

Instructions: Language of the Computer

Computer Organization
502044

Acknowledgement

This slide show is intended for use in class, and is not a complete document. Students need to refer to the book to read more lessons and exercises. Students have the right to download and store lecture slides for reference purposes; Do not redistribute or use for purposes outside of the course.

[2]. David A. Patterson, John L. Hennessy, [2014], **Computer Organization and Design: The Hardware/Software Interface**, 5th edition, Elsevier, Amsterdam.

[3]. John L. Hennessy, David A. Patterson, [2012], **Computer Architecture: A Quantitative Approach**, 5th edition, Elsevier, Amsterdam.

 trantrungtin.tdtu.edu.vn

Syllabus

7.1 Introduction

7.2 Operations of the Computer Hardware

7.3 Operands of the Computer Hardware

7.4 Representing Instructions in the
Computer

7.5 Logical Operations

7.6 Instructions for Making Decisions

7.7 Supporting Procedures in
Computer Hardware

7.8 Communicating with People

7.9 MIPS Addressing for 32-Bit
Immediates and Addresses

Instruction Set

- The collection of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets

The MIPS Instruction Set

- Used as the example throughout the course
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
 - See MIPS Reference Data tear-out card, and Appendices B and E

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 favors regularity
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

MIPS operands

Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register \$zero always equals 0, and register \$at is reserved by the assembler to handle large constants.
2 ³⁰ memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three register operands
	add immediate	addi \$s1,\$s2,20	\$s1 = \$s2 + 20	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory
	load half	lh \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Load word as 1st half of atomic swap
Logical	store condition. word	sc \$s1,20(\$s2)	Memory[\$s2+20]=\$s1; \$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	\$s1 = 20 * 2 ¹⁶	Loads constant in upper 16 bits
	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~ (\$s2 \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	\$s1 = \$s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	\$s1 = \$s2 20	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

Arithmetic Example

- C code:

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

- Compiled MIPS 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 has a 32 by 32-bit register file
 - Used for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word”
- Assembler names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for saved variables
- Design Principle 2: Smaller is faster
 - c.f. main memory: millions of locations

Register Operand Example

- C code:
f = (g + h) - (i + j);
 - f, ..., j in \$s0, ..., \$s4
- Compiled MIPS code:

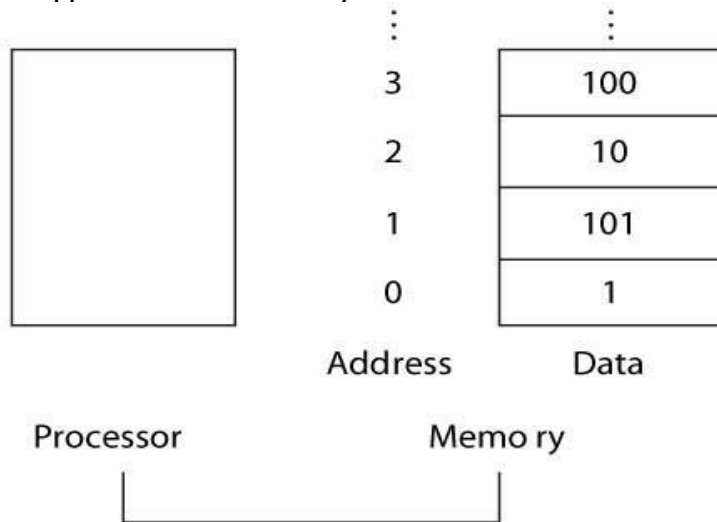
add \$t0, \$s1, \$s2

add \$t1, \$s3, \$s4

sub \$s0, \$t0, \$t1

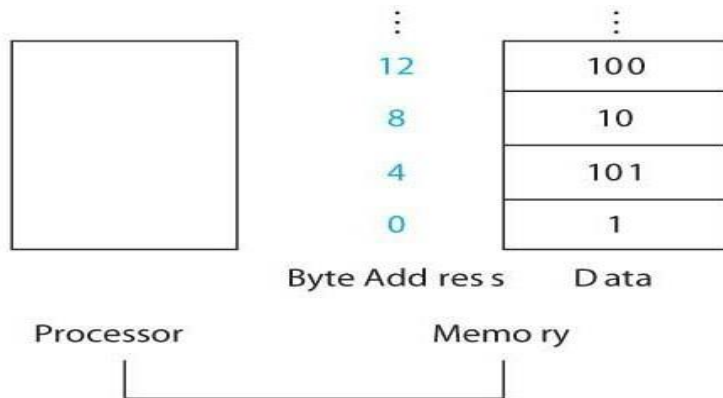
Memory Operands (1)

- 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 Operands (2)

- Memory is byte addressed
 - Each address identifies an 8-bit byte
- Words are aligned in memory
 - Address must be a multiple of 4
- MIPS is Big Endian
 - Most-significant byte at least address of a word
 - c.f. Little Endian: least-significant byte at least address



Memory Operands (3)

- Data is transferred between memory and register using data transfer instructions: lw and sw

Category	Instruction	Example	Meaning	Comments
Data transfer	load word	lw \$s1, 100(\$s2)	$\$s1 \leftarrow \text{memory}[\$s2+100]$	Memory to Register
	store word	sw \$s1, 100(\$s2)	$\text{memory}[\$s2+100] \leftarrow \$s1$	Register to memory

- \$s1 is receiving register
- \$s2 is base address of memory, 100 is called the offset, so (\$s2+100) is the address of memory location

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 of 32
 - 4 bytes per word

lw \$t0, 32(\$s3) # load word add

\$s1 \$s2 \$t0

offset base register

The diagram illustrates the components of the MIPS instruction `lw $t0, 32($s3)`. It shows three registers: `$s1`, `$s2`, and `$t0`. The register `$t0` is the destination register. The register `$s3` is the base register, which is the address of array `A`. The value `32` is the offset, which is the index `8` multiplied by the word size of 4 bytes. The comment `# load word add` indicates the operation being performed.

