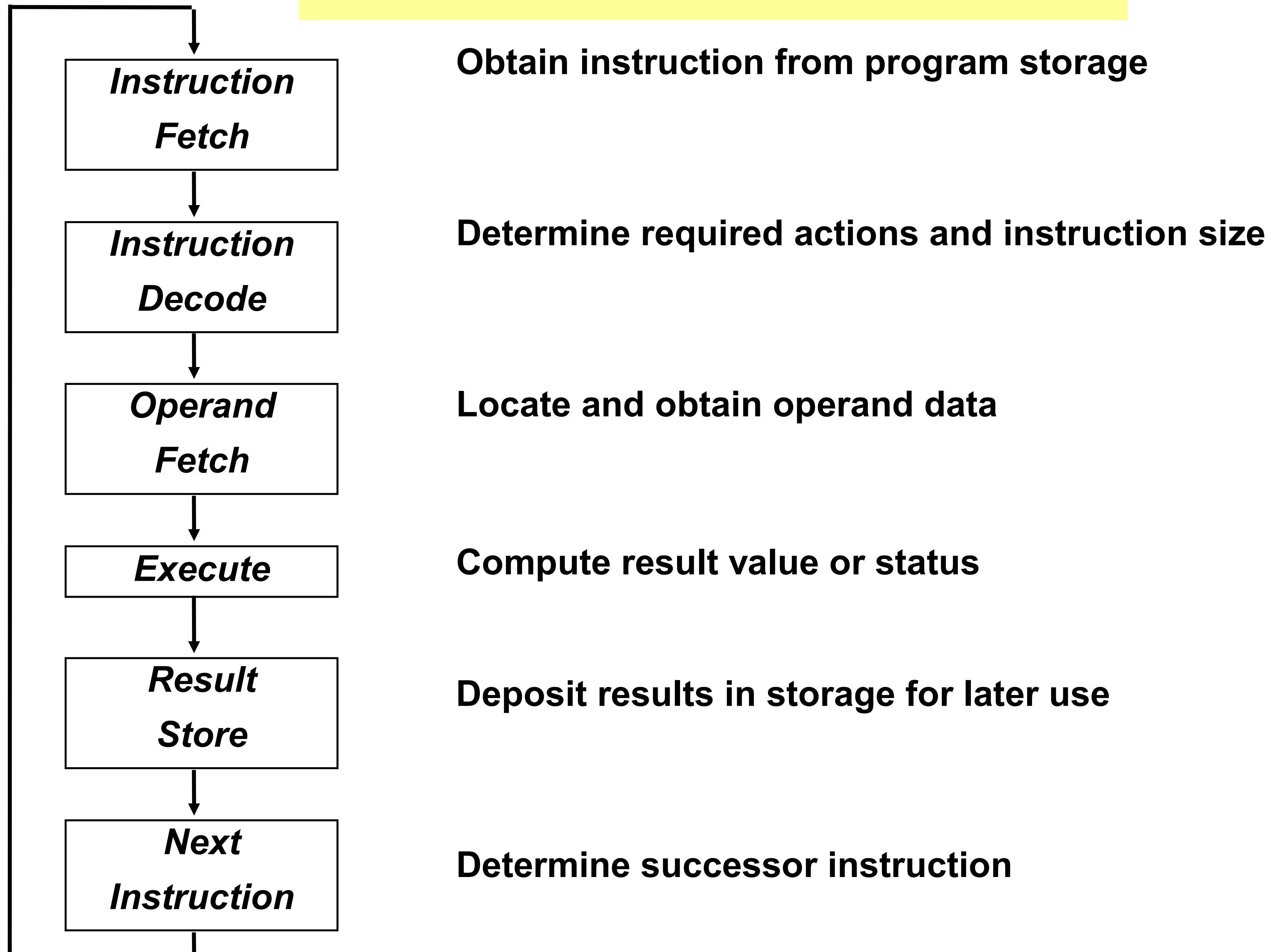
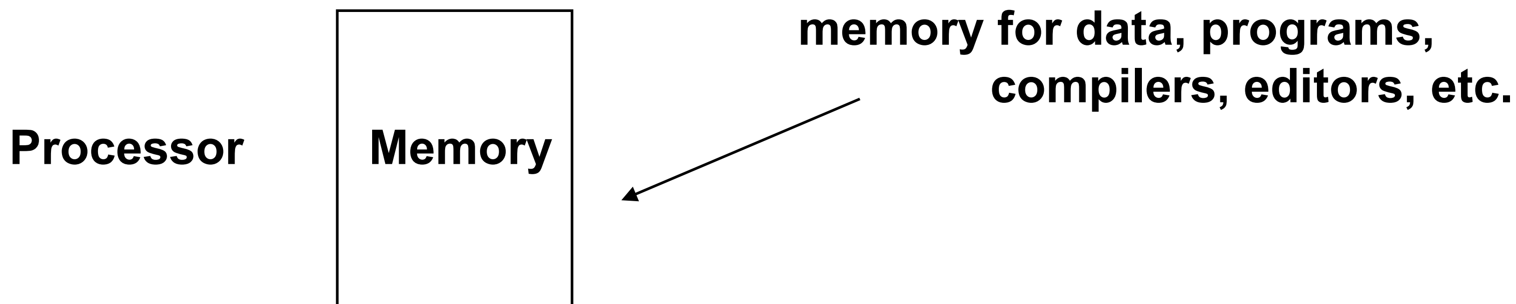
Architecture

