Digital Design

Ranim Tom

September 1, 2025

Contents

Todo list

Abstract

This report is about Digital Design. Main resources are:

- Introduction to Logic Circuits and Logic Design with Verilog [1]
- Digital Electronics: A practical approach with VHDL [2]

Chapter 1

Number Systems and Codes

1.1 Digital Vs Analog

- Contains some nice example about Analog and digital signals
  - The analogy between a continuous time signal and the motion of a clock
- A nice example is the Solar radiation data-logger system (Figure 1-5 in the textbook [2])

1.2 Introduction

Since we will deal with circuit which produce 2 states, high low, 0 1, we need to understand or to invent some number system which deals with such values.

However, our brain is used to decimal system because that's what we are taught. So what we need to do is to adapt to binary system which computer uses to do operations and tasks.

In order to do so, what will do 1st is understand positional number system, then move to binary systems.

1.3 Positional Number system

Definition: In each position of the number, the symbol (digit) takes a different value.

Let's look at Figure 1.1, where the decimal system is presented.

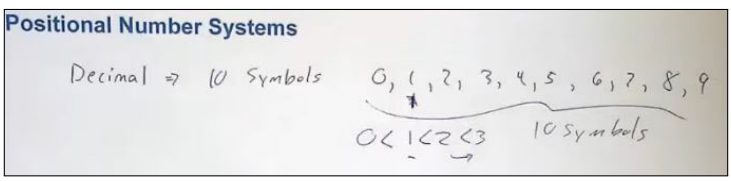


Figure 1.1: Decimal System

- When we define a system, we don't just list its numbers (from $0 \rightarrow 9$)
- But also each digit is bigger then the previous one by the smallest non-zero amount
  - $1 > 0$ by 1
  - $2 > 1$ by 1 and $3 > 2$ by 1

Counting:

When counting, what we actually do is adding extra column by shifting 1 position, as shown in Figure 1.2.

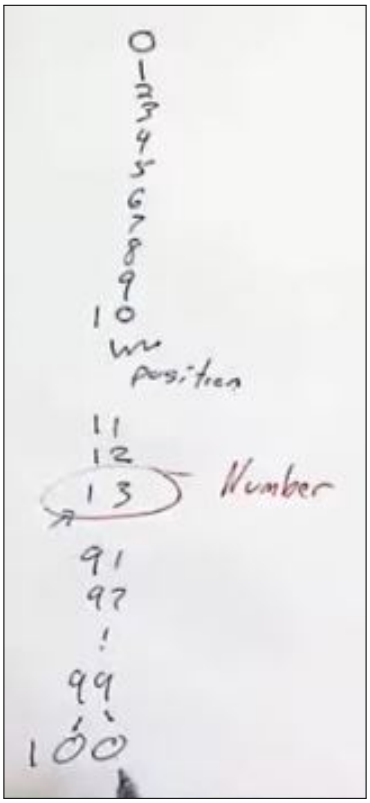


Figure 1.2: Decimal System: Counting

1.3.1 Some Terminology and Definitions

In Figure 1.3, we have several definition and terminology

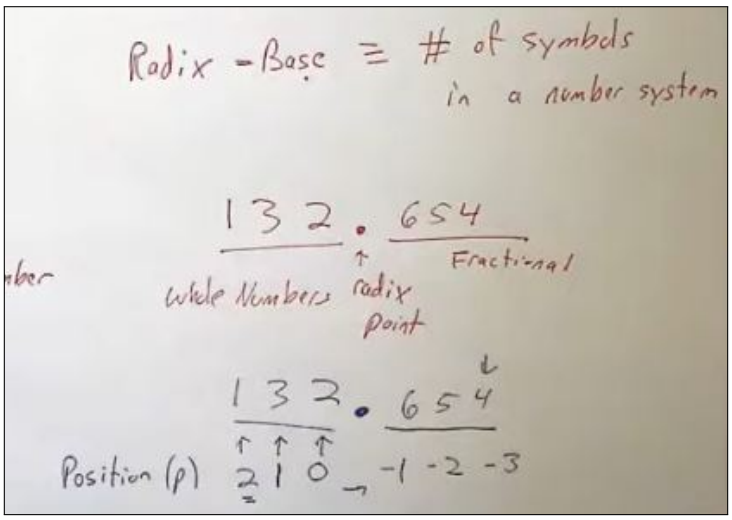


Figure 1.3: Positional Number System Definitions

- Radix = Base
  - If we have base 10, this means we have 10 symbols
  - Base 2, this means we have 2 symbols
- Radix point: separate numbers from fractional part. Radix point is often affiliated relative to the base
  - Base 10: we say decimal point
- Positions: in the example 132.654: each digit gets assigned some position

Also other definitions are shown in Figure 1.4

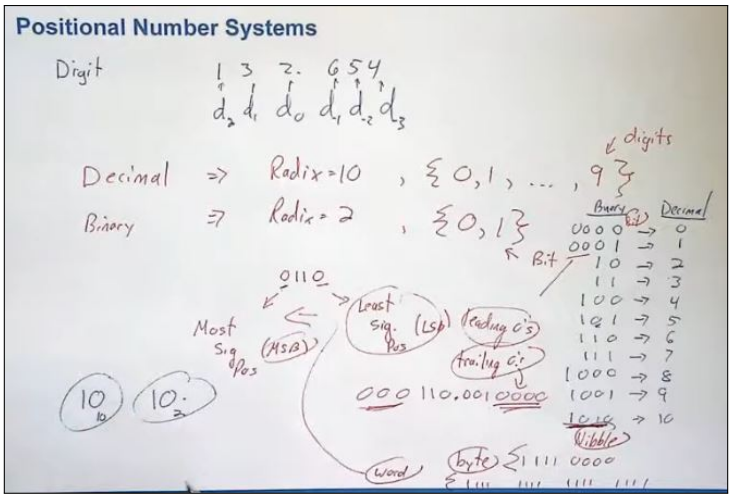


Figure 1.4: Positional Number System Definitions

- For binary: we don't say digit but we say bit
- Grouping bits we have:
  - 4 bits = nibble
  - 8 bits = Byte
  - 16 bits = word

Now base 10 and binary are the most important ones (decimal because we as human we use it, and binary because computers use it). However, what we ca notice that in binary, we run out of numbers directly, and we need to add extra columns frequently.

Also when have long binary numbers, it is hard to communicate with them. That's why hexadecimal system is used (see Figure 1.5)

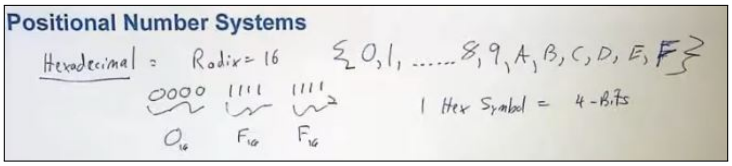


Figure 1.5: Hexadecimal System

1.4 Base Conversion: From $2^n \rightarrow 10$

Now we start base conversion. First we begin with base 10, and see how to decimal system is decomposed (because this decomposition will be used later.)

After this step, we see how to convert base 2 $\rightarrow$ base 10 from the decomposition we leared before. Finally, we can generalize this to any base system.
Source: Section 1.3 $\rightarrow$ 1.8 in [2]

1.4.1 Base 10 Decomposition

In Figure 1.6, we have an example how to decompose 132.654

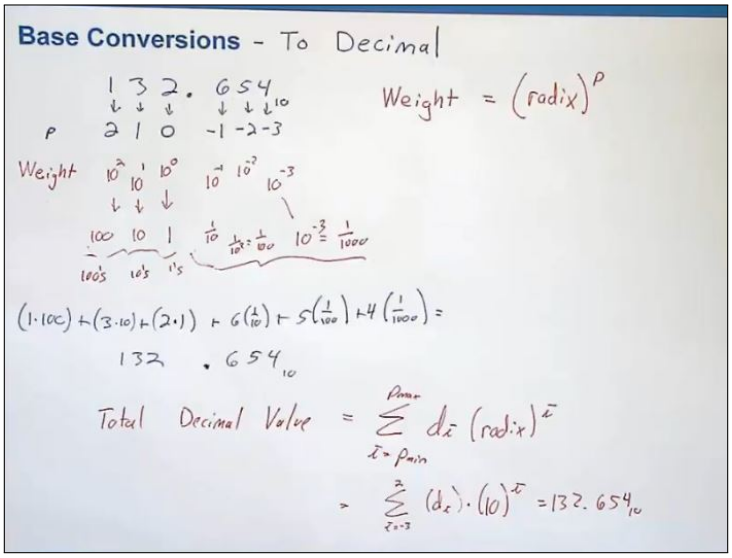


Figure 1.6: Base 10 Decomposition

- The notation shown in red is called ***digit notation***, where it is equal

$$\text{Weight} = (\text{radix})^p \tag{1.1}$$

  - In case of base 10, Equation 1.1 gives Weight $= (10)^p$

- In equation form, we can write the conversion as:

$$\text{Total Decimal Value} = \sum_{i=p_{\min}}^{p_{\max}} d_i \, \text{radix}^i \tag{1.2}$$

Another example in a more clear way is shown in Figure 1.9.

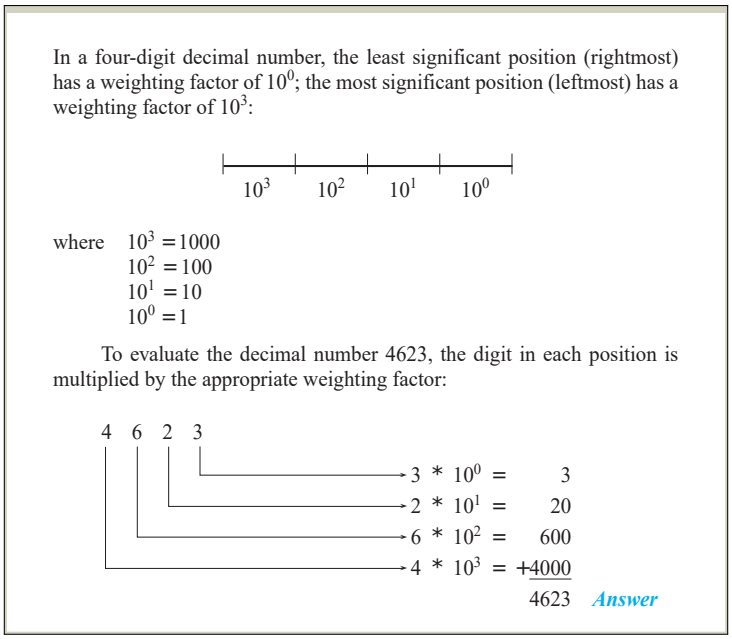


Figure 1.7: Base 10 Description [2]

- Notice that position is represented by 10^x where $x \in \mathbb{N}$

1.4.2 Conversion Base 2 $\rightarrow$ Base 10

We can also apply Equation 1.2 by adjusting the radix = 2, since we are in base 2 now. An example is shown in Figure 1.8

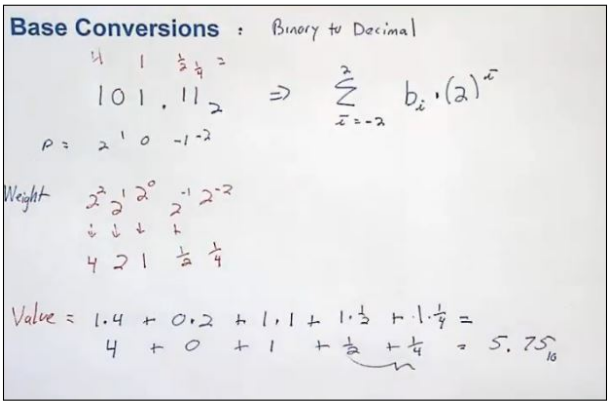


Figure 1.8: Base 2 to base 10 Example

TABLE 1–1 Powers-of-2 Binary Weighting Factors

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$

Figure 1.9: Base 2 Description [2]

- In base 2, we do the same thing, but with power of 2 instead of $10 \leftrightarrow 2^x$ where $x \in \mathbb{N}$

Other example in Figure 1.10.

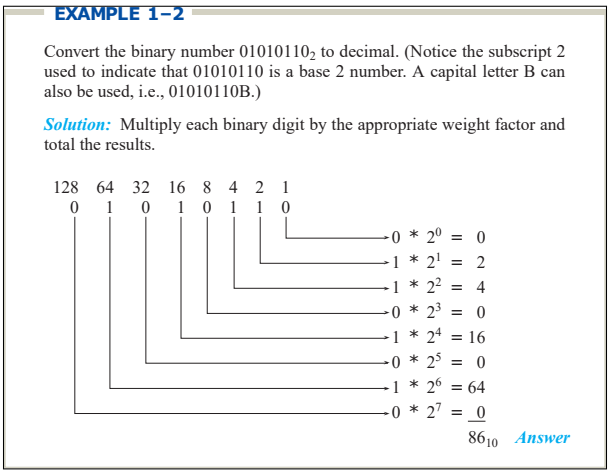


Figure 1.10: Base 2 to 10 Conversion [2]

1.4.3 Conversion Hexadecimal $\rightarrow$ Base 10

Figure 1.11 shows an example from hexadecimal to base 10.

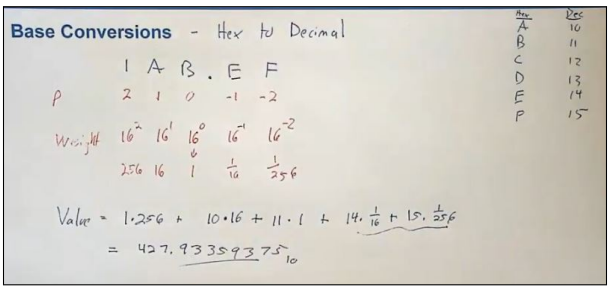


Figure 1.11: Base 2 to base 10 Example

1.5 Base Conversion: Base 10 → Base 2^n

Now we will do the opposite. The algorithmic way is a little bit different in this case: we will treat the integer side different from the fractional side as we will see.

1.5.1 Base 10 → Base 2

An example is shown in Figure 1.12,for the number 11.375_{10}.

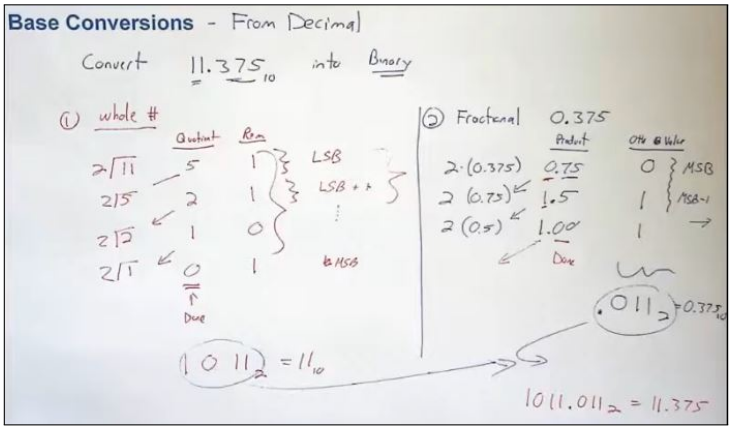
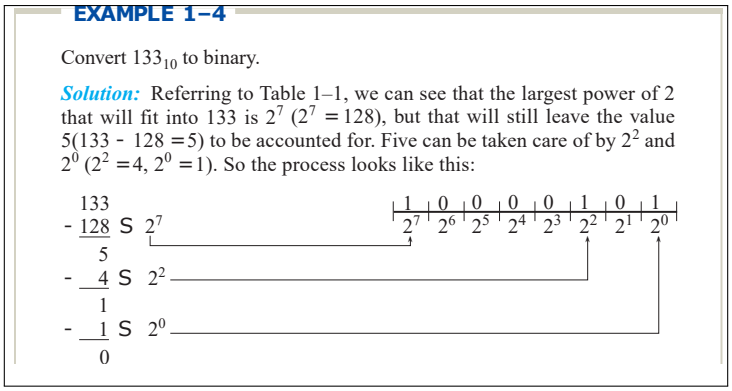


Figure 1.12: Base 10 to 2 Conversion [1]

- First we take the integer part 11
  - We divide by 2 and take the quotient as LSB (that is we are filling positions starting from right side)
  - If we reach quotient = 0, then we stop
- For the fractional part, we multiply by 2 instead of dividing
  - Notice that when we obtain 1.5, we take 0.5 and not 1.5
  - If we reach 0 in the fractional part of the multiplication, we stop
  - In fractional part, we are filling the positions starting from MSB (so starting from left side)

Another example in Figure 1.13.



EXAMPLE 1–4

Convert 133_{10} to binary.

Solution: Referring to Table 1–1, we can see that the largest power of 2 that will fit into 133 is 2^7 ($2^7 = 128$), but that will still leave the value 5(133 - 128 = 5) to be accounted for. Five can be taken care of by 2^2 and 2^0 ($2^2 = 4$, $2^0 = 1$). So the process looks like this:

133
- 128 S 2^7
5
- 4 S 2^2
1
- 1 S 2^0
0

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

Figure 1.13: Base 10 to 2 Conversion [2]

Note: when converting fractional part, in some cases we have some rational numbers where their multiplication won't give 0, so the multiplication won't end.
In such cases, we can stop at certain number of bits required.

1.5.2 Base 10 → Hex

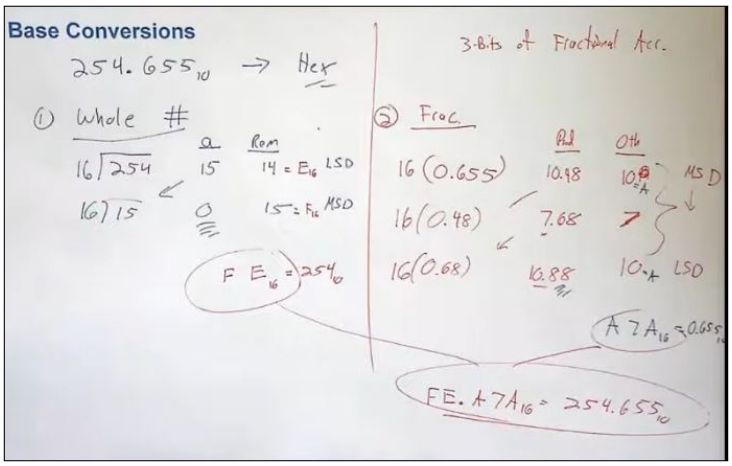


Figure 1.14: Base 10 to 16 Conversion [1]

- Since we are in base 16, we divide by 16 and multiply by 16 when doing the fractional part
- In this example, the multiplication will go for long so we stop at 3 digit length

1.6 Conversion Between Base of 2^n

Suppose for example that we wish to move from base 2 to Hex or vice versa. These situations are easier then decimal cases (from or to)
An example is shown in Figure 1.15.

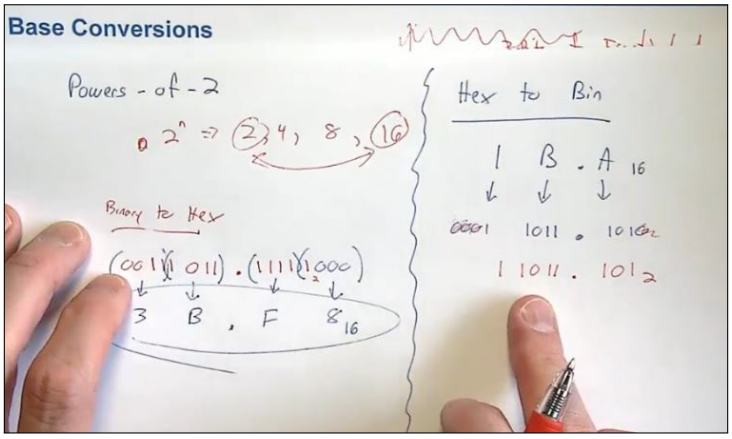


Figure 1.15: Converting between basis of 2^n [1]

- From base 2 to 16: a grouping by 4 sine $2^4 = 16$
- From base 16 to 2: direct mapping. Each digit in base 16 corresponds 4 bits in base 2

1.7 Base 8 (Octal)

1.7.1 Representation

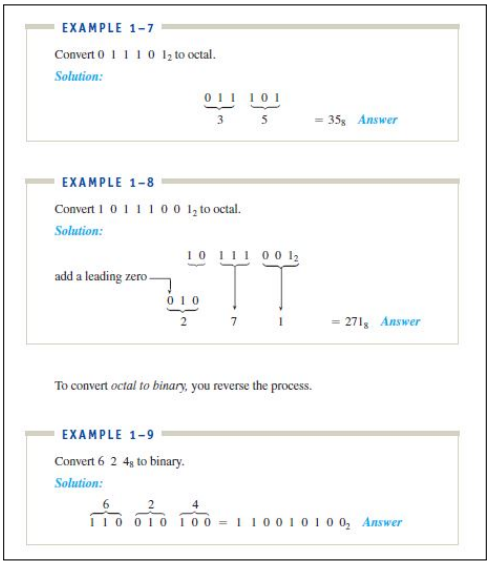
TABLE 1–2 Octal Numbering System

Decimal	Binary	Octal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	10
9	1001	11
10	1010	12

Figure 1.16: Base 8 Table [2]

- Base 8: from $0 \rightarrow 7$
- We use only 3 bits because $2^3 = 8$
- Once we are beyond 7, we add 1

1.7.2 Conversion Base 2 → Base 8 and 8 → 2



EXAMPLE 1–7

Convert $0\ 1\ 1\ 1\ 0\ 1_2$ to octal.

Solution:

$\underbrace{0\ 1\ 1}_{3}\ \underbrace{1\ 0\ 1}_{5} = 35_8$ *Answer*

EXAMPLE 1–8

Convert $1\ 0\ 1\ 1\ 1\ 0\ 0\ 1_2$ to octal.

Solution:

add a leading zero → $\underbrace{0\ 1\ 0}_{2}\ \underbrace{1\ 1\ 1}_{7}\ \underbrace{0\ 0\ 1}_{1}{}_2 = 271_8$ *Answer*

To convert *octal to binary*, you reverse the process.

EXAMPLE 1–9

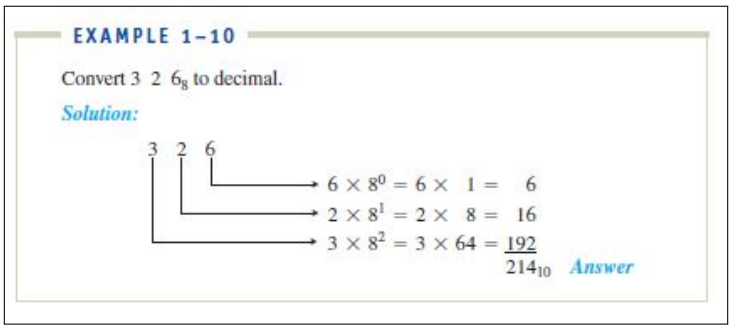
Convert $6\ 2\ 4_8$ to binary.

Solution:

$\underbrace{1\ 1\ 0}_{6}\ \underbrace{0\ 1\ 0}_{2}\ \underbrace{1\ 0\ 0}_{4} = 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0_2$ *Answer*

Figure 1.17: Base 2 to 8 Conversion and the last example 8 → 2 [2]

1.7.3 Conversion Base 8 → 10



EXAMPLE 1–10

Convert $3\ 2\ 6_8$ to decimal.

Solution:

$3\ 2\ 6$

$6 \times 8^0 = 6 \times 1 = 6$

$2 \times 8^1 = 2 \times 8 = 16$

$3 \times 8^2 = 3 \times 64 = 192$

214_{10} *Answer*

Figure 1.18: Base 8 → 10 Conversion [2]

1.8 Hexadecimal Base and BCD

Hex and BC

- To insert tables and Example later

1.9 Comparing Number Systems

TABLE 1–4 Comparison of Numbering Systems

Decimal	Binary	Octal	Hexadecimal	BCD
0	0000	0	0	0000
1	0001	1	1	0001
2	0010	2	2	0010
3	0011	3	3	0011
4	0100	4	4	0100
5	0101	5	5	0101
6	0110	6	6	0110
7	0111	7	7	0111
8	1000	1 0	8	1000
9	1001	1 1	9	1001
10	1010	1 2	A	0001 0000
11	1011	1 3	B	0001 0001
12	1100	1 4	C	0001 0010
13	1101	1 5	D	0001 0011
14	1110	1 6	E	0001 0100
15	1111	1 7	F	0001 0101
16	0001 0000	2 0	1 0	0001 0110
17	0001 0001	2 1	1 1	0001 0111
18	0001 0010	2 2	1 2	0001 1000
19	0001 0011	2 3	1 3	0001 1001
20	0001 0100	2 4	1 4	0010 0000

Figure 1.19: All Bases [2]

1.10 Binary Arithmetic

1.10.1 Binary Addition

What will need to keep in mind while learning arithmetic operation in general, that at the end we will build circuit which do such operations.
First we start by binary addition.
The four possible combinations of adding two binary numbers can be stated as follows:

- $0 + 0 = 0$ carry 0
- $0 + 1 = 1$ carry 0
- $1 + 0 = 1$ carry 0
- $1 + 1 = 0$ carry 1

General Form: $A_0 + B_0 = \sum_0 + C_{\text{out}}$

Note: When the binary sum exceeds 1, you must carry a 1 to the next-more-significant column, as in regular decimal addition.

In a truth table form:

Truth Table for Addition of Two Binary Digits $A_0 + B_0$ in the Least Significant Column			
A_0	B_0	Sum_0	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 1.20: Truth table of binary Addition [2]

What we must keep in mind that when working with circuit which do arithmetic:

- we will have usually 1 circuit for the carry and 1 circuit for bit addition.
- Also we need to keep in mind how many bits we obtain in sum and how many for carry bit
- In general, when do n bit + n bit, we obtain $n+1$ result, where the extra bit is for the carry

An example is shown in Figure 1.21

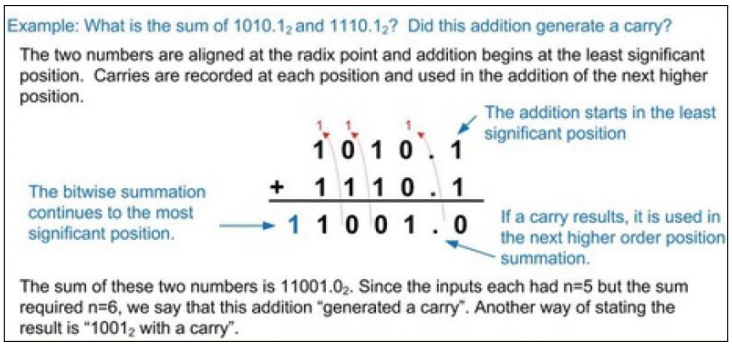


Figure 1.21: Binary Addition Example [1]

1.10.2 Binary Subtraction

Now for binary subtraction, different combinations are shown in Figure 1.22.

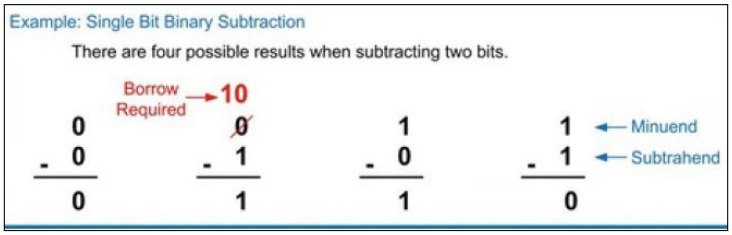


Figure 1.22: Binary Subtraction Possible Values [1]

- In addition we had carry, while in subtraction we have borrow

Let's take 2 example of binary subtraction shown in Figure 1.23.

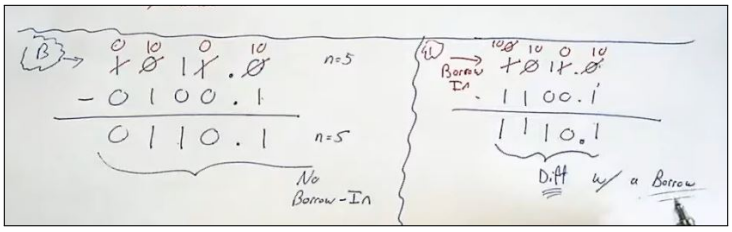


Figure 1.23: Binary Subtraction Examples [1]

- IN the example of the left, we have a result with no borrow left
- While in the right example, we have result but with a borrow in
  - This means we need extra circuit to do this borrow bit

1.11 Unsigned and Signed Numbers

Until now, we have seen base conversions and arithmetic operations for binary numbers. But what also what is important is the code that we can represent using binary numbers.

1.11.1 Unsigned Numbers

For unsigned numbers, we have 2 things.
First, let n be the number of bits we have.

- # of unique codes = 2^n
- ***Range*** N_{unisgned}:
  $0 \leq N_{\text{unisgned}} \leq 2^n - 1$

1.11.2 Signed Numbers

Now for signed numbers, the convention is we allocate a bit for the signal at the MSB position: 0 for positive and 1 for negative.

When we are told we are dealing with signed numbers, we need to directly specify the bit width n: we can't for example say we have a signed number 11_2, then say we have 1111_2, because the signed bit position get shifted in 1111_2.
The correct to say is we have 0011_2 and 1111_2 is a singed system, where $n = 4$ in this case.

Now we have 3 ways to encode signed numbers: signed magnitude, 1^{st} complement and two complement.

Signed Magnitude

Signed magnitude isn't used that much anymore, but we will look how it works.

- Suppose we need to represent -2_{10} using bit width $n = 4$
- First we take absolute value that is 2, then coded for binary. It will be 010
- Then we append at the MSB the signed bit: in this case it will be 1 since we have negative number
- Result: $-2_{10} = 1010_2$ using signed bit system with $n = 4$ bits

1^{st} complement

In this representation, we simply complement the positive representation (flip 1 to 0 and 0 to 1).

Example: Assuming we have 4 bits, $7_{10} = 0111$ and $-7_{10} = 1000$.

Range: $-2^{n-1} - 1 \leq N_{\text{signed,1}} \leq 2^{n-1} - 1$.

For example for 4 bits, the range will be $-7 \leq N_{\text{signed},1} \leq +7$, and notice we use 2 different code to represent the same information , which is the 0 number: $+0_{10} = 0000$ and $-0_{10} = 1111$, which is not very efficient.

Roll over: A nice thing about 1^{st} complement is the phenomena of roll over, where for example (as shown in Figure 1.24), if we reach 8 $(7 + 1)$, its representation is 1000, which is the most negative part in 4 bits width $-7_{10} = 1000$.

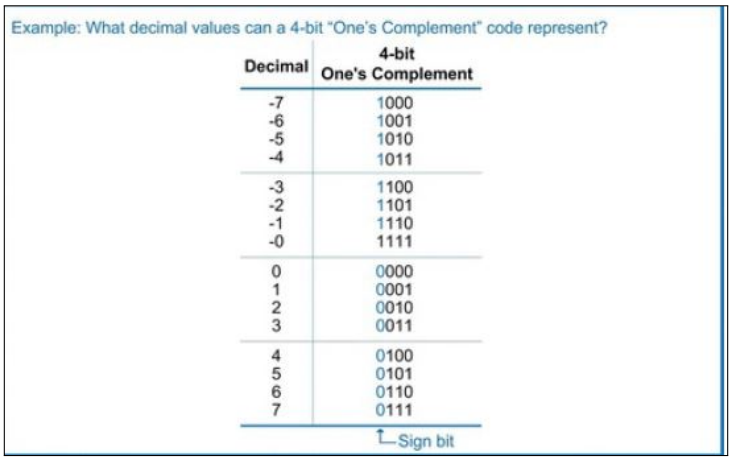


Figure 1.24: 1^{st} complement for 4 bits [1]

This phenomena can be useful in counter circuits.

2^{nd} complement

2^{nd} complement is the widely used today in computer number representation.

It consists of 2 operations:

1. Negation (by simply taking the 1^{st} complement)
2. The adding $+1$

Example: Suppose we want to find binary representation of -1_{10}.

