VHDL,Verilog and Advanced Verilog

Umabalaji,Assistant Professor/ECE, SCSVMV

December 2020

1 Aim

The high level programming languages permit complex design concepts to be communicated as computer programs likewise VHDL permits the behavior of complex electronic circuits to be captured into a design system for automatic circuit synthesis or for system simulation.

2 Purpose of VHDL and Verilog

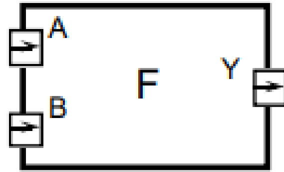
- It is a hardware description language used in electronic design automation to describe digital and mixed signal systems such as Field Programmable Gate Arrays(FPGA) and integrated circuits.
- It is very helpful to think in the view of gates and flip-flops but not as variables or functions.
- It is helpful to know about which part of the circuit is combinatorial and which part is sequential.

3 Why to use an HDL

- It is beneficial to describe problems and faults in the design before implementing it in hardware.
- It is very appropriate to build a prototype of the circuit previously to its manufacturing process because the complexity of an electronic circuit grows exponentially.
- It makes easy for a team of developers to work together.

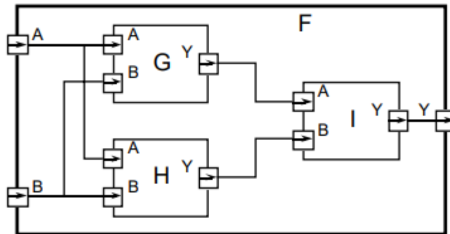
4 How to describe a structure

A digital electronic system can be described as a module with inputs and/or outputs.



In the above diagram, the module F has two inputs A and B and an output Y, which means to say that electrical values on the outputs are some function of the values on the inputs. We specify the module F as a design entity and the inputs and outputs are called ports.

It is very easy to describe the function of a module i.e. how it is composed of sub-modules. Each of the sub-modules is an instance of some entity and the ports of the instances are connected using signals.



The above diagram describes the structural view of description which shows how the entity F might be composed of instances of entity G,H and I. Overall(main) module is F which has two inputs A and B. There are three submodules G,H and I. In many cases, it is not relevant to describe a module structurally. In such cases, a description of the function performed by the module is required, without reference to its actual internal structure. Such a description is called behavioural or functional description.

For example, Consider F modules as a XOR Gate. Then the behavioural description of a F could be a Boolean expression.

$$Y = A . B + A . B$$

In the above exclusive-OR equation, the first sub-module(G) is going to describe the first function A.B and the second sub-module(H) is going to describe the second function A. B. Then both the sub-modules are going to be added in the third sub-module(I). Eventually, the output will be coming out as a function Y which A.B+A. B.

5 Pre MCQ

1. What does a decimal number represents?
   a) Quality
   b) Quantity
   c) Position
   d) None of the above

2. Binary numbers can be converted into equivalent octal numbers by making groups of three bits _____
   a) Starting from the MSB
   b) Starting from the LSB
   c) Ending at the MSB
   d) Ending at the LSB

3. What is the octal equivalent of 58?
   a) 010_2
   b) 110_2
   c) 000_2
   d) 101_2

4. What is the hex equivalent of 916?
   a) 1111_2
   b) 1001_2
   c) 0110_2
   d) 1100_2

5. Which one is the possible technique for representing signed integers?
   a) Signed Magnitude representation
   b) Diminished Radix-complement representation
   c) Radix-Complement representation
   d) All of the above

6. What are the two ways of representing the 0 with signed magnitude representation?
   a) -0 and -0
   b) +0 and +0
   c) -0 and +0
   d) None of the above

7. 2's complement is used to represent signed integers, especially _______ integers.

a) Negative
b) Positive
c) Both A and B
d) None of the above

8. For subtraction of binary number, subtract the _____
a) Minuend from the subtrahend
b) Subtrahend digit from the minuend
c) MSB from the LSB
d) None of the above

9. Floating-point numbers are those numbers which include ______
a) Decimals
b) Fractional parts
c) Integer values
d) All of the above

10. Binary coded decimal or BCD is also known as ______ a) 2841
b) 4821
c) 4281
d) 8421

11. what is the decimal representation of decimal number 5?
a) 0000
b) 1001
c) 0011
d) 0101

12. EXOR is the ____ of the binary number.
a) MSB to the next bit
b) LSB to the next bit
c) MSB of the previous bit
d) LSB of the previous bit

13. Which sign bit is used for representing the positive sign in floating point representation.
a) 0
b) 1
c) either a or b
d) None of the above

14. A basic AND gate consists of _____ inputs and an output
    a) One
    b) Zero
    c) Two
    d) Ten

15. For multiple-input AND and NAND gates, the unused input pin should not be left _______.
    a) ON
    b) Connected
    c) Unconnected
    d) None of the above

16. When a logic circuit diagram is given,You can analyze the circuit to obtain the _____.
    a) Result
    b) Input
    c) Logic Expression
    d) None of the above

17. Boolean algebra is named after _____, who is used it to study human logical reasoning
    a) Anderson,Mary
    b) Acharya Kanad
    c) Dickson,Earle
    d) George Boole

18. A ______ is a table, which consists of every possible combination of inputs and its corresponding outputs.
    a) Last table
    b) Truth table
    c) K-Map
    d) None of the above

19. Canonical form is a unique way of representing ______
    a) SOP
    b) Minterm
    c) Boolean expressions
    d) A Page

20. It is a simple combinational digital circuit built from logic gates
    a) Full adder

b) Half adder
c) Null adder
d) None of the above

21. In a circuit, which subtracts two inputs each of one bit
a) Full subtractor
b) Full adder
c) Half subtractor
d) All of the above

22. It is the converse of decoding and contains 2n(or fewer) input lines and n output lines
a) Subtractor
b) Decoder
c) Multiplexer
d) Encoder

23. It is a very useful combinational circuit used in communication systems
a) Parity Bit checker
b) Parity Bit Generator
c) Both A and B
d) Parity Bit

24. It compares two n-bit values to determine one of them is greater or if they are equal
a) Calculator
b) Multiplexer
c) Comparator
d) None of the above

25. It is a circuit, which has a number of input lines and selection lines with one output line
a) Sequential circuit
b) Multiplexer
c) Counter
d) All of the above

26. It is a circuit, which can remember values for a long time or change values when required
a) Ripple
b) Counter
c) Circuit

d) Memory element

27. It is a sequential circuit that cycles through a sequence of states
a) Multiplexer
b) Demultiplexer
c) Counter
d) Ripple

28. it is a counter where the flip-flops do not change states at exactly the same time, as they do not have a common clock pulse
a) Asynchronous Ripple counter
b) Synchronous Ripple counter
c) Counter
d) None of the above

29. It is a bi-directional counter capable of counting in either of the direction depending on the control signal
a) Up Synchronous Counter
b) Down Synchronous Counter
c) Synchronous Counter
d) All of the above

30. In this logic, output depends not only on the current inputs but also on the past input values. It needs some type of memory to remember the past input values
a) Logical circuit
b) Connected circuit
c) Sequential circuit
d) Parallel circuit

31. There are two types of parity,
a) Even
b) Odd
c) First
d) Both (a) and (b)

32. Decoders often come with an enable signal, so that the device is only activated when the enable equals to ______
a) 2
b) 1
c) 3

d) Either (a) or (b)

33. When more than one input can be active, the priority ______ must be used.
a) Terms
b) Words
c) Encoder
d) None of the above

34. The characteristics equation of any flip-flop describes the _____ of the next state in terms of the present state and inputs.
a) Impact
b) Behavior
c) Path
d) None of the above.

35. The normal data inputs to a flip-flop(D, S and R, J and K, T) are referred to as _____ inputs.
a) Sequential
b) Synchronous
c) Asynchronous
d) Both (a) and (b)

36. A PLA consists of two-level _____ circuits on a single chip.
a) AND-OR
b) NOR-NAND
c) XOR-AND
d) OR-AND

37. It is a single input version of JK Flip-flop formed by typing both the inputs of JK.
a) D Flip-flop
b) S Flip-flop
c) T Flip-flop
d) N Flip-flop

38. In flip-flop the ______ arrow shows positive transition on the clock
a) Upward
b) Downward
c) Vertical
d) Horizontal

39. An n-bit register has a group of flip-flops and some ______
   a) Logic gates
   b) Registers
   c) ROM
   d) None of the above

40. A register can also be used to provide data movements.
   a) Parallel Register
   b) Simple Register
   c) Shift Register
   d) All of the above

41. There are ______ basic types of shift registers.
   a) Six
   b) Four
   c) One
   d) Many

42. In this type of counter, the output of the last stage is connected to the D input of the first stage.
   a) Ring counter
   b) Johnson counter
   c) Straight counter
   d) All of the above

43. A device that exhibits two different stable states and functions as memory element in a binary system is known as ______
   a) Registers
   b) Flip-flops
   c) VLSI
   d) Both (b) and (c)

44. ______ organization is essentially an array of selectively open and closed unidirectional contacts.
   a) ROM
   b) RAM
   c) Computer
   d) All of the above

45. A memory stores data for processing and the instructions for ______.
   a) Result
   b) Execution

c) Process
d) All of the above

46. Half adder consists of _______ & ______ gates.
a) EX-OR & AND
b) EX-OR & OR
c) EX-OR & NOT
d) None of this

47. _____ are used for converting one type of number system into another form.
a) Encoder
b) Logic gate
c) Half adder
d) Full adder

48. A register is a group of _______
a) OR gates
b) OR & AND gates
c) Flip-flops
d) None of these

49. A flip-flop has _____ stable states.
a) Two
b) Three
c) Four
d) Five

50. It does not have any external gate.
a) Simple Register
b) Buffers
c) Memory
d) RAM

6 Prerequisite

- The prerequisite for VHDL programming are the fundamentals of digital electronics and digital circuit design.
- To fully understand the VHDL programs, it is very important to have adequate knowledge of Boolean algebra, logic gates, combinational and sequential logic gates

7 VHDL Content

7.1 Introduction

Very High Speed Integrated Circuit(VHSIC) is a language for describing digital electronic systems and was initiated at 1980. It is very clear that always there was a need for a standard language for describing the structure and function of Integrated Circuits(ICs). Hence it was developed and approved as a standard by the Institute of Electrical and Electronic Engineers(IEEE) in the US.

VHDL is specifically originated to fulfill certain design needs in the entire design process.

1. HDL provides description of the structure of a design i.e. how it is decomposed into sub-designs.

2. It provides the specification of the function of design using familiar programming language forms.

3. Very importantly, it provides a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.

It is a hardware description language which uses the syntax of ADA. It is used for many applications like for describing hardware, as a modeling language, for simulation of hardware, for early performance estimation of system architecture, for synthesis of hardware,for fault simulation, test and verification of designs.

It has been designed and optimized for describing the behavior of digital circuits and systems. It combines features of the following:

- A simulation modeling language
- A design entry language
- A test language
- A netlist language
- A standard language

7.2 How is VHDL used?

It is a general purpose programming language optimized for electronic circuit design. As such, there are many points in the overall design process at which VHDL can help.

- **Design Specification** - HDL plays a major role in to capture the performance and interface requirements of each component in a large circuit. Using a top-down approach to design, a system designer may define the

interface to each component in the system, and describe the acceptance requirements of those components in the form of a high-level test bench. The interface definition (typically expressed as a VHDL entity declaration) and high-level performance specification (the test bench) can then be passed on to other team members for completion or refinement.

- **Design capture** - It is a phase in which the details of the system are entered in a computer based design system. In this phase, you may express your design (or portions of your design) as schematics (either board-level or purely functional) or using VHDL descriptions. The design capture phase may include tools and design entry methods other than VHDL. In many cases, design descriptions written in VHDL are combined with other representations, such as schematics, to form the complete system.

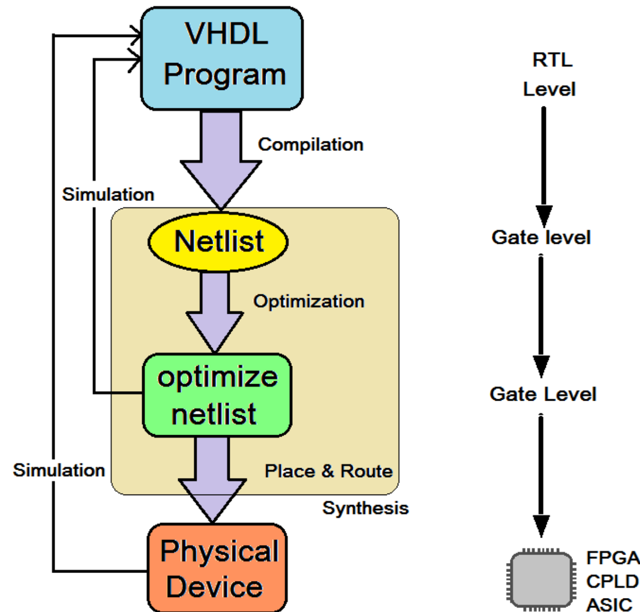
- **For design simulation** - Once entered into a computer based system design, it will be easy to simulate the operation of a circuit to find out if it will meet the functional and timing requirements developed during the specification process.

- **Design Documentation** - The structured programming features of VHDL, coupled with its configuration management features, make VHDL a natural form in which to document a large and complex circuit. The value of using a high-level language such as VHDL for design documentation is pointed out by the fact that the U.S. Department of Defense now requires VHDL as the standard format for communicating design requirements between government subcontractors.

- **As an alternative to schematics** - Schematics have long been a part of electronic system design, and it is unlikely that they will become extinct anytime soon. Schematics have their advantages, particularly when used to depict circuitry in block diagram form. For this reason many VHDL design tools now offer the ability to combine schematic and VHDL representations in a design.

- **As an alternative to proprietary languages** - Proprietary languages such as PALASM, ABEL, CUPL and Altera's AHDL have been developed over the years by PLD device vendors and design tool suppliers, and remain in widespread use today. In fact, there are probably more users of PLD-oriented proprietary languages in the word today than all other HDLs (including Verilog and VHDL) combined.

