

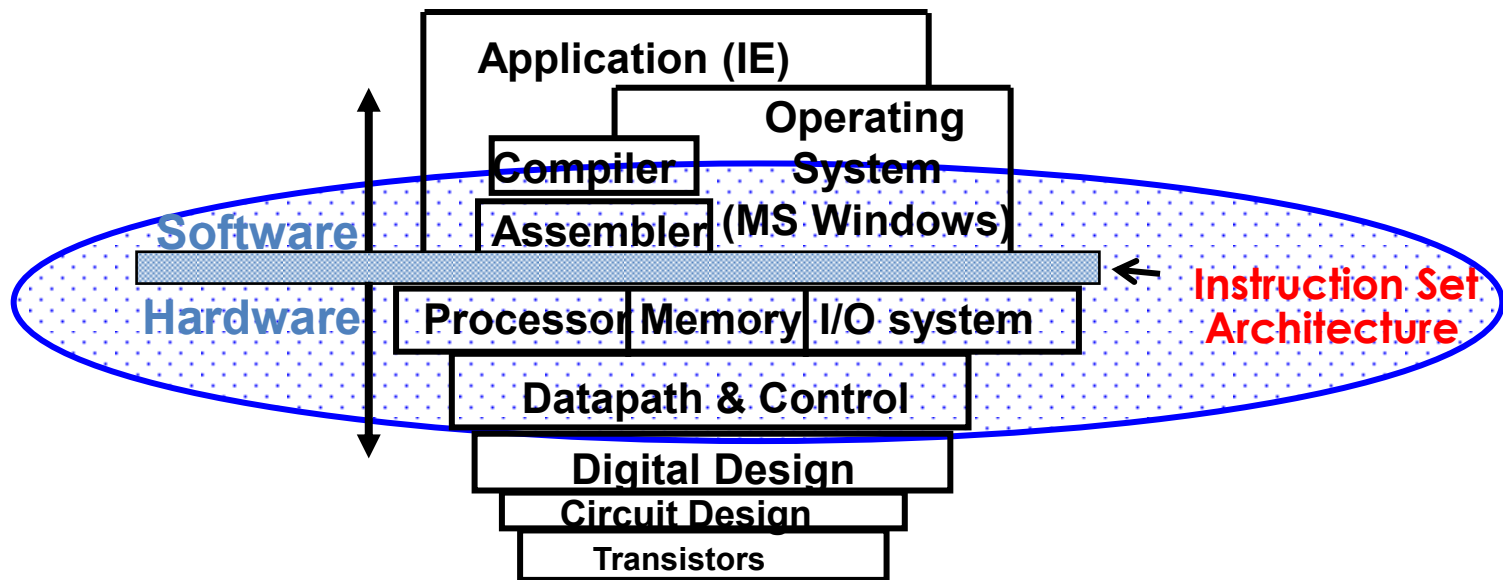
Chapter 2

Instructions: Language of the Computer

Da-Wei Chang, OSES Lab.
CSIE Dept., NCKU

Instruction Set

- Instruction Set : set of instructions of a computer
- **Interface** between hardware and software of a computer
- Different computers have different instruction sets
 - But with many aspects **in common**



The MIPS Instruction Set

- Used as the example throughout the book
- 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 Appendixes A and E

Instructions
in Chapter 2

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)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \mid \$s3)$	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	$\$s1 = \$s2 \mid 20$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	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 ($\$s1 \neq \$s2$) 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	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

Arithmetic Operations

- **Add** and **subtract**, three operands

- Two sources and one destination

add a, b, c ; a gets b + c

sub 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**

Arithmetic Example

- C code:

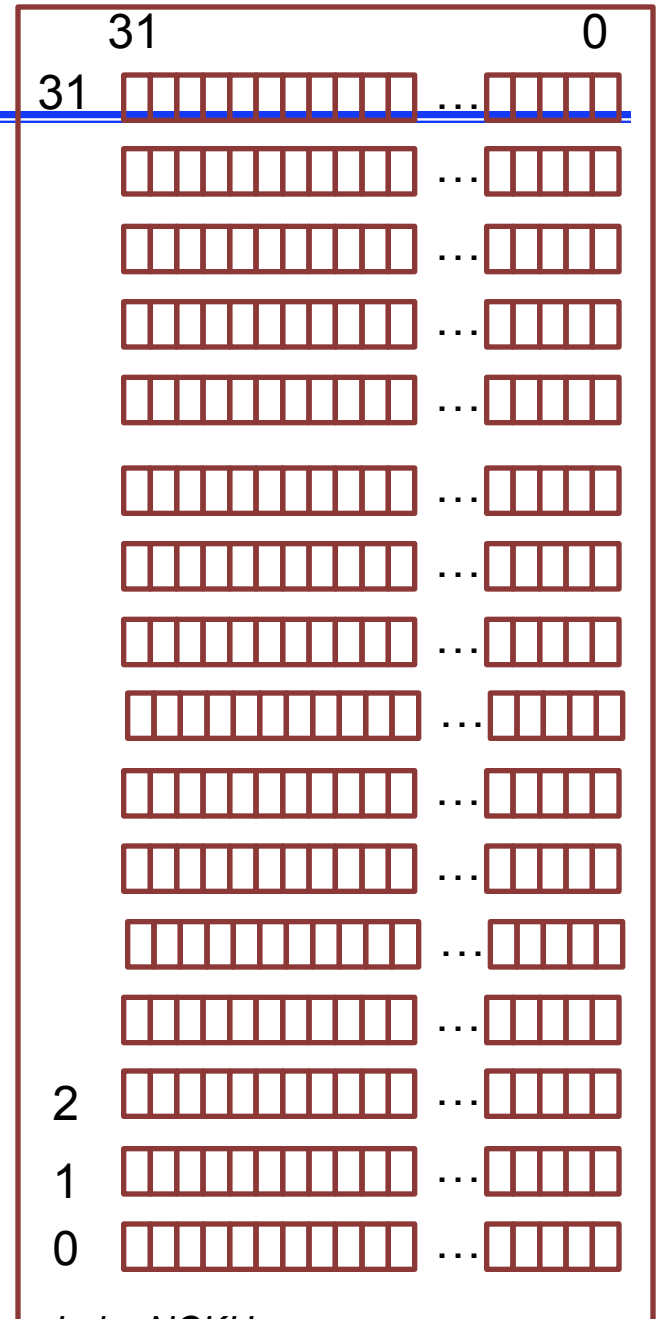
```
f = (g + h) - (i + j);
```

- 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** × 32-bit register file
 - Use for frequently accessed data
 - Registers numbered 0 to 31
 - 32-bit data called a “**word**”
- *Design Principle 2: **Smaller is faster***
 - Smaller register file makes operation fast
 - Much **faster** than main memory (which has millions of locations)



Naming Conventions for Register

- \$t0, \$t1....\$t9 for temporary values
- \$s0, \$s1,...\$s7 for saved variable
- Register 1, called \$at, is reserved for the assembler (see Section 2.12),

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

Register Operand Example

- Find the compiled MIPS code for the following C code:

