CHAPTER 3. Test Bench development

3.1 Introduction

When modeling systems with VHDL, a comprehensive method of testing must be developed which will test the aspects of the design completely. VHDL models are tested using an enclosing model called a ***Test Bench***. A VHDL test bench can be defined as an executable VHDL model which instantiates a ***model under test (MUT)*** and provides the capability to drive the MUT with a set of test vectors and compare the response with the expected response. The test bench is at the top level in the model hierarchy. It is the entity that is simulated when testing a model [5]. Test benches provide the user with the capability to test the MUT thoroughly through simulation.

The VHDL test bench developed must serve the following purposes :

- Simulate the MUT under a variety of test conditions including correct and faulty test inputs, VHDL configurations, maximum/minimum delays and fault conditions.

- Automatically verify that the MUT meets the specifications and log all the errors if it does not meet the specifications. This could be achieved by comparing the output response of the MUT with that of a GOLD model and report success or failure for each test. On-line verification checks the performance of the MUT accurately and also consumes less time than manual verification which is very tedious, cumbersome and inefficient. In addition to the above mentioned advantages, automation also allows for a reduction in time for future maintenance effort because it enables fast and reliable verification of the model when changes are introduced.

- The test bench should be capable of performing regression tests. These are tests used to ensure that a design change in the system does not adversely effect the capabilities of the current design. Regression testing combines the issues of completeness associated with signoff testing and the assumption that there will be changes in the model under test implied by tradeoff analysis.

At higher levels of abstraction, the primary purposes of simulation are to provide executable specifications and to verify the functionality of the design. In addition, at lower levels of abstraction, simulation is used to verify that a set of tests will detect common hardware faults. This is accomplished by exercising the design using a number of stressing test cases. Because of the complexity of the tests, it is a good practice to identify the goals of the verification before the actual simulation activity and to devise a test plan which ensures that the test bench meets the desired goals. The different characteristics of a general test plan have already been explained in section 2.2.

A good set of test vectors must be developed. While exhaustive testing can be adopted to test models of lesser complexity, this could be an impractical method for testing large systems. Common strategies for testing the model at the behavioral level of abstraction are to select tests that will exercise all control flow paths in the model. Paths involving nested control constructs, wait statements and nested procedure calls should be specifically targeted by the test vectors. Test sets developed to test the models at higher levels of abstraction could be reused for testing the models at lower levels. However, additional test sets usually have to be developed to detect specific faults in the lower level models.

Tests should be developed to verify the functionality of the system completely. This includes tests of components of a design as well as tests for the design as a whole. It is important to test each component as it is created, and not to assume correct operation. As components are brought together to form a new level of hierarchy, interdependencies between components may also cause unexpected behavior and so simulations must be performed at this level. Finally the entire design is brought together and testing of the design as a whole may begin, with the knowledge that most of the errors should have been found by testing individual components.

A VHDL test bench typically consists of an architecture body containing an instance of the component to be tested and processes that generate sequences of values on signals connected to the component instance. The architecture body may also contain processes that verify that the component instance produces the expected values on the output signals. Alternatively, we may use the monitoring facilities of the simulator to observe the outputs visually.

The Sobel edge detector was developed and tested at the Behavioral, RTL and Gate levels. At the Behavioral level, a model is described in terms of its input/output response, whereas at the RTL and Gate level the models are described in terms of the interconnected registers and gates respectively.

3.2 Test bench development for a behavioral model

3.2.1 Methodology

This section explains the methodology adopted for developing a test bench for an executable specification.

As stated in the previous section, the main purpose of testing at the behavioral level is to validate the functionality of the executable specification and also define the correct responses to the test vectors, so that, while testing the MUT at the lower levels its response at that level can be compared with the output response of the MUT at the behavioral level.

The test bench module (TBM) typically performs the following functions :

1. At the start of the simulation, the TBM reads the input image data from an external text file into an internal frame buffer and sends it to the MUT in an "appropriate format ", i.e., the input test vectors in the external file could be of type *integer*, but the ports of the MUT may be of *std_logic* data type, which requires the TBM to convert the input data to *std_logic* data type and then send it to the MUT.

2. Once the data has been processed by the MUT, its output responses are stored in internal frame buffers in the TBM. The main function of the TBM here is to decide the time when the output response is to be written into the frame buffers.

3. Finally, the output responses from the frame buffers are written into external text files which are then read by the comparators to compare the output response of the MUT with the expected response and generate a PASS/FAIL indication.

Having identified the basic functions of the TBM, the sub-components needed to implement them are then chosen :

- Clock component : This component is required to drive the entire system.
- MUT : This is the model under test
- Comparator : This component is used to compare the output response of the MUT with the expected response.

3.2.2 Implementation

To illustrate the methodology in section 3.2.1, a test bench model has been developed for the behavioral implementation of the ***Sobel Edge Detector***. This chip implements the Sobel edge detection algorithm which is commonly used in the image segmentation phase of image processing systems. Details of the Sobel algorithm can be found in section 2.3 of this thesis.

3.2.2.1 Test Setup

Figure 3.1 represents the diagram of the test bench module (TBM) used for testing the Sobel Edge Detector component. It includes the MUT, a Clock generator to provide the clock for the entire system, and two comparators to verify the correctness of the outputs.

The stimulus values or test vectors used to test the MUT have been provided in a file, specified as the *INPUT_FILE* in Figure 3.1.The test bench module reads the input image data from the image file and stores it in an internal frame buffer.

Three signals are sent out from TBM to the MUT. The first signal is, ***INPUT*** which carries the image data obtained from the *input image file*, the second signal is ***EDGE_START*** which triggers the MUT to start processing the image data and the third signal is ***THRESHOLD*** which is used to detect if the output pixel is an edge. The TBM drives the Clock by means of a signal ***RUN*** and the output of the Clock generator which is ***CLOCK*** is mapped to the input port *CLOCK* of the MUT. The comparators are triggered by means of a signal ***COMP_START***.

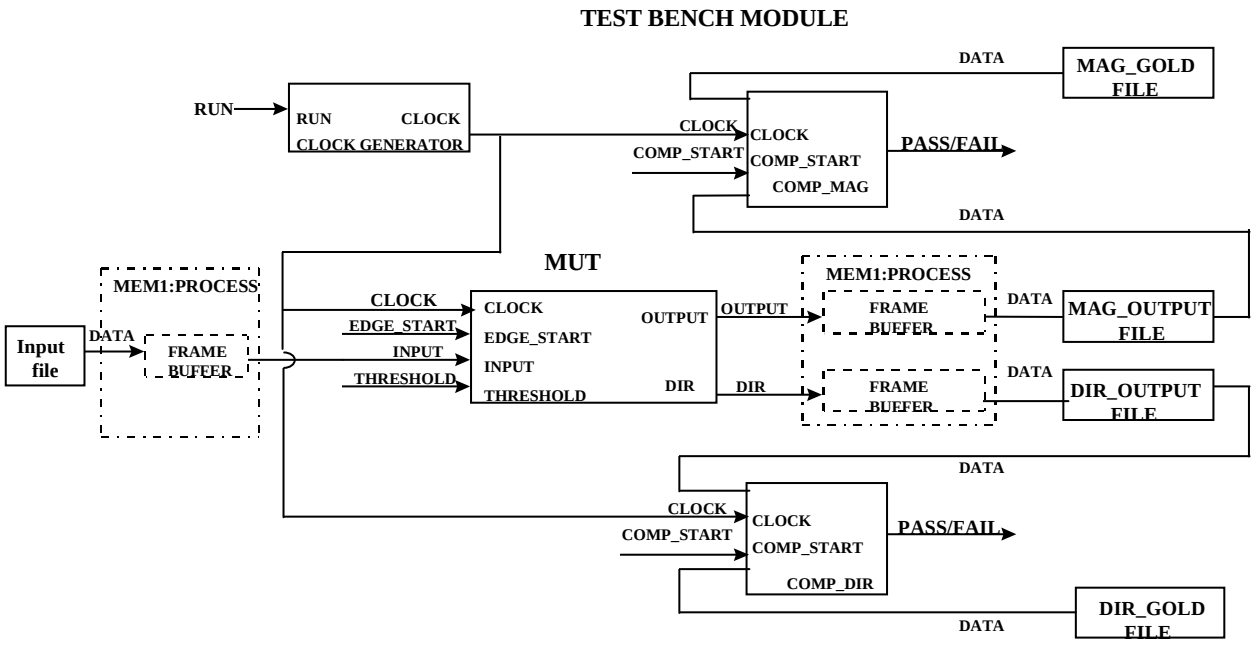


Figure 3.1 Test bench module for Sobel Edge detector

The MUT has two output signals namely ***OUTPUT*** and ***DIR*** which carry the magnitude and direction outputs respectively. These data are sent to the test bench module where they are stored in internal frame buffers and subsequently in separate text files: MAG_OUTPUT_FILE and MAG_DIR_FILE. These outputs are then compared with the magnitude and direction outputs of the ***GOLD*** model in the two separate comparators each of which generates a ***PASS/FAIL*** indication.

