



NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

**SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**

BSc THESIS

**Educational Simulation Framework for Performance
Modeling of RISC Microprocessors**

Konstantinos V. Chasialis

Supervisor: Dimitris Gizopoulos, Professor

ATHENS

September 2021



ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

**Εκπαιδευτικό Περιβάλλον Προσομοίωσης για τη
Μοντελοποίηση Απόδοσης Μικροεπεξεργαστών RISC**

Κωνσταντίνος Β. Χασιαλής

Επιβλέπων: Δημήτρης Γκιζόπουλος, Καθηγητής

ΑΘΗΝΑ

Σεπτέμβριος 2021

BSc THESIS

Educational Simulation Framework for Performance Modeling of RISC Microprocessors

Konstantinos V. Chasialis

S.N.: 1115201600195

SUPERVISOR: **Dimitris Gizopoulos**, Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Εκπαιδευτικό Περιβάλλον Προσομοίωσης για τη Μοντελοποίηση Απόδοσης
Μικροεπεξεργαστών RISC

Κωνσταντίνος Β. Χασιαλής

A.M.: 1115201600195

ΕΠΙΒΛΕΠΩΝ: Δημήτρης Γκιζόπουλος, Καθηγητής

ABSTRACT

One of the most fundamental topics in the field of Computer Science is how hardware works and how it interacts with software in contemporary computing systems. Unfortunately, this topic is hard to understand because students are not able to visualize/simulate and experiment with what they are taught. For that reason, many programs have been developed and their goal is to visualize what happens in the hardware when a program is being executed. Another equally significant topic is about assembly languages. MIPS as RISC ISA is usually the MIPS assembly language that is taught in universities. The main reason that MIPS is used is that assembly languages are very complex, counter-intuitive and difficult to understand while MIPS is simple enough to understand yet complicated enough to teach all the basic points. Additionally, MIPS processors continue to be used in workstations, embedded systems (e.g. routers, set-top boxes, cable modems etc.) and even supercomputers. In this thesis we turn our attention to a widely used project, QtMips. QtMips is a visual educational simulator that uses MIPS RISC ISA and simulates many hardware components during a program execution. We will showcase how it works, which hardware components it simulates and we will add useful extensions to this program by providing simulation for more complex hardware components like branch predictor. Finally, in collaboration with a team from Czech Technical University in Prague, we will develop a more advanced version of this program that uses a modern architecture/instruction set RISC-V.

SUBJECT AREA: Computer Architecture

KEYWORDS: Assembly, RISC, MIPS, RISC-V, Instruction Set, CPU, DRAM, SRAM, CPU, ALU, FPU, Pipeline, Branch Predictors, Cache, Memory, Microprocessors, Performance Modeling, Simulation

ΠΕΡΙΛΗΨΗ

Μια από τις πιο βασικές γνώσεις στον τομέα της Πληροφορικής είναι πως λειτουργεί το υλικό καθώς και η αλληλεπίδραση του υλικού με το λογισμικό σε σύγχρονα υπολογιστικά συστήματα. Δυστυχώς, το κομμάτι αυτό είναι αρκετά δύσκολο στην κατανόηση επειδή οι μαθητές δεν μπορούν να οπτικοποιήσουν/προσομοιώσουν αυτά που διδάσκονται. Γι αυτόν τον λόγο έχουν αναπτυχθεί πολλά προγράμματα που στόχο έχουν να οπτικοποιήσουν αυτήν την αλληλεπίδραση καθώς και πολλές άλλες λειτουργίες του υλικού. Μία άλλη εξίσου σημαντική γνώση αφορά της συμβολικές γλώσσες. Η συμβολική γλώσσα MIPS είναι η πιο συχνά χρησιμοποιούμενη γλώσσα σε μαθήματα πανεπιστημίου. Ο κύριος λόγος που χρησιμοποιείται ευρέως η γλώσσα MIPS είναι γιατί γενικά οι συμβολικές γλώσσες είναι ιδιαίτερα περίπλοκες και δύσκολες στην κατανόηση ενώ η MIPS είναι απλή στην κατανόηση αλλά και όσο περίπλοκη χρειάζεται για να διδαχθούν όλα τα βασικά κομμάτια για τις συμβολικές γλώσσες. Επιπροσθέτως, MIPS μικροεπεξεργαστές χρησιμοποιούνται ακόμη και σήμερα σε σταθμούς εργασίας, ενσωματωμένα συστήματα (π.χ. δρομολογητές, αποκωδικοποιητές, καλωδιακά μόντεμ κ.λπ.) και ακόμη και υπερυπολογιστές. Σε αυτήν την πτυχιακή στρέφουμε το ενδιαφέρον μας σε ένα ευρέως χρησιμοποιημένο project, τον QtMips. Ο QtMips είναι ένας προσομοιωτής για εκπαιδευτικούς σκοπούς που χρησιμοποιεί MIPS RISC ISA και προσομοιώνει πολλές βασικές λειτουργίες του υλικού όταν εκτελείται ένα πρόγραμμα. Θα περιγράψουμε τον τρόπο λειτουργίας του, ποιά κομμάτια του υλικού προσομοιώνει/οπτικοποιεί και θα επεκτείνουμε την λειτουργικότητα του προγράμματος προσθέτοντας προσομοίωση για πιο περίπλοκα κομμάτια του υλικού όπως ο branch predictor. Τέλος, σε συνεργασία με την ομάδα από το Τεχνικό Πανεπιστήμιο της Πράγας, θα συμβάλουμε στην ανάπτυξη μιας πιο εξελιγμένης έκδοσης αυτού του προγράμματος, που χρησιμοποιεί την πιο σύγχρονη αρχιτεκτονική/σύνολο εντολών RISC-V.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αρχιτεκτονική Υπολογιστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Συμβολική Γλώσσα, RISC, MIPS, RISC-V, Σύνολο Εντολών, CPU, DRAM, SRAM, ALU, FPU, Διοχέτευση, Πρόβλεψη Διακλάδωσης, Κρυφή μνήμη, Μικροεπεξεργαστές, Απόδοση, Μοντελοποίηση, Προσομοίωση

CONTENTS

1. INTRODUCTION	12
2. MIPS ARCHITECTURE DESIGN & CONCEPTS	13
2.1 MIPS model	13
2.2 Registers	13
2.3 Instruction formats	14
2.4 Instructions	15
2.4.1 Arithmetic Instructions	15
2.4.2 Logical Instructions	17
2.4.3 Branching Instructions	18
2.4.4 Load & Store Instruction	19
2.5 MIPS Pipeline & Hazards	19
2.5.1 Pipeline	19
2.5.2 Hazards	20
2.5.2.1 Structural Hazards	20
2.5.2.2 Data Hazards	21
2.5.2.3 Control Hazards	21
2.6 MIPS Memory Hierarchy	22
2.6.1 Cache	23
3. VISUAL EDUCATIONAL SIMULATORS	28
4. QTMIPS	34
4.1 Registers	34
4.2 Program Memory	34
4.3 Data Memory	35
4.4 Program Execution	35
4.5 Code Editor & Compiler	36
4.6 OS Emulation	36
4.7 L1 Program/Data Cache	38
4.8 Coreview	38
5. EXTENSIONS TO QTMIPS	40

5.1	Branch Predictor	40
5.1.1	Static Branch Predictors	41
5.1.2	Dynamic Branch Predictors	41
5.2	Cache	43
5.3	Cycle Statistics	44
5.4	Representation of Register Values	44
5.5	Data and Control Hazard Unit	47
6.	EXPERIMENTS	48
6.1	Branch Predictor	48
6.2	L2 Cache	50
7.	QtMips MANUAL	53
8.	EXTENSIONS ON QtMips AND SWITCH TO QtRVSim	57
9.	FUTURE WORK	58
	ABBREVIATIONS - ACRONYMS	59
	REFERENCES	60

LIST OF FIGURES

2.1	MIPS Registers	14
2.2	MIPS Processor (Pipelined)	22
2.3	Memory Hierarchy	24
2.4	Cache	27
3.1	QtSPIM	29
3.2	WinMIPS64	30
3.3	Dinero IV Simulator	31
3.4	MipsIt Windows	31
3.5	MipsIt I//D-Cache	32
3.6	MipsIt Pipeline	32
3.7	MARS	33
4.1	QtMips Registers Dock	34
4.2	QtMips Program Dock	35
4.3	QtMips Memory Dock	35
4.4	QtMips Memory Dock	35
4.5	QtMips Code Editor	36
4.6	QtMips OS Emulation	37
4.7	QtMips L1 Program Cache Dock	38
4.8	QtMips Coreview (Pipelined)	39
4.9	QtMips Coreview (Single-Cycle)	39
5.1	1-Bit Branch Predictor	42
5.2	2-Bit Branch Predictor	42
5.3	QtMips Coreview (Branch Predictor)	43
5.4	QtMips Branch History Table	43
5.5	QtMips Branch Target Buffer	44
5.6	QtMips L2 Cache Dock	45
5.7	QtMips Cycle Statistics Dock	46
5.8	QtMips Registers Notation	46
5.9	QtMips Control & Data Hazard Units	47
6.1	QtMips Program Execution (Branch)	49
6.2	QtMips Program Execution (Cache)	52
7.1	QtMips Program Loader	53
7.2	QtMips Presets	53
7.3	QtMips Core Configuration	54
7.4	QtMips Memory Configuration	55
7.5	Cache Configurations	56

LIST OF TABLES

2.1	MIPS Instruction Formats	15
2.2	MIPS Arithmetic Instructions	16
2.3	MIPS Logical Instructions	17
2.4	MIPS Branching & Jump Instructions	18
2.5	MIPS Load & Store Instructions	19
3.1	Throughput per simulator	33
6.1	Cycles (Branch)	48
6.2	Speedups (Branch)	49
6.3	Cycles (Cache)	51
6.4	Speedups per memory configuration	51
6.5	Cycles (Cache)	52

PREFACE

There are two reasons behind me choosing to extend a tool used for teaching a course as my bachelor thesis.

- I wanted to leave something behind to a university that taught me how to program, taught me how to study and has grown me as a professional and as an individual, and, since this program will be used for educational purposes in Computer Architecture courses, it was exactly what I was looking for.
- To provide help to a beloved professor, Dimitris Gizopoulos, to take his courses to the next level (Don't get me wrong here, they are already very interesting and fun).

1. INTRODUCTION

Computer Architecture is a compulsory course in the curriculum for most Computer Science and Computer Engineering studies. The reason behind this is that every programmer should, at least, have a rough idea of how a processor works internally. One could argue that this is not important for a *Software Engineer* because their main job is to program software and not hardware. This is the compilers job to do, right? To an extent this argument holds, but a Software Engineer that completely ignores all hardware-related components will never be able to program efficiently since they do not take into account important aspects like instruction pipelining, cache behavior, branch mispredictions, register usage and the list goes on and on. In order to be an expert programmer one should at least know what all these mean.

Learning Computer Architecture is not an easy task, teaching it however, is a very demanding job, especially in universities which want to offer a high quality of education. Professors will need all the help they can get. That is where visual simulators of how a processor works come into play. They offer a better understanding of all the important aspects of Computer Architecture by visualizing them and by giving the chance to students to try different configurations on a "processor model" which in turns allows them to dive more deeply onto how a processor really works.

There are many visual simulators available, and each one offers a unique set of features.

In this thesis we turn our attention to QtMips, a recently developed Graphical CPU Simulator with Cache Visualization, which was developed by Ing. Pavel Pisa, Ph.D. and Ing. Karel Koci of the Czech Technical University in Prague, and its source code as well as instructions to build it, can be found here.

On chapter 2 we will first define and analyze how a MIPS processor is designed and how it works, we will present its instruction set and basic concepts of this architecture such as pipelining, hazards and so on. On chapter 3 we will analyze what visual educational simulators offer and how to they aid the teaching of Computer Architecture. On chapter 4 we will present QtMips and what features it offers. On chapter 5 we will present the features that we decided to add to QtMips in order to extend its functionality. More specifically, our additions include:

- 1-bit / 2-bit branch predictors (using branch target buffers).
- L2 (Unified) Cache.
- Control hazard unit.
- Detailed statistics for the program that is executed.
- Other tweaks.

On chapter 6 we will showcase program-executions / experiments ran on the new version of QtMips. On chapter 7 we will present a short manual for the new version of QtMips and on chapter 8 we will discuss what features we added to a modern version of QtMips, QtRVSim. Finally, on chapter 9 we will discuss our future work.

2. MIPS ARCHITECTURE DESIGN & CONCEPTS

MIPS (Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Computer Systems. It contains a set of instructions that are relatively simple to use, implement and fast to execute, in contrast to x86 architecture (CISC), which has more complex and multi-cycle instructions.

2.1 MIPS model

The MIPS32 architecture is based on a fixed-length, regularly encoded instruction set and uses a load/store data model. The architecture is streamlined to support optimized execution of high-level languages. Arithmetic and logic operations use a three-operand format, allowing compilers to optimize complex expressions formulation. Availability of 32 general-purpose registers enables compilers to further optimize code generation for performance by keeping frequently accessed data in registers.

MIPS architecture all contains 4 extensions that are referred to as co-processors (because they were initially implemented as separate chips out of the main CPU).

- CP0: This co-processor is used for virtual memory system, exception handling and various CPU states
- CP1: FPU - It is used to handle floating point arithmetic.
- CP2: Free for each platform to utilize as it needs.
- CP3: Reserved for MIPS ISA extension.

2.2 Registers

MIPS contains 32 general-purpose registers, and each register has a capacity of 32 bits. The register \$0 (or \$zero) is a special register. It has the value of 0 and its contents cannot be altered, writes to it do not have any effect. Register \$31 is used as a link for jump-and-link instruction (jal). It stores the address of the next instruction that should be executed when the program returns from the block of code that it jumped to. The return address stored in this register is used by a jr instruction that should be the last of the subroutine of code. Besides these 32 registers, there are also the registers HI and LO which are accessible by special commands and they are only used to store the results of operations that would not fit in a single register. These are the results of different multiply and divide instructions which all have a two-word result. There is also the program counter which has 32 bits and contains the address of the instruction to be fetched. The two low-order bits always contain zero since MIPS I instructions are 32 bits long and are aligned to their natural word boundaries.

MIPS convention is to name the general-purpose registers by using two-character names following a dollar sign to represent a register. MIPS software separates 18 of the registers into two groups

32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register \$zero always equals 0, and register \$at is reserved by the assembler to handle large constants.
--------------	--	---

Figure 2.1: MIPS Registers

- \$t0–\$t9: ten temporary registers that are not preserved by the callee (called procedure) on a procedure call.
- \$s0–\$s7: eight saved registers that must be preserved on a procedure call (if used, the callee saves and restores them).

2.3 Instruction formats

Instruction format represents how an instruction is encoded. All MIPS instructions occupy 32 bits, the size of a word. Not all instructions have the same format, consider these 3 for example:

add reg1, reg2, reg3 (R-type)

lw reg1, off(reg2) (I-type)

j addr (J-type)

But, since all instructions are encoded using 32 bits, how is MIPS able to differentiate in which format to decode the current instruction? For that reason, MIPS Instructions are divided into three types: R (for register), I (for immediate) and J (for jump). Instruction bits are split in segments, which are called fields. Each instruction is split in a different way, depending on its type.

• R-Type instructions

- *op*: Basic operation of the instruction, traditionally called the opcode.
- *rs*: The first register source operand.
- *rt*: The second register source operand.
- *rd*: The register destination operand. It gets the result of the operation.
- *shamt*: Shift amount (used only for shift operations).
- *funct*: Function. This field, often called the function code, selects the specific variant of the operation in the *op* field.

• I-Type instructions

- *op*: Same as in R-Type.
- *rs*: Same as in R-Type.
- *rt*: If the instruction writes to register this is the destination register, otherwise its the same as in R-Type instructions.
- *address*: A 16-bit (signed) number which, if the instruction uses memory, denotes the memory offset to be accessed, in words (relative to *rs*), if the instruction is a branch, denotes the offset, in words, (relative to the program counter) that the program should jump to if the branch is taken, and if the instruction is an arithmetic it is the number used in the operation.

- **J-Type instruction**

- *op*: Same as in R-Type and I-Type.
 - *target*: This field is 26-bit and denotes to which address we can't the program to jump to. The jump target address is formed by concatenating the high order four bits of the PC (the address of the instruction following the jump), the 26 bits of the target field, and two 0 bits.
- The first three fields of the R-type and I-type formats are the same size and have the same names.
 - The length of the fourth field in I-type is equal to the sum of the lengths of the last three fields of R-type.
 - The formats are distinguished by the values in the first field: each format is assigned a distinct set of values in the first field (*op*) so that the hardware knows whether to treat the last half of the instruction as three fields (R-type) or as a single field (I-type).

On table 2.1 we can see how each instruction format is decoded.

Table 2.1: MIPS Instruction Formats

MIPS Instruction Formats						
Type	Format (bits)					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	function (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

2.4 Instructions

MIPS architecture provides simple instructions that are able to execute various tasks like arithmetic operations (integers and floating point), logical operations, conditional and unconditional branching, data transfer from memory to the CPU and some special instructions. Some of these instructions are presented below.

2.4.1 Arithmetic Instructions

Most of these belong to the R-Type category because they operate on 3 registers. Two of the registers included in the instruction are used for read (*rs*, *rt*) and one of the register is used for storing the result (*rd*). The *opcode* field on R-Type instructions is always zero. So how does MIPS know which operation do apply on the registers? The answer is by checking the *func* field, which, as its name implies, specifies the function for the Arithmetic and Logic Unit (ALU) to use.

The arithmetic instructions that do not belong to the R-Type category belong to the I-Type category. The reason that not all of these instructions belong to the R-Type category is that not all arithmetic instructions require 3 registers (i.e. 3 variables) to be used. Imagine, for example, that we want to add a *constant* value to a register. If MIPS did not support `addi` (add operation in I-Type) category, we would first have to move the constant value to another register and then use the `add` operation which requires 3 registers. By including arithmetic operations in the I-Type category MIPS allows us to do fast arithmetic operations using constant values. This, however, comes with a drawback. Recall that the immediate field on I-Type instructions is only 16-bit long but registers are 32-bit long. This limits us to values between $[0..2^{16}-1](0..65.535)$ if the operation is an unsigned operation and between $[-2^{15}..2^{15}-1](-32.768..32.767)$ (The MIPS processor uses two's complement for signed immediate operands). The 16-bit offset limitation in arithmetic instructions is solved by using an `lui` instruction with combination of `addi` or `ori` for the constructions of large constants.

On table 2.2 we can see some of the most used arithmetic instructions. The full instruction set of MIPS32 can be found in [1].

