Week4 RISCV Control Flows

Friday, March 18, 2022 4:30 AM

0. Program Counter

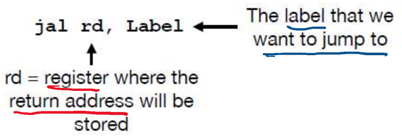
- **Program Counter(PC)** is a register that holds the **memory address of the instruction** being executed
- As **RV32** instructions are 32bits (4 bytes).
  - When moving to the next instruction, PC = PC + 4

1. Revisit Jump instructions: PC to execute functions at different location

1.1 JAL instruction (Jump-and-Link)

Jump instructions need to know where to return when we are finished with the function call

- Jump instructions need to do two things:
  - Store the return address
  - Update the value of PC


jal rd, Label
The label that we want to jump to
rd = register where the return address will be stored

rd = return address

PC = PC + offset (Label)

The **label** that we want to jump gets translated by the assembler to a **20-bit offset**

1.1.a: *rd* (Return address Register) conventions: Use **x1** (**ra**) conventionally

- Though **rd** could be any register
- **Standard convention** designate **x1** to hold **return address**
  - x1 has an alternate name = **ra**

```
jal ra, L1
```

return address register

1.1.b: *rd* for plain branches, a.k.a. Return address are not needed

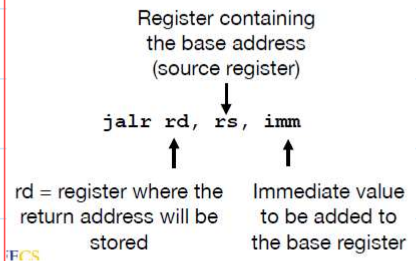
- Specify ***r0*** as the return address destination registers

```
jal x0, L1
```

- Note, j Label is the pseudo instruction for jal x0, Label

1.2 JALR instruction (Jump-and-Link-register)

Recall JAL offset is **limited to 20 bits**. To jump to **anywhere** in memory, use JALR; where ***rs***: destination base addr


Register containing the base address (source register)
jalr rd, rs, imm
rd = register where the return address will be stored
Immediate value to be added to the base register

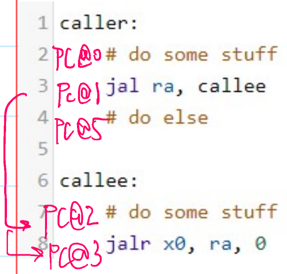
rd = return address

PC = [rs] + imm

1.2.a: Return from a function: set **rd** to **x0**

Caller "remembers" its ***pc(return address)*** at ***ra;*** Return to callee almost never happens: `jalr x0, ra, 0`

1.3 Jump Example: Call a function and return to caller



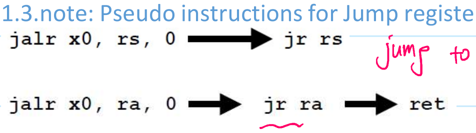
```
caller:
PC@0 # do some stuff
PC@1 jal ra, callee
PC@5 # do else

callee:
PC@2 # do some stuff
PC@3 jalr x0, ra, 0
```

ra = PC@1 @ jump to callee

PC@4

#jump to [ra]+0.

1.3.note: Pseudo instructions for Jump register (JALR)



```
jalr x0, rs, 0 → jr rs
jalr x0, ra, 0 → jr ra → ret
```

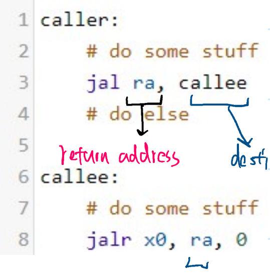
jump to [rs] and not save return address jalr x0, rs, 0 ➡ jr rs Jump to [rs] and Not save return address

jalr x0, ra, 0 ➡ jr ra ➡ ret

return to callee ⟺ ret

1.4. JAL/JALR Summary

- Jump and link (JAL)
  - **jal rd, label** Jump to PC+OFFSET, and save current PC to rd
    - **jal x0, label -> j label** Jump but NOT save PC
- Jump and link Register (JALR)
  - **jalr rd, rs, imm** (rs = source register) Jump to [RS]+imm. Save current PC to rd
    - **jalr x0, rs, 0 -> jr rs**
    - **jalr x0, ra, 0 -> jr ra -> ret**



```
caller:
    # do some stuff
    jal ra, callee
    # do else

callee:
    # do some stuff
    jalr x0, ra, 0
```

return address

destination PC

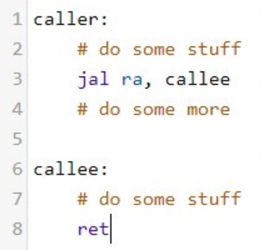
jump to PC=callee;

Save current address to ra (x1)

jump to L6

after function processing.

