EduMIPS64 Documentation

Release 1.4.1

Andrea Spadaccini (and the EduMIPS64 development team)

Jun 07, 2026

CONTENTS

EduMIPS64 is a MIPS64 Instruction Set Architecture (ISA) simulator. It is designed to be used to execute small programs that use the subset of the MIPS64 ISA implemented by the simulator, allowing the user to see how instructions behave in the pipeline, how stalls are handled by the CPU, the status of registers and memory and much more. It is both a simulator and a visual debugger.

The website for the project is http://www.edumips.org, and the code is hosted at http://github.com/EduMIPS64/edumips64. If you find any bugs, or have any suggestion for improving the simulator, please open an issue on github or send an email at bugs@edumips.org.

EduMIPS64 is developed by a group of students of the University of Catania (Italy), and started as a clone of WinMIPS64, even if now there are lots of differences between the two simulators.

This manual will introduce you to EduMIPS64, and will cover some details on how to use it.

This manual describes EduMIPS64 version 1.4.1.

This manual is split into two parts. The first part is independent from the user interface in use and covers the source file format, the supported instruction set, the Floating Point Unit and a set of example programs. The second part documents the user interfaces: a chapter for the desktop (Swing) application, which also includes the command line options of the JAR, a chapter for the interactive command line shell that the JAR exposes in headless mode, and a chapter for the web frontend.

When this manual is opened from inside the running application, only the chapter that is relevant for the active user interface is shown. The full manual (with both UI chapters) is available on Read the Docs and as a PDF.

CHAPTER

ONE

SOURCE FILES FORMAT

EduMIPS64 tries to follow the conventions used in other MIPS64 and DLX simulators, so that old time users will not be confused by its syntax.

There are two sections in a source file, the *data* section and the *code* section, introduced respectively by the *.data* and the *.code* directives. In the following listing you can see a very basic EduMIPS64 program:

```
; This is a comment
        .data
label:  .word   15      ; This is an inline comment

        .code
        daddi   r1, r0, 0
        syscall 0
```

To distinguish the various parts of each source code line, any combination of spaces and tabs can be used, as the parser ignores multiple spaces and only detects whitespaces to separate tokens.

Comments can be specified using the ";" character, everything that follows that character will be ignored. So a comment can be used "inline" (after the directive) or on a row by itself.

Labels can be used in the code to reference a memory cell or an instruction. They are case insensitive. Only a label for each source code line can be used. The label can be specified one or more rows above the effective data declaration or instruction, provided that there's nothing, except for comments and empty lines, between the label and the declaration.

Multiple labels can refer to the same address. This is useful when two or more labels are placed on consecutive lines before an instruction, for example:

```
        .code
        daddi   r1, r0, 1
loop_end:
loop_begin:
        daddi   r1, r1, -1
```

In this case, both *loop_end* and *loop_begin* will point to the same instruction. Each label must still have a unique name; using the same label name twice will result in a parser error.

1.1 Memory limits

EduMIPS64 has a fixed memory size for both data (the *.data* section, capped at 640 kB – i.e., 80000 64-bit values) and instructions (the *.code* section, capped at 128 kB – i.e., 32000 instructions, each occupying 32 bits).

These limits are hardcoded in the simulator.

1.2 The *.data* section

The *data* section contains commands that specify how the memory must be filled before program execution starts. The general form of a *.data* command is:

```
[label:] .datatype value1 [, value2 [, ...]]
```

EduMIPS64 supports different data types, that are described in the following table.

Type	Directive	Bits required
Byte	*.byte*	8
Half word	*.word16*	16
Word	*.word32*	32
Double Word	*.word* or *.word64*	64

Please note that a double word can be introduced either by the *.word* directive or by the *.word64* directive.

All the data types are interpreted as signed. This means that integer literals in the *.data* section must be between -2^(n-1) and 2^(n-1) - 1 (inclusive).

There is a big difference between declaring a list of data elements using a single directive or by using multiple directives of the same type. EduMIPS64 starts writing from the next 64-bit double word as soon as it finds a datatype identifier, so the first *.byte* statement in the following listing will put the numbers 1, 2, 3 and 4 in the space of 4 bytes, taking 32 bits, while code in the next four rows will put each number in a different memory cell, occupying 32 bytes:

```
.data
.byte    1, 2, 3, 4
.byte    1
.byte    2
.byte    3
.byte    4
```

In the following table, the memory is represented using byte-sized cells and each row is 64 bits wide. The address on the left side of each row of the table refers to the right-most memory cell, that has the lowest address of the eight cells in each line.

0	0	0	0	0	4	3	2	1
8	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	2
24	0	0	0	0	0	0	0	3
32	0	0	0	0	0	0	0	4

There are some special directives that need to be discussed: *.space*, *.ascii* and *.asciiz*.

The *.space* directive is used to leave some free space in memory. It accepts as a parameter an integer, that indicates the number of bytes that must be left empty. It is handy when you must save some space in memory for the results of your computations.

The *.ascii* directive accepts strings containing any of the ASCII characters, and some special C-like escaping sequences, that are described in the following table, and puts those strings in memory.

Escaping sequence	Meaning	ASCII code
\0	Null byte	0
\t	Horizontal tabulation	9
\n	Newline character	10
\"	Literal quote character	34
\	Literal backslash character	92

The *.asciiz* directive behaves exactly like the *.ascii* command, with the difference that it automatically ends the string with a null byte.

1.3 The *.code* section

The *code* section contains commands that specify how the memory must be filled when the program will start. The general form of a *.code* command is:

```
[label:] instruction [param1 [, param2 [, param3]]]
```

The *code* section can be specified with the *.text* alias.

The number and the type of parameters depends on the instruction itself.

Instructions can take three types of parameters:

- *Registers* a register parameter is indicated by an uppercase or lowercase "r", or a $, followed by the number of the register (between 0 and 31), as in "r4", "R4" or "$4";
- *Immediate values* an immediate value can be a number or a label; the number can be specified in base 10, base 16 or base 2: base 10 numbers are simply inserted by writing the number, base 16 numbers are inserted by putting before the number the prefix "0x", and base 2 numbers are inserted by putting before the number the prefix "0b". Immediate values can be preceded by the # character.
- *Address* an address is composed by an immediate value followed by a register name enclosed in brackets. The value of the register will be used as base, the value of the immediate will be the offset.

The size of immediate values is limited by the number of bits that are available in the bit encoding of the instruction.

When 16-bit immediates can be used, for example in ALU I-Type instructions, it's also possible to use as an immediate value a memory label. The assembler will put as immediate value the memory address the label points to.

1.3.1 Label offset arithmetic

In the address form of a parameter (e.g. the offset of a load or store instruction) the immediate value can be a simple expression built from memory labels and numeric literals combined with the + and - operators. A leading + or - is also accepted and whitespace is allowed between the operands and the operators. Each operand is either a numeric literal (in base 10, 16 or 2) or a memory label defined in the `.data` section; labels are replaced by their address before the expression is evaluated.

For example, given the data definitions `data1: .word 42` and `data2: .word 43`, the following forms are all accepted:

```
lw r1, data1+8(r0)       ; load from data1 offset by 8 bytes
lw r1, data1-8(r0)       ; load from data1 offset by -8 bytes
lw r1, 0+data1(r0)       ; equivalent to data1(r0)
lw r1, data2-data1(r0)   ; difference between two label addresses
lw r1, data1-8+16(r0)    ; chains of + and - are supported
```

Expressions with an empty operand (for example `data1+` or `data1++0`) are rejected as malformed, and an unknown label anywhere in the expression produces the usual "label not found" error.

You can use standard MIPS assembly aliases to address the first 32 registers, appending the alias to one of the standard register prefixes like "r", "$" and "R". See the next table.

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

1.4 The *#include* command

Source files can contain the *#include filename* command, which has the effect of putting in place of the command row the content of the file *filename*. It is useful if you want to include external routines, and it comes with a loop-detection algorithm that will warn you if you try to do something like "*#include A.s*" in file *B.s* and "*#include B.s*" in file *A.s*.

CHAPTER

TWO

THE INSTRUCTION SET

In this section we will describe the subset of the MIPS64 instruction set that EduMIPS64 recognizes. We can operate two different taxonomic classification: one based on the functionality of the instructions and one based on the type of the parameters of the instructions.

The first classification divides instruction into three categories: ALU instructions, Load/Store instructions, Flow control instructions. The next three subsections will describe each category and every instruction that belongs to those categories.

The fourth subsection will describe instructions that do not fit in any of the three categories.

2.1 ALU Instructions

The Arithmetic Logic Unit (in short, ALU) is a part of the execution unit of a CPU, that has the duty of doing arithmetical and logic operations. So in the ALU instructions group we will find those instructions that do this kind of operations.

ALU Instructions can be divided in two groups: *R-Type* and *I-Type*.

Four of those instructions make use of two special registers: LO and HI. They are internal CPU registers, whose value can be accessed through the *MFLO* and *MFHI* instructions.

> **Note**
>
> Several MIPS instructions have a name ending in `U` (for example `ADDU`, `ADDIU`, `DADDU`, `DADDIU`, `DSUBU`, `SUBU`). Despite the suggestive suffix, these instructions **do not** treat their operands as unsigned: the result is sign-extended exactly like in the non-`U` variant. The only difference is that no Integer Overflow exception is raised. This naming is a well-known historical misnomer in the MIPS architecture.

Here's the list of R-Type ALU Instructions.

- *AND rd, rs, rt*

  Executes a bitwise AND between rs and rt, and puts the result into rd.

- *ADD rd, rs, rt*

  Sums the content of 32-bits registers rs and rt, considering them as signed values, and puts the result into rd. If an overflow occurs then trap.

- *ADDU rd, rs, rt*

  Sums the content of 32-bits registers rs and rt, and puts the result into rd. No integer overflow occurs under any circumstances.

- *DADD rd, rs, rt*

Sums the content of 64-bits registers rs and rt, considering them as signed values, and puts the result into rd. If an overflow occurs then trap.

- *DADDU rd, rs, rt*

  Sums the content of 64-bits registers rs and rt, and puts the result into rd. No integer overflow occurs under any circumstances.

- *DDIV rs, rt* OR *DDIV rd, rs, rt*

  Executes the division between 64-bits registers rs and rt.

  Two-operand form (DDIV rs, rt): Puts the 64-bits quotient in LO and the 64-bits remainder in HI.

  Three-operand form (DDIV rd, rs, rt): Puts the 64-bits quotient in register rd. This is the MIPS64 Release 6 version.

- *DDIVU rs, rt*

  Executes the division between 64-bits registers rs and rt, considering them as unsigned values and putting the 64-bits quotient in LO and the 64-bits remainder in HI.

- *DIV rs, rt*

  Executes the division between 32-bits registers rs and rt, putting the 32-bits quotient in LO and the 32-bits remainder in HI.

- *DIVU rs, rt*

  Executes the division between 32-bits registers rs and rt, considering them as unsigned values and putting the 32-bits quotient in LO and the 32-bits remainder in HI.