3.2.2.2 Clock Generator component

Figure 3.2 shows the block diagram and the input and output waveforms for the Clock generator component used to provide the clock pulses for the entire system.

The Clock generator model has one input namely ***RUN*** and one output ***CLOCK***. From the waveforms we can see that as soon as the input signal *RUN* rises, the clock output is scheduled, and it remains high for the time period specified by ***HI_TIME*** and low for the time period specified by ***LO_TIME***. The sum of HI_TIME and LO_TIME is the clock period.

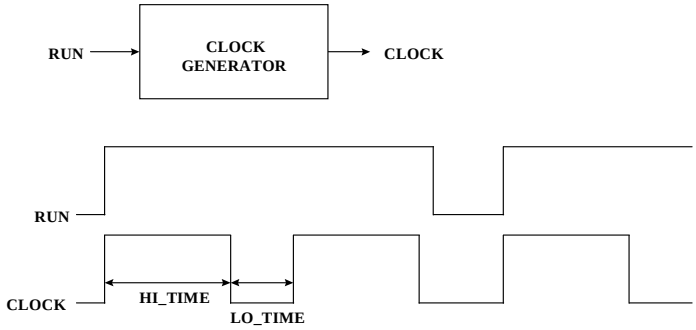


Figure 3.2 Block diagram and input and output waveforms of the Clock Generator component.

Figure 3.3 shows the entity declaration and behavioral architecture body of the Clock Generator .

```
 library IEEE;
use IEEE.STD_LOGIC_1164.all;
 ---------- interface declaration ----------
entity CLOCK_GENERATOR is
   generic(HI_TIME: TIME;                -- the time period when CLOCK is '1'
           LO_TIME: TIME);               -- the time period when CLOCK is '0'
   port (RUN : in STD_LOGIC;             -- the start signal
         CLOCK: out STD_LOGIC);          -- system CLOCK
end CLOCK_GENERATOR;
---------- behavioral description ----------
architecture BEHAVIOR of CLOCK_GENERATOR is
begin
process
        begin
        ------ when RUN='1', generate clock signal ------
        wait until RUN='1';
        while RUN='1' loop
          CLOCK<='1' ;
          wait for HI_TIME;
          CLOCK<='0';
          wait for LO_TIME;
```

```
        end loop;
end process;
end BEHAVIOR;
```

Figure 3.3 Entity and architecture of the Clock generator component.

The entity has two generic constants that are used to specify the shape of the clock waveforms. ***HI_TIME*** specifies the pulse width and ***LO_TIME*** specifies the pulse separation. Instead of hard-coding the timing values in the architecture of the Clock Generator we make use of generics since they promote code reusability and increase the usefulness and lifetime of a design. By providing the actual values of the constants much later in the design cycle, for example in a configuration file, the model can adapt to a variety of environments.

The architecture body contains a process to generate the clock signal. When the signal *RUN* is '1' the clock generator remains high for time period specified by *HI_TIME* and low for time period specified by *LO_TIME*.

3.2.2.3 Sobel Edge Detector component

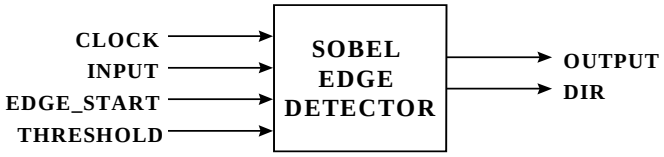


Figure 3.4 Block diagram of the Sobel Edge Detector

Figure 3.4 represents the block diagram of the Sobel Edge detector component. It has four input signals namely ***CLOCK***, ***INPUT***, ***EDGE_START*** and ***THRESHOLD***. The output signals are ***OUTPUT*** and ***DIR*** which are the magnitude and direction outputs of the Edge detector and these are sent to the test bench module. Details of the Sobel Edge detector model can be found in [16].

3.2.2.4 Test Bench Module (TBM)

The source code of the test bench module of the Sobel Edge detector model was divided into several sections for easier readability. The section of the code in Figure 3.5 shows the entity declaration and part of the architecture body. The entity declaration has no port list, since the test bench is self-contained. Various generic constants are used within the entity to define the input and output text files. These are explained in the source code. Apart from defining the input and output files, we also use two generics ***NUM_ROWS*** and ***NUM_COLS*** to define the size of the input image and use the generic constant ***WAIT_CYCLES*** to specify the additional time we need to wait before writing the output values of the Edge detector into the output frame buffer. This is necessary so as to avoid writing the initial invalid data into the frame buffer. This value is

defined as a generic as opposed to a constant, since the number of clock cycles we need to specify varies with the abstraction level. For e.g. when we simulate the model at the behavioral level we specify wait_cycles as 5; however when simulating the model at the RTL level we need to include an additional 7 clock cycles.

Inside the architecture body, two array types namely ***FRAME_IMAGE*** and ***FRAME_DIRECTION*** have been defined. This can be seen in the portion of the source code labeled as (***ARCHITECTURE BODY).*** *FRAME_IMAGE* represents the input and output frame buffers of integer data type. *FRAME_DIRECTION* represents the output frame buffer of bit_vector data type.

The architecture body further includes the component declarations corresponding to the clock generator, edge detector and comparators. The component Clock generator, Edge detector and the comparators are instantiated and their port signals are mapped with port map statements as shown in Figure 3.5.The local signals in the architecture body are the same as the ports of the components edge detector, clock generator and the comparators. An additional signal ***START*** has been included to trigger the test bench module.

```
 library IEEE;
use ieee.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
library BEH_INT;
use BEH_INT.IMAGE_PROCESSING.all;
-------------------------interface declaration---------------------------------
entity TEST is

generic( IN_FILE          :STRING(1 to 11); -- INPUT image file
         OUT_FILE         :STRING(1 to 11); -- OUTPUT file for storing magnitude Outputs
         DIR_FILE         :STRING(1 to 11); -- OUTPUT file which stores Direction Outputs of BIT_VECTOR type
         NUM_ROWS         :NATURAL;          -- number of rows in the INPUT image
         NUM_COLS         :NATURAL;        -- number of columns in the INPUT image
         WAIT_CYCLES      :NATURAL);       -- time required before Outputs  are written into frame buffers.
end TEST;

----------------------architecture description---------------------------------
architecture BENCH of TEST is
type FRAME_IMAGE is array(1 to NUM_ROWS,1 to NUM_COLS) of INTEGER;
type FRAME_DIRECTION is array(1 to NUM_ROWS,1 to NUM_COLS) of BIT_VECTOR(2 downto 0);
----------------------component instantiations---------------------------------
component CLOCK_GENERATOR1
         port( RUN : in STD_LOGIC;
         CLOCK: out STD_LOGIC);
end component ;

component EDGE_DETECTOR1
   port (  CLOCK: in STD_LOGIC;
         EDGE_START : in STD_LOGIC;
         INPUT: in PIXEL;
```

```
        THRESHOLD: in FILTER_OUT;
        OUTPUT: inout PIXEL;
        DIR: inout DIRECTION);
end component;

component COMP_MAG1
        port(CLOCK : in STD_LOGIC;
            COMP_START : in STD_LOGIC:='0');
end component;

component COMP_DIR1
        port(CLOCK : in STD_LOGIC;
            COMP_START : in STD_LOGIC:='0');
end component;
 ------------------------------Signal declarations-------------------------------
signal RUN        : STD_LOGIC:='0';
signal CLOCK      : STD_LOGIC:='0';
signal START      : STD_LOGIC:='0';
signal INPUT, OUTPUT: PIXEL:=0;
signal THRESHOLD: FILTER_OUT:=0;
signal DIR        : DIRECTION;
signal EDGE_START: STD_LOGIC:='0';
signal COMP_START : STD_LOGIC:='0';

begin

START<='0' after 0 ns,'1' after 5 ns,'0' after 105 ns;
THRESHOLD <= 110 after 0 ns;
RUN<=transport '1' after 0 ns,'0' after 100000 ns;

P1 : CLOCK_GENERATOR1
port map(RUN,CLOCK);

P2 : COMP_MAG1
port map(CLOCK,COMP_START);

P3 : COMP_DIR1
port map(CLOCK,COMP_START);

P4 : EDGE_DETECTOR1
port map(CLOCK,EDGE_START,INPUT,THRESHOLD,OUTPUT,DIR);

-------------------------------------Processes---------------------------------
MEMORY1 : process
-- DETAILS IN FIGURE 3.6 & 3.9
end process;
end BENCH;
```

Figure 3.5 Entity and Architecture declarations of the test bench

The architecture body contains a process: ***MEMORY1***. *MEMORY1* process is for implementing the functions of the test bench detailed in section 3.2.1.

