Digital Systems

John SUM
Institute of Technology Management
National Chung Hsing University
Taichung, ROC

March 23, 2026

Contents

1 Logic Gates **3**
1.1 Typical 2-Input Logic Gates 3
1.1.1 Electronic Circuits 3
1.1.2 Truth Tables 3
1.1.3 Notations 4
1.1.4 Non-Ideal Response 5
1.1.5 NAND and NOR Gates 7
1.2 Digital Logic Circuits 7
1.2.1 Two-Input XOR 7
1.2.2 Three-Input XOR 7
1.2.3 On Configuration 9
1.3 Is Arithmetic Be Done By a Digital Logic Circuit? 9

2 Arithmetical Operations **9**
2.1 Decimal number 9
2.2 Binary number 10
2.3 Number Conversion 11
2.3.1 Binary to Decimal 11
2.3.2 Decimal to Binary 12
2.4 Binary Addition 13
2.5 Binary Subtraction 14
2.6 Circuit Design for Add/Sub 16
2.7 Exercises 17

3 More on Binary Number Representations **18**
3.1 Fixed-point 19
3.1.1 Sign-Magnitude 19
3.1.2 2's-Compliment 20
3.1.3 Range 21

3.2 Floating-point 21
3.2.1 Representation 22
3.2.2 Interesting numbers 23
3.2.3 Arithmetic 24
3.2.4 FLOPS 24
3.3 Exercises 24

List of Figures

1 The output signal of an ideal AND gate with two streams of input signals. 5
2 The output signal of an non-ideal AND gate with two streams of input signals. The gate takes some time to reach the saturated output voltage level. 6
3 A configuration of using four NAND gates to implement a two-input XOR gate. 8
4 A configuration of using eight NAND gates to implement a three-input XOR gate. 8
5 Truth table of a full adder. 13
6 Schematic diagram of a 4-bit adder. 14
7 Truth table of an half-adder. 15
8 Schematic diagram of a circuit for doing addition and subtraction of two positive numbers. 17

1 Logic Gates

A computer is a basically a digital system. Loosely speaking, digital system is a system that is able to perform logical and arithmetical operations[1]. It consists of many digital logic circuits. Some of these circuits are implemented by simple logic gates to perform those operations. Some of them are control circuits. They control the activations of the logic gates (or logic circuits) and the connections amongst them.

1.1 Typical 2-Input Logic Gates

Logic gate is the basic building block of a digital system. Common logic gates include AND gate, OR gate, NAND gate, NOR gate, XOR[2] gate and NOT gate. NOT gate which is a single-input-single-output logic gate and the others are two-input-single-output logic gates.

1.1.1 Electronic Circuits

For electronic logic gates, the inputs to the gates are electrical signals (voltage). The logical signal could be defined in many different ways. For instance, 'high' voltage refers to 'TRUE' (or '1') and 'low' voltage refers to 'FALSE' (or '0'). Another setting is that 'positive' voltage refers to 'TRUE' (or '1') and 'negative' voltage refers to 'FALSE' (or '0'). As there are only two types of signal, this kind of signal is called binary (or digital). Output signal of a logic gate is also binary.

There are many specifications for high and low signals. One specification is that 5V for high signal and 0V for low signal. Another specification is that 5V for high and -5V for low. As both input and output signals are binary (i.e. digital), a logic gate performs simple logic function which implies the name logic gate. Therefore, the systems which are composed of these logic gates are digital systems.

Logic gate is itself an electronic circuit which consists of electronic components. In the early 20 century, vacuum tubes were used for such implementation. In 1940s, semiconductor were invented. Vacuum tubes were then replaced by semiconductor transistors to implement the logic gates.

1.1.2 Truth Tables

The logical operation of a logic gate is defined by its truth table. Conventionally, high voltage is represented by '1' and low voltage is represented by '0'. The truth tables for the NOT gate, the AND gate and the OR gate are shown below.

```
NOT GATE
```

[1] Precisely, a digital system can simply perform logical operations. Arithmetical operations are implemented based upon these logical operations.

[2] It should be noted that the function of XOR is not logic. However, in computer science field, we still say that it is a logic gate for convenient.

```
A Z
0 1
1 0

AND GATE
A B Z
0 0 0
0 1 0
1 0 0
1 1 1

OR GATE
A B Z
0 0 0
0 1 1
1 0 1
1 1 1
```

As a summary, the logical operations of those two-input logic gates are depicted in the following table.

A	B	OR	AND	XOR	NOR	NAND
0	0	0	0	0	1	1
0	1	1	0	1	0	1
1	0	1	0	1	0	1
1	1	1	1	0	0	0

1.1.3 Notations

The mathematical notations for the above logical operations are depicted in the following table.

Operation	Notation
NOT	$Z = \overline{A}$
OR	$Z = A + B$
AND	$Z = AB$
XOR	$Z = A \oplus B$
NOR	$Z = \overline{A + B}$
NAND	$Z = \overline{AB}$

Note that these notations are associated with a topic on Boolean algebra which tackles a logical operation with multiple inputs and multiple outputs. Here, those concepts will not be introduced. They are out of the scope of this course. In this course, only three special 3-input logical operations, as shown below, will be introduced with the use of the above notations.

