

COL 216

Holi Semester (Feb-May), 2020-21

Mondays and Thursdays 9:30-10:50 AM, Online

COMPUTER ARCHITECTURE

INSTRUCTIONS: LANGUAGE OF THE COMPUTER

Preeti Ranjan Panda

Dept. of Computer Science and Engineering, IIT Delhi

Acknowledgment

Minor adaptation of lecture slides provided by publishers of: *Computer Organization and Design: The Hardware/Software Interface*

David A. Patterson and John L. Hennessy

Morgan Kaufmann Publisher

Instruction Set

- The set of instructions understood by a computer
 - High-level language has to be translated to this (by compiler)
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Later, complex instructions were gradually introduced (Why?)
- Many modern computers also have simple instruction sets (Why?)

Arithmetic Operations

- Add and subtract, three operands

- Two sources and one destination

```
add a, b, c # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1: Simplicity favours regularity*
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

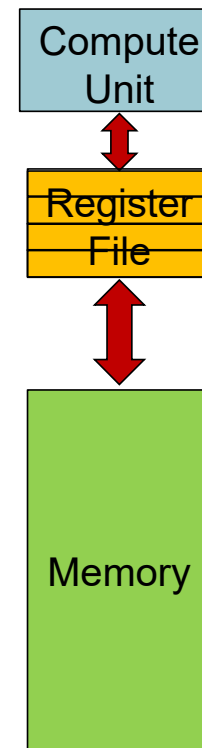
```
f = (g + h) - (i + j);
```

- Assembly language-level code:

```
add t0, g, h    # temp t0 = g + h  
add t1, i, j    # temp t1 = i + j  
sub f, t0, t1   # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS Architecture: 32×32 -bit **register file**
 - Used for frequently accessed data
 - 32-bit data called a “word”
- *Design Principle 2: Smaller is faster*
 - vs. main memory: millions of locations



MIPS Registers

Register File



...and other registers

Register Operand Example

- C code:

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

- Define Register Mapping 

- Compiled MIPS code:

```
add $t0, $s1, $s2
```

```
add $t1, $s3, $s4
```

```
sub $s0, $t0, $t1
```

Register
Mapping

\$s0	f
\$s1	g
\$s2	h
\$s3	i
\$s4	j
\$t0	g+h
\$t1	i+j

Memory Operands

- Register file too small
- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte

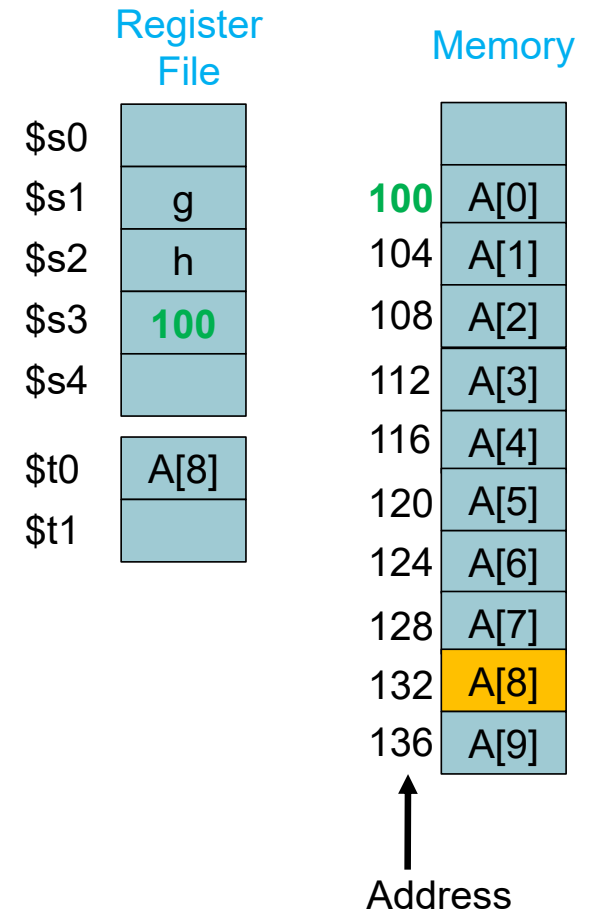
Memory Operand Example 1

- C code:
`g = h + A[8];`
 - `g` in `$s1`, `h` in `$s2`,
 - base address of `A` in `$s3`
- Compiled MIPS code:
 - Index 8 requires offset 32
 - 4 bytes per word

```
lw $t0, 32($s3) # load word  
add $s1, $s2, $t0
```

offset

base register



Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

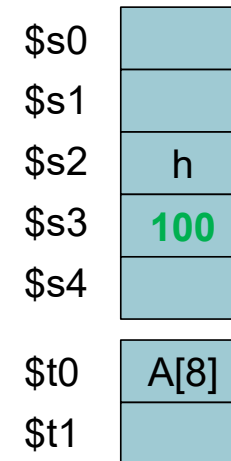
- `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