Execution Cycle



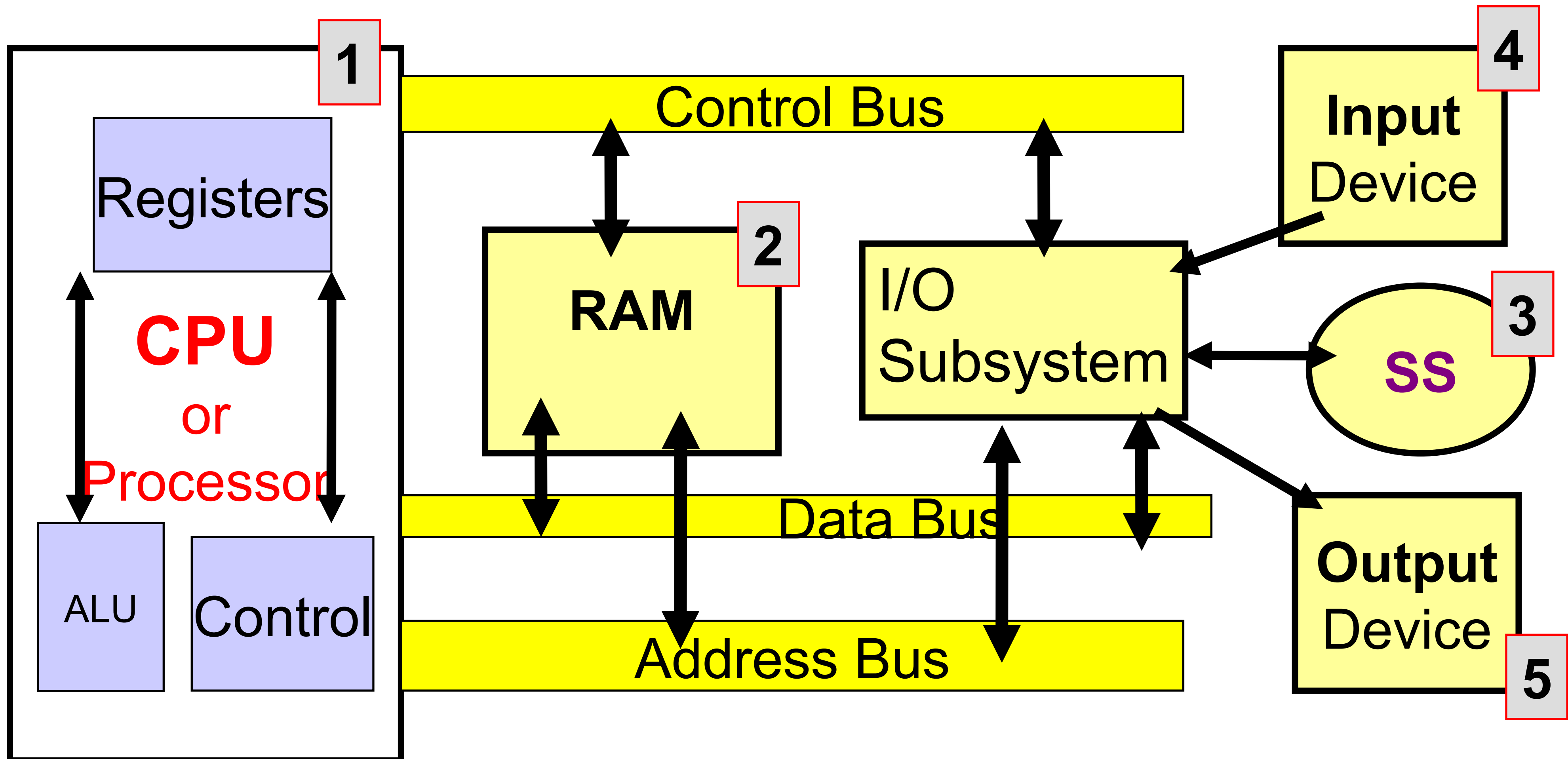
Stored Program Concept

- Instructions are bits
- Programs are stored in memory to be read or written just like data



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

5 Main Hardware Components



What is a computer?

- Components:
  - Central Processing Unit (CPU):
    - ALU and control unit
      - implemented using millions of transistors
      - impossible to understand looking at each transistor
    - Registers: fast, few, $
  - Memory hierarchy
    - Registers, cache, RAM, disk
