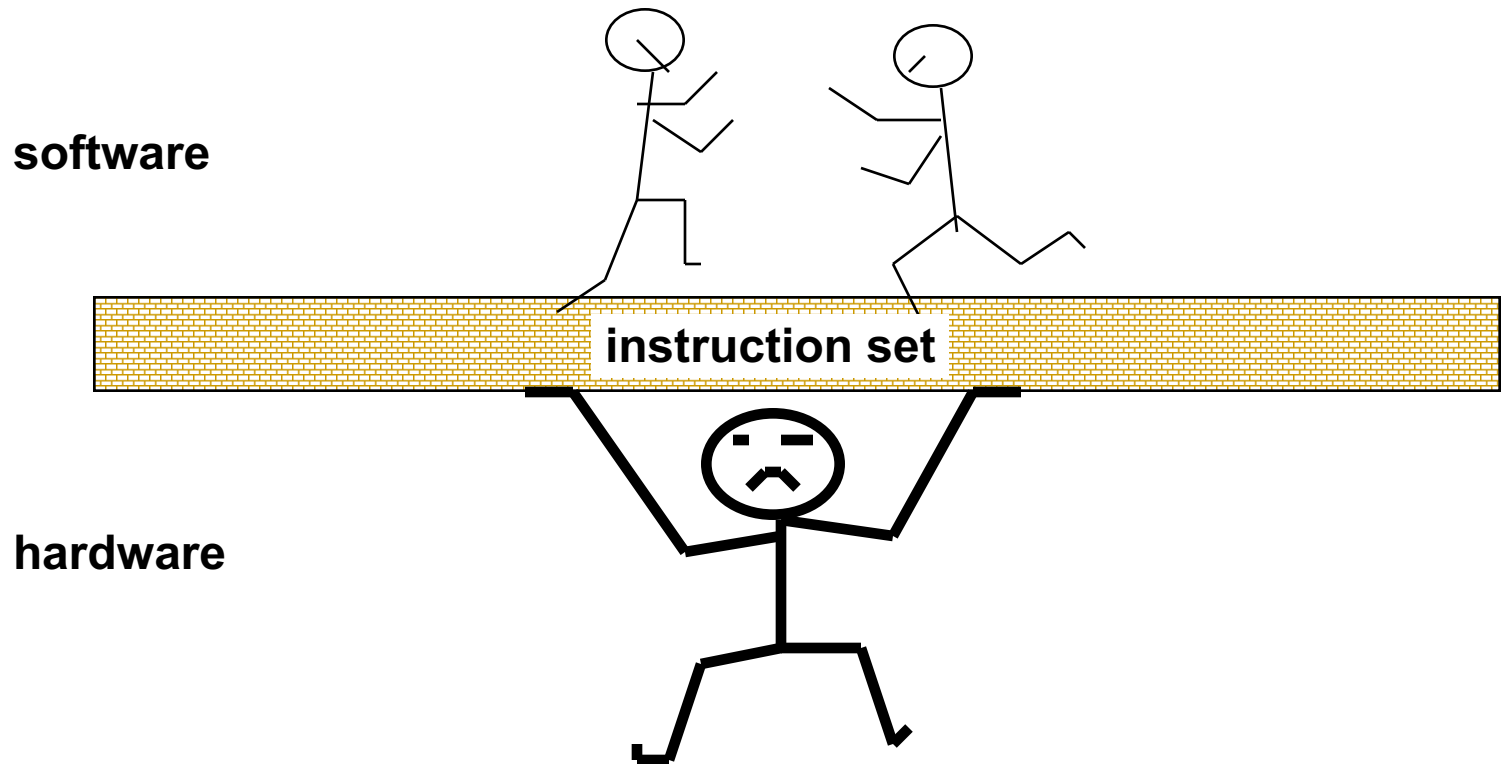


Lecture 3

- **Instruction Set Architecture**

Instruction Set Architecture



Instruction set provides an layer of abstraction to programmers

Levels of Representation

High Level Language
Program

Compiler

Assembly Language
Program

Assembler

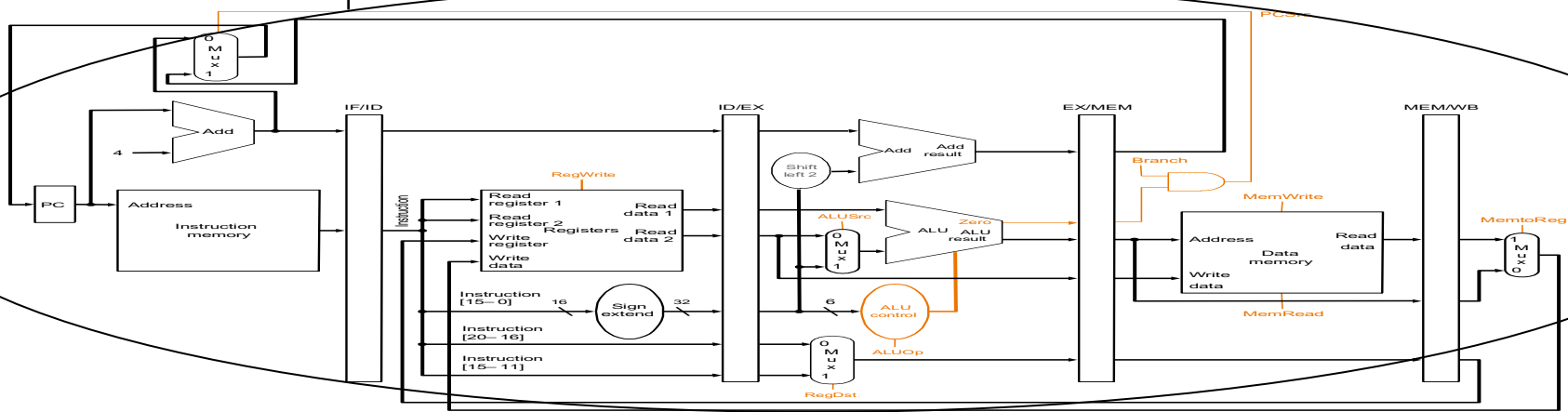
Machine Language
Program

Machine Interpretation

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



ISA Design Principle

To find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost.

