ELEC 3310 Notes

Student HONG, Lanxuan
SID 21035307

1 Combinational Logic

1.1 Encoded Numbers

Our familiar **decimal numbers** are encoded in a base of 10, meaning that each digit position is weighted by a factor of 10. Similarly, the **binary system** has a base of 2, meaning that each digit position is weighted by a factor of 2. Fig.1 shows how binary numbers are converted to decimal numbers, especially noticing negative powers.

Positive Powers of Two (Whole Numbers)									Negative Powers of Two (Fractional Number)					
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
256	128	64	32	16	8	4	2	1	1/2 0.5	1/4 0.25	1/8 0.125	1/16 0.0625	1/32 0.03125	1/64 0.015625

Figure 1: Binary System

In any number system, a number with larger index (left places) always has greater significance than the number at right positions. We use **MSB**(Most Significant Bit) and **LSB** (Lease Significant Bit) to represent their influences to the number while showing the position. There are more than two ways to encoded numbers, including **hexadecimal** and **octal**. They all carry different amount of information with same digit amount. For example, a 4-bit binary number carries 2^4 decimal numbers, which is from 0 to 15.

Another interesting way to encode numbers is by **BCD**, meaning binary coded decimal. It uses all the binary ways to represent each number from 0-9, and use the binary rule to combine them, meaning that each bit in decimal takes 4 bits in binary to represent. The conversion between BCD and binary can be done similarly as the binary system by weights. The only difference is, instead of assigning 2^n weights, they are assigned 1,2,4,8, and then 10,20,40,80.

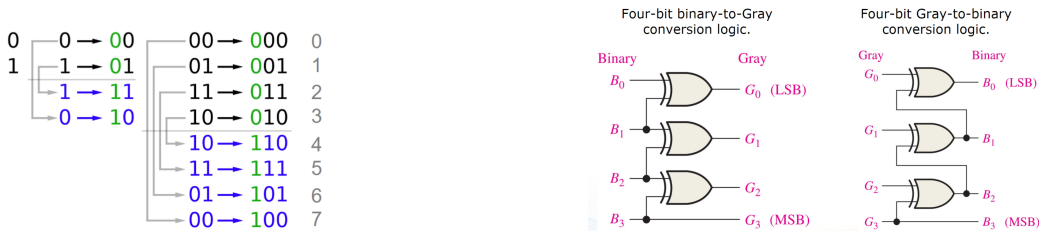


Figure 2: Gray Code

Apart from the weighted code above, an unweighted code, known as **Gray Code**, is also popular in digital circuit design. It is also known as reflected binary code for the way it was constructed. One important property of gray code is that for all adjacent numbers, when changing from k to k+1, only 1 digit will be changed. This avoid dramatic changes in terms of signals. The special construction method is shown in Fig.2.

Gray code can be smartly generated from binary code. For **binary to gray**, we simply need to keep the MSB, and sum up the adjacent digits to get the gray. For **gray to binary**, take the sum of the last binary result and the current digit of gray code to get the binary digit.

1.2 Truth Table

Truth table is an effective way to represent the combination logic of several binary inputs. The output literal(binary number) is related to input literals through a logic function. In terms of abstraction, we can do Boolean algebra for the combination of logic while in terms of circuit, we are using **signals** to convert literals and **gates** (see Fig.3) to implement logic function.

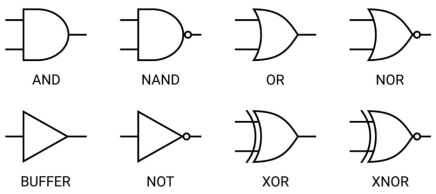


Figure 3: Gates

The key questions to solve in this system are:

- How to find the combinational logic from given inputs and outputs(truth table)?
- How to do Boolean Algebra to simplify the logic?
- How to design a combinational logic circuit and input messages from abstraction result?

1.3 Adder

Before starting to look at each questions separately, we will use a very fundamental but useful combinational logic, which is adder, to explain the whole process. The most basic adder circuit is the **half adder circuit**. It takes two inputs and gets two output (**sum** and **carry**). According to its truth table (see Fig.4), we can directly observe that $S = A \oplus B = A\bar{B} + \bar{A}B$ and $C = AB$. We can also easily derive the logic behind it, meaning that the sum is the LSB and carry is the MSB of the binary addition (**Not Boolean!**) of A+B.

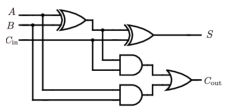
A more advanced version of half adder is the **full adder circuit**. Instead of taking only 2 inputs, it takes a third input C_{in}, adding the carry part of the previous adder and become a 3-input circuit. Based on the observation of half adder, we can add the the third input to the result of A and B. The sum part is still obvious as $S = (A \oplus B) \oplus C_{in}$ while the carry part isn't as obvious as before due to it is influenced by the "carry" from the sum part. The logic expression is $C_{out} = AB + (A \oplus B)C_{in}$ and we will derive it later. The important part here is to show how the logic gate is built from logic expression. The key idea is to **build the inner logic first!** (as shown in Fig.5)

As an engineer, we don't need to build everything from scratch. Actually, think about how a full adder is constructed, we can easily find that it is a combination of 2 half adders! We build a half adder with input A and B, and then make the sum of A and B as the input of the second half adder while C_{in} is the other input. It is trivial that the sum of the second half adder is the total sum, while the carry of the second half adder will be merge to the carry of A and B.[1]

[1]The first half adder will deal with the current digit while the second will deal with a combinational result of last digit with the current digit. The carry of a less significant digit can only be equally weighted as the input of a

Input		Output	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 4: Half Adder



Inputs			Outputs	
A	B	C_{in}	S	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 5: Full Adder

Inspired by this, we can create more adders based on the fundamental half adder and full adder. For example, a 4-bit parallel adder (also known as "Ripple Carry Adder") using 4 full adders in parallel to do bit-wise addition. From the Fig.6, we can see how carries are rippled from a low stage to a high stage.

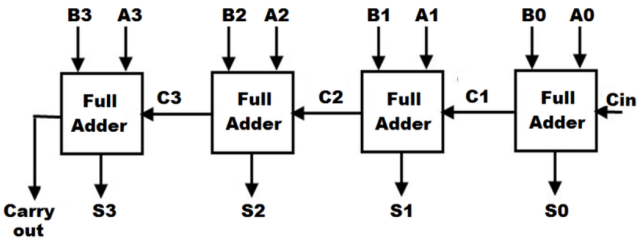


Figure 6: 4-Bit-Parallel Adder

However, for a number with large bits, say 100 bits, this ripple carry adder becomes very slow since each sum and carry output are not finalized until the input carry arrives, which will cause a time delay in the whole process. In order to speed up the addition, the **look-ahead carry adder**(see Fig.7) is designed. The base process of this look-ahead carry adder is:

- $(C_{in} = 0)$The carry will be generated if both inputs are 1: $C_g = AB$
- $(C_{in} = 1)$The carry will be propagate from an input carry to an output carry(whether generated by A and B or combined with input carry)if the sum of A and B is 1: $C_p = A+B$
- The final output is 1 if either of the above happens: $C_{out} = C_g + C_p C_{in}$

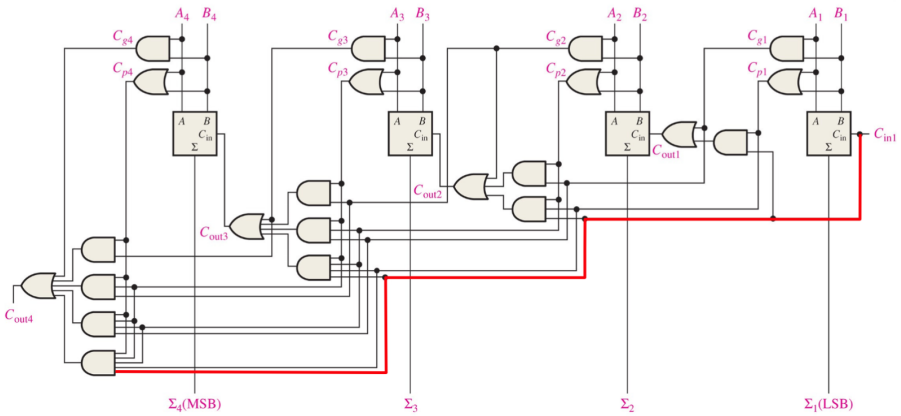


Figure 7: 4-Bit Look Ahead Adder

more significant digit, it will directly change the sum of current digit, and will indirectly add a factor into the carry part (If the sum of $A \oplus B$ and C_{in} is 10, meanwhile the carry of A and B must be 0, cannot be 1).

