The M-Processor

Instruction Set Architecture

Version 1.4

March 20, 2019

Dr. Muhamed F. Mudawar

1. Overview of the M-Processor

1.1 General-Purpose Registers

There are thirty-two general-purpose registers: **R0** to **R31**

There is **No Zero** register, which is hardwired to zero.
All general-purpose registers can be read and written.

The rationale of eliminating the ***zero register*** that exists in many RISC architectures is that the majority of instructions will not benefit from this special register. On the contrary, using the ***zero register*** as the destination will create many NOP instructions that are useless. The use of an immediate constant eliminates the need for a special ***zero*** register in many instructions.

All general-purpose registers are 64-bit wide. They can store integers, memory addresses, and floating-point data. Using general-purpose registers for both integer and floating-point data simplifies the architecture. There is no need for separate load and store instructions for integer and floating-point data. The same load and store instructions transfer all types of data. Second, it eliminates the need to copy registers between two different register files when only one register file is used. Third, the same general-purpose registers are used to pass integer and floating-point parameters to a function, which simplifies the function call convention.

1.2 Program Counter Register

The Program Counter (**PC**) register holds the current instruction address. All instructions are 32-bit long and aligned in memory. The least significant two bits of the memory address are always zero. Therefore, there is no need to store the lower two zero bits of the address in the **PC** register. Instead, the lower 2 bits indicate the current execution level **EL**, as explained in Section 7.3.

If **PC** is the address of a **JAL** instruction, then (**PC+4**) is the address of the next instruction in memory. For example, the **JAL** instruction saves (**PC+4**) in **R31**.

The **PC** register is 64-bit wide. However, a given implementation might limit the number of address bits to reduce the address space.

1.3 Exception Handling Registers

There are thirty-two separate registers for exception and interrupt handling: **E0** to **E31**. They are described in Section 7.2. A program running at the user level (or **EL0**) cannot access these registers. They are accessed only when the processor is running at supervisor level (or **EL1**).

1.4 Counter Registers

There is a third group of thirty-two registers that are used as performance counters: **C0** to **C31**. They are described in Section 7.9. A program running at the user level (or **EL0**) can read these registers.

1.5 Instruction Formats

ALL instructions are 32-bit long and aligned in memory. There are four instruction formats:

The **`J-Format`** is used mainly by Jump instructions
The **`B-Format`** is used mainly by Branch and Jump-Register instructions
The **`I-Format`** is used mainly by Load, Store, and ALU immediate instructions
The **`R-Format`** is used mainly by Register-to-Register computation instructions

	6	26
J	**Op**	**Offset26**

	6	5	5	16
B	**Op**	**a**	**b / UImm5**	**Offset16**

	6	5	5	4	12
I	**Op**	**a**	**b**	**f**	**Imm12**

	6	5	5	4	2	5	5
R	**Op**	**a**	**b**	**f**	**x**	**c**	**d**

Here is a short description of the instruction fields:

`Op:` 6-bit opcode, used in all formats
`a:` 5-bit first source register number
`b:` 5-bit second source or destination register number
`c:` 5-bit third source register number (R-Format only)
`d:` 5-bit destination register number (R-Format only)
`f:` 4-bit function code (I-Format and R-Format)
`x:` 2-bit function extension (R-Format only)

`Offset26:` 26-bit signed offset used in the J-format
`Offset16:` 16-bit signed offset used in the B-format
`UImm5:` 5-bit unsigned immediate used in the B-format
`Imm12:` 12-bit signed immediate used in the I-format

Depending on the opcode, the **`b`** field can be a second source or destination register in the **`B`** and **`I`** formats. However, the **`b`** field is always a second source register in the **`R`** format.

1.6 Software Call Conventions for General-Purpose Registers

R0 to **R31** are the names of the general-purpose registers that may appear in the assembly-language syntax. The register names are not case sensitive (**R0** or **r0** is the same register). The software call convention defines the role of these registers in the function call standard.

Registers **R0** to **R9** are used to pass parameters to a function and return results. Up to ten function parameters (integer and floating-point) can be passed in registers **R0** to **R9**. If there are more than ten parameters then the additional ones must be passed in memory on the runtime stack. Multiple results can be returned in registers **R0**, **R1**, etc. If a function wants to return a structure in memory then **R0** holds the address of the structure (or object).

Registers **R10** to **R19** are used as temporary registers. They are given special names by the assembler (**T0** to **T9** or **t0** to **t9**). In particular, register **R19** (**t9**) may be used by a linker to resolve dynamically a function call. Otherwise, it is used as a temporary register.

Registers **R0** to **R19** can be modified freely by any function without saving their values. However, if a caller function needs to preserve some of them, then it is the responsibility of the caller to save their values in its stack frame before making a function call.

Registers **R20** to **R30** are the callee-saved registers. Any function that wants to use these registers must first save their values in its stack frame before modifying them. It should also restore their original values from its stack frame before returning back to the caller. Registers **R20** to **R28** are given special names **S0** to **S8**. Register **R29** is the Frame Pointer (**FP**) that points to a stack frame. It can also be used as a saved register (**S9**). Register **R30** is the Stack Pointer (**SP**) that points to the top of the stack segment in memory.

Register **R31** is the Link Register (**LR**) that saves the return address of a function. It must be saved by a caller function before making a function call and restored before making a function return.

Register	Special Name	Role in the function call standard	Saver
R0 to R9		Function Parameters and Results	Caller
R10 to R19	**T0 to T9**	Temporary registers / Linker	Caller
R20 to R28	**S0 to S8**	Saved registers	Callee
R29	**FP or S9**	Frame Pointer	Callee
R30	**SP**	Stack Pointer	Callee
R31	**LR**	Link Register	Caller

1.7 Assembly-Language Syntax

An assembly-language program consists of a series of statements. Each statement appears on a single line. Multiple statements cannot be written on the same line.

There are only four types of statements in an assembly-language program:

- Label statement
- Directive statement
- Data allocation statement: allocates static data
- Instruction statement: one instruction is written on a line

1.7.1 Label Statement

A label statement has the following syntax:

```
@label        // Comment
```

A label (or symbol) is a user defined name. It marks the starting address of an instruction or data in memory, such as the starting address of a function or a data array. All labels must begin with the **@** character to distinguish them from reserved words. For example, **@add** is a label, while **add** is a reserved instruction name.

Labels are case sensitive. For example, **@main** and **@Main** are two different labels. A label can include letters **A** to **Z**, **a** to **z**, digits **0** to **9**, and the underscore (**_**) character. Labels that have a digit after **@**, such as **@1** and **@2a**, are valid and can be used in assembly language programs.

1.7.2 Comments

Single-line comments begin with **//** and terminate at the end of a line.
Multiline comments are enclosed between **/*** and ***/**.
Comments can appear anywhere in a source file. They are ignored by the assembler.

1.7.3 Directive Statement

Directives are reserved words that are processed by the assembler. They always begin with a dot character. For example, **.text** and **.data** are directives that tell the assembler the beginning of a text and data segment. Directives are not case sensitive: **.DATA** and **.data** are the same directive.

.text	defines a text segment that contains read-only executable instructions
.data	defines a data segment that contains read/write data (can be read and written)
.stext	defines a system text segment that contains system instructions
.sdata	defines a system data segment that contains read/write system data

Multiple text and data segments might appear in the same source file. The assembler merges multiple text segments together into one text segment, and multiple data segments into one data segment.

1.7.4 Data Allocation Statement

A data allocation statement has the following syntax:

```
[@label] .directive value [, value ...]        // Comment
```

A data allocation statement starts with an optional **@label** that marks the starting address of data, followed by a data directive, followed by a list of one or more data values. Four data allocation directives are defined that specify the size of each listed value: **.byte** (1 byte), **.hword** (half word is 2 bytes), **.word** (word is 4 bytes), and **.dword** (double word is 8 bytes). Memory is allocated by the assembler and the data is placed in a static area of the data segment.

A data allocation statement can list one or more data values, separated by commas. A data value can be an integer (signed and unsigned), a floating-point number, a character, or a string. Data values of different types can also be mixed in the same data allocation statement, as long as they have the same size. The following are examples of data allocation statements:

```
@var1   .byte     'A', -3, 0xAB            // Three bytes
@str1   .byte     "Enter an integer: "     // Null-terminated string
@var2   .hword    0xABCD:10                // 0xABCD is replicated 10 times
@var3   .word     -18, 5.7E-3              // 4-byte integer, single float
@var4   .dword    21, 21.0                 // 8-byte integer, double float
```

The string **@str1** is an array of bytes. It uses the **.byte** directive, because each character in the string is a byte. A string constant enclosed between **" "** is terminated with a null character (a zero byte). Unicode characters and strings should use the **.hword** directive when each character is 2 bytes. The **0xABCD:10** syntax means that the same **0xABCD** value is replicated **10** times. Therefore, **10** half words are allocated starting **@var2** and initialized with the same value. Integer and float-point values can appear on the same line and use the data directive, as long as they have the same size. For example, the signed integer **-18** and the floating-point value **5.7E-3** use the same **.word** directive (4-byte long), but have different binary formats. Similarly, **21** and **21.0** appear on the last line and use the **.dword** directive (8-byte long), but have different binary encodings.

1.7.5 Data Alignment

By default, the assembler aligns all data, according to their size. There is no alignment for **.byte**. However, **.hword**, **.word**, and **.dword** values are aligned. The memory addresses of the listed values are multiple of 2, 4, and 8 bytes, respectively.

The **.align n** directive changes the alignment of the next data allocation statement only, and forces its memory address to become multiple of 2^n. For example, inserting the **.align 3** directive before **@str1** increases the alignment and forces the start address of the string to become multiple of **8**.

```
.align  3                                  // Address @str1 is multiple of 8
@str1   .byte     "Enter an integer: "     // Null-terminated string
```

1.7.6 Allocating Space without Initialization

The **.space** directive allocates **N** bytes in memory, where **N** must be a positive integer constant. It does not initialize memory with zeros. The allocated space should be initialized by the program, and can be read and written.

Example of allocating space in the data segment:

```
@buffer  .space 1000                       // 1000 uninitialized bytes
```

1.7.7 Global Labels

By default, all labels defined in a file are local and visible only inside the file. The **.global** directive changes the scope of a label and makes it global. It has the following syntax:

```
.global    @label [, @label ...]
```

One or multiple labels (separated by commas) can be declared as global and visible outside the scope of the file where it is defined. This applies to function and data labels. For example, the function label **@main** and the data label **@array** are declared as global:

```
.global    @main, @array
```

If the same label is declared global in multiple files then it refers to the same entity and same memory address. One file defines a function or data and makes its label global. The other files reference the label as global. The linker (not assembler) resolves a reference to a global label at link time.

On the other hand, a reference to a local label is resolved within the file in which the label is defined. The same local label name can be redefined and might appear in different source files. It refers to different entities and memory addresses. Each file references only its local labels, but cannot access the local labels in other source files.

1.7.8 Include Directive

The **.include** directive includes the content of another file in the current file:

```
.include   "filename"
```

1.7.9 Constants

Constants can be numeric or string. Numeric constants can be integer or floating-point numbers. String constants are arrays of one or more characters. Numeric constants use the C-language syntax.

A decimal integer constant (base 10) consist of one or more decimal digits (**0** to **9**). An optional sign (**+** or **–**) can be used. Binary constants (base 2) begin with **0b** or **0B** prefix, followed by one or more binary digits (**0** or **1**). Hexadecimal constants (base 16) begin with **0x** or **0X** prefix, followed by one or more hexadecimal digits (**0** to **9**, **A** to **F**, or **a** to **f**). For example, **-5** is a decimal integer constant, **0b10011100** is a binary constant, and **0x12AB** is a hexadecimal constant.

A floating-point constant consists of an optional sign (**+** or **–**), a decimal integer part (one or more decimal digits **0** to **9**), a decimal point (**.**), a decimal fraction part (one or more decimal digits **0** to **9**), followed by an optional exponent (**e** or **E** followed by an optional sign, followed by one or more decimal digits **0** to **9**). For example, **-3.4** and **12.0E-4** are floating-point constants.

An underscore (**_**) can be inserted in a numeric constant to enhance readability. For example, **12_345_678**, **0b1101_0011**, and **-1.234_567E-2** are valid numeric constants.

A character constant is enclosed in single quotes (**' '**). The numeric constant is the ASCII value. Special characters use the escape sequence that starts with a backslash ****. Here is a list of the common escape sequences:

Escape Sequence	Character Definition	ASCII Value
\0	Null Character	**0**
\b	Backspace	**8**
\t	Tab	**9**
\n	Newline	**10**
\f	Form feed	**12**
\r	Carriage return	**13**
\'	Single Quote	**39**
\"	Double Quote	**34**
****	Backslash	**92**

A string constant is a sequence of characters enclosed in single quotes (**' '**) or double quotes (**" "**). In both cases, the string constant is an array of characters. The difference is that a string enclosed in double quotes (**" "**) is null-terminated. The assembler inserts an extra null character at the end of the double-quoted string. However, a single-quoted string is not null-terminated. The following are examples of the two strings. The first one is null-terminated, while the second one is not:

```
@str1   .byte     "String"                  // 7 bytes (Null-terminated)
@str2   .byte     'String'                  // 6 bytes (NOT terminated)
@str3   .byte     'S','t','r','i','n','g'   // 6 bytes (same as str2)
```

1.7.10 Instruction Statement

An instruction statement has the following syntax:

```
[@label]   mnemonic   [[dest =] sources]   // Comment
```

An instruction statement starts with an optional **@label** that marks the address of the instruction, followed by a mnemonic that specifies the operation, followed by an optional destination and assignment operator **=** (if any), followed by one or more source operands (if any). Commas are used to separate the source operands, which can be registers, an immediate constant, or label. The instruction syntax depends mainly on the mnemonic. Arithmetic and load instructions write to a destination register. Store instructions write to memory. Load and store instructions have a unique syntax of enclosing the memory address between square brackets. Branch and Jump instructions do not have a destination register and use labels to specify the target address. The following are examples:

```
ADD      r5 = r1, r3                // r5 = r1 + r3
ADD      r5 = r3, 125               // r5 = r3 + 125
ADD      r5 = r1, r2, r3            // r5 = r1 + r2 + r3
LD       t0 = [r4, 40]              // r10 = MEM8[r4 + 40]
LD       t0 = [r4, r6, 3]           // r10 = MEM8[r4 + r6<<3]
SD       [r4, r6] = s0              // MEM8[r4 + r6] = r20
J        @next                      // PC = @next
JAL      @f                         // r31 = (PC+4); PC = @f
BEQ      r1, r2, @loop              // PC = (r1==r2)? @loop:(PC+4)
ADD.S    r5 = r1, r3                // Single-Precision Add
ADD.D    r5 = r1, r3                // Double-Precision Add
```

Mnemonics are predefined assembly-language names for machine and pseudo instructions. They are not case-sensitive. For example, **ADD** and **add** are the same instruction mnemonic. Some mnemonics have suffixes that indicate the type of operation. The suffix always starts with a dot (**.**). For example, the **ADD.S** and **ADD.D** instructions use the **.S** and **.D** suffixes to indicate single and double-precision floating-point formats and operations.