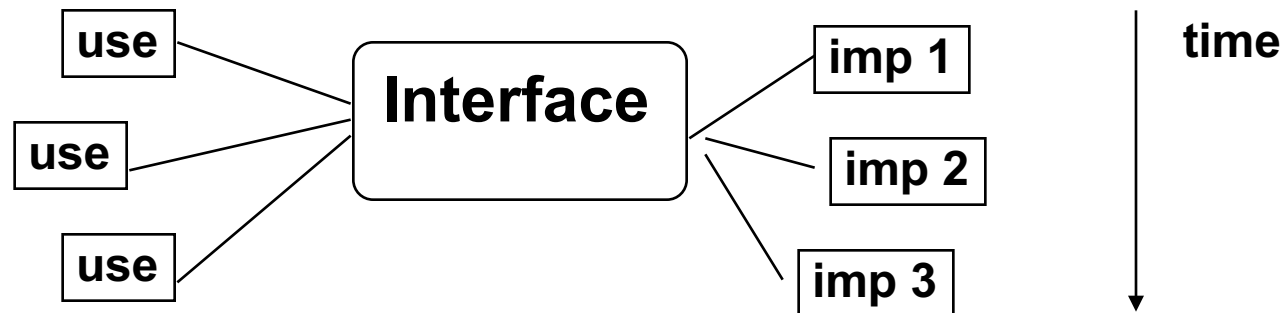
*“It is easy to see by formal-logical methods that there exist certain [instruction set] that are in abstract adequate to control and cause the execution of any sequence of operations....The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: **simplicity of the equipment** demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.”*

Burks, Goldstine, and von Neumann, 1947

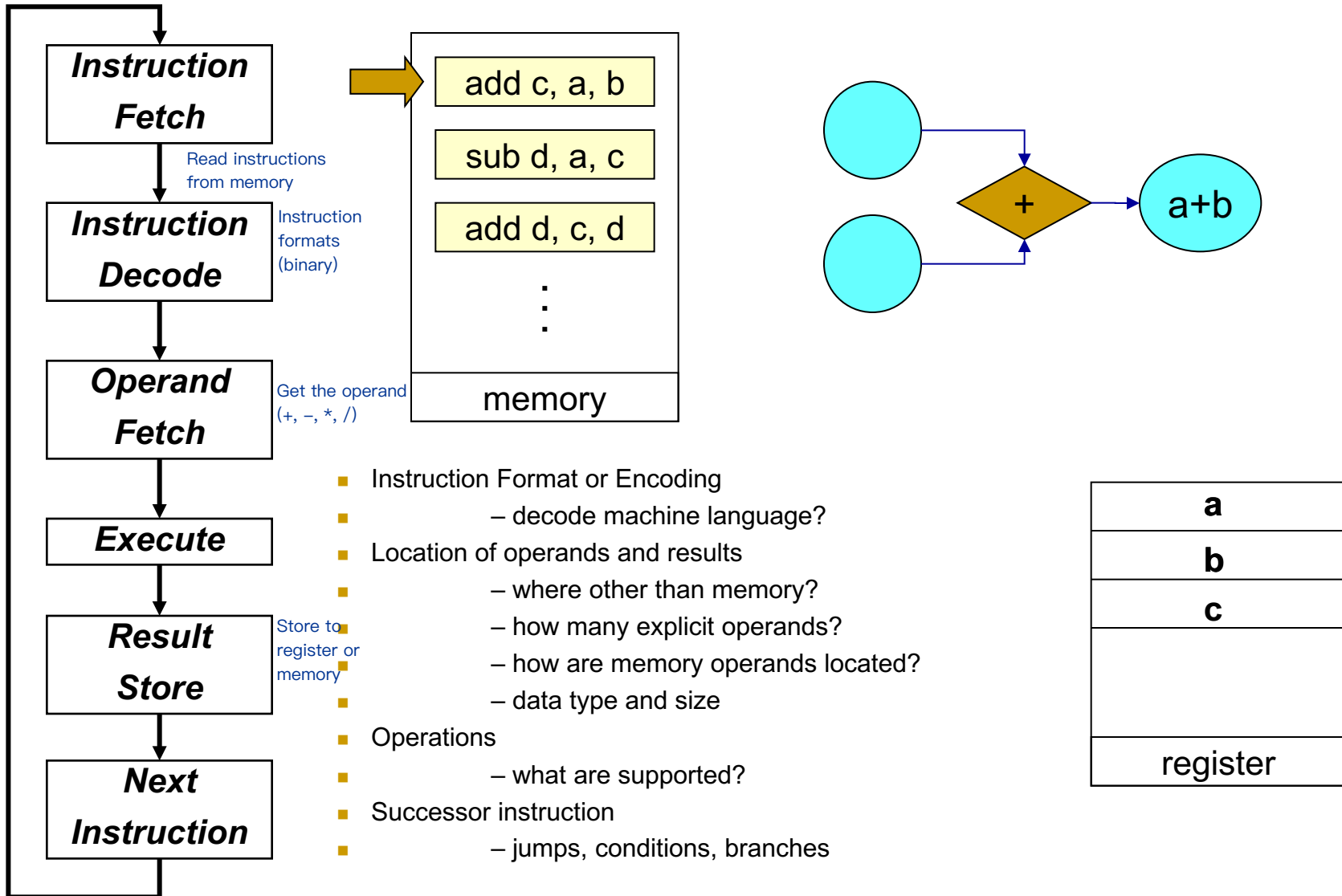
Interface Design

A good interface:

- Lasts through many implementations (portability, compatibility)
- Is used in many different ways (generality)
- Provides **convenient** functionality to higher levels
- Permits an **efficient** implementation at lower levels



Instruction Set Architecture: What Must be Specified?



