**Preface**

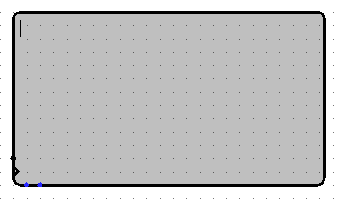
A large portion of this documentation is my personal opinion and ideas I have got while studying respective topics from various sources. It is very likely that in some cases things are very different or total opposite of how I had have understood them. This documentation is not recommended for using as reference. A large portion of this documentation is based on the book “Digital Logic and Computer Design by M. Morris Mano”. This book can be used as reference if needed. It will be very helpful if you inform me about any major or minor error in the documentation.

**Things you need to know**

‘Logic diagram’ means the specific topic-wise logic diagrams with the documentation.

‘Logisim’ software is used for simulation.

 This component in ‘Logisim’ is called a tunnel. Two tunnels of same label act as a hidden wire. This is used to reduce wires in the design in ‘Logisim’.

This component in ‘Logisim’ is called TTY (Teletype writer). The name has a historical background. What it does is, it acts like some kind of output terminal (You can think of it as a monitor).

This component in ‘Logisim’ is called Keyboard. It basically is a keyboard buffer. You can use your physical keyboard to enter 7-bit ASCII characters into your circuit through it.

Last but not least, the software ‘Logisim’ is pronounced as ‘LODJ-uh-sim’. It is not pronounced as ‘LOG-ism’.

**Adders :**

* **Half adder :**

Half adder takes two bits as input and add them. It gives the sum of two bits and carry as output.

* **Full adder :**

Full adder takes three bits as input and add them. It gives the sum of three bits and carry as output. Full adders can be cascaded to form adder for larger input e.g., 4 Bit Parallel binary adders. Two half adders can be cascaded together two form a full adder.

* **Half Subtractor :**

Similar to Half Adder just output ‘Borrow’, B = X’.Y but in Half Adder output ‘Carry’ was C = X.Y

Where X and Y are inputs to the gate in both cases.

* **Full Subtractor :**

Similar to Full Adder, composed of two Half Subtractors instead of two Half Adders.

* **4 Bit Parallel Binary Adder :**

4 Full Adders are cascaded by inserting one’s output carry as another’s third input. For the first Full Adder the third input is kept 0 as there is no previous carry. If two Parallel Adders are cascaded the first Full Adder will take the output carry of previous Parallel Adder as third input.

* **BCD to XS3 Converter :**

XS3 or X3 or Excess-3 is a coding system where 3 is added to BCD. This is why it is called Excess-3. It can be done by using 4 Bit Parallel Binary Adder and providing 0011 as one input. If 3 or we should say 0011 in binary is added two BCD we gate,

|  |  |  |
| --- | --- | --- |
| **BCD** |  | **XS3** |
| 0000 | +0011 | 0011 |
| 0001 | +0011 | 0100 |
| 0010 | +0011 | 0101 |
| 0011 | +0011 | 0110 |
| 0100 | +0011 | 0111 |
| 0101 | +0011 | 1000 |
| 0110 | +0011 | 1001 |
| 0111 | +0011 | 1010 |
| 1000 | +0011 | 1011 |
| 1001 | +0011 | 1100 |

As Decimal can be at most 0 – 9, BCD can’t be greater than 1001. Some important properties of XS3 code :

1. Self-complementing :

A coding system is said to be self-complementing if  the 9’s complement of a decimal number can be achieved by changing 1’s to 0’s and 0’s to 1’s e.g., in XS3 code (0)10 = (0000)2 = (0011)XS3 and (9)10 = (1001)2 = (1100)XS3 , here (0011)XS3 and (1100)XS3 are complement of each other. This is also true for 1 and 8, 2 and 3 and so on.

1. Un weighted code :

Non-weighted or un-weighted codes are those codes in which the digit value does not depend upon their position. XS3 is the only unweighted coding system that is self-complementing.

N.B. In case of weighted coding system if the sum of weights is equal to 9 then the system is self- complementing e.g., 2-4-2-1 coding system, 2 + 4 + 2 + 1 = 9; So, self-complementing.

* **4 Bit BCD Adder :**

As we know, decimal numbers can be of 0 – 9 , so the BCD also can be of 0 – 9. In decimal if we add two digits, the highest sum could be 9 + 9 + 1 = 19 where the 1 is carry from previous sum. So, our BCD adder should be able to generate BCD from 0 – 19. In range 0 – 9 BCD are same as binary 8-4-2-1. From 10 to 19 we have to add extra (6)10 = (0110)2 to the binary to get BCD. But for that we have to first detect binaries of 10 to 19.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **BCD** | | | | | **Decimal** |
| **B4** | **B3** | **B2** | **B1** | **B0** |  |
| 0 | 1 | 0 | 1 | 0 | 10 |
| 0 | 1 | 0 | 1 | 1 | 11 |
| 0 | 1 | 1 | 0 | 0 | 12 |
| 0 | 1 | 1 | 0 | 1 | 13 |
| 0 | 1 | 1 | 1 | 0 | 14 |
| 0 | 1 | 1 | 1 | 1 | 15 |
| 1 | 0 | 0 | 0 | 0 | 16 |
| 1 | 0 | 0 | 0 | 1 | 17 |
| 1 | 0 | 0 | 1 | 0 | 18 |
| 1 | 0 | 0 | 1 | 1 | 19 |

From the table we get that,

1. If B4 is High
2. If B1 AND B3 is High
3. If B3 is High AND (B1 OR B2 is High)

Then the numbers are in 10 – 19 range and we have to add (0110)2 with the binary to convert it to BCD.

So, we will add two BCD numbers normally using an Adder then we will check for above conditions and add either 6 if conditions are met or 0 if not using another adder. As a sample output, (1001)BCD + (1001)BCD + 1 = (11001)BCD = (19)10

* **4 Bit Magnitude Comparator :**

Comparators just compare two numbers and result can be any of three cases greater, less or equal. But for binary numbers designing comparator can be a tedious process if done in traditional combinational circuit analyzing way. Say, we want to compare two 4-bit binary numbers. So, our truth table will have 28 = 256 input combinations analyzing which with k-map will be tedious. So, we follow an algorithmic process.

Say we have two 4-bit binary numbers, A = A3 A2 A1 A0 and B = B3 B2 B1 B0

1. A and B are equal if

(A3 AND B3 are both 0 or 1) OR (A2 AND B2 are both 0 or 1) OR (A1 AND B1 are both 0 or 1) OR (A0 AND B0 are both 0 or 1)

= A3 XNOR B3 + A2 XNOR B2 + A1 XNOR B1 + A0 XNOR B0

= X3 + X2 + X3; [Say]

1. A is greater than B if

(A3 is 1 AND B3 is 0) OR [(A3 is equal to B3) AND (A3 is 1 AND B3 is 0)] OR [(A3 is equal to B3) AND (A2 is equal to B2) AND (A1 is 1 AND B1 is 0)] OR [(A3 is equal to B3) AND (A2 is equal to B2) AND (A1 is equal to B1) AND (A1 is 1 AND B1 is 0)] OR …

= A3 B3’ + X3 A2 B2’ + X3 X2 A1 B1’ + X3 X2 X1 A0 B0’

1. A is smaller than B if

(Same procedure as b just switch Ai with Bi).

Here, steps a, b and c correspond 3 outputs of the comparator. At a particular instance only one of the outputs will be High and remaining will be at Low state.

* **4 Bit Serial Binary Adder :**

This type of adder uses shift register to operate in serial mode. It is recommended to read the Registers section of this document prior to read this serial adder section. The process of serial addition is fairly simple.

