
`addi`	✓	✓	✓	✓	✓
`lw`	✓	✓	✓	✓	✓
`sw`	✓	✓	✓	✓	✓
`beq`	✓	✓	✗	✓	✓
`j`	✓	✗	✗	✗	✓
`xor`	✗	✗	✗	✓	✓
`mult`	✗	✗	✗	✓	✓
`div`	✗	✗	✗	✓	✓
`mfhi`	✗	✗	✗	✓	✓
`mflo`	✗	✗	✗	✓	✓

Table 4.3: The instructions supported by each instruction set

	Default	DNJ	DNJB	DENJ	DE
`li`	✓	✓	✓	✓	✓
`la`	✓	✓	✓	✓	✓
`move`	✓	✓	✓	✓	✓
`subi`	✓	✓	✓	✓	✓
`sgt`	✓	✓	✓	✓	✓
`neg`	✓	✓	✓	✓	✓
`not`	✓	✓	✓	✓	✓
`bge`	✓	✗	✓	✓	✓
`ble`	✓	✗	✓	✓	✓
`b`	✓	✗	✓	✓	✓
`mul`	✗	✗	✗	✓	✓
`rem`	✗	✗	✗	✓	✓

Table 4.4: The pseudo-instructions supported by each instruction set

decimal format and also in hexadecimal and octal format in C-style[7] Some examples of code are presented in Subsection 4.3.1.

The code entered by the user can have errors, which must be shown to the user. The errors detected include invalid or duplicated labels, invalid instruction arguments and referencing unknown labels, assembler directives and instructions. When an error is found in an instruction, the assembler throws a `SyntaxErrorException`. However, instead of stopping the assembler there, the exception is caught and added to a list of exceptions. The assembler then resumes in the next instruction. With this technique, all the errors present in the code can be shown to the user.

The syntax errors are shown to the user in the selected language. To do that, and to keep the `SyntaxErrorException` class platform-independent, the class defines an enumeration of the different possible errors. Each type of error may also need some additional arguments that are shown to the user. The exception has, for that reason, multiple constructors to allow these arguments (at most 2). Each syntax error also stores the line where it occurred.

The implementation of the assembler is fairly simple, and the assembling process is done in two steps:

1. In the first step, the assembler parses the data segment and loads the values of the data memory, while also keeping track of labels and converting the pseudo-instructions into instructions in the text segment. The addresses of the labels are stored in two separate `TreeMaps`, one for the text segment and the other for the data segment. The lines in the code are parsed one by one and each line can belong either to the data segment or to the text segment. The code starts by default in the text segment, and the current segment is tracked by a simple switch. All the resulting instructions are stored in a list, and each of these instructions is represented by the `CodeLine`. This class stores both the line of code of the instruction and the number of the line it was.

2. In the second step, the instructions that resulted from the previous step are parsed, assembled into machine code and loaded into the CPU's instruction memory. Each of the resulting instructions in machine code is represented by the `AssembledInstruction` class. This class stores a reference to the respective `Instruction` class and the machine code using the `Data` class. It also stores the original line of code, the number of the line it was and the labels that referenced the instruction.

In terms of assembler directives, four directives are currently supported:

- `.data`: starts the data segment, where the data memory is initialized.
- `.text`: starts the text segment, where the code is defined.
- `.word`: declares one or more values to be stored in the data memory as 32 bits words.
- `.space`: reserves some bytes in the data memory.

[7] In C-style hexadecimal values are preceded by `0x`, while octal values are preceded by a `0`.

One detail of importance not yet mentioned is that the addresses of both data segment and text segment start at zero. This simplifies the assembling process considerably, as the jump target and load/store addresses will always correspond to the desired addresses when extended to 32 bits, unless the program is very large. It also makes the resulting machine code somewhat easier to understand by the students. However, this can deceive them into thinking that the first address is always zero, which is not correct.

4.3.1 Code Examples

Some examples of code accepted by the simulator are presented in this subsection to better understand the valid syntax. These programs where the most used to test the simulator.

The first example (Listing 4.3) loads a register with the value 1. The register is then added to himself ten times in a loop. The final value of the register should be 1024. The second example (Listing 4.4) copies the 10 elements from the "src" array to "dest". The third example (Listing 4.5) reverses the 10 elements of "array". Note that some of the implemented datapaths won't be able to run all of the examples. For example, pipeline datapaths cannot run the third example because it uses the `j` instruction, which isn't supported. That instruction can, however, be easily replaced by the `b` pseudo-instruction.

```
.data # Data segment
  one:  .word   1
  ten:  .word   10
  res:  .space  4

.text # Text segment (code)
  lw    $t0, one
  lw    $t1, ten
  move  $t4, $t0
loop:
  add   $t0, $t0, $t0
  subi  $t1, $t1, 1
  bge   $t1, $t4, loop
  sw    $t0, res
```

Listing 4.3: First code example

```
.data # Data segment
  src:  .word   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
  len:  .word   10
  dest: .space  40

.text # Text segment
  # Initialize
  lw    $t0, len
  subi  $t0, $t0, 1
  li    $t1, 0
  li    $t2, 0

loop: # Copy loop
  lw    $t3, src($t1)
  sw    $t3, dest($t2)
  subi  $t0, $t0, 1
  addi  $t1, $t1, 4
  addi  $t2, $t2, 4
  bge   $t0, $zero, loop
```

Listing 4.4: Second code example

```
.data # Data segment
  array:.word   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
  len:  .word   10

.text # Text segment (code)
  # Initialize
  move  $t1, $zero # offset for the 1st element
  lw    $t2, len # determine offset for the last element
  subi  $t2, $t2, 1
  add   $t2, $t2, $t2 # multiply by 4
  add   $t2, $t2, $t2

loop: # Copy loop
  bge   $t1, $t2, fim
  lw    $t3, array($t1)
  lw    $t4, array($t2)
  sw    $t3, array($t2)
  sw    $t4, array($t1)
  addi  $t1, $t1, 4
  subi  $t2, $t2, 4
  j     loop

fim:
```

Listing 4.5: Third code example

4.4 Simulation Execution

To execute a program, the GUI first uses the `CPU`'s `Assembler` to assemble the program and initialize the CPU's instruction and data memories, as explained in the previous section. If the program has no errors, the GUI then uses the `CPU` to run the program, either step-by-step or all at once. It is also possible to back step (i.e. return to the previous clock cycle) and to reset the simulation to the initial cycle.

Internally, each clock cycle starts by executing the synchronous behaviour of the synchronous components, and then proceeds to execute the “normal” behaviour all of the components, starting by the synchronous ones. During this process, the outputs of the components usually have their values changed. When this happens, the new value is propagated to the connected input, which then executes the “normal” behaviour of the input’s component and, possibly, continues the data propagation. This only occurs when the new value is changed (i.e. the new value is different from the previous one), avoiding infinite loops that would have resulted from this solution. This also means that the synchronous behaviour of synchronous components cannot cause any data propagation.

In each clock cycle, each wire can be marked as “irrelevant” (gray on the user interface) or relevant. This decision is made based only on the values in the wires and components, and not on the instruction, as it would be fairly difficult to determine the relevant wires and components for each instruction when the CPU and even the instruction set is very generic and configurable. That said, the conditions to mark a wire as irrelevant are very simple: the wire carries one bit with the value zero, a stall is occurring, the wire is not selected by a multiplexer, etc.

For the back step function to work, the synchronous components store their internal states in each clock cycle in a stack. To return to the previous cycle, the `CPU` simply restores the previous state of each synchronous component from its stack and then executes the normal behaviours of all components to propagate the correct values to the rest of the datapath. Resetting state of the CPU to its first clock cycle is done by first removing all the states from the stacks except the first and then using the first state to restore the datapath.

The simulator also tracks the instruction or instructions that are being executed in the datapath. This is easy in unicycle datapaths, as only one instruction is being executed at a time and the program counter points to that instruction. Pipeline datapaths, however, can have five instructions being executed in each cycle. There is also no direct way to determine what instruction is in each stage, except in the first one through the program counter. To solve this problem, the program counter and the pipeline registers store the index of the instruction in that stage. They also store all the previous indexes in a stack to allow the back step function. In each clock cycle transition, the indexes are moved to the next pipeline register, except when the register has the `Write` input inactive, where the index is not changed, or the `Flush` input active, where a `-1` index is stored. This `-1` index represents a `nop` instruction that isn’t in the code, either because of a flush or because the program has reached the end in that stage. The simulation of a program is considered finished when the index in the program counter and all pipeline registers is `-1`.

It is also possible to execute the code all at once. This is done by simply executing the program cycle-by-cycle until the program is considered finished. However, a limit of 1000 clock cycles has been implemented. When the limit is reached, an `InfiniteLoopException` is thrown. This is to stop infinite loops in the code and warn the user. The limit can be reached without infinite loops, though, when the program is very large or has a loop with a large number of iterations.

4.5 Concluding Remarks

This chapter discussed how the internal platform-independent implementation logic, used by the user interfaces of both PC and Android versions, was implemented. The next chapter discusses how the graphical user interfaces of the PC and Android versions were implemented.

Chapter 5

Interface Implementation

The simulator was developed for both computers and touch screen Android devices, especially tablets. This chapter discusses how their user interfaces were implemented. Looking at the UML class diagram in Figure 3.4, the user interface code is defined in the `gui` package, and is the only package that differs between the PC and Android versions.

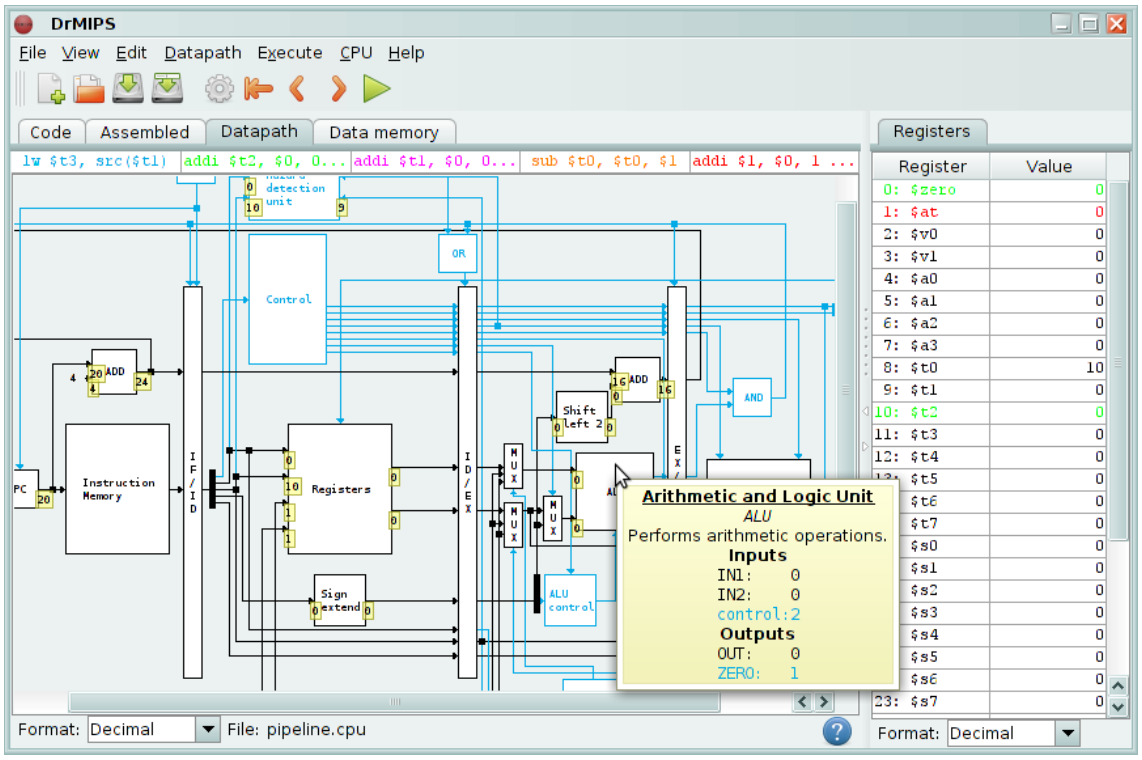
5.1 PC Version

The PC version is the most complete. The GUI is comprised of one window plus some dialog boxes. By default, the interface is shown with a light theme and with the contents split in five tabs, as shown in Figure 5.1. Two tabs can be viewed at once, as the window is split in two sides horizontally, and the user can move any tab from one side to the other by right clicking on the tab title and selecting the only option from the pop-up menu. However, the user can choose to use a dark theme and can also choose to use internal windows instead of tabs, as shown in Figure 5.2. Using internal windows is useful in large screens, and the windows' positions and sizes are remembered on exit. The light and dark themes use the JTattoo look and feel [Hag].

The different tabs or windows, detailed later, are:

- *Code*: contains the code editor.
- *Assembled*: displays the assembled instructions and resulting machine code.
- *Datapath*: displays the datapath and the instruction(s) being executed.
- *Registers*: lists the registers and their values.
- *Data memory*: shows the values in the data memory.

The UML class diagram of the GUI of the PC version is shown in Figure 5.3. `DrMIPS` is the class that contains the `main` function. The `main` function does some initializations, loads the initial translated strings and launches the simulator's main window. A loading dialog, defined by the `DlgLoading` class, is shown while the main window is not visible. `DrMIPS` also defines several parameters used in the interface.

