

Lab 1

Warm-up : the target machine : RISC-V

Objective

- Be familiar with the RISC-V instruction set.
- Understand how it executes on the RISC-V processor with the help of a simulator.
- Write simple programs, assemble, execute.

1.1 The RISC-V processor, instruction set, simulator

EXERCISE #1 ► Lab preparation

Clone the Git repository for this year's labs:

```
git clone https://github.com/Drup/cap-lab22.git
```

If you haven't done so already, follow the instructions to compile `riscv-xxx-gcc` and `spike` on your machine (see `INSTALL.md` file).

EXERCISE #2 ► RISC-V C-compiler and simulator, first test

In the directory `TP01/riscv/`:

- Compile the provided file `ex1.c` with:
`riscv64-unknown-elf-gcc ex1.c -o ex1.riscv`
It produces a RISC-V binary named `ex1.riscv`.
- Execute the binary with the RISC-V simulator:

```
spike pk ex1.riscv
```

This should print:

```
bb1 loader
42
```

If you get a runtime exception, try running `spike -m100 pk ex1.riscv` instead: this limits the RAM usage of `spike` to 100 MB (the default is 2 GB).

- The corresponding RISC-V code can be obtained in a more readable format by:
`riscv64-unknown-elf-gcc ex1.c -S -o ex1.s -fverbose-asm`
(have a look at the generated `.s` file!)

The objective of this sequence of labs is to design **our own (subset of) C compiler for RISC-V**.

EXERCISE #3 ► Documents

Some documentation can be found in the RISC-V ISA on the course webpage and in Appendix A.

<https://compil-lyon.gitlabpages.inria.fr/>

In the architecture course, you already saw a version of the target machine RISC-V. The instruction set is depicted in Appendix A.

1.1.1 Hand exercises

EXERCISE #4 ► TD

On paper, write (in RISC-V assembly language) a program which initializes the t_0 register to 1 and increments it until it becomes equal to 8.

EXERCISE #5 ► TD : sum

Write a program in RISC-V assembly that computes the sum of the 10 first positive integers (excluded 10).

1.1.2 Assembling, disassembling

EXERCISE #6 ► Hand assembling, simulation of the hex code

Assemble by hand (on paper) the instructions:

```

1      .globl main
2 main:
3      addi a0, a0, 1
4      bne a0, a0, main
5 end:
6      ret

```

You will need the set of instructions of the RISC-V machine and their associated opcode. All the info is in the ISA documentation.

To check your solution (**after** you did the job manually), you can redo the assembly using the toolchain:

```
riscv64-unknown-elf-as -march=rv64g asshand.s -o asshand.o
```

asshand.o is an ELF file which contains both the compiled code and some metadata (you can try `hexdump asshand.o` to view its content, but it's rather large and unreadable). The tool `objdump` allows extracting the code section from the executable, and show the binary code next to its disassembled version:

```
riscv64-unknown-elf-objdump -d asshand.o
```

Check that the output is consistent with what you found manually.

EXERCISE #7 ► Hand disassembling

Guess a RISC-V program that assembles itself into :

Listing 1.1: disass.lst

```
disass.o: format de fichier elf64-littleriscv
```

Déassemblage de la section .text:

```

0000000000000000 <main>:
0: 00128313  xx
4: ffdff06f  yy
8: 00008067  zz

```

From now on, we are going to write programs using an easier approach. We are going to write instructions using the RISC-V assembly.

1.2 RISC-V Simulator

EXERCISE #8 ► Execution and debugging

See <https://www.lowrisc.org/docs/tagged-memory-v0.1/spike/> for details on the Spike simulator.

`test_print.s` is a small but complete example using Risc-V assembly. It uses the `println_string`, `print_int`, `print_char` and `newline` functions provided to you in `libprint.s`. Each function can be called with `call print_...` and prints the content of register `a0` (`call newline` takes no input and prints a new-line character).

