A Design and Verification Tutorial

Mahesh Sukhdeo Palve

website : http://digitalbyte.weebly.com
facebook page : https://www.facebook.com/thedigitalbyte

This document created using LaTeX 2_ε

Contents

Chapter 1

Introduction to HDL Design

Digital circuit design is much simple than the analog design. There are some methods, techniques applicable to digital circuit bearing any level of complexity. With the help of '**E**lectronic **D**esign **A**utomation' (EDA), the circuit to be designed can be described at any level of abstraction. This chapter takes a quick review of VHDL and Verilog.

1.1 Languages for design

Languages are used for describing the hardware behavior at higher level of abstraction. There are languages to describe the design, and to verify the functionality of design.

1.1.1 Hardware Description Language (HDL)

Traditional programming languages are not of much use when we want to specify some 'hardware' behavior, and that is why 'Hardware Description Languages' (HDLs) came into existence. As the name implies, HDL are specially used to Describe the Hardware we want to design.

In the world today, two HDLs are widely accepted for designing digital hardware. These are VHDL and Verilog-HDL.

VHDL is acronym for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language.

Verilog-HDL or simply Verilog is much more used HDL due to its simplicity and similarity to C-programming language. However, VHDL is also equally important, and perhaps more powerful HDL.

1.1.2 Hardware Verification Language (HVL)

Like there are languages to describe the hardware, there are some to verify the functionality of those designs. These languages have constructs useful for hardware verification. SystemVerilog is one of the HVL which is getting wide acceptance now-a-days. This language is a superset of Verilog-HDL. It is object-oriented which makes verification of complex designs much easy.

1.2 Hardware Synthesis from HDL

"Synthesis" is the process of 'inferring' certain hardware from the written HDL code. A synthesis tool generates a 'netlist' specifying the hardware resources and their interconnections. HDLs are usually targeted at some specific device, FPGA or CPLD. The synthesis tool knows what kind of resources are available on target device and generates netlist accordingly. Some directives might be available to tell the tool to infer a specific resource of device.

This tutorial aims at HDL coding of a FIFO and its verification in various, but not all, possible ways.

1.3 HDL File structures

VHDL files bear the extension ".hdl". Verilog files are identified by ".v" extension. We shall discuss the file structure of HDL (RTL) file and Testbench files for both HDLs.

1.3.1 VHDL RTL file

HDL files are often termed as RTL (Register Transfer Level) files. A VHDL RTL file is composed of four main parts viz.

- Library
- Packages
- Entity
- Architecture

Note that VHDL is *NOT* case sensitive.

Library and Packages

Library is a collection of pre-compiled functions that we can readily use. It saves us from writing everything from scratch. The most popular VHDL library is

```
library ieee;
```

This appears in almost every VHDL code. Having declared the library, we need to specify which packages we want to use from that library. The most widely used package from library ieee is std_logic_1164. To use everything from this package, we write

```
use ieee.std_logic_1164.all
```

It contains a nine valued logic, and all the basic gates such as AND, OR, NAND, Inverter etc.

Another package is std_logic_unsigned which contains arithmetic operation (add, subtract, comparison etc.).

Entity

The entity resembles the pinouts of a circuit. Inputs to the design and outputs from it (and also the bidirectional in-out ports) are specified in entity. e.g. the enity of a D Flip-Flop is written as

```
entity DFF is
   port ( clk : in std_logic;
   D : in std_logic;
   Q, Qbar : out std_logic);
end DFF;
```

Here, '*entity*', '*is*', '*port*', '*end*', '*in*', '*out*' are keyword of VHDL. Although not a rule, they are written in small case.

Now consider the *clk* port-

```
clk : in std_logic;
```

In this, 'clk' is name of the port. Its direction is 'in' i.e. clk is an input port. 'std_logic' means clk can have any of the nine values specified in std_logic_1164 package. The nine values are

'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

Thus we assign any of above mentioned nine values to the port specified as type 'std_logic'.
Another commonly used type is 'std_logic_vector'.
e.g.

```
par_in : in std_logic_vector (7 downto 0);
```

Here, par_in is a 8-bit input bus.
And, we could define another bus as

```
par_in2 : in std_logic_vector (15 downto 8);
```

Architecture

This part of HDL file elaborates the relationship between the ports defines in entity i.e. the actual design. In VHDL, the architecture can be written using various modeling styles :
Data-flow modeling in which standard library functions are used as it is.
Behavioral modeling in which truth table of design is described with the help of VHDL constructs.
Structural modeling in which design is decomposed into several blocks termed as 'components' and each component is described using either of above two styles. This is useful for very complex designs such as a processor.
Mixed modeling in which any two or all three of above are used. This is more commonly used style rather than using one specific of above.

1.3.2 VHDL Testbench file

Testbench is used for functional simulation of the design. The input ports of design are driven by the testbench and the outputs are collected. We can compare the outputs with expected values or simply display them in the form of waveforms.

