University of Southern California

Viterbi School of Engineering

EE352

Computer Organization and Architecture

Fixed Point Systems

References:

1) Textbook
2) Mark Redekopp’s slide series

Shahin Nazarian

Spring 2010

Data Representation

- In C/C++ variables can be of different types and sizes
  - Integer Types (signed and unsigned)

C Type	Bytes	Bits	MIPS Name
[unsigned] char	1	8	byte
[unsigned] short [int]	2	16	half-word
[unsigned] long [int]	4	32	word
[unsigned] long long [int]	8	64	double-word

  - Floating Point Types

C Type	Bytes	Bits	MIPS Name
float	4	32	single
double	8	64	double

Binary Representation Systems

- **Integer Systems**
  - Unsigned
    - Unsigned (Normal) binary
  - Signed
    - Signed Magnitude
    - 2's complement
    - *Excess-N**
    - *1's complement**
- **Floating Point**
  - For very large and small (fractional) numbers
- **Codes**
  - Text
    - ASCII / Unicode
  - Decimal Codes
    - BCD (Binary Coded Decimal) / (8421 Code)

*** = Not fully covered in this class**

Number Systems

- Number systems consist of
  1. A base (radix) r
  2. r coefficients [0 to r-1]
- Human System: Decimal (Base 10): 0,1,2,3,4,5,6,7,8,9
- Computer System: Binary (Base 2): 0,1
- Human systems for working with computer systems (shorthand for human to read/write binary)
  - Octal (Base 8): 0,1,2,3,4,5,6,7
  - Hexadecimal (Base 16): 0-9,A,B,C,D,E,F (A thru F = 10 thru 15)

Anatomy of a Decimal Number

- A number consists of a string of explicit coefficients (digits)
- Each coefficient has an implicit place value which is a power of the base
- The value of a decimal number (a string of decimal coefficients) is the sum of each coefficient times its place value

Radix (base)

$$(934)_{10} = 9*10^2 + 3*10^1 + 4*10^0 = 934$$

Explicit coefficients

Implicit place values

$$(3.52)_{10} = 3*10^0 + 5*10^{-1} + 2*10^{-2} = 3.52$$

Positional Number Systems (Unsigned)

- A number in base r has place values/weights that are the powers of the base
- Denote the coefficients as: a_i

Left-most digit = Most Significant Digit (MSD)

Right-most digit = Least Significant Digit (LSD)

$$\ldots \quad \frac{a_3}{r^3} \quad \frac{a_2}{r^2} \quad \frac{a_1}{r^1} \quad \frac{a_0}{r^0} \; . \; \frac{a_{-1}}{r^{-1}} \quad \frac{a_{-2}}{r^{-2}} \quad \ldots$$

$$N_r = \Sigma_i(a_i * r^i) = D_{10}$$

Examples

$$(746)_8 = 7*8^2 + 4*8^1 + 6*8^0$$
$$= 448 + 32 + 16 = 486_{10}$$

$$(1A5)_{16} = 1*16^2 + 10*16^1 + 5*16^0$$
$$= 256 + 160 + 5 = 421_{10}$$

Anatomy of a Binary Number

- Same as decimal but now the coefficients are 1 and 0 and the place values are the powers of 2

Most Significant Digit (MSB)

Least Significant Bit (LSB)

$$(1011)_2 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$$

radix (base)

coefficients

place values = powers of 2

Binary Examples

$$(\underset{8}{\underline{1}}\underline{0}\underline{0}\underset{1}{\underline{1}}.\underset{.5}{\underline{1}})_2 = 8 + 1 + 0.5 = 9.5_{10}$$

$$(\underset{128}{\underline{1}}0\underset{32}{\underline{1}}\underset{16}{\underline{1}}000\underset{1}{\underline{1}})_2 = 128 + 32 + 16 + 1 = 177_{10}$$

Powers of 2

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

$2^6 = 64$

$2^7 = 128$

$2^8 = 256$

$2^9 = 512$

$2^{10} = 1024$

___	___	___	___	___	___	___	___	___	___	___
1024	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**

Conversion to Decimal Examples

- Decimal equivalent is…

  … the sum of each coefficient multiplied by its implicit place value (power of the base)

  $= \Sigma_i(a_i * r^i)$ [a_i = coefficient, r = base]

$(11010)_2 = 1*2^4 + 1*2^3 + 1*2^1$
$= 16 + 8 + 2 = (26)_{10}$

$(6523)_8 = 6*8^3 + 5*8^2 + 2*8^1 + 3*8^0$
$= 3072 + 320 + 16 + 3 = (3411)_{10}$

$(AD2)_{16} = 10*16^2 + 13*16^1 + 2*16^0$
$= 2560 + 208 + 2 = (2770)_{10}$

Unique Combinations

- Given *n* digits of base *r*, how many unique numbers can be formed? r^n
  - What is the range? $[0 \text{ to } r^n-1]$

2-digit, decimal numbers (r=10, n=2)					____ 0-9	____ 0-9	100 combinations: 00-99
3-digit, decimal numbers (r=10, n=3)				____	____	____	1000 combinations: 000-999
4-bit, binary numbers (r=2, n=4)			____ 0-1	____ 0-1	____ 0-1	____ 0-1	16 combinations: 0000-1111
6-bit, binary numbers (r=2, n=6)	____	____	____	____	____	____	64 combinations: 000000-111111

Main Point: Given n digits of base r, r^n unique numbers can be made with the range $[0 - (r^n-1)]$

Approximating Large Powers of 2

- Often need to find decimal approximation of a large powers of 2 like 2^{16}, 2^{32}, etc.
- Use following approximations:
  - $2^{10} \approx 10^3$ (1 thousand) = 1 Kilo-
  - $2^{20} \approx 10^6$ (1 million) = 1 Mega-
  - $2^{30} \approx 10^9$ (1 billion) = 1 Giga-
  - $2^{40} \approx 10^{12}$ (1 trillion) = 1 Tera-
- For other powers of 2, decompose into product of 2^{10} or 2^{20} or 2^{30} and a power of 2 that is less than 2^{10}
  - 16-bit half word: 64K numbers
  - 32-bit word: 4G numbers
  - 64-bit dword: 16 million trillion numbers

