

Unit 11

Signed Representation Systems
Binary Arithmetic

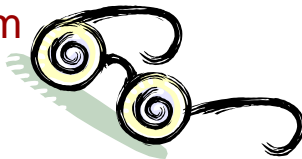
BINARY REPRESENTATION SYSTEMS REVIEW

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)

01000001 = ?

Unsigned
Binary system



65₁₀

BCD System



41_{BCD}

ASCII
system



'A'_{ASCII}

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

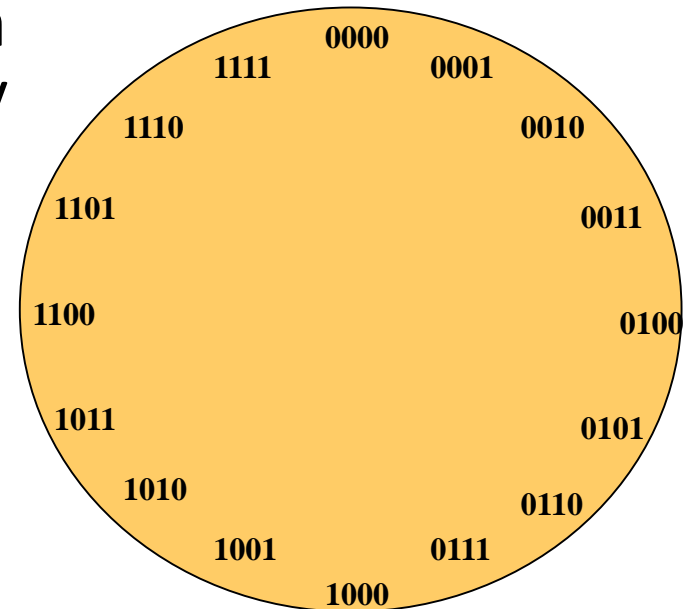
- _____

- _____ *(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

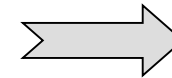
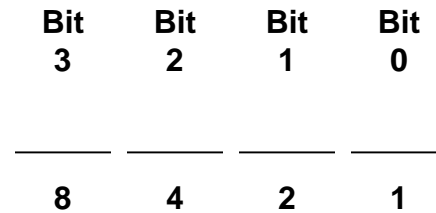
- _____ means negative
- _____ means positive



Signed Magnitude System

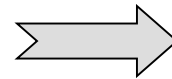
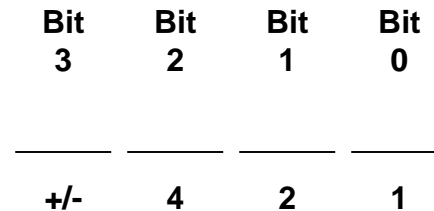
- Use binary place values but now MSB represents the sign (1 if negative, 0 if positive)

4-bit Unsigned



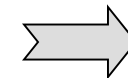
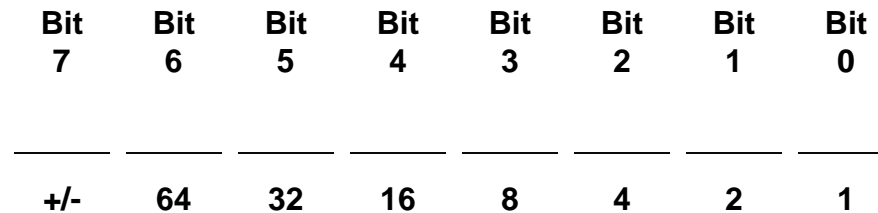
0 to 15

4-bit Signed Magnitude



-7 to +7

8-bit Signed Magnitude



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

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

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 = $[0 \text{ to } 2^{n-1}-1]$

Range with n -bits of Signed Magnitude

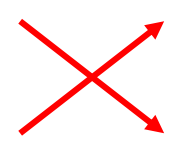
$$[-(2^{n-1}-1) \text{ to } +(2^{n-1}-1)]$$

