LECT - 6

Instructions-Language of Computer

Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. 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

```
leaf_example:
addi $sp, $sp, –12      # adjust stack to make room for 3 items
sw $t1, 8($sp)          # save register $t1 for use afterwards
sw $t0, 4($sp)          # save register $t0 for use afterwards
sw $s0, 0($sp)          # save register $s0 for use afterwards
add $t0,$a0,$a1         # register $t0 contains g + h
add $t1,$a2,$a3         # register $t1 contains i + j
sub $s0,$t0,$t1         # f = $t0 – $t1, which is (g + h)–(i + j)
add $v0,$s0,$zero       # returns f ($v0 = $s0 + 0)
lw $s0, 0($sp)          # restore register $s0 for caller
lw $t0, 4($sp)          # restore register $t0 for caller
lw $t1, 8($sp)          # restore register $t1 for caller
addi $sp,$sp,12         # adjust stack to delete 3 items
jr $ra                  # jump back to calling routine
```

Data Saving in stack

- To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, MIPS soft ware separates 18 of the registers into two groups:

1. $t0–$t9: temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
2. $s0–$s7: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

Local Data on the Stack

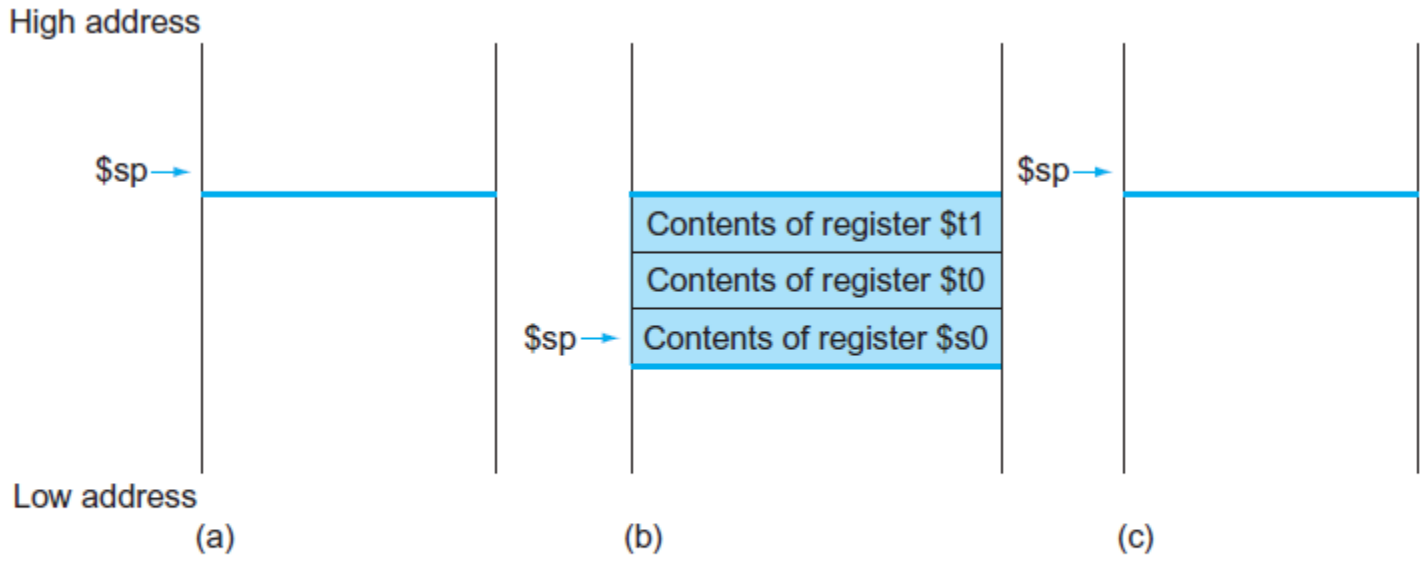

FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Leaf Procedure Example-Another Way

- MIPS code:

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

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

- 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

```
fact:
addi $sp, $sp, –8        # adjust stack for 2 items
sw $ra, 4($sp)           # save the return address
sw $a0, 0($sp)           # save the argument n
slti $t0,$a0,1           # test for n < 1
beq $t0,$zero,L1         # if n >= 1, go to L1
addi $v0,$zero,1         # return 1
addi $sp,$sp,8           # pop 2 items off stack
jr $ra                   # return to caller
L1: addi $a0,$a0,–1      # n >= 1: argument gets (n – 1)
jal fact                 # call fact with (n –1)
lw $a0, 0($sp)           # return from jal: restore argument n
lw $ra, 4($sp)           # restore the return address
addi $sp, $sp, 8         # adjust stack pointer to pop 2 items
mul $v0,$a0,$v0          # return n * fact (n – 1)
jr $ra                   # return to the caller
```

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
```

SwapExample

- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

MIPS Registers

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

Decoding Machine Code

What is the assembly language statement corresponding to this machine instruction?

```
00af8020hex
```

The first step in converting hexadecimal to binary is to find the op fields:

```
(Bits: 31 28 26                          5   2 0)
       0000 0000 1010 1111 1000 0000 0010 0000
```

We look at the op field to determine the operation. Referring to Figure 2.19, when bits 31–29 are 000 and bits 28–26 are 000, it is an R-format instruction. Let's reformat the binary instruction into R-format fields, listed in Figure 2.20:

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

The bottom portion of Figure 2.19 determines the operation of an R-format instruction. In this case, bits 5–3 are 100 and bits 2–0 are 000, which means this binary pattern represents an `add` instruction.

We decode the rest of the instruction by looking at the field values. The decimal values are 5 for the rs field, 15 for rt, and 16 for rd (shamt is unused). Figure 2.14 shows that these numbers represent registers `$a1`, `$t7`, and `$s0`. Now we can reveal the assembly instruction:

```
add $s0,$a1,$t7
```