



AIN SHAMS UNIVERSITY FACULTY OF ENGINEERING I-CREDIT
HOURS ENGINEERING PROGRAMS COMPUTER ENGINEERING AND
SOFTWARE SYSTEMS PROGRAM
2023-2024

ATM Verilog Project

ELECTRONIC DESIGN AUTOMATION
CSE 312

Name	ID
Ahmed Nezar Ahmed Hussien	21P0025
AbdulRahman Hesham Kamel Seleim Abdullah	21P0153
Kirollos Ehab Magdy Halim	21P0006
Farida Waleed Mohamed Mohamed Fakhry	21P0167
Yasmina Nasser Hamam Abdelfadeel	21P0211
Monica Hany Makram Derias	21P0173
Patrick Ramez Eskander Bolous	21P0180

Course Coordinator:

Dr. Eman El Mandoh & Eng. Kareem Waseem

Contents

1 Abstract	1
2 Introduction	1
3 Design	4
3.1 Modules	4
3.1.1 Definitions	4
3.1.2 ATM	5
3.1.3 Authenticator	7
3.1.4 Function	8
4 State Diagram	10
5 Verification	11
5.1 Test Bench	11
5.1.1 Directed Testing	11
5.1.2 Random Transaction Test	17
5.2 Assertions	18
5.2.1 Types of Assertion	18
5.3 Coverage Report	21
5.3.1 Coverage for ATM Module	21
6 Synthesis	24
6.1 ATM Module Schematic:	24
6.2 Authenticator Module Schematic:	25
7 Reference Model	26
7.1 Description	26
7.2 Comparing Output with Given Inputs	33

1 Abstract

The project focuses on ASIC design and verification of a bank ATM core, encompassing system architecture, high-level modeling, and Verilog implementation. Divided into Design and Verification teams, the former defines system components, interactions, and constraints, creating an abstract and platform-independent high-level model. The Reference Model ensures behavioral compliance. The Verification Team crafts a comprehensive testbench, self-checking scenarios, and PSL assertions. The Verilog implementation features a finite state machine, database handling, and an authenticator module. Extensive testing validates balance, withdrawal, deposit, and PIN change operations, emphasizing the practical application of Verilog in ASIC design. The collaborative effort ensures a reliable and well-tested ATM core.

2 Introduction

In this project, we focused on implementing the complete ASIC flow for an ATM system, comprising both the design and verification aspects. The goal is to simulate the core functionality of a bank ATM, including operations such as balance inquiry, withdrawal, deposit, and PIN change.

The project is divided into two teams, The Design Team responsible for system architecture, high-level modeling, and Verilog implementation, and the Verification Team in charge of creating a comprehensive testbench, self-checking test scenarios, and assertion-based verification using Property Specification Language (PSL).

- **Design Team Responsibilities:**

1. **System Architecture:**

- (a) **Detailed Specification:**

Created a detailed specification of the ATM system, defining its functional components, interfaces, and overall structure.

- (b) **Component Interaction:**

Defined how different modules and components interact to achieve the desired functionality.

- (c) **System Constraints:**

Considered system constraints and requirements to guide subsequent design decisions.

2. **High-Level Modeling:**

- (a) **Abstract Representation:**

Developed an abstract representation of the ATM system, focusing on high-level functionalities and interactions.

- (b) **Platform Independence:**

Ensured that the high-level model is platform-independent, allowing for potential future adaptations or changes.

3. Reference Model:

(a) **Behavioral Representation:**

Established a behavioral reference model that serves as a baseline for expected system behavior.

(b) **Specification Compliance:**

Ensured the reference model aligns with the project specifications, acting as a reference for verification.

– **Verilog Implementation:**

* **Module Design:**

Designed individual Verilog modules for distinct operations, such as balance inquiry, withdrawal, deposit, and PIN change.

* **State Machine Integration:**

Integrated a finite state machine to control the flow of operations, ensuring a structured and predictable execution.

* **File Operations:**

Implemented file operations to simulate a database, managing account information and persisting data between sessions.

• **Verification Team Responsibilities:**

1. **Testbench Creation:**

(a) **Comprehensive Scenario Coverage:**

Developed a testbench encompassing a wide range of scenarios, ensuring thorough coverage of functional and edge cases.

(b) **Directed and Random Tests:**

Designed both directed and constraint random tests to verify specific functionalities and explore unpredictable scenarios.

2. **Self-Checking Test Scenarios:**

(a) **Automated Verification:**

Utilized the reference model to automate the verification process, creating self-checking test scenarios that compare Verilog output against expected behavior.

(b) **Consistency Checks:**

Implemented consistency checks to validate that the Verilog implementation aligns with the behavioral reference model.

3. **PSL Assertions:**

(a) **Formal Verification:**

Formulated Property Specification Language (PSL) assertions to formally define desired system properties.

(b) **Dynamic Verification:**

Applied PSL assertions dynamically during simulation, ensuring that critical design properties hold true.

4. **Code Coverage:**

(a) **Analysis Scope:**

Enabled and conducted code coverage analysis to assess the extent to which the Verilog code is exercised by the testbench.

(b) **Targeted Improvement:**

Identified areas of the code with low coverage, facilitating targeted improvements to enhance test completeness.

- **Project Implementation:**

1. **Database Handling:**

Utilized text files to simulate account information, including PINs, account balances, and account numbers.

2. **Finite State Machine:**

Implemented a finite state machine to manage the different stages of ATM operation, ensuring a coherent flow between states.

3. **Authenticator Module:**

Developed a module responsible for authenticating user accounts based on account numbers and PINs, with an additional capability for PIN changes.

- **Testing and Results:**

1. Executed a series of tests to validate the functionality of the Verilog code and ensure that it meets the specified requirements.
2. Verified that balance inquiries, deposits, withdrawals, and PIN changes produce the expected outcomes.
3. Leveraged random testing to explore a broader range of scenarios and uncover potential edge cases.

This project demonstrates a practical application of Verilog in ASIC design, emphasizing the importance of a systematic approach encompassing both design and verification teams to ensure the reliability and correctness of the implemented system. The use of a finite state machine, database simulation, and PSL assertions contributes to a robust and well-tested ATM core design.

3 Design

3.1 Modules

3.1.1 Definitions

- **Constants and Enumerations:**

- **Boolean Constants:**

Table 1: Boolean Constants

Constant	Description
TRUE	Represents a logic high value (1).
FALSE	Represents a logic low value (0).

- **State Machine Definitions:**

Table 2: State Machine Definitions

State	Description
WAITING	3-bit binary representation of the WAITING state in the ATM state machine (3'b000).
AUTHENTICATION	3-bit binary representation of the AUTHENTICATION state (3'b001).
MENU	3-bit binary representation of the MENU state (3'b010).
BALANCE	3-bit binary representation of the BALANCE state (3'b011).
WITHDRAW	3-bit binary representation of the WITHDRAW state (3'b100).
DEPOSIT	3-bit binary representation of the DEPOSIT state (3'b101).
CHANGE_PIN	3-bit binary representation of the CHANGE_PIN state (3'b110).
IDLE	3-bit binary representation of the IDLE state (3'b111).

- **Authentication Definitions:**

Table 3: Authentication Definitions

Definition	Description
FIND_ACCOUNT	Binary representation (0) indicating the phase of finding the account in the Authenticator module.
AUTHENTICATE_ACCOUNT	Binary representation (1) indicating the phase of authenticating the account.

– **Language Definitions:**

Table 4: Language Definitions

Language	Description
ENGLISH	Binary representation (0) indicating the English language setting.
ARABIC	Binary representation (1) indicating the Arabic language setting.

• **Purpose:**

- **Enhances Code Readability:** By using named constants, the code becomes more readable and self-explanatory.
- **Centralized Definitions:** All project-wide constants and enumerations are consolidated in one module for easy access and modification.
- **Maintainability:** Changes to constant values can be made in a single location, affecting the entire project consistently.

• **Usage:**

These constants and enumerations are used throughout the project to specify and check states, control logic, and set language preferences. Provides a consistent and clear representation of various states and settings within the Verilog code.

3.1.2 ATM

Overview:

The ATM module serves as the core of the bank ATM design. It includes functionalities for handling different operations such as balance inquiry, withdrawal, deposit, and changing the PIN. The module interacts with an authenticator and employs a finite state machine to manage its operations.

Input	Description
clk	Clock input.
rst	Reset input.
operation	Operation code (balance, withdrawal, deposit, change PIN).
acc_num	Account number.
pin	Account PIN.
newPin	New PIN for PIN change.
amount	Transaction amount.
language	Language selection.

Table 5: Input Specifications

Output	Description
balance	Current account balance.
success	Operation success flag.
state	Current state of the state machine.