Figure 3.6 shows the first section for the source code of the ***MEMORY1*** process. At the rising edge of the clock, if signal ***START*** is '1' then we initialize some internal variables as seen from the code. We then have a case statement for the internal variable ***BUSY*** which decides the section of the test bench to be executed. If the condition for (*BUSY=1*) is satisfied then the data from the input image file ***IMAGEIN*** is read into a frame buffer named ***INPUT_IMAGE***. A sample input image data file is shown in Figure 3.7. Another two frame buffers ***OUTPUT_MAG_IMAGE*** and ***OUTPUT_DIR_IMAGE*** are also created which store the magnitude and direction outputs respectively. The frame buffers ***OUTPUT_MAG_IMAGE*** and ***OUTPUT_DIR_IMAGE*** are initialized to ***0*** and ***"000"*** respectively. As soon as the input image file is completely read into the frame buffer, *BUSY* is set to 2. In order to check if the entire image file has been read, we have an assertion statement which generates a report ensuring that the image file has been read into the frame buffer successfully. When *BUSY=2* ,the image data from the frame buffer *INPUT_IMAGE* is transferred to the edge detector component via a signal ***INPUT***. Also the signal ***EDGE_START*** is set to '1' for one clock cycle at the beginning of the data transfer to the MUT.

```
 MEMORY1: process
----------------------Declaration of internal variables----------------------
variable VLINE1,VLINE2:LINE; -- for reading and writing from the INPUT and OUTPUT text files
variable BUSY:INTEGER range 0 to 3;     -- internal variable to trigger the different parts of the test bench
variable INPUT_IMAGE : FRAME_IMAGE;             -- frame buffer to store INPUT image data
variable OUTPUT_MAG_IMAGE:FRAME_IMAGE;-- frame buffer to store magnitude outputs
variable OUTPUT_DIR_IMAGE:FRAME_DIRECTION;-- frame buffer to store the Direction outputs
variable Z:INTEGER; -- used for reading the input files
variable I,J:NATURAL:=1;-- used for indexing the frame buffers
variable X,Y:NATURAL:=2; -- used for indexing the frame buffers
variable COUNT:INTEGER:=0;-- used to specify when the outputs can be -- written into the buffers
-------------------------------------------------------------------------------
--FILES--
file IMAGEIN      :TEXT is in IN_FILE;       --INPUT image file
file IMAGEOUT  :TEXT is out OUT_FILE;  --OUTPUT file for storing magnitude Outputs
file DIROUT       :TEXT is out DIR_FILE;  --OUTPUT file which stores Direction Outputs

begin
 wait until rising_edge(CLOCK);
-- set the internal BUSY signal
        if START = '1' then
                BUSY :=1;
                COUNT:=0;
                I:=1;
                J:=1;
                X:=2;
                Y:=2;
        end if;

case BUSY is
```

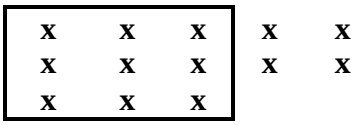
```
when 1 =>
load the image data file to the internal frame buffer----------
        for i in 1 to NUM_ROWS loop
                readline(IMAGEIN,VLINE1);
               for j in 1 to NUM_COLS loop
                        read(VLINE1,Z);
                        INPUT_IMAGE(i,j):= Z;
                       OUTPUT_MAG_IMAGE(i,j):=0;
                        OUTPUT_DIR_IMAGE(i,j):="000";
                end loop;
        end loop;
BUSY:=2;
-------- assert message after loading all the data----------
assert (false) report "array read in";
when 2 =>

COUNT:=COUNT+1;
INPUT <= (INPUT_IMAGE(i,j));
if (I=1) and (J=1) then
EDGE_START<='1';
        else
EDGE_START<='0';
end if;
if (I=NUM_ROWS) and (J=NUM_COLS) then
        I:=1;
        J:=1;
elsif J=NUM_COLS then
        J:=1;
        I:=I+1;
else
        J:=J+1;
end if;
```

Figure 3.6 Section of the code of the Memory Process where the input file is read

```
100 100 100 100 100 0 0 0 0 0
100 100 100 100 100 0 0 0 0 0
100 100 100 100 100 0 0 0 0 0
100 100 100 100 100 0 0 0 0 0
100 100 100 100 100 0 0 0 0 0
100 100 100 100 100 0 0 0 0 0
100 100 100 100 100 0 0 0 0 0
100 100 100 100 100 0 0 0 0 0
100 100 100 100 100 0 0 0 0 0
100 100 100 100 100 0 0 0 0 0
```

Figure 3.7 Format of input image file

Figure 3.9 shows the section of the code where the outputs of the MUT are written into text files. We have an internal variable ***COUNT*** which determines when the output data should be written into the output frame buffers. The variable ***COUNT*** is used here to ensure that only valid data is written into the output frame buffers. As soon as the condition specified for count is satisfied, the magnitude as well as the direction outputs of the Edge detector are written into output frame buffers ***OUTPUT_MAG_IMAGE*** and ***OUTPUT_DIR_IMAGE*** respectively.



X - REPRESENTS INPUT IMAGE DATA

Figure 3.8 Portion of the image data required by the MUT to start the edge detection process.

As seen from Figure 3.8, the MUT can produce valid output data only after (2*NUM_COLS + 3) data have been transferred to it from the test bench. In order to avoid invalid output data being produced before this condition is satisfied, we have defined the condition of (*COUNT >=2*NUMCOLS + WAIT_CYCLES*) in the source code. Apart from this fact, since we also needed a few additional clock cycles to allow for delays in the MUT, we have specified WAIT_CYCLES as 5 in the configuration declaration in Figure 3.12 for the MUT at the behavioral level.

Once the data is written into the frame buffers, an internal variable ***BUSY*** is set to 3. Now the magnitude and direction outputs are written into output text files ***IMAGEOUT*** and ***DIROUT*** respectively. Assertion statements are used to indicate that the outputs have been written successfully to the text files. After the outputs have been written, ***COMP_START*** is set high which triggers the two comparators.

```
--This code is a continuation of the "when 2=>" from Figure 3.6--
------------ write the magnitude and direction outputs into internal frame buffers ----------------------------------
        if COUNT>=(2*NUM_COLS+WAIT_CYCLES) then          --ready to write valid data--
                if not(Y=1) and not(Y=NUM_COLS)  then
                        OUTPUT_MAG_IMAGE(X,Y):= OUTPUT;
                        OUTPUT_DIR_IMAGE(X,Y) := STDLOGIC_TO_BIT(DIR);
                end if;
                Y:=Y+1;
                if Y=NUM_COLS+1 then                                    --start new row--
                        Y:=1;
                        X:=X+1;
                end if;
                if (X=NUM_ROWS-1) and Y=NUM_COLS then       --entire image has been processed--
                        BUSY:=3;
                end if;
```

```
        end if;
--Write the outputs to External text files--
when 3 =>
        RD_START<='0';
        for i in 1 to NUM_ROWS loop
                for j in 1 to NUM_COLS loop
                        write(VLINE1,OUTPUT_MAG_IMAGE(i,j));
                write(VLINE1,' ');
                        write(VLINE2,OUTPUT_DIR_image(i,j));
                        write(VLINE2,' ');
                end loop;
                writeline(IMAGEOUT,VLINE1);
                writeline(DIROUT,VLINE2);
        end loop;
-------- assert message after loading all the data----------------------------
      assert (false) report "magnitude and direction outputs written to file";
      COMP_START<='1';                          --Trigger the Comparators
      BUSY:=0;
when others => null;
end case;
end process MEMORY1;
end BENCH;
```

Figure 3.9 Section of the code for the memory process where the magnitude and direction outputs are written into files.

