UNIT I

NUMBER SYSTEMS AND DIGITAL LOGIC FAMILIES

We are all familiar with the common base 10 (decimal) number system. It's the one we use to balance our checkbook and calculate the gas mileage of our automobiles. Obviously, this base is predicated on the fact that we have ten fingers or digits. Even the word digit is synonymous with a member of a number. Perhaps if we had evolved with eight fingers on each hand, we would all be using a base built upon sixteen digits instead of ten. Such a number base is termed hexadecimal. Did you realize that to add any two decimal numbers, you had to memorize 100 rules? Hundreds more are required for subtraction, multiplication, and division. Although they are simple (e.g. 1+2=3; 4+2=6, 2x5=10,4/2=2, etc), is it any wonder that they require several years to master? It is readily apparent that the larger the number base, the more rules you have to learn.

The simplest number system uses base 2 arithmetic and is termed the binary system. This scheme has only two digits or "bits" to work with, 0 and 1. The rules are few as shown by the addition examples in Table 1.

Table 1. Binary Addition Rules.

Rule	Operation	->	Carry	Result
1	0 + 0	=	0	0
2	0 + 1	=	0	1
3	1 + 0	=	0	1
4	1 + 1	=	1	1
5	1 + 1 + 1	=	1	1

Multiplications and division are accomplished trough repeated additions, and subtractions. Counting is merely another form of addition. By starting with zero and successively adding one, it is a simple matter to generate the following sixteen binary numbers: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111. Binary patterns such as these are often referred to as bit patterns. Each one or zero is a single bit, four bits are termed a nibble, and eight bits are a byte. Most microprocessors operate with 8 bits of data, and for these devices, a byte and a word are synonomous. The latest PCs have a 64-bit data pathway, and for these a word is 8 bytes. The smallest value bit is termed the Least Significant Bit (LSB) and the highest is the Most Significant Bit (MSB).

Although binary arithmetic is simple and easy to learn, it is cumbersome to use. The numbers tend to be lengthy and hard to remember, prone to transpositional errors, and difficult to vocalize. As an example, the number 98 is easy to pronounce and remember. Its binary representation is 1100010, and awkward number at best. Because computers use the binary number system, we must become familiar with binary and even hexadecimal numbers. The bases used most frequently are shown in Table 2.

Table 2. Common number base systems.

Base	Name	Bits	Comments
2	Binary	0,1	Language of computers
8	Octal	0-7	Based on groups of 3 binary bits
10	Decimal	0-9	Our common arithmetic base
16	Hexadecimal	0-F	Groups of 4 bits, requires A to F extensions to our Arabic numbers 0-9

Converting Binary to Decimal

Converting numbers to their decimal equivalents is easy, once you know how to expand an ordinary decimal number into its component parts. For example, 653 is 600 + 50 + 3. The position of the digits in a number like 653 determines the power of ten by which each digit is multiplied. Thus,

$6 \times 10^2 = 600$
$5 \times 10^1 = 50$
$3 \times 10^0 = 3$

653

Binary numbers can be expanded in an identical manner, and then converted to their decimal counterparts. Again taking the binary number 1001,

$1 \times 2^3 = 1000$
$0 \times 2^2 = 0000$
$0 \times 2^1 = 0000$
$1 \times 2^0 = 0001$

1001

If we now carry this one step further and covert the multipliers into their decimal equivalents, we obtain

$1 \times 2^3 = 1 \times 8 = 8$
$0 \times 2^2 = 0 \times 4 = 0$
$0 \times 2^1 = 0 \times 2 = 0$
$1 \times 2^0 = 1 \times 1 = 1$

1001 = 9

A method I prefer is to list the ascending powers of two over the binary digit and add those which have a one and ignore those numbers with a zero. Thus, to convert 1100110 to decimal, we get:

64 32 16 8 4 2 1
1 1 0 0 1 1 0
64 + 32 + 4 + 2 = 102

Or, written in another manner, $(1100110)_2 = (102)_{10}$

Converting Decimal Numbers to Binary

A quick way to convert decimal numbers into their binary counterparts is to repeatedly divide the decimal number by two. The remainders of each division (always a 1 or 0) will be the binary number. As an example, we will convert 102 into decimal. In Table 3, we repeatedly divide the left number by 2 and insert the remainder in the right column. We do this until we get zero in all columns.

Table 3. Algorithm for decimal to binary conversion.

Operand	Divide by 2	Remainder	Comment
102	51	0	LSB
51	25	1	
25	12	1	
12	6	0	
6	3	0	
3	1	1	
1	0	1	MSB
0	0	0	

The result is that $(102)_{10} = (1100110)_2$. It is left to you to verify that this works (i.e. – try other numbers to test the method).

Hexadecimal Numbers

Occasionally we find a need in microprocessor applications to use hexadecimal numbers. This is a numbering system using a base of 16. In order to do this, we have to invent new symbols for quantities whose value exceeds 9 as shown in Table 4.

Table 4. Decimal and Hexadecimal number comparisons.

Base 10	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11
18	12
19	13

Hexadecimal numbers are especially useful when we want to describe large binary numbers. As an example, consider the binary number 1111 0101 1011 0001 which we have grouped in nibbles (4-bits). Each nibble converts to a single hexadecimal number. Our large 16-digit binary number is written in hex as a 4-digit number, F5B1.

Binary codes

Character	Binary Code	Character	Binary Code	Character	Binary Code	Character	Binary Code	Character	Binary Code
A	01000001	Q	01010001	g	01100111	w	01110111	-	00101101
B	01000010	R	01010010	h	01101000	x	01111000	.	00101110
C	01000011	S	01010011	i	01101001	y	01111001	/	00101111
D	01000100	T	01010100	j	01101010	z	01111010	0	00110000
E	01000101	U	01010101	k	01101011	!	00100001	1	00110001
F	01000110	V	01010110	l	01101100	"	00100010	2	00110010
G	01000111	W	01010111	m	01101101	#	00100011	3	00110011
H	01001000	X	01011000	n	01101110	$	00100100	4	00110100
I	01001001	Y	01011001	o	01101111	%	00100101	5	00110101
J	01001010	Z	01011010	p	01110000	&	00100110	6	00110110
K	01001011	a	01100001	q	01110001	'	00100111	7	00110111
L	01001100	b	01100010	r	01110010	(	00101000	8	00111000
M	01001101	c	01100011	s	01110011	)	00101001	9	00111001
N	01001110	d	01100100	t	01110100	*	00101010	?	00111111
O	01001111	e	01100101	u	01110101	+	00101011	@	01000000
P	01010000	f	01100110	v	01110110	,	00101100	_	01011111

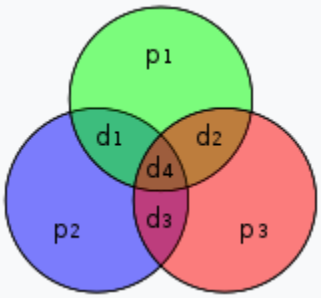
Hamming code

Hamming codes are a family of linear error-correcting codes that generalize the Hamming(7,4)-code, and were invented by Richard Hamming in 1950. Hamming codes can detect up to two-bit errors or correct one-bit errors without detection of uncorrected errors. By contrast, the simple parity code cannot correct errors, and can detect only an odd number of bits in error. Hamming codes are perfect codes, that is, they achieve the highest possible rate for codes with their block length and minimum distance of three.[1]

In mathematical terms, Hamming codes are a class of binary linear codes. For each integer $r \geq 2$ there is a code with block lengthn $= 2r - 1$ and message length $k = 2r - r - 1$. Hence the rate of Hamming codes is $R = k / n = 1 - r / (2r - 1)$, which is the highest possible for codes with minimum distance of three (i.e., the minimal number of bit changes needed to go from any code word to any other code word is three) and block length $2r - 1$. The parity-check matrix of a Hamming code is constructed by listing all columns of length r that are non-zero, which means that the dual code of the Hamming code is the shortened Hadamard code. The parity-check matrix has the property that any two columns are pairwise linearly independent.

Due to the limited redundancy that Hamming codes add to the data, they can only detect and correct errors when the error rate is low. This is the case in computer memory (ECC memory), where bit errors are extremely rare and Hamming codes are widely used. In this context, an extended Hamming code having one extra parity bit is often used. Extended Hamming codes achieve a Hamming distance of four, which allows the decoder to distinguish between when at

most one one-bit error occurs and when any two-bit errors occur. In this sense, extended Hamming codes are single-error correcting and double-error detecting, abbreviated as **SECDED**

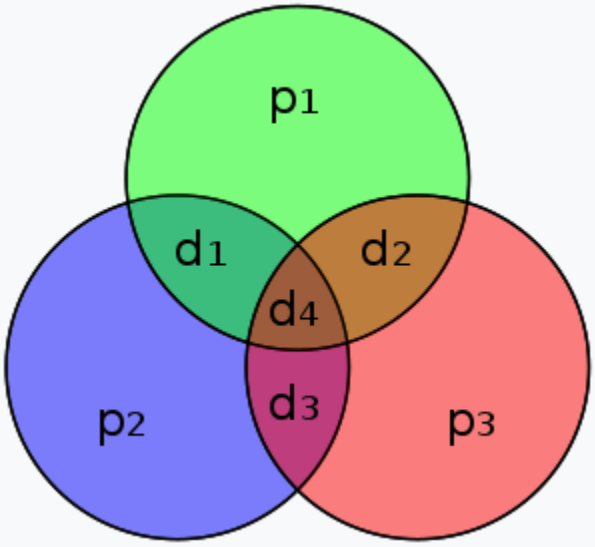


The Hamming(7,4)-code (with $r = 3$)

Named after	Richard W. Hamming
Classification	
Type	Linear block code
Block length	$2^r - 1$ where $r \geq 2$
Message length	$2^r - r - 1$
Rate	$1 - r/(2^r - 1)$
Distance	3
Alphabet size	2
Notation	$[2^r - 1, 2^r - r - 1, 3]_2$-code
Properties	
perfect code	
v	
t	
e	

In telecommunication, **Hamming codes** are a family of linear error-correcting codes that generalize the Hamming(7,4)-code, and were

[7,4] Hamming code[edit]



Graphical depiction of the four data bits and three parity bits and which parity bits apply to which data bits

In 1950, Hamming introduced the [7,4] Hamming code. It encodes four data bits into seven bits by adding three parity bits. It can detect and correct single-bit errors. With the addition of an overall parity bit, it can also detect (but not correct) double-bit errors.

Construction of G and H

The matrix is called a (canonical) generator matrix of a linear (*n*,*k*) code,

and is called a parity-check matrix.

This is the construction of **G** and **H** in standard (or systematic) form. Regardless of form, **G** and **H** for linear block codes must satisfy

, an all-zeros matrix.[3]

Since [7, 4, 3] = [*n*, *k*, *d*] = [$2^m - 1$, $2^m - 1 - m$, *m*]. The parity-check matrix **H** of a Hamming code is constructed by listing all columns of length *m* that are pair-wise independent.

Thus **H** is a matrix whose left side is all of the nonzero n-tuples where order of the n-tuples in the columns of matrix does not matter. The right hand side is just the ($n - k$)-identity matrix.

So **G** can be obtained from **H** by taking the transpose of the left hand side of **H** with the identity k-identity matrix on the left hand side of **G**.

The code generator matrix and the parity-check matrix are:

and

Finally, these matrices can be mutated into equivalent non-systematic codes by the following operations:

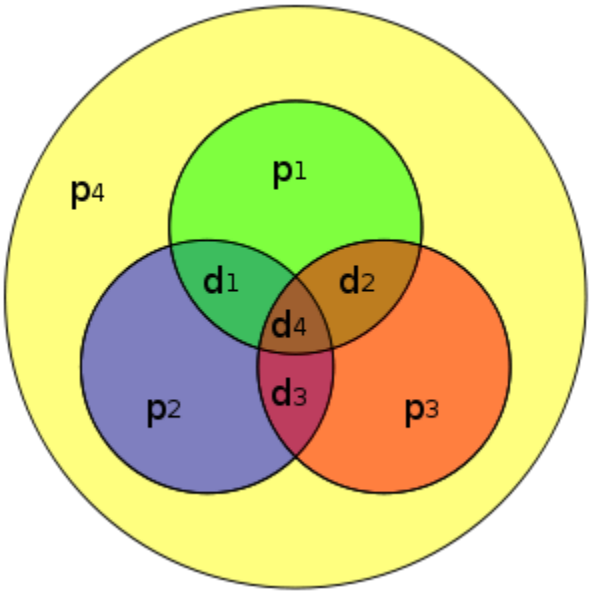
- Column permutations (swapping columns)
- Elementary row operations (replacing a row with a linear combination of rows)

Encoding
Example

From the above matrix we have $2^k = 2^4 = 16$ codewords. Let be a row vector of binary data bits, . The codeword for any of the 16 possible data vectors is given by the standard matrix product where the summing operation is done modulo-2.

For example, let . Using the generator matrix from above, we have (after applying modulo 2, to the sum),

[7,4] Hamming code with an additional parity bit



The same [7,4] example from above with an extra parity bit. This diagram is not meant to correspond to the matrix H for this example.

The [7,4] Hamming code can easily be extended to an [8,4] code by adding an extra parity bit on top of the (7,4) encoded word (see Hamming(7,4)). This can be summed up with the revised matrices:

and

Note that H is not in standard form. To obtain G, elementary row operations can be used to obtain an equivalent matrix to H in systematic form:

For example, the first row in this matrix is the sum of the second and third rows of H in non-systematic form. Using the systematic construction for Hamming codes from above, the matrix A is apparent and the systematic form of G is written as

The non-systematic form of G can be row reduced (using elementary row operations) to match this matrix.

The addition of the fourth row effectively computes the sum of all the codeword bits (data and parity) as the fourth parity bit.

For example, 1011 is encoded (using the non-systematic form of G at the start of this section) into 01100110 where blue digits are data; red digits are parity bits from the [7,4] Hamming code; and the green digit is the parity bit added by the [8,4] code. The green digit makes the parity of the [7,4] codewords even.

Finally, it can be shown that the minimum distance has increased from 3, in the [7,4] code, to 4 in the [8,4] code. Therefore, the code can be defined as [8,4] Hamming code.

DIGITAL LOGIC FAMILIES

In Digital Designs, our primary aim is to create an Integrated Circuit (IC). A Circuit configuration or arrangement of the circuit elements in a special manner will result in a particular Logic Family. Electrical Characteristics of the IC will be identical. In other words, the different parameters like Noise Margin, Fan In, Fan Out etc will be identical. Different ICs belonging to the same logic families will be compatible with each other.

The basic Classification of the Logic Families are as follows:
A) Bipolar Families
B) MOS Families
C) Hybrid Devices

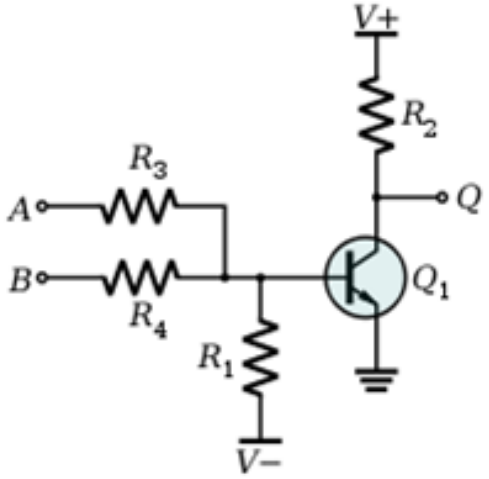
A) Bipolar Families:
1. Diode Logic (DL)
2. Resistor Transistor Logic (RTL)
3. Diode Transistor Logic (DTL)
4. Transistor- Transistor Logic (TTL)
5. Emitter Coupled Logic (ECL) or Current Mode Logic (CML)
6. Integrated Injection Logic (IIL)
B) MOS Families:
1. P-MOS Family
2. N-MOS Family
3. Complementary-MOS Family
• Standard C-MOS
• Clocked C-MOS
• Bi-CMOS
• Pseudo N-MOS
• C-MOS Domino Logic
• Pass Transistor Logic
C) Hybrid Family:
Bi-CMOS Family

Diode Logic

In DL (diode logic), only Diode and Resistors are used for implementing a particular Logic. Remember that the Diode conducts only when it is Forward Biased.
Disadvantages of Diode Logic.
• Diode Logic suffers from voltage degradation from one stage to the next.
• Diode Logic only permits OR and AND functions.

Resistor Transistor Logic

In RTL (resistor transistor logic), all the logic are implemented using resistors and transistors. One basic thing about the transistor (NPN), is that HIGH at input causes output to be LOW (i.e. like a inverter). In the case of PNP transistor, the LOW at input causes output to be HIGH.



Advantage:

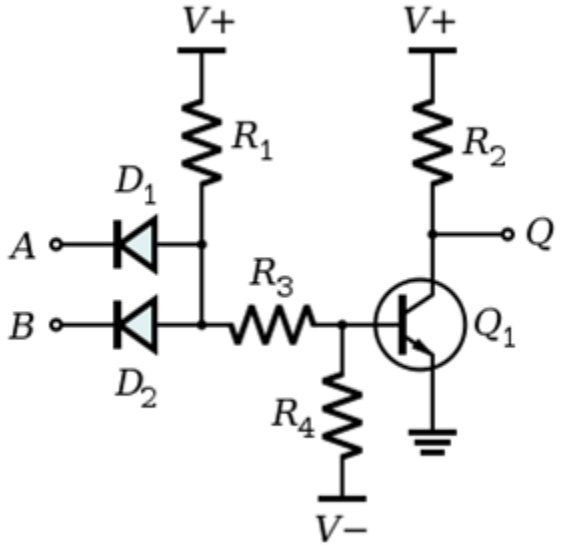
- Less number of Transistors

Disadvantage:

- High Power Dissipation
- Low Fan In

Diode Transistor Logic

In DTL (Diode transistor logic), all the logic is implemented using diodes and transistors.



Disadvantage:

• Propagation Delay is Larger

Transistor Transistor Logic

In Transistor Transistor logic or just TTL, logic gates are built only around transistors.
TTL Logic has the following sub-families:

• Standard TTL.
• High Speed TTL
• Low Power TTL.
• Schhottky TTL.
• Low Power Schottky TTL
• Advanced Schottky TTL
• Advanced Low Power Schottky TTL
• Fast Schottky

Emitter Coupled Logic

The main specialty of ECL is that it is operating in Active Region than the Saturation Region. That is the reason for its high speed operation. As you can see in the figure, the Emitters of the Transistors Q1 and Q2 are coupled together.

Disadvantage:

• Large Silicon Area
• Large Power Consumption

Some Characteristics we consider for the selection of a particular Logic Family are:
• Supply voltage range
• Speed of response
• Power dissipation
• Input and output logic levels
• Current sourcing and sinking capability
• Fan in
• Fan-out
• Noise margin

Introduction of Digital logic families

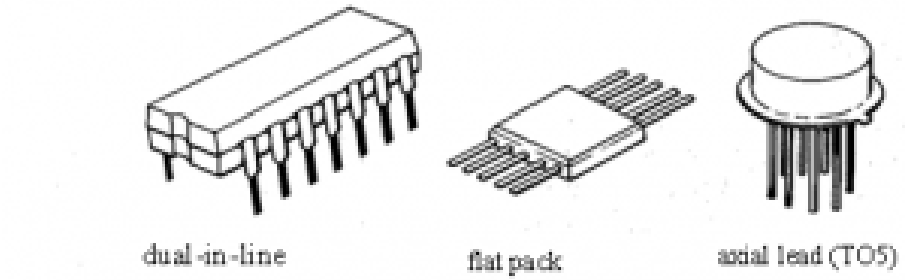
Miniature, low-cost electronics circuits whose components are fabricated on a single, continuous piece of semiconductor material to perform a high-level function. This IC is usually referred to as a monolithic IC first introduced in 1958. The digital ICs are categorized as,
1. Small scale integration SSI <12 no of gates
2. Medium scale integration MSI 12 to 99 no of gates
3. Large scale integration LSI 100 to 9999 no of gates
4. Very large scale integration VLSI 10,000 or more

In this section, we will be concern only with the digital IC. Digital IC can be further categorized into bipolar or unipolar IC.
Bipolar ICs are devices whose active components are current controlled while unipolar ICs are devices whose active components are voltage controlled.

IC Packaging

1. IC packaging Protect the chip from mechanical damage and chemical contamination.
2. Provides a completed unit large enough to handle.
3. So that it is large enough for electrical connections to be made.
4. Material is molded plastic, epoxy, resin, or silicone. Ceramic used if higher thermal dissipation capabilities required. Metal/glass used in special cases.

Three most common packages for ICs are
a) dual-in-line (DIPS) (most common)
b) flat pack
c) axial lead (TO5)



Characteristics of Digital ICs

Input /Output voltage level:

The following currents and voltages are specified which are very useful in the design of digital systems.

High-level input voltage, VIH : This is the minimum input voltage which is recognized by the gate as logic 1.
Low-level input voltage, VIL: This is the maximum input voltage which is recognized by the gate as logic 0.
High-level output voltage, VOH: This is the minimum voltage available at the output corresponding to logic 1.
Low-level output voltage, VOL: This is the maximum voltage available at the output corresponding to logic 0.
High-level input current, IIH : This is the minimum current which must be supplied by a driving source corresponding to 1 level voltage.
Low-level input current, IIL: This is the minimum current which must be supplied by a driving source corresponding to 0 level voltage.
High-level output current, IOH: This is the maximum current which the gate can sink in 1 level.
Low-level output current, IOL: This is the maximum current which the gate can sink in 0 level.
High-level supply current, ICC (1): This is the supply current when the output of the gate is at logic 1.
Low-level supply current, ICC (0): This is the supply current when the output of the gate is at logic (0).
Propagation Delay:
Definition: The time required for the output of a digital circuit to change states after a change at one or more of its inputs. The speed of a digital circuit is specified in terms of the propagation delay time. The delay times are measured between the 50 percent voltage levels of input and output waveforms. There are two delay times, *tp*HL: when the output goes from the HIGH state to the LOW state and *tp*LH, corresponding to the output making a transition from the LOW state to the HIGH state. The propagation delay time of the logic gate is taken as the average of these two delay times.
Fan-in
Defination: Fan-in (input load factor is the number of input signals that can be connected to a gate without causing it to operate outside its intended operating range. expressed in terms of standard inputs or units loads (ULs).
Fan-out
Defination:*Fan-out (output load factor)* is the maximum number of inputs that can be driven by a logic gate. A fanout of 10 means that 10 unit loads can be driven by the gate while still maintaining the output voltage within specifications for logic levels 0 and 1.
Digital IC gates are classified not only by their logic operation, but also by the specific logic circuit family to which it belongs. Each logic family has its own basic electronic circuit upon which more complex digital circuits and functions are developed.
Different types of logic gate families :
RTL : Resistor Transistor Logic gate family
DCTL : Direct Coupled Transistor Logic gate family
RCTL : Resistor Capacitor Transistor Logic gate family
DTL : Diode Transistor Logic gate family
TTL : Transistor Transistor logic gate family
IIL : Integraeted Injection gate family

Parameter	RTL Logic Family	IIL	DTL	Standard TTL	ECL	MOS	CMOS
Basic Gate	NOR	NOR	NAND	NAND	OR-NAOR	NAND	NOR – NAND
Power Dissipation in mW per gate	12	6 nW – 70 uW	8-12	10	40-55	0.2 - 10	1.01
Fan Out	5	Depends on injector current	8	10	25	20	50
Noise Immunity	Normal	Poor	Good	Very Good	Poor	Good	Very Good
Propagation Delay in ns per second	12	25-250	30	10	2	300	70
Speed Power Product	144	< 1	300	100	100	60	d.c. -0.7

- CMOS and TTL Interfaces
- CMOS Logic family
- Noise Margin
- TTL Logic family

Comparison of RTL, DTL, TTL, ECL and MOS families –operation

General characteristics of the TTL families (HC, HCT, LS...)
First there was normal TTL (Transistor Transistor Logic). There were also things like DTL (Diode Transistor Logic).

TTL became very popular, very soon, although CMOS was also already used by many other people.
CMOS and TTL were quite different in handling and levels. CMOS was 3V to 15V. TTL was 4.75V to 5.25V. CMOS used much less energy because it was high-impedance. In CMOS every connection between the power lines, always has two transistors in it's path of which one is always closed.

Comment by Ian Cox of the UK:

CMOS doesn't have NPN & PNP transistors but CMOS N-channel and P-Channel FETS and it is actually in CMOS that BOTH devices turn on and short out the supply for micro-moment during switching. I am not aware of this phonomenom in TTL logic (even old stuff).

CMOS was much more sensitive to static electricity.

After a while LS (Low Power Schottky) TTL was invented, probably by Schottky. It used much less energy and was as fast as normal TTL.
Everybody started using LS. (Except for the fast versions: S and FACT and FAST.)
These chips were for example used for a single critical chip on a board, like a (primitive) MMU which would otherwise slow down the memory access to much.

TTL with it's higher power usage, had always been faster than CMOS. Therefore CMOS was more used for analogue and TTL for digital circuits, but after some time, they invented HC (High speed CMOS) and made (High speed CMOS, TTL-compatible) with it.
This means, that internally a HCT chip is completely High speed CMOS and at the in and outputs are converters, that convert the levels to TTL.

Comment by Ian Cox of the UK:

You advise the use of HC, but AC (Advanced CMOS) or ACT (same again but it's input levels have been doctored to be compatible with old-fashioned TTL) has higher current drive capability (24mA!) and is generally faster still!

HCT is mostly compatible with TTL, except:

- You should never leave an input open, because the circuit can start to oscillate, which costs a lot of energy and can disturb the working of the rest of the circuit.
- You can't use the 74HCT04 as an oscillator (with a crystal), as was often done. But they designed a special version of the 74HCT04 for this: The 74HCU04!

Conclusion:

- Normally use HCT.
- If you want faster chips, use FACT/FAST or whatever.
- You can replace TTL-chips on old boards generally with HCT, unless it's an 7404 near a crystal or the designer of the board relies on the (slow) speed of certain components, which he should never have done!

What is the difference between the 54 and 74 family?

The 54 family is meant for military purposes.
This means it is guaranteed over a larger temperature range and is more expensive. But civilians may also buy it... (When you want to make something that can also be used in cold or hot

climates for example.)

Is the power supply of CMOS only 5V?

Yes, but real CMOS generally can work over a great range of voltages.

In earlier days, chips needed all kinds of weird voltages. Now they generate their own (low current) voltage internally by way of a charge pump, if necessary.

Why would you mix TTL and CMOS devices?

In earlier days not all functions were available in both TTL and CMOS. Now you can get most of the CMOS 4000 series also in TTL. Type numbers are 74HCT4...

Why did TTL-to-CMOS and CMOS-to-TTL require interfacing devices

TTL worked at 0 to 5 volt and CMOS allowed/needed all kinds of strange voltages, but was less critical about these voltages. But there are a lot of different ways that CMOS can be used to implement designs!

Currently even 'TTL' is done in CMOS. HCT means 'High speed CMOS TTL compatible'.

What is ECL?

It means Emittor Coupled Logic and it is extensively used in high speed digital data handling systems. Some NASA sites use ECL to handle baseband data up to 300mbps. TTL just won't perform at that data rate.

J.R. "Zeke" Walton from NASA.

What is BiCMOS?

Bipolar CMOS? Bipolar is generally very fast. But lately CMOS is also very fast... I assume you know that CMOS always has two transistors (an NPN and PNP one) between any connection between the 0 and 5 volt power supplies of which always one is not conducting. Early TTL had very short moments in which the single transistor switched and shorted the 0 and 5 volt which used a lot of energy. DRAM (which had much more gates than TTL became so hot that they warned you never to check with your finger if they were getting hot... It was safer to moisten your finger first so the temperature would stay below 100 degrees C.

CMOS has been 'round for about as long as TTL was always much more careful with energy usage, but it's much harder to produce since it requires those PNP and NPN transistors on the same chip which is a very complicated process and CMOS used to be very slow. The current VLSI however is so dense that when it would have to be done in TTL that it would burn up immediately because of heat problems. That's why CMOS had to be gotten under control and all VLSI is done in CMOS by now. They even had to lower the power voltages to keep the heat production down.
GaAs was also considered or even used for very fast chips.
You probably also know that IBM had a couple of water cooled mainframe computers in a period that their processors couldn't be cooled well enough with air.

What to use in the daily practice?

Just use HCT (and NMOS if must be) components and other TTL-like stuff unless you know why you would want to use anything else... And try to find MCU's with as many peripherals already integrated to save on part costs, board space, CAD, debug and programming time and increase product reliability etc.

Additional question

From: ganswijk@xs4all.nl
To: Chipdir Mailing List
Subject: Re: Two Queries
At 11:54 19990217 -0800, Declan Moriarty wrote:
> I have one out of two queries on topic...above average perhaps ;-)
>
>1. Can anyone point me to a reference that tells me the difference
>between all the 74xx series logic families? I am finding it difficult
>to get some 7400 series chips locally, like today I had problems
>with the 74LS01. I need to know could I shove in a 74ALS01, or 74F01,
>or 74L01. What is the difference between 74HC and 74HCT...that sort of
>thing.

I have written a page about it:
http://www.chipdir.com/chipdir/ttl.htm

The generations of the most economical types were:

74
74LS
74HCT

These are basically compatible but every generation is faster than the last and uses much less energy. HCT is CMOS, so you need to connect all the inputs which was not needed with parts of the other generations. Also the 74ls04 that was often used in an oscillator circuit couldn't be replaced by the 74HCT04, but they produce a special version, the 74HCU04 that can be used in this special way.

All other types of 74 chips were faster or used less energy or whatever. They would have been more expensive, but if that is no problem I don't see a reason not to use them as drop-in replacements.

Beware that the 74HC's may be real CMOS and not TTL compatible as the HCT (=High Speed CMOS TTL compatible). HCT is CMOS that has been made to act as TTL, so 0 and 5V power and probably levels of 0.8 and 2.7 V for low and high.

When a design is really critical I'd check the datasheets, but I'd even do that with HCT...

Example

Since a lot of people seem to have trouble choosing, here a more practical example. Suppose you would like to build a 6809 Unix computer. This would have to involve a simple MMU (Memory Management Unit) which dynamically translates the upper 4 address lines of the 16 logical address to say 8 address lines to form a 20 bit physical address. The translation is called dynamical since it has to be done at every read/write to memory. This way the OS can assign every 4K physical page to every 4K logical page of any task as it chooses. The 6809 is a traditional processor which can't add wait states in his read and writes, so the time from address available to data read is fixed and limited. Normally there is enough time to select the correct chip from the address given en produce the data on the data lines, but with the added time required by the address translation mechanism the complete design can't be done in HCT. The translation mechanism consisting of two 164 bit chips has to be done in a faster technology like 74S. Most of the rest can be done in HCT. Just some components in the timing's most critical path will need to be done in faster technology. This only doubles (?) the cost of a few components, but only ups the cost of the total design a few procents considering the total number of chips involved.

By the way, we have built such a 6809 Unix computer around 1984 and used a couple of them for many years since then, both for database, accounting and embedded software writing. We never built a successor both because it was hard to find a good 16/32 bits processor and because the 80286 was already then a viable similarly functional system with MMU on-board and could run Xenix quite well. With the arrival of the 80386 there was absolutely no incentive to build our own computers anymore. Motherboards were getting cheaper and cheaper. Xenix was still expensive though. Linux changed this of course. MS Windows also became more and more a serious platform.

Overview

Bipolar Technology:

74	Standard TTL	This was the original series. Was superseded by 74LS and later 74HCT for general usage.
74L	Low power and lower speed.	Probably for portable applications.
74LS	Low power Schottky TTLI/O.	Uses Schottky barrier diodes (from memory between the base and collector) to prevent the transistors saturating, hence improving speed when they turn off. Probably about as fast as standard TTL.
74ALS	Advanced LS TTL	Faster than LS and higher output current .
74F	Fast TTL I/O.	Uses lots of current to achieve high speed. Probably not that fast any more.

CMOS Technology:

74C	CMOS	Uses CMOS transistors and hence has switching levels set at half supply unlike all of the above. These can usually be run of supplies from about 3 to 15V.
74AC	High speed TTL I/O	
74ACT	High speed TTL/CMOS Input CMOSOutput	
74FC	High speed TTL/CMOS Input CMOSOutput	
74HC	High speed CMOS	Faster using CMOS transistors, half supply switching. 5v only.
74HCT	High speed CMOS with TTL switching levels	Uses CMOS but designed to switch at TTL levels (ie low = <0.6V, high = >2V)
74AHC	Advanced HC?	Faster than HC?
74AHCT	Advanced HCT?	Faster than AHCT?

More subtle differences

By Andrew Ingraham

1. Can anyone point me to a reference that tells me the difference between all the 74xx series logic families? I am finding it difficult to get some 7400 series chips locally, like today I

had problems with the 74LS01. I need to know could I shove in a 74ALS01, or 74F01, or 74L01. What is the difference between 74HC and 74HCT...that sort of thing.

There are many subtle differences that might come into play when you go between families. Depends on your circuits.

In my opinion, the major differences from the user's perspective, are speed, and input thresholds. Some CMOS families use optimized "CMOS" levels while others use "TTL" or perhaps "LVTTL" levels. TTL's input threshold is around 1.4V, versus 2.5V (Vdd/2) for 5V-CMOS. TTL accepts 2.0V as "high"; CMOS would call this marginally "low" and needs a much higher voltage to be considered "high".

But when you come down to using them in your circuits, you also need to think about things like:

- Input current , in both high and low states. If you substitute a 74HCT with a 74F, the current (i.e., loading) is a lot higher.
- Output drive capability.
- Output drive symmetry. Is it suitable for driving long bus wires, or will pulsewidths suffer?
- Signal integrity. Faster gates make faster output edges. 74F is fast, but watch it if you need to drive more than a few inches (depending on what it drives).
- Power dissipation, if you have a lot of them.
- Clamping at inputs, I/Os, and tri-state outputs. If you use a 3.3V family, will it clamp above 3.3V, or is it 5V-tolerant? You can get both.
- Can unused inputs float? Some are OK with that, others aren't.
- Noise immunity. Faster gates will see noise glitches on their inputs that slower gates miss.

Get as many data books as you can and study them until you familiarize yourself with all these differences. What may be an acceptable substitute in one case, may be a flop in another.

Regards,
Andy

See also

NS's "High-Performance Logic Selection Guide".
TI's TTL databook. It gives a guite good overview of these logic-IC families, with properties, differences, etc...

Difference between 54 and 74 family

By Greg Smith

The specs for 54xx usually show them as being slower than 74xx, although in fact this is probably just a derating for the extended temp range.

ECL

By Greg Smith

(Read this in the voice of Grandpa Simpson:)

Back in the 70's, you had 7400 and 74S00. Maybe you were just getting 74LS00. If you wanted things to run at, say, 300 MHz, you could do it with ECL. ECL logic uses a -5.2 V supply, and switches above and below a certain threshold voltage, I think it's -1.2V. A lot of the devices had + and - outputs for the same function, or differential inputs. You could apply + and - ECL outputs to a twisted pair cable, run it a few feet to a different board, and apply the cable to a differential ECL input, and it would work at very high speeds. Since it switches current from one side to the other, rather than turning it on and off, and since the voltage swings are very small, ECL had far less noise problems than TTL and would run at high speeds on wire-wrapped boards. You needed to use terminating resistors on every signal, though. Lots of power.

I encountered stuff like this inside a 70's era Control Data disk drive. This thing had a 60 MB removable pack about 12" in dia and 6" thick. The entire unit was about the size of a modern office photocopier, and weighed more. The backplane was connected with wire-wrap. The termination resistors in the interface cable used more power than an entire modern HDD.

ECL is not really used any more.

74H00 - higher power, faster than 7400
74L00 - low power, slower than 7400.
These were used before 74S and 74LS, and were a direct power/speed tradeoff. They were already obsolete in the late 70's when I started tinkering with this stuff.

ECL, reaction

By Allan Warrington

Regarding the comment about ECL not being much use for anything.