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
sw $t0, 48($s3) # store word
```

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!

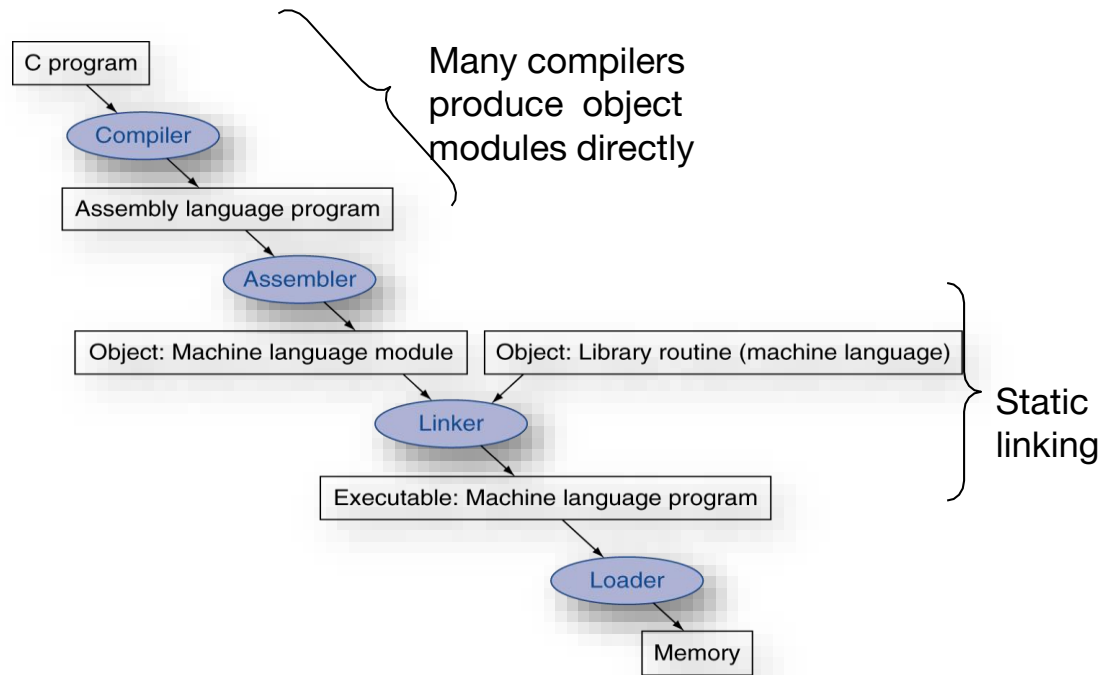
Immediate Operands

- 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 between registers
 - add \$t2, \$s1, \$zero

Translation and Startup



- UNIX: C source files are named x.c, assembly files are x.s, object files are named x.o, statically linked library routines are x.a, dynamically linked library routes are x.so, and executable files by default are called a.out.
- MS-DOS uses the .C, .ASM, .OBJ, .LIB, .DLL, and .EXE to the same effect.

Translation

- Assembler (or compiler) translates program into machine instructions
- Linker produces an executable image
- Loader loads from image file on disk into memory

SPIM Simulator

- SPIM is a software simulator that runs assembly language programs
- SPIM is just MIPS spelled backwards
- SPIM can read and immediately execute assembly language files
- Two versions for different machines
 - Unix xspim(used in lab), spim
 - PC/Mac: QtSpim
- Resources and Download
 - <http://spimsimulator.sourceforge.net>

System Calls in SPIM

- SPIM provides a small set of system-like services through the system call (syscall) instruction.
- Format for system calls
 - Place value of input argument in \$a0
 - Place value of system-call-code in \$v0
 - Syscall

System Calls

Example: print a string

```
.data
str:
.asciiz "answer
is:"

.text
addi
$v0,$zero,4 la
$a0, str syscall
```

Service	System Call Code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_character	11	\$a0 = character	
read_character	12		character (in \$v0)
open	13	\$a0 = filename,	file descriptor (in \$v0)
		\$a1 = flags, \$a2 = mode	
read	14	\$a0 = file descriptor,	bytes read (in \$v0)
		\$a1 = buffer, \$a2 = count	
write	15	\$a0 = file descriptor,	bytes written (in \$v0)
		\$a1 = buffer, \$a2 = count	
close	16	\$a0 = file descriptor	0 (in \$v0)
exit2	17	\$a0 = value	

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one

- Pseudoinstructions: figments of the assembler's imagination

~~move \$s0, \$t1~~ → ~~add \$zero, \$t1~~
~~\$t0, \$t1, L~~ \$t0, \$t0, \$t1
 → slt \$at, \$zero, L
 bne \$at,

- \$at (Register 1): assembler temporary

Assembler Pseudoinstructions (2)

- Pseudoinstructions give MIPS a richer set of assembly language instructions than those implemented by the hardware.
- Register, \$at (assembler temporary), reserved for use by the assembler.
- For productivity, use pseudoinstructions to write assembly programs.
- For performance, use real MIPS instructions

Reading

- Read Appendix A.9 for SPIM
- List of Pseudoinstructions can be found on page 235

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
 - Header: contains size and position of pieces of object module
 - Text segment: translated machine instructions
 - Static data segment: data allocated for the life of the program
 - Relocation info: for instructions and data words that depend on absolute location of loaded program
 - Symbol table: global definitions and external refs
 - Debug info: for associating with source code

Linking Object Modules

- Produces an executable file
 - Merges segments
 - Resolves labels (determine their addresses)
 - Patches location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
- But with virtual memory, no need to do this
- Program can be loaded into absolute location in virtual memory space

Linking Object Modules

Object file header			
	Name	Procedure A	
	Text size	100 _{hex}	
	Data size	20 _{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(X)	
	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	
Object file header			
	Name	Procedure B	
	Text size	200 _{hex}	
	Data size	30 _{hex}	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(Y)	
	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	—	
	A	—	

Linking Object Modules

Executable file header		
	Text size	300 _{hex}
	Data size	50 _{hex}
Text segment	Address	Instruction
	0040 0000 _{hex}	lw \$a0, 8000 _{hex} (\$gp)
	0040 0004 _{hex}	jal 40 0100 _{hex}

	0040 0100 _{hex}	sw \$a1, 8020 _{hex} (\$gp)
	0040 0104 _{hex}	jal 40 0000 _{hex}

Data segment	Address	
	1000 0000 _{hex}	(X)

	1000 0020 _{hex}	(Y)

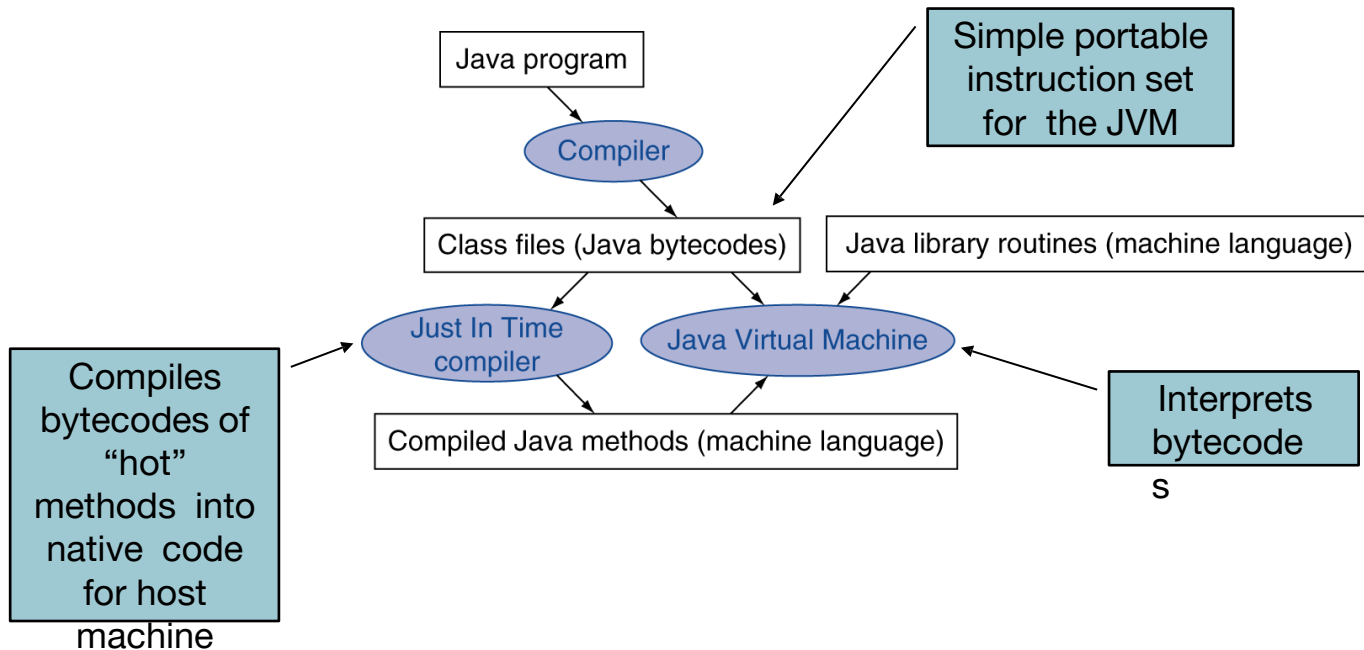
Loading a Program

- Load from file on disk into memory
 - Read header to determine segment sizes
 - Create address space for text and data
 - Copy text and initialized data into memory
 - Set up arguments on stack
 - Initialize registers (including \$sp, \$fp, \$gp)
 - 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 enlarge caused by static linking of all (transitively) referenced libraries
 - Automatically picks up new library versions

Starting Java Applications



An Example MIPS Program

Functional Description: Find the sum of the integers from 1 to N where # N is a value input from the keyboard.

#####

Register Usage: \$t0 is used to accumulate the sum

\$v0 the loop counter, counts down to zero

#####

Algorithmic Description in Pseudocode:

main: v0 << value read from the keyboard (syscall 4) # if (v0 <= 0) stop

t0 = 0; # t0 is used to accumulate the sum # While (v0 > 0) { t0 = t0 + v0; v0 = v0 - 1}

Output to monitor syscall(1) << t0; goto main

#####

.data

prompt: .asciiz "\n\n Please Input a value for N = "

result: .asciiz " The sum of the integers from 1 to N

is "

bye: .asciiz "\n **** Have a good day

An Example MIPS Program(2)

