
Chapter 2.4 ~ 6

Summary of the last class:

- **MIPS**
 - loading words (32-bit)
but addressing bytes
 - arithmetic on registers only
(load/store architecture)

- **Endian & Alignment**

- **Instruction**

Meaning

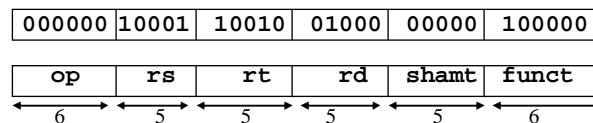
<code>add \$s1, \$s2, \$s3</code>	<code>\$s1 = \$s2 + \$s3</code>
<code>sub \$s1, \$s2, \$s3</code>	<code>\$s1 = \$s2 - \$s3</code>
<code>lw \$s1, 100(\$s2)</code>	<code>\$s1 = Memory[\$s2+100]</code>
<code>sw \$s1, 100(\$s2)</code>	<code>Memory[\$s2+100] = \$s1</code>

Table of Contents

- Ch.1, 4 Introduction & Performance
 - Ch. 2 Instruction: Machine Language
 - 2.1 Introduction
 - 2.2 Operations (arithmetic, memory operations)
 - 2.3 Operands
 - 2.4 Representing instructions
 - 2.5 Logical operations
 - 2.6 Control flow operations
 - 2.7 Supporting procedures
 - 2.8 Comm. with people (ASCII)
 - 2.9 MIPS addressing
 - 2.10-12 Starting a program, Compiler
 - 2.13 Example
 - 2.14-20 Etc.
 - Ch. 3 CPU Implementation: Arithmetic
 - Ch. 5 CPU Implementation: All others
 - Ch. 6-9 Advanced topics
- Key parts of this course

2.4 Representing Instructions in the Computer

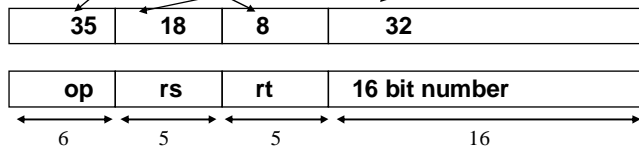
- Instructions, like registers and words of data, are also 32 bits long
 - Example: `add $t0, $s1, $s2`
 - registers have numbers, `$t0=8, $s1=17, $s2=18` (page 140, Table 3.13)
- Instruction Format (machine code):



- *Can you guess what the field names stand for?*
- *How many registers can it specify (rs, rt, rd) ?*
- *How many operations can it support ?*
- *How many functions can it support ?*

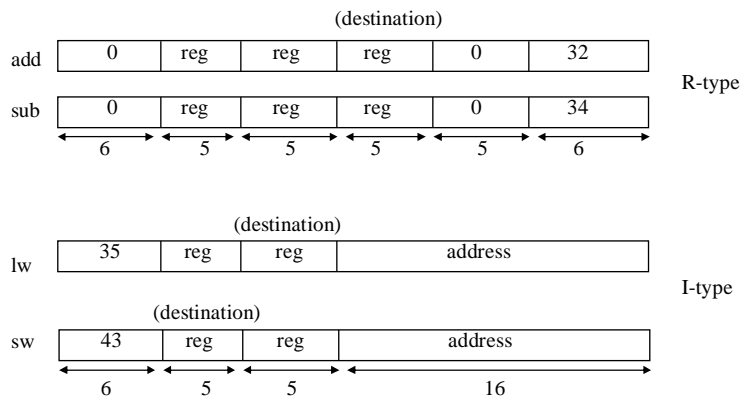
Machine Language

- Consider the load-word and store-word instructions,
- Introduce a new type of instruction format
 - I-type for data transfer instructions
 - other format was R-type for register
- Example: `lw $t0, 32($s2)`



- Regularity principle ?

So far...



Read memory (\$t0) to \$t1 \Rightarrow `lw $t1, ($t0)` \Rightarrow 35 / 9 / 8 / 0
 Write \$t1 to memory (\$t0)
 = Write memory (\$t0) from \$t1 \Rightarrow `sw $t1, ($t0)` \Rightarrow 43 / 9 / 8 / 0

Example

- $A[300] = h + A[300]$
- Assembly code (\$t1 : base of the array A, \$s2 : h)
 - lw \$t0, 1200(\$t1) # temporary register \$t0 = A[300]
 - add \$t0, \$s2, \$t0 # temporary register \$t0 = h+A[300]
 - sw \$t0, 1200(\$t1) # store back into A[300]
- Machine code (hexa & binary format) ?

lw	35	9	8	1200		
add	0	18	8	8	0	32
sw	43	9	8	1200		

- “Where are those instructions are stored?”

