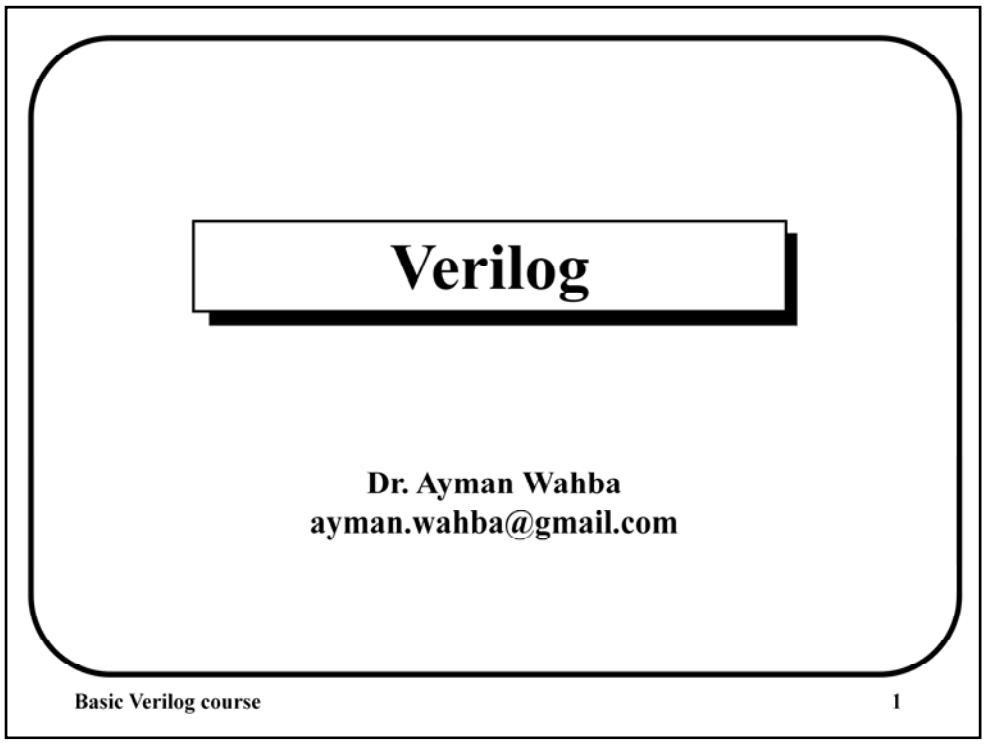


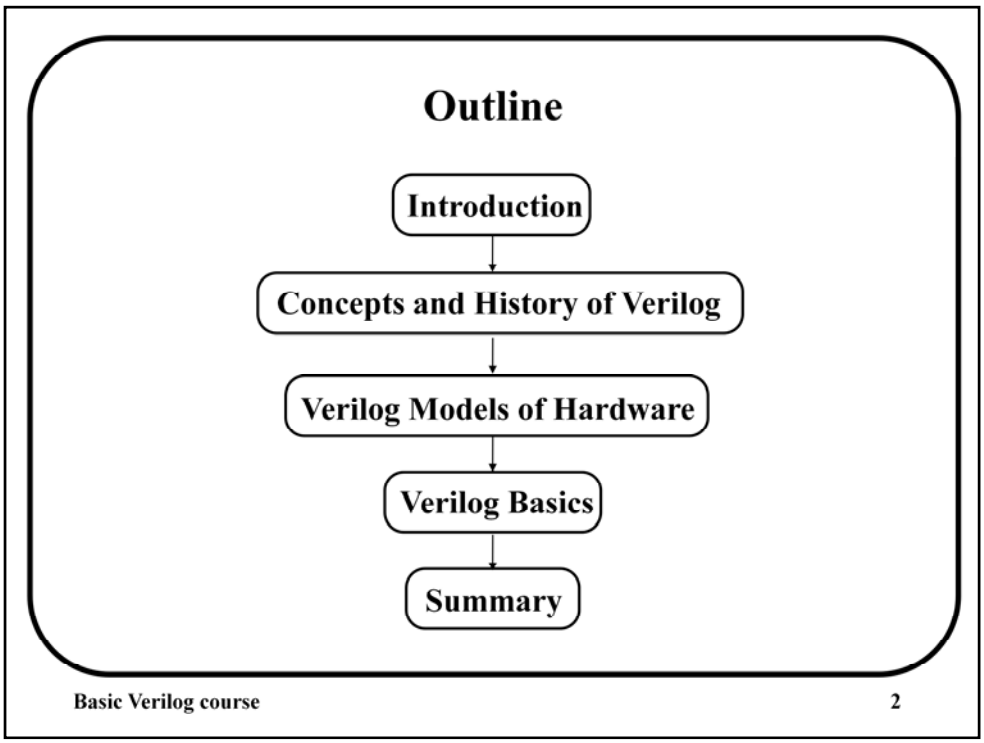
Verilog HDL is one of the two most common Hardware Description Languages (HDL) used by integrated circuit (IC) designers. The other one is VHDL.

HDL's allows the design to be simulated earlier in the design cycle in order to correct errors or experiment with different architectures. Designs described in HDL are *technology-independent*, easy to design and debug, and are usually more readable than schematics, particularly for large circuits.

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

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

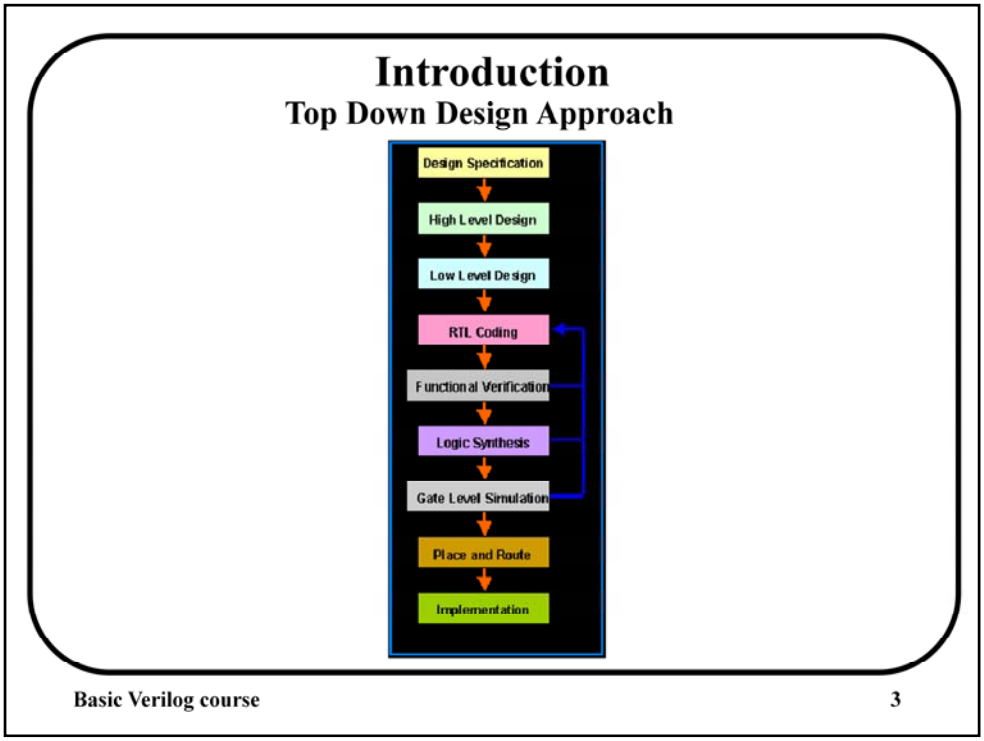
Verilog is used as an input for synthesis programs which will generate a gate-level description (a netlist) for the circuit. Some Verilog constructs are not synthesizable. Also the way the code is written will greatly effect the size and speed of the synthesized circuit.



Digital systems are highly complex. At their most detailed level, they may consist of millions of elements, as would be the case if we viewed a system as a collection of logic gates or transistors. From a more abstract viewpoint, these elements may be grouped into a handful of functional components such as *cache memories*, *floating point units*, *signal processors*, or *real-time controllers*. Hardware description languages have evolved to aid in the design of systems with this large number of elements and wide range of electronic and logical abstractions.

The Verilog language provides the digital system designer with a means of describing a digital system at a wide range of levels of abstraction, and, at the same time, provides access to computer-aided design tools to aid in the design process at these levels.

The language supports the early conceptual stages of design with its behavioral constructs, and the later implementation stages with its structural constructs. During the design process, behavioral and structural constructs may be mixed as the logical structure of portions of the design are designed. The description may be simulated to determine correctness, and some synthesis tools exist for automatic design. Indeed, the Verilog language provides the designer entry into the world of large, complex digital systems design.



The desired design-style of all designers is the top-down design. A real top-down design allows early testing, easy change of different technologies, a structured system design, and offers many other advantages. But it is very difficult to follow a pure top-down design. Due to this fact most designs are mix of both the methods, implementing some key elements of both design styles.

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

1. Behavioral level
2. Register-Transfer Level
3. Gate Level

1. Behavioral level:

This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. *Functions*, *Tasks* and *Always* blocks are the main elements.

There is no regard to the structural realization of the design.

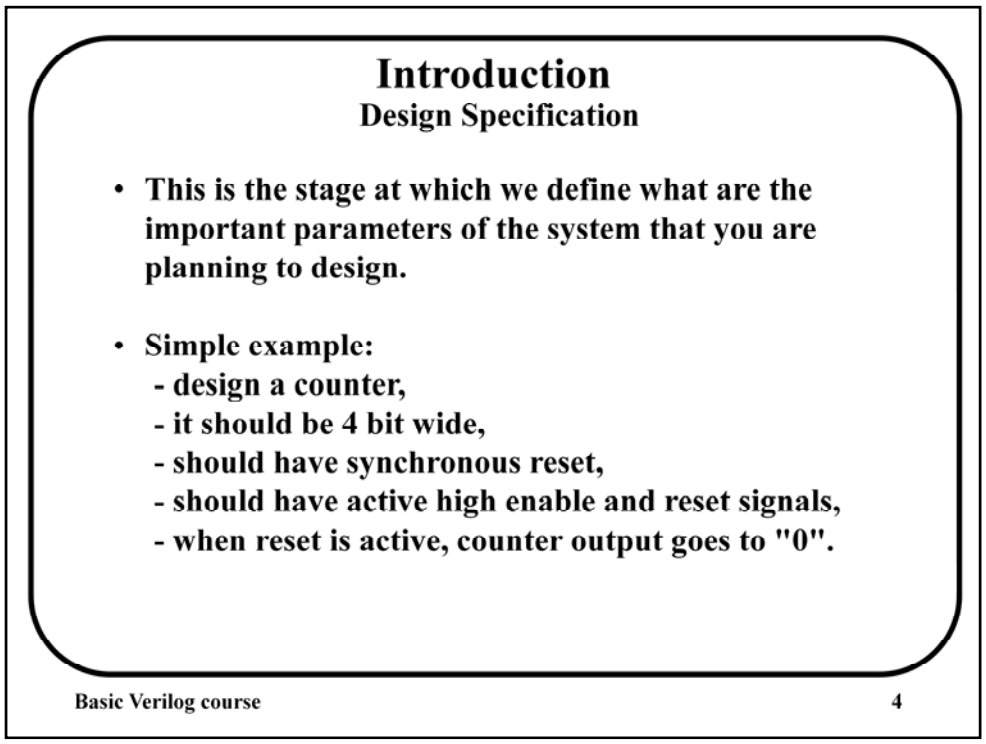
2. Register Transfer Level:

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

3. Gate Level:

Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates).

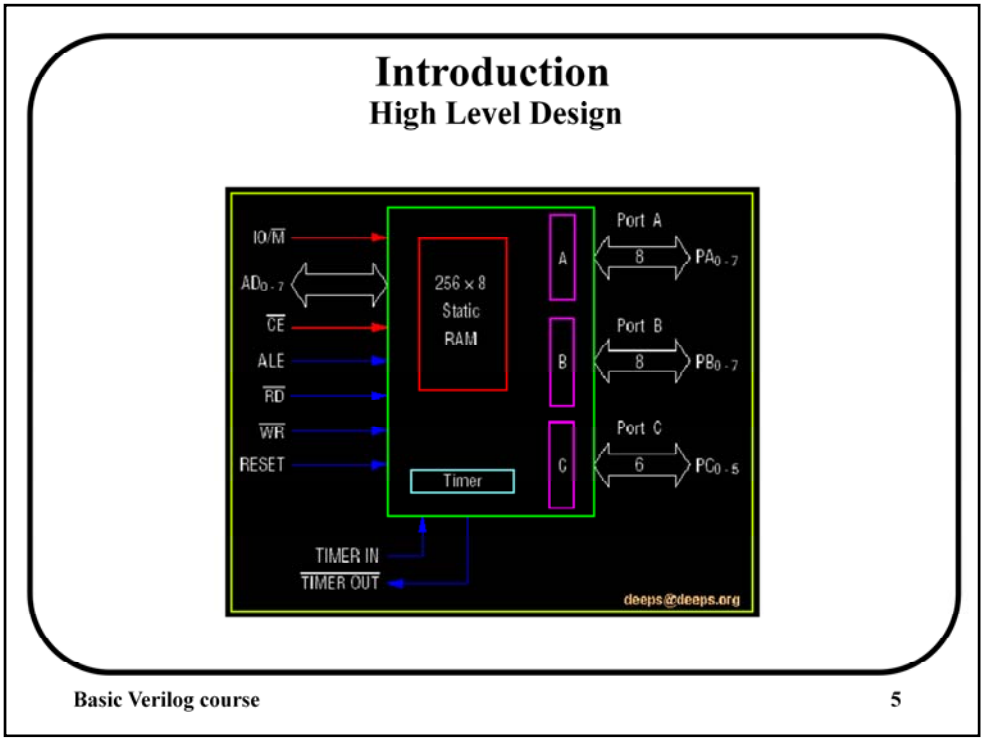
Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.



This is the stage at which we define what are the important parameters of the system that you are planning to design.

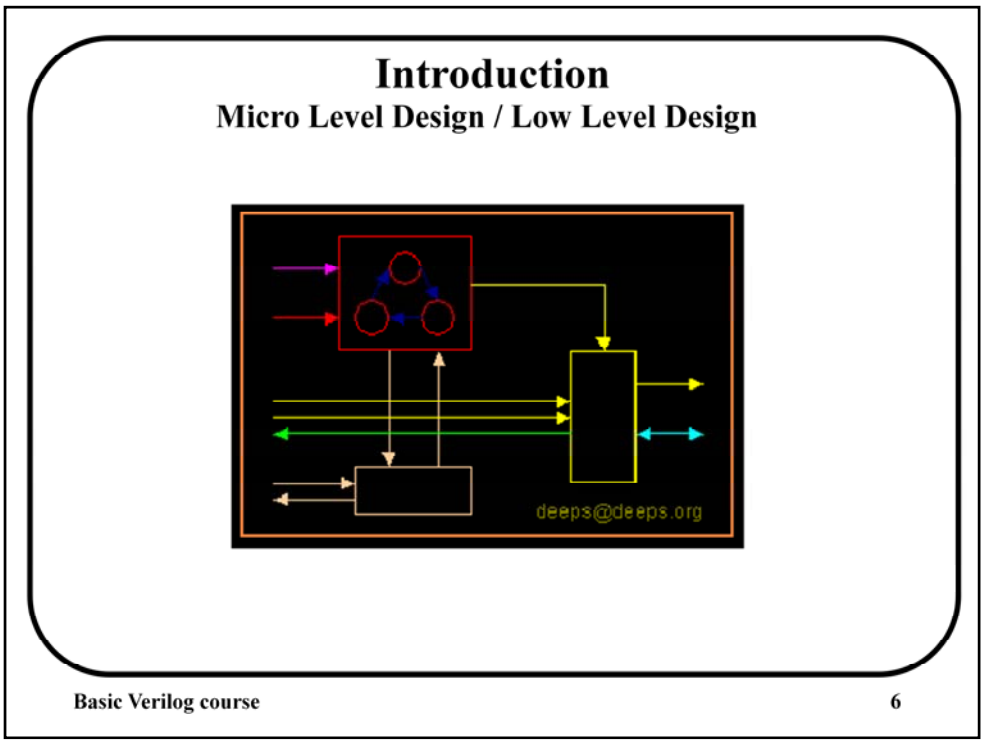
Simple example:

- design a counter,
- it should be 4 bit wide,
- should have synchronous reset,
- should have active high enable and reset signals,
- when reset is active, counter output goes to "0".



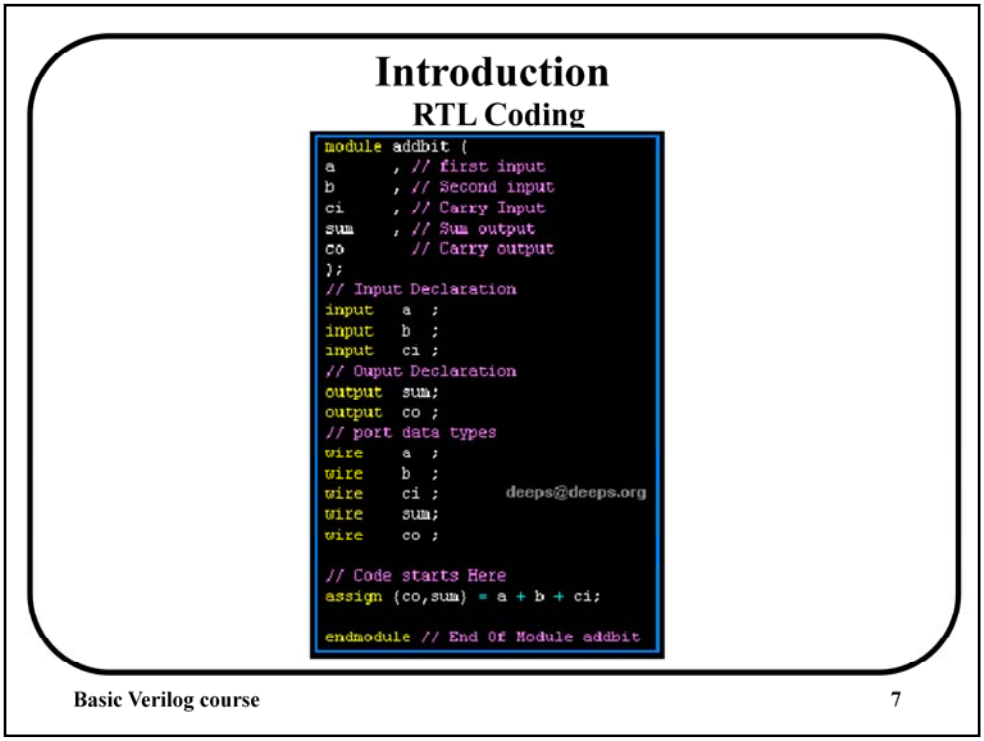
This is the stage at which you define various blocks in the design and how they communicate. Lets assume that we need to design microprocessor, High level design means splitting the design into blocks based on their function, In our case various blocks are:

- Registers,
- ALU,
- Instruction Decoders,
- Memory Interface, etc.

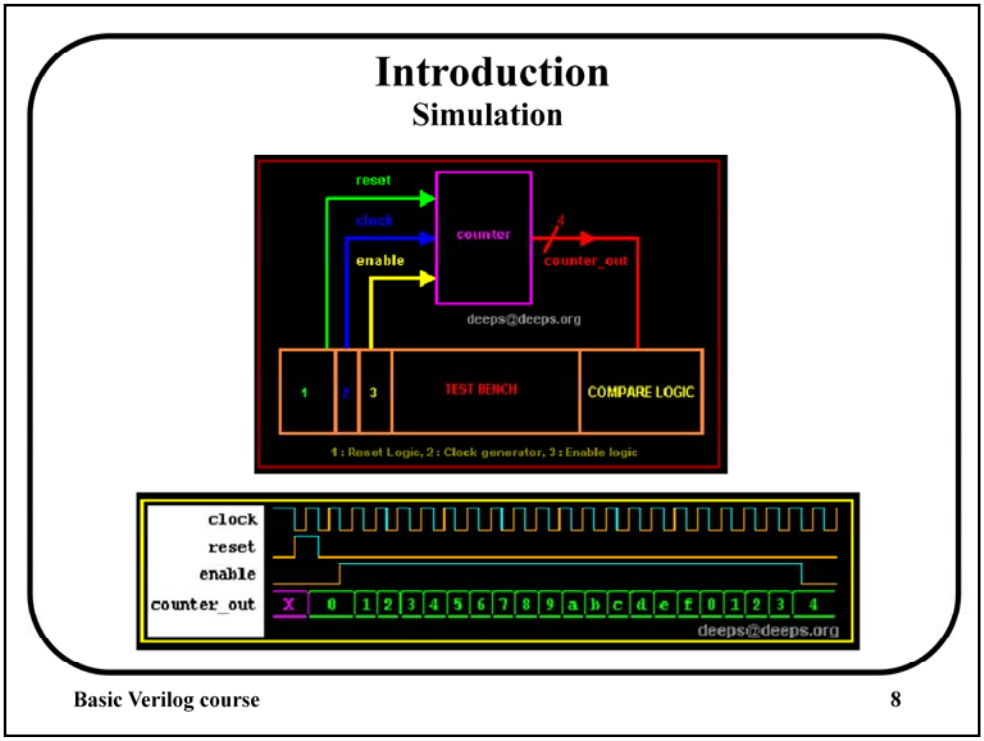


Low level design or Micro design is the phase in which, designer describes how each block is implemented.

It contains details of State machines, counters, Mux, decoders, internal registers.



In RTL coding, Micro Design is converted into Verilog/VHDL code, using synthesizable constructs of the language.

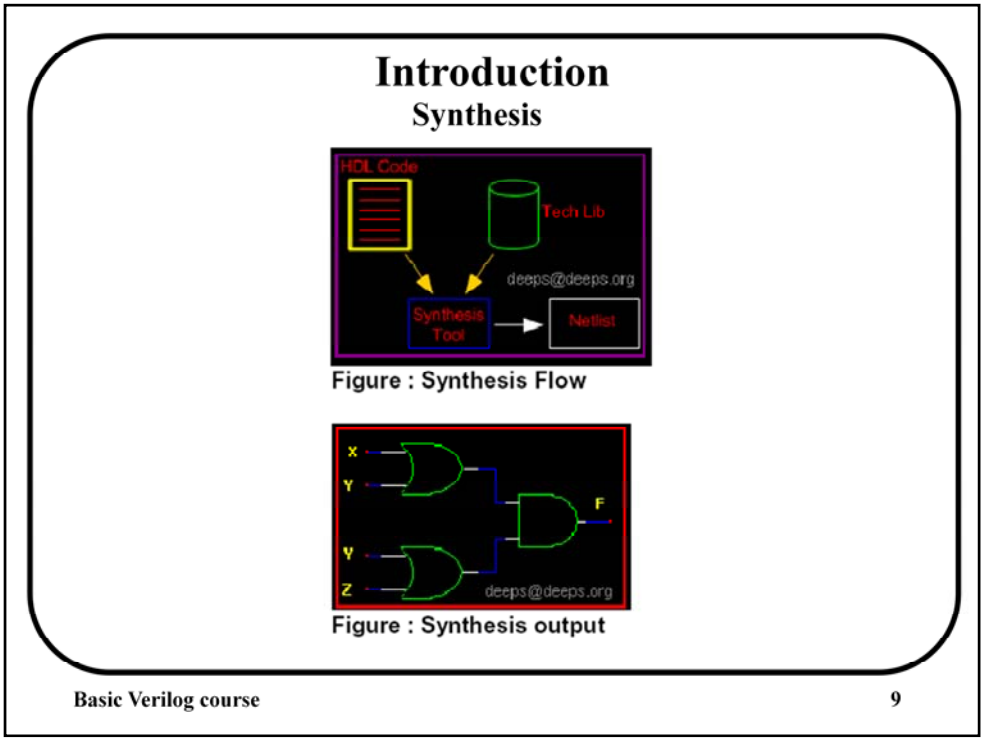


Simulation is the process of verifying the functional characteristics of models at any level of abstraction. We use simulators to simulate the Hardware models. The purpose is to test if the RTL code meets the functional requirements of the specification, see if all the RTL blocks are functionally correct.

To achieve this we need to write testbench, which generates clk, reset and required test vectors.

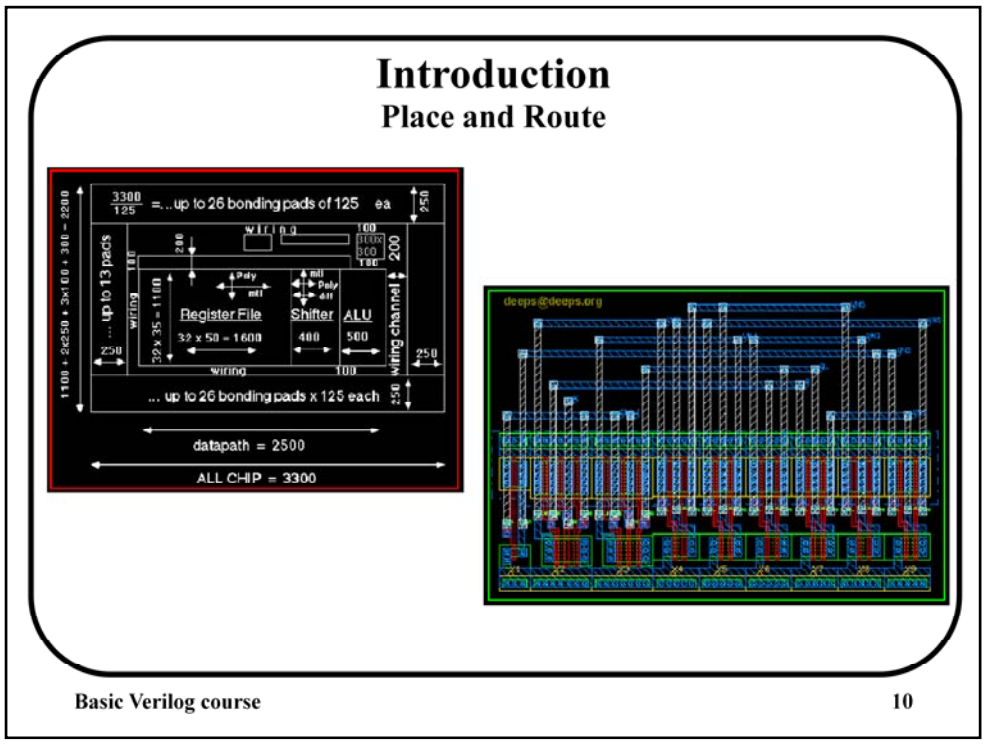
We use waveform output from the simulator to see if the DUT (Device Under Test) is functionally correct. Most of the simulators comes with waveform viewer, As design becomes complex, we write self checking testbench, where testbench applies the test vector, compares the output of DUT with expected value.

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



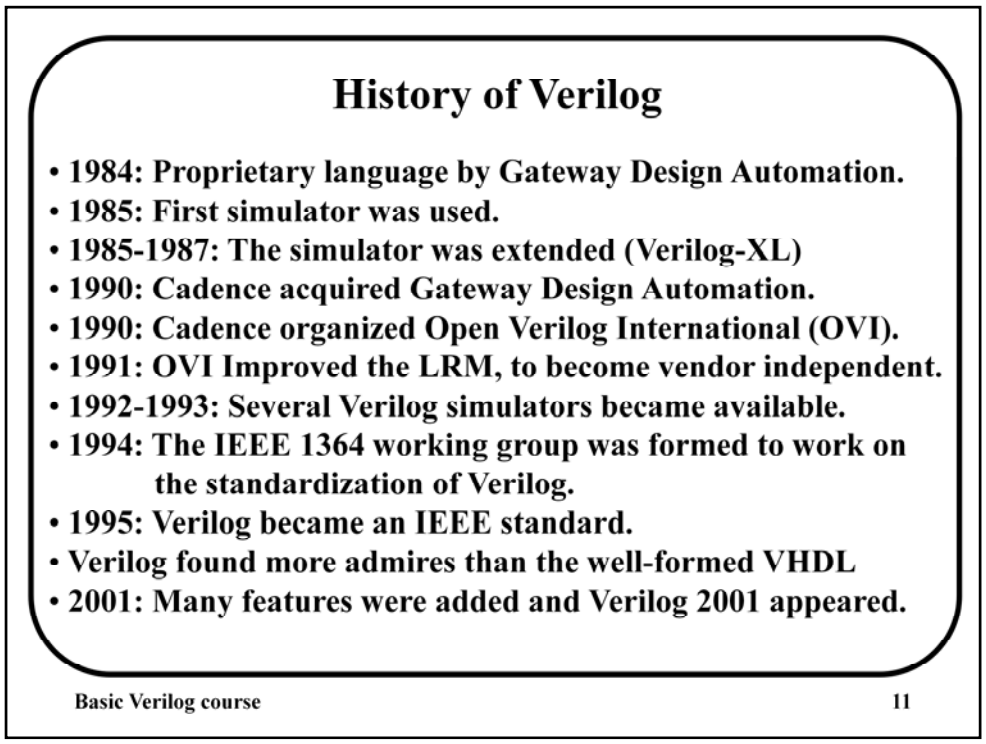
Synthesis is process in which synthesis tool like *design compiler* or *Synplify* takes the RTL in Verilog or VHDL, target technology, and constrains as input and maps the RTL to target technology primitives.

Synthesis tool after mapping the RTL to gates, also do the minimal amount of timing analysis to see if the mapped design is meeting the timing requirements.



Synthesis is process in which synthesis tool like *design compiler* or *Synplify* takes the RTL in Verilog or VHDL, target technology, and constrains as input and maps the RTL to target technology primitives.

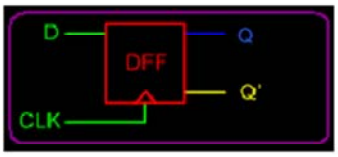
Synthesis tool after mapping the RTL to gates, also do the minimal amount of timing analysis to see if the mapped design is meeting the timing requirements.



- **1984:** Verilog was started initially as a proprietary hardware modeling language by *Gateway Design Automation* Inc. It is rumored that the original language was designed by taking features from the most popular HDL language of the time, called HiLo as well as from traditional computer language such as C.
- **1985:** Verilog simulator was first used.
- **1985-1987:** The simulator was then extended substantially.

  The implementation was the Verilog simulator sold by Gateway. The first major extension was Verilog-XL, which added a few features and implemented the infamous "XL algorithm" which was a very efficient method for doing gate-level simulation.
- **1990:** Cadence Design System, decided to acquire Gateway Automation System. Along with other Gateway product, Cadence now became the owner of the Verilog language, and continued to market Verilog as both a language and a simulator.
- **1990:** Cadence recognized that if Verilog remained a closed language, the pressures of standardization would eventually cause the industry to shift to VHDL. Consequently, Cadence organized Open Verilog International (OVI).
- **1991:** Cadence gave OVI the documentation for the Verilog Hardware Description Language. This was the event which "opened" the language. OVI did a considerable amount of work to improve the Language Reference Manual (LRM), clarifying things and making the language specification as vendor-independent as possible.
- **1992-1993:** Several companies began working on Verilog simulators. In 1992, the first of these were announced, and by 1993 there were several Verilog simulators available from companies other than Cadence. The most successful of these was **VCS**, the Verilog Compiled Simulator, from Chronologic Simulation.
- **1994:** The IEEE 1364 working group was formed to turn the OVI LRM into an IEEE standard.
- **1995:** Verilog became an IEEE standard in December.
- Verilog as a HDL found more admirers than well-formed and federally funded VHDL.
- After many years, new features have been added to Verilog, and new version is called Verilog 2001. This version seems to have fixed lot of problems that Verilog 1995 had.

How does a Verilog code look like?

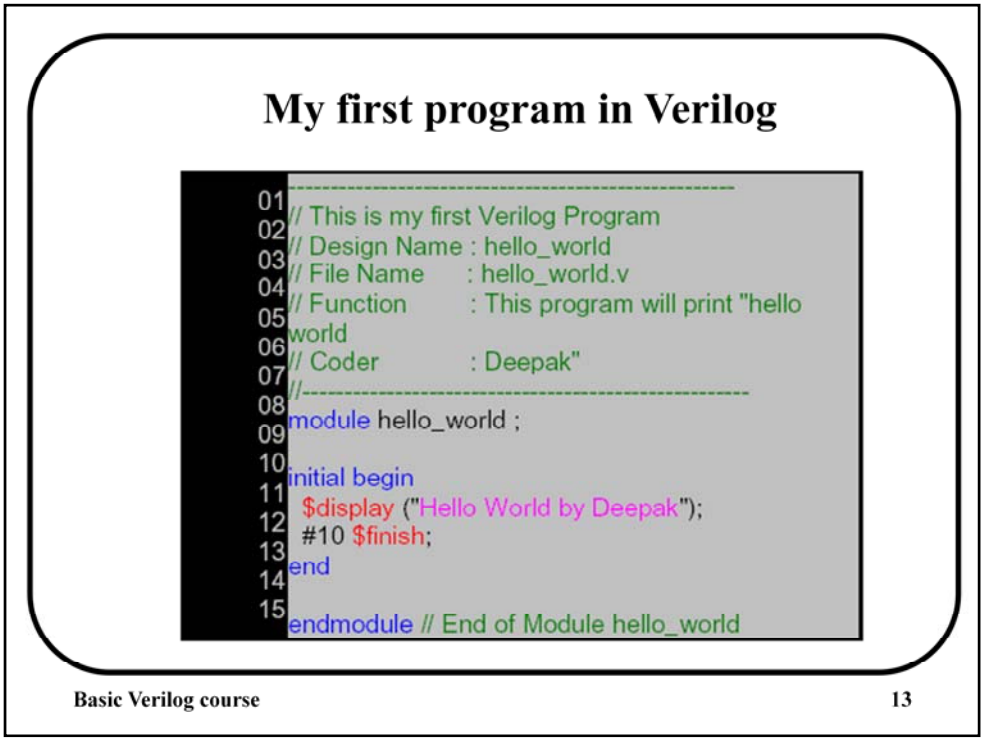



```
module d_ff ( d, clk, q, q_bar);
  input d ,clk;
  ouput q, q_bar;

  always @ (posedge clk)
  begin
    q <= d;
    q_bar <= !d;
  end
endmodule
```

Basic Verilog course 12

One can describe a simple Flip flop as that in the above figure as well as one can describe a complicated designs having 1 million gates. Verilog is one of the HDL languages available in the industry for designing the Hardware.

Many engineers who want to learn Verilog, most often ask this question, how much time it will take to learn Verilog?, Well my answer to them is " **It may not take more then one week, if you happen to know at least one programming language".**



If you refer to any book on programming language it starts with “Hello World" program, once you have written the program, you can be sure that you can do something in that language.

We will show how to write a **"hello world" program** in Verilog.

Words in green are comments, blue are reserved words.

Any program in Verilog starts with reserved word *module <module_name>*

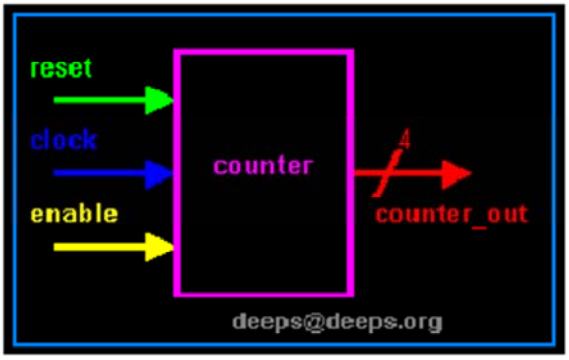
In the above example line 7 contains **module hello_world**;

Line 9 contains the *initial* block, this block gets executed only once after the simulation starts and at time=0 (0ns). This block contains two statements, which are enclosed with in *begin* at line 7 and *end* at line 12.

In Verilog if you have multiple lines within a block, you need to use begin

and end.

Counter Design



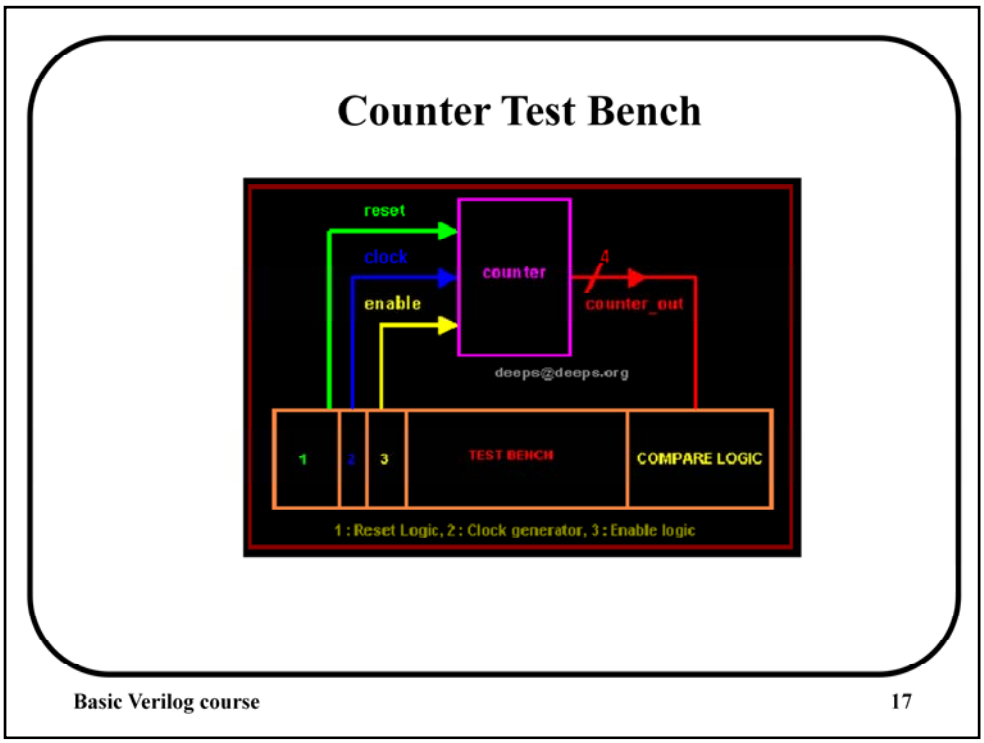
- 4-bit synchronous up counter.
- Active high, synchronous reset.
- Active high enable.

