RISC-V Assembly Programmer's Manual

Authors: several contributors

Version v0.0.0, 2025-02-05: This document is in Development state. Change should be expected.

Table of Contents

Preamble

This document is in the Development state

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

List of tables

Copyright and License Information

Scope

This document aims to provide guidance to assembly programmers targeting the standard RISC-V assembly language, which common open-source assemblers like GNU as and LLVM's assembler support. Other assemblers might not support the same directives or pseudoinstructions; their dialects are outside the scope of this document.

Contributors

See github.com/riscv-non-isa/riscv-asm-manual/graphs/contributors.

Chapter 1. Command-Line Arguments

I think it's probably better to beef up the binutils documentation rather than duplicating it here.

Chapter 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).

2.1. 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. As we will see later, this allows various tricks and simplifications.

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.

Table 1. Registers of the RV32I. Based on RISC-V documentation and Patterson and Waterman "The RISC-V Reader" (2017)

Register	ABI	Use by convention	Preserved?
x0	zero	hardwired to 0, ignores writes	*n/a*
x1	ra	return address for jumps	no
x2	sp	stack pointer	yes
x3	gp	global pointer	*n/a*
x4	tp	thread pointer	*n/a*
x5	t0	temporary register 0	no
x6	t1	temporary register 1	no
x7	t2	temporary register 2	no
x8	s0 *or* fp	saved register 0 *or* frame pointer	yes
x9	s1	saved register 1	yes
x10	a0	return value *or* function argument 0	no
x11	a1	return value *or* function argument 1	no
x12	a2	function argument 2	no
x13	a3	function argument 3	no
x14	a4	function argument 4	no
x15	a5	function argument 5	no
x16	a6	function argument 6	no
x17	a7	function argument 7	no

x18	s2	saved register 2	yes
x19	s3	saved register 3	yes
x20	s4	saved register 4	yes
x21	s5	saved register 5	yes
x22	s6	saved register 6	yes
x23	s7	saved register 7	yes
x24	s8	saved register 8	yes
x25	s9	saved register 9	yes
x26	s10	saved register 10	yes
x27	s11	saved register 11	yes
x28	t3	temporary register 3	no
x29	t4	temporary register 4	no
x30	t5	temporary register 5	no
x31	t6	temporary register 6	no
pc	*(none)*	program counter	*n/a*

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.

2.2. Control registers

(TBA)

2.3. Floating Point registers (RV32F)

(TBA)

2.4. Vector registers (RV32V)

(TBA)

Chapter 3. Addressing

Addressing formats like %pcrel_lo(). We can just link to the RISC-V PS ABI document to describe what the relocations actually do.

Chapter 4. Instruction Set

Official Specifications webpage:

- riscv.org/specifications/

Latest Specifications draft repository:

- github.com/riscv/riscv-isa-manual

4.1. Instructions

Chapter 5. RISC-V ISA Specifications:

- riscv.org/specifications/

5.1. Instruction Aliases

ALIAS line from opcodes/riscv-opc.c

To better diagnose situations where the program flow reaches an unexpected location, you might want to emit there an instruction that's known to trap. You can use an `UNIMP` pseudoinstruction, which should trap in nearly all systems. The **de facto** standard implementation of this instruction is:

- `C.UNIMP`: `0000`. The all-zeroes pattern is not a valid instruction. Any system which traps on invalid instructions will thus trap on this `UNIMP` instruction form. Despite not being a valid instruction, it still fits the 16-bit (compressed) instruction format, and so `0000 0000` is interpreted as being two 16-bit `UNIMP` instructions.
- `UNIMP` : `C0001073`. This is an alias for `CSRRW x0, cycle, x0`. Since `cycle` is a read-only CSR, then (whether this CSR exists or not) an attempt to write into it will generate an illegal instruction exception. This 32-bit form of `UNIMP` is emitted when targeting a system without the C extension, or when the `.option norvc` directive is used.

5.2. Pseudo Ops

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

The following table lists assembler directives:

Table 2. Assembler Directives

Directive	Arguments	Description
.align	integer	align to power of 2 (alias for `.p2align` which is preferred - see .align
.p2align	p2,[pad_val=0],max	align to power of 2
.balign	b,[pad_val=0]	byte align
.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)
.comm	symbol_name,size,align	emit common object to .bss section
.common	symbol_name,size,align	emit common object to .bss section
.ident	"string"	accepted for source compatibility