3.2.2.5 Comparators

Figure 3.10 shows the block diagram of the Comparator component used to compare the outputs of the MUT with the expected response and generate a PASS/FAIL indication. The model has two inputs ***CLOCK*** and ***START*** . The START signal is used to trigger the comparator.

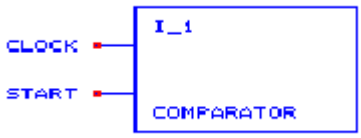


Figure 3.10 Block diagram of the Comparator component.

Figure 3.11 shows a portion of the source code for the Comparator component used to compare the magnitude outputs. When an internal variable *BUSY* is set high, the data from the text files of the GOLD model as well as the MUT, ***GOLD_MAGNITUDE*** and ***TEST_MAGNITUDE*** are read into two frame buffers named *GOLD_MAG_IMAGE* and *TEST_MAG_IMAGE*.

```
 wait until rising_edge(CLOCK);
--Set the internal BUSY signal high to start comparison--
if START ='1' then
        if END_COMP_MAG ='0' then
        BUSY:='1';
        end if;
end if;

 if BUSY='1' then
---Read the files and compare the contents----------------------------------------

        for i in 1 to NUM_ROWS loop
                readline(TEST_MAGNITUDE,VLINE1);
                readline(GOLD_MAGNITUDE,VLINE2);
                 for j in 1 to NUM_COLS loop
                 read(VLINE1,A);
                        read(VLINE2,B);
                        TEST_MAG_IMAGE(i,j):=A;
                        GOLD_MAG_IMAGE(i,j):=B;
                        if FLAG1='0' then
                                 if TEST_MAG_IMAGE(i,j) /= GOLD_MAG_IMAGE(i,j) then
                                --set flag high--
                                FLAG1:='1';
                                end if;
                end if;
                end loop;
        end loop;

if FLAG1='1' then
-- assert message after comparing all the data
         assert (false) report "MAGNITUDE VALUES DO NOT MATCH -- FAIL";
else
        assert (false) report "MAGNITUDE VALUES MATCH -- PASS";
end if;
END_COMP_MAG<='1';
end if;
BUSY:='0';
end process;
end COMPARE;
```

Figure 3.11 Portion of the source code for the Comparator component.

We then compare the magnitude outputs of the MUT with that of the GOLD model one pixel at a time and if there is an error, we set an internal variable ***FLAG1*** to '1'. If *FLAG1* is set high, we generate a report indicating that the outputs of the MUT don't match the outputs of the GOLD model, or else the comparator provides a PASS indication. The Comparator for the direction outputs was developed along similar lines.

3.2.2.6 Configuration declarations

VHDL allows a great deal of flexibility in configuring a specific model for simulation. Configurations allow selection of different architecture bodies for components and also allow generic constants to be specified in them rather than in the component instance and this makes the simulation repeatable. Another use of configurations is to define port maps, particularly to specify type conversion functions. The combination of the selection of architecture bodies and the choice of type conversion functions is a key part of the construction of a mixed-level-of-abstraction model using components from a design database in which components are modeled at multiple levels. Effective use of configuration declarations can provide significant assistance in the configuration management of models. However, care must be taken to centralize the late binding decisions in the configuration declaration rather than distribute this information throughout the different compilation units. For example, if the same test bench can be used with different external data files, the names of the data files should be defined in the configuration declaration rather than in the architecture body of the design entity that reads the file. An example configuration file is shown in Figure 3.12.

```
--declaration of an empty top level component--
 entity TB_CONFIG is
end TB_CONFIG;

architecture TEST_BENCH of TB_CONFIG is
component TEST1
end component;

begin
con: TEST1;
end TEST_BENCH;
------------------------------CONFIGURATION DECLARATION--------------------------
 library IEEE;
use ieee.STD_LOGIC_1164.all;

library BEH_INT;
use BEH_INT.IMAGE_PROCESSING.all;
use BEH_INT.all;
use STD.TEXTIO.all;

configuration CONFIG_B_INT of TB_CONFIG is
for TEST_BENCH

for con:TEST1 use entity WORK.TEST(BENCH)
generic map("tesv_i2.dat","test_o1.dat","test_d1.dat",10,10,5);

        for BENCH

         for P1:CLOCK_GENERATOR1 use entity BEH_INT.CLOCK_GENERATOR(BEHAVIOR)
        generic map(HI_TIME=>75 ns,LO_TIME=>25 ns);
         end for;
```

```
        for P2:COMP_MAG1 use entity WORK.COMP_MAG(COMPARE)
        generic map("test_o1.dat","test_gm.dat",10,10);
        end for;

        for P3:COMP_DIR1 use entity WORK.COMP_DIR(COMPARE)
        generic map("test_d1.dat","test_gd.dat",10,10);
        end for;

        for P4:EDGE_DETECTOR1 use entity BEH_INT.EDGE_DETECTOR(BEHAVIOR)
        generic map(NUM_ROWS=>10,NUM_COLS=>10);
        end for;
        end for;
 end for;
end for;
end;
```

Figure 3.12 Configuration declaration for Sobel Edge detector

The configuration declaration in Figure 3.12 configures the test bench for testing the behavioral architecture of the Sobel Edge detector. In order to declare the input and output text files as generics in the configuration file, we have declared an entity *TB_CONFIG* with an architecture containing one component, TEST1. The configuration declaration CONFIG_B_INT then associates the ***CON*** instance of *TB_CONFIG* with the test bench entity (WORK.TEST (BENCH)), instance ***P1*** with the Clock generator(BEH_INT.CLOCK_GENERATOR (BEHAVIOR)), instance ***P2*** with the component COMP_MAG (WORK.COMP_MAG(COMPARE)) for comparing the magnitude outputs, instance ***P3*** with COMP_DIR(WORK.COMP_DIR(COMPARE)) for comparing the direction outputs and instance ***P4*** with that of the Sobel Edge Detector (BEH_INT.EDGE_DETECTOR(BEHAVIOR)). When we need to test the Edge detector model at lower levels of abstraction then we just need to change the name of the architecture and library in the configuration without having to change anything else in the model. This is explained in detail in section 3.3.3. Generic constants have also been declared in the configuration. For the top level entity ***TEST*** the various input and output file names have been specified. Apart from the file names ***NUM_ROWS*** and ***NUM_COLS*** have been specified as 10 and *WAIT_CYCLES* has been specified as 5 since the model being simulated is at the behavioral level. For the Clock Generator ***HI_TIME*** and ***LO_TIME*** have been specified as 75 ns and 25 ns respectively.

3.2.3 Example Test results

The functionality of the edge detector model was verified effectively by developing a test plan as shown in Table 3.1. The test plan clearly lists the various test goals, stimuli source, gold data source, acceptable outcome and the desired coverage value for each test.

To illustrate the use of the test plan, we will show how to obtain test results for test numbers 1.1A and 1.1B. These two test goals ensure that the MUT detects horizontal and vertical edges and that appropriate direction values are assigned to each edge. In the test plan table, the

Table 3.1 Test Plan

Test Number	Test Goal	MUT Configuration/ Level of Abstraction	Stimuli Source	Gold Data Source	Acceptable Outcome	Coverage Measure	Expected Coverage Value
1.1.A	1.1	Executable Specification	tesv_i1.dat	test_g1.dat	All H edges detected with correct direction	Statement	100%
1.1 B	1.2	Executable Specification	tesv_i1.dat	test_g1.dat	All V edges detected with correct direction	Statement	100%
1.1 C	1.3	Executable Specification	tesv_i2.dat	test_g2.dat	All L edges detected with correct direction	Statement	100%
1.1 D	1.4	Executable Specification	tesv_i2.dat	test_g2.dat	All R edges detected with correct direction	Statement	100%
1.1 E	1.5	Executable Specification	tesv_i1.dat	test_g1.dat	All corners detected with correct direction	Statement	100%
1.1 F	1.6	Executable Specification	tesv_i3.dat	test_g3.dat	Single pixel holes should be filled	Statement	100%
1.1 G	1.7	Executable Specification	tesv_i4.dat	test_g4.dat	Multi-pixel holes remain unfilled	Statement	100%
1.1 H	1.8	Executable Specification	tesv_i5.dat	test_g5.dat	Single pixel corner holes filled	Statement	100%
1.1 I	1.9	Executable Specification	tesv_i6.dat	test_g6.dat	Multi pixel corner holes remain unfilled	Statement	100%
1.1 J	1.10	Executable Specification	tesv_i7.dat	test_g7.dat	Single pixel holes ignored	Statement	100%
1.1 K	1.11	Executable Specification	tesv_i8.dat	test_g8.dat	Noise filtered appropriately and ignored	Statement	100%

