Computer Architecture
(CSC-3501)
Lecture 2
(17 Jan 2008)

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

Announcement

- 1st Pop Quiz Today

Goal and Objectives of Chapter 2

- Goal
  - Understand how computers think and speak
- Objectives
  - Understand the fundamentals of numerical data representation and manipulation in digital computers.
  - Master the skill of converting between various radix systems.
  - Understand how errors can occur in computations because of overflow and truncation.
  - Understand the fundamental concepts of floating-point representation.
  - Gain familiarity with the most popular character codes.
  - Understand the concepts of error detecting and correcting codes.

Information Representation

- People
  - Calculate base 10 number
  - Basic information unit ?
    - Maybe neuron ?
- Computers
  - 2 base number is easy to represent in digital domain
    - It is a state of “on” or “off” in a digital circuit
    - Sometimes these states are “high” or “low” voltage instead of “on” or “off..”
  - So, ***bit*** is the most basic unit in computers
  - A ***byte*** is a group of eight bits.
    - A byte is the smallest possible *addressable* unit of computer storage.
    - The term, “addressable,” means that a particular byte can be retrieved according to its location in memory.
  - A ***word*** is a contiguous group of bytes.
    - Words can be any number of bits or bytes.
    - Word sizes of 16, 32, or 64 bits are most common depending on compute architecture

Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2.
  - The binary system is also called the base-2 system.
  - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.
  - Any integer quantity can be represented exactly using any base (or *radix*).
- The decimal number 5836.47 in powers of 10 is:

$$5 \times 10^{3} + 8 \times 10^{2} + 3 \times 10^{1} + 6 \times 10^{0} + 4 \times 10^{-1} + 7 \times 10^{-2}$$

- The binary number 11001 in powers of 2 is:

$$\begin{aligned} & 1 \times 2^{4} + 1 \times 2^{3} + 0 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} \\ = \; & 16 + 8 + 0 + 0 + 1 = 25 \end{aligned}$$

CSC3501 – S.J. Park

2.3 Decimal to Base 3 Conversions

- **Converting 190 to base 3...**
  - Continue in this way until the quotient is zero.
  - In the final calculation, we note that 3 divides 2 zero times with a remainder of 2.
  - Our result, reading from bottom to top is:

    $190_{10} = 21001_{3}$

3	190	1
3	63	0
3	21	0
3	7	1
3	2	2
	0	

Fractional Value

- Fractional values can be approximated in all base systems.
- Unlike integer values, fractions do not necessarily have exact representations under all radices.
- The quantity ½ is exactly representable in the binary and decimal systems, but is not in the ternary (base 3) numbering system.
- Fractional decimal values have nonzero digits to the right of the decimal point.
- Fractional values of other radix systems have nonzero digits to the right of the *radix point*.
- Numerals to the right of a radix point represent negative powers of the radix:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

$$\begin{aligned} 0.11_2 &= 1 \times 2^{-1} + 1 \times 2^{-2} \\ &= \tfrac{1}{2} + \tfrac{1}{4} \\ &= 0.5 + 0.25 = 0.75 \end{aligned}$$

Conversion Fractional Number from Base 10 to Base 2

- As with whole-number conversions, you can use either of two methods: a subtraction method and an easy multiplication method.
- The subtraction method for fractions is identical to the subtraction method for whole numbers. Instead of subtracting positive powers of the target radix, we subtract negative powers of the radix.
- We always start with the largest value first, n^{-1}, where n is our radix, and work our way along using larger negative exponents.

Subtraction Method

- **The calculation to the right is an example of using the subtraction method to convert the decimal 0.8125 to binary.**
  - Our result, reading from top to bottom is:

    $0.8125_{10} = 0.1101_2$
  - Of course, this method works with any base, not just binary.

```
  0.8125
- 0.5000  = 2^-1 × 1
  0.3125
- 0.2500  = 2^-2 × 1
  0.0625
-      0  = 2^-3 × 0
  0.0625
- 0.0625  = 2^-4 × 1
       0
```