The file structure of VHDL testbench is same as that of VHDL RTL file. However, the entity of Testbench does not have any ports. The RTL design (often termed as Design Under Test (DUT) or Unit Under Test (UUT)) is defined as a component in testbench architecture and is driven and read in architecture body.

1.3.3 Verilog RTL file

Unlike VHDL file which has four parts, the Verilog file has just one thing in it i.e. module. This makes Verilog codes much smaller than VHDL codes.

e.g. **Verilog** code for AND gate :

```
module andGate (a, b, c);

       a, b : input;
         c : output;

       assign c = a & b;

endmodule
```

and same with **VHDL** :

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

entity andGate is
     port (a,b : in std_logic;
            c : out std_logic
         );
 end andGate;

  architecture dataFlow of andGate is
  begin

  c <= a and b;

  end dataFlow;
```

So, that is how simple Verilog is. Moreover, verilog IS case sensitive, but is not highly-typed as VHDL. It means that when we assign a and b to c, it is OK if a and b are buses (vectors) and c is not, or vice versa. This is not permitted in VHDL since it is highly-typed language, posing a compulsion that data-types must match for assignment.

1.3.4 Verilog Testbench file

This is much similar to Verilog RTL file. It contains a module, without port list. An instance of DUT (or UUT) and then inputs are driven while outputs are read in body of module.

The language constructs, data types etc. of either HDL is out of scope of this tutorial. Readers can to refer the book "HDL Chip Design" (Doone Publications) by Douglas J Smith, which comparatively illustrates VHDL and Verilog coding.

Chapter 2

FIFO - The Design Under Test

For this tutorial, the design under test (DUT) is a FIFO. This chapter introduces a FIFO, its inputs-outputs, functionality in details.

First-In-First-Out is a kind of queue used to temporarily store the data and retrieve it. As the name implies, the data that goes-in first, is retrieved first from the queue.

2.1 Input-Output Ports of FIFO

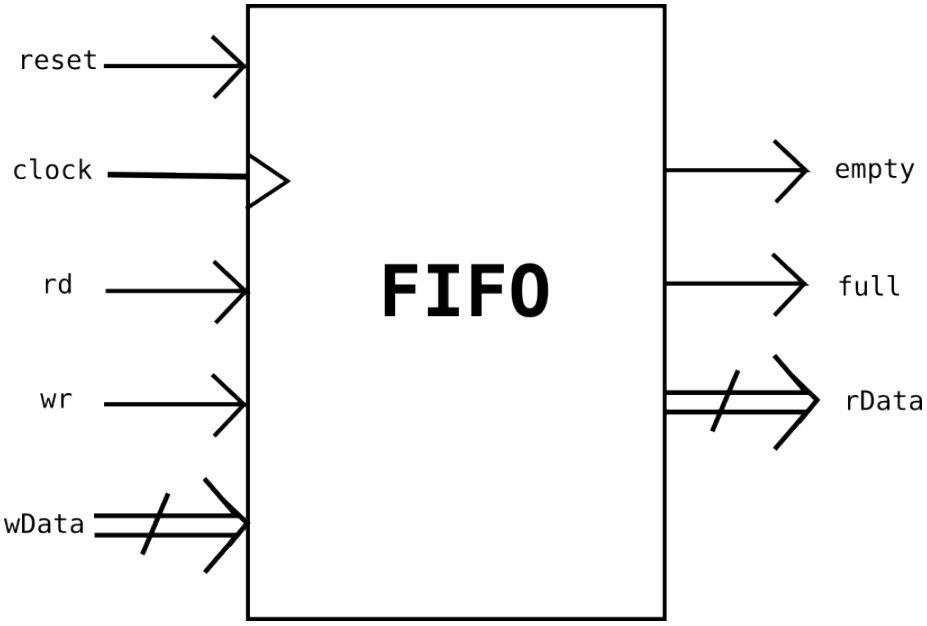


Figure 2.1: I/O ports of FIFO

The input ports are shown on left side pointing inside the block, while output ports appear on right side coming out from the block.

The following table explains meaning of the ports:

Port	Direction	width (bits)	Significance
reset	in	1	all internal registers are reset to zero (0).
clock	in	1	the synchronizing signal for FIFO operation.
rd	in	1	a 1-byte of data is read from FIFO by asserting this signal.
wr	in	1	a 1-byte of data is written to FIFO by asserting this signal.
wData	in	8	8-bit data to be written into FIFO.
empty	out	1	indicates that FIFO contains no data to read from.
full	out	1	indicates that FIFO has run out of storage.
rData	out	8	8-bit data read from the FIFO.

Table 2.1: Meaning of Input - Output Ports

2.2 Internal Register of FIFO