Counter Verilog Code

```
module counter (
clock , // Clock input ot the design
reset , // active high, synchronous Reset input
enable , // Active high enabel signal for counter
counter_out // 4 bit vector output of the counter
); // End of port list

//------------Input Ports--------------------------
input clock ;
input reset ;
input enable ;
//------------Output Ports-------------------------
output [3:0] counter_out ;
//------------Input ports Data Type-----------------
// By rule all the input ports should be wires
wire clock ;
wire reset ;
wire enable ;
//------------Output Ports Data Type----------------
// Output port can be a storage element (reg) or a
Wire reg [3:0] counter_out ;
```

Counter Verilog Code (Continued)

```
//------------Code Starts Here-------------------------
// Since this counter is a positive edge trigged one,
// We trigger the below block with respect to positive
// edge of the clock.

always @ (posedge clock)
begin : COUNTER // Block Name
// At every rising edge of clock we check if reset is active
// If active, we load the counter output with "0000"
  if (reset == 1'b1) begin
  counter_out <= #1 4'b0000;
  end

// If enable is active, then we increment the counter
  else if (enable == 1'b1) begin
    counter_out <= #1 counter_out + 1;
  end
end // end of Block COUNTER
endmodule // end of module counter
```

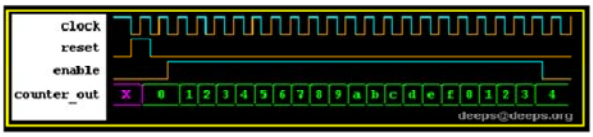


Counter testbench consists of clock generator, reset control, enable control and compare logic. Below is the simple code of testbench without the compare logic.

Counter Test Bench

```
module counter_tb();
// Declare inputs as regs and outputs as wires
reg clock, reset, enable
wire [3:0] counter_out;
// Initialize all variables
initial begin
  clock = 1; // initial value of clock
  reset = 0; // initial value of reset
  enable = 0; // initial value of enable
  #5 reset = 1; // Assert the reset
  #5 reset = 0; // De-assert the reset
  #5 enable = 1; // Assert enable
  #100 enable = 0; // De-assert enable
  #10 $finish; // Terminate simulation
end
```

```
// Clock generator
always begin
  #5 clock = ~clock; // Toggle clock every 5 ticks
end

// Connect DUT to test bench
counter U_counter (
  clock,
  reset,
  enable,
  counter_out
);

endmodule
```



Basic Verilog course 18

A powerful simulator is called Veriwell and is found here

http://sourceforge.net/projects/veriwell/files/VeriWell/Veriwell%202.8.7/veriwell-2.8.7.tar.gz/download

Verilog Syntax and Semantics

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

White Spaces

White space characters are :
- Blank spaces - Tabs - Carriage returns - New-line - Form-feeds

These characters are ignored except when they serve to separate other tokens.

Functional Equivalent Code

```
module addbit(a,b,ci,sum,co);
input a,b,ci;output sum co;
wire a,b,ci,sum,co;
```

Never write code like this.

```
module addbit (
a,
b,
ci,
sum,
co);
input       a;
input       b;
input       ci;
output      sum;
output      co;
wire        a;
wire        b;
wire        ci;
wire        sum;
wire        co;
```

Nice way to write code.

Comments

There are two forms to introduce comments:

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

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

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

Case Sensitivity

Verilog HDL is case sensitive:

- Lower case letters are unique from upper case letters

- All Verilog keywords are lower case

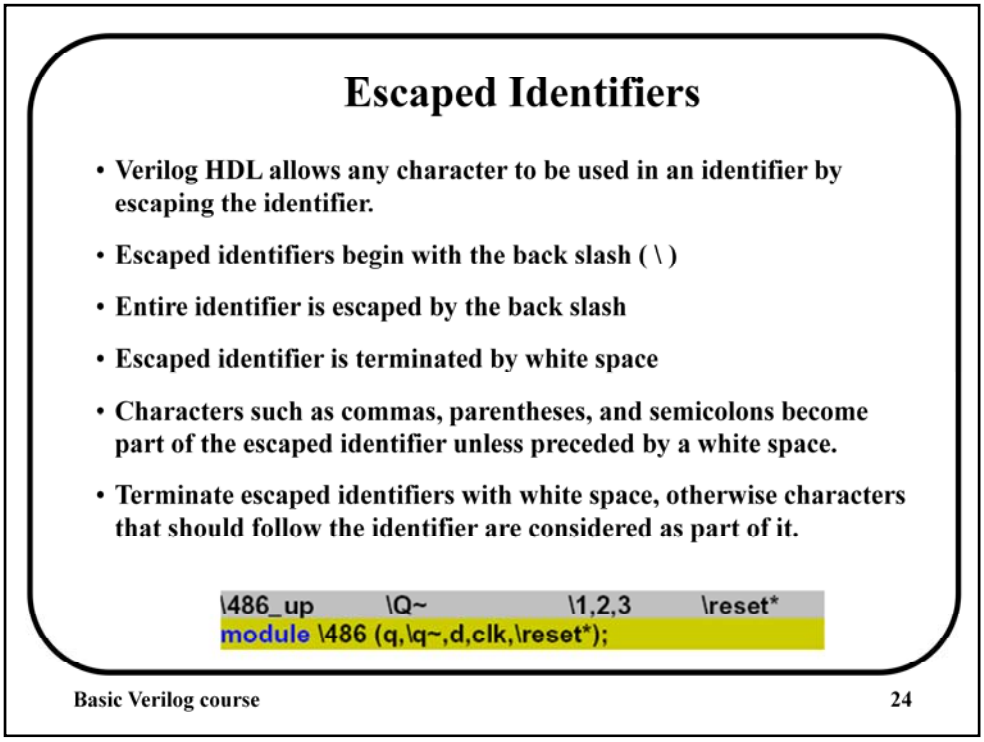
- Recommendation: never use the Verilog keywords as unique names, even if the case is different.

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

Identifiers

- **Identifiers are names used to give an object, such as a register or a module, a name so that it can be referenced from other places in a description.**
- **Identifiers must begin with an alphabetic character or the underscore character (a-z A-Z _)**
- **Identifiers may contain alphabetic characters, numeric characters, the underscore, and the dollar sign (a-z A-Z 0-9 _ $)**
- **Identifiers can be up to 1024 characters long.**

data_input	mu
clk_input	my$clk
i386	A



Verilog HDL allows any character to be used in an identifier by **escaping** the identifier. Escaped identifiers provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

- Escaped identifiers begin with the back slash (\ )
- Entire identifier is escaped by the back slash
- Escaped identifier is terminated by white space
- Characters such as commas, parentheses, and semicolons become part of the escaped identifier unless preceded by a white space.
- Terminate escaped identifiers with white space, otherwise characters that should follow the identifier are considered as part of it.

Integer Numbers

Verilog HDL allows integer numbers to be specified as:

- **Sized or unsized numbers (Unsized size is 32 bits)**
- **In a radix of binary, octal, decimal, or hexadecimal**
- **Radix is case sensitive and hex digits (a,b,c,d,e,f) are insensitive**
- **Spaces are allowed between the size, radix and value**

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

Integer	Stored as	Description
1	00000000000000000000000000000001	unsized 32 bits
8'hAA	10101010	sized hex
6'b10_0011	100011	sized binary
'hF	00000000000000000000000000001111	unsized hex 32 bits

Integer Numbers (continued)

- Verilog expands <value> to be fill the specified <size> by working from *right-to-left.*
- When <size> is *smaller* than <value>, then left-most bits of <value> are truncated
- When <size> is *larger* than <value>, then left-most bits are filled, based on the value of the left-most bit in <value>.
  - Left most '0' or '1' are filled with '0', 'Z' are filled with 'Z' and 'X' with 'X'

Integer	Stored as	Description
6'hCA	001010	truncated, not 11001010
6'hA	001010	filled with two '0' on left
16'bZ	Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z	filled with 16 Z's
8'bx	x x x x x x x x	filled with 8 X's

Real Numbers

Real Number	Decimal notation
1.2	1.2
0.6	0.6
3.5E6	3,500000.0

Signed and Unsigned Numbers

Verilog Supports both the type of numbers, but with certain restrictions.

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

Negative numbers can be specified by putting a minus sign before the size for a constant number, thus become signed numbers.

Verilog internally represents negative numbers in 2's compliment format. An optional signed specifier can be added for signed arithmetic.

32'hDEAD_BEEF	Unsigned or signed positive Number
-14'h1234	Signed negative number

Ports

- **Ports allow communication between a module and its environment.**
- **Ports can be associated by order or by name.**
- **You declare ports to be input, output or inout.**
- **The port declaration syntax is :**
  - **input [range] list_of_identifiers;**
  - **output [range_val] list_of_identifiers;**
  - **inout [range_val] list_of_identifiers;**
- **As a good coding practice, there should be only one port identifier per line.**

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

Example

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

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

endmodule // End Of Module addbit
```

Connecting Modules by Port Order

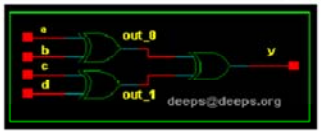
```
module adder (
result      , // Output of the adder
carry       , // Carry output of adder
r1          , // first input
r2          , // second input
ci            // carry input
);
// Input Port Declarations
input   [3:0]  r1       ;
input   [3:0]  r2       ;
input          ci       ;
// Output Port Declarations
output  [3:0]  result   ;
output         carry    ;
// Port Wires
wire    [3:0]  r1       ;
wire    [3:0]  r2       ;
wire           ci       ;
wire    [3:0]  result   ;
wire           carry    ;
// Internal variables
wire           c1       ;
wire           c2       ;
wire           c3       ;
```

```
// Code Starts Here
addbit u0 (
r1[0]       ,
r2[0]       ,
ci          ,
result[0]   ,
c1
);

addbit u1 (
r1[1]       ,
r2[1]       ,
c1          ,
result[1]   ,
c2
);

addbit u2 (
r1[2]       ,
r2[2]       ,
c2          ,
result[2]   ,
c3
);
```

Connecting Modules by Port Order (continued)

```
module parity (
a     , // First input
b     , // Second input
c     , // Third Input
d     , // Fourth Input
y       // Parity  output
);
// Input Declaration
input       a     ;
input       b     ;
input       c     ;
input       d     ;
// Ouput Declaration
output      y     ;
// port data types
wire        a     ;
wire        b     ;
wire        c     ;
wire        d     ;
wire        y     ;
// Internal variables
wire        out_0 ;
wire        out_1 ;
```

```
// Code starts Here
xor u0 (
out_0 ,
a       ,
b
);

xor u1 (
out_1 ,
c       ,
d
);

xor u2 (
y       ,
out_0  ,
out_1
);

endmodule // End Of Module parity
```



Connecting Modules by Port Names

```
module adder (
        result      , // Output of the adder
        carry       , // Carry output of adder
        r1          , // first input
        r2          , // second input
        ci            // carry input
        );
        // Input Port Declarations
        input   [3:0]  r1         ;
        input   [3:0]  r2         ;
        input          ci         ;
        // Output Port Declarations
        output  [3:0]  result     ;
        output         carry      ;
        // Port Wires
        wire    [3:0]  r1         ;
        wire    [3:0]  r2         ;
        wire           ci         ;
        wire    [3:0]  result     ;
        wire           carry      ;
        // Internal variables
        wire           c1         ;
        wire           c2         ;
        wire           c3         ;
```

```
        // Code Starts Here
        addbit u0 (
        .a          (r1[0])       ,
        .b          (r2[0])       ,
        .ci         (ci)          ,
        .sum        (result[0])   ,
        .co         (c1)
        );

        addbit u1 (
        .a          (r1[1])       ,
        .b          (r2[1])       ,
        .ci         (c1)          ,
        .sum        (result[1])   ,
        .co         (c2)
        );

        addbit u2 (
        .a          (r1[2])       ,
        .b          (r2[2])       ,
        .ci         (c2)          ,
        .sum        (result[2])   ,
        .co         (c3)
        );
```

Data Types

Verilog Language has two primary data types:

- Nets: represents structural connections between components.
- Registers: represent variables used to store data.

Every signal has a data type associated with it:

- Explicitly declared with a declaration in your Verilog code.
- Implicitly declared with no declaration but used to connect structural building blocks in your code.
- Implicit declaration is always a net of type wire and is one bit wide.

Types of Nets

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

wire	Default net type, a plain wire
tri	Another name of wire
wand	Wired AND
triand	Another name of wand
wor	Wired OR
trior	Another name for wor
tri1	Wire with built-in pullup
tri0	Wire with built-in pulldown
supply1	Always 1
supply0	Always 0
trireg	Storage node for switch-level modeling

Note : Of all the net types, wire is the one which is most widely used.

Types of Registers

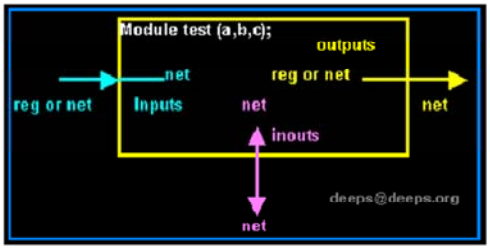
- Registers store the last value assigned to them until another assignment statement changes their value.
- Registers represent data storage constructs.
- You can create arrays of the regs called memories.
- Register data types are used as variables in procedural blocks.
- A register data type is required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

reg	Unsigned variable
integer	Signed variable (32 bits)
time	Unsigned variable (64 bits)
real	Double precision floating point variable

Note : Of all the register types, reg is the one which is most widely used.

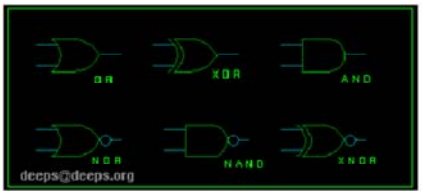
Port Connection Rules

- **Inputs** : internally must always be type *net*, externally the inputs can be connected to variable *reg* or *net* type.
- **Outputs :** internally can be type *net* or *reg*, externally the outputs must be connected to a variable net type.
- **Inouts :** internally or externally must always be type *net*, can only be connected to a variable net type.



Gate Level Modeling

- **Verilog has built-in primitive gates**
- **The gates have one scalar output and multiple scalar inputs. The 1st terminal in the list of gate terminals is an output and the other terminals are inputs.**



and	N-input AND gate
nand	N-input NAND gate
or	N-input OR gate
nor	N-input NOR gate
xor	N-input XOR gate
xnor	N-input XNOR gate

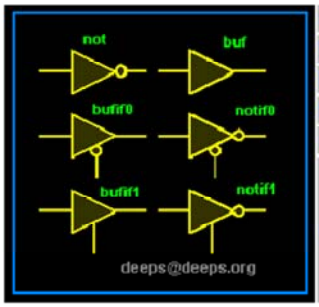
Examples

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

Basic Verilog course 38

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

Transmission Gate Primitives



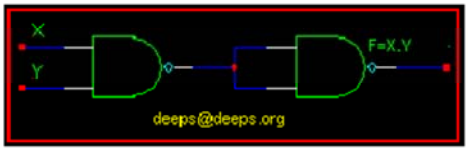
not	N-output invertor.
buf	N-output buffer.
bufif0	Tri-state buffer, Active low en.
bufif1	Tri-state buffer, Active high en.
notif0	Tristate inverter, Low en.
notif1	Tristate inverter, High en.

Examples

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


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

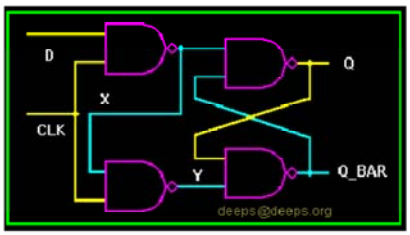
Gate Level Modeling (Example)




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

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

endmodule
```


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

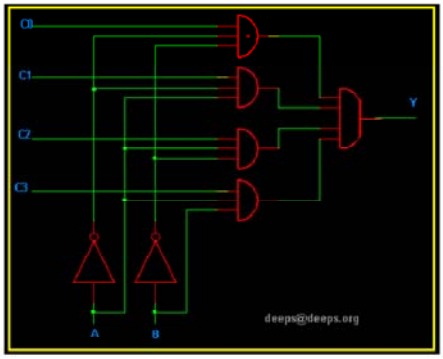
Gate Level Modeling (Example)




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

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

endmodule
```


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

Gate Level Modeling (Example)




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

endmodule
```


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

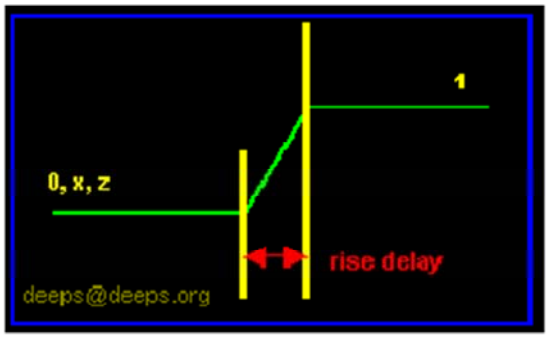
Gate Delays

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

- Rise, Fall and Turn-off delays.
- Minimal, Typical, and Maximum delays.

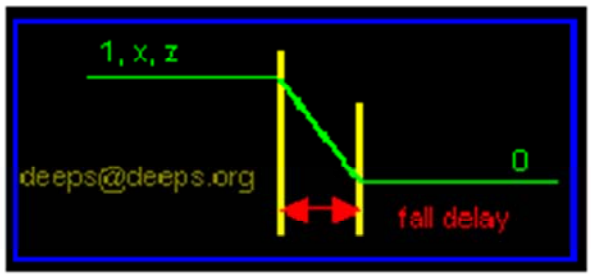
Rise Delay

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



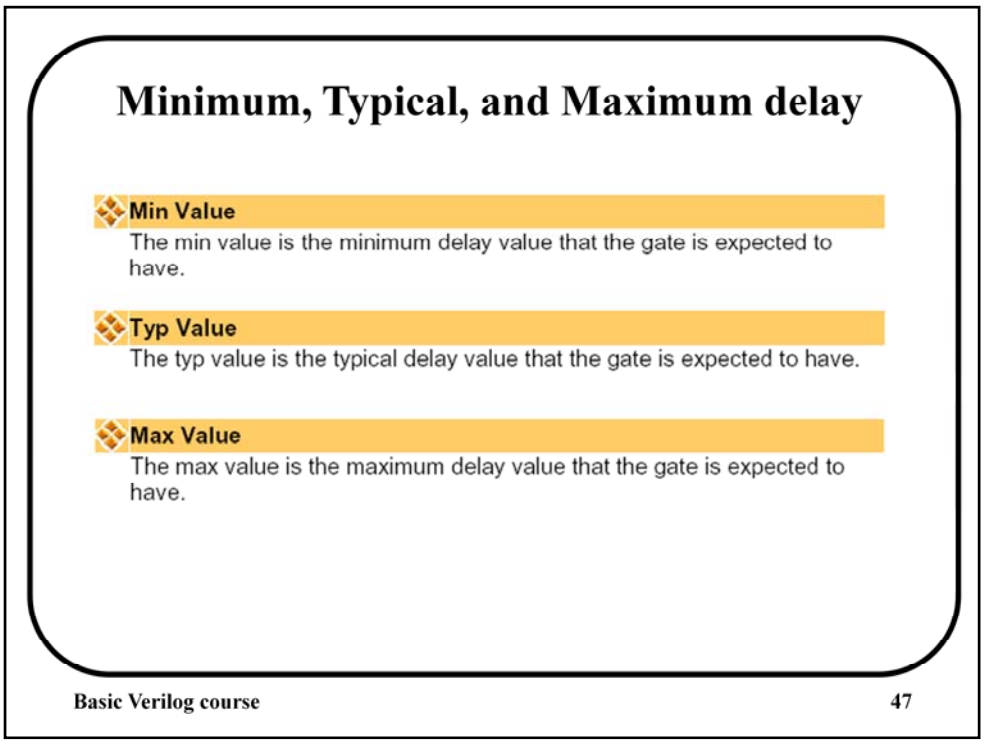
Fall Delay

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



Turn off Delay

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



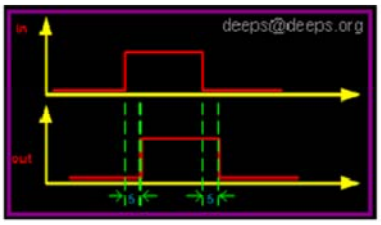
Normally we can have three models of delays, typical, minimum and maximum delay. During compilation of a modules one needs to specify the delay models to use, else Simulator will use the typical model.

Delay Examples

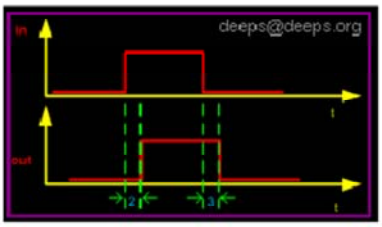
```
// Delay for all transitions
or #5  u_or (a,b,c);

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

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

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

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

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

Gate Delay Example

```
mo dule buf_gate (in,out);
   input in;
   output out;

   buf #(5) (out,in);

endmodule
```




```
mo dule buf_gate (in,out);
   input in;
   output out;

   buf #(2,3) (out,in);