The Test Goal Hierarchy

Functional Verification Tests

1.1 A. Horizontal Edges Detected and Given Appropriate Direction
1.1 B. Vertical Edges Detected and Given Appropriate Direction
1.1 C. Left Diagonal Edges Detected and Given Appropriate Direction
1.1 D. Right Diagonal Edges Detected and Given Appropriate Direction
1.1 E. Corners Detected and Given Appropriate Direction
1.1 F. Single pixel “holes” in edges are filled
1.1 G. Multi-pixel “holes” are unfilled
1.1 H. Single pixel corner “holes” are filled
1.1 I. Multi-pixel corner “holes” are unfilled
1.1 J. Single pixel spots are ignored
1.1 K. Appropriate handling of “rough” edges (effects of aliasing on edge detection for varying edge angles)

first column, test number corresponds to the test goals defined below the table. The input test image shown in Figure 3.13(a) is a synthetically generated image that contains horizontal and vertical edges. This image is placed in file "tesv_i1.dat" as indicated in column ***"Stimuli source".*** The expected response for the input image is shown in Figure 3.13(b). This image is placed in file "tesv_g1.dat" as indicated in column "***Gold data source***". The output response of the MUT after simulation is compared with this image(tesv_g1.dat) to verify its correctness. The edge detected output image produced by simulation is shown in Figure 3.13(c). Both the horizontal and vertical edges for the image have been detected correctly.

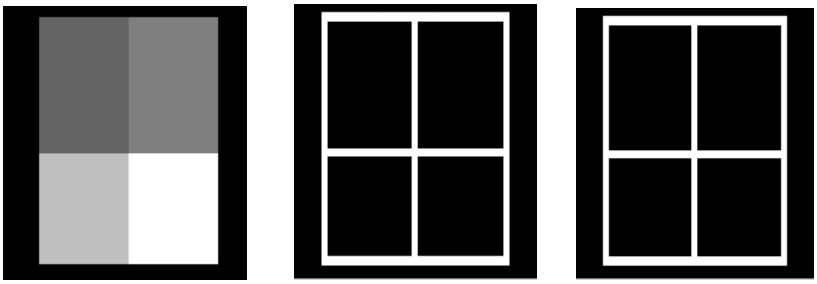


Figure 3.13 a)Input image , b)Expected response, c) Edge detected output image.

The direction values for the horizontal and the vertical edges that were output by the simulation are as follows :

	HORIZONTAL EDGES	VERTICAL EDGES
1	"110" - TOP EDGE	"000" - LEFT EDGE
2	"110" - MIDDLE EDGE	"000" - MIDDLE EDGE
3	"010" - BOTTOM EDGE	"100" - RIGHT EDGE

These directions were computed according to Figure 2.4. The results are in line with the acceptable outcome listed in the table. The coverage of the MUT was verified by the Synopsys simulator which indicated that 100% coverage was obtained. This value was in agreement with the goal specified in the column ***"Coverage value"*** of the Test plan table.

After testing the MUT completely for all its functions with synthetically generated images, it was finally tested with a real world image. The input image along with its edge detected output are shown in Figure 3.14(a) and 3.14(b) respectively.

Figure 3.14(a) Input image of M1A1 tank

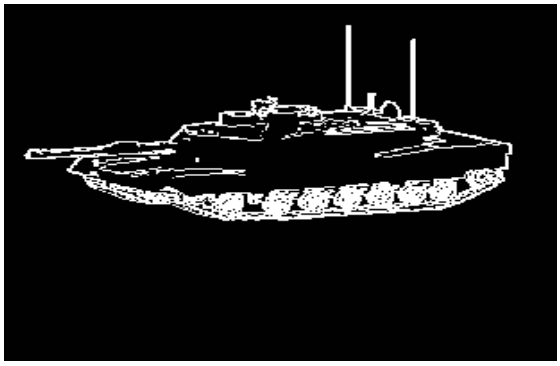


Figure 3.14 (b) Edge detected output image of the M1A1 tank.

3.3 Test bench development for a structural model

3.3.1 Methodology

This section explains the methodology adopted for testing a model at the structural level of abstraction.

The test bench model for the behavioral model can be reused to test the MUT at the structural level. This would simply involve replacing the architecture and the library name in the configuration declaration. We could then simulate the test bench and compare the results of the MUT with the output response of the MUT at the behavioral level. The procedure adopted for testing is detailed below :

1. First we test each sub-component in the MUT separately with its own test bench. Hence it is necessary that test benches be developed for each sub-module that forms a part of the entire system.

2. After testing all the lower level modules, these are integrated by a higher level structural design entity and the resulting model is tested with its test bench. In this way, we work our way up the hierarchy of structural models.

3. All the models are finally integrated together into the top-most level and this model is tested with the test bench developed for the behavioral model.

4. The outputs of the MUT are then compared with the output response of the executable specification in a comparator which verifies the response.

This hierarchy of test benches provides a mechanism for the bottom-up validation of the model and facilitates debugging.

3.3.2 Testing Procedure

In order to illustrate the methodology in Section 3.3.1 let us consider Figure 3.15 which shows the hierarchy of a structural model for the Sobel edge detector. This model is a mixed abstraction model since it does not have the same level of detail at each leaf node. As seen from the figure, the edge detector consists of three interconnected components : a memory system (MEM_PROCESSOR), a window processor (WIN_PROCESSOR) and a direction and magnitude processor (MAG_PROCESSOR). The memory system consists of two components : an Address Generator (ADDR_GEN) and the Memory (MEMORY) component. It stores the input image pixel values and sends out three pixels each clock cycle to the window processor. The window processor as seen from Figure 3.15 consists of four interconnected components: horizontal filter (HOR_FILTER), vertical Filter (VERT_FILTER), left diagonal filter (LEFT_FILTER) and right diagonal filter (RIGHT_FILTER).

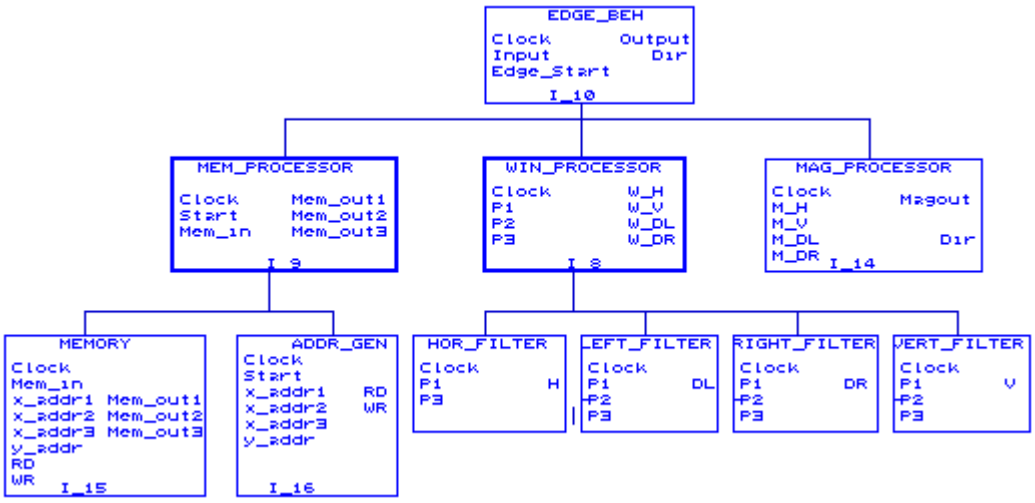


Figure 3.15 Hierarchy of a Structural model of the Sobel edge detector

The four components compute the horizontal, vertical, and left and right diagonal filter outputs respectively. The four filter outputs are passed to the direction and magnitude processor which computes the direction and magnitude output.

The testing procedure adopted for testing the edge detector is based on the methodology illustrated in Section 3.3.1. Once the behavioral models for all the sub-components were developed, each of the components were tested individually to verify their functionality and then these modules were integrated to form the upper level module which was then tested with its own test bench. For the case of the Window processor, we first tested the four filters individually and then tested the Window processor as a whole. The source code for all the test benches can be found in Appendix A. Apart from checking for functionality of all the sub-components, extensive testing was done to check for faults in the memory system. The process of memory testing is explained in detail in section 4.2 of this thesis.