1. We negate the positive part: $+1_{10} = 0001 \rightarrow 1110$
2. We add $+1$ $1110 + 1 = 1111 = -1_{10}$

Now Figure 1.25 shows all numbers coded using 2^{nd} complement assuming 4 bit.

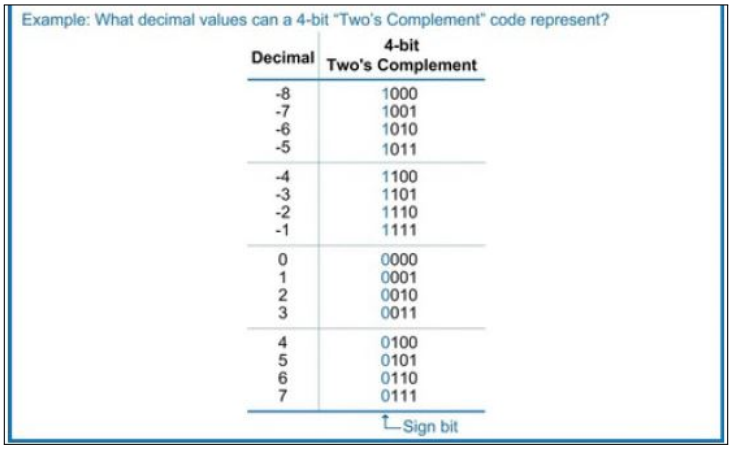


Figure 1.25: 2^{nd} complement for 4 bits [1]

Notice now compared to 1^{st} complement (Figure 1.24), we can represent -8, since 0 is only represented once now, and this has due to the +1, which has eliminate the redundancy.

So now we have 16 code for 16 information (no redundancy), and the range is extended: $-2^{n-1} \leq N_{\text{signed},2} \leq 2^{n-1} - 1$.

More Examples:
Convert $+6_{10}$:

- : 6 is 0110
- We flipp we have 1001
- We add +1 $1001 + 1 = 1010$, and we neglect the carry

Now suppose we are give the code 1001_2, how do we know its decimal representation? We must know in what system we are working.

Suppose we are working in 2^{nd} complement for 4 bits, what we do is:

- Since the code start with 1, this mean the number is negative
- We flip 0110
- We add +1 0111, this corresponds to $+7_{10}$
- Final answer is -7_{10}

2^{nd} complement Arithmetic

The power of 2^{nd} complement is that it enable us to do subtraction using addition.
Let's take for example $6_{10} - 3_{10}$: this can be written as $6_{10} + -3_{10}$. In terms of circuitry, the expression $6_{10} + -3_{10}$ that we can use an adder and search for the negative representation of 3, while the 1^{st} expression $6_{10} - 3_{10}$ tell us to build an subtractor circuit,

which can be more costly in terms of logic circuit.

For $6_{10} + -3_{10}$:

- We search for the representation of -3_{10}. It will be 1101
- Now we add $6_{10} + -3_{10} = 0110 + 1101 = 0011 = 3_{10}$

2^{nd} complement Overflow

Suppose we have to add $7_{10} + 7_{10}$, using 2^{nd} complement for 4 bits. We expect 14_{10} as result. However, using 4 bits we can't represent 14_{10}. The result will give negative number (1110 = -2).

So what we can do is to build a separate circuit which verify if we have overflow by testing the sign bit,

Chapter 2

Digital Signals and Switches

Source: chapter 2 in [2].

2.1 Digital Signals and Clock Waveform

- Contains basic definition about how a digital signal (0 and 5 V), the clock waveform,$\cdots$

2.2 Serial and Parallel Communication

There are 2 way to communicate between 2 digital system: in serial and parallel.

Serial Communication:

- Inexpensive since we only use 1 conductor and 1 I/O
- However it is slower since we can transmit only 1 bit at a time
- Rule: We always transmit the LSB first
- Examples of Serial Communication: USB,ehternet, any form of coaxial cable, DSL,$\cdots$

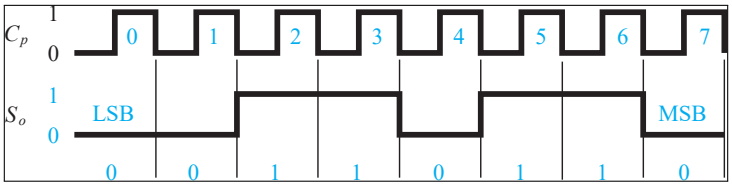


Figure 2.1: Serial Communication Example: Serial transmission of 01101100 [2]

Parallel Communication:

- It is much faster since we can transmit all bits in 1 clock period
- However, in terms of hardware it is much costly:
  - We use many conductors
  - We use also many I/O on each side of the communicating devices

2.3 Relays, Diodes,TTL

ctronics of
c circuit

: to be developed later.
The rest of these sections talk about the electronic aspect of logic design: what electronics we use in digital system, the building block of logic gates.

Chapter 3

Digital Circuitry and Interfacing

3.1 Describing Logic Circuit

There are 3 ways to describe a logic circuit (see chapter 3 (section 3.1) in [1]):

1. Logic Symbol or a block diagram
2. Truth table: which describes all input combination and output
3. Logic Equation
4. Logic Waveforms

3.2 Logic gates

Essential Points:

- AND is 1: if A and B are both high simultaneously
- OR is 1: whenever one of the inputs or both is high

For more details, see sections 3-1 and 3-2 in [2].

3.2.1 NAND and NOR Gate

NAND and NOR gate is jut an AND and OR gates followed by an inverter as shown Figure 3.1 in and Figure 3.2

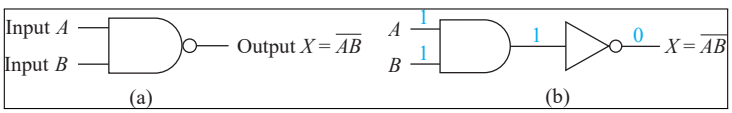


Figure 3.1: NAND Gate [2]

- Note that $\overline{A\,B} \neq \overline{A}\,\overline{B}$
- $\overline{A}\,\overline{B}$ is simply A and B inverted then inputted to an AND gate

Note: See example 3-9 about the application of an NAND gate and a control signal to generate an active low signal.

- Hint: the signal is not active low until the control signal is active (high)

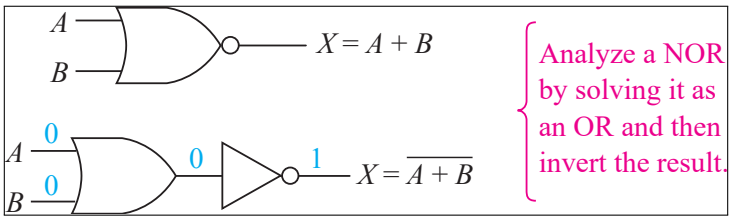


Figure 3.2: NOR Gate [2]

3.2.2 XOR and XNOR

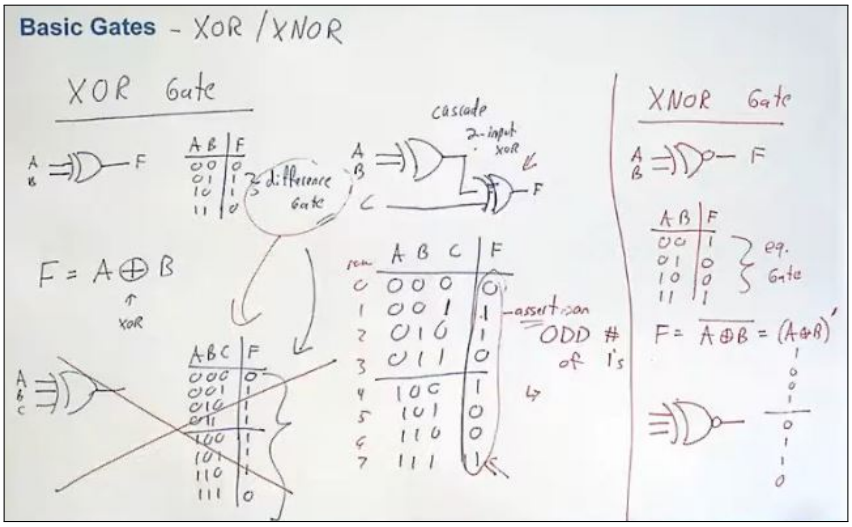


Figure 3.3: XOR and X-NOR Gate

- The XOR gate is called the ***difference gate***: it asserts when A and B are different
- For 3 input XOR gate: the direct definition isn't applied that much
- Instead, we use ***cascaded form*** of 2 XOR gates (notice how the truth table isn't the same at the output column)
  - It can be used in parity check application (output is 1 when the number of 1 is odd)
  - Or can be used in binary adder: when we need to build circuit which add sum of 3 bits, and produce the 1 bit sum

3.3 Digital Circuit Operation

Source: see section 3.2 in [1].

In big picture, this section talks about the electrical description of logic circuit. Its even lower level then gate level description.

Note: I will write only the important details in this section, since all the details are given in electronic/electrical circuit courses.

3.3.1 Power Supply

Take away and main points:

- When we have 2 devices Tx and Rx connected, we have to make sure that voltages and current specification don't exceed specification
- If we do exceed, the excess of current can generate some heat and melt the device

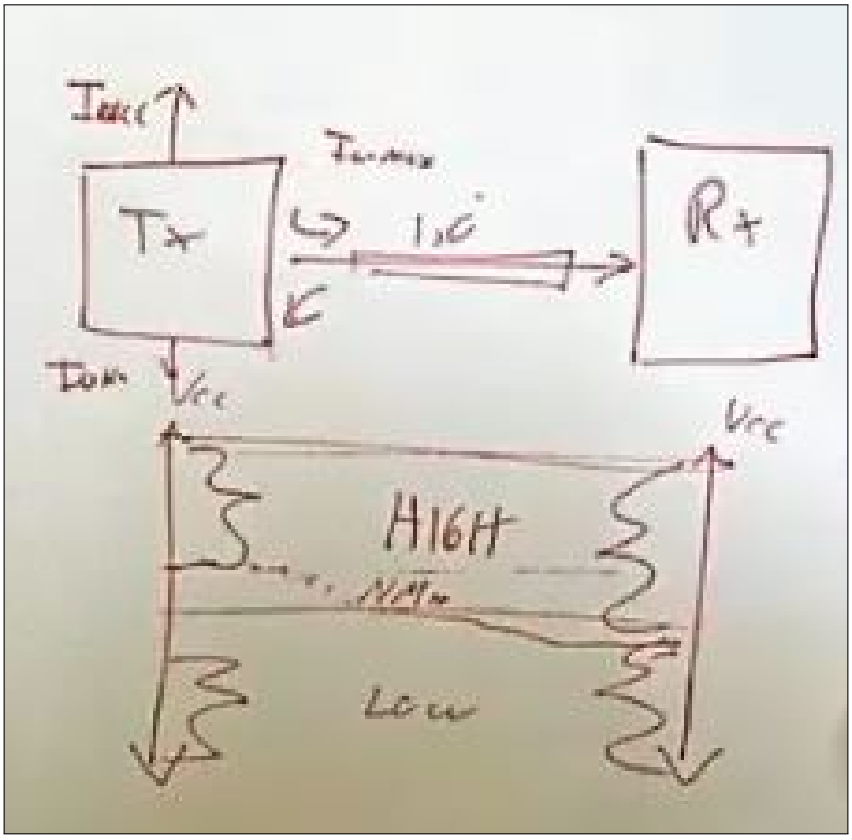


Figure 3.4: Tx-Rx Specification

3.3.2 Switching Characteritic

Main Points:

- For steady and transient state: we have rise time and fall time
  - Important characteristic: design circuit in which transient time is negligible, so the circuit won't have time to respond and deals with values which toggle between 0 and 1
  - If we are only given transient time, this means that rise and fall time are symmetrical (the same) at the rising and falling edge of the pulse

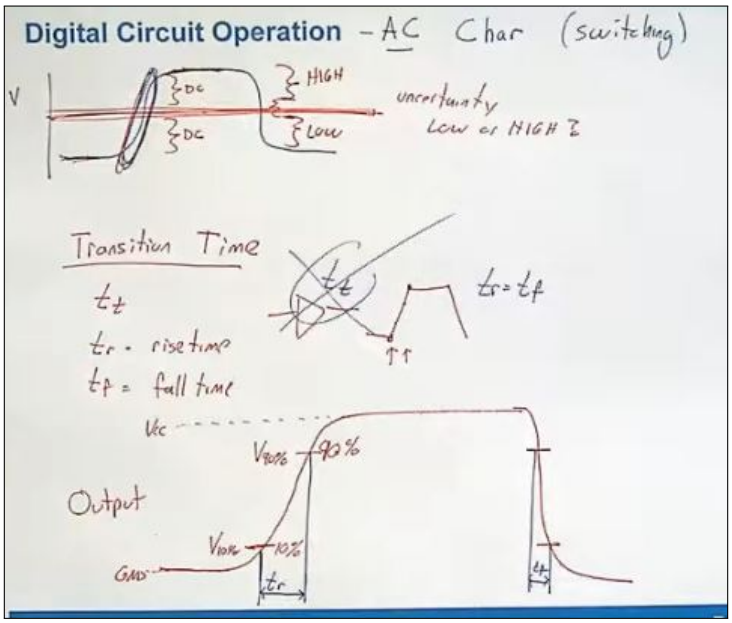


Figure 3.5: Transient Time: rise and fall time

- Propagation delay: how long the circuit take to change values (from high to low or low to high) relative to the input

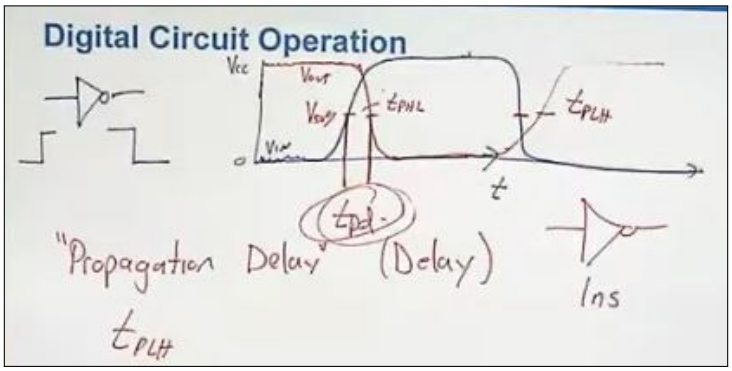


Figure 3.6: Propagation delay

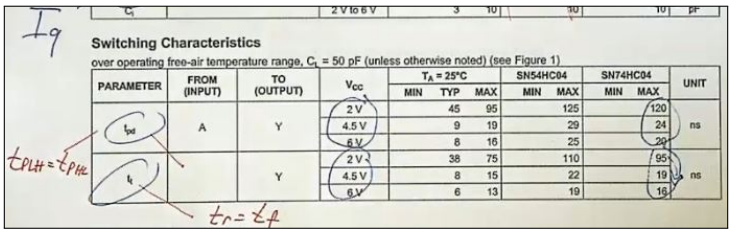
  - What we do in defining propagation delay, is to mark the 50 % values from switching from low to high as input
  - If we are giving only propagation delay, this means switching high to low or low to high are the same

3.3.3 Data Sheets

see section 3.2.7 in [1].

Main Points:

- The specification they give us will run on many input voltage (power supply)
  - If we are operating on different power supply, we have to interpolate



Switching Characteristics

over operating free-air temperature range, C_L = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC}	T_A = 25°C MIN	TYP	MAX	SN54HC04 MIN	MAX	SN74HC04 MIN	MAX	UNIT
t_{pd}	A	Y	2 V		45	95		125		120	ns
			4.5 V		9	19		29		24	
			6 V		8	16		25		20	
t_t		Y	2 V		38	75		110		95	ns
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

Figure 3.7: Example of Specification for some data sheet

- Also we have to pay attention when using some IC to drive many outputs, what is the maximum current that can tolerate, so we don't mess up the IC

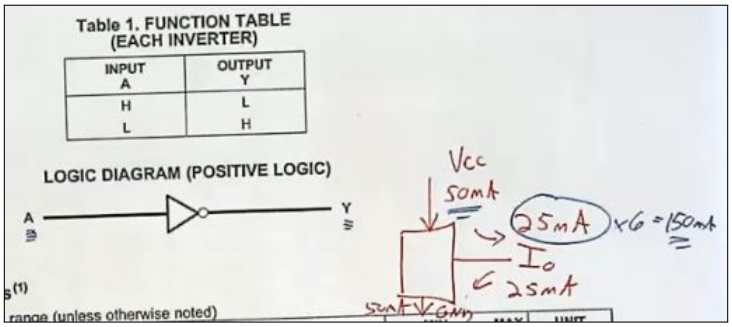


Table 1. FUNCTION TABLE (EACH INVERTER)

INPUT A	OUTPUT Y
H	L
L	H

LOGIC DIAGRAM (POSITIVE LOGIC)

Figure 3.8: Maximum Current

3.4 Logic Families

We have mentioned previously that as designers, we need to make sure that Tx sent the correct level of voltages (high or low) to the Rx, so that the circuit can read correct values of 0 and 1.

Now as circuit going to grow and system going to complexify, we can't just keep track voltages level and do circuit analysis to make sure goes right.
Instead we use the concept of ***logic families***.

Logic families are set of circuit and devices that are designed to work together. They are architecture to set the transistors in a way to build the logic gates, memories,$\cdots$ and any other type of digital system.

They are 2 types of logic families:

1. TTL (transistor transistor logic)
    - The problem in these that they consume allot of power
2. CMOS (complementary metal oxide semiconductor)
    - This technology is used on almost all of our modern devices (cell phones,$\cdots$)

Inside each category (TTL and CMOS) exist many type of families (subclasses). Before diving into the details of each of them, we will explain the concept of Fan-In and Fan-Out

3.4.1 Fan-In and Fan-Out

Fan-In and Fan-Out are the maximum number of input we can have (Fan-In) and maximum number of outputs (Fan-Out) we can drive on a single pin.

- Fan-Out: if we are using CMOS, the $I_o > I_i$, which means we can drive like 2 outputs using single pin, since we have large output current and require small input current
- However, we can't keep driving gates using pin output to infinity, because it will make the square wave having a small rise time, and hence we enter the uncertainty region in which we don't know the value of the pulse exactly if 0 or 1

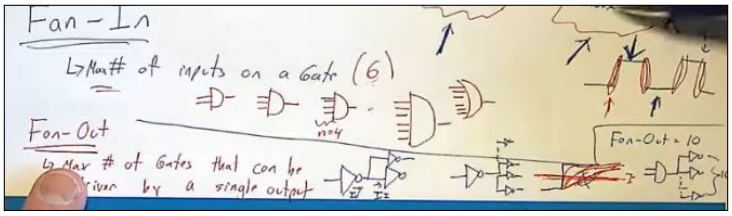


Figure 3.9: Fan-In and Fan-Out

3.4.2 CMOS key points

- CMOS is a type of a method to connect a transistor to form a switch to implement gate
- Transistor components: Drain, gate then source.

  Current flow from drain to source
- There are 2 type of CMOS: P type and N type
- N type: we take the silicon structure (or other semiconductor structure) and doping it with impurities whihc have abundance of electron (this means we have an excess of electrons)
- P type opposite of N type
- C stands for complementary: that is we have 2 type of transistor (P-MOS and N-MOS) to create a switch and conduct the current, to form a specific gate

Chapter 4

Programmable Logic Device

Source: Chapter 4 in [2].

Chapter 5

Boolean Algebra and Reduction Techniques

Source: Chapter 5 from [2].

5.1 Outline

1. Combinational Logic
2. Boolean Algebra Laws and Rules
3. Simplification of Combinational Logic Circuits Using Boolean Algebra
4. Using Quartus® II to Determine Simplified Equations
5. De Morgan's Theorem
6. Entering a Truth Table in VHDL Using a Vector Signal The Universal Capability of NAND and NOR Gates
7. AND–OR–INVERT Gates for Implementing Sum-of-Products Expressions
8. Karnaugh Mapping
9. System Design Applications

5.2 Combinational Logic

Design some useful system using logic gates.

Example: An automobile warning buzzer where the criteria for the activation of the warning buzzer is as follows: The buzzer activates if the headlights are on *and* the driver's door is opened *or* if the key is in the ignition and the door is opened.

The digital implementation is shown in Figure 5.1

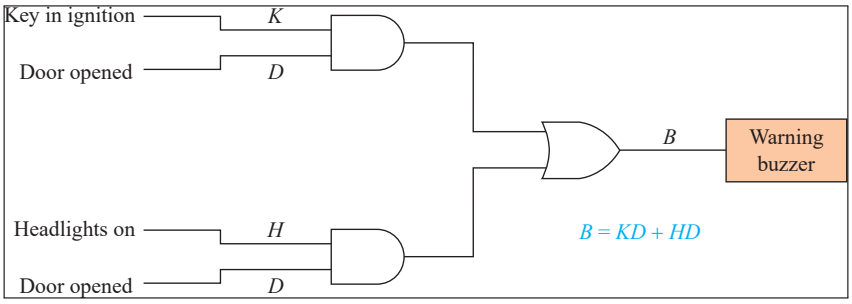


Figure 5.1: Buzzer Logic gates [2]

5.2.1 Gate Reduction

Notice that in the system of Figure 5.1 the variable D is common between the 2 part, so we can factor out the variable D, and hence we obtain the new system shown in Figure 5.2

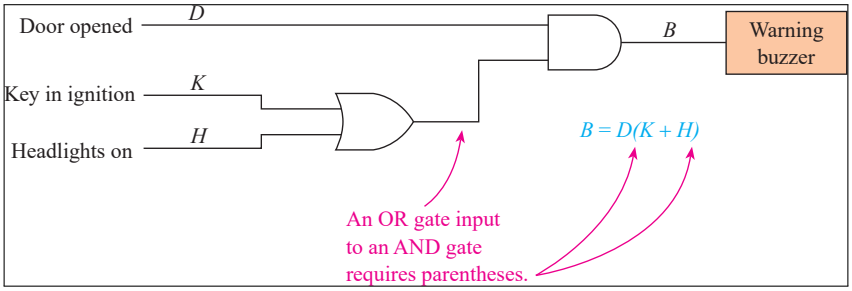


Figure 5.2: Buzzer Logic gates: reduced version [2]

5.3 Boolean Algebra Rules

These boolean algebra are theorems with multiple variables.

Mainly we have 3 rules:

1. Commutative law: structuring variables

   - Addition: $A + B = B + A$
   - Multiplication: $A\,B = B\,A$

   It is widely use in routing algorithm, and if we want to group the input in some particular structure.

   Commutativ law

   :

   There is a picture from [1] in his videos 4.1 c)

2. Associative law: grouping variable won't affect the operation logic

   - Addition: $A + (B + C) = (A + B) + C$
   - Multiplication: $A\,(B\,C) = (A\,B)\,C$

   It is useful for example if we have some Fan-in constraints: if we have for example $A + B + C + D$, this is a 4 input OR gate.

   If the Fan-in constraint is limited to 2 input, we can say that $A + B + C + D = (A + B) + (C + D)$, so instead of 1 OR gate with 4 inputs, we can use 2 OR gate with 2 inputs and solve the Fan-in constraints

3. Distributive law: $A\,(B + C) = A\,B + A\,C$ and $(A + B)\,(C + D) = A\,C + A\,D + B\,C + B\,D$

4. Absorption [1]:

$$\begin{array}{rcl} A + A\,.\,B & = & A \\ A\,.\,(A + B) & = & A \end{array} \tag{5.1}$$

5. Uniting [1]:

$$\begin{array}{rcl} A\,.\,B + A.\overline{B} & = & A \\ (A + B).(A + \overline{B}) & = & A \end{array} \tag{5.2}$$

5.3.1 Circuit Equivalent Version: OR and AND gates

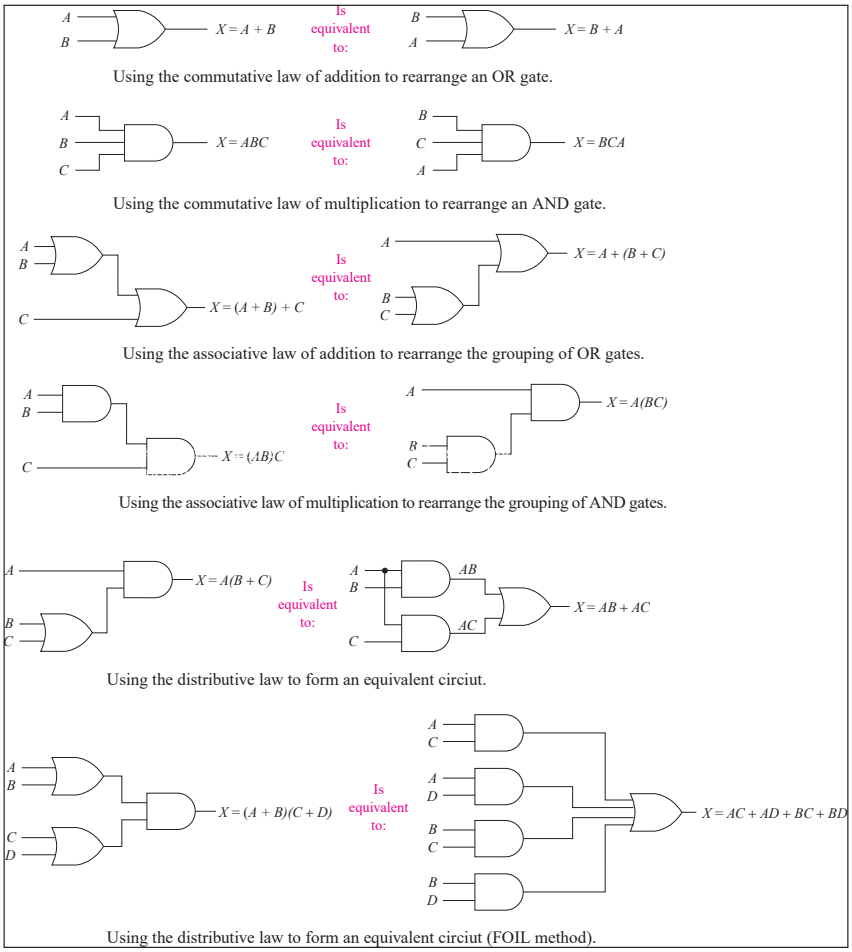


Figure 5.3: Boolean Algebra Applied to OR and AND gates [2]

5.3.2 Boolean Rules Applied to Logic Operation

In addition to the 3 major rules introduced before (commutative,associative and distributive), there are also 10 rules which we can use them when simplifying the combinational logic design.

Note: see section 5-2 page 163 in [2] for more details of these rules.

Rules:

1. Anything ANDed with a 0 is equal to 0 ($A \,.\, 0 = 0$).
2. Anything ANDed with a 1 is equal to itself $\rightarrow$ $A \,.\, 1 = A$
3. Anything ORed with a 0 is equal to itself $\rightarrow$ $A + 0 = A$
4. $\cdots$

5.3.3 Summary of Boolean Algebra Rule

TABLE 5–2 Boolean Laws and Rules for the Reduction of Combinational Logic Circuits

Laws

1		$A + B = B + A$
		$AB = BA$
2		$A + (B + C) = (A + B) + C$
		$A(BC) = (AB)C$
3		$A(B + C) = AB + AC$
		$(A + B)(C + D) = AC + AD + BC + BD$

Rules

1		$A \cdot 0 = 0$
2		$A \cdot 1 = A$
3		$A + 0 = A$
4		$A + 1 = 1$
5		$A \; A = A$
6		$A + A = A$
7		$A \; \overline{A} = 0$
8		$A + \overline{A} = 1$
9		$\overline{\overline{A}} = A$
10	(a)	$A + \overline{A}B = A + B$
	(b)	$\overline{A} + AB = \overline{A} + B$

Figure 5.4: Summary of Boolean Algebra [2]

5.4 Simplifying Combinational Logic Circuits

Now we can apply the rules of previous section when simplifying designs.

- We have some initial design shown in Figure 5.5

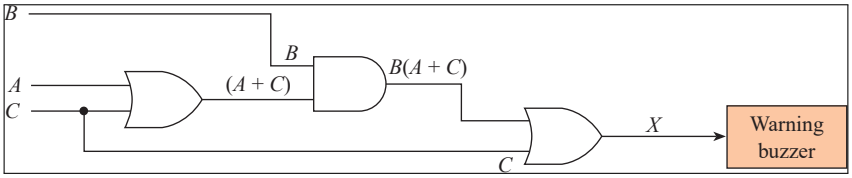


Figure 5.5: Summary of Boolean Algebra [2]

- We start by having the boolean equation

$$X = B\,(A + C\,) + C \tag{5.3}$$

- Applying the rules from section 5.3 we get:

$$X = A\,B + C \tag{5.4}$$

where the associated circuitry is shown in Figure 5.6

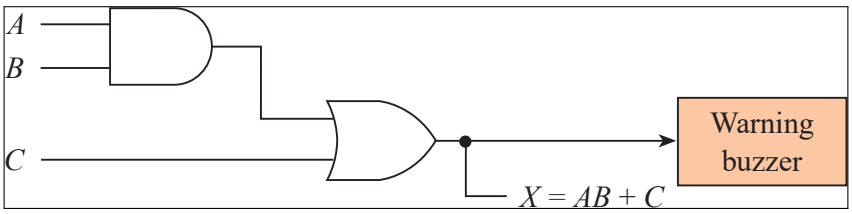


Figure 5.6: Simplified Design Circuitry [2]

For Practice: See example $5-7 \rightarrow 5-9$.

n't Care:

Example 5-9 is important because in the final design, the input pin C appear as a don't care condition.

To relisten video sec 0504.

To be implemented using Verilog later.

5.5 De Morgan Theorem

In previous sections, we didn't talk about the simplification of NAND and NOR, and this is because we need a theorem in order to manipulate these gates, which is known in De Morgan theorem, shown in Equation 5.5

$$\begin{aligned} \overline{A \,.\, B} &= \bar{A} + \bar{B} \\ \overline{A + B} &= \bar{A} \,.\, \bar{B} \end{aligned} \tag{5.5}$$

Using Equation 5.5, we can repeat the same simplification concepts in section 5.4.

For Practice: See example 5-12 and beyond in the book.

Practice De Morgan:

See video sec0505.

De-Morgan theorem allow us to implement OR operation using NAND, and this is important because NAND gate can be implemented using CMOS circuit [1].

De-Morgan Second theorem

- to review this detail. This is mentioned by LaMeres.
- Review Practical Application later in video 4.1c

5.5.1 Bubble Pushing and Application to Microprocessor

De Morgan theorem has also an equivalent circuit application as shown in Figure 5.7

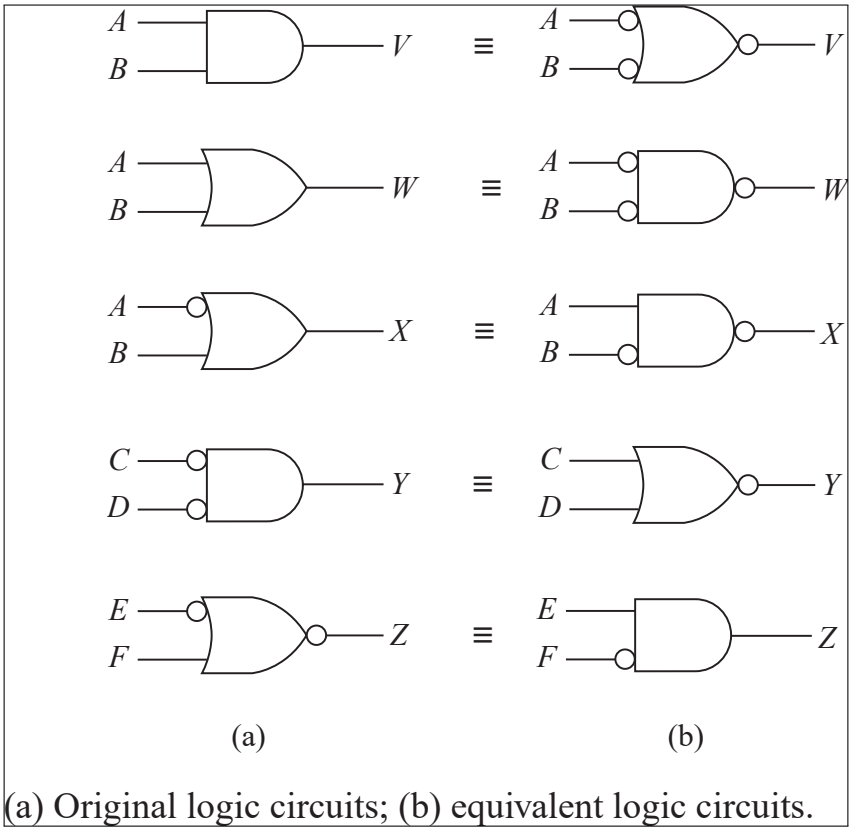


Figure 5.7: Equivalent Circuit using De Morgan Theorem [2]

This type of equivalence is used in application where systems require an active low signal to do some task. An example of such systems is shown in Figure 5.8.

