Unit 11

Signed Representation Systems

Binary Arithmetic

BINARY REPRESENTATION SYSTEMS REVIEW

Interpreting Binary Strings

- Given a string of 1's and 0's, you need to know the *representation system* being used, before you can understand the value of those 1's and 0's.
- Information (value) = Bits + Context (System)

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

Signed Magnitude

2's Complement System

SIGNED SYSTEMS

Unsigned and Signed

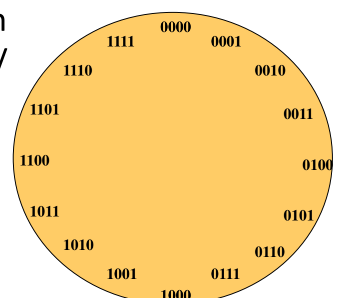
- Normal (unsigned) binary can only represent positive numbers
  - All place values are positive
- To represent BOTH positive and negative numbers we must use the available binary codes differently, some for the positive values and others for the negative values
  - We call these *signed* representations

Signed Number Representation

- 2 Primary Systems
  - ______________________
  - ______________________ *(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
  - _________ means negative
  - _________ 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	
___	___	___	___	⇒ **0 to 15**
8	4	2	1	

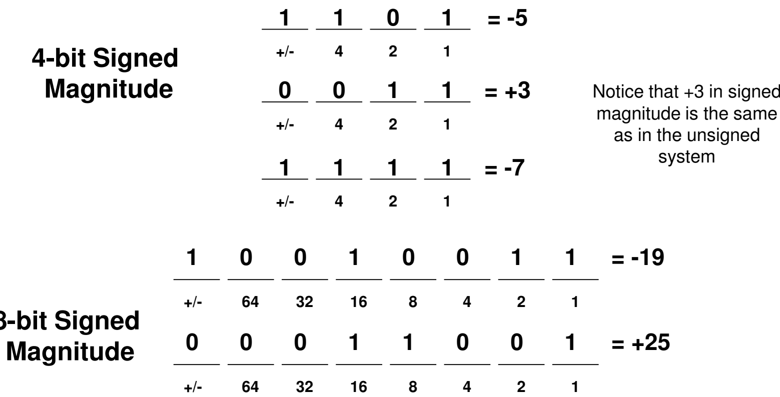
4-bit Signed Magnitude

Bit 3	Bit 2	Bit 1	Bit 0	
___	___	___	___	⇒ **-7 to +7**
+/-	4	2	1	

8-bit Signed Magnitude

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
___	___	___	___	___	___	___	___	⇒ **-127 to +127**
+/-	64	32	16	8	4	2	1	

Signed Magnitude Examples



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 _____

   0000 = 1000 = ____

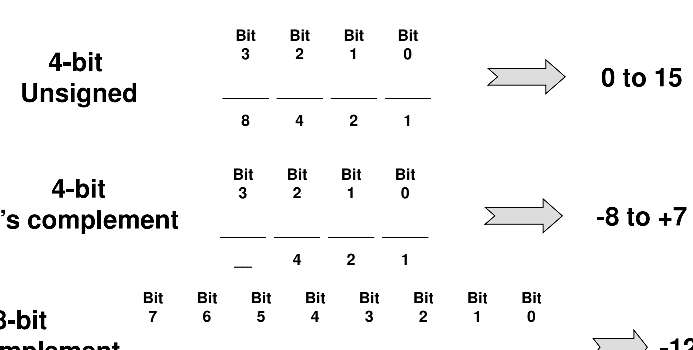
2. Addition and subtraction algorithms for signed magnitude are different than unsigned binary (we'd like them to be the same to use same HW)

```
  4          6
- 6        - 4
----       ----
            -
```

Swap & make res. negative

2's Complement System

- Normal binary place values except MSB has ____________________
  - MSB of 1 = ______



USC Viterbi School of Engineering

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 _______ representation in 2's complement as in normal unsigned binary

2's Complement Range

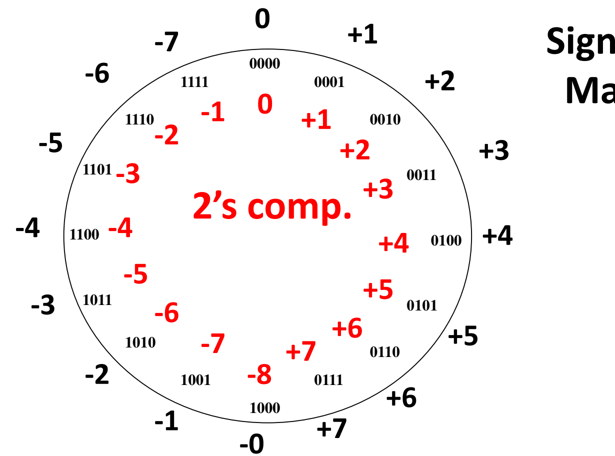
- Given n bits…
  - Max positive value = _____________
    - Includes all n-1 positive place values
  - Max negative value = _________________
    - Includes only the negative MSB place value

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?