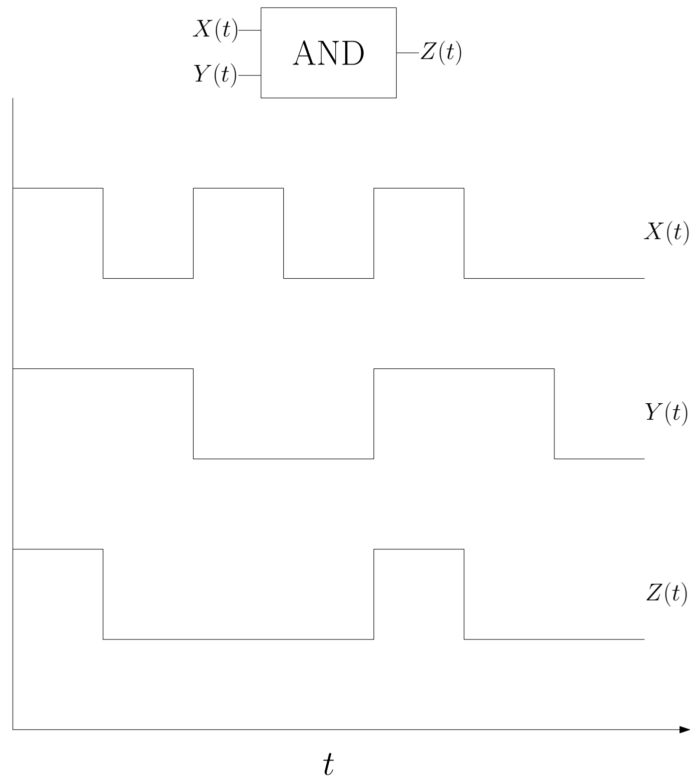


Figure 1: The output signal of an ideal AND gate with two streams of input signals.

Operation	Expression		
3-Input OR	$Z = A + B + C$	$Z = (A + B) + C$	$Z = A + (B + C)$
3-Input AND	$Z = ABC$	$Z = (AB)C$	$Z = A(BC)$
3-Input XOR	$Z = A \oplus B \oplus C$	$Z = (A \oplus B) \oplus C$	$Z = A \oplus (B \oplus C)$

1.1.4 Non-Ideal Response

For illustration, Figure 1 shows the output signal of an ideal AND gate with two streams of binary input signals. The output of the AND gate changes instantaneously when the input signals change. In reality, an AND gate might have several imperfections leading non-ideal response. The output voltage cannot change instantaneously. Figure 2 shows the output signal of an non-ideal AND gate with the same input signals. It takes a short time for the logic gate to change to the saturated output voltage level.

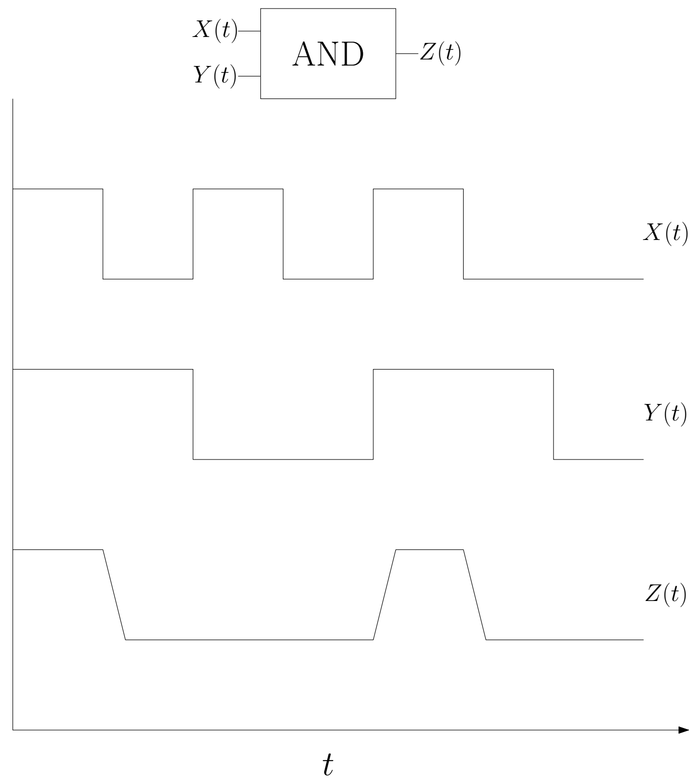


Figure 2: The output signal of an non-ideal AND gate with two streams of input signals. The gate takes some time to reach the saturated output voltage level.

1.1.5 NAND and NOR Gates

Among those logic gates introduced, the NAND and NOR gates are universal logic gates as any logic gate can be implemented simply by NAND (resp. NOR) gates only. For instance, a NOT gate can simply be implemented by a single NAND gate. Connecting both inputs as a single input, the NAND gate can be functioning as a NOT gate. To implement an AND gate, we can simply connect the output of an NAND gate to an NAND-gate-made NOT gate.

1.2 Digital Logic Circuits

The above logic gates are the basic building blocks for any digital logic circuit that performs a logical operation. The logical operation can be with any number of inputs and any number of outputs, such as the 3-input AND operation and the 3-input XOR operation their truth tables given below.

A	B	C	AND	XOR
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Thus, a digital logic circuit is just an electronic circuit which connecting multiple logic gates (equi. multiple electronic circuits) together to perform a single logical operation. Again, the logical operation can be with multiple inputs and multiple outputs.

1.2.1 Two-Input XOR

For illustration, let us give an exemplar digital logic circuit for using NAND only to implement a two-input XOR. To do so, we only need to use four NAND gates. Suppose there are nine NAND gates are available. A configuration of the NAND gates to perform two-input XOR is shown in Figure 3.

1.2.2 Three-Input XOR

Also for illustration, let us give an exemplar digital logic circuit for using NAND only to implement a three-input XOR. To do so, we only need to use eight NAND gates. Suppose there are nine NAND gates are available. A configuration of the NAND gates to perform three-input XOR is shown in Figure 4.

