CONTENTS

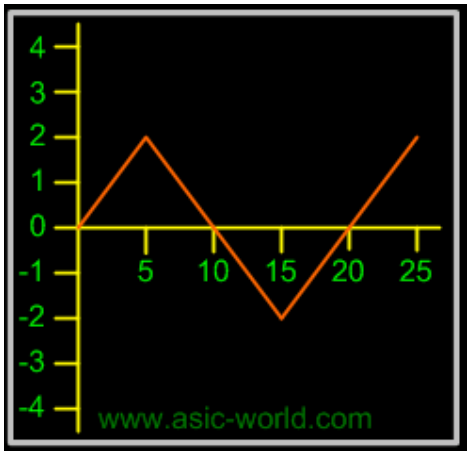
Introduction

The quantities that are to be measured, monitored, recorded, processed and controlled are analog and digital, depending on the type of system used. It is important when dealing with various quantities that we be able to represent their values efficiently and accurately. There are basically two ways of representing the numerical value of quantities: analog and digital.

Analog Representation

Systems which are capable of processing a continuous range of values varying with respect to time are called analog systems. In analog representation a quantity is represented by a voltage, current, or meter movement that is proportional to the value of that quantity. Analog quantities such as those cited above have an important characteristic: they can vary over a continuous range of values.

Diagram of analog voltage vs time

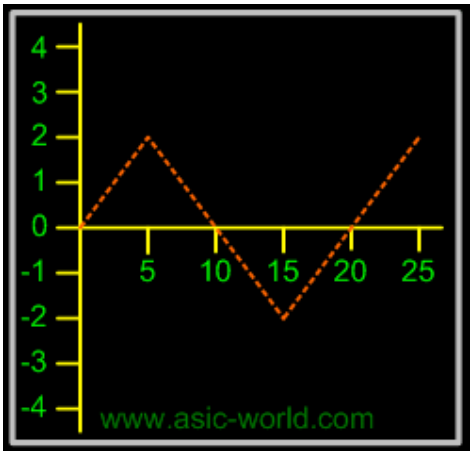


Digital Representation

Systems which process discrete values are called digital systems. In digital representation the quantities are represented not by proportional quantities but by symbols called digits. As an example, consider the digital watch, which provides the time of the day in the form of decimal digits representing hours and minutes (and sometimes seconds). As we know, time of day changes continuously, but the digital watch reading does not change continuously; rather, it changes in steps of one per minute (or per second). In other words, time of day digital representation changes in discrete steps, as compared to the representation of time provided by an analog watch, where the dial reading changes continuously.

Below is a diagram of digital voltage vs time: here input voltage changes from +4 Volts to -4 Volts; it can be converted to digital form by Analog to Digital converters (ADC). An ADC converts continuous signals into samples per second. Well, this is an entirely different theory.

Diagram of Digital voltage vs time



The major difference between analog and digital quantities, then, can be stated simply as follows:

Analog = continuous

- Digital = discrete (step by step)

Advantages of Digital Techniques

- Easier to design. Exact values of voltage or current are not important, only the range (HIGH or LOW) in which they fall.
- Information storage is easy.
- Accuracy and precision are greater.
- Operations can be programmed. Analog systems can also be programmed, but the available operations variety and complexity is severely limited.
- Digital circuits are less affected by noise, as long as the noise is not large enough to prevent us from distinguishing HIGH from LOW (we discuss this in detail in an advanced digital tutorial section).
- More digital circuitry can be fabricated on IC chips.

Limitations of Digital Techniques

Most physical quantities in real world are analog in nature, and these quantities are often the inputs and outputs that are being monitored, operated on, and controlled by a system. Thus conversion to digital format and re-conversion to analog format is needed.

Numbering System

Many number systems are in use in digital technology. The most common are the decimal, binary, octal, and hexadecimal systems. The decimal system is clearly the most familiar to us because it is a tool that we use every day. Examining some of its characteristics will help us to better understand the other systems. In the next few pages we shall introduce four numerical representation systems that are used in the digital system. There are other systems, which we will look at briefly.

- Decimal
- Binary
- Octal
- Hexadecimal

Decimal System

The decimal system is composed of 10 numerals or symbols. These 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Using these symbols as digits of a number, we can express any quantity.

The decimal system is also called the base-10 system because it has 10 digits.

Binary System

In the binary system, there are only two symbols or possible digit values, 0 and 1. This base-2 system can be used to represent any quantity that can be represented in decimal or other base system.

Octal System

The octal number system has a base of eight, meaning that it has eight possible digits: 0,1,2,3,4,5,6,7.

Hexadecimal System

The hexadecimal system uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F as the 16 digit symbols.

Code Conversion

Converting from one code form to another code form is called code conversion, like converting from binary to decimal or converting from hexadecimal to decimal.

Binary-To-Decimal Conversion

Any binary number can be converted to its decimal equivalent simply by summing together the weights of the various positions in the binary number which contain a 1.

Binary	Decimal
11011_2	
$2^4+2^3+0^1+2^1+2^0$	=16+8+0+2+1
Result	27_{10}

Decimal-To-Binary Conversion

Convert 25_{10} to binary

Division	Remainder	Binary
25/2	= 12+ remainder of 1	1 (Least Significant Bit)
12/2	= 6 + remainder of 0	0
6/2	= 3 + remainder of 0	0
3/2	= 1 + remainder of 1	1
1/2	= 0 + remainder of 1	1 (Most Significant Bit)
Result	25_{10}	= 11001_2

Binary-To-Octal / Octal-To-Binary Conversion

Octal Digit	0	1	2	3	4	5	6	7
Binary Equivalent	000	001	010	011	100	101	110	111

Each Octal digit is represented by three binary digits.

Example:

$100\ 111\ 010_2 = (100)\ (111)\ (010)_2 = 4\ 7\ 2_8$

Binary-To-Hexadecimal /Hexadecimal-To-Binary Conversion

Hexadecimal Digit	0	1	2	3	4	5	6	7
Binary Equivalent	0000	0001	0010	0011	0100	0101	0110	0111

Hexadecimal Digit	8	9	A	B	C	D	E	F
Binary Equivalent	1000	1001	1010	1011	1100	1101	1110	1111

Each Hexadecimal digit is represented by four bits of binary digit.

Example:

$1011\ 0010\ 1111_2 = (1011)\ (0010)\ (1111)_2 = B\ 2\ F_{16}$

Decimal to octal

This method uses repeated division by 8.

Example: Convert 177_{10} to octal and binary

Division	Result	Binary
177/8	= 22+ remainder of 1	1 (Least Significant Bit)
22/ 8	= 2 + remainder of 6	6
2 / 8	= 0 + remainder of 2	2 (Most Significant Bit)
Result	177_{10}	= 261_8
Binary		= 010110001_2

Hexadecimal to Decimal/Decimal to Hexadecimal Conversion

Example:

$2AF_{16} = 2 \times (16^2) + 10 \times (16^1) + 15 \times (16^0) = 687_{10}$

Example: convert 378_{10} to hexadecimal and binary:

Division	Result	Hexadecimal
378/16	= 23+ remainder of 10	A (Least Significant Bit)23
23/16	= 1 + remainder of 7	7
1/16	= 0 + remainder of 1	1 (Most Significant Bit)
Result	378_{10}	= $17A_{16}$
Binary		= 0001 0111 1010_2

Octal-To-Hexadecimal Hexadecimal-To-Octal Conversion

- Convert Octal (Hexadecimal) to Binary first.
- Regroup the binary number by three bits per group starting from LSB if Octal is required.
- Regroup the binary number by four bits per group starting from LSB if Hexadecimal is required.

Example:

Convert $5A8_{16}$ to Octal.

Hexadecimal	**Binary/Octal**
5A816	= **0101** 1010 **1000** (Binary)
	= **010** 110 **101** 000 (Binary)
Result	= 2 6 5 0 (Octal)

Binary Codes

Binary codes are codes which are represented in binary system with modification from the original ones. Below we will be seeing the following:

- Weighted Binary Systems
- Non Weighted Codes

Weighted Binary Systems

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example

Decimal	8421	2421	5211	Excess-3
0	0000	0000	0000	0011
1	0001	0001	0001	0100
2	0010	0010	0011	0101
3	0011	0011	0101	0110
4	0100	0100	0111	0111
5	0101	1011	1000	1000
6	0110	1100	1010	1001
7	0111	1101	1100	1010
8	1000	1110	1110	1011
9	1001	1111	1111	1100

8421 Code/BCD Code

The BCD (Binary Coded Decimal) is a straight assignment of the binary equivalent. It is possible to assign weights to the binary bits according to their positions. The weights in the BCD code are 8,4,2,1.

Example: The bit assignment 1001 can be seen by its weights to represent the decimal 9 because:

1x8+0x4+0x2+1x1 = 9

2421 Code

This is a weighted code; its weights are 2, 4, 2 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 2 + 4 + 2 + 1 = 9. Hence the 2421 code represents the decimal numbers from 0 to 9.

5211 Code

This is a weighted code; its weights are 5, 2, 1 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 5 + 2 + 1 + 1 = 9. Hence the 5211 code represents the decimal numbers from 0 to 9.

Reflective Code

A code is said to be reflective when code for 9 is complement for the code for 0, and so is for 8 and 1 codes, 7 and 2, 6 and 3, 5 and 4. Codes 2421, 5211, and excess-3 are reflective, whereas the 8421 code is not.

Sequential Codes

A code is said to be sequential when two subsequent codes, seen as numbers in binary representation, differ by one. This greatly aids mathematical manipulation of data. The 8421 and Excess-3 codes are sequential, whereas the 2421 and 5211 codes are not.

Non Weighted Codes

Non weighted codes are codes that are not positionally weighted. That is, each position within the binary number is not assigned a fixed value.

Excess-3 Code

Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus 0011(3).

Example: 1000 of 8421 = 1011 in Excess-3

Gray Code

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit-distance code. In digital Gray code has got a special place.

Decimal Number	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Binary to Gray Conversion

- Gray Code MSB is binary code MSB.
- Gray Code MSB-1 is the XOR of binary code MSB and MSB-1.
- MSB-2 bit of gray code is XOR of MSB-1 and MSB-2 bit of binary code.
- MSB-N bit of gray code is XOR of MSB-N-1 and MSB-N bit of binary code.

Basic Logical Functions and Gates

While each logical element or condition must always have a logic value of either "0" or "1", we also need to have ways to combine different logical signals or conditions to provide a logical result.

For example, consider the logical statement: "If I move the switch on the wall up, the light will turn on." At first glance, this seems to be a correct statement. However, if we look at a few other factors, we realize that there's more to it than this. In this example, a more complete statement would be: "If I move the switch on the wall up and the light bulb is good and the power is on, the light will turn on."

If we look at these two statements as logical expressions and use logical terminology, we can reduce the first statement to:

Light = Switch

This means nothing more than that the light will follow the action of the switch, so that when the switch is up/on/true/1 the light will also be on/true/1. Conversely, if the switch is own/off/false/0 the light will also be off/false/0.Looking at the second version of the statement, we have a slightly more complex expression:

Light = Switch and Bulb and Power

Normally, we use symbols rather than words to designate the and function that we're using to combine the separate variables of Switch, Bulb, and Power in this expression. The symbol normally used is a dot, which is the same symbol used for multiplication in some mathematical expressions. Using this symbol, our three-variable expression becomes:

Light = Switch •Bulb •Power

When we deal with logical circuits (as in computers), we not only need to deal with logical functions; we also need some special symbols to denote these functions in a logical diagram. There are three fundamental logical operations, from which all other functions, no matter how complex, can be derived. These functions are named and, or, and not.

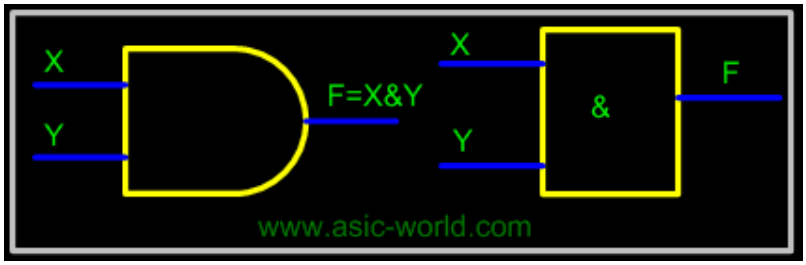
A logic gate is an electronic circuit/device which makes the logical decisions. To arrive at this decisions, the most common logic gates used are OR, AND, NOT, NAND, and NOR gates. The NAND and NOR gates are called universal gates. The exclusive-OR gate is another logic gate which can be constructed using AND, OR and NOT gate.

AND Gate

The AND gate performs logical multiplication, commonly known as AND function. The AND gate has two or more inputs and single output. The output of AND gate is HIGH only when all its inputs are HIGH (i.e. even if one input is LOW, Output will be LOW).

If X and Y are two inputs, then output F can be represented mathematically as F = X.Y, Here dot (.) denotes the AND operation. Truth table and symbol of the AND gate is shown in the figure below.

Symbol

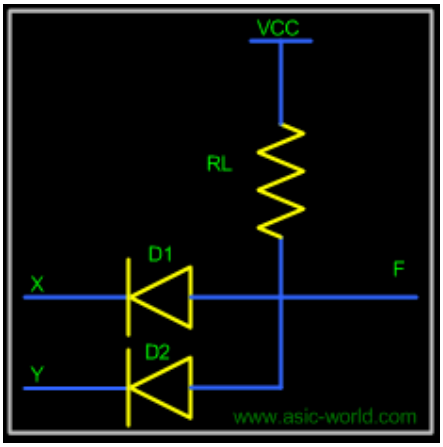


Truth Table

X	Y	F=(X.Y)
0	0	0
0	1	0
1	0	0
1	1	1

Two input AND gate using "diode-resistor" logic is shown in figure below, where X, Y are inputs and F is the output.

Circuit



If X = 0 and Y = 0, then both diodes D1 and D2 are forward biased and thus both diodes conduct and pull F low.

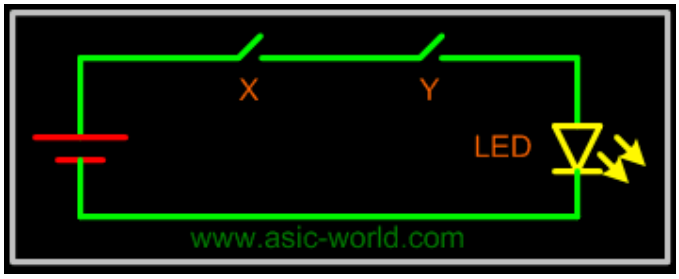
If X = 0 and Y = 1, D2 is reverse biased, thus does not conduct. But D1 is forward biased, thus conducts and thus pulls F low.

If X = 1 and Y = 0, D1 is reverse biased, thus does not conduct. But D2 is forward biased, thus conducts and thus pulls F low.

If X = 1 and Y = 1, then both diodes D1 and D2 are reverse biased and thus both the diodes are in cut-off and thus there is no drop in voltage at F. Thus F is HIGH.

Switch Representation of AND Gate

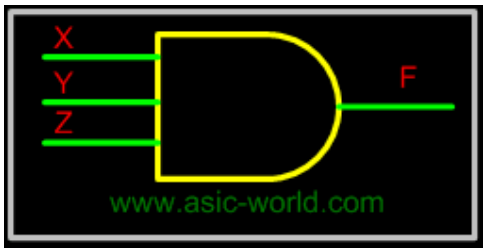
In the figure below, X and Y are two switches which have been connected in series (or just cascaded) with the load LED and source battery. When both switches are closed, current flows to LED.



Three Input AND gate

Since we have already seen how a AND gate works and I will just list the truth table of a 3 input AND gate. The figure below shows its symbol and truth table.

Circuit



Truth Table

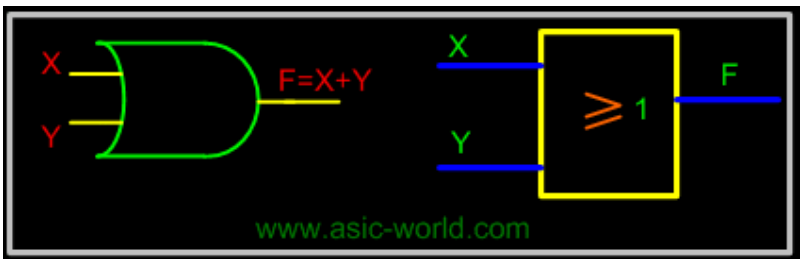
X	Y	Z	F=X.Y.Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

OR Gate

The OR gate performs logical addition, commonly known as OR function. The OR gate has two or more inputs and single output. The output of OR gate is HIGH only when any one of its inputs are HIGH (i.e. even if one input is HIGH, Output will be HIGH).

If X and Y are two inputs, then output F can be represented mathematically as F = X+Y. Here plus sign (+) denotes the OR operation. Truth table and symbol of the OR gate is shown in the figure below.

Symbol

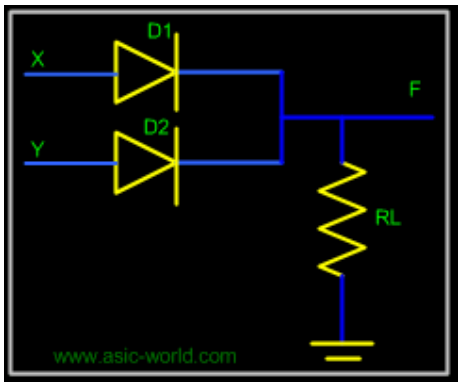


Truth Table

X	Y	F=(X+Y)
0	0	0
0	1	1
1	0	1
1	1	1

Two input OR gate using "diode-resistor" logic is shown in figure below, where X, Y are inputs and F is the output.

Circuit



If X = 0 and Y = 0, then both diodes D1 and D2 are reverse biased and thus both the diodes are in cut-off and thus F is low.

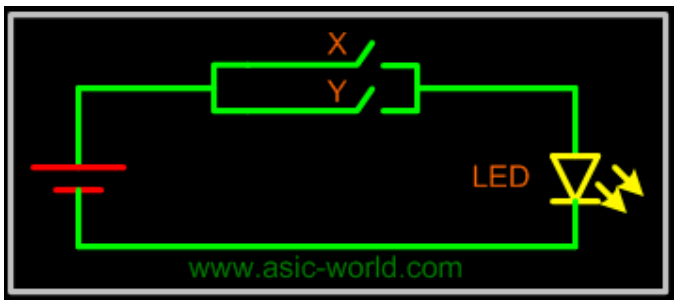
If X = 0 and Y = 1, D1 is reverse biased, thus does not conduct. But D2 is forward biased, thus conducts and thus pulling F to HIGH.

If X = 1 and Y = 0, D2 is reverse biased, thus does not conduct. But D1 is forward biased, thus conducts and thus pulling F to HIGH.

If X = 1 and Y = 1, then both diodes D1 and D2 are forward biased and thus both the diodes conduct and thus F is HIGH.

Switch Representation of OR Gate

In the figure, X and Y are two switches which have been connected in parallel, and this is connected in series with the load LED and source battery. When both switches are open, current does not flow to LED, but when any switch is closed then current flows.



Three Input OR gate

Since we have already seen how an OR gate works, I will just list the truth table of a 3-input OR gate. The figure below shows its circuit and truth table.

Truth Table

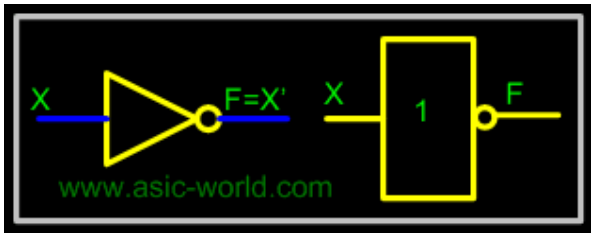
X	Y	Z	F=X+Y+Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

NOT Gate

The NOT gate performs the basic logical function called inversion or complementation. NOT gate is also called inverter. The purpose of this gate is to convert one logic level into the opposite logic level. It has one input and one output. When a HIGH level is applied to an inverter, a LOW level appears on its output and vice versa.

If X is the input, then output F can be represented mathematically as F = X', Here apostrophe (') denotes the NOT (inversion) operation. There are a couple of other ways to represent inversion, F=! X, here! represents inversion. Truth table and NOT gate symbol is shown in the figure below.

Symbol

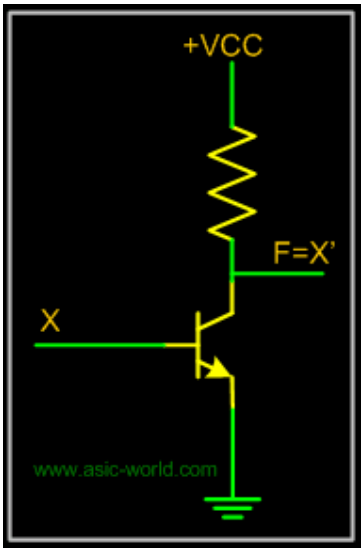


Truth Table

X	Y=X'
0	1
1	0

NOT gate using "transistor-resistor" logic is shown in the figure below, where X is the input and F is the output.

Circuit



When X = 1, the transistor input pin 1 is HIGH, this produces the forward bias across the emitter base junction and so the transistor conducts. As the collector current flows, the voltage drop across RL increases and hence F is LOW.

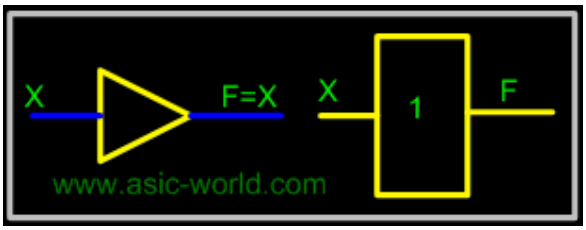
When X = 0, the transistor input pin 2 is LOW: this produces no bias voltage across the transistor base emitter junction. Thus Voltage at F is HIGH.

BUF Gate

Buffer or BUF is also a gate with the exception that it does not perform any logical operation on its input. Buffers just pass input to output. Buffers are used to increase the drive strength or sometime just to introduce delay. We will look at this in detail later.

If X is the input, then output F can be represented mathematically as F = X. Truth table and symbol of the Buffer gate is shown in the figure below.

Symbol



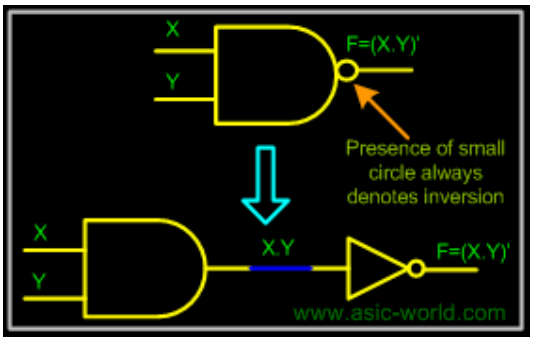
Truth Table

X	Y=X
0	0
1	1

NAND Gate

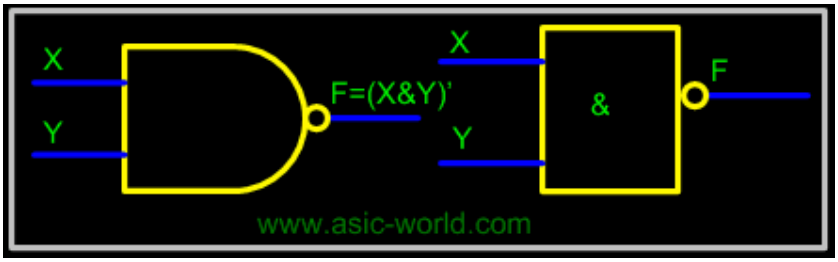
NAND gate is a cascade of AND gate and NOT gate, as shown in the figure below. It has two or more inputs and only one output. The output of NAND gate is HIGH when any one of its input is LOW (i.e. even if one input is LOW, Output will be HIGH).

NAND From AND and NOT



If X and Y are two inputs, then output F can be represented mathematically as F = (X.Y)', Here dot (.) denotes the AND operation and (') denotes inversion. Truth table and symbol of the N AND gate is shown in the figure below.

