UESTC2004: Embedded Processors

Semester 2 – 2020/2021

Data Representations – *Part 1*

Lecture 7

Dr. Lina Mohjazi (Lina.Mohjazi@glasgow.ac.uk)
Lecturer
Glasgow College, UESTC
James Watt School of Engineering

➢Objectives - To understand:

- ❖ Base of number systems: decimal, binary, octal and hexadecimal
- ❖ Textual information stored as ASCII
- ❖ Binary addition/subtraction
- ❖ Binary logical operations
- ❖ Unsigned and signed binary number systems
- ❖ Fixed point binary representations
- ❖ Floating point representations
- ❖ Encoding schemes

➢By the end of the lecture, you should be able to:

- ❖ Convert between numbers represented in different bases
- ❖ Perform simple binary arithmetic and logical operations
- ❖ Read and interpret hexadecimal numbers with reasonable speed

What Does Data Mean?

Definition

- *<u>In General:</u>* Data refers to the information that represent people, events, things, and ideas. Data can be a name, a number, the colors in a photograph, or the notes in a musical composition.

- *<u>In Digital World</u>:* Data is information processed, transmitted or stored by electronic devices. This information may be in the form of digital photographs, music files, software programs, word processing documents, PDFs and countless other types of data.

Data Representation

- The form in which data is stored, processed, and transmitted.

- Data need to be represented in a convenient way that simplifies:
  – common operations: addition, comparison, etc.
  – hardware implementation (cheap, fast, reliable)

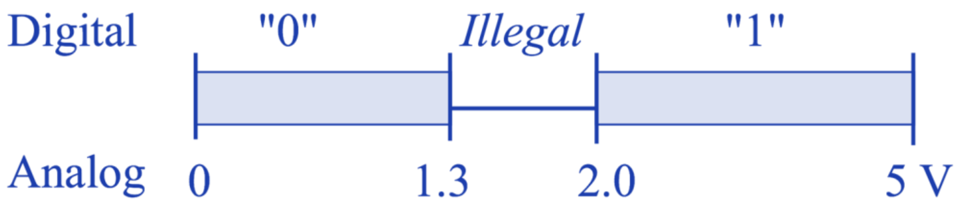
Why Learn Data Representation?

- Data representation is a major part of the software-hardware interface.

- It is important to understand data representation to write correct and high-performance programs.

Electronic Representation of Data

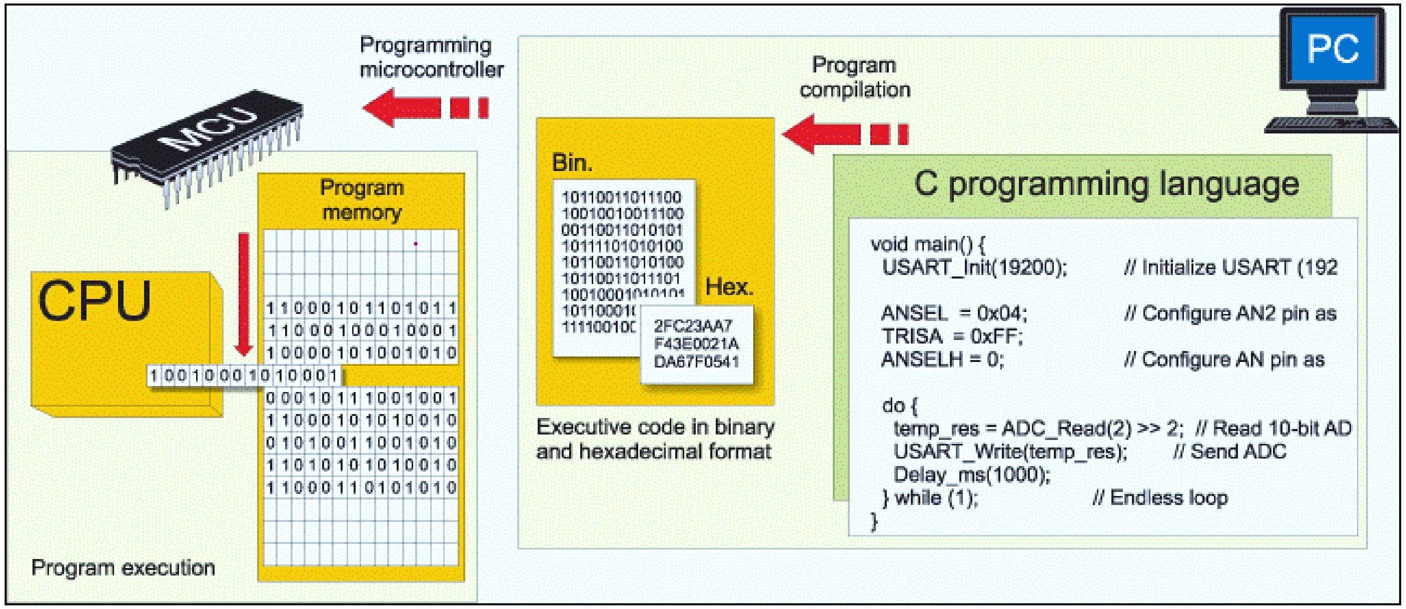
- Electronic data representation has to be simple so that it can be handled by electronic circuitry in digital devices.
- Information can be represented by a voltage level.
- A voltage signal which has only two possibilities (**HIGH** or **LOW**) is a bit– stands for *Bi*nary Digi*t*.
- A bit is a **0** or **1** used in the digital representation of data.
- *Advantage:* easy to do computation, very reliable, simple to implement in electronic hardware (switch).
- *Disadvantage:* too little information per bit, must use many of them.
- On the Cortex-M microcontroller, a voltage between 2 and 5V is considered high, and a voltage between 0 and 1.3V is considered low.



- Different digital logic families assign a different range of voltages to be equal to a logical “0” and a logic “1”.

Number Systems

Compilers and assemblers are used to convert high-level language and assembler language codes into a compact machine code for storage in the microcontroller's memory



```
void main() {
  USART_Init(19200);          // Initialize USART (192

  ANSEL  = 0x04;              // Configure AN2 pin as
  TRISA  = 0xFF;
  ANSELH = 0;                 // Configure AN pin as

  do {
    temp_res = ADC_Read(2) >> 2;  // Read 10-bit AD
    USART_Write(temp_res);        // Send ADC
    Delay_ms(1000);
  } while (1);                // Endless loop
}
```

- Binary data and instructions are stored through billions of tiny transistors.
- Each transistor can either exist in an **ON** or **OFF** state.
- An OFF state is considered a "0", and an ON state a "1".

Number Systems Overview

Number Systems

Binary
- Base value: 2
- Used in digital devices.
- Uses the digits {0,1}
- E.g. $(1011)_2$
- $(\mathbf{01101010})_2=\mathbf{0}*2^7+\mathbf{1}*2^6+\mathbf{1}*2^5+\mathbf{0}*2^4+\mathbf{1}*2^3+\mathbf{0}*2^2+\mathbf{1}*2^1+\mathbf{0}*2^0=64+32+8+2=106$

Octal
- Base value: 8
- Used to present binary numbers in a compact form.
- Uses the digits {0,1,3,4,5,6,7}
- Denoted as $(\mathbf{5643})_8$

Decimal
- Base value: 10
- Used by human beings for counting and measurements
- Uses the digits {0,1,2,3,4,5,6,7,8,9}
- E.g. $(\mathbf{1874})_{10}$

Hexadecimal
- Base value: 16
- Used to present binary numbers in a compact form.
- Uses the digits {0,1,3,4,5,6,7,8,9,A,B,C,D,E,F}
- Denoted as $(\mathbf{45AC})_{16}$ or 0x45AC in C language