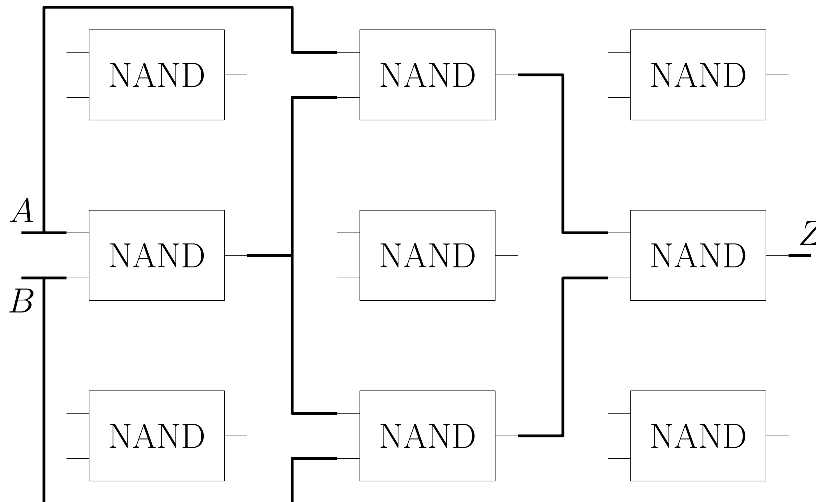


Figure 3: A configuration of using four NAND gates to implement a two-input XOR gate.

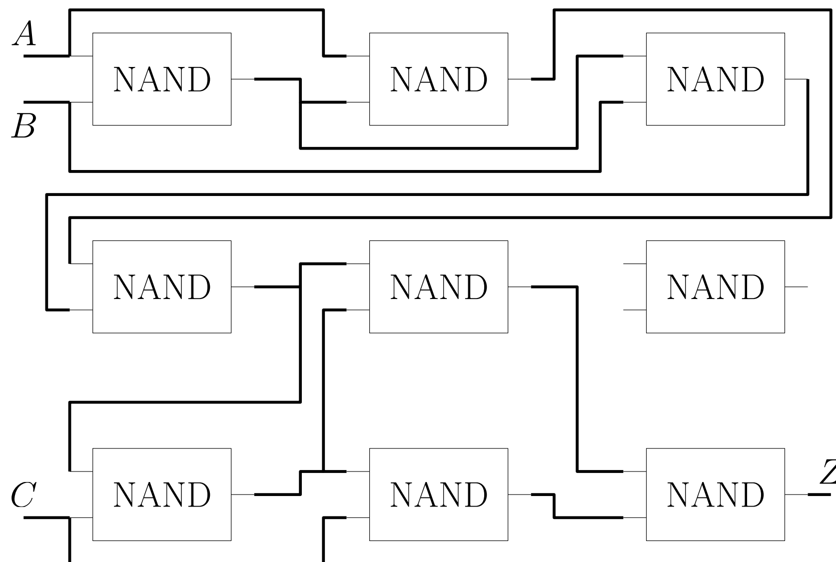


Figure 4: A configuration of using eight NAND gates to implement a three-input XOR gate.

1.2.3 On Configuration

From the above two examples, one can see that a lot of logical operations can be implemented by properly connecting the logic gates. In technical terms, we say that any logical operation can be implemented by properly configuring the logic gates. The examples presented above show two configurations for two different logical operations.

So, a challenging problem to a computer engineer is to design a re-configurable digital logic circuit which consists of a small number of logic gates but it can be configured to perform a large number of logical operations.

1.3 Is Arithmetic Be Done By a Digital Logic Circuit?

From the above arguments, it is clear that any logical operation can be done by a digital logic circuit if the number of logic gates are sufficient. Now, another problem is on arithmetic operation – *Is an algebraic operation (like addition, substraction, multiplication and division) be realized by a digital logic circuit?*

Today, the answer is clearly 'YES'. However, this 'YES' is an outcome of the engineers and scientists working for almost a century. In the early day, just adding two integer numbers were a problem.

2 Arithmetical Operations

To start with, we need to know how numbers are represented in decimal and binary forms.

2.1 Decimal number

In decimal numeral system (base-ten numeral system), we have 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, and so forth to represent positive integers. Look at the numbers, what can you see? Actually, we use symbols '0', '1', '2', '3', '4', '5', '6', '7', '8' and '9' to form the number.

For value which is in between zero and nine, we use one digit. For value which is in between ten and ninety-nine, we use two digits. For value which is in between one hundred to nine hundred and ninety-nine, we use three digits.

Let us start with the first ten numbers,

```
0 1 2 3 4 5 6 7 8 9.
```

The last digit has reached its largest value 9. So, return to the next line starting with two digits.

```
10 11 12 13 14 15 16 17 18 19.
```

The last significant digit of the last number has reached its largest value 9. So, return to the next line starting and the first digit increments by one, we get that

```
20 21 22 23 24 25 26 27 28 29.
```

gain, the least significant digit has reached its largest value 9. So, return to the next line starting and the first digit increments by one. After repeat until the last number is 79, the next line reads

```
80 81 82 83 84 85 86 87 88 89.
```

Again, the least significant digit has reached its largest value 9. So, return to the next line starting and the first digit increments by one. Now, the first digit in the next line is 9, i.e.

```
90 91 92 93 94 95 96 97 98 99.
```