jump back PC=ra+0. return address is discarded.



```
caller:
    # do some stuff
    jal ra, callee
    # do some more

callee:
    # do some stuff
    ret
```

2. Dealing with Memory: **STACK**

2.0 Problem of overwriting registers

- **Problem:** When calling another function, other function needs to use those registers for its computation. Then temp values in register **may overwrite** our values

- **One Solution:** Save all of the registers we are using before we call a function, after function call, restoring the values
- **Where** to save these values? **Stack**

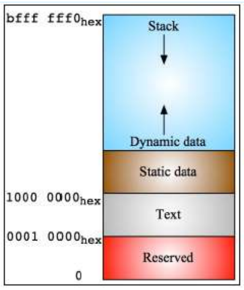
C has two storage classes: **automatic and static**

- **Automatic** variables are local to a function and **discarded after function exits**
- **Static** variables exist across exits from and entries to procedures

Use stack for automatic variables that are not in registers

2.1 Stack Pointer

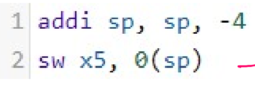
A register that holds the memory address of the location of the **last item placed on the stack**. By convention, **its register x2**



- **Make room for storing**
- When place an item on the stack, **SP decrements by x byte**
  - **PUSH** data in, address **decreases**
  - ***addi sp, sp, -x***

- **Removing from stack**
- When take an item off the stack, **SP increments by x byte**
  - **POP** data out, address **increases**
  - ***addi, sp, sp, x***

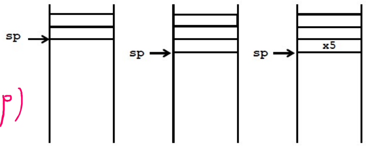
2.1.a. Example: Store x5 (int) on the stack



```
addi sp, sp, -4
sw x5, 0(sp)
```

→ make room

→ save x5 to 0(sp)



2.2 Function calling anatomy

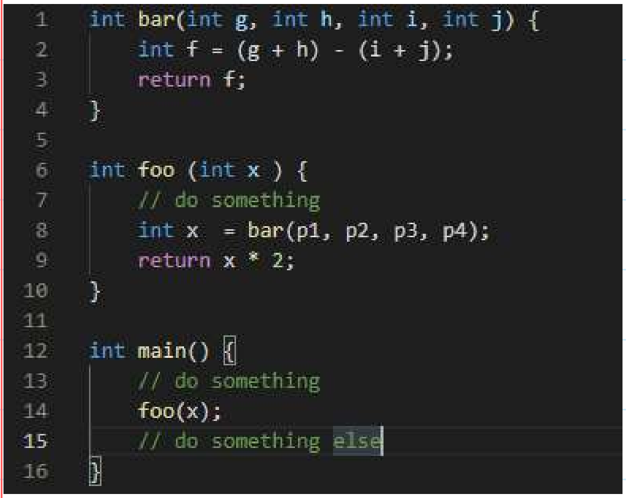
- **Temporary registers**
  - **Saved by callers** before calling a function
- **Saved registers**
  - **Saved by the callee** before we use them
- Function also need to have **place for** where they can expect the **arguments and return value to be**:

- ○ ***x10-x17*** are reserved for ***argument registers***, with new name: ***a0-a7***
- ○ ***a0* and *a1* will serve as return value register**s
  - ▪ If caller has temp values that it wants to use after making function call, it must save those values.
  - ▪ If you want to use a value after function call, that value is the **returned value**

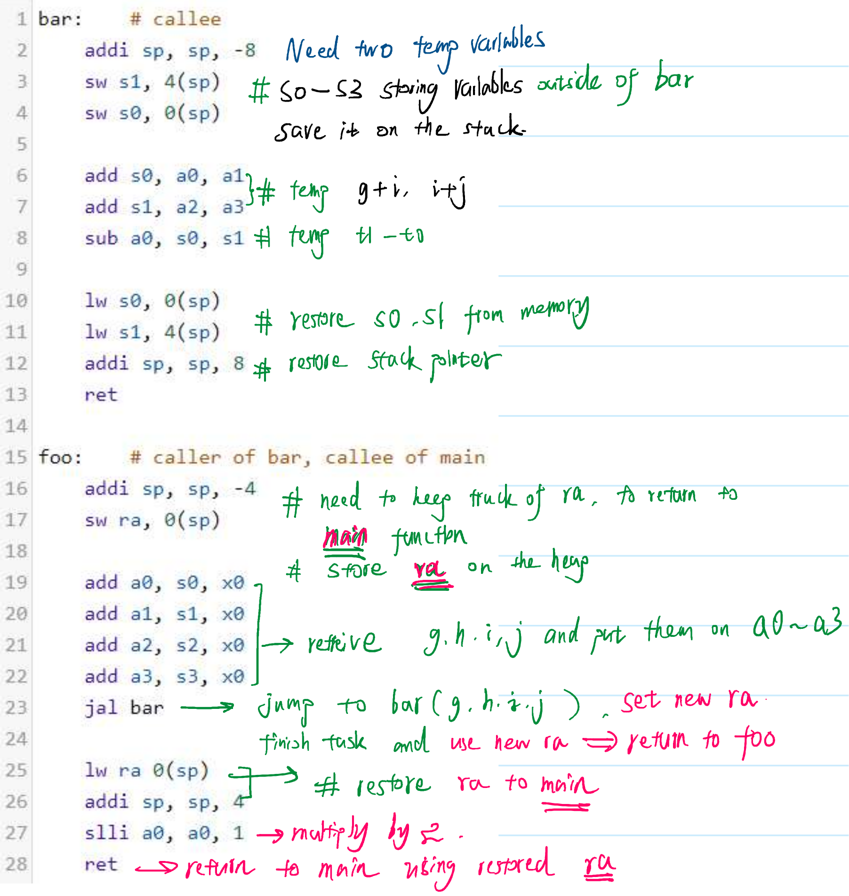
2.2.a. Register calling **Conventions references**

