Chapter 2
Instruction: Language of the Computer

Table of Contents

- Ch.1 Introduction
- Ch. 2 Instruction: Machine Language
- Ch. 3-4 CPU Implementation
- Ch. 5 Cache and VM
- Ch. 6-7 I/O & Multiprocessors

Computer

User interface
keyboard/ mouse
screen/ speaker

Cpu

ALU, Mux, Memory, Sequential circuit,

Software interface (ch.2)

Hardware interface (ch.3-5)

“Textbook subtitle”
Table of Contents

- Ch. 1 Introduction
- Ch. 2 Instructions: Language of the Computer
  - 2.1 Introduction
  - 2.2 Operations (arithmetic, memory operations)
  - 2.3 Operands
  - 2.4 Signed and unsigned numbers
  - 2.5 Representing instructions
  - 2.6 Logical operations
  - 2.7 Control flow operations
  - 2.8 Supporting procedures
  - 2.9 Comm. with people (ASCII)
  - 2.10 MIPS addressing
  - 2.11 Synchronization
  - 2.12 Starting a program, Compiler
  - 2.13 Example
  - 2.14-21 Etc.
- Ch. 3 CPU Implementation: Arithmetic
- Ch. 4 CPU Implementation: Pipeline
- Ch. 5 Cache and Virtual Memory
- Ch. 6-7 I/O and Multiprocessors

Instructions:

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive
  e.g., MIPS Arithmetic Instructions

- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980’s
  - used by NEC, Nintendo, Silicon Graphics, Sony
    (NEC MobileGear, Nintendo64, Sony Playstation, Aibo, STB,
    Routers, Printers, Fax machines,...)

  Design goals: maximize performance and minimize cost, reduce design time
MIPS arithmetic

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

Example:

C code: \[ A = B + C \]

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

MIPS arithmetic

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

C code:

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

MIPS code:

`add $t0, $s1, $s2`
`add $s0, $t0, $s3`
`sub $s4, $s5, $s0`

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?
### MIPS arithmetic

- **How about “memory operand”?**
- **For example**
  - \( \text{my\_account} = \text{my\_account} + \text{deposit\_amount}; \)

- **Data**
  - Data is stored in memory
  - Each variable has its address

- **Translate to MIPS code**
  - \([160] = [160] + [200];\)

- **But, MIPS does not allow memory operands in the arithmetic operation**

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>149000</td>
<td>my_account</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>deposit_amount</td>
</tr>
</tbody>
</table>

### 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**
  - Compiler tries to keep the most frequently used variables in registers and places the rest in memory (spilling registers)
  - Compiler uses load/store to move variables between registers and memory
General Purpose Registers Dominate

- 1975-1995 all machines use general purpose registers
- Advantages of registers
  - registers are faster than memory
  - registers are easier for a compiler to use
    - e.g., \((A*B) - (C*D) - (E*F)\) can do multiplies in any order vs. stack
  - registers can hold variables
    - memory traffic is reduced, so program is sped up (since registers are faster than memory)
    - code density improves (since register named with fewer bits than memory location)

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. (Since 1980, almost every machine uses addresses to level of 8 bits.)
Memory Organization

- For referring a data item, we usually use “word” because a byte is too small to represent data.
- For MIPS, a word is 32 bits or 4 bytes.

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

- If MIPS has 10-bit address pin
  - How many bytes are addressable?
  - How many words are addressable?
  - What is the address range?
- $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$
- $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32}-4$

Memory Addressing

- 2 questions for design of ISA:
  - Since we could read a 32-bit word as four loads of bytes from sequential byte addresses or as one load word from a single byte address,
  => how do byte addresses map onto words?
  - Can a word be placed on any byte boundary?

E.g.,
- A byte can hold a value of upto 255 ($2^8-1$)
- A word can hold a value of upto $2^{32}-1$

- $[160] = 149000 = 24608$ (hexa)
  => 00 02 46 08 (4 bytes or a word)
  (So, the next address is “164”)
- $[160]w = 00 02 46 08$

- $[160]b=00, [161]b=02, [162]b=46, [163]=08$ ??
  => “Big endian”
- $[160]b=08, [161]b=46, [162]b=02, [163]=00$ ??
  => “Little endian”
Addressing Objects: Endianess and Alignment

- **Big Endian:** address of most significant byte = word address
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian:** address of least significant byte = word address
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

Alignment: require that objects fall on address that is multiple of their size
- e.g., “[161]w” incurs “CPU error”

Endian-ness

  - Ch.4 Classful Internet Address
  - Section 4.17 Network Byte Order, pp. 74-75
Data Transfer (Memory) Instructions

- Load and store instructions
- The only instructions to access memory
- Example:

<table>
<thead>
<tr>
<th>C code: A = B + C</th>
<th>MIPS code:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lw $s0, ($s3)</td>
</tr>
<tr>
<td></td>
<td>lw $s1, ($s4)</td>
</tr>
<tr>
<td></td>
<td>lw $s2, ($s5)</td>
</tr>
<tr>
<td></td>
<td>add $s0, $s1, $s2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s4</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s5</td>
</tr>
</tbody>
</table>

- $s3 : the data itself = “160”
  ($s3) : data in memory addressed by $s3 = “00024608”

- Store word has destination last

Data Transfer (Memory) Instructions

- Load and store instructions with indexed addressing
- Example:

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lw $s2, h</td>
</tr>
<tr>
<td></td>
<td>lw $s3, A</td>
</tr>
<tr>
<td>Right ???, (h and A)</td>
<td>lw $t0, 32($s3)</td>
</tr>
<tr>
<td></td>
<td>add $t0, $s2, $t0</td>
</tr>
<tr>
<td></td>
<td>sw $t0, 32($s3)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>$s3</td>
</tr>
</tbody>
</table>

why not $A, A+1, A+2, A+3, .......... ?
Basic ISA Classes

Stack (0 register):
0 address add # tos ← tos + next

Accumulator (1 register):
1 address add A # acc ← acc + mem[A]
1+x address addx A # acc ← acc + mem[A + x]

General Purpose Register:
2 address add A B # EA(A) ← EA(A) + EA(B)
3 address add A B C # EA(A) ← EA(B) + EA(C)

Load/Store:
3 address add Ra Rb Rc # Ra ← Rb + Rc
load Ra Rb # Ra ← mem[Rb]
store Ra Rb # mem[Rb] ← Ra

Comparison of ISA Classes

Bytes per instruction? Number of Instructions? Cycles per instruction?

Comparing Number of Instructions

° Code sequence for C = A + B for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C, R3</td>
</tr>
</tbody>
</table>
So far we’ve learned:

- **MIPS**
  - loading words but addressing bytes
  - arithmetic on registers only

- **Instruction** | **Meaning**
  - `add $s1, $s2, $s3` | $s1 = $s2 + $s3
  - `sub $s1, $s2, $s3` | $s1 = $s2 - $s3
  - `lw $s1, 100($s2)` | $s1 = Memory[$s2+100]
  - `sw $s1, 100($s2)` | Memory[$s2+100] = $s1