Again, the least significant digit has reached its largest value 9. So, return to the next line starting and the first digit increments by one. However, the first digit has already reached its largest value. Now, we need three digits. That is,

```
100 101 102 103 104 105 106 107 108 109.
```

See! That is the way we make up the positive integers. One should note three important numbers, '0', '10' and '100'. They are the first number on the line. If we unify the number of digits to be 3, the numbers are listed as follows.

```
000 001 002 003 004 005 006 007 008 009
010 011 012 013 014 015 016 017 018 019
020 021 022 023 024 025 026 027 028 029
. . .
090 091 092 093 094 095 096 097 098 099
100 101 102 103 104 105 106 107 108 109
```

2.2 Binary number

Now, let us talk about the binary number. In binary numeral system, we use two symbols, '0' and '1'. In each line, we only allow '0' and '1'. '1' is the largest value. So, by the same principle as what we list the numbers in decimal numeral system, we can list all the numbers in binary form. The first line reads

```
0 1.
```

The rightmost digit has reached its largest value. So, we need to start a new line with two digits, i.e.

```
10 11.
```

The rightmost digit has reached its largest value. So, we need to start a new line with two digits. However, the first digit has reached its largest value. So, we need to start with a new line with three digits, i.e.

```
100 101.
```

The rightmost digit has reached its largest value. So, we need to start a new line with the second digit increments by one, i.e.

```
110 111.
```

Again, if we unify the number of digits to three, the above numbers will be listed as follows.

```
000 001
010 011
100 101
110 111
```

So, you see! The numbers '0', '1', '2', '3', '4', '5', '6', '7' in decimal form show quite different appearance in binary form.
Question: How do we know '101' in binary form is '5' in decimal form?
Answer: By counting.
Question: Would there be an easy way for us to identify its value in decimal form?
Answer: By using formulae.
Here are two examples.

$$101_2 = 1 \times 4 + 0 \times 2 + 1 \times 1 = 5_{10}.$$

$$100_2 = 1 \times 4 + 0 \times 2 + 0 \times 1 = 4_{10}.$$

4, 2, and 1 are the basis for three digit binary number. For a binary number with five digits, the basis will be 16, 8, 4, 2 and 1. The decimal value of '10110' is given by

$$10110_2 = 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 22_{10}.$$

Now, it is your turn.

2.3 Number Conversion

First of all, we need to indicate a number if is belongs to a binary number or a decimal number. For a binary number, we use subscript '2'. EG. 101_2. For a decimal number, we use subscript '10'. EG. 21_{10}. In this regard, we can see that 101_2 equals to 5_{10}, and 21_{10} equals to 10101_2. *As an introduction, let us start with the conversion of positive decimal (resp. binary) integers.*

2.3.1 Binary to Decimal

To convert a binary number to a decimal number, one needs the following table. Suppose that we want to convert a number 11011101_2 to decimal.

Weight	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Digit	1	1	0	1	1	1	0	1

Equivalently, the above table can be depicted as the following table.

Weight	128	64	32	16	8	4	2	1
Digit	1	1	0	1	1	1	0	1

Then, the decimal number of 11011101_2 is done by the following formulae.

$$\begin{aligned} 11011101_2 &= 1 \times 128_{10} + 1_{10} \times 64_{10} + 0 \times 32_{10} \\ &+ 1 \times 16_{10} + 1 \times 8_{10} + 1 \times 4_{10} \\ &+ 0 \times 2_{10} + 1 \\ &= 221_{10}. \end{aligned}$$

In summary, let an n-bit positive binary number, say X_2, is specified as $a_{n-1} \cdots a_1 a_0$, where $a_k \in \{0, 1\}$ for $k = 0, \cdots, n-1$. Its decimal number (X_{10}) can be calculated as follows :

$$X_{10} = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \cdots + a_1 \times 2^1 + a_0 \times 2^0.$$

As the largest number is the number with all ones, i.e. $a_k = 1$ for $k = 0, \cdots, n-1$, the largest number that can be represented by an n-bit is

$$2^{n-1} + 2^{n-2} + \cdots + 2^1 + 2^0,$$

which is equal to $2^n - 1$.

2.3.2 Decimal to Binary

To convert a decimal number to binary, the steps are different. We apply long division. Let us have two simple examples.

Convert 5_{10} to binary number.

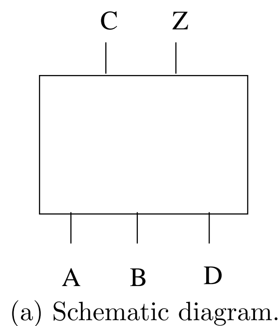
2	5	'1'
2	2	'0'
	'1'	

So, $5_{10} = 101_2$.

Convert 50_{10} to binary number.

2	50	'0'
2	25	'1'
2	12	'0'
2	6	'0'
2	3	'1'
	'1'	

So, $50_{10} = 110010_2$.



(a) Schematic diagram.

A	B	D	C	Z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(b) Truth table.

Figure 5: Truth table of a full adder.