Register names are also reserved. They are not case-sensitive. Register **R0** and **r0** are the same. No dollar sign or any other symbol should be used: **$r0** or **$0** are **invalid** register names. Some registers have aliases. For example, **t0** is **r10**, **s0** is **r20**, and **sp** is **r30**.

Instruction mnemonics are highly readable and can be overloaded. In the above example, the same **ADD** mnemonic uses three different opcodes. The first instruction uses the **ALU** opcode, the second one uses a different **ALUI** opcode with an immediate constant, and the third uses an **ALU3** opcode with three source registers. The assembler translates each instruction properly, based on its syntax. The assembler should generate a **syntax error** if an invalid mnemonic, such as **ADDI**, is used.

2. Control Flow Instructions

Control flow instructions modify the program counter (**PC**) register. These include the unconditional jump, conditional branch, and register indirect jump instructions.

2.1 Unconditional Jump Instructions (J-Format)

	6	26
J	**Op**	**Offset26**

Two Opcodes: **Op = J, JAL**

Assembly Language Syntax:

```
J       @label                  // Jump @label
JAL     @label                  // R31 = (PC+4); Jump @label
```

The **JAL** instruction saves the return address **(PC+4)** in **R31** (implicit register)

Instructions are always aligned in memory.
No instruction misalignment exception can occur.

PC-relative addressing for position-independent code:

Offset26: 26-bit PC-relative signed offset

Jump Target Address = PC + sign_extend(Offset26<<2)
Jump Address Range = $\pm 2^{25}$ **Instructions = ±128 MiBytes**

R31 = Implicit link register for the **JAL** instruction
R31 = PC+4 = Address of next instruction appearing after **JAL**

Assembly-Language Syntax Note:

Mnemonics are not case-sensitive: **JAL** and **jal** are the same instruction
All labels must begin with the **@** symbol
Labels are case sensitive: **@label** and **@Label** are two different labels
Single-line comments begin with **//** and terminate at the end of the line

2.2 Register-to-Register Branch Instructions (B-Format)

	6	5	5	16
B	**Op**	**a**	**b**	**Offset16**

Six Opcodes: **Op = BEQ, BNE, BLT, BGE, BLTU, BGEU**

Compare Two Registers and Branch accordingly
Assembly Language Syntax:

```
BEQ     Ra, Rb, @label            // if (Ra == Rb) branch @label
BNE     Ra, Rb, @label            // if (Ra != Rb) branch @label
BLT     Ra, Rb, @label            // if (Ra <s Rb) branch @label
BGE     Ra, Rb, @label            // if (Ra ≥s Rb) branch @label
BLTU    Ra, Rb, @label            // if (Ra <u Rb) branch @label
BGEU    Ra, Rb, @label            // if (Ra ≥u Rb) branch @label
```

BEQ and **BNE** compare the bits of any data type
BLT and **BGE** compare signed integers
BLTU and **BGEU** compare unsigned integers

Ra: Value of first source register **a**
Rb: Value of second source register **b**

Offset16: 16-bit PC-relative signed offset for position-independent code

Branch Target Address = PC + sign_extend(Offset16<<2)
Branch Address Range = ±2^{15} Instructions = ±2^{17} Bytes

Branch Pseudo-Instructions

The following pseudo-instructions reverse the order of **Ra** and **Rb** in the assembly-language:

```
BGT     Rb, Ra, @label            // Pseudo: BLT  Ra, Rb, @label
BLE     Rb, Ra, @label            // Pseudo: BGE  Ra, Rb, @label
BGTU    Rb, Ra, @label            // Pseudo: BLTU Ra, Rb, @label
BLEU    Rb, Ra, @label            // Pseudo: BGEU Ra, Rb, @label
```

2.3 Register-Immediate Branch Instructions (B-Format)

	6	5	5	16
B	**Op**	**a**	**UImm5**	**Offset16**

Six Opcodes: **Op = BEQI, BNEI, BLTI, BGEI, BLTUI, BGEUI**

Compare Register with Immediate and Branch accordingly
Assembly-Language Syntax:

```
BEQ     Ra, UImm5, @label        // if (Ra == UImm5) branch @label
BNE     Ra, UImm5, @label        // if (Ra != UImm5) branch @label
BLT     Ra, UImm5, @label        // if (Ra <s UImm5) branch @label
BGE     Ra, UImm5, @label        // if (Ra ≥s UImm5) branch @label
BLTU    Ra, UImm5, @label        // if (Ra <u UImm5) branch @label
BGEU    Ra, UImm5, @label        // if (Ra ≥u UImm5) branch @label
```

Ra: Value of first source register **a**
UImm5: 5-bit unsigned immediate with range **0** to **31**.

Offset16: 16-bit PC-relative signed offset for position-independent code

Branch Target Address = PC + sign_extend(Offset16<<2)
Branch Address Range = $\pm 2^{15}$ **Instructions =** $\pm 2^{17}$ **Bytes**

Assembly-Language Note:

The same mnemonics are used for register-register and register-immediate branch instructions. The assembler recognizes the opcode (**BEQ** or **BEQI**) based on the instruction syntax.

Compare with Zero and Branch Pseudo-Instructions

```
BEQZ    Ra, @label               // Pseudo: BEQ  Ra, 0, @label
BNEZ    Ra, @label               // Pseudo: BNE  Ra, 0, @label
BLTZ    Ra, @label               // Pseudo: BLT  Ra, 0, @label
BGEZ    Ra, @label               // Pseudo: BGE  Ra, 0, @label
BLEZ    Ra, @label               // Pseudo: BLT  Ra, 1, @label
BGTZ    Ra, @label               // Pseudo: BGE  Ra, 1, @label
```

Compare with Immediate (0 to 30) and Branch Pseudo-Instructions

```
BLE     Ra, I, @label            // Pseudo: BLT  Ra, I+1, @label
BGT     Ra, I, @label            // Pseudo: BGE  Ra, I+1, @label
BLEU    Ra, I, @label            // Pseudo: BGEU Ra, I+1, @label
BGTU    Ra, I, @label            // Pseudo: BLTU Ra, I+1, @label
```

2.4 LOOP Instructions (B-Format)

	6	5	5	16
B	**Op**	**a**	**b**	**Offset16**

Two Opcodes: **Op = LOOP, LOOPD**

The **LOOP** and **LOOPD** instructions are useful in counter-controlled loops, where **Rb** is a counter.

The **LOOP** instruction increments **Rb**, and then compares its incremented value against **Ra**.
The **LOOPD** instruction decrements **Rb**, and then compares its decremented value against **Ra**.

Assembly-Language Syntax:

```
LOOP     Rb, Ra, @label            // Rb++; if (Rb != Ra) branch @label
LOOPD    Rb, Ra, @label            // Rb--; if (Rb != Ra) branch @label
```

Ra: Value of source register **a**
Rb: Source and destination register **b**

Offset16: 16-bit PC-relative signed offset for position-independent code

Branch Target Address = PC + sign_extend(Offset16<<2)
Branch Address Range = $\pm 2^{15}$ **Instructions =** $\pm 2^{17}$ **Bytes**

Assembly-Language Note:

LOOP and **LOOPD** are similar to **BNE**, except that they pre-increment or pre-decrement **Rb**.
Rb appears before **Ra** in the **LOOP** and **LOOPD** instructions because it is a destination register.

Examples on the **LOOP** and **LOOPD** instructions:

Loop	**Translation**

`for (r2 = 0; r2 < 100; r2++) {`
`  . . .`
`}`
	`set      r2 = 0`
	`set      r3 = 100`
	`@for`
	`  . . .`
	`loop     r2, r3, @for`

`for (r4 = 99; r4 > 1; r2--) {`
`  . . .`
`}`
	`set      r4 = 99`
	`set      r5 = 1`
	`@for`
	`  . . .`
	`loopd    r4, r5, @for`

2.5 Jump-Register Instructions (B-Format)

	6	5	5	16
B	**Op = JR**	**a**	**//**	**Offset16**
B	**Op = JALR**	**a**	**b**	**Offset16**

Two Opcodes: **Op = JR, JALR**

Assembly Language Syntax:

```
JR      Ra, Offset16
JALR    Rb, Ra, Offset16
```

Ra: Value of source register **a**
Rb: Destination register **b** (for **JALR** instruction)

Offset16: 16-bit signed offset. If omitted, it defaults to zero.

JR jumps to indirect address: **PC[63:2] = Ra[63:2] + sign_extend(Offset16)**
The **JR** instruction ignores the **b** field.

JALR (Jump-and-Link-Register) does two things:
1. Saves the return address in a destination register: **Rb = (PC+4)**
2. Jump to indirect address: **PC[63:2] = Ra[63:2] + sign_extend(Offset16)**

The **JR** and **JALR** instructions do not modify the lower 2 bits of the **PC** register.
The lower two bits of **PC** specify the current execution level: **PC[1:0] = EL** (Section 7.3).

Assembly-Language Notes:

The **JR** instruction can be used as a function return or an indirect jump.
If **Ra** is **R31** (return address) then **JR** does a function return. Otherwise, **JR** does an indirect jump.

The **JALR** instruction does an indirect function call and saves the return address in **Rb**.
Rb appears before **Ra** in the **JALR** instruction because it is a destination register.

Jump-Register Pseudo-Instructions

If **Offset16** is omitted, it defaults to zero.

```
JR      Ra                      // Pseudo: JR   Ra, 0
RET                             // Pseudo: JR   R31, 0
JALR    Rb, Ra                  // Pseudo: JALR Rb = Ra, 0
```

2.6 Summary of Opcodes for Control-Flow Instructions

The opcode space is not fully defined. Many opcodes are reserved for future expansion of the architecture. The **Invalid Instruction exception** should be raised if an undefined opcode is used.

	6	5	5	16
J	J = 2	Offset26		
J	JAL = 3	Offset26		
B	BEQI = 8	a	UImm5	Offset16
B	BNEI = 9	a	UImm5	Offset16
B	BLTI = 10	a	UImm5	Offset16
B	BGEI = 11	a	UImm5	Offset16
B	BLTUI = 12	a	UImm5	Offset16
B	BGEUI = 13	a	UImm5	Offset16
B	JR = 14	a	//	Offset16
B	JALR = 15	a	b	Offset16
B	BEQ = 16	a	b	Offset16
B	BNE = 17	a	b	Offset16
B	BLT = 18	a	b	Offset16
B	BGE = 19	a	b	Offset16
B	BLTU = 20	a	b	Offset16
B	BGEU = 21	a	b	Offset16
B	LOOP = 22	a	b	Offset16
B	LOOPD = 23	a	b	Offset16

3. Memory Instructions

Memory instructions transfer data between memory and registers. The load instructions read data from memory. The store instructions write data in memory.

3.1 Load Instructions (I-Format)

	6	5	5	4	12
I	Op = LOAD	a	b	f	Imm12

One Opcode: **Op = LOAD**
Eight Functions: **f = LB, LH, LW, LD, LBU, LHU, LWU, LDU**

Load Byte, Half-word, Word, or Double-word
Transfer 1, 2, 4, or 8 bytes from memory into destination register **b**
Sign-extend or zero-extend the data when loaded

Assembly-Language Syntax:

```
LB      Rb = [Ra, Imm12]          // Rb ← sign_extend(MEM1[Ra+Imm12])
LH      Rb = [Ra, Imm12]          // Rb ← sign_extend(MEM2[Ra+Imm12])
LW      Rb = [Ra, Imm12]          // Rb ← sign_extend(MEM4[Ra+Imm12])
LD      Rb = [Ra, Imm12]          // Rb ← MEM8[Ra+Imm12]

LBU     Rb = [Ra, Imm12]          // Rb ← zero_extend(MEM1[Ra+Imm12])
LHU     Rb = [Ra, Imm12]          // Rb ← zero_extend(MEM2[Ra+Imm12])
LWU     Rb = [Ra, Imm12]          // Rb ← zero_extend(MEM4[Ra+Imm12])
LDU     Rb = [Ra, Imm12]          // Rb ← MEM8[Ra+Imm12]
```

Ra: Value of address register **a**
Rb: Destination register **b**

The **LB**, **LH**, and **LW** instructions sign-extend the data when loaded into **Rb**.
The **LBU**, **LHU**, and **LWU** instructions zero-extend the data when loaded into **Rb**.
The **LD** and **LDU** instructions load 8 bytes into **Rb**. They are both listed for convenience.

Displacement addressing is used for all **LOAD** instructions:

Effective Address is: **EA = Ra + sign_extend(Imm12)**
If **Imm12** is omitted, it defaults to 0

Note: The 4-bit function code **f** can define up to 16 load instructions. However, only 8 are defined. The remaining function codes are reserved for future use.

3.2 Store Instructions (I-Format)

	6	5	5	4	12
I	**Op = STORE**	**a**	**b**	**f**	**Imm12**

One Opcode: **`Op = STORE`**
Four Functions: **`f = SB, SH, SW, SD`**

Store Byte, Half-word, Word, Double-word
Transfer 1, 2, 4, or 8 bytes from source register **b** into memory

Assembly-Language Syntax:

```
SB      [Ra, Imm12] = Rb          // MEM1[Ra + Imm12] ← lower1(Rb)
SH      [Ra, Imm12] = Rb          // MEM2[Ra + Imm12] ← lower2(Rb)
SW      [Ra, Imm12] = Rb          // MEM4[Ra + Imm12] ← lower4(Rb)
SD      [Ra, Imm12] = Rb          // MEM8[Ra + Imm12] ← Rb
```

Ra: Value of address register **a**
Rb: Value of source register **b**

SB, **SH**, and **SW** write the lower 1, 2, and 4 bytes of **Rb** in memory
SD writes all 8 bytes of **Rb** in memory

Displacement addressing is used for all **STORE** instructions:

Imm12: 12-bit signed displacement
Effective Address is: **`EA = Ra + sign_extend(Imm12)`**
If **Imm12** is omitted, it defaults to 0

Note: The 4-bit function code **f** can define up to 16 store instructions. However, only 4 are defined. The remaining function codes are reserved for future use.

