MIPS History

- MIPS is a computer family
  - R2000/R3000 (32-bit); R4000/4400 (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 mult. and division
- Thirty-two 32-bit registers $10,$11,...,$31 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>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>Z0</td>
<td>Always 0</td>
</tr>
<tr>
<td>$1</td>
<td>A0</td>
<td>Don’t use it</td>
</tr>
<tr>
<td>$2-$3</td>
<td>V0-V1</td>
<td>Expression evaluation</td>
</tr>
<tr>
<td>$4-$7</td>
<td>A0-A3</td>
<td>Proc./func. Call parameters</td>
</tr>
<tr>
<td>$8-$15</td>
<td>T0-T7</td>
<td>Temporaries; volatile</td>
</tr>
<tr>
<td>$16-$23</td>
<td>S0-S7</td>
<td>Temporaries; should be saved on calls</td>
</tr>
<tr>
<td>$24-$25</td>
<td>T8-T9</td>
<td>Temporaries; volatile</td>
</tr>
<tr>
<td>$26-$27</td>
<td>K0-K1</td>
<td>Reserved for O.S. Don’t use them</td>
</tr>
<tr>
<td>$28</td>
<td>GP</td>
<td>Pointer to global static memory</td>
</tr>
<tr>
<td>$29</td>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>$30</td>
<td>FP</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$31</td>
<td>RA</td>
<td>Proc./func. return address</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
  
  \[
  a = b + c \\
  d = a + b
  \]

  will be “translated” into assembly language as:

  ```
  load b in register rx
  load c in register ry
  rz <- rx + ry
  store rz in a
  # not destructive; rz still contains the value of a
  rt <- rz + rx
  store rt in d
  ```

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

Half-word address 5

Bye address 0

Half-word address 0

Word address 0

Bye address 8

Half-word address 8

Word address 8

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 word’s address 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
  - Integer (signed and unsigned); Floating-point
- Logical and Shift
  - work on bit strings
- Load and Store
  - for various data types (bytes, words,…)
- 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
### Examples

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>$8, $9, $10</td>
<td>$9 + $10</td>
</tr>
<tr>
<td>Add</td>
<td>$10, $1, $2</td>
<td>$11 + $2</td>
</tr>
<tr>
<td>Sub</td>
<td>$4, $5, $0</td>
<td>$4 - $5</td>
</tr>
<tr>
<td>Addi</td>
<td>$a0, $10, $20</td>
<td>$a0 + $20</td>
</tr>
<tr>
<td>Addi</td>
<td>$a0, $10, $-20</td>
<td>$a0 - $20</td>
</tr>
<tr>
<td>Addi</td>
<td>$10, $0, $0</td>
<td>$10</td>
</tr>
<tr>
<td>Sub</td>
<td>$15, $0, $15</td>
<td>$15 - $15</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 Mulu
- 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)