1. Two numbers are loaded into two shift registers.
2. With every clock pulse two bits are fetched from two registers and their addition is performed.
3. Sum of the two bits is fed back two one of the registers to store the sum.
4. Output carry is sent to a flip-flop to store it till next clock pulse to add with next two fetched bits.

In the designed circuit things need explanations are ‘SelectLine’, ‘ExternalInput’ and the AND gate connected to the flip-flop.

Here, two ‘4-bit bidirectional shift registers with parallel loads’ are used. ‘SelectLine’ determines the mode of operation for registers. It is kept 00 for loading the inputs parallelly to the registers. And to perform the addition it is to be 01. 01 tells the registers to operate into the Right-Shift mode.

We will add the previous carry only when two new bits are fetched a.k.a. registers in Right-Shift mode. This is why the output of ‘SelectLine’ is ANDed with the clock pulse to the flip-flop.

The ‘ExternalInput’ is used for serially inserting another number into the addition process. For adding only two binary numbers it is kept Low.

**Encoders and Decoders :**

* **3 to 8 (n to 2n) Line Decoder/Demultiplexer :**

A decoder takes a binary number, n, as input and turn the nth output line High.

A demultiplexer takes a binary number, n, as input and data, i, as input and supply the i data to the nth output line.

A decoder with an Enable pin acts like a demultiplexer. The Enable pin is considered to take data input. This is why the term Decoder/Demultiplexer is used.

For an example, if we give 1 at enable pin and 101 = (5)10 at select pins, it will supply the 1 to the 5th output line.

Any Boolean function of n variables can be implemented through a n to 2n decoder. To do this we have to express the function in minterms. Then we have to take the output lines from the decoder according to the minterms and apply OR operation on them. E.g., say, F(A, B) = A’ B + A B’, is a Boolean function. We can express it like F = ∑(1, 2). So, from a 2 to 4 Line Decoder we will take output lines 1 and 2 and OR them to get the F.

Decoder is also used to design ROMs which is discussed in ‘Important Implementations’ section of this document.

* **Octal to Binary (2n to n) Encoder :**

A 2n to n line encoder has 2n input lines. Say nth input line is turned High, the output will be binary of n. A typical encoder requires that only a single input line will be turned High at a particular instance and the remaining lines will have to be Low. This drawback is overcome by using special type of encoder called Priority Encoder.

* **4 to 2 Line Priority Encoder :**

Priority Encoder is a special type of encoder that only cares about the input line of highest priority. So, in case of Priority Encoder many input lines can be turned High but it will take only the highest priority input line into account. Say, nth input line is turned High. The priority Encoder won’t care about the condition about input lines 0 to (n – 1). The truth table for 4 to 2 Priority Encoder is :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **D3** | **D2** | **D1** | **D0** |  | **B1** | **B0** |
| 0 | 0 | 0 | 0 |  | x | x |
| 0 | 0 | 0 | 1 |  | 0 | 0 |
| 0 | 0 | 1 | x |  | 0 | 1 |
| 0 | 1 | x | x |  | 1 | 0 |
| 1 | x | x | x |  | 1 | 1 |

Now, K-map for B0,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **00** | **01** | **11** | **10** |
| **00** | x |  | 1 | 1 |
| **01** |  |  |  |  |
| **11** |  |  |  |  |
| **10** | 1 | 1 | 1 | 1 |

Now, K-map for B1,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **00** | **01** | **11** | **10** |
| **00** | x |  |  |  |
| **01** | 1 | 1 | 1 | 1 |
| **11** | 1 | 1 | 1 | 1 |
| **10** | 1 | 1 | 1 | 1 |

* **4 to 1 Line Multiplexer :**

Multiplexer is a digital input line selector. It has multiple input lines, say 2n input lines. And n selection lines. Depending on the input in selection line (Say we have provided (m)2 in selection lines.) the mth input line will be connected to the output. The truth table for 4 to 1 line Multiplexer (MUX) is,

|  |  |  |
| --- | --- | --- |
| **S1** | **S0** | **Output** |
| 0 | 0 | Input0 |
| 0 | 1 | Input1 |
| 1 | 0 | Input2 |
| 1 | 1 | Input3 |

So, here output will be,

Output = S1’ S0’ Input0 + S1’ S0 Input1 + S1 S0’ Input2 + S1 S0 Input3

N.B. This Function derivation from truth table is a little different as the output is not binary value rather variables.

**Basic Flip-Flops :**

Flip-flops on basis of sensitivity to clock pulse can be of two types : Level triggered or Edge triggered. Basic flip-flops are of type Level triggered as they take inputs the whole-time clock is at High level and when clock goes to Low level, they don’t take inputs . In case of edge triggered flip-flops, they take input only when a transition occurs at clock pulse i.e., clock goes to Low to High(PGT) or High to Low(NGT) not during the whole-time clock remains active. Edge triggering can be achieved in multiple methods like Master-Slave designing pattern or Edge-triggered designing pattern.

* **RS Flip-Flop with NOR Latch :**

RS flip-flop has two input pins one is Set pin and another is Reset pin. And two output pins one is complement of another. There can be four input combinations possible :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Set** | **Reset** | **Output** | **~Output** | **Notes** |
| 1 | 0 | 1 | 0 | Output is said to be set |
| 0 | 0 | 1 | 0 | No change condition |
| 0 | 1 | 0 | 1 | Output is said to be clear off or reset |
| 1 | 1 | 0 | 0 | Indeterminant Condition as both Output and its complement become 0 |

1 for both Set and Reset pin is an ambiguous condition. Always this condition should be avoided. \*Only RS flip-flop has this kind of ambiguous situations.

* **Clocked RS Flip-Flop with Nor Latch :**

A clock pulse is ANDed with Set and Reset pins. When the clock pulse is 1 only then the output can be entered into the latch.

The characteristics table for RS flip-flop is :

|  |  |  |  |
| --- | --- | --- | --- |
| **Outputcurrent** | **Set** | **Reset** | **Outputnext** |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | indeterminant |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | indeterminant |

The Excitation table for RS flip-flop is :

|  |  |  |  |
| --- | --- | --- | --- |
| **Outputcurrent** | **Outputnext** | **Set** | **Reset** |
| 0 | 0 | 0 | x |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | x | 0 |

N.B. Characteristics table and excitation table are almost same. When we know the current states and also the next states, we use excitation table to determine the states of flip-flop. There is no indeterminant case in Excitation table as we know 0 to indeterminant or 1 to indeterminant transition will never to occur practically.

* **Master Slave RS Flip-Flop with NOR Latch :**

There is a practical problem with Clocked RS flip-flop (Basically all types of clocked flip-flops). Clocked flip-flops take inputs when clock is High. If frequency is low clock pulse stays high for longer period. During this whole period flip-flop will be able to take input. But we don’t want that (Why? Remember in JK flip-flop we feed back the output to flip-flop again. If it can take input the whole-time clock is High, God knows what will go inside and what will be fed back!). We want flip-flops to take input with clock transitions. So, every time clock goes low to high(PGT) or high to low(NGT) we want flip-slops to take inputs and stop. Wait till next clock transition before taking inputs again. But this behavior can’t be achieved using typical flip-flops only.

One solution is using Master-slave designing pattern. Here instead of one flip-flop, two flip-flops are used. The master flip-flop is connected to input and clock pulse. Output is taken from slave flip-flop and it is given complemented clock pulse. Output of aster ff goes to slave ff. Working procedure is like :

1. Clock pulse goes Low to High
2. Master flip-flop takes input and send to slave flip-flop
3. But slave flip-flop doesn’t accept the inputs yet as it is provided complemented clock pulse
4. The moment clock pulse goes High to Low the slave flip-flop takes input and provides final output
5. But during this period master flip-flop doesn’t accept any input as clock pulse for it is complemented.
6. Thus, this whole construction acts as a single NGT triggered flipflop
7. And this is why almost all flip-flops used in text books and lectures are NGT triggered

In simple words, Master and Slave are not enabled at same time in this type of flip-flop construction.

