



Cairo university
Faculty of Engineering
Electronics and electrical communication Engineering Department

ATM Verification Plan

Submitted by:

Name	Sec.	BN.	ID
عبد النعيم عبد الباسط عبد النعيم	3	1	9190642
علي بدري عبد النعيم عبد المالك	3	4	9190097
كريم احمد ثابت عبد الرحيم	3	22	9190023
كيرلس نشأت نجيب دميان	3	25	9190598
محمد ايمن السيد محمود	3	33	9190975
نبيل ياسر نبيل محمد	4	38	9190943

Table of Contents

1	Introduction.....	3
2	verification environment and test cases:	3
2.1	self – checking testbench.....	3
2.2	Testbench using Test Stimulus using directed &constraint random.....	4
3	Assertions Plan	6
4	Coverage Plan	6
4.1	Code Coverage	6
4.2	Functional Coverage	7
5	Results	8
5.1	self – checking testbench.....	8
5.2	Testbench using Test Stimulus using directed &constraint random.....	9
5.2.1	Simulation wave results.....	9
5.2.2	Assertion results.....	10
5.2.3	Coverage results.....	10

1 INTRODUCTION

The project targets to build ATM core system that controls some auxiliary devices such as: card handling, money counting, ... etc. The operations that should be controlled are: Card handling, language displayed, card password, Timers, de-activating accounts, normal ATM operations and other user interfacing options.

The main hardware components used are control unit and RAM to load and store client's information and modify them as needed. Here came the challenge to verify our design to insure its correct functionality as well as testing all its operations.

Functional verification is a crucial part of any hardware design project, as it ensures that the system meets the specifications and requirements. We defined the need to set verification phases starting with assertions that should take couple of days to identify its quantity and try to implement them based on our systemverilog assertion background from lectures and examine its rate of success using QuestaSim simulator. Another couple of days should be given to decide the cover plan "code and functional coverage" as well as the coverage percentage goal based on the implemented design.

We also intended to build two testbenches self-checking testbench to force some apparent testcases to check the simirality with High level output and another test-bench with constraint random stimuli generation is applied also.

2 VERIFICATION ENVIRONMENT AND TEST CASES:

We start developing our verification environment through two testbenches

2.1 self – checking testbench

It consist of inputs, DUT instantiation, Clock generator, reset generator and monitor to check with final output from high level code. It consist of directed stimulus generation to enter the same inputs and to make same transactions similar to high level entered inputs. The monitor block is to compare between final values in the RAM in the Verilog and final values in the matlab code and we got zero errors.

```

//-----
// Directed stimulus inputs
//-----

#10;
//client 3 : password right -> with draw operation = 90 -> balance = 1841 -> 1751
#10 Card_Inserted = 1;
#10 Acc_Number = 2; Card_Inserted = 0;
#10 Language = 1;
#10 Password = 'b0001_0010_1011_0110;
#10 Operation = 'b0;
#10 Cash_Amount = 90;
#30;
//client 4 : password wrong -> password right -> depoist = 800 -> balance = 7343 -> 8143
#10 Card_Inserted = 1;
#10 Acc_Number = 3; Card_Inserted = 0;
#10 Language = 1;
#10 Password = 'b1;
#10 Password = 'b0010_0001_0111_1110;
#10 Operation = 1;
#10 Money_Counter_Amount = 800;
| | Money_Counter_Valid = 1'b1;
#30;

//client 5 : password wrong -> password wrong -> password right -> show balance operation -> balance = 2653
#10 Card_Inserted = 1;
#10 Acc_Number = 4; Card_Inserted = 0;
#10 Language = 1;
#10 Password = 'b1;
#10 Password = 'b0010_0001_0111_1110;
#10 Password = 'b0010_0101_1001_0000;
#10 Operation = 2;
#30;

```

Figure 1: Directed stimulus inputs

```

//-----
// Monitor
//-----

initial
begin
    @(posedge enable );
    $readmemb("final_output.txt", Expected_Accounts_database);
    error = 0;
    for ( i = 0; i < SAVED_ACCOUNTS ; i = i+ 1)
    begin
        if ( DUT.u_RAM.Accounts[i] == Expected_Accounts_database[i] )
        begin
            $display("Test Case %d is succeeded",i);
        end
        else
        begin
            $display("Test Case %d is failed", i);
            error = error + 1;
        end
    end
    $display("Number of errors = %d", error);
end

```

Figure 2: Monitor block

2.2 Testbench using Test Stimulus using directed &constraint random

As directed tests don't cover all possible scenarios we apply random test stimulus to input ports but with some constraints as there may be an invalid cases. We create 2 classes one for input ports and the other for password input only. To enable randomization for the variable we declare them with **randc** , then we instantiate an object from class using **new()** functions and apply randomization with **instance name.randomize()**.