Table 2.2: MIPS Arithmetic Instructions

Name	Mnemonic	Type	Operation
Add	<code>add</code>	R	$rd = rs + rt$ (rs and rt are interpreted as signed)
Add Unsigned	<code>addu</code>	R	$rd = rs + rt$ (rs and rt are interpreted as unsigned)
Add Immediate	<code>addi</code>	I	$rt = rs + \text{immediate}$ (rs is interpreted as signed)
Add Immediate Unsigned	<code>addiu</code>	I	$rt = rs + \text{immediate}$ (rs is interpreted as unsigned)
Set Less Than	<code>slt</code>	R	$rd = rs < rt ? 1 : 0$ (rs and rt are interpreted as signed)
Set Less Than Unsigned	<code>sltu</code>	R	$rd = rs < rt ? 1 : 0$ (rs and rt are interpreted as unsigned)
Set Less Than Immediate	<code>slti</code>	I	$rt = rs < \text{immediate} ? 1 : 0$ (rs is interpreted as signed)
Set Less Than Immediate Unsigned	<code>sltiu</code>	I	$rt = rs < \text{immediate} ? 1 : 0$ (rs is interpreted as unsigned)
Subtract	<code>sub</code>	R	$rd = rs - rt$ (rs and rt are interpreted as signed)
Subtract Unsigned	<code>subu</code>	R	$rd = rs - rt$ (rs and rt are interpreted as unsigned)
Divide	<code>div</code>	R	$Lo = rs \text{ div } rt, Hi = rs \% rt$ (rs and rt are interpreted as signed)
Divide Unsigned	<code>divu</code>	R	$Lo = rs \text{ div } rt, Hi = rs \% rt$ (rs and rt are interpreted as unsigned)
Multiply	<code>mult</code>	R	$\{Hi, Lo\} = rs * rt$ (rs and rt are interpreted as signed)
Multiply Unsigned	<code>multu</code>	R	$\{Hi, Lo\} = rs * rt$ (rs and rt are interpreted as unsigned)
Shift Right Arithmetic	<code>sra</code>	R	$rd = rt \gg \text{shamt}$ (The difference with <code>srl</code> is that <code>sra</code> does an arithmetic shift, it shifts the sign bit in instead of adding zeros)

An important thing to note here is that MIPS supports both integer and floating point arithmetic. MIPS has a floating point coprocessor (numbered 1) that operates on single precision (32-bit) and double precision (64-bit) floating point numbers. This coprocessor has its own registers, which are numbered \$f0-\$f31. Because these registers are only 32-bits wide, two of them are required to hold doubles. To simplify matters, floating point operations only use even-numbered registers—including instructions that operate on single floats. Values are moved in or out of these registers a word (32-bits) at a time by `lwc1`, `swc1`, `mtc1`, and `mfc1` instructions or by the `l.s`, `l.d`, `s.s`, and `s.d` pseudoinstructions. In this thesis we will omit instructions that operate on floating point numbers because the baseline version of QtMips on which we built our extensions does not support floating point arithmetic. The addition of floating point operations and instructions is an extension that we plan on adding on the future version of the simulator. More about extensions and future work in the chapters below.

2.4.2 Logical Instructions

MIPS also offers a variety of logical instructions. These operate bitwise on two sources and write the result to a destination register. The first source is always a register and the second source is either an immediate or another register. On table 2.3 we present the ones which are most frequently used.

Table 2.3: MIPS Logical Instructions

MIPS Logical Instructions			
Name	Mnemonic	Type	Operation
And	<code>and</code>	R	$rd = rs \& rt$
And Immediate	<code>andi</code>	I	$rt = rs \& \text{immediate}$ (interpreted as unsigned)
Nor	<code>nor</code>	R	$rd = \neg(rs \mid rt)$
Or	<code>or</code>	R	$rd = rs \mid rt$
Or Immediate	<code>ori</code>	I	$rt = rs \mid \text{immediate}$ (interpreted as unsigned)
Shift Left Logical	<code>sll</code>	R	$rd = rt \ll \text{shamt}$
Shift Right Logical	<code>srl</code>	R	$rd = rt \gg \text{shamt}$
Xor	<code>xor</code>	R	$rd = rs \oplus rt$
Xor Immediate	<code>xori</code>	I	$rt = rs \oplus \text{immediate}$

2.4.3 Branching Instructions

MIPS offers instructions that help a program make decisions based on some conditions. Making a decision in a program means that the program can determine which code should be executed and which code should not be executed depending on the input it has in the current moment. These instructions are called branching instructions.

Some of the most frequently used ones are presented on table 2.4.

Table 2.4: MIPS Branching & Jump Instructions

MIPS Branching & Jump Instructions			
Branch On Equal	beq	I	if $rs == rt$ then $PC = PC + 4 + \text{Branch Address}$
Branch On Not Equal	bne	I	if $rs != rt$ then $PC = PC + 4 + \text{Branch Address}$
Branch Less Equal Zero	blez	P	if $rs \leq 0$ then $PC = PC + 4 + \text{Branch Address}$
Branch Great Than Zero	bgtz	P	if $rs > 0$ then $PC = PC + 4 + \text{Branch Address}$
Branch Less Than	blt	P	if $rs < rt$ then $PC = PC + 4 + \text{Branch Address}$
Branch Greater Than	bgt	P	if $rs > rt$ then $PC = PC + 4 + \text{Branch Address}$
Branch Less Than or Equal	ble	P	if $rs \leq rt$ then $PC = PC + 4 + \text{Branch Address}$
Branch Greater Than or Equal	bge	P	if $rs \geq rt$ then $PC = PC + 4 + \text{Branch Address}$
Jump	j	J	$PC = \text{Jump Address} \{PC + 4, \text{address field}, 00\}$
Jump And Link	jal	J	$\$ra = PC + 8; PC = \text{Jump Address}$
Jump Register	jr	J	$PC = rs$
Jump And Link Register	jalr	J	$\$ra = PC + 8; PC = rs$

P means that this instruction is a pseudo-instruction.
For example, blez is translated to:

```
slt $t0, $t1, $t2
bne $t0, $0, L1
```

And ble is translated to:

```
slt $t0, $t1, $t2
beq $t2, $zero, L1
```

2.4.4 Load & Store Instruction

Load instructions move data from memory into a register. The address for the load is the sum of a register specified in the instruction and a constant value that is coded into the instruction. Load instructions belong in the I-Type category.

Store instructions store data from a register to memory. Like load instructions, the memory address in which the value of the register will be written to is the sum of the immediate field in instruction with the register *rs*. These instructions also belong in the I-Type category. MIPS offers the ability to load/store bytes (signed / unsigned), half-words (signed / unsigned), words (signed / unsigned).

Some of the most frequently used ones are presented on table ?? (pseudo-instructions (P) included).

Again, the full instruction set of MIPS32 can be found in [1].

Table 2.5: MIPS Load & Store Instructions

MIPS Load & Store Instructions			
Load Byte	lb	I	rt = MEM[rs + immediate]{b0..7} (interpreted as signed)
Load Byte Unsigned	lbu	I	rt = MEM[rs + immediate]{b0..7} (interpreted as unsigned)
Load Halfword	lh	I	rt = MEM[rs + immediate]{b0..15} (interpreted as signed)
Load Halfword Unsigned	lhu	I	rt = MEM[rs + immediate]{b0..15} (interpreted as h unsigned)
Load Word	lw	I	rt = MEM[rs + immediate]{b0..31}
Store Byte	sb	I	MEM[rs + immediate] = rt{b0..7}
Store Halfword	sbu	I	MEM[rs + immediate] = rt{b0..15}
Store Word	sw	I	MEM[rs + immediate] = rt{b0..31}

2.5 MIPS Pipeline & Hazards

2.5.1 Pipeline

MIPS was designed to support pipelining (table 2.2). Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

It's important to understand that pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction, but instruction throughput is the important metric because real programs execute billions of instructions. The way it achieves that is by partially overlapping the execution of instructions.

By exploiting pipeline, one can achieve high speedup. If we assume perfect conditions,

$$Time\ between\ instructions_{pipelined} = \frac{Time\ between\ instructions_{not-pipelined}}{Pipeline\ Stages}$$

If one wants to build an architecture that implements pipelining, they have to follow three basic steps:

- Identify pipeline stages: This means to identify the common parts of all instructions. These parts are called stages.
- Isolate stages from each other.
- Resolve pipeline hazards that will occur.

The term pipeline hazards refer to situations on which the initiation of a new instruction at every clock cycle is not possible because it may lead to incorrect execution of the program.

MIPS instructions (and most RISC processor instructions) typically take five steps:

1. Fetch instruction from memory (Instruction Fetch - IF): The instruction that the program counter points to is loaded from memory.
2. Read registers while decoding the instruction (Instruction Decode - ID): The regular format of MIPS instructions allows reading and decoding to occur simultaneously. Register identifiers *rs* and *rt* are used to getting values from given registers and immediate instruction field is sign extended to 32-bits.
3. Execute the operation or calculate an address (Execution - EX): Execution stage contains ALU. It operates on top of two 32bit values and outputs another 32bit value as a result. For some operations it also updates HI and LO registers. Values passed to ALU are values loaded from registers from instruction decode stage.
4. Access an operand in data memory (Memory - MEM): Memory stage is dedicated for memory access. As an address is used ALU output from execute stage. For write instructions, the value to be written is value from register from *rt* passed through execute stage from decode stage.
5. Write the result into a register (Write Back - WB): Either the output of ALU or from memory is written to the destination register.

We have presented the pipeline stages and isolated one from the other. Its important to mention that not all stages have an equal propagation delay and that is why the longest stage (the one that needs more time to finish) determines the clock cycle of the pipeline. A single instruction needs 5 of these clock cycles (# of stages) but in each cycle an instruction is always completed, allowing for bigger throughput and faster program execution overall due to the partial overlapping of instructions execution.

By introducing pipelining, we also introduce some hazards that pipelining inevitably brings along.

2.5.2 Hazards

2.5.2.1 Structural Hazards

Structural hazards occur when two instructions try to access the memory concurrently. More specifically, when an instruction is on its IF stage and the CPU tries to load the instruction from memory and another instruction is on its MEM stage and tries to load/store

data into memory. If the hardware cannot support the combination of these instructions, one has to wait on its current stage until the other instruction is finished with its stage. This, however, can be, and is actually solved by providing two separate memories, one for data and one for instructions.

2.5.2.2 Data Hazards

Data hazards occur when one instruction needs to access data from a register but a previous instruction was not finished writing the data to this register. Consider this simple example:

```
addi $t0, $zero, 15
sub $t1, $t1, $t0
```

In order for the program to be executed as programmed to, the sub instruction requires the \$t0 register to have the value 15. But, since the value is written to the register on the last pipeline stage (WB) and sub needs the result on the execution stage to perform the operation (EX), sub would have to wait 2 extra cycles for \$t0 to have the correct result. These kind of instruction sequences exist in a large proportion in every program and if left un-handled it could cause severe stalls on the pipeline. MIPS addresses this issue by introducing something called **forwarding**. Forwarding is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Forwarding is achieved by adding extra hardware to retrieve the missing item early from the internal resources.

However, there is a specific data hazard which cannot be solved, even by using forwarding. Let's alter the above code slightly to observe why this occurs.

```
lw $t0, 8($t4)
sub $t1, $t1, $t0
```

The pipeline will reach to a point where lw is on MEM stage and sub is on EX stage and will attempt to read \$t0. However, since lw will have the result from memory that is to be written to \$t0 on the MEM stage, not in EX, its impossible to forward the correct value to sub because MEM stage has to be executed completely before forwarding. Therefore, sub will still have to wait 1 extra cycle to be executed correctly. These hazards are called load-use hazards and we rely on the compilers to eliminate them if possible.

2.5.2.3 Control Hazards

Control hazards occur when the CPU fetches a control flow (branch or jump) instruction. Notice that we must begin fetching the instruction following the branch on the very next clock cycle. Obviously, the next instruction might be the next in the source code (PC + 4) or it might be somewhere else in the code, depending on the result of the branch. CPU simply does not know which instruction to fetch next and therefore needs to stall for the ALU to resolve the branch and decide where the next instruction to be executed is located. Branch is typically resolved (the branch condition is calculated) at the EX stage, which means the instruction after the branch has to stall for 2 cycles and the next instruction 1 cycle. Sometimes extra hardware is added to resolve branches on ID stage so the stall is only 1 cycle. However, branches are very common in a program and stalling for each

branch is not an efficient solution.

Modern CPUs (and MIPS itself of course) use branch prediction, which basically means that when a branch is fetched, the CPU predicts that it is either taken or not taken. If the prediction is correct, no stalls will get introduced. If the prediction is wrong, we stall the same cycles as without predicting. Its important to note that in modern CPUs branch predictors are very very accurate (98-99%).

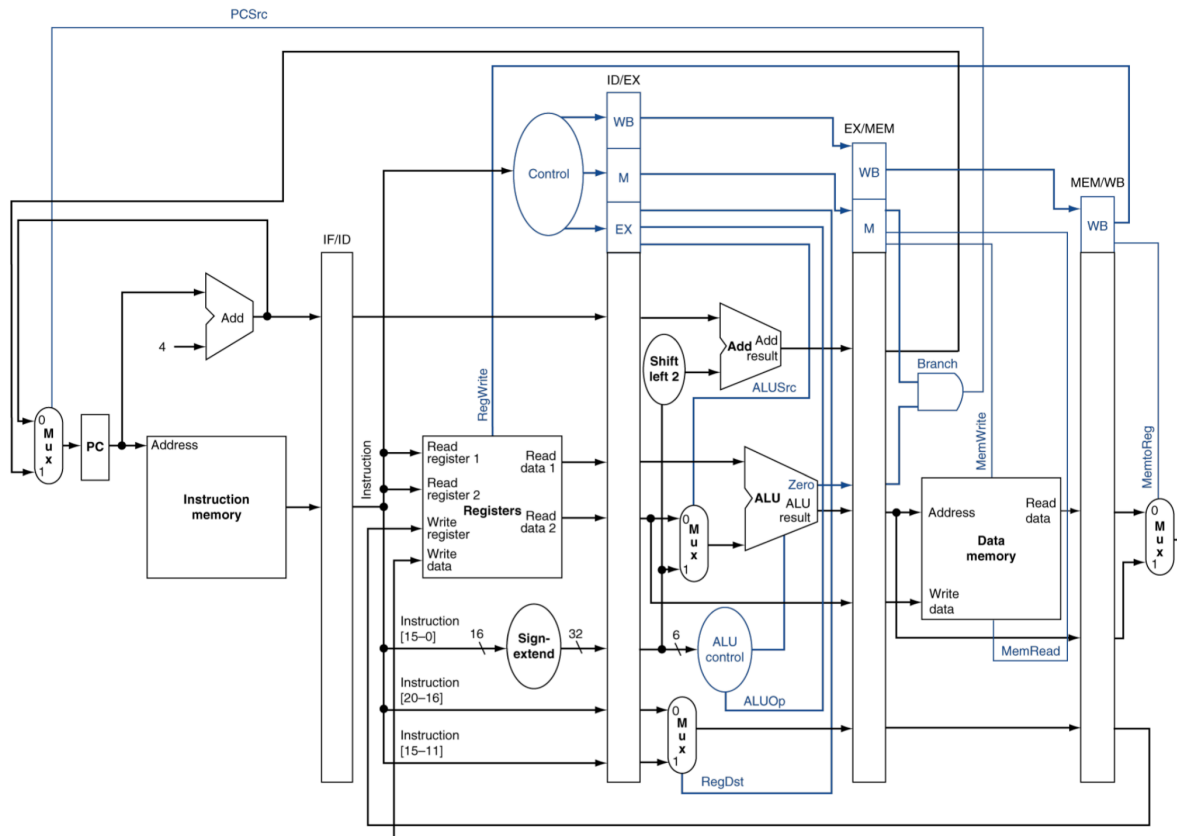


Figure 2.2: MIPS Processor (Pipelined)

2.6 MIPS Memory Hierarchy

As mentioned previously, MIPS employs the load / store architecture which means that access to memory is provided by 2 basic operations: load data from memory and store data to memory. With this approach, its less complex for programmers who write the code but also enables aggressive compiler optimizations because there are no instructions that might read from or write to memory at the same time. Also, structural hazards are completely eliminated when two separate memories (for instructions and data) are available. This approach, however, does not tackle a very important issue when it comes to performance. The issue is that DRAM access cost is significantly larger than CPU access cost. There is a principle that underlies the way the programs operate. This principle is called principle of locality [[1], [2], [6]].

Principle of locality: All programs access a relatively small portion of their address space at any instant of time.

There are two different types of locality:

- *Temporal locality* (locality in time): When an item is referenced, it will tend to be

referenced again soon.

- *Spatial locality* (locality in space): When an item is referenced, items whose addresses are close by will tend to be referenced soon.

We take advantage of the principle of locality by implementing the memory of a computer as a memory hierarchy. A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus are smaller. Today, there are three primary technologies used in building memory hierarchies. Main memory is implemented from DRAM (dynamic random access memory), while levels closer to the processor (caches) use SRAM. DRAM is less costly per bit than SRAM, although it is substantially slower.

Today, a typical computer comes with a 6-level memory hierarchy (figure 2.3), more specifically:

1. *Registers*: This is the fastest memory one can get. Register access is virtually free but there are only a handful of them available in processors.
2. *L1 Cache*: This is the memory level closer to the processor. It offers very fast access (close to register access time) but its size is typically small, around 8 - 64 KB.
3. *L2 Cache*: This is the memory level above L1 Cache and its somewhat slower than L1 Cache but its also relatively fast. It's size varies between 256KB to 8MB.
4. *L3 Cache*: This is the memory level above L2 Cache. It's size varies between 10 - 64 MB.
5. *DRAM*: This is the main memory of the computer as we all know. DRAM is 10 to 100 times slower than a cache. In a typical computer its size is between 4 - 128 GB.
6. *Magnetic Disk*: This is (sometimes) the last level of memory in a computer. Accessing data from the disk is 100000 (!) slower than accessing data from DRAM. However, magnetic disks offer a capacity of 256GB - a lot of TB.

2.6.1 Cache

Cache memory (figure 2.4) refers to the memory areas that are close to the processor and offer very fast data access. Their drawback is that they are relatively small because of their cost. We mentioned before that a memory hierarchy can consist of multiple levels. However, data is copied between only two adjacent levels at a time (L1 to L2 for example). Now let's define some frequently used terms when it comes to caches.

The minimum unit of information that can be either present or not present in the two-level hierarchy is called a *block* or a *line*.

If the data requested by the processor appears in some block in the upper level, this is called a *cache hit*. If the data is not found in the upper level, the request is called a *cache miss*. The lower level in the hierarchy is then accessed to retrieve the block containing the requested data.

