CHAPTER 1
INTRODUCTION

An Automated Teller Machine (ATM) is a computerized device that allows customers to perform financial transactions without the need for direct interaction with a human teller. Common ATM functions include cash withdrawals, balance inquiries, fund transfers, and PIN changes. To design and model the behaviour of an ATM system, a Finite State Machine (FSM) is a powerful tool often used in the field of computer science and engineering.

A Finite State Machine (FSM) is a mathematical model used to represent a system with a limited number of states and transitions between these states based on inputs or conditions. An FSM helps in structuring the logical flow of operations in systems like ATMs, ensuring that each action is performed in a controlled sequence of steps, from inserting the card to completing a transaction.

The FSM used in an ATM system typically includes several key states that represent different stages of interaction between the user and the machine. These states help the ATM navigate through a series of operations.

Modeling the operation of an ATM using a Finite State Machine (FSM) helps in understanding and designing the flow of user interactions and system behaviour in a structured manner.

Each state and transition corresponds to a critical part of the ATM's operation, ensuring that the system functions smoothly and securely. By analysing the FSM, developers and engineers can identify potential issues, streamline processes, and improve the overall user experience.

CHAPTER 2

VLSI

2.1 VLSI :

VLSI stands for Very-Large-Scale Integration. It refers to the process of designing, manufacturing, and integrating a large number of electronic components, such as transistors, diodes, resistors, and capacitors, onto a single semiconductor chip, typically made of silicon.

VLSI (Very Large Scale Integration) is a technology used to create integrated circuits (ICs) by combining thousands, millions, or even billions of transistors and other components onto a single chip. This process significantly reduces the size and cost of electronic devices while increasing performance, making VLSI a key enabler in the development of modern electronic systems.

2.2 WHY VLSI:

The importance of VLSI lies in its ability to enable the development of complex electronic systems, driving innovation and advancements in various fields.

The importance of VLSI lies in its ability to enable the development of complex electronic systems, driving innovation and advancements in various fields.

VLSI (Very Large Scale Integration) has become a cornerstone of modern electronics for several compelling reasons.

The move towards VLSI technology has been driven by the need to improve the performance, efficiency, and affordability of electronic systems while reducing their size.

CHAPTER 3

MODELSIM

3.1 MODEL SIM APP

ModelSim is a simulation tool developed by Mentor Graphics, commonly used for simulating digital logic designs. It's widely adopted in electronic design automation (EDA) for verifying the functionality of digital circuits before they are fabricated. ModelSim supports VHDL, Verilog, and SystemVerilog, making it versatile for engineers working with different hardware description languages (HDLs).

Fig. 3.1. MODEL SIM LOGO

3.1.1 INTRODUCTION:

ModelSim is a powerful simulation and debugging tool developed by Mentor Graphics, tailored for digital circuit design. It allows engineers to validate, analyze, and troubleshoot digital systems by providing a virtual environment where they can simulate hardware descriptions before physically implementing them. ModelSim supports several hardware description languages, including VHDL, Verilog, and SystemVerilog, making it versatile for a broad range of applications in electronic design automation (EDA).

3.1.2 DEVELOPER:

ModelSim was developed by Mentor Graphics, a prominent electronic design automation (EDA) company based in the United States. Founded in 1981 and headquartered in Wilsonville, Oregon, Mentor Graphics established itself as one of the major players in the EDA industry, offering a suite of tools to aid in the design and simulation of electronic systems. Mentor Graphics developed ModelSim as a simulation tool that could work across multiple hardware description languages (HDLs), including VHDL, Verilog, and SystemVerilog.

Mentor Graphics' mission with ModelSim was to create a powerful and versatile environment that would meet the needs of engineers working with FPGAs, ASICs, and complex digital systems. In 2017, Siemens, a global technology company, acquired Mentor Graphics and integrated it into its Siemens Digital Industries Software division. This acquisition further strengthened the development and support of ModelSim as it became part of Siemens' broader offerings for industrial and digital design tools.Today, Siemens continues to develop and support ModelSim and its extended version, Questa Advanced Simulator, which offers enhanced debugging and simulation capabilities for complex systems. With Siemens' backing, ModelSim remains a foundational tool in digital design, widely used by engineers and educators around the world for simulation and verification in EDA workflows.

3.1.3.Importance of ModelSim in Digital Design Verification:

ModelSim plays a crucial role in the design and verification of digital systems by providing a simulation environment that allows engineers to test, analyze, and refine their designs virtually. This pre-manufacturing verification step is invaluable, as errors in hardware design can be costly and time-consuming to fix if discovered post-production. ModelSim enables designers to simulate the behavior of digital components, such as processors, memory blocks, and custom logic circuits, ensuring their functionality under various conditions before implementation. The tool supports multiple hardware description languages (HDLs) like VHDL, Verilog, and SystemVerilog, making it versatile and accessible to engineers with different design needs.

3.1.4. Mixed-Language Simulation and Versatility:

One of ModelSim's standout features is its mixed-language simulation capability, which allows engineers to work with multiple HDLs within a single design environment. This feature is particularly useful in large or collaborative projects where different parts of a design might be developed in different languages. For instance, a project might include a control unit written in VHDL and a data processing module written in Verilog**.**

3.1.5. Waveform Viewer and Signal Debugging

The waveform viewer in ModelSim is a powerful feature for signal analysis, allowing users to observe the behavior of signals over time visually. This is particularly important for debugging and understanding the interactions between components in a digital design. By setting up signals to be monitored during a simulation, designers can view how data flows through the circuit and identify any unexpected behavior.

3.1.6.Automation with TCL Scripting

To enhance efficiency, ModelSim offers support for TCL (Tool Command Language) scripting, which enables users to automate simulation processes and perform batch testing. Through TCL scripts, users can set up complex simulation scenarios, control simulation runs, and extract relevant data, all without requiring manual input. This automation capability is particularly useful for large projects that involve repetitive testing across multiple test cases, where running each simulation manually would be highly time-consuming.

