Instructions: Language of the Computer

(Computer Organization: Chapter 2)

Yanyan Shen
Department of Computer Science and Engineering

The Language a Computer Understands

- Word a computer understands: *instruction*
- Vocabulary of all words a computer understands: *instruction set* (aka *instruction set architecture* or *ISA*)
- Different computers may have *different* vocabularies (i.e., different ISAs)
  - iPhone (*ARM*) not same as Macbook (*x86*)
- Or the *same* vocabulary (i.e., same ISA)
  - iPhone and iPad computers have same instruction set (*ARM*)
- Instructions are represented as numbers and, as such, are indistinguishable from data

MIPS Instruction Fields

- MIPS fields are given names to make them easier to refer to

op	rs	rt	rd	shamt	funct

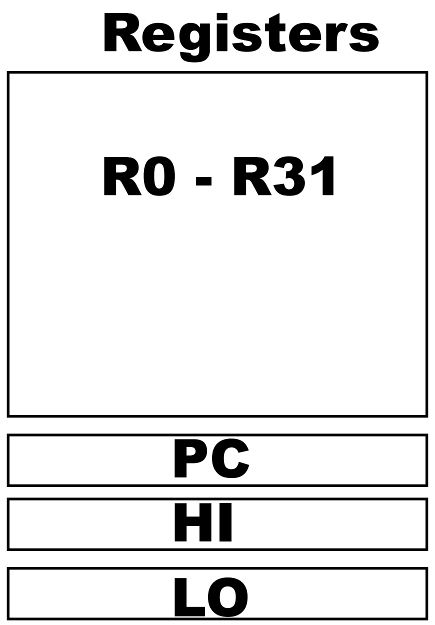
op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

MIPS-32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide

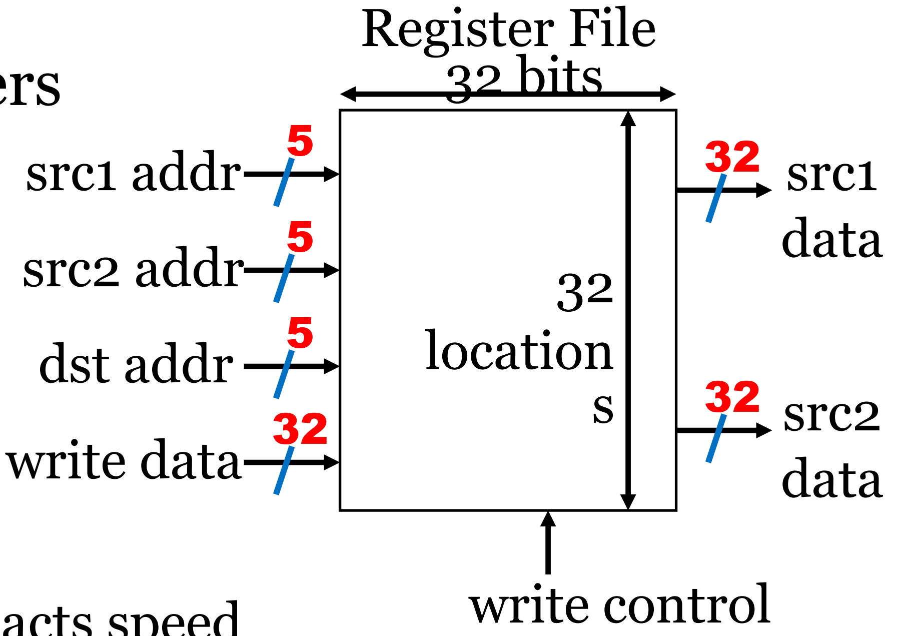
op	rs	rt	rd	sa	funct	R format
op	rs	rt	immediate			I format
op	jump target					J format

MIPS ISA Design

- MIPS Instruction Set has 32 integer registers
- MIPS registers are 32 bits wide
- MIPS calls this quantity a *word*
  - Some computers use 16-bit or 64-bit wide words
  - E.g., Intel 8086 (16-bit), MIPS64 (64-bit)

