**Introduction to FPGA design and Hardware Description Languages**

The development of VHDL was initiated in 1981 by the United States Department of Defence to address the hardware life cycle crisis. The cost of reprocuring electronic hardware as technologies became obsolete was reaching crisis point, because the function of the parts was not adequately documented, and the various components making up a system were individually verified using a wide range of different and incompatible simulation languages and tools. The requirement was for a language with a wide range of descriptive capability that would work the same on any simulator and was independent of technology or design methodology.

VHDL is a structured language meaning that it consists of a common structure for every script divided into boxes that are meaningful itself.

It is used to describe hardware such as the following 4-bit comparator.

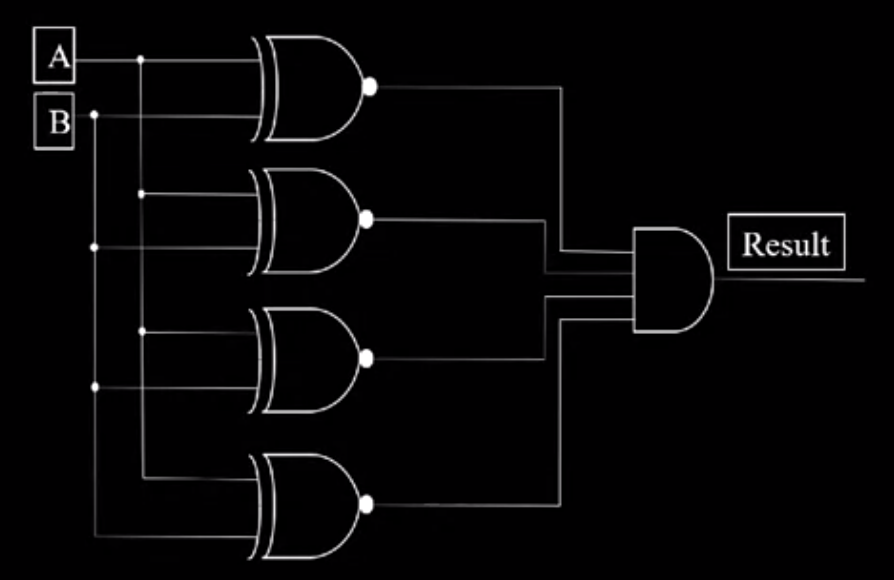


Figure 1. Hardware design of a 4-bit comparator

## FPGA Design Flow

In FPGA, the design flow is the following:

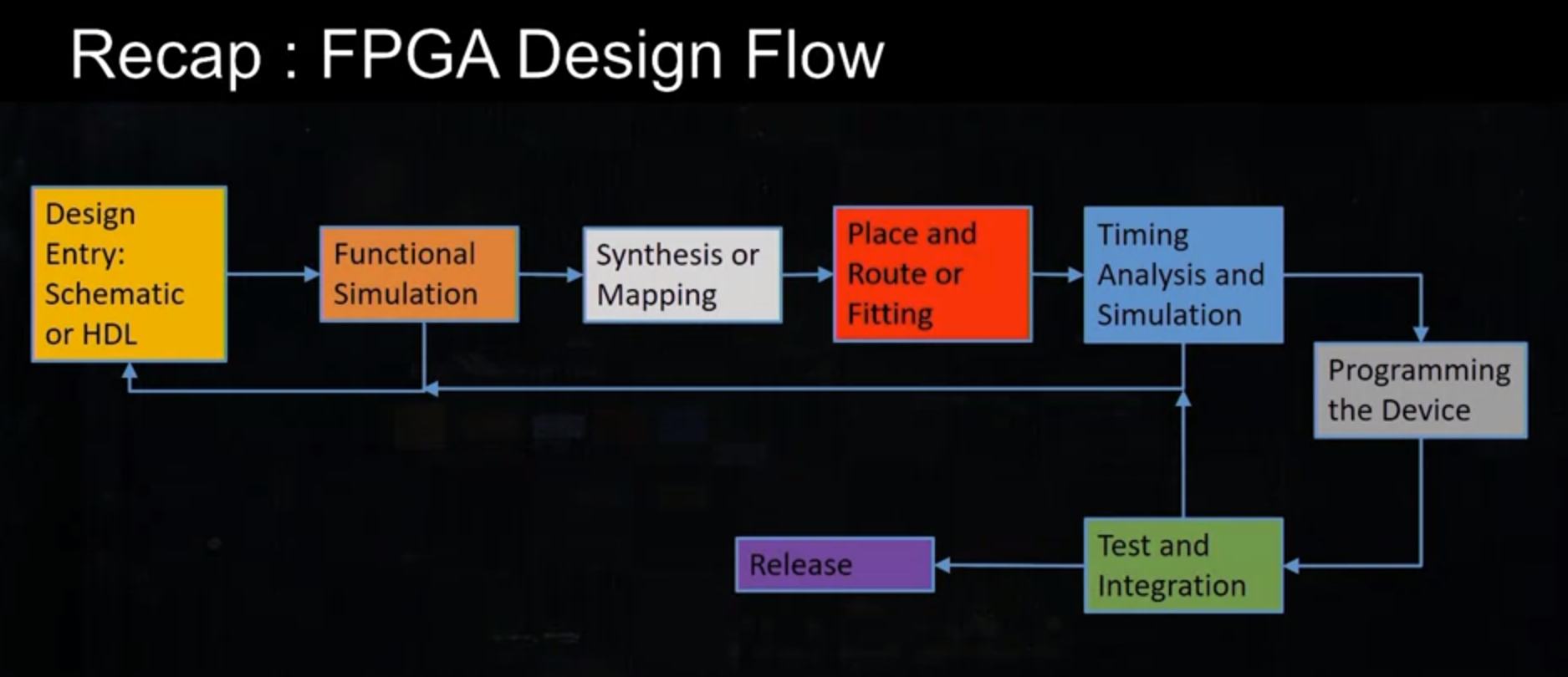


Figure 2. FPGA design flow

**Which is the difference between Schematic and Hardware Desription Languages?**

Schematic is highly correlated to the physical implementation of the circuit since the implementation is based on physical components interconnected. However, for large designs it would be very difficult to transfer the design to FPGA or ASIC. So, using Schematic design entry is easily transferable to other FPGA or ASIC tool flows ? No, **Schematic entry** tools are specific to a vendor tool data entry flow, but **HDLs** are **text and are easily emailed or transferred to other projects**. Basically, an schematic would be difficult to be transferred and understood by any other person while HDL allows the professional to send the design through text.

HDL Key concepts:

* Describe a hardware circuit implementation that the tool chain will interpret and synthesize into FPGA logic cells, to everything is compact and condensed in a logic cell that can be programmable if required.
* HDL is concurrent, not sequential like other languages.
* FPGA logic cells are hardware, they inherently are executed in parallel.
* CPU code is software so it is inherently executed in sequence.
* VHDL is not key sensitive, meaning that variables DATA\_out and data\_Out are the same.

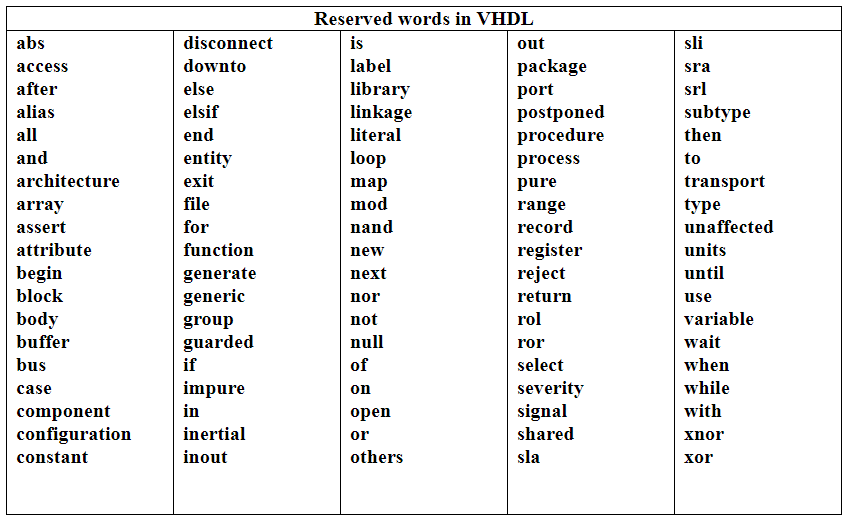
Structure of HDL

VHDL consists of:

Entity: the interface of the system, input and output ports.

Architecture: the logic design of our circuit.

**VHDL Keywords:**



## VHDL Modeling

There are 3 main VHDL modeling methodologies: Structural, Dataflow and Behavioural.

1. Structural Modeling

There are structural keywords that allow the usage of logic elements in the code like library defined primitive gates (and2/or2), Boolean, bitwise logical (and/or) and library user defined functions (and17).

1. Dataflow modeling

Use assignment and select statements for the code.

1. Behavioral modeling

Use assignments with a process, using the sensitivity list of A and B signals for example.

Components of VHDL

Signals are a wire to the entity denoted as std\_logic. The operator signal assignment (<=) initializes a signal. Variables are defined by (:=) operator and it assigns the variable in the process immediately and does not need to wait for an event like a clock edge in the process.

Datatypes in VHDL are divided into:

* Array:
  + String. “abc”
  + Bit\_vector: “1001”
  + Std\_logic\_vector: allows 9 values of 1,0,X,Z,W,L,H,U. i.e. “101Z”
* Scalar
  + Character: ‘a’
  + Bit: ‘1’, ‘0’
  + Std\_logic: ‘1’, ‘0’, ‘X’, ‘Z’
  + Boolean: true, false
  + Real, integer: 3.14 1E+0.27
  + Time: fs, ps, ns, us, ms

Comments are put with two dashes (--) and every sentence must end with semicolon (; ).

Example Codes

1. 2 Bit comparator

A 2 bit comparator take 2 signals of 2 bits and compares the values of the two bits of each signal

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A1A2/B1B2 | 00 | 01 | 10 | 11 |
| 00 | 1 | 0 | 0 |  |
| 01 | 0 | 1 | 0 |  |
| 10 | 0 | 0 | 0 |  |
| 11 | 1 |  |  |  |

Lesson 2. VHDL Logic Design Techniques

In this part of the lesson, we are going to learn about how to implement logic circuits in VHDL, such as AND, NOR, XOR, OR gates. The way those simple logic elements are implemented in VHDL are quite simple. Follow the next example code that illustrates this.

-- Entity

Entity gates is port (

vA, vB : in std\_logic\_vector(3 downto 0);

A, B, C, D : in std\_logic;

W, U, X, Y, Z : out std\_logi;

vX, vY : out std\_logi\_vector(3 downto 0)

);

end entity gates;

-- Architecture

Arhiteture RTL of gates is

Begin

W <= A and B; U<= A nor B; -- AND, NOR

X <= xor D; Y <= xnor D; -- XOR, XNOR

Z <= (A and B) or (C and D); -- AND-OR

vX <= vA and vB; -- Vector bitwise AND

vY <= vA or vB; -- Vector bitwise OR

end architecture RTL;

We can also do a vector reduction that consists of reducing the elements of an array into a single result, being associative and often, but not necessarily, commutative. It is similar to map reduce. It gives back the scalar from an array, basically, adding all the elements into a single element resulted from a sum.

-- Entity

Entity gates is port (

vA, vB, v, vD : in std\_logic\_vector(3 downto 0);

W, U, X, Y, Z : out std\_logi;

);

end entity gates;

-- Architecture

Arhiteture RTL of gates is

Begin

W <= AND\_REDUCE(vA); -- vector reduction AND

U <= NOR\_REDUCE(vB); -- vector reductior NOR

X <= XOR\_REDUCE(vD); -- vector reduction XOR

Y <= OR\_REDUCE(vA) and vB(0);-- OR red, bit AND

Z <= OR\_REDUE(vA and vB); -- bit AND, OR red

end architecture RTL;

Then, for example, in VHDL, you can use either of the following as an XOR reduction for bus A = "1011" : a) z\_out <= XOR ( A ); b) z\_out <= XOR\_REDUCE( A );

2.1. Synchronous Logic: D latch

A classic D Latch represented in Fig. 3, can be modelled easily in VHDL as shown in the code below.