Table 6: Output Specifications

- **State Machine:**

State	Binary Representation	Description
WAITING	3'b000	Initial state. Waits for an account to be found (via acc_found_stat).
AUTHENTICATION	3'b001	Moves to this state when an account is found. Checks the authentication status (acc_auth_stat). If not authenticated (ACCOUNT_NOT_AUTHENTICATED), transitions back to WAITING. If authenticated (ACCOUNT_AUTHENTICATED), transitions to MENU.
MENU	3'b010	Presents the user with menu options based on the operation. Options: BALANCE, WITHDRAW, DEPOSIT, CHANGE_PIN. Transitions to the corresponding states based on the selected operation.
BALANCE	3'b011	Displays the account balance (showBalanceInfo task). Transitions back to WAITING after displaying the balance.
WITHDRAW	3'b100	Processes a withdrawal (withdrawAndUpdate task). Transitions back to WAITING after processing.
DEPOSIT	3'b101	Processes a deposit (Deposit_Money task). Transitions back to WAITING after processing.
CHANGE_PIN	3'b110	Initiates a PIN change process (changePinProcess task in Authenticator module). Transitions back to WAITING after processing.
IDLE	3'b111	No specific operation. Remains in this state until a reset or another operation.

Table 7: Finite State Machine States and Descriptions

- **State Transition Logic:**

- Transitions are determined by the current_state and next_state registers.
- Transitions occur on clock edges or reset events.
- The transition conditions are based on the values of acc_found_stat, acc_auth_stat, and the selected operation.

- **Operations:**

- **Initialization:** Reads account balances from a text file into balance_database.
- **State Machine:** Transitions through states (WAITING, AUTHENTICATION, MENU,...) based on inputs.
- **Operations:** Performs operations such as balance inquiry, withdrawal, deposit, and PIN change based on the state and input conditions. In order to enhance security and manage user interactions, the ATM system incorporates a timeout mechanism. A timeout counter is employed to monitor the duration between user operations. If no operation is entered within a predefined time frame, the system automatically times out, ensuring the security of the user's session and prompting the ATM to return to a default state or take appropriate measures as defined by the system design.
- **File Handling:** Writes updated account balances back to the text file.

- **File Operations:**

- The module reads and writes account balances to a text file for persistent storage.

- **PSL Assertions:**

- There are Property Specification Language (PSL) assertions for checking balance, deposit, and withdrawal operations.

3.1.3 Authenticator

Overview:

The Authenticator module authenticates user accounts using account numbers and PINs. It interfaces with a database and includes a PIN change functionality.

- **Inputs:**

Input	Description
acc_num	Account number.
pin	Account PIN.
newPin	New PIN for PIN change.

Table 8: Input Specifications

- **Outputs:**

Output	Description
acc_index_out	Index of the authenticated account.
acc_found_stat	Account found status.
acc_auth_stat	Account authentication status.

Table 9: Output Specifications

- **Operation:**

Operation	Description
Initialization	Reads account numbers and PINs from text files into acc_num_db and pin_db.
Account Finding	Searches for the account number in the database and sets acc_found_stat accordingly.
PIN Verification	Verifies the PIN for authentication and updates acc_auth_stat.
PIN Change	Changes the PIN if the new PIN is valid.

Table 10: Operation Specifications

3.1.4 Function

- **Task Details:**

1. **Deposit_Money Task:**

Input	Description
amount	The amount to be deposited.
currentBalance	The current account balance.

Table 11: Inputs for Deposit Operation

- **Outputs:**

Output	Description
newBalance	The updated account balance after deposit.
success	Indicates the success or failure of the deposit operation.

Table 12: Outputs for Deposit Operation

- **Operation:**

- * Adds the provided amount to the current account balance.
- * Calculates the newBalance as currentBalance + amount.
- * Sets success to 1 to indicate a successful deposit.
- * Displays a message indicating the success of the deposit operation.

2. **showBalanceInfo Task:**

Input	Description
balance	The current account balance.

Table 13: Inputs for the Operation

Output	Description
success	Indicates the success or failure of displaying the balance.

Table 14: Outputs for the Operation

- **Operation:**

- * Displays the current account balance using \$display.
- * Sets success to 1 to indicate the successful display of the balance.
- * Note: The actual display may vary based on the simulation environment.

3. **withdrawAndUpdate Task:**

Input	Description
<code>amount</code>	The amount to be withdrawn.
<code>currentBalance</code>	The current account balance.

Table 15: Inputs for Withdrawal Operation

Output	Description
<code>newBalance</code>	The updated account balance after withdrawal (if successful).
<code>success</code>	Indicates the success or failure of the withdrawal operation.

Table 16: Outputs for Withdrawal Operation

– **Operation:**

- * Checks if the withdrawal `amount` is feasible (i.e., does not exceed the current balance).
- * If feasible:
 - Calculates the `newBalance` as `currentBalance - amount`.
 - Sets `success` to 1 to indicate a successful withdrawal.
 - Displays a message indicating the success of the withdrawal operation.
- * If not feasible:
 - Leaves the `newBalance` unchanged.
 - Sets `success` to 0 to indicate a failed withdrawal.
 - Displays a message indicating the insufficiency of funds.

4 State Diagram

State	State Name
S0	Waiting
S1	Auth
S2	Menu
S3	Balance
S4	Withdraw
S5	Deposit
S6	Change Pin
S7	Idle



5 Verification

5.1 Test Bench

5.1.1 Directed Testing

- **System Reset Test**

- **Scenario:** Verify the system's behavior upon reset.
- **Code Snippet:**

```
1 // System reset scenario
2 rst = 0; @(negedge clk);
3 if (state != 7) $display("Test_Failed_for_System_Reset");
```

- **Description:**

The System Reset Test in the ATM system's directed testing is designed to verify the system's behavior following a reset. This scenario involves setting the 'rst' signal to 0 and observing the system's response. The expectation is that the system will enter its default state (represented by state 7 in this context). The test checks if this transition happens correctly upon reset. If the system does not transition to the default state as expected, the test bench outputs a failure message. This test ensures the ATM system can return to a known, stable state after a reset, a crucial aspect for maintaining system reliability and integrity.

- **Balance Inquiry Test**

- **Scenario:** Verify if the ATM correctly displays the balance.

- **Code Snippet:**

```

1  for (i = 0; i < 10 ; i = i +1 ) begin
2      rst = 1; operation = 3;
3      acc_num = acc_num_db[i];
4      pin = pin_db[i];
5      amount = 0;
6      language = 0;
7      Newpin = 0;
8      repeat(4) @(negedge clk);
9      if (balance != balance_database[i]) begin
10         $display("Test_Failed");
11     end
12 end

```

- **Wave Form:**

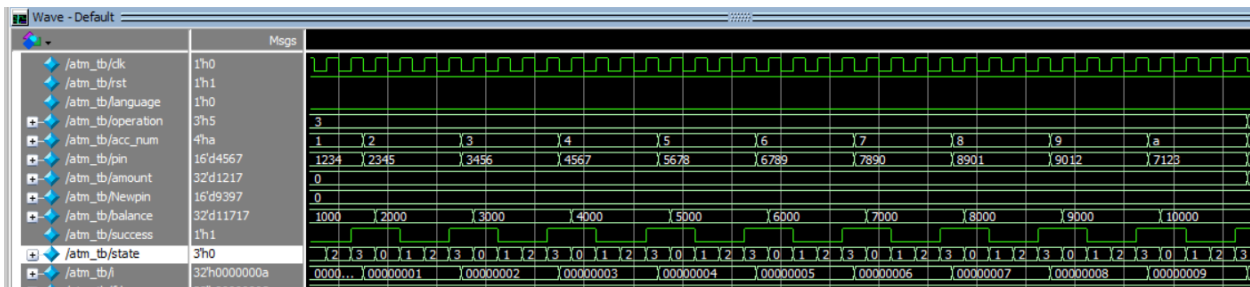


Figure 1: Balance Inquiry Waveform

- **Description:**

The Verilog code snippet is a testbench featuring a loop that iterates ten times. Within each iteration, variables are initialized, clock synchronization is implemented, and a conditional check occurs. The if condition is triggered when the calculated balance ('balance') differs from the expected balance stored in the 'balance_database' array for the current iteration index ('i'). If the condition is met, a "Test Failed" message is displayed, indicating a potential discrepancy between the simulated and expected values during the corresponding iteration of the testbench.