The *hit rate*, or *hit ratio*, is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy. The *miss rate* ($1 -$

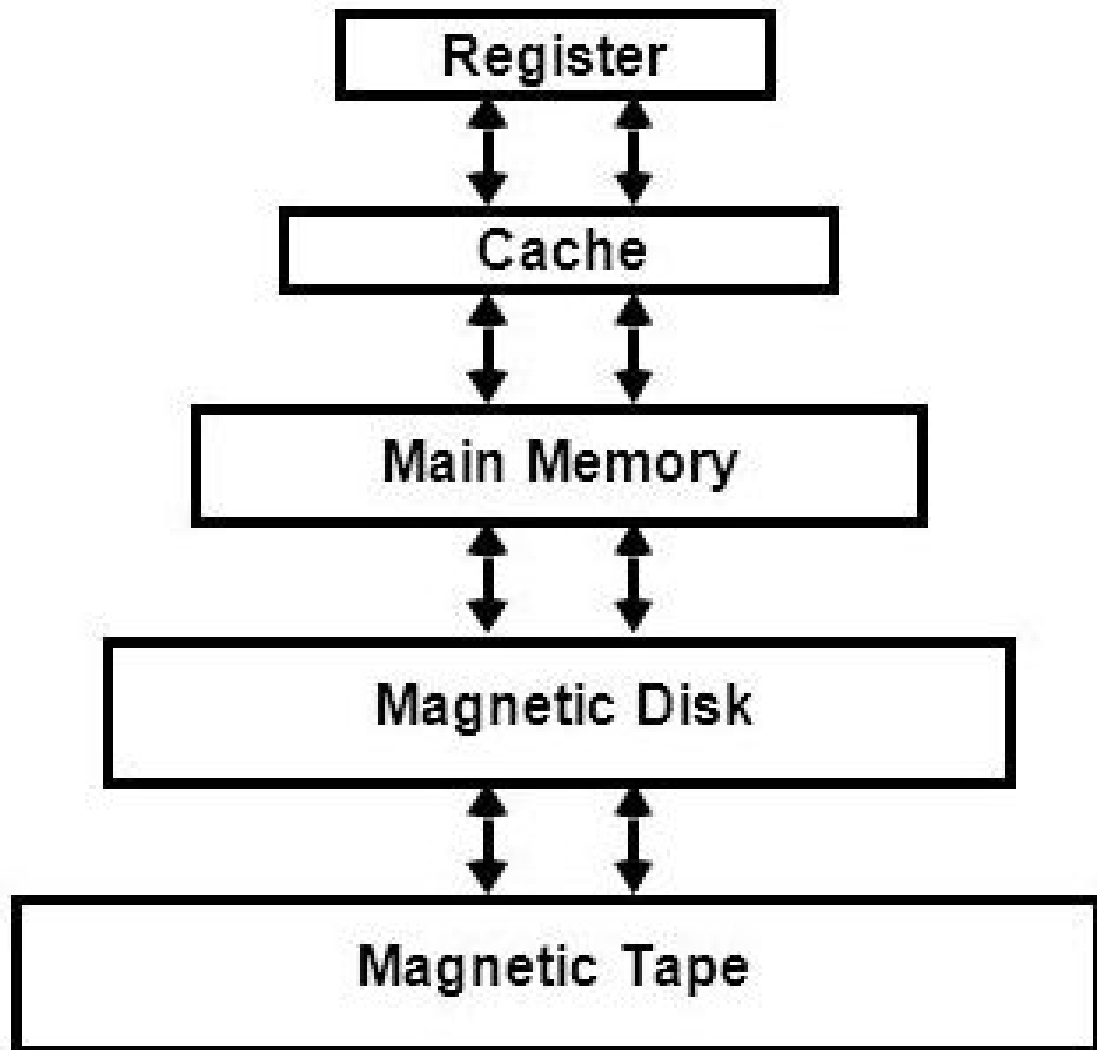


Figure 2.3: Memory Hierarchy

hit rate) is the fraction of memory accesses not found in the upper level.

Since performance is the major reason for having a memory hierarchy, the time to service hits and misses is important. *Hit time* is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss.

Miss penalty is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor. Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy, which is the major component of the miss penalty.

Now let's discuss how values are stored into caches.

The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the address of the word in memory. This cache structure is called *direct mapped*, since each memory location is mapped directly to exactly one location in the cache. In order to find a block, this formula is used:

Block address % *Number of blocks in the cache*.

It is obvious that many addresses will inevitably map to the same cache location. Therefore, we need a way to know if the address we want to access is the right one. For that reason, caches use *tags*. The tags contain the address information required to identify whether a word in the cache corresponds to the requested word. The tag needs only to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache.

Also, imagine that a program finished execution and left leftovers in the cache. The next program might try to access the same locations and even with the same tag, but the data is of course not the data it wants. For that reason cache entries also contain a *valid bit* which basically means if the entry is valid or not at the current time.

We mentioned only one block-placement method, direct mapped. There are however, more flexible ones.

In direct-mapped placement, a block can be placed at exactly one location. On the other side, on a *fully associative* cache, a block can be placed anywhere on the cache and lookup for each block is done using only the tag (and the valid bit). This lookup of course is done in parallel, which requires extra hardware and cost, and that's the reason only small caches are fully associative.

There is a middle ground solution for block placement. It is called *set associative*. In a set associative cache, there are a certain number of locations where each block can be placed. A *n-way set associative cache* consists of a number of sets, each of which consists of n blocks. Each block in the memory maps to a unique set in the cache given by the index field, and a block can be placed in any element of that set. The block lookup is easy. We first map to the set of the cache which the address might be located and then we parallel-search for all blocks inside the set (again using tags and valid bits).

When CPU requests for an address in cache, it might be for reading or it might be for writing. In both cases, a cache miss can occur and a block should be replaced. The problem is which block should be replaced.

When a miss occurs in a direct-mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache, we have a choice of where to place the requested block, and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set-associative cache, we must choose among the blocks in the selected set.

The most commonly used scheme is least recently used (LRU), which we used in the previous example. In an LRU scheme, the block replaced is the one that has been unused for the longest time. There are other replacement policies like R (Random Replacement), LFU (Least frequently used), FIFO, LIFO, TLRU (Time aware least recently used), MRU (Most recently used).

One important aspect about caches is how to handle writes to memory. One possible solution is to write the data only in cache, to avoid main memory access but this would cause inconsistency. The simplest way to keep the main memory and the cache consistent is always to write the data into both the memory and the cache. This scheme is called *write-through*. When a miss occurs in a write-through cache, the most common strategy is to allocate a block in the cache, called write allocate. The block is fetched from memory and then the appropriate portion of the block is overwritten. An alternative strategy is to update the portion of the block in memory but not put it in the cache, called no write allocate.

As one probably can conclude, write-through behaves poorly in terms of performance because when we have a cache miss for write because we should first fetch the words of the block from memory and, after the block is fetched and placed into the cache, we overwrite the word that caused the miss into the cache block but we also write the word to main memory using the full address. These writes will take a long time, likely at least 100 processor clock cycles, and could slow down the processor considerably.

One solution to this problem is to use a write buffer. A write buffer stores the data while it is waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution.

The alternative to a write-through scheme is a scheme called *write-back*. In a write-back scheme, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced. Write-back schemes can improve performance, especially when processors can generate writes as fast or faster than the writes can be handled by main memory; a write-back scheme is, however, more complex to implement than write-through.

Since no data is returned to the requester on write operations, a decision needs to be made on write misses, whether or not data would be loaded into the cache.

- **Write allocate** (also called fetch on write): Data at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read misses.
- **Write no-allocate** also called write around: Data at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, data is loaded into the cache on read misses only.

Both write-through and write-back policies can use either of these write-miss policies, but usually they are paired in this way:[6]

A write-back cache uses write allocate, hoping for subsequent writes (or even reads) to the same location, which is now cached.

A write-through cache uses no-write allocate. Here, subsequent writes have no advantage, since they still need to be written directly to the backing store.

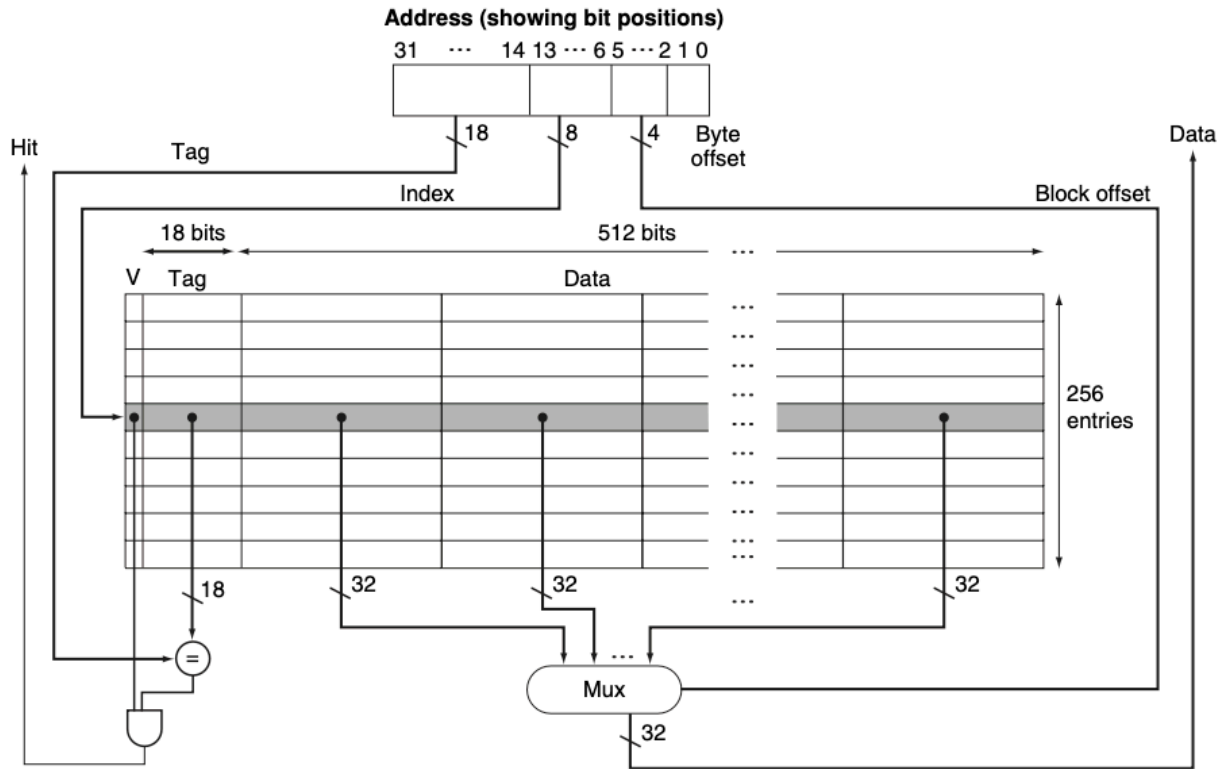


Figure 2.4: Cache

3. VISUAL EDUCATIONAL SIMULATORS

Computer Architecture courses are almost always encountered during the first semesters of a student's academic career. MIPS might seem relatively simple, but it is not. MIPS code, even simplified, remains an assembly language. Programming in an assembly language is very cumbersome, error-prone and very counter-intuitive because assembly is just an interface for machine code. It is the lowest level (closest to the machine) for programming. This is why most of the students usually get lost when they try to understand how assembly works. Moreover, even understanding the theoretical concepts of Computer Architecture like how the processor works, what is pipeline, what is cache and so on, is not a straightforward task. Students at my university, DiT, should consider themselves lucky that they have a professor like Prof. Gizopoulos who is very well-known to the Computer Architecture community and has contributed a lot in the field by doing various research activities. Just to mention a few, his Lab has organized twice (2020, 2021) the prestigious IEEE/ACM International Symposium on Microarchitecture (MICRO) which one of the two most important conferences in Computer Architecture (MICRO-53 and MICRO-54).

However, even the best professors need help. They need tools to visualize what they are explaining to the students, to demonstrate how an assembly program runs and, most importantly, to allow students to experiment themselves with assembly. This is why there are so many visual simulators for MIPS architecture which have been designed and developed by instructors or students in order to aid in the teaching of this difficult subject.

Currently, Prof. Gizopoulos uses a simulator called QtSpim (Figure 3.1) for Computer Architecture I course and WinMIPS64 (Figure 3.2) for Computer Architecture II course. QtSpim offers a huge variety of instructions and it remains the greatest tool if someone wants to experiment a lot with MIPS assembly language without delving into details of the hardware implementation of the CPU. It is basically a great interface to MIPS assembly language but it comes with little to no visualization of the processor and cache. These deficiencies greatly limit its usability for more advanced Computer Architecture courses.

WinMIPS64, on the other hand, offers much more visual docks including visual representation of the pipeline stages, statistics dock which displays how many stalls the program has had and also the type of the stall (data, branch, structural hazards). However, WinMIPS64 also lacks cache support. Because of this lack of cache support, Prof. Gizopoulos uses Dinero IV (Figure 3.3) simulator alongside WinMIPS64 which provides the user with a very detailed presentation about cache misses on all levels of cache and also splits them into instruction and data misses. But there is no graphical representation for the caches when Dinero is used. Just the statistics which need a lot of effort to understand and analyze. Another drawback of WinMIPS64 is that the program execution inside the simulator is very slow.

We should also mention two very important and frequently used simulators, MipsIt (Figures 3.4, 3.5, 3.6) and MARS (Figure 3.7).

MipsIt is the baseline for the QtMips simulator. It is a very old simulator which was developed only for Windows systems.

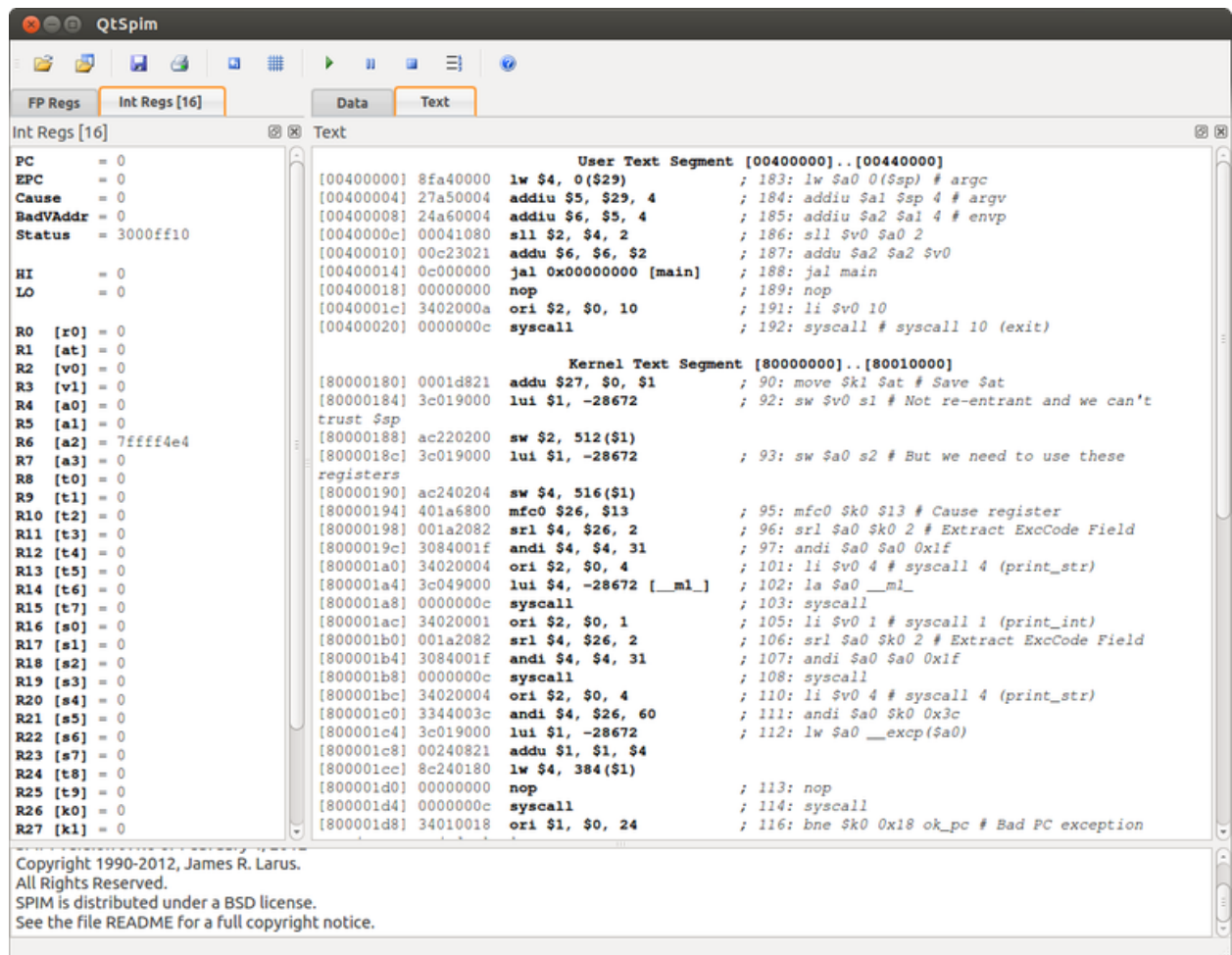


Figure 3.1: QtSPIM

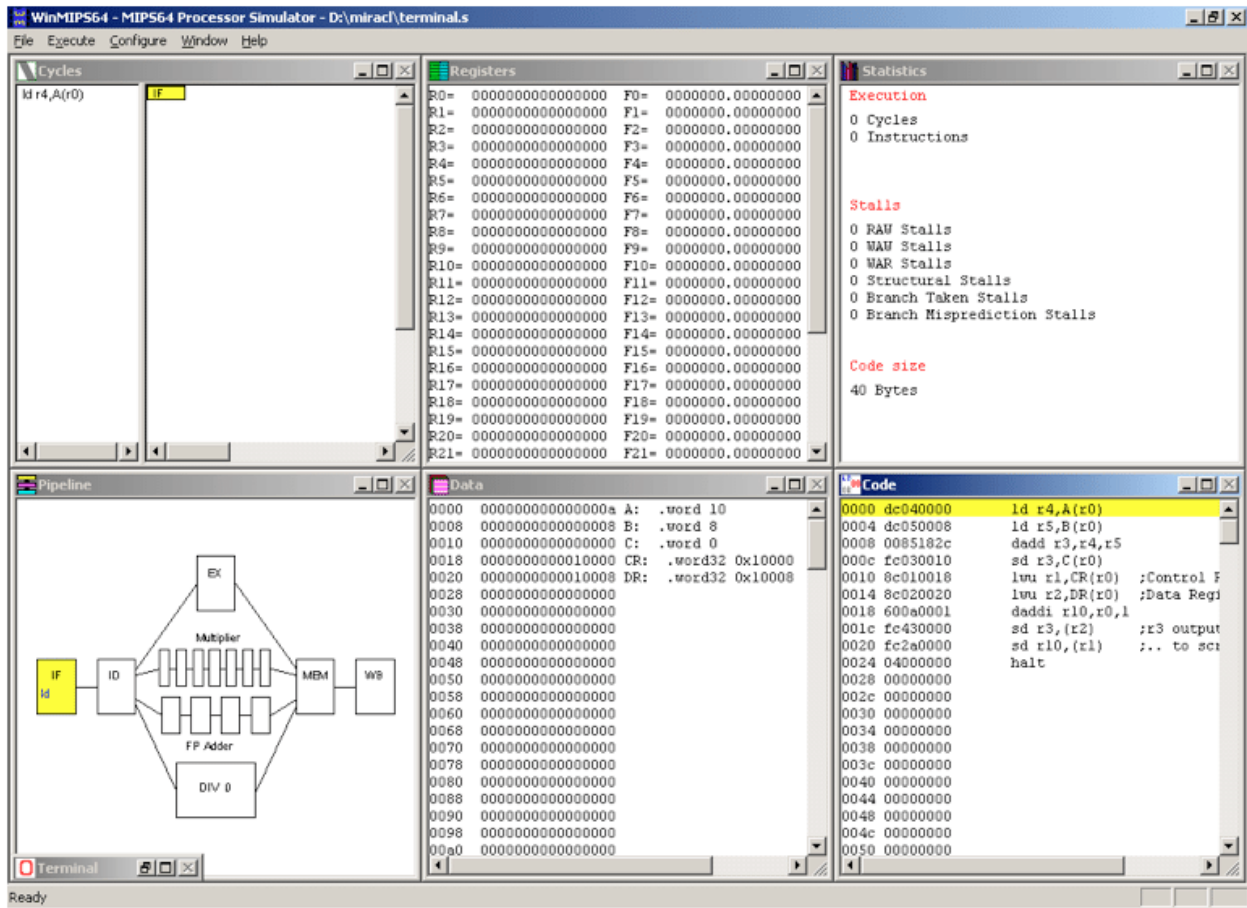


Figure 3.2: WinMIPS64

MipsIt offers some desirable and undesirable features. Some desirable features include:

- Program and data cache simulation.
- Usage statistics.
- Configurable environment.