1. First test assembling and simulation on the file `test_print.s`:

```
riscv64-unknown-elf-as -march=rv64g test_print.s -o test_print.o
```
2. Optionally, run `riscv64-unknown-elf-objdump -D` as in previous exercise. The `-D` option shows all sections, including `.rodata`.
3. The `libprint.s` library must be assembled too:

- ```
riscv64-unknown-elf-as -march=rv64g libprint.s -o libprint.o
```
- We now link these files together to get an executable:  

```
riscv64-unknown-elf-gcc test_print.o libprint.o -o test_print
```

The generated `test_print` file should be executable, but since it uses the Risc-V ISA, we can't execute it natively (try `./test_print`, you'll get an error like `Exec format error`).
  - Run the simulator:  

```
spike pk ./test_print
```

The output should look like:  

```
bbl loader
HI MIF08!
42
a
```

The first line comes from the simulator itself, the next two come from the `println_string`, `print_int` and `print_char` calls in the assembly code.
  - We can also view the instructions while they are executed:  

```
spike -l pk ./test_print
```

Unfortunately, this shows all the instructions in `pk` (Proxy Kernel, a kind of mini operating system), and is mostly unusable. Alternatively, we can run a step-by-step simulation starting from a given symbol. To run the instructions in `main`, we first get the address of `main` in the executable:  

```
$ riscv64-unknown-elf-nm test_print | grep main
0000000000001014c T main
```

This means: `main` is a symbol defined in the `.text` section (T in the middle column), it is global (capital T), and its address is `1014c` (you may not have the same address). Now, run `spike` in debug mode (`-d`) and execute code up to this address (`until pc 0 1014c`, i.e. "Until the program counter of core 0 reaches `1014c`"). Press Return to move to the next instruction and `q` to quit:  

```
$ spike -d pk ./test_print
: until pc 0 1014c
bbl loader
:
core 0: 0x0000000000001014c (0xff010113) addi sp, sp, -16
:
core 0: 0x00000000000010150 (0x00113423) sd ra, 8(sp)
:
core 0: 0x00000000000010154 (0x0000e517) auipc a0, 0xe
:
core 0: 0x00000000000010158 (0x41450513) addi a0, a0, 1044
: q
$
```

**Remark:** For your labs, you may want to assemble and link with a single command (which can also do the compilation if you provide `.c` files on the command-line):

```
riscv64-unknown-elf-gcc -march=rv64g libprint.s test_print.s -o main
```

In real-life, people run compilation+assembly and link as two different commands, but use a build system like a `Makefile` to re-run only the right commands.

### EXERCISE #9 ► Algo in RISC-V assembly

Write (in `minmax.s`) a program in RISC-V assembly that computes the min of two integers, and stores the result in a precise location of the memory that has the label `min`. Try with different values. We use 64 bits of memory to store ints, i.e. use `.dword` directive and `ld` and `sd` instructions.

### EXERCISE #10 ► (Advanced) Algo in RISC-V assembly

Write and execute the following programs in assembly:

- Count the number of non-nul bits of a given integer, print the result.

- Draw squares and triangles of stars (character `'*'`) of size  $n$ ,  $n$  being stored somewhere in memory.

Examples:

$n=3$  square:

```

```

```

```

```

```

$n=3$  triangle:

```
 *
```

```
 * *
```

```
* * *
```

The function `print_char` expects the ASCII encoding of the character you want to print.

### 1.3 Finished?

If you're done with the lab, do the python tutorial at the following address:

<https://docs.python.org/fr/3.10/tutorial/>

# Appendix A

## RISCV Assembly Documentation (ISA), rv64g

### About

- RISCV is an open instruction set initially developed by Berkeley University, used among others by Western Digital, Alibaba and Nvidia.
- We are using the rv64g instruction set: **Risc-V**, 64 bits, **General purpose** (base instruction set, and extensions for floating point, atomic and multiplications), without compressed instructions. In practice, we will use only 32 bits instructions (and very few of floating point instructions).
- Document: Laure Gonnord and Matthieu Moy, for CAP and MIF08.

This is a simplified version of the machine, which is (hopefully) conform to the chosen simulator.