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

f in \$s0, g in \$s1, h in \$s2,
i in \$s3, j in \$s4

```
add $t0, $s1, $s2 // $t0 = g+h
add $t1, $s3, $s4 // $t1 = i+j
sub $s0, $t0, $t1 // f = (g+h) - (i+j)
```

Memory Operands

- **Main** memory used for composite data
 - Arrays, structures, dynamic data
- 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
- To apply arithmetic operations,
- **Load** values from **memory** into **registers**

```
lw $s1, 20($s2)  
=> $s1=Mem[$s2+20]
```

- **Store** result from **register** to **memory**

```
sw $s1, 20($s2)  
=> Mem[$s2+20]=$s1
```

Address

3	
2	
1	
0	

8				
4				
0				

Aligned word

Address

3	LSB
2	
1	
0	MSB

Big and Little Endian

- MIPS is **Big Endian**
 - **Most-significant byte** at least address of a word
 - *c.f.* Little Endian: least-significant byte at least address
- How the data is **0x12FE34DC** stored in
 1. big endian
 2. little endian

Address

3	DC
2	34
1	FE
0	12

big endian

Address

3	12
2	FE
1	34
0	DC

Little endian

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

\$s3	Base address of A
\$s2	h
\$s1	g
\$s0	

.....

X+32	A[8]
X+12	A[3]
X+8	A[2]
X+4	A[1]
X	A[0]

Memory Operand Example 2

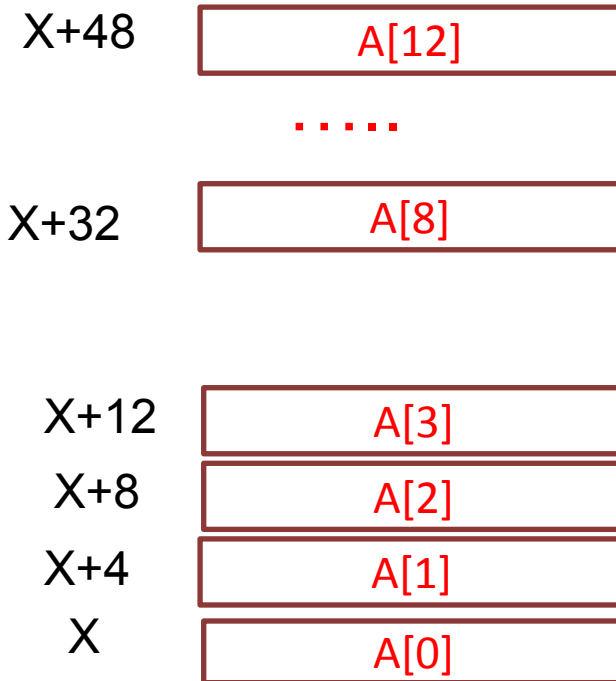
- Convert the following C code to MIPS instruction

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

– *h* in *\$s2*, base address of *A* in *\$s3*

- Compiled MIPS code:

– Index 8 requires offset of 32



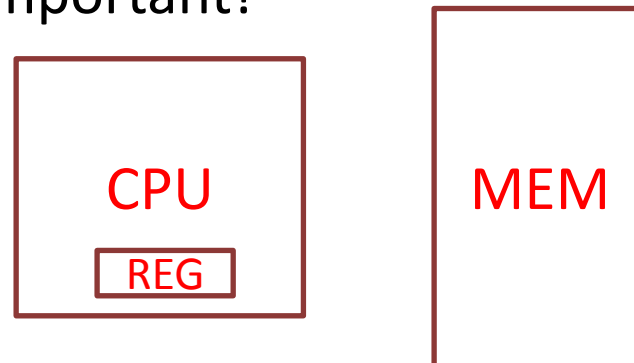
`lw $t0, 32($s3) #load word`

`add $t0, $s2, $t0 # $t0 is a temporary reg.`

`sw $t0, 48($s3)`

Registers vs. Memory

- Registers are **faster** to access than memory
- Operating on **memory** data requires **loads** and **stores**
 - Data are loaded into registers before processed
 - Load/store causes **more** instructions to be executed
 - But no need to process **register and memory data** at the same time => simplified
- Compiler must use **registers** for variables as much as possible
 - Only put **less** frequently used variables to memory
 - **Register** optimization is important!



Immediate Operands

- **Constant** data specified in an **instruction**


```
addi $s3, $s3, 1 => $s3=$s3+1
```

- **No subtract** immediate instruction
 - Just use a negative constant

```
addi $s3, $s3, -1
```

- *Design Principle 3: Make the common case fast*
 - Add 1 and subtract 1 are common
 - Small **constants** are common
 - Immediate operand avoids a **load** instruction

```
lw $s1, 0($s0)
add $s3, $s3, $s1
```



```
addi $s3, $s3, 1
```

The Constant Zero

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

```
add $t2, $s1, $zero # $t2=$s1
```


Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$

- Example

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 32 bits

- 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$

- Example

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Using 32 bits

- $-2,147,483,648$ to $+2,147,483,647$

2s-Complement Signed Integers

- Bit 31 is **sign bit**
 - **1** for negative numbers
 - **0** for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- **Non-negative numbers** have the same unsigned and 2s-complement representation
 - 5 is 101_2 in both unsigned and 2s-complement signed
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Convert n to $-n \Rightarrow$ Complement and add **1**
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$\text{because } x + \bar{x} = 1111 \dots 111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000 \ 0000 \ \dots \ 0010_2$
 - $-2 = 1111 \ 1111 \ \dots \ 1101_2 + 1$
 $= 1111 \ 1111 \ \dots \ 1110_2$

Sign Extension

- Representing a number using **more bits, but still preserve** the numeric value
- Signed values: Replicate the **sign bit** to the left

Examples: 8-bit to 16-bit

+2: 0000 0010 => 0000 0000 0000 0010

-2: 1111 1110 => 1111 1111 1111 1110


- Unsigned values: extend with **0s**


Examples: 8-bit to 16-bit

+2: 0000 0010 => 0000 0000 0000 0010

Why do we need sign extension?

- Because an instruction is 32-bit, the constant or address in the instruction is less than 32-bit. In order to keep the same value when putting the data into register, the constant or address must be signed extended.

1111111110111111 16-bit  0000000000 1111111110111111 32-bit register Wrong

 11111111111111 1111111110111111 Correct

Why do we need sign extension?

- Because an instruction is 32-bit, the constant or address in the instruction is less than 32-bit. In order keep the same value when putting the data into register, the constant or address must remain the same..

```
addi $s3, $s3, 1 => $s3=$s3+1
```



- Examples of MIPS instructions that require sign extension
 - addi: extend immediate value
 - lb, lh: extend loaded byte/halfword (discussed later)
 - beq, bne: extend the displacement (discussed later)

Representing Instructions

- Instructions are encoded in **binary**
 - Called machine code
- MIPS instructions
 - Encoded as **32-bit** instruction words
 - Not many different instruction types
 - Regularity
 - Easier to implement
- Register that are frequently used
 - \$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) => indicate the operation
 - rs: **first source** register number
 - rt: **second source** register number
 - rd: **destination** register number
 - shamt: **shift** amount (00000 for now) => (in Section 2.6)
 - funct: function code (extends **opcode**)=>select the specific variant of the operation in the op field

R-format Example (add, and, ..etc.)

