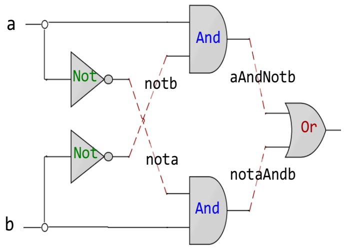
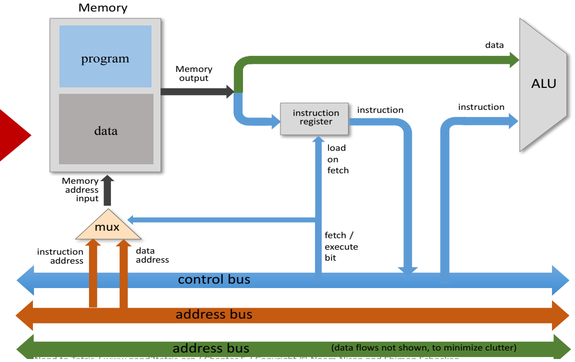
Computer Organization & Assembly Language Programming




```
CHIP Xor {
   IN a, b;
   OUT out;
   PARTS:
   Not(in=a, out=nota);
   Not(in=b, out=notb);
   And(a=nota, b=b, out=w1);
   And(a=a, b=notb, out=w2);
   Or(a=w1, b=w2, out=out);
}
```




```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

Lecture # 06

Data Storage - I

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

Slides of first half of the course are adapted from:
https://www.nand2tetris.org
Download s/w tools required for first half of the course from the following link:
https://drive.google.com/file/d/0B9c0BdDJz6XpZUh3X2dPR1o0MUE/view

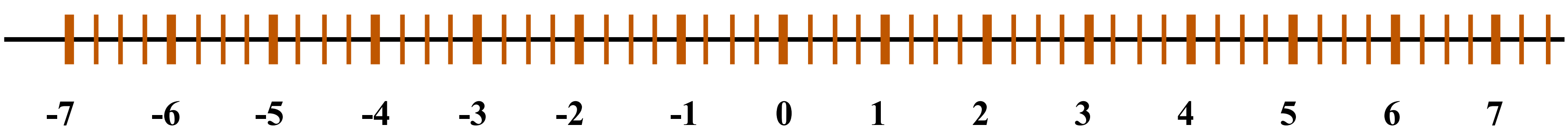
Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- Data Representation in Computers
- Unsigned Numbers
- Signed Numbers
  - Sign magnitude representation & its limitations
  - 1s Complement representation & its limitations
  - 2s Complement
  - Comparisons and pros and cons of each
- Ranges and different Storage Sizes
- Overflow in Unsigned & Signed Numbers
- How the Hardware Detect an Overflow
- Concept of Sign Extension
- Encoding Characters and Strings (ASCII & Unicode)

Different Types of Numbers

- Natural Numbers (**N**): Set of positive numbers
- Whole Numbers (**W**): Set of zero and positive natural numbers
- Integers (**Z**): Set of zero, positive natural numbers and their additive inverses. An integer is a number that can be written without a fractional component
- Real Numbers (**R**): A continuous quantity that can represent a distance along a line (They are called real because they are not imaginary)
- Imaginary Numbers are numbers that when squared gives use a negative number, e.g., sqrt(-1)
- Rational numbers (**Q**): are numbers that can be expressed as ratio of two integers, e.g., $\frac{1}{2}$ and $\frac{2}{4}$ are two fractions that represent the same rational number 0.5
- Irrational Numbers (**Q'**): are numbers that cannot be expressed as ratio of two integers, e.g., $3.14159265358979323846\mathrm{2}$ which is not exactly equal to $\frac{22}{7}$