### A.1 Installing the simulator and getting started

To get the RISCV assembler and simulator, follow instructions of the first lab (git pull on the course lab repository).

### A.2 The RISCV architecture

Here is an example of RISCV assembly code snippet (a proper main function would be needed to execute it, cf. course and lab):

```
1 addi a0, zero, 17 # initialisation of a register to 17
2 loop:
3 addi a0, a0, -1 # subtraction of an immediate
4 j loop # equivalent to jump xx
```

The rest of the documentation is adapted from <https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md> and <https://github.com/jameslzh/riscv-card/blob/master/riscv-card.pdf>

### A.3 RISC-V Assembly Programmer's Manual - adapted for CAP and MIF08

#### A.3.1 Copyright and License Information - Documents

The RISC-V Assembly Programmer's Manual is

© 2017 Palmer Dabbelt [palmer@dabbelt.com](mailto:palmer@dabbelt.com) © 2017 Michael Clark [michaeljclark@mac.com](mailto:michaeljclark@mac.com) © 2017 Alex Bradbury [asb@lowrisc.org](mailto:asb@lowrisc.org)

It is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). The full license text is available at <https://creativecommons.org/licenses/by/4.0/>.

- Official Specifications webpage: <https://riscv.org/specifications/>
- Latest Specifications draft repository: <https://github.com/riscv/riscv-isa-manual>

This document has been modified by Laure Gonnord & Matthieu Moy, in 2019.

### A.3.2 Registers

Registers are the most important part of any processor. RISC-V defines various types, depending on which extensions are included: The general registers (with the program counter), control registers, floating point registers (F extension), and vector registers (V extension). We won't use control nor F or V registers.

#### General registers

The RV32I base integer ISA includes 32 registers, named `x0` to `x31`. The program counter PC is separate from these registers, in contrast to other processors such as the ARM-32. The first register, `x0`, has a special function: Reading it always returns 0 and writes to it are ignored.

In practice, the programmer doesn't use this notation for the registers. Though `x1` to `x31` are all equally general-use registers as far as the processor is concerned, by convention certain registers are used for special tasks. In assembler, they are given standardized names as part of the RISC-V **application binary interface** (ABI). This is what you will usually see in code listings. If you really want to see the numeric register names, the `-M` argument to `objdump` will provide them.

| Register         | ABI                   | Use by convention                          | Preserved? |
|------------------|-----------------------|--------------------------------------------|------------|
| <code>x0</code>  | <code>zero</code>     | hardwired to 0, ignores writes             | <i>n/a</i> |
| <code>x1</code>  | <code>ra</code>       | return address for jumps                   | no         |
| <code>x2</code>  | <code>sp</code>       | stack pointer                              | yes        |
| <code>x3</code>  | <code>gp</code>       | global pointer                             | <i>n/a</i> |
| <code>x4</code>  | <code>tp</code>       | thread pointer                             | <i>n/a</i> |
| <code>x5</code>  | <code>t0</code>       | temporary register 0                       | no         |
| <code>x6</code>  | <code>t1</code>       | temporary register 1                       | no         |
| <code>x7</code>  | <code>t2</code>       | temporary register 2                       | no         |
| <code>x8</code>  | <code>s0 or fp</code> | saved register 0 <i>or</i> frame pointer   | yes        |
| <code>x9</code>  | <code>s1</code>       | saved register 1                           | yes        |
| <code>x10</code> | <code>a0</code>       | return value <i>or</i> function argument 0 | no         |
| <code>x11</code> | <code>a1</code>       | return value <i>or</i> function argument 1 | no         |
| <code>x12</code> | <code>a2</code>       | function argument 2                        | no         |
| <code>x13</code> | <code>a3</code>       | function argument 3                        | no         |
| <code>x14</code> | <code>a4</code>       | function argument 4                        | no         |
| <code>x15</code> | <code>a5</code>       | function argument 5                        | no         |
| <code>x16</code> | <code>a6</code>       | function argument 6                        | no         |
| <code>x17</code> | <code>a7</code>       | function argument 7                        | no         |
| <code>x18</code> | <code>s2</code>       | saved register 2                           | yes        |
| <code>x19</code> | <code>s3</code>       | saved register 3                           | yes        |
| <code>x20</code> | <code>s4</code>       | saved register 4                           | yes        |
| <code>x21</code> | <code>s5</code>       | saved register 5                           | yes        |
| <code>x22</code> | <code>s6</code>       | saved register 6                           | yes        |
| <code>x23</code> | <code>s7</code>       | saved register 6                           | yes        |
| <code>x24</code> | <code>s8</code>       | saved register 8                           | yes        |
| <code>x25</code> | <code>s9</code>       | saved register 9                           | yes        |
| <code>x26</code> | <code>s10</code>      | saved register 10                          | yes        |
| <code>x27</code> | <code>s11</code>      | saved register 11                          | yes        |
| <code>x28</code> | <code>t3</code>       | temporary register 3                       | no         |
| <code>x29</code> | <code>t4</code>       | temporary register 4                       | no         |
| <code>x30</code> | <code>t5</code>       | temporary register 5                       | no         |
| <code>x31</code> | <code>t6</code>       | temporary register 6                       | no         |
| <code>pc</code>  | <i>(none)</i>         | program counter                            | <i>n/a</i> |