.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
.bss		emit .bss section (if not present) and make current
.string	"string"	emit string
.asciz	"string"	emit string (alias for .string)
.equ	name, value	constant definition
.macro	name arg1 [, argn]	begin macro definition \argname to substitute
.endm		end macro definition
.type	symbol, @function	accepted for source compatibility
.option	{arch,rvc,norvc,pic,nopic,relax,norelax,push,pop}	RISC-V options. Refer to .option for a more detailed description.
.byte	expression [, expression]*	8-bit comma separated words
.2byte	expression [, expression]*	16-bit comma separated words
.half	expression [, expression]*	16-bit comma separated words
.short	expression [, expression]*	16-bit comma separated words
.4byte	expression [, expression]*	32-bit comma separated words
.word	expression [, expression]*	32-bit comma separated words
.long	expression [, expression]*	32-bit comma separated words
.8byte	expression [, expression]*	64-bit comma separated words
.dword	expression [, expression]*	64-bit comma separated words
.quad	expression [, expression]*	64-bit comma separated words
.float	expression [, expression]*	32-bit floating point values, see Floating-point literals for the value format.
.double	expression [, expression]*	64-bit floating point values, see Floating-point literals for the value format.

.quad	expression [, expression]*	128-bit floating point values, see Floating-point literals for the value format.
.dtprelword	expression [, expression]*	32-bit thread local word
.dtpreldword	expression [, expression]*	64-bit thread local word
.sleb128	expression	signed little endian base 128, DWARF
.uleb128	expression	unsigned little endian base 128, DWARF
.zero	integer	zero bytes
.variant_cc	symbol_name	annotate the symbol with variant calling convention
.attribute	name, value	RISC-V object attributes, more detailed description see .attribute.
.insn	see description	emit a custom instruction encoding, see .insn

Chapter 6. `.align`

The `.align` directive for RISC-V is an alias to `.p2align`, which aligns to a power of two, so `.align 2` means align to 4 bytes. Because the definition of the `.align` directive varies by architecture, it is recommended to use the unambiguous `.p2align` or `.balign` directives instead.

Chapter 7. `.attribute`

The `.attribute` directive is used to record information about an object file/binary that a linker or runtime loader needs to check for compatibility.

For more information like attribute name, number, value type and description, please refer to attribute section in RISC-V psABI.

`.attribute` take two arguments. The first argument of `.attribute` is the symbolic name of attribute or the attribute number, the prefix `Tag_RISCV_` can be omitted, the second argument can be string or number.

Syntax for `.attribute`:

```
.attribute <NAME_OR_NUMBER>, <ATTRIBUTE_VALUE>

NAME_OR_NUMBER := <attribute-name>
                | [1-9][0-9]*

ATTRIBUTE_VALUE := <string>
                 | <number>
```

Chapter 8. `.option`

8.1. `rvc`/`norvc`

This option will be deprecated soon after `.option arch` has been widely implemented on main stream open source toolchains.

Enable/disable the C-extension for the following code region. This option is equivalent to `.option arch, +c`/`.option arch, -c`, but widely supported by older toolchain versions.

Alternative style:

```
.option push
.option arch, +c   # Alternative of .option rvc
.option pop

.option push
.option arch, -c   # Alternative of .option norvc
.option pop
```

`.option rvc` might set the ELF flag `EF_RISCV_RVC` in some toolchains. That might cause the linker to compress instructions in code regions where that was not intended.

There is a difference between `.option rvc`/`.option norvc` and `.option arch, +c`/`.option arch, -c`. The latter won't set EF_RISCV_RVC in the ELF flags.

8.2. `arch`

Enable and/or disable specific ISA extensions for the following code regions, but without changing the arch attribute and `EF_RISCV_RVC` in the ELF flags, that means it will not raise the minimal execution environment requirement, so the user should take care to the execution of the code regions around `.option push`/`.option arch`/`.option pop`.

Syntax for `.option arch`:

```
.option arch, <EXTENSIONS-OR-FULLARCH>

EXTENSIONS-OR-FULLARCH := <EXTENSIONS>
                        | <FULLARCHSTR>

EXTENSIONS             := <EXTENSION> ',' <EXTENSIONS>
                        | <EXTENSION>

FULLARCHSTR            := <full-arch-string>
```

```
EXTENSION              := <OP> <EXTENSION-NAME> <VERSION>

OP                     := '+'
                        | '-'

VERSION                := [0-9]+ 'p' [0-9]+
                        | [1-9][0-9]*
                        |

EXTENSION-NAME         := Naming rule is defined in RISC-V ISA manual
```

- Extension version can be omitted, the assembler will use the built-in default version for that extension.
- `OP` can be enable (`+`) or disable (`-`).
- Format of `<full-arch-string>` is the same as `-march` option.