Register	Name	Description	Saved by
x0	zero	Always Zero	N/A
x1	ra	Return Address	Caller
x2	sp	Stack Pointer	Callee
x3	gp	Global Pointer	N/A
x4	tp	Thread Pointer	N/A
x5-7	t0-2	Temporary	Caller
x8-x9	s0-s1	Saved Registers	Callee
x10-x17	a0-7	Function Arguments/Return Values	Caller
x18-27	s2-11	Saved Registers	Callee
x28-31	t3-6	Temporaries	Caller

2.3. Function call example:



```c
int bar(int g, int h, int i, int j) {
    int f = (g + h) - (i + j);
    return f;
}

int foo (int x ) {
    // do something
    int x  = bar(p1, p2, p3, p4);
    return x * 2;
}

int main() {
    // do something
    foo(x);
    // do something else
}
```



```
bar:     # callee
    addi sp, sp, -8     Need two temp varlables
    sw s1, 4(sp)        # s0 – s3 storing varlables outside of bar
    sw s0, 0(sp)          save it on the stack.

    add s0, a0, a1 }# temp  g+i, i+j
    add s1, a2, a3 }
    sub a0, s0, s1 # temp  t1 – t0

    lw s0, 0(sp)
    lw s1, 4(sp)        # restore s0 , s1 from memory
    addi sp, sp, 8 # restore stack pointer
    ret

foo:     # caller of bar, callee of main
    addi sp, sp, -4     # need to keep track of ra, to return to
    sw ra, 0(sp)          main function
                        # store ra on the heap
    add a0, s0, x0 ┐
    add a1, s1, x0 │
    add a2, s2, x0 ├→ retrieve  g.h.i,j and put them on a0~a3
    add a3, s3, x0 ┘
    jal bar  ──→ jump to bar(g.h.i.j) , set new ra.
                 finish task and use new ra ⇒ return to foo
    lw ra 0(sp)    ┐→ # restore ra to main
    addi sp, sp, 4 ┘
    slli a0, a0, 1 → multiply by 2 .
    ret ⟶ return to main using restored ra
```

Assume *g, h, I, j* are already in s0-s3 when calling foo

3. Summary of function calls

1. **Put parameters in a place** where function can access them **(argument registers)**
   - ○ In argument registers (***a0 - a7****, i.e.,* ***x10 - x17***)
2. **Transfer control** to function

- JAL / JALR instructions
  - ○ If offset of branching is smaller (20 bits): use JAL
    - ▪ ***jal rd, Label***: Jump to label, and record return address in ***rd***(***ra*** by convention)
      - □ ***jal ra, label*** <=> ***jal label***
    - ▪ If no need to store ***rd***, use pseudo instruction: ***jal x0, Label <=> j Label***
  - ○ Otherwise use JALR
    - ▪ ***jalr rd, rs, imm***: Jump to ***label = rs + imm*** *(offset)***,** store return address ***in rd*** (***ra*** by convention)
    - ▪ If no need to record return address: ***rd = x0;*** and **No** imm offset***: imm = 0***
      - □ ***jalr x0, rs, 0 <=> jr rs***
    - ▪ When ***rs*** is the ***ra,*** which stores the return address of **caller*: jalr x0, ra, 0 <=> jr ra <=> ret***

3. **Acquire (local) storage** needed for function
    - Make rooms for local variables, **decrement (SP)**: for 32 bit system, decrement ***sp*** by 4 bytes
    - **save** the variables we **don't** want to **overwrite:** typically in ***s0 - s11***
4. *Perform tasks in function*
5. Put **result value in a place where** caller can access (**return value registers**)
    - **a0-a1** register
6. Return control to point of origin
    - ***jalr x0, ra, 0 <=> jr ra <=> ret***