Symbol



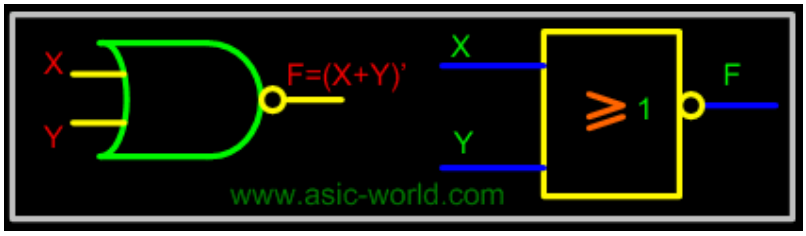
Truth Table

X	Y	F=(X.Y)'
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate

NOR gate is a cascade of OR gate and NOT gate, as shown in the figure below. It has two or more inputs and only one output. The output of NOR gate is HIGH when any all its inputs are LOW (i.e. even if one input is HIGH, output will be LOW).

Symbol



If X and Y are two inputs, then output F can be represented mathematically as F = (X+Y)'; here plus (+) denotes the OR operation and (') denotes inversion. Truth table and symbol of the NOR gate is shown in the figure below.

Truth Table

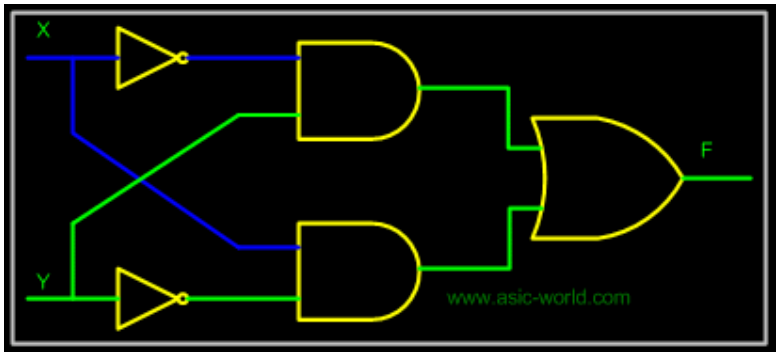
X	Y	F=(X+Y)'
0	0	1
0	1	0
1	0	0
1	1	0

XOR Gate

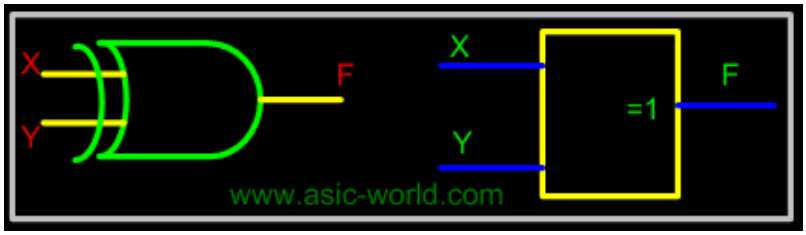
An Exclusive-OR (XOR) gate is gate with two or three or more inputs and one output. The output of a two-input XOR gate assumes a HIGH state if one and only one input assumes a HIGH state. This is equivalent to saying that the output is HIGH if either input X or input Y is HIGH exclusively and LOW when both are 1 or 0 simultaneously.

If X and Y are two inputs, then output F can be represented mathematically as F = X⊕Y, Here ⊕denotes the XOR operation. X⊕Y and is equivalent to X.Y' + X'.Y. Truth table and symbol of the XOR gate is shown in the figure below.

XOR from Simple gates



Symbol



Truth Table

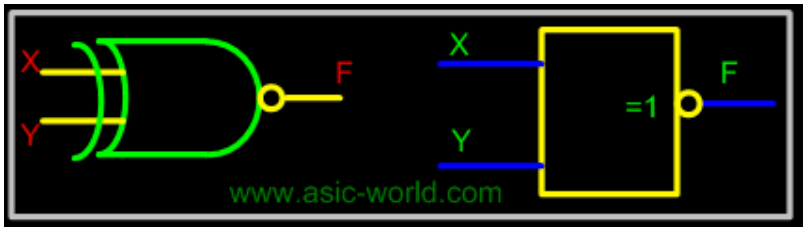
X	Y	F=(X Y)
0	0	0
0	1	1
1	0	1
1	1	0

XNOR Gate

An Exclusive-NOR (XNOR) gate is gate with two or three or more inputs and one output. The output of a two-input XNOR gate assumes a HIGH state if all the inputs assumes same state. This is equivalent to saying that the output is HIGH if both input X and input Y is HIGH exclusively or same as input X and input Y is LOW exclusively, and LOW when both are not same.

If X and Y are two inputs, then output F can be represented mathematically as F = X⊕Y, Here ⊕denotes the XNOR operation. X⊕Y and is equivalent to X.Y + X'.Y'. Truth table and symbol of the XNOR gate is shown in the figure below.

Symbol

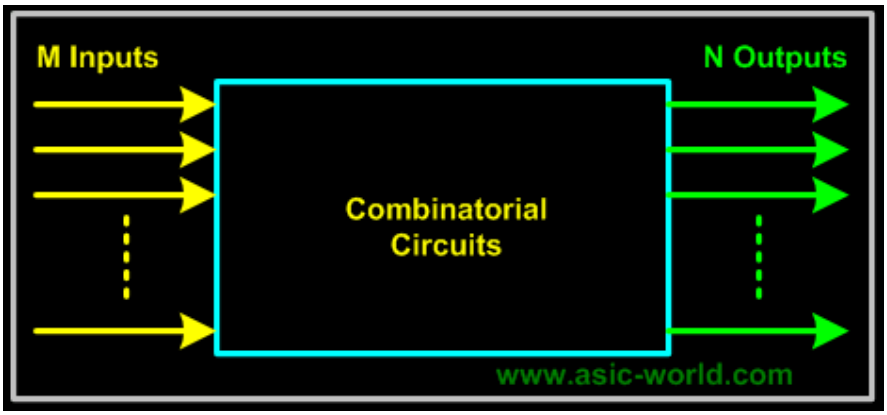


Truth Table

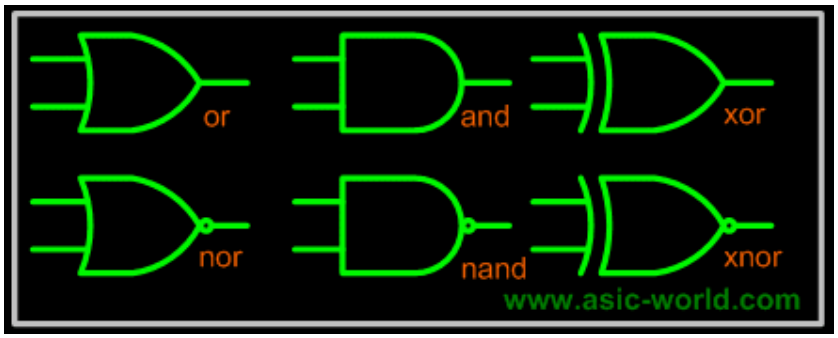
X	Y	F=(X Y)'
0	0	1
0	1	0
1	0	0
1	1	1

Combinational Logic

Combinatorial Circuits are circuits which can be considered to have the following generic structure.



Whenever the same set of inputs is fed in to a combinatorial circuit, the same outputs will be generated. Such circuits are said to be stateless. Some simple combinational logic elements that we have seen in previous sections are "Gates".

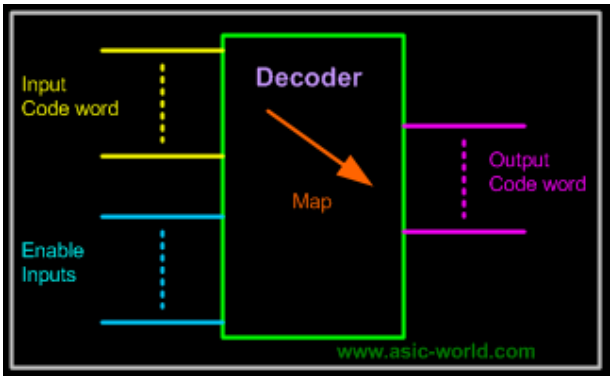


All the gates in the above figure have 2 inputs and one output; combinational elements simplest form are "not" gate and "buffer" as shown in the figure below. They have only one input and one output.

Decoders

A decoder is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different; e.g. n-to-2n, BCD decoders. Enable inputs must be on for the decoder to function, otherwise its outputs assume a single "disabled" output code word.

Decoding is necessary in applications such as data multiplexing, 7 segment display and memory address decoding. Figure below shows the pseudo block of a decoder.



Binary n-to-2^n Decoders

A binary decoder has n inputs and 2^n outputs. Only one output is active at any one time, corresponding to the input value. Figure below shows a representation of Binary n-to-2^n decoder

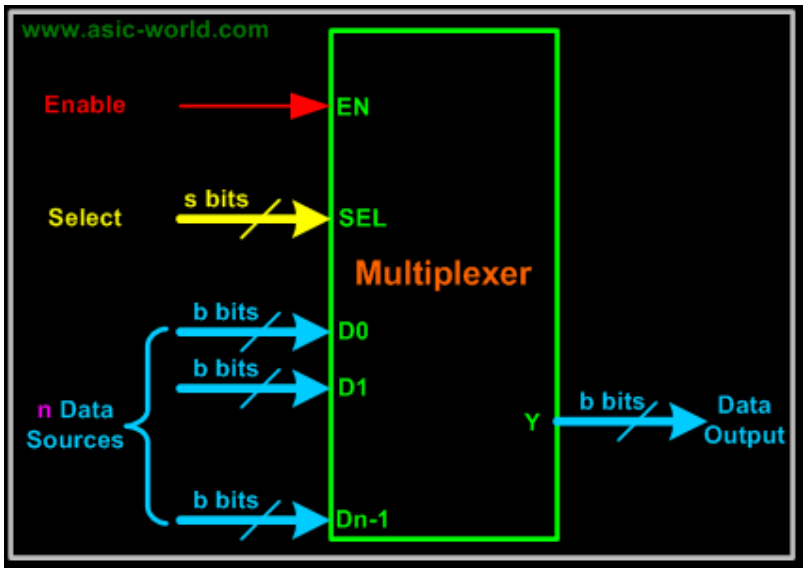
Encoders

An encoder is a combinational circuit that performs the inverse operation of a decoder. If a device output code has fewer bits than the input code has, the device is usually called an encoder. e.g. 2^n-to-n, priority encoders.

The simplest encoder is a 2^n-to-n binary encoder, where it has only one of 2^n inputs = 1 and the output is the n-bit binary number corresponding to the active input.

Multiplexer

A multiplexer (MUX) is a digital switch which connects data from one of n sources to the output. A number of select inputs determine which data source is connected to the output. The block diagram of MUX with n data sources of b bits wide and s bits wide select line is shown in below figure.

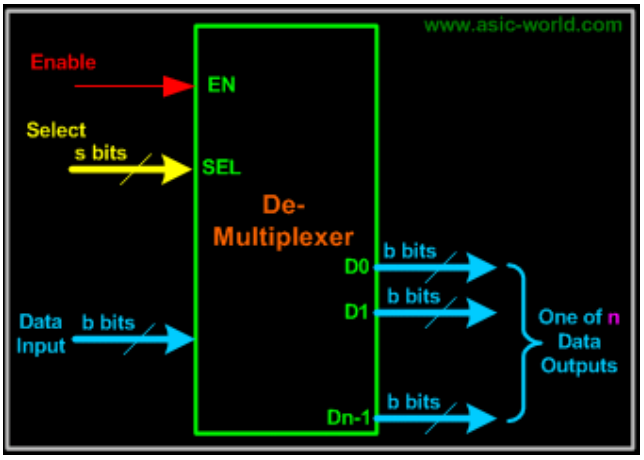


MUX acts like a digitally controlled multi-position switch where the binary code applied to the select inputs controls the input source that will be switched on to the output as shown in the figure below. At any given point of time only one input gets selected and is connected to output, based on the select input signal.

De-multiplexers

They are digital switches which connect data from one input source to one of n outputs. Usually implemented by using n-to-2^n binary decoders where the decoder enable line is used for data input of the de-multiplexer.

The figure below shows a de-multiplexer block diagram which has got s-bits-wide select input, one b-bits-wide data input and n b-bits-wide outputs.



Adders

Adders are the basic building blocks of all arithmetic circuits; adders add two binary numbers and give out sum and carry as output. Basically we have two types of adders.

- Half Adder.
- Full Adder.

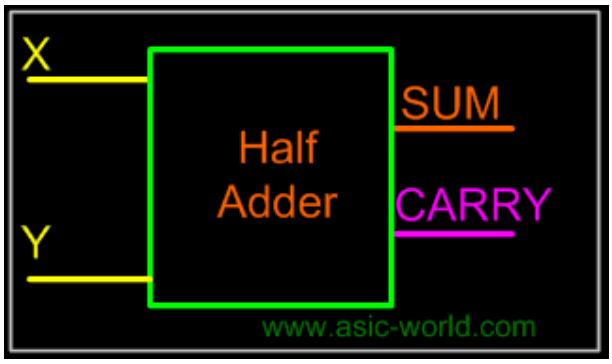
Half Adder

Adding two single-bit binary values X, Y produces a sum S bit and a carry out C-out bit. This operation is called half addition and the circuit to realize it is called a half adder.

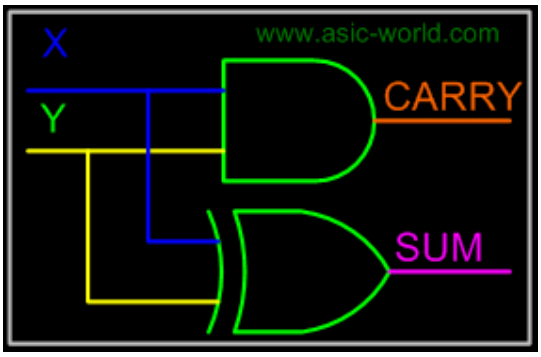
Truth Table

X	Y	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Symbol



Circuit



Full Adder

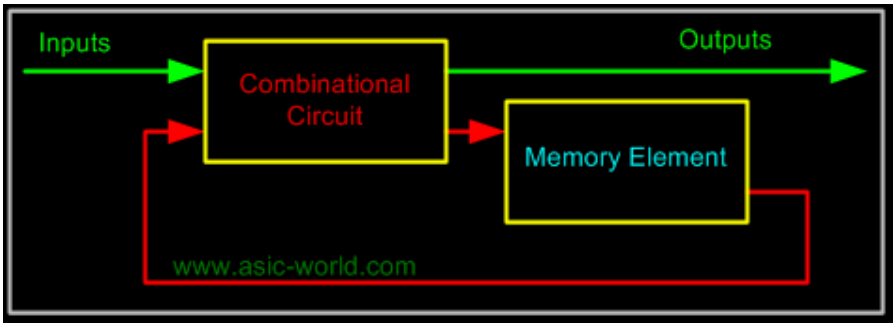
Full adder takes a three-bits input. Adding two single-bit binary values X, Y with a carry input bit C-in produces a sum bit S and a carry out C-out bit.

Truth Table

X	Y	Z	SUM	CARRY
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Sequential Circuits

Digital electronics is classified into combinational logic and sequential logic. Combinational logic output depends on the inputs levels, whereas sequential logic output depends on stored levels and also the input levels.



The memory elements are devices capable of storing binary info. The binary info stored in the memory elements at any given time defines the state of the sequential circuit. The input and the present state of the memory element determine the output. Memory elements next state is also a function of external inputs and present state. A sequential circuit is specified by a time sequence of inputs, outputs, and internal states.

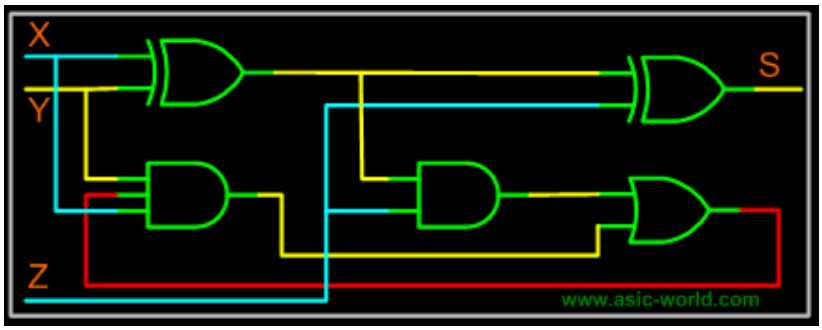
There are two types of sequential circuits. Their classification depends on the timing of their signals:

- Synchronous sequential circuits
- Asynchronous sequential circuits

Asynchronous sequential circuit

This is a system whose outputs depend upon the order in which its input variables change and can be affected at any instant of time.

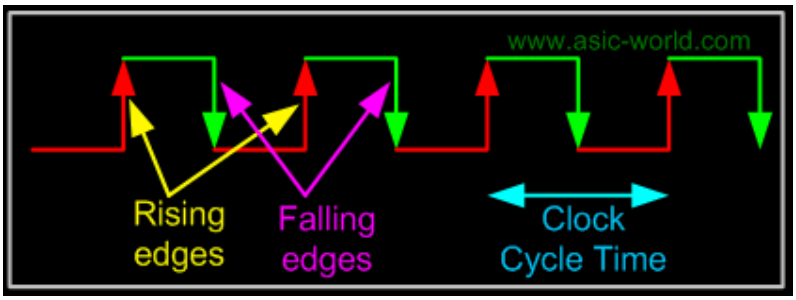
Gate-type asynchronous systems are basically combinational circuits with feedback paths. Because of the feedback among logic gates, the system may, at times, become unstable. Consequently they are not often used.



Synchronous sequential circuits

This type of system uses storage elements called flip-flops that are employed to change their binary value only at discrete instants of time. Synchronous sequential circuits use logic gates and flip-flop storage devices. Sequential circuits have a clock signal as one of their inputs. All state transitions in such circuits occur only when the clock value is either 0 or 1 or happen at the rising or falling edges of the clock depending on the type of memory elements used in the circuit.

Synchronization is achieved by a timing device called a clock pulse generator. Clock pulses are distributed throughout the system in such a way that the flip-flops are affected only with the arrival of the synchronization pulse. Synchronous sequential circuits that use clock pulses in the inputs are called clocked-sequential circuits. They are stable and their timing can easily be broken down into independent discrete steps, each of which is considered separately.



A clock signal is a periodic square wave that indefinitely switches from 0 to 1 and from 1 to 0 at fixed intervals. Clock cycle time or clock period: the time interval between two consecutive rising or falling edges of the clock.

Clock Frequency = 1 / clock cycle time (measured in cycles per second or Hz)

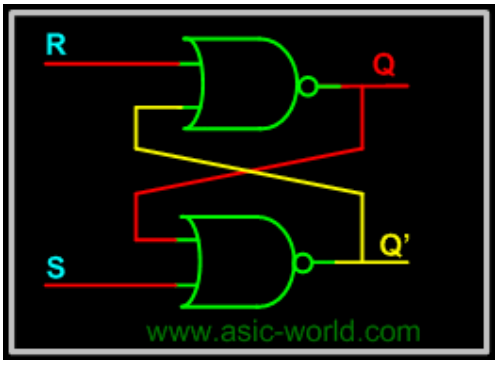
Example: Clock cycle time = 10ns clock frequency = 100 MHz

Latches and Flip-Flops

Latches and Flip-flops are one and the same with a slight variation: Latches have level sensitive control signal input and Flip-flops have edge sensitive control signal input. Flip-flops and latches which use this control signals are called synchronous circuits. So if they don't use clock inputs, then they are called asynchronous circuits.

RS Latch

RS latch have two inputs, S and R. S is called set and R is called reset. The S input is used to produce HIGH on Q (i.e. store binary 1 in flip-flop). The R input is used to produce LOW on Q (i.e. store binary 0 in flip-flop). Q' is Q complementary output, so it always holds the opposite value of Q. The output of the S-R latch depends on current as well as previous inputs or state, and its state (value stored) can change as soon as its inputs change. The circuit and the truth table of RS latch is shown below.

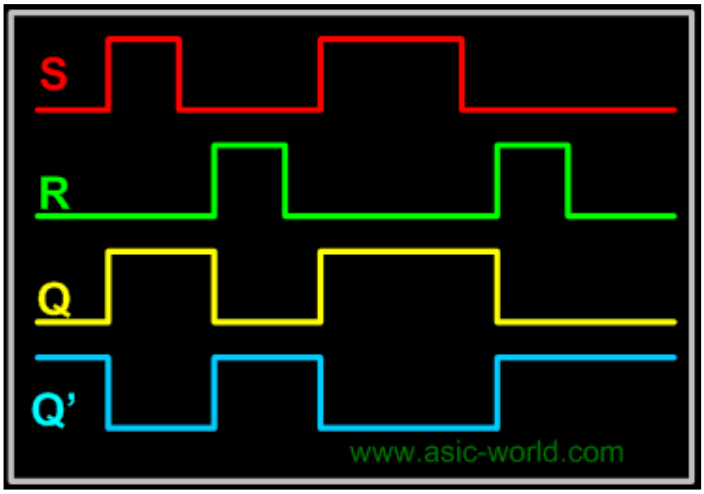


S	R	Q	Q+
0	0	0	0
0	0	1	1
0	1	X	0
1	0	X	1
1	1	X	0

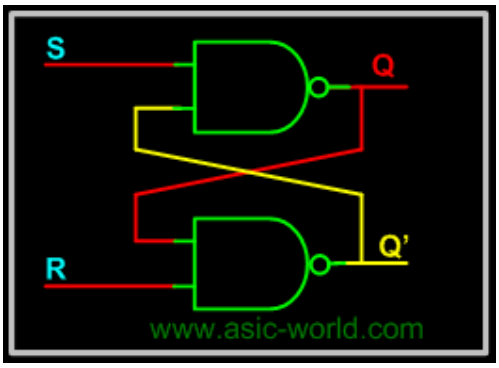
The operation has to be analyzed with the 4 inputs combinations together with the 2 possible previous states.

- **When S = 0 and R = 0:** If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. So it is clear that when both S and R inputs are LOW, the output is retained as before the application of inputs. (i.e. there is no state change).
- **When S = 1 and R = 0:** If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. So in simple words when S is HIGH and R is LOW, output Q is HIGH.
- **When S = 0 and R = 1:** If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. So in simple words when S is LOW and R is HIGH, output Q is LOW.
- **When S = 1 and R =1 :** No matter what state Q and Q' are in, application of 1 at input of NOR gate always results in 0 at output of NOR gate, which results in both Q and Q' set to LOW (i.e. Q = Q'). LOW in both the outputs basically is wrong, so this case is invalid.

The waveform below shows the operation of NOR gate based RS Latch.



It is possible to construct the RS latch using NAND gates (of course as seen in Logic gates section). The only difference is that NAND neither is NOR gate dual form (Did I say that in Logic gates section?). So in this case the R = 0 and S = 0 case becomes the invalid case. The circuit and Truth table of RS latch using NAND is shown below.



S	R	Q	Q+
1	1	0	0
1	1	1	1
0	1	X	0
1	0	X	1
0	0	X	1

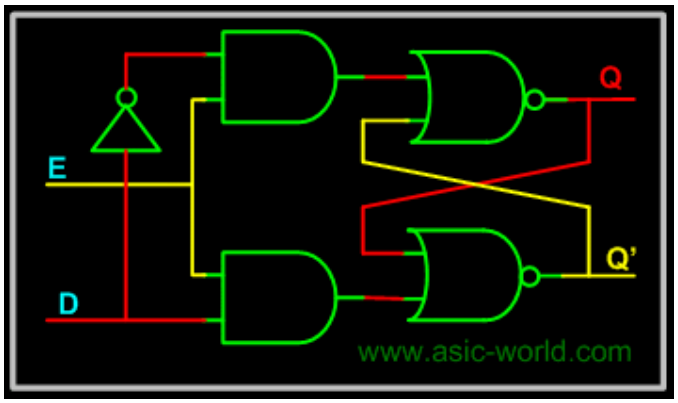
If you look closely, there is no control signal (i.e. no clock and no enable), so these kinds of latches or flip-flops are called asynchronous logic elements. Since all the sequential circuits are built around the RS latch, we will concentrate on synchronous circuits and not on asynchronous circuits.