- *DMUHU rd, rs, rt*

  Executes the multiplication between 64-bits registers rs and rt, considering them as unsigned values and putting the high-order 64-bits doubleword of the result into register rd.

- *DMOD rd, rs, rt*

  Computes the signed modulo (remainder) of 64-bits register rs divided by 64-bits register rt, and puts the result into register rd. This is the MIPS64 Release 6 version. For the legacy form that stores remainder in HI, use DDIV rs, rt followed by MFHI.

- *DMUL rd, rs, rt*

  Executes the multiplication between 64-bits registers rs and rt, putting the low-order 64-bits doubleword of the result into register rd. Note: This is the MIPS64 Release 6 version. To store the result in LO/HI registers, use DMULT instead.

- *DMULT rs, rt*

  Executes the multiplication between 64-bits registers rs and rt, putting the low-order 64-bits doubleword of the result into special register LO and the high-order 64-bits doubleword of the result into special register HI.

- *DMULU rd, rs, rt*

  Executes the multiplication between 64-bits registers rs and rt, considering them as unsigned values and putting the low-order 64-bits doubleword of the result into register rd.

- *DMULTU rs, rt*

  Executes the multiplication between 64-bits registers rs and rt, considering them as unsigned values and putting the low-order 64-bits doubleword of the result into special register LO and the high-order 64-bits doubleword of the result into special register HI.

- *DSLL rd, rt, sa*

  Does a left shift of 64-bits register rt, by the amount specified in the immediate (positive) value sa, and puts the result into 64-bits register rd. Empty bits are padded with zeros.

- *DSLLV rd, rt, rs*

  Does a left shift of 64-bits register rt, by the amount specified in low-order 6-bits of rs treated as unsigned value, and puts the result into 64-bits register rd. Empty bits are padded with zeros.

- *DSRA rd, rt, sa*

  Does a right shift of 64-bits register rt, by the amount specified in the immediate (positive) value sa, and puts the result into 64-bits register rd. Empty bits are padded with zeros if the leftmost bit of rt is zero, otherwise they are padded with ones.

- *DSRAV rd, rt, rs*

  Does a right shift of 64-bits register rt, by the amount specified in low-order 6-bits of rs treated as unsigned value, and puts the result into 64-bits register rd. Empty bits are padded with zeros if the leftmost bit of rt is zero, otherwise they are padded with ones.

- *DSRL rd, rs, sa*

  Does a right shift of 64-bits register rs, by the amount specified in the immediate (positive) value sa, and puts the result into 64-bits register rd. Empty bits are padded with zeros.

- *DSRLV rd, rt, rs*

  Does a right shift of 64-bits register rt, by the amount specified in low-order 6-bits of rs treated as unsigned value, and puts the result into 64-bits register rd. Empty bits are padded with zeros.

- *DSUB rd, rs, rt*

  Subtracts the value of 64-bits register rt from 64-bits register rs, considering them as signed values, and puts the result in rd. If an overflow occurs then trap.

- *DSUBU rd, rs, rt*

  Subtracts the value of 64-bits register rt from 64-bits register rs, and puts the result in rd. No integer overflow occurs under any circumstances.

- *MFLO rd*

  Moves the content of the special register LO into rd.

- *MFHI rd*

  Moves the content of the special register HI into rd.

- *MOVN rd, rs, rt*

  If rt is different from zero, then moves the content of rs into rd.

- *MOVZ rd, rs, rt*

  If rt is equal to zero, then moves the content of rs into rd.

- *MULT rs, rt*

  Executes the multiplication between 32-bits registers rs and rt, putting the low-order 32-bits word of the result into special register LO and the high-order 32-bits word of the result into special register HI.

- *MULTU rs, rt*

Executes the multiplication between 32-bits registers rs and rt, considering them as unsigned values and putting the low-order 32-bits word of the result into special register LO and the high-order 32-bits word of the result into special register HI.

- *OR rd, rs, rt*

  Executes a bitwise OR between rs and rt, and puts the result into rd.

- *SLL rd, rt, sa*

  Does a left shift of 32-bits register rt, by the amount specified in the immediate (positive) value sa, and puts the result into 32-bits register rd. Empty bits are padded with zeros.

- *SLLV rd, rt, rs*

  Does a left shift of 32-bits register rt, by the amount specified in low-order 5-bits of rs treated as unsigned value, and puts the result into 32-bits register rd. Empty bits are padded with zeros.

- *SRA rd, rt, sa*

  Does a right shift of 32-bits register rt, by the amount specified in the immediate (positive) value sa, and puts the result into 32-bits register rd. Empty bits are padded with zeros if the leftmost bit of rt is zero, otherwise they are padded with ones.

- *SRAV rd, rt, rs*

  Does a right shift of 32-bits register rt, by the amount specified in low-order 5-bits of rs treated as unsigned value, and puts the result into 32-bits register rd. Empty bits are padded with zeros if the leftmost bit of rt is zero, otherwise they are padded with ones.

- *SRL rd, rs, sa*

  Does a right shift of 32-bits register rs, by the amount specified in the immediate (positive) value sa, and puts the result into 32-bits register rd. Empty bits are padded with zeros.

- *SRLV rd, rt, rs*

  Does a right shift of 32-bits register rt, by the amount specified in low-order 5-bits of rs treated as unsigned value, and puts the result into 32-bits register rd. Empty bits are padded with zeros.

- *SUB rd, rs, rt*

  Subtracts the value of 32-bits register rt from 32-bits register rs, considering them as signed values, and puts the result in rd. If an overflow occurs then trap.

- *SUBU rd, rs, rt*

  Subtracts the value of 32-bits register rt from 32-bits register rs, and puts the result in rd. No integer overflow occurs under any circumstances.

- *SLT rd, rs, rt*

  Sets the value of rd to 1 if the value of rs is less than the value of rt, otherwise sets it to 0. This instruction performs a signed comparison.

- *SLTU rd, rs, rt*

  Sets the value of rd to 1 if the value of rs is less than the value of rt, otherwise sets it to 0. This instruction performs an unsigned comparison.

- *XOR rd, rs, rt*

  Executes a bitwise exclusive OR (XOR) between rs and rt, and puts the result into rd.

Here's the list of I-Type ALU Instructions.

- *ADDI rt, rs, immediate*

  Executes the sum between 32-bits register rs and the immediate value, putting the result in rt. This instruction considers rs and the immediate value as signed values. If an overflow occurs then trap.

- *ADDIU rt, rs, immediate*

  Executes the sum between 32-bits register rs and the immediate value, putting the result in rt. No integer overflow occurs under any circumstances.

- *ANDI rt, rs, immediate*

  Executes the bitwise AND between rs and the immediate value, putting the result in rt.

- *DADDI rt, rs, immediate*

  Executes the sum between 64-bits register rs and the immediate value, putting the result in rt. This instruction considers rs and the immediate value as signed values. If an overflow occurs then trap.

- *DADDIU rt, rs, immediate*

  Executes the sum between 64-bits register rs and the immediate value, putting the result in rt. No integer overflow occurs under any circumstances.

- *DADDUI rt, rs, immediate*

  Executes the sum between 64-bits register rs and the immediate value, putting the result in rt. No integer overflow occurs under any circumstances.

  Warning

  `DADDUI` is not a standard MIPS64 instruction (it is part of the WinMIPS64 instruction set, accepted by EduMIPS64 for backward compatibility). The parser emits a warning when it is used; prefer `DADDIU` instead.

- *LUI rt, immediate*

  Loads the constant defined in the immediate value in the upper half (16 bit) of the lower 32 bits of rt, sign-extending the upper 32 bits of the register.

- *ORI rt, rs, immediate*

  Executes the bitwise OR between rs and the immediate value, putting the result in rt.

- *SLTI rt, rs, immediate*

  Sets the value of rt to 1 if the value of rs is less than the value of the immediate, otherwise sets it to 0. This instruction performs a signed comparison.

- *SLTIU rt, rs, immediate*

  Sets the value of rt to 1 if the value of rs is less than the value of the immediate, otherwise sets it to 0. This instruction performs an unsigned comparison.

- *XORI rt, rs, immediate*

  Executes a bitwise exclusive OR (XOR) between rs and the immediate value, and puts the result into rt.

2.2 Load/Store instructions

This category contains all the instructions that operate transfers between registers and the memory. All of these instructions are in the form:

```
[label:] instruction rt, offset(base)
```

Where rt is the source or destination register, depending if we are using a store or a load instruction; offset is a label or an immediate value and base is a register. The address is obtained by adding to the value of the register *base* the immediate value *offset*.

The address specified must be aligned according to the data type that is treated. Load instructions ending with "U" treat the content of the register rt as an unsigned value.

List of load instructions:

- *LB rt, offset(base)*

  Loads the content of the memory cell at address specified by offset and base in register rt, treating it as a signed byte.

- *LBU rt, offset(base)*

  Loads the content of the memory cell at address specified by offset and base in register rt, treating it as an unsigned byte.

- *LD rt, offset(base)*

  Loads the content of the memory cell at address specified by offset and base in register rt, treating it as a double word.

- *LH rt, offset(base)*

  Loads the content of the memory cell at address specified by offset and base in register rt, treating it as a signed half word.

- *LHU rt, offset(base)*

  Loads the content of the memory cell at address specified by offset and base in register rt, treating it as an unsigned half word.

- *LW rt, offset(base)*

  Loads the content of the memory cell at address specified by offset and base in register rt, treating it as a signed word.

- *LWU rt, offset(base)*

  Loads the content of the memory cell at address specified by offset and base in register rt, treating it as an unsigned word.

List of store instructions:

- *SB rt, offset(base)*

  Stores the content of register rt in the memory cell specified by offset and base, treating it as a byte.

- *SD rt, offset(base)*

  Stores the content of register rt in the memory cell specified by offset and base, treating it as a double word.

- *SH rt, offset(base)*

  Stores the content of register rt in the memory cell specified by offset and base, treating it as a half word.

- *SW rt, offset(base)*

  Stores the content of register rt in the memory cell specified by offset and base, treating it as a word.

2.3 Flow control instructions

Flow control instructions are used to alter the order of instructions that are fetched by the CPU. We can make a distinction between these instructions: R-Type, I-Type and J-Type.

EduMIPS64 uses a **branch-not-taken** prediction strategy. When a conditional branch instruction is encountered, the processor assumes the branch will not be taken and continues fetching the next sequential instructions. If the branch is actually taken, the speculatively fetched instructions are squashed from the pipeline and execution resumes from the branch target.

Those instructions effectively executes the jump in the ID stage, so often an useless fetch is executed. In this case, two instructions are removed from the pipeline, and the branch taken stalls counter is incremented by two units.

List of R-Type flow control instructions:

- *JALR rs*

  Puts the content of rs into the program counter, and puts into R31 the address of the instruction that follows the JALR instruction, the return value.

- *JR rs*

  Puts the content of rs into the program counter.

