Background Knowledge

CO 2103 Assembly Language

Topics

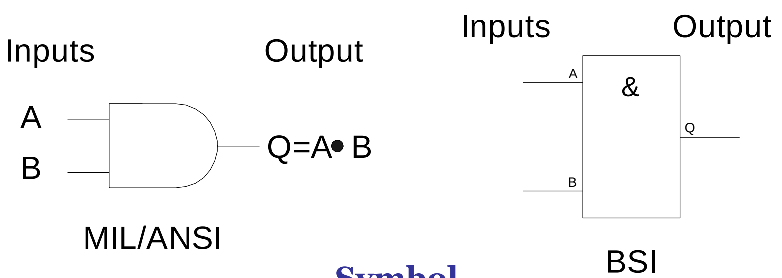
- Digital Logic
- Number Systems
  - Binary, Octal, Decimal, Hexadecimal
  - Conversion between number systems
  - Basic arithmetic operations
- Data Representations
  - Integer – signed/unsigned, sign & magnitude, 1's, 2's complement, BCD, biased
  - Real – floating point
  - Text – ASCII

Digital Logic

- Rationale
  - computers operate electronically, using Logic Gates
  - Logic Gates easily connected to perform more complex functions, form the basic "building blocks" of computers
- Logic Gates
  - electronic circuits, usually written as symbols
  - 1 output, 1 or more inputs
  - information (values) interpreted from inputs/output have/don't have electronic signal (voltage)
  - have voltage = ON = Logic 1
  - no voltage = OFF = Logic 0

Basic Logic Gates - 1

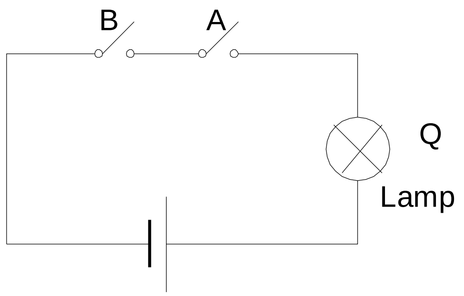
- Basic Logic operations:
  - AND, OR, NOT, Exclusive OR (XOR)
- **AND**



