CHAPTER XV

PROCEDURE CALLS AND SUBROUTINES

PROCEDURE CALLS

INTRODUCTION

- Branches and jumps are important program control constructs, but another important extension of program control are **procedure calls**, often referred to as **subroutines**.
- Three basic steps form of a subroutine call
  - Program control is changed
    - **from** the **current routine**
    - **to** the beginning of the **subroutine** code.
  - Subroutine code is executed.
  - Program control is changed
    - **from** end of **subroutine**
    - **to** the calling routing immediately* **after subroutine call instruction**.

* Note: Not quite accurate for the MIPS architecture.

PROCEDURE CALLS

PROGRAM FLOW

- We can illustrate how subroutine calls change program flow as follows.

Main routine instructions

PC	n
PC+4	n + 1
PC+8	n + 2
PC+12	subroutine call (**Sample 1**)
PC+16	n + 4 *
PC+20	n + 5
	...
PC+4*m	n + m
PC+4*(m+1)	n + m + 1
PC+4*(m+2)	subroutine call (**Sample 2**)
PC+4*(m+3)	n + m + 3 *
	...

Sample 1 subroutine code

p
p + 1
...
p + k
return from subroutine

Sample 2 subroutine code

q
q + 1
...
q + r
return from subroutine

* Note: Not quite accurate for the MIPS architecture.

MACHINE STATE

SAVING MACHINE STATE

- How can program flow be changed to a subroutine?
  - **PC = address of 1st instruction of subroutine**
- And then returned from a subroutine call?
  - **PC = address of instruction after subroutine call instruction**
- The idea is to **save the state of the machine**.
- In the most basic microprocessor, saving the state means to **save** the **PC** in a **known location**!
- Some microprocessors also save other registers during a procedure call.
- **MIPS** only saves the **PC** and then restores the **PC** after the subroutine.

MACHINE STATE

SAVING STATE TO $RA

- For MIPS, the primary location for saving the **PC** is in **$31/$ra**.
- MIPS uses the instruction **jal** <**imm**> (jump and link)
  - **jal** is **J-format** type instruction.
  - **Stores** the **return address** in **$ra**, i.e. **$ra = PC + 4***.
  - **Performs jump** such as with the **j** instruction.
- At the end of the subroutine, the instruction **jr $ra** is executed to return to calling routing.
  - This causes the contents of **$ra** to be put into **PC**
    - i.e. **PC = $ra** which after the original **jal** instruction is **PC = PC + 4***.

* Note: Not quite accurate for the MIPS architecture.

MACHINE STATE

EXAMPLE PROCEDURE CALL

- Below is an example piece of pseudo-code that has been translated in assembly with a main routine and a square root subroutine.

Pseudo-Code

```
b = 6;

a = sqrt(b);

a = a + b;
```

MIPS Assembly

main routine
(use $s0 for a, $s1 for b)

```
lwi $s1, 0x06
move $a0, $s1
jal sqrt
move $s0, $v0
add $s0, $s0, $s1
...
```

square root subroutine
(argument in $a0, result in $v0)

```
sqrt:   ...
        ...
        jr $ra
```

MACHINE STATE

SAVING STATE TO REGISTER

- Another approach to saving the **PC** is the in the form **jalr $<dest>, $<src>** (jump and link register) instruction.
  - **jalr** is roughly an **R-format** type instruction.
  - **Stores** the **return address** in **$<dest>**, i.e. **$5 = PC + 4***.
  - **Performs jump** such as with the **jr <$src>** instruction.
- At the end of the subroutine, to return from the subroutine the following can be executed.
  - **jr $<dest>** (i.e. **jr $5**)
- Another option for returning from a subroutine is to execute
  - **jalr $0, $5**,
  - or even **jalr $<new dest>, $5**.

* Note: Not quite accurate for the MIPS architecture.

MACHINE STATE

EXAMPLE PROCEDURE CALL

- Another example where jalr is used and the subroutine is completely given.

Pseudo-Code	MIPS Assembly: main routine (use $s0 for a, $s1 for b)
b = 6;	lwi $s1, 0x06
	move $a0, $s1
a = decr(b);	jalr $s7, decr
	move $s0, $v0
a = a + b;	add $s0, $s0, $s1
	...

decrement subroutine
(argument in $a0, result in $v0)

```
decr:  subi $v0,$a0,1
       jr $s7
```

MACHINE STATE

EXAMPLE PROCEDURE CALL

- A more complicated example could be as follows.