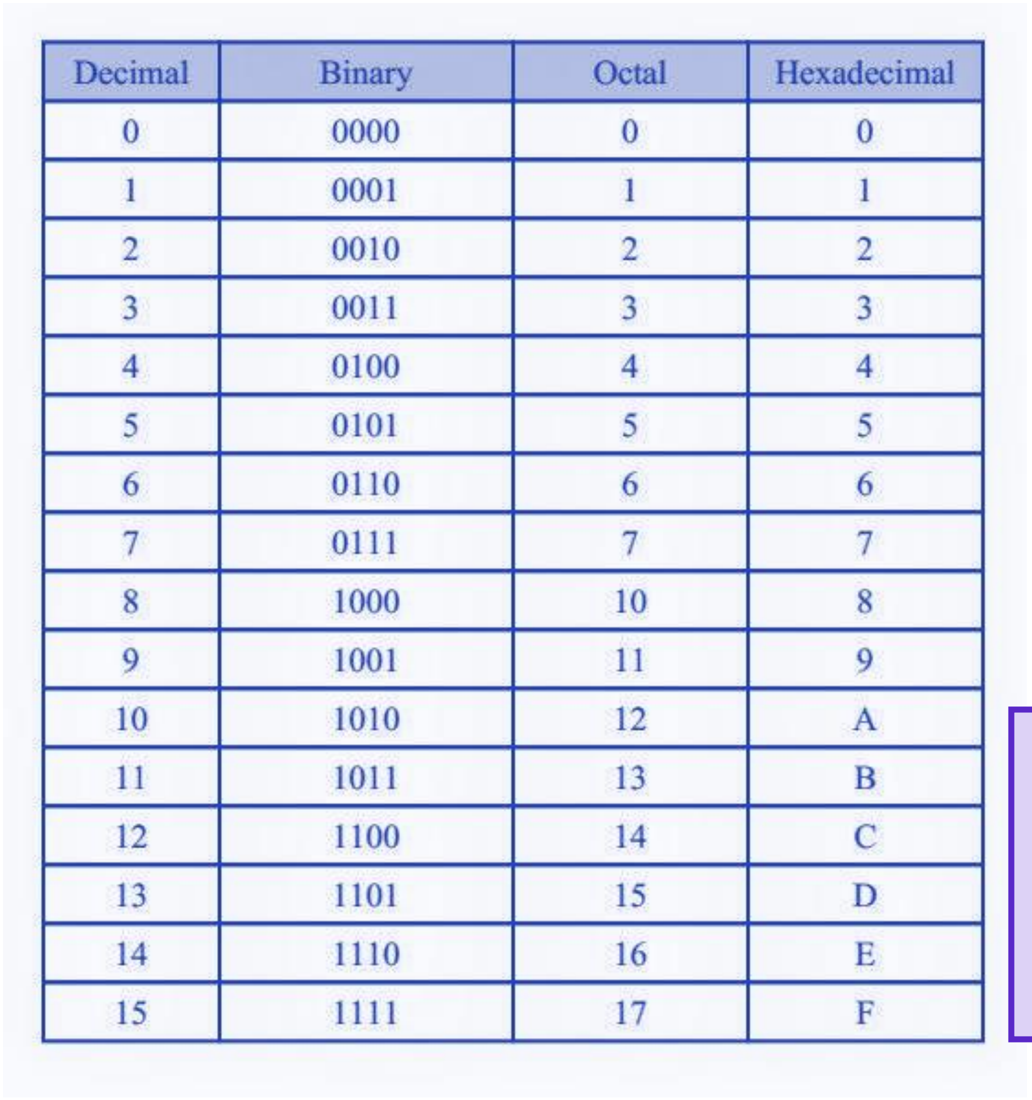
Why Use Octal or Hexadecimal?

- Writing or reading a long sequence of binary bits is cumbersome and error-prone!
- Try to read this binary string: $(10110011010000110001110100011000)_2$, which is the same as hexadecimal $(B3431D18)_{16}$ or octal $(13120616430)_8$

To make life easier:

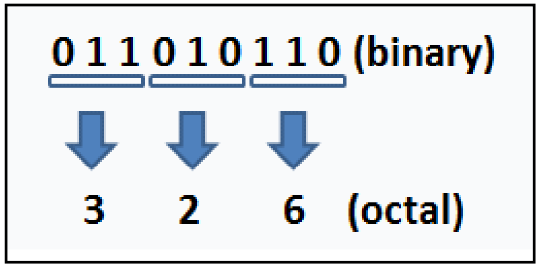
- Large binary numbers are split up into groups to make them more convenient to write, understand, and handle.
- For example, the group of binary digits $(1101\ \ 0101\ \ 1100\ \ 1111)_2$ or $(001\ 101\ \ 010\ 111\ 001\ 111)_2$ are much easier to read and understand than $(1101010111001111)_2$
- $(001\ 101\ 010\ 111\ 001\ 111)_2 = (152717)_8$
  - The **Octal** Numbering System breaks a binary number into groups of **three** and represents each group of three binary numbers with **one** octal number. **Can you guess why?**
- $(1101\ 0101\ 1100\ 1111)_2 = (D5CF)_{16}$
  - The **Hexadecimal** Numbering System breaks a binary number into groups of **four** and represents each group of four binary numbers with **one** hexadecimal number. **Can you guess why?**

Conversion from Binary to Octal or Hexadecimal

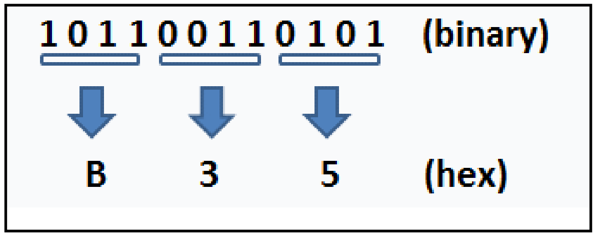


Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Binary to Octal Conversion



Binary to Hexadecimal Conversion



The use of octal numbers has declined as most modern computers no longer base their word length on multiples of three bits, (they are based on multiples of four bits, so hexadecimal is more widely used).

Advantages and Uses of Hexadecimal Numbers

Advantages

- Concise. The number of digits used to signify a given number is usually less than in binary, decimal, or octal.

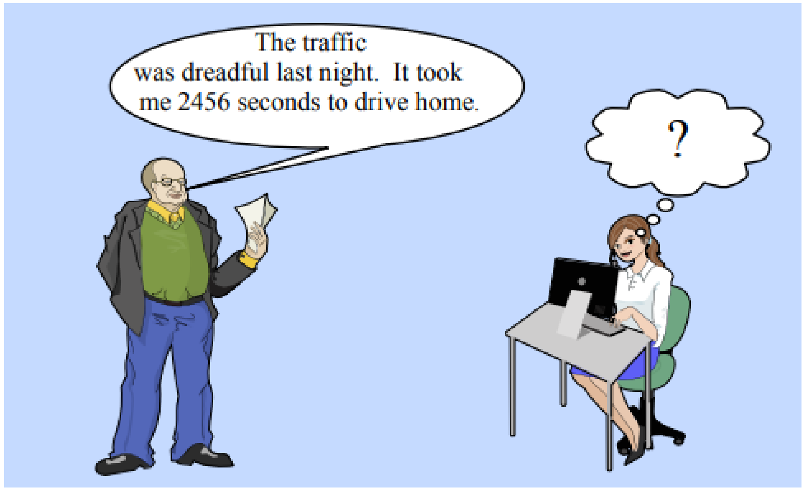
- Fast and simple to convert between hexadecimal numbers and binary.

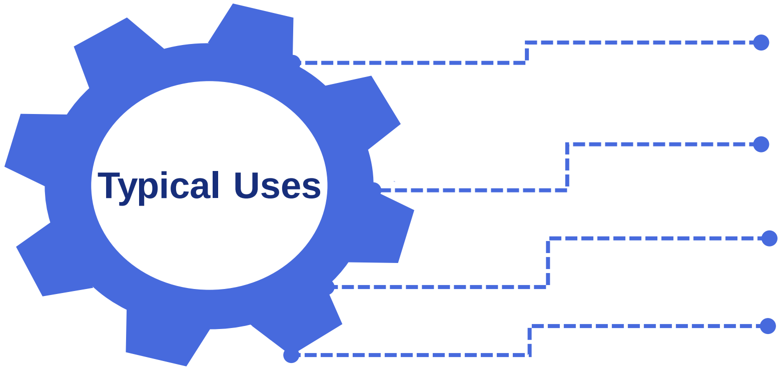
- Hexadecimal can be used to write large binary numbers in just a few digits.

  It allows you to store more information using less space.

- Human-friendly, as humans are used to grouping together numbers and things for easier understanding.

- Lowers the possibility of error occurring.


The traffic was dreadful last night. It took me 2456 seconds to drive home.
?

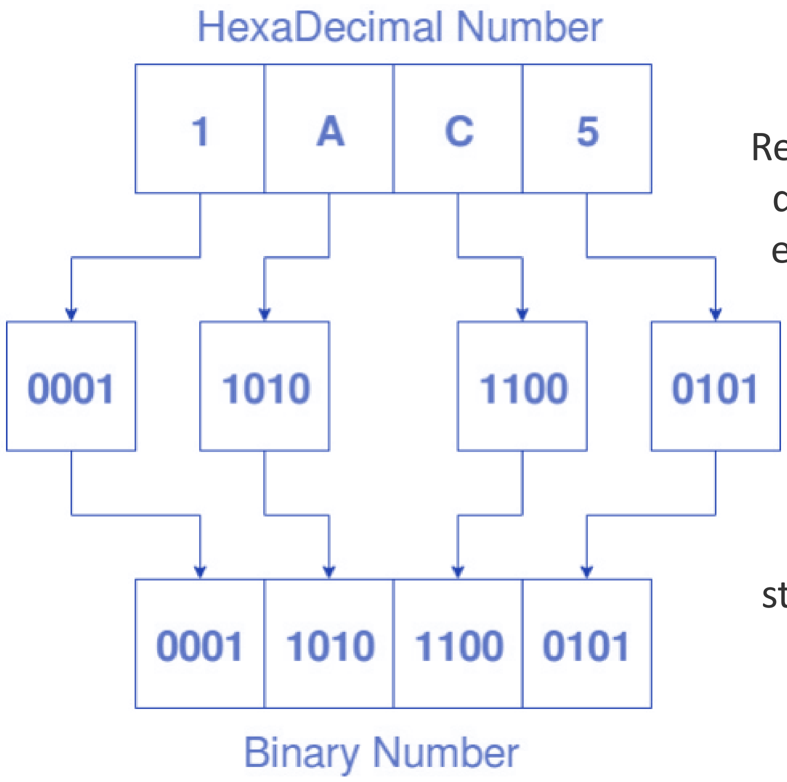


- To define memory locations
- To define colors on webpages
- To represent Media Access Control (MAC) addresses
- To display error messages