Example:

```
.attribute arch, rv64imafdc
# You can only use instructions from the i, m, a, f, d and c extensions.
memcpy_general:
    add     a5,a1,a2
    beq     a1,a5,.L2
    add     a2,a0,a2
    mv      a5,a0
.L3:
    addi    a1,a1,1
    addi    a5,a5,1
    lbu     a4,-1(a1)
    sb      a4,-1(a5)
    bne     a5,a2,.L3
.L2:
    ret

.option push     # Push current options to the stack.
.option arch, +v # Enable vector extension, we can use any instruction in imafdcv
extension.
memcpy_vec:
    mv a3, a0
.Lloop:
    vsetvli t0, a2, e8, m8, ta, ma
    vle8.v v0, (a1)
    add a1, a1, t0
    sub a2, a2, t0
    vse8.v v0, (a3)
    add a3, a3, t0
    bnez a2, .Lloop
    ret
.option pop   # Pop current option from the stack, restore the enabled ISA extension
```

```
status to imafdc.

.option push     # Push current option to the stack.
.option arch, -c # Disable compressed extension, we can't use any instruction in
extension.
memcpy_norvc:
    add     a5,a1,a2
    beq     a1,a5,.L2
    add     a2,a0,a2
    mv      a5,a0
.L3:
    addi    a1,a1,1
    addi    a5,a5,1
    lbu     a4,-1(a1)
    sb      a4,-1(a5)
    bne     a5,a2,.L3
.L2:
    ret
.option pop   # Pop current option from the stack, restore the enabled ISA extension
status to imafdc.

.option push  # Push current option to the stack.
.option arch, rv64imc # Set arch to rv64imc.
    nop
.option pop   # Pop current option from the stack, restore the enabled ISA extension
status to imafdc.
```

A typical use case is with `ifunc`, e.g. the C library is built with `rv64gc`, but a few functions like memcpy provide two versions, one built with `rv64gc` and one built with `rv64gcv`, and then select between them by ifunc mechanism at run-time. However, we don't want to change the minimal execution environment requirement to `rv64gcv`, since the `rv64gcv` version will be invoked only if the execution environment supports the vector extension, so the minimal execution environment requirement still is `rv64gc`.

`.option arch, +` will also enable all required extensions, for example, `rv32i` + `.option arch, +v` will also enable `f`, `d`, `zve32x`, `zve32f`, `zve64x`, `zve64f`, `zve64d`, `zvl32b`, `zvl64b` and `zvl128b` extensions.

We recommend `.option arch, +` and `.option arch, -` are used with `.option push` /`.option pop` instead of a `.option arch, +` / `.option arch, -` pair, because `.option arch, +` will enable all required extensions, but `.option arch, -` only disables the specific extension, so the result might be unexpected, for example: `rv32i` + `.option arch, +v` + `.option arch, -v` will result `rv32ifd_zve32x_zve32f_zve64x_zve64f_zve64d_zvl32b_zvl64b_zvl128b` not `rv32i`. Another example is `.option arch, rv64ifd` + `.option arch, -f`, which results in `rv64ifd`, because `f` will be added back when adding the implied extensions of `d`.

`.option arch, +<ext>, -<ext>` is accepted and will result in enabling the extensions that depend on `ext`, e.g. `rv32i` + `.option arch, +v, -v` will result `rv32ifd_zve32x_zve32f_zve64x_zve64f_zve64d_zvl32b_zvl64b_zvl128b`.

8.3. `pic`/`nopic`

Set the code model to PIC (position independent code) or non-PIC. This will affect the expansion of the `la` pseudoinstruction, refer to listing of standard RISC-V pseudoinstructions.

8.4. `relax`/`norelax`

Enable/disable linker relaxation for the following code region.

A code region followed by `.option relax` will emit `R_RISCV_RELAX`/`R_RISCV_ALIGN` even if the linker does not support relaxation. The suggested usage is using `.option norelax` with `.option push`/`.option pop` if linker relaxation should be disabled for a code region.

Recommended way to disable linker relaxation of specific code region is use `.option push`, `.option norelax` and `.option pop`, that prevent enabled linker relaxation accidentally if user already disable linker relaxation.

8.5. `push`/`pop`

Push/pop current options to/from the options stack.

Chapter 9. `.insn`

Emit an arbitrary instruction. This is useful for custom instructions or for very new instructions which an assembler may not support.

There are three overloads:

- `.insn <value>` - emit a raw instruction with the given value
- `.insn <insn_length>, <value>` - the same, but also verify that the instruction length has the given value in bytes
- `.insn <type> <fields>`

<type> is the instruction type (e.g. r, i, s, cj, ...). These types are specified in the RISC-V ISA specification.

<fields> is a comma-separated list of the instruction fields. The order of the fields is achieved by grouping them and listing them from LSB to MSB. The groups are:

- opcode fields
- function fields
- register fields
- immediates and symbols

E.g. an instruction with the fields (sorted from LSB to MSB):

```
opcode7, rd, func3, rs1, rs2, func7
```

Gets listed as follows:

```
opcode7, func3, func7, rd, rs1, rs2
```

For more examples, refer to the Binutils documentation.