Pseudo-Code

```
a = 6;
b = 4;
c = 10;
d = func(b,c,a);
...

int func(x,y,z)
    return x+y-z;
```

MIPS Assembly

Main routine
(use \$s0-3 for a,b,c,d)

```
lwi $s0, 0x06
lwi $s1, 0x04
lwi $s2, 0x0A
move $a0, $s1
move $a1, $s2
move $a2, $s0
jal func
move $s3, $v0
...
```

func subroutine
(arguments in \$a0-2,
result in \$v0)

```
func:  sub $v0,$a1,$a2
       add $v0,$a0,$v0
       jr $ra
```

MACHINE STATE

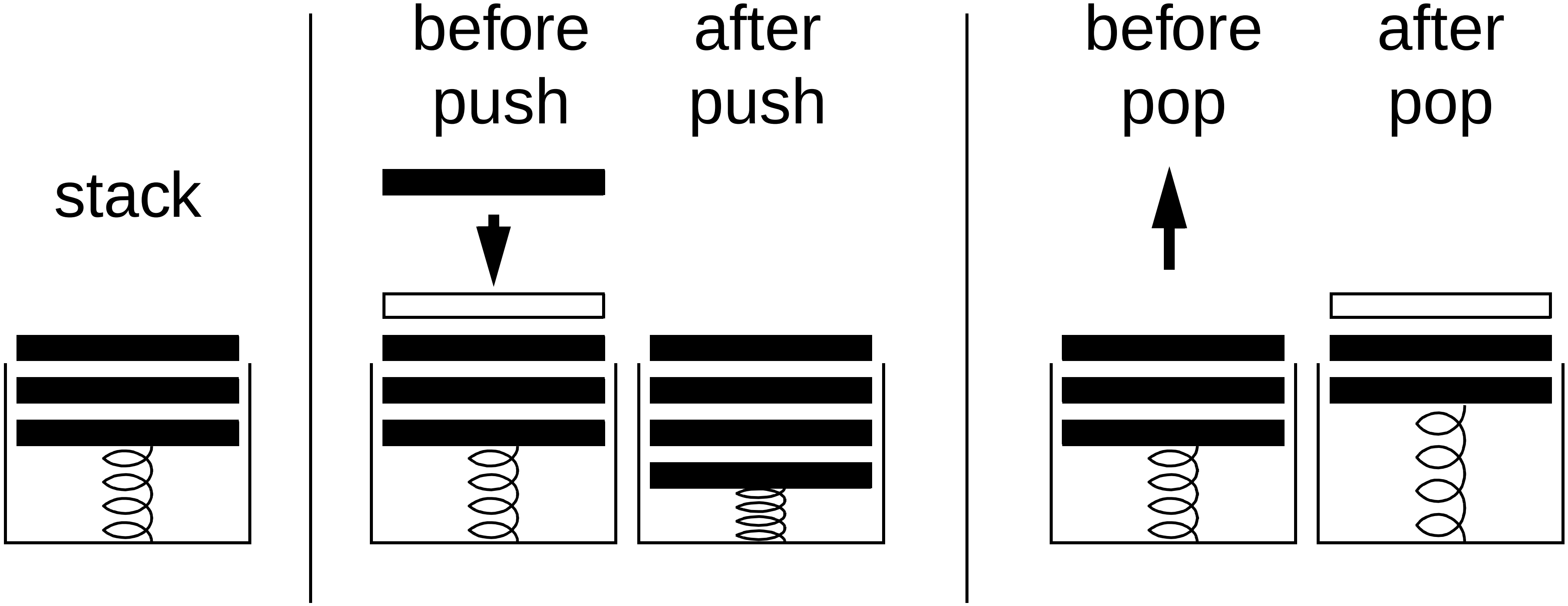
PROBLEMS

- Two problems exist with the subroutine approach discussed so far.
- **Problem 1:**
  - What if we want to call a subroutine within a subroutine?
    - Only one **$ra**, so only one return address is stored with **jal**.
    - If we call a nested subroutine, the return address in **$ra** is lost.
- **Problem 2:**
  - What if we need many temporary registers within the subroutine?
    - We don't want to lose the contents of registers that the calling function might still need!
- **Solution: Stacks**

STACKS

PUSHING AND POPPING

- A stack is a **LIFO** (Last-In, First-Out) data structure.
  - Consider the example of a stack of plates at a cafeteria.



  - A plate can be added to the top of the stack, called a **push**.
  - A plate can be removed from the top of the stack, called a **pop**.