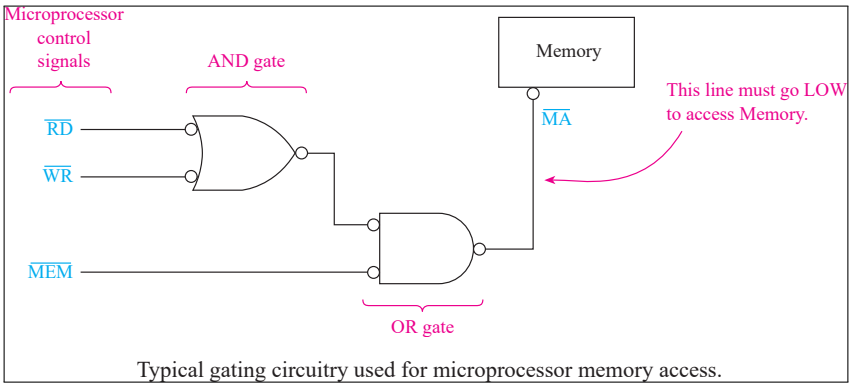


Figure 5.8: Microprocessor: Active Low System [2]

Note: Read in the book to see detailed description of Figure 5.8

5.6 Universality of NAND and NOR

Big conclusion: we can implement any basic gate (AND,OR,$\cdots$) using NAND or NOR gates.

See section 5-7 in [2] for more details.

5.7 Combinational Logic Analysis

source: section 4.2 in [1].

Definition of Logic Analysis: Combinational logic analysis involve that ***given some logic circuit***, we need to find some information about it (like logic expression, truth table,propagation delay,$\cdots$).

5.7.1 Example

Given: Suppose we have the circuit shown in Figure 5.9.

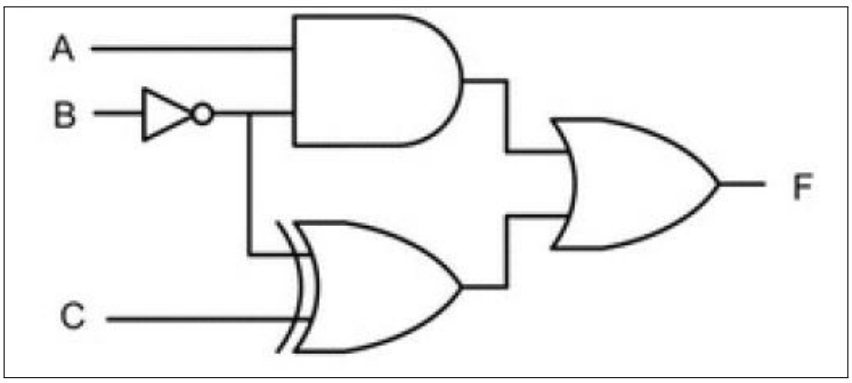


Figure 5.9: Some Given Combinational Logic Circuit [1]

Type of Analysis to be done: The key point in all the type of analysis we do are the evaluation ***internal nodes***.

1. Deriving logic expression:

   We process the circuit from left → right, and evaluate the internal nodes as shown in Figure 5.10.

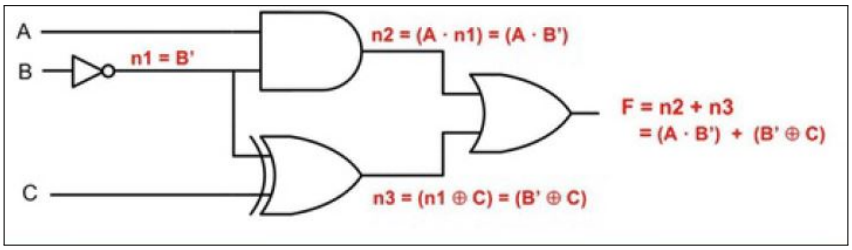


Figure 5.10: Logic Expression Derivation [1]

2. Deriving truth table: shown in Figure 5.11.

A	B	C	n1 = B'	n2 = A·B'	n3 = B'⊕C	F = (A·B') + (B'⊕C)
0	0	0	1	0	1	1
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	1	1
1	0	1	1	1	0	1
1	1	0	0	0	0	0
1	1	1	0	0	1	1

Figure 5.11: Truth table of Figure 5.9 [1]

3. Propagation Delay:

   - This is caused by circuitry of the gates and their response time when changing from $0 \rightarrow 1$ or from $1 \rightarrow 0$
   - We compute the delay over each input and its each path, then take the maximum value as answer

   Example shown in Figure 5.12.

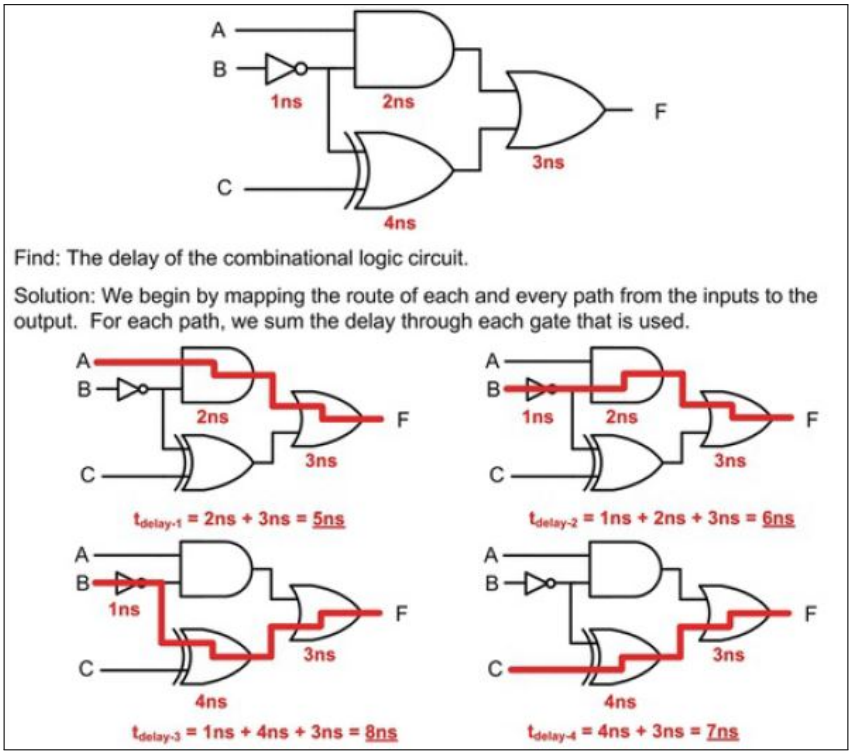


Figure 5.12: Delays over each path for each input [1]

What can be taken from Figure 5.12, is when we do our computations, we need to wait 8 ns before applying any new values for the input. Otherwise, it could be affected by the process before.

5.8 Combinational Logic Synthesis

Synthesis or design, is the word we use when we use when we take some ***functional description of a system*** (like a truth table), and synthesize the logic diagram which can be constructed using gates and other techniques.

5.8.1 Canonical SOP

The word canonical to refer to en expression which no effort has done to simplify and optimize the logic expression (or circuit).
The 1st technique we will use is called SOP, where its topology is shown in Figure 5.13.

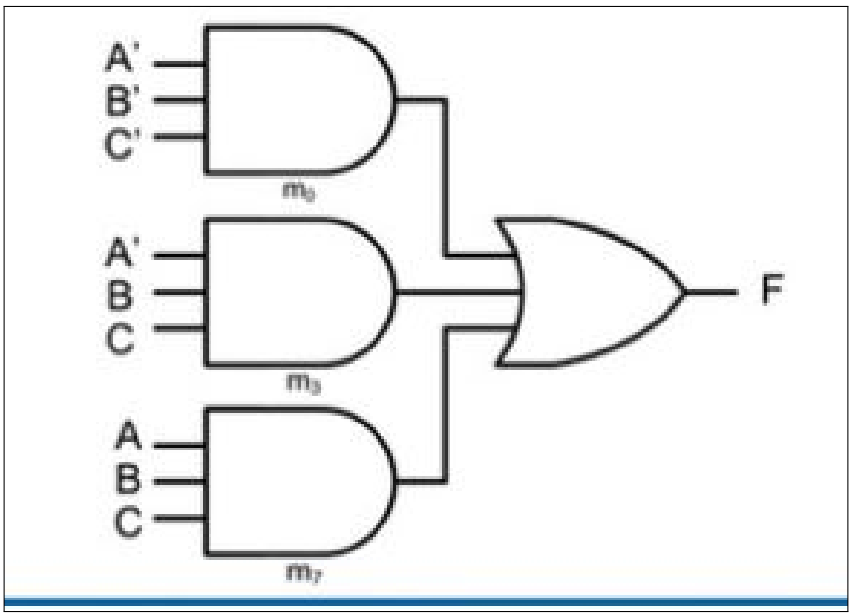


Figure 5.13: SOP Topology: OR gates fed with AND gates [1]

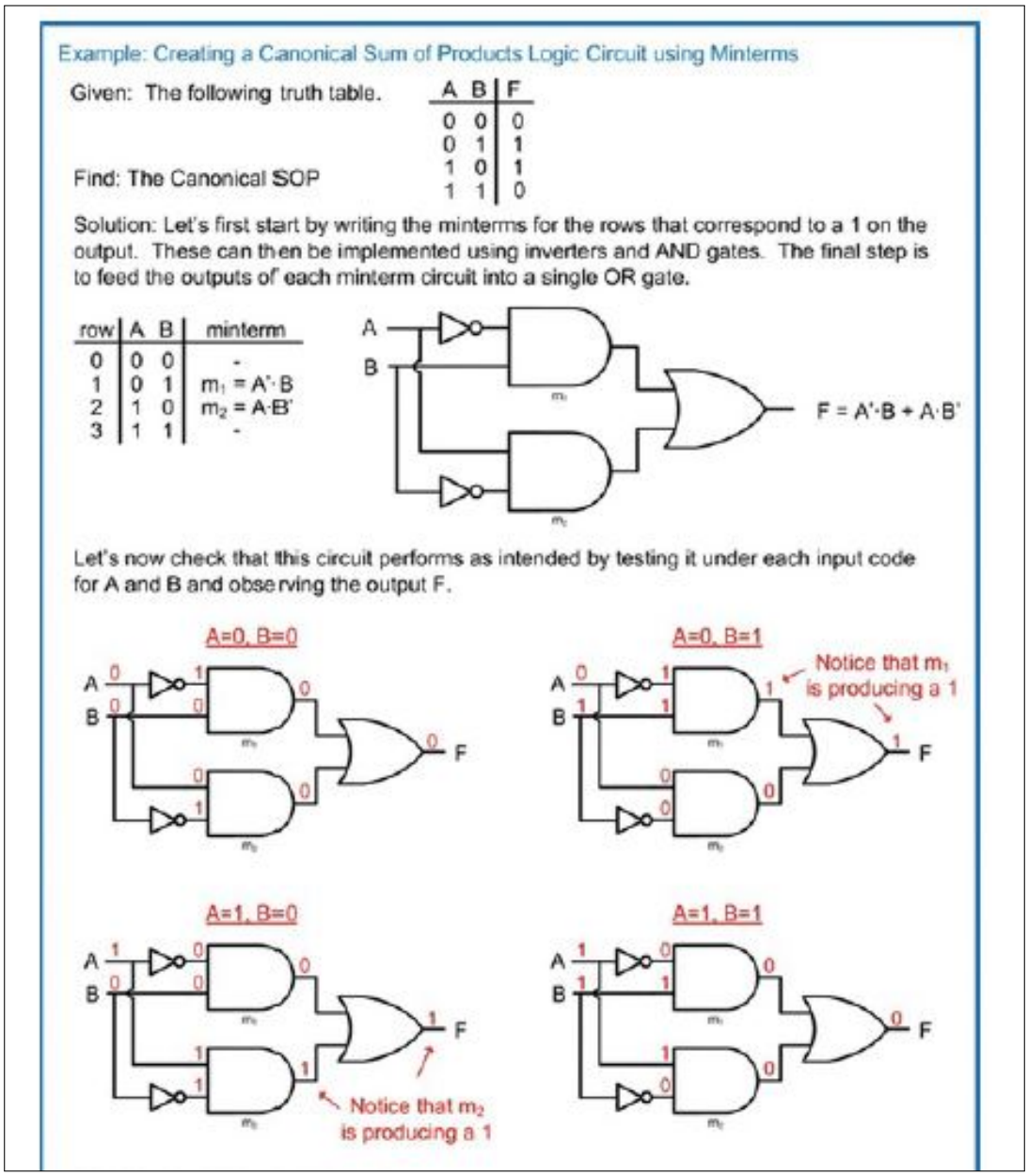
Its logic expression has the form shown in Equation 5.6:

$$X = A\overline{B} + AC + \overline{A}BC \tag{5.6}$$

Each term in Equation 5.6 is called ***minterm***.

5.8.2 XOR Synthesis Example

Suppose for example we are given the truth table of an XOR gate, and we would like to synthetise it using AND and NOT gates. This example is illustrated in



A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

row	A	B	minterm
0	0	0	-
1	0	1	$m_1 = A' \cdot B$
2	1	0	$m_2 = A \cdot B'$
3	1	1	-

Figure 5.14: XOR Gate Synthesise [1]

1. We are given the truth table
2. We add an extra column in the truth table, where it contains the minterm
   - The ***minterm are indexed by the row number***: $m_2 \leftrightarrow$ minterm for row 2
3. We search for ***inputs*** where ***their outputs which gives logic 1***, and these will be the minterms $\rightarrow$ that is row 1 and 2in the case of XOR (Note that we are counting the rows starting from 0)

4. We get the logic expression and we can draw the logic diagram

5.8.3 4 ways

Other then the SOP expression, there are also:

1. Minterm list: for XOR example we can write as $\sum_{A,B}(1, 2)$
    - It is necessary to have the order of the boolean variable correct, so we know the logic values combination which gives output 1 correctly of the minterm
    - Logic Expression
    - Truth table
    - Logic diagram (circuit)

Note:

- To see an example how to get back from a minterm list → truth table, see example 4.17 in [1].

5.9 POS and Maxterm

POS is the 2nd way to represent a logical system. Its form is shown in Equation 5.7.

$$X = (A + \overline{B})(B + C) \tag{5.7}$$

Each term in the POS is called ***maxterm***, which assert a 0 for only 1 input code.

5.9.1 XOR Example using POS

Let's redo the XOR example using a POS approach. The example is shown in Figure 5.15.

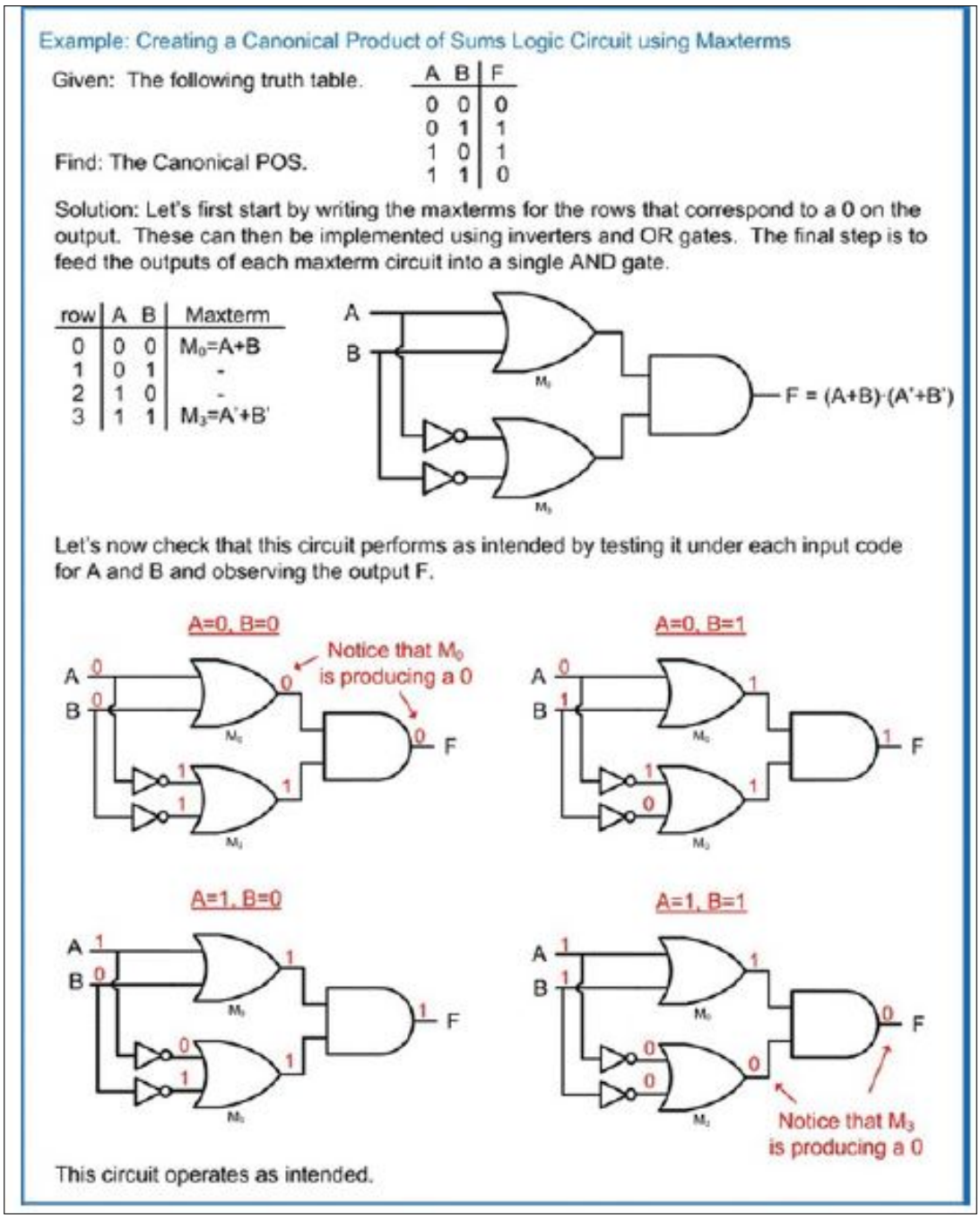


Figure 5.15: XOR Gate Synthesise using POS Approach [1]

1. We look for the rows where output is 0
2. ***Important***: Be aware that in POS approach, we complement the 1 value and not the 0 (unlike SOP approach)
    - In row 3, we have 11 input value, so the corresponding maxterm is $M_3 = \overline{A} + \overline{B}$

Also for POS we have maxterm list (same concept as minterm list).

Note: To go back from a maxterm list → truth table, see example 4.20 in [1].

5.10 Logic Minimization

After introducing the SOP and POS, we now move on to minimizing these expressions. This minimization can be done using 2 methods:

- Boolean Algebra Rules and theorems
- Karnaugh Mapping
    - It will be explained in its own section

5.10.1 Boolean Algebra

An example is shown in Figure 5.16

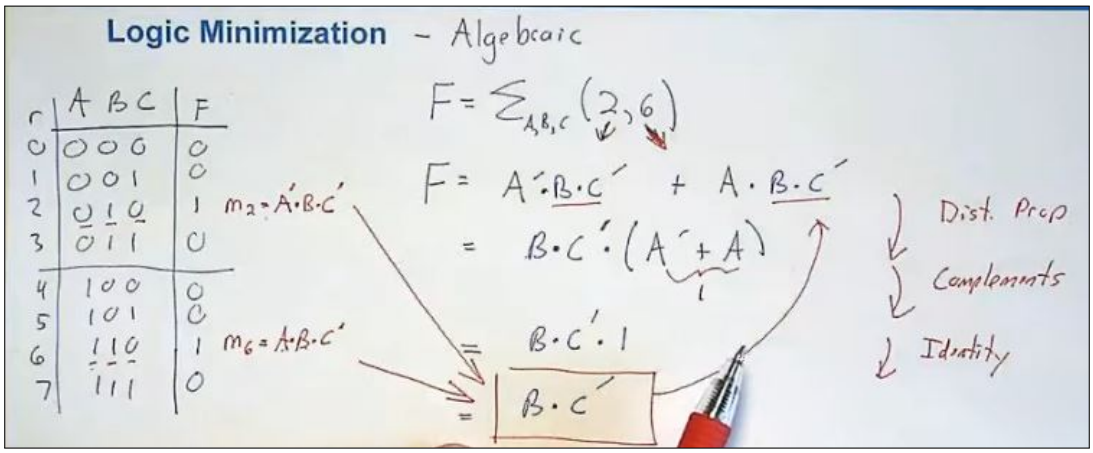


Figure 5.16: Minimization using Boolean Algebra [1]

5.11 Karnaugh Mapping

Karnaugh Mapping is a method where we map the truth table into a gird of cells, then do simplification. Before start applying this method, we first see how to draw a K-map for 2,3 and 4 variables.

5.11.1 Drawing K-map Cells

Each time we need to draw Karnaugh map, we need to have the truth table with us.

2 Input Karnaugh Map:

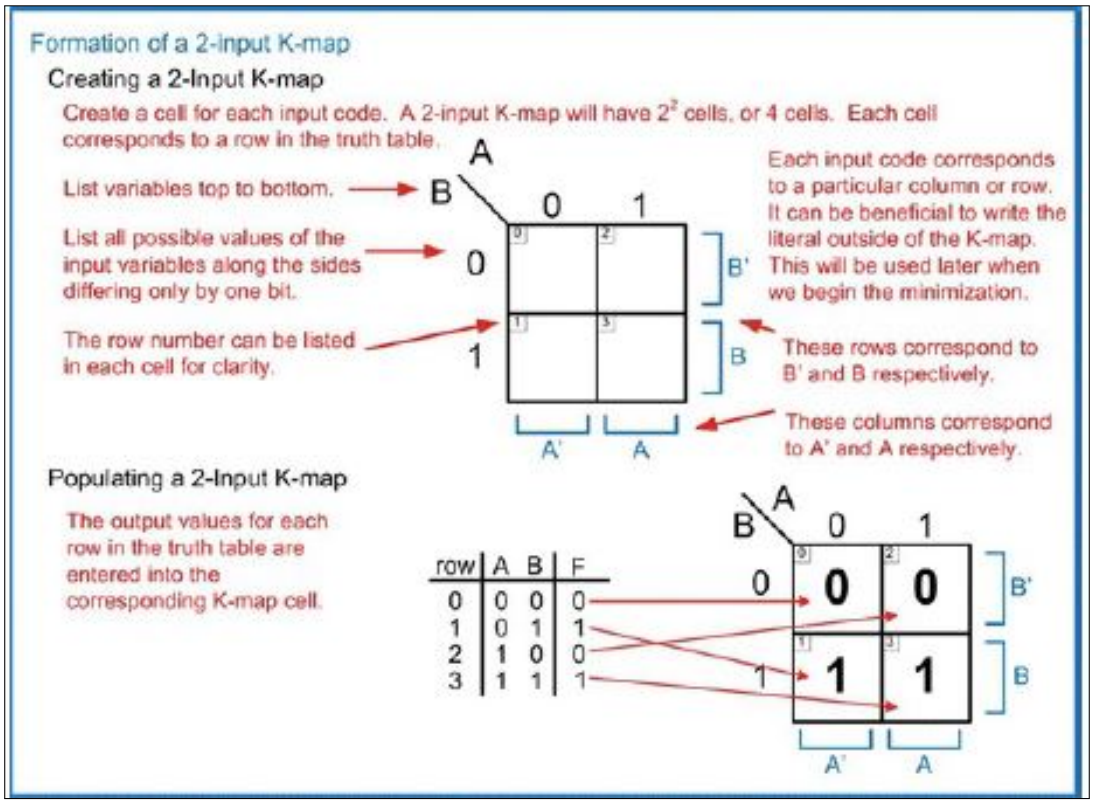


Figure 5.17: 2 Input Karnaugh Map [1]

- Notice that for each row number in the truth table, we label the cell number in the k-map
- Then also we label ***cell variable value***: cell 0 and 1 is where A is complemented.

  This labeling will become very important when using k-map for simplifying SOP or POS expressions.

3 Input Karnaugh Map:
For 3 input, we have some additional rules:

- Any 2 adjacent cells can't differ by more then 1 bit
- Neighbors and adjacency: We can only have up and down, left right adjacency
  - There is no diagonal neighbors

This mean, we will use the order: 00,01,11,10 and not the traditional one: 00,01,10,11. This is because from 01 $\rightarrow$ 10, 2 bits have changed.

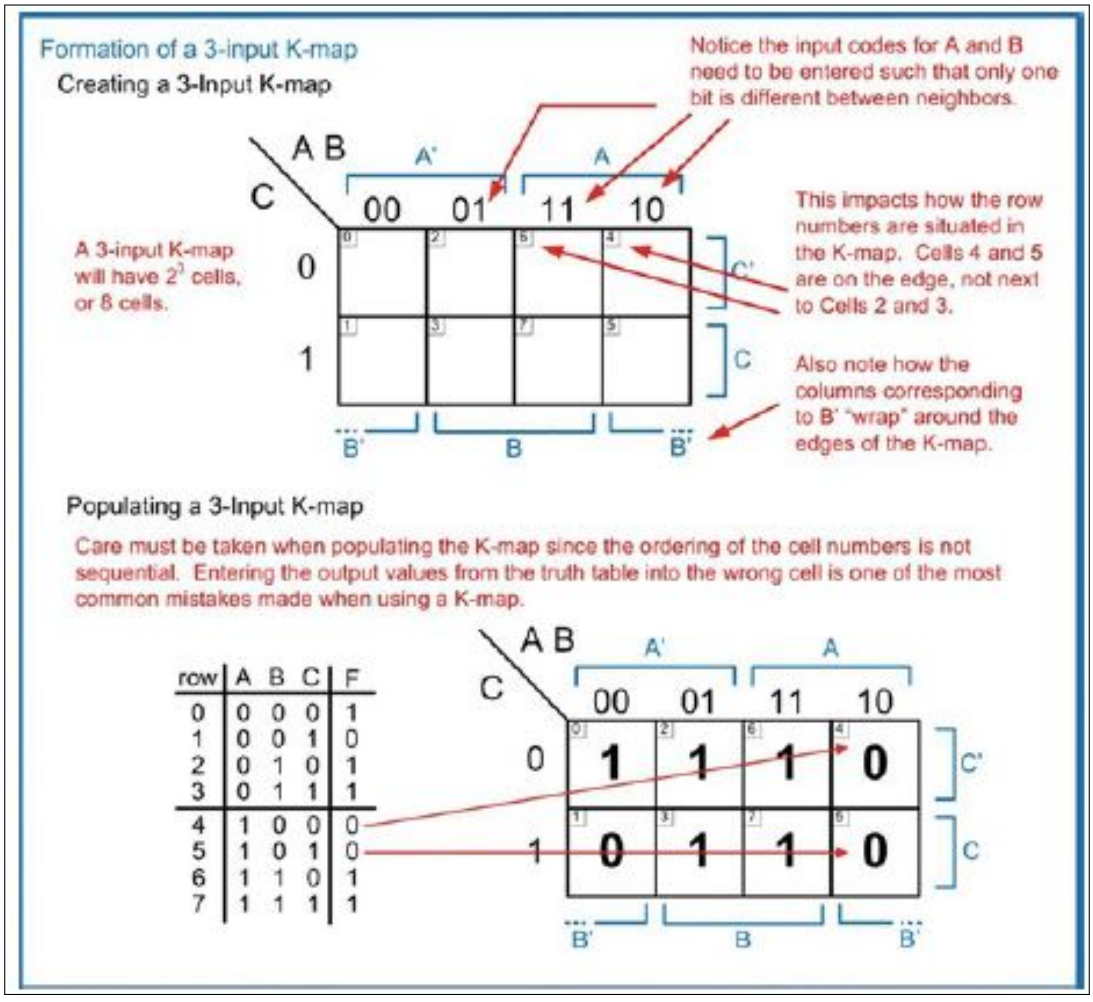


Figure 5.18: 3 Input Karnaugh Map [1]

4 input Karnaugh Map:

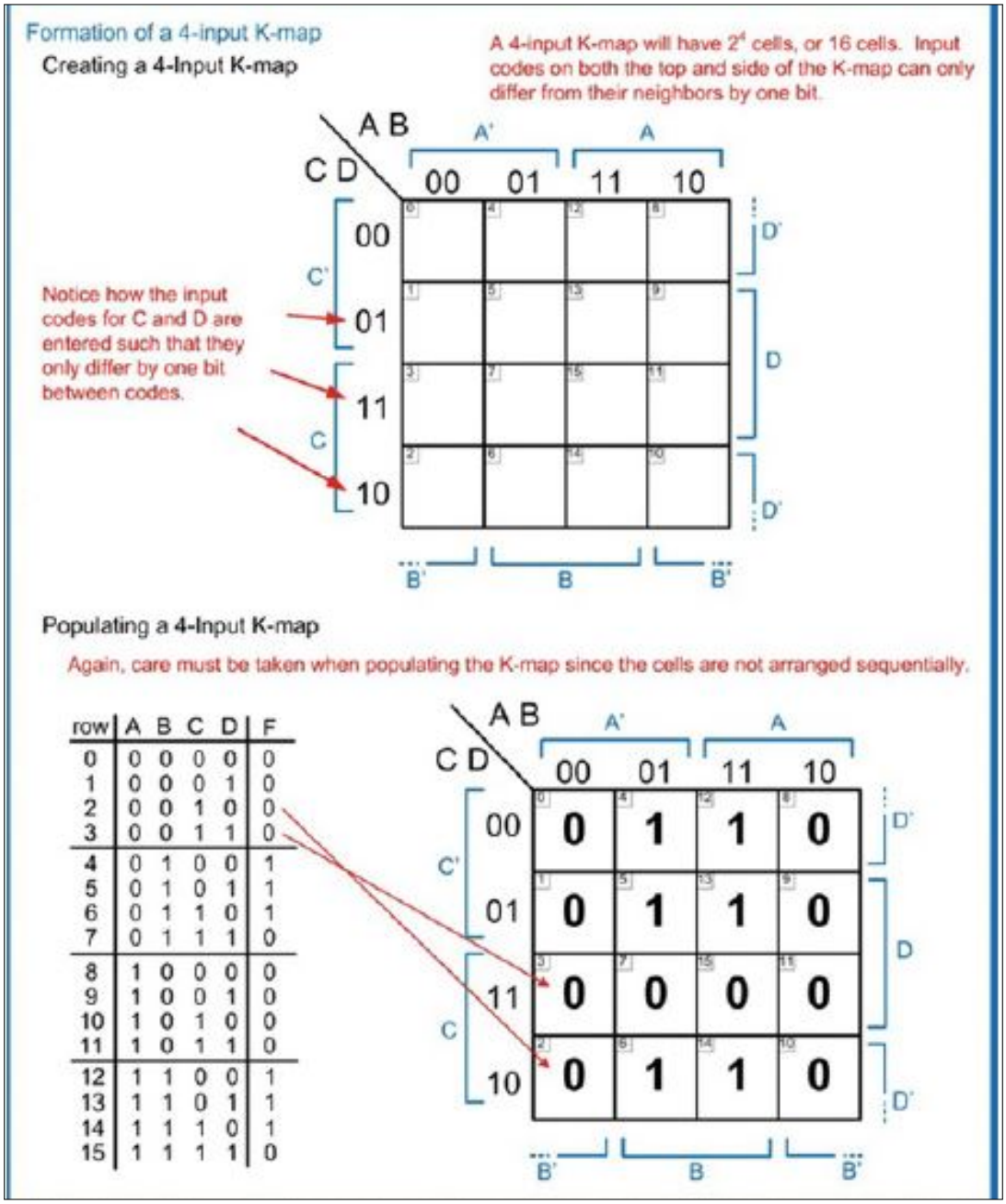


Figure 5.19: 4 Input Karnaugh Map [1]

5.11.2 Minimizing Canonical SOP

After learning how to form a K-map, we will use to minimize a canonical SOP expression. We will do it also for 2,3 and 4 input expression.