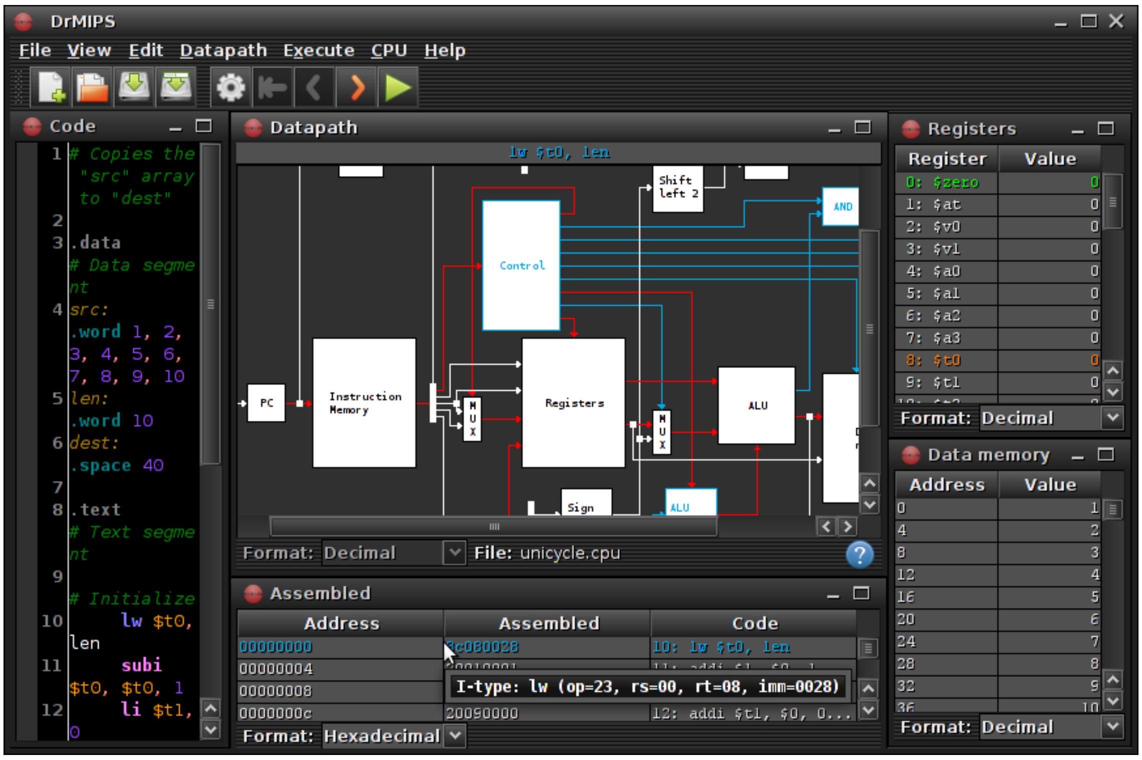


Arithmetic and Logic Unit
ALU
Performs arithmetic operations.
Inputs
IN1: 0
IN2: 0
control: 2
Outputs
OUT: 0
ZERO: 1

Register	Value
0: $zero	0
1: $at	0
2: $v0	0
3: $v1	0
4: $a0	0
5: $a1	0
6: $a2	0
7: $a3	0
8: $t0	10
9: $t1	0
10: $t2	0
11: $t3	0
12: $t4	0
$t5	0
$t6	0
$t7	0
$s0	0
$s1	0
$s2	0
$s3	0
$s4	0
$s5	0
$s6	0
23: $s7	0

Format: Decimal   File: pipeline.cpu   Format: Decimal

Figure 5.1: DrMIPS for the PC with the default options, with the tooltip of a component being shown



```
# Copies the "src" array to "dest"

.data
# Data segment
src:
.word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
len:
.word 10
dest:
.space 40

.text
# Text segment

# Initialize
    lw $t0, len
    subi $t0, $t0, 1
    li $t1, 0
```

Format: Decimal   File: unicycle.cpu

Address	Assembled	Code
00000000	8c080028	10: lw $t0, len
00000004	[illegible]	11: addi $t1, $0, 1
00000008		
0000000c	20090000	12: addi $t1, $0, 0...

I-type: lw (op=23, rs=00, rt=08, imm=0028)

Format: Hexadecimal

Register	Value
0: $zero	0
1: $at	0
2: $v0	0
3: $v1	0
4: $a0	0
5: $a1	0
6: $a2	0
7: $a3	0
8: $t0	0
9: $t1	0

Format: Decimal

Address	Value
0	1
4	2
8	3
12	4
16	5
20	6
24	7
28	8
32	9
36	10

Format: Decimal

Figure 5.2: DrMIPS for the PC using internal windows and dark theme while in “performance mode”

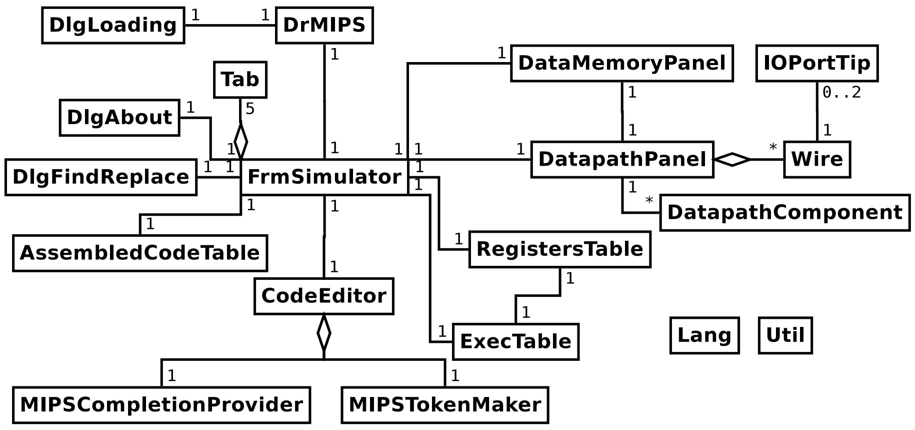


Figure 5.3: UML class diagram of the GUI of the PC version

The interface supports multiple languages, Portuguese and English at this stage, and they can be changed in the menus. The translated strings of each language are stored in a language file with the *lng* extension in the *lang* directory. A small extract of a language file is shown in Listing 5.1.

```
# General
ok=OK
cancel=Cancel
yes=&Yes
no=&No
close=&Close

# Errors
error_opening_file=Error opening file #1!
invalid_file=Invalid file!
wrong_no_args=Wrong number of arguments! Expected #1, found #2.
```

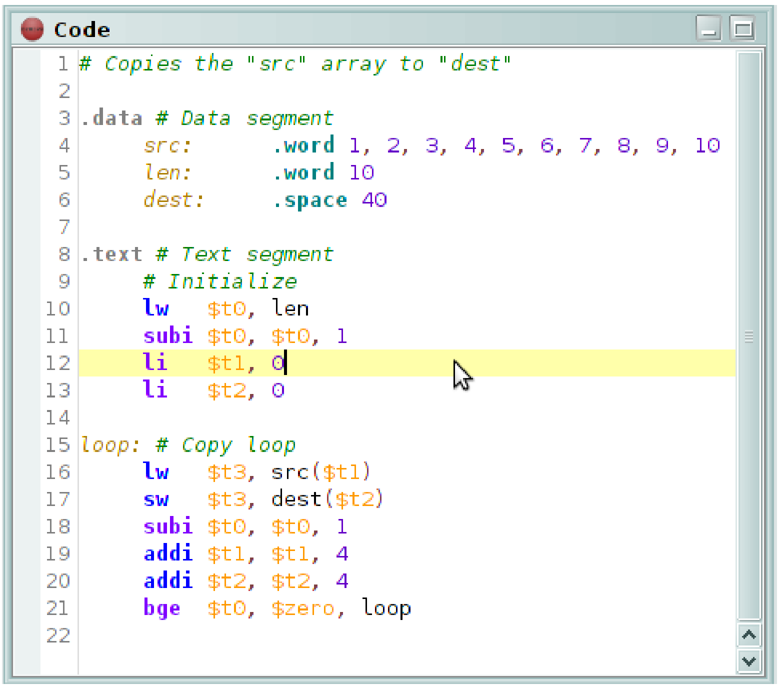
Listing 5.1: Small extract of a language file

Each translatable string is identified by a unique key. The translated strings are defined in the file in a pair `key=string`, one per line. Empty lines and comment lines, started by the `#` character, are also allowed. If a string is missing for the language, its identifier is shown in upper case instead. The mnemonic letter[1] of the string of a menu or button can be identified by preceding it with an ampersand (`&`) character. To include an actual ampersand in the string, a double ampersand (`&&`) must be written. A translatable string can also have arguments. Those are specified in the file using `#1` for the first argument, `#2` for the second, and so on. In terms of code, this is handled by the static `Lang` class. It provides methods to load language files and to retrieve translated strings and their mnemonics. The translated strings and their keys are stored in a `HashMap` to improve speed.

[1]The mnemonic letter of a menu or button in a GUI is usually underlined and allows the menu or button to be activated by pressing that button on the keyboard and the ALT key.

The `FrmSimulator` class implements the main window of the simulator. The interface was designed in Netbeans [Ora13] and consists of a menu bar, defined in a `JMenuBar`, a tool bar, defined in a `JToolBar`, and the content area of the window that contains the five tabs or windows mentioned before. The contents of each tab or window are inside a `JPanel`. When using the default tabbed mode, the content area consists of a horizontal `JSplitPane` with a `JTabbedPane` on both sides. Right-clicking on a tab will show a pop-up menu with one item to move the tab to the other side of the split pane. This is done by re-adding all the tabs to their corresponding tabbed pane, including the moved tab, to the other side, so the order of the tabs is kept. The location of each tab is tracked and handled by the internal `Tab` class. When the user switches to the internal windows mode, the `JSplitPane` is removed and replaced by a `JDesktopPane`. The `JPanel` of each tab is then moved to a `JInternalFrame` inside the desktop pane.

The location of the tabs, the position of the windows, the language and several other properties are remembered on exit and restored on start. This is done by using the `Preferences` class of the `java.util.prefs` package, which provides easy and transparent access and storage of the preferences of the application. The ten most recent code and CPU files opened are also tracked using this method. Remembering the properties of the interface on exit is very important. It would be very annoying for the user if he had to reconfigure everything to his liking every time he used the application, especially if the user prefers to use the internal windows mode.



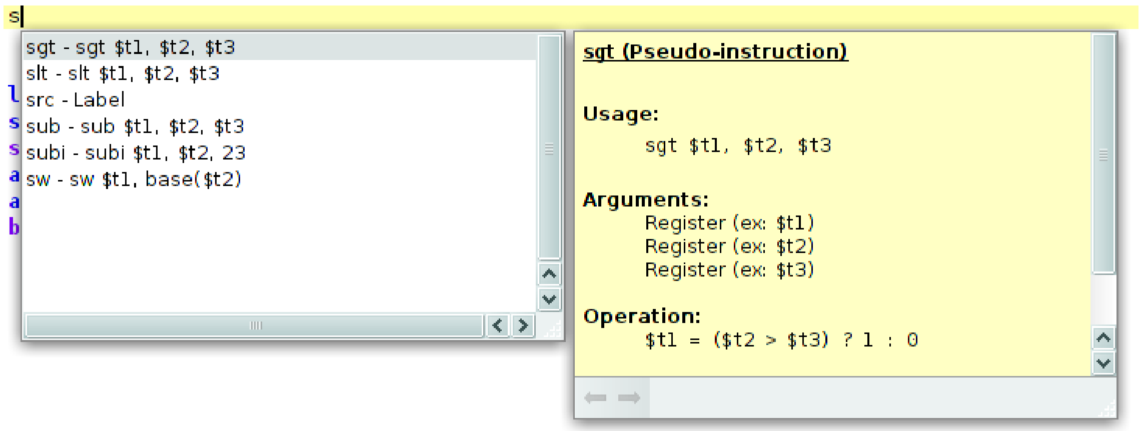
```
# Copies the "src" array to "dest"

.data # Data segment
    src:      .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    len:      .word 10
    dest:     .space 40

.text # Text segment
    # Initialize
    lw   $t0, len
    subi $t0, $t0, 1
    li   $t1, 0
    li   $t2, 0

loop: # Copy loop
    lw   $t3, src($t1)
    sw   $t3, dest($t2)
    subi $t0, $t0, 1
    addi $t1, $t1, 4
    addi $t2, $t2, 4
    bge  $t0, $zero, loop
```

Figure 5.4: The code tab or window in the PC

DrMIPS provides a code editor with syntax-highlighting, auto-complete, search/replace, line numbers, undo and redo thanks to the RSyntaxTextArea component [Fif13b] and AutoComplete library [Fif13a]. The code editor can be seen in Figure 5.4.

The syntax-highlighting rules are not “static” and depend on the loaded CPU datapath, more specifically on the names of the registers and supported instructions and pseudo-instructions. The rules are implemented in the internal `MIPSTokenMaker` class, which extends from the `AbstractTokenMaker` class of RSyntaxTextArea. The most important method in this class is `getTokenList()`, where a line of code is parsed and converted into tokens of several different types. The editor then uses these tokens to display the several elements in the code with different colors and styles. The different types of elements identified include instructions, pseudo-instructions, numbers, assembler directives, comments, registers, labels and punctuation. Note that instructions and pseudo-instructions are highlighted with different colors.

The auto-complete provided is mostly for help and is activated by `<Ctrl>+<Space>`. Like the syntax-highlighting, the auto-complete feature is dynamic. The completions provided include instructions, pseudo-instructions, assembler directives and labels found in the code. Activating the auto-complete feature displays a list of possible completions for the keyword being written. A help frame with the description of the completion and how it is used is also displayed (see Figure 5.5). The help frame for an instruction specifies how it is used, its arguments and the symbolic description defined in the instruction set file, as discussed in Section 4.2. The help frame for a pseudo-instruction additionally lists the actual instructions it is converted to. This auto-complete functionality is provided by the AutoComplete library, although some adjustments were made to allow the auto-completion of keywords started by a dot or underscore and to dynamically include the labels in the list of completions. These adjustment were made in the `MIPSCompletionProvider` class by extending `DefaultCompletionProvider`.



```
s
sgt - sgt $t1, $t2, $t3
slt - slt $t1, $t2, $t3
src - Label
sub - sub $t1, $t2, $t3
subi - subi $t1, $t2, 23
sw - sw $t1, base($t2)
```

sgt (Pseudo-instruction)

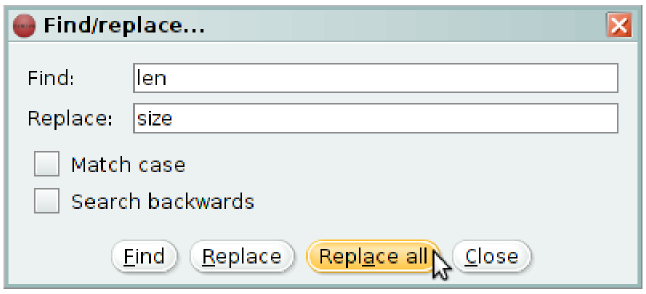
Usage:
```
    sgt $t1, $t2, $t3
```

Arguments:
```
    Register (ex: $t1)
    Register (ex: $t2)
    Register (ex: $t3)
```

Operation:
```
    $t1 = ($t2 > $t3) ? 1 : 0
```

Figure 5.5: Auto-complete list and description frame

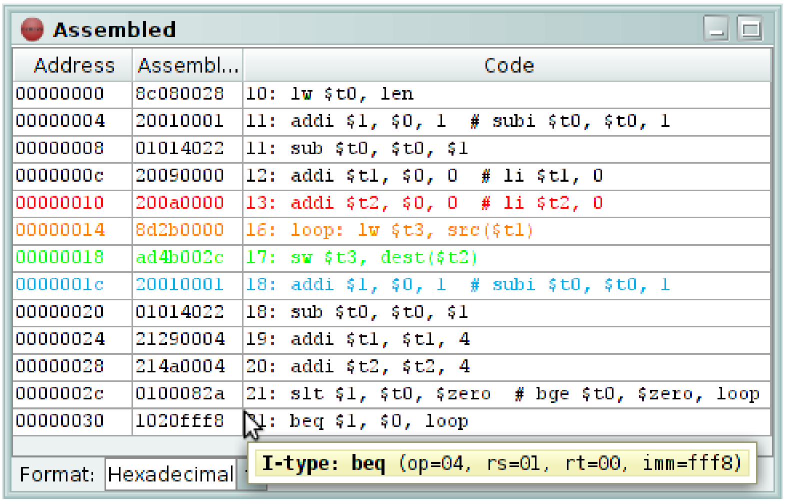
The undo, redo, copy and paste features are handled by the RSyntaxTextArea component automatically. The component also provides methods to search and replace in the code. The search/replace dialog (see Figure 5.6) can be accessed either from the menu or from the `<Ctrl>+F` shortcut. Right-clicking the editor also displays the menu. RSyntaxTextArea provides some more advanced search and replace features, including searches with regular expressions, but they were not included in the interface to keep it simple. Finally, all of the errors present in the code are indicated close to the line numbers as icons when the user assembles the code. Hovering the mouse cursor over one of these icons display a tooltip describing the error.