Comparison of Systems



Signed Mag.

Unsigned and Signed Variables

- In C, ________ variables use unsigned binary (normal power-of-2 place values) to represent numbers

```
unsigned char x = 147;
```

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

- In C, signed variables use the _________________ (Neg. MSB weight) to represent numbers

```
char x = -109;
```

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

IMPORTANT NOTE

- All computer systems use the **2's complement system** to represent **signed integers**!
- So from now on, if we say an integer is **signed**, we are actually saying it uses the 2's complement system unless otherwise specified
  - We will not use "signed magnitude" unless explicitly indicated

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 ___'s):

111011 = ___111011

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

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

pos. 011010 = ___011010

neg. 110011 = ___110011

_____ 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):

~~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 _______ of sign bit):

pos. 00011010 = _________

neg. 11110011 = _________

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.

Data Representation

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

C Type	Bytes	Bits	ATmega328
[unsigned] char	1	8	byte
[unsigned] short [int]	2	16	word
[unsigned] long [int]	4	32	-[1]
[unsigned] long long [int]	8	64	-[1]
int	?[2]	?[2]	?[2]

[1]Can emulate but has no single-instruction support
[2]Varies by compiler/machine (avr-gcc: int = 2 bytes, g++ for x86: int = 4-bytes)

  - Floating Point Types

C Type	Bytes	Bits	ATmega328
float	4	32	N/A[1]
double	8	64	N/A[1]

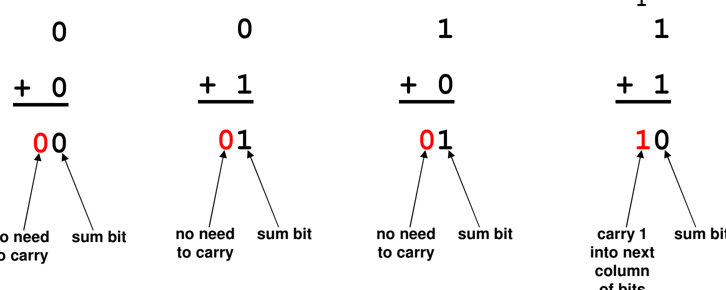
ARITHMETIC

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

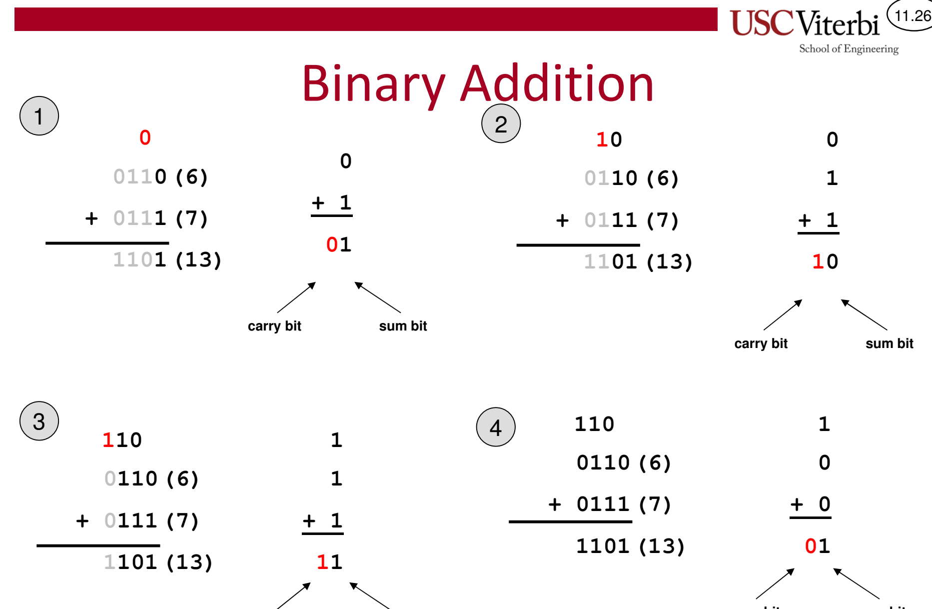
- In decimal addition we ______ when the sum is 10 or more
- In binary addition we carry when the sum is __ or more
- Add bits in binary to produce a sum bit and a carry bit



