EE 109 Unit 8 – MIPS Instruction Set
Architecting a vocabulary for the HW

INSTRUCTION SET OVERVIEW
Instruction Set Architecture (ISA)

- Defines the software interface of the processor and memory system
- Instruction set is the *vocabulary* the HW can understand and the SW is composed with
- 2 approaches
  - CISC = Complex instruction set computer
    - Large, rich vocabulary
    - More work per instruction but slower HW
  - RISC = Reduced instruction set computer
    - Small, basic, but sufficient vocabulary
    - Less work per instruction but faster HW
Components of an ISA

1. Data and Address Size
   – 8-, 16-, 32-, 64-bit

2. Which instructions does the processor support
   – SUBtract instruc. vs. NEGate + ADD instrucs.

3. Length and format of instructions
   – How is the operation and operands represented with 1’s and 0’s

4. Registers accessible to the instructions
   – Faster than accessing data from memory

5. Addressing Modes
   – How instructions can specify location of data operands
# Historic Progression of Data Size & Registers

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Trans. Count</th>
<th>Data Size</th>
<th>GPRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>8088</td>
<td>1979</td>
<td>29K</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>134K</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>80386/486</td>
<td>'85/'89</td>
<td>275K/1.1 8M</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
<td>32</td>
<td>&gt;8</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>42M</td>
<td>32</td>
<td>&gt;= 128</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
<td>64</td>
<td>&gt;= 128</td>
</tr>
<tr>
<td>6-core Core i7</td>
<td>2011</td>
<td>2.27B</td>
<td>64</td>
<td>&gt;= 128</td>
</tr>
<tr>
<td>MIPS</td>
<td>1999</td>
<td>var.</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>
General Instruction Format Issues

• Instructions must specify three things:
  – Operation (OpCode)
  – Source operands
    • Usually 2 source operands (e.g. X+Y)
  – Destination Location

• Example: ADD $8, $9, $10  ($8 = $9 + $10 where $ = Register)

• Binary (machine-code) representation broken into fields of bits for each part

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Src. 1</th>
<th>Src. 2</th>
<th>Dest.</th>
<th>Shift Amount</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>01001</td>
<td>01010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
<tr>
<td>Arith.</td>
<td>$9</td>
<td>$10</td>
<td>$8</td>
<td>Unused</td>
<td>Add</td>
</tr>
</tbody>
</table>
Historical Instruction Formats

• Different instruction sets specify these differently
  – 3 operand instruction set (MIPS, PPC, ARM)
    • Similar to example on previous page
    • Format: ADD DST, SRC1, SRC2 (DST = SRC1 + SRC2)
  – 2 operand instructions (Intel / Motorola 68K)
    • Second operand doubles as source and destination
    • Format: ADD SRC1, S2/D (S2/D = SRC1 + S2/D)
  – 1 operand instructions (Old Intel FP, Low-End Embedded)
    • Implicit operand to every instruction usually known as the Accumulator (or ACC) register
    • Format: ADD SRC1 (ACC = ACC + SRC1)
Historical Instruction Format Examples

- Consider the pros and cons of each format when performing the set of operations
  - \( F = X + Y - Z \)
  - \( G = A + B \)
- Simple embedded computers often use single operand format
  - Smaller data size (8-bit or 16-bit machines) means limited instruc. size
- Modern, high performance processors use 2- and 3-operand formats

<table>
<thead>
<tr>
<th>Single-Operand</th>
<th>Two-Operand</th>
<th>Three-Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD X</td>
<td>MOVE F,X</td>
<td>ADD F,X,Y</td>
</tr>
<tr>
<td>ADD Y</td>
<td>ADD F,Y</td>
<td>SUB F,F,Z</td>
</tr>
<tr>
<td>SUB Z</td>
<td>SUB F,Z</td>
<td>ADD G,A,B</td>
</tr>
<tr>
<td>STORE F</td>
<td>MOVE G,A</td>
<td></td>
</tr>
<tr>
<td>LOAD A</td>
<td>ADD G,B</td>
<td></td>
</tr>
<tr>
<td>ADD B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE G</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(+): Smaller size to encode each instruction
(-): Higher instruction count to load and store ACC value