$2^{16} = 2^6 * 2^{10}$
$\approx 64 * 10^3 = 64{,}000$

$2^{24} = 2^4 * 2^{20}$
$\approx 16 * 10^6 = 16{,}000{,}000$

$2^{28} = 2^8 * 2^{20}$
$\approx 256 * 10^6 = 256{,}000{,}000$

$2^{32} = 2^2 * 2^{30}$
$\approx 4 * 10^9 = 4{,}000{,}000{,}000$

Decimal to Unsigned Binary

- To convert a decimal number, *x*, to binary:
  - Only coefficients of 1 or 0. So simply find place values that add up to the desired values, starting with larger place values and proceeding to smaller values and place a 1 in those place values and 0 in all others

25_{10} =	0	1	1	0	0	1
	32	16	8	4	2	1

For 25_{10} the place value 32 is too large to include so we include 16. Including 16 means we have to make 9 left over. Include 8 and 1

Decimal to Unsigned Binary

73_{10}=	0	1	0	0	1	0	0	1
	128	64	32	16	8	4	2	1
87_{10}=	0	1	0	1	0	1	1	1
145_{10}=	1	0	0	1	0	0	0	1

0.625_{10}=	1	0	1	0	0
	.5	.25	.125	.0625	.03125

Decimal to Another Base

- To convert a decimal number, *x*, to base r:
  - Use the place values of base r (powers of r)
  - Starting with largest place values, fill in coefficients that sum up to desired decimal value without going over

$$75_{10} = \underset{256}{\underline{\ 0\ }}\quad \underset{16}{\underline{\ 4\ }}\quad \underset{1}{\underline{\ B\ }}$$

Conversion Methods Summary

- Base r => Base 10
  - Sum of coefficients * place values
- Base 10 => Base r
  - Find place values and coefficients that add up to desired value

Binary, Octal, and Hexadecimal

- Octal (base 8 = 2^3)
- 1 Octal digit $(\_)_8$ can represent: 0 – 7
- 3 bits of binary $(\_ \_ \_)_2$ can represent: 000-111 = 0 – 7
- Conclusion…
  1 Octal digit = 3 bits

- Hex (base 16=2^4)
- 1 Hex digit $(\_)_{16}$ can represent: 0-F (0-15)
- 4 bits of binary $(\_ \_ \_ \_)_2$ can represent: 0000-1111= 0-15
- Conclusion…
  1 Hex digit = 4 bits

Binary to Octal or Hex

- Make groups of 3 bits starting from radix point and working outward
- Add 0's where necessary
- Convert each group of 3 to an octal digit

$$\underbrace{101}_{5}\,\underbrace{001}_{1}\,\underbrace{110}_{6}.\underbrace{110}_{6}$$

$$516.6_8$$

- Make groups of 4 bits starting from radix point and working outward
- Add 0's where necessary
- Convert each group of 4 to an octal digit

$$\underbrace{0001}_{1}\,\underbrace{0100}_{4}\,\underbrace{1110}_{E}.\underbrace{1100}_{C}$$

$$14E.C_{16}$$

Octal or Hex to Binary

- Expand each octal digit to a group of 3 bits

317.2_8

011001111.010_2

11001111.01_2

- Expand each hex digit to a group of 4 bits

$D93.8_{16}$

110110010011.1000_2

110110010011.1_2

Hexadecimal Representation

- Since values in modern computers are many bits, we use hexadecimal as a shorthand notation (4 bits = 1 hex digit)
  - 11010010 = D2 hex
  - 0111011011001011 = 76CB hex
- Important Point: To interpret the value of a hex number, you must know what underlying binary system is assumed (unsigned, 2's comp. etc.)

Conversion Methods

- Base r => Base 10
  - Sum of coefficients * place values
- Base 10 => Base r
  - Find place values and coefficients that add up to desired value
- Binary ⇔ Octal
  - 3-bits = 1 octal digit
- Binary ⇔ Hex
  - 4-bits = 1 hex digit

Binary Arithmetic

- Can perform all arithmetic operations (+,-,*,÷) on binary numbers
- Can use same methods as in decimal
  - Still use carries and borrows, etc.
  - Only now we carry when sum is 2 or more rather than 10 or more (decimal)
  - We borrow 2's not 10's from other columns
- Easiest method is to add bits in your head in decimal (1+1 = 2) then convert the answer to binary ($2_{10} = 10_2$)

Binary Addition

- In decimal addition we carry when the sum is 10 or more
- In binary addition we carry when the sum is 2 or more
- Add bits in binary to produce a sum bit and a carry bit

			1
0	0	1	1
+ 0	+ 1	+ 0	+ 1
00	01	01	10
no need to carry / sum bit	no need to carry / sum bit	no need to carry / sum bit	carry 1 into next column of bits / sum bit

Binary Addition & Subtraction

$$
\begin{array}{rccccl}
 & 1 & 1 & 1 & & \\
 & 0 & 1 & 1 & 1 & (7) \\
+ & 0 & 0 & 1 & 1 & (3) \\
\hline
 & 1 & 0 & 1 & 0 & (10) \\
 & 8 & 4 & 2 & 1 &
\end{array}
\qquad\qquad
\begin{array}{rccccl}
 & 0 & & 0 & & \\
 & \cancel{1} & {}^{1}0 & \cancel{1} & {}^{1}0 & (10) \\
- & 0 & 1 & 0 & 1 & (5) \\
\hline
 & 0 & 1 & 0 & 1 & (5) \\
 & 8 & 4 & 2 & 1 &
\end{array}
$$

Binary Addition & Subtraction

```
  1 1 1
  0 1 1 1 (7)
+ 0 0 1 1 (3)
  -------
  1 0 1 0 (10)
  8 4 2 1
```

0 0

~~1~~ 1 0 ~~1~~ 1 0 (10)

```
- 0 1 0 1  (5)
  -------
  0 1 0 1  (5)
  8 4 2 1
```

Binary Addition

```
    110
    0110(6)
+   0111(7)
-----------
    1101(13)
    8 4 2 1
```