2.4 Binary Addition

Addition of two binary numbers is similar to the addition of two decimal numbers. Let have three examples.

$$\begin{array}{rr} & 0101 \\ + & 0100 \\ \hline & 1001 \end{array} \qquad \begin{array}{rr} & 0011 \\ + & 0110 \\ \hline & 1001 \end{array} \qquad \begin{array}{rr} & 0111 \\ + & 0011 \\ \hline & 1010 \end{array}$$

The procedure starts from the least significant bit (LSB) and then moves to left until the most significant bit (MSB) has reached.

$$\begin{array}{rr} \text{Carry} & 0000 \\ & 0101 \\ + & 0100 \\ \hline & 0001 \end{array} \qquad \begin{array}{rr} \text{Carry} & 0000 \\ & 0101 \\ + & 0100 \\ \hline & 0001 \end{array} \qquad \begin{array}{rr} \text{Carry} & 1000 \\ & 0101 \\ + & 0100 \\ \hline & 0001 \end{array} \qquad \begin{array}{rr} \text{Carry} & 0000 \\ & 0101 \\ + & 0100 \\ \hline & 1001 \end{array}$$

The bitwise operation is performed by a logic circuit called full adder. Its schematic diagram and truth table are shown in Figure 5. To add two 4-bit (positive) numbers, we would need to have to connect four full-adders in the way as shown in Figure 6. For the sake of explanation, you can imagine that the steps of operation is first done at the rightmost full-adder. Then, the steps move from right to left.

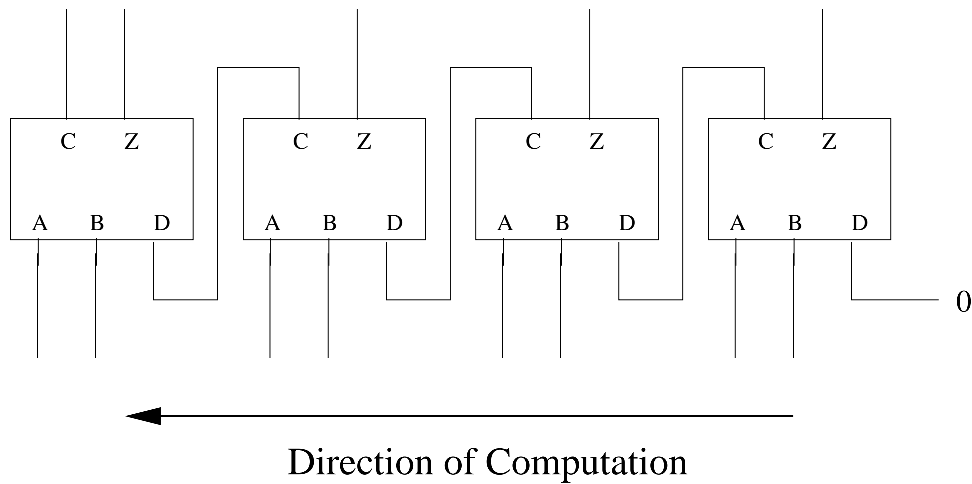


Figure 6: Schematic diagram of a 4-bit adder.

It should be noted that the circuit shown in Figure 6 consists of 8 inputs (4 inputs for each binary number) and 5 outputs. That means, this adder can allow the addition of 1111 and 1111. Moreover, one should note that the D-input of the rightmost adder is set to '0'.

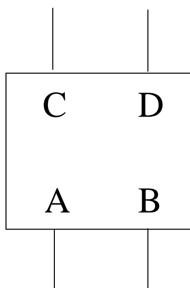
One design of a full-adder is by combining two simpler adders called half-adders. Its schematic diagram and truth table are shown in Figure 7. From the truth table, it is clear that the outputs C and D are essentially implemented by an XOR gate and an AND gate respectively. How two half-adders and an OR gate can implement a full-adder is left as an exercise for you.

2.5 Binary Subtraction

Before introducing the method of binary subtraction, we need to know how negative number is represented in binary. One simple approach is by using the leftmost bit as the sign bit. For a 4-bit format, the following table lists the numbers from -7 to 7.

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

Sign magnitude format.



(a) Schematic diagram.

A	B	C	D
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table.

Figure 7: Truth table of an half-adder.

While this formate is easy to understand, it is not simplest enough for subtraction operation.

Another approach is based on the idea of 2's complement. Here are the representations of -1 to -7 in 2's complement. As a reference, the representations of 0 to 7 are given.

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

2's complement format.

A simple three steps procedure can help you how to convert a negative number from its decimal form to 2's complement. Let say the number is $-M_{10}$

Step 1 Convert M_{10} to binary.

Step 2 Inversion all the bits.

Step 3 Add the inversion by '1'.

Example 1: Convert -6_{10} to its 2's complement, the following three steps give the result.

Step 1 $6_{10} \rightarrow 0110_2$.

Step 2 $0110_2 \rightarrow 1001_2$.

Step 3 $1001_2 + 0001_2 = 1010_2$.

Note that there are 15 numbers (from -7_{10} to 7_{10}) can be represented by 4 bits. For N bits, the range of numbers that can be represented is from $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$.

Example 2: Convert -91_{10} to its 2's complement, we need 8 bits.

Step 1 $91_{10} \rightarrow 01011011_2$.

Step 2 $01011011_2 \rightarrow 10100100_2$.

Step 3 $10100100_2 + 00000001_2 = 10100101_2$.

With the knowledge of 2's complement, it is time to learn the method of subtraction. Let start with an example.

Example 3: The way to calculate the value of $80_{10} - 26_{10}$ is by adding 80_{10} and -26_{10}, i.e.

$$\begin{aligned} 80_{10} - 26_{10} &= 80_{10} + (-26_{10}) \\ &= 01010000_2 + 11100110_2 \\ &= 00110110_2. \end{aligned}$$

The last step is essentially the binary addition.

2.6 Circuit Design for Add/Sub

Figure 8 shows a schematic diagram of a circuit for doing both addition and subtraction of two positive numbers which are in the range of $[0, 7]$. The operation of the "Normal/2's Comp" block is defined in the table below. The pin on the right side of the circuit is to control if the circuit is doing $A + B$ or $A - B$. Here, it is defined that the circuit performs $A + B$ if the signal is '0'. Otherwise, it performs $A - B$.

