Chapter 5. VHDL IMPLEMENTATION OF TESTABLE PROCESSOR

5.1. VHSIC Hardware Description Language (VHDL)

VHDL is a procedural language in the imperative style of Pascal and C. A script is a set of process definition, each of which contains procedural codes. The concept is processes represent hardware modules that execute independently, synchronously and communicate through signal wires. The language allows all hardware levels from digital systems to logic gates to be described independent of technology, methodology, and design tools. To support the wide range of descriptive capabilities required to model hardware from the logic gate to digital systems level, VHDL incorporates user defined data types, attributes, and assertions. In addition, the language supports both algorithmic and data flow oriented descriptions of hardware, and provides the user the definition of functions and packages. The collection of these facilities allows variety of descriptive styles and supports descriptions at many levels of abstraction [DOUG91].

The ability of the designer to define new data types and bus resolution functions means that VHDL can support many technologies while committing to none. VHDL is expected to reduce both the time lag and the cost involved in technology insertion. Because descriptions at different levels may be processed in an arbitrary order, both top-down and bottom-up design approaches are supported. The language is architectural, and supports the mixed level simulation.

Many other excellent languages cover subsets of the capabilities that exist within VHDL, but none is as comprehensive. They do not cover the wide range covered by VHDL. For example, SIM [REID91], which is a set of tools for designing combinational networks
and creating a netlist from equations or minterm specification, has powerful characteristics, but is limited in its scope. It does not treat multiple-output combination networks. It offers very little help in the design of sequential machines, and does not interface with VLSI layout and production mechanisms.

5.2- VHDL Support For BIST

The acceptance of the design for test techniques has been largely due to the possibility of VHDL support to this design style. It is desirable to eventually have available a built-in self-test approach with similarly powerful VHDL support. The standard test and maintenance interface provides a good, and a stable background for high leverage VHDL language development. The high degree of standardization makes it possible to have most testability features previously added to a design using VHDL.

Test specifications, however, cover more than test vectors. They describe the procedures, parameters, controls, signals, and interfaces involved in testing. This information must be expressed in a test description language or TDL. Test specifications written in a TDL can be input to a post processor which converts that information into a test program that controls automatic test equipment. A study has concluded that VHDL is suitable to be used as a TDL [ALLO86].

5.3- VHDL as a Standard Language

While there are many hardware description languages, there was, prior to VHDL, no accepted standard in the industry. Many of the existing languages, for example, Instruction Set Processor Specification (ISPS), A Hardware Programming Language (AHPL), or General
Hardware Description Language (GHDL) are developed to serve the simulators that run them. Whereas, others target a particular technology, design level, or design methodology, VHDL is technology independent, is notified to a particular simulator, and does not enforce a design methodology on a designer. VHDL thus offers a number of benefits over other hardware description languages [ROGER89], [ZAIN91].

5.4- Implementation Methodology And Rules

This section explains the experimental implementation steps and the rules that were followed to include BIST in a design. Figure 17 depicts the process flow of the implementation methodology.

5.4.1- Implementation Steps

1. Writing the behavioural VHDL description program for each portion of the partitioned design. The whole chip has been implemented using eight modules.

2. Compiling and debugging the VHDL codes. These were accomplished by using the VHDL compiler (VHDLAN) and VHDL debugger (VHDLDBX) in SYNOPSYS.

3. Fixing any errors or bugs to ensure the correctness of the VHDL codes.

4. Writing the testbench to allow the simulation process and establish the clock generator circuit. The testbench was used instead of the system control unit.
Figure 17. Implementation Process Flow.
5. Simulating the functionality of each portion of the entire chip using SYNOPSYS simulator (VHDLSIM).