ECL was quite important in the 1970s, 1980s and early 1990s. It was used for very high speed circuits. Early supercomputers e.g. Cray's were made with it. However, it was very power hungry. I think that it isn't much used nowadays. If you try to use any old ECL chips, the logic

levels are typically around 0.9V below VCC for logic high and 1.8V below supply for logic low. Supply is generally 0V and -5V, rather than 5V and 0V.

Propagation

It means transmission in this case and the propagation time or propagation delay is the time in which a signal travels through a gate. It's typically 4 ns for LS TTL, I think. A simple NAND consists of a single gate. An AND of two gates and for example a 74LS138 has usually several gates from input to output so it takes say 6 times 4 ns.

In serious PCB-designs you need to calculate how long it takes for signals to go from serious chip A to serious chip B through all the intermediate TTL-chips. If the signal takes too long the circuit may start to behave badly (at first at higher temperatures etc.)

First you try to calculate everything exactly and then at an extra proof you can put the finished product in an oven and heat it up to check if it still works. This will give a clear indication of how reliable it is.

Some more info ...

TTL normally uses only NPN transistors (although newer derivatives As a result, TTL can sink much better than it can source, whereas CMOS can do both fairly well and has "rail-to-rail" output voltage.

Joe da Silva

Z-state (high impedance)

A digital output is internally usually connected with one transistor to the 5V and one transistor to the 0V. Only one of those is normally conducting electrical current at a certain moment pulling the output pin either to the 0V or the 5V. When none of the two transistors is conducting, the output is in the Z-state (high impedance state).

The different logic families use different technology to implement the transistors, but the principle is probably the same.

Buy the way early chips were made in a technique whereby both of the transistors would conduct current for a short time during transitions. This made them actually short the 5V to the 0V for a very short time. Therefore these parts would use a lot of energy and become very hot. DRAM from around the time that about 16kbit per chip was producable could easily burn your finger so feeling the chip to debug the hardware was not and advisable idea. You should at least wet your finger first. If you like to repair old computers like the TRS-80, Apple2, Commodore PET, BBC etc. it's wise to remember this...

Open inputs

Specifically at this time I'm looking at a 74HC14 hex inverter, but would also like to know if there's a rule of thumb to follow i.e. drive inputs such that the output is always high, or low, or either of the two is OK.

It's not easy to determine what the *optimal* solution is.

The problem with CMOS inputs is that they are very high impedance and may pick up signals from the surrounding environment and the air and may start to oscillate. This may not only cause highly increased power usage but also other undesired effects.

So as long as you tie the inputs to a signal with well defined value at each point of time, like a neighboring data pin, either in- or output or 0V or 5V this problem will be prevented. For some technologies it seems to be recommended to connect the 5V via a relatively low-R resistor (4k7 for example). Power inputs of a chip can normally withstand a power surge, but inputs may not.

What the *optimal* solution is depends on all kinds of factors: Does the technology draw more current with a low or high input?
When connecting the input to a neighboring pin, you'll have to consider the load on the given signal. But all this is in most cases not very important. It's only relevant when for example battery life must optimal or signal speed is crucial. In practical cases I'd just solder it to the 0V or 5V if any is available on the next pin, or in case of a two-input (N)AND or (N)OR which is just used as an invertor, I'd solder both inputs together or otherwise it depends on which signals are available closeby and if they can drive an extra load (which they usually can) and if their reaction speed isn't crucial for the system (which it usally isn't).

In case of your hex invertor you could consider feeding certain signals to two invertors parallel to each other and also tying the outputs together. It will double the load on the input signal, but will also double the drive of the output signal. It might however increase the power usage during the moment of switching when the two gates should have significantly different switching times, but that is of course very unlikely. ;-)

UNIT II

COMBINATIONAL CIRCUITS

Introduction to Combinational Logic Functions

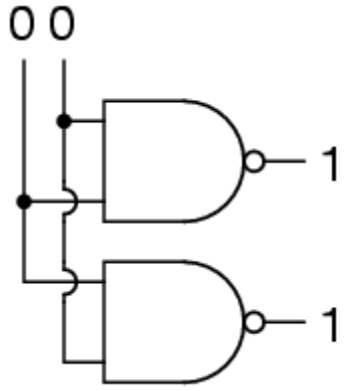
The term “combinational” comes to us from mathematics. In mathematics a combination is an unordered set, which is a formal way to say that nobody cares which order the items came in. Most games work this way, if you rolled dice one at a time and get a 2 followed by a 3 it is the same as if you had rolled a 3 followed by a 2. With combinational logic, the circuit produces the same output regardless of the order the inputs are changed.

There are circuits which depend on the when the inputs change, these circuits are called sequential logic. Even though you will not find the term “sequential logic” in the chapter titles, the next several chapters will discuss sequential logic.

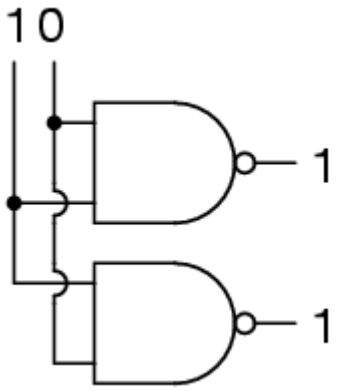
Practical circuits will have a mix of combinational and sequential logic, with sequential logic making sure everything happens in order and combinational logic performing functions like arithmetic, logic, or conversion.

You have already used combinational circuits. Each logic gate discussed previously is a combinational logic function. Let’s follow how two NAND gate works if we provide them inputs in different orders.

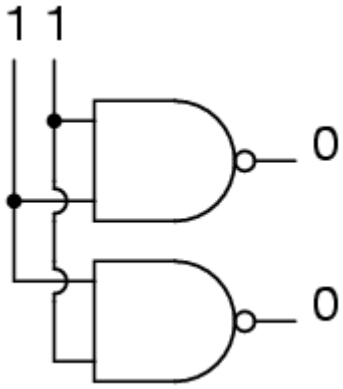
We begin with both inputs being 0.



We then set one input high.



We then set the other input high.



So NAND gates do not care about the order of the inputs, and you will find the same true of all the other gates covered up to this point (AND, XOR, OR, NOR, XNOR, and NOT)

SOP and POS form

Introduction

Let's set the stage for this discussion by defining a function with the following truth table.

#		A	B	C		F
0		0	0	0		0
1		0	0	1		1

#		A	B	C		F
2		0	1	0		0
3		0	1	1		1
4		1	0	0		1
5		1	0	1		0
6		1	1	0		1
7		1	1	1		1

Table 1 - Arbitrary function defined via Truth Table

The first column is merely a convenient label for each row obtained by interpreting the combination of inputs as a straight binary value.

As we will see shortly, we can express this function in two equivalent ways:

The first of these, Eqn (1), is the Standard Sum of Products, or Standard SOP, form while the second, Eqn (2), is the Standard Product of Sums, or Standard POS, form.

The term "standard" here means that the expression consists exclusively of **minterms** (in the case of Standard SOP) or **maxterms** (in the case of Standard POS). What a minterm and maxterm are will be discussed shortly.

As we can see, the names are quite descriptive -- the SOP form takes on the appearance of being a sum of several terms, each of which is the product of several factors while the POS form takes on the appearance of being a product of several factors, each of which is the sum of several terms.

We should keep in mind that "products" and "sums" are concepts from normal algebra; while we borrow the terms and use them in Boolean algebra, we do not mean "multiplication" or "addition" -- a "product" in Boolean algebra is a logical AND operation while a "sum" is a logical OR operation.

In this discussion, we are interested in how to go from a Truth Table to the SOP or POS forms and how to work with them, including translating between them, using Boolean algebraic manipulations only. So we will not be discussing topics such as Karnaugh maps except in passing.

Before we can talk about constructing the SOP and POS forms from the Truth Table, we need to define a couple of more fundamental concepts, namely minterms and maxterms.

Minterm

A "minterm" is a Boolean expression that is True for the minimum number of combinations of inputs; this minimum number is exactly one (the case of a term being True for no combination of inputs is, of course, simply a hard False and is trivial and uninteresting).

Because we want to minimize the coverage, we want to use the Boolean operation that is the most restrictive, which is the AND operation. Since the AND is true only if ALL of the inputs are True, we can craft an expression that is True for only a single combination of inputs by including each input in the product. If the input is uncomplemented, then we require that it be True, while if it is complemented, then we require it to be False. The minterm expression for each combination of inputs is therefore

#		A	B	C		minterm
0		0	0	0		$\overline{A}\,\overline{B}\,\overline{C}$
1		0	0	1		$\overline{A}\,\overline{B}C$
2		0	1	0		$\overline{A}B\,\overline{C}$
3		0	1	1		$\overline{A}BC$
4		1	0	0		$A\,\overline{B}\,\overline{C}$
5		1	0	1		$A\,\overline{B}C$
6		1	1	0		$AB\,\overline{C}$
7		1	1	1		ABC

Table 2 - Table of Minterms

Keep in mind that each minterm is True only for the specific combination of inputs with which it is associated and is False for all others.
In order to be a minterm, the product must cover exactly one of the possible combination of inputs. To do this, each input must be present in either uncomplemented or complemented form. Consider the system of three inputs in the table above. We can combine the last two minterms, namely ABC'+ABC, into a single product, namely AB. Using this "simplifies" the expression and it is still in SOP form, but it is not a minterm and the resulting expression is no longer in standard -- i.e., canonical -- SOP form. This may or may not be important.

If we have an expression that is in SOP form but is not in standard form and we need it to be, then we can simply expand each product by ANDing it with products that are constructed by ORing one of the missing inputs and its complement. Thus, we can expand AB back to minterm form as follows:

$$AB = AB(C+\overline{C}) = ABC + AB\overline{C}$$

Maxterm

A "maxterm" is a Boolean expression that is True for the maximun number of combinations of inputs; this maximum number is exactly one fewer than the total number of possibilities (the case of a term being True for all combinations is, of course, simply a hard True and is trivial and uninteresting).

Because we want to maximize the coverage, we want to use the Boolean operation that is the most permissive, which is the OR operation. While the OR is True if ANY of its inputs is True, for our purposes it is more convenient to recognize that the OR is False only if ALL of its inputs are False; thus, we can craft an expression that is False for only a single combination of inputs by including each input in the sum. If the input is uncomplemented, then we require that it be False, while if it is complemented, then we require it to be True. The maxterm expression for each combination of inputs is therefore

#		A	B	C		maxterm
$\overline{0}$		0	0	0		$A+B+C$
$\overline{1}$		0	0	1		$A+B+\overline{C}$
$\overline{2}$		0	1	0		$A+\overline{B}+C$
$\overline{3}$		0	1	1		$A+\overline{B}+\overline{C}$
$\overline{4}$		1	0	0		$\overline{A}+B+C$
$\overline{5}$		1	0	1		$\overline{A}+B+\overline{C}$
$\overline{6}$		1	1	0		$\overline{A}+\overline{B}+C$
$\overline{7}$		1	1	1		$\overline{A}+\overline{B}+\overline{C}$

Table 3 - Table of Maxterms

Keep in mind that each maxterm is False only for the specific combination of inputs with which it is associated and is True for all others.

In order to be a maxterm, the sum must cover all but exactly one of the possible combination of inputs. To do this, each input must be present in either uncomplemented or complemented form.

Consider the system of three inputs in the table above. We can combine the last two maxterms, namely (A'+B'+C)(A'+B'+C'), into a single sum, namely (A'+B'). Using this "simplifies" the expression and it is still in POS form, but it is not a maxterm and the resulting expression is no longer in standard -- i.e., canonical -- POS form. This may or may not be important.

If we have an expression that is in POS form but is not in standard form and we need it to be, then we can simply expand each sum by ORing it with terms that are constructed by ANDing one of the missing inputs and its complement. Thus, we can expand (A'+B') back to maxterm form as follows:

The above equation may look odd but that is because while we are used to multiplication being distributable over addition (and thus AND being distributable over OR looks natural), we are used to addition not being distributable over multiplication. However, OR is distributable over AND, hence

$$X+YZ = (X+Y)(X+Z)$$

In the above relation,

$$X = \overline{A}+\overline{B}$$

Relationship between Labels, Minterms, and Maxterms

Consider the minterm and maxterm for a particular row. Since the minterm is True ONLY when that input combination is asserted while the maxterm is False ONLY when that combination is asserted, it is clear that each minterm is the logical inverse of the corresponding maxterm (and vice versa, of course).

This relationship is quite obvious if we look at a particular example, say Row 3, and apply DeMorgan's Theorem to it.

The labels in Tables 2 are often used as a synonym for a minterm expression. This makes intuitive sense if we think of row N being "True" when the minterm associated with the row labeled N is True.
In contrast, since a maxterm is the logical inverse of a minterm, we can use an inverted label to refer to a maxterm, as shown in Table 3. Therefore it should be clear how this labeling convention is consistent with, for instance, applying DeMorgan's theorems and other Boolean operations.

Standard SOP (Sum of Products)

As mentioned in the introduction, the SOP form takes on the appearance of being a sum of several terms, each of which is the product of several factors.

To craft the SOP form of a Boolean logic function, we merely need to OR together the minterms associated with each combination of inputs for which the overall output should be True.

By looking at Table 1 we see that we need to sum the minterms associated with rows {1,3,4,6,7}. This is often represented simply as

$$(7) \quad F = \sum(1,3,4,6,7)$$

By expanding the summation and replacing each label with the corresponding minterm, we immediately obtain Eqn 1, which is the **canonical disjunctive form**. Don't let the fancy words confuse you; the term "canonical" just means "standardized" while the term "disjunctive" merely means a logical union, which is the same thing as a logical ORing of sets. This form is more commonly known, particularly among the more application-oriented, simply as the **Standard SOP form**.

Standard POS (Product of Sums)

As mentioned in the introduction, the POS form takes on the appearance of being a product of several factors, each of which is the sum of several terms.

To craft the POS form of a Boolean logic function, we merely need to AND together the maxterms associated with each combination of inputs for which the overall output should be False.

This is not as obvious as the need to OR together the minterms to get the SOP, so let's consider it for a moment. If we AND together several factors, the output will be False as long as any one of the factors is False. A maxterm is False for exactly one combination of inputs, so using a maxterm for a combination for which we want the overall output to be False will achieve this goal and, since that maxterm is True for all other combinations, it will have no effect on the output for them. As long as we do not include the maxterms for any combinations that we do want the output to be True for, we are guaranteed that all of the other maxterms in the expression will be True and, hence, the overall product of them will be True.

By looking at Table 1 we see that we need to take the product of the maxterms associated with rows {0',2',5'}. This is often represented simply as

$$(8) \quad F = \prod(\bar{0},\bar{2},\bar{5})$$

By expanding the product and replacing each label with the corresponding maxterm, we immediately obtain Eqn 2, which is the **canonical conjunctive form**. Don't let the fancy words confuse you; the term "canonical" just means "standardized" while the term "conjunctive" merely means a logical intersection, which is the same thing as a logical ANDing of sets. This form is more commonly known, particularly among the more application-oriented, simply as the **Standard POS form**.

Converting Boolean Expressions into SOP/POS Form

The process of converting any Boolean expression into either POS or SOP form (canonical or

otherwise) is very straightforward.

To get the expression in SOP form, you simply distribute all AND operations over any OR operations and continue doing this as long as possible. When finished, you will have an expression in SOP form. If you want it in canonical form, then you simply expand each term as necessary.

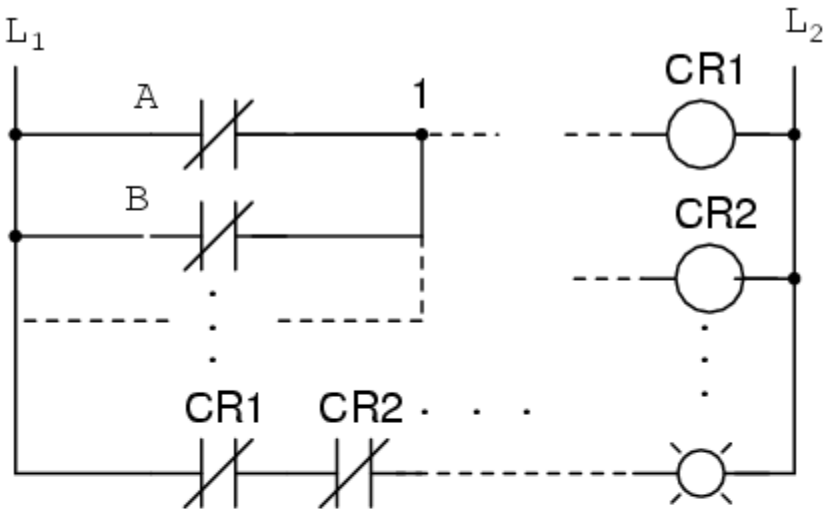
To get the expression in POS form, you simply distribute all OR operations over any AND operations and continue doing this as long as possible. When finished, you will have an expression in POS form. If you want it in canonical form, then you simply exapnd each term as necessary.

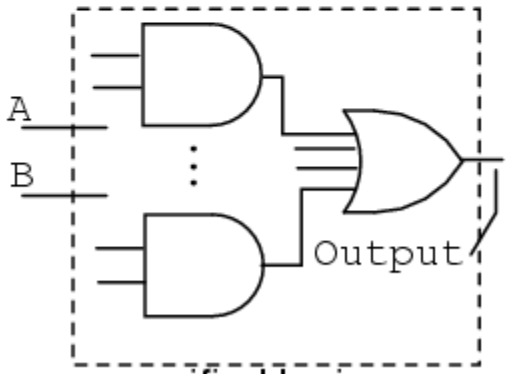
Most people have little problem getting an arbitrary expression into SOP form because the notation used for AND and OR are the same used for multiplication and addition in normal arithmetic and the notion of distributing multiplication over addition has long become internalized. But the fact that addition is not distributable over multiplication has also been internalized and hence doing something that looks the same does not come naturally. But just as AND can be distributed over OR, so too can OR be distributed over AND.

Karnaugh Maps, Truth Tables, and Boolean Expressions

aurice Karnaugh, a telecommunications engineer, developed the Karnaugh map at Bell Labs in 1953 while designing digital logic based telephone switching circuits.

Now that we have developed the Karnaugh map with the aid of Venn diagrams, let’s put it to use. Karnaugh maps reduce logic functions more quickly and easily compared to Boolean algebra. By reduce we mean simplify, reducing the number of gates and inputs. We like to simplify logic to a lowest cost form to save costs by elimination of components. We define lowest cost as being the lowest number of gates with the lowest number of inputs per gate. Given a choice, most students do logic simplification with Karnaugh maps rather than Boolean algebra once they learn this tool.





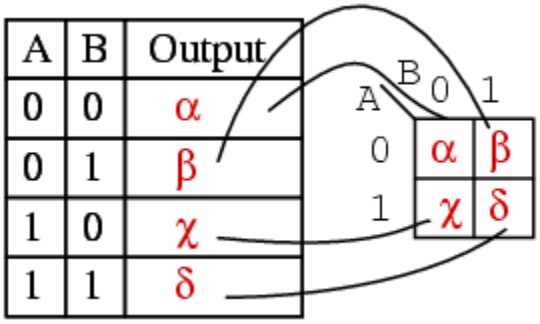
A	B	Output
0	0	α
0	1	β
1	0	χ
1	1	δ

A \ B	0	1
0	α	β
1	χ	δ

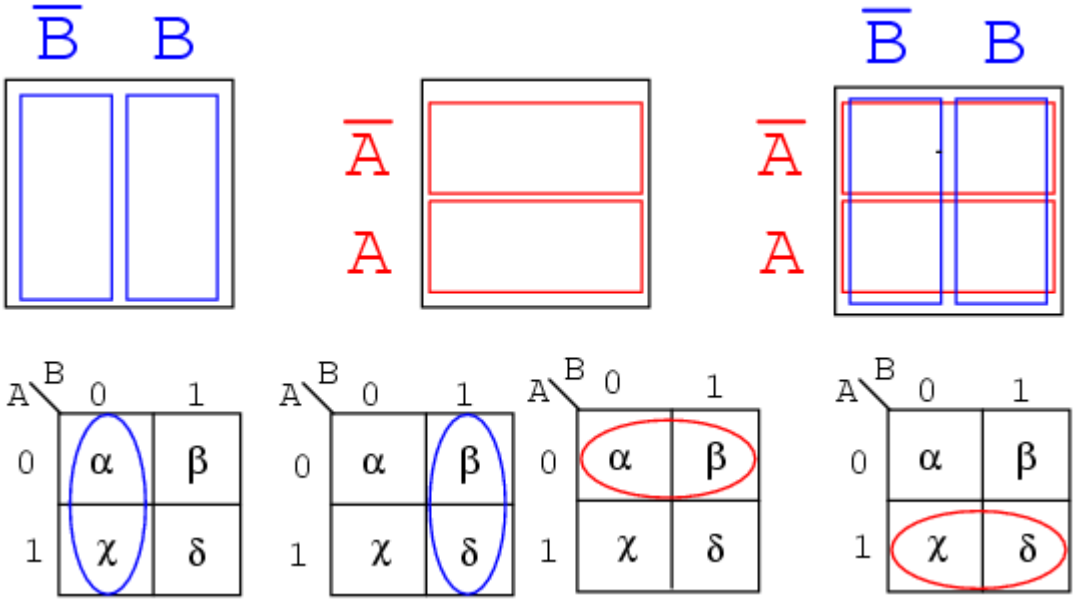
$$\text{Output} = \overline{ABC + ABC + \ldots ABC}$$

We show five individual items above, which are just different ways of representing the same thing: an arbitrary 2-input digital logic function. First is relay ladder logic, then logic gates, a truth table, a Karnaugh map, and a Boolean equation. The point is that any of these are equivalent. Two inputs A and B can take on values of either 0 or 1, high or low, open or closed, True or False, as the case may be. There are 22 = 4 combinations of inputs producing an output. This is applicable to all five examples.

These four outputs may be observed on a lamp in the relay ladder logic, on a logic probe on the gate diagram. These outputs may be recorded in the truth table, or in the Karnaugh map. Look at the Karnaugh map as being a rearranged truth table. The Output of the Boolean equation may be computed by the laws of Boolean algebra and transfered to the truth table or Karnaugh map. Which of the five equivalent logic descriptions should we use? The one which is most useful for the task to be accomplished.



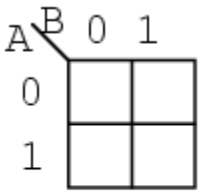
The outputs of a truth table correspond on a one-to-one basis to Karnaugh map entries. Starting at the top of the truth table, the A=0, B=0 inputs produce an output α. Note that this same output α is found in the Karnaugh map at the A=0, B=0 cell address, upper left corner of K-map where the A=0 row and B=0 column intersect. The other truth table outputs β, χ, δ from inputs AB=01, 10, 11 are found at corresponding K-map locations.
Below, we show the adjacent 2-cell regions in the 2-variable K-map with the aid of previous rectangular Venn diagram like Boolean regions.



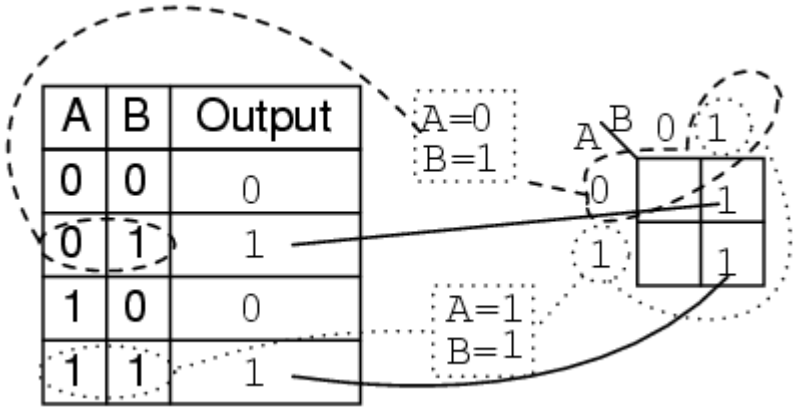
Cells α and χ are adjacent in the K-map as ellipses in the left most K-map below. Referring to the previous truth table, this is not the case. There is another truth table entry (β) between them. Which brings us to the whole point of the organizing the K-map into a square array, cells with any Boolean variables in common need to be close to one another so as to present a pattern that jumps out at us. For cells α and χ they have the Boolean variable B' in common. We know this because B=0 (same as B') for the column above cells α and χ. Compare this to the square Venn diagram above the K-map.

A similar line of reasoning shows that β and δ have Boolean B (B=1) in common. Then, α and β have Boolean A' (A=0) in common. Finally, χ and δ have Boolean A (A=1) in common. Compare the last two maps to the middle square Venn diagram.
To summarize, we are looking for commonality of Boolean variables among cells. The Karnaugh map is organized so that we may see that commonality. Let's try some examples.

A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1



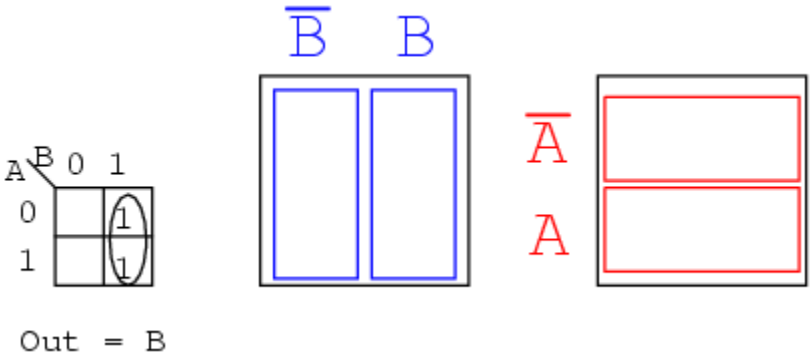
Example:
Transfer the contents of the truth table to the Karnaugh map above.



Solution:

The truth table contains two 1s. the K- map must have both of them. locate the first 1 in the 2nd row of the truth table above.
note the truth table AB address
locate the cell in the K-map having the same address
place a 1 in that cell
Repeat the process for the 1 in the last line of the truth table.
Example:

For the Karnaugh map in the above problem, write the Boolean expression. Solution is below.

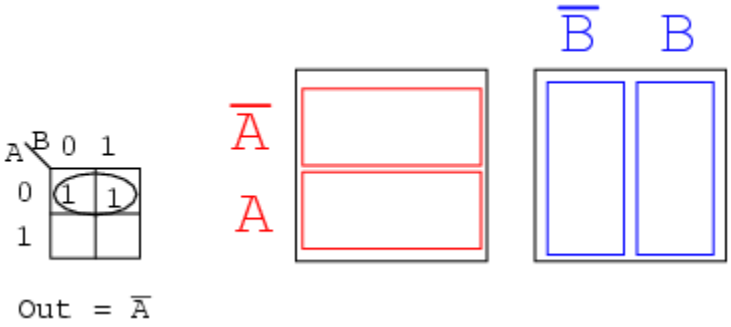


Solution:

Look for adjacent cells, that is, above or to the side of a cell. Diagonal cells are not adjacent. Adjacent cells will have one or more Boolean variables in common.
Group (circle) the two 1s in the column
Find the variable(s) top and/or side which are the same for the group, Write this as the Boolean result. It is B in our case.
Ignore variable(s) which are not the same for a cell group. In our case A varies, is both 1 and 0, ignore Boolean A.
Ignore any variable not associated with cells containing 1s. B' has no ones under it. Ignore B'
Result Out = B
This might be easier to see by comparing to the Venn diagrams to the right, specifically the B column.

Example:

Write the Boolean expression for the Karnaugh map below.

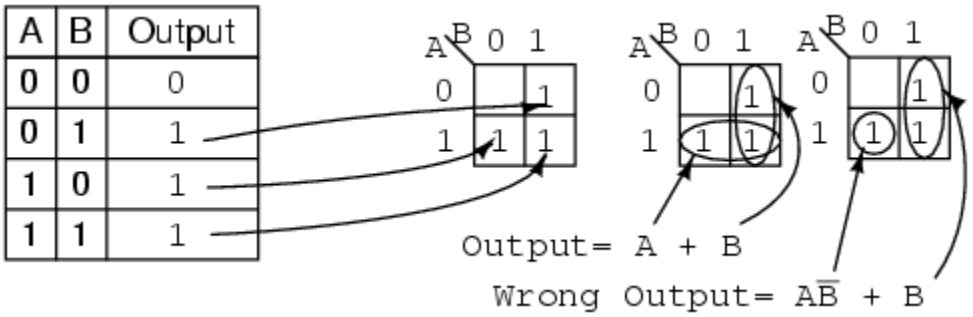


Solution: (above)

- Group (circle) the two 1's in the row
- Find the variable(s) which are the same for the group, Out = A'

Example:

For the Truth table below, transfer the outputs to the Karnaugh, then write the Boolean expression for the result.



Solution:

Transfer the 1s from the locations in the Truth table to the corresponding locations in the K-map.

- Group (circle) the two 1's in the column under **B=1**
- Group (circle) the two 1's in the row right of **A=1**
- Write product term for first group = **B**
- Write product term for second group = **A**
- Write Sum-Of-Products of above two terms **Output = A+B**

The solution of the K-map in the middle is the simplest or lowest cost solution. A less desirable solution is at far right. After grouping the two **1**s, we make the mistake of forming a group of 1-cell. The reason that this is not desirable is that:

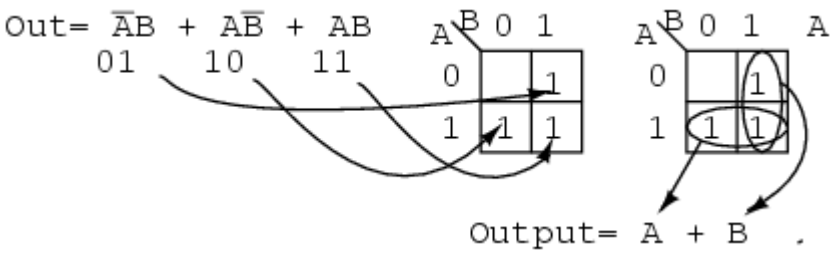
- The single cell has a product term of **AB'**
- The corresponding solution is **Output = AB' + B**
- This is not the simplest solution

The way to pick up this single **1** is to form a group of two with the **1** to the right of it as shown in the lower line of the middle K-map, even though this **1** has already been included in the column group (**B**). We are allowed to re-use cells in order to form larger groups. In fact, it is desirable because it leads to a simpler result.

We need to point out that either of the above solutions, Output or Wrong Output, are logically correct. Both circuits yield the same output. It is a matter of the former circuit being the lowest cost solution.

Example:

Fill in the Karnaugh map for the Boolean expression below, then write the Boolean expression for the result.



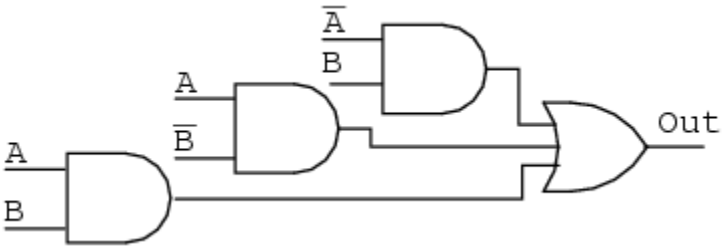
Solution: (above)

The Boolean expression has three product terms. There will be a **1** entered for each product term. Though, in general, the number of **1**s per product term varies with the number of variables in the product term compared to the size of the K-map. The product term is the address of the cell where the **1** is entered. The first product term, **A'B**, corresponds to the **01** cell in the map. A **1** is entered in this cell. The other two P-terms are entered for a total of three **1s**

Next, proceed with grouping and extracting the simplified result as in the previous truth table problem.

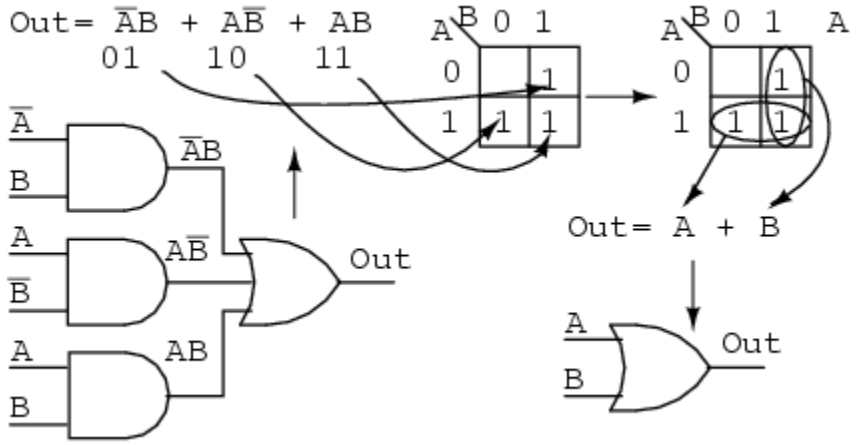
Example:

Simplify the logic diagram below.



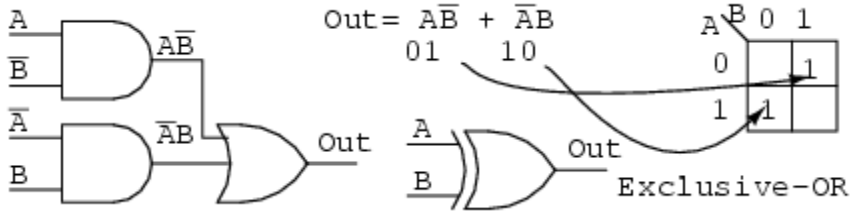
Solution: (Figure below)

- Write the Boolean expression for the original logic diagram as shown below
- Transfer the product terms to the Karnaugh map
- Form groups of cells as in previous examples
- Write Boolean expression for groups as in previous examples
- Draw simplified logic diagram



Example:

Simplify the logic diagram below.



Solution:

- Write the Boolean expression for the original logic diagram shown above
- Transfer the product terms to the Karnaugh map.
- It is not possible to form groups.
- No simplification is possible; leave it as it is.

No logic simplification is possible for the above diagram. This sometimes happens. Neither the methods of Karnaugh maps nor Boolean algebra can simplify this logic further. We show an Exclusive-OR schematic symbol above; however, this is not a logical simplification. It just makes a schematic diagram look nicer. Since it is not possible to simplify the Exclusive-OR logic and it is widely used, it is provided by manufacturers as a basic integrated circuit (7486).

Multiplexer and Demultiplexer

A multiplexer is a circuit that accept many input but give only one output. A demultiplexer function exactly in the reverse of a multiplexer, that is a demultiplexer accepts only one input and gives many outputs. Generally multiplexer and demultiplexer are used together, because of the communication systems are bi directional.

Mutliplexer:

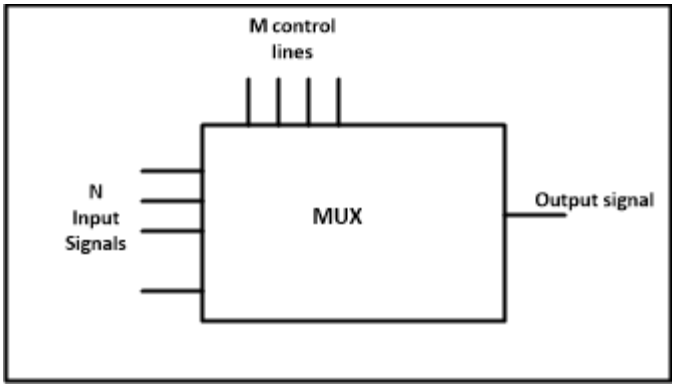
Multiplexer means many into one. A multiplexer is a circuit used to select and route any one of the several input signals to a signal output. An simple example of an non electronic circuit of a multiplexer is a single pole multiposition switch.

Multiposition switches are widely used in many electronics circuits. However circuits that operate at high speed require the multiplexer to be automatically selected. A mechanical switch cannot perform this task satisfactorily. Therefore, multiplexer used to perform high speed switching are constructed of electronic components.

Multiplexer handle two type of data that is analog and digital. For analog application, multiplexer are built of relays and transistor switches. For digital application, they are built from standard logic gates.

