CSC 411

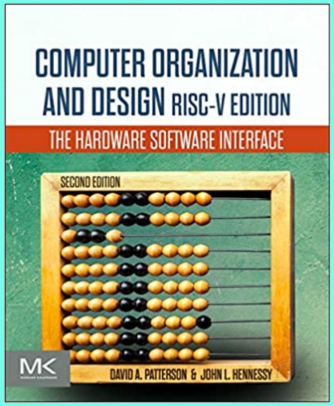
Computer Organization (Spring 2024)
Lecture 14: RISC-V procedures

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



Loops

- Conditional branches are key to writing loops in RISC-V
  - multiple ways of writing a loop

```
# assume x1 holds the value 4 and x2
# is zero, what is the value of x2?
loop:
    bge  x0, x1, done
    addi x1, x1, -1
    addi x2, x2, 2
    beq  x0, x0, loop
done:
```

Procedures

C functions and RISC-V

- C functions
  - each function keeps a <u>local scope</u> separate from <u>global scope</u>
    - local scope doesn't exist in RISC-V, registers are "<u>global</u>" throughout the program — all functions have access to registers, even recursive methods
- Return address
  - need to return to the next instruction after the call, can’t use just a “jump label” instruction as multiple calls may happen from different places
    - treat the <u>return address</u> as an input to the function

---

```c
#include <stdio.h>

int foo(int p, int q) {
    return p + q;
}

int main() {
    int a=1, b=2;
    a = foo(a, a);
    b = foo(b, b);
    printf("%d", a + b);
    return 0;
}
```

```
main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        li      a5,1
        sw      a5,-20(s0)
        li      a5,2
        sw      a5,-24(s0)
        lw      a1,-20(s0)
        lw      a0,-20(s0)
        call    foo(int, int)
        sw      a0,-20(s0)
        lw      a1,-24(s0)
        lw      a0,-24(s0)
        call    foo(int, int)
        sw      a0,-24(s0)
        lw      a4,-20(s0)
        lw      a5,-24(s0)
        add     a5,a4,a5
        mv      a1,a5
        lui     a5,%hi(.LC0)
        addi    a0,a5,%lo(.LC0)
        call    printf
        li      a5,0
        mv      a0,a5
        lw      ra,28(sp)
        lw      s0,24(sp)
        addi    sp,sp,32
        jr      ra
```

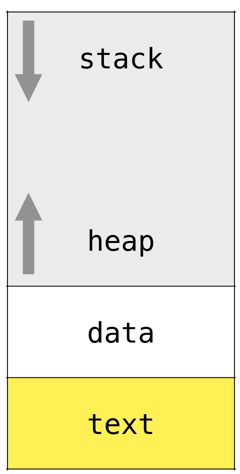
Jumps are not enough!

```
foo(int, int):
        addi    sp,sp,-32
        sw      s0,28(sp)
        addi    s0,sp,32
        sw      a0,-20(s0)
        sw      a1,-24(s0)
        lw      a4,-20(s0)
        lw      a5,-24(s0)
        add     a5,a4,a5
        mv      a0,a5
        lw      s0,28(sp)
        addi    sp,sp,32
        jr      ra
.LC0:
        .string "%d"
```

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

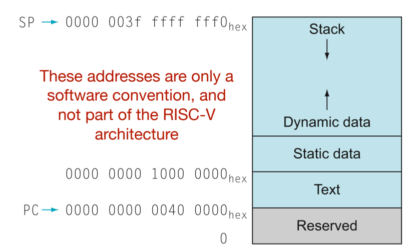
C memory model

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



RISC-V memory model

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



Using registers

- Think about the register file as a scratchpad
  - each procedure uses the scratchpad
  - when a procedure is called, values may have to be saved to **resume work** after returning from the **callee**

caller

```
int main() {
    int a, b, c, d;
    // ...
    a = foo(b, c);
    d = foo(a, a);
    // ...
    return 0;
}
```

callee

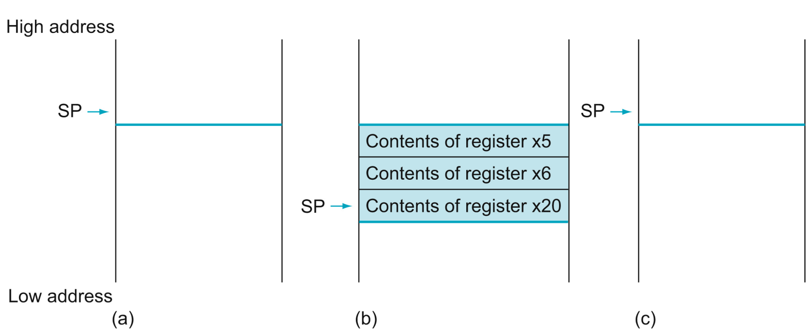
```
int foo(int p, int q) {
    int r = 1;
    for (int i ; i < q ; i += 2) {
        r *= p;
    }
    return r;
}
```

Special registers

- Program Counter (**pc**)
  - keeps track of which line of code will be executed next
  - implemented at a different location of x0-x31 registers
- Stack Pointer (**sp**)
  - the standard calling convention uses register x2 as the stack pointer.
  - points initially to the “base” of the stack and procedures can modify its value accordingly (growing/shrinking the stack)
    - stack grows downward (from high to low addresses)
  - the value of **sp** at the start of a function separates what the function can (equal/lower memory addresses) and cannot modify (higher memory addresses)
    - if a function modifies **sp** internally, it must set **sp** to its original value before returning to the caller

The stack pointer

- 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)



Register usage conventions

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

Register usage 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

- Procedures make use of general purpose registers
- Two options for storing values on registers
  - Callee saves registers
    - assume a caller is using a **saved register**, if callee wants to use the same register, it saves the register value when entering the function and restores it just before returning
  - Caller saves registers
    - assume caller is using a **temporary register**, as callee may freely modify temporaries, caller saves the register value before calling the function and restores it after the call

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

Procedure calling convention

- Place necessary arguments in registers **a0 - a7**
  - if additional space is need, can also use the stack
- Transfer control to procedure
- Acquire storage for procedure
  - save registers if necessary
  - can freely use temporary registers
- Perform procedure's operations
- Place return value in register for caller
- Restore any registers
- Return to place of call
  - address in register **ra**

Jump instructions

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

```
jal rd, imm
```

- Jump and link register
  - instead of using a label (pc-relative addressing), it jumps to "rs1+imm" and saves the 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
```

Examples

Non-leaf procedures

Non-leaf procedures

- Procedures that call other procedures
  - e.g., recursion
- **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

```c
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
```