3.1.7. Simulation Modes: Event-Driven and Cycle-Based

ModelSim offers different simulation modes, including event-driven and cycle-based simulation, to optimize performance based on the specific requirements of a project. Event-driven simulation is often used for designs where changes occur sporadically, as it only processes parts of the circuit that are affected by a change.

3.2 SIMULATION,OPTIMIZATION AND PROCESS

3.2.1 Simulation:

Simulation in ModelSim is a critical process that allows designers to test and validate the functionality of digital circuits in a virtual environment before committing to hardware. ModelSim supports a range of simulations, from functional simulations, which verify the logical correctness of the design, to more complex timing simulations, which ensure that circuits meet timing requirements under real-world conditions. Functional simulation is generally used in the initial design phases to test the core logic and behavior of the digital circuit. Timing simulation, on the other hand, takes into account propagation delays and setup times, providing a more accurate reflection of how the design will perform on actual hardware.

Within ModelSim, simulation is executed by creating a testbench, a dedicated HDL code module that defines inputs and monitors outputs for the circuit being tested. The testbench simulates various input scenarios, allowing designers to observe the circuit's responses and ensure that the design behaves as expected. This process is especially valuable for verifying complex interactions, such as those found in processors, memory units, and communication protocols. ModelSim's waveform viewer enhances the simulation process by visually displaying the behavior of signals over time, making it easier for designers to track signal states, debug issues, and verify timing constraints. Through simulation, ModelSim helps catch errors early, preventing costly revisions and ensuring that designs meet their intended specifications before moving to the hardware stage.

3.2.2 Optimization:

Optimization in ModelSim focuses on improving the simulation process to make it faster, more efficient, and capable of handling larger designs. ModelSim offers different simulation modes, such as event-driven and cycle-based simulation, which can be chosen based on the design's requirements. Event-driven simulation is often used for circuits with sporadic signal changes, as it processes only parts of the design that are affected by changes, thus saving computational power. This mode is ideal for

large, complex designs where certain parts of the circuit remain static over time. Conversely, cycle-based simulation is optimized for synchronous designs with periodic signal changes aligned with clock cycles, making it faster for clock-driven circuits like digital signal processors (DSPs) and synchronous memory.

ModelSim also allows users to improve efficiency by adjusting simulation parameters, such as time resolution, to control the precision of the simulation. For high-level functional testing, designers may use a coarser resolution, which speeds up the simulation at the cost of detail. For timing-sensitive applications, finer resolutions can be set to capture subtle timing variations. Additionally, ModelSim supports optimization through the use of TCL scripting, which automates repetitive tasks, batch tests, and regression testing. By scripting simulation processes, designers can save time, streamline workflows, and ensure consistency across multiple test cases. These optimization techniques make ModelSim adaptable to various design needs, from quick functional checks to detailed timing analysis.

3.2.3 Process:

Running a simulation in ModelSim follows a structured process, starting with setting up the project and progressing through testing and analysis. The first step involves creating a new project in ModelSim, where design files, including the HDL source files and testbenches, are added. Once the files are compiled, ModelSim checks for syntax errors, ensuring that the design is free from basic coding issues. After successful compilation, the designer initiates the simulation environment, where they can configure various simulation settings, such as specifying simulation duration, adding signal watches, and setting up breakpoints for debugging.

The core of the simulation process is the testbench, which drives inputs into the design and captures outputs for analysis. For example, a testbench might simulate different input values for an arithmetic logic unit (ALU) to verify its operations, or it might drive clock and control signals for a memory module to check read/write functionality. Once the simulation runs, ModelSim generates a waveform output that displays signal transitions over time, which can be analyzed to verify logical correctness and timing. ModelSim’s interactive environment allows designers to make

on-the-fly adjustments, re-run simulations with new parameters, and add or remove signals for monitoring, making it a flexible tool for iterative testing.

After completing the simulation, designers review the results in ModelSim's waveform viewer to identify and resolve any issues. They may use debugging features such as variable watches, breakpoints, and value tracing to drill down into specific problem areas. Finally, TCL scripting can automate the simulation process, particularly for larger projects where multiple test cases need to be verified consistently.

3.3 FEATURES OF MODEL SIM

3.3.1 Multi-Language Support:

ModelSim provides support for multiple hardware description languages, including VHDL, Verilog, and SystemVerilog. This makes it versatile and allows designers to work on projects using different languages within a single environment. Mixed-language support is particularly useful for larger projects or collaborative teams where components may be written in different languages, as ModelSim enables seamless integration and simulation without needing to convert code between languages.

3.3.2 Comprehensive Waveform Viewer:

The waveform viewer in ModelSim allows designers to visualize signal behaviors over time, which is essential for debugging and verifying design functionality. Signals can be monitored, and timing relationships can be analyzed in real-time, making it easy to identify issues in the logic or timing of a circuit. The waveform viewer supports zooming, panning, and the setting of markers to examine specific time intervals, which is crucial for detailed analysis.

3.3.3 Advanced Simulation Modes:

ModelSim supports both event-driven and cycle-based simulation modes. Event-driven simulation only processes parts of the circuit that have changed, making it efficient for larger designs with static sections. Cycle-based simulation, on the other

hand, is optimized for clock-driven designs and is typically faster for synchronous circuits. These options give designers the flexibility to choose the most efficient mode based on the specific requirements of their design.

3.3.4 TCL Scripting for Automation:

With TCL (Tool Command Language) scripting, users can automate a wide range of tasks in ModelSim, from compiling and running simulations to setting up test cases and extracting data. TCL scripting is particularly valuable for regression testing, where existing test cases are re-run automatically after any design modifications, ensuring consistency and reliability. This scripting capability streamlines repetitive tasks and enhances productivity in complex projects.

3.3.5 Interactive Debugging Tools:

ModelSim provides several debugging tools, including breakpoints, watches, and variable tracing, to help designers pinpoint and resolve issues in their design. Breakpoints allow users to pause the simulation at specific points, while watches and tracing track the values of variables or signals in real-time. These tools help designers gain deeper insights into their design, making it easier to diagnose and fix errors early in the development cycle.

