

# Revision notes - CS2100

Ma Hongqiang

October 6, 2017

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Number Systems</b>	<b>4</b>
<b>3</b>	<b>Boolean Algebra</b>	<b>15</b>
<b>4</b>	<b>Logic Gates and Circuits</b>	<b>19</b>
<b>5</b>	<b>Kaunaugh Map</b>	<b>23</b>
<b>6</b>	<b>Combinatorial Circuits</b>	<b>26</b>
<b>7</b>	<b>More Building Blocks</b>	<b>35</b>
<b>8</b>	<b>Sequential Logic</b>	<b>42</b>
<b>9</b>	<b>Understanding Performance</b>	<b>50</b>

# 1 Introduction

**Definition 1.1** (Computer).

A computer is a device capable of solving problems according to designed programs. It simply augments our power of storage and speed of calculation.

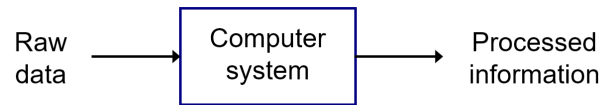


Figure 1: Computers as information processors.

**Definition 1.2** (Hardware Stack).

The hardware stack with the most basic on the top goes like:

- Transistor
- Logic Gate
- Circuits
- Memory
- Processor

**Definition 1.3** (Transistor).

A **transistor** is

- a *solid state switch*. The input switches on or off the output.
- It is also an *amplifier*. The output signal is much stronger than the input so that things can be connected up.

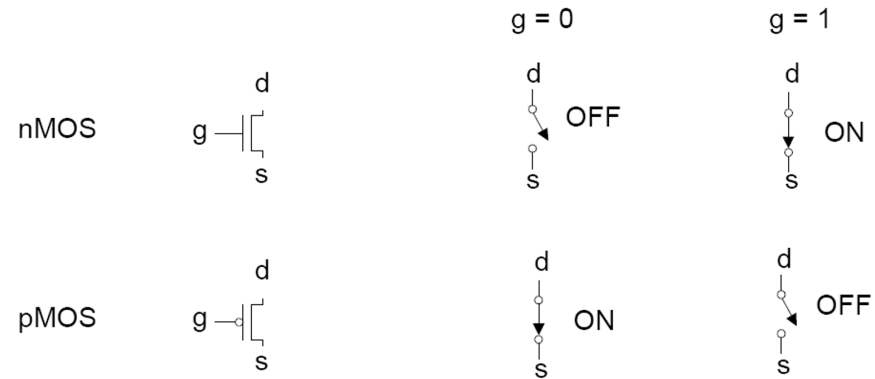
**Definition 1.4** (Boolean logic gates). *t*

To compute, **Boolean logic gates** is built by transistors to compute Boolean logic functions.

The basic Boolean logic gates include:

- NOT
- OR, AND
- NAND, NOR

**Theorem 1.1** (Behaviour of nMOS and pMOS transistor).



Examples of logic gates constructed by nMOS and pMOS include:

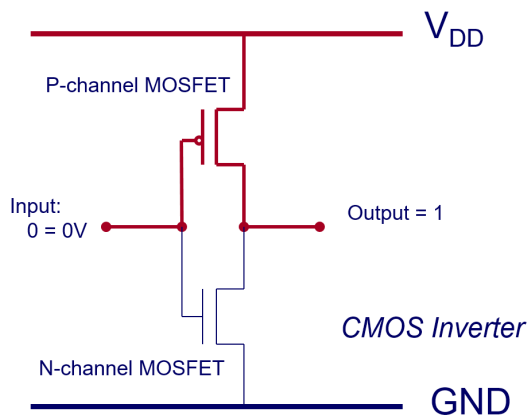


Figure 2: CMOS NOT Gate

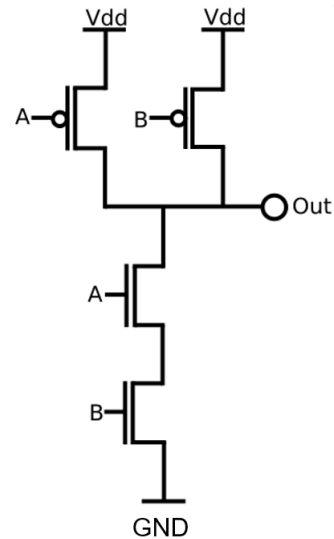


Figure 3: CMOS NAND Gate

## 2 Number Systems

### 2.1 Information Representation

**Definition 2.1** (Bit).

**Bit** is the short form of *binary digit*.

- 0 and 1
- Represent **false** and **true** in logic
- Represent the *low* and *high* states in electronic devices

Other units include

- Byte: 8 bits
- Nibble: 4 bits
- Word: Multiple of byte

Obviously,  $N$  bits can represent up to  $2^N$  values.

Conversely, to represent  $M$  values,  $\log_2 M$  bits are required.

**Definition 2.2** (Weighted-positional Number System).

A **weighted-positional number system** is one whose

- **Base** or **radix** is  $N$ .
- position is important, as the value of each symbol/digit is dependent on its **type** and **position** in the number.
- In general,

$$(a_n a_{n-1} \cdot a_1 a_0 . b_1 b_2 \cdot)_N = \sum_{k=0}^n a_k N^k + \sum_{k=1}^{\infty} b_k N^{-k}$$

For example, in the decimal number system,

$$(593.68)_{10} = 5 \times 10^2 + 9 \times 10^1 + 3 \times 10^0 + 6 \times 10^{-1} + 8 \times 10^{-2}$$

The method of conversion between bases can be found in **MA2213 Revision Note**.

In special cases, binary can be converted to octal and hexadecimal by partitioning the number in groups of 3 and 4 respectively.

## 2.2 Signed Binary Number

Any real number can be converted to a signed binary number. A **signed binary number** is defined by its

- **sign**
- **absolute value**

In general, a signed binary number can be represented as

$$\pm a_{n-1}a_{n-2} \cdots a_0.b_1b_2 \cdots$$

where  $a_i, b_j = 0$  or  $1$  for  $i = [0..n]$ ,  $j = \mathbb{Z}^+$ .

**Definition 2.3** (String Representation of Signed Binary Number).

A **string representation** of signed binary numbers is a **bijection** from **signed binary numbers** to **strings of bits**.

Specifically for binary integers with sign  $s$  and absolute value  $v$ , define the bijective function  $f$ :

$$\begin{aligned} f : (\text{sign}, \text{absolute value}) &\rightarrow \text{binary String} \\ (s, v) &\mapsto \mathbf{str} \end{aligned}$$

where  $\mathbf{str}$  is the string representation  $f$  of that particular signed binary number defined by  $s$  and  $v$ .

**Definition 2.4** (Negation of String Representation).

Negation is a unitary function  $-$ :

$$\begin{aligned} - : \text{binary String} &\rightarrow \text{binary String} \\ \mathbf{str} &\mapsto -\mathbf{str} \end{aligned}$$

where  $-\mathbf{str} := f(-s, v)$

There are three common string representations of signed binary number, namely

- Sign-and-Magnitude  $f_{\text{sm}}$
- 1s complement  $f_{1s}$
- 2s complement  $f_{2s}$

In the rest of this subsection, the length of string  $\mathbf{str}$  is fixed to  $n$ , and  $\mathbf{str} = a_{n-1}a_{n-2} \cdots a_0$ .

### 2.2.1 Sign-and-Magnitude Representation $f_{\text{sm}}$

**Definition 2.5** ( $f_{\text{sm}}$ ).

In **sign-and-magnitude** representation, sign  $s$  is represented by a **sign bit** in the leftmost position of string **str**, i.e,  $a_{n-1}$ .

- 0 for +
- 1 for −

The absolute value  $v$  will occupy the rest  $n - 1$  bits of **str**:  $a_{n-2}a_{n-3} \cdots a_0 := v$ .

For a  $n$  bit sign-and-magnitude representation, the domain of  $f_{\text{sm}}$  is  $[-2^{n-1} + 1, 2^{n-1} - 1] \cap \mathbb{Z}$ .

Clearly, say, for 8-bit sign-and-magnitude representation,

- **Largest value**:  $01111111_{\text{sm}} = +127_{10}$
- **Smallest value**:  $11111111_{\text{sm}} = -127_{10}$
- **Zeros**:  $00000000_{\text{sm}} = +0_{10}$  and  $10000000_{\text{sm}} = -0_{10}$
- **Range**:  $-127_{10}$  to  $+127_{10}$

**Theorem 2.1** (Negation of  $str$  in sign-and-magnitude representation).

To **negate** a **str** in sign-and-magnitude interpretation, **invert the sign bit**<sup>1</sup>. Suppose **str** =  $a_{n-1}a_{n-2} \cdots a_0$ , then

$$-\text{str} = \overline{a_{n-1}}a_{n-2} \cdots a_0$$

**Theorem 2.2** ( $f_{\text{sm}}^{-1}$ ).

$f^{-1}(\text{str})$  is defined<sup>2</sup> as follows:

$$\begin{aligned} f_{\text{sm}}^{-1}(\text{str}) &= f^{-1}(a_{n-1}a_{n-2} \cdots a_0) \\ &:= (-1)^{a_{n-1}} \times \sum_{i=0}^{n-2} (a_i \times 2^i) \end{aligned}$$

### 2.2.2 1s Complement

**Definition 2.6** ( $f_{1s}$  for non-negative binary numbers).

Suppose a **nonnegative** number is defined by  $(+, v)$ . In **1s complement** representation **str**,

- the positive sign defines  $a_{n-1} := 0$ ;
- the absolute value  $v$  will occupy the rest  $n - 1$  bits of **str**:  $a_{n-2}a_{n-3} \cdots a_0 := v$ .

---

<sup>1</sup>Inversion of bit  $b$  is denoted by  $\bar{b}$

<sup>2</sup> $f^{-1}$  is well defined as  $f$  is a bijection

**Definition 2.7** (Negation).

Negation of 1s complement representation is defined as:

$$\begin{aligned} - : \text{binary String} &\rightarrow \text{binary String} \\ \mathbf{str} = a_{n-1}a_{n-2}\cdots a_0 &\mapsto \overline{a_{n-1}a_{n-2}\cdots a_0} := -\mathbf{str} \end{aligned}$$

Essentially, to negate a **String** of 1s complement, **invert all the bits**.

**Definition 2.8** ( $f_{1s}$  for non-positive binary numbers).

The 1s complement representation of a **non-positive binary number** defined by  $(-, v)$  is defined by **negation** of  $f_{1s}((+, v))$ .

Essentially,

$$\mathbf{str} = 1\overline{a_{n-2}a_{n-3}\cdots a_0}$$

Together with the previous definition, for a  $n$  bit 1s complement representation, the domain of  $f_{1s}$  is  $[-2^{n-1} + 1, 2^{n-1} - 1] \cap \mathbb{Z}$ .

Clearly, say, for 8-bit 1s complement representation,

- **Largest value:**  $01111111_{1s} = +127_{10}$
- **Smallest value:**  $10000000_{1s} = -127_{10}$
- **Zeros:**  $00000000_{1s} = +0_{10}$  and  $11111111_{1s} = -0_{10}$
- **Range:**  $-127_{10}$  to  $+127_{10}$ .

**Theorem 2.3** (Sign Bit of 1s Complement).

The leftmost position of string **str**, i.e,  $a_{n-1}$ , still represents the sign:

- 0 for +
- 1 for -

**Theorem 2.4** ( $f_{1s}^{-1}$ ).

$f_{1s}^{-1}(\mathbf{str})$  is defined as follows:

$$\begin{aligned} f_{1s}^{-1}(\mathbf{str}) &= f^{-1}(a_{n-1}a_{n-2}\cdots a_0) \\ &:= ((-2^{n-1} + 1) \times a_{n-1}) + \sum_{i=0}^{n-2} a_i \times 2^i \end{aligned}$$

**2.2.3 2s Complement****Definition 2.9** ( $f_{2s}$  for non-negative binary numbers).

Suppose a **nonnegative** number is defined by  $(+, v)$ . In **2s complement** representation **str**,

- the positive sign defines  $a_{n-1} := 0$ ;
- the absolute value  $v$  will occupy the rest  $n - 1$  bits of **str**:  $a_{n-2}a_{n-3}\cdots a_0 := v$ .

**Definition 2.10** (Negation).

Negation of 2s complement representation is defined as:

$$\begin{aligned} - : \text{binary String} &\rightarrow \text{binary String} \\ \mathbf{str} = a_{n-1}a_{n-2} \cdots a_0 &\mapsto (\text{String})((\text{binary number})\overline{a_{n-1}a_{n-2} \cdots a_0} + 1) := -\mathbf{str} \end{aligned}$$

Essentially, negation of a **String** of 2s complement equals to the sum of this **String** with all bits flipped and 1.