* **Clocked D Flip-Flop with NAND Latch :**

Usually, D flip-flop or D latch is constructed with NAND gates. This flip-flop is used to construct registers. With clock pulse it passes the data at D pin to Output pin. D flip-flop has no No-Change condition. So, by default it can’t retain data. To make it retain data its output needed to be fed back through some logic. This phenomenon is observed at register section.

* **Edge Triggered D Flip-Flop with NAND Latch**

This is another construction method for achieving edge triggering for flip-flops. In the logic diagram, a D flip-flop is designed. Here three basic RS flip-flops are used to design this D flip-flop. This is a PGT triggered D flip-flop with NAND latch. So, 11 to the FF3 means No-Change Condition.

When Clock is Low, the output of NAND gates 2 and 4 of FF1 and FF2 is always 1 (As we know, even a single 0 in NAND gate ensures the output to be 1). So, FF3 gets 11 always while the clock is Low no matter at what state D is. So, final output Q from FF3 remains unchanged.

When D = 1 and clock goes Low to High, output of NAND gate 4 must be 0 as both its inputs are 1 (One is from D and 1 is from Reset pin of Low clock state). So, NAND gate 3 must be 1 so as Reset pin. And NAND gate 1 also must be 1. Moreover, NAND gate 2 must be 0 as both its inputs are 1 (One is from gate 1 and 1 is from clock High) so as the Set pin. So, FF3 will gate Set = 0, Reset = 1 which is the set State for NAND Latch, Output will be Q = 1. Now further changes of D while clock is High won’t do anything. As it will only change one input of gate 3 and one input of gate 1. But output of gates 3 and 1 no longer depend on D. So, Set and Reset pins will remain same as before so as the output.

Almost similar case will happen when D = 0 and clock pulse will change from low to high. But in any case, Low clock pulse will result in steady state.

**Error Detection Codes :**

In data transmission bits of data may be altered due to noise causing error in data transmission. These errors need to be detected and corrected. There are multiple methods to detect and correct these types of errors.

* **Parity Bit Generator For 3 Bit Data :**

Parity bit is an extra bit added to the dataset during transmission for checking the correctness and possible error in data set later while receiving. In simple words, parity bit in a peculiar way resembles the number of 1’s in a dataset. There are two ways to generate a parity bit for a dataset :

1. Even parity :

Let’s say our dataset is D = 001. Are there even number of 1‘s? No. So, if we put another 1 in dataset number of 1’s will be even. This extra 1 will be our parity bit for this dataset. So, final dataset with parity bit will be, D = 1001 and we will say Even Parity of our data set D is 1. Again, let’s say our data set is D = 101. Are there even number of 1‘s? Yes. So, we need not to put another extra 1 to make the number of 1’s even. So, in this case our Even Parity bit will be 0 i.e., D = 0101.

1. Odd Parity :

Again, let’s say our dataset is D = 101. Are there Odd number of 1‘s? No. So, if we put another 1 in dataset number of 1’s will be odd. This extra 1 will be our parity bit for this dataset. So, final dataset with parity bit will be, D = 1101 and we will say Odd Parity of our data set D is 1. Again, let’s say our data set is D = 001. Are there Odd number of 1‘s? Yes. So, we need not to put another extra 1 to make the number of 1’s odd. So, in this case our Odd Parity bit will be 0 i.e., D = 0001.

A table for Even and Odd parity for 3-bit data set :

|  |  |  |
| --- | --- | --- |
| **Data** | **Even Parity** | **Odd Parity** |
| 000 | 0 | 1 |
| 001 | 1 | 0 |
| 010 | 1 | 0 |
| 011 | 0 | 1 |
| 100 | 1 | 0 |
| 101 | 0 | 1 |
| 110 | 0 | 1 |
| 111 | 1 | 0 |

From this table Boolean function for parity bit can easily be generated,

Even Parity = D0 XOR D1 XOR D2

Odd Parity = (D0 XOR D1 XOR D2)’

* **Block Parity Generator For 3x3 bit Data Block :**

Block parity is basically normal parity bit but for 2D array of data (Data block). In case of block parity, both row and column wise parity are generated for a block of data. As, an example, let’s say we have 3 data sets,

DS0 = 001

DS1 = 101

DS2 = 111

So, if we consider only even parities,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | **Parities** |
| **DS0** | 0 | 0 | 1 | 1 |
| **DS1** | 1 | 0 | 1 | 0 |
| **DS2** | 1 | 1 | 1 | 1 |
| **Parities** | 0 | 1 | 1 |  |

Block parity has advantage over normal parity. Say, we have a data set D = 101001. After transmission due to noise, it changes to D’ = 100101. Here, two bits are altered. Normal parity checking won’t be of any use in this case. But using block parity we can detect this error.

* **Bi-Quinary Code :**

Bi-quinary or Bi-Quinary Coded Decimal is a 7-bit weighted decimal coding system where bit weights are like 5-0-4-3-2-1-0 from MSB to LSB. A table for Bi-Quinary code is :

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Weights ->** | **5** | **0** | **4** | **3** | **2** | **1** | **0** |
| **Decimal** | **B6** | **B5** | **B4** | **B3** | **B2** | **B1** | **B0** |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 5 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 6 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 7 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 8 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 9 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

How this can be used to detect error in transmission? See no BQCD number contains more than 2 1’s. So, after transmitting data in BQCD form if data contains more or less than 2 1’s that means error has occurred.

In logic diagram we have used parity bit generator for every row and column to generate corresponding parity bits.

* **Hamming Code Generator for 4 Bit Data :**

It’s a special type of coding system developed by Richard Wesley Hamming**.** In this coding system parity bits are also used. Generally, this in this coding system output is 7 bits long. Among them 4 bits are for data and 3 are for parity bits. To understand this we need an example,

Suppose we have a 4-bit dataset DS = B3 B2 B1 B0. From here we need to get three parity bits combinations P0, P1 and P2.

P0 B1 P2

B3

B0 B2

P1

This is the diagram to follow for getting combinations of parity bits.

P0 = Parity bit for B0 B1 B3

P1 = Parity bit for B0 B2 B3

P2 = Parity bit for B1 B2 B3

And the output is to be decorated in form of

Output = B3 B2 B1 P2 B0 P1 P0

In our logic diagram we have used our pre-designed 3-Bit Parity Generator to generate the parity bits.

**Registers :**

Registers are basically memory devices with extra functionalities. In multiple cases we need registers. In broad sense even counters are considered registers. Keeping that aside a register can be used to switch between operation mode. Let me explain. Computer can operate in Parallel mode e.g., 40Bit Parallel Adder which takes 4 bits together at a time as inputs. Computer can also operate in serial mode e.g., 4-Bit Serial Adder is an example of this. In serial mode computer takes input serially one at a time. With registers we can switch between these modes. We can convert serial data to parallel data and vice versa.

* **4 Bit Parallel In Parallel Out (PIPO) Register with D Flip-Flop :**

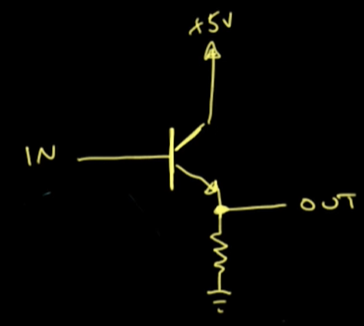
A PIPO register takes in all inputs at a time with single clock pulse and drive them to output. In logic diagram the ‘Load’ pin and feed back path from each flip-flop’s output to input need explanation.