Conversion from Hexadecimal to Binary

Binary	Hex
0000	0x0
0001	0x1
0010	0x2
0011	0x3
0100	0x4
0101	0x5
0110	0x6
0111	0x7
1000	0x8
1001	0x9
1010	0xA
1011	0xB
1100	0xC
1101	0xD
1110	0xE
1111	0xF

Nibble



Represent each hex digit by its binary equivalent nibble

Concatenate or string all the digits together

0x1AC5 = $(1101011000101)_2$

Discard any leading zeros at the left of the binary number

Conversion from Base r to Decimal & from Decimal to Base r

Conversion from Base r to Decimal

Given an ***n***-digit base ***r*** number:
$d_{n-1}d_{n-2}d_{n-3}...d_2d_1d_0$ (base r), the decimal equivalent is given by:

$$d_{n-1}* r^{n-1} + d_{n-2}* r^{n-2} + ... + d_1* r^1 + d_0* r^0$$

For example:
$(A1C2)_{16} = 10*16^3 + 1*16^2 + 12*16^1 + 2*16^0 = (41410)_{10}$
$(10110)_2 = 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = (22)_{10}$

Conversion from Decimal to Base r

Use repeated division/remainder.

For example:
To convert $(261)_{10}$ to hexadecimal:
261/16 => quotient=16 remainder=5
16/16 => quotient=1 remainder=0
1/16 => quotient=0 remainder=1 (quotient=0 stop)
Hence, $(261)_{10} = (105)_{16}$ (Collect the hex digits from the remainder in reverse order)

Conversion from Base r to Decimal & from Decimal to Base r

Conversion from Decimal to Base 2

- Repeatedly divide the Decimal Integer by 2. Each remainder is a binary digit in the translated value:

Division	Quotient	Remainder
37 / 2	18	1
18 / 2	9	0
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1

least significant bit

most significant bit

stop when quotient is zero

$37_{10} = 100101_2$

Conversion from Base r to Decimal & from Decimal to Base r

Let's play a game!

Go to: https://www.menti.com

Enter the code: 9349 2689

Conversion from Base r to Decimal & from Decimal to Base r

Try it yourself!

- Convert the decimal number 108 into:
  - Binary number.
  - Hexadecimal number.

- Convert the binary number 1000011000 into:
  - Hexadecimal number.
  - Decimal number.

Conversion from Base r to Decimal & from Decimal to Base r

Try it yourself!

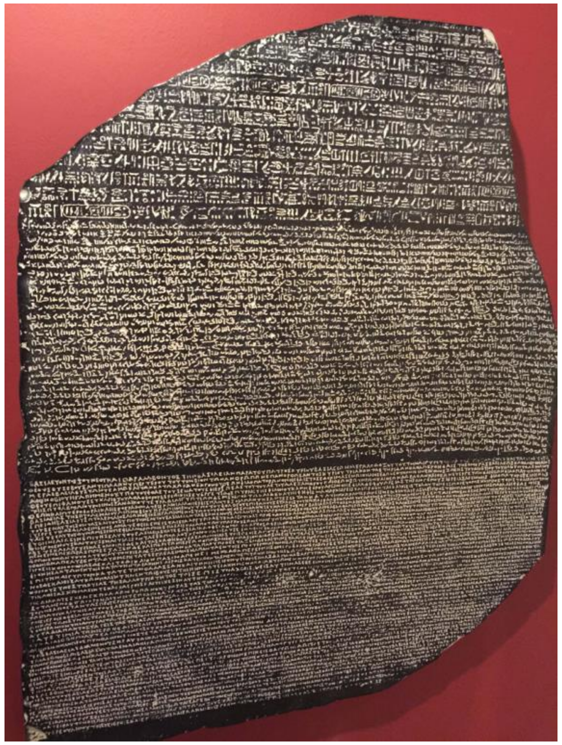
- Convert the decimal number **108** into:
  - Binary number: 1101100
  - Hexadecimal number: 6C

- Convert the binary number **1000011000** into:
  - Hexadecimal number: 218
  - Decimal number: 536

Why Data Representation?

- A microcontroller uses a fixed number of bits to represent a piece of data, which could be a number, a character, or others. Hardware cannot handle infinite long bit sequences.
- A *n*-bit storage location can represent up to 2^n distinct entities.
- E.g. a 3-bit memory location can hold one of these eight binary patterns: 000, 001, 010, 011, 100, 101, 110, or 111. It can represent at most 8 distinct entities:
  - represent numbers 0 to 7,
  - numbers 8881 to 8888,
  - characters 'A' to 'H',
  - or up to 8 kinds of fruits like apple, orange, banana, etc.
- A memory location stores a binary pattern. It is entirely up to the programmer to decide on how these patterns are to be interpreted.
- For example, the 8-bit binary pattern "$(0100\ 0001)_2$" can be interpreted as an unsigned integer 65, or an ASCII character 'A', or any other kind of data known only to the programmer.
- The programmer first decides how to represent a piece of data in a binary pattern before the binary patterns make sense.
- The interpretation of binary pattern is called ***data representation*** or ***encoding***.
- Once the data representation scheme is decided between parties, certain constraints, such as the precision and range will be applied. Hence, it is important to understand data representation to write correct and high-performance programs!

Why Data Representation?



Rosetta Stone 196BC

The moral of the story is unless you know the encoding scheme, there is no way that you can decode the data!

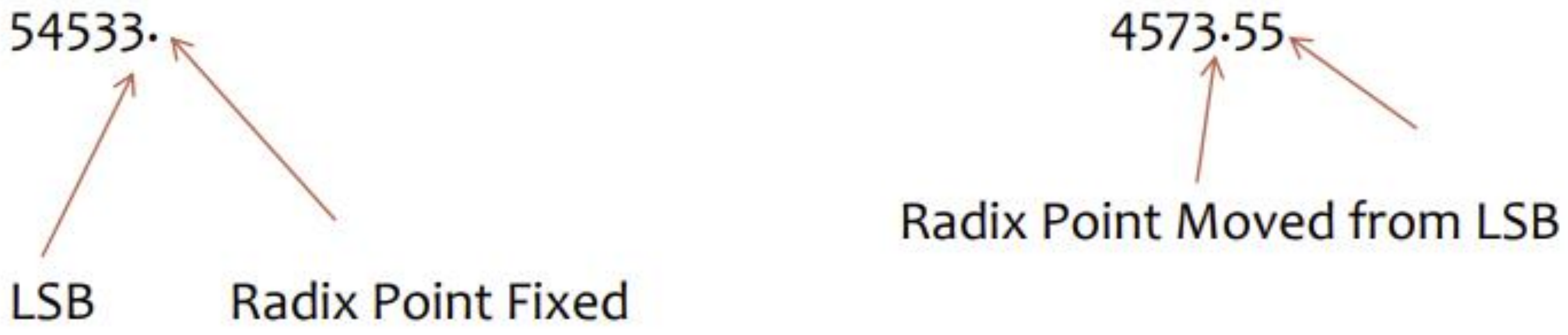
Representing Numbers

Integer Representation

- Numeric data consists of numbers that can be used in arithmetic operations.
- **Integers** are fixed-point numbers with the radix point fixed after the least-significant bit.
- **Real numbers** are floating-point numbers with the position of the radix point varies.

Example:

3774_{10} is a fixed point number (integer)
2949.94_{10} is a floating point number (real number)



Representing Numbers

Integer Representation

- Integers and floating-point numbers have different representations and are processed differently.

- Numbers are represented by a fixed number of bits. These bit sizes are typically **8**-bit, **16**-bit, **32**-bit, **64**-bit. These sizes are usually multiple of 8, because memories are organized on an 8-bit byte basis.

