Chapter 2

Instructions: Language of the Computer

Lecture #3

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:

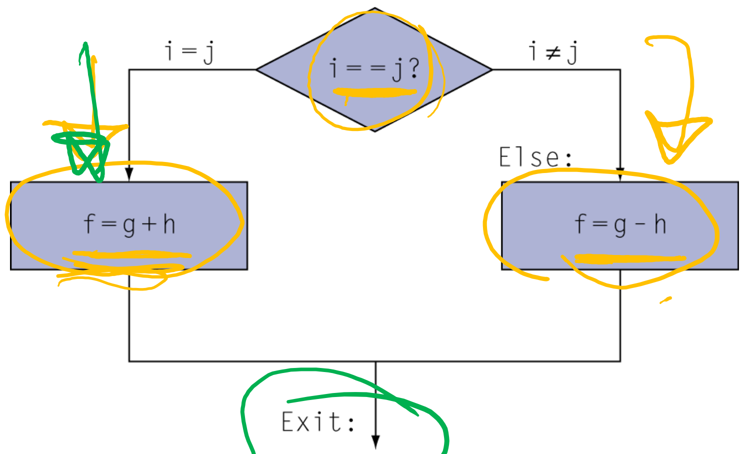
```
if (i==j) f = g+h;
else f = g-h;
```

  - 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:

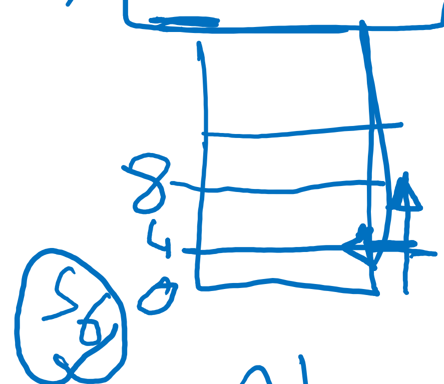
```
while (save[i] == k) i += 1;
```

  - 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: …
```

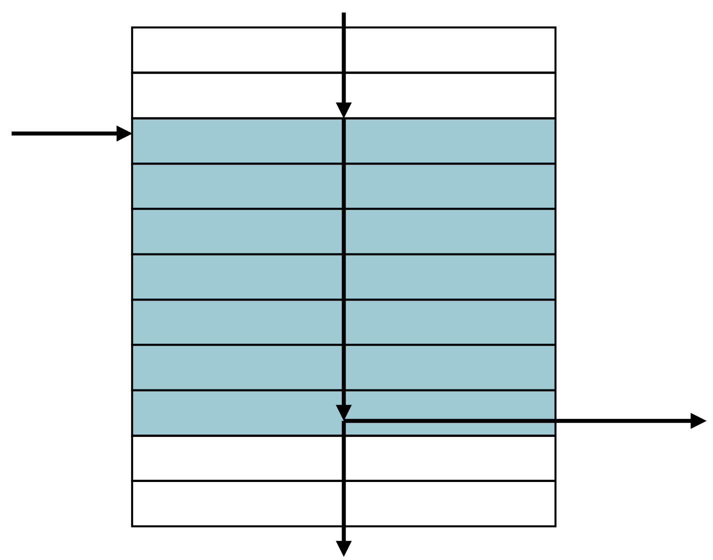
Lw □, off(base)



base+off

Basic Blocks

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



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;

Ityp

- Use in combination with beq, bne

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

$t_0 = 1$

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

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 ⇒ \$t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ \$t0 = 0

Instruction Formats (cont.)

Instruction	Format	op	funct	Instruction	Format	op	funct
add	R	0_{10}	32_{10}	**lw**	I	35	
addi	I	8		**nor**	R	0	39
and	R	0	36	**or**	R	0	37
andi	I	12		**ori**	I	13	
beq	I	4		**sb**	I	40	
bne	I	5		**sh**	I	41	
j	J	2		**sll**	R	0	0
jal	J	3		**slt**	R	0	42
jr	R	0	8	**slti**	I	10	
lb	I	32		**sra/srl**	R	0	3/2
lh	I	33		**sub**	R	0	34
lui	I	15		**sw**	I	43	

- Refer to the **MIPS reference data card** in the internal front cover of the book for the **op** and **funct** values for all instructions

Instruction Formats (cont.)

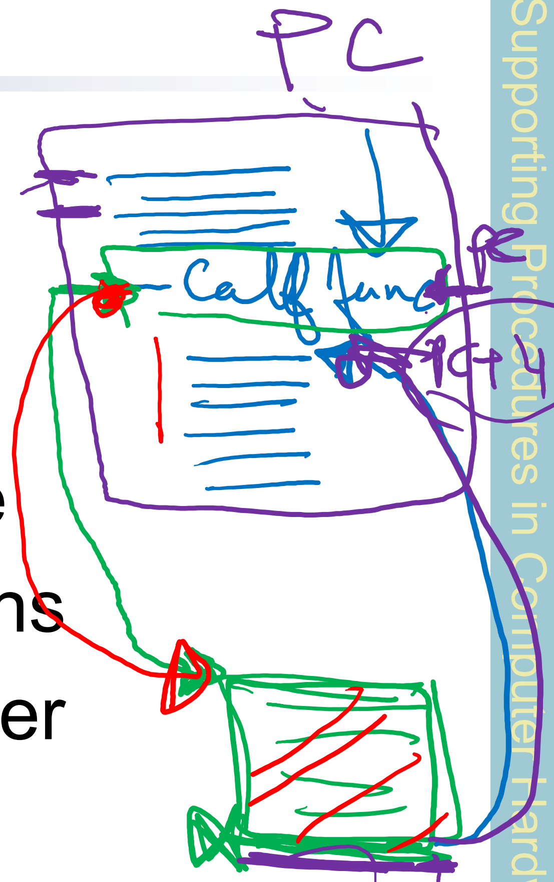
Register-format instructions	
Arithmetic and logic	
add	$t1, $t2, $t3
sub	$t1, $t2, $t3
and	$t1, $t2, $t3
or	$t1, $t2, $t3
nor	$t1, $t2, $t3
slt	$t1, $t2, $t3
Shift	
sll	$t1, $t2, 5
sra/srl	$t1, $t2, 5
Jump	
jr	$ra

Jump-format instructions	
Jump	
j	L1
jal	L1

Immediate-format instructions	
Arithmetic and logic	
addi	$t1, $t2, 15
andi	$t1, $t2, 15
ori	$t1, $t2, 15
slti	$t1, $t2, 15
lui	$t1, 15
Load and store	
lw	$t1, 15 ($t2)
lh	$t1, 15 ($t2)
lb	$t1, 15 ($t2)
sw	$t1, 15 ($t2)
sh	$t1, 15 ($t2)
sb	$t1, 15 ($t2)
Decision making	
beq	$t1, $t2, L1
bne	$t1, $t2, L1

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
- $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 <u>following</u> 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

PC = PC+4

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**.

- **Using a stack (last-in first-out) with procedures:**

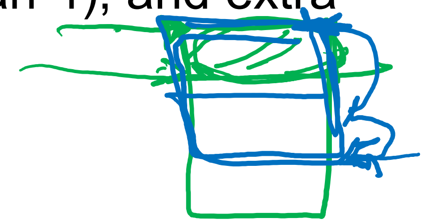
– **Stack:** a linear list for which all insertions (**push** operations) and deletions (**pop** operations) are performed at one end called stack **top**

– The **stack pointer** (**$sp**) points to the most recently allocated address

– Grows from higher addresses to lower addresses

– Ideal structure for spilling registers, extra arguments (more than 4), and extra return values (more than 2).

– Used to save local variables that do not fit in registers



- **Preserving registers:**

– **$s0-$s7**: 8 saved registers that are preserved on a procedure call (the callee saves used ones on the stack and restores them upon return)

– **$t0-$t9**: 10 temporary registers that are not preserved by the callee

- **Types of procedures:**

– **Leaf procedures:** do not call other procedures

– **Nested procedures:** call other procedures

- **Nest procedures** require saving on stack: **argument registers ($a0-$a3), temporary registers ($t0-$t9)** needed after the call, **return address register ($ra)**, and any **saved registers ($s0-$s7)** used by the callee

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

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 1;
  else return n * fact(n - 1);
}
```