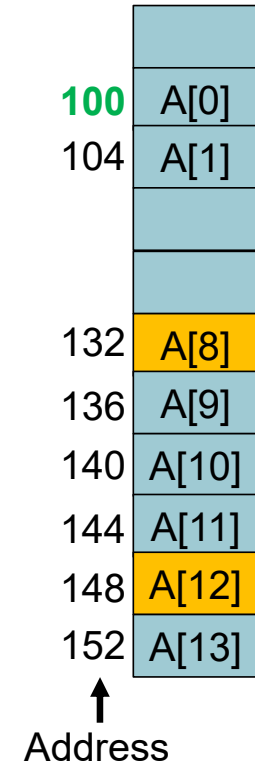
- Index 8 requires offset of 32

```
lw  $t0, 32($s3)  # load word
add $t0, $s2, $t0  # Why?
sw  $t0, 48($s3)  # store word
```

Register File



Memory



Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction
`addi $s3, $s3, 4`
- No subtract immediate instruction
 - Just use a negative constant
`addi $s2, $s1, -1`
- *Design Principle 3: Make the common case fast*
 - Small constants are common
 - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., move/copy between registers

```
add $t2, $s1, $zero
```

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

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

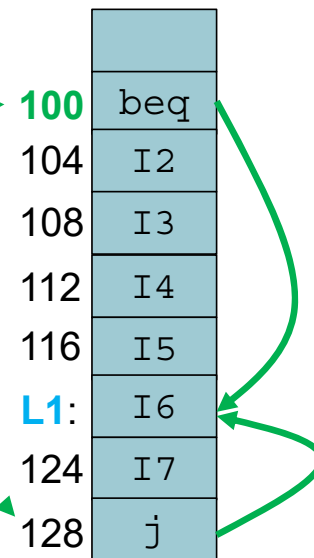
\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 1111 1111 1111 1111 11 00 0011 1111 1111

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq rs, rt, L1` ————→ 100
 - if (`rs == rt`) branch to instruction labeled L1;
- `bne rs, rt, L1`
 - if (`rs != rt`) branch to instruction labeled L1;
- `j L1` ————→ 124
 - unconditional jump to instruction labeled L1

Memory



Compiling If Statements

- C code:

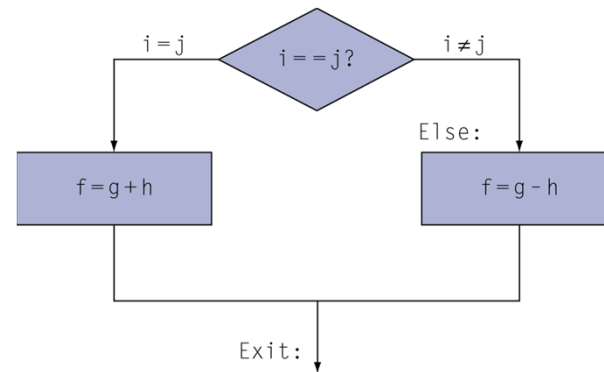
```
if (i==j) f = g+h;  
else f = g-h;
```

- 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



s0	f
s1	g
s2	h
s3	i
s4	j
r5	
r6	

Compiling Loop Statements

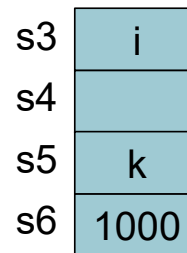
- C code:

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

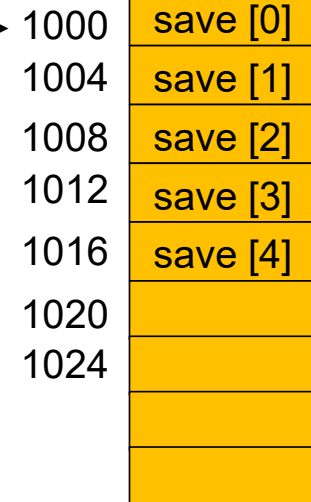
- i in $\$s3$, k in $\$s5$, address of $save$ in $\$s6$

- Compiled MIPS code:

```
Loop: sll $t1, $s3, 2 #t1=i*4 due  
      # to byte addressing  
      add $t1, $t1, $s6 #t1 = addr.  
      # of save[i]  
      lw $t0, 0($t1)  
      bne $t0, $s5, Exit  
      addi $s3, $s3, 1  
      j Loop  
Exit: ...
```



