CZECH TECHNICAL UNIVERSITY IN PRAGUE

F3

Faculty of Electrical Engineering

Master's Thesis

Graphical CPU Simulator with Cache Visualization

Karel Kočí

May 2018

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kočí** Jméno: **Karel** Osobní číslo: **406446**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra řídicí techniky**

Studijní program: **Kybernetika a robotika**

Studijní obor: **Systémy a řízení**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Grafický simulátor činnosti procesoru a činnosti vyrovnávací paměti

Název diplomové práce anglicky:

Graphical CPU Simulator with Cache Visualization

Pokyny pro vypracování:

1. Seznamte se s řetězenou a jednocyklovou verzí modelu procesoru použitého pro výuku předmětu architektury počítačů.
2. Vyhledejte existující simulátory procesoru určené pro výukové účely a proveďte analýzu jejich využitelnosti pro výuku.
3. Navrhněte simulátor architektury MIPS s grafickou reprezentací procesoru a plnění vyrovnávacích pamětí. Vlastní aplikace budou načítané z binárního formátu ELF. Procesor by mel umožňovat práci ve třech režimech: jedno-cyklovém, zřetězeném s bez a s řešením hazardních stavů.
4. Projekt zdokumentujte a publikujte ve zdrojové podobě.

Seznam doporučené literatury:

[1] Patterson, D. A., and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 4rd ed. Morgan Kaufman, 2011. ISBN: 0123747503.
[2] Brorsson, Mats. (2002). MipsIt: a simulation and development environment using animation for computer architecture education. . 10.1145/1275462.1275479.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Pavel Píša, Ph.D., katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **26.09.2017** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce: **30.01.2019**

Ing. Pavel Píša, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Michael Šebek, DrSc. podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta

/ Declaration

I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where states otherwise by reference or acknowledgment, the work presented is entirely my own.

In Prague 25.5.2018

Abstrakt / Abstract

V předmětu Architektura Počítačů vyučovaném na Českém Vysokém Učení Technickém v Praze, Fakultě Elektrotechnické se v tuto chvíli používá starý emulátor MIPS procesory MipsIt. Ten poskytuje vlastnosti důležité pro nahlédnutí do vnitřního fungování procesoru. I přesto, že byl emulátor distribuován se světově uznávanou učebnicí a dané téma tak aplikace samotná je již zastaralá. Podporuje pouze Windows a nejsou k ní zdrojové kódy. Cílem této práce je vytvořit vlastnostmi srovnatelnou náhradu založenou na aktuálních softwarových technologiích.

An old MipsIt emulator of MIPS processor is used as educational processor model in Computer Architectures course taught on Czech Technical University in Prague, Faculty of Electrical Engineering currently. It provides features important for student insight into the inner working of a processor. Even that the emulator has been distributed with world recognized textbook on the topic, the application is already archaic, Windows only and without source code. A goal of this thesis is to provide features equivalent replacement based on current software technologies.

Contents /

1 Introduction 1
2 MIPS Instruction Set Architecture 2
2.1 Data Formats 2
2.2 Registers 3
2.3 Instruction Formats and Instructions Description 3
2.3.1 Arithmetic Instructions 4
2.3.2 Logical Instructions 5
2.3.3 Shift Instructions 5
2.3.4 Branch and Jump Instructions 5
2.3.5 Load and Store Instructions 6
2.3.6 Move Instructions 6
2.3.7 Pseudo Instructions 6
2.4 Pipeline Architecture 7
2.5 Load/Store Architecture 9
2.5.1 Cache 9
3 Features Required for Education 12
3.1 Tutorial Illustrating Basic CPU Structure 12
3.2 Memory Access and cache Usage Tutorial 13
3.3 Pipelines and Hazards Tutorial 14
3.4 Memory Mapped I/O Tutorial 14
4 Existing MIPS simulators 15
4.1 MipsIt 15
4.2 QtSpim 18
4.3 Mars 19
4.4 WeMips 21
4.5 MIPS Simulator (mipssimulator) 21
4.6 Qemu 22
4.7 Hardware Description Based simulation 22
5 MIPS Simulator Design 24
5.1 Simulator Configuration 24
5.2 Registers Simulation 25
5.3 Memory Simulation 26
5.3.1 Cache Implementation 27
5.4 Instruction Decoding 27
5.5 Instructions Execution 28
5.5.1 Hazard Unit 28
5.6 Program Loading 29
5.7 Decoding to Instruction Mnemonic 29
6 MIPS simulator Graphics Design 30
6.1 Simulator Configuration Dialog 30
6.2 Main Simulator Window 32
6.3 CPU Scheme View 33
6.3.1 Scheme Blocks 36
6.3.2 Signals and Buses 36
6.4 Registers View 37
6.5 Program and Data Memory View 38
6.6 Program and Data Cache View 39
7 Conclusion 42
References 43
A Glossary 45
B Content of attached CD 46

Tables /

2.1. MIPS instructions binary format 4
2.2. Hazard resolve map 9

Chapter 1
Introduction

Computers are dominating a lot of industry sectors, including engineering as they are essential production tools. At least basic knowledge of the inner working of a processor and ability to predict its influence on performance, security and safety consequences is important for each programmer expert, computer, processor and embedded systems designer and advanced user. That is why the Computer Architectures course is included in electronics, informatics and robotic specializations at Faculty of Electrical Engineering of Czech Technical University same as at all similar world recognized technical universities. MIPS[1] architecture is selected as basic demonstration model because its first implementation can be easily understood and implemented as connection of basic blocks.

MIPS simulator MipsIt is used for practical parts of Computer Architectures course. Unfortunately this simulator is getting old and its usage becomes problematic. Crashes are common and some functionality such as simulation restart randomly breaks and requires program restart. Updating currently used program is not possible because source code is not freely available. Because of that the goal of this thesis is to implement replacement simulator.

Simulator implemented as part of this thesis is implemented in C++ programing language using Qt toolkit. It's called QtMips. It is supposed to present graphical user interface (GUI) with visualization of microprocessor interworking. Both pipelined and single-cycle CPU are implemented implemented and visualized.

Simulator is also expected to implement cache subsystem with variable size.

Programs to simulator should be loaded in Executable and Linkable file format (ELF). This format is considered as a standard for executables, because it is default output from compilers such as GNU C Compiler (GCC).

The thesis starts with an analysis of MIPS processor design, instruction set, and possible implementations. That is followed by section looking into features required for education. Next section evaluates possibility to use existing simulators instead of MipsIt or as a base for new implementation. It is followed by two chapters describing simulator technical and graphical design.

[1] MIPS is acronym originating from initial goal to design microprocessor without interlocked pipeline stages.

Chapter 2
MIPS Instruction Set Architecture

MIPS is a CPU instruction set architecture (ISA) with a long history of development and usage. It is is one of the first processors (CPU) architecture designs focussed on instruction set complexity reduction (RISC ISA) to achieve higher instruction procession throughput per cycle (IPC) at higher clock frequencies. The architecture has undergone many enhancements and changes from its original design which corresponds to instruction set changes. The original instruction set is as well referenced as MIPS I to distinguish it from later versions now. The first CPU design implementing this architecture is MIPS Computer Systems' R2000 processor which is why original instruction set is also referenced as MIPS R2000. A goal of this thesis is to implement a visual tool which allows learning basic CPU concept. That is which the only subset of R2000 ISA is implemented and described.

Only information and concepts relevant to the selected architecture subset are described in details in the Section 2.1 and later. Everything described in this chapter is implemented in simulator unless stated otherwise.

MIPS ISA is designed as a RISC. It contains set of instructions that are supposed to be easy to implement and fast to execute. It is common that all instructions take single CPU cycle to execute (at least if we ignore pipelining and memory access). In case of MIPS ISA all instructions also have the same fixed size (32 bits).

Architecture of MIPS is divided to core instructions and four extensions. These extensions are called coprocessors. Following coprocessors are defined with their usage[1]:

- *Coprocessor 0 (CP0)*: Virtual memory system, exceptions handling and CPU states including switching between kernel, supervisor and user states.
- *Coprocessor 1 (CP1)*: Floating point unit
- *Coprocessor 2 (CP2)*: Free for platform specific usage (extensions added by chip manufacturer)
- *Coprocessor 3 (CP3)*: Reserved for MIPS ISA extension

These coprocessors are not analyzed in this thesis because they are out of the education scope and because of that they are not implemented in simulator.

2.1 Data Formats

MIPS I specifies and works with three data sizes[1]:

- *Byte*: 8 bits
- *Halfword*: 16 bits or two bytes
- *Word*: 32 bits or two halfwords or four bytes

Bytes in halfword and word type can be configured in either big-endian or little-endian order[1]. In case of big-endian first byte is the most significant one. For little-endian is first byte the least significant one. Simulator developed as part of this thesis implements big-endian operation and instruction set variant. That is default mode in which is CPU initialized and because of not implemented coprocessor 0 there is no possibility to switch it to little-endian.

In some operations in CPU, such as comparing of values, there is a requirement to recognize if data is signed or not. This is explicitly given by instruction specifying given operation. It is also important to note that two's complement signed number representation is used[2].

Some operations such as data load from memory or data store to memory require size type change. Change is either to make type data type smaller, which is done by dropping more significant bytes, or to make bigger data type. In such case there are two approaches. One is for unsigned values. Those are extended by zero bytes. The other one is for signed values. Sign extension is required. That is operation after which initially negative value stays negative even if additional bytes are prepended[3].

MIPS ISA also specifies data types for floating point numbers but those are not relevant to this thesis as FPU is not implemented.

2.2 Registers

MIPS instruction format addresses thirty-two 32-bit general purpose registers where some of them has special use.

Register 0 is hard wired to value zero. Result of any instruction writing to it is discarded and any read is read as zero value.

Register 31 is used as destination register by jump/branch and link instructions (see Section 2.3.4). Its usage in these instructions is explicit and not specified in instruction it self. These instructions are not implemented yet. Therefore this register is in reality like rest of the 29 registers.

On top of the 32 general purpose registers there are other special purpose registers. All with 32-bit size.

The primary one is program counter register. This register is used to store address to executing instruction. It cannot be directly modified (at least not without using coprocessor) but it is instead incremented automatically and modified through jump and branch instructions.

There are two other registers used during multiplication and division. Those are HI and LO. Where HI is called as higher result and LO as lower result. These registers are used in following described situations:

- Both HI and LO are used to store product of integer multiply in case of multiplication. HI contains higher word and LO contains lower word.
- LO stores quotient and HI stores remainder of integer divide, in case of division.

Separate instructions to move to values from and to HI and LO registers are provided by MIPS ISA. Exact instructions using LO and HI are not discussed later in this text as they are not strictly required for the purpose of this thesis.

2.3 Instruction Formats and Instructions Description

MIPS ISA specifies three instruction formats. They are identified as R, I and J. Their binary format is as can be seen visualized in table 2.1.

Type	Format (bits)					
R	opcode (6)	rs (5)	rt (5)	rd (5)	sa (5)	function (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Table 2.1. MIPS instructions binary formats.

An *opcode* field in instruction are top most 6 bits containing operation code. This operation code is for initial identification of instruction. Some of the R instructions share the same operation code and in such case *function* bits are used to fully decode type of instruction.

rs, rt and *rd* are 5 bit wide identifiers of used general purpose register. *rs* is a shortcut for source register, *rt* is a shortcut for target register and *rd* is a shortcut for destination register. *rs* is always, as the name suggests, a data source. It means that it is always read. *rd* is always written to but only R instructions have this field. In case of I instruction type *rt* is used instead instead as destination register except for memory store operations where it specifies the second source register. In case of R instructions *rt* is read.

sa is used for shift instructions. This 5 bits wide field contains number of bits to be shifted.

function is 6 bits wide identifier that together with *opcode* defines instruction type.

immediate is 16 bits wide field containing constant to be used in operation specified by given instruction.

address is 26 bits wide field with low address bits for jump instruction. J format is used only for jump instruction and provides a way for long jumps without need of jumping by register value.

In following subsections some of the MIPS instructions[4] are described. They are grouped together by type.

2.3.1 Arithmetic Instructions

Arithmetic instructions implement arithmetic operations applied on values from *rs* and *rt* registers and writing result to *rd* register, at least in case of R instruction type. Some of them are I instruction type and therefore use *immediate* value instead of *rt* as source. Not all arithmetic instructions are implemented in simulator because they are not essential for education. Because of that only few selected ones are noted and described here.

Instruction `ADD` and its variants `ADDU`, `ADDI` and `ADDUI` serve for adding two numbers. Given instruction does unsigned addition in case of appended `U` after base `ADD`. Arithmetic exception is raised when a result of the addition of two signed operands overflows 32-bit second complement representation range. Doing the same with unsigned addition doesn't cause exception and instead it results to overflow (33-th bit is dropped). Instruction is using format I instead of R when `I` is appended. Second operant used for adding is not from general purpose register but it an immediate value stored in the instruction itself in such case.

Instruction `SUB` and its kin `SUBU` serve for subtracting numbers. Appended `U` serves same purpose as in case of `ADD` that means it is saying that it is unsigned subtraction. And unsigned subtraction cannot cause exception unlike signed one.

Last instructions to be noted here are `SLT` and its variants `SLTU`, `SLTI`, `SLTIU`. Those compare value from register *rs* with value from register *rt* and boolean result (either value 1 or 0) is stored to *rd*. In case of appended `I` to `SLT` it is same as for `ADD`. Meaning that second operand is replaced by value from instruction it self and *rt* is used as output instead. In case of appended `U` instruction does unsigned comparison. Difference between unsigned and signed comparison is in what ever values with most significant bit set are considered as bigger than the ones with cleared one or not. In other words if value should be considered to be unsigned or not.

2.3.2 Logical Instructions

Logical instruction implement boolean operations[5]. Except for different type of operation they are same as arithmetic instructions. They are also mostly of R instruction type. And they too use *rs* and *rt* registers as source of values operation is applied on. And result is too written to *rd* register. Exception for I instruction types applies here as well, where the immediate field from instruction itself is used as second operant.

All logical instructions apply some boolean operation on bit by bit basis. N-th bit from source values are combined according to operation and result is placed again to N-th bit. That is done for all 32 bits. In case of `AND` instruction conjunction is used. For `OR` instruction disjunction is used. There are also `NOR` and `XOR` instructions. Those implement negated disjunction and exclusive disjunction respectively.

There are also immediate versions of some of these instructions. As noted already those are of I type and the one of the inputs is used as value from immediate field of instruction instead of *rt* register. In contrast to arithmetic instructions *immediate* field value is zero extended instead of sign extended. These immediate instruction variants are `ANDI` as analogue for `AND`, `ORI` as analogue for `OR` and `XORI` as analogue for `XOR`.

2.3.3 Shift Instructions

Shift instructions serve for logical and arithmetic shifts[6].

Half of shift instructions take value from general purpose register *rt*, apply shift by *sa* value and writes result to *rd* general purpose register. Instructions `SLL` and `SRL` do left or right logical shift respectively. Instruction `SRA` does arithmetic right shift.

Instructions `SLL`, `SRL` and `SRA` has also variants `SLLV`, `SRLV` and `SRAV` for shifts by value from general purpose register *rs*. They don't use *sa* value from instruction.

2.3.4 Branch and Jump Instructions

Branch and jump instructions manipulate program counter. Jump instructions do it unconditionally while branch instructions compare selected register values.

There are two primary jump instructions: `J` and `JR`. `J` instruction is of J instruction type and does absolute jump in current memory section. This section is specified by upper four most-significant bits in current program counter value. `JR` is a R instruction type and also is an absolute jump but compared to `J` it can jump anywhere in memory (using complete 32 bit addressable space). Target address is given by value in general purpose register *rs*.

There are following branch instructions: `BEQ`, `BNE`, `BLTZ`, `BGTZ`, `BLEZ` and `BGEZ`. They all are of I instruction type. `BEQ` and `BNE` instructions compare two registers (*rs* and *rt*) and if they are equal or not respectively then sign extended *immediate* value is added to current value of program counter. Instructions `BLTZ`, `BGTZ`, `BLEZ`, `BGEZ` compare only single general purpose register (*rs*) against zero. Otherwise if condition is met then same as in case of previous branch instructions sign extended value of *immediate* is added to program counter value. For `BLTZ` is condition if value is less then zero. In case of BGTZ is condition what ever is value greater then zero. For BLEZ and BGEZ conditions are if value is less or greater respectively or equal to zero.

MIPS ISA also specifies jump/branch and link instructions. Those use general purpose register 31 for storing original address before changing program counter.

2.3.5 Load and Store Instructions

Load and store instructions are two types of instructions for receiving and storing data from and to memory. Those are instructions of I type. Register *rs* is used as a source of address. And register *rt* is used either as source for value to be written in case of store instruction or value loaded from memory is written to it in case of load instruction. Immediate field in instruction is used as an offset to address, it is added to value from *rs* register.