3.3.3 Configuration Example

Let us consider the source code shown in Figure 3.16 which is for a structural model of the window processor. As already mentioned, the window processor has four components: a horizontal filter, a vertical filter, a left diagonal filter and a right diagonal filter. The structural model shown in Figure 3.16 is just a skeleton that only includes component and instance declarations. The port maps and generic maps for the structural model have been deferred to the configuration declaration which is shown later in Figure 3.18.

```
 library IEEE;
use IEEE.STD_LOGIC_1164.all;
library BEH_INT;
use BEH_INT.IMAGE_PROCESSING.all;
library STRUC_INT;
use STRUC_INT.all;
---------- interface declaration ----------------------------------------
entity WINDOW_PROCESSOR is
          generic(HORIZ_DELAY,VERT_DELAY,LEFT_DIAG_DELAY,RIGHT_DIAG_DELAY,
              WAIT_TIME: TIME);
           port(     CLOCK: in std_logic:='0';            -- system CLOCK
                     P1,P2,P3: in  PIXEL:=0;              -- input Pixels
                     W_H: inout FILTER_OUT:=0;       -- Horizontal filter output
                      W_V: inout FILTER_OUT:=0;       -- vertical filter output
                     W_DL: inout FILTER_OUT:=0;      -- left diagonal filter output
                     W_DR: inout FILTER_OUT:=0 );    -- right diagonal filter output
end WINDOW_PROCESSOR;
 ---------- structural description ----------------------------------------
architecture STRUCTURE of WINDOW_PROCESSOR is
---------- empty horizontal filter component ------------------------------
component HORIZONTAL_FILTER1
end component;
---------- empty vertical filter component --------------------------------
 component VERTICAL_FILTER1
 end component;
---------- empty left diagonal filter component ---------------------------
 component LEFT_DIAG_FILTER1
  end component;
---------- empty right diagonal filter component --------------------------
component RIGHT_DIAG_FILTER1
end component;
begin
------------- empty components instantiation ----------------------------
HP: HORIZONTAL_FILTER1;
VP: VERTICAL_FILTER1;
LDP: LEFT_DIAG_FILTER1;
RDP: RIGHT_DIAG_FILTER1;
end STRUCTURE;
```

Figure 3.16 Structural model for Window processor

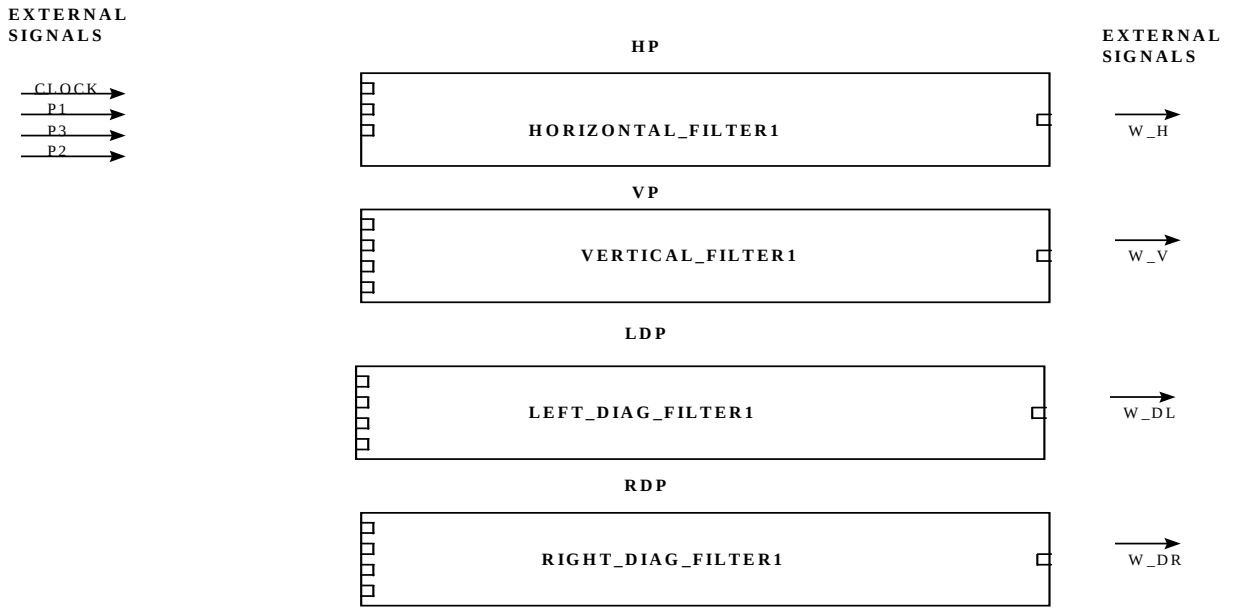


Figure 3.17 Diagram of the Window Processor with empty sockets.

A block diagram for the model in Figure 3.16 is shown in Figure 3.17. The block diagram includes external signals Clock, P1, P2, P3, W_H, W_V, W_DL and W_DR which correspond to the port names of the window processor entity shown in Figure 3.16. Empty (floating) sockets represent the four components: horizontal, vertical, left diagonal and right diagonal filter. The component names appear inside the empty sockets. The instance names appear outside the sockets. As seen from Figure 3.17, there are no connections between the external signals and the sockets. The empty sockets represent the unbound instance of a component. All the connections between the external signals and the internal signals of the components in the sockets are deferred for the configuration.

```
--CONFIGURATION SPECIFICATION FOR WINDOW PROCESSOR
for WINP: WINDOW_PROCESSOR1 use entity  STRUC_INT.WINDOW_PROCESSOR(STRUCTURE)
 generic map(HORIZ_DELAY=> 3 ns, VERT_DELAY=> 3 ns,LEFT_DIAG_DELAY => 3 ns,
             RIGHT_DIAG_DELAY=> 3 ns, WAIT_TIME =>100 ns)
 port map(CLOCK,MEM_OUT1,MEM_OUT2,MEM_OUT3,E_H,E_V,E_DL,E_DR);

for STRUCTURE

for HP:HORIZONTAL_FILTER1 use entity STRUC_STD.HORIZONTAL_FILTER(BEHAVIOR)
generic map(HORIZ_DELAY => HORIZ_DELAY,WAIT_TIME=>WAIT_TIME)
port map (CLOCK=> CLOCK,P1_H=>INT_TO_STDLOGIC8(P1), P3_H=>INT_TO_STDLOGIC8(P3),
          STDLOGIC_TO_INT(H)=>INT_TO_STDLOGIC12(W_H));
end for;
```

```
for VP:VERTICAL_FILTER1 use entity STRUC_INT.VERTICAL_FILTER(BEHAVIOR)
generic map(VERT_DELAY => VERT_DELAY,WAIT_TIME=>WAIT_TIME)
port map (CLOCK=> CLOCK,P1_V=>P1, P2_V=>P2,P3_V=>P3, V=>W_V);
end for;

for LDP: LEFT_DIAG_FILTER1 use entity STRUC_INT.LEFT_DIAG_FILTER(BEHAVIOR)
generic map(LEFT_DIAG_DELAY => LEFT_DIAG_DELAY, WAIT_TIME => WAIT_TIME)
port map (CLOCK=>CLOCK,P1_L=>P1,P2_L=>P2,P3_L=>P3,DL=>W_DL);
end for;

for RDP: RIGHT_DIAG_FILTER1 use entity STRUC_INT.RIGHT_DIAG_FILTER(BEHAVIOR)
generic map(RIGHT_DIAG_DELAY => RIGHT_DIAG_DELAY, WAIT_TIME => WAIT_TIME)
port map (CLOCK=>CLOCK,P1_R=>P1,P2_R=>P2,P3_R=>P3,DR=>W_DR);
end for;

end for;
end for;
```

Figure 3.18 Portion of the source code for the configuration for structural model of the MUT.