- **Deposit Operation Test**

- **Scenario:** Test the deposit functionality.
- **Code Snippet:**

```

1  // deposit 1000 for all accounts
2      for (i = 0; i < 10 ; i = i +1 ) begin
3          rst = 1; operation = 5;
4          acc_num = acc_num_db[i];
5          pin = pin_db[i];
6          amount = 1000;
7          language = 0;
8          Newpin = 0;
9          repeat(4) @(negedge clk);
10         if (balance != balance_database[i] + 1000) begin
11             $display("Test_Failed");
12         end
13     end

```

- **Wave Form:**

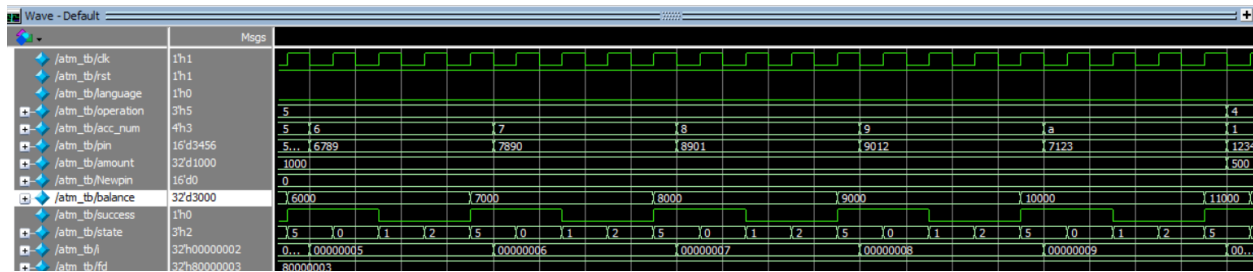


Figure 2: Deposit Waveform

- **Description:**

The for loop initializes or resets variables for each iteration, such as the reset signal ('rst'), operation code ('operation'), account number ('acc_num'), PIN ('pin'), deposit amount ('amount'), language preference ('language'), and a new PIN ('Newpin'). The deposit amount is set to 1000 for each iteration. After synchronizing with the clock signal, the if condition is invoked, checking if the calculated balance ('balance') matches the expected balance plus 1000 units, as per the 'balance_database' array for the current account index ('i'). If the condition is not satisfied, the "Test Failed" message is displayed, indicating a potential issue with the deposit operation for the corresponding account during that cycle of the testbench.

- **Withdrawal Operation Test**

- **Scenario:** Check the withdrawal process.

- **Code Snippet:**

```

1  // Withdraw more than balance
2      for (i = 0; i < 10 ; i = i +1 ) begin
3          rst = 1; operation = 4;
4          acc_num = acc_num_db[i];
5          pin = pin_db[i];
6          amount = balance_database[i] + 100;
7          language = 0;
8          Newpin = 0;
9          repeat(4) @(negedge clk);
10         end

```

- **Wave Form:**

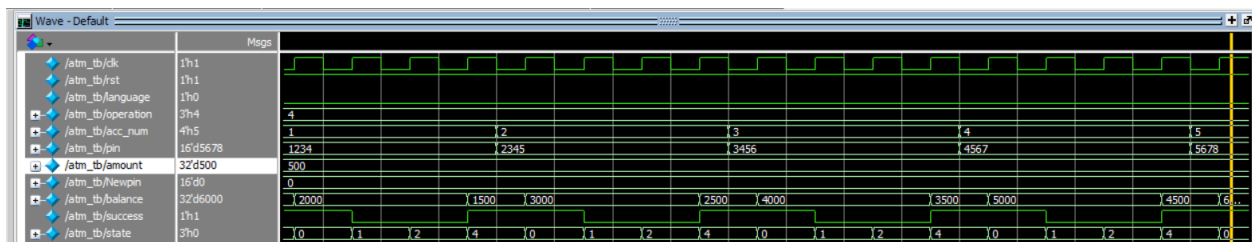


Figure 3: Withdraw Waveform

- **Description:**

The for loop iterates ten times, with each iteration initializing or resetting variables such as the reset signal ('rst'), operation code ('operation'), account number ('acc_num'), PIN ('pin'), withdrawal amount ('amount'), language preference ('language'), and a new PIN ('Newpin'). The withdrawal amount is set to the current balance plus 100 units for each iteration. After synchronizing with the clock signal, the loop completes.

The code doesn't explicitly check for the success or failure of the withdrawal operation. It seems to be setting up a scenario where withdrawal attempts are made for each account with an amount exceeding the current balance. If a failure condition check is required, additional logic would be needed, such as comparing the calculated balance after the withdrawal operation with an expected balance or checking for a specific response from the system.

- **Change PIN Test**

- **Scenario:** Validate the PIN change operation.

- **Code Snippet:**

```

1      // Change pin but having the same pin
2      for ( i = 0; i < 10 ; i = i + 1 ) begin
3          rst = 1;
4          operation = 6;
5          acc_num = acc_num_db[i];
6          pin = pin_db[i];
7          amount = 0;
8          language = 0;
9          Newpin = pin_db[i];
10         repeat(4) @(negedge clk);
11     end

```

- **Wave Form:**

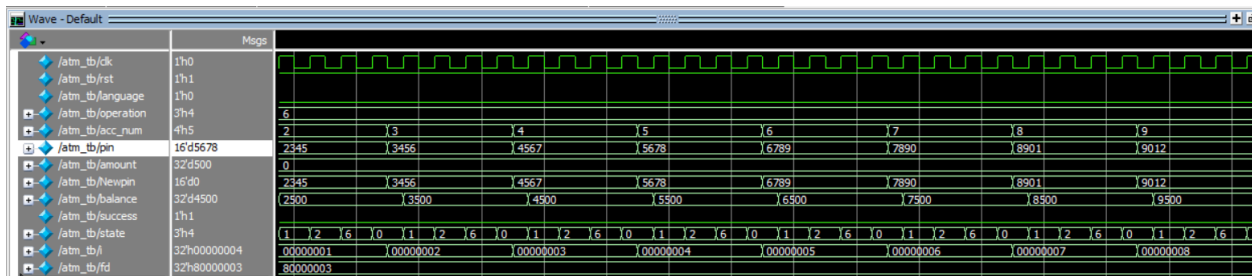


Figure 4: Change pin Waveform

- **Description:**

The for loop iterates ten times, and within each iteration, various variables are initialized or reset, including the reset signal (rst), operation code (operation), account number (acc_num), current PIN (pin), withdrawal amount (amount), language preference (language), and a new PIN (Newpin). The withdrawal amount is set to 0, indicating a non-monetary transaction. Notably, the new PIN (Newpin) is set to the existing PIN for each account. After synchronizing with the clock signal through the repeat(4) @(negedge clk); statement, the loop completes.

The code simulates a scenario where users attempt to update their PINs to their existing values, and it does not explicitly check for the success or failure of this operation. If checking for a specific response or outcome is needed, additional logic would be required in the testbench.

- **Invalid PIN Test**

- **Scenario:** Test system response to an incorrect PIN.

- **Code Snippet:**

```

1  // wrong pin
2      for (i = 0; i < 10 ; i = i +1 ) begin
3          rst = 1;
4          operation = 3;
5          acc_num = acc_num_db[i];
6          pin = pin_db[9-i];
7          amount = 0;
8          language = 0;
9          Newpin = 0;
10         repeat(4) @(negedge clk);
11         if (success != 0) begin
12             $display("Test_Failed");
13         end
14     end

```

- **Wave Form:**

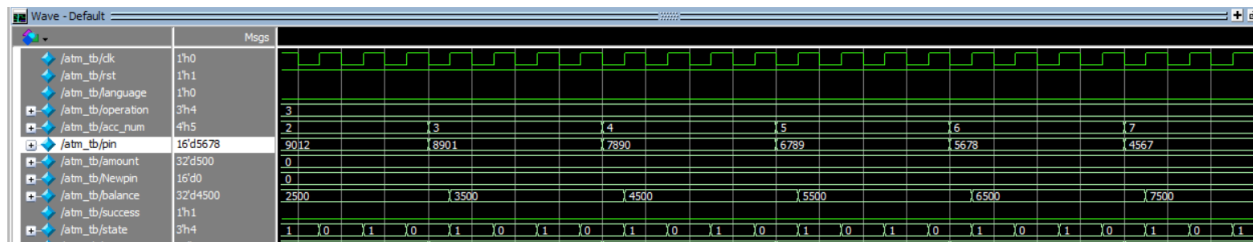


Figure 5: Wrong pin Waveform

- **Description:**

The for loop iterates ten times, and within each iteration, various variables are initialized or reset, including the reset signal (rst), operation code (operation), account number (acc_num), PIN (pin), withdrawal amount (amount), language preference (language), and a new PIN (Newpin). The PIN for each account is set to the reverse of the PIN in pin_db. After synchronizing with the clock signal through the repeat(4) @(negedge clk); statement, the loop checks if the variable success is not equal to 0. If the condition is true, a "Test Failed" message is displayed. This code simulates a scenario where users attempt to access their accounts with incorrect PINs, and it checks for the success or failure of the access operation. If success is not 0, it indicates a failure, and the "Test Failed" message is displayed. If different criteria or response checking is needed, additional logic would be required in the testbench.