1.4 Boolean Expression From Truth Table

Now, after a short introduction on the adder circuit and how the whole process is combined, let's look at more details. Recall the key questions from the beginning, the first one is, how to find the Boolean expression from truth table. From the previous example, some easy combinational logic can be easily observed from the truth table. We can build a base case (all 2-input logic functions) for the logic functions for the convenience of computation. [2] Besides the gates introduced, the comparator logic function, including $A > B, B > A, A \geq B, B \geq A$ also need notice. Take $A > B$ as an example: the only possible case for $A > B$ is A=1 while B=0, so we can easily derive the logic as $Y = A\bar{B}$ and same to others. Here are 2 ways to identify the logic function:

- **SOP**: identify all the 1s, use product to **AND** all literals to get the 1 (only 1 way to make it 1), and then use sum to **OR** all the logic expressions.
- **POS**: identify all the 0s, use sum to **OR** all literals to get the 0(either way for 0 would lead to a 0), and then use product to **AND** all the logic expressions.

The SOP method, also known as **MINTERMS expansion** where each product term is a minterm. The minterm is the term that is the product of each inputs, which means that it has only 1 combination to make it true, otherwise false. The POS method, known as **MAXTERMS expansion** which use maxterm (sum of each literals) and find their sum. The final result would only be 0 if at least one combination of sum is 0.

Here is an example of how to use SOP or POS to find the Combinational logic from truth table. Recall the boolean expression of the full adder, which was given directly without deriving it. (see Fig.5) From its truth table of C_{out}, we can find 4 1s and 4 0s. From the perspective of SOP, we find sum of the all minterms of true results and simplify them.

$$\begin{aligned}
&\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \\
=&(A \oplus B)C + AB \ \textbf{(One way of expression)} \\
=&\bar{A}BC + A\bar{B}C + AB + AB = B(\bar{A}C + A) + A(B + \bar{B}C) \\
=&AB + BC + AC \ \textbf{(Another way of expression)}
\end{aligned}$$

From the perspective of POS, we find the product of all maxterms of false results and simplify them.

$$\begin{aligned}
&(A + B + C)(A + B + \bar{C})(A + \bar{B} + C)(\bar{A} + B + C) \\
=&(A + B)(C + AB + \bar{A}\bar{B}) \\
=&AC + BC + AB
\end{aligned}$$

This is a complete process of how logic circuit is observed, derived and designed. As more circuit will be introduced, this process will be repeated for several times and more insights would gain from it.

1.5 Comparator

The 2-input simple comparator was introduced, based on that logic and the idea of combining simple logic together instead of building from scratch, we can create a more complex comparator

[2] There are a total of 16 combinations of binary results from a binary input of A and B. Most are already introduced as basic logic gates, including **AND, NAND, OR, NOR, XOR, XNOR** and the very basic cases **ZERO, ONE, A, B, NOT'A', NOT'B'**

as well. Let's start from the 4-bit magnitude comparator. In this case, bit-wise operation will be done using 1-bit comparator. The key idea was actually simple that we just start from comparison of MSB and move to lower order bits. The following Boolean logic expression shows an example of the process. (circuit design see Fig.8)

$$
\begin{aligned}
X_{A<B} =& (A_3 < B_3) OR \\
& (A_3 = B_3) AND (A_2 < B_2) OR \\
& (A_3 = B_3) AND (A_2 = B_2) AND (A_1 < B_1) OR \\
& (A_3 = B_3) AND (A_2 = B_2) AND (A_1 = B_1) AND (A_0 < B_0) \\
X_{LT} =& (\bar{A}_3 B_3) + \\
& (\overline{A_3 \oplus B_3})(\bar{A}_2 B_2) + \\
& (\overline{A_3 \oplus B_3})(\overline{A_2 \oplus B_2})(\bar{A}_1 B_1) + \\
& (\overline{A_3 \oplus B_3})(\overline{A_2 \oplus B_2})(\overline{A_1 \oplus B_1})(\bar{A}_0 B_0)
\end{aligned}
$$

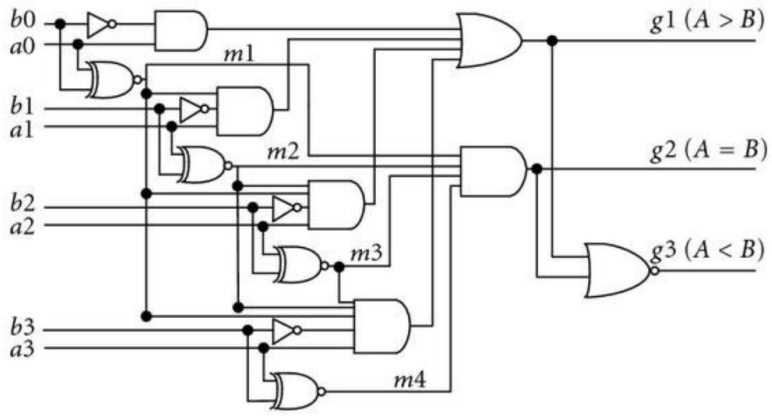


Figure 8: 4 Bit Magnitude Comparator

Another way of comparison is by subtraction. From algebra we know that subtraction can come from addition. This come to a very natural idea that a complement (negative expression) is needed for more complex calculations.

1.6 Signed Binary Numbers

One intuitive way to do subtraction is tho get a complement from a full binary numbers, which is a number with all 1s. Suppose such binary number A, while $S = A + (-B)$, then S is the **first complement** of B, represented by $\overline{B}$. The trick to get a first complement of a digit is simply by **reversing** all the bits. Another way to do the complement is, instead of using such A that actually exist, which would result in a 0 in B's complement [3], we add 1 to A and make the complement non-0. This is known as the **second complement**, where $S = \overline{B} + 1$. The trick for second complement is to invert all the bits, and then add 1.(Add 1 means invert all the last 1s to 0 until meeting a 0, invert it to 1) For a n-digit number, the MSB represents the sign, where 0 means positive number and 1 means negative number. The magnitude of negative number comes from its complements.

[3]The problem here is that there will be a positive 0 and a negative 0 which contradict with the common sense

A less common way is by **sign and magnitude**, which use the MSB to express the sign and other digits for magnitude. In the following addition and subtraction where complements will be used, **only 2nd complement will be applied** since it is a more common method. Here are 3 examples of signed binary addition.

Pay special attention to OVERFLOW condition!!!

- 00001000(8)+11111101(-3)=1 00000101(5), the extra digit 1 shows that it is positive. Discard the 1.
- 00010000(16)+11101000(-24)=11111000(-8), MSB(1) means negative, use 2nd complement for magnitude.

