Instruction encoding

• The ISA defines
  – The format of an instruction (syntax)
  – The meaning of the instruction (semantics)

• Format = Encoding
  – Each instruction format has various fields
  – Opcode field gives the semantics (Add, Load etc …)
  – Operand fields (rs,rt,rd,immed) say where to find inputs (registers, constants) and where to store the output
MIPS Instruction encoding

• MIPS = RISC hence
  – Few (3+) instruction formats

• Recall that R in RISC also stands for “Regular”
  – All instructions of the same length (32-bits = 4 bytes)
  – Formats are consistent with each other
    • Opcode always at the same place (6 most significant bits)
    • rd (destination register) and rs (one of the source registers) always at the same place
    • immed always at the same place
I-type (immediate) format

- An instruction with an immediate constant has the SPIM form:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>$4,$7,78</td>
<td>$4 = $7 + 78</td>
</tr>
</tbody>
</table>

- Encoding of the 32 bits:
  - Opcode is 6 bits
  - Since we have 32 registers, each register “name” is 5 bits
  - This leaves 16 bits for the immediate constant

Opc  rs  rt  immed
I-type format example

\[
\text{addi} \quad $a0,$12,33 \quad \# $a0 \text{ is also } $4 = $12 +33
\]

\# Addi has opcode $8_{10}$ (001000)

\[
\begin{array}{cccc}
\text{Opc} & \text{rs} & \text{rt} & \text{immed} \\
8 & 12 & 4 & 33
\end{array}
\]

Bit position: 31 25 20 15 0

In hex: 0x21840021

(binary 001000 01100 00100 0000 0000 0010 0001)

CSE378 Instr. encoding.
Sign extension

• Internally the ALU (adder) deals with 32-bit numbers
• What happens to the 16-bit constant?
  – Extended to 32 bits
• If the Opcode says “unsigned” (e.g., addiu)
  – Fill upper 16 bits with 0’s
• If the Opcode says “signed” (e.g., addi)
  – Fill upper 16 bits with the msb of the 16 bit constant
  • i.e. fill with 0’s if the number is positive
  • i.e. fill with 1’s if the number is negative
R-type (register) format

- Arithmetic, Logical, and Compare instructions require encoding 3 registers.
- Opcode (6 bits) + 3 registers (5x3 = 15 bits) => 32 - 21 = 11 “free” bits
- Use 6 of these bits to expand the Opcode
- Use 5 for the “shift” amount in shift instructions
R-type example

```
sub $7,$8,$9
Opc =0, funct = 34
```

```
rs          rt            rd
0           8           9           7          0          34
```

Unused bits
Arithmetic instructions in SPIM

- Don’t confuse the SPIM format with the “encoding” of instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>rd,rs,rt</td>
<td>#rd = rs + rt</td>
</tr>
<tr>
<td>addi</td>
<td>rt,rs,immed</td>
<td>#rt = rs + immed</td>
</tr>
<tr>
<td>sub</td>
<td>rd,rs,rt</td>
<td>#rd = rs - rt</td>
</tr>
</tbody>
</table>
Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$8, $9, $10</td>
<td>#$8 = $9 + $10</td>
</tr>
<tr>
<td>add</td>
<td>$t0, $t1, $t2</td>
<td>#$t0 = $t1 + $t2</td>
</tr>
<tr>
<td>sub</td>
<td>$s2, $s1, $s0</td>
<td>#$s2 = $s1 - $s0</td>
</tr>
<tr>
<td>addi</td>
<td>$a0, $t0, 20</td>
<td>#$a0 = $t0 + 20</td>
</tr>
<tr>
<td>addi</td>
<td>$a0, $t0, -20</td>
<td>#$a0 = $t0 - 20</td>
</tr>
<tr>
<td>addi</td>
<td>$t0, $0, 0</td>
<td>#clear $t0</td>
</tr>
<tr>
<td>sub</td>
<td>$t5, $0, $t5</td>
<td>#$t5 = -$t5</td>
</tr>
</tbody>
</table>
Integer arithmetic

- Numbers can be *signed* or *unsigned*
- Arithmetic instructions (+,-,*,/) exist for both signed and unsigned numbers (differentiated by Opcode)
  - Example: add and addu
    - addi and addiu
    - mult and multu
- Signed numbers are represented in 2’s complement
- For add and subtract, computation is the same but
  - add, sub, addi cause *exceptions* in case of *overflow*
  - addu, subu, addiu don’t
How does the CPU know if the numbers are signed or unsigned?

- It does not!
- **You do** (or the compiler does)
- You have to tell the machine by using the right instruction (e.g. add or addu)
Some interesting instructions. Multiply

- Multiplying 2 32-bit numbers yields a 64-bit result
  - Use of HI and LO registers
    mult rs,rt #HI/LO = rs*rt
    multu rs,rt

Then need to move the HI or LO or both to regular registers

mflo rd #rd = LO
mfhi rd #rd = HI
Some interesting instructions. Divide

• Similarly, divide needs two registers
  – LO gets the quotient
  – HI gets the remainder

• If an operand is negative, the remainder is not specified by the MIPS ISA.
Logic instructions

- Used to manipulate bits within words, set-up masks etc.
- A sample of instructions
  
  ```plaintext
  and    rd,rs,rt               #rd=AND(rs,rt)
  andi   rd,rs,immed
  or     rd,rs,rt
  xor    rd,rs,rt
  ```