To receive data from memory a load instruction like LB, LH or LW exist. Second letter in these instruction names correlates with data type to be loaded. Meaning LB loads byte, LH loads half word and LW loads whole word. When type that is less than word is loaded then it is sign extended to whole word (more about that can be found in Section 2.1). There are also derivate instructions LBU and LHU where U stands for unsigned. Those zero extend value instead of sign extend. Instead missing bites are filled with zeroes.

To store data to memory a store instructions like SB, SH and SW exist. Second letter in these instruction names correlates with data type same as in case of load instruction. But there are no unsigned variants of store instructions as they are not required. When word is stored as byte then only least-significant byte is stored.

2.3.6 Move Instructions

Move operations implement various transfer operations between various registers. There are two types of such instruction that are implemented in the simulator. These are instructions moving value from/to LO and HI registers to/from general purpose ones. Second type are conditional moves.

Instructions MFHI and MFLO are moving value from HI and LO register respectively to general purpose specified in *rd*. Instructions MTHI and MTLO do opposite move. They move value from general purpose register specified in *rs* to HI or LO register respectively.

Then there are MOVZ and MOVN instructions. Those conditionally move value from general purpose register *rs* to general purpose register *rd*. Condition is either if value in register *rt* is zero for MOVZ or non-zero for MOVN. If condition is not met then move is not realized.

Part of this category is potentially also instruction LUI. In MIPS manual it is placed together with logic instructions but that is mostly because of its possible implementation. Its usage is for setting constants to register's more-significant half word. 16 bits of instruction immediate field are shifted left by 16 and lower 16 bits from value from *rt* register are concatenated. Result is stored in general purpose register *rt*.

2.3.7 Pseudo Instructions

Not all instructions have to be implemented explicitly in hardware. Some of them are defined in MIPS assembler. Compiler recognizes them but they are just special cases of other instructions. Or they stand for combination of other instructions[7].

There is NOP instructions that have no effect on CPU state when executed. It's also called as no operation. It is R instruction type and is an idiom for SLL instruction with all fields set to zero. In the other words its binary representation is all bits set to zero.

Another pseudo instruction is `MOVE`. This one is for copying value from one general purpose register to another. It is implemented using `ADD` instruction by adding register zero with source register.

Also there is a `BLT` pseudo instruction called branch if less then. It completes set of compare and jump instructions. It is implemented using `SLT` and `BNE` instructions.

One additional branch pseudo instruction is `B`. It's unconditional branch and it is implemented using `BEQ` instruction by comparing register zero (it is same register and because of that values are always equal).

The last pseudo instruction is `LA` called load address. It is intended for loading constant address (whole word) to general purpose register. This has to be done by two instructions and in common it is implemented with combination of `LUI` and `ORI` instructions.

2.4 Pipeline Architecture

MIPS ISA was designed with goal to achieve pipelined execution. In general it is possible to divide instruction execution to almost any arbitrary number of discrete operations. In case of basic MIPS architecture implementation, division into five stages of the pipeline is used. These stages are called[8]:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execution (EX)
- Memory (MEM)
- Write Back (WB)

In instruction fetch stage the instruction is loaded from program memory from program counter's address. This automatically increments program counter by 4 unless previous instruction was jump or successful branch.

Instruction decode stage contains instruction decoder, registers file and compare logic used for branch instructions. In this stage the instruction is used for generating signals for this and all following stages. That is instruction decoder's job. Register identifiers *rs* and *rt* are used for getting values from given registers and *immediate* instruction field is sign extended to 32 bits.

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. In case of I instruction type second value, that would be loaded from *rt* register, is replaced with sign or zero extended value from *immediate* instruction field from decode stage. What ALU operation is used is defined by signal passed from control unit from decode stage.

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. For more in depth information on memory access please see section 2.5. Both output from memory and ALU output are passed to next stage.

Last stage is write back. In this stage is either output from ALU or from memory written to *rt* or *rd* register (depending on instruction type).

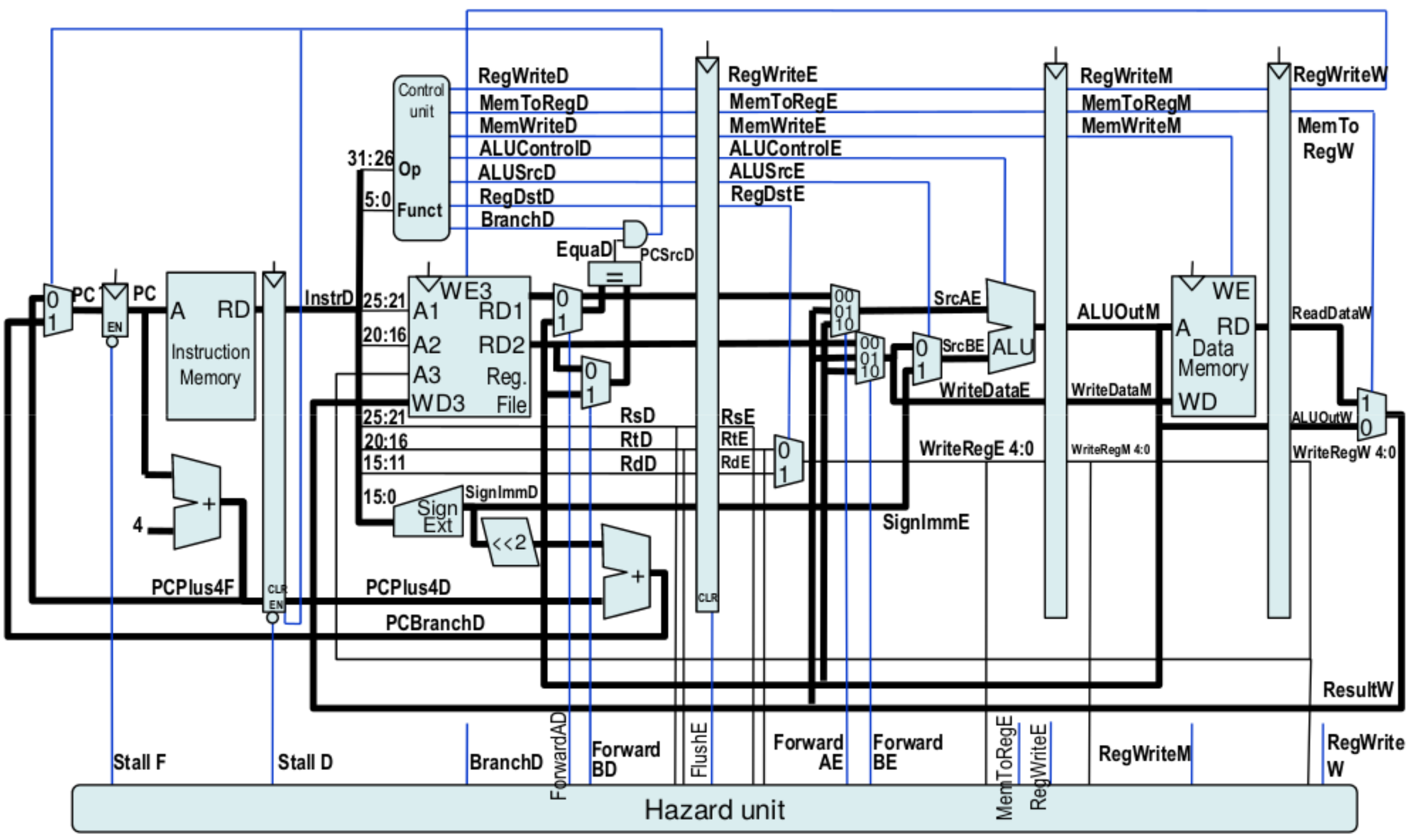


Figure 2.1. Scheme with pipelined CPU as is taught in course Computer Architectures (Source: presentation number four from this course[9]).

With instruction execution divided to stages it is possible to implement pipelined processor. This is architecture where subsequent instructions are evaluated in parallel in different sections of CPU. This means that single instruction needs five CPU cycles to complete but one instruction is always completed every cycle except for special cases. This allows faster CPU clock speed because every single stage needs less time to process instruction than all stages combined. This division of instruction processing logic enables to execute up to five times more instructions in the same interval as if the whole instruction is executed in one cycle. But such speedup would require ideally balanced pipeline design.

Pipelined architecture introduces some obstacles. Specially, for branch instructions we can’t decide what instruction should be loaded next in instruction fetch stage alone. Instruction itself has to be decoded and that makes fetch stage too complicated and longer to resolve. This problem is called control hazard or branch hazard. Simple solution is to insert NOP instructions to pipeline until we decide what next program counter value should be. But this decrements pipelined speed gain. MIPS architecture instead solves this problem simply by accepting that following instruction after any jump or branch instruction will be always executed. Such instruction is in so called delay slot.

Another obstacle in pipelined architecture are data hazards. Impact of data hazards has to be analyzed and prevented for pipelined architecture. It is a problem where instruction requires an input output from some previously executed instruction. Results from two previous instructions cannot be read from registers as they are still in pipeline. Because registers are read in decode stage and previous problematic instructions can be in that case in execute and memory stage. There is also another possible instruction in write back stage. But because write back usually consist only of single multiplexer there is enough time to propagate new value to registers before they are read[8].

The possible data hazards can be accounted and resolved in program compilation phase for a simple pipeline. But such solution prevents future microarchitectural changes and leads to more wasted cycles (bubbles) when pad (NOP) instructions are inserted. It makes compilation and manual program writing much more complex as well. Another solution is to include hazard unit in a CPU design which can resolve hazards by stalling pipeline or forwarding results between instructions.

Table 2.2 contains all possible situation when hazards can occur in MIPS I with 5-stages pipeline design. There are two types of instructions in sense of possible hazards. Those of which we know their result in memory stage. Those are exclusively all load instructions. And those of which we know their result in execution stage. Those are all instructions modifying general purpose registers and at the same time are not load memory instructions. For all instructions we need their inputs at the end of instruction decode stage. In table 2.2 instruction currently being decoded (being in instruction decode stage) requires content of register that is supposed to be changed by previous instruction. Columns of table are possible positions of such instruction in pipeline.

	EX	MEM	WB
Result known in execute stage	Forward	Forward	No hazard
Result known in memory stage	Stall	Forward	No hazard

Table 2.2. Hazard resolve map

2.5 Load/Store Architecture

MIPS ISA uses load/store architecture. Instructions to access memory are strictly separated from rest of ISA. Such memory instructions are designed to be specialized and not to have any side effect except of memory manipulation. There are two types of such instructions. There are load instructions for receiving values from memory to registers. And there are store instructions for storing values to memory.

Such architecture simplifies implementation and compiler optimization [1]. It also limits possible additional data hazards (see Section 2.4). If the data can be loaded and manipulated or even stored (read-modify-write) by single instruction (usual case of CISC designs), then pipeline has to be prolonged, or instruction folded multiple times through pipeline which makes the design much more complex. Because of load/store architecture, where value can be either read or written but not both, its possible to completely ignore memory as a source of hazards.

MIPS ISA specifies 32 bit address size and 32 bit native data type. That means that words can be interchangeably used as both data and addresses. But it also in design limits amount of memory accessible.

2.5.1 Cache

Although load/store architecture forces by its design less memory intensive operations (data has to be held in general purpose registers and are moved to memory only if it is really necessary) it doesn't mitigate the core problem, that is slow memory in comparison with the rest of the CPU. For this purpose caches are used[10].

Cache is specialized data storage with fast access times that tries to serve as memory copy. It has limited size so it can't be complete copy but storing at least some of the data that were lately used improves overall memory access times. When memory is accessed through cache then there are two possible results. Either requested data were lately used and are still stored in cache or they were not recently accessed and are available only in memory it self. First case is called cache hit and second one is called cache miss. Commonly cache hits are resolved in just a single CPU tick. When cache miss is encountered it commonly takes considerably more time than cache hit.

Cache it self is constructed using value store paired with additional meta informations. At minimum cache has to have memory address identifier, called tag, of that specific value stored. On top of that it has to have bits signaling if it contains valid value and in some cases also dirty bit is required (will be addressed when write-back policies are considered). When there is read request then cache checks if it has value with validity bit set and tag matching address and if so then it provides given value instead of accessing memory it self.

Having only one value store makes unefficient cache. Because of that caches are constructed from multiple of such value stores. There are few ways those can be grouped together to create bigger cache.

One way is to just add more separate value stores. This describes parameter called *Degree of associativity* or number of ways through cache. When cache is accessed then it goes through all of its values and looks for valid one with tag matching with address. When there is no match then it looks for first one that is not valid and uses that one to get value from memory and storing it there. When all values are valid then it applies replacement policy and replaces one (changing value and corresponding tag). If there is no expansion of this cache (as described in following paragraphs) then it is called fully associative cache.

Another way is to just increase size of value storage. In that case we can store multiple words in a single store. Words stored on top of directly requested one are those that are on addresses right next to it. This divides memory to blocks of words that are loaded to cache together. It also shortens required size of stored tag because in that case we don't have to look for exact address match but just for address that exactly identifies memory block. Parameter specifying number of words to be used is called *Block size*.

The last way we can expand cache is by adding so called sets. Those are additional value stores with separate tag and other needed bits. They makes stored tag shorter same way as having higher Block size. It also archives it by almost same method. Low part of the address exactly specifies what set should be used and that way it is possible to not store those bits in tag.

When all value stores are marked as valid and tag from none of them matches the needed address (cache miss is encountered) then there is need for replacement. One of values in cache has to be replaced with value from memory from requested address. Unfortunately it is not directly defined which one should be replaced in case of higher than one degree of associativity. There are multiple algorithms to choose which one should be replaced. This thesis is concerned only with three basic ones:

- Random
- Least frequently used

- Least recently used

Cache entry to store data is chosen randomly in case of random algorithm. It is one of the simplest possible algorithms as there is no need to store any additional information.

Another two algorithms require some additional information about cache. They are based on collected access statistics. Difference is what kind of statistic is used. In case of Least frequently used it is just simple counter tracking number of accesses. In case of Least recently used it is time stamp updated every time field is accessed. For both algorithms it is desirable to replace the store with the lowers statistic. In case of Least frequently used it is the one accessed lowest number of times. In case of Least recently used the one replaced is the one with oldest access time.

It is question what should be done with value that is currently stored there in case of replacement. There are two possible approaches. One allows immediate override. Other one requires write to memory.

First approach where we can just override current value without any additional action is called `Write through`. Using this requires every cache value change to be also written to memory. In this case there is no need for dirty bit.

Second approach is where changed value has to be written to memory before it is overwritten. This is because all write requests just modify value in cache store but not in memory. This lowers memory load on writes but it requires additional logic that in case of cache replacement writes changed value back to memory.

Chapter 3
Features Required for Education

Goal of this thesis is to implement application for education. Because of that it is beneficial to first look at existing tutorials as those describe minimal requirements to replace currently used software. If those are fulfilled then migrating from previously used simulator to a new one, that is implemented as part of this thesis, should be much easier. Only change might just be need of screen-shots replacement in documentation for students.

The application is intended to be used in course on Computer architecture at CTU[9]. It is an introduction course taught as part bachelor programs. Students are introduced to problems in CPU design. Course starts with basics such as arithmetics in computers and simple CPU design. It deepens this knowledge with memory access cache, pipelining and I/O. All this is demonstrated on MIPS architecture. Course also goes through other CPU architectures but for purpose of this thesis MIPS is the important one.

MIPS is used primarily because of its simple to decode and simple to understand instruction coding. It is easier to explain concepts to newcomers when instruction are coded in simple and stable way. It might be too confusing if used instruction coding would not have stable coding. Meaning if same bits in different instructions would be used regularly to code different information. But primarily it is architecture that is already used at the moment in course and intention of this thesis is not to replace it.

Relevant tutorials are described in following sections in this chapter. Not all taught tutorials are described here and some of them have wider reach than described. Some of them are also taught as not as single tutorial but multiple ones. In general following sections just contains themes and simulator usages in those tutorials that are relevant to this thesis.

3.1 Tutorial Illustrating Basic CPU Structure

This tutorial is initial introduction to MIPS. It is taught in the third week. In initial weeks students are only introduced to some motivational examples and to computer arithmetics.

Students are primarily introduced to MIPS assembly language as for most of them it is first contact with assembly what so ever. Students are presented with following instructions:

- *ADD*
- *ADDI*
- *SUB*
- *BNE*
- *BEQ*
- *SLT*
- *SLL*
- *J*

- *LW*
- *SW*
- *LUI*
- *LA*
- *JR*
- *JAL*

Please refer to chapter about MIPS architecture (2) for description of these instructions. These instructions are minimal set that simulator has to support (except of course *LA* as that is pseudo instruction).

