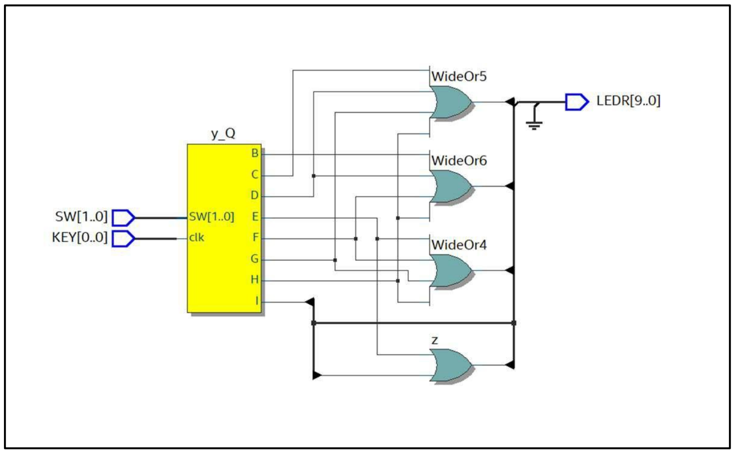
Exp-7

PART 2

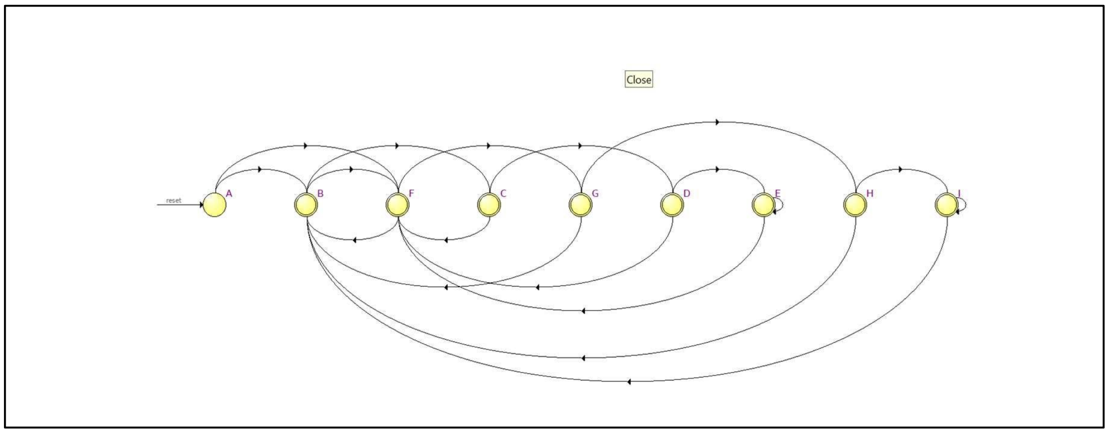
Program:

```
module part2 (SW, KEY, LEDR);
          input [1:0] SW;
          input [0:0] KEY;
          output [9:0] LEDR;
          wire Clock, Resetn, w, z;
          reg [3:0] y_Q, Y_D;
          assign Clock = KEY[0];
          assign Resetn = SW[0];
          assign w = SW[1];
          parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100,
                    F = 4'b0101, G = 4'b0110, H = 4'b0111, I = 4'b1000;
          always @(w, y_Q)
          begin: state_table
                    case (y_Q)
                              A:        if (!w) Y_D = B;
                                        else Y_D = F;
                              B:        if (!w) Y_D = C;
                                        else Y_D = F;
                              C:        if (!w) Y_D = D;
                                        else Y_D = F;
                              D:        if (!w) Y_D = E;
                                        else Y_D = F;
                              E:        if (!w) Y_D = E;
                                        else Y_D = F;
                              F:        if (!w) Y_D = B;
                                        else Y_D = G;
                              G:        if (!w) Y_D = B;
                                        else Y_D = H;
                              H:        if (!w) Y_D = B;
                                        else Y_D = I;
                              I:        if (!w) Y_D = B;
                                        else Y_D = I;
                              default: Y_D = 4'bxxxx;
                    endcase
          end // state_table
          always @(posedge Clock)
                    if (Resetn  == 1'b0)        // synchronous clear
                              y_Q <= A;
                    else
                              y_Q <= Y_D;
          assign z = ((y_Q == E) | (y_Q == I)) ? 1'b1 : 1'b0;
          assign LEDR[3:0] = y_Q;
          assign LEDR[9] = z;
          assign LEDR[8:4] = 5'b0;
endmodule
```