endmodule
```



Verilog Operators
(Relational Operators)

- a<b a less than b
- a>b a greater than b
- a<=b a less than or equal to b
- a>=b a greater than or equal to b
- The result is:
  - 0 if the relation is false
  - 1 if the relation is true
  - x if any of the operands has unknown x bits
- Note: If a value is x or z, then the result of that test is false

Verilog Operators
(Arithmetic Operators)

- **Binary: +, -, *, /, % (the modulus operator)**
- **Unary: +, -**
- **Integer division truncates any fractional part**
- **The result of a modulus operation takes the sign of the first operand.**
- **If any operand bit value is the unknown value x, then the entire result value is x**
- **Register data types are used as unsigned values**
- **Negative numbers are stored in two's complement form**

Verilog Operators
(Equality Operators)

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

- Operands are compared bit by bit, with zero filling if the two operands do not have the same length
- For the = = and ! = operators: the result is x, if either operand contains an x or a z
- For the = = = and ! = = operators: bits with x and z are included in the comparison and must match for the result to be true. The result is always 0 or 1.

Verilog Operators
(Logical Operators)

- ! logic negation
- && logical and
- || logical or
- Expressions connected by && and || are evaluated from left to right.
- Evaluation stops as soon as the result is known.
- The result is a scalar value:
  - 0 if the relation is false.
  - 1 if the relation is true
  - x if any of the operands has unknown x bits

Verilog Operators
(Bit-Wise Operators)

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

- Computations include unknown bits, in the following way:

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

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

Verilog Operators
(Reduction Operators)

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

- Reduction operators are unary.
- They perform a bit-wise operation on a single operand to produce a single bit result.
- Reduction unary NAND and NOR operators operate as AND and OR respectively, but with their outputs negated.
- Unknown bits are treated as described before.

Verilog Operators
(Shift Operators)

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

- The left operand is shifted by the number of bit positions given by the right operand.

- The vacated bit positions are filled with zeroes.

Verilog Operators
(Concatenation Operators)

- Concatenations are expressed using the brace characters { and }, with commas separating the expressions within.
- **Examples**
  {a, b[3:0], c, 4'b1001} // if a and c are 8-bit numbers, the results has 24 bits.
  Unsized constant numbers are not allowed in concatenations.
- Repetition multipliers that must be constants can be used:
  {3{a}} // this is equivalent to {a, a, a}
- Nested concatenations are possible:
  {b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}

Verilog Operators
(Conditional Operators)

- The conditional operator has the following C-like format:

cond_expr ? *true_expr* : *false_expr*

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

Example:

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

Verilog Operators
(Operator Precedence)

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

Behavioral Modeling
(Procedural Blocks)

- There are two types of procedural blocks in Verilog:
  - ***initial***: initial blocks execute only once at time zero (start execution at time zero).
  - ***always***: always blocks loop to execute over and over again, in other words as name means, it executes always.

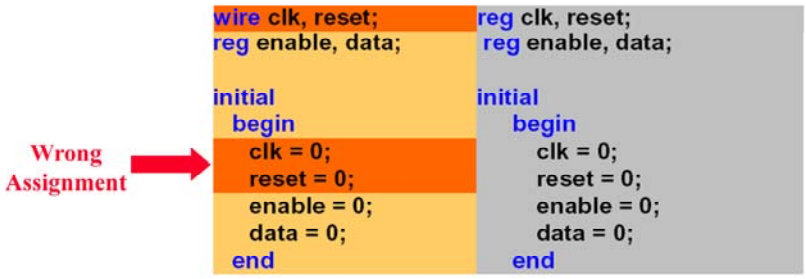
```
initial
  begin
    clk = 0;
    reset = 0;
    enable = 0;
    data = 0;
  end
```

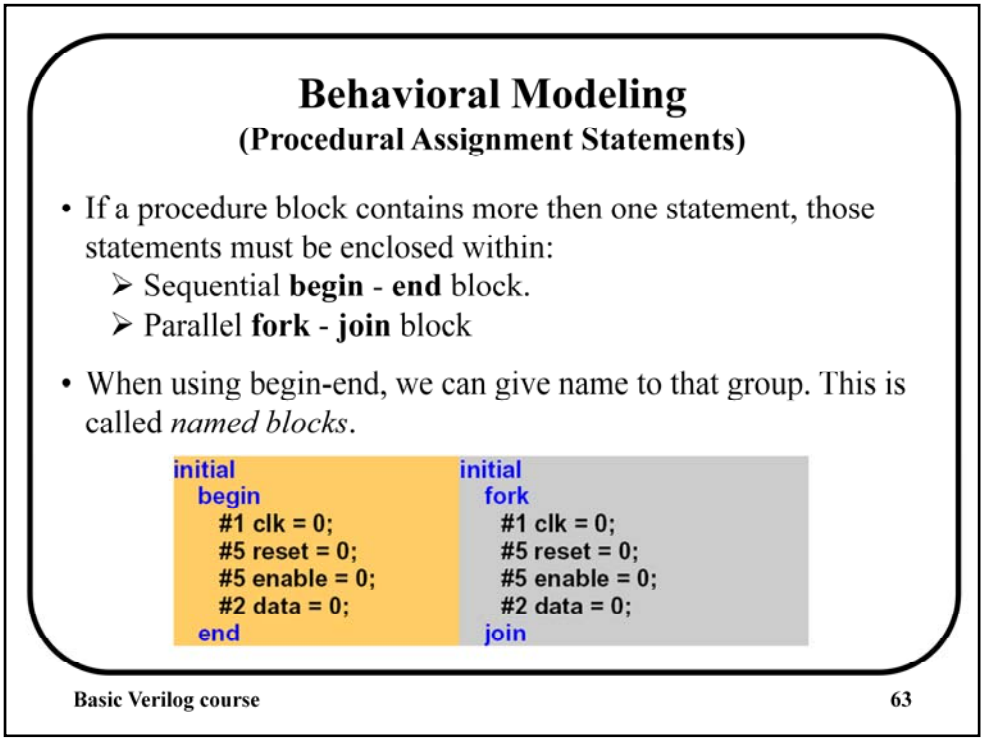
```
always @ (posedge clk)
    begin : D_FF
    if (reset == 1)
      q <= 0;
    else
      q <=d;
    end
```

Behavioral Modeling
(Procedural Assignment Statements)

- Procedural assignment statements *assign values to registers*.
- They *can not assign values to nets* (wire data types).
- You can assign to the register (reg data type):
  - The value of a net (wire),
  - A constant,
  - Another register,
  - A specific value.

Behavioral Modeling
(Procedural Assignment Statements)





Begin:

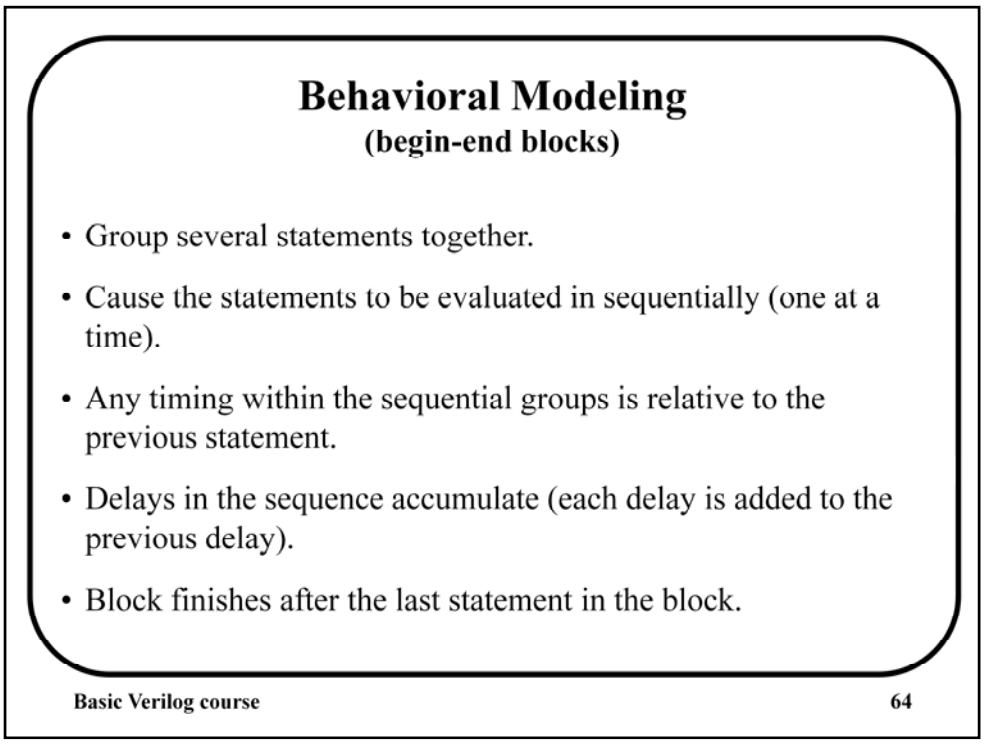
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

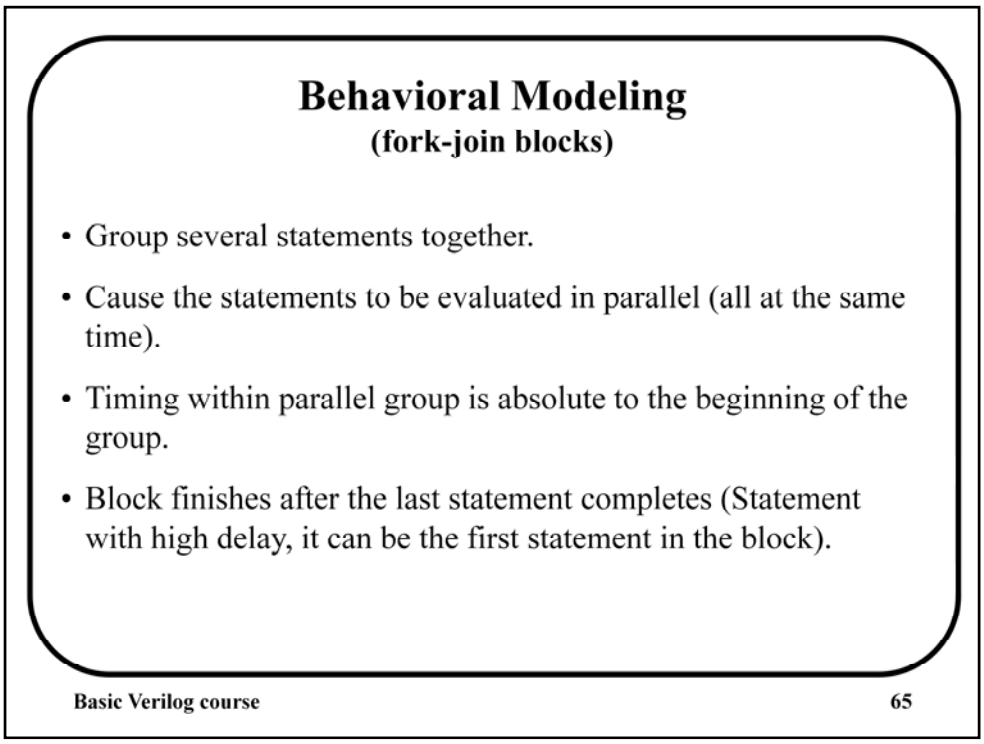
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

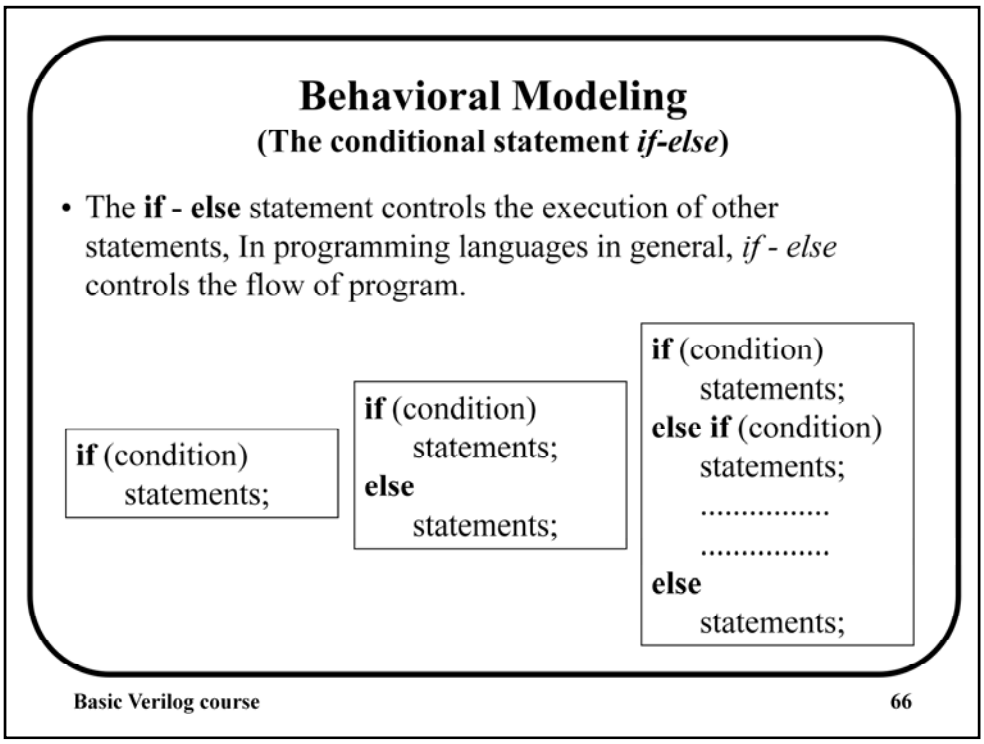
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

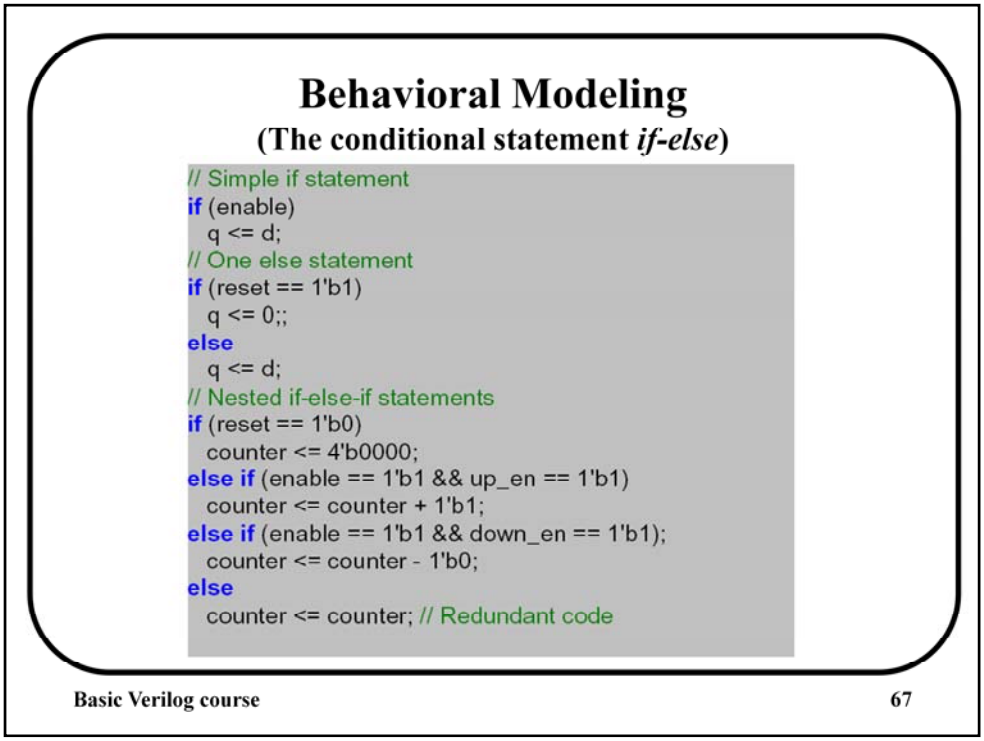
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

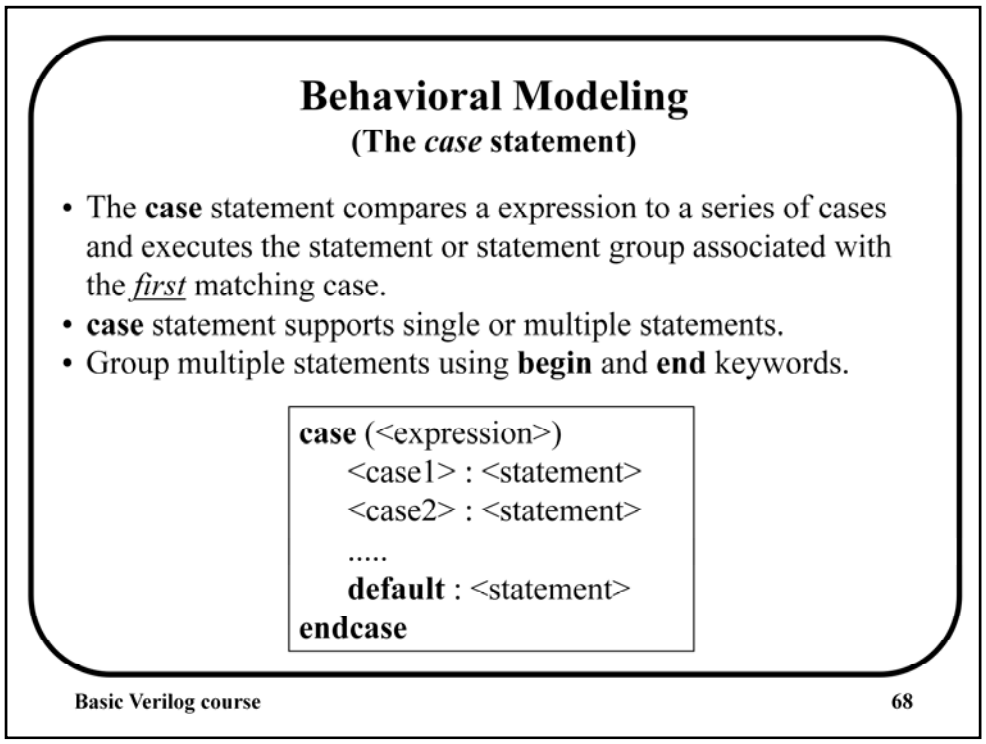
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

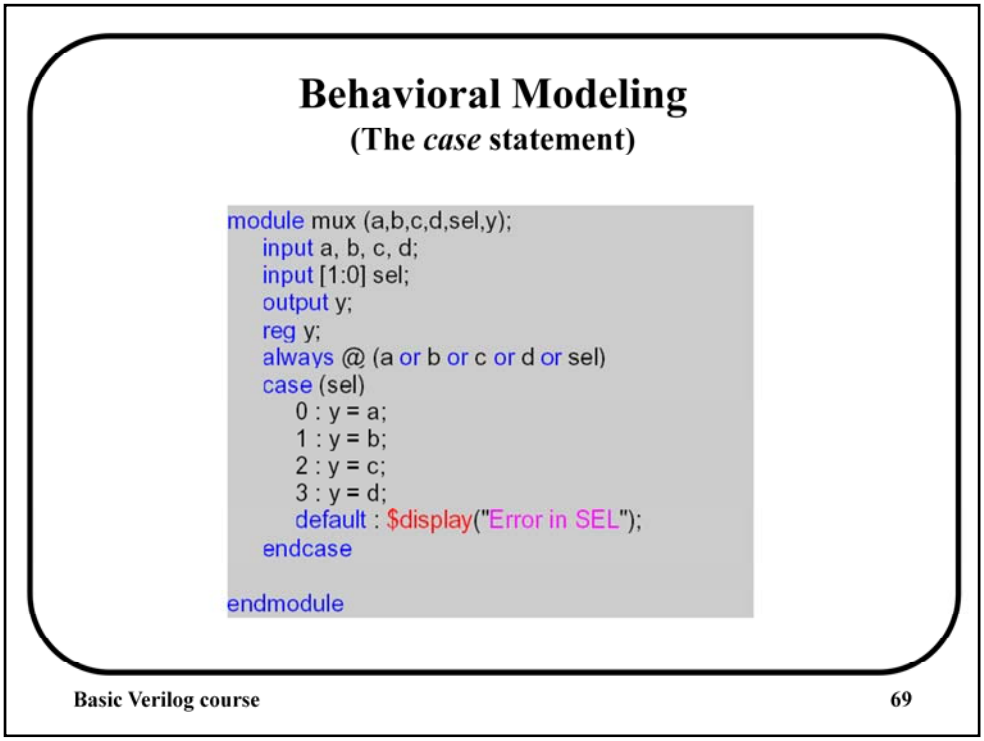
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

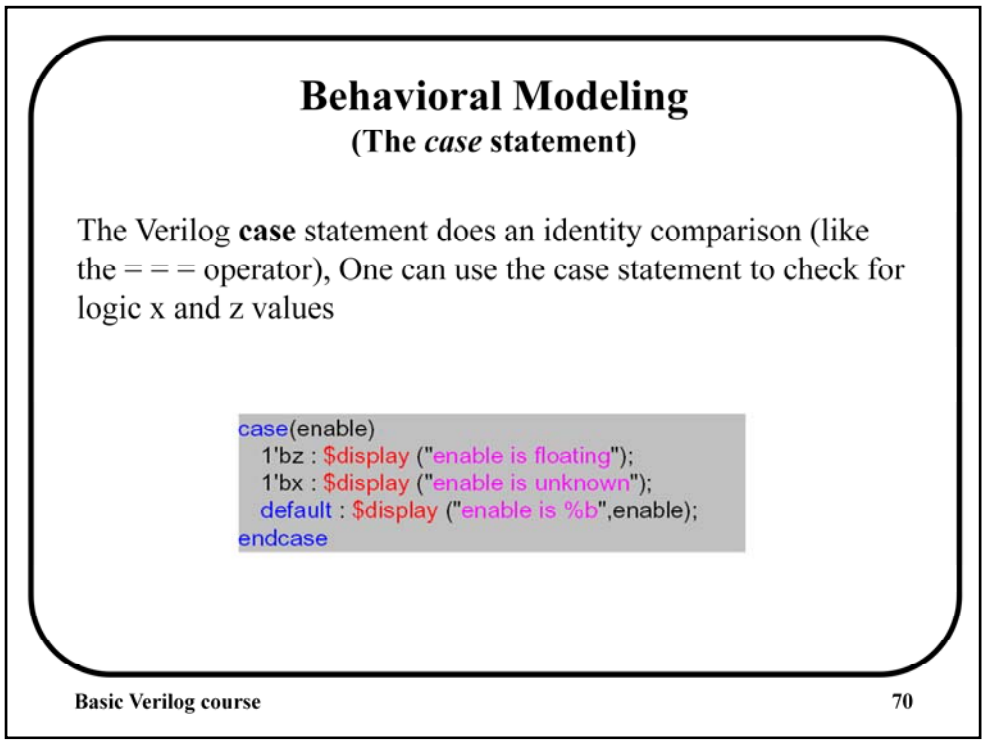
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

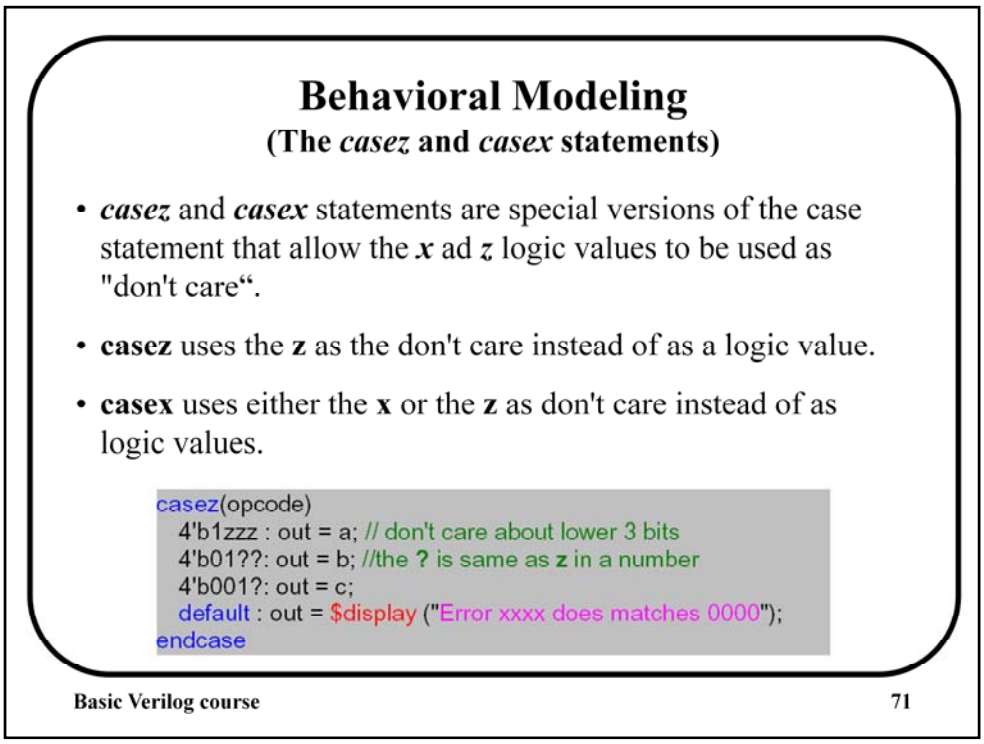
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

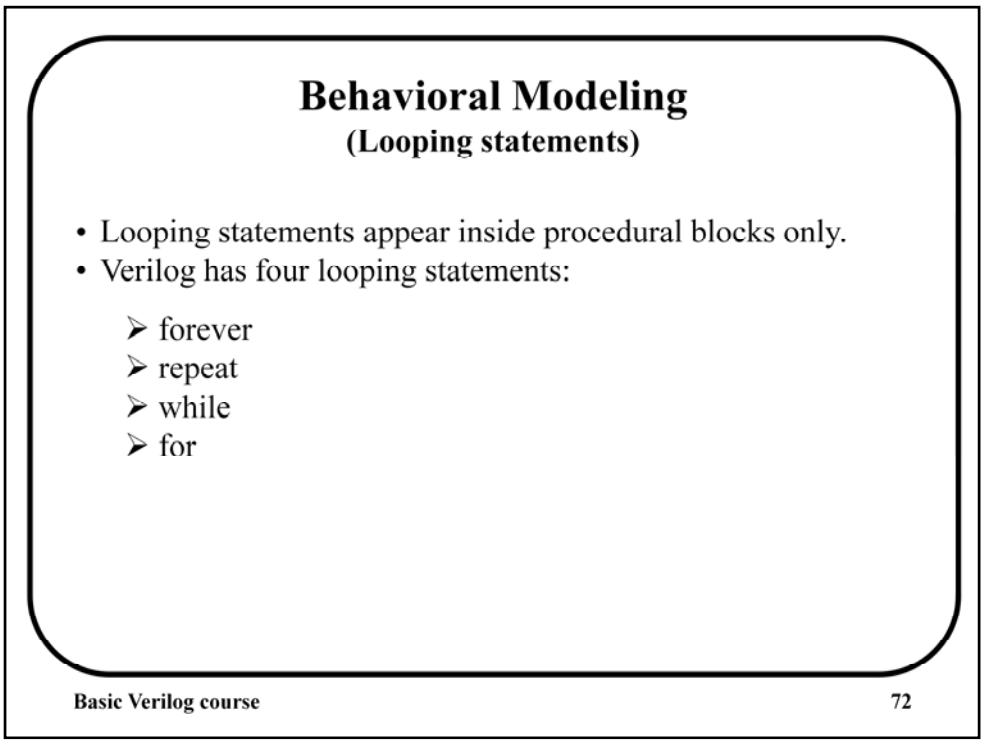
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

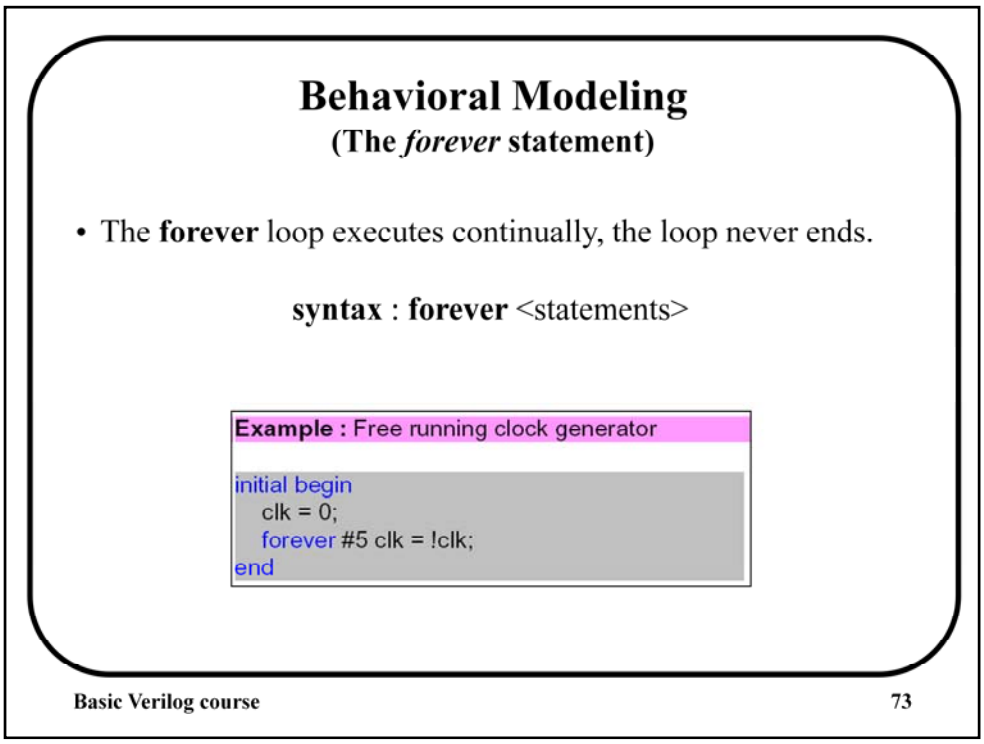
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

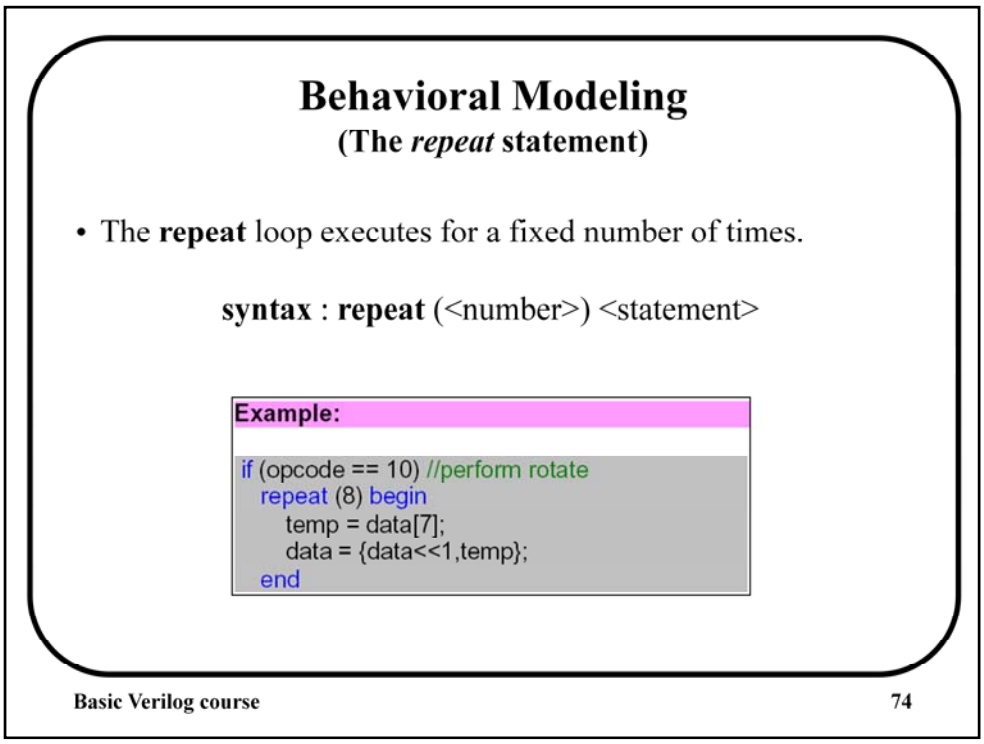
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

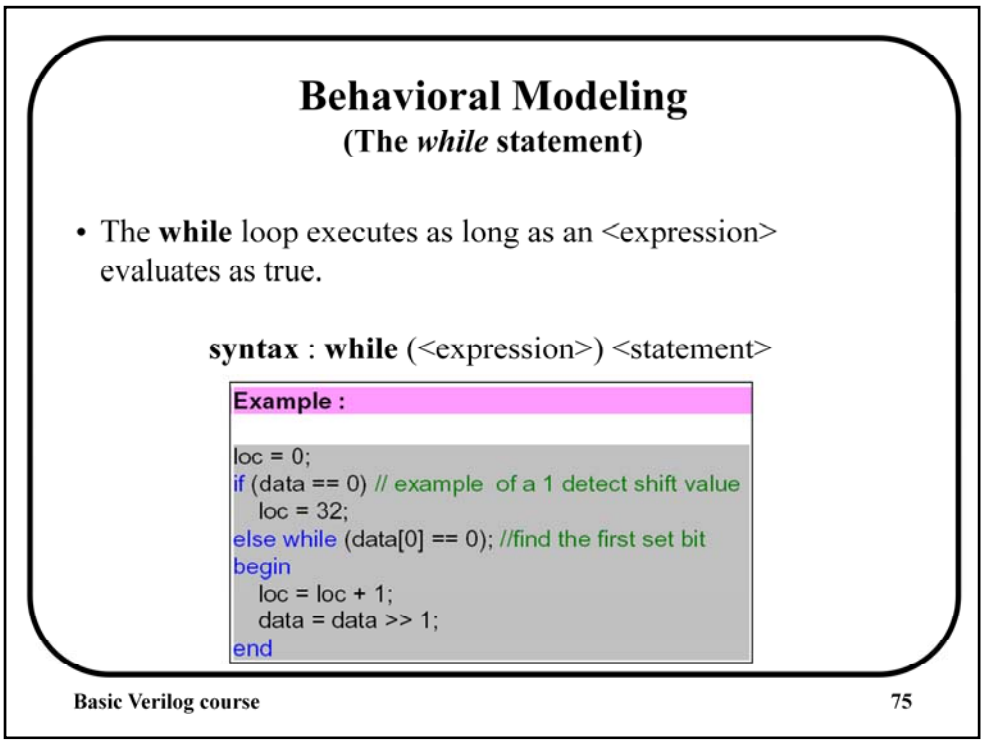
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

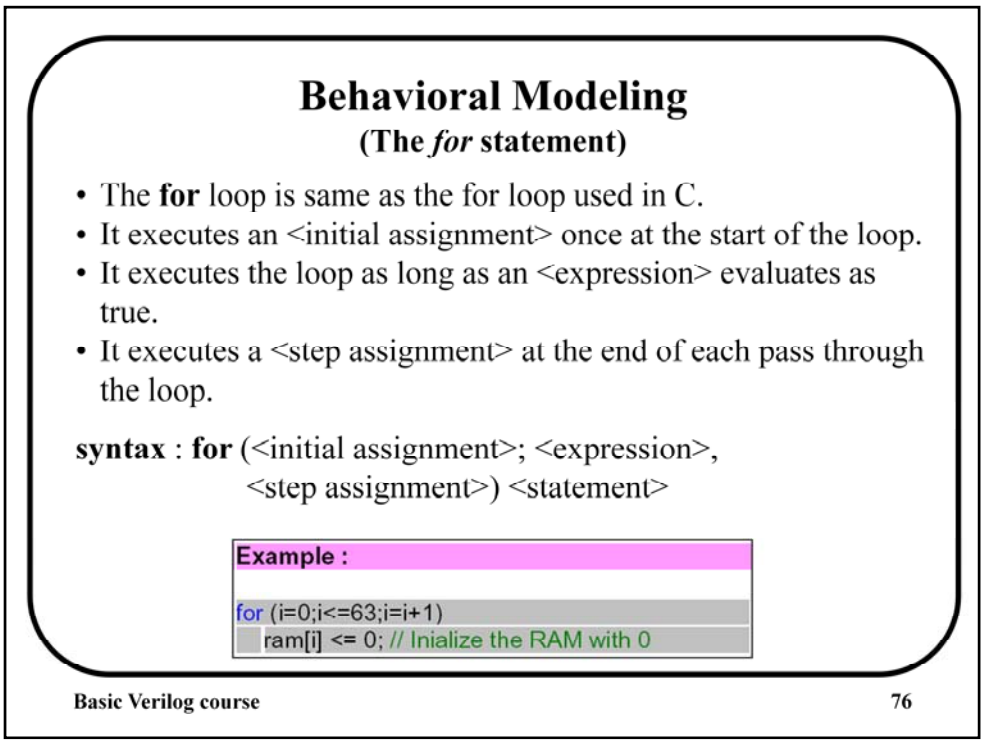
- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.



Begin:

- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.

Behavioral Modeling

(Continuous assignment statements)

- Continuous assignment statements drive nets (wire data type).
- They represent structural connections.
- They can be used for modeling combinational logic.
- They are outside the procedural blocks (always and initial blocks).
- The continuous assign overrides and procedural assignments.
- The left-hand side of a continuous assignment must be net data type.

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

Behavioral Modeling

(Examples on Continuous assignment statements)

Example: 1-bit Adder

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

Example: Tri-State Buffer

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

Behavioral Modeling
(Procedural Block Control)

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

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

- Any change in either *d* or *enable* satisfies the event control and allows the execution of the statements in the procedure.

Behavioral Modeling
(Examples on Procedural Block Control)

Example : 1-bit Adder

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

endmodule
```

Example : 4-bit Adder

```
module adder (a,b,sum,carry);
   input [3:0] a, b;
   output [3:0] sum;
   output carry;
   reg [3:0] sum;
   reg carry;
   always @ (a or b)
   begin
      {carry, sum} = a + b;
   end

endmodule
```

Behavioral Modeling
(Procedural Blocks Concurrency)

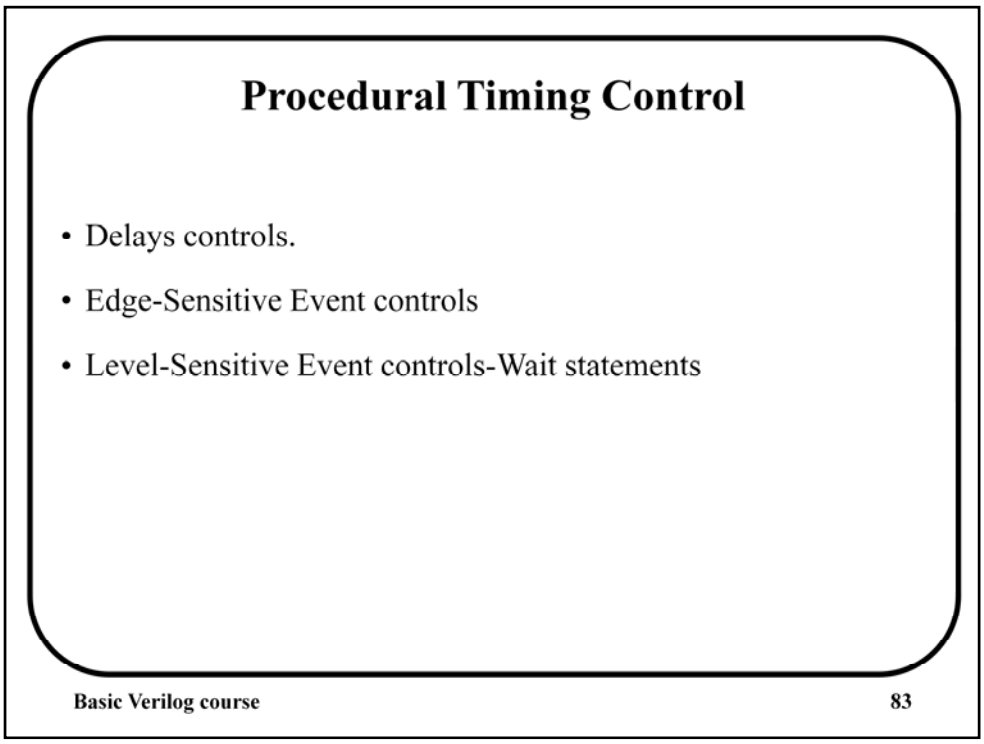
- If we have multiple always blocks inside one module, then all the blocks (i.e. all the always blocks) will start executing at time 0 and will continue to execute concurrently.
- Sometimes this is leads to race condition.

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

  always @ ( c)
    a = c;

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

endmodule
```

Behavioral Modeling
(Race Conditions)

```
initial
  b = 0;

initial
  b = 1;
```

- In the above code it is difficult to say the value of b, as both the blocks are suppose to execute at same time.
- In Verilog if care is not taken, race condition is something that occurs very often.



Begin:

- clk gets 0 after **1** time unit,
- reset gets 0 after **6** time units,
- enable after **11** time units,
- data after **13** units.

All the statements are executed in sequentially.

Fork:

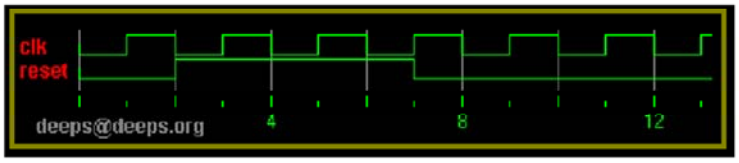
- clk gets value after **1** time unit,
- reset after **5** time units,
- enable after **5** time units,
- data after **2** time units.

All the statements are executed in parallel.

Procedural Timing Control
(Delay Controls)

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

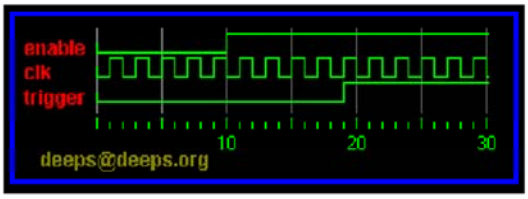
```
module clk_gen (clk,reset);
  output clk,reset;
  reg clk, reset;
  initial begin
     clk = 0;
     reset = 0;
     #2 reset = 1;
     #5 reset = 0;
  end
  always
  #1 clk = !clk;
endmodule
```



Procedural Timing Control
(Edge Sensitive Event Controls)

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

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



Procedural Timing Control
(Level Sensitive Event Controls)

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

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

System Tasks and Functions

- There are tasks and functions that are used to generate input and output during simulation.
- Their names begin with a dollar sign ($).
- The synthesis tools parse and ignore system functions, and hence can be included even in synthesizable models.

System Tasks and Functions
($display, $strobe, $monitor)

- These commands display text on the screen during simulation.
- ***$display*** and ***$strobe*** display once every time they are executed.
- ***$monitor*** displays every time one of its parameters changes.
- 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)
- **%5d**, **%5b** etc. would give exactly 5 spaces for the number instead of the space needed.

System Tasks and Functions
($display, $strobe, $monitor)

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..*);

System Tasks and Functions
($time, $stime, $realtime)

- $time returns the current simulation time as 64-bit integer.
- $stime returns the current simulation time as 32-bit integer.
- $realtime returns the current simulation time as a real number.

System Tasks and Functions
($reset, $stop, $finish)

- **$reset:** resets the simulation back to time 0;
- **$stop:** halts the simulator and puts it in the interactive mode where the user can enter commands;
- **$finish:** exits the simulator back to the operating system.

System Tasks and Functions
($random)

- **$random:** generates a random integer every time it is called.
- If the sequence is to be repeatable, the first time one invokes random give it a numerical argument (a seed). Otherwise the seed is derived from the computer clock.

Syntax

```
data_out = $random (seed);
```

System Tasks and Functions

($fopen, $fclose, $fdisplay, $fstrobe, $fmonitor, $fwrite)

- These commands write more selectively to files.
- **$fopen:** opens an output file and gives the open file a handle for use by the other commands.
- **$fclose:** closes the file and lets other programs access it.
- **$fdisplay,** and **$fwrite** write formatted data to a file whenever they are executed. They are the same except **$fdisplay** inserts a new line after every execution and $write does not.
- **$strobe** also writes to a file when executed, but it waits until all other operations in the time step are complete before writing.
- **$monitor** writes to a file whenever any one of its arguments changes.

System Tasks and Functions

($fopen, $fclose, $fdisplay, $fstrobe, $fmonitor, $fwrite)

Syntax

- handle1=$fopen("filenam1.suffix")
- handle2=$fopen("filenam2.suffix")
- $fstrobe(handle1, format, variable list) //strobe data into filenam1.suffix
- $fdisplay(handle2, format, variable list) //write data into filenam2.suffix
- $fwrite(handle2, format, variable list) //write data into filenam2.suffix all on one line. Put in the format string where a new line is desired.

Modeling Memories

- To help modeling of memory, Verilog provides support of two dimension arrays.
- Behavioral models of memories are modeled by declaring an array of register variables, any word in the array may be accessed by using an index into the array.
- A temporary variable is required to access a discrete bit within the array.

Syntax

```
reg [wordsize:0] array_name [0:arraysize]
```

Modeling Memories
(examples)

Declaration

reg [7:0] my_memory [0:255];

Here [7:0] is width of memory and [0:255] is depth of memory with following parameters

- Width : 8 bits, little endian
- Depth : 256, address 0 corresponds to location 0 in array.

Storing Values

my_memory[address] = data_in;

Reading Values

data_out = my_memory[address];

Modeling Memories
(Reading individual bits)

Bit Read

Sometime there may be need to just read only one bit. Unfortunately Verilog does not allow to read only or write only one bit, the work around for such a problem is as shown below.

```
data_out = my_memory[address];

data_out_it_0 = data_out[0];
```

Modeling Memories
(Initializing Memories)

- A memory array may be initialized by reading memory pattern file from disk and storing it on the memory array.
- To do this, we use system task $readmemb and $readmemh.
- $readmemb is used for binary representation of memory content.
- $readmemh is used for hex representation of memory content.

Modeling Memories
(Initializing Memories)

Syntax

$readmemh("file_name",mem_array,start_addr,stop_addr);

Note : start_addr and stop_addr are optional.

Example : Simple memory

```
module memory;

  reg [7:0] my_memory [0:255];

  initial
  begin
     $readmemh("memory.list", my_memory);
  end
endmodule
```

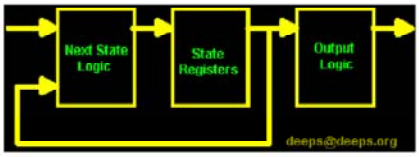
Modeling Memories
(Form of the memory file)

Example : Memory.list file

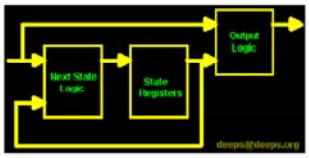
```
//Comments are allowed
1100_1100   // This is first address i.e 8'h00
1010_1010   // This is second address i.e 8'h01
@ 55          // Jump to new address 8'h55
0101_1010  // This is address 8'h55
0110_1001  // This is address 8'h56
```

Modeling Finite State Machines
(Introduction)

- State machines or FSM are the heart of any digital design, of course.
- Counter is a simple form of FSM.
- There are two types of state machines:
  - ➢Moore machine: outputs are only a function of the present state.
  - ➢Mealy machine: outputs are a function of both present state and the inputs.



Moore Machine

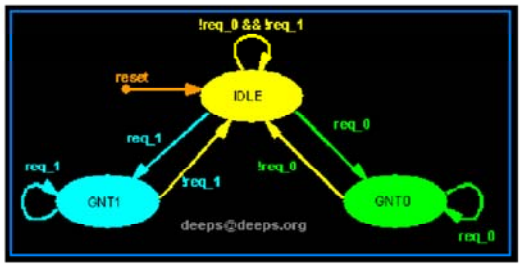


Mealy Machine

Modeling Finite State Machines
(Introduction)

- State machines can also be classified based on type state encoding used.
- Encoding style is also a critical factor which decides speed, and gate complexity of the FSM.
- State encoding types include:
  - Binary encoding,
  - Gray code encoding,
  - One hot encoding,
  - One cold econding,
  - Almost one hot encoding.

Modeling Finite State Machines
(Example)



- The Verilog code of a FSM should have three sections:
  - Encoding style.
  - Next state logic
  - Output logic.

Modeling Finite State Machines
(Encoding Style)

One Hot Encoding

```
parameter [1:0]  IDLE  = 3'b001,
                 GNT0 = 3'b010,
                 GNT1 = 3'b100;
```

Binary Encoding

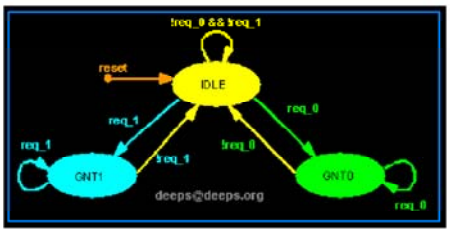
```
parameter [1:0]  IDLE  = 2'b00,
                 GNT0 = 2'b01,
                 GNT1 = 2'b10;
```

Gray Encoding

```
parameter [1:0]  IDLE  = 2'b00,
                 GNT0 = 2'b10,
                 GNT1 = 2'b01;
```

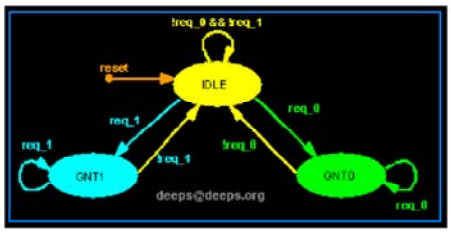
Modeling Finite State Machines
(FSM Code)

```
module fsm (clock , reset, req_0 , req_1, gnt_0 , gnt_1);
//-------------Input Ports-----------------------------
input clock, reset, req_0, req_1;

//-------------Output Ports----------------------------
output gnt_0, gnt_1;

//-------------Input ports Data Type-------------------
wire clock, reset, req_0, req_1;

//-------------Output Ports Data Type------------------
reg gnt_0, gnt_1;

//-------------Internal Constants----------------------
parameter IDLE = 3'b001, GNT0 = 3'b010, GNT1 = 3'b100 ;

//-------------Internal Variables----------------------
reg [2:0] state;
reg [2:0] next_state;
```



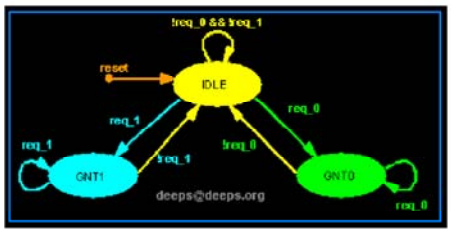
Modeling Finite State Machines
(Next state logic)

```
always @ (state or req_0 or req_1)
begin
  next_state = 3'b000;
  case(state)
    IDLE : if (req_0 == 1'b1)
             next_state = GNT0;
           else if (req_1 == 1'b1)
             next_state= GNT1;
           else
             next_state = IDLE;
    GNT0 : if (req_0 == 1'b1)
             next_state = GNT0;
           else
             next_state = IDLE;
    GNT1 : if (req_1 == 1'b1)
             next_state = GNT1;
           else
             next_state = IDLE;
    default : next_state = IDLE;
  endcase
end
```



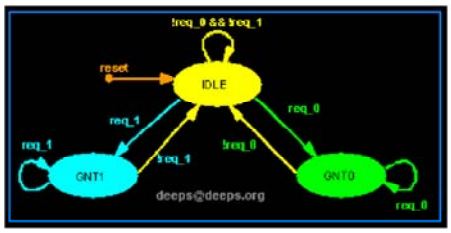
Modeling Finite State Machines
(Output Logic)

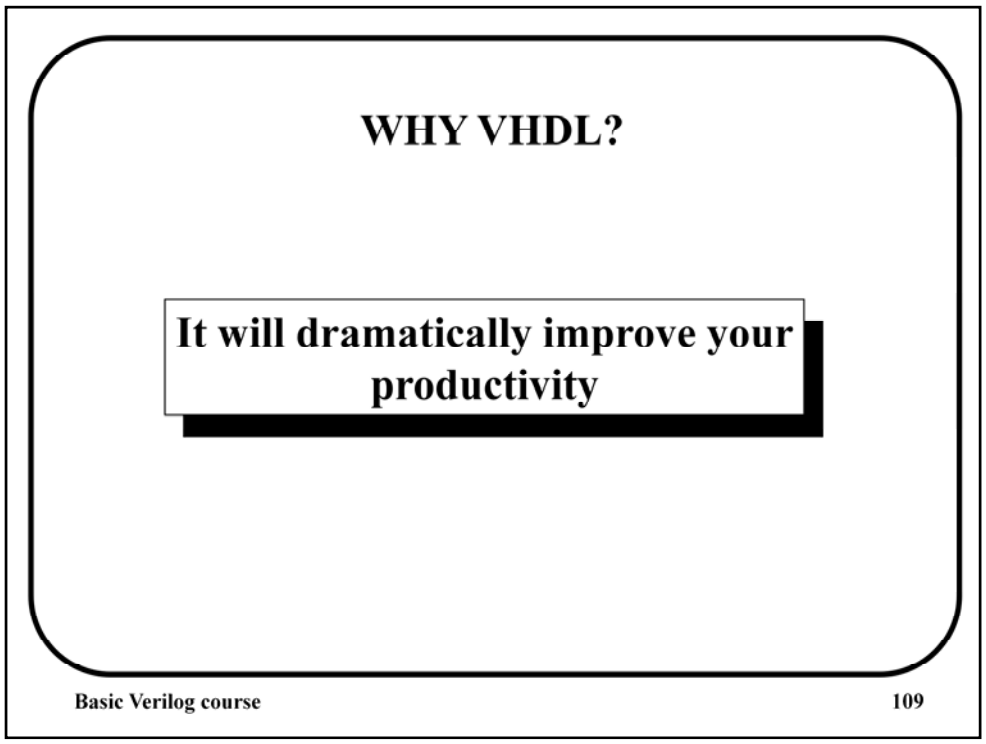
```
always @ (posedge clock)
begin
  if (reset == 1'b1)
  begin
    gnt_0 <= #1 1'b0;
    gnt_1 <= #1 1'b0;
  end
  else
  begin
    case(state)
      IDLE : begin
        gnt_0 <= #1 1'b0;
        gnt_1 <= #1 1'b0; end
      GNT0 : begin
        gnt_0 <= #1 1'b1;
        gnt_1 <= #1 1'b0; end
      GNT1 : begin
        gnt_0 <= #1 1'b0;
        gnt_1 <= #1 1'b1; end
      default : begin
        gnt_0 <= #1 1'b0;
        gnt_1 <= #1 1'b0; end
    endcase
  end
end
```



Modeling Finite State Machines
(State Transition)