7.3 VHDL Design flow



- The Design flow starts with writing a HDL code of your circuit design. There are various companies in the market like XILINX, Altera etc. provide their own software development tools to edit, compile and simulate the HDL code and eventually the circuit is described in RTL at this level.
- The compiled HDL code generates netlist at Gate level. Especially, the compiler in the tool high-level VHDL code in RTL to Gate level.
- The generated netlist is further optimized to get netlist again at gate level and the simulation of the design is done at this stage.
- Finally a physical device is implemented on CPLD/FPGA from this optimized netlist. Once again the final device can be simulated and verified.

7.4 VHDL program Structure

- All the VHDL program consists of two components - Entity and Architecture
- It may have additional components like configuration, package declaration, body, etc. as per requirements.

The structure of the program is:

```
LIBRARY library name;
use library_name.package_name.package_parts;

ENTITY entity_name IS
PORT(port name : port_mode port_type;
PORT(port name : port_mode porttype;
.
.
.
);
END entity_name;

ARCHITECTURE archi_name of entity_name is
declarations
BEGIN
code(sequential or concurrent statements)
.
.
.
END archi_name;
```

7.5 Entity and Architecture

To describe the entity, VHDL provides five different types of primary constructs, called design units.
They are:

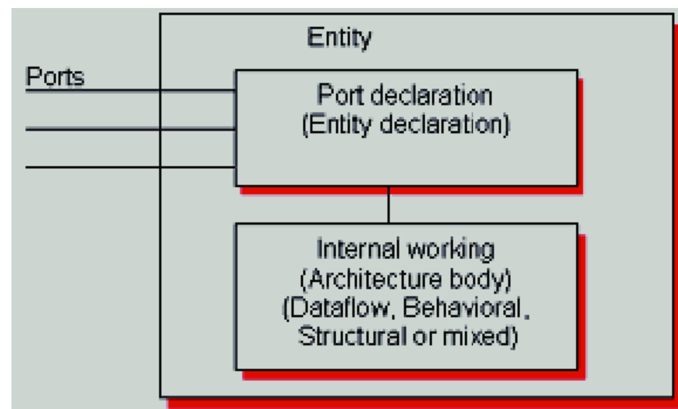
1. Entity declaration
2. Architecture body
3. Configuration declaration
4. Package declaration
5. Package Body

1. **Entity declaration** It describes the external view of the entity. For example, the input and output signal names.
2. **Architecture body** It contains the internal description of the entity. For example, a set of concurrent or sequential statements that represents the behavior of the entity.

3. **Configuration declaration** It is mainly used to create a configuration for an entity. It specifies the binding of one architecture body from the many architecture bodies that may be associated with the entity. Also specifies the bindings of components used in the selected architecture body to other entities.

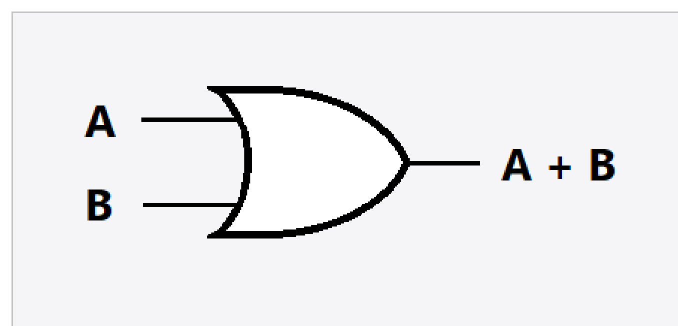
4. **Package declaration** It confined with a set of related declarations, such as type declarations, subtype declarations, and subprogram declarations, which can be shared across two or more design units.

5. **Package body** It contains the definitions of subprograms declared in a package declaration.



7.5.1 Very simple Entity declaration and Architecture Body

Using an OR gate, we can see how the entity is declared. Here in this diagram,



the OR gate has two inputs and one output. So the entity has totally 3 input and output signals.

A & B = input signals
Y(A+B) = output signal

As per the VHDL program structure, the entity declaration for the above OR gate will be,

```
entity OR_gate is
port(A,B : in std_logic; Y : out std_logic);
end OR_gate;
```

For the same OR gate, let us conclude the architecture body,

```
architecture dataflow of OR_gate is
begin
Y <= A AND B;
end dataflow;
```

The above is the very simple explanation of the VHDL code as per the program structure. We can verify this coding by using the truth table of OR gate. Similar way, the same can be applicable to all the digital logic concepts.

7.6 Types of modelling in VHDL

There are three modelling styles are available in VHDL.

1. Dataflow Modelling
2. Behavioral Modelling
3. Structural Modelling

An architecture can be written in one of these three basic coding styles. The difference between these styles is based on the type of concurrent statements used:

- A **dataflow architecture** uses only concurrent signal assignment statements.
- A **behavioral architecture** uses only process statements
- A **structural architecture** uses only component instantiation statements

It is also preferable to go with mixed style of modelling to express the digital concepts in the programmer's view.

7.6.1 Behavioral Modeling

Behavioural modeling describes the highest level of abstraction in the circuit design using VHDL, where the circuit to be designed is described in terms of behaviour of output in response to the changes in the input. It describes a circuit at a functional level. It is a sequential style of modeling wherein a sequence of statements are available in this. In this behavioral modeling let's see the explanation of process statement which is the major syntax.

Process statement

It contains sequential statements that express the functionality of a portion of an entity in sequential terms.

Syntax of process statement

[process-label:] **process** [(sensitivity-list)] [is]
process-item-declarations

begin
sequential-statements;
(variable-assignment-statement
signal-assignment-statement
wait-statement
if-statement
case-statement
loop-statement
null-statement
exit-statement
next-statement
assertion-statement
report statement
procedure-call-statement
return-statement
end process [process-label];

A set of signals to which the process is sensitive is defined by the sensitivity list. Exactly, each time an event occurs on any of the signals in the sensitivity list, the sequential statements within the process are executed in a sequential order, that is, in the order in which they appear. Eventually, the process statement suspends after executing the last sequential statement and will wait for another event to occur on a signal in the sensitivity list.

For example

```
architecture behavior of ent is
begin
```

```
process(A,B,C,D)
variable t1,t2:BIT;
begin
t1 := A and B;
t2 := C and B;
t1 := t1 or t2;
Z <= not t1;
end process;
end behavior;
```

In the above example, the process statement has four signals in its sensitivity list and one variable declaration. If an event occurs on any of the signals A,B,C,D, the process is executed, then one by one statement is executed until the process suspends and wait for another event to occur on a signal in the sensitivity list.

Variable Assignment Statement

In this assignment statement, the variables can be declared and used inside a process statement. It has the form

```
variable-object := expression;
```

This expression is evaluated when the statement is executed and the computed value is assigned to the variable object instantaneously. Variables are created at the time of elaboration and retain their values throughout the entire simulation run. This is due to the process is never excited, it will be either in an active state or in a suspended state for an event to occur.

Signal Assignment Statement Here in this, signals are assigned values using a signal assignment statement.
The general form is

```
signal-object <= expression [ after delay-value];
```

It can appear either inside or outside the process statement. If it occurs outside, it is contemplated to be a concurrent signal assignment statement, perhaps, if it occurs inside a process, it is contemplated to be a sequential signal assignment statement and is executed in sequence w.r.t other sequential statements which appear within that process.
When a signal assignment statements is executed, the value of the expression is computed, and this value is scheduled to be assigned to the signal after the specified delay.

For example

```
counter <= counter + "0010';          Assign after a delay
```

Wait Statemenet

This provides an alternative way to suspend the execution of a process. Basically, there are three forms of the wait statement.

```
wait on sensitivity-list;
wait until boolean-expression;
wait for time-expression;
```

The above all the three can be combined in a single wait statement, like,

```
wait on sensitivity-list until boolean-expression for time-expression;
```

For example

```
wait on A,B,C;
wait until A=B;
```

The execution of the wait statement in the first statement causes the enclosing process to suspend and then wait for an event to occur on signal A,B or C. Once that happens, the process resumes execution from the next statement onwards.

IF statement

It selects the sequence of statements for execution based on the value of a condition. The condition can be any expression that evaluates to a Boolean value. The general form of an if statement is

```
if boolean-expression then
sequential-statements
{ elsif boolean-expression then
sequential statements }
else
sequential-statements
end if;
```

This is executed by checking each condition sequentially until the first true condition is found; then, the set of sequential statements associated with this condition is executed. The if statement can have zero or more elsif clauses and an optional else clause. It is also a sequential statement, therefore the arbitrary nesting of if statements are allowed as given in the above general form.

For example

```
process(CLK, RESET)
begin
if RESET = '1' then
```

```
COUNT <= 0;
elseif CLK'event and CLK='1' then
if (COUNT ¿= 9) then
COUNT ¡= 0;
else
COUNT ¡= COUNT + 1;
end if;
end if;
end process;
```

Case Statement

It selects one of the branches for execution based on the value of the expression. The expression value must be of a discrete type or of a one-dimensional array type. Choices may be expressed as single values, as a range of values or by using the others clause. The general form of this is

```
case expression is
when choices => sequential-statements
when choices => sequential-statements
when others => sequential-statements
end case;
```

For example

```
case SEL is
when "01" => Z <= A;
when "10" => Z <= B;
when others => Z <= 'X';
end case;
```

NULL statement

It is a sequential statement that does not cause any action to take place, execution continues with the next statement. The general form is

```
null
```

Loop Statement

It is used to iterate through a set of sequential statements. The general form is

```
[loop-label:] iteration-scheme loop
sequential-statements
end loop[loop-label];
```

There are three different iteration schemes. The first is the **for** iteration scheme, which has the general form

```
for identifier in range
```

For example

```
process (A)
begin
Z <= "0000";
for i in 0 to 3 loop
if (A = i) then
Z(i) <= '1';
end if;
end loop;
end process;
```

The second form of the iteration scheme is the **while** scheme, which has the form

while boolean-expression

For example

```
J := 0; SUM := 10;
WH_Loop: while J < 20 loop
SUM := SUM 2;
J := J+3;
end loop;
```

The third form of the iteration scheme is one where no iteration scheme is specified. In this form of loop statement, all statements in the loop body are repeatedly executed until some other action causes the loop to exit. This can be done by an exit statement, a next statement, or a return statement.

Exit statement

It is a sequential statement that can be used only inside a loop.

exit [loop-label] [when condition];

For example

```
loop
wait on A,B;
exit when A=B;
end loop;
```

Next statement

This one is also a sequential statement that can be used only inside a loop.

next [loop-label] [when condition];

This statement results in skipping the remaining statements in the current iteration of the specified loop; execution resumes with the first statement in the next iteration of this loop, if one exists. If no loop label is specified, the innermost loop is assumed. The exit statement will terminate the operation but in contrast with that the next statement causes the current loop iteration of the specified loop to be prematurely terminated and execution resumes with the next statement.

For example

```
for I in 0 to 7 loop
if SKIP = '1' then
next;
else
NBUS <= TABLE(I);
wait for 5 ns;
end if;
end loop;
```

Assertion Statement

These are useful in modeling constraints of an entity. For example, if we want to check some signal values which can lies within a specified range, or check the setup and hold times for signals arriving at the inputs of an entity, if the check fails, a message is reported.

The general form is

```
assert boolean-expression
{ report string-expression }
{ severity expression};
```

If the value of the Boolean expression is false, the report message is printed along with the severity level. The expression in the severity clause must be a value of type SEVERITY-LEVEL. The severity level is typically used by a simulator to initiate appropriate actions depending on its value.

Report statement

This can be used to display a message. It is similar to an assertion statement, but without the assertion check. The general form is

```
report string-expression
severity expression
;
```

The expression in the severity clause, if present, must be of the pre-defined type SEVERITY_LEVEL. If not present, the default severity level of NOTE is used.

7.6.2 Dataflow Modeling

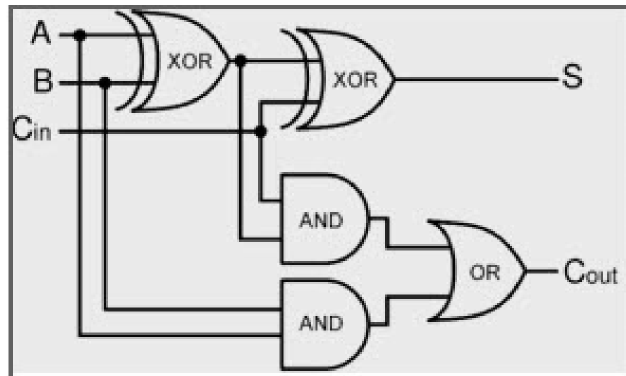
This kind of modeling absolutely conveying about a system in terms of how data flows through the system. Data dependencies in the description match those in a typical hardware implementation. It directly implies the corresponding gate-level implementation. It consists of one or more signal assignment statements. This style is very nearest to RTL description of the circuit.

One of the essential technique for modeling the dataflow is using the **concurrent signal assignment** statement. The architecture body definitely contains atleast single concurrent signal assignment statement that represents the dataflow of the signal.

The elucidation of this concurrent signal assignment statement is that whenever there is an event (a change of value) on either signal, For example the program contains only A and B, then the expression on the right side is evaluated.

Event on (A) or (B) = evaluation on the right side

Let us consider the example of **Full adder**.



Here, there are 3 inputs & 2 outputs and 2 XOR gates, 2 AND gates & 1

OR gate.
Let us consider the boolean equation of a full adder is,

S = A XOR B XOR Cin
Cout = (A AND B) OR (B AND Cin) OR (Cin AND A)

The above equation can be used to write the VHDL program for a full adder simply to express that concurrent signal assignment statements are executed whenever events occur on signals that are used in their expressions.