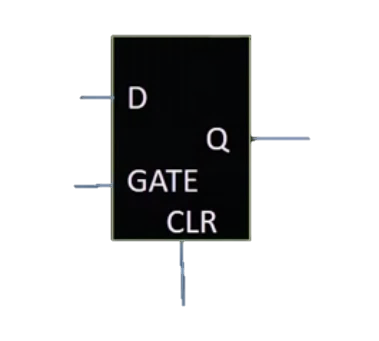


Figure 3. D-Latch

-- Entity

entity DLatches is port (

d, gate, clr : in std\_logic;

q : out std\_logic );

end entity DLatches;

-- Architecture

architecture LArch of DLatches is begin

latch\_proc\_1 : process (gate, d)

begin

if (gate='1') then q <= d;

-- No rising\_edge()

end if;

-- No gate=0 value, so latch inferred

end process latch\_proc\_1;

*-- another Latch example*

latch\_proc\_2 : process (gate, d, clr)

begin

if (clr ='1') then q <= '0';

elsif (gate='1') then q <= d;

end if;

end process latch\_proc\_2;

end architecture LArch;

However, in those examples, there is no clock being an asynchronous circuit. How do we synchronize the circuit? Using a clock as an input

-- Entity

entity DFF is port (

d, clk, reset : in std\_logic;

q : out std\_logic );

end entity DFF;

-- Architecture,

-- could use (clk'event and clk='1')

architecture DFF\_Arch of DFF is

begin dff\_proc\_1 : process (clk)

begin

**if (rising\_edge(clk)) then -- this is the synchronous flip flop**

**(if ( clk'event and clk='1' ) then) – similar implementation of flip-flop.**

if (reset='1') then q <= '0';

-- Sync Reset

else q <= d;

end if;

end if;

end process dff\_proc\_1;

end architecture DFF\_Arch;

Implementation of asynchronous reset on synchronous flip-flop

-- Entity

entity DFF is port (

d, clk, set, reset : in std\_logic;

q : out std\_logic );

end entity DFF;

-- Architecture, Next Slide

-- Architecture

architecture DFF\_Arch of DFF is

begin dff\_proc\_2 : process (clk, set, reset)

begin

if (reset='1') then q <= '0';

-- Async

elsif (rising\_edge(clk)) then

if (set='0') then q <= '1';

-- Sync

else q <= d;

-- Sync

end if;

end if;

end process dff\_proc\_2;

end architecture DFF\_Arch;

2.2. Data register

The Memory Data Register (MDR) or Memory Buffer Register (MBR) is theregister of a computer's control unit that contains the data to be stored in the computer storage (e.g. RAM), or the data after a fetch from the computer storage.

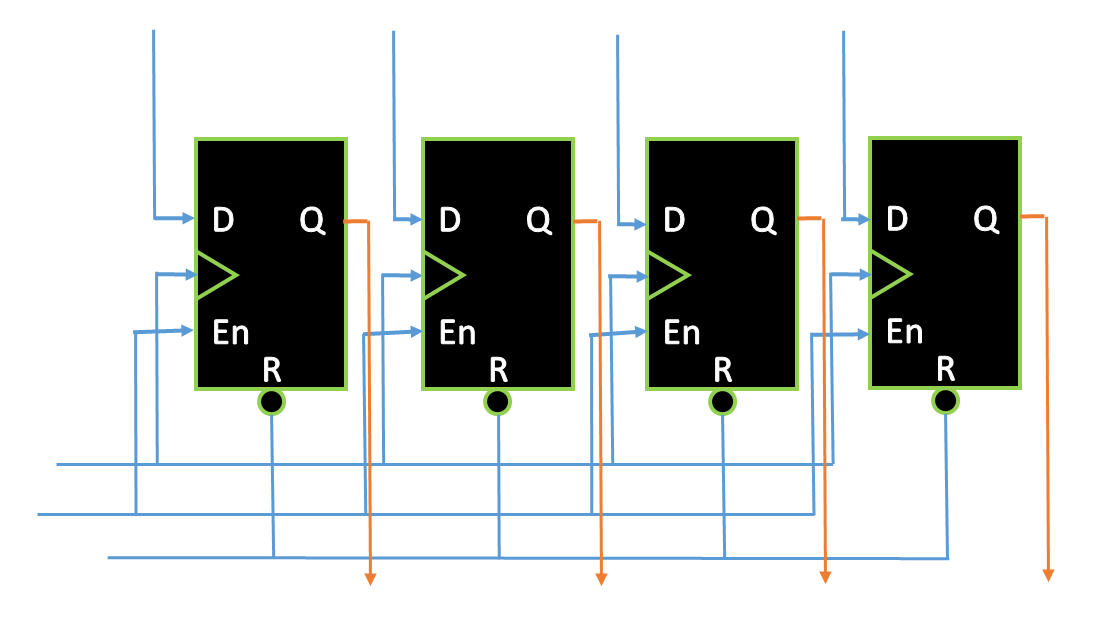


Figure 4. Data register schematic.

The VHDL implementation of a memory register is as follow:

-- Entity

entity Data\_Reg is port (

clk, reset, load : in std\_logic;

d : in std\_logic\_vector(3 downto 0);

q : out std\_logic\_vector(3 downto 0));

end entity Data\_Reg;

-- Architecture

architecture Reg\_Arch of Data\_Reg is

begin dreg\_proc : process (clk, reset, load)

begin

if (reset='0') then q <= "0000";

elsif (rising\_edge(clk)) then

if (load='1') then q <= d;

end if;

end if;

end process dreg\_proc;

end architecture Reg\_Arch;

To implement a shift register in VHDL we will have an asynchronous part of the code, which will be the reset, and a synchronous part of the code, which is the operation of the shifting during the rising edge of the clock. Shift Registers are used for data storage or for the movement of data and are therefore commonly used inside calculators or computers to store data such as two binary numbers before they are added together, or to convert the data from either a serial to parallel or parallel to serial format.

-- Entity

entity Shift\_Reg is port (

clk, reset, shift, d0 : in std\_logic;

q : out std\_logic\_vector(3 downto 0));

end entity Shift\_Reg;

-- Architecture, could SLL or shift\_left(q,1)

architecture SREG\_Arch of Data\_Reg is begin

sreg\_proc : process (clk, reset)

begin

if (reset='0') then q <= "0000";

elsif (rising\_edge(clk)) then

if (shift='1') then

q(0) <= d0;

q(1) <= q(0);

q(2) <= q(1);

q(3) <= q(2);

end if;

end if;

end process sreg\_proc;

end architecture SREG\_Arch;

2.3. Register file

Register file is an array of processor registers in a central processing unit (CPU). Modern integrated circuit-based register files are usually implemented by way of fast static RAMs with multiple ports. Such RAMs are distinguished by having dedicated read and write ports, whereas ordinary multiported SRAMs will usually read and write through the same ports.

The circuit diagram is shown in Figure 5 and the VHDL implementation is shown below.

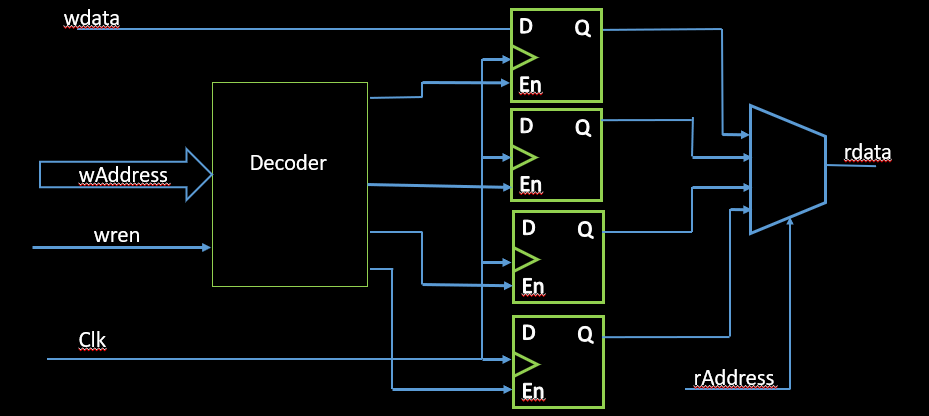


Figure 5. Register File

-- Entity -- use IEEE.numeric\_std.all; integer conversion

entity regFile is

generic (Dwidth : integer := 8;

Awidth : integer := 2 );

port (

clk, wren : in std\_logic;

wdata : in std\_logic\_vector(Dwidth-1 downto 0);

waddr, raddr : in std\_logic\_vector(Awidth-1 downto 0);

rdata : out std\_logic\_vector(Dwidth-1 downto 0) );

end entity regFile;

-- Architecture

architecture RFile\_Arch of regFile is

type array\_type (0 to 2\*\*Awidth-1) of std\_logic\_vector (Dwidth-1 downto 0);

signal array\_reg : array\_type;

begin rf\_proc : process (clk, wren, wdata, waddr, raddr)

begin

if (rising\_edge(clk)) then

if (wren='1') then

array\_reg(to\_integer(unsigned(waddr))) <= wdata;

end if;

rdata <= array\_reg(to\_integer(unsigned(raddr)));

end if;

end process rf\_proc;

end architecture RFile\_Arch;

In the Entity, it is possible to declare generic variables and use them later on the entity and architecture.

2.4. Tri-state and bi-directional bus

A Tri-state Buffer can be thought of as an input controlled switch with an output that can be electronically turned “ON” or “OFF” by means of an external “Control” or “Enable” ( EN ) signal input. This control signal can be either a logic “0” or a logic “1” type signal resulting in the Tri-state Buffer being in one state allowing its output to operate normally producing the required output or in another state were its output is blocked or disconnected. Busses of data are arranged typically in a set of signal with similar name to note that they are coming together, e.g. brainInput [0] to brainInput[7].

-- Entity

entity y\_tri is port (

OE : in std\_logic;

Dout : in std\_logic\_vector(3 downto 0);

Pinout : out std\_logic\_vector(3 downto 0) );

end entity y\_tri;

-- Architecture

architecture tri\_arch of y\_tri is

begin

Pinout <= Dout when (OE='1'),

(others => "ZZZZ") when others ;

end architecture tri\_arch;

VHDL can’t design tri-state buses with insight of the design but tri-state buses are not typically implemented on FPGA. Instead of tri-state buses, multiplexor are more commonly used. Sometimes, bidirectional buses are useful to use the same FPGA pin for input and output. Then, Driving an pin as output '0' or ground while and external device is driving a high voltage '1' value onto the IO pin could burn out the driver on either device. Setting the IO to high impedance or 'Z' will remove the voltage conflict.

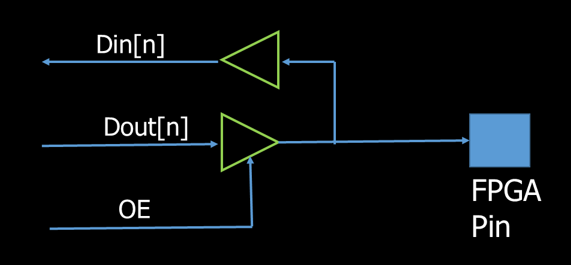


Figure 6. Bi-directional bus.

And the code for the VHdL implementation is.

-- Entity

entity bidir is port (

OE : in std\_logic;

Dout : in std\_logic\_vector(3 downto 0);

Din : out std\_logic\_vector(3 downto 0);

IOpin : inout std\_logic\_vector(3 downto 0) );

end entity bidir;

-- Architecture

architecture bidir\_arch of bidir is

begin bi\_proc : process (OE, Dout)

begin

Din <= IOpin;

if (OE='1') then IOpin <= Dout;

elsif (OE='0') then IOpin <= "ZZZZ";

else IOpin <= "XXXX";

end if;

end process bi\_proc;

end architecture bidir\_arch;

But there is a question now. How to join and split buses of that from multiple FPGA pins like in Figure 7? The next figure shows how this buses are schematically represented and how they are implemented on FPGA.

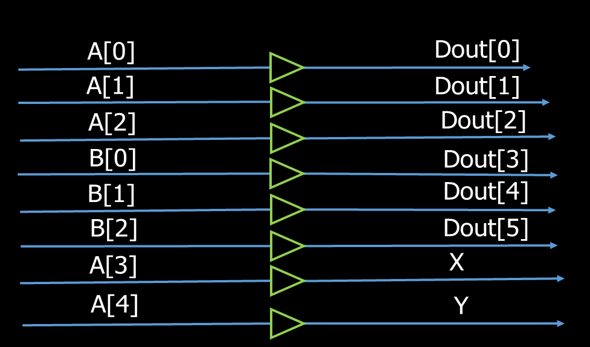


Figure 7. Data bus.

VHDL code.