    - Faster, smaller, more expensive: up the pyramid
  - Input (mouse, keyboard)
  - Output (display, printer)

What is “Computer Architecture”

Computer Architecture =

Instruction Set Architecture

+

Machine Organization

Instruction Set Architecture

.. the view of the computer from the assembly language programmers perspective

Machine Organization (Implementation)

the logic design, and the physical implementation.

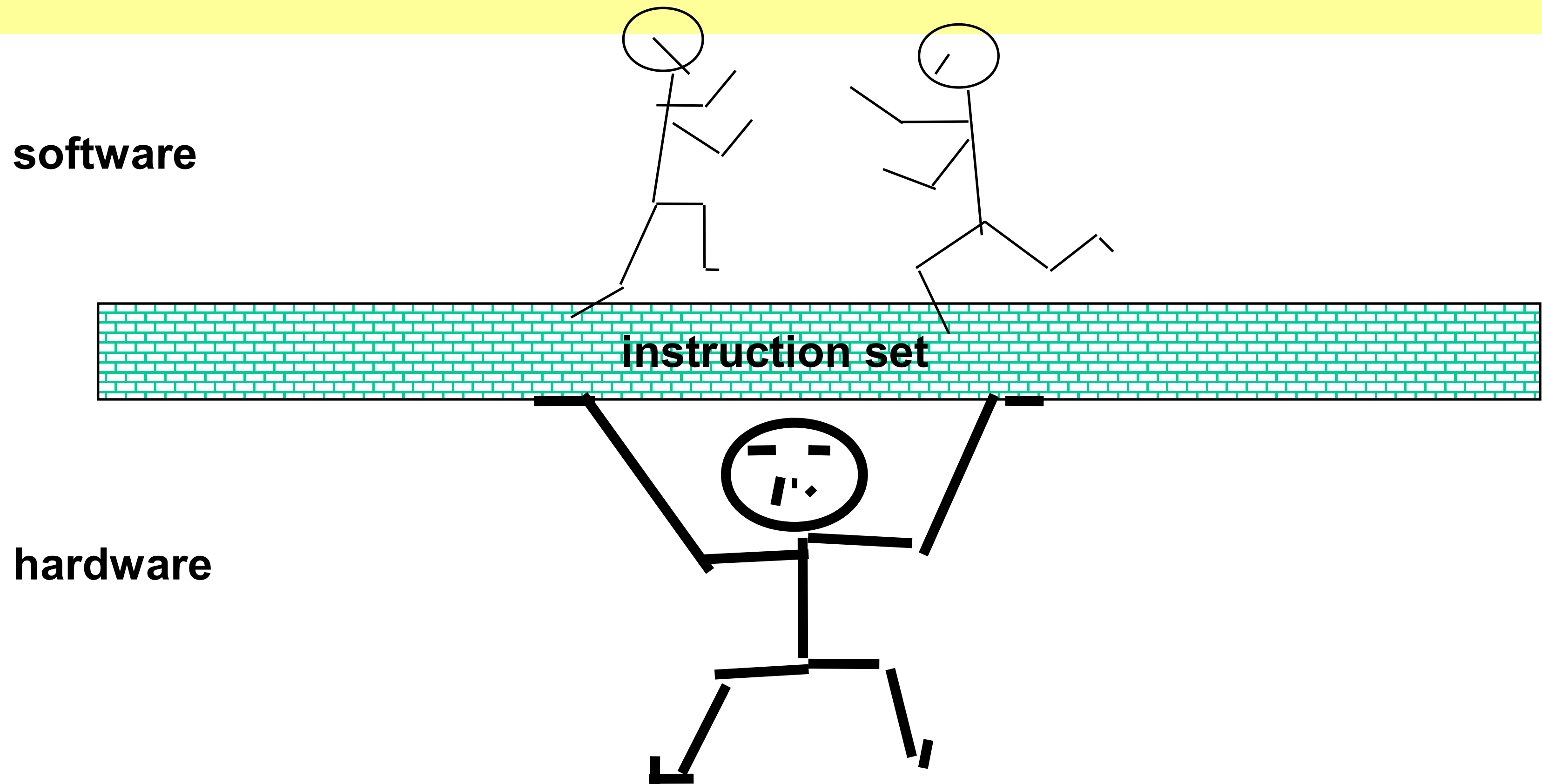
the “electronics” and how they work to implement the ISA

