

San Jose State University
Department of Computer Engineering

CMPE 140 Lab Report

Lab 1 Report

Title System-Level Design Review

Semester SPRING 2019

Date 02/05/19

by





Name Nickolas Schiffer

SID 012279319

Name Salvatore Nicosia

SID 012013599

Lab Checkup Record

Week	Performed By (signature)	Checked By (signature)	Tasks Successfully Completed*	Tasks Partially Completed*	Tasks Failed or Not Performed*
1	SN 		100%		
2	 SN		100%		

* Detailed descriptions must be given in the report.

I. INTRODUCTION

The purpose of this lab is to review system-level design by designing, functionally verifying, and FPGA prototyping a digital system for computing the factorial of n . This system was functionally verified by designing a Data Path (DP) and also by designing using the ASM chart the Control Unit (CU). This was achieved through the complete FPGA implementation procedure and validation using Digilent's Nexys4 DDR board.

II. DESIGN METHODOLOGY

There are two parts to this lab assignment. Part one is to design and test the control unit and data path for computing the factorial of n . The second part is to design, test, and build the fully integrated factorial generator that contains both the control unit and data path. See Figure 1 for the factorial generator module that combines the control unit and data path. The system starts executing when the input "Go" is received and will output a "Done" signal when the execution is completed. When the system detects that the input entered is greater than 12 an "Err" signal will be set indicating the error.

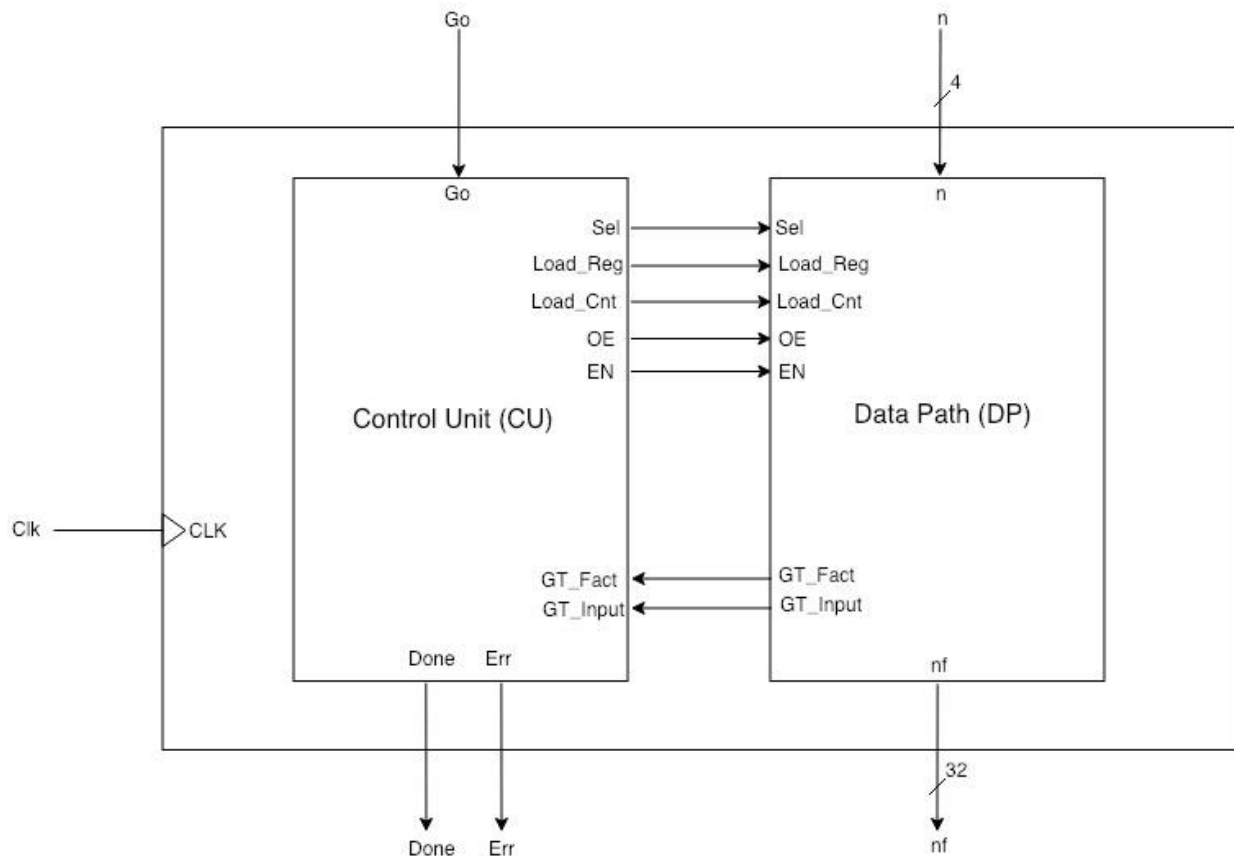


Figure 1: System's CU-DP model.

Data Path:

The data path is a module that has one 4-bit counter input, one 4-bit comparator input used to check if the input is greater than 12. It also contains several input signals that control various sub-modules within the data path design. One input n is connected to a down counter (CNT) with parallel load control and an enable signal in parallel the input n is connected to the comparator to verify that $n \leq 12$. A 2-to-1 mux is connected to the D-input of a data register with a load control signal. There is a comparator (CMP) module connected to the outputs of the down counter with a GT (greater-than) output. In addition, the output of the down counter is connected to combinational multiplier (MUL) for unsigned integers. Lastly BUF is 2-to-1 MUX used for output control. There are three output signal that are sent to the control unit. These signals are GT_Input (High if $n > 12$), GT_Fact (High if greater than count), and $n!$ (for the output of the factorial). See Table 1 for the I/O definitions. See Figure Figure 2 for the data path micro-architecture.