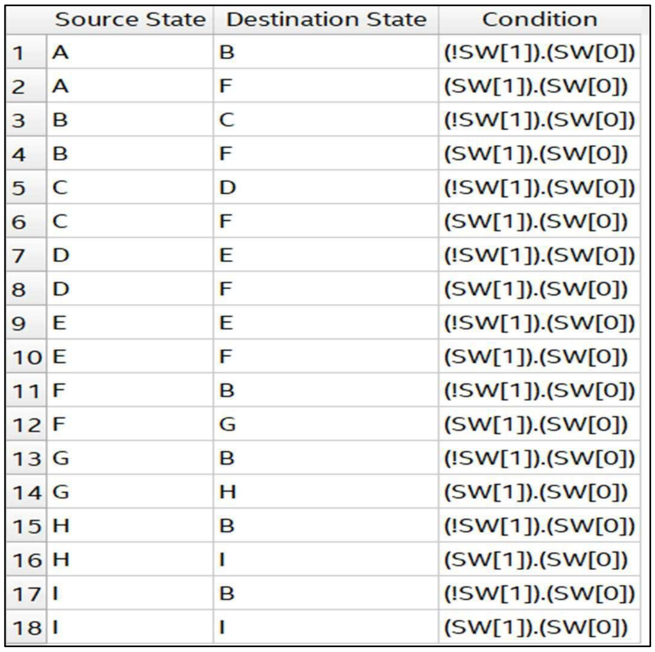
RLT View:



State Machine View:



State Transition Tabel:



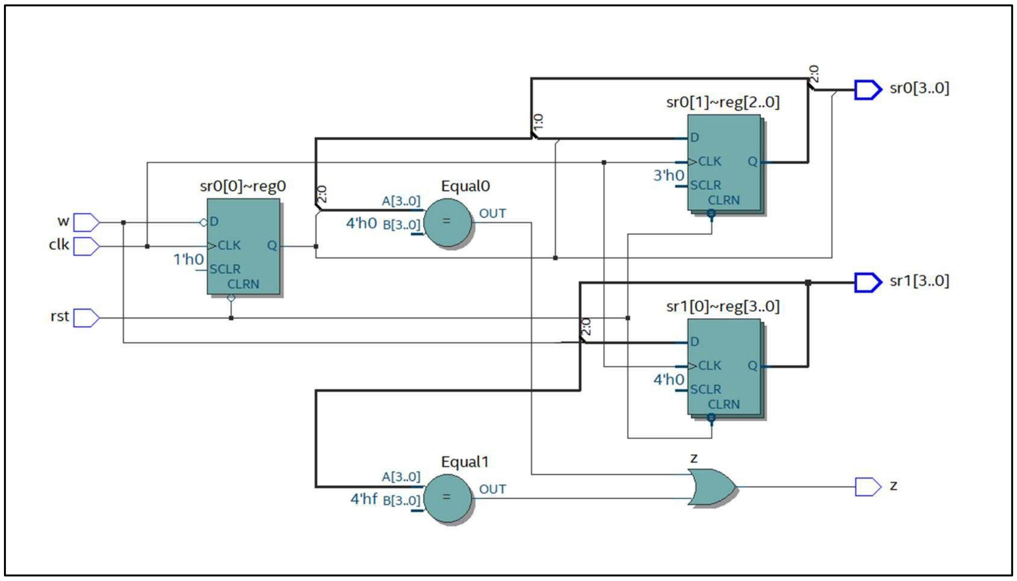
	Source State	Destination State	Condition
1	A	B	(!SW[1]).(SW[0])
2	A	F	(SW[1]).(SW[0])
3	B	C	(!SW[1]).(SW[0])
4	B	F	(SW[1]).(SW[0])
5	C	D	(!SW[1]).(SW[0])
6	C	F	(SW[1]).(SW[0])
7	D	E	(!SW[1]).(SW[0])
8	D	F	(SW[1]).(SW[0])
9	E	E	(!SW[1]).(SW[0])
10	E	F	(SW[1]).(SW[0])
11	F	B	(!SW[1]).(SW[0])
12	F	G	(SW[1]).(SW[0])
13	G	B	(!SW[1]).(SW[0])
14	G	H	(SW[1]).(SW[0])
15	H	B	(!SW[1]).(SW[0])
16	H	I	(SW[1]).(SW[0])
17	I	B	(!SW[1]).(SW[0])
18	I	I	(SW[1]).(SW[0])