5.1.2 Random Transaction Test

- **Scenario:** Assess the system's handling of randomized transactions.
- **Code Snippet:**

```

1      // random testing
2      for (i = 0 ; i < 10 ; i ++ ) begin
3          rst = 1; operation = $urandom_range(3,6);
4          acc_num = i+1;
5          pin = pin_random[acc_num-1];
6          amount = $urandom_range(0,10000);
7          language = $urandom_range(0,1);
8          Newpin = $urandom_range(1000,9999);
9          repeat(4)@(negedge clk);
10     end

```

- **Wave Form:**

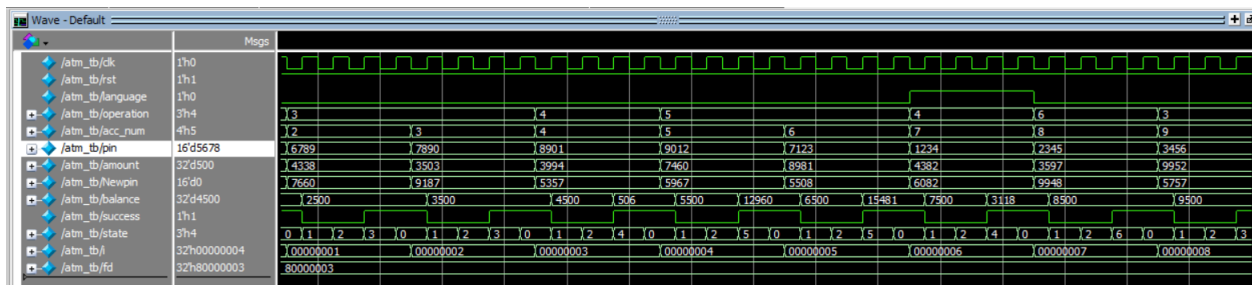


Figure 6: Randomized Test Waveform

- **Description:**

The for loop iterates ten times, and within each iteration, various variables are initialized or reset using random values. The reset signal (rst) is set to 1 for each iteration. The operation variable is assigned a random value between 3 and 6 using \$urandom_range. The acc_num is set to the loop index (i+1), and the pin is assigned a random value from the pin_random array based on the account number. The amount is set to a random value between 0 and 10000, the language is set to a random value of 0 or 1, and the Newpin is assigned a random value between 1000 and 9999. After initializing the variables, the code synchronizes with the clock signal through the repeat(4)@(negedge clk); statement, and the loop continues.

This code simulates a scenario where random transactions are performed on ten accounts, with various parameters such as operation, account number, PIN, amount, language preference, and new PIN being randomly generated. The purpose seems to be testing the system's response to diverse transaction scenarios. If specific criteria or response checking is needed, additional logic would be required in the testbench.

5.2 Assertions

5.2.1 Types of Assertion

1. Reset Assertions:

- **Code Snippet:**

```
1 psl rst_assert: assert always((rst == 0) -> next (state==7)) @(
    posedge clk);
```

- **Description:**

The property `rst_assert` asserts that under the condition where the reset signal (`rst`) is equal to 0 (i.e., asserted), it should lead to the next state (`state`) being equal to 7. This property is expressed using the `assert always` syntax, indicating that this condition should hold for all possible time instances. The implication operator `->` signifies that the subsequent part of the property (`next (state == 7)`) should hold when the antecedent (`rst == 0`) is true. The `@(posedge clk)` at the end indicates that this assertion is triggered on every positive edge of the clock signal (`clk`).

In simpler terms, this property asserts that whenever the reset signal is asserted, it should result in the system transitioning to state 7 in the next clock cycle. This kind of property is often used in formal verification processes to ensure specific behaviors or conditions in a digital design.

2. Balance Assertions:

- **Code Snippet:**

```
1 psl show_balance: assert always
2 ((state == 2 && operation == 3) -> next
3 (balance == balance_database[prev(acc_index)])) abort !rst)
4 @(posedge clk);
```

- **Description:**

The property asserts that under the condition where the current state (`state`) is equal to 2 and the operation code (`operation`) is equal to 3 (indicating a balance display operation), it should lead to the next state (`state`) having a balance equal to the balance stored in the `balance_database` array corresponding to the previous account index (`prev(acc_index)`). This property is expressed using the `assert always` syntax, signifying that this condition should hold for all possible time instances. The implication operator `->` indicates that the subsequent part of the property (`next(balance == balance_database[prev(acc_index)])`) should hold when the antecedent (`state == 2 && operation == 3`) is true. The `@(posedge clk)` at the end specifies that this assertion is triggered on every positive edge of the clock signal (`clk`).

In simpler terms, this property ensures that when the system is in the state of displaying the balance (`state == 2`) and the operation is a balance display (`operation == 3`), the next state should have a balance equal to the balance stored in the database for the account index preceding the current account index. This is a common kind of property used to check correctness in digital design verification.

3. Deposit Assertions:

- **Code Snippet:**

```
1 psl deposit: assert always((state == 2 && operation ==5)
2 -> next (balance == (prev(balance) + prev(amount))) abort !rst) @(
    posedge clk);
```

- **Description:**

The property asserts that under the condition where the current state (state) is equal to 2 and the operation code (operation) is equal to 5 (indicating a deposit operation), it should lead to the next state (state) having a balance equal to the previous balance plus the previous deposit amount (prev(balance) + prev(amount)). This property is expressed using the assert always syntax, indicating that this condition should hold for all possible time instances. The implication operator -> signifies that the subsequent part of the property (next(balance == (prev(balance) + prev(amount)))) should hold when the antecedent (state == 2 && operation == 5) is true. The @(posedge clk) at the end specifies that this assertion is triggered on every positive edge of the clock signal (clk).

In simpler terms, this property ensures that when the system is in the state of processing a deposit (state == 2) and the operation is a deposit operation (operation == 5), the next state should have a balance equal to the sum of the previous balance and the amount deposited in the preceding state. This property is a formal way to verify correctness in the behavior of a digital design related to deposit operations.

4. Withdraw Assertions:

- **Code Snippet:**

```
1 psl withdraw: assert always((state == 2 && operation ==4 && (amount
    <= balance)) -> next
2 (balance == (prev(balance) - prev(amount))) abort !rst)
3 @(posedge clk);
```

- **Description:**

The property asserts that under the condition where the current state ('state') is equal to 2, the operation code ('operation') is equal to 4 (indicating a withdrawal operation), and the withdrawal amount ('amount') is less than or equal to the current balance ('amount <= balance'), it should lead to the next state ('state') having a balance equal to the previous balance minus the previous withdrawal amount ('prev(balance) - prev(amount)'). This property is expressed using the 'assert always' syntax, signifying that this condition should hold for all possible time instances. The implication operator '->' indicates that the subsequent part of the property ('next(balance == (prev(balance) - prev(amount)))') should hold when the antecedent ('state == 2 && operation == 4 && (amount <= balance)') is true. The '@(posedge clk)' at the end specifies that this assertion is triggered on every positive edge of the clock signal ('clk').

In simpler terms, this property ensures that when the system is in the state of processing a withdrawal ('state == 2'), the operation is a withdrawal operation ('operation == 4'), and the withdrawal amount is less than or equal to the current balance, the next state should have a balance equal to the previous balance minus the amount withdrawn in the preceding state. This property is a formal way to verify correctness in the behavior of a digital design related to withdrawal operations.

5. Withdraw insufficient funds Assertions:

- **Code Snippet:**

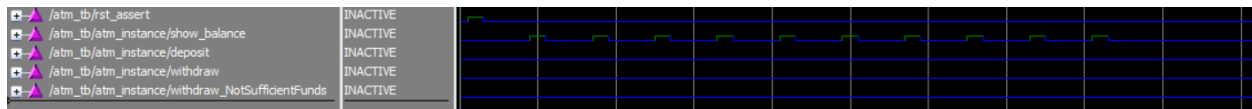
```
1 psl withdraw_NotSufficientFunds: assert always
2 ((state == 2 && operation ==6 && (amount>balance) )
3 -> next (balance == prev(balance)) abort !rst) @(posedge clk);
```

- **Description:**

The property asserts that under the condition where the current state ('state') is equal to 2, the operation code ('operation') is equal to 6 (indicating a withdrawal operation), and the withdrawal amount ('amount') is greater than the current balance ('amount > balance'), it should lead to the next state ('state') having a balance equal to the previous balance ('prev(balance)'). This property is expressed using the 'assert always' syntax, signifying that this condition should hold for all possible time instances. The implication operator '->' indicates that the subsequent part of the property ('next(balance == prev(balance))') should hold when the antecedent ('state == 2 && operation == 6 && (amount > balance)') is true. The '@(posedge clk)' at the end specifies that this assertion is triggered on every positive edge of the clock signal ('clk').