Instruction Set Architecture vs Machine Organization

- You buy a new computer with a NEW Intel CPU - do you have to change the software??
- Why or why not??
- How about if you buy an AMD?
- How about if you buy a MAC?

ISA's: 80x86/Pentium/K6 , PowerPC, DEC Alpha, MIPS, SPARC, HP, DEC VAX

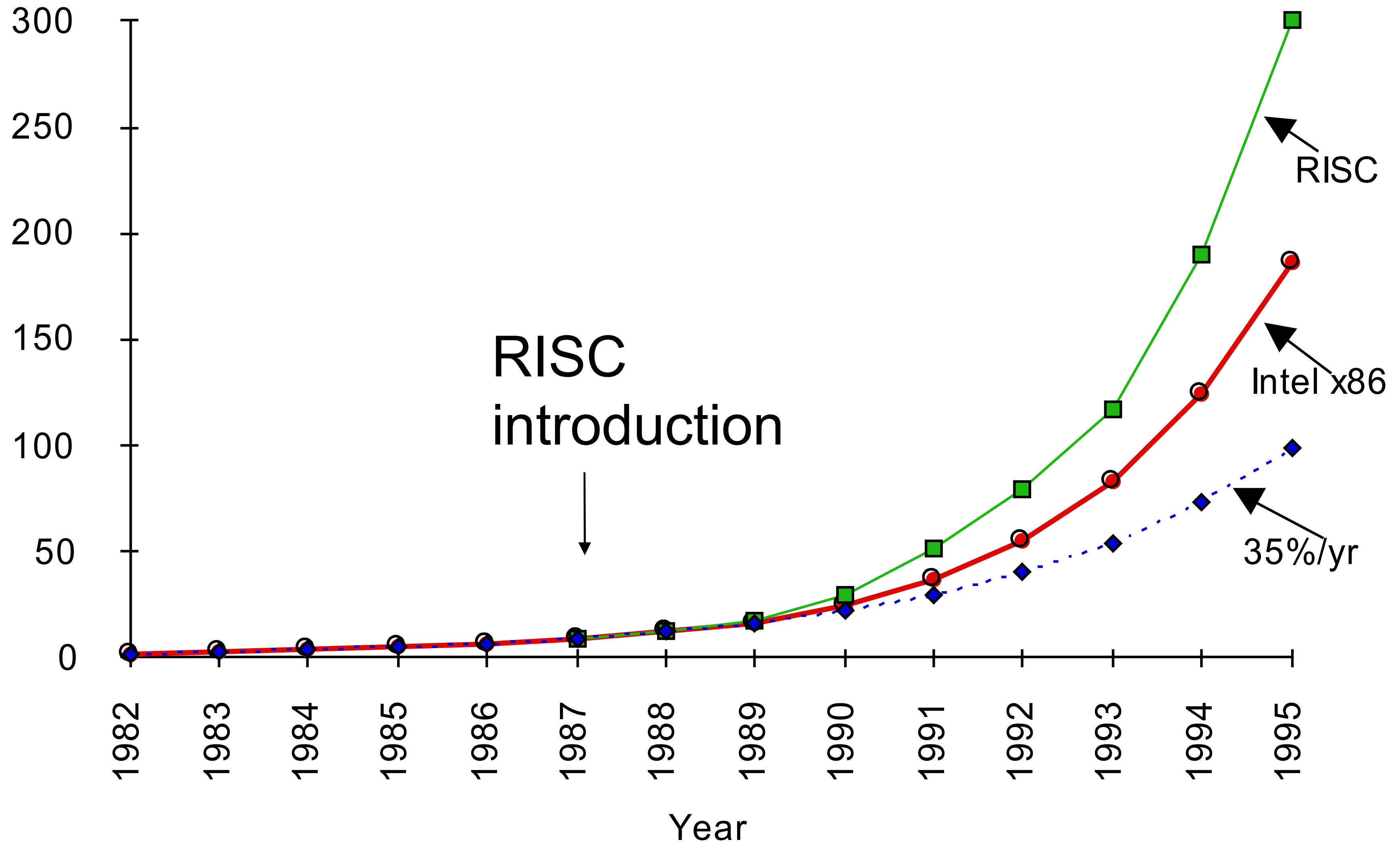
The Instruction Set: a Critical Interface