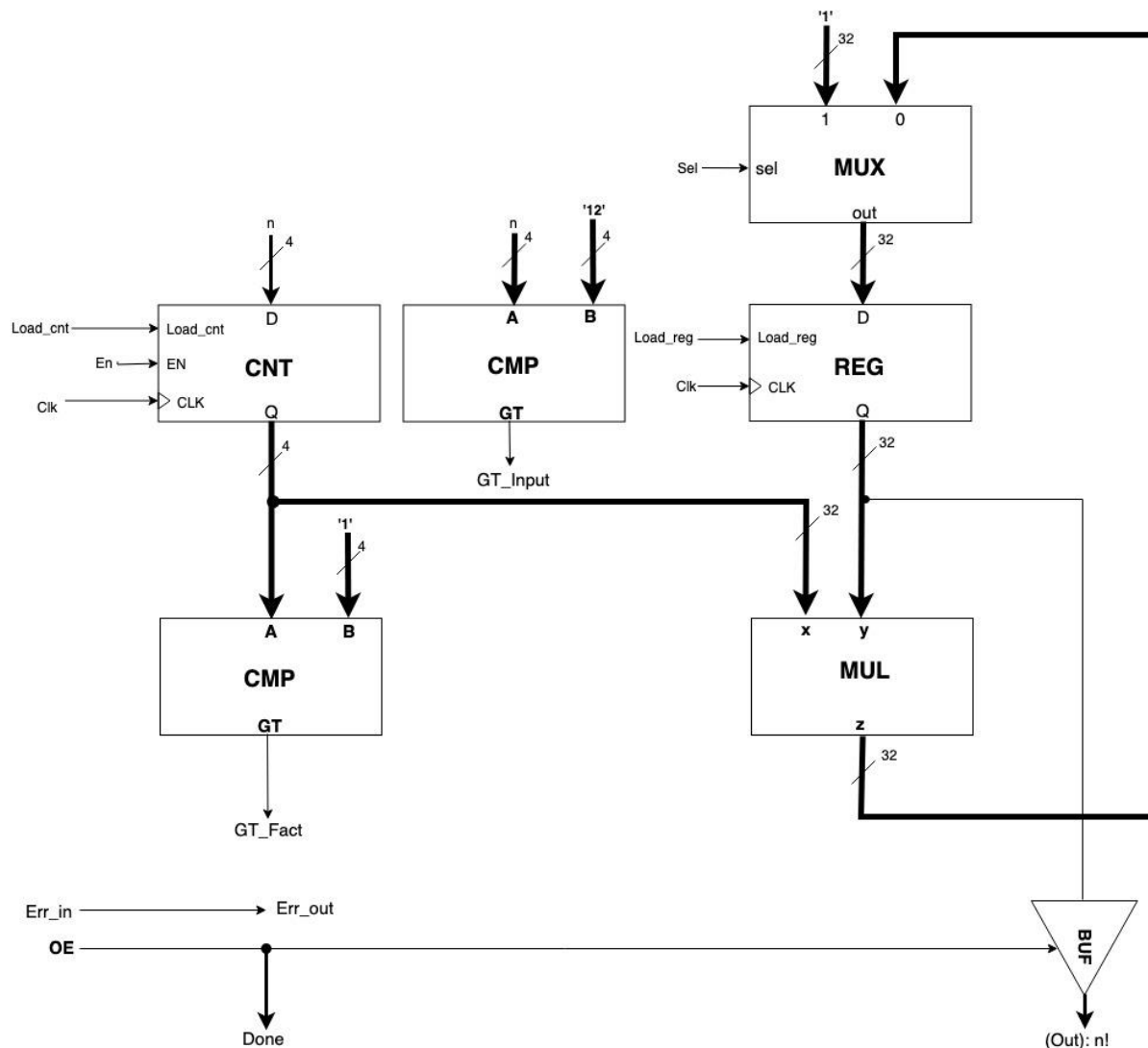


Figure 2: Micro-Architecture of the Data Path

Control Unit:

The control unit is a module that has one control input Go that starts the state machine. There are also three input status signals, GT_Fact, GT_Input. See the data path section for an explanation of their logic. The control unit is a four-state mixes Moore and Mealy finite-state-machine. The state machine's next-state logic cycles through the four states at the rising edge of the clock. At each state, the outputs from the control unit are changed to control in the modules within the data path. The ASM chart and state transition diagram can be found in section C of the appendix Figures 24 and 25, respectively. See Table 2 for the control unit's I/O definitions. Table 3 shows the output function table for the control unit. Figure 3 shows the FSM diagram.

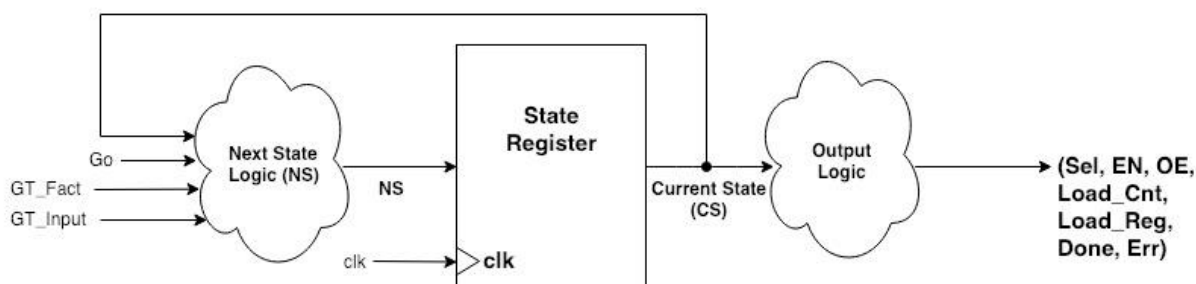


Figure 3: FSM diagram of the Control Unit

Table 1. I/O Definition for Data Path

I/O Signals	Size	Type	Definition
n	4	In	Input data
GT_Input	1	Out	Status Signal from DP
GT_Fact	1	Out	Status Signal from DP
nf	1	Out	Status Signal from DP
Sel	1	In	Mux select signal
Load_Reg	1	In	Control signal: Load_reg
Load_Cnt	1	In	Control signal: Load_cnt
OE	1	In	Output Enable (control)
En	1	In	Enable (control)
clk	1	In	Clock (control)

Table 2. I/O Definition for Control Unit

I/O Signals	Size	Type	Definition
Go	1	In	Input signal
GT_Fact	1	In	Status Signal from DP
GT_Input	1	In	Status Signal from DP
Sel	1	Out	Mux select signal
Load_Reg	1	Out	Control signal: Load_Reg
Load_Cnt	1	Out	Control signal: Load_Cnt
OE	1	Out	Output Enable (control)

En	1	Out	Enable (control)
clk	1	Out	Clock (control)
Done	1	Out	Done signal
Err	1	Out	Error signal

Table 3. Function Table for Control Unit (FSM)

State	Outputs						
	Sel	Load_Cnt	En	Load_Reg	OE	Done	Err
S0	1	1	1	0	0	0	-
S1	1	1	1	1	0	0	-
S2	0	0	0	0	0	0	-
S3	0	0	0	0	1	1	-
S4	0	0	1	1	0	0	-

Table 4. List of Modules and Files Used

Module/File Name	Comments
<i>Factorial_DP.v</i>	Data Path for the Factorial Generator
<i>Factorial_CU.v</i>	Control Unit for the Factorial Generator
<i>Factorial_FPGA.v</i>	Designed module with the function shown in Figure 7
<i>Factorial_Top_tb.v</i>	Test bench file for the designed module shown in Figure 4
<i>Factorial_Top.v</i>	Top-level module for the system shown in Figure 17
<i>Factorial_Top_FPGA.xdc</i>	Design constraint file for Integer divider
<i>CNT.v</i>	Designed module with the function shown in Figure 11
<i>MUX2.v</i>	Designed module with the function shown in Figure 12
<i>Mag_Comp.v</i>	Designed module with the function shown in Figure 15/16
<i>Mult.v</i>	Designed module with the function shown in Figure 13
<i>Reg.v</i>	Designed module with the function shown in Figure 14
<i>bin32_to_8hex.v</i>	Designed module with the function shown in Figure 23
<i>button_debouncer.v</i>	Designed module to manually advance the clock for the registers shown in Figure 20
<i>clk_gen.v</i>	Utility module for converting the on-board clock to 5KHz shown in Figure 19
<i>hex_to_7seg.v</i>	Utility module for converting from hex to 7-segment shown in Figure 21
<i>led_mux.v</i>	Utility module for multiplexing signals to be displayed on the eight 7-segments LEDs on the Nexys4 DDR board shown in Figure 22

III. TESTING PROCEDURE

Factorial Generator:

In order to test the logic for the factorial generator, a simulation test bench is created to test the *Factorial_Top.v* module. All combinations of inputs are tested. The clock is then ticked through all states and the outputs are sent to the BUF output port. The Done signal goes high when the operation is complete. If an error occurs (essentially when $n > 12$), the error flag goes high. The expected outputs are verified at each step of the process. The testbench used is shown in Figure 4. Samples of the waveform of the simulation can be found in Figure 5 and 6 in the testing results section.