Binary Addition & Subtraction

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


Binary Addition



Hexadecimal Arithmetic

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

$$\begin{array}{r} 4\ D_{16} \\ +\ B\ 5_{16} \\ \hline \end{array}$$

__ __
16 1

__ __
16 1

"Taking the 2's complement"

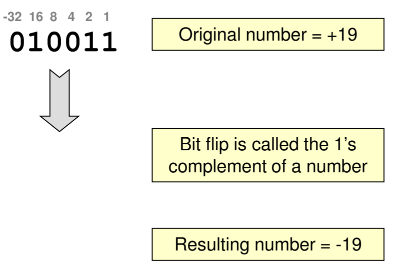
SUBTRACTION THE EASY WAY

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: _______________
    - 0110 = +6 => _____________
  - 2's complement: "__________________________"
    - 0110 = +6 => _________
    - Operation defined as:
      1. Flip/invert/not _________ (1's complement)
      2. Add ___ and drop any _______
         (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)



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

Taking the 2's Complement

101010

Original number = ____

Take the 2's complement yields the negative of a number

Resulting number = ____

Back to original = ____

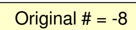
0000

Original # = 0

Take the 2's complement

2's comp. of 0 is __

1000



Original # = -8

Take the 2's complement

Negative of -8 is __

(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
```

ADDITION AND SUBTRACTION

2's Complement Addition/Subtraction

- Addition
  - Sign of the numbers do not matter
  - ___________________
  - ___________________
- Subtraction
  - Any subtraction (A-B) can be converted to addition (_________) by taking the __________________ of B
  - (A-B) becomes (_____________________)
  - Drop any carry-out
- The _______ of the result is produced by performing the above process and need not be considered separately

2's Complement Addition

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

```
  0011        1101
+ 0010      + 0010
------      ------

  0011        1101
+ 1110      + 1110
------      ------
```

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**

```
  1001
+ 0011
------
```

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)
- 0010 (+2)
```

```
  1101 (−3)
- 1110 (−2)
```


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)

→

If unsigned	If signed

Important Note

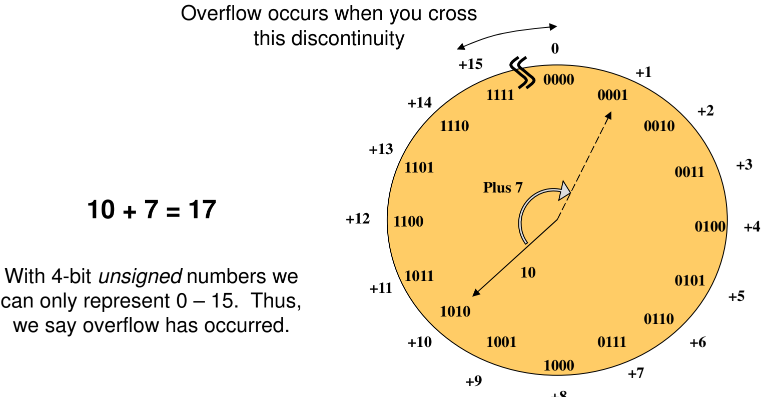
- Almost all computers use 2's complement because…
- The same addition and subtraction ____________ can be used on unsigned and 2's complement (signed) numbers
- Thus we only need one _________________________ to perform operations on both unsigned and signed numbers

OVERFLOW

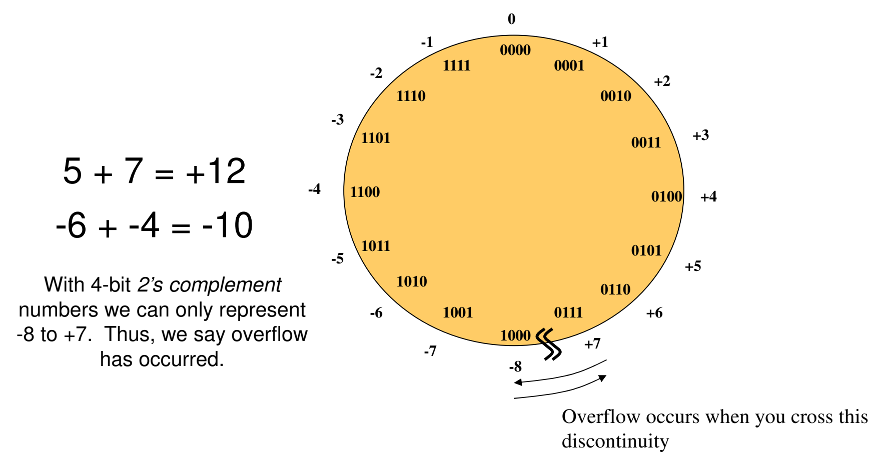
Overflow

- Overflow occurs when the result of an arithmetic operation is ______________ ______________________________
- Conditions and tests to determine overflow depend on ________________ of numbers (signed or unsigned) in the operation