List of I-Type flow control instructions:

- *B offset*

  Unconditionally jumps to offset

- *BEQ rs, rt, offset*

  Jumps to offset if rs is equal to rt.

- *BEQZ rs, offset*

  Jumps to offset if rs is equal to zero.

  > **Warning**
  >
  > `BEQZ` is not a standard MIPS64 instruction (it is part of the WinMIPS64 instruction set, accepted by EduMIPS64 for backward compatibility). The parser emits a warning when it is used; prefer `BEQ rs, r0, offset` instead.

- *BGEZ rs, offset*

  If rs is greater than or equal to zero, does a PC-relative jump to offset.

- *BNE rs, rt, offset*

  Jumps to offset if rs is not equal to rt.

- *BNEZ rs, offset*

  Jumps to offset if rs is not equal to zero.

  > **Warning**
  >
  > `BNEZ` is not a standard MIPS64 instruction (it is part of the WinMIPS64 instruction set, accepted by EduMIPS64 for backward compatibility). The parser emits a warning when it is used; prefer `BNE rs, r0, offset` instead.

List of J-Type flow control instructions:

- *J target*

  Puts the immediate value target into the program counter.

- *JAL target*

  Puts the immediate value target into the program counter, and puts into R31 the address of the instruction that follows the JAL instruction, the return value.

2.4 The *SYSCALL* instruction

The SYSCALL instruction offers to the programmer an operating-system-like interface, making available six different system calls.

System calls expect that the address of their parameters is stored in register R14 ($t6), and will put their return value in register R1 ($at).

System calls follow as much as possible the POSIX convention.

2.4.1 *SYSCALL 0 - exit()*

SYSCALL 0 does not expect any parameter, nor it returns anything. It simply stops the simulator.

Note that if the simulator does not find SYSCALL 0 in the source code, or any of its equivalents (HALT - TRAP 0), it will be added automatically at the end of the source.

2.4.2 *SYSCALL 1 - open()*

The SYSCALL 1 expects two parameters: a zero-terminated string that indicates the pathname of the file that must be opened, and a double word containing an integer that indicates the flags that must be used to specify how to open the file.

This integer must be built summing the flags that you want to use, choosing them from the following list:

- *O_RDONLY (0x01)* Opens the file in read only mode;
- *O_WRONLY (0x02)* Opens the file in write only mode;
- *O_RDWR (0x03)* Opens the file in read/write mode;
- *O_CREAT (0x04)* Creates the file if it does not exist;
- *O_APPEND (0x08)* In write mode, appends written text at the end of the file;
- *O_TRUNC (0x10)* In write mode, deletes the content of the file as soon as it is opened.

It is mandatory to specify one of the first three modes. The fifth and the sixth modes are mutually exclusive: you cannot specify O_APPEND if you specify O_TRUNC (and vice versa).

You can specify a combination of modes by simply adding the integer values of those flags. For instance, if you want to open a file in write only mode and append the written text to the end of file, you should specify the mode 2 + 8 = 10.

The return value of the system call is the new file descriptor associated with the file, that can be further used with the other system calls. If there is an error, the return value will be -1.

2.4.3 *SYSCALL 2 - close()*

SYSCALL 2 expects only one parameter, the file descriptor of the file that is closed.

If the operation ends successfully, SYSCALL 2 will return 0, otherwise it will return -1. Possible causes of failure are the attempt to close a non-existent file descriptor or the attempt to close file descriptors 0, 1 or 2, that are associated respectively to standard input, standard output and standard error.

2.4.4 *SYSCALL 3 - read()*

SYSCALL 3 expects three parameters: the file descriptor to read from, the address where the read data must be put into, the number of bytes to read.

If the first parameter is 0, the simulator will prompt the user for an input, via an input dialog. If the length of the input is greater than the number of bytes that have to be read, the simulator will show again the message dialog.

It returns the number of bytes that have effectively been read, or -1 if the read operation fails. Possible causes of failure are the attempt to read from a non-existent file descriptor, the attempt to read from file descriptors 1 (standard output) or 2 (standard error) or the attempt to read from a write-only file descriptor.

2.4.5 *SYSCALL 4 - write()*

SYSCALL 4 expects three parameters: the file descriptor to write to, the address where the data must be read from, the number of bytes to write.

If the first parameter is one or two (i.e. `stdout` or `stderr`), the simulator pops up the input/output frame and writes the data there.

It returns the number of bytes that have been written, or -1 if the write operation fails. Possible causes of failure are the attempt to write to a non-existent file descriptor, the attempt to write to file descriptor 0 (standard input) or the attempt to write to a read-only file descriptor.

2.4.6 *SYSCALL 5 - printf()*

SYSCALL 5 expects a variable number of parameters, the first being the address of the so-called "format string". In the format string can be included some placeholders, described in the following list:

- *%s* indicates a string parameter;
- *%i* indicates an integer parameter;
- *%d* behaves like *%i*;
- *%%* literal *%*

For each *%s*, *%d* or *%i* placeholder, SYSCALL 5 expects a parameter, starting from the address of the previous one.

When the SYSCALL finds a placeholder for an integer parameter, it expects that the corresponding parameter is an integer value, when if it finds a placeholder for a string parameter, it expects as a parameter the address of the string.

The result is printed in the input/output frame, and the number of bytes written is put into R1.

If there's an error, -1 is written to R1.

2.5 Other instructions

In this section there are instructions that do not fit in the previous categories.

2.5.1 *BREAK*

The BREAK instruction throws an exception that has the effect to stop the execution if the simulator is running. It can be used for debugging purposes.

2.5.2 *NOP*

The NOP instruction does not do anything, and it's used to create gaps in the source code.

2.5.3 *TRAP*

The TRAP instruction is a deprecated alias for the SYSCALL instruction.

2.5.4 *HALT*

The HALT instruction is a deprecated alias for the SYSCALL 0 instruction, that halts the simulator.

> **Warning**
>
> `HALT` is not a standard MIPS64 instruction (it is part of the WinMIPS64 instruction set, accepted by EduMIPS64 for backward compatibility). The parser emits a warning when it is used; prefer `SYSCALL 0` instead.

CHAPTER

THREE

FLOATING POINT UNIT

This chapter[1] describes the Floating Point Unit (FPU) emulated in EduMIPS64.

In the first paragraph we introduce the double format, the special floating point values defined in the IEEE 754 standard and the exceptions that floating point computations can raise.

In the second paragraph we explain how EduMIPS64 allows users to enable or disable the IEEE floating point traps.

In the third paragraph we describe how double precision numbers and special values can be specified in the source programs.

In the fourth paragraph, we introduce the FCSR register, used by the FPU to represent its state. It contains information about rounding, the boolean results of comparison operations and the policies for handling IEEE floating point exceptions.

In the fifth and last paragraph, we present all the MIPS64 floating point instructions that have been implemented in EduMIPS64.

Before starting the discussion about the FPU, we define the domain of floating point double precision numbers as [-1.79E308,-4.94E-324] {0} [4.94E-324,1.79E308].

3.1 Special values

Floating point arithmetics allows the programmer to choose whether to stop the computation or not, if invalid operations are carried on. In this scenario, operations like the division between zeroes or square roots of negative numbers must produce a result that, not being a number (NaN) is treated as something different.

3.1.1 NaN or Invalid Operation

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) defined that invalid arithmetic operations can either signal the error while the program is running (using a trap for the IEEE exception **Invalid Operation**) or return as a result the special value QNan (Quit Not a Number). Another NaN value, that unconditionally raises the same trap once it is detected as being one of the operands, is SNan (Signalling Not a Number). This value is seldom used in applications, and historically it has been used to initialize variables.

3.1.2 Zeroes or Underflows

Another special value defined by the standard is zero. Since the double format does not include the zero in its domain, it is considered a special value. There is a positive zero and a negative zero: the former is used when a representation of a negative number]-4.94E-324,0[) is attempted, and a result is required (as opposed to allowing an **Underflow** trap), while the latter is used when the number that should be represented is [0,4.94E-324[, and the Underflow trap is disabled.

[1] This chapter is part of the Bachelor's degree thesis by Massimo Trubia: "Progetto e implementazione di un modello di Floating Point Unit per un simulatore di CPU MIPS64".

3.1.3 Infinites or Overflows

When a program attempts to represent a value with an extremely large absolute value ( ]-∞,-1.79E308[ ]1.79E308,+∞[), that is outside the domain of double values, the CPU returns either -∞ or +∞. The alternative is to trigger a trap for the exceptional **Overflow** condition.

Infinites can also be returned in case of a division by zero; in that case the sign of the infinite is given by the product of the sign of the zero and the sign of the dividend. The **Divide by zero** trap can be alternatively raised.

3.2 Exception configuration

EduMIPS64 allows the user to enable or disable the traps for 4 of the 5 IEEE exceptions, through the *FPU Exceptions* tab in the *Configure* → *Settings* window. If any of them is disabled, the respective special value will be returned (as described in *Special values*).

In the situation depicted in Figure *Trap configuration for IEEE exceptions*, in which some checkbox are selected, if the CPU does not mask synchronous exceptions (Figure *Option that masks all the synchronous exceptions*) the selected traps will be raised if the IEEE exceptional condition is encountered (Figure *Trap notification window*).

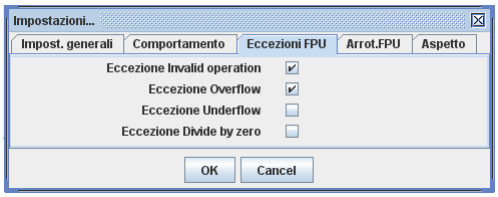

Fig. 1: Trap configuration for IEEE exceptions

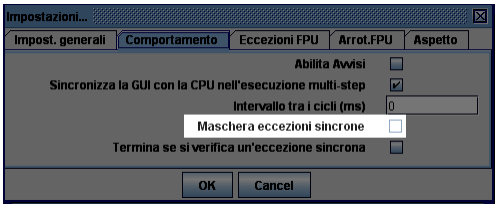

Fig. 2: Option that masks all the synchronous exceptions

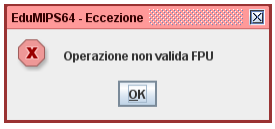

Fig. 3: Trap notification window

3.3 The .double directive

The `.double` directive must be used in the `.data` section of source files, and allows to allocate a memory cell for a *double* value.

The directive can be used in 2 ways:

```
variable-name: .double double_number
variable-name: .double keyword
```

where `double_number` can be represented either in extended notation (`1.0,0.003`), or in scientific notation(`3.7E-12,0.5E32`). `keyword` can be `POSITIVEINFINITY`, `NEGATIVEINFINITY`, `POSITIVEZERO`, `NEGATIVEZERO`, `SNAN` and `QNAN`, thus allowing to directly insert in memory the special values.

3.4 The FCSR register

The FCSR (Floating point Control Status Register) is the register that controls several functional aspects of the FPU. It is 32 bits long and it is represented in the statistics window.

