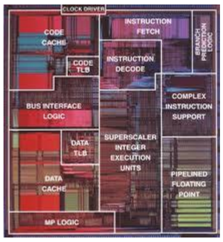
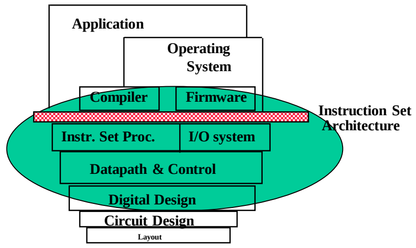
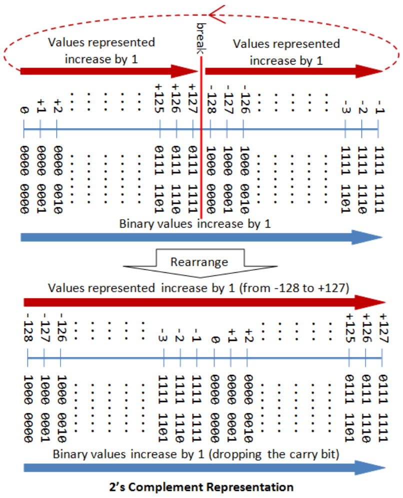
CS/SE 3340
Computer Architecture

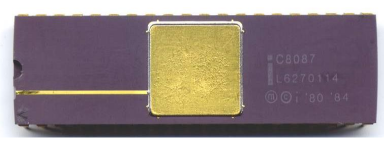


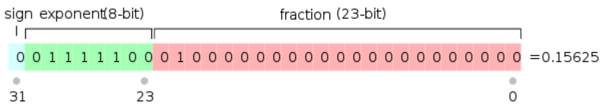


Other Number Representations

Adapted from "Computer Organization and Design, 4th Ed." by D. Patterson and J. Hennessy







Negative Integers

- Natural (positive) integer numbers can be represented easily in memory of a computer in bits (binary format)
- But what about negative integers?
  - How do we handle negative integer numbers in a computer?
- How do we encode negative integers in binary format ?
  - *Can we just think of negative numbers as a positive one with the '-' sign in front of it?*

Representing Negative Integers

- Two common methods: *sign-magnitude representation* and ***complementary representation***
- Sign-magnitude representation
  - Encode sign (+/-) information separately
- Complementary representation
  - Do not encode sign information separately, use a complementary number
  - The complementary has a special relationship (complementary) to the original number and can be derived from it easily

Sign-Magnitude Representation

- The MSb (Most Significant bit) is used to encode the sign
  - ‘1’ means negative, ‘0’ means positive
  - The rest is used to encode the magnitude
- Thus an n-bit word can be used to represent $-(2^{(N-1)} - 1)$ to $+(2^{(N-1)} - 1)$
  - For example an 8-bit word: -127 to +127
  - *Why only 255 possibilities with 8-bit?*

5

Sign-Magnitude Representation Issues

- Zero (0) is represented twice!
  - +0 and -0
  - No such thing as two zeros in math!
- Not intuitive
  - Direct adding a number I and its negative (-I) in binary does not yield 0!
  - Have to treat the sign bit separately
- Not efficient to implement sign handling in hardware!

6

1-complement Representation

- Recall that in complementary representation N (positive) and –N (negative) has a special relationship
  - Must be easy to derive –N from N
- How about reversing all bits of N to get –N?
  - Replacing '1' with '0' and '0' with '1' in N to get -N
  - E.g., in 8-bit words: `9 = 00001001, -9 = 11110110`
- This is 1-complement representation
  - Efficient hardware implementation!
  - Using a NOT (inverse) gate for each bit

1-Complement Representation – cont'd

- An n-bit word can be used to represent

  from $-(2^{(N-1)} - 1)$ to $+(2^{(N-1)} - 1)$

  255 possibilities for 8-bit words, still missing one possibility!
- For 8-bit words

```
+127 = 01111111, -127 = 10000000
+0   = 00000000, -0   = 11111111
```

- Still having the problem of two '0's!
- But at least now I – I = -0!
- How about I + 0 ?
  - Only works with +0!!!