MIPS Register File

- Holds thirty-two 32-bit registers
  - Two read ports
  - One write port



- Registers are
  - Faster than main memory
    - Read/write port increase impacts speed quadratically
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)

Aside: MIPS Register Convention

Name	Register Number	Usage
$zero	0	constant 0 (hardware)
$at	1	reserved for assembler
$v0 - $v1	2-3	returned values
$a0 - $a3	4-7	arguments
$t0 - $t7	8-15	temporaries
$s0 - $s7	16-23	saved values
$t8 - $t9	24-25	temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return addr (hardware)

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

- **Each arithmetic instruction performs one operation**
- **Each specifies exactly three operands that are all contained in the register file** (`$t0,$s1,$s2`**)**

destination ← source1 op source2

- **Instruction Format (R format)**

op	rs	rt	td	sa	funct

MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

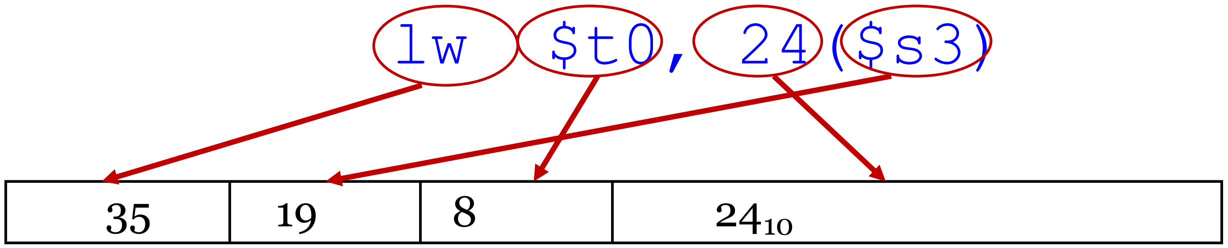
```
lw    $t0, 4($s3)   #load word from memory
 sw    $t0, 8($s3)   #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address

- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{15}$ (or 32,768 bytes) of the address in the base register

Machine Language - Load Instruction

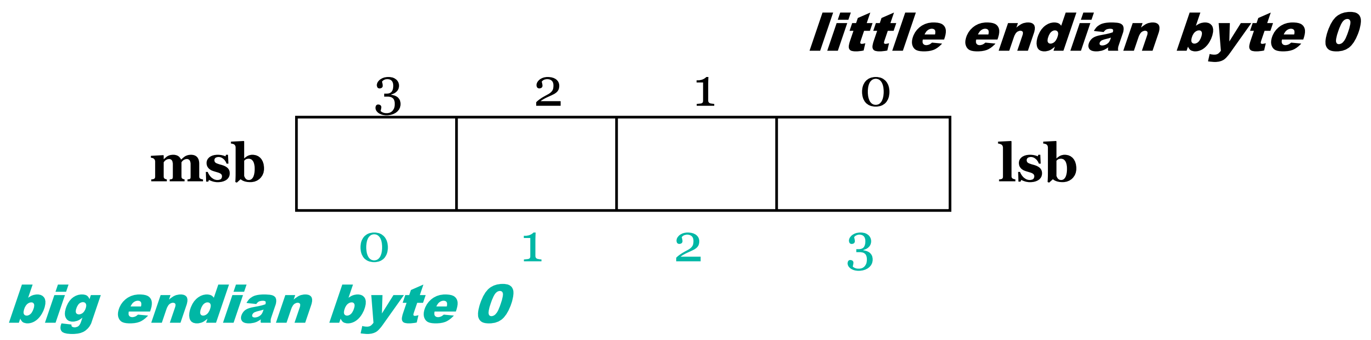
- Load/Store Instruction Format (I format):



- Destination address now in the rt field
  - no longer in the rd field
- Offset is limited to 16 bits
  - so can’t get to every location in memory (with a fixed base address)

Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - Alignment restriction - the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)
- Big Endian: leftmost byte is word address
  - E.g., IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Aside: Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb    $t0, 1($s3)   #load byte from memory
sb    $t0, 6($s3)   #store byte to  memory
```

0x28	19	8	16 bit offset

- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?