Compromise of two extremes

(+): More natural program style
(+): Smaller instruction count
(-): Larger size to encode each instruction
MIPS Instruction Format

- **3 Register operand format**
  - Most ALU instructions use 3 registers as their operands
  - All operations are performed on entire 32-bits (no size distinction)
  - Example: ADD $t0, $t1, $t2

- **Load/Store architecture**
  - Load (read) data values from memory into a register
  - Perform operations on registers
  - Store (write) data values back to memory
  - Different load/store instructions for different operand sizes (i.e. byte, half, word)

**Load/Store Architecture**

1.) Load operands to proc. registers

2.) Proc. Performs operation using register values

3.) Store results back to memory
Which Instructions

• In this class we'll focus on assembly to do the following tasks *(shown with the corresponding MIPS assembly mnemonics)*
  
  – Load variables (data) from memory (or I/O) *[LW,LH,LB]*
  
  – Perform arithmetic, logical, and shift instructions in the CPU *[ADD,SUB,AND,OR,SLL,SRL,SRA]*
  
  – Store variables (data) back to memory after computation is complete *[SW, SH, SB]*
  
  – Compare data *[SLT]*
  
  – "Branch" to other code (to implement if and loops) *[BEQ,BNE,J]*
  
  – Call subroutines/functions *[JAL, JR]*
MIPS INSTRUCTION OVERVIEW
MIPS ISA

• RISC Style

• 32-bit internal / 32-bit external data size
  – Registers and ALU are 32-bits wide
  – Memory bus is logically 32-bits wide (though may be physically wider)

• Registers
  – 32 General Purpose Registers (GPR’s)
    • For integer and address values
    • A few are used for specific tasks/values
  – 32 Floating point registers

• Fixed size instructions
  – All instructions encoded as a single 32-bit word
  – Three operand instruction format (dest, src1, src2)
  – Load/store architecture (all data operands must be in registers and thus loaded from and stored to memory explicitly)
# MIPS GPR’s

<table>
<thead>
<tr>
<th>Assembler Name</th>
<th>Reg. Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>Constant 0 value</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>Assembler temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>$2-$3</td>
<td>Procedure return values or expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>$4-$7</td>
<td>Arguments/parameters</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>$8-$15</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>$16-$23</td>
<td>Saved Temporaries</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>$24-$25</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>$26-$27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>Global Pointer (Global and static variables/data)</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>Return address for current procedure</td>
</tr>
</tbody>
</table>

*Avoid using the yellow (highlighted) registers for anything other than its stated use*
MIPS Programmer-Visible Registers

- **General Purpose Registers (GPR’s)**
  - Hold data operands or addresses (pointers) to data stored in memory

- **Special Purpose Registers**
  - **PC**: Program Counter (32-bits)
    - Holds the address of the next instruction to be fetched from memory & executed
  - **HI**: Hi-Half Reg. (32-bits)
    - For MUL, holds 32 MSB’s of result. For DIV, holds 32-bit remainder
  - **LO**: Lo-Half Reg. (32-bits)
    - For MUL, holds 32 LSB’s of result. For DIV, holds 32-bit quotient

Recall multiplying two 32-bit numbers yields a 64-bit result.
Instruction Format

• 32-bit Fixed Size Instructions broken into 3 types (R-, I-, and J-) based on which bits mean what...

• R-Type
  – Arithmetic/Logic instructions
  – 3 register operands or shift amount

• I-Type
  – Use for data transfer, branches, etc.
  – 2 registers + 16-bit const.