General Purpose Register ISA

General Purpose Register:

■ register-memory

2 address: add R1, A

$R1 = R1 + \text{mem}[A]$

3 address: add R2, R1, A

$R2 = R1 + \text{mem}[A]$

■ register to register (load-store)

add Ra Rb Rc $Ra = Rb + Rc$

load Ra A $Ra = \text{mem}[A]$

store Ra A $\text{mem}[A] = Ra$

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 B and E

MIPS

- RISC Instruction set
 - Adopted in ATI, Broadcom, Cisco, NEC, Nintendo, Silicon Graphics, Sony, TI, etc.
- 3-address load store architecture
- Immediate and displacement **addressing mode** for load and store
- 31 32-bit integer registers (R0 = 0)
- 32 32-bit floating-point registers
- 32-bit HI, LO, PC (program counter)
- Examples:

```
add    $1, $2, $3    # $1 = $2 + $3
addi   $1, $1, 4     # $1 = $1 + 4
lw     $1, 100($2)   # $1 = Memory[$2+100]
sw     $1, 100($2)   # Memory[$2+100] = $1
```

Arithmetic Operations

- One operation must have exactly three operands

src1 src2
Add a, b, c

Destination

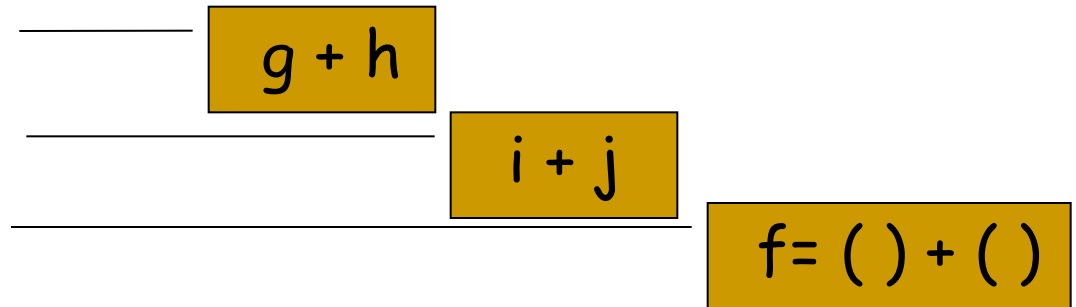
- Arithmetic operations
 - +, -, x, / (more on multiply & divide later)

Design Principle 1: Simplicity favors regularity.

From C to MIPS

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

add t0, g, h
add t1, i, j;
sub f, t0, t1;



Where are the operands stored?

Register operands

- Operands of arithmetic operations must be stored in **registers** in MIPS
 - Registers are primitive used in hardware design that are also visible to programmers

- MIPS has 32 registers

- Why 32? Not 128, 256, 1024?
 - For clock rate consideration, large or small?
 - For compiler, large or small?

- **Design Principle 2 : Smaller is faster**

- Each MIPS register has a name to make it easier to code, e.g.,

\$16 - \$22 → \$s0 - \$s7 (C variables)

\$8 - \$15 → \$t0 - \$t7 (temporary)

```
add t0, g, h
add t1, l, j;
sub f, t0, t1;
```



```
add $t0, $s1, $s2;
add $t1, $s3, $s4;
sub $s0, $t0, $t1
```

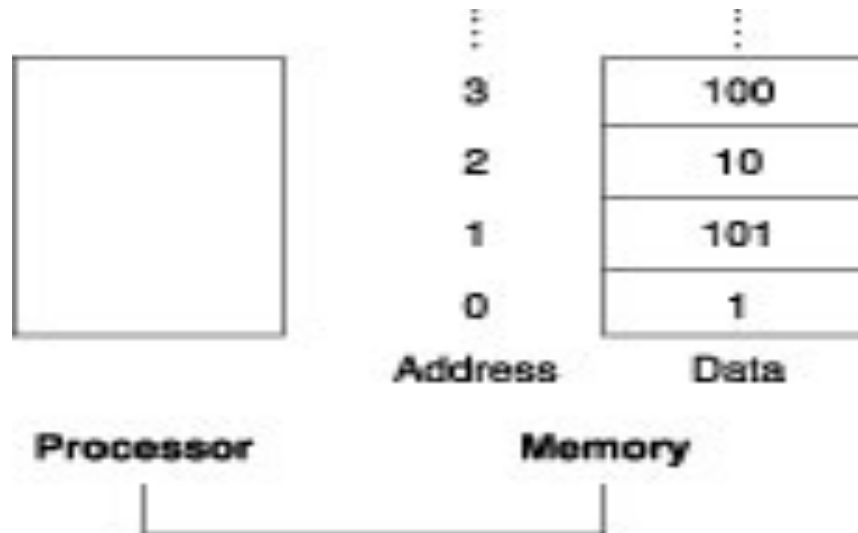
MIPS: Software Conventions for Registers

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Register 1 (\$at) reserved for assembler, 26-27 for operating system

Memory operands

- How to load operands from memory? How to store results to memory?
 - Data transfer instructions
 - lw \$t0, 8(\$s3) # t0 = mem[8+reg[\$s3]]
 - sw \$t0, 8(\$s3) # mem[8+reg[\$s3]] = \$t0



Addressing

■ Example: $g = h + A[8]$

- The starting address of array A is \$s3. Each element of array A is 1 word (4 bytes).

- g and h are stored in registers \$s1 and \$s2

`lw $t0, offset($s3) # t0 gets A[8]`

`add $s1, $s2, $t0 # s1 = s2+t0`

- What is the offset? **Index * 4**

Memory are referenced with byte addresses in MIPS



Addressing (cont.)