**Definition 2.11** ( $f_{2s}$  for negative binary numbers).

The 2s complement representation of a **negative binary number** defined by  $(-, v)$  is defined by **negation** of  $f_{2s}((+, v))$ .

Essentially,

$$\mathbf{str} = \overline{a_{n-1}a_{n-2} \cdots a_0} + 1$$

Together with the previous definition, for a  $n$  bit 2s complement representation, the domain of  $f_{2s}$  is  $[-2^{n-1}, 2^{n-1} - 1] \cap \mathbb{Z}$ .

Clearly, say, for 8-bit 2s complement representation,

- **Largest value:**  $01111111_{2s} = +127_{10}$
- **Smallest value:**  $10000000_{2s} = -128_{10}$
- **Zeros:**  $00000000_{2s} = +0_{10}$
- **Range:**  $-128_{10}$  to  $+127_{10}$ .

**Theorem 2.5** (Sign Bit of 2s Complement).

The leftmost position of string **str**, i.e,  $a_{n-1}$ , still represents the sign:

- 0 for +
- 1 for -

**Theorem 2.6** ( $f_{2s}^{-1}$ ).

$f_{2s}^{-1}(\mathbf{str})$  is defined as follows:

$$\begin{aligned} f_{2s}^{-1}(\mathbf{str}) &= f_{2s}^{-1}(a_{n-1}a_{n-2} \cdots a_0) \\ &:= (-2^{n-1} \times a_{n-1}) + \sum_{i=0}^{n-2} a_i \times 2^i \end{aligned}$$

## 2.3 Generalising complement

**Definition 2.12** ( $(r-1)$ 's complement).

Let  $a_{n-1}a_{n-2} \cdots a_0$  be string representation of a number in radix  $r$ . The  $(r-1)$ 's **complement** is the string  $\overline{a_{n-1}a_{n-2} \cdots a_0}$  where  $\overline{a_i} = r - 1 - a_i$ .

The  $r$ 's **complement** is just the  $(r-1)$ 's complement with 1 added to the least significant bit.



**Theorem 2.7** (Complement on Fractions).

We can extend the operations of complement on fractions.

**Theorem 2.8** (2s Complement Addition/Subtraction).

Algorithm for **addition**,  $A_{2s} + B_{2s}$ :

- Perform binary addition on the two (binary number) String.
- Ignore the carry out of the most significant bit(MSB).
- Check for overflow. Overflow occurs if
  1. the 'carry in' and 'carry out' of the MSB are different, or
  2. result is of opposite sign of  $A_{2s}$  and  $B_{2s}$ .

Algorithm for **subtraction**  $A_{2s} - B_{2s}$ :  $A_{2s} - B_{2s} = A_{2s} + (-B)_{2s}$ .

**Theorem 2.9** (1s Complement Addition/Subtraction).

Algorithm for **addition**,  $A_{1s} + B_{1s}$ :

- Perform binary addition on the two (binary number) String.
- If there is a carry out of the MSB, add 1 to the result.
- Check for overflow. Overflow occurs if
  1. result is of opposite sign of  $A_{1s}$  and  $B_{1s}$ .

Algorithm for **subtraction**  $A_{1s} - B_{1s}$ :  $A_{1s} - B_{1s} = A_{1s} + (-B)_{1s}$ .

## 2.4 Excess- $k$ Representation

**Definition 2.13** ( $f_{\text{excess}-k}$ ).

Suppose a number  $N$  is defined by  $(s, v)$ . Clearly, this number  $N$  equals  $\text{sgn}(s) \times v$ . Its **excess- $k$**  representation ( $k > 0$ ) **str** is defined as

$$\mathbf{str} = f_{\text{excess}-k}((s, v)) := f_{\text{sm}}(N + k)$$

For a  $n$  bit excess- $k$  complement representation, the domain is  $[-k, 2^n - k - 1] \cap \mathbb{Z}$ .

Note:  $k_{\text{excess}-k} = k_2$  numerically.

**Definition 2.14** (Negation).

Negation of excess- $k$  representation of is calculated as:

$$-\mathbf{str} := \text{String}(2 \times k - (\text{binary number})\mathbf{str})$$

Domain of the above negation operation is  $[-\min\{2^n - k - 1, k\}, \min\{2^n - k - 1, k\}]$  if  $k < 2^n$  and  $\emptyset$  otherwise.

There is no **sign bit** for excess- $k$  representation.

**Definition 2.15** ( $f_{\text{excess}-k}^{-1}$ ).

$f_{\text{excess}-k}^{-1}$  is defined as follows:

$$f_{\text{excess}-k}^{-1}(\mathbf{str}) = (\text{binary number})\mathbf{str} - k$$

## 2.5 Floating Point Numbers

**Definition 2.16** (Fixed Point Numbers).

In fixed point representation, the binary point is assumed to be at fixed location. In general, the binary point may be assumed to be at any pre-fixed location.

Fixed point numbers have limited range. Floating point numbers allow us to represent very large or very small numbers.

**Definition 2.17** (Floating Point Numbers).

**Floating point numbers** consists of 3 parts: **sign**, **mantissa** and **exponent**.

- The base (radix) is assumed to be 2.
- Sign bit: 0 for positive, 1 for negative.
- Mantissa is usually in **normalised form**.

Clearly, the trade-off of floating point numbers is

- More bits in mantissa  $\rightarrow$  better precision
- More bits in exponent  $\rightarrow$  larger range of values

**Definition 2.18** (IEEE Standard 754).

IEEE Standard 754 has the following properties:

- Two types of formats
  - **Normalised** numbers
  - **Denormalised** numbers
- Special values
  - Negative zero
  - Infinities
  - Not-a-Number(NaN)
- Distribution of bits in mantissa and exponent: See table below

Parameter	Single	Double
No. of fraction bits	23	52
Maximum exponent	+127	+1023
Minimum exponent	-126	-1022
Exponent bias	+127	+1023
Exponent width in bits	8	11
Format width in bits	32	64

The IEEE Standard 754 admits the following format:



**Definition 2.19** (Normalised Number).

A **normalised number**,  $v$ , represented in IEEE 754 is

$$v = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exponent} - \text{bias}}$$

Note: for normalised numbers, the integer part of the 8 bit fraction part is 1.

- Sign bit is 1 bit, followed by exponent and lastly mantissa
- Exponent must NOT be 0. It must be in  $[1, 2^e - 2]$ , where  $e$  is the number of exponent bits.  
All zero 0 or all one  $2^e - 1$  exponents are reserved for special values and are not used for normalised numbers
- Suppose the exponent bias is  $b$ , then the exponent is in excess- $b$  representation of the true power.
- A normalised fraction part is in the interval  $[1, 2)$ .

**Definition 2.20** (Denormalised numbers).

**Denormalised numbers** are to represent really small (positive or negative) numbers as following:

$$v = (-1)^{\text{sign}} \times 0.\text{fraction} \times 2^{-\text{bias}+1}$$

To identify a number as **denormalised**, exponent must be 0 and mantissa must be non-zero. The system will interpret the exponent to be  $1 - \text{bias}$  instead of  $0_{\text{excess-bias}}$ .

**Definition 2.21** (Special values).

- 0:        exponent = 0        fraction = 0
- $+\infty$ :    exponent =  $2^e - 1$     fraction = 0
- $-\infty$ :    exponent =  $2^e - 1$     fraction = 0
- NaN:     exponent =  $2^e - 1$     fraction  $\neq 0$

**Definition 2.22** (Comparison Rules).

Type	Sign	Exponent	Fraction
$+\infty$	0	$\underbrace{11 \dots 1}_e$	$\underbrace{0 \dots 0}_f$
$-\infty$	1	$\underbrace{11 \dots 1}_e$	$\underbrace{0 \dots 0}_f$
NaN		$\underbrace{11 \dots 1}_e$	non zero
0		$\underbrace{00 \dots 0}_e$	$\underbrace{00 \dots 0}_f$
Denormalised Numbers		$\underbrace{0 \dots 0}_e$	non zero
Normalised Numbers		$[\underbrace{00 \dots 0}_{e-1} 1, \underbrace{11 \dots 1}_{e-1} 0]$	$[\underbrace{00 \dots 0}_f, \underbrace{11 \dots 1}_f]$

- Negative and positive zero compare *equal*
- Every NaN compares *unequal* to every value, including itself
- All values except NaN are strictly smaller than  $+\infty$  and strictly larger than  $-\infty$ .

**Definition 2.23** (Overflow, Underflow). *In floating point representation, greater than the largest representable positive number or smaller than the smallest representable negative number results in **overflow**; greater than the largest representable negative number and smaller than the smallest representable positive number results in **underflow**.*

**Definition 2.24** (Rounding).

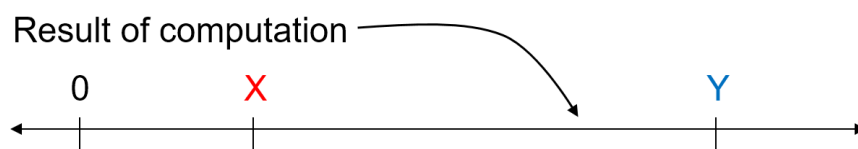
**Rounding** is defined as selecting a representable number as the result, from the two closest representable numbers.

Rounding destroys associativity of all operations on floating point numbers.

**Definition 2.25** (Guard Bit, Round Bit, Sticky Bit).

To enable rounding, IEEE 754 specifies that all arithmetic must be performed with 3 extra bits at the end of the last fraction bit, from the order of the most significant bit to the least:

- Guard bit
- Round bit
- Sticky bit, which equals to 1 if any bits to the right of it is 1



**Theorem 2.10** (Rounding Modes).

There are four rounding modes:

- Round to nearest (default)

```

if (GUARD == 1) {
    if ((ROUND == 1) or (STICKY == 1)) {
        return Y as the mantissa answer
    } else { // Must be (ROUND == 0) and (STICKY == 0)
        // Invoke IEEE tie breaker
        if (LSB, i.e., bit 23 is 1) {
            return Y as the mantissa answer
        } else {
            return X as the mantissa answer
        }
    }
} else {
    return X as the mantissa answer
}

```

- Round towards 0

Always report X as the mantissa answer.  
(if less than 0, then this becomes X)

- Round towards  $+\infty$

```

if (result > 0) {
    report Y as the mantissa answer.
} else {
    report X as the mantissa answer.
}

```

- Round towards  $-\infty$

```

if (result < 0) {
    report Y as the mantissa answer.
} else {
    report X as the mantissa answer.
}

```

**Definition 2.26** (Error).

Rounding yields a representable floating point number  $x'$  that is an approximation of the real number  $x$ .

Define absolute error =  $|x' - x|$ .

Define relative error =  $\frac{|x' - x|}{x}$  (assuming  $x \neq 0$ )

**Definition 2.27** (Machine Epsilon  $\varepsilon$ ).

Informally, **machine epsilon**  $\varepsilon$  is defined as 1 *added to the LSB*.

**Definition 2.28** (Unit in the Last Place(ulp)).

Given an IEEE floating point number  $x$ , say with an exponent  $E$ . The **unit in the last place** of  $x$  is defined as

$$\text{ulp}(x) = \varepsilon \times 2^E$$

**Round to nearest** results in an absolute error that is less than  $\frac{1}{2}\text{ulp}(x)$ .

**Theorem 2.11** (Floating Point Addition).

Given two decimal numbers in floating point notation:

- $X = 0.a_1a_2 \cdots a_n \times 2^p$
- $Y = 0.b_1b_2 \cdots b_n \times 2^q$

To perform  $X + Y$ ,

1. align the decimal point by shifts such that two exponents are the same.
2. If  $p > q$ , then we need to adjust  $Y$  such that  $Y' = 0.\underbrace{00 \cdots 0}_{p-q}b_1b_2 \cdots b_n \times 2^p$ .

This is called **denormalisation shift**.

3. Addition is performed on fraction part of  $X$  and  $Y$ .
4. Normalise the result
5. Round the result

## 3 Boolean Algebra

**Definition 3.1** (Digital Circuit).

**Digital circuit** is circuit with two voltage levels, known as

- High, true, 1, asserted
- Low, false, 0, deasserted

Advantages of digital circuits over analog circuits include:

- More reliable (simpler circuits, less noise-prone)
- Specified accuracy (determinable)
- Abstraction can be applied using simple mathematical model – Boolean Algebra
- Ease design, analysis and simplification of digital circuit – Digital Logic Design

**Definition 3.2** (Type of Logic Blocks).

There are two types of logic blocks, known as

1. **Combinatorial:** *no* memory, output depends *solely* on the input
  - Gates
  - Decoders, multiplexers
  - Adders, multipliers
2. **Sequential:** *with* memory, output depends on *both* input and *current states*
  - Counters, registers
  - Memories

### 3.1 Boolean Algebra

Boolean algebra involves **boolean values** and **connectives**.