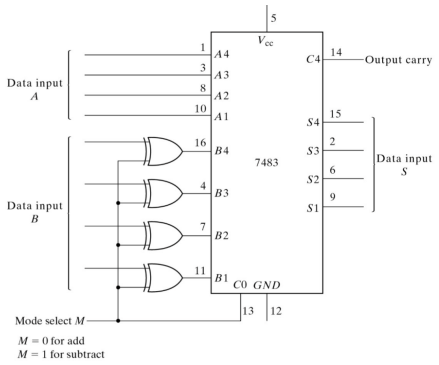


Figure 9: XOR As An Inverter

The following Fig.9 shows a brilliant idea of using XOR gate as inverter, which does 1s complement, and 7483 IC add 1 to the result, resulting in a 2s complement.

1.7 Decoder

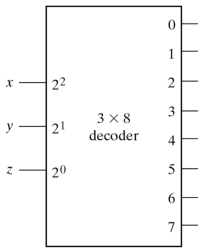
What a decoder does is that it convert the message from binary bits to what human can understand(usually decimal). The decoding process **CANNOT** change the amount of information it is carrying. The decoding process is generating a one-hot code from binary to output terminals, meaning that only **one terminal** is different from others (usually gets **HIGH**). We can also observe that each output is actually a **minterm**[4], which means we can also use the decoder to represent any 3-input logic function by adding **selector** at output terminal.(see Fig.10) Minterm outputs can also be Maxterms with an inverter.

Here are some common questions about decoder:

- Output always HIGH? No, output can be active LOW or active HIGH, depend on the design and application. Usually high, and the one with bubble(inverter) means low.
- What does ENABLE do? Activate the circuit. In sequential logic, the ENABLE is the clock.

The following example shows how (powerful) the enable input can be applied. One concept was recalled more than once, which is building a more complex structure from the fundamental

[4]Based on my personal understanding, minterm carries more information than maxterm, which means that to express a certain and sure information(a correct decoding), minterm should be used for the certain result decoding. In fact, 8 is the max amount of information that 3-digit numbers can carry, then converting it into 8 different "meanings" must use all the information it carries.



Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_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

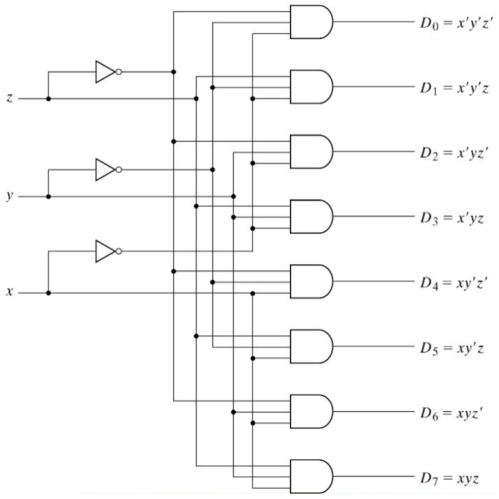


Figure 10: 3-8 Decoder

elements. Here we can build a 4-16 line decoder from two 3-8 decoder featuring Enable inputs. From Fig.11, The Enable input will give 2 different (**ALWAYS DIFFERENT**) signals to either 3-8 decoders, which will 'shut down' either decoder. Note that the first decoder is in charge of the 0-7 in one-hot encoding, while the second decoder is in charge of 8-15. (This is different from a single 3-8 decoder with an enable signal, because a 3-8 decoder will only get 8 outputs and do one-hot encoding from 0-7. It will be all 0s when it is shut down.) We can also build more complex circuit.(see Fig.12)

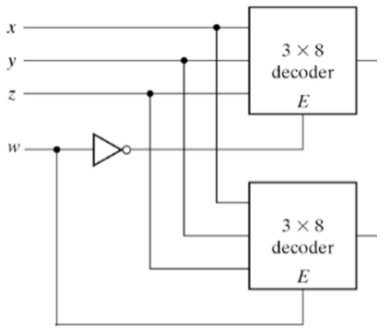


Figure 11: 4-16 Decoder

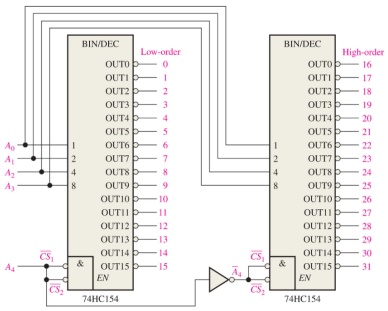


Figure 12: 5 Bit Decoder

One application of such decoder is **BCD-to-Decimal Decoder**, which means converting binary coded decimals(by 4 bits) to decimal numbers. It takes a 4-digit number as input and generate 10 outputs. Another more commonly used decoder, is instead of getting a direct output of decimal, it should be interactive with human, meaning that each output should do a certain "action". For example, the **BCD-to-7 Segment Decoder** takes a 4-bit number representing decimal numbers to drive 7-segment displays.

1.8 Encoder

From its name, encoder converts human language(usually decimal) to computer language(usually binary). It works exactly the opposite as the decoder. For a simple encoder, only 1 decimal input will be given and each digit takes the OR gate of all possible decimal numbers that would make it 1 (true). Under some special cases where more than 1 inputs are given, we will use the **priority encoder**, where the priority of each pins are coded, and it will take the input with higher priority as the final input.

1.9 Multiplexer & Demultiplexer

Multiplexer and demultiplexer are used as an electronic switch connecting inputs with outputs, where the multiplexer switches many inputs to one output and demultiplexer switches one input to many outputs.

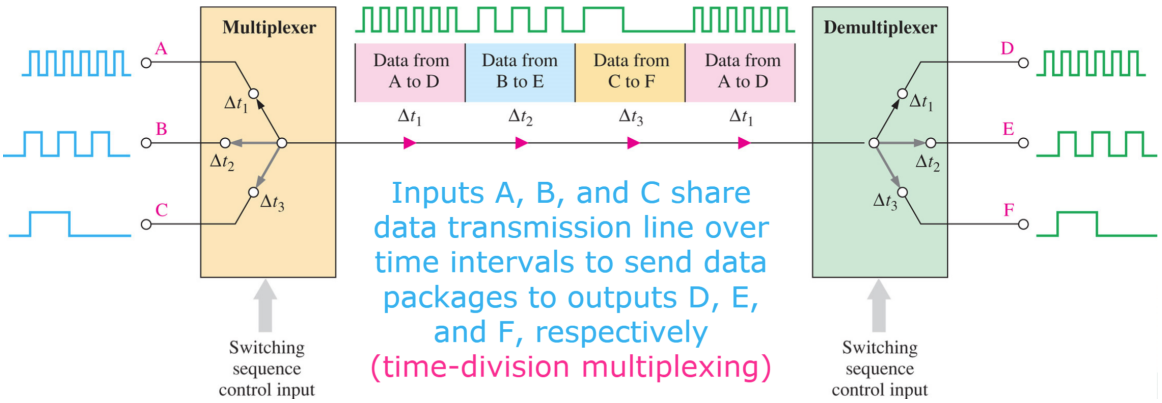


Figure 13: Multiplexer and Demultiplexer

For a **4-to-1 MUX** which takes 4 data and 2 **data select** as inputs while generating 1 output as shown in Fig.14 . The 2-digit data select can represent 4 groups of selection result, where Y would be D_0 if $S_0S_1 = 00$, and similar for other cases. We can write the minterms expressions as:

$$Y = D_0\bar{S}_0\bar{S}_1 + D_1S_0\bar{S}_1 + D_2\bar{S}_0S_1 + D_3S_0S_1$$

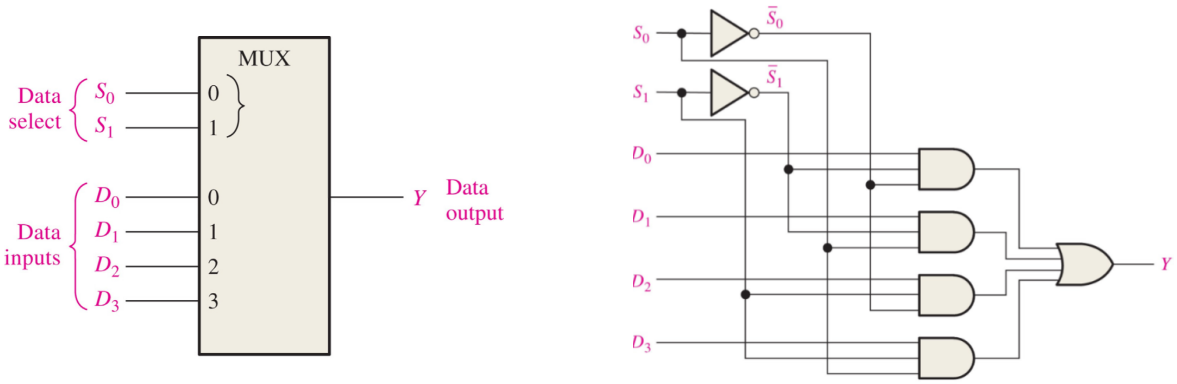


Figure 14: Multiplexer

Since MUX takes the minterms, we can also implement logic functions through the multiplexer. As both of them takes the minterms, what we need to do is simply take the inputs of function as the select input. For example we can implement a 3-input function using an 8-1 MUX by connecting the inputs as select and "select" certain combinations as functional outputs.

Sometimes the MUX might not match all the functions, especially when there are more functional inputs than data select inputs. For example, the implementation of a 4-input function using an 8-1 MUX. The solution procedure is to first feed the function inputs to the MUX select inputs **starting from MSB**. Since the 8-1 MUX only has 3 select input, some function inputs(in this case, the LSB) will be left out. In other words, as the MUX does not provide enough component, we need external logic gates to help functional implementation. We will partition

the truth table according to the values of select inputs, and the decimal digit can be part into 8 different parts with same select input for each part. Then we can express each partition in the function $Y = A_0$ and add the A_0 to the extra gate. Using this method, we can implement all logic functions by MUX no matter how many inputs it has.

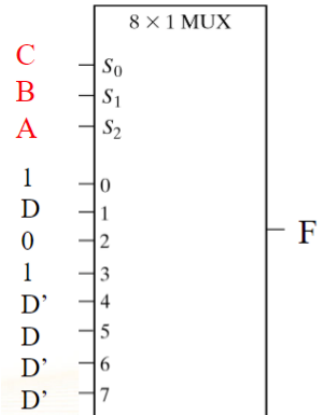


Figure 15: 8-1 MUX for 4-input logic function

Here is a practice question. How to express the 4-input logic function in terms of a 4 choose 1 MUX: $F(A, B, C, D) = \sum m(0, 1, 3, 6, 7, 8, 11, 12, 14)$.

1. Assign A, B to **SELECT** inputs
2. Partition the truth table into 4 parts(for 4 combinations of A, B) with each part 4 empty spaces for the 4 combinations of C, D.
3. Write the combination result of C and D as the data inputs.

When considering the minterms we can think of SOP, functional logic, we should also come to the MUX and decoder. Here is an example of how these elements are combined together in the design: **building a 4-input logic function by 4-1 MUX and 2:4 Decoder**. The idea is, using MUX to partition the truth table as done before, and then use 2:4 decoder to generate minterms for each partition. We can also broaden the application to the multi-bit inputs by either using multi-bit as inputs or more than 1 MUX.

On the other hand, the **demultiplexer** decide which module the input goes. The demultiplexer, using maxterms instead of minterms, can be viewed as the **decoder with enable input**, where the input signal is applied to ENABLE and the select input applied to decoder and assign the input(which is ENABLE) to one of the lines.

1.10 Error Detection & Error Correction

Error can occur in the transfer of digital codes(through the voltage level) due to electrical noise. One way to detect error is to attach a **parity** bit to the code before sending the message to make the total number of 1s either even or odd depending on the system. The error will be detected if there is a change in odd number of bits, which is very likely to happen. But this method will not succeed if there are even number of bit changes, which cannot be detected.

There are 2 process in this detection method: **generate parity** and **detect parity**. Both use XOR gate to count the number of 1s.

It is very common to see **glitches** in the decoder waveforms, which are voltage spike at very short duration caused by propagation delays, as shown in 16. This is not caused by noise but the **design**, where the change of components does not happen at exactly the same time.

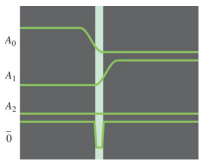


Figure 16: Reason of Glitch

The application of a **strobe pulse** will eliminate glitches, which enables the decoder only during the times when the waveforms are NOT in transition. This design is related to CLOCK.

1.11 Traffic Signal Controller

At the end of the combinational logic part, an example using what have learned so far to develop a control logic for a traffic signal will be presented here. Here is the design strategy:

- The main street green light will stay on for a minimum of 25s OR as long as there is no vehicle on the side street. $(T_L + \bar{V}_S)$
- The side street green light will stay on until there is no vehicle on the side street up to a maximum of 25s. $(T_L V_S)$
- The yellow light will stay on for 4 s between changes from green to red on both the main street and the side street. (T_S)

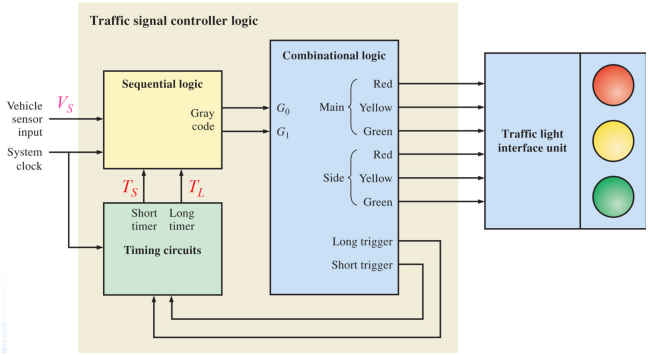


Figure 17: Block Diagram

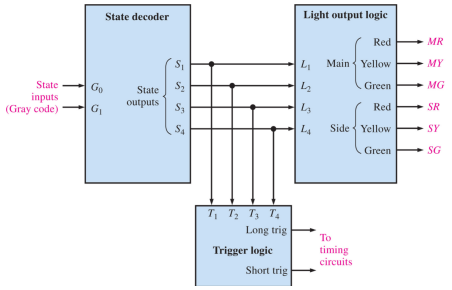


Figure 18: Logic Block

2 Sequential Logic

2.1 A comparison of Combinational Logic and Sequential Logic

The Full Adder will be used as an example to show the main difference. Note that almost all the problems can be solved by both combinational logic and sequential logic, but they have different advantages for different problems.

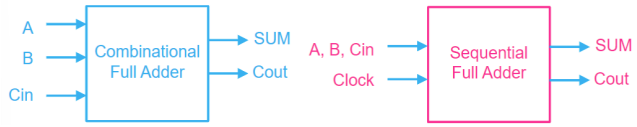


Figure 19: Comparison of Full Adder using Combinational and Sequential Logic

- Data inputs: many for combinational logic and one for sequential logic.
- Control input: Enable for combinational logic(optional) and clock for sequential logic(must).
- Outputs: changes as input change for combinational logic and fixed within the same clock cycle for sequential logic, no matter how input changes, but change with the clock cycles.
- Memory: no memory is needed for combinational logic whose output depend only on current inputs while sequential logic whose output also depend on the past inputs requires memory.

