EE 109 Unit 8 – MIPS Instruction Set

Architecting a vocabulary for the HW

INSTRUCTION SET OVERVIEW

Instruction Set Architecture (ISA)

- Defines the ________________ of the processor and memory system
- Instruction set is the ___________ the HW can understand and the SW is composed with
- 2 approaches
  - ______ = __________ instruction set computer
    - Large, rich vocabulary
    - More work per instruction but slower HW
  - ______ = __________ instruction set computer
    - Small, basic, but sufficient vocabulary
    - Less work per instruction but faster HW

Components of an ISA

1. ______ and Address Size
   - 8-, 16-, 32-, 64-bit
2. Which instructions does the processor support
   - SUBtract instruc. vs. NEGate + ADD instrucs.
3. ___________________ of instructions
   - How is the operation and operands represented with 1’s and 0’s
4. __________ accessible to the instructions
   - Faster than accessing data from memory
5. Addressing Modes
   - How instructions can specify location of data operands
8.5 Historic Progression of Data Size & Registers

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

8.6 General Instruction Format Issues

- Instructions must specify three things:
  - Source operands
    - Usually 2 source operands (e.g. X+Y)
  - 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>
</tbody>
</table>

- Arith. $9 $10 $8 Unused Add

8.7 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)

8.8 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>ADD F,F,Z</td>
</tr>
<tr>
<td>SUB Z</td>
<td>SUB F,Z</td>
<td>ADD G,A</td>
</tr>
<tr>
<td>STORE F</td>
<td>MOVE G,A</td>
<td>ADD G,B</td>
</tr>
<tr>
<td>LOAD A</td>
<td></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

- __________ 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

- _______ 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 ISA

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

- Registers
  - _______ 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 ______-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-$23</td>
<td>Temporaries Saved Temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>$24-$25</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$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**
  - HI: Hi-Half Reg. (32-bits)
    - Holds the address of the next instruction to be fetched from memory & executed
  - LO: Lo-Half Reg. (32-bits)
    - For MUL, holds 32 MSB's of result. For DIV, holds 32-bit remainder

**Instruction Format**

- 32-bit Fixed Size Instructions broken into 3 types (R-, I-, and J-)
  - Based on which bits ____________
    - Arithmetic/Logic instructions
      - 3 register operands or shift amount
    - Use for data transfer, branches, etc.
      - 2 registers + 16-bit const.
    - 26-bit jump address
      - We’ll cover this later

**IMPORTANT R-TYPE INSTRUCTIONS**

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.
R-Type Instructions

- Format
  - 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

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 Rs, Rt, Rt</td>
<td>Order: R[s] – R[t]; SUBU for unsigned</td>
</tr>
<tr>
<td>*</td>
<td>MULT Rs, Rt</td>
<td>Result in HI/LO. Use mfhi and mflo instruction to move results</td>
</tr>
<tr>
<td>/</td>
<td>DIV 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></td>
</tr>
<tr>
<td>~( )</td>
<td>NOR Rd, Rs, Rt</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>SLR Rd, Rs, shamt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SLLV Rd, Rs, Rt</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>SRA Rd, Rs, shamt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SRAV Rd, Rs, Rt</td>
<td></td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>SLT Rd, Rs, Rt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SLTU Rd, Rs, Rt</td>
<td></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>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>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>

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

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

1. \(\text{AND} \quad 0xF0, 0x3C, 0x3C \quad \Rightarrow \quad 0xF0\)
2. \(\text{OR} \quad 0xF0, 0x3C, 0x3C \quad \Rightarrow \quad 0xF0\)
3. \(\text{XOR} \quad 0xF0, 0x3C, 0x3C \quad \Rightarrow \quad 0xF0\)

- \(R[2] = 0x30\)
- \(R[2] = 0xFC\)
- \(R[2] = 0xCC\)
Logical Operations

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

<table>
<thead>
<tr>
<th>Initial Conditions: $1 = 0xF0, $2 = 0x3C</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOR $2,$1,$2 ➞ 0xF0 ➞ 1111 0000</td>
</tr>
<tr>
<td>NOR $2,$1,$2 ➞ 0x3C ➞ 0011 1100</td>
</tr>
<tr>
<td>NOR $2,$1,$2 ➞ R[2] = 0x03 ➞ 0000 0011</td>
</tr>
</tbody>
</table>

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

| NOR \$2,\$1,\$1 ➞ 0xF0 ➞ 1111 0000         |
| NOR \$2,\$1,\$1 ➞ 0x3C ➞ 0011 1100         |
| NOR \$2,\$1,\$1 ➞ R[2] = 0x0F ➞ 0000 1111 |

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 ____________ (see next slides)
- 2 kinds of shifts
  - Logical shifts (used for ____________ numbers)
  - Arithmetic shifts (used for ____________ numbers)

Logical Shift vs. Arithmetic Shift

- Logical Shift
  - Use for unsigned or non-numeric data
  - Will always shift in ______ whether it be a left or right shift

- Arithmetic Shift
  - Use for signed data
  - Left shift will shift in ______
  - Right shift will ______ (________ the sign bit) rather than shift in 0's
    - If negative number...stays negative by shifting in ______
    - If positive...stays positive by shifting 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 $2^n$
8.25 Arithmetic Shift

- Use for operations on 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 $2^n$

```
0xFFFFFFFF = -4  
1...11000 = -1  
0xFF000000 = -16
```

Arithmetic Right Shift by 2 bits: 
MSB replicated and shifted in...