The multiplexer used for digital applications, also called digital multiplexer, is a circuit with many input but only one output. By applying control signals, we can steer any input to the output. Few types of multiplexer are 2-to-1, 4-to-1, 8-to-1, 16-to-1 multiplexer.

Following figure shows the general idea of a multiplexer with n input signal, m control signals and one output signal.



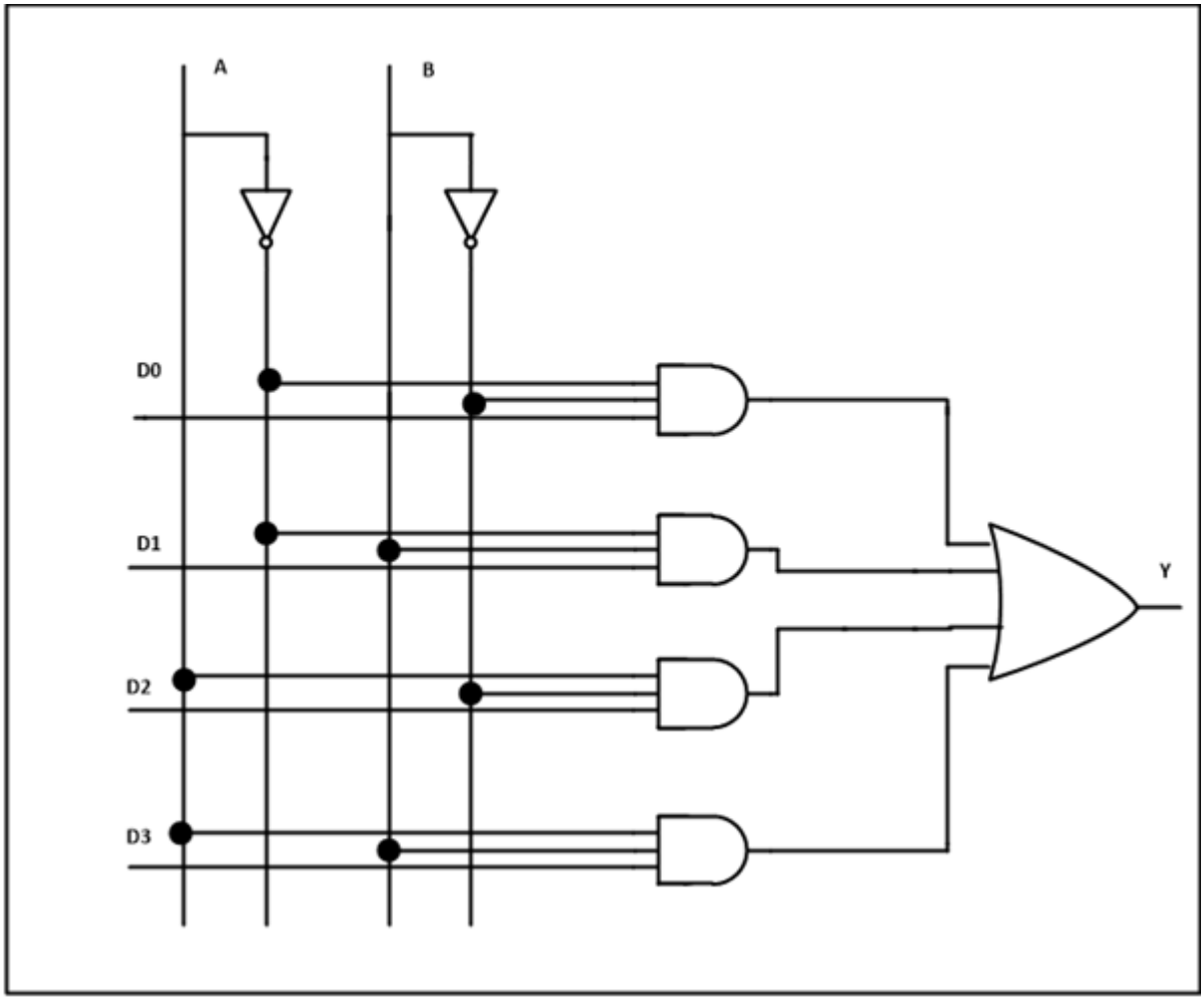
Multiplexer Pin Diagram

Understanding 4-to-1 Multiplexer:

The 4-to-1 multiplexer has 4 input bit, 2 control bits, and 1 output bit. The four input bits are D0,D1,D2 and D3. only one of this is transmitted to the output y. The output depends on the

value of AB which is the control input. The control input determines which of the input data bit is transmitted to the output.

For instance, as shown in fig. when AB = 00, the upper AND gate is enabled while all other AND gates are disabled. Therefore, data bit D0 is transmitted to the output, giving Y = Do.



4 to 1 Multiplexer Circuit Diagram – ElectronicsHub.Org

If the control input is changed to AB =11, all gates are disabled except the bottom AND gate. In this case, D3 is transmitted to the output and Y = D3.

- An example of 4-to-1 multiplexer is IC 74153 in which the output is same as the input.
- Another example of 4-to-1 multiplexer is 45352 in which the output is the compliment of the input.
- Example of 16-to-1 line multiplexer is IC74150.

Applications of Multiplexer:

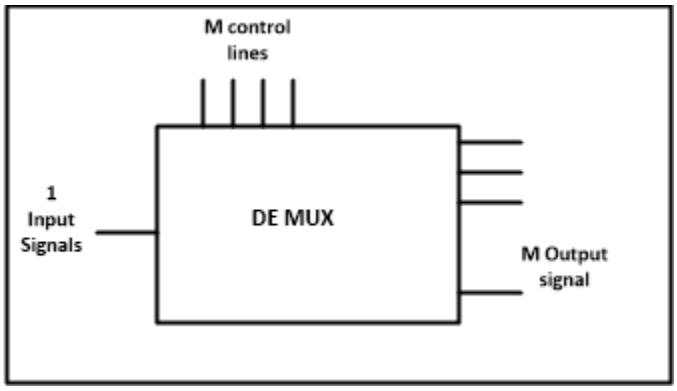
Multiplexer are used in various fields where multiple data need to be transmitted using a single line. Following are some of the applications of multiplexers –

1. **Communication system** – Communication system is a set of system that enable communication like transmission system, relay and tributary station, and communication network. The efficiency of communication system can be increased considerably using multiplexer. Multiplexer allow the process of transmitting different type of data such as audio, video at the same time using a single transmission line.
2. **Telephone network –** In telephone network, multiple audio signals are integrated on a single line for transmission with the help of multiplexers. In this way, multiple audio signals can be isolated and eventually, the desire audio signals reach the intended recipients.
3. **Computer memory** – Multiplexers are used to implement huge amount of memory into the computer, at the same time reduces the number of copper lines required to connect the memory to other parts of the computer circuit.
4. **Transmission from the computer system of a satellite** – Multiplexer can be used for the transmission of data signals from the computer system of a satellite or spacecraft to the ground system using the GPS (Global Positioning System) satellites.

Demultiplexer:

Demultiplexer means one to many. A demultiplexer is a circuit with one input and many output. By applying control signal, we can steer any input to the output. Few types of demultiplexer are 1-to 2, 1-to-4, 1-to-8 and 1-to 16 demultiplexer.

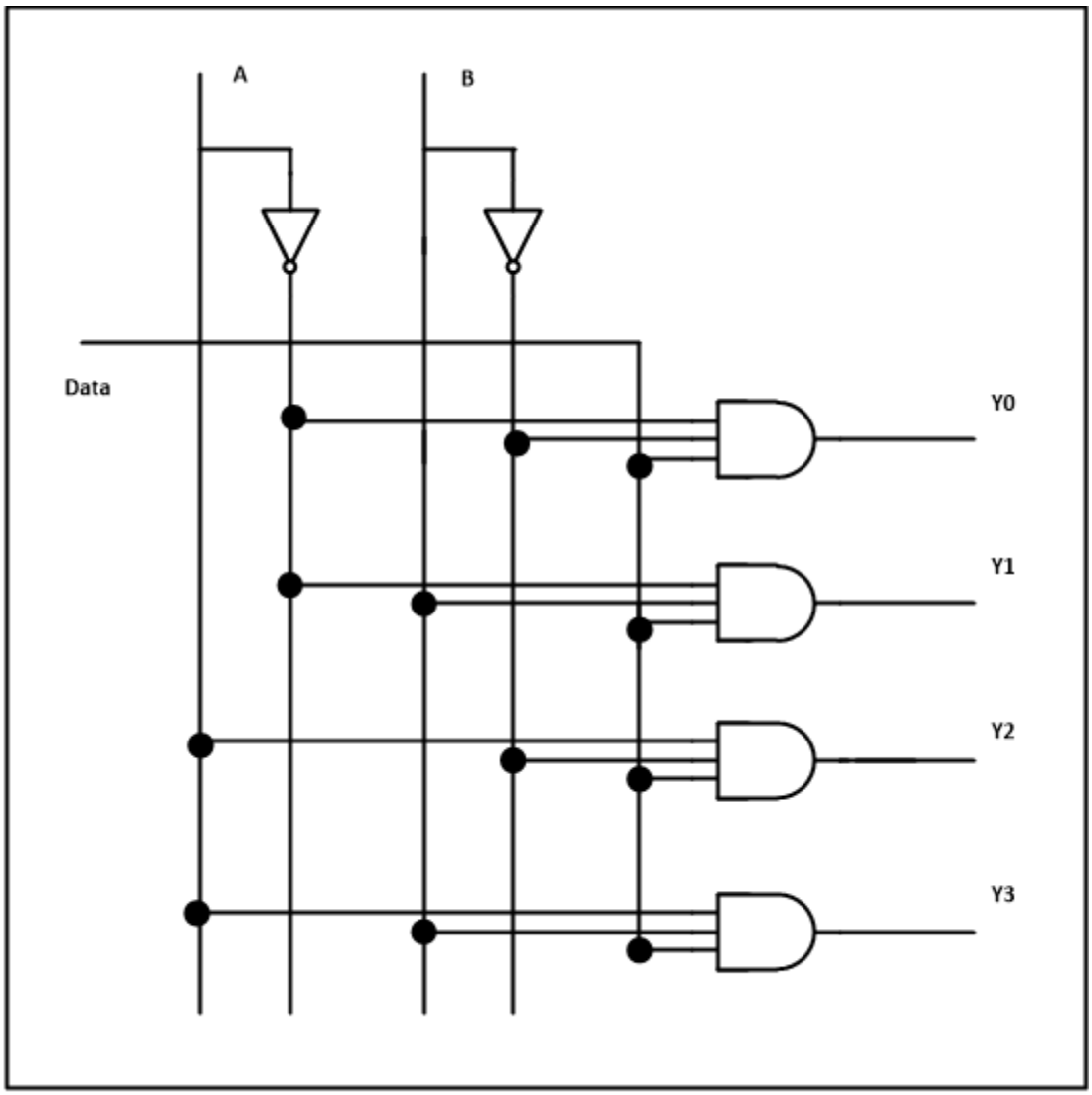
Following figure illustrate the general idea of a demultiplexer with 1 input signal, m control signals, and n output signals.



Demultiplexer Pin Diagram

Understanding 1- to-4 Demultiplexer:

The 1-to-4 demultiplexer has 1 input bit, 2 control bit, and 4 output bits. An example of 1-to-4 demultiplexer is IC 74155. The 1-to-4 demultiplexer is shown in figure below-



1 to 4 Dempultiplexer Circuit Diagram – ElectronicsHub.Org

The input bit is labelled as Data D. This data bit is transmitted to the data bit of the output lines. This depends on the value of AB, the control input.

When AB = 01, the upper second AND gate is enabled while other AND gates are disabled. Therefore, only data bit D is transmitted to the output, giving Y1 = Data.

If D is low, Y1 is low. IF D is high,Y1 is high. The value of Y1 depends upon the value of D. All other outputs are in low state.

If the control input is changed to AB = 10, all the gates are disabled except the third AND gate from the top. Then, D is transmitted only to the Y2 output, and Y2 = Data.

Example of 1-to-16 demultiplexer is IC 74154 it has 1 input bit, 4 control bits and 16 output bit.

Applications of Demultiplexer:

1. Demultiplexer is used to connect a single source to multiple destinations. The main application area of demultiplexer is communication system where multiplexer are used. Most of the communication system are bidirectional i.e. they function in both ways (transmitting and receiving signals). Hence, for most of the applications, the multiplexer and demultiplexer work in sync. Demultiplexer are also used for reconstruction of parallel data and ALU circuits.
2. **Communication System** – Communication system use multiplexer to carry multiple data like audio, video and other form of data using a single line for transmission. This process make the transmission easier. The demultiplexer receive the output signals of the multiplexer and converts them back to the original form of the data at the receiving end. The multiplexer and demultiplexer work together to carry out the process of transmission and reception of data in communication system.
3. **ALU (Arithmetic Logic Unit)** – In an ALU circuit, the output of ALU can be stored in multiple registers or storage units with the help of demultiplexer. The output of ALU is fed as the data input to the demultiplexer. Each output of demultiplexer is connected to multiple register which can be stored in the registers.
4. **Serial to parallel converter** – A serial to parallel converter is used for reconstructing parallel data from incoming serial data stream. In this technique, serial data from the incoming serial data stream is given as data input to the demultiplexer at the regular intervals. A counter is attach to the control input of the demultiplexer. This counter directs the data signal to the output of the demultiplexer where these data signals are stored. When all data signals have been stored, the output of the demultiplexer can be retrieved and read out in parallel.

code converters, adders, subtractors

The logical circuit which converts binary code to equivalent gray code is known as **binary to gray code converter**. The gray code is a non weighted code. The successive gray code differs in one bit position only that means it is a unit distance code. It is also referred as cyclic code. It is not suitable for arithmetic operations. It is the most popular of the unit distance codes. It is also a reflective code. An n-bit Gray code can be obtained by reflecting an n-1 bit code about an axis after 2^{n-1} rows, and putting the MSB of 0 above the axis and the MSB of 1 below the axis. Reflection of Gray codes is shown below.

The 4 bits binary to gray code conversion table is given below,

Decimal Number	4 bit Binary Number ABCD	4 bit Gray Code $G_1G_2G_3G_4$
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1
10	1 0 1 0	1 1 1 1
11	1 0 1 1	1 1 1 0
12	1 1 0 0	1 0 1 0
13	1 1 0 1	1 0 1 1
14	1 1 1 0	1 0 0 1
15	1 1 1 1	1 0 0 0

That means, in 4 bit gray code, (4-1) or 3 bit code is reflected against the axis drawn after $(2^{4-1})^{th}$ or 8^{th} row.

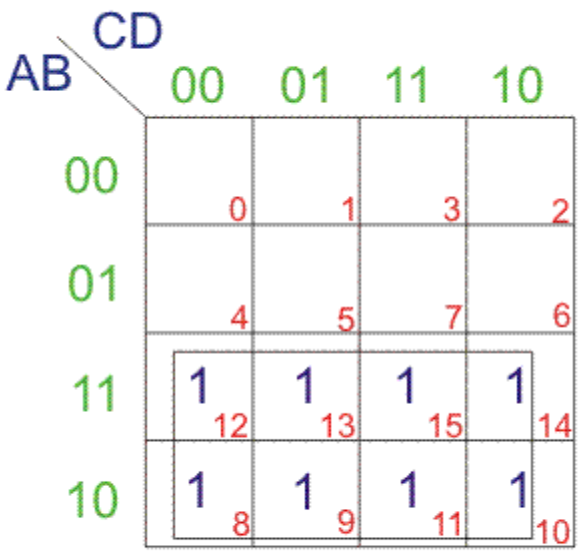
The bits of 4 bit gray code are considered as $G_4G_3G_2G_1$. Now from conversion table,

$$G_4 = \sum m(8,9,10,11,12,13,14,15),\ G_3 = \sum m(4,5,6,7,8,9,10,11)$$

$$G_2 = \sum m(2,3,4,5,10,11,12,13),\ G_1 = \sum m(1,2,5,6,9,10,13,14)$$

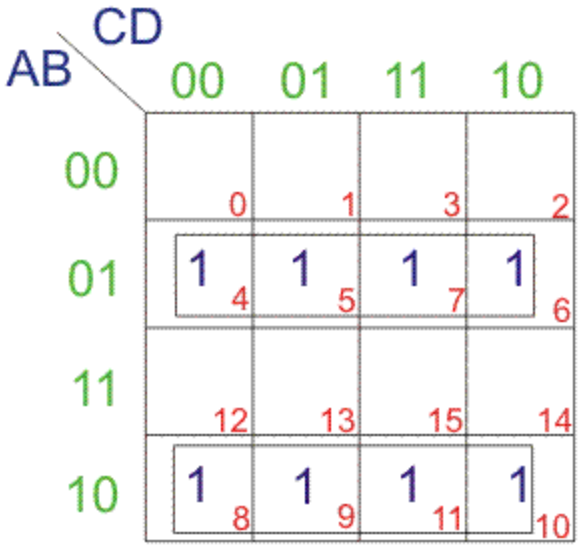
From above SOPs, let us draw K -maps for G_4, G_3, G_2 and G_1.

G_4



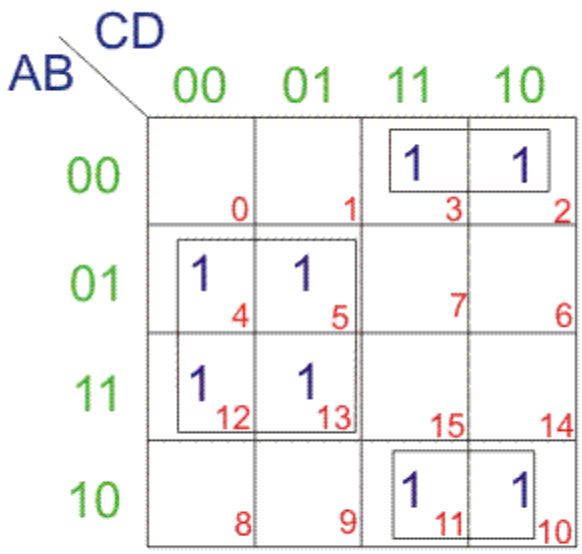
$G_4 = A$

G_3



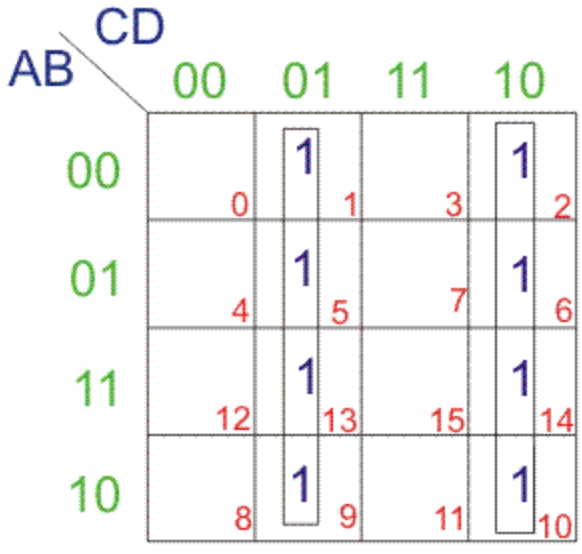
$G_3 = \overline{A}B + A\overline{B} = A \oplus B$

G_2

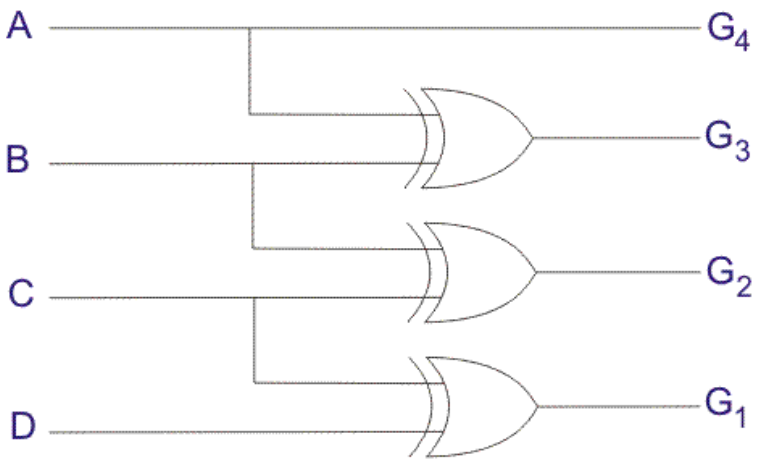


$$G_2 = B\overline{C} + \overline{B}C = B \oplus C$$

G_1



$$G_1 = \overline{C}D + C\overline{D} = C \oplus D$$



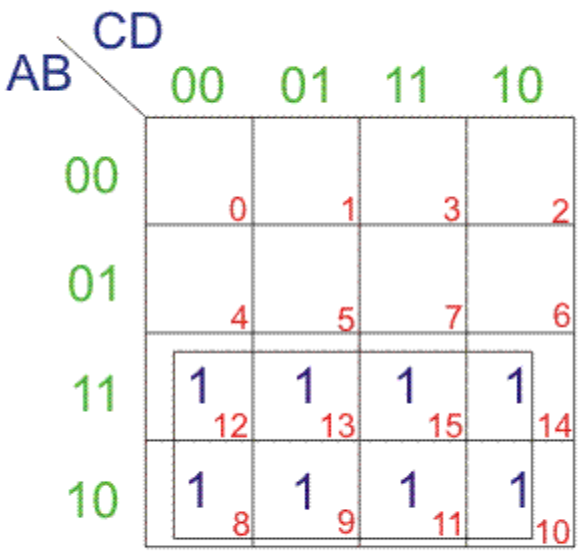
Logic Circuit for Binary to Gray Code Converter

Grey to Binary Code Converter

In **gray to binary code converter**, input is a multiplies gray code and output is its equivalent binary code.
Let us consider a 4 bit gray to binary code converter. To design a 4 bit gray to binary code converter, we first have to draw a conversion table.

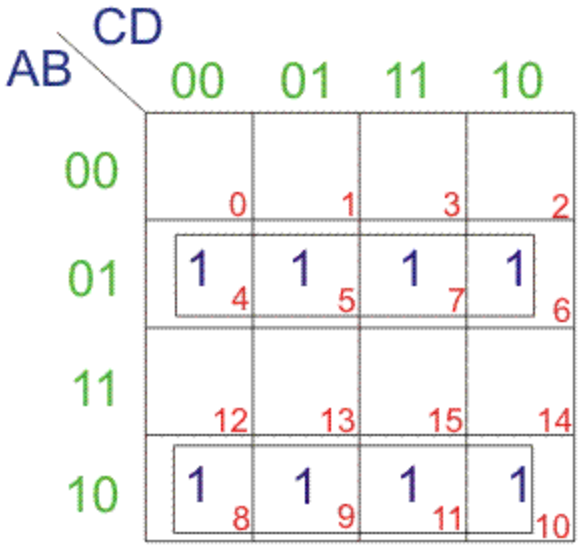
4 bit Gray Code	4 bit Binary Code
A B C D	B_4 B_3 B_2 B_1
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 1	0 0 1 0
0 0 1 0	0 0 1 1
0 1 1 0	0 1 0 0
0 1 1 1	0 1 0 1
0 1 0 1	0 1 1 0
0 1 0 0	0 1 1 1
1 1 0 0	1 0 0 0
1 1 0 1	1 0 0 1
1 1 1 1	1 0 1 0
1 1 1 0	1 0 1 1
1 0 1 0	1 1 0 0
1 0 1 1	1 1 0 1
1 0 0 1	1 1 1 0
1 0 0 0	1 1 1 1

B_4

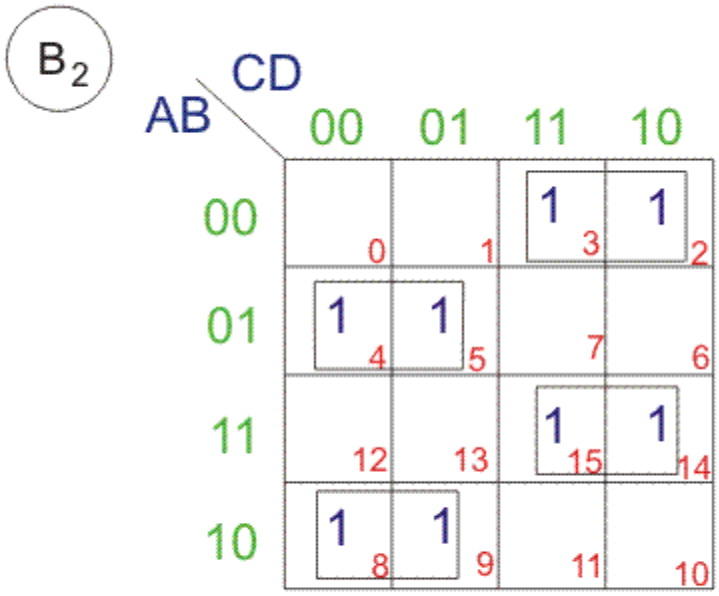


$B_4 = A$

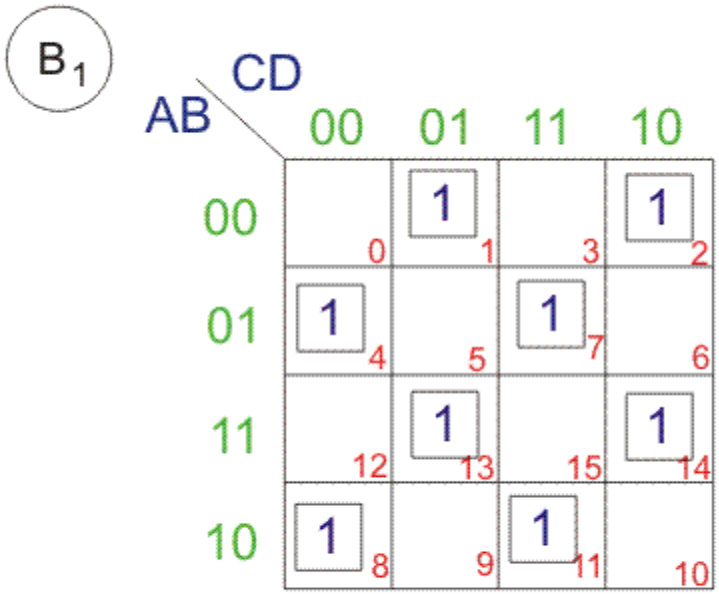
B_3



$B_3 = \overline{A}B + A\overline{B} = A \oplus B$

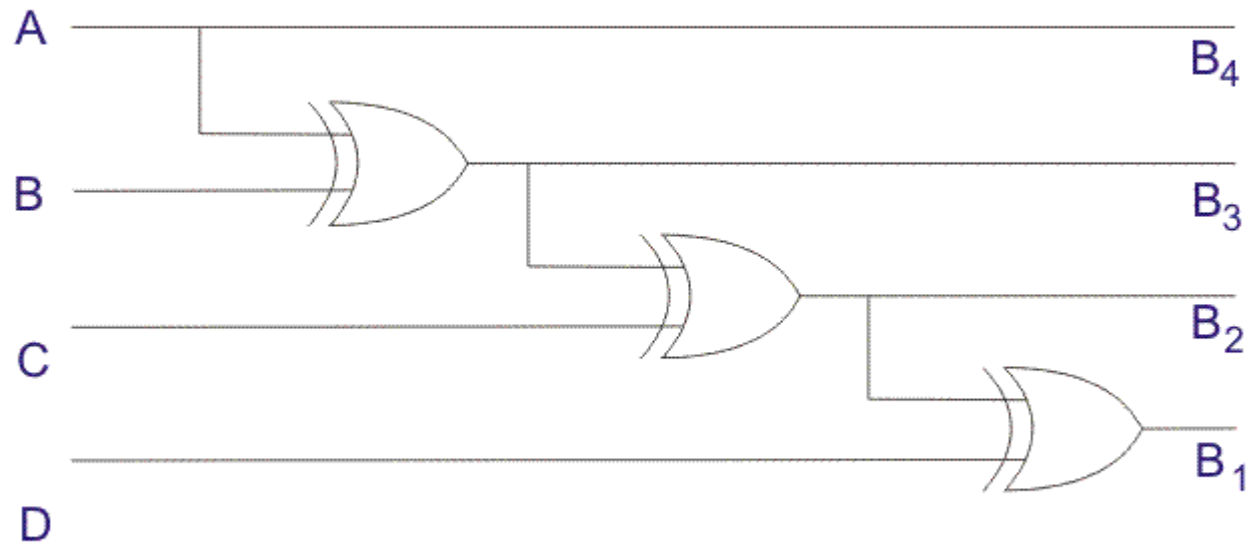


$$
\begin{aligned}
B_2 &= \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + ABC \\
&= A(\overline{B}\,\overline{C} + BC) + \overline{A}(B\overline{C} + \overline{B}C) \\
&= A\overline{(B\overline{C} + \overline{B}C)} + \overline{A}(B\overline{C} + \overline{B}C) \\
&= A(\overline{B \oplus C}) + \overline{A}(\overline{B \oplus C}) = A \oplus B \oplus C
\end{aligned}
$$



$$B_1 = \overline{A}\,\overline{B}\,\overline{C}\,D + \overline{A}\,\overline{B}\,C\,\overline{D} + \overline{A}\,B\,\overline{C}\,\overline{D} + \overline{A}\,BCD + AB\overline{C}\,D + ABC\overline{D} + A\overline{B}\,\overline{C}\,\overline{D}$$
$$+ A\overline{B}CD = A \oplus B \oplus C \oplus D$$

From above gray code we get,



Logic Circuit for Gray to Binary Code Converter

Combinational Logic Circuits

A combinational logic circuit implement logical functions where its outputs depend only on its current combination of input values. On the other hand sequential circuits, unlike combinational logic, have state or memory.

The main difference between sequential circuits and combinational circuits is that sequential circuits compute their output based on input and state, and that the state is updated based on a clock. Combinational logic circuits implement Boolean functions and are functions only of their inputs.

Representing Combinational Logic Functions

There are 3 ways to represent combinational logic functions

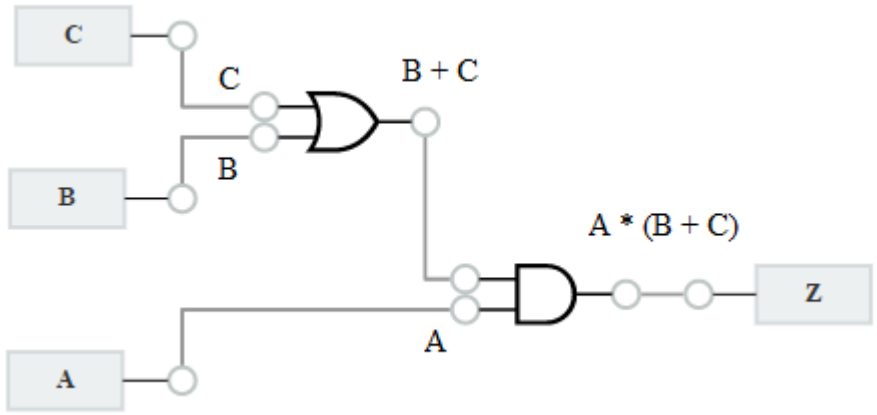
1. **Logic gates** - Logic gates are used as the building blocks in the design of combinational logic circuits. These gates are the AND, OR, NOT, NAND, NOR gates.
2. **Boolean Algebra** - Boolean Algebra specifies the relationship between Boolean variables which is used to design digital circuits using Logic Gates. Every logic circuit can be completely described using the Boolean operations, because the OR, AND gate, and NOT gates are the basic building blocks of digital systems.
3. **Truth table** - A truth table is used in logic to compute the functional values of logical expressions on each combination of values taken by their logical variables. If a combination logic block have more than one bit output, each single-bit output gets its own truth-table. Often they are combined into a single table with multiple output columns, one for each single-bit output.

Combinational Logic Circuit Analysis

To obtain the boolean expressions and truth tables from the combinational logic circuit, we need to analyse the circuit. First ensure that the circuit is combinational - that is there is no feedback of an output to an input that the output depends on.

Boolean Expressions

1. label all inputs -- input variables
2. label all outputs -- output functions
3. label all intermediate signals (outputs that feed inputs)



For each output functions, write it in terms of its input variables and intermediate signals, and then expand intermediate signals until the outputs are expressed only in terms of the inputs.

Truth tables

The truth table can be derived from the Boolean expressions, or by directly working out from the circuit, the outputs for each possible combination of inputs.

If there are n input variables

- there are 2^n possible binary input combinations
- there are 2^n entries in the truth table for each output

From the examples below, change the inputs to observe the outputs

UNIT III

SYNCHRONOUS SEQUENTIAL CIRCUITS

Sequential Logic Circuits

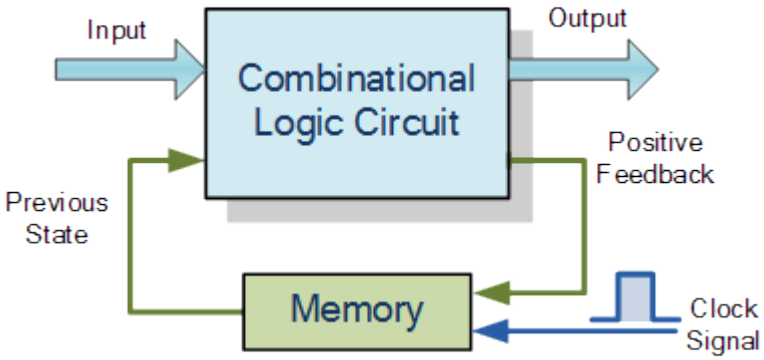
Unlike Combinational Logic circuits that change state depending upon the actual signals being applied to their inputs at that time, **Sequential Logic** circuits have some form of inherent "Memory" built in

This means that sequential logic circuits are able to take into account their previous input state as well as those actually present, a sort of "before" and "after" effect is involved with sequential circuits.

In other words, the output state of a "sequential logic circuit" is a function of the following three states, the "present input", the "past input" and/or the "past output". *Sequential Logic circuits* remember these conditions and stay fixed in their current state until the next clock signal changes one of the states, giving sequential logic circuits "Memory".

Sequential logic circuits are generally termed as *two state* or **Bistable** devices which can have their output or outputs set in one of two basic states, a logic level "1" or a logic level "0" and will remain "latched" (hence the name latch) indefinitely in this current state or condition until some other input trigger pulse or signal is applied which will cause the bistable to change its state once again.

Sequential Logic Representation



The word "Sequential" means that things happen in a "sequence", one after another and in **Sequential Logic** circuits, the actual clock signal determines when things will happen next.

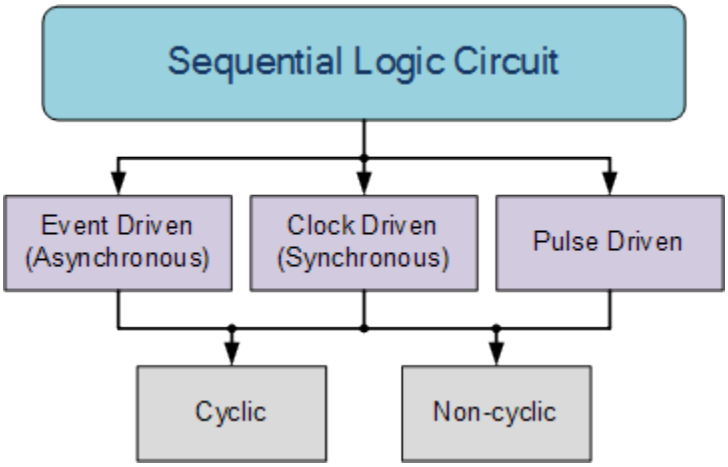
Simple sequential logic circuits can be constructed from standard **Bistable**circuits such as: Flip-flops, Latches and Counters and which themselves can be made by simply connecting together universal **NAND Gates** and/or **NOR Gates** in a particular combinational way to produce the required sequential circuit.