Example of Loading/Storing Bytes

- Given following code sequence and memory state what is the state of the memory after executing the code?

```
    add   $s3, $zero, $zero
    lb    $t0, 1($s3)
    sb    $t0, 6($s3)
```

Memory

Data	Word Address (Decimal)
0x 0 0 0 0 0 0 0 0	24
0x 0 0 0 0 0 0 0 0	20
0x 0 0 0 0 0 0 0 0	16
0x 1 0 0 0 0 0 1 0	12
0x 0 1 0 0 0 4 0 2	8
0x F F F F F F F F	4
0x 0 0 9 0 1 2 A 0	0

- What value is left in $t0? (big endian)
- What word is changed in Memory and to what? (big endian)
- What if the machine was little endian?

MIPS Immediate Instructions

- Small constants are used often in typical code
- Possible approaches?
  - put "typical constants" in memory and load them
  - create hard-wired registers (like $zero) for constants like 1
  - have special instructions that contain constants !

```
addi  $sp, $sp, 4     #$sp = $sp + 4
 slti $t0, $s2, 15    #$t0 = 1 if $s2<15
```

- Machine format (I format): **what about upper 16 bits?**

0x0A	18	8	0x0F

- The constant is kept inside the instruction itself!
- Immediate format limits values to the range $+2^{15}-1$ to -2^{15}

MIPS Control Flow Instructions

- MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl  #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl  #go to Lbl if $s0=$s1
```

  - Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

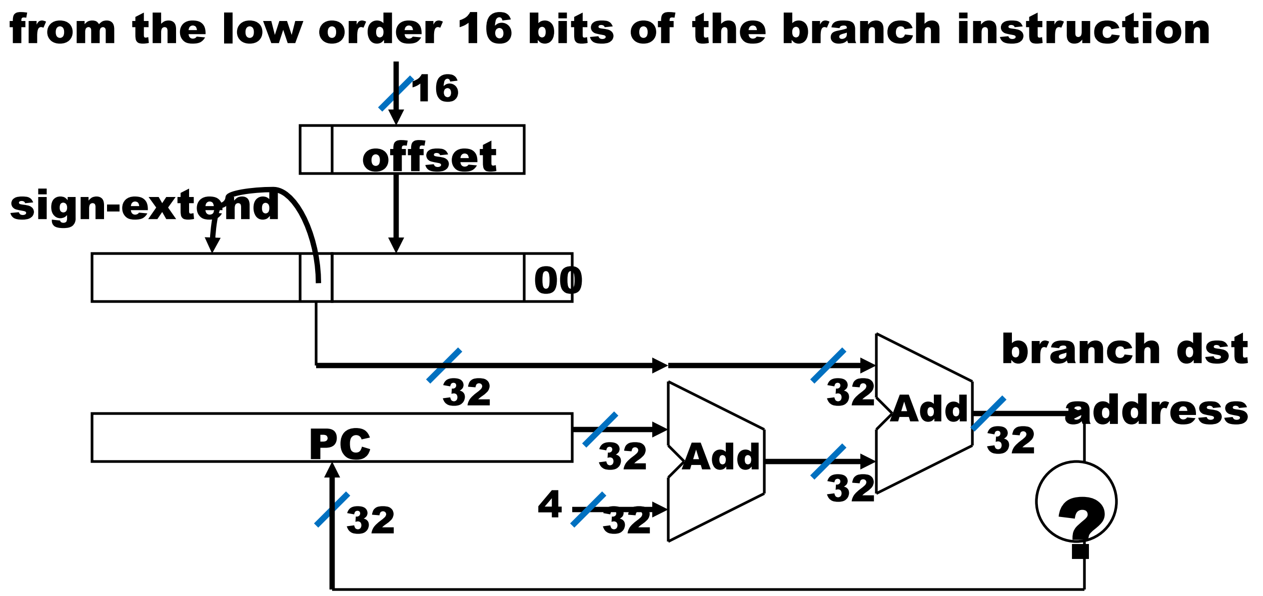
- Instruction Format (I format):

0x05	16	17	16 bit offset

- How is the branch destination address specified?

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway



In Support of Branch Instructions

- We have beq, bne, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, slt
- Set on less than instruction:

```
slt $t0, $s0, $s1      # if $s0 < $s1     then
                       # $t0 = 1          else
                       # $t0 = 0
```

- Instruction format (R format):

0	16	17	8		0x24

- Alternate versions of slt

```
slti $t0, $s0, 25      # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1     # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25     # if $s0 < 25 then $t0=1 ...
```

Aside: More Branch Instructions

- Can use slt, beq, bne, and the fixed value of 0 in register $zero to create other conditions
  - less than   blt $s1, $s2, Label

    ```
    slt  $at, $s1, $s2       #$at set to 1 if
    bne  $at, $zero, Label   #$s1 < $s2
    ```

  - less than or equal to   ble $s1, $s2, Label
  - greater than   bgt $s1, $s2, Label
  - great than or equal to   bge $s1, $s2, Label
- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - Its why the assembler needs a reserved register ($at)

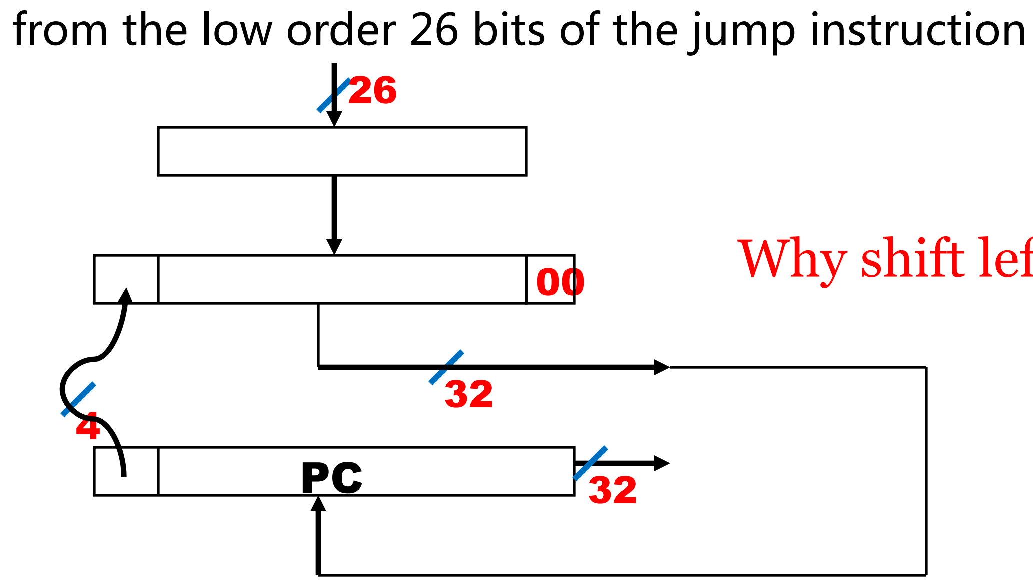
Other Control Flow Instructions

- MIPS also has an unconditional branch instruction or jump instruction:

```
j  label                    #go to label
```

- Instruction Format (J Format):

0x02	26-bit address



Why shift left by two bits?

Aside: Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?

- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

```
        beq   $s0, $s1, L1

                 ⇩

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

Summary

- Instructions: machine language
- MIP-32 ISA
  - Register file
  - R-type, I-type, J-type instruction formats
  - Arithmetic, memory access, control flow
    - Assembly language vs machine code