

Chapter 2

Computer Abstractions and Technology

Lesson 4: MIPS (cont...)

Logical Operations

- ◆ Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- ◆ shamt: how many positions to shift
- ◆ Shift left logical
 - ❑ Shift left and fill with 0 bits
 - ❑ `sll` by i bits multiplies by 2^i
- ◆ Shift right logical
 - ❑ Shift right and fill with 0 bits
 - ❑ `srl` by i bits divides by 2^i (unsigned only)

AND Operations

- ◆ Useful to mask bits in a word
 - Select some bits, clear others to 0

and \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- ◆ Useful to include bits in a word
 - Set some bits to 1, leave others unchanged

or \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- ◆ Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- ◆ MIPS has NOR 3-operand instruction
 - $a \text{ NOR } b == \text{NOT } (a \text{ OR } b)$

`nor $t0, $t1, $zero`

Register 0: always read as zero

\$t1	0000	0000	0000	0000	0011	1100	0000	0000
\$zero	0000	0000	0000	0000	0000	0000	0000	0000
\$t0	1111	1111	1111	1111	1100	0011	1111	1111

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

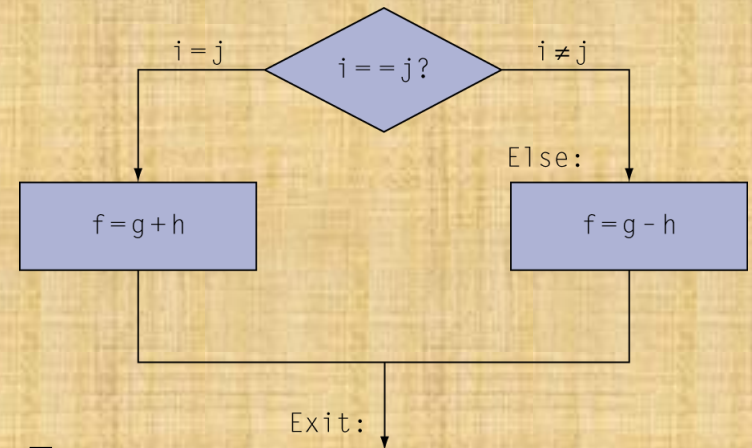
- ◆ C code:

```
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;
```

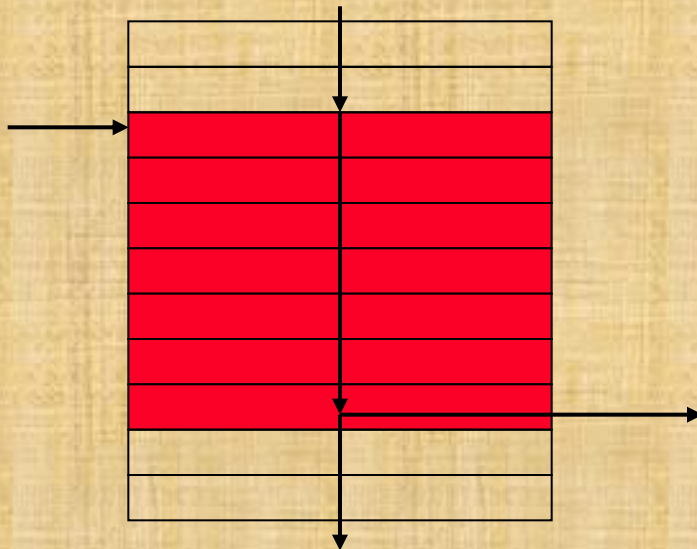
- i in \$s3, k in \$s5, address of save in \$s6

