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

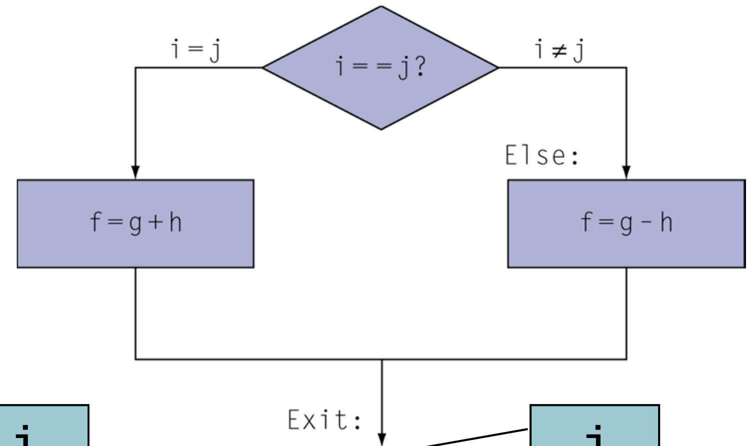
`Loop: sll  $t1, $s3, 2`	80000	0	0	19	9	4	0
`      add  $t1, $t1, $s6`	80004	0	9	22	9	0	32
`      lw   $t0, 0($t1)`	80008	35	9	8	0		
`      bne  $t0, $s5, Exit`	80012	5	8	21	2		
`      addi $s3, $s3, 1`	80016	8	19	19	1		
`      j    Loop`	80020	2	20000				
`Exit: …`	80024						

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

i

j

```
      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: …
```

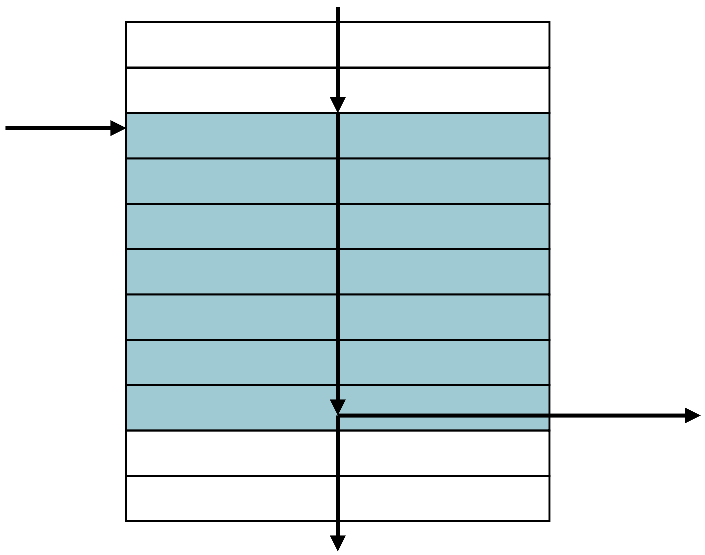
Multiply *i* by 4

Address of save[i]

save[i]

Basic Blocks

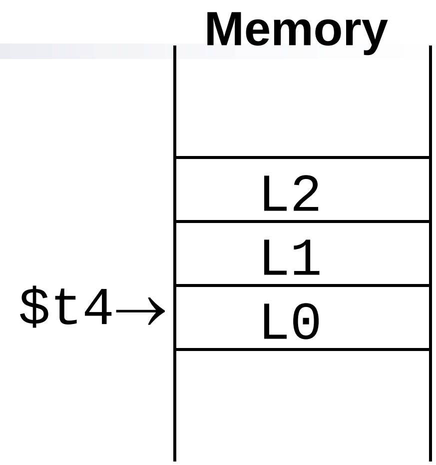
- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



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

Compiling Case Statement

```
switch (k) {
    case 0:  h=i+j;  break; /*k=0*/
    case 1:  h=i+h;  break; /*k=1*/
    case 2:  h=i-j;  break; /*k=2*/
```