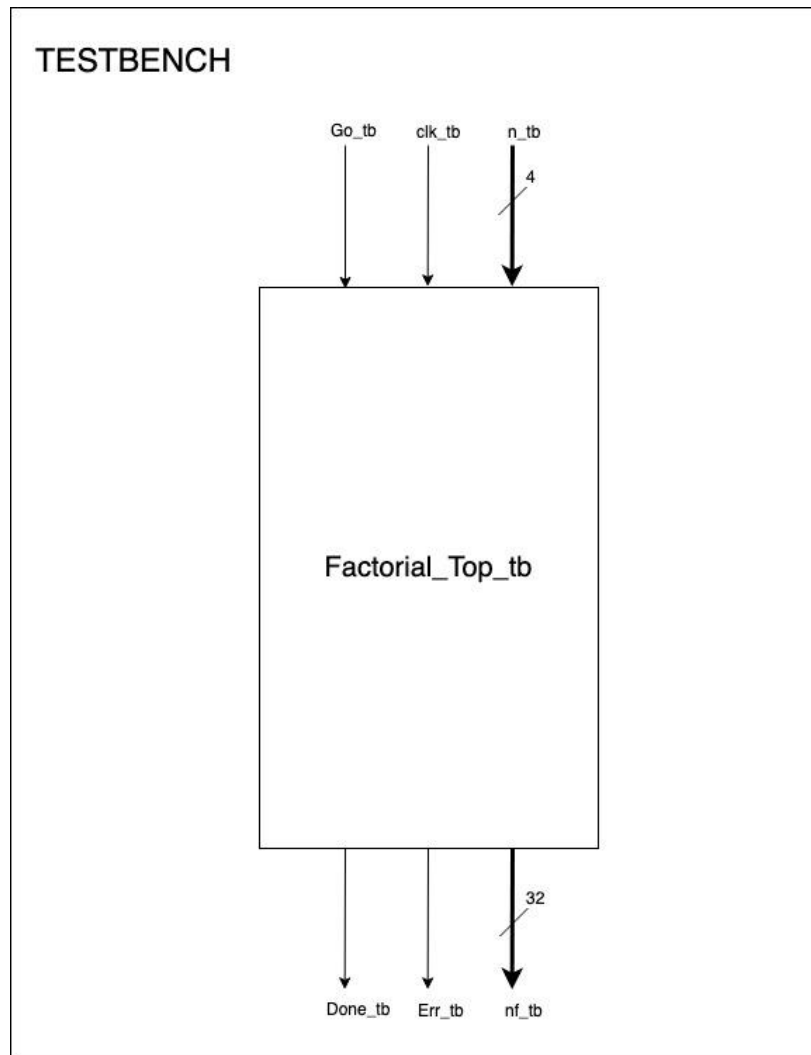


Figure 4: module Factorial_Top_tb.v: Factorial Generator TestBench

IV. TESTING RESULTS



Figure 5: Factorial_Top_tb output 1
Successful Factorial Generation

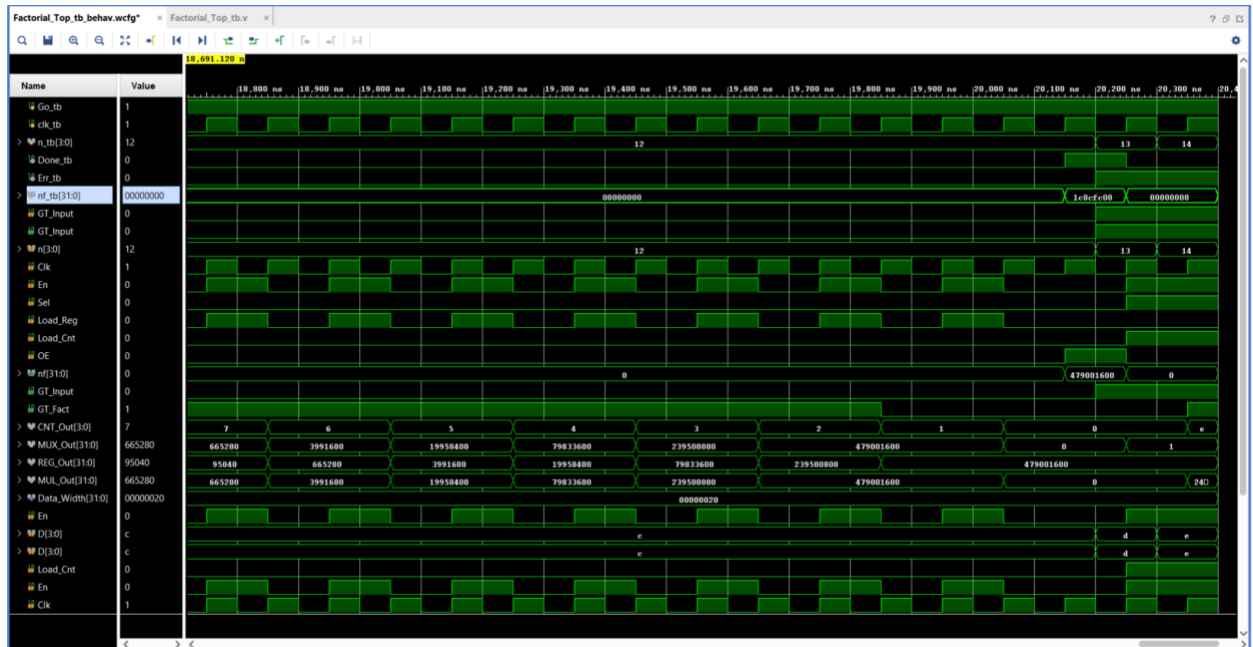


Figure 6: Factorial_Top_tb output 2
Error Condition

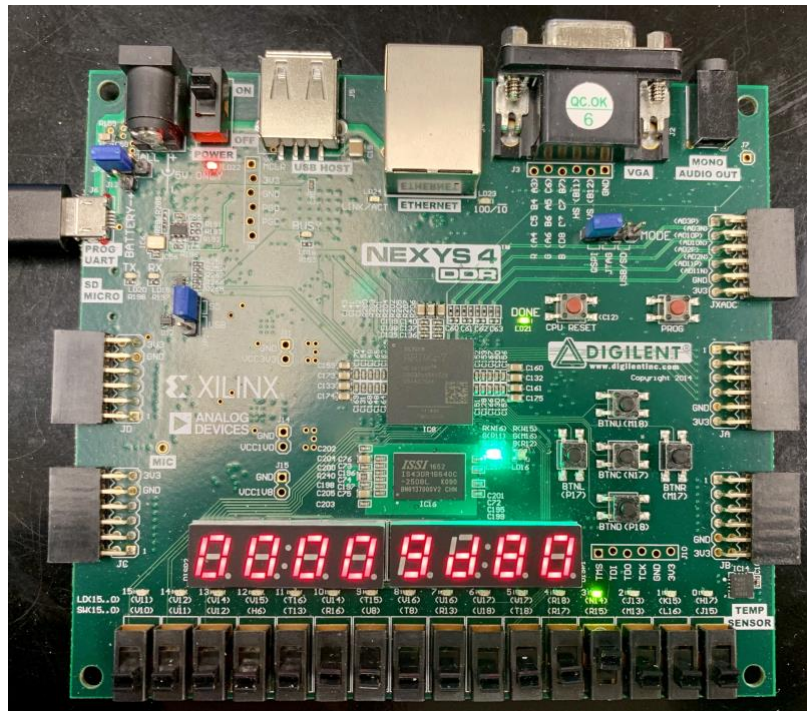


Figure 8: Factorial Generator showing the result of 8!
 $n = 8$, $n! = 9d80$, Done = 1, Err = 0

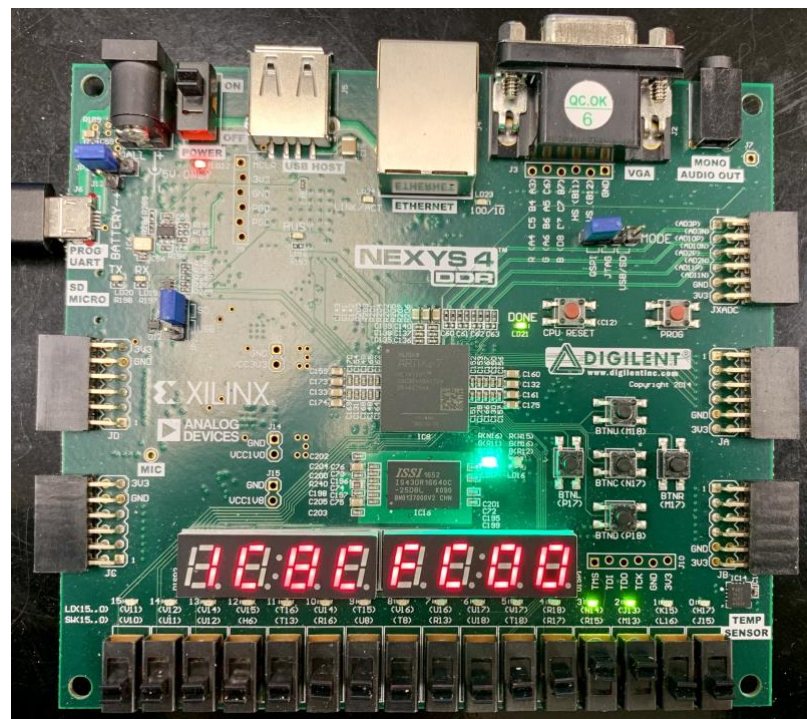


Figure 9: Factorial Generator showing the result of 12!
 $n = 12$, $n! = 1c8cfc00$, Done = 1, Err = 0

