**DESIGN AND IMPLEMENTATION OF AUTOMATED TELLER MACHINE (FSM) CONTROLLER**

**TEAM NAME:** RAPTORS

**TEAM MEMBERS:**

1. Khaja Muzamil Ahmed Mohiuddin
2. Nallam Setty Venkata Kiran

**Mentor Name:** Dr. S. Kalaivani

**College:** B. S. Abdur Rahman Crescent Institute of Science and Technology

**INTRODUCTION**

ATM is an electronic device that allows bank customers to perform various banking transactions without the need for human interaction. It typically provides services such as cash withdrawals, balance inquiries, fund transfers, and more. The ATM operates based on user input and interacts with the bank's backend systems to process transactions.

In Verilog, an ATM can be implemented as a finite state machine (FSM). An FSM is a computational model that consists of a set of states, transitions between states, and actions associated with each state. The states represent different operational modes or states of the ATM, while the transitions define how the ATM moves from one state to another based on user input and other conditions.

**The Verilog code for an ATM typically includes the following elements:**

**Input and Output Ports:** These are the signals that interface the ATM module with the external environment. Inputs can include clock signals, reset signals, user inputs (buttons, keypad), and card presence detection signals. Outputs can include signals for dispensing cash, ejecting the card, displaying information, etc.

**State Declaration:** In Verilog, you can declare an enumerated type to represent the different states of the ATM. Each state is assigned a unique value using a binary representation.

**State Register:** A register variable is used to store the current state of the ATM. It is typically updated on the positive edge of the clock signal or in response to a reset signal.

**Next State Logic:** Based on the current state, input signals, and other conditions, the next state of the ATM is determined. This logic is implemented using a combinational block that assigns the appropriate value to the state next variable.

**State Transition and Output Logic:** In Verilog, you can use a case statement or if-else conditions to define the transitions between states. Each state transition can be associated with certain actions or outputs, such as cash dispensing, card ejection, or communication with the bank's backend systems.

By implementing the ATM as a finite state machine in Verilog, you can effectively model the behavior of an ATM and define its operation based on user input and system conditions. It's important to note that the provided Verilog code is a simplified example and may not cover all the functionality and details required for a real-world ATM. Implementing a fully functional ATM involves additional modules, such as input handling, display control, cash dispenser control, communication interfaces, and security mechanisms to ensure the confidentiality and integrity of transactions.



