URL: https://rv8.io/asm.html
Start from p. 10.

rv8

RISC-V simulator for x86-64

# RISC-V Assembler Reference

This document gives an overview of RISC-V assembly language. First, an introduction to assembler and linker concepts, then sections describing assembler directives, pseudo- instructions, relocation functions, and assembler concepts such as labels, relative and absolute addressing, immediate values, constants and finally control and status registers.

The accompanying RISC-V Instruction Set Reference contains a listing of instruction in the `I` *(Base Integer Instruction Set)* and `M` *(Multiply and Divide)* extension. For detailed information on the RISC-V instruction set refer to the RISC-V ISA Specification.

## Concepts

This section briefly covers some high level concepts that are required to understand the process of assembling and linking executable code from source files.

### Assembly file

An assembly file contains assembly language directives, macros and instructions. It can be emitted by a compiler or it can be handwritten. An assembly file is the input file to the assembler. The standard extensions for assembly files are `.s` and `.S`, with the later indicating that the assembly file should be pre-processed using the C preprocessor.

## Relocatable Object file

A relocatable object file contains compiled object code and data emitted by the assembler. An object file cannot be run, rather it is used as input to the linker. The standard extension for object files is `.o`. The most common cross-platform file format for RISC-V executables is the ELF *(Electronic Linker Format)* object file format. The `objdump` utility can be used to disassemble an object file, `objcopy` can be used to copy and extract sections from ELF files and the `nm` utility can list symbols in an object file.

## ELF Header

An ELF file has an ELF header that contains *magic* to indicate the file is ELF formatted, the architecture of the binary, the endianness of the binary *(little-endian for RISC-V)*, the ELF file type *(Relocatable Object File, Executable File, Shared Library)*, the number of program headers and their offset in the file, the number of section headers and their offset in the file, fields indicating the ELF version and ABI *(Application Binary Interface)* version of the file and finally flags indicating various ABI options such as RVC compression and which floating- point ABI that the executable code in the binary conforms to.

## Program Header

Program Headers provide size and offsets of loadable segments within an executable file or shared object along with protection attributes used by the operating system *(read, write and exec)*. Program headers are not present in relocatable object files and are primarily for use by the operating system to and dynamic linker to map code and data into memory.

## Section Header

Section Headers provice size, offset, type, alignment and flags of the sections contained within the ELF file. Section headers are not required to execute a static binary but are necessary for dynamic linking as well as program linking. Various section types refer to the location of the symbol table, relocations and dynamic symbols in the ELF binary file.

### Sections

An object file is made up of multiple sections, with each section corresponding to distinct types of executable code or data. There are a variety of different section types. This list shows the four most common sections:

- `.text` is a read-only section containing executable code
- `.data` is a read-write section containing global or static variables
- `.rodata` is a read-only section containing const variables
- `.bss` is a read-write section containing uninitialised data

### Program linking

Program linking is the process of reading multiple relocatable object files, merging the sections from each of the source files, calculating the new addresses for symbols and applying relocation fixups to text or data that is pointed to in relocation entries.

### Linker Script

A linker script is a text source file that is optionally input to the linker and it contains rules for the linker to use when calculating the load address and alignment of the various sections when creating an executable output file. The standard extension for linker scripts is `.ld`.

## Assembler Directives

The assembler implements a number of directives that control the assembly of instructions into an object file. These directives give the ability to include arbitrary data in the object file, control exporting of symbols, selection of sections, alignment of data, assembly options for compression, position dependent and position independent code.

