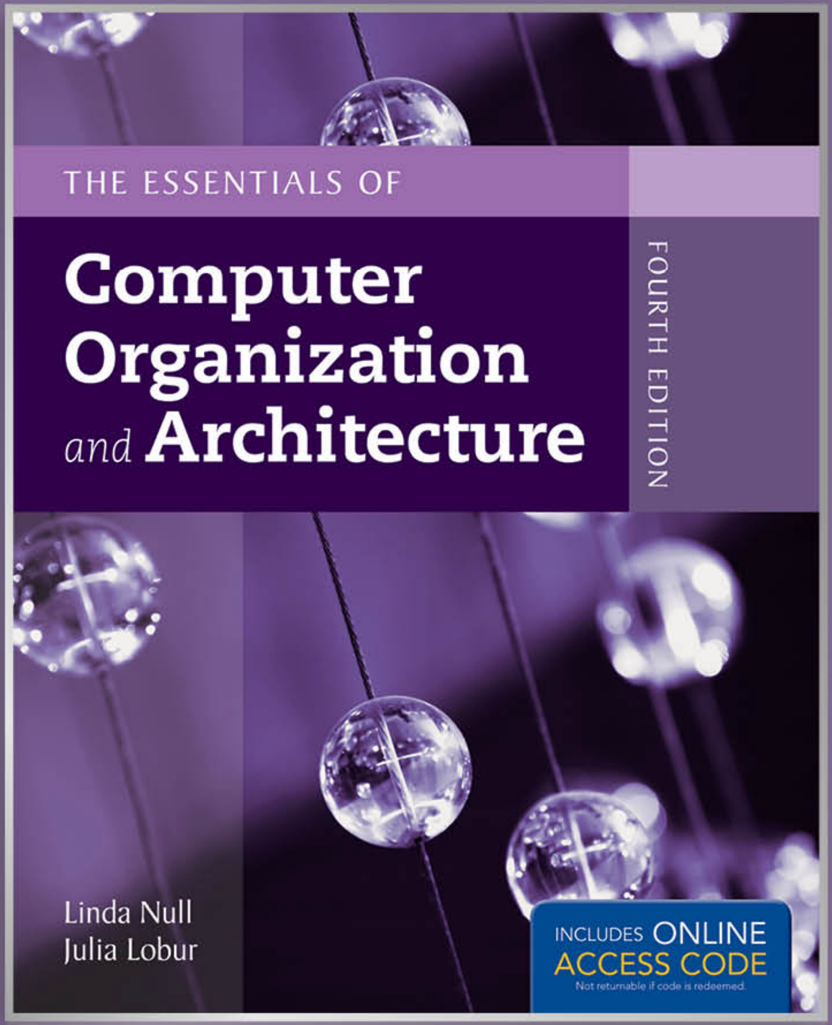
Chapter 2

Data Representation in Computer Systems



Chapter 2 Objectives

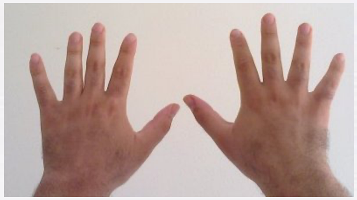
- Understand the fundamentals of **numerical data representation** and manipulation in digital computers.
- Master the skill of **converting between various radix (base) systems**.

Chapter 2 Objectives

- Understand the fundamental concepts of **floating-point representation**.
- Gain familiarity with the most popular **character codes**.
- Understand how **errors** can occur in computations because of **overflow and truncation**.
- Understand the concepts of **error detecting** and **correcting codes**.

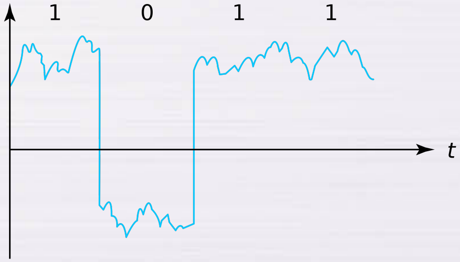
2.1 Introduction

- Decimal system (ten digits)
  - Humans naturally started to use the decimal system because we have ten fingers



- Octal system (eight digits)
  - The Simpsons would most likely use the octal system

- Binary system (two digits)
  - Computers are based on digital circuits, where signals correspond to low (0) or high (1) voltage
  - Thus, binary is common way of representing data in a computer



- Integers can be converted to any base system

2.1 Data Units in Computers

- A ***bit*** is the **most basic unit** of information in a computer.
  - It is a **state** of "on" or "off" in a **digital circuit**.
  - Sometimes these states are "high" or "low" voltage instead of "on" or "off"
- A ***byte*** is a group of **eight bits**.
  - In early computers a byte was sufficient to encode all possible characters
  - Therefore, a byte is the smallest possible *addressable* unit of computer storage.
  - "addressable," means that a particular byte can be retrieved according to its location in memory.