Some of the undesirable features are:

- Bugs in fully pipelined mode.
- Often crashes.
- Non open-source, making it impossible for developers to fix anything.

MARS is a simulator written using Java language which makes it cross-platform. MARS uses its own MIPS assembler, offers an almost complete MIPS ISA, a MIPS code editor, attachable simulated hardware and other various tools such as instruction execution visualization.

However, it does not support pipelined CPU and does not offer cache simulation.

All of the previously mentioned simulators have one thing in common. They have low simulator throughput. Simulator throughput refers to the number of simulated instructions executed per second.

```

moda@netsec:~/computer_architecture/d4-7
--ll-isize 1024
--ll-dsize 1024
--ll-lbsize 8
--ll-gbsize 8
--ll-lbstride 8
--ll-dbstride 8
--ll-lassoc 1
--ll-dassoc 1
--ll-irepl 1
--ll-drepl 1
--ll-ifetch d
--ll-dfetch d
--ll-dwalloc a
--ll-dwback a
--skipcount 0
--flushcount 0
--maxcount 0
--star-interval 0
--informat d
--on-trigger 0x0
--off-trigger 0x0

---Simulation begins.
---Simulation complete.
ll-icache
Metrics                Total          Instrn         Data           Read           Write          Misc
-----
Demand Fetches         757341         757341         0              0              0              0
Fraction of total      1.0000         1.0000         0.0000         0.0000         0.0000         0.0000

Demand Misses          223575         223575         0              0              0              0
Demand miss rate       0.2992         0.2992         0.0000         0.0000         0.0000         0.0000

Multi-block refs       0
Bytes From Memory      1788600
( / Demand Fetches)   2.3617
Bytes To Memory        0
( / Demand Writes)    0.0000
Total Bytes r/w Mem    1788600
( / Demand Fetches)   2.3617

ll-dcache
Metrics                Total          Instrn         Data           Read           Write          Misc
-----
Demand Fetches         242661         0              242661         159631         83030          0
Fraction of total      1.0000         0.0000         1.0000         0.6578         0.3422         0.0000

Demand Misses          51023         0              51023         31547         19476          0
Demand miss rate       0.2103         0.0000         0.2103         0.1976         0.2346         0.0000

Multi-block refs       0
Bytes From Memory      408184
( / Demand Fetches)   1.6821
Bytes To Memory        191680
( / Demand Writes)    2.3086
Total Bytes r/w Mem    599864
( / Demand Fetches)   2.4720

---Execution complete.
(modas@netsec d4-7) $

```

Figure 3.3: Dinero IV Simulator

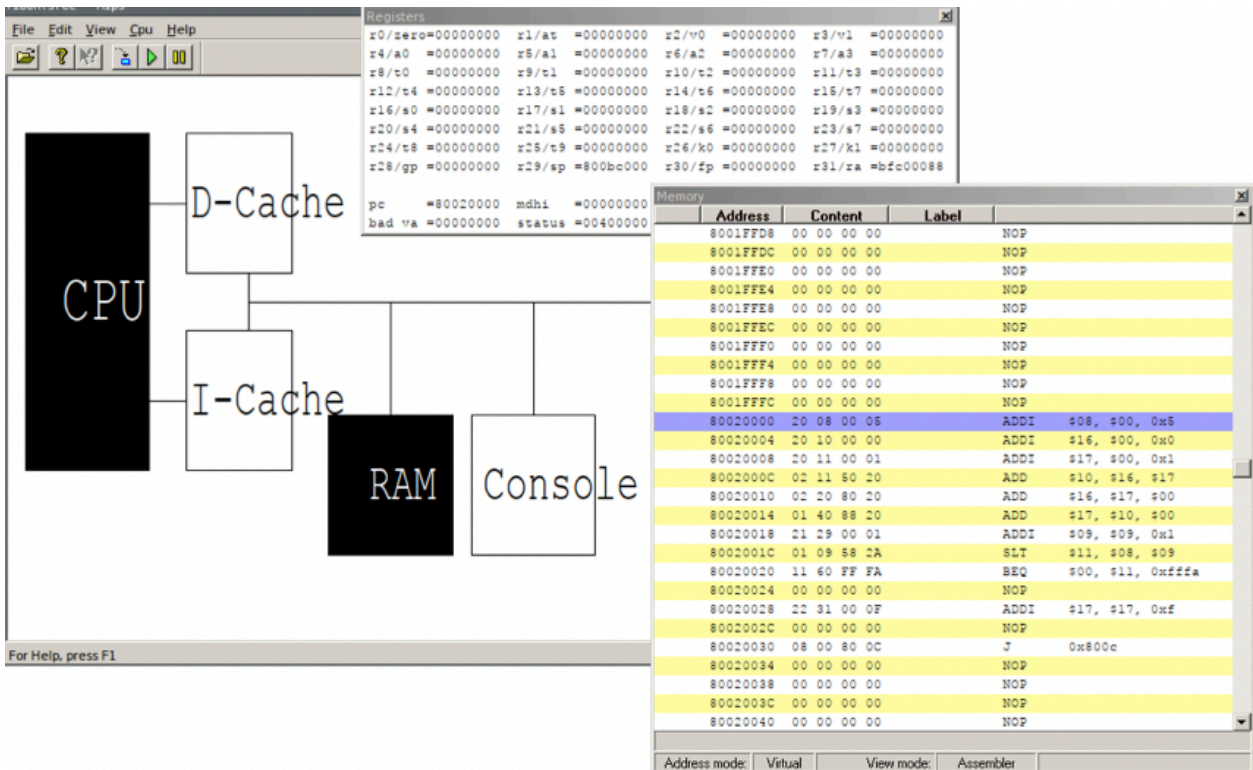


Figure 3.4: MipsIt Windows

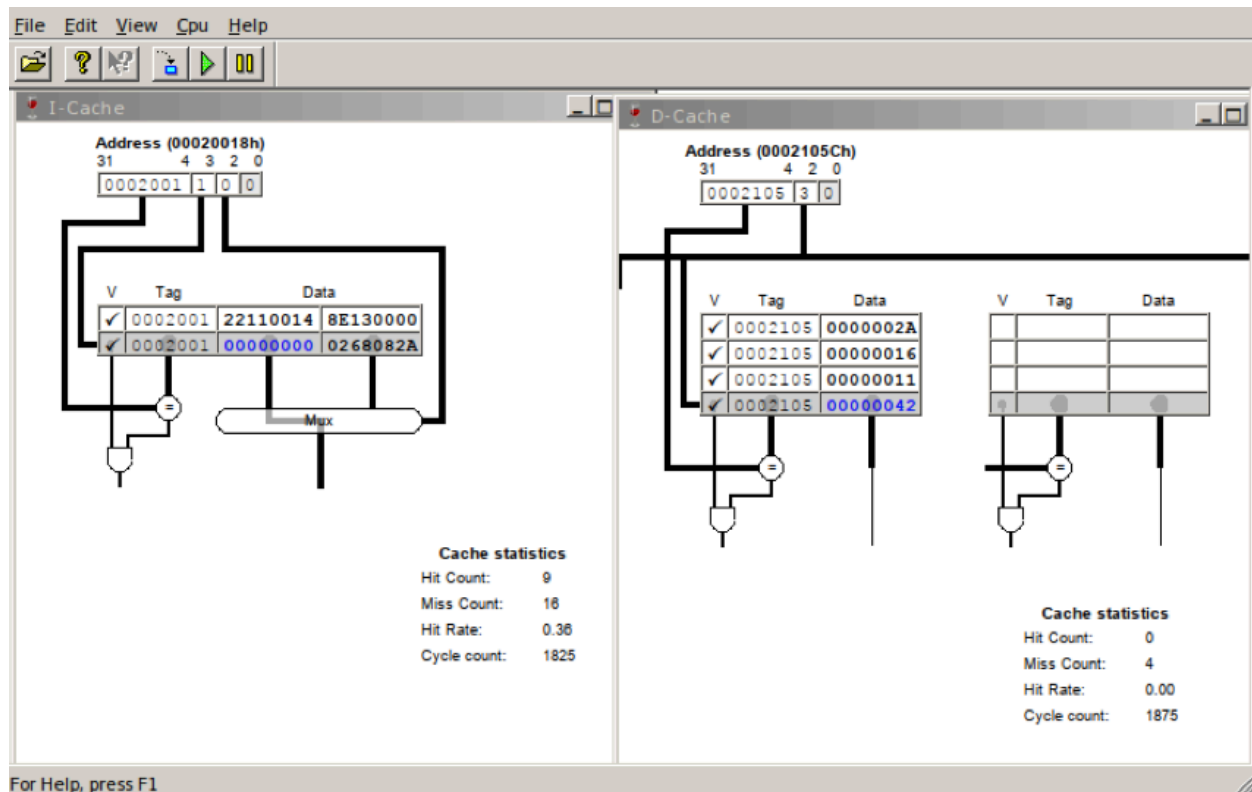


Figure 3.5: MipsIt I/D-Cache

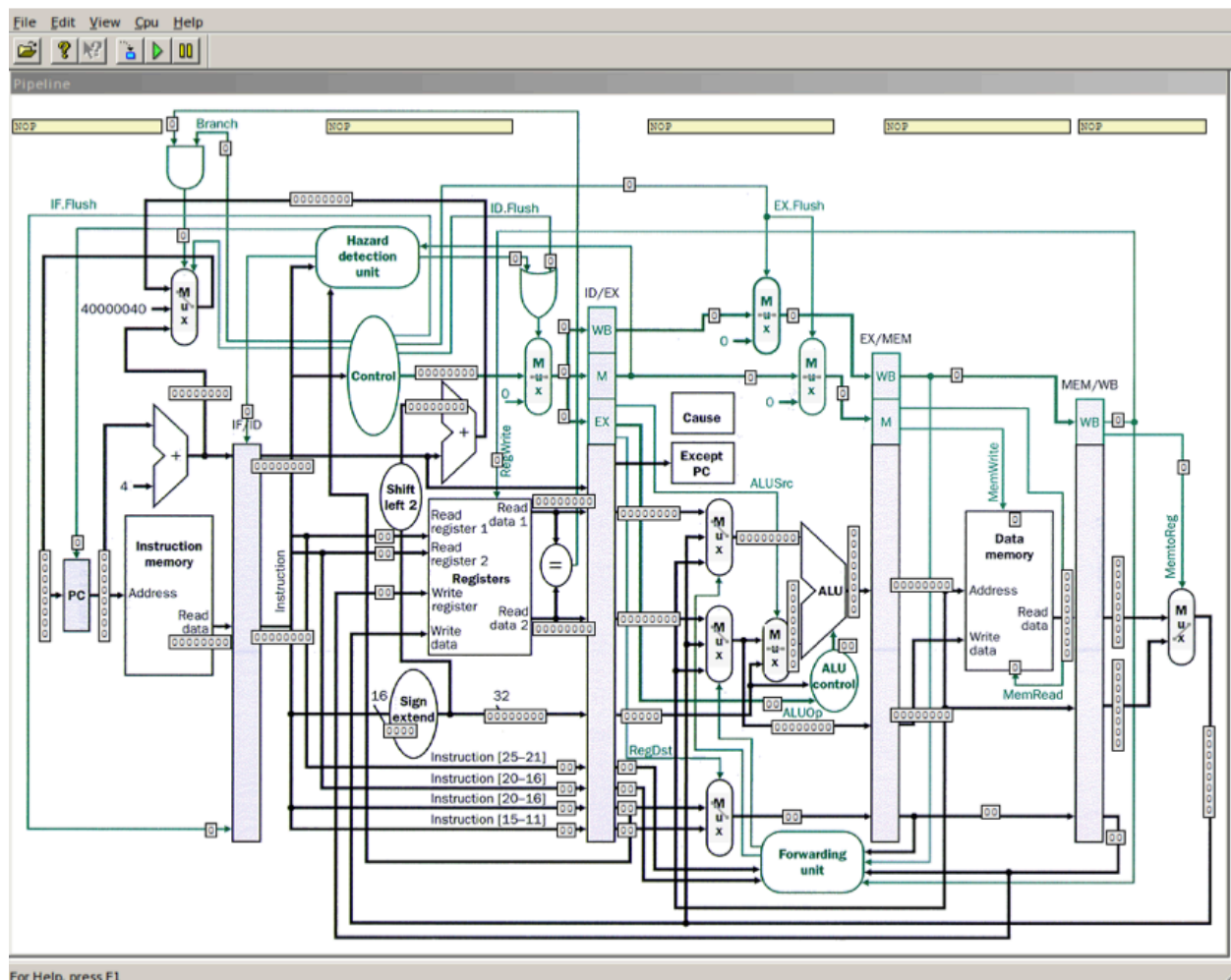


Figure 3.6: MipsIt Pipeline

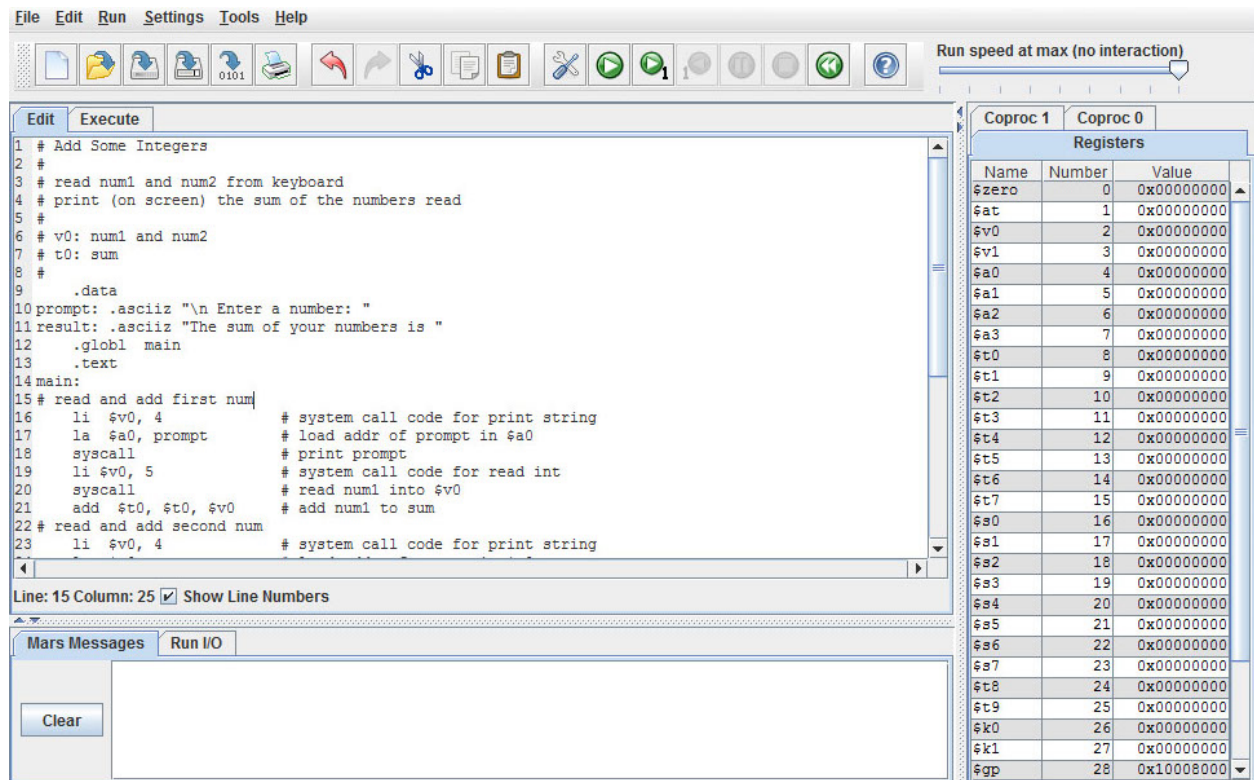


Figure 3.7: MARS

Having low simulation throughput can be a factor of many things such as the programming language used to implement the simulator or the graphical framework used. On table 3.1 we present the throughput of each one of these simulators. We use the same MIPS benchmark program to measure each one of them.

Table 3.1: Throughput per simulator

Simulators & Throughput	
Simulator	Throughput
WinMips	24576
MARS	26810
MipsIt	1200
QtMips	680

As we can see, compared to non-visual simulators, all the above visual simulators have a very low throughput. The more information is displayed / visualized, the less the throughput is. This makes it hard, if not impossible, to execute complex programs in these simulators.

4. QTMIPS

In this chapter we will discuss about QtMips tool. There are two versions of this tool available, a CLI and a GUI one. We will only discuss about the GUI one but the CPU emulation is an independent part so it applies to both versions.

QtMips simulates MIPS processor quite accurately. It provides a detailed visualisation of the core and the user can easily identify all the core parts of the processor like ALU, Control Unit (the unit that emits all signals), Program/Data Memory, Registers, Multiplexers, Adders. What is really helpful is that all connections between these parts are also included in the visualisation and students can easily connect the dots of the theoretical background to the actual truth. Moreover, QtMips includes L1 instruction and data cache as well as statistics regarding cache misses/hits, even program speedup by using cache. QtMips provides docks for each one of these components. The great thing is that all these docks/views are updated during the execution of the program, thus allowing the user to get a better grasp of how the CPU works. In the remaining of this chapter we will discuss these things in a more detailed manner and observe how they look like.

4.1 Registers

Registers dock contains all registers and their corresponding values at any time during the program execution. The values are presented using hexadecimal notation and are updated while the program executes. This makes easy for the user to debug the program, understand how instructions work and generally observe the behavior of the program.



Registers															
\$0/zero	0x0	\$1/at	0x0	\$2/v0	0x0	\$3/v1	0x0	\$4/a0	0x0	\$5/a1	0x0	\$6/a2	0x0	\$7/a3	0x0
\$8/t0	0x0	\$9/t1	0x0	\$10/t2	0x0	\$11/t3	0x0	\$12/t4	0x0	\$13/t5	0x0	\$14/t6	0x0	\$15/t7	0x0
\$16/s0	0x0	\$17/s1	0x0	\$18/s2	0x0	\$19/s3	0x0	\$20/s4	0x0	\$21/s5	0x0	\$22/s6	0x0	\$23/s7	0x0
\$24/t8	0x0	\$25/t9	0x0	\$26/k0	0x0	\$27/k1	0x0	\$28/gp	0x0	\$29/sp	0xbffff00	\$30/fp	0x0	\$31/ra	0x0
pc	0x400018	lo	0x0	hi	0x0										

Figure 4.1: QtMips Registers Dock

4.2 Program Memory

Program dock displays all MIPS instructions that are loaded for execution. It displays the address where each instruction is located, the binary representation (in hexadecimal notation) of the instruction as well as the instruction in human readable form (MIPS assembly language). Not only that, it colors each instruction based on the pipeline stage it is currently at which provides a great visualisation of how pipeline works as the user is able to observe some important details, for example the bubbles inserted to pipeline when a hazard occurs.

