Computer Architecture
(CSC-3501)
Lecture 3
(22 Jan 2008)

Seung-Jong Park (Jay)
http://www.csc.lsu.edu/~sjpark

Announcement

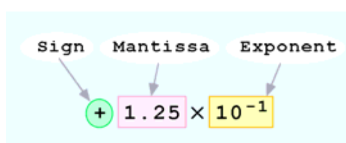
- Today, 1st homework will be uploaded at our class website
  - Due date is the beginning of next lecture
  - Late homework grade will be dropped 20% per date late

Floating Point

- The signed magnitude, one's complement, and two's complement representation that we have just presented deal with integer values only.
- Without modification, these formats are not useful in scientific or business applications that deal with real number values.
- Floating-point representation solves this problem.
- If we are clever programmers, we can perform floating-point calculations using any integer format.
- This is called *floating-point emulation*, because floating point values aren't stored as such, we just create programs that make it seem as if floating-point values are being used.
- Most of today's computers are equipped with specialized hardware that performs floating-point arithmetic with no special programming required.

Floating Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



- Computer representation of a floating-point number consists of three fixed-size fields:
  - The IEEE-754 *single precision* floating point standard uses an 8-bit exponent and a 23-bit significand.
  - The IEEE-754 *double precision* standard uses an 11-bit exponent and a 52-bit significand.

How to Convert to Base-2 Floating Point

- Example:
  - Express 32_{10} in the simplified 14-bit floating-point model.
  - Convert 32_{10} to Base 2
    - $100000_2 = 1.0 \times 2^5$
  - Normalize (leftmost bit of the significand must be 1)
    - $1.0 \times 2^5 = 0.1 \times 2^6$

0	0 0 1 1 0	1 0 0 0 0 0 0 0

00110

For illustrative purposes, we will use a 14-bit model with a 5-bit exponent and an 8-bit significand.

Why Normalization ?

- **The illustrations shown at the right are *all* equivalent representations for 32 using our simplified model.**
- **Not only do these synonymous representations waste space, but they can also cause confusion.**

0	0 0 1 1 0	1 0 0 0 0 0 0 0	0.1×2^6
0	0 0 1 1 1	0 1 0 0 0 0 0 0	0.01×2^7
0	0 1 0 0 0	0 0 1 0 0 0 0 0	0.001×2^8
0	0 1 0 0 1	0 0 0 1 0 0 0 0	0.0001×2^9

- Rule: leftmost bit of the significand must be 1.

How to Represent Negative Exponent 2^{-2}?

- Biased exponent
  - Exponent larger than biased is positive integer exponent
  - Exponent smaller than biased is negative integer exponent
  - Example 1: express 32_{10} in the revised 14-bit floating-point model.
    - We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
    - To use our excess 16 biased exponent, we add 16 to 6, giving 22_{10} (=10110_2).

0	1 0 1 1 0	1 0 0 0 0 0 0 0

- Negative Exponent
  - Example 2: express 0.0625_{10} in the revised 14-bit floating-point model.
    - We know that 0.0625 is 2^{-4}.
    - So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
    - To use our excess 16 biased exponent, we add 16 to -3, giving 13_{10} (=01101_2).

0	0 1 1 0 1	1 0 0 0 0 0 0 0

Another Example of Floating Point

- Example:
  - Express -26.625_{10} in the revised 14-bit floating-point model.
    - We find $26.625_{10} = 11010.101_2$.
      - Integral part $26 = 11010_2$
      - Fractional part $0.625 = 0.101_2$
        - $0.625 \times 2 = 1.25$ 1 (generate 1 and continue with the rest)
        - $0.25 \times 2 = 0.5$ 0 (generate 0 and continue)
        - $0.5 \times 2 = 1.0$ 1 (generate 1 and nothing remains)
    - Normalizing, we have: $26.625_{10} = 11010.101_2 = 0.11010101 \times 2^5$.
    - To use our excess 16 biased exponent, we add 16 to 5, giving 21_{10} (=10101_2). We also need a 1 in the sign bit.

1	1 0 1 0 1	1 1 0 1 0 1 0 1

IEEE-754 Floating Point Standard

- Both the 14-bit model that we have presented and the IEEE-754 floating point standard allow two representations for zero.
  - Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1.
- This is why programmers should avoid testing a floating-point value for equality to zero.
  - Negative zero does not equal positive zero.

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

NaN is used for numbers that don't exist, such as √-1 or log(-5).

History of Character Codes

- The earliest computer coding systems used six bits.
  - Binary-coded decimal (BCD) was one of these early codes. It was used by IBM mainframes in the 1950s and 1960s.
- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
  - EBCDIC was one of the first widely-used computer codes that supported upper *and* lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
  - EBCDIC and BCD are still in use by IBM mainframes today.
- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange) as a replacement for 6-bit codes.
  - While BCD and EBCDIC were based upon punched card codes, ASCII was based upon telecommunications (Telex) codes.
  - Until recently, ASCII was the dominant character code outside the IBM mainframe world.