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

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

No
shift

Note:

\$s1=r17

\$s2=r18

\$t0=r8

op	\$s1	\$s2	\$t0	0	add
----	------	------	------	---	-----

0	17 ₁₀	18 ₁₀	8	0	32
---	------------------	------------------	---	---	----

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

0000001000110010010000000100000₂ = 02324020₁₆

Recap: 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 (lw, sw, addi,...,etc.)



- **Immediate** arithmetic and **load/store** instructions
 - **rt**: destination or source register number
 - Constant: -2^{15} to $+2^{15} - 1$ because 4th field is 16-bit
 - Address: offset added to **base** address in **rs**

addi \$s3, \$s3, 1 \$s3=\$s3+1



lw \$t0, 32(\$s3) \$t0=MEM[\$s3+32]



Example

- Translate the following statement into binary code

Opcode: lw:35₁₀, sw:43₁₀
\$t0:r8, \$t1:r9

lw \$t0, 16(\$t1)

op	rs	rt	Address offset
35	9	8	16

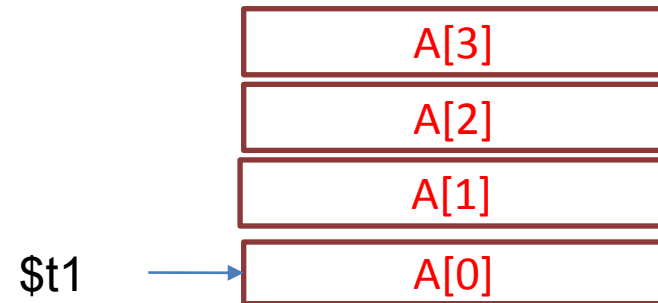
sw \$t0, 16(\$t1)

op	rs	rt	rd	shamt	funct
43	9	8		16	

Example

- Translate the following statement into (1) MIPS instruction (2) binary code, assuming $\$t1$ is the base address of A and $\$s2$ contains h

$$A[3] = h + A[3]$$



Example

- Translate the following statement into (1) MIPS instruction (2) binary code, assuming $\$t1$ is the base address of A and $\$s2$ contains h

$A[3] = h + A[3]$

`lw $t0, 12($t1)`

`add $t0, $s2, $t0`

`sw $t0, 12($t1)`

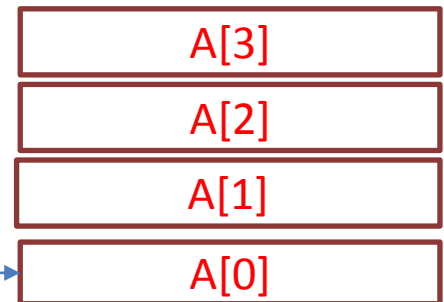
Opcode:

lw:35, sw:43

Add: 0 (funct=32)

$\$t0$:r8,

$\$s2$: r18



op	rs	rt	rd	shamt	funct
35	9	8	12		
0	18	8	8	0	32
43	9	8	12		

Instructions so far

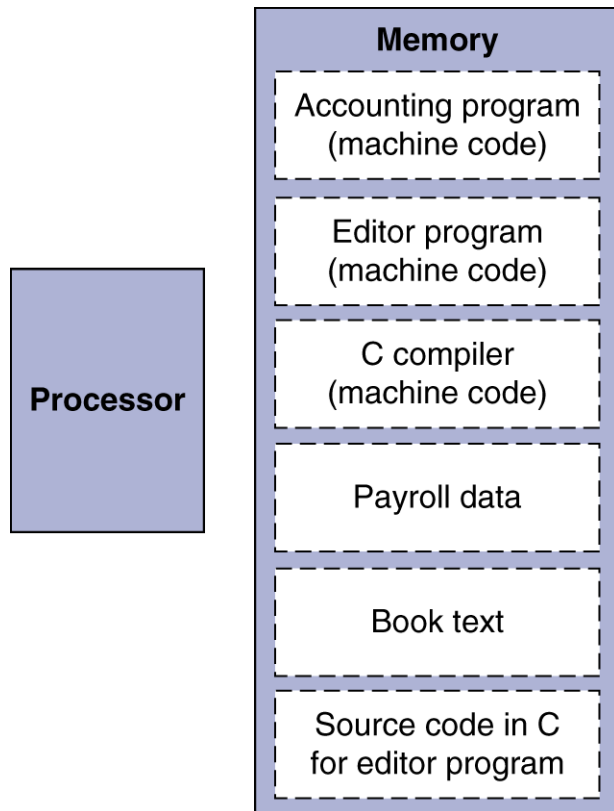
MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2 , \$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2 , \$s3
addi	I	8	18	17	100			addi \$s1,\$s2 , 100
lw	I	35	18	17	100			lw \$s1, 100(\$s2)
sw	I	43	18	17	100			sw \$s1, 100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.6 MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; the *shamt* field, which Section 2.6 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field. Copyright © 2009 Elsevier, Inc. All rights reserved.

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

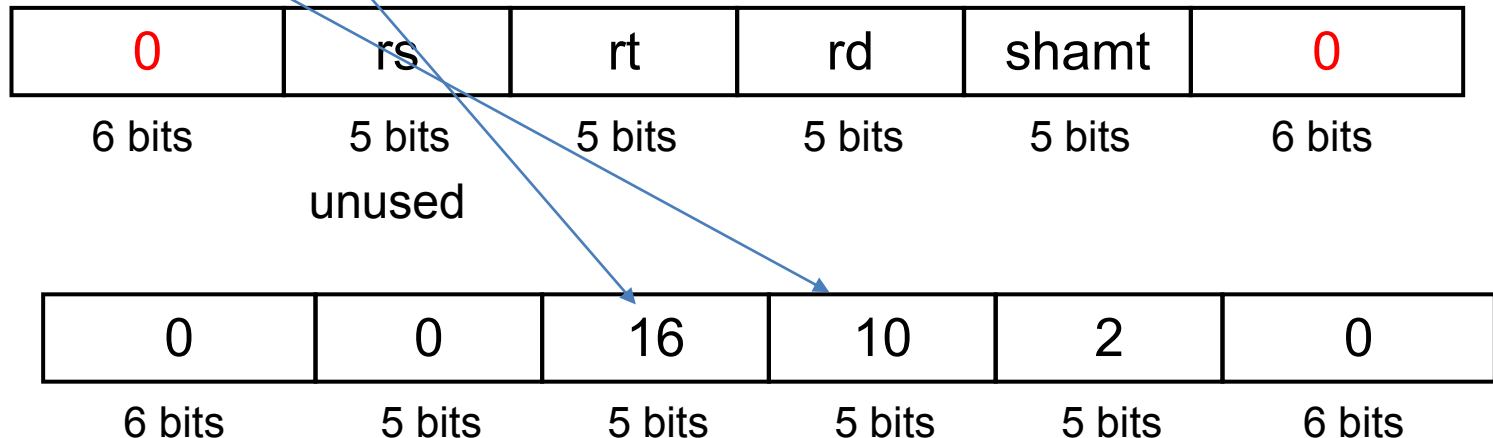
Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