• J-Type
  – 26-bit jump address
  – We'll cover this later

Each type uses portions of the instruction to "code" certain aspects of the instruction. But they all start with an opcode that helps determine which type will be used.
Performing Arithmetic, Logic, and Shift Operations

IMPORTANT R-TYPE INSTRUCTIONS
R-Type Instructions

• Format

<table>
<thead>
<tr>
<th>6-bits</th>
<th>5-bits</th>
<th>5-bits</th>
<th>5-bits</th>
<th>5-bits</th>
<th>6-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs (src1)</td>
<td>rt (src2)</td>
<td>rd (dest)</td>
<td>shamt</td>
<td>function</td>
</tr>
</tbody>
</table>

– rs, rt, rd are 5-bit fields for register numbers
– shamt = shift amount and is used for shift instructions indicating # of places to shift bits
– opcode and func identify actual operation (e.g. ADD, SUB)

• Example:
– ADD $5, $24, $17

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>11000</td>
<td>10001</td>
<td>00101</td>
<td>00000</td>
<td>100000</td>
</tr>
<tr>
<td>Arith. Inst.</td>
<td>$24</td>
<td>$17</td>
<td>$5</td>
<td>unused</td>
<td>ADD</td>
</tr>
</tbody>
</table>
## R-Type Arithmetic/Logic Instructions

<table>
<thead>
<tr>
<th>C operator</th>
<th>Assembly</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>ADD Rd, Rs, Rt</td>
<td>d=destination, s = src1, t = src2</td>
</tr>
<tr>
<td>-</td>
<td>SUB Rd, Rs, Rt</td>
<td>Order: R[s] – R[t]. SUBU for unsigned</td>
</tr>
<tr>
<td>*</td>
<td>MULT Rs, Rt, MULTU Rs, Rt</td>
<td>Result in HI/LO. Use mfhi and mflo instruction to move results</td>
</tr>
<tr>
<td>*</td>
<td>MUL Rd, Rs, Rt</td>
<td>If multiply won’t overflow 32-bit result</td>
</tr>
<tr>
<td>/</td>
<td>DIV Rs, Rt, DIVU Rs, Rt</td>
<td>R[s] / R[t]. Remainder in HI, quotient in LO</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND Rd, Rs, Rt</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR Rd, Rs, Rt</td>
</tr>
<tr>
<td>^</td>
<td>XOR Rd, Rs, Rt</td>
<td>Can be used for bitwise-NOT (~)</td>
</tr>
<tr>
<td>~(</td>
<td>)</td>
<td>NOR Rd, Rs, Rt</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>SLL Rd, Rs, shamt, SLLV Rd, Rs, Rt</td>
<td>Shifts R[s] left by shamt (shift amount) or R[t] bits</td>
</tr>
<tr>
<td>&gt;&gt; (signed)</td>
<td>SRA Rd, Rs, shamt, SRAV Rd, Rs, Rt</td>
<td>Shifts R[s] right by shamt or R[t] bits replicating sign bit to maintain sign</td>
</tr>
<tr>
<td>&gt;&gt; (unsigned)</td>
<td>SRL Rd, Rs, shamt, SRLV Rd, Rs, Rt</td>
<td>Shifts R[s] left by shamt or R[t] bits shifting in 0’s</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>SLT Rd, Rs, Rt, SLTU Rd, Rs, Rt</td>
<td>Order: R[s] – R[t]. Sets R[d]=1 if R[s] &lt; R[t], 0 otherwise</td>
</tr>
</tbody>
</table>
Logical Operations

• Should already be familiar with (sick of) these! 😊
• Logic operations are usually performed on a pair of bits

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X1</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**AND** – Output is true if **both** inputs are true
0 AND x = 0
1 AND x = x
x AND x = x

**OR** – Output is true if **any** input is true
0 OR x = x
1 OR x = 1
x OR x = x

**XOR** – Output is true if **exactly one** input is true
0 XOR x = x
1 XOR x = NOT x
x XOR x = 0

**NOT** – Output is **inverse** of input
Logical Operations

- Logic operations on numbers means performing the operation on each pair of bits