2.1 Data Units in Computers

- A ***word*** is a **consecutive group of bytes**.
  - Word sizes of **16, 32, or 64 bits** are most common.
  - In a word-addressable system, a word is the smallest addressable unit of storage.
- A group of four bits is called a ***nibble***.
  - Bytes, therefore, consist of two nibbles: a "high-order nibble," and a "low-order" nibble.

2.2 Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2.
- The byte 00011001 in powers of 2 is:

Bit positions: 7 6 5 4 3 2 1 0

$$0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 0 + 0 + 0 + 16 + 8 + 0 + 0 + 1$$

$$= 25 \text{ in decimal}$$

- Shortened representation:

$$1 \times 2^4 + 1 \times 2^3 + 1 \times 2^0$$

$$= 16 + 8 + 1 = 25$$

- 2 is what is called the radix (or base) of the binary system

2.2 Positional Numbering Systems

- The decimal number 947 in powers of 10 is:

  $9 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$

- The decimal number 5836.47 in powers of 10 is:

  $5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0$

  $+ 4 \times 10^{-1} + 7 \times 10^{-2}$

- When the radix of a number is something other than 10, the base is denoted by a subscript.
  - Sometimes, the subscript 10 is added for emphasis:

    $11001_2 = 25_{10}$

2.3 Converting Between Bases

- Because **binary numbers are the basis for all data representation in digital computer systems**, it is important that you become proficient with the binary system.
- Your knowledge of the binary numbering system will enable you to **understand the operation of all computer components** as well as the **design of instruction set architectures**.

2.3 Converting Between Bases

- In an earlier slide, we said that **every integer value can be represented exactly using any base system**.
- There are two methods for base conversion: the **subtraction method** and the **division remainder method**.
- We first discuss the **subtraction method** which is intuitive, but cumbersome for large numbers.

2.3 Converting Between Bases

- **Suppose we want to convert the decimal number 190 to base 3.**
  - Make a list of powers of 3
  - Detect the largest power that has a multiple smaller or equal 190
  - Subtract this multiple from 190
  - Repeat this for the obtained difference and the next smaller power of 3

$$\begin{array}{r} 190 \\ -\ 162 \\ \hline 28 \end{array} \quad 162 = 3^4 \times 2$$

2.3 Converting Between Bases

- **Suppose we want to convert the decimal number 190 to base 3.**
  - We know that $3^5 = 243$ so our result will be less than six digits wide. The largest power of 3 that we need is therefore $3^4 = 81$, and
  - $81 \times 2 = 162$.
  - Write down the 2 and subtract 162 from 190, giving 28.

$$\begin{array}{r} 190 \\ -\ 162 \\ \hline 28 \end{array} \quad = 3^4 \times 2$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**
  - The next power of 3 is $3^3 = 27$. We'll need one of these, so we subtract 27 and write down the numeral 1 in our result.
  - The next power of 3, $3^2 = 9$, is too large, but we have to assign a placeholder of zero and carry down the 1.

$$
\begin{array}{rl}
190 & \\
-\ 162 & = 3^4 \times 2 \\
\hline
28 & \\
-\ 27 & = 3^3 \times 1 \\
\hline
1 & \\
-\ 0 & = 3^2 \times 0 \\
\hline
1 &
\end{array}
$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**
  - $3^1 = 3$ is again too large, so we assign a zero placeholder.
  - The last power of 3, $3^0 = 1$, is our last choice, and it gives us a difference of zero.
  - **Our result, reading from top to bottom is:**

  $190_{10} = 21001_3$

```
  190
- 162  = 3^4 × 2
  ---
   28
-  27  = 3^3 × 1
  ---
    1
-   0  = 3^2 × 0
  ---
    1
-   0  = 3^1 × 0
  ---
    1
-   1  = 3^0 × 1
  ---
    0
```

2.3 Converting Between Bases

- Another method of converting integers from decimal to some other radix uses **division**.
- This method is **mechanical and easy**.
- It employs the idea that **successive division by a base is equivalent to successive subtraction by powers of the base**.
- Let's use the **division remainder method** to again convert 190 in decimal to base 3.

