Instruction Set Architecture of MIPS Processor

MIPS Registers

- **CPU:**
  - 32 32-bit general purpose registers – GPRs (r0 – r31);
  - r0 has fixed value of zero. Attempt to writing into r0 is not illegal, but its value will not change;
  - two 32-bit registers – Hi & Lo, hold results of integer multiply and divide
  - 32-bit program counter – PC;
- **Floating Point Processor – FPU (Coprocessor 1 – CP1):**
  - 32 32-bit floating point registers – FPRs (f0 – f31);
  - five control registers;

MIPS Registers (continued)

- **Coprocessor 0 – CP0** is incorporated on the MIPS CPU chip and it provides functions necessary to support operating system: exception handling, memory management scheduling and control of critical resources.
- **Coprocessor 0 (CP0) registers (partial list):**
  - Status register (CP0reg12) – processor status and control;
  - Cause register (CP0reg13) – cause of the most recent exception;
  - EPC register (CP0reg14) – program counter at the last exception;
  - BadVAddr register (CP0reg08) – the address for the most recent address related exception;
MIPS Data Types

- MIPS operates on:
  - 32-bit (unsigned or 2’s complement) integers,
  - 32-bit (single precision floating point) real numbers,
  - 64-bit (double precision floating point) real numbers;

- bytes and half words loaded into GPRs are either zero or sign bit expanded to fill the 32 bits;

- only 32-bit units can be loaded into FPRs; 32-bit real numbers are stored in even numbered FPRs.

- 64-bit real numbers are stored in two consecutive FPRs, starting with even-numbered register.

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \( a = b + c \)

MIPS ‘code’: `add a, b, c`

“The natural number of operands for an operation like addition is three...requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple”

MIPS arithmetic

- Design Principle: simplicity favors regularity.
- Of course this complicates some things...

C code: \( a = b + c + d \)

MIPS code: `add a, b, c
add a, a, d`

- Operands must be registers, only 32 registers provided
- Each register contains 32 bits
- Design Principle: smaller is faster. Why?

Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables

```
Control
Datapath
Memory
Input
Output
I/O
```
Memory Organization

• Viewed as a large, single-dimension array, with an address.
• A memory address is an index into the array.
• "Byte addressing" means that the index points to a byte of memory.

<table>
<thead>
<tr>
<th>0</th>
<th>8 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>2</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>3</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>5</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>6</td>
<td>8 bits of data</td>
</tr>
</tbody>
</table>

... 8 bits of data

MIPS Addressing Modes

• register addressing;
• immediate addressing;
• only one memory data addressing:
  – register content plus offset (register indexed);
  – since r0 always contains value 0:
  – r0 + offset \(\rightarrow\) absolute addressing;
• offset = 0 \(\rightarrow\) register indirect;
• MIPS supports byte addressability:
  – it means that a byte is the smallest unit with its address;
• MIPS supports 32-bit addresses:
  – it means that an address is given as 32-bit unsigned integer;

Memory Organization

• Bytes are nice, but most data items use larger "words".
• For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>0</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>12</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>

... 32 bits of data

• \(2^{32}\) bytes with byte addresses from 0 to \(2^{32}-1\)
• \(2^{30}\) words with byte addresses 0, 4, 8, ... \(2^{32}-4\)
• Words are aligned
  i.e., what are the least 2 significant bits of a word address?

MIPS Alignment

• MIPS restricts memory accesses to be aligned as follows:
  – 32-bit word has to start at byte address that is multiple of 4;
  32-bit word at address \(4n\) includes four bytes with addresses \(4n, 4n+1, 4n+2, \) and \(4n+3\).
  – 16-bit half word has to start at byte address that is multiple of 2;
  16-bit word at address \(2n\) includes two bytes with addresses \(2n\) and \(2n+1\).
MIPS Instructions

- 32-bit fixed format instruction and 3 formats;
- register – register and register-immediate computational instructions;
- single address mode for load/store instructions:
  - register content + offset (called base addressing);
- simple branch conditions:
  - branch instructions use PC relative addressing;
  - branch address = [PC] + 4 + 4×offset
- jump instructions with:
  - 28-bit addresses (jumps inside 256 megabyte regions), or
  - absolute 32-bit addresses.

