Instruction Set Architectures

Interface between hardware and low-level software

Goal: Find a language that makes it easy to build both the hardware and the compiler while maximizing performance and minimizing cost

Programmer's View - add, subtract, and, or, compare, ...

Computer's View - strings of 1s and 0s

The Stored Program Computer

Princeton (Von Neumann) Architecture - stored program computer - data and instructions mixed in same memory

- better storage utilization
- single memory interface
- program as data (dubious advantage)

Harvard Architecture - data and instructions in separate memory

- has advantages in some high performance implementations

Memory Addressing

Memory can be considered a large single-dimensional array

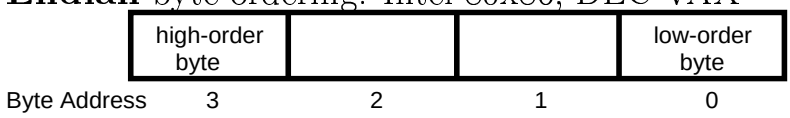
Memory Address - index to that array starting at 0

Addresses at level of 8-bits (byte) - smallest unit addressable

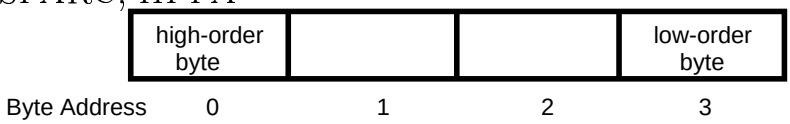
The instruction set architecture
- determines the size of a single load
- can require alignment of a word on byte boundary (multiple of size)
- enforces mapping of byte addresses onto words

Byte Ordering

Little Endian byte ordering: Intel 80x86, DEC VAX

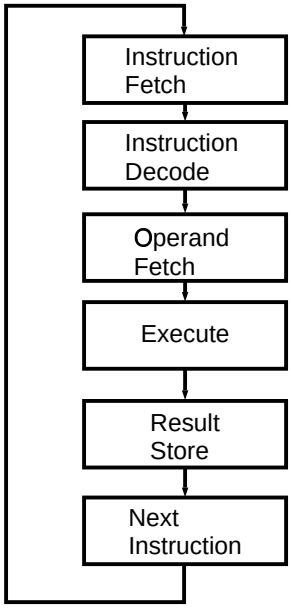


Big Endian byte ordering: IBM 360/370, Motorola 68K, MIPS, SPARC, HPPA



Problem when they need to communicate

Execution Cycle



Basic Issues In Instruction Set Design

What operations (and how many) should be provided

How (and how many) operands are specified

What data types and sizes

How to encode these into consistent instruction formats

Basic ISA Classes

Accumulator

```
    1 address   add A              acc <-- acc + mem[A]
```

Stack

```
    0 address   add                tos <-- tos + next
```

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
```

Comparing ISA Classes

Bytes per instruction? Number of instructions? Cycles per instruction?

Code sequence for C = A + B

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add R3, R1, R2
Pop C			Store C, R3

Why General Purpose Registers since 1975?

Registers are faster than memory

Registers are easier for a compiler to use - e.g., can perform operations in any order

Registers can hold variables

- memory traffic is reduced so program is sped up
- code density improves, since register can be named with fewer bits

Principles of Computer Design

Smaller is faster - number of registers

Simplicity favors regularity

Good design demands compromise - fixed format vs. size

Make the common case fast