Find: len

Replace: size

Figure 5.6: Search/replace dialog in the PC



Address	Assembl...	Code
00000000	8c080028	10: lw $t0, len
00000004	20010001	11: addi $1, $0, 1 # subi $t0, $t0, 1
00000008	01014022	11: sub $t0, $t0, $1
0000000c	20090000	12: addi $t1, $0, 0 # li $t1, 0
00000010	200a0000	13: addi $t2, $0, 0 # li $t2, 0
00000014	8d2b0000	16: loop: lw $t3, src($t1)
00000018	ad4b002c	17: sw $t3, dest($t2)
0000001c	20010001	18: addi $1, $0, 1 # subi $t0, $t0, 1
00000020	01014022	18: sub $t0, $t0, $1
00000024	21290004	19: addi $t1, $t1, 4
00000028	214a0004	20: addi $t2, $t2, 4
0000002c	0100082a	21: slt $1, $t0, $zero # bge $t0, $zero, loop
00000030	1020fff8	21: beq $1, $0, loop

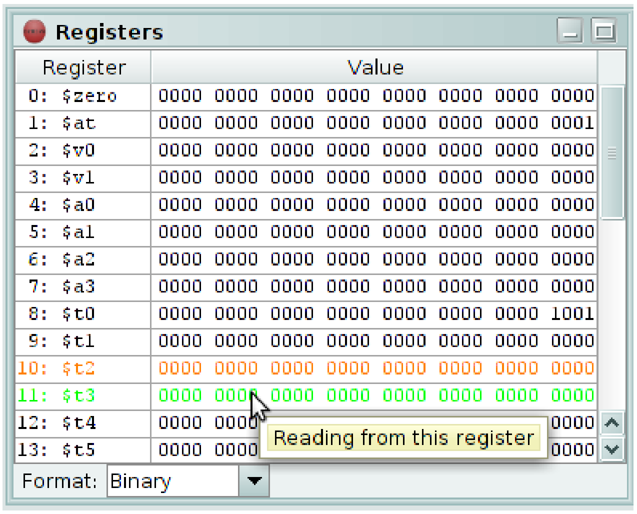
Format: Hexadecimal

I-type: beq (op=04, rs=01, rt=00, imm=fff8)

Figure 5.7: The assembled code tab or window in the PC

The assembled code table displays the resulting machine code and highlights the instructions being executed by the CPU (see Figure 5.7). The table has three columns, showing for each instruction its address, machine code and respective line of code. The number of the line where the instruction appears is also displayed in the last column. Furthermore, if the instruction results from a pseudo-instruction, and if it is the first actual instruction of that pseudo-instruction, a comment with the original pseudo-instruction is added. The instructions currently being executed by the CPU are highlighted with different colors (if using a pipelined datapath). Hovering the mouse cursor over each instruction also displays a tooltip with the type of the instruction and the values of the instruction's fields. The format that the values are displayed can be changed in the combo box below the table. The available formats are binary, decimal and hexadecimal. The table is defined in the `AssembledCodeTable` class and, to highlight the instructions being executed, a custom `TableCellRenderer` was implemented.

The registers table displays the registers, preceded by their indexes, and their values (see Figure 5.8). It also displays the value of the program counter. Like in the assembled code table,


Registers

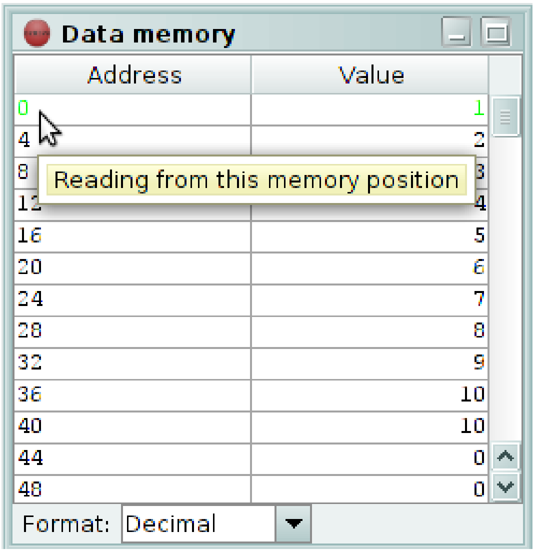
Register	Value
0: $zero	0000 0000 0000 0000 0000 0000 0000 0000
1: $at	0000 0000 0000 0000 0000 0000 0000 0001
2: $v0	0000 0000 0000 0000 0000 0000 0000 0000
3: $v1	0000 0000 0000 0000 0000 0000 0000 0000
4: $a0	0000 0000 0000 0000 0000 0000 0000 0000
5: $a1	0000 0000 0000 0000 0000 0000 0000 0000
6: $a2	0000 0000 0000 0000 0000 0000 0000 0000
7: $a3	0000 0000 0000 0000 0000 0000 0000 0000
8: $t0	0000 0000 0000 0000 0000 0000 0000 1001
9: $t1	0000 0000 0000 0000 0000 0000 0000 0000
10: $t2	0000 0000 0000 0000 0000 0000 0000 0000
11: $t3	0000 0000 0000 0000 0000 0000 0000 0000
12: $t4	0000 0000 … 0000
13: $t5	0000 0000 … 0000

Reading from this register

Format: Binary

Figure 5.8: The registers tab or window in the PC

the values can be shown in several data formats. The registers currently being accessed are highlighted in three different colors: green if being read, red if being written and orange if being read and written and the same time. The values of the registers can be changed by double-clicking them in the table, except the constant register `$zero`. The address of the program counter can also be changed this way, and doing so changes the instruction currently in the unicycle datapath or in the `IF` stage of the pipeline datapath. The address must be multiple of 4, though. The table is defined in the `RegistersTable` class.


Data memory

Address	Value
0	1
4	2
8	3
12	4
16	5
20	6
24	7
28	8
32	9
36	10
40	10
44	0
48	0

Reading from this memory position

Format: Decimal

Figure 5.9: The data memory tab or window in the PC

The data memory table is very similar to the registers table. It displays the contents of the data memory, one 32 bits word per line. Like in the registers table, the values can be displayed in several data formats and can be changed on double-click. The values being accessed are also highlighted when accessed in the same conditions of the register table: green when read, red when written and orange if both. Note that it's not possible to read and write to the same memory address with the default instruction sets, but it's possible to create a new one with an instruction that does that.

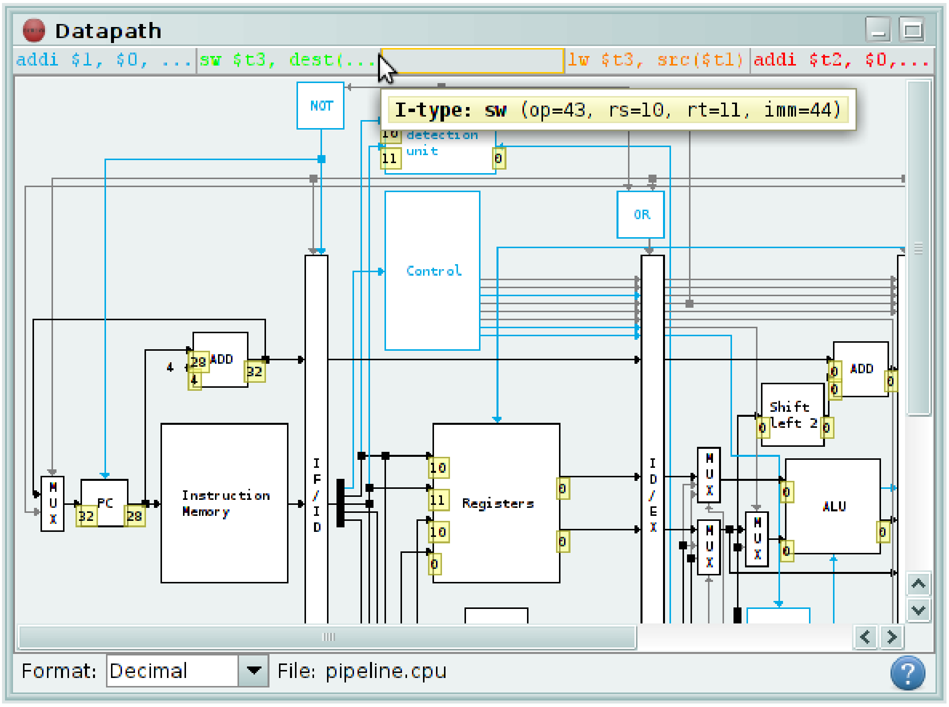


Figure 5.10: The datapath tab or window in the PC

The datapath is probably the most important part of the simulator. It is implemented in the `DatapathPanel` class, which handles all its graphical display. The datapath can be displayed in the normal data mode, as shown in Figure 5.10, or in performance mode, as shown in Figure 5.11. In the normal data mode the simulator shows the data that is flowing in the datapath in each step, while in performance mode the latencies of the components and the critical path of the datapath are displayed instead. The current mode can be switched in the *Datapath* menu.

The components of the datapath are all displayed as rectangles and, usually, with their names in the center of the rectangle. "Auxiliary" components, like forks, concatenators and distributors are displayed as small filled rectangles without name instead. The components are represented internally by the `DatapathComponent` class, which extends from `JPanel`.

Hovering the mouse cursor over a component displays some information about it in a tooltip. The tooltip displays the component's name, identifier, description and if it is synchronous. In the normal data mode the tooltip also displays the values at the inputs and outputs, while in performance mode the latency of the component and the accumulated latencies at the inputs and outputs are shown instead. The inputs and outputs that belong to the control path are shown in light blue. Figures 5.12 and 5.13 show two examples of these tooltips while in the data mode and performance mode, respectively. Additionally, while in performance mode, double-clicking a component presents a dialog box that allows the latency of the component to be changed. The

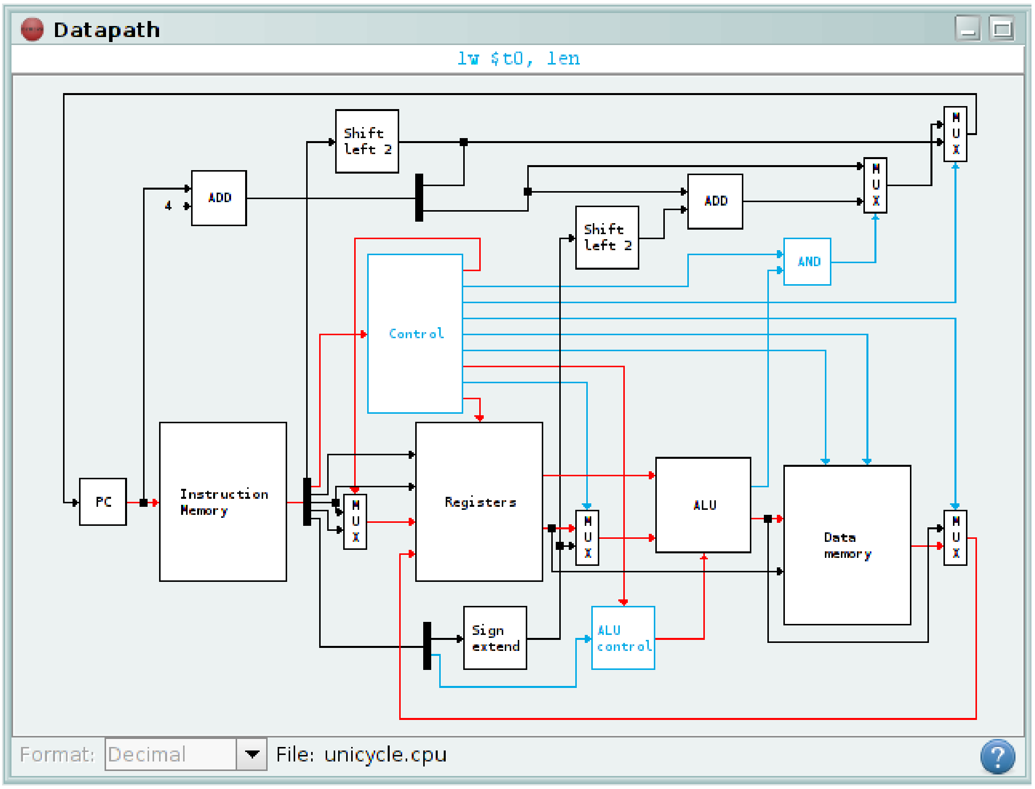
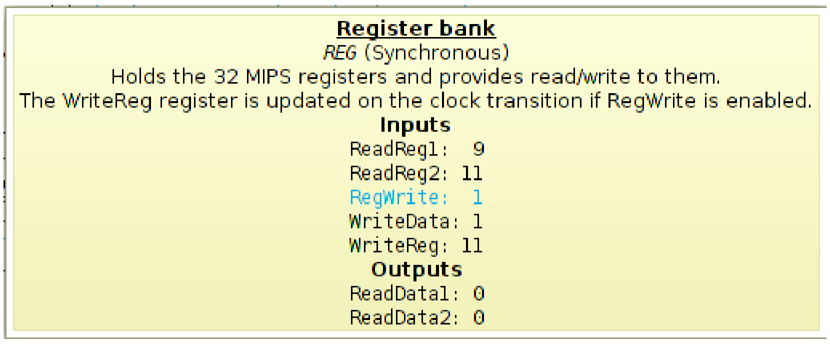


Figure 5.11: The datapath tab or window in performance mode in the PC



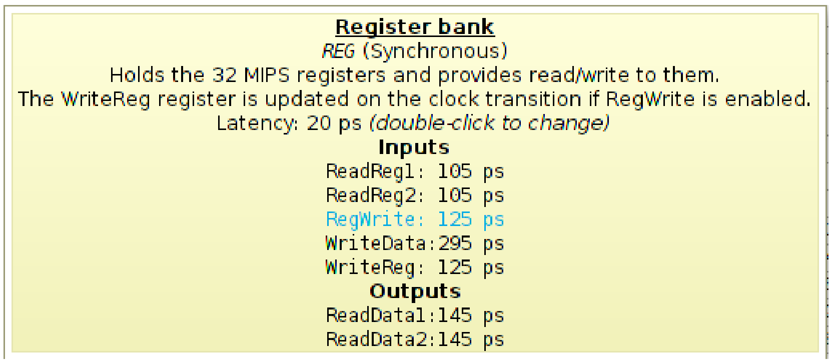
Register bank
REG (Synchronous)
Holds the 32 MIPS registers and provides read/write to them.
The WriteReg register is updated on the clock transition if RegWrite is enabled.
Inputs

```
ReadReg1:  9
ReadReg2: 11
RegWrite:  1
WriteData: 1
WriteReg: 11
```

Outputs

```
ReadData1: 0
ReadData2: 0
```

Figure 5.12: Tooltip of a component while in the normal data mode



Register bank
REG (Synchronous)
Holds the 32 MIPS registers and provides read/write to them.
The WriteReg register is updated on the clock transition if RegWrite is enabled.
Latency: 20 ps *(double-click to change)*
Inputs

```
ReadReg1: 105 ps
ReadReg2: 105 ps
RegWrite: 125 ps
WriteData:295 ps
WriteReg: 125 ps
```

Outputs

```
ReadData1:145 ps
ReadData2:145 ps
```

Figure 5.13: Tooltip of a component while in performance mode

new latency is only temporary, and the original latencies can be restored in the *Datapath* menu.

