COMPSCI 313 S2 2018
Computer Organization

8 MIPS Procedure Calls

Agenda & Reading

- MIPS procedure calls
- MIPS addressing

Procedure Calling

- Steps required
  - Place parameters in registers
  - Transfer control to procedure
  - Acquire storage for procedure
  - Perform procedure's operations
  - Place result in register for caller
  - Return to place of call

Register Usage

- \$a0 – \$a3: arguments (reg's 4 – 7)
- \$v0, \$v1: result values (reg's 2 and 3)
- \$t0 – \$t9: temporaries
  - Can be overwritten by callee
- \$s0 – \$s7: saved
  - Must be saved/restored by callee
- \$gp: global pointer for static data (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

```
jal     ProcedureLabel
```

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

```
jr      $ra
```

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Leaf Procedure Example

- C code:

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

- Arguments g, …, j in $a0, …, $a3
- f in $s0 (hence, need to save $s0 on stack)
- Result in $v0

Leaf Procedure Example

- MIPS code:

```
leaf_example:
        addi    $sp, $sp, -4
        sw      $s0, 0($sp)              #Save $s0 on stack
        add     $t0, $a0, $a1
        add     $t1, $a2, $a3            #Procedure body
        sub     $s0, $t0, $t1
        add     $v0, $s0, $zero          #Result
        lw      $s0, 0($sp)
        addi    $sp, $sp, 4     #Restore $s0
        jr   $ra                #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 Procedures

- C code:

```
int fact (int n) {
        if (n < 1)
                return f;
        else
                return n * fact(n - 1);
}
```

  - Argument n in $a0
  - Result in $v0

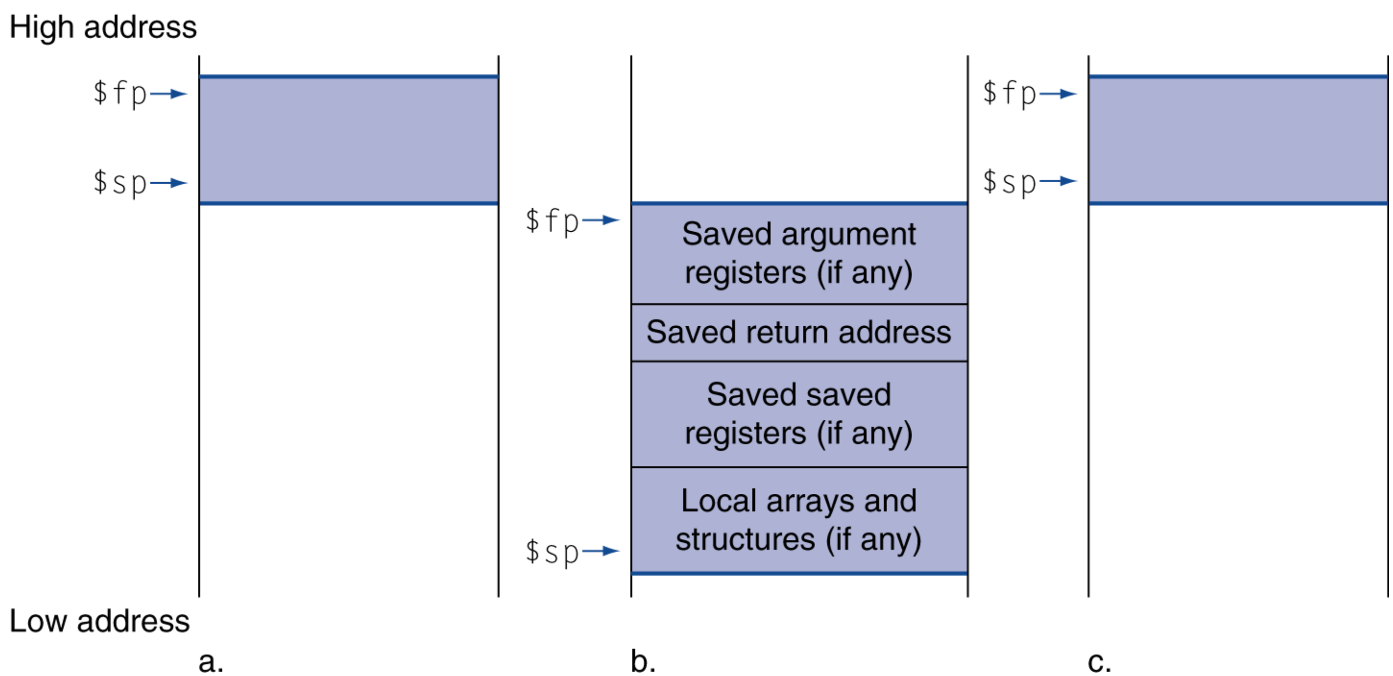
Non-Leaf Procedure Example

- MIPS code:

```
fact:
        addi $sp, $sp, -8   # adjust stack for 2 items
        sw $ra, 4($sp)      # save return address
        sw $a0, 0($sp)      # save argument
        slti $t0, $a0, 1    # test for n < 1
        beq $t0, $zero, L1
        addi $v0, $zero, 1          # if so, result is 1
        addi $sp, $sp, 8    #       pop 2 items from stack
        jr $ra              #       and return
L1:     addi $a0, $a0, -1   # else decrement n
        jal fact            # recursive call
        lw $a0, 0($sp)      # restore original n
        lw $ra, 4($sp)      #       and return address
        addi $sp, $sp, 8    # pop 2 items from stack
        mul $v0, $a0, $v0           # multiply to get result
        jr $ra              # and return
```

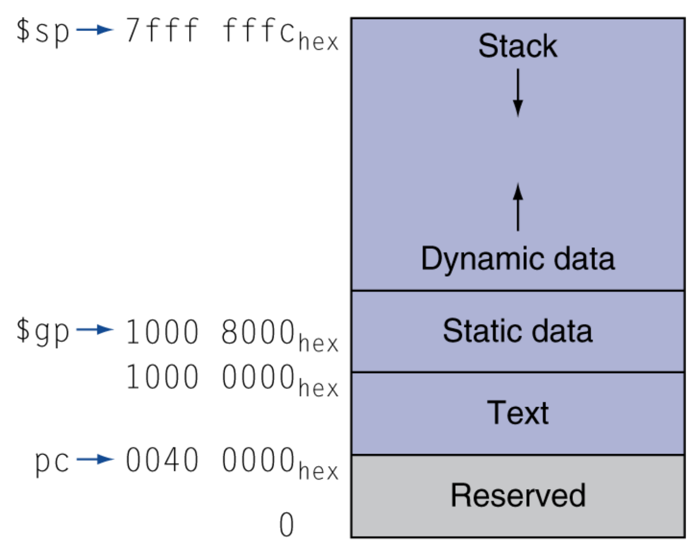
Local Data on the Stack

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage



Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte Operations

- Could use bitwise operations
- MIPS byte load/store
  - String processing is a common case
  - Sign extend to 32 bits in rt

lb rt, offset(rs)
$\#R[t] \text{ <-- } (M[Addr]_7)^{24}::M1[Addr]$

  - Zero extend to 32 bits in rt

lbu rt, offset(rs)
$\#R[t] \text{ <-- } (0)^{24}::M1[Addr]$

  - Store just rightmost byte

sb rt, offset(rs)
$M[Addr] \text{ <-- } R[t]_{7\text{-}0}$

Half-word Operations

- Could use bitwise operations
- MIPS half-word load/store
  - String processing is a common case
  - Sign extend to 32 bits in rt

```
lh      rt, offset(rs)
```

$\#R[t] \text{ <-- } (M[Addr]_{15})^{16}::M1[Addr]$

  - Zero extend to 32 bits in rt

```
lhu     rt, offset(rs)
```

$\#R[t] \text{ <-- } (0)^{16}::M1[Addr]$

  - Store just rightmost half-word

```
sh      rt, offset(rs)
```

$M[Addr] \text{ <-- } R[t]_{15\text{-}0}$

String Copy Example

- C code:
  - Null-terminated string

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

  - Addresses of x, y in \$a0, \$a1
  - i in \$s0

32-bit Constants

- MIPS code:

```
strcpy:
        addi $sp, $sp, -4           # adjust stack for 1 item
        sw $s0, 0($sp)              # save $s0
        add $s0, $zero, $zero       # i = 0
L1:     add $t1, $s0, $a1           # addr of y[i] in $t1
        lbu $t2, 0($t1)             # $t2 = y[i]
        add $t3, $s0, $a0           # addr of x[i] in $t3
        sb $t2, 0($t3)              # x[i] = y[i]
        beq $t2, $zero, L2          # exit loop if y[i] == 0
        addi $s0, $s0, 1            # i = i + 1
        j L1                        # next iteration of loop
L2:     lw $s0, 0($sp)              # restore saved $s0
        addi $sp, $sp, 4            # pop 1 item from stack
        jr $ra                      # and return
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant (by 2 steps)

lui rt, constant

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

`lui $s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

- Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)

Exercise 8.1

- Q1.1. Suppose the program counter (PC) is set to 0x2000 0000.
- Is it possible to use the jump (j) MIPS assembly instruction to set the PC to the address as 0x4000 0000?
- Is it possible to use the branch-on-equal (beq) MIPS assembly instruction to set the PC to this same address?