Register File



Memory

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

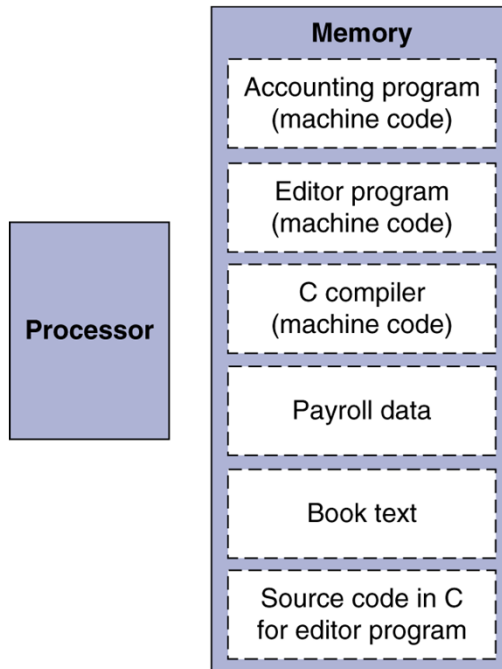
- Why not `blt`, `bge`, etc?
- Hardware for `<`, `≥`, ... slower than `=`, `≠`
 - Combining with branch involves more work per instruction, requiring a slower clock
 - **All instructions penalized!**
- `beq` and `bne` are the common case
- This is a good design compromise

Pseudo-instructions

- Instructions such as `blt` and `not` are pseudo-instructions
 - Implemented in terms of other instructions
- May translate to multiple machine instructions
 - `blt` expanded to 2 instructions (`slt + bne`)

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
 - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
 - Standardized ISAs

Representing Instructions

- Instructions are encoded in binary
 - Called machine code
- MIPS instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!
- Register numbers
 - \$t0 – \$t7 are reg's 8 – 15
 - \$t8 – \$t9 are reg's 24 – 25
 - \$s0 – \$s7 are reg's 16 – 23

MIPS R-format Instructions



- Instruction fields
 - **op**: operation code (opcode)
 - **rs**: first source register number
 - **rt**: second source register number
 - **rd**: destination register number
 - **shamt**: shift amount (00000 for now)
 - **funct**: function code (extends opcode)

R-format Example

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

add \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

Binary encoding of the instruction:

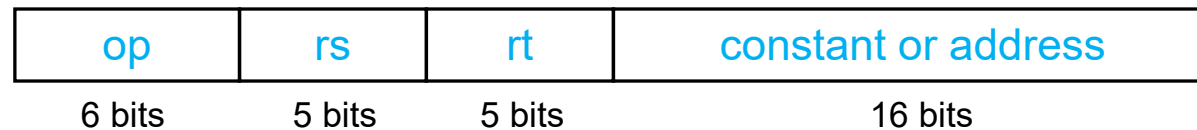
$00000010001100100100000000100000_2 = 02324020_{16}$

Shift Operations

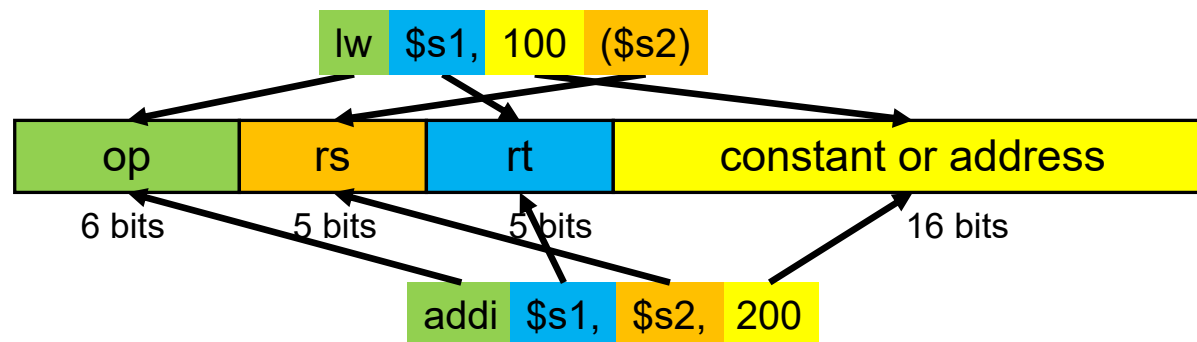


- **shamt**: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - **sll** by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - **srl** by i bits divides by 2^i (unsigned only)

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
 - **rt**: destination or source register number
 - **Constant**: -2^{15} to $+2^{15} - 1$
 - **Address**: offset added to base address in **rs**