The wires that connect the components are drawn in the background of the datapath. At the end of the wires an arrow tip is drawn, however the user can hide these arrows tips in the *Datapath* menu. The wires are represented by the internal `Wire` class. Each wire corresponds to an output, has a start point, an end point and a list of intermediate points. The arrow tips are triangles drawn as a filled `Polygon`. The wires are drawn in different colors depending on a few conditions to better perceive the state of the datapath. A help icon is shown in the bottom right corner of the datapath tab or window that, when hovered, displays a legend as a tooltips with the meaning of the colors.

While in the normal data mode, each wire can have one of these colors:

- Black/white if it is a normal wire.
- Light blue if it is in the control path.
- Gray if it is an irrelevant wire (for the instruction(s) being executed).

While in performance mode, the possible colors are:

- Black/white if it is a normal wire.
- Light blue if it is in the control path.
- Red if it is in the critical path of the datapath.

The components and wires that belong to the control path can be hidden in the *Datapath* menu. This simplifies the datapath considerably as it becomes less cluttered. The values at the control inputs and outputs of the visible components can still be viewed in their tooltips. Hiding the control path allows students to focus on the flow of data in the datapath, hiding the complexities of the control path.

Some of the components' inputs and outputs display a small permanent tip with the current value of the input or output, while in the normal data mode. These data tips are always placed below the input or output's point of entry or exit. They are implemented in the `IOPortTip` class, which extends from `JLabel`, and are created and handled in the `Wire` class. Hovering the mouse cursor over a tip displays a tooltip with the identifier of the input or output. The identifiers were not included in the tips themselves because it would make them larger and clutter the datapath. Showing the tips for all inputs and outputs would also make the datapath cluttered. The datapath is a `JLayeredPane` to allow the data tips to be always on top. These tips can also be hidden in the *Datapath* menu. Showing the values at the inputs and outputs of the most important components directly in the datapath is very important, as the user can view the most important values in the datapath without having to move the mouse.

The instruction or instructions currently being executed are displayed above the datapath in a table with just one line and no header. Using a table like this means that the columns will probably not be aligned with the corresponding pipeline registers in the datapath, but it also means that they are always visible even if the whole datapath doesn't fit in the screen, as can be seen in Figure 5.10. Hovering the mouse cursor over an instruction in the table displays the type of the instruction and the values of its fields, like in the assembled code table. The instructions are colored with the same colors used in the assembled code table for consistency.

It was decided to make the graphical datapath as small as possible, but still readable. This way, a larger portion of the datapath, or even the whole datapath, will be visible. Because of this, some informations, like the names of the inputs and outputs, must be omitted. Finally, the values in the datapath can be also displayed in binary, decimal and hexadecimal formats.

5.2 Android Version

The Android version is very similar to the PC version. Like in the PC version, the application uses a light theme by default, but a dark theme is also available. Figure 5.15 shows the application running on a tablet with Android 4.0.3 using the light theme, while Figure 5.16 shows it running on a smartphone with Android 4.1.2 using the dark theme. The application was targeted for tablets, but it can also be run on smartphones, as seen in the second figure. As the PC and Android versions are very similar, this section will focus more on discussing their differences and presenting the Android version's user interface and features.

The UML class diagram of the the GUI of the Android version is shown in Figure 5.14. The class diagram of this version is considerably simpler than that of the PC version, as almost all of the GUI code was written in the main `DrMIPSActivity` class, except the code that refers to the datapath and its components.

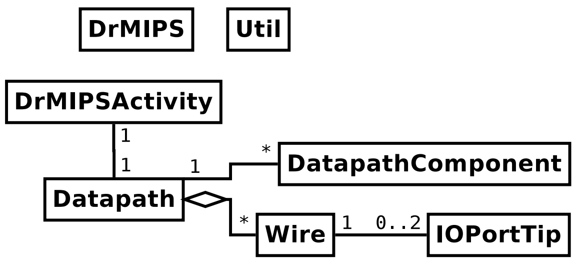
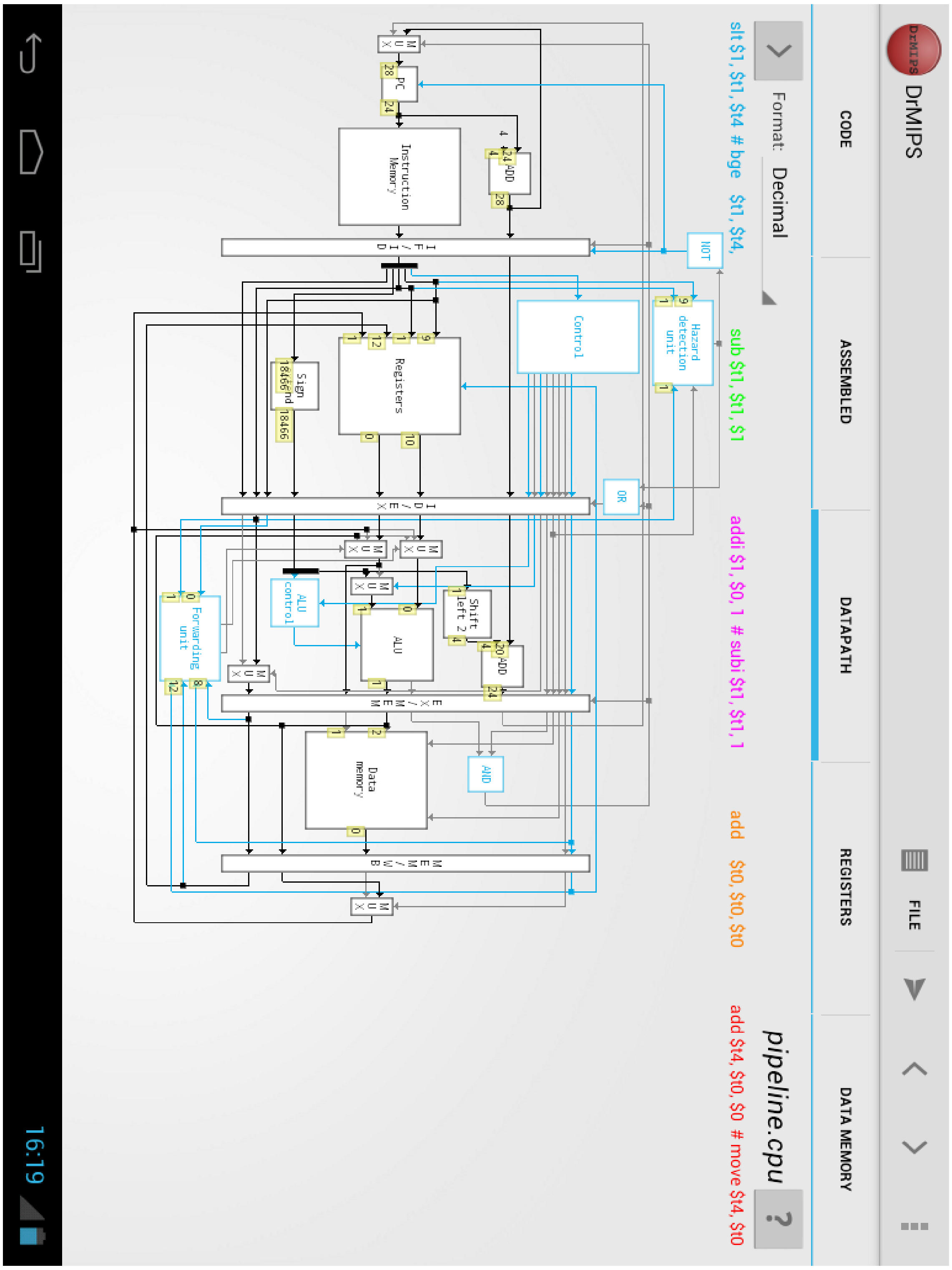


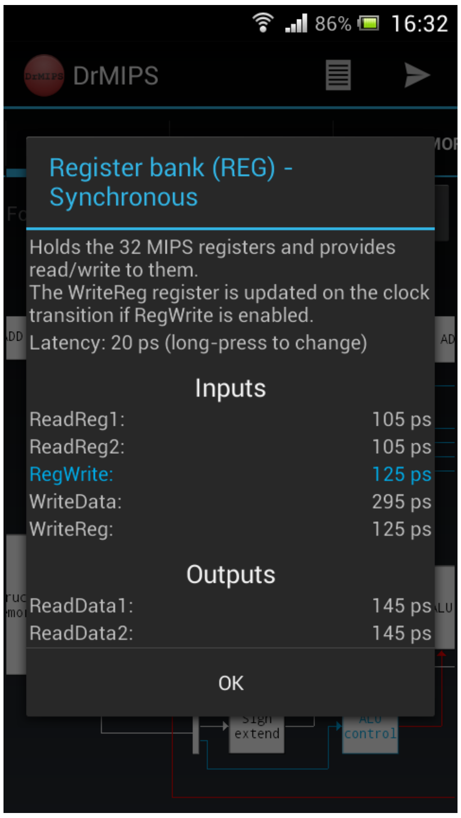
Figure 5.14: UML class diagram of the GUI of the Android version

`DrMIPS` is a custom `Application` class, where the currently loaded CPU is stored and where some interface parameters are defined. It is also where the CPU and instruction set files are extracted to the device's internal or external memory when they don't exist or when the application is upgraded. These and the code files are stored in the application's data directory, preferably in the external memory. This memory, usually mounted in `/mnt/sdcard`, is actually in many devices a partition of the device's internal memory and not the external memory card. But, in case it is the external memory card and is not available, the application stores the files on the application's private directory in the device's internal memory. Storing the files in the external memory allows the user to access them using another application like a file explorer. To know the path to the loaded CPU or code file, the user can press on the file's name in the user interface.



```
slt $1, $t1, $t4 # bge $t1, $t4,
sub $t1, $t1, $1
addi $1, $0, 1 # subi $t1, $t1, 1
add $t0, $t0, $t0
add $t4, $t0, $0 # move $t4, $t0
```

Figure 5.15: DrMIPS for the Android using the light version on a tablet



Register bank (REG) - Synchronous

Holds the 32 MIPS registers and provides read/write to them.
The WriteReg register is updated on the clock transition if RegWrite is enabled.
Latency: 20 ps (long-press to change)

Inputs

ReadReg1:	105 ps
ReadReg2:	105 ps
RegWrite:	125 ps
WriteData:	295 ps
WriteReg:	125 ps

Outputs

ReadData1:	145 ps
ReadData2:	145 ps

OK

Figure 5.16: DrMIPS for the Android using the dark version on a smartphone, displaying the details of a component while in "performance mode"

The application contains only one activity[2], `DrMIPSActivity`, and its contents are split in five tabs using a `TabHost`. These tabs are the same that are in the PC version. The contents of the different tabs are defined in different layout files which are included in the activity layout, without using the new Android fragments feature. The interface was designed in Eclipse [Fou13]. Currently, the application can be shown in Portuguese and English. The translation of the strings is done using the Android's native resource framework. The strings are stored in XML files and the system uses automatically the correct strings based on the language selected in the device's settings.

The application supports screen rotations and other configuration changes without losing its state. This is important because in Android by default, when the device is rotated, or when another device configuration is changed, the running activity is restarted. The state of most graphical components is retained automatically by Android, but some things have to be done "manually", like the index of the selected tab. The current CPU and its state is, as mentioned before, stored in the `DrMIPS` application class, so it is usually retained. Finally, most of the dialogs used are created and updated in the overrided `onCreateDialog()` and `onPrepareDialog()` methods. This way, the dialogs are managed by Android and are not closed when the device is rotated. All

[2]An activity provides a user interface for a single screen in the application. It could be understood as a maximized window in Android.

dialogs are of the type `AlertDialog`. Alert dialogs are created easily by setting the title, names and behaviours of the buttons, and content, which can be a text message, list of items, custom interface or others.

Like in the PC version, several properties like the theme and last code and CPU files used are remembered on exit and restored on start. This is done by using the native `SharedPreferences` class, which works much like the `Preferences` class of standard Java. This mechanism is also used to check if the application has been upgraded by storing the version code in the preferences. Every time the application is started the current version code is compared with the one stored in the preferences. If it is greater, that means the application has been upgraded. If the old version code is not in the preferences that means this is the first time the application is executed and, thus, is considered as upgraded.

The datapath is displayed in the same way as in the PC version, as shown in Figure 5.15. The datapath, implemented in the `Datapath` class, uses a `RelativeLayout` to display the components in the specified positions and with the specified sizes, all measured using density-independent pixels. Density-independent pixel *(dp)* is a unit that makes the graphical components have approximately the same real size in all devices. The operation that is used to convert the *dp* unit into pixels *(px)*, using the screen's density in dots per inch *(dpi)*, is shown in Equation 5.1.