6. Verifying the functionality of each unit of the chip. This includes fixing any malfunction of any part.

7. Starting test process simulation, the flow of the testing process is as shown in Figure 18. The testing process includes the following:

   - Loading the seed.
   - Generating the test patterns. This was started by setting the test-mode signal to HIGH.
   - Toggling clock cycles, which was generated by the Testbench.
   - Storing the patterns in a file for later comparison.
   - Compressing the output result as a signature.
   - Comparing the output signature with a "good" one using the Testbench.

8. Switching to the second test process and looping to step 7.

9. Analyzing and elaborating each unit for synthesis. This is to synthesize the top level of the behavioural design. The Design Analyzer in SYNOPSYS was used.

10. Optimizing the top level design to a specific technology.

11. Generating the test vectors and obtaining their fault coverage percentage using the automatic test pattern generator (ATPG) in SYNOPSYS.

12. Comparing the ATPG patterns with the generated BIST patterns to verify the effectiveness. This was accomplished using a C++ program written by the author.
Figure 18. Test Process Flow.

5.4.2- Implementation Tools

The implementation tools which were used are:

- A hierarchical hardware description language (VHDL) was used in the functional implementation stage.
SYNOPSYS Simulation Graphical Environment tools (SGE) were used from behavioural level through gate level implementation.

The verification tools used is a SYNOPSYS simulator and Debugger (Vhdlan, Gvan) and (Vhlddbx).

A SYNOPSYS Design Analyzer was used to synthesize the implemented design.

Implementation flow using these tools is illustrated in Figure 19 below.

```
+----------------+    +----------------+    +----------------+
|                |    |                |    |                |
|    S G E       |    |    VHDL        |    |    VHDLDBX     |
|                |    |    Source file |    |    debugger    |
|                |    +----------------+    +----------------|
|                |                     |    |                |
|    VHDLAN      |    +----------------+    |                |
|    or GVAN     |    |    VHDLDBX     |    |                |
|                |    |    debugger    |    |                |
|                |    +----------------+    |                |
|                |                      |    +----------------+    |
|                |                  +----------------+    |                |
|                |                          |    |                |
|                |                          |    |    VHDLSIM     |
|                |                          |    |                |
|                |                          |    |                |
|                |                          |    +----------------+    |
|                |                          |    |    wave form   |
|                |                          |    +----------------+    |
|                |                          |    |                |
|                |                          |    +----------------+    |
|                |                          |    |    trace files |
```

Figure 19. Implementation Flow using SYNOPSYS Tools.
5.4.3- VHDL Description

At the top level of the VHDL hardware specification hierarchy is the entity statement, which defines the input and output signals of the block. Associated with each entity is the architecture which represents the description of the design. The behavioural architecture description contains algorithmic statements such as condition statements, assignment statements and loops that functionally describe the behaviour of the design. These statements are included within a processor statement. The statements within a processor statement describe sequential behaviour, and the process statement represents behaviours that operates concurrently with the behaviour of other process statements in the architecture. Processes may assign new values to signals which could then activate other processes. All activated processes within an architecture execute concurrently.

5.4.4- Modeling Strategy

The design is modeled as a set of functional blocks, such as ALU, Q_write, register file, that communicate via signals and busses. The functional blocks are modeled as processes. In all the modules, the VHDL modeling conventions listed below were followed.

- High-level behaviours are modeled as processes or sets of communicating processes.

- All the registers and buffers are modeled as variables for better efficiency.

- Processes communicate with each other using global signals.

- Separate processes were used to model different concurrent behaviour.
5.4.5- Writing VHDL Codes

Writing VHDL codes starts with invoking the simulation Graphical Environment (SGE) which is used to capture the schematic models and assists with top-down VHDL description. SGE produces VHDL templates, structures and testbenches. VHDL templates are completed by adding the behavioural VHDL codes for each symbol. The VHDL description of the AMD2901 modules (Figure 11) are given in details in the following subsequent sections.