Our First Example

- Can we figure out the code? Guess!!!

```c
swap(int v[], int k);
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```assembly
muli $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```

MIPS Instruction (continued)

- Instructions that move data:
  - load to register from memory,
  - store from register to memory,
  - move between registers in same and different coprocessors
- ALU integer instructions,
- Floating point instructions,
- Control-related instructions,
- Special control-related instructions.

Stored Program Concept

- Instructions are bits
- Programs are stored in memory — to be read or written just like data

memory for data, programs, compilers, editors, etc.

- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the "next" instruction and continue
MIPS Instruction Layout

- MIPS instructions are an example of fixed field decoding

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-type instruction</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>R-type instruction</td>
<td>5</td>
<td>rt</td>
<td>rd</td>
<td></td>
</tr>
<tr>
<td>J-type instruction</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU Load & Store Instructions

- \( lw \ rt, \text{offset}(\text{rs}) \); load 32-bits:
  \[ \text{Regs}[rt] \leftarrow \text{Mem}[\text{offset}+\text{Regs}[\text{rs}]] \]
- \( sw \ rt, \text{offset}(\text{rs}) \); store 32-bits:
  \[ \text{Mem}[\text{offset}+\text{Regs}[\text{rs}]] \leftarrow \text{Regs}[rt] \]
- \( lb \ rt, \text{offset}(\text{rs}) \); load 8-bits (byte):
  \[ \text{Regs}[rt] \leftarrow 24\text{-bit sign-extend} || \text{Mem}[\text{offset}+\text{Regs}[\text{rs}]] \]

Note: || means concatenation.

- \( lbu \ rt, \text{offset}(\text{rs}) \); load 8-bits (byte) unsigned:
  \[ \text{Regs}[rt] \leftarrow 24\text{-bit zero-extend} || \text{Mem}[\text{offset}+\text{Regs}[\text{rs}]] \]

- \( sh \ rt, \text{offset}(\text{rs}) \); store 16-bits:
  \[ \text{Mem}[\text{offset}+\text{Regs}[\text{rs}]] \leftarrow \text{Regs}[rt] \]
  \[ (16 \text{ least significant bits taken from the register}) \]
  Plus: \( sb, \text{lh, lihu} \)

FP Load, Store & Move Instructions

- \( lwc1 \ ft, \text{offset}(\text{rs}) \); load into FP register:
  \[ \text{Regs}[ft] \leftarrow \text{Mem}[\text{offset}+\text{Regs}[\text{rs}]] \]
- \( swc1 \ ft, \text{offset}(\text{rs}) \); store from FP register:
  \[ \text{Mem}[\text{offset}+\text{Regs}[\text{rs}]] \leftarrow \text{Regs}[ft] \]
- \( \text{mov.d} \ fd, fs \); move FP double precision between FPRs:
  \[ \text{Regs}[fd] || \text{Regs}[fd+1] \leftarrow \text{Regs}[fs] || \text{Regs}[fs+1] \]
- \( \text{mov.s} \ fd, fs \); move FP single precision between FPRs:
  \[ \text{Regs}[fd] \leftarrow \text{Regs}[fs] \]

Move Instructions

- \( mfc1 \ rt, fs \); move from FPU to CPU:
  \[ \text{Regs}[rt] \leftarrow \text{Regs}[fs] \]
- \( mtc1 \ rt, fs \); move from CPU to FPU:
  \[ \text{Regs}[fs] \leftarrow \text{Regs}[rt] \]
- \( mfc0 \ rt, rd \); move from CP0 to CPU:
  \[ \text{Regs}[rt] \leftarrow \text{CP0Regs}[rd] \]
- \( mtc0 \ rt, rd \); move from CPU to CP0:
  \[ \text{CP0Regs}[rd] \leftarrow \text{Regs}[rt] \]
- \( mfhi \ rd \); move from Hi:
  \[ \text{Regs}[rd] \leftarrow \text{Hi} \]
- \( mflo \ rd \); move from Lo:
  \[ \text{Regs}[rd] \leftarrow \text{Lo} \]
### ALU Integer Instructions

- **add rd, rs, rt**; *add integer*\(^\star\): $\text{Regs}[rd] \leftarrow \text{Regs}[rs] + \text{Regs}[rt]$  
  Note: Instructions flagged by \(^\star\) may cause an arithmetic exception.

- **addi rt, rs, immediate**; *add immediate integer*\(^\star\):  
  $\text{Regs}[rt] \leftarrow \text{Regs}[rs] + 16\text{-bit sign-extend} \ || \ immediate$

- **and rd, rs, rt**; *bit-wise AND 32 bits*:  
  $\text{Regs}[rd] \leftarrow \text{Regs}[rs] \ \text{AND} \ \text{Regs}[rt]$

- **andi rt, rs, immediate**; *bit-wise AND immediate 32 bits*:  
  $\text{Regs}[rt] \leftarrow \text{Regs}[rs] \ \text{AND} \ 16\text{-bit zero-extend} \ || \ immediate$

- **slt rd, rs, rt**; *set less than integer*:  
  if $(\text{Regs}[rs] < \text{Regs}[rt])$ then $\text{Regs}[rd] \leftarrow 1$ else $\text{Regs}[rd] \leftarrow 0$

### ALU Integer Instructions (continued)

- **mul rs, rt**; *multiply integer*: $\text{Hi} \ || \ Lo \leftarrow \text{Regs}[rs] \times \text{Regs}[rt]$  
  Note: *mul* does not generate an arithmetic exception, since a storage for the result is always sufficiently large.

- **div rs, rt**; *divide integer*:  
  $\text{Lo} \leftarrow \text{Regs}[rs] / \text{Regs}[rt]$  
  $\text{Hi} \leftarrow \text{Regs}[rs] \ mod \ \text{Regs}[rt]$  
  Note: *div* does not generate an arithmetic exception and software should test for zero divisor.

- **sll rd, rt, shamt**; *shift left logical*:  
  $\text{Regs}[rd] \leftarrow \text{Regs}[rt] \ll \text{shamt}$

- **lui rt, immediate**; *load upper immediate*:  
  $\text{Regs}[rt] \leftarrow \text{immediate} \ || \ 16\text{-bit zero-bits}$

### Arithmetic FP Instructions

- **add.d fd, fs, ft**; *FP double precision add*:  
  $\text{Regs}[fd][||\text{Regs}[fd+1]] \leftarrow \text{Regs}[fs][||\text{Regs}[fs+1] + \text{Regs}[ft][||\text{Regs}[ft+1]]$

- **mul.s fd, fs, ft**; *FP single precision multiply*:  
  $\text{Regs}[fd] \leftarrow \text{Regs}[fs] \times \text{Regs}[ft]$

  Plus: **add.s, sub.d, sub.s, mul.d, div.d, div.s**, several convert floating point to/from integer instructions, several compare instructions.  
  Note: Any of instructions above may cause FP exception.

  All instructions presented so far, in addition, increment the program counter PC by 4, i.e.  
  
  $$\text{PC} \leftarrow [\text{PC}] + 4$$