Introduction

- Remember that in a stored program computer, instructions are stored in memory (just like data)
- Each instruction is fetched (according to the address specified in the PC), decoded, and executed by the CPU
- The ISA defines the format of an instruction (syntax) and its meaning (semantics)
- An ISA will define a number of different instruction formats.
- Each format has different fields
- The OPCODE field says what the instruction does (e.g. ADD)
- The OPERAND field(s) say where to find inputs and outputs of the instruction.

MIPS Encoding

- The nice thing about MIPS (and other RISC machines) is that it has very few instruction formats (basically just 3)
- All instructions are the same size (32 bits = 1 word)
- The formats are consistent with each other (i.e. the OPCODE field is always in the same place, etc.)
- The three formats:
  - I-type (immediate)
  - R-type (register)
  - J-type (jump)
I-type (immediate) Format

- An immediate instruction has the form:
  \[ \text{XXXI \ rt, rs, immed} \]
- Recall that we have 32 registers, so we need \( ?? \) bits each to specify the \( \text{rt} \) and \( \text{rs} \) registers.
- We allow 6 bits for the opcode (this implies a maximum of \( ?? \) opcodes, but there are actually more, see later).
- This leaves 16 bits for the immediate field.

\[
\begin{array}{cccc}
31 & 25 & 20 & 15 \\
\text{OPC} & \text{rs} & \text{rt} & \text{immed} \\
26 & 21 & 16 & 0
\end{array}
\]

I-type Example

- Example:
  \[ \text{ADDI \ $a0, \ $12, \ 33} \] # \( \text{a0} \) <- \( r12 + 33 \)
- The ADDI opcode is 8, register \( a0 \) is register \# 4.

\[
\begin{array}{ccccccc}
31 & 25 & 20 & 15 & 8 & 12 & 4 & 33 \\
26 & 21 & 16 & 0
\end{array}
\]

- What would this be in binary? In hex?

Load-Store Formats

- A memory address is 32 bits, so it cannot be directly encoded in an instruction.
- Recall the use of a base register + offset (16-bits) in the load-store instructions.
- Thus, we need an \( \text{OPCODE} \), a destination/source register (destination for load, source for store), a base register, and an offset.
- This sounds very similar to the I-type format... example:
  \[ \text{LW \ $14, \ 8($sp)} \] # \( r14 \) is loaded from stack+8
- The LW opcode is 35 (0x23)

\[
\begin{array}{cccc}
31 & 25 & 20 & 15 \\
35 & 29 & 14 & 8 \\
26 & 21 & 16 & 0
\end{array}
\]

R-type (register) format

- General form:
  \[ \text{XXX \ rd, rt, rs} \]
- Arithmetic-logical and comparison instructions require the encoding of 3 registers, the rest can be used to specify the \( \text{OPCODE} \).
- To keep the format as regular as possible, the \( \text{OPCODE} \) has a primary “opcode” and a “function” field.
- We also need 5 bits for the shift-amount, in case of SHIFT instructions.
- The 16 bits used for the immediate field in the I-type instruction are split into 5 bits for \( \text{rd} \), 5 bits for shift-amount, and 6 bits for function (the other fields are the same).

\[
\begin{array}{ccccccc}
31 & 25 & 20 & 15 & 10 & 5 \\
\text{OPC} & \text{rs} & \text{rt} & \text{rd} & \text{sht} & \text{funct} \\
26 & 21 & 16 & 11 & 6 & 0
\end{array}
\]
R-type Example

SUB $7, $8, $9  # r7 <- r8 - r9

- The opcode for all R-type instructions is zero, the function code for
  SUB is 34, the shift amount is zero.

- What is this in binary/hex?

J-type (Jump) Format

- For a jump, we only need to specify the opcode, and we can use
  the other bits for an address:

Branch Addressing

- There are 2 kinds of branches:
  1. EQ/NEQ family (compares 2 regs for (in)equality), example:
     BEQ $14, $8, 1000
  2. Compare-to-zero family (compares 1 reg to zero), example:
     BGEZ $14, 1000

- Both “families” require OPCODE, rs register, and offset

- (1.) requires an additional register (rt)
- (2.) requires some encoding for (>=, <=, <, >)

- Note that we divide the offset by 4. Why?

Branch example

BEQ $14, $8, 1000  # PC := PC+1000 if r14==r8

BGEZ $14, 20  # PC := PC+20 if r14 >= 0

- The opcode for BEQ is 4; for BGEZ is 1, the code for >= is 1

- We only have 26 bits for the address, but MIPS addresses are 32
  bits long...

- Because the address must reference an instruction, which is a
  word address, we can shift the address left by 2 bits (giving us 28
  bits). We get the other 4 bits by combining with the 4 high-order
  bits of the PC.
Assembly Language Version

- Recall our running example:

```assembly
.data
    # begin data segment
array: .space 400  # allocate 400 bytes

.text
    # begin code segment
.globl main  # entry point must be global

main:  move$t0, $0  # $t0 is used as counter
       la $t1, array  # $t1 is pointer into array
start: bge $t0, 100, exit  # more than 99 iterations?
       sw $t0, 0($t1)  # store zero into array
       addi $t0, $t0, 1  # increment counter
       addi $t1, $t1, 4  # increment pointer into array
       j start  # goto top of loop
exit:   j $ra  # return to caller of main...
```

Machine Language Version

<table>
<thead>
<tr>
<th>Encoded:</th>
<th>Machine Ins:</th>
<th>Source Ins:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00004021</td>
<td>addu $8, $0, $0</td>
<td>9: move$t0, $0</td>
</tr>
<tr>
<td>0x3c091001</td>
<td>lui $9, 4097</td>
<td>10: la$t1, array</td>
</tr>
<tr>
<td>0x29010064</td>
<td>slti $1, $8, 100</td>
<td>11: bge$t0, 100, exit</td>
</tr>
<tr>
<td>0x10200005</td>
<td>beq $1, $0, 20</td>
<td></td>
</tr>
<tr>
<td>0xad280000</td>
<td>sw $8, 0($9)</td>
<td>12: sw$t0, 0($t1)</td>
</tr>
<tr>
<td>0x21080001</td>
<td>addi $8, $8, 1</td>
<td>13: addi$t0, $t0, 1</td>
</tr>
<tr>
<td>0x21290004</td>
<td>addi $9, $9, 4</td>
<td>14: addi$t1, $t1, 4</td>
</tr>
<tr>
<td>0x0810000b</td>
<td>j 0x0040002c</td>
<td>15: jstart</td>
</tr>
<tr>
<td>0x03e00008</td>
<td>jr $31</td>
<td>16: j$ra</td>
</tr>
</tbody>
</table>