Students are provided with tutorial and presentation how to write simple assembly language program and how to rewrite basic C program constructs to MIPS assembly. Those are if-else statements and while and for loops. They are primarily presented to introduce assembly language to students. They are expected to already know uses of these constructs in C.

For these types of examples it is required that students can see compiled code in simulator while it is executed. Primary feedback for these code snippets is also from program counter and secondary general purpose registers. Students are expected to understand link between program counter and executed instruction. And they should be able to track and predict program flow in memory.

Also not to confuse students it is preferable not to use pipelined CPU and caching. Explaining delay slot on top of assembler is enough.

Next task is to analyze and write code operating on data memory using the load and store instructions together with load address pseudo instruction. Example like incrementing values in array is used to illustrate their usage.

To correctly visualize example of memory access it is required to present content of memory in simulator. This is not same as visualizing program loaded to memory but it can be implemented almost the same way because both program and data are in same memory. Only needed difference is to instead of doing reverse instruction decoding (decoding instruction to their assembler representation) to just display values in hexadecimal format.

For user friendliness of simulator it would be beneficial to also allow other numerical formats. For example showing values stored in memory in decimal or binary format. Because one of the goals of previous tutorial was to teach students conversions between numerical systems it is not beneficial for usage in this course. Not having easy automatic numerical conversion is a way to force students to do conversions outside of original lecture.

3.2 Memory Access and cache Usage Tutorial

This tutorial interactively presents cache usage for memory access. Students are presented with problem of slow memory access and cache is presented as a solution. It is interworking is explained and parameters defining size and behaviour of cache are presented (they are described in Section 2.5.1). For purpose of standardized parameters description following format is established: “Size/Block size/Degree of associativity”. This format is also used in following paragraphs. It fully describes size and topology of cache, at least in limits of required simulator abilities.

Students should have prepared code from previous lecture. In that lecture they are presented with Bubble sort algorithm[11] and are instructed to rewrite it to MIPS assembler. This algorithm is then used in this lecture to test various cache configurations. As a reference cache implementations are used following configurations:

- 4/1/1: This one is called directly mapped cache
- 4/1/4: This one is called fully associative
- 4/1/2: This one is called cache with limited level of associativity

Students are also instructed to test other combinations of parameters and to find out optimal cache for their algorithm.

Simulator is required to have some cache content visualization and whole cache simulation has to be configurable enough to allow wide range of settings as used in this tutorial.

3.3 Pipelines and Hazards Tutorial

This tutorial introduces pipelining and problems caused by it (those were in depth described in Section 2.4. Students learn about five execution stages and their possible parallel execution. Then they are presented with data hazards. Hazard unit is introduced and described.

Later in tutorial students are provided with MIPS assembler code that they should edit so it can run on CPU with pipeline but without hazard unit. That should give students understanding what kind of problems hazard unit exactly solves and how overcoming them by hand in program can be inefficient.

To ensure that compiler won't interfere with students code they are supposed to include directive `.set noreorder` in their code. That ensures that compiler won't be adding or moving any instructions to fill in delay slot.

To support this usage with simulator it is required to have pipeline scheme and of course also support for pipelined CPU.

Original simulator had scheme visualisation only for pipelined CPU. Because students are using implementation without pipelining in previous tutorials, a scheme visualisation for non-pipelined version has been implemented. Having them all use CPU scheme without pipelining and them presenting them with scheme with it should give them deeper understanding of presented difference.

3.4 Memory Mapped I/O Tutorial

This is last tutorial in which is MIPS simulator used. It's the one presenting memory mapped I/O. Students are presented with concept of memory mapping and with inputs and outputs from CPU. Meaning interacting with external electronics. To illustrate this currently used simulator provided eight lights and eight switches. They are mapped to single byte on address `0xBF900000`. Students should blink with these lights provided pattern. And as next step they should be able to read switches state and use it to select pattern to light up.

To implement this, simulator should have dedicated view for lights and switches. Those should reflect writes and reads to some specific memory address. Having that specific address configurable same as what kind of I/O is used would be beneficial.

Requirements for this tutorial are not fulfilled as part of this thesis. It is outside of this thesis assignment. It is noted here for completeness and as a reminder that this is missing for complete replacement of previous simulator.

Chapter 4
Existing MIPS simulators

There are multiple already existing simulators. This chapter lists existing relevant applications and why their usage is not satisfactory for both education and as code base for this thesis.

Except of already used simulator, MipsIt, all other simulators missed some required feature. Most of them are open-source. Next section discusses which features each of them lacks and why none of them has been used as a base for this work. In general for most of these simulators it falls to same reasoning. To add missing features such as for example schematic view of CPU ELF file loading or cache it would require to heavily modify existing code to allow needed features. It means complete redesign of base code of application in all cases. This would mean that it would be necessary not only to come up with new base code but also on top of that to study existing. That would almost doubled the work and gain in case of using an existing code base would be mitigated.

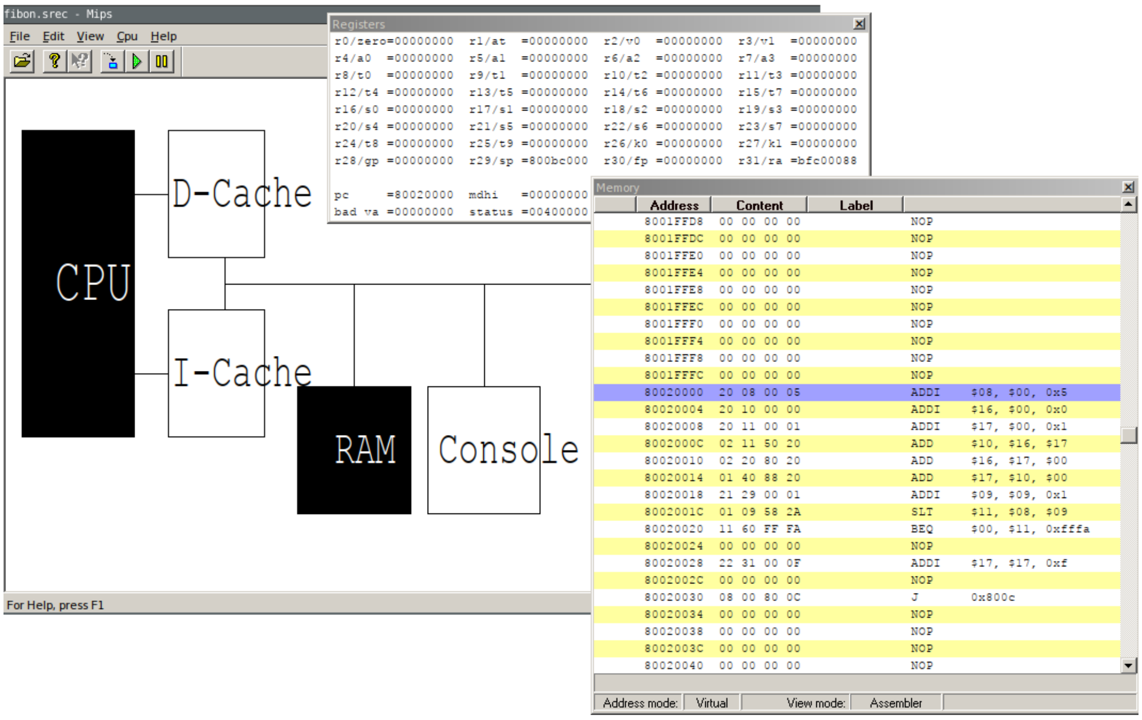
4.1 MipsIt

This is currently used simulator. It contains three simulators together with integrated development environment (IDE) [12]. Those three simulators differ in what CPU they simulate. The simplest one simulates single-cycle CPUand it is simply called *Mips*. Then there are two simulators implementing CPU with pipelining. One of them doesn't implement hazard unit and is called *MipsPipeS* and other one does and is called *MipsPipeXL*.

MipsIt was developed for Microsoft Windows around year 2000. This makes it fairly old program and it has problems to run on new versions of Windows. Primarily it has to be run with Wine[1] on Unix systems. Together with not running on native platform and probably some left over bugs and no following development it now starts to be more and more problematic. It often crashes and it has problems with simulations restarts. Fixing these problems is not easy or even possible as MipsIt is closed source.

It serves as the baseline for this thesis because this program is currently in use and this thesis plans to replace it. Not all features are required so this thesis doesn't copy it but it is heavily inspired by it.

[1] `https://www.winehq.org/`



Registers

```
r0/zero=00000000  r1/at  =00000000  r2/v0  =00000000  r3/v1  =00000000
r4/a0  =00000000  r5/a1  =00000000  r6/a2  =00000000  r7/a3  =00000000
r8/t0  =00000000  r9/t1  =00000000  r10/t2 =00000000  r11/t3 =00000000
r12/t4 =00000000  r13/t5 =00000000  r14/t6 =00000000  r15/t7 =00000000
r16/s0 =00000000  r17/s1 =00000000  r18/s2 =00000000  r19/s3 =00000000
r20/s4 =00000000  r21/s5 =00000000  r22/s6 =00000000  r23/s7 =00000000
r24/t8 =00000000  r25/t9 =00000000  r26/k0 =00000000  r27/k1 =00000000
r28/gp =00000000  r29/sp =800bc000  r30/fp =00000000  r31/ra =bfc00088

pc     =80020000  mdhi   =00000000
bad va =00000000  status =00400000
```

Memory

Address	Content	Label	
8001FFD8	00 00 00 00		NOP
8001FFDC	00 00 00 00		NOP
8001FFE0	00 00 00 00		NOP
8001FFE4	00 00 00 00		NOP
8001FFE8	00 00 00 00		NOP
8001FFEC	00 00 00 00		NOP
8001FFF0	00 00 00 00		NOP
8001FFF4	00 00 00 00		NOP
8001FFF8	00 00 00 00		NOP
8001FFFC	00 00 00 00		NOP
80020000	20 08 00 05		ADDI $08, $00, 0x5
80020004	20 10 00 00		ADDI $16, $00, 0x0
80020008	20 11 00 01		ADDI $17, $00, 0x1
8002000C	02 11 50 20		ADD $10, $16, $17
80020010	02 20 80 20		ADD $16, $17, $00
80020014	01 40 88 20		ADD $17, $10, $00
80020018	21 29 00 01		ADDI $09, $09, 0x1
8002001C	01 09 58 2A		SLT $11, $08, $09
80020020	11 60 FF FA		BEQ $00, $11, 0xfffa
80020024	00 00 00 00		NOP
80020028	22 31 00 0F		ADDI $17, $17, 0xf
8002002C	00 00 00 00		NOP
80020030	08 00 80 0C		J 0x800c
80020034	00 00 00 00		NOP
80020038	00 00 00 00		NOP
8002003C	00 00 00 00		NOP
80020040	00 00 00 00		NOP

Address mode: Virtual   View mode: Assembler

Figure 4.1. MipsIt simulator's graphical presentation of registers and memory.

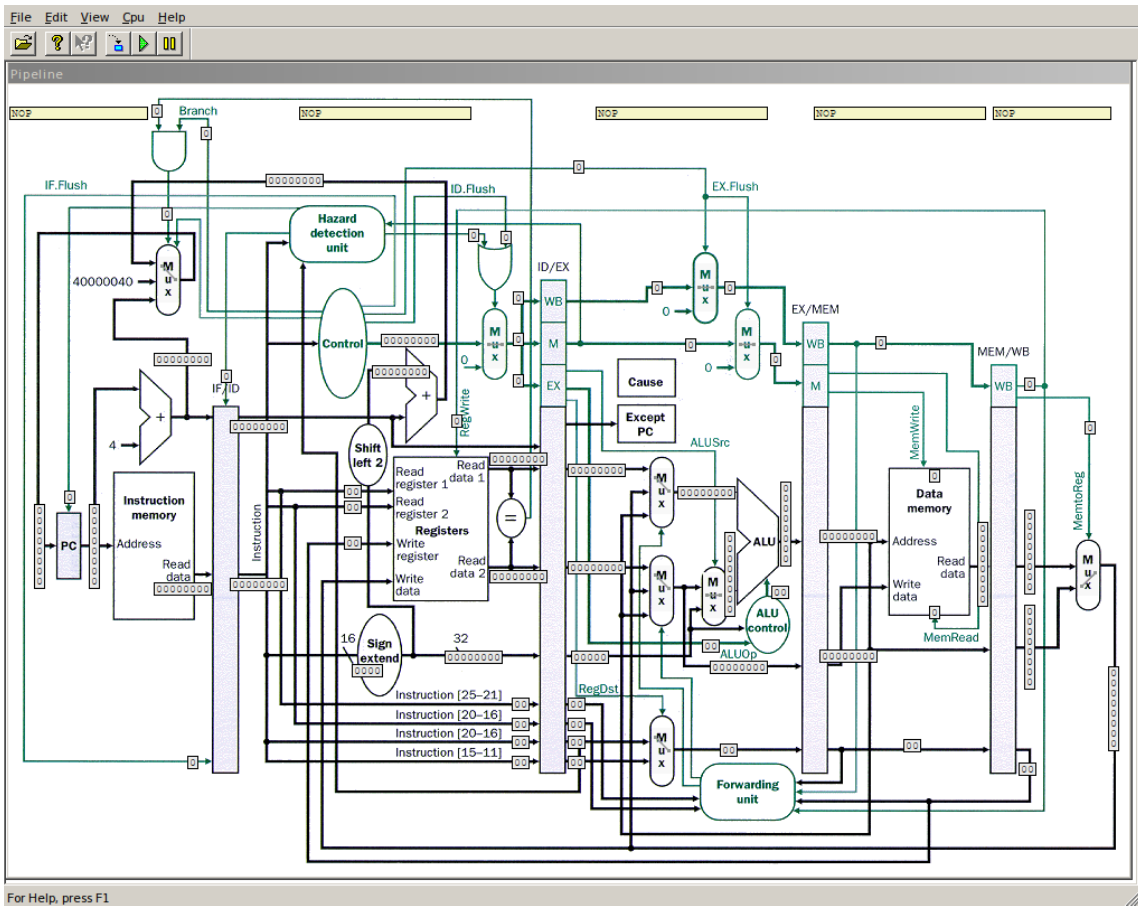


Figure 4.2. MipsIt presentation of pipeline in MipsPipeXL.