- ◆ Compiled MIPS code:

```
Loop:  sll    $t1, $s3, 2    //multiply by 4 (i.e. 22)
       add    $t1, $t1, $s6  //add i*4 to base address
       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 ($rs < \text{constant}$) $rt = 1$; 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 b1t, bge, etc?
- ♦ Hardware for $<$, $\geq$, ... slower than $=$, $\neq$
 - ❑ 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 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

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

♦ 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		

Prepare stack to receive \$s0
Mem address increase ↑
Stack grows ↓ (so pointer decreased by 4)
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

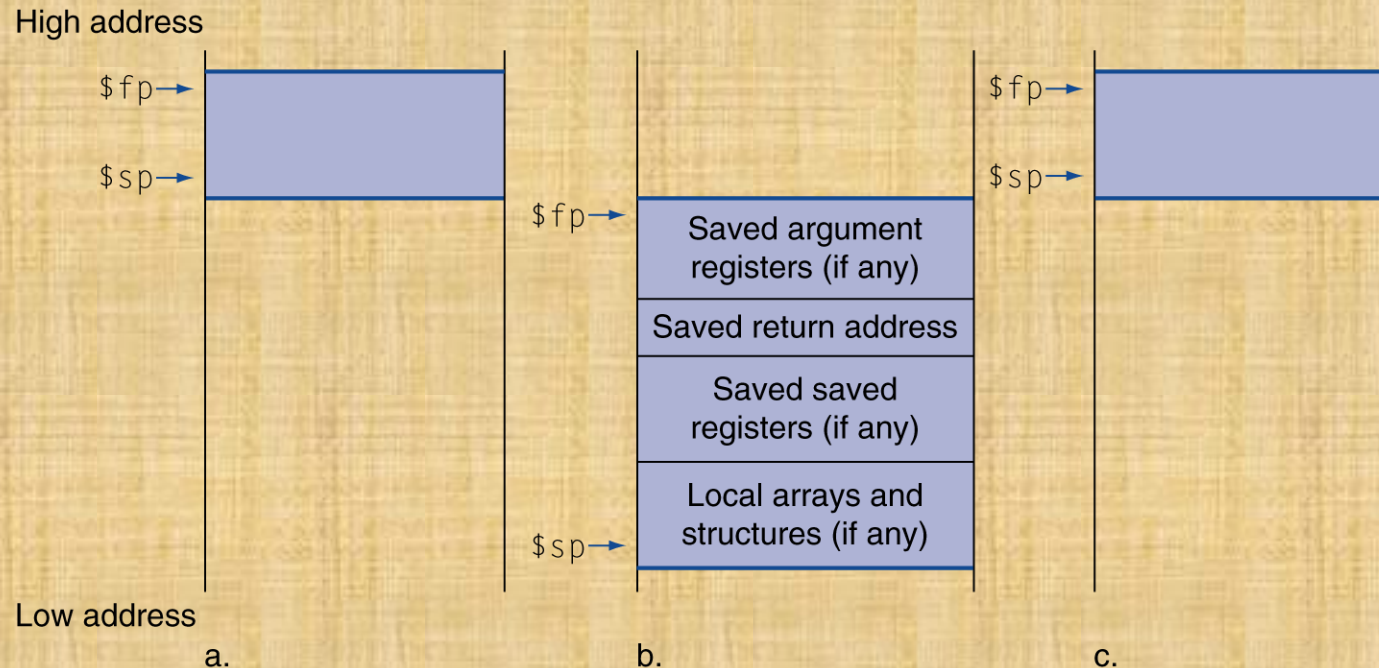
♦ MIPS code:

```
int fact (int n)
{
    if (n < 1) return f;
    else return n * fact(n - 1);
}
```

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

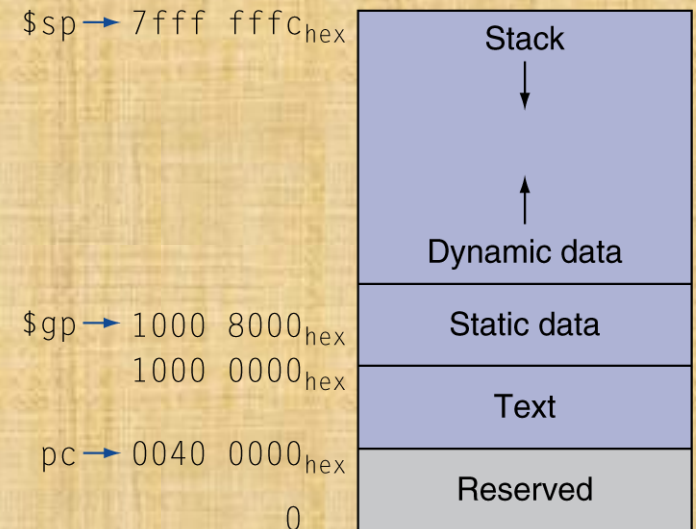
Local Data on the Stack



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

Memory Layout

- ◆ Text: program code
- ◆ Static data: global variables
 - ❑ e.g., static variables in C, constant arrays and strings
 - ❑ \$gp initialized to address allowing $\pm$ offsets into this segment
- ◆ Dynamic data: heap
 - ❑ E.g., malloc in C, new in Java and C++
- ◆ Stack: automatic storage



Next Time

- ◆ More MIPS instructions....