3.3 Indexed Load Instructions (R-Format)

	6	5	5	4	2	5	5
R	**Op = LOADX**	**a**	**b**	**f**	**s**	**//**	**d**

One Opcode: **Op = LOADX** (The **c** field is ignored)
Eight Functions: **f = LB, LH, LW, LD, LBU, LHU, LWU, LDU**

Load Byte, Half-word, Word, or Double-word
Transfer 1, 2, 4, or 8 bytes from memory into destination register **d**
Sign-extend or zero-extend the data when loaded

Ra: Value of address register **a**
Rb: Value of index register **b**
Rd: Destination register **d**
s: Scale factor = **0**, **1**, **2**, or **3**

Assembly-Language Syntax:

```
LB      Rd = [Ra, Rb, s]         // Rd ← sign_extend(MEM1[Ra+Rb<<s])
LH      Rd = [Ra, Rb, s]         // Rd ← sign_extend(MEM2[Ra+Rb<<s])
LW      Rd = [Ra, Rb, s]         // Rd ← sign_extend(MEM4[Ra+Rb<<s])
LD      Rd = [Ra, Rb, s]         // Rd ← MEM8[Ra+Rb<<s]

LBU     Rd = [Ra, Rb, s]         // Rd ← zero_extend(MEM1[Ra+Rb<<s])
LHU     Rd = [Ra, Rb, s]         // Rd ← zero_extend(MEM2[Ra+Rb<<s])
LWU     Rd = [Ra, Rb, s]         // Rd ← zero_extend(MEM4[Ra+Rb<<s])
LDU     Rd = [Ra, Rb, s]         // Rd ← MEM8[Ra+Rb<<s]
```

The **LB**, **LH**, and **LW** instructions sign-extend the data when loaded into **Rd**.
The **LBU**, **LHU**, and **LWU** instructions zero-extend the data when loaded into **Rd**.
The **LD** and **LDU** instructions load 8 bytes into **Rd**. They are both listed for convenience.

Scaled-Index addressing is used for all **LOADX** instructions:

Effective Memory Address is: **EA = Ra + Rb<<s**
If **s** is omitted, it defaults to **0**

Assembly-Language Note:
The same mnemonics are used for **LOAD** and **LOADX** instructions
The assembler recognizes the opcode (**LOAD** or **LOADX**) based on the instruction syntax

3.4 Indexed Store Instructions (R-Format)

	6	5	5	4	2	5	5
R	**Op = STORX**	**a**	**b**	**f**	**s**	**c**	**//**

One Opcode: **Op = STORX** (The **d** field is ignored)
Four Functions: **f = SB, SH, SW, SD**

Store Byte, Half-word, Word, Double-word
Transfer 1, 2, 4, or 8 bytes from source (data) register **c** into memory

Ra: Value of address register **a**
Rb: Value of index register **b**
Rc: Value of data register **c**
s: Scale factor = **0**, **1**, **2**, or **3**

Assembly-Language Syntax:

```
SB      [Ra, Rb, s] = Rc          // MEM1[Ra + Rb<<s] ← lower1(Rc)
SH      [Ra, Rb, s] = Rc          // MEM2[Ra + Rb<<s] ← lower2(Rc)
SW      [Ra, Rb, s] = Rc          // MEM4[Ra + Rb<<s] ← lower4(Rc)
SD      [Ra, Rb, s] = Rc          // MEM8[Ra + Rb<<s] ← Rc
```

SB, **SH**, and **SW** write the lower 1, 2, and 4 bytes of **Rc** into memory
SD writes all 8 bytes of **Rc** into memory

Scaled-Index addressing is used for all **STORX** instructions:

Effective Memory Address is: **EA = Ra + Rb<<s**
If **s** is omitted, it defaults to 0

Assembly-Language Note:
The same mnemonics are used for **STORE** and **STORX** instructions
The assembler recognizes the opcode (**STORE** or **STORX**) based on the instruction syntax

3.5 Memory Alignment for Load and Store Instructions

Memory alignment is enforced on all Load and Store memory addresses:

The memory address of the **LH**, **LHU**, and **SH** instructions must be multiple of **2**.
The memory address of the **LW**, **LWU**, and **SW** instructions must be multiple of **4**.
The memory address of the **LD**, **LDU**, and **SD** instructions must be multiple of **8**.

If the effective memory address is not aligned, then the **address misalignment exception** is raised.

Byte Ordering:

Little Endian byte ordering is used by all Load and Store instructions
The bytes are loaded/stored starting at the least-significant byte

3.6 Summary of Opcodes and Function Codes for Memory Instructions

The **LOAD** and **STORE** opcodes use the I-Format.
Register **a** contains the base address.
Register **b** is a destination for **LOAD**, but a source data register for **STORE**.

The **LOADX** and **STORX** opcodes use the R-Format.
Register **a** contains the base address.
Register **b** contains the index register and **s** is a 2-bit scale factor.
Register **d** is a destination for **LOADX**. It is ignored by **STORX**.
Register **c** contains the source data for **STORX**. It is ignored by **LOADX**.

	6	5	5	4	12		
I	LOAD = 24	a	b	f	Imm12		
I	STORE = 25	a	b	f	Imm12		
R	LOADX = 26	a	b	f	s	//	d
R	STORX = 27	a	b	f	s	c	//

The **LOAD** and **LOADX** opcodes use identical function codes. The **STORE** and **STORX** opcodes also use identical function codes. The function codes are listed in the following table:

	6-bit Opcode	**f** = 4-bit function codes for **LOAD**, **STORE**, **LOADX**, and **STORX**			
I	LOAD = 24	LBU = 0	LHU = 1	LWU = 2	LDU = 3
I	LOAD = 24	LB = 4	LH = 5	LW = 6	LD = 7
I	STORE = 25	SB = 0	SH = 1	SW = 2	SD = 3
R	LOADX = 26	LBU = 0	LHU = 1	LWU = 2	LDU = 3
R	LOADX = 26	LB = 4	LH = 5	LW = 6	LD = 7
R	STORX = 27	SB = 0	SH = 1	SW = 2	SD = 3

4. Integer Instructions

These include integer arithmetic, bitwise logic, integer compare, shift, rotate, integer multiply, and divide instructions.

4.1 ALU Instructions (R-Format)

	6	5	5	4	2	5	5
R	**Op = ALU**	**a**	**b**	**f**	**x=0**	**//**	**d**

Opcode: **Op = ALU** (**c** field is ignored)
Eight functions: **f = ADD, AND, OR, XOR, NADD, CAND, COR, XNOR** (with **x=0**)

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rd: Destination register **d**

Assembly Language Syntax:

```
ADD     Rd = Ra, Rb               // Rd = Ra + Rb
AND     Rd = Ra, Rb               // Rd = Ra & Rb
OR      Rd = Ra, Rb               // Rd = Ra | Rb
XOR     Rd = Ra, Rb               // Rd = Ra ^ Rb
```

Negate (**N**) and Complement (**C**) operate on the **Ra** value:

```
NADD    Rd = Ra, Rb               // Rd = -Ra + Rb
CAND    Rd = Ra, Rb               // Rd = ~Ra & Rb
COR     Rd = Ra, Rb               // Rd = ~Ra | Rb
XNOR    Rd = Ra, Rb               // Rd = ~Ra ^ Rb
```

The **ADD** and **NADD** instructions are used for 64-bit signed/unsigned addition. They do not cause any exception in the case of overflow.

The rationale of negating and complementing **Ra** (rather than **Rb**) is because these same functions are used in the immediate format, where **Rb** is replaced with a signed immediate (Section 4.3).

ALU Pseudo-Instructions (R-Format)

The following pseudo-instructions reverse the order of **Ra** and **Rb** in the assembly-language syntax:

```
SUB     Rd = Rb, Ra               // Pseudo: NADD Rd = Ra, Rb
ANDC    Rd = Rb, Ra               // Pseudo: CAND Rd = Ra, Rb
ORC     Rd = Rb, Ra               // Pseudo: COR  Rd = Ra, Rb
```

4.2 ALU Compare Instructions (R-Format)

	6	5	5	4	2	5	5
R	**Op = ALU**	**a**	**b**	**f**	**x=0**	**//**	**d**

Same **ALU** opcode (**c** field is ignored)
Eight additional functions: **f = EQ, NE, LT, GE, LTU, GEU, MIN, MAX** (with **x=0**)

```
EQ      Rd = Ra, Rb                 // Rd = (Ra == Rb)
NE      Rd = Ra, Rb                 // Rd = (Ra != Rb)
LT      Rd = Ra, Rb                 // Rd = (Ra <s Rb)
GE      Rd = Ra, Rb                 // Rd = (Ra ≥s Rb)
LTU     Rd = Ra, Rb                 // Rd = (Ra <u Rb)
GEU     Rd = Ra, Rb                 // Rd = (Ra ≥u Rb)
MIN     Rd = Ra, Rb                 // Rd = MIN(Ra, Rb)
MAX     Rd = Ra, Rb                 // Rd = MAX(Ra, Rb)
```

EQ and **NE** compare the bits of any data type.
LT does signed integer **<s** comparison, while **LTU** does unsigned integer **<u** comparison.
GE does signed integer **≥s** comparison, while **GEU** does unsigned integer **≥u** comparison.

The result of any compare instruction is either **0** (false) or **1** (true).

ALU Compare Pseudo-Instructions (R-Format)

The following pseudo-instructions reverse the order of **Ra** and **Rb** in the assembly-language syntax:

```
GT      Rd = Rb, Ra                 // Pseudo: LT   Rd = Ra, Rb
LE      Rd = Rb, Ra                 // Pseudo: GE   Rd = Ra, Rb
GTU     Rd = Rb, Ra                 // Pseudo: LTU  Rd = Ra, Rb
LEU     Rd = Rb, Ra                 // Pseudo: GEU  Rd = Ra, Rb
```

4.3 ALU Instructions (I-Format)

	6	5	5	4	12
I	Op = ALUI	a	b	f	Imm12

Opcode: **Op = ALUI**
Eight functions: **f = ADD, AND, OR, XOR, NADD, CAND, COR, SET**

Ra: Value of source register **a**
Rb: Destination register **b**
Imm12: 12-bit signed immediate

Assembly Language Syntax:

```
ADD     Rb = Ra, Imm12          // Rb =  Ra + sign_extend(Imm12)
AND     Rb = Ra, Imm12          // Rb =  Ra & sign_extend(Imm12)
OR      Rb = Ra, Imm12          // Rb =  Ra | sign_extend(Imm12)
XOR     Rb = Ra, Imm12          // Rb =  Ra ^ sign_extend(Imm12)
```

Negate (**N**) and Complement (**C**) operate on the **Ra** value:

```
NADD    Rb = Ra, Imm12          // Rb = -Ra + sign_extend(Imm12)
CAND    Rb = Ra, Imm12          // Rb = ~Ra & sign_extend(Imm12)
COR     Rb = Ra, Imm12          // Rb = ~Ra | sign_extend(Imm12)
SET     Rb = Imm12              // Rb = sign_extend(Imm12)
```

The **ADD** and **NADD** instructions are used for 64-bit signed/unsigned addition. They do not cause any exception in the case of overflow.

The **SET** instruction initializes **Rb** with an immediate (the **Ra** value is not used and ignored).

To enhance readability, the same mnemonics (**ADD**, **AND**, etc.) are used in the I-format and R-format ALU instructions. The assembler recognizes the opcode (**ALUI** or **ALU**) based on the syntax.

ALU Pseudo-Instructions (I-Format)

The following pseudo-instructions change the value of the immediate or use a zero immediate:

```
SUB     Rb = Ra, Imm12          // Pseudo: ADD  Rb = Ra, -Imm12
ANDC    Rb = Ra, Imm12          // Pseudo: ADD  Rb = Ra, ~Imm12
ORC     Rb = Ra, Imm12          // Pseudo: OR   Rb = Ra, ~Imm12
XNOR    Rb = Ra, Imm12          // Pseudo: XOR  Rb = Ra, ~Imm12
MOV     Rb = Ra                 // Pseudo: OR   Rb = Ra, 0
NEG     Rb = Ra                 // Pseudo: NADD Rb = Ra, 0
NOT     Rb = Ra                 // Pseudo: COR  Rb = Ra, 0
```

4.4 ALU Compare Instructions (I-Format)

	6	5	5	4	12
I	**Op = ALUI**	**a**	**b**	**f**	**Imm12**

Same **`ALUI`** opcode
Eight additional functions: **`f = EQ, NE, LT, GE, LTU, GEU, MIN, MAX`**

`Ra:` Value of source register **`a`**
`Rb:` Destination register **`b`**
`Imm12:` 12-bit signed immediate

Assembly Language Syntax:

```
EQ     Rb = Ra, Imm12          // Rb = (Ra == sign_extend(Imm12))
NE     Rb = Ra, Imm12          // Rb = (Ra != sign_extend(Imm12))
LT     Rb = Ra, Imm12          // Rb = (Ra <s sign_extend(Imm12))
GE     Rb = Ra, Imm12          // Rb = (Ra ≥s sign_extend(Imm12))
LTU    Rb = Ra, Imm12          // Rb = (Ra <u sign_extend(Imm12))
GEU    Rb = Ra, Imm12          // Rb = (Ra ≥u sign_extend(Imm12))
MIN    Rb = Ra, Imm12          // Rb = MIN(Ra, sign_extend(Imm12))
MAX    Rb = Ra, Imm12          // Rb = MAX(Ra, sign_extend(Imm12))
```

`EQ` and **`NE`** compare the bits of any data type.
`LT` does signed integer **`<s`** comparison, while **`LTU`** does unsigned integer **`<u`** comparison.
`GE` does signed integer **`≥s`** comparison, while **`GEU`** does unsigned integer **`≥u`** comparison.

The immediate is always sign-extended, but is interpreted as unsigned by **`LTU`** and **`GEU`**.
The result of any compare instruction is either **`0`** (false) or **`1`** (true).

To enhance readability, the same mnemonics (**`EQ`**, **`NE`**, etc.) are used in the I-format and R-format ALU instructions. The assembler recognizes the opcode (**`ALUI`** or **`ALU`**) based on the syntax.