Character Codes (Cont)

- Many of today's systems embrace Unicode, a 16-bit system that can encode the characters of every language in the world.
  - The Java programming language, and some operating systems now use Unicode as their default character code.
- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

Error Detection

- It is physically impossible for any data recording or transmission medium to be 100% perfect 100% of the time over its entire expected useful life.
  - Thus, error detection and correction is critical to accurate data transmission, storage and retrieval.
- Error Detection
  - Parity bit can check some simple bit error
    - Sometimes high-order bit of ASCII coded to enable detection of errors
    - Even parity – set bit to make number of 1's even

      A (01000001) with even parity is 01000001

      C (01000011) with even parity is 11000011

	Parity (odd)	D7	D6	D5	D4	D3	D2	D1	D0
Transmitted:	0	0	1	1	0	1	1	0	1
Received:	0	0	1	1	0	1	0	0	1

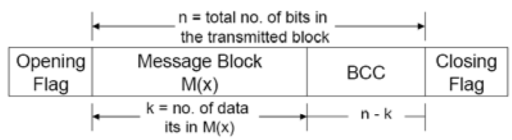
Single bit error

	Parity (odd)	D7	D6	D5	D4	D3	D2	D1	D0
Transmitted:	0	0	1	1	0	1	1	0	1
Received:	0	0	1	1	0	1	0	1	1

Error: two bit or an even number of bits go undetected

Cyclic redundancy checking (CRC)

- CRC utilizes the redundant bits at the end of the block
- It is more powerful method
- Method
  - Original message M(x) / Generator polynomial G(x)
  - Quotient is discarded
  - Remainder is attached to message in BCC (Block Check Character)



- Commonly used cyclic codes
  - CRC-12 G(x) = $x_{12} + x_{11} + x_{3} + x_{2} + x + 1$
  - CRC-16 G(x) = $x_{16} + x_{15} + x_{2} + 1$
  - CRC-CCITT G(x) = $x_{16} + x_{12} + x_{5} + 1$

CRC Example at page 83

- Encode
  - Original Message = 1001011
    - M(x) = $1x^6+0x^5+0x^4+1x^3+0x^2+1x^1+1x^0 = x^6+x^3+x^1+1$
    - G(x) = $1x^3+0x^2+1x^1+1x^0 = x^3+x^1+1$
  - Shift M(x) Make large number before division
    - M(x) X $x^3 = x^9+x^6+x^4+x^3$ = 1001011000
  - M(x)X x^3 / G(x)
    - 1001011000 / 1011 = quotient is 1010 and remainder is 100
  - Add remainder 100 to 1001011000
    - 1001011000 + 100 = 1001011100
- Decode
  - Encoded message / G(x)
    - 1001011100 / 1011 = quotient is 1010100 and remainder is Zero
  - Zero remainder means no error
  - Non-zero remainder means some errors

Error Correction

- Hamming code can detect errors and correct them.
  - Hamming codes are code words formed by adding redundant check bits, or parity bits, to a data word.
  - The *Hamming distance* between two code words is the number of bits in which two code words differ.

    This pair of bytes has a Hamming distance of 3:

1	0	0	0	1	0	0	1
1	0	1	1	0	0	0	1

  - The minimum Hamming distance for a code is the smallest Hamming distance between *all* pairs of words in the code.
  - The minimum Hamming distance for a code, D(min), determines its error detecting and error correcting capability.
  - Hamming codes can *detect* D(min) - 1 errors and *correct* $\left\lfloor \frac{D(Min) - 1}{2} \right\rfloor$ errors

Hamming Code

- Example
  - Using our code words of length 12, number each bit position starting with 1 in the low-order bit.
  - Each bit position corresponding to an even power of 2 will be occupied by a check bit.
  - These check bits contain the parity of each bit position for which it participates in the sum.

_	_	_	_	P	_	_	_	P	_	P	P
12	11	10	9	8	7	6	5	4	3	2	1

- Since 2 (= 2^1) contributes to the digits, 2, 3, 6, 7, 10, and 11. Position 2 will contain the parity for bits 3, 6, 7, 10, and 11.
  - Bit 1checks the digits, 3, 5, 7, 9, and 11, so its value is 1 to make even parity.
  - Bit 4 checks the digits, 5, 6, 7, and 12, so its value is 1.
  - Bit 8 checks the digits, 9, 10, 11, and 12, so its value is also 1.

1	1	0	1	1	0	1	1	1	0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

Hamming Code Cont.

- Suppose an error occurs in bit 5, as shown above. Our parity bit values are:
  - Bit 1 checks digits, 3, 5, 7, 9, and 11. *Its value is 1, but should be zero.*
  - Bit 2 checks digits 2, 3, 6, 7, 10, and 11. The zero is correct.
  - Bit 4 checks digits, 5, 6, 7, and 12. *Its value is 1, but should be zero.*
  - Bit 8 checks digits, 9, 10, 11, and 12. This bit is correct.

1	1	0	1	1	0	1	0	1	0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

  - We have erroneous bits in positions 1 and 4.
  - With *two* parity bits that don't check, we know that the error is in the data, and not in a parity bit.
  - Which data bits are in error? We find out by adding the bit positions of the erroneous bits.
  - Simply, 1 + 4 = 5. This tells us that the error is in bit 5. If we change bit 5 to a 1, all parity bits check and our data is restored.