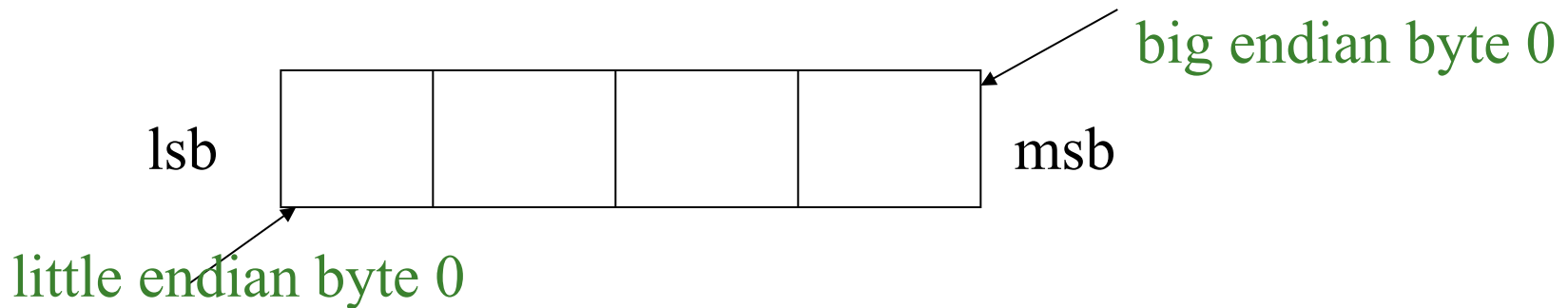
■ Example :

- `g = h + A[i] /* g, h,i, => $s1, $s2, $s4 */`
- What is the MIPS assembly code? Use `add`, `lw` instructions

Addressing (cont.)

■ Byte order: Big Endian vs. Little Endian

- Big endian: byte 0 is 8 most significant bits e.g., IBM/360/370, Motorola 68K, MIPS, Sparc, HP PA
- Little endian: byte 0 is 8 least significant bits e.g., Intel 80x86, DEC Vax, DEC Alpha



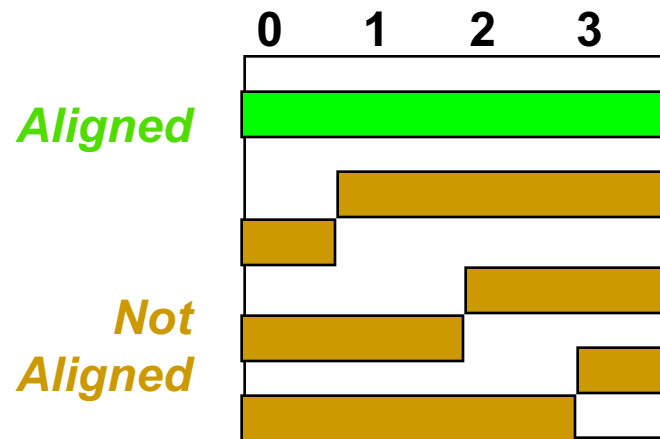
Example :

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

How is “12345678h” stored in memory?

Alignment

- MIPS require that objects fall on address that is multiple of their size
 - Word (4 bytes): aligned if $\text{address} \% 4 = 0$



Constant or Immediate operands

- Small constants are used quite frequently (50% of operands)

e.g., $A = A + 5;$
 $B = B + 1;$
 $C = C - 18;$

- Solutions? Why not?

- put 'typical constants' in memory and load them.
- Example: add constant 4 to register \$s3

```
lw  $t0, AddrConstant4($s1)
add $s3, $s3, $t0
```

- MIPS Instructions:

```
addi $s3, $s3, 4
```

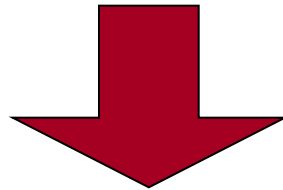
add immediate

Decreasing the instruction count.

Design Principle 3: Make the common case fast.

Representing Instruction in the Computer

add \$t0, \$s1, \$2 # \$t0 = \$s1 + \$2



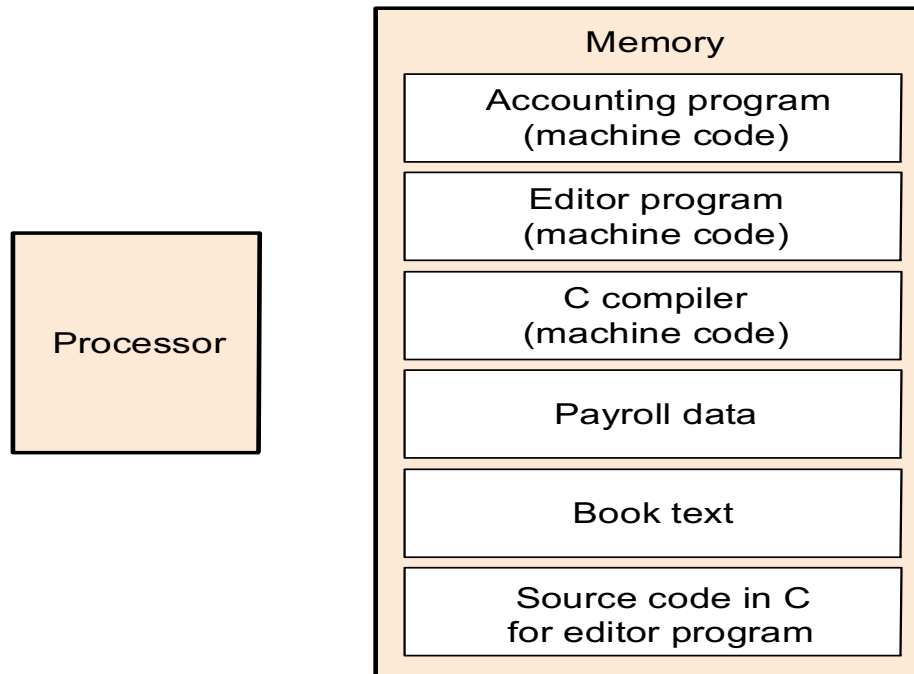
Machine language

001000	11101	11101	01000	00000	100000
--------	-------	-------	-------	-------	--------