Note:
- Most of the programming languages provide support for storing and manipulating rational numbers
- In Computers irrational numbers cannot be fully and accurately represented/manipulated

Unsigned Numbers

Unsigned Numbers

Base 10 number representation (Decimal)

$$521_{10} = 5x10^2 + 2x10^1 + 1x10^0 = 521_{10}$$

Base 2 Number Representation (Binary)

$$1011_2 = 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 = 11_{10}$$

Base 16 Number Representation (Hexadecimal)

$$9E_{16} = 10011110_2$$

Base 8 Number Representation (Octal)

$$46_8 = 100110_2$$

Students should know how to convert a number from one base to another

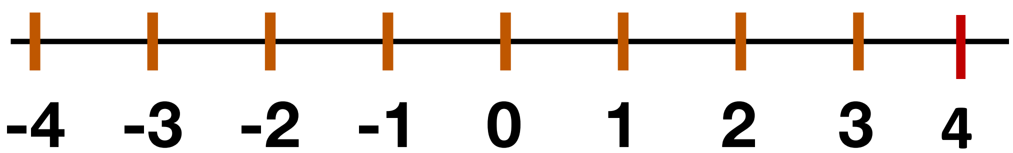
Decimal	Hex	Octal	Binary
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111

Note: These all are weighted and positional number systems, with each bit having a weight depending on its position

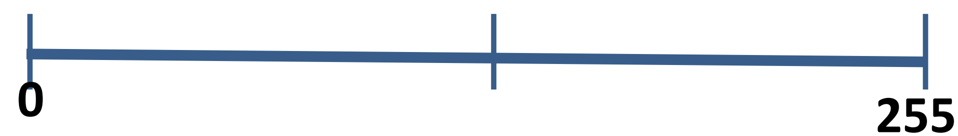
Encoding Signed Numbers

Encoding Signed Numbers

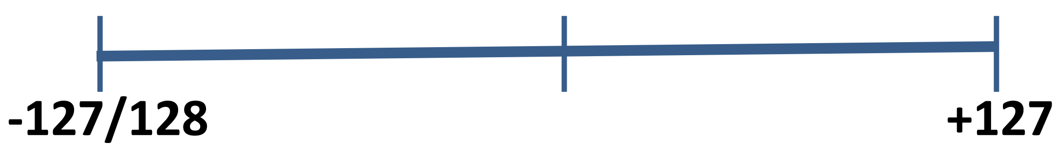
- Theoretically there are three ways to encode the signed numbers:
  - ➢ Sign Magnitude Encoding
  - ➢ 1's Complement Encoding
  - ➢ 2's Complement Encoding



- Unsigned byte range can be represented using a number line as below:



- Signed byte range can be represented using a number line as below:



Sign Magnitude Encoding

How to Encode a Negative Number:

- The most natural way of encoding a signed number is by its sign and magnitude
- MSb is reserved to represent/encode the sign. 0 for positive and 1 for negative and the remaining bits represents the magnitude
- The four bits representations of signed numbers using sign magnitude encoding is shown in the table

Decimal	Binary Bits
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

Sign Magnitude Encoding (cont…)

Limitations:

- Two different encodings for zeros (positive & negative)

  +0 = 0000 and -0 = 1000

- Subtraction can't be done using addition, e.g.:

  +2 + (-3) = -1