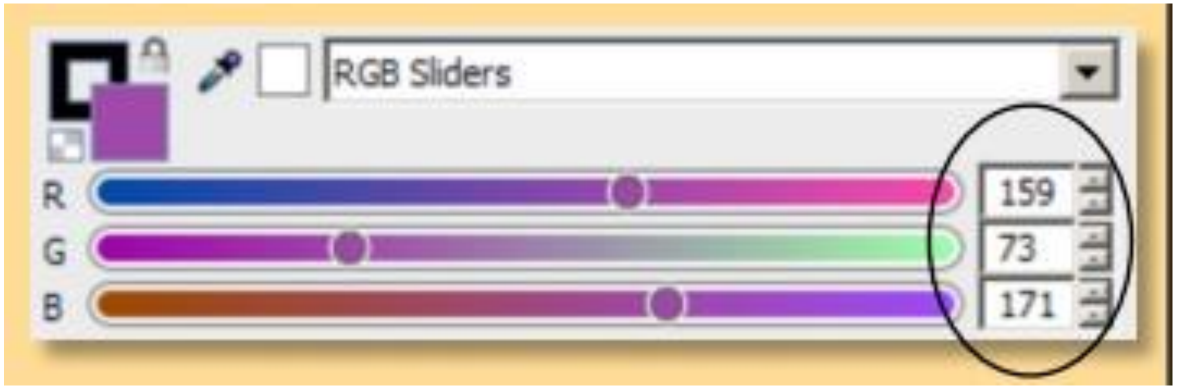
- There are two representation schemes for integers:
  - *Unsigned Integers*: can represent zero and positive integers.
  - *Signed Integers*: can represent zero, positive and negative integers.
    a. Sign-Magnitude representation
    b. 1's Complement representation
    c. 2's Complement representation

Representing Numbers

Integer Representation

A practical example of integer use:

- Colours are often stored using 3 numbers to represent red, green & blue (RGB).
- The screenshot below shows how a colour can be selected in a graphics application by changing the RGB values.
- When the colour is saved, it is stored as three 8-bit binary numbers (each one between 0 and 255).



R	159
G	73
B	171

For example R=159,G=73,B=171 would be stored as: 10011111 01001001 10101011

10011111	01001001	10101011
159	73	171

Integer Storage Sizes

Standard sizes:

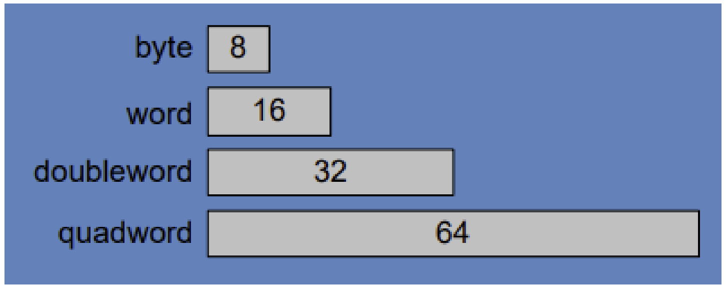


Table 1-4 Ranges of Unsigned Integers.

Storage Type	Range (low–high)	Powers of 2
Unsigned byte	0 to 255	0 to $(2^8 - 1)$
Unsigned word	0 to 65,535	0 to $(2^{16} - 1)$
Unsigned doubleword	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Unsigned quadword	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

What is the largest unsigned integer that may be stored in 20 bits?

Representing Numbers - Integer Representation
Unsigned Integers

- When a fixed binary number is used to hold positive values, it is considered as ***unsigned.***
- Unsigned integers can represent zero and positive integers.
- The range of positive values that can be represented is from **0 to 2^n-1**, where n is the number of bits being used.

Example 1:

If ***n*=8** and the binary pattern is $(01000001)_2$, the value of this unsigned integer is:

$$1*2^0 + 1*2^6 = (65)_{10}$$

Example 2:

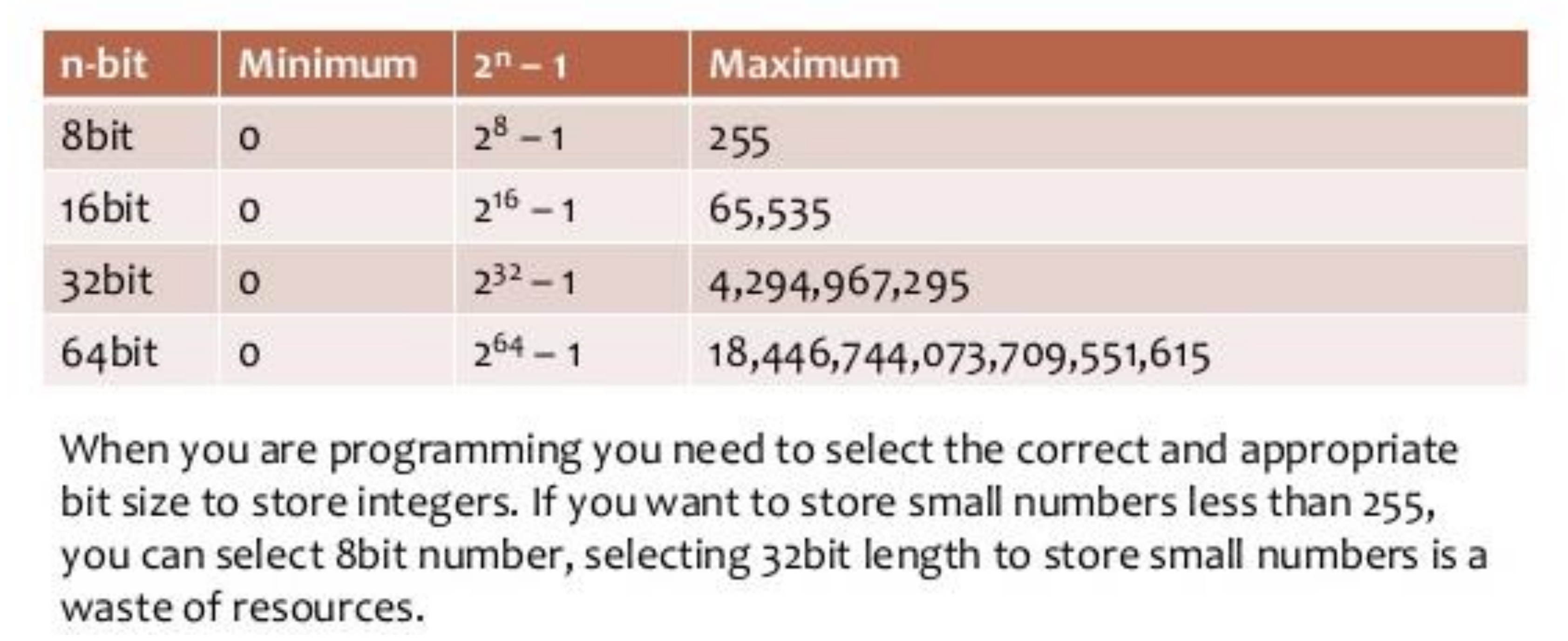
If ***n=16*** and the binary pattern is $(0001000000001000)_2$, the value of this unsigned integer is:

$$1*2^3 + 1*2^{12} = (4104)_{10}$$

Representing Numbers - Integer Representation

Unsigned Integers

An ***n***-bit pattern can represent ***2^n*** distinct integers. An *n*-bit unsigned integer can represent integers from ***0*** to ***(2^n)-1***, as tabulated below

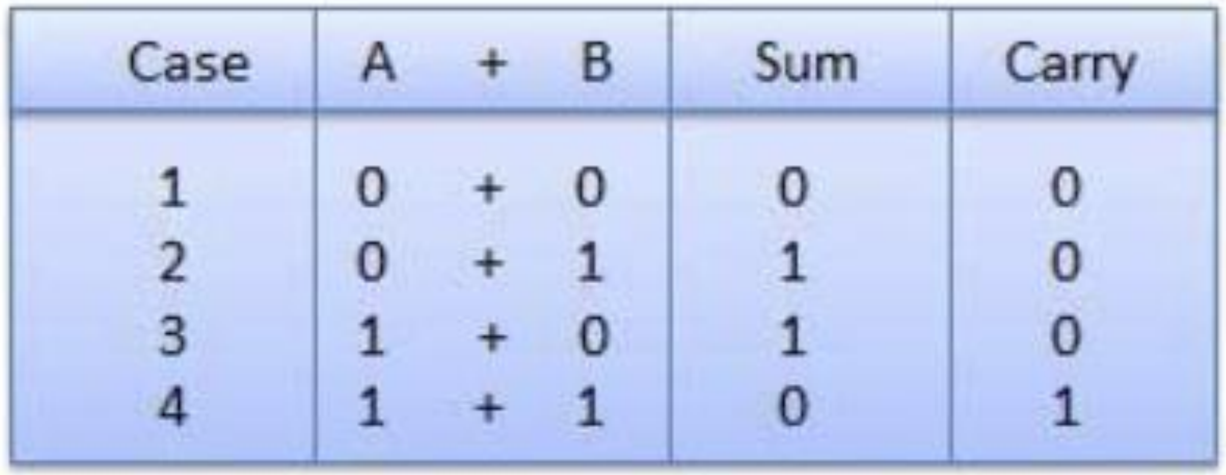


n-bit	Minimum	$2^n - 1$	Maximum
8bit	0	$2^8 - 1$	255
16bit	0	$2^{16} - 1$	65,535
32bit	0	$2^{32} - 1$	4,294,967,295
64bit	0	$2^{64} - 1$	18,446,744,073,709,551,615

When you are programming you need to select the correct and appropriate bit size to store integers. If you want to store small numbers less than 255, you can select 8bit number, selecting 32bit length to store small numbers is a waste of resources.