Generally, flip-flops are controlled through clock pulse. A system is usually given a Master Clock to supply clock pulses to all its components. So, it is not a very good practice to rely on clock pulse only to operate vital devices like registers. This is why ‘Load’ pin is used to control the register along with clock pulse. Clock keeps running and the ‘Load’ pin input decides whether register will take input or not.

The feed back path is needed only in this case for D flip-flop. Unlike RS or JK flip-flop, D flip-flop has no Steady state condition. So, by default D flip-flop can’t retain previous value when ‘Load’ pin is off. This is why output is fed back while Load pin is off. It is exclusively needed for this case only as when ‘Load’ is off 0 goes to D flip-flop but it needs to retain its previous value.

* **4 Bit Parallel In Parallel Out (PIPO) Register with JK Flip-Flop :**

Works same as PIPO with D flip-flop. No feed back path needed as JK flip-flop by default has No-Change condition. Only thing needed to be explained is the Buffer with ‘Load’ pin. Logically a Buffer gate is two NOT gates i.e., F = (A’)’ = A. So, why on earth we need this? To get the idea we need to look inside Buffer gate. Internally Buffer Gate is like,



(Well, not exactly like this but we can take this model to understand) Notice two things carefully here. Input is given to the Base of transistor and output is taken from Emitter of the transistor. Now consider a situation,



If Circuit-2 has low impedance(Resistance actually as DC) then that means load is increased. So, Circuit-2 will draw more current from source. But this increased flow of current will flow through Circuit-1. This can be harmful to Circuit-1. If we place a Buffer between them let’s see what will happen. Buffer a.k.a. transistor in Buffer will receive input through its Base. But we know Base current is very small. So, it will take a very low amount of current as input. As a result, Load is reduced for the source and Circuit-1 also. On the other hand, Emitter will supply the necessary current to Circuit-2 as emitter Emits larger amount of current.

So, Circuit-2 will get what she needs without nagging to Circuit-1 and Source. In former case of D flip-flop this wasn’t done because the inverter at ‘Load’ pin does the same.

* **4 Bit Serial In Serial Out (SISO) Register/Shift Register with D Flip-Flop :**

This is the register that works in serial mode. Inputs are given one at a time and shifted right towards output with every clock pulse. It takes n clock pulses two drive an input to the output pin where nis the length of register or number of flipflops used in register.

* **Bit Bidirectional Shift Register With Parallel Load :**

This is the general register that combines the functionalities of all registers stated above and many more. A general shift register has to have capabilities listed below :

1. A Clear control to clear off the register -> Clr pin
2. A Clock pulse -> Clk pin
3. A Shift-Right Control for operating in Right Shift mode-> 10 in Select Line

and input and output pin for Right-Shift mode

1. A Shift-Left Control for operating in Left Shift mode -> 01 in Select Line

and input and output pin for Left-Shift mode

1. A Parallel-Load Control for operating in Parallel mode -> 00 in Select Line

And n input and output pins for parallel mode

1. A Control state that leaves the information unchanged even though the clock pulses are continuously applied -> Select line do this job

Here, in logic diagram, multiplexers are used to select operation mode based on the select line input. Though Select line input combinations are listed above we are listing it here for better documentation :

|  |  |
| --- | --- |
| **Select Line Input** | **Mode** |
| 00 | Parallel Mode |
| 01 | Left-Shift Mode |
| 10 | Right-Shift Mode |
| 11 | No Change Mode |

**Counters :**

Counters are circuits that counts. There are basically two types of counters based on how to provide clock pulse to them. Synchronous Counters- All of their internal flip-flops get clock pulse at the same time. Asynchronous Counters/ Ripple Counters- Clock pulse is given to the first flip-flop(LSB) and rest of flip-flops use each other’s output as clock. A Counter of n flip-flops can count 2n numbers at most.

* **4 Bit Binary Synchronous Up Counter with JK Flip-Flop :**

This synchronous counter counts from 0000 to onwards. This counter is based on the principle that the LSB of binary number system toggles always and other bits toggle when all of previous bits are High together. This situation is explained below :

|  |  |  |
| --- | --- | --- |
| **B2** | **B1** | **B0** |
| 0 | 0 | 0 |
| 0 | 0<-Here it toggles, B0 = 1 | 1 |
| 0 | 1 | 0 |
| 0<-Here it toggles, B0 = B1 = 1 | 1<-Here it toggles, B0 = 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0<-Here it toggles, B0 = 1 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

So, we can make the decision,

B0 toggles with every clock pulse

B1 toggles while B0 is 1

B2 toggles while (B0 is 1 AND B1 is 1)

… … …

And So on. So, constant 1 is fed to B0 ff, output of B0 is fed to B1 ff and (output of B0 AND output of B1) is fed to B2 and so on.

In logic diagram two pins are needed to be explained- ‘CountEn’ Pin and ‘next’ pin.

The ‘En’ pin or enable pin enables all the flip-flops. But for counting constant 1 is needed to be supplied to J and K pins of the LSB flip-flop as LSB needs to toggle every time and 1 in J and K pins enables toggling. And other flip-flops depend on output from LSB flip-flop to toggle. So, through ‘CountEn’ pin we supply this. If ‘CountEn’ pin is Low, the counter won’t count. This pin is very important for cascading multiple counters.

The ‘next’ pin is also needed for cascading multiple counters. In cascading, ‘next’ pin of one counter is connected to the ‘CountEn’ Pin of another. This whole cascading thing is explained in implementation section.

* **4 Bit Binary Synchronous Down Counter with JK Flip-Flop :**

Same as Synchronous Up counter. Just the output is taken from complement pin of flip-flops to count inversely.

There is another way to design this counter where outputs are taken as previously from the normal output pins of flip-flops but inputs of flip-flops are provided from complemented pins unlike the Up Counter. If this design is used, the counter initially starts from 0000 then goes to backwards like 1111, 1110 etc.

* **4 Bit Binary Synchronous Up Down Counter with JK Flip-Flop :**

This is the combined form of two counters stated above. It has four extra pins for Down counting mode, Up Counting mode and two Next pins for cascading.

|  |  |  |
| --- | --- | --- |
| **DownEn** | **UpEn** | **Action** |
| 0 | 0 | Doesn’t Count |
| 0 | 1 | Counts Upward |
| 1 | 0 | Counts Downward |
| 1 | 1 | Oscillates between last two numbers |

* **4 Bit Binary Synchronous Up Down Counter with Parallel Load with JK Flip-Flop :**

This counter is basically a combined version of previous counter and a PIPO register. Using its parallel load input pins, we can load any initial number to the counter and it will start counting from it. Thus, we can start counting from anywhere we want.

Caution : Text book version of this counter is different than depicted here in the logic diagram. Here we have used flip-flops’ preset pin to load the initial inputs but in text book’s version they did it manually.

* **4 Bit Binary Asynchronous Up Counter with JK Flip-Flop :**

In Asynchronous counter clock pulse is not provided synchronously to all flip-flops. Instead, clock pulse is passed to the first flip-flop only. All other flip-flops used each other’s output as their clock pulse. One important thing to note is that, in case of Asynchronous up counter, flip-flops used are have to be NGT-triggered. And if PGT-triggered flip-flops are used, outputs are to be taken from complement pins of flip-flops. In our logic diagram we have used NGT-triggered flip-flops.