Fig. 1. Model of an ATM Machine

**ABOUT INTEL PRIME QUARTUS**

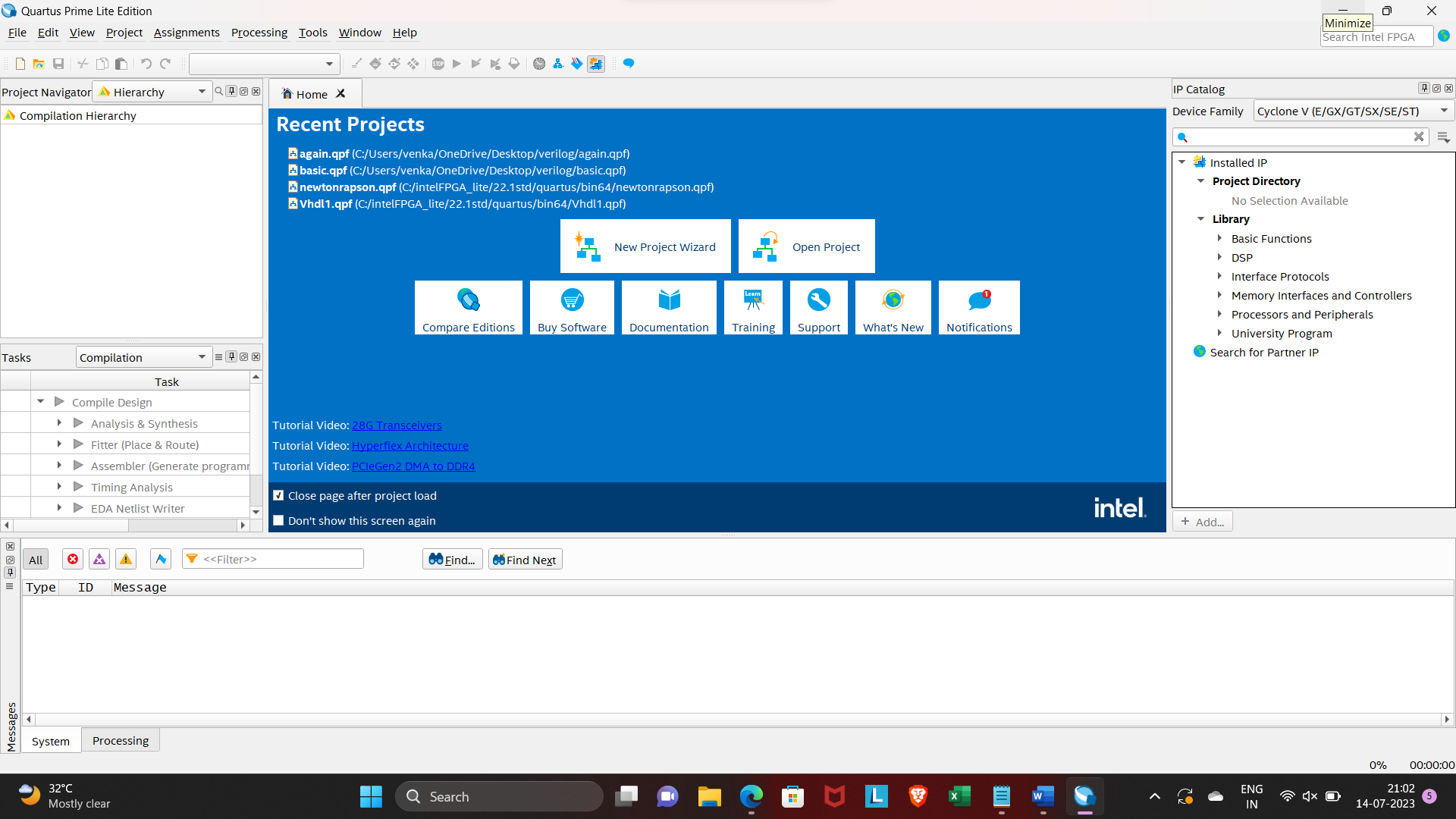


Fig.2. Intel Prime Quartus

**Design Entry:**

Intel Quartus Prime provides a user-friendly design entry environment where you can create or import Verilog source files. Verilog is a hardware description language (HDL) that allows you to describe the behavior and structure of digital circuits.

**Design Analysis and Elaboration:**

After adding the Verilog source files to the project, Quartus Prime performs design analysis and elaboration. It checks the syntax, performs type checking, and constructs an internal representation of the design hierarchy.

**Simulation:**

Quartus Prime integrates a simulator, typically ModelSim, which allows you to verify the functionality of your ATM design before implementing it on an FPGA. You can write testbenches to simulate the behavior of your Verilog code and test various scenarios and inputs.

**Synthesis:**

Synthesis is the process of converting your behavioral-level Verilog code into a gate-level representation that consists of logical elements (gates) and flip-flops. Quartus Prime's synthesis tool analyzes your code, optimizes it for area or speed based on your preferences, and generates a gate-level netlist.

**Design Constraints:**

Design constraints define additional requirements or specifications for your design. These constraints include timing constraints, which specify the desired timing behavior of your design, and I/O constraints, which define the pin assignments and electrical properties of the FPGA.

**Place and Route:**

In the place-and-route stage, Quartus Prime maps the synthesized design onto the specific FPGA device you are targeting. It determines the physical locations of the logical elements on the FPGA chip and creates the necessary routing connections between them.

**Static Timing Analysis (STA):**

STA is performed to ensure that the design meets its timing requirements. Quartus Prime analyzes the timing paths in your design, considering factors such as clock frequency, delays, and signal propagation times. It reports any violations, allowing you to make adjustments if needed.