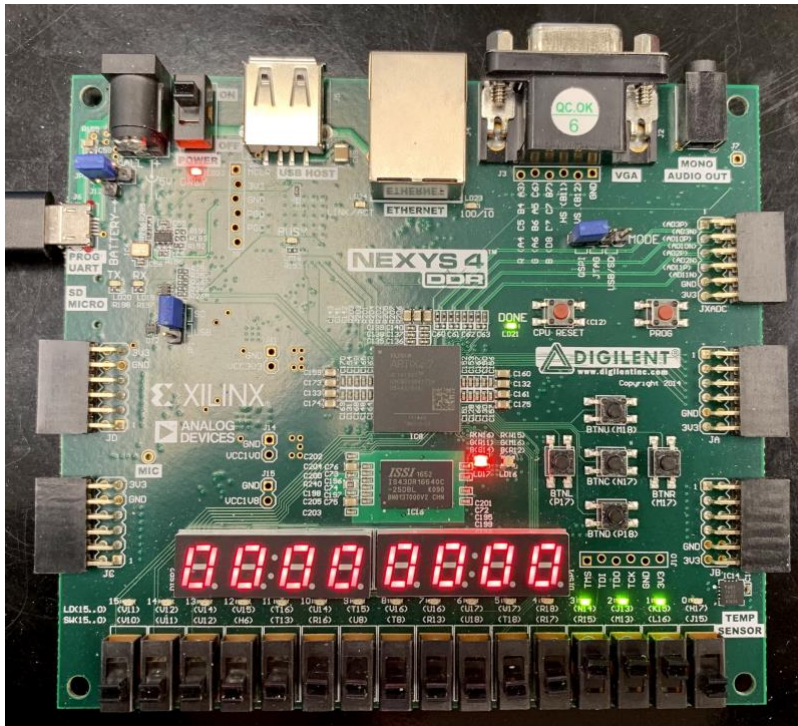


Figure 10: Factorial Generator showing the result of 14!
 $n = 14$, $n! = 0$, Done = 0, Err = 1

VII. CONCLUSION

This lab taught us how to create a 4-bit factorial generator accelerator unit for later use in our MIPS processor design. This was accomplished via creating a control unit and data path pair which are comprised of muxes, a counter, and a register that take in a 4-bit integer (less than 13) and output the factorial result along with a done or error signal. The control unit is driven by a state machine using current and next state registers that depend on the inputs from the data path. The design was thoroughly tested via a top-level test bench and then used in an FPGA implementation that was also physically tested. All of the objectives of the lab were accomplished and verified.

VIII. SUCCESSFUL TASKS

1. Design entry into Verilog HDL for the data path and control module.
2. Functional verification through simulation of design for the top-level design.
3. Hardware validation using Artix-7 FPGA Board for the factorial generator accelerator.

IX. APPENDIX

A. SOURCE CODE:

Factorial_DP.v

```
module Factorial_DP #(parameter Data_Width = 32) (
    input [3:0] n,
    input Clk, En, Sel, Load_Reg, Load_Cnt, OE,
    output [Data_Width - 1:0] nf,
    output GT_Input, GT_Fact
);

    wire [3:0] CNT_Out;
    wire [Data_Width - 1:0] MUX_Out;
    wire [Data_Width - 1:0] REG_Out;
    wire [Data_Width - 1:0] MUL_Out;

    CMP_GT # (4) CMP_INPUT (.A(n), .B(4'd12), .GT(GT_Input));
    CMP_GT # (4) CMP_FACT (.A(CNT_Out), .B(4'b1), .GT(GT_Fact));
    CNT # (4) COUNTER (.D(n), .Q(CNT_Out), .Load_Cnt(Load_Cnt), .En(En), .Clk(Clk));
    MUL # (32) MUL (.x({28'b0,CNT_Out}), .y(REG_Out), .z(MUL_Out));
    MUX2 # (32) MUX (.D1(32'b1), .D0(MUL_Out), .Sel(Sel), .Out(MUX_Out));
    MUX2 # (32) OUT_BUFFER (.D0(32'b0), .D1(REG_Out), .Sel(OE), .Out(nf));
    REG # (32) REG (.D(MUX_Out), .Q(REG_Out), .Load_Reg(Load_Reg), .Clk(Clk));

endmodule
```

Factorial_CU.v

```
module Factorial_CU(
    input go, clk,
    input GT_Fact, GT_Input,
    output reg Sel, Load_Reg, Load_Cnt, OE, En,
    output Done, Err
);

    //Next and Current State Registers
    reg [2:0] CS = 0, NS;
    reg Err_Internal = 0, Done_Internal = 0;

    //assign Err = Err_Internal;
    assign Err = GT_Input;
    assign Done = Done_Internal;

    //encode states
    parameter Idle = 3'd0,
           Load_Cnt_AND_Reg = 3'd1,
           Wait = 3'd2,
           OE_AND_Done = 3'd3,
           Mult_AND_Dec = 3'd4;

    //Next State Logic (combinational) based on State Transition Diagram
    always @ (CS, go)
    begin
        case (CS)
            Idle:
                case({go,GT_Input})
                    2'b11: {NS,Err_Internal} <= {Idle,1'b1};
                    2'b10: {NS,Err_Internal} <= {Load_Cnt_AND_Reg,1'b0};
                    2'b0?: {NS,Err_Internal} <= {Idle,1'b0};
                    default: NS = Idle;
                endcase
            Load_Cnt_AND_Reg: NS <= Wait;
            //Wait2: NS <= Wait;
        endcase
    end
```

```

        Wait:                NS <= GT_Fact ? Mult_AND_Dec : OE_AND_Done;
        OE_AND_Done:         NS <= Idle;
        Mult_AND_Dec:        NS <= Wait;
    endcase
end

//State Register (sequential)
always @ (posedge clk)
    CS = NS;

//Output Logic
always @ (CS)
begin
    case (CS)
        Idle: //S0
        begin
            {Sel, Load_Cnt, En, Load_Reg, OE, Done_Internal} <= 6'b1_1_1_0_0_0;
        end
        Load_Cnt_AND_Reg: //S1
        begin
            {Sel, Load_Cnt, En, Load_Reg, OE, Done_Internal} <= 6'b1_1_1_1_0_0;
        end
        Wait: //S2
        begin
            {Sel, Load_Cnt, En, Load_Reg, OE, Done_Internal} <= 6'b0_0_0_0_0_0;
        end
        OE_AND_Done: //S3
        begin
            {Sel, Load_Cnt, En, Load_Reg, OE, Done_Internal} <= 6'b0_0_0_0_1_1;
        end
        Mult_AND_Dec: //S4
        begin
            {Sel, Load_Cnt, En, Load_Reg, OE, Done_Internal} <= 6'b0_0_1_1_0_0;
        end
    endcase
end

endmodule

```

Factorial_FPGA.v

```

module Factorial_FPGA(
    input go, clk100MHz, man_clk,
    input [3:0] n,
    output [7:0] LEDSEL, LEDOUT,
    output done, err,
    output [3:0] n_out
);

    supply1 [7:0] vcc;
    wire DONT_USE, clk_5KHz;
    wire debounced_clk;
    wire [31:0] nf;
    wire dummy;
    assign dummy = 0;
    assign n_out = n;

    // reg [7:0] LED0, LED1, LED2, LED3, LED4, LED5, LED6, LED7;
    wire [7:0] LED0, LED1, LED2, LED3, LED4, LED5, LED6, LED7;
    wire [3:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7;

    Factorial_Top Top (
        .Go(go),
        .clk(debounced_clk),
        .Done(done),
        .Err(err),
        .n(n),