Representing Numbers - Integer Representation
Unsigned Integers – Binary Addition

Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.

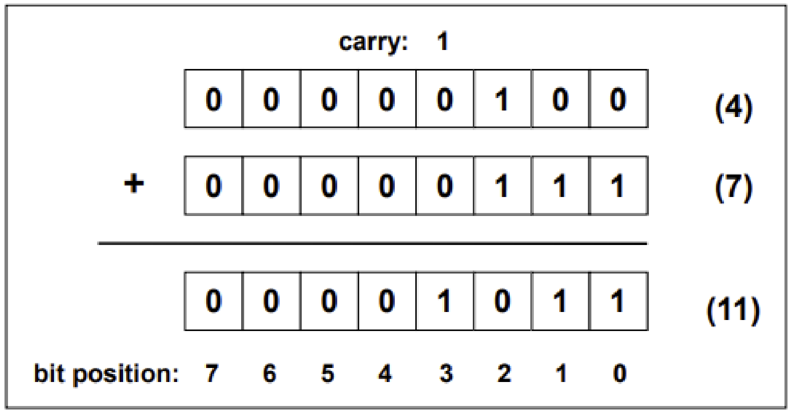


Case	A + B	Sum	Carry
1	0 + 0	0	0
2	0 + 1	1	0
3	1 + 0	1	0
4	1 + 1	0	1

Representing Numbers - Integer Representation

Unsigned Integers – Binary Addition

- Start with the least significant bit (rightmost bit)
- Add each pair of bits
- Include the carry in the addition, if present



Representing Numbers - Integer Representation

Unsigned Integers – Binary Addition

Complete the following binary addition exercises:

$$\begin{array}{r} 1\ 1\ 1\ 1\ 0\ 1_2 \\ +\ 0\ 0\ 1\ 1\ 0\ 1_2 \\ \hline \end{array}$$

$$\begin{array}{r} 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1_2 \\ +\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1_2 \\ \hline \end{array}$$

Representing Numbers - Integer Representation

Unsigned Integers – Binary Addition

$$\begin{array}{ccccccc}
1 & 1 & 1 & 1 & & 1 & \\
 & 1 & 1 & 1 & 1 & 0 & 1_2 \\
+ & 0 & 0 & 1 & 1 & 0 & 1_2 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0_2
\end{array}
\rightarrow
\begin{array}{r}
61_{10} \\
+13_{10} \\
\hline
74_{10}
\end{array}$$

$$\begin{array}{ccccccccc}
1 & & & 1 & & 1 & 1 & 1 & \\
 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1_2 \\
+ & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1_2 \\
\hline
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\rightarrow
\begin{array}{r}
151_{10} \\
+213_{10} \\
\hline
364_{10}
\end{array}$$

Representing Numbers - Integer Representation

Unsigned Integers – Binary Subtraction

Case	A - B	Subtract	Borrow
1	0 - 0	0	0
2	1 - 0	1	0
3	1 - 1	0	0
4	0 - 1	0	1

Example 2: Subtract binary **10100** from **101001**

$$\begin{array}{rcccccc}
 & 0 & 2 & 0 & 2 & & \\
 & \cancel{1} & \cancel{0} & \cancel{1} & \cancel{0} & 0 & 1 \\
- & & 1 & 0 & 1 & 0 & 0 \\
\hline
 & & 1 & 0 & 1 & 0 & 1
\end{array}$$

Verification

$$\begin{array}{rlrl}
 & 101001_2 \rightarrow & & 41_{10} \\
- & \underline{10100_2} & - & \underline{20_{10}} \\
 & & & 21_{10}
\end{array}$$

64	32	16	8	4	2	1
		1	0	1	0	1

$= 16 + 4 + 1$

$= 21_{10}$

Representing Numbers - Integer Representation

Unsigned Integers – Binary Subtraction

Complete the following binary subtraction exercises:

$$\begin{array}{r} 110101_2 \\ -\ 101011_2 \\ \hline \end{array}$$

$$\begin{array}{r} 1001101_2 \\ -\ \ 110111_2 \\ \hline \end{array}$$

Representing Numbers - Integer Representation

Unsigned Integers – Binary Subtraction

$$
\begin{array}{rcccccccl}
 & & 0 & 2 & 0 & 2 & & & \\
 & 1 & \cancel{1} & 0 & \cancel{1} & 0 & 1_2 & \rightarrow & 53_{10} \\
- & 1 & 0 & 1 & 0 & 1 & 1_2 & & -\ 43_{10} \\
\hline
 & 0 & 0 & 1 & 0 & 1 & 0_2 & & 10_{10}
\end{array}
$$

$$
\begin{array}{rccccccccl}
 & & 1 & & & 2 & & & & \\
 & 0 & \cancel{2} & 2 & 0 & \cancel{0} & 2 & & & \\
 & \cancel{1} & 0 & 0 & \cancel{1} & \cancel{1} & 0 & 1_2 & \rightarrow & 77_{10} \\
- & & 1 & 1 & 0 & 1 & 1 & 1_2 & & -\ 55_{10} \\
\hline
 & & 0 & 1 & 0 & 1 & 1 & 0_2 & & 22_{10}
\end{array}
$$

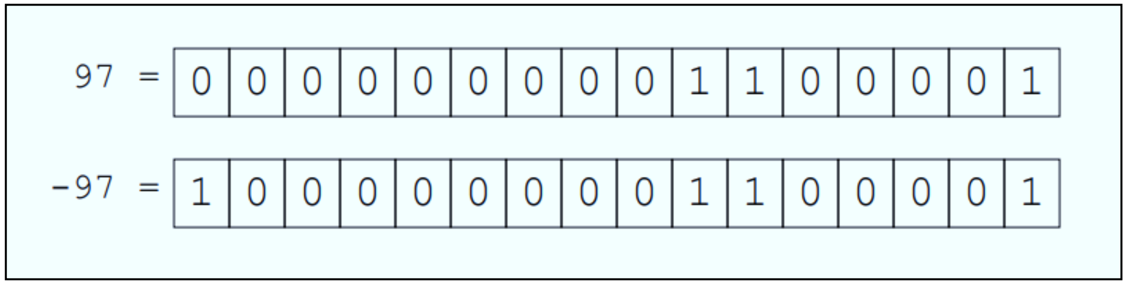
Representing Numbers - Integer Representation

Signed Integers

Sign-Magnitude Representation

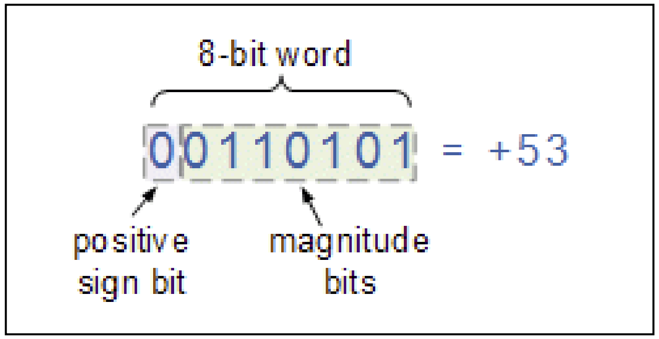
- The most-significant bit (MSB) is the sign bit: **0** represents a **positive** integer and **1** represents a **negative** integer.
- The magnitude of the integer, however, is interpreted differently in different schemes.
- In the Sign-Magnitude scheme, the remaining **n-1** bits represents the magnitude (absolute value) of the integer.

