Verification Plan

1. Objectives and Scope:

Objective: Verify the correctness and functionality of the ATM system under various scenarios.

Scope:

- Verify card insertion, language selection, and request handling.
- Check deposit, withdrawal, and balance display functionalities.
- Validate responses to incorrect password entry.
- Assess system behavior during simultaneous requests.

2. Verification Items:

- **Individual components:**
  - Inputs: insertion, language selection, request, deposit, withdrawal, CardNo, Password.
  - Outputs: Balance Displayed, Deposit accepted and Withdraw Accepted.

```
reg clk, rst, language_selected, card_inserted;
reg [1:0] request;
reg [11:0] deposit_value;
reg [11:0] withdraw_value;
reg [2:0] CardNo;
reg [2:0] Password;
reg [2:0] temp;

wire deposit_accepted;
wire withdraw_accepted;
wire [17:0] balance_displayed;
integer i;
```

- **Features:**
  - Card number and password validation, response to different requests.

3. Verification Methods and Techniques:

- **Methods:**
    - Simulation-based testing using Verilog.
    - Randomized testing for various inputs.
    - Assertions for checking expected conditions.
- **Techniques:**
    - Cover different scenarios: normal transactions, multiple card scenarios, unusual transactions, incorrect password handling, simultaneous requests, and random transactions.
    - Test the same scenarios in the Verilog test bench and then in Visual Studio using C++ and verify the results are the same (Self Checking Test)

4. Test Environment and Infrastructure:

- **Resources:**
    - Appropriate Verilog simulator (e.g., QuestaSim).
    - High level language compiler (e.g., Visual Studio Code)

5. Test Procedures:

- **Procedure:**
    - Execute simulations for each scenario.
    - Check inputs, expected outputs, and system responses.
    - Assess the correctness of balance displayed after transactions.

- **Test Cases:**

Before we begin the testing, we initialize all the inputs with 0 with rst=0(active low) to ensure all variables are ready.

```
initial begin
    rst=0;
    language_selected=0;
    card_inserted=0;
    request=0;
    deposit_value=0;
    withdraw_value=0;
    CardNo=0;
    Password=0;
    @(negedge clk);
```

Initializing Balances:

```
Balances[0] = 18'd125000;
Balances[1] = 18'd130000;
Balances[2] = 18'd135000;
Balances[3] = 18'd140000;
Balances[4] = 18'd120000;
```

****

5.1 Case #1

```
rst=1;
language_selected=1;
card_inserted=1;
CardNo=1;
Password=1;
@(negedge clk);
@(negedge clk);
@(negedge clk);
request=1;
deposit_value=500;
@(negedge clk);
@(negedge clk);
request=2;
withdraw_value=200;
@(negedge clk);
@(negedge clk);
request=1;
deposit_value=500;
@(negedge clk);
@(negedge clk);
request=2;
withdraw_value=200;
@(negedge clk);
@(negedge clk);
request=0;
@(negedge clk);
@(negedge clk);
request=3;
rst=0;
```

Testing Multiple successive withdraw/deposit transactions with constrained values.

CardNo=1, Password=1. Balance[0]

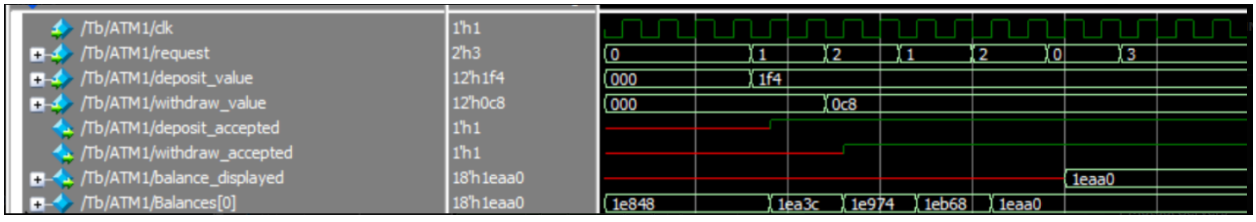
Deposit 500, Withdraw 200, Deposit 500, Withdraw 200.

Expected Results:

125000+500-200+500-200= 125600

Results:

Balanced[0] = $(125600)_{10}$ =$(1eaa0)_{16}$



****

5.2 Case #2

```
@(negedge clk);
@(negedge clk);

    rst=1;
    language_selected=0;
    card_inserted=1;
    CardNo=2;
    Password=2;
    @(negedge clk);
    @(negedge clk);
    @(negedge clk);
    request=1;
    deposit_value=1000;
    @(negedge clk);
    @(negedge clk);
    request=2;
    withdraw_value=800;
    @(negedge clk);
    @(negedge clk);
    withdraw_value=700;
    @(negedge clk);
    @(negedge clk);
    request=1;
    deposit_value=500;
    @(negedge clk);
    @(negedge clk);
    request=2;
    withdraw_value=600;
    @(negedge clk);
    @(negedge clk);
    request=3;
    rst=0;

    @(negedge clk);
    @(negedge clk);
```

Testing Multiple successive withdraw/deposit transactions with constrained values.