ALU Compare Pseudo-Instructions (I-Format)

```
GT     Rb = Ra, Imm12          // Pseudo: GE   Rb = Ra, (Imm12 + 1)
LE     Rb = Ra, Imm12          // Pseudo: LT   Rb = Ra, (Imm12 + 1)
GTU    Rb = Ra, Imm12          // Pseudo: GEU  Rb = Ra, (Imm12 + 1)
LEU    Rb = Ra, Imm12          // Pseudo: LTU  Rb = Ra, (Imm12 + 1)
```

The order of **`Ra`** and **`Imm12`** cannot be reversed in the I-Format. To define the **`GT`**, **`LE`**, **`GTU`**, and **`LEU`** pseudo-instructions, **`Imm12`** must be incremented in the **`GE`**, **`LT`**, **`GEU`**, and **`LTU`** instructions.

4.5 ALU Instructions with Long Immediate (I-Format)

The ALU I-Format encodes a 12-bit signed immediate, which might be sufficient for many instructions. However, for some instructions a larger constant might be required. Instead of loading a large constant from memory, it is better to encode a long immediate (up to 64 bits) as part of the instruction. To achieve this, three new opcodes are defined:

Three Opcodes: **Op = ALUI1, ALUI2, ALUI3**

If the opcode is **ALUI1** or **ALUI2**, then the next instruction in memory must be a **NOP** with a 26-bit immediate. Otherwise, the **Invalid Instruction exception** is raised.

	6	5	5	4	12
I	**ALUI1/ALUI2**	**a**	**b**	**f**	**Imm12**
	NOP	**Imm26A**			

If the opcode is **ALUI3**, then the next two instructions must both be **NOPs**, each carrying a 26-bit immediate. Otherwise, the **Invalid Instruction exception** is raised.

	6	5	5	4	12
I	**ALUI3**	**a**	**b**	**f**	**Imm12**
	NOP	**Imm26A**			
	NOP	**Imm26B**			

The 64-bit immediate (**Imm64**) is defined differently for **ALUI1**, **ALUI2**, and **ALUI3**

ALUI1 ➔ Imm64 = sign_extend(Imm26A:Imm12), where **:** means concatenation of bits.
ALUI2 ➔ Imm64 = (Imm26A:26b0:Imm12), where **26b0** means **26** zero bits.
ALUI3 ➔ Imm64 = (Imm26A:Imm26B:Imm12), where **:** means concatenation of bits.

If the inner 26 bits (with bit range **[37:12]**) of an immediate are all zeros and at least one of the upper 26 bits (bit range **[63:38]**) is non-zero, then **ALUI2** provides a more compact way to encode the 64-bit immediate than **ALUI3**, which requires two **NOP** instructions.

Important Notes:

The same 4-bit function **f** is used with all four opcodes: **ALUI**, **ALUI1**, **ALUI2**, and **ALUI3**.

The same assembly-language mnemonics are used with: **ALUI**, **ALUI1**, **ALUI2**, and **ALUI3**.

A generic signed immediate **Imm** (up to 64 bits) can be used in any ALU immediate instruction.

The assembler chooses the proper opcode (**ALUI**, **ALUI1**, **ALUI2**, or **ALUI3**) depending on the value of the immediate to minimize the number of **NOP**s and the size of the code.

If more **NOP**s appear after **ALUI1**, **ALUI2**, or **ALUI3** than needed by the instruction, then the additional ones have no effect and are discarded by the instruction execution pipeline.

4.6 Function Return (I-Format)

	6	5	5	4	12
I	**Op = RET**	**a**	**b**	**f**	**Imm12**

Although function return is a control instruction, it can be combined with any **ALUI** (I-format) operation. This is frequent in programming when returning a result in a register, or modifying the stack pointer before return.

Opcode: **Op = RET**

The function codes used in **RET** are identical to those defined by **ALUI**:
f = ADD, AND, OR, XOR, NADD, CAND, COR, SET, EQ, NE, LT, GE, LTU, GEU, MIN, MAX

Assembly Language Syntax:

```
RETOP    Rb = Ra, Imm12            // JR R31; OP Rb = Ra, Imm12
```

Ra: Value of source register **a**
Rb: Destination register **b**

Imm12: 12-bit signed immediate

RET instructions execute the following two operations:
1. Jump to the return address: **PC[63:2] = R31[63:2]**
2. Execute any **ALUI** function as specified by the **f** field

R31 is an implicit register in all **RET** instructions.

The **RET** instructions do not modify the lower 2 bits of the **PC** register.
The lower two bits of **PC** specify the current execution level: **PC[1:0] = EL** (Section 7.3).

Examples on the use of the Return instruction:

```
RETSET   R0 = -1                   // PC = R31; R0 = -1
RETADD   SP = SP, 32               // PC = R31; SP = SP + 32
```

The first instruction returns a constant value in register **R0**. The second one updates the stack pointer to free the stack frame, just before returning to the caller. Combining a simple ALU operation with a function return is frequent in programs.

4.7 Shift and Rotate Instructions (I-Format)

	6	5	5	4	6	6
I	**Op = SHIFT**	**a**	**b**	**f**	**l = Imm6**	**r = Imm6**

Opcode: **Op = SHIFT**

Three functions: **f = SHLR, SALR, ROL**

Ra: Value of source register **a**
Rb: Destination register **b**

The 12-bit immediate is divided into two 6-bit fields: **l** and **r**

l: 6-bit left-shift amount (value is 0 to 63)
r: 6-bit right-shift amount (value is 0 to 63)

Assembly Language Syntax:

```
SHLR    Rb = Ra, l, r         // Shift Left then Right
SALR    Rb = Ra, l, r         // Shift Arithmetic Left then Right
ROR     Rb = Ra, r            // Rotate Right
```

The **SHLR** instruction shifts the **Ra** value left by **l** bits, then right by **r** bits. Zeros are inserted when shifting left and right. It is equivalent to two shift operations **SHL** (shift left) and **SHR** (shift right):

SHLR Rb = Ra, l, r ➔ SHL Temp = Ra, l; SHR Rb = Temp, r

The **SALR** instruction does arithmetic shift right. Zeros are inserted when shifting left. However, the most-significant bit of the left-shifted (**Temp**) value is replicated when shifting right. It is also equivalent to two operations, where **SHL** is shift left and **SAR** is shift arithmetic right:

SALR Rb = Ra, l, r ➔ SHL Temp = Ra, l; SAR Rb = Temp, r

The **ROR** instruction rotates the **Ra** value according to the right shift amount **r**. The **l** shift amount is not used. The least-significant bits of **Ra** are rotated to become the most significant bits of the result.

Shift and Rotate Pseudo-Instructions

Shift Left (**SHL**), Shift Right (**SHR**), Shift Arithmetic Right (**SAR**), and Rotate Right (**ROR**):

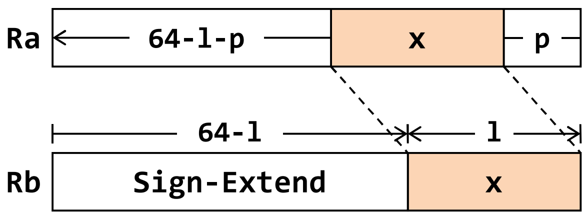
```
SHL     Rb = Ra, l              // Pseudo: SHLR  Rb = Ra, l, 0
SHR     Rb = Ra, r              // Pseudo: SHLR  Rb = Ra, 0, r
SAR     Rb = Ra, r              // Pseudo: SALR  Rb = Ra, 0, r
ROL     Rb = Ra, r              // Pseudo: ROR   Rb = Ra, 64-r
```

Extract and Extend Pseudo-Instructions

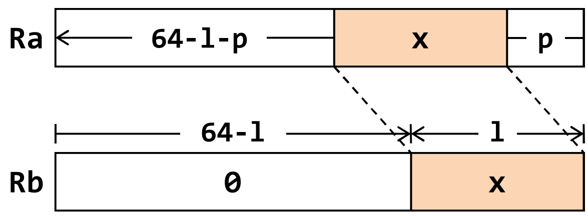
The **SHLR** and **SALR** instructions can be used to extract a bit field. The **EXTR** and **EXTRU** pseudo-instructions extract a signed/unsigned bit field from **Ra** and write the result in **Rb**. The bit field is specified by a length **l** (**1** to **63**) and a bit position **p** (**0** to **63**) in **Ra**. The bit field length **l** must be non-zero and the sum of **p+l** must not exceed **64** bits. These conditions are checked by the assembler when translating the pseudo-instructions. If the bit position **p** is not specified, it defaults to **0**. The **EXT** and **EXTU** pseudo-instructions extend a signed/unsigned bit field of length **l**.

```
EXTR    Rb = Ra, l, p           // Pseudo: SALR  Rb = Ra, 64-l-p, 64-l
EXTRU   Rb = Ra, l, p           // Pseudo: SHLR  Rb = Ra, 64-l-p, 64-l

EXT     Rb = Ra, l              // Pseudo: SALR  Rb = Ra, 64-l, 64-l
EXTU    Rb = Ra, l              // Pseudo: SHLR  Rb = Ra, 64-l, 64-l
```



EXTR: Extract a Signed Bit Field

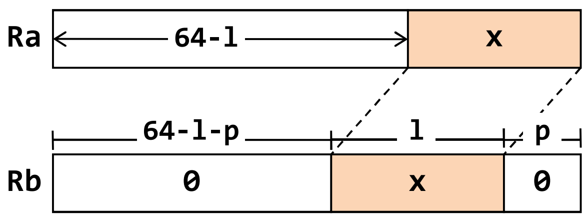


EXTRU: Extract an Unsigned Bit Field

Insert Pseudo-Instruction

INSZ (Insert and Zero) is a pseudo-instruction that inserts a bit field of length **l** from **Ra** into destination register **Rb** at position **p**, and zeroes the lower **p** bits and upper (**64-l-p**) bits of **Rb**. It is equivalent to a **SHLR** instruction. The length field **l** must be non-zero and the sum (**l+p**) must not exceed **64** bits. These conditions are checked by the assembler.

```
INSZ    Rb = Ra, l, p           // Pseudo: SHLR  Rb = Ra, 64-l, 64-l-p
```



INSZ: Insert a bit field and Zero

4.8 Shift and Rotate Instructions (R-Format)

	6	5	5	4	2	5	5
R	**Op = ALU**	**a**	**b**	**f**	**x=1**	**//**	**d**

Same **ALU** opcode (**c** field is ignored)

Four functions: **f = SHL, SHR, SAR, ROR** (with **x=1**)

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rd: Destination register **d**

The **SHL**, **SHR**, **SAR**, and **ROR** (shift and rotate) instructions use **Rb** as a variable shift amount. The lower 6 bits of **Rb** are used as the shift/rotate amount (values 0 to 63).

Assembly Language Syntax:

```
SHL     Rd = Ra, Rb                   // Shift Left
SHR     Rd = Ra, Rb                   // Shift Right
SAR     Rd = Ra, Rb                   // Shift Arithmetic Right
ROR     Rd = Ra, Rb                   // Rotate Right
```

Assembly-Language Note:

To enhance readability, the same mnemonics (**SHL**, **SHR**, etc.) are used in the R-format and I-format. The assembler recognizes the opcode (**ALU** or **SHIFT**) based on the instruction syntax.

SHL, **SHR**, and **SAR** are basic instructions in the R-format. However, they are pseudo-instructions in the I-format (Section 4.7).

4.9 Integer Multiply and Divide Instructions (I-Format)

	6	5	5	4	12
I	**Op = SHIFT**	**a**	**b**	**f**	**Imm12**

Opcode: **`Op = SHIFT`**

Five additional functions: **`f = MUL, DIV, MOD, DIVU, MODU`**

`Ra:` Value of source register **`a`**
`Rb:` Destination register **`b`**
`Imm12:` 12-bit signed immediate

Assembly Language Syntax:

```
MUL     Rb = Ra, Imm12          // Rb = Ra ×s sign_extend(Imm12)

DIV     Rb = Ra, Imm12          // Rb = Ra /s sign_extend(Imm12)
MOD     Rb = Ra, Imm12          // Rb = Ra %s sign_extend(Imm12)
DIVU    Rb = Ra, Imm12          // Rb = Ra /u sign_extend(Imm12)
MODU    Rb = Ra, Imm12          // Rb = Ra %u sign_extend(Imm12)
```

`MUL:` Multiply two 64-bit **signed** integers and write the lower 64-bit of the product to **`Rb`**.

`DIV:` Divide two 64-bit **signed** integers and write the 64-bit **signed quotient** to **`Rb`**.
`MOD:` Divide two 64-bit **signed** integers and write the 64-bit **signed remainder** to **`Rb`**.
`DIVU:` Divide two 64-bit **unsigned** integers and write the 64-bit **unsigned quotient** to **`Rb`**.
`MODU:` Divide two 64-bit **unsigned** integers and write the 64-bit **unsigned remainder** to **`Rb`**.

Programming Notes:

The 12-bit immediate **`Imm12`** is sign-extended, regardless of the operation. Unsigned operations interpret the extended immediate as unsigned.

`MUL` can be used for **signed** and **unsigned** integer multiplication. The lower 64-bit of the product is written to **`Rd`**, and the upper 64-bit of the product is discarded. The lower 64-bit of the product is identical for both signed and unsigned integers. However, the upper 64-bit can be different.

`MUL` can also be used for **32-bit signed/unsigned** integer multiplication. A 32-bit integer is sign or zero-extended when loaded into a 64-bit register. The **`MUL`** instruction computes the correct 64-bit product in both cases.

`DIV` and **`MOD`** can be used independently to compute the **signed** quotient and remainder.
`DIVU` and **`MODU`** can be used independently to compute the **unsigned** quotient and remainder.

4.10 Integer Multiply and Divide Instructions (R-Format)

	6	5	5	4	2	5	5
R	Op = ALU	a	b	f	x=1	//	d

Same **ALU** opcode (**c** field is ignored)

Five additional functions: **f = MUL, DIV, MOD, DIVU, MODU** (with **x=1**)

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rd: Destination register **d**

Assembly Language Syntax:

```
MUL     Rd = Ra, Rb              // Rd = Ra ×s Rb (signed mul)

DIV     Rd = Ra, Rb              // Rd = Ra /s Rb (signed div)
MOD     Rd = Ra, Rb              // Rd = Ra %s Rb (signed mod)
DIVU    Rd = Ra, Rb              // Rd = Ra /u Rb (unsigned div)
MODU    Rd = Ra, Rb              // Rd = Ra %u Rb (unsigned mod)
```

MUL: Multiply two 64-bit **signed** integers and write the lower 64-bit of the product to **Rd**.