Chapter 10. Assembler Relocation Functions

The following table lists assembler relocation expansions:

Table 3. Assembler Relocation Functions

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
%tprel_hi(symbol)	TLS LE "Local Exec"	lui
%tprel_lo(symbol)	TLS LE "Local Exec"	load, store, add
%tprel_add(symbol)	TLS LE "Local Exec"	add
%tls_ie_pcrel_hi(symbol) *	TLS IE "Initial Exec" (HI20)	auipc
%tls_gd_pcrel_hi(symbol) *	TLS GD "Global Dynamic" (HI20)	auipc
%got_pcrel_hi(symbol) *	GOT PC-relative (HI20)	auipc

* These reuse %pcrel_lo(label) for their lower half

Chapter 11. 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
```

Numeric labels are used for local references. References to local labels are suffixed with 'f' for a forward reference or 'b' for a backwards reference.

```
1:
        j 1b
```

Chapter 12. Absolute addressing

The following example shows how to load an absolute address:

```
  lui a0, %hi(msg + 1)
  addi  a0, a0, %lo(msg + 1)
```

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

```
0000000000000000 <.text>:
   0: 00000537            lui a0,0x0
      0: R_RISCV_HI20 msg+0x1
   4: 00150513            addi  a0,a0,1 # 0x1
      4: R_RISCV_LO12_I msg+0x1
```

Chapter 13. Relative addressing

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

```
1:
  auipc a0, %pcrel_hi(msg + 1)
  addi  a0, a0, %pcrel_lo(1b)
```

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

```
0000000000000000 <.text>:
   0: 00000517            auipc a0,0x0
      0: R_RISCV_PCREL_HI20 msg+0x1
   4: 00050513            mv  a0,a0
      4: R_RISCV_PCREL_LO12_I .L1
```

Chapter 14. GOT-indirect addressing

The following example shows how to load an address from the GOT:

```
1:
  auipc a0, %got_pcrel_hi(msg + 1)
  ld  a0, %pcrel_lo(1b)(a0)
```

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

```
0000000000000000 <.text>:
   0: 00000517            auipc a0,0x0
      0: R_RISCV_GOT_HI20 msg+0x1
   4: 00050513            mv  a0,a0
      4: R_RISCV_PCREL_LO12_I .L1
```

Chapter 15. Load Immediate

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

```
  .equ  CONSTANT, 0xdeadbeef

  li  a0, CONSTANT
```

Which, for RV32I, generates the following assembler output, as seen by `objdump`:

```
00000000 <.text>:
   0: deadc537            lui a0,0xdeadc
   4: eef50513            addi  a0,a0,-273 # deadbeef <CONSTANT+0x0>
```

Chapter 16. Load Upper Immediate's Immediate

The immediate argument to `lui` is an integer in the interval [0x0, 0xfffff]. Its compressed form, `c.lui`, accepts only those in the subintervals [0x1, 0x1f] and [0xfffe0, 0xfffff].

Chapter 17. Signed Immediates for I- and S-Type Instructions

All I- and S-type instructions with 12-bit signed immediates --- e.g., `addi` but not `slli` --- accept their immediate argument as an integer in the interval [-2048, 2047]. Integers in the subinterval [-2048, -1] can also be passed by their (unsigned) associates in the interval [0xfffff800, 0xffffffff] on RV32I, and in [0xfffffffffffff800, 0xffffffffffffffff] on both RV32I and RV64I.

Chapter 18. Floating-point literals

The assembler supports the same floating-point literal formats as those defined in the C and C++ standards (i.e., decimal floating-point literals with decimal exponents as well as hexadecimal floating-point literals with binary exponents).

Here are some examples:

- 3.14159
- 0.271828e1
- 0x0.3p-4

The detailed format of the floating point immediate value can be referenced on this page.

Chapter 19. Load Floating-point Immediate

The `Zfa` extension introduces `fli.{h|s|d|q}` instructions for loading a specific set of floating-point immediates, supported values can be found in the RISC-V ISA specification but are also listed below.

The `fli` instruction is used to load a floating point immediate into a floating register, the accepted immediate is defined in Floating-point literals and the reference table can be found in FLI operands reference table.

```
  fli.s fa0, 0x1p-15
  fli.s fa1, 0.00390625
  fli.s fa2, 6.25e-02
```

The tool should reject any value that does not exactly match a floating-point immediate operand for the 'fli' instruction.

RISC-V does not offer a generic pseudoinstruction to load an arbitrary floating point immediate value. Instead, a programmer can use the `.float`/`.double` directive to declare a floating point immediate value in the source code, and then load it into a floating point register using the load global pseudoinstruction (`fl{h|w|d|q}`).