Related Products: **Direct Digital Synthesizer**

Classification of Sequential Logic

As standard logic gates are the building blocks of combinational circuits, bistable latches and flip-flops are the basic building blocks of sequential logic circuits. Sequential logic circuits can be constructed to produce either simple edge-triggered flip-flops or more complex sequential circuits such as storage registers, shift registers, memory devices or counters. Either way sequential logic circuits can be divided into the following three main categories:

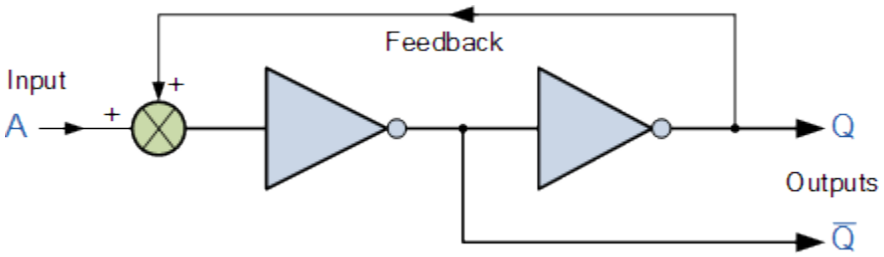
- 1. Event Driven – asynchronous circuits that change state immediately when enabled.
- 2. Clock Driven – synchronous circuits that are synchronised to a specific clock signal.
- 3. Pulse Driven – which is a combination of the two that responds to triggering pulses.



As well as the two logic states mentioned above logic level "1" and logic level "0", a third element is introduced that separates **sequential logic** circuits from their **combinational logic** counterparts, namely *TIME*. Sequential logic circuits return back to their original steady state once reset and sequential circuits with loops or feedback paths are said to be "cyclic" in nature.

We now know that in sequential circuits changes occur only on the application of a clock signal making it synchronous, otherwise the circuit is asynchronous and depends upon an external input. To retain their current state, sequential circuits rely on feedback and this occurs when a fraction of the output is fed back to the input and this is demonstrated as:

Sequential Feedback Loop



The two inverters or NOT gates are connected in series with the output at Q fed back to the input. Unfortunately, this configuration never changes state because the output will always be the same, either a "1" or a "0", it is permanently set. However, we can see how feedback works by examining the most basic sequential logic components, called the SR flip-flop.

SR Flip-Flop

The **SR flip-flop**, also known as a *SR Latch*, can be considered as one of the most basic sequential logic circuit possible. This simple flip-flop is basically a one-bit memory bistable device that has two inputs, one which will "SET" the device (meaning the output = "1"), and is labelled S and another which will "RESET" the device (meaning the output = "0"), labelled R.

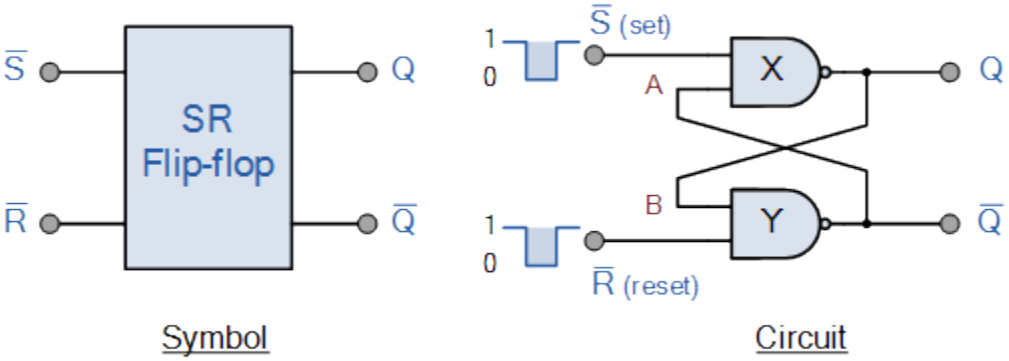
Then the SR description stands for "Set-Reset". The reset input resets the flip-flop back to its original state with an output Q that will be either at a logic level "1" or logic "0" depending upon this set/reset condition.

A basic NAND gate SR flip-flop circuit provides feedback from both of its outputs back to its opposing inputs and is commonly used in memory circuits to store a single data bit. Then the SR flip-flop actually has three inputs, Set, Reset and its current output Qrelating to it's current state or history. The term "Flip-flop" relates to the actual operation of the device, as it can be "flipped" into one logic Set state or "flopped" back into the opposing logic Reset state.

The NAND Gate SR Flip-Flop

The simplest way to make any basic single bit set-reset SR flip-flop is to connect together a pair of cross-coupled 2-input NAND gates as shown, to form a Set-Reset Bistable also known as an active LOW SR NAND Gate Latch, so that there is feedback from each output to one of the other NAND gate inputs. This device consists of two inputs, one called the *Set*, S and the other called the *Reset*, R with two corresponding outputs Q and its inverse or complement Q (not-Q) as shown below.

The Basic SR Flip-flop



The Set State

Consider the circuit shown above. If the input R is at logic level "0" (R = 0) and input S is at logic level "1" (S = 1), the NAND gate *Y* has at least one of its inputs at logic "0" therefore, its output Q must be at a logic level "1" (NAND Gate principles). Output Q is also fed back to input "A" and so both inputs to NAND gate *X* are at logic level "1", and therefore its output Q must be at logic level "0".

Again NAND gate principals. If the reset input R changes state, and goes HIGH to logic "1" with S remaining HIGH also at logic level "1", NAND gate *Y* inputs are now R = "1" and B = "0". Since one of its inputs is still at logic level "0" the output at Q still remains HIGH at logic level "1" and there is no change of state. Therefore, the flip-flop circuit is said to be "Latched" or "Set" with Q = "1" and Q = "0".

Reset State

In this second stable state, Q is at logic level "0", (not Q = "0") its inverse output at Q is at logic level "1", (Q = "1"), and is given by R = "1" and S = "0". As gate *X* has one of its inputs at logic "0" its output Q must equal logic level "1" (again NAND gate principles). Output Q is fed back to input "B", so both inputs to NAND gate *Y* are at logic "1", therefore, Q = "0".

If the set input, S now changes state to logic "1" with input R remaining at logic "1", output Q still remains LOW at logic level "0" and there is no change of state. Therefore, the flip-flop circuits "Reset" state has also been latched and we can define this "set/reset" action in the following truth table.

Truth Table for this Set-Reset Function

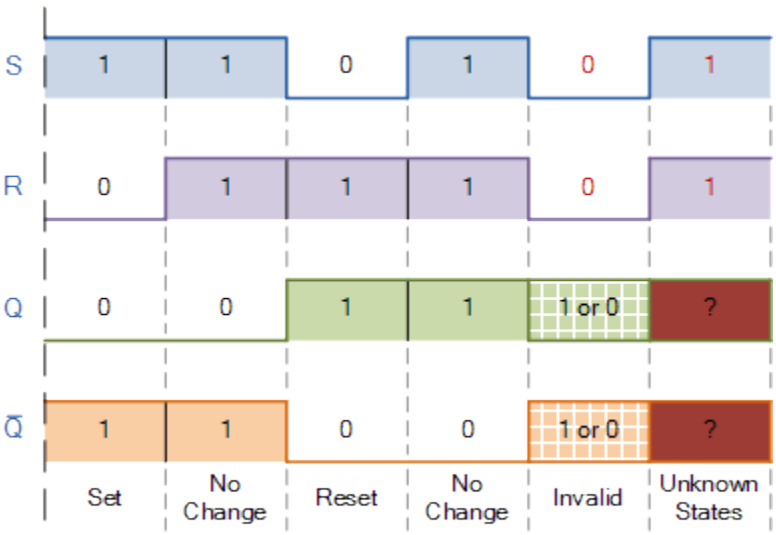
State	S	R	Q	Q	Description

Set	1	0	0	1	Set Q » 1
	1	1	0	1	no change
Reset	0	1	1	0	Reset Q » 0
	1	1	1	0	no change
Invalid	0	0	1	1	Invalid Condition

It can be seen that when both inputs S = “1” and R = “1” the outputs Q and Q can be at either logic level “1” or “0”, depending upon the state of the inputs S or R BEFORE this input condition existed. Therefore the condition of S = R = “1” does not change the state of the outputs Q and Q.

However, the input state of S = “0” and R = “0” is an undesirable or invalid condition and must be avoided. The condition of S = R = “0” causes both outputs Q and Q to be HIGH together at logic level “1” when we would normally want Q to be the inverse of Q. The result is that the flip-flop looses control of Q and Q, and if the two inputs are now switched “HIGH” again after this condition to logic “1”, the flip-flop becomes unstable and switches to an unknown data state based upon the unbalance as shown in the following switching diagram.

S-R Flip-flop Switching Diagram

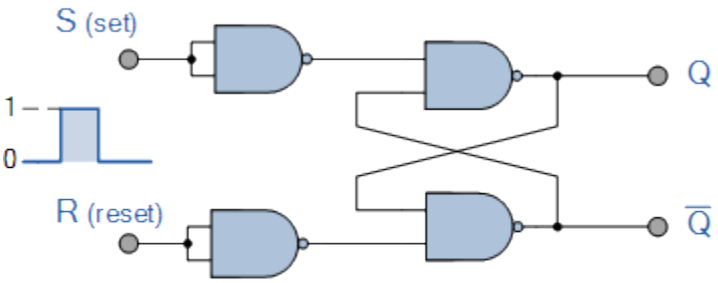


This unbalance can cause one of the outputs to switch faster than the other resulting in the flip-flop switching to one state or the other which may not be the required state and data corruption will exist. This unstable condition is generally known as its **Meta-stable**state.

Then, a simple NAND gate SR flip-flop or NAND gate SR latch can be set by applying a logic "0", (LOW) condition to its Set input and reset again by then applying a logic "0" to its Reset input. The SR flip-flop is said to be in an "invalid" condition (Meta-stable) if both the set and reset inputs are activated simultaneously.

As we have seen above, the basic NAND gate SR flip-flop requires logic "0" inputs to flip or change state from Q to Q and vice versa. We can however, change this basic flip-flop circuit to one that changes state by the application of positive going input signals with the addition of two extra NAND gates connected as inverters to the S and R inputs as shown.

Positive NAND Gate SR Flip-flop

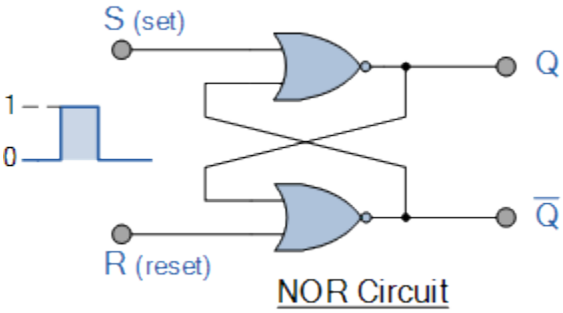


NAND Circuit

S	R	Q	Q̄
0	0	No change	
0	1	0	1
1	0	1	0
1	1	X	X
		(Invalid)	

As well as using NAND gates, it is also possible to construct simple one-bit **SR Flip-flops** using two cross-coupled NOR gates connected in the same configuration. The circuit will work in a similar way to the NAND gate circuit above, except that the inputs are active HIGH and the invalid condition exists when both its inputs are at logic level "1", and this is shown below.

The NOR Gate SR Flip-flop



NOR Circuit

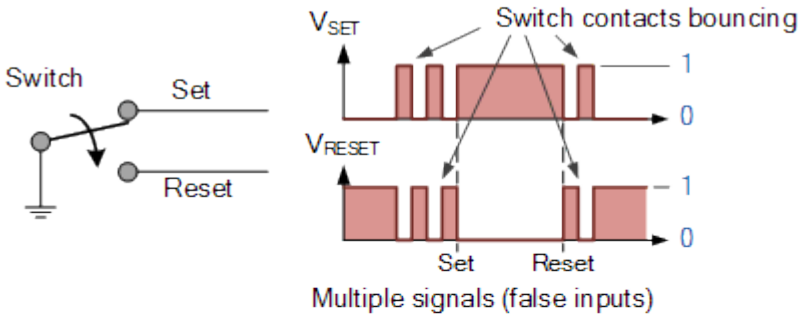
S	R	Q	$\overline{Q}$
0	0	No change	
0	1	1	0
1	0	0	1
1	1	X	X
		(Invalid)	

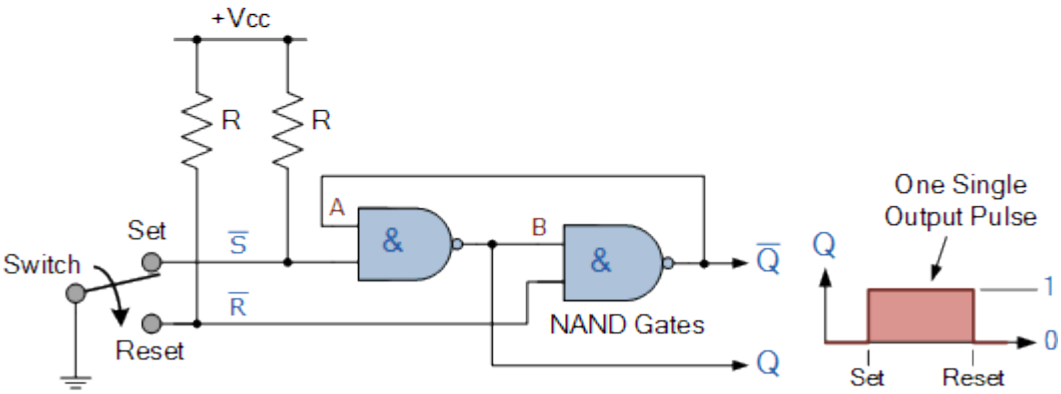
Switch Debounce Circuits

Edge-triggered flip-flops require a nice clean signal transition, and one practical use of this type of set-reset circuit is as a latch used to help eliminate mechanical switch "bounce". As its name implies, switch bounce occurs when the contacts of any mechanically operated switch, push-button or keypad are operated and the internal switch contacts do not fully close cleanly, but bounce together first before closing (or opening) when the switch is pressed.

This gives rise to a series of individual pulses which can be as long as tens of milliseconds that an electronic system or circuit such as a digital counter may see as a series of logic pulses instead of one long single pulse and behave incorrectly. For example, during this bounce period the output voltage can fluctuate wildly and may register multiple input counts instead of one single count. Then set-reset SR Flip-flops or Bistable Latch circuits can be used to eliminate this kind of problem and this is demonstrated below.

SR Flip Flop Switch Debounce Circuit





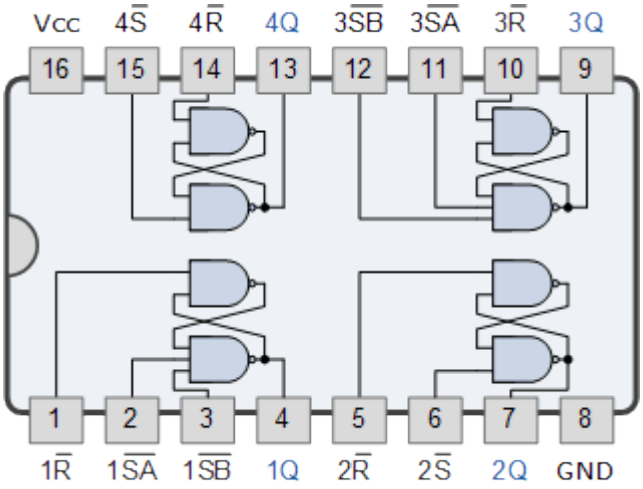
Depending upon the current state of the output, if the set or reset buttons are depressed the output will change over in the manner described above and any additional unwanted inputs (bounces) from the mechanical action of the switch will have no effect on the output at Q.

When the other button is pressed, the very first contact will cause the latch to change state, but any additional mechanical switch bounces will also have no effect. The SR flip-flop can then be RESET automatically after a short period of time, for example 0.5 seconds, so as to register any additional and intentional repeat inputs from the same switch contacts, such as multiple inputs from a keyboards "RETURN" key.

Commonly available IC's specifically made to overcome the problem of switch bounce are the MAX6816, single input, MAX6817, dual input and the MAX6818 octal input switch debouncer IC's. These chips contain the necessary flip-flop circuitry to provide clean interfacing of mechanical switches to digital systems.

Set-Reset bistable latches can also be used as Monostable (one-shot) pulse generators to generate a single output pulse, either high or low, of some specified width or time period for timing or control purposes. The 74LS279 is a Quad SR Bistable Latch IC, which contains four individual NAND type bistable's within a single chip enabling switch debounce or monostable/astable clock circuits to be easily constructed.

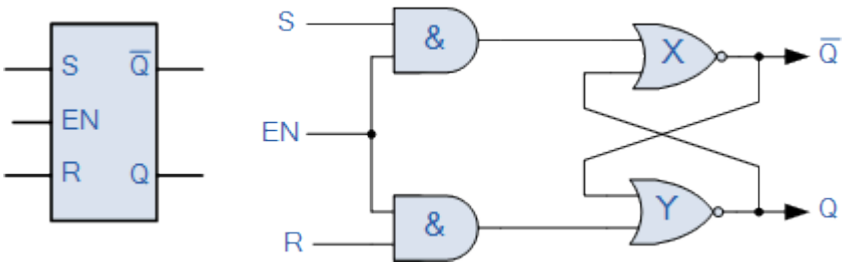
Quad SR Bistable Latch 74LS279



Gated or Clocked SR Flip-Flop

It is sometimes desirable in sequential logic circuits to have a bistable SR flip-flop that only changes state when certain conditions are met regardless of the condition of either the Set or the Reset inputs. By connecting a 2-input AND gate in series with each input terminal of the SR Flip-flop a Gated SR Flip-flop can be created. This extra conditional input is called an "Enable" input and is given the prefix of "EN". The addition of this input means that the output at Q only changes state when it is HIGH and can therefore be used as a clock (CLK) input making it level-sensitive as shown below.

Gated SR Flip-flop



When the Enable input "EN" is at logic level "0", the outputs of the two AND gates are also at logic level "0", (AND Gate principles) regardless of the condition of the two inputs S and R, latching the two outputs Q and Q into their last known state. When the enable input "EN" changes to logic level "1" the circuit responds as a normal SR bistable flip-flop with the two AND gates becoming transparent to the Set and Reset signals.

This additional enable input can also be connected to a clock timing signal (CLK) adding clock synchronisation to the flip-flop creating what is sometimes called a "Clocked SR Flip-flop". So a **Gated Bistable SR Flip-flop** operates as a standard bistable latch but the outputs are only activated when a logic "1" is applied to its EN input and deactivated by a logic "0".

In the next tutorial about **Sequential Logic Circuits**, we will look at another type of simple edge-triggered flip-flop which is very similar to the **RS flip-flop** called a **JK Flip-flop** named

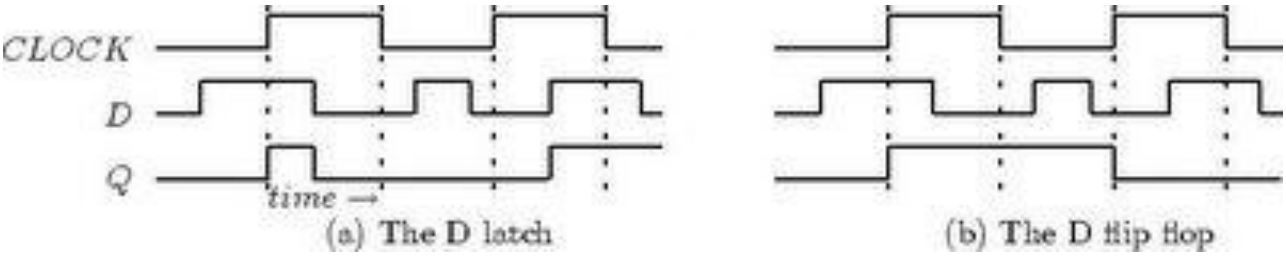
after its inventor, Jack Kilby. The JK flip-flop is the most widely used of all the flip-flop designs as it is considered to be a universal device.

Difference between Level Triggered and Edge Triggered

Level Trigger:
1) The input signal is sampled when the clock signal is either HIGH or LOW.
2) It is sensitive to Glitches.
Example: Latch.

Edge Trigger:
1) The input signal is sampled at the RISING EDGE or FALLING EDGE of the clock signal.
2) It is not-sensitive to Glitches.
Example: Flipflop.

I m sure the timing diagrams below is the best way of explanation.



Synchronous Counter

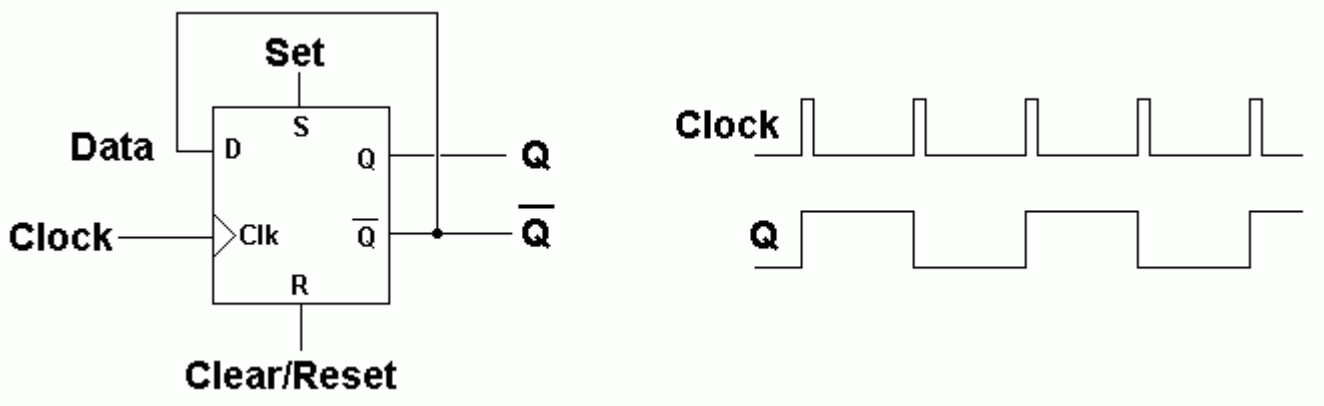
High-frequency operations require that all the FFs of a counter be triggered at the same time toprevent errors. We use a SYNCHRONOUS counter for this type of operation.The synchronous counter is similar to a ripple counter with two exceptions: The clock pulses areapplied to each FF, and additional gates are added to ensure that the FFs toggle in the proper sequence.A logic diagram of a three-state (modulo-8) synchronous counter is shown in figure 3-24, view A.The clock input is wired to each of the FFs to prevent possible errors in the count. A HIGH is wired to theJ and K inputs of FF1 to make the FF toggle. The output of FF1 is wired to the J and K inputs of FF2, oneinput of the AND gate, and indicator A. The output of FF2 is wired to the

other input of the AND gateand indicator B. The AND output is connected to the J and K inputs of FF3. The C indicator is the onlyoutput of FF3.
During the explanation of this circuit, you should follow the logic diagram, view A, and the pulsesequences, view B.Assume the following initial conditions: The outputs of all FFs, the clock, and the AND gate are 0;the J and K inputs to FF1 are HIGH. The negative-going portion of the clock pulse will be usedthroughout the explanation.Clock pulse 1 causes FF1 to set. This HIGH lights lamp A, indicating a binary count of 001. TheHIGH is also applied to the J and K inputs of FF2 and one input of the AND gate. Notice that FF2 and

Modulo N Counter

Frequency Divider
On the rising edge of the clock pulse, D is copied to Q.
Since NOT Q is connected to D, the data is inverted on each rising edge.
This has the effect of dividing the frequency by two.

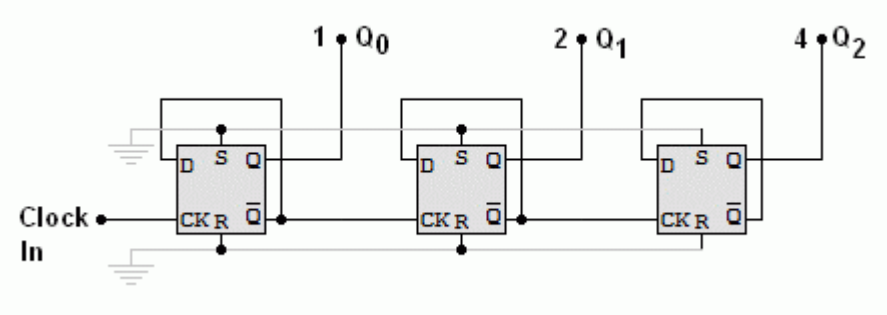
- The D Type Flip Flop is used in Binary Counters.
- Here is the circuit for a one bit counter.
- This can also be used as a frequency divider. It divides the frequency by two.
- The UP in Up Counter is because the counter counts normally with increasing numbers like 0, 1, 2, 3 etc.
- The output of this circuit is high for 50% of the time and low for 50% of the time.
- This is a 1:1 mark space ratio.
- This is true whatever the mark space ratio of the clock pulses.



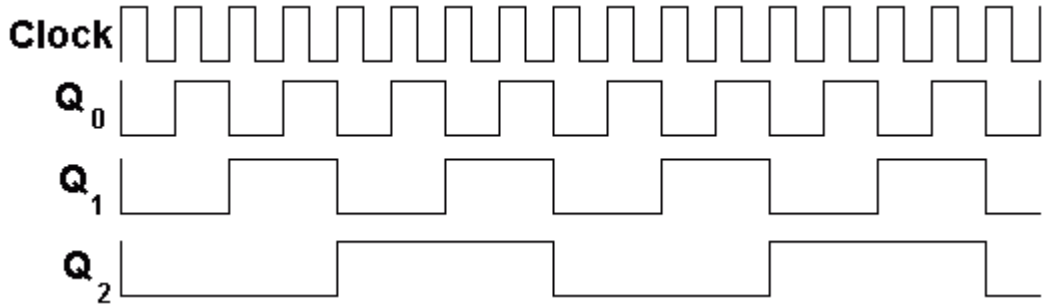
AS A2

b

Three Bit Binary Up Counter
This divides the frequency by 8 (or by two three times).
On the rising edge of the clock pulse the counter output increases by one.



The output from the left flip-flop is worth one (least significant bit LSB).
The output from the middle flip-flop is worth two.
The output from the right flip-flop is worth 4 (most significant bit MSB).
Here is a timing diagram for the three bit counter.



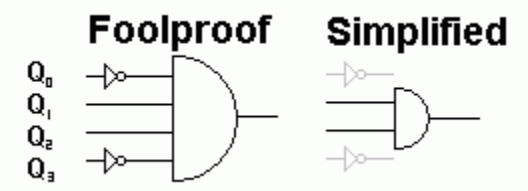
c

Modulo N Up Counter

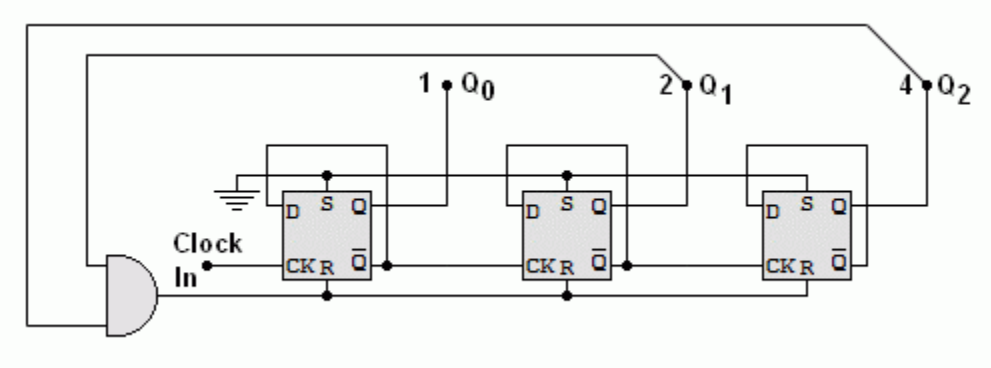
This is a counter that resets at a chosen number. For example a two digit decimal counter, left to its own devices will count from 00 to 99. This is not much use for a clock unless you have 100 second minutes. To fix the problem, the counter must go from 00 to 59. This is achieved by detecting a 6 in the left hand digit and using it to reset the counter to zero. This would be a Modulo 6 Counter or 60 if you included both digits.

d

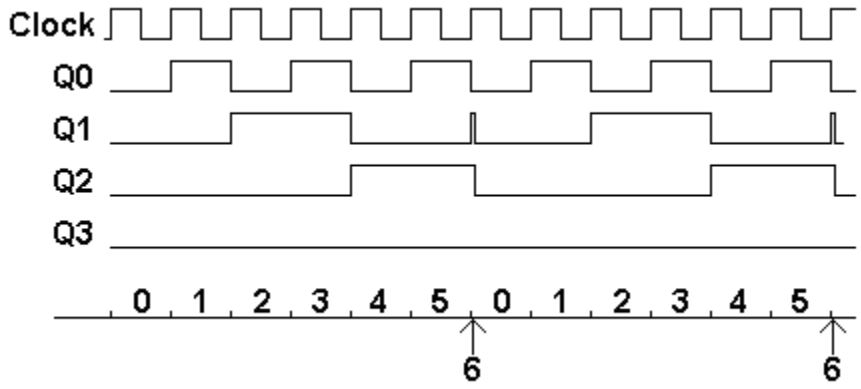
Modulo 6 Counter - Counts from 0 to 5



The circuit above detects a six or 0110 in binary. You could use the fool proof circuit but in fact the simpler circuit works too because the 0110 pattern only occurs once between 0 and 9 in decimal numbers. The output is used to reset the counter.



Here is a timing diagram for the modulo 6 counter. It shows the count going from 0 to 5 in regular time steps. The counter reaches 6 but only for about a microsecond before it resets to zero.



e

Modulo 10 Counter - Counts from 0 to 9

Introduction to Shift Registers

Shift registers, like counters, are a form of *sequential logic*. Sequential logic, unlike combinational logic is not only affected by the present inputs, but also, by the prior history. In other words, sequential logic remembers past events.

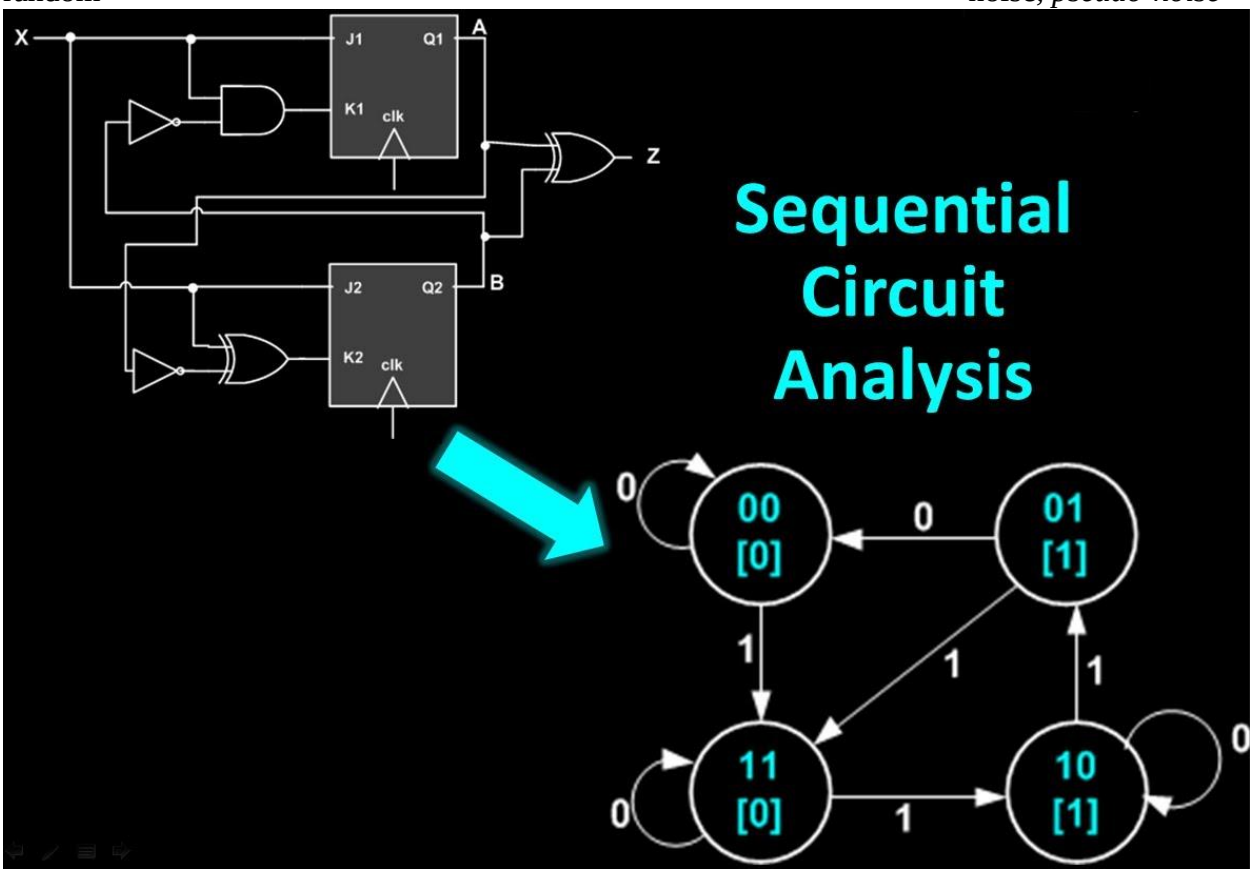
Shift registers produce a discrete delay of a digital signal or waveform. A waveform synchronized to a *clock*, a repeating square wave, is delayed by **"n"** discrete clock times, where **"n"** is the number of shift register stages. Thus, a four stage shift register delays "data in" by four clocks to "data out". The stages in a shift register are *delay stages*, typically type **"D"** Flip-Flops or type **"JK"** Flip-flops.

Formerly, very long (several hundred stages) shift registers served as digital memory. This obsolete application is reminiscent of the acoustic mercury delay lines used as early computer memory.

Serial data transmission, over a distance of meters to kilometers, uses shift registers to convert parallel data to serial form. Serial data communications replaces many slow parallel data wires with a single serial high speed circuit.

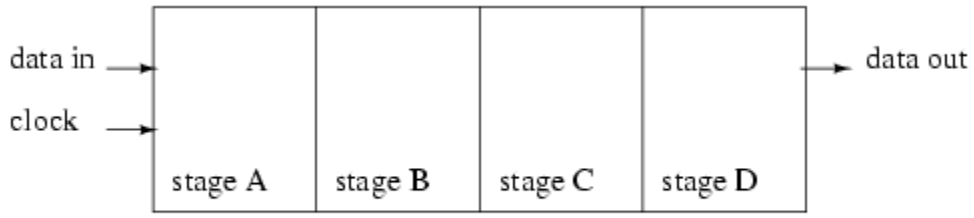
Serial data over shorter distances of tens of centimeters, uses shift registers to get data into and out of microprocessors. Numerous peripherals, including analog to digital converters, digital to analog converters, display drivers, and memory, use shift registers to reduce the amount of wiring in circuit boards.

Some specialized counter circuits actually use shift registers to generate repeating waveforms. Longer shift registers, with the help of feedback generate patterns so long that they look like random noise, *pseudo-noise*



Basic shift registers are classified by structure according to the following types:

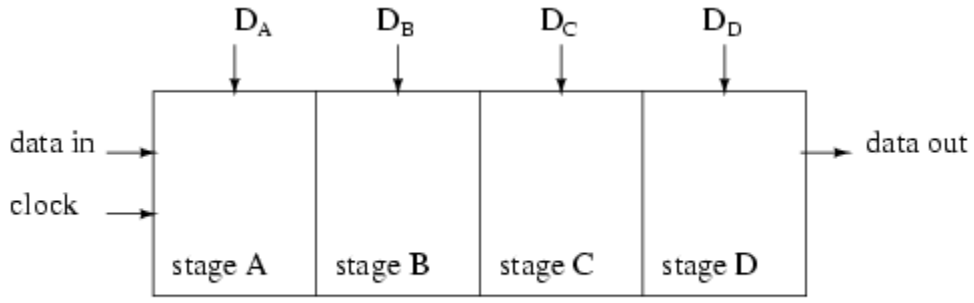
- Serial-in/serial-out
- Parallel-in/serial-out
- Serial-in/parallel-out
- Universal parallel-in/parallel-out
- Ring counter



Serial-in, serial-out shift register with 4-stages

Above we show a block diagram of a serial-in/serial-out shift register, which is 4-stages long. Data at the input will be delayed by four clock periods from the input to the output of the shift register.