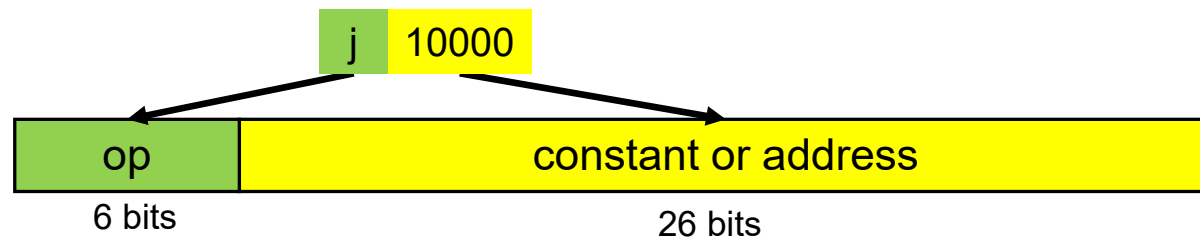
MIPS I-format Instructions



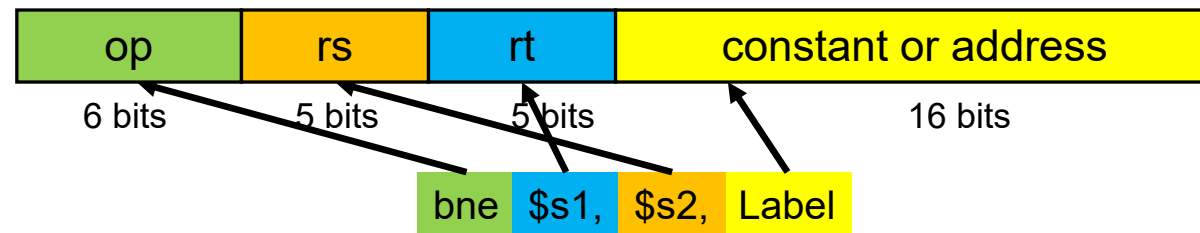
- Immediate arithmetic and load/store instructions
 - **rt**: destination or source register number
 - **Constant**: -2^{15} to $+2^{15} - 1$
 - **Address**: offset added to base address in **rs**
- *Design Principle 4: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

MIPS J-format Instructions

- Jump/Branch instructions
 - Unconditional Jump



- Conditional Jump




Register Usage

- \$v0, \$v1: result values (reg's 2 and 3)
- \$a0 – \$a3: arguments (reg's 4 – 7)
- \$t0 – \$t9: temporaries (reg's 8-15, 24,25)
 - Can be overwritten by callee
- \$s0 – \$s7: saved (reg's 16-23)
 - 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)

\$at (reg 1): assembler temporary
\$k0,\$k1 (reg 26, 27): OS reserved

Homework Problem

How do we implement the C switch/case statement?

- No submission! 
- Discussions allowed
- Needs to be efficient

Homework Problem: How to implement C switch/case?

- Nested IF?
 - $O(n)$ time to reach a case
 - Not easy to implement “break/fall through” rule

```
switch (x) {  
  case 0: A; break;  
  case 1: B; break;  
  case 2: C; if (p) D else break;  
  case 3: E; break;  
  ...  
  default: Y;  
}
```

Switch

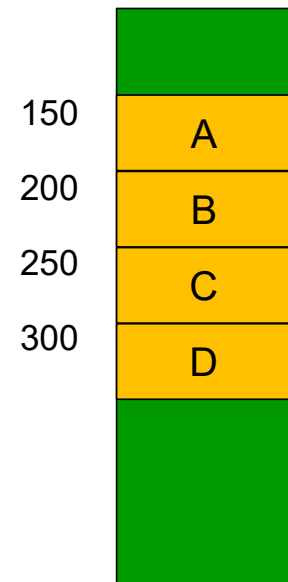
```
if (x == 0) {A;}  
else if (x == 1) {B;}  
else if (x == 2)  
    {C; if (p) {D; E}}  
else if (x == 3) {E;}  
...  
else {Y};
```

Equivalent IF?

Implementing switch: simpler case

- What if A, B, C, D, etc., all had EQUAL #instructions?
- ...and clean BREAKs
- Computed JUMP

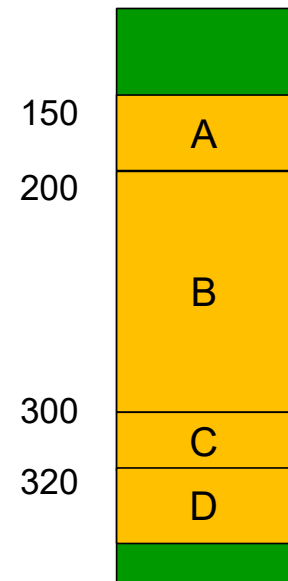
```
switch (i) {  
  case 0: A; break;  
  case 1: B; break;  
  case 2: C; break;  
  case 3: D; break;  
  ...  
  /* ignore default for now */  
}
```



Implementing switch: general case

- In general, A, B, C, D have different lengths

```
switch (i) {  
  case 0: A; break;  
  case 1: B; break;  
  case 2: C; break;  
  case 3: D; break;  
  ...  
  /* ignore default for now */  
}
```



Implementing switch: general case

- Computed JUMP
- Table L of starting locations for A, B, C, D...
- JUMP to L[i] from current location
- Need to stitch up...
 - gaps in cases?
 - default?
 - Table size?
- Explanation for
 - integer type expression
 - constant labels
 - break / fall through

```
switch (i) {  
  case 0: A; break;  
  case 1: B; break;  
  case 2: C; break;  
  case 3: D; break;  
  ...  
  /* ignore default for now */  
}
```