**Definition 3.3** (Boolean Values).

There are *two* **boolean values** in boolean algebra:

- True (1)
- False (0)

**Definition 3.4** (Connectives).

There are *three* **connectives** in boolean algebra, which maps given input boolean value(s) to a single output boolean value.

**Truth tables** defines a connective by providing a listing of every possible combination of inputs and its corresponding outputs.

**Reminder:** Inputs must list in *ascending* **binary sequence**.

The three connectives are:

- Conjunction (AND):  $A \cdot B$

$A$	$B$	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

- Disjunction (OR):  $A + B$

$A$	$B$	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

- Negation (NOT):  $A'$

$A$	$A'$
0	1
1	0

**Theorem 3.1** (Laws of Boolean Algebra).

- Identity laws

$$A + 0 = 0 + A = A$$

$$A \cdot 1 = 1 \cdot A = A$$

- Inverse/Complement laws

$$A + A' = 1$$

$$A \cdot A' = 0$$

- Commutative laws

$$A + B = B + A$$

$$A \cdot B = B \cdot A$$

- Associative laws

$$A + (B + C) = (A + B) + C$$

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$



- Distributive laws

$$A \cdot (B + C) = A \cdot B + A \cdot C$$

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

**Theorem 3.2** (Precedence of Connectives).

The precedence from highest to lowest is

- NOT
- AND
- OR

Parenthesis can be used to overwrite precedence.

**Theorem 3.3** (Duality).

If the AND/OR operators and identity elements 0/1 in a Boolean equation are interchanged, it remains valid.

**Theorem 3.4** (Basic Theorems).

1. Idempotency

$$X + X = X$$

$$X \cdot X = X$$

2. Zero and One Elements

$$X + 1 = 1$$

$$X \cdot 0 = 0$$

3. Involution

$$(X')' = X$$

4. Absorption

$$X + X \cdot Y = X$$

$$X \cdot (X + Y) = X$$

5. Absorption (variant)

$$X + X' \cdot Y = X + Y$$

$$X \cdot (X' + Y) = X \cdot Y$$

## 6. DeMorgan's

$$\begin{aligned}(X + Y)' &= X' \cdot Y' \\ (X \cdot Y)' &= X' + Y'\end{aligned}$$

Demorgan's Theorem can be generalised to more than two variables.

## 7. Consensus

$$\begin{aligned}X \cdot Y + X' \cdot Z + Y \cdot Z &= X \cdot Y + X'Z \\ (X + Y) \cdot (X' + Z) \cdot (Y + Z) &= (X + Y) \cdot (X' + Z)\end{aligned}$$

**Definition 3.5** (Boolean Functions).

**Boolean functions** are functions which takes in **boolean variable** and outputs an expression of these boolean variable.

**Definition 3.6** (Complement of a Function).

Given a Boolean function  $F$ , the **complement** of  $F$ , denoted as  $F'$ , is obtained by interchanging 1 with 0 in the function's output values.

## 3.2 Standard Forms

There are two standard forms:

- Sum-of-Products
- Product-of-Sums

**Definition 3.7** (Literals).

A **literal** is a Boolean variable on its own or in its complemented form.

**Definition 3.8** (Product Term).

A **product term** is a single literal or a logical product(AND) of several literals.

**Definition 3.9** (Sum Term).

A **sum term** is a single literal or a logical sum(OR) of several literals.

**Definition 3.10** (Sum-of-product(SOP) expression).

**Sum-of-Products expression** is a product term or a logical sum(OR) of several product terms.

**Definition 3.11** (Product-of-Sums(POS) expression).

**Product of Sum expression** is a sum term or a logical product(AND) of several sum terms.

**Theorem 3.5.** Every Boolean expression can be expressed in SOP or POS.

**Definition 3.12** (Minterm).

A **minterm** of  $n$  variables is a **product term** that contains  $n$  literals from *all* the variables.

**Definition 3.13** (Maxterm).

A **maxterm** of  $n$  variables is a **sum term** that contains  $n$  literals from *all* the variables.

In general, with  $n$  variables, we have  $2^n$  minterms and  $2^n$  maxterms.

**Definition 3.14** (Ordering of Minterms).

Suppose there are  $n$  ordered variable  $(x_1, x_2, \dots, x_n)$ . Minterms are numbered by a binary encoding of the **complementation pattern** of the ordered variables. The convention assigns the value 1 to the direct form  $x_i$  and 0 to its complemented form  $x'_i$ . The index of the minterm  $x_1 \cdot x_2 \cdots x_n$  is then  $(v_1 v_2 \cdots v_n)_2$  where  $v_i$  is the value of variable  $x_i$ .

**Definition 3.15** (Indexing of Maxterms).

Suppose there are  $n$  ordered variable  $(x_1, x_2, \dots, x_n)$ . Maxterms are numbered by a binary encoding of the **complementation pattern** of the ordered variables. The convention assigns the value 0 to the direct form  $x_i$  and 1 to its complemented form  $x'_i$ . The index of the maxterm  $x_1 + x_2 + \cdots + x_n$  is then  $(v_1 v_2 \cdots v_n)_2$  where  $v_i$  is the value of variable  $x_i$ .

**Theorem 3.6.** Each minterm is the complement of the maxterm of the same index.

$$m'_i = M_i$$

**Definition 3.16** (Canonical Forms).

Canonical form refers to a unique form of representation. It can be shown that

- Sum-of-minterms is the canonical sum-of-product
- Product-of-maxterms is the canonical product-of-sum

**Theorem 3.7** (Defining Function from Sum-of-minterms).

A function  $F$  can be defined by the sum of minterms  $m_i$  for which  $F(m_i) = 1$ .

**Theorem 3.8** (Defining Function from Product-of-Maxterms).

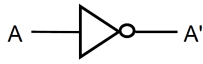
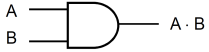
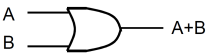
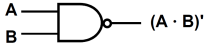
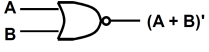
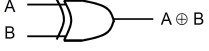
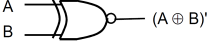
A function  $F$  can be defined by the product of maxterms  $M_i$  for which  $F(M_i) = 0$ .

**Theorem 3.9** (Complementation of Function).

Complementation of functions can be easily done by complementation between sum-of-minterms and product-of-maxterms.

$$\begin{aligned} \left( \sum_{i \in I} m(i) \right)' &= \prod_{i \in I} M(i) \\ \left( \prod_{i \in I} M(i) \right)' &= \sum_{i \in I} m(i) \end{aligned}$$

## 4 Logic Gates and Circuits

Name	Symbol	Truth Table		
NOT Gate		$A$	$A'$	
		0	1	
		1	0	
AND Gate		$A$	$B$	$A \cdot B$
		0	0	0
		0	1	0
		1	0	0
		1	1	1
OR Gate		$A$	$B$	$A + B$
		0	0	0
		0	1	1
		1	0	1
		1	1	1
NAND Gate		$A$	$B$	$(A \cdot B)'$
		0	0	1
		0	1	1
		1	0	1
		1	1	0
NOR Gate		$A$	$B$	$(A + B)'$
		0	0	1
		0	1	0
		1	0	0
		1	1	0
XOR Gate		$A$	$B$	$A \oplus B$
		0	0	0
		0	1	1
		1	0	1
		1	1	0
XNOR Gate		$A$	$B$	$(A \oplus B)'$
		0	0	1
		0	1	0
		1	0	0
		1	1	1

## 4.1 Logic Circuit

**Definition 4.1** (Fan-in).

**Fan-in** refers to the number of inputs of a gate.

Given a Boolean expression, we may implement it as a **logic circuit**.

## 4.2 Universal Gates

{AND, OR, NOT} gates are sufficient for building any Boolean function. Thus the set {AND, OR, NOT} is called a *complete* set of logic.

However, other gates are also used for

- Usefulness
- Economical
- Self-sufficient

Furthermore, {NAND} gate is a complete set of logic; {NOR} gate is also a complete set of logic by duality.

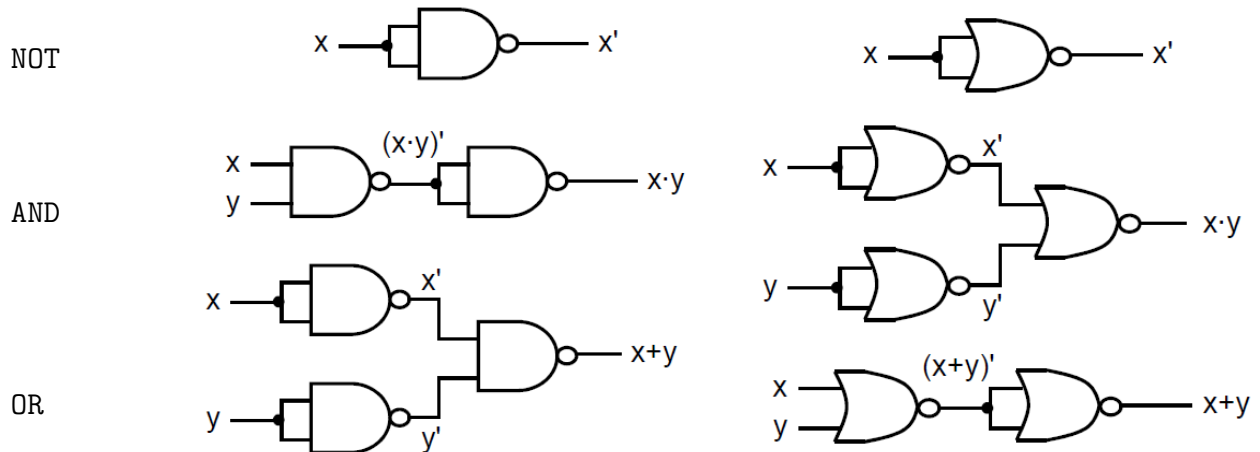


Figure 4: Implementation of OR, AND, OR using NAND and NOR respectively

### 4.2.1 SOP and NAND Circuits

An SOP expression can be easily implemented using

- 2-level AND-OR circuit
- 2-level NAND circuit

A 2-level AND-OR circuit can be converted to a 2-level NAND circuit by

1. Introduce 2 NOT gate after first level AND and before second level OR gates.
2. The first level AND have been converted to NAND gates; the second level negative-OR gate is *equivalent* to NAND gate.

### 4.2.2 POS and NOR Circuits

A POS expression can be easily implemented using

- 2-level OR-AND circuit
- 2-level NOR circuit

A 2-level OR-AND circuit can be converted to a 2-level NOR circuit by

1. Introduce 2 NOT gate after first level OR and before second level AND gates.
2. The first level OR have been converted to NOR gates; the second level negative-AND gate is *equivalent* to NOR gate.

## 5 Kaunaugh Map

**Function simplification** leads to simpler expressions which uses fewer logic gates and makes circuits cheaper, less power consuming and faster.

There are three techniques in function simplification: Boolean Algebra, Karnaugh Maps and Quine-McCluskey.

### 5.1 Boolean Algebra

Algebraic simplification aims to minimise

- Number of literals, and
- Number of terms

### 5.2 Half Adder

**Definition 5.1** (Half Adder).

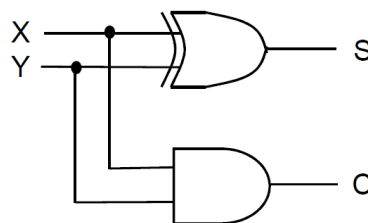
**Half adder** is a circuit that adds 2 single bits ( $X, Y$ ) to produce a result of 2 bits ( $C, S$ ).<sup>3</sup> The truth table for half adder is

$X$	$Y$	$C$	$S$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

In canonical form (sum-of-minterms):

- $C = X \cdot Y$
- $S = X \cdot Y' + X' \cdot Y$ <sup>4</sup>

The half adder can be implemented as



---

<sup>3</sup> $C$  is known as the carry bit, where  $S$  is the sum bit.

<sup>4</sup>In fact,  $S = X \oplus Y$ .