The following are assembler directives for emitting data:

| Directive | Arguments | Description |
|---|---|---|
| `.2byte` | | 16-bit comma separated words (unaligned) |
| `.4byte` | | 32-bit comma separated words (unaligned) |
| `.8byte` | | 64-bit comma separated words (unaligned) |
| `.half` | | 16-bit comma separated words (naturally aligned) |
| `.word` | | 32-bit comma separated words (naturally aligned) |
| `.dword` | | 64-bit comma separated words (naturally aligned) |
| `.byte` | | 8-bit comma separated words |
| `.dtpreldword` | | 64-bit thread local word |
| `.dtprelword` | | 32-bit thread local word |
| `.sleb128` | expression | signed little endian base 128, DWARF |
| `.uleb128` | expression | unsigned little endian base 128, DWARF |

| Directive | Arguments | Description |
|---|---|---|
| `.asciz` | "string" | emit string (alias for .string) |
| `.string` | "string" | emit string |
| `.incbin` | "filename" | emit the included file as a binary sequence of octets |
| `.zero` | integer | zero bytes |

The following are assembler directives for control of alignment:

| Directive | Arguments | Description |
|---|---|---|
| `.align` | integer | align to power of 2 (alias for .p2align) |
| `.balign` | b,[pad_val=0] | byte align |
| `.p2align` | p2,[pad_val=0],max | align to power of 2 |

The following are assembler directives for definition and exporing of symbols:

| Directive | Arguments | Description |
|---|---|---|
| `.globl` | symbol_name | emit symbol_name to symbol table (scope GLOBAL) |
| `.local` | symbol_name | emit symbol_name to symbol table (scope LOCAL) |
| `.equ` | name, value | constant definition |

The following directives are for selection of sections:

| Directive | Arguments | Description |
|---|---|---|
| | | emit .text section (if not present) and make |

| Directive | Arguments | Description |
|---|---|---|
| `.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 |
| `.comm` | symbol_name,size,align | emit common object to .bss section |
| `.common` | symbol_name,size,align | emit common object to .bss section |
| `.section` | [{.text,.data,.rodata,.bss}] | emit section (if not present, default .text) and make current |

The following directives includes options, macros and other miscellaneous functions:

| Directive | Arguments | Description |
|---|---|---|
| `.option` | {rvc,norvc,pic,nopic,push,pop} | RISC-V options |
| `.macro` | name arg1 [, argn] | begin macro definition \argname to substitute |
| `.endm` | | end macro definition |
| `.file` | “filename” | emit filename FILE LOCAL symbol table |

| Directive | Arguments | Description |
|---|---|---|
| `.ident` | “string” | accepted for source compatibility |
| `.size` | symbol, symbol | accepted for source compatibility |
| `.type` | symbol, @function | accepted for source compatibility |

## Assembler Pseudo-instructions

The assembler implements a number of convenience psuedo-instructions that are formed from instructions in the base ISA, but have implicit arguments or in some case reversed arguments, that result in distinct semantics.

The following table lists RISC-V assembler pseudo instructions:

| Pesudo-instruction | Expansion | Description |
|---|---|---|
| `nop` | `addi zero,zero,0` | No operation |
| `li rd, expression` | (several expansions) | Load immediate |
| `la rd, symbol` | (several expansions) | Load address |
| `mv rd, rs1` | `addi rd, rs, 0` | Copy register |
| `not rd, rs1` | `xori rd, rs, -1` | One’s complement |
| `neg rd, rs1` | `sub rd, x0, rs` | Two’s complement |
| `negw rd, rs1` | `subw rd, x0, rs` | Two’s complement Word |

| Instruction | | Description |
|---|---|---|
| `sext.w rd, rs1` | `addiw rd, rs, 0` | Sign-extend word |
| `seqz rd, rs1` | `sltiu rd, rs, 1` | Set if = zero |
| `snez rd, rs1` | `sltu rd, x0, rs` | Set if ≠ zero |
| `sltz rd, rs1` | `slt rd, rs, x0` | Set if < zero |
| `sgtz rd, rs1` | `slt rd, x0, rs` | Set if > zero |
| `fmv.s frd, frs1` | `fsgnj.s frd, frs, frs` | Single-precision move |
| `fabs.s frd, frs1` | `fsgnjx.s frd, frs, frs` | Single-precision absolute value |
| `fneg.s frd, frs1` | `fsgnjn.s frd, frs, frs` | Single-precision negate |
| `fmv.d frd, frs1` | `fsgnj.d frd, frs, frs` | Double-precision move |
| `fabs.d frd, frs1` | `fsgnjx.d frd, frs, frs` | Double-precision absolute value |
| `fneg.d frd, frs1` | `fsgnjn.d frd, frs, frs` | Double-precision negate |
| `beqz rs1, offset` | `beq rs, x0, offset` | Branch if = zero |

