

EL 6463 : Advanced Hardware Design

Ramesh Karri

email: rkarri@nyu.edu

Phone: 917 363 9703 (cell: try this first)

Skype: karriramesh

Introduction

✓ WHAT

- ✓ VHSIC Hardware Description Language (verilog ?)
- ✓ V=Very High Speed Integrated Circuit
- ✓ HDL to describe behavior, data flow, structure

*to simulate
trade name*

✓ WHY

- ✓ Shrinking design cycle
- ✓ Simulation and formal verification
- ✓ Synthesis
- ✓ Reliable design process (eliminate design errors)
- ✓ Documentation and requirements specification

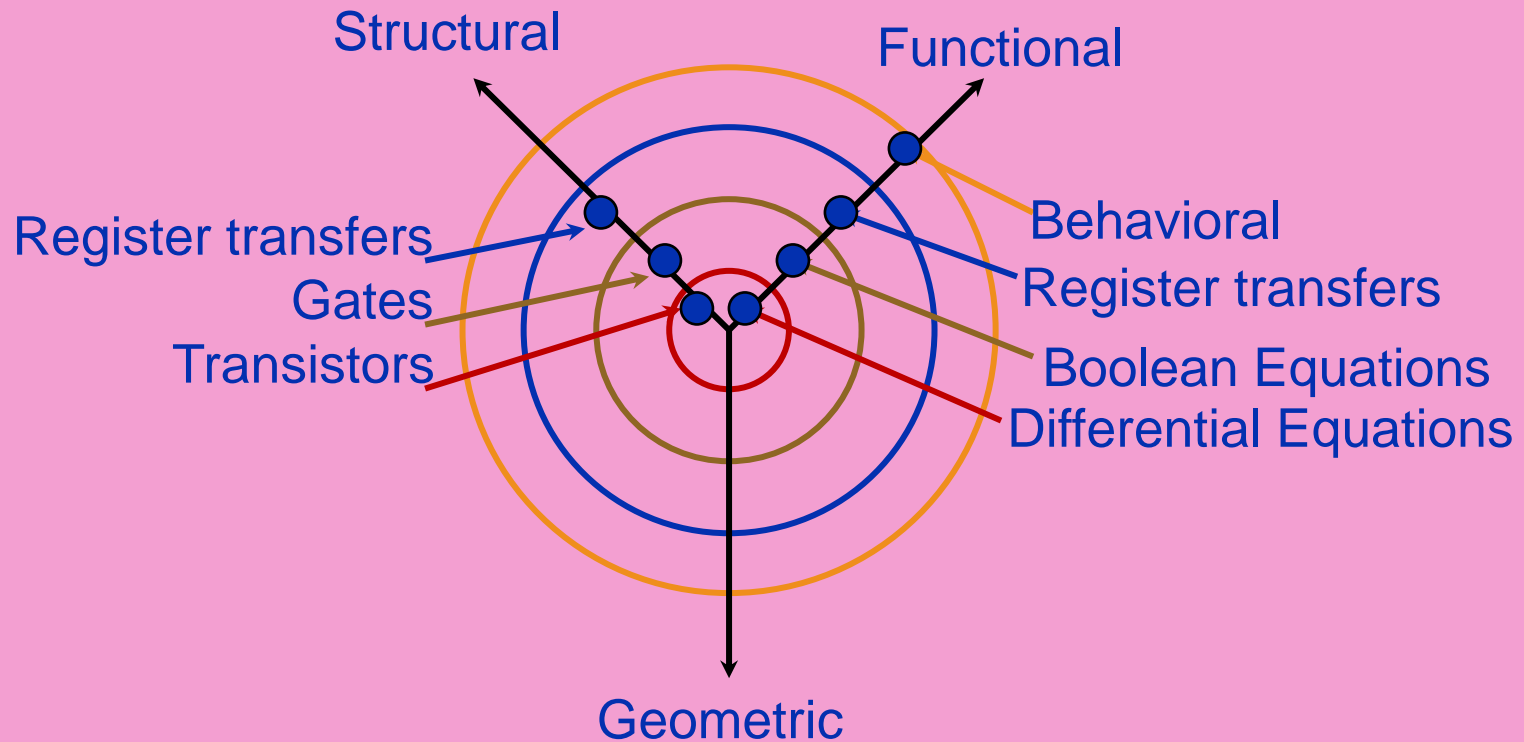
✓ HOW

- ✓ Design using synthesizable VHDL constructs and models
- ✓ Verify behavior and timing using various CAD tools
- ✓ Download synthesized designs onto an FPGA

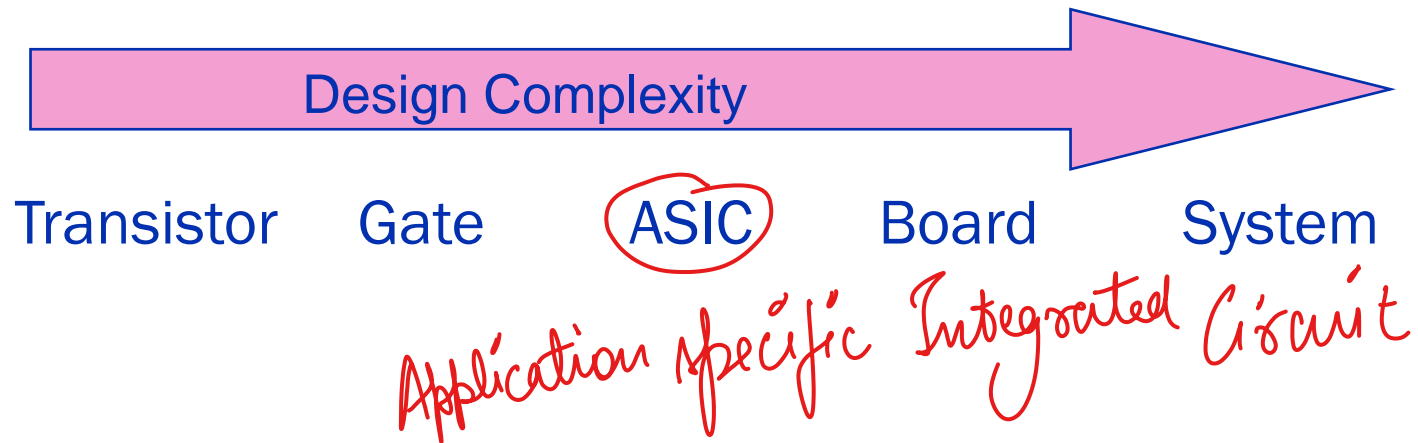
History of VHDL/Verilog

- Both originated in the early 1980s
 - US Department of Defense initiated development of VHDL in the early 1980s
 - because the US military needed a standardized method of describing electronic systems
 - Verilog initially a proprietary tool, later opened by Cadence
- VHDL standardized in 1987 by the IEEE, Verilog in 1995
- Both are accepted as important standard languages for specifying; verifying and designing of electronics
- Multiple versions of each:
 - VHDL IEEE-1076 in 1987, 1993, 2000, 2002, 2008
 - Verilog IEEE-1364 in 1995, 2001, 2005
 - (Also SystemVerilog IEEE-1365 in 2005, 2009, 2017)

