MIPS

MIPS is a “computer family”

- R2000/R3000 (32-bit)
- R4000/4400 (64-bit)
- R8000 (for scientific & graphics applications)
- R10000 (64-bit)

MIPS originated as a Stanford research project
Microprocessor without Interlocked Pipe Stages

MIPS was bought by Silicon Graphics (SGI) & is now independent

MIPS is a RISC
MIPS Registers

Part of the state of a process

Thirty-two 32-bit general purpose registers (GPRs): $0, $1, ..., $31
  • integer arithmetic
  • address calculations
  • temporary values

By convention software uses different registers for different purposes (next slide)

A 32-bit program counter (PC)

Two 32-bit registers HI and LO used specifically for multiply and divide
  • HI & LO concatenated for the product
  • LO for the quotient; HI for the remainder

Thirty-two 32-bit registers: $f0, $f1, ..., $f31 used for floating-point arithmetic
  • often used as 16 64-bit registers for double precision FP

Other special-purpose registers (later)
## MIPS Register Names and gcc Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Use</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>zero</td>
<td>always 0</td>
<td>cannot be written</td>
</tr>
<tr>
<td>$1</td>
<td>$at</td>
<td>reserved for assembler</td>
<td>don’t use it!</td>
</tr>
<tr>
<td>$2, $3</td>
<td>$v0, $v1</td>
<td>function return</td>
<td></td>
</tr>
<tr>
<td>$4 - $7</td>
<td>$a0 - $a3</td>
<td>pass first 4 procedure/</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>function arguments</td>
<td></td>
</tr>
<tr>
<td>$8 - $15</td>
<td>$t0 - $t7</td>
<td>temporaries</td>
<td>caller saved (callee uses them without saving them)</td>
</tr>
<tr>
<td>$16 - $23</td>
<td>$s0 - $s7</td>
<td>temporaries</td>
<td>callee saved (caller assumes they will be available on function return)</td>
</tr>
<tr>
<td>$24, $25</td>
<td>$t8, $t9</td>
<td>temporaries</td>
<td>caller saved</td>
</tr>
<tr>
<td>$26, $27</td>
<td>$k0, $k1</td>
<td>reserved for the OS</td>
<td>don’t use them!</td>
</tr>
<tr>
<td>$28</td>
<td>$gp</td>
<td>pointer to global static memory</td>
<td>points to the middle of a 64KB block in static data (next slide)</td>
</tr>
<tr>
<td>$29</td>
<td>$sp</td>
<td>stack pointer</td>
<td>points to the last allocated stack location (next slide)</td>
</tr>
<tr>
<td>$30</td>
<td>$fp</td>
<td>frame pointer</td>
<td>points to the activation record (later)</td>
</tr>
<tr>
<td>$31</td>
<td>$ra</td>
<td>procedure/function return address</td>
<td></td>
</tr>
</tbody>
</table>
Memory Usage

A software convention

```
+-----------------+----------+
|      stack      | 0x7fffffff |
|                  |
|  dynamic        | 0x10000000 |
|  data           |           |
|  static         |           |
|  data           |           |
|  Code           | 0x04000000 |
|  Reserved       | 0x00000000 |
```

text segment: the code
data segment
- **static data**: objects whose size is known to the compiler & whose lifetime is the whole program execution
- **dynamic data**: objects allocated as the program executes (`malloc`)

stack segment: FIFO process-local storage
MIPS Load-Store Architecture

Most instructions compute on operands stored in registers
- load data into a register from memory
- compute in registers
- the result is stored into memory

For example,
\[
\begin{align*}
a &= b + c \\
d &= a + b
\end{align*}
\]

is “compiled” into:
- load b into register $x$
- load c into register $y$
- $z \leftarrow x + y$
- store $z$ into a
- $z \leftarrow z + x$
- store $z$ into d
MIPS Information Units

Data types and sizes

- byte
- half-word (2 bytes)
- word (4 bytes)
- float (4 bytes using single-precision floating-point format)
- double (8 bytes using double-precision floating-point format)

Memory is byte-addressable

A data type must start on an address evenly divisible by its size in bytes
Big & Little Endian

Every word starts at an address that is divisible by 4.
Which byte in the word is byte 0?
How is the data in \texttt{.byte 0,1,2,3} stored?

Big-endian: \texttt{0x10f14201} is stored in memory as:

- 0: \texttt{0x10}
- 1: \texttt{0xf1}
- 2: \texttt{0x42}
- 3: \texttt{0x01}

Can be read as: \texttt{0x10*2^{24} + 0xf1*2^{16} + 0x42*2^{8} + 0x01*2^{0}}

\textbf{Most} significant byte is the lowest byte address.
Word is addressed by the byte address of the \textbf{most} significant byte.

Little-endian: \texttt{0x10f14201} is stored in memory as:

- 0: \texttt{0x01}
- 1: \texttt{0x42}
- 2: \texttt{0xf1}
- 3: \texttt{0x10}

Can be read as: \texttt{0x01*2^{0} + 0x42*2^{8} + 0xf1*2^{16} + 0x10*2^{24}}

\textbf{Least} significant byte is the lowest byte address.
Word is addressed by the byte address of the \textbf{least} significant byte.
Moving between Big & Little Endian

Big to Little or Little to Big:

Start with item size and reverse each 1/2
Go to each 1/2 and recurse

For example:

0: 0x10
1: 0xf1
2: 0x42
3: 0x01

Reverse 1/2

0: 0x42
1: 0x01
2: 0x10
3: 0xf1

Recursivelly Reverse 1/2

0: 0x01
1: 0x42
2: 0xf1
3: 0x10
Can detect in C/C++ the native endian of the machine:

```c
if ( ((short)"AB") == 5680) return (big); else return(little);
```

Why does this work?
MIPS Information Units

MIPS support both big- and little-endian byte orders

SPIM uses the byte order of the machine its running on
• Intel: little-endian
• Alpha, SPARC, Mac: big-endian

Words in SPIM are listed from left to right, but byte addresses are little-endian within a word

```
[0x7fffebd0] 0x00400018 0x00000001 0x00000005 0x00010aff
```

0x7fffebd2 0x7fffebd4 0x7fffebde