**Generation of Programming Files:**

Once the design is successfully placed and routed, you can generate programming files required to configure the FPGA with your design. These files typically include a programming file (e.g., sof or. pof) and other auxiliary files that contain the configuration data.

**Programming the FPGA:**

With the generated programming files, you can use the Quartus Prime Programmer to configure the FPGA device. This involves connecting the FPGA board to your computer and selecting the appropriate programming file for the specific FPGA device.

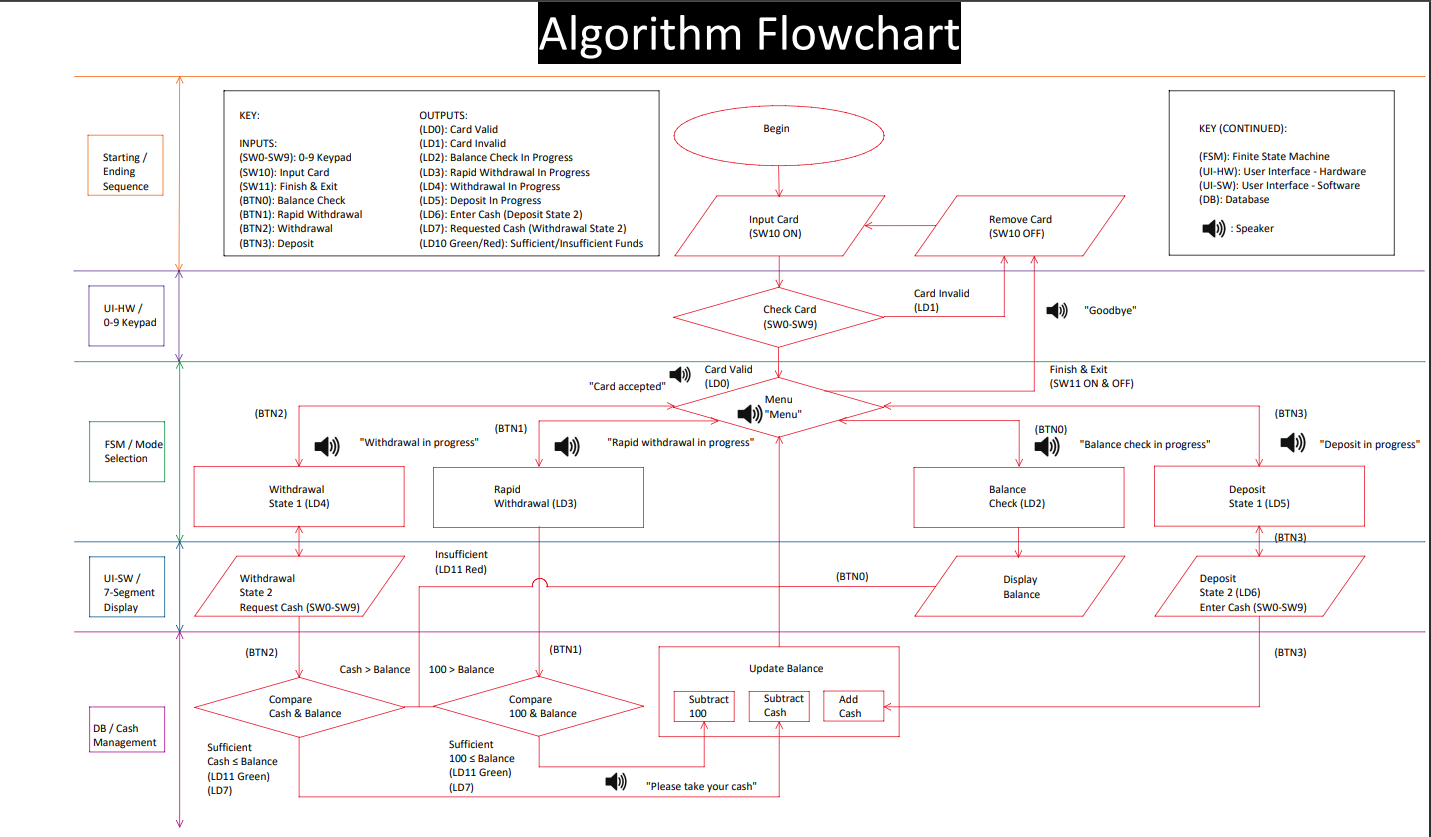
**Verification and Debugging:**

After programming the FPGA, you can test and verify the functionality of your ATM design on the actual hardware. Quartus Prime provides debugging features, such as waveform viewers and RTL (Register Transfer Level) viewers, to help identify and resolve any issues or errors in the design.