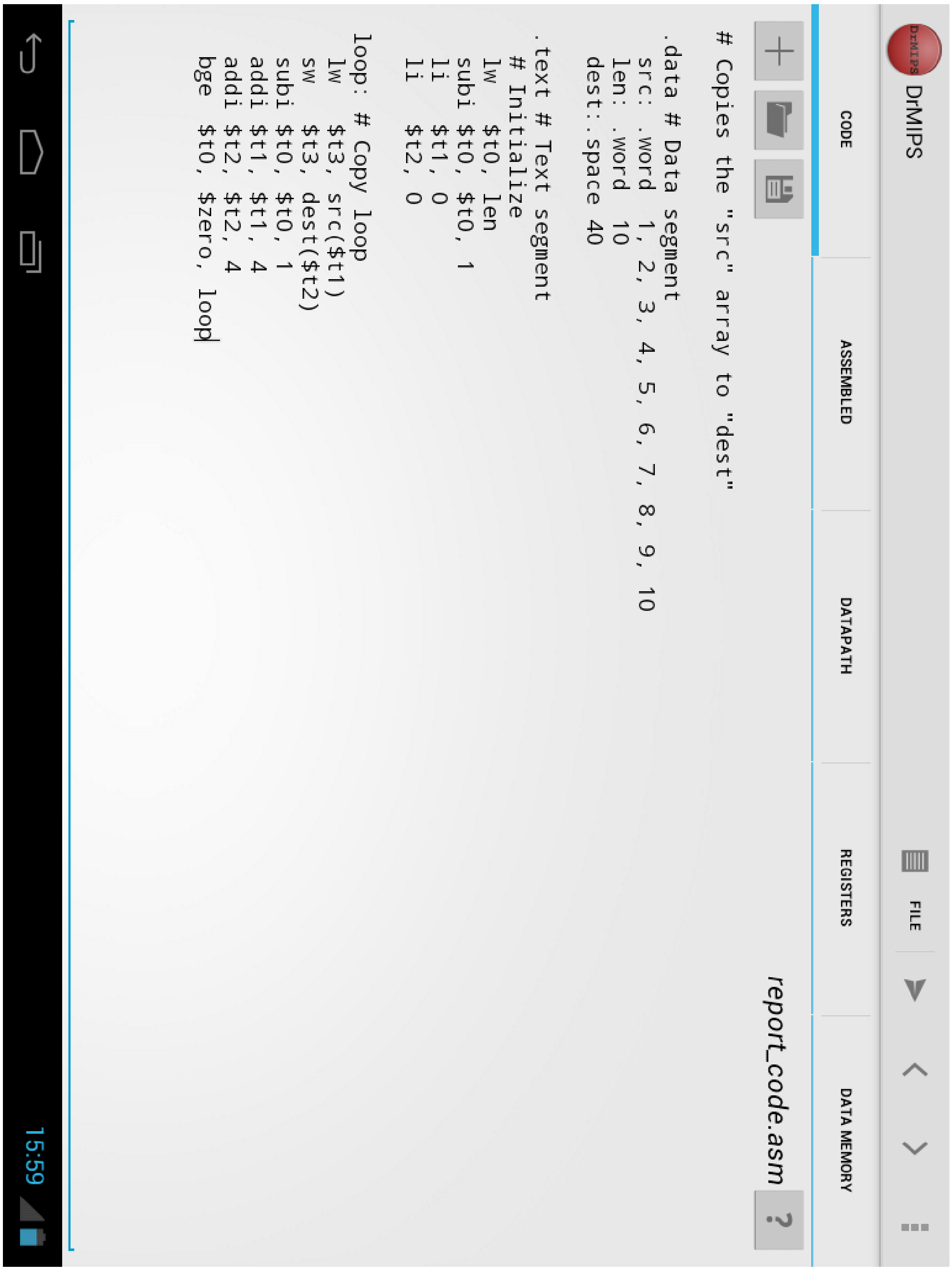
$$px = dp \times \frac{dpi}{160} \tag{5.1}$$

Each component, implemented in the `DatapathComponent` class, is a `TextView` and its name, description, input and output values, or latency can be displayed by pressing the component, as shown if Figure 5.16. Long-pressing it while in performance mode lets the user change the latency. The data tips are also displayed in the same way as in the PC version and, because they are the last user interface components added to the datapath's `RelativeLayout`, they are always shown on top.

The wires are, like in the PC version, defined in the `Wire` class. The implementation of the class is very similar to the one of the PC version, including the colors, but arrow tips are drawn using a filled `Path`. The Android version also includes the *Datapath* menu with the same options: switch between data and performance mode, show or hide control path, data tips and arrow tips, and reset latencies.

This version also displays the instruction or instructions being executed with different colors. They are displayed in a one line table implemented as a `TableLayout` with one `TextView` per instruction. Pressing the instruction presents a small temporary message, a "toast", with the type of the instruction and the values of its fields. Finally, the values can be shown in binary, decimal and hexadecimal formats.

The code editor of this version is a simple `EditText` and not much effort was put into improving it (see Figure 5.17). Touch screens are not very suitable to write code. Furthermore, due to the fact that MIPS assembly programs rely heavily on the dollar sign (`$`) to reference registers, using the default on-screen keyboard may be very annoying, as the dollar sign is usually not on its

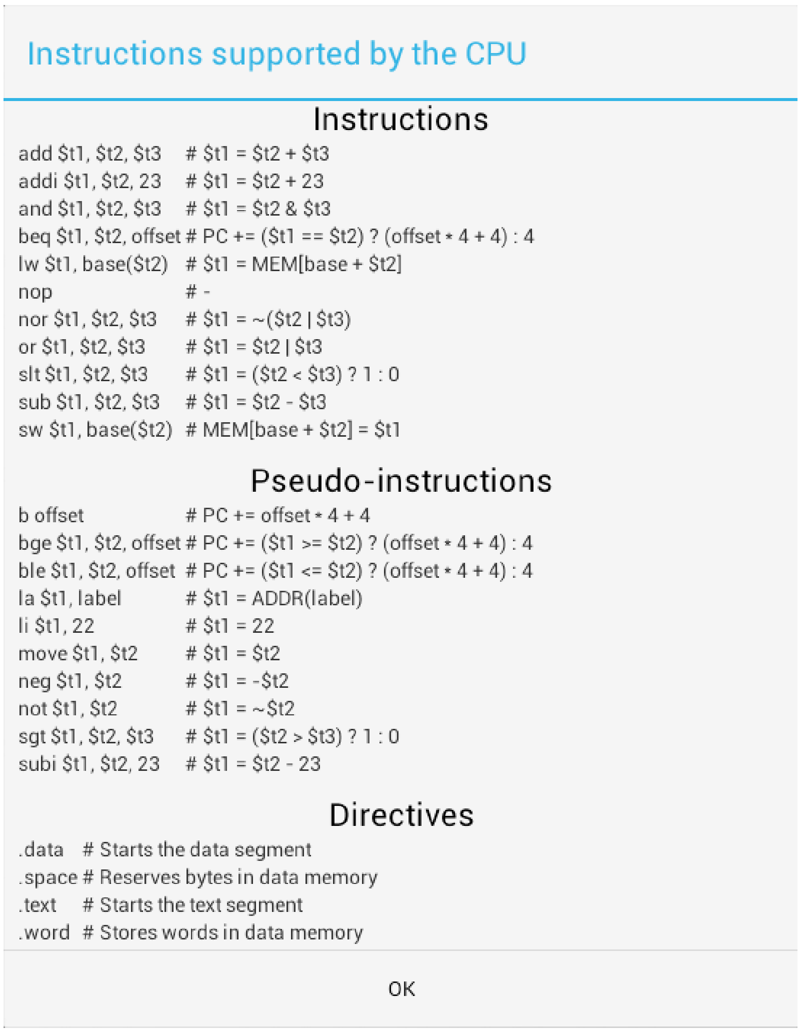


```
# Copies the "src" array to "dest"

.data # Data segment
 src: .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
 len: .word  10
 dest:.space 40

.text # Text segment
 # Initialize
 lw   $t0, len
 subi $t0, $t0, 1
 li   $t1, 0
 li   $t2, 0

loop: # Copy loop
 lw   $t3, src($t1)
 sw   $t3, dest($t2)
 subi $t0, $t0, 1
 addi $t1, $t1, 4
 addi $t2, $t2, 4
 bge  $t0, $zero, loop
```

Figure 5.17: The code tab in Android

first page. Fortunately, alternative on-screen keyboards can be installed. Some of these are very similar to a physical computer keyboard.

The code tab contains some shortcuts to create, open and save the file. It also includes a help button that displays the supported instruction set in a dialog (see Figure 5.18). This dialog lists the instructions, pseudo-instructions and assembler directives that are supported by the currently loaded CPU and what they do.


Instructions supported by the CPU

Instructions

```
add $t1, $t2, $t3    # $t1 = $t2 + $t3
addi $t1, $t2, 23    # $t1 = $t2 + 23
and $t1, $t2, $t3    # $t1 = $t2 & $t3
beq $t1, $t2, offset # PC += ($t1 == $t2) ? (offset * 4 + 4) : 4
lw $t1, base($t2)    # $t1 = MEM[base + $t2]
nop                  # -
nor $t1, $t2, $t3    # $t1 = ~($t2 | $t3)
or $t1, $t2, $t3     # $t1 = $t2 | $t3
slt $t1, $t2, $t3    # $t1 = ($t2 < $t3) ? 1 : 0
sub $t1, $t2, $t3    # $t1 = $t2 - $t3
sw $t1, base($t2)    # MEM[base + $t2] = $t1
```

Pseudo-instructions

```
b offset             # PC += offset * 4 + 4
bge $t1, $t2, offset # PC += ($t1 >= $t2) ? (offset * 4 + 4) : 4
ble $t1, $t2, offset # PC += ($t1 <= $t2) ? (offset * 4 + 4) : 4
la $t1, label        # $t1 = ADDR(label)
li $t1, 22           # $t1 = 22
move $t1, $t2        # $t1 = $t2
neg $t1, $t2         # $t1 = -$t2
not $t1, $t2         # $t1 = ~$t2
sgt $t1, $t2, $t3    # $t1 = ($t2 > $t3) ? 1 : 0
subi $t1, $t2, 23    # $t1 = $t2 - 23
```

Directives

```
.data   # Starts the data segment
.space  # Reserves bytes in data memory
.text   # Starts the text segment
.word   # Stores words in data memory
```

Figure 5.18: The code help dialog

The assembled code, registers and data memory tables are similar to the ones in the PC version. Each table is implemented as a `TableLayout`. Each row is a `TableRow` and each cell is a `TextView`. The first row is the header, which is fixed and never removed when the table is cleared. The rows are highlighted by changing directly the colors of the `TextView` objects in them. Pressing an instruction in the assembled code table presents a temporary message with the instruction's type and field values. Long-pressing a row in the registers or data memory table displays a dialog to allow the value of the respective register or memory address to be changed. The tables can be scrolled both vertically and horizontally, so they can fit the screen without becoming “deformed” in smaller screens.

The application was targeted for Android 4.2.2 Jelly Bean. The minimum Android version currently supported is 2.3.3 Gingerbread. According to the Android Dashboards [Goo13], using Android 2.3.3 as minimum version allows the application to be supported by around 95% of the devices. Supporting lower versions would require changes to the code so, to save time, the minimum version was kept at 2.3.3. Support for lower versions would be useful though, as the number of supported Android devices would increase.

5.3 Concluding Remarks

This chapter presented the graphical user interfaces of both PC and Android versions. It also discussed the differences between the two versions and how they were implemented. The two versions are very similar, both in terms of features and terms of interface layout, and they both use the same internal simulator logic, discussed in the previous chapter. Nevertheless, the PC version is the most complete, containing a complete code editor with syntax-highlighting and auto-complete and including a mode where the contents are split in internal windows.

The next chapter discusses some examples of how the tool can be used by students to better understand several topics of computer architecture.

Chapter 6

Usage Examples

The purpose of this tool is to help students to better understand several topics of computer architecture. Some examples of how this tool can be used in that context are presented and discussed in this chapter. These examples are typical computer architecture exercises. The processes to solve them with the simulator and to understand the resolution are explained.

6.1 First Example

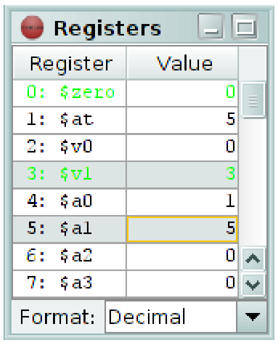
For the program in Listing 6.1, determine the values of the registers `$3` and `$5` after the execution of line 4 and also of line 7 in the second iteration. Determine the final values of those registers as well. Consider that the initial value of register `$1` is `5`.

```
1       add $2, $0, $0
2       li  $3, 3
3       li  $4, 1
4       add $5, $1, $2
5 next: beq $3, $0, out
6       sub $3, $3, $4
7       add $5, $5, $5
8       beq $0, $0, next
9 out:
```

Listing 6.1: Code of the first example

To solve this in the simulator, the user starts by loading the unicycle CPU, inserting the given code in the code editor and assembling the program. To initialize the register `$1` with the value `5`, the user can add an instruction above the given code to do so, like `li $1, 5`, before assembling. Another method is to double-click on the `$1` register in the registers table and edit the value. This method doesn't require any changes to the code, but the edited value is lost every time the program is assembled, except if the *Reset data before assembling* option is disabled in the *Execute* menu. The same method could be used to edit values in the data memory. The user can then execute the program step-by-step, analysing the state of the CPU in each step.

The values of the registers after the execution of the fourth line are shown in Figure 6.1. On that clock cycle, `$3` has the value `3` and `$5` has the value `5`.



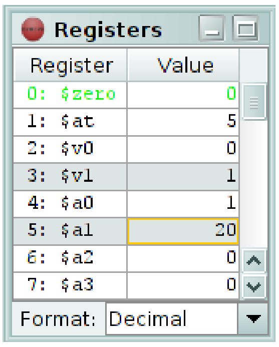
Registers

Register	Value
0: $zero	0
1: $at	5
2: $v0	0
3: $v1	3
4: $a0	1
5: $a1	5
6: $a2	0
7: $a3	0

Format: Decimal

Figure 6.1: Registers after line 4 of the first example

After the execution of the seventh line in the second iteration, registers `$3` and `$5` have the values `1` and `20` respectively, as shown in Figure 6.2.



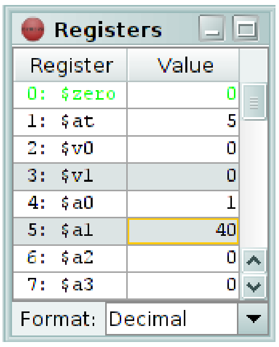
Registers

Register	Value
0: $zero	0
1: $at	5
2: $v0	0
3: $v1	1
4: $a0	1
5: $a1	20
6: $a2	0
7: $a3	0

Format: Decimal

Figure 6.2: Registers after line 7 of the first example in the second iteration

The final values of the registers are displayed in Figure 6.3. Register `$3` has the value `0`, while register `$5` has the value `40`.