Symbol

Truth Table

Inputs		*Output*
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1



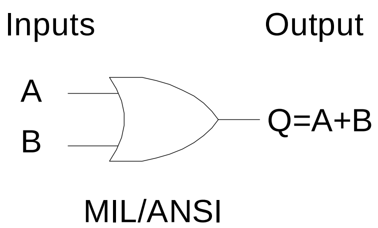
Illustration

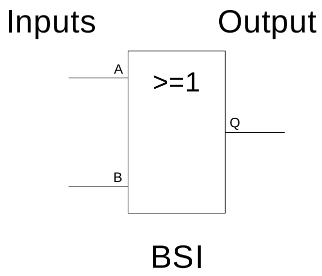
Basic Logic Gates - 2

- **OR**

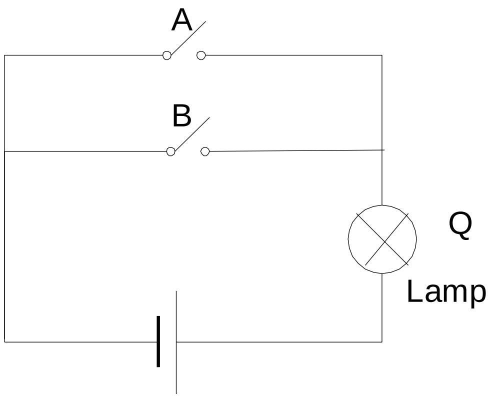
Truth Table

Inputs		*Output*
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1





Symbol



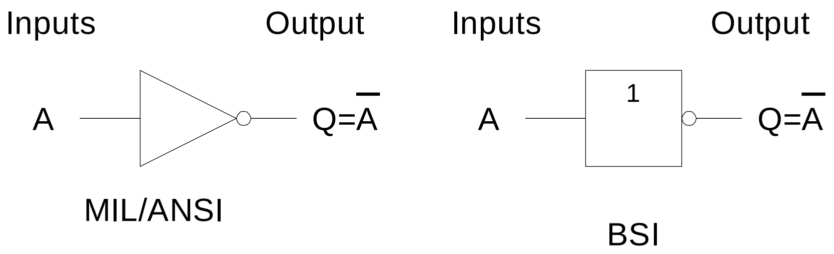
Illustration

Basic Logic Gates - 3

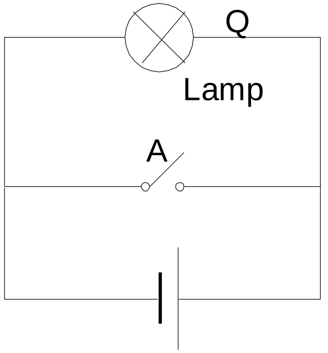
- **NOT**

Truth Table

Input A	Output Q
0	1
1	0



Symbol



Illustration

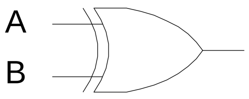
Basic Logic Gates - 4

- **XOR**

Truth Table

Inputs		*Output*
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

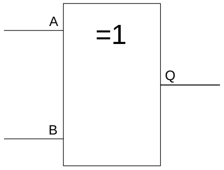
Inputs Output



$Q = A \oplus B$

MIL/ANSI

Inputs Output

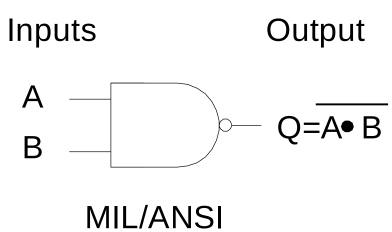


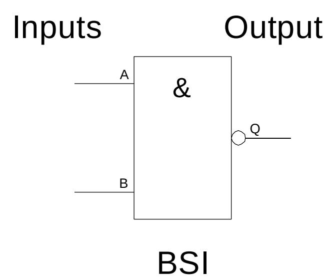
BSI

Symbol

Derived Logic Gates

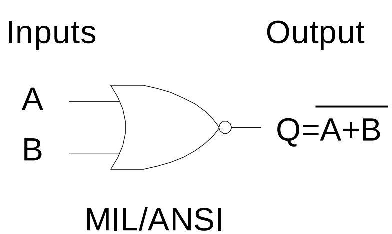
- Derived Logic operations:
  - NAND (Not-AND)
  - NOR (Not-OR)
- **NAND**

