ECS2039 FINAL PROJECT

MICHAEL LOUGHRAN 40327859

Contents

Introduction

This project aims to develop a device using FPGA that will undergo the primary function of a self-checkout machine used in many stores in our modern world. To make this, we will use Xilinx's Vivado to write Verilog and test it, ensuring that it meets the required specification and produce reports and schematics of the design.
The code can be found using the following GitHub repository: 40327859 Final Project

1. The Accumulator Unit

This Unit will require that when the user adds a new item, it will update the current amount required for the purchase.

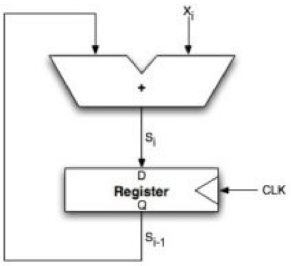

Figure 1. Diagram showing adder followed by register

1.1. Verilog for the Accumulator Unit.

The Verilog used for this is based upon **Figure 1** as shown below:

```
module Accumulator( clk, reset, xi, si);
input clk;
input reset;
input [15:0] xi;
output wire [15:0] si;
reg [15:0] n;
always @(posedge clk)
begin
    if (reset)
        n = 16'b0;
    else
        n = n + xi;
    end
assign si = n;
endmodule
```

1.2. Waveform for the Accumulator Unit.

To test the accumulator, we need to use the provided testbench, which was:

```
module Accumulator_Test;
reg clk_t,reset_t;
reg[15:0] value_t;
wire[15:0] sum_t;
Accumulator test(.clk(clk_t),.reset(reset_t),.xi(value_t),.si(sum_t));
always #10 clk_t = ~clk_t;
initial
begin
reset_t=1;
clk_t=0;
value_t=16'h8888;
#30;
reset_t=0;
value_t=16'h0001;
#30;
value_t=16'hffff;
#30;
value_t=16'h1111;
#30;
$stop;
end
endmodule
```

If we run a Behavioural simulation in Vivado, as shown in **Figure 2** we can see that $4369 + 4730 = 8739$ so this confirms that the Accumulator works correctly.

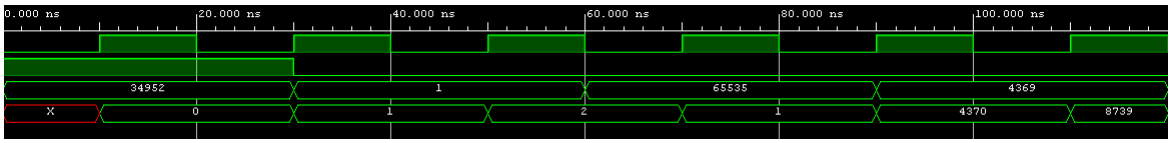

FIGURE 2. Screenshot showing the waveform of behavioural simulation of the Accumulator

1.3. **Synthesis and Implementation for the Accumulator.** If we run Synthesis and Implementation in Vivado, we get a waveform shown in **Figure 3**. The right part of the waveform is not shown. This is due to a timing delay when working with the actual hardware.

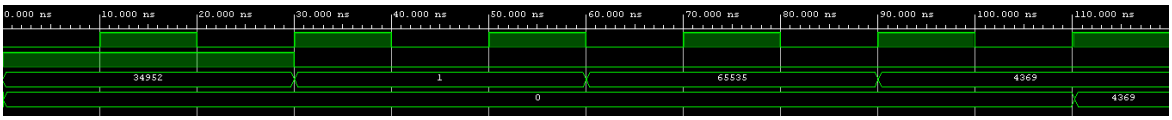

FIGURE 3. Screenshot of the waveform Post-Implementation Functional Simulation of the Accumulator

To compensate for this, we can add a delay of 100 ns which can be done in line 9 in the test bench provided in section 1.2. Which will produce the waveform shown in **Figure 4**

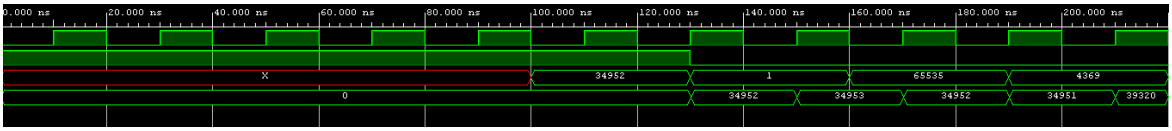

FIGURE 4. Screenshot of the waveform Post-Implementation Functional Simulation of the Accumulator

1.4. **RTL Schematic and Implementation Schematics for the Accumulator.** Using Vivado, we can produce the RTL schematic shown in **Figure 5**.