To get a better understanding observe the 4-Bit binary table,

|  |  |  |  |
| --- | --- | --- | --- |
| **B3** | **B2** | **B1** | **B0** |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |

Notice,

B0 toggles with every clock pulse let’s say of frequency F,

B1 toggles with clock pulse of frequency F/21

B2 toggles with clock pulse of frequency F/22

B3 toggles with clock pulse of frequency F/23

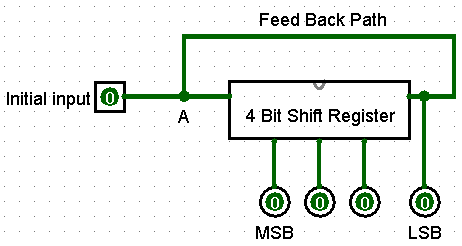
Here, clock frequency is being divided. This is achieved by passing one flip-flops output as another’s clock.

* **4 Bit Binary Asynchronous Down Counter with JK Flip-Flop :**

Similar to Asynchronous Up counter just counts backwards. Just use PGT-triggered flip-flops or take outputs from complement pins of flip-flops if NGT-triggered flip-flops are used to design this. In our logic diagram we have used NGT-triggered flip-flops.

* **4 Bit Ring Counter/** **Straight Ring Counter/ One-Hot Counter with D FF :**

This Ring Counter is a special type of counters. Till now all counters with n flip-flops had 2n states at most. But A Ring counter with n flip-flops has at most n states. A Ring counter is basically a Shift-Register with Parallel Output pins whose last shifted bit is fed back to it as input.

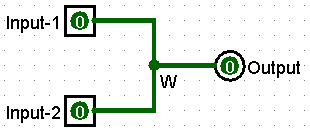


Initially 1 is fed as input in the Shift Register. Then with clock pulses the output sequence generated is :

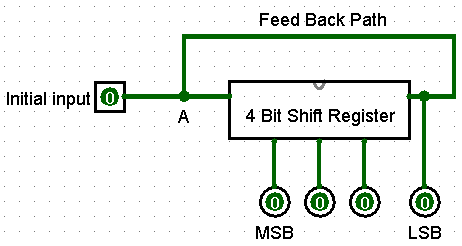
|  |  |  |  |
| --- | --- | --- | --- |
| **B3** | **B2** | **B1** | **B0** |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

This seems simple enough but, in our logic diagram the circuit designed seems quite messed up. Why? This is because Logisim doesn’t support Wired OR operation.

What is Wired OR operation? Well, Consider the figure below,



If either one of the inputs is High, Output will be High. And Output will be Low if and only if both inputs are Low. This is the OR operation. But no OR gate is used. This OR operation is performed at node W only by wiring. This is called Wired OR operation. But in Logisim it will cause an Error unless both inputs are High.



So, using Logisim we can’t simulate the Wired OR at node A. As a result, we can’t feed the initial 1 to the register directly. So, we have had to use flip-flop’s Preset pins for this which created the mess in our logic diagram design.

* **4 Bit Johnson Counter/ Twisted Ring Counter/ Switch-Tail Ring Counter/ Walking Ring Counter/  Möbius counter :**

Similar to Ring counter. The main difference is, instead of the serial output of the Shift Register, complement of the output is fed back to the register. This complement usually is taken from the complement pin of the last flip-flop of the register. Like Ring Counter it is also initially fed with 1. With n flip-flops we get 2n states in Johnson Counter. The states are :

|  |  |  |  |
| --- | --- | --- | --- |
| **B3** | **B2** | **B1** | **B0** |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |

**Logic families(Gates, CMOS and so on) :**

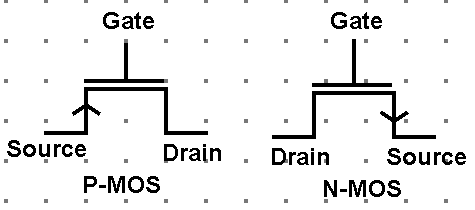
There are multiple logic families based one the components used to fabricate logic gates or ICs. Each logic family has their basic gate as the building block of other gates or ICs. Some logic families are :

1. Resistor-Transistor Logic (RTL)
2. Diode-Transistor Logic (DTL)
3. Transistor-Transistor Logic (TTL)
4. Metal Oxide Semiconductor FET (MOSFET)
5. Complementary Metal Oxide Semiconductor FET (CMOS)
6. Integrated Injection Logic (I2L)
7. Emitter Coupled Logic (ECL)

Common components of these logic families are transistors. Two types of transistors are there. Bipolar transistors like BJT and unipolar transistors like FET. The difference between bi-polar and unipolar transistor is bi-polar transistors depend on flow of both majority and minority carriers(Electrons and holes) but unipolar transistors depend only on the flow of majority carrier.

* **MOSFET :**

There are basically two types of FETs – JFET and MOSFET. Here our concern is about MOSFET. There are two types of MOSFETs. N-Channel MOSFET or NMOS and P-Channel MOSFET or PMOS. A MOSFET has three terminal – Source, Drain and Gate. Depending on the voltage supplied to the Gate terminal, Source and Drain act like short circuit or open circuit.



An important table to get the gist idea of differences in P-MOS and N-MOS operations :

|  |  |  |
| --- | --- | --- |
| **Gate to Source Voltage, VGS** | **P-MOS** | **N-MOS** |
| Positive | Turned Off | Conducting Mode |
| Negative | Conducting Mode | Turned Off |
| Zero | Turned Off | Turned Off |

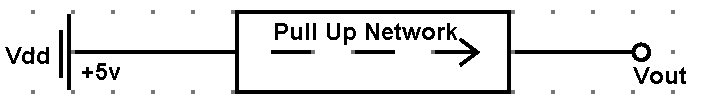
In Conducting mode Source and Drain of MOSFET are short circuited and in Turned Off mode they are open circuited.

* **CMOS :**

Two get the idea of how CMOS works some terms need to be predefined. There are two terms we need to know before exploring operation of CMOS – ‘Pull Up Network’ and ‘Pull Down Network’. What are they?

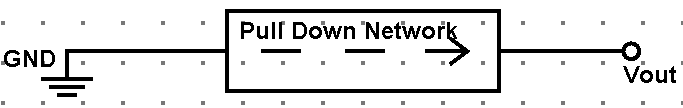
1. Pull Up Network :

In simple words, a network that connects its output to the supply voltage Vdd when in conducting mode is called a Pull-Up Network.



1. Pull Down Network :

In simple words, a network that connects its output to Ground when in conducting mode is called a Pull-Down Network.



Under usual conditions P-MOS is better for Pull-Up networks and N-MOS is better for Pull-Down Networks. But why? Let’s see. For this we need the table mentioned earlier.

|  |  |  |
| --- | --- | --- |
| **Gate to Source Voltage, VGS** | **P-MOS** | **N-MOS** |
| Positive | Turned Off | Conducting Mode |
| Negative | Conducting Mode | Turned Off |
| Zero | Turned Off | Turned Off |

P-MOS is in conducting mode when VGS < 0 and turned off when VGS = 0.

If we connect Source with Vdd then Vs is at constant High voltage (+5v or logical 1)

As, VGS = VG – VS and VS = 5v, depending on input G a.k.a. VG P-MOS will be at either conducting mode or turned off mode. But as Source is connected to +5v, in conducting mode it works as a Pull-Up Network.

N-MOS is in conducting mode when VGS > 0 and turned off when VGS = 0.

If we connect Source with Ground then Vs is at constant Low voltage (0v or logical 1)

As, VGS = VG – VS and VS = 0v, depending on input G a.k.a. VG N-MOS will be at either conducting mode or turned off mode. But as Source is connected to Ground, in conducting mode it works as a Pull-Down Network.

So, in Conducting Mode P-MOS works as Pull-Up Network and N-MOS works as Pull-Down network.

Now, how all these are going to help us to in designing Logic gates? Let’s see with an example. Let’s design a basic NAND gate with CMOS.