**All represent with
binary numbers**

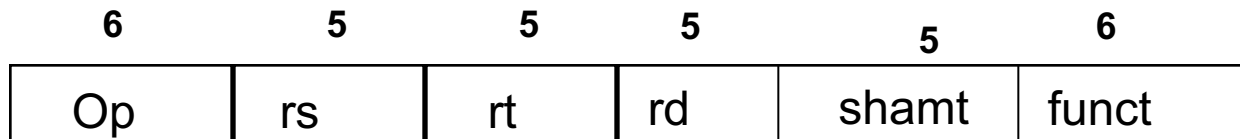
Stored-Program Concept

- Computers built on 2 key principles:
 - 1) Instructions are represented as numbers
 - 2) Thus, entire programs can be stored in memory to be read or written just like numbers



MIPS Instruction Format

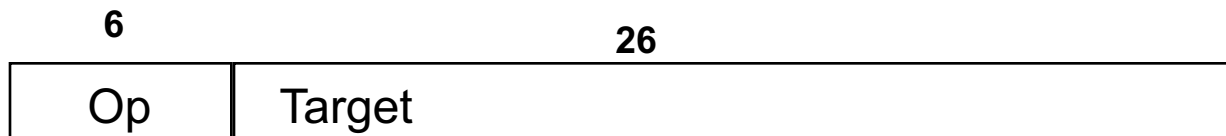
R-type instruction: register - register



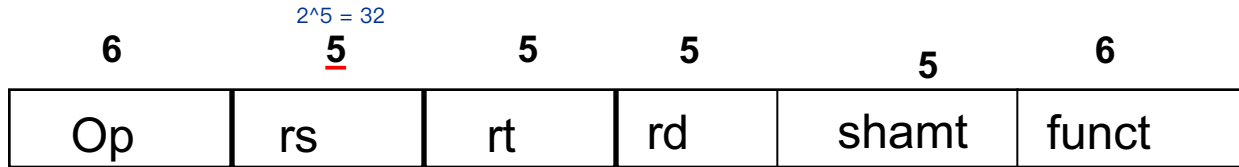
I-type instruction : register - immediate



J-type instruction: jump / call



Instruction Format



op: operation code

rs: the first source register

rt: the second source register

rd: the destination register

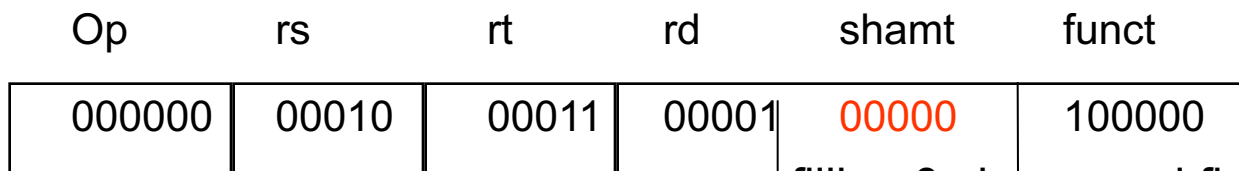
shamt: shift amount

funct: function field (specific variant of the op)

example: add – op = 0, func = 0x20

addu – op = 0, func = 0x21

Example: ADD \$1, \$2, \$3



filling 0s in unused fields

Instruction format (cont.)

- Can we use the same format for lw/sw instruction?
 - 5-bit constant is too small to index arrays or data structures
 - More instruction formats

Design Principle 4 : Good design demands good compromises

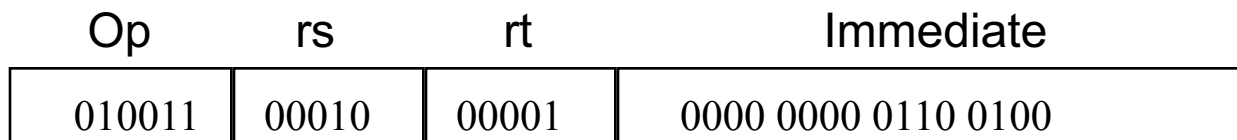
I-Type <op> rt, rs, immediate



op: operation code
rt : destination register
rs: source register
immediate: immediate value

Example:

1. addi \$1, \$2, 100 # \$1 = \$2 + 100 (immediate addressing)
2. lw \$1, 100(\$2) # \$1 = mem[\$2+100] (Displacement)
base



- Immediate fields represent both negative and positive constants

Logical operations

- Bitwise operations
 - View contents of register as 32 bits rather than as a single 32-bit number
- Shift left, shift right, and, or, not

Logical Shift

- Move (shift) all the bits in a word to the left or right by a number of bits, filling the emptied bits with 0s.
- Shift **right** by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000



signed extension

0000 0000 0001 0010 0011 0100 0101 0110

- Shift **left** by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000



0011 0100 0101 0110 0111 1000 0000 0000

Instruction encoding for logical shift

6	5	5	5	5	6
opcode	rs	rt	rd	shamt	funct

srl \$10, \$16, 4 # \$t2 = \$s0 >> 4 bits

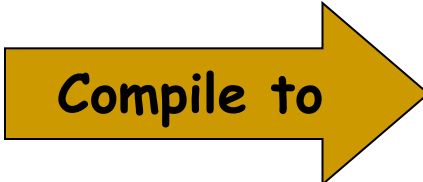
sll \$10, \$16, 4 # \$t2 = \$s0 << 4 bits

6	5	5	5	5	6
0	0	16	10	4	0

filling 0s in unused fields

Shift vs. Multiplication

- Shift for multiplication: in binary
 - Multiplying by 4 is the same as shifting left by 2:
 - $11_2 \times 100_2 = 1100_2$
 - $1010_2 \times 100_2 = 101000_2$
 - Multiplying by 2^n is same as shifting left by n
- Shift is faster than multiplication
 - a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