In simpler terms, this property ensures that when the system is in the state of processing a withdrawal ('state == 2'), the operation is a withdrawal operation with insufficient funds ('operation == 6 && amount > balance'), the next state should have a balance equal to the previous balance. This property is a formal way to verify correctness in the behavior of a digital design related to withdrawal operations with insufficient funds.

6. Assertion Waveform:



5.3 Coverage Report

5.3.1 Coverage for ATM Module

1. Branch Coverage

Branches Coverage (93.33%)			
Search...			
Source	Line Number	Coverage	
if (counter >= 4) begin	45	100%	
if (trst) begin	52	100%	
case (current_state)	63	88.88%	
if (acc_found_stat == 'TRUE') begin	65	100%	
if (acc_auth_stat == 'ACCOUNT_NOT_AUTHENTICATED') begin	74	100%	
if (operation == 'BALANCE') begin	85	100%	
if (authenticatedFlag == 'TRUE') begin	102	100%	
if (authenticatedFlag == 'TRUE') begin	109	100%	
if (authenticatedFlag == 'TRUE') begin	116	100%	
if (authenticatedFlag == 'TRUE') begin	123	50%	

Figure 7: Branch Coverage

2. Condition Coverage

Conditions Coverage (100%)			
Search...			
Conditions	Line Number	Coverage	
{counter >= 4}	45	100%	
{operation == 3}	85	100%	
{operation == 4}	88	100%	
{operation == 5}	91	100%	
{operation == 6}	94	100%	

Figure 8: Condition Coverage

3. Statement Coverage

Statements Coverage (98.03%)			
Page Size: 50			
File Name	Line Number	Item	Hits
Search...	Search...	Search...	Search...
Verilog_Code/atm.v			

Figure 9: Statement Coverage

4. Assertion Coverage

Assertions Coverage (100%)		
Search...		
Assertions ↑	Failure Count	Pass Count
Search...	Search...	Search...
\\atm_tb#atm_instance /deposit	0	1
\\atm_tb#atm_instance /show_balance	0	1
\\atm_tb#atm_instance /withdraw	0	1
\\atm_tb#atm_instance /withdraw_NotSufficientFunds	0	1

5. Toggle Coverage

Toggles Coverage (59.09%)				
Page Size: 25				
Search...				
SCALAR Signal / Value	Hits		Coverage	
Search...	↑->1	->0	Search...	Search...
acc_auth_stat	1	1	100%	
acc_found_stat	1	1	100%	
acc_index[0-3]	1	1	100%	
acc_num[0-3]	1	1	100%	
amount[0-13]	1	1	100%	
authenticatedFlag	1	1	100%	
balance[13-0]	1	1	100%	
clk	1	1	100%	
counter[1-0]	1	1	100%	
current_state[2-0]	1	1	100%	
language	1	1	100%	
newPin[0-13]	1	1	100%	
next_state[2-0]	1	1	100%	
operation[0-2]	1	1	100%	
pin[0-13]	1	1	100%	

Figure 10: Toggle Hits

Toggles Coverage (59.09%)				
Page Size: 25				
Search...				
SCALAR Signal / Value	Hits		Coverage	
Search...	↑->1	->0	Search...	Search...
amount[14-15]	0	0	0%	
balance[31-14]	0	0	0%	
counter[3-2]	0	0	0%	
fd[31-0]	0	0	0%	
i[3-0]	0	0	0%	

Figure 11: Toggle Misses

- **Rationale for Toggle Total Misses:**

amount[14-15] : The largest binary number that can be expressed with 15 bits, is 111 1111 1111 1111, However, it is impractical for this value to be deposited or withdrawn at an ATM.

fd[31-0] / i : 'fd' and 'i' are both integers. 'fd' is utilized to represent file descriptors, while 'i' serves as a counter specifically designed for iterating through 'for' loops.

6. FSM Coverage

```
250 # FSM Coverage:
251 #   Enabled Coverage      Active   Hits   Misses % Covered
252 #   -----
253 #   FSMs                  0       0     0    100.0
254 #   States                0       0     0    100.0
255 #   Transitions           0       0     0    100.0
```

Figure 12: FSM Coverage

7. Overall Coverage

Design Units Coverage Summary (90.52%)					
Coverage Type ↑	Bins	Hits	Misses	Coverage	
Search...	Search...	Search...	Search...	Search...	Search...
Assertions	4	4	0	100%	
Branches	38	36	2	94.73%	
Conditions	9	9	0	100%	
Statements	85	84	1	98.82%	
Toggles	442	261	181	59.04%	

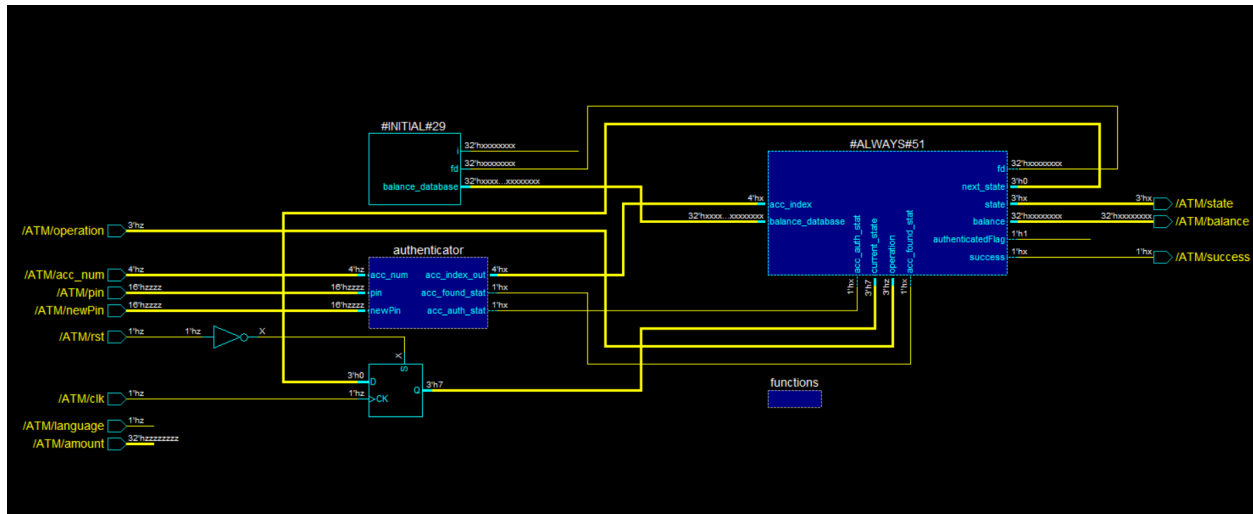
Figure 13: ATM Coverage Summary

Instance Coverage Summary (89.13%)				
Coverage Type ↑	Hits	Misses	Coverage	
Search...	Search...	Search...	Search...	Search...
Assertions	4	0	100%	
Branches	36	2	94.73%	
Conditions	9	0	100%	
Statements	84	1	98.82%	
Toggles	197	181	52.11%	