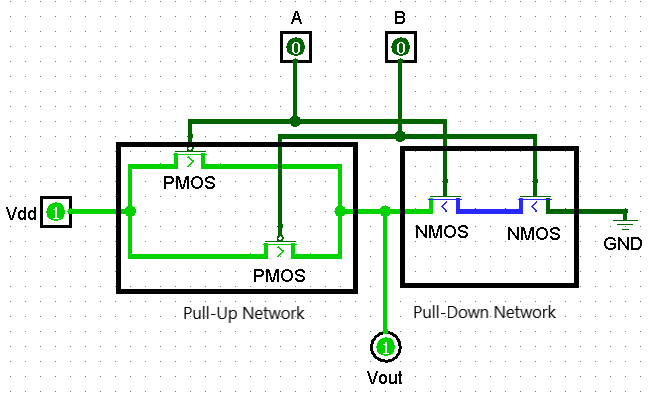
Our Boolean function for NAND gate is, F = (A B)’

Now, observe the truth able of NAND gate

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **F = (A B)’** |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Think in terms of network type. First three outputs are 1 (Highlighted as yellow). That means they need a Pull-Up Network. Again, last one output is 0 (Highlighted as blue). That means it need a Pull-Down Network.

This is how the idea of using both P-MOS and N-MOS together in CMOS actually is. The 1’s in output of Boolean functions are contribution of P-MOS and the 0’s are contribution of N-MOS. And if we merge them together, we get the complete output of a Boolean function.

With out loss of generality, it can be said that, if we consider a Boolean function F as a Pull-Up network, complement of F, F’ will be the Pull-Down network for same set of inputs.

**NAND Gate Designed in CMOS**

When merged together at a time only one of Pull-Up network and Pull-Down network remains in Conducting mode and another remains in Turned Off mode. So, both can’t contribute in output together. We get this ‘Only one works at a time’ property because we design Pull-Up network from given function F and pull-Down network from complement of F that is F’.

So, incase of our NAND get,

Pull-Up Network, F = (A B)’ = A’ + B’ -> 2 PMOSs, Connected parallelly, inputs given : A, B

Pull-Down Network, F’ = A B -> 2 NMOSs, Connected in Series, input given : A, B

\*Notice same set of inputs, A B are given to both PMOSs and NMOSs. One rule of thumb, PMOSs are always given inverted inputs of variables in Pull-Up function i.e., Pull-Up function, F = A’ + B’ but PMOSs take A and B as inputs not A’ and B’. Again, NMOSs are always given as it is input as in Pull-Down function i.e., Pull-Down function, F’ = AB and NMOSs are also taking A and B as inputs.

* **TTL :**

\*As academic lecture and my study contradicts in case of TTL, I don’t prefer to discuss my opinion about this topic.

**Memory Unit :**

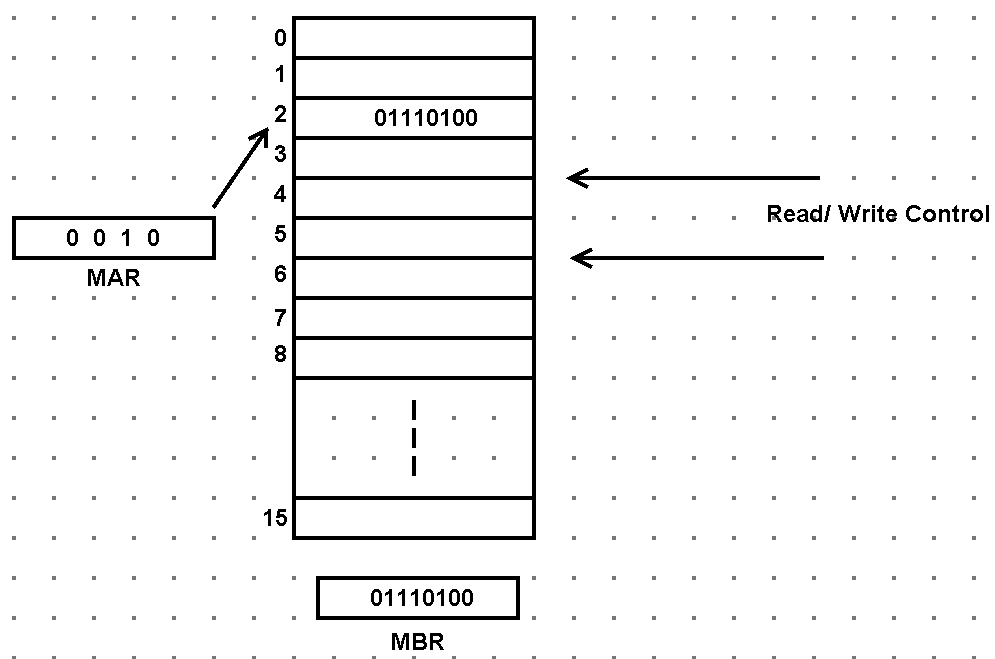
A memory unit is a collection of storage registers together with the associated circuits needed to transfer data in and out of the registers. Storage Registers used in memory unit are called memory registers. Registers can be of two types operational and storage registers. In memory unit we use storage registers.

The memory unit at its basic can be divided into three components,

1. Memory Address Register (MAR)
2. Memory Buffer Register (MBR)
3. A collection of n registers to store binary information as groups called words.

The working procedure of a memory unit is,

1. A register is selected from collection of n registers by passing the number(Address) of register we want to select through MAR.
2. We enable Read or Write mode to read from or write into the selected register.
3. The MBR is the interface that is used to read or write data. In Write mode, our data is given as input to the MBR. MBR writes the data into the selected Memory register. Similarly, in read mode, selected Memory register sends the stored information into the MBR and we read it from MBR.



As an example, here memory address 2 is selected as 0010 is passed to the MAR. Let’s say write control is enabled. Information 01110100 is written to the MBR then from MBR this information is transferred to address 2.

This memory unit can be of two types based on the way of access,

1. Random Access Memory or RAM :

In Ram the registers are thought of as being separated in space. Every unit (Say flip-flop for now) of a single addressed register can be accessed randomly and arbitrarily in RAM.

1. Sequential Access Memory :

In sequential memory each memory location passes read/write head in sequence. But data is only read or write when the requested location is reached. Magnetic-tape is an example of this.

Based on the component used to construct the memory unit it can be of two types :

1. Volatile :

This type of memory unit loses stored information when the power is turned off or with time. Memory units constructed with semi-conductors are of this type.

1. Non-volatile :

This type of memory unit retains stored information when the power is turned off. Magnetic core, magnetic disk etc. retains their stored information after removal of power.

