Interpreter: [RISC-V Interpreter](https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/)

### **Registers Types**

* zero: Hold “zero” value
* t0 → t6: Temporary value - discarded after function calls / Caller
* s0 → s11: Saved value - saved after function calls / Callee
* a0 → a1: Function arguments/return value
* a2 → a7: Function arguments

Caller & Callee:

* Caller: used to hold temporary values that need not be preserved after calls.
* Callee: used to hold long-lived values that need to be preserved during calls.

⇒ All are programming conventions.

# **Instructions**

**XLEN:** the width of an x register in bits

## **R-type (Register type)**

* Operate purely on registers.
* Format: rd = rs1 op rs2
* Arithmetic:
  + ADD, SUB: Addition and subtraction. **Ignore arithmetic overflow.**
  + SLT: “Less than”. If rs1 < rs2 then rd = 1
  + SLTU: Similar, but rs1 and rs2 are treated as **unsigned numbers.**
* Logical: AND, XOR, OR - bitwise
* Shifting:
  + SLL: Shift left **logical** - rs2 zero(s) are inserted to the Least Significant Bit.
  + SRL: Shift right **logical** - rs2 zero(s) are inserted to the Most Significant Bit.
  + SRA: Shift right **arithmetic** - **signed bit(s) is copied** to the Most Significant Bit.

## **I-type (Immediate type)**

* Use only two registers as operands and a 12-bit constant Imm
  + Source registers rs1
  + Destination registers rd
* Format op rd rs1 Imm or rd = rs1 op Imm
* ALU arithmetic & logic immediate:
  + ADDI: Add the 12-bit Imm to register rs1. **Ignore arithmetic overflow.**
    - Subtraction is handled by ADDI because RISC-V is designed to be minimal.
  + SLTI: “Less than”. If rs1 < sign-extended Imm then rd = 1 , else rd = 0. Both are treated as signed numbers.
  + SLTIU: “Less than”, but rs1 and Imm are treated as unsigned numbers.
  + ANDI, ORI, XORI - bitwise
  + SLLI: Logical left shift on the value in register rs1 by Imm bits, stored in rd.
  + SRLI: Logical right shift on the value in register rs1 by Imm bits, stored in rd.
  + SRAI: Arithmetic right shift on the value in register rs1 by Imm bits.
* Memory Load:
  + Memory types:
    - B: byte - 8 bits
    - H: half a word - 16 bits
    - W: word - 32 bits (word in RISC-V is not a word in human language)
  + LB, LH, LW: Loads 8-bit, 16-bit, and 32-bit value from memory and ***sign-extends*** this to XLEN bits before storing in register rd.
    - LW is usually used to load saved register values from stack as well! See below!
  + LBU, LHU: Loads 8-bit and 16-bit value from memory and ***zero-extends*** this to XLEN bits before storing in register rd.
    - **Note:** There is no LWU because it functions the same as LW.

## **S-type**

* SB: Store first 8 bits value of rs2 to memory at pos rs1 + sign-extend(Imm)
* SH: Store first 16 bits value of rs2 to memory at pos rs1 + sign-extend(Imm)
* SW: Store first 32 bits value of rs2 to memory at pos rs1 + sign-extend(Imm)
  + **Most important!** We usually use this to save registers to stack:

addi sp, sp, -4 # Move stack pointer back 4 bytes = 1 word = 32 bit = space for 1 register  
sw t0, 0(sp) # Save to first space in stack  
  
… # some calculation thingies  
  
lw t0, 0(sp) # load from first space in stack  
addi sp, sp, 4 # Move stack pointer forward to original (it’s called deallocation)

## **B-type**

* Used as **if-else statement**
* Most can be inferred by name: beq - equal to, blt - less than, bge - greater or equal, …
* Format: op rs1, rs2, Imm. In the RISC-V context, Imm is just the name for a **flag** in the program - the branch will **jump** to that flag in the code.

Example: This can later be combined with **jump** commands (J-type) to make an if-else block as well

BGE t0, t1, if\_block

# whatever code here

// code1  
J END # Jump to end

if\_block:

# whatever code here

// code2

END:

## **U-type**

* LUI: Initialize the upper 20 bits of a register. Used to initialize big Hex values.