■ `a *= 8; (in C)`  `sll $s0,$s0,3 (in MIPS)`

OR & ORI

- `or $t0, $t1, $t2` # $\$t0 = \$t1 \mid \$t2$
- `ori $6, $6, 0x00ff`

Before: \$6

00001100001000001100001001010010

`ori $6, $6, 0x00ff`

After : \$6

00001100001000001100001011111111

AND & ANDI

- `and $t0, $t1, $t2` # `$t0 = $t1 & $t2`
- `andi $6, $6, 0x0000`

Before: \$6 00001100001000001100001001010010

`andi $6, $6, 0x0000`

After : \$6 00001100001000000000000000000000

NOR

- $\sim(A) = 1$ if $A = 0$
 $\sim(A) = 0$ if $A \neq 0$
- Nor instruction
 - ❑ `nor $t0, $t1, $t3` # $t0 = \sim(\$t1 \mid \$t3)$
 - ❑ $(\text{nor } \$t0, \$t1, \$t3) = \sim(\$t1)$, if $\$t3 = 0$

\$+3	00000000000000000000000000000000
------	----------------------------------

\$+1 | 11111111111111111111111111111111000000000

\$+0 | 0000000000000000000000000000000011111111

Integer operations: multiply & divide

- Multiply **mult rs, rt**
 - Multiplying two 32-bit numbers can yield a 64-bit number.
 - Use of HI and LO registers to store a 64-bit value
 - higher 32 bits are stored in HI
 - lower 32 bits are stored in LO
 - **MFHI rd -- move from hi \$rd = HI**
 - **MFLO rd -- move from lo \$rd = LO**

Multiply

- `mult $t1, $t2` `# t1 * t2`

\$t1	01111111111111111111111111111111
X \$t2	01000000000000000000000000000000

00011111111111111111111111111111	11000000000000000000000000000000
Hi	Lo

`mfhi $t3`

`$t3`

00011111111111111111111111111111

`mflo $t4`

`$t4`

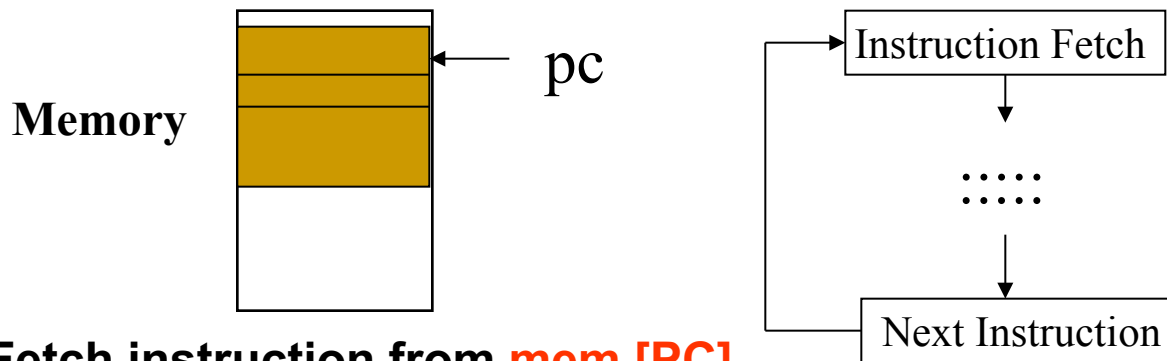
11000000000000000000000000000000

Divide

- `div rs, rt # rs / rt`
 - Quotient stored in Lo
 - Remainder stored in Hi
 - Use mfhi, mflo to get values in LO and HI

Instructions for making decisions

- Decision making instructions (e.g., branch, procedure call)
 - ❑ alter the control flow,
 - ❑ i.e., change the "next" instruction to be executed
 - ❑ I.e. change the **program counter (PC)**



- Fetch instruction from **mem [PC]**
- without decision making instruction
 - next instruction = **mem [PC + instruction_size]**

Branch instructions

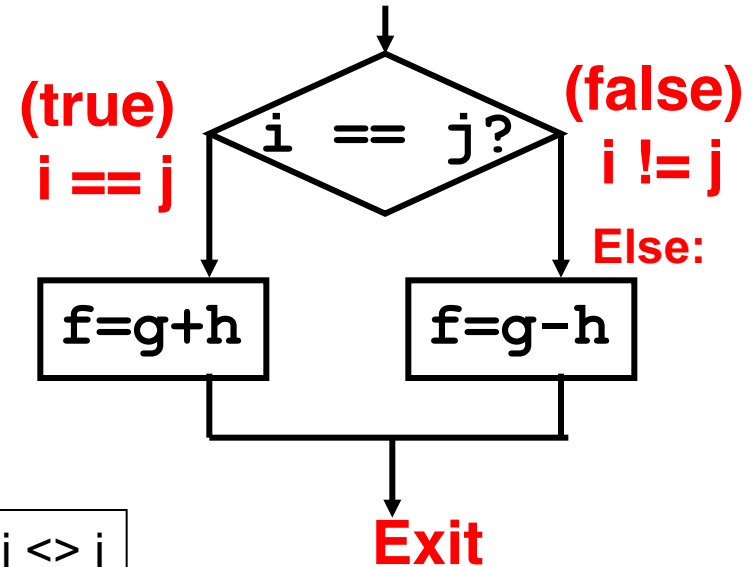
- Conditional branches
 - beq reg1, reg2, L1
 - Go to statement L1 if [reg1] == [reg2]
 - bne reg1, reg2, L2
 - Go to statement L2 if [reg1] != [reg2]
- Unconditional branches
 - j L1

Compiling C “if” into MIPS

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

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

```
bne $s3, $s4, Else    # go to Else if i <> j
add $s0, $s1, $s2     # f = g+h
J    Exit              # go to Exit
Else: sub $s0, $s1, $s2 # f = g - h
Exit:
```



Note: Compiler automatically creates labels to handle decisions (branches) appropriately

Compiling C “while” into MIPS

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

i: \$s3, k: \$s5

array base address: \$s6



Loop:

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

```
# $t1 = 4*i
# $t1=address of save[i]
# $t0 = save[i]
# go to Exit if save[i] != k
# i = i+1
# go to loop
```

Exit:

Can you optimize this code? Hint: use only one branch instruction in loop

Test for Inequalities

- `slt reg1, reg2, reg3`

```
if (reg2 < reg3)
    reg1 = 1;           # set
else reg1 = 0;         # reset
```

- Example : if ($g < h$) goto Less; ($g: \$s0, h: \$s1$)

```
slt    $t0, $s0, $s1    # $t0 = 1 if g<h
bne    $t0, $0, Less    # goto Less if $t0!=0
```

- Note that MIPS architecture doesn't include "branch on less than" because it is too complicated.