Registers

Register	Value
0: $zero	0
1: $at	5
2: $v0	0
3: $v1	0
4: $a0	1
5: $a1	40
6: $a2	0
7: $a3	0

Format: Decimal

Figure 6.3: Registers at the end of the first example

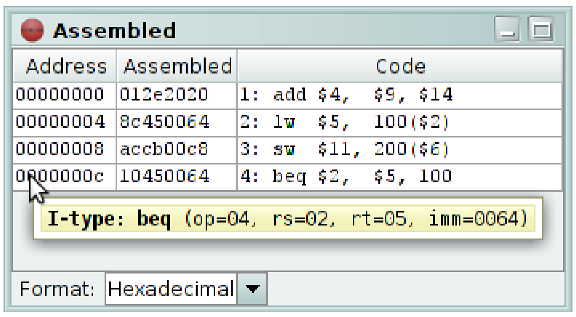
Obviously, the simulator is more useful in problems of this kind, where the programs are more extensive or have a loop with a higher number of iterations.

6.2 Second Example

Determine the resulting machine code and the values of all control signals for the following instructions:

1. `add $4,  $9, $14`
2. `lw  $5,  100($2)`
3. `sw  $11, 200($6)`
4. `beq $2,  $5, 100`

Finding the resulting machine code of the instructions is easy using the simulator. To do so, the user must simply insert the instructions in the code editor and assemble them. The assembled code table will then display the internal machine code for the instructions, as shown in Figure 6.4. The user may also hover the mouse cursor over any instruction in the table to know of what type the instruction is and to see the values of its fields, as shown in the figure for the `beq` instruction, to better understand the machine code representation.


Assembled

Address	Assembled	Code
00000000	012e2020	1: add $4, $9, $14
00000004	8c450064	2: lw $5, 100($2)
00000008	accb00c8	3: sw $11, 200($6)
0000000c	10450064	4: beq $2, $5, 100

I-type: beq (op=04, rs=02, rt=05, imm=0064)

Format: Hexadecimal

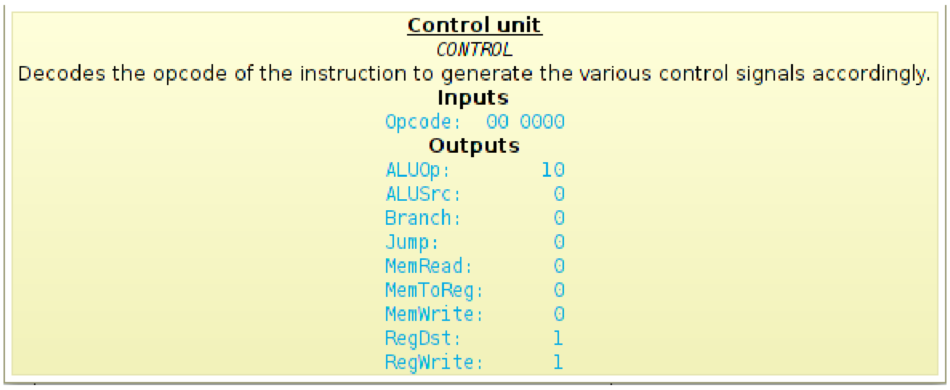
Figure 6.4: The machine code of the instructions of the second example

The internal representation in machine code of the given instructions is shown in Table 6.1. The type of each instruction is also displayed. The machine code of each instruction is shown in hexadecimal format.

Instruction	Machine code (hex)	Type
`add $4,  $9, $14`	012e2020	R
`lw  $5,  100($2)`	8c450064	I
`sw  $11, 200($6)`	accb00c8	I
`beq $2,  $5, 100`	10450064	I

Table 6.1: Machine code and types of the instructions of the second example

Finding the values of the control signals for the execution of each instruction is also easy. The user assembles the instructions and then executes them, one at a time. For each instruction, the user checks the values of the outputs of the control unit by hovering the mouse cursor over the control unit in the graphical datapath and reading the tooltip that is displayed. The tooltips displayed for the `add`, `lw`, `sw` and `beq` instructions are shown in figures 6.5, 6.6, 6.7 and 6.8, respectively.



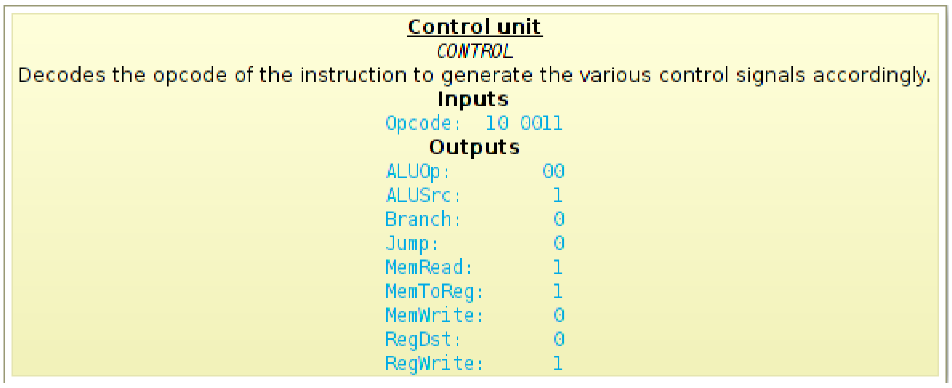
Control unit
CONTROL
Decodes the opcode of the instruction to generate the various control signals accordingly.
Inputs

```
Opcode:   00 0000
```

Outputs

```
ALUOp:        10
ALUSrc:        0
Branch:        0
Jump:          0
MemRead:       0
MemToReg:      0
MemWrite:      0
RegDst:        1
RegWrite:      1
```

Figure 6.5: The values in binary of the control signals for the *add* instruction



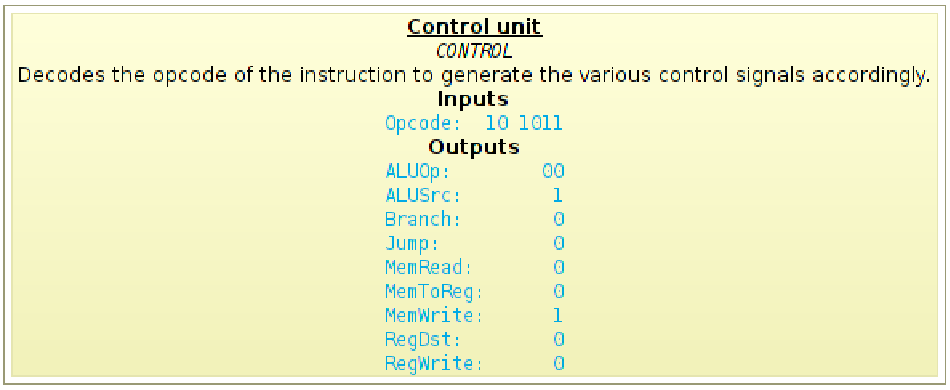
Control unit
CONTROL
Decodes the opcode of the instruction to generate the various control signals accordingly.
Inputs

```
Opcode:   10 0011
```

Outputs

```
ALUOp:        00
ALUSrc:        1
Branch:        0
Jump:          0
MemRead:       1
MemToReg:      1
MemWrite:      0
RegDst:        0
RegWrite:      1
```

Figure 6.6: The values in binary of the control signals for the *lw* instruction



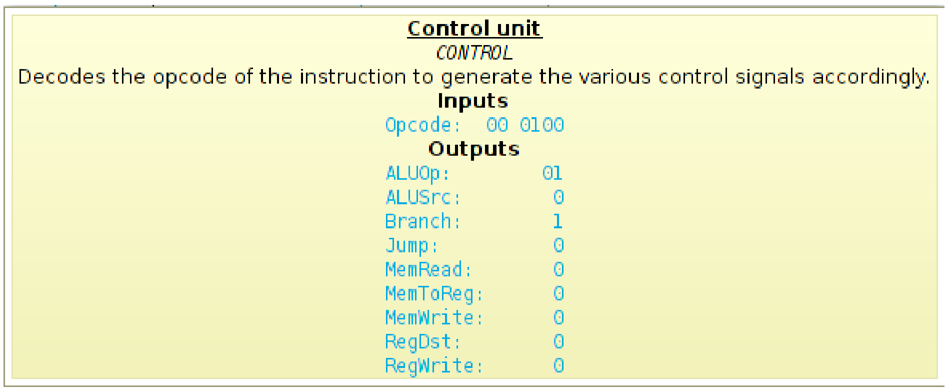
Control unit
CONTROL
Decodes the opcode of the instruction to generate the various control signals accordingly.
Inputs

```
Opcode:   10 1011
```

Outputs

```
ALUOp:        00
ALUSrc:        1
Branch:        0
Jump:          0
MemRead:       0
MemToReg:      0
MemWrite:      1
RegDst:        0
RegWrite:      0
```

Figure 6.7: The values in binary of the control signals for the *sw* instruction



Control unit
CONTROL
Decodes the opcode of the instruction to generate the various control signals accordingly.
Inputs

```
Opcode:  00 0100
```

Outputs

```
ALUOp:       01
ALUSrc:       0
Branch:       1
Jump:         0
MemRead:      0
MemToReg:     0
MemWrite:     0
RegDst:       0
RegWrite:     0
```

Figure 6.8: The values in binary of the control signals for the *beq* instruction

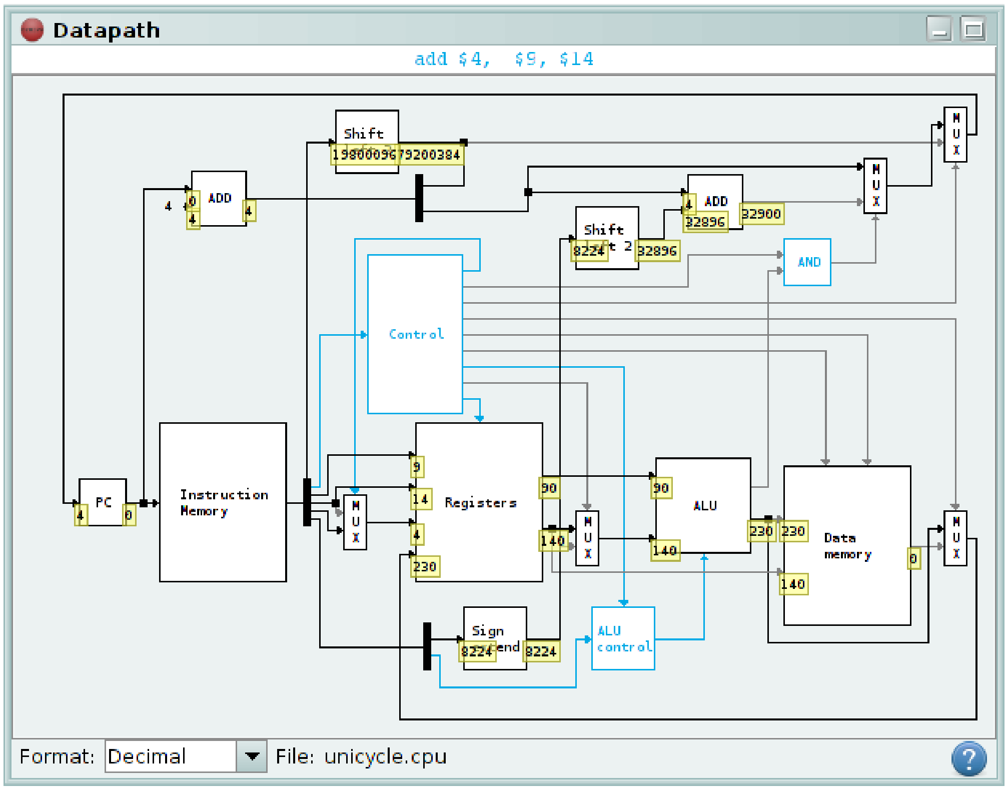
It is important to note that the values of the control signals depend only on the `opcode` of the instruction, so the arguments of the instructions don't affect them. The values of the control signals in binary for the given instructions are summarized in Table 6.2.

Signal	`add`	`lw`	`sw`	`beq`
ALUOp	10	00	00	01
ALUSrc	0	1	1	0
Branch	0	0	0	1
Jump	0	0	0	0
MemRead	0	1	0	0
MemToReg	0	1	0	0
MemWrite	0	0	1	0
RegDst	1	0	0	0
RegWrite	1	1	0	0

Table 6.2: Values of the control signals for the instructions of the second example

Besides helping determine the values of the control signals, the simulator can help to understand why the signals have these values and how the execution cycle works. Using the instruction `add $4,  $9, $14` to illustrate this, the user would insert this instruction in the code editor and assemble it. As the instruction accesses registers, the user would also edit the values of these registers, doing as explained before. Considering that the values for the registers `$4`, `$9` and `$14` would be `40`, `90` and `140`, the datapath would look like Figure 6.9.

The user can now analyse the datapath to understand how the instruction is executed. For example, the user can look at the program counter and see that its value increases by four. Following the path at the top of the datapath, he/she can see that an adder increases the current value of the program counter, and that this new value is selected by both multiplexers. The user can see how the register bank reads the correct registers and outputs the correct values. The user can see how the ALU calculates the correct result and how that result ends up in the `WriteData` input of the register bank. He/she can also see that the data memory isn't used because all inputs and outputs are gray. Being able to understand how the datapath works is very important for a student.



add $4, $9, $14

Format: Decimal File: unicycle.cpu

Figure 6.9: The datapath for the instruction *add $4, $9, $14*

6.3 Third Example

Consider the latencies of the components of the unicycle CPU specified in Table 6.3. Determine the critical path of the datapath and the minimum clock period possible. Additionally, determine the maximum amount of time the control unit can use to generate the `MemRead` and `ALUOp` control signals without degrading the CPU's performance.

	Inst. Mem.	Control Unit	Reg. bank	ALU	ALU Control	Data Mem.
Latency (*ps*)	200	50	100	100	50	200

Table 6.3: Latencies of the components for the third example

To discover the critical path of the datapath in the simulator with the given parameters, the user starts by loading the unicycle CPU and enabling the *Performance Mode* in the *Datapath* menu. Then the user edits the latencies of the components to comply with the given parameters by double-clicking on the components and inserting the new latencies. The user should make sure that the specified components have the given latencies and that the others have zero latency. The critical path will then be visible in the graphical datapath, as shown in Figure 6.10.

