RISC-V Introduction

E. Sanchez

Politecnico di Torino
Dipartimento di Automatica e Informatica

RISC-V

RISC-V is an open standard Instruction Set Architecture (ISA), having few instructions in 3 different versions: 32, 64, and 128 bits.

The RISC-V ISA is defined as a base integer ISA plus extensions and user customizations.

In the original name, V states for 5, variations and vectors.

The main interest is due to a common free and open standard to which software can be ported and running on any developed RISC-V hardware.

RISC-V (II)

History:

- In May 2010 was proposed as part of the Parallel Computing Laboratory (Par Lab) at UC Berkeley, leaded by Prof. David Patterson
- RISC-V (fifth version of the ISA), then V means the roman number, but also the possibility to add new variants and vector instructions
- In January 2015, the first RISC-V Workshop was held, and the RISC-V Foundation started with 36 Founding Members

RISC-V (III)

History:

- **In November 2018, the RISC-V Foundation announced a joint collaboration with the Linux Foundation.**
- **In 2024, the RISC-V foundation counts with more than 200 members including a wide range of companies, academic institutions, and individuals.**

RISC-V Generalities

- Freely licensed open standard
- Load-store architecture
- Fixed 32-bit instruction set
- Additional ISA extensions, for example for security, floating point, and vector operations
- Addressing modes:
  - Displacement
  - Immediate
  - Register indirect

RISC-V Generalities (II)

Data types:

- **Byte (8 bits)**
- **Half Words (16 bits)**
- **Words (32 bits)**
- **Double Words (64 bits)**
- **32-bit single precision floating-point**
- **64-bit double precision floating-point.**

Relative (+-) number representation: two's complement

Given a binary number A of N bits, two's complement (2C) can be defined as :

$$\overline{\overline{A}} = 2^N - A = \overline{A} + 1$$

Using a 3-bit representation :