```

        .text
main:    li      $v0, 4          # system call code for print_str
        la      $a0, prompt    # load address of prompt into a0
        syscall                # print the prompt message
        li      $v0, 5          # system call code for read_int
        syscall                # reads a value of N into v0
        blez    $t0, 0         # if ( v0 <= 0 ) go to done
loop:    li      $t0, $t0, $v0   # clear $t0 to zero
        add     $v0, $v0, -1    # sum of integers in register $t0
        addi    $v0, loop       # summing in reverse order
        bnez    $v0, 4         # branch to loop if $v0 is != zero
        li      $a0, result     # system call code for print_str
        la      $v0, 1          # load address of message into $a0
        syscall                # print the string
        li      $a0, $t0        # system call code for print_int
        move     $a0, $t0        # a0 = $t0
done:    syscall                # prints the value in register $a0
        b       $v0, 4          # system call code for print_str
        li      $a0, bye        # load address of msg. into $a0
        la      $v0, 1          # print the string
        syscall                # terminate program
        li      $v0, 10         # return control to system
        syscall

```

Notes

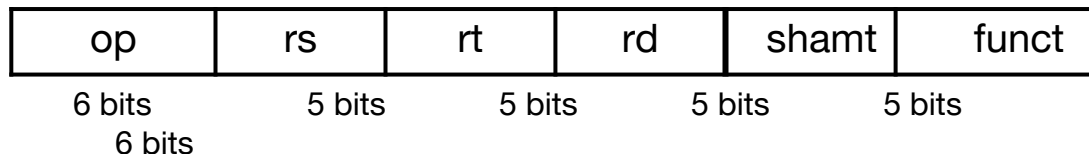
Four Important Number Systems

System	Why?	Remarks
Decimal	Base 10: (10 fingers)	Most used system
Binary	Base 2: On/Off systems	3-4 times more digits than decimal
Octal	Base 8: Shorthand notation for working with binary	3 times less digits than binary
Hex	Base 16	4 times less digits than binary

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

$$00000010001100100100000000100000_2 = 02324020_{16}$$

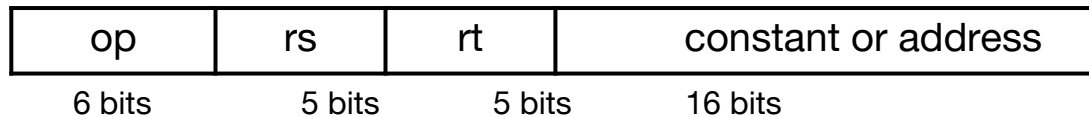
Hexadecimal

- Base 16
 - Compact representation of bit strings
 - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
 - 1110 1100 1010 1000 0110 0100 0010 0000

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 I-format Example

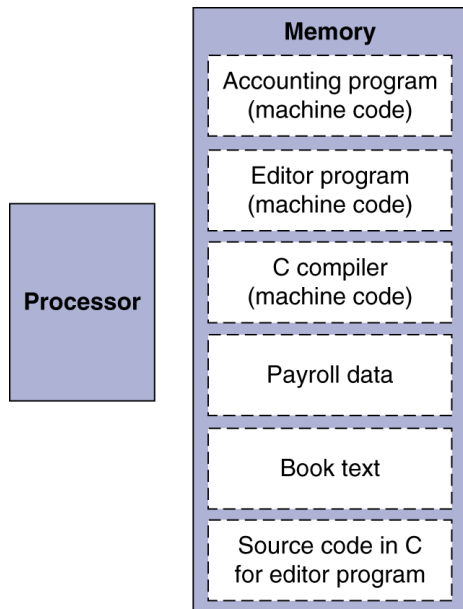
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

lw \$t0, 32(\$s3) # Temporary reg \$t0 gets A[8]

lw	\$s3	\$t0	address
6 bits	5 bits	5 bits	16 bits
35	19	8	32
6 bits	5 bits	5 bits	16 bits
100011	10011	01000	0000000000100000
6 bits	5 bits	5 bits	16 bits

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

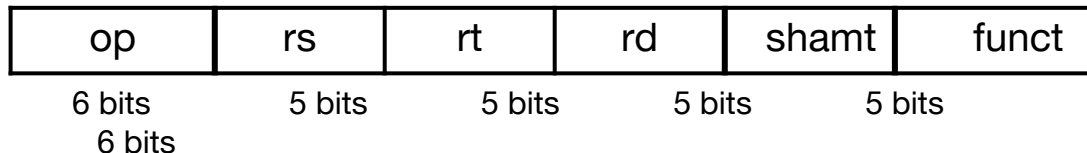
Logical Operations

- 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



- 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

	0000 0000 0000 0000 000	0 110	1 1100 0000
\$t2	0000 0000 0000 0000 001	1	0 0000 0000
\$t1		110	
\$t0	0000 0000 0000 0000 000	0 110	0 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 000	0 110	1 1100 0000
\$t1	0000 0000 0000 0000 00	11 110	0 0000 0000
\$t0	0000 0000 0000 0000 00	11 110	1 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

\$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, h in $\$s0, \$s1, \$s2$

- Compiled MIPS

code: bne $\$s3, \$s4$, Else

add $\$s0, j$ $\$s1, \$s2$

Exit sub $\$s1, \$s2$

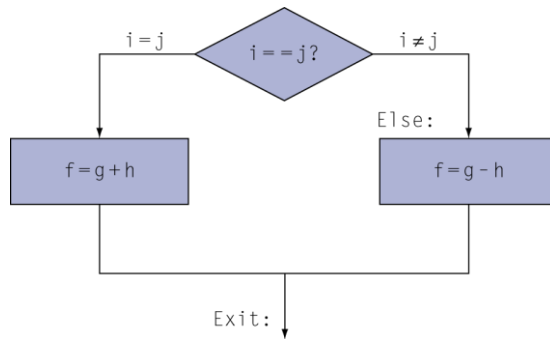
Else:

Exit:

$\$s0,$

...

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

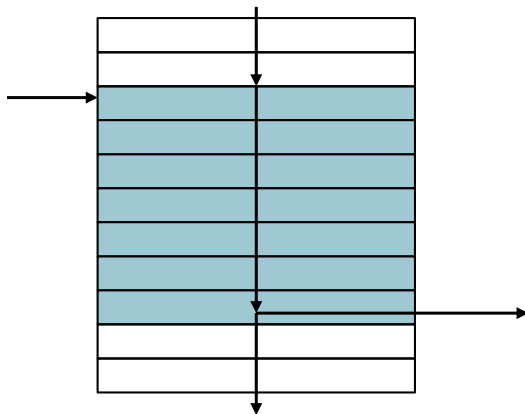
- Compiled MIPS code:

```
Loop:      sll      $t1,    $s3,    2
           add      $t1,    $t1,    $s6
           lw       $t0,    0($t1)
           bne      $t0,    $s5, Exit
           addi     $s3,    $s3,    1
           beq     $t1, $s2, 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

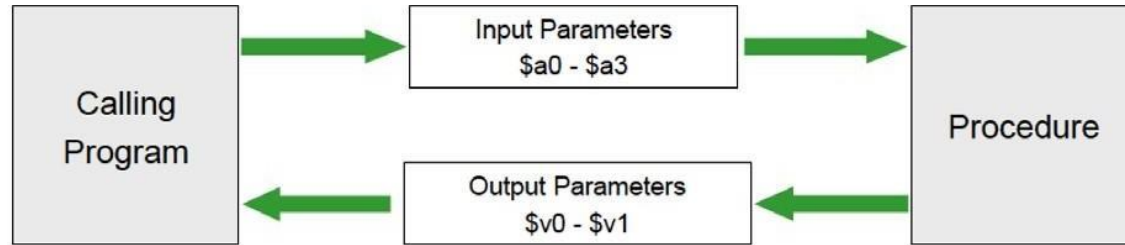
- Why not blt, bge, etc?
- Hardware for $<$, \geq , ... slower than $=$, \neq
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

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

- Procedure (function) performs a specific task and returns results to caller.



Procedure Calling

- Calling program
 - Place parameters in registers \$a0 - \$a3
 - Transfer control to procedure
- Called procedure
 - Acquire storage for procedure, save values of required register(s) on stack \$sp
 - Perform procedure's operations, restore the values of registers that it used
 - Place result in register for caller \$v0 - \$v1
 - Return to place of call by returning to instruction whose address is saved in \$ra

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 for dynamic data (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 (2)

■ MIPS

leaf_example:

addi \$sp, \$sp, -4

Save \$s0 on stack

sw \$s0, 0(\$sp)

Procedure

add \$t0, \$a0, \$a1

body Result

a \$t1, \$a2, \$a3

d \$s0, \$t0, \$t1

Restore \$s0

d

Return

su

b

Leaf Procedure Example (3)

- MIPS code for calling function:

```
main:  
  
...  
jal leaf_example  
...
```

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 (2)

- C code:

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

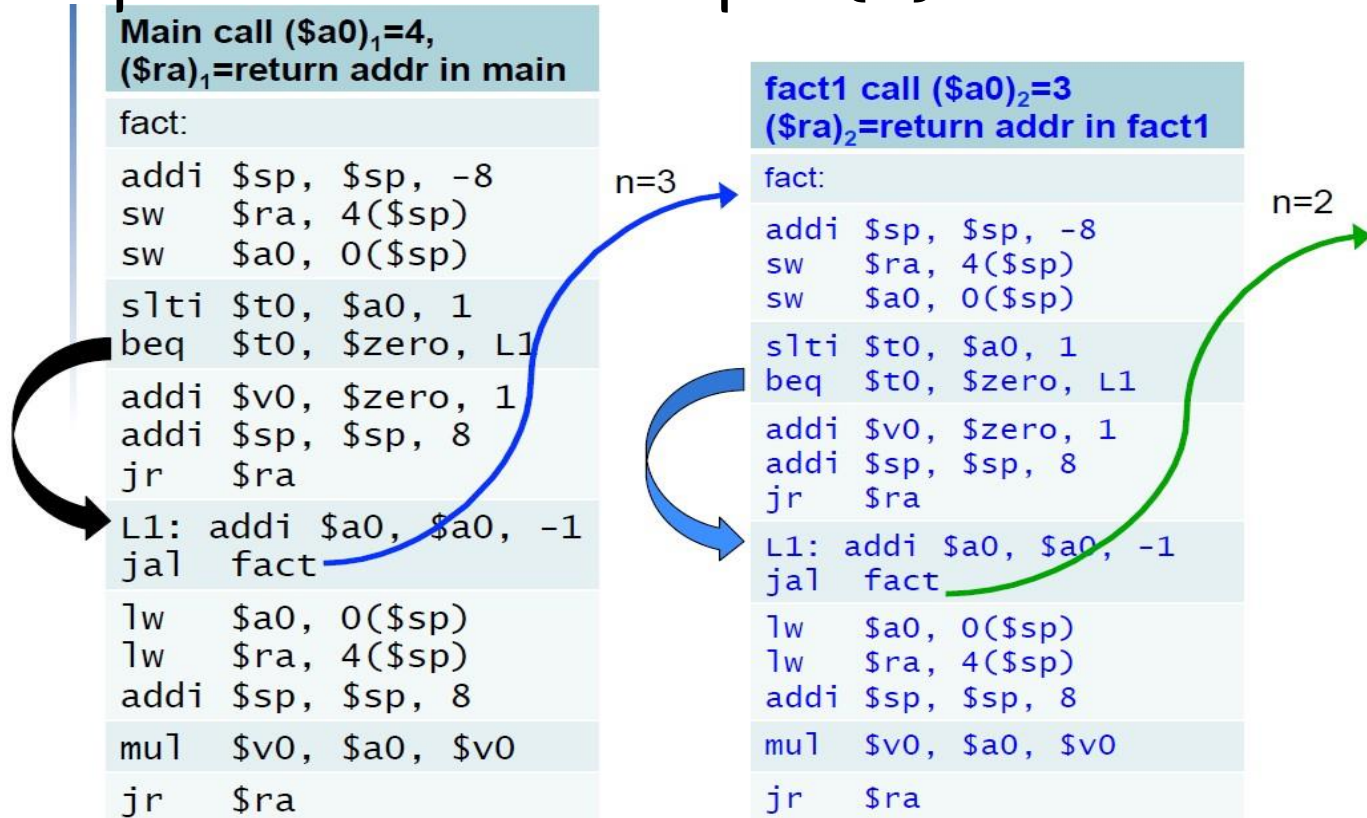
- Argument n in \$a0
- Result in \$v0

Non-Leaf Procedure Example (3)

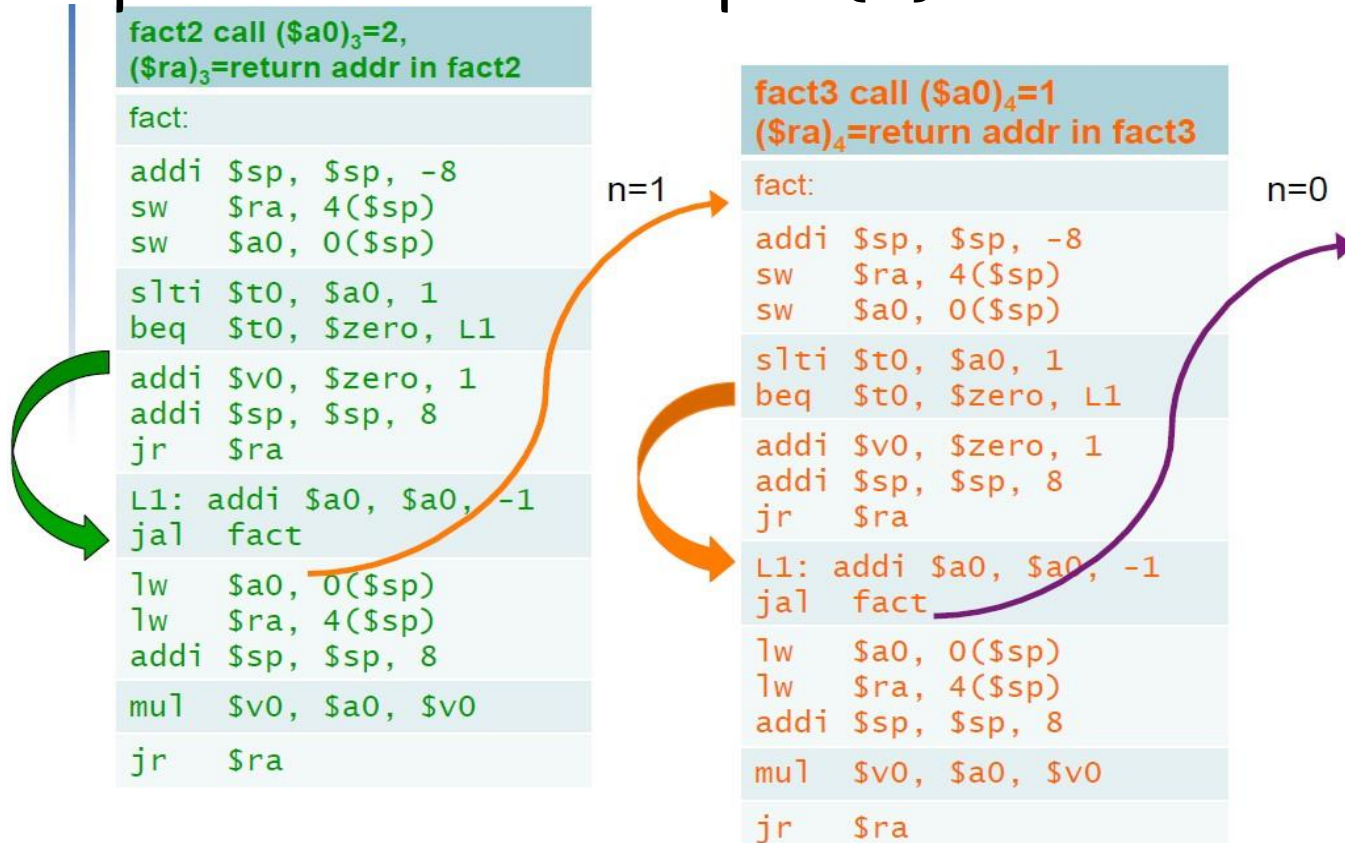
■ MIPS

fact:				
	add	\$sp,	\$sp, -8	# adjust stack for 2 items
i	\$ra,	4(\$sp)		# save return address
s	\$a0,	0(\$sp)		# save argument
w				
s				
w				
	slti	\$t0,	\$a0, 1	# test for n < 1
	beq	\$t0,	\$zero,	L1
	add	\$v0,	\$zero,	1 # if so, result is 1
i	\$sp,	\$sp,	8	# pop 2 items from stack
ad	\$ra			# and return
di				
jr				
L1:	addi	\$a0,	\$a0, -1	# else decrement n
	jal	fact		# recursive call