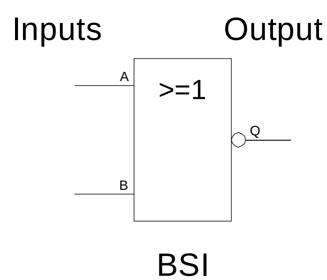




Inputs		*Output*
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

- **NOR**

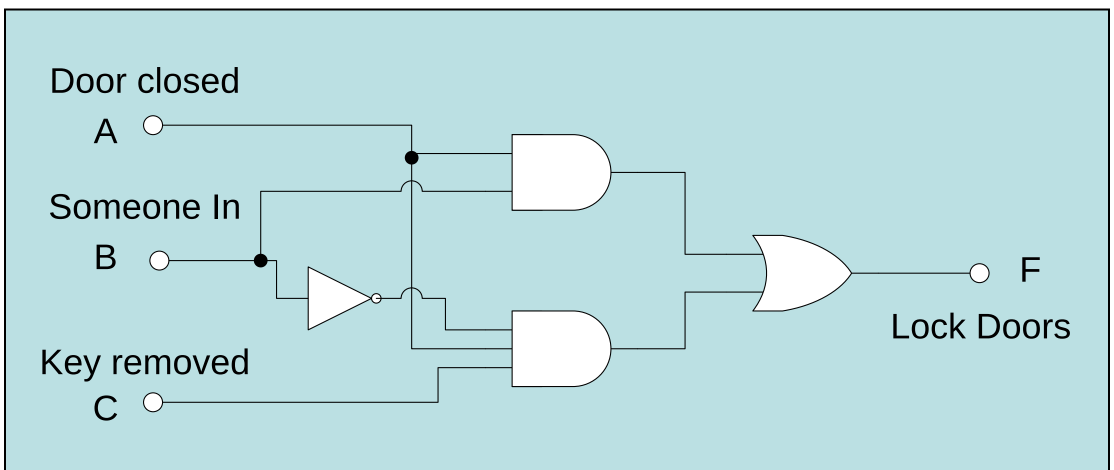




Inputs		*Output*
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

Logic Circuit

- Combine gates into logic circuits to perform useful functions
- Example: "Auto Lock" the doors if:
  - someone is in the car AND the doors are closed, OR
  - NO one is in the car AND the key is removed AND the doors are closed



Truth Table?

Number Systems - 1

- Decimal – base 10
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Binary – base 2
  - 0, 1
- Octal – base 8
  - 0, 1, 2, 3, 4, 5, 6, 7
- Hexadecimal (Hex) – base 16
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Number Systems - 2

- The base is the number of available symbols (0, 1, 2, … A, B, … F) to form the numbers
- Why do we have different number systems?
  - Decimal – part of our life; we have 10 fingers?
  - Binary – part of computer's life; 0 and 1 only
  - Hexadecimal – convenient shorthand for binary
- All systems follow same rules of counting, i.e. when we reach the last symbol we add another 'digit' to the left
  - Decimal – 0, 1, … 9, 10, 11 … 99, 100, …
  - Binary – 0, 1, 10, 11, 100, … 111, 1000, … 1111, 10000, …
  - Hexadecimal – 0, 1, … F, 10, 11, … 1F, 20, … FF, 100, …

Number Systems - 3

- The base is usually appended, in subscript, to the number to indicate which number system it belongs
  - Decimal numbers – 24_d, 133_d, 1000_{10}, 3010_{10}
  - Binary numbers – 10_b, 110_b, 1000_2, 10111_2
  - Hex numbers – 24_h, 346_h, 1000_{16}, 3010_{16}
- If no base indicated, usually is decimal number

Place (Position) Value

- Decimal

Position	7	6	5	4	3	2	1	0
Value	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0
	...	...	...	10,000	1,000	100	10	1

- Binary

Position	7	6	5	4	3	2	1	0
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1

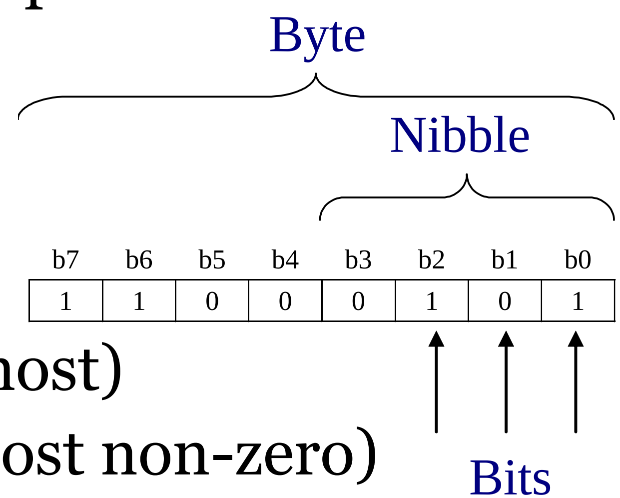
- Hexadecimal

Position	7	6	5	4	3	2	1	0
Value	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0
	...	...	...	65536	4096	256	16	1

- For base-n
  - each position to the left has n times value to its right
  - the value at x position is given by n^x
- Note the right most is smallest, and starts with n^o called least significant digit

Terms for Binary

- Bit – binary digit
- Nibble – group of 4-bit
- Byte – group of 8-bit
- Word – usually 16-bit (2 bytes); dependent on hardware
- Doubleword – usually 32-bit
- Quadword – usually 64-bit



- LSB – least significant bit (right most)
- MSB – most significant bit (left most non-zero)
- Bits are numbered from right: $\ldots b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