$+ 3_{10} \rightarrow 011_{2C}$

$- 1_{10} \rightarrow 111_{2C}$

$000_{2C} \rightarrow + 0_{10}$

$100_{2C} \rightarrow - 4_{10}$

Binary	Decimal
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

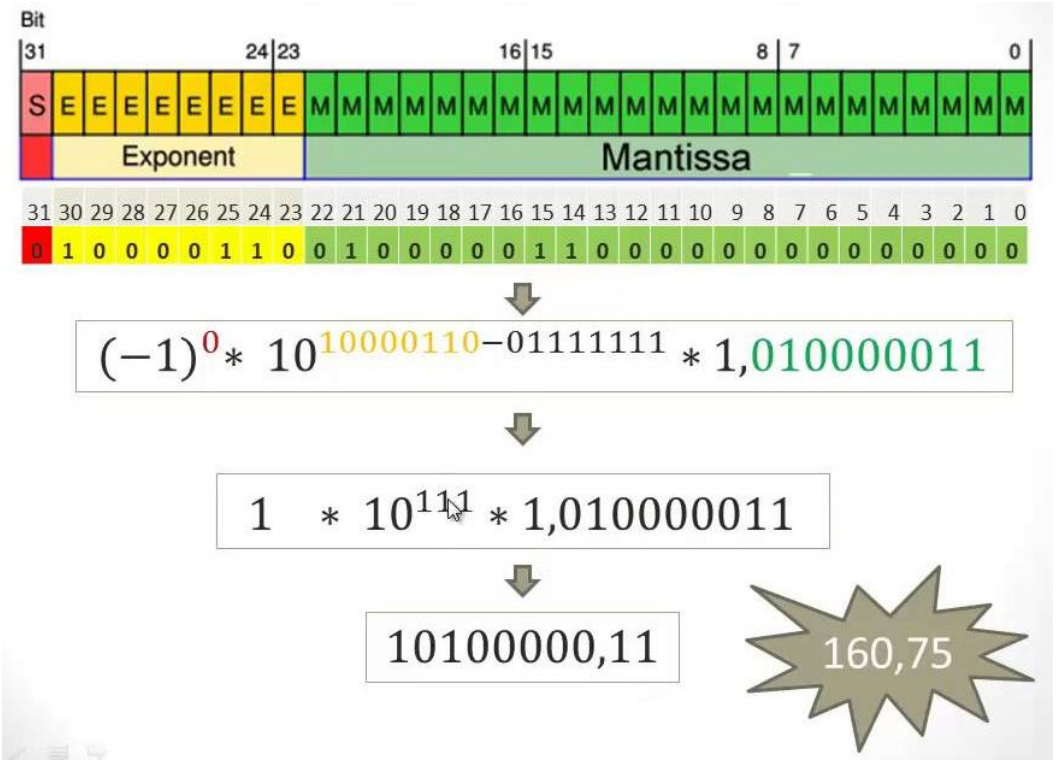
IEEE 754 32-bit representation

The IEEE 754 representation is composed by:

Sign: 1 bit

Exponent: 8 bits

Mantissa: 23 bits.



Instruction length encoding

RISC-V allows a variable encoding in multiples of 16-bit blocks if code size reduction or specific implementations are needed.

- **Fixed-length 32-bit instructions for the base versions**
  - **32-bit instructions have their lowest two bits set to 11**
  - **16-bit instructions may have their lowest two bits equal to 00, 01, or 10**
- **In the case of 32-bit instructions, code memory alignment is a constraint, in other cases, the alignment is relaxed**
- **Little-endian memory system.**

RISC-V Instruction set organization

It counts with three base instructions sets plus a variety of optional extensions.

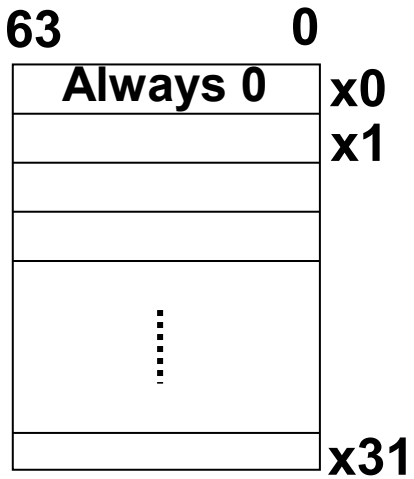
- **For example, a 64-bit RISC-V possible ISA is defined as RV64IMAFD, known as RV64G: general-purpose scalar instruction set.**
- **The 32-bit version is called RV32G.**

Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

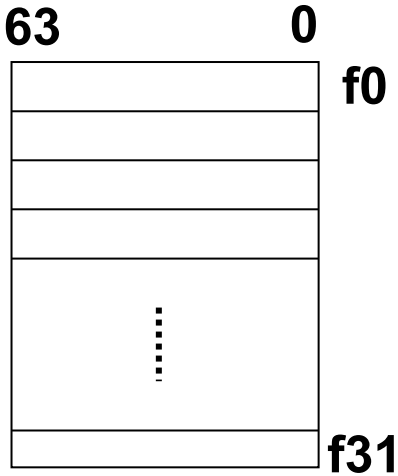
RISC-V registers

RV64G register bank:

- Register bank or register file of 32 elements
- 32 Integer registers
  - x0, x1, x2… x31
  - x0 is hardwired to 0.
- Floating point single (32-bit) and/or double precision (64-bit).



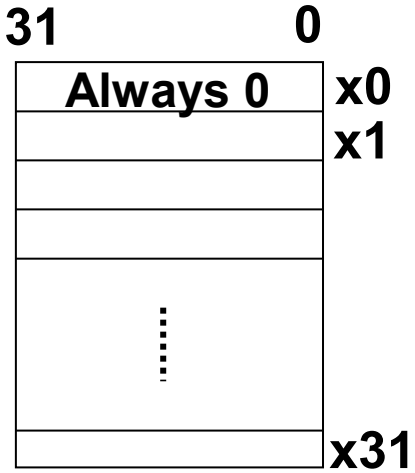
General-purpose Integer Registers



Floating-point Registers (FPRs)

RISC-V registers for 32-bit version

RV32I register bank:



General-purpose Integer Registers

Register	ABI Name	Description	Register	ABI Name	Description
x0	zero	hardwired zero	x16	a6	function argument 6
x1	ra	return address	x17	a7	function argument 7
x2	sp	stack pointer	x18	s2	saved register 2
x3	gp	global pointer	x19	s3	saved register 3
x4	tp	thread pointer	x20	s4	saved register 4
x5	t0	temporary register 0	x21	s5	saved register 5
x6	t1	temporary register 1	x22	s6	saved register 6
x7	t2	temporary register 2	x23	s7	saved register 7
x8	s0 / fp	saved register 0 / frame pointer	x24	s8	saved register 8
x9	s1	saved register 1	x25	s9	saved register 9
x10	a0	function argument 0 / return value 0	x26	s10	saved register 10
x11	a1	function argument 1 / return value 1	x27	s11	saved register 11
x12	a2	function argument 2	x28	t3	temporary register 3
x13	a3	function argument 3	x29	t4	temporary register 4
x14	a4	function argument 4	x30	t5	temporary register 5
x15	a5	function argument 5	x31	t6	temporary register 6

ABI: Application Binary Interface

Addressing Modes

Addressing modes for data transfers are Immediate and Displacement, both counting with an encoded field of 12 bits.

Immediate Addressing Mode

- `addi x1, x2, 32`

$$x1 \leftarrow x2 + 32$$

- `addi x1, x0, 32`

$$x1 \leftarrow 32$$

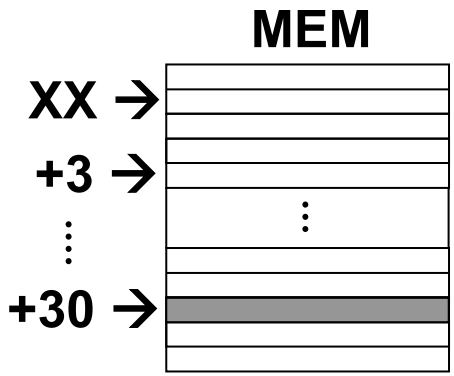
Addressing Modes

Displacement

- `lw x1, 30(x2)`

x2 = XX

x1 ← MEM[x2 + 30]



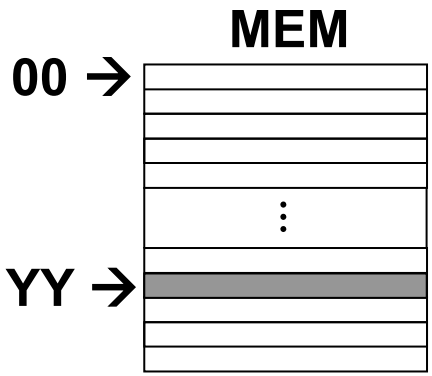
Addressing Modes

Displacement – Register indirect

- `lw x1, 0(x2)` → *Register Indirect*

x2 = YY

x1 ← MEM[x2]

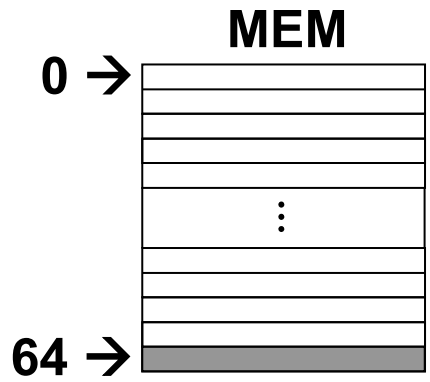


Addressing Modes

Displacement – Absolute Addressing

- `lw x1, 64(x0)` → ***Absolute Addressing***

x1 ← MEM[64]