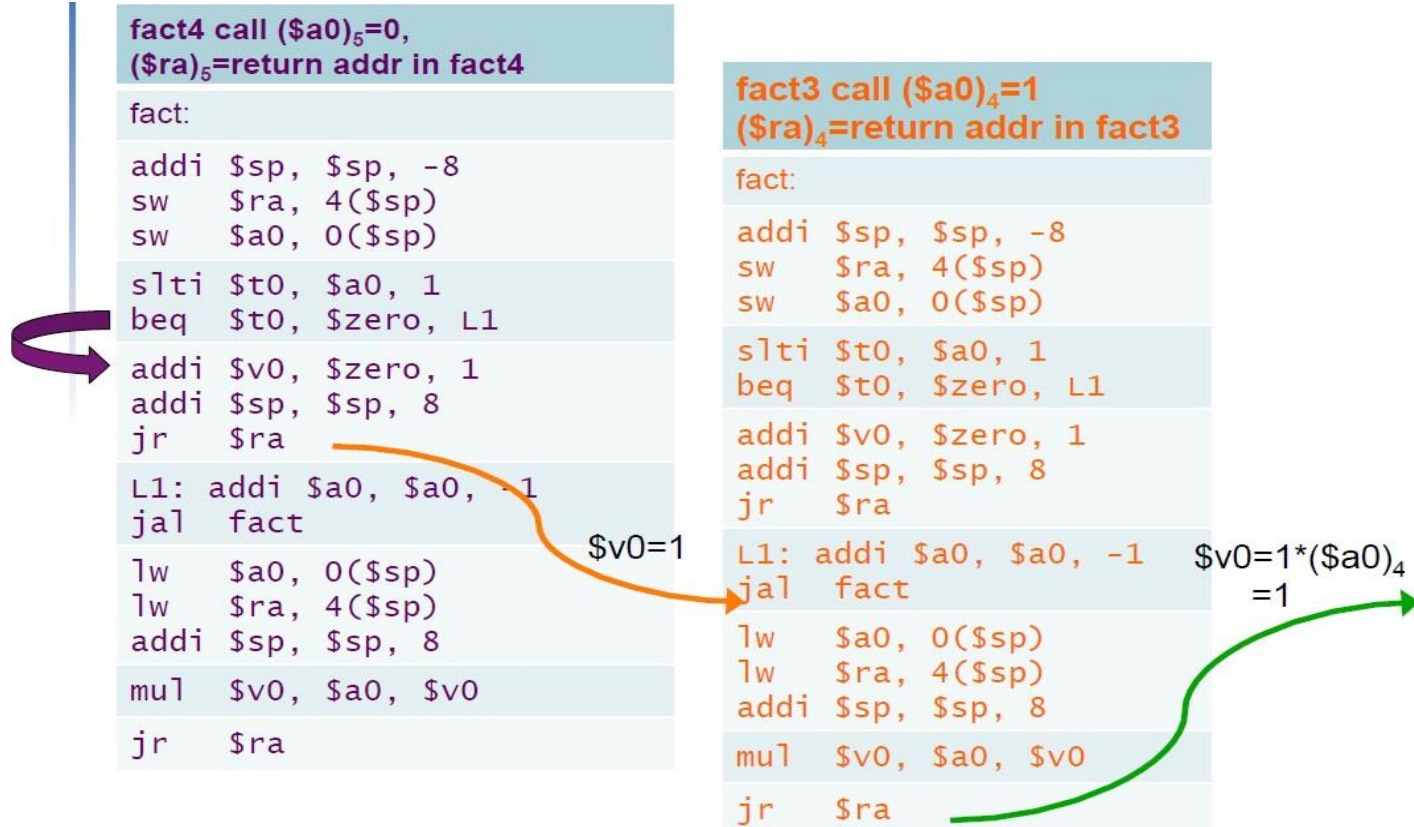
Non-Leaf Procedure Example (4)



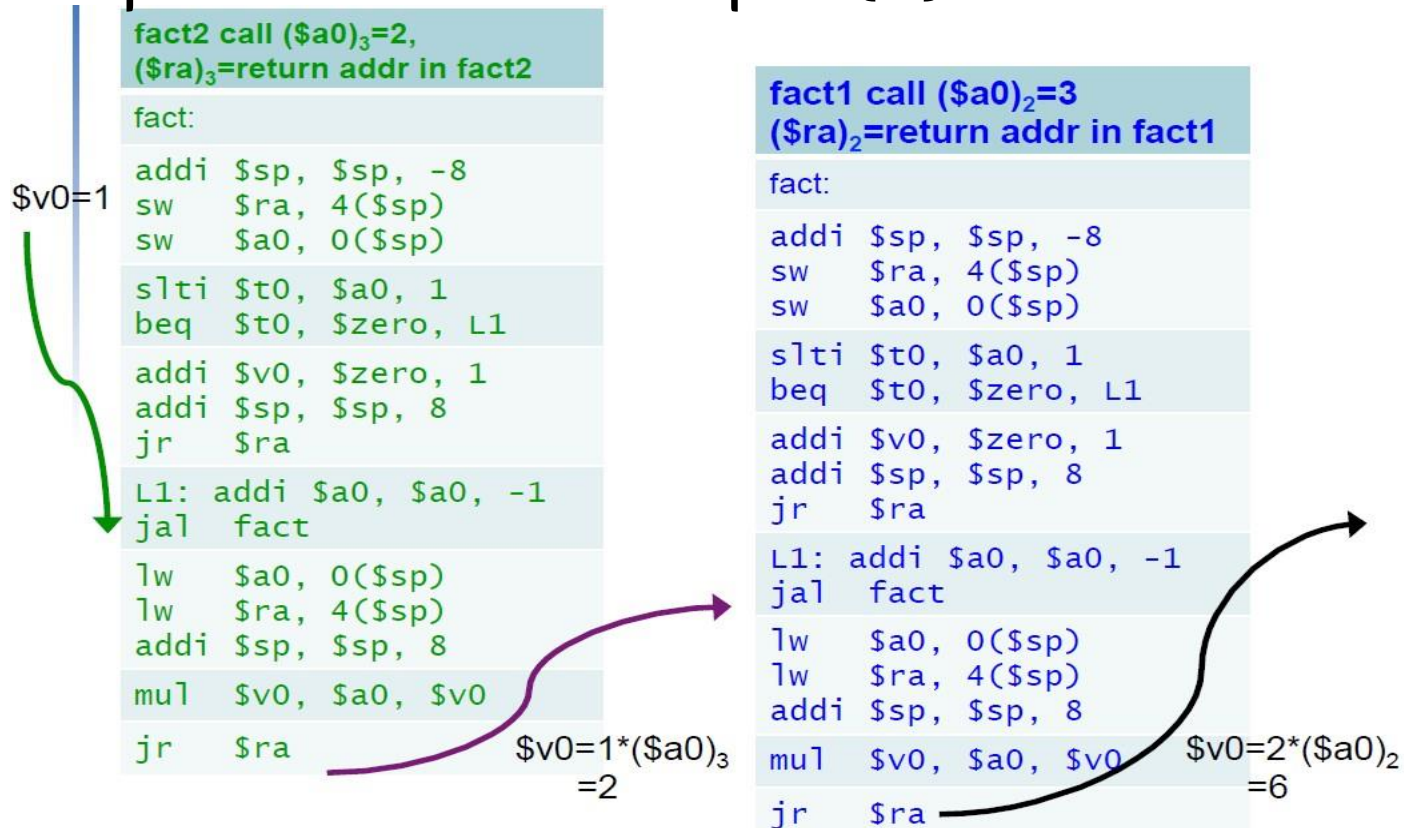
Non-Leaf Procedure Example (5)



