The instruction set

In this section we will 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.

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.

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*

  Executes the division between 64-bits registers rs and rt, putting the 64-bits quotient in LO and the 64-bits remainder in HI.

- *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.

- *DMULHU 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.

- *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.

- *DMULXU 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 thread 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 thread 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, 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.

- *DSRLV rd, rt, rs*

  Does a right shift of 64-bits register rt, by the amount specified in low-order 6-bits of rs thread 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 to 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 to 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 thread 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 thread 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, 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.

- *SRLV rd, rt, rs*

  Does a right shift of 32-bits register rt, by the amount specified in low-order 5-bits of rs thread 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 to 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 to 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.

- *DADDU 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.

- *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.

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 a signed 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.

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.

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.

- *BGEZ rs, offset*

  If rs is greather 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.

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.

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.

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.

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 (0x0)* Opens the file in read only mode;
- *O_WRONLY (0x02)* Opens the file in write only mode;
- *O_RDWR (0x01)* 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 (0x08)* 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 fourth and the fifth modes are exclusive, you can not 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.

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.

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.

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 two or three, the simulator will pop the input/output frame, and write there the read data.

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.

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:

- %c indicates a string parameter;
- %i indicates an integer parameter;
- %f behaves like %i;
- %% literal %

For each %i, %f or %s 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.

Other instructions

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

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.

NOP

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

TRAP

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

HALT

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