Unsigned Overflow



2's Complement Overflow



Overflow in Addition

- Overflow occurs when the result of the addition cannot be represented with the given number of bits.
- Tests for overflow:
  - Unsigned: ________________
  - Signed: _____________________

	If unsigned	If signed
1101 + 0100 0001		

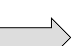
	If unsigned	If signed
0110 + 0101 1011		

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
- 1000
```



FLOATING POINT

Floating Point

- Used to represent very small numbers (fractions) and very large numbers
  - Avogadro's Number: $+6.0247 * 10^{23}$
  - Planck's Constant: $+6.6254 * 10^{-27}$
  - Note: 32 or 64-bit integers can't represent this range
- Floating Point representation is used in HLL's like C by declaring variables as `float` or `double`

Fixed Point

- Unsigned and 2's complement fall under a category of representations called "Fixed Point"
- The radix point is assumed to be in a fixed location for all numbers [Note: we could represent fractions by implicitly assuming the binary point is at the _________. Variables just store bits…you can assume the binary point is anywhere you like]
  - Integers: 10011101. (binary point to right of LSB)
    - For 32-bits, unsigned range is 0 to ~4 billion
  - Fractions: .10011101 (binary point to left of MSB)
    - Range [0 to 1)

Bit storage

Fixed point Rep.

- **Main point**: By ___________ the radix point, we limit the range of numbers that can be represented
  - Floating point allows the radix point to be in a different location for each value

Floating Point Representation

- Similar to scientific notation used with decimal numbers
  - ±D.DDD * 10 $^{\pm exp}$
- Floating Point representation uses the following form
  - ___________________
  - 3 Fields: _____, __________, ______________ (also called _____________________)

Normalized FP Numbers

- Decimal Example
  - $+0.754*10^{15}$ is not correct scientific notation
  - Must have exactly _____ significant digit before decimal point: _______________
- In binary the only _______________is '1'
- Thus normalized FP format is:

  ____
- FP numbers will **always be normalized** before being __________ in memory or a reg.
  - The *1.* is actually ___________ but assumed since we always will store normalized numbers
  - If HW calculates a result of $0.001101*2^5$ it must normalize to $1.101000*2^2$ before storing

IEEE Floating Point Formats

- Single Precision (32-bit format)
  - 1 Sign bit (0=pos/1=neg)
  - __ Exponent bits
    - _________ representation
    - More on next slides
  - ___ fraction (significand or mantissa) bits
  - Equiv. Decimal Range:
    - 7 digits x $10^{\pm 38}$

1	8	23
S	Exp.	Fraction

- Double Precision (64-bit format)
  - 1 Sign bit (0=pos/1=neg)
  - ___ Exponent bits
    - _________ representation
    - More on next slides
  - ___ fraction (significand or mantissa) bits
  - Equiv. Decimal Range:
    - 16 digits x $10^{\pm 308}$

1	11	52
S	Exp.	Fraction

Floating Point vs. Fixed Point

- Single Precision (32-bits) Equivalent Decimal Range:
  - 7 significant decimal digits * $10^{\pm 38}$
  - Compare that to 32-bit signed integer where we can represent ±2 billion. How does a 32-bit float allow us to represent such a greater range?
  - FP allows for range but sacrifices precision (can't represent all numbers in its range)
- Double Precision (64-bits) Equivalent Decimal Range:
  - 16 significant decimal digits * $10^{\pm 308}$

Exponent Representation

- Exponent needs its own sign (+/-)
- Rather than using 2's comp. system we use Excess-N representation
  - Single-Precision uses Excess-127
  - Double-Precision uses Excess-1023
  - This representation allows FP numbers to be easily compared
- Let E' = stored exponent code and E = true exponent value
- For single-precision: E' = E + 127
  - 2^1 => E = 1, E' = 128_{10} = 10000000_2
- For double-precision: E' = E + 1023
  - 2^{-2} => E = -2, E' = 1021_{10} = 01111111101_2

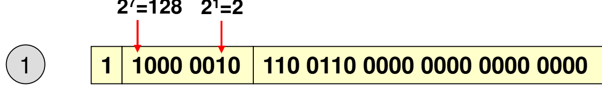
2's comp.	E' (stored Exp.)	Excess-127
	1111 1111	
	1111 1110	
	1000 0000	
	0111 1111	
	0111 1110	
	0000 0001	
	0000 0000	

Comparison of 2's comp. & Excess-N

Q: Why don't we use Excess-N more to represent negative #'s

Single-Precision Examples

1.



2. **+0.6875 = +0.1011**