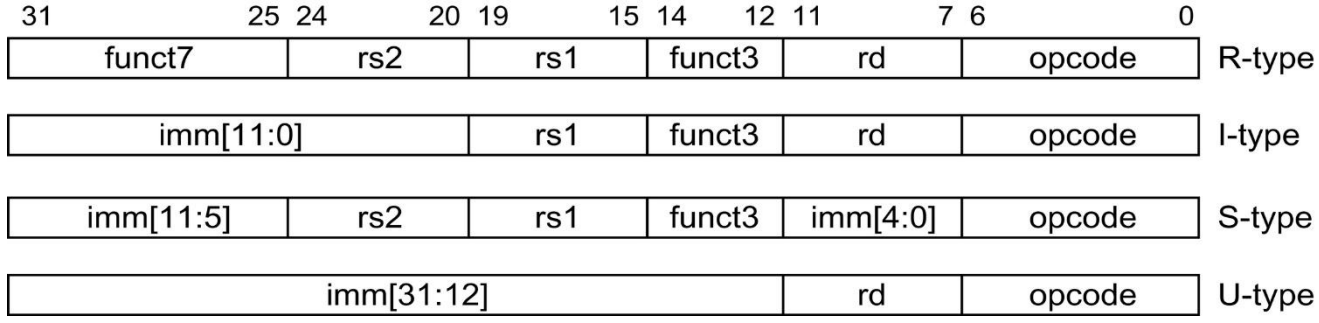
Instruction Format

A CPU instruction is a single 32-bit aligned word, that may include:

- **a 7-bit primary opcode**
- **a 12-bit field for displacement addressing, immediate or PC relative branches.**

The CPU instruction formats are:

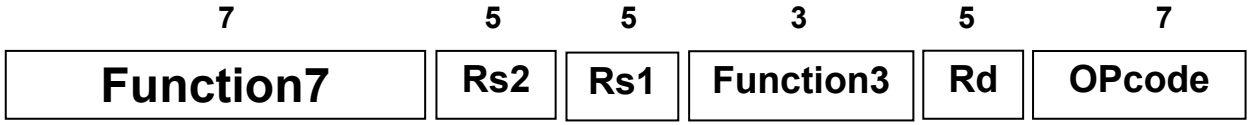
- **Register: R-type**
- **Immediate: I-Type**
- **Store: S-Type**
- **Jump: U-Type**



Instruction Format – Register

R-Type instructions

Register – Register operations.

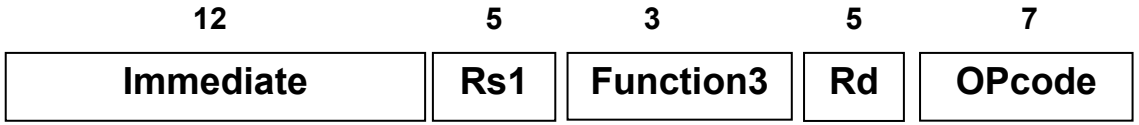


Field	Description
Opcode	7-bit primary operation code
Rs1 – Rs2	5-bit specifier for the source registers
Rd	5-bit specifier for the destination register
Function3	SLTI / ADDI / ANDI /ORI / XORI / SUB / SRA
Function7	Help on selecting the specific type of operation

Instruction Format – Immediate

I-Type instructions

Immediate value is loaded with a sign extension if required.

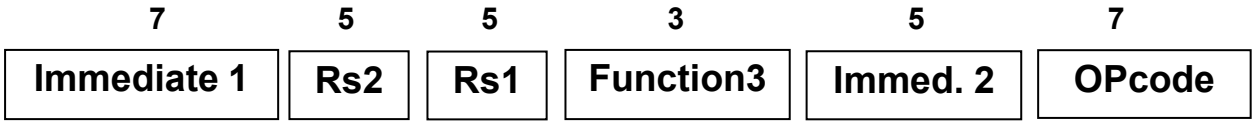


Field	Description
Opcode	7-bit primary operation code
Rs1	5-bit specifier for the source register
Rd	5-bit specifier for the destination register
Function3	SLTI / ADDI / ANDI /ORI / XORI / Width
Immediate	12-bit signed *immediate* used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement

Instruction Format – Store

S-Type instructions

Store, compare and branch instructions.

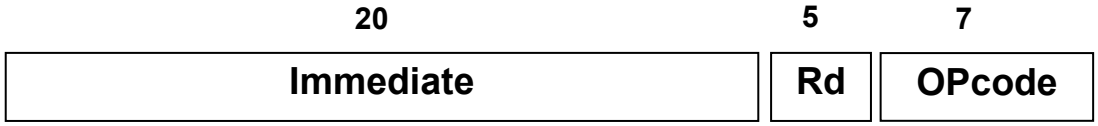


Field	Description
Opcode	STORE
Rs2	5-bit specifier for the source register to copy in memory
Rs1	5-bit specifier for the register to add to the sign extended 12-bits immediate value [Imm1][Imm2]
Function3	SLTI / ADDI / ANDI /ORI / XORI / SUB / SRA
Immediate 1 and 2	These two values compose the 12-bit signed value.