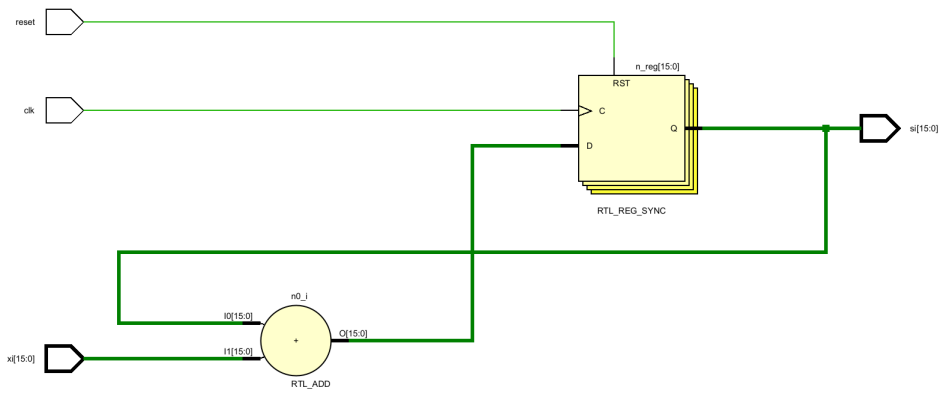

FIGURE 5. Image showing RTL schematic of the Accumulator

Similarly, we can produce the Implementation schematic shown in **Figure 6**

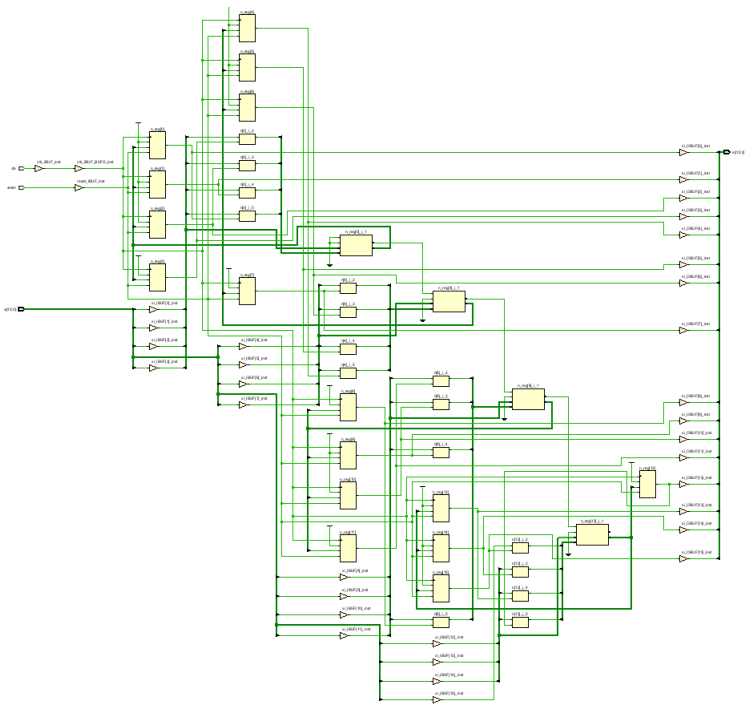

FIGURE 6. Image showing Implementation schematic of the Accumulator

Comparing both schematics, we can see that the implementation schematic is much larger and contains more components than RTL because Implementation looks at Gate level, which is a much lower abstraction hence the more features required in the schematic.