```
   0010          2
+) 1011      +) -3
-------      -----
   1101         -5
```

- How to do subtraction using Sign Magnitude?
  - ➢ If the numbers have same sign, add magnitudes and keep the sign
  - ➢ If the numbers have different signs, then subtract the smaller magnitude from the larger one. The sign of the larger magnitude is the sign of the result
  - ➢ Note: So you need a separate hardware for subtraction

Decimal	Binary Bits
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

1's Complement Encoding

How to Encode a Negative Number:

- Take 1's complement of the positive number to represent it's corresponding negative number
- The four bits representations of signed numbers using 1's complement encoding is shown in the table
- Whenever, a signed number has its MSb as 1, that means it is a negative number. So take its 1's complement and represent it with a negative sign

Decimal	Binary Bits
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	1111
-1	1110
-2	1101
-3	1100
-4	1011
-5	1010
-6	1001
-7	1000

1s Complement Encoding (cont…)

Limitations:

- Two different encodings for zeros (positive & negative)

  +0 = 0000 and -0 = 1000

- You can do the subtraction using addition, however, doesn't always work:

  +1 + (-1) = 0

```
  0001            1
+)1110        +) -1
------        -----
  1111           -0
```

Decimal	Binary Bits
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	1111
-1	1110
-2	1101
-3	1100
-4	1011
-5	1010
-6	1001
-7	1000

2s Complement Encoding

How to Encode a Negative Number:

- Take 2's complement of the positive number to represent it's corresponding negative number
- The four bits representations of signed numbers using 2's complement encoding is shown in the table
- Whenever, a signed number has its MSb as 1, that means it is a negative number. So take its 2's complement and represent it with a negative sign

Decimal	Binary Bits
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
+/-0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

2s Complement Encoding (cont…)

Limitations Resolved:

- Single encoding for zero (no concept of negative zero)

  +0 = 0000 and -0 = 0000

- Subtraction can be done using addition, so you don't need a separate hardware for subtraction. For example:

  +1 + (-1) = 0

  ```
    0001         1
  +) 1111    +) -1
    0000         0
  ```

  +2 + (-3) = -1

  ```
    0010         2
  +) 1101    +) -3
    1111        -1
  ```

- 7+1 becomes -8 (called overflow. More on it later)

  ```
     0111          7
  +) 0001     +)   1
     1000         -8
  ```

Decimal	Binary Bits
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
+/-0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Comparison of 4 bit Signed and Unsigned Numbers

Binary Bits	Unsigned	SM	1s Comp	2's Comp
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

Mapping Signed ↔ Unsigned

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

=

- 16

+ 16

Ranges of Signed Numbers

Range for Unsigned Numbers:

0 to $2^n - 1$

Range for signed Numbers (2's Comp):

-2^{n-1} to $2^{n-1} - 1$

Range for signed Numbers (SM & 1's Comp):

$-(2^{n-1} - 1)$ to $2^{n-1} - 1$

Note: Since 2's complement has only one way of representing/encoding zero, so we have one additional number on the negative side

Decimal	2s Comp	1s Comp	SM
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
-0	0000	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	-	-

Integer Ranges with Different Storage Sizes

Storage	Minimum	Maximum
Unsigned (8 bits)	0	255
Signed (8 bits)	-128	127
Unsigned (16 bits)	0	65535
Signed (16bits)	-32768	32767
Unsigned (32 bits)	0	4294967295
Signed (32bits)	-2147483648	2147483647
Unsigned (64 bits)	0	18446744073709551615
Signed (64 bits)	-9223372036854775808	9223372036854775807

The range of 64 bit integers is large enough for most needs. Of course there are exceptions, like 20! = 5109094217170944000

Overflow after Addition When using 2's Complement Encoding

Overflow in Unsigned Addition

- Overflow is a condition that occurs when a calculation produces a result that is greater in magnitude than what a given register or a storage location can store
- An overflow can be detected by the hardware if there is a carry out from the most significant bit after addition (Check Carry Flag after addition, if set then overflow)
- Consider addition of two 4-bit unsigned numbers:

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

Normal Case:

```
     1001         9
+)   0101    +)   5
   ------    ------
     1110        14
```

Overflow Case:

```
     1010        10
+)   0111    +)   7
   ------    ------
    10001        17
     0001         1
```

Overflow in Signed Addition

- Overflow will never occur when you add a positive number to a negative number. It will occur only when the two operands have same sign, but the result hasn't
- Overflow will occur when you add two negative numbers and get a positive result called Negative Overflow

```
    1010          -6
+)  1001      +)  -7
--------      ------
   10011         -13
    0011           3
```

There is carry out from the MSb, so, an overflow has occurred, because **0011** means **+3**, when evaluated in 2's complement

- Overflow will occur when you add two positive numbers and get a negative result called Positive Overflow