5.4.5.1- ALU

The function of the ALU process is to perform arithmetic logic operation on data inputs received from the data bus 'R' and 'S'. The output 'Fout' is modeled as a variable. The ALU is modeled as a process which drives the internal function variable 'Fout' according to the value of 'Iop'. This process is sensitive to changes on 'Iop' and any of the signals used in the right side of its assignment operator. After executing its last statement, it suspends operation until an event occurs on 'Iop', 'R', or 'Cn'. The flags are modeled as another process which can be executed concurrently with the ALU_value process. Flag processes drive the signals 'Feg0' and 'F3'. The other output flags are in GP_Terms process, which drive the signals 'P_n', 'G_n', 'Cnp4', and 'Ovr'. The carry generate and propagate terms are calculated in this process. The process is sensitive to the signals listed in its sensitivity list. It suspends itself after executing its last statement until a change occurs on one of these signals. The three processes (i.e., ALU_value, GP_Terms, and Flag) are operating concurrently.
5.4.5.2- ALU_MUXplexer

ALU_mux is a source selector for the ALU. It has eight modes of operation depending on the value of 'Isource'. In each mode, sources are selected for the two ALU inputs. The inputs to the ALU_mux are port 'A' of the RAM, port 'B' of the RAM, Q_register output 'Q_reg', an external data input 'Din' and logical '0'. This process drives the internal input signals 'R' and 'S' which are inputs to the ALU as well. Based on the source code in 'Isource', ALU_mux process selects values to be assigned to the 'S' and 'R' signals. This process suspends itself after executing its last statement, and resumes when a change occurs on one of the sensitivity list signals.

5.4.5.3- Register File (RAM)

The RAM has 16 memory locations which are 4-bit wide. It also has two ports 'A' and 'B'. The address at port 'A' is only for reading. The address at 'B' is for reading and writing. As shown in Figure 20, the register file is modeled using an array variable RAM for greater efficiency. The register file behaviour is modeled as a process which suspends processing until an input signal changes. If the clock signal is HIGH, a memory read operation is performed depending on the value of the two address ports 'AddrA' and 'AddrB'. If the clock is LOW and the 'WR-EN' is HIGH, a memory write operation is performed depending on the 'AddrB' port. The process maintains its internal memory variable, writing data to it and driving its output signals, 'A_latch' and 'B_latch', as directed by changes on its input signals. 'A_latch' and 'B_latch' are read by other concurrent statements in the behavioural model.
entity RAM16X4 is
    Port ( A : In std_logic_vector (3 downto 0);
          B : In std_logic_vector (3 downto 0);
          CP : In std_logic;
          R_MUX : In std_logic_vector (3 downto 0);
          WR_EN : In std_logic;
          A_LATCH : Out std_logic_vector (3 downto 0);
          B_LATCH : Out std_logic_vector (3 downto 0));
end RAM16X4;

architecture BEHAVIORAL of RAM16X4 is
    type MEMORY is array (0 to 15) of std_logic_vector(3 downto 0);
    begin
        RAM16 : process (A, B, CP, WR_EN, R_MUX)
            variable RAM: MEMORY := ("0001", "0010", "0000", "0000", "0000", "0000", "0000", "0000", "0000", "0000", "0000", "0000", "0000", "0000", "0000", "0000");
            begin
            -- modelled as a variable for greater efficiency;
            begin
                if CP = '1' then
                    -- clock is high; drive data output signals:
                    A_LATCH <= RAM(CONV_INTEGER(A));
                    B_LATCH <= RAM(CONV_INTEGER(B));
                else if WR_EN = '1' then
                    -- clock is low and write is enabled; write RAM data:
                    RAM(CONV_INTEGER(B)) := R_MUX;
                end if;
            end if;
        end process RAM16;
    end process;

Figure 20. A Portion of the Register File VHDL Description.