Example:

# value of last 12 bits < 2048

LUI t0, 0xABCDE # t0 = 0xABCDE000

ADDI t0, t0, 0x123 # t0 = 0xABCDE123

# value of last 12 bits >= 2048 i.e. the sixth digit > 7

LUI t0, 0xABCDF # t0 = 0xABCDF000

ADDI t0, t0, 0xFFFFFFHG = 0xABCDEFGH

## **J-type**

* JAL: Jump to Imm address and store return address in rd
  + rd = pc + 4
  + Can use LABEL in place of Imm
* JALR: Jump to rs1 + sign-extend(Imm) address and return address in rd
  + rd = pc + 4
  + Syntax can be rd, Imm(rs1) or rd, rs1, Imm
  + **Important: Pseudo-code** JR rs1 **#is allowed on the exam! Use that instead!**

Example:

MAIN: ADDI a0, a0, 10

ADD a1, a1, 20

JAL ra, SUM

J EXIT

SUM: ADD a0, a0, a1

JR ra

EXIT:

# a0 should now be 30, you can save it to other registers i.e.  
addi t0, zero, a0

**Loops in RISC-V**

* Structure/Steps:
  + Initialize loop control variable i.e. index
  + Code block (we’ll use FOR flag as an example):
    - Check if the loop control variable still valid i.e. BGE t0, t1, END, equal to if (i >= n) break;
    - Calculations as needed
    - J FOR # Jump back to top

**Array access**

* In exams, you will most likely be given the address of the array index 0 i.e. &a[0] = a in C. We’ll assume it’s saved in t0.
* To access array element index i (assume saved in t1), we do similar to C:

SLLI t2, t1, 2 # t2 = index \* 4

ADDI t3, t0, t2 # t3 = a + index \* 4

LW t4, 0(t3) # t4 = \*(a + index \* 4 + 0) # 0 is the addition, but you can’t pass t2 directly, so we need t3

ADDI t1, t1, 1 # i = i + 1

* The only difference is that in Assembly, the program doesn’t auto-determine the length of each element e.g. int is 4 bytes (or a word) so you need to do your own addition as in (a + index \* 4)
* For char array (strings), since each char is only **one** byte, you can directly use (a + index) to find a[i].

**Sample - Array access and Loops**

la t0, array # load array - t0 now hold address of a[0]

lw t1, array\_len # load length - can probably ignore this

# In real exams, they will give you the pointer address e.g. “The starting address of the array is stored in t0 register” or something similar

addi t2, zero, 0 # t2 = i = 0

addi t3, zero, 0 # t3 = sum = 0

j loop

loop:

bge t2, t1, end # if i >= n -> jump to end

lw t4, 0(t0) # t4 = \*(a + 0)

add t3, t3, t4 # sum = sum + a[i]

addi t0, t0, 4 # jump to a[i+1] - each int is 4 bytes

addi t2, t2, 1 # i = i + 1

j loop

end:

Note: Instead of the above, you can use t4 = t0 + index \* 4 for instance, to emulate a for-loop of some sort, as mentioned in the “Array access” section.

## Functions

Caller function (can be main):

* Passes arguments to callee
* Jumps to callee
* If the caller function needs to save any variable, it’s the caller’s responsibility. Variables can be saved on the stack (see below).

Callee function:

* **Performs** the function
* **Returns** the result to the caller
* **Returns** to the point of call

Control Transfer:

* **Caller** uses JAL(Jump and Link) to jump to **callee**
* The **callee** returns control to the caller using JR RA (Jump Register to Return Address)
* When having more than 1 caller function, save ra to stack, so that you can load ra later and use jr ra to return. If you don’t, multiple ra will be saved to the same register, and you’ll jump to who-know-where.
  + Recursive functions are simply multiple ra in the stack.

Example:

# using caller-saved convention

# Caller solve function (non-leaf, main will call solve)

solve:

addi a0, zero, 42 # Argument 1 (a = 42)

addi a1, zero, 7 # Argument 2 (b = 7)

addi sp, sp, -4

sw ra, 0(sp)

jal ra, sum\_square # Call sum function, return address saved in ra

add s7, a0, zero # Store result in s7 (y = sum result)

lw ra, 0(sp)

addi, sp, sp, 4

jr ra

# Caller sum function (non-leaf)

sum\_square:

addi sp, sp, -4

sw ra, 0(sp) # add return address to stack

jal ra, square # call square function, loose ra

addi t0, a0, 0 # a1 = b, set a0 = b to call square function

addi a0, a1, 0 # t0 = a^2

jal ra, square # call square func, a0 = b^2

addi t1, a0, 0 # t1 = b^2

add a0, t0, t1 # a0 = a^2 + b^2

lw ra, 0(sp)