```
  .data
.VAL:
  .float .0x1p+17
  .text
  flw fa0, .VAL, t0
```

Table 4. FLI operands reference table

Value	Example legal input values
-1.0	-0x1p+0, -1.0, -1.0e+0
Minimum positive normal	min
1.0 x 2 ^ -16	0x1p-16, 0.0000152587890625, 1.52587890625e-05
1.0 x 2 ^ -15	0x1p-15, 0.000030517578125, 3.0517578125e-05
1.0 x 2 ^ -8	0x1p-8, 0.00390625, 3.90625e-03
1.0 x 2 ^ -7	0x1p-7, 0.0078125, 7.8125e-03
0.0625 (2 ^ -4)	0x1p-4, 0.0625, 6.25e-02
0.125 (2 ^ -3)	0x1p-3, 0.125, 1.25e-01
0.25	0x1p-2, 0.25, 2.5e-01
0.3125	0x1.4p-2, 0.3125, 3.125e-01
0.375	0x1.8p-2, 0.375, 3.75e-01
0.4375	0x1.cp-2, 0.4375, 4.375e-01

0.5	0x1p-1, 0.5, 5.0e-01
0.625	0x1.4p-1, 0.625, 6.25e-01
0.75	0x1.8p-1, 0.75, 7.5e-01
0.875	0x1.cp-1, 0.875, 8.75e-01
1.0	0x1p+0, 1.0, 1.0e+00
1.25	0x1.4p+0, 1.25, 1.25e+00
1.5	0x1.8p+0, 1.5, 1.5e+00
1.75	0x1.cp+0, 1.75, 1.75e+00
2.0	0x1p+1, 2.0, 2.0e+00
2.5	0x1.4p+1, 2.5, 2.5e+00
3	0x1.8p+1, 3.0, 3.0e+00
4	0x1p+2, 4.0, 4.0e+00
8	0x1p+3, 8.0, 8.0e+00
16	0x1p+4, 16.0, 1.6e+01
128 (2 ^ 7)	0x1p+7, 128.0, 1.28e+02
256 (2 ^ 8)	0x1p+8, 256.0, 2.56e+02
2 ^ 15	0x1p+15, 32768.0, 3.2768e+04
2 ^ 16	0x1p+16, 65536.0, 6.5536e+04
Positive infinity	inf
Canonical NaN	nan

A value can be expressed in various forms within the same format. For example, 6.5536e+04 can be alternatively written as 6553.6e+01 or 65.536e+03. The table provides one possible representation, but any equivalent exact value may be used.

Chapter 20. Load Address

The following example shows the `la` pseudoinstruction which is used to load symbol addresses using the correct sequence based on whether the code is being assembled as PIC:

```
la  a0, msg + 1
```

For non-PIC this is an alias for the `lla` pseudoinstruction documented below.

For PIC this is an alias for the `lga` pseudoinstruction documented below.

The `la` pseudoinstruction is the preferred way for getting the address of variables in assembly unless explicit control over PC-relative or GOT-indirect addressing is required.

Chapter 21. Load Local Address

The following example shows the `lla` pseudoinstruction which is used to load local symbol addresses:

```
  lla a0, msg + 1
```

This generates the following instructions and relocations as seen by `objdump`:

```
0000000000000000 <.text>:
   0: 00000517            auipc a0,0x0
      0: R_RISCV_PCREL_HI20 msg+0x1
   4: 00050513            mv  a0,a0
      4: R_RISCV_PCREL_LO12_I .L0
```

Chapter 22. Load Global Address

The following example shows the `lga` pseudoinstruction which is used to load global symbol addresses:

```
  lga a0, msg + 1
```

This generates the following instructions and relocations as seen by `objdump` (for RV64; RV32 will use `lw` instead of `ld`):

```
0000000000000000 <.text>:
   0: 00000517            auipc a0,0x0
      0: R_RISCV_GOT_HI20 msg+0x1
   4: 00053503            ld  a0,0(a0) # 0 <.text>
      4: R_RISCV_PCREL_LO12_I .L0
```

Chapter 23. Load and Store Global

The following pseudoinstructions are available to load from and store to global objects:

- `l{b|h|w|d} <rd>, <symbol>`: load byte, half word, word or double word from global [1]
- `l{bu|hu|wu} <rd>, <symbol>`: load unsigned byte, half word, or word from global [1]
- `s{b|h|w|d} <rd>, <symbol>, <rt>`: store byte, half word, word or double word to global [2]
- `fl{h|w|d|q} <rd>, <symbol>, <rt>`: load half, float, double or quad precision from global [2]
- `fs{h|w|d|q} <rd>, <symbol>, <rt>`: store half, float, double or quad precision to global [2]

The following example shows how these pseudoinstructions are used:

```
  lw  a0, var1
  fld fa0, var2, t0
  sw  a0, var3, t0
  fsd fa0, var4, t0
```

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