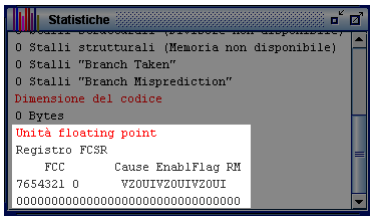

Fig. 4: FCSR register in EduMIPS64

The **FCC** field is 8 bits wide, from 0 to 7. The conditional instructions (`C.EQ.D,C.LT.D`) use it to save the boolean result of comparisons between two registers.

The Cause, Enables and Flag fields handle the dynamics of IEEE exceptions described in *Special values*. Each of them is composed of 5 bits, V (Invalid Operation), Z (Divide by Zero), O (Overflow), U (Underflow) and I (Inexact); the latter is not yet used.

The **Cause** field bits are set if the corresponding IEEE exceptions occur during the execution of a program.

The **Enable** field bits are set through the configuration window and show the IEEE exceptions for which traps are enabled, as shown in Figure *Trap configuration for IEEE exceptions*.

The **Flag** field shows the exceptions that have happened but, since the trap is not enabled for that particular exception, have returned special values (the ones described in *Special values*).

The **RM** field describes the rounding method currently in use to convert floating point numbers to integers (see the description of the `CVT.L.D` instruction).

3.5 Instruction set

This section describes the MIPS64 FPU instruction implemented by EduMIPS64; they are listed in alphabetic order. The operations performed by the instruction are described using a notation according to which the i-th memory cell is represented as `memory[i]`, and the FCC fields of the FCSR register are `FCSR_FCC[cc]`, `cc` [0,7].

In some instructions, to avoid ambiguity, the registers are represented as `GPR[i]` and `FPR[i]`, `i` [0,31], but in most cases we just use the `rx` or `fx` notation, with `x` {d,s,t}. The three letters are used to indicate the purpose of each register (destination, source, third). Lastly, the values returned by conversion operations are represented with the following notation: `convert_conversiontype(register[,rounding_type])`, where the `rounding_type` parameter is optional.

- *ADD.D fd, fs, ft*

  Description: `fd = fs + ft`

  Exceptions: Overflow and underflow traps are generated if the result cannot be represented according to IEEE 754. Invalid operation is raised if fs or ft contain QNaN or SNan, or if an invalid operation ($+\infty - \infty$) is executed.

- *BC1F cc, offset*

  Description: `if FCSR_FCC[cc] == 0 then branch`

If FCSR_FCC[cc] is false, do a PC-relative branch.

Example:

```
C.EQ.D 7,f1,f2
BC1F 7,label
```

In this example, C.EQ.D checks if f1 and f2 are equal, writing the results of the comparison in the 7th bit of the FCC field of the FCSR register. After that, BC1F jumps to label if the result of the comparison is 0 (false).

- *BC1T cc, offset*

  Description: if FCSR_FCC[cc] == 1 then branch

  If FCSR_FCC[cc] is true, do a PC-relative branch.

  Example:

```
C.EQ.D 7,f1,f2
BC1T 7,label
```

  In this example, C.EQ.D checks if f1 and f2 are equal, writing the results of the comparison in the 7th bit of the FCC field of the FCSR register. After that, BC1T jumps to label if the result of the comparison is 1 (true).

- *C.EQ.D cc, fs, ft*

  Description: FCSR_FCC[cc] = (fs==ft)

  Checks if fs is equal to ft, and saves the result of the comparison in FCSR_FCC[cc]. See examples for BC1T, BC1F.

  Exceptions: Invalid Operation can be thrown if fs or ft contain QNaN (trap is triggered if it is enabled) or SNaN (trap is always triggered).

- *C.LT.D cc, fs, ft*

  Description: FCSR_FCC[cc] = (fs<ft)

  Checks if fs is smaller than ft, and saves the result of the comparison in FCSR_FCC[cc].

  Example:

```
C.LT.D 2,f1,f2
BC1T 2,target
```

  In this example, C.LT.D checks if f1 is smaller than f2, and saves the result of the comparison in the second bit of the FCC field of the FCSR register. After that, BC1T jumps to target if that bit is set to 1.

  Exceptions: Invalid Operation can be thrown if fs or ft contain QNaN (trap is triggered if it is enabled) or SNaN (trap is always triggered).

- *CVT.D.L fd, fs*

  Description: fd = convert_longToDouble(fs)

  Converts a long to a double.

  Example:

```
DMTC1 r6,f5
CVT.D.L f5,f5
```

In this example, DMTC1 copies the value of GPR r6 to FPR f5; after that CVT.D.L converts the value stored in f5 from long to double. If for instance r6 contains the value 52, after the execution of DMTC1 the binary representation of 52 gets copied to f5. After the execution of CVT.D.L, f5 contains the IEEE 754 representation of 52.0.

Exceptions: Invalid Operation is thrown if fs contains QNaN, SNaN or an infinite.

- *CVT.D.W fd, fs*

  Description: `fd = convert_IntToDouble(fs)`

  Converts an int to a double.

  Example:

```
MTC1 r6,f5
CVT.D.W f5,f5
```

  In this example, MTC1 copies the lower 32 bit of the GPR r6 into the FPR f5. Then, CVT.D.W, reads f5 as an int, and converts it to double.

  If we had `r6=0xAAAAAAAABBBBBBBB`, after the execution of MTC1 we get `f5=0xXXXXXXXXBBBBBBBB`; its upper 32 bits (`XX..X`) are now UNDEFINED (haven't been overwritten). CVT.D.W interprets f5 as an int (`f5=-1145324613`), and converts it to double(`f5=0xC1D1111111400000 =-1.145324613E9`).

  Exceptions: Invalid Operation is thrown if fs contains QNaN, SNaN or an infinite.

- *CVT.L.D fd, fs*

  Description: `fd = convert_doubleToLong(fs, CurrentRoundingMode)`

  Converts a double to a long, rounding it before the conversion.

  Example:

```
CVT.L.D f5,f5
DMFC1 r6,f5
```

  CVT.L.D converts the double value in f5 to a long; then DMFC1 copies f5 to r6; the result of this operation depends on the current rounding modality, that can be set in the *FPU Rounding* tab of the *Configure* → *Settings* window, as depicted in Figure *FPU Rounding*.

  Exceptions: Invalid Operation is thrown if fs contains an infinite value, any NaN or the result is outside the long domain [-2 ^{63}, 2 ^{63} -1].

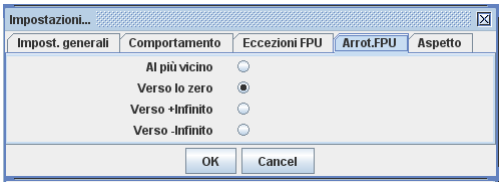

Fig. 5: FPU Rounding

Table 1: Rounding examples

Type	RM field	f5 register	r6 register
To nearest	0	6.4	6
To nearest	0	6.8	7
To nearest	0	6.5	6 (to even)
To nearest	0	7.5	8 (to even)
Towards 0	1	7.1	7
Towards 0	1	-2.3	-2
Towards ∞	2	4.2	5
Towards ∞	2	-3.9	-3
Towards -∞	3	4.2	4
Towards -∞	3	-3.9	-4

- *CVT.W.D fd, fs*

  Description: `fd = convert_DoubleToInt(fs, CurrentRoundingMode)`

  Converts a double to an int, using the current rounding modality.

  Exceptions: Invalid Operation is thrown if fs contains an infinite value, any NaN or the result is outside the signed int domain [-2 ^{31}, 2 ^{31} -1].

- *DIV.D fd, fs, ft*

  Description: `fd = fs / ft`

  Exceptions: Overflow or Underflow are raised if the results cannot be represented using the IEEE 754 standard. Invalid Operation is raised if fs or ft contain QNaN or SNan, or if an invalid operation is executed (0/0, ∞/∞). Divide by zero is raised if a division by zero is attempted with a dividend that is not QNaN or SNaN.

- *DMFC1 rt,fs*

  Description: `rt = fs`

  Executes a bit per bit copy of the FPR fs into the GPR rt.

- *DMTC1 rt, fs*

  Description: `fs = rt`

  Executes a bit per bit copy of the GPR rt into the FPR fs.

- *L.D ft, offset(base)*

  Description: `ft = memory[GPR[base] + offset]`

  Loads from memory a doubleword and stores it in ft.

> **Warning**
>
> `L.D` is not present in the MIPS64 ISA; it is an alias for `LDC1` that EduMIPS64 accepts for compatibility with WinMIPS64. The parser emits a warning when it is used; prefer `LDC1` instead.

- *LDC1 ft, offset(base)*

  Description: `ft = memory[GPR[base] + offset]`

  Loads from memory a doubleword and stores it in ft.

- *LWC1 ft, offset(base)*

  Description: `ft = memory[GPR[base] + offset]`

  Loads from memory a word and stores it in ft.

- *MFC1 rt, fs*

  Description: `rt = readInt(fs)`

  Reads the fs FPR as an int and writes its value to the rt GPR as long. *Example*:

```
MFC1 r6,f5
SD r6,mem(R0)
```

  Let `f5=0xAAAAAAAABBBBBBBB`; `MFC1` reads f5 as an int (lower 32 bits), interpreting `BBBBBBBB` as `-1145324613`, and writes the value to f6 (64 bits). After the execution of `MFC1`, `r6=0xFFFFFFFFBBBBBBBB = -1145324613`. So the `SD` instruction will write to memory a doubleword with this value, since the sign in r6 was extended.

- *MOVF.D fd, fs, cc*

  Description: `if FCSR_FCC[cc] == 0 then fd=fs`

  If FCSR_FCC[cc] is false, copies fs to fd.

- *MOVT.D fd, fs, cc*

  Description: `if FCSR_FCC[cc] == 1 then fd=fs`

  If FCSR_FCC[cc] is true, copies fs to fd.

- *MOV.D fd, fs*

  Description: `fd = fs`

  Copies fs to fd.

- *MOVN.D fd, fs, rt*

  Description: `if rt != 0 then fd=fs`

  If rt is not zero, copies fs to fd.

- *MOVZ.D fd, fs, rt*

  Description: `if rt == 0 then fd=fs`

  If rt is equal to zero, copies fs to fd.

- *MTC1 rt, fs*

  Description: fs = $rt_{0..31}$

  Copies the lower 32 bit of rt to fs.

  Example:

```
MTC1 r6,f5
```

  Let `r5=0xAAAAAAAABBBBBBBB`; `MTC1` reads the lower 32 bits of r5 copying them to the 32 lower bits of f5. The higher 32 bits of f5 are not overwritten.

- *MUL.D fd, fs, ft*

  Description: `fd = fs × ft`

Exceptions: Overflow or Underflow are raised if the results cannot be represented using the IEEE 754 standard. Invalid Operation is raised if fs or ft contain QNaN or SNan, or if an invalid operation is executed (multiply by ∞ OR BY QNaN).

- *S.D ft, offset(base)*

  Description: `memory[base+offset] = ft`

  Copies ft to memory.