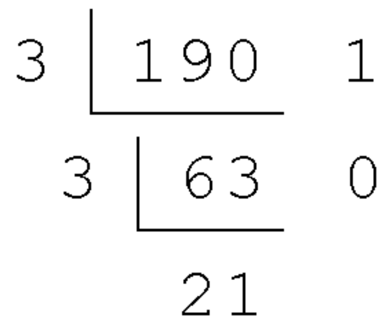
2.3 Converting Between Bases

- **Converting 190 to base 3...**
  - First we **take the number** that we wish to convert and **divide it by the new base** in which we want to express our result.
  - **Record** the **quotient and** the **remainder**.
  - In this case, 3 divides 190 63 times, with a remainder of 1.

$$3 \,\underline{\big|\; 190}\quad 1$$
$$63$$

2.3 Converting Between Bases

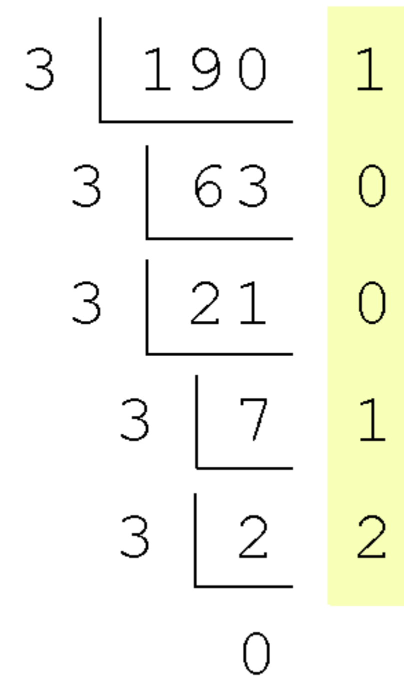
- **Converting 190 to base 3...**
  - 63 is evenly divisible by 3.
  - Our remainder is zero, and the quotient is 21.



2.3 Converting Between Bases

- **Converting 190 to base 3...**
  - Continue in this way until the quotient is zero.
  - In the final calculation, we note that 3 divides 2 zero times with a remainder of 2.
  - **Our result, reading from bottom to top is:**

$$190_{10} = 21001_3$$



2.3 Converting Between Bases

- Fractional values like ⅓ **can be approximated in all base systems**.
- Unlike integer values, fractions **do not necessarily have exact representations** under all radices.
- Example:

$$\tfrac{1}{3} = 0.33333\ldots_{10} \approx 0.33_{10} = 3 \times 10^{-1} + 3 \times 10^{-2}$$

$$\tfrac{1}{3} = 1 \times 3^{-1} = 0.1_3$$

2.3 Converting Between Bases

- Fractional values have **nonzero digits to the right of the *radix point***.
- Digits to the right of a radix point **represent negative powers of the radix**:

$$0.11_2 = 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

Positions: 0 -1 -2

2.3 Converting Between Bases

- **The subtraction method for fractions is basically identical to the subtraction method for whole numbers.**
- Instead of subtracting positive powers of the target base, we **subtract negative powers of the base**.
- Start with power -1, then -2, and so on

2.3 Converting Between Bases

- **Subtraction method to convert the decimal 0.8125 to binary.**
  - Our result, reading from <u>top to bottom</u> is:
    $0.8125_{10} = 0.1101_2$
  - Of course, this method works with any base, not just binary.

$$
\begin{array}{rl}
 & 0.8125 \\
- & 0.5000 \quad = 2^{-1} \times 1 \\
\hline
 & 0.3125 \\
- & 0.2500 \quad = 2^{-2} \times 1 \\
\hline
 & 0.0625 \\
- & 0 \quad = 2^{-3} \times 0 \\
\hline
 & 0.0625 \\
- & 0.0625 \quad = 2^{-4} \times 1 \\
\hline
 & 0
\end{array}
$$

2.3 Converting Between Bases

- **Multiplication method to convert the decimal 0.8125 to base 2.**
  - Multiply by base
  - The first product carries into the integer part.

$$\begin{array}{r} .8125 \\ \times \quad 2 \\ \hline 1.6250 \end{array}$$

2.3 Converting Between Bases

- **Converting 0.8125 to binary . . .**
  - Ignoring the value in the integer part at each step,
  - continue multiplying each fractional part by the base.