- Assuming three sequential words in memory starting at the address in `$t4` have the addresses of the labels L0, L1, and L2 and **k** is in **`$s2`**

```
        add   $t1, $s2, $s2         #$t1 = 2*k
        add   $t1, $t1, $t1         #$t1 = 4*k
        add   $t1, $t1, $t4         #$t1 = addr of JumpT[k]
        lw    $t0, 0($t1)           #$t0 = JumpT[k]
        jr    $t0                   #jump based on $t0
L0:     add   $s3, $s0, $s1         #k=0 so h=i+j
        j     Exit
L1:     add   $s3, $s0, $s3         #k=1 so h=i+h
        j     Exit
L2:     sub   $s3, $s0, $s1         #k=2 so h=i-j
Exit:   . . .
```

More Conditional Operations

- Set dest 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

Procedure Calling

- Steps required
  1. Place parameters in a place where the procedure can access them
  2. Transfer control to procedure
  3. Acquire storage (resources) for procedure
  4. Perform procedure’s operations
  5. Place result in a place where the caller can access them.
  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

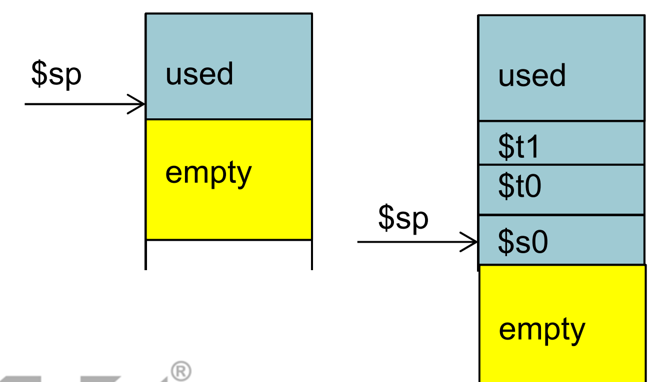
- C code:

```
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
  - Will need $t0, and $t1 in the calculation of `f`

Stack

- The best way to store registers is a **stack**
- A stack is a first-in-last-out data structure
- Stack pointer points to the last element in the stack (or the first empty place).
- Traditionally stack grows from higher to lower addresses



The stack
The stack after ***pushing*** $t1 $t0 and $s0

Procedure Call

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

```
leaf_example:
addi   $sp, $sp, -12#adjust stack to make room for 3 items
sw     $t1, 8($sp)   # push $t1
sw     $t0, 4($sp)   # push $t0
sw     $s0, 0($sp)   # push $s0
```

??

Save registers

Procedure Call

```
add     $t0, $a0, $a1        #$t0 = g+h
add     $t1, $a2, $a3        #$t1 = i+j
sub     $s0, $t0, $t1        #$s0 = (g+h)-(i+j)

add     $v0, $s0, $zero      #put the result in $v0

lw      $s0, 0($sp)   #restore $s0
add     $t0, 4($sp)   #restore $t0
sub     $t1, 8($sp)   #restore $t1
addi    $sp, $sp, 12  #restore $sp

jr      $ra    #jump back to the calling routing
```

Do calculation

put result in $v0

Clean up (remove data from the stack)

Return control to caller

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
sub $s0, $t0, $t1
	Procedure body

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);
}
```