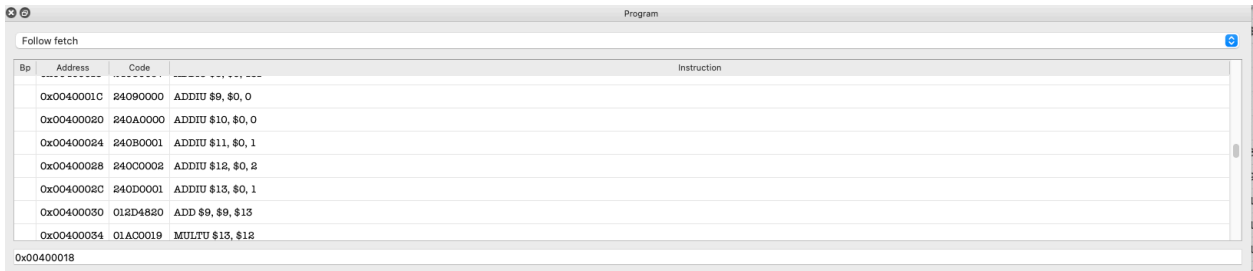


Figure 4.2: QtMips Program Dock

4.3 Data Memory

The representation of data memory layout is also very detailed and flexible. It allows the user to observe the values of each address in memory as well as subsequent addresses after. The reason I called it flexible is because the user can choose to see the memory in terms of bytes, half-words, or words which is very handy when someone is examining what is happening in-memory when he/she runs their program. Another very interesting feature of this dock is that it also displays if a value is cached or not.

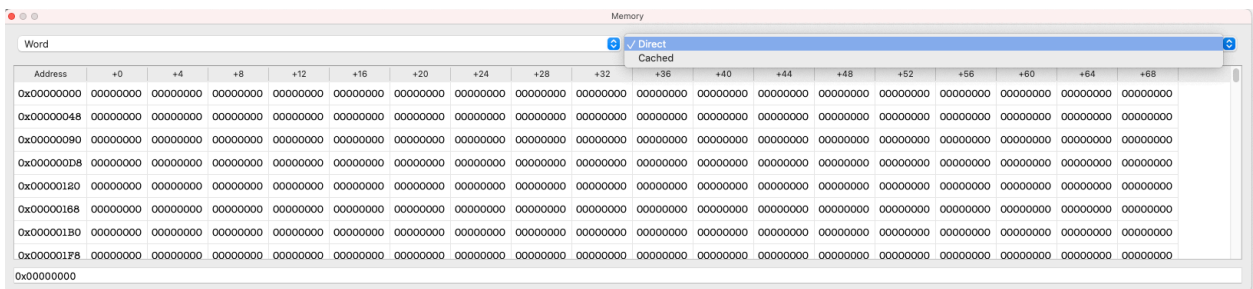


Figure 4.3: QtMips Memory Dock

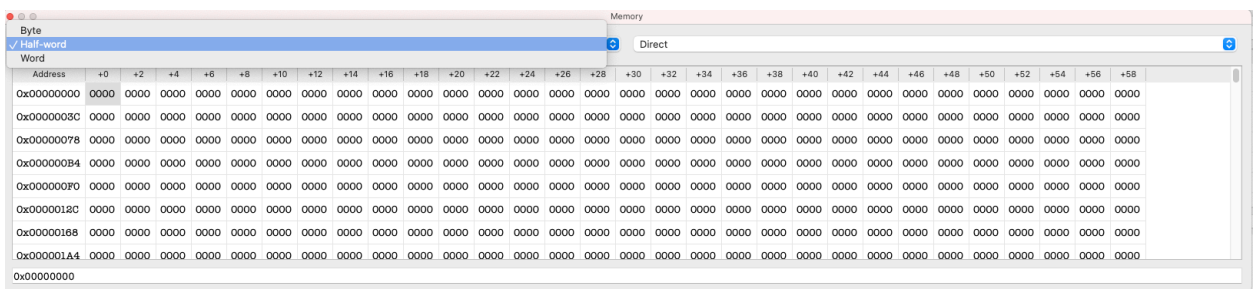


Figure 4.4: QtMips Memory Dock

4.4 Program Execution

QtMips allows for step-by-step program execution in the following ways:

- 1-step program execution.
- 2-step program execution.
- 5-step program execution.

- 10-step program execution.
- Maximum speed.

Users can also pause the program any time.

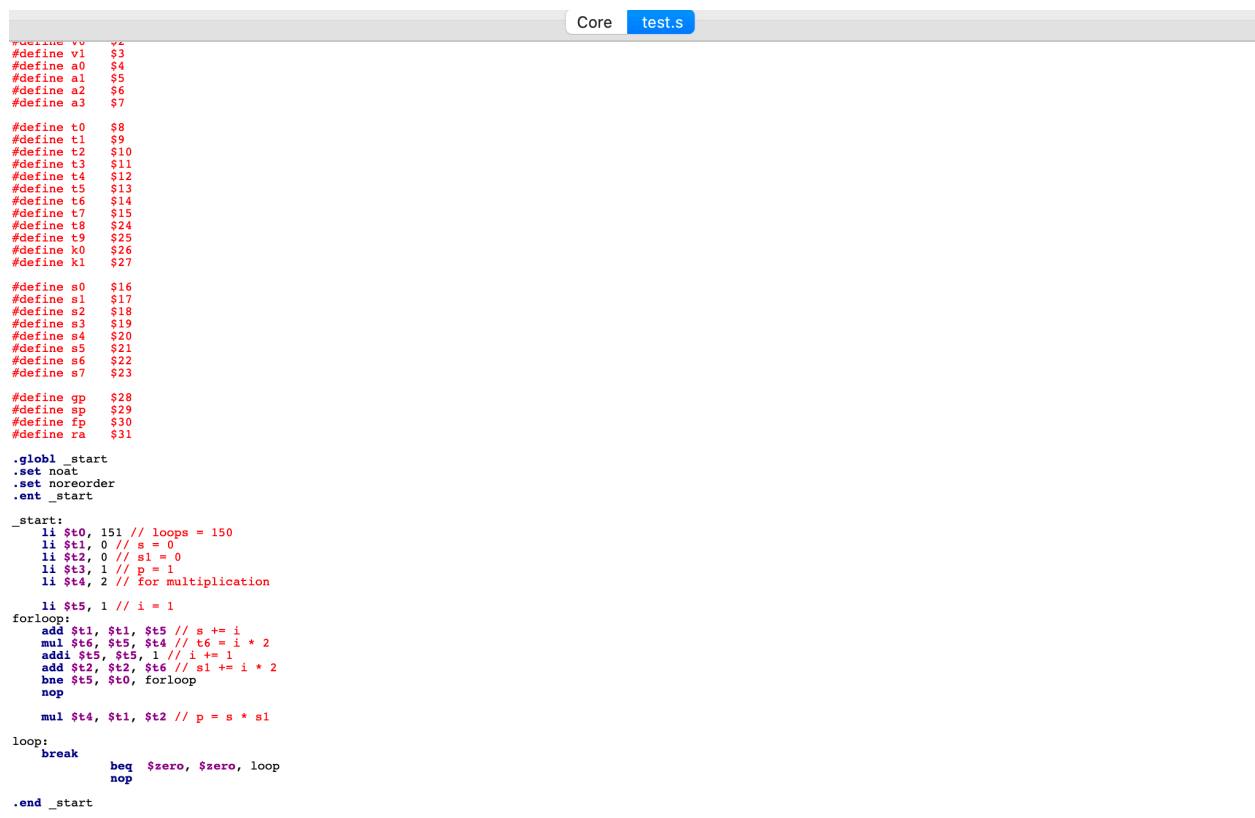
This feature is great for teaching purposes as students can observe in a very detailed manner the effects of each instruction on memory, registers and pipeline.

It is also very handy for debugging since students can step on each instruction and better understand the behavior of their program.

4.5 Code Editor & Compiler

QtMips, similarly to MipsIt, also offers a code editor, as displayed in Figure 4.5, which is very convenient as users can alter their code instantly and re-run their program.

A compiler is also embedded in the program.



```

Core test.s
#define v0 $2
#define v1 $3
#define a0 $4
#define a1 $5
#define a2 $6
#define a3 $7

#define t0 $8
#define t1 $9
#define t2 $10
#define t3 $11
#define t4 $12
#define t5 $13
#define t6 $14
#define t7 $15
#define t8 $24
#define t9 $25
#define t0 $26
#define k1 $27

#define s0 $16
#define s1 $17
#define s2 $18
#define s3 $19
#define s4 $20
#define s5 $21
#define s6 $22
#define s7 $23

#define gp $28
#define sp $29
#define fp $30
#define ra $31

.globl _start
.set noat
.set noreorder
.ent _start
_start:
    li $t0, 151 // loops = 150
    li $t1, 0 // s = 0
    li $t2, 0 // s1 = 0
    li $t3, 1 // p = 1
    li $t4, 2 // for multiplication

    li $t5, 1 // i = 1
forloop:
    add $t1, $t1, $t5 // s += i
    mul $t6, $t5, $t4 // t6 = i * 2
    addi $t5, $t5, 1 // i += 1
    add $t2, $t2, $t6 // s1 += i * 2
    bne $t5, $t0, forloop
    nop

    mul $t4, $t1, $t2 // p = s * s1

loop:
    break      beq $zero, $zero, loop
    nop
.end _start

```

Figure 4.5: QtMips Code Editor

4.6 OS Emulation

QtMips also emulates the OS, by providing system calls, interrupts, exceptions (like overflow or zero division) and, generally, operating system services. The various configurations for OS emulation are displayed on Figure 4.6

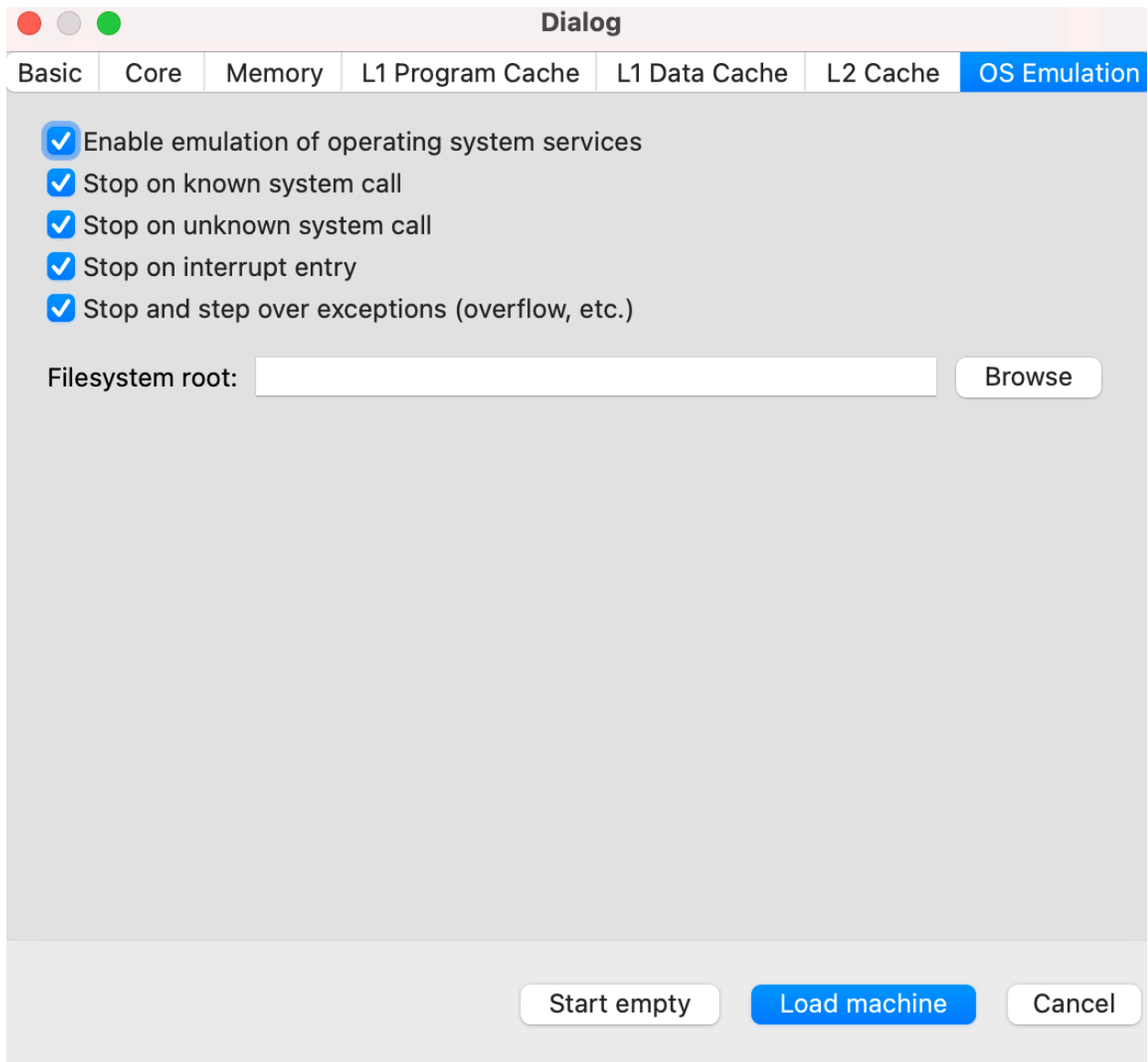


Figure 4.6: QtMips OS Emulation

4.7 L1 Program/Data Cache

QtMips visualises L1 Program Cache by showing which program address is accessed (displays only one entry at once) at any moment during the program execution. Basically the instruction that was fetched is the one that will get accessed. This dock is also very flexible because it grows or shrinks based on degree of associativity, block size and number of sets. Figure 4.7 shows an L1 cache dock with the following configurations:

- *block size*: 8 words.
- *degree of associativity*: 2
- *number of sets*: 2

As you can see the address in blue is the one that the program is accessing right now. The various statistics which are displayed provide substantial assistance for a student who is trying to learn more about caches.

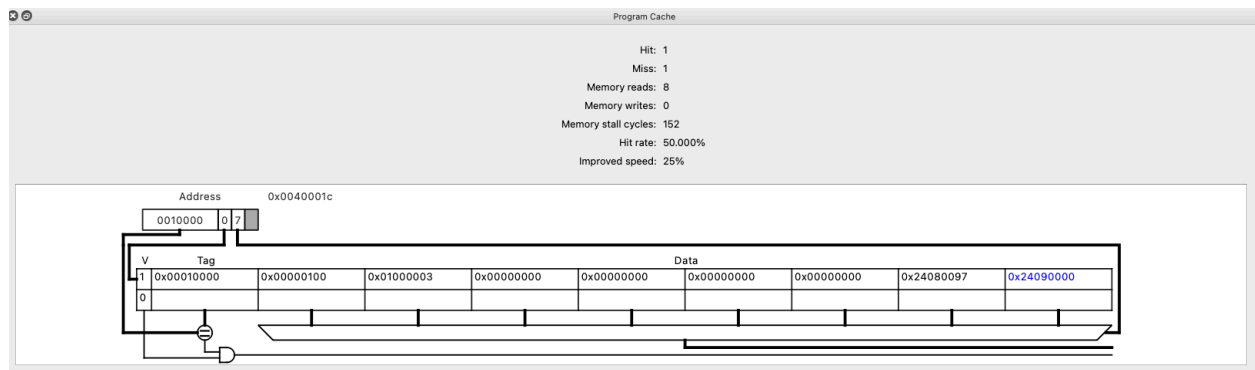


Figure 4.7: QtMips L1 Program Cache Dock

L1 Data Cache dock is identical to Program Cache dock.

4.8 Coreview

This is the main dock of QtMips. On Figure 4.8 we display the coreview when a user has configured pipelined CPU, whereas on Figure 4.9 we display the coreview of a single-cycle CPU. On both of these modes of the coreview, user is able to identify all the processor components, how they are connected to each other, what values are passed through each of the components, what hardware circuits are used (adders, multiplexers and so on), which signals are emitted by the control unit and generally visualises how data flows in the CPU.

The main difference between pipelined and non-pipelined mode is that in pipelined mode the pipeline stages are also visualised as well as which instruction is in which stage at any given moment of the program execution.

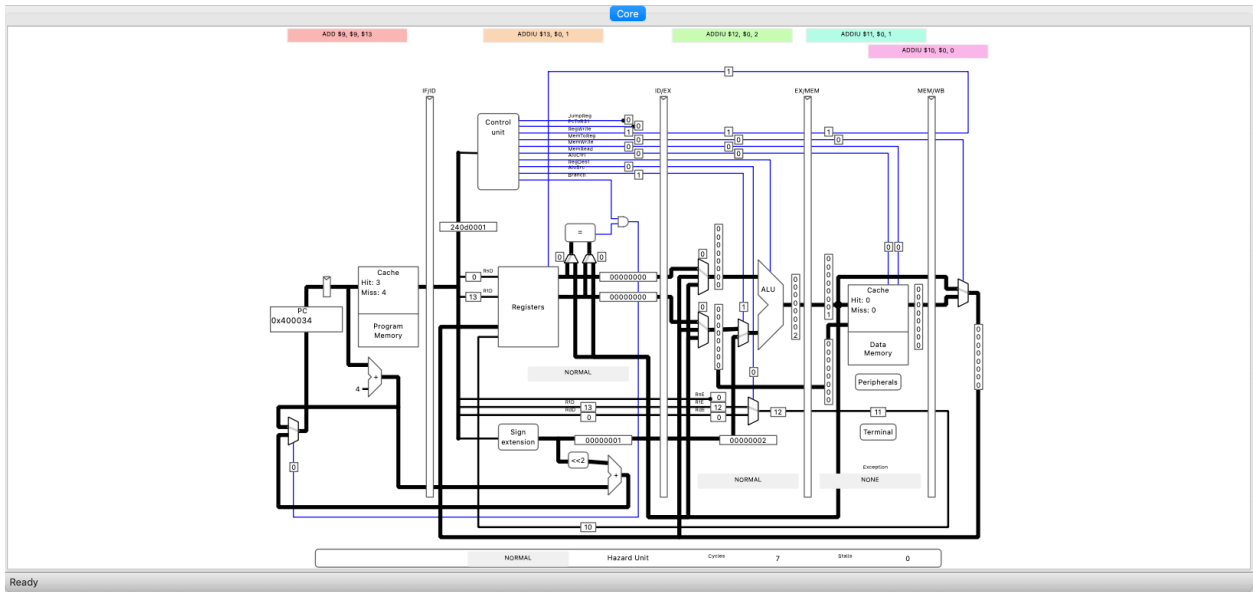


Figure 4.8: QtMips Coreview (Pipelined)

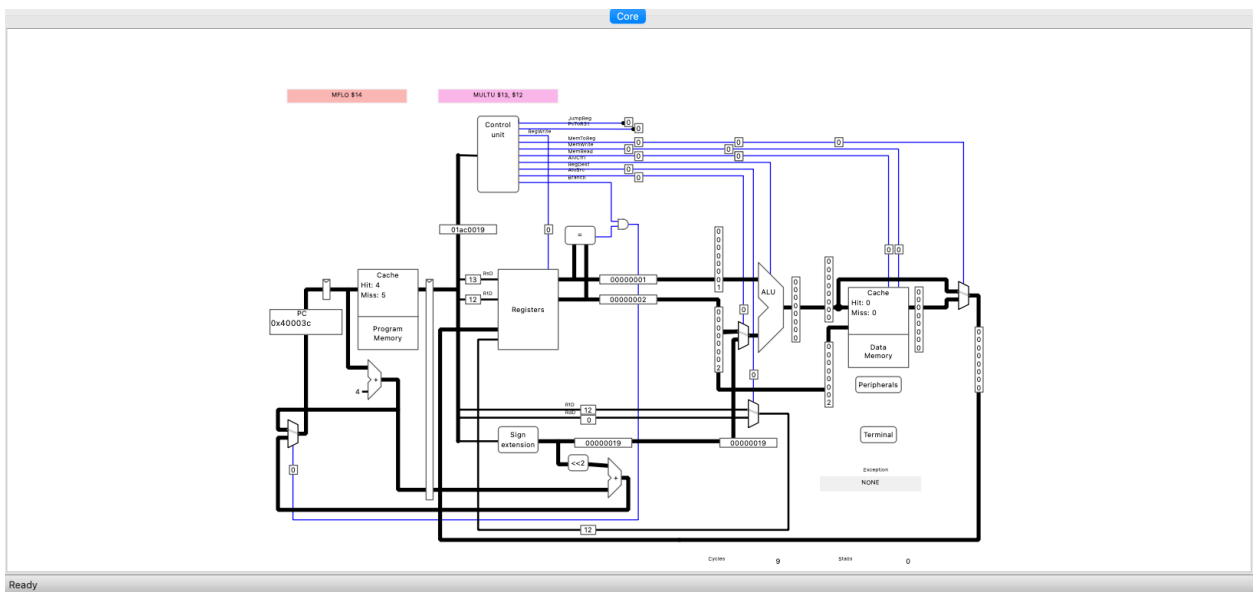


Figure 4.9: QtMips Coreview (Single-Cycle)

5. EXTENSIONS TO QTMIPS

Despite all the functionality QtMips offers, me and Prof.Gizopoulos thought that it still lacks some important features that are required for an advanced Computer Architecture course. To be more specific, QtMips provides only one level of cache, L1. This however is far from reality even in relatively low-end computing systems. We decided to add one more cache level (L2), which is unified. Unified means that this cache is not separated to a data and instruction component, it contains both. Moreover, as you might have noticed, QtMips is not very versatile regarding branch handling. Users are forced to use delay slot in a pipelined mode which means that they should add 1 nop instruction after each jump or branch instruction if they do not have a useful, and most importantly, independent on the branch instruction to place in the delay slot.