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

Shift Operations

- Shift **left** logical
 - Shift left and fill with 0 bits
 - `sll` by i bits = **multiplies** by 2^i (`00000011 << 2 => 00001100`)
- **Instruction format for `sll`: op:0, funct: 0**
- **`shamt`**: how many **positions** to shift

`sll $t2, $s0, 2 # reg $t2 = reg $s0 << 2bits`

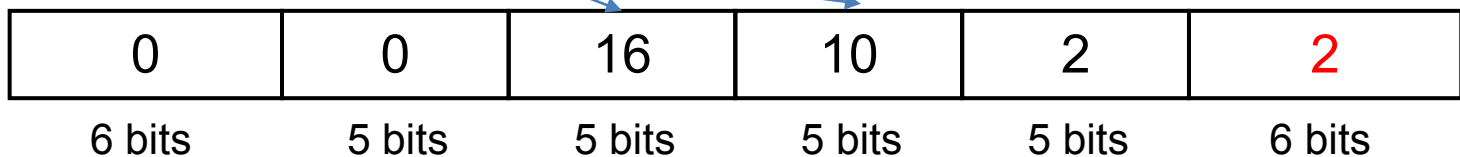


Shift Operations

- Shift **right** logical (srl)
 - Shift right and fill with 0 bits
 - srl by i bits = **divides** by 2^i (unsigned only)
- **Instruction format for srl: op:0, funct: 2**
- **shamt**: how many **positions** to shift



srl \$t2, \$s0, 2 # reg \$t2 = reg \$s0 >> 2bits



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

- Instruction format for and: op:0, funct: 100100_2



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

- Instruction format for and: op:0, funct: 100101₂



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

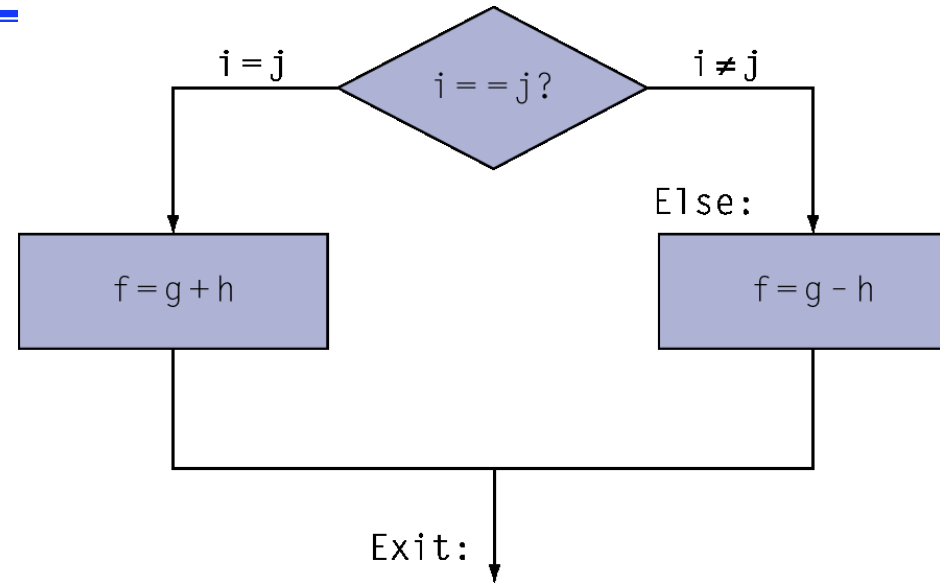
- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq rs, rt, L1`
 - if (`rs == rt`) branch to instruction labeled **L1**;
- `bne rs, rt, L1`
 - if (`rs != rt`) branch to instruction labeled **L1**;
- `j L1`
 - **unconditional** jump to instruction labeled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

Compiling If Statements

- C code:

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

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



- Compiled MIPS code:

```
bne $s3, $s4, Else // if (i!=j) goto ELSE  
add $s0, $s1, $s2 // f= g+h  
j Exit  
Else: sub $s0, $s1, $s2 // f= g-h  
Exit: ...
```

Assembler calculates addresses

Compiling Loop Statements

- Convert the following C code to MIPS instruction, assume **i** is in \$s3, **k** in \$s5, **base address of save** is in \$s6

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

- Compiled MIPS code:

```
Loop:  sll    $t1, $s3, 2  # $t1=i*4
       add    $t1, $t1, $s6  #address of save[i]
       lw     $t0, 0($t1)    # load save[i]
       bne    $t0, $s5, Exit # branch if save[i]!= k
       addi   $s3, $s3, 1
       j      Loop
Exit:  ...
```

\$s3 i

\$s5 k

Save[i]

Save[2]

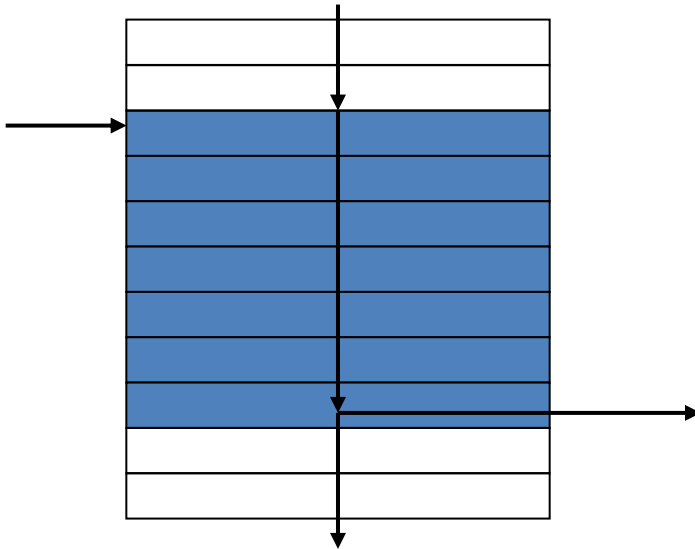
Save[1]

\$s6 → Save[0]

Basic Blocks

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

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



- A compiler identifies **basic blocks** for optimization

More Conditional Operations

- Set result to **1** if a condition is true
 - Otherwise, set to 0
- `slt rd, rs, rt` //set (on) less than
 - if ($rs < rt$) $rd = 1$; else $rd = 0$;
- `slti rt, rs, constant` //immediate mode
 - if ($rs < \text{constant}$) $rt = 1$; else $rt = 0$;
- Used in **combination** with **beq**, **bne** to create relative conditions(equal, not equal, less than....)
- **For example**

<code>if (\$s1 < \$s2)</code>		<code>slt \$t0, \$s1, \$s2</code>
<code>branch to L</code>		<code>bne \$t0, \$zero, L</code>

Why not use a single instruction for **blt** (branch less than), **bge** (branch greater than), etc?