3.3.6 Code Coverage Analysis:

ModelSim includes code coverage analysis features, which help designers determine how much of their HDL code has been tested. Code coverage includes statement coverage, branch coverage, and condition coverage, among others. This feature is useful in ensuring that all parts of the code are exercised during testing, which is essential for identifying untested or redundant code and improving the overall reliability of the design.

3.3.7 Regression Testing:

ModelSim's regression testing capabilities allow designers to re-run previously defined test cases whenever the design is modified. This ensures that changes do not introduce new errors or affect other parts of the circuit. Regression testing is automated through TCL scripting, making it highly efficient for large projects where multiple iterations are common. It is crucial for maintaining consistency and reliability in designs that undergo frequent updates.

3.4 WHY AND HOW TESTBENCH NEEDED FOR MODEL SIM?

3.4.1 Why?

In digital circuit design, verifying that each component behaves as expected before hardware implementation is critical to avoid costly errors and time-consuming revisions. Testbenches fulfill this role by creating a simulation environment that allows designers to observe and validate the behavior of a circuit, or *design-under-test (DUT)*, without physical hardware. A testbench acts as a virtual testing harness for the DUT, driving inputs to the circuit and monitoring its outputs in response. This is particularly crucial in complex digital systems, where manual testing would be impractical or incomplete due to the sheer number of possible input combinations and operating scenarios.The need for a testbench arises from the limitations of hardware-based testing. In a lab setting, testing every possible scenario might involve expensive prototyping and multiple iterations, whereas a testbench enables designers to run simulations early in the design cycle, identifying logical or functional flaws before committing to physical fabrication. Testbenches help validate both basic functionality and edge cases—the unusual, extreme conditions under which a circuit might behave unexpectedly. By simulating these cases, designers ensure their circuits are robust and will function as intended in various real-world conditions.Automated testing is another significant benefit of using testbenches in ModelSim. With a testbench, a designer can define a series of test vectors, which are sequences of inputs that simulate different operational scenarios, and apply them to the DUT. This automation ensures that the DUT’s behavior is consistent across multiple simulation runs, promoting reliability as the design evolves.

3.4.2 How?

Creating a testbench in ModelSim involves setting up a new HDL module that interacts with the DUT, generating inputs and monitoring outputs to test the circuit's behavior in response to various conditions. The testbench itself doesn't have any inputs or outputs; instead, it directly instantiates the DUT and defines the stimuli (inputs) that will drive the design. This modular setup allows the testbench to operate independently of the DUT, promoting reusable code and flexible testing.

The process starts with defining the testbench module in Verilog, VHDL, or SystemVerilog, depending on the language of the DUT. Inside the testbench, the designer creates *test vectors*, which are sequences of inputs applied to the DUT to simulate different operational states. These vectors are designed to cover typical use cases as well as edge cases. For example, in an ALU design, test vectors might include a variety of arithmetic operations (like addition, subtraction, multiplication, and division) with different operands to ensure the ALU performs correctly under all possible inputs.

Once the test vectors are set up, the testbench applies these inputs to the DUT over specific intervals. This is often done by setting up time delays between each input change, allowing the DUT's response to stabilize before the next input is applied. In ModelSim, the simulation timeline can be controlled precisely, letting designers define exact intervals for each test vector.

This timing control is essential for testing sequential circuits, where the behavior depends not only on the current inputs but also on the sequence and timing of previous input.

CHAPTER 4

QUARTUS PRIME

4.1 QUARTUS PRIME APP

Quartus Prime is a comprehensive design software suite developed by Intel (formerly Altera) for the development of digital circuits on FPGAs (Field Programmable Gate Arrays), CPLDs (Complex Programmable Logic Devices), and other programmable logic devices. The software provides a complete solution for the design, synthesis, simulation, and implementation of digital systems, making it highly valuable for both academic and industry applications.

Fig. 4.1 QUARTUS PRIME LOGO

4.2 STEPS INVOLED IN QUARTUS PRIME

In Quartus Prime, synthesis is the process of converting your design written in Hardware Description Language (HDL) like VHDL or Verilog into a network of logic gates and interconnections that can be programmed onto an FPGA or CPLD. Here's a step-by-step guide to performing synthesis in Quartus Prime.

4.2.1. Create a New Project

- Open Quartus Prime.
- Go to File > New Project Wizard.
- Follow the wizard steps to create a new project, specifying your design files and target device.

4.2.2. Add HDL Files

In the Project Navigator, right-click on Files and select Add/Remove Files in Project.Add your HDL (VHDL/Verilog) files here, which describe the logic of your design.

4.2.3. Set Top-Level Entity

Specify the top-level entity (the main module or entity for synthesis).

Go to Assignments > Settings > General, and under Top-level entity, select the name of the top module in your design.

4.2.4. Configure Synthesis Settings (Optional)

Go to Assignments > Settings > Compiler Settings > Synthesis to customize synthesis options.You can adjust optimization levels, area, and timing constraints, depending on your requirements.

4.2.5. Run Analysis & Synthesis

In the Tasks pane, select Analysis & Synthesis or go to Processing > Start > Analysis & Synthesis.

Quartus Prime will then parse, analyse, and synthesize your design into a netlist.

Check the Messages tab to see if there are any errors or warnings.

4.2.6. Review Synthesis Results

Once synthesis completes, you can view the generated RTL Viewer and Technology Map Viewer to analyse the synthesized logic.

Go to Tools > Netlist Viewers and select RTL Viewer or Technology Map Viewer.

4.2.7. Optimize (Optional)

If needed, make adjustments to improve performance, such as adding timing constraints, adjusting logic, or reconfiguring synthesis options.

After adjustments, rerun synthesis to see the impact.