Disadvantages of Signed Magnitude

1. Wastes a combination to represent _____

$$0000 = 1000 = \underline{\hspace{2cm}}$$

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)

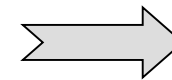
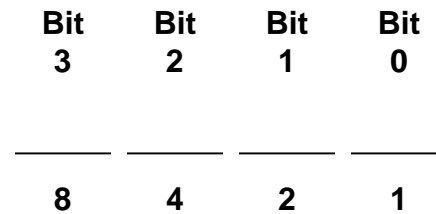
$\begin{array}{r} 4 \\ - 6 \\ \hline \end{array}$		$\begin{array}{r} 6 \\ - 4 \\ \hline \end{array}$
Swap & make res. negative		

2's Complement System

- Normal binary place values except MSB has

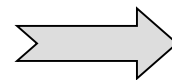
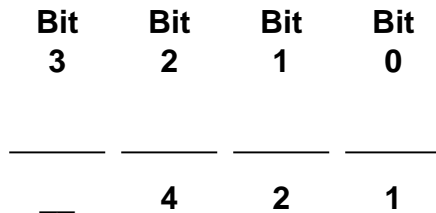
– MSB of 1 = _____

**4-bit
Unsigned**



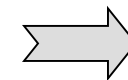
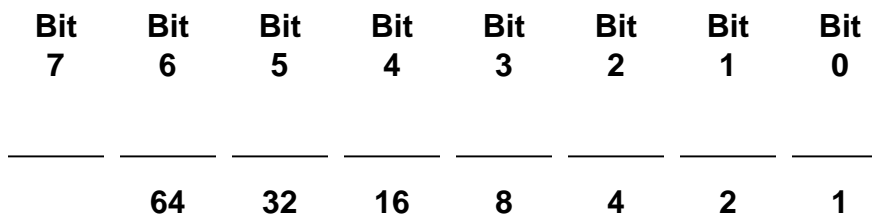
0 to 15

**4-bit
2's complement**



-8 to +7

**8-bit
2's complement**



-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**

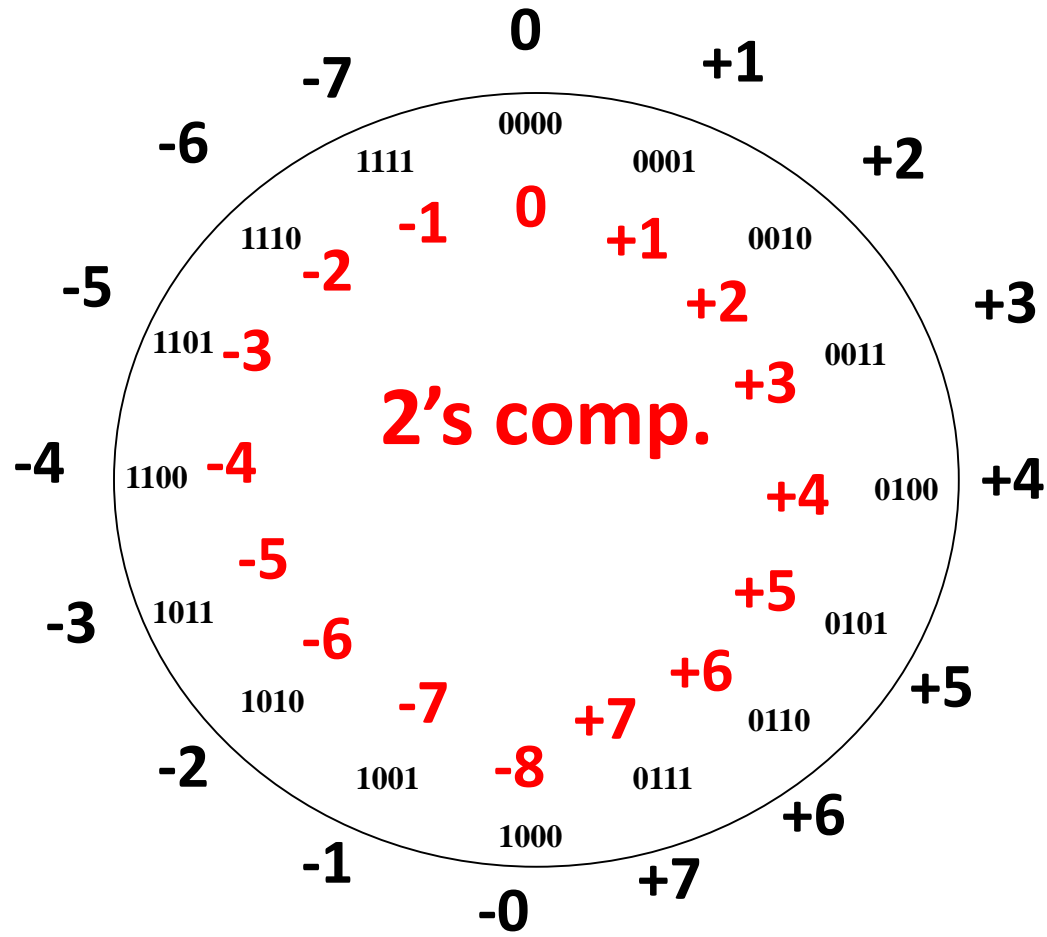
1	0	0	0	0	0	0	0	1	= -127
-128	64	32	16	8	4	2	1		
0	0	0	1	1	0	0	1	1	= +25
-128	64	32	16	8	4	2	1		