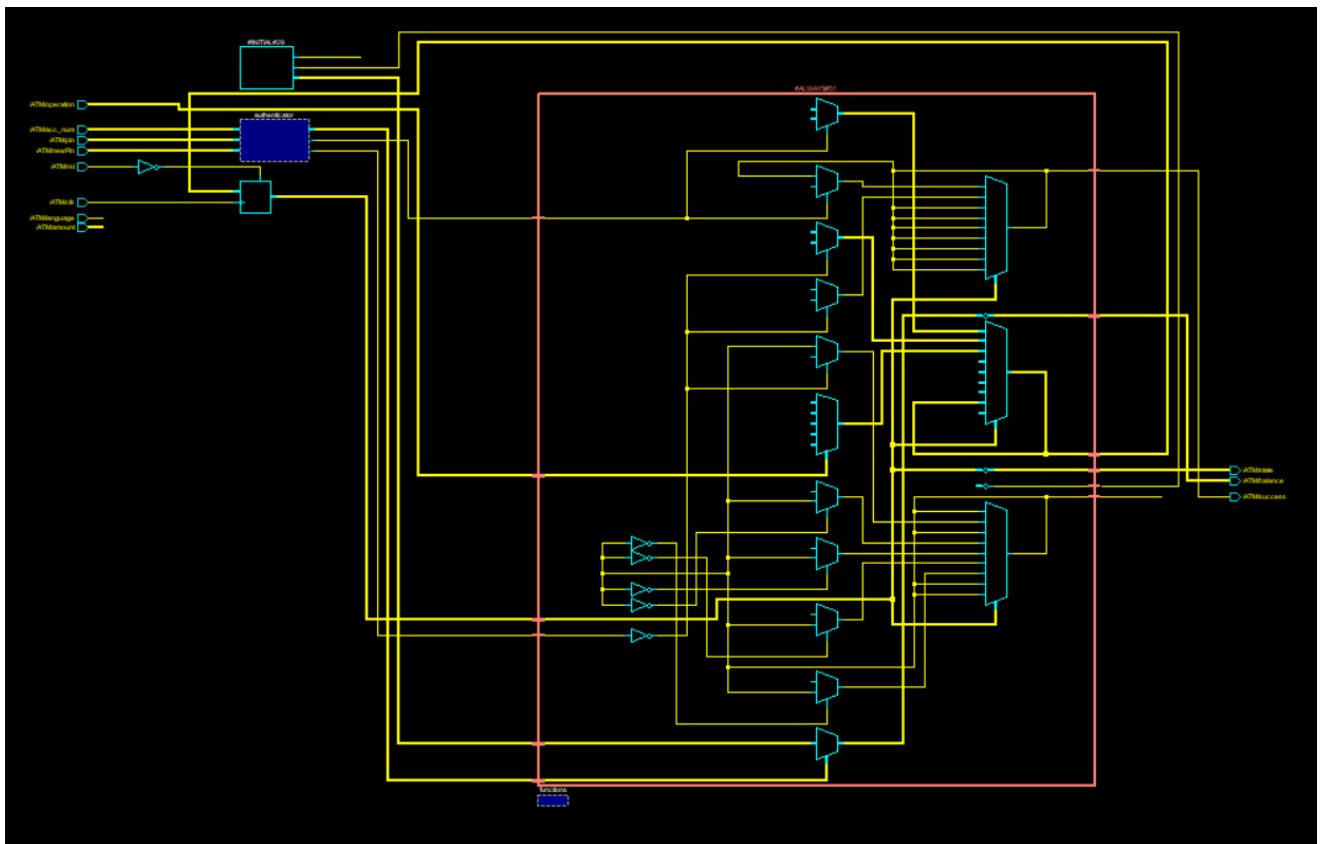
Figure 14: Authenticator Coverage Summary

6 Synthesis

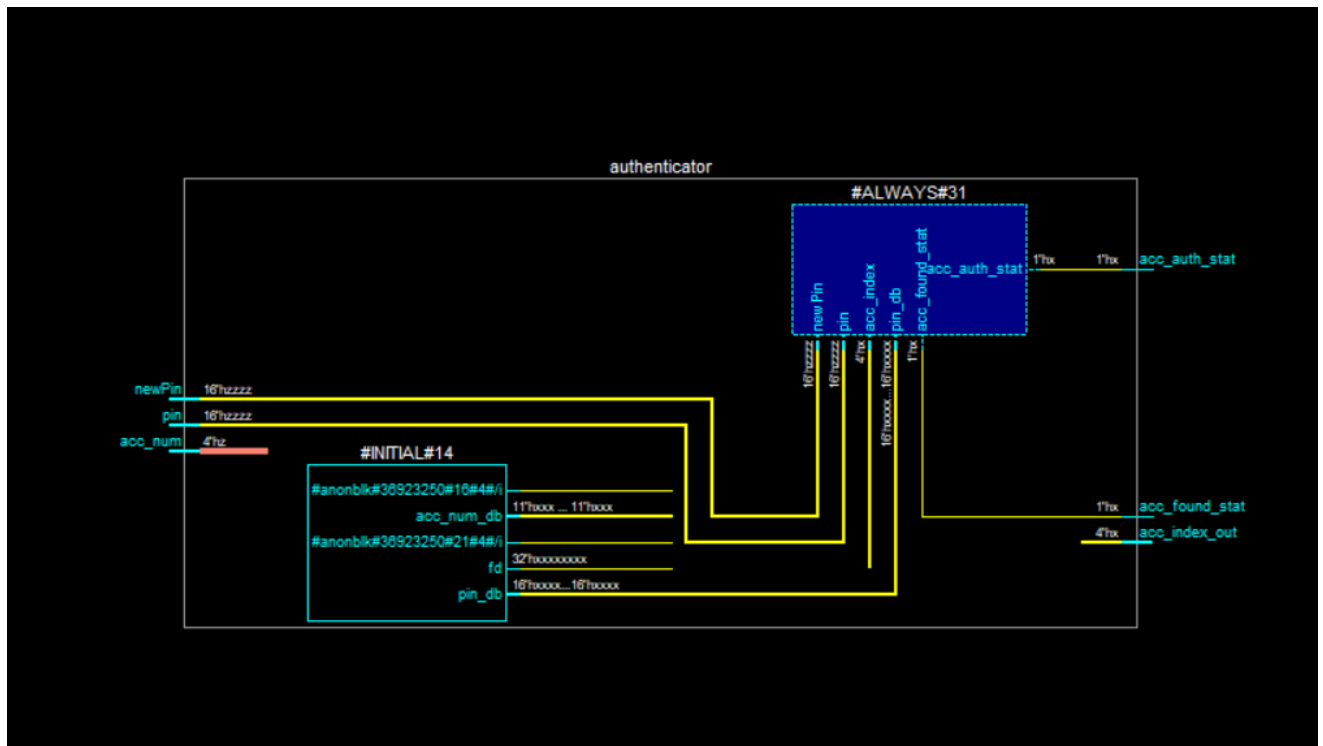
6.1 ATM Module Schematic:



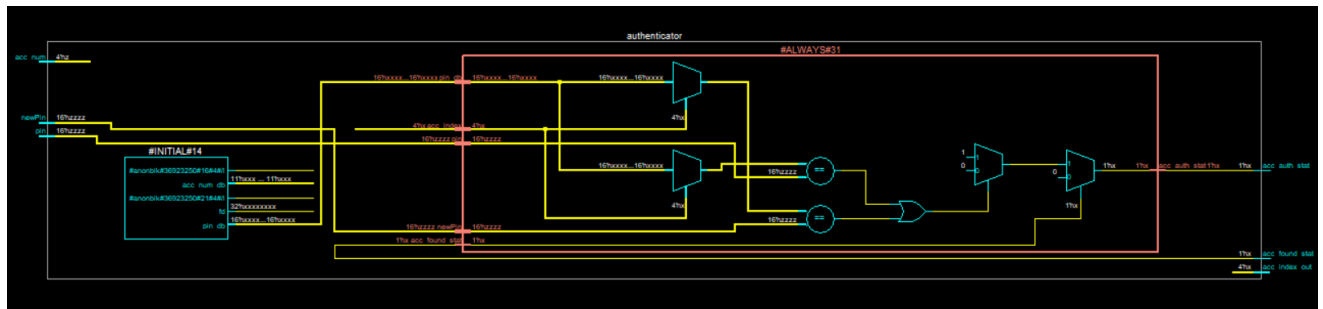
Detailed View:



6.2 Authenticator Module Schematic:



Detailed View:



7 Reference Model

7.1 Description

A reference model is a conceptual framework or template that provides a common understanding, structure, and set of guidelines for a particular domain or application. It serves as a reference point to facilitate communication, analysis, and the development of systems or solutions within that domain.

The reference model utilized in this ATM project primarily consists of a Python-based simulation acting as a benchmark for the Verilog hardware description. The Python script serves as a comprehensive simulation environment emulating the ATM's functionality. Through this Python model, user interactions, account operations, and system responses are simulated. Upon receiving user inputs such as account numbers and PINs, the Python script validates these details against a pre-existing dataset, providing a basis for comparison and validation.

Conversely, the Verilog implementation represents the hardware logic of the ATM system. The Verilog code encompasses modules for authentication, state management, and operational functionalities. It employs finite state machines to control and execute ATM operations based on inputs received. The Verilog code interacts with a balance database, managing account balances and processing user requests.