*Registers of the RV32I. Based on RISC-V documentation and Patterson and Waterman "The RISC-V Reader" (2017)*

As a general rule, the **saved registers** `s0` to `s11` are preserved across function calls, while the **argument**

**registers** a0 to a7 and the **temporary registers** t0 to t6 are not. The use of the various specialized registers such as sp by convention will be discussed later in more detail.

### A.3.3 Instructions

#### Arithmetic

add, addi, sub, classically.

```
addi a0, zero, 42
```

initialises a0 to 42.

#### Labels

Text labels are used as branch, unconditional jump targets and symbol offsets. Text labels are added to the symbol table of the compiled module.

```
loop:
 j loop
```

Jumps and branches target is encoded with a relative offset in bytes. It is relative to the beginning of the current instruction. For example, the self-loop above corresponds to an offset of 0 bytes.

#### Branching

Test and jump, within the same instruction:

```
beq a0, a1, end
```

tests whether a0=a1, and jumps to 'end' if its the case.

#### Absolute addressing

The following example shows how to load an absolute address:

```
.section .text
.globl _start
_start:
 lui a0, %hi(msg) # load msg(hi)
 addi a0, a0, %lo(msg) # load msg(lo)
 jal ra, puts
2: j 2b

.section .rodata
msg:
 .string "Hello World\n"
```

which generates the following assembler output and relocations as seen by objdump:

```
0000000000000000 <_start>:
0: 000005b7 lui a1,0x0
 0: R_RISCV_HI20 msg
4: 00858593 addi a1,a1,8 # 8 <.L21>
 4: R_RISCV_L012_I msg
```

## Relative addressing

The following example shows how to load a PC-relative address:

```
.section .text
.globl _start
_start:
1: auipc a0, %pcrel_hi(msg) # load msg(hi)
 addi a0, a0, %pcrel_lo(1b) # load msg(lo)
 jal ra, puts
2: j 2b

.section .rodata
msg:
 .string "Hello World\n"
```

which generates the following assembler output and relocations as seen by objdump:

```
0000000000000000 <_start>:
0: 00000597 auipc a1,0x0
 0: R_RISCV_PCREL_HI20 msg
4: 00858593 addi a1,a1,8 # 8 <.L21>
 4: R_RISCV_PCREL_LO12_I .L11
```

## Load Immediate

The following example shows the `li` pseudo instruction which is used to load immediate values:

```
li a0, 0x76543210
```

which generates the following assembler output as seen by objdump (generated code will be different depending on the constant):

```
0: 76543537 lui a0,0x76543
4: 2105051b addiw a0,a0,528
```

## Load Address

The following example shows the `la` pseudo instruction which is used to load symbol addresses:

```
.section .text
.globl _start
_start:

 la a0, msg

.section .rodata
msg:
 .string "Hello World\n"
```

### A.3.4 Assembler directives for CAP and MIF08

Both the RISC-V-specific and GNU `.-`prefixed options.