### 5.3 Karnaugh Maps

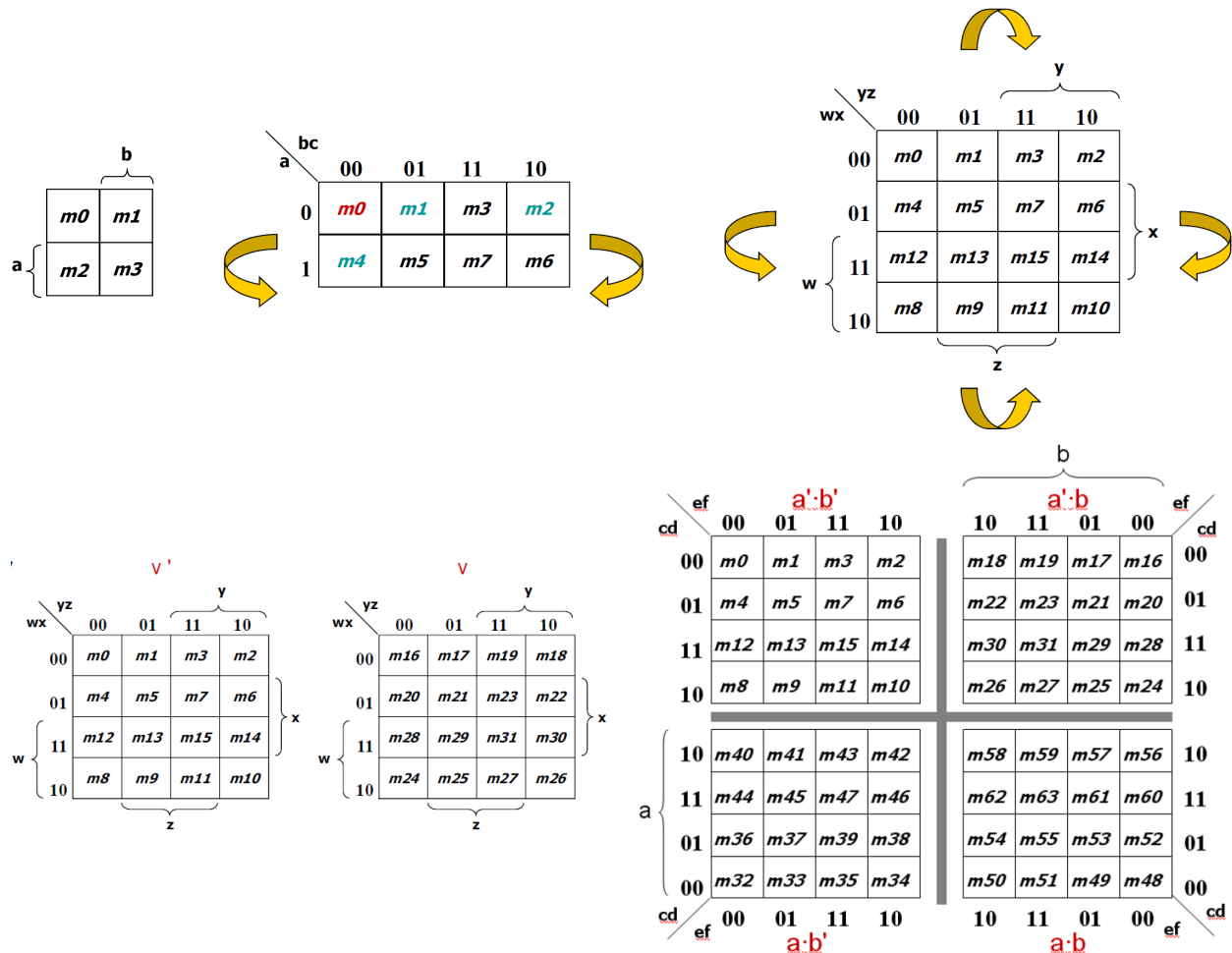
**Karnaugh Maps** is a systematic method to obtain simplified (minimal) sum-of-products(SOP) expressions. Its objective is to obtain *fewest* product terms and literals.

**Definition 5.2** (Karnaugh Map).

**Karnaugh Map** is an abstract form of Venn diagram, organised as a matrix of squares, where

- Each square represents a minterm
- Two adjacent squares represent minterms that differ by *exactly one* literal

Layouts of Karnaugh Maps from 2 variables to 6 variables are as following: Based on the



unifying theorem

$$A + A' = 1$$

In a K-map, each cell containing 1 corresponds to a minterm of a given function  $F$ .

Each **valid grouping** of *adjacent cells* containing 1 then corresponds to a **simpler product term** of  $F$ .



**Definition 5.3** (Valid Grouping).

- A valid grouping admits a rectangular shape.
- A valid grouping must have size in **powers of two**:  $1, 2, 4, 8, \dots$
- Grouping  $2^n$  adjacent cells eliminates  $n$  variables.

In simplification,

1. Group as many cells as possible, by considering **prime implicants**.
2. Select as few groups as possible to cover all the cells(minterms) of the function, by considering **essential prime implicants**.

If a function is not in sum-of-minterms form,

- Convert it into sum-of-products form
- Expand the SOP expression into sum-of-minterms expression.

**Definition 5.4** (Implicant).

**Implicant** is a product term that could be used to cover minterms of the function.

**Definition 5.5** (Prime Implicant).

**Prime implicant** is a product term obtained by combining the *maximum* possible number of minterms from adjacent squares in the map.

**Definition 5.6** (Essential Prime Implicant).

**Essential Prime Implicant** is a prime implicant that includes at least one minterm that is not covered by any other prime implicant.

**Theorem 5.1** (Algorithm for minimal SOP Expression).

- Circle all prime implicants on the K-map.
- Identify and select all essential prime implicants for the cover.
- Select a minimum subset of the remaining prime implicants to complete the cover.

**Theorem 5.2** (Algorithm for simplified POS Expression).

- Group maxterms of  $F$ , equivalently minterms of  $F'$ , identified as 0 entry in K-map of  $F$ . This gives the SOP of  $F'$ .
- The simplified POS expression of  $F$ , use DeMorgan's law.

**Definition 5.7** (Don't care conditions).

Outputs that can be either 1 or 0 are called **don't care conditions**, denoted by  $X$ .

The set of don't care minterms are denoted as  $\sum d$ .

Don't care conditions can be used to help simplify Boolean expression further in K-maps.

## 6 Combinatorial Circuits

In combinational circuit, each output depends entirely on the immediate(present) input.

### 6.1 Gate Level Design

**Theorem 6.1** (Gate Level Design Procedure).    1. *State problems*

2. *Determine and label the inputs and outputs of circuit*

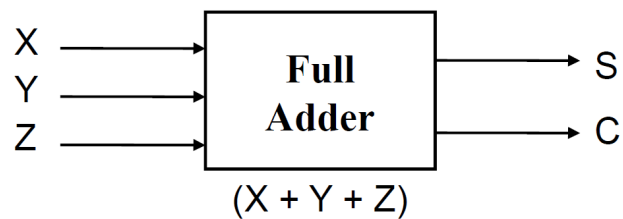
3. *Draw the truth table*

4. *Obtain simplified Boolean functions.*

5. *Draw logic diagram.*

#### 6.1.1 Full Adder

**Full adder** adds three bits  $X, Y, Z$ , which includes the carry, and output a sum bit  $S$  and carry bit  $C$ . Truth table: Simplified formulae:

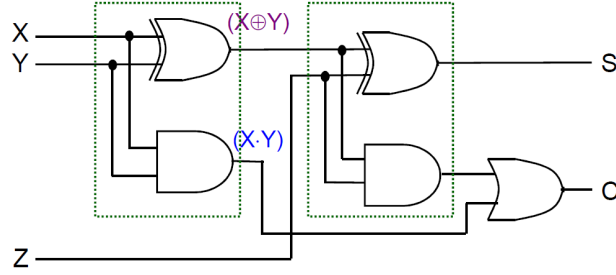


$X$	$Y$	$Z$	$C$	$S$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$C = X \cdot Y + (X \oplus Y) \cdot Z$$

$$S = X \oplus (Y \oplus Z)$$

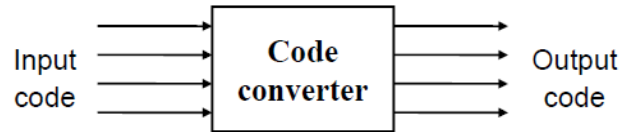
Full Adder can be made from half adders.



## 6.2 Code Converters

**Definition 6.1** (Code Converters).

**Code converter** takes an input code and translates to its equivalent output code.



**Definition 6.2** (Binary Code Decimal).

Binary code decimal is a representation system for coding a number in which each digit of a decimal number is represented individually by its binary equivalent.

Decimal digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

**Definition 6.3** ( $f_{\text{BCD}}$ ).

Let  $(a_0a_1 \dots a_{n-1})_{10}$  be a decimal number. Its Binary Code Decimal is given by

$$f_{\text{BCD}}(a_0a_1 \dots a_{n-1}) = s_{0,1}s_{0,2}s_{0,3}s_{0,4} \dots s_{n-1,4}$$

where  $s_{i,1}s_{i,2}s_{i,3}s_{i,4}$  is the BCD of decimal  $a_i$  defined from the truth table.

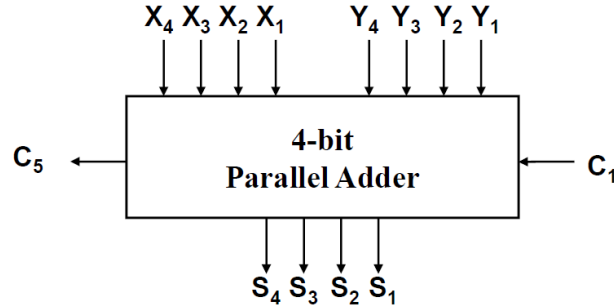
As a result, the length of binary code decimal is always in multiple of 4.

## 6.3 Block Level Design

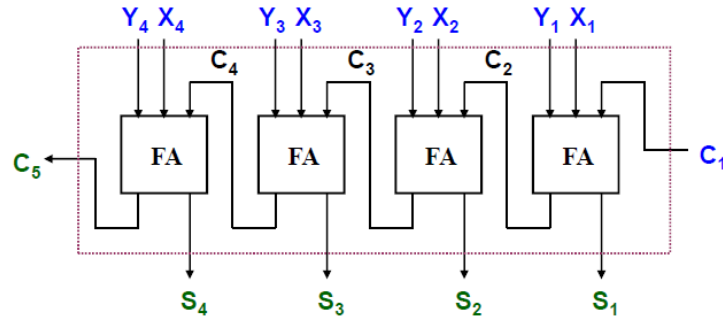
Block level design method relies on algorithms or formulae of the circuit, which are obtained by decomposing the main problem to sub-problems recursively.

### 6.3.1 4-bit adder

Consider a circuit to add two 4-bit unsigned numbers together and a carry-in, to produce a 5-bit result. With the idea that  $C_{i+1}S_i = X_i + Y_i + C_i$ , which is the same function of full



adder, 4-bit adder is implemented by cascading 4 full adders via their carries. The above is



called **parallel adder**, as inputs are presented in parallel.

### 6.3.2 BCD-to-Excess-3 Converter

Excess-3 code can be converted from BCD code using truth table. Therefore, gate-level design can be used since there are only 4 inputs.

However, alternative design is also possible, by identifying

$$\text{Excess-3 code} = \text{BCD Code} + 0011_2$$

	BCD				Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0
10	1	0	1	0	X	X	X	X
11	1	0	1	1	X	X	X	X
12	1	1	0	0	X	X	X	X
13	1	1	0	1	X	X	X	X
14	1	1	1	0	X	X	X	X
15	1	1	1	1	X	X	X	X

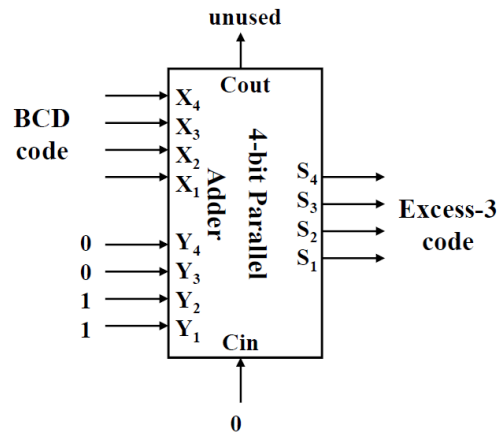
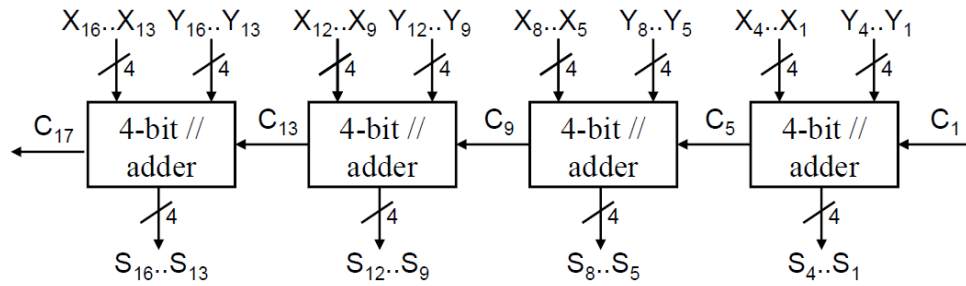


Figure 5: BCD-to-Excess-3 Code Converter

### 6.3.3 16-bit Parallel Adder

Larger parallel adders can be built from smaller ones.