CardNo=2, Password=2 . Balance[1]

Deposit 1000, Withdraw 800, Withdraw 700 Deposit 500, Withdraw 600.

Expected Results:

130000 + 1000 – 800 – 700 + 500 -600 = 129,400

Results:

Balance[1]= $(129400)_{10} = (1f978)_{16}$.

****

5.3-Case #3

```
rst=1;
language_selected=0;
card_inserted=1;
CardNo=3;
Password=3;
@(negedge clk);
@(negedge clk);
@(negedge clk);
request=1;
deposit_value=100;
@(negedge clk);
@(negedge clk);
request=2;
withdraw_value=1500;
@(negedge clk);
@(negedge clk);
request=3;
rst=0;
@(negedge clk);
@(negedge clk);
```

Testing Multiple successive withdraw/deposit transactions with constrained values.

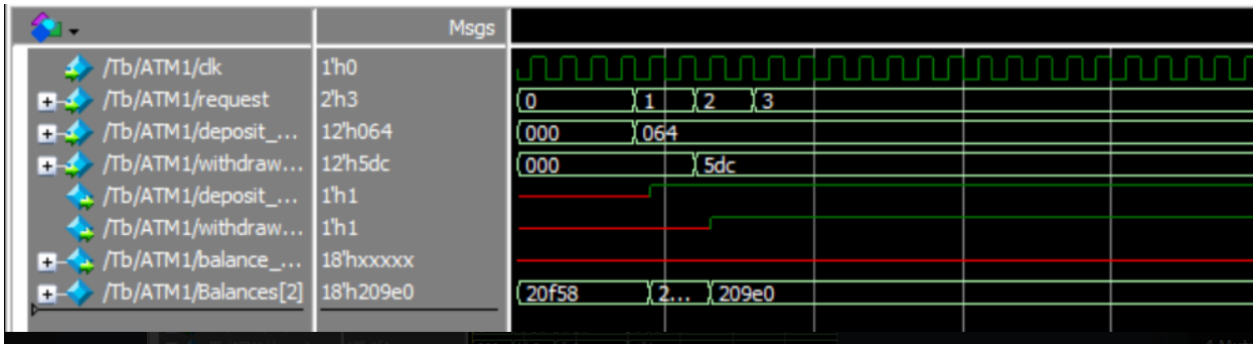
CardNo=3, Password=3 . Balance[2]

Deposit 100, Withdraw 1500

Expected Results:

135,000 + 100 – 1500 = 133,600

Results:



Balance[2] = $(133600)_{10} = (209e0)_{16}$.

****

5.4-Case #4

```
rst=1;
language_selected=0;
card_inserted=1;
CardNo=4;
Password=4;
@(negedge clk);
@(negedge clk);
@(negedge clk);
request=1;
deposit_value=3000;
@(negedge clk);
@(negedge clk);
request=2;
withdraw_value=2500;
@(negedge clk);
@(negedge clk);
request=1;
deposit_value=4000;
@(negedge clk);
@(negedge clk);
request=3;
rst=0;
@(negedge clk);
@(negedge clk);
```

Testing Multiple successive deposit/withdraw transactions with a deposit value greater than 2000.

CardNo=4, Password=4 . Balance[3]

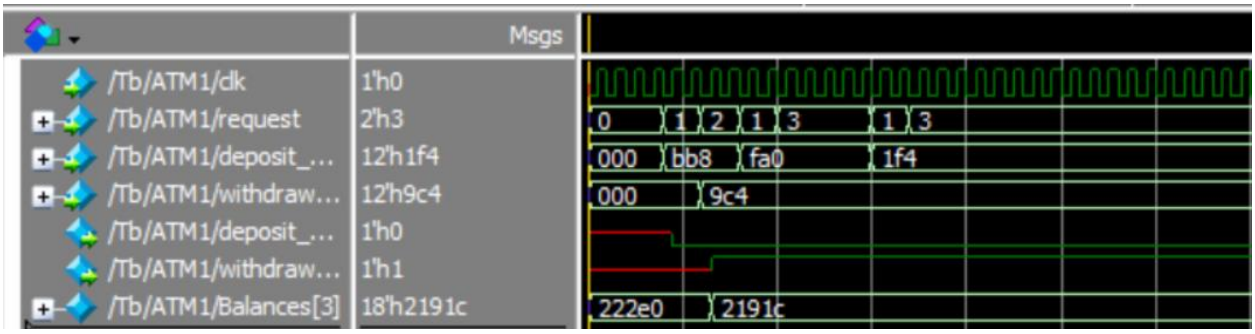
Deposit 3000, withdraw 2500, deposit 4000

Expected Results:

140,000 -2500 = 137,500

Deposit value of 2500 and 4000 should not be accepted

Results:



Balance[3] = $(137500)_{10} = (2191c)_{16}$.