The following table lists assembler directives:

| Directive           | Arguments | Description                                            |
|---------------------|-----------|--------------------------------------------------------|
| <code>.align</code> | integer   | align to power of 2 (alias for <code>.p2align</code> ) |



| Directive | Arguments                    | Description                                                   |
|-----------|------------------------------|---------------------------------------------------------------|
| .file     | “filename”                   | emit filename FILE LOCAL symbol table                         |
| .globl    | symbol_name                  | emit symbol_name to symbol table (scope GLOBAL)               |
| .local    | symbol_name                  | emit symbol_name to symbol table (scope LOCAL)                |
| .section  | [{.text,.data,.rodata,.bss}] | emit section (if not present, default .text) and make current |
| .size     | symbol, symbol               | accepted for source compatibility                             |
| .text     |                              | emit .text section (if not present) and make current          |
| .data     |                              | emit .data section (if not present) and make current          |
| .rodata   |                              | emit .rodata section (if not present) and make current        |
| .string   | “string”                     | emit string                                                   |
| .equ      | name, value                  | constant definition                                           |
| .word     | expression [, expression]*   | 32-bit comma separated words                                  |
| .balign   | b,[pad_val=0]                | byte align                                                    |
| .zero     | integer                      | zero bytes                                                    |

### A.3.5 Assembler Relocation Functions

The following table lists assembler relocation expansions:

| Assembler Notation | Description        | Instruction / Macro |
|--------------------|--------------------|---------------------|
| %hi(symbol)        | Absolute (HI20)    | lui                 |
| %lo(symbol)        | Absolute (LO12)    | load, store, add    |
| %pcrel_hi(symbol)  | PC-relative (HI20) | auipc               |
| %pcrel_lo(label)   | PC-relative (LO12) | load, store, add    |

### A.3.6 Instruction encoding

**Credit** This is a subset of the RISC-V greencard, by James Izhu, licence CC by SA, <https://github.com/jameslzhu/riscv-card>

### Core Instruction Formats

|                       |    |    |    |     |    |     |    |        |    |             |   |        |   |        |
|-----------------------|----|----|----|-----|----|-----|----|--------|----|-------------|---|--------|---|--------|
| 31                    | 27 | 26 | 25 | 24  | 20 | 19  | 15 | 14     | 12 | 11          | 7 | 6      | 0 |        |
| funct7                |    |    |    | rs2 |    | rs1 |    | funct3 |    | rd          |   | opcode |   | R-type |
| imm[11:0]             |    |    |    |     |    | rs1 |    | funct3 |    | rd          |   | opcode |   | I-type |
| imm[11:5]             |    |    |    | rs2 |    | rs1 |    | funct3 |    | imm[4:0]    |   | opcode |   | S-type |
| imm[12:10:5]          |    |    |    | rs2 |    | rs1 |    | funct3 |    | imm[4:1 11] |   | opcode |   | B-type |
| imm[31:12]            |    |    |    |     |    |     |    |        |    | rd          |   | opcode |   | U-type |
| imm[20 10:1 11 19:12] |    |    |    |     |    |     |    |        |    | rd          |   | opcode |   | J-type |

“imm[x:y]” means “bits x to y from binary representation of imm”. “imm[y|x]” means “bits y, then x of imm”.