PART 3

Program:

```
module part3 (clk,rst,w,z,sr0,sr1);
  input clk,rst,w;
  output reg z;
  output reg [3:0] sr0,sr1;

  always @(posedge clk or negedge rst) begin
    if (!rst)
      sr0 <= 4'b0000;
    else
      sr0 <= {sr0[2:0], ~w};
  end

  always @(posedge clk or negedge rst) begin
    if (!rst)
      sr1 <= 4'b0000;
    else
      sr1 <= {sr1[2:0], w};
  end

  always @(sr0 or sr1) begin
    z = (sr0 == 4'b0000) || (sr1 == 4'b1111);
  end

endmodule
```

RLT View:



PART 4

Program:

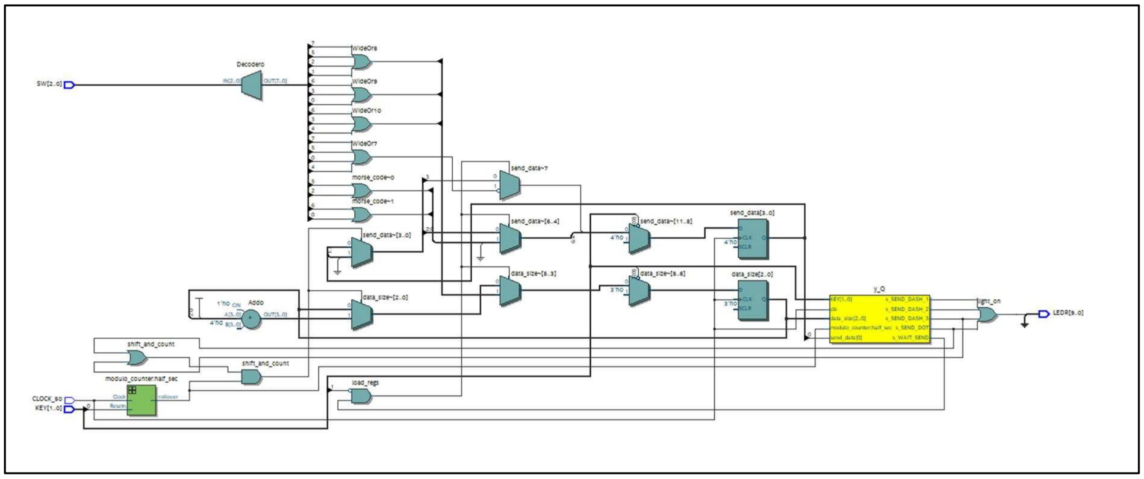
```
// This is a Morse code encoder circuit. It displays the Morse code for
// letters A to H on LEDR[0]. The letter to be displayed is selected using
// SW[3..0], using A = 000, B = 001, ..., H = 111.
module part4 (SW, CLOCK_50, KEY, LEDR);
	/*****************************************************************/
	/****	PARAMETER DECLARATIONS				****/
	/*****************************************************************/
	// SW switch patterns, Morse codes, and code lengths are defined below (in the Morse
	// code, 0 = dot, 1 = dash)
	parameter A_SW = 3'b000, A_MORSE = 4'b0010, A_LENGTH = 3'd2; /* .-   */
	parameter B_SW = 3'b001, B_MORSE = 4'b0001, B_LENGTH = 3'd4; /* -... */
	parameter C_SW = 3'b010, C_MORSE = 4'b0101, C_LENGTH = 3'd4; /* -.-. */
	parameter D_SW = 3'b011, D_MORSE = 4'b0001, D_LENGTH = 3'd3; /* -..  */
	parameter E_SW = 3'b100, E_MORSE = 4'b0000, E_LENGTH = 3'd1; /* .    */
	parameter F_SW = 3'b101, F_MORSE = 4'b0100, F_LENGTH = 3'd4; /* ..-. */
	parameter G_SW = 3'b110, G_MORSE = 4'b0011, G_LENGTH = 3'd3; /* --.  */
	parameter H_SW = 3'b111, H_MORSE = 4'b0000, H_LENGTH = 3'd4; /*....... */

	parameter	s_WAIT_SEND = 3'b000, s_WAIT_BLANK = 3'b001, s_SEND_DOT = 3'b010,
					s_SEND_DASH_1 = 3'b011, s_SEND_DASH_2 = 3'b100,
s_SEND_DASH_3 = 3'b101,
					s_RELEASE_SEND = 3'b110;

	/*****************************************************************/
	/****	PORT DECLARATIONS	****/
	/*****************************************************************/
	input [2:0] SW;
	input [1:0] KEY;
	input CLOCK_50;
	output [9:0] LEDR;

	/*****************************************************************/
	/****	LOCAL WIRE DECLARATIONS	****/
	/*****************************************************************/
	wire Clock, Resetn, go, half_sec_enable, load_regs, shift_and_count, light_on;
	reg [3:0] morse_code;
	reg [2:0] morse_length;
	reg [3:0] send_data;
	reg [2:0] data_size;
	wire [1:0] pulse_cycle;
	reg [3:0] y_Q, Y_D;

	/*****************************************************************/
	/****	IMPLEMENTATION	****/
```

```
/*****************************************************************/
assign Clock = CLOCK_50;
assign Resetn = KEY[0];
assign go = ~KEY[1];

// FSM State Table
always @(go, y_Q, send_data, data_size, half_sec_enable)
begin: state_table
        case (y_Q)
                s_WAIT_SEND:
                        if (go) Y_D = s_WAIT_BLANK;
                        else Y_D = s_WAIT_SEND;
                s_WAIT_BLANK:// sync with the half-second pulses
                        if (!half_sec_enable)
                                Y_D = s_WAIT_BLANK;
                        else if (send_data[0] == 1'b0)
                                        Y_D = s_SEND_DOT;
                        else
                                        Y_D = s_SEND_DASH_1;
                s_SEND_DOT:             // wait here for one half-second period
                        if (!half_sec_enable)
                                Y_D = s_SEND_DOT;
                        else if (data_size == 'd1) // check if we are done with this letter
                                        Y_D = s_RELEASE_SEND;
                        else
                                        Y_D = s_WAIT_BLANK;
                s_SEND_DASH_1:          // wait for three half-second periods
                        if (!half_sec_enable)
                                Y_D = s_SEND_DASH_1;
                        else
                                Y_D = s_SEND_DASH_2;
                s_SEND_DASH_2:          // wait for two more half-second periods
                        if (!half_sec_enable)
                                Y_D = s_SEND_DASH_2;
                        else
                                Y_D = s_SEND_DASH_3;
                s_SEND_DASH_3:          // wait for one more half-second period
                        if (!half_sec_enable)
                                Y_D = s_SEND_DASH_3;
                        else if (data_size == 'd1) // check if we are done with this letter
                                Y_D = s_RELEASE_SEND;
                        else
                                Y_D = s_WAIT_BLANK;
                s_RELEASE_SEND:
                        if (~go) Y_D = s_WAIT_SEND;
                        else Y_D = s_RELEASE_SEND;

                default: Y_D = 3'bxxx;
```

