Chapter 2

Instructions: Language of the Computer

Dr. Randa Mohamed

Chapter 2 (Continue)

- Categories of MIPS Instructions (Continue)
- Compiling If statements
- Compiling Loop Statements
- Procedure Calling

Categories of MIPS Instructions

- Arithmetic
- Logical
- Data transfer
- **Conditional Branch**
- **Unconditional Jump**

Conditional Branch

- Change the next instruction to be executed.
- Branch to a labeled instruction if a condition is true. Otherwise, continue sequentially

1. **beq $s0, $s1, L1**
   - if ($s0 == $s1) branch to instruction labeled L1;
2. **bne $s0, $s1, L1**
   - if ($s0 != $s1) branch to instruction labeled L1;

- Example:

```
         beq $s3, $s4, Mylabel
         add $s0, $s1, $s2
Mylabel: sub …
```

More Conditional Branch

- Set result to 1 if a condition is true
  - Otherwise, set to 0

3. **`slt $s0, $s1, $s2`**
   - if ($s1 < $s2) $s0 = 1; else $s0 = 0;
4. **`slti $s0, $s1, constant`**
   - if ($s1 < constant) s0 = 1; else s0 = 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
- `beq` and `bne` are the common case
- This is a good design compromise

More Conditional Branch

5. `sltu $s0, $s1, $s2`
   - Set on less than unsigned
   - if ($s1 < $s2) $s0 = 1; else $s0 = 0;
6. `sltiu $s0, $s1, constant`
   - Set on less than immediate unsigned
   - if ($s0 < constant) s0 = 1; else s0 = 0;

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

Unconditional Jump

- Jump to a labeled instruction

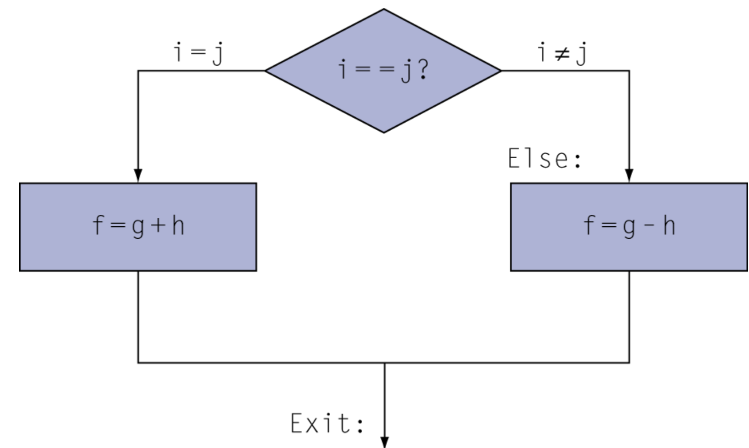
1. `j L1`
   - unconditional jump to instruction labeled L1 (target address)
2. `jr $ra`
   - unconditional jump to address saved in register $ra
   - Used for procedure call and branch far away
3. `jal L1`
   - unconditional jump to instruction labeled L1, and save current address in register $ra

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: …
```

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