****

5.5-Case #5

```
rst=1;
language_selected=1;
card_inserted=1;
CardNo=6;
Password=6;
@(negedge clk);
@(negedge clk);
@(negedge clk);
request=1;
deposit_value=500;
@(negedge clk);
@(negedge clk);
request=3;
rst=0;
@(negedge clk);
@(negedge clk);
```

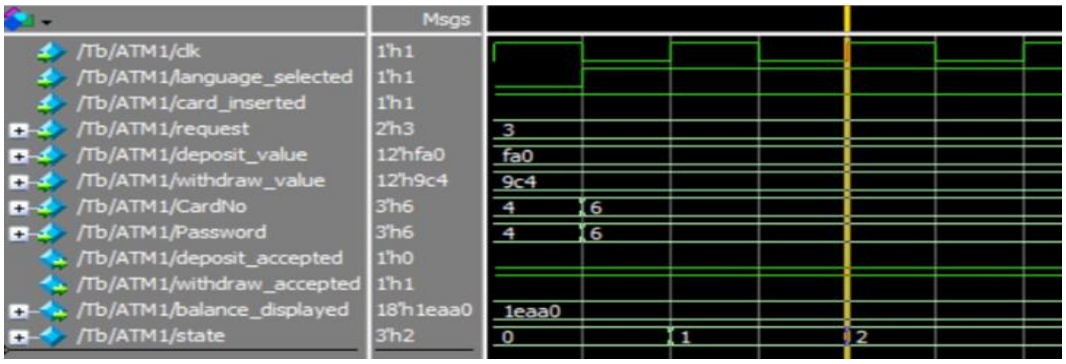
Testing what would happen given a wrong password or/and wrong card number

cardNo=6, password=6.

Expected Results:

All balances would stay the same regardless of the withdraw/ deposit request because user with cardNo=6 does not exist.

Results:



Balances[0]= 125,000

Balances[1]= 130,000

Balances[2]= 135,000

Balances[3]= 140,000 Balances[4]= 120,000

****

5.6-Case #6

```
// no.6 two requsts at a time
    rst=1;
    language_selected=0;
    card_inserted=1;
    CardNo=2;
    Password=2;
    @(negedge clk);
    @(negedge clk);
    @(negedge clk);
    request=1;
    deposit_value=100;
    request=2;
    withdraw_value=500;
    @(negedge clk);
    @(negedge clk);
    request=3;
    rst=0;

    @(negedge clk);
    @(negedge clk);
/*
```

Testing how the program responds given two different requests at the same time (eg. Deposit and Withdraw)

CardNo=2, password=2, Balances[1]

Expected Results:

First request is ignored and only the second one is proceeded.

Request 1 (deposit) is ignored and request 2 (withdraw) is completed.

130,000- 500 = 129,500

Result:

Balance[1] = $(129500)_{10} = (1f9dc)_{16}$

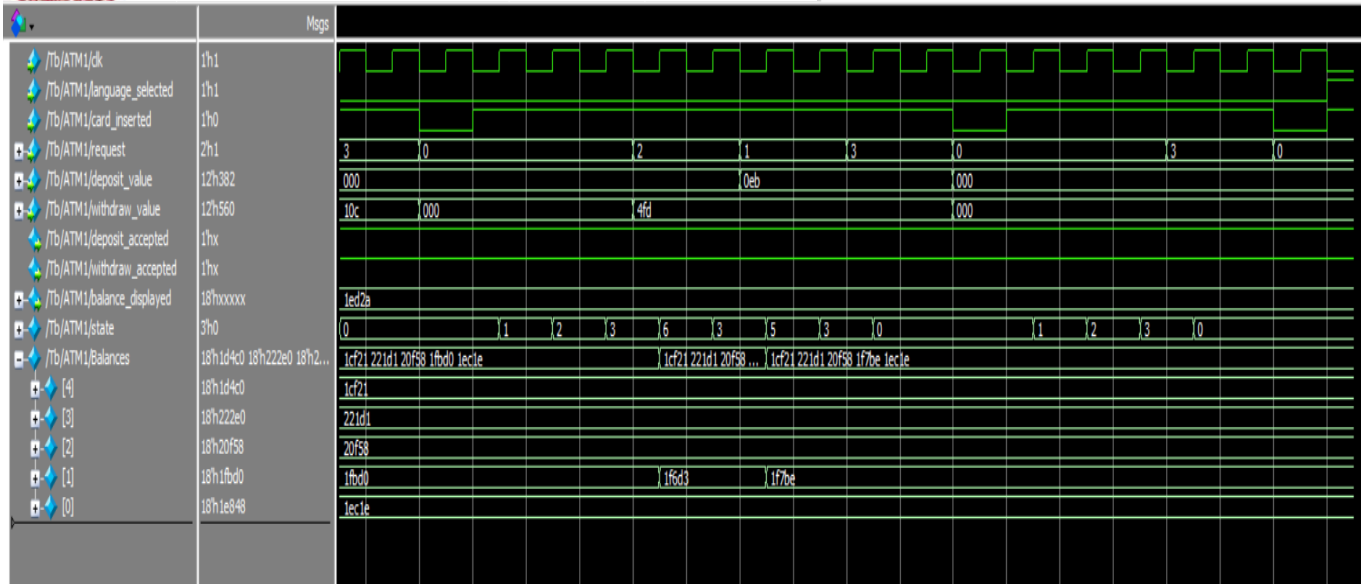
****

5.7. RANDOM TESTING

Here we decided to initialize a for loop that loops 5000 times testing different customers with randomized inputs

```
for(i = 0;i < 5000; i=i+1) begin

 //Resting the values
      rst=0;
      language_selected=0;
      card_inserted=0;
      request=0;
      deposit_value=0;
      withdraw_value=0;
      CardNo=0;
      Password=0;


 @(negedge clk);

      rst=1;
      card_inserted = $random();
      language_selected=$random();
      temp = $urandom_range(1,5);
      CardNo = temp;
      Password = temp;

      @(negedge clk);
      @(negedge clk);
      @(negedge clk);//after password

      request=$random();
      if(request==1) begin
          deposit_value=  1 + ($random()) % (2000 - 1);
      end
      else if(request==2) begin
          withdraw_value=  1 + ($random()) % (2000 - 1);
      end

      @(negedge clk);
      @(negedge clk);
while(request != 3) begin

      request=$random();

      if(request==1) begin
          deposit_value=  1 + ($random()) % (2000 - 1);
      end
      else if(request==2) begin
          withdraw_value=  1 + ($random()) % (2000 - 1);
      end

      @(negedge clk);
      @(negedge clk);

end
end
end
endmodule
```

RESULTS:



****

6.Coverage Report & Assertions:

```
Coverage Report Summary Data by file

================================================================================
=== File: ATM.v
================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                        33        29         4    87.87%
    Conditions                      13         8         5    61.53%
    FSM States                       7         7         0   100.00%
    FSM Transitions                 15        11         4    73.33%
    Statements                      57        53         4    92.98%
    Toggles                        262       165        97    62.97%

================================================================================
=== File: Tb.v
================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                         6         6         0   100.00%
    Conditions                       5         5         0   100.00%
    Statements                     184       184         0   100.00%
    Toggles                        182       141        41    77.47%


TOTAL ASSERTION COVERAGE: 100.00%  ASSERTIONS: 4

Total Coverage By File (code coverage only, filtered view): 83.75%
```

Coverage report with a total coverage of 83.75%
Branches covered with a 87.87%
Conditions covered with a 61.53%

ASSERTIONS:

```
// Assertion 1:
// psl assert always (request == 2 -> next withdraw_accepted) @(posedge clk);

// Assertion 2:
// psl assert always (request == 1 && deposit_value > 2000 -> next !deposit_accepted) @(posedge clk);

// Assertion 3: deposits cannot be negative.
// psl assert always (deposit_value >= 0) @(posedge clk);

// Assertion 4: withdrawals cannot be negative.
// psl assert always (withdraw_value >= 0) @(posedge clk);
```

7. Verification Schedule:

- **Timeline:**
    - Conduct simulations for each scenario.
    - Regular reviews and updates based on feedback.

8. Criteria for Acceptance:

- **Acceptance Criteria:**
    - All test scenarios pass.
    - Expected outputs match simulation results.
    - No critical errors or issues.

9. Reporting and Documentation:

- **Documentation:**
    - Record test results, including pass/fail status.
- **Reporting:**
    - Generate comprehensive reports after each simulation.

Conclusion:

This verification plan provides a structured approach to validate the ATM system's functionality. Regular updates and reviews will ensure thorough testing and identification of potential issues. Execute simulations with various inputs and scenarios to achieve comprehensive verification.