A **16-bit parallel adder** can be constructed from four 4-bit parallel adders:

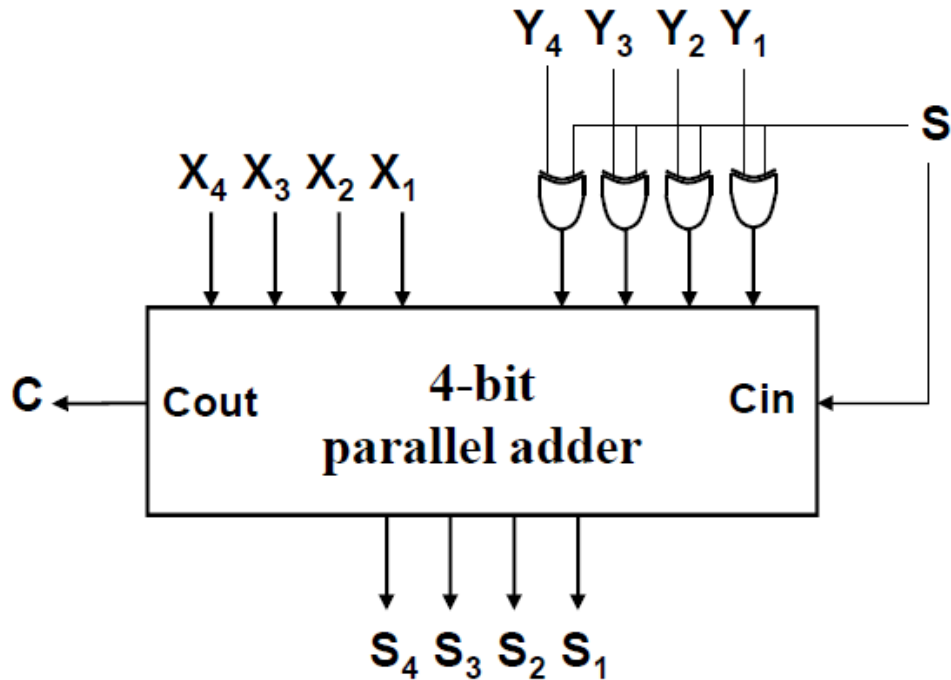


#### 6.3.4 4-bit Adder cum Subtractor

**4-bit Adder cum Subtractor** is a circuit that can perform both addition and subtraction, using a parallel adder with a control signal. Recall

$$\begin{aligned}
 X - Y &= X + (-Y) \\
 &= X + 2\text{s complement of } Y \\
 &= X + 1\text{s complement of } Y + 1
 \end{aligned}$$

Therefore, XOR gates are used to flip bits<sup>5</sup> and control signal  $S$  is connected to input carry-in.



<sup>5</sup>Note  $x \text{ XOR } 0 = x$ , and  $x \text{ XOR } 1 = x'$

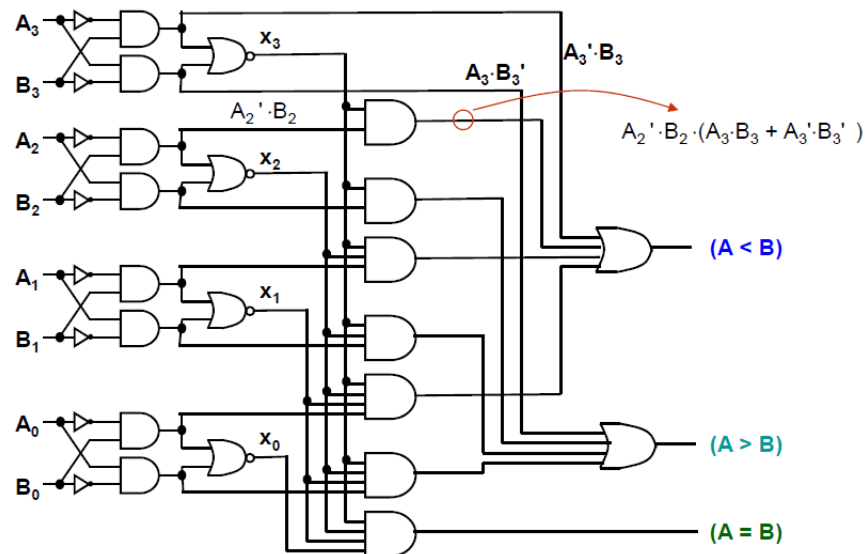
When  $S = 1$ , it subtracts by adding  $X$  with  $Y'$  and  $S = 1$ , and when  $S = 0$ , it adds by adding  $X$  with  $Y$  with  $S = 0$ .

### 6.3.5 Magnitude Comparator

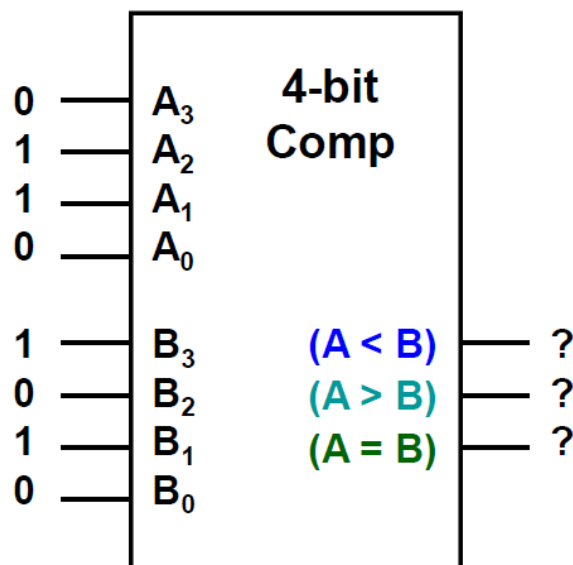
**Magnitude comparator** compares 2 values  $A$  and  $B$ , to output either  $A > B$ ,  $A = B$  or  $A < B$ .

The key idea is that  $X \cdot Y'$  outputs 1 when  $X > Y$  and 0 otherwise. Therefore,  $X = Y$  if and only if  $(X \cdot Y') \text{ NOR } (X' \cdot Y) = X \cdot Y + X' \cdot Y' = 1$ .

We first build a 4-bit magnitude comparator using the above logic. Let  $A = A_3A_2A_1A_0$ ,  $B = B_3B_2B_1B_0$ . Denote  $x_i = A_i \cdot B_i + A'_i \cdot B'_i$ . This generates the block diagram of 4-bit



magnitude comparator



## 6.4 Circuit Delays

**Definition 6.4** (Circuit Delay).

Given a logic gate with delay  $t$ . If inputs are stable at times  $t_1, \dots, t_n$ , then the earliest time in which the output will be stable is

$$\max(t_1, \dots, t_n) + t$$

Suppose a full adder has delay  $t_1, t_2$  for  $X, Y$  and  $t_3$  for carry in,  $S$  will have delay

$$S_{\text{delay}} = \max\{\max\{t_1 + t_2\} + t, t_3\} + t$$

$C$  will have delay

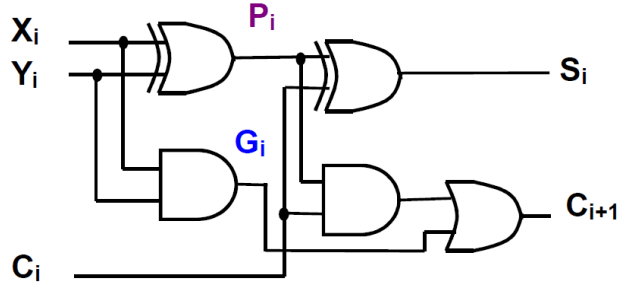
$$C_{\text{delay}} = \max\{\max\{t_1, t_2\} + t, t_3\} + 2t$$

According to the above, a  $n$ -bit ripple-carry parallel adder will experience the following delay<sup>6</sup>

$$\begin{aligned} S_n &= 2nt \\ C_{n+1} &= (2n + 1)t \end{aligned}$$

Therefore, propagation delay of ripple-carry parallel adders is proportional to the number of bits it handles.

### 6.4.1 Carry Look-ahead Adder



Consider the full adder, define intermediate signals  $P_i, G_i$  as follows

$$\begin{aligned} P_i &= X_i \oplus Y_i \\ G_i &= X_i \cdot Y_i \end{aligned}$$

Therefore, the output  $S_i, C_{i+1}$  can be given in terms of  $C_i, P_i, G_i$ :

$$\begin{aligned} S_i &= P_i \oplus C_i \\ C_{i+1} &= G_i + P_i \cdot C_i \quad (\#) \end{aligned}$$

---

<sup>6</sup> $n$  is of index 1.



We can regard,  $G_i$  as the **carry generate** signal, since  $G_i = 1$  suggests both  $X_i$  and  $Y_i$  is 1, which definitely *generates* a carry  $C_{i+1} = 1$ .

Also,  $P_i$  can be regarded as the **carry propagate** signal, as  $P_i = 1$  suggests exactly  $X_i = 1$  or  $Y_i = 1$  but not both. Therefore,  $C_{i+1} = 1$  if  $C_i = 1$  and  $P_i = 1$ , which suggests that the status of carry in  $C_i$  is *propagated* to carry out  $C_{i+1}$ .

For the 4-bit ripple carry adder, the equation for  $C_{i+k}, k = 1, \dots, 4$  is only dependent on  $G_j, P_j, C_j, 1 \leq j < i + k$ , according to the recursively relation (#). By expanding the recursive relation into an iterative expression we have

$$C_{i+k} = \prod_{j=0}^{k-1} P_j \cdot C_i + \sum_{j=0}^{k-1} G_{i+j} \prod_{l=j+1}^{k-1} P_{i+l}$$

which is a two level sum of product expressions in terms of  $G, P, C$ .

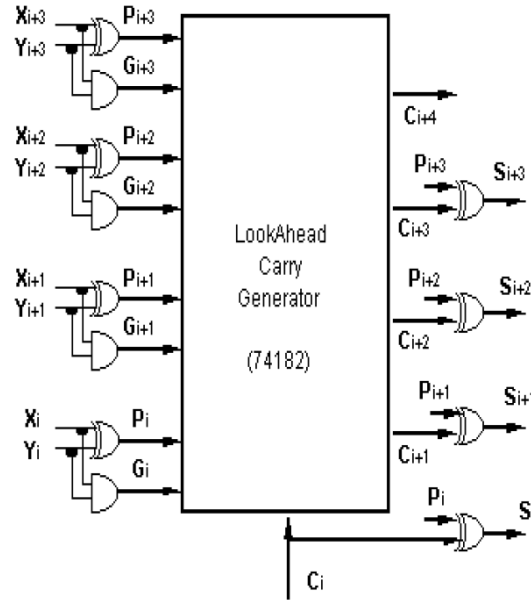


Figure 6:  $X_i, Y_i$  are preprocessed outside the block. Block inputs  $P, G$  only and outputs  $C$  only.

The generation of  $P, G$  of each bit takes time  $t$  from XOR and AND gate; generation of each carry  $C_{i+k}$  takes time  $2t$  from the sum of product expression; generation of sum signals  $S_{i+k}$  of each bit takes time  $t$  from  $P_{i+k}, C_{i+k-1}$ . Therefore, the whole process takes time  $4t$ .

Larger block carry look-ahead adder can be built from 4-bit carry look-ahead adder. Two *additional* output is needed: **block carry generate** and **block carry propagate**.