4.3 WHY AND HOW RTL IS NEEDED FOR QUARTUS PRIME ?

Quartus Prime is an FPGA (Field-Programmable Gate Array) design software developed by Intel (formerly Altera). It is a comprehensive suite for designing, simulating, synthesizing, and programming FPGAs and CPLDs. Here's a summary of its core functionalities:

4.3.1. Design Entry

HDL Support: Quartus Prime supports hardware description languages like VHDL, Verilog, and System Verilog, allowing users to design digital systems.

Block Design: Offers a graphical design entry through the Block Diagram Editor, where you can connect functional blocks visually, which can be useful for complex systems.

IP Integration: Includes a library of Intellectual Property (IP) blocks, such as processors, memory controllers, and DSP modules, that can be incorporated into designs.

4.3.2. Synthesis

Converts HDL code or block designs into a netlist—a low-level description of the hardware in terms of logic gates and connections.

Optimization of the design for timing, area, or power, based on the FPGA architecture.

4.3.3. Placement and Routing (Fitting)

Placement: Assigns synthesized logic elements (like logic gates and registers) to specific locations on the FPGA.

Routing: Determines the interconnections between these elements. This is crucial for meeting timing constraints and ensuring design reliability.

4.3.4. Timing Analysis

Static Timing Analysis (STA): Quartus Prime checks timing constraints by analyzing paths and ensuring the design can meet speed requirements.

Timing Closure: Tools for adjusting and optimizing timing issues in critical paths.

4.3.5. Simulation and Debugging

Simulation: Quartus Prime integrates with tools like ModelSim (Intel edition) for simulating HDL code before programming the FPGA, which helps verify logic and timing.

Signal Tap Logic Analyzer: A built-in tool that allows in-system debugging, enabling you to capture and analyze signals directly on the FPGA in real-time.

4.3.6. Programming and Configuration

After synthesis and fitting, Quartus Prime generates a programming file (e.g., .sof or .pof) that can be uploaded to the FPGA.

Using the Programmer Tool, you can directly download the design onto the FPGA through JTAG or other interfaces.

4.3.7. Power Analysis

Performs power estimation and optimization to predict the design's power consumption, helping to ensure it meets thermal and energy requirements.

4.3.8. IP and Platform Designer

- Integrates third-party IP cores and components, making it easier to build complex systems.
- The Platform Designer enables the integration of processor systems (e.g., Nios II soft processor) and other complex IPs for embedded system designs.

4.3.9. Advanced Optimization Tools

Quartus Prime offers advanced optimization for specific applications, like high-speed DSP designs, data-intensive applications, and machine learning models, using Intel's High-Level Synthesis (HLS) or OpenCL.

CHAPTER 5

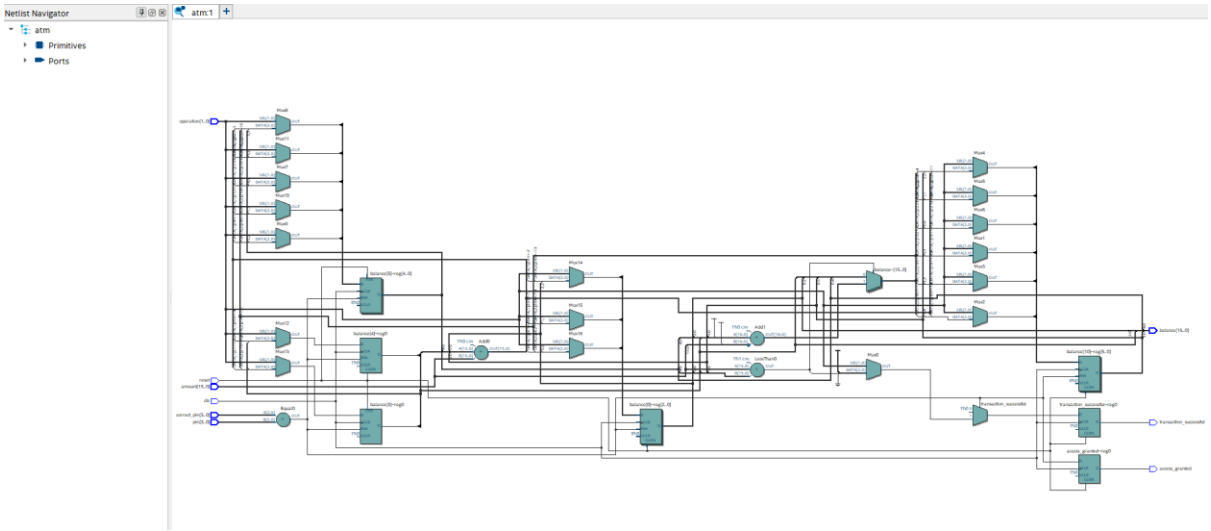
PROGRAM

5.1 RTL CODE

```
module atm (
    input clk,
    input reset,
    input [1:0] operation,  // 00: Check Balance, 01: Deposit, 10: Withdraw
    input [15:0] amount,    // Amount for deposit/withdrawal
    input [3:0] pin,        // User PIN input
    input [3:0] correct_pin, // Correct PIN for verification
    output reg [15:0] balance, // Balance output
    output reg access_granted, // 1 if correct PIN entered
    output reg transaction_successful // 1 if transaction completed successfully
); // States for ATM operations
    parameter CHECK_BALANCE = 2'b00;
    parameter DEPOSIT       = 2'b01;
    parameter WITHDRAW      = 2'b10;
    // Initial balance
    initial begin
        balance = 16'd1000; // Example initial balance
    end
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            balance <= 16'd1000;       // Reset balance to initial
            access_granted <= 0;
            transaction_successful <= 0;
        end else begin
            // PIN verification
            if (pin == correct_pin) begin
```

```
            access_granted <= 1;
            transaction_successful <= 0;
            case (operation)
               CHECK_BALANCE: begin
                  transaction_successful <= 1;
               end
               DEPOSIT: begin
                  balance <= balance + amount;
                  transaction_successful <= 1;
               end
               WITHDRAW: begin
                  if (amount <= balance) begin
                     balance <= balance - amount;
                     transaction_successful <= 1;
                  end else begin
                     transaction_successful <= 0; // Insufficient funds
                  end
               end
               default: begin
                  transaction_successful <= 0;
               end
            endcase
         end else begin
            access_granted <= 0;
            transaction_successful <= 0;
         end
      end
   end
endmodule
```