```
always @ (posedge clock)
begin
  if (reset == 1'b1)
  begin
    state <= #1 IDLE;
  end
  else
  begin
    state <= #1 next_state;
  end
end
```

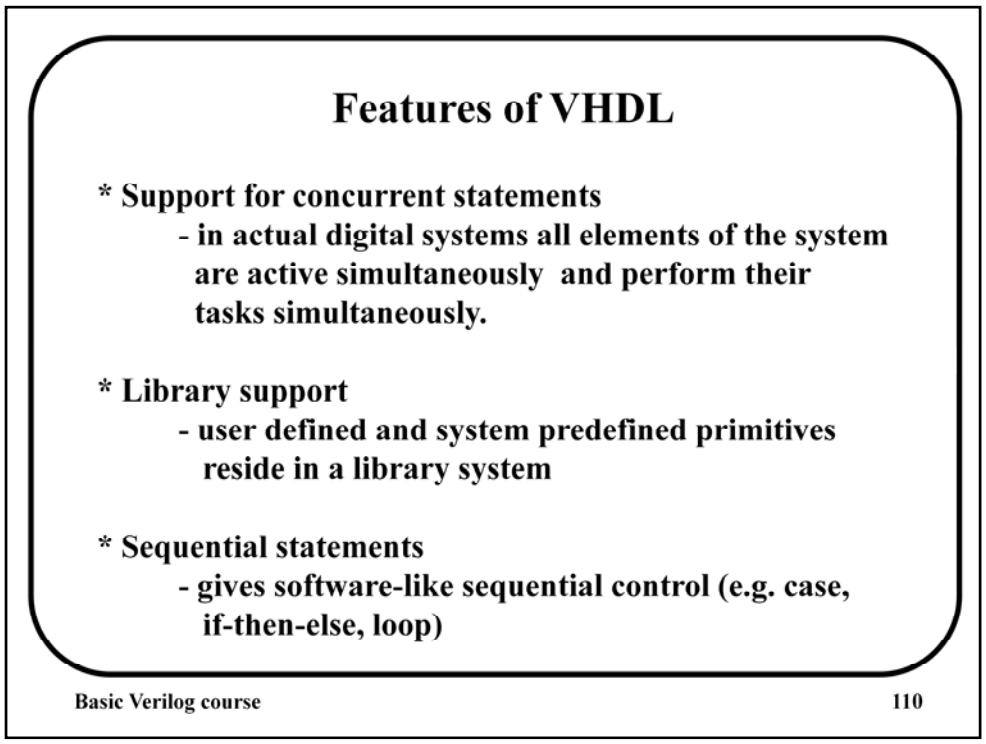




Can you really expect to get your projects done faster using VHDL than using other existing design method ? The answer is yes, but probably not the first time you use it, and only **if you apply VHDL in a structured manner**. VHDL, like any high level design language, is of most benefit when you use a structured, top-down approach to design. Real increases in productivity will come later **when you have accumulated a library of re-usable VHDL components**. Productivity increases will also occur **when you begin to use VHDL to enhance communication between team members**, and take advantage of the more powerful tools for simulation and design verification that are available.

VHDL makes it easy to build and use libraries of commonly-used VHDL modules. As you discover the benefits of reusable code you will soon find yourself thinking of ways to write your VHDL statements that will make them general-purpose; writing portable code will become an automatic reflex.

Another way of improving the performance is that VHDL is a standard language. Using VHDL you can greatly improve your chances of moving into more advanced tools without having to re-enter your circuit descriptions.



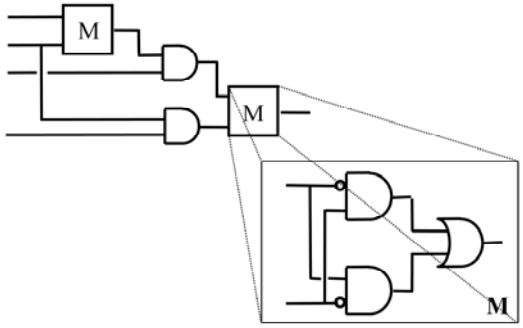
* Digital circuits are characterized by the occurrence of concurrent actions. A hardware description language must be able to create concurrently executed statements. VHDL simulators treat these statements as if they are executed at the same time.

* Reuse is an important issue to improve the productivity. Libraries in VHDL provide storage for compiled design units. This design units can be reused in larger designs, thus saving the time for creating and validating these units.

* Besides the concurrent statements, VHDL provides also the ability to write and control the execution of sequential statements. Sequential statements are useful for modeling in that they ease the description and produce more readable code.

Features of VHDL (contd.)

* **Support for design hierarchy**

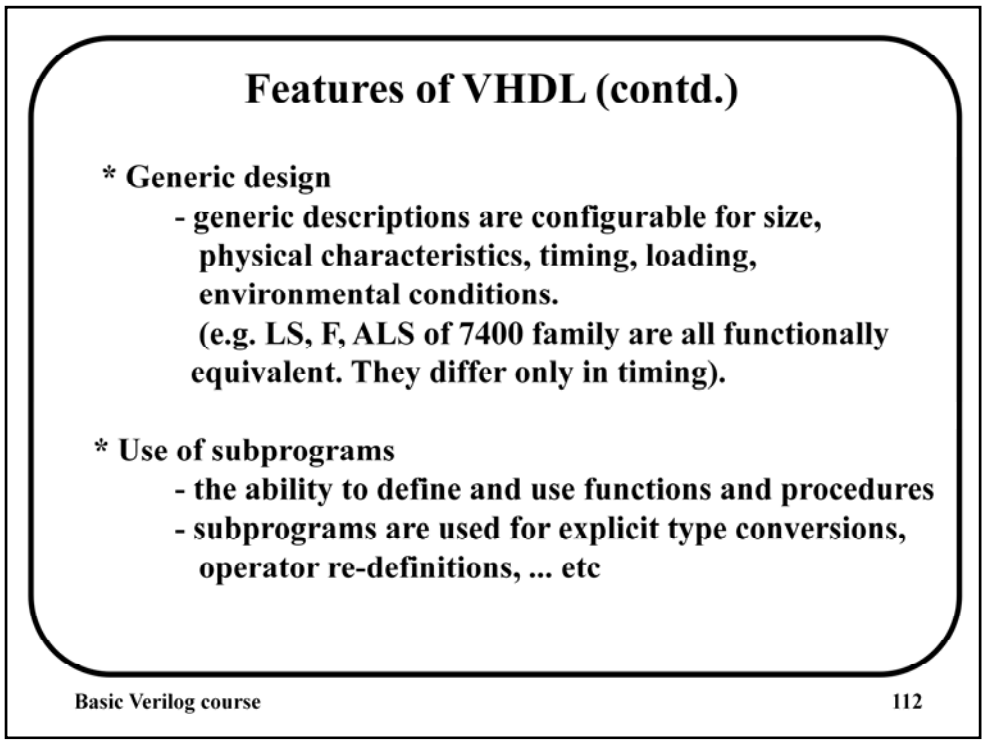


Basic Verilog course 111

* Flat designs work well for small designs where the details of the underlying definition of a functional block does not distract the designer from understanding the functionality of the chip-level design.

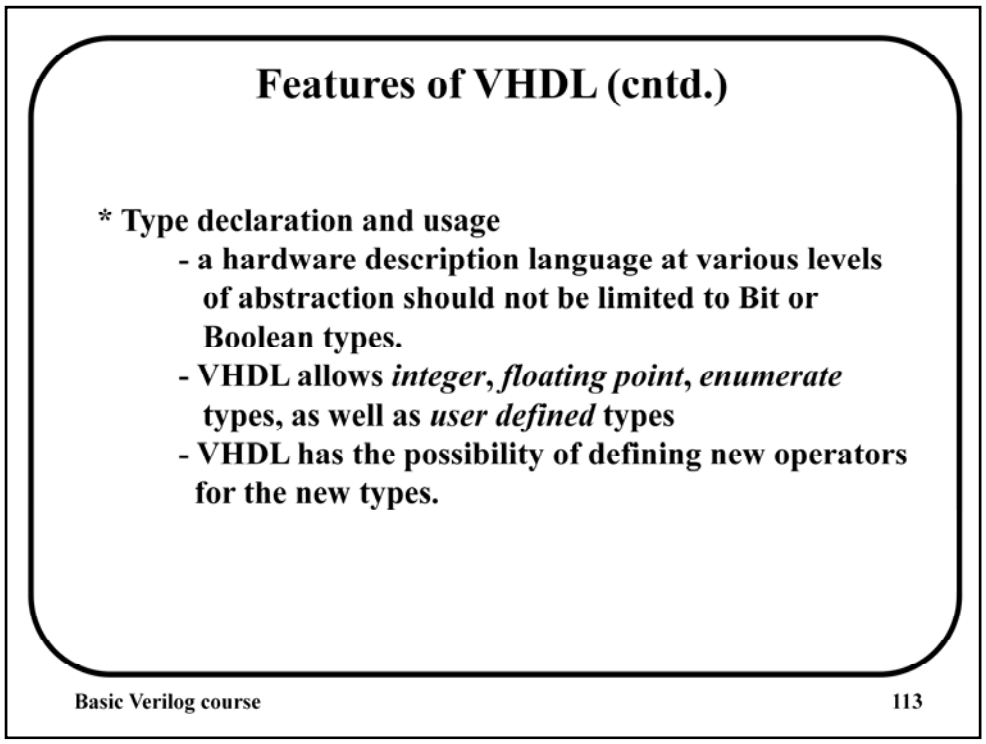
* Hierarchical schemes proved to be useful in large designs consisting of multiple complex functional components. Using multiple levels of the hierarchy can clarify the interconnection of components.

* VHDL gives you the possibility to make hierarchical as well as flat designs.



* VHDL allows you to generate parameterized components. Parameterized components are similar to any other components but they have one or more parameters defining the size or the feature set of the component (e.g *n* bit adder may be described, with a delay *d*. Later in the design process the values of *n* and *d* are specified).

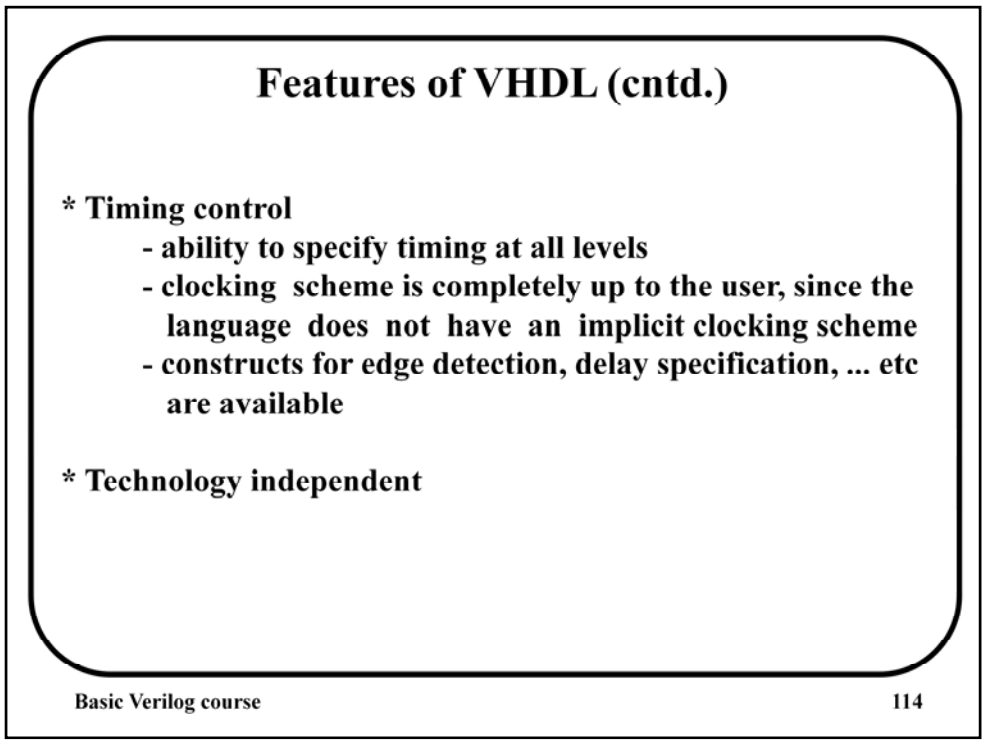
* Using VHDL you can write functions and procedures, which are high level design constructs that compute values or define processes for type conversion, operator overloading, … etc.



In a lot of digital systems you need to use arithmetic functions such as additions and subtractions, to describe adders, subtractors, incrementers, and decrementers. At high level of abstraction, you may describe these functions using integers and floating point data types.

VHDL allows the use of integers, floating point, and enumerate data types. It has already arithmetic operators defined for these types.

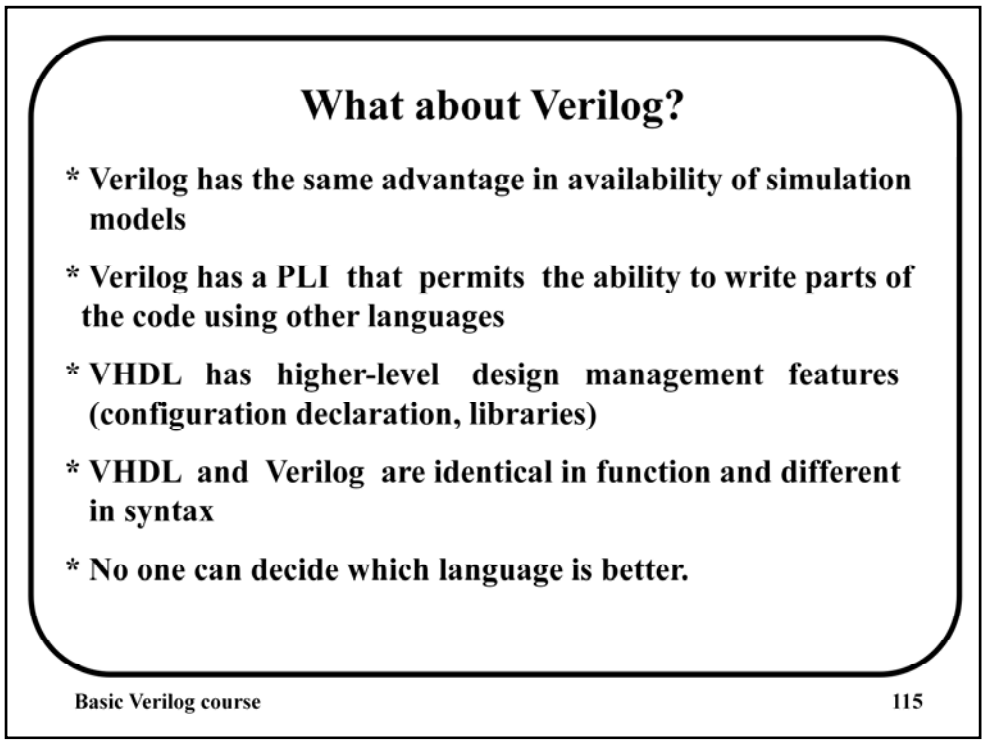
VHDL allows you also to define new data types, and to define your special operators that work on these new types.



* VHDL has different ways for expressing timing information about the design, and the different types of delay. The language has constructs that can express propagation delays, and inertial delays.

* Clocks are defined by the user, since VHDL has no implicit clocking scheme. The language offers constructs that can detect the rising and falling edges of any signal.

* VHDL is not addressing a certain class of devices to which the design is to be mapped. VHDL permits you to create a design without having to first choose a device for implementation. You don’t have to become familiar with a device’s architecture in order to optimize your design . Instead, you can concentrate on creating your design.



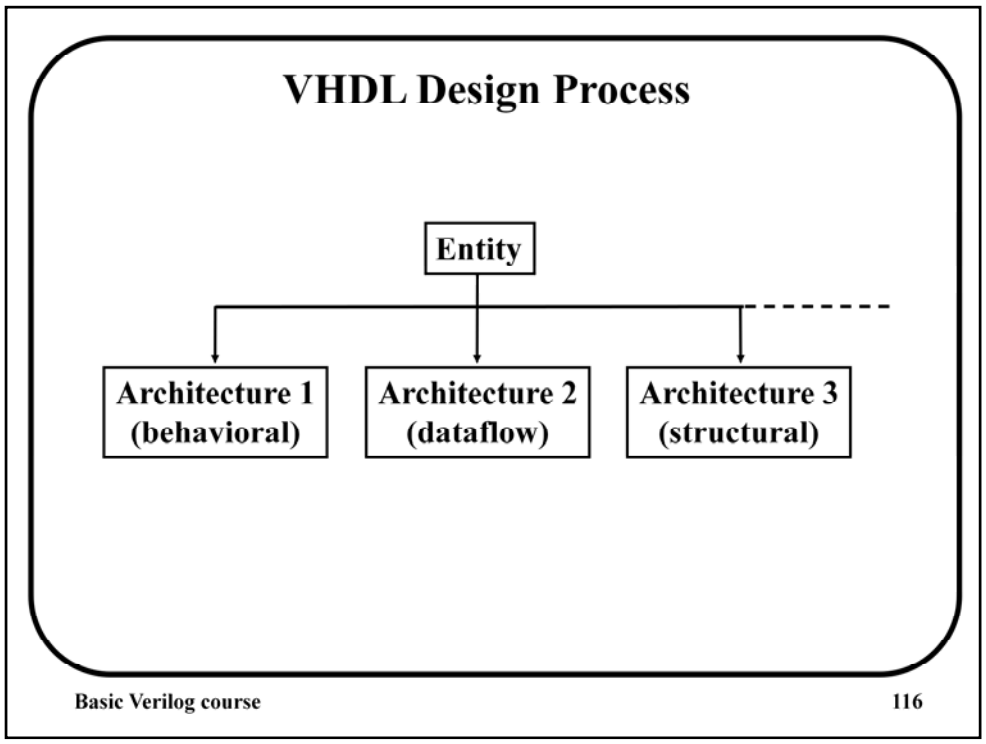
VHDL is not the only language. Verilog HDL has been around for many years as well. How does Verilog differ from VHDL ?

* Verilog has the same advantage in availability of simulation models. Soon after its introduction (in the mid 1980s), Verilog established itself as a standard language for simulation libraries. The wide spread popularity of Verilog has left VHDL somewhat behind. The VITAL (**V**HDL **I**nitiative **T**oward **A**SIC **L**ibraries) initiative enhanced VHDL with a standard method of delay annotation that is similar to that used in Verilog. VITAL made it easier for ASIC vendors to quickly create VHDL models from existing Verilog libraries.

* Another feature of Verilog is a programming language interface (PLI). The PLI makes it possible for simulation model writers to go outside of Verilog when necessary to create faster simulation models or to create functions (using the C language) that would be difficult or inefficient to implement directly in Verilog. On the other hand, a group was formed in 1997 with the aim of designing a procedural interface for VHDL. The interface defines the semantics for a mixed language design and define the access methodology during runtime of foreign models.

* In VHDL's favor is its higher-level design management features. These design management features, which include the configuration declaration and VHDL's library features, make VHDL an ideal language for large projects.

* VHDL and Verilog are identical in function. The syntax is different (with Verilog looking very much like C, and VHDL looking more like Pascal or Ada), but the basic concepts are the same. Both languages are easy to learn, but hard to master.



The basic building blocks of any VHDL design are: the **entity declaration** and the **architecture body**.

•The entity describes the *interface* of the design

•while the architecture describes how the design works.

VHDL allows you to write your designs using various styles of architectures. The styles are behavioral, dataflow, and structural descriptions, or any combination of them. These styles allow you to describe a design at different levels of abstraction, from using algorithms to gate-level primitives.

One entity may have more than one architecture bodies. They must be all equivalent from the functional point of view. A behavioral description may give the specification of the circuit, while another, say structural, description may describe how the circuit is to be implement by interconnecting components.

Entity Declaration

* An entity declaration describes the interface of the component
* PORT clause indicates input and output ports
* An entity can be thought of as a symbol for a component

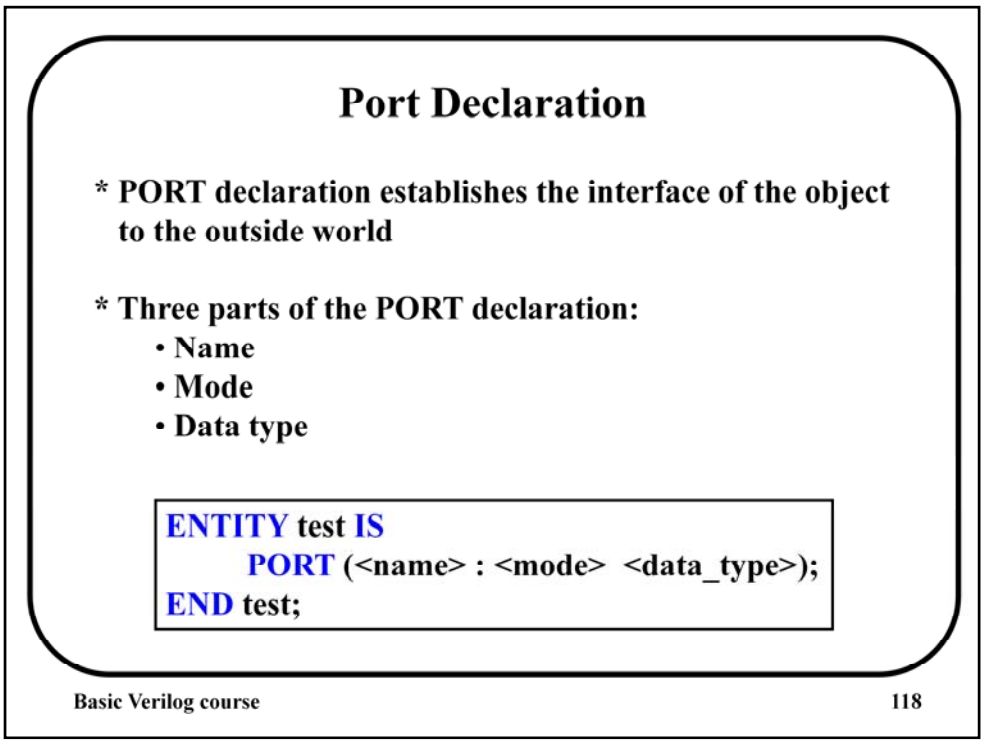
```
ENTITY half_adder IS
    PORT (x, y, enable: IN bit;
          carry, result: OUT bit);
END half_adder;
```

Basic Verilog course 117

* An entity declaration describes the inputs and outputs of a design entity. It can also describe parameterized values as we will see later.

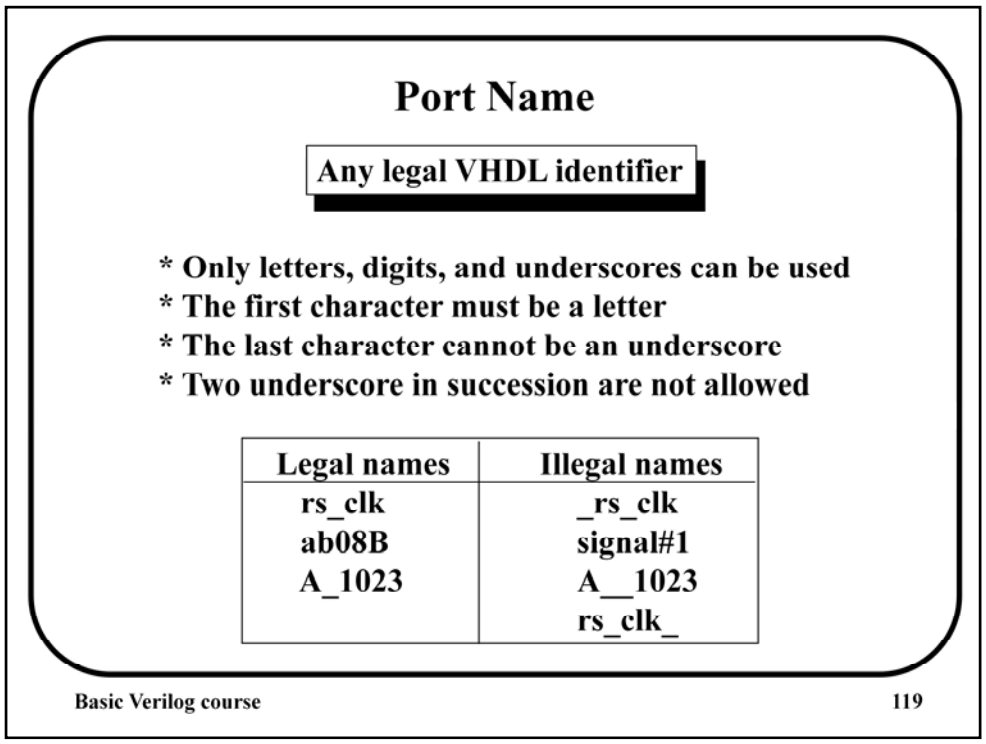
* Each I/O signal in an entity declaration is referred to as a port (which is similar to a pin in a schematic symbol).

* The entity declaration is analogous to a schematic symbol, which describes a component's connections to the rest of the design.



* The set of ports defined for an entity is referred to as a **port declaration**. Each port must have:

- Name
- Mode (direction)
- Type

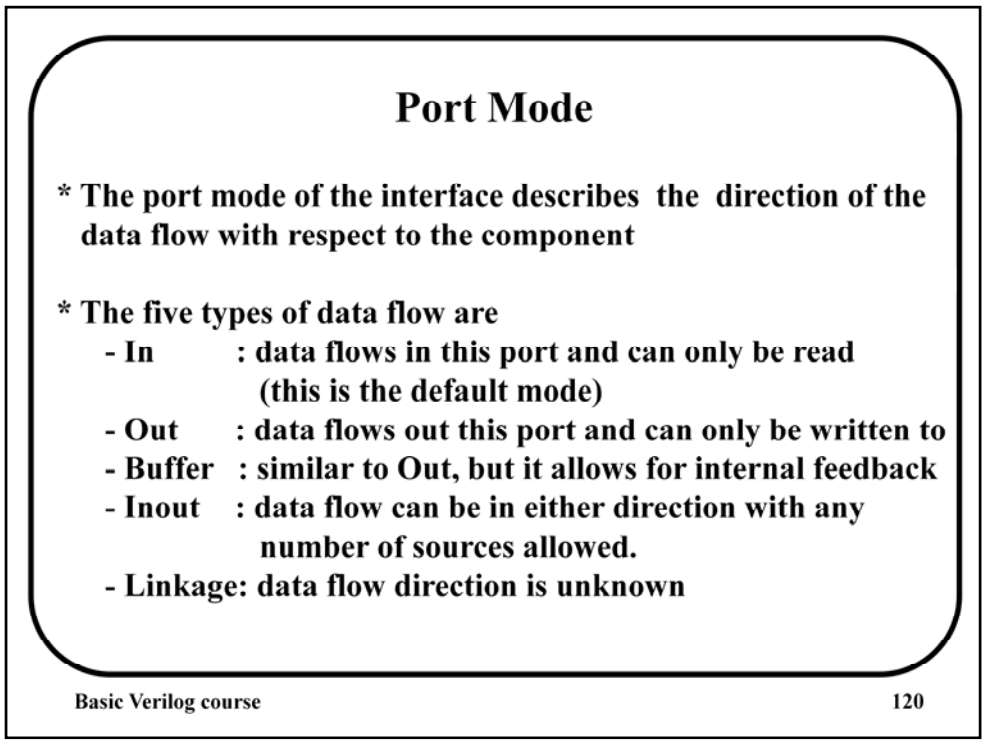


The port name can be any legal VHDL identifier. VHDL identifiers are made up of alphabetic, numeric and/or underscore characters. The following rules apply:

* The first character must be a letter
* The last character cannot be an underscore
* Two underscore in succession are not allowed

VHDL reserved words cannot be used as VHDL identifiers.

Uppercase and lowercase letters are equivalent when used in identifiers



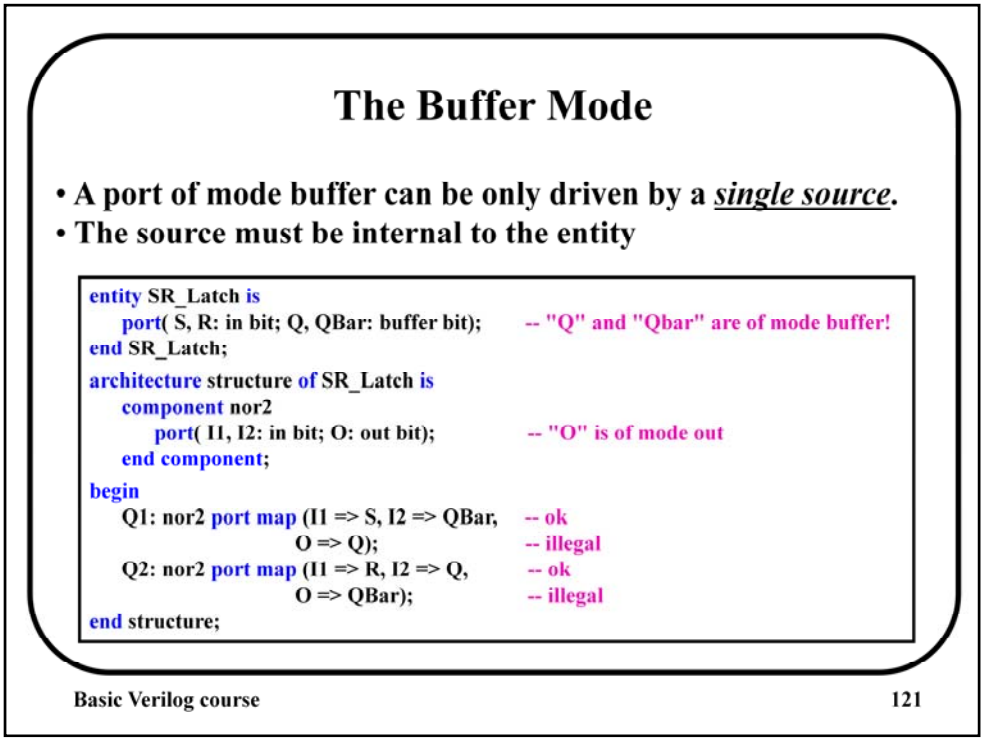
The mode describes the direction in which data is transferred through a port.

If the mode of a port is not specified, the port is of the default mode **in**.

There are 5 different modes:

1. **In**: Data flows only into the entity. The driver for a port of mode **in** is external to the entity.
2. **Out**: Data flows only from its source to the output port of the entity. The driver for a port of mode **out** is inside the entity. Mode **out** does not allow for feedback because such a port is not considered readable within the entity.
3. **Buffer**: A port that is declared as mode **buffer** is similar to a port that is declared as mode out, except that it does allow for internal feedback. Mode buffer is used for ports that must be readable within the entity, such as the counter outputs (the next state of a counter is calculated from the present state).
4. **Inout**: For bi-directional signals, you must declare a port as mode **inout**, which allows data to flow into or out of the entity. In other words, the signal driver can be inside or outside of the entity. Mode **inout** also allows for internal feedback. Mode **inout** is usually used for Data buses.
5. **Linkage**: This mode is used when the data flow direction is unknown.

Mode **inout** can replace any of the other modes. Although using only mode **inout** for all ports would be legal, it would reduce the readability of the code, making it difficult to discern the source of signals.



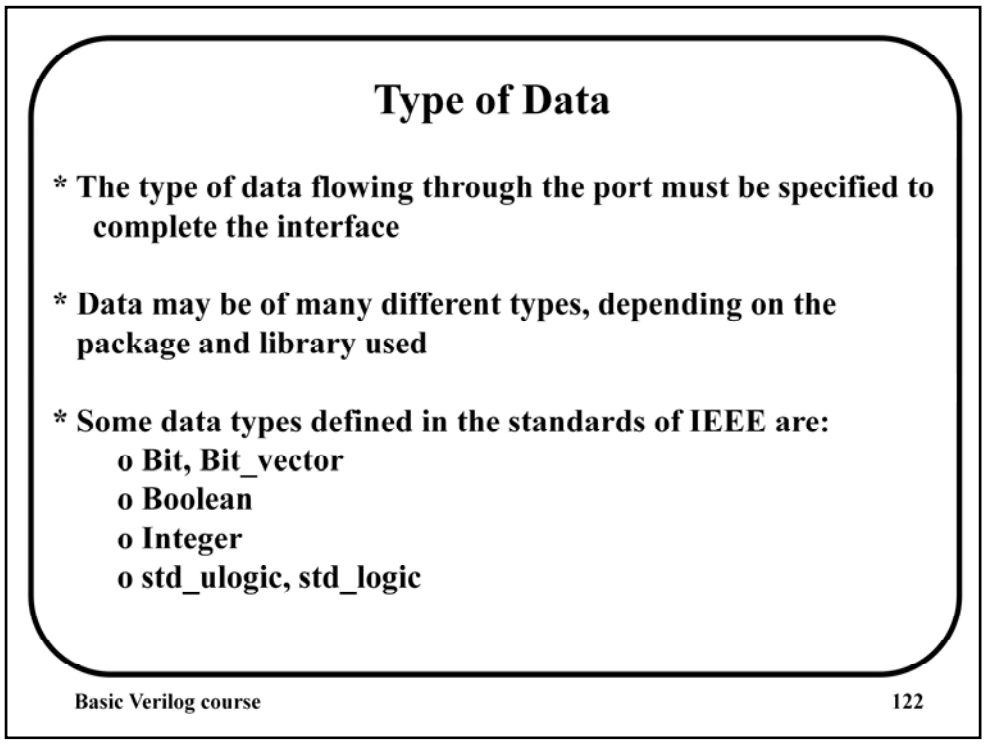
•As we said earlier, ports of mode *buffer* can be both read and written.

•However, there is only a single source allowed to drive any net containing a buffer port, and that source must be internal to the entity in which the port is defined.

•The shown example is illegal in VHDL'87 and VHDL'93 because a port of mode *out* or *inout* must not be connected with a port of mode *buffer*:

•The component instantiation statements in this example are illegal because port "O" of “nor2" is of mode "out" and hence cannot be associated with a buffer port.

• In most situations, the use of buffer ports is discouraged. However, the unnecessary restrictions on buffer ports were removed in VHDL 2000, so that buffer ports are more useful now (if your tool supports VHDL 2000).



In addition to specifying identifiers and modes for ports, you must also declare the data types for ports.

The types provided by the **IEEE 1076/93** standard that are most useful and well-supported and that are applicable to synthesis are:

- Boolean
- Bit
- bit_vector
- integer

The most useful and well-supported types for synthesis provided by the **IEEE std_logic_1164** package are the types std_ulogic and std_logic, and arrays of these types. As the names imply, “standard logic” is intended to be a standard type used to describe circuits for synthesis and simulation.

Architecture Body # 1

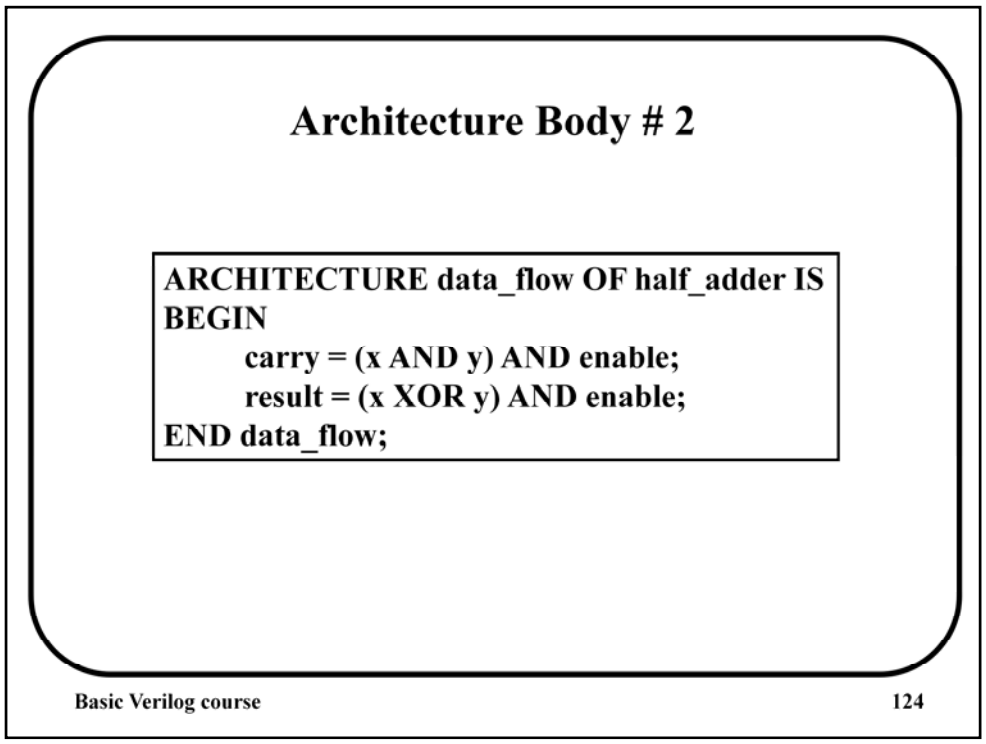
* **Architecture declarations describe the operation of the component**
* **Many architectures may exist for one entity, but only one may be active at a time**