> **Warning**
>
> `S.D` is not present in the MIPS64 ISA; it is an alias for `SDC1` that EduMIPS64 accepts for compatibility with WinMIPS64. The parser emits a warning when it is used; prefer `SDC1` instead.

- *SDC1 ft, offset(base)*

  Description: `memory[base+offset] = ft`

  Copies ft to memory.

- *SUB.D fd, fs, ft*

  Description: `fd = fs-ft`

  Exceptions: Overflow and underflow traps are generated if the result cannot be represented according to IEEE 754. Invalid operation is raised if fs or ft contain QNaN or SNan, or if an invalid operation ($+\infty - \infty$) is executed.

- *SWC1 ft, offset(base)*

  Description: `memory[base+offset] = ft`

  Copies the lower 32 bits of ft to memory.

CHAPTER

FOUR

CODE EXAMPLES

In this chapter you'll find some sample listings that will be useful in order to understand how EduMIPS64 works.

4.1 SYSCALL

It's important to understand that examples for SYSCALL 1-4 refer to the *print.s* file, that is the example for SYSCALL 5. If you want to run the examples, you should copy the content of that example in a file named *print.s* and include it in your code.

Some examples use an already existing file descriptor, even if it doesn't truly exist. If you want to run those examples, use the SYSCALL 1 example to open a file.

4.1.1 SYSCALL 0

When SYSCALL 0 is called, it stops the execution of the program. Example:

```
.code
    daddi   r1, r0, 0    ; saves 0 in R1
    syscall 0            ; exits
```

4.1.2 SYSCALL 1

Example program that opens a file:

```
                .data
error_op:       .asciiz     "Error opening the file"
ok_message:     .asciiz     "All right"
params_sys1:    .asciiz     "filename.txt"
                .word64     0xF

                .text
open:           daddi       r14, r0, params_sys1
                syscall     1
                daddi       $s0, r0, -1
                dadd        $s2, r0, r1
                daddi       $a0,r0,ok_message
                bne         r1,$s0,end
                daddi       $a0,r0,error_op

end:            jal         print_string
                syscall 0
```

(continues on next page)

(continued from previous page)

```
                #include    print.s
```

In the first two rows we write to memory the strings containing the error message and the success message that we will pass to print_string function, and we give them two labels. The print_string function is included in the print.s file.

Next, we write to memory the data required from SYSCALL 1 (row 4, 5), the path of the file to be opened (that must exist if we work in read or read/write mode) and, in the next memory cell, an integer that defines the opening mode.

In this example, the file was opened using the following modes: *O_RDWR | O_CREAT | O_APPEND*. The number 15 (0xF in base 16) comes from the sum of the values of these three modes (3 + 4 + 8).

We give a label to this data so that we can use it later.

In the .text section, we save the address of params_sys1 (that for the compiler is a number) in register r14; next we can call SYSCALL 1 and save the content of r1 in $s2, so that we can use it in the rest of the program (for instance, with other SYSCALL).

Then the print_string function is called, passing error_op as an argument if r1 is equal to -1 (rows 13-14) or else passing ok_message as an argument if everything went smoothly (rows 12 and 16).

4.1.3 SYSCALL 2

Example program that closes a file:

```
                .data
params_sys2:    .space 8
error_cl:       .asciiz     "Error closing the file"
ok_message:     .asciiz     "All right"

                .text
close:          daddi       r14, r0, params_sys2
                sw          $s2, params_sys2(r0)
                syscall     2
                daddi       $s0, r0, -1
                daddi       $a0, r0, ok_message
                bne         r1, $s0, end
                daddi       $a0, r0, error_cl

end:            jal         print_string
                syscall     0

                #include    print.s
```

First we save some memory for the only argument of SYSCALL 2, the file descriptor of the file that must be closed (row 2), and we give it a label so that we can access it later.

Next we put in memory the strings containing the error message and the success message, that will be passed to the print_string function (rows 3, 4).

In the .text section, we save the address of params_sys2 in r14; then we can call SYSCALL 2.

Now we call the print_string function using error_cl as a parameter if r1 yields -1 (row 13), or we call it using ok_message as a parameter if all went smoothly (row 11).

Note: This listing needs that registry $s2 contains the file descriptor of the file to use.

4.1.4 SYSCALL 3

Example program that reads 16 bytes from a file and saves them to memory:

```
                .data
params_sys3:    .space      8
ind_value:      .space      8
                .word64     16
error_3:        .asciiz     "Error while reading from file"
ok_message:     .asciiz     "All right"

value:          .space      30

                .text
read:           daddi       r14, r0, params_sys3
                sw          $s2, params_sys3(r0)
                daddi       $s1, r0, value
                sw          $s1, ind_value(r0)
                syscall     3
                daddi       $s0, r0, -1
                daddi       $a0, r0,ok_message
                bne         r1, $s0,end
                daddi       $a0, r0,error_3

end:            jal         print_string
                syscall     0

                #include    print.s
```

The first 4 rows of the .data section contain the arguments of SYSCALL 3, the file descriptor of the file from which we must read, the memory address where the SYSCALL must save the read data, the number of bytes to read. We give labels to those parameters that must be accessed later. Next we put, as usual, the strings containing the error message and the success message.

In the .text section, we save the params_sys3 address to register r14, we save in the memory cells for the SYSCALL parameters the file descriptor (that we suppose to have in $s2) and the address that we want to use to save the read bytes.

Next we can call SYSCALL 3, and then we call the print_string function passing as argument error_3 or ok_message, according to the success of the operation.

4.1.5 SYSCALL 4

Example program that writes to a file a string:

```
                .data
params_sys4:    .space      8
ind_value:      .space      8
                .word64     16
error_4:        .asciiz     "Error writing to file"
ok_message:     .asciiz     "All right"
value:          .space      30

                .text

write:          daddi       r14, r0,params_sys4
```

(continues on next page)

(continued from previous page)

```
                sw          $s2, params_sys4(r0)
                daddi       $s1, r0,value
                sw          $s1, ind_value(r0)
                syscall     4
                daddi       $s0, r0,-1
                daddi       $a0, r0,ok_message
                bne         r1, $s0,end
                daddi       $a0, r0,error_4

end:            jal         print_string
                syscall     0

                #include    print.s
```

The first 4 rows of the .data section contain the arguments of SYSCALL 4, the file descriptor of the file to which we must write, the memory address from where the SYSCALL must read the bytes to write, the number of bytes to write. We give labels to those parameters that must be accessed later. Next we put, as usual, the strings containing the error message and the success message.

In the .text section, we save the params_sys4 address to register r14, we save in the memory cells for the SYSCALL parameters the file descriptor (that we suppose to have in $s2) and the address from where we must take the bytes to write.

Next we can call SYSCALL 4, and then we call the print_string function passing as argument error_4 or ok_message, according to the success of the operation.

4.1.6 SYSCALL 5

Example program that contains a function that prints to standard output the string contained in $a0:

```
                .data
params_sys5:    .space  8

                .text
print_string:
                sw      $a0, params_sys5(r0)
                daddi   r14, r0, params_sys5
                syscall 5
                jr      r31
```

The second row is used to save space for the string that must be printed by the SYSCALL, that is filled by the first instruction of the .text section, that assumes that in $a0 there's the address of the string to be printed.

The next instruction puts in r14 the address of this string, and then we can call SYSCALL 5 and print the string. The last instruction sets the program counter to the content of r31, as the usual MIPS calling convention states.

4.1.7 A more complex usage example of SYSCALL 5

SYSCALL 5 uses a not-so-simple arguments passing mechanism, that will be shown in the following example:

```
                .data
format_str:     .asciiz   "%dth of %s:\n%s version %i.%i is being tested!"
s1:             .asciiz   "June"
s2:             .asciiz   "EduMIPS64"
```

(continues on next page)

(continued from previous page)

```
fs_addr:        .space    4
                .word     5
s1_addr:        .space    4
s2_addr:        .space    4
                .word     0
                .word     5
test:
                .code
                daddi     r5, r0, format_str
                sw        r5, fs_addr(r0)
                daddi     r2, r0, s1
                daddi     r3, r0, s2
                sd        r2, s1_addr(r0)
                sd        r3, s2_addr(r0)
                daddi     r14, r0, fs_addr
                syscall   5
                syscall   0
```

The address of the format string is put into R5, whose content is then saved to memory at address fs_addr. The string parameters' addresses are saved into s1_addr and s2_addr. Those two string parameters are the ones that match the two %s placeholders in the format string.

Looking at the memory, it's obvious that the parameters matching the placeholders are stored immediately after the address of the format string: numbers match integer parameters, while addresses match string parameters. In the s1_addr and s2_addr locations there are the addresses of the two strings that we want to print instead of the %s placeholders.

The execution of the example will show how SYSCALL 5 can handle complex format strings like the one stored at format_str.

CHAPTER

FIVE

THE DESKTOP USER INTERFACE

EduMIPS64 ships as a Java desktop application based on Swing. This chapter describes the desktop GUI, which is inspired by the WinMIPS64 user interface. In fact, the main window is identical, except for some menus.

This chapter also documents the command line options that are supported by the desktop JAR. The same JAR can be run as a graphical application or in headless / CLI mode (see *Command line options*).

The EduMIPS64 main window is composed by a menu bar and six frames, showing different aspects of the simulation. There's also a status bar, that has the double purpose to show the content of memory cells and registers when you click them and to notify the user that the simulator is running when the simulation has been started but verbose mode is not selected.

The status bar also shows the CPU status. It can show one of the following four states:

- *READY* The CPU hasn't executed any instructions (no program is loaded).
- *RUNNING* The CPU is executing a series of instructions.
- *STOPPING* The CPU has found a termination instruction, and is executing the instructions that are already in the pipeline before terminating the execution.
- *HALTED* The CPU is stopped: a program just finished running.

Note that the CPU status is different from the simulator status. The simulator may execute a number of CPU cycles and then stop executing, allowing the user to inspect memory and registers: in this state, between CPU cycles, the CPU stays in *RUNNING* or *STOPPING* state. Once the CPU reaches the *HALTED* state, the user cannot run any CPU cycle without loading a program again (the same program, or a different one).

There are more details in the following sections.

5.1 The menu bar

The menu bar contains six menus:

5.1.1 File

The File menu contains menu items about opening files, resetting or shutting down the simulator, writing trace files.

- *Open...* Opens a dialog that allows the user to choose a source file to open.
- *Open recent* Shows the list of the recent files opened by the simulator, from which the user can choose the file to open
- *Reset* Resets the simulator, keeping open the file that was loaded but resetting the execution.
- *Write Dinero Tracefile...* Writes the memory access data to a file, in xdin format.
- *Exit* Closes the simulator.

The *Write Dinero Tracefile...* menu item is only available when a whole source file has been executed and the end has already been reached.

5.1.2 Execute

The Execute menu contains menu items regarding the execution flow of the simulation.