2. The Read Only Memory Unit

Now that the Accumulator has been completed, we need a way of storing the price of items. This can be accomplished by using a ROM, a code which will be then addressed to the ROM returning a value. We will use a 16-bit data field, giving a maximum value of $65535 = 2^{16}$, meaning the prices must be below this value. The Codes and prices are as follows:

TABLE 1. Codes for ROM

Code	Price
0	0
1	25
2	4
3	100
4	12
5	5
6	7
7	91
8	25
9	44
10	17
11	55
12	200
13	11
14	97
15	242

2.1. **Verilog for ROM.** Using the table provided we can incorporate this in Verilog, which is as shown:

```
module ROM(clk, address, data);
input clk;
input[3:0] address;
output reg[15:0] data;

always @ (posedge clk)
begin
case(address)
    4'b0000: data = 0;
    4'b0001: data = 25;
    4'b0010: data = 4;
    4'b0011: data = 100;
    4'b0100: data = 12;
    4'b0101: data = 5;
    4'b0110: data = 7;
    4'b0111: data = 91;
    4'b1000: data = 25;
    4'b1001: data = 44;
    4'b1010: data = 17;
    4'b1011: data = 55;
    4'b1100: data = 200;
    4'b1101: data = 11;
    4'b1110: data = 97;
    4'b1111: data = 242;
    default: data = 0;
    endcase
  end
endmodule
```

2.2. **Waveform and Test bench for the ROM Unit.** The following testbench was used to produce a waveform for the ROM:

```
module ROM_Test();
reg clk_tb;
reg[3:0] address_tb;
wire[15:0] data_tb;
ROM test(.clk(clk_tb),.address(address_tb),.data(data_tb));
always #10 clk_tb = ~clk_tb;
initial begin
clk_tb = 0;
#100
address_tb = 4'b0000;
#30
address_tb = 4'b0001;
#30
address_tb = 4'b0010;
#30
address_tb = 4'b0011;
#30
address_tb = 4'b0100;
#30
address_tb = 4'b0101;
#30
address_tb = 4'b0110;
#30
address_tb = 4'b0111;
#30
address_tb = 4'b1000;
#30
address_tb = 4'b1001;
#30
address_tb = 4'b1010;
#30
address_tb = 4'b1011;
#30
address_tb = 4'b1100;
#30
address_tb = 4'b1101;
#30
address_tb = 4'b1110;
#30
address_tb = 4'b1111;
#30
$stop;
end
endmodule
```

Using Behavioural Simulation, this testbench produced the waveform shown in **Figure 7** we can see that it does indeed match **Table 1**

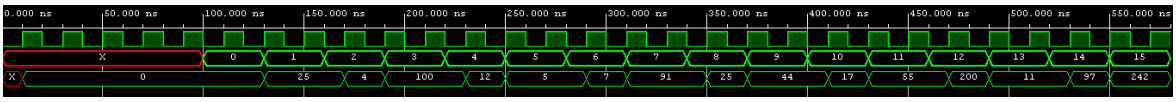

FIGURE 7. Screenshot of the Behavioural Simulation of the ROM Simulation of the Accumulator

2.3. **Synthesis and Implementation for the ROM.** If we run Synthesis and Implementation in Vivado, we can obtain reports which provide Data as shown in **Table 2**

TABLE 2. Data from Synthesis and Implementation ROM Reports

Parameter	Value
Total Number of used Slices	12
Total Number of used LUTs	4
Total Number of used FLIPFLOPS	8
Maximum Delay Path	4.087 ns
Total Power Consumption	2.981 W

3. The Customized Subtraction Unit

The design requires a subtraction unit which will calculate the difference in the amount owed and the account balance which will output the change value and signal success as 1, however we need to ensure that if the account balance is less than the amount owed the change output will be 0 and signal success will be set to 0.