-- Entity

entity bus\_js is port (

A : in std\_logic\_vector(4 downto 0);

B : in std\_logic\_vector(2 downto 0);

X, Y : out std\_logic;

Dout : out std\_logic\_vector(5 downto 0) );

end entity bus\_js;

-- Architecture

architecture js\_arch of bus\_js is

begin js\_proc : process (A,B) begin

Dout <= (B(2) & B(1) & B(0) &

A(2) & A(1) & A(0));

X <= A(3);

Y <= A(4);

end process js\_proc;

end architecture js\_arch;

3. Modular designs: Components, Generate and Loops in VHDL

In VHDL, we can create a component that can be instantiated in another script trough instantiation rules. For example, if we have a 4-bit adder, we can create a 16-bit adder from the instantiation of those 4-bit adders in multiple adders combined for an array. See next code that illustrates this case.

architecture Add16\_Arch of Add16 is

component Add4 port (

A,B : in std\_logic\_vector(3 downto 0);

Cin : in std\_logic; Cout : out std\_logic);

Sum : in std\_logic\_vector(3 downto 0) );

end component;

Begin

Add4\_u1 : Add4 port\_map (

A=> A(3 downto 0), B=>(3 downto 0), Z=> Cin(0), Sum(3 downto 0) );

...

Add4\_u4 : Add4 port\_map (

A=> A(15 downto 12), B=>(15 downto 12), Z=> Cin(3), Sum(15 downto 12) );

end architecture Add16\_Arch;

**3.1. looping in VHDL**

While and for loops are used in VHDL in similar way to c implementation of those iterative methods.

-- Architecture

architecture Loop\_Arch of my\_loop is begin

while (I <= 8) loop

if (B = '1') then

Z(I) <= A(I);

end if;

I := I + 1;

end loop;

end architecture Loop\_Arch;a

1. **Test benches in VHDL**

A testbench is a program written in any language for the purpose of exercising and verifying the functional correctness of the hardware model as coded. This is also known as a test fixture or a test harness. A testbench is a powerful tool for generating test stimulus and test results.

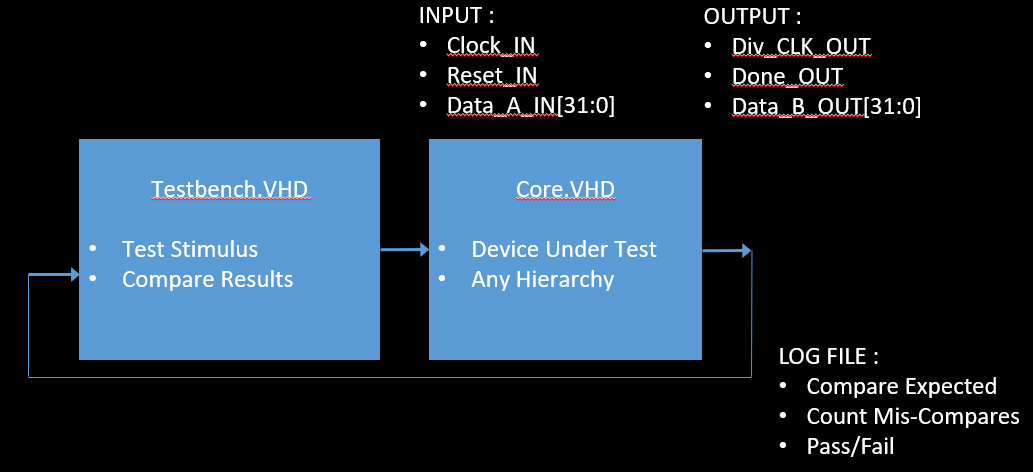


Figure 7. Workflow of testbenches on VHDL.

In a test bench we have:

1. Top-level testbench declaration
2. Stimulus, Response, and Component Signal declarations
3. Component (Device Under Test) instantiations
4. Test Monitor which logs results and reports mis-compares

An example of a testbench is shown in the next code.

-- Entity : no port list !

entity tb\_adder is end entity tb\_adder;

-- Architecture

architecture test\_arch of tb\_adder is

component Add4 port (

Data1,Data2 : in std\_logic\_vector(3 downto 0);

Cin : in std\_logic;

Cout : out std\_logic;

Sum : out std\_logic\_vector(3 downto 0) );

end component Add4;

signal a\_tb, b\_tb : std\_logic\_vector(3 downto 0); -- INPUT

signal Cin : std\_logic; -- INPUT

signal Sum\_tb : std\_logic\_vector(3 downto 0); -- OUTPUT

signal Cout\_tb : std\_logic; -- OUTPUT

signal expect : std\_logic\_vector(3 downto 0); --expected

begin

-- DUT Instantiation

DUT : Add4 port map (

Data1 => a\_tb, Data2 => b\_tb,

Cin => Cin, Cout => Cout\_tb, Sum => Sum\_tb);

-- Stimulus by hand drawn waves, poor coverage

stim\_proc : process begin

wait for 0ns;

Add by hand the stimulus that is wanted to test the code

a\_tb <= "0010"; b\_tb <= "0010"; Cin <= '0'; expect <= "0100";

wait for 10ns;

a\_tb <= "1111"; b\_tb <= "0000"; Cin <= '1'; expect <= "0000";

wait for 10ns;

a\_tb <= "0010"; b\_tb <= "0100"; Cin <= '1'; expect <= "0111";

wait;

end process stim\_proc;

-- Monitor, use ieee.std\_logic\_textio.all;

-- use std.textio.all;

txt\_out : process (Sum\_tb, Cout\_tb)

variable str\_o : line;

begin

write(str\_o, string'(" a=")); write(str\_o, a\_tb);

write(str\_o, string'(" b=")); write(str\_o, b\_tb);

write(str\_o, string'(" cin=")); write(str\_o, Cin);

write(str\_o, string'(" sum=")); write(str\_o, Sum\_tb);

write(str\_o, string'(" cout=")); write(str\_o, Cout\_tb);

write(str\_o, string'(" expect=")); write(str\_o, expect);

assert false report time'image(now) & str\_o.all

severity note;

end process txt\_out;

end architecture test\_arch;

* 1. Synchronous testbench

It is also possible to create test benches for synchronous circuits where we need to control the Clock while the design is being tested. In the next example we have a synchronous testbench for the testing of the circuit design of Fig. 8

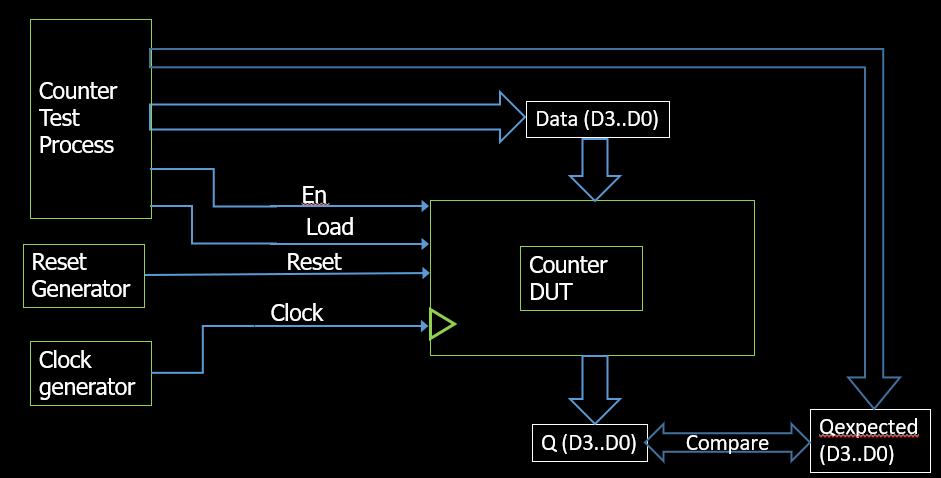


Figure 8. Circuit design of a counter.

-- Testbench Entity : No port list

entity Counter\_tb is end entity Counter\_tb;

-- Testbench Architecture

architecture Counter\_arch of Counter\_tb is

component Counter port (

d : in std\_logic\_vector(3 downto 0);

clk, reset, load, en : in std\_logic;

q : out std\_logic\_vector(3 downto 0));

end component Counter;

constant delay : integer := 10; -- wait

constant n : integer := 4; -- width counter

constant T : time := 20 ns; -- clock period

signal clock : std\_logic := '0'; -- clock generator

signal reset : std\_logic := '0'; -- reset generator

-- Architecture

signal data\_tb : std\_logic\_vector(n-1 downto 0) :=

"0000";

signal load : std\_logic := '0'; -- stimulus

signal en : std\_logic := '0'; -- stimulus

signal q\_tb : std\_logic\_vector(3 downto 0);

-- output

signal check : std\_logic\_vector(n-1 downto 0) :=

"0000"; -- compare to count

1. Memory in VHDL

Memories are a common element in most digital design systems. Earlier in this course, we described the register file circuit in which individual registers were enabled for access by decoding an address. We will extend this to include RAM and ROM memories. Register files are useful constructs that allow addressing of registers. Here we assume each flip-flop represents an n-bit wide register. A RAM, Rapid Access Memorym are elements that keep states of data to be storage during a certain time. The schematic of a memory is shown in Fig. 9 and the example code in VHDL is as follows.

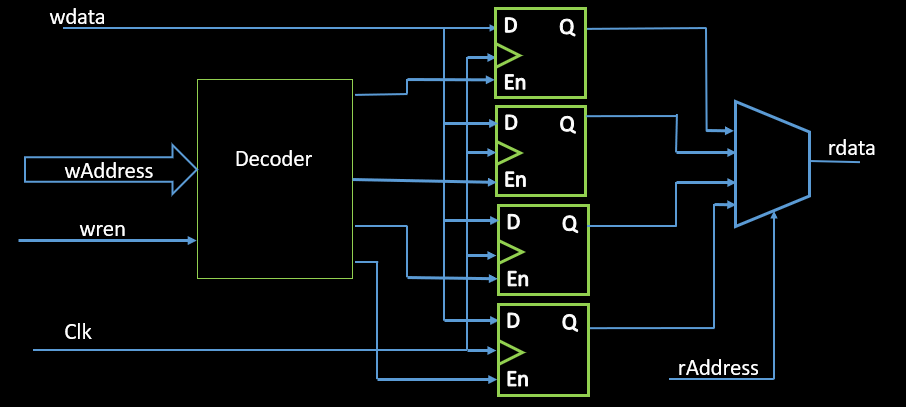


Figure 9. Memory circuit schematic.

The VHDL code is

-- Entity

entity DPRAM is

generic (D\_Width : integer := 8;

A\_Width : integer := 10 ); -- 2\*\*10 = 1024

port (

clk, we : in std\_logic;

d : in std\_logic\_vector(D\_Width-1 downto 0);

w\_add, r\_add : in std\_logic\_vector(A\_Width-1 downto 0);

q : out std\_logic\_vector(D\_Width-1 downto 0) );

end entity DPRAM;

architecture DPR\_Arch of DPRAM is -- Architecture

type ram\_type is array (0 to 2\*\*A\_Width-1) of

std\_logic\_vector (D\_Width-1 downto 0);

impure function read\_file(txt\_file : in string) return ram\_type is

file ram\_file : text open read\_mode is txt\_file;

variable txt\_line : line;

variable txt\_bit : bit\_vector(D\_Width-1 downto 0);

variable txt\_ram : ram\_type;

begin for i in ram\_type'range loop

readline(ram\_file, txt\_line);

read(txt\_line, txt\_bit);

txt\_ram(i) := to\_stdlogicvector(txt\_bit);

end loop; return txt\_ram;

end function;

A ROM memory can also be implemented in VHDL and it is as follows.

entity ROM is -- Entity

generic (D\_Width : integer := 8;

A\_Width : integer := 3 ); -- 2\*\*3 = 8 address

port (

clk : in std\_logic;

addr : in std\_logic\_vector(A\_Width-1 downto 0);

data : out std\_logic\_vector(D\_Width-1 downto 0) );

end entity ROM;

architecture ROm\_Arch of ROM is -- Architecture

signal rom\_d, data\_reg : std\_logic\_vector

(D\_Width-1 downto 0);

signal addr\_sel : std\_logic\_vector (2 downto 0);

begin

addr\_sel <= addr;

rom\_proc : process (clk) begin

data\_reg <= rom\_d;

end process rom\_proc;

lookup\_proc : process begin -- Lookup Table

case(addr\_sel) is

when "000" => rom\_d <= "10000000"; when "100" => rom\_d <= "00000000";

when "001" => rom\_d <= "10101010"; when "101" => rom\_d <= "10011001";

when "010" => rom\_d <= "01010101"; when "110" => rom\_d <= "10000001";

when "011" => rom\_d <= "10000011"; when "111" => rom\_d <= "11110000";

when others => rom\_d <= "00000000"; -- +700 cases possible, X, U

end case;

end process lookup\_proc;

data <= data\_reg;

end architecture ROM;

1. Finite State Machine

Finite state machines are very important part of digital design and software design too. The state machine concept provides a highly reliable, maintainable, and methodical way to design circuits that perform a sequence of operations with great predictability. State machines are always in a known state. Good designs make use of finite state machines. Finite state machines are categorized into two types; **moore or mealy**.