- *Single Cycle* Executes a single simulation step
- *Run* Starts the execution, stopping when the simulator reaches a *SYSCALL 0* (or equivalent) or a *BREAK* instruction, or when the user clicks the Stop menu item (or presses F9).
- *Multi Cycle* Executes some simulation steps. The number of steps executed can be configured through the Setting dialog.
- *Stop* Stops the execution when the simulator is in "Run" or "Multi cycle" mode, as described previously.

This menu is only available when a source file is loaded and the end of the simulation is not reached. The *Stop* menu item is available only in "Run" or "Multi Cycle" mode.

Note that the simulator slows down when updating the UI. If you want to execute long (thousands of cycles) programs quickly, disable the "Sync graphics with CPU in multi-step execution" option.

5.1.3 Configure

The Configure menu provides facilities for customizing EduMIPS64 appearance and behavior.

- *Settings...* Opens the Settings dialog, described in the next sections of this chapter;
- *Change Language* Allows the user to change the language used by the user interface. English, Italian and Simplified Chinese are supported. This change affects every aspect of the GUI, from the title of the frames to the online manual and warning/error messages.

The *Settings...* menu item is not available when the simulator is in "Run" or "Multi Cycle" mode, because of potential race conditions.

5.1.4 Tools

This menu contains only an item, used to invoke the Dinero Frontend dialog.

- *Dinero Frontend...* Opens the Dinero Frontend dialog.

This menu is not available until you have executed a program and the execution has reached its end.

5.1.5 Window

This menu contains items related to operations with frames.

- *Tile* Sorts the visible windows so that no more than three frames are put in a row. It tries to maximize the space occupied by every frame.

The other menu items simply toggle the status of each frame, making them visible or minimizing them.

5.1.6 Help

This menu contains help-related menu items.

- *Manual...* Shows the Help dialog.
- *About us...* Shows a cute dialog that contains the names of the project contributors, along with their roles.

5.2 Frames

The GUI is composed by seven frames, six of which are visible by default, and one (the I/O frame) is hidden.

5.2.1 Cycles

The Cycles frame shows the evolution of the execution flow during time, showing for each time slot which instructions are in the pipeline, and in which stage of the pipeline they are located.

5.2.2 Registers

The Registers frame shows the content of each register. By left-clicking on them you can see in the status bar their decimal (signed) value, while double-clicking on them will pop up a dialog that allows the user to change the value of the register.

5.2.3 Statistics

The Statistics frame shows some statistics about the program execution.

Note that during the last execution cycle the cycles counter is not incremented, because the last execution cycle is not a full CPU cycle but rather a pseudo-cycle whose only duties are to remove the last instruction from the pipeline and increment the counter of executed instructions.

The Statistics frame also displays L1 cache statistics when the cache simulator is enabled:

- **L1I Reads** - Number of read accesses to the L1 instruction cache
- **L1I Read Misses** - Number of read misses in the L1 instruction cache
- **L1D Reads** - Number of read accesses to the L1 data cache
- **L1D Read Misses** - Number of read misses in the L1 data cache
- **L1D Writes** - Number of write accesses to the L1 data cache
- **L1D Write Misses** - Number of write misses in the L1 data cache

These statistics help analyze cache performance and understand memory access patterns in your program. Cache misses indicate when the processor needs to access slower main memory instead of the faster cache.

5.2.4 Pipeline

The Pipeline frame shows the actual status of the pipeline, showing which instruction is in which pipeline stage. Different colors highlight different pipeline stages.

5.2.5 Memory

The Memory frame shows memory cells content, along with labels and comments taken from the source code. Memory cells content, like registers, can be modified double-clicking on them, and clicking on them will show their decimal value in the status bar. The first column shows the hexadecimal address of the memory cell, and the second column shows the value of the cell. Other columns show additional info from the source code.

5.2.6 Code

The Code window shows the instructions loaded in memory. The first column shows the address of the instruction, while the second column shows the hexadecimal representation of the instructions. Other columns show additional info taken from the source code.

5.2.7 Input/Output

The Input/Output window provides an interface for the user to see the output that the program creates through the SYSCALLs 4 and 5. Actually it is not used for input, as there's a dialog that pops up when a SYSCALL 3 tries to read from standard input, but future versions will include an input text box.

5.3 Dialogs

Dialogs are used by EduMIPS64 to interact with the user in many ways. Here's a summary of the most important dialogs:

5.3.1 Settings

In the Settings dialog various aspects of the simulator can be configured. Clicking on the "OK" button will cause the options to be saved, while clicking on "Cancel" (or simply closing the window) will cause the changes to be ignored. Don't forget to click "OK" if you want to save your changes. The "Reset to defaults" button restores all settings to their default values after asking for confirmation; this is useful to recover from a misconfigured simulator without having to change each option manually.

The Main Settings tab allow to configure forwarding and the number of steps in the Multi Cycle mode.

The Behavior tab allow to enable or disable warnings during the parsing phase, the "Sync graphics with CPU in multi-step execution" option, when checked, will synchronize the frames' graphical status with the internal status of the simulator. This means that the simulation will be slower, but you'll have an explicit graphical feedback of what is happening during the simulation. If this option is checked, the "Interval between cycles" option will influence how many milliseconds the simulator will wait before starting a new cycle. Those options are effective only when the simulation is run using the "Run" or the "Multi Cycle" options from the Execute menu.

The last two options set the behavior of the simulator when a synchronous exception is raised. If the "Mask synchronous exceptions" option is checked, the simulator will ignore any Division by zero or Integer overflow exception. If the "Terminate on synchronous exception" option is checked, the simulation will be halted if a synchronous exception is raised. Please note that if synchronous exceptions are masked, nothing will happen, even if the termination option is checked. If exceptions are not masked and the termination option is not checked, a dialog will pop out, but the simulation will go on as soon as the dialog is closed. If exceptions are not masked and the termination option is checked, the dialog will pop out, and the simulation will be stopped as soon as the dialog is closed.

The last tab allows to change the appearance of the user interface. There are options to change the colors associated to the different pipeline stages, an option to choose whether memory cells are shown as long or double values and an option to set the UI font size.

The Cache tab allows you to configure the L1 cache simulator settings:

- **L1 Data Cache Size** - Size of the L1 data cache in bytes (default: 1024)
- **L1 Instruction Cache Size** - Size of the L1 instruction cache in bytes (default: 1024)
- **L1 Data Cache Block Size** - Size of each cache block in bytes (default: 16)
- **L1 Instruction Cache Block Size** - Size of each cache block in bytes (default: 16)
- **L1 Data Cache Associativity** - Number of cache ways, determining cache organization (default: 1)
- **L1 Instruction Cache Associativity** - Number of cache ways, determining cache organization (default: 1)

These settings allow you to experiment with different cache configurations to understand their impact on program performance. The cache simulator models separate L1 data and instruction caches, which is common in modern processors.

Note that the UI scaling with font size is far from perfect, but it should be enough to make the simulator usable with high-resolution displays (e.g., 4k).

5.3.2 L1 Cache Simulator

EduMIPS64 includes an integrated L1 cache simulator that models the behavior of separate instruction and data caches. The cache simulator operates automatically during program execution and provides detailed statistics about cache performance.

The cache simulator models:

- **L1 Instruction Cache** - Caches instruction fetches from memory
- **L1 Data Cache** - Caches data reads and writes from memory operations

Each cache can be independently configured with different parameters:

- **Size** - Total capacity of the cache in bytes
- **Block Size** - Size of each cache line in bytes (also called cache block)
- **Associativity** - Number of ways in the cache set (1 = direct-mapped, >1 = set-associative)

The cache simulator uses a Least Recently Used (LRU) replacement policy for set-associative caches. Cache statistics are updated in real-time during execution and displayed in the Statistics frame.

Cache performance can significantly impact program execution time, especially for programs with poor memory locality. Use the cache simulator to:

- Analyze memory access patterns in your programs
- Experiment with different cache configurations
- Understand the performance impact of cache misses
- Learn about cache behavior in real processors

The cache configuration can be changed through the Settings dialog (Cache tab) and takes effect when the simulation is reset.

5.3.3 Forwarding

EduMIPS64 models a 5-stage MIPS64 pipeline (IF, ID, EX, MEM, WB) and supports optional data forwarding to reduce the number of stalls caused by Read After Write (RAW) hazards. Forwarding can be toggled from the Main Settings tab of the Settings dialog.

When forwarding is **disabled**, source registers are read from the register file in the ID stage. An instruction that reads a register written by one of the two previous instructions must wait until the producer reaches WB, which results in two RAW stalls.

When forwarding is **enabled**, the simulator implements two forwarding paths that match the canonical Hennessy & Patterson MIPS pipeline:

- **EX** $\rightarrow$ **EX**: an ALU result produced at the end of EX can be forwarded to the EX input of the immediately following instruction (no stall).
- **MEM** $\rightarrow$ **EX**: a value produced at the end of MEM (typically a load result, or an ALU result one cycle later) can be forwarded to the EX input of an instruction in EX (no stall).

There is **no EX** $\rightarrow$ **ID forwarding path** and, by symmetry, no MEM $\rightarrow$ ID forwarding path. This has two visible consequences for branch and register-based jump instructions (`BEQ`, `BNE`, `BEQZ`, `BNEZ`, `BGEZ`, `JR`, `JALR`), which evaluate their condition or target in the ID stage:

- **ALU** $\rightarrow$ **branch**: if the source register of the branch is being produced by the immediately preceding ALU instruction, the value is not yet available at the start of ID, and the simulator inserts one RAW stall.

- **Load → branch**: if the source register of the branch is being produced by the immediately preceding load, the value is only available at the end of MEM, while the branch needs it at the start of ID. The simulator inserts two RAW stalls: this is the union of the load-use hazard (one stall) and the branch-in-ID hazard (one extra stall).

Both behaviors match the description in Hennessy & Patterson.

The following table summarises the typical RAW stalls for the most common producer/consumer pairs:

Producer → consumer	No forwarding	With forwarding
ALU → ALU	2	0
ALU → store (data operand)	2	0
Load → ALU (load-use)	2	1
ALU → branch / register jump	2	1
Load → branch / register jump	2	2

5.3.4 Dinero Frontend

The Dinero Frontend dialog allows to feed a DineroIV process with the trace file internally generated by the execution of the program. In the first text box there is the path of the DineroIV executable, and in the second one there must be the parameters of DineroIV.

The lower section contains the output of the DineroIV process, from which you can take the data that you need.

5.3.5 Help

The Help dialog brings up the on-line manual, which is an HTML copy of this document.

5.4 Command line options

Several command line options are available. They are described in the following list, with the long name enclosed in round brackets. Long and short names can be used in the same way.

- *-v (–version)* prints the simulator version and exits.
- *-h (–help)* prints a help message with a brief summary of command line options, then exits.
- *-f (–file) filename* opens *filename* in the simulator
- *-r (–reset)* resets the stored configuration to the default values
- *-d (–debug)* enters Debug mode
- *-hl (–headless)* Runs EduMIPS64 in headless mode (no gui)
- *–verbose* enables verbose mode in headless/CLI mode, showing detailed simulator output such as start/end messages and progress indicators. By default, the CLI runs in quiet mode, which minimizes simulator output to avoid confusion with program output (e.g., from SYSCALL 5).