Binary Addition

```
  0
 0110(6)          0
+ 0111(7)       + 1
---------       ---
 1101(13)        01
```

carry bit   sum bit

Binary Addition

```
   10                 0
  0110 (6)            1
+ 0111 (7)          + 1
------              ---
  1101 (13)          10
```

carry bit   sum bit

Binary Addition

```
   110                1
   0110 (6)           1
+  0111 (7)         + 1
   ----             ---
   1101 (13)         11
```

carry bit — sum bit

$1+1+1 = 3_{10} = 11_2$

Binary Addition

```
   110
   0110(6)
 + 0111(7)
 ---------
   1101(13)
```

```
   1
   0
 + 0
 ----
  01
```

carry bit   sum bit

Binary Subtraction

- If you can't perform subtraction in one column borrow from higher order columns
- When you borrow you are borrowing a 2, not a 10 as in decimal

```
    1 0 1 0 (10)
-   0 1 1 1 (7)
-----------
    0 0 1 1 (3)
```

Binary Subtraction

```
    1 0 1 0 (10)
-   0 1 1 1 (7)
-----------
    0 0 1 1 (3)
```

Can't perform 0 – 1, so we must borrow

Binary Subtraction

$$
\begin{array}{rcccccl}
 & & & \color{red}{0} & & & \\
 & 1 & 0 & \cancel{1} & \color{red}{1}0 & & (10) \\
- & 0 & 1 & 1 & 1 & & (7) \\
\hline
 & 0 & 0 & 1 & 1 & & (3)
\end{array}
$$

We borrow the 1 (which is worth 2) from the next column and now we can perform 10-1 = 1

And now we go to the next column

Binary Subtraction

```
         0
    1 0 1̸10 (10)
-   0 1 1 1 (7)
-----------
    0 0 1 1 (3)
```

Can’t perform 0 – 1, so we must borrow

Binary Subtraction

We get the borrow from this column and work back to the current column

$$
\begin{array}{rl}
 & 0\ \ 1\ 10 \\
 & \cancel{1}1\cancel{0}\ \cancel{1}1 0 & (10) \\
- & 0\ \ 1\ \ 1\ \ 1 & (7) \\
\hline
 & 0\ \ 0\ \ 1\ \ 1 & (3)
\end{array}
$$

We can't borrow from the column next to us (it is 0 as well) so we must try to borrow from the next column and then work our way back to the current column where we perform 10 – 1 = 1

Binary Subtraction

```
      0  1 10
      1 1 0 1 1 0 (10)
  -   0   1   1   1 (7)
  ------------------
      0   0   1   1 (3)
```

We can perform 1 - 1 = 0

Binary Subtraction

$$
\begin{array}{rl}
 & 0\ \ 1\ \ 10 \\
 & \cancel{1}\ \ \cancel{0}\ \ \cancel{1}\ 10\ (10) \\
- & 0\ \ 1\ \ 1\ \ 1\ (7) \\
\hline
 & 0\ \ 0\ \ 1\ \ 1\ (3)
\end{array}
$$

We can perform 0 - 0 = 0

Binary Multiplication

- Like decimal multiplication, find each partial product and shift them, then sum them up
- Multiplying two *n*-bit numbers yields at most a *2*n*-bit product

```
        0 1 1 0(6)
     *  0 1 0 1(5)
     ----------
        0 1 1 0
      0 0 0 0
    0 1 1 0
+ 0 0 0 0
----------------
  0 0 1 1 1 1 0  <-- Sum of the partial products
```

Binary Multiplication

```
      0 1 1 0(6)
*     0 1 0 1(5)
  -------------
      0 1 1 0
```

First partial product

Binary Multiplication

```
      0 1 1 0(6)
*     0 1 0 1(5)
   -----------
      0 1 1 0
    0 0 0 0
```

Second partial product

Binary Multiplication

```
        0 1 1 0(6)
*       0 1 0 1(5)
----------------
        0 1 1 0
      0 0 0 0
    0 1 1 0
```

Third partial product

Binary Multiplication

```
          0 1 1 0(6)
     *    0 1 0 1(5)
     ____________
          0 1 1 0
        0 0 0 0
      0 1 1 0
    0 0 0 0
```

Fourth partial product

Binary Multiplication

```
           0 1 1 0(6)
       *   0 1 0 1(5)
      -------------
           0 1 1 0
         0 0 0 0
       0 1 1 0
   + 0 0 0 0
   -------------------
     0 0 1 1 1 1 0
```

Sum up the partial products

Binary Division

- Use the same long division techniques as in decimal

```
                0 1 0 1 r.1   (5 r.1)10
              ---------
(2)10    10  | 1 0 1 1        (11)10
              -1 0
             -----
                 0 1
                -0 0
                -----
                   1 1
                  -1 0
                  -----
                   0 1
```

Binary Division

10 (2) goes into 1, 0 times. Since it doesn't, bring in the next bit

```
      0
   ________
10 | 1 0 1 1
```

Binary Division

10 (2) goes into 10, 1 time. Multiply, subtract, and bring down the next bit

```
      0 1
    ________
10 | 1 0 1 1
    -1 0 ↓
   ______
       0 1
```

Binary Division

10 (2) goes into 01, 0 times. Multiply, subtract, and bring down the next bit

```
      0 1 0
    ________
10 | 1 0 1 1
    -1 0
   -----
       0 1
      -0 0
      -----
         1 1
```

Binary Division

10 (2) goes into 11, 1 time. Multiply and subtract. The remainder is 1

```
    0 1 0 1 r.1
   _________
10 |1 0 1 1
   -1 0
  ------
      0 1
     -0 0
    ------
        1 1
       -1 0
      ------
        0 1
```

Hexadecimal Arithmetic

- **Same style of operations**
  - **Carry when sum is 16 or more, etc.**

$$
\begin{array}{rrr}
1 & 1 & \\
 & 4 & D_{16} \\
+ & B & 5_{16} \\
\hline
1 & 0 & 2_{16}
\end{array}
$$

$$13+5 = 18_{10} = \underset{16}{\underline{1}}\;\underset{1}{\underline{2}}_{16}$$

$$1+4+11 = 16_{10} = \underset{16}{\underline{1}}\;\underset{1}{\underline{0}}_{16}$$