Instruction Format – Jump

U-Type instructions

Control instructions for unconditional and conditional branches.



Field	Description
Opcode	7-bit primary operation code
Rd	5-bit specifier for the destination register for returning the PC
Immediate	20-bit signed *offset value* used PC-relative branches

RISC-V instructions

Instructions of the RV32G, grouped by function:

- **Load and store**
- **ALU operations**
- **Branches and Jumps**
- **Floating Point**
- **Miscellaneous**

In the following some of the most used instructions and pseudoinstructions are described.

Load and Store

- **RISC-V processors use a load-store architecture.**
- **The main memory is accessed only through load and store instructions.**

Load – Examples

`lw` load word

```
lw x2, 28(x8)            #  x2 ← MEM[x8 + 28]
```

`lb` load in register a signed Byte

```
lb x1, 28(x8) #  x1 ← ([MEM[x8 + 28]]7)24 ## MEM[x8 + 28]
```

`lbu` load Byte unsigned

```
lbu x1, 28(x8)     #  x1 ← 024 ## MEM[x8 + 28]
```

Load – Examples (II)

`lw` load a word variable named *VAR* in a register

```
lw x2, VAR              #lw is a pseudoinstruction in this case
```

Translated as:

```
                        # auipc → add upper immediate to pc
auipc x2, 0x10000       # x2 ← PC + 0x10000000, VAR effective address
lw x2, 0(x2)            # x2 ← MEM[VAR]
```

la load absolute address in a register

```
la x5, VECT                 #la is a pseudoinstruction
```

Translated as:

```
auipc x5, 0x10000   #  x5 ← PC + 0x10000000
addi x5, x5, -23    #  x5 ← x5 – 23  (effective address of VECT)
```

Store – Examples

`sw` **store word**

```
sw x2, 28(x8)   # MEM[x8 + 28] ← x2 that is a 32-bit register
```

`sb` **store the LSB of a register in memory**

```
sb x1, 28(x8)  # [MEM[x8 + 28] ←8 x1  - saves only the x1 LSB
```

`sh` **store the first two LSB of a register in memory**

```
sh x1, 28(x8)  # [MEM[x8 + 28] ←16 x1  - the first 2 LSB
```

Store – Examples (II)

`sw` **store a register value in a word variable named *VAR***

```
la x2, VAR     #no pseudoinstructions for the store is available
sw x5, 0(x2)
```

Translated as:

```
auipc x2, 0x10000       #  x2 ← PC + 0x10000000
addi x2, x2, -48        #  x2 ← effective address of VAR
sw x5, 0(x2)            # MEM[VAR] ← x5
```

Load / Store additional examples

Example instruction	Instruction name	Meaning
`ld x1,80(x2)`	Load doubleword	Regs[x1] ← Mem[80+Regs[x2]]
`lw x1,60(x2)`	Load word	Regs[x1] $\leftarrow_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
`lwu x1,60(x2)`	Load word unsigned	Regs[x1] $\leftarrow_{64}$ 0^{32} ## Mem[60+Regs[x2]]
`lb x1,40(x3)`	Load byte	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
`lbu x1,40(x3)`	Load byte unsigned	Regs[x1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[x3]]
`lh x1,40(x3)`	Load half word	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
`flw f0,50(x3)`	Load FP single	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x3]] ## 0^{32}
`fld f0,50(x2)`	Load FP double	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x2]]
`sd x2,400(x3)`	Store double	Mem[400+Regs[x3]] $\leftarrow_{64}$ Regs[x2]
`sw x3,500(x4)`	Store word	Mem[500+Regs[x4]] $\leftarrow_{32}$ Regs[x3]$_{32..63}$
`fsw f0,40(x3)`	Store FP single	Mem[40+Regs[x3]] $\leftarrow_{32}$ Regs[f0]$_{0..31}$
`fsd f0,40(x3)`	Store FP double	Mem[40+Regs[x3]] $\leftarrow_{64}$ Regs[f0]
`sh x3,502(x2)`	Store half	Mem[502+Regs[x2]] $\leftarrow_{16}$ Regs[x3]$_{48..63}$
`sb x2,41(x3)`	Store byte	Mem[41+Regs[x3]] $\leftarrow_{8}$ Regs[x2]$_{56..63}$