If ***n*=16**



97 =	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
−97 =	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1

If ***n*=8**

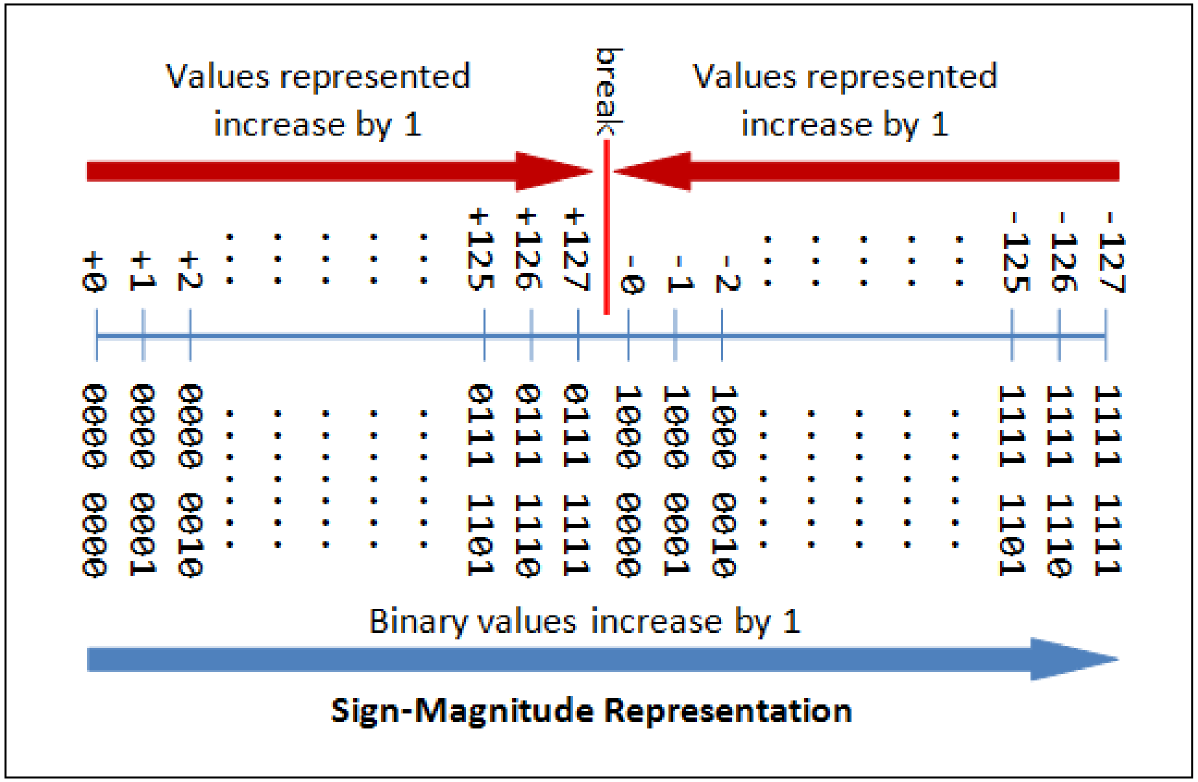


Representing Numbers - Integer Representation

Signed Integers

Sign-Magnitude Representation

- Total possible numbers representable by an n-bit register using signed magnitude is equal to: 2^{n-1}, as the MSB is always reserved for the sign
- Largest representable magnitude, in this method, is $(2^{n-1}-1)$
- Range of numbers: $-(2^{n-1}-1)$, … , -1, -0, +0, +1, …, $+(2^{n-1}-1)$



If ***n*=8**

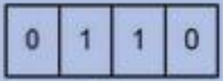
Representing Numbers - Integer Representation

Signed Integers

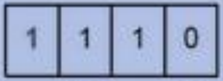
Sign-Magnitude Representation

Examples

Q: Show the signed magnitude representations of +6, -6, +13 and -13 using a 4-bit register and an 8-bit register



Signed-Magnitude Representation of +6



Signed-Magnitude Representation of -6

- Note that: $(6)_{10} = (110)_2$, $(13)_{10} = (1101)_2$
- For a 4-bit register, the leftmost bit is reserved for the sign, which leaves 3 bits only to represent the magnitude
- The largest magnitude that can be represented = $2^{(4-1)} - 1 = 7$
- Because 13 > 7, the numbers +13 and –13 can NOT be represented using the 4-bit register

University of Glasgow

Representing Numbers - Integer Representation

Signed Integers

Sign-Magnitude Representation - Addition

Let's see how the addition rules work with signed magnitude numbers . . .

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Once we have worked our way through all eight bits, we are done.

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

In this example, we were $\longrightarrow$ careful to pick two values whose sum would fit into seven bits. If that is not the case, we have a problem.

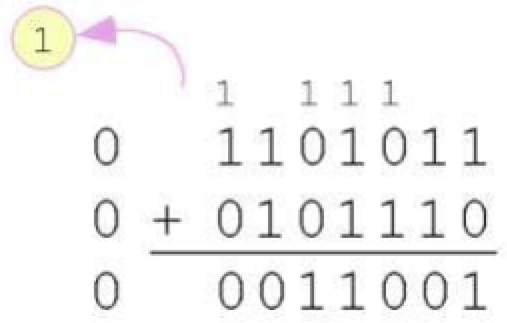
Representing Numbers - Integer Representation

Signed Integers

Sign-Magnitude Representation - Addition

- Example:
  - Using ***signed magnitude*** binary arithmetic, find the sum of 107 and 46.
- We see that the carry from the seventh bit ***overflows*** and is discarded, giving us the erroneous result:

  107 + 46 = 25.



Representing Numbers - Integer Representation

Signed Integers

Sign-Magnitude Representation - Addition

$$\begin{array}{rrrl} & & {\scriptstyle 0\,0\,0} & \\ & 0 & 0011 & (+3_{10}) \\ + & 0 & 0100 & (+4_{10}) \\ \hline & 0 & 0111 & (+7_{10}) \end{array}$$

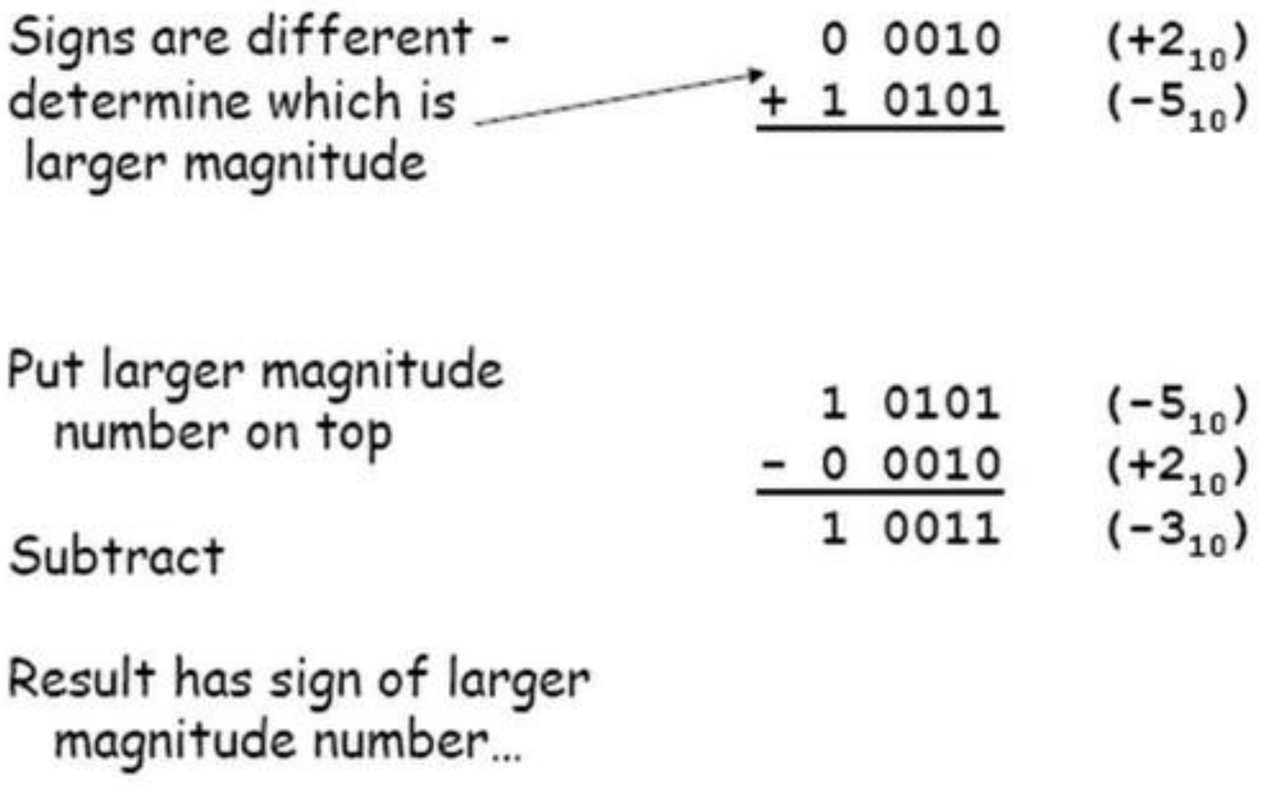
$$\begin{array}{rrrl} & & {\scriptstyle 0\,0\,0} & \\ & 1 & 0011 & (-3_{10}) \\ + & 1 & 0100 & (-4_{10}) \\ \hline & 1 & 0111 & (-7_{10}) \end{array}$$

If sign is the same, just add the magnitude

Representing Numbers - Integer Representation

Signed Integers

Sign-Magnitude Representation – Addition



Signs are different - determine which is larger magnitude

$$\begin{array}{rrr} & 0\ 0010 & (+2_{10}) \\ + & 1\ 0101 & (-5_{10}) \\ \hline \end{array}$$

Put larger magnitude number on top

Subtract

$$\begin{array}{rrr} & 1\ 0101 & (-5_{10}) \\ - & 0\ 0010 & (+2_{10}) \\ \hline & 1\ 0011 & (-3_{10}) \end{array}$$

Result has sign of larger magnitude number...