-too complicated; *see the next slide*

Why not blt and bgt

- because hardware for $<$, \geq , ... **slower** than $=$, \neq
 - **Greater or less than** use **2s-complement** subtraction, but $=$ or \neq use xor
 - Combining **$<$ or \geq** with **branch** involves more work per instruction, requiring a slower clock
 - All instructions **penalized!**
 - **beq** and **bne** are the common case, no need to create instructions for blt and bgt
- => a good design compromise

Signed vs. Unsigned Comparison

- **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**

`$t0 = 0 or 1 ??`

`$t0 = 1` because $-1 < +1$

`sltu $t0, $s0, $s1` **# unsigned**

`$t0 = 0 or 1???`

`$t0 = 0` because $+4,294,967,295 > +1$

Supporting Procedures in Computer Hardware

- Steps required to execute a procedure
 1. Place **parameters** in registers, where the procedure can access (in register **\$a0** to **\$a3**)
 2. Transfer control to procedure (using **jal** instruction)
 3. Acquire storage for procedure (**save the registers** that you are going to use)
 4. Perform procedure's operations
 5. Place results in registers for caller (register **v0** and **v1**)
 6. Return to place of call (**restore saved register**, and run **jr \$ra**)

Procedure Calling Steps

- Required Steps

Procedure A

1. Set arguments
(\$a0 ~ \$a3)
2. jal B
add \$t0, \$t2, \$t3

Procedure B

3. Save saved registers
(\$s0 ~ \$s7)
4. Execute specified jobs

.....
.....
.....
5. Set return values (\$v0 and \$v1)
6. Restore saved registers (\$s0 ~ \$s7)
jr \$ra

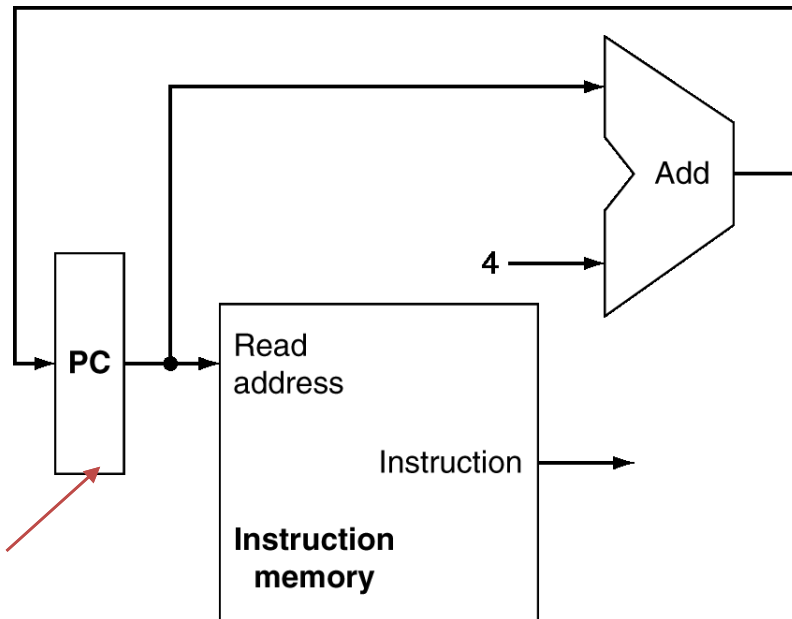
Naming Conventions for Procedure Calling

- **\$a0-\$a3**: four arguments to pass parameters
- **\$v0-\$v1**: registers to store return values
- **\$ra** : return address
- \$t0, \$t1....\$t9 for temporary values
 - Can be **overwritten** by callee without saving
- \$s0, \$s1,...\$s7 for saved variable
 - Must be saved/restored by **callee**

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

Program Counter

- **Program Counter** is used to indicate the **next** instruction to be executed



1000 lw ..
1004 add \$t0
1008 sll ...
100C sll ...
.....

Before add (when
executing lw)
\$pc = 1004

32-bit
register

Procedure Call Instructions

- Procedure call: **jump and link**

jal ProcedureLabel

- Address of the following instruction are in **\$ra**
- Jumps to **target** address

```
int main()
```

```
{  ...  
  t1 = fact(8);  
  t2 = fact(3);  
  t3 = t1 + t2;
```



```
1000 xxx ..  
1004 jal fact  
1008 sll ..  
100C sll ..
```

Before jal

\$pc = 1004

\$ra = XXX

```
...
```

```
}
```

```
int fact(int n)
```

```
{  
  int i, f = 1;  
  for (i = n; i > 1; i--)  
    f = f * i;  
  return f;  
}
```

```
.....
```

```
2010 fact:...  
2014.....  
2018 .....
```

After jal

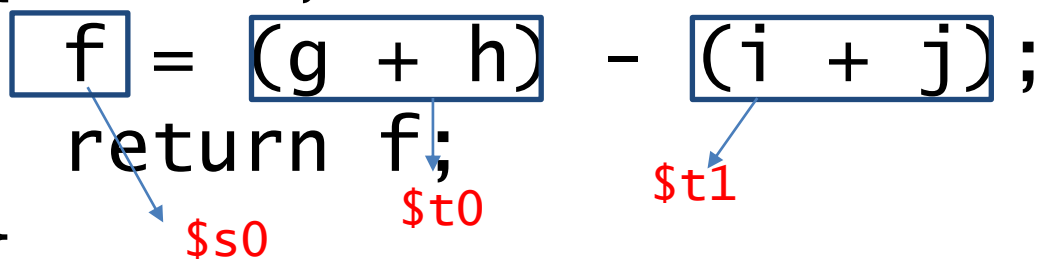
\$pc = 2010

\$ra = 1008

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, 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)

$\$a0$	g
$\$a1$	h
$\$a2$	i
$\$a3$	j
$\$s0$	f

Three register $\$s0, \$t0, \$t1$ are saved used in leaf_example=> need to be saved

```

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