$$\begin{array}{r} .8125 \\ \times\quad 2 \\ \hline 1.6250 \end{array}$$

$$\begin{array}{r} .6250 \\ \times\quad 2 \\ \hline 1.2500 \end{array}$$

$$\begin{array}{r} .2500 \\ \times\quad 2 \\ \hline 0.5000 \end{array}$$

2.3 Converting Between Bases

- **Converting 0.8125 to binary . . .**
  - Finished when the product is zero, or until you have reached the desired number of binary places (rounding).
  - **Result, reading integer part from top to bottom is:** $0.8125_{10} = 0.1101_2$
  - This method also works with any base. Just use the target base as the multiplier.

$$
\begin{array}{r}
.8125 \\
\times \quad 2 \\
\hline
1.6250
\end{array}
\qquad
\begin{array}{r}
.6250 \\
\times \quad 2 \\
\hline
1.2500
\end{array}
\qquad
\begin{array}{r}
.2500 \\
\times \quad 2 \\
\hline
0.5000
\end{array}
\qquad
\begin{array}{r}
.5000 \\
\times \quad 2 \\
\hline
1.0000
\end{array}
$$

2.3 Converting Between Bases

- The discussed methods work for any **base x** to **base y** conversion where x and y are integers greater or equal 2
- In most cases it is easier to do a **2-step conversion** from **base x to base 10** and then from **base 10 to base y**

2.3 Converting Between Bases

- Binary system is the most important base system for computers.
- However, it is difficult to read longer binary strings
  - For example: $000100011100_2 = 284_{10}$
- For compactness and ease of reading, binary values are usually expressed using the **hexadecimal** (base 16) system.

2.3 Converting Between Bases

- The hexadecimal numbering system uses the digits 0 to 9 and the letters A to F.
  – The decimal number 12 is C_{16}.
  – The decimal number 26 is $1A_{16}$.
- It is easy to convert between base 16 and base 2, because $16 = 2^4$.
- To **convert from binary to hexadecimal**, all we need to do is **group the binary digits into groups of four** (hextets).

2.3 Converting Between Bases

- Using groups of hextets, the binary number 11010100011011_2 (= 13595_{10}) in hexadecimal is:

0011	0101	0001	1011
3	5	1	B

If the number of bits is not a multiple of 4, pad on the left with zeros.

- Octal (base 8) values are derived from binary by using groups of three bits ($8 = 2^3$):

011	010	100	011	011
3	2	4	3	3

Octal was very useful when computers used six-bit words.

2.4 Signed Integer Representation

- The conversions we have so far presented have involved only unsigned numbers.
- To represent **signed integers**, computer systems allocate the **leftmost bit** to indicate the **sign** of a number.
  - The high-order bit is the leftmost bit. It is also called the most significant bit.
  - **0** is used to indicate a **positive number**; **1** indicates a **negative number**.
- The **remaining bits** contain the **value** of the number (but this can be interpreted different ways)

2.4 Signed Integer Representation

- There are three common ways in which **signed binary integers** may be expressed:
  - **Signed magnitude**
  - **One's complement**
  - **Two's complement**

- In an 8-bit word, ***signed magnitude*** representation places the **absolute value (magnitude)** of the number in the **7 bits to the right of the sign bit**.

2.4 Signed Integer Representation

- For example, in 8-bit **signed magnitude** representation:

+3 is: `00000011`

- 3 is: `10000011`

- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
  - **ignore the signs of the operands during the calculation, apply the appropriate sign afterwards**.

2.4 Signed Integer Representation

- **Binary addition rules for bits:**

```
0 + 0 =  0     0 + 1 =  1
1 + 0 =  1     1 + 1 = 10
```

- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.

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

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Convert 75 and 46 **to binary**, and **arrange as a sum**, **separate** the **sign** bits **from magnitude** bits.

```
0   1001011
0 + 0101110
  ---------
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Find the sum **starting with the rightmost bits** and work left.

$$
\begin{array}{l@{\quad}r}
0 & 1001011 \\
0 & +\ 0101110 \\
\hline
 & 1
\end{array}
$$

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- In the second bit, we have a carry, so we note it above the third bit.

```
             1
0     1001011
0 +   0101110
  -----------
           01
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- The third and fourth bits also give us carries.

```
       1 1 1
0   1001011
0 + 0101110
   --------
       1001
```

2.4 Signed Integer Representation

- 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 careful to pick two values whose sum would fit into seven bits.

If that is not the case, we have a problem: *overflow*

2.4 Signed Integer Representation

- 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 wrong result: 107 + 46 = 25.