Technology => Dramatic Changes

- Moore's Law: # of transistors on an IC doubles every 2 years
- Processor
  – logic capacity: improves about 30% per year
  – clock rate: about 20% per year
- Memory
  – DRAM capacity: about 60% per year
  – Memory speed: about 10% per year
- Disk
  – capacity: about 60% per year

Processor Performance



MHz Myth

- Measuring MHz to figure out how fast your CPU is, is like measure RPM's in a car to decide how fast it will go. A 1hp motor running at 5,000 rpm's is **not** going to make the same car go faster than a 300hp motor at 3,000 rpm's.
- So MHz is a silly rating. It rates how fast **a particular chip** is going - which is an OK indicator relative to other versions of that exact same chip, but the ratings are almost worthless when comparing across processors. (Most application tests done cross platform rate the Macs as anywhere from 40%-400% faster than PC's at the same clock rate or MHz).

Numbers

- Bits are just bits (no inherent meaning)
  conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0101 0110 0111 1000 1001...
  decimal: 0...2^n-1
- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers

Negative #'s: Possible Representations

Sign Magnitude:	One's Complement	**Two's Complement**
000 = +0	000 = +0	000 = 0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?
  - Negating a two's complement #: invert all bits and add 1
  - Remember: “negate” and “invert” are quite different!

Addition & Subtraction – 2s Complement

- Just like in grade school (carry/borrow 1s)

```
  0111        0111        0110
+ 0001      + 0111      + 0101
  ----        ----        ----
  1000        1110        1011
```

- Two's complement operations easy
  - subtraction using addition of negative numbers

```
  0111 ( 7)
+ 1010 (-6)
```

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  1111
+ 0001
  0000
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

Detecting Overflow

- No overflow when adding a positive and a negative #
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ?
  - Can overflow occur if A is 0 ?