Non-Leaf Procedure Example (6)




Non-Leaf Procedure Example (7)



Non-Leaf Procedure Example (8)

\$v0 = 6



Main call (\$a0)₁=4,
(\$ra)₁=return addr in main

fact:

```
addi $sp, $sp, -8
sw    $ra, 4($sp)
sw    $a0, 0($sp)
```

```
slli $t0, $a0, 1
beq   $t0, $zero, L1
```

```
addi $v0, $zero, 1
addi $sp, $sp, 8
jr    $ra
```

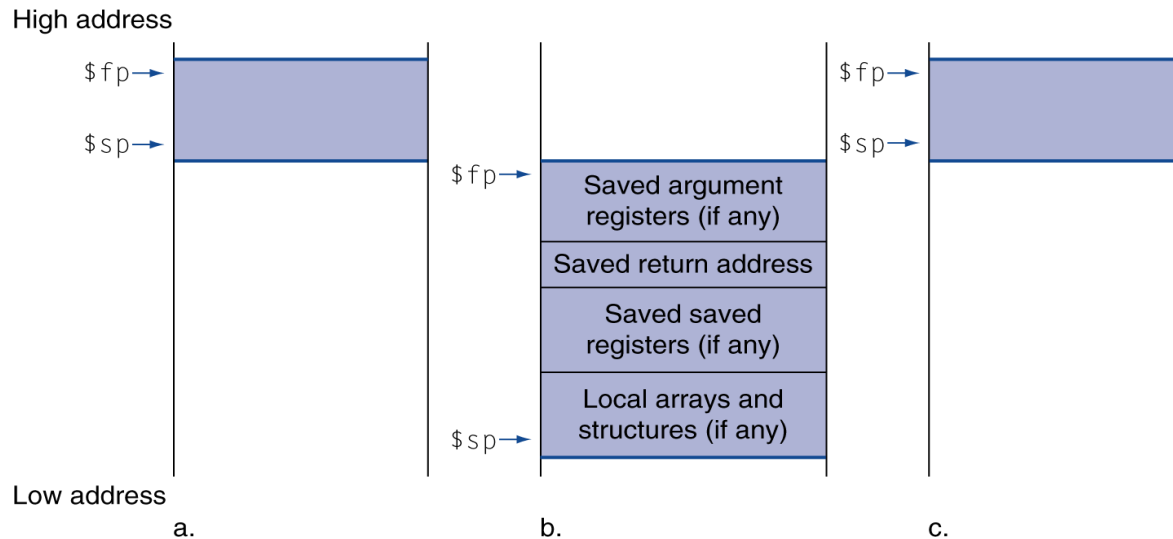
```
L1: addi $a0, $a0, -1
jal   fact
```

```
lw    $a0, 0($sp)
lw    $ra, 4($sp)
addi  $sp, $sp, 8
```

```
mul   $v0, $a0, $v0      # $v0 = 6 * ($a0)1 = 24
```

```
jr    $ra
```

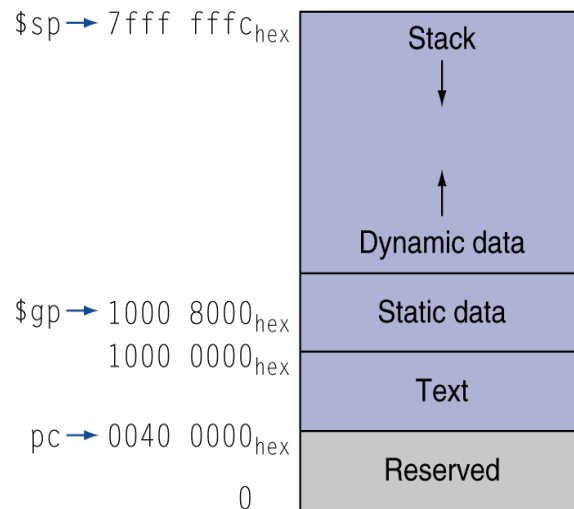
Local Data on the Stack



- 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



Register Summary

- The following registers are preserved on call

Register Number	Mnemonic Name	Conventional Use	Register Number	Mnemonic Name	Conventional Use
\$0	zero	Permanently 0	\$24, \$25	\$t8, \$t9	Temporary
\$1	\$at	Assembler Temporary (reserved)	\$26, \$27	\$k0, \$k1	Kernel (reserved for OS)
\$2, \$3	\$v0, \$v1	Value returned by a subroutine	\$28	\$gp	Global Pointer
\$4-\$7	\$a0-\$a3	Arguments to a subroutine	\$29	\$sp	Stack Pointer
\$8-\$15	\$t0-\$t7	Temporary (not preserved across a function call)	\$30	\$fp	Frame Pointer
\$16-\$23	\$s0-\$s7	Saved registers (preserved across a function call)	\$31	\$ra	Return Address

Character Data

- Byte-encoded character sets
 - ASCII: (7-bit) 128 characters
 - 95 graphic, 33 control
 - Latin-1: (8-bit) 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

ASCII Representation of Characters

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

ASCII Characters

- American Standard Code for Information Interchange (ASCII).
- Most computers use 8-bit to represent each character. (Java uses Unicode, which is 16-bit).
- Signs are combination of characters.
- How to load a byte?
 - lb, lbu, sb for byte (ASCII)
 - lh, lhu, sh for half-word instruction (Unicode)

Byte/Halfword Operations

- Could use bitwise 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:

- 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

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]
	be \$t2, \$zero, L2	#	exit loop if y[i] == 0
	q		
	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