One More Example

$$\begin{array}{rccccl} & & 0 & & & \\ & 0 & \cancel{1} & {}^{1}0 & 0 & (4) \\ - & 0 & 1 & 1 & 0 & (6) \\ \hline & & & & & (-2) \end{array}$$

- **What about negative numbers?**
  - **Current unsigned binary system can only represent positive numbers**
  - **Given n-bits => Range: [0 to 2^n-1]**

Signed Magnitude

2's Complement System

SIGNED SYSTEMS

Binary Representation Systems

- Integer Systems
  - Unsigned
    - Unsigned (Normal) binary
  - Signed
    - Signed Magnitude
    - 2's complement
    - *1's complement**
    - *Excess-N**
- Floating Point
  - For very large and small (fractional) numbers
- Codes
  - Text
    - ASCII / Unicode
  - Decimal Codes
    - BCD (Binary Coded Decimal) / (8421 Code)

*** = Not covered in this class**

Unsigned and Signed

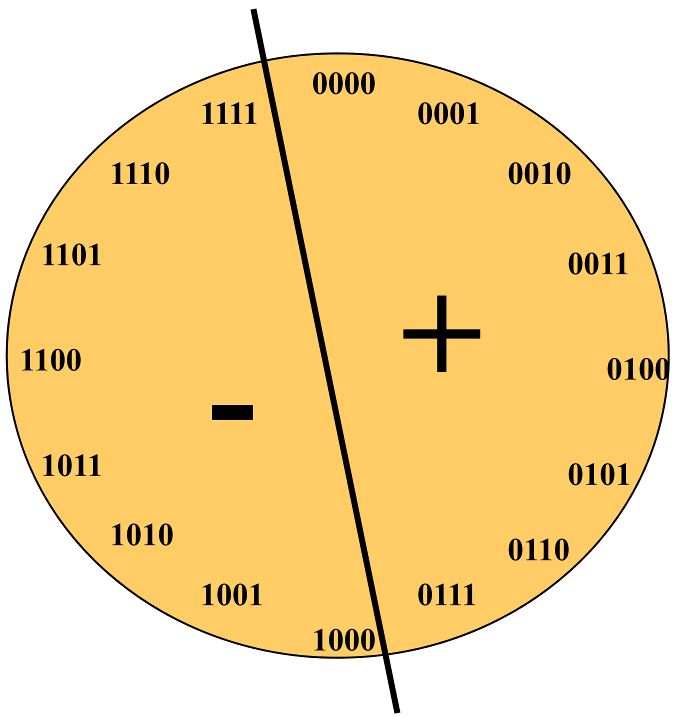
- Normal (unsigned) binary can only represent positive numbers
  - All place values are positive
- To represent negative numbers we must use a modified binary representation that takes into account sign (pos. or neg.)
  - We call these *signed* representations

Signed Number Representation

- 2 Primary Systems
  - Signed Magnitude
  - *Two's Complement (most widely used for integer representation)*

Signed numbers

- All systems used to represent negative numbers split the possible binary combinations in half (half for positive numbers / half for negative numbers)
- In both signed magnitude and 2's complement, positive and negative numbers are separated using the MSB
  - MSB=1 means negative
  - MSB=0 means positive



Signed Magnitude System

- Use binary place values but now MSB represents the sign (1 if negative, 0 if positive)

4-bit Unsigned

Bit 3	Bit 2	Bit 1	Bit 0
____	____	____	____
8	4	2	1

0 to 15

4-bit Signed Magnitude

Bit 3	Bit 2	Bit 1	Bit 0
____	____	____	____
+/-	4	2	1

-7 to +7

8-bit Signed Magnitude

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
____	____	____	____	____	____	____	____
+/-	64	32	16	8	4	2	1

-127 to +127

Signed Magnitude Examples

4-bit Signed Magnitude

1	**1**	**0**	**1**	**= -5**
+/-	4	2	1	

0	**0**	**1**	**1**	**= +3**
+/-	4	2	1	

Notice that +3 in signed magnitude is the same as in the unsigned system

1	**1**	**1**	**1**	**= -7**
+/-	4	2	1	

8-bit Signed Magnitude

1	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**= -19**
+/-	64	32	16	8	4	2	1	

0	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**= +25**
+/-	64	32	16	8	4	2	1	

Important: Positive numbers have the same representation in signed magnitude as in normal unsigned binary

Signed Magnitude Range

- Given n bits…
  - MSB is sign
  - Other n-1 bits = normal unsigned place values
    - Range with n-1 unsigned bits = [0 to $2^{n-1}-1$]

Range with n-bits of Signed Magnitude

$$[\ -2^{n-1}-1 \text{ to } +2^{n-1}-1]$$

Disadvantages of Signed Magnitude

1. Wastes a combination to represent -0

$$0000 = 1000 = 0_{10}$$

2. Addition and subtraction algorithms for signed magnitude are different than unsigned binary (we'd like them to be the same to use same HW)

```
   4          6
 - 6        - 4
 ---        ---
Swap & make
res. negative  -
```

2's Complement System

- Normal binary place values except MSB has negative weight
  - MSB of 1 = -2^{n-1}

4-bit Unsigned

Bit 3	Bit 2	Bit 1	Bit 0
____	____	____	____
8	4	2	1

→ **0 to 15**

4-bit 2's complement

Bit 3	Bit 2	Bit 1	Bit 0
____	____	____	____
-8	4	2	1

→ **-8 to +7**

8-bit 2's complement

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
____	____	____	____	____	____	____	____
-128	64	32	16	8	4	2	1

→ **-128 to +127**

2's Complement Examples

4-bit 2's complement

1	0	1	1	= -5
-8	4	2	1	

0	0	1	1	= +3
-8	4	2	1	

1	1	1	1	= -1
-8	4	2	1	

Notice that +3 in 2's comp. is the same as in the unsigned system

8-bit 2's complement

1	0	0	0	0	0	0	1	= -127
-128	64	32	16	8	4	2	1	

0	0	0	1	1	0	0	1	= +25
-128	64	32	16	8	4	2	1	

Important: Positive numbers have the same representation in 2's complement as in normal unsigned binary