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;`

$$\begin{array}{cccccccc} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & = +147 \\ \hline 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 & \end{array}$$

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

`char x = -109;`

$$\begin{array}{cccccccc} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & = -109 \\ \hline -128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 & \end{array}$$

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):

$$111011 = \underline{\quad} 111011$$


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

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

pos.	011010 = <u> </u> 011010	___ bit is just repeated as many times as necessary
neg.	110011 = <u> </u> 110011	

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 _____ of sign bit):

pos. $00011010 = \underline{\hspace{2cm}}$

neg. $11110011 = \underline{\hspace{2cm}}$

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

¹Can emulate but has no single-instruction support

²Varies 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

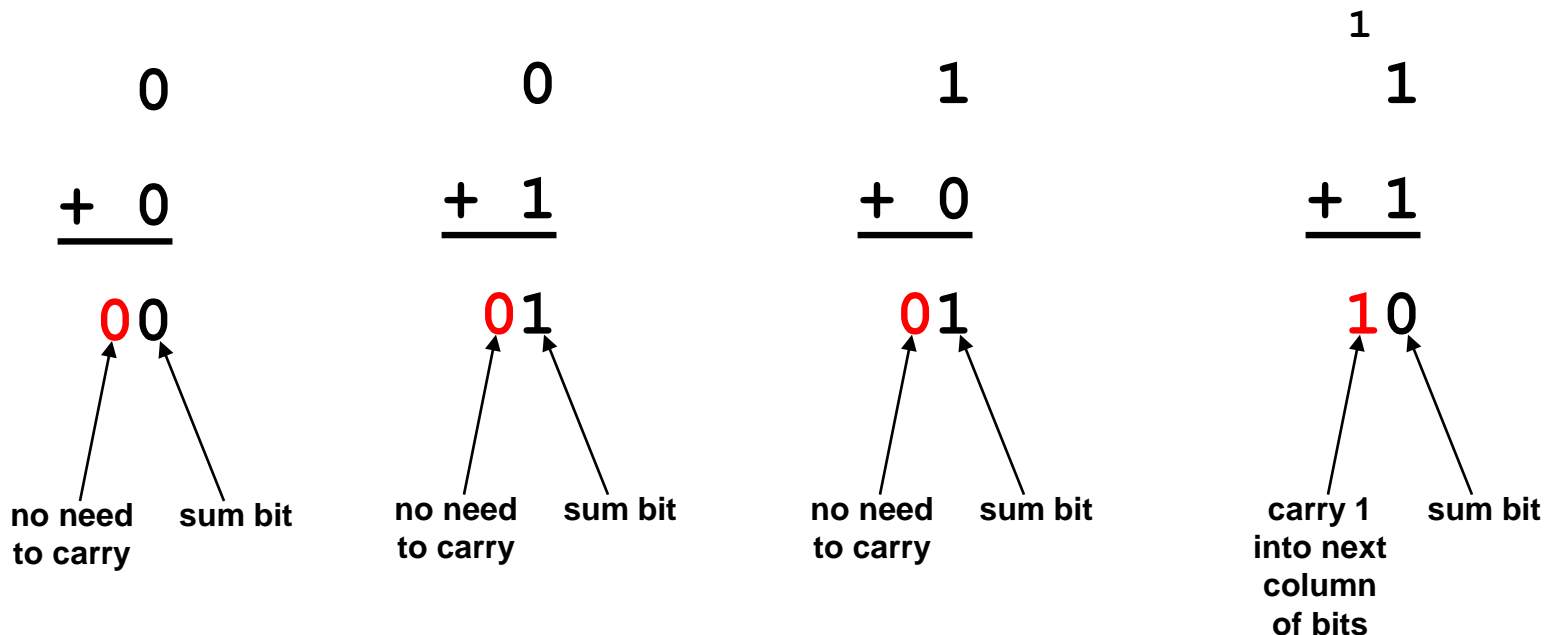
ARITHMETIC

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

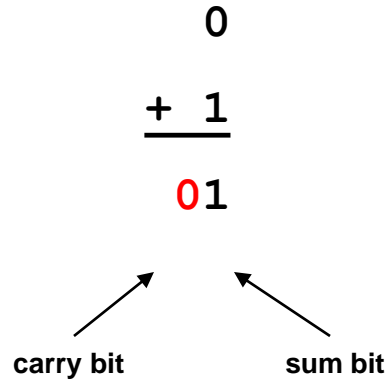
$$\begin{array}{r} 0\ 1\ 1\ 1 \\ +\ 0\ 0\ 1\ 1 \\ \hline \end{array}$$

$$\begin{array}{r} 1\ 0\ 1\ 0 \\ -\ 0\ 1\ 0\ 1 \\ \hline \end{array}$$

Binary Addition

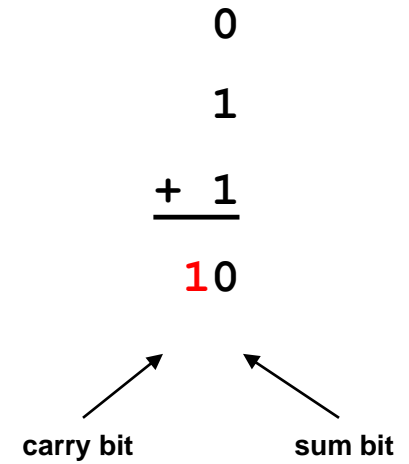
1

$$\begin{array}{r}
 0 \\
 0110 \text{ (6)} \\
 + 0111 \text{ (7)} \\
 \hline
 1101 \text{ (13)}
 \end{array}$$



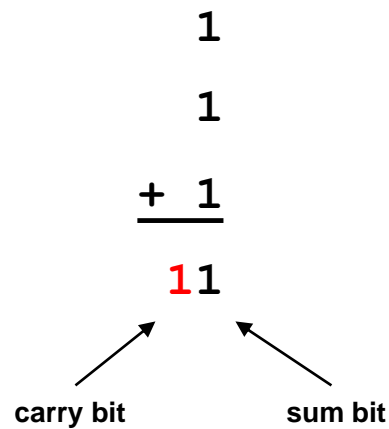
2

$$\begin{array}{r}
 10 \\
 0110 \text{ (6)} \\
 + 0111 \text{ (7)} \\
 \hline
 1101 \text{ (13)}
 \end{array}$$