Rules and Steps:

1. Circle the group of 1
    - Should contains the largest number of 1
    - Only we circle a group of power of 2 $\rightarrow 2, 4, 8, 16$
        - We can't circle a group which contains three 1 for example
    - Circles can't encompass each other $\leftrightarrow$ no cricle inside a cricle
        - However, we can have circle with common 1
2. Create a SOP term from each circle
    - If variable value is 1, don't complement it
    - If variable value is 0, complement it
    - If variable have both value, we leave it

Examples:
We start first by 2 input case

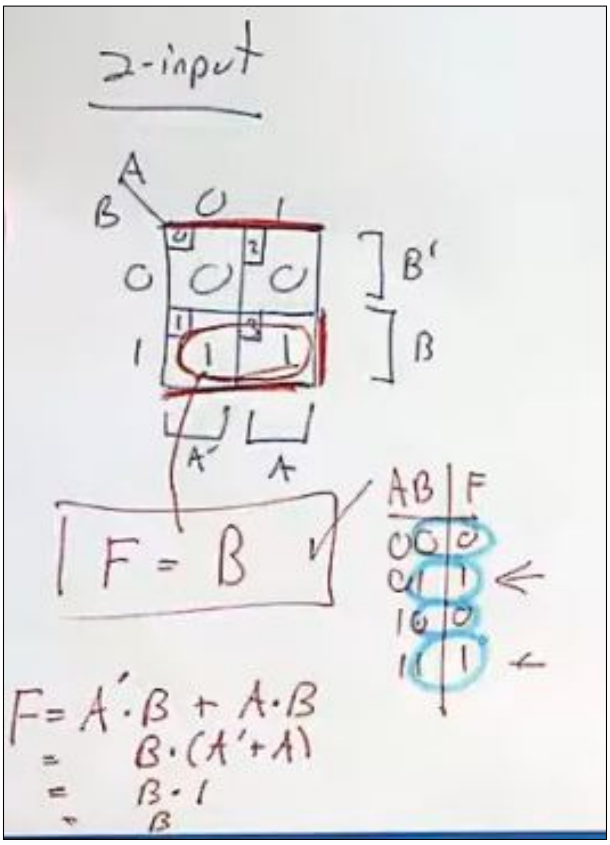


Figure 5.20: 2 Input Karnaugh Map Simplification [1]

- Since the circle contains both A value, we leave it
- For B variable: we only have 1, so we take and don't complement it since we are doing SOP
- Notice that if we have SOP expression, it would take more logic gate

3 and 4 Input Case:

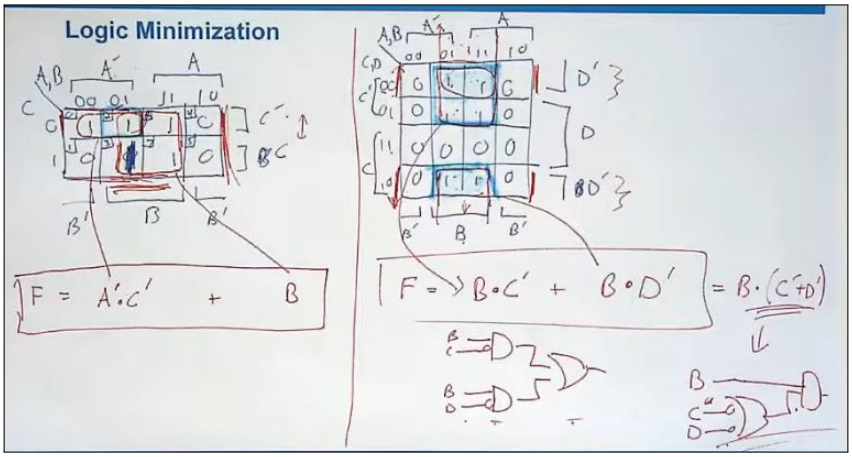


Figure 5.21: 3 and 4 Input Karnaugh Map Simplification [1]

- In 3 input case
  - We have 2 group of circles, and they have common 1
  - Labeling is important since it helps us to write the simplified logic expression
  - ***Important:*** the number of minterms is equal to the number of circles ↔ for each circle, we have simplified minterm
  - In the 1st group where we have two 1: B have 2 values so we dont' include it
- In 4 input case
  - It shows us that we could have group at the edges ↔ which wrap around

5.11.3 Minimizing Canonical POS

In POS expressions, we do exactly the same thing as we do with SOP, but with ***duality***. That is:

- We circle group of 0 instead of 1
- The variables which have 0 value we leave them uncomplemented

An example is shown in Figure 5.22, for 3 and 4 input variable case.

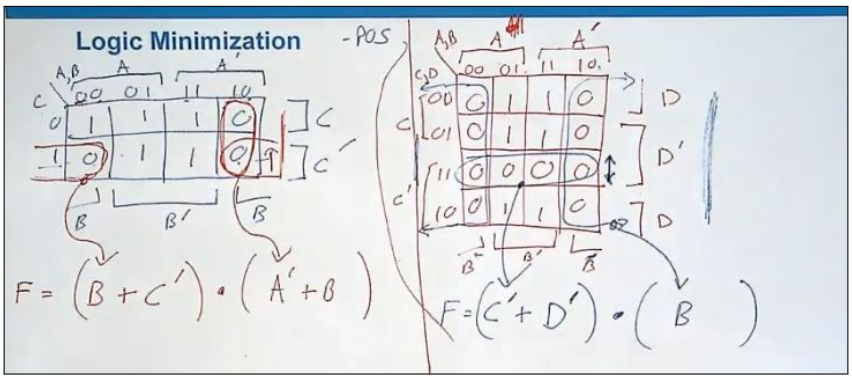


Figure 5.22: 3 and 4 Input Karnaugh Map Simplification for POS expressions [1]

5.11.4 Special Case:Minimal SOP and POS

Now we have learned to minimize SOP and POS expressions, a question can be asked: what is the minimum level of minimization we can do?

An example is shown in Figure 5.23.

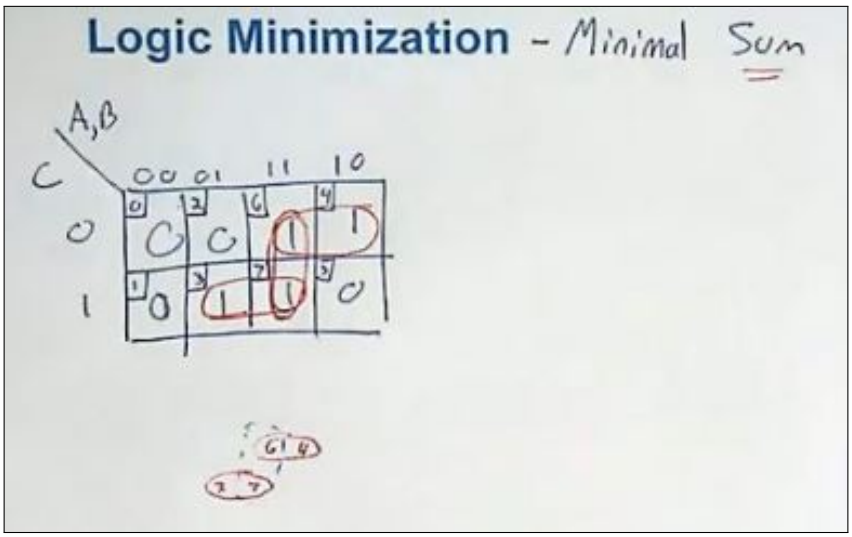


Figure 5.23: Minimal Sum expression [1]

We can see that we have 3 group of circles, but does the vertical circle is necessary ? actually no, it wasn't.

If we want to identify the ***non redundant circles***, we can follow the procedure shown in Figure 5.24

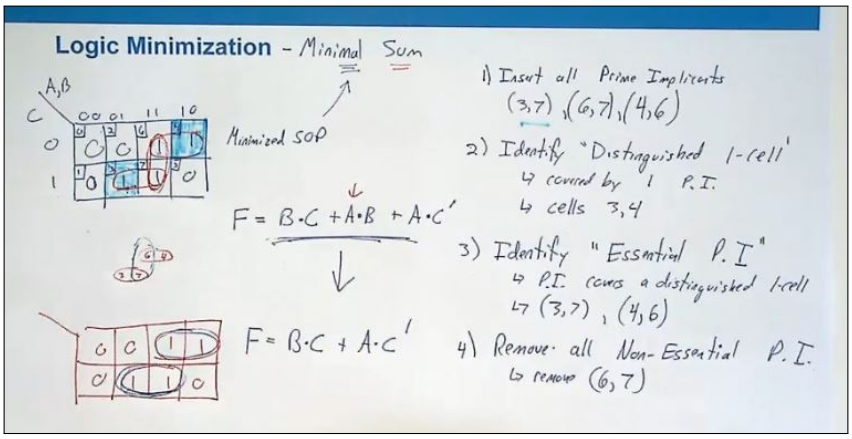


Figure 5.24: Minimal Sum expression [1]

Key ideas:

- The blue cells are called ***distinguishable cells*** (cells covered by 1 prime implant only)
- Any circle (prime implant) which doesn't cover these distinguishable cells is redundant

We can see that the amount of logic in the 2nd logic expression is less than the 1st one.

5.11.5 Don't Care Conditions

In real systems, we could have some situations where an input variable combination doesn't happen in real life. In this case, we put X as output in the truth table.

When we minimize the logic expression using K-map for example, we put X in the cells, and we ***take them in our advantage to form the largest circles possible***, as shown in

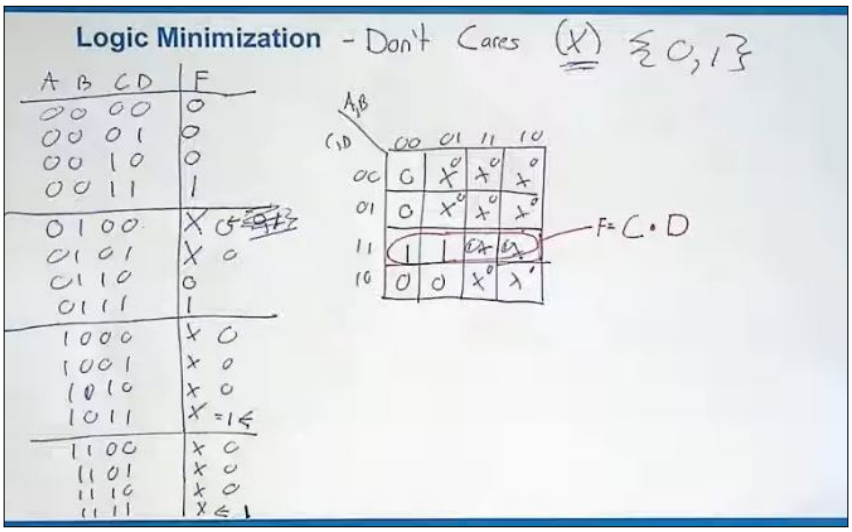


Figure 5.25: Dont' care condition [1]

5.12 SOP and POS

Most boolean reduction technique ends up to a 2 form:

1. POS (Product of Sum)

$$X = (A + \overline{B})(B + C) \tag{5.8}$$

2. SOP (Sum of Product)

$$X = A\overline{B} + AC + \overline{A}BC \tag{5.9}$$

Usually SOP expression are easier to implement their truth table. However POS expression have fewer logic gates when sketching.

Example: See page 202.

$$X = \overline{A\overline{B} + \overline{C}D} \tag{5.10}$$

Equation 5.10 can be simplified using De Morgan theorem twice, where we obtain the following:

$$\begin{aligned} X &= (\overline{A} + B)(C + \overline{D}) \leftarrow \text{POS} \\ X &= \overline{A}C + \overline{A}\overline{D} + BC\,B\overline{D} \leftarrow \text{SOP} \end{aligned} \tag{5.11}$$

5.13 Karnaugh Mapping

Karnaugh mapping is a powerful technique for boolean reduction, which can work from $2 \rightarrow 4$ variables maximum.

Steps for Karnaugh mapping:

1. Transform the Boolean equation to be reduced into an SOP expression.
2. Fill in the appropriate cells of the K-map (we fill the 1 values).

   Note: Method to name Karnaugh cells is shown in Figure 5.26. We will adopt this method of naming the cells.

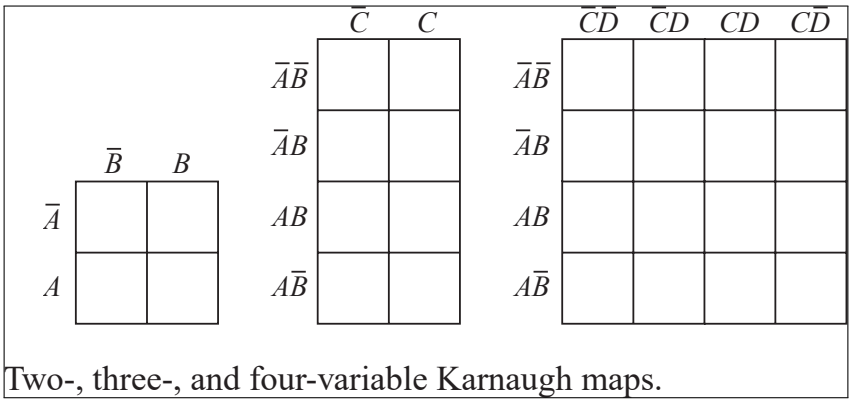


Figure 5.26: Naming Method for Karnaugh Cells [2]

3. *The important step*: Encircle adjacent cells in groups of two, four, or eight. (The more adjacent cells encircled, the sipler the final equation is; adjacent means a side is touching, not diagonal.)
4. Find each term of the final SOP equation by determining which variables remain constant within each circle.

5.13.1 Example

Consider the following equation:

$$X = \overline{A}(\overline{B}C + \overline{BC})\,\overline{A}B\overline{C} \tag{5.12}$$

First, we transform the equation to an SOP expression:

$$X = \overline{A}\overline{B}C + \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} \tag{5.13}$$

Next step is to fill the Karnaugh cell as shown in Figure 5.27

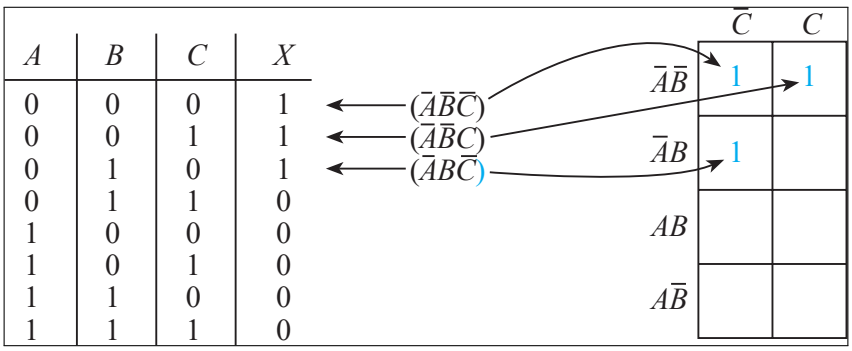


Figure 5.27: Filling Karnaugh Cells [2]

After filling, we circle adjacent cells in groups of two, four, or eight as shown in Figure 5.28

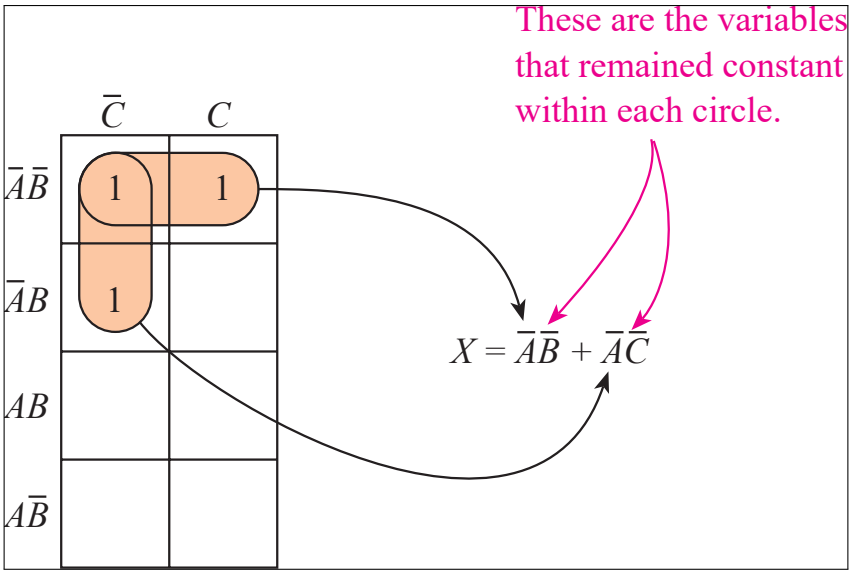


Figure 5.28: Circling Adjacent Cells [2]

To write the simplified boolean equation, we see what boolean variables remain constant in each encircled group:

- Group 1: C change but variables $\overline{A}$ and B remain constant so we keep them
- Group 2: $\overline{A}$ is the same, B change and $\overline{C}$ is the same

See other examples in the book for more practice, and more choices for forming circles and adjacent cells.

Chapter 6

Boolean Algebra Reduction Technique V2

Source: [1].

6.1 Introduction

Some concepts: in order to understand how to design combinational logic circuit, we need to know what are the allowable boolean operation and manipulation we can do.

First, we study boolean algebra in the context of 3 operations: AND, OR and NOT. We will not include XOR and XNOR gate for now.

Before stating the theorem, we need to define some axioms.
Axioms are some very simple facts that we don't need to prove them, unlike a theorem where a prove is needed.

Axioms:

To write them later.
An important axiom is about precedence:

- NOT precedes AND
- AND precedes OR

Some Examples:

- $F = A + \overline{B}$: the NOT must be done first then we do the OR operation.
- $F = \overline{A} \,.\, \overline{B} + C$
  - We do complement first
  - Then AND
  - Finally OR

6.2 Signal Variable Theorem

Now we turn into theorems.

6.2.1 De Morgan Duality

The first theorem we will see is known as *De Morgan Duality*.

Note: there is also a 2^{nd} theorem for De-Morgan know simply as *De Morgan theorem*, whihc is often more used.

De-Morgan duality let us double the number of theorems we have. To illustrate this concept, let's take an example:

- Suppose we have the following boolean expression:

$$A \,.\, 0 \;=\; 0 \tag{6.1}$$

- De-Morgan duality tells us that the Equation 6.1 is true for any change in 1 of the elements in it:
  - $0 \rightarrow 1$
  - $1 \rightarrow 0$
  - $. \rightarrow +$
  - $+ \rightarrow .$
- If we change Equation 6.1 as the following:

$$A \,.\, 1 \;=\; 1 \tag{6.2}$$

  Equation 6.2 remains also true.

Notice that in De-Morgan duality, we don't change the boolean variable A, we only change the operation or the logical values.

6.2.2 Identity Theorem

$$\begin{aligned} A \,+\, 0 &= A \\ A \,.\, 1 &= A \end{aligned} \tag{6.3}$$

The identity theorem can be used to simplify gates. In practice, we have some OR gate where one of the input is always 0, we can replace the OR gate by a wire.

Notice that we use axioms about OR and AND operations in order to prove the identity theorems.

6.2.3 Null Element

$$\begin{aligned} A \,+\, 1 &= 1 \\ A \,.\, 0 &= 0 \end{aligned} \tag{6.4}$$

6.2.4 Idenpotent Theorem

Boolean Algebra with their selfs.

$$\begin{aligned} A + A &= A \\ A \,.\, A &= A \end{aligned} \tag{6.5}$$

6.2.5 Complement Theorem

$$\begin{aligned} A + \overline{A} &= 1 \\ A \,.\, \overline{A} &= 0 \end{aligned} \tag{6.6}$$

Chapter 7

Exclusive-OR and Exclusive-NOR Gates

Source: Chapter 6 in [2]

7.1 Outline

1. The Exclusive-OR Gate
2. The Exclusive-NOR Gate
3. Parity Generator/Checker

7.2 The Exclusive-OR Gate

	Truth Tables for an OR Gate versus an Exclusive-OR Gate				
A	***B***	***X***	***A***	***B***	***X***
0	0	0	0	0	0
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0
(OR)			**(Exclusive-OR)**		

Figure 7.1: XOR Truth Table [2]

- The difference between OR and XOR:
  - The OR gets an high state when we have both A and B high
  - However in XOR we don't (we exclude this case)

Symbol of XOR:

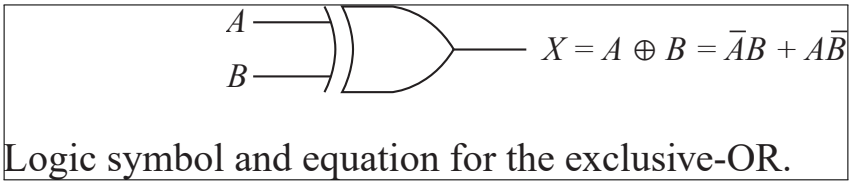


Figure 7.2: XOR Symbol [2]

7.3 The Exclusive-NOR Gate

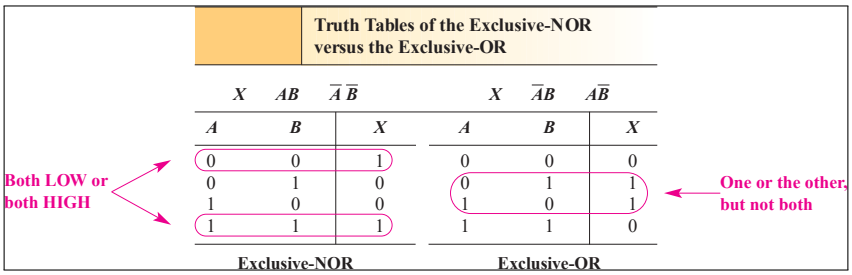


Figure 7.3: X-NOR Truth Table [2]

- The X-NOR is the complement of XOR

Symbol of X-NOR:

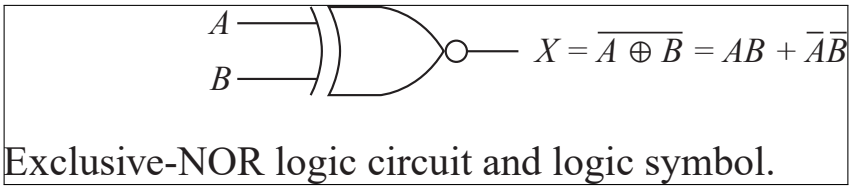


Figure 7.4: X-NOR Symbol [2]

7.4 Parity Generator/Checker

The XOR and X-NOR are useful in parity checker generator circuits.
A parity code is to way to verify if we have some error in digital transmission as shown in Figure 7.5

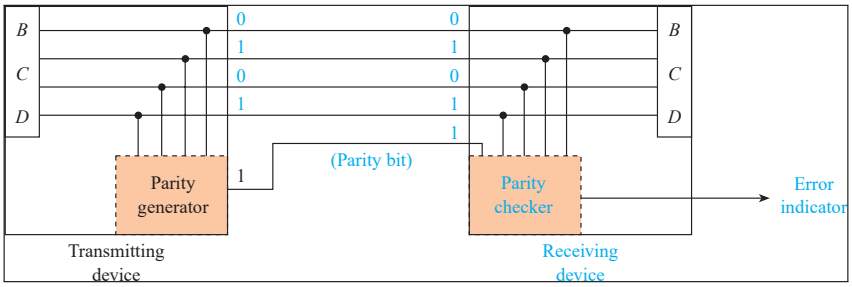


Figure 7.5: Odd parity checker system [2]

To make this generation using some logic system, we can use XOR or X-NOR as shown in Figure 7.6

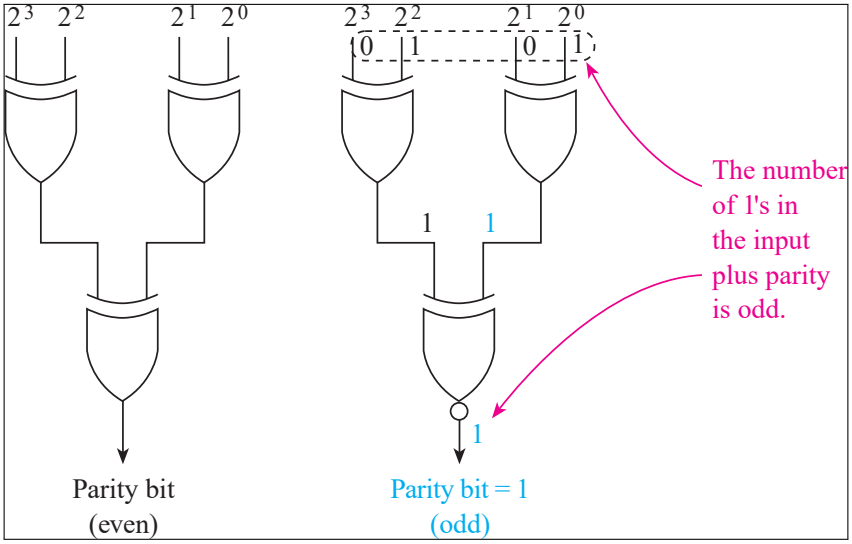


Figure 7.6: 4 Parity system: Even and Odd generation [2]

This system is used to generate parity bits for 4 bits input (even or odd). We can extend this idea to 5 or 8 bit input as shown in Figure 7.7 and Figure 7.8

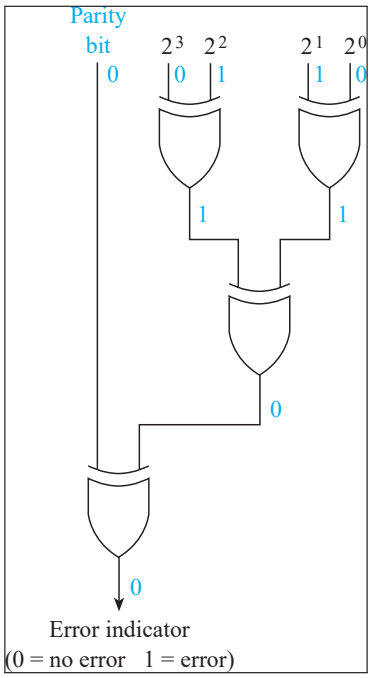


Figure 7.7: 5 Bits Parity system [2]

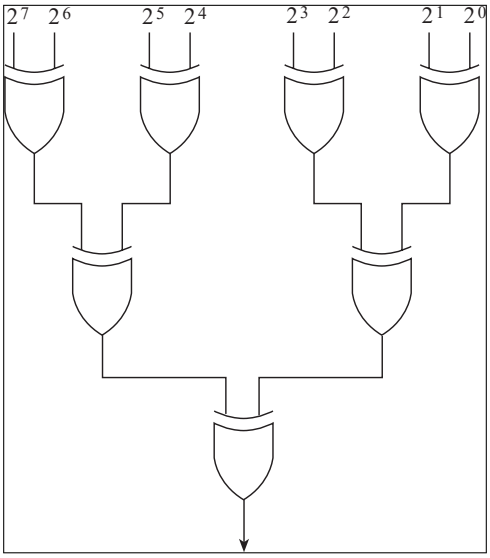


Figure 7.8: 8 Bits Parity system [2]

To redo these examples in Verilog later

Parity Chec
Verilog:

Chapter 8

Code Converters, Multiplexers and Demultiplexers

Source: chapter 6 in [1] and Chapter 8 in [2].

8.1 Outline

1. Comparators
2. VHDL Comparator Using IF-THEN-ELSE
3. Decoding
4. Decoders Implemented in the VHDL Language
5. Encoding
6. Code Converters
7. Multiplexers
8. Demultiplexers
9. System Design Applications
10. FPGA Design Applications Using LPMs

8.2 Decoders

Definition of Decoding: Decoding is the process of converting some code (such as binary, BCD, or hex) into a singular active output representing its numeric value.

Another definition of a decoder is it's a device that decodes some ***encoded information***. We say encoded because usually we want to encode some information and save some space, and the output of the decoder, will decode each input into some non-compact form.
From this definition, we can say that a decoder has number of outputs larger than the number of inputs.
Example: a BCD to decimal decoder shown in Figure 8.1

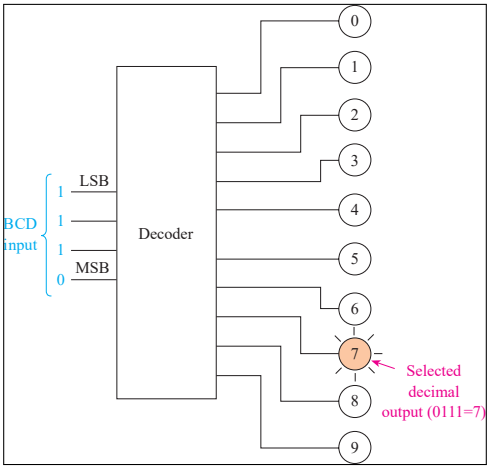


Figure 8.1: BCD to Decimal Decoder [2]

There are various type of decoders, and in next section we see various of them.

8.2.1 One-hot Decoder

The 1st type of decoder we will see is called a ***1 hot decoder***, where for each input code, only 1 output code is asserted.

First we examine a 2-to-4 one hot decoder. Its block diagram is shown in Figure 8.2

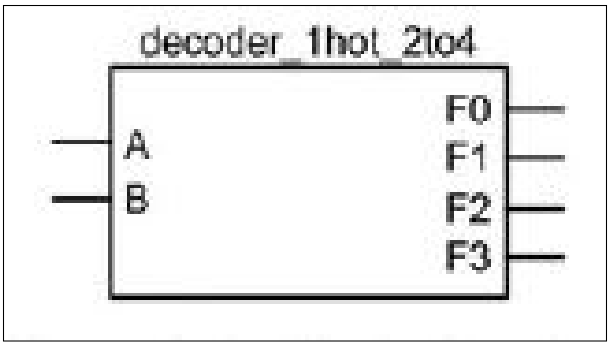


Figure 8.2: 2-to-4 One hot decoder [1]

In order to design the logic circuit, we need to write its truth table, as shown in Figure 8.3.

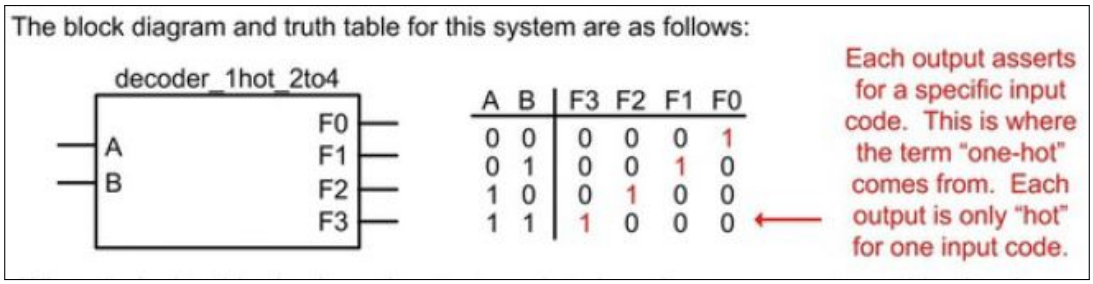


Figure 8.3: 2-to-4 One hot decoder Truth table [1]

Note: Unlike previous chapters, where we dealt with 1 output circuit only, the 2-to-4 one hot decoder has 4 output → for each output, we will have 1 circuitry which activate it.

