Instruction Set Architectures

CS 365 Lecture 2
Prof. Yih Huang

Instruction Set Architecture
-- Definition --

- Data types and structures
  - Encoding and representation
- Instruction set
- Instruction format
- Addressing modes, accessing data and instructions
- Exception conditions
Examples

- IBM 360/370
- Motorola PowerPC
- DEC VAX, Alpha
- HP PA-RISC
- Sun Sparc
- SGI MIPS
- Intel X86

Registers

- Registers are a small set of storage cells inside the processor.
  - MIPS provides 32
  - x86 provides 8 for arithmetic
- Registers are directly accessible through machine instructions.
- Registers are much much faster than memory.
### MIPS Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Register #</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>By assembler only</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>More temporaries</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>Reserved for the operating system</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return pointer</td>
</tr>
</tbody>
</table>

---

### Discussion

- Usages are software “conventions.” They are not built into the hardware.
- Except registers 0 and 31, all others are treated the same by the hardware.
  - Register 0 is hardwired 0. Writing to it will not changed its value.
  - Register 31 is implied in the jal instruction.
MIPS Arithmetic

- All instructions have 3 operands, all of them registers
- Example

  C code: \[ A = B + C + D; \]
  \[ E = F - A; \]

  MIPS code:
  \[
  \text{add } \$t0, \$s1, \$s2 \\
  \text{add } \$s0, \$t0, \$s3 \\
  \text{sub } \$s4, \$s5, \$s0
  \]

- Variables must be moved between registers and memory before and after computation.

Arithmetic Instruction Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>6 bits</td>
</tr>
<tr>
<td>rs</td>
<td>5 bits</td>
</tr>
<tr>
<td>rt</td>
<td>5 bits</td>
</tr>
<tr>
<td>rd</td>
<td>5 bits</td>
</tr>
<tr>
<td>shamt</td>
<td>5 bits</td>
</tr>
<tr>
<td>funct</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- opcode=000000
- rd = rs \text{ funct } rt
- Functions (see the third table of Fig 3.18)
  - ADD/SUB/MULT/DIV Signed/Unsigned
  - AND, OR, XOR, NOR
  - Set On Less Signed/Unsigned
- We ignore “shamt” (shift amount) for now
Exercises

- add $t0, $s1, $s2

```
000000 10001 10010 01000 00000 100000
```

opcode  rs  rt  rd  shamt  funct

- xor $a2, $t8, $v0

```
```

opcode  rs  rt  rd  shamt  funct

Arithmetic with Immediates

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- opcode=001XXX
- rt = rs  op Immediate
- XXX determines operations
  - addi, addiu (add immd. signed/unsigned)
  - andi, ori, xori
  - lui (Load Upper Immediate)
- See the 2nd row in 1st table of Fig. 2.25
Exercises

☐ add $t1, $s3, 10

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Func Code</th>
<th>Function</th>
<th>32-Bit Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>10011</td>
<td>01001</td>
<td>00000000000001010</td>
</tr>
</tbody>
</table>

XXX=000  rs=$s3  rt=$t1  Immediate = 10

☐ add $t1, $s3, -10

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Func Code</th>
<th>Function</th>
<th>32-Bit Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>10011</td>
<td>01001</td>
<td>000000000000-1010</td>
</tr>
</tbody>
</table>

XXX=000  rs=$s3  rt=$t1  Immediate = -10

Load 32-bit Constants

☐ Example: to load the number below into $s0,

\[
\begin{array}{l}
00000000 00111101 00001001 00000000 \\
\end{array}
\]

\[
61 \quad 2304
\]

☐ Step 1: lui $s0, 61 (opcode: 001111)
  – The value of $s0 becomes

\[
\begin{array}{l}
00000000 00111101 00001001 00000000 \\
\end{array}
\]

☐ Step 2: addi $0, $s0, 2304
Memory

- A set of data entries indexed by addresses
- Typically the basic data unit is byte
- In 32 bit machines, 4 bytes grouped to words
- Have you seen the DRAM chips in your PC?

Where Are the Variables?

- Each variable has a home in memory.
  - A particular location in the memory is assigned to store the content of the variable.
- The address of a variable is determined by
  - The compiler in high level languages
  - You if you do machine programming
- Variables are moved from memory to registers before computations.
Example: \( C = A + B \)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>0002</td>
</tr>
</tbody>
</table>

Processor

- S0
- S1
- S2
- Registers

Discussions

- Registers are part of the processor
  - Fast but limited in numbers
- Memory is slooooooooooow.