We apply directed stimulus to card inserted variable only as to fix it equals one only the force it zero after one clock cycle.

We apply constraints to the values of input ports as to avoid some invalid cases, they are shown in the table below:

Input name	Constraint
Language	Make 1,2 has chance 90/100, 0 has chance 10/100, 3 has chance 0/100
Money_Counter_Valid	Make 1 has chance 70/100, 0 has chance 30/100
In_Another_Operation_Wanted	Make 1 has chance 20/100, 0 has chance 80/100
Card_Valid	Make 1 has chance 99/100, 0 has chance 1/100
Card_Number	Make it take a number from 0 to 9 only
Password	Make it take a password from one of the ten account's password

```
class myclass
#( byte    NUM_OF_STATES='d11,          NUM_OF_STATES_WIDTH=$clog2(NUM_OF_STATES),
  byte    NUM_SUPPORTED_LANGUAGES='d3 , NUM_SUPPORTED_LANGUAGES_WIDTH=$clog2(NUM_SUPPORTED_LANGUAGES),
  byte    PASSWORD_WIDTH='d16,
  byte    BALANCE_WIDTH='d16,
  byte    NUM_OF_TRIES='d3 ,          NUM_OF_TRIES_WIDTH=$clog2(NUM_OF_TRIES),
  byte    NUM_OF_OPERATION='d4 ,      NUM_OF_OPERATION_WIDTH=$clog2(NUM_OF_OPERATION),
  byte    SAVED_ACCOUNTS='d10 ,      ACCOUNT_NUMBER_WIDTH=$clog2(SAVED_ACCOUNTS)
);
  randc bit          [ACCOUNT_NUMBER_WIDTH-1:0]      Card_Valid;
  randc bit          [1:0]                          Card_Number;
  randc bit          [1:0]                          Language;
  randc bit          [BALANCE_WIDTH-1:0]             KeyPad_Balance;
  randc bit          [NUM_OF_OPERATION_WIDTH-1:0]    Operation_Selection;
  randc bit          [1:0]                          In_Another_Operation_Wanted;
  randc bit          [1:0]                          Money_Counter_Valid ;
  randc bit          [BALANCE_WIDTH-1:0]             Money_Counter_Amount ;

//constraints
constraint const1 {Language dist { [1:2]:/90, 3:/0, 0:/10}; }
constraint const2 {Money_Counter_Valid dist {1:/70, 0:/30}; }
constraint const3 {In_Another_Operation_Wanted dist {0:/80, 1:/20};}
constraint const4 {Card_Valid dist {1:/99, 0:/1};}
constraint const5 {Card_Number inside {[0:9]};}
```

Figure 3: Inputs class declaration

```
class passwordclass#(byte PASSWORD_WIDTH = 'd16);
  rand bit          [PASSWORD_WIDTH-1:0]      KeyPad_Password;
  constraint const3 {KeyPad_Password inside {
    'b0001000001100011,
    'b0001100100101100,
    'b0001001010110110,
    'b0010000101111110,
    'b0010010110010000,
    'b0001000100001010,
    'b0000010101111101,
    'b0001000101100110,
    'b001000000001101,
    'b0001100110011011};
  }
endclass
```

Figure 4: Password class declaration

3 ASSERTIONS PLAN

Our goal was to test all the properties of the ATM, that the ATM is behaving in the correct way such as overwriting “or updating” memory data in different locations that could be done by asserting the change in data when choosing one of the following operations: “withdraw – deposite – change password”. These assertions fire if the corresponding memory locations’ balance or password are not updated.