**Initial Conditions:** $1 = 0xF0$, $2 = 0x3C$

1. **AND $2$, $1$, $2$**
   
   $R[2] = 0x30$

2. **OR $2$, $1$, $2$**
   
   $R[2] = 0xFC$

3. **XOR $2$, $1$, $2$**
   
   $R[2] = 0xCC$
Logical Operations

• Logic operations on numbers means performing the operation on each pair of bits

Initial Conditions: $1 = \text{0xF0}, \quad 2 = \text{0x3C}$

1. NOR $2, 1, 2$
   \[ \text{R}[2] = \text{0x03} \]

Bitwise NOT operation can be performed by NOR’ing register with itself

\begin{align*}
\text{NOR} & \quad 0xF0 \quad 0xF0 \quad 1111\ 0000 \\
\text{NOR} & \quad 0x3C \quad 0x03 \quad 0000\ 0011
\end{align*}
Shift Operations

• Shifts data bits either left or right
  – Bits shifted out and dropped on one side
  – Usually (but not always) 0’s are shifted in on the other side

• In addition to just moving bits around, **shifting is a fast way to multiply or divide a number by powers of 2** (see next slides)

• 2 kinds of shifts
  – **Logical shifts (used for unsigned numbers)**
  – **Arithmetic shifts (used for signed numbers)**

### Right Shift by 2 bits:

```
0 0 0 0 1 1 0 0
Original Data
```

```
0 0 0 0 0 1 1 0
Shifted by 2 bits
```

0’s shifted in...

### Left Shift by 2 bits:

```
0 0 0 0 1 0 1 0
Original Data
```

```
0 0 0 0 0 0 1 1
Shifted by 2 bits
```

0’s shifted in...
Logical Shift vs. Arithmetic Shift

• **Logical Shift**
  – Use for unsigned or non-numeric data
  – Will always shift in 0’s whether it be a left or right shift

• **Arithmetic Shift**
  – Use for signed data
  – Left shift will shift in 0’s
  – Right shift will sign extend (replicate the sign bit) rather than shift in 0’s
    • If negative number...stays negative by shifting in 1’s
    • If positive...stays positive by shifting in 0’s

---

Left shift

Right shift

Copies of MSB are shifted in
Logical Shift

- 0’s shifted in
- Only use for operations on **unsigned** data
  - Right shift by n-bits = Dividing by $2^n$
  - Left shift by n-bits = Multiplying by $2^n$

\[
\begin{align*}
0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0 \times C \\
\begin{array}{cccccccc}
0 & \ldots & 0 & 1 & 1 & 0 & 0 \\
\end{array} &= +12
\end{align*}
\]

 Logical Right Shift by 2 bits:

\[
\begin{align*}
0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0 \\
\begin{array}{cccccccc}
0 & 0 & \ldots & 0 & 0 & 1 & 1 \\
\end{array} &= +3
\end{align*}
\]

 Logical Left Shift by 3 bits:

\[
\begin{align*}
0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 6 \times 0 \\
\begin{array}{cccccccccccc}
\ldots & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{array} &= +96
\end{align*}
\]
Arithmetic Shift

- Use for operations on \textit{signed} data
- Arithmetic Right Shift – replicate MSB
  - Right shift by \(n\)-bits = Dividing by \(2^n\)
- Arithmetic Left Shift – shifts in 0’s
  - Left shift by \(n\)-bits = Multiplying by \(2^n\)

\[
0 \times \text{FFFFFFFC} \\ 1 \ldots 1 \ 1 \ 0 \ 0 = -4
\]

\textbf{Arithmetic Right Shift by 2 bits:}

\[
0 \times \text{FFFFFFFF} \\ 1 \ldots 1 \ 1 \ 0 \ 0 = -1
\]

\textit{MSB replicated and shifted in...}

\textbf{Arithmetic Left Shift by 2 bits:}

\[
0 \times \text{FFFFFFF0} \\ 1 \ldots 1 \ 0 \ 0 \ 0 \ 0 = -16
\]

\textit{0’s shifted in...}

\textit{Notice if we shifted in 0’s (like a logical right shift) our result would be a positive number and the division wouldn’t work}