For large bit problems, using sequential logic is much more easier than using combinational logic.

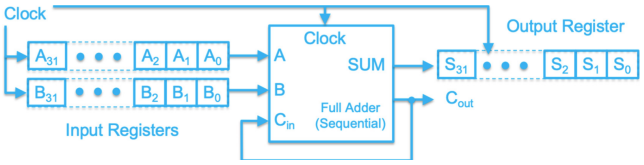


Figure 20: 32 bit Adder By Sequential Logic

2.2 Memory and D-flip-flop

As mentioned in the comparison section, all sequential logic circuit needs memory, which is called **register**, to store the bits. The construction of memory is achieved through **feedback**, which is connecting the outputs of a logic circuit back to its input. For example, we can simply use 2 inverters to store the input. A **clock** signal is required to **synchronized** (control) all registers and blocks to change at the same pace.

Memory component for single-bit storage is called **flip-flop** or **latch**, but there is a difference between flip-flop and latch. We will mainly focus on **D flip-flop**(see Fig.21) where D means data.

A D flip flop has 2 inputs, where D means data and C means clock. Its 2 outputs represent the bit stored, either 0 or 1, following the edge of the clock. The Clock input will **NOT** be explicitly shown in sequential logic later. The triangle symbol indicate the flip-flop is **edge sensitive** and will become a latch which is **level sensitive**. In this truth table, the ↑ means the

Truth table for a positive edge-triggered D flip-flop.

Inputs		Outputs		
D	CLK	Q	$\overline{Q}$	Comments
0	↑	0	1	RESET
1	↑	1	0	SET

↑ = clock transition LOW to HIGH

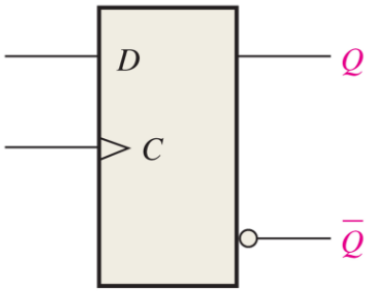


Figure 21: D Flip Flop

output Q changes with the **rising** edge of the clock. The **SET** and **RESET** are states of the flip-flop, where RESET means setting Q to binary 0 and SET to 1.

What a flip-flop does is that it stores a binary bit, where SET stores 1 and RESET stores 0. The following Fig.22 shows the timing diagram (a more common way to express the sequential logic) of a d flip-flop. Fig.23 shows the timing diagram of d latch, by comparison we can tell the difference between level sensitive (clock act as enable input to switch on/off the inputs) and edge sensitive (clock edge act as enable input to switch on/off the inputs).

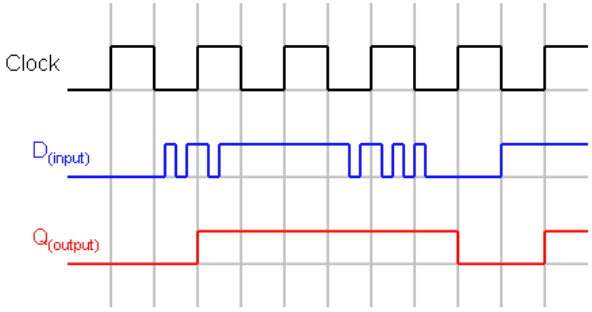


Figure 22: Timing Diagram of D Flip-flop

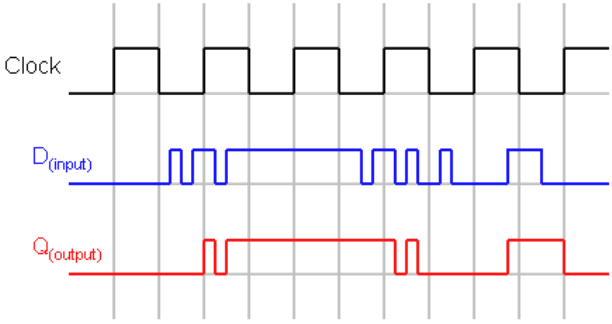


Figure 23: Timing Diagram of D Latch

One important reason why we use clock is to keep all the inputs synchronous. Synchronous means the output changes based on the clock edges, while synchronous inputs means the input have priority and override the synchronous inputs and being independent to the clock edge. The following Fig.24 shows one important application of asynchronous inputs for the system to interact with normal operations. (PRESET set Q to 1 while CLR set Q to 0)

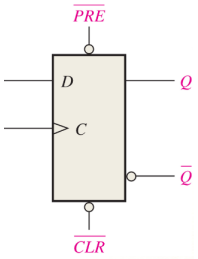


Figure 24: Asynchronous Inputs

Error in flip flop: For synchronous inputs there will be **clock-to-output delay** and for asynchronous input there are **preset/clear-input-to-output delay**. **Setup time** and **Hold time** are also important issues to consider for the flip flop to work as expected.

2.3 State Diagram and State Transition

Looking back into the traffic light problem in the combinational logic part. Recall that the whole logic block of this problem can be separated into 3 parts, which are sequential logic part, timing circuits and sequential logic part. The combinational logic part, which is used to decode the state inputs into the behavior of outputs are solved. In this part we focus on the sequential logic part, which takes the sensor and system clock as input and generate the states.(see Fig.25)

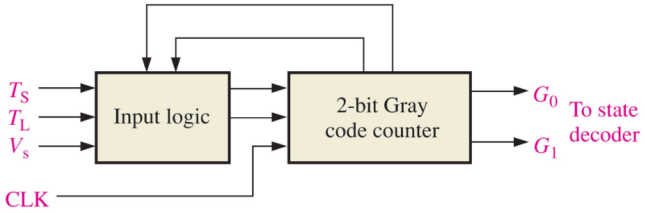


Figure 25: Traffic Light Problem Revisit

For the gray code counter, we can use 2 flip flop with the same clock system to represent 4 different states (2-bit result for 4 states). And notice that in a D flip-flop, the next state is the same as the input data D.

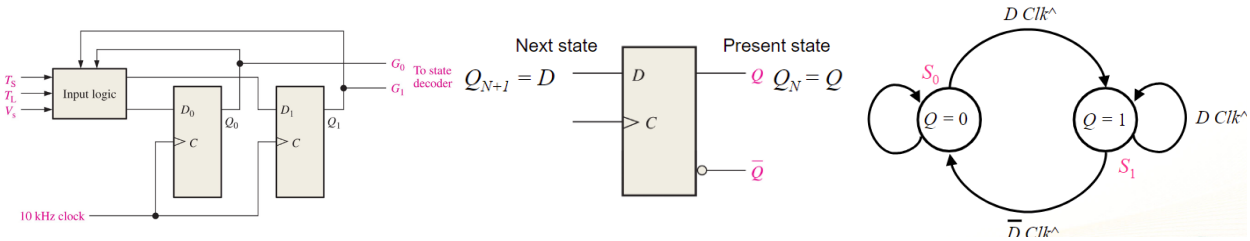
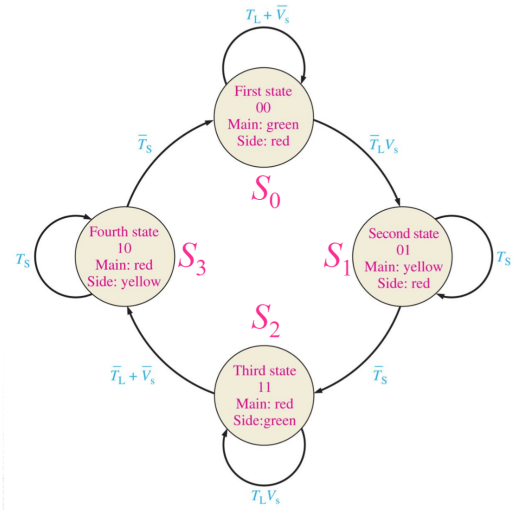


Figure 26: State Diagram of Gray Code Counter

Now we can apply the state diagram to the whole problem and decide the flip flop inputs based on "next state", which means that we can design a combinational logic to decide which combination of input T and V will result in state changes operated by flip flop.