DIV: Divide two 64-bit **signed** integers and write the 64-bit **signed quotient** to **Rd**.
MOD: Divide two 64-bit **signed** integers and write the 64-bit **signed remainder** to **Rd**.
DIVU: Divide two 64-bit **unsigned** integers and write the 64-bit **unsigned quotient** to **Rd**.
MODU: Divide two 64-bit **unsigned** integers and write the 64-bit **unsigned remainder** to **Rd**.

Programming Notes:

MUL can be used for **signed** and **unsigned** integer multiplication. The lower 64-bit of the product is written to **Rd**, and the upper 64-bit of the product is discarded. The lower 64-bit of the product is identical for both signed and unsigned integers. However, the upper 64-bit can be different.

MUL can also be used for **32-bit signed/unsigned** integer multiplication. A 32-bit integer is sign or zero-extended when loaded into a 64-bit register. The **MUL** instruction computes the correct 64-bit product in both cases.

DIV and **MOD** can be used independently to compute the **signed** quotient and remainder.
DIVU and **MODU** can be used independently to compute the **unsigned** quotient and remainder.

4.11 Add-Shifted Instruction (R-Format)

	6	5	5	4	2	5	5
R	Op = ALU	a	b	n	x=2	//	d
R	Op = ALU	a	b	n	x=3	//	d

Same **ALU** opcode, but with extensions **x=2** and **x=3** (**c** field is ignored)

Two instructions: **ADDS (x=2)** and **NADDS (x=3)**

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rd: Destination register **d**

The function field specifies the shift amount: **n = 0 to 15**.
The shift amount **n** is restricted to **4** bits, to enable fast hardware implementation.

Assembly Language Syntax:

```
ADDS     Rd = Ra, Rb, n               // Rd =  Ra + Rb<<n
NADDS    Rd = Ra, Rb, n               // Rd = -Ra + Rb<<n
```

Programming Note:

The **ADDS** and **NADDS** instructions combine a shift-left operation with addition. It can be used to calculate memory addresses, or multiply the value of a register by a small constant.

Examples:

```
ADDS     R4 = R2, R3, 3               // R4 =  R2 + R3<<3
ADDS     R6 = R5, R5, 4               // R6 =  R5 + R5<<4 = R5 * 17
NADDS    R7 = R5, R5, 4               // R6 =-R5 + R5<<4 = R5 * 15
```

If **R2** is the address of an array **A** of double words (8-byte elements) and **R3** is an index **i** then the first **ADDS** instruction computes the address of **A[i]**.

The second **ADDS** instruction computes **R5** by **17**.
The **NADDS** instruction computes **R5** by **15**.

4.12 Summary of Opcodes and Function Codes for ALU Instructions (I-Format)

The **ALUI**, **ALUI1**, **ALUI2**, **ALUI3** opcodes use identical function codes. Register **a** is source, while register **b** is destination.

	6	5	5	4	12
I	ALUI = 32	a	b	f	Imm12
I	ALUI1 = 33	a	b	f	Imm12
I	ALUI2 = 34	a	b	f	Imm12
I	ALUI3 = 35	a	b	f	Imm12

	6-bit Opcode	f = 4-bit function codes for **ALUI**, **ALUI1**, **ALUI2**, and **ALUI3**			
I	ALUI = 32	ADD = 0	NADD = 1	AND = 2	CAND = 3
I	ALUI1 = 33	OR = 4	COR = 5	XOR = 6	SET = 7
I	ALUI2 = 34	EQ = 8	NE = 9	LT = 10	GE = 11
I	ALUI3 = 35	LTU = 12	GEU = 13	MIN = 14	MAX = 15

The **ALUI** opcode sign-extends **Imm12** to 64 bits. However, **ALUI1**, **ALUI2**, **ALUI3** have a longer immediate. **ALUI1** and **ALUI2** use one **NOP** instruction to extend the immediate, while **ALUI3** uses two **NOP** instructions to extend the immediate to 64 bits.

The **NOP** instruction carries a 26-bit immediate **Imm26** to instructions that require a long immediate. As a stand-alone instruction, a **NOP** has no effect and is discarded by the execution pipeline.

J	NOP = 0	Imm26

The **RET** opcode combines a function return (jump to return address in **R31**) with an **ALUI** function. **RET** uses the same function codes defined in **ALUI**, as shown in the above table.

I	RET = 36	a	b	f	Imm12
I	SHIFT = 37	a	b	f	Imm12

The **SHIFT** opcode defines nine function codes, as shown below (**SHLR** is defined twice).

	6-bit Opcode	f = 4-bit function codes for **SHIFT**			
I	SHIFT = 37	SHLR = 0	SHLR = 1	SALR = 2	ROR = 3
I	SHIFT = 37	MUL = 8			
I	SHIFT = 37	DIV = 12	MOD = 13	DIVU = 14	MODU = 15

If a function code is not defined then it should not be used. Otherwise, the **Invalid Instruction exception** is raised.

4.13 Summary of Opcodes and Function Codes for ALU Instructions (R-Format)

The **ALU** opcode has four extensions (**x = 0 to 3**). Each extension defines multiple functions. Registers **a** and **b** are source, while register **d** is destination. Field **c** is not used and ignored.

	6	5	5	4	2	5	5
R	ALU = 40	a	b	f	x=0	//	d
R	ALU = 40	a	b	f	x=1	//	d
R	ALU = 40	a	b	n	x=2	//	d
R	ALU = 40	a	b	n	x=3	//	d

The **ALU** opcode with extension **x = 0** defines the same function codes as **ALUI**, except that **SET** is replaced by **XNOR**.

The **ALU** opcode with extension **x = 1** defines shift and rotate instructions by a variable amount, as well as integer multiply and divide instructions. Nine functions are defined.

The **ALU** opcode with extensions **x = 2** and **x = 3** defines the **ADDS** and **NADDS** instructions. The 4-bit function field is used as a left-shift amount **n**.

	6-bit Opcode		**f** = 4-bit function code for the **ALU** opcode			
R	ALU = 40	x=0	ADD = 0	NADD = 1	AND = 2	CAND = 3
R	ALU = 40	x=0	OR = 4	COR = 5	XOR = 6	XNOR = 7
R	ALU = 40	x=0	EQ = 8	NE = 9	LT = 10	GE = 11
R	ALU = 40	x=0	LTU = 12	GEU = 13	MIN = 14	MAX = 15
R	ALU = 40	x=1	SHL = 0	SHR = 1	SAR = 2	ROR = 3
R	ALU = 40	x=1	MUL = 8			
R	ALU = 40	x=1	DIV = 12	MOD = 13	DIVU = 14	MODU = 15
R	ALU = 40	x=2	ADDS (n = 0 to 15)			
R	ALU = 40	x=3	NADDS (n = 0 to 15)			

If a function code is not defined then it should not be used. Otherwise, the **Invalid Instruction exception** is raised.

4.14 Summary of ALU Pseudo-Instructions (I-Format and R-Format)

The following pseudo-instructions change the value of the immediate or use a zero immediate. A long immediate **Imm** (up to 64 bits) can be used by inserting one or two **NOP** after the instruction. The **NOP** carries a 26-bit immediate extension. The assembler should optimize the use of **NOP** instructions.

```
SUB      Rb = Ra, Imm               // Pseudo: ADD   Rb = Ra, -Imm
ANDC     Rb = Ra, Imm               // Pseudo: ADD   Rb = Ra, ~Imm
ORC      Rb = Ra, Imm               // Pseudo: OR    Rb = Ra, ~Imm
XNOR     Rb = Ra, Imm               // Pseudo: XOR   Rb = Ra, ~Imm
MOV      Rb = Ra                    // Pseudo: OR    Rb = Ra, 0
NEG      Rb = Ra                    // Pseudo: NADD  Rb = Ra, 0
NOT      Rb = Ra                    // Pseudo: COR   Rb = Ra, 0

GT       Rb = Ra, Imm               // Pseudo: GE    Rb = Ra, (Imm + 1)
LE       Rb = Ra, Imm               // Pseudo: LT    Rb = Ra, (Imm + 1)
GTU      Rb = Ra, Imm               // Pseudo: GEU   Rb = Ra, (Imm + 1)
LEU      Rb = Ra, Imm               // Pseudo: LTU   Rb = Ra, (Imm + 1)

SHL      Rb = Ra, l                 // Pseudo: SHLR  Rb = Ra, l, 0
SHR      Rb = Ra, r                 // Pseudo: SHLR  Rb = Ra, 0, r
SAR      Rb = Ra, r                 // Pseudo: SALR  Rb = Ra, 0, r
ROL      Rb = Ra, r                 // Pseudo: ROL   Rb = Ra, 64-r

EXTR     Rb = Ra, l, p              // Pseudo: SALR  Rb = Ra, 64-l-p, 64-l
EXTRU    Rb = Ra, l, p              // Pseudo: SHLR  Rb = Ra, 64-l-p, 64-l
EXT      Rb = Ra, l                 // Pseudo: SALR  Rb = Ra, 64-l, 64-l
EXTU     Rb = Ra, l                 // Pseudo: SHLR  Rb = Ra, 64-l, 64-l
INSZ     Rb = Ra, l, p              // Pseudo: SHLR  Rb = Ra, 64-l, 64-l-p
```

The following pseudo-instructions reverse the order of **Ra** and **Rb** in the assembly-language syntax:

```
SUB      Rd = Rb, Ra                // Pseudo: NADD  Rd = Ra, Rb
ANDC     Rd = Rb, Ra                // Pseudo: CAND  Rd = Ra, Rb
ORC      Rd = Rb, Ra                // Pseudo: COR   Rd = Ra, Rb

GT       Rd = Rb, Ra                // Pseudo: LT    Rd = Ra, Rb
LE       Rd = Rb, Ra                // Pseudo: GE    Rd = Ra, Rb
GTU      Rd = Rb, Ra                // Pseudo: LTU   Rd = Ra, Rb
LEU      Rd = Rb, Ra                // Pseudo: GEU   Rd = Ra, Rb
```

5. Three Source Registers

The rationale of having three source registers (**Ra**, **Rb**, and **Rc**) in an ALU instruction, rather than limiting it to only two, is that two dependent operations can be performed in a single instruction. This will result in a more compact and efficient code than forcing all ALU instructions to have only two source registers.

There are many choices for combining two ALU operations in one instruction with three source registers. However, only a short list of instructions is presented here for the most common operations that can be implemented efficiently when combined.

5.1 Three-Operand ALU Instructions (R-Format)

	6	5	5	4	2	5	5
R	Op = ALU3	a	b	f	x=0	c	d

Opcode: **Op = ALU3**

f = ADD, AND, OR, XOR, NADD, CAND, COR, XNOR (with **x=0**)

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rc: Value of third source register **c**
Rd: Destination register **d**

Assembly Language Syntax:

```
ADD      Rd = Ra, Rb, Rc           // Rd =  Ra + Rb + Rc
AND      Rd = Ra, Rb, Rc           // Rd =  Ra & Rb & Rc
OR       Rd = Ra, Rb, Rc           // Rd =  Ra | Rb | Rc
XOR      Rd = Ra, Rb, Rc           // Rd =  Ra ^ Rb ^ Rc
```

Negate (**N**) and Complement (**C**) operate on the **Ra** value only:

```
NADD     Rd = Ra, Rb, Rc           // Rd = -Ra + Rb + Rc
CAND     Rd = Ra, Rb, Rc           // Rd = ~Ra & Rb & Rc
COR      Rd = Ra, Rb, Rc           // Rd = ~Ra | Rb | Rc
XNOR     Rd = Ra, Rb, Rc           // Rd = ~Ra ^ Rb ^ Rc
```

The **ADD** and **NADD** instructions do not cause any exception in the case of overflow.

To enhance readability, the same mnemonics are used in all instruction formats. The assembler recognizes the opcode (**ALU3**, **ALU**, or **ALUI**) based on the instruction syntax.

5.2 Integer Compare Instructions with Logical AND/OR (R-Format)

	6	5	5	4	2	5	5
R	Op = ALU3	a	b	f	x=1	c	d

Compare instructions can be combined with logical **AND/OR** operations for a more compact and faster evaluation of Boolean expressions.

Same **ALU3** opcode, with extension **x=1**

f = ANDEQ, ANDNE, ANDLT, ANDGE, ANDLTU, ANDGEU
f = OREQ, ORNE, ORLT, ORGE, ORLTU, ORGEU
f = MIN, MAX, MINU, MAXU

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rc: Value of third source register **c**
Rd: Destination register **d**

Integer Compare Instructions with logical AND:

```
ANDEQ    Rd = Ra, Rb, Rc           // Rd = Ra && (Rb == Rc)
ANDNE    Rd = Ra, Rb, Rc           // Rd = Ra && (Rb != Rc)
ANDLT    Rd = Ra, Rb, Rc           // Rd = Ra && (Rb <s Rc)
ANDGE    Rd = Ra, Rb, Rc           // Rd = Ra && (Rb ≥s Rc)
ANDLTU   Rd = Ra, Rb, Rc           // Rd = Ra && (Rb <u Rc)
ANDGEU   Rd = Ra, Rb, Rc           // Rd = Ra && (Rb ≥u Rc)
```

Integer Compare Instructions with logical OR:

```
OREQ     Rd = Ra, Rb, Rc           // Rd = Ra || (Rb == Rc)
ORNE     Rd = Ra, Rb, Rc           // Rd = Ra || (Rb != Rc)
ORLT     Rd = Ra, Rb, Rc           // Rd = Ra || (Rb <s Rc)
ORGE     Rd = Ra, Rb, Rc           // Rd = Ra || (Rb ≥s Rc)
ORLTU    Rd = Ra, Rb, Rc           // Rd = Ra || (Rb <u Rc)
ORGEU    Rd = Ra, Rb, Rc           // Rd = Ra || (Rb ≥u Rc)
```

Integer Minimum and Maximum (Signed and Unsigned):

```
MIN      Rd = Ra, Rb, Rc           // Rd = MIN (Ra, Rb, Rc)
MAX      Rd = Ra, Rb, Rc           // Rd = MAX (Ra, Rb, Rc)
MINU     Rd = Ra, Rb, Rc           // Rd = MINU(Ra, Rb, Rc)
MAXU     Rd = Ra, Rb, Rc           // Rd = MAXU(Ra, Rb, Rc)
```

Integer Compare Pseudo-Instructions (R-Format)