\textit{Notice there is no difference between an arithmetic and logical left shift. We always shift in 0’s.}
MIPS Logical Shift Instructions

- **SRL instruction** – Shift Right Logical
- **SLL instruction** – Shift Left Logical

**Format:**
- $SxL \ rd, \ rt, \ shamt$
- $SxLV \ rd, \ rt, \ rs$

**Notes:**
- shamt limited to a 5-bit value (0-31)
- $SxLV$ shifts data in $rt$ by number of places specified in $rs$

**Examples**
- **SRL** $5, \ 12, \ 7$  // Shifts data in reg. $12$ right by $7$ places
- **SLLV** $5, \ 12, \ 20$  // If $20=5$, shift data in $12$ left by $5$ places

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0000</td>
<td>01100</td>
<td>00101</td>
<td>00111</td>
<td>000010</td>
</tr>
<tr>
<td>Arith. Inst.</td>
<td>unused</td>
<td>$12$</td>
<td>$5$</td>
<td>$7$</td>
<td>SRL</td>
</tr>
<tr>
<td>000000</td>
<td>10100</td>
<td>01100</td>
<td>00101</td>
<td>00000</td>
<td>000100</td>
</tr>
<tr>
<td>Arith. Inst.</td>
<td>$20$</td>
<td>$12$</td>
<td>$5$</td>
<td>unused</td>
<td>SLLV</td>
</tr>
</tbody>
</table>
MIPS Arithmetic Shift Instruction

• SRA instruction – Shift Right Arithmetic
• No arithmetic left shift (use SLL for arithmetic left shift)
• Format:
  – SRA rd, rt, shamt
  – SRAV rd, rt, rs
• Notes:
  – shamt limited to a 5-bit value (0-31)
  – SRAV shifts data in rt by number of places specified in rs
• Examples
  – SRA $5, $12, 7
  – SRAV $5, $12, $20
I-Type Instructions

• I-Type (Immediate) Format

<table>
<thead>
<tr>
<th>6-bits</th>
<th>5-bits</th>
<th>5-bits</th>
<th>16-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs (src1)</td>
<td>rt (src/dst)</td>
<td>immediate</td>
</tr>
</tbody>
</table>

– rs, rt are 5-bit fields for register numbers
– I = Immediate is a 16-bit constant
– opcode identifies actual operation

• Example:

– ADDI $5, $24, 1
– LW $5, -8($3)

LW is explained in the next section but is an example of an instruction using the I-type format
Immediate Operands

• Most ALU instructions also have an immediate form to be used when one operand is a constant value.

• Syntax: ADDI Rs, Rt, imm
  – Because immediates are limited to 16-bits, they must be extended to a full 32-bits when used by the processor.
  – Arithmetic instructions always **sign-extend** to a full 32-bits even for unsigned instructions (addiu).
  – Logical instructions always **zero-extend** to a full 32-bits.

• Examples:
  – ORI  $10, $14, -4  // R[10] = R[14] | 0x0000FFFC

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>ANDI</td>
</tr>
<tr>
<td>ADDIU</td>
<td>ORI</td>
</tr>
<tr>
<td>SLTI</td>
<td>XORI</td>
</tr>
<tr>
<td>SLTIU</td>
<td></td>
</tr>
</tbody>
</table>

Note: SUBI is unnecessary since we can use ADDI with a negative immediate value.
Set If Less-Than

- **SLT $rd, $rs, $rt**
  - Compares $rs value with $rt value and stores Boolean (1 = true, 0 = false) value into $rd
  - C code equivalent: `bool rd = (rs < rt);`
  - $rd can only be 0x00000001 or 0x00000000 after execution
  - Assumes signed integer comparison

- **SLTI $rd, $rs, immediate**
  - Same as above but now 2\(^{nd}\) source is a constant