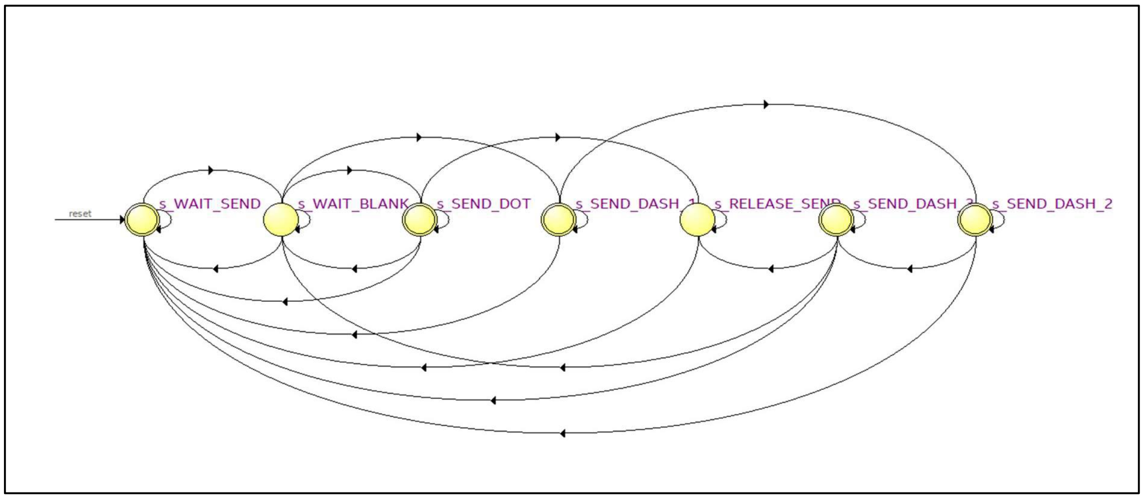
```
		endcase
	end // state_table

	// FSM State flip-flops
	always @(posedge Clock)
		if (Resetn  == 1'b0)		// synchronous clear
			y_Q <= s_WAIT_SEND;
		else
			y_Q <= Y_D;

	// FSM outputs
	// turn on the Morse code light in the states below
	assign light_on = ( (y_Q == s_SEND_DOT) | (y_Q == s_SEND_DASH_1) |
		(y_Q == s_SEND_DASH_2) | (y_Q == s_SEND_DASH_3) );
	// specify when to load the Morse code into the shift register, and length into the counter
	assign load_regs = (y_Q == s_WAIT_SEND) & go;
	// specify when to shift the Morse code bits and decrement the length counter
	assign shift_and_count = ((y_Q == s_SEND_DOT) | (y_Q == s_SEND_DASH_3)) &
half_sec_enable;

	/* Create an enable signal that is asserted once every 0.5 of a second. */
	modulo_counter half_sec( .Clock(CLOCK_50), .Resetn(Resetn), .rollover(half_sec_enable) );
		defparam half_sec.n = 25;
		defparam half_sec.k = 25000000;

	/* Letter selection */
	always @(*)
	case (SW)
		A_SW:  begin morse_code = A_MORSE; morse_length = A_LENGTH; end
		B_SW:  begin morse_code = B_MORSE; morse_length = B_LENGTH; end
		C_SW:  begin morse_code = C_MORSE; morse_length = C_LENGTH; end
		D_SW:  begin morse_code = D_MORSE; morse_length = D_LENGTH; end
		E_SW:   begin morse_code = E_MORSE; morse_length = E_LENGTH; end
		F_SW:   begin morse_code = F_MORSE; morse_length = F_LENGTH; end
		G_SW:  begin morse_code = G_MORSE; morse_length = G_LENGTH; end
		H_SW:   begin morse_code = H_MORSE; morse_length = H_LENGTH; end
	endcase

	/* Store the Morse code to be sent in a shift register, and its length in a counter */
	always@(posedge CLOCK_50)
	begin
		if (~Resetn)
		begin
			send_data <= 'd0;
			data_size <= 'd0;
		end
		else
			if (load_regs)
```

```
                    begin
                        send_data <= morse_code;
                        data_size <= morse_length;
                    end
                    else if (shift_and_count) // shift and decrement when appropriate
                    begin
                        send_data[2:0] <= send_data[3:1];
                        send_data[3] <= 1'b0;
                        data_size <= data_size - 1'b1;
                    end
        end

        assign LEDR[0] = light_on;
        assign LEDR[9:1] = 9'b0;
endmodule

module modulo_counter(Clock, Resetn, rollover);
        /*****************************************************************/
        /****       PARAMETER DECLARATIONS       ****/
        /*****************************************************************/
        parameter               n = 4;
        parameter               k = 16;

        /*****************************************************************/
        /****       PORT DECLARATIONS      ****/
        /*****************************************************************/
        input   Clock, Resetn;
        output  rollover;
        reg             [n-1:0]  Q;

        /*****************************************************************/
        /****       IMPLEMENTATION      ****/
        /*****************************************************************/
        always@(posedge Clock)
        begin
            if (!Resetn)
                Q <= 'd0;
            else if (Q == k-1)
                Q <= 'd0;
            else
                Q <= Q + 1'b1;
        end

        assign rollover = (Q == k-1);
endmodule
```