```

$\$s0$ $\$t0$ $\$t1$

Three local variables

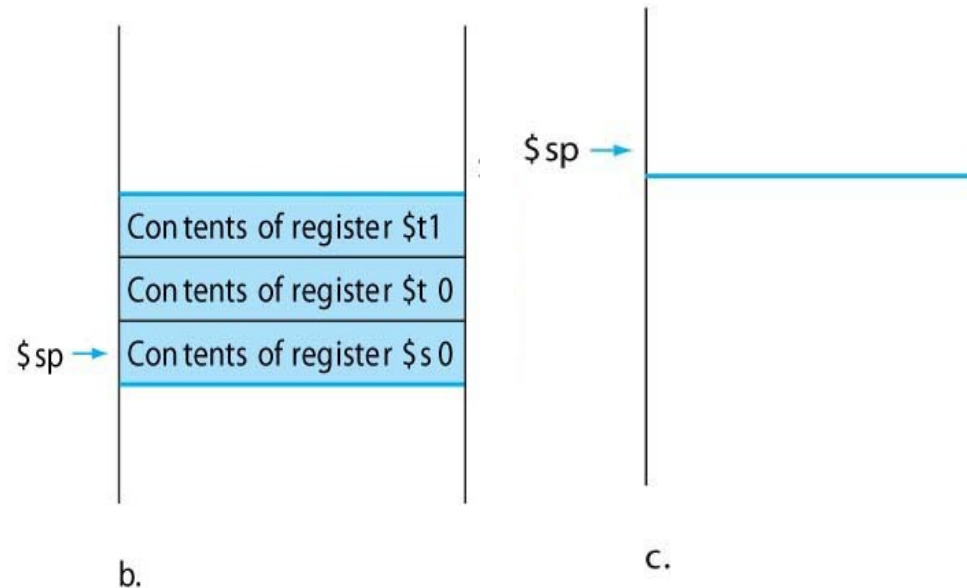
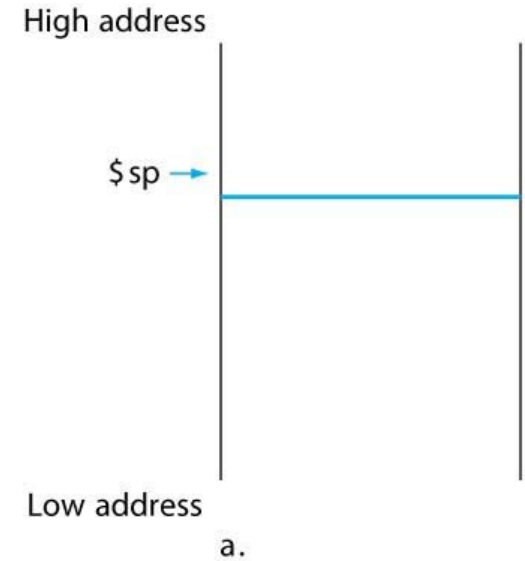
```

leaf_example:
    addi $sp, $sp, -12
    sw   $t1, 8($sp)
    sw   $t0, 4($sp)
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    lw   $t0, 4($sp)
    lw   $t1, 8($sp)
    addi $sp, $sp, 12
    jr   $ra

```

\$a0	g
\$a1	h
\$a2	i
\$a3	j

\$s0	f
\$v0	



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

\$s0 \$t0 \$t1

Improved Version

\$a0	g
\$a1	h
\$a2	i
\$a3	j

\$s0	f
\$v0	

```
leaf_example:
    addi $sp, $sp, -12
--sw--$t1,-8($sp)---
--sw--$t0,-4($sp)---
    sw    $s0, 0($sp)
    add   $t0, $a0, $a1
    add   $t1, $a2, $a3
    sub   $s0, $t0, $t1
    add   $v0, $s0, $zero
    lw    $s0, 0($sp)
-lw--$t0,-4($sp)---
-lw--$t1,-8($sp)---
    addi  $sp, $sp, 12
    jr    $ra
```


Caller maintains \$t0~\$t9
 Callee maintains \$s0~\$s7
 Therefore, **no need to maintain the states of \$t0 and \$t1 !!**

=> 2 sw and 2 lw instructions are reduced

Leaf and Non-Leaf Procedures

- Leaf procedure: one that doesn't call other procedures
- Non-leaf procedure: one that calls **other** procedures (a procedure can be both **caller** and **callee**)


Leaf procedure is a callee



```
int leaf(int arg1, arg2)
{
    int f;

    f = g-h;
    return f;
}
```

both caller and callee

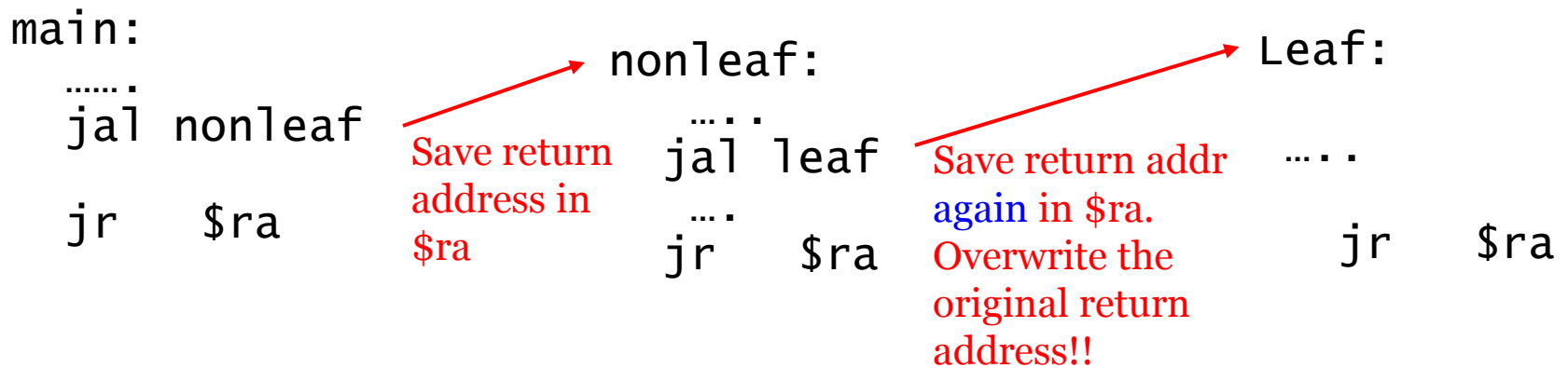
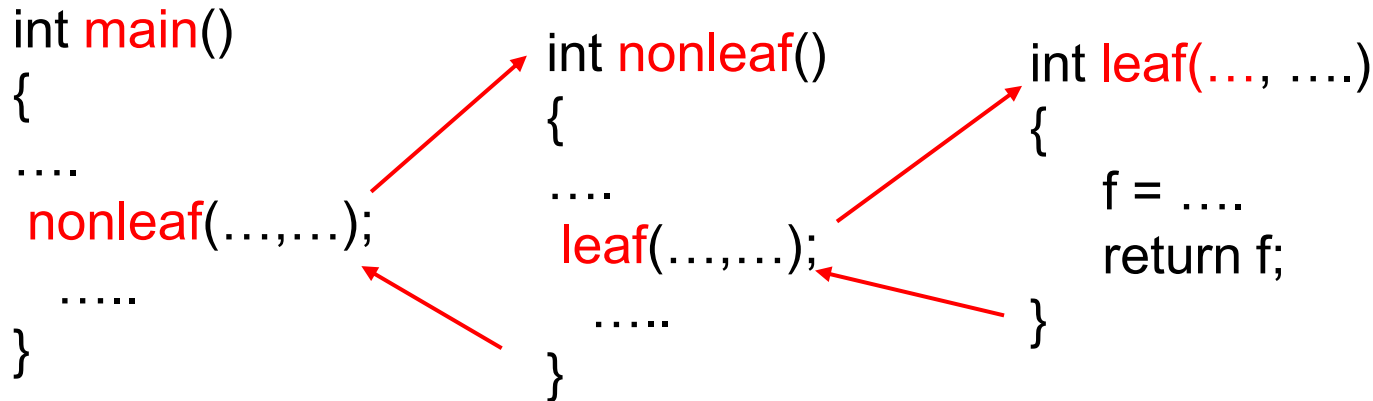


```
int nonleaf()
{
    int x ;

    x =leaf(arg1,arg2);
    return x;
}
```

