

CSC 411

Computer Organization (Spring 2023)
Lecture 11: Procedures

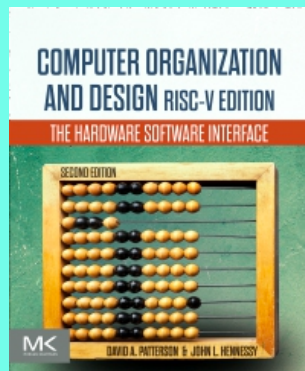
Prof. Marco Alvarez, University of Rhode Island

```
addi x12, x0, 0x000002CF
sw x12, 0(x13)
lb x11, 1(x13)
```

```
addi x12, x0, 0x000002CF
sw x12, 0(x13)
lb x11, 0(x13)
```

Disclaimer

Some of the following slides are adapted from:
Computer Organization and Design (Patterson and Hennessy)
The Hardware/Software Interface



Procedure calling (functions)

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

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

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

What information we must keep track of?

```

#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

```

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"

Jumps are not enough!

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

RISC-V memory allocation

Stack/Heap

- space for the run-time stack (local procedures)
- dynamically allocated data

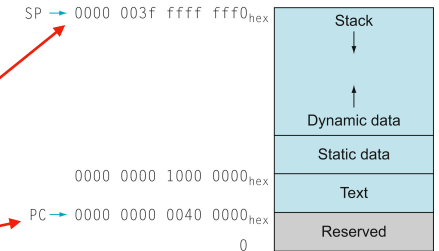
Static data

- global variables

Text

- instructions

Addresses are only a software convention



Procedure calling steps

- Place **arguments** in **registers x10 to x17**
 - so the function can access them
- Transfer control to function
- Acquire storage needed by function
 - save registers if necessary
- Perform function's operations
- Place **return value** in register for caller
- Return to place of call
 - address in **register x1**

Register usage

Registers to pass parameters and return values

- a0 - a7 (x10 - x17) — eight **argument registers**
- use memory (stack) if more space is needed
- a0 - a1 (x10 - x11) — two **return-value registers**

One return address register

- ra (x1)

Saved registers

- s0 - s1 (x8 - x9) and s2 - s11 (x18 - x27)

Calling convention

- **Caller** places the arguments in **argument registers**
- **Caller** allocates stack space by moving the **stack pointer** down
- **Callee** saves **saved registers** if needed
 - callee is allowed to freely write over **temporary registers**
- **Caller** puts return address in the **ra register**
 - JAL instruction, returns from the function call
- ... then the function is executed

Register usage

- x5 – x7, x28 – x31
 - temporary registers, **not preserved by the callee**
- x8 – x9, x18 – x27
 - saved registers, **callee saves and restores them**

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

Register usage

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5-7	Temporaries	no
x8-x9	8-9	Saved	yes
x10-x17	10-17	Arguments/results	no
x18-x27	18-27	Saved	yes
x28-x31	28-31	Temporaries	no

Procedure call instructions

- Procedure call: **jump and link**

jal x1, <label>

- address of following instruction put in x1
- jumps to target address

- Procedure return: **jump and link register**

jalr x0, 0(x1)

- like jal but jumps to 0 + address in x1
- use x0 as rd (cannot be changed)
- can also be used for computed jumps (case/switch)

Pseudoinstructions

▸ Pseudoinstructions are NOT machine instructions

- assembly instructions that help programmers
- translated to machine instructions by the assembler

▸ Examples

- mv (copy value from one register to another)
- li (load immediate, set the value of a register to a constant)
- j, jr
- jal, jalr, ret
- nop (no operation)

nop	addi x0, x0, 0	No operation
li rd, immediate	<i>Myriad sequences</i>	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if ≠ zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero
fmv.s rd, rs	fsgnj.s rd, rs, rs	Copy single-precision register
fabs.s rd, rs	fsgnjx.s rd, rs, rs	Single-precision absolute value
fneg.s rd, rs	fsgnjn.s rd, rs, rs	Single-precision negate
fmv.d rd, rs	fsgnj.d rd, rs, rs	Copy double-precision register
fabs.d rd, rs	fsgnjx.d rd, rs, rs	Double-precision absolute value
fneg.d rd, rs	fsgnjn.d rd, rs, rs	Double-precision negate
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if ≠ zero
blez rs, offset	bge x0, rs, offset	Branch if ≤ zero
bgez rs, offset	bge rs, x0, offset	Branch if ≥ zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if ≤
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if ≤, unsigned

<https://web.eecs.utk.edu/~smarz1/courses/ece356/notes/assembly>

Leaf procedure example

▸ Need a place to save old values

- use the stack (LIFO — last-in-first-out)
- basically using push/pop
- stack is in memory (use the sp — stack pointer)
- grow stack down from high to low addresses