Representing Numbers - Integer Representation

Signed Integers

Sign-Magnitude Representation - Addition

- Need two separate operations
  - Addition
  - Subtraction
- Several decisions:
  - Signs same or different?
  - Which operand is larger?
  - What is the sign of final result?
  - Two zeroes (+0, -0)

Advantages
-Easy to understand

Disadvantages
–Two different 0s
– Hard to implement in logic

Representing Numbers - Integer Representation

Signed Integers

Advantages & Disadvantages of Sign-Magnitude Representation

Advantages:

1. The easiest for humans to understand.

Disadvantages:

1. Positive and negative integers need to be processed separately.
2. This adds a fair amount of complexity when performing arithmetic
3. For performing addition or subtraction we need to first check the sign and magnitude of the operands. The result of these checks determines which operation we need to perform, the order of the operands and the sign of the result.
4. More circuitry is needed to do arithmetic, therefore, increasing the hardware cost.
5. There are two ways of representing the value zero:

0	=	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	=	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

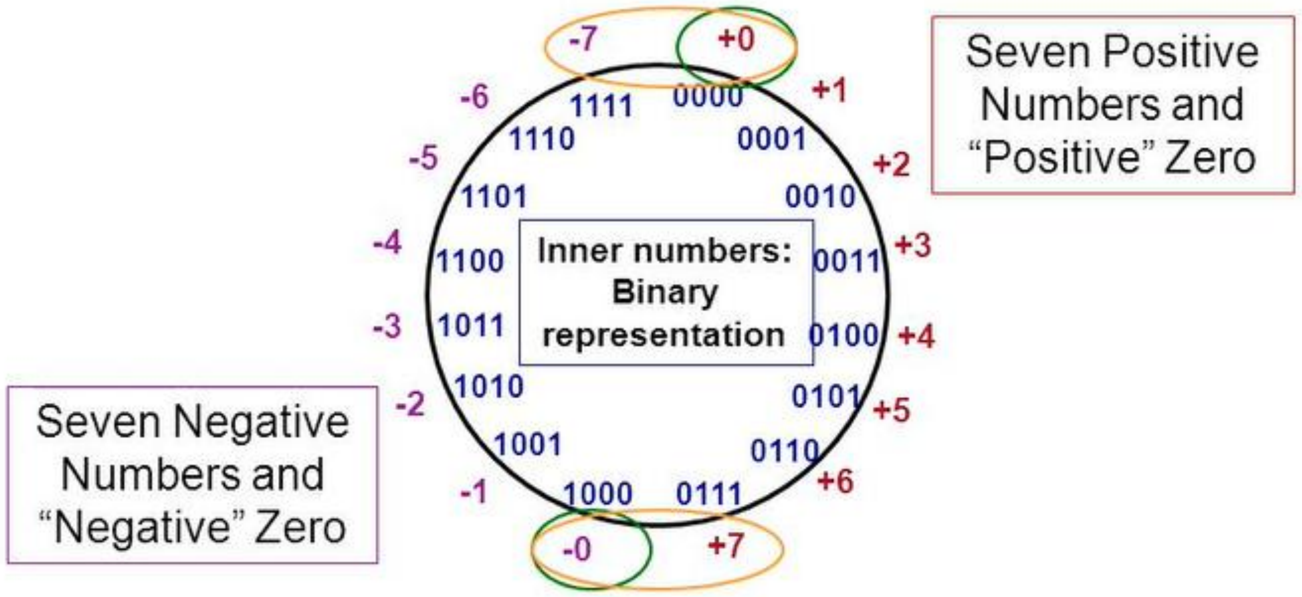
This could lead to inefficiency and confusion!

Representing Numbers - Integer Representation

Signed Integers

Advantages & Disadvantages of Sign-Magnitude Representation

5. There are two ways of representing the value zero:


-7
+0
0000
+1
0001
+2
0010
+3
0011
+4
0100
+5
0101
+6
0110
0111
+7
-0
1000
-1
1001
-2
1010
-3
1011
-4
1100
-5
1101
-6
1110
1111
Inner numbers: Binary representation
Seven Positive Numbers and "Positive" Zero
Seven Negative Numbers and "Negative" Zero

- **Two different representations for 0!**

This could lead to inefficiency and confusion!

Representing Numbers - Integer Representation

Signed Integers

1's Complement Representation

- Again, MSB is the sign bit: **0** represents a positive integer and **1** represents a negative integer.

- The remaining bits represents the magnitude of the integer, as follows:

  - For positive integers, the absolute value of the integer is the magnitude of the remaining bits.
    Similar to the Sign-Magnitude representation!

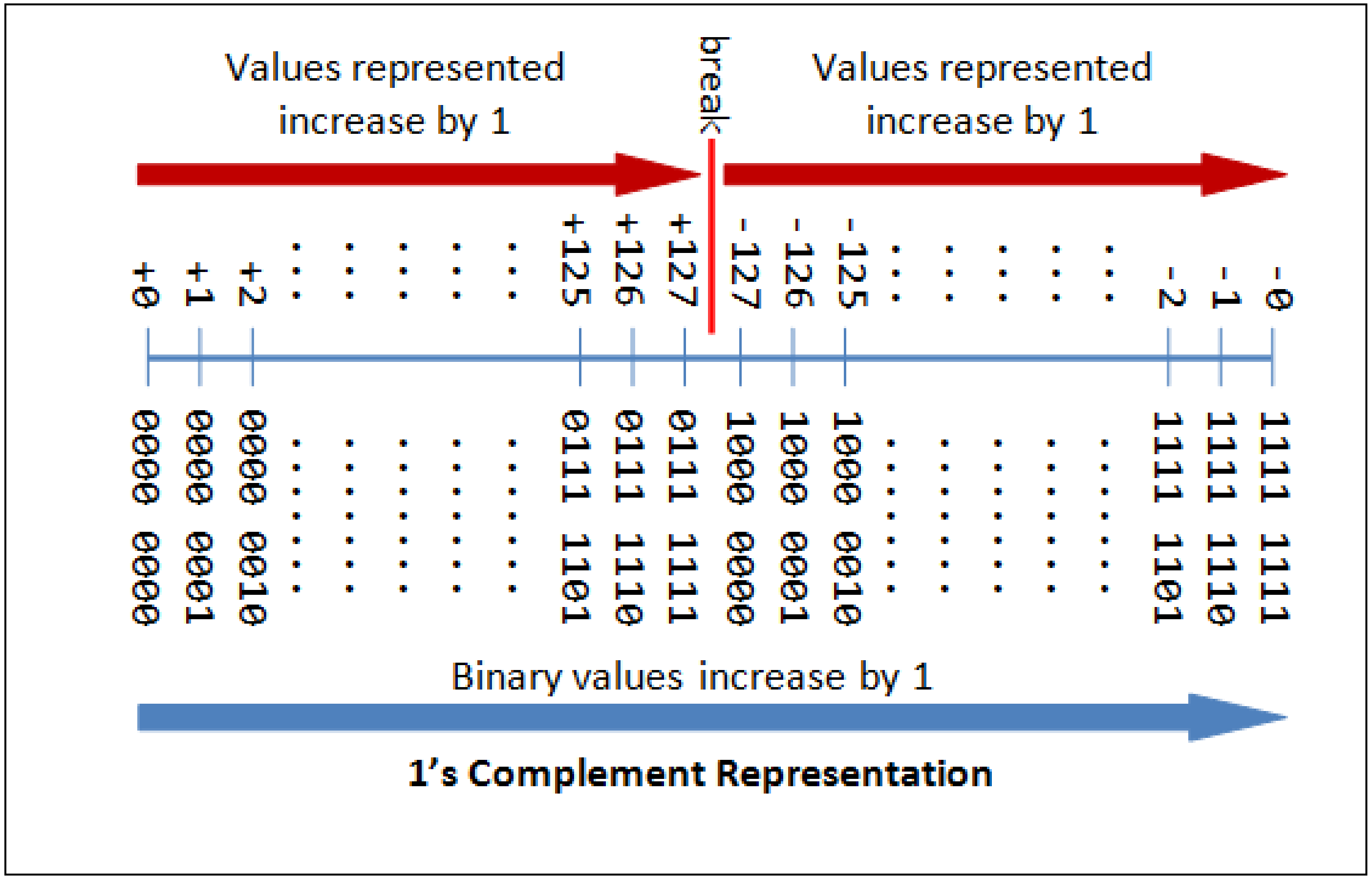
  - For negative integers, the absolute value of the integer is the magnitude of the ***complement*** (inverse) of the remaining bits (**hence called 1's complement**).