Floating Point (a brief look)

- IEEE 754 floating point standard:
  - single precision:
    - 1 bit sign, 8 bit exponent, 23 bit fraction
  - double precision:
    - 1 bit sign, 11 bit exponent, 52 bit significand
- Leading "1" bit of significand is implicit and is not actually represented

Floating Point

- 2^3 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5}
- 8 4 2 1. 1/2 1/4 1/8 1/16 1/32
- .5 .25 .125 .0625 .03125
- 11.011 = 3.375
- 101.101 = 5.625
- -.001 = - .125
- 11.011 = 3.375 = 1.1011×2^1
- 101.101 = 5.625 = 1.01101×2^2
- -.001 = - .125 = -1.0×2^{-3}

3 components:
1. sign
2. exponent
3. fraction

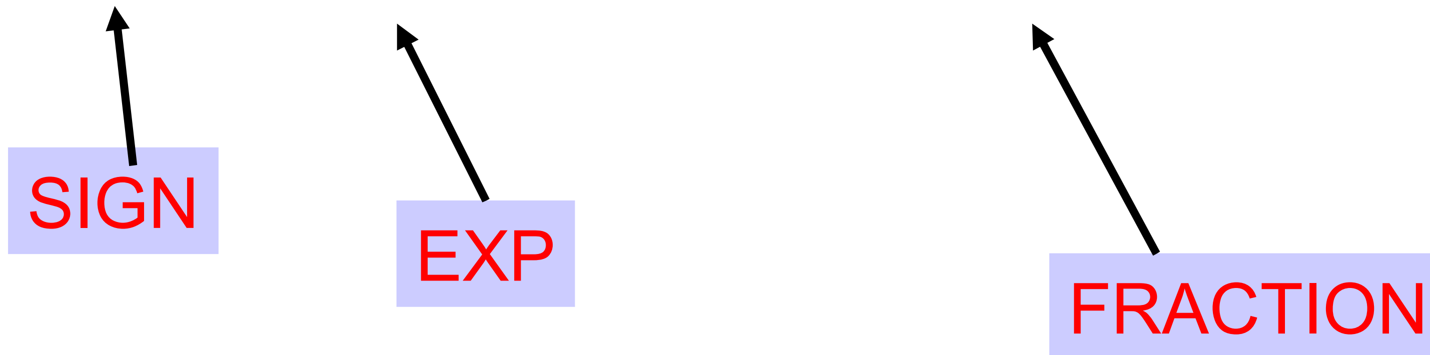
(base is implied = 2)

IEEE 754 floating-point standard

- Exponent is “biased” to make sorting easier
  - bias of 127 for single precision (add 127 encoding and subtract 127 decoding)
  - all 0s is smallest exponent (0-127= -127)
  - all 1s is largest (255-127 =128)
  - summary: $(-1)^{\text{sign}} \times (1+\text{fraction}) \times 2^{\text{exponent}-\text{bias}}$
  - Example:
    - decimal: -.75 = -3/4 = -(½ + ¼)
    - binary: -.11 = -1.1×2^{-1}
    - floating point: exponent = -1+127 = 126 = **01111110**
    - IEEE single precision: 1**01111110**10000000000000000000000

IEEE 754 floating-point standard

- C0A0 0000
- what floating point # is this??
- 1100 0000 1010 0000 0000 0000 0000 0000



Sign =1 , so negative
Exponent = 129, so exp = 129-127 = 2
Fraction = 0100…, so fraction = 1.01

So its $-1.01 \times 2^2 = -101 = -5.0_{10}$

IEEE 754 floating-point standard

- FP 13.25 => Express as 8 hex digits

IEEE 754 floating-point standard

- FP 13.25 => Express as 8 hex digits
- 1101.01 = 1.10101 x 2^3
- 0100 0001 0101 0100 0000 0000 0000 0000

SIGN

EXP

FRACTION

Sign =0 , so positive
Exponent = 3, so exp = 3+127 = 130
Fraction = 1.101010…, so fraction = .10101

So its 13.25 = 1.10101 x 2^5 = 4154 0000_{16}

Floating Point Complexities

- Operations are somewhat more complicated
  - addition
    - 1. compare exponents and shift smaller mantissa right until exponents match
    - 2. add mantissas
    - 3. normalize the sum
    - 4. Round to fit if needed

Floating Point Complexities

- Operations are somewhat more complicated
  - addition: 0.5 + -0.4375
  - $1.0 \times 2^{-1} + -1.11 \times 2^{-2}$
    - 1. compare exponents and shift smaller mantissa right until exponents match $1.0 \times 2^{-1} + -.111 \times 2^{-1}$
    - 2. add mantissas $= 1.0 + (-.111) = .001$
    - 3. normalize the sum 1.0×2^{-4}
    - 4. Round to fit if needed