Multiplication Method

- **Using the multiplication method to convert the decimal 0.8125 to binary, we multiply by the radix 2.**
  - The first product carries into the units place.
  - Ignoring the value in the units place at each step, continue multiplying each fractional part by the radix.
  - You are finished when the product is zero, or until you have reached the desired number of binary places.
  - Our result, reading from top to bottom is:

    $0.8125_{10} = 0.1101_2$
  - This method also works with any base. Just use the target radix as the multiplier.

```
 .8125
 ×   2
1.6250

 .6250
 ×   2
1.2500

 .2500
 ×   2
0.5000

 .5000
 ×   2
1.0000
```

Hexadecimal Number

- The binary numbering system is the most important radix system for digital computers.
- However, it is difficult to read long strings of binary numbers
  - For example: $11010100011011_2 = 13595_{10}$
- For compactness and ease of reading, binary values are usually expressed using the hexadecimal, or base-16, numbering system.
- The hexadecimal numbering system uses the numerals 0 through 9 and the letters A through F.
  - The decimal number 26 is $1A_{16}$.
- It is easy to convert between base 16 and base 2, because $16 = 2^4$.
- Thus, to convert from binary to hexadecimal, all we need to do is group the binary digits into groups of four.
- Using groups of hextets, the binary number 11010100011011_2 (= 13595_{10}) in hexadecimal is:

0011	0101	0001	1011
3	5	1	B

CSC3501 – S.J. Park

Signed Integer Representation

- There are three ways in which signed binary numbers may be expressed:
  - Signed magnitude,
  - One's complement and
  - Two's complement.
- In an 8-bit word, signed magnitude representation places the absolute value of the number in the 7 bits to the right of the sign bit.
- For example, in 8-bit signed magnitude,
  - positive 3 is: 00000011
  - Negative 3 is: 10000011
- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.

Addition of Signed Magnitude Integer

- Example 1:
  - <u>Using signed magnitude binary</u> arithmetic, find the sum of 75 and 46.
- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.

```
          1 1 1
0     1 0 0 1 0 1 1
0  +  0 1 0 1 1 1 0
0     1 1 1 1 0 0 1
```

- Example2:
  - Using signed magnitude binary arithmetic find the sum of 107 and 46.
- We see that the carry from the seventh bit *overflows* and is discarded, giving us the erroneous result: 107 + 46 = 25.

```
1
      1   1 1 1
0     1 1 0 1 0 1 1
0  +  0 1 0 1 1 1 0
0     0 0 1 1 0 0 1
```


Signed Magnitude Representation

- The signs in signed magnitude representation work just like the signs in pencil and paper arithmetic.
  - Example**:** Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.
  - Because the signs are the same, all we do is add the numbers and <u>supply the negative sign when we are done.</u>

```
            1 1
1     0 1 0 1 1 1 0
1  +  0 0 1 1 0 0 1
1     1 0 0 0 1 1 1
```

- Mixed sign addition (or subtraction) is done the same way.
  - Example**:** Using signed magnitude binary arithmetic, find the sum of 46 and - 25.
  - The sign of the result gets the sign of the number that is larger.
    - Note the “borrows” from the second and sixth bits.

```
        0 2       0 2
0     0 1 0 1 1 1 0
1  +  0 0 1 1 0 0 1
0     0 0 1 0 1 0 1
```

Signed Magnitude: Pro and Cons

- Problems
  - Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.
    - `Subtraction requires an additional hardware different to addition if you have signed magnitude method`
  - Another disadvantage of signed magnitude is that it allows two different representations for zero:
    - positive zero and negative zero.
  - For these reasons (among others) computers systems employ *complement systems* for numeric value representation.
    - Subtraction also use same addition hardware if you use 2's complement

1's Complement

- In complement systems, negative values are represented by some difference between a number and its base.
- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.
- For example, in 8-bit one's complement,
  - positive 3 is: `00000011`
  - Negative 3 is: `11111100`
- In one's complement, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- ***Complement systems are useful because they eliminate the need for different subtraction.***

1's Complement Example

- With one's complement addition, the carry bit is "carried around" and added to the sum.
  - Example**:** Using one's complement binary arithmetic, find the sum of 48 and – 19
- Although the "end carry around" adds some complexity, one's complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- ***Two's complement solves this problem.***

```
1  1 1
   00110000
   11101100
   --------
   00011100
        + 1
   --------
   00011101
```


