Systems I: Computer Organization and Architecture

Lecture 2: Number Systems and Arithmetic

Number Systems - Base 10

The number system that we use is base 10:

1734 = 1000 + 700 + 30 + 4

= 1x1000 + 7x100 + 3x10 + 4x1

= $1 \times 10^3 + 7 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

724.5 = 7x100 + 2x10 + 4x1 + 5x0.1

= $7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$

Why use base 10?

Number Systems - Base 2

For computers, base 2 is more convenient (why?)

10011_2 = 1x16 + 0x8 + 0x4 + 1x2 + 1x1 = 19_{10}

100010_2 = 1x32 + 0x16 + 0x8 + 0x4 + 1x2 + 0x1 = 34_{10}

101.001_2 = 1x4 + 0x2 + 1x1 + 0x0.5 + 0x0.25 + 1x0.125

= 5.125_{10}

<u>Example</u> - 1101011_2 = ?

10110111_2 = ?

10100.1101_2 = ?

Number Systems - Base 16

Hexadecimal (base 16) numbers are commonly used because it is convert them into binary (base 2) and vice versa.

$8CE_{16}$ = 8x256 + 12x16 + 14x1

= 2048 + 192 + 14

= 2254

3F9 = 3x256 + 15x16 + 9x1

= 768 + 240 + 9 = 1017

Number Systems - Base 16 (continued)

Base 2 is easily converted into base 16:

100011001110_2 = 1000 1100 1110 = 8 C E $_{16}$

11101101110101001_2 = 1 1101 1011 1010 1001 = 1 D B A 9_{16}

$10110001010000010111_2 = ?_{16}$

$1011010100101110 11_2 = ?_{16}$

Number Systems - Base 16 (continued)

Converting base 16 into base 2 works the same way:

$F3A5_{16}$ = 1111 0011 1010 0101_2

$76EF_{16}$ = 0111 0110 1110 1111_2

$AB3D_{16} = ?_2$

$15C.38_{16} = ?_2$

Number Systems – Base 8

Octal (base 8) numbers used to be commonly used because it is convert them into binary (base 2) and vice versa. However, the absence of 8 and 9 is not obvious enough and they were frequently mistaken for decimal values.

$4316_8 = 4 \times 8^3 + 3 \times 8^2 + 1 \times 8^1 + 6 \times 8^0$

$= 4 \times 512 + 3 \times 64 + 1 \times 8 + 6 \times 1$

$= 2048 + 192 + 8 + 6$

$= 2254_{10}$

Number Systems - Base 8 (continued)

Base 2 is easily converted into base 8:

$100011001110_2 = 100\ 011\ 001\ 110 = 4\ 3\ 1\ 6_8$

$11101101110101001_2 = 11\ 101\ 101\ 110\ 101\ 001 = 355651_8$

$10110001010000010111_2 = ?_8$

$1011010100101111011_2 = ?_8$

Number Systems - Base 8 (continued)

Converting base 8 into base 2 works the same way:

$36351_8 = 11\ 110\ 011\ 101\ 001\ 01_2$

$73357_8 = 111\ 011\ 011\ 101\ 111_2$

$2436_8 = ?_2$

$1573_8 = ?_2$

Converting From Decimal to Binary

19
9 R 1
4 R 1
2 R 0
1 R 0
0 R 1

10011_2

Converting From Decimal to Hexadecimal

16 | 237

14 R 13

0 R 14

$\rightarrow ED_{16}$

Converting From Decimal to Octal

8 | 237

29 R 5

3 R 5

0 R 3

$\rightarrow 355_{8}$

Binary, Octal, Decimal and Hexadecimal Equivalents

Binary	Decimal	Octal	Hex.	Binary	Decimal	Octal	Hex.
0000	0	0	0	1000	8	10	8
0001	1	1	1	1001	9	11	9
0010	2	2	2	1010	10	12	A
0011	3	3	3	1011	11	13	B
0100	4	4	4	1100	12	14	C
0101	5	5	5	1101	13	15	D
0110	6	6	6	1110	14	16	E
0111	7	7	7	1111	15	17	F

Addition of Binary Numbers

C		*101111000*
X	190	10111110
Y	+ 141	+ 10001101
X+Y	331	101001011

C		*011111110*
X	127	01111111
Y	+ 63	+ 00111111
X+Y	190	10111110

Addition of Binary Numbers (continued)

C		*001011000*
X	174	10101101
Y	+ 44	+ 00101100
X+Y	217	11011001

C		*000000000*
X	170	10101010
Y	+ 85	+ 01010101
X+Y	255	11111111

Addition of Hexadecimal Numbers

C	*1100*	*1*	*1*	*0*	*0*
X	$19B9_{16}$	1	9	11	9
Y	$C7E6_{16}$	12	7	14	6
X+Y	$E19F_{16}$	14	17	25	15
		14	16+1	16+9	15
		E	1	9	F

Complements

There are several different ways in which we can represent negative numbers:

- Signed-Magnitude Representation
- 1s Complement Representation
- 2s Complement Representation

Signed-Magnitude Representation

- In signed-magnitude representation, the sign bit is set to '1' if negative and cleared to '0' if positive:

6	00000110
+13	00001101
19	00010011
-6	10000110
+ +13	00001101
+ 7	10010011 ***(= -19)***

1s Complement Representation

- In signed-magnitude representation, the sign bit is set to ‘1’ if negative and the other bits are also reversed.