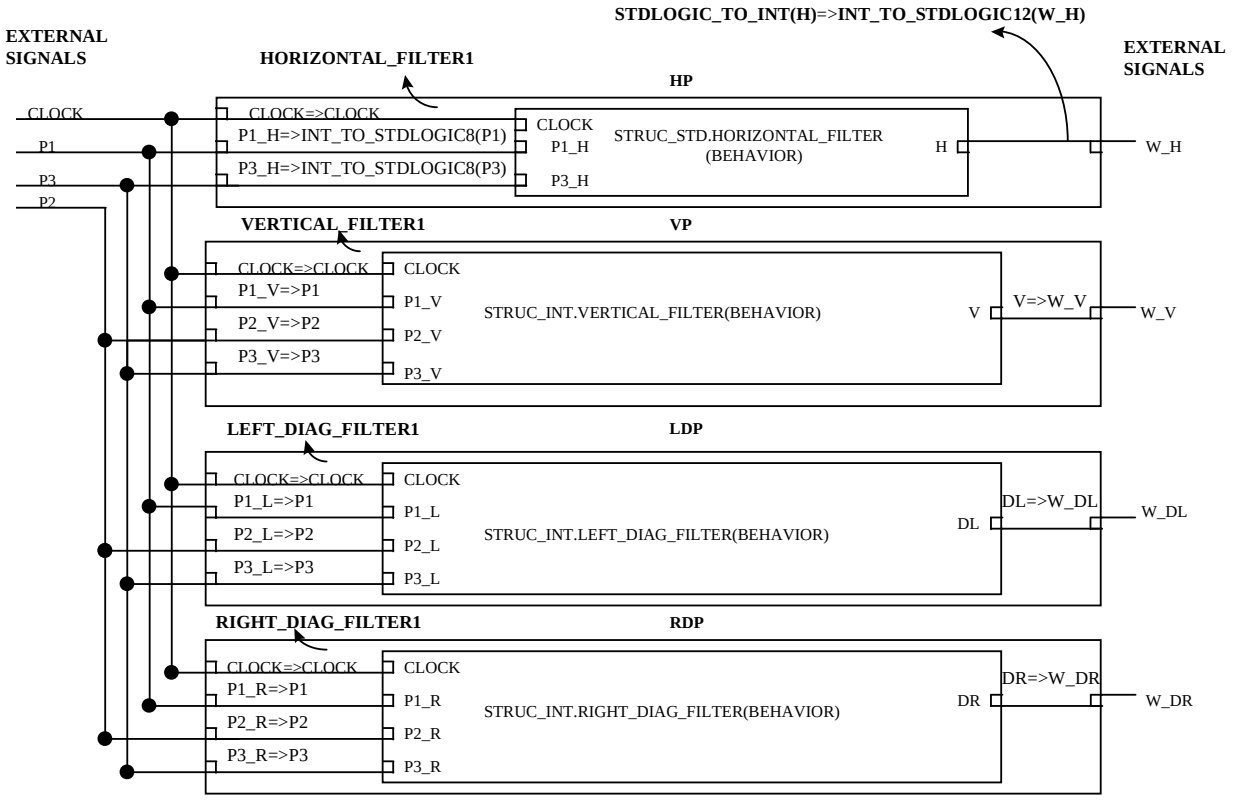


Figure 3.19 Diagram for Window Processor with the chips inserted in the sockets.

Let us consider Figure 3.18. It shows the source code of the configuration declaration for the structural model shown in Figure 3.16. The effect of the configuration is shown in Figure 3.19. Note that the configuration specifies interconnections to the component instances and specifies the names of library “chips” that are to be inserted into the floating sockets. As seen from Figure 3.18, the generic maps and port maps for the sub-components of the window processor have been moved to the configuration declaration. This is done mainly to facilitate the implementation of mixed data type models.

In the configuration example shown in Figure 3.18, the horizontal filter is of std_logic data type whereas the rest of the components of the window processor are of integer data type. Hence, the inputs to the horizontal filter have to be converted to std_logic data type before they can be connected to the input ports of the component. This is shown in the block diagram in Figure 3.19 as well as in the source code in Figure 3.18 by the following statements :

```
P1_H => INT_TO_STDLOGIC8(P1);
P3_H=> INT_TO_STDLOGIC8(P3);
```

Here the external signals P1 and P3 are first converted from integer to std_logic data type and then connected to the input ports of the Horizontal filter P1_H and P3_H respectively. Similarly the output H of the Horizontal filter is converted back to integer data type from std_logic data type by the following statement :

```
STDLOGIC_TO_INT(H)=>INT_TO_STDLOGIC12(W_H);
```

For the other sub-components of the window processor , the external signals are simply connected to the input ports of each component. This can be seen clearly from both the block diagram in Figure 3.19 and the source code in Figure 3.18. In this manner we can easily implement mixed data type models.

3.4 Test Bench development of Register Transfer Level model

The test bench for the behavioral model can be reused to test the function of the RTL model. In order to achieve this we had to replace only the architecture and library names for all of the filters in the Window Processor and the Magnitude and Direction processor in the configuration declaration from ***STRUC_INT*** to ***STRUC_RTL***. Also the generic constant ***WAIT_CYCLES*** in the *component* ***TEST*** in the configuration declaration must be altered to 13 since an additional five clock cycles are required for the register delays in comparison with the MUT at the structural level of abstraction. The testing methodology was similar to the one adopted for testing structural level models. Each of the sub-components were first tested with their individual test benches and then these were integrated into the upper level models which were tested with their own test bench model.

3.5 Test Bench development for gate level models

Having synthesized the MUT at the RTL level into a netlist, it is important to verify the functionality of the synthesized netlist to ensure it still matches the intended functionality. This step is important since a synthesized tool may make certain assumptions or interpretations of the VHDL code that may not match those intended by the modeler.

The RTL models which were synthesized were the four filters in the Window Processor and the Magnitude Processor.

3.5.1 Methodology

The methodology adopted for testing synthesized circuits is explained below.

The test bench model developed for the behavioral model of the MUT can be reused to test the model at the gate level. One additional feature is required, namely a comparison process which compares the output of the model at the behavioral level with that at the gate level and generates a PASS/FAIL indication must be added.

The procedure adopted for testing is detailed below :

1. The circuits (at the gate as well as the behavioral level) are first instantiated in the test bench along with the Clock Generator component.

2. A comparison process is then developed, which compares the outputs of the two models, one bit at a time and generates a PASS/ FAIL indication.

3. In order to specify the generic values a configuration needs to be developed for the test bench. Initially some arbitrary values are specified for all the generic constants of the executable specification of the model and the models are simulated. The delay time between the models at the behavioral and gate level can be extracted by observing the simulator outputs.

4. This extracted value can be then be back annotated into the generic constants of the executable specification, and the test vectors can be re-applied to compare the outputs of the two models.

3.5.2 Implementation

In order to illustrate the above methodology, we will use the horizontal filter circuit. Figure 3.20 shows a portion of the source code for the test bench used to test this model. The test bench developed compares the outputs of the behavioral and the gate level model of the

horizontal filter. We simulate the behavioral as well as the gate level models simultaneously and compare the resultant outputs to ensure that the outputs match.

The test bench developed compares the outputs of the models. The architecture body includes component declarations corresponding to the Clock generator (***CLOCK1***), Horizontal filter component at the behavioral level (***HORIZONTAL_FILTER1***) and the synthesized model of the Horizontal filter component (***HORIZONTAL_FILTER2***). The *COMPARE* process shown in Figure 3.20 is now explained in detail. In this process, the outputs of *HORIZONTAL_FILTER1* and *HORIZONTAL_FILTER2* are compared and if they are not equal, an internal variable *FLAG* is set to high else the *FLAG* is set to low. We then have two assertion statements which alert the user if the outputs of the two components match depending on the value of the flag variable.

```
--COMPONENT INSTANTIATIONS—
component CLOCK_GENERATOR1
        port (RUN : in STD_LOGIC;
         CLOCK: out STD_LOGIC);
end component;
component HORIZONTAL_FILTER1
        port (CLOCK: in STD_LOGIC:='0';
            P1,P3: in STD_LOGIC_VECTOR(7 downto 0):="00000000";
            H: out STD_LOGIC_VECTOR(11 downto 0):="000000000000");
end component;

component horizontal_filter2
        port (CLOCK: in std_logic:='0';
            P1,P3: in STD_LOGIC_VECTOR(7 downto 0):="00000000";
            H: out STD_LOGIC_VECTOR(11 downto 0):="000000000000");
end component;

Begin

L1: CLOCK_GENERATOR1
PORT MAP(RUN,CLOCK);

L2: HORIZONTAL_FILTER1
PORT MAP(CLOCK,P1,P3,H);

L3: HORIZONTAL_FILTER2
PORT MAP(CLOCK,P1,P3,H1);

--PROCESSES--
COMPARE : process
variable FLAG : STD_LOGIC := '0';
begin
wait until rising_edge(CLOCK);
if H / = H1 then
  FLAG :='1';
  else
 FLAG := '0';
```

```
 end if;
 if FLAG ='1' then
-- assert message after comparing all the data
          assert (not FLAG = '1') report "OUTPUTS DO NOT MATCH -- FAIL"
          severity WARNING ;
else
          assert (not (not FLAG = '1') )report "OUTPUTS MATCH -- PASS"
          severity WARNING ;
end if;
end process;
end COMPARE;
```

Figure 3.20 Portion of the source code of the test bench model to compare the outputs of the Horizontal filter at the behavioral and gate level.