Decimal to Binary - 1

- Allocation based on position value
- Start with left most being the largest position value smaller than the number to be converted
- Example: convert 98_d into its binary equivalence

Position	7	6	5	4	3	2	1	0
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
		1	**1**	**0**	**0**	**0**	**1**	**0**

  - 128 > 98 > 64, start at pos 64 – enter 1
  - 98-64=34, next right position, 34>32, next at pos 32 – enter 1
  - 98-(64+32)=2, next right position, 16/8/4>2, next at pos 2 – enter 1
  - $98_d \equiv 1100010_b$

Decimal to Binary - 2

- Successive division by 2 and concatenate the remainders from each step to form the resultant binary number
- Example: convert 98_d into its binary equivalence

		Remainder
2	98	0
2	49	1
2	24	0
2	12	0
2	6	0
2	3	1
	1	

  – $98_d \equiv 1100010_b$

Binary to Decimal

- Add all position values that have 1 on them
- Example: convert 1100010_b into its decimal equivalence

Position	7	6	5	4	3	2	1	0
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
		1	**1**	**0**	**0**	**0**	**1**	**0**

  – 64+32+2=98
  – $1100010_b \equiv 98_d$
- In general, for a binary number $\ldots fedcba_b$
  – decimal = $a*2^0+b*2^1+c*2^2+d*2^3+e*2^4+f*2^5+\ldots$
  – This is called Expansion Method

Decimal to Hex - 1

- Similar concept as decimal to binary
  - allocation based on position value
  - successive division by its base, 16 and concatenate the remainders
- Example: convert 1234_d to its hex equivalence

Position	7	6	5	4	3	2	1	0
Value	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0
	...	...	...	65536	4096	256	16	1
						4	D	2

  - 4096>1234>256, start at pos 256, but 1234/256=4.8, enter 4
  - 1234-(4*256)=210, 210>16, next at pos 16, but 210/16=13.1, enter D_h (≡13)
  - 1234-(4*256+13*16)=2, last at pos 1, enter 2
  - $1234_d \equiv 4D2_h$

Decimal to Hex - 2

- Conversion using successive division by 16
  - Solving problem in previous slide

		Remainder
16	1234	2
16	77	13 (D_h)
	4	

  - $1234_d \equiv 4D2_h$

Hex to Decimal

- Similar concept with binary to decimal
  - Sum up products of the position value and the number on it
  - for a hex number $\ldots fedcba_h$
    - $decimal = a*16^0+b*16^1+c*16^2+d*16^3+e*16^4+f*16^5+\ldots$
- Example: convert $4D2_h$ into decimal
  - for calculation, use $D_h \equiv 13$
  - $4D2_h = 2*16^0+13*16^1+4*16^2=2*1+13*16+4*256=1234$
  - $4D2_h \equiv 1234_d$

Binary vs Hex

Hex is a convenient shorthand for Binary

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Binary to Hex

- Group bits by fours, starting from the right (i.e. least significant bits)
- Add leading zeros as necessary to complete the last group
- Convert each group to equivalent hex digits
- Example: convert 101001_b into hex
  - $0010{,}1001_b \rightarrow 0010_b \equiv 2_h$, $1001_b \equiv 9_h$
  - $0010{,}1001_b \equiv 29_h$

Hex to Binary

- Expand each hex digit to the equivalent 4-bit binary form
- Example: convert 29_h into binary
  - $2_h \equiv 0010_b$, $9_h \equiv 1001_b$
  - $29_h \equiv 0010{,}1001_b$

Why Hex?

- We are used to decimal. So, we need decimal
- Computer only understand binary. So, we need binary
  - however, binary is difficult (too long) to read, write and remember; e.g. $11111010001111_b = 16015_d = 3E8F_d$, it is useful to read/write in shorter form (decimal or hexadecimal)
- But, why Hex?
- Try convert the four numbers into decimal and hex: 11001101_b, 100011_b, 10111001_b, 11111100_b
  - is it easier to convert between binary and hex than binary and decimal?
  - for the four numbers, which is hardest to read, remember and write?
  - with answers to the above two questions, is hex useful?