Two signal assignment statements(S and Cout) are used to represent the dataflow of the full adder entity. Whenever an event occurs on signals A,B or Cin, expressions of both statements are evaluated. The dataflow model for a 1-bit full adder using VHDL is,

```
entity full_adder is
port(A,B,Cin: in Bit; S,Cout:out Bit);
end full_adder;

architecture fa of full_adder is
begin
S <= A xor B xor Cin;
Cout <= (A and B) or (B and Cin) or (Cin and A);
end fa;
```

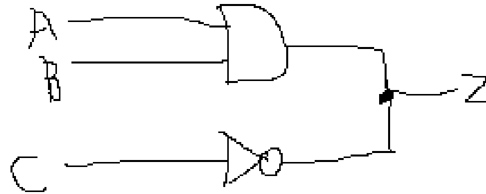
It is very substantial to note down that whether the signal assignment statement is inside or outside the process statement.

- The signal assignment statement inside the process statement is called sequential signal assignment statement.
- while the signal assignment statements that appear outside of a process are called concurrent signal assignment statements.

Concurrent signal assignment statements are event-triggered, that is, they are executed whenever there is an event on a signal that appears in its expression.

Sequential signal assignment statements are not event-triggered and are executed in sequence in relation to the other sequential statements that appear within the process.

In some cases, each concurrent signal assignment statement creates a driver for the signal being assigned. We should look after when there is more than one assignment to the same signal. In these cases, the signal has more than one driver(process that assigns values to the signal) and a specific mechanism is needed to compute the effective value of the signal.



In this diagram, we have one AND gate(with two inputs) and one NOT gate. There are 3 drivers for signal Z. How is the value of Z determined? It is resolved by using a user-defined resolution function that considers the values of both the drivers for Z and determines the effective value.

Consider the following architecture body

```
architecture example1 of archi is
begin
Z <= ’1’ after 2ns,’0’ after 5ns,’1’ after 10ns;
Z <= ’0’ after 4ns, ’1’ after 5ns, ’0’ after 20ns;
Z <= ’1’ after 10ns, ’0’ after 20ns;
end archi;
```

Each driver has a sequence of transactions where each transaction defines the value to appear on the signal and the time at which it is to appear. The resolution function resolves the value for the signal Z from the current value of each of its drivers.

The very interesting concept in VHDL is the **conditional Signal Assignment Statement**. It selects different values for the target signal based on the specified, possibly different, conditions. It is like an if statement in major high level programming languages.
Whenever an event occurs on a signal used in either any of the waveform expressions or any of the conditions, the conditional signal assignment statement is executed by evaluating the conditions one at a time.

Example

Z <= IN0 after 10ns when S0 = '0' and S1 = '0' else
IN1 after 10ns when S0 = '1' and S1 = '0' else
IN2 after 10ns when S0 = '0' and S1 = '1' else
IN3 after 10ns;

In the above example, the statement is executed any time an event occurs on signals IN0,IN1,IN2,IN3,S0 or S1. The first condition is checked, if it is false, then the second condition is checked, if it even false, then the third condition is checked and so on. Eventually, S0 = '0' and S1 = '1', then the value of IN2 is scheduled to be assigned to signal Z after 10ns.

The next interesting concept in dataflow modeling is **Selected Signal Assignment Statement**. This select different values for a target signal based on the value of a select expression. Whenever an event occurs on a signal in the select expression or on any signal used in any of the waveform expressions, the statement is executed.

7.6.3 Structural Modeling

In structural style of modeling, an entity is described as a set of interconnected components. The top-level design entity's architecture describes the description of lower-level design entities. This is most useful and efficient when a complex system is described as an interconnection of moderately complex design entities. This approach allows each design entity to be independently designed and verified before being used in the higher level description.

There are two major factors to be considered while writing the coding in structural modeling,

- Component declaration
- Component instantiation

Let us consider an example of structural modeling,

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

entity half_adder is
port (a, b: in std_logic;
sum, carry_out: out std_logic);
end half_adder;

architecture structure of half_adder is
```

```
component xor_gate
port (i1, i2: in std_logic;
o1: out std_logic);
end component;

component and_gate
port (i1, i2: in std_logic;
o1: out std_logic);
end component;

begin
u1: xor_gate port map (i1 => a, i2 => b, o1 => sum);
u2: and_gate port map (i1 => a, i2 => b, o1 => carry_out);
end structure;
```

In the above example, the program starts with an usual entity which describes the number of signals, that is, input and output ports. Then followed by architecture body which has two separation, one is the component declaration and the another one is the component instantiation part.

The circuit diagram of a half-adder consists of one xor gate and one or gate. So, the major criteria to proceed these gates are first one is component declaration and next one is the component instantiation.

The component declaration of xor gate is

```
component xor_gate
port (i1, i2: in std_logic;
o1: out std_logic);
end component;
```

The component declaration declares the name and the interface of a component. The interface specifies the mode and type of ports.

Component instantiation of xor gate is

```
u1: xor_gate port map (i1 => a, i2 => b, o1 => sum);
```

It defines a subcompoent of the entity in which it appears. It associates the signals in the entity with the ports of that subcomponent.

The same way if you take the AND gate, it is also categorized with component declaration and component instantiation in the above half adder program.

7.6.4 VHDL delays

In VHDL there are two different kind of delay,

1. Transport delay
2. Inertial delay

Transport delay

It is used to model the delay introduced by wire connection or a PCB connection. It is not the default delay implemented in VHDL and must be specified. This delay model is useful to describe delay line, PCB delay, wire delay. This delay represents pure propagation delay; that is, any changes on an input are transported to the output, after the specified delay. The keyword transport must be used in a signal assignment statement.

For Example

b <= transport a after 20ns;

In the above example, **b** takes the value of **a** after 20ns of transport delay. This means that no matter how fast **a** changes his value, **b** will follow the behavior of **a** after the amount of time specified in the delay statement.

Inertial Delay

The inertial delay model is the default delay implemented in VHDL because it's behavior is very similar to the delay of the device. It models the delays repeatedly found in switching circuits. An input value must be constant for a specified pulse rejection limit duration before the value is allowed to propagate to the output. The value appears at the output after the specified inertial delay. If the input is not stable for the specified limit, no output change occurs. When used with signal assignments, the input value is represented by the value of the expression on the right-hand-side and the output is represented by the target signal. The general form is

signal-object <= [[reject pulse-rejection-limit] inertial]expression after inertial-delay-value;

If no pulse rejection limit is specified, the default pulse rejection limit is the inertial delay value itself. The pulse rejection limit cannot be negative or greater than the value of the inertial delay.

The general form with example is

```
b <= a after 20ns;
```

In the above example **b** take the value of **a** after 20 ns second of inertial delay. This means that if a value varies faster than 20 ns **b** remain unchanged. At simulation start **a** and **b** are 0; **a** change from 0 to 1, and **b** change its value after 20 ns; then **a** changes value going to 0 and then to 1 in 10 ns.

This delay is less than inertial delay of 20 ns. So **b** remains unchanged.

VHDL transport and inertial delay model allow the designer to model different type of behavior on VHDL hardware implementation. They are very useful in test bench modeling and in VHDL macro model delay modeling such as RAM, ROM, and peripheral interfacing.

7.7 VHDL data types

VHDL is a strongly typed language. This means that every object assumes the value of its nominated type. To put it very simply, the data type of the left-hand side (LHS) and right-hand side (RHS) of a VHDL statement must be the same. The VHDL 1076 specification describes four classes of data types.

- Scalar types represent a single numeric value or, in the case of enumerated types, an enumeration value. The standard types that fall into this class are integer, real (floating point), physical, and enumerated. All of these basic types can be thought of as numeric values.
- Composite types represent a collection of values. There are two classes of composite types: arrays containing elements of the same type, and records containing elements of different types.
- Access types provide references to objects in much the same way that pointer types are used to reference data in software programming languages.
- File types reference objects (typically disk files) that contain a sequence of values.

Integer type

The maximum range of a VHDL integer type is $(2_{31}1)$ to 2_{31}-1 . Integer types are defined as subranges of this anonymous built-in type. Multidigit numbers in VHDL can include underscores (_) to make them easier to read. VHDL Compiler encodes an integer value as a bit vector whose length is the minimum necessary to hold the defined range. VHDL Compiler encodes integer ranges that include negative numbers as 2's-complement bit vectors.
The syntax of an integer type definition is

```
type type_name is range integer_range ;
```

type_name is the name of the new integer type, and integer_range is a subrange of the anonymous integer type. Example 4–5 shows some integer type definitions.

Array type

An array is an object that is a collection of elements of the same type. VHDL supports N-dimensional arrays, but VHDL Compiler supports only one-dimensional arrays. Array elements can be of any type. An array has an index whose value selects each element. The index range determines how many elements are in the array and their ordering (low to high, or high $_{\text{downto}}$ low). An index can be of any integer ype. You can declare multidimensional arrays by building one-dimensional arrays where the element type is another one-dimensional array.

For example

```
type BYTE is array (7 downto 0) of BIT;
type VECTOR is array (3 downto 0) of BYTE;
```

Record types

A record is a set of named fields of various types, unlike an array, which is composed of identical anonymous entries. A record's field can be of any previously defined type, including another record type.

Predefined datatypes

IEEE VHDL describes two site-specific packages, each containing a standard set of types and operations: the STANDARD package and the TEXTIO package.
The STANDARD package of data types is included in all VHDL source files by an implicit use clause. The TEXTIO package defines types and operations for communication with a standard programming environment (terminal and file I/O). This package is not needed for synthesis, and therefore VHDL Compiler does not support it.

7.8 VHDL operators

Highest precedence first,
left to right within same precedence group,
use parenthesis to control order.
Unary operators take an operand on the right.
"result same" means the result is the same as the right operand.
Binary operators take an operand on the left and right.
"result same" means the result is the same as the left operand.

* multiplication, numeric * numeric, result numeric
/ division, numeric / numeric, result numeric
mod modulo, integer mod integer, result integer
rem remainder, integer rem integer, result integer

+ unary plus, + numeric, result numeric
- unary minus, - numeric, result numeric

+ addition, numeric + numeric, result numeric
- subtraction, numeric - numeric, result numeric
& concatenation, array or element array or element,

sll shift left logical, logical array sll integer, result same
srl shift right logical, logical array srl integer, result same
sla shift left arithmetic, logical array sla integer, result same
sra shift right arithmetic, logical array sra integer, result same
rol rotate left, logical array rol integer, result same
ror rotate right, logical array ror integer, result same

= test for equality, result is boolean
/= test for inequality, result is boolean
< test for less than, result is boolean
<= test for less than or equal, result is boolean
> test for greater than, result is boolean
>= test for greater than or equal, result is boolean

and logical and, logical array or boolean, result is same
or logical or, logical array or boolean, result is same
nand logical complement of and, logical array or boolean, result is same
nor logical complement of or, logical array or boolean, result is same
xor logical exclusive or, logical array or boolean, result is same
xnor logical complement of exclusive or, logical array or boolean, result is same

7.8.1 VHDL Functions

The function is a subprogram that either defines an algorithm for computing values or describes a behavior. The important feature of functions is that they are used as expressions that return values of specified type. This is the main difference from another type of subprograms: procedures, which are used as statements.

The result returned by a function can be of either scalar or complex type.

Functions can be either pure (which is default) or impure. Pure functions always return the same value for the same set of actual parameters. Impure

functions may return different values for the same set of parameters. Additionally, an impure function may have side effects", like updating objects outside of their scope, which is not allowed for pure functions.

The function definition consists of two parts:

- function declaration, which consists of name, parameter list and type of the values returned by the function;
- function body, which contains local declarations of nested subprograms, types, constants, variables, files, aliases, attributes and groups, as well as sequence of statements specifying the algorithm performed by the function.

The function declaration is optional and function body, which contains a copy of it, is sufficient for correct specification. However, if a function declaration exists, the function body declaration must appear in the given scope.

Function Declaration

The function declaration can be preceded by an optional reserved word pure or impure, denoting the character of the function. If the reserved word is omitted it is assumed by default that the function is pure.

The function name, which appears after the reserved word function, can be either an identifier or an operator symbol (if the function specifies the operator). Specification of new functions for existing operators is allowed in VHDL and is called operator overloading. See respective topic for details.

The parameters of the function are by definition inputs and therefore they do not need to have the mode (direction) explicitly specified. Only constants, signals and files can be function parameters. The object class is specified by a reserved word (constant, signal or file, respectively) preceding the parameter's name. If no reserved word is used, it is assumed by default that the parameter is a constant.

In case of signal parameters the attributes of the signal are passed into the function, except for 'STABLE, 'QUIET, 'TRANSACTION and 'DELAYED, which may not be accessed within the function.

If a file parameter is used, it is necessary to specify the type of the data appearing in the opened file.

Example 1 contains several examples of function declarations.

Function Body

Function body contains a sequence of statements that specify the algorithm to

be realized within the function. When the function is called, the sequence of statements is executed.

A function body consists of two parts: declarations and sequential statements. At the end of the function body, the reserved word end can be followed by an optional reserved word function and the function name. Examples 2 through 4 illustrate the function bodies.

For example

```
type Int_Data is file of NATURAL;
function Func_1 (A,B,X: REAL) return REAL;
function "*" (a,b: Integer_new) return Integer_new; function Add_Signals (signal In1,In2: REAL) return REAL; function End_Of_File (file File_name: Int_Data) return BOOLEAN;
```

The first function above is called Func_1, it has three parameters A, B and X, all of REAL type and returns a value also of REAL type.

The second function defines a new algorithm for executing multiplication. Note that the operator is enclosed in double quotes and plays the role of the function name.

The third function is based on signals as input parameters, which is denoted by the reserved word signal preceding the parameters.

The fourth function declaration is a part of the function checking for end of file, consisting of natural numbers. Note that the parameter list uses the Boolean type declaration.

7.8.2 VHDL procedure

We start our discussion of subprograms with procedures. There are two aspects to using procedures in a model: first the procedure is declared, then elsewhere the procedure is called. The syntax rule for a procedure declaration is