The Y-Chart

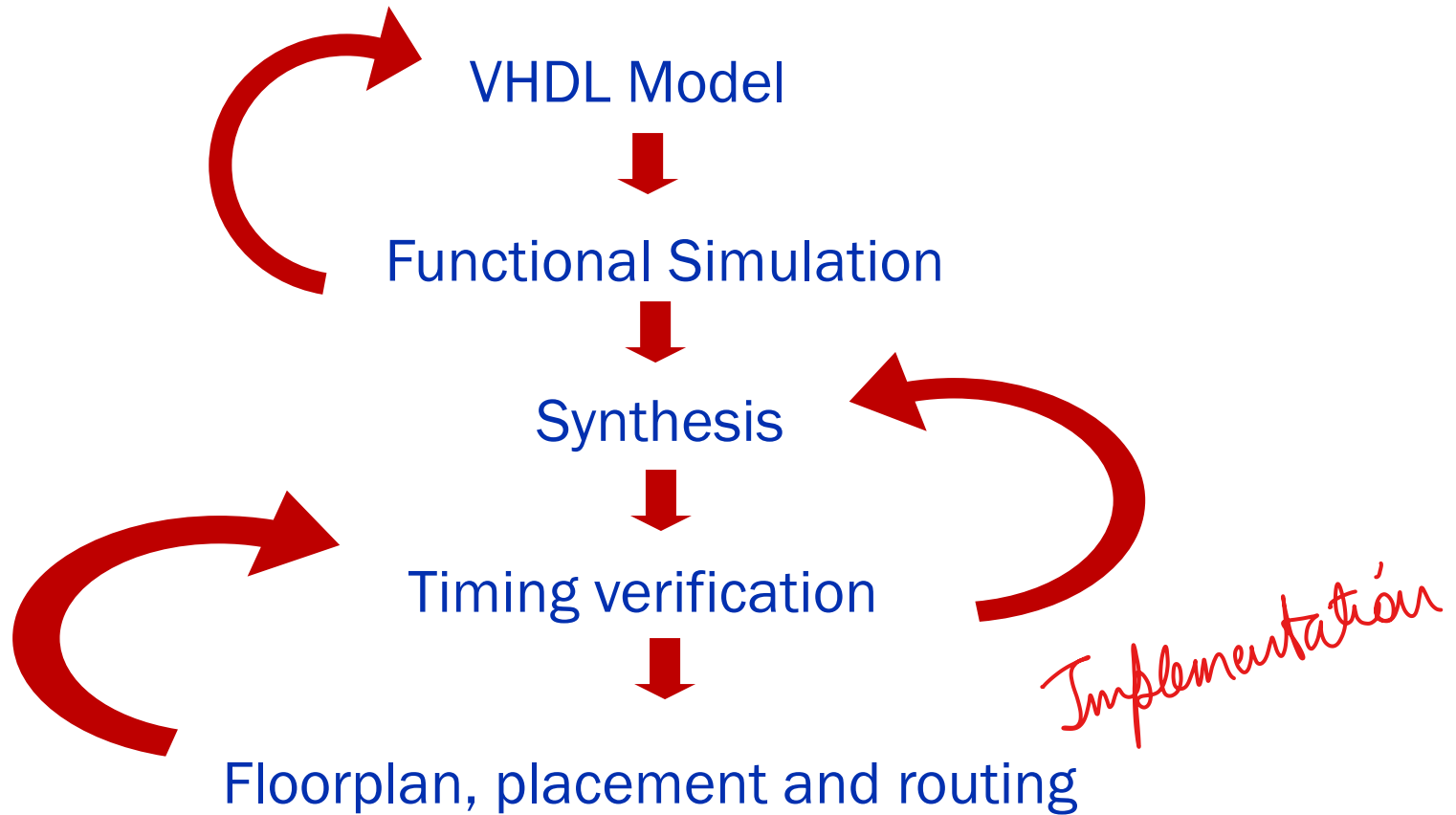


VHDL modeling capability



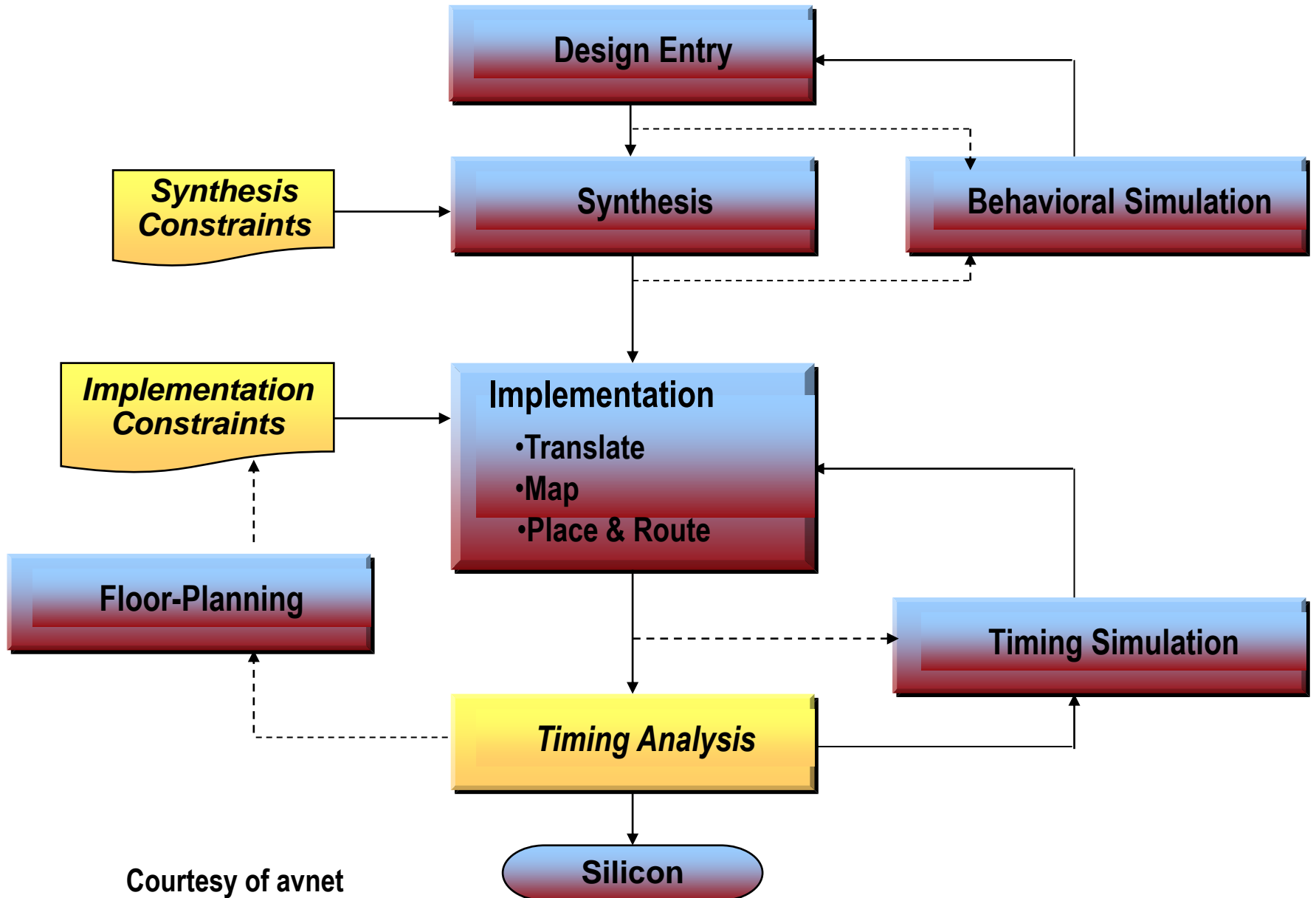
Develop Gate, Block and ASIC level VHDL models

Top-down VHDL-based VLSI design methodology



EL 5473 (bottom up) vs EL 5493 (Top down)

Xilinx Design Process



Courtesy of avnet

Tools we will be using...

- **Xilinx Vivado Design Suite HLx**
 - Used for synthesis, Place and Route, and programming FPGAs
 - Also contains a simulator
 - Also contains Vivado HLS tool for C-based design
 - <https://www.xilinx.com/products/design-tools/vivado.html>
 - Accessible through **Han Solo** (use FastX or ssh and X11 forwarding)
- **Mentor ModelSim (optional)** *for DFT Design for Testability*
 - Used for functional and timing simulations
 - ModelSim provides a complete HDL simulation environment that enables you to verify functional and timing models of your design, and HDL source code.
 - Available only on Han Solo
- **Textbook: check the syllabus for a list**
 - Xilinx Manuals, User Guides, and course notes provided by the instructor.
- **Xilinx Basys/Nexys FPGA boards**
 - These can be purchased from Digilent
 - **Not needed for Fall 2020**

Real World – High Level Design Flow

1. Design Specification –

- Specifying the behavior expected of the final design.
- Designer puts enough detail into the spec. so that the design can be built.
- Coordinated with software and other devices that communicate with the IC.