Moore state machines only depend on the output of the state.

Mealy state machines depend on the output of the state and the inputs.

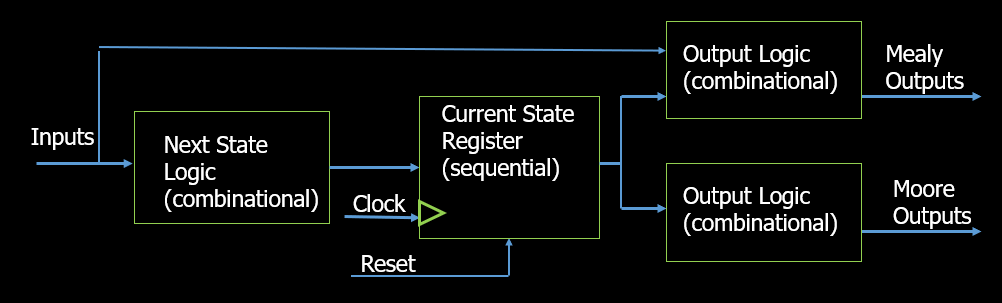
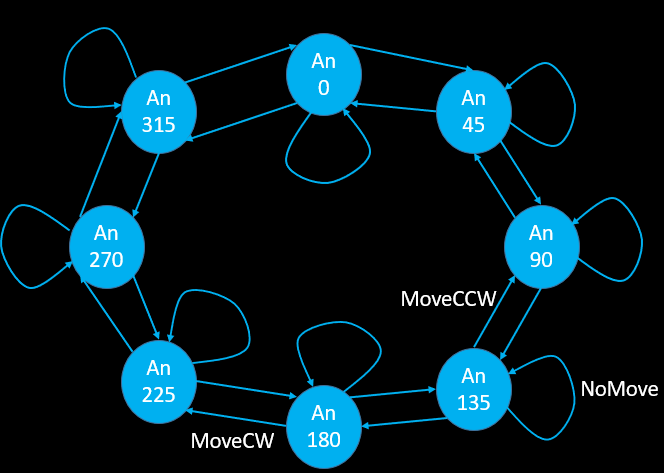


Figure 10. Finite state machine schematic.

Imagine that we want to translate Table 1 into VHDL code. We need to define each step and create the sequential process.

Table 1. finite State Machine

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Current State** | **Next State** | **Next State** | **Next State** | **Output** | **Output** |
|  | Input = Move  CW | Input = Move  CCW | Input = NoMove | Desired  Position | PosError |
| An0 | An45 | An315 | An0 | Current State | DesPos - PhyPos |
| An45 | An90 | An0 | An45 | .. | .. |
| An90 | An135 | An45 | An90 | .. | .. |
| An135 | An180 | An90 | An135 | .. | .. |
| An180 | An225 | An135 | An180 | .. | .. |
| An225 | An270 | An180 | An225 | .. | .. |
| An270 | An315 | An225 | An270 | .. | .. |
| An315 | An0 | An270 | An315 | .. | .. |



entity AngleFSM is -- Entity

generic (

S\_Width : integer := 3; -- State Width

An0 : std\_logic\_vector(3 downto 0) := "000";

An45 : std\_logic\_vector(3 downto 0) := "001";

An90 : std\_logic\_vector(3 downto 0) := "010";

An135 : std\_logic\_vector(3 downto 0) := “011";

An180 : std\_logic\_vector(3 downto 0) := “100";

An225 : std\_logic\_vector(3 downto 0) := “101";

An270 : std\_logic\_vector(3 downto 0) := “110";

An315 : std\_logic\_vector(3 downto 0) := “111" );

port (

clk, reset , MoveCw, MoveCCW : in std\_logic;

PhyPosition : in std\_logic\_vector(S\_Width-1 downto 0);

DesPosition, PosError : out

std\_logic\_vector(S\_Width-1 downto 0));

end entity AngleFSM;

architecture FSM\_Arch of AngleFSM is -- Architecture

signal CurrentState, NextState :

std\_logic\_vector (S\_Width-1 downto 0);

begin

comb\_proc : process (MoveCw, MoveCCW, PhyPosition,

CurrentState)

begin

case(CurrentState) is

when An0 =>

if (MoveCW = '1') then NextState <= An45;

elsif (MoveCCW = '1') then NextState <= An315;

else NextState <= An0;

when An45 =>

if (MoveCW = '1') then NextState <= An90;

elsif (MoveCCW = '1') then NextState <= An0;

else NextState <= An45;

-- ... Insert States An90 to An315 here

when An315 => -- Last State, others states

if (MoveCW = '1') then NextState <= An0;

elsif (MoveCCW = '1') then NextState <= An270;

else NextState <= An315;

when others => NextState <= An0;

end case;

end process comb\_proc;

clk\_proc : process (clk, reset) begin

if (reset = '1') then CurrentState <= PhyPosition;

elsif (rising\_edge(clk)) then CurrentState <= NextState;

end process clk\_proc;

end architecture FSM\_Arch;

-- Output Logic

-- Moore Output

DesPosition <= CurrentState;

-- Mealy Output

PosError <= DesPosition - Phy Position;

end architecture FSM\_Arch;

**Introduction to Verilog**

**Verilog**, standardized as **IEEE 1364**, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. In Verilog, a design consists of modules.

A simple AND gate in Verilog would look something like the following:

// And gate

//

module andgate (x1, x2, f);

input x1, x2;

output f;

assign f = x1 & x2;

endmodule

Verilog means Verifying Logic. Verilog - modeling language created by Gateway Automation in 1984. Then acquired by Cadence that opened the standardization. Now it is public domain -> Open Verilog International (OVI) (now known as Accellera) organization. Verilog was later submitted to IEEE and became IEEE Standard 1364-1995, commonly referred to as Verilog-95.

A Verilog code includes:

* Structural modeling (Gate-level)
  + **Use predefined or user-defined primitive gates.**
* Dataflow modeling
  + **Use assignment statements (**assign**)**
* Behavioral modeling
  + **Use procedural assignment statements (**always**)**

But how to implement a simple circuit like a 4-bit comparator into a Verilog code?

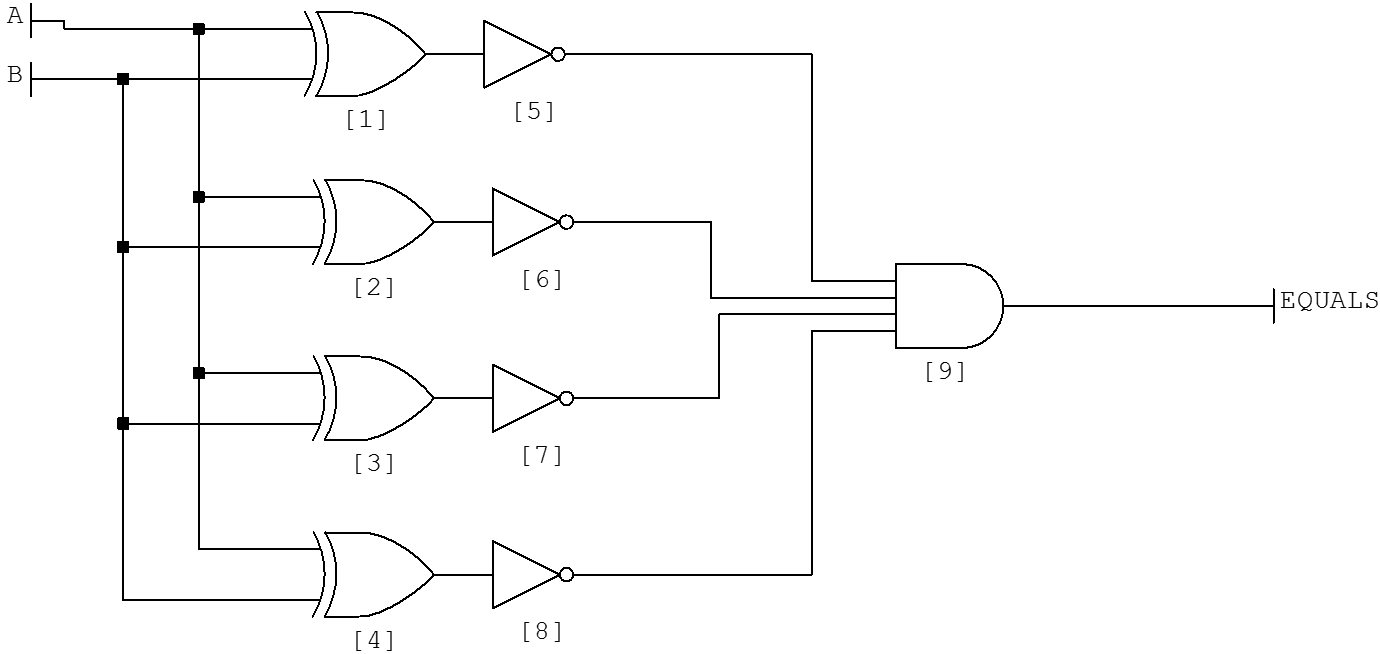


Figure 11. 4 bit comparator

module comparator(

input[3:0] a,b, output out); // Verilog-2001 Syntax

wire n0, n1, n2, n3;

xnor xnor0( n0, a[0], b[0] );

xnor xnor1( n1, a[1], b[1] );

xnor xnor2( n2, a[2], b[2] );

xnor xnor3( n3, a[3], b[3] );

and and0( out, n0, n1,n2, n3 );

endmodule

module COMPARATOR

(A, B, Y); // Verilog-1995 Syntax

input [3:0] A, B,

output Y;

assign Y = & ( (A~^B));

endmodule

module COMPARATOR

(input [3:0] A, B;

output Y);

integer N;

reg Y;

always @(A or B)

begin: COMPARE

Y = 0;

if (A == B)

Y = 1;

end

endmodule

1. Features of Verilog

Many syntax rules are much like in C software:

* Whitespace is ignored
* Comments are either //… or /\* … \*/
* Identifiers are words for variables, function names, etc. and can begin with \_ or a letter and can include letters, digits, and underscores and are CASE SENSITIVE.
* Keywords can’t be used as identifiers, they are assign, case, while, wire, reg, and, or, nand, and module just to name a few.
* The number of keywords and capabilities have expanded as Verilog and then System Verilog was developed as shown on the next slide.

1. Verilog Fundamentals

**Assignments operators**

The fundamental statement in Verilog is the assignment statement. All assignment statements outside of an always block are concurrent – they happen at the same time, and are not sequential. The output of the operation on the right hand side of the = symbol is continuously assigned to the variable on the left hand side, as in

assign A = B & C; // an and gate

The variable on the LHS must be a net, not a reg when outside of an always block.

For example, next diagram cam be translated into: Answer: assign y = (a | b) & ~c; The use of Parens are very important because they control the execution order and how they are executed.

C

a

b

y

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

always @( posedge clk)

begin

Q2=Q1; Q1=D; // shift register

Q1=D; Q2=Q1; //single or parallel ff.

end

This code would be synthesized into a single D-flip-flop. However, they are sequential and not concurrent.

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

Using nonblocking statements, the intent of the previous example to create a shift register is preserved no matter the order.

always @( posedge clk)

begin

Q2<=Q1; Q1<=D; // shift register

Q1<=D; Q2<=Q1; // shift register

Q1<=D; Q2<=D; // parallel ff

end

Generally one would not code all 3 lines, but we are showing what would happen in each case.