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

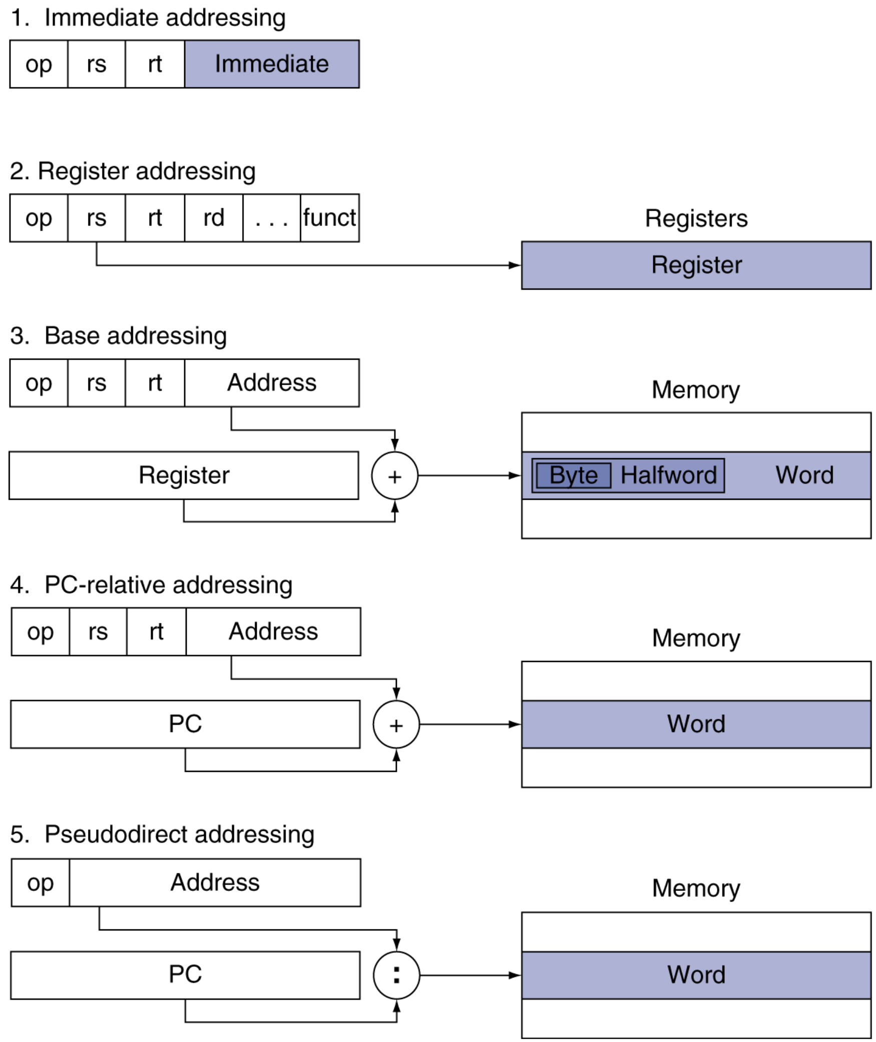
`Loop: sll  $t1, $s3, 2`	80000	0	0	19	9	4	0
`      add  $t1, $t1, $s6`	80004	0	9	22	9	0	32
`      lw   $t0, 0($t1)`	80008	35	9	8	0		
`      bne  $t0, $s5, Exit`	80012	5	8	21	2		
`      addi $s3, $s3, 1`	80016	8	19	19	1		
`      j    Loop`	80020	2	20000				
`Exit: …`	80024						

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
        ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

Addressing Mode Summary



Exercise 8.2

- Q2.1. Assume $t0 holds the value 0x00101000. What is the value of $t2 after the following instructions?
    - slt $t2, $0, $t0
    - bne $t2, $0, ELSE
    - j DONE
  - ELSE:
    - addi $t2, $t2, 2
  - DONE:

Summary

- On completion of this module, you are able to
  - Write simple programs with procedure calls
  - Write simple programs with MIPS addressing

- References:
  - Textbook - Computer Organization and Design - The Hardware Software Interface
    - Chapter 2