STACKS

STACK OPERATION

- Which way should a **stack grow** in memory?
  - It is customary for a stack to **grow from larger** memory addresses **to smaller** memory addresses.
- Use a stack pointer (**SP**) to point to top of stack. This is **$29/$sp** on MIPS.
- **push**: To place a new item onto the stack
  - first decrement **SP**,
  - then store item at the new location pointed to by **SP**.
- **pop**: To retrieve an item from the stack
  - first copy item pointed to by **SP** into desired destination,
  - then increment **SP**.
- **Many processors deviate slightly from this, but with the same idea.**

STACKS

MEMORY MODEL

- Following the previous slide, we can think of our memory model as follows if **SP = 0x00FFFFF4** and the bottom of the stack is **0x01000000**.

push: SP = SP - 4

pop: SP = SP + 4

	0x00FFFFE0	
	0x00FFFFE4	
	0x00FFFFE8	
	0x00FFFFEC	
	0x00FFFFF0	
SP →	0x00FFFFF4	0x77777777
	0x00FFFFF8	0x01234567
	0x00FFFFFC	0x76543210
	0x01000000	0x45553323

- We can see that the stack grows from larger address to smaller address.

STACKS

PUSH AND POP ON MIPS

- The following instructions perform a **push** of **R15** onto the stack.

```
sub $sp, 0x04
sw $15, $sp
```

- The following instructions perform a **pop** from the stack into **R15**.

```
lw $15, $sp
add $sp, 0x04
```

- Many processors actually have the instructions **push** and **pop**, but MIPS removes these to have fewer opcodes (i.e. RISC).

STACKS

PUSH ON MIPS

- A **push** on MIPS is performed and illustrated as follows.

Given that
R15=0x77777777

Push of R15 onto stack

```
sub $sp, 0x04
sw $15, $sp
```

Before push: R15=0x77777777

	0x00FFFFF0	
	0x00FFFFF4	
SP →	0x00FFFFF8	0x01234567
	0x00FFFFFC	0x76543210
	0x01000000	0x45553323

After push: R15=0x77777777

	0x00FFFFF0	
SP →	0x00FFFFF4	0x77777777
	0x00FFFFF8	0x01234567
	0x00FFFFFC	0x76543210
	0x01000000	0x45553323

STACKS

POP ON MIPS

- A **pop** on MIPS is performed and illustrated as follows.

Pop from stack to R15

```
lw $15, $sp
add $sp, 0x04
```

Now R15=0x01234567

Before pop: R15=0x????????

	0x00FFFFF0	
	0x00FFFFF4	
SP →	0x00FFFFF8	0x01234567
	0x00FFFFFC	0x76543210
	0x01000000	0x45553323

After pop: R15=0x01234567

	0x00FFFFF0	
	0x00FFFFF4	
	0x00FFFFF8	0x01234567
SP →	0x00FFFFFC	0x76543210
	0x01000000	0x45553323

STACKS

NESTED PROCEDURE CALLS

- Procedure calls can now be nested since **$ra** can be saved on the stack.

Main routine

```
...
...
...
jal func1
...
...
...
```

func1

```
sub $sp, 0x04
sw $ra, $sp
...
jal func2
...
lw $ra, $sp
add $sp, 0x04
jr $ra
```

Save return address (push): sub $sp, 0x04 / sw $ra, $sp

Restore return address (pop): lw $ra, $sp / add $sp, 0x04

func2

```
...
...
jr $ra
```

STACKS

NESTED PROCEDURE CALLS

- This example can be thought of in a higher level language as

```
complex Z addcomplex(complex X, complex Y) {
    Z.real = X.real + Y.real;
    Z.imaginary = X.imaginary + Y.imaginary;
    return Z;
}

complex W funcAadd2B(complex U, complex V) {
    W = addcomplex(U, V);
    W = addcomplex(W, V);
    return W;
}

main {
    complex A = 5 + i6, B = 2 + i7, C;
    C = funcAadd2B(A, B);
}
```

STACKS

EXAMPLE NESTED CALL

- Say that we want to write a function **funcAadd2B** that calculates **A+2B** where **A** and **B** are complex numbers.
  - (**$a0**,**$a1**) contains (**real**,**imaginary**) part of **A**.
  - (**$a2**,**$a3**) contains (**real**,**imaginary**) part of **B**.
  - (**$v0**,**$v1**) contains (**real**,**imaginary**) part of answer.
- To make life easier, also design function **addcomplex** that adds two complex numbers **X** and **Y**.
  - (**$a0**,**$a1**) contains (**real**,**imaginary**) part of **X**.
  - (**$a2**,**$a3**) contains (**real**,**imaginary**) part of **Y**.
  - (**$v0**,**$v1**) contains (**real**,**imaginary**) part of answer.

STACKS

EXAMPLE NESTED CALL

- This example could be implemented as follows in assembly.

Main routine

```
...
...
...
jal funcAadd2B
...
...
...
```

funcAadd2B

```
sub $sp, 0x04
sw $ra, $sp
jal addcomplex
move $a0,$v0
move $a1,$v1
jal addcomplex
lw $ra, $sp
add $sp, 0x04
jr $ra
```

addcomplex

```
add $v0,$a0,$a2
add $v1,$a1,$a3
jr $ra
```