6	00000110
+13	00001101
19	00010011

-6		11111001	
+ +13		00001101	
+ 7	1	00000110	***(= +6)***
	overflow bit		

2s Complement Representation

- In 2s complement representation, we subtract the absolute value from 2^n:

100000000
00000110
11111010

-6		11111010	
+ +13		00001101	
+ 7	1	00000111	***(= +7)***

2s Complement Representation (continued)

- The 2s complement representation can also be found by reversing the bits (into 1s complement) and then adding 1:

```
6 => 00000110 =>11111001
              +        1
               11111010

43 => 00101011 => 11010100
                +        1
                  11010101
```

Overflow

- If an addition operation produces a result that exceeds our number system's range, ***overflow*** has occurred.
- Addition of two numbers of the same sign produces overflow; addition two numbers of opposite sign cannot cause overflow.

-3	1101	+5	0101
+6	0110	+6	0110
+3	1 0011 = ***+3***	+11	1011 = ***-5***
-8	1000	+7	0111
-8	1000	+7	0111
-16	1 0000 = ***0***	+14	1110 = -2

Subtraction

- Subtraction works in a similar fashion, but the borrow (an initial carry bit) is a '1':

			1	← *initial carry*
	+4	0100	0100	
-	+3	- 0011	+1100	
	+1		1 0001	

			1
	+3	0011	0011
-	+4	- 0100	+1011
	- 1		1 0001

Subtraction (continued)

			1
	+3	0011	0011
-	- 4	- 1100	+0011
	- 7		0111

			1
	-3	1011	1011
-	-4	- 1100	+0011
	-1		1111

Binary Multiplication

- Multiplication is repeated addition of the multiplicand, where the number of additions depends on the multiplier:

```
     11          1011   <- multiplicand
  x  13          1101   <- multiplier
  -----          ----
     33          1011   <- shifted multiplicands
    11          0000
  -----        1011
    143       1011
              --------
              10001111  <- product
```

Partial Product Method

- It is more convenient to add each shifted multiplicand as it is created to a ***partial product***:

```
        1011                 11
      x 1101          x      13
      ------               ----
         0000                143
        1011
       -----
       01011
       0000
      ------
      001011
      1011
     -------
     0110111
     1011
    --------
    10001111
```

partial products: 0000, 01011, 001011, 0110111

shifted multiplicands: 1011, 0000, 1011, 1011

Partial Product Method – Another Example

```
   -5            1011
x  -3            1101
                -----
                00000      shifted multiplicand
                11011
               ------
               111011
               00000
              -------
              1111011
              11011        shifted and negated multiplicand
             --------
             11100111
             00101
             --------
             00001111
```

Binary Division

```
            10011
      ┌─────────
 1011 │ 11011001
        1011          shifted divisor
        ----
         0101         reduced divisor
         0000
         ----
          1010
          0000
          ----
          10100
           1011
          -----
           10011
            1011
           -----
            1000      remainder
```

shifted divisor

reduced divisor

remainder

Binary Representation of Decimal Numbers

Decimal Digit	**BCD (8421)**	**2421**	**Excess-3**
0	0000	0000	0011
1	0001	0001	0100
2	0010	0010	0101
3	0011	0011	0110
4	0100	0100	0111
5	0101	1011	1000
6	0110	1100	1001
7	0111	1101	1010
8	1000	1110	1011
9	1001	1111	1100

Floating Point Representations

- The floating point representation of a number has two part: fraction and an exponent:

  $+6132.789 = +0.6132789 \times 10^4$
- In general, a number can be expressed as

  $m \times r^e$

  where m is the mantissa, r is the radix and e is the exponent.
- Because we know that computer always use binary numbers (radix = 2), only m and e need to be represented.
- Therefore, we can represent 1001.11 using m = 01001110 and e = 000100 (because $1011.11 = +0.101111 \times 2^{+4}$)