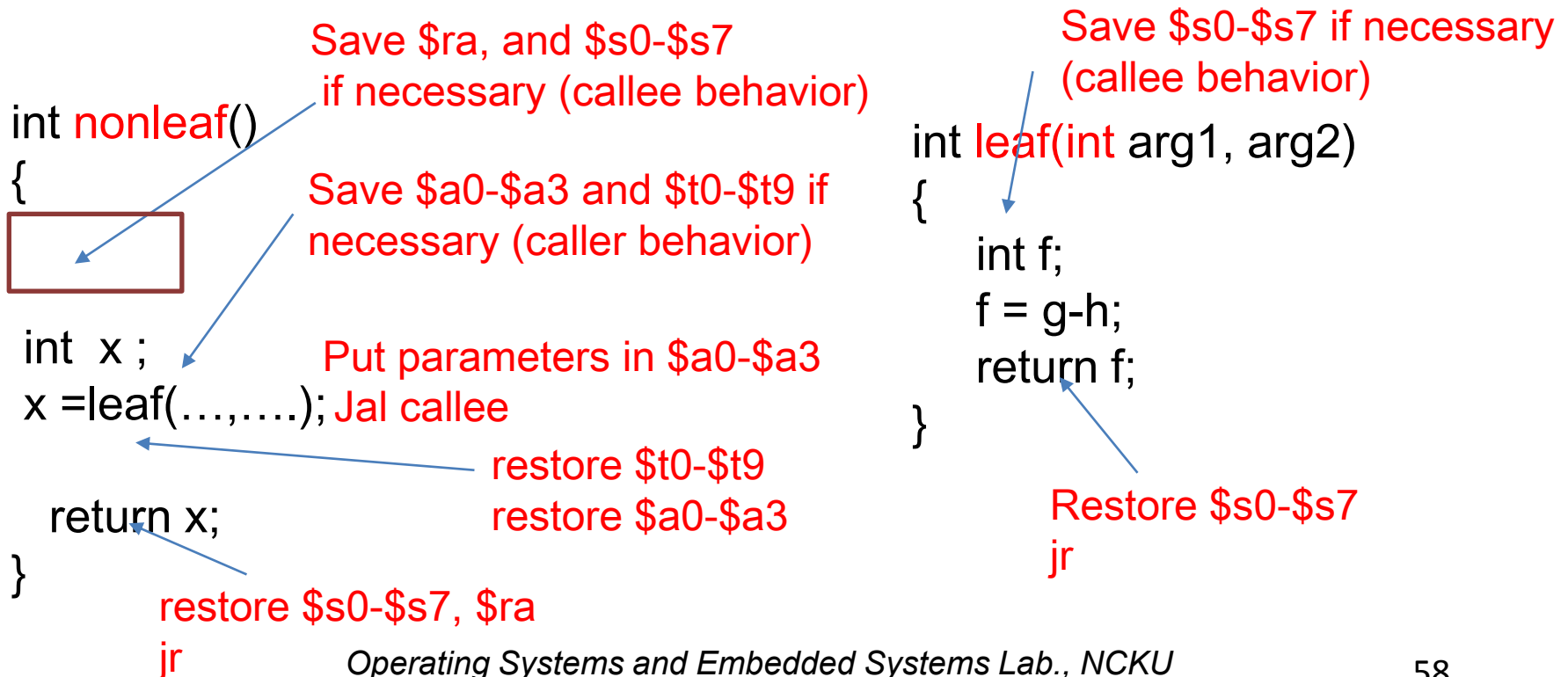
Issues in Non-leaf Procedures

- Additional challenge for non-leaf procedure



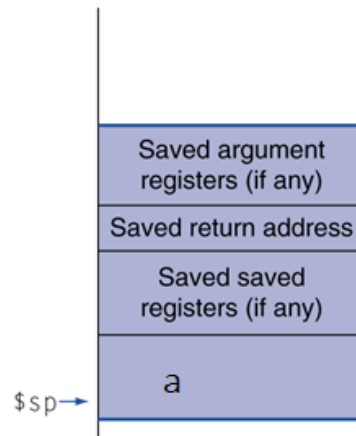
Register State Reservation for Non-leaf Procedures

- Maintain register states by **saving** them before the function executes, and **restoring** them after the function completes.
- **Caller** take care of registers **\$a0-\$a3** and **\$t0-\$t9**
- **Callee** take care of **\$s0-\$s7**, **\$ra**.
- A **nonleaf** procedure is both a caller and a callee.

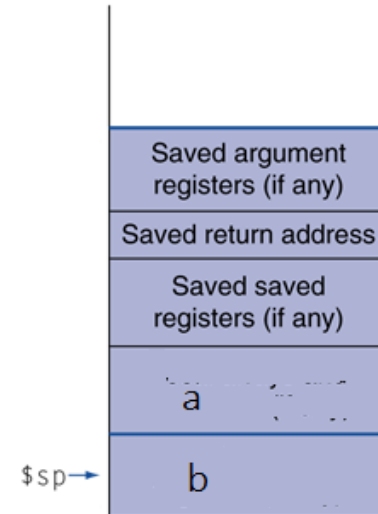


Local Data on the Stack

```
Func test() {  
1  → int a;  
    .... Stmts..  
2  → int b;  
    .....Stmts...  
}
```



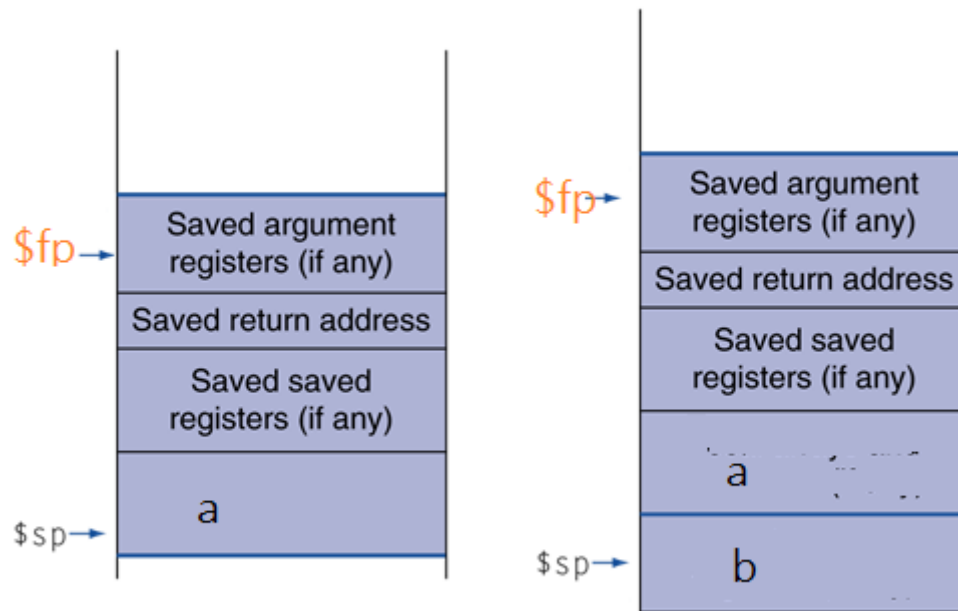
At 1



At 2

- local data are also preserved by **callee**
 - e.g., C automatic variables
- Therefore, \$sp value may change in callee => local variable have different **offsets**
 - e.g. a is \$sp in case 1, and \$sp+4 in case 2
 - hard to use **\$sp** to access local data
 - Define a new pointer => **\$fp**