The *–debug* flag has the effect to activate Debug mode. In this mode, a new frame is available, the Debug frame, and it shows the log of internal activities of EduMIPS64. It is not useful for the end user, it is meant to be used by EduMIPS64 developers.

CHAPTER

SIX

THE INTERACTIVE COMMAND LINE INTERFACE

In addition to the desktop GUI, the EduMIPS64 JAR can be run as an interactive command line shell. The shell is meant for batch-style experimentation with assembly programs (running test programs in a script, scripted course material, automated grading, debugging from a terminal, etc.) and offers the same simulator core as the desktop GUI, just with a textual interface.

This chapter describes the *commands* available inside the shell. The command line *options* used to launch the JAR (`--headless`, `--file`, `--verbose`, ...) are documented in *Command line options*; in particular, the shell is reached by launching the JAR with `--headless` (with or without `--verbose`).

When `--verbose` is enabled, the shell prints additional informative messages — a welcome banner on startup, "execution started" / "execution finished" messages around `run`, progress dots during long runs, parser warnings after `load`, and so on. Without `--verbose` the shell produces only the program output (e.g. the strings printed by `SYSCALL 5`) and the explicit replies of `show`, `config` and similar commands. The quiet default is convenient when EduMIPS64 is embedded in a pipeline of other tools.

6.1 The prompt

On startup, the shell prints a stylised banner showing the name **EduMIPS64** in heavy block letters, with gold "Edu", bright "MIPS" and red "64" matching the colours of the project logo. Each segment fades from a bold bright tone at the top to a deeper shade at the bottom, which gives the blocks a soft drop-shadow / 3-D effect. The version, codename and build date appear right below. The banner uses ANSI colours and Unicode block characters, so it appears at its best on any modern terminal (Linux/macOS, Windows Terminal, Codespaces, ...). When the standard output is not a TTY, when `COLUMNS` is too small, or when the encoding is not UTF-8, the banner falls back automatically to a plain one-line version string, which keeps scripts and pipelines clean. If you do not want the banner at all, pass `--no-banner` on the command line.

After the banner, the shell prints a coloured prompt of the form:

```
edumips64 [READY] >
```

The text in brackets reflects the current CPU status (`READY`, `RUNNING`, `HALTED` or `STOPPING`) and changes colour accordingly, giving an at-a-glance hint of what the simulator is doing.

Commands are read one line at a time and tokenized at whitespace. Pressing Enter on an empty line reprints the help. The shell loop runs until the `exit` command is issued (or until `Ctrl+D` / EOF is sent on standard input).

Every command accepts `-h` / `--help`, which prints its specific usage. The top-level `help` command lists every available command together with a short description; this is the easiest way to discover the shell's capabilities.

6.2 Browsing the user manual from the shell

The shell ships with the chapters of this user manual bundled inside the JAR, so you can read them from the terminal without leaving EduMIPS64. Type `help topics` to see the list, then `help <topic>` to read a specific chapter, for example:

```
edumips64 [READY] > help topics
edumips64 [READY] > help cli
edumips64 [READY] > help instructions
```

The chapters served by `help` are the same reStructuredText sources used to build the website and PDF (`cli-interface.rst`, `source-files-format.rst`, `instructions.rst`, `examples.rst`, `fpu.rst`, ...), copied verbatim into the JAR by the build. Long chapters are paged through a built-in "more"-style pager; press `Enter` / `space` to advance, `q` to quit. When stdin is not a TTY the pager is bypassed and the chapter is dumped in one go.

6.3 Available commands

The shell exposes a small number of commands. They map directly to the simulator's lifecycle: load a source file, single-step or run it, inspect the resulting state, and optionally dump a Dinero trace file for cache analysis.

6.3.1 load

Provide a new file to execute:

```
> load path/to/program.s
```

`load` parses the given file and prepares the simulator for execution. On success the CPU enters the `RUNNING` state, ready to be advanced with `step` or `run`.

If the parser reports errors the file is not loaded and the error description is printed. If only warnings are produced, the file is still loaded — matching the behaviour of the GUI; with `--verbose` the warnings are also printed.

A new `load` cannot be issued while a previous program is still running; use `reset` first to bring the CPU back to a loadable state.

6.3.2 step

Make the CPU state machine advance N cycles:

```
> step          # advances by 1 cycle (default)
> step 10       # advances by 10 cycles
```

After every cycle the contents of the pipeline are printed, so `step` is the canonical way to follow the execution one instruction at a time and observe how instructions traverse IF / ID / EX / MEM / WB and the FPU stages.

If the program reaches its end (`SYSCALL 0` or `BREAK`) before all the requested cycles have been executed, `step` stops at that point and prints the corresponding message.

6.3.3 run

Execute the program without intervention:

```
> run
```

The simulator advances until the program terminates with `SYSCALL 0` (or equivalent) or with a `BREAK` instruction. With `--verbose`, the shell prints a *start* / *end* banner around the run and a progress dot every thousand cycles, plus a

final summary with the total number of executed cycles and the wall-clock time it took. Use `run` to obtain the program's output (`SYSCALL 4` / `SYSCALL 5`) and final state quickly, then `show` to inspect the result.

6.3.4 show

Inspect the state of the simulated CPU. `show` is a group of sub-commands; each one prints a different aspect of the simulator state to standard output.

- `show registers` — prints all 32 integer general-purpose registers (`R0`–`R31`) with their current value.
- `show register N` — prints the content of integer register `N` (`0 N 31`).
- `show fps` — prints all 32 floating-point registers (`F0`–`F31`).
- `show fp N` — prints the content of floating-point register `N` (`0 N 31`).
- `show fcsr` — prints the Floating-Point Control and Status Register.
- `show hi` / `show lo` — print the contents of the special `HI` and `LO` registers used by multiply / divide instructions.
- `show memory` — prints the contents of the simulated main memory.
- `show symbols` — prints the symbol table (labels declared in the `.data` and `.code` sections together with their addresses).
- `show pipeline` — prints which instruction is currently in each stage of the pipeline.

Calling `show` without a sub-command prints the list of available sub-commands.

6.3.5 dinero

Write a Dinero tracefile to a file:

```
> dinero trace.xdin
```

This produces a textual trace of the memory accesses performed by the program in the format expected by the Dinero IV cache simulator, suitable for offline cache analysis. `dinero` can be issued at any point during execution; the tracefile reflects the accesses observed so far.

6.3.6 config

Print the current configuration values:

```
> config
```

This is the same set of preferences that the desktop GUI exposes in the *Settings* dialog (forwarding on/off, tracefile path, cache parameters, behavioural options, ...). It is read-only — the shell prints the current values and exits the command. The values are loaded from the same configuration store used by the GUI, so changes made in the GUI are visible from the shell and vice versa.

6.3.7 reset

Reset the CPU state machine:

```
> reset
```

`reset` re-initializes memory, registers, the symbol table, the I/O manager, the cache simulator and the parser, bringing the CPU back to the `READY` state. Issue `reset` to load a different program after the current one has been loaded, or to start a program over from scratch after a partial run.

6.3.8 help

Show the list of available commands together with a short description of each one. `help` is the entry point for discovering what the shell can do; combine it with the per-command `-h` / `--help` option to see the parameters accepted by each sub-command.

6.3.9 exit

Quit the shell and terminate the JVM. The same effect can be achieved by sending `Ctrl+D` (EOF) on standard input.

6.4 A typical session

Putting the commands together, a typical interactive session looks like this:

```
$ java -jar edumips64.jar --headless --verbose
> load examples/hello.s
File loaded: /.../examples/hello.s
> step 5
... (5 cycles of pipeline contents)
> run
Hello, world!
Execution finished in 42 cycles, 3 ms
> show registers
... (R0..R31)
> show pipeline
... (final pipeline state)
> dinero hello.xdin
> reset
> load examples/sum.s
> run
> exit
```

Standard input from `SYSCALL 3` is read directly from the terminal, so programs that ask for user input work transparently in the shell.

6.5 Scripting tips

Because the shell reads commands from standard input one per line, a script can be fed in directly:

```
$ java -jar edumips64.jar --headless < session.txt
```

A `session.txt` containing for example:

```
load examples/hello.s
run
show registers
exit
```

will load the program, run it, print the final register file and quit. Combined with `--verbose` and shell redirection, this makes it straightforward to integrate EduMIPS64 in automated test suites or to capture reproducible traces of an execution.

CHAPTER

SEVEN

THE WEB USER INTERFACE

EduMIPS64 is also available as a web application that runs entirely in the browser. The simulator core is cross-compiled from Java to JavaScript and runs as a Web Worker, while the user interface is built with React. The production deployment is hosted at https://web.edumips.org.

This chapter describes the web frontend. For the source file format, the supported instruction set, the FPU and example programs, refer to the other chapters of this manual: those are independent of the user interface.

7.1 Layout overview

The window is split into a top toolbar and two main areas:

- On the left, the **code editor** (a Monaco-based MIPS64 editor).
- On the right, a stack of collapsible panels showing the runtime state of the simulation: **Issues**, **Statistics**, **Pipeline**, **Registers**, **Memory**, **Standard Output**, **Cache Configuration** and **General Settings**.

7.2 The top toolbar

The toolbar at the top of the window groups every action that controls the simulator. Each button has a tooltip that describes its effect.

- **EduMIPS64 logo and "Web Version" label** — indicate the running build. A coloured chip is shown next to the label when the build is not the production one:
  - a yellow `PR #N` chip identifies a per-pull-request preview build and links back to the originating pull request on GitHub;
  - a blue `dev` chip identifies a local development build.
- **CPU status chip** — shows the current state of the simulated CPU with a colour code:
  - `READY` (green) — no program is loaded yet, or the CPU has just been reset;
  - `RUNNING` (yellow) — a program is loaded and the CPU is in the middle of an execution;
  - `STOPPING` / `STOPPED` (red) — a termination instruction has been fetched and the pipeline is draining, or the program has ended.
- **Load** — parses the contents of the editor and loads the resulting program into the simulator. The button is disabled while the simulator is running, and is hidden once a program has been loaded successfully.
- **Single Step** — executes one CPU cycle.
- **Multi Step** — executes a configurable number of CPU cycles in a single click. The number of steps is shown in the button's tooltip and can be changed in the *General Settings* panel ("Multi Step Size").