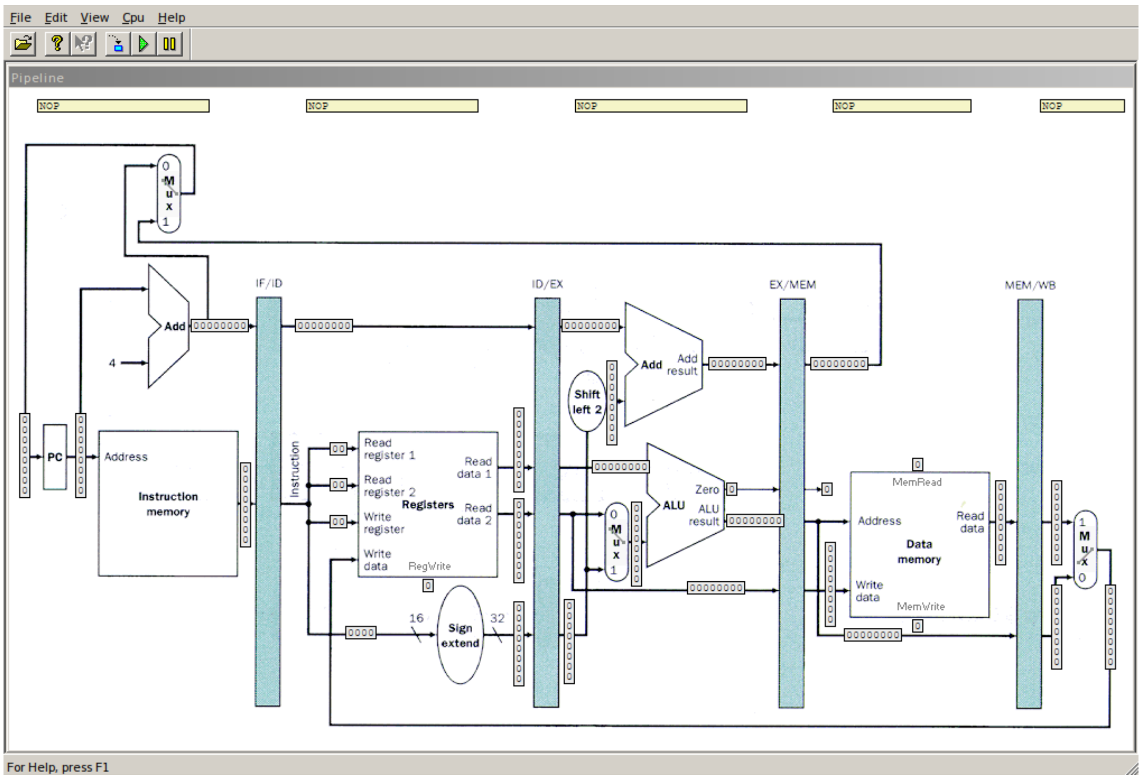
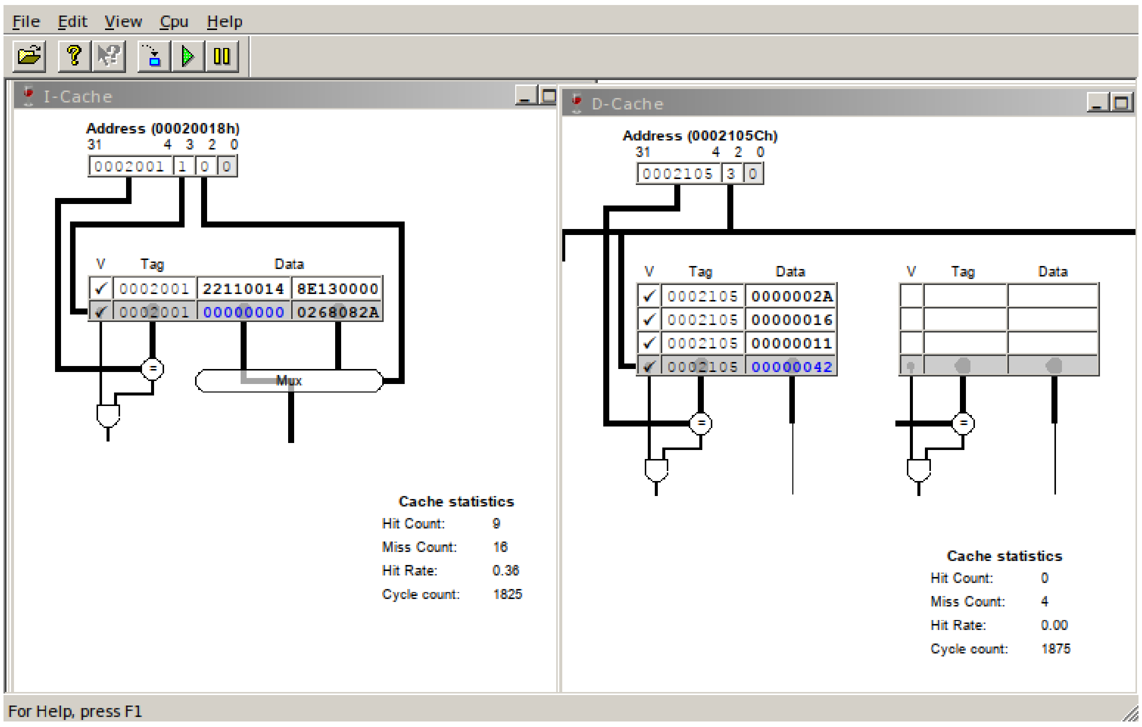


Figure 4.3. MipsIt presentation of pipeline in MipsPipeS without hazard unit.

MipsIt is only simulator described in this thesis that has cache simulation. It simulates isolated instruction (program) and data cache. Parameters such as their size, associativity or policies can be configured. MipsIt also tracks cache usage statistics. Hit and miss count with hit rate is displayed and updated in simulator.

MipsIt simulators expect input in SREC file format[13]. It can be obtained using MIPS compiler and program `objcopy`. This means that GCC can be used to compile code for MIPS and then it can be run in MipsIt.



I-Cache

Address (00020018h)

31	4	3	2 0
0002001	1	0	0

V	Tag	Data	Data
✓	0002001	22110014	8E130000
✓	0002001	00000000	0268082A

Cache statistics

Hit Count:	9
Miss Count:	16
Hit Rate:	0.36
Cycle count:	1825

D-Cache

Address (0002105Ch)

31	4	2 0
0002105	3	0

V	Tag	Data	V	Tag	Data
✓	0002105	0000002A			
✓	0002105	00000016			
✓	0002105	00000011			
✓	0002105	00000042			

Cache statistics

Hit Count:	0
Miss Count:	4
Hit Rate:	0.00
Cycle count:	1875

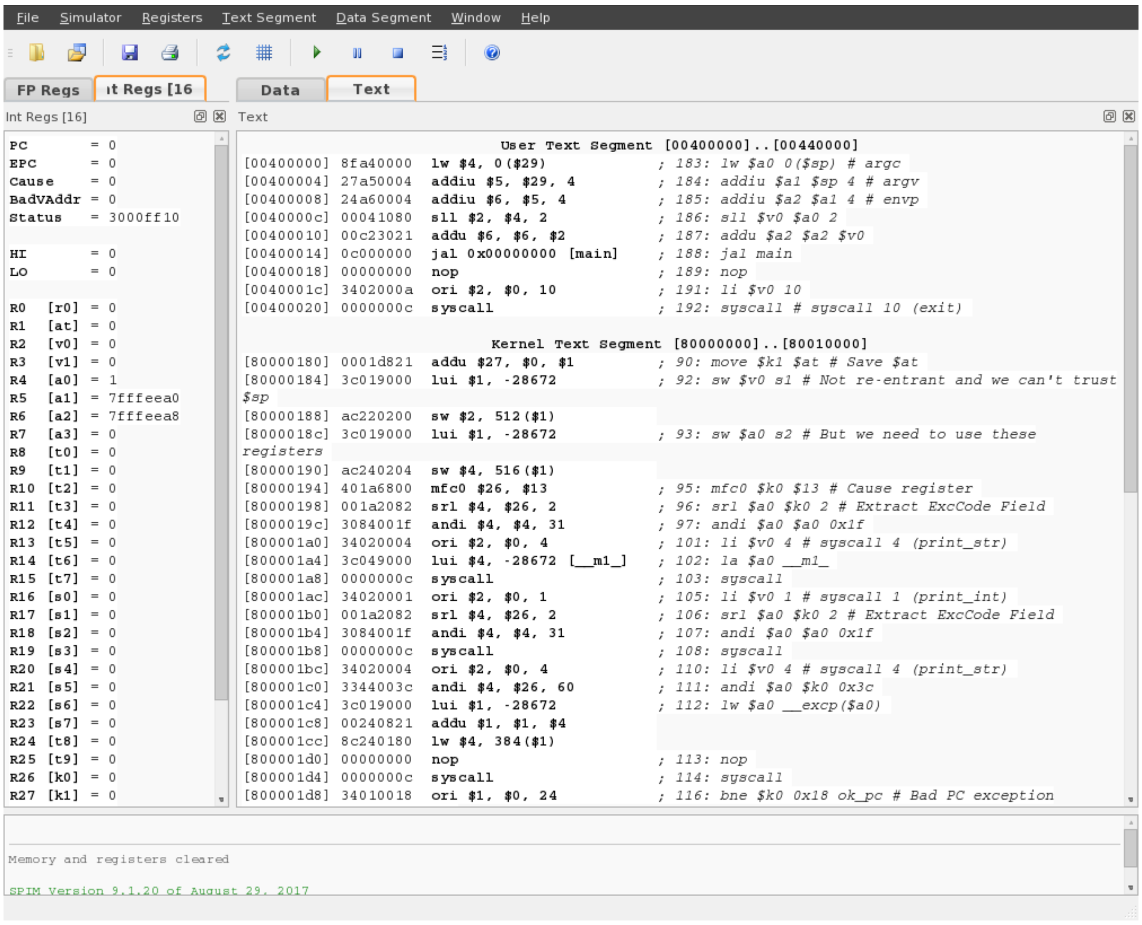
Figure 4.4. MipsIt cache visualization. Instruction cache is on the left and data cache is on the right.

4.2 QtSpim

QtSpim[1] is probably the closest candidate on using it as a code base for this thesis. It's implemented in C++ and base on Qt library. It's licensed under copy-left BSD license [2]. It has complete MIPS CPU implementation including some very advanced features for a simulator such as operation system support. It shows memory in nice and compact way.

Between required but missing features belong cache simulation and CPU circuitry view. It also only loads assembler code. This can be bypassed by simple tool that would export assembler code from ELF file but previous missing features are more problematic.

[1] http://spimsimulator.sourceforge.net/
[2] https://opensource.org/licenses/BSD-3-Clause



```
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] = 7fffeea0
R6   [a2] = 7fffeea8
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
R23  [s7] = 0
R24  [t8] = 0
R25  [t9] = 0
R26  [k0] = 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
[800001c4] 3c019000  lui $1, -28672           ; 112: lw $a0 __excp($a0)
[800001c8] 00240821  addu $1, $1, $4
[800001cc] 8c240180  lw $4, 384($1)
[800001d0] 00000000  nop                      ; 113: nop
[800001d4] 0000000c  syscall                  ; 114: syscall
[800001d8] 34010018  ori $1, $0, 24           ; 116: bne $k0 0x18 ok_pc # Bad PC exception
```

```
Memory and registers cleared

SPIM Version 9.1.20 of August 29, 2017
```

Figure 4.5. QtSpim simulator window with memory and registers view.

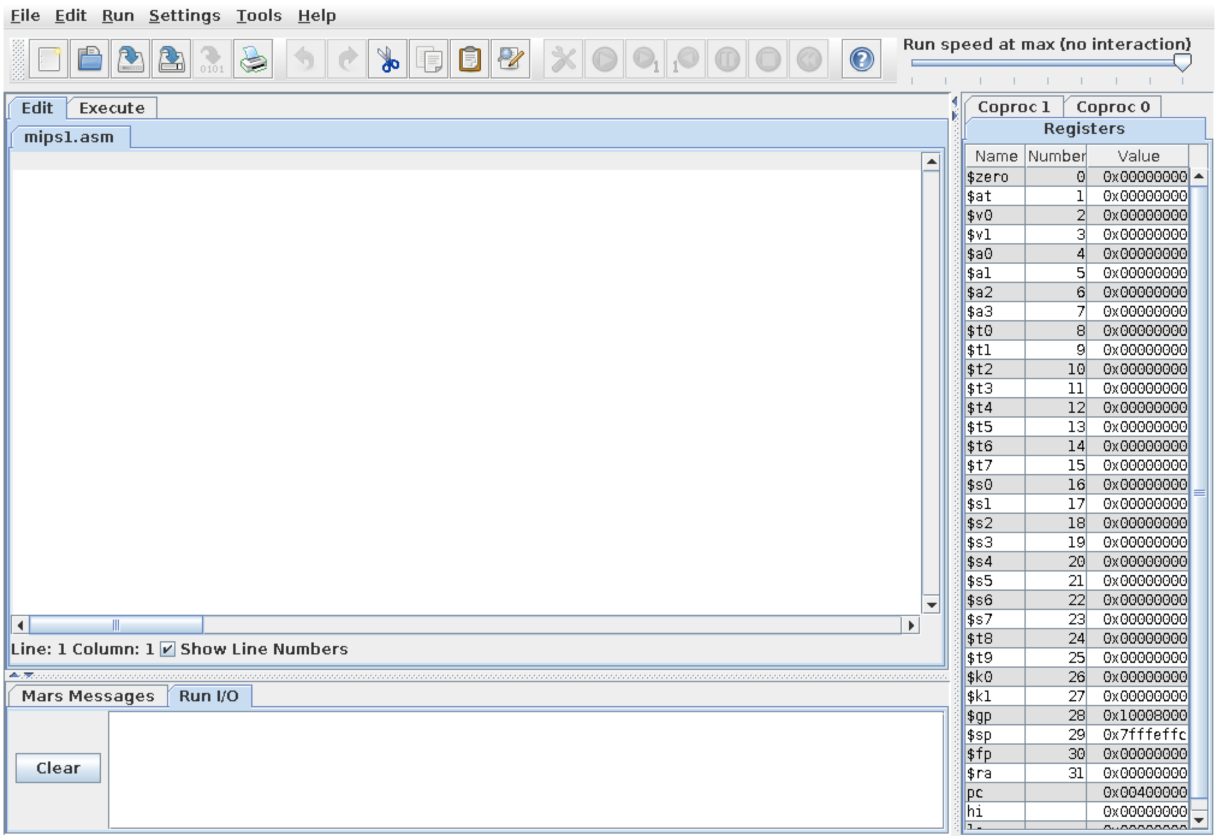
Problem with this program is that it only really simulates instructions behaviour. After short dive into code it is clear that adding circuitry view would required basically append complete circuitry simulation in parallel to existing instructions simulation. Although we could use input and output from QtSpim simulation the idea of getting signals from simulation is pointless as it is based on different idea. It tries to be as effective as possible while our approach is to simulate hardware much more closely. This divide makes code base of QtSpim almost unusable for us and would require redesign.

4.3 Mars

Mars[1] is MIPS assembly simulator developed on Missouri State University. It's written in Java and is licensed under MIT license [2].

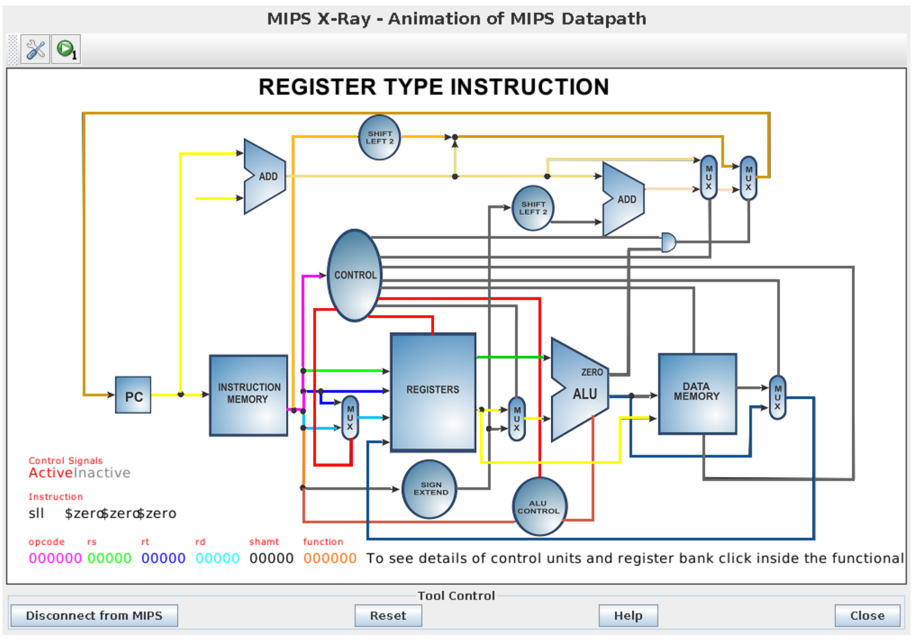
Mars implements its own MIPS assembler parser. It is designed around code editor and implements almost complete MIPS ISA and that is including coprocessors. It was developed and is used for education. It contains various tools for education such as simple attachable simulated hardware such as hexadecimal display with keyboard. It is also able to visualize instruction execution in CPU scheme (as visible on Picture 4.7).

[1] http://courses.missouristate.edu/KenVollmar/MARS/
[2] https://mit-license.org/



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
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000

Figure 4.6. Primary Mars simulator window notably with code editor and registers.



To see details of control units and register bank click inside the functional

Figure 4.7. Mars simulator CPU scheme visualization.

In the course this thesis implements simulator for MARS is suggested to students as an alternative to MipsIt. It almost fits as a replacement for it. Unfortunately it is missing cache simulation. It also does not support preprocessing macros but that is something that is not essential.