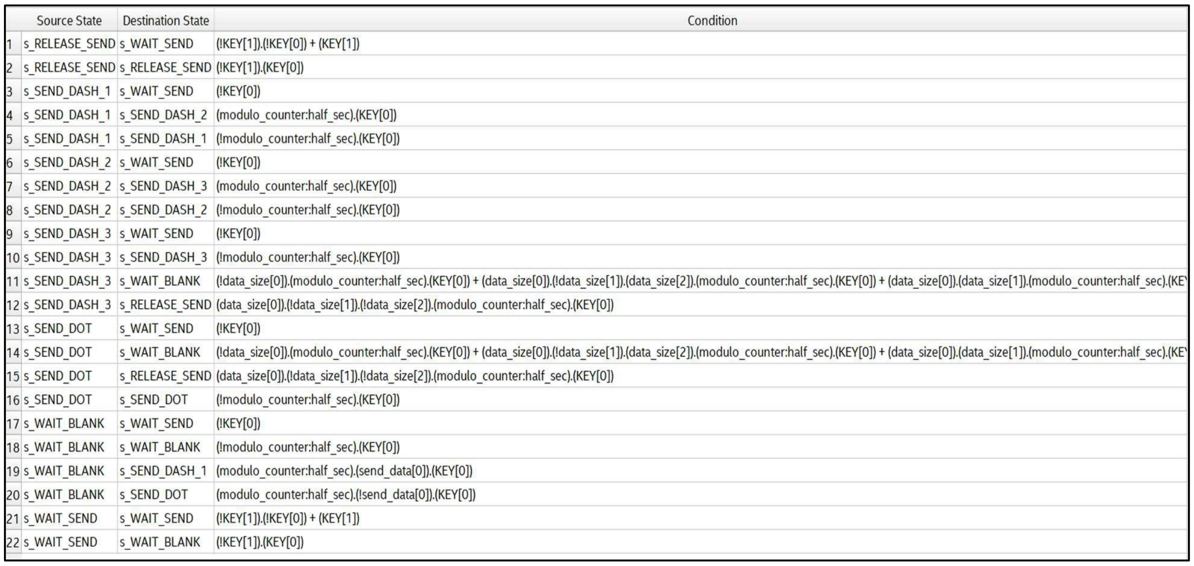
RLT View:



State Transition View:



Transition Table:



	Source State	Destination State	Condition
1	s_RELEASE_SEND	s_WAIT_SEND	(!KEY[1]).(!KEY[0]) + (KEY[1])
2	s_RELEASE_SEND	s_RELEASE_SEND	(!KEY[1]).(KEY[0])
3	s_SEND_DASH_1	s_WAIT_SEND	(!KEY[0])
4	s_SEND_DASH_1	s_SEND_DASH_2	(modulo_counter:half_sec).(KEY[0])
5	s_SEND_DASH_1	s_SEND_DASH_1	(!modulo_counter:half_sec).(KEY[0])
6	s_SEND_DASH_2	s_WAIT_SEND	(!KEY[0])
7	s_SEND_DASH_2	s_SEND_DASH_3	(modulo_counter:half_sec).(KEY[0])
8	s_SEND_DASH_2	s_SEND_DASH_2	(!modulo_counter:half_sec).(KEY[0])
9	s_SEND_DASH_3	s_WAIT_SEND	(!KEY[0])
10	s_SEND_DASH_3	s_SEND_DASH_3	(!modulo_counter:half_sec).(KEY[0])
11	s_SEND_DASH_3	s_WAIT_BLANK	(!data_size[0]).(modulo_counter:half_sec).(KEY[0]) + (data_size[0]).(!data_size[1]).(data_size[2]).(modulo_counter:half_sec).(KEY[0]) + (data_size[0]).(data_size[1]).(modulo_counter:half_sec).(KE
12	s_SEND_DASH_3	s_RELEASE_SEND	(data_size[0]).(!data_size[1]).(!data_size[2]).(modulo_counter:half_sec).(KEY[0])
13	s_SEND_DOT	s_WAIT_SEND	(!KEY[0])
14	s_SEND_DOT	s_WAIT_BLANK	(!data_size[0]).(modulo_counter:half_sec).(KEY[0]) + (data_size[0]).(!data_size[1]).(data_size[2]).(modulo_counter:half_sec).(KEY[0]) + (data_size[0]).(data_size[1]).(modulo_counter:half_sec).(KE
15	s_SEND_DOT	s_RELEASE_SEND	(data_size[0]).(!data_size[1]).(!data_size[2]).(modulo_counter:half_sec).(KEY[0])
16	s_SEND_DOT	s_SEND_DOT	(!modulo_counter:half_sec).(KEY[0])
17	s_WAIT_BLANK	s_WAIT_SEND	(!KEY[0])
18	s_WAIT_BLANK	s_WAIT_BLANK	(!modulo_counter:half_sec).(KEY[0])
19	s_WAIT_BLANK	s_SEND_DASH_1	(modulo_counter:half_sec).(send_data[0]).(KEY[0])
20	s_WAIT_BLANK	s_SEND_DOT	(modulo_counter:half_sec).(!send_data[0]).(KEY[0])
21	s_WAIT_SEND	s_WAIT_SEND	(!KEY[1]).(!KEY[0]) + (KEY[1])
22	s_WAIT_SEND	s_WAIT_BLANK	(!KEY[1]).(KEY[0])