5.4.5.4- R_Multiplexer

R_mux is an up/down shifter at the input of the register file (RAM). This process drives the RAM input and the signals 'R_mux', 'RAM0', and 'RAM3', selecting values to be assigned as directed by the value of the current destination code in 'Idest'. R_mux is read by the RAM process as its data input bus.
5.4.5.5- QRegister

The Q_register is 4-bit wide. Its operation is controlled by the three most significant bits 'Idest' in the instruction. Q_register is formulated as a process 'Q_write' which writes a new value to the internal signal 'Q_reg' whenever 'CP' is rising and 'Idest' is '000' or '100' or '110'. The process is sensitive to the signals in its sensitivity list.

5.4.5.6- QMultiplexer

Q_mux is an up/down shifter. Two process are concurrently operated to model the Q_mux, Q_mux_value and Q_IO process. Q_mux_value is a register storage which selects a value to drive on 'Q_mux' according to the destination value in 'Idest'. It becomes active in response to a change on any of its input signals, 'Idest', 'F', 'Q_reg', 'Q0' and 'Q3'. Q_mux is read by the process Q_IO. The signals 'Q0' and 'Q3' are driven by Q_IO process as directed by the value of the current destination code in 'Idest'. Q_IO process is sensitive to 'F', 'Q_reg', 'Idest', 'Q0' and 'Q3' signals.

5.4.5.7- YOutput.

Y_output is a destination selector which decides whether to load the ALU output 'F' or the 'A' port to be forwarded to the external data output. This process drives the output signal 'Y', selecting a value to be assigned based on 'OE_n' and 'Idest'.

5.4.5.8- TOPAMD2901

Following the completion of the VHDL codes for each module, it is therefore necessary to create a symbol for the top-level entity. SGE was used to create the top-level symbol that takes input and output port names from the schematic and enters them into the top symbol. The VHDL description for the top-level includes the entity declaration, a complete architecture named SCHEMATIC, and a configuration statement. The entity declaration begins the definition of the behavioural model of the AMD2901. It declares the model’s name and its input/output signals, or ports. This declaration defines the model's interface with its enclosing designs. It defines the part of the model that is externally visible. Following this entity declaration is its corresponding architecture body. The architecture body defines the schematic of the model. Each module is represented by a component that is associated with its entity definition.

5.4.5.9- Testbench

For the top-level entity, SGE also creates a testbench template. The testbench template instantiates the top-level as a unit under test and creates a block named 'TB' where the added VHDL processes was put to drive the input ports and subsequently to read the output ports of the top-level during simulation. Simulation for each module of the design was performed to validate the functional behaviour of each portion as well as for the top-level. The testbench is completed with the appropriate VHDL codes to test and simulate each unit of the design. The simulation process and the related results will be discussed in the next chapter.
5.5- Experimental Implementation

For testing purposes, the entire bit-slice was divided into two blocks, the datapath (the algorithmic logic unit (ALU) with its multiplexer) and the memory (the register file 16x4 RAM with its associated shift registers). A bit-slice data path seems to be an ideal example for applying exhaustive test patterns to each bit-slice. The two experiments are implemented using both the BILBO structure and the combination of LFSR and MISR structures respectively. The test controller for both experiments is modeled and added to the existing AMD2901 circuitry. The following are the implementation behavioural descriptions of these structures using VHDL codes.

5.5.1- BILBO Structure.