```
ARCHITECTURE behavior1 OF half_adder IS
BEGIN
      PROCESS (enable, x, y)
      BEGIN
        IF (enable = '1') THEN
          result <= x XOR y;
          carry <= x AND y;
        ELSE
          carry <= '0';
          result <= '0';
      END PROCESS;
END behavior1;
```


* Every architecture body is associated with an entity declaration. An architecture describes the contents of an entity; that is, it describes its function. If the entity declaration is viewed as a black box, then the architecture body is the internal view of the black box.

* Here is an architecture body written in a behavioral style. This description is written in an algorithmic way. Behavioral descriptions are sometimes referred to as high-level descriptions because of their resemblance to high-level programming languages. Rather than specifying the structure or netlist of a circuit, you specify a set of statements that, when executed in sequence, model the function or behavior of the entity.

* The advantage of high-level descriptions is that you don’t need to focus on the gate-level implementation of the design; instead, you can focus your efforts on accurately modeling its function.



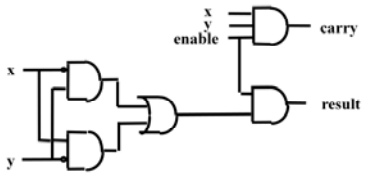
* This is dataflow architecture because it specifies how data will be transferred from signal to signal and input to output without the use of sequential statements.

* The major difference between behavioral and dataflow descriptions is that one uses processes (which is used for the **sequential** execution of statements), and the other does not (and thus the statements are executed **concurrently**).

Architecture Body # 3

* To make the structural architecture, we need first to define the gates to be used.

* In the shown example, we need NOT, AND, and OR gates



Basic Verilog course 125

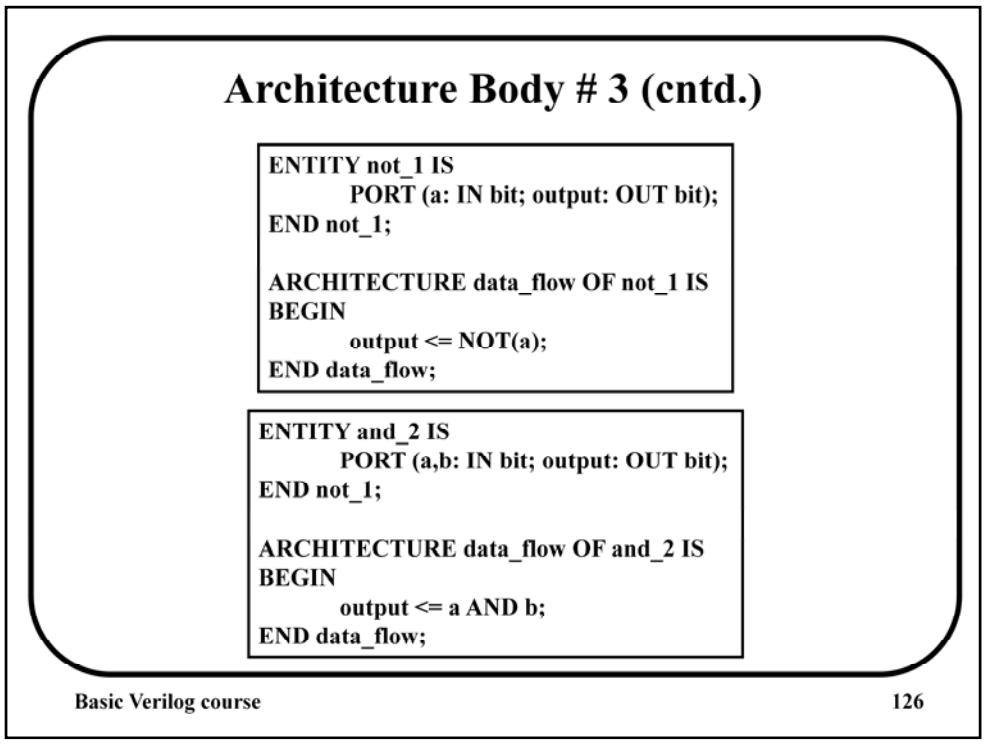
* Structural VHDL describes the arrangement and interconnection of components

* Structural descriptions support the use of predefined components

* Structural descriptions may connect simple gates or complex, abstract components

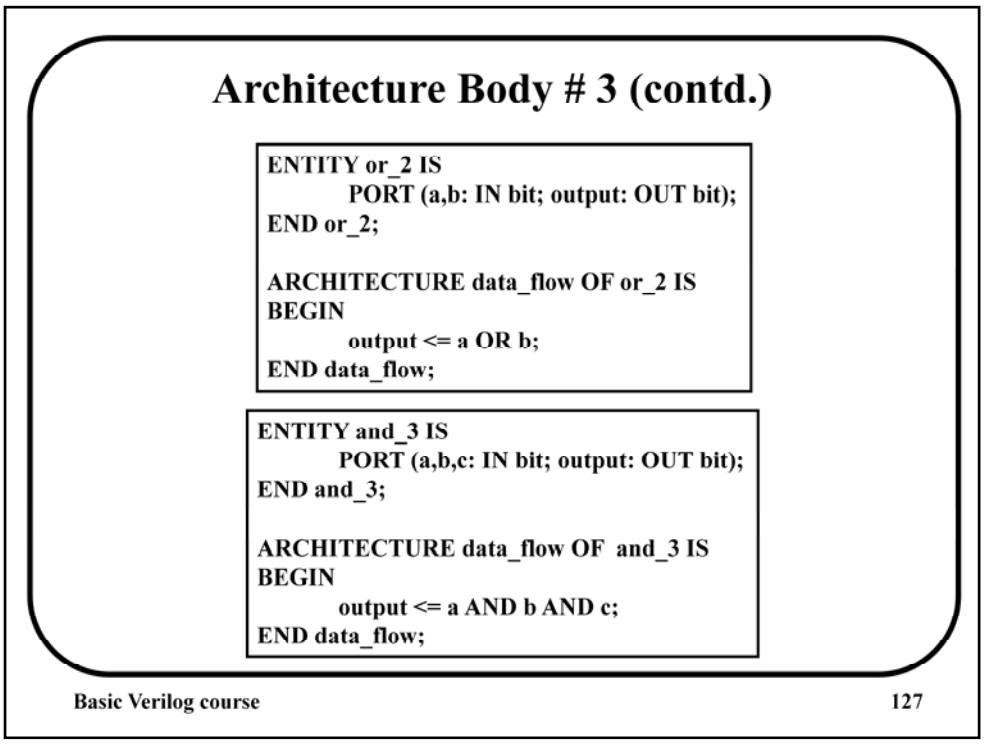
* To make a structural description, you must know the components that you are going to use in the design. The above design requires *not_1*, *and_2*, *and_3*, and *or_2* gates.

* We will start by writing the VHDL descriptions for these gates.



* The VHDL description of any gate is like the VHDL description of any design entity. It has an entity declaration and an architecture body.

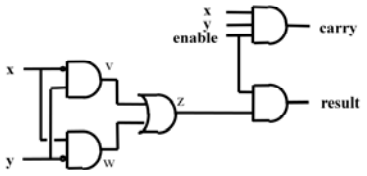
* The VHDL code shows a dataflow description of the *not_1* gate and the *and_2* gate.



* Here is the dataflow description of the components *or_2*, and *and_3*.

Architecture Body # 3 (contd.)

```
ARCHITECTURE structural OF half_adder IS
  COMPONENT and2 PORT(a,b: IN bit; output: OUT bit); END COMPONENT;
  COMPONENT and3 PORT(a,b,c: IN bit; output: OUT bit); END COMPONENT;
  COMPONENT or2 PORT(a,b: IN bit; output: OUT bit); END COMPONENT;
  COMPONENT not1 PORT(a: IN bit; output: OUT bit); END COMPONENT;

  FOR ALL: and2 USE ENTITY work.and_2(dataflow);
  FOR ALL: and3 USE ENTITY work.and_3(dataflow);
  FOR ALL: or2 USE ENTITY work.or_2(dataflow);
  FOR ALL: not1 USE ENTITY work.not_2(dataflow);

  SIGNAL v,w,z,nx,nz: BIT;

BEGIN
  c1: not1 PORT MAP (x,nx);
  c2: not1 PORT MAP (y,ny);
  c3: and2 PORT MAP (nx,y,v);
  c4: and2 PORT MAP (x,ny,w);
  c5: or2 PORT MAP (v,w,z);
  c6: and2 PORT MAP (enable,z,result);
  c7: and3 PORT MAP (x,y,enable,carry);
END structural;
```



Before writing this description, the component descriptions must be already compiled and stored in a library. The default library is called *work*.

When we investigate this VHDL code we can distinguish 4 different zones.

1. The first zone is called the **component declaration zone**, in which the components to be used in the design are declared. We will use some component called and2 that has 2 inputs called (a,b)and 1 output called (output). We will use another component called and3 and a third component called o2 and a forth component called not1. Here we declared only the interface of the components to be used. But what does and2 do ?? What does or2 do ?? Till now we don't know. It is the role of the second zone of the VHDL code.

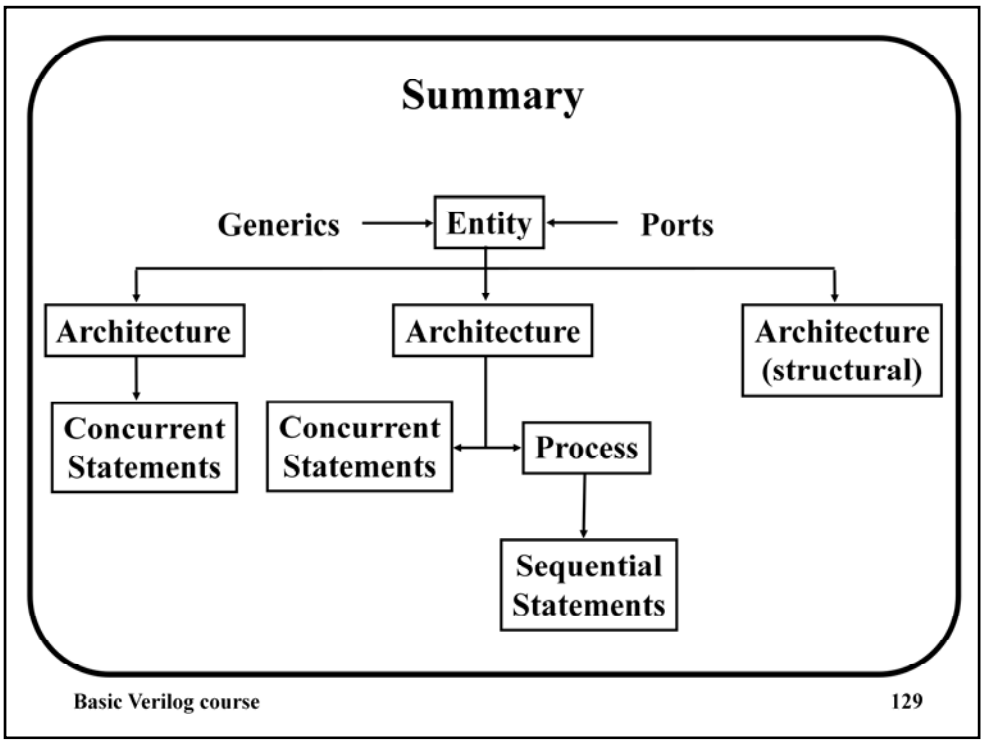
2. The second zone of the program is called the **configuration zone**. In this zone we specify where to find the complete description of the components declared in the first part. The line

FOR ALL: and2 USE ENTITY work.and_2(dataflow);

means that: anywhere in the program when you find something called and2, you must look for its description in a library called ***work***, under the entity called ***and_2*** and the architecture ***dataflow***.

3. The third zone is called the **signal declaration** zone, in which we declare all the internal signals that are to be used. For example the signal *v* is an internal signal. To use it we must declare it first.

4. This part of the design is called the **component instantiation** zone, in which we instantiate components to describe the gates of the design and their interconnects. Note that each line has a unique label (in the above example labels are named C1, C2, … C7). Labels are important for referring to the different instances.



The basic building blocks of any VHDL design are the **entity declaration** and the **architecture bodies**.

An entity gives a black box description of the design. It describes how the entity is connected to the outside world. It defines the ports of the entity, which may be of **mode** in, out, buffer, inout, or linkage, as well as the **types of data** to be transmitted on these ports.

An entity may have **more than one architecture**, but only one of them may be active at a time.

There are different styles of writing VHDL architecture bodies:

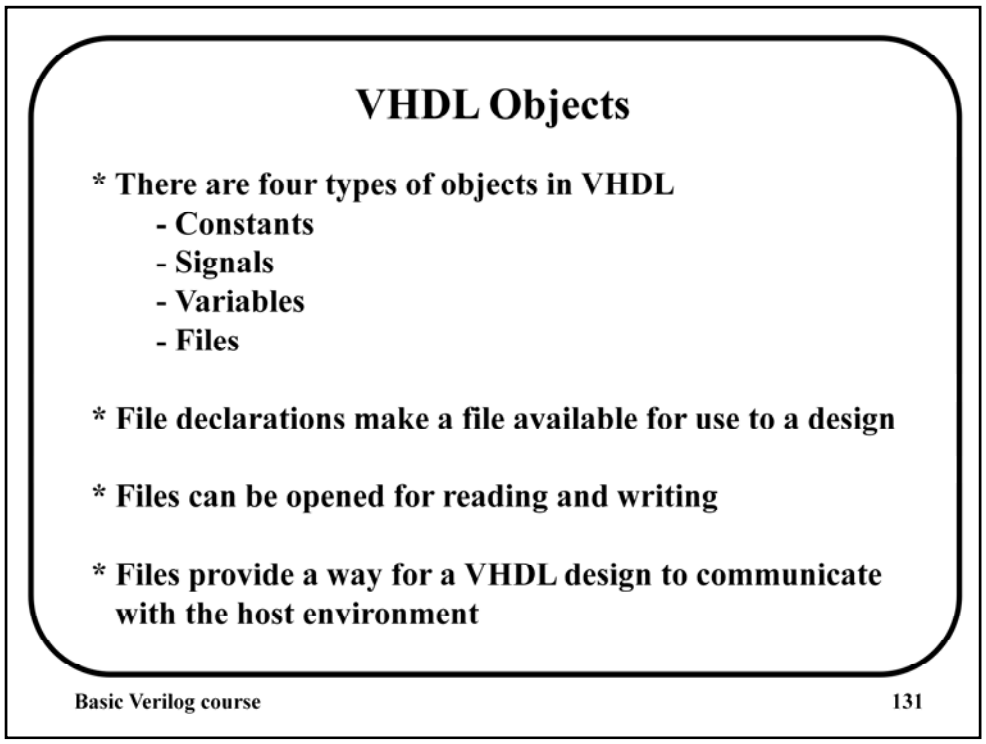
* Behavioral descriptions, that contain sequential statements written within a process. Behavioral descriptions may also contain concurrent statements.

* Dataflow descriptions, that describes how the data flows through the design. Data flow descriptions contain only concurrent statements. They don't have sequential statements.

* Structural descriptions, that gives the list of gates (components) used in the design, and how these gates are interconnected.

VHDL BASICS

* Data Objects
* Data Types
* Types and Subtypes
* Attributes
* Sequential and Concurrent Statements
* Procedures and Functions
* Packages and Libraries
* Generics
* Delay Types

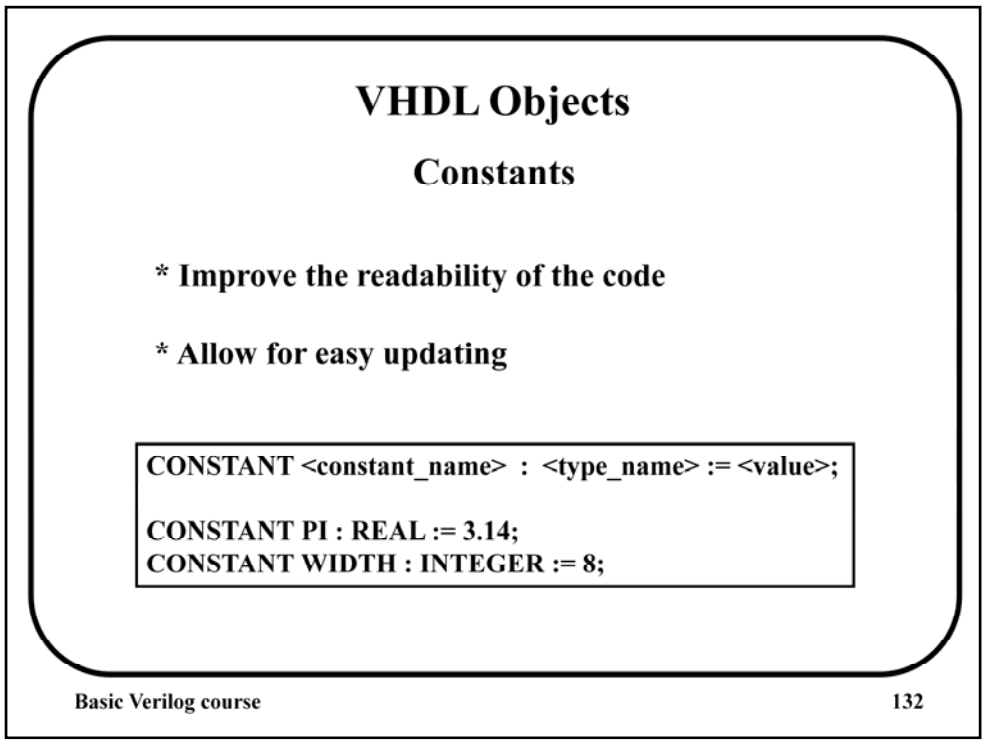


* Data objects hold values of specified types. They belong to one of four classes:

1. Constants
2. Signals
3. Variables
4. Files

* Data objects must be declared before they are used.

* Files contain values of a specified type. Files may be used to read in stimulus, and write out the results when using test benches.



* Constants hold values that cannot be changed within the design.

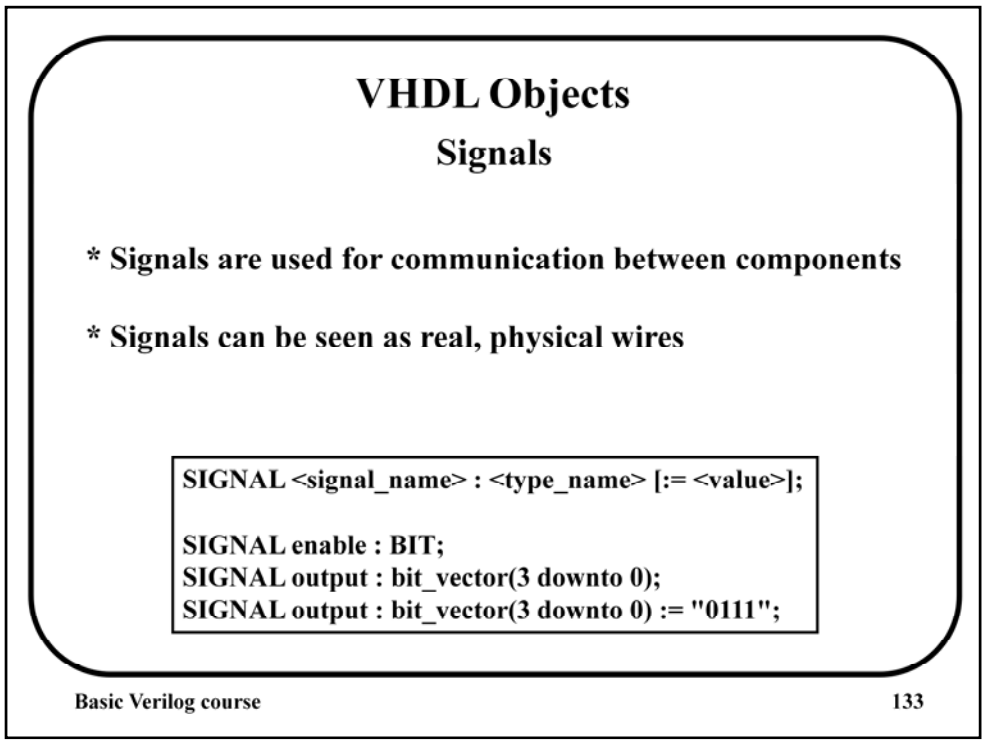
* Constants are generally used to improve the readability of code, and may also make it easier to modify code: instead of changing a value each place that it appears, you need to change only the value of the constant.

* Example:

In the statement:

CONSTANT WIDTH: INTEGER := 8;

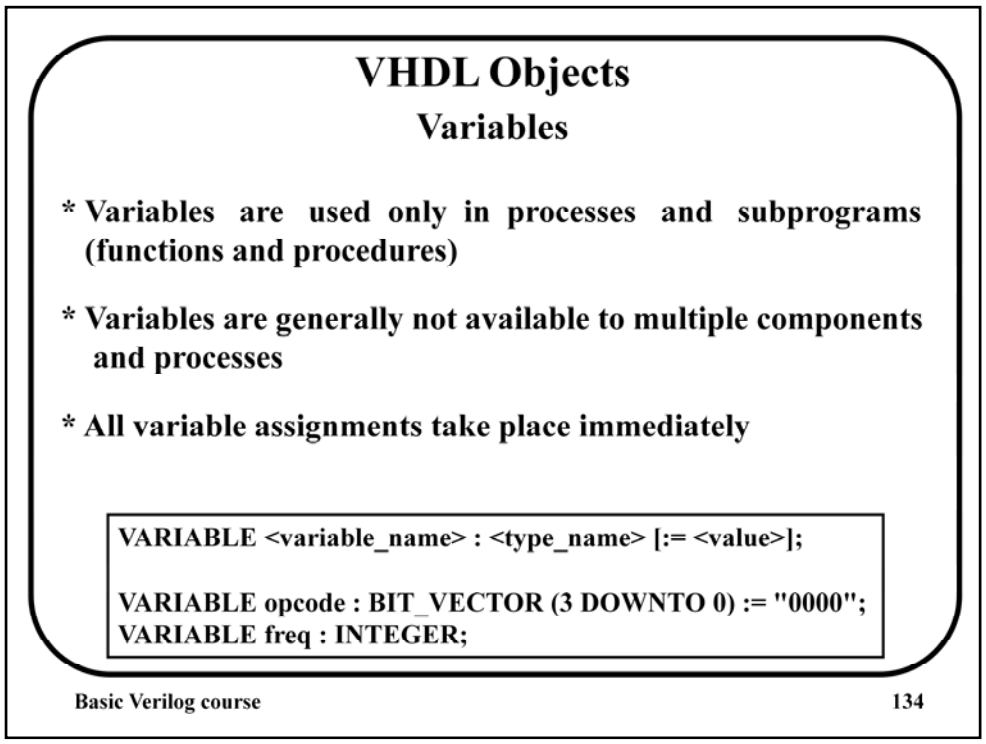
The identifier WIDTH may express the width of a register, and it may be used several times in the code to describe a circuit. If at any time you need to make another description using a 16-bit register, all what you need to do is to change the above statement.



* Signals can represent wires, and can therefore interconnect components.

* Signals cal also represent the state of memory elements: In the above example "output" may represent the current state of a counter (the memory elements of the counter, or the wires attached to the outputs of those memory elements).

* Initial values may be assigned to signals but are rarely meaningful for synthesis. It is a misconception to believe that when you assign an initial value to a signal, the memory elements will power-up in that initialized state. Initial values are only useful for simulation purposes.



* Variables are used only in processes and subprograms, and thus they must be declared in the declarative region of a process or subprogram.

* Unlike signals, variables do not represent wires or memory elements.

* Variables can be used in writing high-level simulation models.

* For synthesis, they may be used for computational purposes, but variable synthesis is not well defined, so they are usually avoided. They are used only as index holders for the for-loops.

* Variable assignments are immediate, not schedules. The variable assignment and initialization symbol ":=" indicates immediate assignment.

Signals versus Variables

* A key difference between variables and signals is the assignment delay

```
ARCHITECTURE signals OF test IS
   SIGNAL a, b, c, out_1, out_2 : BIT;
BEGIN
   PROCESS (a, b, c)
   BEGIN
     out_1 <= a NAND b;
     out_2 <= out_1 XOR c;
   END PROCESS;
END signals;
```

Time	a b c	out_1	out_2
0	0 1 1	1	0
1	1 1 1	1	0
1+d	1 1 1	0	0

Signals versus Variables (cont. 1)

```
ARCHITECTURE variables OF test IS
   SIGNAL a, b, c: BIT;
   VARIABLE out_3, out_4 : BIT;
BEGIN
   PROCESS (a, b, c)
   BEGIN
     out_3 := a NAND b;
     out_4 := out_3 XOR c;
   END PROCESS;
END variables;
```

Time	a b c	out_3	out_4
0	0 1 1	1	0
1	1 1 1	0	1

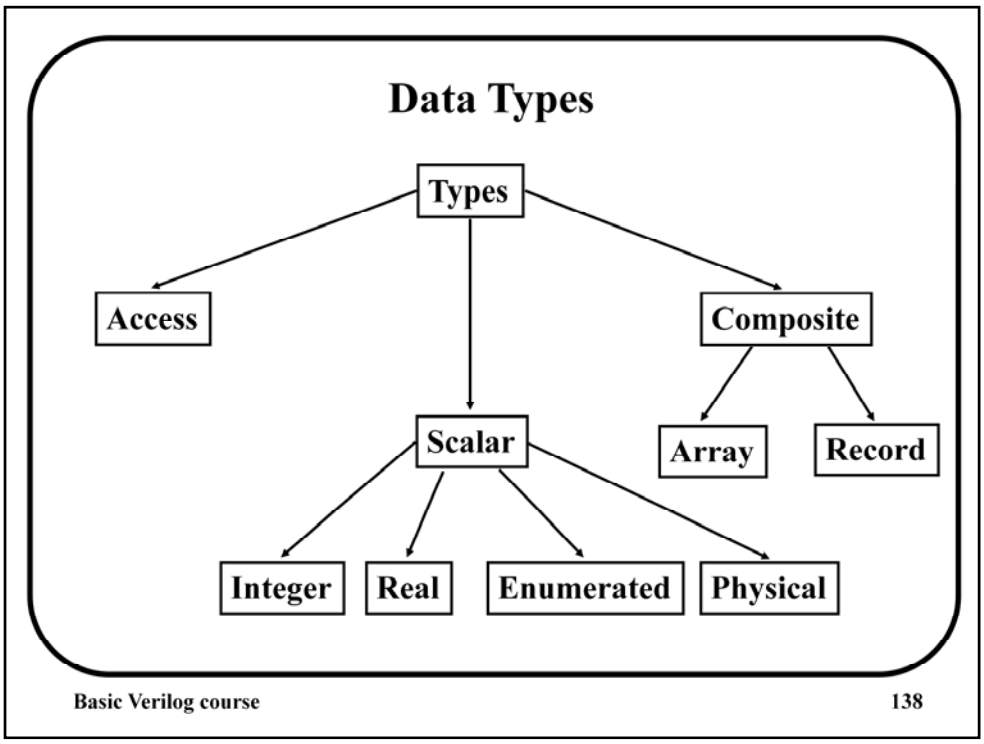
VHDL Objects

Scoping Rules

* VHDL limits the visibility of the objects, depending on where they are declared

* The scope of the object is as follows
  - o Objects declared in a package are global to all entities that use that package
  - o Objects declared in an entity are global to all architectures that use that entity
  - o Objects declared in an architecture are available to all statements in that architecture
  - o Objects declared in a process are available to only that process

* Scoping rules apply to constants, variables, signals and files



* We will discuss the categories of data types that are most useful for synthesis, **scalar**, and **composite** types. We will not discuss **access** types in full details.

* VHDL is a strongly typed language. Data objects of different base types cannot be assigned to one another without the use of a type-conversion function. A base type is either a type itself or a type assigned to a subtype.

* An assignment like the following one where "a" and "b" are integers, will produce an error.

```
a <= b + '1';
```

The reason is that '1' is a bit and cannot be used the "+" operator to be added to integers, unless the operator is **overloaded**.

* Scalar types have an order, which allows relational operators to be used with them. There are four categories of scalar types: enumeration, integer, real, and physical types.

* Composite types can hold multiple values at a time, contrary to scalar types that can hold only one value at a time. Composite types consist of **array** types and **record** types.

Scalar Types

* **Integer Types**
  - **Minimum range for any implementation as defined by standard: -2,147,483,647 to 2,147,483,647**

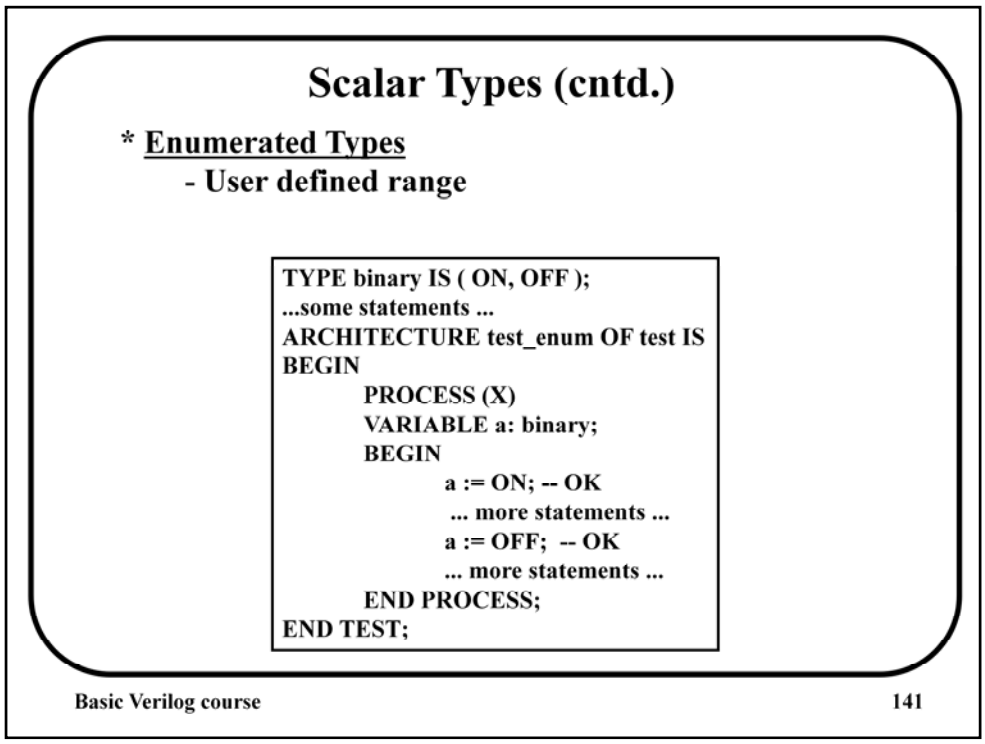
```
ARCHITECTURE test_int OF test IS
BEGIN
        PROCESS (X)
        VARIABLE a: INTEGER;
        BEGIN
                a := 1;  -- OK
                a := -1;  -- OK
                a := 1.0;  -- bad
        END PROCESS;
END TEST;
```


* VHDL supports integers in the range from -2,147,483,647 (i.e. $-(2^{31}-1)$) to 2,147,483,647 (i.e. $2^{31}-1$).

Scalar Types (cntd.)

* **Real Types**
  - **Minimum range for any implementation as defined by standard: -1.0E38 to 1.0E38**

```
ARCHITECTURE test_real OF test IS
BEGIN
        PROCESS (X)
        VARIABLE a: REAL;
        BEGIN
                a := 1.3;  -- OK
                a := -7.5;  -- OK
                a := 1;  -- bad
                a := 1.7E13;  -- OK
                a := 5.3 ns;  -- bad
        END PROCESS;
END TEST;
```



* An enumeration type is a list of values that an object of that type may have.

* Enumeration types are often defined for state machines:

type states is (idle, ready, jam, error);

* As a scalar type, an enumeration type is ordered. The order in which the values are listed in the type declaration defines their relation. The left most value is less than all the other values. Each value is greater that the one to its left. Thus in the above example, *idle* is less than *ready*. *jam* is greater than *ready* and less than *error*.

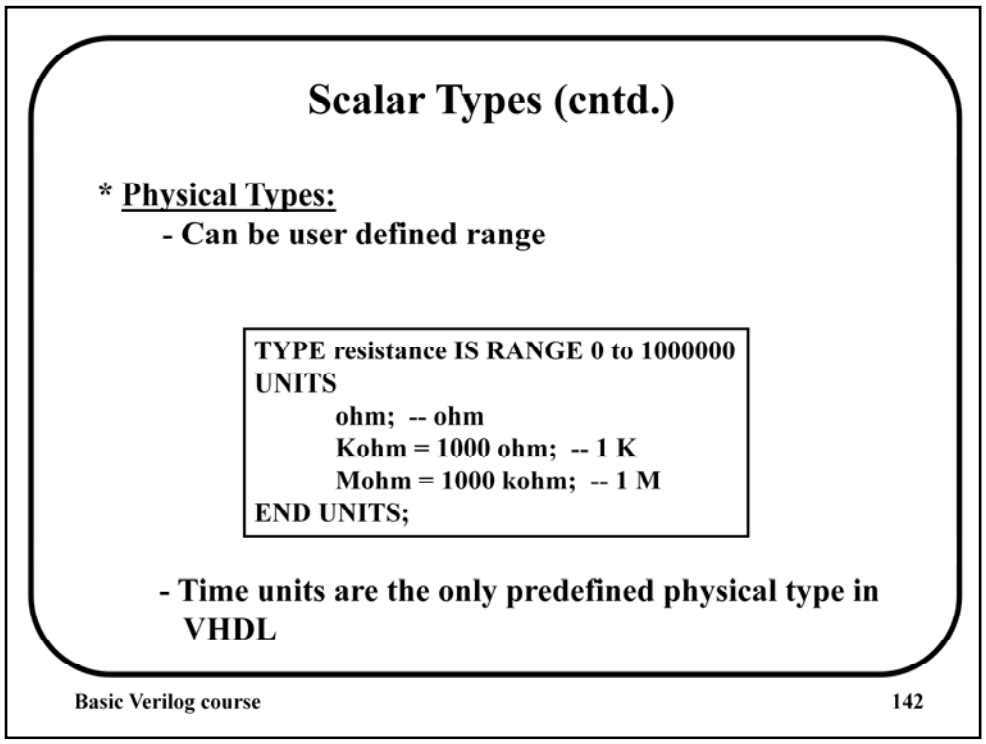
* There are two enumeration types predefined by the IEEE 1076 standard: bit and Boolean.

type boolean **is** (FALSE, TRUE);

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

* The IEEE 1164 standard defines an additional type std_ulogic. This type defines a 9-value logic system, and it is declared as follows:

```
type std_ulogic is ('U',   -- Uninitialized
                    'X',   -- Forcing Unknown
                    '0',   -- Forcing 0
                    '1',   -- Forcing 1
                    'Z',   -- High Impedance
                    'W',   -- Weak Unknown
                    'L',   -- Weak 0
                    'H',   -- Weak 1
                    '-',   -- Don't care
                   );
```



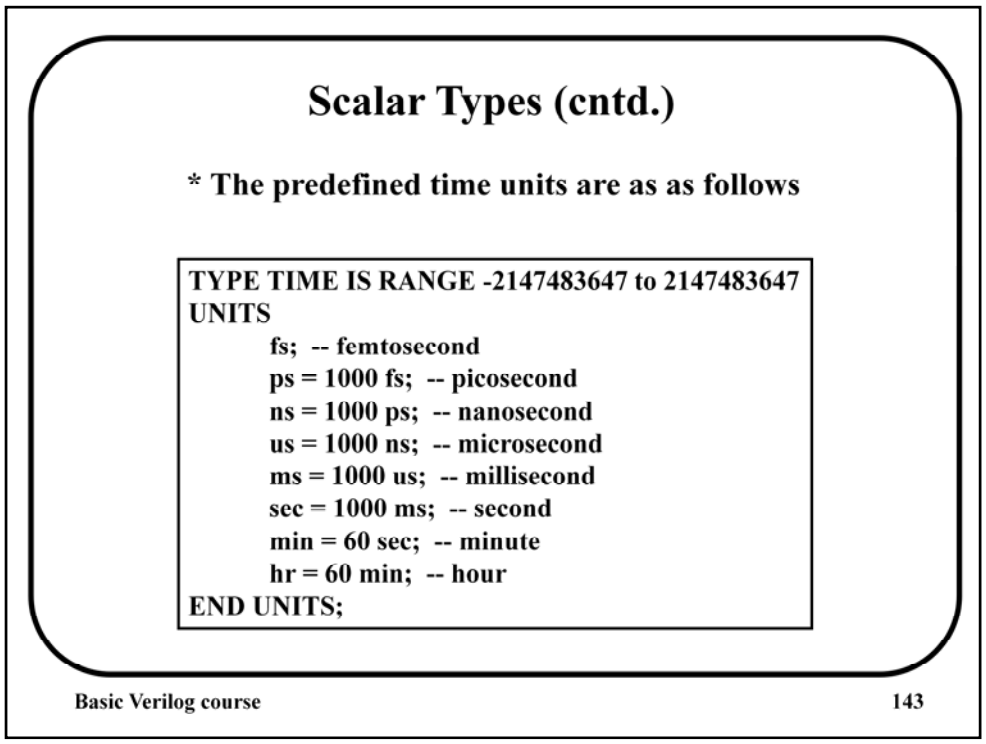
* Physical type values are used as measurement units.

* There will be always a **primary unit** (for example ohm), and multiples of it (e.g Kohm).

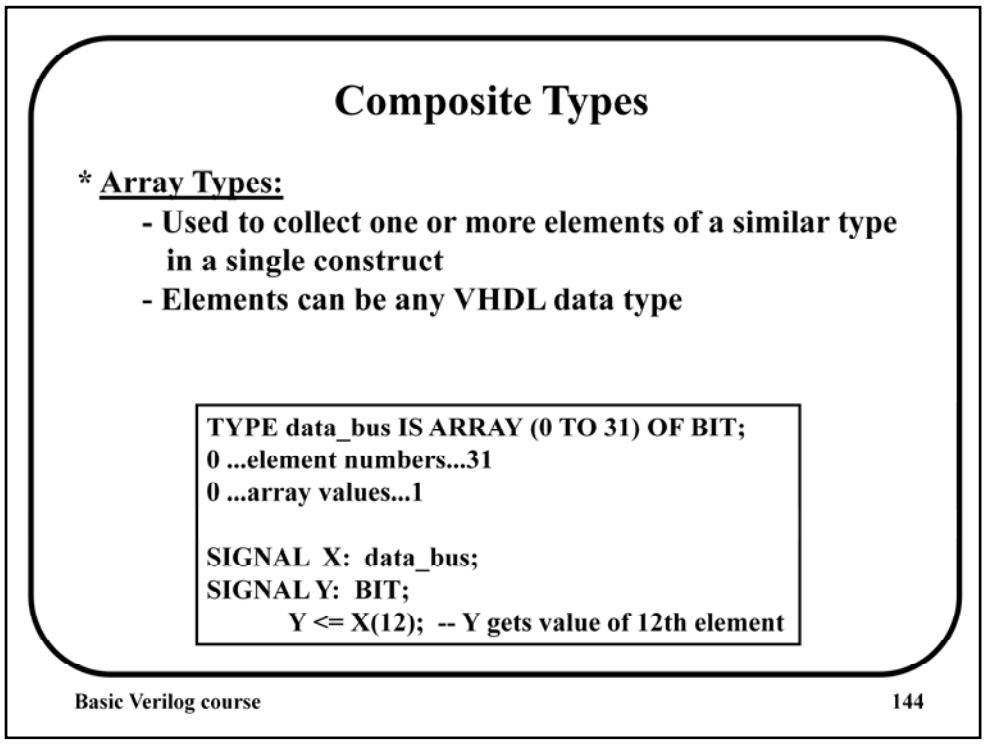
* Physical types have a range which must be defined by the user.

* Physical types do not carry meaning for synthesis, but they are used frequently in simulation.

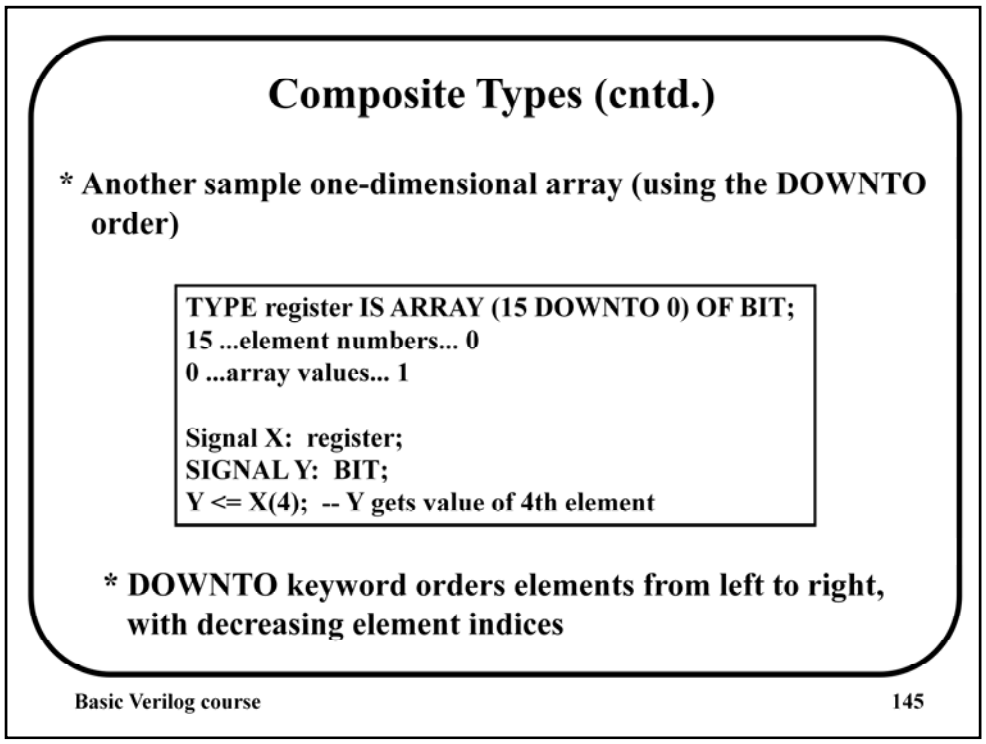
* Physical types have very little to do with logic design, but may bring to mind some interesting ideas of how VHDL can be used to simulate models of systems that do not represent logic circuits but some other types of systems.



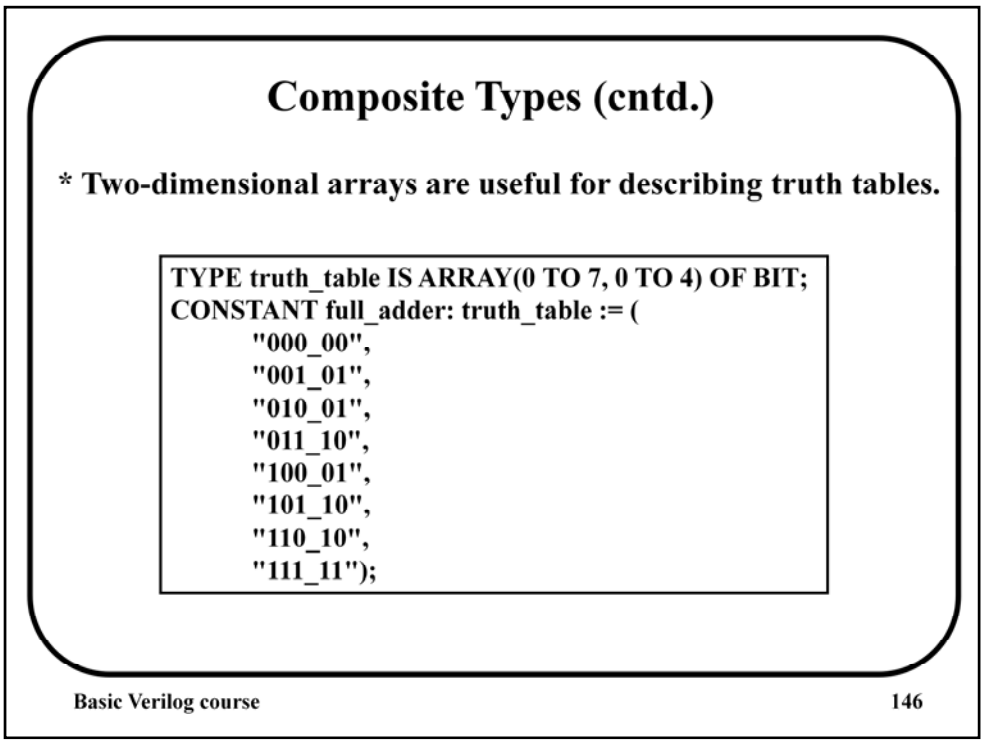
* Time is the only predefined physical type. Its range includes the range of integers. Its primary unit is fs (femtosecond).



* An object of an array type consists of multiple elements of the same type.
* Array types are commonly used for buses as shown above.

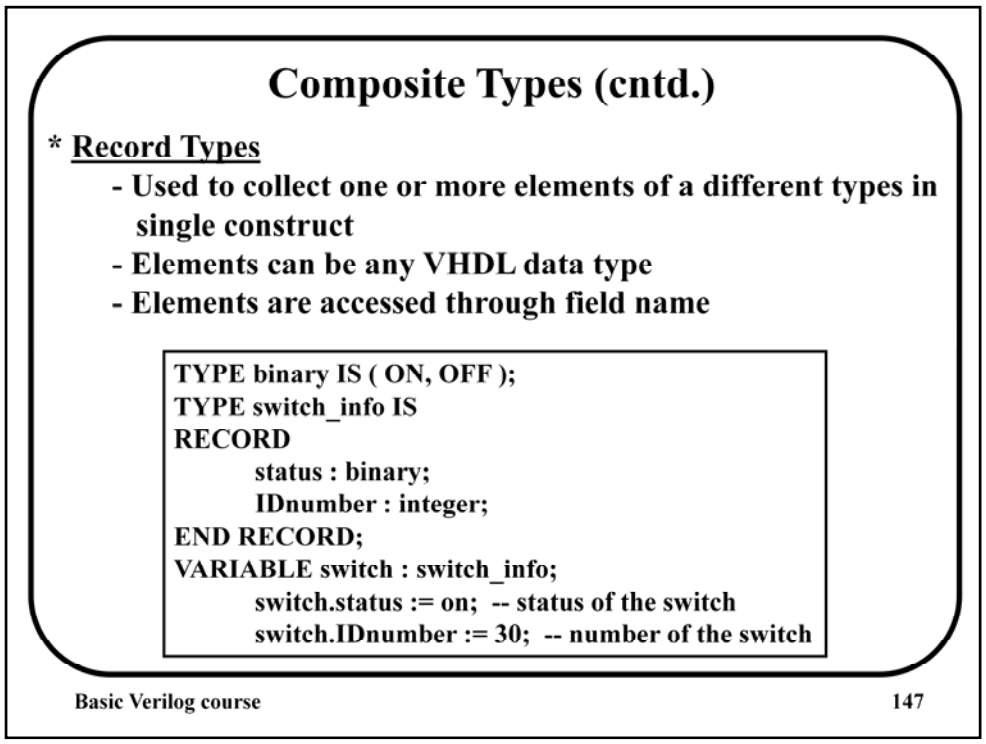


* The range of the array can also be defined using the **DOWNTO** keyword. It orders the elements from left to right.



* Arrays can be of any dimensions. Two-dimensional arrays are usually used for representing truth tables.

* An underline character is inserted to distinguish between the input and output sides of the table. An underline character can be inserted between any two adjoining digits.



* An object of a record type has multiple elements of different types.
* Individual fields of a record can be referenced by element name.

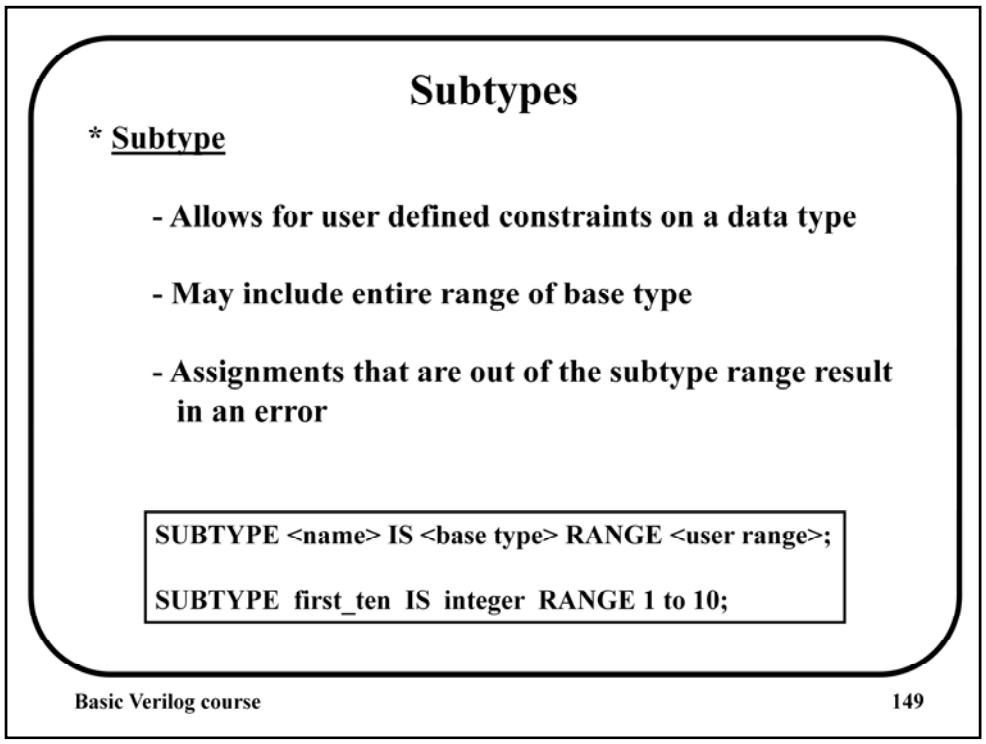
Access Types

* Access

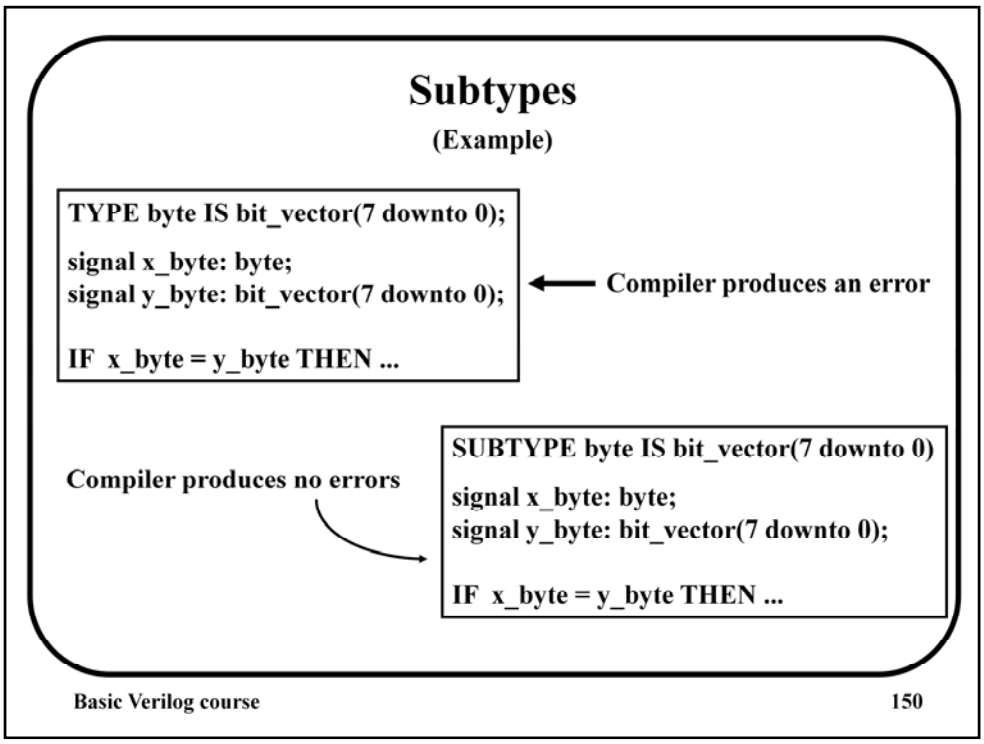
  - Similar to pointers in other languages

  - Allows for dynamic allocation of storage

  - Useful for implementing queues, fifos, etc.



* A subtype is a type with constraint. Subtypes are mostly used to define objects of a base type with a constraint.



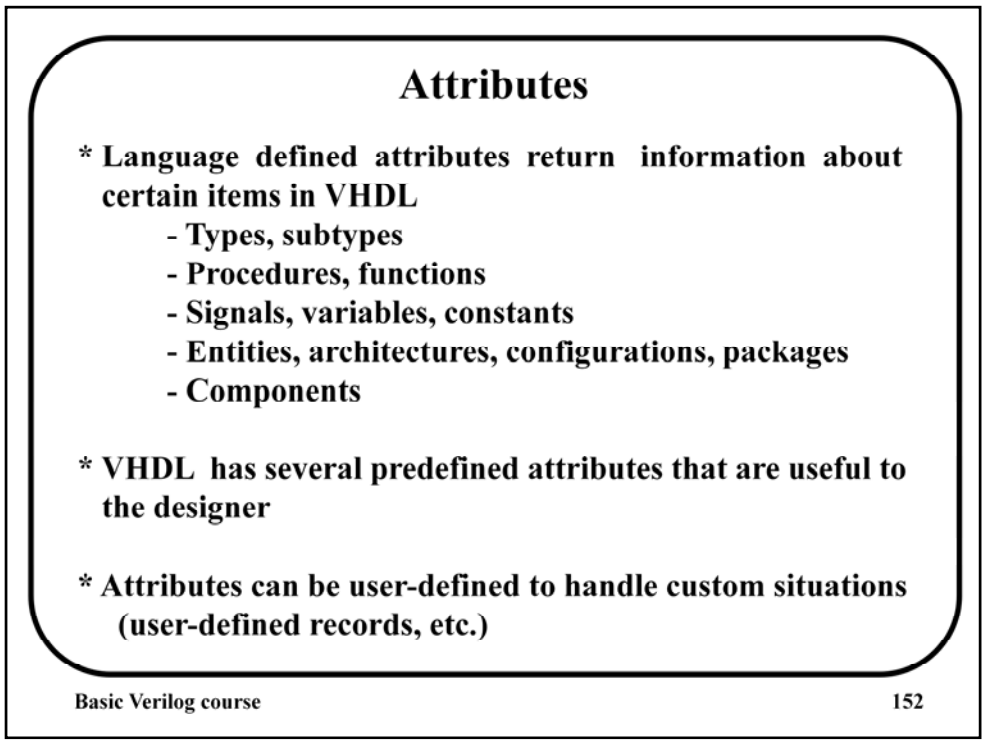
* In the first case, although *byte* is based on the bit_vector type, it is considered as an independent type. Type checking rules require that operands in the comparison statement be of the same type. The operands of the comparison statement in the first example are of types *byte*, and *bit_vector*, resulting in a type mismatch.

* A subtype is a type with a constraint. So in the second case, *byte* is not an independent type. It is related to its base type which is *bit_vector*. The operands of the comparison statement are of type *bit_vector*, and the code will not result in a compile-time error (because the base types are the same).

* Whereas different types will often produce compile-time errors for type mismatches, subtypes of the same base type can be interchanged.

Summary

* VHDL has several different data types available to the designer
* Enumerated types are user defined
* Physical types represent physical quantities
* Arrays contain a number of elements of the same type or subtypes
* Records may contain a number of elements of different types or subtypes
* Access types are basically pointers
* Subtypes are user defined restrictions on the base type



* An attribute provides information about items such as entities, architectures, types, and signals.

* There are several value, signal, and range attributes.

    * Scalar types have value attributes. The value attributes are **'left**, **'right**, **'high**, **'low**, and **'length** (this is pronounced as tick-left, tick-right ... etc).

    * Important signal attributes include the **'event, 'stable, and 'last_value** attributes.

    * A useful range attribute is the **'range** attribute

Attributes

(Signal Attributes)

* General form of attribute use is:

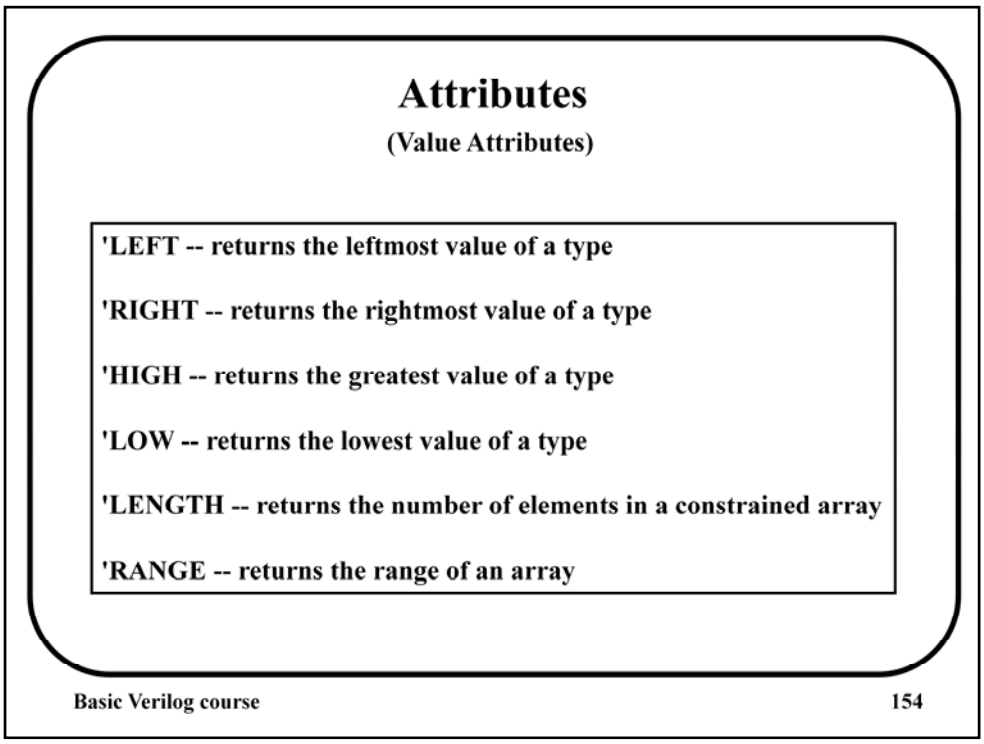
<name> ' <attribute_identifier>

* Some examples of signal attributes

X'EVENT -- evaluates TRUE when an event on signal X has just
-- occurred.

X'LAST_VALUE -- returns the last value of signal X

X'STABLE(t) -- evaluates TRUE when no event has occurred on
-- signal X in the past t" time



* The attribute **'left** returns the leftmost value of type, and **'right** returns the rightmost value.

* The attribute **'high** returns the greatest value of a type.

  - For enumerated types, **'high** is the same as **'right**.

  - For integer ranges, **'high** returns the greatest integer in the range

  - For other ranges **'high** returns the value to the right of the keyword *to*, or the value to the left of the keyword *downto*.

* The attribute **'low**, is exactly the inverse of the attribute **'high**.

* The attribute **'length** returns the number of elements in a constrained array.

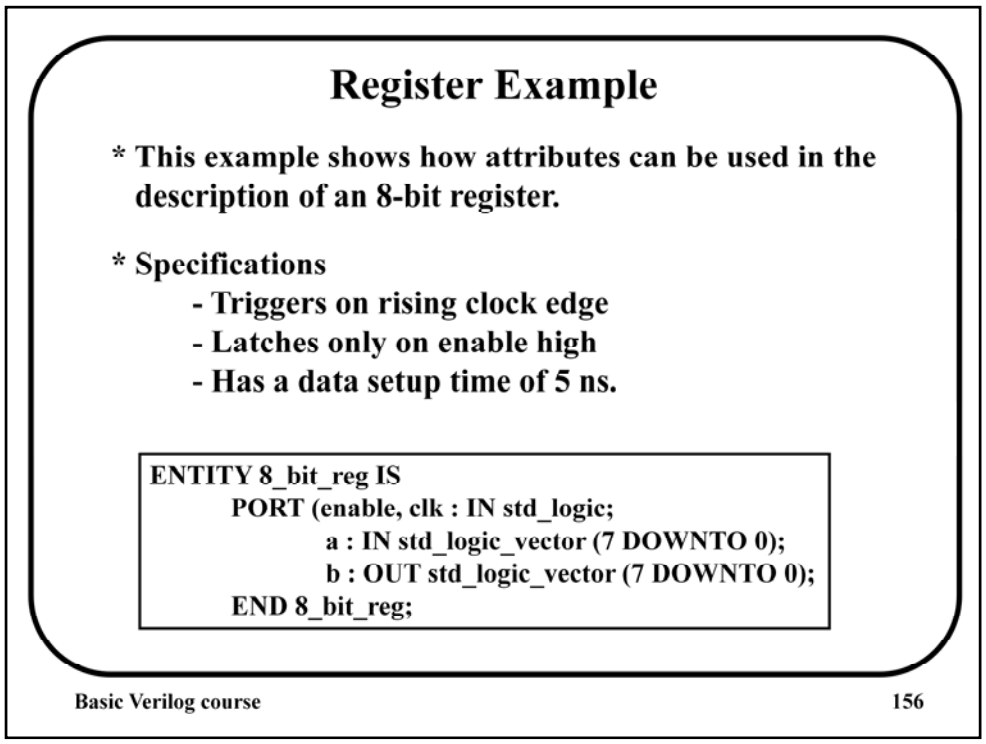
Attributes

(Example)

```
TYPE count is RANGE 0 TO 127;
TYPE states IS (idle, decision,read,write);
TYPE word IS ARRAY(15 DOWNTO 0) OF bit;
```

count'left = 0	states'left = idle	word'left = 15
count'right = 127	states'right = write	word'right = 0
count'high = 127	states'high = write	word'high = 15
count'low = 0	states'low = idle	word'low = 0
count'length = 128	states'length = 4	word'length = 16

```
count'range = 0 TO 127
word'range = 15 DOWNTO 0
```



* This example will help us understanding how attributes can be used in describing circuits.

* We will write a description of an 8-bit register, that triggers on the rising edge of the clock provided that an enable signal is high. The register has a setup time of 5 ns (i.e the data must stable on the input lines of the register for a period of time equal to 5 ns before the rising clock edge).

* The entity of the register will have three inputs:the clock *clk*, the *enable* line, which are of type std_logic, and a vector of input data lines called *a*. The vector of outputs is called *b*.

Register Example (contd.)

* A signal having the type std_logic may assume the values: 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', or '-'
* The use of 'STABLE detects for setup violations

```
ARCHITECTURE first_attempt OF 8_bit_reg IS
        BEGIN
            PROCESS (clk)
            BEGIN
              IF (enable = '1') AND a'STABLE(5 ns) AND
                 (clk = '1') THEN
                    b <= a;
              END IF;
            END PROCESS;
        END first_attempt;
```

* What happens if clk was 'X'?

Basic Verilog course 157

* This process will be executed only when an event occurs on the clk.

* the condition clk = '1' makes sure that after the event on the clock, its value is '1'.

* The condition a'STABLE(5 ns) checks for the setup time.

* The condition enable ='1' check for the enable line.

* If all of the above conditions are true, then the statement b <= a, will be executed.

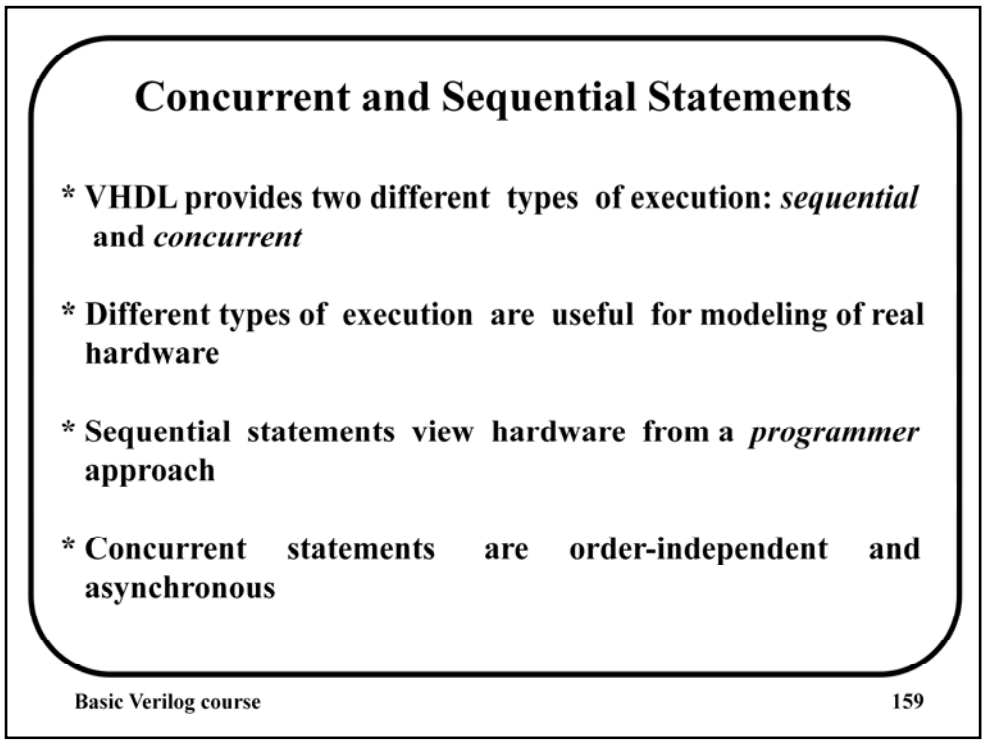
* If we examine the above code thoroughly, we find that the register may trigger due to a transition from 'X' to '1' on the clock, which is not required. So the above description may be modified as in the following description.

Register Example (contd.)

* The use of 'LAST_VALUE ensures the clock is rising from a 0 value