* **Binary Cell :**

The building block RAM is called Binary-Cell can be constructed using flip-flop and some logic gates. A single Binary-Cell is capable of storing one bit of binary information. A typical Binary-Cell has 4 pins,

Select pin -> To select the cell for read/write operation

Read/Write pin -> To control read/write mode of the cell. In our logic diagram, 0 is for Write mode and 1 is for Read mode

Input pin -> To write or store data into the cell as input

Output pin -> To read stored data from the cell

The procedure needed to be followed for demonstration,

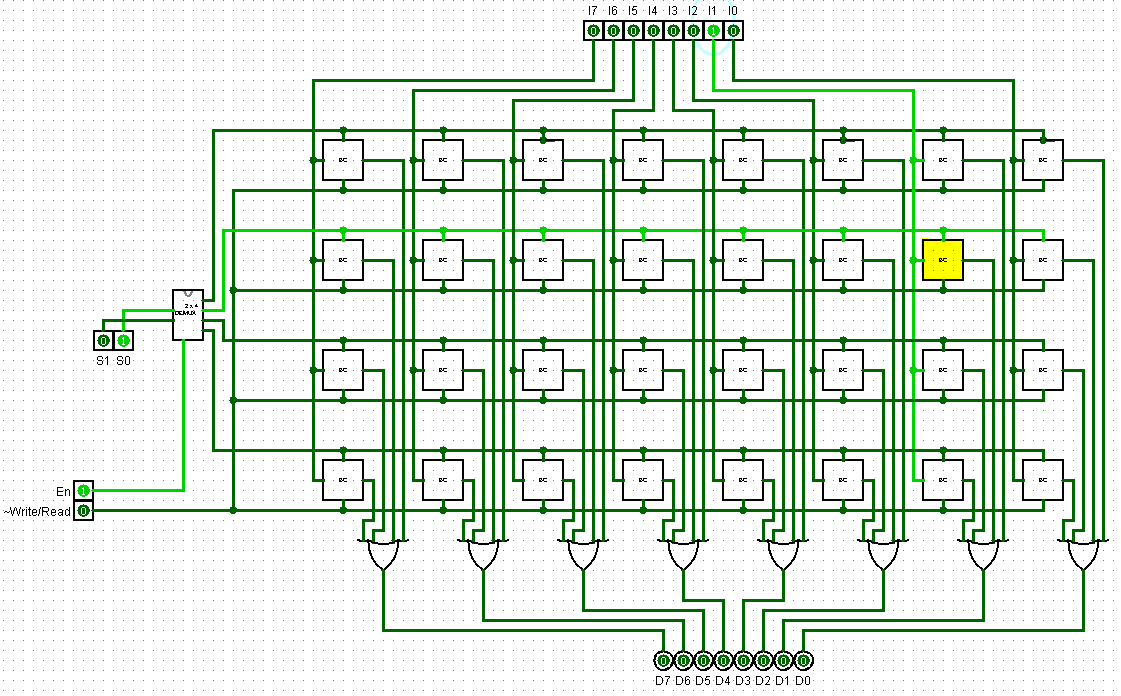
1. Turn High the select pin to select the cell
2. Keep Read/Write pin Low for Writing data to the cell
3. For demo, Give input 1 to the input pin
4. Make the Select pin Low to deselect the cell
5. Notice the output pin of internal RS flip-flop is reflecting your stored data into the flip-flop
6. At this point changing input arbitrarily won’t affect the cell as it is not selected
7. Turn Read/Write pin High to Read the stored data from the cell
8. Now select the cell again by turning the Select pin High
9. The stored data will be provided to the Output pin.

* **4x8 Bit RAM :**

In our logic diagram, this is a small RAM consist of 4 memory address each capable of storing 8 bits of information. So, here 4 memory addresses are possible and they are 00, 01, 10and 11. We have given them to a 2 to 4 Line Decoder to select a line from 0 – 3 based on our input address values. 0 – 3 total 4 lines, each are connected to 8 Binary-Cells’ Select pin. So, if 00 is given, first row of BC will be selected, for 01 second row and so on. The ~Write/ Read pin is connected to each BC to operate the RAM in Read or Write mode.

Thing the RAM as a 2-dimensional array, RAM[4][8]. We can access any BC in the RAM. Let’s say we want to write data into the RAM[1][6] Cell (Remember to count from 0). How to do that?

1. Keep the ~Write/ Read pin Low (In Write mode)
2. Give input 01 at the Select Lines, row 1 will be selected
3. Now give your input to the I1 pin. As only row 1 is selected, only the 6th cell of row 1 will be modified.



* **Memory Unit :**

As explained earlier, a basic memory unit is made of a Collection of Memory Registers (RAM in our case), a MAR and an MBR. In our logic diagram, a 2-Bit PIPO register is used for MAR and a 8-Bit PIPO register is used for MBR.

As explained earlier, the data from MBR goes into RAM if in Write mode and goes to output if in Read mode, we have used a Demultiplexer to switch between output lines for MBR.

Similarly, MBR stores data from outside of memory unit if in Write mode and from RAM if in Read mode. This is the reason for messed up network above the MBR. This mess an be cleared up by using a Multiplexer.

The 8 input OR gate is connected to the Load Control of MBR. It serves the purpose that the MBR stores data only if data is available. It is done for a practical reason. Say, we are taking input from a keyboard to MBR. We have pressed ‘A’ key. The MBR will store ASCII value of ‘A’. But after storing ‘A’, the keyboard buffer will be empty, no more character will be left. If MBR continues to take input it will take 0 as input and override the stored ‘A’. There may be other efficient way to deal with these problems. But it’s just a demo model. What else can be expected!

**Important implementations :**

* **F(A, B) = A’ B Implemented with CMOS :**

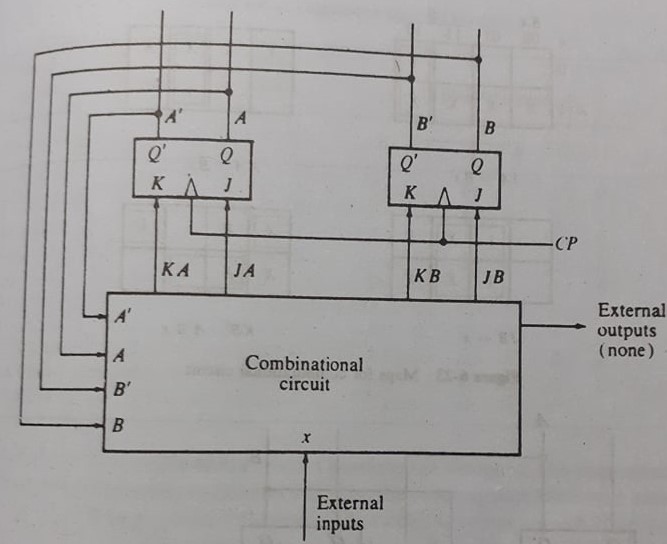
Here, function for Pull-Up Network, F = A’ B -> 2 PMOSs, Connected in Series, Inputs : A B’ (As PMOS always takes inverted version of literals that are in Pull-Up Function)

Here, function for Pull-Down Network, F = (A’ B)’ = A + B’ -> 2 NMOSs, Connected Parallelly, Inputs : A B’ (As NMOS always takes same version of literals that are in Pull-Down Function)

Connect Vdd with Pull-Up network’s source and Ground with Pull-Down network’s source. Take output from common Drain of both networks.

* **Randomly given Sequence Counter (Synchronous) :**

Synchronous counter is a sequential circuit. Every sequential synchronous circuit has a common designing pattern to follow.



This figure is the block diagram for common pattern of sequential circuit. Usually in case of sequential circuit we know the Present state and Next state (As in case of counter we know it will count 0001 after 0000 and so on). The main task here is to determine the Combinational circuit portion of the diagram. That means, we have to determine Boolean Functions KA, JA, KB, JB, External Output in terms of A, A’, B, B’ and External Inputs. To do this we need the excitation table. We will demonstrate the procedure by designing a counter that counts 1 -> 3 -> 2 -> 5 -> 4 sequence.

To design this, we have to follow some procedures :

1. Get the state diagram :

Ours is 1 -> 3 -> 2 -> 5 -> 4 -> Back to 1

Or, we can say 001 -> 011 -> 010 -> 101 -> 100 -> Back to 001

1. From the diagram obtain the state table :

A state table of sequential circuit is where the present states, next states, external inputs and external outputs are listed. In our case,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Present State** | | | **Next** **State** | | |
| **C** | **B** | **A** | **C** | **B** | **A** |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |

Notice the table. When 001 at present state, 011 at next state. Because in our sequence we count 1 then 3 and so on. Another important thing, there are no external inputs or outputs. This is because counter doesn’t have any. Their flip-flop states act as output. If it had had External input, it would be placed in the Present State section of the table.

