Review: Computer Organization

Instruction: Language of the Machine

MIPS arithmetic

- **All instructions have 3 operands**
- **Operand order is fixed (destination first)**

Example:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

MIPS arithmetic

- **Design Principle: simplicity favors regularity. Why?**
- **Of course this complicates some things...**

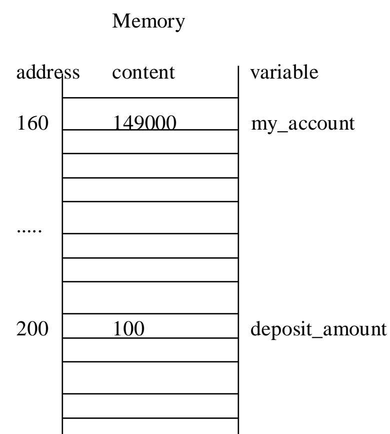
C code:
```
A = B + C + D;
E = F - A;
```

MIPS code:
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- **Operands must be registers, only 32 registers provided**
- **Design Principle: smaller is faster. Why?**

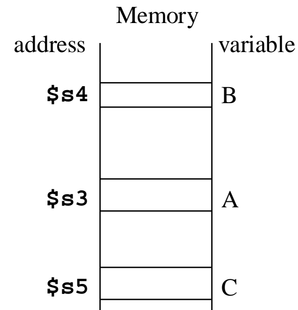
MIPS arithmetic

- **How about "memory operand" ?**
- **For example**
  - **my_account = my_account + deposit_amount;**
- **Data**
  - **Data is stored in memory**
  - **Each variable has its address**
- **Translate to MIPS code**
  - **[160] = [160] + [200];**
- **But, MIPS does not allow memory operands in the arithmetic operation**



Data Transfer (Memory) Instructions

- Load and store instructions
- The only instructions to access memory
- Example:



C code: `A = B + C`

MIPS code:

```
lw $s0, ($s3)
lw $s1, ($s4)
lw $s2, ($s5)
add $s0, $s1, $s2
```

- $s3 : the data itself = “160”
  ($s3) : data in memory addressed by $s3 = “00024608”

- Store word has destination last

So far we’ve learned:

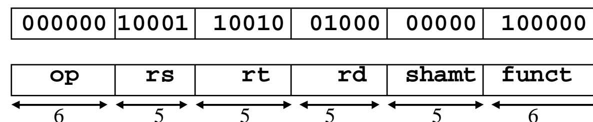
- MIPS
  — loading words but addressing bytes
  — arithmetic on registers only

- <u>Instruction</u> / <u>Meaning</u>

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`

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):**



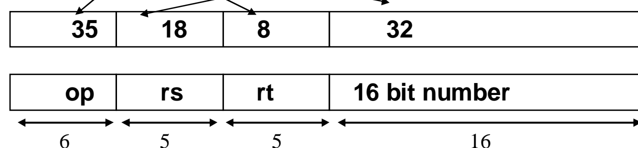
- *Can you guess what the field names stand for?*

- *How many registers can it specify (rs, rt, rd) ?*
- *How many operations can it support ?*
- *How many functions can it support ?*

Machine Language

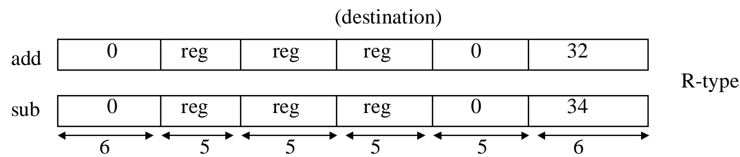
- **Consider the load-word and store-word instructions,**

- **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)`

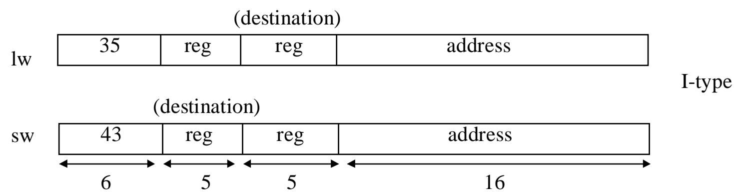


- **Regularity principle ?**

So far…



R-type

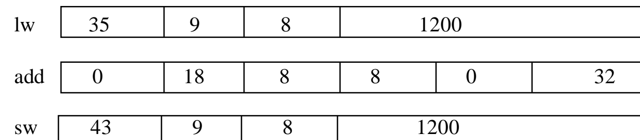


I-type

Read memory ($t0) to $t1 => lw $t1, ($t0) => 35 / 9 / 8 / 0
Write $t1 to memory ($t0)
= Write memory ($t0) from $t1 => sw $t1, ($t0) => 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) ?**



- **<u>"Where are those instructions are stored?"</u>**

2.5 MIPS logical instructions

Instruction	*Example*	*Meaning*	*Comment*
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

(Immediate instruction would be introduced later …)

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):**

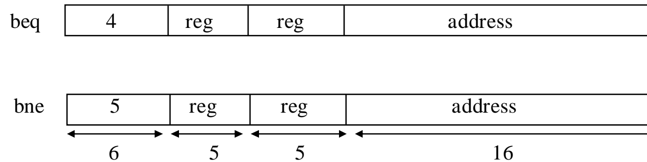
```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- **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: 0~2^{16} (64K)
- Can be negative? "Must be"
- Can be byte-boundary? "Cannot be"
- Branch address = PC + offset||00
- Ranges: -2^{17} ~ 2^{17}

13

* Addresses in Branches

- **Instructions:**

```
bne $t4,$t5,Label
beq $t4,$t5,Label
```

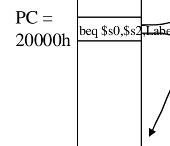
Next instruction is at Label if $t4!=$t5
Next instruction is at Label if $t4=$t5

- **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 ???
      - The jumping address is 20000-7fff ~ 2000+7fff = 18001~27fff

      => 20000-7fff*4 ~ 20000+7fff*4 = <u>**0 ~ 40000**</u> (-4)

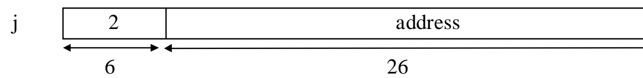
(Sign bit)



14

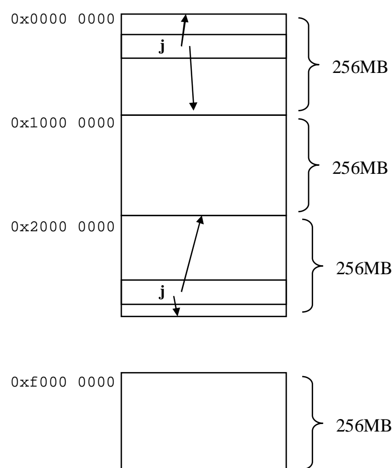
Representing j Instructions

- `j Label`



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

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]
  I = I + j;
  if (I != h) go to Loop;

- A is an array of 100 elements, base in $s5
- g, h, I and j are stored to $s1, $s2, $s3 and $s4

- Loop:

```
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
      slt $t0, $s1, $s2       else
                                 $t0 = 0
```

- 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

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.?