```

```

        .nf(nf)
    );

    button_debouncer DBNC (
        .clk(clk_5KHz),
        .button(man_clk),
        .debounced_button(debounced_clk)
    );

    led_mux LED (
        .clk(clk_5KHz),
        .rst(dummy),
        .LED0(LED0),
        .LED1(LED1),
        .LED2(LED2),
        .LED3(LED3),
        .LED4(LED4),
        .LED5(LED5),
        .LED6(LED6),
        .LED7(LED7),
        .LEDSEL(LEDSEL),
        .LEDOUT(LEDOUT)
    );

    clk_gen CLK (clk100MHz, dummy, DONT_USE, clk_5KHz);

    bin32_to_8hex B2HEX (
        .value(nf),
        .dig0(HEX0),
        .dig1(HEX1),
        .dig2(HEX2),
        .dig3(HEX3),
        .dig4(HEX4),
        .dig5(HEX5),
        .dig6(HEX6),
        .dig7(HEX7)
    );

    hex_to_7seg H2LED0 (
        .HEX(HEX0),
        .s(LED0)
    );

    hex_to_7seg H2LED1 (
        .HEX(HEX1),
        .s(LED1)
    );

    hex_to_7seg H2LED2 (
        .HEX(HEX2),
        .s(LED2)
    );

    hex_to_7seg H2LED3 (
        .HEX(HEX3),
        .s(LED3)
    );

    hex_to_7seg H2LED4 (
        .HEX(HEX4),
        .s(LED4)
    );

    hex_to_7seg H2LED5 (
        .HEX(HEX5),
        .s(LED5)
    );

    hex_to_7seg H2LED6 (
        .HEX(HEX6),
        .s(LED6)
    );

```

```

);

hex_to_7seg H2LED7 (
    .HEX(HEX7),
    .s(LED7)
);
endmodule

```

Factorial_Top_tb.v

```

module Factorial_Top_tb;

reg Go_tb, clk_tb;
reg [3:0] n_tb;
wire Done_tb, Err_tb;
wire [31:0] nf_tb;
reg [31:0] nf_inf;

Factorial_Top DUT (
    .n(n_tb),
    .Go(Go_tb),
    .clk(clk_tb),
    .Done(Done_tb),
    .Err(Err_tb),
    .nf(nf_tb)
);

task automatic tick;
begin
    clk_tb <= 1'b0;
    #50;
    clk_tb <= 1'b1;
    #50;
end
endtask

function automatic [31:0] factorial;
input [3:0]n;
reg [31:0]nl, nf;
begin
    nl = n;
    nf = 1;
    while (nl > 1)
    begin
        nf = nf * nl;
        nl = nl - 1;
    end
    factorial = nf;
end
endfunction

initial
begin
    $display("Factorial Top Test Begin");
    clk_tb = 0;
    n_tb = 4'd1;
    tick;
    while (n_tb < 15)
    begin
        Go_tb = 1;
        clk_tb = 0;
        tick;
        while (!(Done_tb || Err_tb))
        begin
            tick;
        end
    end
end

```

```

        if (Done_tb)
        begin
            $display("%0d!      = %0d",n_tb, nf_tb);
            nf_inf = factorial(n_tb);
            $display("%0d! inf = %0d\n",n_tb, nf_inf);
        end
        else if (Err_tb)
        begin
            $display("Error received, input = %0d", n_tb);
        end
        if ((nf_tb != nf_inf) && (n_tb <= 12))
        begin
            $display("Error: got %0d, expected %0d for $0d!",nf_tb, nf_inf, n_tb);
            $stop;
        end

        n_tb = n_tb + 4'd1;
    end

    $display("Test Complete");
    $finish;
end

endmodule

```

Factorial_Top.v

```

module Factorial_Top(
    input Go, clk,
    input [3:0] n,
    output Done, Err,
    output [31:0] nf
);

    wire Sel, Load_Reg, Load_Cnt, OE, En;
    wire GT_Fact, GT_Input;

    Factorial_DP DP (
        .Clk(clk),
        .En(En),
        .n(n),
        .Sel(Sel),
        .Load_Reg(Load_Reg),
        .Load_Cnt(Load_Cnt),
        .OE(OE),
        .GT_Fact(GT_Fact),
        .GT_Input(GT_Input),
        .nf(nf)
    );

    Factorial_CU CU (
        .En(En),
        .go(Go),
        .clk(clk),
        .Sel(Sel),
        .Load_Reg(Load_Reg),
        .Load_Cnt(Load_Cnt),
        .OE(OE),
        .GT_Fact(GT_Fact),
        .GT_Input(GT_Input),
        .Done(Done),
        .Err(Err)
    );
endmodule

```

Factorial_Top_FPGA.xdc

```
#Clock
set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports {clk100MHz}];
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk100MHz}];

#switches
#n
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports {n[0]}];
set_property -dict {PACKAGE_PIN L16 IOSTANDARD LVCMOS33} [get_ports {n[1]}];
set_property -dict {PACKAGE_PIN M13 IOSTANDARD LVCMOS33} [get_ports {n[2]}];
set_property -dict {PACKAGE_PIN R15 IOSTANDARD LVCMOS33} [get_ports {n[3]}];

#Buttons
set_property -dict {PACKAGE_PIN P18 IOSTANDARD LVCMOS33} [get_ports {go}];
set_property -dict {PACKAGE_PIN M18 IOSTANDARD LVCMOS33} [get_ports {man_clk}];

#LEDs
#n!
set_property -dict {PACKAGE_PIN K13 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[0]}];
set_property -dict {PACKAGE_PIN K16 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[1]}];
set_property -dict {PACKAGE_PIN P15 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[2]}];
set_property -dict {PACKAGE_PIN L18 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[3]}];
set_property -dict {PACKAGE_PIN R10 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[4]}];
set_property -dict {PACKAGE_PIN T11 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[5]}];
set_property -dict {PACKAGE_PIN T10 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[6]}];
set_property -dict {PACKAGE_PIN H15 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[7]}];

set_property -dict {PACKAGE_PIN J17 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[0]}];
set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[1]}];
set_property -dict {PACKAGE_PIN T9 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[2]}];
set_property -dict {PACKAGE_PIN J14 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[3]}];
set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[4]}];
set_property -dict {PACKAGE_PIN T14 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[5]}];
set_property -dict {PACKAGE_PIN K2 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[6]}];
set_property -dict {PACKAGE_PIN U13 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[7]}];

#Inputs out
#n
set_property -dict {PACKAGE_PIN H17 IOSTANDARD LVCMOS33} [get_ports {n_out[0]}];
set_property -dict {PACKAGE_PIN K15 IOSTANDARD LVCMOS33} [get_ports {n_out[1]}];
set_property -dict {PACKAGE_PIN J13 IOSTANDARD LVCMOS33} [get_ports {n_out[2]}];
set_property -dict {PACKAGE_PIN N14 IOSTANDARD LVCMOS33} [get_ports {n_out[3]}];

#Done/Err
#
set_property -dict {PACKAGE_PIN V11 IOSTANDARD LVCMOS33} [get_ports {done}];
#
set_property -dict {PACKAGE_PIN V12 IOSTANDARD LVCMOS33} [get_ports {err}];

set_property -dict {PACKAGE_PIN R11 IOSTANDARD LVCMOS33} [get_ports {done}]; //Green
set_property -dict {PACKAGE_PIN N16 IOSTANDARD LVCMOS33} [get_ports {err}]; //Red
```

CNT.v

```
module CNT #(parameter Data_Width = 4)(
    input [Data_Width - 1:0] D,
    input Load_Cnt, En, Clk,
    output reg [Data_Width - 1:0] Q
);

always @ (posedge Clk)
begin
    if (En)
        begin
            if (Load_Cnt)
                Q <= D;
            else
                begin
                    Q <= Q - 1'b1;
                end
        end
end
```



```
        end
    end
endmodule
```

