CSC 411

Computer Organization (Fall 2024)
Lecture 13: RISC-V procedures

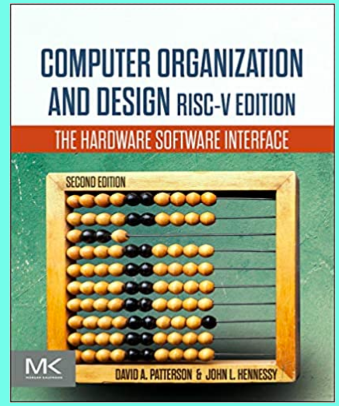
Prof. Marco Alvarez, University of Rhode Island

Disclaimer

Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

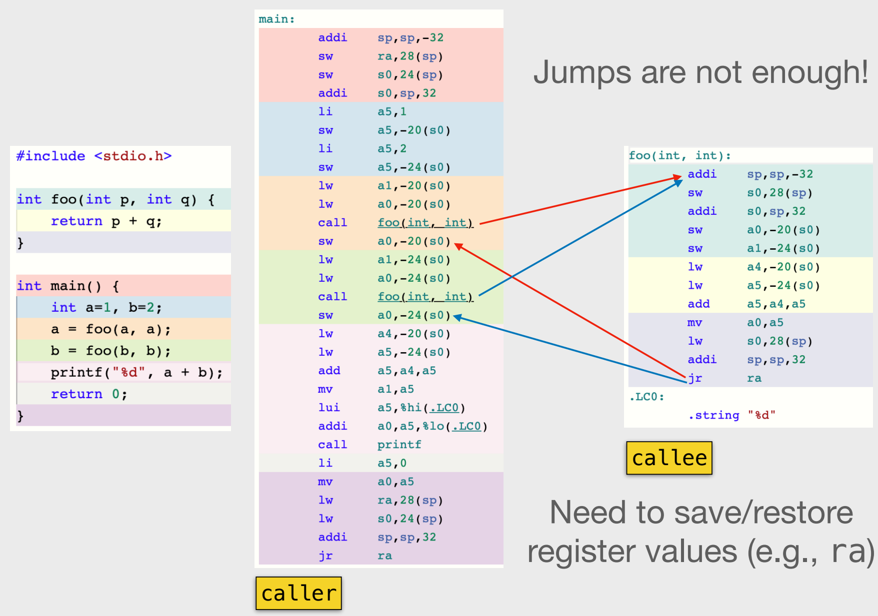
The Hardware/Software Interface



Procedures in RISC-V

- C functions maintain local scope separate from global scope
  - RISC-V has no inherent concept of local scope
    - all registers are "globally" accessible throughout the program, including recursive function calls
- Return addresses
  - need to return to the instruction after the call
  - simple "jump label" instructions are insufficient due to multiple call sites
  - solution: treat the return address as an input to the function

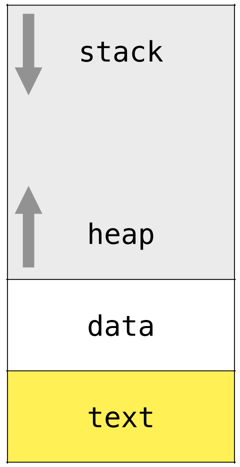
Jumps are not enough!



Need to save/restore register values (e.g., ra)

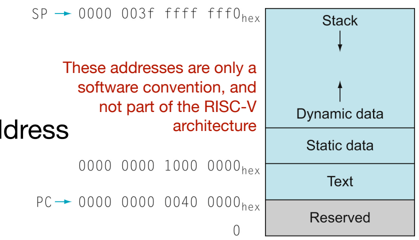
C memory model

- Memory is divided into four segments
  - code/text
  - static/data
  - heap
  - stack



RISC-V memory model

- Text segment
  - contains instructions
  - each (real) instruction is a **32-bit word**
- Static data segment
  - global variables
  - global pointer (**gp**) stores base address
    - allows offsets into this segment
- Dynamic data
  - stack: run-time stack for local procedures data
  - heap: dynamically allocated data

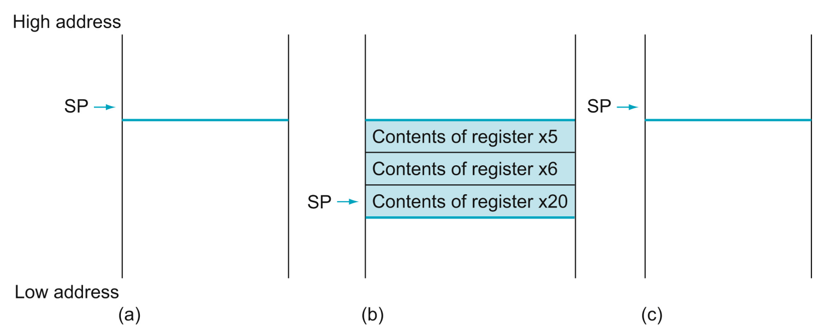


Register usage in RISC-V

- Register file as a scratchpad
  - each procedure uses the register file
  - values may need saving before calls to **resume work** after returning
- Special registers
  - Program Counter (**pc**)
    - tracks the next instruction to be executed
    - implemented separately from x0-x31 registers
  - Stack Pointer (**sp**)
    - uses register x2
    - points to the current "top" of the stack
    - stack grows downward (from high to low addresses)
    - must be restored to its original value before returning