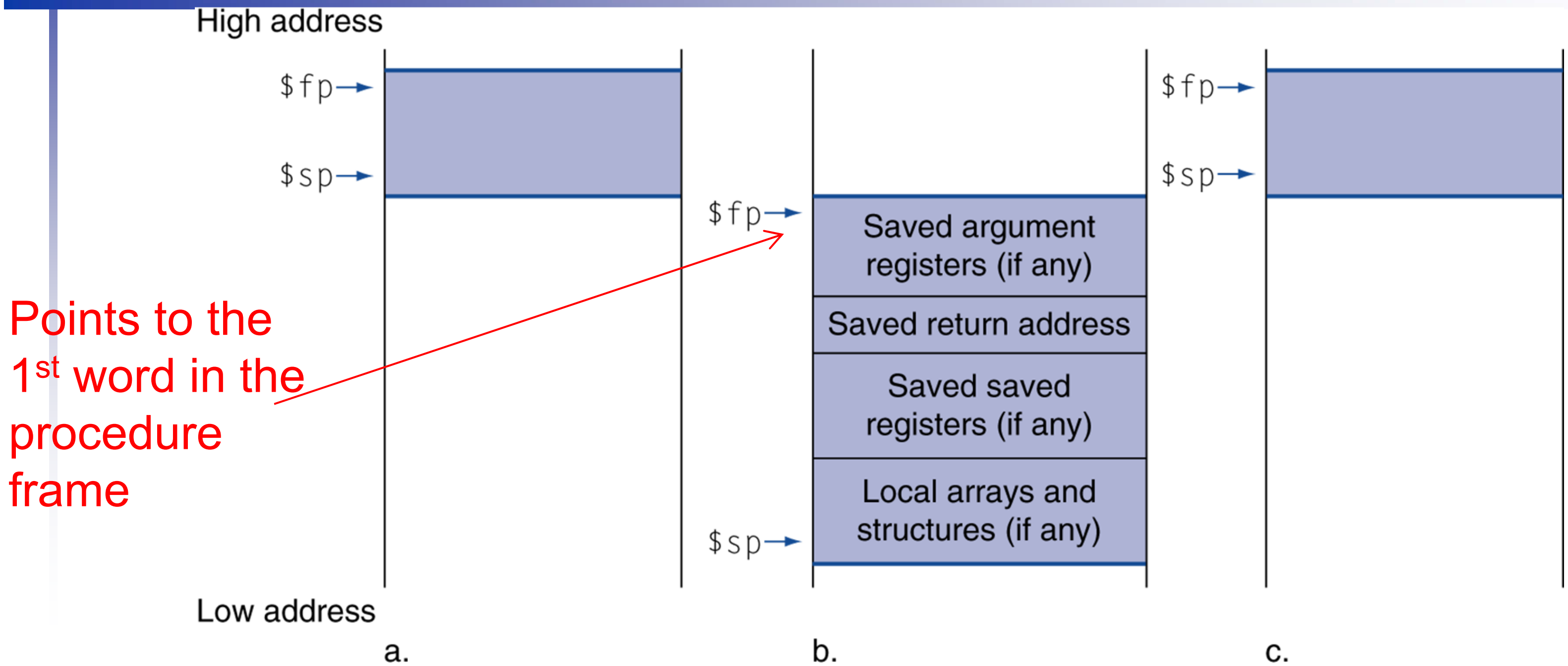
  - Argument n in $a0
  - Result in $v0

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
```

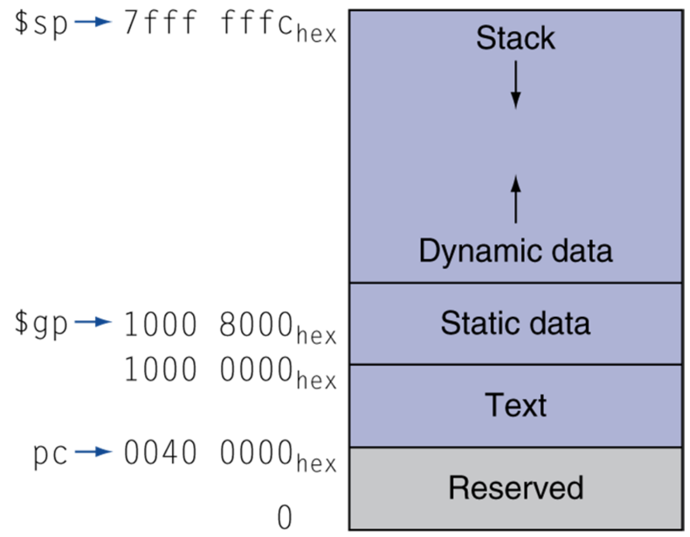
Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- **Procedure frame (activation record)**
  - Used by some compilers to manage stack storage
  - Fixed, does not change during the function execution
  - A stable base register to address for local memory reference

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



Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

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

Load 4,000,000 in $s0

`lui $s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

Zero extended

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