MUX2.v

```
module MUX2 #(parameter Data_Width = 32) (
    input [Data_Width - 1:0] D0, D1,
    input Sel,
    output reg [Data_Width - 1:0] Out
);

    always @ (D0, D1, Sel)
        begin
            if (Sel)
                Out <= D1;
            else
                Out <= D0;
            end
        end

endmodule
```

Mag_Comp.v

```
module CMP_GT #(parameter Data_Width = 4) (
    input [Data_Width - 1:0] A, B,
    output reg GT);

    always @ (A or B)
        begin
            // GT = (A > B) ? 1'b1 : 1'b0;
            GT <= 1'b0;

            if (A > B)
                GT <= 1'b1;
            end
        end

endmodule
```

Mult.v

```
module MUL #(parameter Data_Width = 32) (
    input [Data_Width - 1:0] x, y,
    output reg [Data_Width - 1:0] z
);

    always @ (x or y)
        begin
            z <= x * y;
        end

endmodule
```

Reg.v

```
module REG #(parameter Data_Width = 32) (  
    input [Data_Width - 1:0] D,  
    input Clk, Load_Reg,  
    output reg [Data_Width - 1:0] Q  
);  
  
    always @ (posedge Clk)  
        begin  
            if (Load_Reg)  
                Q <= D;  
        end  
endmodule
```

bin32_to_8hex.v

```
module bin32_to_8hex(  
    input wire [31:0] value,  
    output wire [3:0] dig0,  
    output wire [3:0] dig1,  
    output wire [3:0] dig2,  
    output wire [3:0] dig3,  
    output wire [3:0] dig4,  
    output wire [3:0] dig5,  
    output wire [3:0] dig6,  
    output wire [3:0] dig7  
);  
    //      assign dig0 = value % 10;  
    //      assign dig1 = (value / 10) % 10;  
    //      assign dig2 = (value / 100) % 10;  
    //      assign dig3 = (value / 1000) % 10;  
    //      assign dig4 = (value / 10000) % 10;  
    //      assign dig5 = (value / 100000) % 10;  
    //      assign dig6 = (value / 1000000) % 10;  
    //      assign dig7 = (value / 10000000) % 10;  
  
    // assign dig0 = value          % (1 << 4);  
    // assign dig1 = (value / (1 << (4 * 1))) % (1 << 4);  
    // assign dig2 = (value / (1 << (4 * 2))) % (1 << 4);  
    // assign dig3 = (value / (1 << (4 * 3))) % (1 << 4);  
    // assign dig4 = (value / (1 << (4 * 4))) % (1 << 4);  
    // assign dig5 = (value / (1 << (4 * 5))) % (1 << 4);  
    // assign dig6 = (value / (1 << (4 * 6))) % (1 << 4);  
    // assign dig7 = (value / (1 << (4 * 7))) % (1 << 4);  
  
    //      assign dig0 = value          % (1 << 4);  
    //      assign dig1 = (value >> (4 * 1)) % (1 << 4);  
    //      assign dig2 = (value >> (4 * 2)) % (1 << 4);  
    //      assign dig3 = (value >> (4 * 3)) % (1 << 4);  
    //      assign dig4 = (value >> (4 * 4)) % (1 << 4);  
    //      assign dig5 = (value >> (4 * 5)) % (1 << 4);  
    //      assign dig6 = (value >> (4 * 6)) % (1 << 4);  
    //      assign dig7 = (value >> (4 * 7)) % (1 << 4);  
  
    assign {dig7, dig6, dig5, dig4, dig3, dig2, dig1, dig0} = value;  
endmodule
```

button_debouncer.v

```
module button_debouncer #(parameter depth = 16) (  
    input wire clk,                /* 5 KHz clock */  
    input wire button,            /* Input button from constraints */  
    output reg debounced_button  
);  
  
    localparam history_max = (2**depth)-1;  
  
    /* History of sampled input button */  
    reg [depth-1:0] history;  
  
    always @ (posedge clk)  
    begin  
        /* Move history back one sample and insert new sample */  
        history <= { button, history[depth-1:1] };  
  
        /* Assert debounced button if it has been in a consistent state throughout history */  
        debounced_button <= (history == history_max) ? 1'b1 : 1'b0;  
    end  
endmodule
```

clk_gen.v

```
module clk_gen  
(input clk100MHz, rst, output reg clk_4sec, clk_5KHz);  
    integer count1, count2;  
  
    always @ (posedge clk100MHz)  
    begin  
        if (rst)  
            begin  
                count1 = 0; clk_4sec = 0;  
                count2 = 0; clk_5KHz = 0;  
            end  
        else  
            begin  
                if (count1 == 200000000)  
                    begin  
                        clk_4sec = ~clk_4sec;  
                        count1 = 0;  
                    end  
                if (count2 == 10000)  
                    begin  
                        clk_5KHz = ~clk_5KHz;  
                        count2 = 0;  
                    end  
                count1 = count1 + 1;  
                count2 = count2 + 1;  
            end  
        end  
    end  
endmodule
```

hex_to_7seg.v

```
module hex_to_7seg  
(input [3:0] HEX, output reg [7:0] s);  
    always @ (HEX)  
    begin  
        case (HEX)  
            0:      s = 8'b10001000;  
            1:      s = 8'b11101101;  
            2:      s = 8'b10100010;  
            3:      s = 8'b10100100;  
            4:      s = 8'b11000101;  
            5:      s = 8'b10010100;  
            6:      s = 8'b10010000;  
            7:      s = 8'b10101101;  
            8:      s = 8'b10000000;  
            9:      s = 8'b10000100;  
        endcase  
    end  
endmodule
```

```

        /*
        TODO: Verify that these are correct
        */
        4'hA: s = 8'b10000001;
        4'hB: s = 8'b11010000;
        4'hC: s = 8'b10011010;
        4'hD: s = 8'b11100000;
        4'hE: s = 8'b10010010;
        4'hF: s = 8'b10010011;
        default:s = 8'b01111111;
    endcase
end
endmodule

```

led_mux.v

```

module led_mux (input clk, rst,
                input [7:0] LED7, LED6, LED5, LED4, LED3, LED2, LED1, LED0,
                output [7:0] LEDSEL, LEDOUT
);

reg [2:0] index;
reg [15:0] led_ctrl;

assign {LEDSEL, LEDOUT} = led_ctrl;

always @ (posedge clk) index <= (rst) ? 3'b0 : (index + 3'd1);

always @ (index, LED0, LED1, LED2, LED3, LED4, LED5, LED6, LED7)
begin
    case (index)
        0: led_ctrl <= {8'b11111110, LED0};
        1: led_ctrl <= {8'b11111101, LED1};
        2: led_ctrl <= {8'b11111011, LED2};
        3: led_ctrl <= {8'b11110111, LED3};
        4: led_ctrl <= {8'b11101111, LED4};
        5: led_ctrl <= {8'b11011111, LED5};
        6: led_ctrl <= {8'b10111111, LED6};
        7: led_ctrl <= {8'b01111111, LED7};
        default: led_ctrl <= {8'b11111111, 8'hFF};
    endcase
end
endmodule