3.1. **Verilog for the Subtraction Unit.** Which can be seen below:

```
module Subtraction (
    input [15:0] money,
    input [15:0] balance,
    output wire [15:0] change,
    output wire success
);

assign change = (money >= balance) ? (money - balance) : 16'h0;
assign success = (money >= balance) ? 1'b1 : 1'b0;

endmodule
```

3.2. **Post-Synthesis Functional and Implementation Timing Simulations for Subtraction Unit.** Using the test bench provided below we can run simulations to obtain the required waveforms.

```
module Subtraction_Test();
reg[15:0] money_t, balance_t;
wire[15:0] change_t;
wire success_t;
Subtraction test(.money(money_t),.balance(balance_t),
.change(change_t),.success(success_t));
initial begin
balance_t =16'd10;
money_t =16'd15;
#50;
balance_t = 16'd10;
money_t = 16'd7;
#50;
$stop;
end
endmodule
```

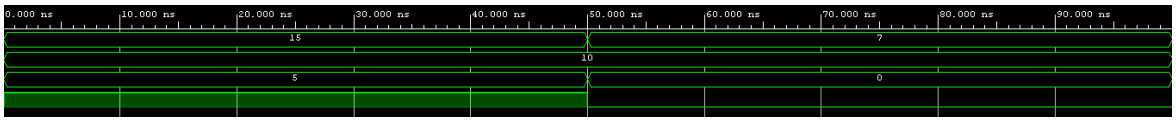

FIGURE 8. Screenshot of the Post-Synthesis Functional Timing Simulation of the Subtraction Unit

The Post-Synthesis Functional Timing Simulation can be seen in **Figure 8**
Running the Post-Implementation Timing Simulations, we get as shown in **Figure 9**

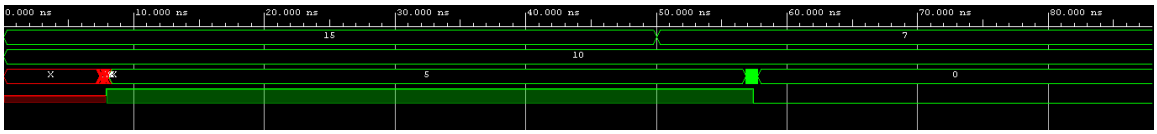

FIGURE 9. Screenshot of the Post-Implementation Timing Simulation of the Subtraction Unit

Within **Figure 9**, we can further zoom into the two blocks in around the 7 ns **Figure 11** and 57 ns **Figure 11**, which give the following:

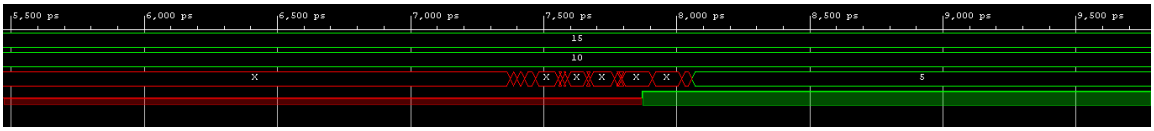

FIGURE 10. Zoomed Screenshot of the Post-Implementation Timing Simulation of the Subtraction Unit around 7 ns

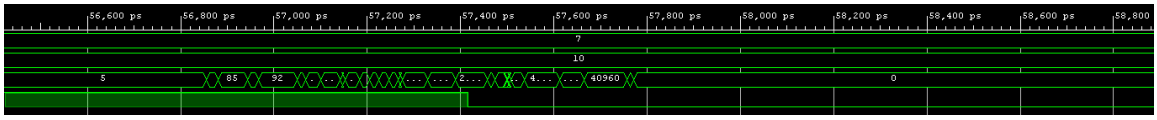

FIGURE 11. Zoomed Screenshot of the Post-Implementation Timing Simulation of the Subtraction Unit around 57 ns

When we Compare both simulations, we can see that Post-Implementation has some instability compared to the Post-Synthesis Simulation. This is because the simulation has implemented the design onto the selected hardware, which will consider real-world delays and disturbances. Nevertheless, the function is still correct in both instances.