```
subprogram_body
procedure identifier [ ( parameter_interface_list ) ] is
subprogram_declarative_part
begin
{ sequential_statement }
end [ procedure ] [ identifier ] ;
```

The identifier in a procedure declaration names the procedure. The name may be repeated at the end of the procedure declaration. The sequential statements in the body of a procedure implement the algorithm that the procedure is to perform and can include any of the sequential statements that we have

seen in previous chapters. A procedure can declare items in its declarative part for use in the statements in the procedure body. The declarations can include types, subtypes, constants, variables and nested subprogram declarations. The items declared are not accessible outside of the procedure; we say they are local to the procedure. The actions of a procedure are invoked by a procedure call statement, which is yet another VHDL sequential statement.
A procedure with no parameters is called simply by writing its name, as shown by the syntax rule
procedure_call_statement <= procedure_name ;

When the last statement in the procedure is completed, the procedure returns. We can write a procedure declaration in the declarative part of an architecture body or a process. If a procedure is included in an architecture body's declarative part, it can be called from within any of the processes in the architecture body. On the other hand, declaring a procedure within a process hides it away from use by other processes.

For example

```
architecture rtl of control_processor is
type func_code is (add, subtract);
signal op1, op2, dest : integer;
signal Z_flag : boolean;
signal func : func_code;
...
begin
alu : process is
procedure do_arith_op is
variable result : integer;
begin
case func is
when add =<
result := op1 + op2;
when subtract =>
result := op1 – op2;
end case;
dest <= result after Tpd;
Z_flag <= result = 0 after Tpd;
end procedure do_arith_op;
begin
...
do_arith_op;
...
end process alu;
...
end architecture rtl;
```

7.8.3 Resolved signals and library

IEEE Std_Logic_1164 Resolved Subtypes

VHDL provides a very general mechanism for specifying what value results from connecting multiple outputs together. It does this through resolved subtypes and resolved signals, which are an extension of the basic signals we have used in previous chapters. However, most designs simply use the resolved subtypes defined in the standard-logic package, std_logic_1164. In this tutorial, we will restrict our attention to those subtypes.
First, recall that the package provides the basic type std_ulogic, defined as
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '–');
and an array type std_ulogic_vector, defined as
type std_ulogic_vector is array (natural range <>) of std_ulogic;

We have not mentioned it before, but the "u" in "ulogic" stands for unresolved. Signals of these types cannot have multiple sources. The standard-logic package also provides a resolved subtype called std_logic. A signals of this type can have multiple sources. The package also declares an array type of standard-logic elements, analogous to the bit_vector type, for use in declaring array signals:
type std_logic_vector is array (natural range <>) of std_logic;

The standard defines the way in which contributions from multiple sources are resolved to yield the final value for a signal. If there is only one driving value, that value is used. If one driver of a resolved signal drives a forcing value ('X', '0' or '1') and another drives a weak value('W', 'L' r 'H'), the forcing value dominates. On the other hand, if both drivers drive different values with the same strength, the result is the unknown value of that strength ('X' or 'W'). The high-impedance value, 'Z', is dominated by forcing and weak values. If a "don't care" value ('_') is resolved with any other value, the result is the unknown value 'X'. The interpretation of the "don't care" value is that the model has not made a choice about its output state. Finally, if an "uninitialized" value ('U') is to be resolved with any other value, the result is 'U', indicating that the model has not properly initialized all outputs.

In addition to this multivalued logic subtype, the package std_logic_1164 declares a number of subtypes for more restricted multivalued logic modeling. The subtype declarations are
subtype X01 is resolved std_ulogic range 'X' to '1'; — ('X','0','1')
subtype X01Z is resolved std_ulogic range 'X' to 'Z'; — ('X','0','1','Z')
subtype UX01 is resolved std_ulogic range 'U' to '1'; — ('U','X','0','1')
subtype UX01Z is resolved std_ulogic range 'U' to 'Z'; — ('U','X','0','1','Z')

The standard-logic package provides the logical operators and, nand, or, nor, xor, xnor and not for standard-logic values and vectors, returning values

in the range ‘U’, ‘X’, ‘0’ or ‘1’. In addition, there are functions to convert between values of the full standard-logic type, the subtypes shown above and the predefined bit and bit-vector types.

8 Verilog Content

The standard-logic package provides the logical operators and, nand, or, nor, xor, xnor and not for standard-logic values and vectors, returning values in the range ‘U’, ‘X’, ‘0’ or ‘1’. In addition, there are functions to convert between values of the full standard-logic type, the subtypes shown above and the predefined bit and bit-vector types.

Verilog can be used to describe designs at four levels of abstraction:

- Algorithmic level (much like c code with if, case and loop statements)
- Register transfer level (RTL uses registers connected by Boolean equations).
- Gate level (interconnected AND, NOR etc.).
- Switch level (the switches are MOS transistors inside gates)

The language also defines constructs that can be used to control the input and output of simulation.

Verilog has a C-like syntax. However, it is philosophically different than most programming languages since it is used to describe hardware rather than software. In particular:

- Verilog statements are concurrent in nature; except for code between begin and end blocks, there is no defined order in which they execute. In comparison, most languages like C consist of statements that are executed sequentially; the first line in main() is executed first, followed by the line after that, and so on.
- Synthesizable Verilog code is eventually mapped to actual hardware gates. Compiled C code, on the other hand, is mapped to some bits in storage that a CPU may or may not execute.

More recently Verilog is used as an input for synthesis programs which will generate a gate-level description (a netlist) for the circuit. Some Verilog constructs are not synthesizable. Also the way the code is written will greatly effect the size and speed of the synthesized circuit. Most readers will want to synthesize their circuits, so nonsynthesizable constructs should be used only for test benches. These are program modules used to generate I/O needed to simulate

the rest of the design. The words "not synthesizable" will be used for examples and constructs as needed that do not synthesize.

There are two types of code in most HDLs:
Structural, which is a verbal wiring diagram without storage.
assign a=b c — d; /* "—" is a OR */
assign d = e (c);
Here the order of the statements does not matter. Changing e will change a.
Procedural which is used for circuits with storage, or as a convenient way to write conditional logic.
always @(posedge clk) // Execute the next statement on every rising clock edge.
count <= count+1;
Procedural code is written like c code and assumes every assignment is stored in memory until over written. For synthesis, with flip-flop storage, this type of thinking generates too much storage. However people prefer procedural code because it is usually much easier to write, for example, if and case statements are only allowed in procedural code. As a result, the synthesizers have been constructed which can recognize certain styles of procedural code as actually combinational. They generate a flip-flop only for left-hand variables which truly need to be stored. However if you stray from this style, beware. Your synthesis will start to fill with superfluous latches.

Verilog source text files consists of the following lexical tokens:

1. White Space - White spaces separate words and can contain spaces, tabs, new-lines and form feeds. Thus a statement can extend over multiple lines without special continuation characters.

2. Comments - Comments can be specified in two ways (exactly the same way as in C/C++):
   - Begin the comment with double slashes (//). All text between these characters and the end of the line will be ignored by the Verilog compiler.
   - Enclose comments between the characters /* and */. Using this method allows you to continue comments on more than one line. This is good for "commenting out" many lines code, or for very brief in-line comments.

3. Numbers - Number storage is defined as a number of bits, but values can be specified in binary, octal, decimal or hexadecimal. Examples are 3'b001, a 3-bit number, 5'd30, (=5'b11110), and 16'h5ED4, (=16'd24276)

4. Identifiers - Identifiers are user-defined words for variables, function names, module names, block names and instance names.Identifiers begin with a letter or underscore (Not with a number or $) and can include any number of letters, digits and underscores. Identifiers in Verilog are case-sensitive.

5. Operators - Operators are one, two and sometimes three characters used to perform operations on variables. Examples include >, +, , , !=.

6. Verilog Keywords - These are words that have special meaning in Verilog. Some examples are assign, case, while, wire, reg, and, or, nand, and module. They should not be used as identifiers.

8.1 Gate-level Modelling

Primitive logic gates are part of the Verilog language. Two properties can be specified, drive_strength and delay. Drive_strength specifies the strength at the gate outputs. The strongest output is a direct connection to a source, next comes a connection through a conducting transistor, then a resistive pull-up/down. The drive strength is usually not specified, in which case the strengths defaults to strong1 and strong0. Delays: If no delay is specified, then the gate has no propagation delay; if two delays are specified, the first represent the rise delay, the second the fall delay; if only one delay is specified, then rise and fall are equal. Delays are ignored in synthesis. This method of specifying delay is a special case of "Parameterized Modules"

8.1.1 Basic gates

These implement the basic logic gates. They have one output and one or more inputs. In the gate instantiation syntax shown below, GATE stands for one of the keywords and, nand, or, nor, xor, xnor.

Syntax

GATE (drive_strength) (delays)
instance_name1(output, input_1, input_2,..., input_N),
instance_name2(outp,in1, in2,..., inN);
Delays is
(rise, fall) or
rise_and_fall or
(rise_and_fall)

8.1.2 buf,not Gates

These implement buffers and inverters, respectively. They have one input and one or more outputs. In the gate instantiation syntax shown below, GATE stands for either the keyword buf or not

Syntax

GATE (drive_strength) (delays)
instance_name1(output_1, output_2,
..., output_n, input),
instance_name2(out1, out2, ..., outN, in);

8.1.3 Three-State Gates; bufif1, bufif0, notif1, notif0

These implement 3-state buffers and inverters. They propagate z (3-state or high-impedance) if their control signal is deasserted. These can have three delay specifications: a rise time, a fall time, and a time to go into 3-state.

8.2 Data Types

8.2.1 Valueset

Verilog consists of only four basic values. Almost all Verilog data types store all these values:
0 (logic zero, or false condition)
1 (logic one, or true condition)
x (unknown logic value) x and z have limited use for synthesis.
z (high impedance state)

8.2.2 Wire

A wire represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block. See “Functions” on p. 19, and “Procedures: Always and Initial Blocks” on p. 18. A wire does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module. Other specific types of wires include:
wand (wired-AND);:the value of a wand depend on logical AND of all the drivers connected to it.