Synthesis:



5.1.1 Explanation about RTL code

5.1.1.1 Module Declaration: Defines the ATM module with inputs and outputs.

Inputs:

clk: Clock signal that drives the sequential logic.

reset: Resets the ATM state, including balance and status flags.

operation: Specifies the operation to be performed (00 for balance check, 01 for deposit, 10 for withdrawal).

amount: Holds the amount of money for deposit or withdrawal operations.

pin and correct_pin: Used for PIN verification, ensuring secure access.

Outputs:

balance: Stores the current balance.

access_granted: Indicates if the correct PIN has been entered.

transaction_successful: Shows if the transaction completed successfully.

5.1.1.2 These parameters define operation codes:

CHECK_BALANCE: Code 00 for checking balance.

DEPOSIT: Code 01 for depositing funds.

WITHDRAW: Code 10 for withdrawing funds.

Using parameters makes the code more readable and maintainable.

Initial Block: Sets an initial balance of 1000 units for the account at the beginning of the simulation. This is primarily for testing purposes and helps simulate an existing balance in the account.

5.1.1.3 Detailed Explanation of the always Block:

Triggering Condition: This block executes on every rising edge of clk (clock) or when reset is high.

Reset Condition:

When reset is high, the balance is reset to the initial value of 1000, and access_granted and transaction_successful are set to 0, clearing any previous transaction or access status.

PIN Verification and Transaction Handling:

If reset is not active, the code first checks if the entered pin matches correct_pin.

If the PIN is correct (pin == correct_pin), access_granted is set to 1, allowing access to the account. transaction_successful is initially set to 0 until the transaction completes.

5.1.1.4 Transaction Handling (controlled by operation):

CHECK_BALANCE (when operation == 00): Sets transaction_successful to 1, as checking the balance doesn't change it.

DEPOSIT (when operation == 01): Adds amount to balance, completing a deposit. transaction_successful is set to 1.

WITHDRAW (when operation == 10): Checks if amount is less than or equal to balance.

If true, it subtracts amount from balance (successful withdrawal) and sets transaction_successful to 1.

If false, transaction_successful remains 0, signaling insufficient funds.

Invalid Operation Handling:

If the operation value doesn't match any defined case, transaction_successful is set to 0 as a default response.

Incorrect PIN:

If the PIN is incorrect (pin != correct_pin), access_granted and transaction_successful are both set to 0, denying any transactions.

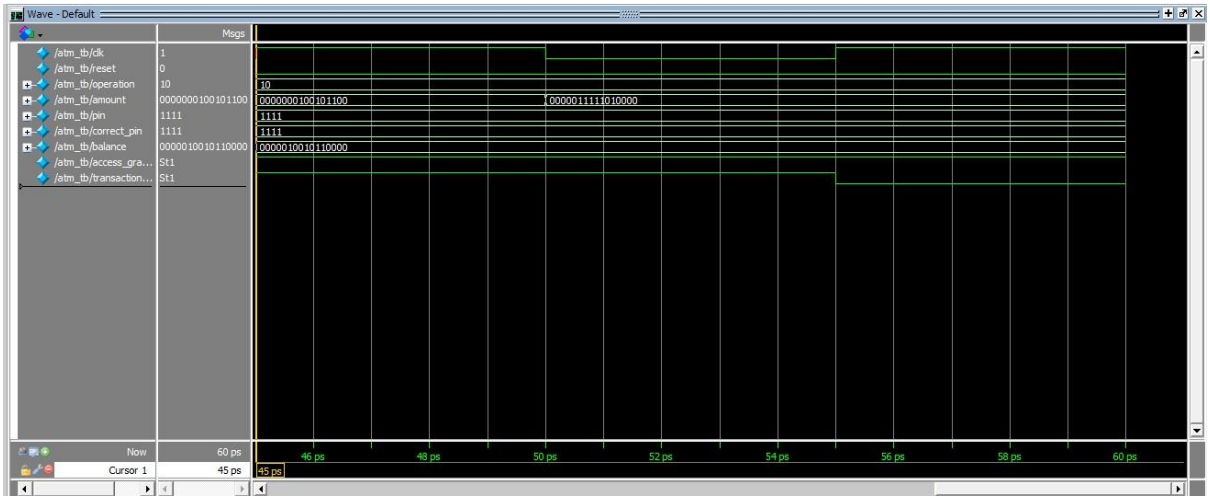
5.2 TEST BENCH CODE

```
module atm_tb;
    // Testbench signals
    reg clk;
    reg reset;
    reg [1:0] operation;
    reg [15:0] amount;
    reg [3:0] pin;
    reg [3:0] correct_pin;
    wire [15:0] balance;
    wire access_granted;
    wire transaction_successful;
    // Instantiate ATM module
    atm uut (
        .clk(clk),
        .reset(reset),
        .operation(operation),
        .amount(amount),
        .pin(pin),
        .correct_pin(correct_pin),
        .balance(balance),
        .access_granted(access_granted),
        .transaction_successful(transaction_successful)
    );
    // Clock generation
    initial begin
        clk = 0;
```

```
        forever #5 clk = ~clk;
    end
    // Test sequence
    initial begin
        // Initial conditions
        reset = 1;
        pin = 4'b0000;
        correct_pin = 4'b1111;  // Set a test PIN
        #10 reset = 0;
        // Test case 1: Incorrect PIN attempt
        pin = 4'b1010;
        operation = 2'b00; // Check Balance
        #10;
        $display("Test 1 - Access Granted: %b, Balance: %d", access_granted,
balance);
        // Test case 2: Correct PIN, Check Balance
        pin = correct_pin;
        operation = 2'b00; // Check Balance
        #10;
        $display("Test 2 - Access Granted: %b, Balance: %d", access_granted,
balance);
        // Test case 3: Correct PIN, Deposit
        operation = 2'b01; // Deposit
        amount = 16'd500;
        #10;
        $display("Test 3 - Balance after deposit: %d, Transaction Successful: %b",
balance, transaction_successful);
        // Test case 4: Correct PIN, Withdraw with sufficient balance
        operation = 2'b10; // Withdraw
        amount = 16'd300;
        #10;
```

```
        $display("Test 4 - Balance after withdrawal: %d, Transaction Successful: %b",
balance, transaction_successful);
        // Test case 5: Correct PIN, Withdraw with insufficient balance
        operation = 2'b10; // Withdraw
        amount = 16'd2000;
        #10;
        $display("Test 5 - Balance after failed withdrawal: %d, Transaction Successful:
%b", balance, transaction_successful);
        // End test
        $finish;
    end
endmodule
```