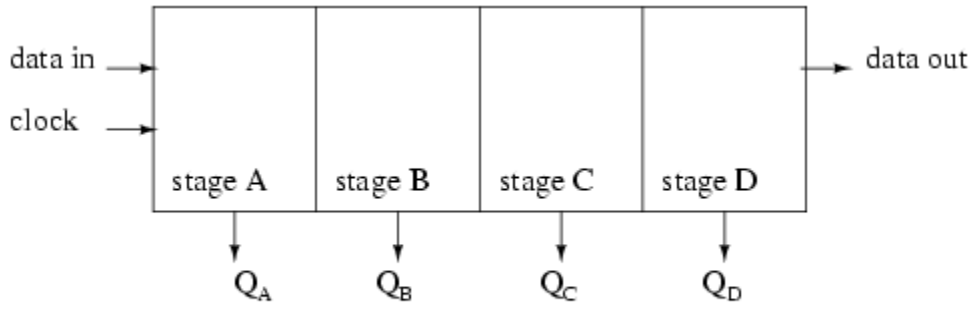
Data at "data in", above, will be present at the Stage **A** output after the first clock pulse. After the second pulse stage **A** data is transfered to stage **B** output, and "data in" is transfered to stage **A** output. After the third clock, stage **C** is replaced by stage **B**; stage **B** is replaced by stage **A**; and stage A is replaced by "data in". After the fourth clock, the data originally present at "data in" is at stage **D**, "output". The "first in" data is "first out" as it is shifted from "data in" to "data out".



Parallel-in, serial-out shift register with 4-stages

Data is loaded into all stages at once of a parallel-in/serial-out shift register. The data is then shifted out via "data out" by clock pulses. Since a 4- stage shift register is shown above, four clock pulses are required to shift out all of the data. In the diagram above, stage **D** data will be present at the "data out" up until the first clock pulse; stage **C** data will be present at "data out" between the first clock and the second clock pulse; stage **B** data will be present between the second clock and the third clock; and stage **A** data will be present between the third and the fourth clock. After the fourth clock pulse and thereafter, successive bits of "data in" should appear at "data out" of the shift register after a delay of four clock pulses.

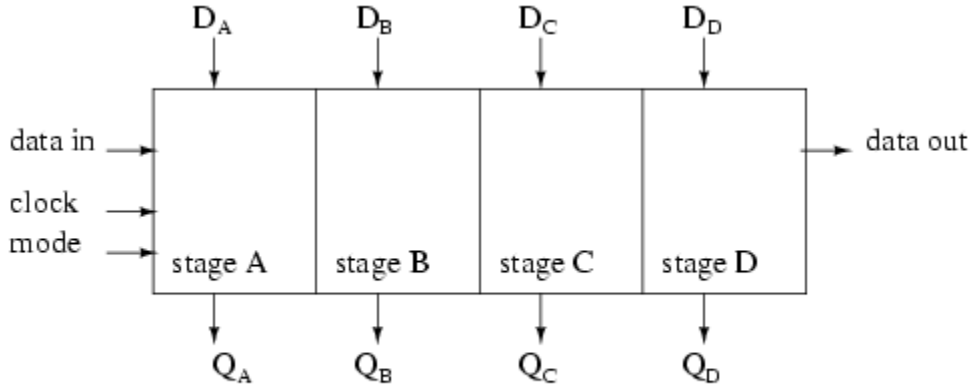
If four switches were connected to D_A through D_D, the status could be read into a microprocessor using only one data pin and a clock pin. Since adding more switches would require no additional pins, this approach looks attractive for many inputs.



Serial-in, parallel-out shift register with 4-stages

Above, four data bits will be shifted in from "data in" by four clock pulses and be available at Q_A through $Q_D$for driving external circuitry such as LEDs, lamps, relay drivers, and horns.

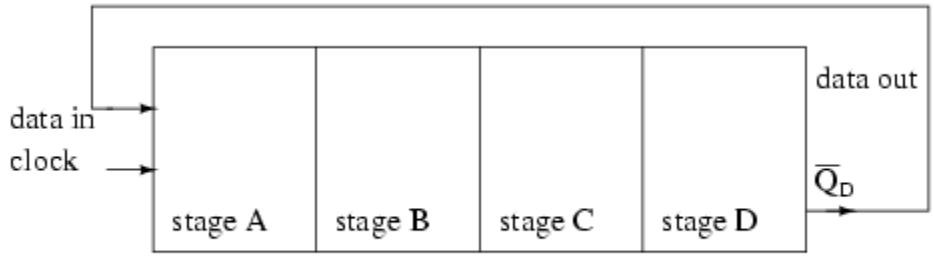
After the first clock, the data at "data in" appears at Q_A. After the second clock, The old Q_A data appears at Q_B; Q_A receives next data from "data in". After the third clock, Q_B data is at Q_C. After the fourth clock, $Q_C$data is at Q_D. This stage contains the data first present at "data in". The shift register should now contain four data bits.



Parallel-in, parallel-out shift register with 4-stages

A parallel-in/parallel-out shift register combines the function of the parallel-in, serial-out shift register with the function of the serial-in, parallel-out shift register to yield the universal shift register. The "do anything" shifter comes at a price– the increased number of I/O (Input/Output) pins may reduce the number of stages which can be packaged.

Data presented at D_A through D_D is parallel loaded into the registers. This data at Q_A through Q_D may be shifted by the number of pulses presented at the clock input. The shifted data is available at Q_A through Q_D. The "mode" input, which may be more than one input, controls parallel loading of data from D_A through D_D, shifting of data, and the direction of shifting. There are shift registers which will shift data either left or right.



Ring Counter, shift register output fed back to input

If the serial output of a shift register is connected to the serial input, data can be perpetually shifted around the ring as long as clock pulses are present. If the output is inverted before being fed back as shown above, we do not have to worry about loading the initial data into the "ring counter".

Design of Synchronous Sequential Circuits Objectives 1. Design of synchronous sequential circuits with an example. 2. Construction of state diagrams and state tables/ 3. Translation of State transition table into excitation table. 4. Logic diagram construction of a synchronous sequential circuit Sequential Circuit Design Steps *ƒ* The design of sequential circuit starts with verbal specifications of the problem (See Figure 1). Figure 1: Sequential Circuit Design Steps *ƒ* The next step is to derive the state table of the sequential circuit. A state table represents the verbal specifications in a tabular form. *ƒ* In certain cases state table can be derived directly from verbal description of the problem. *ƒ* In other cases, it is easier to first obtain a state diagram from the verbal description and then obtain the state table from the state diagram. *ƒ* A state diagram is a graphical representation of the sequential circuit. *ƒ* In the next step, we proceed by simplifying the state table by minimizing the number of states and obtain a reduced state table. 1 *ƒ* The states in the reduced state table are then assigned binary-codes. The resulting table is called output and state transition table. *ƒ* From the state transition table and using flip-flop's excitation tables, flip-flops input equations are derived. Furthermore, the output equations can readily be derived as well. *ƒ* Finally, the logic diagram of the sequential circuit is constructed. *ƒ* An example will be used to illustrate all these concepts. Sequence Recognizer *ƒ* A sequence recognizer is to be designed to detect an input sequence of '1011'. The sequence recognizer outputs a '1' on the detection of this input sequence. The sequential circuit is to be designed using JK and D type flip-flops. *ƒ* A sample input/output trace for the sequence detector is shown in Table 1. Table 1: Sample Input/Output Trace Input 0 1 1 0 1 0 1 1 0 1 1 1 0 1 0 1 1 1 0 0 Output 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 *ƒ* We will begin solving the problem by first forming a state diagram from the verbal description. *ƒ* A state diagram consists of circles (which represent the states) and directed arcs that connect the circles and represent the transitions between states. *ƒ* In a state diagram: 1. The number of circles is equal to the number of states. Every state is given a label (or a binary encoding) written inside the corresponding circle. 2. The number of arcs leaving any circle is 2n , where n is the number of inputs of the sequential circuit. 3. The label of each arc has the notation x/y, where x is the input vector that causes the state transition, and y is the value of the output during that present

state. 4. An arc may leave a state and end up in the same or any other state. 2 *f* Before we begin our design, the following should be noted. 1. We do not have an idea about how many states the machine will have. 2. The states are used to "remember" something about the history of past inputs. For the sequence 1011, in order to be able to produce the output value 1 when the final 1 in the sequence is received, the circuit must be in a state that "remembers" that the previous three inputs were 101. 3. There can be more than one possible state machine with the same behavior. Deriving the State Diagram Let us begin with an initial state (since a state machine must have at least one state) and denote it with 'S0' as shown in Figure 2 (a). *f f f f f f f* Two arcs leave state 'S0' depending on the input (being a 0 or a 1). If the input is a 0, then we return back to the same state. If the input is a 1, then we have to remember it (recall that we are trying to detect a sequence of 1011). We remember that the last input was a one by changing the state of the machine to a new state, say 'S1'. This is illustrated in Figure 2 (b). 'S1' represents a state when the last single bit of the sequence was one. Outputs for both transitions are zero, since we have not detected what we are looking for. Again in state 'S1', we have two outgoing arcs. If the input is a 1, then we return to the same state and if the input is a 0, then we have to remember it (second number in the sequence). We can do so by transiting to a new state, say 'S2'. This is illustrated in Figure 2 (c). Note that if the input applied is '1', the next state is still 'S1' and not the initial state 'S0'. This is because we take this input 1 as the first digit of new sequence. The output still remains 0 as we have not detected the sequence yet. State 'S2' represents detection of '10' as the last two bits of the sequence. If now the input is a '1', we have detected the third bit in our sequence and need to remember it. We remember it by transiting to a new state, say 'S3' as shown in Figure 2 (d). If the input is '0' in state 'S2' then it breaks the sequence and we need to start all over again. This is achieved by transiting to initial state 'S0'. The outputs are still 0. In state 'S3', we have detected input sequence '101'. Another input 1 completes our detection sequence as shown in Figure 2 (e). This is signaled by an output 1. However we transit to state 'S1' instead of 'S0' since this input 1 can be counted as first 1 of a new sequence. Application of input 0 to state 'S3' means an input sequence of 1010. This implies the last two bits in the sequence were 10 and we transit to a state that remembers this input sequence, i.e. state 'S2'. Output remains as zero. 3 Figure 2: Deriving the State Diagram of the Sequence Recognizer Deriving the State Table A state table represents time sequence of inputs, outputs, and states in a tabular form. The state table for the previous state diagram is shown in Table 2. *f f f f* The state table can also be represented in an alternate form as shown in Table 3. Here the present state and inputs are tabulated as inputs to the combinational circuit. For every combination of present state and input, next state column is filled from the state table. The number of flip-flops required is equal to □log2(number of states)□. 4 Thus, the state machine given in the figure will require two flip-flops □log2(4)□=2. We assign letters A and B to them. *f* Table 2: State Table of the Sequence Recognizer Present Next State Output State X=0 X=1 X=0 X=1 S0 S0 S1 0 0 S1 S2 S1 0 0 S2 S0 S3 0 0 S3 S2 S1 0 1 Table 3: Alternative Format of Table 2 Inputs of Combinational Circuit Present State Input Next State Output S0 0 S0 0 S0 1 S1 0 S1 0 S2 0 S1 1 S1 0 S2 0 S0 0 S2 1 S3 0 S3 0 S2 0 S3 1 S1 1 State Assignment The states in the constructed state diagram have been assigned symbolic names rather than binary codes. *f f f f* It is necessary to replace these symbolic names with binary codes in order to proceed with the design. In general, if there are m states, then the codes must contain n bits, where $2n \geq m$, and each state must be assigned a unique code. There can be many possible assignments for our state machine. One possible assignment is show in Table 4. Table 4: State Assignment State

Assignment S0 00 S1 01 S2 10 S3 11 *f* The assignment of state codes to states results in state transition table as shown. 5 It is important to mention here that the binary code of the present state at a given time t represents the values stored in the flip-flops; and the next-state represents the values of the flip-flops one clock period later, at time t+1. *f* Table 5: State Transition Table Inputs of Combinational Circuit 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 1 0 0 0 1 1 0 1 0 1 0 0 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 1 1 General Structure of Sequence Recognizer *f* The specifications required using JK and D type flip-flops. *f* Referring to the general structure of sequential circuit shown in Figure 3, our synthesized circuit will look like that as shown in the figure. Observe the feedback paths. Figure 3: General Structure of the Sequenc Recognizer What remains to be determined is the combinational circuit which specifies the external outputs and the flip-flop inputs. *f f* The state transition table as shown can now be expanded to construct the excitation table for the circuit. 6 Since we are designing the sequential circuit using JK and D type flip-flops, we need to correlate the required transitions in state transition table with the excitation tables of JK and D type-flip-flops. *f f f* The functionality of the required combinational logic is encapsulated in the excitation table. Thus, the excitation table is next simplified using map or other simplification methods to yield Boolean expressions for inputs of the used flip-flops as well as the circuit outputs. Deriving the Excitation Table The excitation table (See Table 6) describes the behavior of the combinational portion of sequential circuit. Table 6: Excitation Table of the Sequence Recognizer Present State Input Flip-flops Inputs A B X A B Y JA KA DB 0 0 0 0 0 0 0 X 0 0 0 1 0 1 0 0 X 1 0 1 0 1 0 0 1 X 0 0 1 1 0 1 0 0 X 1 1 0 0 0 0 0 X 1 0 1 0 1 1 1 0 X 0 1 1 1 0 1 0 0 X 0 0 1 1 1 0 1 1 X 1 1 For deriving the actual circuitry for the combinational circuit, we need to simplify the excitation table in a similar way we used to simplify truth tables for purely combinational circuits. *f f f f* Whereas in combinational circuits, our concern were only circuit outputs; in sequential circuits, the combinational circuitry is also feeding the flip-flops inputs. Thus, we need to simplify the excitation table for both outputs as well as flip-flops inputs. We can simplify flip-flop inputs and output using K-maps as shown in Figure 4. Finally the logic diagram of the sequential circuit can be made as shown in Figure 5.

Moore or Mealy model

Finite automata may have outputs corresponding to each transition. There are two types of finite state machines that generate output −

- Mealy Machine
- Moore machine

Mealy Machine

A Mealy Machine is an FSM whose output depends on the present state as well as the present input.

It can be described by a 6 tuple $(Q, \sum, O, \delta, X, q_0)$ where −

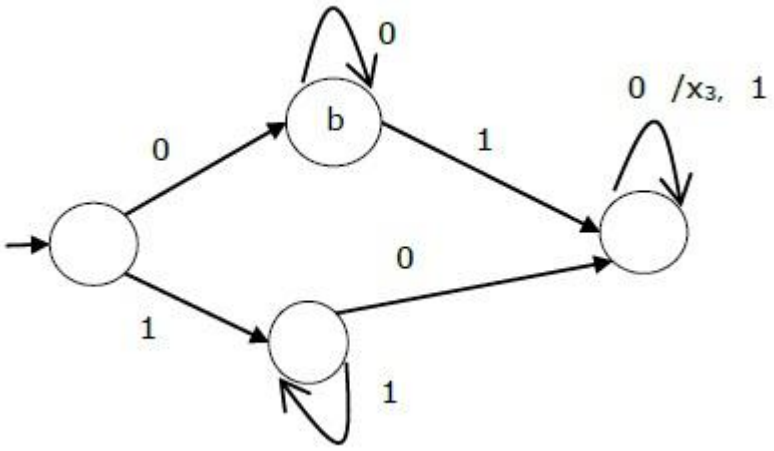
- **Q** is a finite set of states.

- $\sum$ is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where $\delta: Q \times \sum \rightarrow Q$
- **X** is the output transition function where $X: Q \times \sum \rightarrow O$
- $\mathbf{q_0}$ is the initial state from where any input is processed ($q_0 \in Q$).

The state table of a Mealy Machine is shown below −

Present state	**Next state**			
	input = 0		**input = 1**	
	State	**Output**	**State**	**Output**
→ a	b	x_1	c	x_1
b	b	x_2	d	x_3
c	d	x_3	c	x_1
d	d	x_3	d	x_2

The state diagram of the above Mealy Machine is −



Moore Machine

Moore machine is an FSM whose outputs depend on only the present state.

A Moore machine can be described by a 6 tuple (Q, ∑, O, δ, X, q_0) where −

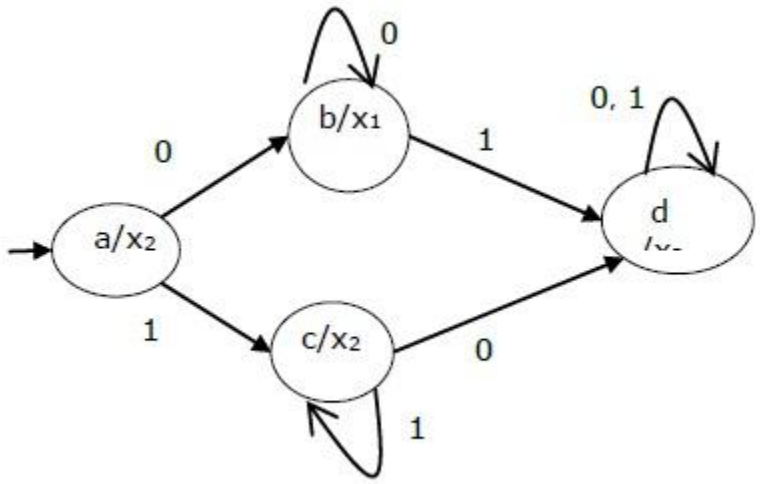
- **Q** is a finite set of states.
- ∑ is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where δ: Q × ∑ → Q
- **X** is the output transition function where X: Q → O
- **q_0** is the initial state from where any input is processed (q_0 ∈ Q).

The state table of a Moore Machine is shown below −

Present state	**Next State**		**Output**
	Input = 0	**Input = 1**	
→ a	b	c	x_2

b	b	d	x_1
c	c	d	x_2
d	d	d	x_3

The state diagram of the above Moore Machine is −



Mealy Machine vs. Moore Machine

The following table highlights the points that differentiate a Mealy Machine from a Moore Machine.

Mealy Machine	**Moore Machine**
Output depends both upon present state and present input.	Output depends only upon the present state.
Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
Output changes at the clock edges.	Input change can cause change in output change as soon as logic is done.

Mealy machines react faster to inputs	In Moore machines, more logic is needed to decode the outputs since it has more circuit delays.

Moore Machine to Mealy Machine

Algorithm 4

Input − Moore Machine

Output − Mealy Machine

Step 1 − Take a blank Mealy Machine transition table format.

Step 2 − Copy all the Moore Machine transition states into this table format.

Step 3 − Check the present states and their corresponding outputs in the Moore Machine state table; if for a state Q_i output is m, copy it into the output columns of the Mealy Machine state table wherever Q_i appears in the next state.

Example

Let us consider the following Moore machine −

Present State	**Next State**		**Output**
	a = 0	**a = 1**	
→ a	d	b	1
b	a	d	0
c	c	c	0
d	b	a	1

Now we apply Algorithm 4 to convert it to Mealy Machine.

Step 1 & 2 −

Present State	Next State			
	a = 0		a = 1	
	State	Output	State	Output
→ a	d		b	
b	a		d	
c	c		c	
d	b		a	

Step 3 −

Present State	Next State			
	a = 0		a = 1	
	State	Output	State	Output
=> a	d	1	b	0
b	a	1	d	1
c	c	0	c	0
d	b	0	a	1

Mealy Machine to Moore Machine

Algorithm 5

Input − Mealy Machine

Output − Moore Machine

Step 1 − Calculate the number of different outputs for each state (Q_i) that are available in the state table of the Mealy machine.

Step 2 − If all the outputs of Qi are same, copy state Q_i. If it has n distinct outputs, break Q_i into n states as Q_{in} where **n** = 0, 1, 2.....

Step 3 − If the output of the initial state is 1, insert a new initial state at the beginning which gives 0 output.

Example

Let us consider the following Mealy Machine −

Present State	**Next State**			
	a = 0		**a = 1**	
	Next State	**Output**	**Next State**	**Output**
→ a	d	0	b	1
b	a	1	d	0
c	c	1	c	0
d	b	0	a	1

Here, states ‘a’ and ‘d’ give only 1 and 0 outputs respectively, so we retain states ‘a’ and ‘d’. But states ‘b’ and ‘c’ produce different outputs (1 and 0). So, we divide **b** into **b_0, b_1** and **c** into **c_0, c_1.**

Present State	Next State		Output
	a = 0	**a = 1**	
→ a	d	b_1	1
b_0	a	d	0
b_1	a	d	1
c_0	c_1	C_0	0
c_1	c_1	C_0	1
d	b_0	a	0

MOD Counters

The job of a counter is to count by advancing the contents of the counter by one count with each clock pulse.

Counters which advance their sequence of numbers or states when activated by a clock input are said to operate in a “count-up” mode. Likewise, counters which decrease their sequence of numbers or states when activated by a clock input are said to operate in a “count-down” mode. Counters that operate in both the UP and DOWN modes, are called bidirectional counters.

Counters are sequential logic devices that are activated or triggered by an external timing pulse or clock signal. A counter can be constructed to operate as a synchronous circuit or as an asynchronous circuit. With synchronous counters, all the data bits change synchronously with the application of a clock signal. Whereas an asynchronous counter circuit is independent of the input clock so the data bits change state at different times one after the other.

Then counters are sequential logic devices that follow a predetermined sequence of counting states which are triggered by an external clock (CLK) signal. The number of states or counting sequences through which a particular counter advances before returning once again back to its original first state is called the **modulus** (MOD). In other words, the modulus (or just modulo) is the number of states the counter counts and is the dividing number of the counter.

Modulus counters, or simply *MOD counters*, are defined based on the number of states that the counter will sequence through before returning back to its original value. For example, a 2-bit counter that counts from 00_2 to 11_2 in binary, that is 0 to 3 in decimal, has a modulus value of 4 ($00 \rightarrow 01 \rightarrow 10 \rightarrow 11$, return back to 00) so would therefore be called a modulo-4, or mod-4, counter. Note also that it has taken 4 clock pulses to get from 00 to 11.

As in this simple example there are only two bits, (n = 2) then the maximum number of possible output states (maximum modulus) for the counter is: $2^n = 2^2$ or 4. However, counters can be designed to count to any number of 2^n states in their sequence by cascading together multiple counting stages to produce a single modulus or MOD-N counter.

Therefore, a "Mod-N" counter will require "N" number of flip-flops connected together to count a single data bit while providing 2^n different output states, (n is the number of bits). Note that N is always a whole integer value.

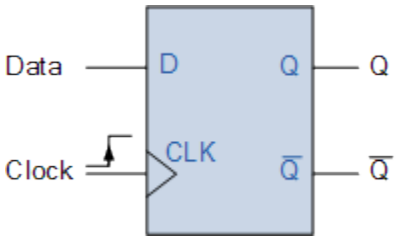
The we can see that MOD counters have a modulus value that is an integral power of 2, that is, 2, 4, 8, 16 and so on to produce an n-bit counter depending on the number of flip-flops used, and how they are connected, determining the type and modulus of the counter.

D-type Flip-flops

MOD counters are made using "flip-flops" and a single flip-flop can produce a count of 0 or 1, giving a maximum count of 2. There are different types of flip-flop designs we could use, the S-R, the J-K, J-K Master-slave, the D-type or even the T-type flip-flop to construct a counter. But to keep things simple, we will use the D-type flip-flop, (DFF) also known as a Data Latch, because a single data input and external clock signal are used, and is also positive edge triggered.

The D-type flip-flop, such as the TTL 74LS74, can be made from either S-R or J-K based edge-triggered flip-flops depending on whether you want it to change state either on the positive or leading edge (0 to 1 transition) or on the negative or trailing edge (1 to 0 transition) of the clock pulse. Here we will assume a positive, leading-edge triggered flip-flop. You can find more information in the following link about **D-type flip-flops**.

D-type Flip-flop and Truth Table



CLK	D	Q	$\overline{Q}$
↑	H	H	L
↑	L	L	H
L	X	no change	

The operation of a D-type flip-flop, (DFF) is very simple as it only has a single data input, called "D", and an additional clock "CLK" input. This allows a single data bit (0 or 1) to be stored under the control of the clock signal thus making the D-type flip-flop a synchronous device because the data on the inputs is transferred to the flip-flops output only on the triggering edge of the clock pulse.

So from the truth table, if there is a logic "1" (HIGH) on the Data input when a positive clock pulse is applied, the flip-flop SET's and stores a logic "1" at "Q", and a complimentary "0" at Q. Likewise, if there is a LOW on the Data input when another positive clock pulse is applied, the flip-flop RESET's and stores a "0" at "Q", and a resulting "1" at Q.

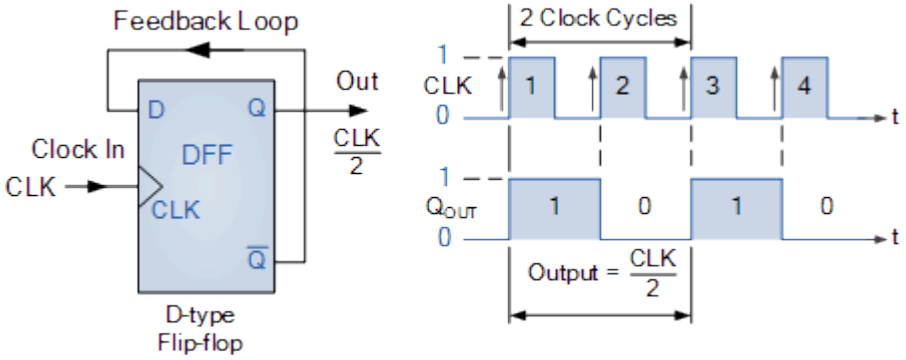
Then the output "Q" of the D-type flip-flop responds to the value of the input "D" when the clock (CLK) input is HIGH. When the clock input is LOW, the condition at "Q", either "1" or "0" is held until the next time the clock signal goes HIGH to logic level "1". Therefore the output at "Q" only changes state when the clock input changes from a "0" (LOW) value to a "1" (HIGH) making it a positive edge triggered D-type flip-flop. Note that negative edge-triggered flip-flops work in exactly the same way except that the falling edge of the clock pulse is the triggering edge.

So now we know how an edge-triggered D-type flip-flop works, lets look at connecting some together to form a MOD counter.

Divide-by-Two Counter

The edge-triggered D-type flip-flop is a useful and versatile building block to construct a MOD counter or any other type of sequential logic circuit. By connecting the Q output back to the "D" input as shown, and creating a feedback loop, we can convert it into a binary divide-by-two counter using the clock input only as the Q output signal is always the inverse of the Q output signal.

Divide-by-two Counter and Timing Diagram



The timing diagrams show that the "Q" output waveform has a frequency exactly one-half that of the clock input, thus the flip-flop acts as a frequency divider. If we added another D-type flip-flop so that the output at "Q" was the input to the second DFF, then the output signal from this second DFF would be one-quarter of the clock input frequency, and so on. So for an "n" number of flip-flops, the output frequency is divided by 2n, in steps of 2.

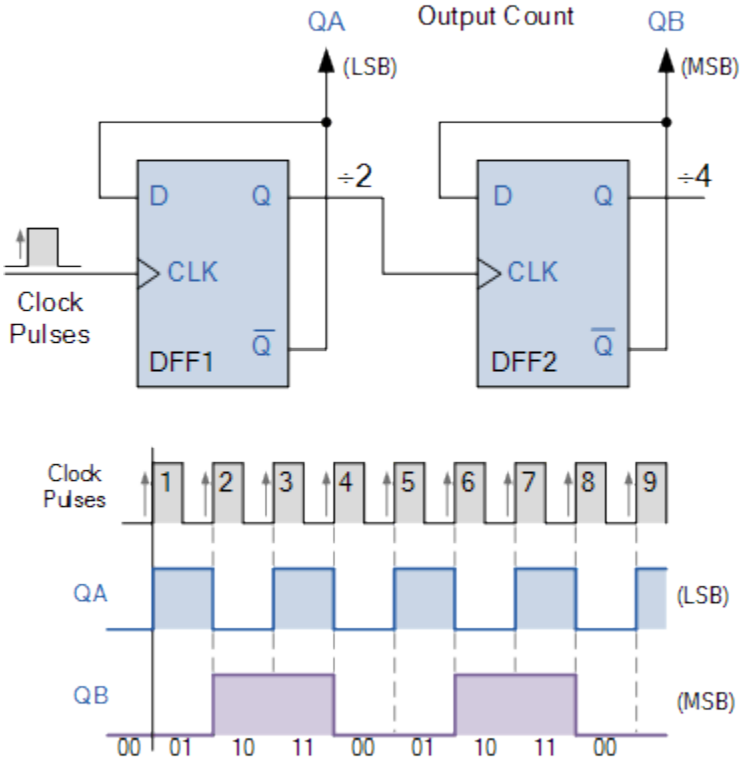
Note that this method of **frequency division** is very handy for use in sequential counting circuits. For example, a 60Hz mains frequency signal could be reduced to a 1Hz timing signal by using a divide-by-60 counter. A divide-by-6 counter would divide the 60Hz down to 10Hz which is then feed to a divide-by-10 counter to divide the 10Hz down to a 1Hz timing signal or pulse, etc.

MOD-4 Counter

Technically as well as being a 1-bit storage device, a single flip-flop on its own could be thought of as a MOD-2 counter, as it has a single output resulting in a count of two, either a 0 or 1, on the application of the clock signal. But a single flip-flop on its own produces a limited counting sequence, so by connecting together more flip-flops to form a chain, we can increase the counting capacity and construct a MOD counter of any value.

If a single flip-flop can be considered as a modulo-2 or MOD-2 counter, then adding a second flip-flop would give us a MOD-4 counter allowing it to count in four discrete steps. The overall effect would be to divide the original clock input signal by four. Then the binary sequence for this 2-bit MOD-4 counter would be: 00, 01, 10, and 11 as shown.

MOD-4 Counter and Timing Diagram

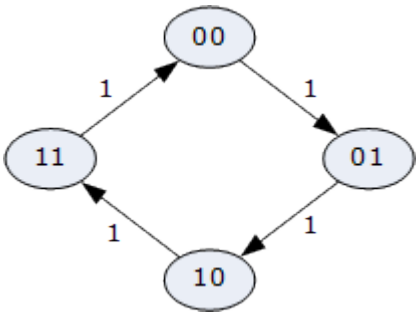


Note that for simplicity, the switching transitions of QA, QB and CLK in the above timing diagram are shown to be simultaneous even though this connection represents an asynchronous counter. In reality there would be a very small switching delay between the application of the positive going clock (CLK) signal, and the outputs at QA and QB.

We can show visually the operation of this 2-bit asynchronous counter using a truth table and state diagram.

MOD-4 Counter State Diagram

Clock Pulse	Present State			Next State		State Diagram
	Q_B	Q_A		Q_B	Q_A	
0 (start)	0	0	⇒	0	1	
1	0	1	⇒	1	0	
2	1	0	⇒	1	1	
3	1	1	⇒	0	0	
4 (repeat)	0	0	⇒	0	1	

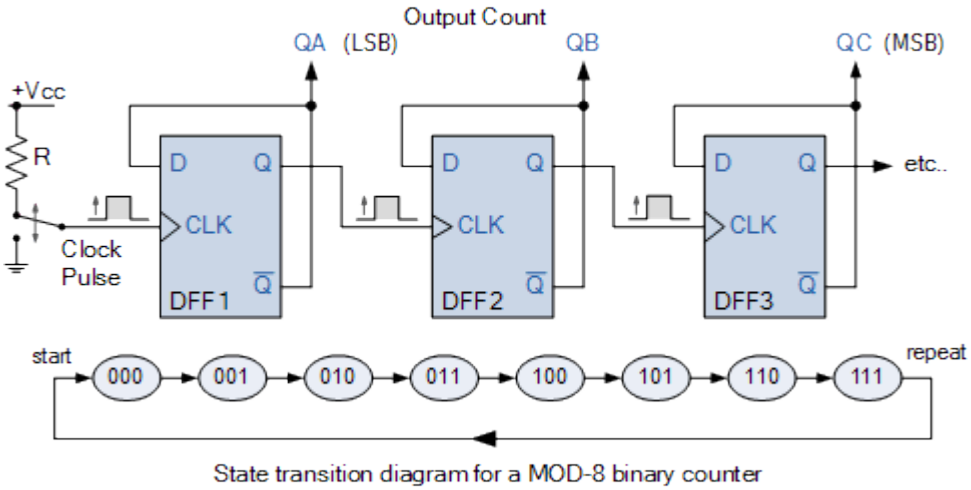


We can see from the truth table of the counter, and by reading the values of QA and QB, when QA = 0 and QB = 0, the count is 00. After the application of the clock pulse, the values become QA = 1, QB = 0, giving a count of 01 and after the next clock pulse, the values become QA = 0, QB = 1, giving a count of 10. Finally the values become QA = 1, QB = 1, giving a count of 11. The application of the next clock pulse causes the count to return back to 00, and thereafter it counts continuously up in a binary sequence of: 00, 01, 10, 11, 00, 01 …etc.

Then we have seen that a MOD-2 counter consists of a single flip-flop and a MOD-4 counter requires two flip-flops,allowing it to count in four discrete steps. We could easily add another

flip-flop onto the end of a MOD-4 counter to produce a MOD-8 counter giving us a 2^3 binary sequence of counting from 000 up to 111, before resetting back to 000. A fourth flip-flop would make a MOD-16 counter and so on, in fact we could go on adding extra flip-flops for as long as we wanted.

MOD-8 Counter and State Diagram



State transition diagram for a MOD-8 binary counter

We can therefore construct mod counters to have a natural count of 2^n states giving counters with mod counts of 2, 4, 8, 16, and so on, before repeating itself. But sometimes it is necessary to have a modulus counter that resets its count back to zero during the normal counting process and does not have a modulo that is a power of 2. For example, a counter having a modulus of 3, 5, 6, or 10.

Counters of Modulo "m"

Counters, either synchronous or asynchronous progress one count at a time in a set binary progression and as a result an "n"-bit counter functions naturally as a modulo 2^ncounter. But we can construct mod counters to count to any value we want by using one or more external logic gates causing it to skip a few output states and terminate at any count resetting the counter back to zero, that is all flip-flops have Q = 0.

In the case of modulo "m" counters, they do not count to all their possible states, but instead count to the "m" value and then return to zero. Obviously, "m" is a number smaller than 2^n, (**$m < 2^n$**). So how do we get a binary counter to return to zero part way through its count.

Fortunately, as well as counting, up or down, counters can also have additional inputs called *CLEAR* and *PRESET* which makes it possible to clear the count to zero, (all Q = 0) or to preset the counter to some initial value. The TTL 74LS74 has active-low Preset and Clear inputs.

Let's assume for simplicity that the CLEAR inputs are all connected together and are active-high inputs allowing the flip-flops to operate normally when the Clear input is equal to 0 (LOW). But if the Clear input is at logic level "1" (HIGH), then the next positive edge of the clock signal will reset all the flip-flops into the state Q = 0, regardless of the value of the next clock signal.