addi sp, sp, 4

jr ra # Return to caller

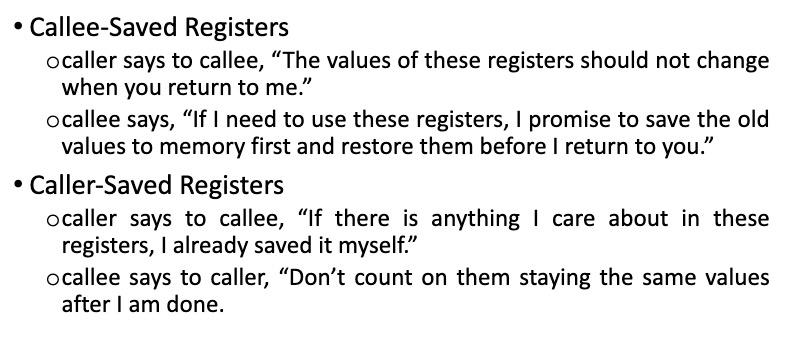
# Callee function (leaf)

square:

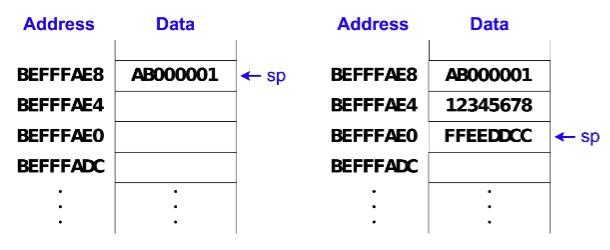
# code for squaring a number

jr ra

## Caller and Callee Saved Registers - Convention



## Stack

* Memory used to temporary save variables
* Last-in-first-out (LIFO) queue
* **Expands:** uses more memory when more space needed
* **Contracts:** uses less memory when the space is no longer needed
* Grows down (from higher to lower memory addresses)
* Stack pointer: SP points to top of the stack
* Reason for decreasing SP when saving memory: Avoid collision with heap pointer, act as the top of the memory (two pointers).

Example:

ADDI sp, sp, -12

# add three more words(4 bytes) to stack, intuitively

SW t0, 0(sp) # store t0 to mem[sp: sp + 3]

SW t1, 4(sp) # store t1 to mem[sp + 4: sp + 7]

SW t2, 8(sp) # store t2 to mem[sp + 8: sp + 11]  
 ... # Some code here

LW t0, 0(sp) # load mem[sp: sp + 3] to t0

LW t1, 4(sp) # load mem[sp + 4: sp + 7] to t1

LW t2, 8(sp) # load mem[sp + 8: sp + 11] to t2

ADDI sp, sp, 12 # deallocation

## Bubble Sort

# i: t0, j: t1, swapped = t2, t3 = n - 1

bubbleSort:

ADD t0, zero, zero # i = 0

ADDI t3, a1, -1 # t3 = n - 1

For1:

BGE t0, t3, Exit1 # if (i >= n - 1) goto Exit1

ADD t2, zero, zero # swapped = false

ADD t1, zero, zero # j = 0

SUB t4, t3, t0 # t4 = n - 1 - i

For2:

BGE t1, t4, Exit2 # if (j >= n - i - 1) goto Exit2

SLLI t5, t1, 2 # t5 = j\*4 (byte offset for int)

ADD t5, a0, t5 # t5 = &arr[j]

LW t6, 0(t5) # t6 = arr[j]

LW S0, 4(t5) # S0 = arr[j + 1]

IF:

BGE S0, t6, ENDIF # if (arr[j] <= arr[j + 1]) goto ENDIF

SW S0, 0(t5) # arr[j] = arr[j + i]

SW t6, 4(t5) # arr[j + 1] = arr[j]

ADDI t2, zero, 1 # swapped = true

ENDIF:

ADDI t1, t1, 1 # j++

J For2 # Jump to For2

Exit2:

BEQ t2, zero, Exit1 # if (!swapped) goto Exit1

ADDI t0, t0, 1 # i++

J For1 # Jump to For1

Exit1:

swap:

LW t0, 0(a0) # Load \*xp into t0

LW t1, 0(a1) # Load \*yp into t1

SW t0, 0(a1) # Store t0 into \*yp

SW t1, 0(a0) # Store t1 into \*xp

JR ra # Return to caller