97	=	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
-97	=	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0

Representing Numbers - Integer Representation

Signed Integers

1's Complement Representation



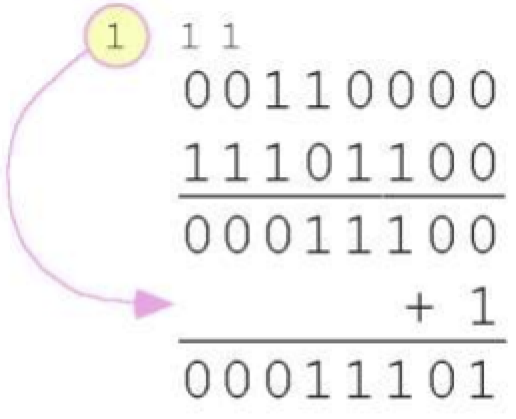
If *n*=8

Representing Numbers - Integer Representation

Signed Integers

1's Complement Representation - Addition

- With one's complement addition, the carry bit is "carried around" and added to the sum.
  - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19



```
 1   1 1
     00110000
     11101100
     --------
     00011100
          + 1
     --------
     00011101
```

We note that 19 in one's complement is 00010011,
so -19 in one's complement is: 11101100.

Representing Numbers - Integer Representation

Signed Integers

Advantages & Disadvantages of 1’s Complement Representation

Advantages:

1. Inversion of the bits in the number is done using simple **NOT** gates.

2. Additions and subtractions are done on signed numbers without checking the sign or magnitude of the numbers.

3. Only adder units are needed to perform both addition and subtraction

Disadvantage:

➢ There are two ways of representing the value zero:

0	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	=	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Despite the advantages provided by 1’s complement, it is not used in any of the modern computers!

Representing Numbers - Integer Representation

Signed Integers

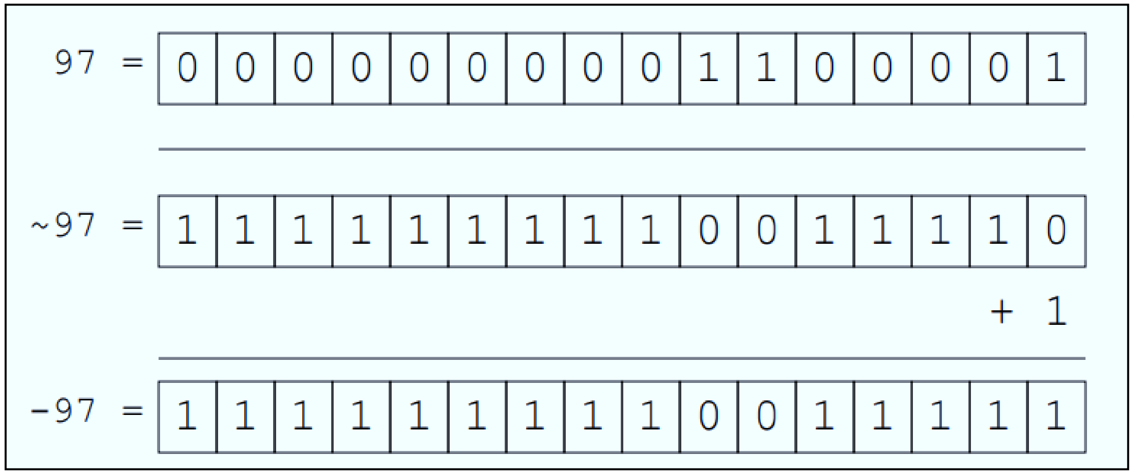
2's Complement Representation

- Again, MSB is the sign bit: **0** represents a positive integer and **1** represents a negative integer.

- The remaining bits represents the magnitude of the integer, as follows:

  - For positive integers, the absolute value of the integer is the magnitude of the remaining bits.
    Similar to the Sign-Magnitude representation!

  - For negative integers, the absolute value of the integer is the magnitude of the ***complement*** of the remaining bits plus one (**hence called 2's complement**).



```
 97  = 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1
       -------------------------------
~97  = 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0
                                   + 1
       -------------------------------
-97  = 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1
```

Invert

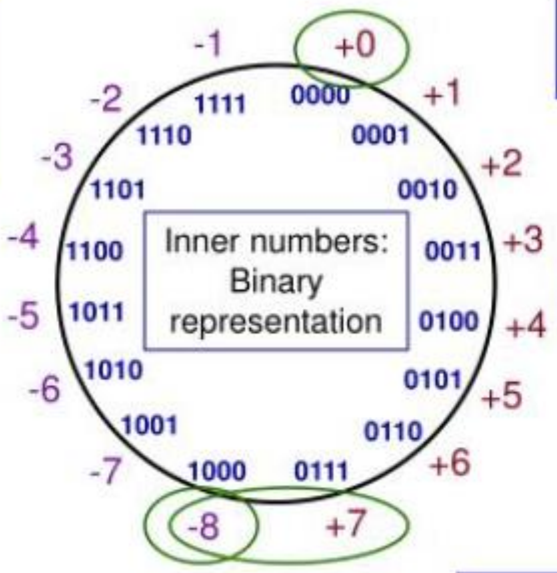
Add 1; if the addition causes a carry bit past the most significant bit, discard the high carry

Representing Numbers - Integer Representation

Signed Integers

2's Complement Representation

Re-order Negative Numbers to Eliminate Discontinuities



Note: Example is shown for 4-bit numbers

Eight Positive Numbers

If *n*=4

- Only one discontinuity now
- Only one zero
- One extra negative number

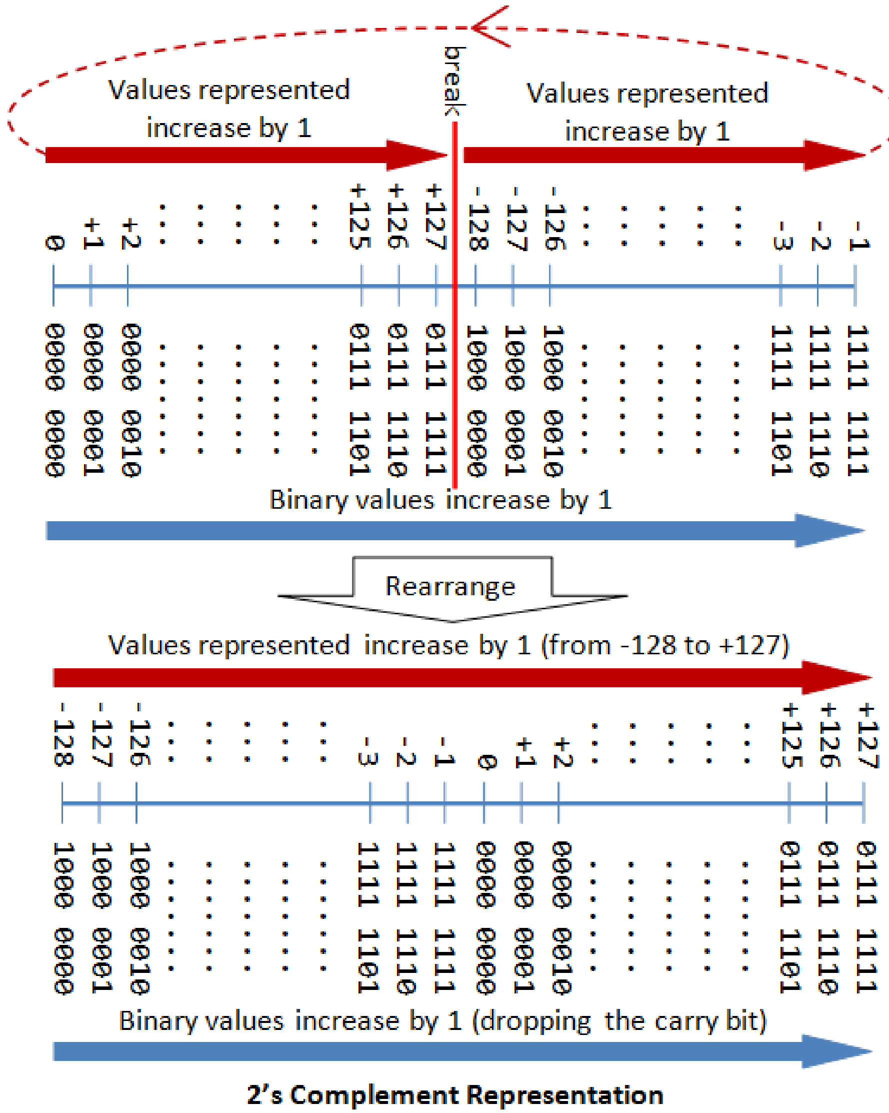
Note: Negative numbers still have 1 for MSB

Representing Numbers - Integer Representation

Signed Integers

2's Complement Representation

If *n*=8



Values represented increase by 1
break
Values represented increase by 1
0
+1
+2
+125
+126
+127
-128
-127
-126
-3
-2
-1
0000 0000
0000 0001
0000 0010
0111 1101
0111 1110
0111 1111
1000 0000
1000 0001
1000 0010
1111 1101
1111 1110
1111 1111
Binary values increase by 1
Rearrange
Values represented increase by 1 (from -128 to +127)
Binary values increase by 1 (dropping the carry bit)
2's Complement Representation