D Latch

The RS latch seen earlier contains ambiguous state; to eliminate this condition we can ensure that S and R are never equal. This is done by connecting S and R together with an inverter. Thus we have D Latch: the same as the RS latch, with the only difference that there is only one input, instead of two (R and S). This input is called D or

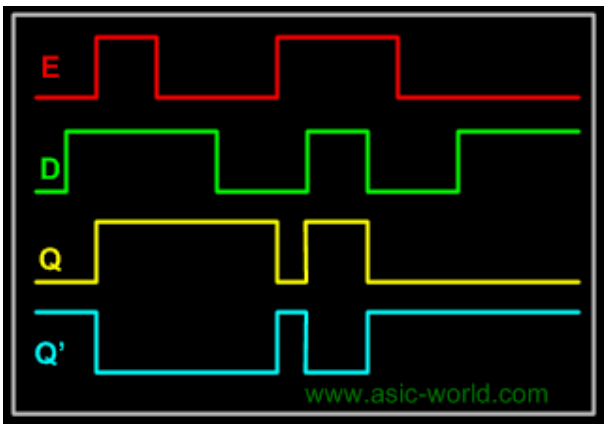
Data input. D latch is called D transparent latch for the reasons explained earlier. Delay flip-flop or delay latch is another name used. Below is the truth table and circuit of D latch.

In real world designs (ASIC/FPGA Designs) only D latches/Flip-Flops are used.



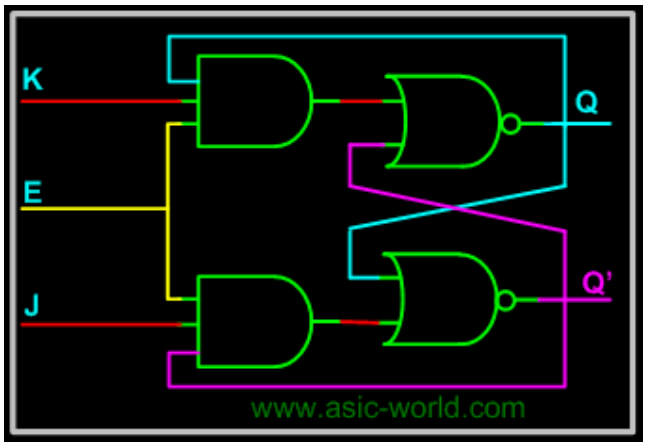
D	Q	Q+
1	X	1
0	X	0

Below is the D latch waveform, which is similar to the RS latch one, but with R removed.



JK Latch

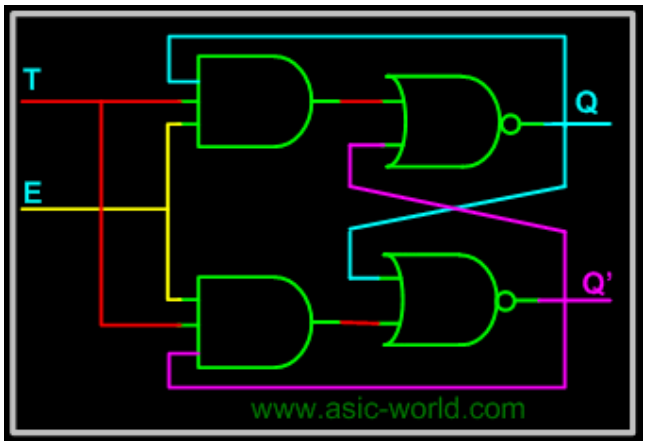
The ambiguous state output in the RS latch was eliminated in the D latch by joining the inputs with an inverter. But the D latch has a single input. JK latch is similar to RS latch in that it has 2 inputs J and K as shown figure below. The ambiguous state has been eliminated here: when both inputs are high, output toggles. The only difference we see here is output feedback to inputs, which is not there in the RS latch.



J	K	Q
1	1	0
1	1	1
1	0	1
0	1	0

T Latch

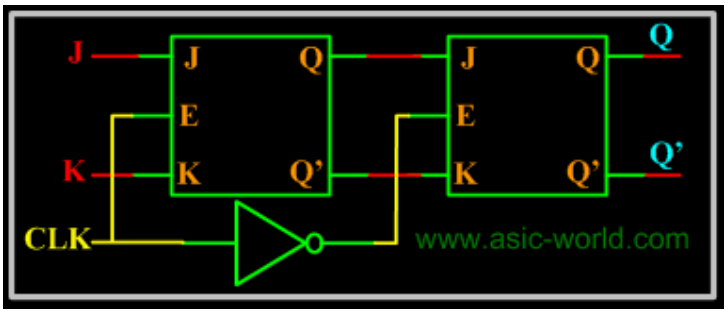
When the two inputs of JK latch are shorted, a T Latch is formed. It is called T latch as, when input is held HIGH, output toggles.



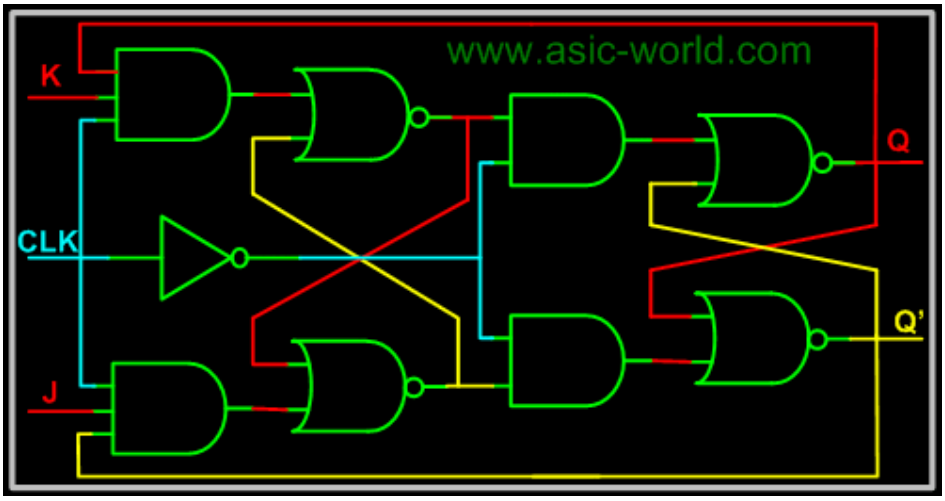
T	Q	Q+
1	0	1
1	1	0
0	1	1
0	0	0

JK Master Slave Flip-Flop

All sequential circuits that we have seen in the last few pages have a problem (All level sensitive sequential circuits have this problem). Before the enable input changes state from HIGH to LOW (assuming HIGH is ON and LOW is OFF state), if inputs changes, then another state transition occurs for the same enable pulse. This sort of multiple transition problem is called racing. If we make the sequential element sensitive to edges, instead of levels, we can overcome this problem, as input is evaluated only during enable/clock edges.



In the figure above there are two latches, the first latch on the left is called master latch and the one on the right is called slave latch. Master latch is positively clocked and slave latch is negatively clocked.



VHDL

CONTENTS

CHAPTER 1

INTRODUCTION TO VLSI

Introduction

Integrated circuits were made possible by experimental discoveries which showed that semiconductor devices could perform the functions of vacuum tubes, and by mid-20th-century technology advancements in semiconductor device fabrication. The integration of large numbers of tiny transistors into a small chip was an enormous improvement over the manual assembly of circuits using discrete electronic components. The integrated circuit's mass production capability, reliability, and building-block approach to circuit design ensured the rapid adoption of standardized ICs in place of designs using discrete transistors. There are two main advantages of ICs over discrete circuits - cost and performance. Cost is low because the chips, with all their components, are printed as a unit by photolithography and not constructed a transistor at a time. Performance is high since the components switch quickly and consume little power, because the components are small and close together. As of 2006, chip areas range from a few square mm to around 250 mm^2, with up to 1 million transistors per mm^2.

Advances in Integrated circuits

Among the most advanced integrated circuits are the microprocessors, which control everything from computers to cellular phones to digital microwave ovens. Digital memory chips are another family of integrated circuit that is crucially important to the modern information society. While the cost of designing and developing a complex integrated circuit is quite high, when spread across typically millions of production units the individual IC cost is minimized. The performance of ICs is high because the small size allows short traces, which in turn allows low power logic (such as CMOS) to be used at fast switching speeds.

ICs have consistently migrated to smaller feature sizes over the years, allowing more circuitry to be packed on each chip. As the feature size shrinks, almost everything improves - the cost per unit and the switching power consumption go down, and the speed goes up. However, IC's with nanometer-scale devices are not without their problems, principal among which is leakage current, although these problems are not insurmountable and will likely be solved or at least ameliorated by the introduction of high-k dielectrics. Since these speed and power consumption gains are apparent to the end user, there is fierce competition among the manufacturers to use finer geometries. This process, and the expected progress over the next few years, is well described by the International Technology Roadmap for Semiconductors, or ITRS.

SSI, MSI, LSI

The first integrated circuits contained only a few transistors. Called "**Small-Scale Integration**" (**SSI**), they used circuits containing transistors numbering in the tens.SSI circuits were crucial to early aerospace projects, and vice-versa. Both the Minuteman missile and Apollo program needed lightweight digital computers for their inertially-guided flight computers; the Apollo guidance computer led and motivated the integrated-circuit technology, while the Minuteman missile forced it into mass-production.These programs purchased almost all of the available integrated circuits from 1960 through 1963, and almost alone provided the demand that funded the production improvements to get the production costs from $1000/circuit (in 1960 dollars) to merely $25/circuit (in 1963 dollars).

The next step in the development of integrated circuits, taken in the late 1960s, introduced devices which contained hundreds of transistors on each chip, called "**Medium-Scale Integration**" (**MSI**). They were attractive economically because while they cost little more to produce than SSI devices, they allowed more complex systems to be produced using smaller circuit boards, less assembly work, and a number of other advantages. Further development, driven by the same economic factors, led to "**Large-Scale Integration**" (**LSI**) in the mid 1970s, with tens of thousands of transistors per chip. LSI circuits began to be produced in large quantities around 1970, for computer main memories and pocket calculators.

VLSI

The final step in the development process, starting in the 1980s and continuing on, was "Very Large-Scale Integration" (VLSI), with hundreds of thousands of transistors, and beyond (well past several million in the latest stages). For the first time it became possible to fabricate a CPU on a single integrated circuit, to create a microprocessor. In 1986 the first one megabit RAM chips were introduced, which contained more than one million transistors. Microprocessor chips produced in 1994 contained more than three million transistors. This step was largely made possible by the codification of "design rules" for the CMOS technology used in VLSI chips, which made production of working devices much more of a systematic endeavor.

ULSI, WSI, SOC

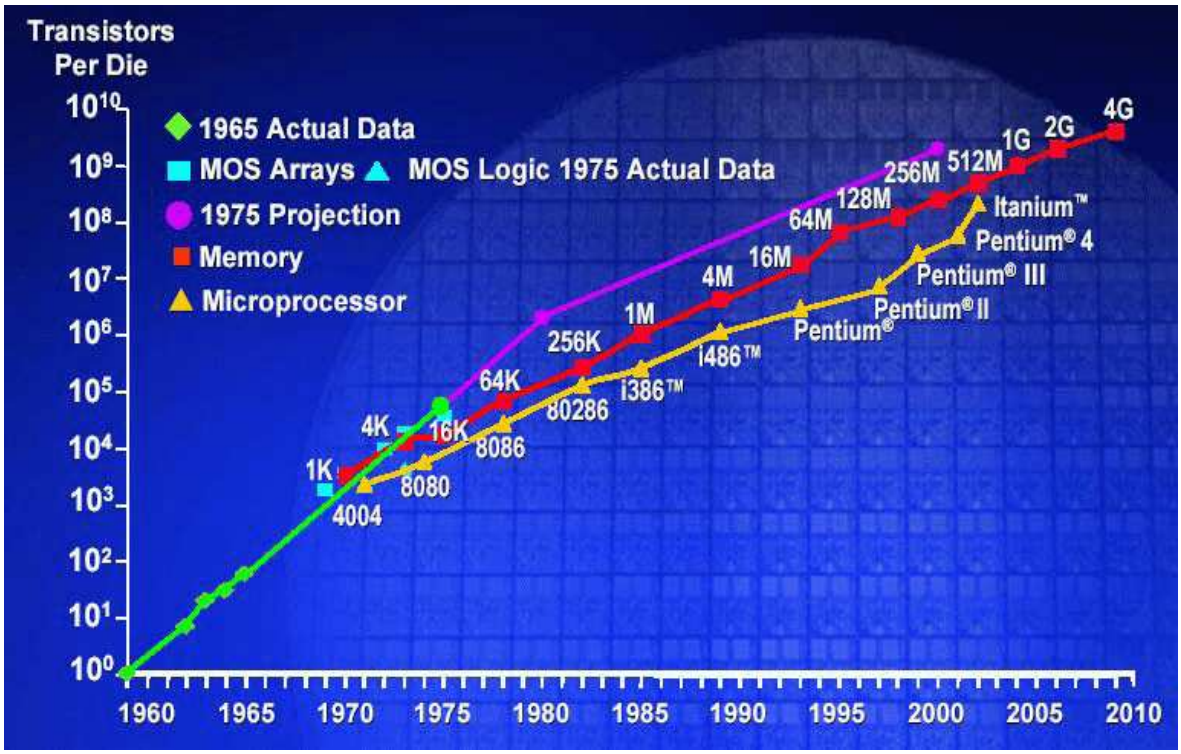
To reflect further growth of the complexity, the term **ULSI** that stands for "**Ultra-Large Scale Integration**" was proposed for chips of complexity more than 1 million of transistors. However there is no qualitative leap between VLSI and ULSI, hence normally in technical texts the "VLSI" term covers ULSI as well, and "ULSI" is reserved only for cases when it is necessary to emphasize the chip complexity, e.g. in marketing.

The most extreme integration technique is **wafer-scale integration** (**WSI**), which uses whole uncut wafers containing processors as well as memory. Attempts to take this step commercially in the 1980s (e.g. by Gene Amdahl) failed, mostly because of defect-free manufacturability problems, and it does not now seem to be a high priority for industry. The WSI technique failed commercially, but advances in semiconductor manufacturing allowed for another attack on the IC complexity, known as **System-on-Chip** (**SOC**) design. In this approach, components traditionally manufactured as separate chips to be wired together on a printed circuit board are designed to occupy a single chip that contains memory, microprocessor, peripheral interfaces, Input/Output logic control, data converters, and other components, together composing the whole electronic system.

Other developments

In the 1980s programmable integrated circuits were developed. These devices contain circuits whose logical function and connectivity can be programmed by the user, rather than being fixed by the integrated circuit manufacturer. This allows a single chip to be programmed to implement different LSI-type functions such as logic gates, adders, and registers. Current devices named FPGAs (Field Programmable Gate Arrays) can now implement tens of thousands of LSI circuits in parallel and operate up to 400 MHz. The techniques perfected by the integrated circuits industry over the last three decades have been used to create microscopic machines, known as MEMS. These devices are used in a variety of commercial and defense applications, including projectors, ink jet printers, and accelerometers used to deploy the airbag in car accidents. In the past, radios could not be fabricated in the same low-cost processes as microprocessors. But since 1998, a large number of radio chips have been developed using CMOS processes. Examples include Intel's DECT cordless phone, or Atheros's 802.11 card.

Moore's Law



The growth of **complexity of integrated circuits follows a trend called "Moore's Law", first observed by Gordon Moore of Intel. Moore's Law in its modern interpretation states that the number of transistors in an integrated circuit doubles every two years. By the year 2000 the largest integrated circuits contained hundreds of millions of transistors. It is difficult to say whether the trend will continue.**

Popularity of ICs

Only a half century after their development was initiated, integrated circuits have become ubiquitous. Computers, cellular phones, and other digital appliances are now inextricable parts of the structure of modern societies. That is, modern computing, communications, manufacturing and transport systems, including the Internet, all depend on the existence of integrated circuits. Indeed, many scholars believe that the digital revolution brought about by integrated circuits was one of the most significant occurrences in the history of mankind.

Why VLSI?

Integration improves the design:

- Lower parasitic = higher speed.
- Lower power.
- Physically smaller.
- Integration reduces manufacturing cost-no manual assembly.

Challenges in VLSI Design

- Multiple levels of abstraction: transistors to CPUs.
- Multiple and conflicting constraints: low cost and high performances are often at odds.
- Short design time: Late products are often irrelevant.

Dealing with Complexity

Divide-and-conquer: limit the number of components you deal with at any one time.

Group several components into larger components:

- transistors form gates;
- gates form functional units;
- Functional units form processing elements, etc.

Top-down vs. Bottom-up Design

- Top-down design adds functional detail. Create lower levels of abstraction from upper levels.
- Bottom-up design creates abstractions from low-level behavior.
- Good design needs both top-down and bottom-up efforts.

Design Strategies

IC design productivity depends on the efficiency with which the design may be converted from concept to architecture, to logic and memory, to circuit and hence to a physical layout. A good design strategy with a good design system should provide for consistent descriptions in various abstraction levels. The role of good design strategies is to reduce complexity, increase productivity, and assure working product.

Design is a continuous trade-off to achieve adequate results for:

- Performance - speed, power, function, flexibility
- Size of die (hence cost of die)
- Time to design
- Ease of test generation and testability

Hardware Description Languages (HDLs)

IEEE standardized Language

- **VHDL**
- **VerilogHDL**

What is VHDL?

- **VHDL: V**HSIC **H**ardware **D**escription **L**anguage
  –**VHSIC: V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
- Developed originally by DARPA
  –for specifying digital systems
- International IEEE standard (IEEE 1076-1993)
- Hardware Description, Simulation, Synthesis
- Practical benefits:
  –a mechanism for digital design and reusable design documentation
  –Model interoperability among vendors
  –Third party vendor support
  –Design re-use.

VHDL vs. C/Pascal

C/Pascal:

–Procedural programming languages.

–Typically describe procedures for computing a math's function or manipulation of data.

(e.g., sorting, matrix computing)

–A program is a recipe or a sequence of steps for how to perform a computation or

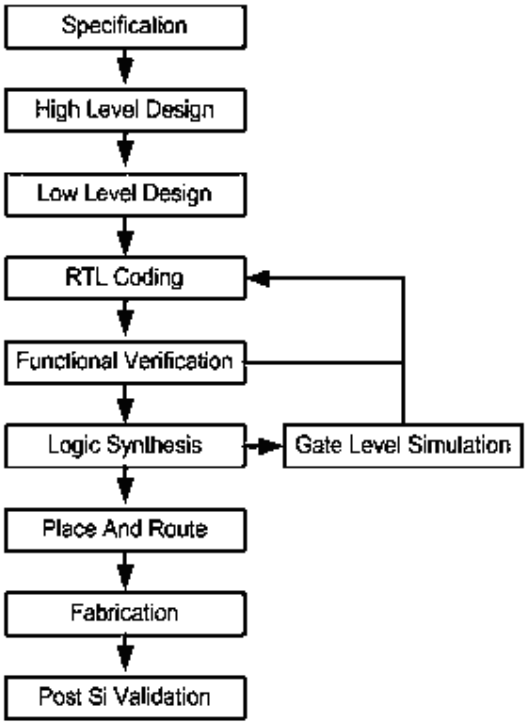
manipulate data.

VHDL:

– A language to describe digital systems.

–Purposes: simulation and synthesis of digital systems.

Design Flow



- **SPECIFICATION**

This is the stage at which we define what are the important parameters of the system/design that you are planning to design. A simple example would be: I want to design a counter; it should be 4 bit wide, should have synchronous reset, with active high enable; when reset is active, counter output should go to "0".

- **HIGH LEVEL DESIGN**

This is the stage at which you define various blocks in the design and how they communicate. Let's assume that we need to design a microprocessor: high level design means splitting the design into blocks based on their function; in our case the blocks are registers, ALU, Instruction Decode, Memory Interface, etc.

➢ MICRO DESIGN/LOW LEVEL DESIGN

Low level design or Micro design is the phase in which the designer describes how each block is implemented. It contains details of State machines, counters, Mux, decoders, internal registers. It is always a good idea to draw waveforms at various interfaces. This is the phase where one spends lot of time.

➢ RTL CODING

In RTL coding, Micro design is converted into Verilog/VHDL code, using synthesizable constructs of the language. Normally we like to lint the code, before starting verification or synthesis.

➢ SIMULATION

Simulation is the process of verifying the functional characteristics of models at any level of abstraction. We use simulators to simulate the Hardware models. To test if the RTL code meets the functional requirements of the specification, we must see if all the RTL blocks are functionally correct. To achieve this we need to write a **test bench**, which generates clk, reset and the required test vectors. We use the waveform output from the simulator to see if the DUT (Device Under Test) is functionally correct.

➢ SYNTHESIS

Synthesis is the process in which synthesis tools like design compiler or Synplify take RTL in Verilog or VHDL, target technology, and constrains as input and maps the RTL to target technology primitives. Synthesis tool, after mapping the RTL to gates, also do the minimal amount of timing analysis to see if the mapped design is meeting the timing requirements. **(Important thing to note is, synthesis tools are not aware of wire delays, they only know of gate delays).**

- **Formal Verification:** Check if the RTL to gate mapping is correct.
- **Scan insertion:** Insert the scan chain in the case of A.

- **PLACE & ROUTE**

The gate level net list from the synthesis tool is taken and imported into place and route tool in Verilog net list format. All the gates and flip-flops are placed; clock tree synthesis and reset is routed. After this each block is routed. The P&R tool output is a GDS file, used by foundry for fabricating the ASIC.

- **GATE LEVEL SIMULATION (OR) SDF/TIMING SIMULATION**

There is another kind of simulation, called **timing simulation**, which is done after synthesis or after P&R (Place and Route). Here we include the gate delays and wire delays and see if DUT works at rated clock speed.

- **POST SILICON VALIDATION**

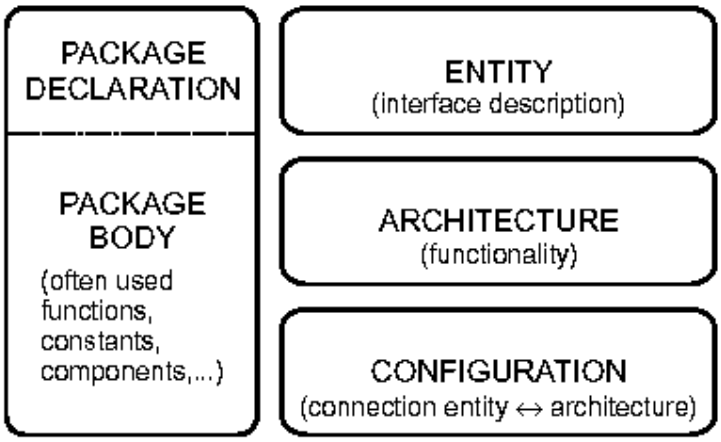
Once the chip (silicon) is back from fab, it needs to put in real environment and tested before it can be released into Market. Since the speed of simulation with RTL is very slow (number clocks per second), there is always possibility to find a bug in Post silicon validation.

Note: As design becomes complex, we write **SELF CHECKING TESTBENCH**, where test bench applies the test vector, then compares the output of DUT with expected values.

CHAPTER 2

BASIC COMPONENTS OF A VHDL MODEL

The purpose of VHDL descriptions is to provide a model for digital circuits and systems. This abstract view of the real physical circuit is referred to as entity. An entity normally consists of five basic elements, or design units.



In VHDL one generally distinguishes between the external view of a module and its internal description. The external view is reflected in the entity declaration, which represents an interface description of a 'black box'. The important part of this interface description consists of signals over which the individual modules communicate with each other.

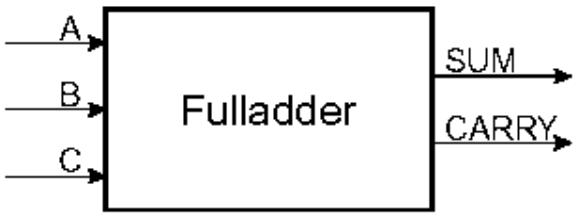
The internal view of a module and, therefore, its functionality is described in the architecture body. This can be achieved in various ways. One possibility is given by coding a behavioral description with a set of concurrent or sequential statements. Another possibility is a structural description, which serves as a base for the hierarchically designed circuit architectures. Naturally, these two kinds of architectures can also be combined. The lowest hierarchy level, however, must consist of behavioral descriptions. One of the major VHDL features is the capability to deal with multiple different architectural bodies belonging to the same entity declaration.

Being able to investigate different architectural alternatives permits the development of systems to be done in an efficient top-down manner. The ease of switching between different architectures has another advantage, namely, quick testing. In this case, it is necessary to bind one architecture to the entity in order to have a unique hierarchy for simulation or synthesis. Which architecture should be used for simulation or synthesis in conjunction with a given entity is specified in the configuration section. If the architecture body consists of a structural description, then the binding of architectures and entities of the instantiated submodules, the so-called components, can also be fixed by the configuration statement.