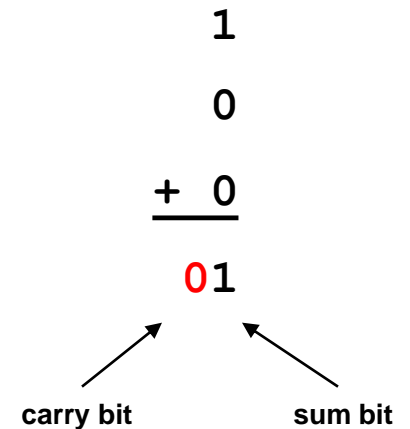
3

$$\begin{array}{r}
 110 \\
 0110 \text{ (6)} \\
 + 0111 \text{ (7)} \\
 \hline
 1101 \text{ (13)}
 \end{array}$$



4

$$\begin{array}{r}
 110 \\
 0110 \text{ (6)} \\
 + 0111 \text{ (7)} \\
 \hline
 1101 \text{ (13)}
 \end{array}$$



Hexadecimal Arithmetic

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

$$\begin{array}{r} 4 \text{ D}_{16} \\ + \text{ B } 5_{16} \\ \hline \end{array} \qquad \begin{array}{r} \underline{\quad} \underline{\quad} \\ \color{red}{16} \color{red}{1} \\ \underline{\quad} \underline{\quad} \\ \color{red}{16} \color{red}{1} \end{array}$$

"Taking the 2's complement"

SUBTRACTION THE EASY WAY

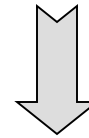
Taking the Negative

- Given a number in signed magnitude or 2's complement how do we find its negative (i.e. $-1 * X$)
 - Signed Magnitude: _____
 - $0110 = +6 \Rightarrow$ _____
 - 2's complement: “ _____ ”
 - $0110 = +6 \Rightarrow$ _____
 - 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)

-32 16 8 4 2 1
010011



Original number = +19

Bit flip is called the 1's complement of a number

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}
101010 Original number = _____

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

Resulting number = _____

2 **0000** Original # = 0

Take the 2's complement

2's comp. of 0 is _____

3 **1000** Original # = -8

Take the 2's complement

Back to original = _____

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

$$\begin{array}{r} 1001 \\ 0110 \\ + \quad 1 \\ \hline 0111 \end{array} \quad \begin{array}{l} x = -7 \\ \text{Bit flip (1's comp.)} \\ \text{Add 1} \\ -x = -(-7) = +7 \end{array}$$

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

$$\begin{array}{r} 0011 \\ + 0010 \\ \hline \end{array}$$

$$\begin{array}{r} 1101 \\ + 0010 \\ \hline \end{array}$$

$$\begin{array}{r} 0011 \\ + 1110 \\ \hline \end{array}$$

$$\begin{array}{r} 1101 \\ + 1110 \\ \hline \end{array}$$

Unsigned and Signed Addition

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

$$\begin{array}{r} 1001 \\ + 0011 \\ \hline \end{array}$$

If unsigned If signed

2's Complement Subtraction

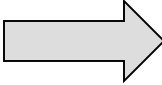
- Take the 2's complement of the subtrahend and add to the original minuend
- Drop any extra carry out

$$\begin{array}{r} 0011 (+3) \\ - 0010 (+2) \\ \hline \end{array}$$

$$\begin{array}{r} 1101 (-3) \\ - 1110 (-2) \\ \hline \end{array}$$

Unsigned and Signed Subtraction

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

	<u>If unsigned</u>	<u>If signed</u>	
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 _____