While the Verilog code forms the core hardware description, the Python script acts as a crucial reference model. It facilitates the validation and benchmarking of the Verilog implementation. By comparing the outputs and behaviors of the Verilog hardware with the results obtained from the Python simulation, the team ensures coherence and accuracy in the ATM system's functionalities across both software and hardware domains. And this is done via :

1. **generate_data:** This function reads data from a CSV file named "data.csv" using `pd.read_csv` ("Reference_CodePython\data.csv"). and then returns a DataFrame containing account information, including account numbers, PINs, and balances.
2. **Writing Inputs and Outputs to CSV Files:** The main program includes a loop that captures inputs (account number, amount, operation, pin, new pin, language) in a dictionary named `inputs`. After the user finishes operations, it creates or appends to CSV files ("inputs.csv" and "outputs.csv") using Pandas DataFrames. Inputs are stored in "inputs.csv," and the outputs (new PIN, amount, operation, success) are stored in "outputs.csv." This allows for easy tracking and analysis of user interactions and system responses.

Main:-

```
1 if __name__ == "__main__":
2     data = generate_data()
3     print("Welcome to ATM")
4
5     while True:
6         print("Enter your card")
7         account_number = int(input("Enter your account number: "))
8         pin = int(input("Enter your pin: "))
9         inputs["Pins"].append(pin)
10        inputs["Account"].append(account_number)
11        if card_handling(account_number, pin):
12            print("Card accepted")
13            operations()
14            target_size = len(inputs["Account"])
15            inputs["Amount"] += [0] * (target_size - len(inputs["Amount"]))
16            inputs["Newpin"] += [0] * (target_size - len(inputs["Newpin"]))
17            inputs["Operations"] += [0] * (target_size - len(inputs["Operations"]))
18            data.to_csv("Reference_Code\\Python\\outputs.csv", header=False, index=False)
19            pd.DataFrame(inputs).to_csv("Reference_Code\\Python\\inputs.csv", header=False, index=False)
20        else:
21            print("Card not accepted")
22            print("Please try again")
23            time.sleep(3)
24            os.system("cls")
```

- The provided code constitutes a simulated ATM program.
- It initializes data through a function and prompts users in a perpetual loop to input their account number and PIN.
- If the entered card details are accepted, the program acknowledges the acceptance, performs operations through an unspecified function, and updates data and input records in CSV files.
- In case of card rejection, users are prompted to retry after a brief delay. It is essential to ensure the presence of required imports and the necessary implementation of functions like `generate_data()` and `operations()` for the program's proper functionality.

Functions:

1. Card Handling:-

```
1 def card_handling(account_number, pin):
2     card_accepted = False
3     if account_number in data["Account_Number"].values:
4         if pin in data[data["Account_Number"] == account_number]["Pin"].values:
5             card_accepted = True
6     return card_accepted
```

- The card_handling function validates an account number and PIN against a dataset named data.
- It checks the existence of the specified account_number in the "Account Number" column.
- If the account number exists, it confirms the presence of the provided pin in the corresponding "Pin" column.
- If both conditions are true, card_accepted is set to True, signaling a valid combination. The function returns this value. Assumes a dataset with columns "Account Number" and "Pin," designed for banking or authentication systems.

2. Choose Language:-

```
1 def choose_language():
2     print("Choose your language:")
3     print("1. English")
4     print("2. Arabic")
5     language_choice = input("Enter your choice (1 or 2): ")
6     inputs["Language"].append(language_choice)
7     if language_choice == "1":
8         return "English"
9     elif language_choice == "2":
10        return "Arabic"
11    else:
12        print("Invalid language choice. Defaulting to English.")
13        return "English"
```

- The choose_language function prompts users to choose between English and Arabic.
- It presents a menu with numeric choices and stores the user's selection in inputs["Language"].
- The function returns "English" for option 1 and "Arabic" for option 2 after evaluating the input.
- In the case of an invalid choice, it defaults to English, notifying the user. Suitable for language selection in a program, ensuring proper recording of the user's language preference.

3. Operation:-

```
1 def operations():
2     language = choose_language()
3     if language == "English":
4         instructions = {
5             "1": "Change PIN",
6             "2": "Withdraw",
7             "3": "Deposit",
8             "4": "Balance Enquiry",
9             "5": "Exit"
10        }
11    else:
12    while True:
13        print("\nChoose an option:")
14        for key, value in instructions.items():
15            print(f"{key}. {value}")
16        choice = input(f"Enter your choice ({language}): ")
17        if choice == "1":
18            inputs["Operations"].append(6)
19            change_pin(account_number)
20        elif choice == "2":
21            inputs["Operations"].append(4)
22            withdraw(account_number)
23            exit()
24            break
25        elif choice == "3":
26            inputs["Operations"].append(5)
27            deposit(account_number)
28        elif choice == "4":
29            inputs["Operations"].append(3)
30            balance_enquiry(account_number)
31        elif choice == "5":
32            exit()
33            break
34        else:
35            print("Invalid option. Please try again.")
```

- The operations function manages main ATM operations based on the selected language.
- It displays a menu with options for changing the PIN, withdrawing, depositing, checking the balance, and exiting.
- Using a perpetual loop, it prompts users to choose an option, appending an operation code to inputs["Operations"].
- The function calls corresponding operations (e.g., change_pin, withdraw, deposit, balance_enquiry) based on the user's choice. Note: Assumes existence.

4. Withdraw:-

```
1 def withdraw(account_number):
2     amount = float(input("Enter the amount to withdraw: "))
3     index = data[data["Account Number"] == account_number].index[0]
4     balance = data.at[index, "Balance"]
5     inputs["Amount"].append(amount)
6
7     if amount <= 0:
8         print("Invalid amount. Please enter a positive value.")
9     elif amount > balance:
10        print("Insufficient balance. Your current balance is:",
11              balance)
12    else:
13        data.at[index, "Balance"] -= amount
14        print("Withdrawal successful. Your new balance is:", data.
15              at[index, "Balance"])
```

- The code defines a withdraw function for user withdrawals based on the provided account_number.
- It prompts for a withdrawal amount, updates the account balance from data, and handles scenarios like invalid amounts and insufficient funds.
- The code assumes a data dataset and uses inputs to track user inputs, necessitating integration into a broader system with proper dataset management.

5. Deposit:-

```
1 def deposit(account_number):
2     amount = float(input("Enter the amount to deposit: "))
3     inputs["Amount"].append(amount)
4     index = data[data["Account Number"] == account_number].index[0]
5     data.at[index, "Balance"] += amount
6     print(f"Deposit successful. Your new balance is: {data.at[index, '
    Balance']})")
```

- The code defines a deposit function allowing users to deposit into an account using the provided account_number.
- It prompts for a deposit amount, appends it to inputs["Amount"], and retrieves the account index in the data dataset.
- The function updates the account balance in the dataset by adding the deposited amount, accompanied by a success message displaying the updated balance.
- Assumptions include the existence of a data dataset with columns like "Account Number" and "Balance," and a global variable named inputs for tracking user inputs in a broader system.

6. Balance enquiry:-

```
1 def balance_enquiry(account_number):
2     index = data[data["Account Number"] == account_number].index[0]
3     balance = data.at[index, "Balance"]
4     print(f"Your current balance is: {balance}")
```

- The code defines a balance_enquiry function for users to check the current balance based on the provided account_number.
- It retrieves the account index in the data dataset, accesses the current balance, and prints a message displaying the user's account balance.
- The function offers a straightforward means for users, typically in a banking or account management context, to inquire about their account balances.
- Assumptions include the existence of a data dataset with columns like "Account Number" and "Balance," indicating integration into a larger system for proper dataset management.

7. Change pin:-

```
1 def change_pin(account_number):
2     while True:
3         old_pin = int(input("Enter your old 4-digit PIN: "))
4         if old_pin in data[data["Account Number"] == account_number]["Pin"].values:
5             print("Old PIN verified.")
6             break
7         else:
8             print("Incorrect old PIN. Please try again.")
9
10    while True:
11        new_pin = input("Enter your new 4-digit PIN: ")
12        inputs["Newpin"].append(new_pin)
13
14        if new_pin.isdigit() and len(new_pin) == 4:
15            index = data[data["Account Number"] == account_number].index[0]
16            data.at[index, "Pin"] = int(new_pin)
17            print(f"Your new PIN has been set to: {new_pin}")
18            break
19        else:
20            print("Invalid PIN. Please enter a 4-digit number.")
```

- The code defines change_pin for users to modify their 4-digit PIN.
- Two nested loops ensure accurate old PIN verification and entry of a valid new PIN.
- Users verify the old PIN in the first loop; in the second loop, a new 4-digit PIN is entered and updated in the dataset.
- A success message is displayed once the PIN is changed.

8. Exit:-

```
1 def exit():  
2     print("Thank you for using ATM. Have a great day!")  
3     time.sleep(3)  
4     os.system('cls')
```

- The exit function in the ATM program displays a farewell message.
- It introduces a 3-second delay using `time.sleep(3)`.
- The function clears the console screen with `os.system('cls')`.
- It ensures a courteous exit experience for users.