The package is the last element mentioned here. It contains declarations of frequently used data types, components, functions, and so on. The package consists of a package declaration and a package body. The declaration is used, like the name implies, for declaring the above-mentioned objects. This means, they become visible to other design units. In the package body, the definition of these objects can be carried out, for example, the definition of functions or the assignment of a value to a constant. The partitioning of a package into its declaration and body provides advantages in compiling the model descriptions.

Entity Declaration

An entity declaration specifies the name of an entity and its interface. This corresponds to the information given by the symbols in traditional design methods based on drawing schematics. Signals that are used for communication with the surrounding modules are called ports.



Interface of a full-adder module

Example:

```
entity FULLADDER is

  port ( A, B, C : in bit ;

        SUM, CARRY : out bit );

end FULLADDER;
```

The module FULLADDER has five interface ports. Three of them are the input ports A, B and C indicated by the VHDL keyword **in**. The remaining two are the output ports SUM and CARRY indicated by **out**. The signals going through these ports are chosen to be of the type bit. This is one of the predefined types besides integer, real and others types provided by VHDL. The type bit consists of the two characters '0' and '1' and represents the binary logic values of the signals.

Every port declaration implicitly creates a signal with the name and type specified. It can be used in all architectures belonging to the entity in one of the following port modes:

in: The port can only be read within the entity and its architectures.

out: This port can only be written.

inout: This port can be read and written. This is useful for modeling bus systems.

buffer: The port can be read and written. Each port must have only one driver.

Syntax :

entity entity name **is**

[generics]

[ports]

[declarations (types, constants, signals)]

[definitions (functions, procedures)]

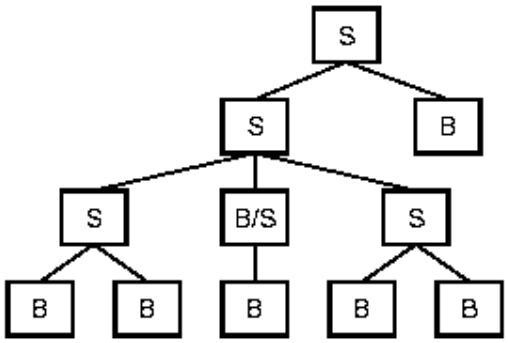
[**begin** -- normally not used

statements]

end [entity name] **;**

Architecture

The second important component of a VHDL description is the architecture. This is where the functionality and the internal implementation of a module are described. In general, a complex hierarchically structured system may have the topology.



Hierarchical circuit design

S: structural description

B: behavioral description

B/S: mixed description

In order to describe such a system both behavioral and structural descriptions are required. A behavioral description may be of either concurrent or sequential type. Overall, VHDL architectures can be classified into the three main types:

- Data flow modeling.
- Behavioral modeling.
- Structural modeling.

Syntax :

architecture architecture name **of** entity name **is**

[arch declarative part]

begin

[arch statement part]

end [architecture name] **;**

The architecture specifies the implementation of the entity entity name. A label architecture name must be assigned to the architecture. In case there are multiple architectures associated with one entity this label is then used within a configuration

Statement to bind one particular architecture to its entity. The architecture block consists of two parts: the arch declarative part before the keywords **begin** and the arch statement part after the keywords **begin**. In the declaration part local types, signals, components etc. are declared and subprograms are defined. The actual model description is done in the statement part. In contrast to programming languages like C, the major concern of VHDL is describing hardware which primary works in parallel and not in a sequential manner. Therefore, a special simulation algorithm is used to achieve a virtual concurrent processing. This algorithm is explained in the following section.

Configuration :

It is used to create a configuration for an entity. To binding of components used in the selected architecture body to other entities.

Package Declaration :

It contains a set of declarations that may possibly be shared by many design units.

Package Body :

It contains the behavior of the subprogram and the values of the deferred constants declared in a package declaration.

CHAPTER 3

BASIC LANGUAGE ELEMENTS

This describes the facilities in VHDL, which are drawn from the familiar programming language repertoire. If you are familiar with the Ada programming language, you will notice the similarity with that language. This is both a convenience and a nuisance. The convenience is that you don't have much to learn to use these VHDL facilities. The problem is that the facilities are not as comprehensive as those of Ada, though they are certainly adequate for most modeling purposes.

Lexical Elements

Comments

Comments in VHDL start with two adjacent hyphens ('--') and extend to the end of the line. They have no part in the meaning of a VHDL description.

Identifiers

Identifiers in VHDL are used as reserved words and as programmer-defined names. They must conform to the rule:

Letter { [underline] letter_or_digit }

Note that case of letters is not considered significant, so the identifiers cat and Cat are the same. Underline characters in identifiers are significant, so This_Name and ThisName are different identifiers.

Numbers

Literal numbers may be expressed either in decimal or in a base between two and sixteen. If the literal includes a point, it represents a real number, otherwise it represents an integer. Decimal literals are defined by:

integer [integer] [exponent]

Example :

```
 0      1     123_456_789        987E6        -- integer literals

0.0     0.5      2.718_28          12.4E-9       -- real literals
```

Based literal numbers are defined by:

base # based_integer [based_integer] # [exponent]

The base and the exponent are expressed in decimal. The exponent indicates the power of the base by which the literal is multiplied. The letters A to F (upper or lower case) are used as extended digits to represent 10 to 15.

Example :

```
2#1100_0100#   16#C4#    4#301#E1          -- the integer 196

2#1.1111_1111_111#E+11    16#F.FF#E2     -- the real number 4095.0
```

Characters

Literal characters are formed by enclosing an ASCII character in single-quote marks.

Example :

```
'A' '*' '"' ' '
```

Strings

Literal strings of characters are formed by enclosing the characters in double-quote marks. To include a double-quote mark itself in a string, a pair of double-quote marks must be put together. A string can be used as a value for an object which is an array of characters.

Example :

```
"A string"

""                                  -- empty string

"A string in a string: ""A string"". "   -- contains quote marks
```

Bit Strings

VHDL provides a convenient way of specifying literal values for arrays of type bit.

Syntax :

```
base_specifier " bit_value "
```

Base specifier B stands for binary, O for octal and X for hexadecimal.

Examples :

B"1010110" -- length is 7

O"126" -- length is 9, equivalent to B"001_010_110"

X"56" -- length is 8, equivalent to B"0101_0110"

Data Types and Objects

VHDL provides a number of basic, or *scalar*, types, and a means of forming *composite* types. The scalar types include numbers, physical quantities, and enumerations and there are a number of standard predefined basic types. The composite types provided are arrays and records. A data type can be defined by a type declaration:

type identifier **is** type_definition **;**

Type_definition :

scalar_type_definition

composite_type_definition

Scalar_type_definition :

integer_type_definition

physical_type_definition

floating_type_definition

enumeration_type_definition

Composite_type_definition :

array_type_definition

record_type_definition

Integer Types

An integer type is a range of integer values within a specified range.

Syntax :

type identifier **is range** range_constraint**;**

The expressions that specify the range must of course evaluate to integer numbers. Types declared with the keyword **to** are called *ascending* ranges, and those declared with the keyword **downto** are called *descending* ranges. The VHDL standard allows an implementation to restrict the range, but requires that it must at least allow the range –2147483647 to +2147483647.

Example :

type byte_int is range 0 to 255 ;

type signed_word_int is range –32768 to 32767 ;

type bit_index is range 31 downto 0 ;

There is a predefined integer type called integer. The range of this type is implementation defined, though it is guaranteed to include –2147483647 to +2147483647.

Physical Types

A physical type is a numeric type for representing some physical quantity, such as mass, length, time or voltage. The declaration of a physical type includes the specification of a base unit, and possibly a number of secondary units, being multiples of the base unit.

Syntax :

type identifier **is range** range_constraint

units

base_unit_declaration

{ secondary_unit_declaration }

end units

Example :

```
type length is range 0 to 1E9
    units
        um;
        mm = 1000 um;
        cm = 10 mm;
        m = 1000 mm;
        in = 25.4 mm;
        ft = 12 in;
        yd = 3 ft;
        rod = 198 in;
        chain = 22 yd;
```

```
        furlong = 10 chain;
    end units;

type resistance is range 0 to 1E8
    units
        ohms;
        kohms = 1000 ohms;
        Mohms = 1E6 ohms;
    end units;
```

The predefined physical type time is important in VHDL, as it is used extensively to specify delays in simulations. Its definition is:

```
type time is range implementation_defined
    units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr = 60 min;
    end units;
```

Floating Point Types

A floating point type is a discrete approximation to the set of real numbers in a specified range. The precision of the approximation is not defined by the VHDL language standard, but must be at least six decimal digits. The range must include at least –1E38 to +1E38.

Syntax :

type identifier **is range** range_constraint **;**

Examples :

type signal_level is range –10.00 to +10.00 ;

type probability is range 0.0 to 1.0 ;

There is a predefined floating point type called real. The range of this type is implementation defined, though it is guaranteed to include –1E38 to +1E38.

Enumeration Types

An enumeration type is an ordered set of identifiers or characters. The identifiers and characters within a single enumeration type must be distinct, however they may be reused in several different enumeration types.

Syntax :

type identifier **is (** enumeration_literal) **;**

Example :

type logic_level is (unknown, low, undriven, high);

type alu_function is (disable, pass, add, subtract, multiply, divide);

type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');

There are a number of predefined enumeration types, defined as follows:

```
type severity_level is (note, warning, error, failure);

type boolean is (false, true);

type bit is ('0', '1');

type character is ( NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS,
                    HT, LF, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3,
                    DC4, NAK, SYN, ETB, CAN,EM, SUB, ESC, FSP,
                    GSP, RSP, USP, ' ', '!', '"', '#', '$', '%', '&', ''',  '(', ')',
                    '*', '+', ',', '-', '.', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                    ':', ';', '<', '=', '>', '?', '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
                    'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O','P', 'Q', 'R', 'S', 'T', 'U',
                    'V', 'W',   'X', 'Y', 'Z', '[', '\', ']', '^', '_', '`', 'a', 'b', 'c', 'd',
                    'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
                    'u', 'v', 'w', 'x', 'y', 'z', '{', '|', '}', '~', DEL);
```

Note that type character is an example of an enumeration type containing a mixture of identifiers and characters. Also, the characters '0' and '1' are members of both bit and character . Where '0' or '1' occur in a program, the context will be used to determine which type is being used.

Arrays

An array in VHDL is an indexed collection of elements all of the same type. Arrays may be one-dimensional (with one index) or multidimensional (with a number of indices). In addition, an array type may be constrained, in which the bounds for an index are established when the type is defined, or unconstrained, in which the bounds are established subsequently.

Syntax :

type identifier **is array** index_constraint **of** *element*_subtype_indication**;**

Example :

```
type word is array (31 downto 0) of bit;

type memory is array (address) of word;

type transform is array (1 to 4, 1 to 4) of real;

type register_bank is array (byte range 0 to 132) of integer;
```

An example of an unconstrained array type declaration:

```
type vector is array (integer range <>) of real;
```

The symbol '<>' (called a box) can be thought of as a place-holder for the index range, which will be filled in later when the array type is used. For example, an object might be declared to be a vector of 20 elements by giving its type as:

```
vector(1 to 20)
```

There are two predefined array types, both of which are unconstrained. They are defined as:

```
type string is array (positive range <>) of character;

type bit_vector is array (natural range <>) of bit;
```

The types positive and natural are subtypes of integer. The type bit_vector is particularly useful in modeling binary coded representations of values in simulations of digital systems. An element of an array object can referred to by indexing the name of the object. For example, suppose a and b are one and two-dimensional array objects respectively. Then the indexed names a(1) and b(1, 1) refer to elements of these arrays. Furthermore, a contiguous slice of a one-dimensional array can be referred to by using a range as an index. For example a (8 **to** 15) is an eight-element array which is part of the array . Sometimes you may need to write a literal value of an array type. This can be done using an array aggregate, which is a list of element values. Suppose we have an array type declared as:

```
type a is array (1 to 4) of character;
```

and we want to write a value of this type containing the elements 'f', 'o', 'o', 'd' in that order. We could write an aggregate with *positional* association as follows:

```
('f', 'o', 'o', 'd')
```

In this case, the index for each element is explicitly given, so the elements can be in any order. Positional and named association can be mixed within an aggregate, provided all the positional associations come first. Also, the word **others** can be used in place of an index in a named association, indicating a value to be used for all elements not explicitly mentioned. For example, the same value as above could be written as:

```
('f', 4 => 'd', others => 'o')
```

Records

VHDL provides basic facilities for records, which are collections of named elements of possibly different types.

Syntax :

```
type identifier is
    record
        element_declaration
        { element_declaration }
    end record
```

Example :

```
type instruction is
    record
        op_code : processor_op ;
        address_mode : mode ;
        operand1, operand2: integer range 0 to 15 ;
    end record;
```

When you need to refer to a field of a record object, you use a selected name. For example, suppose that r is a record object containing a field called f. Then the name r.f refers to that field. As for arrays, aggregates can be used to write literal values for records. Both positional and named association can be used, and the same rules apply, with record field names being used in place of array index names.

Subtypes

The use of a subtype allows the values taken on by an object to be restricted or constrained subset of some base type.

Syntax :

subtype identifier **is** [*resolution_function_*name] range [constraint] ;

There are two cases of subtypes. Firstly a subtype may constrain values from a scalar type to be within a specified range.

Example :

```
subtype pin_count is integer range 0 to 400;

subtype digits is character range '0' to '9';
```

Secondly, a subtype may constrain an otherwise unconstrained array type by specifying bounds for the indices.

Example :

```
subtype id is string(1 to 20);

subtype word is bit_vector(31 downto 0);
```

There are two predefined numeric subtypes, defined as:

```
subtype natural is integer range 0 to highest_integer ;

subtype positive is integer range 1 to highest_integer ;
```

Object Declarations

An object is a named item in a VHDL description which has a value of a specified type. There are three classes of objects:

- Constants
- Variables
- Signals

Constants

Declaration and use of constants and variables is very much like their use in programming languages. A constant is an object which is initialised to a specified value when it is created, and which may not be subsequently modified.

Syntax :

constant identifier_list : subtype_indication [:= expression] ;

Constant declarations with the initialising expression missing are called deferred constants, and may only appear in package declarations. The initial value must be given in the corresponding package body.

Example :

```
constant e : real := 2.71828;

constant delay : Time := 5 ns;

constant max_size : natural;
```

Variables

A variable is an object whose value may be changed after it is created.

Syntax :

variable identifier_list : subtype_indication [:= expression] ;

The initial value expression, if present, is evaluated and assigned to the variable when it is created. If the expression is absent, a default value is assigned when the variable is created. The default value for scalar types is the leftmost value for the type, that is the first in the list of an enumeration type, the lowest in an ascending range, or the highest in a descending range. If the variable is a composite type, the default value is the composition of the default values for each element, based on the element types.

Example :

```
variable count : natural := 0;

variable trace : trace_array;
```

Assuming the type trace_array is an array of boolean, then the initial value of the variable trace is an array with all elements having the value false.

Signals

Signals represent wires in a logic circuit. Signals can be declared in all declarative regions in VHDL except for functions and procedures. Assignments to signals are not immediate, but scheduled to be executed after a delta delay.

Syntax :

```
signal identifier_list : subtype_indication [ := expression ] ;
```

Example :

```
signal foo : bit_vector (0 to 5) := B"000000" ;

signal aux : bit ;

signal max_value : integer ;
```

The declaration assigns a name to the signal foo ; a type, with or without a range restriction (bit_vector(0 to 5)); and optionally an initial value. Initial values on signals are usually ignored by synthesis. Signals can be assigned values using an assignment statement (e.g., aux <= '0' ;). If the signal is of an array type, elements of the signal's array can be accessed and assigned using indexing or slicing methods.

Expressions and Operators

Expressions in VHDL are much like expressions in other programming languages. An expression is a formula combining primaries with operators. Primaries include names of objects, literals, function calls and parenthesized expressions.

Type	Operators						Precedence
Logical	and	or	nand	nor	xor	xnor	Lowest
Relational	=	/=	<	<=	>	>=	
Adding	+	-	&				
Unary (sign)	+	-					
Multiplying	*	/	mod	rem			
Miscellaneous	**	abs	not				Highest

Operators and precedence

The logical operators **and**, **or**, **nand**, **nor**, **xor** and **not** operate on values of type bit or Boolean, and also on one-dimensional arrays of these types. For array operands, the operation is applied between corresponding elements of each array, yielding an array of the same length as the result. For bit and Boolean operands, **and**, **or**, **nand**, and **nor** are 'short-circuit' operators, that is they only evaluate their right operand if the left operand does not determine the result. So **and** and **nand** only evaluate the right operand if the left operand is true or '1', and **or** and **nor** only evaluate the right operand if the left operand is false or '0'.

The relational operators =, /=, <, <=, > and >= must have both operands of the same type, and yield Boolean results. The equality operators (= and /=) can have operands of any type. For composite types, two values are equal if all of their corresponding elements are equal. The remaining operators must have operands which are scalar types or one-dimensional arrays of discrete types.

The sign operators (+ and –) and the addition (+) and subtraction (–) operators have their usual meaning on numeric operands. The concatenation operator (&) operates on one-dimensional arrays to form a new array with the contents of the right operand following the contents of the left operand. It can also concatenate a single new element to an array, or two individual elements to form an array. The concatenation operator is most commonly used with strings.

The multiplication (*) and division (/) operators work on integer, floating point and physical types. The modulus (**mod**) and remainder (**rem**) operators only work on integer types. The absolute value (**abs**) operator works on any numeric type. Finally, the exponentiation (**) operator can have an integer or floating point left operand, but must have an integer right operand. A negative right operand is only allowed if the left operand is a floating point number.

CHAPTER 4

MODELING TYPES

DATAFLOW MODELING

This kind of description specifies a dataflow through the entity based on concurrent signal assignment statements. A structure of the entity is not explicitly defined by this description but can be derived from it. As an example, consider the following implementation of the entity FULLADDER.

Example :

```
architecture CONCURRENT of  FULLADDER is

    begin

        SUM <= A xor B xor C after 5 ns;

        CARRY <= (A and B) or (B and C) or (A and C) after 3 ns;

end CONCURRENT;
```

Two concurrent signal assignment statements describe the model of the entity FULLADDER. The symbol <= indicates the signal assignment. This means that the value on the right side of the symbol is calculated and subsequently assigned to the signal on the left side. A concurrent signal assignment is executed whenever the value of a signal in the expression on the right side changes. In general, a change of the current value of a signal is called an event. Due to the fact that all signals used in this example are declared as ports in the entity declaration.

The arch declarative part remains empty. Information about a possibly existing delay time of the modeled hardware is provided by the after clause. If there is an event

on one of the inputs A, B or C at time T, the expression A xor B xor C is computed at this time T, but the target signal (the output SUM) is scheduled to get this new value at time T + 5 ns. The signal assignment for CARRY is handled in exactly the same way except for the smaller delay time of 3 ns. If explicit information about the delay time is missing then it is assumed to be 0 ns by default. Nevertheless, during the VHDL simulation the signal assignment is executed after an infinitesimally small delay time, the so-called delta-delay. This means that the signal assignment is executed immediately after an event on a signal on the right side is detected and the calculation of the new expression value is performed.

Syntax :

[label **:**] signal name **<=** [**transport**] expression [**after** time expression] **;**

Up to now the label was not used. With this element it is possible to assign a label to the statement, which can be useful for documentation. Furthermore, it is possible to assign several events with different delay times to the target signal. In this case the values to be assigned and their delay times have to be sorted in ascending order. The keyword transport affects the handling of multiple signal events coming in short time one after another.

Example :

```
architecture VER1 of MUX is

    begin

        OUTPUT <= A ;

end VER1;
```

Conditional Signal Assignment statement

In this case there are different assignment statements related to one target signal. The selection of one assignment statement is controlled by a set of conditions condition. The conditional signal assignment statement can be compared with the well-known if - elsif - else structure.

Syntax :

[label **:**] signal name **<=** expression **when** condition **else**

expression **when** condition **else**

expression **;**

The conditional signal assignment is activated as soon as one of the signals belonging to the condition or expression changes.

Example :

```
Z <=  A  when (X > 3)  else
      B  when (X < 3)  else
      C ;
```

Each time one signal either in expression or condition changes its value the complete statement is executed. Starting with the first condition, the first true one selects the expression that is computed and the resulting value is assigned to the target signal signal name.

Selected Signal Assignment statement

With this statement a choice between different assignment statements is made. The selection of the right assignment is done by the value of select expression. The statement resembles a case structure.

Syntax :

```
[ label : ]  with  select_expression  select
        signal name  <=  expression  when  value ,
                         expression  when  value ,
                         expression  when  others ;
```

The selected signal assignment is activated as soon as one of the signals belonging to the selection condition or expression changes.

Example :

```
with  MYSEL  select
    Z <= A when 15 ,
         B when 22 ,
         C when others ;
```

Unaffected Statement

No action to be take place in a sequential statement and execution continues with the next statement. It is represented by using the keyword **unaffected**. It is used in a conditional or selected signal assignment statement where, for certain conditions. It may be useful or necessary to explicitly specify that no action needs to be performed.

Example :

```
with mux_sel select
    Z <= A when "00",
         B when "01",
         C when "10",
         unaffected when others ;
```

Block statement

A block statement defines an internal block representing a portion of a design. Blocks may be hierarchically nested to support design decomposition.

In order to efficiently group concurrent assignments, block statements may be used. A block may contain declarations of data types, signals, and so on, all of which are locally used. The body of the block statement contains any of the concurrent statements mentioned previously.

A guarded block contains an additional boolean expression guard expression, which drives an implicit signal GUARD of boolean type. This signal can be used within a block for the control of concurrent assignments. If concurrent statements have an associated GUARD signal, they are known as Guarded Signal Assignments.

Syntax :

```
Label : block [ ( guard expression ) ]
        [ use clause ]
        [ subprogram decl , subprogram body ]
        [ type decl ]
```

```
        [ subtype decl ]

        [ constant decl ]

        [ signal decl ]

        [ component decl ]

    begin

        [ concurrent statements ]

end  block  [ label ] ;
```

Guarded Signal Assignment is a special form of the concurrent assignment. The assignment is activated after the GUARD signal, which must be of the boolean type, is evaluated to true. The GUARD signal can be explicitly declared and used; however, it is more common to use it implicitly within a Guarded Block.

Syntax :

```
[ label : ] signal name <=  guarded expression [ after time expr ] ;
```

Example :

```
U1 : block ( clk='1' and not clk'stable )

        signal temp  :  std_logic  ;

    begin

        temp  <=  guarded D ;

          Q  <=  temp ;

          Q'  <=  not temp

end block  U1 ;
```

Any declarations appearing within the block are visible only within the block, that is,

between block ……. end block.

BEHAVIORAL MODELING

Behavioral descriptions are based on the process environment. A process statement as a whole is treated as a concurrent statement within the architecture. Therefore, in the simulation time a process is continuously executed and it never gets finished. The statements within the process are executed sequentially without the advance of simulation time. To ensure that simulation time can move forward every process must provide a means to get suspended. Thus, a process is constantly switching between the two states: the execution phase in which the process is active and the statements within this process are executed, and the suspended state. The change of state is controlled by two mutually exclusive implementations:

- With a list of signals in such a manner that an event on one of these signals invokes a process. This can be compared with the mechanism used in conjunction with concurrent signal assignment statements. There, the statement is executed whenever a signal on the right side of the assignment operator <= changes its value. In case of a process, it becomes active by an event on at least one signal belonging to the sensitivity list. All statements between the keywords begin and end process are then executed sequentially.

Syntax :

[proc label **:**] **process (** sensitivity list **)**

[proc declarative part]

begin

[sequential statement part]

end process [proc label] **;**

The sensitivity list is a list of signal names within round brackets, for Example

(A, B, C **)**.

- With wait statements in such a way that the process is executed until it reaches a wait statement. At this instance it gets explicitly suspended. The statements within the process are handled like an endless loop which is suspended for some time by a wait statement.