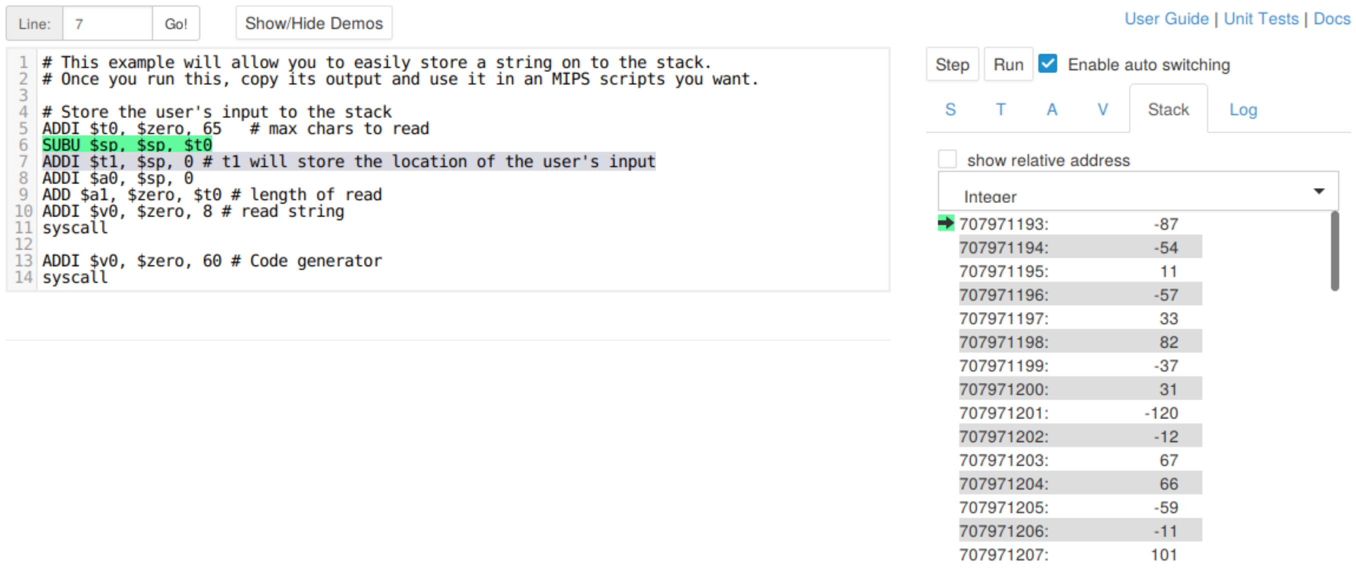
It wasn't used as a code base for this thesis because it is implemented with design where most of the functionality is added on top of simple assembler simulator as additional tools. They are connected more like observers than as an integrated parts of simulator. Implementing cache simulation in such segregated code base would be challenging.

The biggest problem is that Mars does not support pipelined CPU. It is not designed with pipeline in mind and adding it would require a lot of changes not only in tools but also in simulator core. That would require probably complete program redesign.

4.4 WeMips

WeMips[1] is a web based MIPS assembler simulator. Because of that it is implemented in HTML and JavaScript.

WeMips support of MIPS ISA is limited. It supports only few instructions, has no pipeline and memory access is only experimental. It is noted here because of its prominence in Internet search results. It might be suitable for extension thanks to its minimal implementation but this minimal implementation also means that there is not much code to reuse. Choice of programing language and libraries in this case outweighs gain in code base.



```
# This example will allow you to easily store a string on to the stack.
# Once you run this, copy its output and use it in an MIPS scripts you want.

# Store the user's input to the stack
ADDI $t0, $zero, 65   # max chars to read
SUBU $sp, $sp, $t0
ADDI $t1, $sp, 0 # t1 will store the location of the user's input
ADDI $a0, $sp, 0
ADD $a1, $zero, $t0 # length of read
ADDI $v0, $zero, 8 # read string
syscall

ADDI $v0, $zero, 60 # Code generator
syscall
```

☑ Enable auto switching

☐ show relative address

Integer

```
707971193:   -87
707971194:   -54
707971195:    11
707971196:   -57
707971197:    33
707971198:    82
707971199:   -37
707971200:    31
707971201:  -120
707971202:   -12
707971203:    67
707971204:    66
707971205:   -59
707971206:   -11
707971207:   101
```

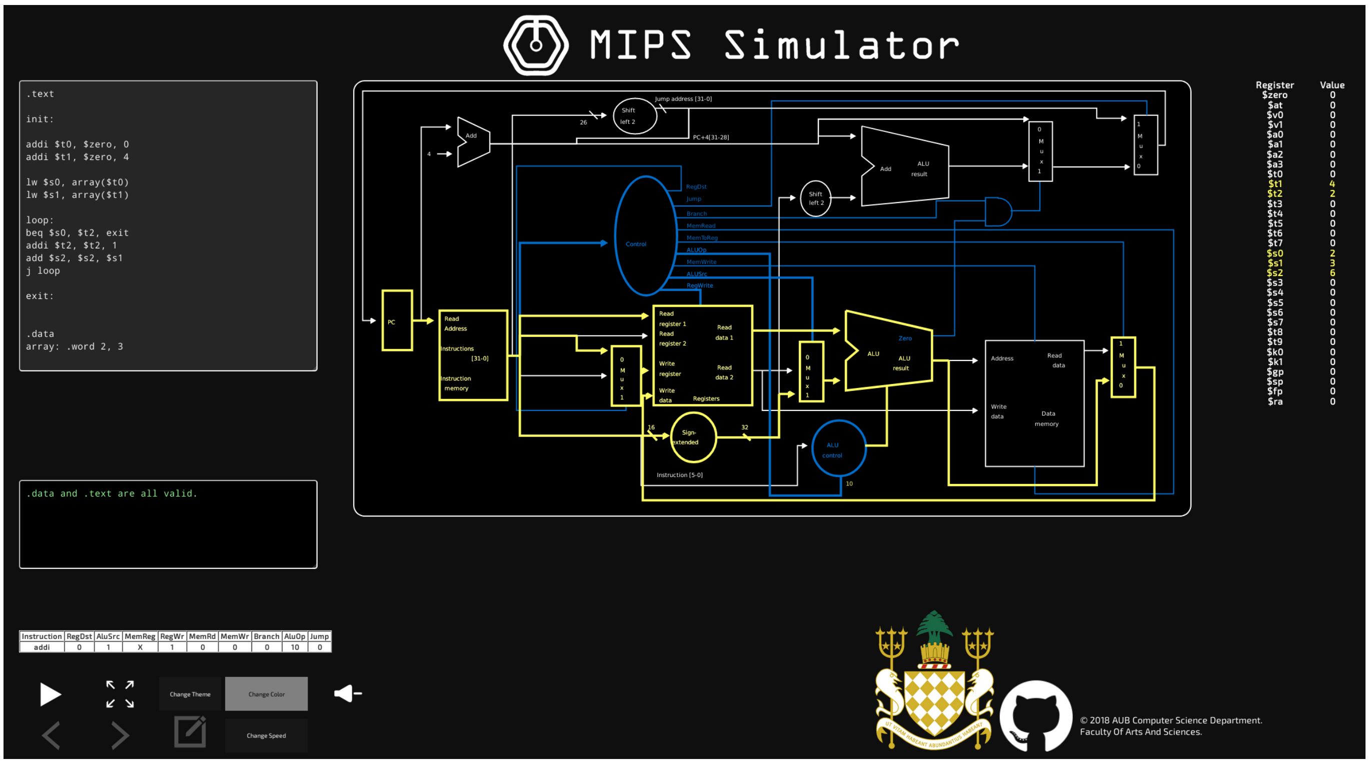
Figure 4.8. WeMips assembly simulator.

4.5 MIPS Simulator (mipssimulator)

MipsSimulator[2] is another web based simulator implemented using HTML and JavaScript. It is licensed with GPLv3 license[3].

This is new project developed at the same time as this thesis. Which is also answer why it was not used as the code base. It is noted here because author seems to be trying to implement minimal but yet visually descriptive MIPS assembly simulator.

[1] http://rivoire.cs.sonoma.edu/cs351/wemips/
[2] http://mipssimulator.com/
[3] https://www.gnu.org/licenses/gpl-3.0.en.html



```
.text

init:

addi $t0, $zero, 0
addi $t1, $zero, 4

lw $s0, array($t0)
lw $s1, array($t1)

loop:
beq $s0, $t2, exit
addi $t2, $t2, 1
add $s2, $s2, $s1
j loop

exit:


.data
array: .word 2, 3
```

```
.data and .text are all valid.
```

Register	Value
$zero	0
$at	0
$v0	0
$v1	0
$a0	0
$a1	0
$a2	0
$a3	0
$t0	0
$t1	4
$t2	2
$t3	0
$t4	0
$t5	0
$t6	0
$t7	0
$s0	2
$s1	3
$s2	6
$s3	0
$s4	0
$s5	0
$s6	0
$s7	0
$t8	0
$t9	0
$k0	0
$k1	0
$gp	0
$sp	0
$fp	0
$ra	0

Instruction	RegDst	AluSrc	MemReg	RegWr	MemRd	MemWr	Branch	AluOp	Jump
addi	0	1	X	1	0	0	0	10	0

Figure 4.9. Website with MIPS Simulator going trough default code and animating data propagation in scheme.

4.6 Qemu

Qemu[1] is emulator and hypervisor with wide range of supported architectures. It is designed as an generic emulator and virtualizer. MIPS is just one of the many supported architectures. Thanks to Qemu aim on virtualization it is quick and complete ecosystems exists around it.

Qemu implements complete MIPS ISA and not only in version I. Other MIPS versions are also supported including emulation of existing microprocessors such as R2000.

Qemu was not used in this thesis because its goals are different from this thesis' ones. This thesis cares less about efficiency and speed and more about simulations aspects. We are more interested in simulating interworking of CPU than in emulation of CPU behaviour. Because of that using Qemu is out of scope.

4.7 Hardware Description Based simulation

Open-source implementations of MIPS CPU in hardware description language such as VHDL exist. Those can be used to implement CPU on field-programmable gate array (FPGA). Advantage of this is that it is almost same like working with real microprocessor. It is even possible to connect real hardware peripheries. However, that is not what is needed for this thesis because in such case it is not possible to study data flow in processor.

[1] https://www.qemu.org/

These hardware centric implementations of MIPS ISA can be used in software together with GHDL[1]. GHDL is compiler/simulator for VHDL for regular operation systems. It allows VHDL implementation to be simulated on PC.

This approach was heavily consider early in this thesis research. Plan was to use an existing MIPS I VHDL implementation and by inserting C code to it (which is GHDL extension of VHDL) it would be possible to interconnect GUI and simulator.

Advantage of this is that it would result to simulation that is as close to real CPU as possible. All signals and buses are already implemented and only output to GUI would have to be added. It would also resulted in less work to do in this thesis. When original VHDL code would be only minimally modified then it would be easy to pull fixes from original upstream project.

Unfortunately we found out that there is no appropriate VHDL implementation that would implement five stage pipeline and cache at the same time. Also having implementation with pipeline would still require changes that it can be also run without pipeline. Another problem is how to configure simulation parameters without need to recompile VHDL code every time.

One of the considered VHDL implementations was Plasma[2]. It is licensed as completely free in public domain. This is implementation with two or three pipeline stages and with cache. It supports GCC as program compiler. It is well documented and contains wide range of additional features. Unfortunately missing five way pipeline is major problem and because of that this implementation was not used.

The second considered VHDL implementation was miniMIPS[3]. It is licensed with LGPL license. It has smaller code base than Plasma and it is five way pipeline implementation. Unfortunately it does not contain cache implementation and also it requires its own assembler compiler (not using GCC).

[1] `http://ghdl.free.fr/`
[2] `https://opencores.org/project/plasma`
[3] `https://opencores.org/project/minimips`

Chapter 5
MIPS Simulator Design

Simulator itself was implemented in C++ using Qt toolkit. This was primarily decision of submitter but nevertheless it is good choice. Object oriented programming language is suitable for GUI abstraction and Qt toolkit is cross platform which makes it good choice for education tool that students could potentially run on their own computers without need of complicated first setup. Qt toolkit has other benefits but they are primarily related to graphics and because of that they are discussed in beginning of Chapter 6.

CPU emulation is a core part of QtMips. It was designed to be separate from graphics visualisation and because of that was implemented as a dynamically linked library. CPU emulation can be divided to following parts:

- Simulator configuration
- Registers simulation
- Memory simulation (this includes caches and memory mapped inputs/outputs)
- Instruction decoding to control signals
- Execute instruction by applying control signals
- Program loading
- Reverse instruction decoding

All parts are described in depth in following sections. They don't serve as reference for related code. Although they specify what exact file they talk about. They rather contain description of basic concepts used in code they reference.

The command line interface application was implemented outside of the thesis scope and is not described here. It was developed for testing but it can also be used for automatic simulations.

5.1 Simulator Configuration

Simulator is required to be configurable as discussed in Section 3. This is ensured by dedicated class that creates storage for configuration variables. Object instantiated from this class is then required for simulator initialization.

Configuration class is not just simple value storage with setters and getters. It also has to check some configuration limitations. For example there is possibility to disable delay slot but that is possible to do only when no pipelining is used. So when pipelining is enabled then it is required to always report delay slot as being enabled. There are more of these limitations. They arise from logical limitations and architecture design. There is no benefit on digesting them here separately but it is important to note that they are implemented there and they ensure that values later received from configuration are verified and sanitized.

In short following configuration options exist:

- pipelined: what ever pipelining is enabled

- delay_slot: if delay slot should be simulated (effectively configurable only if pipelined is set to false)
- hazard_unit: how hazard unit should behave. It allows no hazard unit or hazard unit that only stalls on top of full hazard unit (effectively configurable only if pipelined is set to true otherwise set to no hazard unit)
- memory_access_time_read: allows setting number of cycles memory read access should take. This is used only for statistics. It's not simulated.
- memory_access_time_write: same as memory_access_time_read but for write access
- elf: path to elf file to be loaded as a program to memory (note that this one is not verified)
- cache_program: object of cache configuration class for program memory cache
- cache_data: object of cache configuration class for data memory cache

Configuration allows setup of two caches. One for program loading and another one for data manipulation. Also referred as program and data memory accesses. Both such cache configurations should be separately configurable. For this abstraction they are implemented as one additional class describing cache configuration. This class is implemented with same design decisions as primary configuration class. Implemented options are:

- enabled: If cache is enabled or not
- sets: number of sets
- blocks: blocks size
- associativity: degree of associativity
- replace_mentpolicy: replacement policy to be used
- write_policy: write back policy to be used

For simplicity configuration also allows some presets. Those are implemented as method that just simply sets needed option for given preset. Graphics visualization later requires to know if some preset is chosen but that is not supported and is later handled explicitly as part of GUI implementation.

It's possible to add additional presets. It should be as easy as adding new symbol to enum and new case to switch-case statement in relevant code. On top of that relevant radio button has to be added in GUI.

Configuration classes can also be saved using Qt class `QSettings` and then loaded back. This is used for preserving configuration of simulator between application launches when GUI is used.

Configuration classes are implemented in source files *machineconfig.h* and *machineconfig.cpp*.

5.2 Registers Simulation

Registers are core part of the CPU. But from simulator point of view it is as easy to implement them as defining variables to store their content in. In QtMips there is more in depth abstraction built on top of class used as simple value storage. For general purpose registers and LO and HI registers there are simple getter and setter methods. But there are specialized methods for program counter. They directly correspond to branch and jump instructions. There are three types of branches/jumps (for simplicity referred here as a jumps because in case of registers we don't care if they are conditional or not). There is a relative jump. This jump expects signed value that is added to current program counter value. This way it is possible to jump both forward and backward in code on near enough addresses. This jump is the one that most of the branch instructions are implemented by.

Then there is an absolute jump. This jump simply overrides current value of program counter register. This way it is possible to jump everywhere in program memory. This jump type is used exclusively in jump register instruction.

And the last jump type is absolute jump but only with 28 bits. This is exclusively used by jump instruction (instruction named jump). It allows program to do absolute jump but because of limited space in instruction coding it can't specify all 32 bits. Instead it specifies maximum possible 26 bits. Those are binary left shifted by two bits because two lowers bits are not required. Using them would lead to unaligned jump. That gives us 28 bits. Using four upper most bits from current program counter value gives us complete 32 bit address. It is not possible to jump with such instruction anywhere in memory but it allows absolute jumps at least in current memory segment.

Registers abstraction is implemented as a single class in file *registers.h* and *registers.cpp*.

5.3 Memory Simulation

With memory implementation it was required to decide what is correct level of simulation. If limited memory size should be implemented or not. For simplicity, route where all memory is accessible and usable was chosen. This is mainly because to have meaningful limited memory size simulation we would need virtual memory and that requires coprocessor 1 (see Chapter 2 for coprocessors).

This establishes that QtMips simulates whole 32bit addressable memory space. There are no added limitation on used memory. This means that user can potentially use all memory of host system up to 32bit addressable size which is approximately 4GB of allocated space. This means that unless user needs whole 32bit address space, which is unlikely, it should never be directly allocated. This thought leads to implementation where memory is allocated in segments as is needed. Searching for such segment is done using tree[14] where nodes contain multiple edges (links) to other nodes. Every node is indexed by bits from address. What bits are used is given by depth of given node in tree. Leafs of the tree then contain allocated memory segments (in code referred as memory sections). In other words there are tables that are linked in sequence and indexed by subsequent bit segments from address. Depth and width of this tree is configurable but limitations are that number of bits used per node multiplied by tree depth plus number of bits used for indexing memory section is combined 32 bits. This is limitation of described algorithm.

By decision if some section that was never written is read then it is read as zero. And newly allocated section is zeroed.