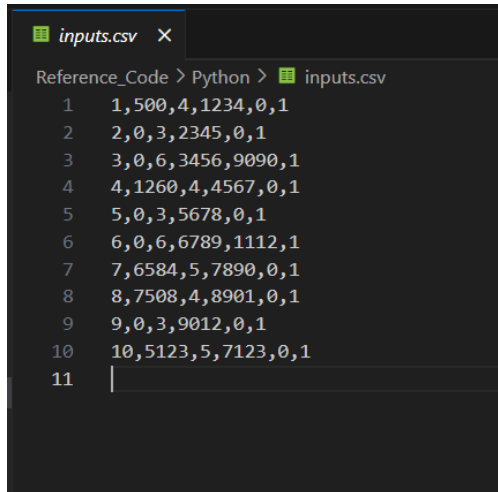
7.2 Comparing Output with Given Inputs

- **Brief Description:**

We saved the inputs scenario applied to the reference model in a CSV file, preserving the corresponding outputs in a separate CSV file. Subsequently, we applied the same input dataset using inputs CSV file for the Verilog design, conducting a thorough comparison of its outputs with the previously saved reference output file.

- **Comparing files:**

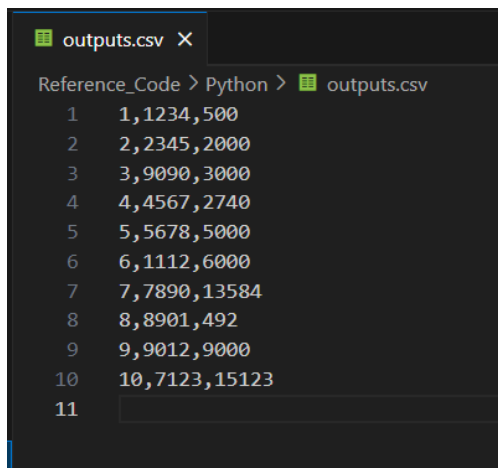
- **Inputs file:**



A screenshot of a code editor window titled 'inputs.csv'. The editor shows a list of 10 rows of data, each containing five comma-separated integers. The rows are numbered 1 through 10 on the left margin. The data is as follows:

Line	Input 1	Input 2	Input 3	Input 4	Input 5
1	1	500	4	1234	0,1
2	2	0	3	2345	0,1
3	3	0	6	3456	9090,1
4	4	1260	4	4567	0,1
5	5	0	3	5678	0,1
6	6	0	6	6789	1112,1
7	7	6584	5	7890	0,1
8	8	7508	4	8901	0,1
9	9	0	3	9012	0,1
10	10	5123	5	7123	0,1

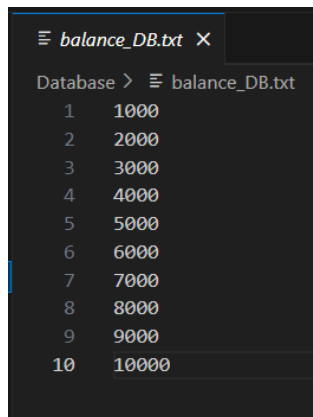
- **Outputs file:**



A screenshot of a code editor window titled 'outputs.csv'. The editor shows a list of 10 rows of data, each containing three comma-separated integers. The rows are numbered 1 through 10 on the left margin. The data is as follows:

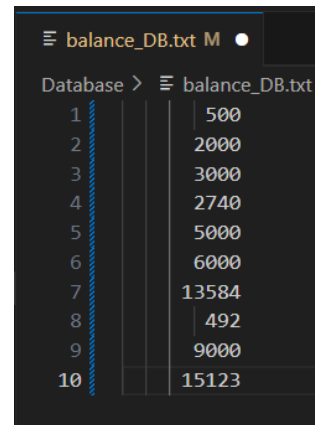
Line	Output 1	Output 2	Output 3
1	1	1234	500
2	2	2345	2000
3	3	9090	3000
4	4	4567	2740
5	5	5678	5000
6	6	1112	6000
7	7	7890	13584
8	8	8901	492
9	9	9012	9000
10	10	7123	15123

– Database files:



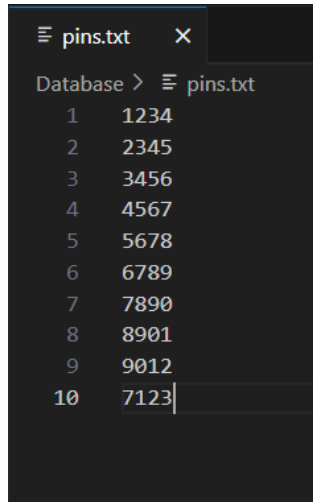
	balance_DB.txt
1	1000
2	2000
3	3000
4	4000
5	5000
6	6000
7	7000
8	8000
9	9000
10	10000

Figure 15: Balance Before



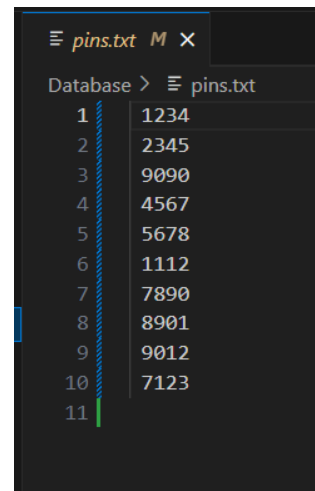
	balance_DB.txt
1	500
2	2000
3	3000
4	2740
5	5000
6	6000
7	13584
8	492
9	9000
10	15123

Figure 16: Balance After



	pins.txt
1	1234
2	2345
3	3456
4	4567
5	5678
6	6789
7	7890
8	8901
9	9012
10	7123

Figure 17: Pins Before



	pins.txt
1	1234
2	2345
3	9090
4	4567
5	5678
6	1112
7	7890
8	8901
9	9012
10	7123
11	

Figure 18: Pins After

– Comparison transcript:

# counter = 0		# Account Pin authenticated 5678	
# Account Pin authenticated 1234		# counter = 0	
# counter = 0		# counter = 0	
# counter = 0		# counter = 0	
# counter = 0		# Account Balance: 5000	
# Withdrawal successful! New balance: 500		# counter = 0	
# counter = 0		# Comparing Outputs of Balance for Test Case	5: PASSED
# Comparing Outputs of Balance for Test Case	1: PASSED	# Account Pin authenticated 6789	
# Account Pin authenticated 2345		# counter = 0	
# counter = 0		# counter = 0	
# counter = 0		# counter = 0	
# Account Balance: 2000		# PIN changed successfully	
# counter = 0		# Account Pin authenticated 1112	
# counter = 0		# counter = 0	
# counter = 0		# Comparing Outputs of Balance for Test Case	6: PASSED
# Comparing Outputs of Balance for Test Case	2: PASSED	# Account Pin authenticated 7890	
# Account Pin authenticated 3456		# counter = 0	
# counter = 0		# counter = 0	
# counter = 0		# counter = 0	
# PIN changed successfully		# counter = 0	
# Account Pin authenticated 9090		# Deposit successful! New balance: 13584	
# counter = 0		# counter = 0	
# Comparing Outputs of Balance for Test Case	3: PASSED	# Comparing Outputs of Balance for Test Case	7: PASSED
# Account Pin authenticated 4567		# Account Pin authenticated 8901	
# counter = 0		# counter = 0	
# counter = 0		# counter = 0	
# counter = 0		# counter = 0	
# Withdrawal successful! New balance: 2740		# Withdrawal successful! New balance: 492	
# counter = 0		# counter = 0	
# Comparing Outputs of Balance for Test Case	4: PASSED	# Comparing Outputs of Balance for Test Case	8: PASSED
# Account Pin authenticated 5678		# Account Pin authenticated 9012	
		# counter = 0	
		# counter = 0	
		# counter = 0	
		# counter = 0	
		# Account Balance: 9000	
		# counter = 0	
		# Comparing Outputs of Balance for Test Case	9: PASSED
		# Account Pin authenticated 7123	

(a) Balance Comparison result

(b) Balance Comparison result

```
# Comparing Pins for Test Case      1: PASSED
# Comparing Pins for Test Case      2: PASSED
# Comparing Pins for Test Case      3: PASSED
# Comparing Pins for Test Case      4: PASSED
# Comparing Pins for Test Case      5: PASSED
# Comparing Pins for Test Case      6: PASSED
# Comparing Pins for Test Case      7: PASSED
# Comparing Pins for Test Case      8: PASSED
# Comparing Pins for Test Case      9: PASSED
# Comparing Pins for Test Case     10: PASSED
# ** Note: Cases are in C:/Users/James/Desktop/1234
```

Figure 20: Pins Comparison result