2. HDL Capture –

- Designer enters VHDL code for entities of the design and checks them for correct syntax.

3. RTL Simulation – *Register Transfer level*

Using Modelsim

- Verify the correctness of the RTL VHDL description
- The VHDL simulator reads the VHDL description, compiles it into an internal format, and then executes the compiled format using test vectors.
- • Designer can look at the output of the simulation and determine whether or not the design is initially working properly.
- The designer then loads the VHDL into an automatic testbench that thoroughly applies stimulus and checks the results.
- The designer runs the simulation for as long as needed to generate enough output data to determine if the design is correct. At the beginning of the design process, this may be only a few vectors to make sure the design resets properly. But later, more and more of the vectors are run as the design starts to function properly.

Real World – High Level Design Flow

Using Xilinx

4. RTL Synthesis –



- The VHDL synthesis tools convert the VHDL description into a netlist in the target FPGA or ASIC technology.
- The VHDL synthesis step is to create a design that implements the required functionality and matches the designer's constraints in speed, area, or power.
- The synthesis tools generate a timing report and area report
 - Timing report – shows the timing of the critical paths of the design. The designer examines the timing of the critical paths closely because these paths ultimately determine how fast the design can run.
 - Area report – Shows the designer how much of the resources of the chip the design has consumed. The designer can tell if the design is too big for a particular chip and the designer needs to target a larger chip.

Real World – High Level Design Flow

Functional Gate Simulation

- Designer may do a quick check on the output of the synthesis tool to make sure that the synthesis tool produced a design that is functionally correct.
- If proper design rules are followed for the input VHDL description, the synthesis tool should never generate an output that is functionally different from the RTL VHDL input, unless the tool has a bug.
- To do this, the designer reads the output VHDL netlist from the synthesis tool, plus a library of the synthesis primitives into the VHDL simulator and runs the simulation using the RTL verification vectors.
- For a completely functional simulation no timing is back annotated.

Real World – High Level Design Flow

6. Place and Route

- Place and route tools are used to take the design netlist and implement the design in the target technology device.
- P&R tools place each primitive (gate, LUT, etc.) from the netlist into an appropriate location on the target device and then route signals between the primitives to connect the devices according to the netlist.
- P&R tools are architecture and device dependant. FPGA vendors provide the tools because the differences in architectures are large enough that writing a common tool for all architectures would be difficult.
- Inputs to the P&R tools are the netlist (EDIF), timing constraints and placement constraints.
 - Timing constraints – Gives the P&R tools an indication of which signals have critical timing associated with them and to route these nets in the most timing efficient manner.
 - Placement constraints – Some P&R tools allow the designer to specify the placement of large parts of the design. This process is also known as floorplanning. Floorplanning allows the designer to pick locations on the chip for large blocks of the design so that routing wires are as short as possible.

The other output from the P&R tools is a file used to generate the timing file. This file describes the actual timing of the programmed FPGA device of the final design. The common format is SDF (standard delay format).

Real World – High Level Design Flow

7. Post Layout Timing Simulation

- After the place and route process has completed, the designer will want to verify the results of the place and route process.
- This simulation combines the netlist used for P&R with the timing file (SDF) from the place and route process into a simulation that checks both the functionality and timing of the design.
- Post route gate-level simulation, if done properly, also uses the same simulator as the RTL simulation. (Same testbench)

8. Static Timing

- For designs of 10k-100k gates, post route timing simulation can be a good method of verifying design functionality and timing. However, as designs gets larger, or if the designer does not have test vectors, the designer can use static timing analysis to make sure the design meets timing requirements.
- A static timing analyzer traces each path in the design and keeps track of the timing from a clock edge or an input. A timing report is generated.

Modelsim

Real World – High Level Design Flow

9. On chip Debug –

- This technique provides the designers with the ability to debug their design in the target system, at target speed, at the VHDL RTL level.
- The VHDL for the device is read into a tool that automatically creates and inserts a small debug core into the device that probes internal signals.
- The debug core is created based on information from the designer about what signals are to be probed.
- The debug core communicates through the JTAG port on the device to an HDL debugger executing on a host platform.
- The HDL debugger sends and receives data from the debug core and displays this data in context with an HDL for the design.
- Waveforms of the internal device can also be displayed, providing the ability to trace down problems in the design.

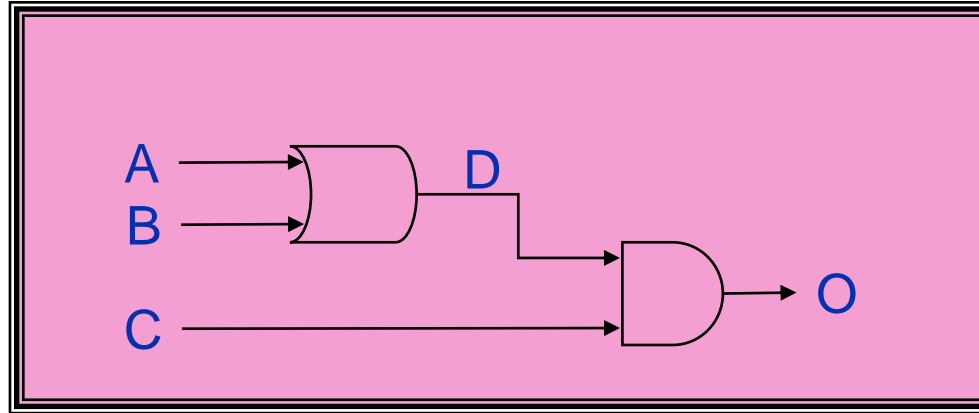
Modeling a digital design (in VHDL)

- ✓ Interaction with the external world (**entity declaration**)
 - ✓ names of ports
 - ✓ types: bit, std_logic_vector, integer...
 - ✓ direction: in, out
 - ✓ width
- ✓ What does it do? (**architecture**)
 - ✓ Structure
 - ✓ Function

Modeling a digital design (in Verilog)

- ✓ Interaction with external world (**module declaration**)
 - ✓ names of ports
 - ✓ types: wire, reg, integer, real...
 - ✓ direction: input, output
 - ✓ width
- ✓ What does it do? (**module body**)
 - ✓ Structure
 - ✓ Function

VHDL Entity declaration



entity name **port name** **port direction**

```
entity or_and is
  port ( a, b, c: in bit;
         o: out bit );
end or_and;
```

port type

Diagram illustrating the mapping of VHDL entity declaration syntax to its components:

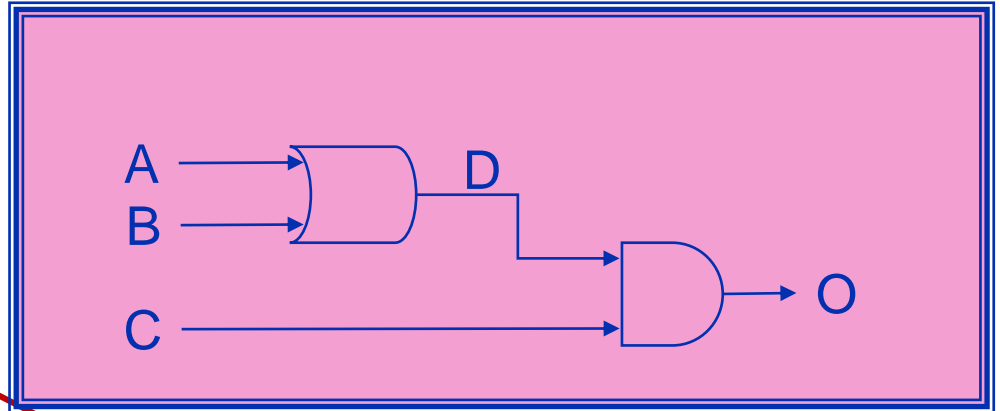
- entity name** points to `or_and` in `entity or_and is` and `or_and` in `end or_and;`.
- port name** points to `a, b, c` in `port (a, b, c: in bit;` and `o` in `o: out bit);`.
- port direction** points to `in` in `in bit;` and `out` in `o: out bit);`.
- port type** points to `bit` in `in bit;` and `bit` in `o: out bit);`.

