## Pseudoinstructions

To make hand-written assembly coding easier, most RISC-V assemblers accept the following instructions as input, and produce the equivalent RISC-V instructions (so dissassembly might not look familiar)

| name | pseudo-instruction | meaning |
|---|---|---|
| branch if = 0 | beqz rs1, label | jump to label if rs1 == 0 |
| branch if ≠ 0 | bnez rs1, label | jump to label if rs1 ≠ 0 |
| jump | j label | jump to label |
| jump register | jr offset | jalr |
| load address | la rd, symbol | rd ← symbol address |
| load immediate | li rd, expr | rd ← expr value |
| move | mv rd, rs | rd ← rs |
| negate | neg rd, rs | rd ← -1 * rs |
| no operation | nop | pc advances |
| bitwise not | not rd, rs | rd ← ¬ rs |
| return | ret | pc ← ra |
| set = zero | seqz rd, rs | rd ← rs == 0 ? 1 : 0 |
| set ≠ 0 | snez rd, rs | rd ← rs ≠ 0 ? 1 : 0 |

## Registers/calling conventions

| reg | name | use | saved by |
|---|---|---|---|
| x0 | zero | constant 0 | - |
| x1 | ra | return addr | caller |
| x2 | sp | stack pointer | callee |
| x3 | gp | global pointer | - |
| x4 | tp | thread pointer | - |
| x5-x7 | t0-t2 | temporaries | caller |
| x8 | s0/fp | saved reg/ frame pointer | callee |
| x9 | s1 | saved reg | callee |
| x10-x11 | a0-a1 | args / return values | caller |
| x12-x17 | a2-a7 | function args | callee |
| x18-x27 | s2-s11 | saved registers | callee |
| x28-x31 | t3-t6 | temporaries | caller |

*callee saved* registers must be saved (to the stack) by a function if it modifies them.

*caller saved* registers are assumed to be over-written, so they must be saved by the caller before they call any other function if they need those values.

To save something on the stack, first decrement the stack pointer (usually this is done once, at the top of a function), then use a *store* operator with an immediate offset from the `sp`. Similarly, to restore from the stack, use a *load* operator into the register being restored. See the factorial example on this card for an example. Compilers often use a *frame pointer*, stored in `fp`, to simplify accounting of what's currently in scope. It's common for hand-written assembly to just use the stack pointer.

*RISC-V is an open instruction set architecture, meaning anyone can implement and modify it. Many implementations already exist, and the outlook is exciting. RISC-V already defines a number of variants, including I (integer-only), M (includes multiplier) 32/64 (word-size in bits), E (minimized, for embedded), F (floating point), etc. The version documented here is approximately RV32IM, as supported by Jupiter.*

RISC-V®

RISC-V®

# basic assembly programmer's quick reference card

*v0.1 2019/09/18*

*RV32IM*

Prepared by Dylan McNamee as supporting material for a CS 2 class. Comments, corrections, suggestions welcome to *dylan.mcnamee@gmail.com*.

# Hello, world!

Jupiter is an open source RISC-V assembly IDE. It can be downloaded from `https://github.com/andrescv/Jupiter`. A RISC-V assembly program consists of *sections*[1], indicated by *assembler directives,* which start with a ".", along with variable/function declarations, indicated by a ":", as well as assembly instructions.

Below is "Hello, world" in RISC-V assembly. Type it into Jupiter's Edit screen, then assemble it (F3) and run it with the green "play" button. You can also single-step with the orange button, and observe the registers and memory locations get updated as each instruction executes.

```
# Hello, world in RVI32.  Note: comments start with #
.data # .data => read-write variables - 3 are defined here:
        # name    type        value
        hello:    .string     "Hello, world!\n"
        aByte:    .byte       0x42
        aWord:    .word       0xcafef00d
.globl __start # .globl symbols are visible outside this file
.text          # .text => program instructions
__start:
        la        a1, hello   # la is a pseudoinstruction
        addi      a0, x0, 4   # a0 <- 4 (print_string)
        ecall                 # executes the call specified in a0
        addi      a1, x0, 0   # could also use li a1, 0
        addi      a0, x0, 17  # exit in Jupiter
        ecall                 # doesn't return
```

# Jupiter environment calls

*Environment calls* are how an assembly program interacts with the environment, such as reading input, or printing output. They often take arguments in register `a1`, the system call code is loaded in `a0`, and the `ecall` instruction initiates the system call. Any return value is left in `a0`

| name | code | args | return |
|---|---|---|---|
| print_int | 1 | a1 (i32) | |
| print_string | 4 | a1 (addr) | |
| read_int | 5 | | i32 in a0 |
| read_string | 8 | a0(addr), a1(len) | |
| sbrk (alloc mem) | 9 | a1 (amount) | addr in a0 (or 0) |
| exit | 17 | a0 (i32) exit value | |

[1] valid sections include: `.data`, `.text` (as in the example) as well as `.bss` for uninitialized data, and `.rodata` for read-only variables.