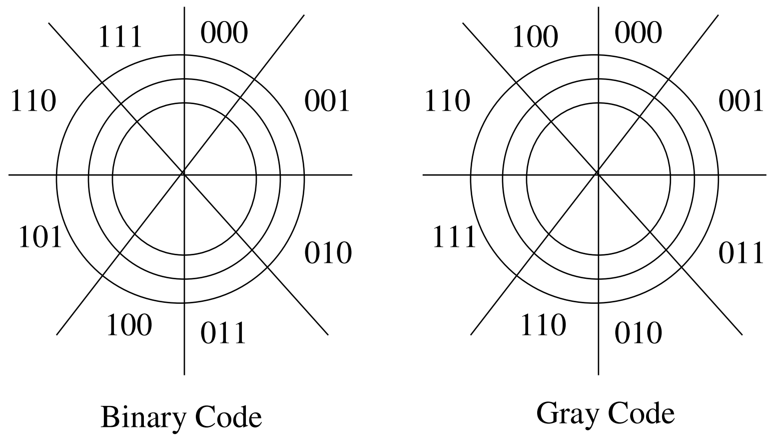
Floating Point Representations (continued)

- A floating-point number is ***normalized*** if the most significant place in the mantissa is nonzero.
  - 350 is normalized
  - 00035 is not normalized.
  - 00011010 is not normalized, but 11010000 is normalized; this requires changing the exponent to 4.
- The standard method of storing exponent is excess-64, where
  - an exponent of 1000000 is zero
  - an exponent of 1000011 is positive
  - an exponent of 0110100 is negative.

Gray Codes

- Sometimes electromechanical applications of digital systems (machine tools, automotive brake systems and copiers) require a digital value that indicates a mechanical position.
- A standard binary code may see more than one bit change from one position to another, which could lead to an incorrect reading if mechanical assembly is imperfect.

Binary Code vs. Gray Code



ASCII representation of characters

- ASCII (***A***merican ***S***tandard ***C***ode for ***I***nformation ***I***nterchange) is a numeric code used to represent characters.
- All characters are represented this way including:
  - words (character strings)
  - numbers
  - punctuation
  - control characters
- There are separate values for upper case and lower case characters:

A	65	z	122
B	66	*blank*	32
Z	90	$	52
a	97	0	48
b	98	9	57

Control Codes

- ASCII (a 7-bit code) has $2^7 = 128$ values.
- We only need 62 for alphanumeric characters. Even after accounting for common punctuation, there are far more available code values than we need. What do we use them for?
- Control codes include DEL (for delete), NUL (for null). STX (Start of Text), CR (for carriage return), etc.

Error Detection Codes

- An error is a corruption of the data from its correct state.
- There are several codes that allow use to detect an error. These include:
  - Parity
  - CRC
  - Checksum

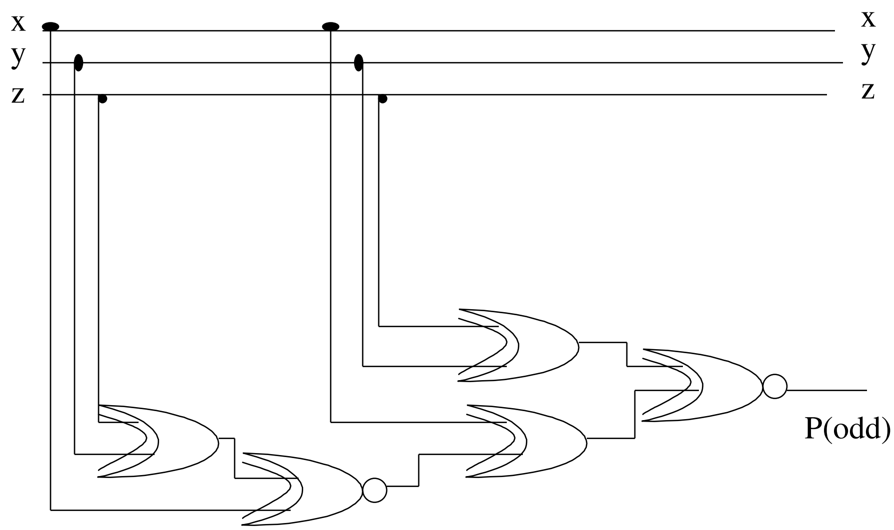
Parity

- Parity is an extra bit appended to our data which indicates whether the data bits add up to an even (for even parity) or odd (for odd parity) value.

Parity Generation

Message (xyz)	**P(odd)**	**P(even)**
000	1	0
001	0	1
010	1	0
011	0	1
100	1	0
101	0	1
110	1	0
111	0	1

Odd Parity



CRC

- CRC (***C***yclic ***R***edundancy ***C***heck) – is an error detecting code.
- CRC can spot single-bit errors as well as clustered error.

Checksum

- Checksum codes involve adding bytes modulo 256.
- This allows checksums to spot one-byte errors.
- Checksums can use other modulos which would allow for spotting different errors as well.