VHDL Architecture body

architecture name

entity name

```
architecture Do_it of or_and is
  signal D: bit;
begin
  D <= A or B;
  O <= D and C;
end Do_it;
```



```
architecture Do_it of or_and is
  signal D: bit;
begin
  O <= D and C;
  D <= A or B;
end Do_it;
```

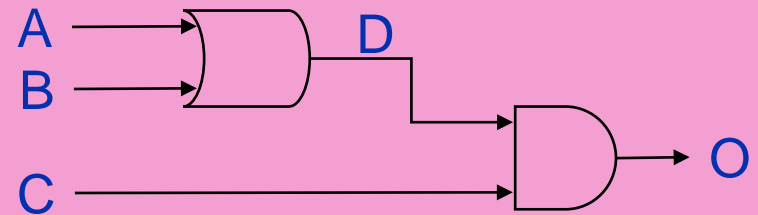
- ✓ Concurrent Statements
 - ✓ can be synthesized and simulated
 - ✓ order of statements is not important

OR-AND gate VHDL Model

```
library IEEE;  
use IEEE.std_logic_1164.all;
```

```
entity or_and is  
  port ( a, b, c: in bit;  
         o: out bit );  
end or_and;
```

```
architecture Do_it of or_and is  
  signal D: bit;  
begin  
  D <= A or B;  
  O <= D and C;  
end Do_it;
```



```

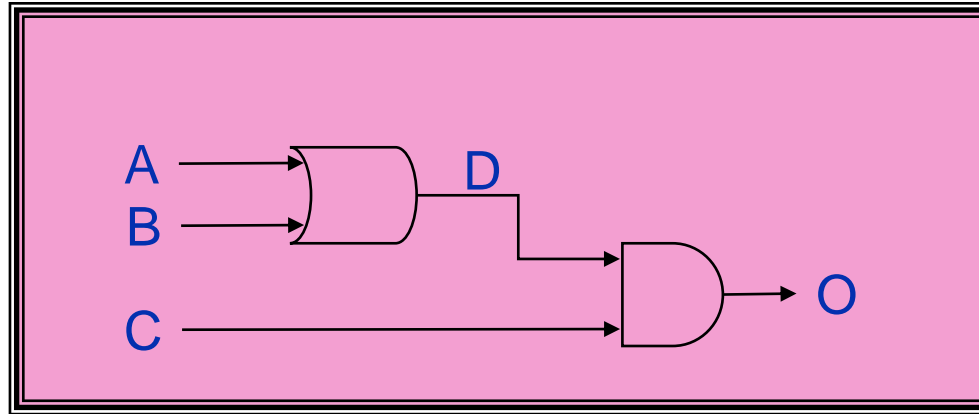
LIBRARY      IEEE;
USE          IEEE.STD_LOGIC_1164.ALL;
ENTITY xyz IS
  PORT (a, b, enable: in bit;
        z: out bit_vector(3 DOWNT0 0)
        );
END xyz;

ARCHITECTURE rtl OF xyz IS
  Signal abar, bbar: bit;
BEGIN
  z(3)<= not (a and b and enable);
  z(0)<= not(abar and bbar and enable);
  abar <= not a;
  z(2)<= not (a and bbar and enable);
  bbar <= not b;
  z(1)<= not (abar and b and enable);
END rtl;

```

Draw the gate level diagram for this VHDL model

Verilog Entity declaration



module name **port names**

port direction **module** or_and(

input wire a, b, c,

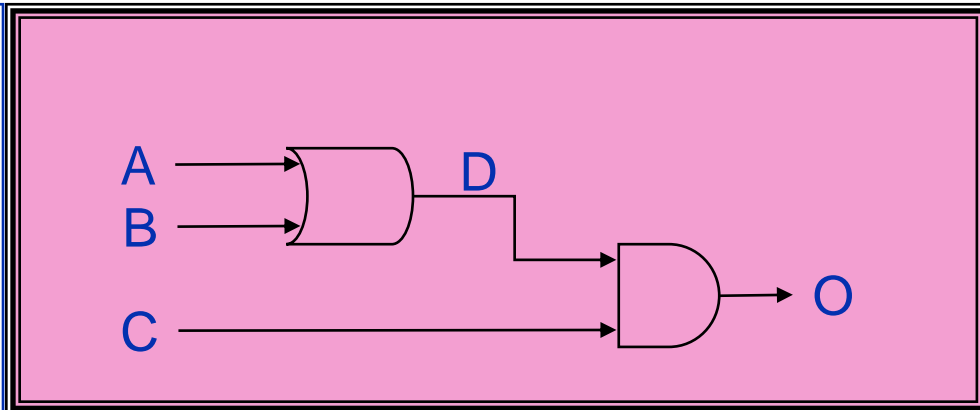
output wire o

);

port type

OR-AND gate Verilog Model

```
module or_and(  
    input wire a,b,c,  
    output wire o  
);  
wire d;  
  
assign d = a | b;  
assign o = d & c;  
  
endmodule
```



Declarations of intermediate signals are optional, but recommended (clean code)

As in VHDL, the assignments are concurrent – can be in any order

Verilog is often more concise than the equivalent VHDL.

```

module xyz (
    input wire a, b, o_e,
    output wire x,y,z
);

```

```

    wire t;

```

```

    assign x = !(a & b) & o_e;

```

```

    assign y = (a ^ b) & o_e;

```

```

    assign t = !a | b;

```

```

    assign z = (t | y) & o_e;

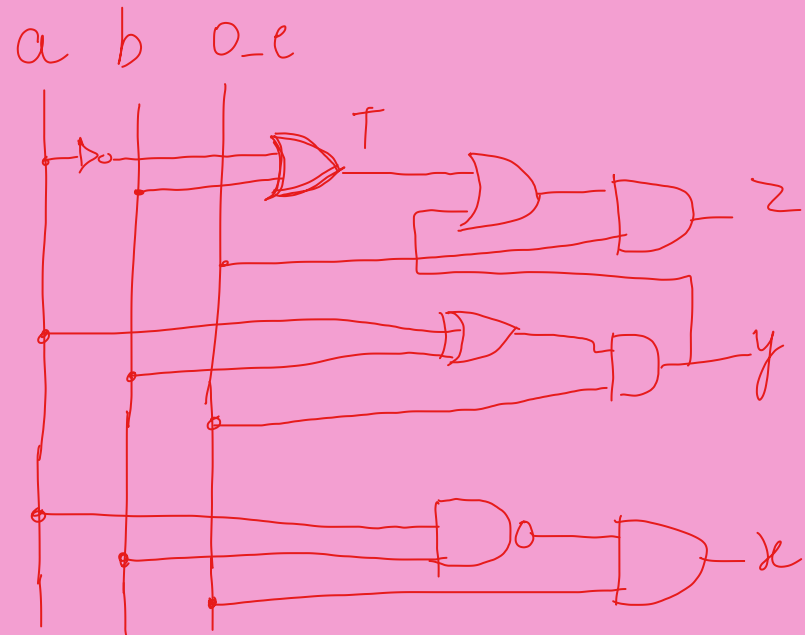
```

```

endmodule

```

XOR



Draw the gate level diagram for this Verilog model

VHDL (circular) left rotate by one bit (4-bit input)