2's Complement

- To express a value in two's complement:
  - If the number is positive, just convert it to binary and you're done.
  - If the number is negative, find the one's complement of the number and then add 1.
- Example:
  - In 8-bit one's complement, positive 3 is: `00000011`
  - Negative 3 in one's complement is: `11111100`
  - Adding 1 gives us -3 in two's complement form: `11111101`.
- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.
  - Example**:** Using one's complement binary arithmetic, find the sum of 48 and - 19

```
1  1 1
   00110000
 + 11101101
   --------
   00011101
```

Binary Arithmetic

Decimal	1's complement	2's complement
10	00001010	00001010
+ (–3)	11111100	11111101
+7	1 00000110	1 00000111
	carry 1	discarded
	00000111	

2's complement is better because
- only one representation for zero
- simpler addition without subtraction

Overflow Detection of 2's Complement

- While we can't always prevent overflow, we can always *detect* overflow.
- In complement arithmetic, an overflow condition is easy to detect.
- Example:
  - Using two's complement binary arithmetic, find the sum of 107 and 46.
- We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: 107 + 46 = -103.

```
  1 1   1 1 1
  0 1 1 0 1 0 1 1
+ 0 0 1 0 1 1 1 0
  1 0 0 1 1 0 0 1
```

<u>Rule for detecting signed two's complement overflow:</u>
When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred.

Booth Algorithm for Multiplication of 2's Complement Numbers

- One of the many interesting products of this work is Booth's algorithm.
- In most cases, Booth's algorithm carries out multiplication faster and more accurately than naïve pencil-and-paper methods.
- Method
  - In Booth's algorithm, the first 1 in a string of 1s in the multiplier is replaced with a subtraction of the multiplicand.
  - Shift the partial sums until the last 1 of the string is detected.
  - Then add the multiplicand.

```
      0011
    x 0110
    + 0000
   - 0011
  + 0000
 + 0011
   00010010
```


Booth Algorithm

- Idea

Consider a positive multiplier consisting of a block of 1s surrounded by 0s. For example, 00111110. The product is given by :

$$M \times \text{"}0\,0\,1\,1\,1\,1\,1\,0\text{"} = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = M \times 62$$

where M is the multiplicand. The number of operations can be reduced to two by rewriting the same as

$$M \times \text{"}0\,1\,0\,0\,0\,0\text{-}1\,0\text{"} = M \times (2^6 - 2^1) = M \times 62$$

In fact, it can be shown that any sequence of 1's in a binary number can be broken into the difference of two binary numbers:

$$(\ldots 0\overbrace{1\ldots 1}^{n}0\ldots)_2 \equiv (\ldots 1\overbrace{0\ldots 0}^{n}0\ldots)_2 - (\ldots 0\overbrace{0\ldots 1}^{n}0\ldots)_2.$$

- Example 2.24 (p.61)

```
             00110101
 x           01111110
 + 0000000000000000
 + 111111111001011
 + 00000000000000
 + 0000000000000
 + 000000000000
 + 00000000000
 + 0000000000
 + 000110101
 10001101000010110
```

2n bit

Subtract multiplicand
= Add 2's Complement of the multiplicand
= 11001011 then extend sign
= 111111111001011

Add multiplicand

Ignore all bits over 2n.

Increasing Bit Width

- A value can be extended from *N* bits to *M* bits (where *M* > *N*) by using:
  - Sign-extension
  - Zero-extension

Sign-Extension

- Sign bit is copied into most significant bits.
- Number value remains the same.

- **Example 1:**
  - 4-bit representation of 3 = 0011
  - 8-bit sign-extended value: 00000011
- **Example 2:**
  - 4-bit representation of -5 = 1011
  - 8-bit sign-extended value: 11111011

Zero-Extension

- Zeros are copied into most significant bits.
- Number value may change.

- **Example 1:**
  - 4-bit value = 0011
  - 8-bit zero-extended value: 00000011
- **Example 2:**
  - 4-bit value = 1011
  - 8-bit zero-extended value: 00001011