Present State Q_1	Present State Q_0	Next State Q_1	Next State Q_0	Input Conditions
0	0	0	0	$T_L + \overline{V}_s$
0	0	0	1	$\overline{T}_L V_s$
0	1	0	1	T_S
0	1	1	1	$\overline{T}_S$
1	1	1	1	$T_L V_s$
1	1	1	0	$\overline{T}_L + \overline{V}_s$
1	0	1	0	T_S
1	0	0	0	$\overline{T}_S$

Figure 27: Timing Diagram and Transmission Table

3 Finite State Machine (FSM)

For sequential logic, **Finite State Machines (FSMs)** are the mathematical formalism that act similar roles as the truth table in combinational logic. In other word, it refers to a sequential circuit having a limited number of states occurring in a prescribed order.

3.1 Counters

The number of states in a binary counter is called **modulus**, for example, 8 modulus up binary counter is counting from 000 to 111. There are also 3-bit gray-code counter counting from 000 to 100. There are 2 counter types based on how FF are clocked:

- **Asynchronous (Ripple) Counters**: the first FF(LSB) is clocked by the external clock pulse and then each successive FF is clocked by the output of the preceding FF.
- **Synchronous Counters**: the clock input is connected to all of the FFs so that they are clocked simultaneously.

Here is an example to design a 2-bit binary up counter.

1. Specification of the desired circuit A 2-bit binary up counter with no in/output.
2. Create a state diagram.

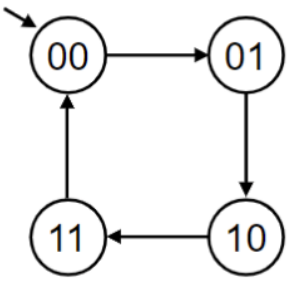


Figure 28: state Diagram

3. Create a state table from state diagram.

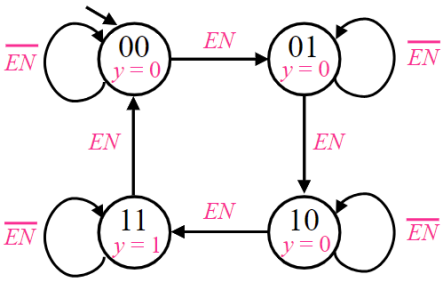
Q_1Q_0 (Present State)	$Q_1^+Q_0^+$ (Next State)
00	01
01	10
10	11
11	00

Table 1: State Table

4. Derive the next state logic expression. For D-Flip flop, next state is the D input.

$$D_1 = Q_1^+ = Q_1 \oplus Q_0$$
$$D_0 = Q_0^+ = \overline{Q_0}$$

5. Implement the circuit by logic.



Q_i : present state, Q_i^+ : next state

Input	Present State	Next State	Output
EN	Q_1Q_0	$Q_1^+Q_0^+$	*y*
0	Q_1Q_0	Q_1Q_0	0/1
1	0 0	0 1	0
1	0 1	1 0	0
1	1 0	1 1	0
1	1 1	0 0	1

Figure 29: FSM Example

We can also add input and output to the counter for multi-purpose function. The following State Diagram and State Table shows a new design with ENABLE input and output of 1 at the last state.

Note that both circuits have repeat blocks. This block is also applicable for n-bit binary up counter while the right most AND gate mark as the end of counting(output).

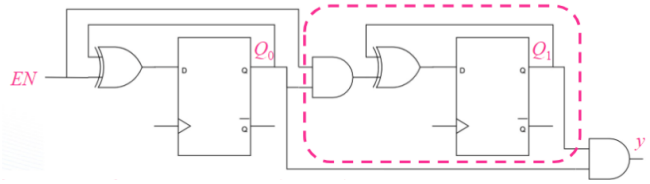


Figure 30: Repeat blocks

The following example (see Fig.31) shows a cascading of synchronous counter, for example, a modulus-100 counter using 2 cascade decade counters. The **cascaded counters**, also named countdown chains, are often used to divide a high frequency clock signal to obtain highly accurate pulse frequencies. Counter 2 is inhibited by the LOW on its CTEN input until counter 1 reaches its terminal state and TC output goes HIGH. Counter 2 goes from its initial state to its second state during the first clock pulse (clock cycle 10) after 1 TC HIGH. And so the DIV 10 means scaling down the frequency of the next counter by 10, i.e., for every 10 cycles for counter 1, counter 2 goes through 1 cycle.

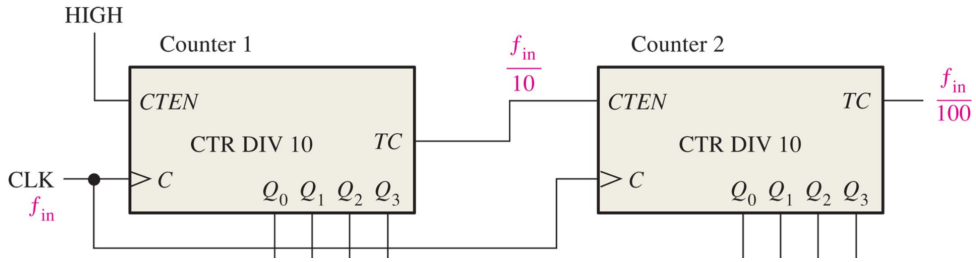


Figure 31: Asynchronous Cascading

The total number of states in this modulus is $10 \times 10 = 100$. In real practice, the overall modulus required is less than that achieved by full-modulus cascading. The strategy is to **PRESET** the state to a certain values instead of starting from 0. This can also be done by adding **truncated sequence** to manually, asynchronously, control the sequence.

One application of counter is to convert parallel data into serial data input.

3.2 Flip Flops

3.2.1 JK Flip Flop

Another type of Flip Flop other than D flip flop is **JK Flip Flop**. There are 4 states naming **toggle, no change, reset, and set**.(see Fig.32)

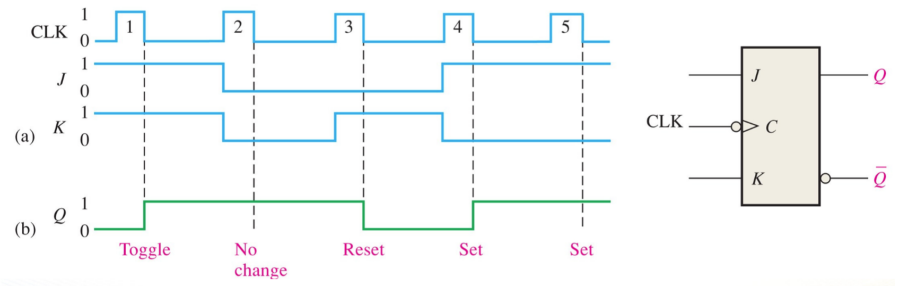


Figure 32: JK Flip Flop Timing Table

Boolean expression of next state Q^+ from Q and J, K can be derived from truth table.

$$Q^+ = \bar{J}\bar{K}Q + J\bar{K}\bar{Q} + J\bar{K}Q + JK\bar{Q} = J\bar{Q} + \bar{K}Q$$

We can also build the JK flip flop from D-FF. We can also build D-FF from JK FF by giving data input D to J and $\bar{D}$ to K.