4. Finite State Machine

With the other modules created we need to instantiate them to that they will all work together accordingly. We will be using a rising edge clock and the system will be based on the Finite State Machine shown in **Figure 12**

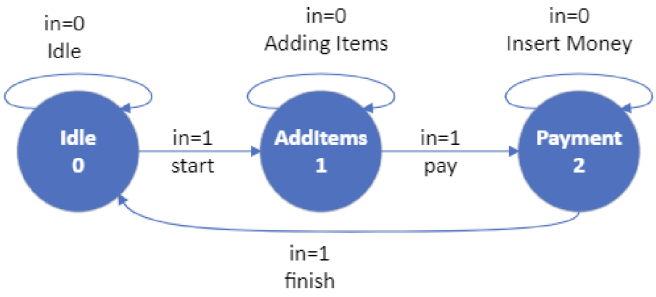

Figure 12. Image showing the FSM of the Self Checkout Machine

4.1. **Verilog for FSM.** Which can be seen below:

```
module FSM (
    input clk, reset, in,
    input [3:0] code,
    input [15:0] money,
    output [15:0] balance, change,
    output reg [1:0] state
);

parameter idle = 0, additems = 1, payment = 2;

wire [15:0] rom_data;
ROM rom_inst (
    .clk(clk),
    .address(code),
    .data(rom_data)
);

wire [15:0] total_cost;
Accumulator accumulator_inst (
    .clk(clk),
    .reset(reset),
    .xi((state == additems) ? rom_data : 16'b0),
    .si(total_cost)
);

wire success;
Subtraction subtraction_inst (
    .money(money),
    .balance(total_cost),
    .change(change),
    .success(success)
);

always @(posedge clk or posedge reset) begin
    if (reset) begin
        state <= idle;
    end else begin
        case (state)
            idle: begin
                if (in) state <= additems;
            end
            additems: begin
                if (in) state <= payment;
            end
            payment: begin
                if (in && success) state <= idle;
            end
        endcase
    end
end
assign balance = total_cost;
endmodule
```

4.2. **Behaviour Simulation Waveform for the FSM.** To test the system, we need to use the provided testbench, which was:

```
module FSM_Test;
reg clk_t, reset_t, in_t;
reg[3:0] code_t;
reg[15:0] money_t;
wire[15:0] balance_t, change_t;
wire[1:0]state_t;
FSM test(.clk(clk_t),.reset(reset_t),.in(in_t),.code(code_t),
.money(money_t),.balance(balance_t),.change(change_t),.state(state_t));
always #5  clk_t=~clk_t;
initial begin
reset_t = 1;
clk_t = 0;
in_t = 0;
money_t = 16'd0;
#100;
in_t= 1;
reset_t= 0;
#10;
in_t = 0;
code_t=4'd1;
#10;
code_t=4'd2;
#10;
code_t=4'd3;
#10;
code_t=4'd4;
#15;
in_t = 1;
money_t=16'hffff;
#10;
$stop;
end
endmodule
```

If we run a Behavioural Simulation on the testbench, we get the following waveform:

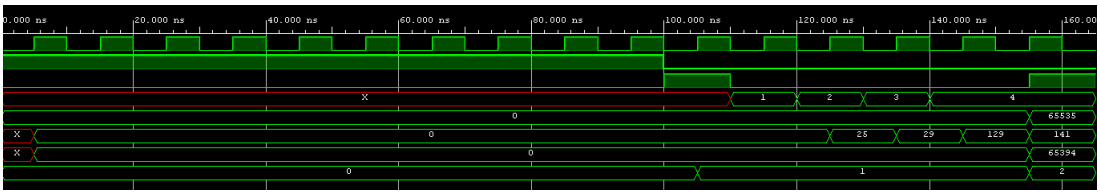

FIGURE 13. Zoomed Screenshot of the Post-Implementation Timing Simulation of the Subtraction Unit around 7 ns