Notice that from the definition of one hot decoder, only 1 of the output will be activated. To design the circuitry, we take each column alone and apply the technique we learned in chapter 5. The SOP expression and the circuitry is shown in Figure 8.4.

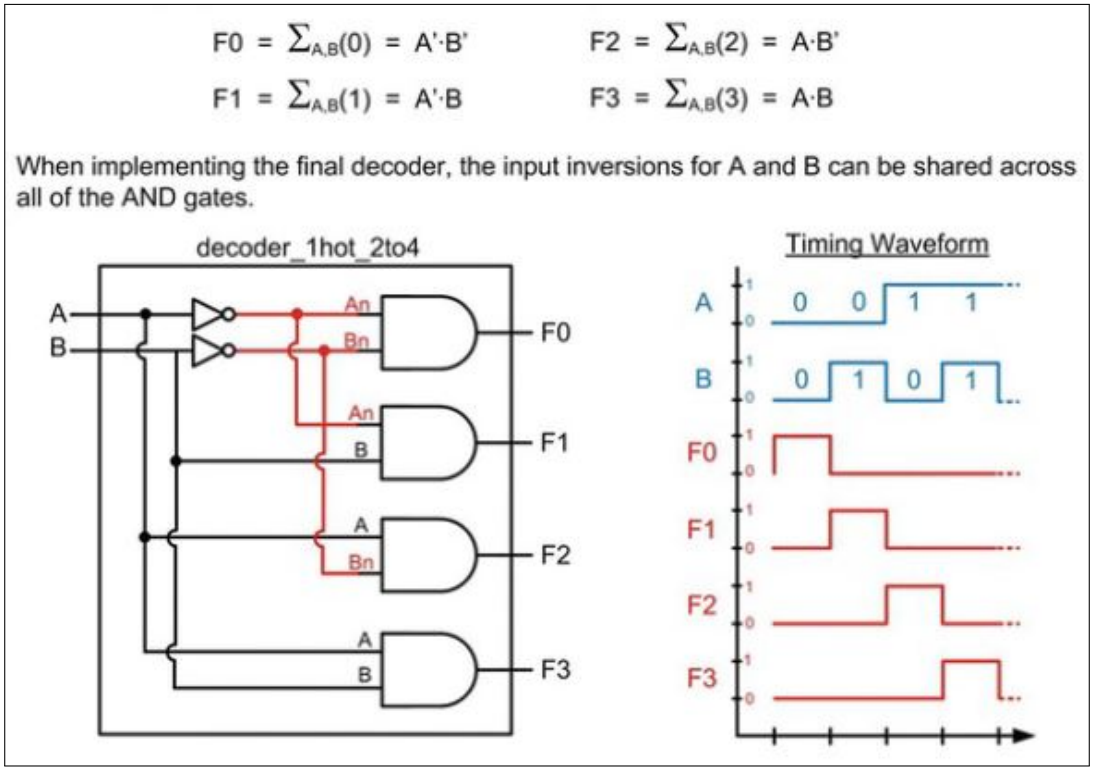


Figure 8.4: 2-to-4 One hot decoder SOP Expressions and Logic Circuit [1]

8.2.2 3-Bit Binary-to-Octal Decoding

Note: See example 6.2 in chapter 6 [1].

Key Design: To design a decoder, it is useful first to make a truth table of all possible input/output combinations.

Truth Tables for an Octal Decoder

(a) Active-HIGH Outputs

Input			Output							
2^2	2^1	2^0	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Note: The selected output goes HIGH.

(b) Active-LOW Outputs

Input			Output							
2^2	2^1	2^0	0	1	2	3	4	5	6	7
0	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0

Note: The selected output goes LOW.

Figure 8.5: Binary to Octale Decoder Truth Table [2]

Before the design is made, we must decide if we want an active-HIGH-level output or an active-LOW-level output to indicate the value selected.

- Most devices used in digital electronics are designed to activate from a LOW-level signal, so most decoder designs use active-LOW outputs,

Logic Gates and Circuit:

First, we concentrate on 1 line of a truth table: for example 001 → 3. The circuitry is shown in Figure 8.6

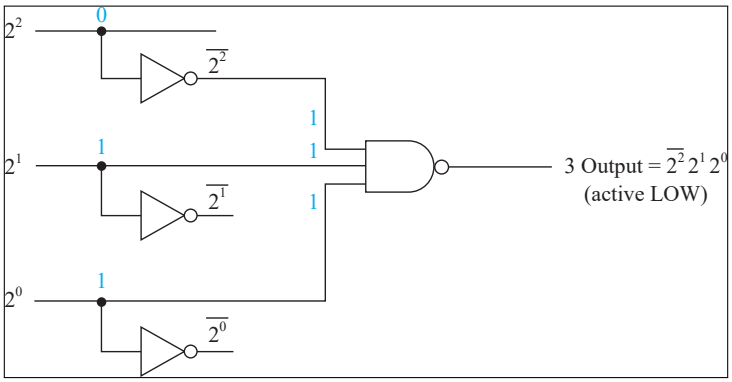


Figure 8.6: Binary to Octale Active Low Decoder Circuit: 001 → 3 [2]

The complete circuitry for other input output combination is shown in Figure 8.7

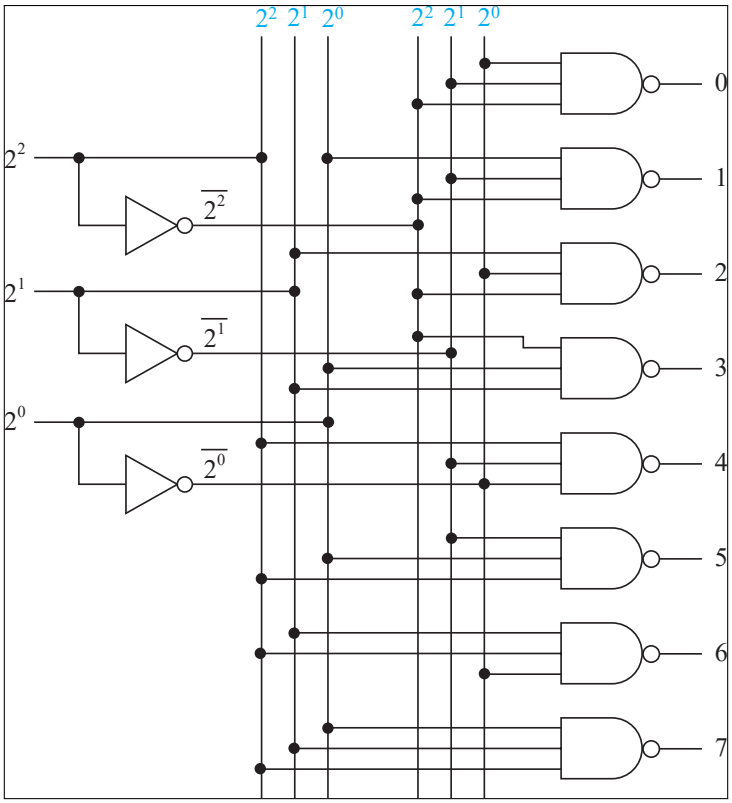
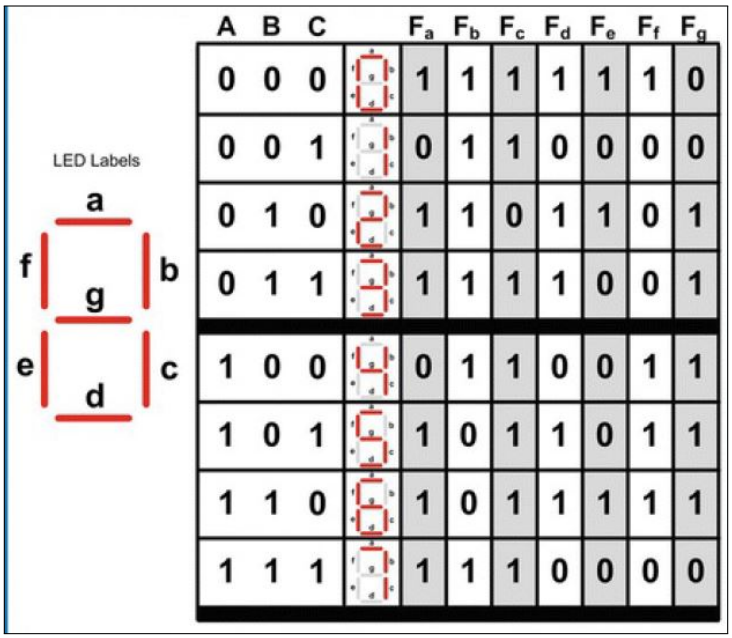


Figure 8.7: Binary to Octale Active Low Decoder Circuit: Complete Circuit [2]

8.2.3 7 segment display

7 segment display can also be designed using decoders. For each binary input, there will be some segments (which are the output) activated. Its truth table is shown in Figure 8.8.



A	B	C	F_a	F_b	F_c	F_d	F_e	F_f	F_g
0	0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	0	1	1	0	1
0	1	1	1	1	1	1	0	0	1
1	0	0	0	1	1	0	0	1	1
1	0	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	0	0	0	0

Figure 8.8: 7 Segment display using decoders [1]

Same for the one-hot-decoder, for each output (column), we apply reduction technique seen in chapter 5 (in this case it will be Karnaugh map) to derive the logic expression for each output F_n, and draw the logic diagram.

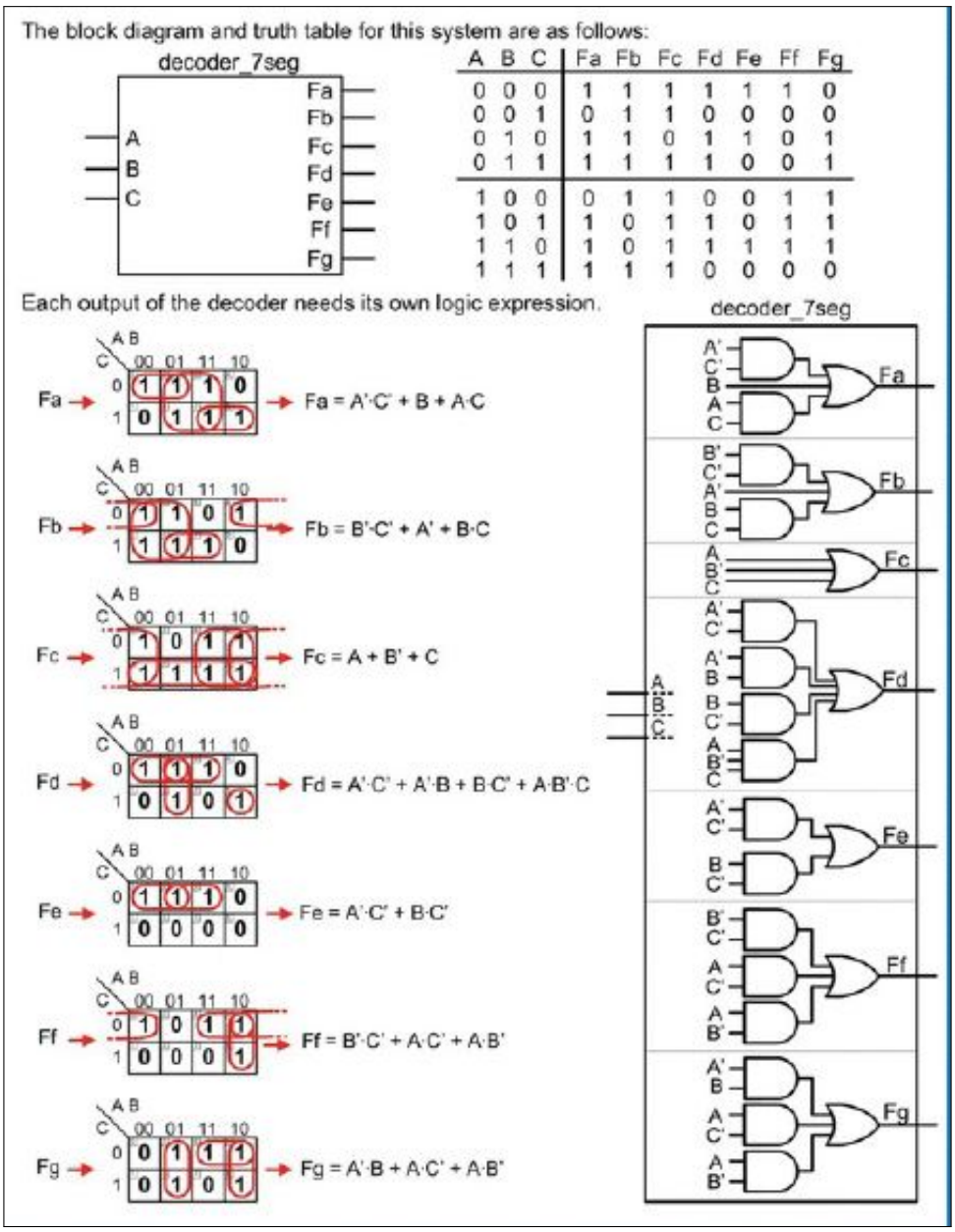


Figure 8.9: 7 Segment display SOP Expression [1]

8.3 Encoders

Definition: Encoding is the opposite process from decoding. Encoding is used to generate a coded output (such as BCD or binary) from a singular active numeric input line. An example is shown in Figure 8.10.

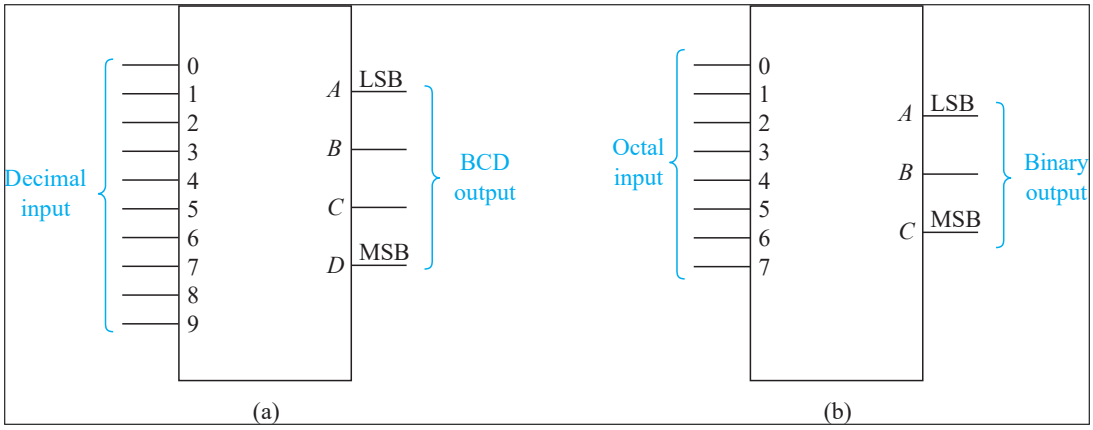


Figure 8.10: Example of Encoder Circuit a) decimal-to-BCD encoder and b) octal-to-binary encoder [2]

Also notice that an encoder always have less outputs then inputs, since the outputs are encoded into some compact representation.

Same as for decoders, there various type of encoders, and in next sections we examine various of them.

8.3.1 One hot Encoder

We take the example 4-to-2 one hot decoder. Since the number of inputs is 4, this means that we can generate 16 possible input combinations. However, we only need 4 of them (we assume the 1st 4), and the other we put them as don't care condition, and use them to optimize the design.

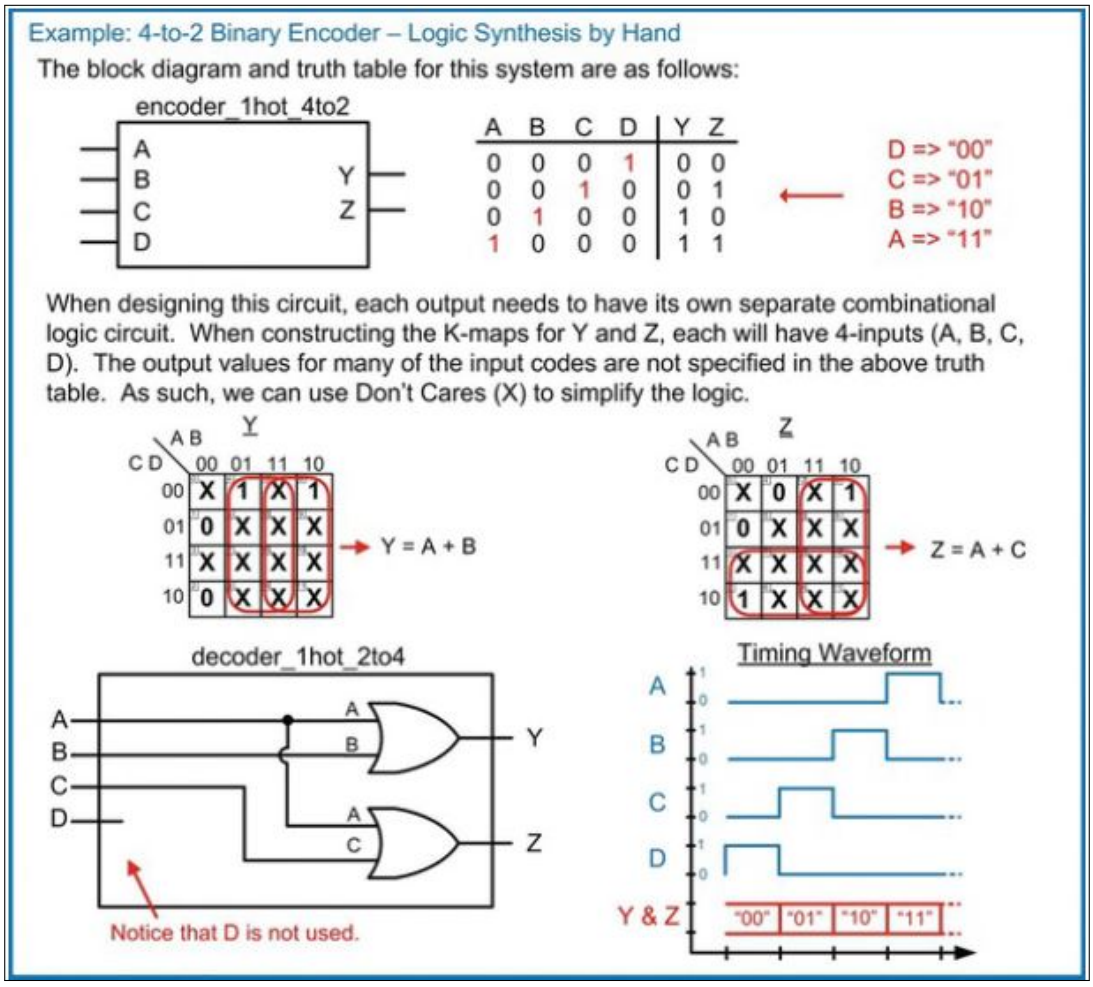


Figure 8.11: 2-to-4 one hot encoder [1]

Also for each output, we derive its own logic circuit using karnaugh map.

8.3.2 Decimal to BCD Encoder

As an example we take the decimal to BCD encoder. Its truth table is shown in Figure 8.12.

Decimal-to-BCD Encoder Truth Table				
	BCD Output			
Decimal Input	*D*	*C*	*B*	*A*
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Figure 8.12: Truth Table for an Decimal to BCD Encoder [2]

- Output A is high for all odd decimal input numbers (1, 3, 5, 7, and 9)
- The B output for decimal inputs 2, 3, 6, and 7
- ...

This means we need to use OR gates. The circuitry is shown in Figure 8.13.

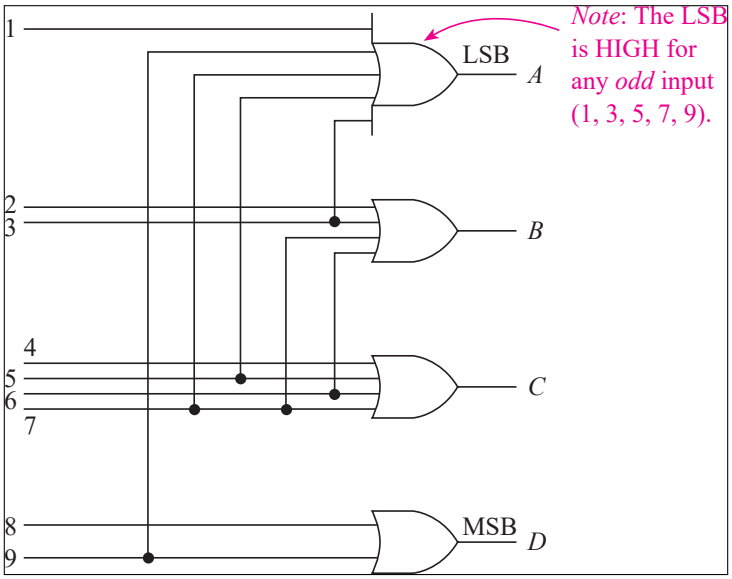


Figure 8.13: Decimal to BCD Encoder Logic Circuit [2]

8.4 Multiplexers

Definition: A multiplexer is a device capable of funneling several data lines into a single line for transmission to another point.

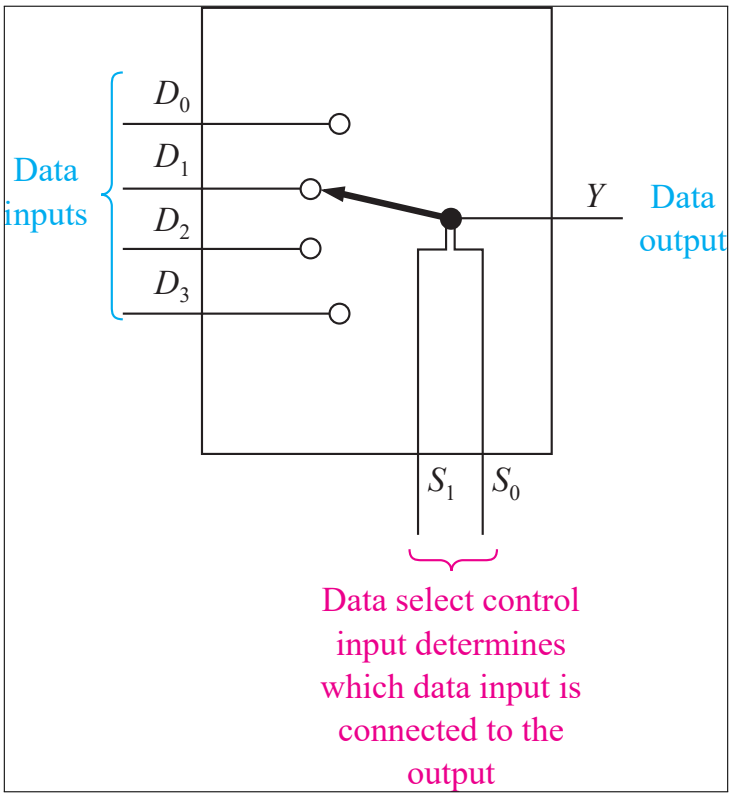


Figure 8.14: Functional diagram of a four-line multiplexer [2]

Another way to see the MUX is some kind of router, which select 1 line and rout it to the output.

8.4.1 2-to-1 MUX

First we begin with 2 to 1 MUX. Its block diagram is shown in Figure 8.15.

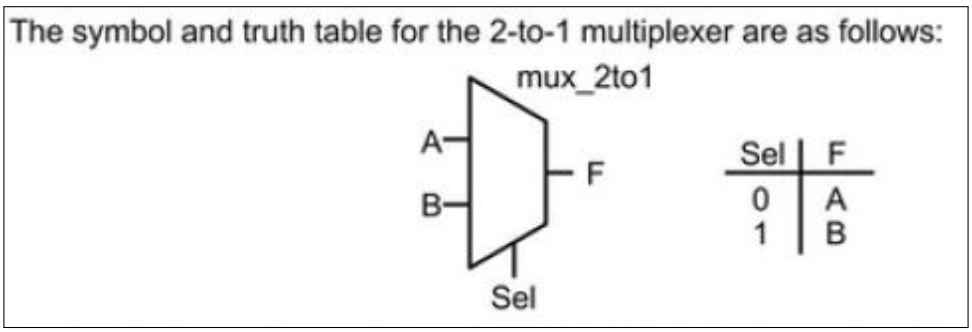


Figure 8.15: 2-to-1 MUX functional block diagram [1]

The MUX device passes one of the input (A or B) through a select line. Now in order to synthesise the logic circuit of a 2-to-1 MUX, we need to turn the table in Figure 8.15 into a more recognisable form, as the truth tables we encountered before. This is illustrated in Figure 8.16.

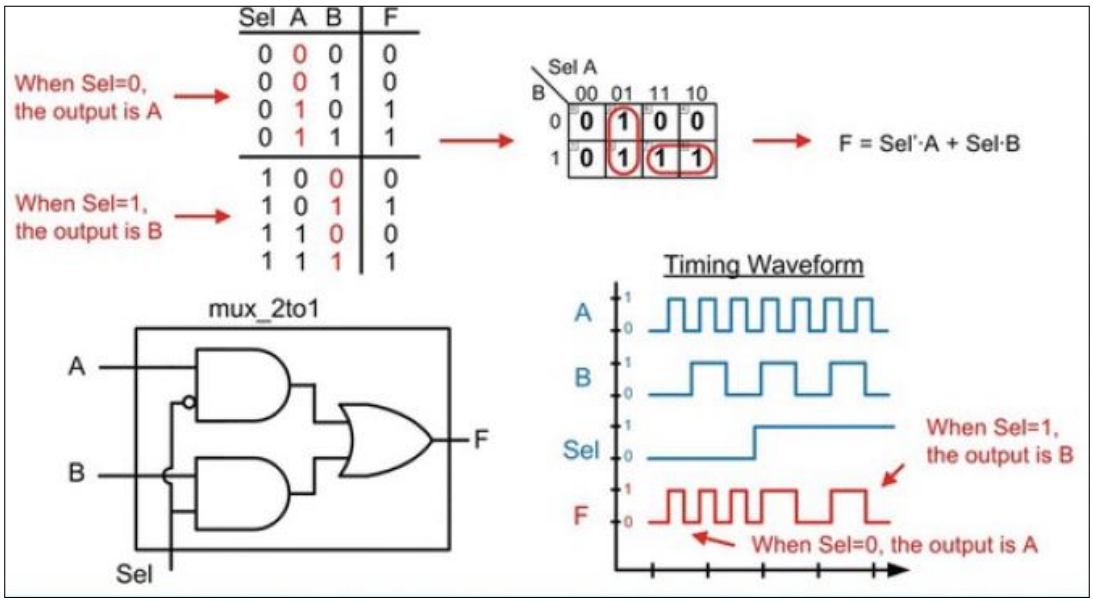


Figure 8.16: 2-to-1 MUX truth table and logic circuit [1]

Notice that when $S = 0$, F copies A, and when $S = 1$, F copies B.

8.4.2 4-to-1 MUX

Suppose now we have more data at the inputs of a MUX, for example 4 inputs. In 2-to-1 MUX, in order to route 2 inputs, we need 1 select line. So in general, ***to route*** 2^n ***input line, we need*** n ***select lines***. In case of 4-to-1 MUX, we need to 2 select line S_0S_1.

To design the logic circuit, it is not necessary to write the truth table. It is only sufficient to dictate for each combination of the select line S_0S_1, what input lines (A,B,C,D) will be routed to the output (as shown in Figure 8.17), and extend the logic circuit in Figure 8.16.

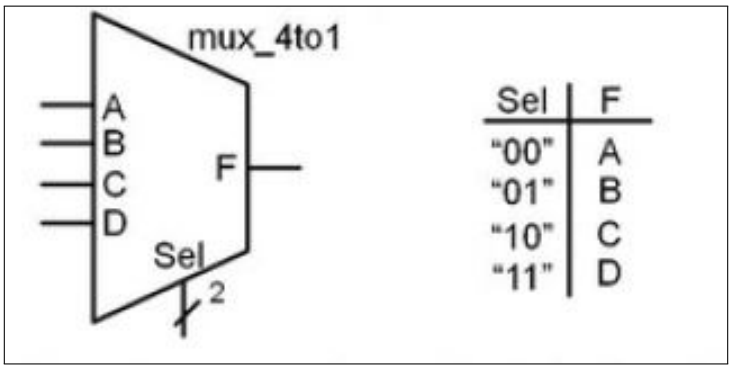


Figure 8.17: 4-to-1 MUX: Select lines combination to route the input lines [1]

The 4-to-1 MUX logic circuit is shown in Figure 8.18.

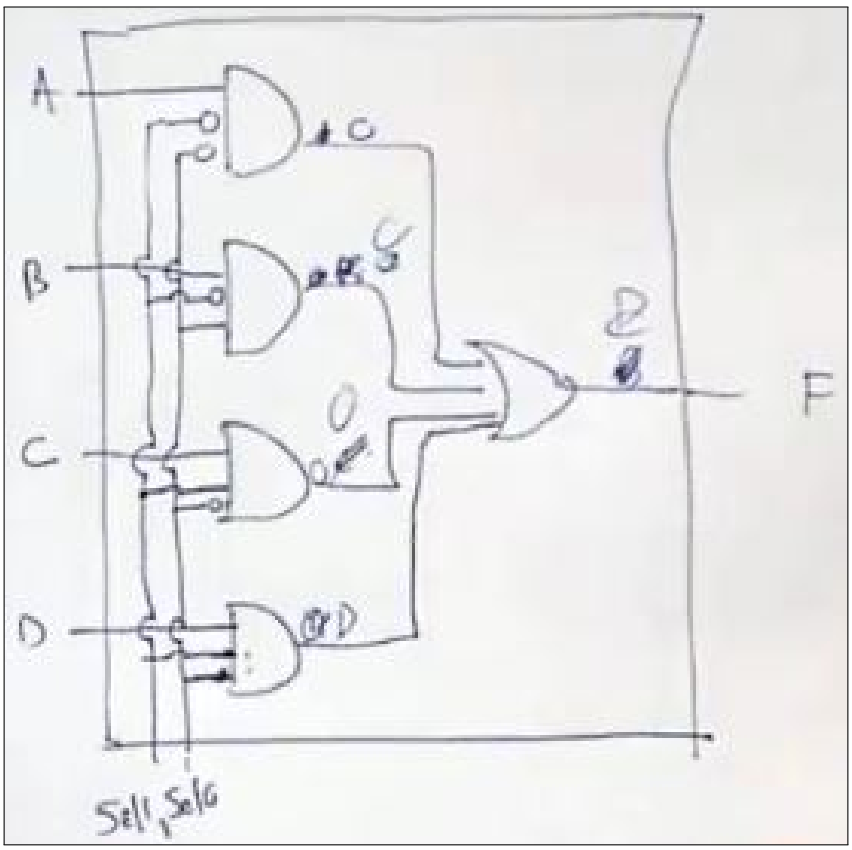


Figure 8.18: 4-to-1 MUX truth table and logic circuit [1]

8.5 DeMux

DeMux has the complementary function of a MUX: the MUX routes several inputs to 1 output only, whereas the DeMux routes 1 input to several output.

8.5.1 2-to-1 DeMux

First we start by a 2-to-1 DeMux. Its functional block diagram is shown in Figure 8.19.

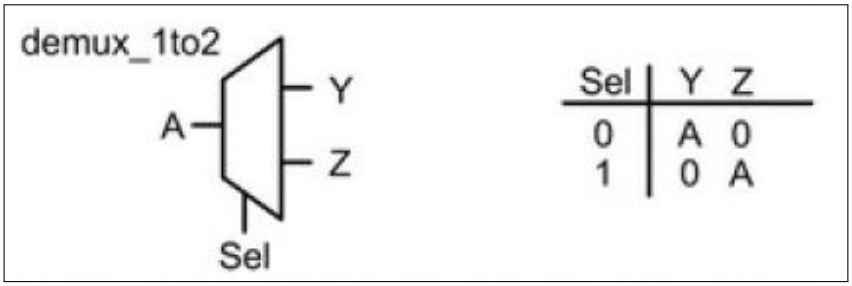


Figure 8.19: 2-to-1 DeMux Functional diagram [1]

If the selection line is 0, A is mapped to Y, otherwise A is mapped to Z. Since we have 2 outputs, we need a combinational circuit for each output. We will do the same as we did in the case of 2-to-1 Mux: transform the block diagram of Figure 8.19 into a more synthetisable logic design table.