Object oriented programming is used in memory implementation for great benefits. There is an abstraction class referred as memory access that creates common memory access interface. This layer covers less than word size accesses for read and write requests. Because of that it also has to handle miss match between MIPS simulator and host system endianness. But primarily it allows implementation of additional classes that would be inserted in front of the memory it self. There is almost no need in using memory directly and because of that memory access class is used instead. Implementing new child class and passing its object instead of real memory object to simulator allows easy cache addition (see Section 5.3.1).

Memory class allows only access, ignoring access to internal tree structures, with data of size word. Memory access class on top of that allows access not only to words but also to halfwords and bytes. This covers all supported data types (see Section 2.1).

Memory is implemented in *memory.h* and *memory.cpp* source files.

5.3.1 Cache Implementation

Cache can be easily added on top of memory simulation thanks to design described in Section 5.3. It's done by implementing child of memory access class.

In following paragraph are used configuration names as described in Section 5.1. They are also in depth described in Section 2.5.1.

Cache implementation it self is fairly straightforward. Class implementing it is allocating array of structures containing field for data, tag and dirty and valid booleans. Depending on configuration it also allocates array for tracking access times or access counter. Then, when there is read or write requested (thanks to memory access class it is always of word type) it locates relevant index, verifies tag and validity and optionally receives data from memory. On top of that there is also possibility to invalidate whole cache content.

Cache tracks access statistics by incrementing cache miss and hit counters. Adding these counters we can then get total number of accesses to memory. Let's use h as number of hits, m as number of misses and c as total number of accesses (meaning $c = m + h$). Having cycles penalisation for memory access as p (meaning number of cycles needed to access memory) then we can calculate memory stalled cycles (effectively wasted cycles) by $m * (p - 1)$. Another interesting cache statistic is speed improvement. This calculates speed improvement relative to speed of CPU without cache. It is calculated as $\frac{(m+h) \cdot p}{h + m \cdot p} \cdot 100\%$. But probably most usable implemented statistic is cache usage effectiveness. This value informs user on how well is cache used in simulated program. It doesn't take in consideration effect on program execution speed but it is just plain hit to miss rate statistic. It is calculated using this formula: $\frac{h}{m+h} \cdot 100\%$.

Cache is implemented in *cache.h* and *cache.cpp* source files.

5.4 Instruction Decoding

MIPS instructions are well designed and allow easy decoding of control signals from opcode it self. This is handy when designing hardware but using this in software can by less optimal. Chosen way in this thesis is lookup table. Instead of decoding signals from opcode it self we use that opcode to look them up in table with already predefined signals. This simplifies implementation as it is just protected array access instead of some bits decoding. It also mitigates any need for explicit handling of some instructions that would otherwise be needed. The chosen approach however creates a problem that every instruction has to be hand crafted and can't be simply let be as a result of opcode binary decode. Because there is not huge amount of generated signals it is viable to just hand fill them.

Following signals are decode from opcode:

- Supported: Whether is instruction with given opcode supported
- Write result to register: If result of operation should be written to register
- *rd* field used: Whether given instruction has rd field
- ALU operation: This is non-boolean value that defines used ALU operation

- ALU source: Whether should one of the inputs to ALU be from register or from sign extended immediate instruction field
- Memory read: If memory should be read
- Memory write: If data should be written to memory
- Memory data type: This is non-boolean value that specifies what type of data memory should output/write to (see Section 2.1 for list of available data types)

With noted exceptions they are all boolean values.

There is one group of instructions that share opcode 0. Those are all mostly arithmetic or logic operations. For them instruction field function is directly used as an ALU operation. There are also other instructions that have to be specially handled and those are all jump and branch instructions. Those are handled explicitly in their own method using switch-case statement. Reasoning behind their separation is in following Section 5.5.

Instruction decoding is implemented in *core.cpp* source file.

5.5 Instructions Execution

Instruction execution can be divided to 5 stages, as described in Chapter 2 about MIPS architecture. These stages are in code implemented as a separate methods. They always take as an argument a structure containing all signals (stored in variables) passed from previous to this stage (of course with exception of fetch stage as there is no preceding one). On top of that to correctly implement jump logic it is required to have additional method just for it. We can't correctly implement it as part of fetch method because jump requires signals from decode stage.

Whole CPU is simulated by periodically calling method called `step`. This simulates system clock. This method is the central point of CPU simulation implementation.

There are two major ways methods implementing stages are connected in `step` method. One is where methods are called in sequence order (from fetch to write-back stage) where next stage takes output from previous one. The last executed methods is on top of that the one handling jump logic. This way is implements non-pipelined CPU. Another way to implement `step` method is to have signals stored between CPU steps and call stage methods in reverse order (from write-back to fetch stage). Last method run is, same as in previous case, method handling jump logic. This way it implements pipelined CPU. These two implementations are implemented separately and inheritance is used where `step` method is defined as virtual.

Instruction execution is implemented in source files *core.h* and *core.cpp*.

5.5.1 Hazard Unit

Hazard unit is required only when pipelining is used as described in Section 2.4 (although even then it is possible to deactivate it). Because of that it is implemented in `step` method for pipelined CPU.

Hazards are detected by checking used registers in instruction. With exception of general purpose register zero. This register can't change value and because of that it can't cause hazard. Implementation checks if currently decoded instruction is not using as either *rs* and/or *rt* register that is used by instruction being in either execute or memory stage as output (either *rt* or *rd* depending on instruction). Two possible resolutions are applied according to Table 2.2.

Forward is implemented by changing stored value that is in next `step` call passed to executed stage. As source is used value from execute or memory stage depending on which stage causes hazard.

Stall is implemented by not executing method handling jumps (program counter is not updated). Also signals passed from decode to execute stage are wiped. Effectively setting them to state that would instruction NOP (no operation) set them. This means that execution of fetched instruction is hold off by one cycle and to pipeline is effectively inserted NOP instruction.

5.6 Program Loading

Program to be simulated is in ELF file format. Various libraries exists for loading this file format. For this project libelf from project elfutils [1] was used. It was selected because there are multiple other implementations of libelf API and that makes it more or less standard for ELF file reading/writing. Additional reason is that elfutils it native implementation from GNU project, platform which QtMips was developed on.

Considered alternative was usage of GCC library. But that would tie up QtMips to GCC. GCC library also supports more then just ELF files reading and we don't need any of those additional features.

Program loader reads file from path that is passed to it from simulator configuration (see Section 5.1). It opens it and passes resulting file descriptor to libelf. Then using libelf it is verified that it is really ELF file and that it has correct endianness and is compiled for MIPS ISA. When all these checks are done then it goes through all sections that are present in loaded ELF file and select only those marked as to be loaded. Then when requested it can byte by byte dump those sections to program memory (its implementation is described in Section 5.3). Exact address where given section will be loaded to is read from ELF file[15].

Program loader is implemented in *programloader.h* and *programloader.cpp*.

5.7 Decoding to Instruction Mnemonic

Conversion from binary representation of instruction back to assembler representation is required Because we want to show executed instructions back to the user. We could use debug informations contained in ELF file but that is complicated and result is not certain as program might be compiled without them or they might be no longer valid. Also for students it is beneficial to see real representation of pseudo instructions rather then their original form. Because of that the same approach as for instruction decoding (see Seciton 5.4) was used. There is a lookup table with few configuration flags and string representation of instructions.

This conversion doesn't have to be perfect. Where syntax like `ADDI $3, 0x24` is valid it is shown as more verbatim `ADDI $3, $3, 0x24`.

It's implemented in *instruction.cpp* source file.

[1] `https://sourceware.org/elfutils/`

Chapter 6
MIPS simulator Graphics Design

Graphical simulator is implemented as an executable application that uses MIPS simulator library (discussed in Chapter 5).

As was already noted simulator is implemented in C++ using Qt toolkit. But other options were also considered. There are two considerable cross-platform full fledge GUI toolkits that are available on Linux: GTK+ and Qt. There are of course other toolkits but those are not so prominent. GTK+ is C library that implements just GUI toolkit itself. It is not tied up to some other unrelated features. In comparison Qt toolkit for example provides even alternative interface for standard input and outputs. GTK+ would be minimalistic choice and would probably be even better in some cases such as scheme rendering. That is because it is build on top of very versatile library Cairo. But in the end Qt toolkit was chosen because of single integration feature: signals & slots.

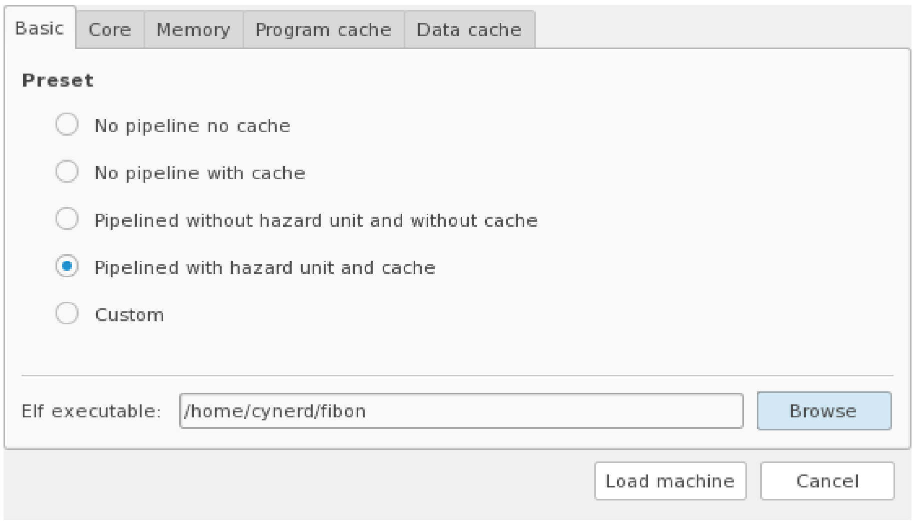
Signals & slots is mechanism that allows one way communication between objects. Objects implementing this communication are required to just define signals and slots. Signal is source. It then can be connected to other signal (creating chain of signal propagation) or to slot. Slot is special method in class that is called when signal is emitted. This commonly happens immediately after signal is emitted[16].

Thanks to this design there is no need for interconnected objects to know each other. Bounding is done externally after both objects are inicialized. Bound is also automatically dissolved when one of the objects is deleted. This behavior simplifies code and is perfect for building GUI implementation on top of changing simulator implementation. When simulator instance is replaced then there is no need to explicitly unbound old.

In following sections we describe parts of GUI in simulator. Similarly as in previous chapter this is not meant as code reference but rather as description of concepts used in implementation and some of the encountered problems (such as problem with rendering fonts).

6.1 Simulator Configuration Dialog

Simulator configuration dialog is the first input shown to user when GUI version of simulator is started. It presents user with graphical way to set options that were mentioned in Section 5.1. Whole window is divided to five tabs splitting configuration to groups. First tab is named 'Basic'. It contains preset selection and ELF executable path.



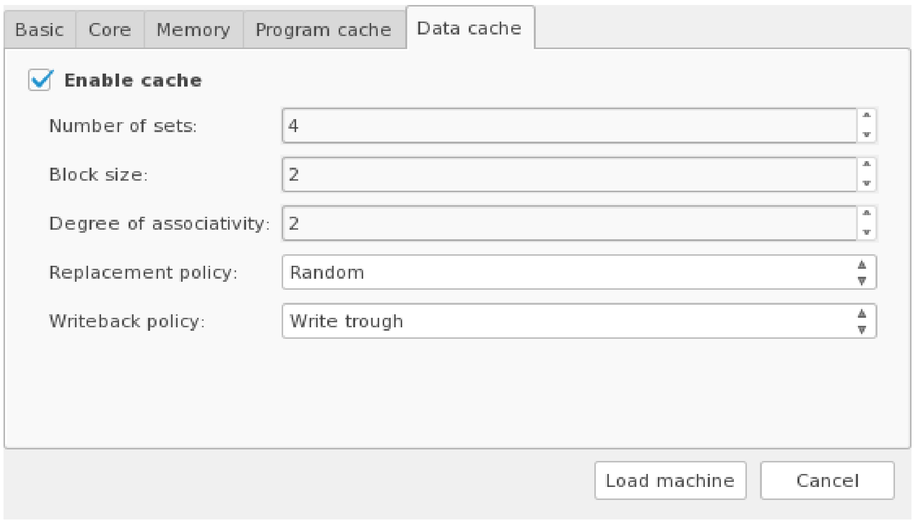
Preset

- ( ) No pipeline no cache
- ( ) No pipeline with cache
- ( ) Pipelined without hazard unit and without cache
- (●) Pipelined with hazard unit and cache
- ( ) Custom

Elf executable: /home/cynerd/fibon

Figure 6.1. Configuration dialog with presets and ELF file selection.

Second tab named 'Core' contains primary CPU configuration. Those are more accurately options pipelined, delay_slot and hazard_unit. Third tab labeled 'Memory' contains memory access time configurations. Fourth and fifth tabs are constructed from same template. Fourth tab is for 'Program cache' and fifth one is for 'Data cache' configuration. They both contain field for configuration of all cache options. There is only one exception. On fourth tab there is input for write_policy (labeled as 'Writeback policy') hidden. That is because program cache is never used for memory writes and having this option present would be confusing.



✓ **Enable cache**

Number of sets:	4
Block size:	2
Degree of associativity:	2
Replacement policy:	Random
Writeback policy:	Write trough

Figure 6.2. Configuration dialog with cache configuration tab.

Dialog itself is implemented mostly as Qt form designed in Qt Designer. There is additional C++ code added to tweak some appearence and to sync form values with simulator configuration (sync it self is described more in depth in next paragraph). Cache tabs are implemented as one additional Qt design form that is instanciated twice for both program and data cache.

Input field from this dialog has to be applied to simulation configuration object. This is done by implementing slots connected to signals from relevant input. Such slot method converts input value to correct data format and calls appropriate setter from simulator configuration object. Every such slot method also calls method updating status and content of all inputs. It resets content of all inputs to the one given by relevant getter from simulator object. Depending on this configuration it also disables or enables some of the inputs. This effectively allows implementation of configuration logic to be solely in simulator configuration class. GUI it self is then appropriately updated. Another indirectly gained feature from this implementation is that configuration values that are not valid are preserved if different value is forced because of some other option. Example of this can be if pipelined is disabled then hazard_unit is forced to be disabled too. But when pipeline is re-enabled then hazard_unit jumps back to originally chosen preference.

As noted in Section 5.1 simulator configuration supports setting of presets but it is not able to then verify if configuration is some specific preset (this is done to simplify code itself). But for GUI it is required to show user if preset is used or if configuration is custom. It is easy to know which preset is configured when user changes preset. The same way it is not hard to switch it back to custom when any other input is changed (with exception of ELF path). It's implemented just by connecting appropriate signals and slots. When program is restarted then preset choice should be persistent too. Because of that it is required to on top of just storing simulation configuration it self also store this preset (using `QSettings` object). That way it is possible to restore whole configuration including preset choice.

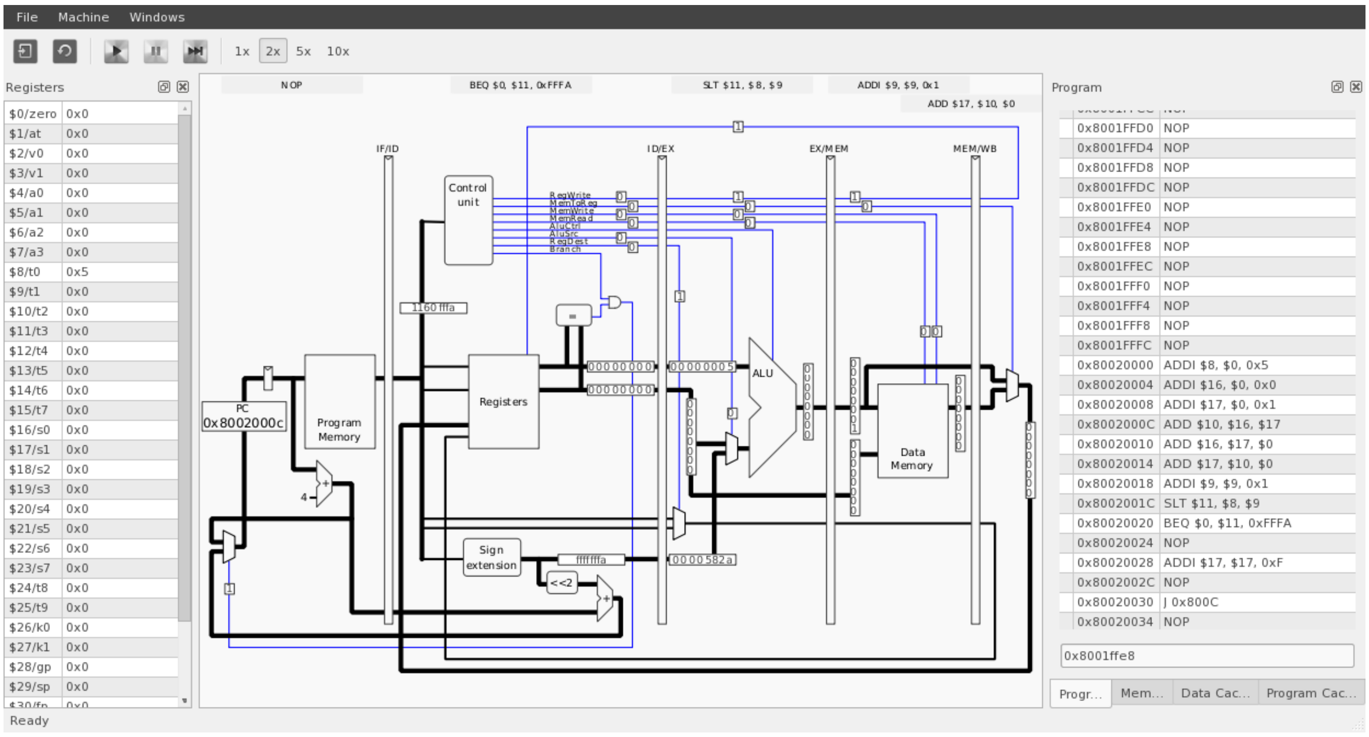
Dialog can be exited either by successfully initialization simulation or by canceling it. If there is no existing simulation and dialog is exited then this causes whole program exit. Back to simulation initialization when this fails then dialog is not closed and error message is displayed. Common source of this is invalid ELF executable path. When simulation is successfully initialized then dialog is hidden and main simulator window is made accessible. This also causes current configuration to be stored for future use (using `QSettings` object).