We implemented five concurrent assertions as shown in figure 5.

```
CARD_INSERT_ASSERT : assert property (insert_card);
NEW_PASSWORD_ASSERT : assert property (new_password);
DEACTIVATE_ASSERT : assert property (deactivating_account);
BALANCE_UPDATE_DEPOSITE_ASSERT: assert property (balance_update_deposite);
BALANCE_UPDATE_WITHDRAW_ASSERT: assert property (balance_update_withdraw);
WITH_DRAW_ERROR_ASSERT : assert property (with_draw_more_than_balance);
```

Figure 5: Assertions

The table below shows the illustration of each assertion:

Assertion name	Illustration
CARD_INSERT_ASSERT	To check functionality of transition from card insert state to check status state when a card inserted
WITH_DRAW_ERROR_ASSERT	To check that there is no error or change in account balance when entering a large cash amount than account’s balance
BALANCE_UPDATE_WITHDRAW_ASSERT	To check if the account’s balance is changed after withdraw operation
BALANCE_UPDATE_DEPOSITE_ASSERT	To check if the account’s balance is changed after a correct withdraw operation
DEACTIVATE_ASSERT	To check that the account is de-activated when entering password 5 times wrong
NEW_PASSWORD_ASSERT	To check that the password is changed inside the RAM

4 COVERAGE PLAN

Based on the two types of the coverage: “code coverage – functional coverage”, we intended to have 100% code coverage and over 85% functional coverage by writing two cover groups for each implemented block “finite state machine” cover group with transition pre-defined bins to observe any violation in our control unit. The second cover group to observe the correct functionality of the memory accessing when defining previous mentioned specific operations.

4.1 Code Coverage

For the code coverage, as mentioned before, we intend to reach 100% code coverage. In QuestaSim Simulator we enabled the options shown in the following table:

Code Coverage Type	Illustration
Statement coverage	Executable statement such as continuous assignment, procedural statement blocks , arithmetic and logical operations are covered
Branch coverage	Number of true or false branch is covered
Condition coverage	Ratio of the number of checked test cases to the total number of cases present
Expression coverage	How logical expressions are evaluted
Finite state machine coverage	Number of states are visited compared to the total states and transitions from one state to another.

Figure 6: Code Coverage enabled options.

By enabling these options, we would be able to make sure that there is no dead code in our design. Besides, we would also make sure that all the test cases would be “or almost all of them” would be induced in our verification environment so that all the states and memory locations are accessed without introducing any fault.

The final report attached to the sent files will show detailed coverage report for the previous mentioned code coverage options with some mentioned in the results section.

4.2 Functional Coverage

We want to make sure that our design is covered functionally correct by checking the transitions made by the FSM and memory accessing in different scenarios. That is the reason there are two cover groups:

Functional Coverage	Illustration
state_transition	Checking the accessed states based on the input account number with its password and the selected options to check whether the right transitions are made or not.
memory_address	Check the valid and invalid memory locations accessed with bins and illegal_bins in the covergroup.

```
covergroup state_transition (ref reg [NUM_OF_STATES_WIDTH-1:0] current_state) @(posedge clk);
  cover_point_states: coverpoint current_state {
    bins VALID_STATE[] = {Idle, Status_Check, Language_Select, Inserting_Password,
                          DeActivate_Account, Operation_Options, Withdraw, Deposit,
                          Change_Password, Balance_Display, Another_Operation};
  }

  bins INACTIVE_ACCOUNT = (Idle => Status_Check => Idle);
  bins DE_ACTIVATION = (Idle => Status_Check => Language_Select => Inserting_Password [*1:NUM_OF_TRIES] => DeActivate_Account => Idle);
  bins WITHDRAWING = (Idle => Status_Check => Language_Select => Inserting_Password [*1:NUM_OF_TRIES] => Operation_Options => Withdraw => Balance_Display => Another_Operation => Idle);
  bins DEPOSITING = (Idle => Status_Check => Language_Select => Inserting_Password [*1:NUM_OF_TRIES] => Operation_Options => Deposit => Balance_Display => Another_Operation => Idle);
  bins BALANCE_DISPLAYING = (Idle => Status_Check => Language_Select => Inserting_Password [*1:NUM_OF_TRIES] => Operation_Options => Balance_Display => Another_Operation => Idle);
  bins CHANGING_PASSWORD = (Idle => Status_Check => Language_Select => Inserting_Password [*1:NUM_OF_TRIES] => Operation_Options => Change_Password => Another_Operation => Idle);
endgroup
state_transition S1;
```

Figure 7: state_transition covergroup.

```
covergroup address @(posedge clk);
  cover_point_address: coverpoint Acc_Number {
    bins Valid_addresses[] = {[0:9]};
    illegal_bins invalid_addresses [] = {[10:15]};
  }
endgroup
address S2;
```

Figure 8: memory_address covergroup.