```
0000000000000000 <.text>:
   0: 00000517            auipc a0,0x0
      0: R_RISCV_PCREL_HI20 var1
   4: 00052503            lw  a0,0(a0) # 0 <.text>
      4: R_RISCV_PCREL_LO12_I .L0
   8: 00000297            auipc t0,0x0
      8: R_RISCV_PCREL_HI20 var2
   c: 0002b507            fld fa0,0(t0) # 8 <.text+0x8>
      c: R_RISCV_PCREL_LO12_I .L0
  10: 00000297            auipc t0,0x0
      10: R_RISCV_PCREL_HI20  var3
  14: 00a2a023            sw  a0,0(t0) # 10 <.text+0x10>
      14: R_RISCV_PCREL_LO12_S  .L0
  18: 00000297            auipc t0,0x0
      18: R_RISCV_PCREL_HI20  var4
  1c: 00a2b027            fsd fa0,0(t0) # 18 <.text+0x18>
      1c: R_RISCV_PCREL_LO12_S  .L0
```

[1] the first operand is implicitly used as a scratch register.

[2] the last operand specifies the scratch register to be used.

Chapter 24. Constants

The following example shows loading a constant using the `%hi` and `%lo` assembler functions.

```
  .equ  UART_BASE, 0x40003080

  lui a0, %hi(UART_BASE)
  addi  a0, a0, %lo(UART_BASE)
```

Which generates the following assembler output as seen by `objdump`:

```
0000000000000000 <.text>:
   0: 40003537            lui a0,0x40003
   4: 08050513            addi  a0,a0,128 # 40003080 <UART_BASE>
```

Chapter 25. Far Branches

The assembler will convert conditional branches into far branches when necessary, via inserting a short branch with inverted conditions past an unconditional jump. For example

```
target:
  bne a0, a1, target
.rep 1024
  nop
.endr
  bne a0, a1, target
```

ends up as

```
       0: 00b51063            bne a0,a1,0 <target>
...
    1004: 00b50463            beq a0,a1,100c <target+0x100c>
    1008: ff9fe06f            j 0 <target>
```

Chapter 26. Function Calls

The following pseudoinstructions are available to call subroutines far from the current position:

- `call <symbol>`: call away subroutine [1]
- `call <rd>, <symbol>`: call away subroutine [2]
- `tail <symbol>`: tail call away subroutine[[3]
- `jump <symbol>, <rt>`: jump to away routine [4]

The following example shows how these pseudoinstructions are used:

```
  call  func1
  tail  func2
  jump  func3, t0
```

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

```
0000000000000000 <.text>:
   0: 00000097            auipc ra,0x0
      0: R_RISCV_CALL func1
   4: 000080e7            jalr  ra # 0x0
   8: 00000317            auipc t1,0x0
      8: R_RISCV_CALL func2
   c: 00030067            jr  t1 # 0x8
  10: 00000297            auipc t0,0x0
      10: R_RISCV_CALL  func3
  14: 00028067            jr  t0 # 0x10
```

[1] `ra` is implicitly used to save the return address.

[2] similar to `call <symbol>`, but `<rd>` is used to save the return address instead.

[3] If the `Zicfilp` extension is available, `t2` is implicitly used as a scratch register. Otherwise,`t1` is implicitly used as a scratch register.

[4] similar to `tail <symbol>`, but `<rt>` is used as the scratch register instead.

Chapter 27. Floating-point rounding modes

For floating-point instructions with a rounding mode field, the rounding mode can be specified by adding an additional operand. e.g. `fcvt.w.s` with round-to-zero can be written as `fcvt.w.s a0, fa0, rtz`. If unspecified, the default `dyn` rounding mode will be used.

Supported rounding modes are as follows (must be specified in lowercase):

- `rne`: round to nearest, ties to even
- `rtz`: round towards zero
- `rdn`: round down
- `rup`: round up
- `rmm`: round to nearest, ties to max magnitude
- `dyn`: dynamic rounding mode (the rounding mode specified in the `frm` field of the `fcsr` register is used)

Chapter 28. Control and Status Registers

The following code sample shows how to enable timer interrupts, set and wait for a timer interrupt to occur:

```
.equ RTC_BASE,      0x40000000
.equ TIMER_BASE,    0x40004000

# setup machine trap vector
1:      auipc   t0, %pcrel_hi(mtvec)        # load mtvec(hi)
        addi    t0, t0, %pcrel_lo(1b)       # load mtvec(lo)
        csrrw   zero, mtvec, t0

# set mstatus.MIE=1 (enable M mode interrupt)
        li      t0, 8
        csrrs   zero, mstatus, t0

# set mie.MTIE=1 (enable M mode timer interrupts)
        li      t0, 128
        csrrs   zero, mie, t0

# read from mtime
        li      a0, RTC_BASE
        ld      a1, 0(a0)

# write to mtimecmp
        li      a0, TIMER_BASE
        li      t0, 1000000000
        add     a1, a1, t0
        sd      a1, 0(a0)