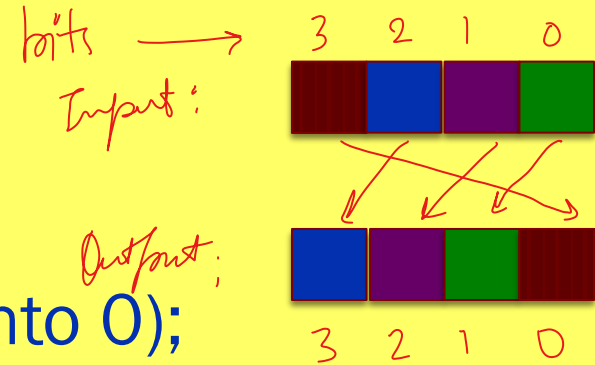
```
library IEEE;  
use IEEE.std_logic_1164.all;
```

```
entity Circular_rot_left_1 is  
port (In : in std_logic_vector(3 downto 0);  
      Out : out std_logic_vector(3 downto 0)  
);
```

```
end Circular_rot_left_1;
```

*⇒ This is just wiring no circuit b/c
this is static shifting. (NOT Data Dependent)*

```
architecture Do_it of Circular_rott_left_1 is  
begin  
  Out(3 downto 1) <= In(2 downto 0);  
  Out(0) <= In(3);  
end Do_it;
```



*In [2:0] inclusive
Verilog*

*Out(0) <= In(3)
Out(1) <= In(0)
Out(2) <= In(1)
Out(3) <= In(2)*

VHDL (circular) left rotate by one bit (28-bit inp)

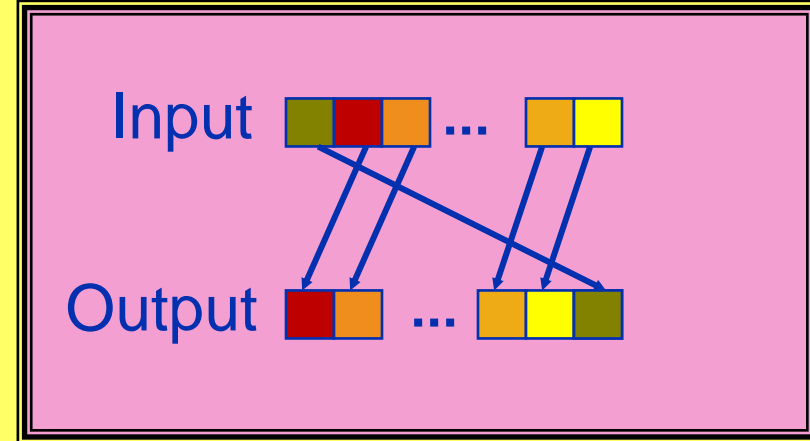
```
library IEEE;  
use IEEE.std_logic_1164.all;
```

```
entity Circular_rot_left_1 is  
port (
```

```
    In : in std_logic_vector(27 downto 0);  
    Out: out std_logic_vector(27 downto 0));  
end Circular_rot_left_1;
```

```
architecture Do_it of Circular_rott_left_1 is  
begin
```

```
    Out(27 downto 1) <= In(26 downto 0);  
    Out(0) <= In(27);  
end Do_it;
```



3 factors with which
we can play →

Direction :

Size of bus

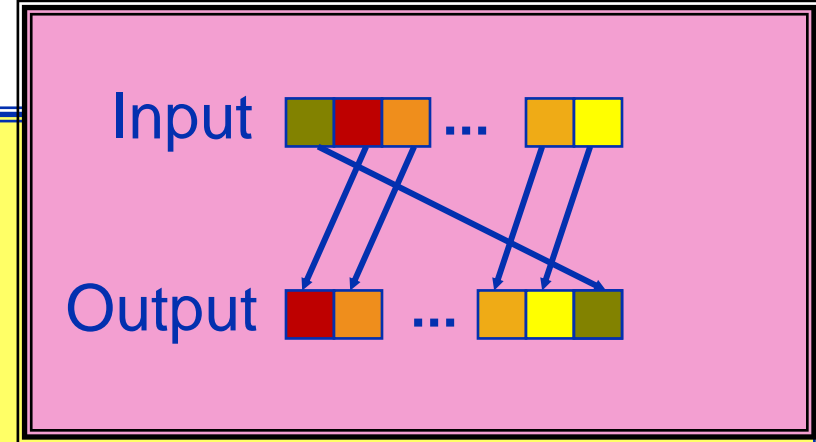
Amount of Rotate

Barrel Shifter:

Inputs

Verilog (circular) left rotate by one bit (28-bit inp)

```
module Circular_rot_left_1 (  
    input wire [27:0] In,  
    output wire [27:0] Out  
);  
  
    assign Out[27:1] = In[26:0];  
    assign Out[0] = In[27];  
  
endmodule
```



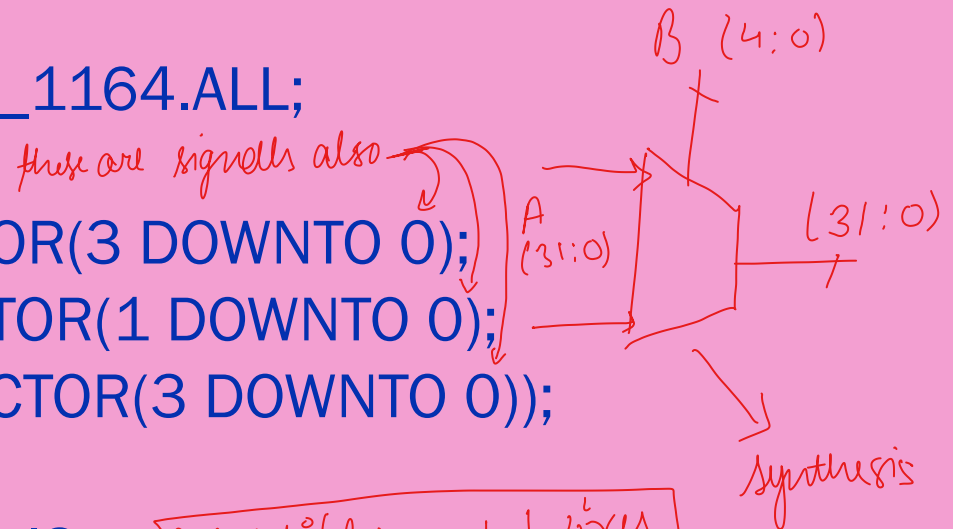
VHDL Data dependent circular left rotate

$O = A \lll B$ (4-bit input)

```
LIBRARY      IEEE;
USE          IEEE.STD_LOGIC_1164.ALL;

ENTITY leftrotate IS
    PORT (a: in STD_LOGIC_VECTOR(3 DOWNTO 0);
          b: in STD_LOGIC_VECTOR(1 DOWNTO 0);
          o: out STD_LOGIC_VECTOR(3 DOWNTO 0));
END leftrotate;

ARCHITECTURE rtl OF leftrotate IS
BEGIN
    WITH b(1 DOWNTO 0) SELECT
        o <= a(2 DOWNTO 0) & a(3) WHEN "01",
            a(1 DOWNTO 0) & a(3 DOWNTO 2) WHEN "10",
            a(0) & a(3 DOWNTO 1) WHEN "11",
            a WHEN OTHERS;
END rtl;
```



Multiplexer + Wires

Verilog Data dependent circular left rotate

$O = A \lll B$ (4-bit input)

```
module leftrotate (  
    input wire [3:0] a,  
    input wire [1:0] b,  
    output wire [3:0] o  
);
```

```
    assign o = (b == 2'b01) ? {a[2:0], a[3]} :  
                (b == 2'b10) ? {a[1:0], a[3:2]} :  
                (b == 2'b11) ? {a[0], a[3:1]} :  
                a;
```

```
endmodule
```

C ternary

*Verilog concatenate
{ }*

Data dependent circular left rotate

$O = A \lll B$ (32 bit input)

```
LIBRARY      IEEE;
USE          IEEE.STD_LOGIC_1164.ALL;
ENTITY       leftrotate IS
    PORT (a: in STD_LOGIC_VECTOR(31 DOWNTO 0);
          b: in STD_LOGIC_VECTOR(4 DOWNTO 0);
          o: out STD_LOGIC_VECTOR(31 DOWNTO 0));
END leftrotate;

ARCHITECTURE rtl OF leftrotate IS
BEGIN
```