- **Run All** — executes the program until it terminates with a `SYSCALL 0` (or equivalent) or a `BREAK` instruction, or until it is paused or stopped manually. Between batches of cycles the simulator can wait a configurable delay (`Execution Delay`) so that long runs remain visually observable.
- **Pause** — interrupts a running execution at the current cycle. Single Step / Multi Step / Run All can then be used to continue.
- **Stop** — halts the running execution and resets the CPU to the `READY` state, clearing registers, memory and pipeline.
- **Clear** — empties the code editor, leaving only an empty assembly skeleton (`.data` and `.code` directives plus a final `SYSCALL 0`). The Clear button is disabled while the CPU is running.
- **Restore default sample** — replaces the editor contents with the bundled sample program shipped with EduMIPS64 (the same one shown on a fresh installation). This is useful to recover a known-good starting point after experimenting, or to discard the persisted editor contents (see *Saving and loading* below). The button is disabled while the CPU is running.
- **Open Code** — opens a local file (typically a `.s` file) and loads its contents into the editor.
- **Save Code** — saves the current contents of the editor to a local file named `code.s`.
- **Help (?)** — opens this manual inside the application, with a navigation drawer on the left and a language selector that lets you switch between English, Italian and Chinese. The Help dialog also includes an *About* tab that shows the version of the simulator and a description of the running build.

Buttons that would have no effect in the current state are automatically disabled. For example, `Single Step`, `Multi Step` and `Run All` are disabled until a program has been loaded with `Load`, and `Pause` is only available while a long execution is in progress.

7.3 The code editor

The code editor is based on Monaco — the editor that powers Visual Studio Code — and is dedicated to writing MIPS64 assembly. It supports all the usual code-editor conveniences (multi-cursor, find & replace, line numbers, undo/redo, automatic layout, automatic light/dark theme based on the OS preference) plus a number of EduMIPS64-specific features described below.

7.3.1 Syntax highlighting

The editor provides syntax highlighting for MIPS64 sources:

- labels (lines starting with an identifier followed by `:`);
- every instruction supported by EduMIPS64 (the list of valid instructions is computed at runtime from the simulator core);
- directives that start with `.`, e.g. `.data`, `.code`, `.word`;
- register names of the form `rNN`;
- numeric literals;
- string literals;
- comments starting with `;`.

7.3.2 Live validation

The simulator parses the code in the background while you type and reports any **errors** and **warnings** directly in the editor:

- errors are underlined in red;
- warnings are underlined with a yellow squiggle;
- hovering over an underlined region shows the description of the problem in a tooltip;
- the affected line is also marked in the editor's "minimap"-style gutter.

The same problems are summarized in the **Issues** panel on the right (see below). Warnings do not block execution; errors do — pressing `Load` while the program contains errors will surface a popup with the parser's message.

7.3.3 Hover-based information

Once a program has been parsed (i.e. once it has been loaded with the `Load` button), hovering over an instruction in the editor opens a tooltip with information about it:

- **Address** — the memory address at which the instruction has been placed.
- **OpCode** — the assembly opcode (e.g. `DADD`, `LD`).
- **Binary** — the 32-bit binary encoding.
- **Hex** — the same encoding in hexadecimal, zero-padded to 8 digits.
- **CPU Stage** — only shown if the instruction is currently inside the pipeline; identifies the stage in which it sits at the current cycle (e.g. `Instruction Fetch (IF)`, `Execute (EX)`, `FPU Multiplier (3)`).

7.3.4 Real-time pipeline-stage visualization

While the program is running, the line of source corresponding to the instruction currently in each pipeline stage is highlighted with a colour that identifies the stage. The colour code is shared with the **Pipeline** panel:

Stage	Highlight colour
Instruction Fetch (IF)	Yellow
Instruction Decode (ID)	Blue
Execute (EX)	Red
Memory Access (MEM)	Green
Write Back (WB)	Magenta
FPU Adder (1..4)	Dark green
FPU Multiplier (1..7)	Teal
FPU Divider	Olive

The highlight follows the instructions through the pipeline as the simulation advances, providing an at-a-glance view of which lines of source code are active in which stage at any given cycle. Combined with the per-instruction hover tooltip described above, this makes it easy to inspect the state of the pipeline at any point during the execution.

The editor becomes read-only while a program is loaded into the simulator. Use `Stop` to reset the CPU and edit the source again.

7.3.5 Saving and loading

The editor's contents can be persisted using the **Save Code** and **Open Code** toolbar buttons. `Save Code` triggers a download of the current source as `code.s`; `Open Code` lets you pick a local file and replaces the editor's contents with it.

In addition, the editor automatically persists its contents in the browser's local storage as you type, so an accidental page reload does not wipe a non-trivial program back to the bundled sample. The last edited source is restored the next time the page is opened in the same browser. Use the **Restore default sample** toolbar button to discard the persisted contents and bring back the original example program.

7.3.6 Optional Vi mode

A *Vi mode* for the editor can be toggled in the *General Settings* panel. When enabled, the editor honours basic vi keybindings (modes, motions, search), which is convenient if you are used to editing sources from a terminal.

7.3.7 Font size

The editor's font size can be increased or decreased from the *General Settings* panel; the chosen size is also used by other monospaced elements of the UI.

7.4 The Issues panel

The **Issues** panel on the right of the window mirrors the diagnostics that the editor surfaces inline:

- every entry shows the line and column of the problem and a short description from the parser;
- a warning icon (yellow triangle) marks warnings, an error icon (red circle) marks errors;
- the panel header shows two count chips, one for warnings and one for errors. The chips are hidden when there is nothing to report;
- every entry is clickable: selecting an issue scrolls the editor so the offending line is centred in the viewport, places the cursor at the reported column and focuses the editor so you can start fixing the problem right away.

The Issues panel is expanded by default so problems are visible at a glance.

7.5 Runtime panels

The right-hand side of the window stacks several collapsible "accordion" panels. Each panel can be expanded or collapsed independently by clicking on its header; the expansion state is persisted across page reloads.

When a panel is collapsed and its contents change because of a simulation step, a small pulsating dot appears next to the panel's title. This makes it easy to spot interesting changes (for example, a register being written) without having to keep every panel expanded. The pulsating indicator can be disabled in *General Settings* ("Accordion Change Alerts").

7.5.1 Statistics

Counters about the program execution:

- number of executed cycles;
- number of executed instructions;
- CPI — cycles per instruction (`cycles / instructions`);
- RAW, WAW and structural stalls;
- L1 instruction-cache reads and misses;

- L1 data-cache reads, read misses, writes and write misses (only meaningful when the cache simulator is configured — see *Cache Configuration*).

7.5.2 Pipeline

Shows a graphical representation of the CPU pipeline that resembles the classic Swing UI's pipeline diagram. The five integer stages (IF, ID, EX, MEM, WB) are drawn as connected blocks, with the FPU functional units — the FP Adder (4 stages), FP Multiplier (7 stages) and FP Divider — laid out around them. Each block:

- lights up with the stage's colour while it holds an instruction, and shows the instruction's mnemonic inside the block;
- stays as an empty outline when the corresponding stage is idle or holds a pipeline bubble (e.g. branch-flush slots, end-of-program drain bubbles): just like Swing's pipeline widget, bubbles are rendered as empty stages;
- renders with a hatched fill, the dedicated *Stall* colour and a short stall-type label whenever a stall actually occurred in the current cycle. The labels match the Swing cycle widget's classification:
  - **RAW** — Read-After-Write data hazard (typically on the ID stage when forwarding is disabled);
  - **WAW** — Write-After-Write hazard between two FP instructions competing for the same destination register;
  - **Struct: Div / EX / FU** — structural hazard at the FP Divider, the integer EX stage or another FP functional unit;
  - **Struct: Mem / Add / Mul** — structural hazard caused by an instruction stuck in MEM, in the FP Adder's last stage (A4) or the FP Multiplier's last stage (M7).

  WAR (Write-After-Read) hazards are *not* possible in this MIPS implementation: the in-order issue at ID combined with the late writeback at WB orders all reads before later writes, so the simulator never raises one.

Stalls are identified by the same logic that updates the CPU's stall counters, so the Web pipeline widget always agrees with the totals shown in the *Statistics* panel.

The per-stage colours (including the *Stall* colour) can be customised from *General Settings* → *Pipeline Colors* (see below) and are persisted in browser local storage.

7.5.3 Registers

The contents of the integer general-purpose registers, the floating-point registers and the FCSR are shown in this panel. Values are displayed in their hexadecimal representation; hovering on a value shows the corresponding decimal interpretation as a tooltip.

7.5.4 Memory

The current contents of the simulated main memory, organized in addressable cells. Each row shows the address (in hexadecimal) and the value stored at that address; tooltips reveal the decimal value and the source-code labels and comments associated with the cell.

7.5.5 Standard Output

A read-only text area that collects everything the program prints via `SYSCALL 4` (write integer) and `SYSCALL 5` (write string). `SYSCALL 3` (read string) is supported through a popup dialog: when the running program issues a read, an *Input* dialog appears asking for the value to feed back to the program. The dialog enforces the maximum length declared by the program and can be cancelled.

7.5.6 Cache Configuration

Lets you configure the parameters of the L1 cache simulator:

- **Size** — total capacity of the cache, in bytes;
- **Block Size** — size of a single cache line, in bytes;
- **Associativity** — number of ways per set (`1` is direct mapped, `>1` is set-associative).

The L1 instruction cache and the L1 data cache are configured independently. The fields are disabled while the simulator is running; the new configuration takes effect on the next reset.

7.5.7 General Settings

Persistent settings that influence the simulator and the UI. All values are saved in the browser's local storage and survive page reloads.

- **Editor Vi Mode** — toggles basic vi keybindings in the code editor.
- **Font Size** — font size for the code editor and other monospaced panels; can be adjusted with the `-` and `+` buttons.
- **Accordion Change Alerts** — enables or disables the pulsating indicator shown on collapsed panels when their contents change.
- **CPU Forwarding** — enables or disables operand forwarding in the pipeline. Disabled while the simulator is running because changing it requires a reset.
- **Multi Step Size** — number of cycles executed by a single click of the *Multi Step* toolbar button.
- **Execution Delay (ms)** — delay inserted between successive internal batches of cycles during *Run All*. Increasing it slows down long runs so that the visual feedback (line highlighting, panel updates) can be followed in real time. The change is applied live, even mid-execution.
- **Pipeline Colors** — per-stage colours used by the *Pipeline* diagram. Each entry (`IF`, `ID`, `EX`, `MEM`, `WB`, `FP Adder`, `FP Multiplier`, `FP Divider`, `Stall`) can be edited with a colour picker, and the *Reset to defaults* button restores the original palette (the same RGB values the Swing UI uses by default).

7.6 Running EduMIPS64 as a desktop or CLI application

The web frontend is convenient because it requires no installation, but EduMIPS64 is primarily distributed as a Java desktop application that can also be run from the command line. The desktop JAR exposes additional features (a richer settings dialog, the Dinero frontend for cache trace analysis, CLI options for batch / headless execution, a tracefile writer) that are documented in the full manual available on Read the Docs.

To install the desktop application or run EduMIPS64 from the command line, see the project's GitHub repository:

- Project page: https://www.edumips.org
- Source code, releases and installation instructions: https://github.com/EduMIPS64/edumips64

If you find a bug or want to suggest an improvement to the web frontend, please open an issue on GitHub.