Here, its important to note why users should add 1 instruction (either nop or other, if possible) and not more. QtMips resolves branches on ID stage instead of EX. This means that if an instruction is fetched, we would have to stall the pipeline for 1 cycle to decide which instruction should be fetched next (depends on branch result). QtMips does not stall, so the next instruction after a branch will always be fetched. Therefore, if one wants to guarantee that their program will be executed correctly, they should either add 1 nop or they should add a useful instruction that does not depend on the branch.

As mentioned, branches are resolved on ID stage. We thought that this also limits potential experimentation for users so we decided to add another option, resolving branches on EX stage. We also updated the coreview to include the components for branch predictor (if its enabled). Before presenting the implementation for our branch predictor, lets discuss a few things about predictors first.

5.1 Branch Predictor

Branch predictor is a hardware component that, as its name implies, attempts to predict if a branch will be taken or not. If implemented efficiently, branch predictors can achieve very high performance boost. The reason behind this is that branches are very common in every program and adding nop instructions or bubbles after every branch would cause a huge bottleneck for performance. Modern branch predictors can achieve more than 90% (!) accuracy for programs. But how are predictors able to do such a great job? There is a rule that is called 90/10 which might explain why these branch predictors are so accurate. The rule says: *In software engineering, it is often a better approximation that 90% of the execution time of a computer program is spent executing 10% of the code*, which basically means that most of the execution time of a computer program is spent looping again and again and executing the same piece of code. If branch predictors were able to identify when a program is looping then its easy to predict the result of each branch. This is exactly what happens but let's dive into the details of how this could be implemented.

5.1.1 Static Branch Predictors

Static prediction is the simplest branch prediction technique because it does not rely on information about the dynamic history of code executing. Instead, it predicts the outcome of a branch based solely on the branch instruction. These types of branch predictors are completely ignorant of the program and always predict the same outcome, taken for example. Naive as it is, this would yield some good results compared to always stalling on branches. That is because if a prediction is wrong then a bubble will be inserted to replace the wrongly-fetched instruction, but this has the same penalty as stalling on every branch or adding nop after every branch. By taking into account the 90/10 rule, we could argue that a branch predictor which always says taken would yield a nice performance boost. Indeed, these type of branch predictors can achieve an accuracy of 60-70%. But we can do better. If branch predictors could somehow keep a history of previous branch outcomes they could easily adapt their predictions for the next branches.

5.1.2 Dynamic Branch Predictors

Dynamic branch prediction uses information about taken or not taken branches gathered at run-time to predict the outcome of a branch. Dynamic branch predictor are separated in many categories. We will keep our focus on *one-level* branch predictors. One-level branch predictors are also split in two categories, those who use 1-bit for history and 2-bit. These two types of predictors are basically FSM - Finite States Machines that alter their state based on the result of previous branches.

1-bit branch predictors (Figure 5.1) use a 1-bit saturating counter (essentially a flip-flop) that records the last outcome of the branch. If the outcome of the last branch was not taken, then the prediction for the next branch will also be not taken. Same goes for the taken case. This is the most simple version of dynamic branch predictor possible, although it is not very accurate. 2-bit branch predictors (Figure 5.2), on the other hand, are a state machine with four states:

- Strongly not taken.
- Weakly not taken.
- Weakly taken.
- Strongly taken.

When a branch is evaluated, the corresponding state machine is updated. When branches are evaluated as not taken the machine starts moving towards the taken area, but the next state depends on the current state. If it is strong not taken it goes to weak not taken, etc. The advantage of the two-bit counter scheme over a one-bit scheme is that a conditional jump has to deviate twice from what it has done most in the past before the prediction changes. This behavior aims to reduce branch mispredictions for loops, which is where programs spend most of their execution time.

Obviously, in order for branch predictors to be useful, the prediction should be done on the first stage, when branch is fetched. This requires extra hardware in order to be able

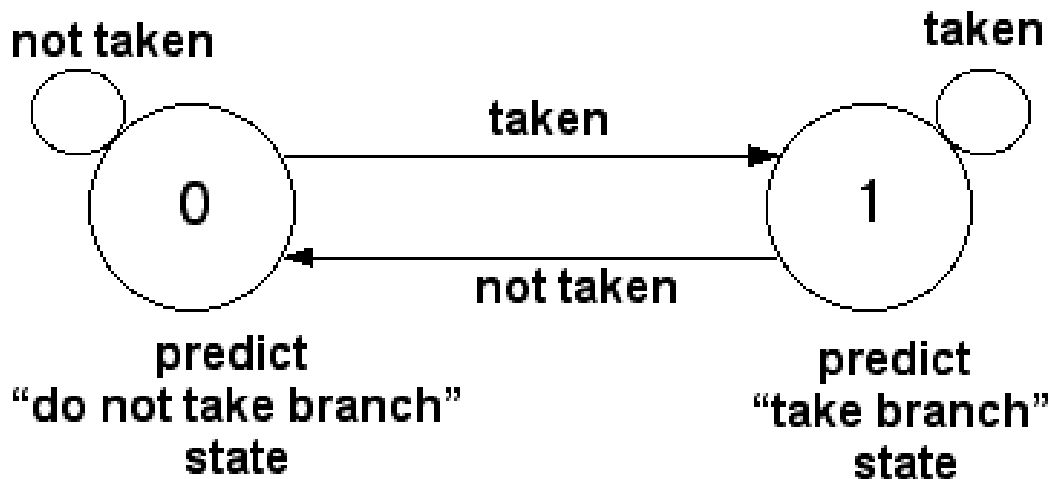


Figure 5.1: 1-Bit Branch Predictor

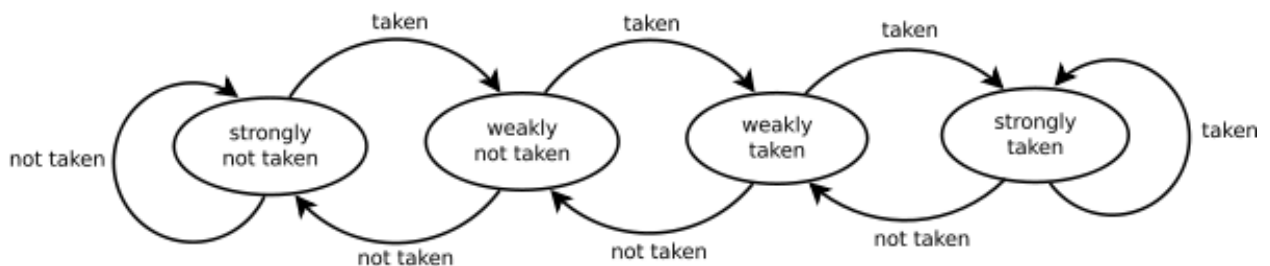


Figure 5.2: 2-Bit Branch Predictor

to know if a the instruction is a branch or jump. But what else do we need? When CPU fetches a branch instruction, the branch predictor should know 2 things:

- The branch history of this instruction (For example, for 1-bit predictors, whether is was taken / not taken).
- The branch address that the program should jump to in case the prediction is taken. Recall that CPU does not have the required hardware to decode or calculate the branch address in IF stage.

For that reason, branch predictor contains 2 buffers (like caches) of the same size. One buffer is used to associate a branch instruction to its branch history and that's why this buffer is called *Branch History Table*. The second buffer is used to associate a branch instruction to the branch address / target that the program should jump to and that's why the second buffer is called *Branch Target Buffer*. It is easy to see that when a branch instruction is initially accessed, both these buffers will contain some values which depend on the implementation of the branch predictor. For example, branch history table might contain NOT TAKEN for all entries. Branch target buffer, however, will contain invalid entries which means that even if branch predictor omits TAKEN, the CPU does not have a way to know where to jump. This is called a BTB miss and it resembles one of caches. When a BTB miss occurs, the pipeline stalls until branch result and target can be evaluated and then BTB is updated with the correct address so that consequent accesses to this branch instruction will have the correct address.

All the concepts mentioned above are integrated into our new version of QtMips tool. They have been added both in the graphical interface and in the core. On Figure 5.3 we display

the coreview of the CPU that is configured to use branch predictor. New docks have also been added that show branch history table (Figure 5.4) and branch target buffer (Figure 5.5) as 2-D arrays and are updated as the program runs.

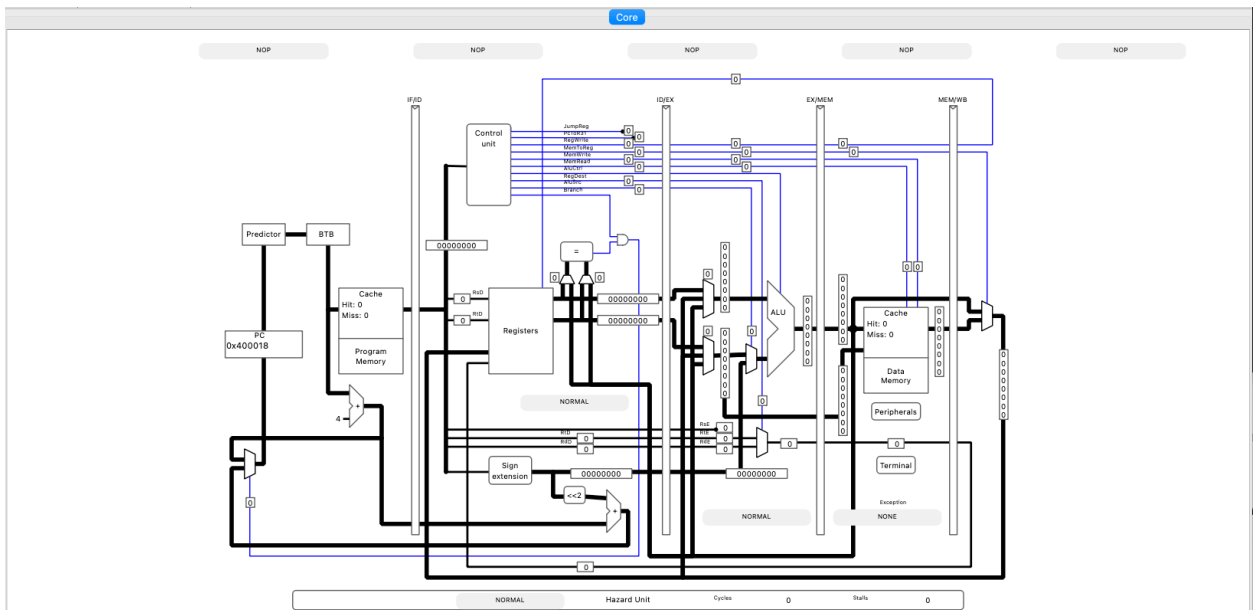


Figure 5.3: QtMips Coreview (Branch Predictor)

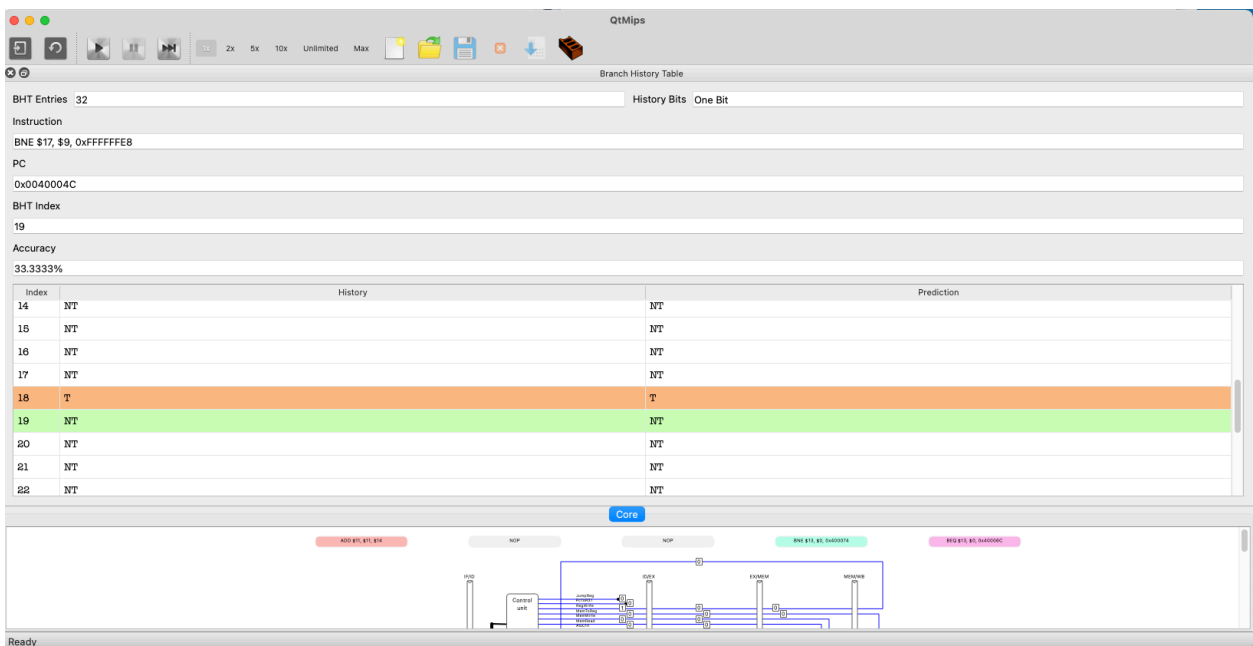


Figure 5.4: QtMips Branch History Table

5.2 Cache

We also decided to add another level of cache memory (L2). This cache does not separate data from instructions and that's why its called *unified*. We also added a new dock (Figure 5.6) displaying the data that is currently accessed from the cache. User is now free to choose between the following options regarding memory:

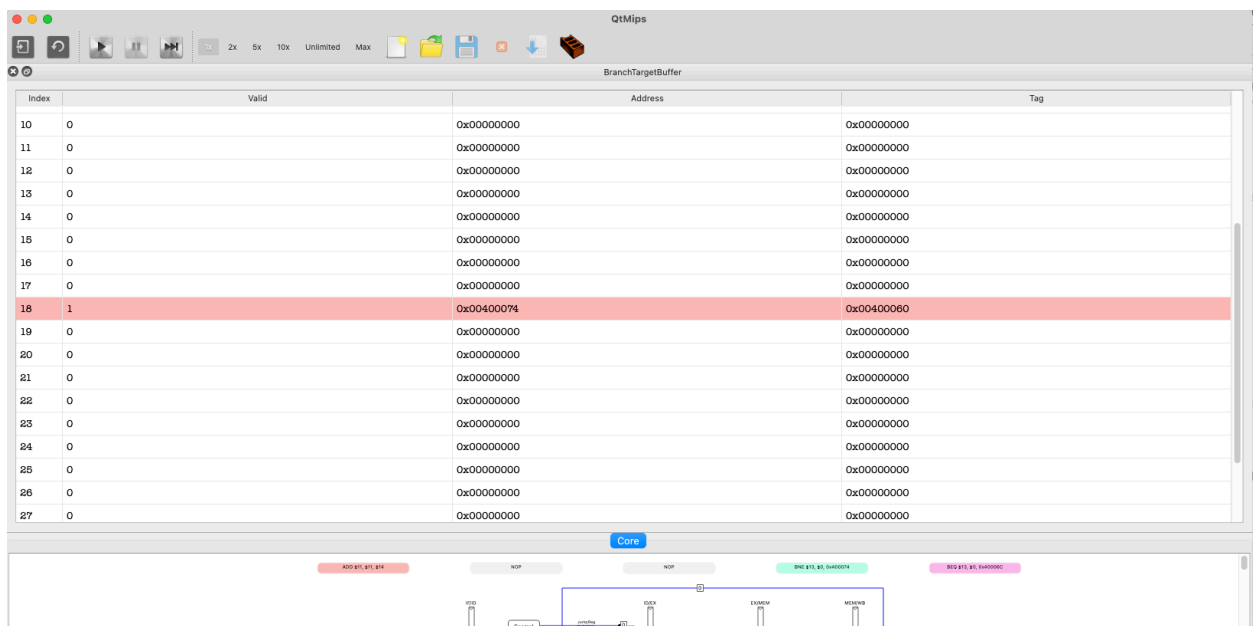


Figure 5.5: QtMips Branch Target Buffer

- DRAM only.
- DRAM + L1 Data Cache.
- DRAM + L1 Program Cache.
- DRAM + L1 Data Cache + L2 Unified Cache.
- DRAM + L1 Program Cache + L2 Unified Cache.
- DRAM + L1 Data Cache + L1 Program Cache + L2 Unified Cache.

We also added some minor tweaks and did some bug fixing for QtMips.

5.3 Cycle Statistics

There was a Cycles counter shown in the coreview which used to count how many cycles the program needed until the current moment. However, this counter did not take into consideration possible stalls or cache misses. We thought this was not very helpful for students so we added a new dock (Figure 5.7) that displays all kinds of stalls that might occur as well as the total number of cycles.