**Initial Conditions:**

- $1 = 0xffffffff,
- $2 = 0x00000000
- $3 = 0x000000ff

**Examples:**

- **SLT $4, $1, $2**
  - $4 = 0x00000001

- **SLT $4, $3, $3**
  - $4 = 0x00000000

- **SLTI $4, $2, 35**
  - $4 = 0x00000001

- **SLTI $4, $3, -7**
  - $4 = 0x00000000
Loading (Reading) and Storing (Writing) Data From and To Memory

DATA TRANSFER AND MEMORY ACCESS INSTRUCTIONS
Physical Memory Organization

- Physical view of memory as large 2-D array of bytes (8K rows by 1KB columns) per chip (and several chips)
- Address is broken into fields of bits that are used to identify where in the array the desired 32-bit word is
  - Processor always accesses memory chunks the size of the data bus, selecting only the desired bytes as specified by the instruction
MIPS Supported Data Sizes

Integer
• 3 Sizes Defined
  – Byte (B)
    • 8-bits
  – Halfword (H)
    • 16-bits = 2 bytes
  – Word (W)
    • 32-bits = 4 bytes

Floating Point
• 3 Sizes Defined
  – Single (S)
    • 32-bits = 4 bytes
  – Double (D)
    • 64-bits = 8 bytes
    • (For a 32-bit data bus, a double would be accessed from memory in 2 reads)
MIPS Memory Organization

• We can logically picture memory in the units (sizes) that we actually access them

• Most processors are byte-addressable
  – Every byte (8-bits) has a unique address
  – 32-bit address bus => 4 GB address space

• However, 32-bit logical data bus allows us to access 4-bytes of data at a time

• Logical view of memory arranged in rows of 4-bytes
  – Still with separate addresses for each byte

Recall variables live in memory & need to be loaded into the processor to be used.

Logical Byte-Oriented View of Mem.

Logical Word-Oriented View
Memory & Data Size

- Little-endian memory can be thought of as right justified
- Always provide the **LS-Byte address** of the desired data
- Size is explicitly defined by the instruction used
- Memory Access Rules
  - Halfword or Word access **must** start on an address that is a multiple of that data size (i.e. half = multiple of 2, word = multiple of 4)

### Registers:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Half</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 15 0</td>
<td>31   0</td>
<td></td>
</tr>
</tbody>
</table>

### (Assume start address = N)

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
</tr>
<tr>
<td>LH</td>
</tr>
<tr>
<td>LW</td>
</tr>
</tbody>
</table>

- **Byte operations only access the byte at the specified address**
- **Halfword operations access the 2-bytes starting at the specified address**
- **Word operations access the 4-bytes starting at the specified address**

### Instructions:

- **LB**: Used to load a 1-byte var. (char)
- **LH**: Used to load a 2-byte variable (int)
Memory Read Instructions (Signed)

**LB (Load Byte)**
Provide address of desired byte

- If address = 0x02
- Reg. = 0x00000013

**LH (Load Half)**
Provide address of starting byte

- If address = 0x00
- Reg. = 0xFFFFF87C

**LW (Load Word)**
Provide address of starting byte

- If address = 0x00
- Reg. = 0x5A13F87C
Memory Read Instructions (Unsigned)

**GPR**

<table>
<thead>
<tr>
<th>31</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Extend</td>
<td>Byte</td>
<td></td>
</tr>
</tbody>
</table>

**LBU (Load Byte)**

Provide address of desired byte

If address = 0x01
Reg. = 0x000000F8

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Extend</td>
<td>Half</td>
<td></td>
</tr>
</tbody>
</table>

**LHU (Load Half)**

Provide address of starting byte

If address = 0x00
Reg. = 0x0000F87C

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td></td>
</tr>
</tbody>
</table>

**LW (Load Word)**

Provide address of starting byte

If address = 0x00
Reg. = 0x5A13F87C

**Memory**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>000004</td>
<td>000000</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5A 13 F8 7C</td>
<td>000004</td>
<td>000000</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>000004</td>
<td>000000</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5A 13 F8 7C</td>
<td>000004</td>
<td>000000</td>
<td></td>
</tr>
</tbody>
</table>
Memory Write Instructions