$O = A \lll B$ (32 bit input)

Swift
WITH b(4 DOWNTO 0) SELECT

**O <= a(30 DOWNTO 0) & a(31) WHEN "00001",
a(29 DOWNTO 0) & a(31 DOWNTO 30) WHEN "00010",
a(28 DOWNTO 0) & a(31 DOWNTO 29) WHEN "00011",
a(27 DOWNTO 0) & a(31 DOWNTO 28) WHEN "00100",
a(26 DOWNTO 0) & a(31 DOWNTO 27) WHEN "00101",
a(25 DOWNTO 0) & a(31 DOWNTO 26) WHEN "00110",
a(24 DOWNTO 0) & a(31 DOWNTO 25) WHEN "00111",
a(23 DOWNTO 0) & a(31 DOWNTO 24) WHEN "01000",
a(22 DOWNTO 0) & a(31 DOWNTO 23) WHEN "01001",
a(21 DOWNTO 0) & a(31 DOWNTO 22) WHEN "01010",
a(20 DOWNTO 0) & a(31 DOWNTO 21) WHEN "01011",
a(19 DOWNTO 0) & a(31 DOWNTO 20) WHEN "01100",
a(18 DOWNTO 0) & a(31 DOWNTO 19) WHEN "01101",
a(17 DOWNTO 0) & a(31 DOWNTO 18) WHEN "01110",
a(16 DOWNTO 0) & a(31 DOWNTO 17) WHEN "01111",
a(15 DOWNTO 0) & a(31 DOWNTO 16) WHEN "10000",**

concatenate VHDL
b

O=A <<<B (32 bit input)

```
a(14 DOWNT0 0) & a(31 DOWNT0 15) WHEN "10001",  
a(13 DOWNT0 0) & a(31 DOWNT0 14) WHEN "10010",  
a(12 DOWNT0 0) & a(31 DOWNT0 13) WHEN "10011",  
a(11 DOWNT0 0) & a(31 DOWNT0 12) WHEN "10100",  
a(10 DOWNT0 0) & a(31 DOWNT0 11) WHEN "10101",  
a(9 DOWNT0 0) & a(31 DOWNT0 10) WHEN "10110",  
a(8 DOWNT0 0) & a(31 DOWNT0 9) WHEN "10111",  
a(7 DOWNT0 0) & a(31 DOWNT0 8) WHEN "11000",  
a(6 DOWNT0 0) & a(31 DOWNT0 7) WHEN "11001",  
a(5 DOWNT0 0) & a(31 DOWNT0 6) WHEN "11010",  
a(4 DOWNT0 0) & a(31 DOWNT0 5) WHEN "11011",  
a(3 DOWNT0 0) & a(31 DOWNT0 4) WHEN "11100",  
a(2 DOWNT0 0) & a(31 DOWNT0 3) WHEN "11101",  
a(1 DOWNT0 0) & a(31 DOWNT0 2) WHEN "11110",  
a(0) & a(31 DOWNT0 1) WHEN "11111",  
a WHEN OTHERS;
```

```
END rtl;
```

Homework 1: task 1

- Part 1: Set up Vivado Tools
- Through VPN + Han Solo (Preferred)
- On your own machines
- Xilinx offers a free version of the tools that you can download and install on your own computers.
 - Vivado can be installed on Windows and Linux platforms
 - If you use a Mac, you will need to use BootCamp or set up a Virtual Machine with a Windows or Linux guest (VirtualBox and VMware (Workstation Player) are free options).

Homework 1: task 2

Tasks (using both VHDL and Verilog):

- Implement and Simulate a design that can perform the 32-bit Data Dependent Left Rotate
- Implement and Simulate a design that can perform the 32-bit Data Dependent Right Rotate (i.e., you cannot hard code the shift amount)
 - Functionally simulate the model
 - Draw and explain the modeled design
- Film a short video explaining the simulation of 32-bit Data Dependent Right Rotate

VDHL NANDXOR Model

Entity NANDXOR is

```
port (A,B, C: in bit; D: out bit);  
end NANDXOR;
```

architecture DELTA of NANDXOR is

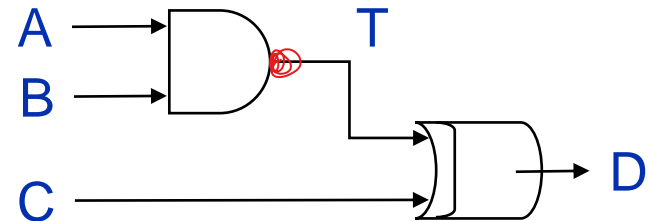
```
signal T : bit;
```

```
begin
```

```
    T <= A nand B;
```

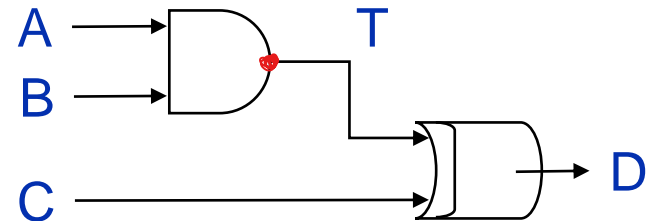
```
    D <= T xor C;
```

```
end DELTA;
```



Verilog NANDXOR Model

```
module NANDXOR (  
    input wire A,B, C,  
    output wire D  
);  
  
    wire T;  
    assign T = !(A & B);  
    assign D = T ^ C;  
  
endmodule
```



Functional Simulation of NANDXOR

```
--NANDXOR2
```

```
--force file
```

```
force A 0 0;
```

```
force A 1 20;
```

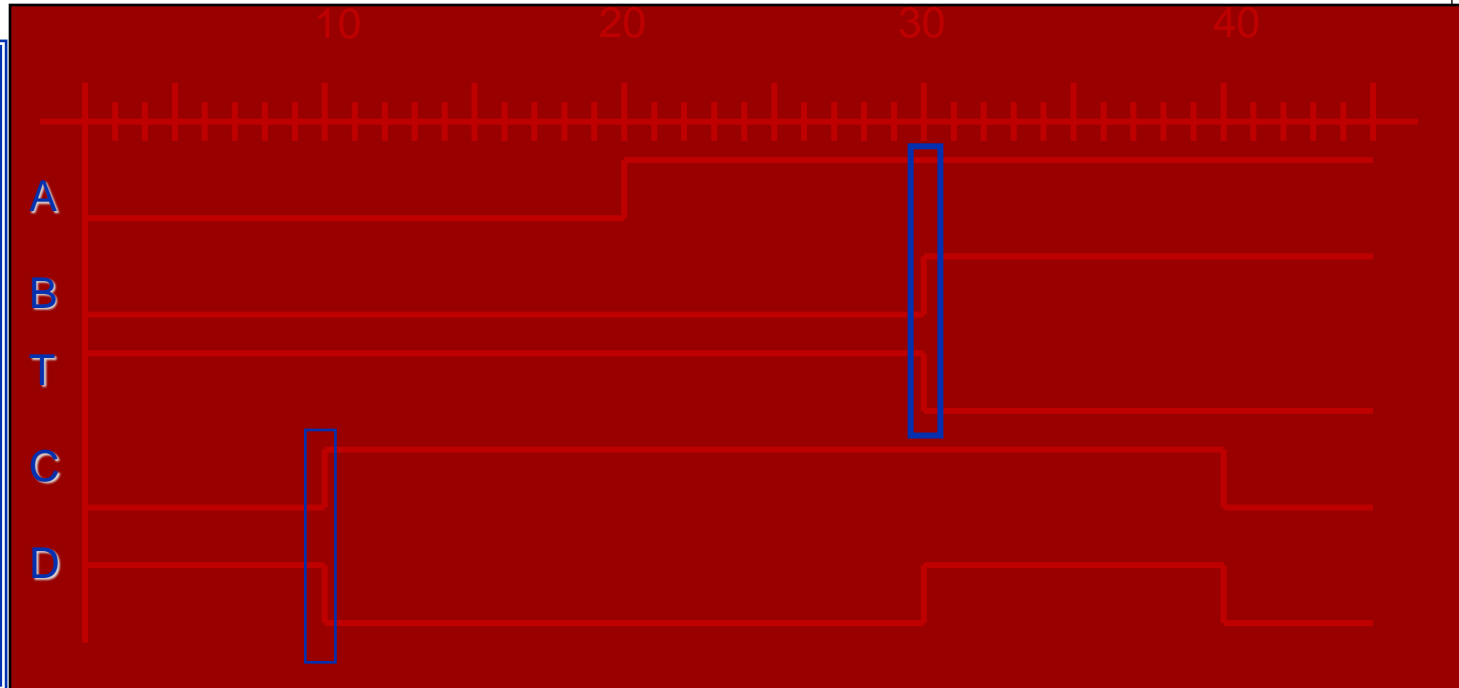
```
force B 0 0;
```

```
force B 1 30;
```

```
force C 0 0;
```

```
force C 1 10;
```

```
force C 0 40;
```