| Pseudo-instruction | Expansion | Description |
|---|---|---|
| `bnez rs1, offset` | `bne rs, x0, offset` | Branch if ≠ zero |
| `blez rs1, offset` | `bge x0, rs, offset` | Branch if ≤ zero |
| `bgez rs1, offset` | `bge rs, x0, offset` | Branch if ≥ zero |
| `bltz rs1, offset` | `blt rs, x0, offset` | Branch if < zero |
| `bgtz rs1, 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` | `bltu rt, rs, offset` | Branch if ≤, unsigned |
| `j offset` | `jal x0, offset` | Jump |
| `jr offset` | `jal x1, offset` | Jump register |

| | | |
|---|---|---|
| `ret` | `jalr x0, x1, 0` | Return from subroutine |

## Relocation Functions

The relocation function directives create synthesize operand values that are resolved at program link time and are used as immediate parameters to specific instructions. The sections on absolute and relative addressing give examples of using the relocation functions.

The following table lists assembler functions used to generate relocations:

| Assembler Notation | Description | Instructions |
|---|---|---|
| `%hi(symbol)` | Absolute (HI20) | lui |
| `%lo(symbol)` | Absolute (LO12) | loads, stores, adds |
| `%pcrel_hi(symbol)` | PC-relative (HI20) | auipc |
| `%pcrel_lo(label)` | PC-relative (LO12) | loads, stores, adds |
| `%tprel_hi(symbol)` | TLS LE (Local Exec) | auipc |
| `%tprel_lo(label)` | TLS LE (Local Exec) | loads, stores, adds |
| `%tprel_add(offset)` | TLS LE (Local Exec) | add |

## 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

Absolute addresses are used in position dependent code. An absolute address is formed using two instructions, the U-Type `lui` *(Load Upper Immediate)* instruction to load `bits[31:20]` and an I-Type or S-Type instruction such as `addi` *(add immediate)*, `lw` *(load word)* or `sw` *(store word)* that fills in the low 12 bits relative to the upper immediate.

The following example shows how to load an absolute address:

```
.section .text
.globl _start
_start:
            lui  a1,      %hi(msg)       # load msg(hi)
            addi a1, a1,  %lo(msg)       # load msg(lo)
            jalr 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_LO12_I       msg
```

## Relative Addressing

Relative addresses are used in position independent code. A PC-relative address is formed using two instructions, the U-Type `auipc` *(Add Upper Immediate Program Counter)* instruction to load `bits[31:20]` relative to the program counter of the `auipc` instruction followed by an I-Type or S-Type instruction such as `addi` *(add immediate)*, `lw` *(load word)* or `sw` *(store word)*.

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

```
.section .text
.globl _start
_start:
1:          auipc a1,     %pcrel_hi(msg) # load msg(hi)
            addi  a1, a1, %pcrel_lo(1b)  # load msg(lo)
            jalr  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 `li` *(load immediate)* instruction is an assembler pseudo instruction that is used to synthesize constants. The `li` pseudo instruction will emit a sequence starting with `lui` followed by `addi` and `slli` *(shift left logical immediate)* to construct constants by shifting and adding.

The following example shows the `li` psuedo instruction being used to load an immediate value:

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

.equ CONSTANT, 0xcafebabe

        li a0, CONSTANT
```

which generates the following assembler output as seen by objdump:

```
0000000000000000 <_start>:
   0:   00032537                lui     a0,0x32
   4:   bfb50513                addi    a0,a0,-1029
```

```
    8:   00e51513                slli    a0,a0,0xe
    c:   abe50513                addi    a0,a0,-1346
```

## Load Address

The `la` *(load address)* instruction is an assembler pseudo- instruction used to load the address of a symbol or label. The instruction can emit absolute or relative addresses depending on the `-fpic` or `-fno-pic` assembler command line options or an `.options pic` or `.options nopic` assembler directive. The pseduo-instruction emits a relocation so that the address of the symbol can be fixed up during program linking.