Addition

- Same ADD algorithm for all bases
  - add digit to digit, at same value position, from right to left (from lsb to msb)
  - when the sum reaches/exceeds the base, carry to left

Adding Decimal Numbers

- Example: 1234 + 567 = 1801
  - 4 + 7 = 11
    - 11 reaches/exceeds base 10
    - therefore carry (10) to the left
    - leaving 11 – 10 = 1 at original pos
  - 1 + 3 + 6 = 10
    - 10 reaches/exceeds base 10
    - therefore carry (10) to the left
    - leaving 10 – 10 = 0
  - 1 + 2 + 5 = 8

$$
\begin{array}{lcccc}
\textit{carry:} & & 1 & 1 & \\
 & 1 & 2 & 3 & 4 \\
+ & & 5 & 6 & 7 \\
\hline
 & 1 & 8 & 0 & 1
\end{array}
$$

Adding Binary Numbers

- Example: $10001111_b + 110110_b = 11000101_b$

carry:			1	1	1	1	1		
		1	0	0	0	1	1	1	1
	+			1	1	0	1	1	0
		1	1	0	0	0	1	0	1

Adding Hex Numbers

- Example: $1234_h + 3FB_h = 162F_h$
  - $4_h + B_h = 4 + 11 = 15 = F_h$
  - $3_h + F_h = 3 + 15 = 18 = 12_h$
    - 18 reaches/exceeds base 16
    - therefore carry (16) to left
    - leaving $18 - 16 = 2 = 2_h$
  - $1_h + 2_h + 3_h = 6_h$

$$
\begin{array}{lcccc}
\textit{carry}: & & 1 & & \\
 & 1 & 2 & 3 & 4 \\
+ & & 3 & F & B \\
\hline
 & 1 & 6 & 2 & F
\end{array}
$$

Subtraction

- Same SUBTRACT algorithm for all bases
  - subtract digit by digit, at same value position, from right to left (from lsb to msb)
  - when there is not enough to subtract, borrow from the left, if left position has not enough to borrow, borrow from afar (next left to left)
  - each borrow has value equivalent to the base

Subtract Decimal Numbers

- Example: 1234 – 567 = 667
  - 4<7, borrow 1 (=10), giving 14 – 7 = 7
  - 3-1 (borrowed)<6, borrow 1 (=10), giving 12 – 6 = 6
  - 2-1 (borrowed)<5, borrow 1 (=10), giving 11 – 5 = 6

borrow:	-1	-1	-1	
value:		10	10	10
	1	2	3	4
–		5	6	7
		6	6	7

Subtract Binary Numbers

- Example: $11001010_b - 11001_b = 10110001_b$

borrow:		-1	-1				-1	
value:			2	2				2
	1	1	0	0	1	0	1	0
−				1	1	0	0	1
	1	0	1	1	0	0	0	1

Subtract Hex Numbers

- Example:
  $AB31_h - FE1_h = 9B50_h$
  - $1 - 1 = 0$
  - $3 < E$ ($\equiv 14$), borrow 1 ($=16$), giving $(16+3) - 14 = 5$
  - B ($=11$) – 1 (borrowed) < F ($\equiv 15$),
    borrow 1 ($=16$),
    giving $(16+11-1) - 15 = 11$ ($\equiv$B)
  - A – 1 (borrowed) = 9
  - mentally equate each hex to decimal, and vice versa

borrow:	-1	-1		
value:		16	16	
	A	B	3	1
–		F	E	1
	9	B	5	0

Terms in Addition and Subtraction

- $X + Y = Z$
  - X = Augend
  - Y = Addend
  - Z = Sum
  - other terms: Carry
- $X - Y = Z$
  - X = Minuend
  - Y = Subtrahend
  - Z = Difference, or Remainder (less common)
  - other terms: Borrow

Data Representation

- Integers
  - unsigned
  - Signed
    - sign & magnitude
    - 1's complement
    - 2's complement
    - biased - n
  - BCD
- Real
  - floating point
- Text
  - ASCII