Simulation:



5.2.1 Explanation about Test bench code

5.2.1.1 Declaring the Testbench Signals

```
    reg clk;
    reg reset;
    reg [1:0] operation;
    reg [15:0] amount;
    reg [3:0] pin;
```

```
reg [3:0] correct_pin;
wire [15:0] balance;
wire access_granted;
wire transaction_successful;
```

Registers (reg) are used as inputs to the atm module in the testbench, so we can set their values in our test sequence.

Wires (wire) are used to connect the output signals from the atm module, so they can be observed by the testbench.

5.2.1.2 Instantiating the ATM Module

```
atm uut (
    .clk(clk),
    .reset(reset),
    .operation(operation),
    .amount(amount),
    .pin(pin),
    .correct_pin(correct_pin),
    .balance(balance),
    .access_granted(access_granted),
    .transaction_successful(transaction_successful)
);
```

The ATM module (atm) is instantiated and connected to the testbench signals.

uut stands for "unit under test," a common naming convention in testbenches.

This connection allows us to set input signals (like clk, operation, etc.) and observe the outputs (balance, access_granted, transaction_successful).

5.2.1.3 Clock Generation

```
initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
```

This initial block generates a clock signal (clk) that toggles every 5 time units.
The forever loop continuously inverts clk (from 0 to 1 and vice versa), producing a square wave.
A clock is essential in synchronous designs, as the module performs actions on each positive edge of clk.

5.2.1.4Test Sequence

```
initial begin
    // Initial conditions
    reset = 1;
    pin = 4'b0000;
    correct_pin = 4'b1111;  // Set a test PIN
    #10 reset = 0;
```

The test sequence is defined in an initial block.
Reset is asserted (set to 1) initially to initialize the module.
After 10 time units (#10), the reset signal is set to 0, deactivating the reset.

5.2.2.Test Cases

Test Case 1: Incorrect PIN Attempt

```
pin = 4'b1010;
operation = 2'b00; // Check Balance
#10;
$display("Test 1 - Access Granted: %b, Balance: %d", access_granted, balance);
```

Sets an incorrect PIN (1010) and tries a balance check.
Waits for 10 time units and displays access_granted and balance.
Expected Result: access_granted should be 0, and the balance should remain unchanged.

Test Case 2: Correct PIN, Check Balance

```
pin = correct_pin;
operation = 2'b00; // Check Balance
```

```
    #10;
    $display("Test 2 - Access Granted: %b, Balance: %d", access_granted, balance);
```

Enters the correct PIN and checks balance.

Expected Result: access_granted should be 1, and transaction_successful should be 1.

Test Case 3: Correct PIN, Deposit

```
    operation = 2'b01; // Deposit
    amount = 16'd500;
    #10;
    $display("Test 3 - Balance after deposit: %d, Transaction Successful: %b",
balance, transaction_successful);
```

Uses the correct PIN to perform a deposit of 500.

Expected Result: balance should increase by 500, and transaction_successful should be 1.

Test Case 4: Correct PIN, Withdraw with Sufficient Balance

```
    operation = 2'b10; // Withdraw
    amount = 16'd300;
    #10;
    $display("Test 4 - Balance after withdrawal: %d, Transaction Successful: %b",
balance, transaction_successful);
```

Uses the correct PIN to perform a withdrawal of 300, which is within the balance limit.

Expected Result: balance should decrease by 300, and transaction_successful should be 1.

Test Case 5: Correct PIN, Withdraw with Insufficient Balance

```
    operation = 2'b10; // Withdraw
    amount = 16'd2000;
    #10;
```

$display("Test 5 - Balance after failed withdrawal: %d, Transaction Successful: %b", balance, transaction_successful);

Attempts a withdrawal of 2000, which exceeds the balance.

Expected Result: balance remains unchanged, and transaction_successful should be 0 (indicating insufficient funds).

End of Simulation

$finish;

$finish terminates the simulation once all test cases have run.