# loop
loop:
        wfi
        j loop

# break on interrupt
mtvec:
        csrrc  t0, mcause, zero
        bgez t0, fail       # interrupt causes are less than zero
        slli t0, t0, 1      # shift off high bit
        srli t0, t0, 1
        li t1, 7            # check this is an m_timer interrupt
        bne t0, t1, fail
        j pass

pass:
        la a0, pass_msg
        jal puts
        j shutdown
```

```
fail:
        la a0, fail_msg
        jal puts
        j shutdown

.section .rodata

pass_msg:
        .string "PASS\n"

fail_msg:
        .string "FAIL\n"
```

Chapter 29. A listing of standard RISC-V pseudoinstructions

Table 5. Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning	Comment
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address	With `.option nopic` (Default)
la rd, symbol	auipc rd, symbol@GOT[31:12] l{w|d} rd, symbol@GOT[11:0](rd)	Load address	With `.option pic`
lla rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load local address	
lga rd, symbol	auipc rd, symbol@GOT[31:12] l{w|d} rd, symbol@GOT[11:0](rd)	Load global address	
l{b|h|w|d} rd, symbol	auipc rd, symbol[31:12] l{b|h|w|d} rd, symbol[11:0](rd)	Load global	
l{bu|hu|wu} rd, symbol	auipc rd, symbol[31:12] l{bu|hu|wu} rd, symbol[11:0](rd)	Load global, unsigned	
s{b|h|w|d} rd, symbol, rt	auipc rt, symbol[31:12] s{b|h|w|d} rd, symbol[11:0](rt)	Store global	
fl{w|d} rd, symbol, rt	auipc rt, symbol[31:12] fl{w|d} rd, symbol[11:0](rt)	Floating-point load global	
fs{w|d} rd, symbol, rt	auipc rt, symbol[31:12] fs{w|d} rd, symbol[11:0](rt)	Floating-point store global	
nop	addi x0, x0, 0	No operation	
li rd, immediate	*Myriad sequences[1]	Load immediate	
mv rd, rs	addi rd, rs, 0	Copy register	
not rd, rs	xori rd, rs, -1	Ones' complement	
neg rd, rs	sub rd, x0, rs	Two's complement	
negw rd, rs	subw rd, x0, rs	Two's complement word	
sext.b rd, rs	slli rd, rs, XLEN - 8 srai rd, rd, XLEN - 8	Sign extend byte	This is a single instruction when `Zbb` extension is available.

Pseudoinstruction	Base Instruction(s)	Meaning	Comment
sext.h rd, rs	slli rd, rs, XLEN - 16 srai rd, rd, XLEN - 16	Sign extend halfword	This is a single instruction when Zbb extension is available.
sext.w rd, rs	addiw rd, rs, 0	Sign extend word	
zext.b rd, rs	andi rd, rs, 255	Zero extend byte	
zext.h rd, rs	slli rd, rs, XLEN - 16 srli rd, rd, XLEN - 16	Zero extend halfword	This is a single instruction when Zbb extension is available.
zext.w rd, rs	slli rd, rs, XLEN - 32 srli rd, rd, XLEN - 32	Zero extend word	This is a single instruction when Zba extension is available.
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	fsgnjn.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	

Pseudoinstruction	Base Instruction(s)	Meaning	Comment
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	
vfneg.v vd, vs	vfsgnjn.vv vd, vs, vs	Floating-point vector negate	
vfabs.v vd, vs	vfsgnjx.vv vd, vs, vs	Floating-point vector absolute value	
vmclr.m vd	vmxor.mm vd, vd, vd	Vector clear mask register	
vmfge.vv vd, va, vb, vm	vmfle.vv vd, vb, va, vm	Vector Floating-point >=	
vmfgt.vv vd, va, vb, vm	vmflt.vv vd, vb, va, vm	Vector Floating-point	
vmmv.m vd, vs	vmand.mm vd, vs, vs	Vector copy mask register	
vmnot.m vd, vs	vmnand.mm vd, vs, vs	Vector invert mask bits	
vmset.m vd	vmxnor.mm vd, vd, vd	Vector set all mask bits	
vmsge.vi vd, va, i, vm	vmsgt.vi vd, va, i-1, vm	Vector >= Immediate	
vmsgeu.vi vd, va, i, vm	vmsgtu.vi vd, va, i-1, vm	Vector >= Immediate, unsigned	

Pseudoinstruction	Base Instruction(s)	Meaning	Comment
