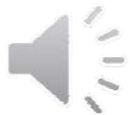# Chapter 2

## Instructions: Language of the Computer

# **Conditional Operations**

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1
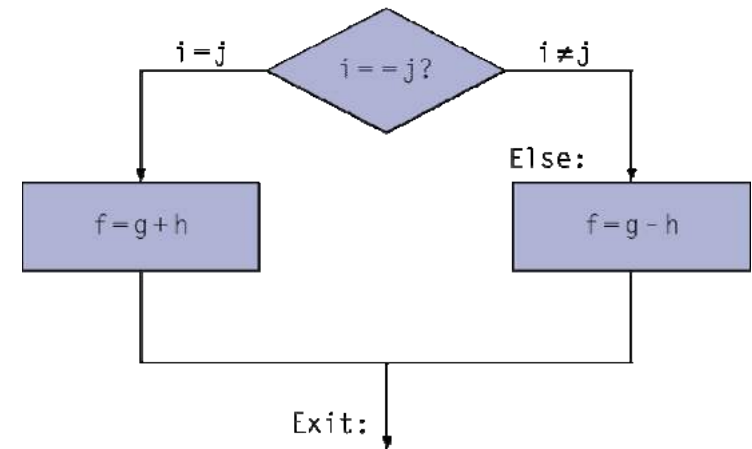
# Compiling If Statements

- C code:

    ```
    if (i==j) f = g+h;
    else f = g-h;
    ```

    - f, g, … in $s0, $s1, …



- Compiled MIPS code:

    ```
              bne $s3, $s4, Else
              add $s0, $s1, $s2
              j    Exit
    Else: sub $s0, $s1, $s2
    Exit: …
    ```

Assembler calculates addresses

# Compiling Loop Statements

- C code:

  while (save[i] == k) i += 1;

  - i in $s3, k in $s5, address of save in $s6

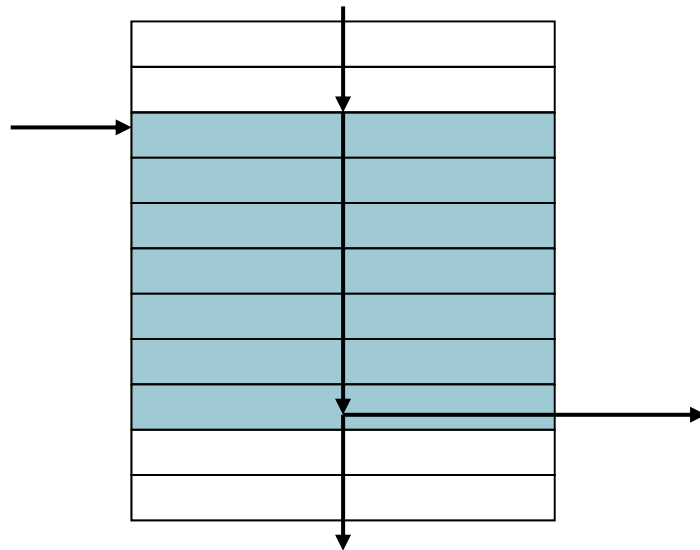- Compiled MIPS code:

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

# Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)

- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

# More Conditional Operations

- Set result to 1 if a condition is true
    - Otherwise, set to 0
- `slt rd, rs, rt`
    - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
    - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

# Branch Instruction Design

- Why not blt, bge, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

# Signed vs. Unsigned

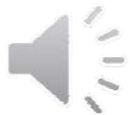- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt   $t0,  $s0,  $s1   # signed`
    - $-1 < +1 \Rightarrow$ $t0 = 1
  - `sltu $t0,  $s0,  $s1   # unsigned`
    - $+4,294,967,295 > +1 \Rightarrow$ $t0 = 0