ALU operations

All operations are performed on operands held in the processor registers

Instruction types

- **Immediate and three-operand Instructions**
- **Shift instructions**
- **Multiply and divide instructions if M extension is available**

2's complement arithmetic

- **Add**
- **Subtract**
- **Multiply**
- **Divide.**

ALU operations: x0 usage

- **Loading a constant in a register**

  addi **immediate with x0 as source operand**

```
addi x1, x0, 10      #x1 ← 10
```

- **Mov register to register**

  add **x2 and x0 as source operands**

```
add  x1,x0,x2      #x1 ← x2
```

ALU – shift and slt operations

sll shift left logical x3 ← x2, x1 times

```
sll  x3,x2,x1      #x3 ← x2<<x1
```

slli shift left logical x3 ← x2, #immediate value times

```
slli  x3,x2,4      #x3 ← x2<<4
```

slti set less than immediate

```
slti  x3,x2,4     #IF (x2 < 4) x3 ← 1
                  #ELSE x3 ← 0
```

ALU – shift and slt operations (II)

seqz set if *rs* = 0

```
        seqz x1,x2
    Translated as:
        sltiu x1, x2, 1     #IF (x2 == 0) x1 ← 1
                            #ELSE x1 ← 0
```

snez set if *rs* ≠ 0

```
        snez x1,x2
    Translated as:
        sltu x1, x0, x2     #IF (x2 != 0) x1 ← 1
                            #ELSE x1 ← 0
```

ALU – other examples

lui Load Upper Immediate

It places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.

```
lui x1,0x12345     #x1 ← 0x12345 ## 0^11..0
```

It is possible to build 32-bit constants as:

```
lui x1,0x12345          #x1 ← 0x12345 ## 0^11..0

addi x1, x1, 0x678      #x1 ← x1 + 0x678
                        #x1 ← 0x12345678
```

Branches and Jumps

Control transfer instructions are mainly unconditional jumps and conditional branches.

- **PC-relative jumps**
- **Jump to register**
- **Conditional branches.**

Jump instructions

j label a PC relative jump having a ± 1MiB range.

```
        j label1
   Translated as:
        jal  x0, label1
```

jal label a PC relative jump and link having a ± 1MiB range saves the return value in x1.

```
        jal label1
   Translated as:
        jal  x1, label1
```

Jump instructions (II)

jal x3, label a PC relative jump and link having a ± 1MiB range saves the return value in x3.

```
jal x3, label
```

jr rs1 an indirect jump to register. ± 2kiB + rs1 range

```
jr x3
Translated as:
jalr x0, 0(x3)
```

Jump instructions (III)

jalr rs1 an indirect jump and link to register. ± 2kiB + rs1 range
saves the return value in x1.

```
        jalr x3
  Translated as:
        jalr x1, 0(x3)
```

jalr rd, rs, offset an indirect jump and link to register.
saves the return value in rd.

```
    jalr x3, x2, 0  #jumps to x2+0, saves return in x3
```

Jump instructions (IV)

call sub1 call a subroutine sub1. return address in x1.

```
        call sub1
    Translated as:
        auipc x1, sub1  # x1 ← sub1 relative addr.
        jalr x1, offset(x1)
```