Let  $P_0, P_1, P_2, P_3$  be the 4 carry propagate bits of the 4-bit carry look-ahead adder. Let  $G_0, G_1, G_2, G_3$  be the 4 carry generate bits of the 4-bit carry look-ahead adder. Then the **block** carry propagate and generate bits,  $P^*$  and  $G^*$ , respectively are defined as

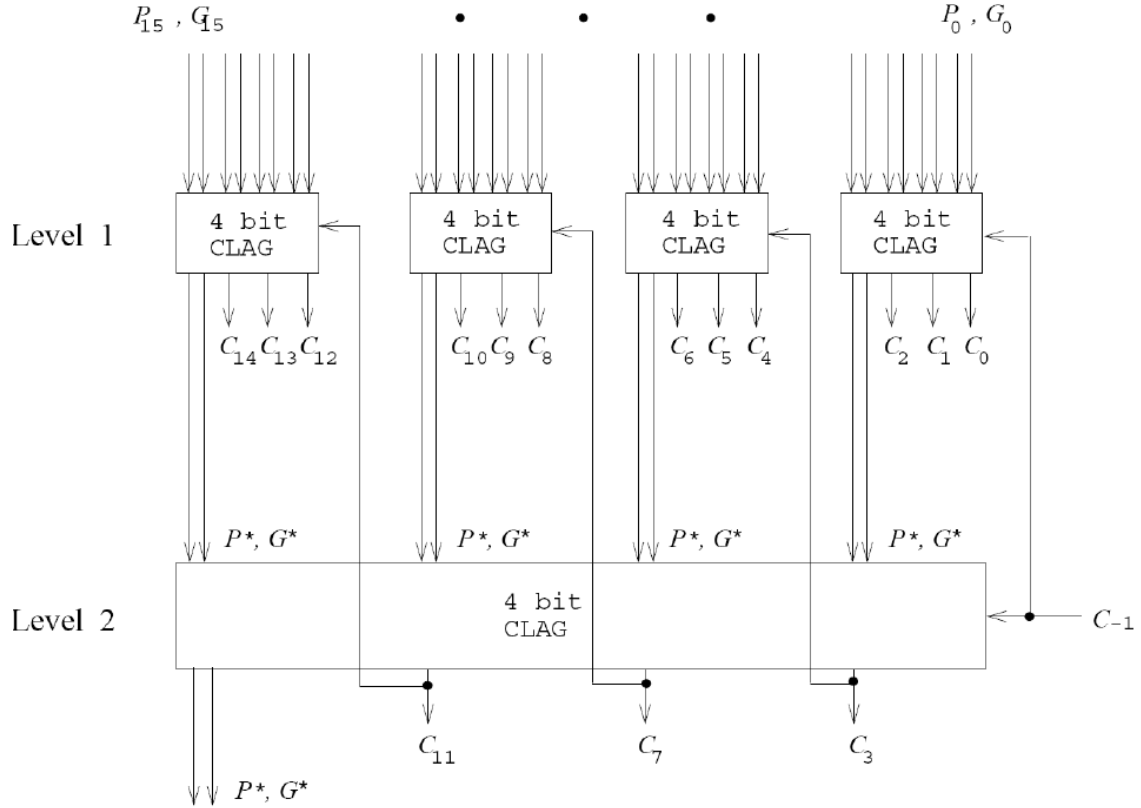
$$P^* = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$

$$G^* = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3$$

It is easy to see that the carry out bit of block 4-bit carry look-ahead adder is

$$C_3 = G^* + P^* \cdot C_{-1}$$

where  $C_{-1}$  is the carry in to the 4-bit block. The sequence of availability of output is that



- Time 0:  $P_0 \sim P_{15}, G_0 \sim G_{15}, C_{-1}$ .
- Time 1:  $C_0 \sim C_2$ , all  $P^*, G^*$  between level 1 and level 2.
- Time 2:  $C_4, C_7, C_{11}, P^*, G^*$  after level 2
- Time 3: Rest  $C_4 \sim C_{14}$ .<sup>7</sup>

<sup>7</sup> $C_4$  is dependent on  $C_3$ , which is dependent on  $P^*$  and  $G^*$ , which is dependent on  $P, G$ .

## 7 More Building Blocks

### 7.1 Decoder

**Definition 7.1** (Decoder).

A **decoder** converts binary information from  $n$  input lines to  $2^n$  output lines.

#### 7.1.1 Truth table

The truth table for  $2^n$  output, when input is enumerated in increasing sequence, is diagonal. The column of output is arranged according to the increasing order of minterm of the function.

**Theorem 7.1** (Building functions using decoder).

Any boolean function with  $n$  input with  $m$  output can be built using a  $n : 2^n$  decoder, which generates the minterms, and  $m$  OR gates to form the sum.

Decoders often come with an **enable control** signal, so that the device is only activated when the enable  $E = 1$ .

In most MSI decoders, enable signal is zero-enable, usually denoted by  $E'$ . The decoder is

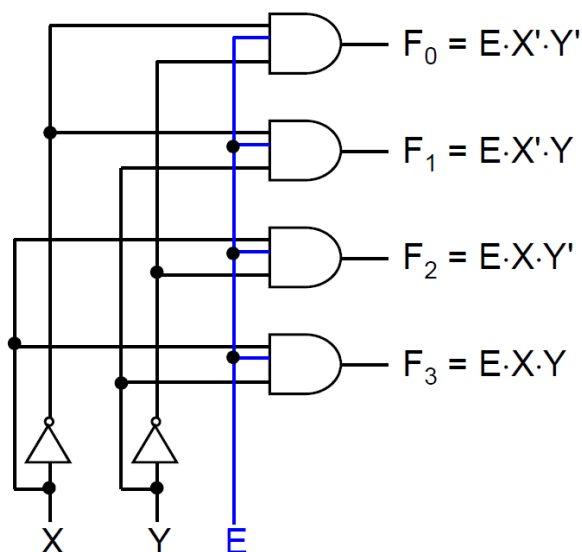


Figure 7: Implementation of 2:4 Encoder with **one-enable** control  $E = 1$

enabled when signal  $E$  is low.

#### 7.1.2 Larger Decoder

Larger decoders can be constructed, with one inverter from smaller ones by treating  $E$  as the most significant bit which selects the smaller decoders.

### 7.1.3 Implementing functions

We may implement the functions using a decoder in several ways.

Suppose a function is specified as  $f(A, B, C) = \sum m(0, 1, 4, 6, 7) = \prod M(2, 3, 5)$ , we may implement it

- using a decoder with active high outputs<sup>8</sup> with a OR gate on minterms:

$$f = m_0 + m_1 + m_4 + m_6 + m_7$$

- using a decoder with active low outputs<sup>9</sup> with a NAND gate on minterms:

$$f = (m'_1 \cdot m'_2 \cdot m'_4 \cdot m'_6 \cdot m'_7)'$$

- Using a decoder with active high outputs with a NOR gate on maxterms:

$$f = (m_2 + m_3 + m_5)'$$

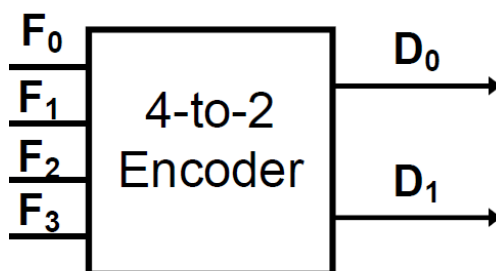
- Using a decoder with active low outputs with a AND gate on maxterms:

$$f = m'_2 \cdot m'_3 \cdot m'_5$$

## 7.2 Encoders

**Definition 7.2** (Encoder).

Given  $2^n$  input lines, of which *exactly* 1 is high, the **encoder** provides a  $n$  bit code that corresponds to that input line.



### 7.2.1 Truth Table

For the truth table of an encoder, when exactly 1 out of  $2^n$  inputs is high, say  $F_i$ , the output  $D_n D_{n-1} \cdots D_1 D_0$  is the binary string  $i_2$ ; if more than 1 input are high, the output becomes don't care.

---

<sup>8</sup>Given any input, only one of the output will be 1 and rest 0

<sup>9</sup>Given any input, only one of the output will be 0 and rest 1

$F_0$	$F_1$	$F_2$	$F_3$	$D_1$	$D_0$
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1
0	0	0	0	X	X
0	0	1	1	X	X
0	1	0	1	X	X
0	1	1	0	X	X
0	1	1	1	X	X
1	0	0	1	X	X
1	0	1	0	X	X
1	0	1	1	X	X
1	1	0	0	X	X
1	1	0	1	X	X
1	1	1	0	X	X
1	1	1	1	X	X

Figure 8:  $D_0 = F_1 + F_3$ ,  $D_1 = F_2 + F_3$

The implementation of a specified output is the sum of inputs whose specified output are high, given the benefits of don't cares.

Therefore, encoders can be designed using OR gate.

### 7.2.2 Priority Encoder

In priority encoder, each of the inputs is assigned a **priority**.

The **most** significant bit of the input has the **highest** priority while the least significant bit has the lowest priority.

If two input lines goes high, only the *higher* priority one will be considered as high. This generates a truth table with don't cares in inputs.

$F_3$	$F_2$	$F_1$	$F_0$	$D_1$	$D_0$
1	X	X	X	1	1
0	1	X	X	1	0
0	0	1	X	0	1
0	0	0	X	0	0

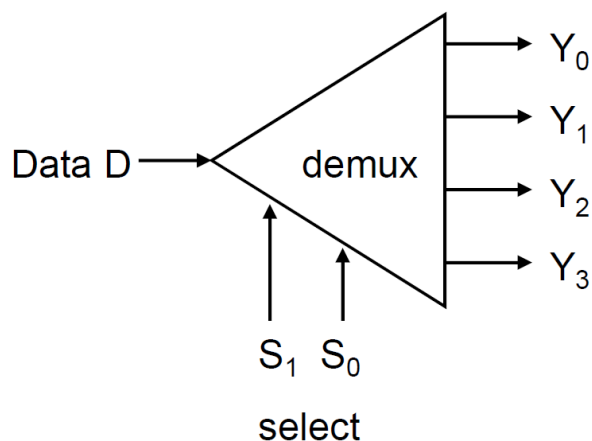
Figure 9: Truth table of priority encoder

## 7.3 Demultiplexers

**Definition 7.3** (Demultiplexers).

Given an input line and a set of  $n$  selection lines, a **demultiplexer** directs data from the input to *one* selected output line out of  $2^n$ .

Suppose the selection lines admits a  $N = (S_{n-1} \dots S_0)_2$  binary number, the output line  $Y_N$  will correspondingly be selected such that  $Y_N = D$ , the input.



### 7.3.1 Truth Table

The truth table for outputs from demultiplexers of  $n$  selection lines, when the selection lines is enumerated in increasing sequence, is diagonal  $D$ , where  $D$  is the input.

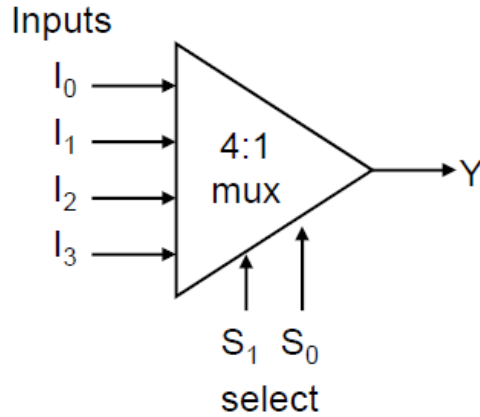
There is similarity of truth table between demultiplexers and decoders. In fact, a demultiplexer of  $n$  selection line can be implemented using a  $n : 2^n$  decoder with selection lines connected to the input of decoders and data input connected to the enable bit.

$S_1$	$S_0$	$Y_0$	$Y_1$	$Y_2$	$Y_3$
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

## 7.4 Multiplexers

**Definition 7.4** (Multiplexers).

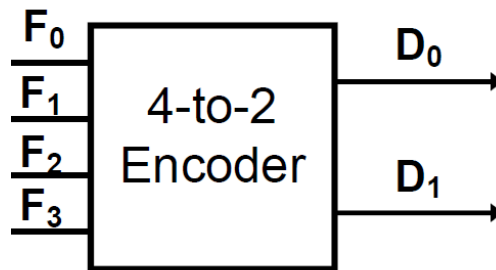
A **multiplexer** is a device with has  $2^n$  input lines,  $n$  selection lines and 1 output line. It steers one of  $2^n$  inputs to a single output line.



#### 7.4.1 Truth Table

The output  $Y$  equals to  $I_i$ , the  $i$ th input, where binary representation of  $i$  equals to the binary string given by selection lines  $S_{n-1} \dots S_0$ .

Therefore, a  $2^n : 1$  multiplexer can be made from connecting selection lines to an  $n : 2^n$  decoder and adding the  $2^n$  output to the  $2^n$  input lines, each with AND gate, and OR the  $2^n$  processed input. It is also common to see enable bit in multiplexers.



#### 7.4.2 Larger Multiplexers

Larger multiplexers can be constructed from smaller ones, by separating selection lines into multiple hierarchies of multiplexers.

#### 7.4.3 Implementing functions

Just like decoder, Boolean functions can be implemented using multiplexers. Specifically, a  $2^n : 1$  multiplexer can implement a Boolean function of  $n$  input variables, as follows:

- Express in sum-of-minterms form.
- Connect  $n$  variables to the  $n$  selection lines.
- Put a 1 on data input if it is a minterm of the function or 0 otherwise.