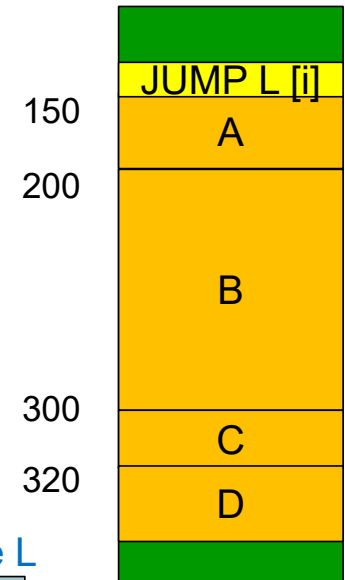
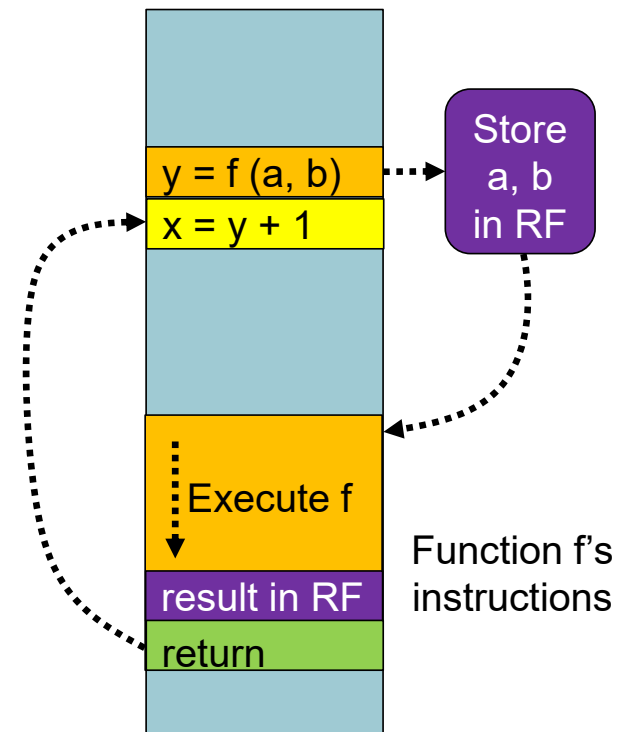


Table L

0	0
1	50
2	150
3	170

Procedure/Function 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 (How?)