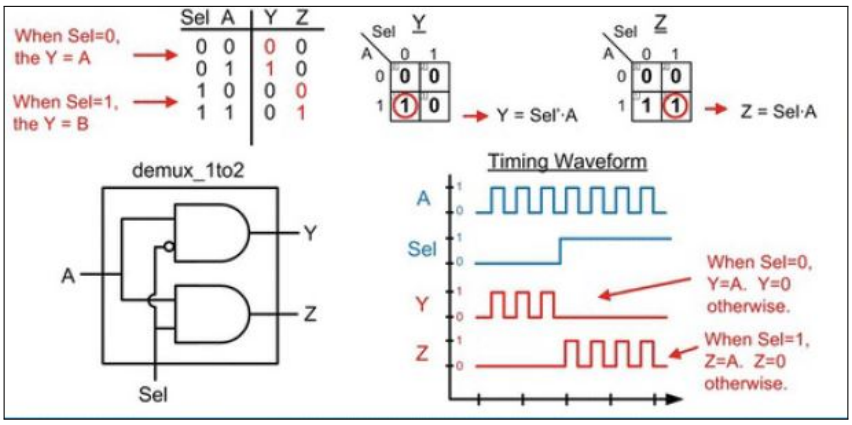


Figure 8.20: 2-to-1 DeMux logic circuit [1]

8.5.2 4-to-1 DeMux

4-to-1 DeMu

: *to write explanation later*

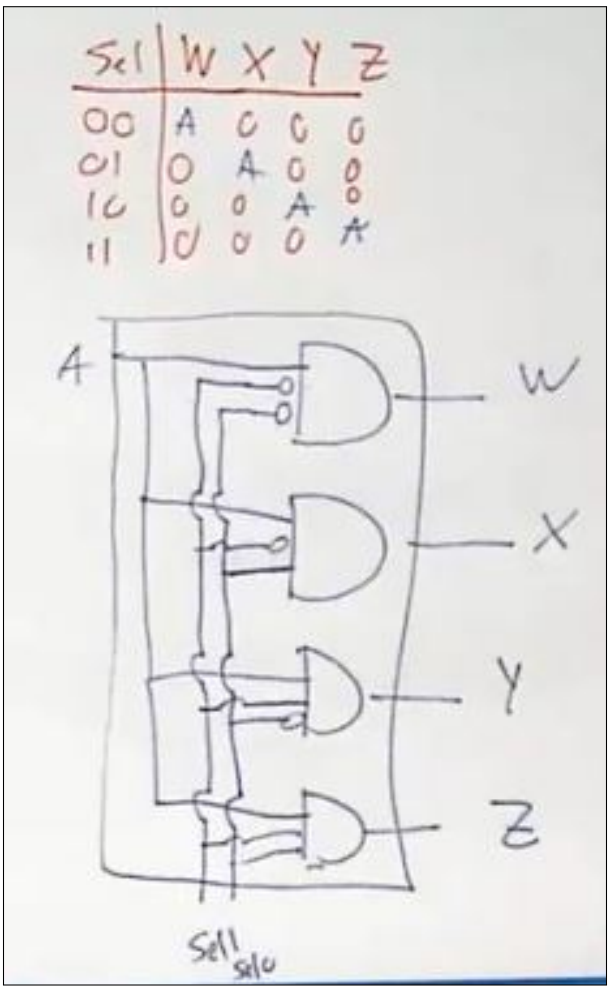


Figure 8.21: 4-to-1 DeMux logic circuit [1]

8.6 Comparators

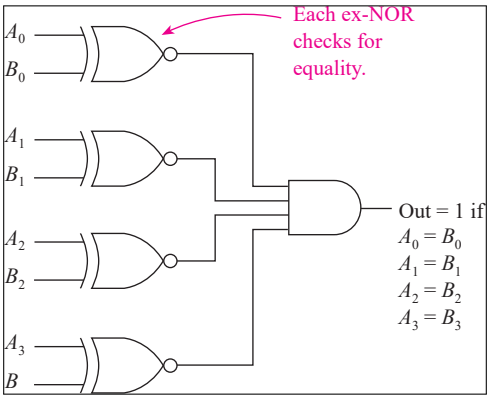


Figure 8.22: 4 bits Comparator using X-NOR gates [2]

- For 2 bits comparison, we can use 1 X-NOR gate: two bits are equal (0-0 or 1-1) if the output of
- For more then 2 bits: we need additional X-NOR gate followed by an AND gate as shown in Figure 8.22

8.6.1 Comparator using 7485 IC package

In Figure 8.23, we have an example of using the 7485 to compare a stream of 8-bits.

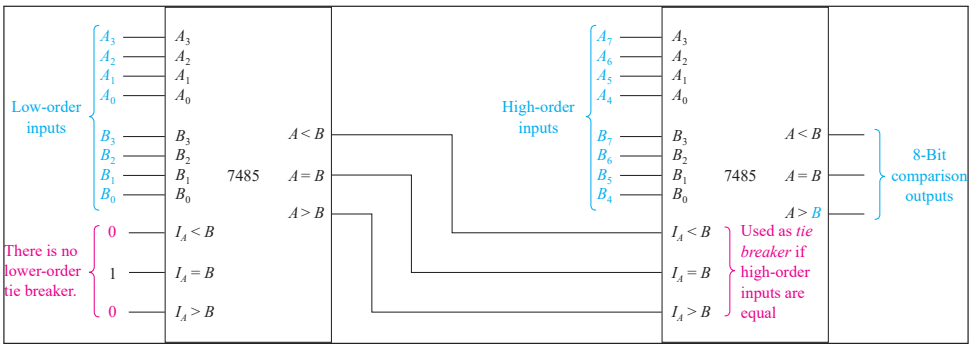


Figure 8.23: 8 bits Comparator 7485 IC package [2]

To understand the concept of tight break, we can do the example shown in Figure 8.24

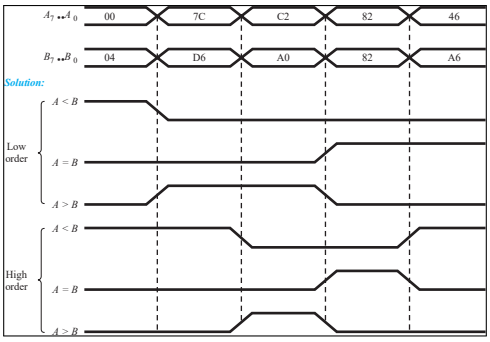


Figure 8.24: 4 bits Comparator using X-NOR gates [2]

- The number are in Hexadecimal representation
- Look at the last 2 digit: 46 and A6
  - The LSB part is an equality ($6 = 6$), so the reuslting waveform is $A = B$
  - In the MSB part: $A > 4$ so the result will be $A > B$, and the final result is $A > B$ (which means A6 > 46)

8.7 Code Converters

Often it is important to convert a coded number into another form that is more usable by a computer or digital system.

This conversion can be made using software or hardware.

8.7.1 BCD-to-Binary Conversion

For BCD to binary, we begin by example shown in Figure 8.25

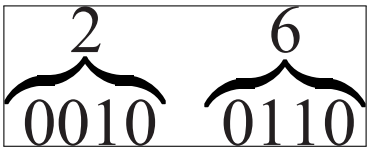
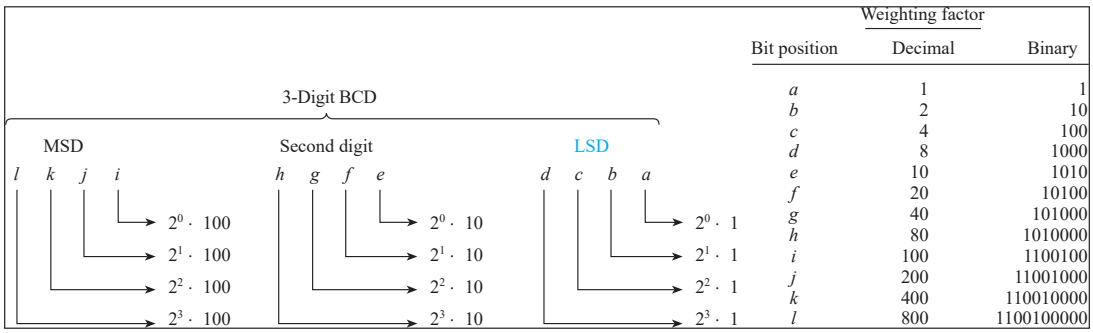


Figure 8.25: BCD to Binary: 26 [2]

- In Figure 8.25, we have the example from decimal to BCD
- If we want form BCD → binary: If we apply regular binary weighting to each bit, you would come up with 38 ($2^1 + 2^3 + 2^5 = 38$) which is wrong
- The 2^{nd} part has a weighting factor of 20

In Figure 8.26, we have the weighting rule for proper conversion from BCD to binary.



Bit position	Weighting factor: Decimal	Weighting factor: Binary
a	1	1
b	2	10
c	4	100
d	8	1000
e	10	1010
f	20	10100
g	40	101000
h	80	1010000
i	100	1100100
j	200	11001000
k	400	110010000
l	800	1100100000

Figure 8.26: Weighting factors for BCD bit positions [2]

In Figure 8.27 we have some example for more practice.

EXAMPLE 8–10

Using the weighting factors given in Figure 8–34, convert the BCD equivalent of 26_{10} to binary.

Solution:

2 6
0010 0110

0010
0100
+ 10100
11010_2 ***Answer***

Check: $11010_2 = 26_{10}$

EXAMPLE 8–11

Convert the BCD equivalent of 348_{10} to binary.

Solution:

3 4 8
0011 0100 1000

1000
101000
1100100
11001000
101011100_2 ***Answer***

Check: $101011100_2 = 348_{10}$

Figure 8.27: BCD to Binary Examples [2]

8.7.2 Gray Coding

Gray Codin

To write it later.

Chapter 9

Arithmetic Operations and Circuits

9.1 Introduction

This chapter will be divided into 2 main part: arithmetic operation and building logic circuit which implement these operations.

9.2 Outline

1. Binary Arithmetic
2. Two's-Complement Representation
3. Two's-Complement Arithmetic
4. Hexadecimal Arithmetic
5. BCD Arithmetic
6. Arithmetic Circuits
7. Four-Bit Full-Adder ICs
8. VHDL Adders Using Integer Arithmetic
9. System Design Applications
10. Arithmetic/Logic Units
11. FPGA Applications with VHDL and LPMs

9.3 Binary Arithmetic

9.3.1 Binary Addition

The four possible combinations of adding two binary numbers can be stated as follows:

- $0 + 0 = 0$ carry 0
- $0 + 1 = 1$ carry 0
- $1 + 0 = 1$ carry 0
- $1 + 1 = 0$ carry 1

General Form: $A_0 + B_0 = \sum_0 + C_{\text{out}}$

Note: When the binary sum exceeds 1, you must carry a 1 to the next-more-significant column, as in regular decimal addition.

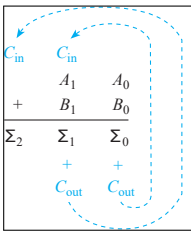
In a truth table form:

Truth Table for Addition of Two Binary Digits $A_0 + B_0$ in the Least Significant Column

A_0	B_0	Sum_0	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 9.1: Truth table of binary Addition [2]

More Significant Bits:



A_1	B_1	C_{in}	Σ_1	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 9.2: Truth table of binary Addition: More significant bits [2]

Examples:

EXAMPLE 7–1

Perform the following decimal additions. Convert the original decimal numbers to binary and add them. Compare answers. **(a)** 5 + 2; **(b)** 8 + 3; **(c)** 18 + 2; **(d)** 147 + 75; **(e)** 31 + 7.

Solution:

	Decimal	Binary
(a)	5 + 2 7	0000 010 + 0000 0010 0000 0111 = 7_{10}✓
(b)	8 + 3 11	0000 1000 + 0000 0011 0000 1011 = 11_{10}✓
(c)	18 + 2 20	0001 0010 + 0000 0010 0001 0100 = 20_{10}✓
(d)	147 + 75 222	1001 0011 + 0100 1011 1101 1110 = 222_{10} ✓
(e)	31 + 7 38	0001 1111 + 0000 0111 0010 0110 = 38_{10}✓

Figure 9.3: Example for Binary Addition [2]

In example e:

- $1 + 1 = 2 \rightarrow 10$ in binary, so we put 0 and borrow 1 in the next column
- We have $1 + 1 + 1 = 3 \rightarrow 11$ in binary, we put 1 and borrow 1 in next column.

  So on we keep this for other columns.

9.3.2 Binary Subtraction

The four possible combinations of subtracting two binary numbers can be stated as follows:

- 0 - 0 = 0 borrow 0
- 0 - 1 = 1 borrow 1
- 1 - 0 = 1 borrow 0
- 1 - 1 = 0 borrow 0

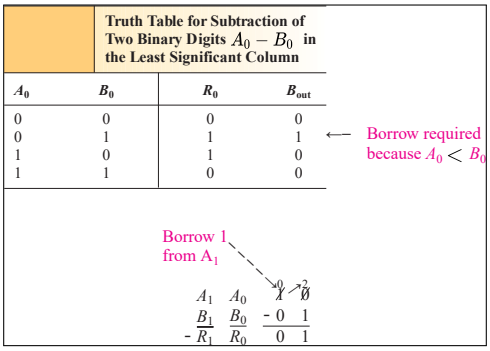


Figure 9.4: Truth table of binary Subtraction [2]

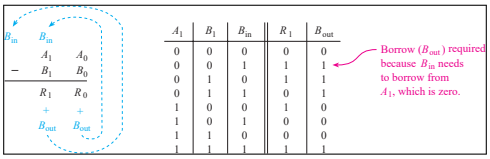


Figure 9.5: Truth table of binary subtraction: More significant bits [2]

Example of Binary Subtraction

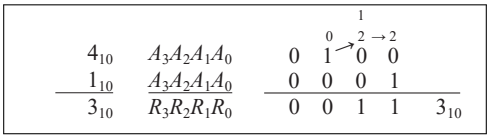


Figure 9.6: Example of Binary Subtraction [2]

More practice:

EXAMPLE 7–2

Perform the following decimal subtractions. Convert the original decimal numbers to binary and subtract them. Compare answers. **(a)** 27 - 10; **(b)** 9 - 4; **(c)** 172 - 42; **(d)** 154 - 54; **(e)** 192 - 3.

Solution:

	Decimal	Binary
(a)	27	0001 1011
	- 10	- 0000 1010
	17	0001 0001 $= 17_{10}$ ✓
(b)	9	0000 1001
	- 4	- 0000 0100
	5	0000 0101 $= 5_{10}$ ✓
(c)	172	1010 1100
	- 42	- 0010 1010
	130	1000 0010 $= 130_{10}$ ✓
(d)	154	1001 1010
	- 54	- 0011 0110
	100	0110 0100 $= 100_{10}$ ✓
(e)	192	1100 0000
	- 3	- 0000 0011
	189	1011 1101 $= 189_{10}$ ✓

Figure 9.7: Example of Binary Subtraction [2]

Note: digital computers use a much easier method for subtracting binary numbers, called *two's complement*. We do, however need to know the standard method for subtracting binary numbers.

9.3.3 Multiplication and Division

Multiplicati
Division

: Do them later from the book.

9.4 Two's-Complement Representation

The importance of Two's Complement: using Two's complement, we can directly do subtraction by doing normal addition: that is if we have 15 - 20, we convert the -20 to its two's complement representation, then do normal addition.
Also, we can do multiplication by series of addition (and division is the inverse of multiplication), till we reach advanced things like computing sin and cos,$\cdots$.

So big idea: addition and two's complement are 2 important notion to master because from them we can implement so many other operations.

Two's-complement method:

- Used to represent *both* negative and positive number

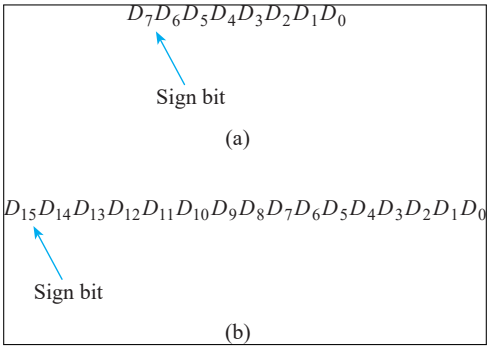


Figure 9.8: Two Complement: a) 8-bit ; b) 16 bits [2]

Some information about Two's-complement representation:

- The MSB is used to determine the sign of a number: 1 is negative and 0 is positive
- Range of numbers: we take 8 bit as an example
  - Positive numbers: 0000 0000 to 0111 1111 (0 to 127)
  - Negative numbers: 1111 1111 to 1000 0000 (-1 to -128)

  As a general rule, for N bits: from $-(2^{N-1}) \rightarrow 2^N - 1$

9.4.1 Conversion: Two Complement and Decimal

Two Complement $\rightarrow$ Decimal:

- If the decimal number is positive, the two's-complement number is the true binary equivalent of the decimal number
- If the decimal number is negative, the two's-complement number is found by
  1. Complementing each bit of the true binary equivalent of the decimal number (this is called the one's complement).
  2. Adding 1 to the one's-complement number to get the magnitude bits. (The sign bit will always end up being 1 since the number is negative.)

Decimal → Two Complement

- If the two's-complement number is positive (sign bit = 0) do a regular binary-to-decimal conversion.
- If the two's-complement number is negative (sign bit the decimal (sign bit = 1) will be -, and the decimal number is found by

  1. Complementing the entire two's-complement number, bit by bit.
  2. Adding 1 to arrive at the true binary equivalent.
  3. Doing a regular binary-to-decimal conversion to get the decimal numeric value.

9.4.2 Table: Two's Complement from -8 → +7

	Signed Two's-Complement Numbers +7 Through −8
Decimal	**Two's Complement**
+ 7	0000 0111
+ 6	0000 0110
+ 5	0000 0101
+ 4	0000 0100
+ 3	0000 0011
+ 2	0000 0010
+ 1	0000 0001
0	0000 0000
− 1	1111 1111
− 2	1111 1110
− 3	1111 1101
−4	1111 1100
−5	1111 1011
−6	1111 1010
−7	1111 1001
− 8	1111 1000

Figure 9.9: Two Complement Table [2]

Practice: see example from the books.

9.5 Hexadecimal Arithmetic

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

Figure 9.10: Hexadecimal Table [2]

The procedure for adding hex digits is as follows:

1. 1. Add the two hex digits by working with their decimal equivalents.
2. If the decimal sum is less than 16, write down the hex equivalent.
3. If the decimal sum is 16 or more, subtract 16, write down the hex result in that column, and carry 1 to the next-more-significant column.

Practice: See examples from the book to make practice.

9.6 BCD Arithmetic

Note about BCD:

Digital electronics naturally works in binary, and we have to group four binary digits together to get enough combinations to represent the 10 different decimal digits. This 4-bit code is called binary-coded decimal (BCD).
BCD Addition:

1. Add the BCD numbers as regular true binary numbers.
2. If the sum is 9 (1001) or less, it is a valid BCD answer; leave it as is.
3. If the sum is greater than 9 or there is a carry-out of the MSB, it is an invalid BCD number; do step 4.
4. If it is invalid, add 6 (0110) to the result to make it valid. Any carry-out of the MSB is added to the next-more-significant BCD number.
5. Repeat steps 1 to 4 for each group of BCD bits.

9.7 Arithmetic Circuit: Adders

First, we review the concept of binary addition.
The concept are for fixed points ↔ no fractions.

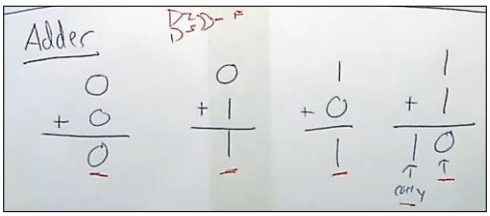


Figure 9.11: Binary Add Cases

Key design: When we need to implement addition cases using combinational logic, we need to remember that these circuits have only 1 bit as output: for each output, we have some circuit which implement this output.

In $1 + 1 = 0$ case, we have 1 as carry bit extra compared to other cases.
For circuitry, we have 1 circuit for the sum bit, and another circuit if we need to output the carry.

Now we move to the circuit section.
The key in designing any combinational circuit is to transform the requirement into a truth table ↔ we need to transform the cases of Figure 9.11 into a truth table. This is shown in Figure 9.12.

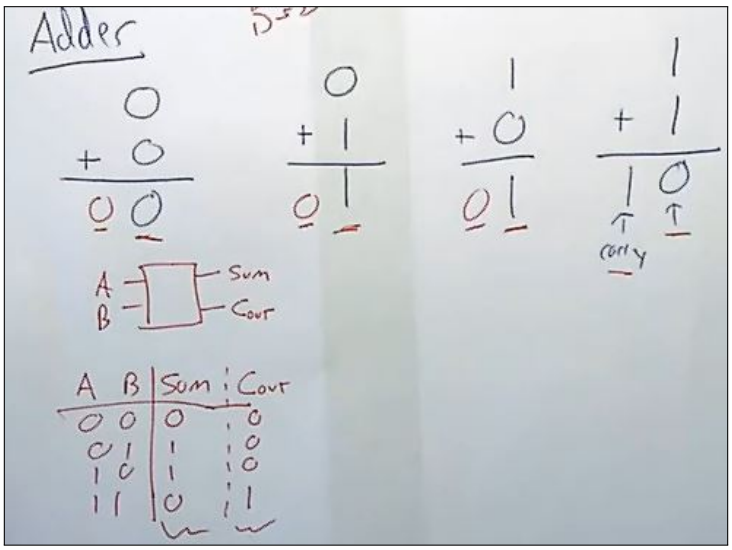


Figure 9.12: Binary Add Cases

We can do a SOP expression, but by inspection, the Sum output corresponds to an XOR pattern, and C_{out} corresponds to an AND gate pattern. The logic circuit is shown in Figure 9.13.

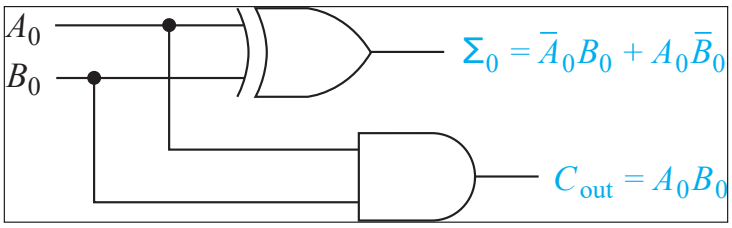


Figure 9.13: Half Adder Logic Circuit: Sum $= A \oplus B$

This adder circuit is called *half adder* because we didn't do anything with the C_{out} bit.

9.7.1 Full Adder

To understand the concept more of a half and full adder, let's take the examples shown in Figure 9.14.

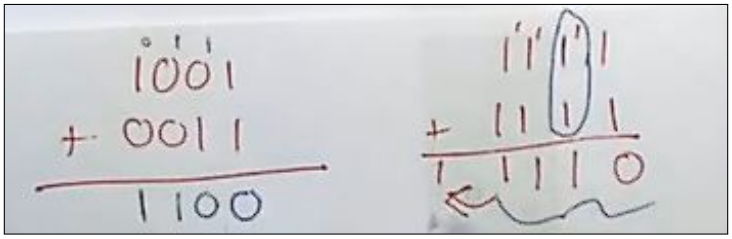


Figure 9.14: 4 bits addition

Key design: First, we need to know that when designing a adder circuit, we don't design to add 2 bits only, but we design to add a chunk of bytes together.

- In the 1st example, the sum was 4 bits
- In the 2nd example, the sum were 5 bits, where the last bit is the C_{out}

- Also what can we learn from these examples, that we need 3 input when adding multiple bits: A and B, and also the carry bit coming from previous sum of lower bits positions
  - A more general rule is shown in Figure 9.15

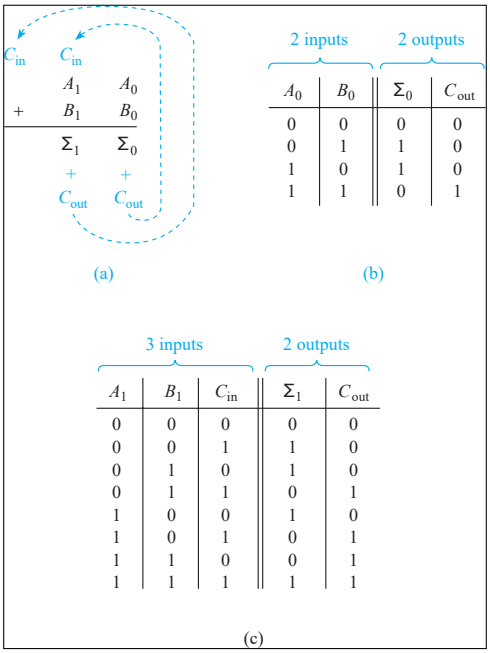


Figure 9.15: 2 bits and 3 bits addition truth table [2]

What we will do is design a circuit which implement the 3 bit sum used in the example of Figure 9.14. First we put the truth table corresponding to such a situation, then use K-map to derive the logic expression for the Sum and Carry, and finally synthesize the logic circuit. These steps are shown in Figure 9.16

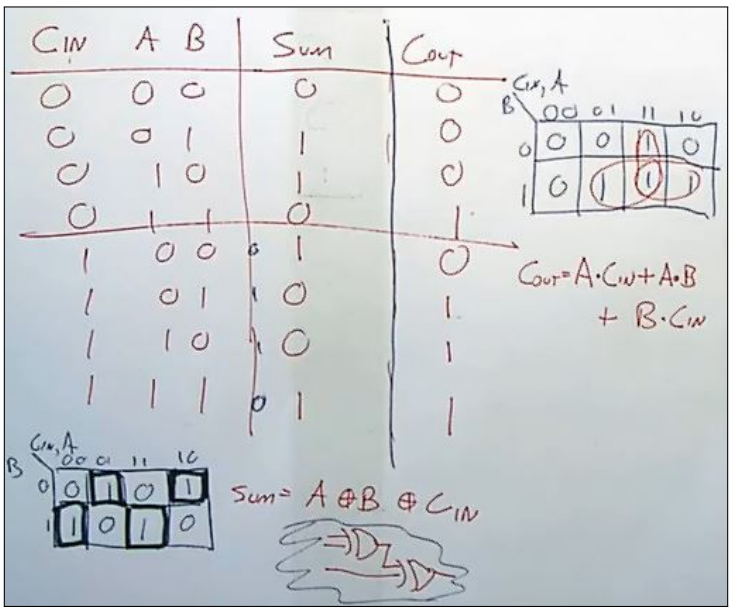


Figure 9.16: 3 bit Addition: Truth table and K-map

- The Sum bit K-map corresponds to checkerboard pattern

  to review the concept of checker board patter and where I used it

checkerboar pattern for addition:

9.7.2 Reusing Halfadder to implement Full Adder

Key design: Reusing components already built in some new design.

It is true that for a full adder, we have derived the logic expressions shown in Figure 9.16, but we will try to use the circuitry done in half adder (Figure 9.13) and use them in full adder.

For the Sum bit, the extension is easy since the sum bit in full adder (Figure 9.16) can be view as:

$$\text{Sum}_{\text{full adder}} = \overbrace{A \oplus B}^{\text{Old Half Adder Input}} \oplus C_{\text{in}} \tag{9.1}$$

Equation 9.1 tells us to take the first 2 bits A and B input to the first half adder circuit, then take the input C_{in} with $A \oplus B$ to the 2^{nd} half adder circuit. This is shown in Figure 9.17.

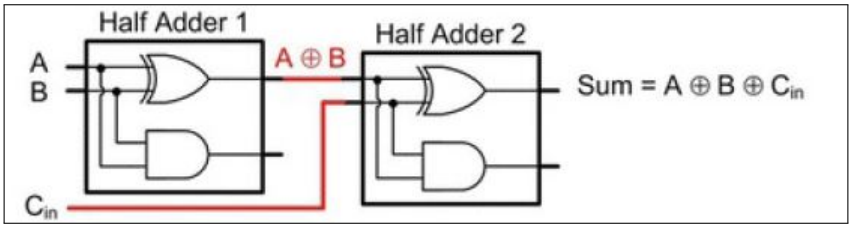


Figure 9.17: Full adder using half adders

Now for the C_{out} the story is a little bit more complicated, sine the equation is not that obvious about the usage of half adder circuitry. The extension can be done using the equivalence relation shown in Figure 9.18, between the old C_{out} equation in Figure 9.16, and the new one.

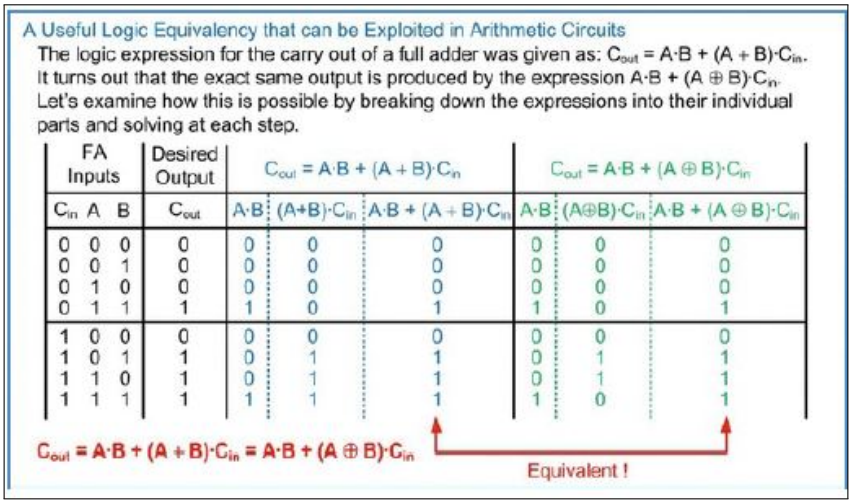


Figure 9.18: C_{out} Equivalence logic equations

The final full adder circuit from a half adder for Sum bit and C_{out} is shown in Figure 9.19.

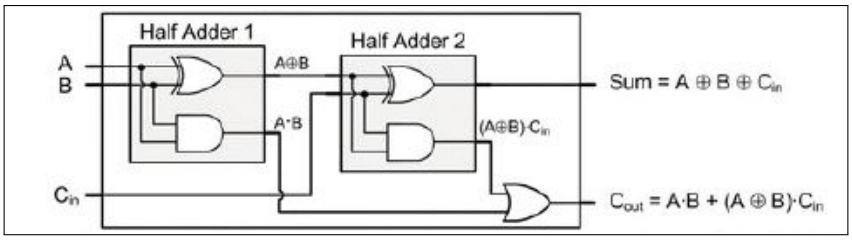


Figure 9.19: Full adder from 2 half adder: Sum bit and C_{out}

9.7.3 Multiple bit Full Adder: Ripple Carry Adder

Now all the previous work and circuitry was done for 1 column of bit addition in Figure 9.14. Now if we wish to do for all the other columns, we can extend the circuitry of Full adder from previous sections as shown in Figure 9.20.