Syntax :

[process label **:**] **process**

[proc declarative part]

begin

[sequential statements]

wait ...**;** -- at least one wait statement

[sequential statements]

end process [proc label] **;**

The structure of a process statement is similar to the structure of an architecture. In the process declarative part various types, constants and variables can be declared; functions and procedures can be defined. The sequential statement part contains the description of the process functionality with ordered sequential statements. An implementation of the full adder with a sequential behavioral description is given below:

Example :

```
architecture  SEQUENTIAL of  FULLADDER is
      begin
        process (A, B, C)
            variable TEMP : integer;
            variable SUM CODE : bit vector(0 to 3) := "0101";
            variable CARRY CODE : bit vector(0 to 3) := "0011";
         begin

         if A = '1' then
                        TEMP := 1;
            else
                        TEMP := 0;
        end if;
         if B = '1' then
                       TEMP := TEMP + 1;
         end if;
```

```
        if C = '1' then
                TEMP := TEMP + 1;
        end if;          -- variable TEMP now holds the number of ones
            SUM <= SUM CODE(TEMP);
         CARRY <= CARRY CODE(TEMP);
    end process;
end SEQUENTIAL;
```

The functionality of this behavioral description is based upon a temporary variable TEMP which counts the number of ones on the input signals. With this number one element, or one bit, is selected from each of the two predefined vectors SUM CODE and CARRY CODE. The initialization of these two vectors reacts the truthtable of a full-adder module. The reason for this unusual coding is the attempt to explain the characteristics of a variable. A variable differs not only in the assignment operator (:=) from that of a signal (<=). It is also different with respect to time when the new computed value becomes valid and, therefore, readable to other parts of the model.

Every variable gets the new calculated value immediately, whereas the new signal value is not valid until the beginning of the next delta-cycle, or until the specified delay time elapses. If the above example had been coded with a signal as the temporary counter instead of the variable, then the correct functionality of this architecture as a full adder could not be ensured. After an event at time T on one of the input signals A, B or C, which are members of the sensitivity list, the process is executed once. The simulation continues with executing the second if statement at time T because computing a sequential statement does not advance the simulation time. Therefore, the signal TEMP still holds the same value it had before the process activation! This means that the intended counting of ones does not work with TEMP declared as signal.

In general, signal assignment statements within a process have to be handled with care, especially if the target signal will be read or rewritten in the following code before the process gets suspended. If this effect is taken into consideration, the process statement provides an environment in which a person familiar with programming languages like C or Pascal can easily generate a VHDL behavioral description. This remark, however, should not be understood that the process statement is there for people switching to VHDL. In reality, some functions can be implemented much more easily in a sequential manner.

Example :

```
architecture SEQUENTIAL of DFF is
    begin
        process (CLK, NR)
            begin
                if (NR = '0') then
                    Q <= (others => '0');
                elsif (CLK'event and CLK = '1') then
                    Q <= D;
                end if;
        end process;
end SEQUENTIAL;
```

In the above example, the attribute CLK'event is used to detect an edge on the CLK signal. This is equivalent to an event on CLK. The ability to detect edges on signals is based upon the storage of all events in event queues for every signal. Therefore, old values can be compared with the actual ones or even read. In contrast, variables always get the new assigned value immediately and the old value is not stored. Subsequently, during the simulation more memory is required for a signal for a variable. In complex system descriptions this fact should be taken into consideration.

Sequential Signal Assignment statement

The syntax of a sequential signal assignment is very similar to the concurrent assignment statement, except for a label which can not be used.

Syntax :

signal name **<=** [**transport**] expression [**after** time expr] **;**

Variable Assignment statement

A variable assignment statement is very similar to a signal assignment. As already mentioned, a variable differs from a signal in that it gets its new value immediately upon assignment. Therefore, the specification of a delay time in a variable assignment is not possible. Attention must be paid to the assignment operator which is := for a variable and <= for a signal.

Syntax :

variable name **:=** expression **;**

Wait statement

This statement may only be used in processes without a sensitivity list. The purpose of the wait statement is to control activation and suspension of the process.

Syntax :

Wait [**on** signal names] **;**

Wait [**until** condition]

Wait [**for** time expression] **;**

The arguments of the wait statement have the following interpretations:

- on signal names:

The process gets suspended at this line until there is an event on at least one signal in the list signal names. The signal names are separated by commas; brackets are not used. It can be compared to the sensitivity list of the process statement.

- until condition:

The process gets suspended until the condition becomes true.

- for time expression:

The process becomes suspended for the time specified by time expression.

- without any argument:

The process gets suspended until the end of the simulation.

A sensitivity list of a process is functionally equivalent to the wait on ... appearing at the end of the process. There are four different ways to use the wait-statement:

```
wait on A, B;
```

suspends a process until an occurrence of a change is registered. Here, execution will resume when a new event is detected on either signal A or B.

```
wait until X > 10;
```

suspends a process until the condition is satisfied; in this case, until the value of a signal is > 10.

```
wait for 10 ns;
```

suspends a process for the time specified; here, until 10 ns of simulation time elapses.

```
wait;
```

suspends a process indefinitely. . . Since a VHDL-process is always active, this statement at the end of a process is the only way to suspend it. This technique may be used to execute initialization processes only once.

The example below models an architecture, which simulates a Producer/Consumer problem using two processes. The processes are synchronized through a simple handshake protocol, which has two wires, each with two active states.

Example :

```
entity  PRO CON  is
   ...
end  PRO CON ;
```

```
architecture BEHAV of PRO CON is
        signal PROD: boolean := false;    --item produces a semaphore
        signal CONS: boolean := true;     --item consumes a semaphore
      begin
          PRODUCER: process producer model
            begin
                  PROD <= false;
                  wait until CONS;                  ----produce item
                  PROD <= true;
                  wait until not CONS;
          end process;
        CONSUMER: process consumer model
            begin
                CONS <= true;
                 wait until PROD;
                CONS <= false;
                ...consume item
                wait until not PROD;
        end process;
end BEHAV;
```

If-else statement

This branching statement is equivalent to the ones found in other programming languages and, therefore, needs no further explanation.

Syntax :

if condition **then**

sequential statements **;**

[**elsif** condition **then**

sequential statements **;**]

[**else**

sequential statements **;**]

end if ;

Case statement

This statement is also identical to its corresponding equivalent found in other programming languages.

Syntax :

case expression **is**

when choices **=>** sequential statements **;**

[**when** others **=>** sequential statements **;**]

end case ;

Either all-possible values of expression must be covered with choices or the case statement has to be completed with an others branch.

Example :

```
case  BCD  is                          ------Decoder:  BCD to 7-Segment
        when "0000" => LED := "1111110";
        when "0001" => LED := "1100000";
        when "0010" => LED := "1011011";
        when "0011" => LED := "1110011";
        when "0100" => LED := "1100101";
        when "0101" => LED := "0110111";
        when "0110" => LED := "0111111";
        when "0111" => LED := "1100010";
```

```
        when "1000" => LED := "1111111";
        when "1001" => LED := "1110111";
        when others => LED := "-------";          ------don't care
    end  case;
```

Null statement

This statement is used for an explicit definition of branches without any further commands. Therefore, it is used primarily in case statements, and also in if clauses.

Syntax :

null ;

Loop statement

This is a conventional loop structure found in other programming languages.

Syntax :

[loop label **:**] **while** condition **loop** | --controlled by condition

for identifier **in** value1 to | downto value2 **loop** | --with counter

loop --endless loop

sequential statements

end loop [loop label] **;**

Example :

```
J := 0 ;

U1 : while J < 20 loop

     J := J + 2 ;

end loop U1 ;
```

The while...loop statement has a Boolean iteration scheme. If the iteration condition evaluates true, executes the enclosed statements once. The iteration condition is then reevaluated. As long as the iteration condition remains true, the loop is repeatedly executed. When the iteration condition evaluates false, the loop is skipped and execution continues with the next loop iteration.

```
for i in 0 to 3 loop

     Z ( i ) := A ( i ) and B ( i ) ;

end loop ;
```

The for...loop statement has an integer iteration scheme. The integer range determines the number of repetitions.

```
Z : = 2 ;   sum : = 1 ;

V1 :  loop

      Z : = Z  +  3  ;

       sum : = sum * 5  ;

   exit  when  sum  > 100  ;

end loop  V1  ;
```

The basic loop statement has no iteration scheme. It executes enclosed statements repeatedly until it encounters an exit or next statement.

Exit and Next statement

With these two statements a loop iteration can be terminated before reaching the keyword end loop. With next the remaining sequential statements of the loop are skipped and the next iteration is started at the beginning of the loop. The exit directive skips the remaining statements and all remaining loop iterations. In nested loops both statements skip the innermost enclosing loop if loop label is left out. Otherwise, the loop labeled loop label is terminated. The optional condition expression can be specified to determine whether or not to execute these statements.

Syntax :

next [loop label] [**when** condition] **;**

Example :

```
for I in 0 to MAX LIM loop

  if (DONE(I) = true) then

          next;                    -----Jump to end loop

  end if;
```

```
            Q(I) <= A(I);
        end loop;
            L1: while J < 10 loop outer loop
            L2: while K < 20 loop inner loop...
                next L1 when J = K; jump out of the inner loop...
        end loop L2;
    end loop L1;                    ------jump destination
```

Syntax :

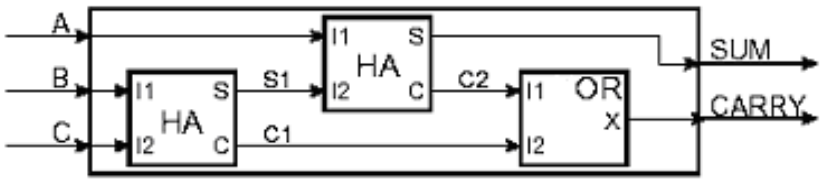
exit [loop label] [**when** condition] **;**

Example:

```
    for I in 0 to MAX LIM loop
        if (Q(I) <= 0) then
            exit;                   ------jump out of the loop
        end if;
            Q(I) <= (A * I);
    end loop;                       ------jump destination
```

STRUCTURAL DESCRIPTION

In structural descriptions the implementation of a system or model is described as a set of interconnected components, which is similar to drawing schematics. Such a description can often be generated with a VHDL netlister in a graphical development tool. Since there are many different ways to write structural descriptions, to explain all of them in one section would be more confusing than enlightening. Therefore, only one alternative approach is presented here.



Structural implementation of a full adder

As an introductive example, consider the implementation of a full-adder circuit. The components HA and XOR are assumed to be predefined elements.

Example :

```
architecture STRUCTURAL of FULLADDER is
    signal  S1, C1, C2  : bit ;
        component HA
            port (  I1, I2 : in bit ;  S, C : out bit ) ;
        end component ;
        component OR
            port (  I1, I2 : in bit ;  X  : out bit ) ;
        end component;
    begin
      INST HA1 : HA port map ( I1 => B, I2 => C, S => S1, C => C1 ) ;
      INST HA2 : HA port map ( I1 => A, I2 => S1, S => SUM, C => C2 ) ;
      INST OR : OR port map ( I1 => C2, I2 => C1, X => CARRY) ;
end STRUCTURAL ;
```

Component declaration

In the declarative part of the architecture, all objects which are not yet known to the architecture have to be declared. In the example above, these are the signals S1, C1 and C2 used for connecting the components together, excluding the ports of the entity FULLADDER. In addition, the components HA and XOR have to be declared. The declaration of a component consists of declaring its interface ports and generics to the actual model.

Often used components could be selected from a library of gates defined in a package and linked to the design. In this case the declaration of components usually is done in the package, which is visible to the entity. Therefore, no further declaration of the components is required in the architecture declarative part.

The actual structural description is done in the statement part of the architecture by the instantiation of components. The components' reference names INST HA1, INST HA2 and INST XOR, also known as instance names, must be unique in the architecture. The port maps specify the connections between different components, and between the components and the ports of the entity. Thus, the components' ports (so-called formals) are mapped to the signals of the architecture (so-called actuals) including the signals of the entity ports. For example, the input port I1 of the half adder INST HA1 is connected to the entity input signal B, input port I2 to C, and so on. The instantiation of a component is a concurrent statement. This means that the order of the instances within the VHDL code is of no importance.

Syntax :

```
component component name
  [ generic ( generic list : type name  [ := expression ]  ; |
              generic list : type name  [ := expression ]  ) ; ]
  [ port ( signal list : mode type name ;
           signal list : mode type name ); ]
end component ;
```

Component instantiation

A component instantiation statement defines a subcomponent of the design entity in which it appears, associates signals or values with the ports of that subcomponent, and associates values with generics of that subcomponent. This subcomponent is one instance of a class of components defined by a corresponding component declaration, design entity, or configuration declaration.

Syntax :

component label **:** component name **port map (** Association-list **) ;**

The Association of ports to the connecting signals during the instantiation can be done through the positional notation. Alternatively, it may be done by using the named notation, using the already familiar format

Two types of association

- Positional Association
- Named Association

Positional Association

Each actual in the component instantiation is mapped by position with each port in the component declaration. The ordering of the actuals is therefore important.

An association-list form

actual1, actual2 , actual3, …… actualn

Example :

V1 : nand2 port map (S1, S2, S3) ;

If a port in a component instantiation is not connected to any signal. This purpose the keyword **Open** is used.

```
V1  : nand2  port map (  S1, open, S3 )  ;
```

Named Association

The ordering of the association is not important since the mapping between the actuals and formals is explicitly specified

An association-list form

```
formal1 => actual1, formal2 =>  actual2 ,……. formaln => actualn
```

Example :

```
component  nand2
    port ( A, B  :  in std_logic  ;
                C  :  out std_logic )  ;
end  component  ;
begin
V1: nand2 port map (A => S1, B =>S2, C => S3 ) ;
```

It is important to note that the symbol '=>' is used within a port map in contrast to the symbol '<=' used for concurrent or sequential signal assignment statements. If one of the ports has no signal connected to it, a reserved word open may be used. A function call can replace the signal name. This allows direct type conversions at the component instantiation.

CHAPTER 5

GENERICS AND CONFIGURATIONS

GENERICS

It allow static information to be communicated to a block from its environment for all architectures of a design unit. These include timing information like setup, hold, delay times, part sizes, and other parameters.

Syntax :

[**generic (** list-of-generics-and-their-types **) ;**]

It can be declared any one of the following :

- Entity Declaration
- Component Declaration
- Component Instantiation
- Configuration Specification
- Configuration Declaration

The generic size can be used inside the entity and in the architecture that matches the entity. In this example, the generic size is defined as an integer with an initial value 8. The sizes of the input and output ports of the entity increment are set to be 8 bits unless the value of the generic is overwritten by a generic map statement in the component instantiation of the entity.

Example :

```
entity increment is
    generic ( size : integer := 8 ) ;
    port ( ivec : in bit_vector (0 to size-1) ;
        ovec : out bit_vector (0 to size-1)) ;
end increment ;
```

The other ways of specifying the value of a generic are in a component instantiation.

```
U1 : and2 generic map (10) port map (  D, S1 ) ;

U2 : or2 generic map (M=>8) port map ( C, S2) ;
```

CONFIGURATIONS

Used to bind component instances to design entities and collect architectures to make, typically, a simulatable test bench. One configuration could create a functional simulation while another configuration could create the complete detailed logic design. With an appropriate test bench the results of the two configurations can be compared.

A configuration does not have any simulation semantics associated with it; it only specifies how a top-level entity is organized in terms of lower-level entities. The component names and the entity names, as well as the port names and their order, are different. The binding information can be specified using a configuration.

Two types of binding

- Configuration Specification
- Configuration Declaration

Configuration specification

To bind component instantiations to specific entities stored in design libraries. It appears in the declarations part of the architecture or block in which the components are instantiated

Syntax :

for list-of-comp-labels **:** component-name binding-indication **;**

The binding-indication specifies the entity represented by the entity-architecture pair, and the generic and port bindings, and one of its forms is

use entity entity-name [**(** architecture-name **)**]

[**generic map (** generic-association-list **)**]

[**port map (** port-association-list **)**] - - - - - - Form 1

The list of component labels may be replaced with the keyword all to denote all instance of a component; it may also be the keyword others to specify all as yet unbound instances of a component. The generic map is used to specify the values for the generics or provide the mapping between the generic parameters of the component and the entity to which it is bound. The port map is used to specify the port bindings between the component and the bound entity.

Example :

```
Library ieee;
 use ieee.std_logic_1164.all ;
    entity  HA
      port ( A, B   :  in std_logic  ;
          Sum,Ca  :  out std_logic )  ;
    end  HA  ;
architecture HA_str  of  HA  is
     component  xor2
       port ( A, B   :  in std_logic  ;
                 C  :  out std_logic )  ;
     end  component  ;
     component  and2
       port ( A, B   :  in std_logic  ;
                 C  :  out std_logic )  ;
     end  component  ;
```

```
    for X1 : xor2 use entity work.xor2 ( xor_arch );
    for A1 : and2 use entity work.and2 ( and_arch );
begin
  X1 : xor2 port map ( A, B, Sum ) ;
  A1 : and2 port map ( A, B, Ca) ;
end  HA_str ;
```

Configuration declarations

The binding of component instances to design entities is performed by configuration specifications; such specifications appear in the declarative part of the block in which the corresponding component instances are created. In certain cases, however, it may be appropriate to leave unspecified the binding of component instances in a given block and to defer such specification until later. A configuration declaration provides the mechanism for specifying such deferred bindings.

Syntax :

configuration identifier **of** *entity*_name **is**

configuration_declarative_part

use_clause

| attribute_specification

block_configuration

end [**configuration**] [configuration_simple_name] **;**

The entity name identifies the name of the entity declaration that defines the design entity at the apex of the design hierarchy. For a configuration of a given design entity, both the configuration declaration and the corresponding entity declaration must reside in the same library. If a simple name appears at the end of a configuration declaration, it must repeat the identifier of the configuration declaration.

NOTES

—A configuration declaration achieves its effect entirely through elaboration. There are no behavioral semantics associated with a configuration declaration.

—A given configuration may be used in the definition of another, more complex configuration.

Example :

— An architecture of a microprocessor:

```
architecture Structure_View of Processor is
    component ALU port ( ••• ); end component;
    component MUX port ( ••• ); end component;
    component Latch port ( ••• ); end component;
 begin
     A1: ALU port map ( ••• ) ;
     M1: MUX port map ( ••• ) ;
     M2: MUX port map ( ••• ) ;
     M3: MUX port map ( ••• ) ;
     L1: Latch port map ( ••• ) ;
     L2: Latch port map ( ••• ) ;
end Structure_View ;
```

```
— A configuration of the microprocessor:
  library TTL, Work ;
  configuration V_config of Processor is
   use Work.all ;
     for Structure_View
       for A1: ALU use configuration TTL.SN74LS181 ;
         end for ;
          for M1,M2,M3: MUX use entity Multiplex4 (Behavior) ;
            end for ;
              for all: Latch
                end for ;                    — use defaults
          end for ;
  end configuration V_config ;
```

A block configuration defines the binding of components in a block, where a block may be an architecture body, a block statement, or a generate statement.

Syntax of Block configuration :

```
for  block-name
     component-configurations
     block-configurations
end  for  ;
```

A block-name is the name of an architecture body, a block statement label, or a generate statement label.

Syntax of Component-configuration :

for list-of-comp-labels **:** component-name

[binding-indication **;**]

[block-configurations]

end for ;

There are two other forms of binding indication

use configuration configuration-name - - - - - - Form 2

use open - - - - - - Form 3

In Form 2, the binding indication specifies that the component instance are to be bound to a configuration of a lower-level entity as specified by the configuration name. In Form 3,the binding indication specifies that the binding is not yet specified and is to be deferred. Both these forms of binding indication may also be used in a configuration specification.

Example :

```
Library TTL ;
  configuration HA_config of HA is
    for HA_XA
      for X1 : xor2
        use entity work.Xor2 ( xor_2 ) ;
      end for ;
      for A1 : and2
        use entity work.and2 ( and_2 ) ;
      end for ;
    end for ;
  end HA_config ;
```

Direct Instantiation

It is possible to directly instantiate the entity-architecture pair or a configuration in a component instantiation statement. This saves the additional binding step necessary when using components. Two additional forms of the component instantiation statement that can be used to directly instantiate an entity or a configuration.

Syntax :

Component-label **:** **entity** entity-name

[**(** architecture-name **)**]

[**generic map (** generic-association-list **)**]

[**port map (** port-association-list **)**] **;**

Component-label **:** **configuration** config-name

[**generic map (** generic-association-list **)**]

[**port map (** port-association-list **)**] **;**

NOTE

No configuration declaration is necessary or possible in this case, since the component instantiations directly instantiate the appropriate entity-architecture pairs or configurations. No components declarations are necessary or possible.

Example :

```
architecture  HA_str  of  HA  is
begin
  X1 : entity work.xor2 ( xor2 ) port map ( A, B, S ) ;
  A1 : configuration TTL.and2  port map ( A, B,C ) ;
end HA_str ;
```

CHAPTER 6

SUBPROGRAMS AND PACKAGES

SUBPROGRAMS

There are two kinds of subprograms: procedures and functions. Both procedures and functions written in VHDL must have a body and may have declarations.

Procedures perform sequential computations and return values in global objects or by storing values into formal parameters.

Functions perform sequential computations and return a value as the value of the function. Functions do not change their formal parameters.

Subprograms may exist as just a procedure body or a function body. Subprograms may also have a procedure declarations or a function declaration. When subprograms are provided in a package, the subprogram declaration is placed in the package declaration and the subprogram body is placed in the package body.

Procedure Declaration

Syntax :

procedure identifier [(formal parameter list)] ;

Formal parameters are separated by semicolons in the formal parameter list. Each formal parameter is essentially a declaration of an object that is local to the procedure. The type definitions used in formal parameters must be visible at the place where the procedure is being declared. No semicolon follows the last formal parameter inside the parenthesis. Formal parameters may be constants, variables, signals or files. The default is variable. Formal parameters may have modes **in**, **inout** and **out** Files do not have a mode. The default is **in**.If no type is given and a mode of **in** is used, constant is the default.

Example :

```
procedure build ( A : in integer;
                  B : inout signal bit_vector;
                  C : out real;
                  D : file ) ;
```

Procedure Body

Syntax :

```
procedure identifier [ ( formal parameter list ) ] is
        subprogram declaration
      | subprogram body
      | type declaration
      | subtype declaration
      | constant, object declaration
      | variable, object declaration
      | alias declaration
      | use clause
  begin
        sequential statement(s)
end procedure identifier ;
```

The procedure body formal parameter list is defined above in Procedure Declaration. When a procedure declaration is used then the corresponding procedure body should have exactly the same formal parameter list.

Example :

```
type op_code is ( add, sub, mul, div ) ;

procedure ALU ( A, B : in integer ;

                  p : in op_code ;

                  Z : out integer ; ) is
  Begin
```

```
case op is
   when add => Z <= A + B ;
   when sub => Z <= A - B ;
   when mul => Z <= A * B ;
   when others => Z <= A / B ;
end case ;
```

Procedure Call

Procedures are invoked by using procedure calls.

Syntax :

[Label] procedure-name (list-of-actuals) ;

The actuals specify the expressions, signals, variables, or files, that are to be passed into the procedure and the names of objects that are to receive the computed values from the procedure.

Example :

CheckTiming (tPLH, tPHL, Clk, D, Q);

Function Declaration

Syntax :

function identifier [(formal parameter list)] **return** a_type ;

Formal parameters are separated by semicolons in the formal parameter list. Each formal parameter is essentially a declaration of an object that is local to the function. The type definitions used in formal parameters must be visible at the place where the function is being declared. No semicolon follows the last formal parameter inside the parenthesis.

Formal parameters may be constants, signals or files. The default is constant. Formal parameters have the mode **in**. Files do not have a mode.

The reserved word **function** may be preceded by nothing, implying **pure**, **pure** or **impure** . A **pure function** must not contain a reference to a file object, slice, subelement, shared variable or signal with attributes such as 'delayed, 'stable, 'quiet, 'transaction and must not be a parent of an impure function.

Note -- The inout and **out** are not allowed for functions. The default is **in**.

Example :

function random **return** float ;
function is_even (A : integer) **return** boolean ;

Function Body

Syntax :

function identifier [(formal parameter list)] **return** a_type **is**
subprogram declaration
| subprogram body
| type declaration
| subtype declaration
| constant, object declaration
| variable, object declaration
| alias declaration
| use clause
begin
sequential statement(s)
return some_value; -- of type a_type
end function identifier ;

The function body formal parameter list is defined above in Function Declaration. When a function declaration is used then the corresponding function body should have exactly the same formal parameter list.

Example :

```
function random return float is
        variable X : float;
  begin
            return X;        -- compute X
end function random ;
```

Function Call

A function call is an expression and can also be used in large expressions.