5.2.3. Test Bench with Error

5.2.3.1. Missing Clock Generation

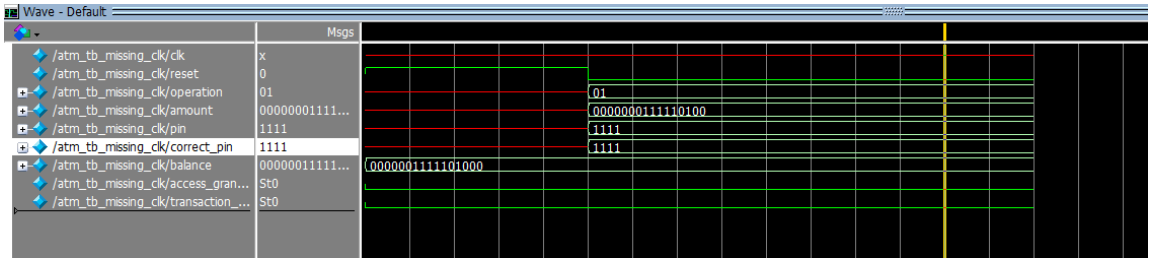
In this testbench, the clk signal is not toggled, so the ATM module won't function as expected since it relies on a clock edge.

```
module atm_tb_missing_clk;
  reg clk;
  reg reset;
  reg [1:0] operation;
  reg [15:0] amount;
  reg [3:0] pin;
  reg [3:0] correct_pin;
  wire [15:0] balance;
  wire access_granted;
  wire transaction_successful;
  atm uut (
    .clk(clk),
    .reset(reset),
    .operation(operation),
    .amount(amount),
    .pin(pin),
    .correct_pin(correct_pin),
    .balance(balance),
```

```
        .access_granted(access_granted),
        .transaction_successful(transaction_successful));
    initial begin
        reset = 1;
        #10 reset = 0;
        // Test operation without clock toggling
        pin = 4'b1111;
        correct_pin = 4'b1111;
        operation = 2'b01;
        amount = 16'd500;
        #20;
        $display("Test Missing Clock - Balance: %d, Transaction Successful: %b",
balance, transaction_successful);
        $finish;
    end
endmodule
```

Error: The clock (clk) is not toggled. The ATM module won't process operations because it depends on the clock signal.

Simulation:



5.2.3.2.Incorrect Reset Behaviour

This testbench deactivates the reset signal too soon, potentially leaving the module in an undefined state.

```
module atm_tb_early_reset;
    reg clk;
    reg reset;
```

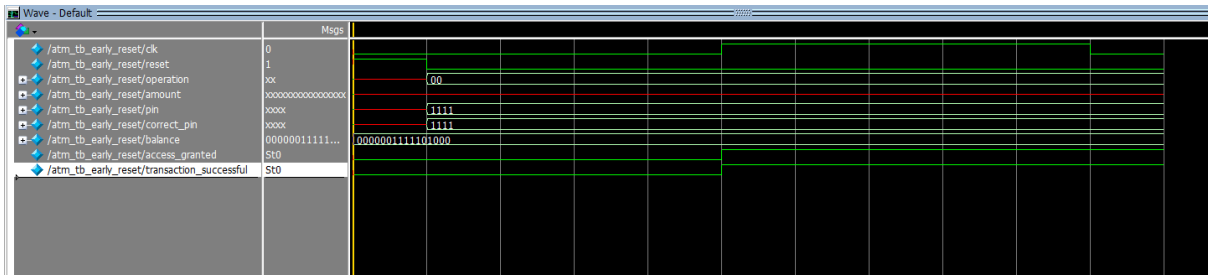
```
reg [1:0] operation;
reg [15:0] amount;
reg [3:0] pin;
reg [3:0] correct_pin;
wire [15:0] balance;
wire access_granted;
wire transaction_successful;
atm uut (
    .clk(clk),
    .reset(reset),
    .operation(operation),
    .amount(amount),
    .pin(pin),
    .correct_pin(correct_pin),
    .balance(balance),
    .access_granted(access_granted),
    .transaction_successful(transaction_successful) );

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
initial begin
    reset = 1;
    #1 reset = 0;  // Reset deactivated too early
    // Attempt to perform operations
    pin = 4'b1111;
    correct_pin = 4'b1111;
    operation = 2'b00; // Check Balance
    #10;
```

```
        $display("Test Early Reset - Balance: %d, Access Granted: %b", balance,
access_granted);
        $finish;
    end
endmodule
```

Error: The reset signal is deactivated after just one time unit (#1), potentially causing the module to miss initialization.

Simulation:



5.2.3.3.Incorrect PIN Test

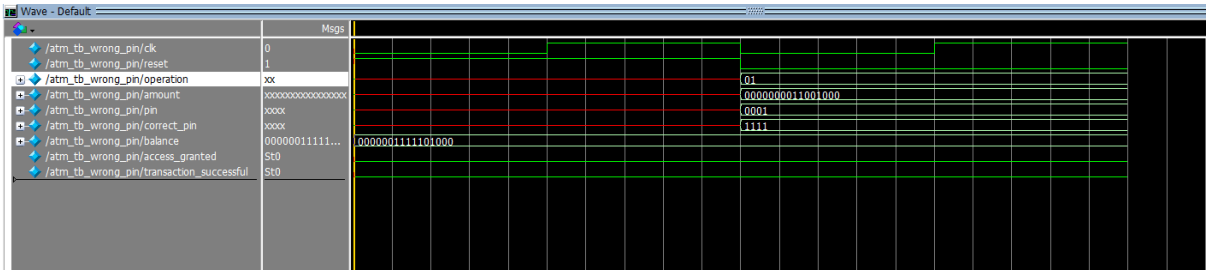
This testbench sets an incorrect PIN but expects the transaction to succeed, which demonstrates a misunderstanding of the module's behavior.

```
module atm_tb_wrong_pin;
    reg clk;
    reg reset;
    reg [1:0] operation;
    reg [15:0] amount;
    reg [3:0] pin;
    reg [3:0] correct_pin;
    wire [15:0] balance;
    wire access_granted;
    wire transaction_successful;
    atm uut (
        .clk(clk),
        .reset(reset),
```

```
        .operation(operation),
        .amount(amount),
        .pin(pin),
        .correct_pin(correct_pin),
        .balance(balance),
        .access_granted(access_granted),
        .transaction_successful(transaction_successful) );
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
    initial begin
        reset = 1;
        #10 reset = 0;
        pin = 4'b0001;  // Incorrect PIN
        correct_pin = 4'b1111;
        operation = 2'b01; // Deposit
        amount = 16'd200;
        #10;
        $display("Test Wrong PIN - Balance: %d, Access Granted: %b, Transaction
Successful: %b", balance, access_granted, transaction_successful);
        $finish;
    end
endmodule
```

Error: The testbench sets the wrong PIN (0001) but might incorrectly expect the transaction to proceed.