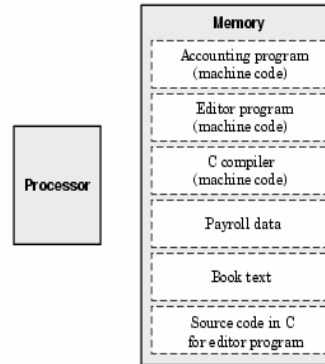
Example

- What's the address for A[300]?
- What's the address for add inst?
- CPU processing
 - Get inst. from memory
 - Get operand from memory
 - Execute the inst.
 - Repeat with the next inst. (most probably it is in the next address, 2010+4 not 2010+1)
- “Stored program architecture” = von Neuman (1903-1957, Hungary) architecture

Memory		
address	content	variable
160	149000	A[0]
.....
????		A[300]
.....	
.....	
.....	
2010	35/9/8/1200	lw inst.
????	0/18/8/8/0/32	add inst.
????	43/9/8/1200	sw inst.
.....	
.....	

Stored Program Architecture

- Instructions are bits
- Programs are stored in memory
— to be read or written just like data
- Fetch & Execute Cycle
 - Instructions are fetched and put into a special register
 - Bits in the register "control" the subsequent actions
 - Fetch the "next" instruction and continue



2.5 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
• nor	nor \$1,\$2,\$3	\$1 = ~(\$2 \$3)	3 reg. operands; Logical NOR
• xor	xor \$1,\$2,\$3	\$1 = \$2 ⊕ \$3	3 reg. operands; Logical XOR
• 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 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.	sra \$1,\$2,10	\$1 = \$2 >> 10	Shift right (sign extend)
• shift right arithm.	srav \$1,\$2,\$3	\$1 = \$2 >> \$3	Shift right arith. by variable
<u>(Immediate instruction would be introduced later ...)</u>			
• 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

2.6 Instructions for Making Decisions

- Control flow instructions ← in addition to arithmetic inst. & memory inst.

- alter the control flow,
- i.e., change the "next" instruction to be executed (otherwise, it is address of current inst.+4)

- MIPS conditional branch instructions (compare&branch type):

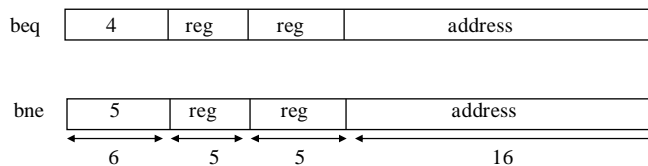
```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

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

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

Representing beq/bne Instructions

- `bne $t0, $t1, Label`
`beq $t0, $t1, Label`



Offset as in lw/sw instructions

What's the base address in this case?

⇒ "PC": it is called "PC-relative addressing"

⇒ Ranges: $0 \sim 2^{16}$ (64K)

- Can be negative? "Must be"

- Can be byte-boundary? "Cannot be"

- Branch address = PC + offset||00

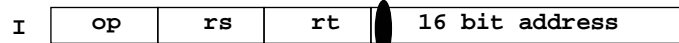
- Ranges: $-2^{17} \sim 2^{17}$

* Addresses in Branches

- Instructions:

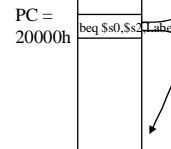
```
bne $t4,$t5,Label    Next instruction is at Label if $t4!=$t5
beq $t4,$t5,Label    Next instruction is at Label if $t4=$t5
```

- Formats:



- Next PC = current PC + Label : "PC-relative addressing"

- PC = program counter
- most branches are local (principle of locality)
- Branching range is ???
 - When "current PC"=20000h
 - The jumping address is 20000~2ffff ??? (Sign bit)
 - The jumping address is 20000-7fff ~ 2000+7fff = 18001~27fff
 - => 20000-7fff*4 ~ 20000+7fff*4 = **0 ~ 40000** (-4)



Control

- MIPS unconditional branch instructions:

```
j label
```

- Same example: if (i==j) h = i + j;

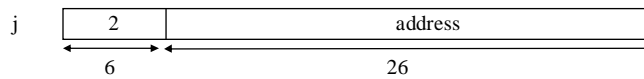
```
bne $s0, $s1, Label
add $s3, $s0, $s1
Label:     ....
```