Let us consider a FIFO that can store upto 8-bytes of data. This requires eight 8-bit register to store data. Further, two registers are required to point to the top-of-the-FIFO and bottom-of-the-FIFO. And finally, one register is required to keep count of bytes contained in FIFO to indicate '*empty*' and '*full*' conditions.

The figure bellow shows a rough schematic of internal register. It does not show any interconnections or data-flow between them.

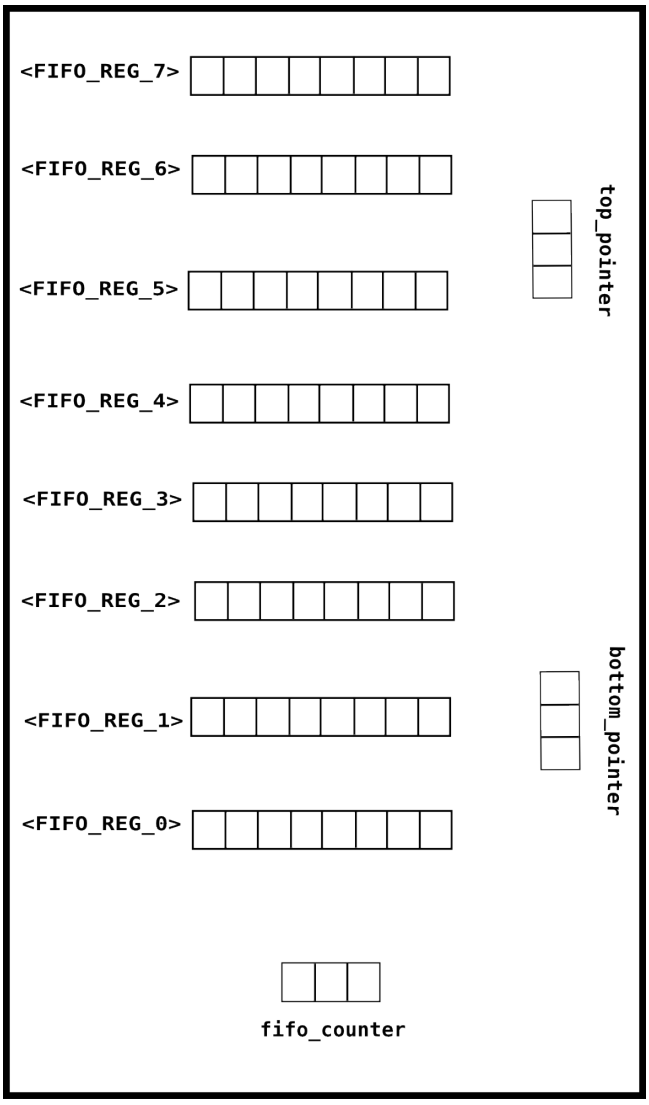


Figure 2.2: Internal Registers of FIFO

The *Pointer* registers are 3-bit long since we need 3-bit address to point to any fifo register. Also, the *Counter* register is 3-bit register that keeps count (from 000 to 111) of number of registers written at any time.

2.3 Operation of FIFO

Upon assertion of reset signal, all the registers including pointers and counter, are reset to 0. Then, at each rising edge of clock signal, following conditions are checked:

- If ***'rd = 1'*** and ***'wr = 0'***
  - The data pointed by bottom_register is placed on *rData* output port.
  - bottom_register is incremented by 1.
  - counter register is decremented by 1.

- If ***'rd = 0'*** and ***'wr = 1'***
  - The data present on *wData* input port is written to register pointed by top_register.
  - top_register is incremented by 1.
  - counter register is incremented by 1.

- If ***'rd = 1'*** and ***'wr = 1'***
  - The data pointed by bottom_register is placed on *rData* output port.
  - The data present on *wData* input port is written to register pointed by top_register.
  - bottom_register is incremented by 1.
  - top_register is incremented by 1.
  - counter register is unaffected.

The *full* and *empty* flag outputs are asserted for following conditions:

- full = '1' if
  - counter register = 111_b indicating that all registers are written, and
  - input wr = 1.

- empty = '1' if
  - counter register = 000_b indicating that no register are written, and
  - input rd = 1.

Note that *full* and *empty*, both can be '0' at any instant, but cannot be '1' simultaneously.

Chapter 3

FIFO : RTL and Testbench using VHDL

This chapter elaborates the design of FIFO described in previous chapter using VHDL and its verification using VHDL testbench.

3.1 VHDL RTL code for FIFO

3.1.1 library and package declaration

We use the library "ieee" and its package
i) std_logic_1164,
ii) std_logic_unsigned
for our design.Thus, the vhd file should start from :

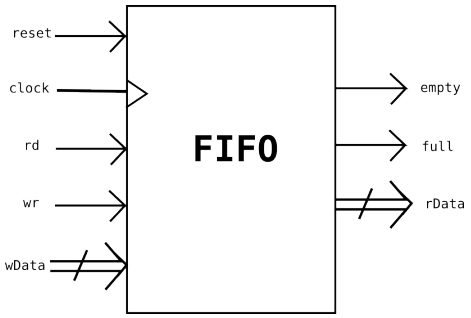
```
library ieee;
use ieee.std_logic_1164.all
use ieee.std_logic_unsigned.all
```

The package "std_logic_1164" allows us to use the 9 valued logic values to be assigned to the ports and internal signals of the design. By using the package "std_logic_unsigned", we can directly use '+' and '-' operations, as mentioned in architecture below.

3.1.2 entity

The entity defines the input-output ports.

For the I/O ports shown in figure below:



we define an entity with name "fifo" as:

```
entity fifo is
    port (
           clk, reset, rd, wr:   in     std_logic;
           wData :   in      std_logic_vector(7 downto 0);
           rData :   out      std_logic_vector(7 downto 0);
           full, empty :   out      std_logic
         );
end fifo;
```

Note that order in which ports are declared is not important. However, direction (in / out / inout) and their widths should be declared correctly.

3.1.3 architecture

In the architecture, we describe the design using certain *modeling style*. For FIFO, using "**Behavioral Modeling**" should be a wise choice. The design can be described at Register-Transfer level of abstraction in this style.

internal signals

As discussed in previous chapter, the FIFO consists of registers. These are defined as VHDL signals. We have two pointers (top pointer and bottom pointer) and a counter, all of width 3-bits. They can be defined as:

```
signal topPtr, btmPtr, cntr : std_logic_vector(2 downto 0);
```

Otherwise, all three signals (registers) can be defined separately as:

```
signal topPtr : std_logic_vector(2 downto 0);
signal btmPtr : std_logic_vector(2 downto 0);
signal cntr : std_logic_vector(2 downto 0);
```

This makes no difference to the design.

The 8 FIFO registers that store incoming bytes is defined as an array of eight 8-bit signals:

```
type fiforegtype is array(0 to 7) of std_logic_vector(7 downto 0);
signal fiforeg : fiforegtype := (others=> "00000000");
```

All eight registers are initialized to value zero using the := operator. The statement (others => "00000000") assigns the 8-bit zero value to each element of array.
To access 0^{th} register, we can use ***fiforeg(0)***, to access 1^{st} register, ***fiforeg(1)***, and so on!

All the signals are declared before '*begin*' statement of the architecture.

architecture body - design logic

The behavioral modeling method is identified by the *process* statement.

```
process (sen_list)
begin
  -----
  -----
end process;
```

The sen_list in brackets is the list (called sensitivity list) of all the signal that are responsible for executing that process statement. In other words, transition on any of the signals mentioned in sensitivity list will trigger the process statement.

When process is triggered, everything written withing *begin* and *end process* will be executed *sequentially*. Thus, process is a sequential statement in the concurrent VHDL language. Note that two process statements withing same architecture are executed concurrently.

Also, note that, if a signal is missed in sensitivity list, transition on that signal would have no effect on process, even if desirable.

The thumb rule is that, for combinational logic, sensitivity list contains all input signals (that appear inside process block), and for sequential logic, sensitivity list contains the clock and the reset signal.

Since FIFO is a sequential circuit, we use a process statement:

```
process (clk, reset)
begin

end process;
```

Rest of the code explained below resides within this process block.

For sequential circuits with a reset signal, first of all the reset state of the system should be defined. In case of our FIFO, when reset is asserted i.e. when reset is high, all internal registers as well as all outputs are reset to zero. This is written in VHDL with if-statement as:

```
if (reset = '1') then
    topPtr <= "000";
    btmPtr <= "000";
    cntr <= "000";

    fiforeg(0) <= "00000000";
    fiforeg(1) <= "00000000";
    fiforeg(2) <= "00000000";
    fiforeg(3) <= "00000000";
    fiforeg(4) <= "00000000";
    fiforeg(5) <= "00000000";
    fiforeg(6) <= "00000000";
    fiforeg(7) <= "00000000";

    full <= '0';
    empty <= '0';

    rData <= "00000000";
```

If reset is low, then next code follows. Remaining functioning of circuit is on rising edge of the clock signal.

Hence in the else-part of if-statement, we check whether clock is rising:

```
elsif (clk'event and clk = '1') then
```

Now, we check for the 'rd' and 'wr' inputs.

If 'wr' signal is asserted, and if counter is less than 111_b, then input wData is written to next available FIFO register (i.e. at register pointed by top pointer), the top pointer is incremented, and counter is also incremented. This is written as:

```
if (rd = '0' and wr = '1') then
   if (cntr < "111") then
      fiforeg(conv_integer(topPtr)) <= wData;
      topPtr <= topPtr + "001";
      cntr <= cntr + "001";
    end if;
```

Here, *topPtr* is a binary value while we have used integer for indexing *fiforeg* array. That is why the VHDL function **conv_integer** is used which converts binary to integer. Also, the '+' operator is supported by the "*std_logic_unsigned*" package.

Similarly, when 'rd' signal is asserted, then content of FIFO register pointed by bottom pointer is placed on rData output, the bottom pointer is incremented while counter is decremented. This falls under else part of above if-statement:

```
elsif (rd = '1' and wr = '0') then
    if (cntr > "000") then
        rData <= fifoReg(conv_integer(btmPtr));
        btmPtr <= btmPtr + "001";
        cntr <= cntr - "001";
     end if;
```

Finally, we check if both 'rd' and 'wr' are asserted simultaneously. In this case, the counter remains unchanged while both pointers incremented. wData is written into FIFO and rData reads from FIFO. This is last else part:

```
elsif (rd = '1' and wr = '0') then
     rData <= fifoReg(conv_integer(btmPtr));
     btmPtr <= btmPtr - "001";
     fiforeg(conv_integer(topPtr)) <= wData;
     topPtr <= topPtr + "001";
end if;
```

Now, the two output flags empty and full are described with following code. For *empty* to be high, the condition is that no FIFO register has data to read from but 'rd' input is asserted. Thus:

```
    if (cntr = "000" and rd = '1') then
        empty <= '1';
        rData <= (others => 'Z');
    else
        empty <= '0';
    end if;
```

When 'rd' signal is asserted to an empty FIFO, the rData output should present a High Impedance.

And, for *full* to be high, the condition is that all FIFO registers are written and 'wr' signal is asserted to write new byte. Thus:

```
    if (cntr = "111" and wr = '1') then
        full <= '1';
    else
        full <= '0';
    end if;
```

This ends the process block.

Complete architecture

The complete code of architecture is:

```
architecture behav of fifo is
    signal topPtr, btmPtr : std_logic_vector(2 downto 0);
    signal cntr : std_logic_vector(2 downto 0);

    type fiforegtype is array(0 to 7) of std_logic_vector(7 downto 0);
    signal fiforeg : fiforegtype := (others=> "00000000");
begin

    process (clk, reset)
    begin

    if (reset = '1') then
      topPtr <= "000";
  btmPtr <= "000";
  cntr <= "000";
  fiforeg(0) <= "00000000";
  fiforeg(1) <= "00000000";
  fiforeg(2) <= "00000000";
  fiforeg(3) <= "00000000";
  fiforeg(4) <= "00000000";
```

```
  fiforeg(5) <= "00000000";
  fiforeg(6) <= "00000000";
  fiforeg(7) <= "00000000";
  full <= '0';
  empty <= '0';
  rData <= "00000000";
    elsif (clk'event and clk = '1') then

      if (rd = '0' and wr = '1') then
        if (cntr < "111") then
            fiforeg(conv_integer(topPtr)) <= wData;
            topPtr <= topPtr + "001";
            cntr <= cntr + "001";
            end if;
        elsif (rd = '1' and wr = '0') then
            if (cntr > "000") then
            rData <= fifoReg(conv_integer(btmPtr));
            btmPtr <= btmPtr + "001";
            cntr <= cntr - "001";
            end if;
        elsif (rd = '1' and wr = '0') then
            rData <= fifoReg(conv_integer(btmPtr));
            btmPtr <= btmPtr - "001";
            fiforeg(conv_integer(topPtr)) <= wData;
            topPtr <= topPtr + "001";
        end if;

        if (cntr = "000" and rd = '1') then
            empty <= '1';
            rData <= (others => 'Z');
        else
            empty <= '0';
        end if;

        if (cntr = "111" and wr = '1') then
            full <= '1';
        else
            full <= '0';
        end if;
    end if;
  end process;
end behav;
```

3.2 Verification of FIFO

Now that we have written a VHDL code for the FIFO, let us check whether it is functionally correct, with the help of a VHDL Testbench.
A testbench is another .vhd file from which we feed out RTL file the inputs and read outputs from it. This tutorial shows two ways of writing a VHDL Testbench.

3.2.1 A VHDL Testbench for FIFO

The file structure of Testbench is just same as that of RTL file. We start from library and package declaration:

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

Note that we have not used std_logic_unsigned package here, but another package *std_logic_textio*. This package contains functions to access a text file. So, in the testbench, we would read an external text file.

Next is the entity declaration. The entity of a VHDL Testbench does not have a port.

```
entity tb_file is end;
```

As simple as that! You could give any name instead of *tb_file*.

Then, in the architecture, we define a *component*. This component is the design under test (DUT) i.e. the FIFO. Ports of component are exactly same as ports of DUT. And, the name of component is same name of entity of DUT.

```
component fifo is
    port (
          clk, reset, rd, wr: in std_logic;
          wData : in std_logic_vector(7 downto 0);
          rData : out std_logic_vector(7 downto 0);
          full, empty : out std_logic
          );
end component;
```

Then, every port of component is declared as a *signal* of same width.