```
ARCHITECTURE behavior OF 8_bit_reg IS
BEGIN
        PROCESS (clk)
        BEGIN
                IF (enable ='1') AND a'STABLE(5 ns) AND
                  (clk = '1') AND (clk'LASTVALUE = '0') THEN
                  b <= a;
                END IF;
        END PROCESS;
END behavior;
```


* This architecture is more robust that the last one. It makes use of the attribute 'LASTVALUE, to ensure that the last value of the clock before the transition was '0'.



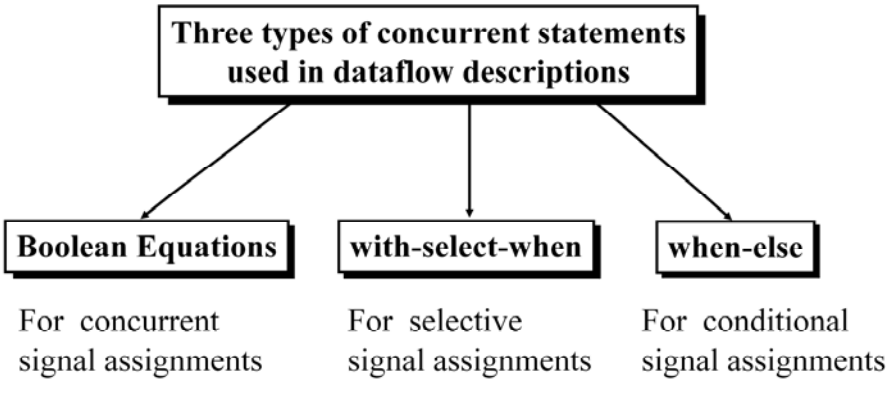
* Till now we discussed the relationship between a design entity and its entity declaration and architecture body. We saw that an entity declaration consists of an interface that contains a list of ports. Ports are of a specific type and have a mode (direction of flow). We also saw that architecture bodies may be written in any of three styles: behavioral, dataflow, and structural (or a combination of them).

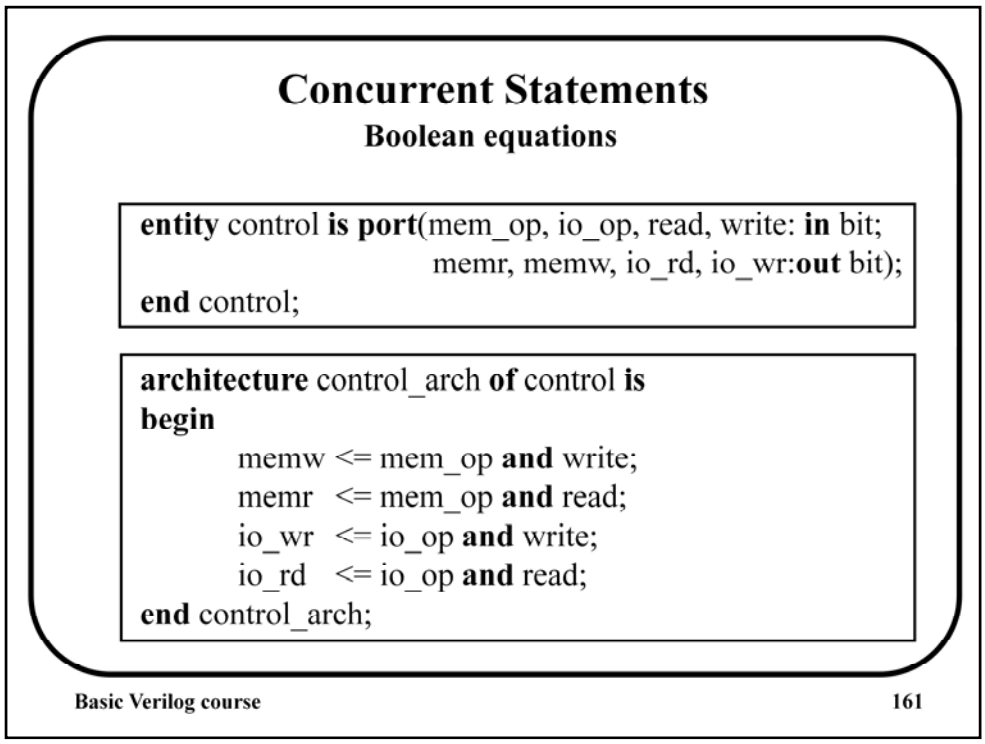
* VHDL offers two types of statements: concurrent statements, and sequential statements. Concurrent statements are commonly used in dataflow and structural descriptions, while sequential statements are usually used in behavioral descriptions.

* Sequential statements are always written within a process as we will explain later. Concurrent statements lie outside of a process.

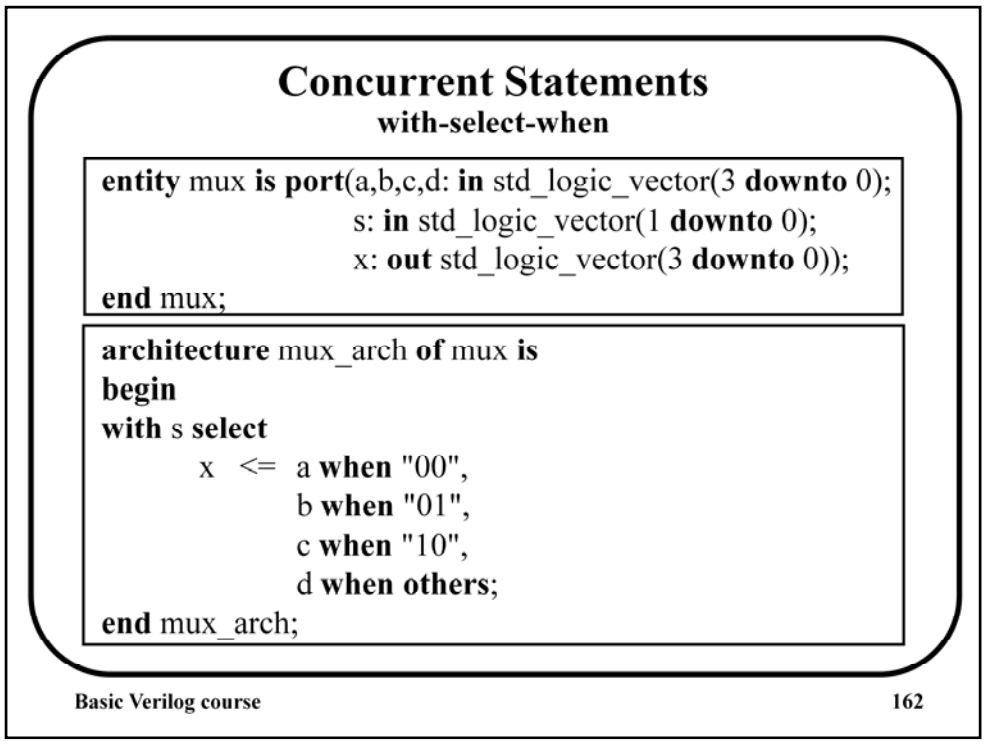
* Conceptually, concurrent statements execute concurrently, therefore their order is not important.

Concurrent Statements





* Boolean equations can be used in both concurrent and sequential signal assignment statements. Here we illustrate the use of Boolean equations in concurrent statements.



* The *with-select-when* statement provides **selective signal assignment**, which means that a signal is assigned a value based on the value of a selection signal.

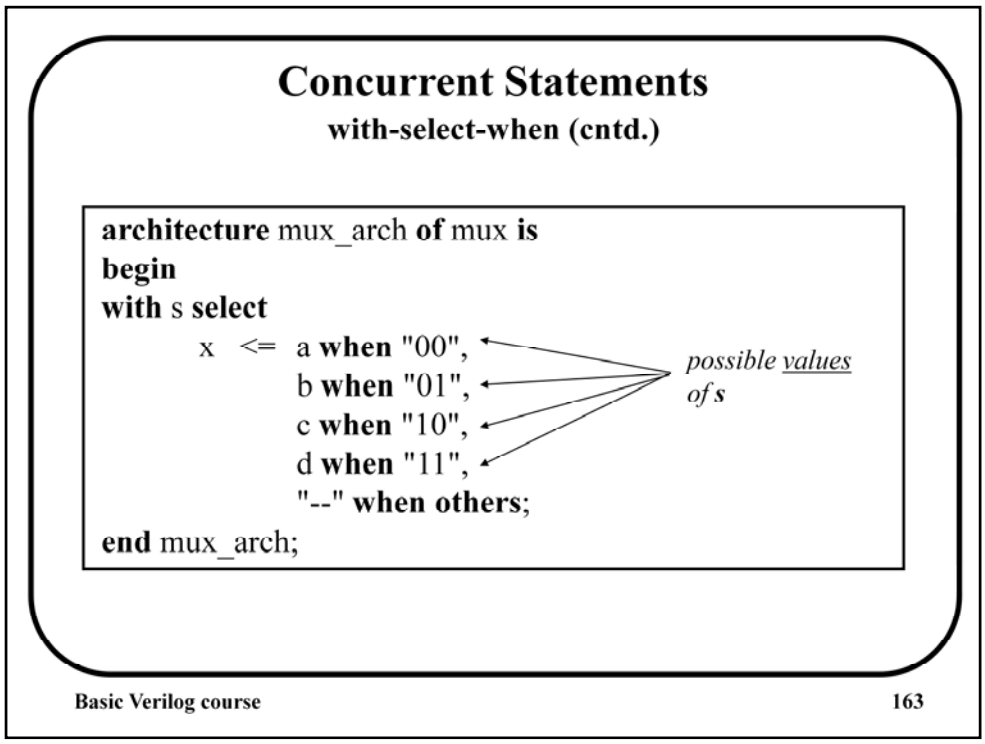
* In the above example, *x* is assigned a value based on the current value of the selection signal *s*.

***All** values of the selection signal must be listed in the when clauses, and must be mutually exclusive.

* The reserved word **others** is used to indicate all other possible values for *s*. Here we used **others** instead of "11" for the following reason:

- *s* is of type std_logic_vector and there are 9 possible values for an object of type std_logic. If "11" were specified instead of others, only 4 of the 81 values would be covered in the with-select-when statement. ("1X", "Z0" ... etc. would not be specified).

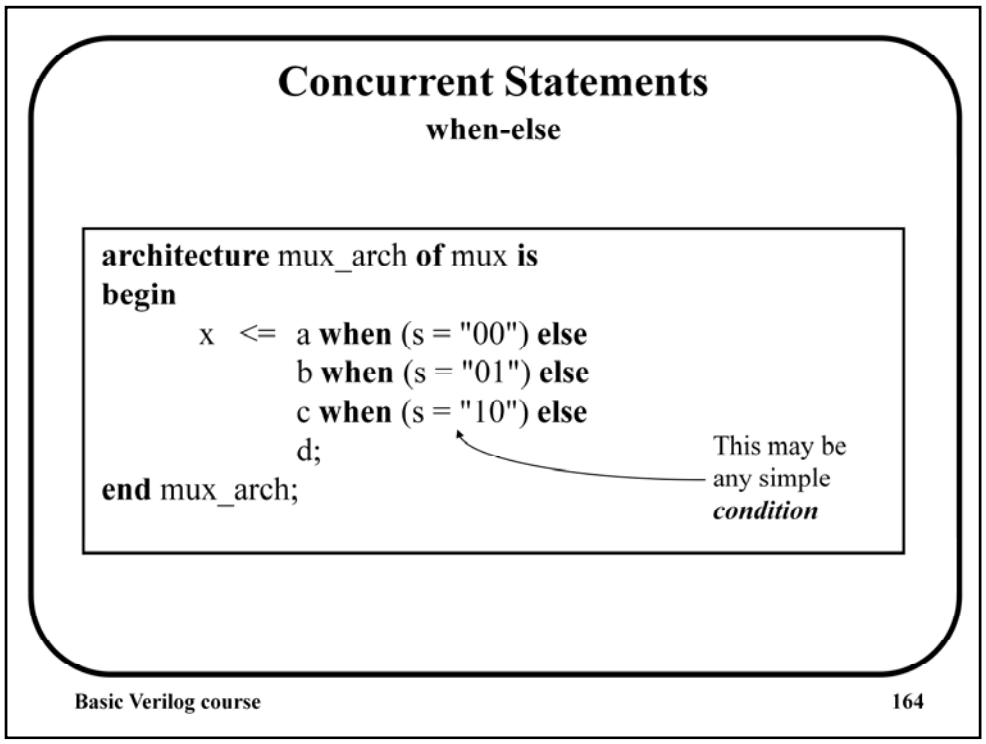
-



* You can explicitly specify "11" as one of the values of *s*, however, **others** is still required to specify **all** possible values of *s*.

* The metalogical value "--" is used to assign the don't care value to *x*.

* The last **when** indicates that signal *x* will receive the value "--" if signal *s* takes on a value other than those explicitly listed.

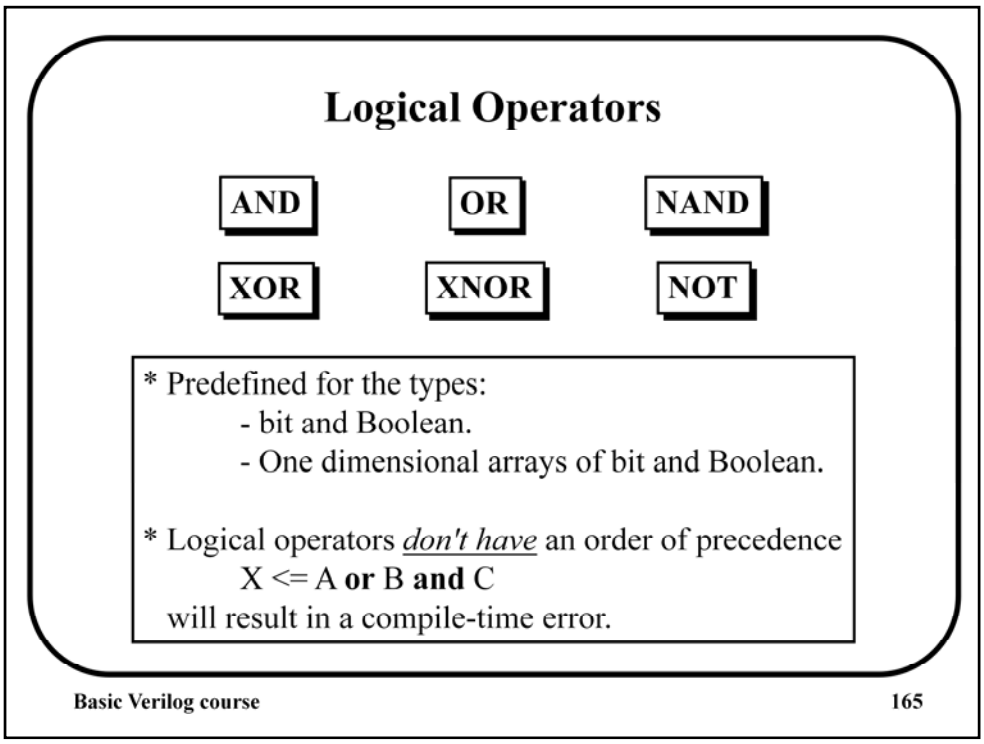


* A selective signal assignment describes logic based on mutually exclusive combinations of values of the selection signal. That is not necessarily true for conditional signal-assignment.

* The when-else statement provides for conditional signal assignment, which means that a signal is assigned a value based on a condition.

* In the above example, x is assigned a value based on the evaluation of the conditions. x is assigned a value based on the first condition that is true.

* The when conditions in the when-else statement can specify any simple expression. (They are not necessarily mutually exclusive).



* Logical operators are the cornerstone of Boolean equations.

* To use these operators with arrays of bits or Booleans, the two operands must be of the same length.

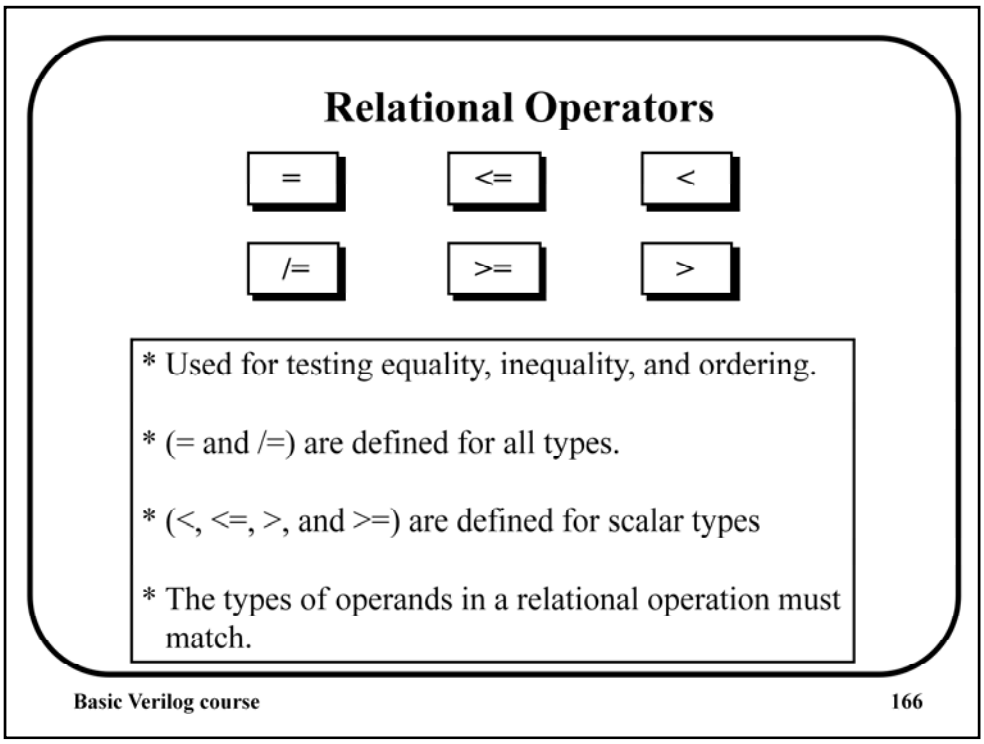
* The IEEE 1164 standard also overloads these operators for the types std_ulogic, std_logic, and their one-dimensional arrays.

* The logical operators in VHDL don't have an order of precedence.

```
X <= A or B and C
```

will result in a compile-time error. This expression should be written as:

```
X <= A or (B and C)
```



* Relational operators are used for testing equality, inequality, and ordering.

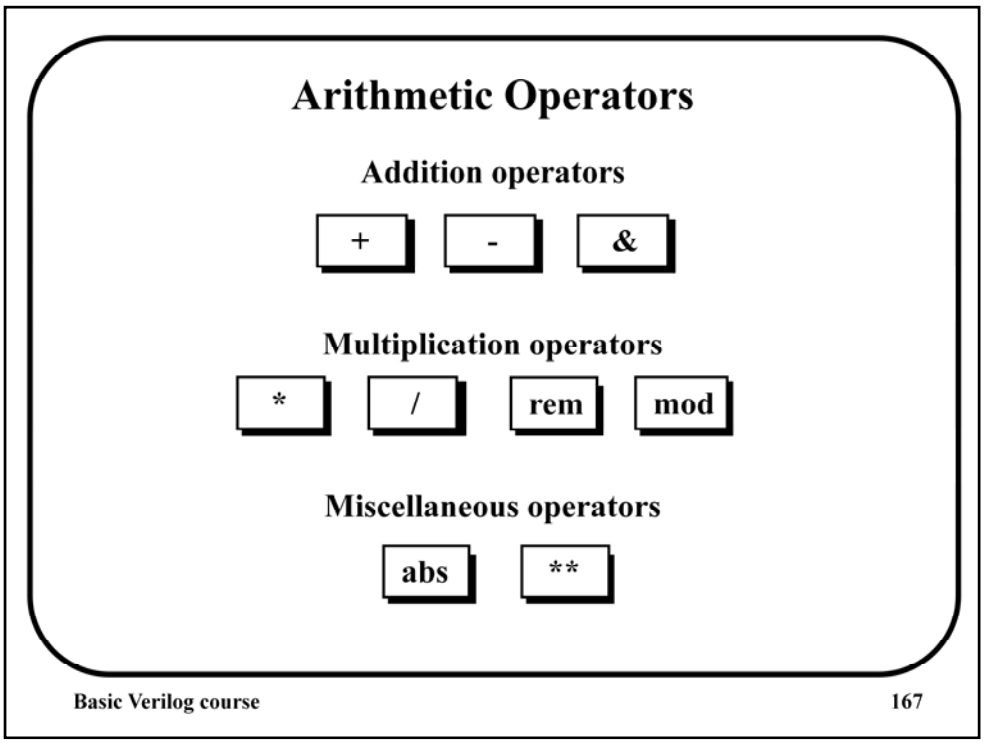
* The equality and inequality operators are defined for all types.

* The magnitude operators (<, <=, >, and >=) are defined for scalar types or an array with a discrete range.

* Arrays are equivalent only f their lengths are equivalent and all corresponding elements of both arrays are equivalent.

* The result of any relational operation is Boolean (True or False).

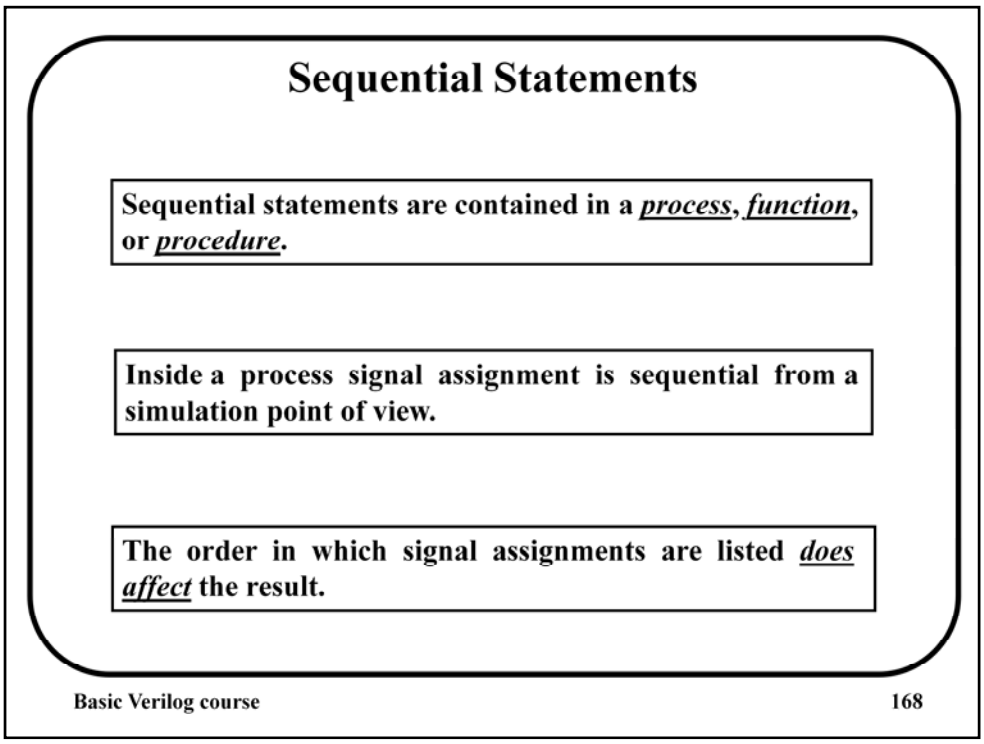
* The types of operands in a relational operation must match.



Since we started talking about operators, we will discuss also arithmetic operators: addition, subtraction, concatenation, multiplication, division, remainder, modulus, and absolute value.

* All arithmetic operators are predefined for the types integer and real.

* Most operators require both operands to be of the same type
  - Exception: operands of physical type may be multiplied and divided by integers and real numbers



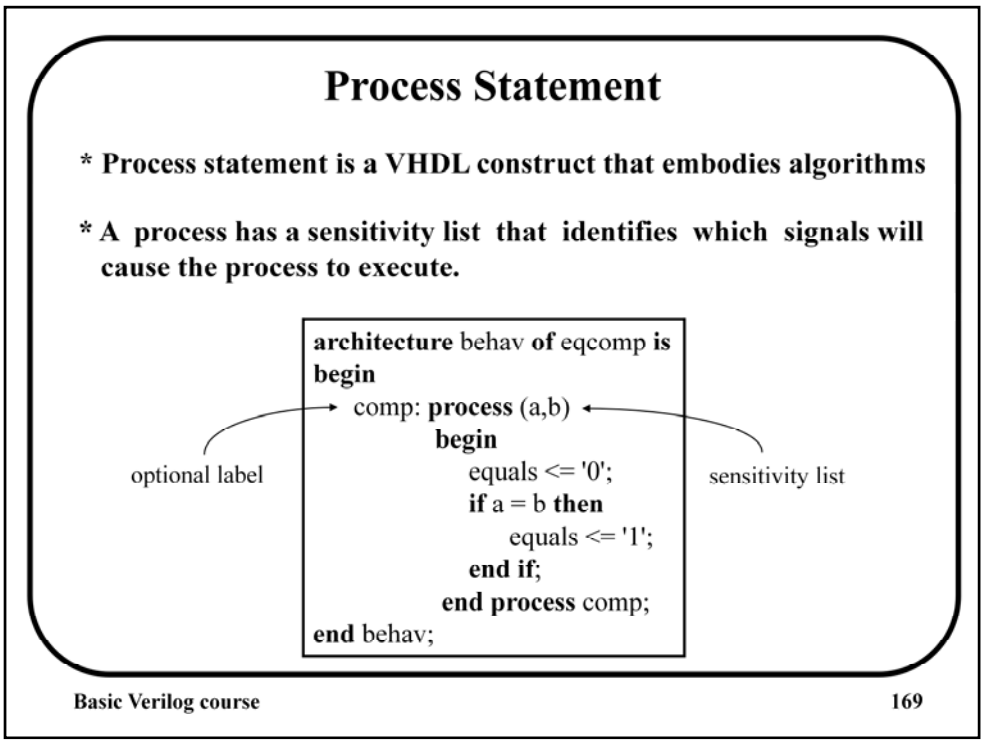
* Sequential statements are those statements contained in a process, function, or procedure.

* The collection of statements that make up a process (that is, the process itself) constitutes a concurrent statement. If a design has multiple processes, then those processes are concurrent with respect to one another and to other concurrent statements within the architecture. Inside a process, however, signal assignment is sequential.

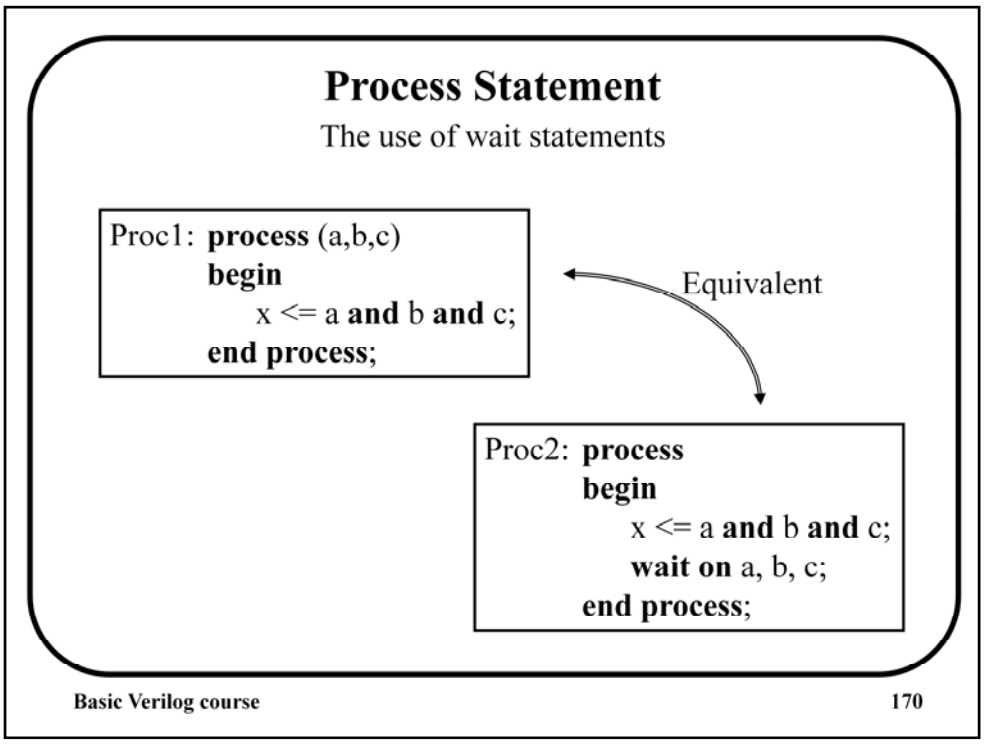
* Don't confuse sequential statements with sequential logic. Sequential signal assignment can be used to describe combinational logic.

* Sequential execution is most often found in behavioral descriptions.

* Before showing sequential statements we will first introduce you the processes.

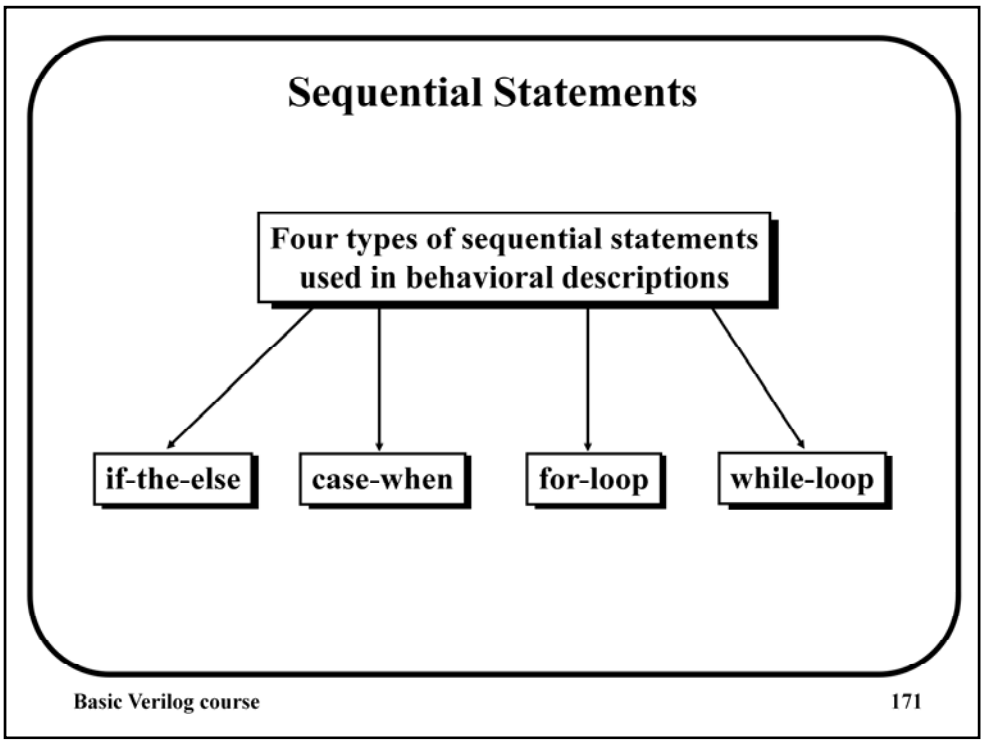


* A process statement is one of VHDL's design constructs for embodying algorithms.
* A process statement begins with an optional label (comp in the above example), followed immediately by a colon (:), then the reserved word **process** and a sensitivity list.
* A sensitivity list identifies (for a simulator) which signals will cause the process to execute.
* The process statement is completed with the reserved words **end process** and optionally the process label.
* A process is either being executed or suspended. It is executed when one of the signals in its sensitivity list has an event: a change in value. When this happens the sequential statements inside the process are executed. A process continues to execute until the last statement is reached. At this point, the process suspends itself and is executed again only when there is an event on a signal in the sensitivity list.



* An explicit sensitivity list is not required: A wait statement can also be used to suspend a process.
* Both of the indicated two processes will execute when a change in value of signals a, b, or c occurs.
* When a model is simulated, it goes through an initialization phase and then repeated simulation cycles. The initialization phase starts with the current simulation time set to 0 ns. In general, if an explicit initialization value is not specified for a signal, then the signal will have the initial value '0' if it is of type bit or 'U' (uninitialized) if it is of type std_logic. Next, each process is executed until it suspends. After the initialization phase, simulation cycles are run, and each consists of the following steps:

1. Signals are updated. Updating will cause signals to transition and events to occur on these signals.
2. Each process that is sensitive to a signal that has had an event on it in the current simulation cycle is executed.

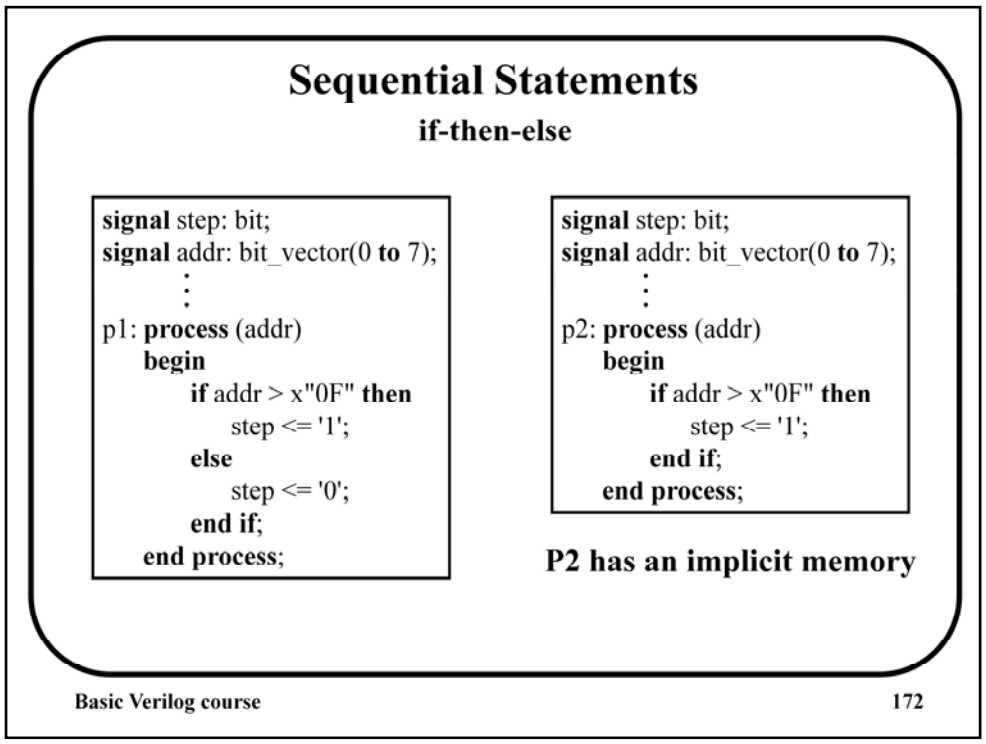


* The if-then-else statement is used to select a set of statements to execute based on a Boolean evaluation of a condition or a set of conditions.

* Case statements are used to specify a set of statements to execute based on the value of a given selection signal. A case-when statement can be used as the equivalent of a with-select-when statement.

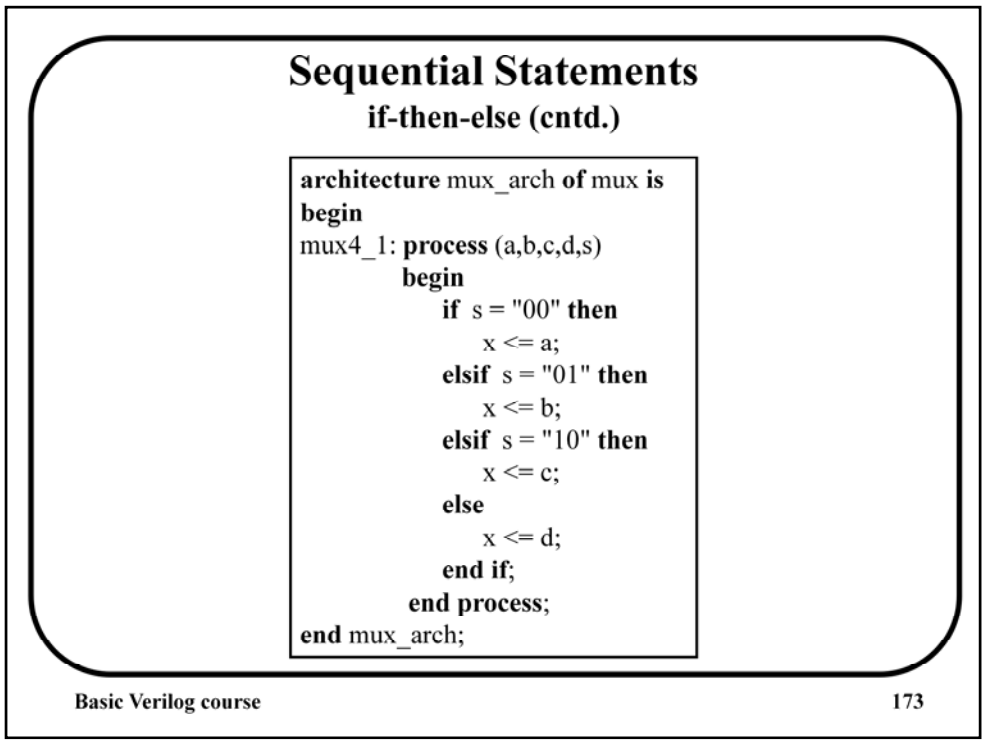
* The for-loop executes all the statements within a loop for a fixed number iterations based on a controlling value.

* The while-loop continues to execute as long as a controlling logical condition evaluates true.



* Here is an example for the **if-then-else** statement. If the condition of the **if** statement is true, the sequential statement or statements following the keyword ***then*** are executed. If the condition evaluates false, the sequential statement or statements after the keyword ***else*** are executed. The statement is closed with the words ***end if***.

* In the above example the process P2, is not equivalent to P1. The process P2 implies that *step* should retain its value if *addr* is less than or equal to **0F**. This is referred to as implied or implicit memory. Thus once step is asserted, it will remain forever asserted.



* The if-the-else can be expanded further to include an elsif to allow for further conditions to be specified.

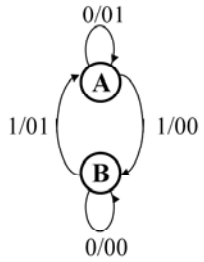
* The when-else construct can be rewritten with an if-then-else statement.

* The above example describes the 4 bit-multiplexer that we have already described using the when-else statement.

Sequential Statements

case-when

```
case present_state is
    when  A  =>  y <= '0'; z <= '1';
        if  x = '1'  then
            next_state <= B;
        else
            next_state <= A;
        end if;
    when B  =>  y <= '0'; z <= '0';
        if  x = '1'  then
            next_state  <= A;
        else
            next_state <= B;
        end if;
end case;
```



inputs: x
outputs: y,z

* Case statements are used to specify a ***set of statements*** to execute based on the value of a given selection signal.

* In the shown example, the case statement is used to compute the next-state based on the value of the current-state.

* We should have the following two lines at the beginning of the architecture description:

type States is (A,B);

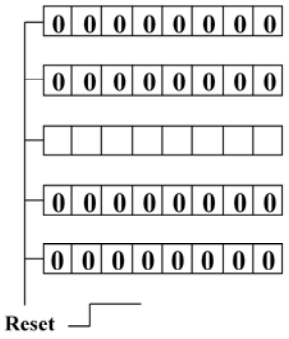
signal present_state, next_state: States;

* The reserved word ***others*** may be used to complete the descriptions if not all the states are listed with the when clauses. In the above example we had only two states, so no ***others*** clause is needed.

Sequential Statements

for-loop

```
type register is bit_vector(7 downto 0);
type reg_array is array(4 downto 0) of register;
signal fifo: reg_array;

process (reset)
begin
    if reset = '1' then
        for i in 4 downto 0 loop
            if i = 2 then
                next;
            else
                fifo(i) <= (others => '0');
            end if;
        end loop;
    end if;
end process;
```



Basic Verilog course 175

* Loop statements are used to implement repetitive operations.

* The for statement will execute for a specific number of iterations based on a controlling *value* (while-loop as we will see later depends on a controlling *condition*).

* In a for-loop, the loop variable ***i*** is automatically declared.

* The two limits of the loop variable must be constant values.

* The reserved word **next** is used to skip one iteration for the for loop.

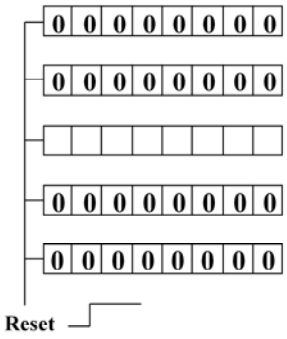
* The code fifo(i) <= (others => '1') implies that fifo(i) is an aggregate, or an array of more than one element. This is used instead of writing:

fifo(i) <= "00000000";

Sequential Statements

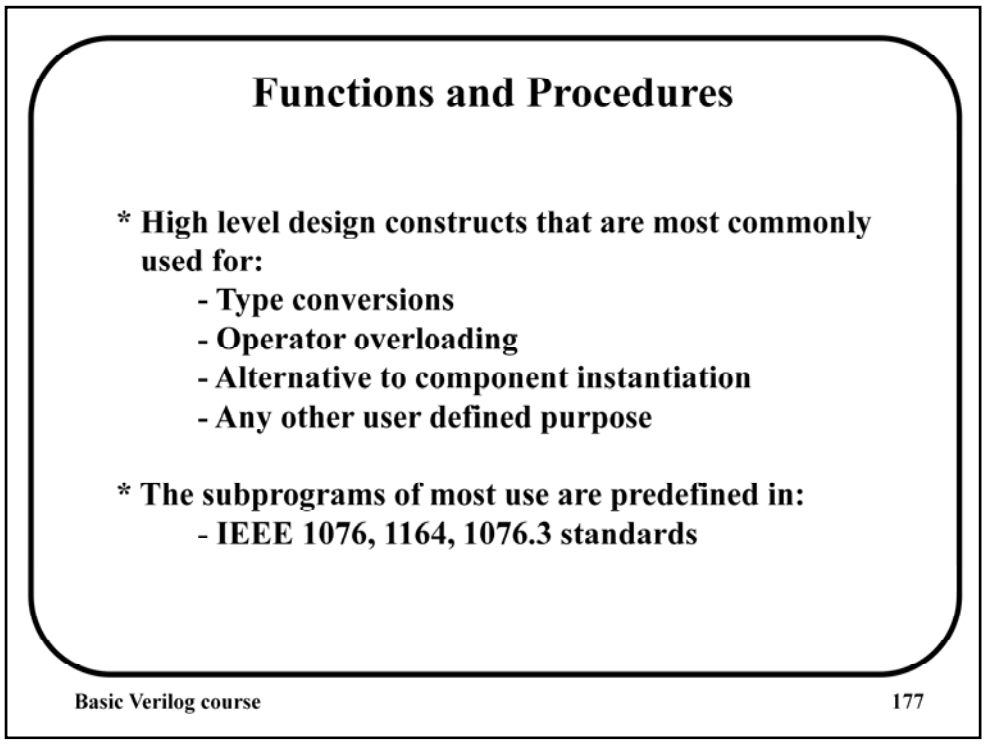
while-loop