5.4 Representation of Register Values

We also added a feature which allows the user to display register values in decimal, octal and hexadecimal notation instead of only hexadecimal, as displayed on Figure 5.8

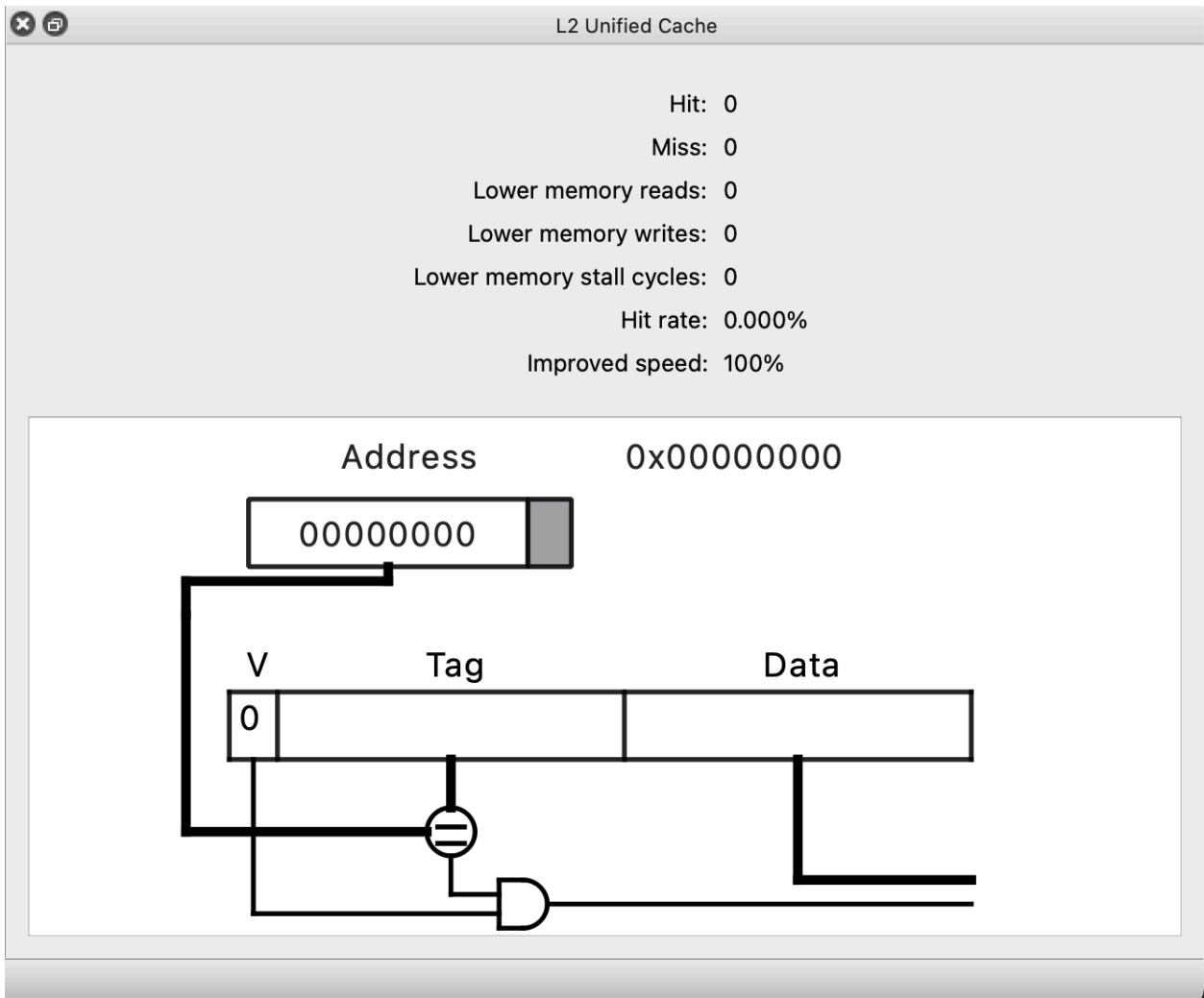


Figure 5.6: QtMips L2 Cache Dock

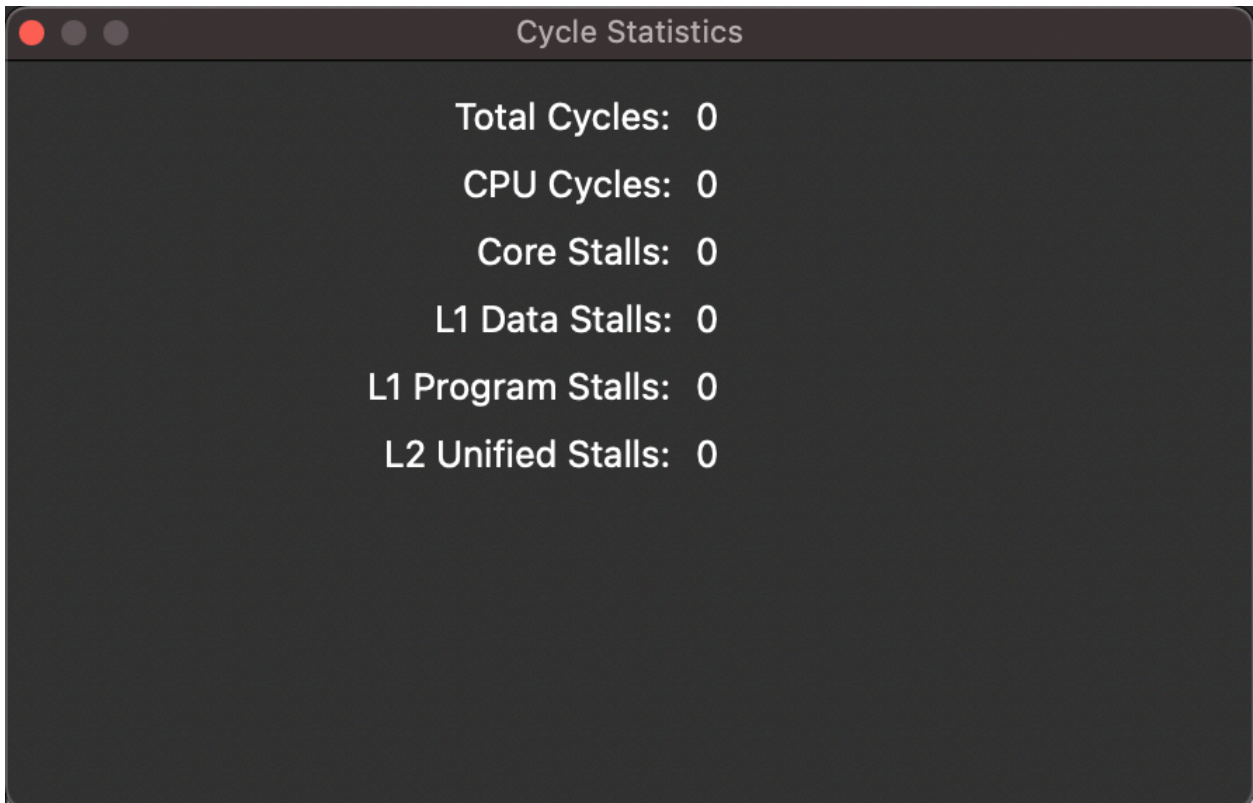


Figure 5.7: QtMips Cycle Statistics Dock



Figure 5.8: QtMips Registers Notation

5.5 Data and Control Hazard Unit

We also separated Hazard Unit to Control Hazard Unit and Data Hazard Unit. The motivation behind this change is because these hazards are presented separately on literature and we would like to give students the option to experiment separately with these hazards. For the same reason we added Stall-on-Branch option for branches where the pipeline stalls as many cycles as needed until a branch is resolved. On Figure 5.9 we display the various configurations regarding data and control hazard units.

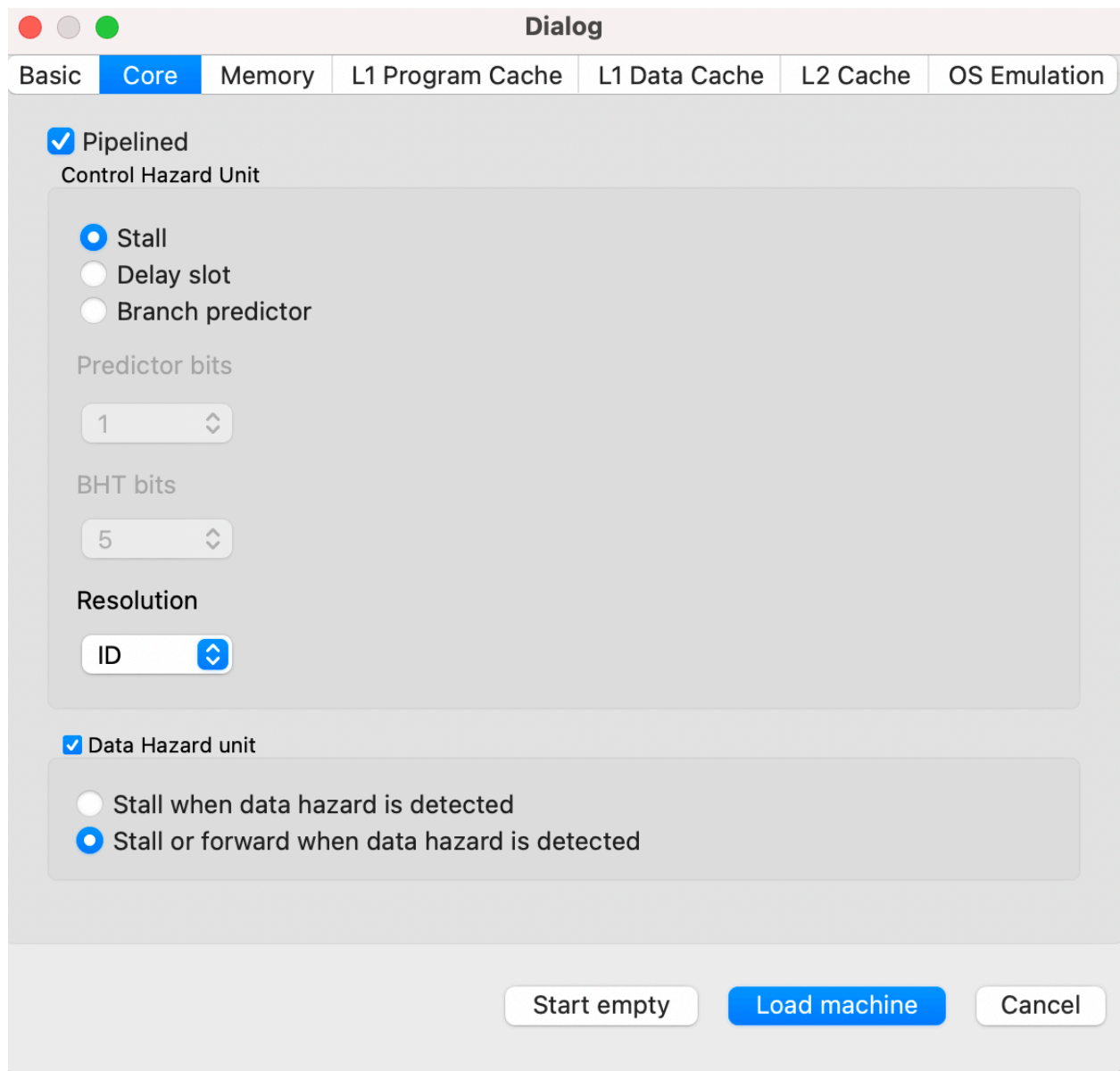


Figure 5.9: QtMips Control & Data Hazard Units

6. EXPERIMENTS

In this chapter we will showcase some program executions that better display what the newly-added functionalities on QtMips offer to the tool in terms of program speedup (cycles before / cycles after) and educationally.

6.1 Branch Predictor

We use the following program as a baseline for our experiments regarding branch predictor:

```
int main(void) {
    int s = 0;
    for (int i = 0 ; i < 1000 ; i++) {
        for (int j = 0 ; j < 1500 ; j++) {
            s += i + j;
        }
    }
}
```

On table 6.1 we observe the total number of cycles for various branch unit configurations. On table 6.2 we observe the speedups for the base configuration (stall-on-branch) vs branch predictor configurations.

Table 6.1: Cycles (Branch)

Cycles (Branch)		
Branch Configuration	Branch Resolution Stage	Cycles
Stall On Branch	ID	10506008
Stall On Branch	EX	12007008
Delay Slot	ID	10506008
Delay Slot	EX	12007008
One-Bit Branch Predictor	ID	9009012
One-Bit Branch Predictor	EX	9013016
Two-Bit Branch Predictor	ID	9009016
Two-Bit Branch Predictor	EX	9013024

Even for this small program that does not execute many loops we can observe a respectable amount of speedup by using the branch predictor. The speedup is significantly higher when resolving branches on EX stage because the penalty without the predictor is 2 cycles instead of 1.

The results between stall on branch and delay slot are the same because no delay slot is filled with a useful instruction. This is not always not be the case. Branch delay slot

Table 6.2: Speedups (Branch)

Speedups (Branch)		
Pair of Configurations	Branch Resolution Stage	Speedup
Stall/One-Bit Branch Predictor	ID	16%
Stall/One-Bit Branch Predictor	EX	33%
Stall/Two-Bit Branch Predictor	ID	16%
Stall/Two-Bit Branch Predictor	EX	33%

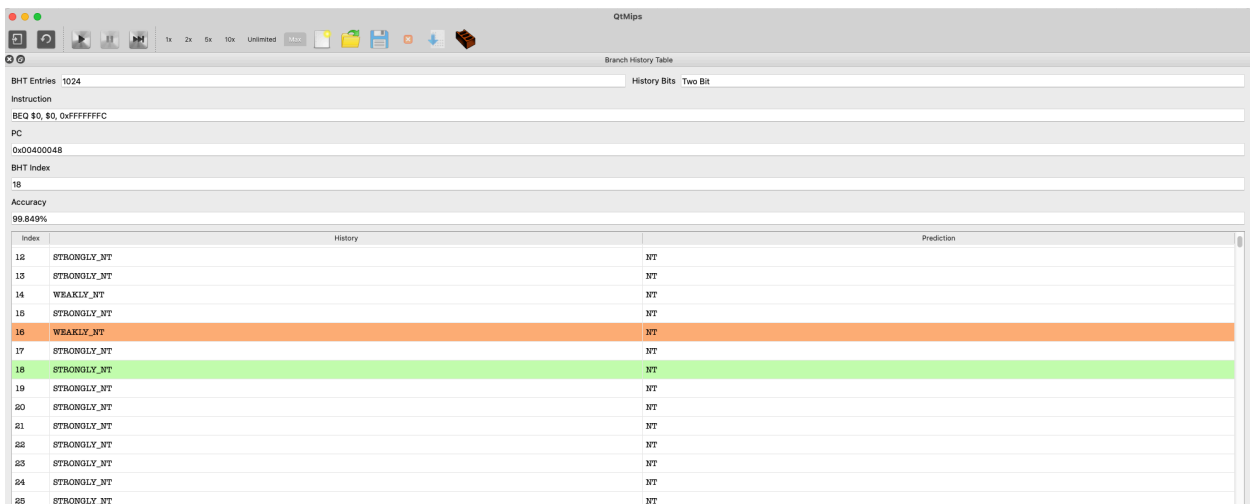


Figure 6.1: QtMips Program Execution (Branch)

is a useful feature that speeds up the execution if a useful and independent instruction is placed just after branch. Below we present a program that utilizes this behavior of delay slot in order to achieve better performance compared to the stall-on-branch version of the program.

To better understand that delay slot is a useful feature, we measured a program that contains useful instructions in the delay slots and we compare it with the stall-on-branch approach. For this program, in which branches are resolved in EX, total cycles with stall-on-branch were 1512 and the total cycles with delay slot were 1212. The program with delay slot enabled is 24% faster.

6.2 L2 Cache

We use the following program as a baseline for our experiments regarding L2 cache:

```
#define ARRAY_LEN 512

int main(void) {
    int arr1[ARRAY_LEN];
    int arr2[ARRAY_LEN];
    int s = 0;

    // Write to arrays.
    for (int i = 0 ; i < ARRAY_LEN ; i++) {
        arr1[i] = i + 1;
    }
    for (int i = 0 ; i < ARRAY_LEN ; i++) {
        arr2[i] = i * 2;
    }

    // Read from arrays.
    for (int i = 0 ; i < ARRAY_LEN ; i++) {
        s += arr1[i];
    }
    for (int i = 0 ; i < ARRAY_LEN ; i++) {
        s += arr2[i];
    }
}
```

We used the following settings for memories (DRAM and Caches).

DRAM:

- Access read: 80 cycles
- Access write: 80 cycles

L1 Program/Data Caches:

- Number of sets: 32

- Words per block: 4
- Degree of associativity: 4
- Access latency: 1 cycle

L2 Unified Cache:

- Number of sets: 64
- Words per block: 8
- Degree of associativity: 8
- Access latency: 15 cycles

Table 6.3: Cycles (Cache)

Cycles (Cache)	
Memory Configuration	Cycles
DRAM	1.286.834
DRAM + L1 Program Cache	267.930
DRAM + L1 Program/Data Cache	264.074
DRAM + L1 Program Cache + L2 Cache	265.430
DRAM + L1 Program/Data Cache + L2 Cache	222.178

On table 6.5 we can see the cycles for each memory configuration. On table 6.4 we see the speedup table for the base configuration (DRAM only) vs DRAM with caches various enabled.

Table 6.4: Speedups per memory configuration

Speedups (Cache)	
Memory Configuration	Speedup
DRAM/(DRAM + L1 Program Cache)	4.8
DRAM/(DRAM + L1 Program/Data Cache)	4.9
DRAM/(DRAM + L1 Program Cache + L2 Cache)	4.8
DRAM/(DRAM + L1 Program/Data Cache + L2 Cache)	5.7

As we can observe, adding another level of cache offers a significant difference in speedup, even for this small program.

Moving on, we showcase a sample program execution (a program that sorts an array using BubbleSort) to compare the cycles of a simple machine without pipeline and caches versus a fully pipelined machine with all levels of cache and branch predictor available.

As we can see, the speedup is around 3300%!.

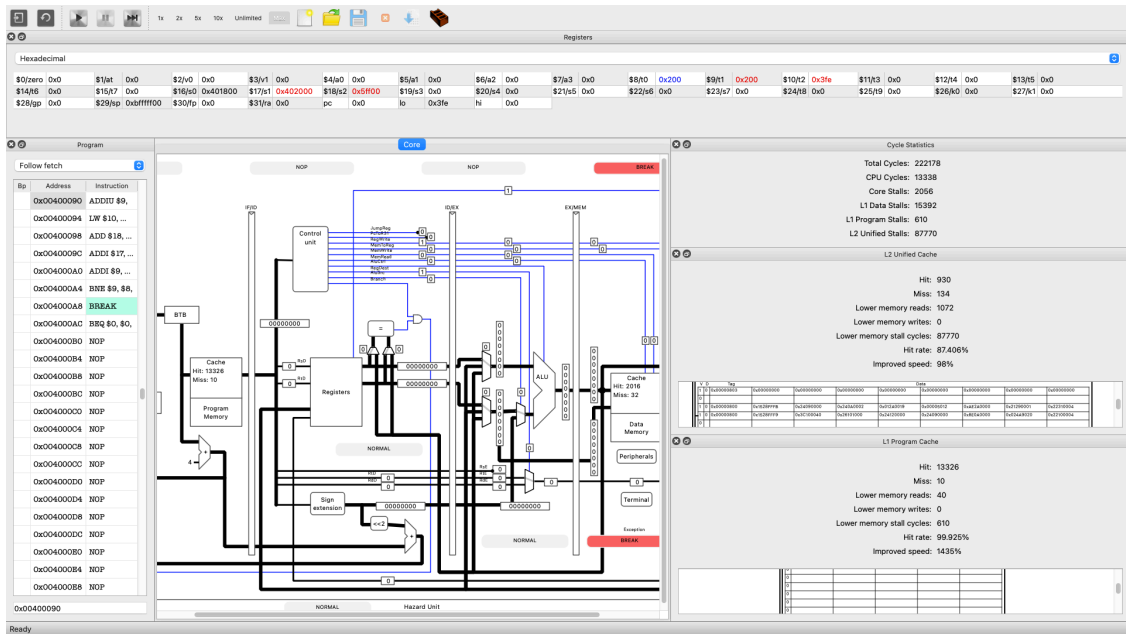


Figure 6.2: QtMips Program Execution (Cache)

Table 6.5: Cycles (Cache)

Cycles (Cache)	
Machine Configuration	Cycles
Simple Machine	1.537.276
Full Pipeline + Cache + Branch Predictor	44.789

7. QTMIPS MANUAL

On this chapter we will present a short manual for the new version of QtMips.