2's Complement Range

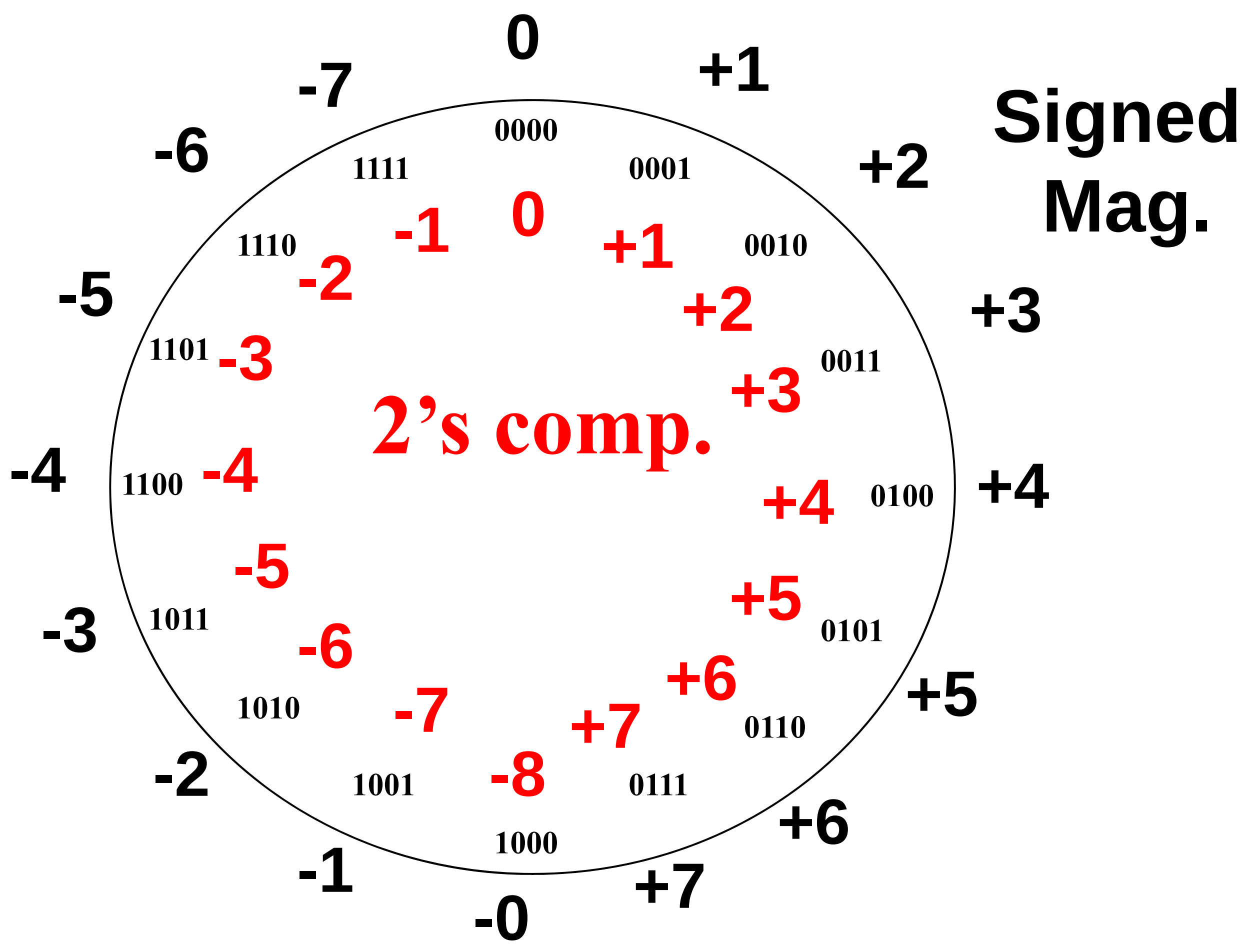
- Given n bits…
  - Max positive value = 011…11
    - Includes all n-1 positive place values
  - Max negative value = 100…00
    - Includes only the negative MSB place value

Range with n-bits of 2's complement

$$[\ -2^{n-1} \text{ to } +2^{n-1}-1]$$

- Side note – What decimal value is 111…11?
  - -1_{10}

Comparison of Systems



Taking the Negative

- Given a number in signed magnitude or 2's complement how do we find its negative (i.e. -1 * X)
  - Signed Magnitude: Flip the sign bit
    - 0110 = +6 => 1110 = -6
  - 2's complement: "Take the 2's complement"
    - 0110 = +6 => -6 = 1010
    - Operation defined as:
      1. Flip/invert/not all the bits (1's complement)
      2. Add 1 and drop any carry (i.e. finish with the same # of bits as we start with)

Taking the 2's Complement

- Invert (flip) each bit (take the 1's complement)
  - 1's become 0's
  - 0's become 1's
- Add 1 (drop final carry-out, if any)

```
-32 16  8  4  2  1
  0  1  0  0  1  1      Original number = +19

  1  0  1  1  0  0      Bit flip is called the 1's complement of a number
+                1
------------------
  1  0  1  1  0  1      Resulting number = -19
```

Important: Taking the 2's complement is equivalent to taking the negative (negating)

Taking the 2's Complement

1

-32	16	8	4	2	1
1	0	1	0	1	0

```
 101010      Original number = -22

 010101      Take the 2's complement yields the negative of a number
+     1
 ------
 010110      Resulting number = +22

 101001      Taking the 2's complement again yields the original number (the operation is symmetric)
+     1
 ------
 101010      Back to original = -22
```

2

```
 0000        Original # = 0

 1111        Take the 2's complement
+   1
 ----
 0000        2's comp. of 0 is 0
```

3

```
 1000        Original # = -8

 0111        Take the 2's complement
+   1
 ----
 1000        Negative of -8 is -8
```

Negative of -8 is -8

(i.e. no positive equivalent, but this is not a huge problem)