The following example uses the `la` psuedo instruction to load a symbol address:

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

        la a0, msg

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

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

```
0000000000000000 <_start>:
   0:   00000517                auipc   a0,0x0
                        0: R_RISCV_PCREL_HI20   msg
```

```
    4:   00850513                addi    a0,a0,8 # 8 <_sta
                         4: R_RISCV_PCREL_LO12_I .L11
```

## Constants

Constants are emitted to the symbol table of the object but they do not take any space in the code or data sections. Constants can be referenced in expressions which emit relocations.

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

```
.equ UART_BASE, 0x40003000

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

This example uses the `li` pseudo instruction to load a constant and writes a string using polled IO to a UART:

```
.equ UART_BASE,   0x40003000
.equ REG_RBR,     0
.equ REG_TBR,     0
.equ REG_IIR,     2
.equ IIR_TX_RDY, 2
.equ IIR_RX_RDY, 4

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

```
1:      auipc a0, %pcrel_hi(msg)    # load msg(hi)
        addi  a0, a0, %pcrel_lo(1b)  # load msg(lo)
2:      jal   ra, puts
3:      j     3b

puts:
        li    a2, UART_BASE
1:      lbu   a1, (a0)
        beqz  a1, 3f
2:      lbu   a3, REG_IIR(a2)
        andi  a3, a3, IIR_TX_RDY
        beqz  a3, 2b
        sb    a1, REG_TBR(a2)
        addi  a0, a0, 1
        j     1b
3:      ret

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

## Control and Status Registers

Control and status registers are typically used to update privileged processor state however there are a few non-privileged instructions that access control and status registers such as the CSR pseudo-instructions `rdcycle`, `rdtime`, `rdinstret` for access to counters and `frcsr`, `frrm`, `frflags`, `fscsr`, `fsrm`, `fsflags`, `fsrmi` and `fsflagsi` for controling round mode and accessing floating point accrued exception state.

The following instructions allow reading, writing, setting and clearing bits in CSRs *(control and status registers)*:

| CSR Operation | Description |
|---|---|
| `CSRRW rd, csr, rs1` | Control and Status Register Atomic Read and Write |
| `CSRRS rd, csr, rs1` | Control and Status Register Atomic Read and Set Bits |
| `CSRRC rd, csr, rs1` | Control and Status Register Atomic Read and Clear Bits |
| `CSRRWI rd, csr, imm5` | Control and Status Register Atomic Read and Write Immediate |
| `CSRRSI rd, csr, imm5` | Control and Status Register Atomic Read and Set Bits Immediate |
| `CSRRCI rd, csr, imm5` | Control and Status Register Atomic Read and Write Immediate |

The following code sample shows how to enable interrupts, enable timer interuppts, and then set and wait for a timer interrupt to occur. The example uses CSR instructions and access to a platform specific MMIO *(memory mapped input output)* region:

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

# setup machine trap vector
1:      auipc   t0, %pcrel_hi(mtvec)        # load mtvec(
        addi    t0, t0, %pcrel_lo(1b)       # load mtvec(
        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 les
        slli    t0, t0, 1      # shift off high bit
        srli    t0, t0, 1
        li      t1, 7          # check this is an m_timer
        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"
```

## References

- RISC-V Foundation
- RISC-V ISA Specification
- RISC-V GNU Toolchain

This page was generated by GitHub Pages.