VHDL Signal Data Object

- ✓ Signal

- ✓ Models a physical wire
- ✓ If not initialized, default value is the left-most value of its type
- ✓ Ports are signals
- ✓ "<=" is used for signal assignment

VHDL Basic Data Types

✓ Some Predefined VHDL Data Types

- ✓ Bit: '0' or '1'
- ✓ Boolean: FALSE or TRUE
- ✓ Integer: range $-(2^{31}-1)$ to $+(2^{31}-1)$
- ✓ Std_logic:???

-- logic state system (unresolved) -----

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

);-- attribute

ENUM_ENCODING of std_ologic: type is "U D 0 1 Z D 0 1 D";

-- *** industry standard logic type *** -----

----- SUBTYPE std_logic IS resolved

std_ulogic;

VHDL Data Objects

- ✓ Constant
 - ✓ Cannot be updated
 - ✓ Can be declared inside or outside a process
- ✓ Variable
 - ✓ Can be initialized
 - ✓ Updated as soon as a variable assignment stmt is executed
 - ✓ ":= " is used for variable assignment
 - ✓ Used for local storage in processes, procedures and functions
- ✓ Signal
 - ✓ Models a physical wire
 - ✓ If not initialized, default value is the left-most value of its type
 - ✓ Ports are signals
 - ✓ "<=" is used for signal assignment

VHDL Basic Data Types

- ✓ Some Predefined VHDL Data Types
 - ✓ Bit: '0' or '1'
 - ✓ Boolean: FALSE or TRUE
 - ✓ Integer: range $-(2^{31}-1)$ to $+(2^{31}-1)$
- ✓ Enumerated Data Types
 - ✓ Example 1: type color is (red, green, yellow);
signal A: color;
A <= green;
 - ✓ Some Attributes
 - ✓ color'left is red
 - ✓ color'right is yellow
 - ✓ Example 2: type state_type is (s0, s1, s2, s3);
signal state: state_type := s1;

VHDL Record Data Types

```
constant length: integer := 8;  
subtype byte_vector is bit_vector (length-1 downto 0);  
type byte_and_ix is record  
    byte: byte_vector;  
    ix: integer range 0 to length;  
end record;
```

```
signal x, y, z: byte_and_ix;
```

Write (read) to a record by field name

```
    x.byte <= "11001100";
```

```
    x.ix <= 2;
```

Write (read) to a record by record name

```
    y <= x;
```

VHDL Array Data Type

✓ Unconstrained Arrays

- ✓ type std_logic_vector is array (natural range<>) of std_logic;
- ✓ type bit_vector is array (integer range <>) of bit;
- ✓ variable sample_vector: bit_vector (2 downto -3);

✓ Constrained Arrays

- ✓ subtype key56set is std_logic_vector(55 downto 0);
- ✓ type key_56_tbl is array (0 to 5) of key56set;

✓ Array Attributes

- ✓ sample_vector'left is 2 sample_vector'right is -3
sample_vector'high is 2 sample_vector'low is -3
- ✓ sample_vector'length is 5 sample_vector'range is (2 downto -3)
- ✓ sample_vector'reverse_range is (-3 to 2)

55	0
	1
	2
	3
	4
0	5

Verilog Wire Data Object

✓ Wire

- ✓ Models a physical wire
- ✓ If not initialized, default value is X
- ✓ Ports are (usually) wires
- ✓ "assign [name] =" is used for wire assignment

Verilog Basic Data Types

- ✓ Verilog is more flexible than VHDL
- ✓ There are two major groups of types, 'net' and 'variable'
- ✓ Net types are like wires, and can be set like VHDL's std logic
 - ✓ E.g. wire x; assign x = 'Z';
- ✓ Variable types are used for more complex types
 - ✓ E.g. reg, integer, real, realtime, time

Verilog Data Objects

- ✓ Constant (``define`, `param`, `localparam`)
 - ✓ Cannot be updated
 - ✓ Can be declared inside or outside a module (depends on scope)
- ✓ Reg-types *Variable in VHDL*
 - ✓ Can have initial values
 - ✓ Updated as soon as a variable assignment stmt is executed
 - ✓ `“:=”` or `“<=”` is used for blocking/non-blocking assignment
 - ✓ Used for local storage in processes, procedures and functions
- ✓ Wire
 - ✓ Models a physical wire
 - ✓ Cannot be initialized
 - ✓ Ports are usually wires
 - ✓ `“assign [name] =”` is used for wire assignment

Verilog arrays

- ✓ Arrays

- ✓ `reg [ARRAY_WIDTH-1:0] big_array [0:ARRAY_LENGTH-1];`

- ✓ No standard array attributes in Verilog

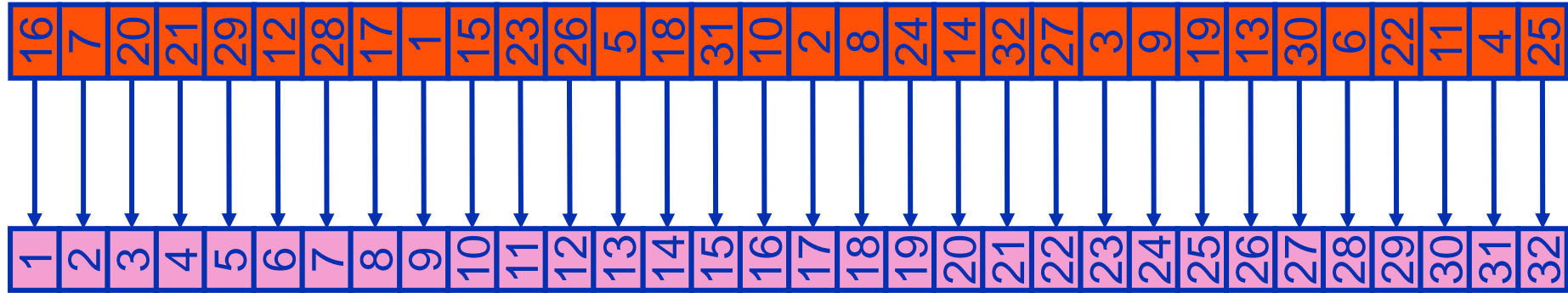


Verilog limitations

- ✓ No enumerated data types
- ✓ No attributes
- ✓ No records
- ✓ SystemVerilog introduces these features, but they aren't in pure Verilog

Permutation

Input



Output

Using Array attributes in VHDL

```
library IEEE; use IEEE.std_logic_1164.all;
entity P_box is port (
    In32 : in std_logic_vector(31 downto 0);
    Out32 : out std_logic_vector(31 downto 0));
end P_box;
architecture Do_it of P_box is
type In32array_type is array (0 to 31) of integer;
constant In32table : In32array_type := In32array_type '(
    15, 6, 19, 20, 28, 11, 27, 16, 0, 14, 22, 25, 4, 17, 30, 9,
    1, 7, 23, 13, 31, 26, 2, 8, 18, 12, 29, 5, 21, 10, 3, 24);
begin
    process (In32)
    begin
        for i in In32table'Range loop
            Out32(i) <= In32(In32table(i));
        end loop;
    end process
end Do_it;
```