Guidelines:

* One generally should not mix “<=” or “=” in the same procedure.
* “<=” best mimics what physical flip-flops do; use it only for “always @ (posedge clk ..) type procedures describing sequential circuits.
* “=” best corresponds to what c/c++ code would do; use it for combinational procedures, within or outside of always blocks.

|  |  |
| --- | --- |
| **Operators, by precedence** | **Description** |
| **[ ]** | bit-select or part-select |
| **( )** | parenthesis, sets precedence order |
| **! ,~** | Logical and bit-wise NOT |
| **&, |, ~&, ~|, ^, ~^, ^~** | Reduction AND, OR, NAND, NOR, XOR, XNOR;  &(4’b0101) = 0 & 1 & 0 & 1 = 1’b0 |
| **+, -** | unary sign, plus, minus; i.e. +18, -8 |
| **{ }** | Concatenation: {3’b101, 3’b110} creates 6’b101110; |
| **{{ }}** | Replication; {3{3’b110}} creates 9’b110110110 |
| **\*, /, %** | Multiply, divide, modulus; Note: / and % may not be supported for synthesis |
| **+, -** | binary add, subtract. |
| **<<, >>** | **Shift left, shift right; X<<2 multiplies X by 4** |
| **<, <=, >, >=** | **comparison tests. Reg and wire variables are taken as positive numbers** |
| **==, !=** | **logical equality test, logical inequality test** |
| **===, !==** | **case equality, case inequality; not synthesizable** |
| **&** | **bit-wise AND; AND together all the bits in a word**  **If X=3’b101 and Y=3’b110, then X&Y = 3’b100, X^Y = 3b011** |
| **^, ~^, ^~** | **bit-wise XOR, bit-wise XNOR** |

**Practical example: how to implement a multiplexor? There are several ways in Verilog.**

sel

a

b

y

1. **Conditional Assignment**

module mux(y, a, b, sel);

output y;

input a, b, sel;

assign y = sel ? a : b;

endmodule

The implementation would look something like this:

sel

a

b

y

1. Gate primitives

module mux(y, a, b, sel);

output y;

input a, b, sel;

and g1(y1, a, nsel),

g2(y2, b, sel);

or g3(y, y1, y2);

not g4(nsel, sel);

endmodule

nsel

g4

sel

a

b

y

y1

y2

g3

g1

g2

g4

1. Combinational Logic

module mux(y, a, b, sel);

output y;

input a, b, sel;

reg y;

always @(a or b or sel)

if (sel) y = b;

else y = a;

endmodule

sel

a

b

y

1. User-Defined Primitive

primitive mux(y, a, b, sel);

output y;

input a, b, sel;

table

1?0 : 1;

0?0 : 0;

?11 : 1;

?01 : 0;

11? : 1;

00? : 0;

endtable

endprimitive

sel

a

b

y

**Verilog Modules, Port Modes and Data Types**

**Input, Output, Inout[[1]](#footnote-1) -** These keywords declare input, output and bidirectional ports of a module or task. Input and inout ports are of type wire. An output port can be configured to be of type wire, reg, wand, wor or tri. The default is wire.

module example(a, b, e, c)

input a; // An input, defaults to a wire

output b, e; // Outputs default to wire

output [1:0] c; /\* 2-bit output, must be declared \*/

reg [1:0] c; // c port declared as reg

**Buses -** Suppose we have this Verilog code[[2]](#footnote-2)[[3]](#footnote-3):

module invert(input [3:0] a, output [3:0] y);

assign y = ~a;

endmodule

What circuit would this produce? We’d get a bank of inverters, one for every bit between the 4-bit bus a and the 4-bit bus y. In a, the bits from most significant to least are a[3], a[2], a[1] and a[0] and can be accessed individually using this nomenclature. This is the little-endian order, because the least significant bit has the smallest bit number. If it had been written a[0:3], this would be big-endian order, with the MSB accessed with the smallest bit number.

**Data Types – Nets and Registers**

Nets: Represent connections between hardware elements, must be driven continuously, used to wire up instantiations. Include types wire, wor, tri and wand.

Registers: Retain the last value assigned, often used to represent storage elements. Includes types reg and integer

**Integers** are general-purpose variables.

* For synthesis used mainly as loop indexes, parameters, and constants.
* Implicitly of type **reg,** however they store data as signed numbers whereas explicitly declared **reg** types store them as unsigned.
* If they hold numbers which are not defined at compile time, their size will default to 32-bits.
* If they hold constants, the synthesizer adjusts them to the minimum width needed at compilation.

**Reg** represents storage.

* Only reg type variable can be assigned to in an always block.
* Reg does not mean register. It can be modeled as a wire or as a storage cell depending on context. When used in combinational expressions in an always block, no storage is implemented. When used in sequential statements (begin/end, if, for, case, etc.), the a latch or FF will be created.

**supply0** and **supply1** define wires tied to logic 0 (ground) and logic 1 (power), respectively.

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

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

Port Rules

When connecting modules,

**Input**s must be Nets (wire, etc.)

**Output**s can be Nets or Registers

**Inout** must be Nets

The Left Hand Side (LHS) of **procedural** **assignment**s must be of a **Register** type. For **continuous assignment**s outside of procedural blocks, LHS must be **Nets** (wires).

These rules pictured visually on the next slide.

How to implement a circuit like the following in Verilog?

sel

a[0]

b[0]

outp[0]

a[1]

b[1]

outp[1]

module bus\_mux(a, b, sel, outp);

parameter n = 2;

input [n-1:0] a;

input [n-1:0] b;

input sel;

output outp;

wire [n-1:0] sel\_bus;

assign sel\_bus = {n{sel}}; //replicates 2 times

assign outp = (~sel\_bus & a) | (sel\_bus & b);

**Verilog structure**

Modules are the basic building blocks in Verilog. A module definition always starts with the keyword module, followed by the **module name**, **port list**, **port declarations**, and **parameters**. This is followed in any order by:

* Variable definitions (local scope)
* Dataflow statements
* Instantiation of lower modules
* Behavioral Blocks
* Tasks or Functions

Followed by the keyword endmodule (no semi-colon).

**module** module\_name (port\_list);

**endmodule**

I. Port declarations

II. Variable definitions

III. Parameters

IV. Data Flow statements (assign …)

V. Module Instantiations

VI. Behavioral Blocks (begin…end)

VII. Tasks or Functions

VIII. Timing Specifications

**Module Instantiation**

Modules can be instantiated by using ordered port lists, or by port names which don’t need to follow the module order. For example, here are 2 ways to instantiate a full adder:

reg [3:0] A, B; // top level signals in caps

reg C\_IN;

wire [3:0] SUM;

wire C\_OUT;

fulladd4 faordered(SUM, C\_OUT, A, B, C\_IN);

fulladd4 fa\_name(.sum(SUM), .c\_out(C\_OUT),.a(A),.b(B), .cin(C\_IN));

Module Instantiation Gotcha

Modules can be instantiated by using ordered port lists, or by port names which don’t need to follow the module order. However, there is a possible gotcha when using the port list as seen here:

reg [3:0] A, B; // top level signals in caps

reg C\_IN;

wire [3:0] SUM;

wire C\_OUT;

fulladd4 faordered(C\_OUT, SUM, A, B, C\_IN);

fulladd4 fa\_name(.c\_out(C\_OUT), .sum(SUM), .b(B), .cin(C\_IN), .a(A));

**How to make Bigger Circuits from Smaller**

Recall the Verilog code for a 2-input mux:

module mux2(a, b, sel, y);

input [3:0] a, b;

input sel;

output [3:0] y;

assign y = sel ? b : a;

endmodule

We can cascade the 2-input mux several times to make a 4-input mux:

module mux4(a, b, c, d, sel, y);

input [3:0] a, b, c, d;

input sel[1:0];

output [3:0] y;

wire[3:0] low, high;

mux2 lowmux(a, b, sel[0], low);

mux2 highmux(c, d, sel[0], high);

mux2 outmux(low, high, sel[1], y);

endmodule

We could have also made this by nesting conditional statements.

`timescale 1 ns/ 100 ps // time resolution/precis for sim

module mult\_acc ( // module and name

input [7:0] data1, data2, // I. port declarations

input clk, aclr, // I. port declarations

output reg [15:0] mac\_out ); // I. port declarations

wire [15:0] mult\_out, add\_out; // II. variable definition

parameter mult\_size = 8; // III. parameter

assign add\_out = mult\_out + mac\_out; // IV.dataflow

multiplr #(.width\_in(mult\_size)) // V. instantiation

u1 (.in\_a(data1), .in\_b(data2), .mult\_out(mult\_out));

always @ (posedge clk, posedge aclr) begin

if (aclr) // VI. behavioral block

mac\_out <= 16'h0000;

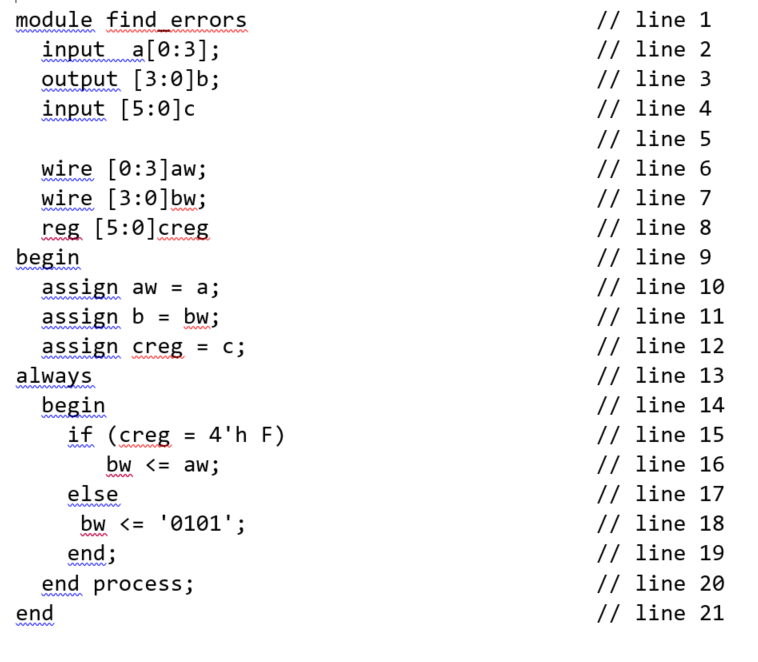
else

mac\_out <= adder\_out;

end // end behavioral block

endmodule

**Typs and considerations**



Line 1- No start paren

Line 2 - dimensions in wrong place, semi-colon instead of comma

Line 8 – missing semi-colon

Line 13 – missing sensitivity list

Line 15 - incorrect logic test (= instead of ==), array size mismatch

Line 16 - cannot assign to a wire inside of always

Line 20 - no end needed, extra

LinLine 21 - needs to be endmodule

1. Combinational Circuits in Verilog

Combinational circuits don’t need to be synchronized and they work with basic combinational elements. The following logic gates can be translated in Verilog code as it is shown.

W

U

A

B

A

A

C

B

B

C

D

D

C

D

X

Y

Z

module gates ( // module and name

input A, B, C, D,

input [3:0] vA, vB,

output W, U, X, Y, Z,

output [3:0] vX, vY);

assign W = A & B; // scalar AND Gate

assign U = ~(A | B); // scalar NOR Gate

assign X = C ^ D; //scalar XOR Gate

assign Y = C ~^ D; //scalar XNOR Gate

assign Z = (A & B) | (C & D); // AND-OR gates

assign vX = vA & vB; // Vector bitwise AND

assign vY = vA | vB; // Vector bitwise OR

endmodule

module gates ( // module and name

input [3:0] vA, vB, vC, vD,

output W, U, X, Y, Z);

assign W = & vA; // Vector Reduction AND Gate

assign U = ~| vB; // Vector reduction NOR Gate

assign X = ^ vD; // Vector reduction XOR Gate

assign Y = | vA & vB ; // bitwise or reduction?

assign Z = | (vA & vB); // bitwise AND,

// reduction OR

endmodule

It is crucial to control the order of the evaluation with parenthesis and remove ambiguity on the operations.

**Procedural Combinational Logic with Always Procedures.** What circuit will this produce since Y is a rag does this mean that we will have a register reg is different from a wire the signals of type **reg hold their values between simulation Delta'**s this is necessary as the always construct is only executed and simulation when there's a change on the sensitivity list in this case A B C or D.

module always\_combo (

input A, B, C, D,

output reg Y);

always @(A or B or C or D)

begin

if ((C==1) && (D==1))

Y <= 0;

else if ((A==1) || (B==1))

Y <= 1;

else

Y <= 0;

end

endmodule

In always blocks, if you don’t completely assign all the signals for all ases then latches may be inferred. This happens because if the output isn't defined then a memory element needs to retain it until the next signal change. EVERY control path of the sensitivity list must be assigned to avoid extra latches. Don't forget the final else in the if then else statements or the default state in case statements to prevent inferred latches. If you do this and write complete code, you'll get combinational circuits you intended.