# Chapter 2

**Instructions: Language of the 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

- 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 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
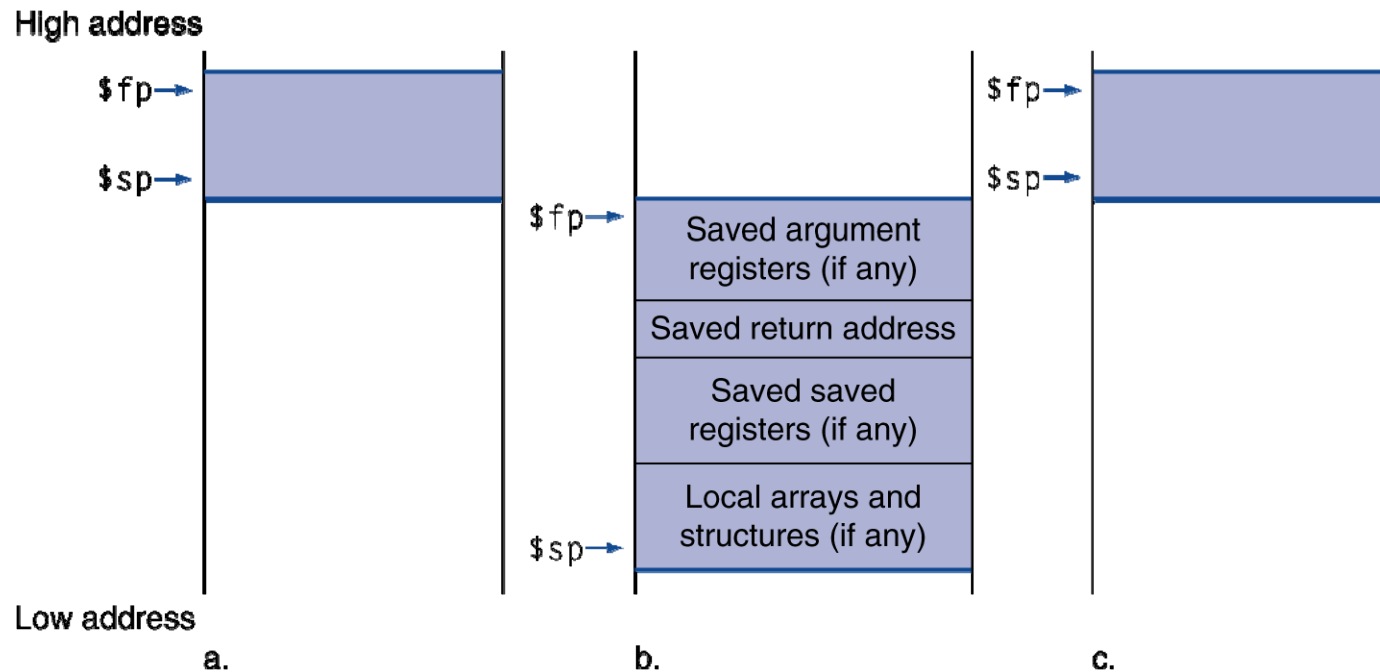
# Non-Leaf Procedure Example

- 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

High address

| | | | |
|---|---|---|---|
| $fp→ | | $fp→ | |
| $sp→ | | $sp→ | |

$fp→

| Saved argument registers (if any) |
|---|
| Saved return address |
| Saved saved registers (if any) |
| Local arrays and structures (if any) |

$sp→

Low address

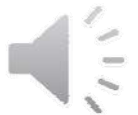a.                              b.                              c.

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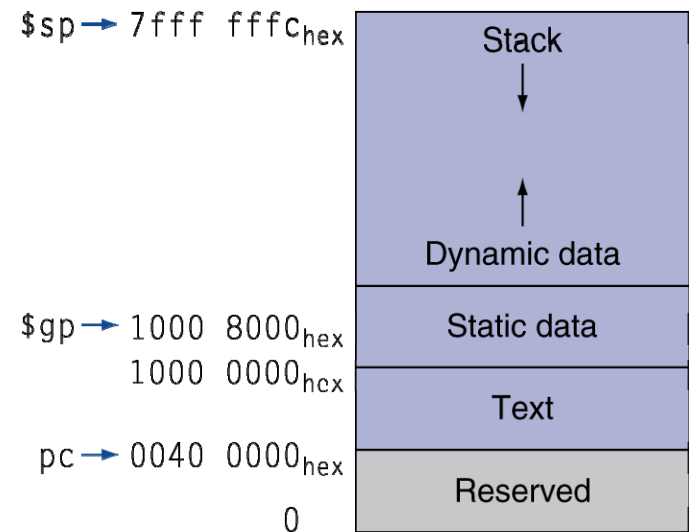  - Used by some compilers to manage stack storage