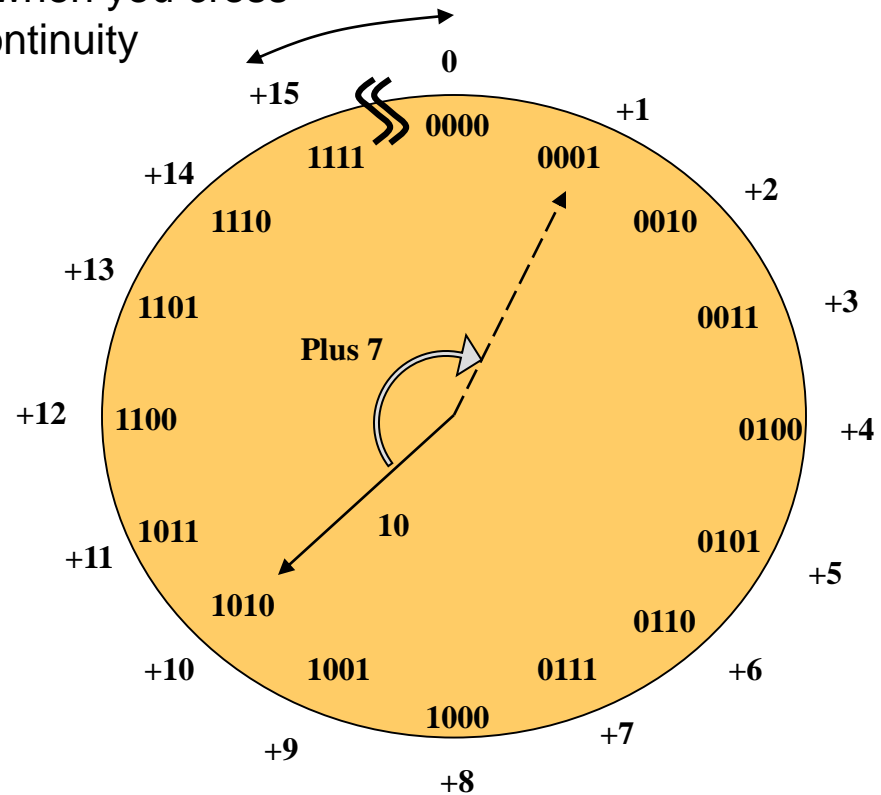
- Conditions and tests to determine overflow depend on _____
of numbers (signed or unsigned) in the operation

Unsigned Overflow

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.