**Time Quest Timing Analyzer:**

Quartus Prime also provides a tool called Time Quest Timing Analyzer, which helps designers analyze and optimize the timing characteristics of their designs. In the case of an ATM, this tool can be used to ensure that the VLSI code meets the required timing constraints, such as clock frequencies and signal delays.

Throughout the entire process, Quartus Prime offers a range of tools, utilities, and reports to assist in the design, optimization, and verification of your ATM machine implemented in Verilog. It's important to consult the Quartus Prime documentation and user guides for detailed instructions on using the software and its specific features relevant to your design.

Overall, Intel Quartus Prime is a versatile software suite that can be used to simulate and verify VLSI code for an ATM, allowing designers to ensure the correctness and performance of their designs before they are implemented in hardware.

**PROGRAM**

**1. CODE:**

`define true 1'b1

`define false 1'b0

`define FIND 1'b0

`define AUTHENTICATE 1'b1

`define WAITING 3'b000

`define MENU 3'b010

`define BALANCE 3'b011

`define WITHDRAW 3'b100

`define WITHDRAW\_SHOW\_BALANCE 3'b101

`define TRANSACTION 3'b110

module authentication(

input [11:0] accNumber,

input [3:0] pin,

input action,

input deAuth,

output reg wasSuccessful,

output reg [3:0] accIndex

);

reg [11:0] acc\_database [0:9];

reg [3:0] pin\_database [0:9];

//initializing the database with arbitrary accounts

initial begin

acc\_database[0] = 12'd2749; pin\_database[0] = 4'b0000;

acc\_database[1] = 12'd2175; pin\_database[1] = 4'b0001;

acc\_database[2] = 12'd2429; pin\_database[2] = 4'b0010;

acc\_database[3] = 12'd2125; pin\_database[3] = 4'b0011;

acc\_database[4] = 12'd2178; pin\_database[4] = 4'b0100;

acc\_database[5] = 12'd2647; pin\_database[5] = 4'b0101;

acc\_database[6] = 12'd2816; pin\_database[6] = 4'b0110;

acc\_database[7] = 12'd2910; pin\_database[7] = 4'b0111;

acc\_database[8] = 12'd2299; pin\_database[8] = 4'b1000;

acc\_database[9] = 12'd2689; pin\_database[9] = 4'b1001;

end

always @ (deAuth) begin

if(deAuth == `true)

wasSuccessful = 1'bx;

end

//looping through the database, trying to find a match for the given accNumber and pin

// if action is set to find then it'll simply ry to find a match for the given accNumber and returns its index

integer i;

always @(accNumber or pin) begin

wasSuccessful = `false;

accIndex = 0;

//loop through the data base

for(i = 0; i < 10; i = i+1) begin

//found a match for accNumber

if(accNumber == acc\_database[i]) begin

if(action == `FIND) begin

wasSuccessful = `true;

accIndex = i;

end

if(action == `AUTHENTICATE) begin

if(pin == pin\_database[i]) begin

wasSuccessful = `true;

accIndex = i;

end

end

end

end

end

endmodule

//

module ATM(

input clk,

input exit,

input [11:0] accNumber,

input [3:0] pin,

input [11:0] destinationAcc,

input [2:0]menuOption,

input [10:0] amount,

output reg error,

output reg [10:0] balance

);