Arithmetic Left Shift by 2 bits: 
0's shifted in...

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.

8.26 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
- SLL $5$, $12$, $20$ // If $20=5$, shift data in $12$ left by 5 places

```
opcode  rs  rt  rd  shamt  func
000000  0000  01100  00101  00111  00010  SRL
000000  1000  01100  00101  00000  00010  SLLV
```

8.27 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 $5$, $12$, $20$

```
opcode  rs  rt  rd  shamt  func
000000  0000  01100  00101  00111  00010  SRA
000000  10100  01100  00101  00000  00100  SRAV
```

8.28 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.
  - __________ instructions always sign-extend to a full 32-bits even for unsigned instructions (addiu)
  - __________ 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>SLTI</td>
<td>ORI</td>
</tr>
<tr>
<td></td>
<td>XORI</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: _____________
  - $rd can only be 0x00000001 or 0x00000000 after execution
  - Assumes signed integer comparison
- SLTI $rd, $rs, immediate
  - Same as above but now 2nd source is a constant

Initial Conditions:
$1 = 0xffffffff,
$2 = 0x00000000
$3 = 0x00000000

SLT $4, $1, $2
$4 = 0x00000000

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

SLTI $4, $2, $35
$4 = 0x00000000

SLTI $4, $3, $1
$4 = 0x00000000

SLTI $4, $3, $7
$4 = 0x00000000

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 ________
  - 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 ________ of data at a time
- Logical view of memory arranged in rows of 4-bytes
  - Still with separate addresses for each byte

**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)

**Memory Read Instructions (Signed)**

**Register: (Assume start address = N)**
- **Byte**
  - LB: Used to load a 1-byte var. (char)
  - N+3  N+2  N+1  N
  - Byte operations only access the byte at the specified address
- **Half**
  - LH: Used to load a 2-byte variable (short)
  - N+3  N+2  N+1  N
  - Halfword operations access the 2-bytes starting at the specified address
- **Word**
  - LW: Used to load a 4-byte variable (int)
  - N+3  N+2  N+1  N
  - Word operations access the 4-bytes starting at the specified address

**Memory**:
- **LB (Load Byte)**
  - Provide address of desired byte
  - If address = 0x02
    - Reg. = 0x_____
  - If address = 0x00
    - Reg. = 0x_____
- **LH (Load Half)**
  - Provide address of starting byte
  - If address = 0x00
    - Reg. = 0x_____
- **LW (Load Word)**
  - Provide address of starting byte
  - If address = 0x00
    - Reg. = 0x_____
Memory Read Instructions (Unsigned)

- **LBU (Load Byte)**
  - Provide address of desired byte
  - Example:
    - If address = 0x01
      - Reg. = 0x__________
    - If address = 0x00
      - Reg. = 0x__________

- **LHU (Load Half)**
  - Provide address of starting byte
  - Example:
    - If address = 0x00
      - Reg. = 0x__________

- **LW (Load Word)**
  - Provide address of starting byte
  - Example:
    - If address = 0x00
      - Reg. = 0x__________

Memory Write Instructions

- **SB (Store Byte)**
  - Provide address of desired byte
  - Example:
    - If address = 0x02
      - Reg. = 0x12345678

- **SH (Store Half)**
  - Provide address of starting byte
  - Example:
    - If address = 0x02
      - Reg. = 0x12345678

- **SW (Store Word)**
  - Provide address of starting byte
  - Example:
    - If address = 0x00
      - Reg. = 0x12345678

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 –
  - Halfword @ FFE6 –
  - Word @ A18E –
  - Halfword @ FFE5 –

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 = ________________
  - LBU $2, -1($4) // $2 = ________________
  - LH $2, 0xFFFC($4) // $2 = ________________
More LOAD Examples

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

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)

Loading an Immediate

• If immediate (constant) ______________________
  - Use ______ or _________ instruction with _____ register
  - Examples
    - ADDI $2, $0, 1    // $2 = 0 + 1 = 1  (Loads const. 1 into $2)
    - ORI $2, $0, 0xF110 // $2 = 0 | 0xF110 = 0xF110  (Loads 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
    _________________________________ instruction
  - To load $2 with 0x12345678
    - _______________________
    - _______________________

"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 memory.

<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</td>
<td>Assume x @ 0x10000004</td>
</tr>
<tr>
<td></td>
<td>ORI $8, $8, 0x0004</td>
<td>&amp; y @ 0x10000008 &amp; z @ 0x1000000C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assume array 'a' starts @ 0x10000000C</td>
</tr>
<tr>
<td>x = y + z;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x += dat[1];</td>
<td>LW  $9, 0($8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LW $9, 0($8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SW $9, 0($8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LH $9, 4($8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SH $9, 4($8)</td>
<td></td>
</tr>
<tr>
<td>char a[100];</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>