1. Determine how many flip-flops are needed :

In our case highest number our device has to store in itself is 5 which is 101. So, we need at most 3 bits a.k.a. 3 flip-flops.

1. Assign letters to each flipflop :

Let’s call our flip-flop A, B and C same as the state table.

1. Choose the type of flip-flop to be used :

We will use JK flip-flops.

1. Derive the circuit’s excitation table :

Now is the crucial part. From the State table of circuit and Excitation Table of flip-flop, we have to derive excitation table for circuit.

Now, Excitation table for JK flip-flop is,

|  |  |  |  |
| --- | --- | --- | --- |
| **Outputpresent** | **OutputNext** | **J** | **K** |
| 0 | 0 | 0 | x |
| 0 | 1 | 1 | x |
| 1 | 0 | x | 1 |
| 1 | 1 | x | 0 |

And, again the state table derived at step 2 is,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Present State** | | | **Next** **State** | | |
| **C** | **B** | **A** | **C** | **B** | **A** |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |

Now for each of A, B and C’s transition from Present State to Next State we will determine the inputs of J and K pins for each three flip-flops which are JA, KA, JB, KB, JC and KC respectively. The circuit’s excitation table will be,

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Present State** | | | **Next** **State** | | | **FF A** | | **FF B** | | **FF C** | |
| **A** | **B** | **C** | **A** | **B** | **C** | **JA** | **KA** | **JB** | **KB** | **JC** | **KC** |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | x | 1 | x | x | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | x | x | 0 | x | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | x | x | 1 | 1 | x |
| 1 | 0 | 1 | 1 | 0 | 0 | x | 0 | 0 | x | x | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | x | 1 | 0 | x | 1 | x |

From this table we can determine functions for flip-flop inputs JA, KA, JB, KB, JC and KC in terms of Present State A, B and C. Simplifying by K-Map we get,

JA = A’ B B’

KA = A B’ C’

JB = A’ C

KB = A B B’

JC = 1

KC = A + B

1. Draw the logic diagram :

Just put three flip-flops and give their J an K pins respective inputs determined from step 6 and the sequential circuit is designed.

* **Boolean Function Implementation with Decoder :**

Any Boolean function of n variables can be implemented by a n to 2n line decoder. To do this we have to express the Boolean function in terms of min-terms.

Suppose we have a Boolean function, F(A, B, C) = ∑ (1, 3, 5, 6). We need a 3-to-8-line decoder. We have to simply take output lines 1, 3, 5 and 6. And then we have to OR them. The output will be our Boolean function F.

A decoder can implement multiple Boolean functions together in a circuit. Say F has a combination min-terms, G has another combination of min-terms and H has another combination of min-terms. We can implement all three Boolean functions together with a decoder.

* **Boolean Function Implementation with Multiplexer :**

A multiplexer with n Selection Lines can implement a Boolean function with n + 1 variables. If we look inside a multiplexer, a multiplexer is basically a decoder with a built in OR gate. In decoder we would OR the output lines but in multiplexer OR is built-in.

Let’s say we have a 3 variable Boolean function, F(A, B, C). We will put B and C into the Selection Lines. Now depending on the combination of B and C, any of four input line will be connected to the output. Now our goal is feeding the correct input line with correct data. Some input lines don’t depend on the remaining variable A. They are feed with constant input 0 or 1. Other input lines depend on A. They are feed with A or A’ as needed.

Here is an example. Let’s say our Boolean function is, F(A, B, C) = ∑ (1, 3, 5, 6). The truth table is,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Minterm** | **A** | **B** | **C** | **F** |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 0 |
| 5 | 1 | 0 | 1 | 1 |
| 6 | 1 | 1 | 0 | 1 |
| 7 | 1 | 1 | 1 | 0 |

Here four colors represent four combinations of B and C a.k.a. four input lines.

Yellow -> Input Line 0 connected to output. F is 0 in both cases where A = 0 and A = 1. So, F is independent of A and constantly 0. So, constant 0 is given to Input Line 0.

Green -> Input Line 1 is connected to output. F is 1 in both cases where A = 0 and A = 1. So, F is independent of A and constantly 1. So, constant 1 is given to Input Line 1.

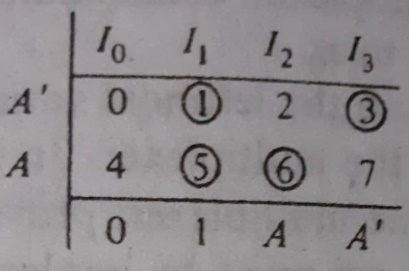
Blue -> Input Line 2 is connected to output. F is 0 when A is 0 and F is 1 when A is 1. So, it can be said that F = A. So, A is given to Input Line 2.

Purple -> Input Line 3 is connected to Output. F is 0 when A is 1 and F is 1 when A is 0. So, it can be said that F = A’. So, A’ is given to Input Line 3.

Thus, we can implement any Boolean function of n + 1 variables with multiplexer that has n Selection Lines.

There is an easy technique to perform the procedure described above.

1. Write the input lines in a row
2. Write all the min-terms dividing into two rows
3. Write A’ and A at the left of those two rows
4. Circle down the min-terms of the function
5. Inspect each column separately



If min-terms in a column are not circled, put constant 0 to the corresponding Input line.

If both min-terms in a column are circled, put constant 1 to the corresponding Input line.

If bottom min-term is circled but top min-term is not circled put A to the corresponding Input line.

If top min-term is circled but bottom min-term is not circled put A’ to the corresponding Input line.

* **Cascaded Decoders :**

Two decoders with enable pins can be cascaded together. The enable pin is used like the extra bit and MSB. In our logic diagram we have constructed 4-to-16-Line Decoder by cascading two 3-to-8-Line decoders.

* **Cascaded Synchronous Counter :**

Two synchronous counters can be cascaded to form a bigger counter. One counter’s chained AND Gates’ output is given to other counter’s first flip-flop’s J and K pins as inputs. Thus, two synchronous counters are cascaded.

* **Read Only Memory (ROM) :**

A ROM is a device that includes both the decoder and the OR gates within a single IC package. It is used to implement complex combinational circuits.

As an example, say we have a Seven-Segment Display. We want the display to show octal digits that means 0 – 7. A Seven-Segment Display has 8 pins in total. So, we need 8 Boolean Functions for 8 pins of three variables A, B and C which are essentially the 3 bits needed for the octal number system 000 to 111. This is where the ROM comes in handy.

In our logic diagram of ROM, if we enter 001, we get output pin P1 and P7 turned High. So, the output we get is the word (Binary string) 10000010. This 001 is said to be an address which contains the Read-Only word 10000010.

Thus, a ROM using n to 2n decoder has 2n addresses containing at most 2n fixed (Read-Only) words each of length say m. This ROM is described as 2nxm ROM.

Which of the outputs of the decoder will be connected to a particular OR gate inside a ROM is configured by “Programming” the ROM.

The programming of the ROM is called Mask Programming which is done by the manufacturer during the last step of fabrication. There is another type of ROM called Programmable ROM (PROM). The links in the PROM are broken by application of current pulse through the output terminals. In both cases mentioned above this programming process is irreversible. There is another type of ROM called Erasable PROM (EPROM) which can be restructured in initial state using special UV light. Another type of ROM can be erased with electrical signals instead of UV which is called Electrically Alterable ROM (EAROM). Another type of ROM is called Electrically Erasable PROM (EEPROM). EEPROMs are organized as arrays of floating-gate transistors. EEPROMs can be programmed and erased in-circuit, by applying special programming signals. Now-a-days microcontrollers use EEPROMs.