```
signal clk, reset, rd, wr: std_logic;
signal wData :  std_logic_vector(7 downto 0);
signal rData :  std_logic_vector(7 downto 0);
signal full, empty :  std_logic;
```

The sequence in which these are defined is not important! The component and signals are declared before *begin* statement of architecture.

In architecture body, we *instantiate* the component. Syntax of instantiation is:

```
<intance_name> : <component_name> port map (<port_list_in_correct_sequence>);
```

Thus, we instantiate our FIFO with name dut as:

```
dut : fifo port map (clk, reset, rd, wr, wData, rData, full, empty);
```

Note that, port map should contain the port list in the exactly same order as in port list of component. Failing this will cause an error.

Next we write a process for clock signal:

```
process
begin
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
end process;
```

The process does not have a sensitivity list here. This is because we use *wait for* statement inside the process block. A process should have either a sensitivity list or a wait - statement, but cannot have both.

In process, we are toggling the clock signal after 10 nano-seconds, forever. Another time value can be chosen to shorten or elongate the clock period, and also, ON and OFF time of clock can be different.

Then, we write another process to stimulate out DUT. Before that, let us discuss the text file we are going to read in that process. This is a simple text file, say "data.txt". We store the values of inputs to be given and the expected values outputs. Since, we have already defined a process for clock, it's not needed to define it in this file.

As a simple example, lets put time, reset, rd, wr, wData and rData in the file. The file should look like:

```
-- anything starting with double-dash is a comment
-- time reset rd wr wData rData
000.00 1 0 0 00000000 00000000
010.00 1 0 0 00000000 00000000
020.00 0 0 1 00110010 00000000
030.00 0 0 0 00000000 00000000
040.00 0 1 0 00000000 00110010
050.00 0 0 1 11111111 00110010
060.00 0 0 1 00110011 00110010
070.00 0 1 0 00000000 11111111
```

```
080.00 0 0 1 00001111 11111111
090.00 0 1 0 00000000 00110011
100.00 0 0 0 00000000 00110011
110.00 0 1 0 00000000 00001111
120.00 0 0 0 00000000 00001111
```

The time is specified in nanoseconds. Other values are considered binary. Lets now see how we deal with this file in our testbench.

Before the *begin* statement of process, we define some VHDL variables.

```
variable tempclk, tempreset, temprd, tempwr, full, empty : std_logic;
variable tempwData, temprData : std_logic_vector(7 downto 0);

file vector_file : text is in "data.txt";

variable tm : time;
variable r : real;
variable s : character;
variable l : line;
variable gv,gn : boolean;
```

Next code appears inside process block:

```
while not endfile(vector_file) loop

readline (vector_file, l);

read (l, r, good => gn);

assert gn report "bad value of TIME";

next when not gn;
```

This starts a loop which continues until end of file is reached. A line is read from the file. Then, it is checked whether the line starts with a boolean value (gn is boolean). When not, an error is asserted that "bad value of TIME", and the line is skipped. Thus, if a line does not start with time, it is skipped, That is how we have written comments starting with double-dash.

Next, convert the time value into nanoseconds and assign it to our variable *tm*, and wait for that much of time:

```
tm := r * 1 ns;
if (now < tm) then
    wait for tm-now;
end if;
```

Note that, *read (l, r)* will read a real value (r is real). Next we read characters with variable s:

```
read (l, s);
read (l, tempreset, gv);
assert gv report "bad value of RESET";

read (l, s);
read (l, temprd, gv);
assert gv report "bad value of 'rd'";

read (l, s);
read (l, tempwr, gv);
assert gv report "bad value of 'wr'";

read (l, s);
read (l, tempwData, gv);
assert gv report "bad value of 'wData'";

read (l, s);
read (l, temprData, gv);
assert gv report "bad value of 'rData'";
```

And then, assign this values read from file to the input signal:

```
wData <= tempwData;
rd <= temprd;
wr <= tempwr;
reset <= tempreset;
```

and, end the loop:

```
end loop;
```

So far, we have done following:

- read a *real* value, multiply by 1 nanosecond, and assign it to time variable tm.
- read a character as value of *reset* signal
- read a character as value of *rd* signal
- read a character as value of *rw* signal
- read a character as value of *wData* signal
- read a character as value of *rData* signal
- Assign all read values to inputs of DUT

The RTL code of DUT is executed by the simulator. The output rData from DUT is collected by the Testbench. Now, we should verify that the value received from DUT is same as that stored in text file (i.e. expected value).

```
assert (temprData = rData) report " VECTOR MISMATCH . . . TEST FAILED !";
```

When the end of file is reached, we assert that simulation is completed, and put a wait statement (to wait forever).