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

pseudoinstruction	Base Instruction	Meaning
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, 0(rs)	Jump register
jalr rs	jalr x1, 0(rs)	Jump and link register
ret	jalr x0, 0(x1)	Return from subroutine
call offset	auipc x1, offset[31:12] + offset[11] jalr x1, offset[11:0](x1)	Call far-away subroutine
tail offset	auipc x6, offset[31:12] + offset[11] jalr x0, offset[11:0](x6)	Tail call far-away subroutine

<https://web.eecs.utk.edu/~smarz1/courses/ece356/notes/assembly>

Leaf procedure example

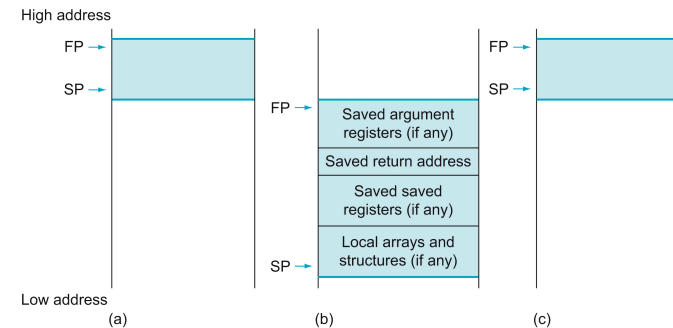
```
int leaf_example(int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}

// arguments g, ..., j in x10, ..., x13
// f in x20
// temporaries x5, x6
// need to save x5, x6, x20 on stack

leaf_example:
    addi sp, sp, -12    # save register values on stack
    sw   x5, 8(sp)
    sw   x6, 4(sp)
    sw   x20, 0(sp)
    add  x5, x10, x11.  # perform operations
    add  x6, x12, x13
    sub  x20, x5, x6
    addi x10, x20, 0     # copy result to return register
    lw   x20, 0(sp)     # restore register values from stack
    lw   x6, 8(sp)
    lw   x5, 16(sp)
    addi sp, sp, 12
    jalr x0, 0(x1)      # return to caller (can use jr x1 or jr ra)
```

Local data on the Stack

- Local data allocated by **callee**
- Procedure frame (activation record)
 - used by some compilers to manage stack storage



Sum array example

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

// arguments p in x10, n in x11
// return value in x10
```

What does this code do?

```
int foo(char *s) {
    int c = 0;
    while (*s++) {
        ++c;
    }
    return c;
}
```

String copy example

```
void strcpy (char x[], char y[]) {
    int i = 0;
    while ((x[i]=y[i]) != '\0') {
        i += 1;
    }
}
```

RISC-V code

```
// addresses of x, y in x10, x11
// i in x19
void strcpy (char x[], char y[]) {
    int i = 0;
    while ((x[i]=y[i]) != '\0')
        i += 1;
}

strcpy:
    addi sp, sp, -8 // adjust stack for 1 doubleword
    sd   x19, 0(sp) // push x19
    add  x19, x0, x0 // i=0
L1:
    add  x5, x19, x11 // x5 = addr of y[i]
    lbu  x6, 0(x5)    // x6 = y[i]
    add  x7, x19, x10 // x7 = addr of x[i]
    sb   x6, 0(x7)    // x[i] = y[i]
    beq  x6, x0, L2    // if y[i] == 0 then exit
    addi x19, x19, 1   // i = i + 1
    jal  x0, L1        // next iteration of loop
L2:
    ld   x19, 0(sp)    // restore saved x19
    addi sp, sp, 8     // pop 1 doubleword from stack
    jalr x0, 0(x1)     // and return
```

Non-leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
 - its return address
 - any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-leaf procedure example

```
int fact (int n) {
    if (n < 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
// argument n in x10, result in x10

fact:
    addi sp, sp, -8 # save register values on stack
    sw   x1, 4(sp)  # save return address
    sw   x10, 0(sp) # save n
    addi x5, x10, -1 # x5 = n-1
    bge  x5, x0, L1  # if n >= 1 go to L1
    addi x10, x0, 1  # set return value to 1
    addi sp, sp, 8   # pop stack (no need to restore values)
    jalr x0, 0(x1)   # return (base case)
L1:
    addi x10, x10, -1 # n = n-1
    jal  x1, fact     # make recursive call
    addi x6, x10, 0   # move result from recursive call to x6
    lw   x10, 0(sp)   # restore caller's n
    lw   x1, 4(sp)    # restore caller's return address
    addi sp, sp, 8     # pop stack
    mul  x10, x10, x6  # set return value
    jalr x0, 0(x1)    # return
```