Forms for this dialog are named *NewDialog.ui* and *NewDialogCache.ui*. C++ code paired with these forms is in source files *newdialog.h* and *newdialog.cpp*.

6.2 Main Simulator Window

Main simulator windows is primary interface for user interaction. It is implemented as window with tool bar, menu, status bar and primarily space for CPU simulator scheme. There are also additional dockable windows. Those can be opened either by using menu or by double clicking on some parts of the CPU scheme. In default those windows are opened as docked on right side of the simulator window. User can move them to different position or undock them. Their state is preserved between program restarts using `QSettings`.

Parts of simulator GUI such as various dockable windows and CPU scheme are discussed later In following sections. Because of that this sections describes only parts of simulator window that are not strictly part of some other section.



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	0x5
$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	0x0

Program

0x8001FFD0	NOP
0x8001FFD4	NOP
0x8001FFD8	NOP
0x8001FFDC	NOP
0x8001FFE0	NOP
0x8001FFE4	NOP
0x8001FFE8	NOP
0x8001FFEC	NOP
0x8001FFF0	NOP
0x8001FFF4	NOP
0x8001FFF8	NOP
0x8001FFFC	NOP
0x80020000	ADDI $8, $0, 0x5
0x80020004	ADDI $16, $0, 0x0
0x80020008	ADDI $17, $0, 0x1
0x8002000C	ADD $10, $16, $17
0x80020010	ADD $16, $17, $0
0x80020014	ADD $17, $10, $0
0x80020018	ADDI $9, $9, 0x1
0x8002001C	SLT $11, $8, $9
0x80020020	BEQ $0, $11, 0xFFFA
0x80020024	NOP
0x80020028	ADDI $17, $17, 0xF
0x8002002C	NOP
0x80020030	J 0x800C
0x80020034	NOP

0x8001ffe8

Figure 6.3. Main simulator window.

Menu in main window allows control of simulator and possibility to open all possible windows. In Qt menus are composed of actions. Those have a name, description and also global keyboard shortcut. Thanks to this QtMips supports some quick keyboard shortcuts making its usage more pleasant. Shortcuts can be seen in window menu on right side of action button but to mention few important ones:

- Ctrl+O: This is standard shortcut for opening new file. In QtMips it reopens simulator configuration window.
- Ctrl+Shift+R: Pressing this shortcut causes simulation reload. It wipes current CPU status and reloads program from original ELF file.
- Ctrl+S: Using this shortcut when simulation is paused does one CPU step. It's handy for stepping the simulated program.

Status bar is currently mostly unused. It only signals change in simulation state. There are four known possible states at the moment. Either simulation is *Ready* which means that it's not running that it's paused. Or it can be *Running*. Another state is failure one reported as *Trapped*. And last one is *Exited* for state where simulator reached end of program.

Main window is implemented in source files *mainwindow.h* and *mainwindow.cpp* and its implementation class is the one instanciated in GUI program `main` function.

6.3 CPU Scheme View

CPU scheme is graphical representation of internal interconnection of implemented CPU. This can considerably vary depending on simulator configuration. That requires some level of flexibility. This was achieved by dividing scheme to blocks and connections between them. Used blocks, their positions and connections between them depends on current simulator configuration.

Primary change to scheme is if pipelined is set or not. Meaning if we should visualize scheme with pipelining or not. This was archived by creating one base class and then two children. One for non-pipelined and another one for pipelined scheme. Additional changes triggered by other configuration options are then implemented as conditional branches in code.

Scheme visualization is implemented using Qt Graphics View Framework. This framework provides widget `QGraphicsView` that is used as graphics container. Graphics it self is then build in `QGraphicsScene` from objects of `QGraphicsItem` type[17]. This creates abstraction where items are placed to scene and scene itself is then transformed to view. CPU scheme uses this to render smallest picture that is scaled up to fit view. Additional code was implemented that tries to automatically maximize used space by scheme. It also ensures that scheme stays readable even with small resolution by allowing scroll over limited are of scene. Additional features of Qt Graphics View such as manual zoom were suppressed because those are not needed for scheme navigation.

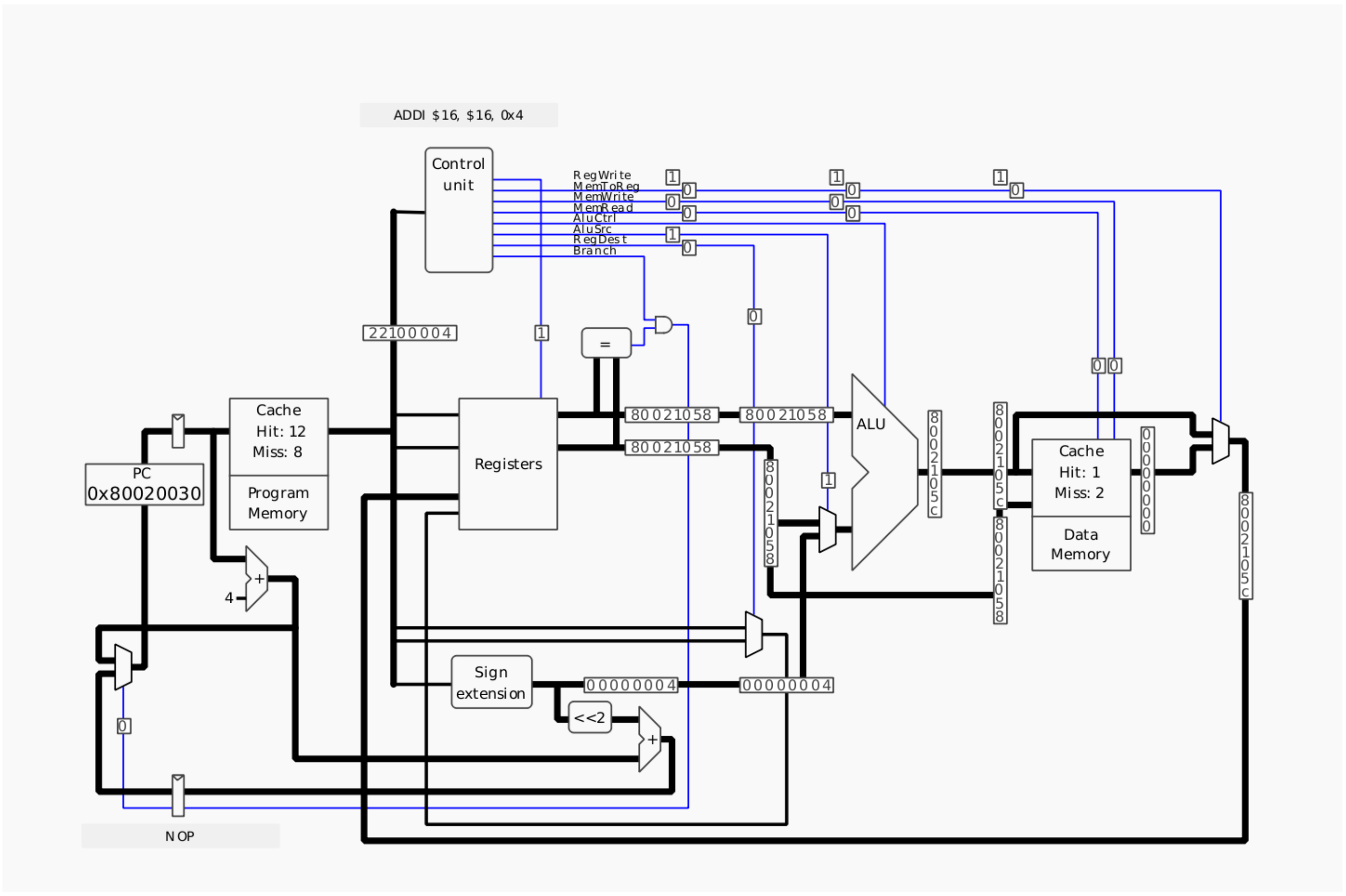


Figure 6.4. Single-cycle CPU scheme with program and data cache.

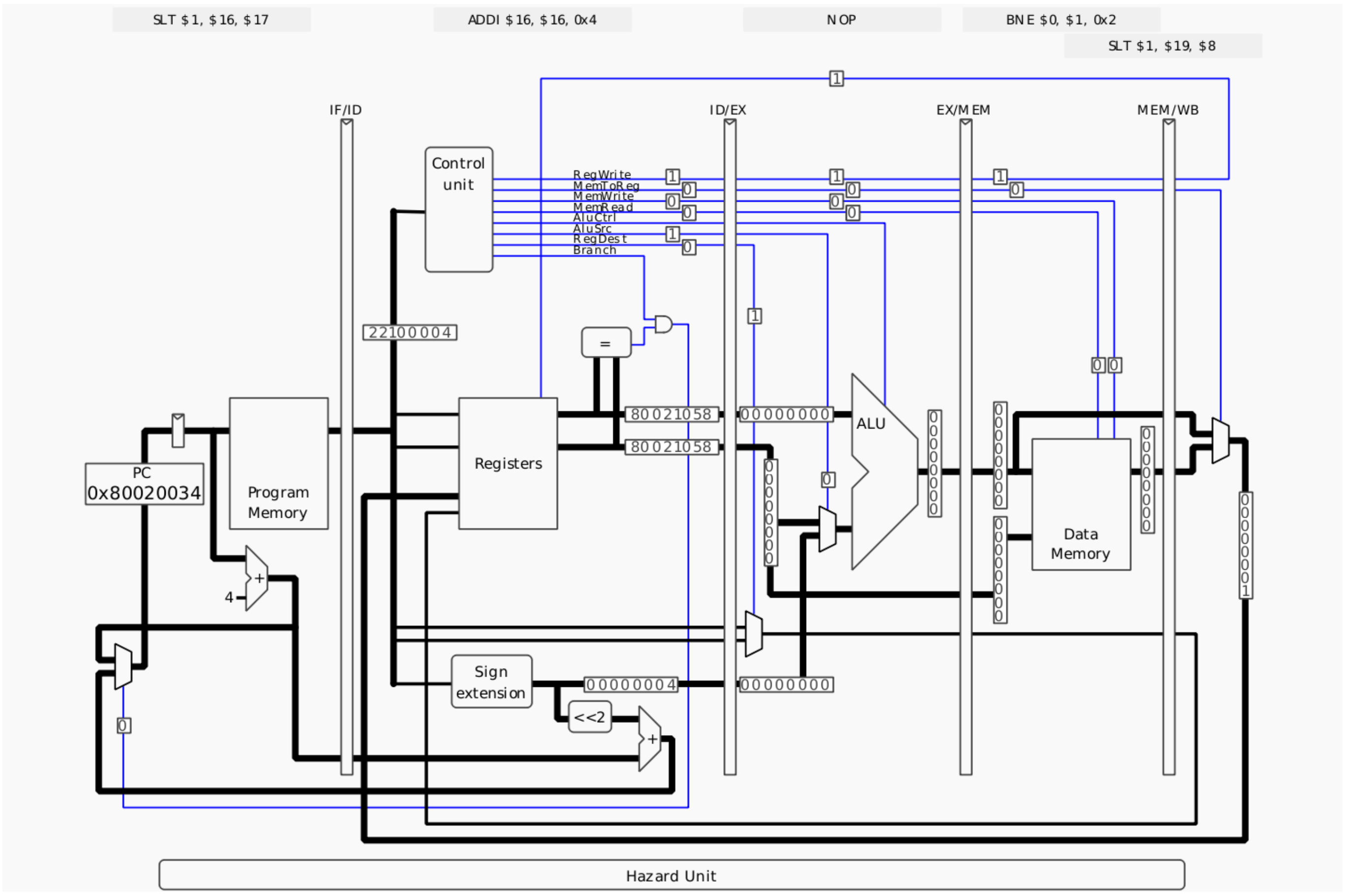


Figure 6.5. Pipelined CPU scheme with hazard unit and without cache.

Scheme visualization is not presented with all connections, for example connections to hazard unit were skipped. That is because scheme is already cluttered and adding connections that are suppose to be there for complete functionality of all instructions would result in too complex scheme. In future this can be solved by allowing user to switch between complicated and simplified scheme. At the moment, only simplified scheme is available because of simplicity and readability.

During implementation of scheme view the major limitation of Qt Graphics View Framework was discovered. It is using double precision floating data format for storing position of graphics items. But this data format has variable precision[18]. There are ranges of numbers that can't be represented. When such number is encountered then it is represented by closest possible representation. Doing multiplication between relatively small number and big number increases effect of this rounding error. When this is applied on group of relatively close by numbers then it translates them to something that could be described as clusters. This is caused by varying initial rounding error. This is problem because scene implemented for CPU view is relatively small. It has resolution just 720 times 540 pixels. But then it is possibly scaled to five times bigger resolution. Although scheme it self almost never positions anything on less than pixel width Qt class `QGraphicsSimpleTextItem` does exactly that and is used for adding labels all over scheme. Result of this is glitches in font alignment as illustrated in Picture 6.6. This is design problem in Qt Graphics View Framework that could be overcome by scaling items size and position translation instead of using view transformations. But that defeats purpose of this framework usage.

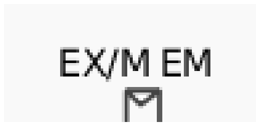


Figure 6.6. Graphical glitch with invalid font alignment caused by floating point precision error multiplication.

Scheme itself is implemented in source files *coreview.h* and *coreview.cpp*. But on top of that there is also whole directory containing additional classes. It is called *coreview*. And classes implemented there are described in following subsections.

6.3.1 Scheme Blocks

Scheme itself is build from separate blocks added to scene (thanks to being `QGraphicsItem` child). Blocks itself are in multiple cases used in scheme multiple times. That simplifies scheme creation. Having more general blocks also shortens code. Blocks implemented for the scheme are these:

- Adder: Used to visualize two values addition
- Alu: Visualizes ALU in scheme
- And: Logic AND gate visualized in distinctive shape style [19]
- Constant: Label defining constant in scheme
- Latch: Implements latch isolating pipeline stages.
- LogicBlock: General block for some complicated logic function. It is rendered as rectangle with rounded corners. Label is placed to it to describe function it implements.
- Multiplexer: Logic block that serve as a selector between multiple buses.
- ProgramCounter: Block displaying current value of program counter register
- Registers: Block illustrating general purpose registers

There is no in depth description for every block shape and its usage. It is not fundamental for this thesis. They are clearly visible in complete scheme.

One of the additional blocks present in scheme that was not mentioned yet is instruction view. This is rectangle with gray background presenting executed instruction. Text shown is generated by decoding instruction to instruction mnemonic as described in Section 5.7. It is used to show what instruction is being processed in what pipeline stage or in whole CPU when pipelining is not configured.

For completeness there was also need for junctions between connections that are not part of any visible block. Because of that there is class exactly for that. It has no graphical visualization. It's just technical block used for connections interconnection.