```
assert false report " SIMULATION COMPLETE !";
wait;
```

This is the end of another process. The complete architecture of this testbench:

```
architecture testbench of tbf is
    component fifo is
        port (
            clk, reset, rd, wr: in std_logic;
            wData : in std_logic_vector(7 downto 0);
            rData : out std_logic_vector(7 downto 0);
            full, empty : out std_logic );
    end component;

    signal clk, reset, rd, wr: std_logic;
    signal wData :  std_logic_vector(7 downto 0);
    signal rData :  std_logic_vector(7 downto 0);
    signal full, empty :  std_logic;
begin

uut : fifo port map(clk, reset, rd, wr, wData, rData, full, empty);

process

variable tempclk, tempreset, temprd, tempwr, full, empty : std_logic;
variable tempwData, temprData : std_logic_vector(7 downto 0);
file vector_file : text is in "data.txt";
variable tm : time;
variable r : real;
variable s : character;
variable l : line;
variable gv,gn : boolean;

begin

while not endfile(vector_file) loop
readline (vector_file, l);
```

```
    read (l, r, good => gn);
    assert gn report "bad value of TIME";
    next when not gn;

    tm := r * 1 ns;
    if (now < tm) then
        wait for tm-now;
    end if;

    read (l, s);
    read (l, tempreset, gv);
    assert gv report "bad value of RESET";
    read (l, s);
    read (l, temprd, gv);
    assert gv report "bad value of 'rd'";
    read (l, s);
    read (l, tempwr, gv);
    assert gv report "bad value of 'wr'";
    read (l, s);
    read (l, tempwData, gv);
    assert gv report "bad value of 'wData'";
    read (l, s);
    read (l, temprData, gv);
    assert gv report "bad value of 'rData'";

    wData <= tempwData;
    rd <= temprd;
    wr <= tempwr;
    reset <= tempreset;

    end loop;

    assert (temprData = rData) report " VECTOR MISMATCH . . . TEST FAILED !";

    assert false report " SIMULATION COMPLETE !";
    wait;
    end process;

    process
    begin
        clk <= '1';
        wait for 5 ns;
        clk <= '0';
        wait for 5 ns;
    end process;
end testbench;
```

The advantage we get from this kind of Testbench is that we can *assert* whenever there is output mismatch to easily identify the error. However, it is required that the *data.txt* file be written very carefully. To check every condition, like empty and full FIFO, tests must be written accordingly.

3.2.2 Another VHDL Testbench for FIFO

The another VHDL Testbench we shall discuss is much simple than previous one. The library and package declaration:

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

The entity again has no ports:

```
entity tb is end;
```

And, the architecture contains a component which is instantiated same as in previous testbench. The process for clock signal is also the same. The only difference is in stimulus process:

```
process
begin
    reset <= '1';    rd <= '0'; wr <= '0'; wData <= "00000000";
    wait for 40 ns;

    rd <= '0';    wr <= '1';   wData <= "00110101";
    wait for 20 ns;

    rd <= '1'; wr <= '0'; wData <= "00000000";
    wait for 20 ns;

    -- assign another set of values to inputs
    -- wait for some time

    -- assign another set of values to inputs

    wait; -- finally, wait forever
end process;
```

Thus, we just stimulate the inputs, and output is observed in the waveforms. Any errors should be then identified by observations. It is not impossible to match outputs with expected values within testbench. We could write *assert* statements here too.

Writing various possible combinations is important while writing testbench using either approach, to check whether RTL is functionally correct.

Chapter 4

FIFO : RTL and Testbench using Verilog

In this chapter, the FIFO is designed using the Verilog-HDL. Also, testbench is written using Verilog to test functionality of the design.

4.1 Verilog RTL code for FIFO

For verilog file structure, refer section 1.3.3 and 1.3.4. The module for our FIFO is:

```
module (clk, reset, rd, wr, wData, full, empty, rData);

   input clk, reset, rd, wr;
   input [7:0] wData;
   output full, empty;
   output [7:0] rData;

    -----
    -----
    -----

endmodule
```

Other way is to mention port direction like:

```
module (input clk, input reset, input rd, input wr, input [7:0] wData,
         output full, output empty, output [7:0] rData);

    -----
    -----
    -----

endmodule
```

Internal registers were defined as *signal* in VHDL. Here, they are defined as *reg*:

```
reg [3:0] topPtr, btmPtr, cntr;
reg [7:0] fifoReg [0:7];
```

Note the fifoReg register array. In VHDL, we defined an array type and then a signal for the same type. Here, *reg [7:0]* makes an 8-bit register, and [0:7] makes an array of the 8-bit registers.

The outputs are also defined as registers.

```
reg empty, full;
reg [7:0] rData;
```

By defining this, the outputs become *registered outputs*, and we can assign values to them inside *process-like* blocks of Verilog. The process-like block is called *always block*. It is written as:

```
always @ (sen_list)
begin

end
```

The sensitivity list of always block is separated by 'or', unlike comma (,) as in VHDL. The parameters to be put in sensitivity list are same as VHDL i.e. clock and reset for sequential logic, and all relevant inputs for combinational logic.

For FIFO, the always block is:

```
always @ (posedge clk or posedge reset)
begin

end
```

The edge of clock signal is mentioned in sensitivity list (unlike in if-condition for VHDL). Thus, the always block is triggered for rising edges (positive edge / leading edge) of reset and clock signals.

Then, we check the presence of reset signal. If reset is high then all registers are reset to zero.