Practical example. Imagine that we create the following code, where the last else statement is missing, then, the value of D from the sensitivity list is missing and the synthesizer will consider this structure as a latch. For that reason, never forget to assign all signals in an always block.

module always\_combo (

‘1’

A

Latch

D

C Q

R

input A, B, C, D,

B

output reg Y);

Y

always @(A or B or C or D)

C

begin

if ((C==1) && (D==1))

D

Y <= 0;

else if ((A==1) || (B==1))

Y <= 1;

end

endmodule

1. Synchronous Circuits in Verilog

This is an example of the fundamental synchronous circuit, the D flip flop. Synchronous circuits are essential to the design of advanced digital circuits. The use of a clock, and an edge triggered circuit allows us to avoid logical hazards inherent and purely combinational circuits. The always block is necessary as is the *posedge* attribute of the clock. Q must be a reg. The flip flop is described in one line as the output gets the input on the rising clock edge. Even more fundamental than a D flip-flop is the D latch. Knowing what you know, how would you describe a D latch in Verilog? As we learned previously, sometimes we get latches by mistake, but how do we create one when that's what we really want?

// D flip flop

//

module DFF (

input D, Ck,

output reg Q

);

always @(posedge Ck)

Q <= D;

endmodule

Here's the code for a D latch with and without an asynchronous clear. Note the sensitivity list includes data input as well as the clock, and that the clock is not edge triggered, so this produces a level sensitive circuit. The use of a keyword does not imply the logical or function, it's just the separator for the list. Why then don't just use commas? I can't explain. Try not to be confused by the syntax. One way to understand this is that in simulation, the always block code is evaluated when there's an event on clk, or an event on d, or an event on aclr. So the first always block gives us a synchronous type of D latch, the second always blocks gives an asynchronous D latch.

// D latches

//

module DLatches ( d, clk, aclr, qldc, qld);

input d, clk, aclr;

output reg qldc, qld;

always @(clk or d) // no posedge!

begin

if (clk == 1) qld <= d;

end

always @(clk or d or aclr)

// note d is on the sensitivity list

begin

if (aclr == 1) qldc <= 0;

else if (clk == 1) qldc <= d;

end

endmodule

How can we make D Flip-flops in Verilog, including those with asynchronous or synchronous reset?

D

Q

R

S

// D Flip Flops

//

module DFF (d, clk, clr, reset, qld, qlda, qlds);

input d, clk, clr, reset;

output reg qld, qlda, qlds;

always @(posedge clk) // with posedge!

begin

qld <= d; //standard FF

if (reset == 0) qlds <= 0;

else qlds <= d; // FF with sync reset

end

always @(posedge clk or negedge clr)

begin

if (clr == 0) qlda <= 0;

else qlda <= d; // FF with async reset

end

endmodule

D

Q

R

S

En

How can we make Clock Enable in Verilog?

// D Flip Flops with clock enable

//

module DFFe (d, clk, ce, clr, reset, qlda, qlds);

input d, clk, ce, clr, reset;

output reg qld, qlda, qlds;

always @(posedge clk) //

begin // FF with sync reset & clk enable

if (reset == 1) qlds <= 0;

else if (ce == 1) qlds <= d;

end

always @(posedge clk or posedge clr)

begin // FF with async reset & clk enable

if (clr == 1) qlda <= 0;

else if (ce == 1) qlda <= d;

end

endmodule

1. Synchronous logic: registers

Data is oftentimes stored in a register for later use. How can we make a data register in Verilog?

Q0

Q1

Q2

Q3

D0

D1

D2

D3

Reset

Load

Clk

D

Q

En

R

D

Q

En

R

D

Q

En

R

D

Q

En

R

// Data Register

//

module Dreg (

input wire [3:0] d,

input wire clk, reset, load,

output reg [3:0] q

);

always @(posedge clk or negedge reset)

begin

if (!reset) // asynchronous reset

q <= 0;

else if (load == 1)

q <= d;

end

endmodule

How can we make a shift register in Verilog?

D

Q

En

R

D

Q

En

R

D

Q

En

R

D

Q

En

R

Clk

Shift

Reset

Q3

// Shift Register

//

module Shifter (clk, reset, D0, shift, Q);

input clk, reset, D0, shift;

output reg [3:0] Q;

always @(posedge clk) //

begin

if (!reset)

Q <= 4’b0000;

else if (shift == 1)

begin

Q <= Q << 1;

Q[0] <= D0;

end

else ;

end

endmodule

How can we make a binary counter in Verilog?

A binary counter is a simple counter that has an initial value of 0 at the time of reset. It value keeps incrementing by 1 at each clock cycle. And finally it reaches its maximum value, say 1111 in binary for a 4 bit counter, it again reaches 0000 in binary and keeps counting one.

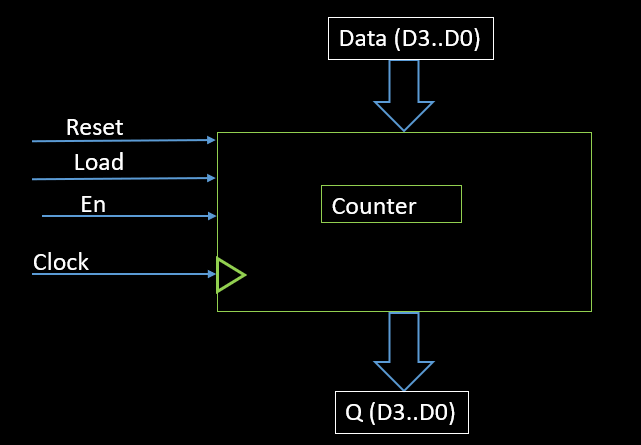


Figure 12. binary counter.

// Binary Counter

//

module Counter (

input wire [3:0] d,

input wire clk, reset, load, en,

output reg [3:0] q

);

always @(posedge clk)

begin

if (reset) // synchronous reset

q <= 0;

else if (load == 1’b1)

q <= d;

else if (en == 1’b1)

q <= q + 1;

end

endmodule

In the case of a register file where diiffernt register are located and we want to access them in order, the implementation would be seen like this.

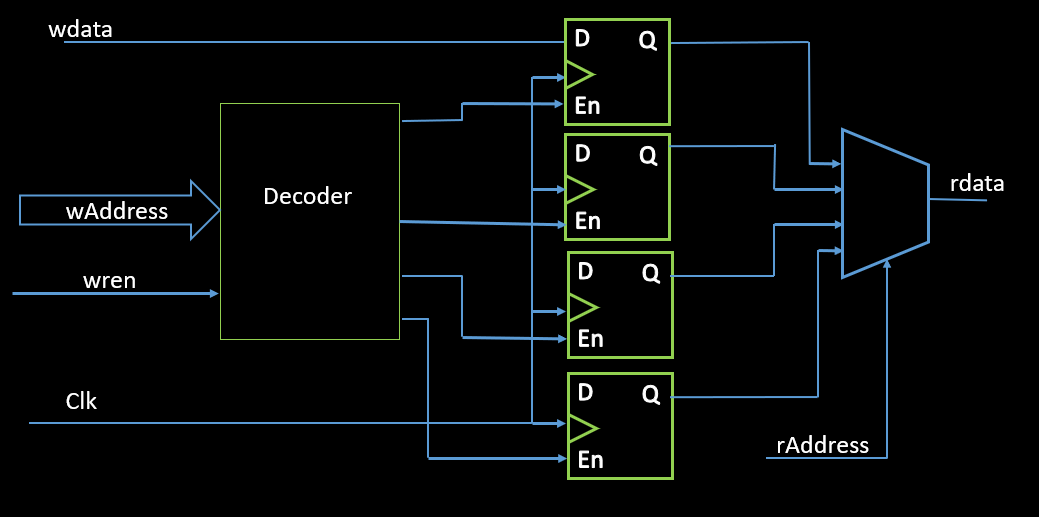


Figure 13. Register file.

// Data Register File

// 4 x 8

module regFile #(

parameter Dwidth = 8, // #bits in word

Awidth = 2 // #address bits

)

(

input wire clk, wren,

input wire [(Dwidth1):0] wdata,

input wire [(Awidth1):0] waddr, raddr,

output wire [(Dwidth-1):0] rdata

);

// Signal Declaration

reg [Dwidth-1:0] array\_reg [2\*\*Awidth-1:0];

always @(posedge clk)

if (wren) // synchronous enable

array\_reg[waddr] <= wdata;

assign rdata = array\_reg[raddr];

endmodule

1. Tri-state and bidirectional buses in Verilog.

External connections on FPGA pins are often in a group of related signals known as a bus.

The I/O structures of FGPAs often allow the bus to be tri-stated, so that multiple drivers can be attached to the bus at the same time, with only one active.

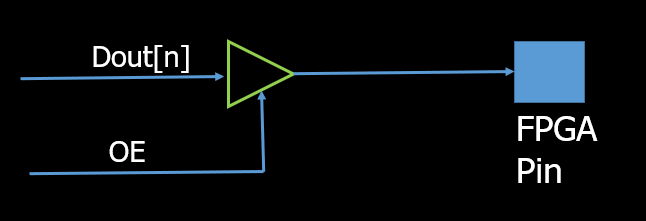


Figure . Tri-state bus.

// Tri-state bus

//

module Tri (

input wire [3:0] Dout,

input wire OE,

output wire [3:0] Pinout

);

assign Pinout = OE ? Dout : 4'bz;

// or **assign** Pinout = (OE == 1) ? Dout :

(OE == 0) ? 4'bz : 4'bx;

endmodule

The I/O structure of the FPGA will also allow us to create Bi-directional buses, in which the external pin can be treated as either an input or an output, depending on the state of the enable signal.

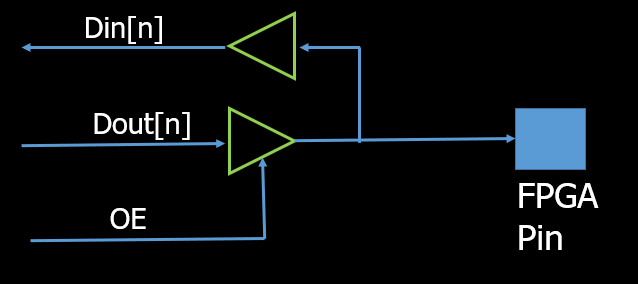


Figure . Bi-directional bus.

// Bi-directional bus

//

module BiDir (

output wire [3:0] Din,

input wire [3:0] Dout,

input wire OE,

inout wire [3:0] IOpin

);

assign Din = IOpin;

assign IOpin = (OE == 1) ? Dout :

(OE == 0) ? 4'bz : 4'bx;

endmodule

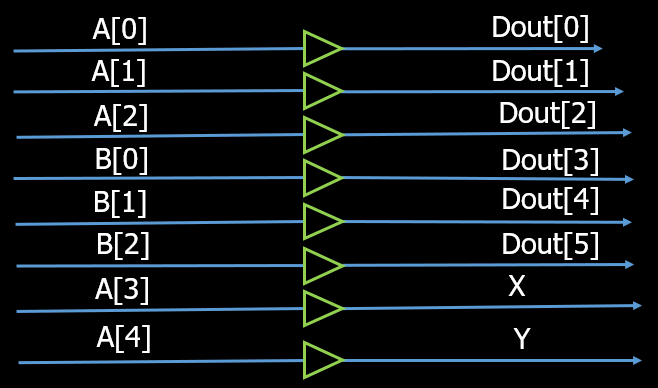


Figure . Joining and splitting buses

// Manipulating Buses

//

module BusMe (

input wire [4:0] A,

input wire [2:0] B,

output wire X,Y,

output wire [5:0] Dout

);

assign Dout = {B,A[2:0]};

assign X = A[3];

assign Y = A[4];

endmodule

1. Modular design in Verilog

Given the principal design entity in Verilog is a module, one could infer that Verilog designs should be modular. Many tools are provided in Verilog to make this happen, including

* Component Instantiation
* Looping
* Generate Blocks
* Tasks and Functions

Module module\_name\_top (port connection list)

instance\_name\_1 (port connection list),

instance\_name\_2 (port connection list),

.......  
instance\_name\_n (port connection list);

endmodule

More often than not, at the top level, modular Verilog design consists of component instantiations only, like the code that we see here. Use of component or module instantiations is the fundamental way to build hierarchy in a design. Consider this example which creates a 16-bit adder using nothing but instantiations of 4-bit adders.

// 16-bit Adder

// built with 4 instantiations of

// 4-bit Adders

module Add16\_top (

input [15:0] A,

input [15:0] B,

input Cin,

output Cout,

output [15:0] Sum

);

wire Cin2, Cin3, Cin4; //intermediate Cin

add4 add4\_1 (.Data1(A[3:0]), .Data2(B[3:0]), .Cin(Cin), .Cout(Cin2), .Sum(Sum[3:0]));

add4 add4\_2 (.Data1(A[7:4]), .Data2(B[7:4]), .Cin(Cin2), .Cout(Cin3), .Sum(Sum[7:4]));

add4 add4\_3 (.Data1(A[11:8]), .Data2(B[11:8]), .Cin(Cin3), .Cout(Cin4), .Sum(Sum[11:8]));

add4 add4\_4 (.Data1(A[15:12]), .Data2(B[15:12]), .Cin(Cin4), .Cout(Cout), .Sum(Sum[15:12]));

endmodule

1. Looping in Verilog

There are several looping constructs within Verilog, including

* repeat
* while
* forever
* for

The for loop statement is written and behaves just like it does in C, as is the while loop. The forever loop is explicitly an infinite loop. The repeat statement executes a statement or block of statements a fixed number of times.