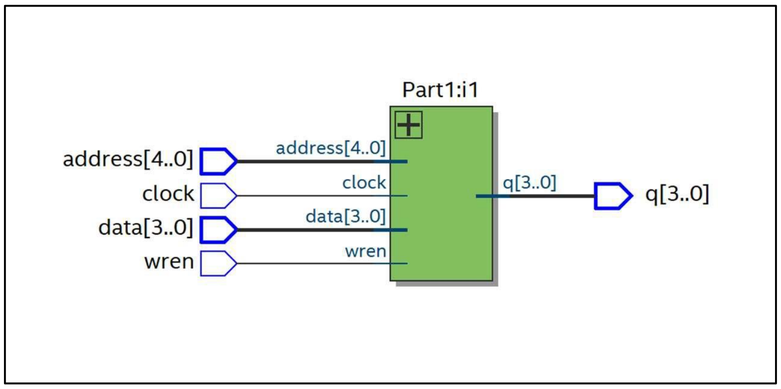
LABORATORY EXERCISE – 8

PART 1

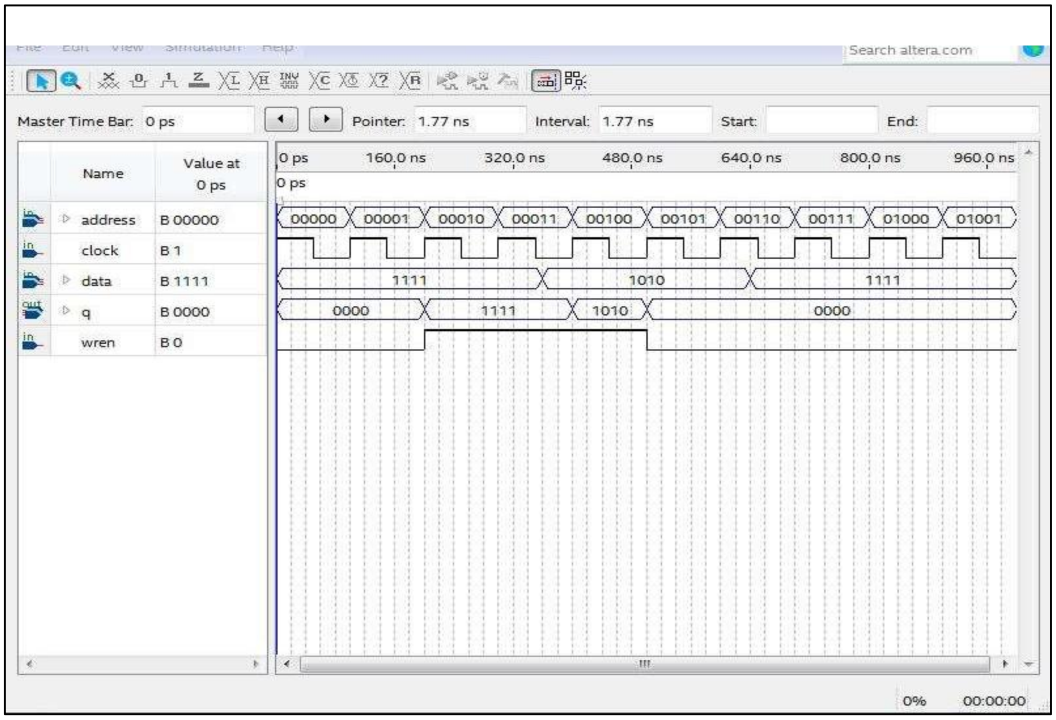
Program:

```
3 module part1(address,clock,data,wren,q);
        input clock,wren;
        input [3:0] data;
        input [4:0] address;
        output [3:0] q;
        Part1 i1(address,clock,data,wren,q);
Endmodule
```