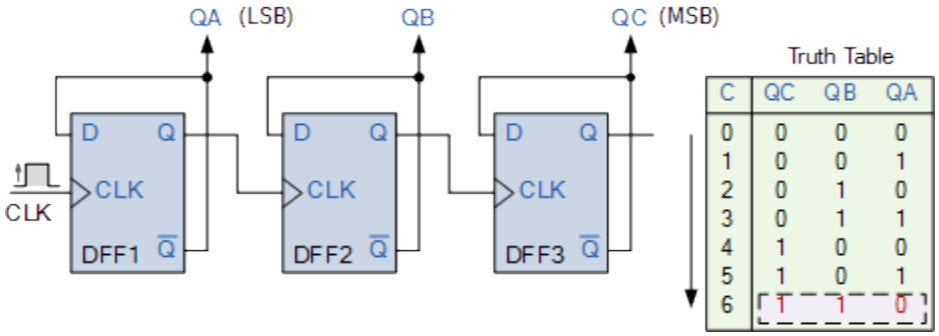
Note also that as all the Clear inputs are connected together, a single pulse can also be used to clear the outputs (Q) of all the flip-flops to zero before counting starts to ensure that the count actually starts from zero. Also some larger bit counters have an additional ENABLE or INHIBIT input pin which allows the counter to stop the count at any point in the counting cycle and hold its present state, before being allowed to continue counting again. This means the counter can be stopped and started at will without resetting the outputs to zero.

A Modulo-5 Counter

Suppose we want to design a MOD-5 counter, how could we do that. First we know that "m = 5", so 2^n must be greater than 5. As $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, and 8 is greater than 5, then we need a counter with three flip-flops (N = 3) giving us a natural count of 000 to 111 in binary (0 to 7 decimal).

Taking the MOD-8 counter above, the truth table for the natural count is given as:

MOD-8 Counter and Truth Table

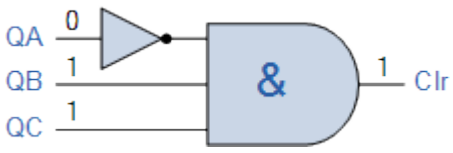


Truth Table

C	QC	QB	QA
0	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

As we are constructing a MOD-5 counter, we want the counter to reset back to zero after a count of 5. However, we can see from the attached truth table that the count of six gives us the output condition of: QA = 0, QB = 1, and QC = 1.

We can decode this output state of 011 (6) to give us a signal to clear (Clr) the counter back to zero with the help of a 3-input AND gate (TTL 74LS11) and an inverter or NOT gate, (TTL 74LS04).

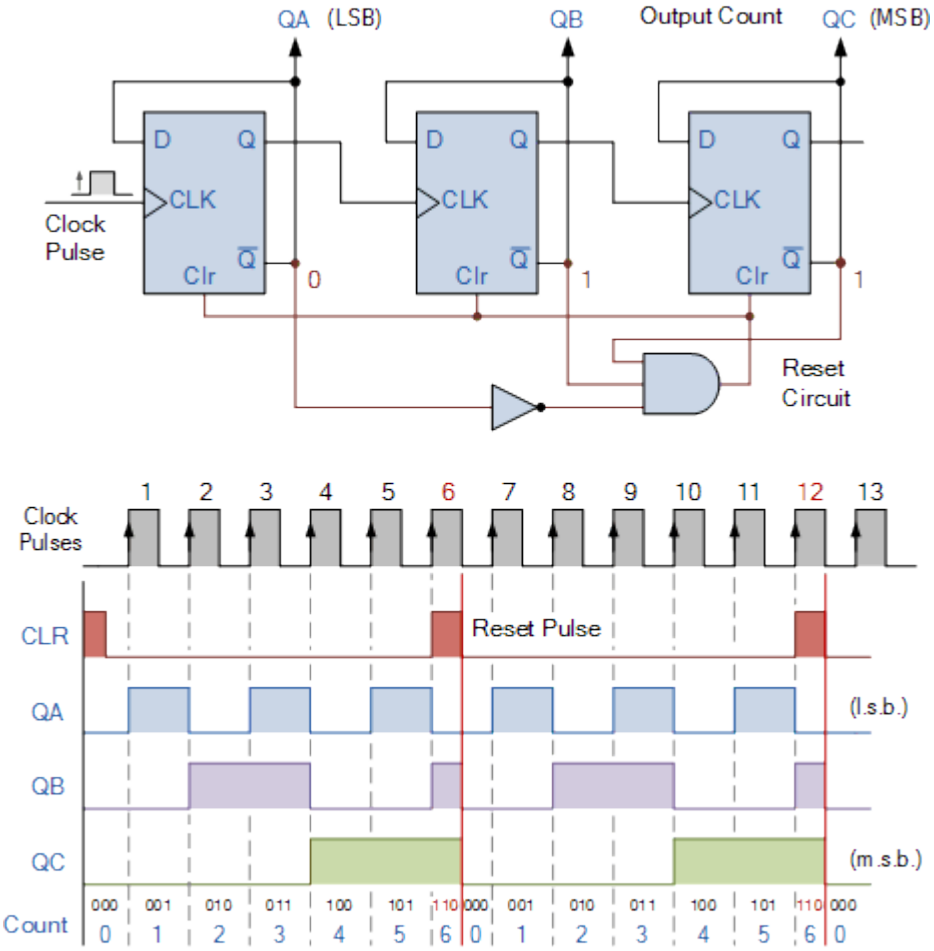


The inputs of the combinational logic circuit of the inverter and AND gate are connected to QA, QB, and QC respectively with the output of the AND gate at logic level "0" (LOW) for any combinations of the input other than the one we want.

In binary code, the output sequence count will look like this: 000, 001, 010, 011, 100, 101. But when it reaches the state of 011 (6), the combinational logic circuit will detect this 011 state and produce an output at logic level "1" (HIGH).

We can then use the resulting HIGH output from the AND gate to reset the counter back to zero after its output of 5 (decimal) count giving us the required MOD-5 counter. When the output from the combinational circuit is LOW it has no effect on the counting sequence.

MOD-5 Counter and Truth Table



Then we can use combinational logic decoding circuits around a basic counter, either synchronous or asynchronous to produce any type of MOD Counter we require as each of the counters unique output states can be decoded to reset the counter at the desired count.

In our simple example above, we have used a 3-input AND gate to decode the 011 state, but the first time that QA and QB are both at logic 1 is is when the count reaches six, so a 2-input AND gate connected to QA and QB could be used without the complication of the third input and the inverter.

However, one of the disadvantages of using asynchronous counters for producing a MOD counter of a desired count is that undesired effects called “glitches” can occur when the counter reaches its reset condition. During this brief time the outputs of the counter may take on an incorrect value, so it is sometimes better to use synchronous counters as modulo-m counters as all the flip-flops are clocked by the same clock signal so change state at the same time.

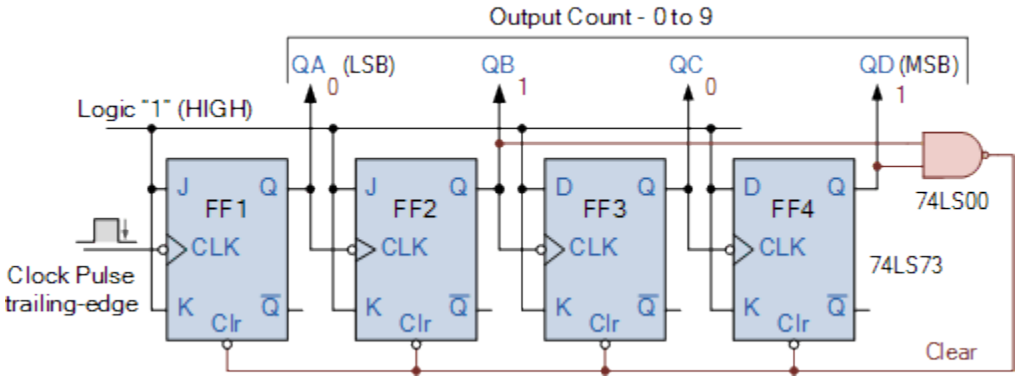
Modulus 10 Counter

A good example of a modulo-m counter circuit which uses external combinational circuits to produce a counter with a modulus of 10 is the Decade Counter. Decade (divide-by-10) counters such as the TTL 74LS90, have 10 states in its counting sequence making it suitable for human interfacing where a digital display is required.

The decade counter has four outputs producing a 4-bit binary number and by using external AND and OR gates we can detect the occurrence of the 9th counting state to reset the counter back to zero. As with other mod counters, it receives an input clock pulse, one by one, and counts up from 0 to 9 repeatedly.

Once it reaches the count 9 (1001 in binary), the counter goes back to 0000 instead of continuing on to 1010. The basic circuit of a decade counter can be made from JK flip-flops (TTL 74LS73) that switch state on the negative trailing-edge of the clock signal as shown.

MOD-10 Decade Counter



MOD Counter Summary

We have seen in this tutorial about **MOD Counters** that binary counters are sequential circuits that generate binary sequences of bits as a result of a clock signal and the state of a binary counter is determined by the specific combination formed by all the counters outputs together.

The number of different output states a counter can produce is called the modulo or modulus of the counter. The Modulus (or MOD-number) of a counter is the total number of unique states it passes through in one complete counting cycle with a mod-n counter being described also as a divide-by-n counter.

The modulus of a counter is given as: 2^n where n = number of flip-flops. So a 3 flip-flop counter will have a maximum count of $2^3 = 8$ counting states and would be called a MOD-8 counter. The maximum binary number that can be counted by the counter is $2^n–1$ giving a maximum count of $(111)_2 = 2^3–1 = 7_{10}$. Then the counter counts from 0 to 7.

Common MOD counters include those with MOD numbers of 2, 4, 8 and 16 and with the use of external combinational circuits can be configured to count to any predetermined value other than

one with a maximum 2^n modulus. In general, any arrangement of a "m" number of flip-flops can be used to construct any MOD counter.

A common modulus for counters with truncated sequences is ten (1010), called MOD-10. A counter with ten states in its sequence is known as a decade counter. Decade counters are useful for interfacing to digital displays. Other MOD counters include the MOD-6 or MOD-12 counter which have applications in digital clocks to display the time of day

State Reduction

Any design process must consider the problem of minimising the cost of the final circuit. The two most obvious cost reductions are reductions in the number of flip-flops and the number of gates.

The number of states in a sequential circuit is closely related to the complexity of the resulting circuit. It is therefore desirable to know when two or more states are equivalent in all aspects. The process of eliminating the equivalent or redundant states from a state table/diagram is known as **state reduction**.

Example: Let us consider the state table of a sequential circuit shown in Table 6.

Present State	Next State		Output	
	x = 0	x = 1	x = 0	x = 1
A	B	C	1	0
B	F	D	0	0
C	D	E	1	1
D	F	E	0	1
E	A	D	0	0
F	B	C	1	0

Table 6. State table

It can be seen from the table that the present state A and F both have the same next states, B (when x=0) and C (when x=1). They also produce the same output 1 (when x=0) and 0 (when x=1). Therefore states A and F are equivalent. Thus one of the states, A or F can be removed from the state table. For example, if we remove row F from the table and replace all F's by A's in the columns, the state table is modified as shown in Table 7.

Present State	Next State		Output	
	x = 0	x = 1	x = 0	x = 1

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

Table 7. State F removed

It is apparent that states B and E are equivalent. Removing E and replacing E's by B's results in the reduce table shown in Table 8.

Present State	Next State		Output	
	x = 0	x = 1	x = 0	x = 1
A	B	C	1	0
B	A	D	0	0
C	D	B	1	1
D	A	B	0	1

Table 8. Reduced state table

The removal of equivalent states has reduced the number of states in the circuit from six to four. Two states are considered to be **equivalent** if and only if for every input sequence the circuit produces the same output sequence irrespective of which one of the two states is the starting state.

UNIT IV

ASYNCHRONOUS SEQUENTIAL CIRCUITS AND PROGRAMMABLELOGIC DEVICES

Asynchronous Sequential Circuits

Asynchronous sequential circuits have state that is not synchronized with a clock. Like the synchronous sequential circuits we have studied up to this point they are realized by adding state feedback to combinational logic that implements a next-state function. Unlike synchronous circuits, the state variables of an asynchronous sequential circuit may change at any point in time. This asynchronous state update – from next state to current state – complicates the design process. We must be concerned with hazards in the next state function, as a momentary glitch may result in an incorrect final state. We must also be concerned with races between state variables on transitions between states whose encodings differ in more than one variable. In this chapter we look at the fundamentals of asynchronous sequential circuits. We start by showing how to analyze combinational logic with feedback by drawing a flow table. The flow table shows us which states are stable, which are transient, and which are oscillatory. We then show how to synthesize an asynchronous circuit from a specification by first writing a flow table and then reducing the flow table to logic equations. We see that state assignment is quite critical for asynchronous sequential machines as it determines when a potential race may occur. We show that some races can be eliminated by introducing transient states. After the introduction of this chapter, we continue our discussion of asynchronous circuits in Chapter 23 by looking at latches and flip-flops as examples of asynchronous circuits. 22.1 Flow Table Analysis Recall from Section 14.1 that an asynchronous sequential circuit is formed when a feedback path is placed around combinational logic as shown in Figure 22.1(a). To analyze such circuits, we break the feedback path as shown in Figure 22.1(b) 383 384 EE108A Class Notes C in L n out m state s (a) (b) C in L n out m current state s next state Figure 22.1: Asynchronous sequential circuit. (a) A sequential circuit is formed when a feedback path carrying state information is added to combinational logic. (b) To analyze an asynchronous sequential circuit, we break the feedback path and look at how the next state depends on the current state. and write the equations for the next state variables as a function of the current state variables and the inputs. We can then reason about the dynamics of the circuit by exploring what happens when the current state variables are updated, in arbitrary order if multiple bits change, with their new values. At first this may look just like the synchronous sequential circuits we discussed in Section 14.2. In both cases we compute a next state based on current state and input. What's different is the dynamics of how the current state is updated with the next state. Without a clocked state register, the state of an asynchronous sequential circuit may change at any time (asynchronously). When multiple bits of state are changing at the same time (a condition called a race. The bits may change at different rates resulting in different end states. Also, a synchronous circuit will eventually reach a steady state where the next state and outputs will not change until the next clock cycle. An

asynchronous circuit on the other hand may never reach a steady state. It is possible for it to oscillate indefinitely in the absence of input changes. We have already seen one example of analyzing an asynchronous circuit in this manner - the RS flip-flop of Section 14.1. In this section we look at some additional examples and introduce the flow table as a tool for the analysis and synthesis of asynchronous circuits. Consider the circuit shown in Figure 22.2(a). Each of the AND gates in the figure is labeled with the input state ab during which it is enabled. For example, the top gate, labeled 00, is enabled when a is high and b is low. To analyze the circuit we break the feedback loop as shown in Figure 22.2(b). At this point we can write down the next-state function in terms of the inputs, a and b, and the current state. This function is shown in tabular form in the flow table of Figure 22.2(c). Figure 22.2(c) shows the next state for each of the eight combinations of inputs and current state. Input states are shown horizontally in Gray-code order. Current states are shown vertically. If the next state is the same as the current state, this state is stable since updating the current state with the next Copyright (c) 2002-2007 by W.J Dally, all rights reserved 385 out a b out a b state next (a) (b) State Next 00 01 11 10 0 1 0 1 1 1 0 0 (c) 00 11 01 00 11 01 1 0 Figure 22.2: An example asynchronous sequential circuit. (a) The original circuit. (b) With feedback loop broken. (c) Flow table showing next-state function. Circled entries in the flow table are stable states. 386 EE108A Class Notes state doesn't change anything. If the next state is different than the current state, this state is transient since as soon as the current state is updated with the next state, the circuit will change states. For example, suppose the circuit has inputs ab = 00 and the current state is 0. The next state is also 0, so this is a stable state - as shown by the circled 0 in the leftmost position of the top row of the table. If from this state input b goes high, making the input state ab = 01, we move one square to the right in the table. In this case, the 01 AND gate is enabled and the next-state is 1. This is an unstable or transient situation since the current state and next state are different. After some amount of time (for the change to propagate) the current state will become 1 and we move to the bottom row of the table. At this point we have reached a stable state since the current and next state are now both 1. If there is a cycle of transient states with no stable states we have an oscillation. For example, if the inputs to the circuit of Figure 22.2 are ab = 11, the next state is always the complement of the current state. With this input state, the circuit is never stable, but instead will oscillate indefinitely between the 0 and 1 states. This is almost never a desired behavior. An oscillation in an asynchronous circuit is almost always an error. So, what does the circuit of Figure 22.2 do? By this point the estute reader will have realized that its an RS flip-flop with an oscillation feature added. Input a is the reset input. When a is high and b is low, the state is made 0 when a is lowered the state remains 0. Similarly b is the set input. Making b high while a is low sets the state to 1 and it remains at 1 when b is lowered. The only difference between this flip-flop and the one of Figure 14.2 is that when both inputs are high the circuit of Figure 22.2 oscillates while the circuit of Figure 14.2 resets. To simplify our analysis of asynchronous circuits we typically insist that the environment in which the circuits operate obey the fundamental mode restriction: Fundamental-Mode: Only one input bit may be changed at a time and the circuit must reach a stable state before another input bit is changed. A circuit operated in fundamental-mode need only worry about one input bit changing at a time. Multiple-bit input changes are not allowed. Our setupand hold-time restrictions on flip-flops are an example of a fundamental-mode restriction. The clock and data inputs of the flip flop are not allowed to

change at the same time. After the data input changes, the circuit must be allowed to reach a steady-state (setup time) before the clock input can change. Similarly, after the clock input changes, the circuit must be allowed to reach a steadystate (hold time) before the data input can change. We will look at the relation of setup and hold time to the design of the asynchronous circuits that realize flip-flops in more detail in Chapter 23. In looking at a flow-table, like the one in Figure 22.2, operating in the fundamental mode means that we need only consider input transitions to adjacent Copyright (c) 2002-2007 by W.J Dally, all rights reserved 387 Toggle in a b in a b Figure 22.3: A toggle circuit alternates pulses on it input in between its two outputs a and b. squares (including wrapping from leftmost to rightmost). Thus, we don't have to worry about what happens when the input changes from 11 (oscillating) to 00 (storing). This can't happen. Since only one input can change at a time, we must first visit state 10 (reset) or 01 (set) before getting to 00. In some real world situations, it is not possible to restrict the inputs to operate in fundamental mode. In these cases we need consider multiple input changes. This topic is beyond the scope of this book and the interested reader is referred to some of the texts listed in Section

22.4. 22.2 Flow-Table Synthesis: The Toggle Circuit We now understand how to use a flow-table to analyze the behavior of an asynchronus circuit. That is, given a schematic, we can draw a flow table and understand the function of the circuit. In this section we will use a flow table in the other direction. We will see how to create a flow table from the specification of a circuit and then use that flow table to synthesize a schematic for a circuit that realizes the specification. Consider the specification of a toggle circuit - shown graphically in Figure 22.3.

The toggle circuit has a single input in and two outputs a and b. 1 Whenever in is low, both outputs are low. The first time in goes high, output a goes high. On the next rising transition of in, output b goes high. On the third rising input, a goes high again. The circuit continues steering pulses on in alternately between a and b. The first step in synthesizing a toggle circuit is to write down its flow table. We can do this directly from the waveforms of Figure 22.3. Each transition of the input potentially takes us to a new state. Thus, we can partition the waveform into potential states as shown in Figure 22.4. We start in state A. When in rises we go to state B where output a is high. when in falls again we 1In practice a reset input rst is also required to initialize the state of the circuit. 388 EE108A Class Notes B C D A in a b A Next (in) 0 1 A C B B Out (a,b) C A D D 00 10 00 01 A B C D State Figure 22.4:

A flow table is created from the specification of the toggle circuit by creating a new state for every input transition until the circuit is obviously back to the same state. go to state C. Even though C has the same output as A, we know its a different state because the next transition on in will cause a different output. The second rising edge on in takes us to state D with output b high. When in falls for the second time we go back to state A. We know that this state is the same as state A since the behavior of the circuit at this point under all possible inputs is indistinguishable from where we started. Once we have a flow table for the toggle circuit, the next step is to assign binary codes to each of the states. This state assignment is more critical than with synchronous machines. If two states X and Y differ in more than one state bit, a transition from X to Y requires first visiting a transient state with one state bit changed before arriving at Y. In some cases, a race between the two state bits may result. We discuss races in more detail in Section

22.3. For now, we pick a state assignment (shown in Figure 22.5(a) where each state transition switches only a single bit. With the state assignment, realizing the logic for the toggle circuit is a simple matter of combinational logic synthesis. We redraw the flow table as a Karnaugh map in Figure 22.5(b). The Karnaugh map shows the symbolic next state function - i.e., each square shows the next state name (A through D) for that input and current state. The arrows show the path through the states followed during operation of the circuit. Understanding this path is important for avoiding races and hazards. We refer to this Karnaugh map showing the state transitions as a trajectory map since it shows the trajectory of the state variables. We redraw the Karnaugh map with state names replaced by their binary codes in Figure 22.5(c), and separate maps for the two state variables s0 and s1 Copyright (c) 2002-2007 by W.J Dally, all rights reserved 389 00 01 11 10 Code Next (in) 0 1 A C B B Out (a,b) C A D D 00 10 00 01 A B C D State A B C D D A in s0 s1 B C 01 11 11 10 00 in s0 s1 10 01 00 1 1 1 0 0 in s0 s1 0 1 0 in s0 s1 1 1 1 0 0 0 1 0 (a) (b) (c) (d) (e) Figure 22.5:

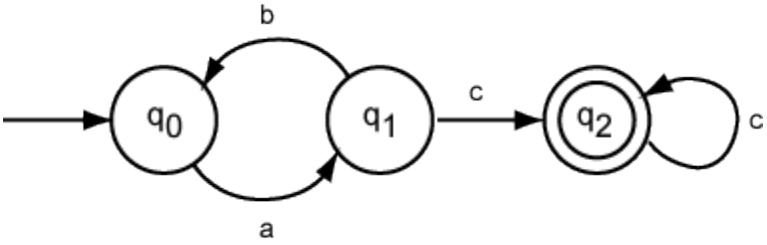
Implementing the toggle circuit from its flow table. (a) Flow table with state assignment. (b) Flow table mapped to Karnaugh map. (c) Next state codes mapped to Karnaugh map. (d) Karnaugh map for s0. (e) Karnaugh map for s1. 390 EE108A Class Notes are shown in Figure 22.5(d) and (e) respectively. From these Karnaugh maps we write down the equations for s0 and s1: s0 = (s1 ∧ in) ∨ (s0 ∧ in) ∨ (s0 ∧ s1), (22.1) s1 = (s1 ∧ in) ∨ (s0 ∧ in) ∨ (s0 ∧ s1). (22.2) The last implicant in each expression is required to avoid a hazard that would otherwise occur. Asynchronous circuits must be hazard free along their path through the input/state space. Because the current state is being constantly fed back a glitch during a state transition can result in the circuit switching to a different state - and hence not implementing the desired function. For example, suppose we left the s0 ∧ s1 term out of (22.2). When in goes low in state B, s0 might go momentarily low before s1 comes high. At this point the middle term of both equations becomes false and s1 never goes high - the circuit goes to state A rather than C. All that remains to complete our synthesis is to write the output equations. Output a is true in state 01 and output b is true in state 10. The equations are thus: a = s1 ∧ s0, (22.3) b = s1 ∧ s0. (22.4) 22.3 Races and State Assignment To illustrate the problem of multiple state variables changing simultaneously, consider an alternate state assignment for the toggle circuit shown in Figure 22.6(a). Here we observe that the two outputs, a and b can also serve as state variables, so we can add to the outputs just one additional state variable c to distinguish between states A and C giving the codes shown in the figure.2 With this state assignment, the transition from state A (cab = 000) to state B (110) changes both c and a. If the logic is designed so that in going high in state A makes both c and a go high, they could change in an arbitrary order. Variable a could change first, variable c could change first, or they could change simultaneously. If they change simultaneously, we go from state A directly to state B with no intermediate stops. If a changes first, we go first to state 010 which is not assigned and then, if the logic in state 010 does the right thing, to state 110. If c changes first, the machine will go to state C (100) where the high input will then drive it to state D. Clearly, we cannot allow c to change first. This situation where multiple state variables can change at the same time is called a race. The state variables are racing to see which one can change first. 2Note that the bit ordering of the codes is c,a,b. Copyright (c) 2002-2007 by W.J Dally, all rights reserved 391 000 110 100 001 Code (c,a,b) Next (in) 0 1 A C B B Out (a,b) C A

D D 00 10 00 01 A B C D State A B1 B B C D1 C A D D in c a b (a) (b) Figure 22.6: An alternate state assignment for the toggle circuit requires multiple state variables to change on a single transition. (a) The flow table with the revised state assignment. (b) A trajectory map showing the introduction of transient states B1 = 010 and D1 = 101. When the outcome of the race affects the end state - as in this case - we call the race a critical race. To avoid the critical race that could occur if we allow both a and c to change at the same time, we specify the next-state function so that only a can change in state A. This takes us to a transient state 010, which we will call B1. When the machine reaches state B1, the next state logic then enables c to change. The introduction of this transient state is illustrated in the trajectory map of Figure 22.6(b). When the input goes high in state A, the next state function specifies B1 rather than B. This enables only a single transition, downward as shown by the blue arrow - which corresponds to a rising, to state B1. A change in c is not enabled in this state to avoid a horizontal transition into the square marked D1. Once the machine reaches state B1, the next state function becomes B which enables the change in c, a horizontal transition, to stable state B. A transient state is also required for the transition from state C 100 to state D 001. Both variables c and b change between these two states. An uncontrolled race in which variable c changes first could wind up in state A 000 which is not correct. To prevent this race, we enable only b to change when in rises in state C. This takes us to a transient state D1 (101). Once in state D1, c is allowed to fall, taking us to state D (001). Figure 22.7 illustrates the process of implementing the revised toggle circuit. Figure 22.7(a) shows a Karnaugh map of the next-state function. Each square of the Karnaugh map shows the code for the next state for that present state and input. Note that where the next state equals the present state the state is stable. Transient state B1 (at 010 - the second square along the diagonal) is not stable since it has a next state of 110. 392 EE108A Class Notes 000 010 110 110 100 101 100 000 001 001 in c a b 0 x x x 1 1 0 0 1 3 2 4 5 7 6 8 9 11 10 12 13 15 14 00 01 11 10 c in ba 00 01 11 10 in c a b 0 x x 1 0 0 x 0 0 0 x x x 0 0 0 0 1 3 2 4 5 7 6 8 9 11 10 12 13 15 14 00 01 11 10 c in ba 00 01 11 10 in c a b 0 x x 0 1 1 x 1 0 0 x x x 1 1 1 0 1 3 2 4 5 7 1 8 9 11 10 12 13 15 14 00 01 11 10 c in ba 00 01 11 10 in c a b 0 x x 0 0 0 x 1 1 (a) (b) abc (c) (d) Figure 22.7: Implementation of the toggle circuit with the alternate state assignment of Figure 22.6(a). (a) Karnaugh map showing 3-bit next state function c,a,b. (b) Karnaugh map for a. (c) Karnaugh map for b. (d) Karnaugh map for c. Copyright (c) 2002-2007 by W.J Dally, all rights reserved 393 From the next-state Karnaugh map we can write the individual Karnaugh maps for each state variable. Figure 22.7 (b) through (d) show the Karnaugh maps for the individual variables - a, b, and c respectively. Note that the states that are not visited along the state trajectory (the blank squares in Figure 22.7(a)) are don't cares. The machine will never be in these states, thus we don't care what the next state function is in an unvisited state. From these Karnaugh maps we write the state variable equations as: $a = (in \wedge b \wedge c) \vee (in \wedge a)$, (22.5) $b = (in \wedge a \wedge c) \vee (in \wedge b)$, (22.6) $c = a \vee (b \wedge c)$. (22.7) Note that we don't require separate equations for output variables since a and b are both state variables and output variables.

Transition table

Example Transition Table

A transition table shows the transitions between states for each input character. Here is a finite automaton and a table that shows its transitions:



States	Inputs			
	a	b	c	all else
q_0	q_1	-	-	-
q_1	-	q_0	q_2	-
q_2	-	-	q_2	-

The column labeled **States** shows current states of the automaton, and the characters under the heading **Inputs** shows the labels of the transitions. The cell at row q_0 column 'a' (for example) shows that when the automaton is in state q_0, character 'a' causes a transition to state q_1.

When there is no transition out of the current state for the given character, the table contains a dash. You can think of this as representing the "reject state." The final column shows that characters other than 'a', 'b', or 'c' immediately lead to the reject state.

Utomata theory and sequential logic, a state transition table is a table showing what state (or states in the case of a nondeterministic finite automaton) a finite semiautomaton or finite state machine will move to, based on the current state and other inputs. A state table is essentially a truth table in which some of the inputs are the current state, and the outputs include the next state, along with other outputs.

A state table is one of many ways to specify a *state machine*, other ways being a state diagram, and a *characteristic equation*.

Common forms

One-dimensional state tables

Also called **characteristic tables**, single-dimension state tables are much more like truth tables than the two-dimensional versions. Inputs are usually placed on the left, and separated from the outputs, which are on the right. The outputs will represent the next state of the machine. A simple example of a state machine with two states and two combinational inputs follows:

A	B	Current State	Next State	Output
0	0	S_1	S_2	1
0	0	S_2	S_1	0
0	1	S_1	S_2	0
0	1	S_2	S_2	1
1	0	S_1	S_1	1
1	0	S_2	S_1	1
1	1	S_1	S_1	1
1	1	S_2	S_2	0

S_1 and S_2 would most likely represent the single bits 0 and 1, since a single bit can only have two states.

Two-dimensional state tables

State transition tables are typically two-dimensional tables. There are two common forms for arranging them.

- One of the dimensions indicates current states, while the other indicates events. The row/column intersections indicate the next state for a particular event, and (optionally) an incidental action associated with this state transition.

State Transition Table

Event **State**	**E_1**	**E_2**	**…**	**E_n**
S_1	—	S_y / A_j	…	—
S_2	—	—	…	S_x / A_i
…	…	…	…	…
S_m	S_z / A_k	—	…	—

(S: state, E: event, A: action, —: illegal transition)

- One of the dimensions indicates current states, while the other indicates next states. The row/column intersections indicate the event which will lead to a particular next state.

State Transition Table

State (Next) **State (Current)**	**S_1**	**S_2**	**…**	**S_m**
S_1	—	—	…	E_x / A_i
S_2	E_y / A_j	—	…	—

…	…	…	…	…
S_m	—	E_z / A_k	…	—

(S: state, E: event, A: action, —: impossible transition)

Other forms

Simultaneous transitions in multiple finite state machines can be shown in what is effectively an n-dimensional state transition table in which pairs of rows map (sets of) current states to next states.[1] This is an alternative to representing communication between separate, interdependent state machines.

At the other extreme, separate tables have been used for each of the transitions within a single state machine: "AND/OR tables"[2] are similar to incomplete decision tables in which the decision for the rules which are present is implicitly the activation of the associated transition.

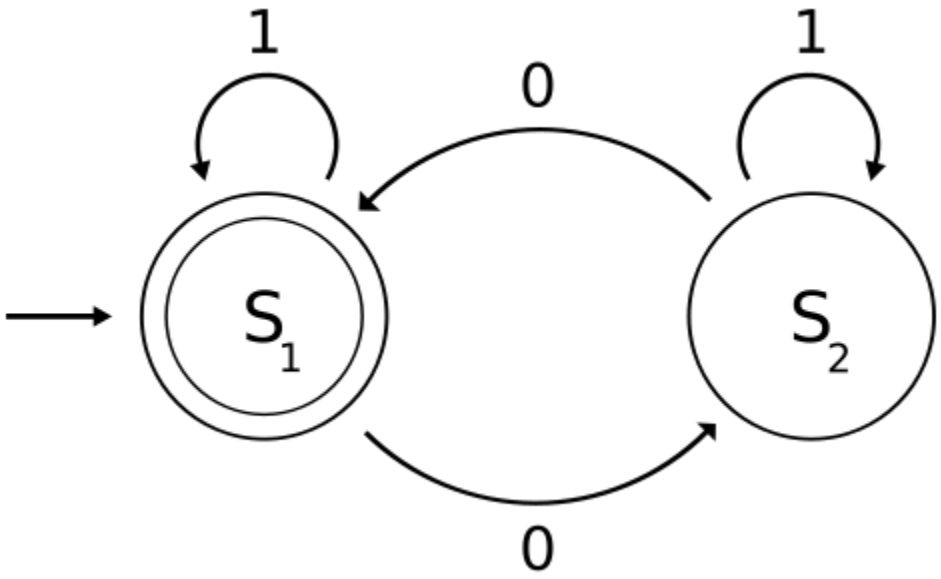
Example

An example of a state transition table for a machine **M** together with the corresponding state diagram is given below.

State Transition Table

Input / **State**	**1**	**0**
S_1	S_1	S_2
S_2	S_2	S_1

State Diagram



All the possible inputs to the machine are enumerated across the columns of the table. All the possible states are enumerated across the rows. From the state transition table given above, it is easy to see that if the machine is in **S_1** (the first row), and the next input is character **1**, the machine will stay in **S_1**. If a character **0** arrives, the machine will transition to **S_2**as can be seen from the second column. In the diagram this is denoted by the arrow from **S_1** to **S_2** labeled with a **0**.

For a nondeterministic finite automaton (NFA), a new input may cause the machine to be in more than one state, hence its non-determinism. This is denoted in a state transition table by a pair of curly braces *{ }* with the set of all target states between them. An example is given below.

State Transition Table for an NFA

Input **State**	**1**	**0**	**ε**
S_1	S_1	{ S_2, S_3 }	Φ
S_2	S_2	S_1	Φ
S_3	S_2	S_1	S_1

Here, a nondeterministic machine in the state **S_1** reading an input of **0** will cause it to be in two states at the same time, the states **S_2** and **S_3**. The last column defines the legal transition of states of the special character, ε. This special character allows the NFA to move to a different state when given no input. In state **S_3**, the NFA may move to **S_1** without consuming an input character. The two cases above make the finite automaton described non-deterministic.

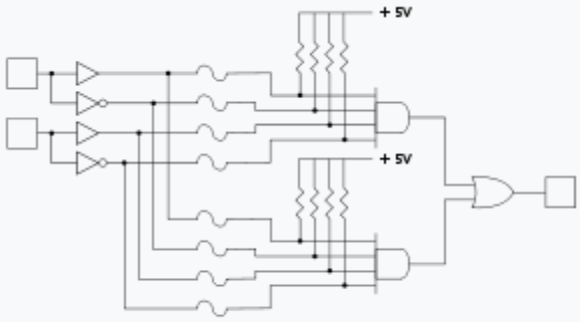
Transformations from/to state diagram

It is possible to draw a state diagram from the table. A sequence of easy to follow steps is given below:

1. Draw the circles to represent the states given.
2. For each of the states, scan across the corresponding row and draw an arrow to the destination state(s). There can be multiple arrows for an input character if the automaton is an NFA.
3. Designate a state as the start state. The start state is given in the formal definition of the automaton.
4. Designate one or more states as accept state. This is also given in the formal definition.

Programmable logic device

A **programmable logic device** (**PLD**) is an electronic component used to build reconfigurable digital circuits. Unlike a logic gate, which has a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed, that is, reconfigured.



Simplified programmable logic device

A simplified PAL device. The programmable elements (shown as a fuse) connect both the true and complemented inputs to the AND gates. These AND gates, also known as product terms, are ORed together to form a sum-of-products logic array.

Using a ROM as a PLD

Before PLDs were invented, read-only memory (ROM) chips were used to create ***arbitrary*** combinational logic functions of a number of inputs. Consider a ROM with *m* inputs (the address lines) and *n* outputs (the data lines). When used as a memory, the ROM contains 2^m words of *n* bits each.

Now imagine that the inputs are driven not by an *m*-bit address, but by *m* independent logic signals. Theoretically, there are 2^{2m} possible Boolean functions of these *m* input signals. By *Boolean function* in this context is meant a single function that maps each of the 2^m possible combinations of the *m* Boolean inputs to a single Boolean output. There are 2^{2m}possible distinct ways to map each of 2^m inputs to a Boolean value, which explains why there are 2^{2m} such Boolean functions of *m* inputs.

Now, consider that each of the *n* output pins acts, independently, as a logic device that is specially selected to sample **just one** of the possible 2^{2m} such functions. At any given time, only one of the 2^m possible input values can be present on the ROM, but over time, as the input values span their full possible domain, each output pin will map out its particular function of the 2^m possible input values, from among the 2^{2m} possible such functions. Note that the structure of the ROM allows just *n* of the 2^{2m} possible such Boolean functions to be produced at the output pins. The ROM therefore becomes equivalent to *n* separate logic circuits, each of which generates a chosen function of the *m* inputs.