The configuration for testing the horizontal filter components is shown in figure 3.21.

```
configuration CONFIG_HOR of TB is
for compare

        for L1:CLOCK_GENERATOR1 use entity BEH_STD.CLOCK_GENERATOR(BEHAVIOR)
        generic map(HI_TIME=>75 ns,LO_TIME=>25 ns);
        end for;

        for L2:HORIZONTAL_FILTER1 use entity BEH_STD.HORIZONTAL_FILTER(BEHAVIOR)
        generic map(HORIZ_DELAY=>1 NS,WAIT_TIME=>100 NS);
        end for;

        for L3: HORIZONTAL_FILTER2 use entity STRUC_GATE.HORIZONTAL_FILTER(SYN_SYN);
        end for;
end for;
end ;
```

Figure 3.21 Configuration for the Horizontal filter

The configuration shown in Figure 3.21 is similar to the source code of Figure 3.12. In the configuration specification statements, we choose the component instances of Clock generator, behavioral model of the Horizontal filter and the synthesized model of the Horizontal filter from the *BEH_INT*, *BEH_STD* and *STRUC_GATE* libraries respectively. Initially we define the generic value (***WAIT_TIME***) for the executable specification of the horizontal filter as 0 ns and apply the test bench in Figure 3.20.The simulation output and the report generated for this test are shown in Figures 3.22 and 3.23. From Figure 3.22 we can see that signal ***H1*** which is the output of the synthesized model is delayed by 100 ns compared to the signal ***H***, the output of the behavioral model. Hence we now have the exact time delay associated with the gate level model. We can back-annotate this value of 100 ns into the behavioral model by specifying *WAIT_TIME* as 100 ns as shown in Figure 3.21. The resultant simulation output and the report is shown in Figures 3.24 and 3.25. As seen from Figure 3.24, the two outputs H and H1 match perfectly. In the reports

shown in Figures 3.23 and 3.25 we have shown the results only after 605 ns, because this is the earliest time for which valid data is generated from the horizontal filters.

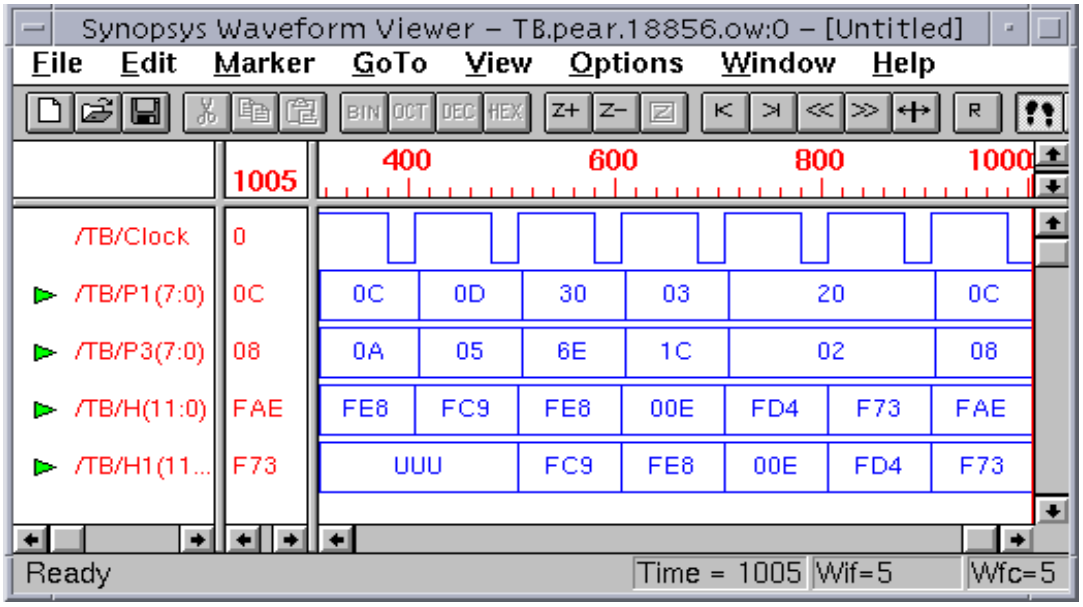


Figure 3.22 Simulation Output with generic constant *wait_time* = 0 ns for the behavioral model of the Horizontal filter

Assertion WARNING at 605 ns in design unit TB(COMPARE) from process /TB/COMPARE:
"OUTPUTS DO NOT MATCH – FAIL"
Assertion WARNING at 705 ns in design unit TB(COMPARE) from process /TB/COMPARE:
"OUTPUTS DO NOT MATCH – FAIL"
Assertion WARNING at 805 ns in design unit TB(COMPARE) from process /TB/COMPARE:
"OUTPUTS DO NOT MATCH – FAIL"
Assertion WARNING at 905 ns in design unit TB(COMPARE) from process /TB/COMPARE:
"OUTPUTS DO NOT MATCH – FAIL"

Figure 3.23 Report generated for the Simulation shown in figure 3.22.

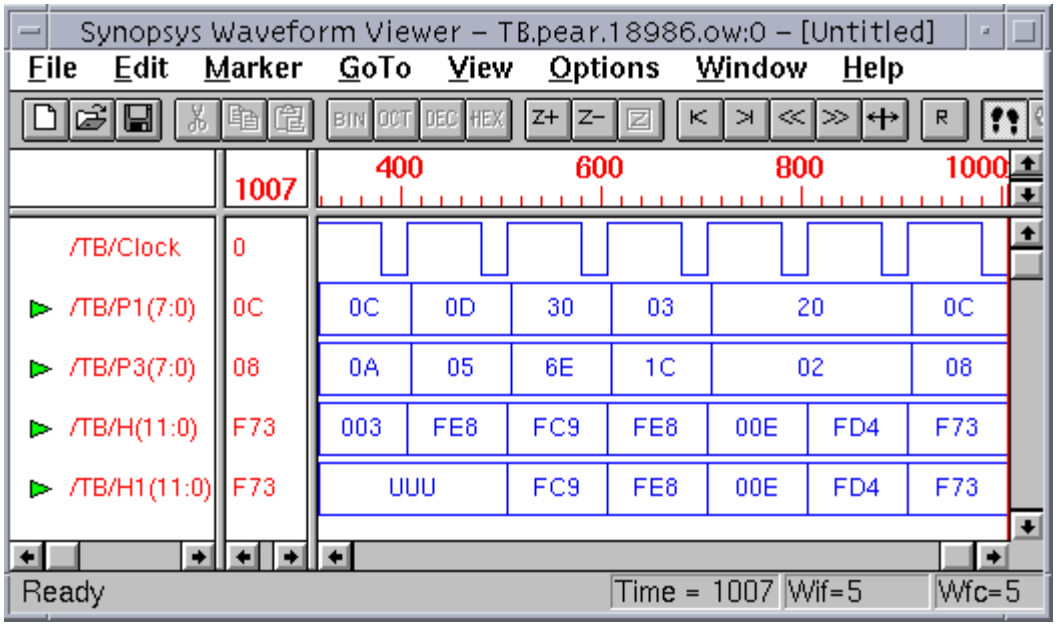


Figure 3.24 Simulation Output with generic constant *wait_time* = 100 ns for the behavioral model of the Horizontal filter

Assertion WARNING at 605 ns in design unit TB(COMPARE) from process /TB/COMPARE:
"OUTPUTS MATCH – PASS"
Assertion WARNING at 705 ns in design unit TB(COMPARE) from process /TB/COMPARE:
"OUTPUTS MATCH – PASS"
Assertion WARNING at 805 ns in design unit TB(COMPARE) from process /TB/COMPARE:
"OUTPUTS MATCH – PASS"
Assertion WARNING at 905 ns in design unit TB(COMPARE) from process /TB/COMPARE:
"OUTPUTS MATCH – PASS"

Figure 3.25 Report generated for the Simulation shown in figure 3.24.