**Works for any
sized table**



“Equivalent” (Pure) Verilog

```
module P_box (  
    input wire [31:0] In32,  
    output reg [31:0] Out32  
);
```

```
localparam len_table = 32;
```

```
reg [5:0] In32table [0:len_table-1];
```

```
initial begin
```

```
    In32table[0] = 15;
```

```
    In32table[1] = 6;
```

```
    ... (rest of table)
```

```
    In32table[31] = 24;
```

```
end;
```

```
integer i;
```

```
always @(In32) begin
```

```
    for (i = 0; i < len_table; i = i + 1) begin
```

```
        Out32[i] <= In32[In32table[i]];
```

```
    end
```

```
end
```

```
endmodule
```

Array elements
must be initialized
one by one if done
explicitly
(There is also
\$readmemh() magic
function) for tables
in external files)

Table length
stored as localparam

“Equivalent” SystemVerilog

```
module P_box (  
    input wire [31:0] In32,  
    output reg [31:0] Out32  
);  
localparam len_table = 32;  
reg [5:0] In32table [0:len_table-1] = {  
    15, 6, 19, 20, 28, 11, 27, 16, 0, 14, 22, 25, 4, 17, 30, 9,  
    1, 7, 23, 13, 31, 26, 2, 8, 18, 12, 29, 5, 21, 10, 3, 24};  
  
integer i;  
always @(in32) begin  
    for (i = 0; i < len_table; i = i + 1) begin  
        Out32[i] <= In32[In32table[i]];  
    end  
end  
endmodule
```

**Table length
stored as localparam**



Or use SystemVerilog!!

Quick primer: Verilog testbenches

- ✓ Create module with no ports
- ✓ Define regs for inputs to Design Under Test (DUT),
- ✓ Define wires for outputs
- ✓ Define instance of DUT
- ✓ Create an initial block which defines input traces
- ✓ Simulate and check outputs are correct

Testbenches for Verilog Examples

```
module or_and(  
    input wire a,b,c,  
    output wire o  
);  
wire d;  
  
assign d = a | b;  
assign o = d & c;  
  
endmodule
```

```
module test_or_and;  
reg a,b,c;  
wire o;
```

Test I/O signals

```
or_and oa (  
    .a(a), .b(b), .c(c), .o(o)  
);
```

Instance of
Design under test

```
initial begin: TEST_BLOCK
```

Input test traces

```
    a <= 0;  
    b <= 0; //correct output o=0  
    c <= 0;  
    #10;
```

10 unit delay

```
    a <= 1;  
    b <= 0; //correct output o=0  
    c <= 0;  
    #10;  
    //etc: do all 8 possibilities
```

```
end  
endmodule
```

Testbenches for Verilog Examples

```
module test_or_and;
```

```
reg a,b,c;
```

```
wire o;
```

```
or_and oa (
```

```
    .a(a), .b(b), .c(c), .o(o)
```

```
);
```

```
initial begin: TEST_BLOCK
```

```
    a <= 0;
```

```
    b <= 0;
```

```
    c <= 0; //correct output o = 0
```

```
    #10
```

```
    a <= 1;
```

```
    b <= 0;
```

```
    c <= 0; //correct output o = 0
```

```
    #10 a <= 0;
```

```
    b <= 1;
```

```
    c <= 0; //correct output o = 0
```

```
    #10
```

```
    a <= 1;
```

```
    b <= 1;
```

```
    c <= 0; //correct output o = 0
```

```
    #10
```

```
    a <= 0;
```

```
    b <= 0;
```

```
    c <= 1; //correct output o = 0
```

```
    #10
```

```
    a <= 1;
```

```
    b <= 0;
```

```
    c <= 1; //correct output o = 1
```

```
    #10
```

```
    a <= 0;
```

```
    b <= 1;
```

```
    c <= 1; //correct output o = 1
```

```
    #10
```

```
    a <= 1;
```

```
    b <= 1;
```

```
    c <= 1; //correct output o = 1
```

```
end
```

```
endmodule
```

1) How to include internal wires/signals?
2) How to define simulation time
3) VHDL Testbench vs Verilog + B

Testbenches for Verilog Examples

```
module test_Circular_rot_left_1;

reg [27:0] In;
wire [27:0] Out;

Circular_rot_left_1 c (
    .In(In),
    .Out(Out)
);

initial begin: TEST_BLOCK
    In      <=    28'b10101010101010101010101010101010;
    //Out should be 28'b01010101010101010101010101010101
end
endmodule
```

Testbenches for Verilog Examples

```
module test_nandxor;
reg a,b,c;
wire d;

nandxor n (
    .a(a),    .b(b),    .c(c),    .d(d)
);

initial begin: TEST_BLOCK
    a <= 0;
    b <= 0;
    c <= 0; //correct output d = 1
    #10
    a <= 1;
    b <= 0;
    c <= 0; //correct output d = 1
    #10
    a <= 0;
    b <= 1;
    c <= 0; //correct output d = 1
    #10
```

```
    a <= 1;
    b <= 1;
    c <= 0; //correct output d = 0
    #10
    a <= 0;
    b <= 0;
    c <= 1; //correct output d = 0
    #10
    a <= 1;
    b <= 0;
    c <= 1; //correct output d = 0
    #10
    a <= 0;
    b <= 1;
    c <= 1; //correct output d = 0
    #10
    a <= 1;
    b <= 1;
    c <= 1; //correct output d = 1
    #10
end
endmodule
```


Testbenches for Verilog Examples

```
module test_P_box;

reg [31:0] In;
wire [31:0] Out;

P_box p (
    .In32(In),
    .Out32(Out)
);

initial begin: TEST_BLOCK
    In <= 32'hDEADBEEF;
    //Out should be 7b7fc9f7
    #10
    In <= 32'h7b7fc9f7;
    //Out should be 9dc37fff
end
endmodule
```

Quick primer: Verilog testbenches

- ✓ Checking output of simulations is time-consuming
- ✓ Easy to make mistakes
- ✓ Verilog lets you check the outputs automatically
- ✓ Fully automated testing

Auto Testbenches for Verilog

```
module auto_test_nandxor;
```

```
reg a,b,c;  
wire d;
```

```
nandxor n (  
    .a(a),  
    .b(b),  
    .c(c),  
    .d(d)  
);
```

```
initial begin: TEST_BLOCK
```

```
    a <= 0;
```

```
    b <= 0;
```

```
    c <= 0;
```

```
    #10
```

```
    if (d !== 1) $fatal();
```

```
    #10
```

Execution time

Automatically check output,
\$fatal() will terminate sim. on error

```
#10
```

```
a <= 0;
```

```
b <= 1;
```

```
c <= 0;
```

```
#10
```

```
if (d !== 1) $fatal();
```

```
#10
```

```
//etc .....
```

```
a <= 1;
```

```
b <= 1;
```

```
c <= 1;
```

```
#10
```

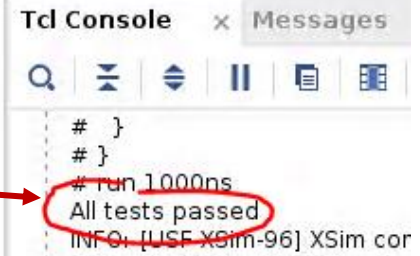
```
if (d !== 1) $fatal();
```

```
$display("All tests passed");
```

```
end
```

```
endmodule
```

Print on completion



The screenshot shows a 'Tcl Console' window with a search bar and several icons. The output text is as follows:

```
# }  
# }  
# Run 1000ns  
All tests passed  
INFO: [USF-XSim-96] XSim cor
```

A red circle highlights the line 'All tests passed'.