- Why Data Representation?
  - computers only understand 0 and 1
  - everything else need to be represented in 0s and 1s
  - so called coding or encoding
  - the reverse process of encoding, i.e. determining the meaning of the 0s and 1s, is called decoding

Unsigned Integer

- Natural numbers, only positive
- Binary number unmodified
- All bits represent the magnitude of the number
- Minimum is zero
- Maximum depends on the size of the binary code used
  - for 1 byte (8 bits), maximum number will be 11111111_b $= 2^8 - 1 = 255$
  - for n bits code, maximum will be $2^n - 1$
- Not the most useful though most computer support

Signed Integer

- Signed integer is more important – various representations:
  - sign & magnitude
  - 1's complement
  - 2's complement
  - biased - m
- 2's complement most common – implemented in most computers for arithmetic

Sign & Magnitude

- Leftmost ("most significant") bit represents the sign of the integer: 0 is +ve, 1 is –ve
- Remaining bits to represent its magnitude
- Two representations for zero: usually use the all 0s, i.e. $000\ldots000_b$
- Range for n bits:
  $-(2^{n-1}-1) \leq \text{S \& M} \leq +(2^{n-1}-1)$
- Example: $-7 \leq$ 4-bit S & M $\leq +7$; $2^{4-1}-1 = 7$

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign & Magnitude	+0	+1	+2	+3	+4	+5	+6	+7	–0	–1	–2	–3	–4	–5	–6	–7

1's Complement - 1

- Leftmost ("most significant") bit represents the sign of the integer: 0 is +ve, 1 is –ve
- Remaining bits to represent its magnitude
- Negative numbers are the complement of the positive numbers
- Two representations for zero: usually use the all 0s, i.e. $000\ldots000_b$
- Range for n bits (same as S & M):
  $-(2^{n-1}-1) \leq 1\text{'s} \leq +(2^{n-1}-1)$

1's Complement - 2

- Example: $-7 \leq$ 4-bit 1's $\leq +7$; $2^{4-1}-1 = 7$

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1's Complement	+0	+1	+2	+3	+4	+5	+6	+7	–7	–6	–5	–4	–3	–2	–1	–0

- Encoding by example: for 4-bit 1's Complement code, determine the code for -6
  - for positive number, simply convert to binary (use only n-1 bits)
  - for 4-bit, $+6 \equiv 0110_b$ (note MSB is 0 for +ve number)
  - complement each bit of +6 gives:
  - $-6 \equiv 1001_b$ (note MSB is 1)

+6 =	0	1	1	0
1's =	1	0	0	1

2's Complement - 1

- Leftmost ("most significant") bit represents the sign of the integer: 0 is +ve, 1 is –ve
- Remaining bits to represent its magnitude
- Only one bit pattern for zero
- Most useful property: $X - Y = X + (-Y)$
- no need for a separate subtractor (S & M) or carry-out adjustments (1's Complement)
- Range for n bits (one extra negative number): $-2^{n-1} \leq 2's \leq +(2^{n-1}-1)$

2's Complement - 2

- Example: $-8 \le$ 4-bit 2's $\le +7$; $2^{4-1} = 8$ and $2^{4-1}-1 = 7$

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2's Complement	+0	+1	+2	+3	+4	+5	+6	+7	–8	–7	–6	–5	–4	–3	–2	–1

- The 2's codes for x and -x add to a power of 2
  - 4-bit code: $c+(-c)=2^4$
  - 8-bit code: $c+(-c)=2^8$
- Mathematically $x+(-x) = 0$, then $2^n \equiv 0$ giving:
  $(-c) = 0-c = 2^n-c = [(2^n-1)-c]+1$
  - Note that (2^n-1) is $1111..1_b$, making subtraction a cinch!
  - Roles of (-c) and c can be reversed
  - 2^n-c → Change Sign Rule I, $[(2^n-1)-c]+1$ → Change Sign Rule II