As mentioned previously, users can use the embedded code editor / compiler to develop and build their programs.

QtMips also offers another alternative. Users can directly load a compiled (binary) MIPS program in elf format as displayed on Figure 7.1. A MIPS program can be compiled to an elf binary using the mips-elf-gcc toolchain.

```
mips-elf-gcc -ggdb -c foo.s -o foo.o
mips-elf-gcc -ggdb -nostdlib -nodefaultlibs -nostartfiles foo.o -o foo
```

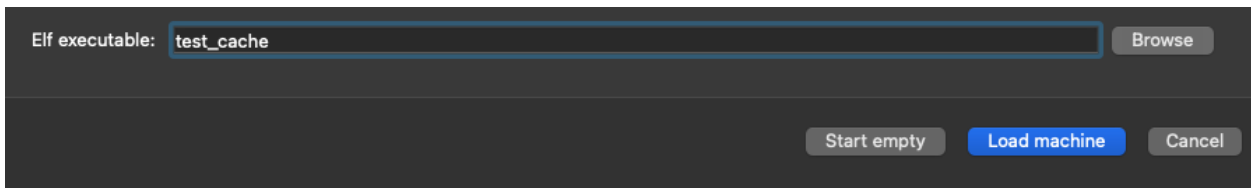


Figure 7.1: QtMips Program Loader

After specifying a program to load, users are asked to configure the simulated machine. As mentioned before, QtMips is a very versatile tool. It allows for various CPU configurations and various cache configurations.

QtMips has come preset configurations that the user can pick, as displayed on Figure 7.2

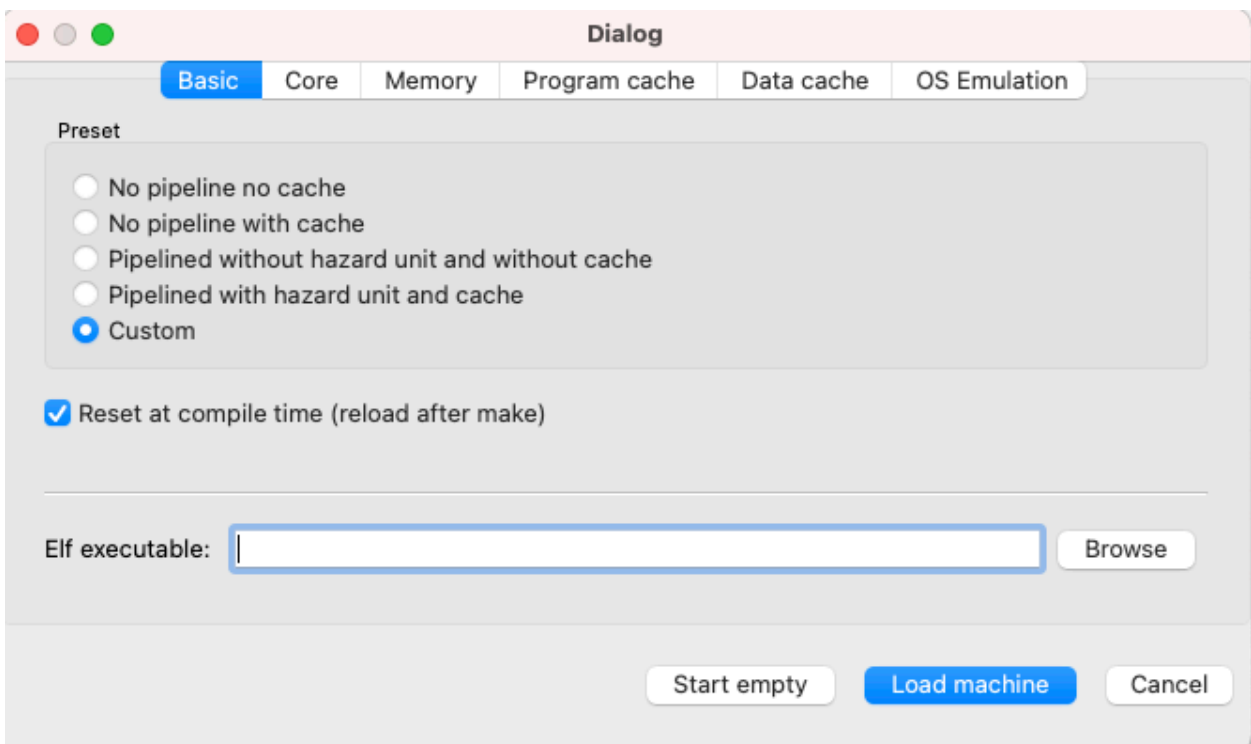


Figure 7.2: QtMips Presets

Obviously, user is free to use custom configurations. For CPU core user can choose between:

- Pipelined CPU or single-cycle CPU.
- Whether or not delay slot is enabled or not.
- Whether or not branch predictor is enabled or not.
- Branch predictor configuration (bits used for BHT/BTB and type of predictor).
- Branch resolution stage.
- Whether hazard unit enabled or not.
- If hazard unit is enabled, user can choose to stall when a hazard is encountered or use forwarding. This allows users to observe the performance that forwarding provides.

QtMips is smart enough to disable configurations that do not make sense. For example, the user is not allowed to disable delay slot in pipelined mode.

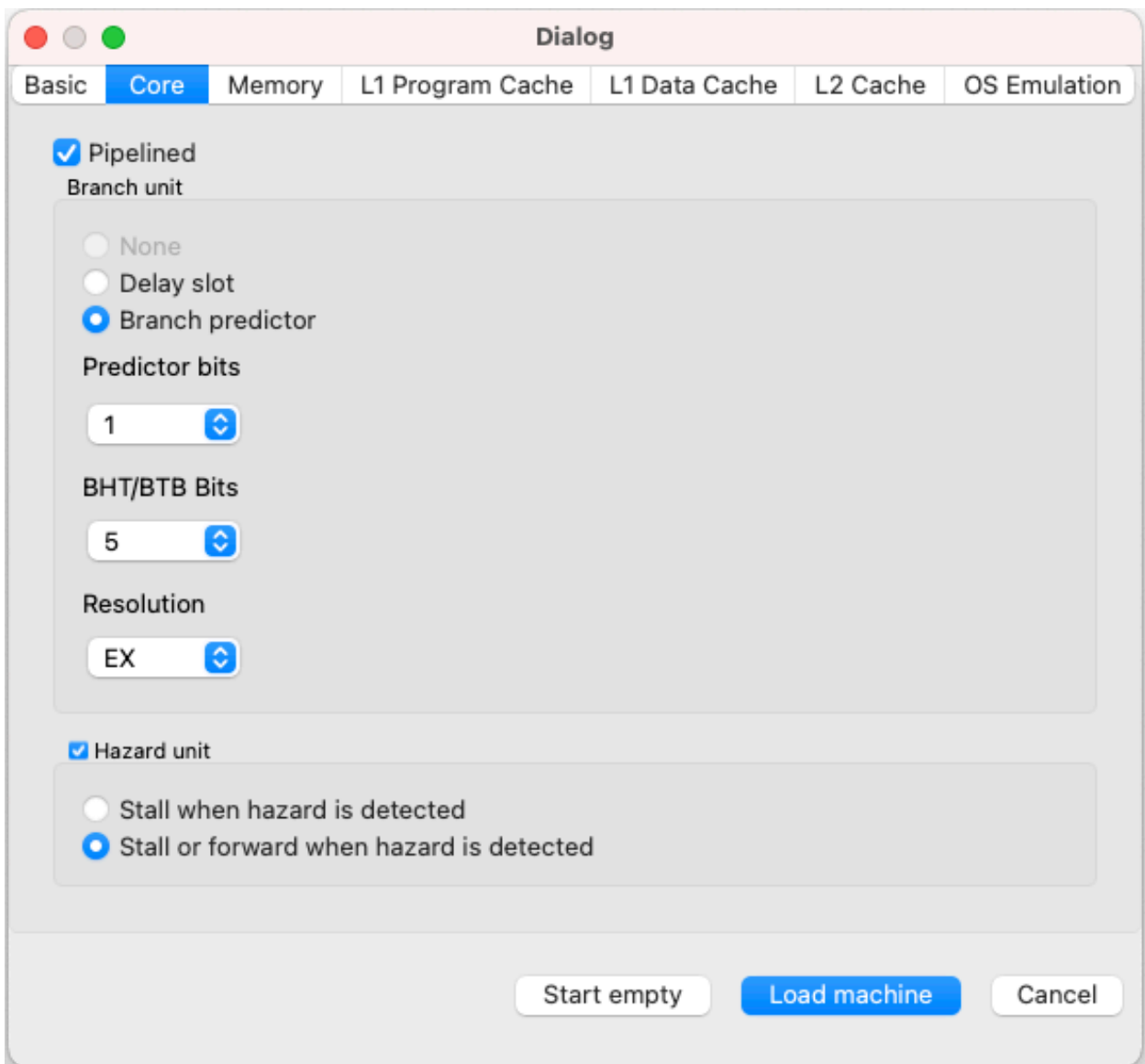


Figure 7.3: QtMips Core Configuration

Regarding Memory, user can set their desired access times for read access, write access and burst. This versatility allows user to experiment even more with memory access penalties.

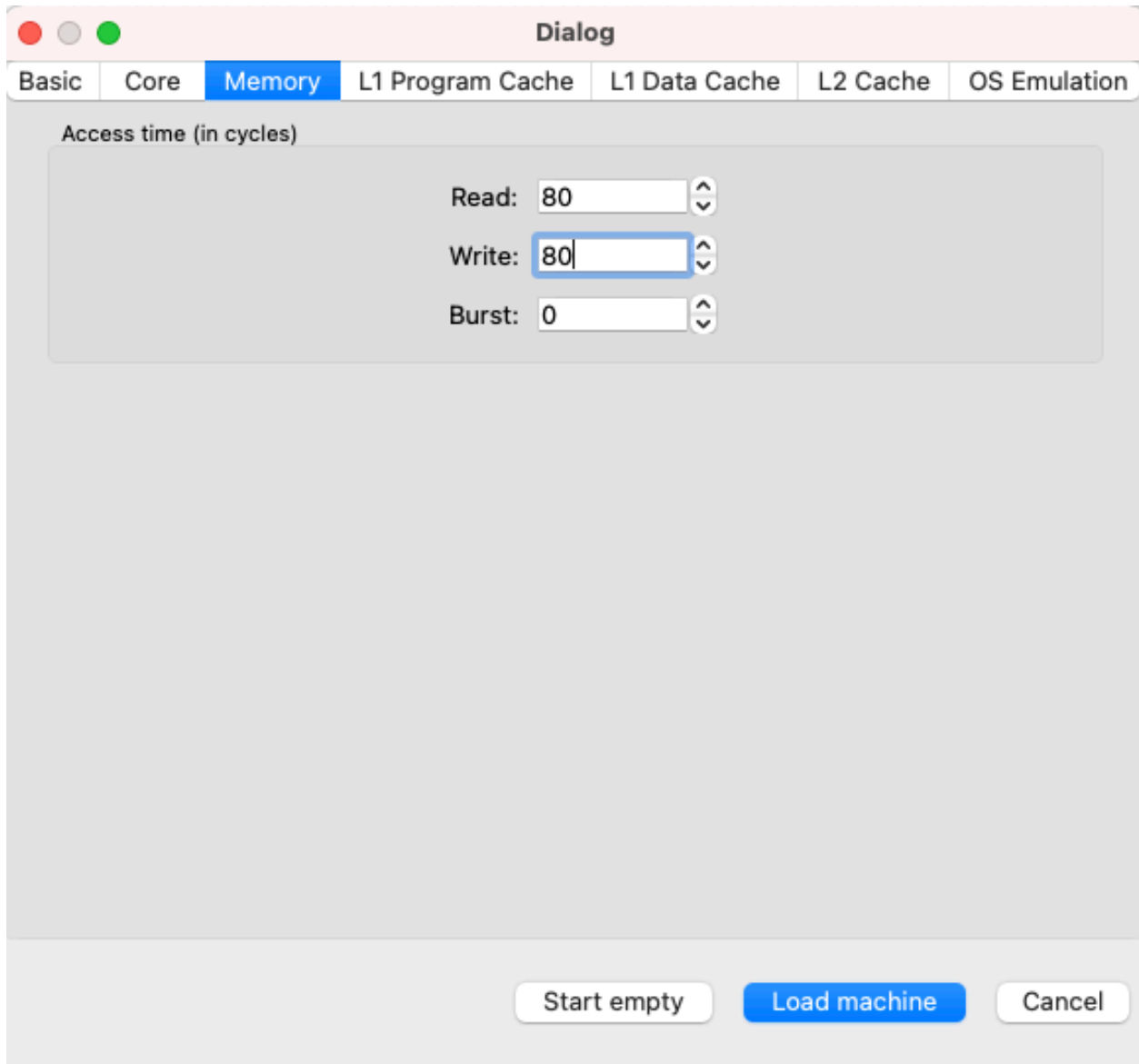


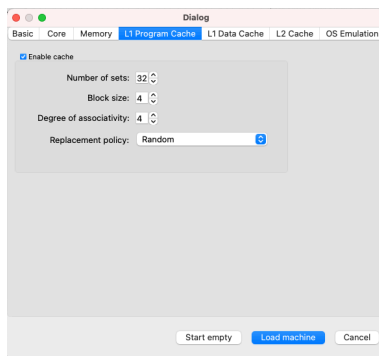
Figure 7.4: QtMips Memory Configuration

Users can pick the desired configuration regarding cache memories (Figure 7.5). They can choose between:

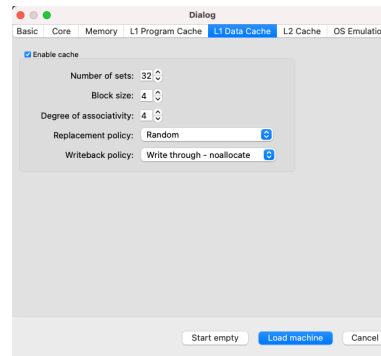
- L1 Program Cache.
- L1 Data Cache.
- L1 Program Cache + L1 Data Cache.
- L1 Program Cache + L2 Cache.
- L1 Data Cache + L2 Cache.
- L1 Program Cache + L1 Data Cache + L2 Cache.

Users can also configure cache parameters individually. More specifically:

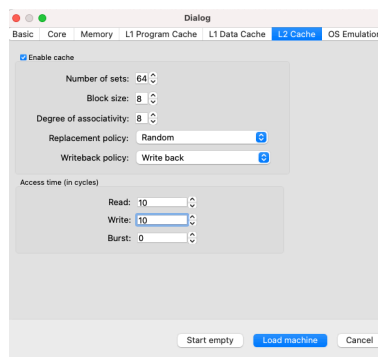
- The number of sets.
- The block size.
- The degree of associativity.
- The replacement policy.
- The writeback policy (for data & L2 caches only).



(a) L1 Program Cache Configuration



(b) L1 Data Cache Configuration



(c) L2 Cache Configuration

Figure 7.5: Cache Configurations

After specifying the desired configurations and a program to load, users can start executing their program. QtMips, as mentioned before, allows for step by step execution which helps users to better understand and debug their program.

8. EXTENSIONS ON QTMIPS AND SWITCH TO QTRVSIM

The science of computer architecture is moving from MIPS to RISC-V. Since we want the courses of the university to stay as updated as possible to what is happening in the real world, our initial plan with Prof. Gizopoulos was to completely change QtMips to QtRISC-V, which is an updated version of QtMips which uses RISC-V instruction set. However, this task would take a considerable amount of time for one person to develop and debug. Luckily, after getting in contact with the developers of QtMips, we were told that they also planned on changing to QtMips to QtRISC-V but they call it QtRVSim (for reasons that we do not understand and we do not like this new acronym), so we decided team up with them and help them extend QtRVSim.

QtRVSim does not have a delay slot and branches are calculated by ALU in EX and performed in MEM. Developers have added a trivial branch predictor which always predicts taken. As we discussed previously, this approach brings a lot of bottleneck when branches are not taken but now the penalty is 3 bubbles instead of 1 or 2, since branches are performed in MEM. Adding a branch predictor would significantly increase the performance of the programs in QtRVSim and would allow students to understand in more depth how valuable branch predictors are.

Therefore, we decided to add the following predictors for the RISC-V version.

- *One-Bit Branch Predictor*: The same as the one described previously.
- *Two-Bit Branch Predictor*: The same as the one described previously.
- *RISC-V Default*: This is the predictor which RISC-V uses by default. It is also called *Back Taken* because of the way it works. This predictor predicts TAKEN if the branch jump address jumps backwards to the program ($PC < \text{Branch/Jump Dest}$).
- *Forward Taken Predictor*: This predictor works the opposite way of RISC-V Default, it predicts TAKEN if the branch / jump destination is after PC.
- *Not Taken Predictor*: A trivial predictor that always predicts NOT TAKEN.
- *Taken Predictor*: Also a trivial predictor that always predicts TAKEN.

9. FUTURE WORK

As mentioned previously, QtRVSim is under development. We believe that we can build, along with the team from the Czech University, an excellent modern tool that will have the potential to be used both for introductory and advanced Computer Architecture courses in many universities around the world.

More specifically, our plans include:

- Adding an FPU which will be able to handle floating point instructions. This is something that is also missing from QtMips and we believe its a very important feature that could heavily assist in teaching because it would allow students to experiment more with RISC-V code.
- Adding more levels of cache. For the same reasons that we did on QtMips.
- Adding support for I/O. This feature is also missing from both versions and we believe that by adding this feature we can move one step further. For example, assigning more complex projects during the semester.
- Add a visualisation of the encoding of RISC-V instructions.

ABBREVIATIONS - ACRONYMS

MIPS	Microprocessor without Interlocked Pipelined Stages
RISC	Reduced Instruction Set Computer
FPU	Floating Point Unit
ALU	Arithmetic Logical Unit
DRAM	Dynamic Random Access Memory
SRAM	Static Random Access Memory
CPU	Central Processing Unit
I/O	Input/Output

BIBLIOGRAPHY

- [1] David A. Patterson, John L. Hennessy (2008) *Computer Organization and Design The Hardware Software Interface* Morgan Kaufmann, 2nd ed.
- [2] David A. Patterson, John L. Hennessy (2017) *Computer Architecture: A Quantitative Approach* Morgan Kaufmann, 6th ed.
- [3] Karel Koci (2018) *Graphical CPU Simulator with Cache Visualization*, Master Thesis - Czech Technical University In Prague.
- [4] Wikipedia: *MIPS Architecture*.
- [5] Wikipedia: *Branch Predictor*.
- [6] Wikipedia: *Cache (computing)*