**RV32I Base Integer Instructions - CAP subset**

| Inst  | Name                | FMT | Opcode  | funct3 | funct7 | Description (C)              | Note                         |
|-------|---------------------|-----|---------|--------|--------|------------------------------|------------------------------|
| add   | ADD                 | R   | 0110011 | 0x0    | 0x00   | rd = rs1 + rs2               | zero-extends                 |
| sub   | SUB                 | R   | 0110011 | 0x0    | 0x20   | rd = rs1 - rs2               |                              |
| xor   | XOR                 | R   | 0110011 | 0x4    | 0x00   | rd = rs1 ^ rs2               |                              |
| or    | OR                  | R   | 0110011 | 0x6    | 0x00   | rd = rs1   rs2               |                              |
| and   | AND                 | R   | 0110011 | 0x7    | 0x00   | rd = rs1 & rs2               |                              |
| slt   | Set Less Than       | R   | 0110011 | 0x2    | 0x00   | rd = (rs1 < rs2)?1:0         |                              |
| sltu  | Set Less Than (U)   | R   | 0110011 | 0x3    | 0x00   | rd = (rs1 < rs2)?1:0         |                              |
| addi  | ADD Immediate       | I   | 0010011 | 0x0    |        | rd = rs1 + imm               | zero-extends                 |
| xori  | XOR Immediate       | I   | 0010011 | 0x4    |        | rd = rs1 ^ imm               |                              |
| ori   | OR Immediate        | I   | 0010011 | 0x6    |        | rd = rs1   imm               |                              |
| andi  | AND Immediate       | I   | 0010011 | 0x7    |        | rd = rs1 & imm               |                              |
| lb    | Load Byte           | I   | 0000011 | 0x0    |        | rd = M[rs1+imm][0:7]         |                              |
| lw    | Load Word           | I   | 0000011 | 0x2    |        | rd = M[rs1+imm][0:31]        |                              |
| lbu   | Load Byte (U)       | I   | 0000011 | 0x4    |        | rd = M[rs1+imm][0:7]         |                              |
| sb    | Store Byte          | S   | 0100011 | 0x0    |        | M[rs1+imm][0:7] = rs2[0:7]   |                              |
| sw    | Store Word          | S   | 0100011 | 0x2    |        | M[rs1+imm][0:31] = rs2[0:31] |                              |
| beq   | Branch ==           | B   | 1100011 | 0x0    |        | if(rs1 == rs2) PC += imm     | zero-extends<br>zero-extends |
| bne   | Branch !=           | B   | 1100011 | 0x1    |        | if(rs1 != rs2) PC += imm     |                              |
| blt   | Branch <            | B   | 1100011 | 0x4    |        | if(rs1 < rs2) PC += imm      |                              |
| bge   | Branch ≥            | B   | 1100011 | 0x5    |        | if(rs1 ≥ rs2) PC += imm      |                              |
| bltu  | Branch < (U)        | B   | 1100011 | 0x6    |        | if(rs1 < rs2) PC += imm      |                              |
| bgeu  | Branch ≥ (U)        | B   | 1100011 | 0x7    |        | if(rs1 ≥ rs2) PC += imm      |                              |
| jal   | Jump And Link       | J   | 1101111 |        |        | rd = PC+4; PC += imm         |                              |
| jalr  | Jump And Link Reg   | I   | 1100111 | 0x0    |        | rd = PC+4; PC = rs1 + imm    |                              |
| lui   | Load Upper Imm      | U   | 0110111 |        |        | rd = imm << 12               |                              |
| auipc | Add Upper Imm to PC | U   | 0010111 |        |        | rd = PC + (imm << 12)        |                              |