**SB (Store Byte)**
- Provide address of desired byte
- Reg. = 0x12345678

**SH (Store Half)**
- Provide address of starting byte
- Reg. = 0x12345678

**SW (Store Word)**
- Provide address of starting byte
- Reg. = 0x12345678

---

**Memory**

- If address = 0x02
  - SB (Store Byte): 000004 000000
  - SH (Store Half): 000004 000000
  - SW (Store Word): 000004 000000
MIPS Memory Alignment Limitations

- Bytes can start at any address
- Halfwords must start on an even address
- Words must start on an address that is a multiple of 4
- Examples:
  - Word @ A18C – good (multiple of 4)
  - Halfword @ FFE6 – good (even)
  - Word @ A18E – invalid (non-multiple of 4)
  - Halfword @ FFE5 – invalid (odd)
Load Format (LW,LH,LB)

• Syntax: LW $rt, offset($rs)
  – $rt = Destination register
  – offset($rs) = Address of desired data
  – Operation: $rt = Mem[offset + $rs]
  – offset limited to 16-bit signed number

• Examples
  – LW $2, 0x40($3)  // $2 = 0x5A12C5B7
  – LBU $2, -1($4)  // $2 = 0x000000F8
  – LH $2, 0xFFFC($4)  // $2 = 0xFFFF97CD

<table>
<thead>
<tr>
<th>$2</th>
<th>old val.</th>
<th>F8BE97CD</th>
<th>0x002048</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3</td>
<td>00002000</td>
<td>134982FE</td>
<td>0x002044</td>
</tr>
<tr>
<td>$4</td>
<td>0000204C</td>
<td>5A12C5B7</td>
<td>0x002040</td>
</tr>
</tbody>
</table>

Registers | Memory | Address
More LOAD Examples

• Examples
  - LB $2, 0x45($3)  // $2 = 0xFFFFFFFFF82
  - LH $2, -6($4)    // $2 = 0x00001349
  - LHU $2, -2($4)   // $2 = 0x0000F8BE

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2</td>
<td>0x000204C</td>
<td>0x002040</td>
</tr>
<tr>
<td>$3</td>
<td>0x0002000</td>
<td>0x002044</td>
</tr>
<tr>
<td>$4</td>
<td>F8BE97CD</td>
<td>002048</td>
</tr>
<tr>
<td></td>
<td>134982FE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5A12C5B7</td>
<td></td>
</tr>
</tbody>
</table>
Store Format (SW,SH,SB)

- **SW $rt, offset($rs)**
  - $rt = Source register
  - offset($rs) = Address to store data
  - Operation: Mem[ offset + $rs ] = $rt
  - offset limited to 16-bit signed number

- **Examples**
  - SW $2, 0x40($3)
  - SB $2, -5($4)
  - SH $2, 0xFFFE($4)

### Registers
- $2: 123489AB
- $3: 00002000
- $4: 0000204C

### Memory
- **Address**
  - 0x002048
  - 0x002044
  - 0x002040

- **89AB97CD**
- **123489AB**
Loading an Immediate

• If immediate (constant) 16-bits or less
  – Use ORI or ADDI instruction with $0 register
  – Examples
    • ADDI $2, $0, 1 // $2 = 0 + 1 = 1
    • ORI $2, $0, 0xF110 // $2 = 0 | 0xF110 = 0xF110

• If immediate more than 16-bits
  – Immediates limited to 16-bits so we must load constant with a 2 instruction sequence using the special LUI (Load Upper Immediate) instruction
  – To load $2 with 0x12345678
    • LUI $2,0x1234
    • ORI $2,$2,0x5678

$2 12340000  LUI
OR 00005678
$2 12345678  ORI
"Be the Compiler"

TRANSLATING HIGH-LEVEL CODE
Translating HLL to Assembly

- HLL variables are simply locations in memory
  - A variable name really translates to an address in assembly