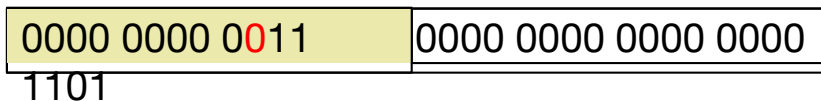
32-bit Constants

- Most constants are small
 - 16-bit immediate is sufficient
- For the occasional 32-bit constant

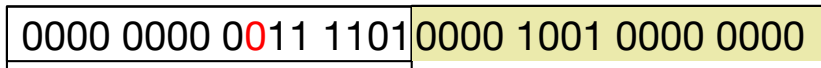
lui rt, constant

- Copies 16-bit constant to left 16 bits of rt
- Clears right 16 bits of rt to 0

lui \$s0,61



ori \$s0,\$s0,2304



Branch Addressing

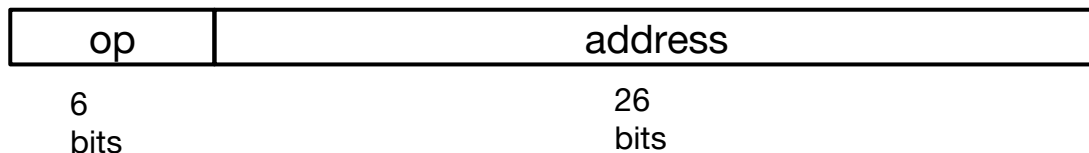
- Branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - Forward or backward



- PC-relative addressing
 - Target address = $PC + \text{offset} \times 4$
 - 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



- PseudoDirect jump addressing
 - Target address = $\text{PC}_{31 \dots 28} : (\text{address} \times 4)$
32 bits = 4 bits 28 bits

Target Addressing Example

- Loop code from earlier example

- Assume Loop at location 80000

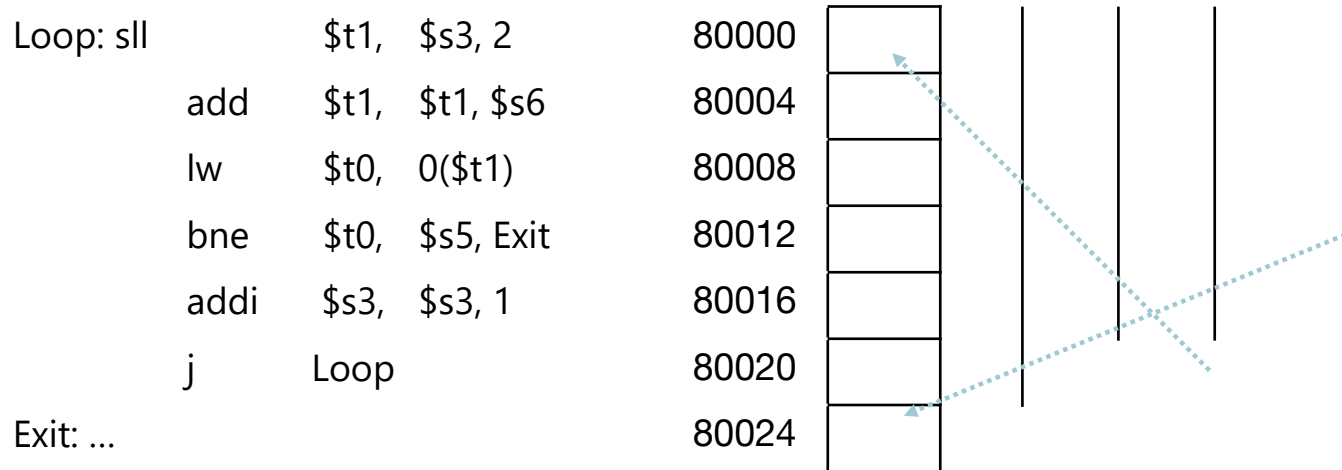
```

Loop: sll      $t1, $s3, 2      80000
      add     $t1, $t1, $s6     80004
      lw      $t0, 0($t1)       80008
      bne     $t0, $s5, Exit    80012
      addi    $s3, $s3, 1       80016
      j       Loop             80020
Exit: ...
      
```

0	0	19	9	4	0
0	9	22	9	0	32
35	9	8		0	
5	8	21		2	
8	19	19		1	
2				20000	

Bài tập

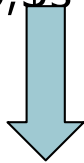
- Dịch thành 6 lệnh mã mã: nhị phân và thập lục phân



Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

beq \$s0,\$s1, L1



written as

bne \$s0,\$s1, L2 j L1

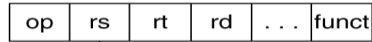
L2: ...

Addressing Mode Summary

1. Immediate addressing



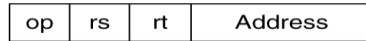
2. Register addressing



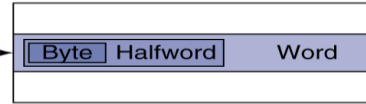
Registers

Register

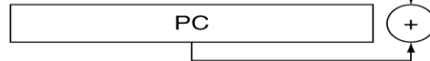
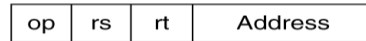
3. Base addressing



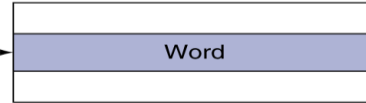
Memory



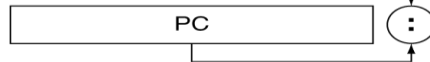
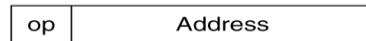
4. PC-relative addressing



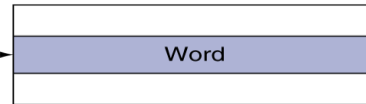
Memory




5. Pseudodirect addressing



Memory



Synchronization (Parallelism)

- Two processors sharing an area of memory
 - P1 writes, then P2 reads
 - Data race if P1 and P2 don't synchronize
 - Result depends on order of accesses
- Hardware support required
 - Atomic read/write memory operation
 - No other access to the location allowed between the read and write
- Could be a single instruction
 - E.g., atomic swap of register  memory
 - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: ll rt, offset(rs)
- Store conditional: sc rt, offset(rs)
 - Succeeds if location not changed since the ll
 - Returns 1 in rt
 - Fails if location is changed
 - Returns 0 in rt

- Example: atomic swap (to test/set lock variable)

```
ld $t0,$zero,$s4 ll      ;copy exchange value  
                        $t1,0($s1) sc ;load linked  
                        $t0,0($s1) ;store conditional  
beq $t0,$zero,try add    ;branch store fails  
$s4,$zero,$t1           ;put load value in $s4
```

C Sort Example

- 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

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
- MIPS: typical of RISC ISAs
 - c.f. x86