Summary of the last class:

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

- **Endian & Alignment**

- **Instruction**
  - **add $s1, $s2, $s3**  \hspace{1cm} $s1 = s2 + s3$
  - **sub $s1, $s2, $s3**  \hspace{1cm} $s1 = s2 - s3$
  - **lw $s1, 100($s2)**  \hspace{1cm} $s1 = \text{Memory}[s2+100]$
  - **sw $s1, 100($s2)**  \hspace{1cm} \text{Memory}[s2+100] = s1
Table of Contents

- Ch. 1 Introduction
- Ch. 2 Instructions: Language of the Computer
  - 2.1 Introduction
  - 2.2 Operations (arithmetic, memory operations)
  - 2.3 Operands
  - 2.4 Signed and unsigned numbers
  - 2.5 Representing instructions
  - 2.6 Logical operations
  - 2.7 Control flow operations
  - 2.8 Supporting procedures
  - 2.9 Comm. with people (ASCII)
  - 2.10 MIPS addressing
  - 2.11 Synchronization
  - 2.12 Starting a program, Compiler
  - 2.13 Example
  - 2.14-21 Etc.
- Ch. 3 CPU Implementation: Arithmetic
- Ch. 4 CPU Implementation: Pipeline
- Ch. 5 Cache and Virtual Memory
- Ch. 6-7 I/O and Multiprocessors

2.4 Signed and Unsigned Numbers

- Bits are just bits (no inherent meaning)
  — conventions define relationship between bits and numbers

- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: 0...2\(^n-1\)

- 32-bit number
  - 0000 0000 ..... 0000 = 0
  - 0000 0000 ..... 0001 = 1
  - ..... 
  - 1111 1111 ..... 1110 = 4,294,967,294
  - 1111 1111 ..... 1111 = 4,294,967,295 (4Billion)
Exercise #1

• Convert to 32-bit, 2’s complement binary number : 56479
  - 56479/2 = 28239...1
  - 28239/2 = 14119...1
  - 14119/2 = 7059...1
  - 7059/2 = 3529...1
  - 3529/2 = 1764...1
  - 1764/2 = 882...0
  - 882/2 = 441...0
  - 441/2 = 220...1
  - 220/2 = 110...0
  - 110/2 = 55...0
  - 55/2 = 27...1
  - 27/2 = 13...1
  - 13/2 = 6...1
  - 6/2 = 3...0
  - 3/2 = 1...1

0000 0000 0000 0000
1101 1100 1001 1111
= 0000DC9F

Numbers

• Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers
  - e.g., no subi instruction in MIPS; addi can add a negative number

• How do we represent negative numbers?
  - i.e., which bit patterns will represent which numbers?
Possible Representations

- **Sign Magnitude:**
  - 000 = +0
  - 001 = +1
  - 010 = +2
  - 011 = +3
  - 100 = -0
  - 101 = -1
  - 110 = -2
  - 111 = -3

- **One's Complement:**
  - 000 = +0
  - 001 = +1
  - 010 = +2
  - 011 = +3
  - 100 = -3
  - 101 = -2
  - 110 = -1
  - 111 = -0

- **Two's Complement:**
  - 000 = +0
  - 001 = +1
  - 010 = +2
  - 011 = +3
  - 100 = -4
  - 101 = -3
  - 110 = -2
  - 111 = -1

- **Issues:** balance, number of zeros, ease of operations
- Which one is best? Why?
  - 1's complement looks simpler because (–a) = invert a
  - 2's complement looks complex because (-a) = invert a +1

### Exercise #2

- Convert to 32-bit, 2's complement binary number : -2048
  - 2048/2 = 1024...0
  - 1024/2 = 512...0
  - 512/2 = 256...0
  - 256/2 = 128...0
  - 128/2 = 64...0
  - 64/2 = 32...0
  - 32/2 = 16...0
  - 16/2 = 8...0
  - 8/2 = 4...0
  - 4/2 = 2...0
  - 2/2 = 1...0

  0000 0000 0000 0000
  0000 1000 0000 0000
  => (2's complement)

  1111 1111 1111 1111
  1111 0111 1111 1111 + 1
  =
  1111 1111 1111 1111
  1111 1000 0000 0000
  = FFFFE800
Exercise #3

• Convert 2’s complement binary number to hexadecimal and decimal:
  – 1111 1111 1111 1111 1111 1110 0011 1001
• To hexadecimal (don’t care about sign)
  – FFFFFFE39
• To decimal
  – Sign : negative
  – Magnitude : convert it first
    • 1111 1111 1111 1111 1110 0011 1001 => negate
    • 0000 0000 0000 0000 0000 0001 1100 0110 +1 =>
    • 0000 0000 0000 0000 0000 0001 1100 0111
    • 1+2+4+64+128+256 =>
    • 455
  – - 455

Possible Representations (Advanced Topic)

• With n-bit binary numbers (e.g. 3-bit, $2^n$=8)
  – Positive: $N < 2^{n-1}$ (e.g. 2=010)
  – Negative: $-N$ (-2)

• Sign Magnitude
  – $-N$ => sign bit=1, magnitude=10 => 110

• One’s Complement
  – $-N$ => $(2^n-1)-N = 2^n-N-1 = 8-2-1 = 5 = 101$

• Two’s Complement
  – $-N$ => $(2^n-1)-N +1 = 2^n-N = 8-2 = 6 = 110
Possible Representations

