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
- 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 and rs always at the same place
    - immed always at the same place etc.
I-type (Immediate) Instruction Format

- An instruction with the immediate format 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
  - Each register “name” is 5 bits since there are 32 registers
  - That leaves 16 bits for the immediate constant

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>
I-type Instruction Example

Addi $a0,$12,33  # $a0 is also $4 = $12 +33
# Addi has opcode 08

In binary:
0010 0001 1000 0100 0000 0000 0010 0001

In hex: 21840021
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

<table>
<thead>
<tr>
<th>Opc</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shft</th>
<th>func</th>
</tr>
</thead>
</table>

10/05/2005 CSE378 Instr. encoding. 6
R-type (Register) Instruction 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

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shft</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
R-type example

Sub

Opc = 0 & funct = 34

$7, 8, 9$

rs         rt         rd

0          8           9           7          0          34

Unused bits
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:
  \[ \text{LW} \ rt, rs, \text{offset} \quad \#rt = \text{Memory}[rs+\text{offset}] \]

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

- For bytes (or half-words) only the lower byte (or half-word) of a register is addressable
  - For load you need to specify if data is sign-extended or not
    \[ \text{LB} \ rt, rs, \text{offset} \quad \#rt = \text{sign-ext( Memory}[rs+\text{offset}]) \]
    \[ \text{LBU} \ rt, rs, \text{offset} \quad \#rt = \text{zero-ext( Memory}[rs+\text{offset}]) \]
    \[ \text{SB} \ rt, rs, \text{offset} \quad \#\text{Memory}[rs+\text{offset}]= \text{least signif. byte of rt} \]
Load-Store format

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

• Example

\[ \text{LW } \$14,8($sp) \#14 \text{ loaded from top stack + 8} \]
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}
  
  – The assembler creates it to help you
  – SPIM will recognize it and transform it into Addi (with sign-extension) or Ori (zero extended)

  \[
  \text{Addi} \quad \$14, \$0, 8 \quad \#\$14 = \$0+8
  \]
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 constant 0x1B234567 in register $t0
  
  LUI $t0,0x1B23  #note the use of hex constants
  ORI $t0,$t0,0x4567
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 (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
  ```
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
Flow of Control -- Conditional branch instructions

• You can compare directly
  – Equality or inequality of two registers
  – One register with 0 (> , < , ≥ , ≤)

• and branch to a target specified as
  – a signed displacement expressed in number of instructions
    (not number of bytes) from the instruction following the branch
  – in assembly language, it is highly recommended to use labels and branch to labeled target addresses because:
    • the computation above is too complicated
    • some pseudo-instructions are translated into two real instructions
Examples of branch instructions

Beq    rs,rt,target         #go to target if rs = rt
Beqz   rs, target           #go to target if rs = 0
Bne    rs,rt,target         #go to target if rs != rt
Bltz   rs, target           #go to target if rs < 0

etc.

but note that you cannot compare directly 2 registers for <, >
...


Comparisons between two registers

• Use an instruction to set a third register
  \[ \text{slt } \quad \text{rd,rs,rt} \quad \# \text{rd} = 1 \text{ if rs} < \text{rt} \text{ else rd} = 0 \]
  \[ \text{sltu } \quad \text{rd,rs,rt} \quad \# \text{same but rs and rt are considered unsigned} \]

• Example: Branch to Lab1 if $5 < $6
  \[ \text{slt } \quad \$10,$5,$6 \quad \# \text{$10} = 1 \text{ if $5} < \text{$6} \text{ otherwise $10} = 0 \]
  \[ \text{bnez } \quad \$10,\text{Lab1} \quad \# \text{branch if $10 =1, i.e., $5<$6} \]

• There exist pseudo instructions to help you!
  \[ \text{blt } \quad \$5,$6,\text{Lab1} \quad \# \text{pseudo instruction translated into} \]
  \[ \text{# slt } \quad \$1,$5,$6 \]
  \[ \text{# bne } \quad \$1,$0,\text{Lab1} \]
  Note the use of register 1 by the assembler and the fact that computing the address of Lab1 requires knowledge of how pseudo-instructions are expanded
Unconditional transfer of control

• Can use “beqz $0, target”
  – Very useful but limited range (± 32K instructions)
• Use of Jump instructions
  j  target    #special format for target byte address (26 bits)
  jr $rs      #jump to address stored in rs (good for switch statements and transfer tables)
• Call/return functions and procedures
  jal  target  #jump to target address; save PC of following instruction in $31 (aka $ra)
  jr $31       # jump to address stored in $31 (or $ra)
  jalr  rs,rd  #jump to address stored in rs; rd = PC of following instruction in rd with default rd = $31