Function Call Instructions

- Procedure/Function 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

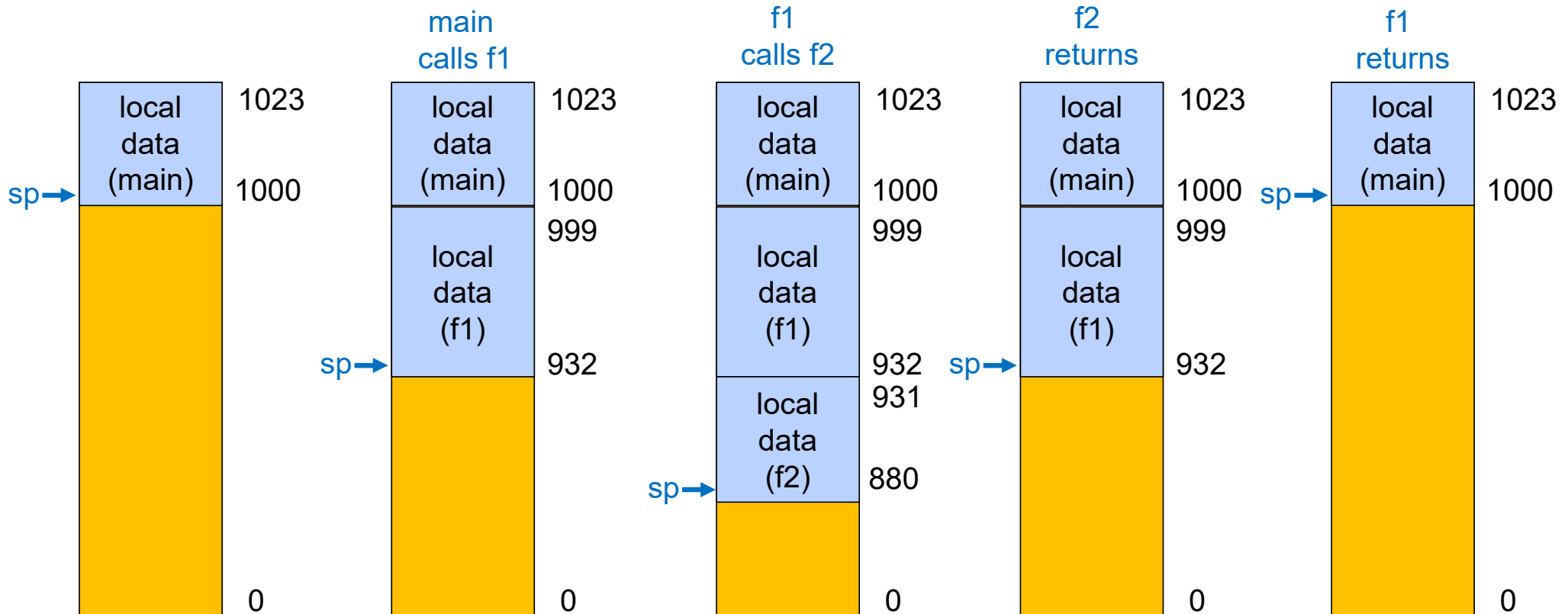
Leaf Procedure Example

- C code:

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

- Arguments g, \dots, j in $\$a0, \dots, \$a3$
- f in $\$s0$ (hence, need to **save $\$s0$ on stack**)
- Result in $\$v0$

Execution Stack



Stack Pointer (sp) points to Top of Stack

Leaf Procedure Example

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

■ MIPS code:

- g, h, i, j in \$a0, ..., \$a3, Result in \$v0
- f in \$s0 (hence, need to save \$s0 on stack)