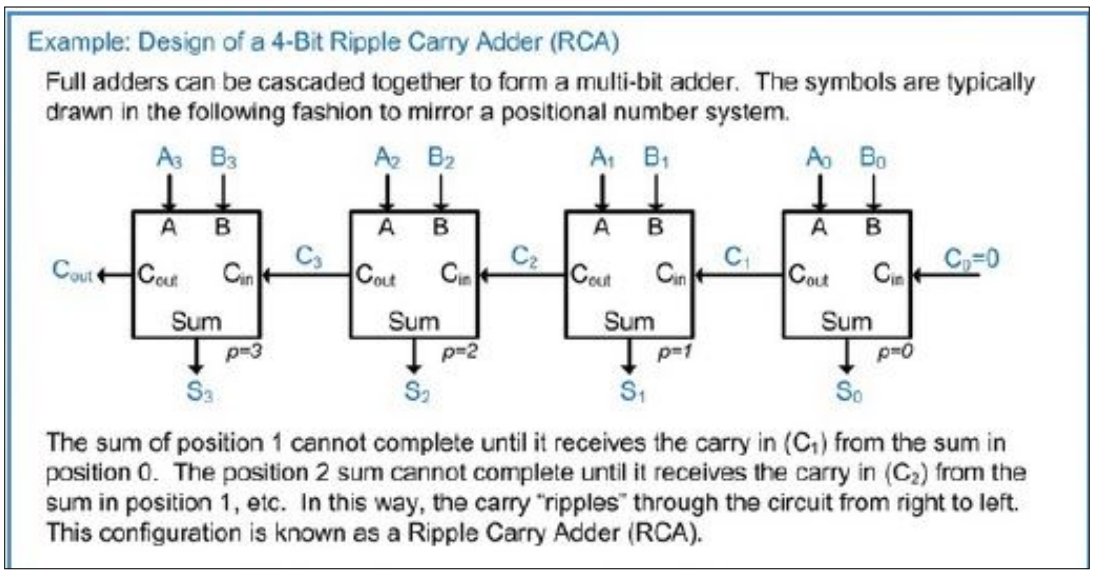


Figure 9.20: C_{out} Equivalence logic equations

9.8 Timing Analysis of RCA

RCA works well, but we have to pay attention that we need to wait for C_1 in order to compute for sum in A_1B_1 input, and so on for the other position bits.

RCA timing analysis

listen to video 12.1b again

9.9 Verilog Application for Adder Circuit

der IC
kage

. This will be done later:
Points left to do:

- The adder circuit are available via IC package
  - See section 7-7 in [2], like the 7483 family (a 4-bit full adder)
- It talks about to use these IC to make addition (4 bit, 8-bit, Two's Complement operation)
- Then some application using VHDL to simulate Adders
  - This need to be done later in Verilog
- The ALU explanation
- Section 7-11: Simulate all of the above (ALU, adders) via a combination of macro-function, VHDL and Library of Parameterized Modules (LPM), available in Quartus
  - To see if we can do the same thing via Verilog

Chapter 10

Sequential Logic

10.1 Introduction

- Sequential system have outputs which depends on past input and past outputs
- This means that sequential system can take more intelligent decisions, compared to combinational, where output of a combinational system depends only on instantaneous inputs (ti can't ***store*** values of pas inputs to learn from it for example)
- This motivate the following point: how we can ***store information*** ?
- We will begin our discussions using ascendant level of difficulty, beginning with cross coupled inverter pair, and finishing with the most used storage device used in any digital system, which is the D-Flip flop

10.2 Storage Device

10.2.1 Cross Coupled Inverter

Let's take a look at Figure 10.1

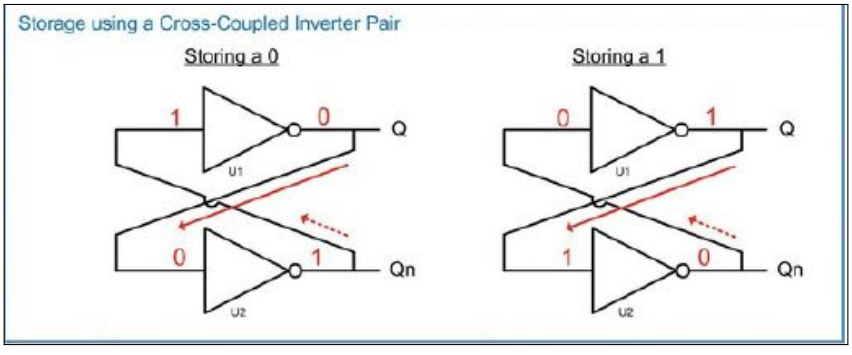


Figure 10.1: Cross inverter topology to store 1 and 0

- This common topology is called cross coupled inverter
- Each output of the inverter is fed to the other inverter (below it), then the output is fed back to the 1^{st} one.

- At the left part, we input some test signal (1) and hold it, the output Q is 0
  - Notice that this output will still the same since the beneath inverter keep inputting 1 to the above inverter, without the intervention of some outsider system
  - This mean that Q will always be 0. In other words, the cross coupled inverter will ***hold*** (store) the value 0
- Same concept for storing 1

10.2.2 Metastability

Now we discuss another notion we need to understand when building storage device, and that is ***metastability***.

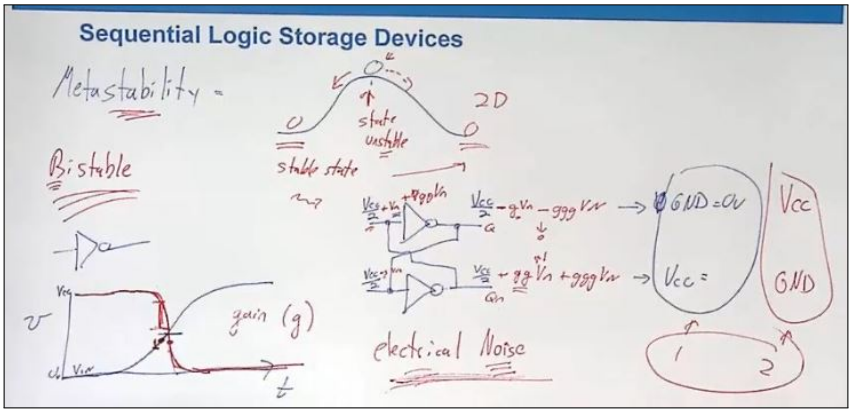


Figure 10.2: Metastability concepts

- Metastability: is a state where a device is an stable state, but any tiny change can push the device into another stable state.

  An example is the ball shown in Figure 10.2
  - The ball is sitting on the top of the hill $\leftrightarrow$ this is the metastable case
  - Any perturbation of the ball will move it left or right
- Similar to cross coupled inverter, if we input for example Vcc/2 (or some value), and since we have some noise and gain in the inverter, the output Q will be Vcc/2 -g.n and Q' will be Vcc/2+ g.n
- Due to the feedback, the gain factor (g) will be multiplied , and hence will reach a final state of Q = 0 and Q' = 1
- Q could be also = 1, if the noise is negative voltage for example
- The take away: we begin with a value of Vcc/2, and end up with 2 different value Q = 0 or Q = 1
- key concept: We don't to build circuit in which we have no idea where the output will stabilize

10.2.3 SR Latch

Going back to the cross coupled inverter, we have seen that throughout the feedback, an hold state can be achieved. However, there is no input at all to the system, which means we don't know what is the initial value of the output Q.

The SR latch have been developed to solve this issue, where we simply replace the inverters by NOR gate.

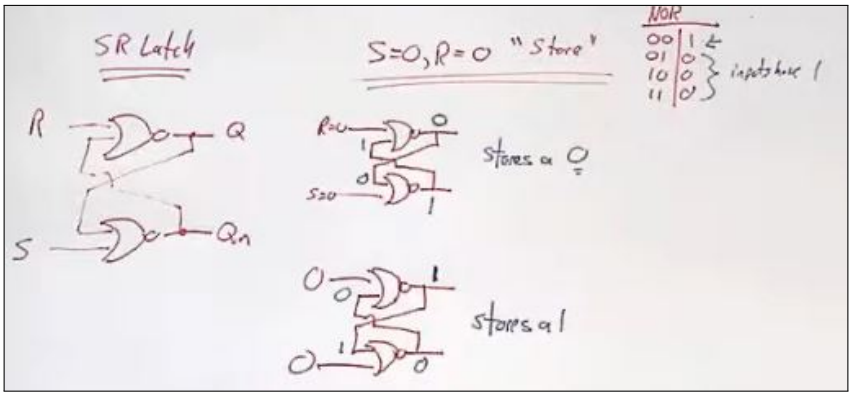


Figure 10.3: SR latch in Store state

- Notice that now, since we have NOR gates, we can drive input at pints SR
  - S stand for Set and R stands for Reset
- Recall that NOR gate: always 0 whenever one of the inputs is 1

Store case
We begin the analysis with saving or storing result

- Doing tracking SR = 00 and manipulating other pins (10 or 01), we found that for SR = 00, we store either 0 (for 10) or 1 (for 01)

Note when doing the analysis of SR latch: when analyzing the output of SR latch for a given inputs, start from the S pin, compute the output (Q') and then drive it back along with the R pin to compute the output Q.

Set and Reset State

- Now we change either S or R as shown in Figure 10.4

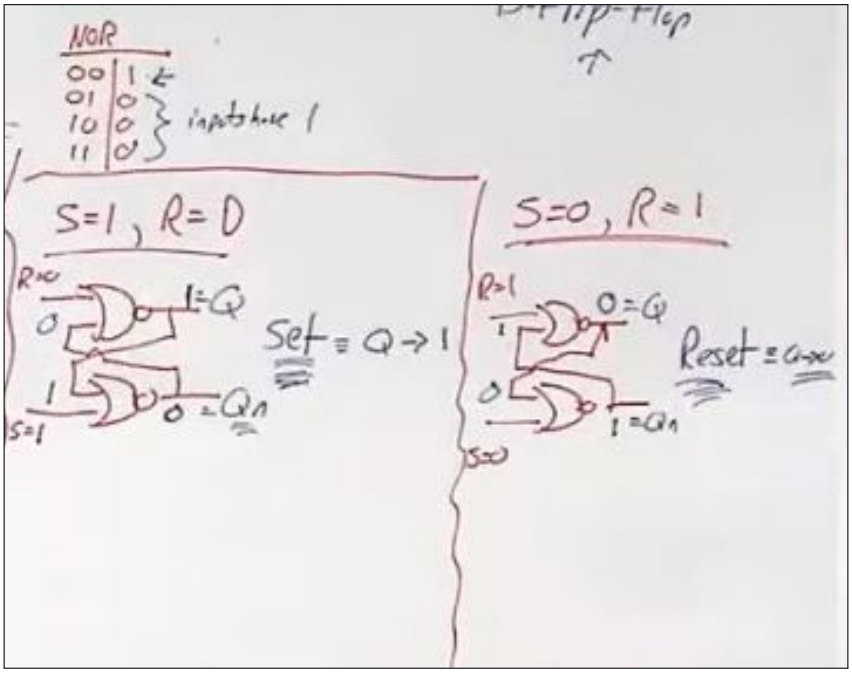


Figure 10.4: SR latch in Set Reset State

  - In the set state (S = 1): the output will be 0 whatever the other input is
  - The 0 get feedback to R = 0; giving rise to Q = 1 $\leftrightarrow$ for a set state, we have stored 1
  - Same analysis for reset case (R = 1 and S = 0) $\leftrightarrow$ we store a 0 (Q = 0)

Don't Use Case: the don't use case is for SR = 11, which give QQ' = 00.

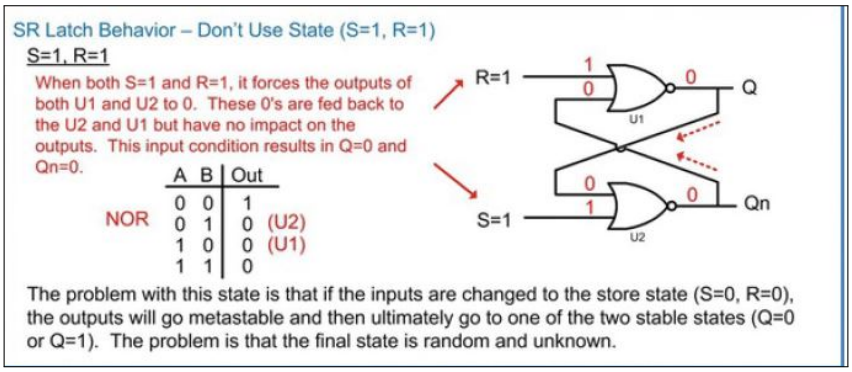


Figure 10.5: SR latch in don't use case

There are 2 problem in such a circuit:

1. The Q and Q' are not complemented
2. When going from SR = 11 → SR = 01 or 10, there is no problem in such cases, we know what is the output is.

   But when going from SR = 11 → SR = 00 (store case), we know that we have 2 outputs ***but we don't know which on of them***

A truth table for SR latch cases is shown in Figure 10.6

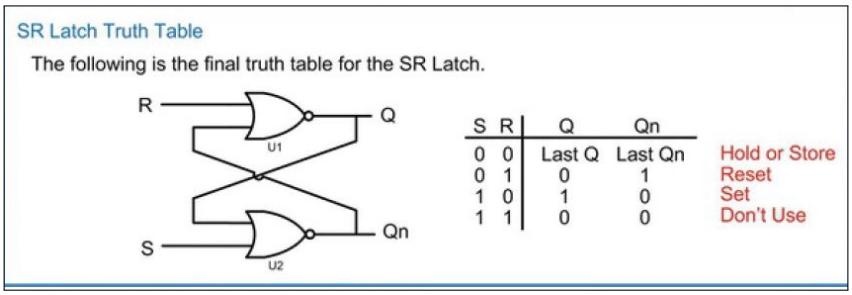


Figure 10.6: SR latch truth table

10.2.4 S'R' Latch: the complement SR latch

The S'R' latch will have similar behavior as SR latch. But instead of a NOR gate, we use a NAND gate (will give 1 whenever one of the input is 0).

See section 7.1.4 in the book [1] for analysis details and truth table derivation.
The final truth table is shown in Figure 10.7.

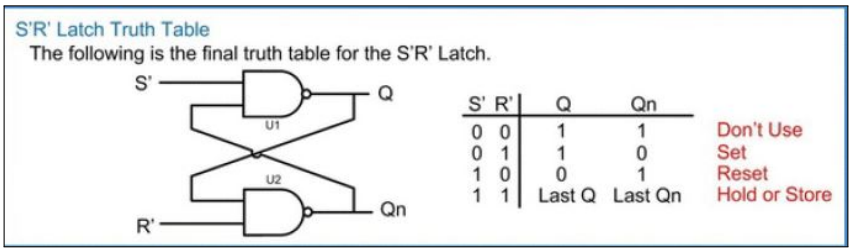


Figure 10.7: S'R' latch truth table

10.2.5 SR latch with enable

Now we move to a more sophisticated device where we combine the SR latch with enable pin (clock pin), and using NAND gates, as shown in Figure 10.8.

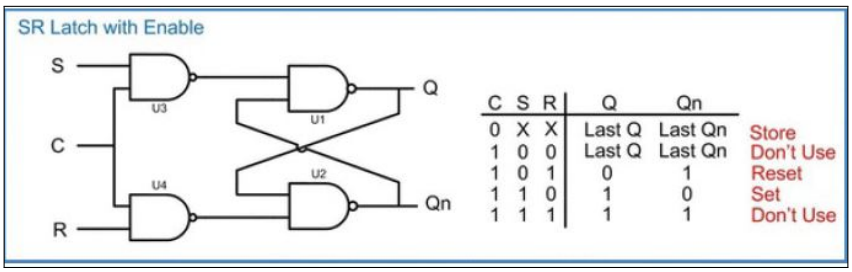


Figure 10.8: S'R' latch truth table

- The idea is to use NAND gate to invert the polarity of the SR pin, so we have the behavior of S'R' latch (in Figure 10.8)
- Now the clock signal will store (for C = 0), and set or reset Q for C = 1
- We still have a don't use case for C = 1, S = 1 and R = 1
- A don't use store case: for C = 1 and SR = 00, we can use the device for storing, but we won't do that because we want store mode for C = 0

10.2.6 D-Latch

In the SR latch with enable, we can reduce the truth table when S and R are complementary to each other, and this can be done by putting an inverter in front of the S pin and removing the R pin, hence we obtain the D-latch shown in Figure 10.9.

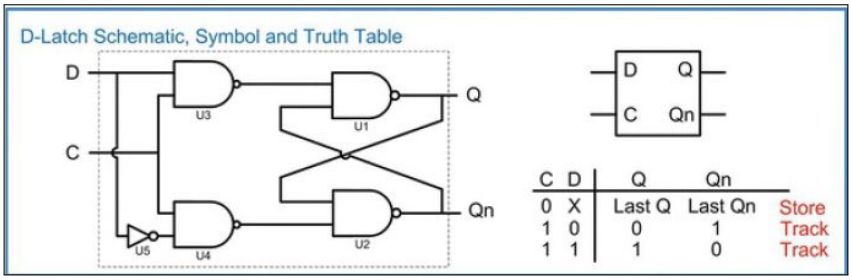


Figure 10.9: D latch

- Note: D stands for data

Notice that now Q track the state of D for C = 1, and gets into a hold state for C = 0.

A timing diagram illustrating the D-latch behavior is shown in Figure 10.10.

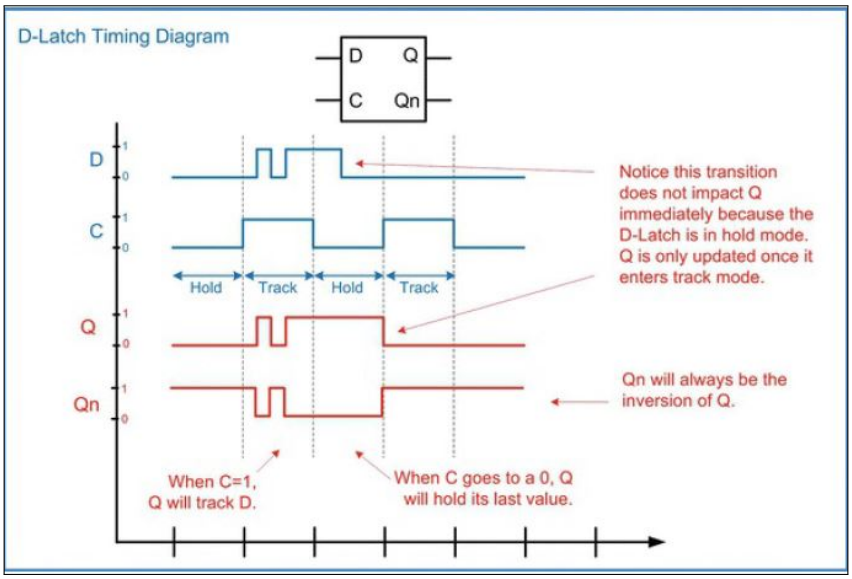


Figure 10.10: D latch timing diagram

- When C = 1, the Q is the same as D
- When C = 0 (hold state), even if D change during this interval, Q will be in a hold state (keep the past value)

Take away: if D change, this doesn't mean necessarily that Q change, we need to see the C if we are in hold mode or track mode.

10.2.7 D-Flip Flop

For a D-Flip flop, we will use 2 D-latches to make events at ***rising edge of a clock*** instead of taking values at intervals of a clock as in D-latches.

The composition of a D-flip flop is shown in Figure 10.11.

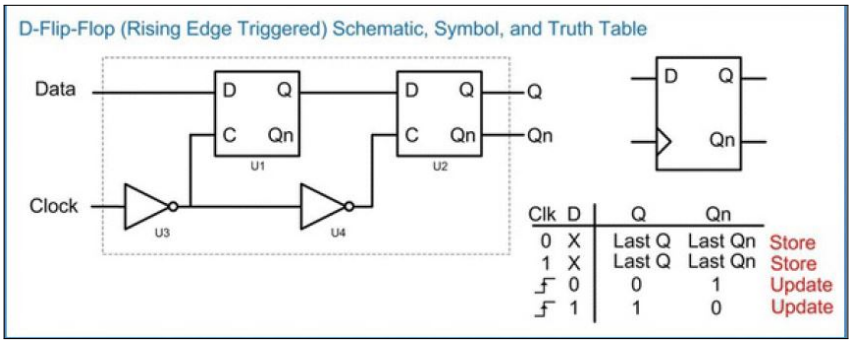


Figure 10.11: D flip flop

- Notice that the symbol of a D-flip flop has > sign, which stands for a rising edge D-flip flop

The way the 2 D-latches of Figure 10.11 are complementary to each other: 1 will be in a track mode and the other is in store mode.

Timing diagram is shown in Figure 10.12.

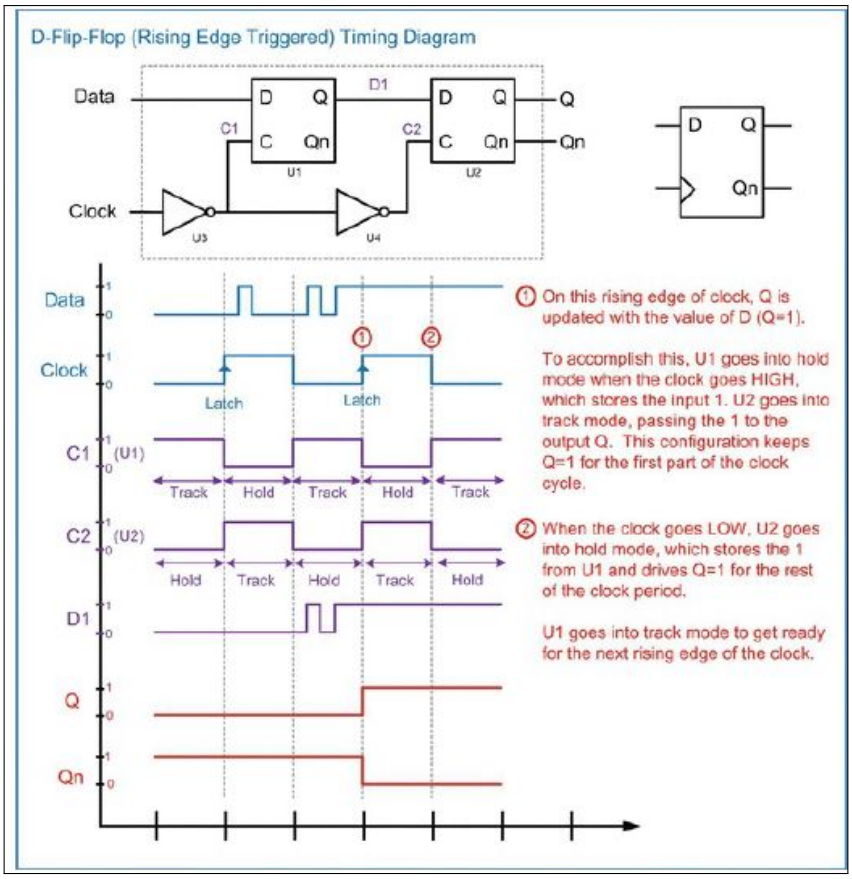


Figure 10.12: D flip flop timing diagram

To derive the output Q:

- Derive the output D1, which is controlled by C1
- Then derive Q, which is controlled by C2 (which is the same waveform as the initial clock)
  - Be aware that in the U2 D-latch, the data now are the D1 → the track and hold state are derived from D1
- At the end, we will have event on rising edge of the clock: the D-flip flop will track D on rising edge , and enter hold mode when we have 0 or 1 value of the clock

10.3 Timing Consideration for Sequential Logic

Now turn attention to timing characteristics in sequential logic.

Back in Figure 10.12, we notice that the data (D signal) values are stable at the rising edge of the clock: we have either 0 or 1. We don't have special case where at the rising edge of the clock, the D signal is also rising (see Figure 10.13), this cause the D-flip flop to enter in a metastable case , and it won't know whether it should take 0 or 1.

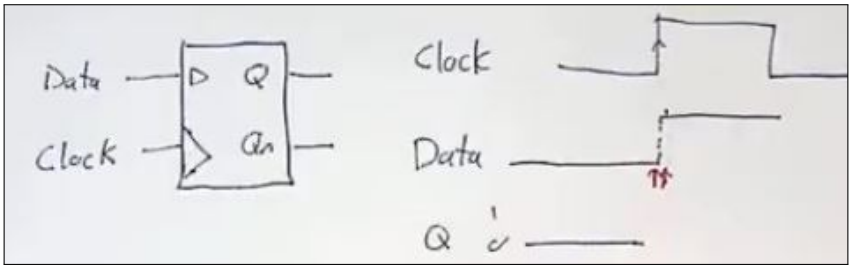


Figure 10.13: D flip flop Special Case: the data is rising with the clock signal

10.3.1 $t_s t_h$

That's why when transitioning, we should apply data transition in certain region called $t_s t_h$, which corresponds set-up hold period (see Figure 10.14).

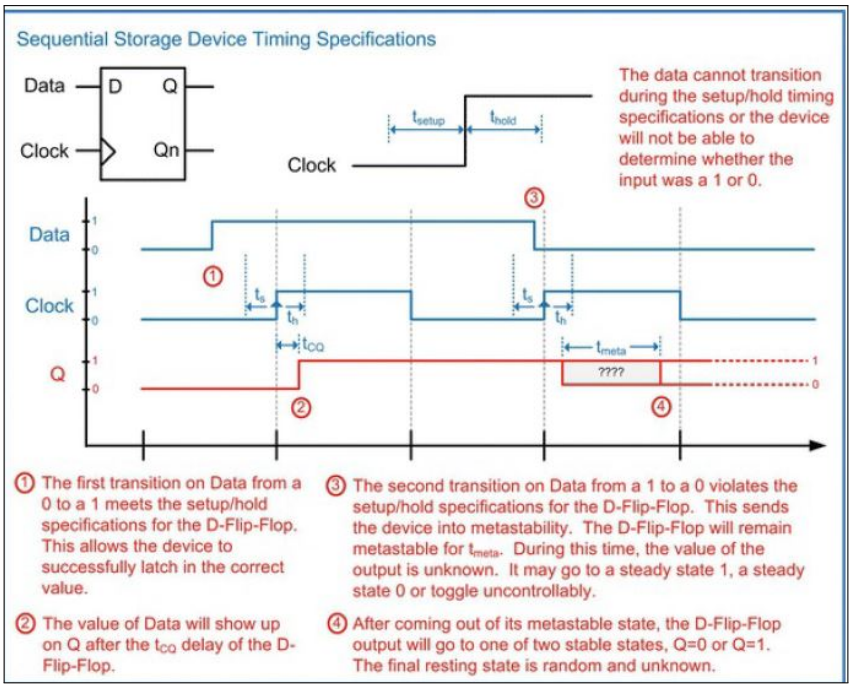


Figure 10.14: Set up Hold region

- t_s: how long the data should stable ***before the clock event***
- t_h: how long the data should stable ***after the clock event***

10.3.2 t_{meta}

Another timing characteristic is the t_{meta}: it is the time interval in which the D-flip flop oscillates between unknown values because we have violate the setup hold conditions.

- In Figure 10.14 , the condition is violated at 3) because we didn't respect t_s

10.3.3 t_{CQ}: clock to Q

The final timing characteristic is called t_{CQ}: clock to Q. That is , even if we respect the setup hold interval, there is a propagation delay in order to update the value by the D-flip flop.

10.4 Common Sequential Circuits

Using sequential devices, we can build complicated logic system, but even with only sequential devices, we can create some useful circuit. This is the goal of this section.

10.4.1 T flip flop

The T flip flop is just a D flip flop, but the Q' is wired up directly to the D signal (see Figure 10.15).

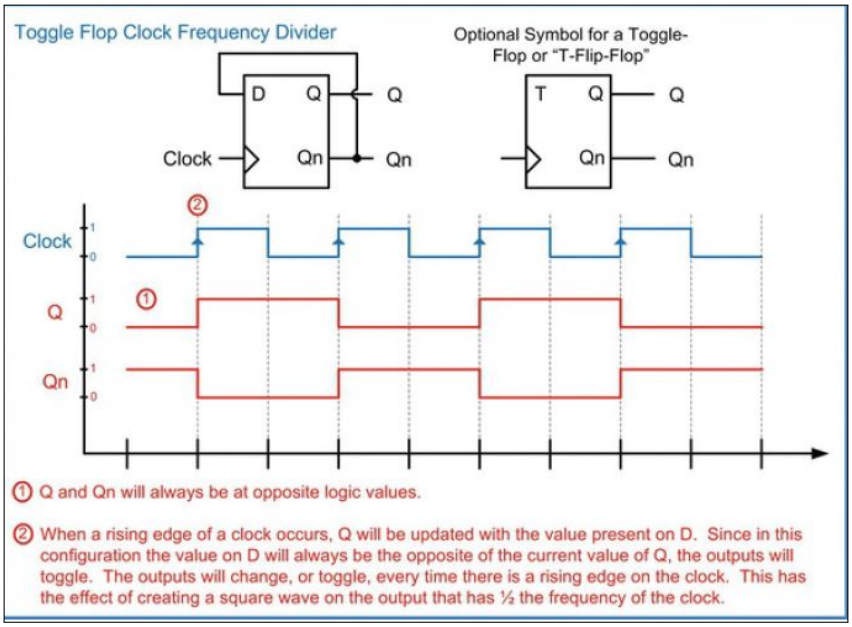


Figure 10.15: T flip flop

- Since the D = Q', then Q will take the previous value of Q' at each rising edge
- The resulting waveform for Q is toggling between 0 and 1, but with doubling period (because we won't change value till the next rising edge)

Hence, the T flip-flop is a ***clock divider*** (see Figure 10.16). We can use the t-flip flop to obtain new clock frequencies

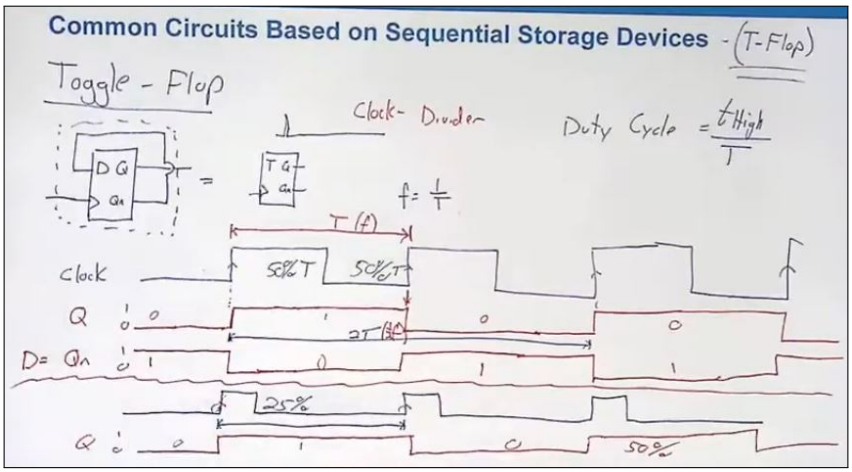


Figure 10.16: T flip flop Clock divider and clock cleaning

- Duty cycle $= \frac{t_{\text{high}}}{T}$, where T is the period between 2 rising edges
- A perfect duty cycle for a clock would be 50 %
- If we have some small duty cycle, then the D-flip flop (or any other flip flop) won't have the time to read the data, that's why we don't want small duty cycles and we can use the T flip flop to make a perfect duty cycle

Timing Consideration for T flip flop

Now we have said that a T flip flop have Q' directly fed to D, which means in theory that Q' and D have same waveform, because there is only a wire between them (see Figure 10.17). This means that the T flip flop will enter metastability region because we have violate the $t_s t_H$ interval.

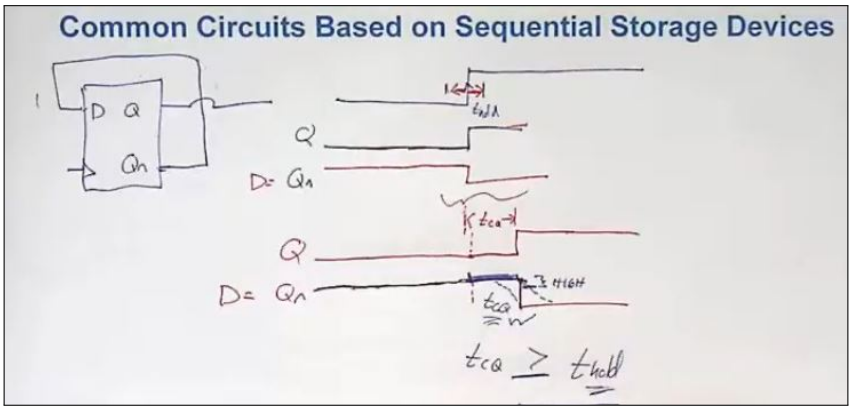


Figure 10.17: T flip flop Clock divider and clock cleaning

However, in reality we have propagation delay t_{CQ}, so if we have $t_{CQ} > t_H$, we won't enter metastability region.

Take away: if we respect $t_{CQ} > t_H$, then we won't have problems.