```
if (reset)
begin
   topPtr <= 3'b0;
   btmPtr <= 3'b0;
   cntr <= 3'b0;

   fifoReg[0] <= 7'b0;
   fifoReg[1] <= 7'b0;
   fifoReg[2] <= 7'b0;
   fifoReg[3] <= 7'b0;
   fifoReg[4] <= 7'b0;
   fifoReg[5] <= 7'b0;
   fifoReg[6] <= 7'b0;
   fifoReg[7] <= 7'b0;

   rData <= 7'b0;
end
```

Since there are multiple statements for if-statement, they are enclosed inside begin-end statements.

Note that *then* keyword (as in VHDL) is not used in Verilog if-statement. Also, the condition is written as (reset) which means if reset is equal to 1. This can also be written as (reset == 1'b1).

Next logic follows when reset is low, in else part:

```
else begin
case ({rd, wr})

2'b 01 : if (cntr < 3'b111)
          begin
          fifoReg[topPtr] <= wData;
          topPtr <= topPtr + 1'b1;
          cntr <= cntr + 1'b1;
          end

2'b 10 : if (cntr > 3'b000)
          begin
          rData <= fifoReg[btmPtr];
          btmPtr <= btmPtr +1'b1;
          cntr <= cntr - 1'b1;
          end

          else
             rData <= 8'bZZZZZZZZ;

 2'b 11 :
          begin
          fifoReg[topPtr] <= wData;
          topPtr <= topPtr + 1;
          rData <= fifoReg[btmPtr];
          btmPtr <= btmPtr +1;
          end

  default : ;
  endcase
end // this is end of if-else statement
```

In else-part (i.e. on positive edge of clock), we check rd and wr signals. rd, wr is concatenation operation of these two inputs. So, case (rd, wr) is 2'b 01 means when read = 0, and wr = 1. Next logic is just similar to that in VHDL (refer section 3.1.3).

Here, we have not mentioned full and empty flag outputs. For them, we write another always block.

```
always @ (posedge clk or posedge reset)
begin

if (reset) begin
     full <= 1'b0;
     empty <= 1'b0;
     end
else begin
     if (wr & ~rd & (cntr==3'b111))
        begin
           full <= 1'b1;
           empty <= 1'b0;
           end
     else if (wr & ~rd & (cntr==3'b111))
         begin
           empty <= 1'b1;
           full <= 1'b0;
           end
   end
  end
```

The condition (wr & rd & ($cntr == 3'b111$)) means wr is 1_b, rd is 0_b, and cntr is 111_b

In Verilog, the same register cannot be assigned in two different always blocks. That is why, we have not written rData in this always block.
The statement $rData <= 8'bZZZZZZZZ$ for rd on empty FIFO appeared in first always block. These two always blocks could be merged into single block.

4.2 Verilog Testbench for FIFO

Like the VHDL Testbench does not have ports in entity, the Verilog Testbench module does not have a port list. So, a testbench, say named tb, starts with

```
module tb;
```

and, ends with

```
endmodule
```

Every input of the DUT is declared as a *reg* and every output as a *wire* in the testbench:

```
reg clk, reset, rd, wr;
reg [7:0] wData;
wire full, empty;
wire [7:0] rData;
```

and, the DUT is instantiated with syntax:

```
<DUT_name> <instance_name> (port_list);
```

Thus, an instance of DUT, named DUT for our FIFO is:

```
fifo DUT (clk, reset, rd, wr, wData, full, empty, rData);
```

Note that, instance name is written first in VHDL Testbench, while in Verilog Testbench DUT name is written first. Also, the port list should be in a correct order. Verilog does not indicate an error if port list is not in correct order, since Verilog is not typed language!

Like we wrote *process* statements without a sensitivity list in VHDL Testbench, here we use same concept, *always* statements without sensitivity list. The *wait for* and *wait* statements are replaced here with # (hash) which indicates delay. For example,

```
always
begin
    reset = 1'b1; rd = 1'b0; wr = 1'b0; wData = 8'b00000000;
#20 reset = 1'b0; rd = 1'b0; wr = 1'b1; wData = 8'b11000101;
#20 reset = 1'b0; rd = 1'b1; wr = 1'b0; wData = 8'b00000000;
end
```

Another important statement in Verilog Testbench is *initial* which is executed only once in the simulation.

```
initial
clk = 1'b0;
```

This assigns clock signal a zero value at the start of simulation i.e. at time 0 ns. To toggle the clock at 10 ns interval:

```
always
#10  clk = ~ clk;
```

To stop or finish the simulation at 3000 ns, we could write:

```
initial
#3000  $stop;
```

or

```
initial
#3000  $finish;
```

Assertions were used in VHDL to print messages on screen. Verilog supports *$monitor* statement for printing the signal valeus:

```
$monitor ($time, "ns");   // this will print time instant e.g. 0 ns

$monitor ("rd = %b", rd);  // This prints value of rd signal
            in binary notation. To print in hexadedcimal notation:
$monitor ("wData = %h", wData);
```

This is similar to *printf()* statements in ANSI C.