## Pseudo Instructions

| Pseudoinstruction            | Base Instruction(s)                                           | Meaning                         |
|------------------------------|---------------------------------------------------------------|---------------------------------|
| la rd, symbol                | auipc rd, symbol[31:12]<br>addi rd, rd, symbol[11:0]          | Load address                    |
| {lb lh lw ld} rd, symbol     | auipc rd, symbol[31:12]<br>{lb lh lw ld} rd, symbol[11:0](rd) | Load global                     |
| {sb sh sw sd} rd, symbol, rt | auipc rt, symbol[31:12]<br>s{b h w d} rd, symbol[11:0](rt)    | Store global                    |
| {flw fld} rd, symbol, rt     | auipc rt, symbol[31:12]<br>fl{w d} rd, symbol[11:0](rt)       | Floating-point load global      |
| {fsw fsd} rd, symbol, rt     | auipc rt, symbol[31:12]<br>fs{w d} rd, symbol[11:0](rt)       | Floating-point store global     |
| nop                          | addi x0, x0, 0                                                | No operation                    |
| li rd, immediate             | <i>Myriad sequences</i>                                       | Load immediate                  |
| mv rd, rs                    | addi rd, rs, 0                                                | Copy register                   |
| not rd, rs                   | xori rd, rs, -1                                               | One's complement                |
| neg rd, rs                   | sub rd, x0, rs                                                | Two's complement                |
| negw rd, rs                  | subw rd, x0, rs                                               | Two's complement word           |
| sext.w rd, rs                | addiw rd, rs, 0                                               | Sign extend word                |
| seqz rd, rs                  | sltiu rd, rs, 1                                               | Set if = zero                   |
| snez rd, rs                  | sltu rd, x0, rs                                               | Set if ≠ zero                   |
| sltz rd, rs                  | slt rd, rs, x0                                                | Set if < zero                   |
| sgtz rd, rs                  | slt rd, x0, rs                                                | Set if > zero                   |
| fmv.s rd, rs                 | fsgnj.s rd, rs, rs                                            | Copy single-precision register  |
| fabs.s rd, rs                | fsgnjx.s rd, rs, rs                                           | Single-precision absolute value |
| fneg.s rd, rs                | fsgnjn.s rd, rs, rs                                           | Single-precision negate         |
| fmv.d rd, rs                 | fsgnj.d rd, rs, rs                                            | Copy double-precision register  |
| fabs.d rd, rs                | fsgnjx.d rd, rs, rs                                           | Double-precision absolute value |
| fneg.d rd, rs                | fsgnjd.d rd, rs, rs                                           | Double-precision negate         |
| beqz rs, offset              | beq rs, x0, offset                                            | Branch if = zero                |
| bnez rs, offset              | bne rs, x0, offset                                            | Branch if ≠ zero                |
| blez rs, offset              | bge x0, rs, offset                                            | Branch if ≤ zero                |
| bgez rs, offset              | bge rs, x0, offset                                            | Branch if ≥ zero                |
| bltz rs, offset              | blt rs, x0, offset                                            | Branch if < zero                |
| bgtz rs, offset              | blt x0, rs, offset                                            | Branch if > zero                |
| bgt rs, rt, offset           | blt rt, rs, offset                                            | Branch if >                     |
| ble rs, rt, offset           | bge rt, rs, offset                                            | Branch if ≤                     |
| bgtu rs, rt, offset          | bltu rt, rs, offset                                           | Branch if >, unsigned           |
| bleu rs, rt, offset          | bgeu rt, rs, offset                                           | Branch if ≤, unsigned           |
| j offset                     | jal x0, offset                                                | Jump                            |
| jal offset                   | jal x1, offset                                                | Jump and link                   |
| jr rs                        | jalr x0, rs, 0                                                | Jump register                   |
| jalr rs                      | jalr x1, rs, 0                                                | Jump and link register          |
| ret                          | jalr x0, x1, 0                                                | Return from subroutine          |
| call offset                  | auipc x1, offset[31:12]<br>jalr x1, x1, offset[11:0]          | Call far-away subroutine        |
| tail offset                  | auipc x6, offset[31:12]<br>jalr x0, x6, offset[11:0]          | Tail call far-away subroutine   |
| fence                        | fence iorw, iorw                                              | Fence on all memory and I/O     |

## RV32M Multiply Extension (basic instructions)

| Inst | Name      | FMT | Opcode  | funct3 | funct7 | Description (C)        |
|------|-----------|-----|---------|--------|--------|------------------------|
| mul  | MUL       | R   | 0110011 | 0x0    | 0x01   | rd = (rs1 * rs2)[31:0] |
| div  | DIV       | R   | 0110011 | 0x4    | 0x01   | rd = rs1 / rs2         |
| rem  | Remainder | R   | 0110011 | 0x6    | 0x01   | rd = rs1 % rs2         |