Syntax :

f unction-name (list-of-actuals)

The actuals may be associated by position or using named association.

sum : = sum + largest (max_coins, collection) ;

PACKAGES

A package is used as a collection of often used data types, components, functions, and so on. Once these object are declared and defined in a package, they can be used by different VHDL design units. In particular, the definition of global information and important shared parameters in complex designs or within a project team is recommended to be done in packages. It is possible to split a package into a declaration part and the so-called body. The advantage of this splitting is that after changing definitions in the package body only this part has to be recompiled and the rest of the design can be left untouched. Therefore, a lot of time consumed by compiling can be saved.

Package declarations

A package declaration defines the interface to a package. The scope of a declaration within a package can be extended to other design units.

Syntax :

package *package*_name **is**

package_declarative_part :

subprogram_declaration

| type_declaration

| subtype_declaration

| constant_declaration

| signal_declaration

| alias_declaration

| component_declaration

| attribute_declaration & specification

| use_clause

end [**package**] [*package*_name] ;

If a name appears at the end of the package declaration, it must repeat the name of the package declaration. If a package declarative item is a type declaration, then that protected type definition must not be a protected type body. Items declared immediately within a package declaration become visible by selection within a given design unit wherever the name of that package is visible in the given unit. Such items may also be made directly visible by an appropriate use clause.

NOTE—Not all packages will have a package body. In particular, a package body is unnecessary if no subprograms, deferred constants, or protected type definitions are declared in the package declaration.

Example :

— A package declaration that needs no package body:

```
package TimeConstants is
        constant tPLH : Time := 10 ns;
        constant tPHL : Time := 12 ns;
        constant tPLZ : Time := 7 ns;
end TimeConstants ;
```

— A package declaration that needs a package body:

```
package MY PACK is
  type SPEED is (STOP, SLOW, MEDIUM, FAST);
     component HA
        port (I1, I2 : in bit; S, C : out bit);
     end component;
   constant DELAY TIME : time;
  function INT2BIT VEC (INT VALUE : integer) return bit vector;
end MY PACK;
```

Package bodies

A package body defines the bodies of subprograms and the values of deferred constants declared in the interface to the package.

Syntax :

```
package body package_name is
    package_body_declarative_part
      subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | alias_declaration
      | use_clause
end [ package body ] [ package_name ] ;
```

The name at the start of a package body must repeat the same name. In addition to subprogram body and constant declarative items, a package body may contain certain other declarative items to facilitate the definition of the bodies of subprograms declared in the interface.

Items declared in the body of a package cannot be made visible outside of the package body. If a given package declaration contains a deferred constant declaration, then a constant declaration with the same identifier must appear as a declarative item in the corresponding package body. This object declaration is called the full declaration of the deferred constant. The subtype indication given in the full declaration must conform to that given in the deferred constant declaration. Within a package declaration that contains the declaration of a deferred constant, and within the body of that package, the use of a name that denotes the deferred constant is only allowed in the default expression for a local generic, local port, or formal parameter. The result of evaluating an expression that references a deferred constant before the elaboration of the corresponding full declaration is not defined by the language.

Example :

```
package body MY PACK is
        constant DELAY TIME : time := 1.25 ns;
        function INT2BIT VEC (INT VALUE : integer)return bit vector is
 begin
                -- sequential behavioral description
        end INT2BIT VEC;
end MY PACK;
```

The binding between the package declaration and the body is established by using the same name. In the above example it is the package name MY PACK.

Packages must be made visible before their contents can be used. The USE clause makes packages visible to entities, architectures, and other packages.

Syntax :

Use library_name **.** Package_name **. all ;**

Example :

```
-- use only the binary and add_bits3 declarations
  USE work .my_stuff.binary, my_stuff.add_bits3;
... ENTITY declaration...
... ARCHITECTURE declaration ...
```

```
-- use all of the declarations in package my_stuff

   USE work .my_stuff.ALL;

   ... ENTITY declaration...

   ... ARCHITECTURE declaration...
```

CHAPTER 7

ADVANCED FEATURES

- Generate Statements
- Aliases
- Attributes

GENERATE STATEMENTS

Concurrent statements can be conditionally selected or replicated during the elaboration phase. The generate Statement provides for a compact description of regular structures such as memories, registers, and counters.

Two forms of the Generate Statement

- For-generation Scheme
- If-generation Scheme

For-Generation

Concurrent statements can be replicated a predetermined number of times.

Syntax :

generate-label **: for** generate-identifier **in** discrete-range **generate**

[block-declarations

begin]

concurrent statements

end generate [generate-label] **;**

The values in the discrete range must be globally static, that is, they must be computable at elaboration time. These statements can also use the generate identifier in their expressions, and its value would be substituted during elaboration for each replication. The type of identifier is defined by the discrete range. Declarations, if present, declare items that are visible only within the generate statement.

Example :

```
U1 : for  F  in  3  downto  0  generate

     sum ( F )   <=  A ( F )  xor B ( F ) xor C ( F )

     ca ( F + 1 )  <=  A ( F )  and B ( F ) and C ( F )

end  generate  U1  ;
```

If-Generation

Concurrent statements can be conditionally elaborated.

Syntax :

```
generate-label  : if  expression  generate

          [ block-declarations

  begin  ]

          concurrent statements

end  generate  [ generate-label ]  ;
```

This statement allows for conditional selection of concurrent statements based on the value of an expression. This expression must be a globally static expression, that is, the value must be computable at elaboration time. Any declarations present are again local to the generate statement.

Example :

```
V1 : if  User = low_Dly  generate
      Z  <=  A  after  2 ns  ;
end  generate  V1  ;

V2 : if  User = high_Dly  generate
      Z  <=  A  after  25 ns  ;
end  generate  V2  ;
```

ALIASES

An alias is an alternate name for an existing object. By using an alias of an object, you actually use the object to which it refers. By assigning to an alias, you actually assign to the object to which the alias refers.

Syntax :

alias identifier **:** subtype_indication **is** name **;**

A reference to an alias is interpreted as a reference to the object or part corresponding to the alias.

Example :

```
variable instr : bit_vector(31 downto 0);
      alias op_code : bit_vector(7 downto 0) is instr(31 downto 24);
```

declares the name op_code to be an alias for the left-most eight bits of instr.

```
signal vec : std_logic_vector (4 downto 0) ;

alias mid_bit : std_logic is vec(2) ;


    -- Assignment :

        mid_bit <= '0' ;

   -- is the same as

        vec(2) <= '0' ;
```

Aliases are often useful in unbound function calls. For instance, if you want to make a function that takes the AND operation of the two left most bits of an arbitrary array parameter. If you want to make the function general enough to handle arbitrary sized arrays, this function could look like this:

```
function left_and (arr: std_logic_vector) return std_logic is

  begin

        return arr(arr'left) and arr(arr'left-1) ;

end left_and ;
```

---- Function does not work for ascending index ranges of arr.

This function will only work correctly if the index range of arr is descending (downto). Otherwise, arr'left-1 is not a valid index number. VHDL does not have a simple attribute that will give the one-but-leftmost bit out of an arbitrary vector, so it will be difficult to make a function that works correctly both for ascending and descending index ranges. Instead, you could make an alias of arr, with a known index range, and operate on the alias:

```
function left_and (arr : std_logic_vector) return std_logic is
alias aliased_arr : std_logic_vector (0 to arr'length-1) is arr ;
  begin
      return aliased_arr(0) and aliased_arr(1) ;
end left_and ;
```

---- Function works for both ascending and descending index ranges of arr.

ATTRIBUTES

It is a value, function, type, range, signal, or constant that can associated with certain names. Such as an entity name, an architecture name, a label, or a signal

User-defined Attributes

User-defined attributes are constants of any type, except access or file type. They are declared using attribute declarations User-defined attributes are useful for annotating language models with tool-specific information.

Attribute Declarations

It declares the name of the attribute and its type.

Syntax & Example :

attribute attribute-name **:** value-type **;**

```
type Farads is range 0 to 5000 ;
  units
     pf ;
  end units ;
attribute capacitance : Farads ;
```

These user-defined attribute with a name and to assign a value to the attribute.

Attribute Specification

It is used to associate a user-defined attribute with a name and to assign a value to the attribute.

Syntax :

attribute attribute-name **of** item-names **:** name-class **is** expression **;**

The item-names is a list of one or more names of an entity, architecture, configuration, component, label, signal, variable, constant, type, subtype, package, procedure, or function. The name-class indicates the class type, that is, whether it is an entity, architecture, label, or others. The expression, whose value must belong to the type of attribute, specifies the value of the attribute.

Example : attribute length of RX_Rdy : signal is 3 micron ;

attribute capacitance of clk, rst : signal is 20 pf ;

The item-name in the attribute specification can also be replaced with the keyword all to indicate all names belonging to that name-class.

Example :

```
attribute capacitance of  all  :  variable is 10 pf ;
```

After having created an attribute and then associated it with a name, the value of the attribute can then be used in an expression. An attribute indicates a specific property of the signal, and is of a defined type. Using attributes at the right places creates a very flexible style of writing VHDL code.

Syntax :

```
item-name ‘ attribute-name
```

Example :

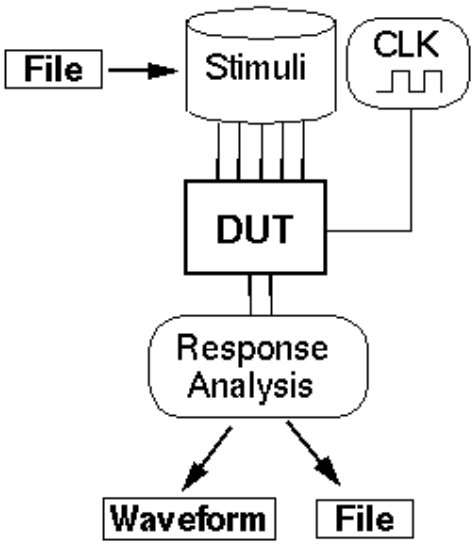
```
signal vector_up : bit_vector (4 to 9) ;

signal vector_dwn : bit_vector (25 downto 0) ;

vector_dwn’LEFT        -- returns integer 25

vector_dwn’RANGE     -- returns range 25 to 0

X'EVENT                  -- TRUE when there is an event on signal X

Y'HIGH                    -- returns the highest value in the range of Y

vector_up’RIGHT       -- returns integer 9

vector_up’RANGE      -- returns range 4 to 9
```

TEST BENCHES

Testbenches have become the standard method to verify High-Level Language designs. Typically, testbenches perform the following tasks:

- Instantiate the design under test (DUT)
- Stimulate the DUT by applying test vectors to the model
- Output results to a terminal or waveform window for visual inspection
- Optionally compare actual results to expected results

Testbenches are written in the industry-standard VHDL or Verilog hardware description languages. Testbenches invoke the functional design, then stimulate it. Complex testbenches perform additional functions—for example, they contain logic to determine the proper design stimulus for the design or to compare actual to expected results. The remaining sections of this note describe the structure of a well-composed testbench, and provide an example of a self-checking testbench—one that automates the comparison of actual to expected testbench results.



- Stimuli transmitter to testvectors
- Needs not to be synthesizable
- No ports to the outside
- Environment for DUT
- Verification and validation of the design
- Several output methods and input methods

Syntax :

```
Entity testbench_name is
end testbench_name ;

architecture testbench_archname of testbench_name is
    signal declarations
    component declarations
  begin
    UUT : component instantiation;
   stimuli;

end testbench_archname ;
```

Example :

```
library ieee;  use ieee.std_logic_1164.all;
entity testnand is
end testnand;

architecture testgate of testnand is
        component my_nand is
            port ( A, B : in std_logic; Y : out std_logic );
        end component;
        signal A, B, Y : std_logic;
```

```
    begin
        UUT : my_nand port map ( A, B, Y );
process
 begin
    A <= '0';wait for 20 ns;
    B <= '0'; wait for 20 ns;
end process;
process
  begin
    A <= '0';wait for 20 ns;
    B <= '1'; wait for 20 ns;
 end process;
 process
   begin
   A <= '0';wait for 20 ns;
   B <= '0'; wait for 20 ns;
 end process;
 process
   begin
    A <= '0';wait for 20 ns;
    B <= '1'; wait for 20 ns;
 end process;
End testgate;
```

A testbench that instantiates and provides stimulus to the shift register.

```
library IEEE;

use IEEE.std_logic_1164.all;

entity testbench is

end entity testbench;

architecture test_reg of testbench is

component shift_reg is

port (clock : in std_logic;

reset : in std_logic;

load : in std_logic;

sel : in std_logic_vector(1 downto 0);

data : in std_logic_vector(4 downto 0);

shiftreg : out std_logic_vector(4 downto 0));

end component;

signal clock, reset, load: std_logic;

signal shiftreg, data: std_logic_vector(4 downto 0);

signal sel: std_logic_vector(1 downto 0);

constant ClockPeriod : TIME := 50 ns;
```

```
begin

UUT : shift_reg port map (clock => clock, reset => reset,

load => load, data => data,

shiftreg => shiftreg);

process begin

clock <= not clock after (ClockPeriod / 2);

end process;

process begin

reset <= ’1’;

data <= "00000";

load <= ’0’;

set <= "00";

wait for 200 ns;

reset <= ’0’;

load <= ’1’;

wait for 200 ns;

data <= "00001";

wait for 100 ns;

sel <= "01";
```

```
load <= ’0’;

wait for 200 ns;

sel <= "10";

wait for 1000 ns;

end process;

end architecture test_reg;
```

PROGRAMS

```
--Design Unit : 4X1 Mux

--File Name : MUX.vhd

--Program for AND gate


        library ieee;

        use ieee.std_logic_1164.all;

        entity and3 is

        port(a,b,c : in std_logic;

            d: out std_logic);

        end and3 ;

        architecture data of and3 is

        begin

        d <=a and b and c;

        end data;
```

--Program for OR gate

```
library ieee;
use ieee.std_logic_1164.all;
entity or4 is
port(a,b,c,d : in std_logic;
    e: out std_logic);
end or4 ;
architecture data of or4 is
begin
e <=a or b or c or d;
end data;
```

--Program for NOT gate

```
library ieee;
use ieee.std_logic_1164.all;
entity inv is
port(a : in std_logic;
    b: out std_logic);
end inv ;
architecture data of inv is
begin
b <=not a;
end data;
```

--Program for 4x1 MUX

```
library ieee;
use ieee.std_logic_1164.all;
entity mux4x1 is
port(a,b,c,d,sel_1,sel_2 : in std_logic;
    muxout : out std_logic);
end mux4x1 ;
architecture str of mux4x1 is
component and3
port(a,b,c : in std_logic;
    d: out std_logic);
end component;
component or4
port(a,b,c,d : in std_logic;
    e: out std_logic);
end component;

component inv
port(a : in std_logic;
    b: out std_logic);
end component;
signal a1,a2,a3,a4,inv1,inv2 : std_logic;
begin
```

```
        n1 : inv port map (sel_1,inv1);
        n2 : inv port map (sel_2,inv2);
        u1 : and3 port map (a,inv1,inv2,a1);
        u2 : and3 port map (b,inv1,sel_2,a2);
        u3 : and3 port map (c,sel_1,inv2,a3);
        u4 : and3 port map (d,sel_1,sel_2,a4);
        u5 : or4 port map (a1,a2,a3,a4,muxout);
        end str ;

--Design Unit: 3 bit counter
--File Name :counter.vhd
--Program for AND gate

        library ieee;
        use ieee.std_logic_1164.all;
        entity and2 is
        port(a,b : in std_logic;
            c: out std_logic);
        end and2 ;
        architecture data of and2 is
        begin
        c <=a and b;
        end data;
```

```
--Program for T flipflop

        library ieee;
        use ieee.std_logic_1164.all;
        entity tff is
        port(reset,clock,t : in std_logic;
            q,q1 : inout std_logic);
        end tff ;
        architecture beh of tff is
        begin
          q1 <= not q;
        process(reset,clock,t)
        begin
        if (reset='1')then
          q <= '0';
        elsif (clock'event and clock='1') then
           if (t='0') then
             q <= q;
             else
              q <= not q ;
        end if; end if;
      end process;
      end beh;
```

```
--Program for counter
        library ieee;
        use ieee.std_logic_1164.all;
        entity count3bit is
        port(rst,clk : in std_logic;
            count : inout std_logic_vector(2 downto 0));
        end count3bit ;
        architecture str of count3bit is
        component and2
        port(a,b : in std_logic;
            c: out std_logic);
        end component;
        component tff
        port(reset,clock,t : in std_logic;
            q,q1 : inout std_logic);
        end component;
        signal a1 : std_logic;
        signal high : std_logic := '1';
        begin
        u1 : and2 port map (count(1),count(0),a1);
        u2 : tff port map (rst,clk,a1,count(2));
        u3 : tff port map (rst,clk,count(0),count(1));
        u4 : tff port map (rst,clk,high,count(0));
        end str ;
```

```
--Design Unit : Mealy machine

--File Name : mealy.vhd


        library ieee;

        use ieee.std_logic_1164.all;

        Entity mealy is

        port(clk,in1,reset:in std_logic;

        out1 : out std_logic_vector(1 downto 0));

        end mealy;

        Architecture mealy of mealy is

        type state_type is (s0,s1,s2,s3);

        signal state:state_type;

        begin

        p1:    process(clk.reset)

        begin

        if reset='1' then state<=s0 ;

        elsif clk'event and clk='1' then

        case state is

        when s0=>if in1='1' then state<=s1;

        end if;

        when s1=>if in1='0' then state<=s2;

        end if;

        when s2=>if in1='1' then state<=s3;

        end if;
```

```
when s3=>if in1='0' then state<=s0;

end if;  end case;  end if;

end process;

p2:    process(state,in1)

begin

case state is

when s0=>if in1='1' then out1<="01";

       else out1<="00";

end if;

when s1=>if in1='0' then out1<="10";

       else out1<="01";

end if;

when s2=>if in1='1' then out1<="11";

       else out1<="10";

end if;

when s3=>if in1='0' then out1<="00";

       else out1<="11";

end if: end case;

end process; end mealy;
```

```
--Design Unit : Comparator

--File Name : compar.vhd

        library IEEE;

        use IEEE.std_logic_1164.all;

        entity comparator is

        port(x,y:in std_logic_vector(3 downto 0);

          eq,gr,le:out std_logic);

        end entity comparator;

        architecture iterative of comparator is

        begin

          process(x,y)

           variable eqi:std_logic;

        begin

         if ( x<y)then

          le<='1';

        elsif ( x>y)then

         gr<='1';

         if ( x=y)then

        eq<='1'; end process;

        end architecture iterative;
```

```
--Design Unit : 16x7 ROM

--File Name : Rom.vhd


        library ieee;

        use ieee.std_logic_1164.all;

        entity rom16X7 is

          port (address : in INTEGER range 0 to 15;

                data : out std_logic_vector (6 downto 0));

        end entity rom16X7;

        architecture sevenseg of rom16X7 is

           type rom_array is array (0 to 15) of std_logic_vector(6 downto 0);

           constant rom : rom_array := ("1110111",

                                        "0010010",

                                        "1011101",

                                        "1011011",

                                        "0111010",

                                        "1101011",

                                        "1101111",

                                        "1010010",

                                        "1111111",

                                        "1111011",

                                        "1101101",

                                        "1101101",
```

```
                                "1101101",
                                "1101101",
                                "1101101",
                                "1101101");
            begin
              data <= rom(address);
           end architecture sevenseg;
--Design Unit : Shift Register
--File Name : shiftreg.vhd
--Program for D flipflop

        library ieee; use ieee.std_logic_1164.all;
        entity dff is
        port ( d,clk : in std_logic ;
             q : out std_logic );
        end dff ;
        architecture beh of dff is
        begin
        process(clk,d)
        begin
        wait until clk'event and clk='1' ;
        q <= d;
        end process;
        end beh;
```

--Program for Shift Register using generate statement

```
library ieee;
use ieee.std_logic_1164.all;
entity shift is
 port(din,clk: in std_logic;
    qout: out std_logic);
end shift;
Architecture gen_shift of shift is
component dff
  port(d,clk:in std_logic;
   q:out std_logic);
   end dff;
   signal qsh:std_logic_vector(0 to 7);
   begin
   qsh(0)<=din;
   g1:for i in 0 to 6 generate
   dffx:dff port map(qsh(i),clk,qsh(i+1));
 end generate;
   qout<=qsh(4);
end;
```

```
--Design Unit : JK flipflop

--File Name : JKff.vhd

        library ieee;

        use ieee.std_logic_1164.all;

        entity JK_FF is

          port (J,K,Clock,Reset: in std_logic;

                q,qbar : out std_logic);

        end entity JK_FF;

        architecture sig of JK_ff is

         signal state:std_logic;

        begin

        p0: process(Clock,Reset) is

           begin

              if (Reset = '0') then

                state <='0';

              elsif rising_edge(Clock) then

                case std_logic_vector'(J,K)is

                  when "11" =>

                    state <=not state;

                  when "10" =>

                    state <= '1';

                  when "01" =>

                    state <= '0';
```

```
            when others =>
            null;
           end case;
         end if;
       end process p0;
    q <= state;
    qbar <= not state;
    end architecture sig;

--Design Unit : ALU
--File Name : Alu.vhd

    library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic _arith.all;
    use ieee.std_logic _unsigned.all;
    entity alu is
      port(reset:in std_logic;
         b, sel  :in std_logic_vector(3 downto 0);
         acc, prod  :inout std_logic_vector(7 downto 0);
         flag :inout std_logic_vector(1 downto 0);
    end alu;
    architecture Behavioral of alu is
    signal count:std_logic_vector(3 downto 0);
```

```
begin
    Process(count,prod)
    begin
        case sel is
            when "0000"=>--addition
                acc<=acc+b;
            when "0001"=>--subtration
                acc<=acc-b;
            when "0010"=>--multiplication
                if(count< b)then
                    acc<=acc+prod; count<=count+'1';
                end if;
            when "0011"=>--divion
                if(prod>=b)then
                    prod<=prod-b;
                    acc<="0000"&count+'1';
                    count<=count+'1';
                end if;
            when "0100"=>--increment
                acc<=acc+'1';
            when "0101"=>--decrement
                acc<=acc-'1';
            when "0110"=> --compare
```

```
    if(acc<b)then
        flag(1) <='1';
    elsif(acc >=b) then
        flag(1) <='0';
    end if;
    if( acc=b) then
        flag(0) <='0';
    else
        flag(0) <='1';
    end if;
when "1000" =>--and
    acc <= acc and prod;
when "1001"=>--or
    acc<= acc or prod;
when "1010"=>--nand
    acc<= acc nand prod;
when "1011"=>--nor
    acc<= acc nor prod;
when "1100"=>--xor
    acc<= acc xor prod;
when "1101"=>--xnor
    acc<= acc xnor prod;
when "1110"=>--not
```

```
            acc<=not acc;
        when others => acc <= acc;
    end case;
    if(reset'event and reset='1') then
        acc <= "00000000";
    end if;
    if(sel'event and sel = "0011") then
        prod <=acc;
        acc <= "00000000";
        count <="0000";
    end if;
    if(sel'event and sel = "0010") then
        acc <= "00000000";
        prod <= acc;
        count <= "0000";
    end if;
    if(sel (3) ='1') then
         prod <= "0000" & b;
    end if;    end process;
end Behavioral;
```

```
--Design Unit : Pseudo Random Bit Sequence Generator

--File Name : prbs_gen.vhd


          entity prbsgen is

            generic(length : Positive := 8; tap1 : Positive := 8; tap2 : Positive := 4);

            port(clk, reset : in Bit; prbs : out Bit);

          end prbsgen;


           architecture v2 of prbsgen is

            signal prreg : Bit_Vector(length downto 0);

          begin

            prreg <= (0 => '1', others=> '0') when reset = '1' else

                  (prreg((length - 1) downto 0) & (prreg(tap1) xor prreg(tap2)))

                  when clk'event and clk = '1' else

                  prreg;

            prbs <= prreg(length);

        end v2;


--Design Unit : 7-Segment Decoder

--File Name : segdec.vhd


          library ieee;

          use ieee.std_logic_1164.all;
```

```
entity adcout is port(  nib0:  in std_logic_vector(3 downto 0);
 nib1:  in std_logic_vector(3 downto 0);
clk:   in bit;
dis0:     out bit;
dis1:     out bit;
ssdout:  out std_logic_vector(7 downto 0) );
end adcout;

architecture arch_adc of adcout is
signal tmp : std_logic_vector(3 downto 0);
begin
--               a
--             --
--          f|         | b
--             --              --  seven segment display format ( .gfedcba )
--       e|    g    | c
--              --
--           d

process(clk,tmp,nib0,nib1)
            begin
            if clk = '0' then
              tmp <= nib0;
              dis0 <= '0';
```

```
            dis1 <= '1';
          elsif clk = '1' then
            tmp <= nib1;
          dis0 <= '1';
          dis1 <= '0';
end if;
  if    tmp = "0000" then
          ssdout <= "11000000";
  elsif tmp = "0001" then
          ssdout <= "11111001";
  elsif tmp = "0010" then
          ssdout <= "10100100";
  elsif tmp = "0011" then
          ssdout <= "10110000";
  elsif tmp = "0100" then
       ssdout <= "10011001";
  elsif tmp = "0101" then
          ssdout <= "10010010";
  elsif tmp = "0110" then
          ssdout <= "10000010";
  elsif tmp = "0111" then
          ssdout <= "11111000";
  elsif tmp = "1000" then
```

```
            ssdout <= "10000000";
        elsif tmp = "1001" then
            ssdout <= "10010000";
        elsif tmp = "1010" then
            ssdout <= "10001000";
        elsif tmp = "1011" then
            ssdout <= "10000011";
        elsif tmp = "1100" then
            ssdout <= "11000110";
        elsif tmp = "1101" then
            ssdout <= "10100001";
        elsif tmp = "1110" then
            ssdout <= "10000110";
        elsif tmp = "1111" then
            ssdout <= "10001110";
        end if;
    end process;
end arch_adc;

--Design Unit : 3-bit 1-of-9 Priority Encoder
--File Name : prienc.vhd

.library ieee;
use ieee.std_logic_1164.all;
```

```
entity priority is
port ( sel : in std_logic_vector (7 downto 0);
       code :out std_logic_vector (2 downto 0));
end priority;
architecture archi of priority is
begin
  code <= "000" when sel(0) = '1' else
          "001" when sel(1) = '1' else
          "010" when sel(2) = '1' else
          "011" when sel(3) = '1' else
          "100" when sel(4) = '1' else
          "101" when sel(5) = '1' else
          "110" when sel(6) = '1' else
          "111" when sel(7) = '1' else
          "---";
end archi;
```

