RISC-V Assembly Programmer's Manual

Copyright and License Information

Command-Line Arguments

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

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).

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.

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

Register	ABI	Use by convention	Preserved?
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*

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.

Control registers

(TBA)

Floating Point registers (RV32F)

(TBA)

Vector registers (RV32V)

(TBA)

Addressing

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

Instruction Set

Official Specifications webpage: - https://riscv.org/specifications/

Latest Specifications draft repository: - https://github.com/riscv/riscv-isa-manual

Instructions

RISC-V User Level ISA Specification

https://riscv.org/specifications/

RISC-V Privileged ISA Specification

https://riscv.org/specifications/privileged-isa/

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` pseudo-instruction, 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.

Pseudo Ops

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

The following table lists assembler directives:

Directive	Arguments	Description
.align	integer	align to power of 2 (alias for .p2align)
.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

Directive	Arguments	Description
.type	symbol, @function	accepted for source compatibility
.option	{rvc,norvc,pic,nopic,push,pop}	RISC-V options
.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
.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
.p2align	p2,[pad_val=0],max	align to power of 2
.balign	b,[pad_val=0]	byte align
.zero	integer	zero bytes

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
%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

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
```

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
```

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
```

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
```

Load Immediate

The following example shows the `li` pseudo instruction 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>
```

Load Address

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

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

Which generates the following assembler output and relocations for non-PIC 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
```

And generates the following assembler output and relocations for PIC as seen by `objdump`:

```
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
```

Load and Store Global

The following pseudo instructions 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]
- `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 pseudo instructions 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
```

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>
```

Function Calls

The following pseudo instructions 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 pseudo instructions 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
```

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)

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"
```

A listing of standard RISC-V pseudoinstructions

Pseudoinstruction	Base Instruction(s)	Meaning

Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12]; addi rd, rd, symbol[11:0]	Load 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
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*	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	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
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

Pseudoinstruction	Base Instruction(s)	Meaning
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 x6, offset[31:12]; jalr x1, x6, offset[11:0]	Call far-away subroutine
tail offset	auipc x6, offset[31:12]; jalr x0, x6, offset[11:0]	Tail call far-away subroutine
fence	fence iorw, iorw	Fence on all memory and I/O

Pseudoinstructions for accessing 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

Pseudoinstruction	Base Instruction(s)	Meaning
fsflagsi imm	csrrwi x0, fflags, imm	Write FP exception flags, immediate

---

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. `t1` is implicitly used as a scratch register. ↩
4. similar to `tail <symbol>`, but `<rt>` is used as the scratch register instead. ↩