2's Complement System Facts

- Normal binary place values but MSB has negative weight
- MSB determines sign of the number
  - 0 = positive / 1 = negative
- Special Numbers
  - 0 = All 0's (00…00)
  - -1 = All 1's (11…11)
  - Max Positive = 0 followed by all 1's (011..11)
  - Max Negative = 1 followed by all 0's (100…00)
- To take the negative of a number (e.g. -7 => +7 or +2 => -2), requires taking the complement
  - 2's complement of a # is found by flipping bits and adding 1

1001	x = -7
0110	Bit flip (1's comp.)
+ 1	Add 1
0111	-x = -(-7) = +7

Summary

- Signed Magnitude
  - Range ($-2^{n-1} - 1$ to $+2^{n-1} - 1$)
- 2's Complement
  - Range (-2^{n-1} to $+2^{n-1} - 1$)

Unsigned and Signed Variables

- Unsigned variables use unsigned binary (normal power-of-2 place values) to represent numbers

1	0	0	1	0	0	1	1	= +147
128	64	32	16	8	4	2	1	

- Signed variables use the 2's complement system (Neg. MSB weight) to represent numbers

1	0	0	1	0	0	1	1	= -109
-128	64	32	16	8	4	2	1	

Zero and Sign Extension

- Extension is the process of increasing the number of bits used to represent a number without changing its value

Unsigned = Zero Extension (Always add leading 0's):

111011 = 00111011

Increase a 6-bit number to 8-bit number by zero extending

2's complement = Sign Extension (Replicate sign bit):

pos. 011010 = 00011010

neg. 110011 = 11110011

Sign bit is just repeated as many times as necessary

Zero and Sign Truncation

- Truncation is the process of decreasing the number of bits used to represent a number without changing its value

Unsigned = Zero Truncation (Remove leading 0's):

$\cancel{00}111011 = 111011$

Decrease an 8-bit number to 6-bit number by truncating 0's. Can't remove a '1' because value is changed

2's complement = Sign Truncation (Remove copies of sign bit):

pos. $\cancel{00}011010 = 011010$

neg. $\cancel{111}10011 = 10011$

Any copies of the MSB can be removed without changing the numbers value. Be careful not to change the sign by cutting off ALL the sign bits

Translating Hexadecimal

- Hex place values (16^2, 16^1, 16^0) can ONLY be used if the number is positive
- If hex represents unsigned binary
  1. Apply hex place values
     - B2 hex = $11*16^1 + 2*16^0 = 178_{10}$
- If hex represents signed value (2's comp.)
  1. Determine the sign by looking at the underlying MSB
  2. If pos., apply hex place values (as if it were unsigned)
  3. If neg., take the 16's complement and apply hex place values to find the neg. number's magnitude

Translating Hexadecimal

- Given 6C hex
  - If it is unsigned, apply hex place values

    $6C\ hex = 6*16^1 + 12*16^0 = 108_{10}$
  - If it is signed…
    - Determine the sign by looking at MSD
      - 0-7 hex has a 0 in the MSB [i.e. positive]
      - 8-F hex has a 1 in the MSB [i.e. negative]
      - Thus, 6C (start with 6 which has a 0 in the MSB is positive)
    - Since it is positive, apply hex place values $6C\ hex = 6*16^1 + 12*16^0 = 108_{10}$

Translating Hexadecimal

- Given B2 hex
  - If it is unsigned

    B2 hex = $11*16^1 + 2*16^0 = 178_{10}$
  - If it is signed
    - Determine the sign: MSD = 'B' thus negative
      [i.e. (- ____)]
    - Take the 16's complement and apply hex place values to find the neg. number's magnitude

Taking the 16's Complement

- Taking the 2's complement of a binary number yields its negative and is accomplished by finding the 1's complement (bit flip) and adding 1
- Taking the 16's complement of a hex number yields its negative and is accomplished by finding the 15's complement and adding 1
  - 15's complement is found by subtracting each digit of the hex number from F_{16}

Original value B2:	FF	
	- B2	Subtract each digit from F
	4D	15's comp. of B2
	+ 1	Add 1
16's comp. of B2:	4E	16's comp. of B2

Finding the Value of Hex Numbers

- 8A hex representing a signed (2's comp.) value
  - Step 1: Determine the sign: Neg.
  - Step 2: Take the 16's comp. to find magnitude

$$\begin{array}{r} \text{FF} \\ \underline{-\ \text{8A}} \\ 75 \\ \underline{+\ \ \ 1} \\ 76 \end{array}$$

  - Step 3: Apply hex place values ($76_{16} = +118_{10}$)
  - Step 4: Final value: B2 hex = -118_{10}
- 7C hex representing a signed (2's comp.) value
  - Step 1: Determine the sign: Pos.
  - Step 2: Apply hex place values ($7C_{16} = +124_{10}$)
- 7C hex representing an unsigned value = $+130_{10}$

ARITHMETIC AND OVERFLOW

2's Complement Addition/Subtraction

- Addition
  - Sign of the numbers do not matter
  - Add column by column
  - Drop any final carry-out
- Subtraction
  - Any subtraction (A-B) can be converted to addition (A + -B) by taking the 2's complement of B
  - (A-B) becomes (A + 1's comp. of B + 1)
  - Drop any carry-out

2's Complement Addition

- No matter the sign of the operands just add as normal
- Drop any extra carry out

0000
0011 (3)
+ 0010 (2)
0101 (5)

0000
1101 (-3)
+ 0010 (2)
1111 (-1)

Drop final carry-out

~~1~~110
0011 (3)
+ 1110 (-2)
0001 (1)

~~1~~100
1101 (-3)
+ 1110 (-2)
1011 (-5)

Unsigned and Signed Addition

- Addition process is the same for both unsigned and signed numbers
  - Add columns right to left
- Examples:

		If unsigned	If signed
	1 1		
	1001	(9)	(-7)
+	0011	(3)	(3)
	1100	(12)	(-4)

2's Complement Subtraction

- Take the 2's complement of the subtrahend and add to the original minuend
- Drop any extra carry out

```
   0011 (+3)          1101 (-3)
 - 0010 (+2)        - 1110 (-2)
 ----------         ----------

   1111_                 1_
   0011                1101
   1101  1's comp. of +2   0001  1's comp. of -2
 +    1  Add 1        +    1  Add 1
 ----------         ----------
   0001                1111
```