# Basic instruction set

The table below shows enough instructions for you to write many useful programs in RISC-V assembly.

**rd** refers to the destination register

**rs** refers to a source register

**imm** is an immediate value such as 0 or 0xf00d

When loading or storing from memory, parentheses are used to describe indirection - the value inside of parentheses is a pointer, and that memory location is operated on.

| name | format | meaning |
|---|---|---|
| load word | lw rd, imm(rs) | rd ← (rs+imm) |
| store word | sw rs1, imm(rs2) | (rs2+imm) ← rs1 |
| shift left | sll rd, rs1, rs2 | rd ← rs1 << rs2 |
| shift left imm | slli rd, rs1, imm | rd ← rs1 << imm |
| shift right | srl rd, rs1, rs2 | rd ← rs1 >> rs2 |
| shift right arith | sra rd, rs1, rs2 | rd ← rs1 >> rs2 |
| xor(imm) | xor(i) rd, rs1, rs2(/imm) | rd ← rs1 ⊕ (rs2 or imm) |
| or(imm) | or(i) rd, rs1, rs2(/imm) | rd ← rs1 \| (rs2 or imm) |
| and(imm) | and rd, rs1, rs2(/imm) | rd ← rs1 & (rs2 or imm) |
| add(imm) | add(i) rd, rs1, rs2(/imm) | rd ← rs1 + (rs2 or imm) |
| subtract | sub rd, rs1, rs2 | rd ← rs1 - rs2 |
| multiply(unsigned) | mul(u) rd, rs1, rs2 | rd ← rs1 * rs2 |
| divide(unsigned) | div(u) rd, rs1, rs2 | rd ← rs1 / rs2 |
| remainder (unsigned) | rem(u) rd, rs1, rs2 | rd ← rs1 % rs2 |
| set less-than | slt(i) rd, rs1, rs2(/imm) | rd ← rs1 < (rs2 or imm) |
| set less-than unsigned | sltu(i) rd, rs1, rs2(/imm) | rd ← rs1 < (rs2 or imm) |
| branch if == | beq rs1, rs2, label | jumps to label if rs1 == rs2 |
| branch if ≠ | bne rs1, rs2, label | jumps to label if rs1 ≠ rs2 |
| branch if < | blt(u) rs1, rs2, label | jumps to label if rs1 < rs2 |
| branch if ≥ | bge(u) rs1, rs2, label | jumps to label if rs1 ≥ rs2 |
| jump and link | jal label | jumps to label, ra ← return |
| jump and link reg | jalr rd, label | jumps to label, rd ← return |

# Conditionals and jumps

```
.data
    prompt:   .string "give me a number for analysis:"
    big_msg:  .string "wow-that's a big number!"
    small_msg:.string "aww, what a cute number"
.globl      __start
.text
__start:
    la      a1, prompt
    li      a0, 4                # print_string
    ecall
    li      a0, 5                # read_int
    ecall
    li      t0, 6                # threshold for comparison
    blt     a0, t0, smaller      # jump if small input
# fall through to here if not smaller
    li      a0, 4
    la      a1, big_msg
    ecall                        # print msg call
    j       done
smaller:
    li      a0, 4                # print msg call
    la      a1, small_msg
    ecall
done:
    li      a0, 17               # exit call
    li      a1, 0                # exit code (0 == ok)
    ecall
```

# *Functions, the stack and recursion*

```
.text  # recursive implementation of factorial
.globl __start
fact:                       # arg: n in a0, returns n! in a1
    addi  sp, sp, -8        # reserve our stack area
    sw ra, 0(sp)            # save the return address
    li t0, 2
    blt a0, t0, ret_one     # 0! and 1! == 1
    sw a0, 4(sp)            # save our n
    addi a0, a0, -1
    jal fact                # call fact (n-1)
                            # a1 <- fact(n-1)
    lw t0, 4(sp)            # t0 <- n
    mul a1, t0, a1          # a1 <- n * fact(n-1)
    j done
ret_one:
    li a1, 1
done:
    lw ra, 0(sp)            # load the return address
    addi sp, sp, 8          # restore the stack pointer
    jr ra                   # and return

__start:
    li a0, 5                # compute 5!
    jal fact
    li a0, 1                # print it
    ecall
    li a0, 17
    ecall                   # and exit
```