<table>
<thead>
<tr>
<th>C operator</th>
<th>Assembly</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x,y,z;</td>
<td>LUI $8, 0x1000 ORI $8, $8, 0x0004 LW $9, 4($8) LW $10, 8($8) ADD $9,$9,$10 SW $9, 0($8)</td>
<td>Assume x @ 0x10000004 &amp; y @ 0x10000008 &amp; z @ 0x1000000C</td>
</tr>
<tr>
<td>... x = y + z;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char a[100];</td>
<td>LUI $8, 0x1000 ORI $8, $8, 0x0000C LB $9, 1($8) ADDI $9,$9,-1 SB $9,1($8)</td>
<td>Assume array ‘a’ starts @ 0x10000000C</td>
</tr>
<tr>
<td>... a[1]--;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Translating HLL to Assembly

<table>
<thead>
<tr>
<th>C operator</th>
<th>Assembly</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int dat[4], x; … x = dat[0]; x += dat[1];</td>
<td>LUI $8, 0x1000 ORI $8, $8, 0x0010 LW $9, 0($8) LW $10, 4($8) ADD $9,$9,$10 SW $9, 16($8)</td>
<td>Assume dat @ 0x10000010 &amp; x @ 0x10000020</td>
</tr>
<tr>
<td>unsigned int y; short z; y = y / 4; z = z &lt;&lt; 3;</td>
<td>LUI $8, 0x1000 ORI $8, $8, 0x0010 LW $9, 0($8) SRL $9, $9, 2 SW $9, 0($8) LH $9, 4($8) SLA $9, $9, 3 SH $9, 4($8)</td>
<td>Assume y @ 0x10000010 &amp; z @ 0x10000014</td>
</tr>
</tbody>
</table>
Not Used in Fall 2014

OLD
Shorthand Notation

- Shorthand notation for describing processor / memory (assembly) operations
- \( R[n] = \) value of GPR \( n \)
- \( F[n] = \) value of FP reg. \( n \)
- \( M[n] = \) value of memory at address \( n \)
- Examples
  - \( R[5] = M[40 + R[1]] \)
  - Use the value of R1 + 40 as the address to read memory & place data in R5
Unsigned Multiplication Review

- Same rules as decimal multiplication
- Multiply each bit of Q by M shifting as you go
- An m-bit * n-bit mult. produces an m+n bit result
- Notice each partial product is a shifted copy of M or 0 (zero)

\[
\begin{array}{c}
1010 \\
* 1011 \\
\hline
1010 \\
1010_\text{PP (Partial Products)} \\
0000_\text{PP (Partial Products)} \\
+ 1010 \\
\hline
01101110 \text{ P (Product)}
\end{array}
\]

M (Multiplicand)  
Q (Multiplier)
Signed Multiplication Techniques

- When multiplying signed (2’s comp.) numbers, some new issues arise
- Must sign extend partial products (out to 2n bits)

**Without Sign Extension... Wrong Answer!**

\[
\begin{align*}
1001 & \quad = -7 \\
\times 0110 & \quad = +6 \\
\hline
0000 & \\
1001_\text{__} & \\
1001_\text{__} & \\
+ 0000 & \\
\hline
00110110 & \quad = +54
\end{align*}
\]

**With Sign Extension... Correct Answer!**

\[
\begin{align*}
1001 & \quad = -7 \\
\times 0110 & \quad = +6 \\
\hline
00000000 & \\
1111001_\text{__} & \\
111001_\text{__} & \\
+ 00000 & \\
\hline
11010110 & \quad = -42
\end{align*}
\]
Signed Multiplication Techniques

- Also, must worry about negative multiplier
  - MSB of multiplier has negative weight
  - If MSB=1, multiply by -1 (i.e. take 2’s comp. of multiplicand)

```
1100  =  -4
* 1010  =  -6
00000000
1111100__
000000__
+ 11100    =  -40
11011000  =  -40

1100  =  -4
* 1010  =  -6
00000000
1111100__
000000__
+ 00100    =  +24
00011000  =  +24
```

Main Point: Signed and Unsigned Multiplication require different techniques...Thus different instructions.