Chapter 2

Instructions: Language of the Computer

Fall 2018
Soontae Kim
School of Computing
KAIST

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments
  - Temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

  - Argument n in $a0
  - Result in $v0

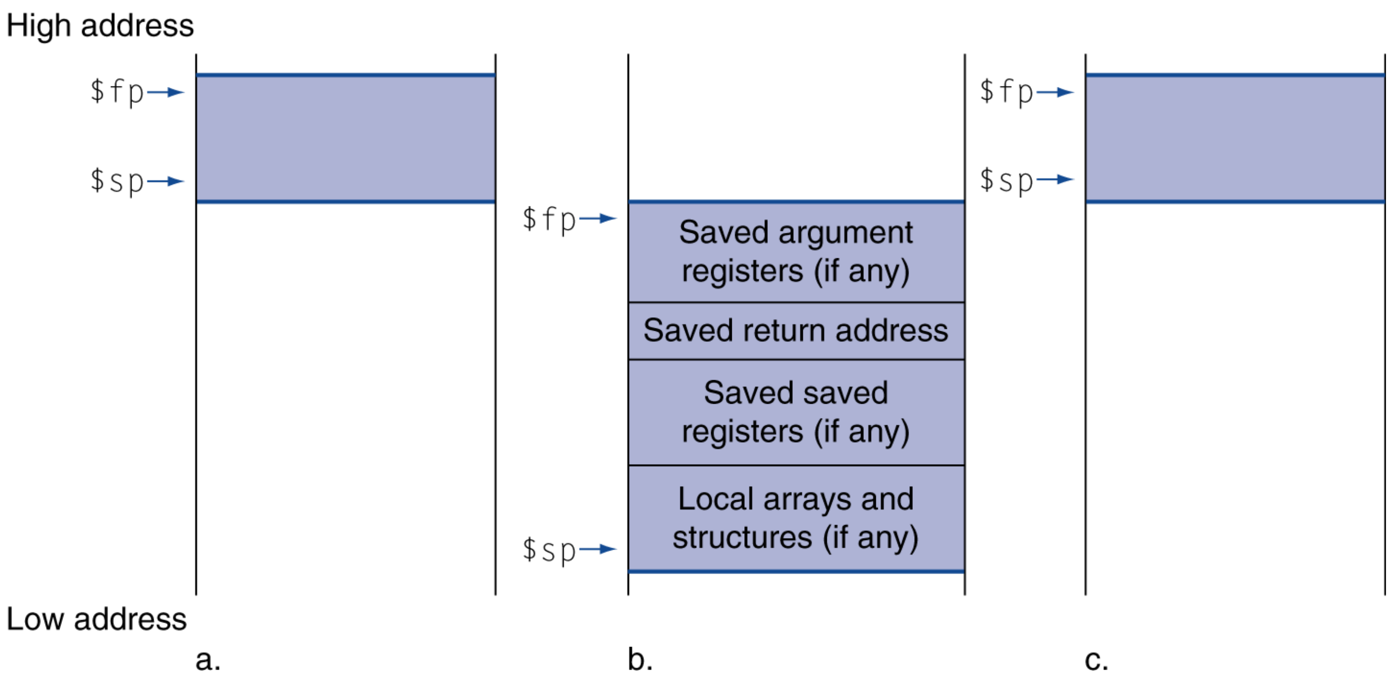
- n! = n * (n-1)!, 0!=1

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
```

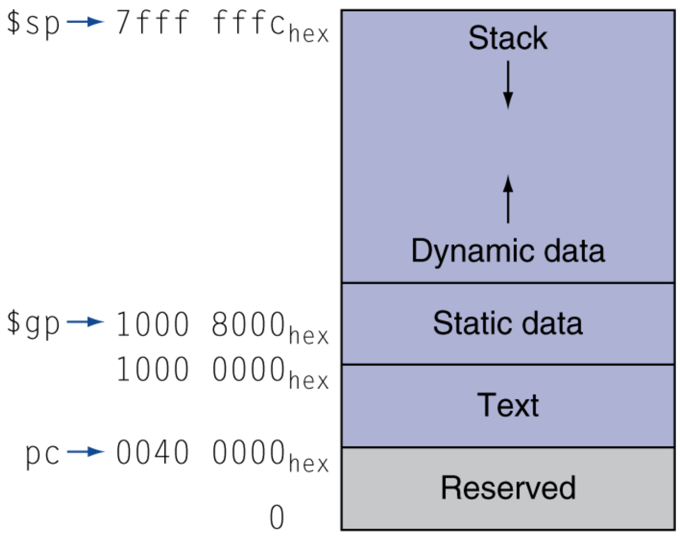
Local Data on the Stack



- 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



Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

FIGURE 2.14 MIPS register conventions. Register 1, called $at, is reserved for the assembler (see Section 2.11), and registers 26–27, called $k0–$k1, are reserved for the operating system. This information is also found in Column 2 of the MIPS Reference Data Card at the front of this book.

Program Counter (PC): the register containing the address of the instruction being executed

Byte/Halfword Operations

- MIPS byte/halfword load/store
  - String processing is a common case

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

  - Sign extend to 32 bits in rt

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

  - Zero extend to 32 bits in rt

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

  - Store just rightmost byte/halfword

String Copy Example

- C code (naïve):
  - 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

String Copy Example

- 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

  `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 0011 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 pointing to next instruction

Jump Addressing

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

op	address
6 bits	26 bits

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

Target Addressing Example

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

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Address						
80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
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