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	

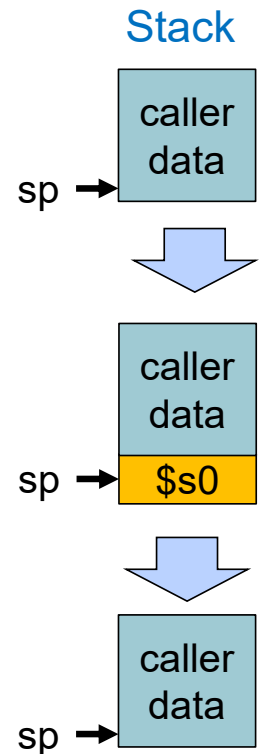
Save \$s0 on stack

Procedure body

Result

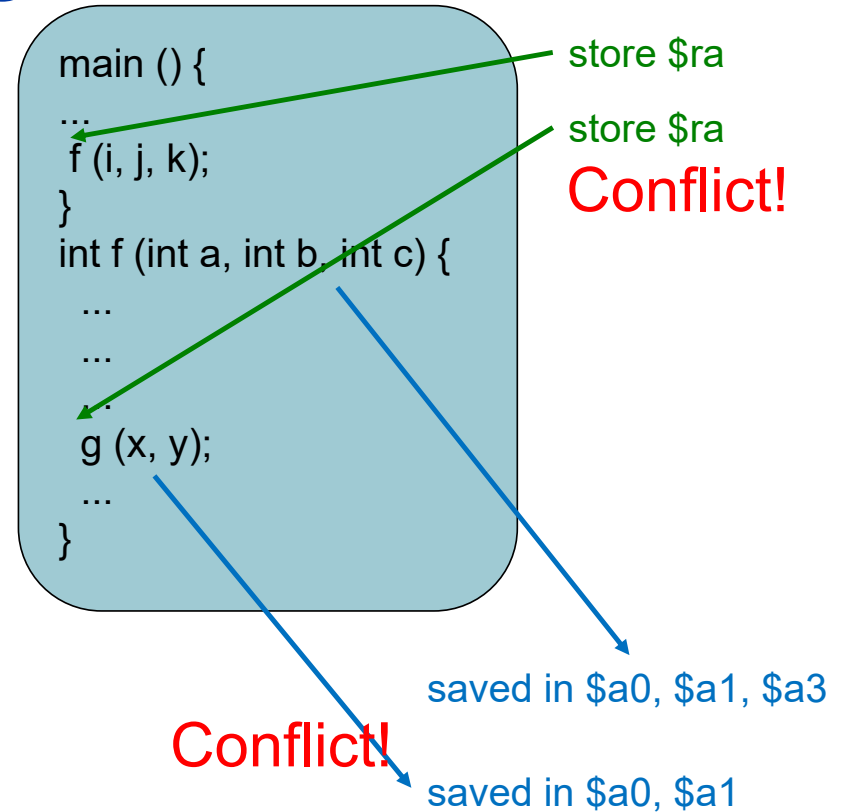
Restore \$s0

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; /* 0!=1 */
    else return n * fact(n - 1);
}
```

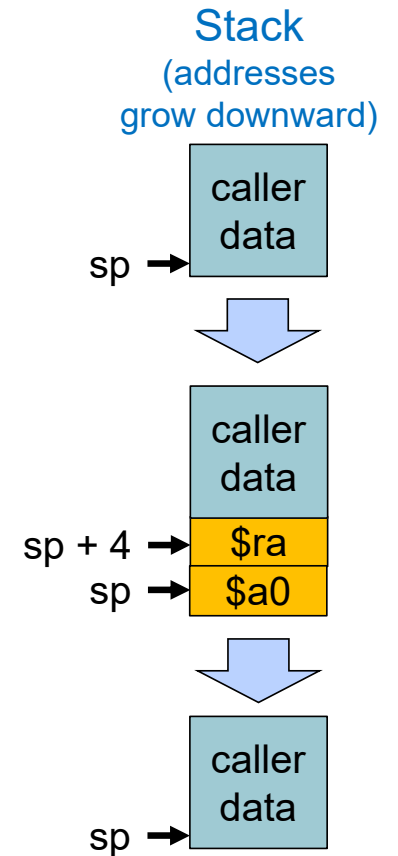
- Argument n in `$a0`
- Result in `$v0`

Non-Leaf Procedure Example

- MIPS code:

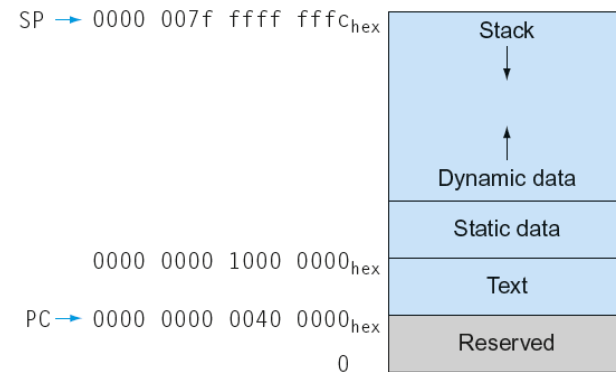
```
int fact (int n){
    if (n < 1) return 1; /* 0!=1 */
    else return n * fact(n - 1);
} Argument n in $a0, Result in $v0
```

fact:		
addi \$sp, \$sp, -8	# adjust stack for 2 items	
sw \$ra, 4(\$sp)	# save return address	
sw \$a0, 0(\$sp)	# save argument n	
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	

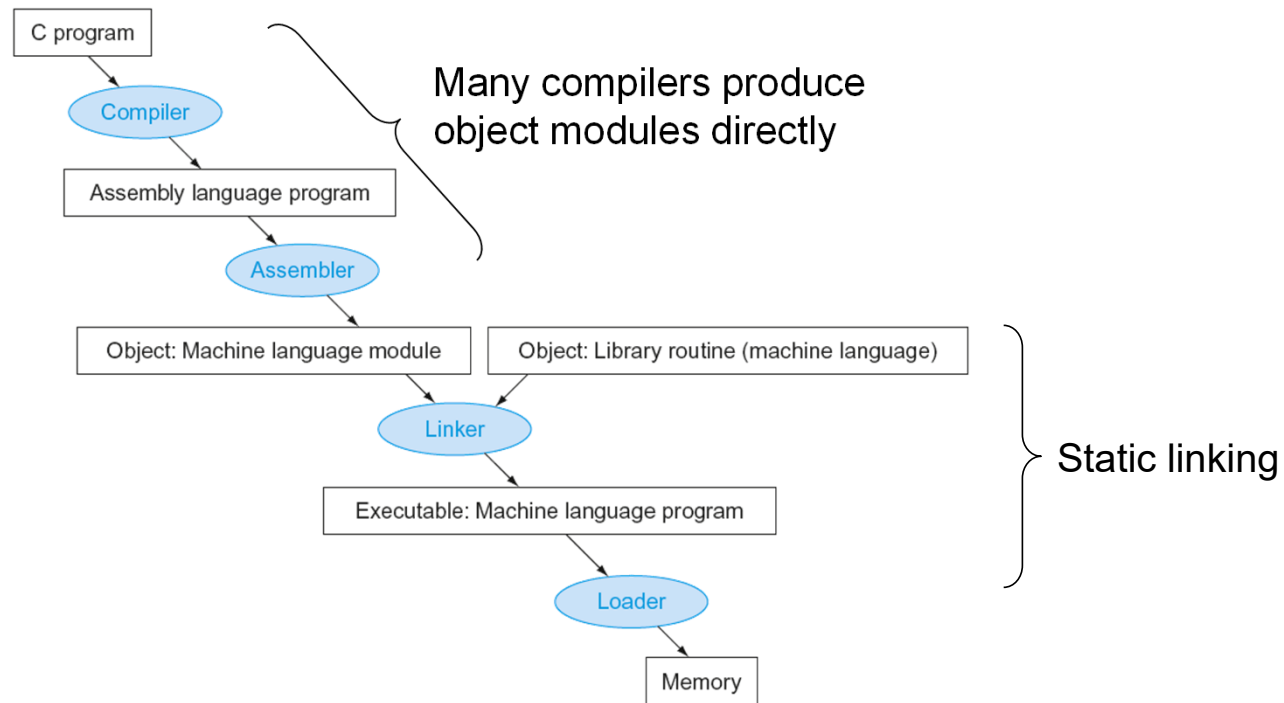


Memory Layout

- **Text**: program code
- **Static data**: global variables
 - e.g., static variables in C, constant arrays and strings
- **Dynamic data**: heap
 - E.g., malloc in C, new in C++/Java
- **Stack**: automatic storage (local to functions)



Translation and Startup



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
 - **Header**: describes contents of object module
