MIPS History

- MIPS is a computer family
  - R2000/R3000 (32-bit); R4000/R4400 (64-bit); R10000 (64-bit) etc.
- MIPS originated as a Stanford research project under the direction of John Hennessy
  - Microprocessor without Interlocked Pipe Stages
- MIPS Co. bought by SGI
- MIPS used in previous generations of DEC (then Compaq, now HP) workstations
- Now MIPS Technologies is in the embedded systems market
- MIPS is a RISC

ISA MIPS Registers

- Thirty-two 32-bit registers $0, 1, …, 31$ used for
  - integer arithmetic, address calculation, temporaries, special-purpose functions (stack pointer etc.)
- A 32-bit Program Counter (PC)
- Two 32-bit registers (HI, LO) used for multi- and division
- Thirty-two 32-bit registers $80, 81, …, 83$ used for floating-point arithmetic
  - Often used in pairs: 16 64-bit registers
- Registers are a major part of the “state” of a process

MIPS Register names and conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>$Z$</td>
<td>Always 0</td>
<td>No-op on write</td>
</tr>
<tr>
<td>$1$</td>
<td>$A$</td>
<td>Reserved for assembler</td>
<td>Don’t use it</td>
</tr>
<tr>
<td>$2$ - $3$</td>
<td>$v_0$, $v_1$</td>
<td>Expression/Function Return</td>
<td></td>
</tr>
<tr>
<td>$4$ - $7$</td>
<td>$a_0$, $a_3$</td>
<td>Procedure/function Call Parameters</td>
<td></td>
</tr>
<tr>
<td>$8$ - $15$</td>
<td>$t_0$, $t_7$</td>
<td>Temporaries; volatile</td>
<td>Not saved on proc. calls</td>
</tr>
<tr>
<td>$16$ - $23$</td>
<td>$s_0$, $s_7$</td>
<td>Temporaries</td>
<td>Should be saved on calls</td>
</tr>
<tr>
<td>$24$ - $25$</td>
<td>$t_8$, $t_9$</td>
<td>Temporaries; volatile</td>
<td>Not saved on proc. calls</td>
</tr>
<tr>
<td>$26$ - $27$</td>
<td>$k_0$, $k_1$</td>
<td>Reserved for O.S.</td>
<td>Don’t use them</td>
</tr>
<tr>
<td>$28$</td>
<td>$gp$</td>
<td>Pointer to global static memory</td>
<td></td>
</tr>
<tr>
<td>$29$</td>
<td>$sp$</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>$30$</td>
<td>$fp$</td>
<td>Frame pointer</td>
<td></td>
</tr>
<tr>
<td>$31$</td>
<td>$ra$</td>
<td>Procedure/function return address</td>
<td></td>
</tr>
</tbody>
</table>

MIPS = RISC = Load-Store architecture

- Every operand must be in a register
  - Except for some small integer constants that can be in the instruction itself (see later)
- Variables have to be loaded in registers
- Results have to be stored in memory
- Explicit Load and Store instructions are needed because there are many more variables than the number of registers

Example

- The HLL statements
  
  \[
  \begin{align*}
  a &= b + c \\
  d &= a + b
  \end{align*}
  \]

  will be “translated” into assembly language as:

  ```
  load $b$ in register $rx$
  load $c$ in register $ry$
  $rz < - rx + ry$
  store $rz$ in $a$
  $rx < - tz + rx$
  store $rz$ in $d$
  ```

  # not destructive; $rz$ still contains the value of $a$

MIPS Information units

- Data types and size:
  - Byte
  - Half-word (2 bytes)
  - Word (4 bytes)
  - Float (4 bytes; single-precision format)
  - Double (8 bytes; double-precision format)
- Memory is byte-addressable
- A data type must start at an address evenly divisible by its size (in bytes)
- In the little-endian environment, the address of a data type is the address of its lowest byte
Addressing of Information units

Byte address 2
Half-word address 2
Byte address 5

SPIM Convention

Words listed from left to right but little endianness within words

Assembly Language programming or How to be nice to your TAs

- Use lots of detailed comments
- Don’t be too fancy
- Use lots of detailed comments
- Use words (rather than bytes) whenever possible
- Use lots of detailed comments
- Remember: The address of a word is evenly divisible by 4
- Use lots of detailed comments
- The word following the word at address i is at address i+4
- Use lots of detailed comments

MIPS Instruction types

- Few of them (RISC philosophy)
- Arithmetic
- Logical and Shift
- Load and Store
- Compare (of values in registers)
- Branch and jumps (flow of control)
- Includes procedure/function calls and returns

Notation for SPIM instructions

- Opcode rd, rs, rt
- Opcode rt, rs, imm
- where
  - rd is always a destination register (result)
  - rs is always a source register (read-only)
  - rt can be either a source or a destination (depends on the opcode)
  - imm is a 16-bit constant (signed or unsigned)

Arithmetic instructions in SPIM

- Don’t confuse the SPIM format with the “encoding” of instructions that we’ll see soon:
  Opcode Operands Comments
  Add rd, rs, rt \#rd = rs + rt
  Addi rt, rs, imm \#rt = rs + imm
  Sub rd, rs, rt \#rd = rs - rt
### Integer arithmetic

- Numbers can be **signed** or **unsigned**
- Arithmetic instructions (+, -, *, /) exist for both signed and unsigned numbers (differentiated by Opcode)
  - Example: Add and Addu
    - Add
    - Addu
  - Addi and Addiu
    - Addi
    - Addiu
  - Mul and Muli
  - 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)