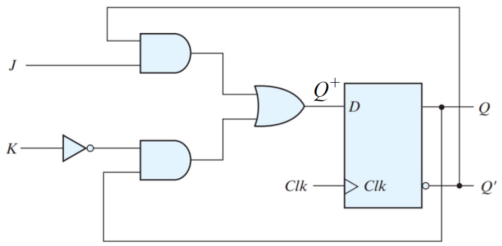


Figure 33: JK FF from D-FF

3.3 Design & Implementation

The general form of a sequential circuit contains clock input, data input, combinational circuit and register(flip-flop). According to whether combinational logic change the output with clock edge or not, FSM can be separated into 2 types(see Fig.34): **Mealy Machine**(asynchronous) and **Moore Machine**(synchronous).

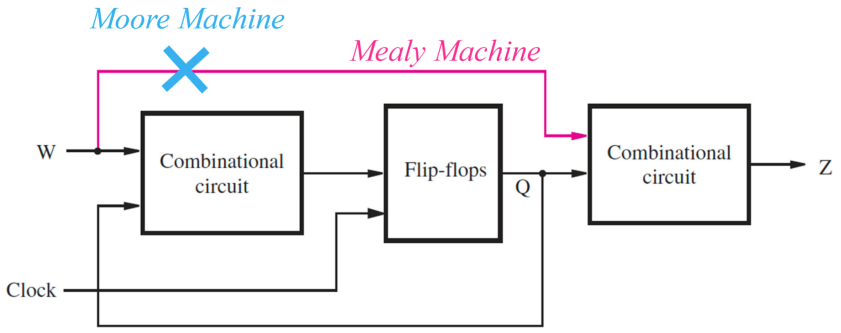


Figure 34: Mealy and Moore

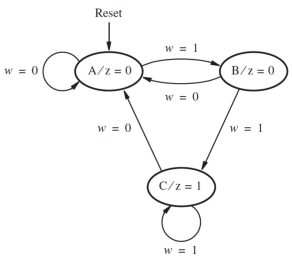
2 more steps should be added to the previous procedure of designing FSM:

1. specification of the desired circuit
2. create state diagram and state table
3. **perform state minimization:** combine the same output from different state
4. **perform state assignment**
5. derive the next-state and output logic expression, implement the design

3.3.1 Design Example of Moore Machine

Sequence/bit change detector: The circuit has 1 input, w, and 1 output z, where z is 1 if w is 1 during last two clock cycles. Changes occur on positive clock edge.

Intuitively we can observe that 11 comes from AND gate, with one input from current state and the other input from last state by taking the input and output of a D-flip-flop. Then take another D-flip-flop to delay the result by once. The problem can also be solved(simplify and minimize the states) by carefully design the state diagram(see Fig.35).



	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

Figure 35: State Design of Moore Machine

3.3.2 Design Example of Mealy Machine

Sequence/bit change detector: The circuit has 1 input, w, and 1 output z, where z is 1 if w is 1 during two **consecutive** clock cycles. Changes occur on positive clock edge.

The following Fig.36 shows the state of the solution by Mealy Machine. Different from the Moore Machine, the output does not depend on the state, which means the output should be written on the arrow, not inside the bubble.

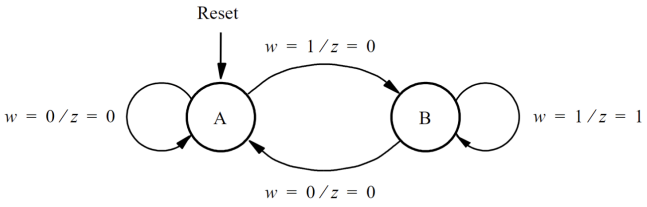


Figure 36: State Diagram of Mealy Machine

3.4 Analysis

A state table consists of 4 sections: **present state, inputs, next state, and outputs**. Treat PS and inputs as input and fill the columns by listing all binary combinations. Derive state and output equations from the circuit. Here is the strategy to figure out "what the circuit does" from a complicated circuit:

1. Derive the logic expression of "output value" and "next state values" from inputs(present state and input).
2. Build the state table from inputs and results.
3. Draw state diagram and do **state reduction**

Here is an example:

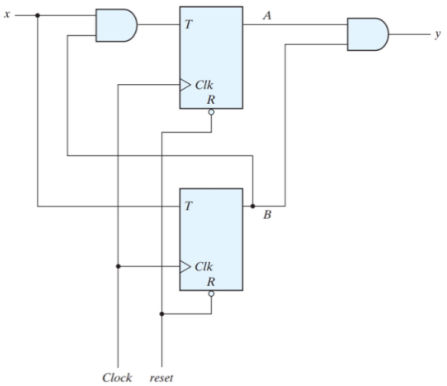


Figure 37: Analysis Example

1. Express the next state and output:

$$Q^+ = T \oplus Q$$
$$y = AB$$
$$A^+ = Bx \oplus A$$
$$B^+ = x \oplus B$$

2. Build truth table(state table) from boolean expression

Present State		Input	Next State		Output
A	B	x	A^+	B^+	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

Figure 38: State Table

3. Convert state table into state diagram.

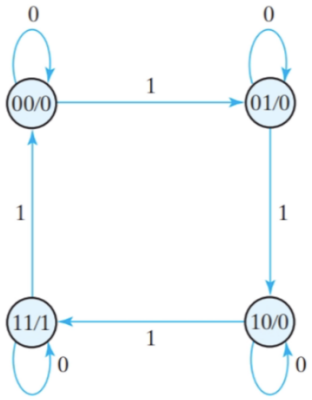


Figure 39: State Diagram

3.5 Registers

A register is a digital circuit with 2 basic functions: **data storage** and **data movement**. It is constructed by a group of flip-flops.

Load input is to control whether the data is to be loaded into the register in the next clock edge. If load value is 0, the register will load in new input data, otherwise it will load the old value, i.e. output value. We can use a D-Flip Flop and a 2-1 MUX to implement load.

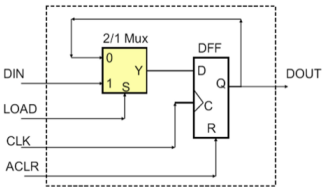


Figure 40: Load

3.5.1 Shift Registers

Shift registers are a type of sequential logic circuit used primarily for the storage of digital data. The shift capability of a register permits the movement of data from stage to stage within the register or into or out of the register upon application of clock pulses.

There are different types of shift registers depending of how input and output performs. For example, serial/parallel-in, serial/parallel out, and shift left/right.

3.5.2 Shift Register Counters

The **ring counter** takes the output of the last flip flop and connect it back to the data input of the first flip flop.

One famous variant of ring counter is the **Johnson counter**. It takes the complement output of the last ff is connected back to the data input of the first flip-flop. This produce 2n states, where n is the number of ffs in the counter.

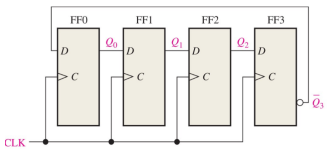


Figure 41: Johnson Counter

Data can be transferred in parallel or serial. Serial transfer consumed less wires and requires using a clock to synchronous every registers.