```

B. MODULES:

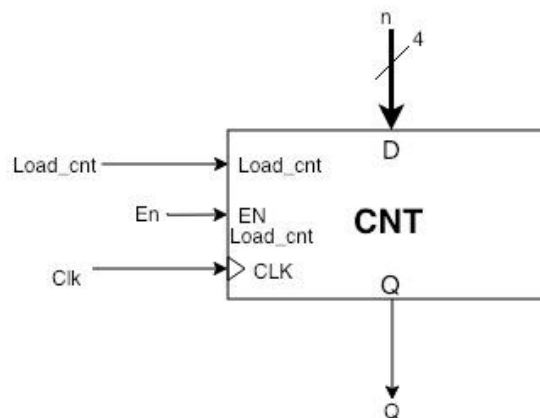


Figure 11: *CNT.v* module for down counter

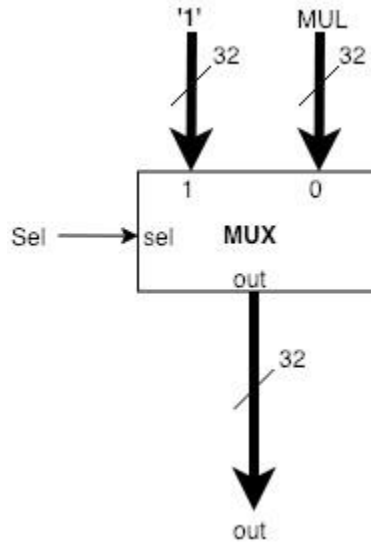


Figure 12: *MUX2.v module for Multiplexer*

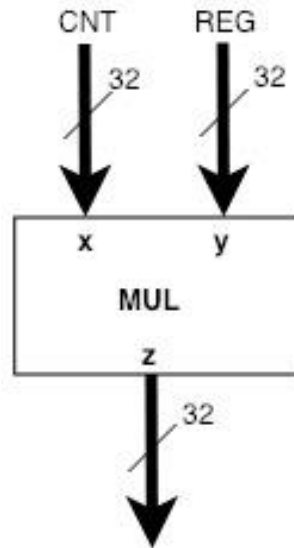


Figure 13: *Mul.v module for Multiplier*

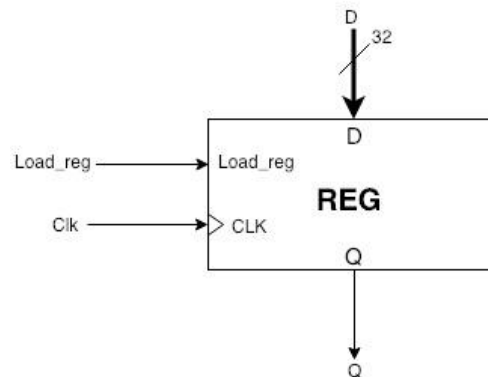


Figure 14: *Reg.v module for Data Register*

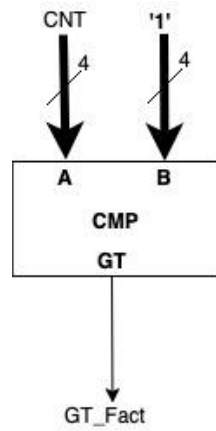


Figure 15: *Mag_Comp.v* module

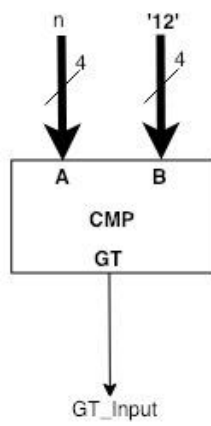


Figure 16: *Mag_Comp.v* module

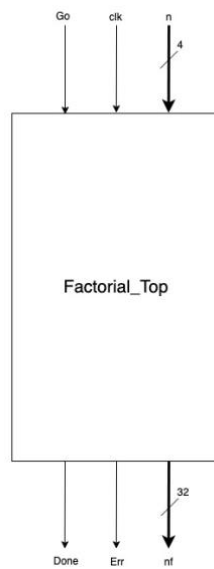


Figure 17: *Factorial_Top.v* module

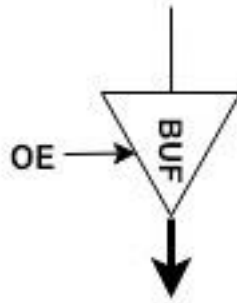


Figure 18: *MUX2.v* module this figure shows the buffer as a mux

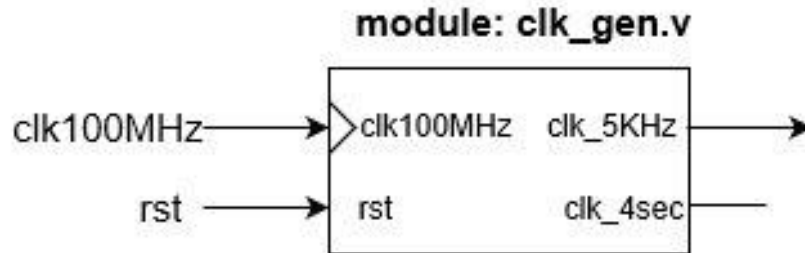


Figure 19: *clk_gen.v* module: sets the clock frequency from 100MHz to 5KHz which will then be connected to the *lex_mux* module. In addition, *clk_gen* module has a reset which is also connected to the *led_mux* module. Lastly this module also sets a clock frequency of 4 seconds.

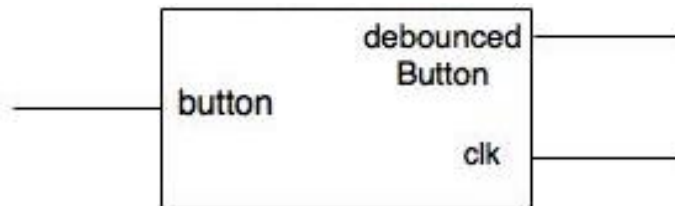


Figure 20: *button_debouncer.v* module: Used to manually control the clock signal going to the registers.

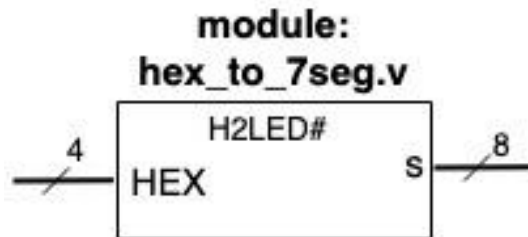


Figure 21: *hex_to_7seg.v* module

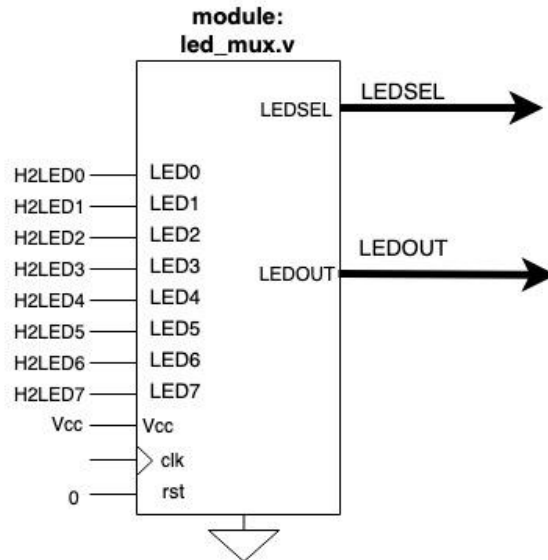


Figure 22: *led_mux.v* module: takes the input from the 8 *hex_to_7 seg.v* converters and depending on the combination the multiplexer will output the correct signal to enable the segments of the 7segments LED output.