The advantage of using a ROM in this way is that any conceivable function of all possible combinations of the *m* inputs can be made to appear at any of the *n* outputs, making this the most general-purpose combinational logic device available for *m* input pins and *n* output pins.

Also, PROMs (programmable ROMs), EPROMs (ultraviolet-erasable PROMs) and EEPROMs (electrically erasable PROMs) are available that can be programmed using a standard PROM programmer without requiring specialised hardware or software. However, there are several disadvantages:

- they are usually much slower than dedicated logic circuits,
- they cannot necessarily provide safe "covers" for asynchronous logic transitions so the PROM's outputs may glitch as the inputs switch,
- they consume more power[*clarification needed*],
- they are often more expensive than programmable logic, especially if high speed is required.

Since most ROMs do not have input or output registers, they cannot be used stand-alone for sequential logic. An external TTL register was often used for sequential designs such as state machines. Common EPROMs, for example the 2716, are still sometimes used in this way by hobby circuit designers, who often have some lying around. This use is sometimes called a 'poor man's PAL'.

Early programmable logic

In 1969, Motorola offered the XC157, a mask-programmed gate array with 12 gates and 30 uncommitted input/output pins.[1]

In 1970, Texas Instruments developed a mask-programmable IC based on the IBM read-only associative memory or ROAM. This device, the TMS2000, was programmed by altering the metal layer during the production of the IC. The TMS2000 had up to 17 inputs and 18 outputs with 8 JK flip flop for memory. TI coined the term programmable logic array for this device.[2]

In 1971, General Electric Company (GE) was developing a programmable logic device based on the new PROM technology. This experimental device improved on IBM's ROAM by allowing multilevel logic. Intel had just introduced the floating-gate UV erasable PROM so the researcher at GE incorporated that technology. The GE device was the first erasable PLD ever developed, predating the Altera EPLD by over a decade. GE obtained several early patents on programmable logic devices.[3][4][5]

In 1973 National Semiconductor introduced a mask-programmable PLA device (DM7575) with 14 inputs and 8 outputs with no memory registers. This was more popular than the TI part but cost of making the metal mask limited its use. The device is significant because it was the basis for the field programmable logic array produced by Signetics in 1975, the 82S100. (Intersil actually beat Signetics to market but poor yield doomed their part.)[6][7]

In 1974 GE entered into an agreement with Monolithic Memories to develop a mask-programmable logic device incorporating the GE innovations. The device was named the 'Programmable Associative Logic Array' or PALA. The MMI 5760 was completed in 1976 and could implement multilevel or sequential circuits of over 100 gates. The device was supported by a GE design environment where Boolean equations would be converted to mask patterns for configuring the device. The part was never brought to market.[8]

PLA

In 1970, Texas Instruments developed a mask-programmable IC based on the IBM read-only associative memory or ROAM. This device, the TMS2000, was programmed by altering the metal layer during the production of the IC. The TMS2000 had up to 17 inputs and 18 outputs with 8 JK flip flop for memory. TI coined the term programmable logic array for this device.[2]

A programmable logic array (PLA) has a programmable AND gate array, which links to a programmable OR gate array, which can then be conditionally complemented to produce an output.

PAL]

PAL devices have arrays of transistor cells arranged in a "fixed-OR, programmable-AND" plane used to implement "sum-of-products" binary logic equations for each of the outputs in terms of the inputs and either synchronous or asynchronous feedback from the outputs.

MMI introduced a breakthrough device in 1978, the programmable array logic or PAL. The architecture was simpler than that of Signetics FPLA because it omitted the programmable OR array. This made the parts faster, smaller and cheaper. They were available in 20 pin 300 mil DIP packages while the FPLAs came in 28 pin 600 mil packages. The PAL Handbook demystified the design process. The PALASM design software (PAL assembler) converted the engineers' Boolean equations into the fuse pattern required to program the part. The PAL devices were soon second-sourced by National Semiconductor, Texas Instruments and AMD.

After MMI succeeded with the 20-pin PAL parts, AMD introduced the 24-pin 22V10 PAL with additional features. After buying out MMI (1987), AMD spun off a consolidated operation as Vantis, and that business was acquired by Lattice Semiconductor in 1999.

How PLDs retain their configuration

A PLD is a combination of a logic device and a memory device. The memory is used to store the pattern that was given to the chip during programming. Most of the methods for storing data in an integrated circuit have been adapted for use in PLDs. These include:

- Silicon antifuses
- SRAM
- EPROM or EEPROM cells
- Flash memory

Silicon antifuses are connections that are made by applying a voltage across a modified area of silicon inside the chip. They are called antifuses because they work in the opposite way to normal fuses, which begin life as connections until they are broken by an electric current.

SRAM, or static RAM, is a volatile type of memory, meaning that its contents are lost each time the power is switched off. SRAM-based PLDs therefore have to be programmed every time the circuit is switched on. This is usually done automatically by another part of the circuit.

An EPROM cell is a MOS (metal-oxide-semiconductor) transistor that can be switched on by trapping an electric charge permanently on its gate electrode. This is done by a PAL

programmer. The charge remains for many years and can only be removed by exposing the chip to strong ultraviolet light in a device called an EPROM eraser.

Flash memory is non-volatile, retaining its contents even when the power is switched off. It can be erased and reprogrammed as required. This makes it useful for PLD memory.

As of 2005, most CPLDs are electrically programmable and erasable, and non-volatile. This is because they are too small to justify the inconvenience of programming internal SRAM cells every time they start up, and EPROM cells are more expensive due to their ceramic package with a quartz window.

PLD programming languages

Many PAL programming devices accept input in a standard file format, commonly referred to as 'JEDEC files'.They are analogous to software compilers. The languages used as source code for logic compilers are called hardware description languages, or HDLs.

PALASM, ABEL and CUPL are frequently used for low-complexity devices, while Verilog and VHDL are popular higher-level description languages for more complex devices. The more limited ABEL is often used for historical reasons, but for new designs VHDL is more popular, even for low-complexity designs.

For modern PLD programming languages, design flows, and tools, see FPGA and Reconfigurable computing.

PLD programming devices

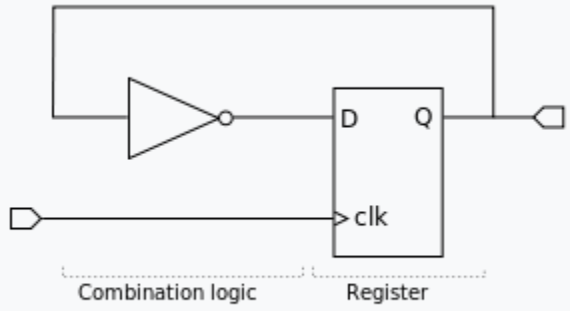
A device programmer is used to transfer the boolean logic pattern into the programmable device. In the early days of programmable logic, every PLD manufacturer also produced a specialized device programmer for its family of logic devices. Later, universal device programmers came onto the market that supported several logic device families from different manufacturers. Today's device programmers usually can program common PLDs (mostly PAL/GAL equivalents) from all existing manufacturers. Common file formats used to store the boolean logic pattern (fuses) are JEDEC, Altera POF (programmable object file), or Xilinx BITstream.[

UNIT V
VHDL

Register-transfer level

Register-transfer-level abstraction is used in hardware description languages (HDLs) like Verilog and VHDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. Design at the RTL level is typical practice in modern digital design

RTL description



Example of a simple circuit with the output toggling at each rising edge of the input. The inverter forms the combinational logic in this circuit, and the register holds the state.

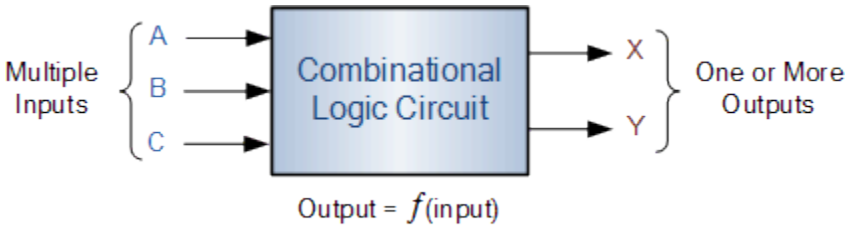
A synchronous circuit consists of two kinds of elements: registers and combinational logic. Registers (usually implemented as D flip-flops) synchronize the circuit's operation to the edges of the clock signal, and are the only elements in the circuit that have memory properties. Combinational logic performs all the logical functions in the circuit and it typically consists of logic gates.

For example, a very simple synchronous circuit is shown in the figure. The inverter is connected from the output, Q, of a register to the register's input, D, to create a circuit that changes its state on each rising edge of the clock, clk. In this circuit, the combinational logic consists of the inverter.

When designing digital integrated circuits with a hardware description language, the designs are usually engineered at a higher level of abstraction than transistor level (logic families) or logic gate level. In HDLs the designer declares the registers (which roughly correspond to variables in computer programming languages), and describes the combinational logic by using constructs that are familiar from programming languages such as if-then-else and arithmetic operations.

This level is called *register-transfer level*. The term refers to the fact that RTL focuses on describing the flow of signals between registers.

Combinational Logic Circuits

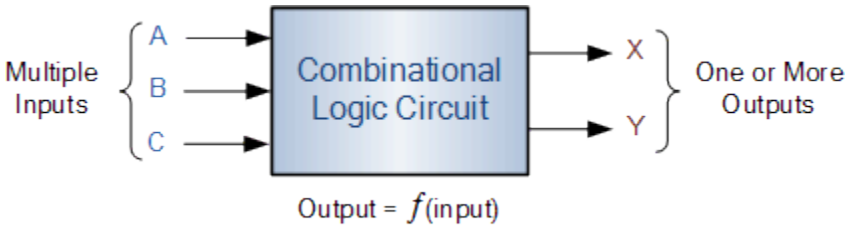


Unlike Sequential Logic Circuits whose outputs are dependant on both their present inputs and their previous output state giving them some form of **Memory**

The outputs of **Combinational Logic Circuits** are only determined by the logical function of their current input state, logic "0" or logic "1", at any given instant in time.

The result is that combinational logic circuits have no feedback, and any changes to the signals being applied to their inputs will immediately have an effect at the output. In other words, in a **Combinational Logic Circuit**, the output is dependant at all times on the combination of its inputs. So if one of its inputs condition changes state, from 0-1or 1-0, so too will the resulting output as by default combinational logic circuits have "no memory", "timing" or "feedback loops" within their design.

Combinational Logic



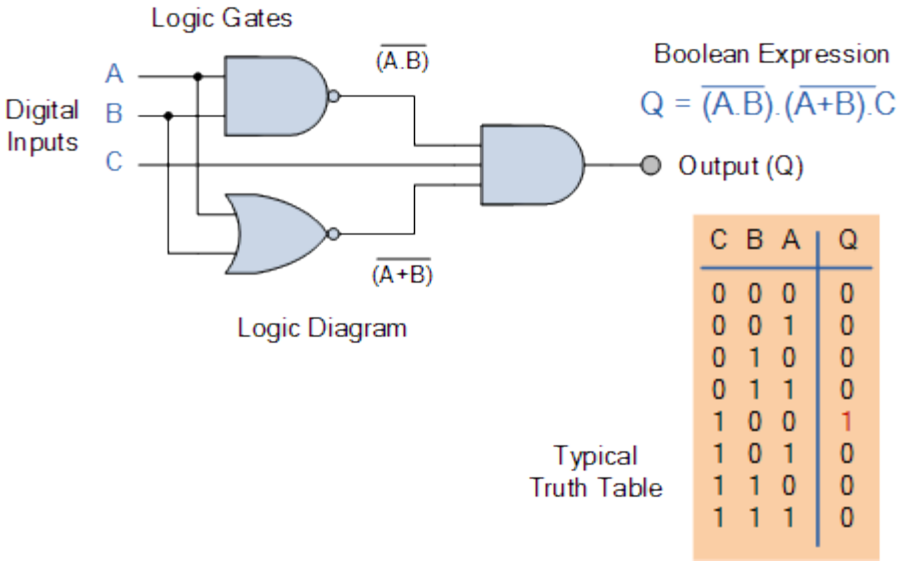
Combinational Logic Circuits are made up from basic logic **NAND**, **NOR** or **NOT** gates that are "combined" or connected together to produce more complicated switching circuits. These logic gates are the building blocks of combinational logic circuits. An example of a combinational circuit is a decoder, which converts the binary code data present at its input into a number of different output lines, one at a time producing an equivalent decimal code at its output.

Combinational logic circuits can be very simple or very complicated and any combinational circuit can be implemented with only NAND and NOR gates as these are classed as "universal" gates.

The three main ways of specifying the function of a combinational logic circuit are:

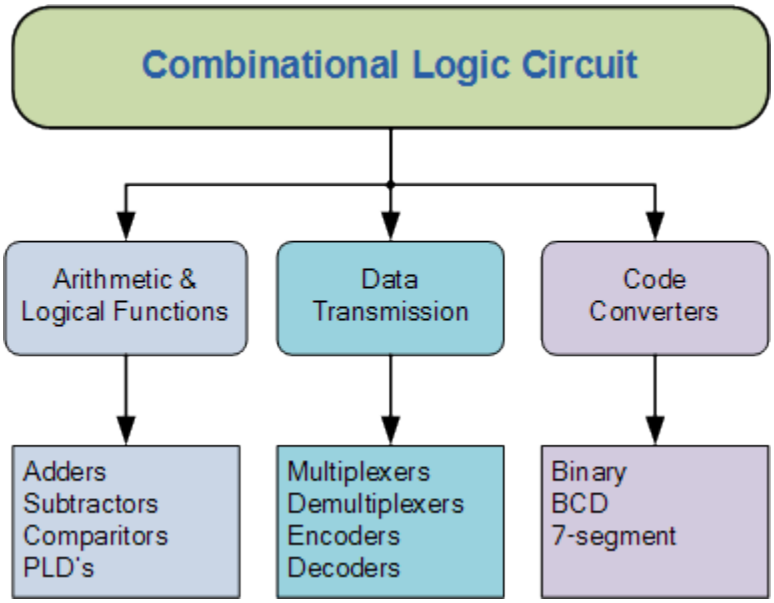
- 1. Boolean Algebra – This forms the algebraic expression showing the operation of the logic circuit for each input variable either True or False that results in a logic "1" output.
- 2. Truth Table – A truth table defines the function of a logic gate by providing a concise list that shows all the output states in tabular form for each possible combination of input variable that the gate could encounter.
- 3. Logic Diagram – This is a graphical representation of a logic circuit that shows the wiring and connections of each individual logic gate, represented by a specific graphical symbol, that implements the logic circuit.

and all three of these logic circuit representations are shown below.



As combinational logic circuits are made up from individual logic gates only, they can also be considered as "decision making circuits" and combinational logic is about combining logic gates together to process two or more signals in order to produce at least one output signal according to the logical function of each logic gate. Common combinational circuits made up from individual logic gates that carry out a desired application include **Multiplexers**, **De-multiplexers**, **Encoders**, **Decoders**, **Full** and **Half Adders** etc.

Classification of Combinational Logic



One of the most common uses of combinational logic is in **Multiplexer** and **De-multiplexer** type circuits. Here, multiple inputs or outputs are connected to a common signal line and logic gates are used to decode an address to select a single data input or output switch.

A multiplexer consist of two separate components, a logic decoder and some solid state switches, but before we can discuss multiplexers, decoders and de-multiplexers in more detail we first need to understand how these devices use these "solid state switches" in their design.

Solid State Switches

Standard TTL logic devices made up from **Transistors** can only pass signal currents in one direction only making them "uni-directional" devices and poor imitations of conventional electro-mechanical switches or relays. However, some CMOS switching devices made up from **FET's** act as near perfect "bi-directional" switches making them ideal for use as solid state switches.

Solid state switches come in a variety of different types and ratings, and there are many different applications for using solid state switches. They can basically be sub-divided into 3 different main groups for switching applications and in this combinational logic section we will only look at the **Analogue** type of switch but the principal is the same for all types including digital.

Solid State Switch Applications

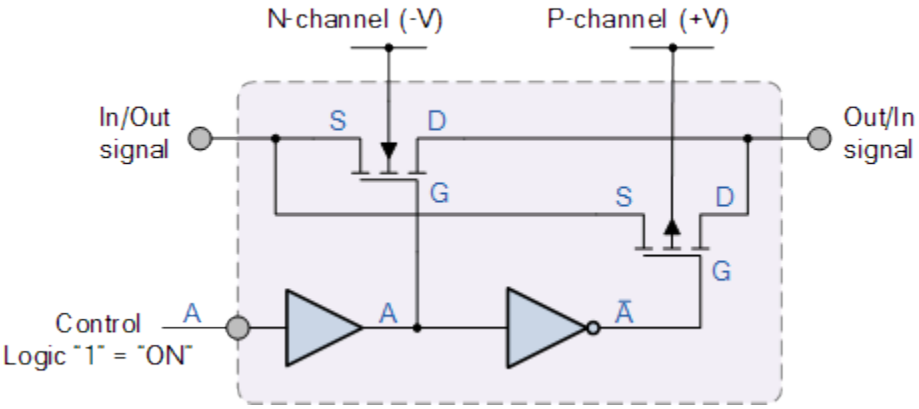
- Analogue Switches – Used in Data Switching and Communications, Video and Audio Signal Switching, Instrumentation and Process Control Circuits …etc.
- Digital Switches – High Speed Data Transmission, Switching and Signal Routing, Ethernet, LAN's, USB and Serial Transmissions …etc.

- Power Switches – Power Supplies and General “Standby Power” Switching Applications, Switching of Larger Voltages and Currents …etc.

Analogue Bilateral Switches

Analogue or “Analog” switches are those types that are used to switch data or signal currents when they are in their “ON” state and block them when they are in their “OFF” state. The rapid switching between the “ON” and the “OFF” state is usually controlled by a digital signal applied to the control gate of the switch. An ideal analogue switch has zero resistance when “ON” (or closed), and infinite resistance when “OFF” (or open) and switches with R_{ON} values of less than 1Ω are commonly available.

Solid State Analogue Switch

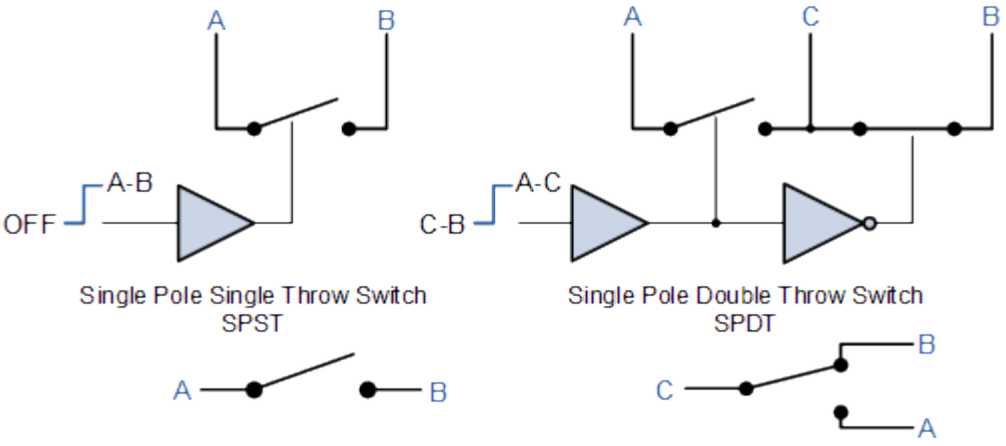


By connecting an N-channel MOSFET in parallel with a P-channel MOSFET allows signals to pass in either direction making it a **Bi-directional** switch and as to whether the N-channel or the P-channel device carries more signal current will depend upon the ratio between the input to the output voltage. The two MOSFET’s are switched “ON” or “OFF” by two internal non-inverting and inverting amplifiers.

Contact Types

Just like mechanical switches, analogue switches come in a variety of forms or contact types, depending on the number of “poles” and “throws” they offer. Thus, terms such as “SPST” (single-pole single throw) and “SPDT” (single-pole double-throw) also apply to solid state analogue switches with “make-before-break” and “break-before-make” configurations available.

Analogue Switch Types



Individual analogue switches can be grouped together into standard IC packages to form devices with multiple switching configurations of SPST (single-pole single-throw) and SPDT (single-pole double-throw) as well as multi channel multiplexers.

The most common and simplest analogue switch in a single IC package is the 74HC4066 which has 4 independent bi-directional "ON/OFF" Switches within a single package but the most widely used variants of the CMOS analogue switch are those described as "Multi-way Bilateral Switches" otherwise known as the "Multiplexer" and "De-multiplexer" IC´s and these are discussed in the next tutorial.

Combinational Logic Summary

Then to summarise, **Combinational Logic Circuits** consist of inputs, two or more basic logic gates and outputs. The logic gates are combined in such a way that the output state depends entirely on the input states. Combinational logic circuits have "no memory", "timing" or "feedback loops", there operation is instantaneous. A combinational logic circuit performs an operation assigned logically by a Boolean expression or truth table.

Examples of common combinational logic circuits include: half adders, full adders, multiplexers, demultiplexers, encoders and decoders all of which we will look at in the next few tutorials.

Sequential Logic Circuits

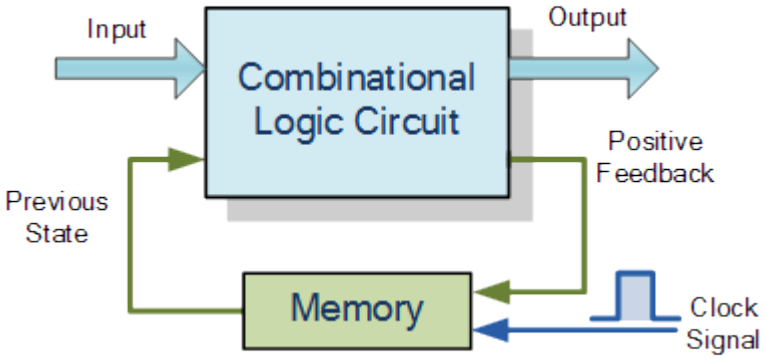
This means that sequential logic circuits are able to take into account their previous input state as well as those actually present, a sort of "before" and "after" effect is involved with sequential circuits.

In other words, the output state of a "sequential logic circuit" is a function of the following three states, the "present input", the "past input" and/or the "past output". *Sequential Logic*

circuits remember these conditions and stay fixed in their current state until the next clock signal changes one of the states, giving sequential logic circuits "Memory".

Sequential logic circuits are generally termed as *two state* or **Bistable** devices which can have their output or outputs set in one of two basic states, a logic level "1" or a logic level "0" and will remain "latched" (hence the name latch) indefinitely in this current state or condition until some other input trigger pulse or signal is applied which will cause the bistable to change its state once again.

Sequential Logic Representation



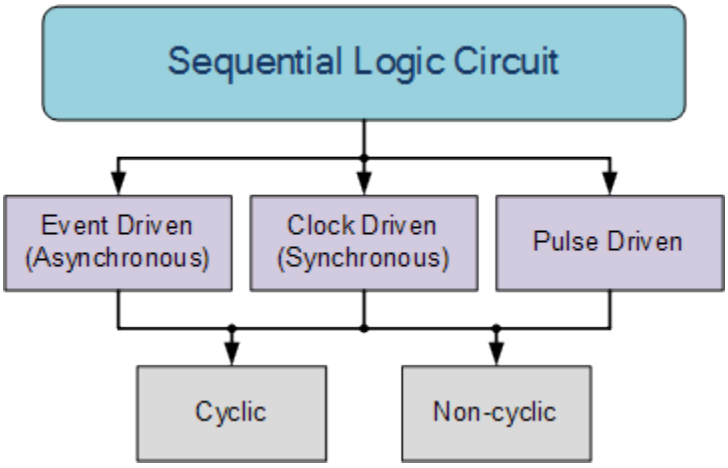
The word "Sequential" means that things happen in a "sequence", one after another and in **Sequential Logic** circuits, the actual clock signal determines when things will happen next. Simple sequential logic circuits can be constructed from standard **Bistable**circuits such as: Flip-flops, Latches and Counters and which themselves can be made by simply connecting together universal **NAND Gates** and/or **NOR Gates** in a particular combinational way to produce the required sequential circuit.

Related Products: **Direct Digital Synthesizer**

Classification of Sequential Logic

As standard logic gates are the building blocks of combinational circuits, bistable latches and flip-flops are the basic building blocks of sequential logic circuits. Sequential logic circuits can be constructed to produce either simple edge-triggered flip-flops or more complex sequential circuits such as storage registers, shift registers, memory devices or counters. Either way sequential logic circuits can be divided into the following three main categories:

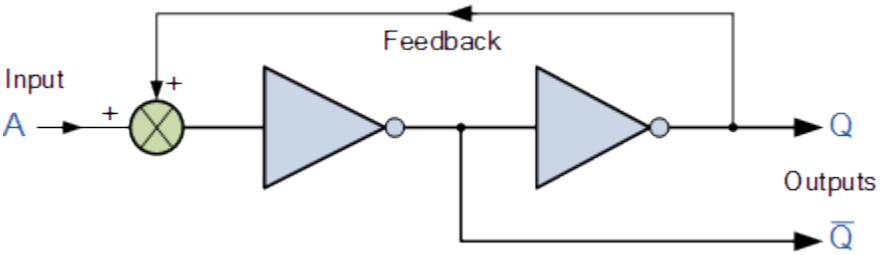
- 1. Event Driven – asynchronous circuits that change state immediately when enabled.
- 2. Clock Driven – synchronous circuits that are synchronised to a specific clock signal.
- 3. Pulse Driven – which is a combination of the two that responds to triggering pulses.



As well as the two logic states mentioned above logic level "1" and logic level "0", a third element is introduced that separates **sequential logic** circuits from their **combinational logic** counterparts, namely *TIME*. Sequential logic circuits return back to their original steady state once reset and sequential circuits with loops or feedback paths are said to be "cyclic" in nature.

We now know that in sequential circuits changes occur only on the application of a clock signal making it synchronous, otherwise the circuit is asynchronous and depends upon an external input. To retain their current state, sequential circuits rely on feedback and this occurs when a fraction of the output is fed back to the input and this is demonstrated as:

Sequential Feedback Loop



The two inverters or NOT gates are connected in series with the output at Q fed back to the input. Unfortunately, this configuration never changes state because the output will always be the same, either a "1" or a "0", it is permanently set. However, we can see how feedback works by examining the most basic sequential logic components, called the SR flip-flop.

SR Flip-Flop

The **SR flip-flop**, also known as a *SR Latch*, can be considered as one of the most basic sequential logic circuit possible. This simple flip-flop is basically a one-bit memory bistable device that has two inputs, one which will "SET" the device (meaning the output = "1"), and is labelled S and another which will "RESET" the device (meaning the output = "0"), labelled R.

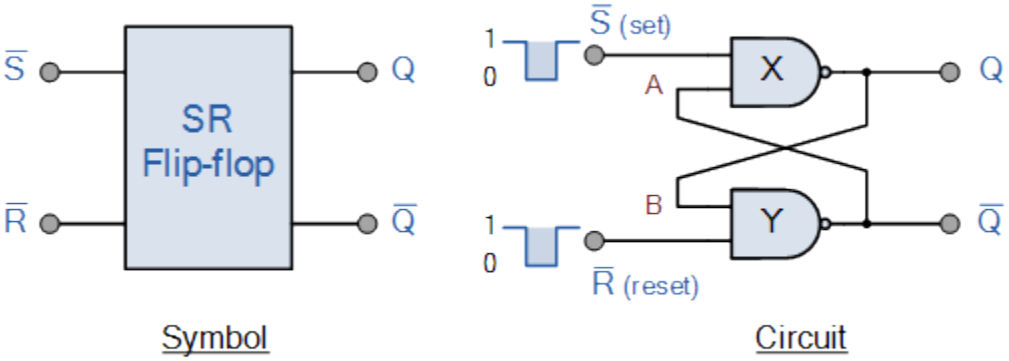
Then the SR description stands for “Set-Reset”. The reset input resets the flip-flop back to its original state with an output Q that will be either at a logic level “1” or logic “0” depending upon this set/reset condition.

A basic NAND gate SR flip-flop circuit provides feedback from both of its outputs back to its opposing inputs and is commonly used in memory circuits to store a single data bit. Then the SR flip-flop actually has three inputs, Set, Reset and its current output Qrelating to it’s current state or history. The term “Flip-flop” relates to the actual operation of the device, as it can be “flipped” into one logic Set state or “flopped” back into the opposing logic Reset state.

The NAND Gate SR Flip-Flop

The simplest way to make any basic single bit set-reset SR flip-flop is to connect together a pair of cross-coupled 2-input NAND gates as shown, to form a Set-Reset Bistable also known as an active LOW SR NAND Gate Latch, so that there is feedback from each output to one of the other NAND gate inputs. This device consists of two inputs, one called the *Set*, S and the other called the *Reset*, R with two corresponding outputs Q and its inverse or complement Q (not-Q) as shown below.

The Basic SR Flip-flop



The Set State

Consider the circuit shown above. If the input R is at logic level “0” (R = 0) and input S is at logic level “1” (S = 1), the NAND gate *Y* has at least one of its inputs at logic “0” therefore, its output Q must be at a logic level “1” (NAND Gate principles). Output Q is also fed back to input “A” and so both inputs to NAND gate *X* are at logic level “1”, and therefore its output Q must be at logic level “0”.

Again NAND gate principals. If the reset input R changes state, and goes HIGH to logic “1” with S remaining HIGH also at logic level “1”, NAND gate *Y* inputs are now R = “1” and B = “0”. Since one of its inputs is still at logic level “0” the output at Q still remains HIGH at logic level “1” and there is no change of state. Therefore, the flip-flop circuit is said to be “Latched” or “Set” with Q = “1” and Q = “0”.

Reset State

In this second stable state, Q is at logic level "0", (not Q = "0") its inverse output at Q is at logic level "1", (Q = "1"), and is given by R = "1" and S = "0". As gate *X* has one of its inputs at logic "0" its output Q must equal logic level "1" (again NAND gate principles). Output Q is fed back to input "B", so both inputs to NAND gate *Y* are at logic "1", therefore, Q = "0".

If the set input, S now changes state to logic "1" with input R remaining at logic "1", output Q still remains LOW at logic level "0" and there is no change of state. Therefore, the flip-flop circuits "Reset" state has also been latched and we can define this "set/reset" action in the following truth table.

Truth Table for this Set-Reset Function

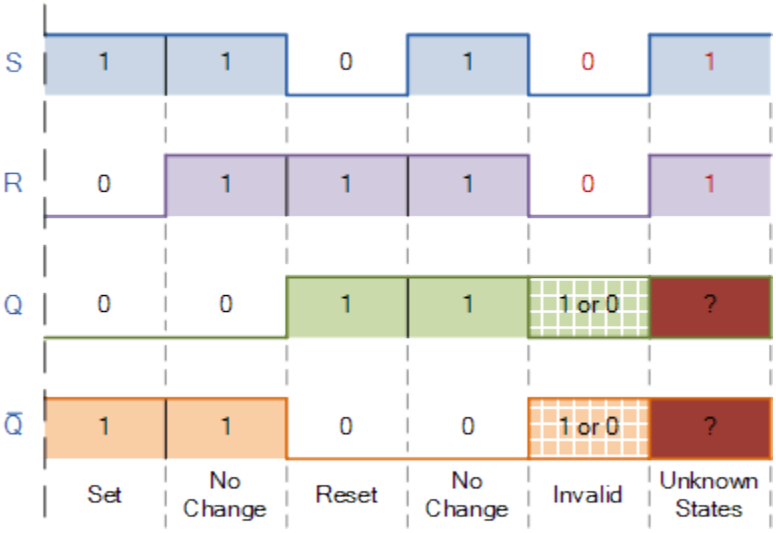
State	S	R	Q	Q	Description
Set	1	0	0	1	Set Q » 1
	1	1	0	1	no change
Reset	0	1	1	0	Reset Q » 0
	1	1	1	0	no change
Invalid	0	0	1	1	Invalid Condition

It can be seen that when both inputs S = "1" and R = "1" the outputs Q and Q can be at either logic level "1" or "0", depending upon the state of the inputs S or R BEFORE this input condition existed. Therefore the condition of S = R = "1" does not change the state of the outputs Q and Q.

However, the input state of S = "0" and R = "0" is an undesirable or invalid condition and must be avoided. The condition of S = R = "0" causes both outputs Q and Q to be HIGH together at logic level "1" when we would normally want Q to be the inverse of Q. The result is that the flip-

flop looses control of Q and Q, and if the two inputs are now switched "HIGH" again after this condition to logic "1", the flip-flop becomes unstable and switches to an unknown data state based upon the unbalance as shown in the following switching diagram.

S-R Flip-flop Switching Diagram

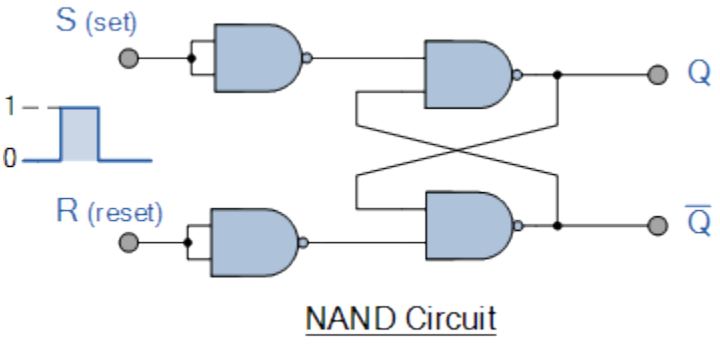


This unbalance can cause one of the outputs to switch faster than the other resulting in the flip-flop switching to one state or the other which may not be the required state and data corruption will exist. This unstable condition is generally known as its **Meta-stable**state.

Then, a simple NAND gate SR flip-flop or NAND gate SR latch can be set by applying a logic "0", (LOW) condition to its Set input and reset again by then applying a logic "0" to its Reset input. The SR flip-flop is said to be in an "invalid" condition (Meta-stable) if both the set and reset inputs are activated simultaneously.

As we have seen above, the basic NAND gate SR flip-flop requires logic "0" inputs to flip or change state from Q to Q and vice versa. We can however, change this basic flip-flop circuit to one that changes state by the application of positive going input signals with the addition of two extra NAND gates connected as inverters to the S and R inputs as shown.

Positive NAND Gate SR Flip-flop

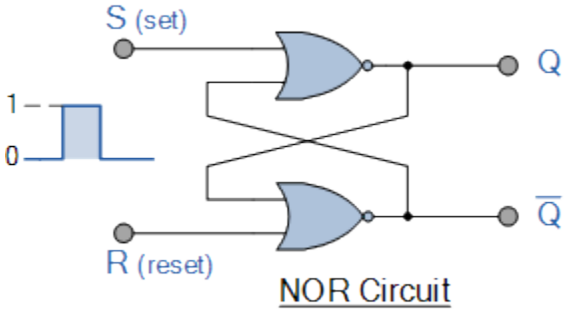


NAND Circuit

S	R	Q	Q̄
0	0	No change	
0	1	0	1
1	0	1	0
1	1	X (Invalid)	X

As well as using NAND gates, it is also possible to construct simple one-bit **SR Flip-flops** using two cross-coupled NOR gates connected in the same configuration. The circuit will work in a similar way to the NAND gate circuit above, except that the inputs are active HIGH and the invalid condition exists when both its inputs are at logic level "1", and this is shown below.