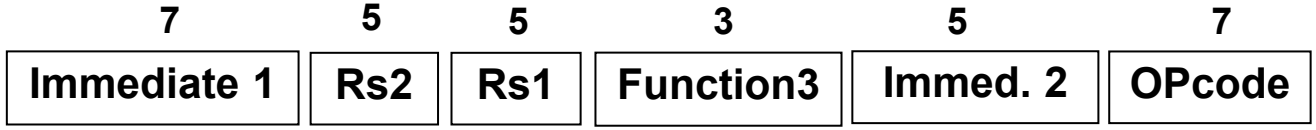
ret return from a subroutine

```
        ret
    Translated as:
        jalr x0, 0(x1)
```

Branch instructions

B-Type instructions

- **Special type for branch instructions with a range of ± 4kiB and a 12-bit immediate value.**



- **The offset must be a multiple of 2.**
- **Branch instructions compare two registers Rs1 and Rs2**

Branch instructions (II)

B-Type instructions

- **BEQ and BNE take the branch if rs1 and rs2 are equal or unequal.**
- **BLT and BLTU take the branch if rs1 is less than rs2**
- **BGE and BGEU take the branch if rs1 is greater than or equal to rs2**
- **BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU.**

Branch instructions (II)

B-Type instructions

- **BEQ and BNE take the branch if rs1 and rs2 are equal or unequal.**
- **BLT and BLTU take the branch if rs1 is less than rs2**
- **BGE and BGEU take the branch if rs1 is greater than or equal to rs2**
- **BGT, BGTU, ... can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU.**

Signed and unsigned comparisons are allowed

Branch instructions – examples

beqz rs, label **branch if rs = 0**

```
        beqz x3, label
    Translated as:
        beq x3, x0, label
```

bnez rs, label **branch if rs ≠ 0**

```
        bnez x3, label
    Translated as:
        bne x3, x0, label
```

Branch instructions – examples (II)

bltz rs, label branch if rs < 0

```
        bltz x3, label
    Translated as:
        blt x3, x0, label
```

bgtz rs, label branch if rs > 0

```
        bgtz x3, label
    Translated as:
        blt x0, x3, label
```

ISA for the RV64G

- **RV64G instruction set:**
- **This is the 64-bit RISC-V possible ISA defined as RV64IMAFD general-purpose scalar instruction set.**
- **The 32-bit version is called RV32G.**

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement + contents of a GPR*
`lb, lbu, sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh, lhu, sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw, lwu, sw`	Load word, store word (to/from integer registers)
`ld, sd`	Load doubleword, store doubleword
Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
`add, addi, addw, addiw, sub, subi, subw, subiw`	Add and subtract, with both word and immediate versions
`slt, sltu, slti, sltiu`	set-less-than with signed and unsigned, and immediate
`and, or, xor, andi, ori, xori`	and, or, xor, both register-register and register-immediate
`lui`	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
`auipc`	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
`sll, srl, sra, slli, srli, srai, sllw,slliw, srli, srliw, srai, sraiw`	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
`mul, mulw, mulh, mulhsu, mulhu, div,divw, divu, rem, remu, remw, remuw`	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions
Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
`jal,jalr`	Jump and link address relative to a register or the PC
Floating point	*All FP operation appear in double precision (.d) and single (.s)*
`flw, fld, fsw, fsd`	Load, store, word (single precision), doubleword (double precision)
`fadd, fsub, fmult, fiv, fsqrt, fmadd, fmsub, fnmadd, fnmsub, fmin, fmax, fsgn, fsgnj, fsjnx`	Add, subtract, multiply, divide, square root, multiply-add, multiply-subtract, negate multiply-add, negate multiply-subtract, maximum, minimum, and instructions to replace the sign bit. For single precision, the opcode is followed by: .s, for double precision: .d. Thus fadd.s, fadd.d
`feq, flt, fle`	Compare two floating point registers; result is 0 or 1 stored into a GPR
`fmv.x.*, fmv.*.x`	Move between the FP register abd GPR, "*" is s or d
`fcvt.*.l, fcvt.l.*, fcvt.*.lu, fcvt.lu.*, fcvt.*.w, fcvt.w.*, fcvt.*.wu, fcvt.wu.*`	Converts between a FP register and integer register, where "*" is S or D for single or double precision. Signed and unsigned versions and word, doubleword versions

Miscellaneous instructions

nop **No operation instruction**

```
addi   x0,x0,0      #nop
```