Simulation:



5.2.3.4.Insufficient Wait Time

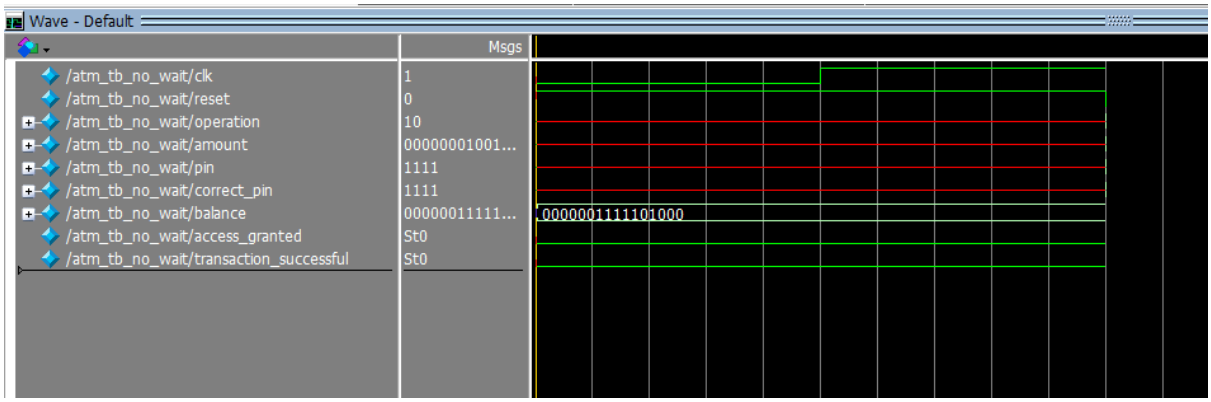
This testbench attempts to observe outputs too quickly without waiting long enough for the operation to complete.

```
module atm_tb_no_wait;
    reg clk;
    reg reset;
    reg [1:0] operation;
    reg [15:0] amount;
    reg [3:0] pin;
    reg [3:0] correct_pin;
    wire [15:0] balance;
    wire access_granted;
    wire transaction_successful;
    atm uut (
        .clk(clk), .reset(reset),.operation(operation), .amount(amount),.pin(pin),
.correct_pin(correct_pin),.balance(balance),.access_granted(access_granted),
.transaction_successful(transaction_successful));
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
    initial begin
        reset = 1;
        #10 reset = 0;
```

```
        pin = 4'b1111;
        correct_pin = 4'b1111;
        operation = 2'b10; // Withdraw
        amount = 16'd300;
        $display("Test No Wait - Balance: %d, Transaction Successful: %b", balance,
transaction_successful);
        $finish;
    end
endmodule
```

Error: The testbench attempts to print the balance and transaction status immediately after setting inputs, without waiting for the operation to complete.

Simulation:



5.2.3.5.Incorrect Operation Code

This testbench uses an undefined operation code, which might lead to unexpected behavior.

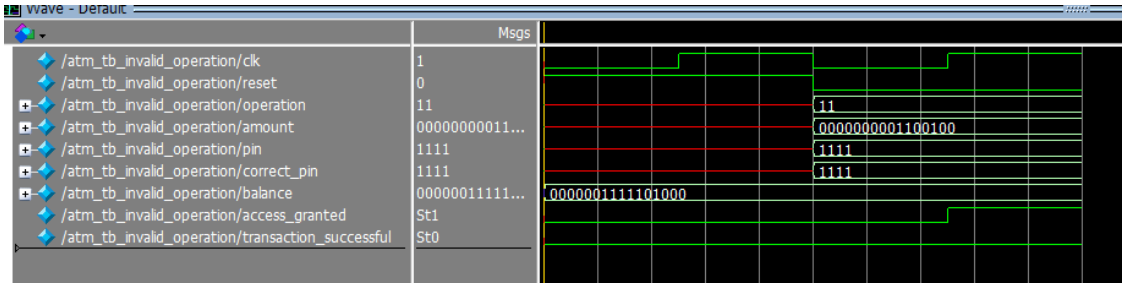
```
module atm_tb_invalid_operation;
    reg clk;
    reg reset;
    reg [1:0] operation;
```

```
    reg [15:0] amount;
    reg [3:0] pin;
    reg [3:0] correct_pin;
    wire [15:0] balance;
    wire access_granted;
    wire transaction_successful;
    atm uut (
        .clk(clk),
       .reset(reset),
        .operation(operation),
        .amount(amount),
        .pin(pin),
        .correct_pin(correct_pin),
        .balance(balance),
        .access_granted(access_granted),
        .transaction_successful(transaction_successful)
    );
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
    initial begin
        reset = 1;
        #10 reset = 0;
        pin = 4'b1111;
        correct_pin = 4'b1111;
        operation = 2'b11; // Invalid operation
        amount = 16'd100;
        #10;
        $display("Test Invalid Operation - Balance: %d, Transaction Successful: %b",
balance, transaction_successful);
```

```
        $finish;
    end
endmodule
```

Error: The testbench uses 2'b11 as the operation code, which is undefined in the atm module.

Simulation:



CHAPTER 6

CONCLUSION

This project successfully designed and implemented an Automated Teller Machine (ATM) using Verilog HDL. The ATM's functionality was modeled using a Finite State Machine (FSM), ensuring secure and efficient transactions. The Verilog code was simulated using ModelSim, verifying the correctness of the design. Synthesis on Xilinx Spartan-6 FPGA platform demonstrated optimal resource utilization.

6.1.Key Achievements:

1. Implemented ATM functionality using Verilog HDL.
2. Successfully simulated and verified the design.
3. Optimized resource utilization on FPGA platform.
4. Demonstrated secure transaction processing.
5. Validated FSM-based design approach.

6.2.Future Scope:

1. Integrating additional security features.
2. Expanding transaction types (e.g., mobile payments).
3. Implementing user authentication using biometrics.
4. Developing a graphical user interface (GUI).
5. Porting the design