Number	$A+B$	$A-B$
0000	0000	0000
0001	0001	1111
0010	0010	1110
0011	0011	1101
0100	0100	1100
0101	0101	1011
0110	0110	1010
0111	0111	1001

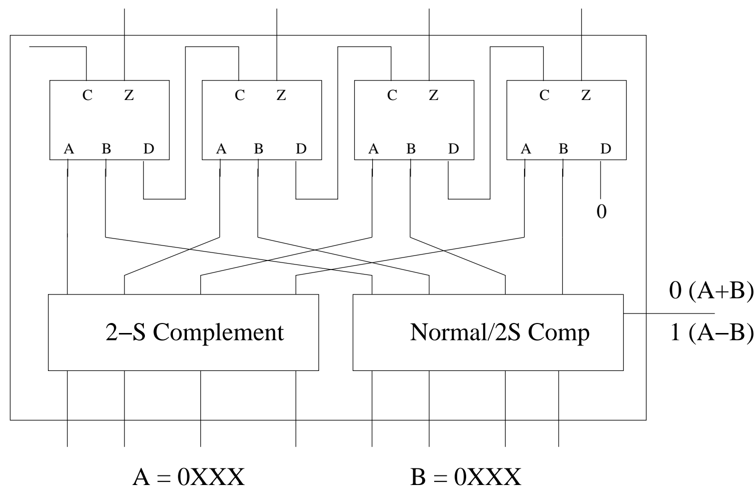


Figure 8: Schematic diagram of a circuit for doing addition and subtraction of two positive numbers.

Clearly, the circuit shown in Figure 8 is not the only design. There are many others.

2.7 Exercises

Question 1

1. What are the bases for a binary number with eight digits?
2. What is the value of '10000000' in decimal form?
3. What is the value of '10101001' in decimal form?
4. What is the value of '11011011' in decimal form?
5. What is the value of '00101010' in decimal form?
6. What is the value of '11111111' in decimal form?
7. What are the bases for a binary number with sixteen digits?
8. What is the value of '1000000000000000' in decimal form?
9. What is the value of '1111111111111111' in decimal form?
10. What is the largest number (in decimal form) of a binary number with thirty-two digits?
11. What is the largest number (in decimal form) of a binary number with sixty-four digits?

(Hint: The largest value of a binary number with N digits is $2^N - 1$. Why?)

Question 2

(a) Using an XOR and an AND gates to implement a half-adder.

(b) Using two half-adders and an OR to implement a full-adder.

Question 3

In the question, the binary number is of 8-bit format.

(a) Convert 20_{10} into binary number.

(b) Represent -12_{10} 2'S complement.

(c) Show the steps of obtaining the value of $20_{10} - 12_{10}$.

Question 4

In the question, the binary number is of 16-bit format.

(a) Convert 20_{10} into binary number.

(b) Represent -12_{10} 2'S complement.

(c) Show the steps of obtaining the value of $20_{10} - 12_{10}$.

Question 5

In the question, the binary number is of 16-bit format.

(a) Convert 58_{10} into binary number.

(b) Represent -78_{10} 2'S complement.

(c) Show the steps of obtaining the value of $58_{10} - 78_{10}$.

(d) Convert the answer in (c) in sign-magnitude form.

3 More on Binary Number Representations

Previous section on computer arithmetic has introduced two formats for binary number, the unsigned integer and 2's-compliment. In fact, there are many other formats for number representation and they could be categorized as fixed-point format and floating point format. Unsigned integer and 2's-compliment are two special cases of fixed-point format.

3.1 Fixed-point

3.1.1 Sign-Magnitude

To represent a decimal number, say -4.75, in binary format, we need to have a binary point and a sign bit. One representation called sign-magnitude format is given below.

$$\begin{aligned}
-4.75_{10} &= -\left(2^2 + 2^{-1} + 2^{-2}\right) \\
&= -\left(0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3}\right) \\
&= 10100.110_2.
\end{aligned}$$

The general form of a $(m + q + 1)$-bit format is given by

$$\text{Sign} \times \left(a_m \times 2^m + \cdots a_1 \times 2^1 + a_0 \times 2^0 + b_1 \times 2^{-1} \cdots b_q \times 2^{-q}\right),$$

where m is the number of integer bits and q is the number of fractional bits. Remember that the leftmost bit is sign bit.

To convert a fractional number into binary, we apply the method of subtraction instead of long division.

$$0.75 = 2^{-1} + 0.25 = 2^{-1} + 2^{-2}.$$

Note that not all fractional number can be represented by finite number of fractional bits. Here gives you three examples.

$$\begin{aligned}
0.236_{10} &= 0.0011110001101010011111011111\cdots \\
0.4852_{10} &= 0.0111110000110110000100010011 0\cdots \\
0.1_{10} &= 0.0001100110011001100110011 0011\cdots
\end{aligned}$$

Even for the decimal numbers 0.1 and 0.2, they cannot be represented by finite number of fractional bits. In this regard, these numbers have to be truncated (i.e. chopped) or rounded. To illustrate the ideas, let the number of fractional bits be 5.

Truncation : The idea of truncation is simply by chopping. Only the first five fractional bits are kept and ignore the others. Another name for this method is called round-down.

$$\begin{aligned}
0.236_{10} &= 0.00111 \\
0.4852_{10} &= 0.01111 \\
0.1_{10} &= 0.00011.
\end{aligned}$$

The maximum error, called precision error, committed due to truncation is 2^{-q}. In the above examples, the precision error is 2^{-5}.