As shown in the figure, the critical path ends in the register bank. So, to discover the minimum clock period possible, the user hovers the mouse cursor over the register bank and consults the tooltip (see Figure 6.11). The input with the highest accumulated latency is `WriteData`, which

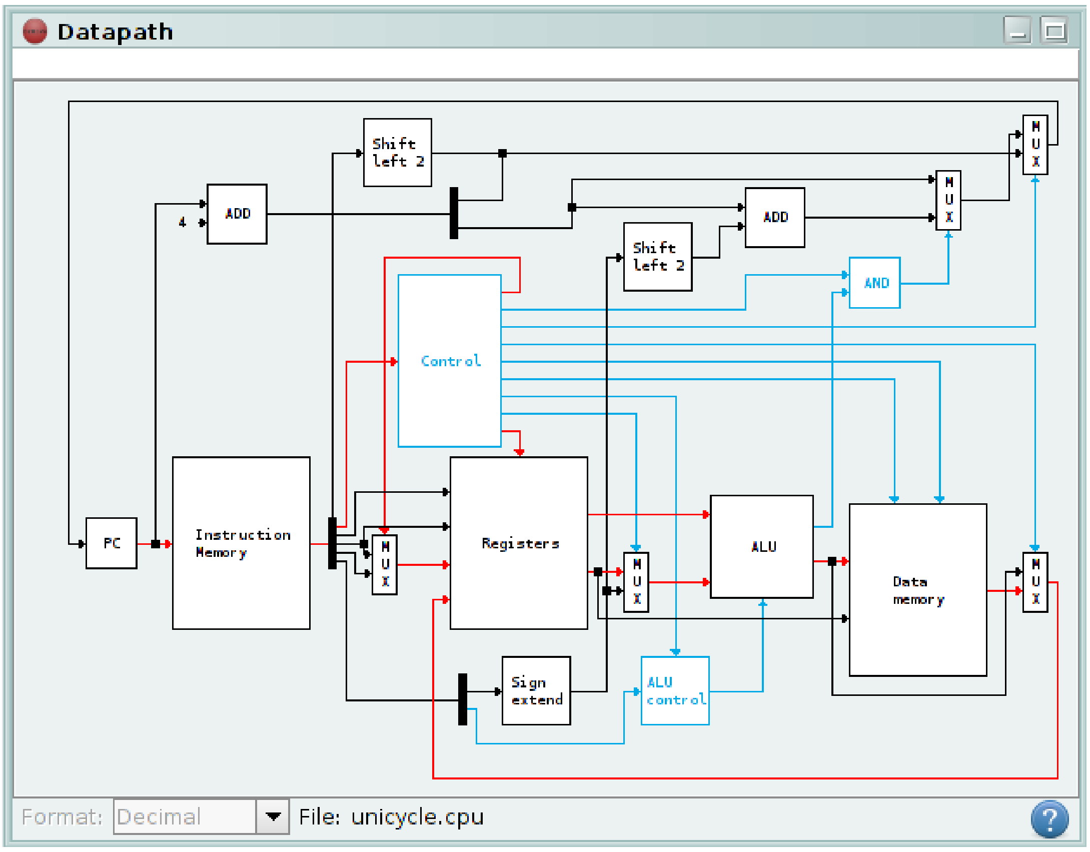
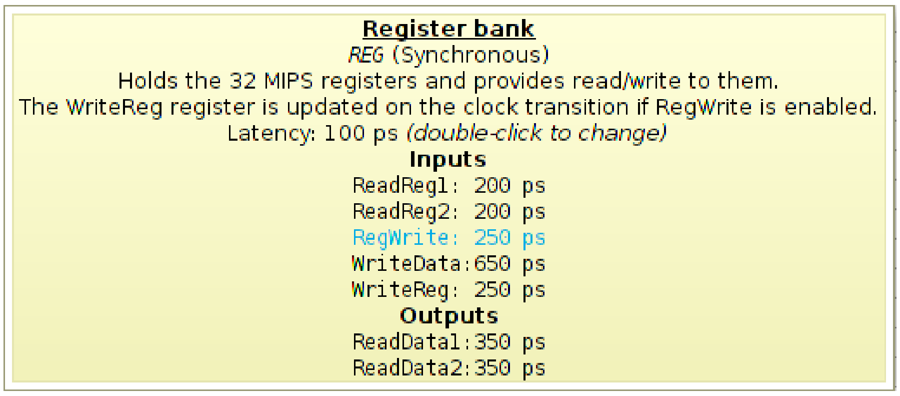


Figure 6.10: Critical path of the datapath in the third example

is where the critical path ends. The value of the accumulated latency at that input, 650 *ps*, is the minimum clock period.



Register bank
REG (Synchronous)
Holds the 32 MIPS registers and provides read/write to them.
The WriteReg register is updated on the clock transition if RegWrite is enabled.
Latency: 100 ps *(double-click to change)*
Inputs

```
ReadReg1:  200 ps
ReadReg2:  200 ps
RegWrite:  250 ps
WriteData:650 ps
WriteReg:  250 ps
```

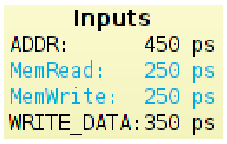
Outputs

```
ReadData1:350 ps
ReadData2:350 ps
```

Figure 6.11: Tooltip of the register bank in the third example

To determine the maximum amount of time the control unit can use to generate the `MemRead` control signal it is necessary to analyse the datapath. That control signal is used by the data memory. The data memory is in the critical path, so the user must consult the accumulated latencies at the inputs of the component (see Figure 6.12). The highest value is 450 *ps* and the value at the

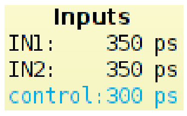
`MemRead` input is 250 *ps*, therefore the answer is $450-250+50=250ps$, where 50 is the latency already defined in the control unit.



```
Inputs
ADDR:      450 ps
MemRead:   250 ps
MemWrite:  250 ps
WRITE_DATA:350 ps
```

Figure 6.12: Accumulated latencies at the inputs of the data memory in the third example

Determining the maximum time to generate the `ALUOp` control signal is trickier. That signal is used by the ALU control unit, which is not in the critical path. However, it is connected directly to the ALU, which is in the critical path, so the user can consult the accumulated latencies at the inputs of that component (see Figure 6.13). The highest value is 350 *ps* and the value at the input connected to the ALU control is 300 *ps*. Remembering the latencies of the control unit and ALU control, which are both 50 *ps*, the answer is $350-300+50+50=150ps$.



```
Inputs
IN1:    350 ps
IN2:    350 ps
control:300 ps
```

Figure 6.13: Accumulated latencies at the inputs of the ALU in the third example

6.4 Fourth Example

For the program in Listing 6.2, identify all the hazards that occur in the pipeline version of the CPU, indicating which ones are solved by forwarding and which ones are solved by stalling.

```
add $3, $4, $2
sub $5, $3, $1
lw  $6, 200($3)
add $7, $3, $6
```

Listing 6.2: Code of the fourth example

The simulator can help the user to identify the hazards and understand why they occur. The user starts by loading the pipeline CPU, inserting the given code in the code editor and assembling the program. Then, he/she executes the program step-by-step, keeping attention to some elements of the graphical datapath. When one or both of the outputs of the forwarding unit is active (i.e. is blue), a forwarding is occurring. When the output of the hazard detection unit is active, a stall is occurring.

The first hazard occurs in the fourth clock cycle and is solved by forwarding. As shown in Figure 6.14, one of the outputs of the forwarding unit is active. In this case, the value for the first input of the ALU is forwarded from the `EX/MEM` pipeline register in the `MEM` stage. The forwarding unit has the value `3` both at an input on the left and at an input on the right, so the hazard is in register `$3`. The user can see the instructions that are in the `EX` and `MEM` stages by looking at the top of the datapath tab or window (not visible in the figure). The instructions causing this hazard are `add $3, $4, $2` and `sub $5, $3, $1`.

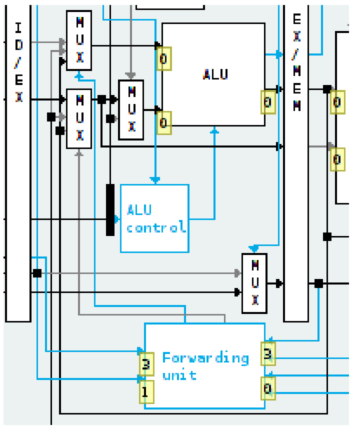


Figure 6.14: First hazard in the code of the fourth example

The second hazard occurs in the next clock cycle and is also solved by forwarding. As shown in Figure 6.15, the hazard is again in register `$3`. This time, the value for the first input of the ALU is forwarded from the `WB` stage, thus the instructions causing the hazard are `add $3, $4, $2` and `lw $6, 200($3)`.

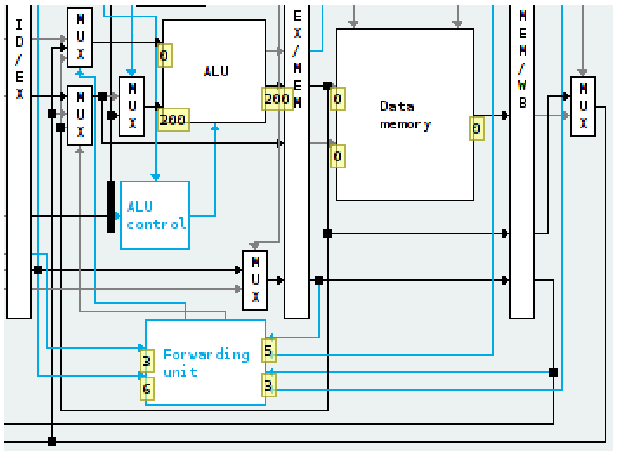


Figure 6.15: Second hazard in the code of the fourth example

Another hazard occurs in the same cycle. This one is solved by a stall. As shown in Figure 6.16, the output of the hazard detection unit is active. The unit has the value `6` both at an input on the left and at an input on the right, so the hazard is in register `$6`. The instructions causing the hazard are `lw $6, 200($3)` and `add $7, $3, $6`. By looking at the inputs of some components, specifically the program counter and the first two pipeline registers, the user can understand how the stall is introduced in the pipeline.

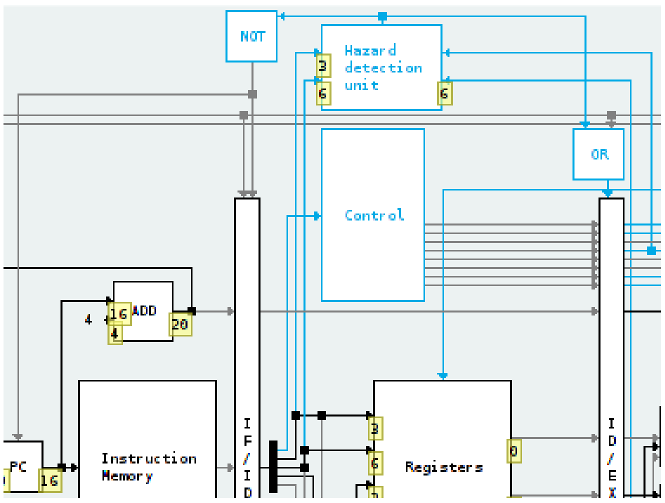


Figure 6.16: Third hazard in the code of the fourth example

The last hazard occurs two clock cycles later and is solved by forwarding. As shown in Figure 6.17, the hazard is in register `$6` and the value for the second input of the ALU is forwarded from the `WB` stage. The instructions causing the hazard are `lw $6, 200($3)` and `add $7, $3, $6`.

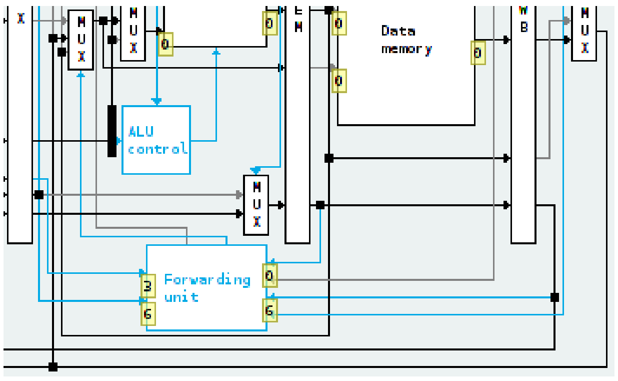
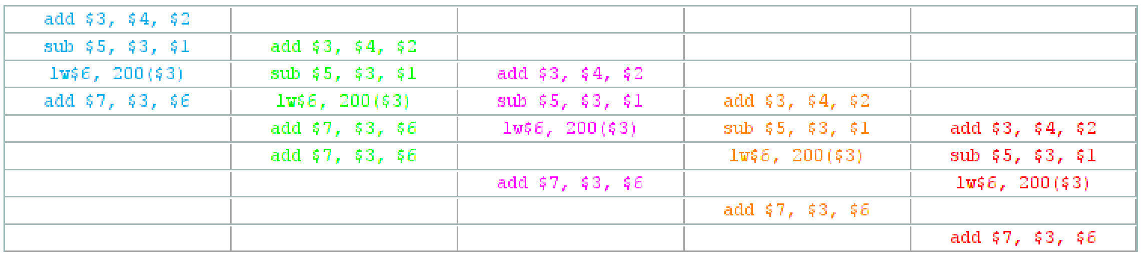


Figure 6.17: Last hazard in the code of the fourth example

The simulator also includes two variations of the pipeline datapath: one without any sort of hazard detection and resolution and another that only implements forwarding. Using these variations, the user can understand what happens without hazard detection and why it is necessary.

Even though the simulator currently doesn't display timing diagrams, the instructions that are currently in each stage of the pipeline are shown at the top of the datapath tab or window. The instructions being executed are also highlighted in the assembled code tab/window. The states of the pipeline shown in the simulator in each clock cycle are shown in Figure 6.18, condensed into a single figure. The effect of the stall is very visible in the figure.



add $3, $4, $2				
sub $5, $3, $1	add $3, $4, $2			
lw$6, 200($3)	sub $5, $3, $1	add $3, $4, $2		
add $7, $3, $6	lw$6, 200($3)	sub $5, $3, $1	add $3, $4, $2	
	add $7, $3, $6	lw$6, 200($3)	sub $5, $3, $1	add $3, $4, $2
	add $7, $3, $6		lw$6, 200($3)	sub $5, $3, $1
		add $7, $3, $6		lw$6, 200($3)
			add $7, $3, $6	
				add $7, $3, $6

Figure 6.18: State of the pipeline in each clock cycle of the fourth example

6.5 Concluding Remarks

Some examples of how the simulator can be used to help students to better understand several topics of computer architecture were discussed in this chapter. These examples were demonstrated using the PC version of the simulator, but the Android version could also be used in a similar way.

As presented, DrMIPS can be used to solve several computer architecture problems and understand their resolution. Some of these problems could be solved with other simulators, but each simulator could only be used for a small set of problems. Problems involving processor performance, discussed in the third example, cannot be solved with any of the tools discussed in Chapter 2. Other problems, like the ones involving processor pipelining, are harder to solve and understand with other simulators. Overall, DrMIPS is more versatile than other simulators while also being easy to use.

Chapter 7

Conclusion

There are many educational CPU simulators, as discussed in Chapter 2. However, they are not very versatile. The existing simulators usually can only simulate either the unicycle or the pipeline version of the CPU, provide very simple code editors and their simulated datapaths are not configurable.