- For best performance,
  - Use registers for important data
  - Minimize the traffic to/from memory
Load and Store Instructions

- Move data between memory and registers.
- Example:


  MIPS code:  
  \[
  lw \quad t0, \quad 32(s3) \\
  add \quad t0, \quad s2, \quad t0 \\
  sw \quad t0, \quad 28(s3)
  \]

- Store word (sw) has destination last

Instruction Format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- Load: opcode=100XXX
  - $rt = Memory[rs + Offset]$
- Store: opcode = 101XXX
  - Memory[rs + Offset] = $rt$
- XXX determines data sizes (byte, half word, word)
- See the 5th and 6th rows in 1st table of Fig. 2.25
Example

- `lw $t2, 24($s3)`

  
<table>
<thead>
<tr>
<th>100011</th>
<th>10011</th>
<th>01010</th>
<th>00000000000001100</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXX=011</td>
<td>rs=$s3</td>
<td>rt=$t2</td>
<td>Offset = 24</td>
</tr>
</tbody>
</table>

- `sw $v1, -8($sp)`

  
  | XXX= | rs=$v1 | rt=$sp | Offset = -8 |

Stored Program Concept

- Instructions are bits
- Programs are stored in memory just like data
- How does the processor distinguish instructions from data?
  - It does not (until recently)
  - AMD64 does make the distinction
Instruction Execution Cycles

- A special register called **Program Counter (PC)** points to the address of the next instruction for execution.
- CPU reads from memory the instruction pointed by PC.
- The control logic makes the designated function carried out.
- PC is increased to point to the next instruction
  - PC += 4 in the case of MIPS

Example

- Assembly Instructions
  
  ```
  lw $t2, 24($s3)
  add $s0, $s1, $s2
  ```

- Machine instructions in memory
  
  ```
  ... 1004: 10001110 01101010 00000000 00001100
  1008: 00000010 00110010 10000000 00100000
  ... 
  ```
Example Continued

<table>
<thead>
<tr>
<th>Memory</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1004</td>
<td>S0</td>
</tr>
<tr>
<td></td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>S2</td>
</tr>
<tr>
<td></td>
<td>Registers</td>
</tr>
<tr>
<td>1008</td>
<td>00010010</td>
</tr>
<tr>
<td>1008</td>
<td>10000000</td>
</tr>
<tr>
<td>1008</td>
<td>00100000</td>
</tr>
</tbody>
</table>

Control Instructions

- Sequential execution is achieved by increasing PC += 4.
- Non-sequential executions:
  - If, switch, loops in C++
  - Need to give a different “next” instruction address other than PC+4.
- Control instructions update the PC.
- They are also called **branch instructions**, for they allow the processor to branch from the default execution path.
Conditional Braches

- MIPS conditional branch instructions:

  \[\text{bne } \$t0, \$t1, \text{Label}\]
  \[\text{beq } \$t0, \$t1, \text{Label}\]

- Example: if (i==j) \(h = i + j\);

  \[\text{bne } \$s0, \$s1, \text{Label}\]
  \[\text{add } \$s3, \$s0, \$s1\]
  \[\text{Label: \ldots}\]

Instruction Format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- \(\text{opcode} = 000XXX\), where \(XXX \neq 000\)
- \(\text{PC=PC+Offset, if test condition==true}\)
- Test conditions
  - \(XXX=100, \$rs==\$rt\) (beq)
  - \(XXX=101, \$rs!=\$rt\) (bne)
  - \(XXX=110, \$rs<0\) (blez)
  - \(XXX=111, \$rs>0\) (bgtz)
Unconditional Branches

- Assembly: `j label`

<table>
<thead>
<tr>
<th>opcode</th>
<th>26-bit Immediate</th>
</tr>
</thead>
</table>

- Action:

Control

- MIPS unconditional branch instructions:
  
  `j label`

- Example:

  ```
  if (i!=j) 
      h=i+j; 
      beq $s4, $s5, Lab1 
      add $s3, $s4, $s5 
      j Lab2 
  else 
      h=i-j; 
      Lab1: sub $s3, $s4, $s5 
      Lab2: ...
  ```
Exercise

☐ Give the machine instruction of the “beq” in the previous page.

Supporting Procedure Calls

F() {
    ...
    G();
    ...
}  

G() {
    ...
    H();
    ...
}  

H() {
    ...
    ...
}  

☐ Observe the control flow thru the 3 proc.
☐ Need machine instructions to remember the return point.
**JAL: Jump and Link**

- Jump to address of procedure, while storing the return address (PC+4) in register $31 ($ra) and jump to target.

| 000011 | 26-bit Immediate |

- To return from the procedure, execute `jr $31`

---

**jr: jump register instruction**

- `jr $s1`  # jump to address in $s1

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</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>

- Opcode=000000, funct=001000
- Action: PC←rs.
Summary: MIPS Instruction Formats

R-format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

I-format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>16-bit Immediate</th>
</tr>
</thead>
</table>

J-format

<table>
<thead>
<tr>
<th>opcode</th>
<th>26-bit Immediate</th>
</tr>
</thead>
</table>

Assembly vs. Machine Languages

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
Assembly can provide 'pseudoinstructions'
  – e.g., “move $t0, $t1” exists only in Assembly
  – would be implemented using “add $t0,$t1,$zero”

When considering performance you must count real instructions