// And scalar g with vector A

//

Module scalarAnd

#(parameter N = 4);

(input g,

input [N-1:0] a,

output [N-1:0] y);

reg [N-1:0] tmp, y;

integer i; //loop index, not signal

always @(a or g)

begin

for(i=0; i<N; i=i+1)

begin

tmp[i] = a[i] & g;

end

y = tmp;

end

endmodule

1. ***Generate* in Verilog**

Although loops can be used to generate data or test patterns, in Verilog a common use of loops for synthesis is replication of many identical circuits within generate blocks.

The generate … end generate block specifies how an object is to be repeated. Variables used to specify the repetition are called genvars. The index variable of a for loop in a generate block must be a genvar.

// Generate n XOR gates

//

module XorGen

#(parameter width = 4,

delay = 10)

(output [1:width] xout,

input [1:width] xin1, xin2

);

generate

genvar i;

for (i=1; i<=width; i=i+1)

begin

assign #delay

xout[i] = xin1[i] ^ xin2[i];

end

endgenerate

endmodule

1. **Testbenches in Verilog**

What is a testbench? A testbench is a program written in any language for the purposes of exercising and verifying the functional correctness of the hardware model as coded. In our case, Verilog is going to be used for both the model and the test code. The use of the testbench to auto-generate a stimulus for the unit under test is much more efficient than entering test vectors manually in the simulator, Modelsim in this case. The testbench is also sometimes called a test fixture or a test harness. Its a very powerful tool for auto generating test stimulus and test results, particularly when combined with a simulator like Modelsim in which the Verilog test code will run.

A testbench can have several functional sections, including:

* Top-level testbench declaration
* Stimulus and Response Signal declarations
* Component declarations
* Component (Device Under Test) instantiations
* External Stimulation Device Models
* Test Process which applies the stimulus to the DUT
* Test Monitor which reports results

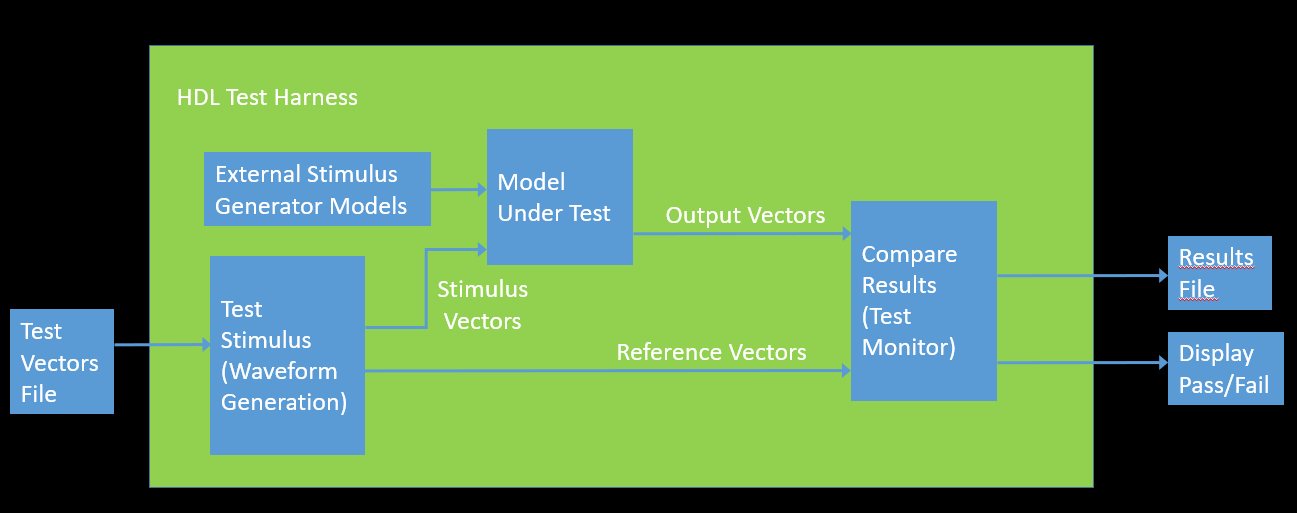


Figure . Diagram of a Testbench structure.

`timescale 1 ns / 1 ps // set timescale to

//nanoseconds, ps precision

module Adder\_tb(); // no sensitivity list!

// signal declarations

reg [3:0] a\_tb, b\_tb; // data input stimulus

reg Cin; // data input stimulus

wire [3:0] y\_tb; // data output response

wire Co\_tb; // data output response

reg [3:0] expected; // expected sum result

// DUT instantiation

add4 DUT(.A(a\_tb), .B(b\_tb), .Cin(Cin), .Sum(y\_tb), .Cout(Co\_tb));

// Adder Testbench, continued

//Test stimulus generation

initial

begin

#0 a\_tb=2; b\_tb=2; Cin=0; expected=4;

#10 a\_tb=15; b\_tb=0; Cin=1; expected=0;

#10 a\_tb=2; b\_tb=4; Cin=1; expected=7;

#10 $stop;

end

// Test Results

initial

$monitor("time=%d, a=%b, b=%b, Cin=%b, sum=%b, cout=%b, expected sum=%b",

$time, a\_tb, b\_tb, Cin, y\_tb, Co\_tb, expected);

endmodule

A more efficient way of implementing a testbench would be by adding loops for the control signals.

integer i, j, k;

//Adder Testbench, continued

//Test stimulus generation

initial

begin // loop over number of a inputs

// possible

for(i = 0; i<16; i = i+1) begin

a\_tb <= i;

for(j=0; j<16; j = j+1) begin

b\_tb <= j;

for (k=0; k<2; k=k+1) begin

Cin <= k;

#(10);

expected <= a\_tb + b\_tb + Cin;

end

end

end

end

We can use assertion statements to control errors and situations. A helpful feature available in VHDL and system Verilog is the assertion statement. Here we use the assertion to assert a true condition, the tests, and if it passes, then we can print a message, or for most of the time, just omit the past message and continue, or if the test fails, we display an error message and record the result. The dollar sign finish as a system function, which causes the simulation to end. We like to use assertions because they can be enabled or disabled globally in simulations unlike our previous tests. They can also have different severity levels, including fatal, in which case the simulation will stop, error, or warning, and info, which allows us to respond to the test results in different ways. The assertions are ignored by synthesis.

Next step is to learn how to write simple testbenches for synchronous circuits and how to use external signal generators to create stimulus for those synchronous circuits. An additional complication comes from creating a testbench for a synchronous circuit like this counter some provision needs to be made for the clock signal we can do this by creating a clock generator model and Verilog and using it to drive the counter under test again. We will need some kind of comparison to verify that the counter is counting as it should.

Let’s define a testbench for testing a counter design.

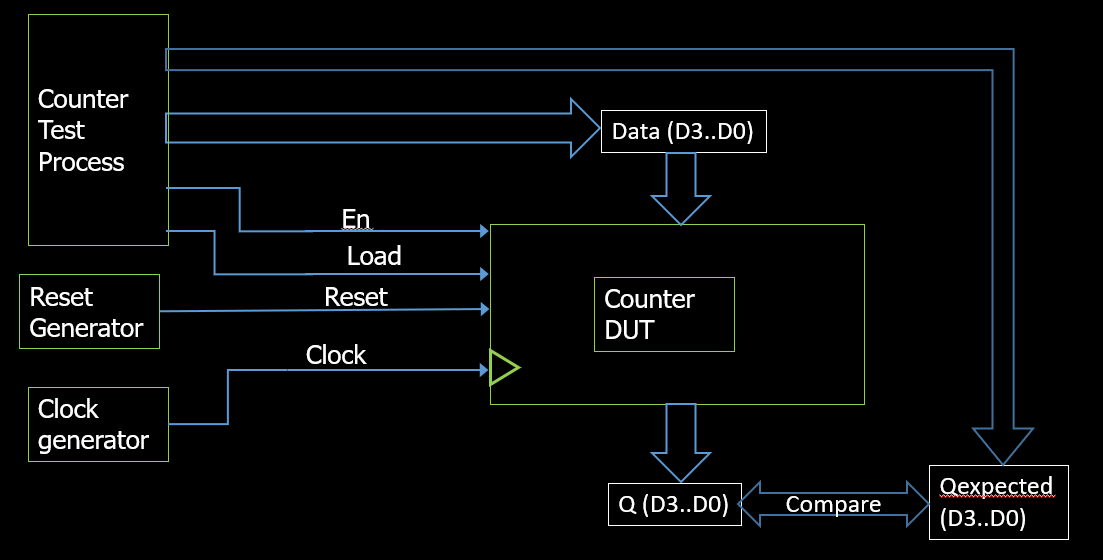


Figure . Counter diagram for the testbench evaluation[[4]](#footnote-4).

timescale 1 ns / 1 ps // set timescale

// to nanoseconds, ps precision //Testbench entity declaration

module Counter\_tb (); // top level, no external ports //constant declarations

parameter delay = 10; //ns defines the wait period.

localparam n = 4; // width of counter in bits

localparam T = 20; // clock period

// signal declarations

reg clock = 0; //clock if needed, from generator model

reg reset = 0; // reset if needed

reg [n-1:0] data\_tb = 4'b0000; // data input stimulus

reg load = 0, en = 0; // input stimulus

wire [3:0] q; // output to check

reg [n-1:0] checkcount= 4'b0000;// variable to compare

// to count

// Component Instances

// instantiate the device under test

Counter DUT ( // Device under Test

// Inputs

.d(data\_tb),

.clk(clock),

.reset(reset),

.load(load),

.en(en),

// Outputs

.q(q)

);

// External Device Simulation Processes

// clock driver

always

begin

clock = 1'b1;

#(T/2);

clock = 1'b0;

#(T/2);

end

// reset driver

initial

begin

reset = 1'b1;

#(T/2);

reset = 1'b0;

end

// Test Process

initial // test generation process

begin

@(negedge reset) // wait for reset inactive

@(negedge clock) // wait for one clock

// Test load

load = 1'b1;

en = 1'b0;

data\_tb = 4'b1010;

@(negedge clock) // wait for one clock

if (q != 4'b1010)

$display("Load failure %b", q);

// Test Process, continued

// test count

checkcount = 4'b1010; // compare variable

load = 1'b0;

en = 1'b1;

repeat (2\*\*n)

begin

checkcount = checkcount + 1; // count

@(negedge clock) // wait for one clock

if (q != checkcount)

$display("Count failure at time %g/t at

count %b", $time, q);

end

$stop; // end simulation

end

endmodule

1. **Memory in Verilog**

Memories are a common element in most digital systems. Earlier in this course we described a register file circuit, in which individual registers were enabled for access by decoding an address. We will now extend this to include RAM and ROM memories[[5]](#footnote-5).

Register Files are useful constructs that allow addressing of registers. Here we assume each flip flop represents an n-bit register.

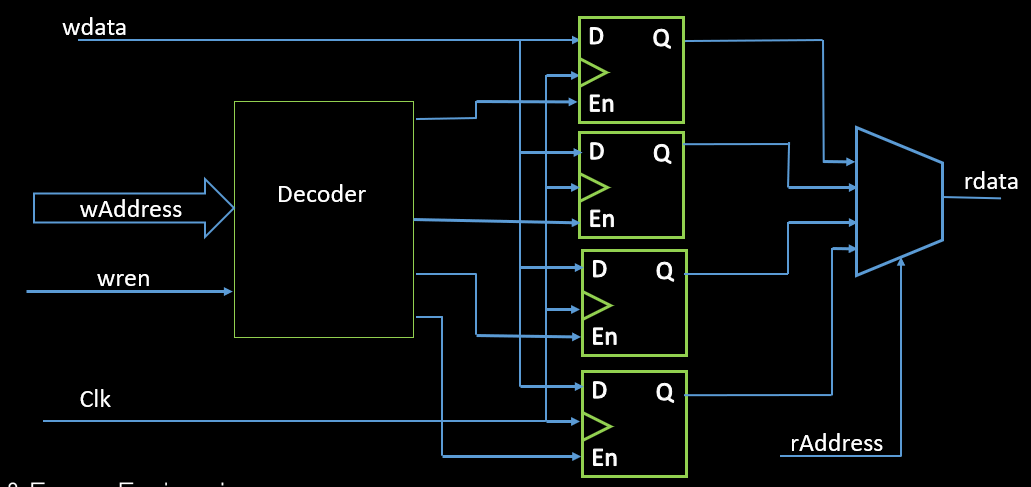


Figure . Memory diagram of a Register File.

module DPRAM

#(

parameter Data\_width = 8, //# of bits in word

Addr\_width = 10 // # of address bits)

( //ports

input wire clk,

input wire we,

input wire [(Addr\_width-1):0] w\_addr, r\_addr,

input wire [(Data\_width-1):0] d,

output wire [(Data\_width-1):0] q

);

// signal declarations

reg [Data\_width-1:0] ram [2\*\*Addr\_width-1:0];

// 2 dimensional array for RAM storage

reg [Data\_width-1:0] data\_reg;

// read output reg.