2's Complement - 3

- Change Sign Rule I
  - Subtract from 2^n
- Change Sign Rule II (recommended)
  - Flip all the bits
  - Add 1
- Change Sign Rule III
  - Scan right to left to the first bit with value 1
  - Flip all bits to its left
- Encoding 2's:
  - for positive number: simply convert to binary (use only n-1 bits, with MSB as 0)
  - for negative number: apply either of the 3 change sign rules to the positive code

2's Complement - 4

- Encoding example: assuming 4-bit code, convert 4, 6, -6, -7 into 2's complement code
  - positive numbers: simply convert to binary
    - $4 \equiv 0100_b$, $6 \equiv 0110_b$; note MSB is 0
  - negative numbers: convert its positive value to binary and apply sign change (any 1 rule)
    - $-6 \rightarrow 6 \equiv 0110_b$ $\underline{\rightarrow \text{flip all bits} \rightarrow}$ 1001_b $\underline{\rightarrow \text{add 1} \rightarrow}$ 1010_b; $-6 \equiv 1010_b$
    - $-7 \rightarrow 7 \equiv 0111_b$ $\underline{\rightarrow \text{flip all bits} \rightarrow}$ 1000_b $\underline{\rightarrow \text{add 1} \rightarrow}$ 1001_b; $-7 \equiv 1001_b$
    - note MSB is 1 for negative numbers

2's Complement - 5

- Decoding 2's:
  - for positive number: leading 0 indicates value is positive – simply convert to decimal
  - for negative number: leading 1 indicates value is negative – apply change sign rule, then convert to decimal (remember the negative sign)
- Decoding example: assuming 4-bit 2's code, determine the decimal equivalent of 0101_b, 0111_b, 1011_b, 1110_b
  - positive numbers (MSB is 0): simply convert binary to decimal
    - $0101_b \equiv 5$, $0111_b \equiv 7$
  - negative numbers (MSB is 1): change sign to positive and then convert to decimal
    - 1011_b $\underline{\rightarrow \text{flip all bits} \rightarrow}$ 0100_b $\underline{\rightarrow \text{add 1} \rightarrow}$ $0101b \equiv 5$; $1011_b \equiv -5$
    - 1110_b $\underline{\rightarrow \text{flip all bits} \rightarrow}$ 0001_b $\underline{\rightarrow \text{add 1} \rightarrow}$ $0010b \equiv 2$; $1110_b \equiv -2$
    - remember the negative sign

Biased – m (Excess – m)

- Integer N represented by $N + m$
- For n bits, normally use $m = 2^{n-1}$ (half range $2^n/2$)
- Like 2's complement, asymmetric
- Used when important to compare and sort numbers
- Example: for 4-bit code, $m = 2^{4-1} = 8$
  - 0 is represented by $0+8 = 8 \equiv 1000_b$
  - -8 is represented by $-8+8 = 0 \equiv 0000_b$ (smallest)
  - 7 is presented by $7+8 = 15 \equiv 1111_b$ (largest)

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bias-8	–8	–7	–6	–5	–4	–3	–2	–1	0	1	2	3	4	5	6	7

Binary Coded Decimal (BCD)

- Use 4 bits (1 nibble) to represent each decimal digit – direct binary-decimal conversion
- Easy for human to understand
- Wastes some bit patterns (can use one of them for sign)
- Not efficient for storage

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BCD	0	1	2	3	4	5	6	7	8	9	-	-	-	-	-	-

Summary of Integers
4-bit Code Representations

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign & Magnitude	+0	+1	+2	+3	+4	+5	+6	+7	–0	–1	–2	–3	–4	–5	–6	–7
1's Complement	+0	+1	+2	+3	+4	+5	+6	+7	–7	–6	–5	–4	–3	–2	–1	–0
2's Complement	+0	+1	+2	+3	+4	+5	+6	+7	–8	–7	–6	–5	–4	–3	–2	–1
Excess-8	–8	–7	–6	–5	–4	–3	–2	–1	0	1	2	3	4	5	6	7
BCD	0	1	2	3	4	5	6	7	8	9	-	-	-	-	-	-

