Chapter 2: Instructions
How we talk to the computer
The Instruction Set Architecture

- that part of the architecture that is visible to the programmer
  - instruction formats
  - opcodes (available instructions)
  - number and types of registers
  - storage access, addressing modes
  - exceptional conditions
Overall goals of ISA

- Can be implemented by simple hardware
- Can be implemented by fast hardware
- Instructions do useful things
- Easy to write (or generate) machine code – i.e. good interface to compiler
The Instruction Execution Cycle

Instruction Fetch

Instruction Decode

Operand Fetch

Execute

Result Store

Next Instruction

Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine successor instruction
Key ISA decisions

instruction length
  ▪ are all instructions the same length?

how many registers?

where do operands reside?
  ▪ e.g., can you add contents of memory to a register?

instruction format (encoding)
  ▪ which bits designate what??

operands
  ▪ how many? how big? immediates?
  ▪ how are memory addresses computed?

operations
  ▪ what operations are provided??
Running examples

We’ll look at four example ISA’s:

- Digital’s **VAX** (1977) - uber-complex (“eval polynomial”)
- Intel’s **x86** (1978) - ugly, but ubiquitous (IBM PC)
- **MIPS** - focus of text, used in SGI/embedded machines
- **PowerPC** - used in Mac’s, IBM high-end computers, ...

- VAX and x86 are **CISC** (“Complex Instruction Set Computers”)
- **MIPS** and PowerPC are **RISC** (“Reduced Instruction Set Computers”)
  - almost all machines of 80’s and 90’s are RISC
    - including VAX’s successor, the DEC Alpha
Instruction Length

Variable:

- x86 - Instructions vary from 1 to 17 Bytes long
- VAX - from 1 to 54 Bytes

Fixed:

- MIPS, PowerPC, and most other RISC's:
  all instruction are 4 Bytes long
Instruction Length

- **Variable-length instructions (x86, VAX):**
  - require multi-step fetch and decode.
  - difficult to determine start of next instruction
    - this is a bottleneck in x86 architecture
    - trace cache added to P4 to address this,
      but abandoned in Core 2 Duo because of power
    + allow for a more flexible and compact instruction set.

- **Fixed-length instructions (RISC's)**
  + allow easy fetch and decode.
  + simplify pipelining and parallelism.
  - inefficient use of instruction bits (some things can not be
    encoded in one instr e.g., 32 bit immediates)

- **Future: Post-RISC ideas**
  - allow limited set of variable length encodings, but ensure that
    the encoding allows for fast decoding + fast determination of
    next instruction (e.g., Heads and Tails architecture by Asanovic)
How many registers?

All computers have a small set of **registers**, stored in a register file (“RF”)

RF: Small Memory to hold values that will be used soon
Typical instruction will use 2 or 3 registers as input

**Advantages of a larger number:**
- Fewer memory operations
- Easier to do several operations at once
- Higher Performance, as long as does not impact cycle time

**Advantages of a small number of registers:**
- Less hardware
- Faster access (shorter wires, fewer gates)
- Faster context switch (when all registers need saving)

Generally, these factors have become less important over time
How many registers?

VAX - 16 registers
  R15 is program counter (PC)
  Cute: Loading R15 is a jump instruction, Add is branch, etc.

x86 - 8 general purpose regs Fine print - some restrictions apply
  Plus floating point and special purpose registers

x86-64 16 general purpose regs

Most RISC's have 32 int and 32 floating point regs
  Plus some special purpose ones
    • PowerPC has 8 four-bit “condition registers”, a “count register” (to hold loop index), and others.
    Itanium has 128 fixed, 128 float, and 64 “predicate” registers
Where do operands reside?

**Stack machine:**

“Push A” loads memory into 1st register (“top of stack”), moves other regs down

“Pop C” takes top of stack, puts in memory at C

“Add” combines contents of first two registers, moves rest up.

**Accumulator machine:**

Only 1 register (called the “accumulator”)

Instructions include

“store A” (store what’s in acc into A)

“load A” (load what’s in A into acc, overwriting)

“add A” (add what’s in A to value in acc)
Where do operands reside?

**Register-Memory machine:**

Arithmetic instructions can use data in registers and/or memory

Instructions include:

Add A, B, C

//Get values from memory locations B and C, add them and put into memory in A

**Load-Store Machine (aka Register-Register Machine):**

Arithmetic instructions can only use data in registers.

Instructions are like you know from SPARC

Load R2, A
Store B, R3
Add R4, R3, R2
Load-store architectures

can do:

\[
\begin{align*}
&\text{add } r1 = r2 + r3 \\
&\text{load } r3, M(\text{address}) \\
&\text{store } r1, M(\text{address})
\end{align*}
\]

⇒ forces heavy dependence on registers, which is exactly what you want in today’s CPUs

can’t do:

\[
\begin{align*}
&\text{add } r1 = r2 + M(\text{address})
\end{align*}
\]

- more instructions
+ fast implementation (e.g., easy pipelining)
Where do operands reside?

**VAX: register-memory**

Very general. 0, 1, 2, or 3 operands can be in reg/mem

**x86: register-memory ...**

But floating-point registers are a stack. Not as general as VAX instructions

**RISC machines:**

Always load-store machines

Stack, Accumulator machines?
Comparing the Number of Instructions

**Code sequence for** \( C = A + B \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register-Memory</th>
<th>Load-Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Add C,A,B</td>
<td>Load R1, A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></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>

What else might you compare for?
Comparing the Number of Instructions

Code sequence for $A = X*Y + X*Z$

Stack  Accumulator  Register-Memory  Load-Store
Key ISA decisions

- instruction length
  - are all instructions the same length?
- how many registers?
- where do operands reside?
  - e.g., can you add contents of memory to a register?

instruction format
- which bits designate what??

operands
- how many? how big?
- how are memory addresses computed?

operations
- what operations are provided??
Instruction formats
-what does each bit mean?

Having many different instruction formats...
  • complicates decoding
  • uses instruction bits (to specify the format)

Machine needs to determine quickly,
  • “This is a 6-byte instruction”
  • “This instruction probably branches to XXX”
  • ...