Rounding : The method of rounding is the same as what we have learnt in elementary mathematics, round to nearest number. Here are two examples.

$$\begin{aligned}[0.236] &= 0.24 \text{ (Round to two decimal points!)} \\ [0.4852] &= 0.485 \text{ (Round to three decimal points!)}.\end{aligned}$$

By the same principle, by rounding the above binary numbers to five fractional bits, we get that

$$\begin{aligned}0.236_{10} &= 0.01000 \\ 0.4852_{10} &= 0.10000 \\ 0.1_{10} &= 0.00011.\end{aligned}$$

As compared with the numbers obtained by truncation, the first two numbers are represented differently. The precision error, committed due to truncation is $2^{-(q+1)}$. In the above examples, the precision error is 2^{-6}.

Guard bits : It is clear that precision error will be amplified if there are a large amount of numbers to be added or multiplied. To overcome such problem, the number to be operated in the CPU is converted to a higher precision format, say from 16 bits to 20 bits or 32 bits. All arithmetic operations are thus conducted in this high precision level. Here is an example. Suppose the numbers are represented in 8-bit fixed point with 5 fractional bits, i.e. $F(8,5)$. In the CPU, the number of bits is 16.

$$\begin{aligned}0.4852_{10} \times 0.1_{10} &= 0.10000 \times 0.00011 \\ &= 000.10000 \times 000.00011 \\ &= 000.1000000000000 \times 000.0001100000000 \\ &= 000.0000100000000 + 000.0000010000000 \\ &= 000.0000110000000 \\ &= 000.000011\end{aligned}$$

The extra bits are called the guard bits.

3.1.2 2's-Compliment

To represent a fractional number in 2's-compliment, the idea and the conversion method are the same as for integer number. To make it easier to understand, we consider the number which can be represented by finite number of fractional bits.

$$3.71875_{10} = 11.10111_2.$$

Now, we would like to convert -3.71875_{10} into binary. Here, we assume that the number of integer bits is 2 and the number of fractional bits is 5.

Step 1 $3.71875_{10} \rightarrow 011.10111_2$.

Step 2 $011.10111_2 \rightarrow 100.01000_2$.

Step 3 $100.01000_2 + 000.00001_2 = 100.01001_2$.

So, we can check that the beauty of 2's-compliment format in subtraction preserves.

$$\begin{aligned} 3.71875_{10} - 3.71875_{10} &= 3.71875_{10} + (-3.71875_{10}) \\ &= 011.10111_2 + 100.01001_2 \\ &= 000.00000_2 \\ &= 0_{10}. \end{aligned}$$

The addition in the second line is implemented by 8-bit full adder.

3.1.3 Range

For a fixed-point with m integer bits and q fractional bits, the maximum and minimum numbers it can be represented are given below.

$$\text{Max} = +\underbrace{111\cdots111}_{m \text{ bits}}.\underbrace{111\cdots111}_{q \text{ bits}}. \quad \text{Min} = -\underbrace{111\cdots111}_{m \text{ bits}}.\underbrace{111\cdots111}_{q \text{ bits}}.$$

The maximum number is given by

$$\begin{aligned} \text{Max} &= \sum_{i=0}^{m} 2^i + \sum_{j=1}^{q} 2^{-j} \\ &= 2^{-q} \times \left(\sum_{i=0}^{m+q} 2^i \right) \\ &= 2^{-q} \left(2^{m+q+1} - 1 \right) \\ &= 2^{m+1} - 2^{-q}. \end{aligned}$$

Therefore, the binary numbers that can be represented are in the range

$$\left[-\left(2^{m+1} - 2^{-q}\right), \left(2^{m+1} - 2^{-q}\right) \right].$$

The interval between two consecutive numbers is 2^{-q}.

3.2 Floating-point

As the range of fixed-point format is finite, it is not possible to represent the numbers outside the range. In this regard, we need another format to represent large numbers. It is the floating-point format.

Recall that the speed of light is 299792458 meters per second (m/s). Sometimes, we write it approximately as $3.0 \times 10^8 m/s$. It is already a base-10 floating-point format. Three important information in this representation, (i) the sign (i.e. '+'), (ii) the significant (i.e. 3.0) and (iii) the exponent (i.e. 8).

3.2.1 Representation

To illustrate how a fractional number can be represented in floating-point, let us consider the same number 3.71875_{10} (equivalently 11.10111_2). To convert it to a floating-point, we carry the following steps. Here, we assume that there is one sign bit and the number of significant bits is 11. The number of exponent bits is 5 and the exponent bias is 15. It is exactly the half-precision floating-point format as defined in IEEE 754-2008 standard.

Step 1 $3.71875_{10} \rightarrow 011.10111_2$.

Step 2 $011.10111_2 \rightarrow 1.110111 \times 2^1$. Significant bits are 11011100000.

Step 3 Exponent $= 1 + 15 = 16$. Exponent bits are 10000.

Step 4 $3.71875_{10} \rightarrow 0100001101110000_{hf}$.

Step 2 is the normalization step. The subscript hf stands for half-float.

Let us consider another slightly different number 24.71875_{10}. Clearly, the binary bits for the fractional part is the same. For the integer part, it is represented by 11000_2. So, the conversion can be done by the following steps.

Step 1 $24.71875_{10} \rightarrow 11000.10111_2$.

Step 2 $11000.10111_2 \rightarrow 1.100010111 \times 2^4$.

Step 3 Exponent $= 4 + 15 = 19$. Exponent bits are 10011.

Step 4 $24.71875_{10} \rightarrow 0100111000101110_{hf}$.

