Computer Instructions

CSE 410, Spring 2007
Computer Systems

http://www.cs.washington.edu/410

Reading and References

• Readings
  » Computer Organization and Design
    • Section 2.1, Introduction
    • Section 2.2, Operations of the Computer Hardware
    • Section 2.3, Operands of the Computer Hardware
    • Section 2.4, Representing Instructions

• Other References
  » See MIPS Run, D Sweetman
    • section 8.6, Instruction encoding
    • section 10.2, Endianness

A very simple organization

Instructions in main memory

• Instructions are stored in main memory
  » each byte in memory has a number (an address)

• Program counter (PC) points to the next instruction
  » All MIPS instructions are 4 bytes long, and so instruction addresses are always multiples of 4

• Program addresses are 32 bits long
  » $2^{32} = 4,294,967,296 = 4 \text{ GigaBytes (GB)}$
Instructions in memory

<table>
<thead>
<tr>
<th>instruction addresses</th>
<th>:</th>
<th>:</th>
<th>:</th>
<th>:</th>
<th>:</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>instruction value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>instruction value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fetch/Execute Cycle (Preview)

• Operation of a computer:
  while (processor not halted) {
    fetch instruction at memory location (PC)
    PC = PC + 4 (increment to point to next instruction)
    execute fetched instruction
  }

• Instructions execute sequentially unless a jump or branch changes the PC to cause the next instruction to be fetched from somewhere else

Some common storage units

Note that a byte is 8 bits on almost all machines.
The definition of word is less uniform (4 and 8 bytes are common today).

<table>
<thead>
<tr>
<th>unit</th>
<th># bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
</tr>
<tr>
<td>half-word</td>
<td>16</td>
</tr>
<tr>
<td>word</td>
<td>32</td>
</tr>
<tr>
<td>double word</td>
<td>64</td>
</tr>
</tbody>
</table>

Alignment

• An object in memory is “aligned” when its address is a multiple of its size
• Byte: always aligned
• Halfword: address is multiple of 2
• Word: address is multiple of 4
• Double word: address is multiple of 8
• Alignment simplifies load/store hardware
System organization so far

<table>
<thead>
<tr>
<th>instructions and data</th>
<th>main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>program counter</td>
<td>increments by 4</td>
</tr>
<tr>
<td>registers</td>
<td>32-bit</td>
</tr>
<tr>
<td>functional units</td>
<td></td>
</tr>
</tbody>
</table>

MIPS Registers

- 32 bits wide
  - 32 bits is 4 bytes
  - same as a word in memory
  - signed values from \(-2^{31}\) to \(+2^{31}-1\)
  - unsigned values from 0 to \(2^{32}-1\)
- easy to access and manipulate
  - 32 registers (not related to being 32 bits wide)
  - on chip, so very fast to access

Register addresses

- 32 general purpose registers
- how many bits does it take to identify a register?
  - 5 bits, because \(2^5 = 32\)
- 32 registers is a compromise selection
  - more would require more bits to identify
  - fewer would be harder to use efficiently

Register numbers and names

<table>
<thead>
<tr>
<th>number</th>
<th>name</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero</td>
<td>always returns 0</td>
</tr>
<tr>
<td>1</td>
<td>at</td>
<td>reserved for use as assembler temporary</td>
</tr>
<tr>
<td>2-3</td>
<td>v0, v1</td>
<td>values returned by procedures</td>
</tr>
<tr>
<td>4-7</td>
<td>a0-a3</td>
<td>first few procedure arguments</td>
</tr>
<tr>
<td>8-15, 24, 25</td>
<td>t0-t9</td>
<td>temps - can use without saving</td>
</tr>
<tr>
<td>16-23</td>
<td>a0-a7</td>
<td>temps - must save before using</td>
</tr>
<tr>
<td>26, 27</td>
<td>k0, k1</td>
<td>reserved for kernel use - may change at any time</td>
</tr>
<tr>
<td>28</td>
<td>gp</td>
<td>global pointer</td>
</tr>
<tr>
<td>29</td>
<td>sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp or a8</td>
<td>frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra</td>
<td>return address from procedure</td>
</tr>
</tbody>
</table>
How are registers used?

- Many instructions use 3 registers
  » 2 source registers
  » 1 destination register

- For example
  » `add $t1, $a0, $t0`
    - add a0 and t0 and put result in t1
  » `add $t1,$zero,$a0`
    - move contents of a0 to t1 (t1 = 0 + a0)

R-format instructions: 3 registers

- 32 bits available in the instruction
- 15 bits for the three 5-bit register numbers
- The remaining 17 bits are available for specifying the instruction
  » 6-bit op code - basic instruction identifier
  » 5-bit shift amount
  » 6-bit function code

R-format fields

<table>
<thead>
<tr>
<th>op code</th>
<th>source 1</th>
<th>source 2</th>
<th>dest</th>
<th>shamt</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- some common R-format instructions
  » arithmetic: `add`, `sub`, `mult`, `div`
  » logical: `and`, `or`, `sll`, `srl`
  » comparison: `slt` (set on less than)
  » jump through register: `jr`

Bits are just bits

- The bits mean whatever the designer says they mean when the ISA is defined
- How many possible 3-register instructions are there?
  » $2^{17} = 131,072$
  » includes all values of op code, shamt, function
- As the ISA develops over the years, the encoding tends to become less logical
System organization again

- Instructions and data
- Main memory
- Functional units
- Registers: 32 bits wide, 32 in number
- Instructions: 32 bits wide, 32 in number
- Program counter: increments by 4

Transfer from memory to register

- Load instructions
  - Word: lw rt, address
  - Half word: lh rt, address
  - Byte: lb rt, address
  - Signed load: sign bit is extended into the upper bits of destination register
  - Unsigned load: 0 in upper bits of register

Transfer from register to memory

- Store instructions
  - Word: sw rt, address
  - Half word: sh rt, address
  - Byte: sb rt, address

The “address” term

- There is one basic addressing mode: offset + base register value
- Offset is 16 bits (± 32 KB)
- Load word pointed to by s0, add t1, store
  - lw $t0,0($s0)
  - add $t0,$t0,$t1
  - sw $t0,0($s0)
I-format fields

- The contents of the base register and the offset value are added together to generate the address for the memory reference.
- Can also use the 16 bits to specify an immediate value, rather than an address.

The eye of the beholder

- Bit patterns have no inherent meaning.
- A 32-bit word can be seen as:
  - a signed integer (± 2 Billion)
  - an unsigned integer or address pointer (0 to 4B)
  - a single precision floating point number
  - four 1-byte characters
  - an instruction

Big-endian, little-endian

- A 32-bit word in memory is 4 bytes long.
- but which byte is which address?
- Consider the 32-bit number 0x01234567
  - four bytes: 01, 23, 45, 67
  - most significant bits are 0x01
  - least significant bits are 0x67
Data in memory - big endian

Big endian - **most** significant bits are in byte 0 of the word

<table>
<thead>
<tr>
<th>byte #</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>67</td>
</tr>
<tr>
<td>6</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
</tr>
</tbody>
</table>

Data in memory - little endian

Little endian - **least** significant bits are in byte 0 of the word

<table>
<thead>
<tr>
<th>byte #</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>01</td>
</tr>
<tr>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>67</td>
</tr>
</tbody>
</table>