```
type register is bit_vector(7 downto 0);
type reg_array is array(4 downto 0) of register;
signal fifo: reg_array;

process (reset)
    variable i: integer := 0;
begin
    if reset = '1' then
        while i <= 4 loop
            if  i /= 2 then
                fifo(i) <= (others => '0');
            end if;
            i := i + 1;
        end loop;
    end if;
end process;
```



* The while statement will execute for an operation as long as a controlling logical conditions evaluates true.

* If we rewrite the previous example using the while-loop, an extra step is required to initialize the controlling variable ***i***.

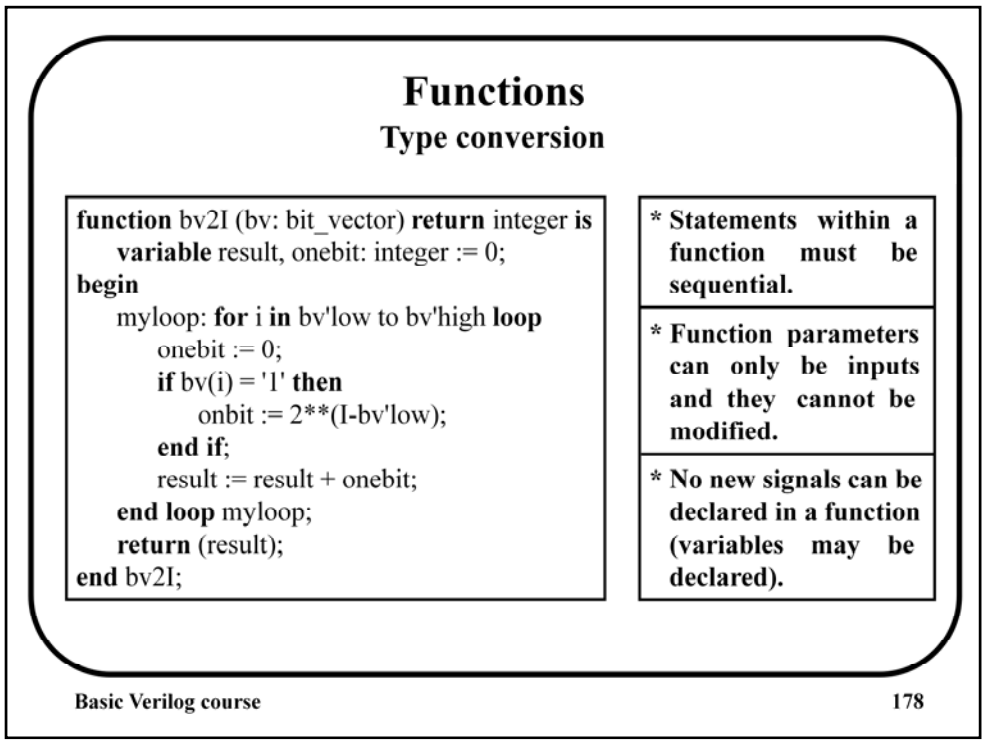


* Functions and procedures are high-level design constructs that compute values or define processes for type conversion, operator overloading, or as an alternative to component instantiation.

* A set of predefined functions are already defined in the IEEE 1076, 1164, and 1076.3 standards. These functions are the most commonly used by VHDL programmers. However, users can define their own functions.

* Functions and procedures can be declared, defined, and invoked much the same way as in software languages.

* Values returned or altered by functions or procedures may or may not have a direct hardware significance. For example, we can use functions to represent Boolean equations, or for type conversion. Boolean expressions may correspond to actual logic circuits, the other application do not represent hardware structures.



* The first line declares the function *bv2I* and defined the input parameter of type **bit_vector**, and the return type as **integer**. Note that the input parameter in this example is unconstrained because the widths of the bit_vectors that will be passed to the function are not known apriori. Function parameters can only be inputs and they cannot be modified.

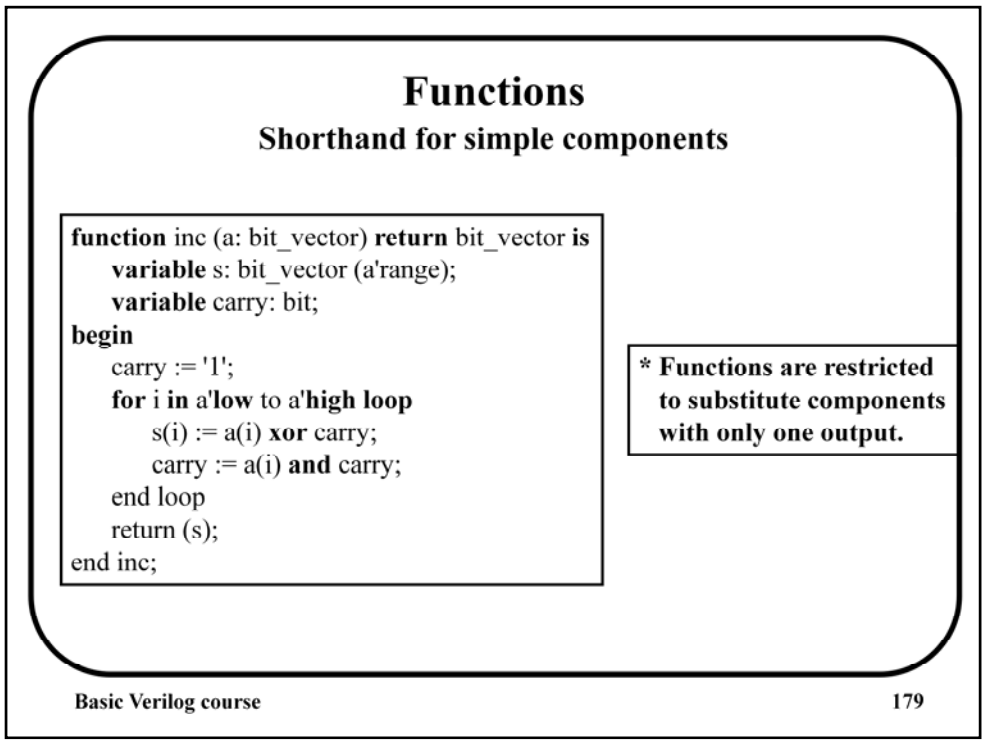
* The second line is the function declarative region, in which variables can be declared. Here two variables are declared as integers and initialized to 0. No new signals can be declared in a function. Only variables can be declared in its declarative part.

* The function definition is enclosed between its begin and its end. All statements within the function definition *must* be *sequential statements*.

* Here we used the attributes 'low and 'high to return the lowest and highest indices of the bit_vector. bv'low is considered as the least significant bit, and bv'high as the most significant bit.

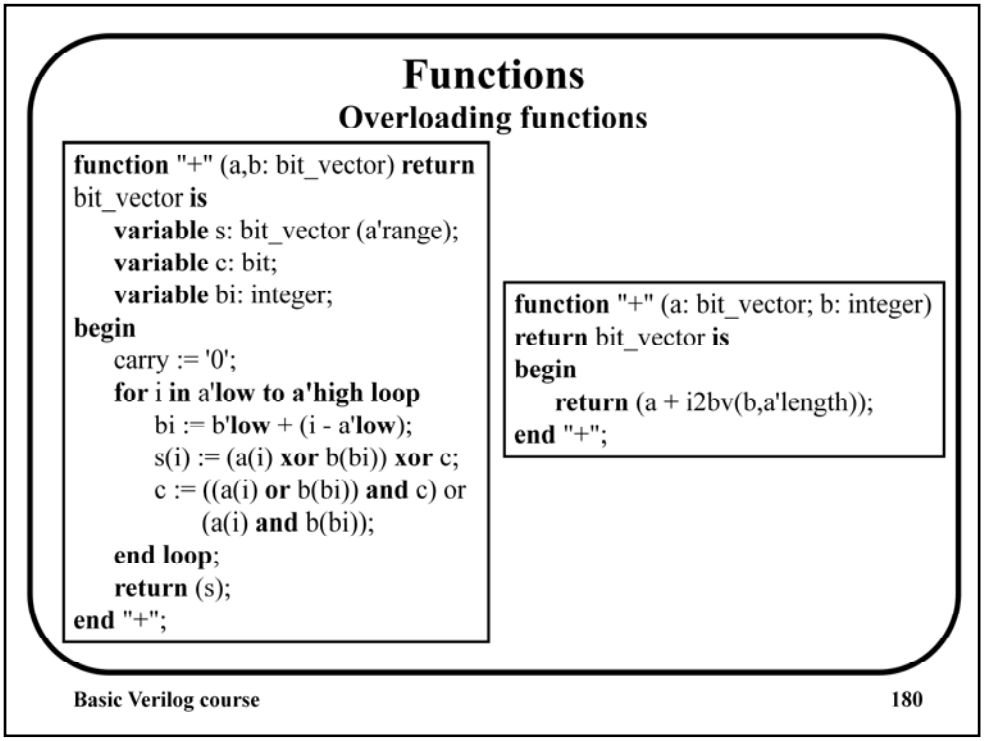
* The loop is used to ascend the bit_vector from LSB to MSB, with the loop variable i used to index the bit_vector.

* When the loop finishes, the result is returned by the reserved word **return**.



* Functions are sometimes used in place of component instantiations because they provide a way to write concise, C-like code.

* Functions are restricted to substituting for components with one output.



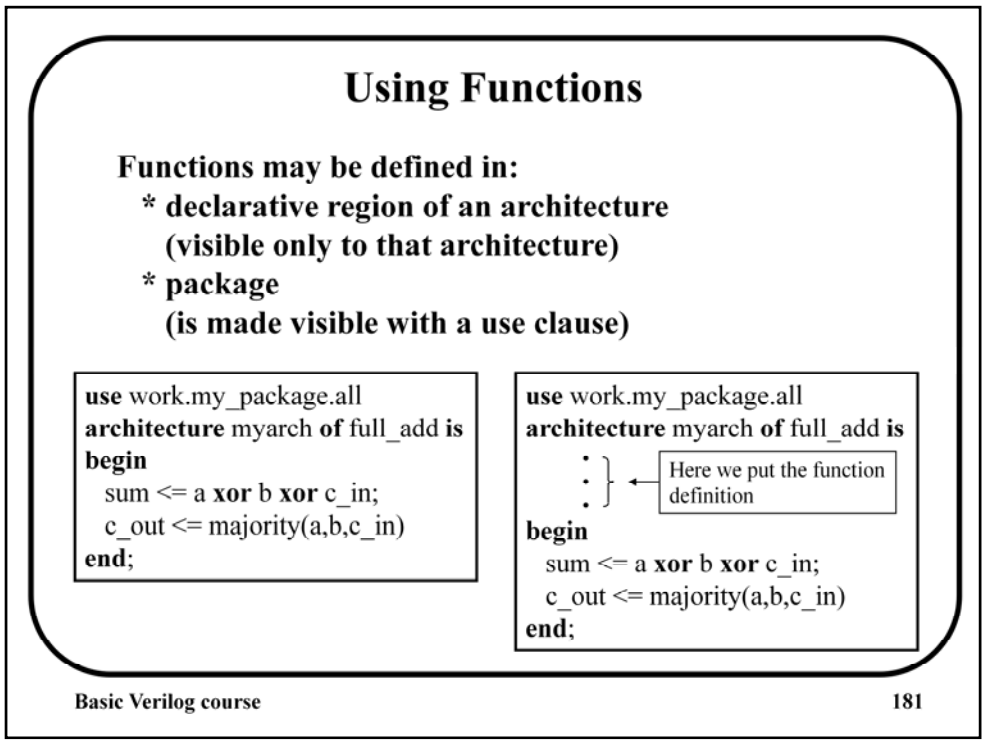
* A powerful use of functions is to overload operators. An overloaded operator is one for which there is a user-defined function that handles an operation for *various data types*.

* The + operator is defined by the IEEE 1076 standard to operate on numeric types (integer, real, and physical types), but not with types such as std_logic or bit_vector.

* To add an integer to a signal of type std_logic, an overloaded operator is required. Other useful addition operations include, among others, the addition of:

- bit_vector to an integer,
- bit_vector to a bit,
- std_logic_vector to an integer,
- std_logic_vector to a std_logic.

* You can create several functions that define the same operation for different types.

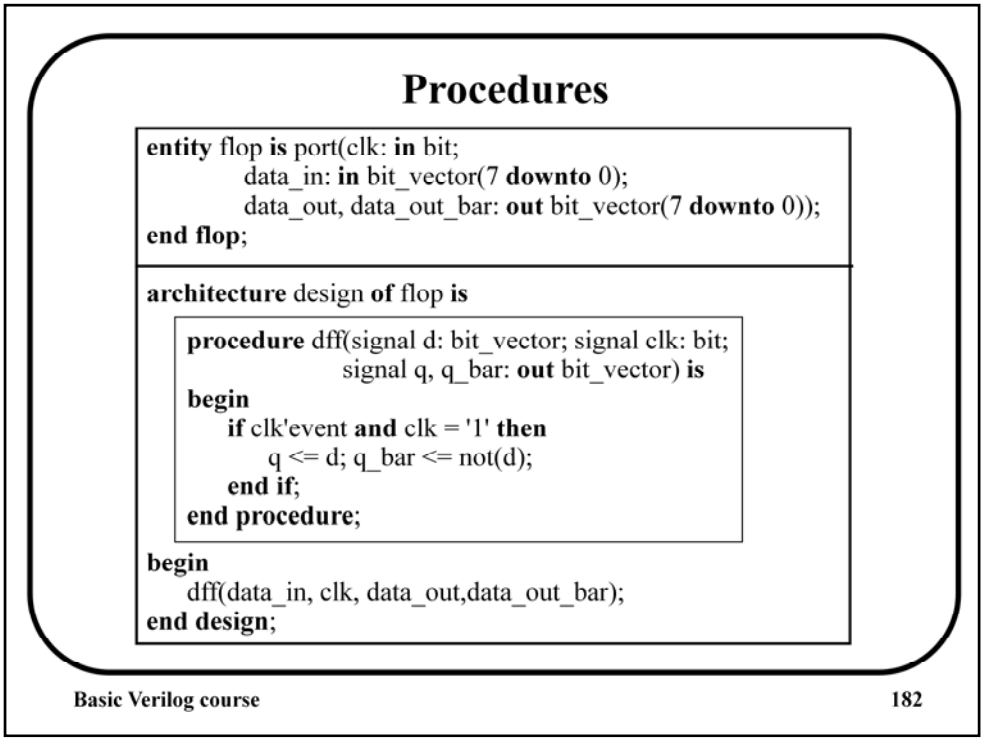


* A function may be defined in the declarative region of an architecture, in which case the function definition also serves as the function declaration.

* Alternatively, a package may be used to declare a function with the definition of that function occurring in the associated package body. We will talk about packages later.

* A function declared in the declarative region of an architecture is visible only to that architecture.

* A function declared in a package can be made visible, with the use clause.

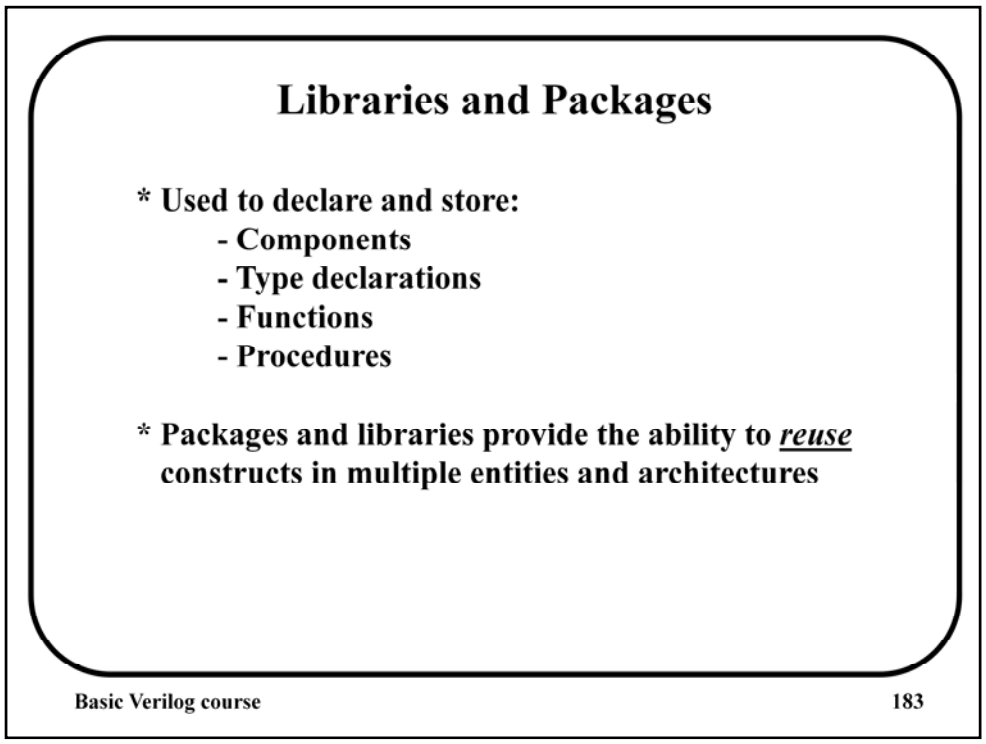


* Like functions, procedures are high-level design constructs to compute values or define processes that you can use for type conversions or operator overloading, or as an alternative to component instantiation.

* Procedures differ from functions in a few ways:
    1. A procedure can return more than one value. This is accomplished with parameters. If a parameter is declared as mode out or inout, then the parameter is returned. (A parameter in a function can only be of mode in).
    2. Procedure can have a wait statement whereas function cannot.

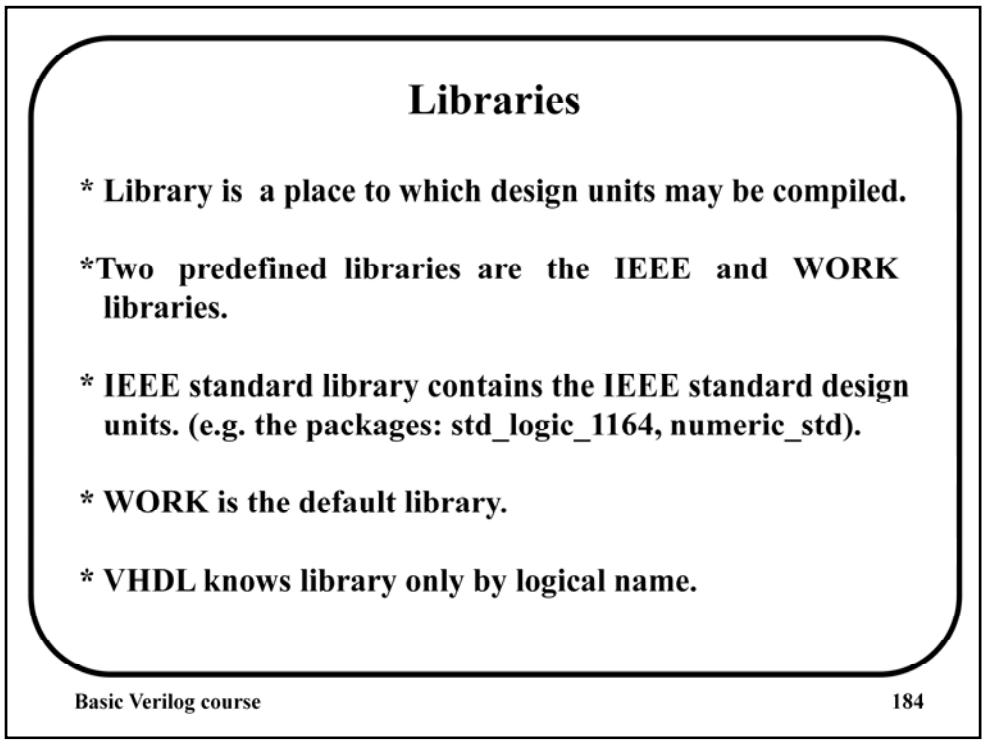
* similarities between functions and procedures:
    1. All statements within a procedure must be sequential
    2. Procedures cannot declare signals.
    3. Procedures are declared the same way as functions: either in the architecture declarative region or in the package body.

* If the class of data object is not defined the default value is in.



* Increased complexity of devices requires configuration and revision control. There is a need for using libraries of previous designs and modification of these libraries

* Libraries and packages are used to declare and store components, type declarations, functions, procedures so that they can be reused later in other designs.



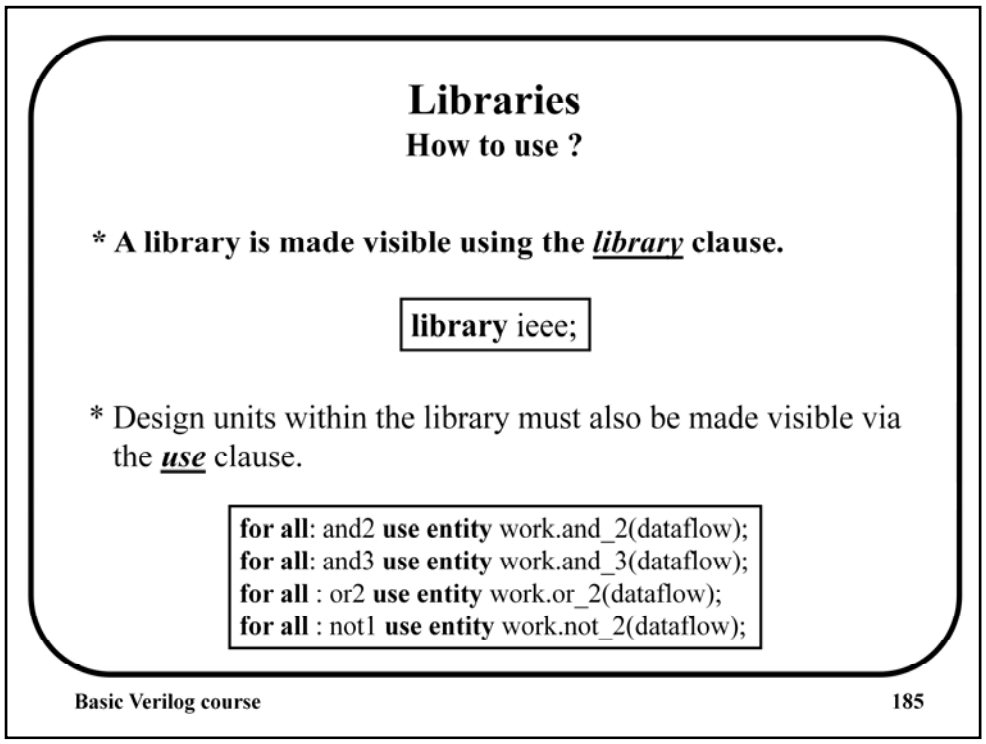
* A library is a place (directory) to which design units may be compiled. For example, entity declaration and architecture body pairs may be compiled to a library.

* Design units within a library may be used within other entities, provided that the *library and the design unit* are visible.

* The IEEE and the work library are predefined. The IEEE library is a storage place for for IEEE standard design units such as the packages std_logic_1164, numeric_std, and numeric_bit.

* The work library is a storage place for the designs you are currently working with, and it is the default place to which a VHDL processing tool compiles design units. The work library is a place in which to compile designs while they are in development. After verifying the accuracy of a design unit that is currently in the work library and that you wish to reuse in subsequent designs or as part of a team project, you may want to compile the design into an appropriate library.

* The name of a library is not necessarily the name of a directory. For example, IEEE does not necessarily represent a directory in your home or root directory. Rather, it is a logical name for a storage place.

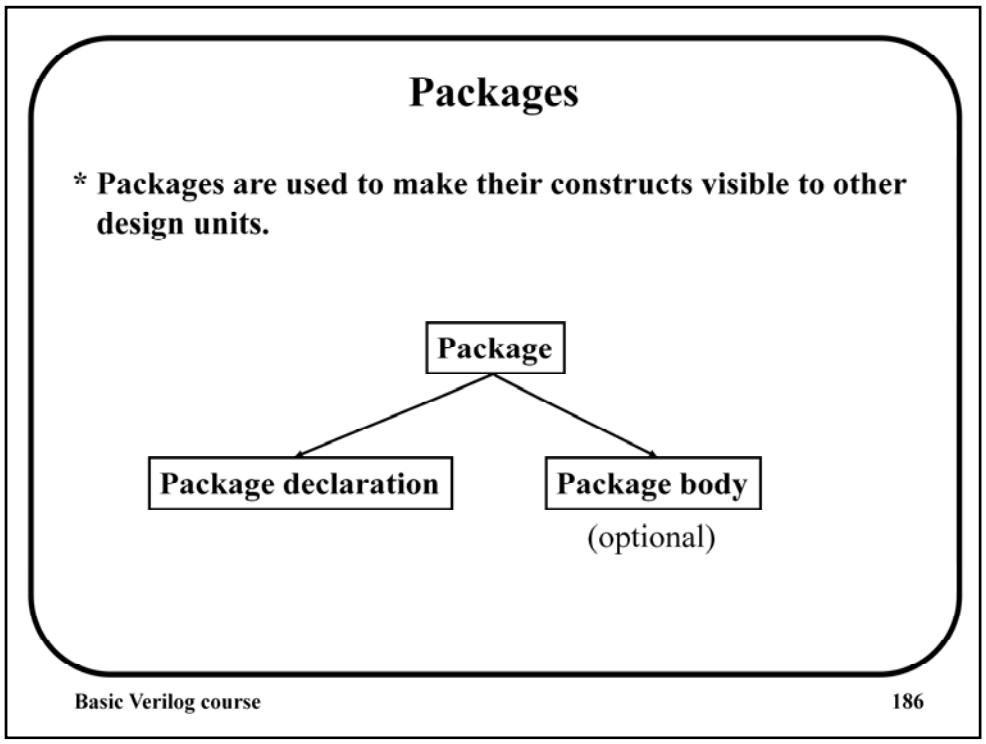


* n order to use design units from a library, the library must be made visible by way of a ***library*** clause, as in

library ieee;

* This clause makes the library accessible. It does not, however, allow you to immediately begin using the design units or items within the library. You must first make those components, packages, declarations, functions, procedures and so on visible via a ***use*** clause.

* The work library is implicitly visible in all designs, so it is never required to use a *library* clause to make the work library visible.

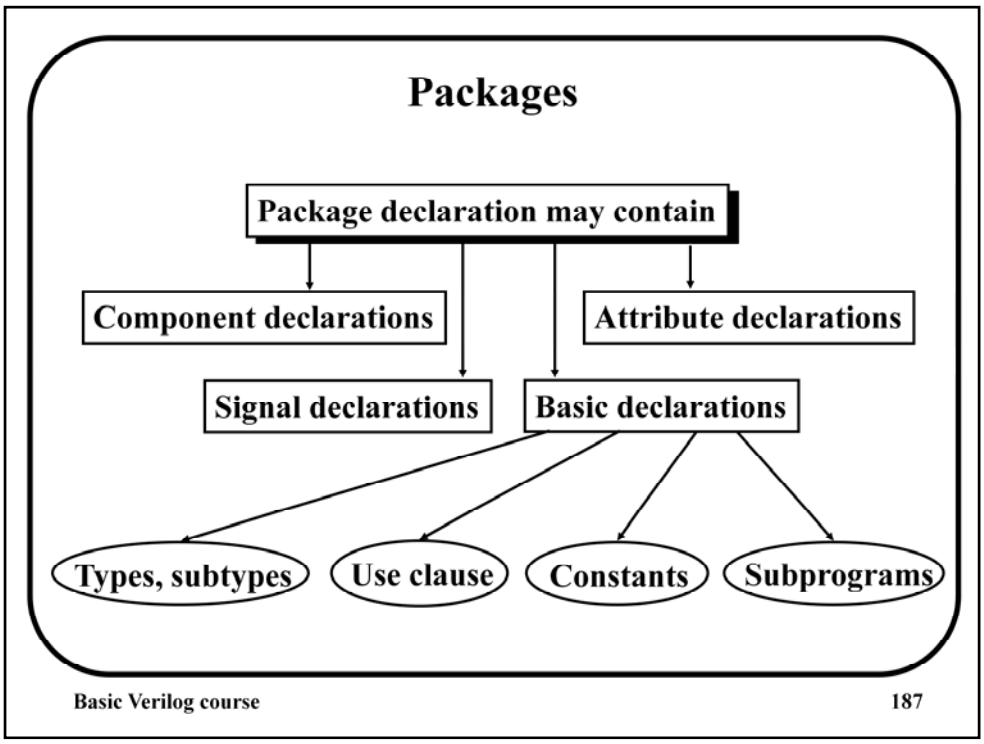


* A package is a design unit that can be used to make its type, component function, and other declarations visible to design units other that itself. This is in contrast to an architecture declarative region, for which the type, component, function, and other declarations cannot be made visible to other design units.

* A package consists of a package declaration, and *optionally*, a package body

* A package declaration is used to declare items such as types, components, functions, and procedures.

* A package body is where the functions and procedures in a package declaration are defined. A package body that does not declare any function or procedure does not need an associated package body.



* The package declaration is used to declare items as shown in the above figure. It can be used to declare:

1. Components
2. Attributes
3. Signals
4. Other basic declarations such as types, subtypes, constants, procedures and functions.

Package Declaration

Example of a package declaration

```
package my_package is
        type binary is (on, off);
        constant pi : real : = 3.14;
        procedure add_bits3 (signal a, b, en : in bit;
        signal temp_result, temp_carry : out bit);
end my_package;
```

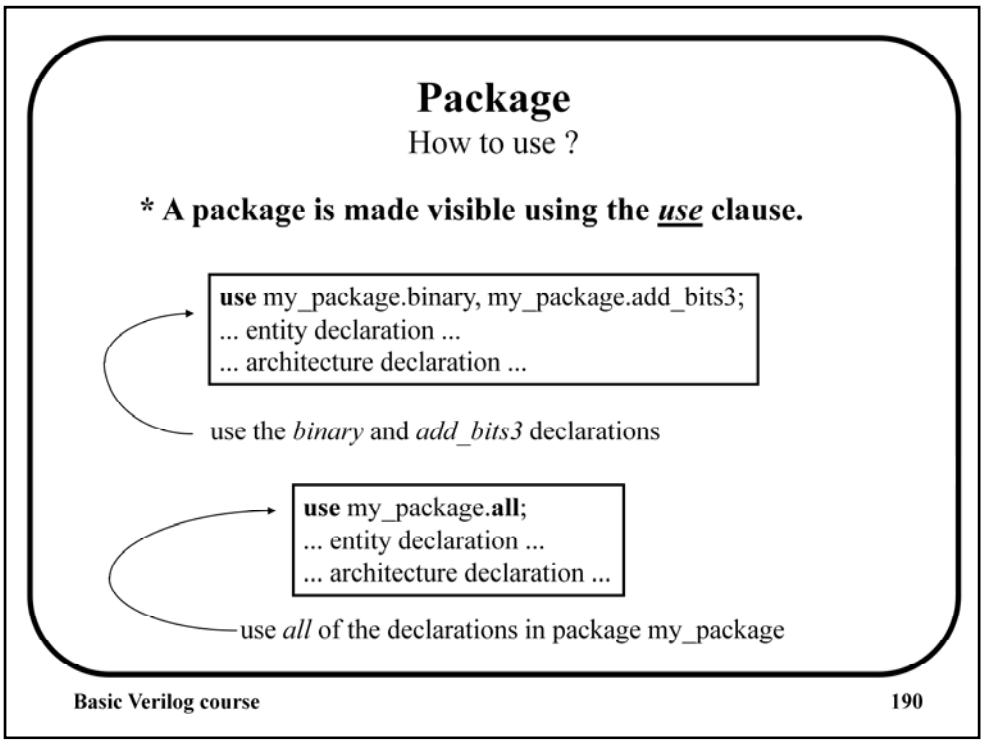
The procedure body is defined in the "package body"

Package Body

* **The package declaration contains only the declarations of the various items**

* **The package body contains *subprogram bodies* and other declarations not intended for use by other VHDL entities**

```
package body my_package is
    procedure add_bits3 (signal a, b, en : in bit;
    signal temp_result, temp_carry : out bit) is
    begin
        temp_result <= (a xor b) and en;
        temp_carry <= a and b and en;
    end add_bits3;
end my_package;
```

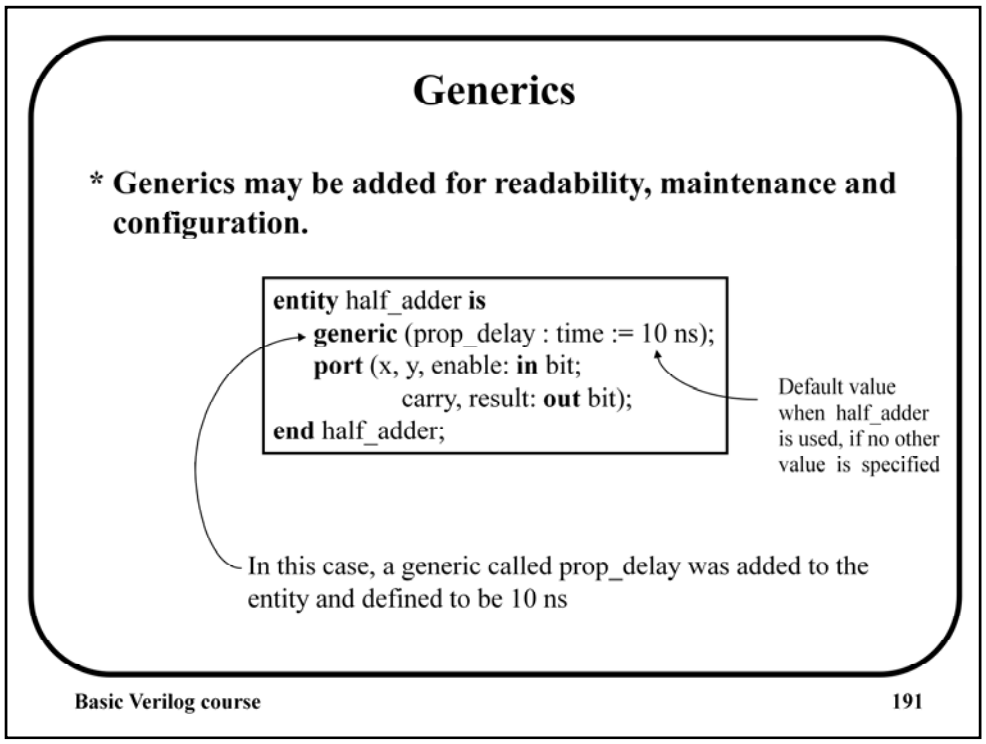


* A package is made visible with a use clause of the form:

use library_name.package_name.item

if the library_name is omitted, the work library is considered by default.

* If you wish all the items in the package to be visible, you can use the reserved word **all**. Unless there were a specific item in a package that conflicted with the definition of an item in another package, it is unlikely that you would specify anything but **all**.



* In addition to inputs and outputs of a hardware, other conditions may influence the way it operates. These include the environment where the hardware components is used, and the physical characteristics of the hardware component itself.

* It should not be necessary to generate a new hardware description for every specific condition. Furthermore, many hardware components in various logic families, for example, the LS, F, and ALS series of the 7400 logic family, are functionally equivalent, and differ only in their timing characteristics.

* VHDL allows the designer to configure the *generic* description of a component when it is used in a design. Generic descriptions may be configurable for size, physical characteristics, timing, loading, and environmental conditions.

* Usage of generic parameters and passing of values to these parameters is done in much the same way as it is with ports. The syntax of constructs related to ports and generics are similar, except that the generic clause and generic map constructs use the keyword GENERIC instead of PORT.

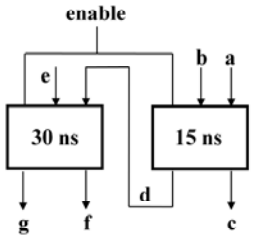
Generics (cntd.)

```
architecture data_flow of half_adder is
begin
        carry = (x and y) and enable after prop_delay;
        result = (x xor y) and enable after prop_delay;
end data_flow;
```


* Here in the architecture we use the parameter prop_delay instead of specifying a value. When we instantiate this component, we can send different values for prop_delay. So there is no need to change the description every time the value of prop_delay differs. We need only to send a different parameter.

Generics (cntd.)

```
architecture structural of two_bit_adder is
  component adder generic( prop_delay: time);
      port(x,y,enable: in bit; carry, result: out bit);
  end component;

  for c1: adder use entity work.half_adder(data_flow);
  for c2: adder use entity work.half_adder(data_flow);

  signal d: bit;

begin
  c1: adder generic map(15 ns) port map (a,b,enable,d,c);
  c2: adder generic map(30 ns) port map (e,d,enable,g,f);
end structural;
```



* Usage of generic parameters and passing of values to these parameters is done in much the same way as it is with ports. The syntax of constructs related to ports and generics are similar, except that the generic clause and generic map constructs use the keyword GENERIC instead of PORT.

Delay Types

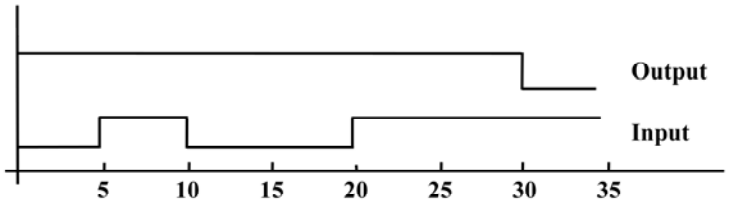
* Delay is created by scheduling a signal assignment for a future time

* There are two main types of delay supported VHDL
  - Inertial
  - Transport

Inertial Delay

* **Inertial delay is the default delay type**
* **It absorbs pulses of shorter duration than the specified delay**

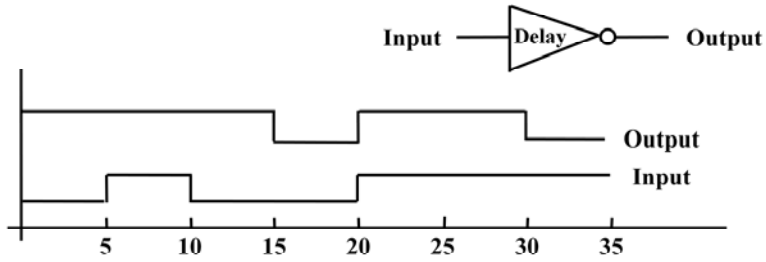
```
-- Inertial is the default
Output <= not Input after 10 ns;
```



Transport Delay

*** Must be explicitly specified by user**
*** Passes all input transitions with delay**

```
-- TRANSPORT must be specified
Output <= transport not Input after 10 ns;
```



Summary

* VHDL is a worldwide standard for the description and modeling of digital hardware
* VHDL gives the designer many different ways to describe hardware
* Familiar programming tools are available for complex and simple problems
* Sequential and concurrent modes of execution meet a large variety of design needs
* Packages and libraries support design management and component reuse

References

D. R. Coehlo, *The VHDL Handbook*, Kluwer Academic Publishers, 1989.

R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.

Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, McGraw-Hill, 1993.

***IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1993.**

References

J. Bhasker, *A VHDL Primer*, Prentice Hall, 1995.

Perry, D.L., *VHDL, McGraw-Hill*, 1994.

K. Skahill, *VHDL for Programmable Logic*, Addison-Wesley, 1996