```
    0110           6
+)  0101      +)   5
--------      ------
    1011          11
```

There is no carry out from the MSb, however, an overflow has occurred, because **1011** means **-5**, when evaluated in 2's complement

Decimal	Binary
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Is This Signed Addition an Overflow?

- Consider the following example in which two four bit numbers are added. There is a carry out from the MSb and the result is in 5 bits. Is this an example of overflow:

```
     1111
 +)  1110
 ---------
 [1] 1101
```

- This is not an overflow by definition. Because even after truncating the 5 bits result in 4 bits (bit width of the datatype) the result is correct

```
     1111                      -1
 +)  1110                 +)   -2
 ---------                ---------
 [1] 1101  ---------->         -3
           Truncate
```

- **Sign Extension:** It is the concept of increasing the number of bits of a binary number while preserving its sign and magnitude. This can be done by padding the left side with sign bit

Decimal	Binary
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

How does the Hardware Detect an Overflow?

- Detecting overflow after adding two unsigned numbers:
  - This can be detected by the hardware if there is a carry out from the most significant bit (Check Carry Flag (CF) after addition, if set then overflow)
- Detecting overflow after adding two signed numbers:
  - This can be detected by the hardware if the carry-in in the MSb and carry-out from the MSb are different (Check Overflow Flag (OF) after addition, if set then overflow)
- Remember, the hardware is responsible for setting /resetting these two flags
- For 4 bits signed numbers (in 2s complement representation) detect the overflow in following examples:

```
 [1]              [0]              [1]              [1]              [0]
    1111             1010             0110             1110             0010
+)  1110         +)  1001         +)  0101         +)  0101         +)  0101
---------        ---------        ---------        ---------        ---------
[1] 1101         [1] 0011         [0] 1011         [1] 0011         [0] 0111
```

Encoding Characters/Strings Inside Computers

Representing Characters And Strings (ASCII)

- The ASCII code is used to give to each symbol / key from the keyboard a unique number called ASCII code
- It can be used to convert text into ASCII code and then into binary code
- The 8-bit ASCII table contains 256 codes (from 0 to 255)
- This slide shows some common ASCII codes

Char	ASCII Code (Decimal)
a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122

Char	ASCII Code (Decimal)
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90

Char	ASCII Code (Decimal)
space	32
!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47
:	58
;	59
<	60
=	61
>	62
?	63
@	64
[	91
\	92
]	93
^	94
_	95
`	96
{	123
|	124
}	125
~	126
‘	145
’	146
“	147
”	148
•	149
˜	152

Char	ASCII Code (Decimal)
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

Char	ASCII Code (Decimal)
€	128
£	163
¥	165
$	36
©	169
™	153
°	176
˜	152
¡	161
¿	191

Representing Characters And Strings (Unicode)

- Today the Unicode Standard is the universal character-encoding standard used for representation of text for computer processing
- Unlike 7-bit standard ASCII, which can encode the English language alphabets only, Unicode can encode a variety of languages spoken around the world
- The Unicode is a standard scheme for representing plain text, however, it is not a scheme for representing rich text
- Unicode is platform, program, and language independent
- The common encoding formats used by Unicode are UTF-8, UTF-16 and UTF-32 (Unicode Transformation Format)
- UTF-8 is the default encoding form for a wide variety of Internet standards and uses one byte. The first 128 Unicode code points represent the ASCII characters, which means that any ASCII text is also a UTF-8 text
- The W3C (World Wide Web Consortium) specifies that all XML processors must read UTF-8 and UTF-16 encoding

Things To Do

- Practice converting signed and unsigned numbers from one base to another base, e.g., decimal, binary, octal, hex. Confirm your working by using online base conversion calculators:

https://www.branah.com/ascii-converter

https://www.binaryconvert.com/index.html

- Write down a C program that checks the minimum and maximum value that can be stored in signed and unsigned data types like `char`, `short`, `int`, `long`, and `long long`. Does this has something to do with the h/w and operating system (32 bit or 64 bit)
- Write down a C program that verify as the what happens when a signed or unsigned variable of char data type overflows

Coming to office hours does NOT mean you are academically week!