Case/Switch Statement

- “Switch” can be turned to a chain of “if-then-else” statements.
- Use jump address table to encode address.
- Need a new instruction:
jr (jump register)
 - ❑ Meaning an unconditional jump to the address specified in a register.
 - ❑ Provides full 32bits address

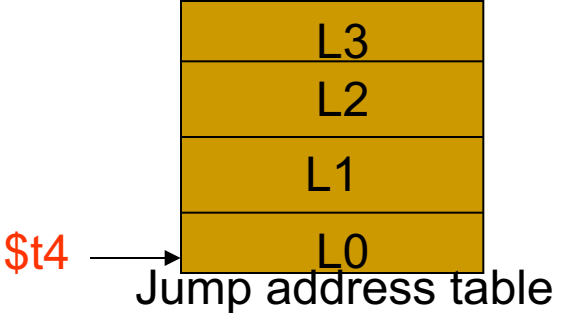
Example

Switch (k)

```
{
  case 0: f = i+j; break;
  case 1: f = g+h; break;
  case 2: f = g-h; break;
  case 3: f = i-j; break;
}
```

k -> s5
f -> s0
g -> s1
h -> s2
i -> s3
j -> s4
4 -> t2

```
slt  $t3, $s5, $zero # Test if k < 0
bne  $t3, $zero, Exit # if k < 0, go to Exit
slt  $t3, $s5, $t2    # Test if k < 4
beq  $t3, $zero, Exit # if k >= 4, go to Exit
sll  $t1, $s5, 2      # $t1 = 4*k
add  $t1, $t1, $t4     # $t1 = Addr of JumpTable[k]
lw   $t0, 0($t1)      # $t0 = JumpTable[k]
jr   $t0              # jump based on register $t0
L0:  add $s0, $s3, $s4
     j     Exit
L1:  add $s0, $s1, $s2
     j     Exit
L2:  sub $s0, $s1, $s2
     j     Exit
L3:  sub $s0, $s3, $s4
Exit:
```



Jump address table

-
- Add one slide

How codes are stored in memory

And jump table

Code example

```
■ .data
■ JumpTable: .word L0, L1, L2, L3

■ .text
■ .globl main
■ main:
■     la $t4, JumpTable # load Addr of JumpTable to $t4
■     li $s5, 1          # k = 1
■     li $s0, 2          # f = 2
■     li $s1, 3          # g = 3
■     li $s2, 4          # h = 4
■     li $s3, 5          # i = 5
■     li $s4, 6          # j = 6
■     li $t2, 4          # t2 = 4

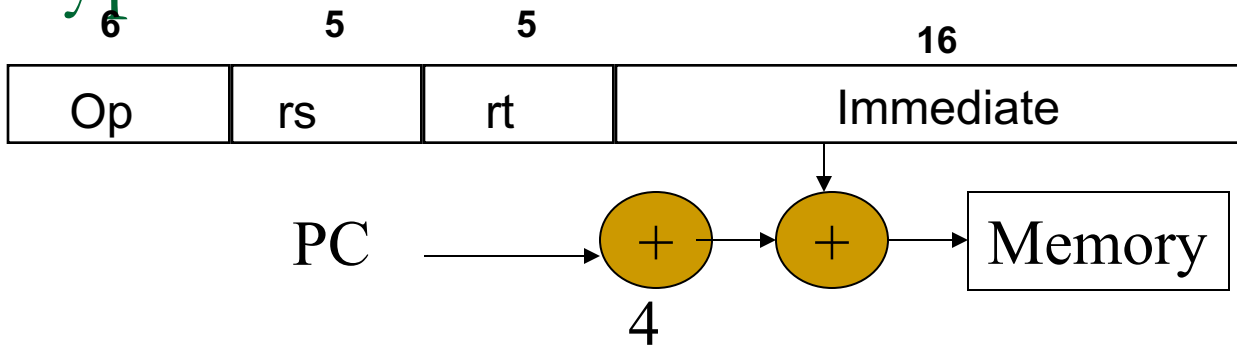
■ #=====
■     slt $t3, $s5, $zero # Test if k < 0
■     bne $t3, $zero, Exit # if k<0, go to Exit
■     slt $t3, $s5, $t2   # Test if k<4
■     beq $t3,$zero, Exit # if k>=4, go to Exit
■     sll $t1, $s5, 2     # $t1= 4*k
■     add $t1,$t1, $t4     # $t1 = Addr of JumpTable[k]
■     lw  $t0, 0($t1)     # $t0 = JumpTable[k]
■     jr  $t0             # jump based on register $t0
■ L0:    add $s0, $s3, $s4
■     j   Exit
■ L1:    add $s0, $s1, $s2
■     j   Exit
■ L2:    sub $s0, $s1, $s2
■     j   Exit
■ L3:    sub $s0, $s3, $s4
■ Exit:
■ #=====

■ move $a0, $s0          # Print Result
■ li $v0, 1
■ syscall

■ li $v0, 10             # Exit Program
■ syscall
```

Instruction encoding for conditional branches

- I-Type



op: operation code
rt : destination register
rs: source register
Immediate: immediate value

PC-relative addressing mode

$$\text{New PC} = \text{PC} + 4 + \text{Immediate} \times 4$$

Example:

bne \$1, \$2, L1 # L1 = (PC of bne) + 4 + Immediate × 4

Op	rs	rt	Immediate
000101	00001	00010	0000 0000 0001 1001 (25)