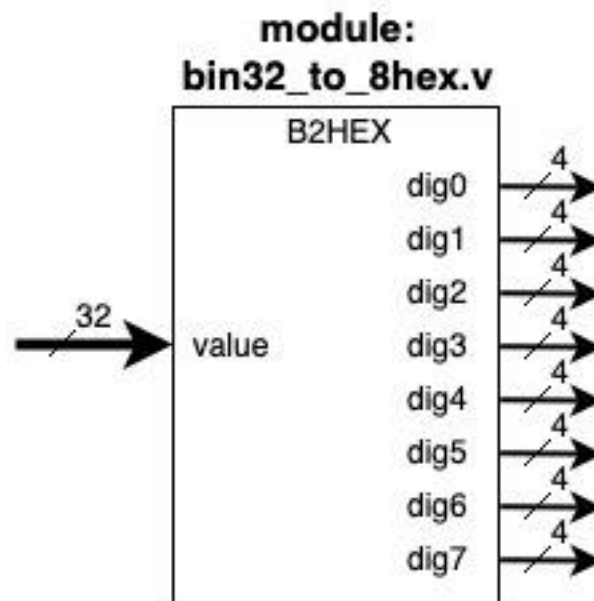


Figure 23: *bin32_to_8hex.v* module: takes the input from the *Factorial_Top.v* and converts the value into hex

C. ASM chart and State Transition Diagram:

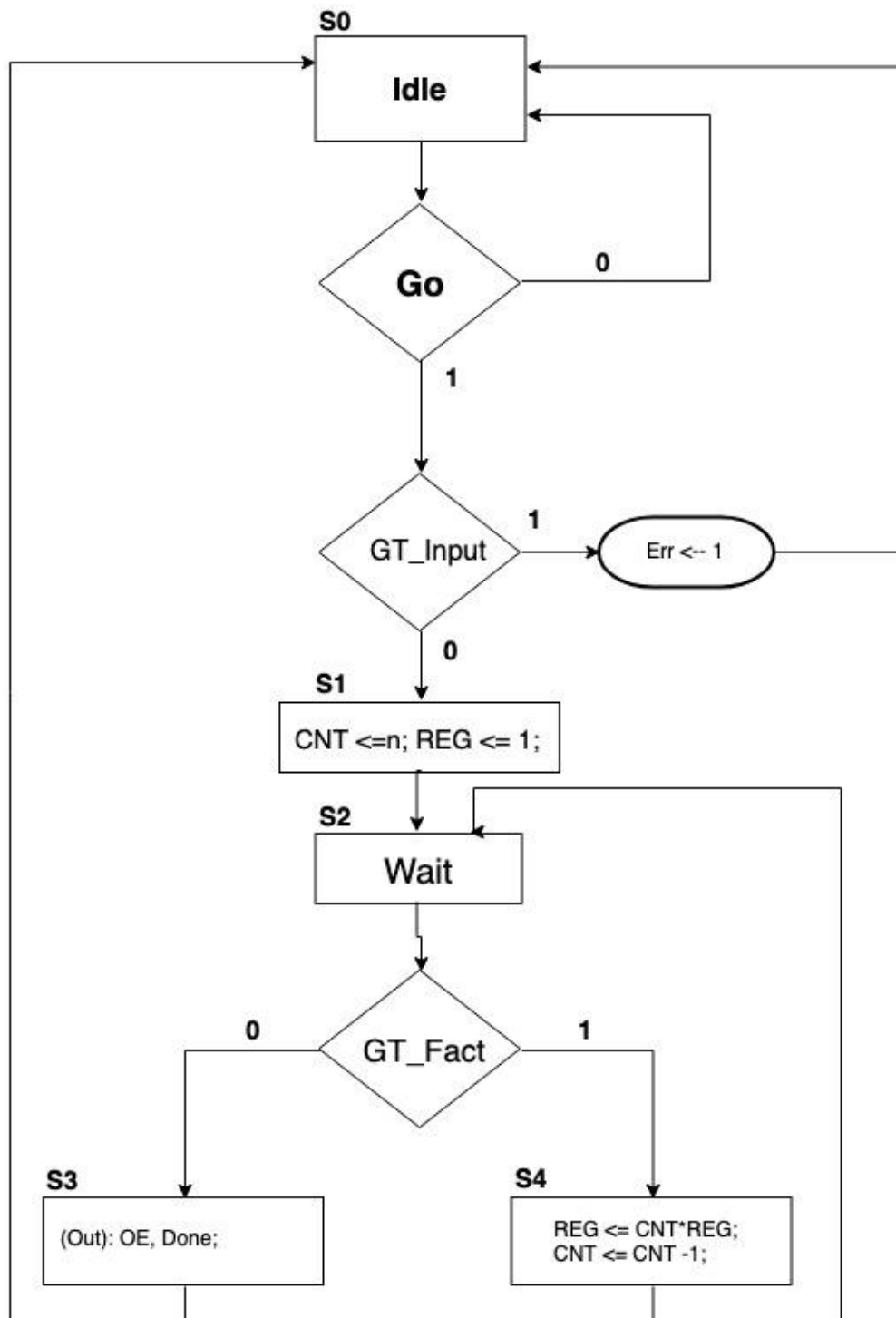


Figure 24: ASM chart of a Mealy State Machine

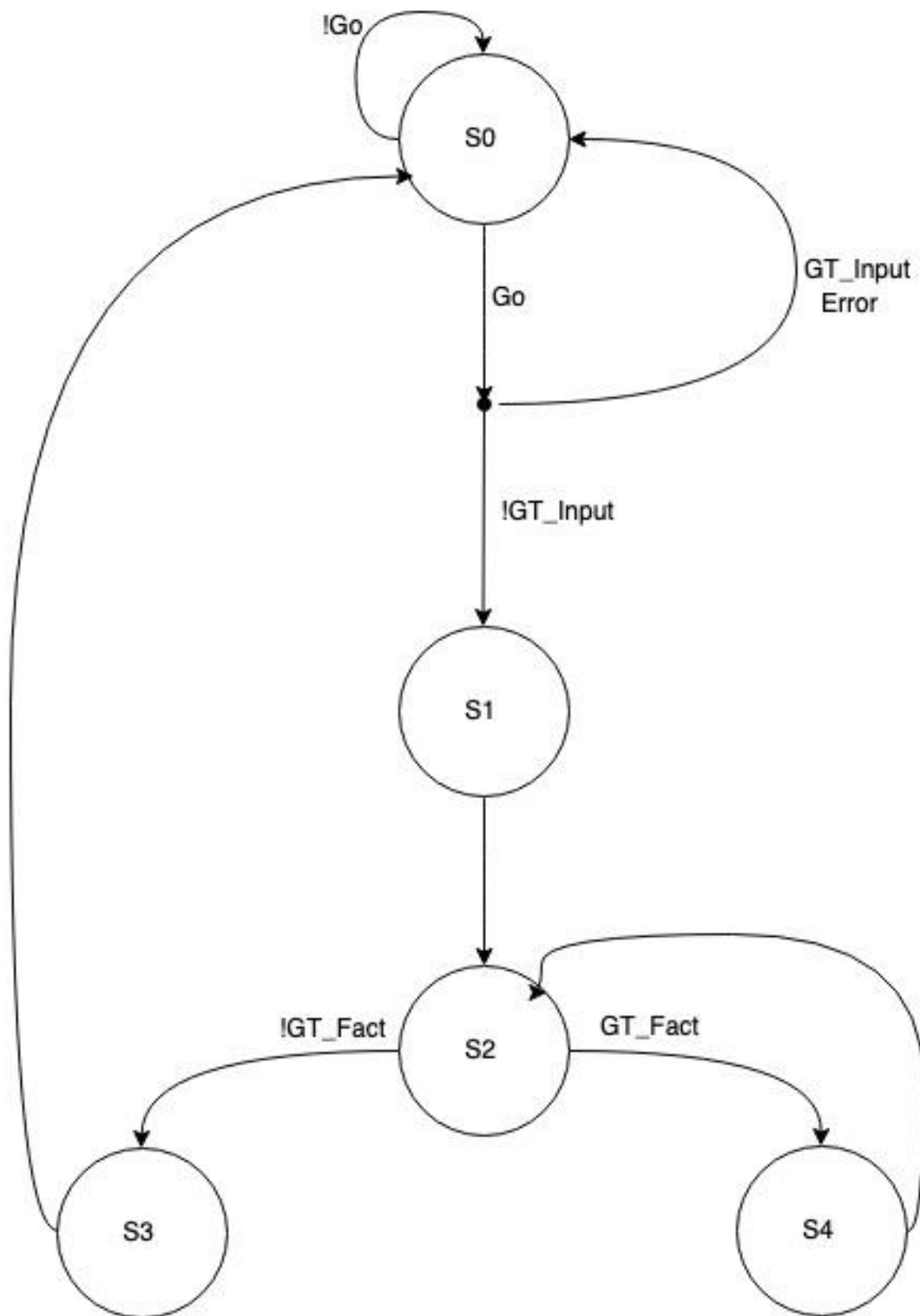


Figure 25: State Transition Diagram of a Mealy State Machine