 - e.g., segment sizes
 - **Text segment**: translated instructions
 - **Static data segment**: data allocated for the life of the program
 - **Symbol table**: global definitions and external refs
 - **Debug info**: for associating with source code

Linking Object Modules

- Produces an executable image
 1. Merges segments
 2. Resolve labels (determine their addresses)
 3. Patch location-dependent and external refs

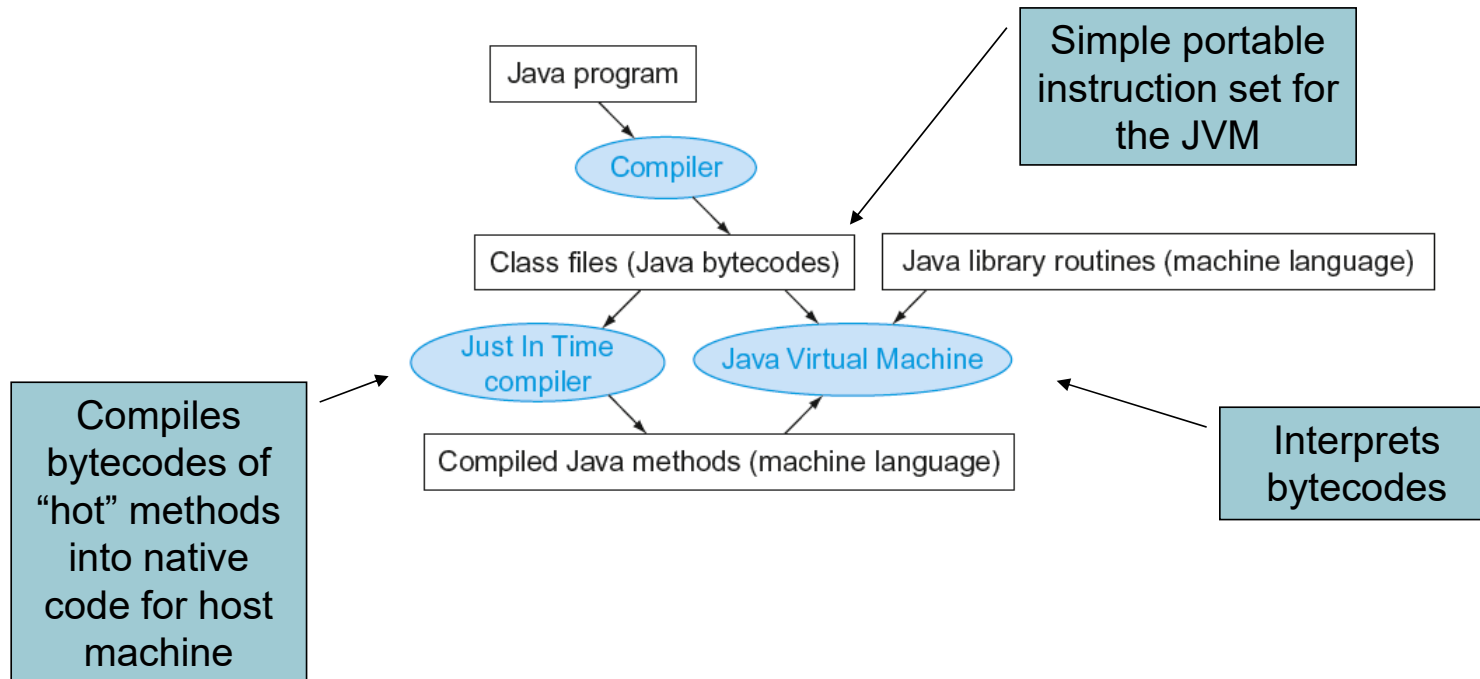
Loading a Program

- Load from image file on disk into memory
 1. Read header to determine segment sizes
 2. Copy text and initialized data into memory
 3. Set up arguments on stack
 4. Initialize registers (SP...)
 5. Jump to startup routine
 - Copies arguments to a0, ... and calls main
 - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
 - Requires procedure code to be relocatable
 - Avoids image bloat caused by static linking of all (transitively) referenced libraries
 - Automatically picks up new library versions

Starting Java Applications



Concluding Remarks

- Design principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Make the common case fast
 4. Good design demands good compromises
- Layers of software/hardware
 - Compiler, assembler, hardware