5 RESULTS

5.1 self – checking testbench

Here is a comparison between Matlab output and HDL output

Matlab	HDL
<pre>000100000110001100100110111100110 000110010010110000001111001000000 001000110010011000000110110101110 00100001011111100001111110011111 001001011001000000001010010111011 000100010000101000000011011010111 000001010111110100011100010000000 000100010110011000011000100000100 001000000000110100011101001000111 000110011001101100010110011110111</pre>	<pre>000100000110001100100110111100110 000110010010110000001111001000000 001000110010011000000110110101110 00100001011111100001111110011111 001001011001000000001010010111011 000100010000101000000011011010111 000001010111110100011100010000000 000100010110011000011000100000100 001000000000110100011101001000111 000110011001101100010110011110111</pre>
Figure 9: Matlab output	Figure 10: Verilog output

We got zero errors when comparing between two outputs as shown in figure 11.


```

# Test Case      0 is succeeded
# Test Case      1 is succeeded
# Test Case      2 is succeeded
# Test Case      3 is succeeded
# Test Case      4 is succeeded
# Test Case      5 is succeeded
# Test Case      6 is succeeded
# Test Case      7 is succeeded
# Test Case      8 is succeeded
# Test Case      9 is succeeded
# Number of errors = 0

```

Figure 11: compare result between 2 outputs

5.2 Testbench using Test Stimulus using directed &constraint random

5.2.1 Simulation wave results

The wave shows how the input and output port behaves based on the current situation and the intended final state it should be in, here are some explanations on the ports transitions

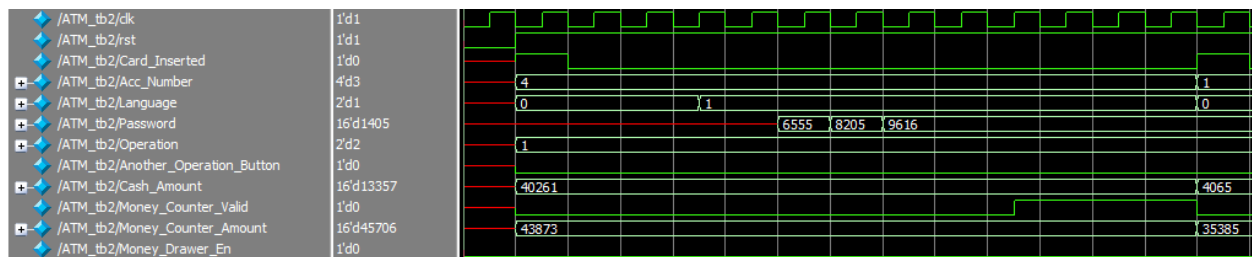


Figure 12: Deposit option transaction

Signal's name	Illustration
Card_Inserted	It should be high in the beginning of each transaction to initiate the operation.
Acc_Number	Insert one of the reserved IDs to load its password and balance to compare the inserted values to those loaded. Inserted once at the beginning of the transaction.
Password	The value of password is re-inserted each time the machine finds it incorrect for limited number of times pre-specified (5 in the current case). We find that it is inserted correct at the third time.
Operation	One of four specified values is inserted for each transaction. In the current case is deposit option selected which is option one.
Another_operation_option	It indicates if another transaction or option is to be made after finishing the first one. In the current case it is off.

Here is another case when the inputs were stimulated to insert wrong password five consecutive times which results deactivating the account.