Drop final carry-out

Unsigned and Signed Subtraction

- Subtraction process is the same for both unsigned and signed numbers
  - Convert A – B to A + Comp. of B
  - Drop any final carry out
- Examples:

	If unsigned	If signed
1100	(12)	(-4)
- 0010	(2)	(2)

⇒

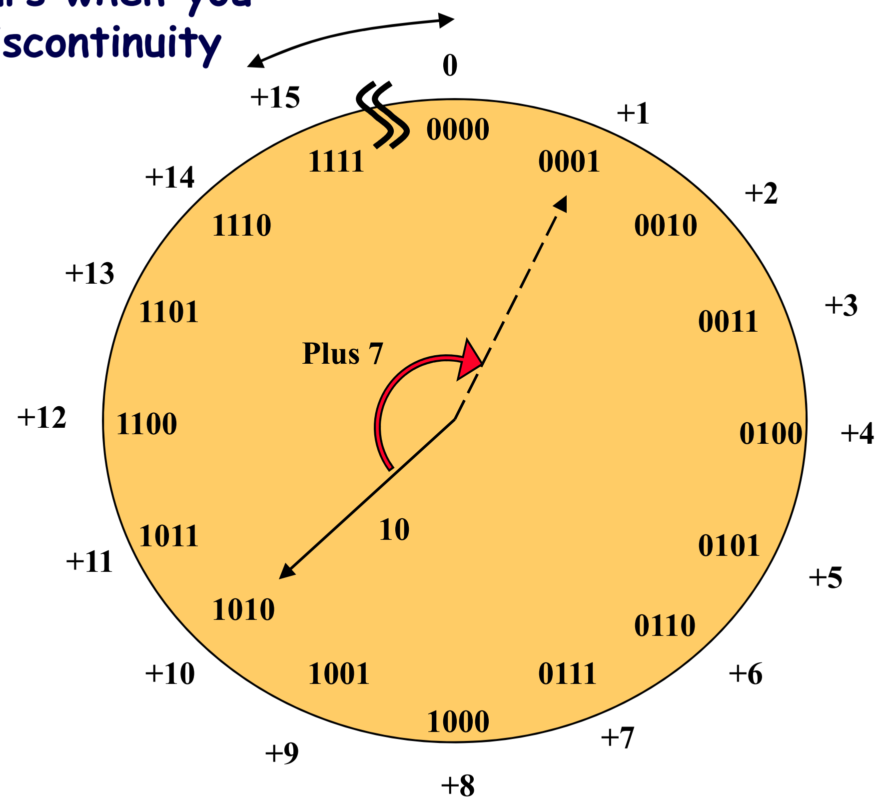
```
 11_1_
  1100   A
  1101   1's comp. of B
+    1   Add 1
  1010   (10)          (-6)
         If unsigned   If signed
```

Overflow

- Overflow occurs when the result of an arithmetic operation is too large to be represented with the given number of bits
- Conditions and tests to determine overflow depend on sign

Unsigned Overflow

Overflow occurs when you cross this discontinuity



10 + 7 = 17

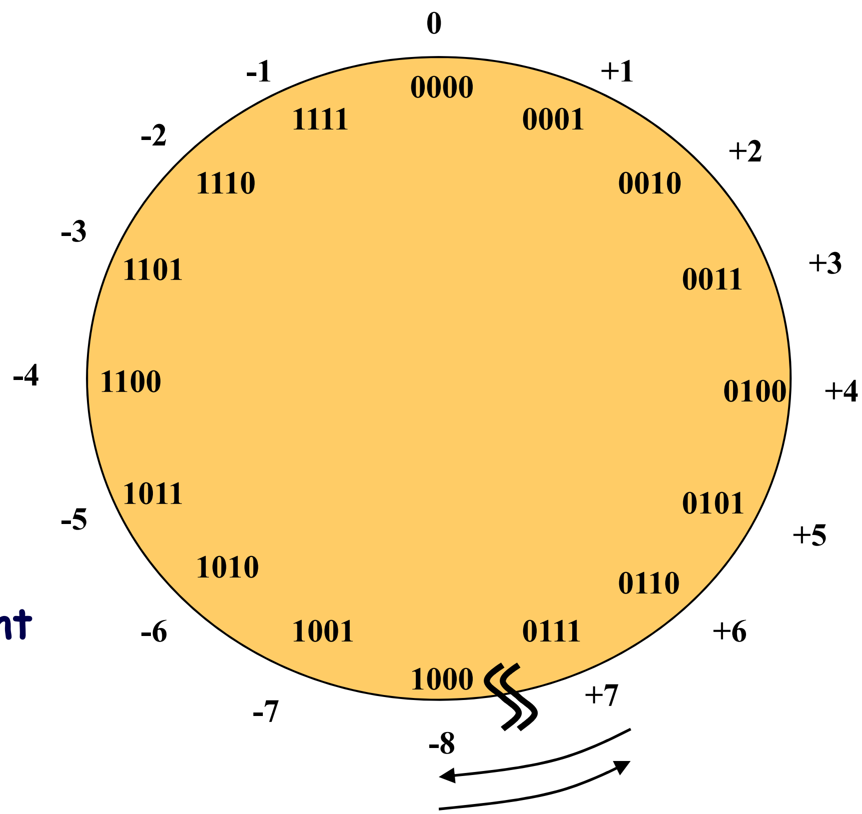
With 4-bit *unsigned* numbers we can only represent 0 – 15. Thus, we say overflow has occurred

2's Complement Overflow

$$5 + 7 = +12$$

$$-6 + -4 = -10$$

With 4-bit *2's complement* numbers we can only represent -8 to +7. Thus, we say overflow has occurred



Overflow occurs when you cross this discontinuity

Overflow in Addition

- Overflow occurs when the result of the addition cannot be represented with the given number of bits
- Tests for overflow:
  - Unsigned: if Cout = 1
  - Signed: if p + p = n or n + n = p