```
beq $s0, $s1, Label
j skip_add
Label:
    add $s3, $s0, $s1
skip_add:     ....
```

Less efficient, but works...

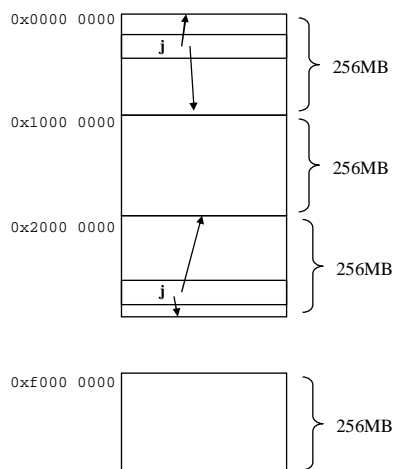
Representing j Instructions

- `j Label`



PC-relative addressing as in beq/bne instructions?
 NO. Simply because it can specify a larger range.
 Branch address = offset||00
 =>Range: $0 \sim 2^{28}$ (256MB)
 Not enough: Branch address (32-bit full address)
 = Upper 4-bit from PC || offset || 00
“Pseudo-direct addressing”

Addresses in Jumps



Memory is virtually divided into sixteen 256MB chunks and jump address is limited to the chunk where the “j” instruction is located.

Loop

- Loop: $g = g + A[i]$
 $l = l + j$;
if ($l \neq h$) go to Loop;
- A is an array of 100 elements, base in \$s5
- g, h, l and j are stored to \$s1, \$s2, \$s3 and \$s4
- Loop:

```
add    $t1, $s3, $s3    #
add    $t1, $t1, $t1    #
add    $t1, $t1, $s5    # $t1 = ???
lw     $t0, 0($t1)

add    $s1, $s1, $t0
add    $s3, $s3, $s4    # step is $s4 (j)

bne    $s3, $s2, Loop
```

Control Flow

- We have: beq, bne, what about Branch-if-less-than (“blt”)?
- New instruction:

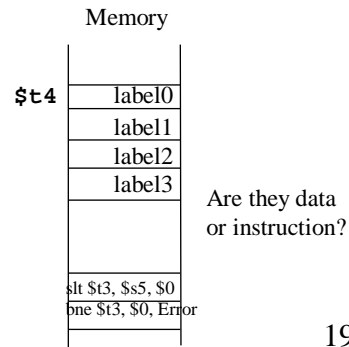
```
if $s1 < $s2 then
    $t0 = 1
else
    $t0 = 0
```

```
slt $t0, $s1, $s2
```
- Can use this instruction to build “blt \$s1, \$s2, Label”
(slt + bne = blt)
 - can now build general control structures
- Note that the assembler needs a register to do this,
 - there are policy of use conventions for registers

case/switch and jump address table

- switch (k) {
 - case 0: goto label0; /* k=0 */
 - case 1: goto label1; /* k=1 */
 - case 2: goto label2; /* k=2 */
 - case 3: goto label3; /* k=3 */
- Notes: (i) slt / slti (I-type), (ii) j / jr (R-type), (iii) \$zer0, (iv) add \$t2, \$t2, \$t2 ???

- ; if k<0, goto Error
 - slt \$t3, \$s5, \$zero
 - bne \$t3, \$zero, Error
- ; if k>=4, goto error
 - slti \$t3, \$s5, 4
 - beq \$t3, \$zero, Error
- ; jumping address is stored in \$t4+k*4
 - add \$t1, \$s5, \$s5
 - add \$t1, \$t1, \$t1 ; k*4
 - add \$t1, \$t1, \$t4 ; \$t4+k*4
 - lw \$t0, 0(\$t1) ; read jumping address
 - jr \$t0 ; jump



So far:

- Instruction** **Meaning**
 - add \$s1,\$s2,\$s3 \$s1 = \$s2 + \$s3
 - sub \$s1,\$s2,\$s3 \$s1 = \$s2 - \$s3
 - lw \$s1,100(\$s2) \$s1 = Memory[\$s2+100]
 - sw \$s1,100(\$s2) Memory[\$s2+100] = \$s1
 - bne \$s4,\$s5,L Next instr. is at Label if \$s4 ≠ \$s5
 - beq \$s4,\$s5,L Next instr. is at Label if \$s4 = \$s5
 - j Label Next instr. is at Label

- Formats:**

R	op	rs	rt	rd	shamt	funct	(add, sub)
I	op	rs	rt	16 bit address			(lw,sw)
J	op	26 bit address					(j)

Which type is bne, beq, or slt inst.?