The NOR Gate SR Flip-flop



NOR Circuit

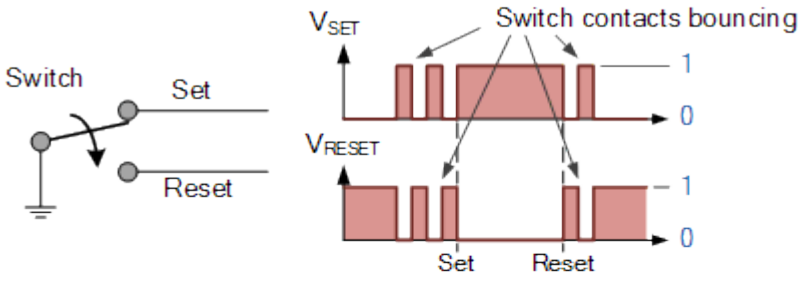
S	R	Q	$\overline{Q}$
0	0	No change	
0	1	1	0
1	0	0	1
1	1	X	X
		(Invalid)	

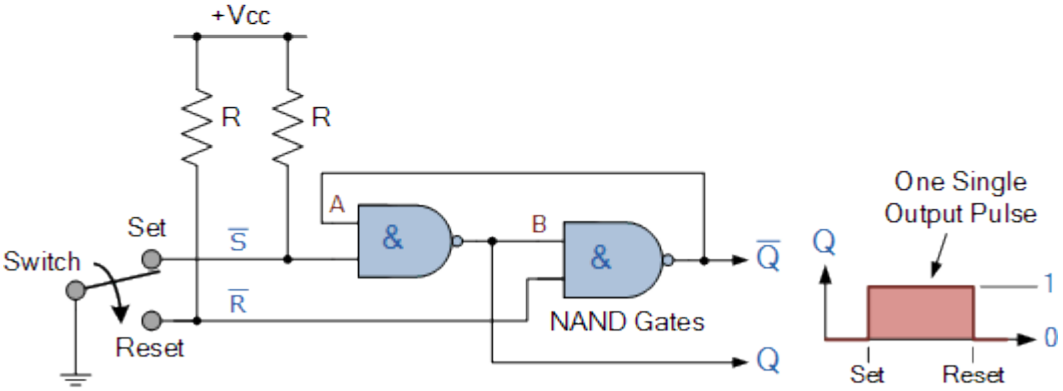
Switch Debounce Circuits

Edge-triggered flip-flops require a nice clean signal transition, and one practical use of this type of set-reset circuit is as a latch used to help eliminate mechanical switch "bounce". As its name implies, switch bounce occurs when the contacts of any mechanically operated switch, push-button or keypad are operated and the internal switch contacts do not fully close cleanly, but bounce together first before closing (or opening) when the switch is pressed.

This gives rise to a series of individual pulses which can be as long as tens of milliseconds that an electronic system or circuit such as a digital counter may see as a series of logic pulses instead of one long single pulse and behave incorrectly. For example, during this bounce period the output voltage can fluctuate wildly and may register multiple input counts instead of one single count. Then set-reset SR Flip-flops or Bistable Latch circuits can be used to eliminate this kind of problem and this is demonstrated below.

SR Flip Flop Switch Debounce Circuit





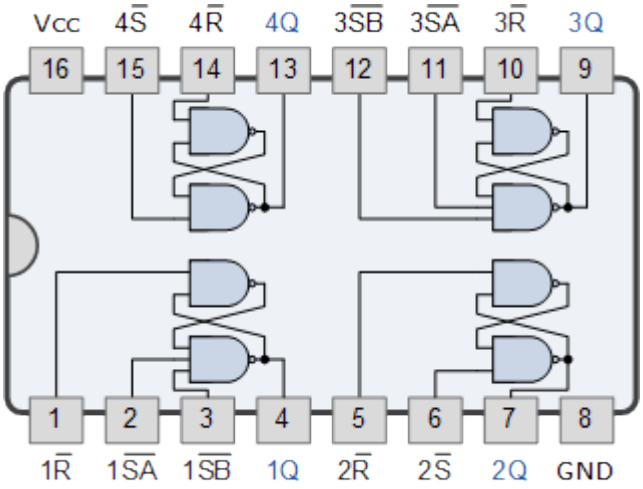
Depending upon the current state of the output, if the set or reset buttons are depressed the output will change over in the manner described above and any additional unwanted inputs (bounces) from the mechanical action of the switch will have no effect on the output at Q.

When the other button is pressed, the very first contact will cause the latch to change state, but any additional mechanical switch bounces will also have no effect. The SR flip-flop can then be RESET automatically after a short period of time, for example 0.5 seconds, so as to register any additional and intentional repeat inputs from the same switch contacts, such as multiple inputs from a keyboards "RETURN" key.

Commonly available IC's specifically made to overcome the problem of switch bounce are the MAX6816, single input, MAX6817, dual input and the MAX6818 octal input switch debouncer IC's. These chips contain the necessary flip-flop circuitry to provide clean interfacing of mechanical switches to digital systems.

Set-Reset bistable latches can also be used as Monostable (one-shot) pulse generators to generate a single output pulse, either high or low, of some specified width or time period for timing or control purposes. The 74LS279 is a Quad SR Bistable Latch IC, which contains four individual NAND type bistable's within a single chip enabling switch debounce or monostable/astable clock circuits to be easily constructed.

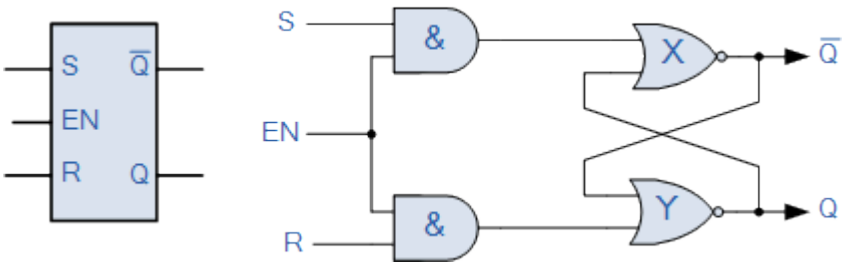
Quad SR Bistable Latch 74LS279



Gated or Clocked SR Flip-Flop

It is sometimes desirable in sequential logic circuits to have a bistable SR flip-flop that only changes state when certain conditions are met regardless of the condition of either the Set or the Reset inputs. By connecting a 2-input AND gate in series with each input terminal of the SR Flip-flop a Gated SR Flip-flop can be created. This extra conditional input is called an "Enable" input and is given the prefix of "EN". The addition of this input means that the output at Q only changes state when it is HIGH and can therefore be used as a clock (CLK) input making it level-sensitive as shown below.

Gated SR Flip-flop



When the Enable input "EN" is at logic level "0", the outputs of the two AND gates are also at logic level "0", (AND Gate principles) regardless of the condition of the two inputs S and R, latching the two outputs Q and Q into their last known state. When the enable input "EN" changes to logic level "1" the circuit responds as a normal SR bistable flip-flop with the two AND gates becoming transparent to the Set and Reset signals.

This additional enable input can also be connected to a clock timing signal (CLK) adding clock synchronisation to the flip-flop creating what is sometimes called a "Clocked SR Flip-flop". So a **Gated Bistable SR Flip-flop** operates as a standard bistable latch but the outputs are only activated when a logic "1" is applied to its EN input and deactivated by a logic "0".

In the next tutorial about **Sequential Logic Circuits**, we will look at another type of simple edge-triggered flip-flop which is very similar to the **RS flip-flop** called a **JK Flip-flop** named

after its inventor, Jack Kilby. The JK flip-flop is the most widely used of all the flip-flop designs as it is considered to be a universal device.

1 VHDL Sub-Programs, Packages, & Libraries EL 310 Erkay Savaş Sabancı University 2 Motivation • Structuring VHDL programs – modularity • Design reuse • Manage the complexity • Available VHDL tools: – procedures, – functions, – packages, – libraries 3 Issues • VHDL programs model physical systems • There may have some issues we have to deal with such as: • Can wait statements be used in a procedure? • Can signals be passed to procedures and be modified within the procedure? • How are procedures synthesized? • Can functions operate on signals? 4 Packages & Libraries • Groups of procedures and functions that are related can be aggregated into a module that is called package. • A package can be shared across many VHDL models. • A package can also contains user defined data types and constants. • A library is a collection of related packages. • Packages and libraries serve as repositories for functions, procedures, and data types. 5 Functions • A function computes and returns a value of specified type using the input parameters. • Function declaration: – function rising_edge(signal clock: in std_logic) return Boolean; • Parameters are used but not modified within the function. – the mode of input parameters is in. – functions do not have parameters of mode out. – In fact, we do not have to specify the mode. 6 Function Definition function rising_edge(signal clock: std_logic) return Boolean is -- -- declarative region: declare variables local to the function -- begin -- -- body -- return (value); end function rising_edge; formal parameters – The function is called with the actual parameters. – Example: rising_edge(enable); – Types of formal and actual parameters must match. – Actual parameters could be variable, signal, constant or an expression. 7 Functions • When no class is specified, parameters are by default constants. • wait statements cannot be used in functions' body. – functions execute in zero simulation time. • Functions cannot call a procedure that has a wait statement in it. • Parameters have to be mode in. – signals passed as parameters cannot be assigned values in the function body. 8 Pure vs. Impure Functions • VHDL'93 supports two distinct types of functions: • Pure functions – always return the same value when called with the same parameter values. • Impure functions – may return different values even if they are called with the same parameter values at different times. – All the signals in the architecture is visible within the function body – Those signals may not appear in the function parameter list (e.g. ports of the entity) 9 Example library IEEE; use IEEE.std_logic_1164.all; entity dff is port(d, clock: in std_logic; q, qbar: out std_logic); end entity dff; architecture beh of dff is function rising_edge(signal clock:std_logic) return Boolean is variable edge: Boolean:=FALSE; begin edge:= (clock = '1' and clock'event); return (edge); end function rising_edge; begin output: process is begin wait until (rising_edge(clock)); q <= d after 5 ns; qbar <= not d after 5 ns; end process output; end architecture beh; declarative region of architecture 10 Type Conversion Functions function to_bitvector(svalue: std_logic_vector) return bit_vector is variable outvalue: bit_vector(svalue'length-1 downto 0); begin for i in svalue'range loop –- scan all elements of the array case svalue(i) is when '0' => outvalue(i) := '0'; when '1' => outvalue(i) := '1'; when others => outvalue(i) := '0'; end case; end loop; return outvalue; end function to_bitvector; Unconstrained array 11 std_logic_arith type UNSIGNED is array (NATURAL range <>) of STD_LOGIC; type tbl_type is array (STD_ULOGIC) of STD_ULOGIC; constant tbl_BINARY : tbl_type := ('X', 'X', '0', '1', 'X', 'X', '0', '1', 'X'); function CONV_INTEGER(ARG: UNSIGNED) return INTEGER is variable result: INTEGER; variable tmp: STD_ULOGIC; -- synopsys built_in SYN_UNSIGNED_TO_INTEGER -- synopsys

subpgm_id 366 begin -- synopsys synthesis_off assert ARG'length <= 31 report "ARG is too large in CONV_INTEGER" severity FAILURE; result := 0; 12 std_logic_arith for i in ARG'range loop result := result 2; tmp := tbl_BINARY(ARG(i)); if tmp = '1' then result := result + 1; elsif tmp = 'X' then assert false report "There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es)." severity warning; assert false report "CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, and it has been converted to 0." severity WARNING; return 0; end if; end loop; return result; -- synopsys synthesis_on end; 13 std_logic_arith function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT is variable tmp: STD_ULOGIC; -- synopsys built_in SYN_FEED_THRU -- synopsys subpgm_id 370 begin -- synopsys synthesis_off tmp := tbl_BINARY(ARG); if tmp = '1' then return 1; elsif tmp = 'X' then assert false report "CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, and it has been converted to 0." severity WARNING; return 0; else return 0; end if; -- synopsys synthesis_on end; 14 Resolution Functions • For system busses, we use resolved signals CPU MEM Disk I/O 15 Resolution Functions • Wired-logic implementations Z weak pull up device X1 0 X2 0 Xn 0 Any Xi = 0 turns the switch on driving Z=0 switch with active low input 16 Resolution Functions entity wired_and is port(X1, X2, X3: in bit; Z : out resolved_bit); end entity; architecture beh of wired_and is begin x_process: process(X1) is begin Z <= X1; end process; y_process: process(X2) is begin Z <= X2; end process; z_process: process(X3) is begin Z <= X3; end process; end architecture beh; 17 Resolved Types in the IEEE 1164 Standard type std_ulogic is (‘U’, -- uninitialized ‘X’, -- forcing unknown ‘0’, -- forcing 0 ‘1’, -- forcing 1 ‘Z’, -- high impedance ‘W’, -- weak unknown ‘L’, -- weak 0 ‘H’, -- weak 1 ‘-’, -- don’t care); function resolved (s: std_ulogic_vector) return std_ulogic; subtype std_logic is resolved std_ulogic; Resolution function must perform an associative operations so that the order in which the multiple signal drivers are examined does not affect the resolved value of the signal. 18 Resolution Table for std_logic - U X X X X X X X X H U X 0 1 H W W H X L U X 0 1 L W L W X W U X 0 1 W W W W X Z U X 0 1 Z W L H X 1 U X X 1 1 1 1 1 X 0 U X 0 X 0 0 0 0 X X U X X X X X X X U U U U U U U U U U U X 0 1 Z W L H - 19 Example: Multiple Drivers entity resol is end entity; architecture beh of resol is signal X, Y, Z, T: std_logic; begin X <= 'U', '1' after 10 ns, '0' after 20 ns; Y <= '1', 'W' after 10 ns, '-' after 20 ns; Z <= 'W', 'Z' after 10 ns, 'L' after 20 ns; x_process: process(X) is begin T <= X; end process; y_process: process(Y) is begin T <= Y; end process; z_process: process(Z) is begin T <= Z; end process; end architecture beh; 20 Example: Waveforms 21 Resolution Function architecture beh of resol02 is function resolution (drivers: std_ulogic_vector) return std_logic is type my_array_type is array (std_logic, std_logic) of STD_ULOGIC; constant my_array : my_array_type := (('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'), ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X'), ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X'), ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X'), ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X'), ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X'), ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'), ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X')); variable tmp: std_ulogic; begin tmp := my_array(drivers(drivers'length-1),drivers(drivers'length-2)); for i in drivers'length-3 downto 0 loop tmp := my_array(tmp, drivers(i)); end loop; return tmp; end function; 22 Resolution Function signal Z: std_ulogic; begin Z <= resolution("1HWW") after 20 ns; end architecture beh; 23 Example: Resolution Functions • Multi-chip module with multiple die chip carrier global error signal die 24 Example: Resolution Functions library IEEE; use IEEE.std_logic_1164.all; entity mcm is –- and empty entity declaration end entity mcm; architecture beh of mcm is function wire_or(sbus:std_ulogic_vector) return std_ulogic is begin for i in sbus’range loop if sbus(i) = ‘1’ then return ‘1’; end if; end loop; return ‘0’; end function wire_or; subtype wire_or_logic is

wire_or std_ulogic; signal error_bus: wire_or_logic; ... resolution function base type 25 Example: Resolution Functions ... subtype wire_or_logic is wire_or std_ulogic; signal error_bus: wire_or_logic; begin chip1: process is begin ... error_bus <= '1' after 2 ns; ... end process chip1; chip2: process is begin ... error_bus <= '0' after 2 ns; ... end process chip2; end architecture beh; 26 Synthesis Example entity func is port(data: in std_logic_vector(7 downto 0); count: out integer range 0 to 7); end entity func; architecture beh of func is function ones(signal data: std_logic_vector) return integer is variable count: integer range 0 to 7; begin for i in data'range loop if data(i) = '1' then count:= count + 1; end if; end loop; return (count); end function ones; begin check: process(data) is begin count <= ones(data); end process check; end architecture beh; 27 Synthesis Example (cont.) • count <= ones(data); • each time this function is called, internal variables are initialized, the function is executed, and output value is returned. • Internal variables do not retain value across different invocations. • The loop is unrolled if data(0) = '1' then count := count + 1; end if; if data(1) = '1' then count := count + 1; end if; ... • When synthesized, it will produce a combinational circuit with a long dependency. 28 Synthesis Example (cont.) 29 Yet Another Example entity func02 is port(data: in std_logic_vector(7 downto 0); parity: out std_logic); end entity func02; architecture beh of func02 is function parity_func(signal data: std_logic_vector) return std_logic is variable parity: std_logic:='0'; begin for i in data'range loop parity := parity xor data(i); end loop; return (parity); end function parity_func; begin parity_process: process(data) is begin parity <= parity_func(data); end process check; end architecture beh; 30 Yet Another Example 31 Procedures • Similar to functions • distinguishing feature is that procedures can modify input parameters. • Syntax: – procedure read_v1d(file fname: in text; v: out std_logic_vector); – a procedure that reads data from a file called fname. • Parameters can be of in, out, inout modes. – default class is constant for input parameters. – default class is variable for out and inout mode parameters. 32 Procedures • Variables declared within a procedure are initialized in each call • Variables do not retain values across procedure calls. • Example: entity CPU is port(write_data: out std_logic_vector(31 downto 0); ADDR: out std_logic_vector(2 downto 0); MemRead, MemWrite: out std_logic; read_data: in std_logic_vector(31 downto 0); S: in std_logic); end entity CPU; 33 Example: CPU + Memory architecture beh of CPU is procedure mread(address: in std_logic_vector(2 downto 0); signal R: out std_logic; signal S: in std_logic; signal ADDR: out std_logic_vector(2 downto 0); signal data: out std_logic_vector(31 downto 0)) is begin ADDR <= address; R <= '1'; wait until S = '1'; data <= read_data; R <= '0'; end procedure mread; ... end architecture beh; 34 Example: CPU + Memory architecture beh of CPU is ... procedure mwrite(address: in std_logic_vector(2 downto 0); signal data: in std_logic_vector(31 downto 0); signal ADDR: out std_logic_vector(2 downto 0); signal W: out std_logic; signal DO: out std_logic_vector(31 downto 0)) is begin ADDR <= address; DO <= data; W <= '1'; wait until S = '1'; W <= '0'; end procedure mwrite; ... end architecture beh; 35 Example: CPU + Memory architecture beh of CPU is ... -- CPU description here begin process is begin -- -- behavioral description -- end process; process is begin -- -- behavioral description -- end process; end architecture beh; 36 Using Procedures • Signals can be passed to procedures and updated within procedures • Signals cannot be declared within procedures • Visibility rules apply here – procedure can update signals visible to it even if these signals do not appear in the parameter list. – This is sometimes called as side effect of procedure. – updating signals that do not appear in the parameter list is a poor programming practice. • A process calling a procedure with a wait statement cannot have sensitivity list 37 Concurrent vs. Sequential Procedure Calls • Concurrent procedure call – procedure calls can be made in concurrent signal assignments – execution of a procedure is

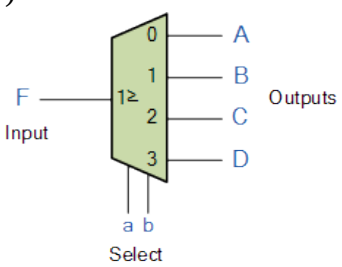
concurrent with other concurrent procedures, CSA statements, or processes. – The procedure is invoked when there is an event on a signal that is a parameter to the procedure. – Parameter list cannot include a variable in a concurrent procedure call (shared variables can be included though). • Sequential procedure call – executed within the body of a process. – executions is determined by the order. 38 Example: Concurrent Procedure Call entity serial_adder is port(a, b, clk, reset: in std_logic; z: out std_logic); end entity; architecture structural of serial adder is component comb is port(a, b, c_in: in std_logic; z, carry: out std_logic); end component; procedure dff(signal d, clk, reset: in std_logic; signal q, qbar: out std_logic) is begin if(reset =‘0’) then q <= ‘0’ after 5 ns; qbar <= ‘1’ after 5 ns; elsif(rising_edge(clk)) then q <= d after 5 ns; qbar <= not d after 5 ns; end if; end procedure; signal s1, s2: std_logic; begin C1: comb port(a=>a, b=>b, c_in=>s1, z=>z, carry=>s2); dff(clk=>clk, reset=>reset, d=>s2, q=>s1, qbar=>open); end architecture structural; behavioral description explicitly associating formal and actual parameters 39 Example: Sequential Procedure Call entity serial_adder is port(a, b, clk, reset: in std_logic; z: out std_logic); end entity; architecture structural of serial adder is component comb is port(a, b, c_in: in std_logic; z, carry: out std_logic); end component; procedure dff(signal d, clk, reset: in std_logic; signal q, qbar: out std_logic) is ... end procedure; signal s1, s2: std_logic; begin C1: comb port(a=>a, b=>b, c_in=>s1, z=>z, carry=>s2); process begin dff(clk=>clk, reset=>reset, d=>s2, q=>s1, qbar=>open); wait on clk, reset, s2; end process; end architecture structural; 40 Synthesis & Procedures • In-lining approach – during synthesis, the compiler replaces the procedure calls with the corresponding code (flattening) – therefore, inferencing techniques are applicable • Latch inference – local variables do not retain value across invocations; they will be synthesized into wires – signals of mode out may infer latches. – for example, if procedure resides in a conditional block of code, latches will be inferred for output signals of the procedure 41 Synthesis & Procedures • Wait statements – recall that synthesis compilers allows only one wait statement in a process. – procedures that are being called within a process that has already a wait statement cannot have a wait statement of its own. – Therefore, wait statements are generally not supported in procedures for synthesis. • For synchronous logic – use if statements in the procedure 42 Example entity proc_call is port(data:in std_logic_vector(7 downto 0); count: out std_logic); end entity proc_call; architecture beh of proc_call is procedure ones(signal data: in std_logic_vector; signal count: out std_logic) is variable parity: std_logic:=‘0’; begin for i in data’range loop parity := parity xor data(i); end loop; count <= parity; end procedure ones; begin ones(data, count); -- concurrent procedure call end architecture beh; 43 Example 44 Yet Another Example entity proc_call is port(reset, clock: in std_logic; data:in std_logic_vector(7 downto 0); count: out std_logic); end entity proc_call; architecture beh of proc_call is procedure ones(signal data: in std_logic_vector; signal count: out std_logic) is variable parity: std_logic:=‘0’; begin for i in data’range loop parity := parity xor data(i); end loop; count <= parity; end procedure ones; begin process(reset, clock) is begin if reset = ‘0’ and rising_edge(clock) then ones(data, count); end if; end process; end architecture beh; 45 Yet Another Example 46 Subprogram Overloading • Subprograms – functions and procedures • Sometimes, it is convenient to have two or more subprograms with the same name – function count (oranges: integer) return integer; function count (apples: bit) return integer; – dff(clk, d, q, qbar); dff(clk, d, q, qbar, reset, clear); • In this case, subprograms are overloaded. – compiler will decide which subprogram to call based on the number and type of arguments. – and, or etc are overloaded in std_logic_1164 package. 47 Operator Overloading • An operator that behaves different depending on the operands is said to be overloaded – For example, “” and “+” are defined for certain predefined types of the language such as integers. – What if we want to use these operators to do multiplication and addition one

data types we newly defined. • Example – function “” (arg1, arg2: std_logic_vector) return std_logic_vector; – function “+” (arg1, arg2: signed) return signed; – these examples are taken from std_logic_arith.vhd package. 48 Operator Overloading • You can define a new multi-value logic and logic operations on the data of this new type. type MVL is (‘U’, ‘0’, ‘1’, ‘Z’); function “and” (L, R: MVL) return MVL; function “or” (L, R: MVL) return MVL; function “not” (R: MVL) return MVL; • Example: signal a, b, c: MVL; a <= ‘z’ or ‘1’; b <= “or”(‘0’, ‘1’); c <= (a or b) and (not c); 49 Packages • A package provides a convenient mechanism to store items that can be shared across distinct VHDL programs: – Those items are type definitions, functions, procedures, etc. – We group logically related sets of functions and procedures into a package. – When working with a large design project consisting of many small VHDL programs, it is convenient to have common procedures and functions in separate packages. – For example, a package containing definition of new types for registers, instructions, memories, etc. will certainly be useful for microprocessor design. 50 Package Declaration • Package declaration – contains information about what is available in the package that we can use in our VHDL programs. – In other words, it contains interface or specifications of the functions and procedures that are available in the package. – For example, – list of functions and procedures, – what are the parameters they take – the type of input parameters – What it returns, – what is the type of the returning value – etc. 51 Package Declaration: Syntax package package-name is package-item-declarations these may be: -- subprogram declarations -- type declarations -- subtype declarations -- constant declarations -- signal declarations -- variable declarations -- file declarations -- alias declarations -- component declarations -- attribute declarations -- attribute specifications -- disconnection specifications -- use clauses end [package] [package-name]; 52 Package Declaration: Example package synthesis_pack is constant low2high: time := 20 ns; type alu_op is (add, sub, mul, div, eql); attribute pipeline: Boolean; type mvl is (‘U’, ‘0’, ‘1’, ‘Z’); type mvl_vector is array (natural range <>) of mvl; subtype my_alu_op is alu_op range add to div; component nand2 port(a, b: in mvl; c: out mvl); end component; end synthesis_pack; • Items declared in a package declaration can be accessed by other design units by using library and use clauses. use work.synthesis_pack.all; 53 Yet Another Example use work.synthesis_pack.all; -- include all declarations -- from package synthesis_pack package program_pack is constant prop_delay: time; -- deferred constant function “and” (l, r: mvl) return mvl; procedure load (signal array_name: inout mvl_vector; start_bit, stop_bit, int_value: in integer); end_package program_pack; 54 Package Declaration: std_logic_1164 package std_logic_1164 is type std_ulogic is (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’ , ‘-’); type std_ulogic_vector is array (natural range <>) of std_ulogic; function resolved(s: std_ulogic_vector) return std_ulogic; subtype std_logic is resolved std_logic; type std_logic_vector is array (natural range <>) of std_logic; function “and”(l, r: std_logic_vector) return std_logic_vector; function “and”(l, r: std_ulogic_vector) return std_ulogic_vector; ... end package std_logic_1164; 55 Package Body • It basically contains the code that implements the subprograms • Syntax: package body package-name is package-body-item-declarations These are: -- subprogram bodies -- complete constant declarations -- subprogram declarations -- type and subtype declarations -- file and alias declarations -- use clauses end [package body] [package-name]; 56 Package Body: Example package body program_pack is use work.tables.all; constant prop_delay: time := 15 ns; function “and”(l, r: mvl) return mvl; begin return table_and(l, r); -- table_and is a 2-D constant defined in -- another package, called “tables” in the -- current working directory end “and”; procedure load(signal array_name: inout mvl_vector; start_bit, stop_bit, int_value: in integer) is begin -- procedure behavior here end load; end program_pack; 57 Libraries • Design units (design files) –

entity declaration, – architecture body, – configuration declaration, – package declaration, – package body • Each design unit is analyzed (compiled) and placed in a design library. – recall that libraries are generally implemented as directories and are referenced by a logical name. – this logical name corresponds to physical path to the corresponding directory. 58 WORK Compilation Process • VHDL analyzer verify the syntactic and semantic correctness of the source • then compiles each design unit into an intermediate form. • Each intermediate form is stored in a design library called working library. design file VHDL Analyzer intermediate form STD IEEE LIB1 design libraries 59 Libraries: Visibility • Implicit visibility – In VHDL, the libraries STD and WORK are implicitly declared. • Explicit visibility is achieved through – library clause – use clause – Example: library IEEE; use IEEE.std_logic_1164.all; • Once a library is declared, all of the subprograms, type declarations in this library become visible to our programs through the use of use clause. 60 Context Clauses • Examples for context clauses: – library IEEE; use IEEE.std_logic_1164.all; • Context clauses only applies the following design entity – if a file contains more than one entity, context clauses must precede each of them to provide the appropriate visibility to each design entity. 61 How to Create a Library? 62 How to Add Packages to a Library? 63 Package Declaration library IEEE; use IEEE.std_logic_1164.all; package my_pack_a is subtype word is std_logic_vector(15 downto 0); function "+" (op1, op2: word) return word; function "-" (op1, op2: word) return word; function "" (op1, op2: word) return word; end package my_pack_a; 64 Package Body library IEEE; use IEEE.std_logic_1164.all; use IEEE.std_logic_signed.all; use IEEE.std_logic_arith.all; package body my_pack_a is function "+" (op1, op2: word) return word is variable result: word; variable a, b, c: integer; begin a := conv_integer(op1); b := conv_integer(op2); c := a + b; result := conv_std_logic_vector(c, 16); return result; end function; ... end package body my_pack_a; 65 Package Body ... function "-" (op1, op2: word) return word is variable result: word; variable a, b, c: integer; begin a := conv_integer(op1); b := conv_integer(op2); c := a - b; result := conv_std_logic_vector(c, 16); return result; end function; function "" (op1, op2: word) return word is variable result: word; variable a, b, c: integer; begin a := conv_integer(op1); b := conv_integer(op2); c := a b; result := conv_std_logic_vector(c, 16); return result; end function; end package body my_pack_a; 66 Example using this Package library IEEE; library my_lib; use IEEE.std_logic_1164.all; use my_lib.my_pack_a.all; entity aritmetik is end entity; architecture beh of aritmetik is signal a, b, c: word; begin a <= x"abcd"; b <= x"1347"; c <= a + b; end architecture beh; 67 Simulation Results 68 Summary • Hierarchy • Functions • Procedures • subprogram overloading • operator overloading • packages • librarie

The Demultiplexer

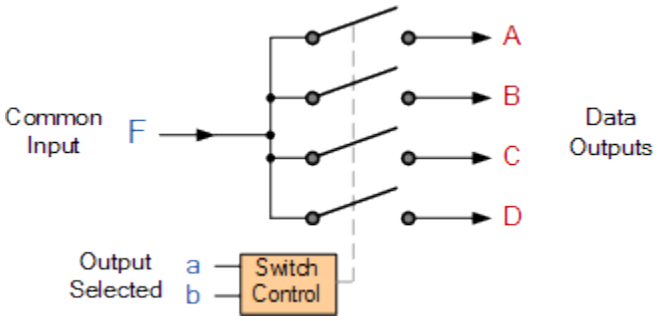
)



The data distributor, known more commonly as a **Demultiplexer** or "Demux" for short, is the exact opposite of the Multiplexer we saw in the previous tutorial.3

The demultiplexer takes one single input data line and then switches it to any one of a number of individual output lines one at a time. The **demultiplexer** converts a serial data signal at the input to a parallel data at its output lines as shown below.

1-to-4 Channel De-multiplexer



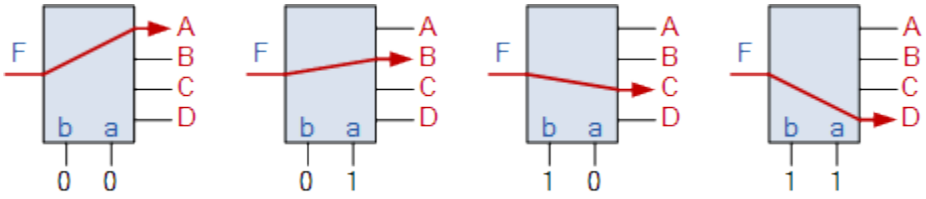
Output Select		Data Output Selected
a	b	
0	0	A
0	1	B
1	0	C
1	1	D

The Boolean expression for this 1-to-4 **Demultiplexer** above with outputs A to D and data select lines a, b is given as:

F = abA + abB + abC + abD

The function of the **Demultiplexer** is to switch one common data input line to any one of the 4 output data lines A to D in our example above. As with the multiplexer the individual solid state switches are selected by the binary input address code on the output select pins "a" and "b" as shown.

Demultiplexer Output Line Selection



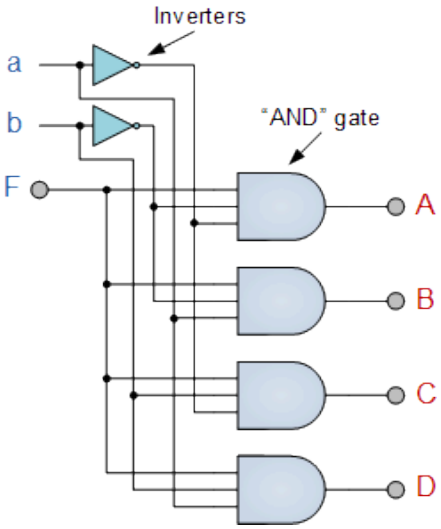
As with the previous **multiplexer circuit**, adding more address line inputs it is possible to switch more outputs giving a 1-to-2^n data line outputs.

Some standard demultiplexer IC´s also have an additional "enable output" pin which disables or prevents the input from being passed to the selected output. Also some have latches built into their outputs to maintain the output logic level after the address inputs have been changed.

However, in standard decoder type circuits the address input will determine which single data output will have the same value as the data input with all other data outputs having the value of logic "0".

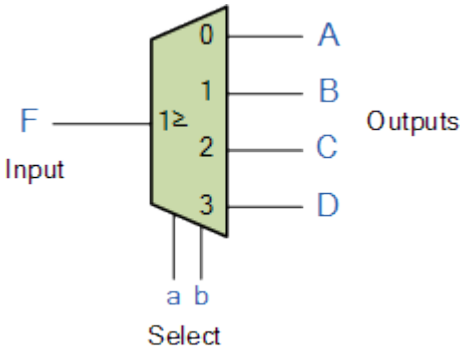
The implementation of the Boolean expression above using individual logic gates would require the use of six individual gates consisting of AND and NOT gates as shown.

4 Channel Demultiplexer using Logic Gates



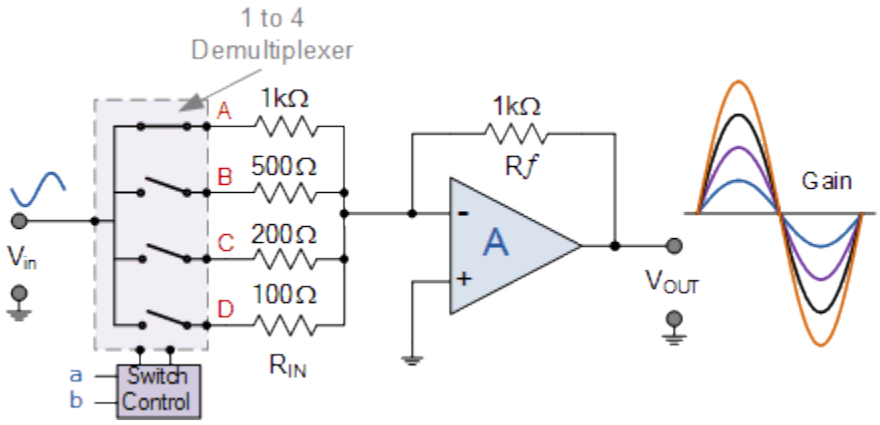
The symbol used in logic diagrams to identify a demultiplexer is as follows.

The Demultiplexer Symbol



Again, as with the previous multiplexer example, we can also use the demultiplexer to digitally control the gain of an operational amplifier as shown.

Digitally Adjustable Amplifier Gain



The circuit above illustrates how to provide digitally controlled adjustable/variable op-amp gain using a demultiplexer. The voltage gain of the inverting operational amplifier is dependent upon the ratio between the input resistor, Rin and its feedback resistor, R*f* as determined in the **Op-amp** tutorials.

The digitally controlled analogue switches of the demultiplexer select an input resistor to vary the value of Rin. The combination of these resistors will determine the overall gain of the amplifier, (Av). Then the voltage gain of the inverting operational amplifier can be adjusted digitally simply by selecting the appropriate input resistor combination.

Standard **Demultiplexer** IC packages available are the TTL 74LS138 1 to 8-output demultiplexer, the TTL 74LS139 Dual 1-to-4 output demultiplexer or the CMOS CD4514 1-to-16 output demultiplexer.

Another type of demultiplexer is the 24-pin, 74LS154 which is a 4-bit to 16-line demultiplexer/decoder. Here the individual output positions are selected using a 4-bit binary coded input. Like multiplexers, demultiplexers can also be cascaded together to form higher order demultiplexers.

Unlike multiplexers which convert data from a single data line to multiple lines and demultiplexers which convert multiple lines to a single data line, there are devices available which convert data to and from multiple lines and in the next tutorial about combinational logic devices, we will look at **Encoders** which convert multiple input lines into multiple output lines, converting the data from one form to another.