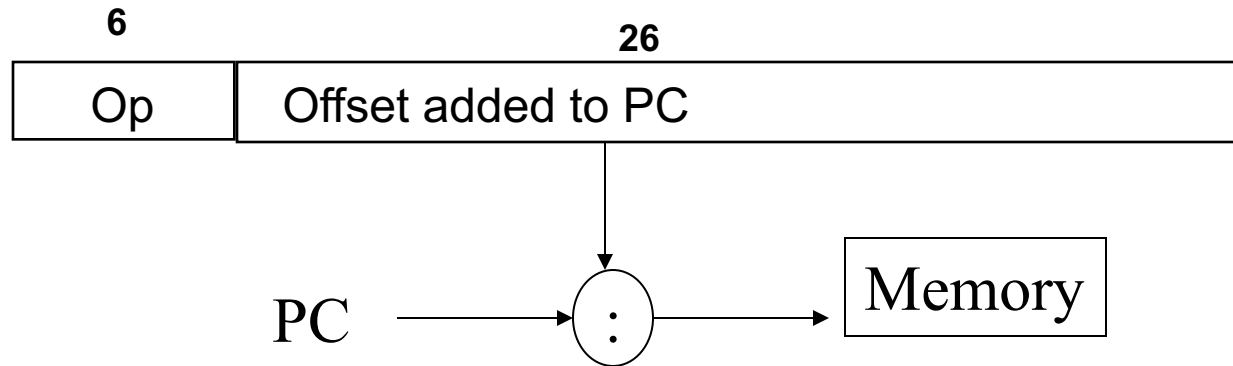
The last two bits of an instruction address should be 0s since each instruction is 4B

Branch Address Example

PC
80016 **bne \$t0, \$s5, Exit** New PC = PC + 4 + I * 4
80020 inst 1; 80028 = 80016 + 4 + 4I
80024 inst 2; I = 2
80028 Exit:

000101	00001	00010	? 2
--------	-------	-------	------------

J-Type <op> target



Example:

J 100 # PC = 100 concatenated with the upper bits of PC

PC 00001111111111111111111100000000

Immediate 000000000000000000000000011001

Target address 00000000000000000000000001100100

Branching Far Away

- What if we want to branch farther than can be represented in the 16 bits of the conditional branch instruction?

```
beq  $s0, $s1, L1
```

```
    bne $s0, $s1, L2  
    j L1  
L2:
```

Hint: use bne and jump

MIPS arithmetic instructions

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>	<i>Comments</i>
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; <u>exception possible</u>
subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; <u>exception possible</u>
add immediate	addi \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; <u>exception possible</u>
add unsigned	addu \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; <u>no exceptions</u>
subtract unsigned	subu \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; <u>no exceptions</u>
add imm. unsign.	addiu \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; <u>no exceptions</u>
multiply	mult \$2,\$3	Hi, Lo = $\$2 \times \3	64-bit signed product
multiply unsigned	multu \$2,\$3	Hi, Lo = $\$2 \times \3	64-bit unsigned product
divide	div \$2,\$3	Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3	Lo = quotient, Hi = remainder
divide unsigned	divu \$2,\$3	Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3	Unsigned quotient & remainder
Move from Hi	mfhi \$1	$\$1 = \text{Hi}$	Used to get copy of Hi
Move from Lo	mflo \$1	$\$1 = \text{Lo}$	Used to get copy of Lo

MIPS logical instructions

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>	<i>Comment</i>
and	and \$1,\$2,\$3	\$1 = \$2 & \$3	3 reg. operands; Logical AND
or	or \$1,\$2,\$3	\$1 = \$2 \$3	3 reg. operands; Logical OR
xor	xor \$1,\$2,\$3	\$1 = \$2 ^ \$3	3 reg. operands; Logical XOR
nor	nor \$1,\$2,\$3	\$1 = ~(\$2 \$3)	3 reg. operands; Logical NOR
and immediate	andi \$1,\$2,10	\$1 = \$2 & 10	Logical AND reg, constant
or immediate	ori \$1,\$2,10	\$1 = \$2 10	Logical OR reg, constant
xor immediate	xori \$1, \$2,10	\$1 = ~\$2 & ~10	Logical XOR reg, constant
shift left logical	sll \$1,\$2,10	\$1 = \$2 << 10	Shift left by constant
shift right logical	srl \$1,\$2,10	\$1 = \$2 >> 10	Shift right by constant
shift right arithm.	sra \$1,\$2,10	\$1 = \$2 >> 10	Shift right (sign extend)
shift left logical	sllv \$1,\$2,\$3	\$1 = \$2 << \$3	Shift left by variable
shift right logical	srlv \$1,\$2, \$3	\$1 = \$2 >> \$3	Shift right by variable
shift right arithm.	srav \$1,\$2, \$3	\$1 = \$2 >> \$3	Shift right arith. by variable

MIPS data transfer instructions

<i>Instruction</i>	<i>Comment</i>
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)

MIPS jump, branch, compare instruction

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>
branch on equal	beq \$1,\$2,100	if (\$1 == \$2) go to PC+4+100 <i>Equal test; PC relative branch</i>
branch on not eq.	bne \$1,\$2,100	if (\$1!= \$2) go to PC+4+100 <i>Not equal test; PC relative</i>
set on less than	slt \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0 <i>Compare less than; 2's comp.</i>
set less than imm.	slti \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0 <i>Compare < constant; 2's comp.</i>
set less than uns.	sltu \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0 <i>Compare less than; natural numbers</i>
set l. t. imm. uns.	sltiu \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0 <i>Compare < constant; natural numbers</i>
jump	j 10000	go to 10000 <i>Jump to target address</i>
jump register	jr \$31	go to \$31 <i>For switch, procedure return</i>
jump and link	jal 10000	\$31 = PC + 4; go to 10000 <i>For procedure call</i>

MIPS Instruction Set Design Principle

- Principle 1: Simplicity favors regularity
 - All operation takes three operands
- Principle 2: Smaller is faster
 - # of registers is 32
- Principle 3: Good design demands good compromises
 - 3 instruction format
- Principle 4: Make the common case fast
 - 52% of arithmetic operations involve constants
 - 69% of spice are constants
 - Immediate operands