These pseudo-instructions switch the place of **Rb** and **Rc** in the assembly-language syntax:

```
ANDGT   Rd = Ra, Rc, Rb          // Pseudo: ANDLT    Rd = Ra, Rb, Rc
ANDLE   Rd = Ra, Rc, Rb          // Pseudo: ANDGE    Rd = Ra, Rb, Rc
ANDGTU  Rd = Ra, Rc, Rb          // Pseudo: ANDLTU   Rd = Ra, Rb, Rc
ANDLEU  Rd = Ra, Rc, Rb          // Pseudo: ANDGEU   Rd = Ra, Rb, Rc

ORGT    Rd = Ra, Rc, Rb          // Pseudo: ORLT     Rd = Ra, Rb, Rc
ORLE    Rd = Ra, Rc, Rb          // Pseudo: ORGE     Rd = Ra, Rb, Rc
ORGTU   Rd = Ra, Rc, Rb          // Pseudo: ORLTU    Rd = Ra, Rb, Rc
ORLEU   Rd = Ra, Rc, Rb          // Pseudo: ORGEU    Rd = Ra, Rb, Rc
```

Translating Boolean Expressions with Logical AND/OR

Compare instructions with logical AND/OR options can be used to translate Boolean expressions and reduce the number of branch instructions. The following are examples:

Translating a Boolean expression with logical AND operators:

```
if ((R1 < R2) && (R3 >= R4) && (R5 != R6)) {IF-Block}

LT      R7 = R1, R2              // R7 = (R1 < R2)
ANDGE   R7 = R7, R3, R4          // R7 = R7 && (R3 >= R4)
ANDNE   R7 = R7, R5, R6          // R7 = R7 && (R5 != R6)
BEQZ    R7, @next                // Branch if False
. . .                            // IF-Block
@next
```

Translating a Boolean expression with logical OR operators:

```
if ((R1 < R2) || (R3 >= R4) || (R5 != R6)) {IF-Block}

LT      R7 = R1, R2          // R7 = (R1 < R2)
ORGE    R7 = R7, R3, R4      // R7 = R7 || (R3 >= R4)
ORNE    R7 = R7, R5, R6      // R7 = R7 || (R5 != R6)
BEQZ    R7, @next            // Branch if False
. . .                        // IF-Block
@next
```

5.3 Select Instructions (R-Format)

	6	5	5	4	2	5	5
R	Op = ALU3	a	b	f	x=2	c	d

Same **ALU3** opcode, with extension **x=2**

f = SEL (Select), **SELN** (Select if Negative), **SELP** (Select if Positive)

Ra: Value of source register **a**, that controls the selection
Rb: Value of source register **b**, which is selected if the condition is true
Rc: Value of source register **c**, which is selected if the condition is false
Rd: Destination register **d**

Assembly-Language Syntax:

```
SEL     Rd = Ra, Rb, Rc          // Rd = (Ra != 0)? Rb : Rc
SELN    Rd = Ra, Rb, Rc          // Rd = (Ra <  0)? Rb : Rc
SELP    Rd = Ra, Rb, Rc          // Rd = (Ra >  0)? Rb : Rc

SELZ    Rd = Ra, Rc, Rb          // Pseudo: SEL Rd = Ra, Rb, Rc
```

Translating Conditional Expressions

Select instructions are useful for translating conditional expressions without the use of branch instructions. They always write destination register **Rd**, which simplifies their implementation. Here is an example on translating a conditional expression:

```
R1 = (R2 > 0)? R4+R5: R4-R5;

ADD     R6 = R4, R5              // R6 = R4 + R5
SUB     R7 = R4, R5              // R7 = R4 - R5
SELP    R1 = R2, R6, R7          // R1 = (R2 > 0)? R6: R7
```

5.4 Integer Multiply-Add Instructions (R-Format)

	6	5	5	4	2	5	5
R	**Op = ALU3**	**a**	**b**	**f**	**x=2**	**c**	**d**

Same **ALU3** opcode, with extension **x=2**

Two functions: **f = MADD, NMADD**

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rc: Value of third source register **c**
Rd: Destination register **d**

Assembly Language Syntax:

```
MADD     Rd = Ra, Rb, Rc          // Rd =  Ra × Rb + Rc
NMADD    Rd = Ra, Rb, Rc          // Rd = -Ra × Rb + Rc
```

MADD: Multiply two 64-bit signed integers **Ra** and **Rb**, then add the product to 64-bit signed **Rc**. Write the lower 64-bit of the result to **Rd**. The upper 64-bit of the result is discarded.

NMADD: Multiply two 64-bit signed integers **Ra** and **Rb**, then subtract the product from 64-bit **Rc**. Write the lower 64-bit of the result to **Rd**. The upper 64-bit of the result is discarded.

Programming Notes:

MADD and **NMADD** can be used for **signed** and **unsigned** integer computations, because the lower 64-bit of the product is the same whether the integers are signed or unsigned.

MADD and **NMADD** can be used also for **32-bit signed** and **unsigned** integer computations. A 32-bit integer is first sign-extended or zero-extended when loaded into a 64-bit register. The **MADD** and **NMADD** instructions compute the correct 64-bit result in both cases.

5.5 Summary of Function Codes for Three-Operand ALU Instructions

Only one **ALU3** opcode is defined for all three-operand integer instructions. It has three source registers **a**, **b**, and **c**, and destination register **d**.

	6	5	5	4	2	5	5
R	**ALU3 = 41**	**a**	**b**	**f**	**x**	**c**	**d**

The **ALU3** function codes are listed below for **x = 0** to **2**.

	6-bit Opcode		**f** = 4-bit function codes for the **ALU3**			
R	**ALU3 = 41**	**x=0**	**ADD = 0**	**NADD = 1**	**AND = 2**	**CAND = 3**
R	**ALU3 = 41**	**x=0**	**OR = 4**	**COR = 5**	**XOR = 6**	**XNOR = 7**
R	**ALU3 = 41**	**x=1**	**ANDEQ = 0**	**ANDNE = 1**	**ANDLT = 2**	**ANDGE = 3**
R	**ALU3 = 41**	**x=1**	**ANDLTU = 4**	**ANDGEU = 5**	**MINU = 6**	**MAXU = 7**
R	**ALU3 = 41**	**x=1**	**OREQ = 8**	**ORNE = 9**	**ORLT = 10**	**ORGE = 11**
R	**ALU3 = 41**	**x=1**	**ORLTU = 12**	**ORGEU = 13**	**MIN = 14**	**MAX = 15**
R	**ALU3 = 41**	**x=2**	**SEL = 0**	**SELN = 1**	**SELP = 2**	
R	**ALU3 = 41**	**x=2**	**MADD = 4**	**NMADD = 5**		

If a function code is not defined then it should not be used. Otherwise, the **Invalid Instruction exception** is raised.

6. Floating-Point Instructions

Floating-point instructions use the same general-purpose register **R0** to **R31**. There is no need for separate load and store instructions. The same load/store instructions are used to load/store integer and floating point data. The same branch instructions are used to branch on floating-point comparison results. In addition, there is no need to copy data between two register files because only one exists.

Floating-point instructions operate on 32-bit single-precision or 64-bit double-precision data. For single-precision, the lower 32-bit of a register is read and the upper 32-bit is ignored. A 32-bit computed result is written to the lower 32-bit of a register and the upper 32-bit is zeroed. For double precision, the full 64-bit register is read and written.

All floating-point instructions use the R-Format. The function field **f** is extended to 5 bits. The 1-bit **p** field specifies the precision of the floating-point operation: **p=0** means single-precision, while **p=1** means double-precision.

	6	5	5	5	1	5	5
R	**Op**	**a**	**b**	**f**	**p**	**c**	**d**

Three floating-point opcodes are defined. The **FPU1** opcode defines floating-point instructions that have a single operand. It reads source register **Ra** and writes destination register **Rd**. The **FPU1** opcode ignores the **b** and **c** fields.

The **FPU2** opcode defines floating-point instructions that have two source operands: **Ra** and **Rb**. The result is written to **Rd**. The **FPU2** opcode ignores the **c** field.

The **FPU3** opcode defines floating-point instructions that have three source operands: **Ra**, **Rb**, and **Rc**. The result is written to **Rd**. An **FPU3** instruction combines two floating-point operations.

6.1 Floating-Point Instructions with one Source Operand (R-Format)

	6	5	5	5	1	5	5
R	**Op = FPU1**	**a**	**//**	**f**	**p**	**//**	**d**

Opcode: **Op = FPU1**, with precision **p=0** (single-precision) and **p=1** (double-precision)

Single-Precision: **f = ABS.S, NEG.S, SQRT.S, CVTS.D, CVTS.I, CVTI.S, RINT.S**
Double-Precision: **f = ABS.D, NEG.D, SQRT.D, CVTD.S, CVTD.I, CVTI.D, RINT.D**

Ra: Value of source register **a**
Rd: Destination register **d**

The **b** and **c** fields are not used and ignored.

Assembly Language Syntax:

```
ABS.S     Rd = Ra          // Absolute value, Single precision
ABS.D     Rd = Ra          // Absolute value, Double precision
NEG.S     Rd = Ra          // Negate, Single precision
NEG.D     Rd = Ra          // Negate, Double precision
SQRT.S    Rd = Ra          // Square Root, Single precision
SQRT.D    Rd = Ra          // Square Root, Double precision
CVTS.D    Rd = Ra          // Convert to Single precision, Double
CVTD.S    Rd = Ra          // Convert to Double precision, Single
CVTS.I    Rd = Ra          // Convert to Single precision, Integer
CVTD.I    Rd = Ra          // Convert to Double precision, Integer
CVTI.S    Rd = Ra          // Convert to Integer, Single precision
CVTI.D    Rd = Ra          // Convert to Integer, Double precision
RINT.S    Rd = Ra          // Round to Integral, Single precision
RINT.D    Rd = Ra          // Round to Integral, Double precision
```

The **RINT** instruction rounds a single or double precision floating-point number into an integer according to the rounding mode, but keeps the integer value in the floating-point representation.

6.2 Floating-Point Compare Instructions (R-Format)

	6	5	5	5	1	5	5
R	Op = FPU2	a	b	f	p	//	d

Floating-point compare instructions compare two source register values **Ra** and **Rb** and write a Boolean result (**0** or **1**) in destination register **Rd**.

Opcode: **Op = FPU2**, with precision **p=0** (single-precision) and **p=1** (double-precision)

Single-Precision: **f = EQ.S, NE.S, LT.S, GE.S, INF.S, NAN.S**
Double-Precision: **f = EQ.D, NE.D, LT.D, GE.D, INF.D, NAN.D**

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rd: Destination register **d**

Assembly Language Syntax:

```
EQ.S     Rd = Ra, Rb          // Rd = (Ra == Rb), Single precision
EQ.D     Rd = Ra, Rb          // Rd = (Ra == Rb), Double precision
NE.S     Rd = Ra, Rb          // Rd = (Ra != Rb), Single precision
NE.D     Rd = Ra, Rb          // Rd = (Ra != Rb), Double precision
LT.S     Rd = Ra, Rb          // Rd = (Ra <  Rb), Single precision
LT.D     Rd = Ra, Rb          // Rd = (Ra <  Rb), Double precision
GE.S     Rd = Ra, Rb          // Rd = (Ra >= Rb), Single precision
GE.D     Rd = Ra, Rb          // Rd = (Ra >= Rb), Double precision
INF.S    Rd = Ra, Rb          // Rd = INF(Ra) || INF(Rb), Single
INF.D    Rd = Ra, Rb          // Rd = INF(Ra) || INF(Rb), Double
NAN.S    Rd = Ra, Rb          // Rd = NAN(Ra) || NAN(Rb), Single
NAN.D    Rd = Ra, Rb          // Rd = NAN(Ra) || NAN(Rb), Double
```

The **INF** instruction returns true if **Ra** or **Rb** is Infinity.
The **NAN** instruction returns true if **Ra** or **Rb** is Not-a-Number.

Floating-Point Compare Pseudo-Instructions

Compare pseudo-instructions reverse the order of **Ra** and **Rb** in the assembly-language syntax:

```
GT.S     Rd = Rb, Ra          // Pseudo: LT.S Rd = Ra, Rb
GT.D     Rd = Rb, Ra          // Pseudo: LT.D Rd = Ra, Rb
LE.S     Rd = Rb, Ra          // Pseudo: GE.S Rd = Ra, Rb
LE.D     Rd = Rb, Ra          // Pseudo: GE.S Rd = Ra, Rb
```

6.3 Floating-Point Arithmetic Instructions (R-Format)

	6	5	5	5	1	5	5
R	Op = FPU2	a	b	f	p	//	d

Same **FPU2** opcode, with precision **p=0** (single-precision) and **p=1** (double-precision)

Single-Precision: **f = ADD.S, NADD.S, MUL.S, DIV.S, MIN.S, MAX.S**
Double-Precision: **f = ADD.D, NADD.D, MUL.D, DIV.D, MIN.D, MAX.D**

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rd: Destination register **d**

Assembly Language Syntax:

```
ADD.S     Rd = Ra, Rb          // Rd =  Ra + Rb, Single precision
ADD.D     Rd = Ra, Rb          // Rd =  Ra + Rb, Double precision
NADD.S    Rd = Ra, Rb          // Rd = -Ra + Rb, Single precision
NADD.D    Rd = Ra, Rb          // Rd = -Ra + Rb, Double precision
MUL.S     Rd = Ra, Rb          // Rd =  Ra × Rb, Single precision
MUL.D     Rd = Ra, Rb          // Rd =  Ra × Rb, Double precision
DIV.S     Rd = Ra, Rb          // Rd =  Ra / Rb, Single precision
DIV.D     Rd = Ra, Rb          // Rd =  Ra / Rb, Double precision
MIN.S     Rd = Ra, Rb          // Rd = MIN(Ra,Rb), Single precision
MIN.D     Rd = Ra, Rb          // Rd = MIN(Ra,Rb), Double precision
MAX.S     Rd = Ra, Rb          // Rd = MAX(Ra,Rb), Single precision
MAX.D     Rd = Ra, Rb          // Rd = MAX(Ra,Rb), Double precision
```

Subtract Pseudo-Instructions

Subtract pseudo-instructions reverse the order of **Ra** and **Rb** in the assembly-language syntax:

```
SUB.S     Rd = Rb, Ra          // Pseudo: NADD.S Rd = Ra, Rb
SUB.D     Rd = Rb, Ra          // Pseudo: NADD.D Rd = Ra, Rb
```

6.4 Floating-Point Instructions with Three Source Registers (R-Format)

There are many choices for combining two floating-point operations into one instruction with three source registers. However, only the most common operations that can be implemented efficiently when combined are presented.