--Design Unit : Fibonacci series

--File Name : Fibo.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Fibonacci is
port
(
   Reset      : in    std_logic;
   Clock      : in    std_logic;
```

```
    Number      : out   unsigned(31 downto 0)
);
end entity Fibonacci;
architecture fibo_arch of Fibonacci is
    signal  Previous    : natural;
    signal  Current     : natural;
    signal  Next_Fib    : natural;
begin
    Adder:
    Next_Fib <= Current + Previous;
    Registers:  process (Clock, Reset) is
    begin
        if Reset = '1' then
            Previous <= 1;
            Current  <= 1;
        elsif rising_edge(Clock) then
            Previous <= Current;
            Current  <= Next_Fib;
        end if;
    end process Registers;
    Number <= to_unsigned(Previous, 32);
end architecture fibo_arch;
```

Verilog

CONTENTS

INTRODUCTION

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). A hardware description Language is a language used to describe a digital system, for example, a microprocessor or a memory or a simple flip-flop. This just means that, by using a HDL one can describe any hardware (digital) at any level.

Verilog is one of the HDL languages available in the industry for designing the Hardware. Verilog allows us to design a Digital design at Behavior Level, Register Transfer Level (RTL), Gate level and at switch level. Verilog allows hardware designers to express their designs with behavioral constructs, deterring the details of implementation to a later stage of design in the final design.

Abstraction Levels of Verilog

Verilog supports a design at many different levels of abstraction. Three of them are very important:

Behavioral level
Register-Transfer Level
Gate Level

Behavioral level

This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

Register-Transfer Level

Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing possibility, operations are scheduled to occur at certain times. Modern definition of a RTL code is **"Any code that is synthesizable is called RTL code".**

Gate Level

Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this net list is used for gate level simulation and for backend.

History of Verilog

Verilog was started initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. It is rumored that the original language was designed by taking features from the most popular HDL language of the time, called Hilo as well as from traditional computer language such as C. At that time, Verilog was not standardized and the language modified itself in almost all the revisions that came out within 1984 to 1990.

Verilog simulator was first used beginning in 1985 and was extended substantially through 1987.The implementation was the Verilog simulator sold by Gateway. The first major extension was Verilog-XL, which added a few features and implemented the infamous "XL algorithm" which was a very efficient method for doing gate-level simulation.

The time was late 1990. Cadence Design System, whose primary product at that time included thin film process simulator, decided to acquire Gateway Automation System. Along with other Gateway product, Cadence now became the owner of the Verilog language, and continued to market Verilog as both a language and a simulator. At the same time, Synopsys was marketing the top- down design methodology, using Verilog. This was a powerful combination.

In 1990, Cadence recognized that if Verilog remained a closed language, the pressures of standardization would eventually cause the industry to shift to VHDL. Consequently, Cadence organized Open Verilog International (OVI), and in 1991 gave it the documentation for the Verilog Hardware Description Language. This was the event which "opened" the language.

OVI did a considerable amount of work to improve the Language Reference Manual (LRM), clarifying things and making the language specification as vendor-independent as possible. In 1990 soon it was realized, that if there were too many companies in the market for Verilog, potentially everybody would like to do what Gateway did so far - changing the language for their own benefit. This would defeat the main purpose of releasing the language to public domain. As a result in 1994, the IEEE 1364 working group was formed to turn the OVI LRM into an IEEE standard. This effort was concluded with a successful ballot in 1995, and Verilog became an IEEE standard in December, 1995.

When Cadence gave OVI the LRM, several companies began working on Verilog simulators. In 1992, the first of these were announced, and by 1993 there were several Verilog simulators available from companies other than Cadence. The most successful of these was VCS, the Verilog Compiled Simulator, from Chronologic Simulation. This was a true compiler as opposed to an interpreter, which is what Verilog-XL was. As a result, compile time was substantial, but simulation execution speed was much faster.

In the meantime, the popularity of Verilog and PLI was rising exponentially. Verilog as a HDL found more admirers than well-formed and federally funded VHDL. It was only a matter of time before people in OVI realized the need of a more universally accepted standard. Accordingly, the board of directors of OVI requested IEEE to form a working committee for establishing Verilog as an IEEE standard. The working committee 1364 was formed in mid 1993 and on October 14, 1993, it had its first meeting. The standard, which combined both the Verilog language syntax and the PLI in a single volume, was passed in May 1995 and now known as IEEE Std. 1364-1995.

After many years, new features have been added to Verilog, and new version is called Verilog 2001. This version seems to have fixed lot of problems that Verilog 1995 had. This version is called 1364-2000. Only waiting now is that all the tool vendors implementing it.

Verilog HDL Syntax and Semantics

Lexical Conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog HDL is a case-sensitive language. All keywords are in lowercase.

White Space

White space can contain the characters for blanks, tabs, newlines, and form feeds. These characters are ignored except when they serve to separate other tokens. However, blanks and tabs are significant in strings.

White space characters are:

Blank spaces

Tabs

Carriage returns

New-line

Form-feeds

Comments

There are two forms to introduce comments.

Single line comments begin with the token **//** and end with a **carriage return**

Multi Line comments begin with the token **/*** and end with the token***/**

Examples of Comments

```
/* 1-bit adder example for showing
few verilog */
module addbit (
a,
b,
ci,
sum,
co);
// Input Ports
input       a;
input       b;
input       ci;
// Output ports
output      sum;
output      co;
// Data Types

wire        a;
wire        b;
wire        ci;
wire        sum;
wire        co;
```

Case Sensitivity

Verilog HDL is case sensitive

Lower case letters are unique from upper case letters

All Verilog keywords are lower case

Examples of unique names

```
input               // a Verilog
Keyword wire        // a Verilog
Keyword
WIRE                // a unique name ( not a keyword)
Wire                // a unique name (not a keyword)
```

Identifiers

Identifiers are names used to give an object, such as a register or a module, a name so that it can be referenced from other places in a description.

Identifiers must begin with an alphabetic character or the underscore character (**a-z A-Z _**).Identifiers may contain alphabetic characters, numeric characters, the underscore, and the dollar sign (**a-z A-Z 0-9 _ $**) Identifiers can be up to 1024 characters long.

Escaped Identifiers

Verilog HDL allows any character to be used in an identifier by **escaping** the identifier. Escaped identifiers provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal). Escaped identifiers begin with the back slash (\ ) Entire identifier is escaped by the back slash Escaped identifier is terminated by white space o Characters such as commas, parentheses, and semicolons become part of the escaped identifier unless preceded by a white space. Terminate escaped identifiers with white space, otherwise characters that should follow the identifier are considered as part of it.

Numbers in Verilog

You can specify constant numbers in decimal, hexadecimal, octal, or binary format. Negative numbers are represented in 2's complement form. When used in a number, the question mark (?) character is the Verilog alternative for the z character. The underscore character (_) is legal anywhere in a number except as the first character, where it is ignored.

Integer Numbers

Verilog HDL allows integer numbers to be specified as Sized or unsized numbers (Unsized size is 32 bits) In a radix pf binary, octal, decimal, or hexadecimal Radix is case and hex digits (a,b,c,d,e,f) are insensitive Spaces are allowed between the size, radix and value.

Syntax: <size>'<radix><value>

Verilog expands **<value>** to be fill the specified **<size> by** working from right-to-left

When **<size>** is smaller than **<value>**, then left-most bits of **<value>** are truncated

When **<size>** is larger than **<value>**, then left-most bits are filled, based on the value of the left-most bit in <**value>.**
Left most '0' or '1' are filled with '0', 'Z' are filled with 'Z' and 'X' with 'X'

Example of integer numbers

8’hCA 11001010
16’bZ filled with 16 Z's

Real Numbers

Verilog supports real constants and variables
Verilog converts real numbers to integers by rounding
Real Numbers can not contain 'Z' and 'X'
Real numbers may be specified in either decimal or scientific notation

<value>.<value>

<mantissa>E<exponent>

Real numbers are rounded off to the nearest integer.

Example of Real Numbers

1.2,0.6

Signed and Unsigned Numbers

Verilog supports both the type of numbers, but with certain restrictions. Like in C language we don't have int and unint types to say if a number is signed integer or unsigned integer.

Any number that does not have negative sign prefix is a positive number. Or indirect way would be "Unsigned".

Negative numbers can be specified by putting a minus sign before the size for a constant number, thus become signed numbers. Verilog internally represents negative numbers in 2's compliment format. An optional signed specifier can be added for signed arithmetic.

Examples

32'hDEAD_BEEF Unsigned or signed positive Number

-14'h1234 Signed negative number

Ports:

Ports allow communication between a module and its environment. All but the top-level modules in a hierarchy have ports. Ports can be associated by order or by name.

You declare ports to be **input, output** or **inout**. The port declaration syntax is :
input [range_val:range_var] list_of_identifiers;
output [range_val:range_var] list_of_identifiers;
inout [range_val:range_var] list_of_identifiers;

Examples: Port Declaration

```
input              clk          ; // clock input
input   [15:0]     data_in      ; // 16 bit data input bus
output  [7:0]      count        ; // 8 bit counter output
inout              data_bi      ; // Bi-Directional data bus
```

Examples : **A complete Example in Verilog**

```
module addbit (
a      , // first input
b      , // Second input
ci     , // Carry Input
sum    , // Sum output
co       // Carry output
);
// Input Declaration
input   a  ;
input   b  ;
input   ci ;
// Ouput Declaration
output  sum;
output  co ;
// port data types
wire    a  ;
wire    b  ;
wire    ci ;
wire    sum;
wire    co ;

// Code starts Here
assign {co,sum} = a + b + ci;

endmodule // End Of Module addbit
```

Data Types

Verilog Language has two primary data types

Nets - represents structural connections between components.

Registers - represent variables used to store data.

Every signal has a data type associated with it:

Explicitly declared with a declaration in your Verilog code.

Implicitly declared with no declaration but used to connect structural building blocks in your code.

Implicit declaration is always a **net** of type wire and is one bit wide.

Types of Nets

Each net type has functionality that is used to model different types of hardware (such as **PMOS, NMOS, CMOS**, etc)

Net Data Type		Functionality
wire	**tri**	Interconnecting wire - no special resolution function
wor	**trior**	Wired outputs OR together (models ECL)
wand	**triand**	Wired outputs AND together (models open-collector)
tri0	**tri1**	Net pulls-down or pulls-up when not driven
supply0	**supply1**	Net has a constant logic 0 or logic 1 (supply strength)

Register Data Types

Registers store the last value assigned to them until another assignment statement changes their value. Registers represent data storage constructs. You can create arrays of the regs called memories. register data types are used as variables in procedural blocks. A register data type is required if a signal is assigned a value within a procedural block. Procedural blocks begin with keyword initial and always.

Data types	Functionality
reg	Unsigned variable
integer	Signed variable - 32 bits
time	Unsigned integer - 64 bits
real	Double precision floating point variable

Strings

A string is a sequence of characters enclosed by double quotes and all contained on a single line. Strings used as operands in expressions and assignments are treated as a sequence of eight-bit ASCII values, with one eight-bit ASCII value representing one character. To declare a variable to store a string, declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character. Strings can be manipulated using the standard operators. When a variable is larger than required to hold a value being assigned, Verilog pads the contents on the left with zeros after the assignment. This is consistent with the padding that occurs during assignment of non-string values.

Certain characters can be used in strings only when preceded by an introductory character called an escape character. The following table lists these characters in the right-hand column with the escape sequence that represents the character in the left-hand column.

Special characters in string

\n	New line character
\t	Tab character
\\	Backslash (\) character
\"	Double quote (") character
\ddd	A character specified in 1-3 octal digits (0 <= d <= 7)
%%	Percent (%) character

Example

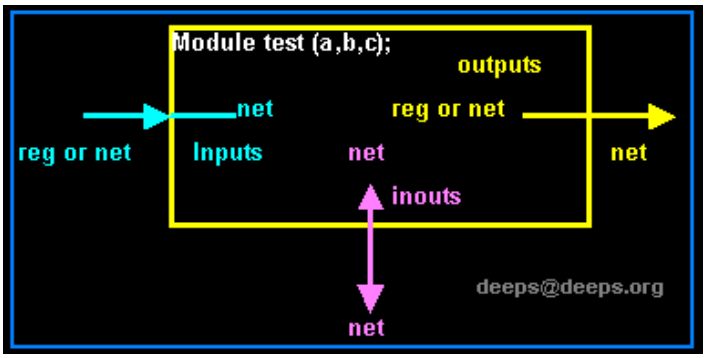
```
reg [8*17:0] version   ; // Declare a register variable that is 18 bytes
initial
  version = "model version 1.0";
```

Port Connection Rules

Inputs : internally must always be type net, externally the inputs can be connected to variable reg or net type.

Outputs : internally can be type net or reg, externally the outputs must be connected to a variable net type.

Inouts : internally or externally must always be type net, can only be connected to a variable net type.



Width matching: It is legal to connect internal and external ports of different sizes. But beware, synthesis tools could report problems.

Unconnected ports: unconnected ports are allowed by using a **","** The net data types are used to connect structure

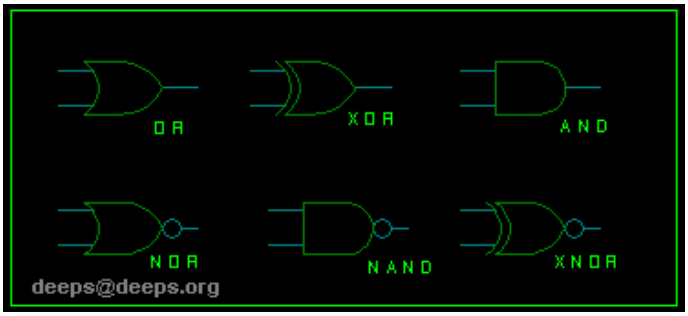
A net data type is required if a signal can be driven a structural connection.

Gate Level Modeling

Introduction

Verilog has built in primitives like gates, transmission gates, and switches. This are rarely used for in design work, but are used in post synthesis world for modeling the ASIC/FPGA cells, this cells are then used for gate level simulation or what is called as SDF simulation.

Gate Primitives



The gates have one scalar output and multiple scalar inputs. The 1st terminal in the list of gate terminals is an output and the other terminals are inputs.

And N-input and gate

Nand N-input nand gate

Or N-input or gate

Nor N-input nor gate

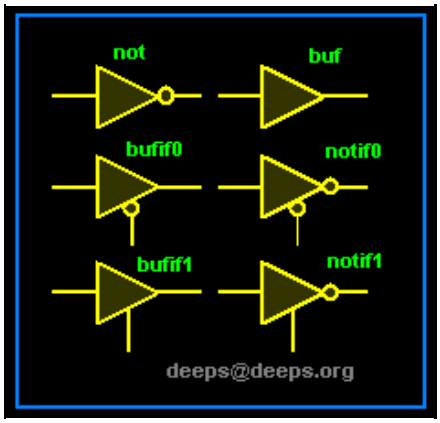
Xor N-input xor gate

Xnor N-input xnor gate

Examples

```
and U1(out,in);
and U2(out,in1,in2,in3,in4);
xor U3(out,in1,in2,in3);
```

Transmission Gate Primitives

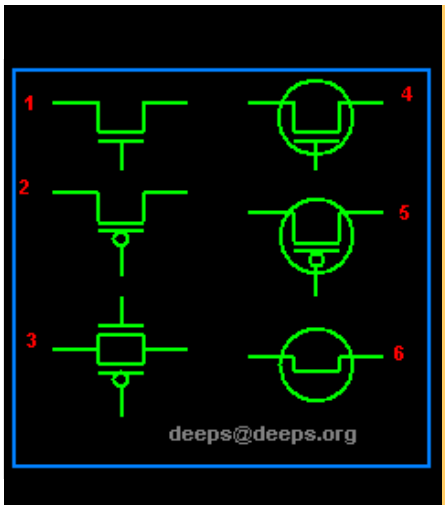


not	N-output invertor.
buf	N-output buffer.
bufif0	Tri-state buffer, Active low en.
bufif1	Tristate inverter, Low en.
notif0	Tri-state buffer, Active high en.
notif1	Tristate inverter, High en.

Examples

```
bufif0 U1(data_bus,data_drive,  data_enable_low);
buf  U2(out,in);
not U3(out,in);
```

Switch Primitives



1	pmos	Uni-directional PMOS switch
	rpmos	Resistive PMOS switch
2	nmos	Uni-directional NMOS switch
	rnmos	Resistive NMOS switch
3	cmos	Uni-directional CMOS switch
	rcmos	Resistive CMOS switch
4	tranif1	Bi-directional transistor (High)
	tranif1	Resistive transistor (High)
5	tranif0	Bi-directional transistor (Low)
	rtranif1	Resistive Transistor (Low)
6	tran	Bi-directional pass transistor
	rtran	Resistive pass transistor

Syntax: keyword unique_name (inout1, inout2, control);

tranif0 my_gate1 (net5, net8, cnt);
rtranif1 my_gate2 (net5, net12, cnt);

Transmission gates tran and rtran are permanently on and do not have a control line. Tran can be used to interface two wires with separate drives, and rtran can be used to weaken signals. Resistive devices reduce the signal strength which appears on the output by one level. All the switches only pass signals from source to drain, incorrect wiring of the devices will result in high impedance outputs.

Logic Values and signal Strengths

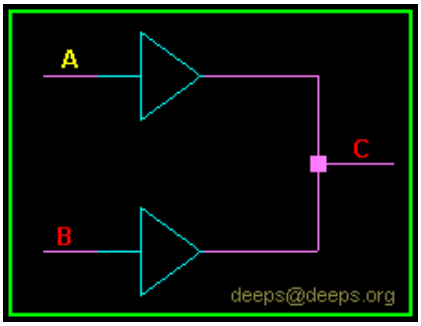
The Verilog HDL has got four logic values

0	zero, low, false
1	one, high, true
z or **Z**	high impendence, floating
x or **X**	unknown, uninitialized, contention

Verilog Strength Levels

Strength Level	Strength	Specification Keyword	
7	Supply Drive	supply0	supply1
6	Strong Pull	strong0	strong1
5	Pull Drive	pull0	pull1
4	Large Capacitance	large	
3	Weak Drive	weak0	weak1
2	Medium Capacitance	medium	
1	Small Capacitance	small	
0	Hi Impedance	highz0	highz1

Examples

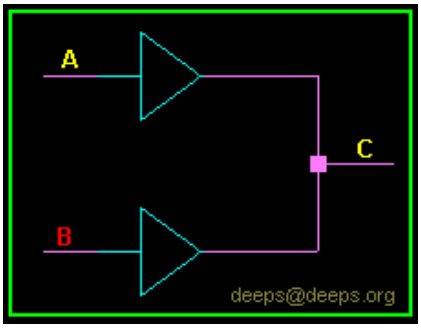


Two buffers that has output

A : Pull 1

B : Supply 0

Since supply 0 is stronger then pull 1, Output C takes value of B.



Two buffers that has output

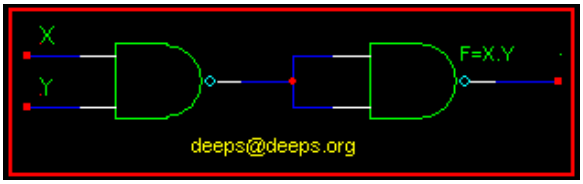
A : Supply 1

B : Large 1

Since Supply 1 is stronger then Large 1, Output C takes the value of A

Designing Using Primitives

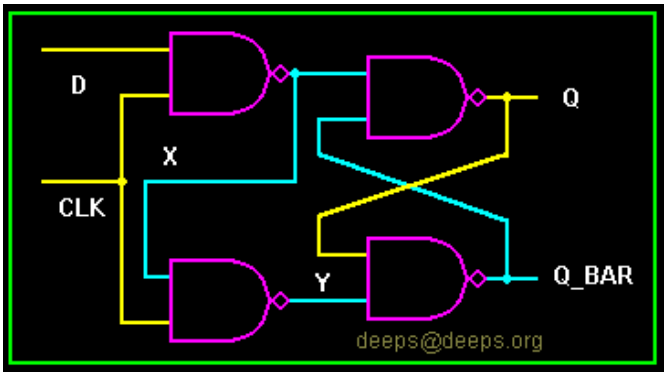
AND Gate from NAND Gate



Verilog code

```
// Structural model of AND gate from two NANDS
module and_from_nand(X, Y, F);

input X, Y;
output F;
wire W;
// Two instantiations of the module NAND
nand U1(X, Y, W);
nand U2(W, W, F);

endmodule
```

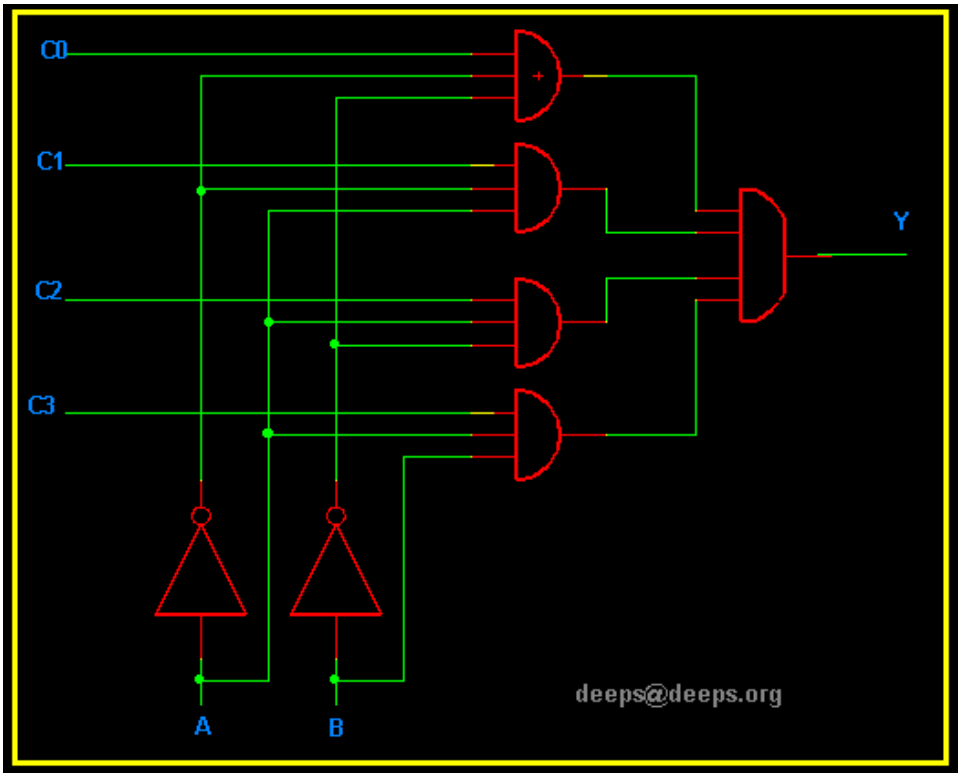
D-Flip flop from NAND Gate



Verilog Code