• Subtraction: a-b (a,b>0)
  – One's Complement
  • If a<b: answer = -(b-a), where (b-a)>0
    – a-b = a+(b) = a+(2^k-b) = 2^k-(b-a) : OK
    – This is exactly the representation of -(b-a): OK
  • If a>b: answer = (a-b), where (a-b)>0
    – a-b = a+(b) = a+(2^k-b) = (a-b)+2^k = (a-b)+8 : OK
    – This is not the representation of (a-b): ???
    – 2^k is just ignored, but “-1” must be compensated
  • Ex) 0111 (7) 0111 (7)
    - 0110 (6) => + 1001 (-6)
    ------------------ ------------------
    0001 (1) 10000 ➔ ignore 2^k (MSB) & “+1” (one more add)

Possible Representations

• Subtraction: a-b (a,b>0)
  – Two’s Complement
  • If a<b: answer = -(b-a), where (b-a)>0
    – a-b = a+(b) = a+(2^k-b) = 2^k-(b-a) : OK
    – This is exactly the representation of -(b-a): OK
  • If a>b: answer = (a-b), where (a-b)>0
    – a-b = a+(b) = a+(2^k-b) = (a-b)+2^k : OK
    – This is not the representation of (a-b), but 2^k is just ignored: OK
  • Ex) 0111 (7) 0111 (7)
    - 0110 (6) => + 1010 (-6)
    ------------------ ------------------
    0001 (1) 10001 ➔ 2^k (MSB), that’s it !!!

• That's why two’s complement is mostly used in digital computers ➔ Ease of subtraction !!!
MIPS

• 32 bit signed numbers:

0000 0000 0000 0000 0000 0000 0000 0000 two = 0 ten
0000 0000 0000 0000 0000 0000 0000 0001 two = + 1 ten
0000 0000 0000 0000 0000 0000 0000 0010 two = + 2 ten

... 0111 1111 1111 1111 1111 1111 1111 1110 two = + 2,147,483,646 ten
0111 1111 1111 1111 1111 1111 1111 1111 two = + 2,147,483,647 ten
1000 0000 0000 0000 0000 0000 0000 0000 two = - 2,147,483,648 ten
1000 0000 0000 0000 0000 0000 0000 0001 two = - 2,147,483,647 ten
1000 0000 0000 0000 0000 0000 0000 0010 two = - 2,147,483,646 ten

... 1111 1111 1111 1111 1111 1111 1111 1111 two = - 1 ten
1111 1111 1111 1111 1111 1111 1111 1110 two = - 2 ten
1111 1111 1111 1111 1111 1111 1111 1111 two = - 3 ten

maxint

minint

2.5 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):

```
000000 10001 10010 01000 00000 100000
```

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

• 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 ?
• 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)

```
| 35 | 18 | 8 | 32 |
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

• Regularity principle?

---

**So far…**

- **R-type**
  - `add`: 0 | reg | reg | reg | 0 | 32
  - `sub`: 0 | reg | reg | reg | 0 | 34
  - Both use 6 | 5 | 5 | 5 | 5 | 6

- **I-type**
  - `lw`: 35 | reg | reg | address
  - `sw`: 43 | reg | reg | address
  - Both use 6 | 5 | 5 | 16

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

<table>
<thead>
<tr>
<th>lw</th>
<th>35</th>
<th>9</th>
<th>8</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>18</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>sw</td>
<td>43</td>
<td>9</td>
<td>8</td>
<td>1200</td>
</tr>
</tbody>
</table>

• “Where are those instructions are stored?”

Example

• What’s the address for $A[300]$?

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
</tr>
<tr>
<td>160</td>
</tr>
<tr>
<td>.......</td>
</tr>
<tr>
<td>$???$</td>
</tr>
</tbody>
</table>

• What’s the address for add inst?

<table>
<thead>
<tr>
<th>CPU processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get inst. from memory</td>
</tr>
<tr>
<td>Get operand from memory</td>
</tr>
<tr>
<td>Execute the inst.</td>
</tr>
<tr>
<td>Repeat with the next inst. (most probably it is in the next address, 2010+4 not 2010+1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
</tr>
<tr>
<td>2010</td>
</tr>
<tr>
<td>$???$</td>
</tr>
<tr>
<td>$???$</td>
</tr>
</tbody>
</table>

• “Stored program architecture” = von Neuman (1903-1957, Hungary) architecture
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

2.6 MIPS logical instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ~(($2</td>
<td>$3)</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 ⊕ $3</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>slv $1,$2,$3</td>
<td>$1 = $2 &lt;&lt; $3</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>sr $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>srav $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>

(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 |
2.7 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

beq

| 4 | reg | reg | address |

bne

| 5 | reg | reg | address |

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
* 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 ???
        - The jumping address is 20000-7fff ~ 20000+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**

```
<table>
<thead>
<tr>
<th></th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>26</td>
</tr>
</tbody>
</table>
```

PC-relative addressing as in beq/bne instructions? NO. Simply because it can specify a larger range.

Branch address = offset || 0

=> Range: 0~2^32 (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:
  \[
  \begin{align*}
  \text{if} & \quad s1 < s2 \text{ then} \\
  & \quad t0 = 1 \\
  \text{slt} & \quad t0, s1, s2 \\
  \text{else} & \quad t0 = 0
  \end{align*}
  \]

- Can use this instruction to build "blt \( s1, s2, \text{Label} \)"
  \( \text{slt + bne = blt} \)
  — 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 */

- k are stored in $s5, the labels are stored in memory starting at $t4
- Notes: (i) slt / slti (I-type), (ii) j / jr (R-type), (iii) $zero, (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:**

<table>
<thead>
<tr>
<th>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td>26 bit address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  (add, sub) (lw, sw) (j)