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

2.4 Signed Integer Representation

- Signs in signed magnitude representation

  Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.

$$
\begin{array}{cccccccccc}
 & & & & 1 & 1 & & & & \\
1 & & & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & & + & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\hline
1 & & & 1 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}
$$

- The **signs** of the numbers to be added **are both negative**,
- We add the magnitudes and **use the negative sign for the sum**

2.4 Signed Integer Representation

- Mixed sign addition

  Using signed magnitude binary arithmetic, find the sum of 46 and - 25.

$$
\begin{array}{cccccccccc}
 & & & 0 & 2 & & & 0 & 2 \\
0 & & 0 & \cancel{1} & 0 & 1 & 1 & \cancel{1} & 0 \\
1 & + & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\hline
0 & & 0 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}
$$

- **Determine** number with the **larger magnitude**
- **Subtract smaller** magnitude **from larger** magnitude
- The **sign of the number with the larger magnitude** becomes the sign of the sum

  – Note the "**borrows**" from the second and sixth bits.

2.4 Signed Integer Representation

- **Signed magnitude** representation is easy for humans, but it **requires complicated computer hardware**.
- Another disadvantage of signed magnitude is that it allows **two different representations for zero**: positive zero and negative zero.

  00000000 10000000

- For these reasons computers systems employ ***complement systems*** for number representation.

2.4 Signed Integer Representation

- In **one's complement representation**, positive numbers are the same as in sign-magnitude, and **negative numbers** are the bit **complement** of the corresponding **positive number**.

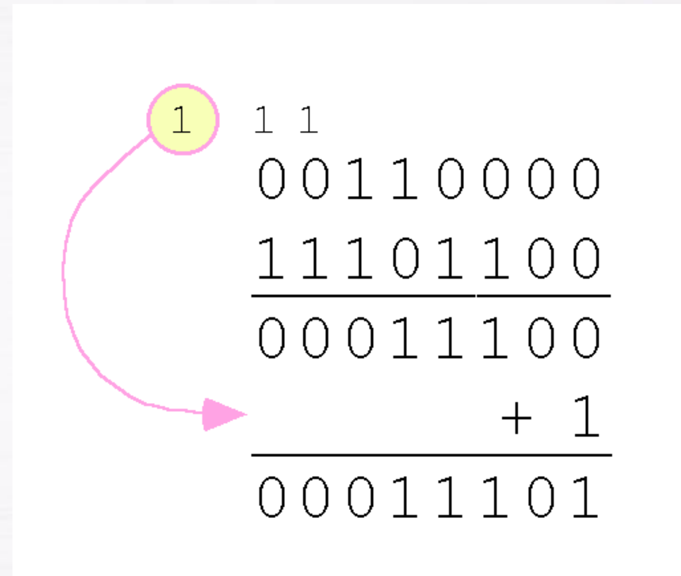
	+	–
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

- **negative numbers** are indicated by a **1 in the high order bit.**

- **difference** of two values is found by **adding the minuend to the complement of the subtrahend**.

2.4 Signed Integer Representation

- With one's complement addition, the **carry bit** (if there is one) is **"carried around"** and added to the sum.
  - Example: Compute 48 - 19



We note that 19 in binary is **00010011**,

so -19 in one's complement is: **11101100**.

2.4 Signed Integer Representation

- One’s complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having **two different representations for zero**:

  00000000 11111111

- Two’s complement solves this problem.

2.4 Signed Integer Representation

- To express a value in two's complement representation:
  - If the number is **positive**, just **convert it to binary** and you're done.
  - If the number is **negative**, find the **one's complement** of the number and then **add 1**.
- Example:
  - In 8-bit binary, 3 is:
    `00000011`
  - -3 using one's complement representation is:
    `11111100`
  - Adding 1 gives us -3 in two's complement form:
    `11111101`.

2.4 Signed Integer Representation

- With two's complement **addition**, all we do is **add** our **two binary numbers**. Just **discard any carries from the high order bit.**
  - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19.

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

We note that 19 in binary is: **00010011**

so -19 using one's complement is: **11101100**

and -19 using two's complement is: **11101101**

2.4 Signed Integer Representation

- **Excess-M representation** is another way to represent signed integers as binary values.
  - Excess-M representation is intuitive because the binary string with **all 0s represents the smallest number**, whereas the binary string **with all 1s represents the largest number**.
- An unsigned binary integer ***M*** (called the *bias*) **represents the value 0**, whereas **all zeroes** in the bit pattern **represents** the integer ***-M***.

2.4 Signed Integer Representation

- For ***n*-bit patterns**, we choose a bias of $\boldsymbol{M = 2^{n-1} - 1}$.
  - For example, if we were using **4-bit** representation, the bias should be $\mathbf{2^{4-1} - 1 = 7}$.