Floating Point - 1

- All previous representations only encode integers (whole numbers)
- Floating point numbers are real numbers, i.e. with decimal point, in binary
  - format: $\pm 1.\text{xxxxxx}\ldots \times 2^{\text{yyyy}\ldots}$
- In computer, floating point numbers are stored with 3 data – sign, mantissa and exponent
  - format: $-1^{S} \times M \times 2^{E}$
    - S=sign, M=mantissa (1.xxxx...), E=exponent (yyyy...)
    - exponent is represented in bias-m

Floating Point - 2

- Single-precision floating point numbers:
  - occupy 32 bits, give approx range of $\pm 10^{-38} \ldots 10^{38}$
  - exponent encoded in bias-127 ($2^{n-1} - 1$)
- Double-precision floating point numbers:
  - occupy 64 bits, give approx range of $\pm 10^{-308} \ldots 10^{308}$
  - Exponent encoded in bias-1023 ($2^{n-1} - 1$)
- *More on this in tutorial*

Bit No	Size	Field Name
31	1 bit	Sign (S)
23-30	8 bits	Exponent (E)
0-22	23 bits	Mantissa (M)

Bit No	Size	Field Name
63	1 bit	Sign (S)
52-62	11 bits	Exponent (E)
0-51	52 bits	Mantissa (M)

ASCII

- ASCII ≡ American Standard Code for Information Interchange
- Representation of non-numerical data, i.e. character encoding
- Use 7-bit code to represent 128 characters (including control characters, e.g. line feed)
- In byte data system, MSB set as 0 or used as parity bit for error checking

ASCII Table

Low 4 Bits	High 3 Bits							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EMT	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	&	n	~
1111	SI	US	/	?	O	_	o	DEL

2's Complement Addition - 1

- Adding n-bit 2's Complement codes gives an n-bit result
  - use the coded representations, treating them as unsigned values (normal binary)
  - add the values and discard any carry-out bit
- Overflow rule for addition:
  - overflow occurs if (check MSB, i.e. sign bit)
    - $(+A) + (+B) = -C$
    - $(-A) + (-B) = +C$
- Overflow – result exceeds range

2's Complement Addition - 2

- Examples (4-bit 2's code):
- $4 + 3 = ?$

	0	1	0	0	≡	4
+	0	0	1	1	≡	3
	0	1	1	1	≡	7

- $2 + (-8) = ?$

	0	0	1	0	≡	2
+	1	0	0	0	≡	-8
	1	0	1	0	≡	-6

- $5 + 7 = ?$

	0	1	0	1	≡	5
+	0	1	1	1	≡	7
	1	1	0	0	≡	-4

- above result shows overflow - incorrect

2's Complement Subtraction - 1

- Subtracting n-bit 2's Complement codes gives an n-bit result
  - use the coded representations, treating them as unsigned values (normal binary)
  - change the sign and add
    - X – Y = X + (–Y), i.e. obtain –Y from Y first
- Overflow rule for subtraction:
  - overflow occurs if (check MSB, i.e. sign bit)
    - (+A) – (–B) = –C
    - (–A) – (+B) = +C

2's Complement Subtraction - 2

- Examples (4-bit 2's code):
- $4 - \mathbf{3} = 4 + (-3) = ?$

	0	1	0	0	≡	4
+	1	1	0	1	≡	-3
(1)	0	0	0	1	≡	1

- $5 - \mathbf{7} = 5 + (-7) = ?$

	0	1	0	1	≡	5
+	1	0	0	1	≡	-7
	1	1	1	0	≡	-2

- $2 - \mathbf{(-8)} = 2 + 8 = ?$

	0	0	1	0	≡	2
+	?	?	?	?	≡	8
	?	?	?	?	≡	?

- there is no representation for +8 in 4-bit 2's
- what will happen?

Summary

- Computers are made up of logic circuits
  - Logic operations AND, OR, NOT, XOR, NAND, NOR recapped
- Computers only understand 0s and 1s, therefore need to know binary and other related matters
  - number systems recapped: binary, hexadecimal
  - data representation: integers (unsigned, S&M, 1's, 2's, bias-m, BCD, floating point, ASCII)
  - arithmetic on 2's – most useful representation for integers