Discussion: RISC-V

YUCHENG@SHANGHAITECH.EDU.CN

RISC-V Registers

- In RISC-V, we have two methods of storing data, one of them is main memory, the other is through registers.
- Registers are much faster than using main memory, but are very limited in space (32-bits).
- Note that you should ALWAYS use the named registers (e.g. s0 rather than x8).

REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

RISC-V Instructions

- Instruction Syntax is rigid:

  `op dst, src1, src2`

  1 operator, 3 operands

  - `op` = operation name (“operator”)
  - `dst` = register getting result (“destination”)
  - `src1` = first register for operation (“source 1”)
  - `src2` = second register for operation (“source 2”)
- Keep hardware simple via regularity
- One operation per instruction, at most one instruction per line
- Assembly instructions are related to C operations (=, +, -, *, /, &, |, etc.)

Memory

- RISC-V does not require word alignment.
- But you'd better do this.

- **sw** stands for store word.
  - sw s2, 4(sp) → store 32 bits (1 word) data into the address store in sp plus 4 bytes.
- **lw** stands for load word.
  - lw sp, -4(sp) → load 32 bits data from the address (sp – 4) into sp.

- This two instruction use memory on stack.
- If you want to use memory on heap, use environment call 9.
- sp, s0-s11, ra, which you should maintain them value but need to use now: **push them on stack.**

What do the snippets of RISC-V code do?

Assume we have an array in memory that contains int* arr = {1,2,3,4,5,6,0}.

Let register `s0` hold the address of the zeroth element in arr. You may assume integers are four-bytes and our values are word-aligned.

What do the snippets of RISC-V code do?

Assume that all the instructions are run one after the other in the same context.

a)
```
lw   t0, 12(s0)
```
--> Sets t0 equal to arr[3]

b)
```
slli t1, t0, 2
add  t2, s0, t1
lw   t3, 0(t2)
addi t3, t3, 1
sw   t3, 0(t2)
```
--> Increments arr[t0] by 1

c)
```
lw   t0, 0(s0)
xori t0, t0, 0xFFF
addi t0, t0, 1
```
--> Sets t0 to -1 * arr[0]

Label and Branch

- Giving a line name by adding label.
- Then, you can go the label by jump or branch.
- You can use label in function call, if-else, loop, etc
- Let your label easy to understand, that makes you easy to finish the given tasks.

RISC-V Calling Conventions

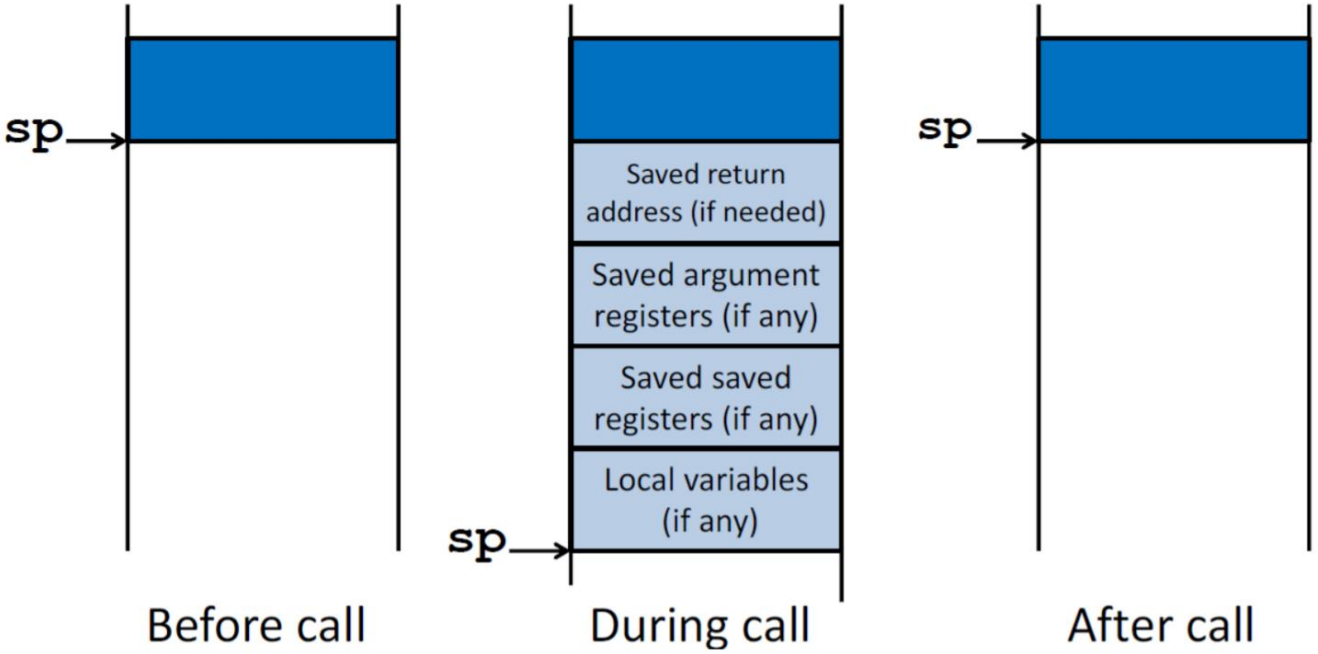
- Values saved by the **caller** before jumping to a function using `jal`
  - ra: Return address, used in function call.
  - a0-a1: Function argument and return values, also argument of environment call.
  - a2-a7: Function argument, used to pass parameters in function call.
  - t0-t6: Temporaries, cannot trust them after function call.

- Values restored by the **callee** before returning from a function using `jalr`
  - sp: Stack pointer. We subtract from sp to create more space and add to free space. The stack is mainly used to save (and later restore) the value of registers that may be overwritten.
  - s0-s11: Saved registers, should not change after function call.

Function Call

- Caller & Callee
  - Caller invoke callee.
  - Callee should make sure he haven’t change caller saved registers.
- Steps of function call
  - Caller put parameters into registers a0-a7.
  - Caller put next line’s address into ra and jump to the function label. (using jal)
  - Callee pushes s0-s11, sp onto stack. (attention: ra’s saver is not callee)
  - Callee execution.
  - Callee extract value from stack.
  - Callee jump to ra’s address.

The Stack's Condition



Example: sumSquare

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

```
sumSquare:
"push"  addi sp,sp,-8        # make space on stack
        sw ra, 4(sp)         # save ret addr
        sw a1, 0(sp)         # save y
        add a1,a0,x0         # set 2nd mult arg
        jal mult             # call mult
"pop"   lw a1, 0(sp)         # restore y
        add a0,a0,a1         # ret val = mult(x,x)+y
        lw ra, 4(sp)         # get ret addr
        addi sp,sp,8         # restore stack
        jr ra
mult:   ...
```

Choosing Your Registers

- Minimize register footprint
  - Optimize to reduce number of registers you need to save by choosing which registers to use in a function
  - Only save when you absolutely have to
- Function does NOT call another function
  - Use only t0-t6 and there is nothing to save!
- Function calls other function(s)
  - Values you need throughout go in s0-s11, others go in t0-t6
  - At each function call, check number arguments and return values for whether you or not you need to save