//initializing the balance database with an arbitrary amount of money

reg [15:0] balance\_database [0:9];

initial begin

$display("Welcome to the ATM");

balance\_database[0] = 16'd500;

balance\_database[1] = 16'd500;

balance\_database[2] = 16'd500;

balance\_database[3] = 16'd500;

balance\_database[4] = 16'd500;

balance\_database[5] = 16'd500;

balance\_database[6] = 16'd500;

balance\_database[7] = 16'd500;

balance\_database[8] = 16'd500;

balance\_database[9] = 16'd500;

end

reg [2:0] currState = `WAITING;

wire [3:0] accIndex;

wire [3:0] destinationAccIndex;

wire isAuthenticated;

wire wasFound;

reg deAuth = `false;

authentication authAccNumberModule(accNumber, pin, `AUTHENTICATE, deAuth, isAuthenticated, accIndex);

authentication findAccNumberModule(destinationAcc, 0, `FIND, deAuth, wasFound, destinationAccIndex);

//main block of module with asynchronous exit

always @(posedge clk or isAuthenticated or menuOption or exit) begin

//restart the error

error = `false;

if(exit == `true) begin

//transition to the waiting state

currState = `WAITING;

//deathenticate the current user

deAuth = `true;

#20;

end

if(currState == `MENU) begin

//set the selected option as the current state

if((menuOption >= 0) & (menuOption <= 7))begin

currState = menuOption;

end else

currState = menuOption;

end

//switch case for the menu options

//the rest is pretty straight forward

case (currState)

`WAITING: begin

if (isAuthenticated == `true) begin

currState = `MENU;

$display("Logged In.");

end

else if(isAuthenticated == `false) begin

$display("Account number or password was incorrect");

currState = `WAITING;

end

end

`BALANCE: begin

balance = balance\_database[accIndex];

$display("Account %d has balance %d", accNumber, balance\_database[accIndex]);

currState = `MENU;

end

`WITHDRAW: begin

if (amount <= balance\_database[accIndex]) begin

balance\_database[accIndex] = balance\_database[accIndex] - amount;

balance = balance\_database[accIndex];

currState = `MENU;

error = `false;

end

else begin

currState = `MENU;

error = `true;

end

end

`WITHDRAW\_SHOW\_BALANCE: begin

if (amount <= balance\_database[accIndex]) begin

balance\_database[accIndex] = balance\_database[accIndex] - amount;

balance = balance\_database[accIndex];

currState = `MENU;

error = `false;

$display("Account %d has balance %d after withdrawing %d", accNumber, balance\_database[accIndex], amount);

end

else begin

currState = `MENU;

error = `true;

end

end

`TRANSACTION: begin

if ((amount <= balance\_database[accIndex]) & (wasFound == `true) & (balance\_database[accIndex] + amount < 2048)) begin

currState = `MENU;

error = `false;

balance\_database[destinationAccIndex] = balance\_database[destinationAccIndex] + amount;

balance\_database[accIndex] = balance\_database[accIndex] - amount;

$display("Destination account %d after transaction has a total balance of %d", destinationAcc, balance\_database[destinationAccIndex]);

end

else begin

currState = `MENU;

error = `true;

end

end

endcase

end

endmodule

**2. TEST BENCH:**

`define true 1'b1

`define false 1'b0

`define FIND 1'b0

`define AUTHENTICATE 1'b1

`define WAITING 3'b000

`define GET\_PIN 3'b001

`define MENU 3'b010

`define BALANCE 3'b011

`define WITHDRAW 3'b100

`define WITHDRAW\_SHOW\_BALANCE 3'b101

`define TRANSACTION 3'b110

`define DONE 3'b111

module atm\_tb();

reg clk, exit;

reg [11:0] accNumber;

reg [3:0] pin;

reg [11:0] destinationAccNumber;

reg [2:0] menuOption;

reg [10:0] amount;

wire error;

wire [10:0] balance;

ATM atmModule(clk, exit, accNumber, pin, destinationAccNumber, menuOption, amount, error, balance);

initial begin

clk = 1'b0;

end

always @(error) begin

if(error == `true)

$display("Error!, action causes an invalid operation.");

end

initial begin

//incorrect PIN

accNumber = 12'd2278;

pin = 4'b0100;