wor (wired-OR);: the value of a wor depend on logical OR of all the drivers connected to it.
tri (three-state;): all drivers connected to a tri must be z, except one (which determines the value of the tri)
Syntax
wire [msb:lsb] wire_variable_list;
wand [msb:lsb] wand_variable_list;
wor [msb:lsb] wor_variable_list;
tri [msb:lsb] tri_variable_list;

8.2.3 Reg

Declare type reg for all data objects on the left hand side of expressions in inital and always procedures, or functions. See “Procedural Assignments” on page 12. A reg is the data type that must be used for latches, flip-flops and memory. However it often synthesizes into leads rather than storage. In multi-bit registers, data is stored as unsigned numbers and no sign extension is done for what

the user might have thought were two's complement numbers.

Syntax
reg [msb:lsb] reg_variable_list;

8.2.4 Input,Output and Inout

These keywords declare input, output and bidirectional ports of a module or task. Input and inout ports are of type wire. An output port can be configured to be of type wire, reg, wand, wor or tri. The default is wire.

Syntax
input [msb:lsb] input_port_list;
output [msb:lsb] output_port_list;
inout [msb:lsb] inout_port_list;

8.2.5 Integer

Integers are general-purpose variables. For synthesois they are used mainly loops-indicies, parameters, and constants. They are of implicitly of type reg. However they store data as signed numbers whereas explicitly declared reg types store them as unsigned. If they hold numbers which are not defined at compile time, their size will default to 32-bits. If they hold constants, the synthesizer adjusts them to the minimum width needed at compilation.

Syntax
integer integer_variable_list;
... integer_constant ... ;

8.2.6 Supply0, Supply1

Supply0 and supply1 define wires tied to logic 0 (ground) and logic 1 (power), respectively.

Syntax
supply0 logic_0_wires;
supply1 logic_1_wires;

8.2.7 Time

Time is a 64-bit quantity that can be used in conjunction with the $time system task to hold simulation time. Time is not supported for synthesis and hence is used only for simulation purposes.

Syntax
time time_variable_list;

8.2.8 Parameter

Parameters allows constants like word length to be defined symbolically in one place. This makes it easy to change the word length later, by change only the parameter.

Syntax

parameter par_1 = value,
par_2 = value, ...;
parameter [range] parm_3 = value

8.3 Operators

8.3.1 Arithmetic Operators

These perform arithmetic operations. The + and - can be used as either unary (-z) or binary (x-y) operators.

Operators
+ (addition)
- (subtraction)
(multiplication)
/ (division)
% (modulus)

8.3.2 Relational Operators

Relational operators compare two operands and return a single bit 1or 0. These operators synthesize into comparators. Wire and reg variables are positive Thus (-3'b001) = = 3'b111 and (-3d001)>3d110. However for integers -1< 6.

< (less than)
<= (less than or equal to)
> (greater than)
>= (greater than or equal to)
== (equal to)
!= (not equal to)

8.3.3 Bit-wise Operators

Bit-wise operators do a bit-by-bit comparison between two operands.

$\sim (bitwiseNOT)$
$\&(bitwiseAND)$
$| \, (bitwiseOR)$
$\wedge \, (bitwiseXOR)$
$\wedge \, or \wedge \sim (bitwiseXNOR)$

8.3.4 Logical Operators

Logical operators return a single bit 1 or 0. They are the same as bit-wise operators only for single bit operands. They can work on expressions, integers or groups of bits, and treat all values that are nonzero as "1". Logical operators are typically used in conditional (if ... else) statements since they work with expressions.

! (logical NOT)
&& (logical AND)
$|| \, (logicalOR)$

8.3.5 Reduction Operators

Reduction operators operate on all the bits of an operand vector and return a single-bit value. These are the unary (one argument) form of the bit-wise operators above.

& (reduction AND)
$| \, (reductionOR)$
$\sim \&(reductionNAND)$
$\sim | \, |(reductionNOR)$
$\wedge \, (reductionXOR)$
$\sim \wedge or \wedge \sim (reductionXNOR)$

8.3.6 Shift Operators

Shift operators shift the first operand by the number of bits specified by the second operand. Vacated positions are filled with zeros for both left and right shifts (There is no sign extension).

<< (shift left)
>> (shift right)

8.3.7 Concatenation Operator

The concatenation operator combines two or more operands to form a larger vector.

{n{item}} (n fold replication of an item)

8.3.8 Conditional Operator: “?”

Conditional operator is like those in C/C++. They evaluate one of the two expressions based on a condition. It will synthesize to a multiplexer (MUX).

(cond) ? (result if cond true):
(result if cond false)

8.3.9 Operator Precedence

The following table shows the precedence of operator from highest to lowest. Operators on the same level evaluate from left to right. It is strongly recommended to use parentheses to define order of precedence and improve the readability of your code.

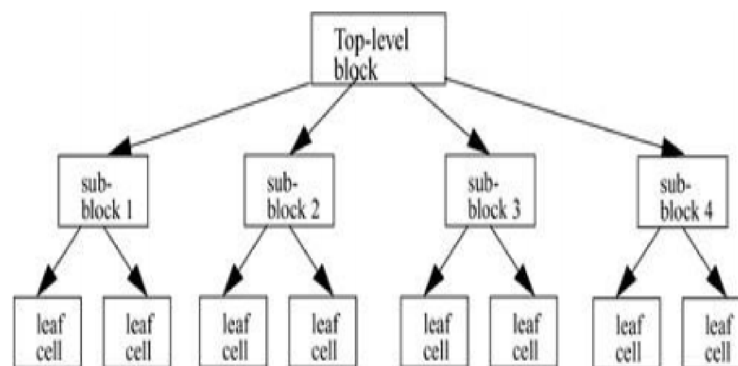
<table>
<tr><th>Operator</th><th>Name</th></tr>
<tr><td>[ ]</td><td>bit-select or part-select</td></tr>
<tr><td>( )</td><td>parenthesis</td></tr>
<tr><td>!, ~</td><td>logical and bit-wise NOT</td></tr>
<tr><td>&, |, ~&, ~|, ^, ~^, ^~</td><td>reduction AND, OR, NAND, NOR, XOR, XNOR;
If X=3’B101 and Y=3’B110, then X&Y=3’B100, X^Y=3’B011;</td></tr>
<tr><td>+, -</td><td>unary (sign) plus, minus; +17, -7</td></tr>
<tr><td>{ }</td><td>concatenation; {3’B101, 3’B110} = 6’B101110;</td></tr>
<tr><td>{{ }}</td><td>replication; {3{3'B110}} = 9'B110110110</td></tr>
<tr><td>*, /, %</td><td>multiply, divide, modulus; <u>/ and % not be supported for synthesis</u></td></tr>
<tr><td>+, -</td><td>binary add, subtract.</td></tr>
<tr><td><<, >></td><td>shift left, shift right; X<<2 is multiply by 4</td></tr>
<tr><td><, <=, >, >=</td><td>comparisons. Reg and wire variables are taken as positive numbers.</td></tr>
<tr><td>= =, !=</td><td>logical equality, logical inequality</td></tr>
<tr><td>= = =, != =</td><td>case equality, case inequality; <u>not synthesizable</u></td></tr>
<tr><td>&</td><td>bit-wise AND; AND together all the bits in a word</td></tr>
<tr><td>^, ~^, ^~</td><td>bit-wise XOR, bit-wise XNOR</td></tr>
<tr><td>|</td><td>bit-wise OR; AND together all the bits in a word</td></tr>
<tr><td>&&,</td><td rowspan="2">logical AND. Treat all variables as False (zero) or True (nonzero).
logical OR. (7||0) is (T||F) = 1, (2||-3) is (T||T) =1,
(3&&0) is (T&&F) = 0.</td></tr>
<tr><td>||</td></tr>
<tr><td>? :</td><td>conditional. x=(cond)? T : F;</td></tr>
</table>

8.4 Design Methodologies

There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology.

8.4.1 Top-down design methodology

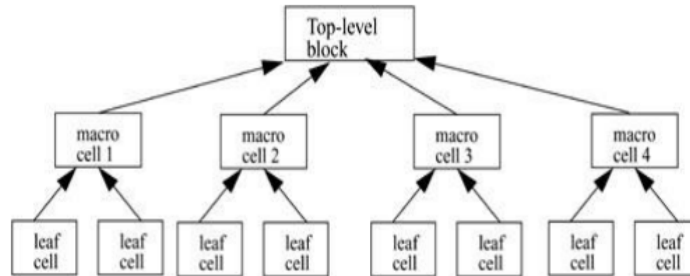
This designing approach allows early testing, easy change of different technologies, a well structures system design and offers many other advantages.



In this method, top-level block is defined and sub-blocks necessary to build the top-level block are identified. We further subdivide, sub-blocks until cells cannot be further divided, we call these cells as leaf cells.

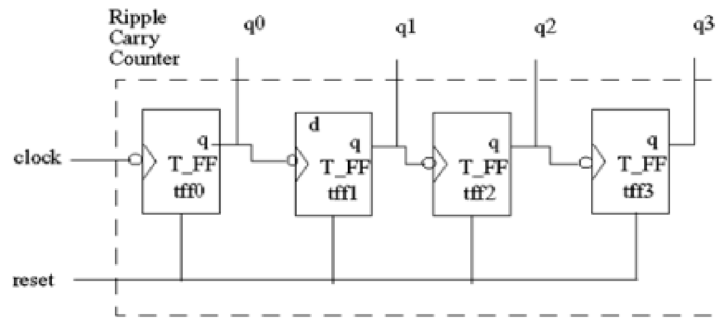
8.4.2 Bottom-up design methodology:

We first identify the available building blocks and try to build bigger cells out of these, and continue process until we reach the top-level block of the design. Most of the time, the combination of these two design methodologies are used to design. Logic designers decide the structure of design and break up the functionality into blocks and sub blocks. And designer will design a optimized circuit for leaf cell and using these will design top level design.



A hierarchical modeling concept is illustrated with an example of 4-bit Ripple Carry Counter. The ripple carry counter is made up of negative edge-triggered toggle flip-flops (T_FF). Each of the T_FFs can be made up from negative edge-triggered D-flipflops (D_FF) and inverters

6.



Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks.
In a top-down design methodology, we first have to specify the functionality of the ripple carry counter, which is the top-level block. Then, we implement the counter with T_FFs. We build the T_FFs from the D_FF and an additional inverter gate. Thus, we break bigger blocks into smaller building sub- blocks until we decide that we cannot break up the blocks any further. A bottom-up methodology flows in the opposite direction. We combine small building blocks and build bigger blocks; e.g., we could build D_FF from and/ or gates, or we could build a custom D_FF from transistors. Thus, the bottom-up flow meets the top-down flow at the level of the D_FF.

8.5 Modules

Verilog provides the concept of a module. A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality

that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design. In Verilog, a module is declared by the keyword module. A corresponding keyword endmodule must appear at the end of the module definition.

module ¡module_name¿ (¡module_terminal_list>);
...
<module internals>
...
... endmodule
Specifically, the T-flipflop could be defined as a module as follows:
module T_FF (q, clock, reset);
.
.
<functionality of T-flipflop>
.
.
endmodule

Verilog is both a behavioral and a structural language. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design. The levels are defined below.

- Behavioral or algorithmic level: This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.
- Dataflow level: At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.
- Gate level: The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gatelevel logic diagram.
- Switch level: This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details. Verilog allows the designer to mix and match all four levels of abstractions in a design.

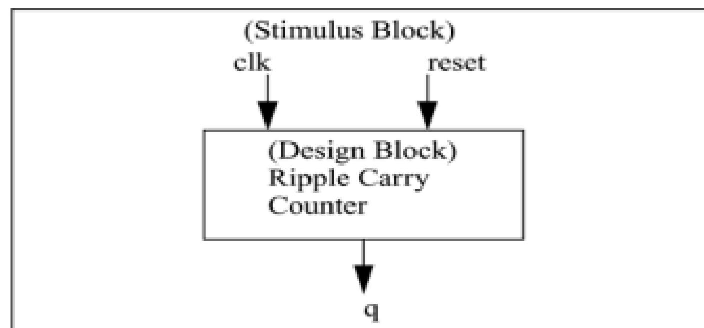
8.5.1 Module Instances

A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name,variables, parameters, and I/O interface. The process of creating objects from a module template is called instantiation, and the objects are called instances.

Example of Module Instantiation module ripple_carry_counter(q, clk, reset);
output [3:0] q;
input clk, reset;
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);
endmodule

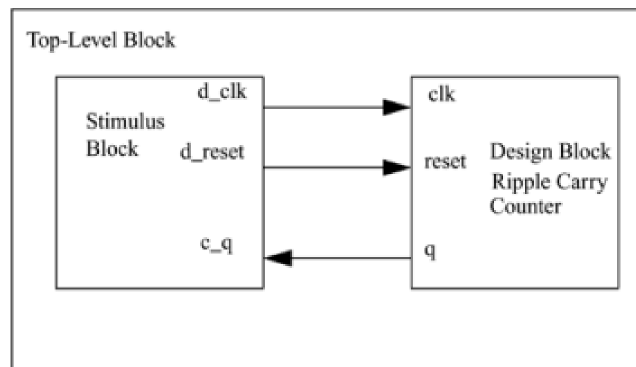
8.5.2 Components of a simulation

Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results. We call such a block the stimulus block. It is good practice to keep the stimulus and design blocks separate. The stimulus block can be written in Verilog. A separate language is not required to describe stimulus. The stimulus block is also commonly called a test bench. Different test benches can be used to thoroughly test the design block.
Two styles of stimulus application are possible. In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block.



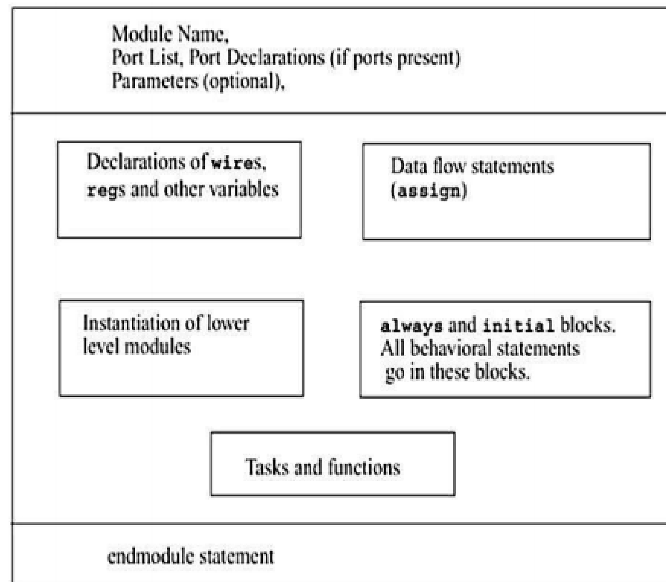
The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top- level dummy module. The stimulus block interacts with

the design block only through the interface. This style of applying stimulus is shown in thee below Figure. The stimulus module drives the signals d_clk and d_reset, which are connected to the signals clk and reset in the design block.It also checks and displays signal c_q, which is connected to the signal q in the design block. The function of top-level block is simply to instantiate the design and stimulus blocks. Either stimulus style can be used effectively.



8.6 Modules

Module is a basic building block in Verilog. A module definition always begins with the keyword module. The module name, port list, port declarations, and optional parameters must come first in a module definition. Port list and port declarations are present only if the module has any ports to interact with the external environment. The five components within a module are: variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions. These components can be in any order and at any place in the module definition. The endmodule statement must always come last in a module definition. All components except module, module name, and endmodule are optional and can be mixed and matched as per design needs. Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.