10.4.2 Binary Up Counter

Another circuit we can build is called binary up counter. In Figure 10.18, we have an example of 3 bit binary up counter.

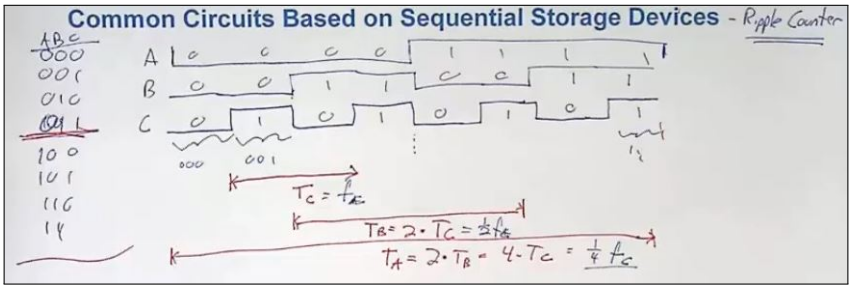


Figure 10.18: 3 bit binary up counter

Notice that we have frequency division from LSB to the MSB.

For a realization of this counter, we can use T flip flop as shown in Figure 10.19.

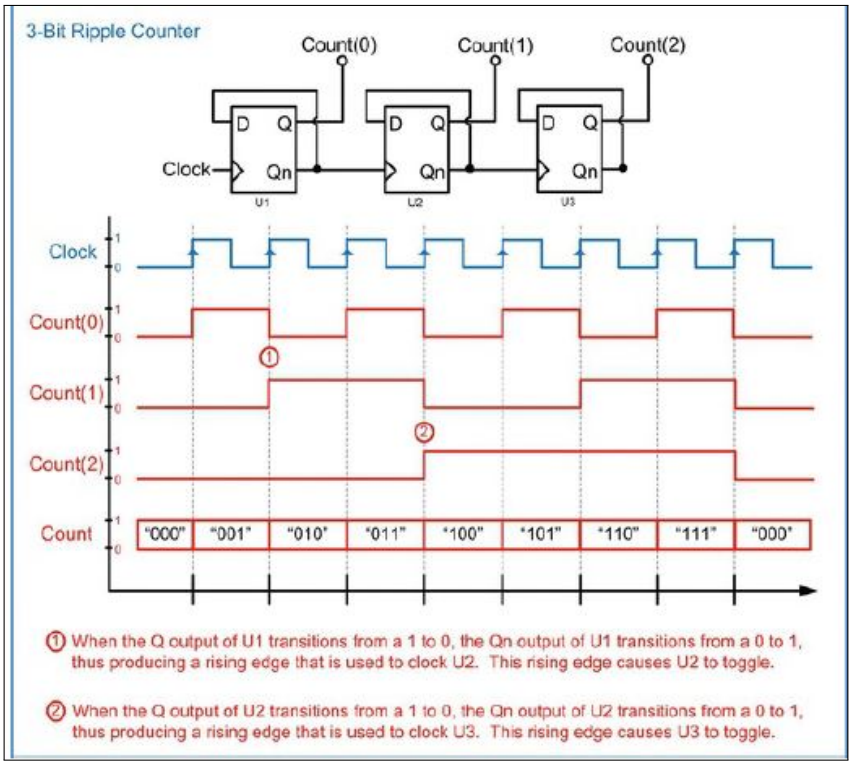


Figure 10.19: 3 bit binary up counter circuitry using T flip-flop

- The clock of bit B is fed from bit C, and from Q' pin.
  - We choose Q' because we need to transition using falling edge and not rising edge

Ripple Counter Delay: there is also the issue of delaying t_{CQ} in ripple counter. For 3 bit counter we don't observe the effect of delay, but for higher order delay counter (like 32 bit for instance) we observe the effect of delay because they are accumulating (see video explanation for details).

10.4.3 Switch Debouncing

Refer to 7.3.3 in [1]. Main points:

- Switch are mechanical devices
- After pressing the switch, there is some transient state where the mechanical part go back and forth and hence generate a pseudo square wave until reaching the steady state
- This transient state must be taken care
- They use some S'R' latches to solve the issues

10.4.4 Shift Registers

Shift registers are formed by cascading D-flip flop, such that the output of D_{n-1} is the input to D_n flip flop, as shown in Figure 10.20.

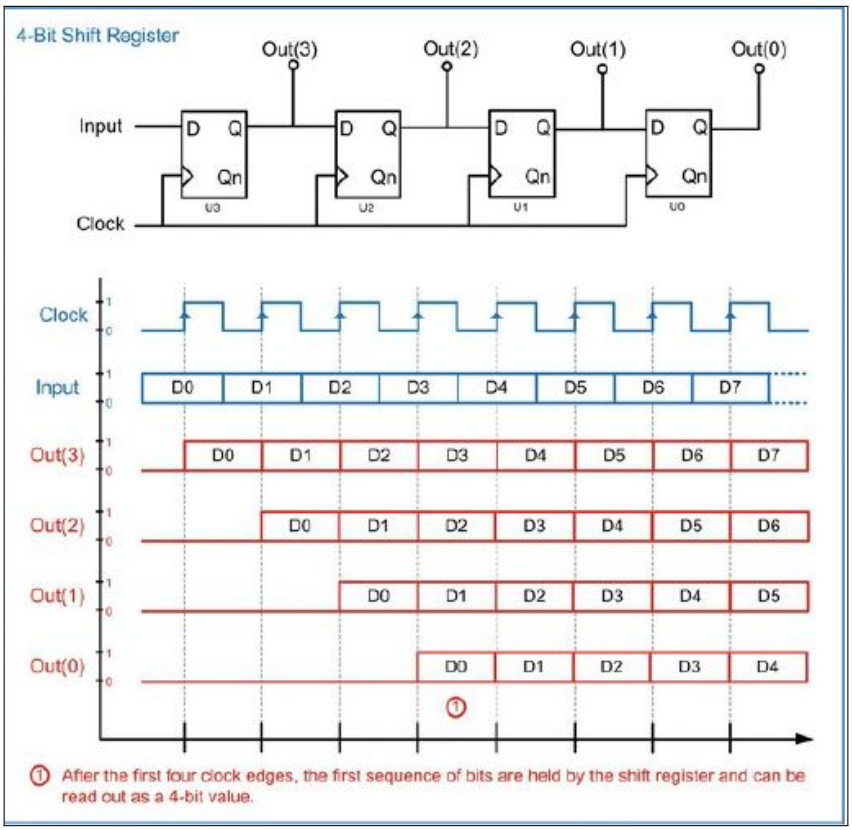


Figure 10.20: Shift Register

- This is a 4 bit shift register
- An application: converting serial data to parallel
- If we look after 4 clock cycle, we have $D_0D_1D_2D_3$
- If we want the next 4 bits, we wait again 4 clock cycles (since it is a 4 bit shift register)

Chapter 11

Finite State Machine

Finite state machine (FSM) are circuits that can be used to design some processes in which we need to remember the logs of past inputs and current input to take some action and produce some output.

11.1 Window Example

In Figure 11.1, we have some example about some design which open and close a window, using 1 switch button, and a motor which do the open or close action

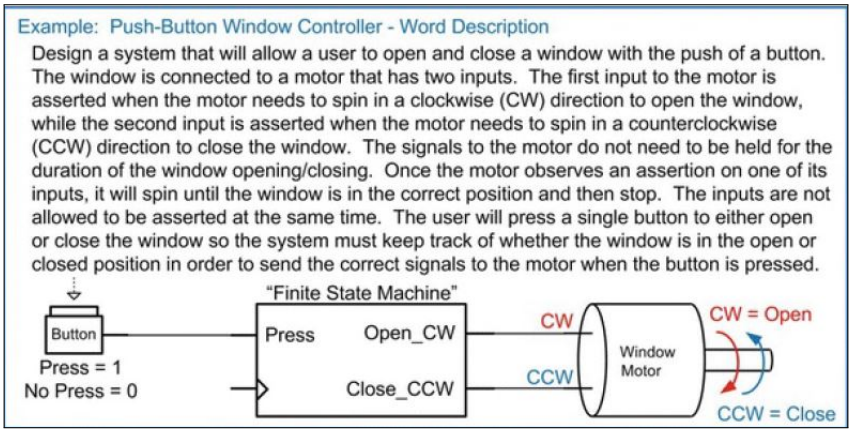


Figure 11.1: FSM for window example

- Word description: In order to design the system, we need to know on which state the window is: if we want to open the window for example, it should be closed (and vice versa). This motivates the use of FSM

11.1.1 State Diagram and Transition tables

- State Diagram: it is the phase where we start to design our FSM (we don't care about circuits implementation yet here)
- State Transition table: it is simply the state diagram in a tabular form

The state diagram along with its associated state transition table is shown in Figure 11.2.

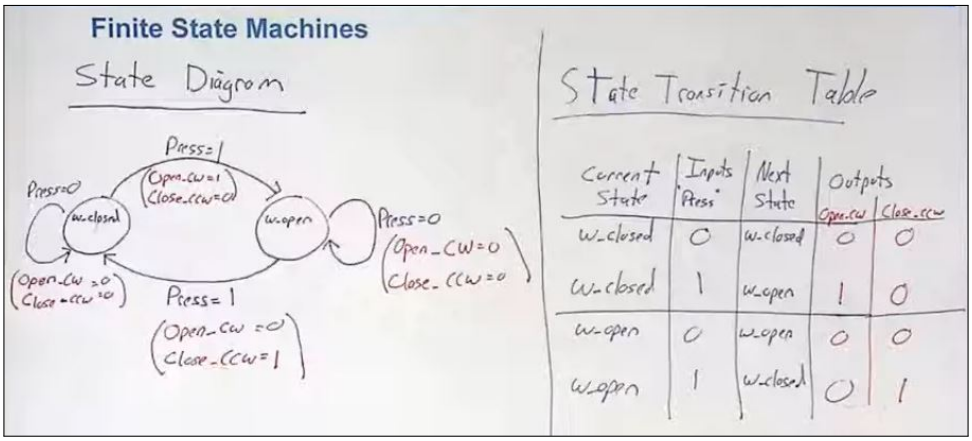


Figure 11.2: State Diagram and Transition table

- There 2 states (which corresponds to our desired outputs:) open and close
- We make transitions (the arrows) according to input we get (in this case the input is the push button)
- If we took w-closed state: if the button is unpressed (0) we remain at the same state, and if it is pressed we move to next state (w-open)
- Same concept when we take w-open state

11.1.2 Logic Synthesis for a FSM: Melay and Moore machine

Once the behavior of the FSM is specified, we can proceed for logic synthesis. There are 3 components for a state machine (shown in Figure 11.3):

1. Next state logic: combinational system which produce next state signal based on current state + input
2. state memory: implemented using D-flip flop which move the next state signal to the Q output, and hence it becomes a current state.
    - At every rising edge, the current state will be updated using the next state logic in 1 (based on input + current state)
3. Output logic: combinational logic that create the output of the system. This block take the current state as input, and whether the machine is Moore or melay, we take the input signal.

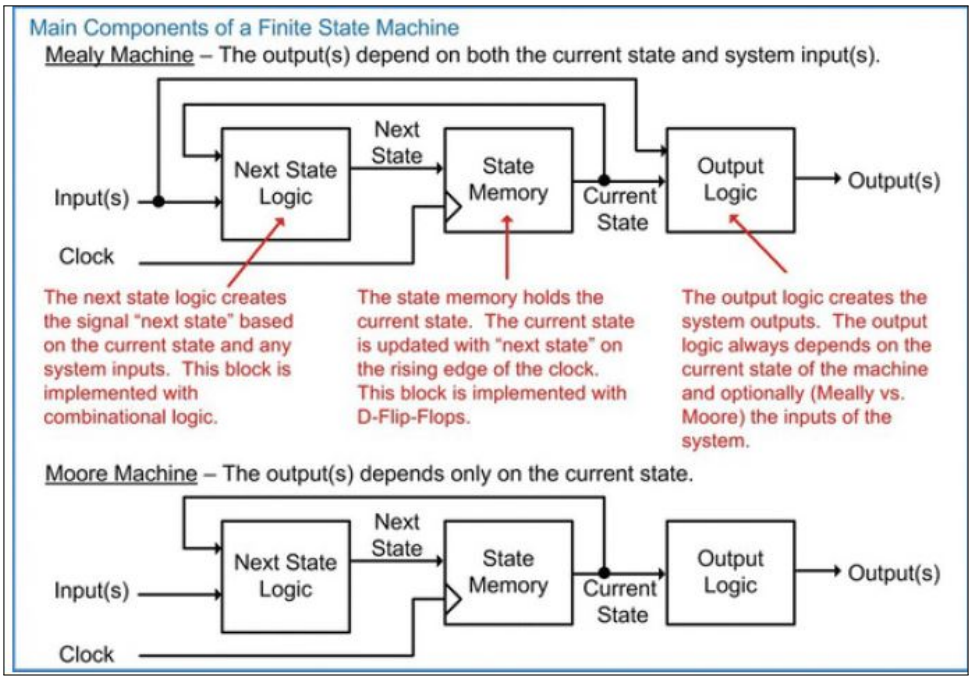


Figure 11.3: Logic Synthesis for FSM: Melay vs Moore machine

11.1.3 Encoding States

The state diagram shown in Figure 11.2 contains ***descriptive names*** (w-open and w-closed). These names need to be encoded in order to be implemented in a logic circuit. There are 3 types of encoding shown in Figure 11.4.

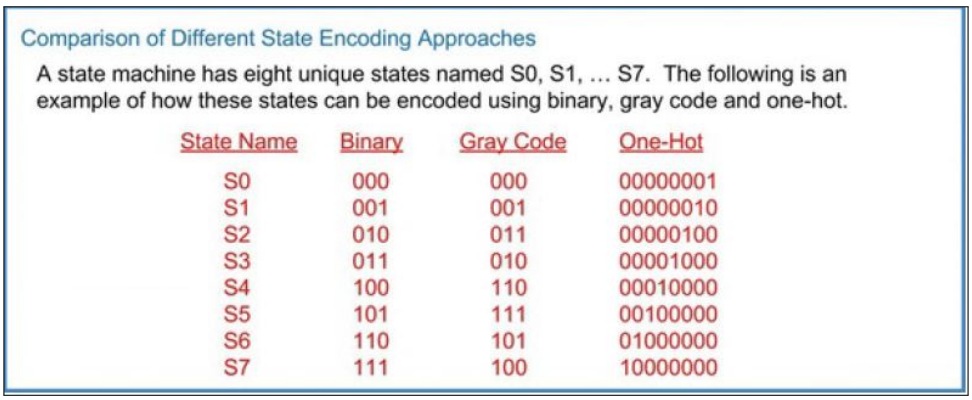


Figure 11.4: State Encoding Types

- For binary and gray: number of bits n follow the rule: 2^n = # of states
- Gray encoding: only 1 bit change between each transition
  - Require less power then normal binary coding since we are changing 1 bit only
  - General rule for creating n-bit gray code shown in Figure 11.5

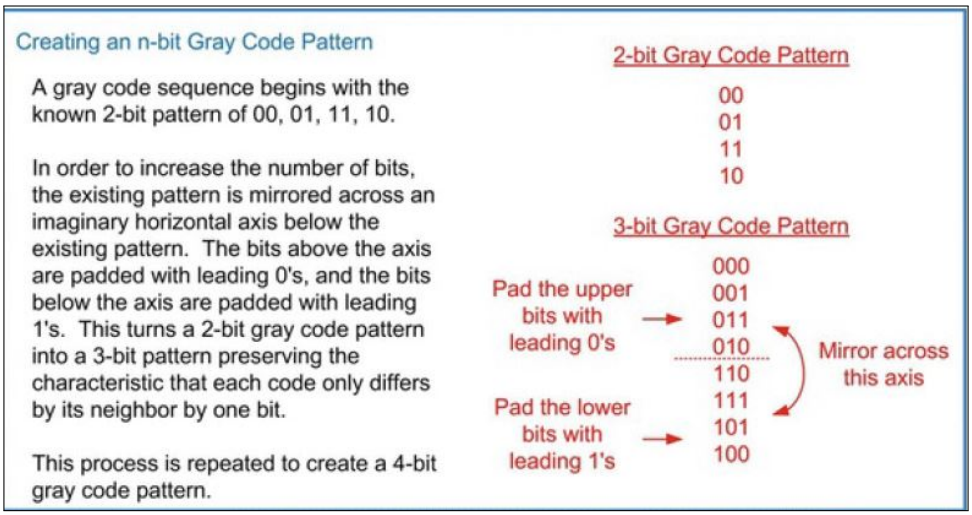


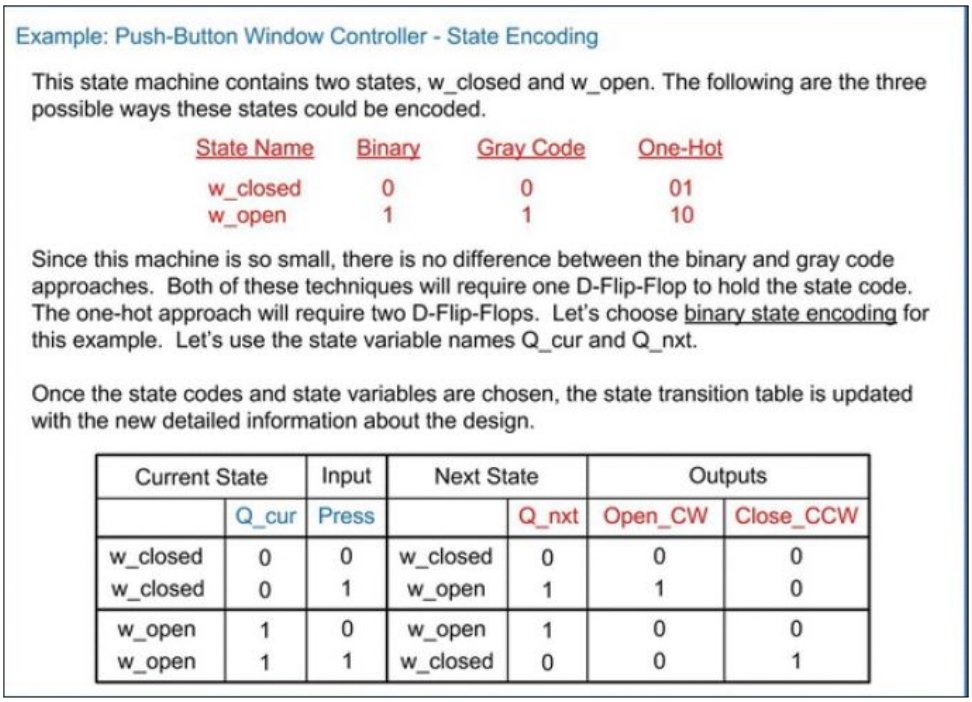
Figure 11.5: State Encoding Types

- For 1 hot: number of bits = number of state

11.1.4 Synthesising the Window Example

Now let's get back to the window design, and try to implement the state diagram/table in Figure 11.2.

First, we show the state coding using different code, and we will choose the binary code as our option for this example.
Once the state code is done, the transition table need to be updated to incorporate the coding details, as show in Figure 11.6



Example: Push-Button Window Controller - State Encoding

This state machine contains two states, w_closed and w_open. The following are the three possible ways these states could be encoded.

State Name	Binary	Gray Code	One-Hot
w_closed	0	0	01
w_open	1	1	10

Since this machine is so small, there is no difference between the binary and gray code approaches. Both of these techniques will require one D-Flip-Flop to hold the state code. The one-hot approach will require two D-Flip-Flops. Let's choose binary state encoding for this example. Let's use the state variable names Q_cur and Q_nxt.

Once the state codes and state variables are chosen, the state transition table is updated with the new detailed information about the design.

Current State		Input	Next State		Outputs	
	Q_cur	Press		Q_nxt	Open_CW	Close_CCW
w_closed	0	0	w_closed	0	0	0
w_closed	0	1	w_open	1	1	0
w_open	1	0	w_open	1	0	0
w_open	1	1	w_closed	0	0	1

Figure 11.6: State Encoding Types

Now, we can use synthesis the next state logic and output logic (show in Figure 11.7 and Figure 11.8) using combinational design technique encountered in chapter 5.

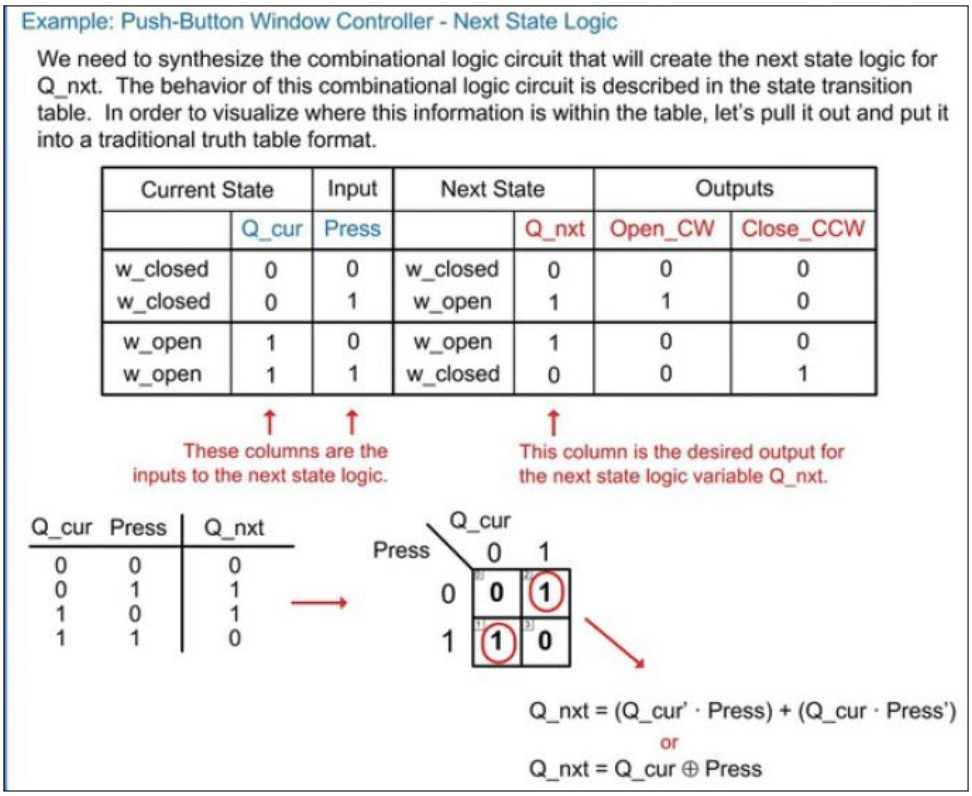


Figure 11.7: Window example: next state logic

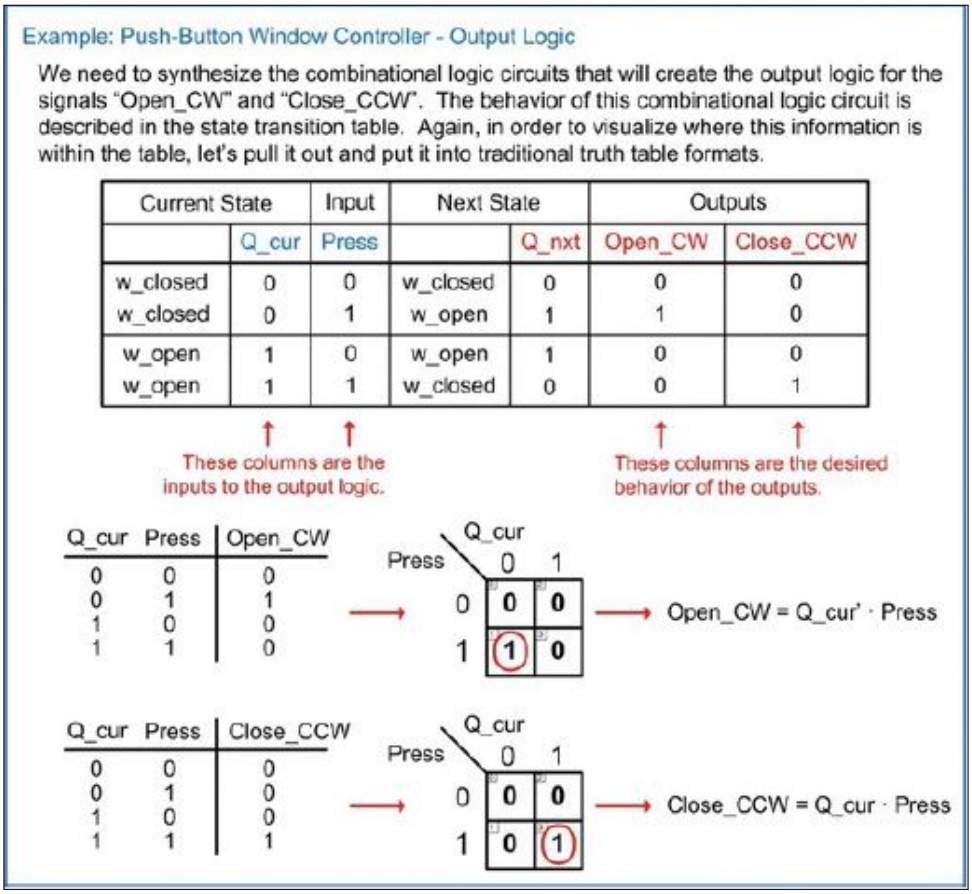


Figure 11.8: Window example: output logic

11.1.5 Recap of FSM design process

11.2 Vending Machine Example

In Figure 11.9, we have the word descriptions of the required vending machine system.

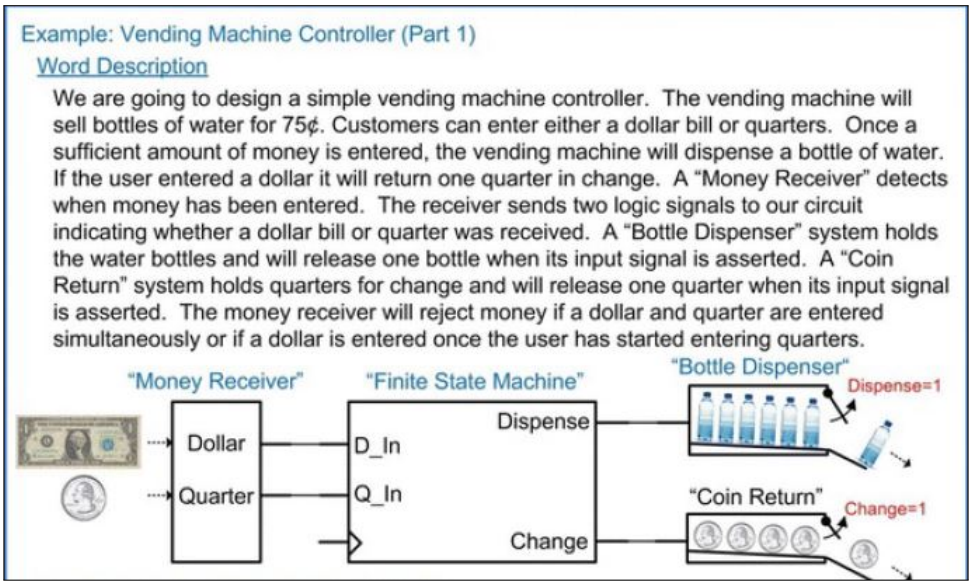


Figure 11.9: Vending machine words description

- We sell the bottle for 75 cents
- In the front end, we have a machine that accepts a dollar and a quarter
- If the machine only accept 1 dollar, we don't have to use a FSM, it would be a combinational logic (to do, actually 2 because we need to return 25 cents)
- But because we accept quarter, we need to track whether we have 1,2,$\cdots$ quarter

The state diagram and transition table are shown in Figure 11.10.

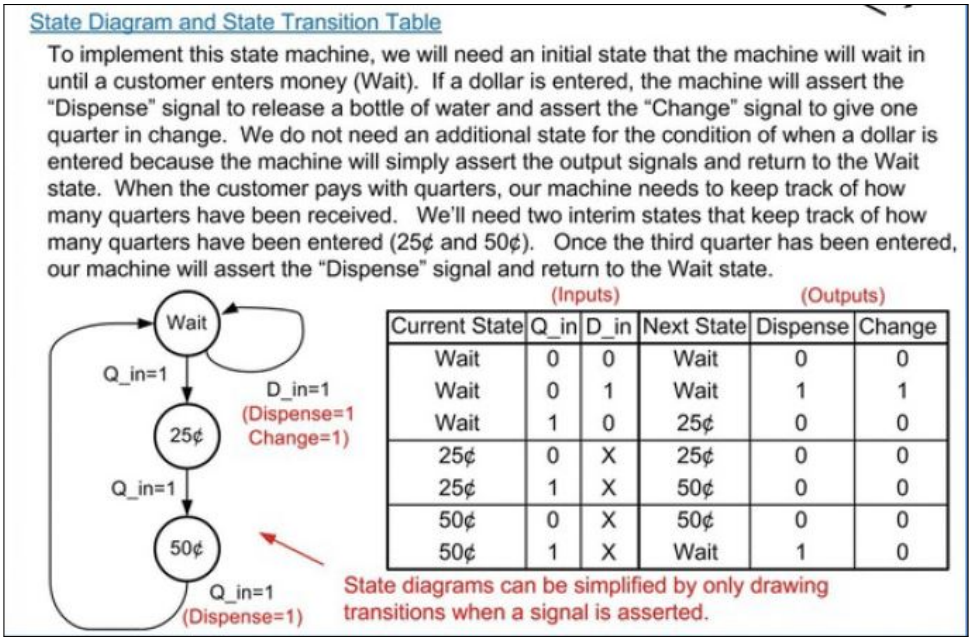


Figure 11.10: Vending machine state diagram and transition table

- Notice that when current state is at 25 cent or 50 cent, we don't care if we have 1 dollar as input
- For now we don't handle the situation when 1 dollar and 1 quarter are inputted simultaneously

11.2.1 Coding State Memory and Updating Table

Now once we have the transition table, we assign codes to states. We choose binary coding. The updated transition table with the codes is shown in Figure 11.11.

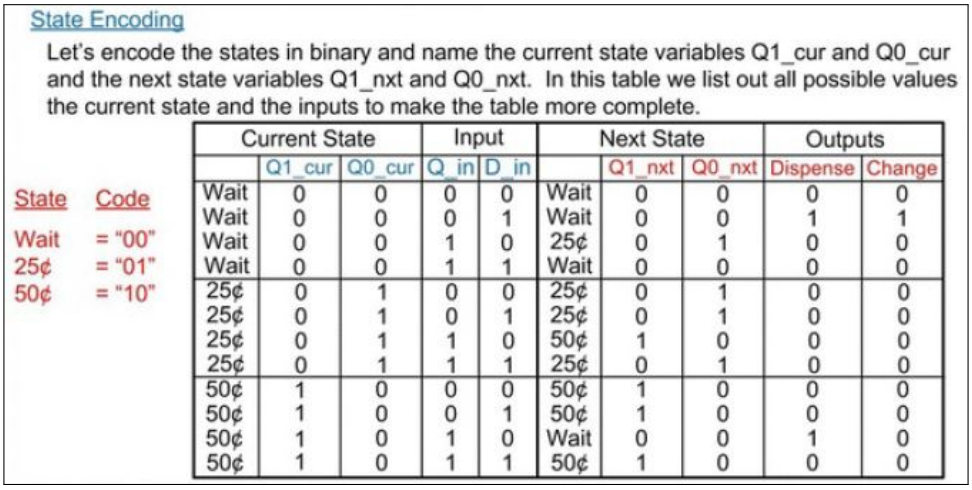


Figure 11.11: Vending machine state diagram and transition table

M: Vend-
machine
thesising

11.2.2 Synthesising the circuits

I will do later when implementing in verilog

Bibliography

[1] B. J. LaMeres, *Introduction to Logic Circuits and Logic Design with Verilog*, 1st ed. Springer Publishing Company, Incorporated, 2018.

[2] W. Kleitz, *Digital Electronics: A practical approach with VHDL*. Pearson, 2012.