The stack pointer in action

- The stack before, during, and after a procedure call
  - **sp** always points to the "top" of the stack (the last word added to the stack)



RISC-V register conventions

- Parameter (argument) registers
  - **a0 - a7** (x10 - x17) — used to pass parameters
  - **a0 - a1** (x10 - x11) — used to return values
- Return address register
  - **ra** (x1) — stores the return address
- Saved registers
  - **s0 - s1** (x8 - x9) and **s2 - s11** (x18 - x27)
  - must be preserved across procedure calls
  - callee must save and restore if used
- Temporary registers
  - **t0 - t2** (x5 - x7) and **t3 - t6** (x28 - x31)
  - not preserved by the callee

RISC-V register conventions

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Register management strategies

- Callee saves registers
  - assume a caller is using a **saved registers**
  - callee saves and restores if it needs to use them
- Caller saves registers
  - assume callee may modify **temporary registers**
  - caller saves and restores temporary registers if needed after the call

`Note that all register conventions are just calling conventions, register usage might vary depending on implementations/optimizations`

Procedure calling convention

- Place arguments in registers **a0 - a7**
  - use stack if more space needed
- Transfer control to procedure
- Acquire storage for procedure
  - save registers if necessary
- Perform procedure's operations
- Place return value in **a0 - a7**
- Restore any saved registers
- Return to caller
  - address in register **ra**

Jump instructions

- Jump and link (jal)
  - used for function calls — jumps to `pc + imm` and saves return address `pc + 4` in `rd`

```
jal rd, imm
```

- Jump and link register (jalr)
  - used for returns and indirect calls
  - jumps to `rs1 + imm` and saves return address `pc + 4` in `rd`

```
jalr rd, imm(rs1)
```

pseudo-instruction	equivalent RISC-V instruction
`j label`	`jal x0, label`
`jr rs1`	`jalr x0, 0(rs1)`
`ret`	`jalr x0, 0(x1)`

Examples: leaf procedures

Practice

Leaf procedure example

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

// arguments g, ..., j in a0, ..., a3
// f in temporary register t0
// saved registers s0, s1
// need to save s0, s1 on stack
```

```
leaf_example:
    addi sp, sp, -8   # reserve space for 2 registers in the stack
    sw   s0, 4(sp)
    sw   s1, 0(sp)
    add  s0, a0, a1   # perform operations
    add  s1, a2, a3
    sub  t0, s0, s1
    addi a0, t0, 0    # copy result to return register
    lw   s1, 0(sp)    # restore register values from stack
    lw   s0, 4(sp)
    addi sp, sp, 8
    jalr x0, 0(ra)    # return to caller (can use jr x1 or jr ra)
```

Practice

```
int sum_array(int *p, int n) {

}

// arguments p in a0, n in a1
// return value in a0

// s0 (sum) — saved register
// t0 (i)
// t1 (address of p[i])
// t2 (value of p[i])
// t3 (offset)
```

```
sum_array:
    addi sp, sp, -4
    sw   s0, 0(sp)
    add  t0, x0, x0
    add  s0, x0, x0
loop:
    beq  t0, a1, exit
    slli t3, t0, 2
    add  t1, a0, t3
    lw   t2, 0(t1)
    add  s0, s0, t2
    addi t0, t0, 1
    j    loop
exit:
    add  a0, x0, s0
    lw   s0, 0(sp)
    addi sp, sp, 4
    ret
```

Practice

```
// addresses x, y in a0, a1
// i in s1
void strcpy (char *x, char *y) {
    int i = 0;
    while ((x[i] = y[i]) != '\0')
        i += 1;
}
```

```
strcpy:
    addi sp, sp, -4     # adjust stack for 1 word
    sw   s1, 0(sp)      # push s1
    add  s1, x0, x0     # i=0
L1:
    add  t0, s1, a1     # t0 = addr of y[i]
    lbu  t1, 0(t0)      # t1 = y[i]
    add  t2, s1, a0     # t2 = addr of x[i]
    sb   t1, 0(t2)      # x[i] = y[i]
    beq  t1, x0, L2     # if y[i] == 0 then exit
    addi s1, s1, 1      # i = i + 1
    j  L1               # next iteration of loop
L2:
    lw   s1, 0(sp)      # restore saved s1
    addi sp, sp, 4      # pop 1 word from stack
    ret                 # and return
```

Example: non-leaf procedures

Non-leaf procedures

- Procedures that call other procedures
  - including recursive calls
- **Caller** needs to save on the stack …
  - the return address
  - any arguments and temporaries needed after the call
- Restore from the stack after the call

Practice

```
int fact (int n) {
    if (n < 2) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
// argument n in a0, result in a1
```

```
fact:
    addi sp, sp, -8    # allocate space for 2 words on stack
    sw   ra, 4(sp)     # save return address
    sw   a0, 0(sp)     # save n
    addi t0, x0, 2     # t0 = 2
    bge  a0, t0, L1    # if n >= 2 go to L1 (recursive case)
    addi a0, x0, 1     # set return value to 1
    addi sp, sp, 8     # pop stack (no need to restore values)
    ret                # return (base case)
L1:
    addi a0, a0, -1    # n = n-1
    jal  ra, fact      # make recursive call
    addi t1, a0, 0     # move result from recursive call to t1
    lw   a0, 0(sp)     # restore caller's n
    lw   ra, 4(sp)     # restore caller's return address
    addi sp, sp, 8     # pop stack
    mul  a0, a0, t1    # set return value
    ret                # return
```