For testing purposes, the BILBO register is modeled using a generate statement. The generate statement provides the capability of creating replicated structures. The generate in this modeling uses a FOR loop scheme to generate different BILBO registers based on the value of the generic parameters. In this proposed design for the first experiment the testing was accomplished using two BILBOs. In implementing these two structures, BILBO1 is 12-bit wide and BILBO2 is 8-bit wide. Both are of the same structure, the only difference being the width of each structure. As shown in Figure 21, the BILBO is modeled as two processes i.e. the selector process and the feedback process. The BILBO uses 'seed' and 'poly' specified via generic parameters for later input. The polynomial specifies the feedback flow. The BILBO inputs 'b_in' 2-bit port switches the -BILBO into one of four modes which are parallel input, shift mode, MISR mode, and reset mode respectively. The feedback process drives the internal 'forward' and 'feedback' signals. The two processes are concurrently executed.
ENTITY bilbo IS
GENERIC (seed, poly : BIT_VECTOR);
PORT (clk : IN BIT; b_in : IN BIT_VECTOR (2 DOWNTO 1);
p_in : IN BIT_VECTOR; s_in, ser_fb : IN BIT;
p_out : OUT BIT_VECTOR := seed; s_out : OUT BIT);
CONSTANT size : INTEGER := seed'LENGTH;
BEGIN
ASSERT seed'LENGTH = p_out'LENGTH REPORT "Bad seed length" SEVERITY NOTE;
ASSERT poly'LENGTH = p_out'LENGTH + 1 REPORT "Bad poly length" SEVERITY NOTE;
END bilbo;

ARCHITECTURE structural OF bilbo IS
SIGNAL forward : BIT_VECTOR (seed'LENGTH DOWNTO 0) := '0' & seed;
SIGNAL feedback : BIT_VECTOR (size DOWNTO 0);
BEGIN
m: FOR i IN 0 TO size - 1 GENERATE
clocking : PROCESS (clk)
BEGIN
IF (clk = '1') THEN
CASE b_in IS
WHEN "11" => forward(i) <= p_in(i);
WHEN "00" => forward(i) <= forward(i+1);
WHEN "01" => forward(i) <= forward(i+1) XOR p_in(i);
WHEN "10" => forward(i) <= '0'; --can use seed(i)
END CASE;
END IF;
END PROCESS;
feedbacking : PROCESS (forward(i), feedback(i))
BEGIN
IF poly(i) = '1' THEN
feedback(i+1) <= feedback(i) XOR forward(i);
ELSE
feedback(i+1) <= feedback(i);
END IF;
END PROCESS;
END GENERATE;
forward(size) <=
feedback(size) WHEN ser_fb = '1' AND poly(size) = '1' ELSE
s_in WHEN ser_fb = '0' ELSE
'0';
feedback(0) <= '0';
s_out <= forward(0);
p_out <= forward(size - 1 DOWNTO 0);
END structural;

Figure 21. VHDL Description of BILBO Structure.
5.5.2- LFSR Structure

For the second experiment the LFSR was used to generate test pattern sequences. It uses a seed and a polynomial. It generates serial and parallel outputs. The LFSR is 12-bit wide and used to model generative sequences of cells. The 'feedback' and 'forward' signals are driven internally by a process statement according to the 'poly' input, the 'feedback' XORed with the 'forward' signal to build the polynomial of the LFSR. The LFSR was initiated using the 'seed' input as a start test pattern. The pattern sequence is altered by different 'seed' inputs.

5.5.3- MISR Structure

The MISR is used to compress the signature while it is monitoring the output of the ALU during test process. It uses a 'seed' and a 'poly'. The parallel input is XORed with data being shifted into the shift register. The serial input is XORed with the left most bit. 'P_in' and 'S_in' are ignored if set to zero. The MISR process contains one generate statement to construct the 8-bit register. The process is sensitive to any change on the clock signal. It drives the forward and feedback signals internally. To control the feedback flow the 'poly' input is used. A high level portion of the behavioural VHDL description of the MISR implementation structure is shown in Figure 22.
ENTITY miar IS
  GENERIC (seed, poly: BIT_VECTOR);
  PORT (clk: IN BIT; p_in: IN BIT_VECTOR; s_in: IN BIT;
    p_out: OUT BIT_VECTOR= seed; s_out: OUT BIT);
  CONSTANT size: INTEGER := seed'LENGTH;
BEGIN
  ASSERT seed LENGTH = p_out' LENGTH REPORT " Bad seed length" SEVERITY NOTE;
  ASSERT poly LENGTH = p_out' LENGTH + 1 REPORT " Bad poly length" SEVERITY NOTE;
END miar;