```
Main(){
…
..
Fun1(x);
…..
}

Fun1(int x ){
int y = x+1;
return y;
}
```

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
- $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

Procedure Calling Summary

- The calling program, or **caller**, puts the parameter values in **$a0–$a3** and uses **jal X** to jump to procedure X (sometimes named the **callee**).
- The callee then performs the calculations, places the results in **$v0 and $v1**, and returns control to the caller using **jr $ra**.

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)
    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

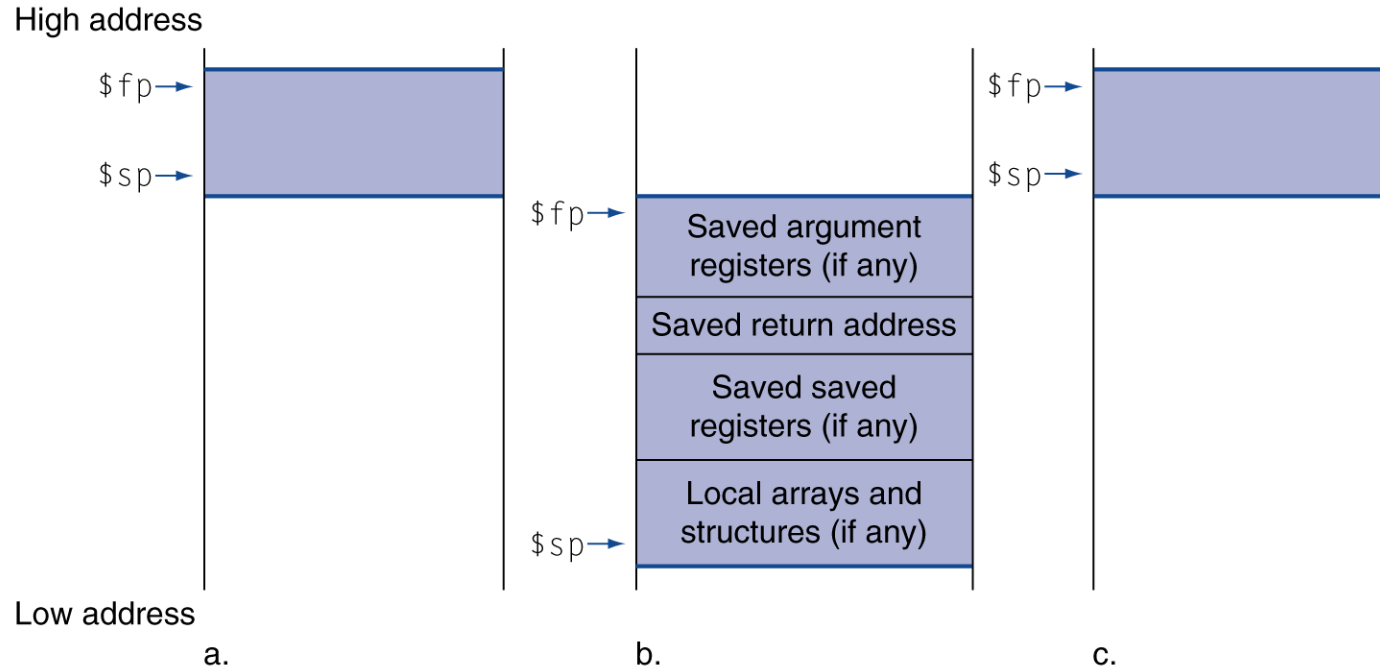
MK MORGAN KAUFMANN

Leaf Procedure Example

- MIPS code: Calling program:

```
add $s0, $t0,$t1
 jal leaf_example
 sub $s0,$s0,$v0
```

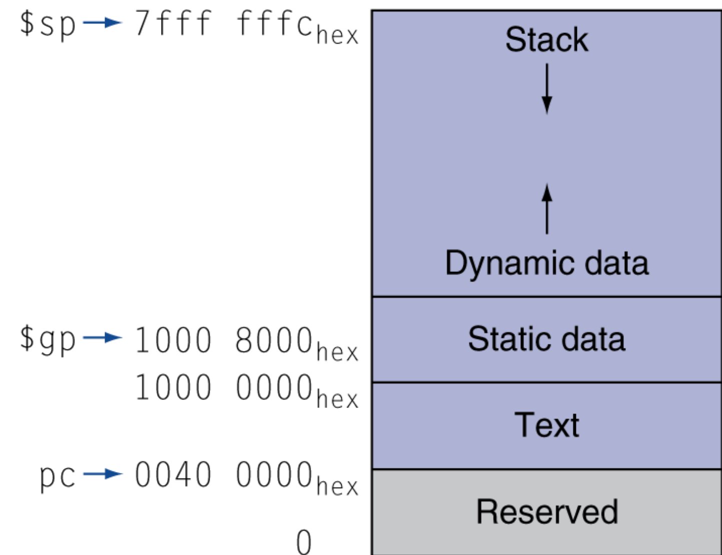
Local Data on the Stack



- Local data allocated by callee
- Procedure frame
  - 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
- Dynamic data: heap
- Stack



Problems to solve

- 2.12
  - Note that overflow occurs when the result is outside the range of value that can be stored in 32 bits(n):
  - Unsigned operation: Range: 0 to $+2^n - 1$

  (0 to +4,294,967,295)
  - Signed operations: Range: -2^{n-1} to $+2^{n-1} - 1$

  (–2,147,483,648 to +2,147,483,647)
- 2.19, 2.22, 2.23, 2.26, 2.29