a0

arg

  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

n * fact(n-1)

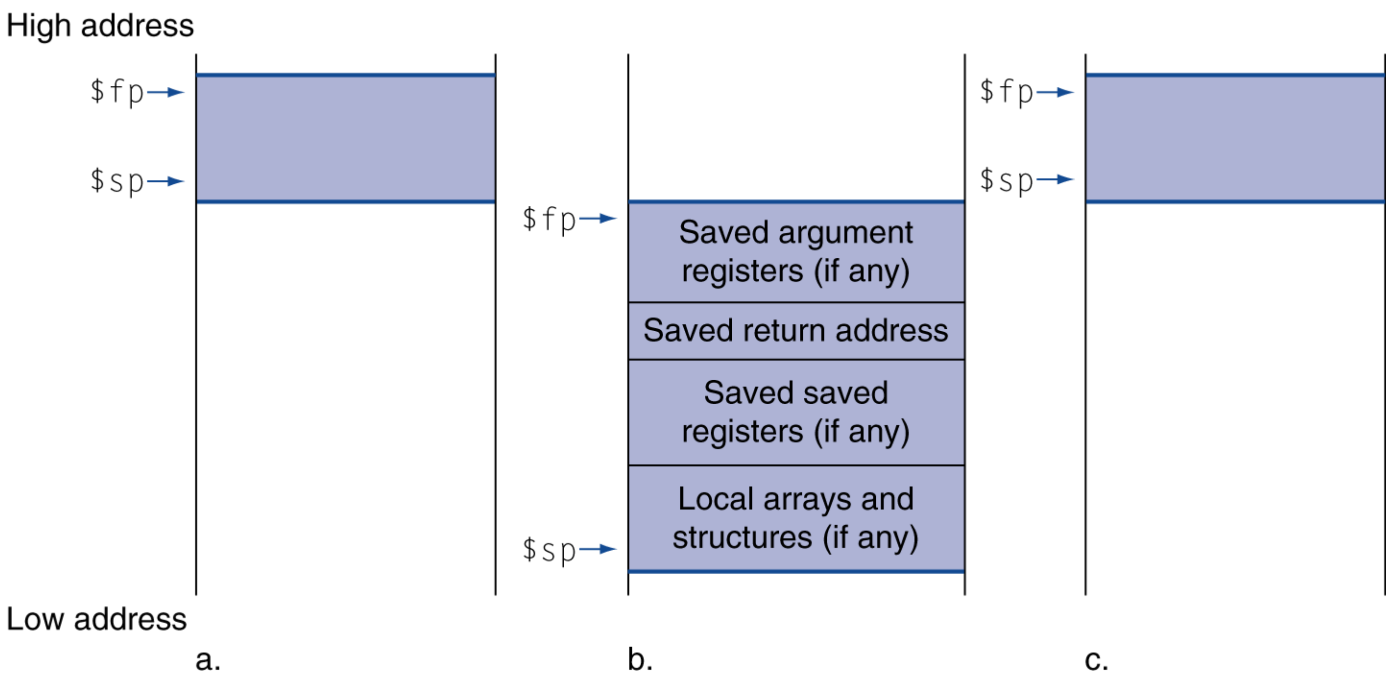
- MIPS code:

a0 = 1 t0 = 1 return

t0 = 0 else

```
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
```

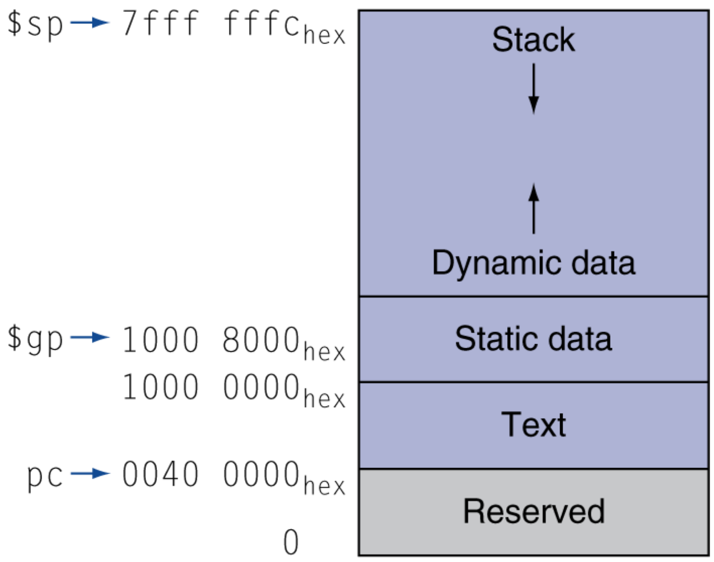
Local Data on the Stack



- Local data allocated by callee
- 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



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

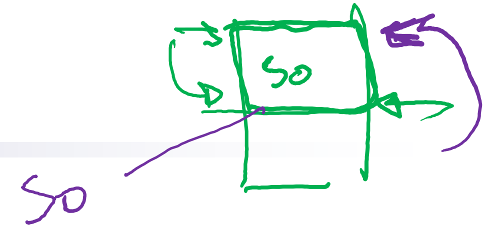
String Copy Example

- C code:

```
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
```

C Sort Example **Self Study**

- Illustrates use of assembly instructions for a C bubble sort function
- 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
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, **n** in $a1, i in $s0, j in $s1

The Procedure Body

```
        move $s2, $a0          # save $a0 into $s2
        move $s3, $a1          # save $a1 into $s3
```
Move params

```
        move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3    # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
```
Outer loop

```
        beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
        addi $s1, $s0, –1      # j = i – 1
for2tst: slti $t0, $s1, 0      # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2       # $t1 = j * 4
        add  $t2, $s2, $t1     # $t2 = v + (j * 4)
        lw   $t3, 0($t2)       # $t3 = v[j]
        lw   $t4, 4($t2)       # $t4 = v[j + 1]
        slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
```
Inner loop

```
        move $a0, $s2          # 1st param of swap is v (old $a0)
        move $a1, $s1          # 2nd param of swap is j
        jal  swap              # call swap procedure
```
Pass params & call

```
        addi $s1, $s1, –1      # j –= 1
        j    for2tst           # jump to test of inner loop
```
Inner loop

```
exit2:  addi $s0, $s0, 1       # i += 1
        j    for1tst           # jump to test of outer loop
```
Outer loop

The Full Procedure

```
sort:    addi $sp,$sp, -20         # make room on stack for 5 registers
         sw $ra, 16($sp)           # save $ra on stack
         sw $s3,12($sp)            # save $s3 on stack
         sw $s2, 8($sp)            # save $s2 on stack
         sw $s1, 4($sp)            # save $s1 on stack
         sw $s0, 0($sp)            # save $s0 on stack
         …                         # procedure body
         …
         exit1: lw $s0, 0($sp)     # restore $s0 from stack
         lw $s1, 4($sp)            # restore $s1 from stack
         lw $s2, 8($sp)            # restore $s2 from stack
         lw $s3,12($sp)            # restore $s3 from stack
         lw $ra,16($sp)            # restore $ra from stack
         addi $sp,$sp, 20          # restore stack pointer
         jr $ra                    # return to calling routine
```