2.4 Signed Integer Representation

- The **binary value** of a **signed integer** using excess-M representation is **determined by adding *M* to that integer**.
  - Assuming that we are using **excess-7** representation, the **integer 0_{10} is represented as $0 + 7 = 7_{10} = 0111_2$.**
  - The integer **-7 is represented as $-7 + 7 = 0_{10} = 0000_2$** .
  - To find the **decimal value** of the **excess-7 binary number** 1111_2 **subtract 7**: **$1111_2 = 15_{10}$ and $15 - 7 = 8$**;

2.4 Signed Integer Representation

- Lets compare our representations:

Decimal	Binary (for absolute value)	Signed Magnitude	One's Complement
2	00000010	00000010	00000010
−2	00000010	10000010	11111101
100	01100100	01100100	01100100
−100	01100100	11100100	10011011

Decimal	Binary (for absolute value)	Two's Complement	Excess-127
2	00000010	00000010	10000001
−2	00000010	11111110	01111101
100	01100100	01100100	11100011
−100	01100100	10011100	00011011

2.4 Signed Integer Representation

- When we use any finite number of bits to represent a number, the result of our calculations may become too large or too small to be stored in the computer (**overflow**).
- For **unsigned numbers**, an overflow occurred if a **carry out of the leftmost bit** occurs
- For **signed numbers in complement representation**, an overflow occurred if the **carry in and carry out of the sign bit differs**

2.4 Signed Integer Representation

- Signed numbers in complement representation

Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	No
1100 + 1110	1010	Yes	No	Yes
1100 + 1010	0110	Yes	Yes	No

2.4 Signed Integer Representation

- We can do **binary multiplication and division by 2** very easily using an ***arithmetic shift*** operation
- A ***left arithmetic shift*** **inserts a 0 in for the rightmost bit** and shifts everything else left one bit; in effect, it **multiplies by 2**
- A ***right arithmetic shift*** shifts everything one bit to the right, but copies the sign bit; it **divides by 2**

2.4 Signed Integer Representation

Example:

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

00001011 (+11)

We **shift left** one place, resulting in:

00010110 (+22)

The sign bit has not changed, so the value is valid.

To multiply 11 by 4, we simply perform a left shift twice.

2.4 Signed Integer Representation

Example:

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

00001100 (+12)

We **shift right** one place, resulting in:

00000110 (+6)

(Remember, we carry the sign bit as we shift.)

To divide 12 by 4, we right shift twice.

2.4 Signed Integer Representation

- How to implement binary **multiplication by arbitrary number**?
- **Booth's multiplication algorithm** replaces arithmetic operations with bit shifting to the extent possible.

2.4 Signed Integer Representation

Booth's multiplication algorithm:

- Multiplies two signed binary values in two's complement notation

```
              0011    (multiplicand)
x             0110    (multiplier)
------------------
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

- Multiplies two signed binary values in two's complement notation
- Examines **adjacent pairs** of bits of the multiplier including an **implicit bit 0** below the least significant bit
- **Iterates over these pairs** from least to most significant bit
- If the multiplicand and multiplier are N-bits, then the **product will be 2N-bits**, all bits over 2N are ignored

```
      0011      (multiplicand)
x     0110(0)   (multiplier)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
        0011       (multiplicand)
x       0110(0)    (multiplier)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
        0011      (multiplicand)
x       0110(0)   (multiplier)
+ 00000000        (shift)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
          0011        (multiplicand)
x         0110(0)     (multiplier)
+ 00000000            (shift)
- 0000011             (subtract)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
            0011       (multiplicand)
x           0110(0)    (multiplier)
+ 00000000             (shift)
- 0000011              (subtract)
+ 000000               (shift)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
          0011        (multiplicand)
x         0110(0)     (multiplier)
+ 00000000            (shift)
- 0000011             (subtract)
+ 000000              (shift)
+ 00011               (add)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **add two's complement of multiplicand** to product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
          0011       (multiplicand)
x         0110(0)    (multiplier)
+ 00000000           (shift)
+ 1111101            (subtract)
+ 000000             (shift)
+ 00011              (add)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **add two's complement of multiplicand** to product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
          0011       (multiplicand)
x         0110(0)    (multiplier)
----------------
+ 00000000           (shift)
+ 1111101            (subtract)
+ 000000             (shift)
+ 00011              (add)
----------------
  00010010
```

We see that 3 × 6 = 18!