```
    // This example illustrates the different components of a module
// Module name and port list
// SR_latch module
module SR_latch(Q, Qbar, Sbar, Rbar);
//Port declarations output Q, Qbar; input Sbar, Rbar;
// Instantiate lower-level modules
// In this case, instantiate Verilog primitive nand gates
// Note how the wires are connected in a cross-coupled fashion.
nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);
// endmodule statement
endmodule
```

```
    // Module name and port list
// Stimulus module
module Top;
// Declarations of wire, reg, and other variables
wire q, qbar;
reg set, reset;
// Instantiate lower-level modules
// In this case, instantiate SR_latch Feed inverted set and reset signals to the
SR latch
SR_latch m1(q, qbar,  set,  reset);
// Behavioral block, initial
initial
```

```
begin
$ monitor($time, " set = %b, reset= %b, q= %b",set,reset,q);
set = 0; reset = 0;
#5 reset = 1;
#5 reset = 0;
#5 set = 1;
end
// endmodule statement
endmodule
```

From the above example following characteristics are noticed:

• In the SR latch definition above, all components described in Figure 2.2 need not be present in a module. We do not find variable declarations, dataflow (assign) statements, or behavioral blocks (always orinitial).

• However, the stimulus block for the SR latch contains module name, wire, reg, and variable declarations, instantiation of lower level modules, behavioral block (initial), and endmodule statement but does not contain port list, port declarations, and data flow (assign)statements.

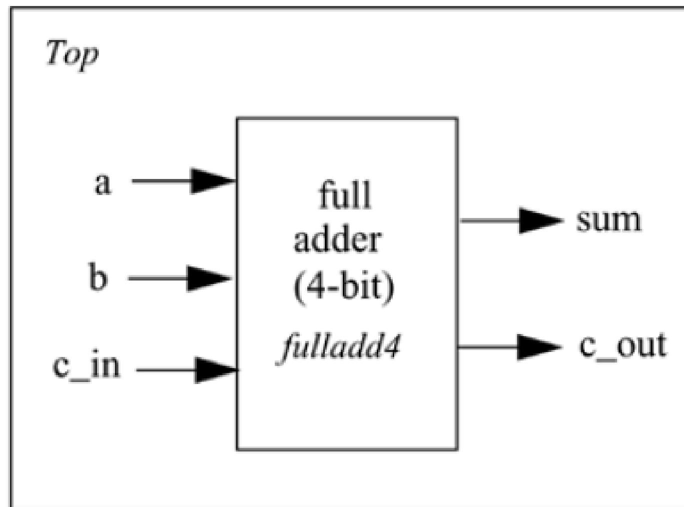
• Thus, all parts except module, module name, and endmodule are optional and can be mixed and matched as per design needs.

8.6.1 Port

Ports provide the interface by which a module can communicate with its environment. For example, the input/output pins of an IC chip are its ports. The environment can interact with the module only through its ports. The internals of the module are not visible to the environment. This provides a very powerful flexibility to the designer. The internals of the module can be changed without affecting the environment as long as the interface is not modified. Ports are also referred to as terminals.

8.6.2 List of ports

A module definition contains an optional list of ports. If the module does not exchange any signals with the environment, there are no ports in the list. Consider a 4-bit full adder that is instantiated inside a top- level module Top.



From the above figure, the module Top is a top-level module. The module fulladd4 is instantiated below Top. The module fulladd4 takes input on ports a, b, and c_in and produces an output on ports sum and c_out. Thus, module fulladd4 performs an addition for its environment. The module Top is a top-level module in the simulation and does not need to pass signals to or receive signals from the environment. Thus, it does not have a list of ports. The module names and port lists for both module declarations in Verilog are as shown in below example.

Example of List of Ports
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation

8.6.3 Port Declaration

All ports in the list of ports must be declared in the module. Ports can be declared as follows:

input - input port
output - output port
inout - Bidirectional port
Each port in the port list is defined as input, output, or inout, based on the direction of the port signal. Thus, for the example of the the port declarations will be as shown in example below.

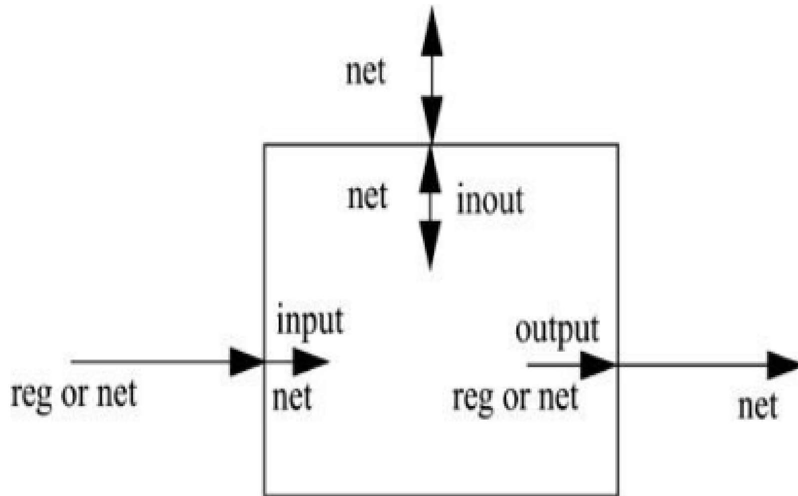
Example for Port Declarations
module fulladd4(sum, c_out, a, b, c_in);

```
//Begin port declarations section
output[3:0] sum;
output c_cout;
input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
... endmodule
```

All port declarations are implicitly declared as wire in Verilog. Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout. Input or inout ports are normally declared as wires. However, if output ports hold their value, they must be declared as reg. Ports of the type input and inout cannot be declared as reg because reg variables store values and input ports should not store values but simply reflect the changes in the external signals they are connected to.

8.6.4 Port connection rules

A port as consisting of two units, one unit that is internal to the module and another that is external to the module. The internal and external units are connected. There are rules governing port connections when modules are instantiated within other modules. The Verilog simulator complains if any port connection rules are violated.



Inputs
Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.

outputs
Internally, outputs ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.
inouts
Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to a net.

Width matching
It is legal to connect internal and external items of different sizes when making intermodule port connections. However, a warning is typically issued that the widths do not match.

unconnected ports
Verilog allows ports to remain unconnected. For example, certain output ports might be simply for debugging, and you might not be interested in connecting them to the external signals.

8.6.5 Connecting ports to an external signal

There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition. These two methods cannot be mixed.
These methods are
Connecting by ordered list

The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition. Consider the module fulladd4.To connect signals in module Top by ordered list, the Verilog code is shown in below example. Notice that the external signals SUM, C_OUT, A, B, and C_IN appear in exactly the same order as the ports sum, c_out, a, b, and c_in in module definition of fulladd4.

Example

```
module Top; //Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
//Instantiate fulladd4, call it fa_ordered.
//Signals are connected to ports in order (by position)
fulladd4 fa_ordered (SUM, C_OUT, A, B, C_IN);
...
<stimulus>
... endmodule
module fulladd4(sum, c_out, a, b, c_in);
output[3:0] sum;
output c_cout;
input [3:0] a, b;
input c_in;
...
<module internals>
...
endmodule
```

Connecting ports by name

For large designs where modules have, say, 50 ports, remembering the order of the ports in the module definition is impractical and error-prone. Verilog provides the capability to connect external signals to ports by the port names, rather than by position. We could connect the ports by name in above example by instantiating the module fulladd4, as follows. Note that you can specify the port connections in any order as long as the port name in the module definition correctly matches the external signal.

```
// Instantiate module fa_byname and connect signals to ports by name

fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

Note that only those ports that are to be connected to external signals must be specified in port connection by name. Unconnected ports can be dropped. For example, if the port c_out were to be kept unconnected, the instantiation

of fulladd4 would look as follows.
The port c_out is simply dropped from the port list. // Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);

Another advantage of connecting ports by name is that as long as the port name is not changed, the order of ports in the port list of a module can be rearranged without changing the port connections in module instantiations.

8.6.6 Hierarchical Names

Every module instance, signal, or variable is defined with an identifier. A particular identifier has a unique place in the design hierarchy. Hierarchical name referencing allows us to denote every identifier in the design hierarchy with a unique name. A hierarchical name is a list of identifiers separated by dots (".") for each level of hierarchy. Thus, any identifier can be addressed from any place in the design by simply specifying the complete hierarchical name of that identifier. The top-level module is called the root module because it is not instantiated anywhere. It is the starting point.
To assign a unique name to an identifier, start from the top-level module and trace the path along the design hierarchy to the desired identifier.

8.7 Dataflow Modeling

For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connects every gate individually. Also, gatelevel modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level. Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.

8.7.1 Continuous Assignments

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword assign. The syntax of an assign statement is as follows.

continuous_assign ::= assign [drive_strength] [delay3] list_of_net_assignments ;

list_of_net_assignments ::= net_assignment { , net_assignment }

net_assignment ::= net_lvalue = expression

The default value for drive strength is strong1 and strong0. The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates. Continuous assignments have the following characteristics:

- The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
- Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right- hand-side operands changes and the value is assigned to the left-hand-side net
- The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
- Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

8.7.2 Implicit Continuous Assignment

Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once.

Implicit Net Declaration
If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name. If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

// Continuous assign. out is a net.

wire i1, i2;
assign out = i1 i2; //Note that out was not declared as a wire
//but an implicit wire declaration for out

//is done by the simulator

Example 4-to-1 Multiplexer using logic equations

// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model

module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram output out;
input i0, i1, i2, i3;
input s1, s0;
//Logic equation for out
assign out = $(\sim s1\& \sim s0\$i0) \wedge (\sim s1\&s0\&i1)|(s1\& \sim s0\&i2) \wedge (s1\&s0\&i3)$;
endmodule

conditional operator

There is a more concise way to specify the 4-to-1 multiplexers. Example of 4-to-1 Multiplexer, Using Conditional Operators

// Module 4-to-1 multiplexer using data flow. Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram output out;
input i0, i1, i2, i3;
input s1, s0;
// Use nested conditional operator
assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule

In the simulation of the multiplexer, the gate-level module can be substituted with the dataflow multiplexer modules described above. The stimulus module will not change. The simulation results will be identical. By encapsulating functionality inside a module, we can replace the gate-level module with a dataflow module without affecting the other modules in the simulation. This is a very powerful feature of Verilog.

The Behavioral or procedural level

8.8 Behavioral Modelling

Verilog has four levels of modelling:

1. The switch level which includes MOS transistors modelled as switches.
2. The gate level.

3. The Data-Flow level.
4. The Behavioral or procedural level

Verilog procedural statements are used to model a design at a higher level of abstraction than the other levels. They provide powerful ways of doing complex designs. However small changes n coding methods can cause large changes in the hardware generated. Procedural statements can only be used in procedures.

8.8.1 Procedural Assignments

Procedural assignments are assignment statements used within Verilog procedures (always and initial blocks). Only reg variables and integers (and their bit/part-selects and concatenations) can be placed left of the "=" in procedures. The right hand side of the assignment is an expression which may use any of the operator types.

8.8.2 Delay in Assignment (not for synthesis)

In a delayed assignment t time units pass before the statement is executed and the left-hand assignment is made. With intra-assignment delay, the right side is evaluated immediately but there is a delay of t before the result is place in the left hand assignment. If another procedure changes a right-hand side signal during Δ t, it does not effect the output. Delays are not supported by synthesis tools.

Syntax for Procedural Assignment

variable = expression
Delayed assignment
#Δt variable = expression;
Intra-assignment delay
variable = #Δt expression;

8.8.3 Blocking Assignment

Procedural (blocking) assignments (=) are done sequentially in the order the statements are written. A second assignment is not started until the preceding one is complete.

Syntax

Blocking
variable = expression;
variable = #Δt expression;
grab inputs now, deliver ans.
later.
#Δt variable = expression;

grab inputs later, deliver ans.
later

8.8.4 Non-Blocking(RTL) Assignments

RTL (nonblocking) assignments (<=), which follow each other in the code, are started in parallel. The right hand side of nonblocking assignments is evaluated starting from the completion of the last blocking assignment or if none, the start of the procedure. The transfer to the left hand side is made according to the delays. An intra-assignment delay in a non-blocking statement will not delay the start of any subsequent statement blocking or non-blocking. However a normal delays will are cummulative and will delay the output.

For synthesis

The following example shows interactions between blocking and non-blocking for simulation. Do not mix the two types in one procedure for synthesis.
begin ... end
begin ... end block statements are used to group several statements for use where one statement is syntactically allowed. Such places include functions, always and initial blocks, if, case and for statements. Blocks can optionally be named.

• One must not mix "<=" or "=" in the same procedure.
• "<=" best mimics what physical flip-flops do; use it for "always @ (posedge clk ..) type procedures. • "=" best corresponds to what c/c++ code would do; use it for combinational procedures.
Syntax
Non-Blocking
variable <= expression;
variable <= #Δt expression;
#Δt variable ¡= expression;

8.8.5 begin ... end

begin ... end block statements are used to group several statements for use where one statement is syntactically allowed. Such places include functions, always and initial blocks, if, case and for statements.. Blocks can optionally be named.

Syntax
begin : block_name
reg [msb:lsb] reg_variable_list;
integer [msb:lsb] integer_list;

```
parameter [msb:lsb] parameter_list;
... statements ...
end
```

8.8.6 for Loops

Similar to for loops in C/C++, they are used to repeatedly execute a statement or block of statements. If the loop contains only one statement, the begin ... end statements may be omitted.

Syntax

```
for (count = value1;
count </<=/>/>= value2;
count = count +/- step)
begin
... statements ...
end
```

8.8.7 While Loops

The while loop repeatedly executes a statement or block of statements until the expression in the while statement evaluates to false. To avoid combinational feedback during synthesis, a while loop must be broken with an @(posedge/negedge clock) statement. For simulation a delay inside the loop will suffice. If the loop contains only one statement, the begin ... end statements may be omitted.

Syntax

```
while (expression)
begin
... statements ...
end
```

8.8.8 forever Loops

The forever statement executes an infinite loop of a statement or block of statements. To avoid combinational feedback during synthesis, a forever loop must be broken with an @(posedge/negedge clock) statement. For simulation a delay inside the loop will suffice. If the loop contains only one statement, the begin ... end statements may be omitted. It is

Syntax

```
forever
```

```
begin
... statements ...
end
```

8.9 Procedures: Always and Initial Blocks

8.9.1 Always Block

The always block is the primary construct in RTL modeling. Like the continuous assignment, it is a concurrent statement that is continuously executed during simulation. This also means that all always blocks in a module execute simultaneously. This is very unlike conventional programming languages, in which all statements execute sequentially. The always block can be used to imply latches, flip-flops or combinational logic. If the statements in the always block are enclosed within begin ... end, the statements are executed sequentially. If enclosed within the fork ... join, they are executed concurrently (simulation only).
The always block is triggered to execute by the level, positive edge or negative edge of one or more signals (separate signals by the keyword or). A double-edge trigger is implied if you include a signal in the event list of the always statement. The single edge-triggers are specified by posedge and negedge keywords.
Procedures can be named. In simulation one can disable named blocks. For synthesis it is mainly used as a comment.

Syntax 1

```
always @(event_1 or event_2 or ...)
begin
... statements ...
end
```

Syntax 2

```
always @(event_1 or event_2 or ...)
begin: name_for_block
... statements ...
end
```

8.9.2 Initial Block

The initial block is like the always block except that it is executed only once at the beginning of the simulation. It is typically used to initialize variables and specify signal waveforms during simulation. Initial blocks are not supported for synthesis.

Syntax

```
initial
```

```
begin
... statements ...
end
```

8.10 Functions

Functions are declared within a module, and can be called from continuous assignments, always blocks, or other functions. In a continuous assignment, they are evaluated when any of its declared inputs change. In a procedure, they are evaluated when invoked.
Functions describe combinational logic, and by do not generate latches. Thus an if without an else will simulate as though it had a latch but synthesize without one. This is a particularly bad case of synthesis not following the simulation. It is a good idea to code functions so they would not generate latches if the code were used in a procedure.
Functions are a good way to reuse procedural code, since modules cannot be invoked from within a procedure.

8.10.1 Function Declaration

A function declaration specifies the name of the function, the width of the function return value, the function input arguments, the variables (reg) used within the function, and the function local parameters and integers.

Syntax, Function Declaration

```
function [msb:lsb] function_name;
input [msb:lsb] input_arguments;
reg [msb:lsb] reg_variable_list;
parameter [msb:lsb] parameter_list;
integer [msb:lsb] integer_list;
... statements ...
endfunction
```

8.10.2 Function Return Values

When you declare a function, a variable is also implicitly declared with the same name as the function name, and with the width specified for the function name (The default width is 1-bit).