Also important to note is that none of these simulators have a version for Android tablets. As discussed in Chapter 3, tablets are becoming a very popular platform and Android is one of the most popular operating systems for both tablets and smartphones. Thus, an Android MIPS simulator is certainly innovative.

To aid the students and teachers, an educational MIPS simulator, called DrMIPS, was developed. As discussed in chapters 4 and 5, the simulator is fairly versatile, intuitive and configurable. It supports both unicycle and pipeline versions of the CPU, displays the datapath graphically and has a "performance mode" showing the latencies and critical path. It was developed for the PC and for Android devices, and both versions are very similar. The development of the tool for two quite different platforms implied additional effort, but the way the code was structured reduced it considerably.

7.1 Objectives Accomplishment

The main objective of this work, which was to create a MIPS simulator to support computer architecture teaching and learning, was accomplished. The more specific objectives of the work were also achieved. The tool simulates both unicycle and pipeline versions of the processor and the pipeline version includes hazard detection and resolution. It allows the step-by-step execution of an assembly program while displaying detailed information about the CPU, including a graphical representation of the datapath. The performance of the processor is simulated and its critical path can be viewed. The datapaths can be configured and additional ones can be created. And finally, the tool was developed for both computers and Android tablets, as planned.

Besides those initial objectives, some additional ones were implemented. The most important one is that the instruction sets used by the datapaths can also be configured. Other additions include a dark theme, support for multiple languages and, for the PC version, two different modes to layout the window contents and a complete code editor. The end result seems to be very positive, but only real testing with real students can reveal how much the tool is effective in helping students to learn about computer architecture.

The simulator is, in a way, an integrated environment that aggregates several features that are found scattered through the other existing tools, plus some additional features. The simulators that were discussed in Chapter 2 are compared with the developed simulator in Table 7.1. The table shows that DrMIPS is more versatile than the others. It is one of the few that contains a code editor with syntax-highlighting (in the PC version), that supports both the unicycle and pipeline implementations, and that allows the configuration of the datapath. It is also the only one that provides a version for Android, and that shows the latencies of the components and the critical path of the CPU.

Syscalls and floating point operations are, at least for the moment, not supported, so simulators like MARS and SPIM are more adequate to debug real and complete assembly programs, which is not the focus of this work. Timing diagrams for the pipelined datapaths are also not displayed. The release of the code as open-source is planned.

	SPIM	MARS	ProcSim	MIPS-Datapath	WebMIPS	EduMIPS64	DrMIPS
Open-source	Yes	Yes	No	Yes	Yes	Yes	Planned
Code editor	No	Yes	Yes	Yes	Yes	No	Yes
Editor syntax-highlighting	No	Yes	No	No	No	No	PC only
Unicycle simulation	Yes	Yes	Yes	Yes	No	No	Yes
Pipeline simulation	No	No	No	Partial	Yes	Yes	Yes
Floating point support	Yes	Yes	No	No	No	Yes	No
Syscall support	Yes	Yes	No	No	No	Yes	No
Edit data during execution	Yes	Yes	No	No	No	Yes	Yes
Datapath visualization	No	No	Yes	Yes	Yes	Simple	Yes
Datapath configuration	No	No	Yes	No	No	No	Yes
Timing diagrams	No	No	No	No	No	Yes	No
Latencies & critical path	No	No	No	No	No	No	Yes
Native Android version	No	No	No	No	No	No	Yes
Written in	C++,Qt	Java	Java	C++	ASP	Java	Java

Table 7.1: Comparison of the related tools with DrMIPS

In terms of performance, the simulator runs fairly well. The PC version of the simulator starts in a few seconds and the simulation is executed without any considerable delays in the computers of today. The Android version is a bit slower, and on lower-end devices some small delays can be experienced, especially when using a pipelined CPU. But the application is still very usable in those cases.

7.2 Future Work

The simulator can be improved in several ways. One useful improvement would be the inclusion of timing diagrams when simulating pipelined datapaths. These diagrams help to understand how pipelining works and how hazards affect it. The different stages could be represented by simple colored rectangles or squares. However, the number of clock cycles displayed in the diagram should be limited to avoid slowing the interface, especially on Android and when running all the instructions at once. Some additional performance statistics, like CPI, time and number of cycles spent to execute the program would also be useful, as the calculation of these statistics is also taught in computer architecture courses.

Another useful improvement would be the inclusion of a graphical interface to create and edit CPU files. The CPUs are defined in JSON files and, at the moment, must be created and edited manually with a text editor. The most difficult part of creating the graphical editor would probably be the interface to create/edit components. Each type of component has different properties and there are quite a few different types of components.

The simulator supports the instructions detailed in the reference book [PH05] for the presented datapaths, plus some additional instructions. The supported instruction set could be expanded with more instructions. Two important ones are `jal` and `jr`, which allow the creation of subroutines. Shift instructions are another example of important instructions. These, and some other instructions, can be added by creating new CPU and instruction set files without changing the code.

As mentioned in the end of Section 5.2, the minimum Android version supported by the Android application is 2.3.3. Even though most devices are already supported, it would be useful to support as many devices as possible by lowering the minimum supported version. The code would have to suffer changes, though. Other improvements include optimizing the code, improving the descriptions of the components and changing the latencies of the components in the provided datapaths to more realistic values.

References

[Ara10] Kenzo Araujo. Bem Vindo ao Projeto EKS-MIPS, June 2010. Accessed on June 6, 2013. http://eks.kenzo.inf.br.

[Atl] Atlassian. Free source code hosting for Git and Mercurial by Bitbucket. Accessed on May 30, 2013. http://bitbucket.org.

[BGM04] Irina Branovic, Roberto Giorgi, and Enrico Martinelli. WebMIPS: A New Web-Based MIPS Simulation Environment for Computer Architecture Education. *Workshop on Computer Architecture Education, 31st International Symposium on Computer Architecture*, 2004. Accessed on June 6, 2013. http://www4.ncsu.edu/~efg/wcae/2004/submissions/giorgi.pdf.

[Bro02a] Mats Brorsson. MipsIt – A Simulation and Development Environment Using Animation for Computer Architecture Education. *Workshop on Computer Architecture Education, 29th International Symposium on Computer Architecture*, 2002. Accessed on June 6, 2013.

[Bro02b] Mats Brorsson. MipsIt – Development Environment and Simulation Software, April 2002. Accessed on June 6, 2013. http://www.bostream.nu/mats.brorsson/mipsit.

[But11] Margaret Butler. Android: Changing the Mobile Landscape. *IEEE Pervasive Computing*, 10(1):4 – 7, January-March 2011.

[Con13] Software Freedom Conservancy. Git, 2013. Accessed on June 6, 2013. http://git-scm.com.

[dC06] Universidad de Córdoba. UCO.MIPS, 2006. Accessed on June 6, 2013. http://www.uco.es/~p02pasea/ucomipsim.

[Dig13] Digia. Qt, 2013. Accessed on June 6, 2013. http://qt.digia.com.

[edu] EduMIPS64 Students Questionnaire. Accessed on June 6, 2013. http://www.diit.unict.it/users/spadaccini/edumips64-survey.html.

[Fif13a] Fifesoft. AutoComplete | Fifesoft, 2013. Accessed on May 27, 2013. http://fifesoft.com/autocomplete.

[Fif13b] Fifesoft. RSyntaxTextArea | Fifesoft, 2013. Accessed on May 27, 2013. http://fifesoft.com/rsyntaxtextarea.

[Fou13] The Eclipse Foundation. Eclipse - The Eclipse Foundation open source community website, 2013. Accessed on June 6, 2013. http://www.eclipse.org.

[FPC06] Ariane Felix, Christiane Pousa, and Milene Carvalho. DIMIPSS: Um simulador didático e interativo do MIPS. *Workshop sobre Educação em Arquitetura de Computadores*, pages 49 – 52, 2006. Accessed on June 6, 2013. http://www.ppgee.pucminas.br/weac/2006/PDF/WEAC-2006-Artigo-08.pdf.

[Gar05] James Garton. ProcessorSim – A Visual MIPS R2000 Processor Simulator, 2005. Accessed on June 6, 2013. http://jamesgart.com/procsim.

[GC] Andrew Gascoyne-Cecil. MIPS-Datapath. Accessed on June 6, 2013. http://mi.eng.cam.ac.uk/~ahg/MIPS-Datapath.

[GLPHB08] J. Gómez-Luna, A. Palacios, E. Herruzo, and J.I. Benavides. UCO.MIPSIM: PIPELINED COMPUTER SIMULATOR FOR TEACHING PURPOSES. 2008. Accessed on June 6, 2013. http://e-spacio.uned.es:8080/fedora/get/taee:congreso-2008-1043/S2C03.pdf.

[Gooa] Google. Android – Discover Android. Accessed on June 6, 2013. http://www.android.com/about.

[Goob] Google. Android SDK | Android Developers. Accessed on June 6, 2013. http://developer.android.com/sdk/index.html.

[Goo13] Google. Dashboards | Android Developers, June 2013. Accessed on June 14, 2013. http://developer.android.com/about/dashboards/index.html.

[Hag] Michael Hagen. JTattoo. Accessed on May 31, 2013. http://www.jtattoo.net.

[Jam95] Tariq Jamil. RISC versus CISC. *IEEE Potentials*, 14(3):13 – 16, August/September 1995.

[JdSGM07] Nelson Gonçalves Junior, Renata Lopes da Silva, Ronaldo Gonçalves, and João Martini. R10k: Um Simulador de Arquitetura Superescalar. *Workshop sobre Educação em Arquitetura de Computadores*, pages 23 – 30, 2007. Accessed on June 6, 2013. http://www.ppgee.pucminas.br/weac/2007/PDF/WEAC-2007-Artigo-04.pdf.

[Jim13] Jimmat. Assembly Emulator - Android Apps on Google Play, May 2013. Accessed on May 29, 2013. http://play.google.com/store/apps/details?id=gr.ntua.ece.assembly.emulator.

[jsoa] Introducing JSON. Accessed on May 24, 2013. http://www.json.org.

[jsob] JSON in Java. Accessed on May 31, 2013. http://www.json.org/java/index.html.

[KBH11] Md Tahsin Kabir, Mohammad Tahmid Bari, and Abul L. Haque. ViSiMIPS: Visual Simulator of MIPS32 Pipelined Processor. *The 6^{th} International Conference on Computer Science & Education*, pages 788 – 793, August 2011. Accessed on June 6, 2013.

[Koc08] Çetin Koca. MIPSim - MIPS Assembly Language Simulator, 2008. Accessed on June 6, 2013. http://www.mipsim.com/mipsim.

[Lar] James Larus. SPIM: A MIPS32 Simulator. Accessed on June 6, 2013. http://spimsimulator.sourceforge.net.

[Lar90] James Larus. SPIM S20: A MIPS R2000 Simulator. Technical report, Computer Sciences Department, University of Wisconsin, 1990. Accessed on June 6, 2013. http://phoenix.goucher.edu/~kelliher/f2005/cs220/spim.pdf.

[MFN09] Marcelo Matos, Ramon Figueiredo, and Davidson Nogueira. PS - CAS MIPS, 2009. Accessed on June 6, 2013. https://sites.google.com/site/pscasmips.

[MVP09] Davidson Maia, Marcelo Vieira, and Ramon Pessoa. PS – CAS MIPS: Um Simulador De Pipeline Do Processador MIPS 32 Bits Para Estudo de Arquitetura de Computadores. *Workshop sobre Educação em Arquitetura de Computadores*, pages 56 – 59, 2009. Accessed on June 6, 2013. http://www.ppgee.pucminas.br/weac/2009/PDF/WEAC-2009-Artigo-12.pdf.

[Ora] Oracle. Learn about Java Technology. Accessed on June 6, 2013. http://www.java.com/en/about.

[Ora13] Oracle. Welcome to NetBeans, 2013. Accessed on June 6, 2013. http://netbeans.org.

[Per09] João Luís Silva Campos Pereira. Educational package based on the MIPS architecture for FPGA platforms. Master Thesis, Faculdade de Engenharia da Universidade do Porto, June 2009. Accessed on June 6, 2013. http://repositorio-aberto.up.pt/bitstream/10216/59975/1/000135086.pdf.

[PH05] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware/Software Interface*. Morgan Kaufmann, 3rd edition, 2005.

[PLT13] PLT. The Racket Language, 2013. Accessed on May 31, 2013. http://racket-lang.org.

[PSP$^+$12] Davide Patti, Andrea Spadaccini, Maurizio Palesi, Fabrizio Fazzino, and Vincenzo Catania. Supporting Undergraduate Computer Architecture Students Using a Visual MIPS64 CPU Simulator. *IEEE Transactions on Education*, 55(3):406 – 411, August 2012. Accessed on June 6, 2013.

[SAA10] Elder Schemberger, Kenzo Araujo, and Sidgley Andrade. Eksmips - um simulador para o processador mips. June 2010. Accessed on June 6, 2013. http://www.kenzo.inf.br/eks/downloads/eks_mips.pdf.

[SAPJ10] Guilherme C. R. Sales, Márcio R. D. Araújo, Flávio L. C. Pádua, and Fábio L. Corrêa Júnior. MIPS X-Ray: A Plug-in to MARS Simulator for Datapath Visualization. 2010.

[SCB08] Hessam Sarjoughian, Yu Chen, and Kevin Burger. A Component-based Visual Simulator for MIPS32 Processors. *38th Frontiers in Education Conference*, pages F3B–9 – F3B–14, October 2008.

[Sco12] Mike Scott. WinMIPS64, April 2012. Accessed on June 6, 2013. http://indigo.ie/~mscott.

[Shi12] Richard Shim. Tablets Impact the Notebook Market: Enter the Ultrabook. *Information Display*, 28(2 and 3):12 – 14, February/March 2012. Accessed on June 6, 2013. http://sid.calcey.net/Portals/InformationDisplay/IssuePDF/03_2012.pdf#page=14.

[Tea] The EduMIPS64 Team. EduMIPS64. Accessed on June 6, 2013. http://www.edumips.org.

[UG13] Evoreto UG. M32 Assembly - Android Apps on Google Play, March 2013. Accessed on May 29, 2013. http://play.google.com/store/apps/details?id=de.fhaachen.m32.

[Uni12] Missouri State University. MARS (MIPS Assembler and Runtime Simulator), February 2012. Accessed on June 6, 2013. http://courses.missouristate.edu/kenvollmar/mars.

[VS06] Dr. Kenneth Vollmar and Dr. Pete Sanderson. MARS: An Education-Oriented MIPS Assembly Language Simulator. March 2006. Accessed on June 6, 2013. http://www.cs.missouristate.edu/~vollmar/MARS/fp288-vollmar.pdf.

[web08] O Projeto WebSimple, 2008. Accessed on June 6, 2013. http://201.17.130.17/matheus/simple.