# Chapter 2

## Instructions: Language of the Computer

# 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

```
$sp → 7fff fffc_hex    | Stack        |
                       |      ↓       |
                       |              |
                       |      ↑       |
                       | Dynamic data |
$gp → 1000 8000_hex    |--------------|
      1000 0000_hex    | Static data  |
                       |--------------|
                       | Text         |
pc → 0040 0000_hex     |--------------|
      0                | Reserved     |
```

# 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 0111 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
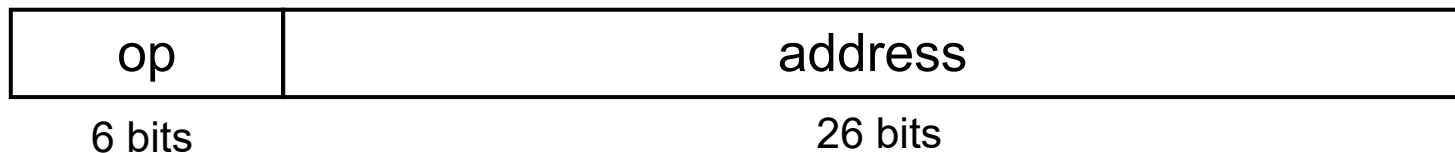
| op | rs | rt | constant or address |
|----|----|----|---------------------|
| 6 bits | 5 bits | 5 bits | 16 bits |

- 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 |

- (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  80000 | 0 | 0 | 19 | 9 | 4 | 0 |

```
Loop:  sll   $t1, $s3, 2      80000
       add   $t1, $t1, $s6    80004
       lw    $t0, 0($t1)      80008
       bne   $t0, $s5, Exit   80012
       addi  $s3, $s3, 1      80016
       j     Loop             80020
Exit:  …                      80024
```

| Addr | Col1 | Col2 | Col3 | Col4 | Col5 | Col6 |
|------|------|------|------|------|------|------|
| 80000 | 0 | 0 | 19 | 9 | 4 | 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:   …
```
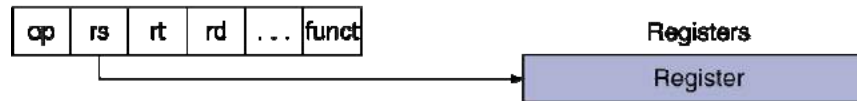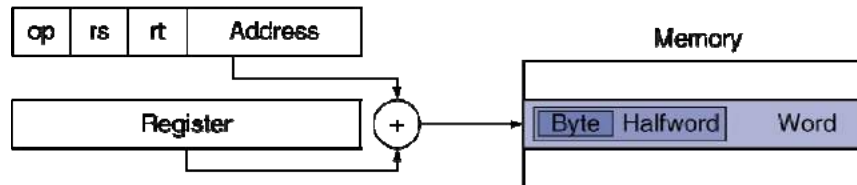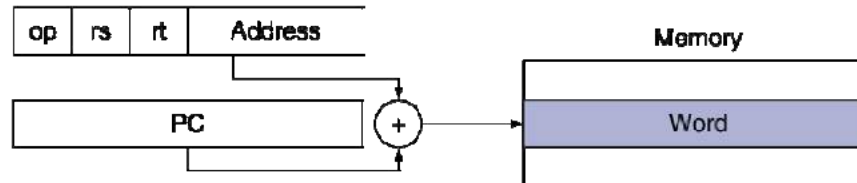
# Addressing Mode Summary

1. Immediate addressing

| op | rs | rt | Immediate |
|----|----|----|-----------|

2. Register addressing

| op | rs | rt | rd | . . . | funct |
|----|----|----|----|-------|-------|

Registers

| Register |
|----------|

3. Base addressing

| op | rs | rt | Address |
|----|----|----|---------|

| Register | + |

Memory

| Byte | Halfword | Word |

4. PC-relative addressing

| op | rs | rt | Address |
|----|----|----|---------|

| PC | + |

Memory

| Word |

5. Pseudodirect addressing

| op | Address |
|----|---------|

| PC | : |

Memory

| Word |