8.10.3 Function Call

A function call is an operand in an expression. A function call must specify in its terminal list all the input parameters.

8.10.4 Function Rules

The following are some of the general rules for functions:

- Functions must contain at least one input argument.
- Functions cannot contain an inout or output declaration.
- Functions cannot contain time controlled statements
- - Functions cannot enable tasks.
- Functions must contain a statement that assigns the return value to the implicit function name register.

8.11 Tasks

A task is similar to a function, but unlike a function it has both input and output ports. Therefore tasks do not return values. Tasks are similar to procedures in most programming languages. The syntax and statements allowed in tasks are those specified for functions.

Syntax

```
task task_name;
input [msb:lsb] input_port_list;
output [msb:lsb] output_port_list;
reg [msb:lsb] reg_variable_list;
parameter [msb:lsb] parameter_list;
integer [msb:lsb] integer_list;
... statements ...
endtask
```

8.12 System Task and Functions

These are tasks and functions that are used to generate input and output during simulation. Their names begin with a dollar sign ($). The Synopsys Verilog HDL Compiler/Design Compiler and many other synthesis tools parse and ignore system functions, and hence can be included even in synthesizable models.
System tasks that extract data, like $monitor need to be in an initial or always block.

8.12.1 Display Selected Variables; $display, $strobe, $monitor

These commands have the same syntax, and display text on the screen during simulation. They are much less convenient than waveform display tools. $display and $strobe display once every time they are executed, whereas $monitor displays every time one of its parameters changes. The difference between

$display and $strobe is that $strobe displays the parameters at the very end of the current simulation time unit rather than exactly where it is executed. The format string is like that in C/C++, and may contain format characters. Format characters include %d (decimal), %h (hexadecimal), %b (binary), %c (character), %s (string) and %t (time), %m (hierarchy level). %5d, %5b etc. would give exactly 5 spaces for the number instead of the space needed.

Syntax $display ("format_string", par_1, par_2, ...);
$strobe ("format_string", par_1, par_2, ...);
$monitor ("format_string", par_1, par_2, ...);
$displayb (as above but defaults to binary..
$strobeh (as above but defaults to hex..
$monitoro (as above but defaults to octal..

9 Advanced Topics in Verilog HDL and synthesis

There are threee delay models in verilog

1. Distributed delay model
2. Lumped Delay model
3. Pin-to-Pin(path) delay model

9.1 Distributed delay model

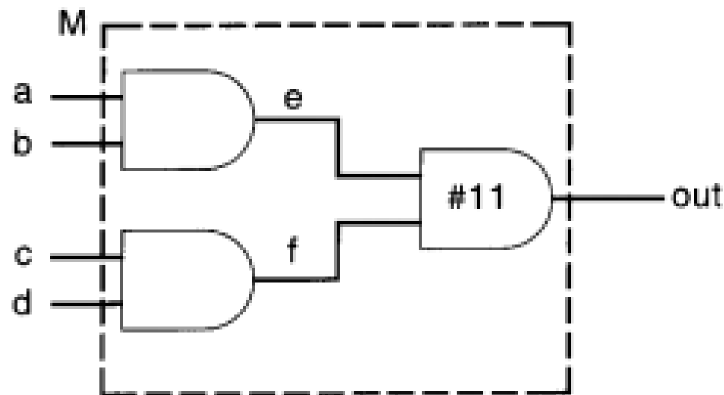
Distributed delays are specified on a per basis model. Delay values are assigned to individual element in the circuit. Distributed delays can be modeled by assigning delay values to individual gates or by using delay values in individual assign statements. When inputs of any gate changes, the output of the gate changes after the delay values specified.

```
//Distributed delays in gate-level modules
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Delay is distributed to each gate.
and #5 a1(e, a, b);
and #7 a2(f, c, d);
and #4 a3(out, e, f);
endmodule

//Distributed delays in data flow definition of a module
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Distributed delay in each expression
assign #5 e = a & b;
assign #7 f = c & d;
assign #4 out = e & f;
endmodule
```

Distributed delay provides detailed delay modelling. Delay in each element of the circuit are specified.

9.2 Lumped Delays

Lumped delays are specified on a per basis model. They can be specified as a single delay on the output gate of the module. The cumulative delay of all paths is lumped at one location.



In this example we computed the maximum delay from any input to the output. The entire delay is lumped into the output gate. After a delay, primary output changes any input to the module M changes.

```
Module M(out, a , b, c,d);
output out;
input a,b,c,d;
wire e,f;

    and a1(e,a,b);
and a2(f,c,d);
and #11 a3(out,e,f);
endmodule
```

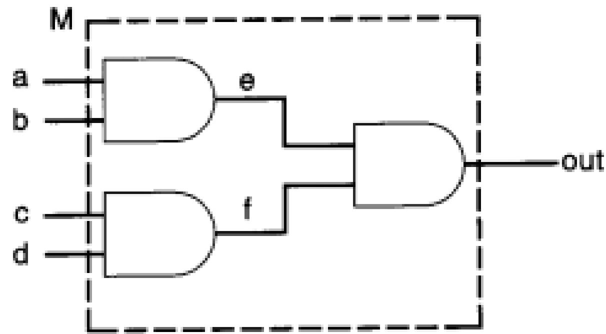
9.3 Pin-to-Pin delays

Another method of delay specification for a module is pin-to-pin timing. Delays are assigned individually to paths from each input to each output. Thus, delays can be separately specified for each input/output path. Thus, delays can be separately specified for each input/output path.

Pin-to-pin delays for standard parts can be directly obtained from data books. This delay for modules of a digital circuits are obtained by circuit characterization, using a low-level simulator like SPICE.

Although pin-to-pin delays are very detailed, for large circuits they are easier to model than distributed delays because the designer writing delay models needs to know only the I/O pins of the module rather than the internals of the module. The internals of the module may be designed by using gates, data flow,

behavioral statements, or mixed design, but the pin-to-pin delay specification remains the same. These are also known as path delays.



9.3.1 Path delays

9.3.2 Specify blocks

A delay between a source pin and a destination pin of a module is called a module path delay. Path delays are assigned in verilog within the keywords specify and endspecify. The statements within these keywords consititute a specify block.
Specify block contains the statements to do the following:

- Assign pin-to-pin timing delays across module paths
- Set up timing checks in the circuits
- Define specparam constants

Example

```
module M(out,a,b,c,d);
output out;
input a,b,c,d;
wire e,f;
specify
(a => out) = 9;
(b => out) = 9;
(c => out) = 11;
(d => out) = 11;
endspecify
```

```
//gate instantiations
and a1(e,a,b);
and a2(f,c,d);
and a3(out,e,f);
endmodule
```

The specify block is a separate block in the module and does not appear under any other block, such as initial or always. The meaning of the statements within specify blocks needs to be clarified.

9.3.3 Inside Specify Blocks

Parallel connection
Every path delay statement has a source field and a destination field. In the path delay, a,b,c,d are in the position of the source field and out is the destination field.
A parallel connection is specified by the symbol => and is used as shown:

(<source_field> => <destination_fields>) = <delay_value>;
In a parallel connection, each bit in the source field connects to its corresponding bit in the destination field. If the source and destination fields are vectors, they must have the same number of bits; otherwise, there is a mismatch. Thus a parallel connection specified delays from each bit in source to each bit in destination.

Full connection
A full connection is specifed by the symbol *> and is used as shown:
(<source_field> *> <destination_fields>) = <delay_value>;
In this, each bit in the source field connects to every bit in the destination field. If the source and the destination are vectors, then they need not have the same number of bits. A full connection describes the delay between each bit of the source and every bit in the destination.
A full connection is particularly useful for specifying a delay between each bit of an input vector and every bit in the output vector when bit width of the vectors is large.

Edge sensitive paths

An edge sensitive path construct is used to model the timing of input to output delays,which occurs only when a specified edge occurs at the source signal.

specparam statements
Special parameters can be declared for use inside a specify block. They are declared by the keyword **specparam**. Instead of using hardcoded delay numbers to specify pin-to-pin delays, it si common to define specify parameters by using specparam and then to use those parameters inside the specify block. The val-

ues of this are often used to store values for nonsimulation tools, such as delay calculators, synthesis tools, and layout estimators.

Conditional path delays

Based on the states of input signals to a circuit, the pin-to-pin delays might change. Verilog allows path delays to be assigned conditionally, based on the value of the signals in the circuit. A conditional path delay is expressed with the if conditional statement. The conditional expression can contain any logical,bitwise, reduction, concatenation or conditional operator.

Rise,fall and turn-off delays

Pin-to-pin timing can also be expressed in more detail by specifying rise, fall, and turn-off delay values. One,two, three,six, or twelve delay values can be specified for any path. Four, five, seven, eight, nine, ten, or eleven delay value specification is illegal. The order in which delays are specified must be strictly followed.

Example

```
//Specify one delay only. Used for all transitions.
specparam t_delay = 11;
(clk =< q) = t_delay;

//specify two delays, rise and fall
//Rise used for transitions 0 -<1, 0->z, z-> 1
//Fall used for transitions 1 ->0,1->z, z->0
specparam t_rise = 9, t_fall = 13;
(clk => q) = (t_rise, t_fall);

//Rise used for transitions 0->1, z->1
//Fall used for transitions 1->0,z->0
//Turn-off used for transitions 0->z, 1->z
specparam t_rise = 9; t_fall = 13, t_turnoff = 11;
(clk =>q) = (t_rise, t_fall, t_turnoff);
//Specify three delays, rise, fall and turn-off
```

Min,Max and typical delays

Min,max and typical values can also be specified for pin-to-pin delays.

Example

```
//Specify three delays, rise, fall and turn-off
//Each delay has a min:type:max value
specparam t_rise = 8:9:0, t_fall = 12:13:14, t_turnoff = 10:11:12;
(clk => q) = (t_rise, t_fall, t_turnoff);
```

9.4 Switch-level Modeling

Verilog provides the ability to design at MOS-transistor level, however with increase in complexity of the circuits design at this level is growing tough. Verilog however only provides digital design capability and drive strengths associated to them. Analog capability is not into picture still. As a matter of fact transistors are only used as switches.
//MOS switch keywords
nmos
pmos
Whereas the keyword nmos is used to model a NMOS transistor, pmos is used for PMOS transistors.
Instantiation of NMOS and PMOS switches
nmos n1(out, data, control); // instantiate a NMOS switch
pmos p1(out, data, control); // instantiate a PMOS switch

9.4.1 CMOS switches

Instantiation of a CMOS switch.

cmos c1(out, data, ncontrol, pcontrol); // instantiate a cmos switch
The 'ncontrol' and 'pcontrol' signals are normally complements of each other

9.4.2 Bidirectional switches

These switches allow signal flow in both directions and are defined by keywords tran,tranif0 ,and tranif1

Instantiation
tran t1(inout1, inout2); // instance name t1 is optional
tranif0(inout1, inout2, control); // instance name is not specified
tranif1(inout1, inout2, control); // instance name t1 is not specified

9.4.3 Delay specification of switches

pmos, nmos, rpmos, rnmos
- Zero(no delay) pmos p1(out,data, control);
- One (same delay in all) pmos(1) p1(out,data, control);c
- Two(rise, fall) nmos(1,2) n1(out,data, control);
- Three(rise, fall, turnoff)mos(1,3,2) n1(out,data,control);

9.4.4 An Instance: Verilog code for a NOR- gate

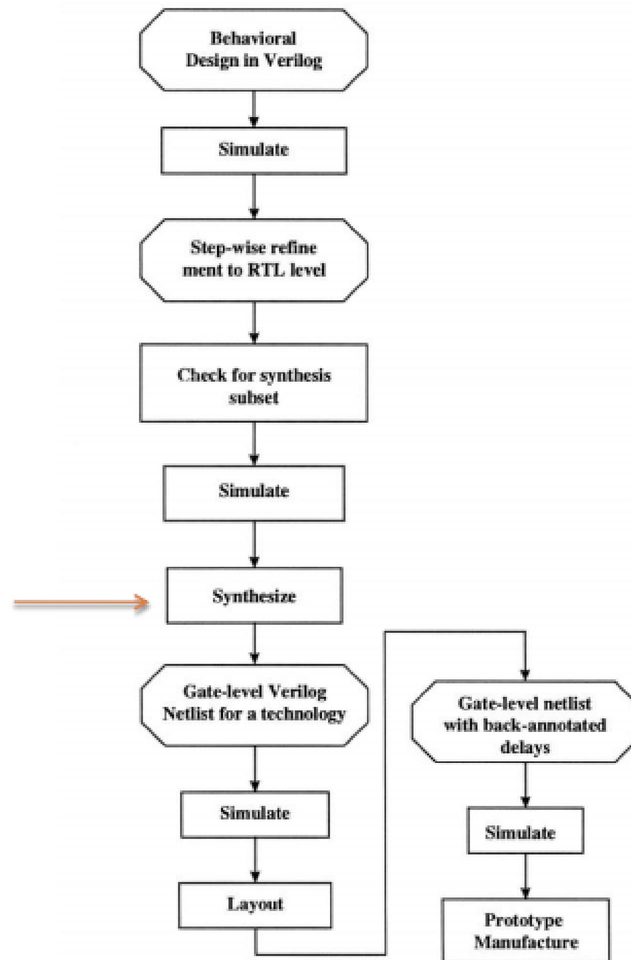
```
// define a nor gate, my_nor
module my_nor(out, a, b);
output out;
input a, b;
//internal wires
wire c;
// set up pwr n ground lines
supply1 pwr;// power is connected to Vdd
supply0 gnd; // connected to Vss
// instantiate pmos switches
pmos (c, pwr, b);
pmos (out, c, a);
//instantiate nmos switches
nmos (out, gnd, a);
```

Stimulus to test the NOR-gate

```
// stimulus to test the gate
module stimulus;
reg A, B;
wire OUT;
//instantiate the my_nor module
my_nor n1(OUT, A, B);
//Apply stimulus
initial
begin
//test all possible combinations
A=1'b0; B=1'b0;
5 A=1'b0; B=1'b1;
5 A=1'b1; B=1'b0;
5 A=1'b1; B=1'b1;
end
//check results
initial
$ monitor($time, "OUT = %b, B=%b, OUT, A, B);
endmodule
```

9.5 Verilog Synthesis flow

A top-down design starts with a behavioral description and is finally sent to the fab after complete placement, layout and final verification as shown in this diagram.