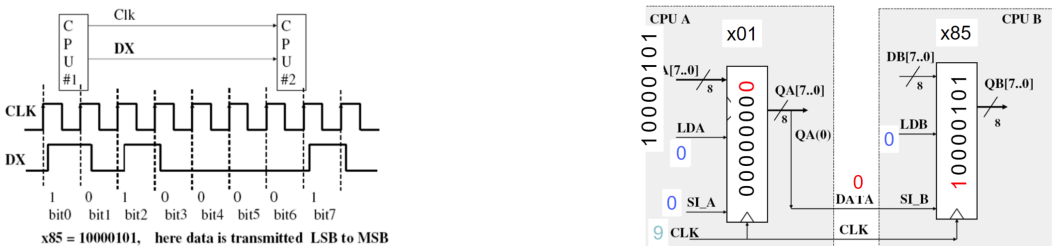


Figure 42: Serial Data Transfer

3.6 Counters Revisit

We can build simple counter by register and adder. For more complex counter design we can also follow the design process of a FSM and create state diagram and state table. Note that for a counter each counting value is a state.

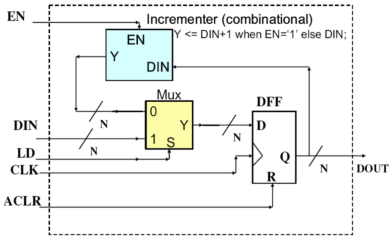


Figure 43: Counter

Another way to design a counter is by **toggle mode**. LSB will toggle every clock cycle. For an up counter, a higher bit will toggle if **all** the previous bit becomes 1; for a down counter, a higher bit will toggle if the previous bit becomes 0(toggle to 0). We can also use MUX mode to create up/down counter selector.

State sequence for a 3-bit binary counter.

Clock Pulse	Q_2	Q_1	Q_0
Initially	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8 (recycles)	0	0	0

Figure 44: 3-bit Binary Up Counter in Toggle Mode

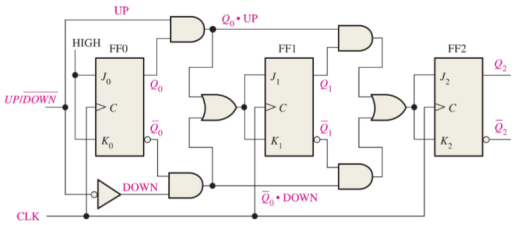


Figure 45: 3-bit Up Down Counter

A variation of binary counter is BCD counter. Extra attention should be paid on conditions that the even-bits will not toggle. We can also modify the clock edge(ripple counter or asynchronous counter) to control when a higher bit counter will toggle.

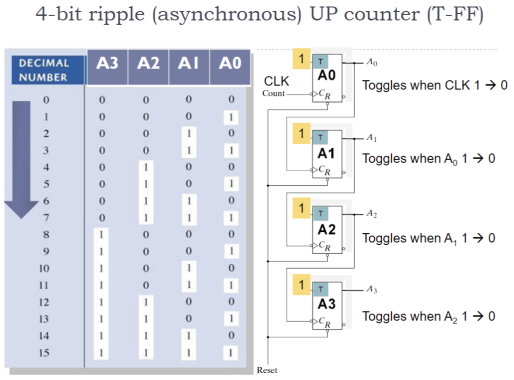


Figure 46: Asynchronous Up Counter

3.6.1 Karnaugh Map

K-map is another way to present truth table. The expression within K-map are minterms, while the layout sequence of K-map follows gray code. This makes adjacent cells on K-map have boolean adjacency(differ by only 1 variable). Here is a k-map.

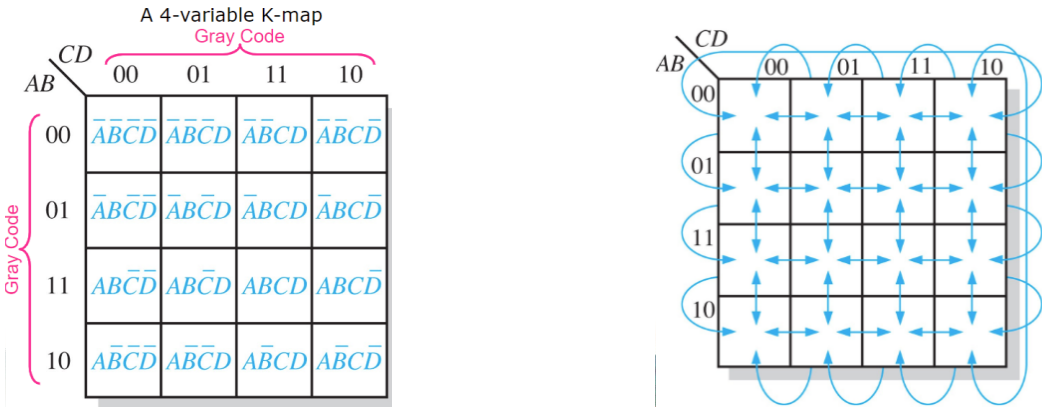


Figure 47: K-map

The reason to construct a K-map is to do logic minimization. Logic minimization starts with grouping the **1**s on the K-map by enclosing those adjacent cells containing 1s. The goal is to maximize the size of the groups and to minimize the number of groups.

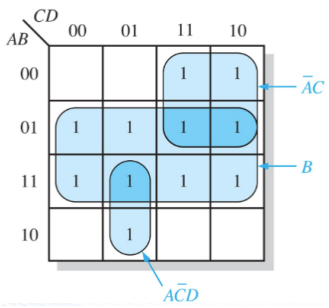


Figure 48: K-map for Minterms

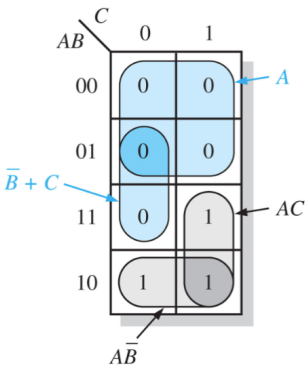


Figure 49: K-map for Maxterms

3.6.2 3-bit Gray Code Sync. counter

How to choose JK flip flop for all 3 bits?

Q_N	Q_{N+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Present State			Next State		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	0	1	0
0	1	0	1	1	0
1	1	0	1	1	1
1	1	1	1	0	1
1	0	1	1	0	0
1	0	0	0	0	0

Then build the 6 K-map(J and K for all 3 bits) from the above input mappings and transition tables.

3.7 Latches and Flip-flop Design

Recall that a **sequential logic** is the combinational logic with a **feedback** path, which connect output back to input so that the output in sequential logic can depend on the sequence of its **past** input bits. This is the reason why sequential logic has a **memory**. There are several feedback built from basic combinational components.

1. **Inverters with Feedback** Odd number of inverters with feedback will keep oscillating between 0 and 1, while even number of inverters will store 0/1 between the 2 inverters. This is a **read-only** component with memory element 0 and 1. It can be improved to a **read-write** component by adding load input(**transmission gate**).

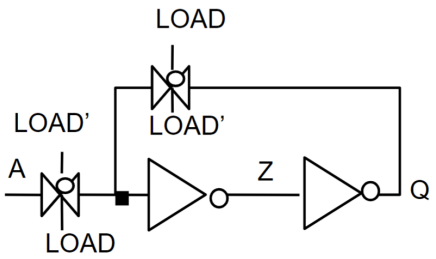


Figure 50: Inverters with Feedback

2. **AND/OR Gates with feedback**

   We can establish **RESET/MEMORY** by AND gate, and **RESET/MEMORY** with OR gate. The problem of single gate is that for AND gate, once reset, it cannot be set again, while for OR gate, once set, it cannot be reset again. So we can combine both gates to construct a