4.3. **RTL Schematic Schematics for the FSM.** As shown in **Figure 1**

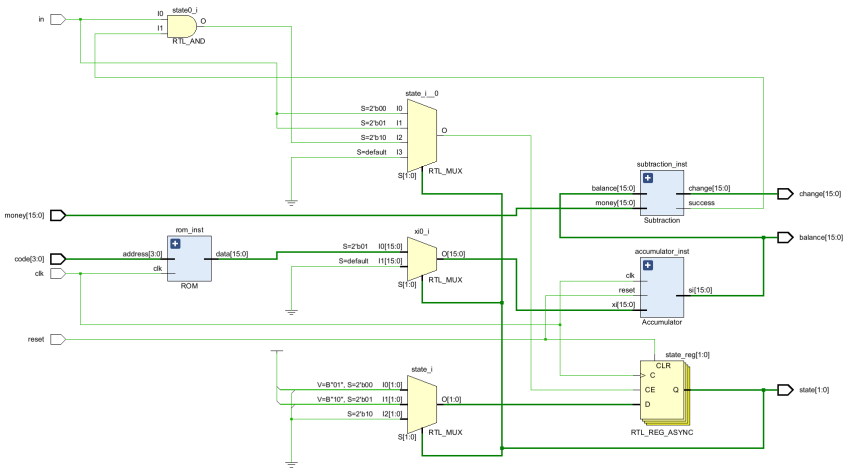

FIGURE 14. RTL schematic of the FSM

If we expand this, we get what is shown in **Figure 15**

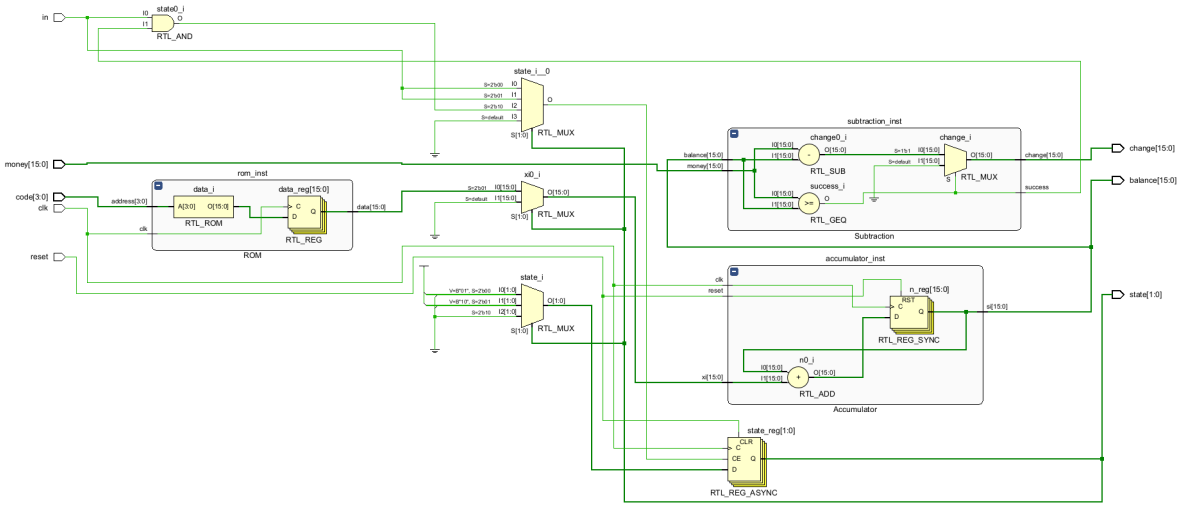

FIGURE 15. RTL schematic of the FSM

4.4. **Post-Synthesis Functional and Post-Implementation Timing Simulations for FSM.** If we run Synthesis and Implementation in Vivado, we can obtain reports which provide Data as shown in **Table 3**.

TABLE 3. Data from Synthesis and Implementation ROM Reports

Parameter	Value
Total Number of used Slices	73
Total Number of used LUTs	46
Total Number of used FLIPFLOPS	27
Maximum Delay Path	8.573 ns
Total Power Consumption	21.133 W

CONCLUSION

In conclusion, the FPGA-based self-checkout machine project seeks to revolutionize the retail industry by providing a cost-effective, flexible, and easily adaptable solution for streamlining the checkout process. Leveraging Xilinx Vivado and Verilog for the design and testing phases ensures that the final product will be reliable. Future improvements could be that the system could have a countermeasure to avoid the flaw in the FSM where if the user could keep setting it to 1, allowing them to bypass the payment phase, which will cost the client money. Another improvement could be that the chip's efficiency could be improved to a lower wattage.