#30

//valid credentials

accNumber = 12'd2178;

pin = 4'b0100;

#30

//withdraw some money and then show the balance

amount = 100;

menuOption = `WITHDRAW\_SHOW\_BALANCE;

clk = ~clk;#5clk = ~clk;

#30

//show the balance

menuOption = `BALANCE;

clk = ~clk;#5clk = ~clk;

#30

//withdraw too much money, resulting in an error

amount = 2500;

menuOption = `WITHDRAW;

clk = ~clk;#5clk = ~clk;

#30

//the balance wont change because an error happened during withdrawal

menuOption = `BALANCE;

clk = ~clk;#5clk = ~clk;

#30

//transfer some money to the destination account with number 2816

amount = 50;

destinationAccNumber = 2816;

menuOption = `TRANSACTION;

clk = ~clk;#5clk = ~clk;

#30

//transfer too much money to the destination account with number 2816 which exceeds 2047 and cuases an error

amount = 2550;

destinationAccNumber = 2816;

menuOption = `TRANSACTION;

clk = ~clk;#5clk = ~clk;

#30

//exit the system

exit = 1;

#30

exit = 0;

#30

//log in using the account with number 2816

accNumber = 12'd2816;

pin = 4'b0110;

#30

//you'll see that the balance is more than the default value because we had trasnsferred some money to this account a while ago

menuOption = `BALANCE;

clk = ~clk;#5clk = ~clk;

#30;

end

endmodule

**SIMULATION RESULTS**

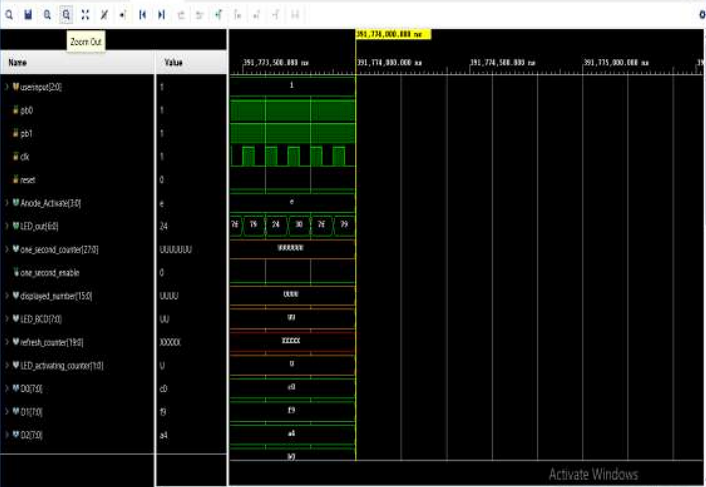
****

Fig.4 simulation

**FPGA IMPLEMENTATION**

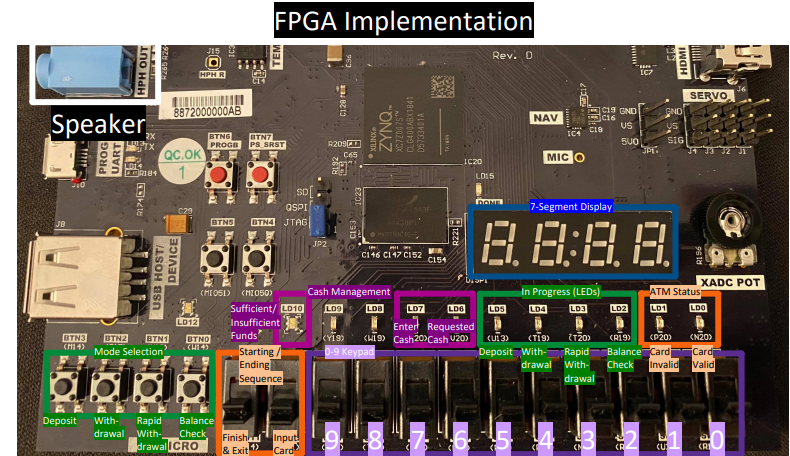


Fig.4. Implementation of Verilog code in FPGA