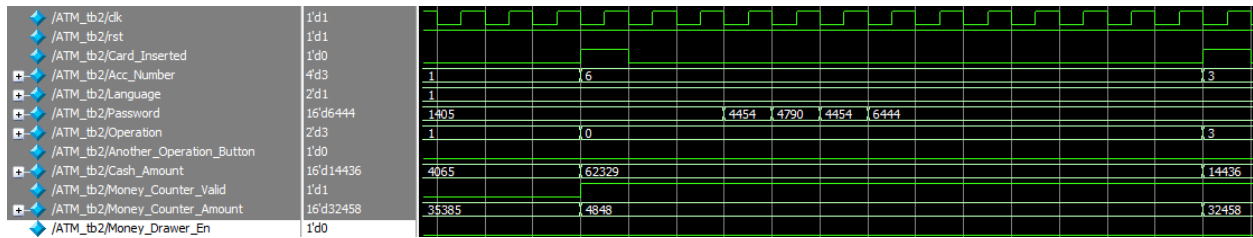


Figure 13: Inserting wrong passwords.

5.2.2 Assertion results





Name	Assertion Type	Language	Enable	Failure	Pass Count	Active Count	Memory	Peak Memory	Peak Memory Time	Cumulative Threads	ATV	Assertion Expression	Included
 /ATM_tb2/CARD_INSERT_ASSERT	Concurrent	SVA	on	0	180	0	0B	80B	15000 ps	180 off	assert(@(posedge clk) ...	✓	
/ATM_tb2/NEW_PASSWORD_ASSERT	Concurrent	SVA	on	0	7	0	0B	80B	1135000 ps	7 off	assert(@(posedge clk) ...	✓	
/ATM_tb2/DEACTIVATE_ASSERT	Concurrent	SVA	on	0	112	0	0B	400B	235000 ps	705 off	assert(@(posedge clk) ...	✓	
 /ATM_tb2/BALANCE_UPDATE_DEPOSITE_ASSERT	Concurrent	SVA	on	0	6	0	0B	80B	115000 ps	6 off	assert(@(posedge clk) ...	✓	
 /ATM_tb2/BALANCE_UPDATE_WITHDRAW_ASSERT	Concurrent	SVA	on	0	1	0	0B	80B	14045000 ps	1 off	assert(@(posedge clk) ...	✓	
 /ATM_tb2/WITH_DRAW_ERROR_ASSERT	Concurrent	SVA	on	0	4	0	0B	160B	1185000 ps	8 off	assert(@(posedge clk) ...	✓	

Figure 14:Assertion results

As shown in figure 11, none of the assertions were fired which indicates the right operation of the design as intended with all the cases of operations tested. Also the assertions are included to the waveform to observe if each *Operation* value, the corresponding assertion approve or fire.

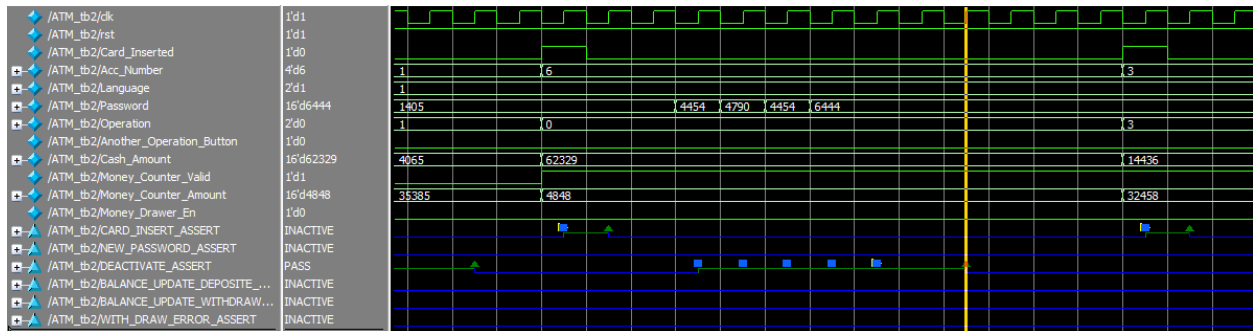


Figure 15: Assertions added to the waveform.

5.2.3 Coverage results

Instance	Assertions	Branches	Conditions	Covergroups	Expressions	FSM States	FSM Transitions	Statements	Total
Total	100%	96.77%	85.71%	100%	100%	100%	70.83%	99.19%	94.06%
ATM_tb2	100%	88.88%	83.33%	100%	-	-	-	98.27%	94.09%
DUT	-	100%	100%	-	100%	100%	70.83%	100%	95.13%

Figure 16: Total coverage results.

For more results, please refer to the coverage report.