2-complement Representation

- Let tweak the relationship in 1-complement scheme a bit: get –N by inversing all bits and then add 1
  - That is, get 1-complement of N and then add 1 to it
  - For example: with 8-bit words, `9 = 00001001,`
    `-9 = 11110111 (11110110 + 1)`
- This is 2-complement representation
  - Can be implemented easily and efficiently in hardware (inverse and then +1)
- An n-bit word can be used to represent numbers from $-2^{(N-1)}$ to $+(2^{(N-1)} - 1)$

2-complement Representation – cont'd

```
+10    =  00001010
-10    =  11110110
+127   =  01111111
-127   =  10000001
+0     =  00000000
-0     =  00000000
```

- No more two '0's problem!!!
- And I – I = 0!
- *Complementary representation's added benefit: the MSb can be used to encode the sign (+/-)!*

Integer Representations

Example: Using 4 bit numbers

unsigned	sign/magnitude	1's complement	2's complement	
1111 [15]	0111 [7]	0111 [7]	0111	[7]
1110	0110	0110	0110	[6]
1101	0101	0101	0101	[5]
1100	0100	0100	0100	[4]
1011	0011	0011	0011	[3]
1010	0010	0010	0010	[2]
1001	0001	0001	0001	[1]
1000	0000, **1**000 [-0]	0000, **1**111 [-0]	0000	[0]
0111	**1**001	**1**110	**1**111	[-1]
0110	**1**010	**1**101	**1**110	[-2]
0101	**1**011	**1**100	**1**101	[-3]
0100	**1**100	**1**011	**1**100	[-4]
0011	**1**101	**1**010	**1**011	[-5]
0010	**1**110	**1**001	**1**010	[-6]
0001	**1**111 [-7]	**1**000 [-7]	**1**001	[-7]
0000 [0]			**1**000	[-8]

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- *Replicate the sign bit to the left*
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- In MIPS instruction set
  - `addi`: extend immediate value
  - beq, bne: extend the displacement

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation (in base 10)
  - -2.34×10^{56} normalized
  - $+0.002 \times 10^{-4}$ not normalized
  - $+987.02 \times 10^{9}$
- In binary (base 2)
  - $\pm 1.\mathtt{xxxxxxx}_2 \times 2^{\mathtt{yyyy}}$
- Types **`float`** and **`double`** in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  – Portability issues for scientific code
- Now almost universally adopted
- Two representations
  – *Single precision* (32-bit)
  – *Double precision* (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

15

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

16

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

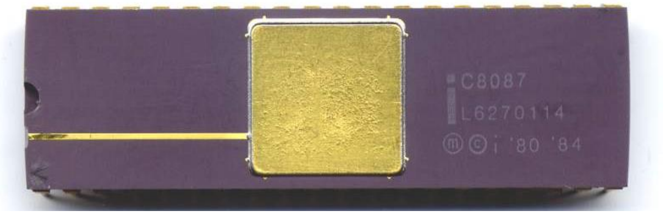
- Represent –0.625 in floating point binary
  $-0.625 = -5/8 = -1 \times 101_2 \times 2^{-3} = \mathbf{-1 \times 1.01_2 \times 2^{-1}}$
  normalized
  - S = 1
  - Fraction = $01000\ldots00_2 = (0/2^1 + 1/2^2 + 0/2^3 + \ldots)$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 101111111110010000…00

Floating-Point Example – cont'd

- What number is represented by the single-precision float below?
  11000000101000…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (\mathbf{1} + 0.01_2) \times 2^{(129 - 127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

Was removed when encoded

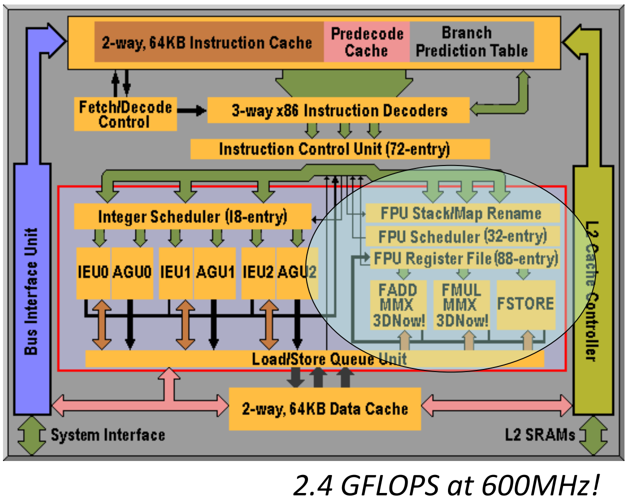
Intel C8087 FPU



- Introduced by Intel in 1980, cost ~$150
- Runs at 4MHz ~ 10MHz
- Can perform 50,000 FLOPS using around 2.4 watts
- Boost application performance by %20 to %500

http://en.wikipedia.org/wiki/Intel_8087

AMD Athlon



2.4 GFLOPS at 600MHz!

http://www.pctechguide.com/amd-technology/amd-athlon