vmsge.vv vd, va, vb, vm	vmsle.vv vd, vb, va, vm	Vector >= Vector	
vmsgeu.vv vd, va, vb, vm	vmsleu.vv vd, vb, va, vm	Vector >= Vector, unsigned	
vmsgt.vv vd, va, vb, vm	vmslt.vv vd, vb, va, vm	Vector > Vector	
vmsgtu.vv vd, va, vb, vm	vmsltu.vv vd, vb, va, vm	Vector > Vector, unsigned	
vmslt.vi vd, va, i, vm	vmsle.vi vd, va, i-1, vm	Vector < immediate	
vmsltu.vi vd, va, i, vm	vmsleu.vi vd, va, i-1, vm	Vector < immediate, unsigned	
vneg.v vd,vs	vrsub.vx vd,vs,x0	Vector negate	
vnot.v vd,vs,vm	vxor.vi vd, vs, -1, vm	Vector not	
vncvt.x.x.w vd,vs,vm	vnsrl.wx vd,vs,x0,vm	Vector narrow convert element	
vwcvt.x.x.v vd,vs,vm	vwadd.vx vd,vs,x0,vm	Vector widen convert, integer-integer	
vwcvtu.x.x.v vd,vs,vm	vwaddu.vx vd,vs,x0,vm	Vector widen convert, integer-integer, unsigned	
vl1r.v v3, x0	vl1re8.v v3, x0	Equal to vl1re8.v	
vl2r.v v2,x0	vl2re8.v v2, x0	Equal to vl2re8.v	
vl4r.v v4,x0	vl4re8.v v4, x0	Equal to vl4re8.v	
vl8r.v v8,x0	vl8re8.v v8, x0	Equal to vl8re8.v	
vmsge{u}.vx vd, va, x	vmslt{u}.vx vd, va, x vmnand.mm vd, vd, vd	Vector >= scalar, unmasked	
vmsge{u}.vx vd, va, x, v0.t	vmslt{u}.vx vd, va, x, v0.t vmxor.mm vd, vd, v0	Vector >= scalar, masked	When vd≠v0
vmsge{u}.vx vd, va, x, v0.t, vt	vmslt{u}.vx vt, va, x vmandn.mm vd, vd, vt	Vector >= scalar, masked	When vd=v0
vmsge{u}.vx vd, va, x, v0.t, vt	vmslt{u}.vx vt, va, x vmandn.mm vt, v0, vt vmandn.mm vd, vd, v0 vmor.mm vd, vt, vd	Vector >= scalar, masked	For any vd
call offset	auipc x1, offset[31:12] jalr x1, x1, offset[11:0]	Call far-away subroutine	

Pseudoinstruction	Base Instruction(s)	Meaning	Comment
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	Tail call far-away subroutine Tail call far-away subroutine	It will use `x7` as scratch register when `Zicfilp` extension is available.
fence	fence iorw, iorw	Fence on all memory and I/O	
pause	fence w, 0	PAUSE hint	

[1] The compiler can generate different instruction sequences to load a specific numeric value into a register.

Chapter 30. Pseudoinstructions for accessing control and status registers

Table 6. Pseudoinstructions for acccessing control and status registers

Pseudoinstruction	Base Instruction(s)	Meaning
rdinstret[h] rd	csrrs rd, instret[h], x0	Read instructions-retired counter
rdcycle[h] rd	csrrs rd, cycle[h], x0	Read cycle counter
rdtime[h] rd	csrrs rd, time[h], x0	Read real-time clock
csrr rd, csr	csrrs rd, csr, x0	Read CSR
csrw csr, rs	csrrw x0, csr, rs	Write CSR
csrs csr, rs	csrrs x0, csr, rs	Set bits in CSR
csrc csr, rs	csrrc x0, csr, rs	Clear bits in CSR
csrwi csr, imm	csrrwi x0, csr, imm	Write CSR, immediate
csrsi csr, imm	csrrsi x0, csr, imm	Set bits in CSR, immediate
csrci csr, imm	csrrci x0, csr, imm	Clear bits in CSR, immediate
frcsr rd	csrrs rd, fcsr, x0	Read FP control/status register
fscsr rd, rs	csrrw rd, fcsr, rs	Swap FP control/status register
fscsr rs	csrrw x0, fcsr, rs	Write FP control/status register
frrm rd	csrrs rd, frm, x0	Read FP rounding mode
fsrm rd, rs	csrrw rd, frm, rs	Swap FP rounding mode
fsrm rs	csrrw x0, frm, rs	Write FP rounding mode
fsrmi rd, imm	csrrwi rd, frm, imm	Swap FP rounding mode, immediate
fsrmi imm	csrrwi x0, frm, imm	Write FP rounding mode, immediate
frflags rd	csrrs rd, fflags, x0	Read FP exception flags
fsflags rd, rs	csrrw rd, fflags, rs	Swap FP exception flags
fsflags rs	csrrw x0, fflags, rs	Write FP exception flags
fsflagsi rd, imm	csrrwi rd, fflags, imm	Swap FP exception flags, immediate
fsflagsi imm	csrrwi x0, fflags, imm	Write FP exception flags, immediate