	6	5	5	5	1	5	5
R	**Op = FPU3**	**a**	**b**	**f**	**p**	**c**	**d**

Opcode: **Op = FPU3**, with precision **p=0** (single-precision) and **p=1** (double-precision)

Single-Precision: **f = ADD.S, NADD.S, MADD.S, NMADD.S, MIN.S, MAX.S**
Double-Precision: **f = ADD.D, NADD.D, MADD.D, NMADD.D, MIN.D, MAX.D**

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rc: Value of third source register **c**
Rd: Destination register **d**

Assembly Language Syntax:

```
ADD.S     Rd = Ra, Rb, Rc        // Rd =  Ra + Rb + Rc, Single precision
ADD.D     Rd = Ra, Rb, Rc        // Rd =  Ra + Rb + Rc, Double precision
NADD.S    Rd = Ra, Rb, Rc        // Rd = -Ra + Rb + Rc, Single precision
NADD.D    Rd = Ra, Rb, Rc        // Rd = -Ra + Rb + Rc, Double precision
MADD.S    Rd = Ra, Rb, Rc        // Rd =  Ra × Rb + Rc, Single precision
MADD.D    Rd = Ra, Rb, Rc        // Rd =  Ra × Rb + Rc, Double precision
NMADD.S   Rd = Ra, Rb, Rc        // Rd = -Ra × Rb + Rc, Single precision
NMADD.D   Rd = Ra, Rb, Rc        // Rd = -Ra × Rb + Rc, Double precision
MIN.S     Rd = Ra, Rb, Rc        // Rd = MIN(Ra,Rb,Rc), Single precision
MIN.D     Rd = Ra, Rb, Rc        // Rd = MIN(Ra,Rb,Rc), Double precision
MAX.S     Rd = Ra, Rb, Rc        // Rd = MAX(Ra,Rb,Rc), Single precision
MAX.D     Rd = Ra, Rb, Rc        // Rd = MAX(Ra,Rb,Rc), Double precision
```

6.5 Floating-Point Compare Instructions with Logical AND/OR (R-Format)

Floating-point compare instructions can be combined with logical **AND/OR** operations for a more compact and faster evaluation of Boolean expressions.

	6	5	5	5	1	5	5
R	Op = FPU3	a	b	f	p	c	d

Same **FPU3** opcode, with precision **p=0** (single-precision) and **p=1** (double-precision)

Ra: Value of first source register **a**
Rb: Value of second source register **b**
Rc: Value of third source register **c**
Rd: Destination register **d**

Assembly Language Syntax:

```
ANDEQ.S    Rd = Ra, Rb, Rc       // Rd = Ra && (Rb == Rc), Single-precision
ANDEQ.D    Rd = Ra, Rb, Rc       // Rd = Ra && (Rb == Rc), Double-precision
ANDNE.S    Rd = Ra, Rb, Rc       // Rd = Ra && (Rb != Rc), Single-precision
ANDNE.D    Rd = Ra, Rb, Rc       // Rd = Ra && (Rb != Rc), Double-precision
ANDLT.S    Rd = Ra, Rb, Rc       // Rd = Ra && (Rb <  Rc), Single-precision
ANDLT.D    Rd = Ra, Rb, Rc       // Rd = Ra && (Rb <  Rc), Double-precision
ANDGE.S    Rd = Ra, Rb, Rc       // Rd = Ra && (Rb >= Rc), Single-precision
ANDGE.D    Rd = Ra, Rb, Rc       // Rd = Ra && (Rb >= Rc), Double-precision

OREQ.S     Rd = Ra, Rb, Rc       // Rd = Ra || (Rb == Rc), Single-precision
OREQ.D     Rd = Ra, Rb, Rc       // Rd = Ra || (Rb == Rc), Double-precision
ORNE.S     Rd = Ra, Rb, Rc       // Rd = Ra || (Rb != Rc), Single-precision
ORNE.D     Rd = Ra, Rb, Rc       // Rd = Ra || (Rb != Rc), Double-precision
ORLT.S     Rd = Ra, Rb, Rc       // Rd = Ra || (Rb <  Rc), Single-precision
ORLT.D     Rd = Ra, Rb, Rc       // Rd = Ra || (Rb <  Rc), Double-precision
ORGE.S     Rd = Ra, Rb, Rc       // Rd = Ra || (Rb >= Rc), Single-precision
ORGE.D     Rd = Ra, Rb, Rc       // Rd = Ra || (Rb >= Rc), Double-precision
```

Pseudo-instructions reverse the order of **Rb** and **Rc** in the assembly-language syntax:

```
ANDGT.S    Rd = Ra, Rc, Rb       // Pseudo: ANDLT.S      Rd = Ra, Rb, Rc
ANDGT.D    Rd = Ra, Rc, Rb       // Pseudo: ANDLT.D      Rd = Ra, Rb, Rc
ANDLE.S    Rd = Ra, Rc, Rb       // Pseudo: ANDGE.S      Rd = Ra, Rb, Rc
ANDLE.D    Rd = Ra, Rc, Rb       // Pseudo: ANDGE.D      Rd = Ra, Rb, Rc

ORGT.S     Rd = Ra, Rc, Rb       // Pseudo: ORLT.S       Rd = Ra, Rb, Rc
ORGT.D     Rd = Ra, Rc, Rb       // Pseudo: ORLT.D       Rd = Ra, Rb, Rc
ORLE.S     Rd = Ra, Rc, Rb       // Pseudo: ORGE.S       Rd = Ra, Rb, Rc
ORLE.D     Rd = Ra, Rc, Rb       // Pseudo: ORGE.D       Rd = Ra, Rb, Rc
```

6.6 Summary of Opcodes and Functions for Floating-Point Instructions

Three floating-point opcodes are defined: **FPU1**, **FPU2**, and **FPU3**, with one, two, and three source registers. Two floating-point precisions are defined: **p = 0** indicates single-precision (**.S** extension), while **p = 1** indicates double-precision (**.D** extension).

	6	5	5	5	1	5	5
R	FPU1 = 42	a	//	f	p	//	d
R	FPU2 = 43	a	b	f	p	//	d
R	FPU3 = 44	a	b	f	p	c	d

The **FPU1**, **FPU2**, and **FPU3** functions are listed below. The same function name is listed twice for single precision (**p=0**) and double-precision (**p=1**).

	6-bit Opcode		**f** = 5-bit function codes for the **FPU1**, **FPU2**, and **FPU3** opcodes			
R	FPU1 = 42	p=0	ABS = 0	NEG = 1	SQRT = 2	
R	FPU1 = 42	p=0	CVTSD = 4	CVTSI = 5	CVTIS = 6	RINT = 7
R	FPU1 = 42	p=1	ABS = 0	NEG = 1	SQRT = 2	
R	FPU1 = 42	p=1	CVTDS = 4	CVTDI = 5	CVTID = 6	RINT = 7
R	FPU2 = 43	p=0	EQ = 0	NE = 1	LT = 2	GE = 3
R	FPU2 = 43	p=0	INF = 4	NAN = 5		
R	FPU2 = 43	p=0	ADD = 8	NADD = 9	MUL = 10	DIV = 11
R	FPU3 = 43	p=0	MIN = 12	MAX = 13		
R	FPU2 = 43	p=1	EQ = 0	NE = 1	LT = 2	GE = 3
R	FPU2 = 43	p=1	INF = 4	NAN = 5		
R	FPU2 = 43	p=1	ADD = 8	NADD = 9	MUL = 10	DIV = 11
R	FPU3 = 43	p=1	MIN = 12	MAX = 13		
R	FPU3 = 44	p=0	ANDEQ = 0	ANDNE = 1	ANDLT = 2	ANDGE = 3
R	FPU3 = 44	p=0	OREQ = 4	ORNE = 5	ORLT = 6	ORGE = 7
R	FPU3 = 44	p=0	ADD = 8	NADD = 9	MADD = 10	NMADD = 11
R	FPU3 = 44	p=0	MIN = 12	MAX = 13		
R	FPU3 = 44	p=1	ANDEQ = 0	ANDNE = 1	ANDLT = 2	ANDGE = 3
R	FPU3 = 44	p=1	OREQ = 4	ORNE = 5	ORLT = 6	ORGE = 7
R	FPU3 = 44	p=1	ADD = 8	NADD = 9	MADD = 10	NMADD = 11
R	FPU3 = 44	p=1	MIN = 12	MAX = 13		

If a function code is not defined then it should not be used. Otherwise, the **Invalid Instruction exception** is raised.

7. Privileged Architecture

The M-Architecture defines two levels of execution: user level (or **EL0**) and supervisor level (or **EL1**). User level is for the normal execution of programs. A program can access registers and memory that are defined only at **EL0**. Supervisor level is for handling software exceptions and hardware interrupts. An operating system runs at **EL1** and provides services to application programs that run at **EL0**. An operating system has access to all hardware registers and memory defined at both **EL0** and **EL1**. A program running at **EL0** can obtain services from the operating system by executing a system call (**SCALL**) instruction that changes the execution level from **EL0** to **EL1**. The **SCALL** instruction changes the program counter (**PC**) register to start executing the system call handler, which runs at **EL1**. The last instruction in a system call handler is an exception return (**ERET**) instruction that transfers control back to the application program and changes the execution level from **EL1** back to **EL0**.

7.1 System Call Instruction

The **SCALL** instruction is the interface between an application program and the underlying operating system. It uses a 12-bit immediate to represent the service code:

```
SCALL    code                              // System Call with a service code
```

The Linux OS has close to 400 systems calls. Windows has close to 700. However, a simulator might define only few system services, mainly for input and output. System calls receive their parameters and return their result in registers, similar to functions. Here are examples of system calls, which are similar to the ones defined in the MARS and SPIM tools.

Instruction	Service	Arguments	Result
SCALL 0	Print Character	**R0 =** Character to print	*Console output*
SCALL 1	Print Integer	**R0 =** Integer to print	*Console output*
SCALL 2	Print Float	**R0 =** Float to print	*Console output*
SCALL 3	Print Double	**R0 =** Double to print	*Console output*
SCALL 4	Print String	**R0 =** String address	*String must be null-terminated*
SCALL 5	Read Character		**R0 =** Character read
SCALL 6	Read Integer		**R0 =** Integer read
SCALL 7	Read Float		**R0 =** Float read
SCALL 8	Read Double		**R0 =** Double read
SCALL 9	Read String	**R0 =** Buffer address **R1 =** Buffer size	*At most* (*size* – 1) *chars are read* *Null char is inserted*

Read Character (**SCALL 5**) reads one character directly from the keyboard without displaying it on the console output. It *does not wait* for the user to press the **enter key**. On the other hand, system calls 6 to 9 wait until the user presses the **enter key**.

Read String (**SCALL 9**) reads at most (N – 1) characters into a buffer of N bytes. The newline character *is not stored* in the buffer. A null character is always appended at end of string.

Instruction	Service	Arguments	Result
SCALL 10	Open File	**R0 =** Pathname address **R1 =** 0 (read), 1 (write)	**R0 =** File descriptor **R0** *is negative if error*
SCALL 11	Read from File	**R0 =** File descriptor **R1 =** Buffer address **R2 =** Bytes to read	**R0 =** Number of bytes read **R0 = 0** *if end-of-file* **R0** *is negative if error*
SCALL 12	Write to File	**R0 =** File descriptor **R1 =** Buffer address **R2 =** Bytes to write	**R0 =** Number of bytes written **R0** *is negative if error*
SCALL 13	Close File	**R0 =** File descriptor	
SCALL 14	Allocate Memory	**R0 =** Number of bytes	**R0 =** Memory address
SCALL 15	Exit Program	**R0 =** Termination value	

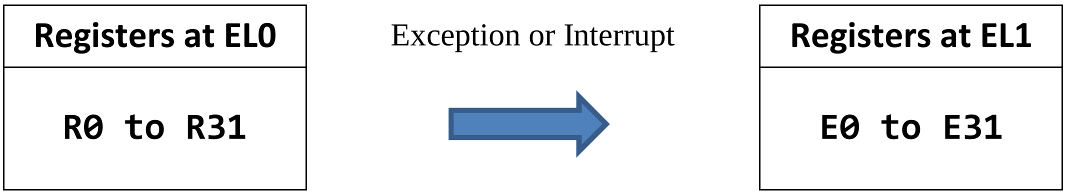
Open File (**SCALL 10**) has two arguments: **R0** is the address of a string in memory that contains the pathname (or filename) and **R1** contains the flags. If **R1** is **0** then the file is open as read-only. If the file is not found on the disk, the system call fails and returns a negative number in **R0**. Otherwise, it returns a valid file descriptor in **R0**. If **R1** is **1** then the file is open as write only. If the file is not found then it is created on the disk. If **R1** is **2** then the open file can be read and written. More options can be added, such as create a new file, empty the content of the open file, or append at end of file.

The above list of system calls is used mainly by a simulator. However, to implement a small operating system kernel on a hardware prototype, a new list of system calls should be defined for process management, memory management, and filesystem management.

Software libraries can be built on top of system calls. They hide the system call interface implemented by the underlying operating system.

7.2 System Registers for Exception Handling

To enable the efficient handling of exceptions and interrupts, the M-architecture defines a separate set of thirty-two 64-bit exception registers (**E0** to **E31**) that can be accessed only when the processor is running at **EL1**. When a synchronous exception or asynchronous hardware interrupt occurs, the processor automatically switches to the set of exception registers without software intervention. Exception registers can be accessed directly by any instruction running at supervisor level (or **EL1**). On the other hand, an instruction running at user level (or **EL0**) is not allowed to access an exception register. Otherwise, it will cause a synchronous exception.



The following exception registers have special use and facilitate exception handling:

EBASE Exception Base (or **E0**): holds the address where exception handling starts. This register must be initialized to a specific memory address by the system software.

EPC Exception Program Counter (or **E16**): holds the address of the instruction that caused the exception and the saved execution level **EL**. This is the case for all synchronous exceptions, system call (**SCALL**), and debugger call (**DCALL**) instructions. For asynchronous interrupts, **EPC** points to the instruction that should resume execution after handling the interrupt. The **EPC** register is used by the exception return (**ERET**) instruction to return from an exception.