ARCHITECTURE structural OF miar IS
  SIGNAL forward: BIT_VECTOR (size DOWNTO 0) := '0' & seed;
  SIGNAL feedback: BIT;
BEGIN
  m: FOR i IN 0 TO size - 1 GENERATE
    PROCESS (clk)
    BEGIN
      IF (clk = 1) THEN
        IF poly (i + 1) = 1 THEN
          forward (i) <= forward (i + 1) XOR p_in (i) XOR feedback;
        ELSE
          forward (i) <= forward (i + 1) XOR p_in (i);
        END IF;
      END IF;
    END PROCESS;
  END GENERATE;
  forward (size) <= s_in;
  feedback <= forward (0);
  s_out <= forward (size - 1 DOWNTO 0);
END structural.

Figure 22. Behavioural VHDL Description of the Signature Analysis Implementation Structure.

5.5.4- Test Controller

The test controller circuitry is added to control the test process. It is used to multiplex the internal signals with the input ports. The circuit is modeled as a concurrent process which responds to the test-mode signal. The test controller is behaviourally described for two different structures. It controls the BILBO modes in the first experiment by producing different output signals according to the test procedures that are executed. For the second experiment, it is used to control the two different test procedures.
5.5.5- Data Path Testing

The arithmetic and logic unit (ALU) is the main portion of the AMD2901 data path. Therefore, in order to make sure that all possible bit patterns are contained in the test vector sequence, the linear feedback shift register is implemented to have a maximum cycle length of $2^{12} - 1 = 4095$ words. The 8-bit output words of the ALU are clocked into the 8-bit MISR. The 'S', 'R', 'Cin', and 'I_{3.5}' are constructed from the 12 outputs of the LFSR. The 'F' 4-bit output result and the 4-bit status of the ALU operation are signed out as an output signature. The test vectors pass through the ALU multiplexer which would be simultaneously tested by applying fixed input data from the data in 'Din' port and getting the two operands 'A' & 'B' from the LFSR output. During testing, the outputs of the multiplexer are fed to the ALU. Only the outputs of the ALU are collected by the signature register for the verification of the test. This scheme accomplishes the first test mode.

5.5.6- Memory Testing

In developing the test style for the register file (memory) of the AMD2901 processor chip, a somewhat different approach has been taken for the following reasons:

- Specialized test pattern generators have been avoided because of the extra design overhead involved and because they are specialized and cannot be used as shared test hardware to test other blocks of logic in the same way that an LFSR can.

- The size of the memory on the AMD2901 chip is very small, i.e., the register file has only 16 x 4 words. Therefore, the efficiency of the test method in terms of the number of accesses of each address is of little importance.

For these reasons a form of exhaustive pattern test, which is not normally considered for memory, has been adopted. Besides, recent works on self-test for RAMs have also
pursued the aim of minimizing test length, often at the expense of using complex pattern generators to reproduce well known memory test sequences [DISL89].

The register file on the AMD2901 chip has separate read and write address ports making the read and write operations independent. The data input of the memory is connected directly to an LFSR. The data output of the memory is connected directly to an MISR.

5.5.7- RAM Self-test Strategy

Inserting the BIST circuit into the RAM is a two step process:

- First, replace the RAM address, data in, and control inputs from the system by the corresponding signals from the test control circuit and connect the RAM outputs to the BIST circuits.

- Second, connect the BIST select signal to the test control circuit and the BIST clock to the chip output ports.

The BIST circuitry adds a two to one multiplexer delay on the RAM address, data in, and control signals. As maybe expected, the hardware overhead is more acceptable for larger RAMs than for smaller ones.

5.6- Results

The VHDL implementation of the testable processor which incorporates BIST features into AMD2901 bit-slice processor has been illustrated. The functional simulation of the entire chip and the self test process simulation results will be verified in the next chapter.