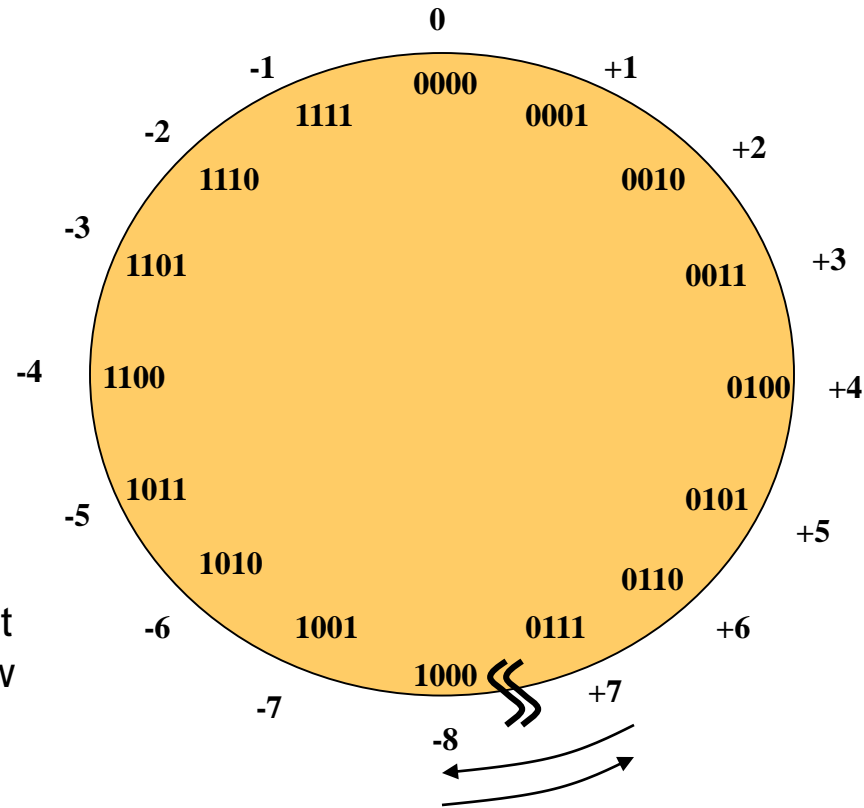


2's Complement Overflow

$$5 + 7 = +12$$

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

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: _____
 - Signed: _____

	<u>If unsigned</u>	<u>If signed</u>
	1101	0110
+	0100	0101
	0001	1011

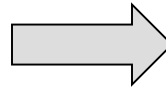
	<u>If unsigned</u>	<u>If signed</u>
	0110	0110
+	0101	0101
	1011	1011

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 $C_{out} = 0$
 - Signed: if addition is $p + p = n$ or $n + n = p$

If unsigned If signed

$$\begin{array}{r}
 0111 \\
 - 1000 \\
 \hline
 \end{array}$$



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)
- **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

Bit storage**Fixed point Rep.**

Floating Point Representation

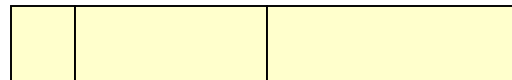
- Similar to scientific notation used with decimal numbers

– $\pm D.DDD * 10^{\pm exp}$

- Floating Point representation uses the following form

– _____

– 3 Fields: _____, _____, _____
 (also called _____)



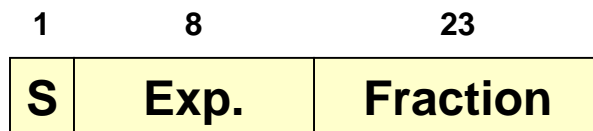
Normalized FP Numbers

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

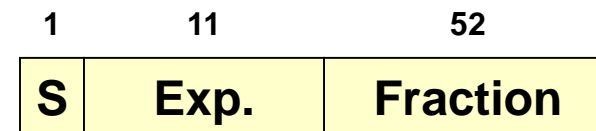
- FP numbers will **always be normalized** before being _____ in memory or a reg.
 - The **1.** is actually _____ but assumed since we always will store normalized numbers
 - If HW calculates a result of $0.001101 \cdot 2^5$ it must normalize to $1.101000 \cdot 2^2$ before storing

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 $10^{\pm 38}$



- 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 $10^{\pm 308}$



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

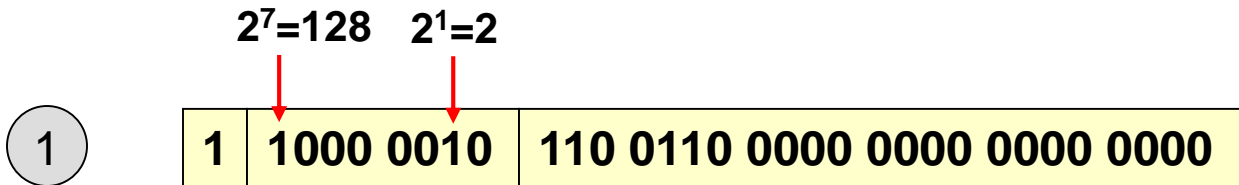
- Exponent needs its own sign (+/-)
- Rather than using 2's comp. system we use Excess-N representation
 - Single-Precision uses Excess-127
 - Double-Precision uses Excess-1023
 - This representation allows FP numbers to be easily compared
- Let E' = stored exponent code and E = true exponent value
-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

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



2 **+0.6875 = +0.1011**