The 64-bit **EPC** register is shown below. The lower two bits hold the saved **EL**. Only one bit specifies **EL0** or **EL1**, but a second bit is reserved for future expansion of the architecture to four execution levels. The upper 62 bits specify the address of the instruction that caused the exception. A given implementation might reduce the size of the address field.

	62	**2**
EPC	**ADDR** = Address of Instruction that Caused Exception	**EL**

EADDR Exception Address register (or **E17**): holds the address that caused the exception (if any).

ECODE Exception Code register (or **E18**): holds a function code that allows the exception handler to execute different exception functions. When a system call instruction is executed, such as **SCALL 0x123**, the **ECODE** register becomes **0x123**.

ESP Exception Stack Pointer (or **E30**): holds the address to a dedicated stack in memory used by exception handlers.

ELR Exception Link Register (or **E31**): holds the return address of a system function.

Seven system registers are defined as scratch registers for exception handling. In addition, two registers are used for interrupt status and control.

E1-E7 Scratch registers for exception handling and system software.

INTS Interrupt Status register (or **E19**): holds the status bits of the pending interrupts.

INTM Interrupt Mask register (or **E20**): holds the mask bits that enable or disable interrupts.

Only fifteen system registers are defined for exception and interrupt handling. These are listed below. The remaining ones are reserved for future use. Reading an undefined system register returns zero. Writing an undefined system register has no effect.

Number	Name	Description
E0	**EBASE**	Exception Base address for exception and interrupt handlers
E1-E7		Scratch Registers used by exception and interrupt handlers
E8-E15		Reserved for future use
E16	**EPC**	Exception Program Counter: holds the instruction address and exception level
E17	**EADDR**	Exception Address: holds the bad address of an exception
E18	**ECODE**	Exception Code Register for system call and synchronous exceptions
E19	**INTS**	Interrupt status bits for pending interrupts that are awaiting service
E20	**INTM**	Interrupt Mask bits that enable or disable interrupts
E21-E29		Reserved for future use
E30	**ESP**	Exception Stack Pointer: alternative stack for exception handling
E31	**ELR**	Exception Link Register: saves the return address of a system function

7.3 The Program Counter Register

The 64-bit **PC** register is shown below. It has the same format as the **EPC** register. The lower two bits specify the current execution level **EL**. Only one bit specifies **EL0** or **EL1**, but a second bit is reserved for future expansion of the architecture to four execution levels. The upper 62 bits specify the address of the current instruction. A given implementation might reduce the size of the address field.

	62	2
PC	**ADDR** = Address of Current Instruction	**EL**

The **JR**, **JALR**, **RET**, and all branch instructions never change the current execution level **EL** in the **PC** register. They only change the **PC** address (**PC.ADDR** or **PC[63:2]**), as explained in Section 2.5.

7.4 Exception Handling

An exception is something that interrupts the normal execution of a program. In the M-architecture, there are four major types of exceptions: **SCALL** instruction that transfers control to the operating system, **DCALL** instruction that transfers control to the debugger, a ***synchronous*** exception caused by a running instruction, and an ***asynchronous*** interrupt caused by a hardware device external to program execution. In all cases, the processor transfers control to an exception handler.

This is what happens when an exception occurs:

1. The instruction address and execution level are saved in the **EPC** register: **EPC = PC**
2. The exception code is saved in the **ECODE** register. For **SCALL**, **ECODE = Imm12**
3. If the exception is caused by a bad address then it is saved in the **EADDR** register
4. The execution level changes to **EL1**: **PC.EL = 1** (or **PC[1:0] = 1**)
5. The processor jumps to the starting address of the exception handler: **PC.ADDR = EBASE + n**

The starting address of the exception handler depends on the exception type. There are four exception types and four starting addresses for all exceptions, as shown in the following table. All system calls start at **PC=EBASE**. There are 16 instructions (64 bytes) between **EBASE** and **EBASE+0x40** that are used to save registers in memory and jump to the target system call function according to the exception code (**ECODE**). A table of addresses maps the exception code (**ECODE**) to its function in memory. Similarly, the debugger handler starts at **EBASE+0x40**, a synchronous exception handler starts at **EBASE+0x80**, and an asynchronous interrupt handler starts at **EBASE+0xC0**.

Exception Type	Handler Address
System Call	**PC = EBASE + 0x00**
Debugger Call	**PC = EBASE + 0x40**
Synchronous	**PC = EBASE + 0x80**
Asynchronous	**PC = EBASE + 0xC0**

7.5 Exception Return

The **ERET** instruction returns from an exception handler. It has the following syntax:

```
ERET     Imm12                          // Exception Return
```

This is what happens when the **ERET** instruction is executed:

1. The **PC** execution level is restored from the **EPC** register: **PC.EL = EPC.EL**
2. The processor jumps to the return address: **PC.ADDR = EPC.ADDR + Imm12**

The 12-bit immediate (**Imm12**) is used as an offset. To return from a system call use: **ERET 1**. The exception handler returns to the instruction that follows the **SCALL** instruction. However, to restart the execution of an instruction that caused a synchronous exception, use **ERET 0**.

7.6 Writing an Exception Handler

The following **.sdata** directive defines a system data segment that starts at address **0x80100000**. Two arrays are defined: **@scall_pointers** is an array of pointers (word addresses) to system call functions and **@exception_pointers** is a second array of pointers to exception functions. The pointers are entry points to functions. The two arrays are indexed using the system call instruction code and exception code, respectively.

```
.sdata   0x80100000                // System data segment at 0x80100000
@scall_pointers                    // System call function addresses
.word    0x80000100                // Pointer to SCALL 0 function
.word    0x80000254                // Pointer to SCALL 1 function
.word    0x8000040C                // Pointer to SCALL 2 function
         . . .
@exception_pointers                // Exception function addresses
.word    0x8002F3AC                // Pointer to exception handler 0
.word    0x8002F4B8                // Pointer to exception handler 1
         . . .
```

The following **.stext** directive defines the start address of the system call handler at **0x80000000**. This is the value of the **EBASE** register. This handler allocates a stack frame of 8 bytes. It uses **esp** that points to a dedicated stack for system calls and exception handling. It saves the **epc** register on the stack, uses **ecode** to address an array of function pointers, loads the system function address into **e2**, calls the system function (**jalr** instruction), then restores the **epc** value from the stack, and returns back to normal program execution. The **eret 1** instruction transfers control back to the address that follows the **SCALL** instruction that initiated the exception.

```
.stext   0x80000000                // System text segment at 0x80000000
add      esp = esp, -8             // Allocate a stack frame of 8 bytes
sd       [esp] = epc               // Save the Exception Program Counter
set      e1 = @scall_pointers      // e1 = array address = 0x80000100
lw       e2 = [e1, ecode, 2]       // e2 = system call function address
jalr     e31, e2                   // call system function
ld       epc = [sp]                // Restore the Exception Program Counter
add      esp = esp, 8              // Restore the Exception Stack Pointer
eret     1                         // Return to instruction after SCALL

.stext   0x80000100                // System call 0 function handler
. . .                              // Instructions that handle scall 0
jr       e31                       // Return to main handler

.stext   0x80000254                // System call 1 function handler
. . .                              // Instructions that handle scall 1
jr       e31                       // Return to main handler
```

7.7 Synchronous Exceptions

A synchronous exception can be raised by a running instruction for a number of reasons. Here is a short list of common synchronous exceptions that are defined by the hardware:

ECODE	Description
0	Invalid instruction with undefined opcode or function code
1	Illegal access to a system register
2	Misaligned address exception caused by a load or store instruction
3	Illegal Instruction address: program is not allowed to fetch instruction at the given address
4	Illegal Load address: program is not allowed to load data at the given address
5	Illegal Store address: program is not allowed to store data at the given address
> 5	Reserved for future use

The above list is not complete. Other synchronous exceptions can be added as needed. The **EPC** register holds the address of the instruction that caused the exception. If **ECODE** is **2** to **5**, the bad address that caused the exception is saved in the **EADDR** register. The above synchronous exceptions terminate the execution of the running program with an error message that indicates which instruction (at what address) caused the exception. In particular, if **ECODE** is **3** to **5** then the address generated by the program violates memory protection enforced by the operating system and architecture. For example, a program is not allowed to address directly the memory area occupied by the operating system. Similarly, a program is not allowed to read and write its text segment in memory.

On the other hand, there are synchronous exceptions (not listed above) that do not terminate program execution. For example, a page fault occurs when a virtual page does not exist in main memory. The page fault exception handler allocates a page in main memory and maps the virtual page to its physical page using a page table. A program can restart the instruction that caused the page fault.

Handling synchronous exceptions is similar to handling system calls. The synchronous exception handler starts at **EBASE + 0x80** (address **0x80000080**). It uses **ecode** to address an array of exception pointers, loads the function address into **e2**, and jumps into the exception handling function (**jr** instruction). Depending on **ecode**, the exception handling function might terminate the execution of the program, or resume program execution by executing the **eret** instruction.

```
.stext   0x80000080               // System text segment at 0x80000080
set      e1 = @exception_pointers // e1 = array address = 0x8002F3AC
lw       e2 = [e1, ecode, 2]      // load e2 = exception function address
jr       e2                       // jump to exception handling function
```

7.8 Asynchronous Interrupts

An interrupt is described as ***asynchronous***. It is caused by the hardware. For example, it can be caused by a hardware device, such as a timer, a disk controller, a keyboard, or a mouse that want attention from the CPU. It can also be caused by a hardware reset or malfunction that terminates the execution of the program. Handling asynchronous interrupts is similar to handling synchronous exceptions. The bits in the **INTS** (Interrupt Status) register specify which interrupts are pending. The bits in the **INTM** (Interrupt Mask) register can be modified to enable or disable interrupts.

This section is left empty for details about asynchronous interrupts.

7.9 Counter Registers

Counter registers are used for performance evaluation. They are constantly updated by the hardware. Thirty-two 64-bit counter registers are defined: **C0** to **C31**. They are read-only and cannot be written by the software. They can be read at any execution level (**EL0** or **EL1**) without causing any exception.

Hardware counters are cleared by hardware reset. They always count up as long as they are connected to a power supply. Only five counter registers are defined in the table below. The remaining ones are reserved for future use. The **ICNT0** and **ICNT1** registers count the number of instructions that have completed at **EL0** and **EL1** since the last reset. The **CCNT0** and **CCNT1** registers count the number of processor cycles used at **EL0** and **EL1**. The **TIME** register counts the number of microseconds since the last reset. A program can read these registers at the beginning and end of its execution to determine its instruction count, clock cycles, and execution time.

Other counter registers can also be added to the hardware to count the number of cache misses in the instruction, data, and other caches. These counter registers increase the cost, but are useful for system performance evaluation.

If a counter register is undefined, then it is not implemented in the hardware. Reading an undefined counter register always returns 0. Writing an undefined register has no effect.

Number	Name	Description
C0	**ICNT0**	Count of Instructions that have completed at EL0 since last reset
C1	**ICNT1**	Count of Instructions that have completed at EL1 since last reset
C2-C3		Reserved for future use
C4	**CCNT0**	Count of Clock Cycles at EL0 since last reset
C5	**CCNT1**	Count of Clock Cycles at EL1 since last reset
C6-C7		Reserved for future use
C8	**TIME**	Time in microseconds since last reset
C9-C31		Reserved for future use

7.10 System Instructions

Three system instructions are defined to initiate a system call (**SCALL**), a debugger call (**DCALL**), or a return from an exception (**ERET**). They all use the same **SYS** opcode, which is of the I-Format.

	6	5	5	4	12
I	**Op = SYS**	**//**	**//**	**f**	**Imm12**

Opcode: **Op = SYS**
Three functions: **f = SCALL, DCALL, SRET**

Assembly Language Syntax:

```
SCALL    Imm12                        // System Call
DCALL    Imm12                        // Debugger Call
SRET     Imm12                        // System Return
```

Imm12: 12-bit service code used by **SCALL** and **DCALL**
Imm12: 12-bit signed offset used by **SRET** to return to the same instruction, next one, or elsewhere

7.11 Special Move Instructions

Special move instructions copy data between different register files. Four instructions are defined: **MOVRC** (**MOV Rd = Ca**), **MOVEC** (**MOV Ed = Ca**), **MOVER** (**MOV Ed = Ra**), and **MOVRE** (**MOV Rd = Ea**). Moving data to a **C** register is not possible because it is read-only. Only one **MOV** opcode (R-Format) is defined with four functions.

	6	5	5	4	2	5	5
R	**Op = MOV**	**a**	**//**	**f**	**x**	**//**	**d**

Opcode: **Op = MOV**
Four functions: **f = MOVRC, MOVEC, MOVER, MOVRE**

Assembly Language Syntax:

```
MOV      Rd = Ca                      // MOVRC can run at EL0
MOV      Ed = Ca                      // MOVEC is restricted to EL1
MOV      Ed = Ra                      // MOVER is restricted to EL1
MOV      Rd = Ea                      // MOVRE is restricted to EL1
```

For clarity, the assembly-language syntax uses only one **MOV** mnemonic. The assembler recognizes the function (**MOVRC**, **MOVEC**, **MOVER**, or **MOVRE**) from the instruction syntax. **MOVRC** is not privileged and can run at **EL0** or **EL1**. However, **MOVEC**, **MOVER**, and **MOVRE** are privileged and can be executed only by system software running at **EL1**. They cause an exception if executed at **EL0**.

7.12 Summary of System Instructions

The **SYS** opcode (I-Format) defines three function codes: **SCALL, DCALL,** and **SRET**. It ignores the **a** and **b** register fields.

	6	5	5	4	12
I	SYS = 60	//	//	SCALL=0	Imm12
I	SYS = 60	//	//	DCALL=1	Imm12
I	SYS = 60	//	//	SRET=2	Imm12

The **MOV** opcode (R-Format) defines four function codes: **MOVRC**, **MOVEC**, **MOVER**, and **MOVRE**. It ignores the **b** and **c** fields. The **x** field specifies the minimum execution level of the instruction. **MOVRC** can run at **EL0** or higher (**x=0**). However, **MOVEC**, **MOVER**, and **MOVRE** are privileged instructions that should run at **EL1** (**x=1**).

	6	5	5	4	2	5	5
R	MOV = 61	a	//	MOVRC=0	x=0	//	d
R	MOV = 61	a	//	MOVEC=0	x=1	//	d
R	MOV = 61	a	//	MOVER=1	x=1	//	d
R	MOV = 61	a	//	MOVRE=2	x=1	//	d

If an opcode or function code is not defined then it should not be used. Otherwise, the **Invalid Instruction exception** is raised.