	If unsigned	If signed
1 1		
1101	(13)	(-3)
+ 0100	(4)	(4)
0001	(17)	(+1)
	Overflow Cout = 1	No Overflow n + p

	If unsigned	If signed
0 1		
0110	(6)	(6)
+ 0101	(5)	(5)
1011	(11)	(-5)
	No Overflow Cout = 0	Overflow p + p = n

Overflow in Subtraction

- Overflow occurs when the result of the subtraction cannot be represented with the given number of bits
- Tests for overflow:
  - Unsigned: if Cout = 0
  - Signed: if addition is p + p = n   or   n + n = p

	If unsigned	If signed
0111	(7)	(7)
- 1000	(8)	(-8)
	(-1)	(15)

Desired Results

0111_			
0111	A		
0111	1's comp. of B		
+ 1	Add 1		
1111		(15)	(-1)
		If unsigned Overflow Cout = 0	If signed Overflow p + p = n

Hex Addition and Overflow

- Same rules as in binary
  - Add left to right
  - Drop any carry (carry occurs when sum > F_{16})
- Same addition overflow rules
  - Unsigned: Check if final Cout = 1
  - Signed: Check signs of inputs and result

```
 1  1
  7AC5
+ C18A
  3C4F
```

If unsigned: Overflow, Cout = 1

If signed: No Overflow, p + n

```
 0 1  1
  6C12
+ 549F
  C0B1
```

If unsigned: No Overflow, Cout = 0

If signed: Overflow, p + p = n

Hex Subtraction and Overflow

- Same rules as in binary
  - Convert A – B to A + Comp. of B
  - Drop any final carry out
- Same subtraction overflow rules
  - Unsigned: Check if final Cout = 0
  - Signed: Check signs of addition inputs and result

```
  B1ED          1
- 76FE  =>     B1ED
               8901
             +    1
             ------
               3AEF
```

If unsigned
No Overflow
Cout = 1

If signed
Overflow
n + n = p

```
  0001          0
- 0002  =>     0001
               FFFD
             +    1
             ------
               FFFF
```

If unsigned
Overflow
Cout = 0

If signed
No Overflow
p + n

Unsigned Multiplication Review

- Same rules as decimal multiplication
- Multiply each bit of Q by M shifting as you go
- An m-bit * n-bit mult. produces an m+n bit result
- Notice each partial product is a shifted copy of M or 0 (zero)

```
    1010  M (Multiplicand)
  * 1011  Q (Multiplier)
  ------
    1010
   1010_  PP(Partial
  0000__     Products)
+ 1010___
--------
01101110  P (Product)
```

Signed Multiplication Techniques

- When multiplying signed (2's comp.) numbers, some new issues arise
- Must sign extend partial products (out to 2n bits)

Without Sign Extension… Wrong Answer!

```
      1001  = -7
   *  0110  = +6
   --------
      0000
     1001_
    1001__
 +  0000___
 ----------
  00110110  = +54
```

With Sign Extension… Correct Answer!

```
      1001  = -7
   *  0110  = +6
   --------
  00000000
  1111001_
  111001__
+ 00000___
----------
  11010110  = -42
```

Signed Multiplication Techniques

- Also, must worry about negative multiplier
  - MSB of multiplier has negative weight
  - If MSB=1, multiply by -1 (i.e. take 2's comp. of multiplicand)

With Sign Extension but w/o consideration of MSB… Wrong Answer!

```
    1100  = -4
  * 1010  = -6
00000000
1111100_
000000__
+ 11100___
11011000  = -40
```

Place Value: -8
Multiply by -1

With Sign Extension and w/ consideration of MSB… Correct Answer!

```
    1100  = -4
  * 1010  = -6
00000000
1111100_
000000__
+ 00100___
00011000  = +24
```

Main Point: Signed and Unsigned Multiplication require different techniques…Thus different instructions

Binary Representation Systems

- Integer Systems
  - Unsigned
    - Unsigned (Normal) binary
  - Signed
    - Signed Magnitude
    - 2's complement
    - *1's complement**
    - *Excess-N**
- Floating Point
  - For very large and small (fractional) numbers
- Codes
  - Text
    - ASCII / Unicode
  - Decimal Codes
    - BCD (Binary Coded Decimal) / (8421 Code)

* = Not covered in this class

Binary Codes

- Using binary we can represent any kind of information by coming up with a code
- Using n bits we can represent 2^n distinct items

Colors of the rainbow:
- Red = 000
- Orange = 001
- Yellow = 010
- Green = 100
- Blue = 101
- Purple = 111

Letters:
- 'A' = 00000
- 'B' = 00001
- 'C' = 00010

.
.
.

- 'Z' = 11001

ASCII Code

- Used for representing text characters
- Originally 7-bits but usually stored as 8-bits = 1- byte in a computer
- Example:
  - `printf("Hello\n");`
  - Each character is converted to ASCII equivalent
    - 'H' = 0x48, 'e' = 0x65, …
    - \n = newline character is represented by either one or two ASCII character
      - LF (0x0A) = line feed (moves cursor down a line)
      - CR (0x0D) = carriage return character (moves cursor to start of current line)
      - Newline for Unix / Mac = LF only
      - Newline for Windows = CR + LF

ASCII Table

LSD/MSD	0	1	2	3	4	5	6	7
0	NULL	DLW	SPACE	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	TAB	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Unicode

- Supersedes ASCII and incorporates most alphabets/characters of other languages
- 16-bit (2-byte) character code
- Used by Java

BCD

- Rather than convert a decimal number to binary which may lose some precision (i.e. 0.1_{10} = infinite binary fraction), BCD represents each decimal digit as a separate group of bits (exact decimal precision)
  - Each digits is represented as a separate 4-bit number (using place values 8,4,2,1 for each dec. digit)
  - Often used in financial and other applications where decimal precision is needed

$$(439)_{10}$$

BCD Representation: **0100 0011 1001**

This is the Binary Coded Decimal (BCD) representation of 439

Unsigned Binary Rep.: 110110111_2

This is the binary representation of 439 (i.e. using power of 2 place values)

Important: Some processors have specific instructions to operate on #'s represented in BCD