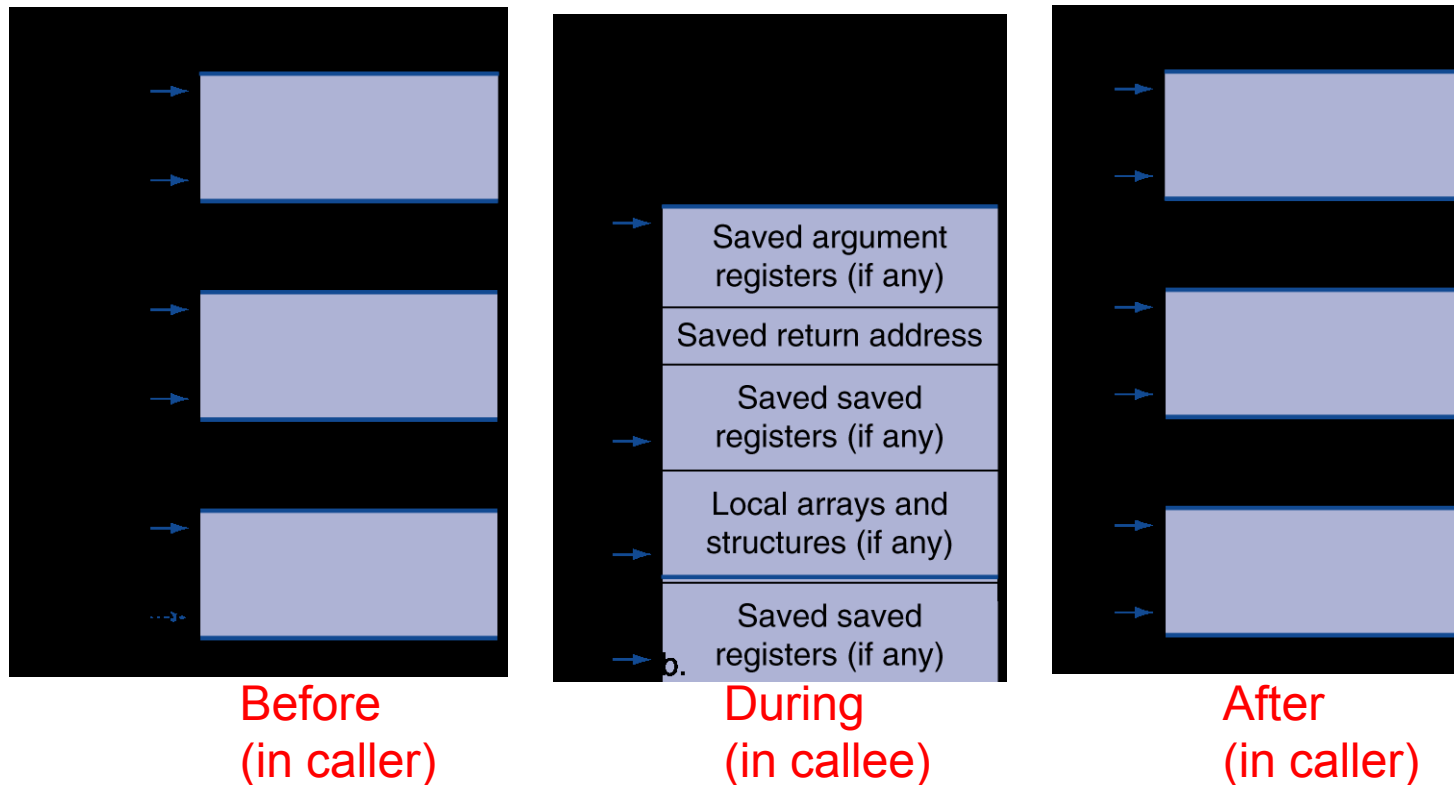
Frame pointer



```
Func test() {  
    int a;  
    .... //statements..  
    int b;  
    .....//statements  
}
```

- Frame pointer : a stable base register for local memory-reference
- (\$fp) points to the first word of the frame (activation record) of a procedure

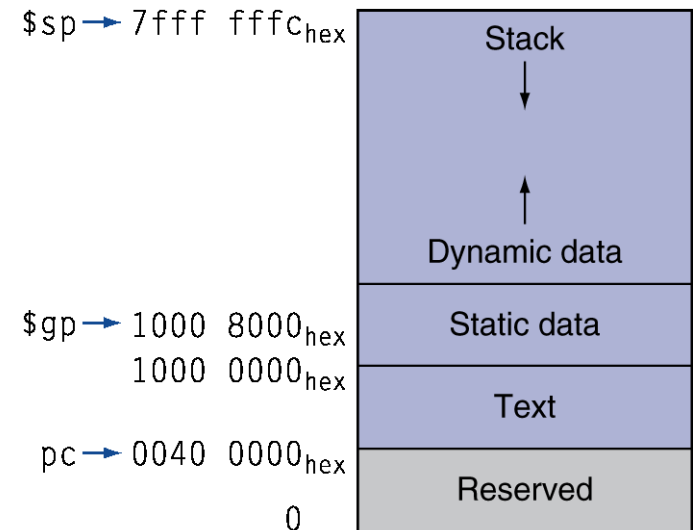
Summary: stack allocation before and after call



- Local data allocated by callee
 - e.g., C automatic variables

MIPS Memory Layout

- **Static Text:** program code, start at 00400000_{16}
- **Static data:** static variables in C, constant arrays and strings
 - Start at $1000\ 0000_{16}$
 - **\$gp (global pointer)** initialized to 10008000_{16}
 - allowing \pm offsets into this segment (1000 0000 to 1000 ffff)
- **Dynamic data:** heap,
 - E.g., malloc in C, new in Java
 - Grow up toward stack
- **Stack:** automatic storage
 - \$sp initialized to $7ffffffc_{\text{hex}}$



Only a software convention,
Not a part of MIPS architecture !

Summary: Register Conventions

- Caller takes care of **\$a0-\$a3** and **\$t0~\$t9** and callee takes care of **\$ra** and **\$s0~\$s7**
 - Must be saved/restored by **callee**
- **\$ra, \$s0~\$s7, \$gp, \$sp, \$fp** are preserved on a procedure call

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

Register 1, \$at, is reserved for the assembler.

Registers 26-27, called \$ko-\$k1, are reserved for the operating system.