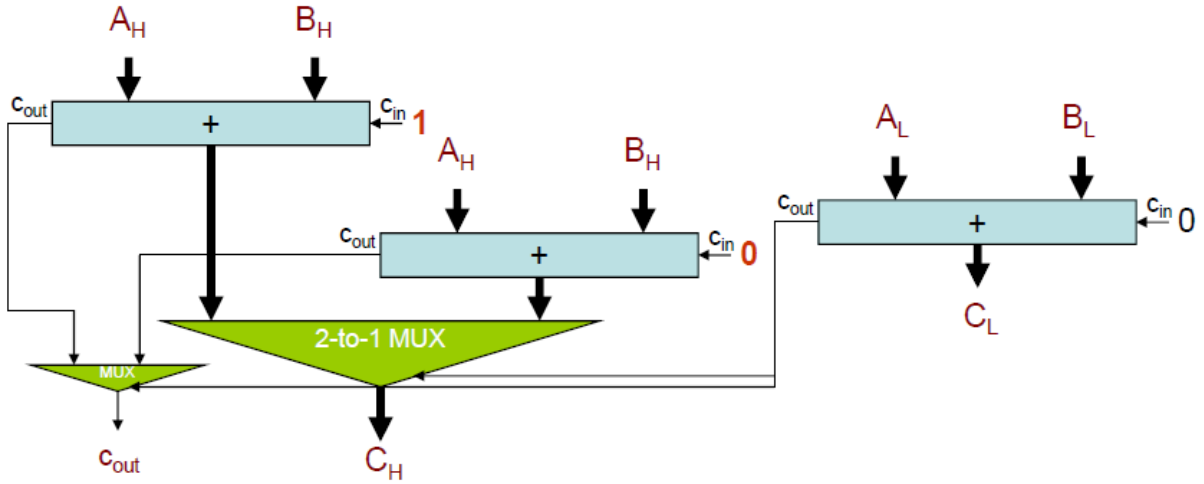
A Boolean function of  $n$  input variables can be implemented by a smaller  $2^{n-1} : 1$  multiplexer.

- Express Boolean function in sum-of-minterms form
- Reserve one variable for input lines and use the rest for selection lines.
- Use a truth table and deduce multiplexer input by comparing the reserved variable and the function value for corresponding selection line values. It may take 1, 0,  $V$ ,  $V'$ , one of the four possibilities.

## 7.5 Carry-select Adders

Carry-Select Adders reduce waiting time of the carry chain by divide-and-conquer using multiplexers.

To add two  $n$ -bit numbers  $A$  and  $B$  to produce the result  $C$ , split  $A, B, C$  into two equal halves:  $A_H, A_L, B_H, B_L, C_H, C_L$ .<sup>10</sup> The idea is that the addition of  $A_L + B_L$  will either have a



carry or not, so it computes the two scenarios for  $A_H + B_H + c$  along with  $A_L + B_L$ , and the carry out  $c$  from  $A_L + B_L$  will select the final carry out and  $C_H$ .

## 7.6 Shifters

Shifting is a common operation, as left shift by 1 bit is equivalent to multiplying by 2, and right shift by 1 bit is equivalent to dividing by 2, for positive numbers.

### 7.6.1 Arithmetic Shift

In arithmetic left shift, 0 is used to fill in the LSB.

In arithmetic right shift, the original MSB is duplicated at MSB; for 2's complement, only the part other than the sign bit is shifted.

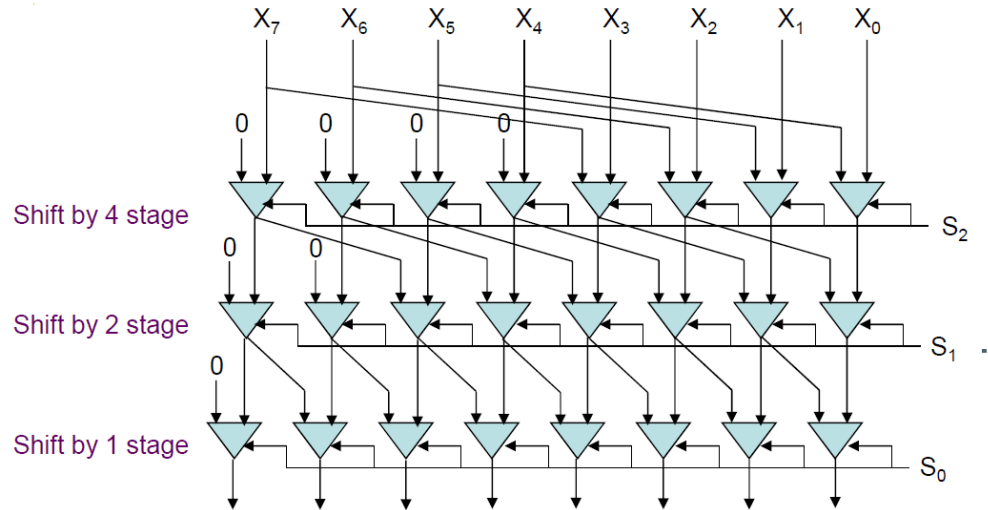
<sup>10</sup> $H$  stands for high,  $L$  stands for low.



### 7.6.2 Barrel Shifters

Barrel Shifters perform fast shifting in  $O(\log n)$  time by always shifting in the power of 2.

<sup>11</sup> The fast shifting is implemented using multiplexers. At each shifting stage, the selection



line  $S_k$ , calculated from the amount of total shifts, we decide whether to shift by  $2^k$  bits or remain unchanged.

---

<sup>11</sup>Shifting by 11 is performed by shifting  $8 + 2 + 1$ , a total of 3 times.

## 8 Sequential Logic

There are two types of sequential circuits:

- **Synchronous:** outputs change only at specific time
- **Asynchronous:** outputs change at any time

**Definition 8.1** (Finite State Machines).

**Finite State Machines** are built with combinatorial logic and **memory**, which stores the state.

Next state depends on current state and inputs.

### 8.1 $S - R$ Latch

$S - R$  latch consists of two **inputs**  $S$  and  $R$ , stands for SET and RESET respectively.

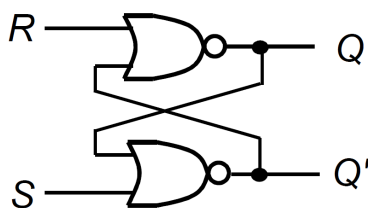
It has two **complementary output**  $Q$  and  $Q'$ .  $Q = 1 \Leftrightarrow$  latch is in SET state;  $Q = 0 \Leftrightarrow$  latch is in RESET state.

#### 8.1.1 Characteristic Table

$S$	$R$	$Q$	$Q'$	
0	0	NC	NC	No change to present state
1	0	1	0	Latch SET
0	1	0	1	Latch RESET
1	1	0	0	Invalid Condition

From this table, we have  $Q(t+1) = S + R' \cdot Q(t)$ , with restriction  $S \cdot R = 0$ .

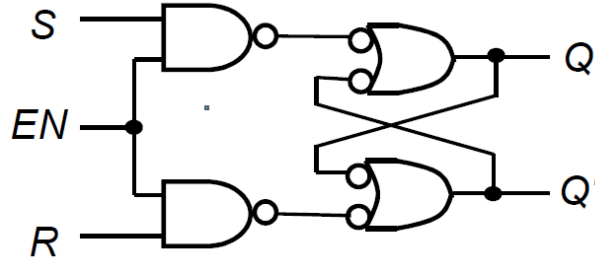
The implementation of  $S - R$  latch is as follows A  $S - R$  latch is gated if it has an enable



input( $EN$ ). Its output will change only when  $EN$  is high. Its implementation becomes<sup>12</sup>

---

<sup>12</sup>Note the position of  $S$  and  $R$  relative to  $Q$



## 8.2 Gated $D$ Latch

If  $D := S$  and  $R := S' = D'$ , a gated  $S - R$  latch becomes a gated  $D$  latch.

$D$  latch eliminates the invalid state by admitting the following characteristic table Hence,

$EN$	$D$	$Q(t+1)$	
1	0	0	RESET
1	1	1	SET
0	X	$Q(t)$	No change

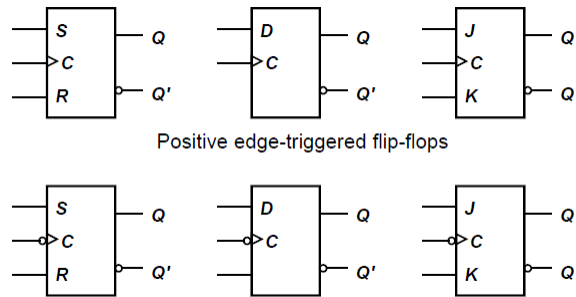
when  $EN = 1$ ,  $Q$  follows  $D$  input in a sense  $Q(t+1) = D$ , and when  $EN = 0$ ,  $Q(t+1) = Q(t)$ .

## 8.3 Flip-flops

**Definition 8.2** (Flip-flops).

**Flip-flops** are **synchronous bistable** devices. Output changes state at a specified point on a triggering input called the **clock**.

Flip-flops change state either at the positive edge or at the negative edge of the clock signal. Flip-flop family has  $S - R$  flip-flop,  $D$  flip-flop and  $J - K$  flip-flop.



## 8.4 $S - R$ flip-flop

$S - R$  flip-flop has the only difference from the  $S - R$  latch in that its output changes only on the triggering edge of the clock pulse.

Its characteristic table is

$S$	$R$	$CLK$	$Q(t+1)$	
0	0	X	$Q(t)$	No change
0	1	$\uparrow$	0	RESET
1	0	$\uparrow$	1	SET
1	1	$\uparrow$	?	Invalid

## 8.5 $D$ flip-flop

$D$  flip-flop has single input  $D$  and is similar to gated  $D$  latch. Its characteristic table is  $D$

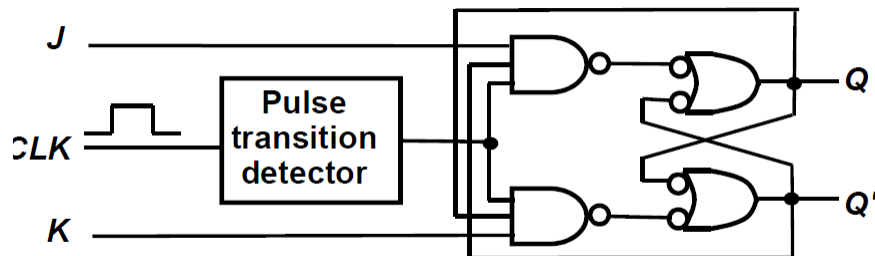
$D$	$CLK$	$Q(t+1)$	
1	$\uparrow$	1	SET
0	$\uparrow$	0	RESET

flip-flop is useful in parallel data transfer.

## 8.6 $J - K$ flip-flop

$J - K$  flip-flop is an enhancement of  $S - R$  flip-flop( $J := S, K := R$ ) by replacing the invalid state( $S = 1, R = 1$ ) by a **toggle** state, at which  $Q(t+1) = Q(t)'$ .

$J - K$  flip-flop is achieved by feeding  $Q$  and  $Q'$  to the pulse steering NAND gates.



It admits the following characteristic table:

$J$	$K$	$CLK$	$Q(t+1)$	
0	0	$\uparrow$	$Q(t)$	No change
0	1	$\uparrow$	0	RESET
1	0	$\uparrow$	1	SET
1	1	$\uparrow$	$Q(t)'$	Toggle

From the table, we have  $Q(t+1) = J \cdot Q(t)' + K' \cdot Q$ .