// RAM initialization from an external file

initial

$readmemh(“initalRAM.txt”, ram);

// body

// write operation

always @(posedge clk)

begin

if (we)

ram[w\_addr] <= d; // write data

data\_reg <= ram[r\_addr]; // read data to reg

end

// read operation

assign q = data\_reg;

endmodule

**Implementation of a Read Only Memory (ROM)**

module ROM

#(parameter Data\_width = 8, //# of bits in word

Addr\_width = 3 // # of address bits)

( //ports

input wire clk,

input wire [Addr\_width-1:0] addr,

output wire [Data\_width-1:0] data

);

// signal declarations

reg [Data\_width-1:0] rom\_data, data\_reg;

// body

always @(posedge clk) // output register

data\_reg <= rom\_data;

always @\*

case(addr) // lookup table

3’b000: rom\_data = 8’b1000\_0000;

3’b001: rom\_data = 8’b1010\_1010;

3’b010: rom\_data = 8’b0101\_0101;

3’b011: rom\_data = 8’b1000\_0011;

3’b100: rom\_data = 8’b0000\_0000;

3’b101: rom\_data = 8’b1001\_1001;

3’b110: rom\_data = 8’b1000\_0001;

3’b111: rom\_data = 8’b1111\_0000;

endcase

assign data = data\_reg;

endmodule

1. **Finite State machine in Verilog**

Finite State machines are a very important part of digital design and software design, too. The state machine concept provides a highly reliable, maintainable and methodical way to design circuits that perform a sequence of operations with great predictability. State machines are always in a known state. It is a very good practice to work with finite state machines as they are very well defined and error proof.

FSMs are broadly categorized into 2 types: Moore or Mealy.

* In Moore machines the output only depends on the state.
* In Mealy machines the inputs and the state drive the output.

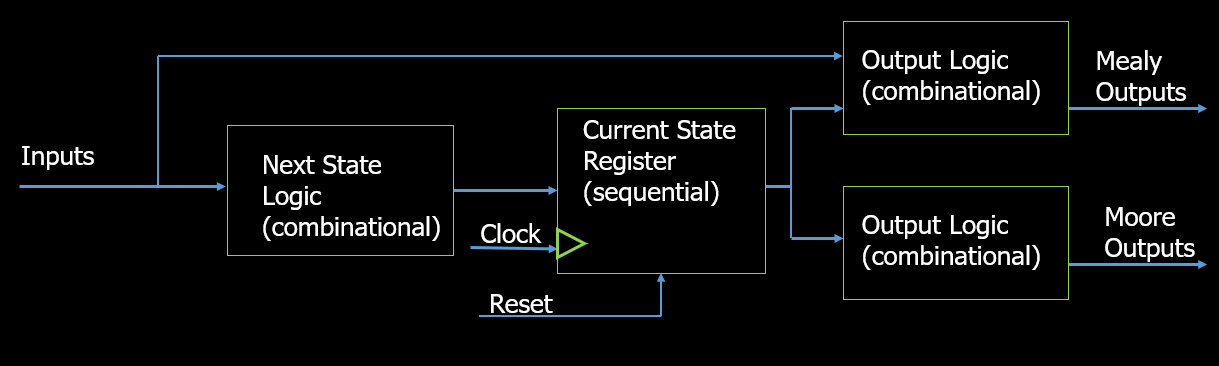


Figure . Finite state machine diagram.

Practical implementation of a Finite state machine:

MoveCW

NoMove

MoveCCW

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Current State | Next State | Next State | Next State | Output | Output |
|  | Input = MoveCW | Input = MoveCCW | Input = NoMove | DesiredPosition | PosError |
| An0 | An45 | An315 | An0 | Current State | Dpos - PhyPos |
| An45 | An90 | An0 | An45 | .. | .. |
| An90 | An135 | An45 | An90 | .. | .. |
| An135 | An180 | An90 | An135 | .. | .. |
| An180 | An225 | An135 | An180 | .. | .. |
| An225 | An270 | An180 | An225 | .. | .. |
| An270 | An315 | An225 | An270 | .. | .. |
| An315 | An0 | An270 | An315 | .. | .. |

1st – State encoding

We need to encode all the states on a number and give an input and output that will be the next state or the output of the system.

TABLE X. Encoding of the finite state machine.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Binary | Gray | Johnson | One-Hot |
| 0 | 000 | 000 | 0000 | 00000001 |
| 1 | 001 | 001 | 0001 | 00000010 |
| 2 | 010 | 011 | 0011 | 00000100 |
| 3 | 011 | 010 | 0111 | 00001000 |
| 4 | 100 | 110 | 1111 | 00010000 |
| 5 | 101 | 111 | 1110 | 00100000 |
| 6 | 110 | 101 | 1100 | 01000000 |
| 7 | 111 | 100 | 1000 | 10000000 |

**Why do we use different encoding?**

Binary gives the simplest implementation in terms of logic resources and latches. However, if we have a single bit error on the system, that would be easily translated into a problem on the behavior of the whole implementation. Gray and Johnson implementations are the most robust implementations because they avoid errors more easily. However, One-hot implementation is the easier strategy as we save the state of the machines in different flip-flops, that means that each state has a different flip-flop instead of combining flip-flops for keeping the states.

The code for finite state machines are often very big but, they are robust, error proof and very precise. The implementation of the this finite state machine’s code on binary encoding is shown in figure .

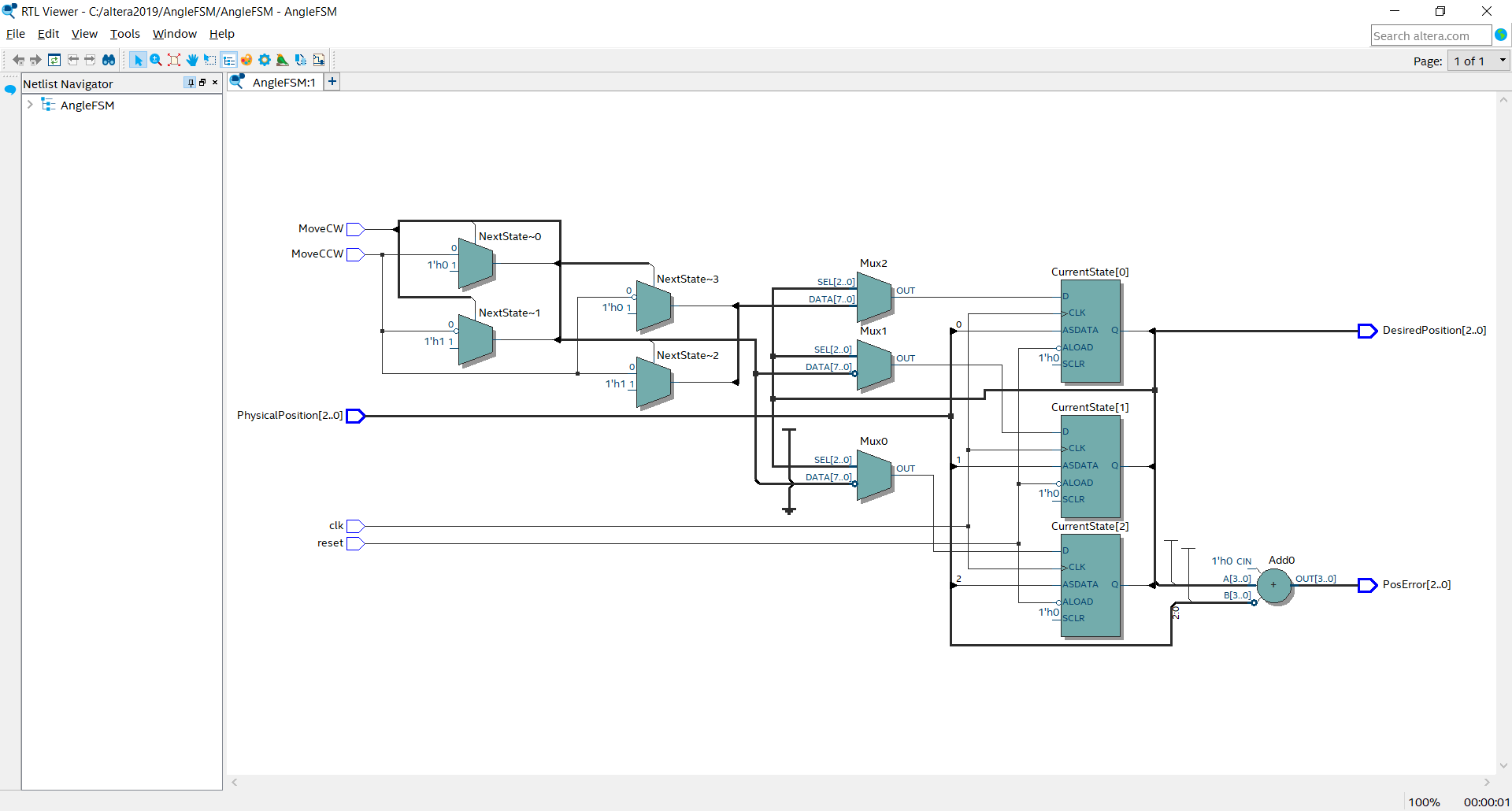


Figure . Binary encoding implementation of the finite state machine.

module AngleFSM

#( // Binary encoding of states

parameter State\_width = 3,

An0 = 3'b000,

An45 = 3'b001,

An90 = 3'b010,

An135 = 3'b011,

An180 = 3'b100,

An225 = 3'b101,

An270 = 3'b110,

An315 = 3'b111)

( //ports

input wire clk, reset, MoveCW, MoveCCW,

input wire [(State\_width-1):0] PhysicalPosition,

output wire [(State\_width-1):0] DesiredPosition, PosError

);

reg [(State\_width-1):0] CurrentState, NextState;

// body of FSM is case statement

// Next State Logic

always @(MoveCW or MoveCCW or PhysicalPosition or CurrentState)

begin: Combinational

case (CurrentState)

An0:

if (MoveCW ==1)

NextState = An45;

else if (MoveCCW == 1)

NextState = An315;

else

NextState = An0;

An45:

if (MoveCW ==1)

NextState = An90;

else if (MoveCCW == 1)

NextState = An0;

else

NextState = An45;

... //states An90 to An270 here

An315:

if (MoveCW ==1)

NextState = An0;

else if (MoveCCW == 1)

NextState = An270;

else

NextState = An315;

default:

NextState = PhysicalPosition;

endcase

end

// Current State Register

always @(posedge clk or negedge reset)

begin: Sequential

if (!reset)

CurrentState = PhysicalPosition;

else

CurrentState = NextState;

end

// Output Logic

// Moore Outputs

assign DesiredPosition = CurrentState;

// Mealy Outputs

assign PosError = DesiredPosition - PhysicalPosition;

endmodule

1. Stephen A. Edwards. 2009 March. *Verilog 1995, 2001, and System Verilog3.1*. [Online]. Available: www.cs.columbia.edu/~sedwards/classes/2004/e [↑](#footnote-ref-1)
2. Sarah L. Harris and David Money Harris, “Hardware Description Languages,” in *Digital Design and Computer Architecture*. [↑](#footnote-ref-2)
3. Waltham, MA, Morgan Kaufman, 2016, pp. 178 [↑](#footnote-ref-3)
4. Pong P. Chu, “Regular Sequential Circuit” in *Embedded SOPC Design with NIOS II Processor and Verilog Examples*, Hoboken, NJ, Wiley, 2012, ch. 5, sec. 5.2, pp. 107-110 [↑](#footnote-ref-4)
5. D. Smith, “Test Harnesses” in *HDL Chip Design, A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog,* Madison, AL, Doone Publications, 1996, ch. 11, pp. 323-344. [↑](#footnote-ref-5)