- Immediate constant limited to 16 bits (zero-extended). If longer mask needed, use lui (see in a few slides)

- There is a pseudo-instruction “not”
  
  ```plaintext
  not    rt,rs                  #does 1’s complement (bit by bit
                              #complement of rs in rt)
  ```
Pseudo-instructions

• Pseudo-instructions are there to help you
• The assembler recognizes them and generates real SPIM code
• Careful! One pseudo-instruction can generate several machine language instructions
  – mul rd,rs,rt is two instruction (mult followed by mflo)
  – not rs,rt is one instruction (which one?)
Loading small constants in a register

- If the constant is small (i.e., can be encoded in 16 bits) use the immediate format with li (Load immediate)
  \[
  \text{li} \quad 14,8 \quad \#14 = 8
  \]
- But, there is no opcode for li!
- li is a \textit{pseudoinstruction}
  - SPIM will recognize it and transform it into addi (with sign-extension) or ori (zero extended)
  \[
  \text{ori} \quad 14,0,8 \quad \#14 = 0+8
  \]
Loading large constants in a register

- If the constant does not fit in 16 bits (e.g., an address)
- Use a two-step process
  - lui (load upper immediate) to load the upper 16 bits; it will zero out automatically the lower 16 bits
  - Use ori for the lower 16 bits (but not li, why?)
- Example: Load the constant 0x1B234567 in register $t0
  lui $t0,0x1B23 \#note the use of hex constants
  ori $t0,$t0,0x4567
Example of use of logic instructions

- Create a *mask* of all 1’s for the low-order byte of $6$. Don’t care about the other bits.
  
  ```
  ori $6,$6,0x00ff  #$6[7:0] set to 1’s
  ```

- Clear high-order byte of register 7 but leave the 3 other bytes unchanged
  
  ```
  lui $5,0x00ff  #$5 = 0x00ff0000
  ori $5,$5,0xffff  #$5 = 0x00ffffff
  and $7,$7,$5  #$7 =0x00…… (…whatever was there before)
  ```
Shift instructions

• Logical shifts -- Zeroes are inserted
  sll rd,rt,shm  #left shift of shm bits; inserting 0’s on
                  #the right
  srl rd,rt,shm  #right shift of shm bits; inserting 0’s
                  #on the left

• Arithmetic shifts (useful only on the right)
  – sra rd,rt,shm  # Sign bit is inserted on the left

• Example let $5 = 0xff00 0000$
  sll $6,$5,3  #=$6 = 0xf800 0000
  srl $6,$5,3  #=$6 = 0x1fe0 0000
  sra $6,$5,3  #=$6 = 0xffe0 0000
Load and Store instructions

- MIPS = RISC = Load-Store architecture
  - Load: brings data from memory to a register
  - Store: brings data back to memory from a register

- Each load-store instruction must specify
  - The unit of info to be transferred (byte, word etc.) through the Opcode
  - The address in memory

- A memory address is a 32-bit byte address
- An instruction has only 32 bits so ....
Addressing in Load/Store instructions

• The address will be the sum
  – of a base register (register rs)
  – a 16-bit offset (or displacement) which will be in the immed field and is added (as a signed number) to the contents of the base register

• Thus, one can address any byte within ± 32KB of the address pointed to by the contents of the base register.
Examples of load-store instructions

- **Load word from memory:**
  \[
  \text{lw} \quad rt, rs, offset \quad \#rt = \text{Memory}[rs+\text{offset}]
  \]

- **Store word to memory:**
  \[
  \text{sw} \quad rt, rs, offset \quad \#\text{Memory}[rs+\text{offset}] = rt
  \]

- **For bytes (or half-words) only the lower byte (or half-word) of a register is addressable**
  - For load you need to specify if it is sign-extended or not
    \[
    \begin{align*}
    \text{lb} & \quad rt, rs, offset \quad \#rt = \text{sign-ext}( \text{Memory}[rs+\text{offset}] ) \\
    \text{lbu} & \quad rt, rs, offset \quad \#rt = \text{zero-ext}( \text{Memory}[rs+\text{offset}] ) \\
    \text{sb} & \quad rt, rs, offset \quad \#\text{Memory}[rs+\text{offset}] = \text{least signif. byte of rt}
    \end{align*}
    \]
Load-Store format

• Need for
  – Opcode (6 bits)
  – Register destination (for Load) and source (for Store) : rt
  – Base register: rs
  – Offset (immed field)

• Example

```
lw $14,8($sp)  #$14 loaded from top of stack + 8
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>29</td>
<td>14</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>
How to address memory in assembly language

• Problem: how do I put the base address in the right register and how do I compute the offset

• Method 1 (most recommended). Let the assembler do it!

```
.data             #define data section
xyz:             .word 1             #reserve room for 1 word at address xyz
........          #more data
.text             #define program section
.....             # some lines of code
lw $5, xyz        # load contents of word at add. xyz in $5
```

• In fact the assembler generates:

```
lw $5, offset ($gp)     #$gp is register 28
```
Generating addresses

• Method 2. Use the pseudo-instruction “la” (Load address)

  la $6,xyz       #$6 contains address of xyz
  lw $5,0($6)     #$5 contains the contents of xyz

  – la is in fact lui followed by ori
  – This method can be useful to traverse an array after loading the base address in a register

• Method 3

  – If you know the address (i.e. a constant) use li or lui + ori