## 8.7 Pulse Detection Unit

The delay in the NOT gate is used for positive and negative edge detection:

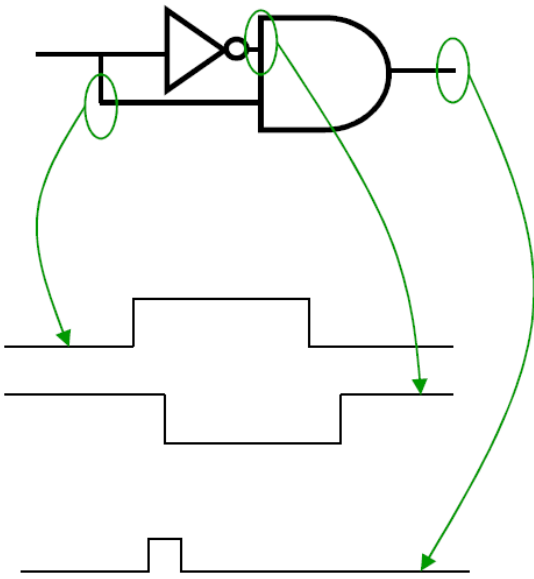


Figure 10: Positive Edge Detection

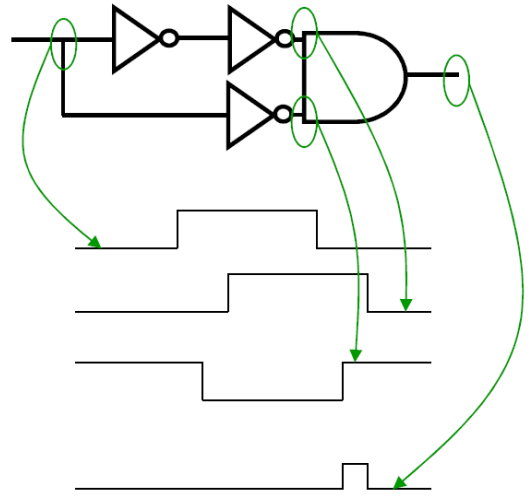
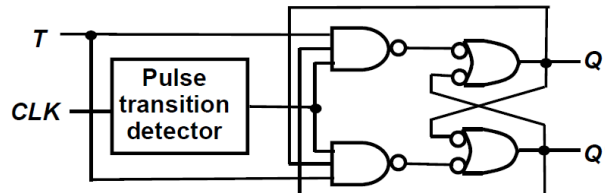


Figure 11: Negative Edge Detection

## 8.8 $T$ flip-flop

$T$  flip-flop is the single input version of the  $J - K$  flop-flop, formed by  $J := T$  and  $K := T$ . When  $T = 0$ , there is no change on  $Q$ ; when  $T = 1$ ,  $Q$  toggles.



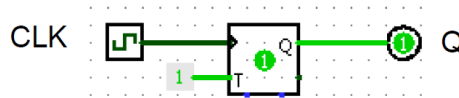
It admits the following characteristic table: From the table, we have  $Q(t+1) = T \cdot Q(t)' +$

$T$	$CLK$	$Q(t+1)$	
0	$\uparrow$	$Q(t)$	No change
1	$\uparrow$	$Q(t)'$	Toggle

$T' \cdot Q(t)$ .

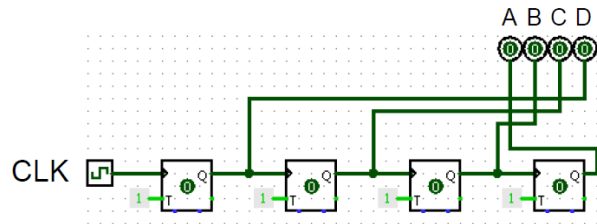
### 8.8.1 Frequency Divider

Frequency divider can be implemented using a  $T$  flip-flop with  $T$  set to 1. Therefore, the output  $Q$  will toggle on every rising edge of  $CLK$ , effectively making the output half the frequency of the clock.



### 8.8.2 4 bit countdown counter

The 4 bit countdown counter  $ABCD$  which starts from 1111, can be implemented as follows:



## 8.9 Asynchronous Inputs

$S-R$ ,  $D$  and  $J-K$  inputs are **synchronous inputs**, as data on these inputs are transferred to the flip-flop's output only on the *triggered* edge of the clock pulse.

On the other hand, **asynchronous** inputs affect the state of the flip-flop *independent* of the clock. Typical asynchronous inputs are **preset**(PRE) and **clear**(CLR).

- When  $PRE = 1$ ,  $Q$  is *immediately* set to 1.
- When  $CLR = 1$ ,  $Q$  is *immediately* cleared to 0.

Therefore, flip-flop in normal operation mode when *both* PRE and CLR are low.

In a sense, PRE and CLR overrides  $Q$ .

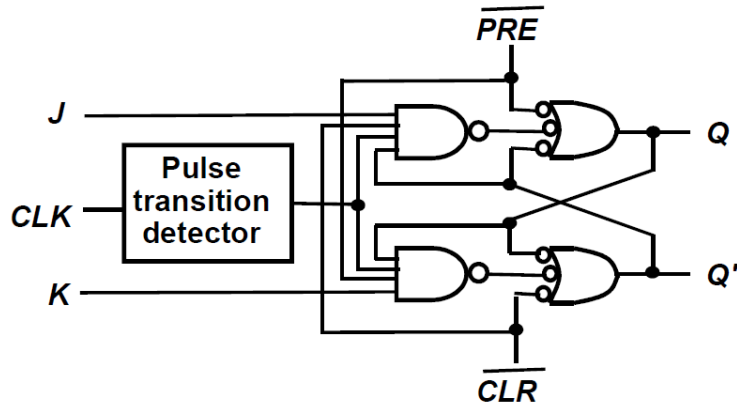


Figure 12: A  $J - K$  flop-flop with active low PRESET and CLEAR asynchronous inputs

## 8.10 Design Methodology for Sequential Logic

We illustrate the design procedures in the design of 3 bit counter.

Step 1 Formulate the problem as a truth table.

$A(t)$	$B(t)$	$C(t)$	$A(t + 1)$	$B(t + 1)$	$C(t + 1)$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Step 2 Assign one flip-flop for each of the **current** state bits  $X(t)$ .

Here we use 3  $J - K$  flip-flop for  $A$ ,  $B$ ,  $C$ .

Step 3 Draw truth table for flip-flop inputs. Note any output will be dependent on **all** the input.

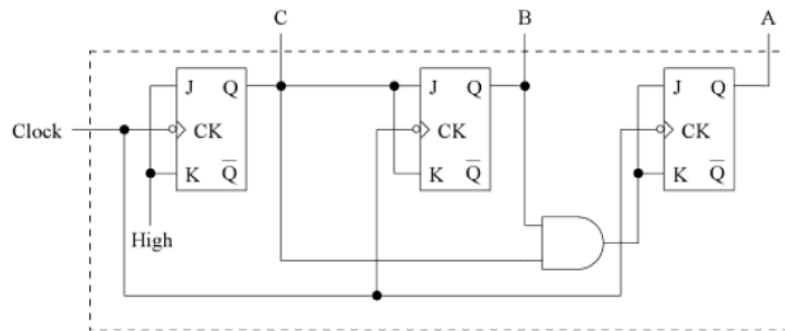
$A(t)$	$B(t)$	$C(t)$	$A(t+1)$	$B(t+1)$	$C(t+1)$	$J_A$	$K_A$
0	0	0	0	0	1	0	X
0	0	1	0	1	0	0	X
0	1	0	0	1	1	0	X
0	1	1	1	0	0	1	X
1	0	0	1	0	1	X	0
1	0	1	1	1	0	X	0
1	1	0	1	1	1	X	0
1	1	1	0	0	0	X	1

Step 4 Using K map, generate minimum SOP for all inputs in the assigned flip-flops.

$$J_A = BC$$

$$K_A = BC$$

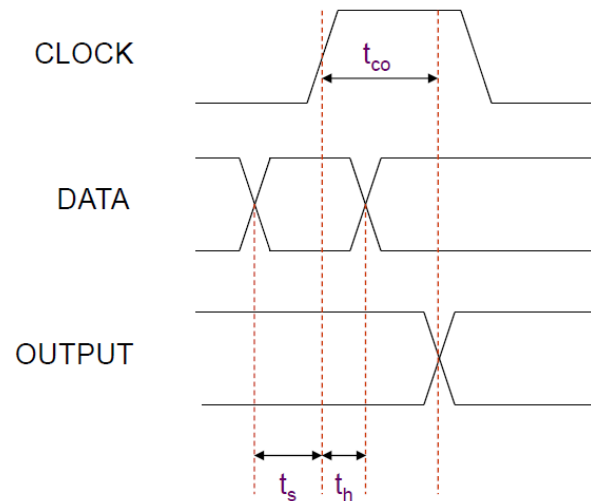
Step 5 Implement in circuit



## 8.11 Metastability

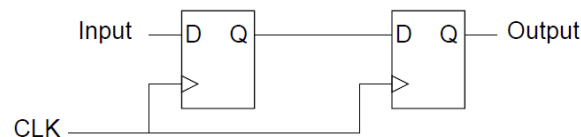
In reality, nothing is instantaneous. Data will have a **setup time**  $t_s$  and a **hold time**  $t_h$  to observe, while **clock-to-output time**  $t_{co}$ , which is the propagation delay, is associated with the clock and output.





**Metastability** is introduced if setup and hold times are violated, as flip-flop may oscillate in an indeterminate state between 0 and 1. Although metastability cannot be absolutely avoided in practice, two way to resolve it can be

- Make sure clock period is long enough.
- Use flip-flop chain Probability of metastability gets closer and closer to zero as number



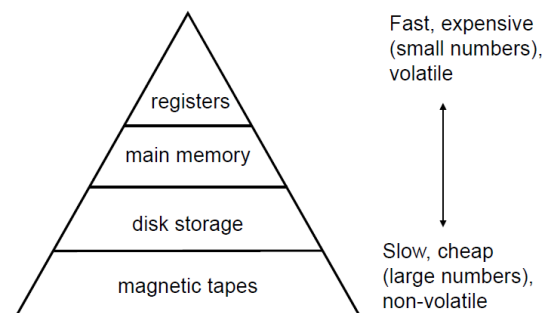
of flip-flops connected in series increase.

## 8.12 Memory Hierarchy

**Memory** stores programs and data.

We define

- 1 byte = 8 bits
- 1 KB =  $2^{10}$  bytes
- 1 MB =  $2^{20}$  bytes
- 1 GB =  $2^{30}$  bytes
- 1 TB =  $2^{40}$  bytes



## 9 Understanding Performance

### 9.1 Defining Performance

To improve performance, we are interested to minimise

- Response time/**Execution time**: the time between the start and the completion of a task.

Execution time is *inversely* related to the performance.

$$\text{performance} = \frac{1}{\text{execution time}}$$

- **Throughput** – Total amount of work done in a given time  
Decreasing response time always improves throughput.

**Remark:** *elapsed* time do not equal execution time.

- CPU execution time(CPU time) is the time CPU spends working on a task
- CPU execution time does *not* include time (1) waiting for I/O or (2) running other programmes.

### 9.2 CPU Execution Time

**Definition 9.1** (CPU Execution Time).

CPU execution time for a program is given by

$$\begin{aligned}\text{CPU execution time} &= \# \text{ of CPU clock cycle} \times \text{clock cycle time (s)} \\ &= \frac{\# \text{ of CPU clock cycle}}{\text{clock rate (Hz)}}\end{aligned}$$

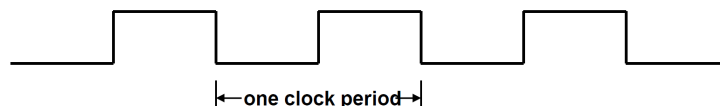
Therefore, there are 2 ways to improve performance, by either

1. Reducing the length of the clock cycle, or
2. Reducing the # of clock cycles required for a program

**Definition 9.2** (Clock Rate).

Clock rate is the inverse of clock cycle time.

$$\text{Clock rate(CR)} = \frac{1}{\text{Clock Cycle Time(CC)}}$$



### 9.3 Clock Cycles per Instruction

Different instructions take different amount of time to execute. On average, we have

$$\# \text{ of CPU cycles} = \# \text{ of Instructions} \times \text{Average Clock Cycles per Instruction (CPI)}$$

where Clock Cycles per Instruction is defined as:

**Definition 9.3** (Clock Cycle per Instruction).

**Clock cycles per instruction** is the average number of clock cycles each instruction takes to execute.

This allows the measurement of clock cycles per instruction for different instruction class. Overall, we can calculate **effective CPI** defined as below.

**Definition 9.4** (Effective CPI).

Overall effective CPI is calculated by a weighted average of clock cycle per instruction among all classes of instructions.

$$\text{Overall effective CPI} = \sum_{i \in I} \text{CPI}_i \times \text{IC}_i$$

where IC stands for the **percentage** of instruction count, serving as the weight.

### 9.4 CPU performance

**Definition 9.5** (Performance Equation).

The performance equation of CPU is

$$\text{CPU time} = \text{instruction count (IC)} \times \text{CPI} \times \text{clock cycle} = \frac{\text{IC} \times \text{CPI}}{\text{clock rate}}$$

This equation gives us a way to calculate average CPI.

There are multiple factors affecting CPU performance.

	Instruction Count	CPI	Clock Cycle
Algorithm	*	*	
Programming language	*	*	
Compiler	*	*	
ISA	*	*	*
Processor Organisation		*	*
Technology			*