To represent a negative number in floating format, simply turn the sign bit in '1'. For instance,

$$\begin{aligned} -3.71875_{10} &= 1100001101110000_{hf} \\ -24.71875_{10} &= 1100111000101110_{hf} \end{aligned}$$

Let us consider the number 2^{-16}. Based on the normalization step, the number should be represented as follows :

$$2^{-16} = 1.00000000000 \times 2^{-16}.$$

However, the minimum power allowed is -14. In such case, there is no scale-down normalization. Instead, normalization is performed in a manner of scale-up, i.e.

$$2^{-16} = 0.01000000000 \times 2^{-14}.$$

Those numbers smaller than 2^{-14} are called subnormal numbers. So, the conversion of 2^{-16} can be done by the following steps.

Step 1 $2_{10}^{-16} \rightarrow 0.0000000000000001_2$.

Step 2 $0.0000000000000001_2 \rightarrow 0.01 \times 2^{-14}$.

Step 3 As the number is smaller than 2^{-14}, exponent bits are 00000.

Step 4 $2_{10}^{-16} \rightarrow 0000000100000000_{hf}$.

Step 3 is not the same as before. The exponent 00000 is to indicate that the number is subnormal, i.e. smaller than 2^{-14}.

To clarify this point, let us consider the number $2^{-14} + 2^{-16}$. So, the conversion of 2^{-16} can be done by the following steps.

Step 1 $\left(2^{-14} + 2^{-16}\right)_{10} \rightarrow 0.0000000000000101_2$.

Step 2 $0.0000000000000101_2 \rightarrow 1.01 \times 2^{-14}$.

Step 3 Exponent $= -14 + 15 = 1$. Exponent bits are 00001.

Step 4 $\left(2^{-14} + 2^{-16}\right)_{10} \rightarrow 0000010100000000_{hf}$.

3.2.2 Interesting numbers

The smallest positive *subnormal* number that can be represented by the above half-precision floating-point is given by

$$\begin{aligned} 0\ 00000\ 0000000001 &= 0.0000000001_2 \times 2^{-14} \\ &= 2^{-11} \times 2^{-14} \\ &= 2^{-25}. \end{aligned}$$

The largest positive *subnormal* number that can be represented by the above half-precision floating-point is given by

$$\begin{aligned} 0\ 00000\ 1111111111 &= 0.1111111111_2 \times 2^{-14} \\ &= \left(1 - 2^{-11}\right) \times 2^{-14} \\ &= 2^{-14} - 2^{-25}. \end{aligned}$$

Here are a few other interesting numbers.

$$0\ 00001\ 0000000000 = 2^{-14}.$$

$$\begin{aligned} 0\ 01111\ 0000000000 &= 1.0000000000_2 \times 2^{15-15} \\ &= 1. \end{aligned}$$

$$\begin{aligned} 0\ 11110\ 1111111111 &= 1.1111111111_2 \times 2^{15} \\ &= \left(2 - 2^{-11}\right) \times 2^{15} \\ &= 65520_{10}. \end{aligned}$$

Note that the largest exponent is 11110, not 11111. The bit pattern 11111 is reserved for other use. Numbers larger than 65520 cannot be represented.

$$
\begin{aligned}
0\ 11111\ 00000000000 &= \infty \\
1\ 11111\ 00000000000 &= -\infty
\end{aligned}
$$

Here, only the 16-bit half-precision floating-point has been introduced. In many processors, the number of bits for a floating-point could be 32 bits, 64 bits and even 128 bits. Their principles of representing a number are almost the same. It consists of a sign bit, a number of bits for exponent and a number of bits for significant. For the exponent, it embraces with an exponent bias. If the exponent bits are all zeros, the number is subnormal.

3.2.3 Arithmetic

Floating-point arithmetic is more complicated than fixed-point arithmetic. For instance, addition of two numbers will need to normalize the numbers with same exponent before doing the addition.

$$
\begin{aligned}
& 1.00010010011 \times 2^8 + 1.01011000000 \times 2^6 \\
= \;& 1.00010010011 \times 2^8 + 0.01010110000 \times 2^8 \\
= \;& 1.01101000011 \times 2^8.
\end{aligned}
$$

For multiplication, the computation could be intensive (see the second step).

$$
\begin{aligned}
& \left(1.00010010000 \times 2^8\right) \times \left(1.01000000000 \times 2^6\right) \\
= \;& (1.00010010000 \times 1.01000000000) \times 2^{14} \\
= \;& (1.00010010000 + 0.01000100100) \times 2^{14} \\
= \;& 1.01010110100 \times 2^{14}.
\end{aligned}
$$

Here, the number 1.01000000000 consists of one bit '1' in the fractional part. So, there is only one addition. If there are many bits '1', the number of addition could be large.

3.2.4 FLOPS

The number of steps in performing a single floating-point arithmetic, like addition or multiplication, is more than fixed-point arithmetic. Today, the number of floating-point operations in a second (equivalently, floating-point operations per second (FLOPS)) becomes the default measure for computer performance. Today, an Nvidia GPU can have processing speed up close to 10^{15} FLOPS.

3.3 Exercises

1. Convert -32.125 in 16-bit 2's compliment fixed-point format with 5 integer bits and 10 fractional bits.

2. Describe the steps how two fixed-point numbers are multiplied.
3. Convert -32.125 in IEEE 754 half-precision floating point format.
4. Convert the following half-precision floating numbers to decimal numbers.
    (a) 1000001010101010.
    (b) 0011100010100000.
5. Describe the steps how two floating numbers are multiplied.