```
module dff(Q,Q_BAR,D,CLK);
output Q,Q_BAR;
input D,CLK;

nand U1 (X,D,CLK) ;
nand U2 (Y,X,CLK) ;
nand U3 (Q,Q_BAR,X);
nand U4 (Q_BAR,Q,Y);

endmodule
```

Multiplexer from primitives



Verilog Code

```
//Module 4-2 Mux
module mux (c0,c1,c2,c3,A,B,Y);
input c0,c1,c2,c3,A,B;
ouput Y;
//Invert the sel signals
not (a_inv, A);
not (b_inv, B);
// 3-input AND gate
and (y0,c0,a_inv,b_inv);
and (y1,c1,a_inv,B);
and (y2,c2,A,b_inv);
and (y3,c3,A,B);
// 4-input OR gate
or (Y, y0,y1,y2,y3);

endmodule
```

Gate and Switch delays

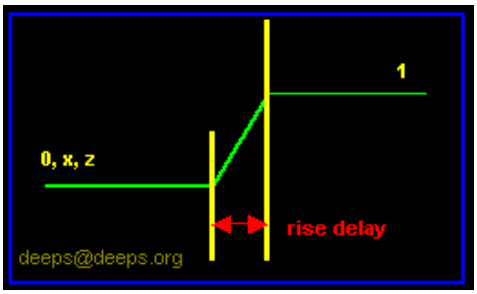
In real circuits , logic gates haves delays associated with them. Verilog provides the mechanism to associate delays with gates.

Rise, Fall and Turn-off delays.

Minimal, Typical, and Maximum delays.

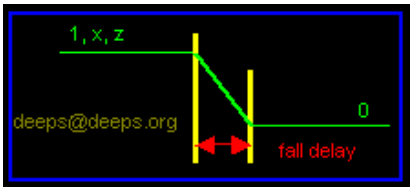
Rise Delay

The rise delay is associated with a gate output transition to 1 from another value (0,x,z).



Fall Delay :

The fall delay is associated with a gate output transition to 0 from another value (1,x,z).



Turn-off Delay

The fall delay is associated with a gate output transition to z from another value (0,1,x).

Min Value

The min value is the minimum delay value that the gate is expected to have.

Max Value

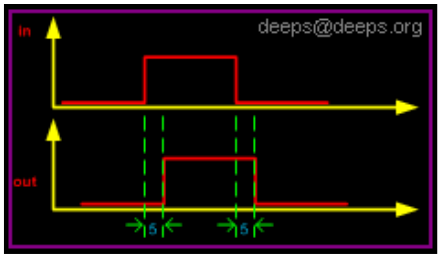
The max value is the maximum delay value that the gate is expected to have.

Examples

```
// Delay for all transitions
or #5  u_or (a,b,c);

// Rise and fall delay
and #(1,2) u_and (a,b,c);

// Rise, fall and turn off delay
nor # (1,2,3) u_nor (a,b,c);

//One Delay, min, typ and max
nand #(1:2:3) u_nand (a,b,c);

//Two delays, min,typ and max
buf #(1:4:8,4:5:6) u_buf (a,b);

//Three delays, min, typ, and max
notif1 #(1:2:3,4:5:6,7:8:9) u_notif1 (a,b,c);
```

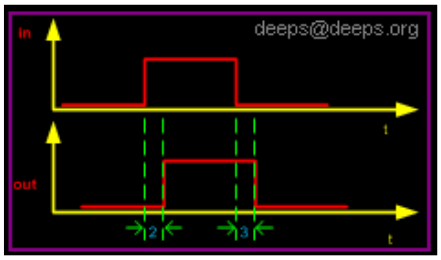
Gate Delay Code Example

```
module not_gate (in,out);
  input in;
  output out;

  not #(5) (out,in);

endmodule
```



Gate Delay Code Example

```
module not_gate (in,out);
  input in;
  output out;

  not #(2,3) (out,in);

endmodule
```



Normally we can have three models of delays, typical, minimum and maximum delay. During compilation of a modules one needs to specify the delay models to use, else Simulator will use the typical model.

N-Input Primitives

The **and, nand, or, nor, xor,** and **xnor** primitives have one output and any number of inputs

The single output is the first terminal

All other terminals are inputs

Examples

```
// Two input AND gate
and u_and (out, in1, in2);

// four input AND gate
and u_and (out, in1, in2, in3, in4);

// three input XNOR gate
xnor u_xnor (out, in_1, in_2, in_3);
```

N-Output Primitives

The **buf** and **not** primitives have any number of outputs and one input

The output are in first terminals listed.

The last terminal is the single input.

Examples

```
// one output Buffer gate
buf u_buf (out,in);

// four output Buffer gate
buf u_buf (out_0, out_1, out_2, out_3, in);

// three output Invertor gate
not u_not (out_a, out_b, out_c, in);
```

Verilog Operators

Arithmetic Operators

Binary: +, -, *, /, % (the modulus operator)

Unary: +, -

Integer division truncates any fractional part

The result of a modulus operation takes the sign of the first operand

If any operand bit value is the unknown value x, then the entire result value is x

Register data types are used as unsigned values

- negative numbers are stored in two's complement form

Relational Operators

a<b	a less than b a>b
	a greater than b
a<=b	a less than or equal to b a>=b
	a greater than or equal to b

The result is a scalar value:

0 if the relation is false

1 if the relation is true

x if any of the operands has unknown x bits

Note: If a value is x or z, then the result of that test is false

Equality Operators

a === b	a equal to b, including x and z
a !== b	a not equal to b, including x and z
a == b	a equal to b, resulting may be unknown
a != b	a not equal to b, result may be unknown

Logical Operators

!	logic negation
&&	logical and
||	logical or

Expressions connected by && and || are evaluated from left to right

Evaluation stops as soon as the result is known

The result is a scalar value:

- 0 if the relation is false
- 1 if the relation is true
- x if any of the operands has unknown x bits

Bit-wise Operators

~	negation
&	and
|	inclusive or
^	exclusive or
^~ or ~^	exclusive nor (equivalence)

Computations include unknown bits, in the following way:

- ~x = x
- 0&x = 0
- 1&x = x&x = x
- 1|x = 1
- 0|x = x|x = x
- 0^x = 1^x = x^x = x
- 0^~x = 1^~x = x^~x = x

When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions

Reduction Operators

&	**and**
~&	**nand**
|	**or**
~|	**nor**
^	**xor**
^~ or ~^	**xnor**

Reduction operators are unary.

Shift Operators

<<	**left shift**
>>	**right shift**

The left operand is shifted by the number of bit positions given by the right operand. The vacated bit positions are filled with zeroes.

Concatenation Operator

Concatenations are expressed using the brace characters { and }, with commas separating the expressions within

Examples

{a, b[3:0], c, 4'b1001} // if a and c are 8-bit numbers, the results has 24 bits

Unsized constant numbers are not allowed in concatenations

Repetition multipliers that must be constants can be used:

{3{a}} // this is equivalent to {a, a, a} Nested

concatenations are possible:

{b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}

Conditional Operator

The conditional operator has the following C-like format:

cond_expr ? true_expr : false_expr

The true_expr or the false_expr is evaluated and used as a result depending on whether cond_expr evaluates to true or false

Example

out = (enable) ? data : 8'bz; // Tri state buffer

Operator precedence

Operator	Symbols
Unary, Multiply, Divide, Modulus	**+ - ! ~ * / %**
Add, Subtract, Shift.	**+, - , <<, >>**
Relation, Equality	**<,>,<=,>=,==,!=,===,!===**
Reduction	**&, !&,^,^~,|,~|**
Logic	**&&, ||**
Conditional	**?:**

Behavioral Modeling

Verilog HDL Abstraction Levels

Behavioral Models: Higher level of modeling where behavior of logic is modeled.

RTL Models: Logic is modeled at register level

Structural Models: Logic is modeled at both register level and gate level.

Procedural Blocks

Verilog behavioral code is inside procedures blocks, but there is a exception, some behavioral code also exist outside procedures blocks. We can see this in detail as we make progress.

There are two types of procedural blocks in Verilog

initial : initial blocks execute only once at time zero (start execution at time zero).

always : always blocks loop to execute over and over again, in other words as name means, it executes always.

Example : initial and always

```
initial                 always @ (posedge clk)
  begin                   begin : D_FF
    clk = 0;              if (reset == 1)
    reset = 0;              q <= 0;
    enable = 0;           else
    data = 0;               q <=d;
  end                     end
```

Procedural Assignment Statements

Procedural assignment statements assign values to registers and can not assign values to nets (wire data types)

You can assign to the register (reg data type) the value of a net (wire), constant, another register, or a specific value.

Example : Bad and Good procedural assignment

```
wire clk, reset;
reg enable, data;

initial
  begin
    clk = 0;
    reset = 0;
    enable = 0;
    data = 0;
  end
```

```
reg clk, reset;
reg enable, data;

initial
    begin
      clk = 0;
      reset = 0;
      enable = 0;
      data = 0;
    end
```

Procedural Assignment Groups

If a procedure block contains more then one statement, those statements must be enclosed within

Sequential **begin** - **end** block

Parallel **fork** - **join** block

When using begin-end, we can give name to that group. This is called named blocks.

Example : "begin-end" and "fork - join"

```
initial                    initial
  begin                      fork
    #1 clk = 0;                #1 clk = 0;
    #5 reset = 0;              #5 reset = 0;
    #5 enable = 0;             #5 enable = 0;
    #2 data = 0;               #2 data = 0;
  end                        join
```

Begin : clk gets 0 after 1 time unit, reset gets 0 after 6 time units, enable after 11 time units, data after 13 units. All the statements are executed in sequentially.

Fork : clk gets value after 1 time unit, reset after 5 time units, enable after 5 time units, data after 2 time units. All the statements are executed in parallel.

The Conditional Statement if-else

The **if - else** statement controls the execution of other statements, In programming language like c, if - else controls the flow of program.

```
if (condition)
  statements;
```

```
if (condition)
  statements;
else
  statements;
```

```
if (condition)
   statements;
else if (condition)
   statements;
.....
.....
else
   statements;
```

Example

```
// Simple if statement
if (enable)
   q <= d;
// One else statement
if (reset == 1'b1)
   q <= 0;;
else
   q <= d;
// Nested if-else-if statements
if (reset == 1'b0)
   counter <= 4'b0000;
else if (enable == 1'b1 && up_en == 1'b1)
   counter <= counter + 1'b1;
else if (enable == 1'b1 && down_en == 1'b1);
   counter <= counter - 1'b0;
else
   counter <= counter; // Redundant code
```

The Case Statement

The **case** statement compares a expression to a series of cases and executes the statement or statement group associated with the first matching case

Case statement supports single or multiple statements. Group multiple statements using begin and end keywords.

```
case (<expression>)

   <case1> : <statement>

   <case2> : <statement>

   .....

   default : <statement>

endcase
```

Example

```
module mux (a,b,c,d,sel,y);
    input a, b, c, d;
    input [1:0] sel;
    output y;
    reg y;
    always @ (a or b or c or d or sel)
    case (sel)
        0 : y = a;
        1 : y = b;
        2 : y = c;
        3 : y = d;
        default : $display("Error in SEL");
    endcase

endmodule
```

The Verilog case statement does an identity comparison (like the === operator), One can use the case statement to check for logic x and z values

Example with z and x

```
case(enable)
   1'bz : $display ("enable is floating");
   1'bx : $display ("enable is unknown");
   default : $display ("enable is %b",enable);
endcase
```

The casez and casex statement

Special versions of the case statement allow the x ad z logic values to be used as "don't care"

casez uses the **z** as the don't care instead of as a logic value **casex** uses either the **x** or the **z** as don't care instead of as logic values

Example casez

```
casez(opcode)
   4'b1zzz : out = a; // don't care about lower 3 bits
   4'b01??: out = b; //the ? is same as z in a number
   4'b001?: out = c;
   default : out = $display ("Error xxxx does matches 0000");
endcase
```

Looping Statements

Looping statements appear inside a procedural blocks only, Verilog has four looping statements like any other programming language.

forever
repeat
while
for

The forever statement

The **forever** loop executes continually, the loop never ends

syntax : **forever** <statement>

Example : Free running clock generator

```
initial begin
   clk = 0;
   forever #5 clk = !clk;
end
```

The repeat statement

The **repeat** loop executes statement fixed <number> of times

syntax : **repeat** (<number>) <statement>

Example:

```
if (opcode == 10) //perform rotate
  repeat (8) begin
     temp = data[7];
     data = {data<<1,temp};
  end
```

The while loop statement

The **while** loop executes as long as an <expression> evaluates as true

syntax : **while** (<expression>) <statement>

Example :

```
loc = 0;
if (data = 0) // example  of a 1 detect shift value
   loc = 32;
else while (data[0] == 0); //find the first set bit
begin
   loc = loc + 1;
   data = data << 1;
end
```

The for loop statement

The for loop is same as the for loop used in any other programming language. Executes an <initial assignment> once at the start of the loop. Executes the loop as long as an <expression> evaluates as true. Executes a <step assignment> at the end of each pass through the loop.

syntax : **for** (<initial assignment>; <expression>, <step assignment>) <statement>

Example :

```
for (i=0;i<=63;i=i+1)
   ram[i] <= 0; // Inialize the RAM with 0
```

Continuous Assignment Statements

Continuous assignment statements drives nets (wire data type). They represent structural connections.

They are used for modeling Tri-State buffers.
They can be used for modeling combinational logic.
They are outside the procedural blocks (always and initial blocks).
The continuous assign overrides and procedural assignments.
The left-hand side of a continuous assignment must be net data type.

syntax : **assign** (strength, strength) # delay net = expression;

Example: 1-bit Adder

```
module adder (a,b,sum,carry);
    input a, b;
    output sum, carry;
    assign #5 {carry,sum} = a+b;
endmodule
```

Example: Tri-State Buffer

```
module tri_buf(a,b,enable);
    input a, enable;
    output b;
    assign b = (enable) ? a : 1'bz;
endmodule
```

Propagation Delay

Continuous Assignments may have a delay specified, Only one delay for all transitions may be specified. A minimum:typical:maximum delay range may be specified.

Example : Tri-State Buffer

```
module tri_buf(a,b,enable);
    input a, enable;
    output b;
    assign #(1:2:3) b = (enable) ? a : 1'bz;
endmodule
```

Procedural Block Control

Procedural blocks become active at simulation time zero, Use level sensitive even controls to control the execution of a procedure.

```
always @ (d or enable)
if (enable)
   q = d;
```

An event sensitive delay at the begining of a procedure, any change in either d or enable satisfies the even control and allows the execution of the statements in the procedure. The procedure is sensitive to any change in d or enable.

Combo Logic using Procedural Coding

To model combinational logic, a procedure block must be sensitive to any change on the input.

Example : 1-bit Adder

```
module adder (a,b,sum,carry);
   input a, b;
   output sum, carry;
   reg sum, carry;
   always @ (a or b)
   begin
      {carry} = a + b;
endmodule
```

The statements within the procedural block work with entire vectors at a time.

Example : 4-bit Adder

```
module adder (a,b,sum,carry);
   input [3:0] a, b;
   output [3:0] sum;
   output carry;
   reg [3:0] sum;
   reg carry;
   always @ (a or b)
   begin
endmodule
```

A procedure can't trigger itself

Once cannot trigger the block with the variable that block assigns value or drive's.

```
always @ (clk)
   #5 clk = !clk;
```

Procedural Block Concurrency

If we have multiple always blocks inside one module, then all the blocks (i.e. all the always blocks) will start executing at time 0 and will continue to execute concurrently. Sometimes this is leads to race condition, if coding is not done proper.

```
module procedure (a,b,c,d);
  input a,b;
  output c,d;

  always @ ( c)
    a = c;

  always @ (d or a)
    b = a &d;

endmodule
```

Procedural Timing Control

Procedural blocks and timing controls.

Delays controls.

Edge-Sensitive Event controls

Level-Sensitive Event controls-Wait statements

Named Events

Delay Controls

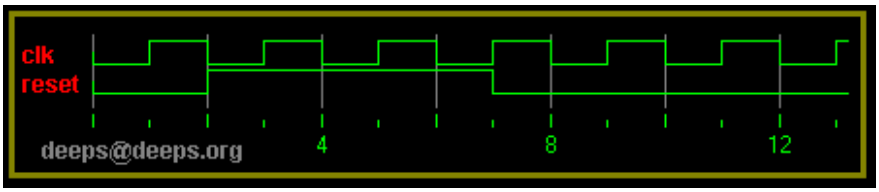
Delays the execution of a procedural statement by specific simulation time.

#<time> <statement>;

Example :

```
module clk_gen (clk,reset);
   output clk,reset;
   reg clk, reset;
   initial begin
       clk = 0;
       reset = 0;
       #2 reset = 1;
       #5 reset = 0;
   end
   always
   #1 clk = !clk;
endmodule
```

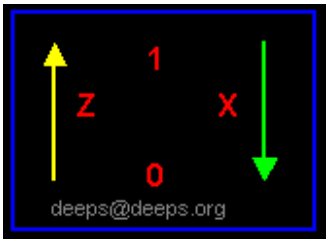
Waveform



Edge sensitive Event Controls

Delays execution of the next statement until the specified transition on a signal.

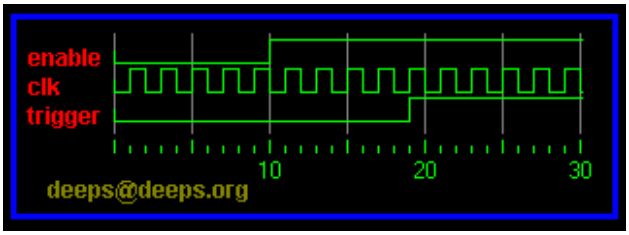
@ (<posedge>|<negedge> signal) <statement>;



Example :

```
always @ (posedge enable)
begin
    repeat (5) // Wait for 5 clock cycles
        @ (posedge clk) ;
    trigger = 1;
end
```

Waveform



Level-Sensitive Even Controls (Wait statements)

Delays execution of the next statement until the <expression> evaluates as true

syntax: wait (<expression>) <statement>;

Example :

```
while (mem_read == 1'b1) begin
  wait (data_ready) data = data_bus;
  read_ack = 1;
end
```

Intra-Assignment Timing Controls

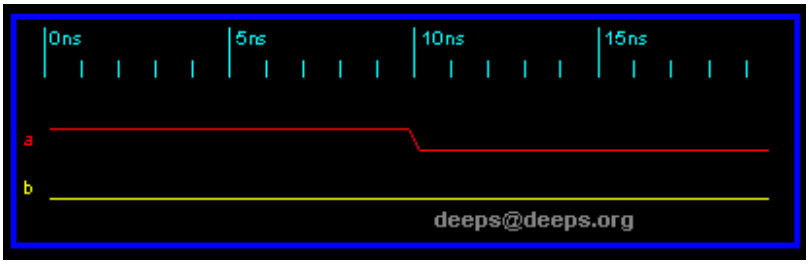
Intra-assignment controls evaluate the right side expression right always and assigns the result after the delay or event control.

In non-intra-assignment controls (delay or event control on the left side) right side expression evaluated after delay or event control.

Example :

```
initial begin
   a = 1;
   b = 0;
   a = #10 0;
   b = a;
end
```

Waveform



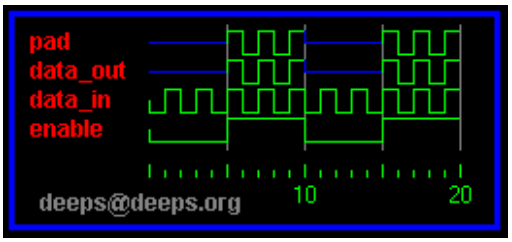
Modeling Combinational Logic with Continuous Assignments

Whenever any signal changes on the right hand side, the entire right-hand side is re-evaluated and the result is assigned to the left hand side

Example : Tri-state buffer

```
module tri_buf (data_in,data_out, pad,enable);
   input data_in, enable;
   output data_out;
   inout pad;
   wire pad, data_out;
   assign pad = (enable) ? data_in : 1'bz;
   assign data_out = pad;
endmodule
```

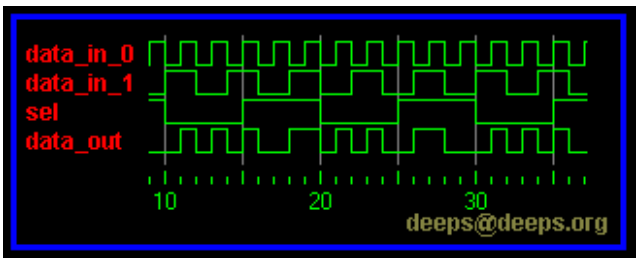
Waveform



Example : 2:1 Mux

```
module mux2x1 (data_in_0,data_in_1, sel, data_out);
   input data_in_0, data_in_1;
   output data_out;
   input  sel;
   wire data_out;
   assign data_out = (sel) ? data_in_1 : data_in_0;
endmodule
```

Waveform



Task and Function

Task

Tasks are used in all programming languages, generally known as Procedures or sub routines. Many lines of code are enclosed in task....end task brackets. Data is passed to the task, the processing done, and the result returned to a specified value. They have to be specifically called, with data in and outs, rather than just ‘‘wired in’’to the general netlist. Included in the main body of code they can be called many times, reducing code repetition.

Task are defined in the module in which they are used. it is possible to define task in separate file and use compile directive 'include to include the task in the file which instantiates the task.

Task can include timing delays, like posedge, negedge, # delay. task can have any number of inputs and outputs.

The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task by the caller are used.

Task can take drive and source global variables, when no local variables are used. When local variables are used, it basically assigned output only at the end of task execution. Task can call another task or function.

Task can be used for modeling both combinational and sequential logic.

A task must be specifically called with a statement, it cannot be used within an expression as a function can.

Syntax

Task begins with keyword task and end's with keyword endtask input and output are declared after the keyword task.

Local variables are declared after input and output declaration.

Example : Simple Task

```
task convert;
  input [7:0] temp_in;
  output [7:0] temp_out;
  begin
    temp_out = (9/5) *( temp_in + 32)
end
endtask
```

Example : Task using Global Variables

```
task convert;
  begin
    temp_out = (9/5) *( temp_in + 32);
  end
endtask
```

Calling a Task

Lets assume that task in example 1 is stored in a file called mytask.v. Advantage of coding task in separate file is that, it can be used in multiple module's.

```
module temp_cal (temp_a, temp_b,
                 temp_c, temp_d);
  input [7:0] temp_a, temp_c;
  output [7:0] temp_b, temp_d;
  reg [7:0] temp_b, temp_d;
  `include "mytask.v"

  always @ (temp_a)
    convert (temp_a, temp_b);

  always @ (temp_c)
    convert (temp_c, temp_d);

endmodule
```

Function

A Verilog HDL function is same as task, with very little difference, like function cannot drive more then one output, can not contain delays.

Function is defined in the module in which they are used. it is possible to define function in separate file and use compile directive. Include to include the function in the file which instantiates the task.

Function cannot include timing delays, like posedge, negedge, # delay. Which means that function should be executed in "zero" time delay.

Function can have any number of inputs and but only one output.The variables declared within the function are local to that function. The order of declaration within the function defines how the variables passed to the function by the caller are used.

Function can take drive and source global variables, when no local variables are used. When local variables are used, it basically assigned output only at the end of function execution. Function can be used for modeling combinational logic. Function can call other functions, but cannot call task.

Syntax

Function begins with keyword function and end's with keyword endfunction

input are declared after the keyword function. Ouputs are delcared.

Example : Simple Function

```
function myfunction;
  input a, b, c, d;
  begin
    myfunction = ((a+b) + (c-d));
  end
endfunction
```

Calling a Function

Lets assume that function in above example is stored in a file called myfunction.v. Advantage of coding function in separate file is that, it can be used in multiple module's.

```
module func_test(a, b, c, d, e, f);

   input a, b, c, d, e ;
   output f;
   wire f;
   `include "myfunction.v"

   assign f =  (myfunction (a,b,c,d)) ? e :0;

endmodule
```