6.3.2 Signals and Buses

Where scheme contains considerable amount of blocks it contains much more connections between them. Wiring such connection line by line would require larger amount of time and would not be versatile enough when change has to be done to scheme. Instead of that algorithmic path was chosen. Connections are routed automatically between dynamic points called connectors. Those are moved relative to scheme blocks. That causes automatic route recalculation. That way most of the routes have to be only initialized with initial and terminating connector and everything else is done automatically.

Chosen routing algorithm is very simple. This thesis is not implementing any interactive routing tool and because of that some manual work to get path right is acceptable.

Whole algorithm is based on rectangular intersection. Every connector has orientation. This can be stretched to strait line with given orientation. When initial and terminal connectors are given then algorithm looks for intersection of such two lines and given point is used as additional point through connection is routed by. When there is no such point then no additional point is inserted and line is drawn directly between initial and terminal connector.

This algorithm of course only routes wires in L shape. But by adding additional axes to two initial ones it's possible to construct much more complicated paths. Same algorithm is applied but not directly between initial two start lines but between them and additional axes.

In the scheme there are two basic types of connections between various blocks. There are buses and signals. Primary difference is in source but also in width. All signals are generated from some of the logical blocks and are mostly just single boolean (with notable exception such as ALU operation). Buses carry data such as instruction or values through CPU and are mainly 32 bits with.

There are also buses that are not 32 bit in width. They are created by splitting bits from 32 bit bus. These narrower buses are visualized using narrower lines than whole 32 bits width bus is.

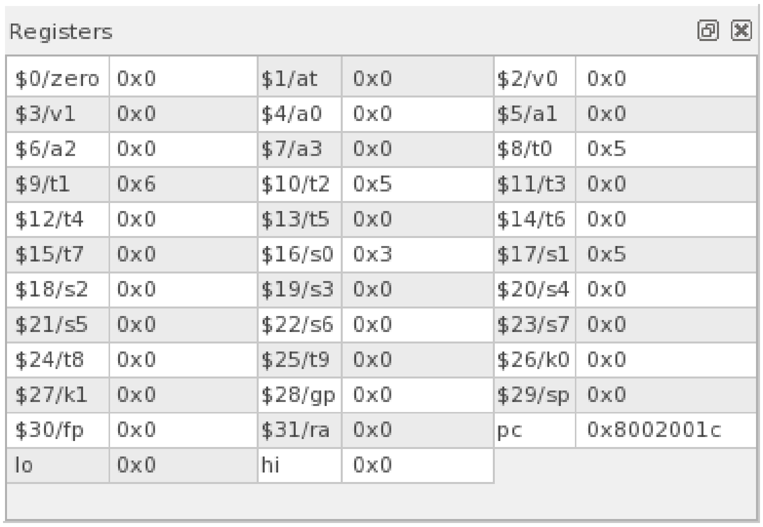
All signals are visualized by same line width and instead of black color used for buses it uses blue.

Buses and signals are implemented as a children on top of common parent implementing routing algorithm.

6.4 Registers View

Registers view is implemented as dockable window as described in Section 6.2. Its intent is to give user overview over all registers present in CPU. The simplest way to do so is table. Such table has two rows where left one contains name of register and right one its value. Lines alternate background color for better orientation. This form is optimal for higher then wider window sizes but is inefficient for wider then higher ones. Because of that when window is resized to be wider then higher then its content is recomposed to rows where multiple columns are presented just to better fill in horizontal space.

New Qt layout and widget was implemented for this specific view to fulfill all described behaviour. In code it is called `StaticTable`.



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	0x5
$9/t1	0x6	$10/t2	0x5	$11/t3	0x0
$12/t4	0x0	$13/t5	0x0	$14/t6	0x0
$15/t7	0x0	$16/s0	0x3	$17/s1	0x5
$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	0x0
$30/fp	0x0	$31/ra	0x0	pc	0x8002001c
lo	0x0	hi	0x0		

Figure 6.7. Window with registers view that is wider than higher.

6.5 Program and Data Memory View

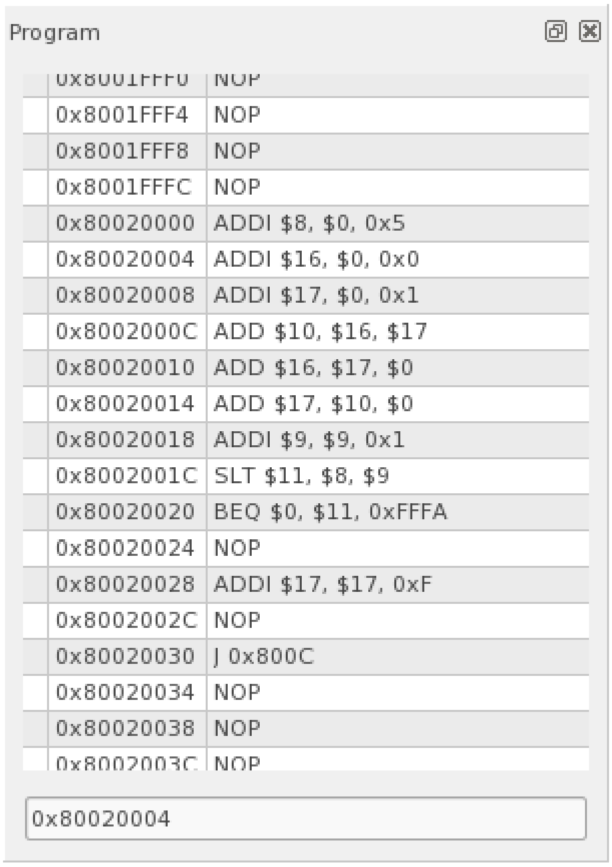
Program and data memory views are two separate dockable windows that are in reality mostly implemented the same way. Primary content is scrollable view of memory. It cannot be as simple as static table in scroll view because whole 32 bits addressable space is simulated and allocating widget for every word in memory would be too memory demanding. Instead dynamic loading is used. Widget containing data is always higher than needed and clipped using scroll widget. When one of the ends is too close to border of scroll area (too close to visible area) then new data are loaded, original widget is moved to center position and whole content is shifted. This way an sensation of infinite scrollable memory view is achieved.

But scrolling is just too slow and too tedious task if jump to far away address is needed. Because of that additional control element was introduced. Under scroll area is input field allowing user to enter exact address. Memory view is then centered on such address.

Format used for presenting data it self is same as in case of registers view (Section 6.4). The `StaticTable` is used.

Although in implementation program and data memory views are almost same they differ in what data they present. Where data memory view shows words in hexadecimal format program memory view uses decoding to instruction mnemonic to show instructions in their MIPS assembler representation. It is needless to say that they both show same data (same memory).

Program memory view is implemented in source files *programdock.h* and *programdock.cpp*. Data memory view is implemented in source files *memorydock.h* and *memorydock.cpp*. They both share common ancestor implemented in source files *memoryview.h* and *memoryview.cpp*.



Program

0x8001FFF0	NOP
0x8001FFF4	NOP
0x8001FFF8	NOP
0x8001FFFC	NOP
0x80020000	ADDI $8, $0, 0x5
0x80020004	ADDI $16, $0, 0x0
0x80020008	ADDI $17, $0, 0x1
0x8002000C	ADD $10, $16, $17
0x80020010	ADD $16, $17, $0
0x80020014	ADD $17, $10, $0
0x80020018	ADDI $9, $9, 0x1
0x8002001C	SLT $11, $8, $9
0x80020020	BEQ $0, $11, 0xFFFA
0x80020024	NOP
0x80020028	ADDI $17, $17, 0xF
0x8002002C	NOP
0x80020030	J 0x800C
0x80020034	NOP
0x80020038	NOP
0x8002003C	NOP

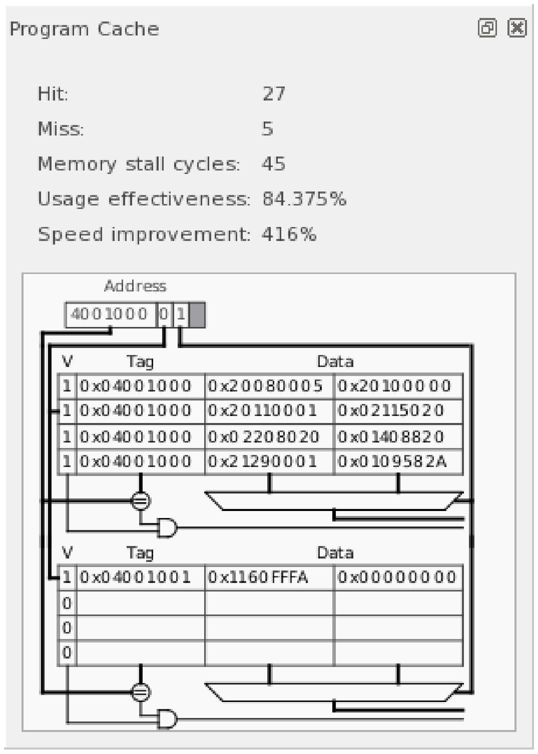
0x80020004

Figure 6.8. Window with program memory view.

6.6 Program and Data Cache View

Program and data cache views are two separate dockable windows. But they are instances of the same class. There are two primary parts of this window. Part where cache statistics are displayed and part where cache content is visualized.

Cache statics are simple labels with values updated in them by signals from simulator. What exactly they mean is described more in depth in Section 5.3.1.



Hit:	27
Miss:	5
Memory stall cycles:	45
Usage effectiveness:	84.375%
Speed improvement:	416%

V	Tag	Data	
1	0x04001000	0x20080005	0x20100000
1	0x04001000	0x20110001	0x02115020
1	0x04001000	0x02208020	0x01408820
1	0x04001000	0x21290001	0x0109582A

V	Tag	Data	
1	0x04001001	0x1160FFFA	0x00000000
0			
0			
0			

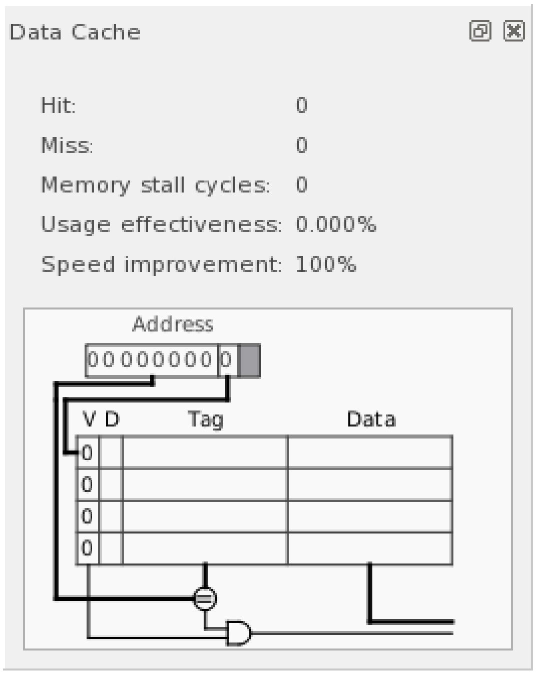
Figure 6.9. Program cache view configured with default cache.

Cache content visualization is based on same approach as CPU scheme. It uses Qt Graphics View Framework. It uses same code for scene view which behaviour was described in Section 6.3.

Graphics scene it self consist of repeating block and some additional graphical elements. Every block visualizes separate cache block and number of them is given by cache associativity. They are aligned in sequence under each other. Thanks to that it is easy to add signal wires without complicated algorithm. Every block draws all wires that has to be displayed and thanks to their alignment they connect each other automatically.

Content of cache itself is filled from slots. When scene is initialized it requires simulator's cache class instance and during initialization it also connects those slots to signals from cache.

Program and data cache views are implemented in source files *cacheview.h* and *cacheview.cpp*.


Data Cache

Hit:	0
Miss:	0
Memory stall cycles:	0
Usage effectiveness:	0.000%
Speed improvement:	100%

Figure 6.10. Data cache view configured with directly mapped cache and before any usage.

Chapter 7
Conclusion

Graphical MIPS CPU simulator was developed as part of this thesis. It was developed with consideration of current simulator usage and designed to be intuitive. It is able to simulate considerable portion of MIPS ISA including both single-cycle and pipelined CPU. Simulator in current state can fulfill all roles which are provided by MipsIt with exception of memory mapped I/O (but that wasn't goal of this thesis).

Simulator was implemented with testing framework. All supported instructions are automatically tested using unit tests. GUI and complete simulation was tested by hand with MIPS assembly programs from Computer Architectures tutorials.

Development of simulator is not yet complete. There are possible improvements outside of this thesis such as to add memory mapped I/O or implementation of excluded instructions. It would be also beneficial to extend feedback in GUI such as to add highlighting of changed values. Moreover, it will be important to gather feedback from the students using the simulator during the APO course and subsequently address their concerns and suggestions as well as reports of previously undiscovered problems in future updates of the simulator.

References

[1] Inc. MIPS Technologies. *MIPS32™ Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture*. 2001.

[2] Thomas Finley. *Two's Complement*. 2000. `https://www.cs.cornell.edu/%7Etomf/notes/cps104/twoscomp.html`. [Online; accessed 22-April-2018] .

[3] Wikipedia, The Free Encyclopedia. *Sign extension*. 2018. `https://en.wikipedia.org/wiki/Sign_extension`. [Online; accessed 21-April-2018] .

[4] Inc. MIPS Technologies. *MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set*. 2001.

[5] Wikipedia, The Free Encyclopedia. *Boolean algebra*. 2018. `https://en.wikipedia.org/wiki/Boolean_algebra`. [Online; accessed 20-April-2018] .

[6] Wikipedia, The Free Encyclopedia. *Arithmetic shift*. 2018. `https://en.wikipedia.org/wiki/Arithmetic_shift`. [Online; accessed 20-April-2018] .

[7] Wikibooks, Pen books for open world. *MIPS Assembly/Pseudoinstructions*. 2018. `https://en.wikibooks.org/wiki/MIPS_Assembly/Pseudoinstructions`. [Online; accessed 20-April-2018] .

[8] Patterson, D. A. John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 2011 .

[9] CTU in Praguecourse. *Computer Architecture*. `https://cw.fel.cvut.cz/wiki/courses/b35apo/start`. [Online; accessed 20-May-2018] .

[10] Pavel Píša. *Computer Architecture: course on CTU in Prague*. 2014.

[11] Wikipedia, The Free Encyclopedia. *Bubble sort*. 2018. `https://en.wikipedia.org/wiki/Bubble_sort`. [Online; accessed 17-April-2018] .

[12] Mats Brorsson. *MipsIt—A Simulation and Development Environment Using Animation for Computer Architecture Education*. 2002.

[13] Wikipedia, The Free Encyclopedia. *SREC (file format)*. `https://en.wikipedia.org/wiki/SREC_(file_format)`. [Online; accessed 23-April-2018] .

[14] Wikipedia, The Free Encyclopedia. *Tree (data structure)*. 2018. `https://en.wikipedia.org/wiki/Tree_(data_structure)`. [Online; accessed 21-April-2018] .

[15] Joseph Koshy. *libelf by Example*. 2012.

[16] Qt Documentation. *Signals and Slots*. `https://doc.qt.io/qt-5/signalsandslots.html`. [Online; accessed 8-May-2018] .

[17] Qt Documentation. *Graphics View Framework*.
`https://doc.qt.io/qt-5/graphicsview.html`. [Online; accessed 20-May-2018] .

[18] Institute of Electrical, and Electronics Engineers. *EEE Std 754 — IEEE Standard for Floating-Point Arithmetic*. 2008.
`http://standards.ieee.org/findstds/standard/754-2008.html`.

[19] Institute of Electrical, and Electronics Engineers. *IEEE/ANSI 91/91a — IEEE Standard Graphic Symbols for Logic Functions*. 1991.
`https://standards.ieee.org/findstds/standard/91-91a-1991.html`.

Appendix A
Glossary

ALU	Arithmetic Logic Unit
API	Application Programming Interface
C	C Programing Language
C++	C++ Programing Language
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CTU	Czech Technical University
ELF	Executable and Linkable Format
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
GCC	GNU Compiler Collection
GNU	GNU's Not Unix operation system
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IPC	Instruction Per Cycle
ISA	Instruction Set Architecture
MIPS	Microprocessor without Interlocked Pipeline Stages
NOP	No Operation computer instruction
PC	Personal Computer
Qt	Cross-platform application framework and widget toolkit
RISC	Reduced Instruction Set Computer
SVG	Scalable Vector Graphics
VHDL	VHSIC (Very Hight Speed Integrated Circuit) Hardware Description Language
Wine	Wine is not an emulator (Windows compatibility layer)
XML	Extensible Markup Language

Appendix B
Content of attached CD

CD attached with this thesis contains directory `QtMips` with source codes of developed simulator. Another directory `Thesis` contains sources for this thesis text and thesis itself in PDF format.

Current version of QtMips is provided online: `https://github.com/Cynerd/QtMips`