Design Flow

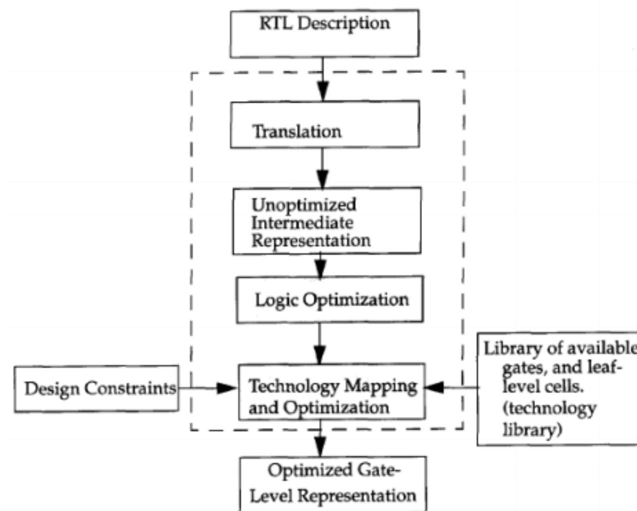
1. Write a high-level behavioral description of the planned design. This step starts with concepts and ends up with a high level description in the Verilog language. This description can have various levels of detail and essentially has architectural elements and algorithmic elements.
2. Next we perform stepwise refinement to the RTL level. This is again simulated and verified for functional correctness.
3. Synthesize the HDL description with the synthesizer. Detail synthesis flow is explained in later part of this ppt.

4. The output of a synthesizer is a gate-level Verilog description. Compare the output of the gate-level simulation (step 3) against the output of the original Verilog description.

5. After this the layout of the design is prepared followed by post-layout verification.

Synthesis flow: RTL to gates

To fully utilize the benefits of logic synthesis, the designer must first understand the flow from the high-level RTL description to a gate-level netlist.



Stages in Synthesis flow

- RTL description : The designer describes the design at a high level by using RTL constructs.
- Translation : The RTL description is converted by the logic synthesis tool to an unoptimized, intermediate, internal representation.
- Logic optimization : The logic is now optimized to remove redundant logic. Various technology independent boolean logic optimization techniques are used.
- Technology mapping and optimization : In this step, the synthesis tool takes the internal representation and implements the representation in gates, using the cells provided in the technology library.

- Technology library : The technology library contains library cells provided by ABC Inc. The term standard cell library and the term technology library are identical and are used interchangeably.

- Design constraints : Design constraints typically include the following:
  Timing-The circuit must meet certain timing requirements.
  An internal static timing analyzer checks timing.
  Area-The area of the final layout must not exceed a limit.
  Power-The power dissipation in the circuit must not exceed a threshold

- Optimized gate-level description :
  After the technology mapping is complete, an optimized gate-level netlist described in terms of target technology components is produced.
  The designer modifies the RTL or reconstrains the design to achieve the desired results. This process is iterated until the netlist meets the required constraints.

10 Post MCQ

1. Which of the following HDLs are IEEE standards?
a) VHDL and Verilog
b) C and C++
c) Altera and xilinx
d) Quartus II and MaxPlus II

2. An HDL can't describe Hardware at Gate level as well as switch level?
a) True
b) False

3. VHDL is based on which of the following programming languages?
a) ADA programming language
b) C
c) Assembly
d) PHP

4. Which of the following is a characteristic of VHDL?
a)Case sensitive
b) Use of simple data types
c) Based on C programming language
d) Strongly typed language

5. Which of the following can be the name of an entity?
a) NAND
b) NAND_gate
c) NAND gate

d) AND

6. Which of the following mode of the signal is bidirectional?
a) IN
b) OUT
c) INOUT
d) Buffer

7. Which of the following is the default mode for a port variable?
a) IN
b) OUT
c) INOUT
d) Buffer

8. The statements in between the keyword BEGIN and END are called
a) Concurrent statements
b) Netlist
c) Declaration statement
d) Entity function

9. Which of the following can't be declared in the declaration part of the architecture?
a) Signals
b) subprograms
c) components
d) Library

10. What is the difference between SIGNAL and VARIABLE?
a) The value of SIGNAL never varies whereas VARIABLE can change its value
b) SIGNAL can be used for input or output whereas VARIABLE acts as intermediate signals
c) SIGNAL depends upon VARIABLE for various operations
d) SIGNAL is global and VARIABLE is local to the process in which it is declared

11. The most basic form of behavioral modeling in VHDL is
a) IF statements
b) Assignment statements
c) Loop statements
d) Wait statements

12. The main problem with behavioral modeling is
a) Asynchronous delays
b) Simulation
c) No delay
d) Supports single driver only

13. The inertia value in inertial delay model is equal to
a) Initial value
b) Delay
c) Input value at a specific time
d) Output value at a specific time

14. Transport delay is a kind of
a) Synthesis delay
b) Simulation delay
c) Inertial delay
d) Wire delay

15. The keyword TRANSPORT in any assignment statement specifies
a) Transport delay
b) Transfer the right operand immediately to left operand
c) Transporting the value of left operand to right operand
d) Inertial delay

16. What is the basic unit of structural modeling?
a) Process
b) Component declaration
c) Component instantiation
d) Block

17. What do you mean by component instantiation?
a) To use the component
b) To describe external interface of the component
c) To declare the gate level components
d) To remove any component from the design

18. Which of the following function is used to map the component?
a) COMPONENT INSTANTIATE
b) PORT MAP
c) GENERIC MAP
d) Use

19. Which of the following is not a type of VHDL modeling?
a) Behavioral modeling
b) Dataflow modeling
c) Structural modeling
d) Component modeling

20. What is the basic unit of behavioral description?
a) Structure
b) Sequence

c) Process
d) Dataflow

21. The signal assignment is considered as a
a) Concurrent statement
b) Sequential statement
c) Subprogram
d) Package declaration statement

22. The concurrent assignment statement is activated whenever
a) The execution is scheduled
b) The value of the target is needed
c) The waveform associated changes its value
d) The process is terminated

23. If there is more than one process in a VHDL code, How they are executed?
a) One after the other
b) Concurrently
c) According to sensitivity list
d) Sequentially

24. Sensitivity list of a process contains
a) Constants
b) variables
c) signals
d) Literals

25. Which of the following is not a in-built package in VHDL?
a) STD_LOGIC_1164
b) TEXTIO
c) STANDARD
d) STD

26. Which of the following sequential statement can't be used in a function?
a) WAIT
b) IF
c) CASE
d) LOOP

27. How many return arguments can be there in the function?
a) 1
b) 2
c) 3
d) 4

28. Netlist is a ____ representation of a circuit.
a) graphical
b) text-based
c) handwritten
d) pictorial

29. RTL stands for
a) resistor-transfer logic
b) register-transistor logic
c) register-transfer logic
d) none of these

30. Verilog HDL originated at
a) AT&T Bell laboratories
b) Defense Advanced Research Projects Agency(DARPA)
c) Gateway Design Automation
d) Institute of Electrical and Electronics Engineers(IEEE)

31. Verilog may be written at the Behavioral, Structural, Gate, Switch, and Transistor levels.
a) True
b) False

32. Which of the following is true about parameters?
a) The default size of a parameter in most synthesizers is the size of an integer, 32 bits
b) Parameters enable Verilog code to be compatible with VHDL.
c) Parameters cannot accept a default value
d) None of the above

33. Which of the following is true about the always block?
a) There can be exactly one always block in a design.
b) There can be exactly one always block in a module
c) Execution of an always block occurs exactly once per simulation run.
d) An always block may be used to generate a periodic signal.

34. For describing circuits like flip flops ____ statement is used
a) Always
b) Entity
c) component
d) Initial

35. In VHDL, Bus is a type of
a) Signal
b) Constant
c) variable

d) None of the above

36. Predefined data for a VHDL object is called
a)Generic
b) Constants
c) Attribute
d) Variables

37. Which of the following statement is used in structural modeling?
a) Portmap()
b) Process ()
c) if-else
d) for-loop

38. A function call can be a concurrent as well as a sequential statement.
a) True
b) False

39. The sequential assignment statement is activated, whenever
a) The waveform associated changes its value
b) The process is terminated
c) The execution is scheduled
d) All of the above

40. Sequential statements are synthesizable
a) True
b) False

41. Which logic level is not supported by verilog?
a) U
b) X
c) Z
d) 0

42. If a net has no driver, it gets the value
a) 0
b) X
c) Z
d) 1

43. The process of creating objects from a module template is called
a) Instantiation
b) Declaration
c) Instances
d) None of these

44. _____ statements contain expressions, operators and operands.
a) Continuous Assignment Statement
b) initial
c) always
d) Assignment statements

45. The _____ operator is used to specify blocking assignments.
a) <=
b) ::=
c) =
d) $

46. The conditional statements are used to decide whether or not a statement should be ______
a) evaluated
b) declared
c) simplified
d) executed

47. Statements in a parallel block are executed ____
a) Sequentially
b) concurrently
c) with delay
d) without delay

48. Distributed delays can be modeled by assigning delay values to
a) separate gates
b) group gates
c) individual gates
d) individual assign statements

49. A standard format called the _____ is popularly used for back-annotation
a) standard delay format (SDF)
b) SDF format manual
c) Delay calculator format
d) None of these

50. Distributed delays are _____ than lumped delays but difficult to model for large designs
a) less accurate
b) accurate
c) more accurate
d) not much accurate

51. _____ process is used repeatedly to obtain a final circuit that meets all timing requirements.

a) resimulate
b) Delay-back annotation
c) full connection
d) Distributed delays

52. Path delays are also known as
a) turn-off delays
b) Rise delay
c) fall delay
d) state dependent path delays(SDPS)

53. Value of the _____ signal is determined from the values of data and control signals.
a) out
b) in
c) inout
d) buffer

54. The tran switch acts as a ____ between the two signals.
a) Bidirectional switch
b) buffer
c) control signals
d) power signals

55. Resistive devices have a ______ source-to-drain impedance
a) low
b) normal
c) high
d) moderate

56. _____ modeling is at a very low level of design abstraction.
a) dataflow
b) behavioral
c) structural
d) switch-level

57. If a _____ was found in the gate level design this would require redesign of thousands of gates.
a) bug
b) time
c) delay
d) human error

58. _____ tools convert the RTL description into optimized netlist.
a) simulation
b) synthesis

c) testing
d) logic synthesis

59. ______ typically include the timing, area and power.
a) logic synthesis
b) design constraints
c) fabrication
d) modeling techniques

60. ______ is an important technique used to break the design into smaller blocks.
a) function verification
b) verilog coding techniques
c) design partitioning
d) logic synthesis.

61. _____ are used for signal values that are driven on a net.
a) Driving strength
b) logic values
c) strength values
d) multiple signals

62. A collection of functionality, area, timing information and power information of the cell are called the
a) design constraints
b) cell characterization
c) technology library
d) synthesis tool

63. Delays are assigned individually to paths from each input to each output. This is called
a) Delta delay
b) lumped delay
c) pin-to-pin delay
d) None of the above

64. Verilog allows path delays to be assign ____, based on the value of the signals in the circuit.
a) sequentially
b) individually
c) conditionally
d) definetely

65. The _____ is the minimum time the data cannot change after the active clock edge.
a) hold time

b) setup time
c) width time
d) clock time

66. How can you instantiate a NMOS switch.
a) nmos n1;
b) nmos n1(out,data,control);
c) nmos n1(data, control);
d) nmos n1(out,data);

11 Assignments

1. Write a Verilog code for an AOI gate module using wire
2. Write a VHDL code for ripple counter using behavioral modeling.
3. Write a Verilog code for RAM using behavioral modeling.
4. Write a VHDL code for 4 bit adder using behavioral modeling.
5. Write a verilog code for a shift register using behavioral modeling
6. Write a VHDL code for XOR gate using structural modeling.
7. Write a Veilog code for a very simple processor using behavioral modeling.
8. Write a VHDL code for ROM using behavioral modeling.
9. Write a Verilog code for Asynchronous counter using behavioral modeling.
10. Write a VHDL code for up-down counter using behavioral modeling.

12 Conclusion

1. Software programming languages and HDL are very different language.

2. Because both have many basic level differences even if they seem to have few similarities

3. In contrast to most software programming languages like C, HDLs includes explicit notion of time which is the primary attribute of hardware.

4. Unlike the traditional digital system development approaches, this technique offers several advantages to the VLSI market segments particularly quick time to market, ASIC migration, flexibility, vendor independent compilation and synthesis among others.

5. This is the present and future of digital revolution for hardware modeling of complex digital systems

13 References

1. A VHDL primer by Bhaskar Jayaram, Third edition.

2. Verilog HDL, A Guide to digital design and synthesis, by samir Palnitkar, Pearson education.

3. https://www.tutorialspoint.com/vlsi_design/vlsi_design_verilog_introduction.htm

4. http://cva.stanford.edu/people/davidbbs/classes/ee108a/winter0607

5. https://www.cs.upc.edu/ jordicf/Teaching/secretsofhardware/VerilogIntroduction_Nyasulu.pdf

6. http://athena.ecs.csus.edu/ arad/csc273/intro_verilog_hdl.pdf

7. https://course.ccs.neu.edu/cs3650/ssl/TEXT-CD/Content/Tutorials/VHDL/vhdl-tutorial.pdf

8. https://www.sanfoundry.com/1000-vhdl-questions-answers/

9. https://nptel.ac.in/content/storage2/courses/106105159/Week

10. https://www.eng.auburn.edu/ nelsovp/courses/elec4200/Slides/VHDL

11. http://www.csun.edu/edaasic/roosta/lecture

12. https://learning.oreilly.com/library/view/vhdl/9788131732113/xhtml/chapter005.xhtml