RLT View:



Output Simulation:



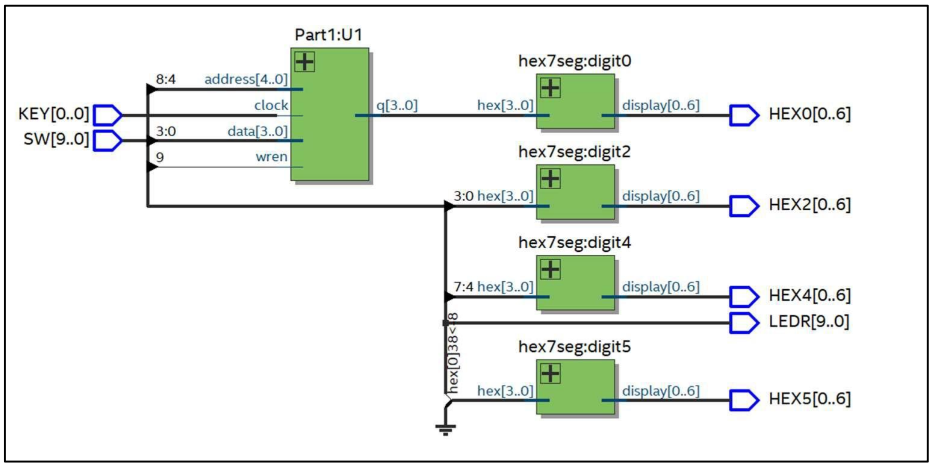
PART 2

Program:

```
// This code instantiates a 32 x 4 memory
//
// inputs: KEY0 is the clock, SW3-SW0 provides data to write into memory.
// SW8-SW4 provides the memory address, SW9 is the memory Write input.
// outputs: 7-seg displays HEX5-4 show the memory address, HEX2
// displays the data input to the memory, and HEX0 show the contents read
// from the memory. LEDGR shows the status of the SW switches.

module part2 (KEY, SW, HEX5, HEX4, HEX2, HEX0, LEDR);
	input [0:0] KEY;
	input [9:0] SW;
	output [0:6] HEX5, HEX4, HEX2, HEX0;
	output [9:0] LEDR;

	wire Clock, Write;
	wire [4:0] Address;
	wire [3:0] DataIn, DataOut;

	assign Clock = KEY[0];
	assign Write = SW[9];
	assign DataIn = SW[3:0];
	assign Address = SW[8:4];

	// instantiate memory module
	// module ram32x4 (address, clock, data, wren, q);
	ram32x4 U1 (Address, Clock, DataIn, Write, DataOut);

	// display the data input, data output, and address on the 7-segs
	hex7seg digit0 (DataOut[3:0], HEX0);
	hex7seg digit2 (DataIn[3:0], HEX2);
	hex7seg digit5 ({3'b0, Address[4]}, HEX5);
	hex7seg digit4 (Address[3:0], HEX4);

	assign LEDR[3:0] = DataIn;
	assign LEDR[8:4] = Address;
	assign LEDR[9] = Write;
endmodule

//-----------------------------------------------------------------
module hex7seg (hex, display);
	input [3:0] hex;
	output [0:6] display;

	reg [0:6] display;
```

```
        always @ (hex)
                case (hex)
                        4'h0: display = 7'b0000001;
                        4'h1: display = 7'b1001111;
                        4'h2: display = 7'b0010010;
                        4'h3: display = 7'b0000110;
                        4'h4: display = 7'b1001100;
                        4'h5: display = 7'b0100100;
                        4'h6: display = 7'b0100000;
                        4'h7: display = 7'b0001111;
                        4'h8: display = 7'b0000000;
                        4'h9: display = 7'b0000100;
                        4'hA: display = 7'b0001000;
                        4'hb: display = 7'b1100000;
                        4'hC: display = 7'b0110001;
                        4'hd: display = 7'b1000010;
                        4'hE: display = 7'b0110000;
                        4'hF: display = 7'b0111000;
                endcase
endmodule
```

RLT View:



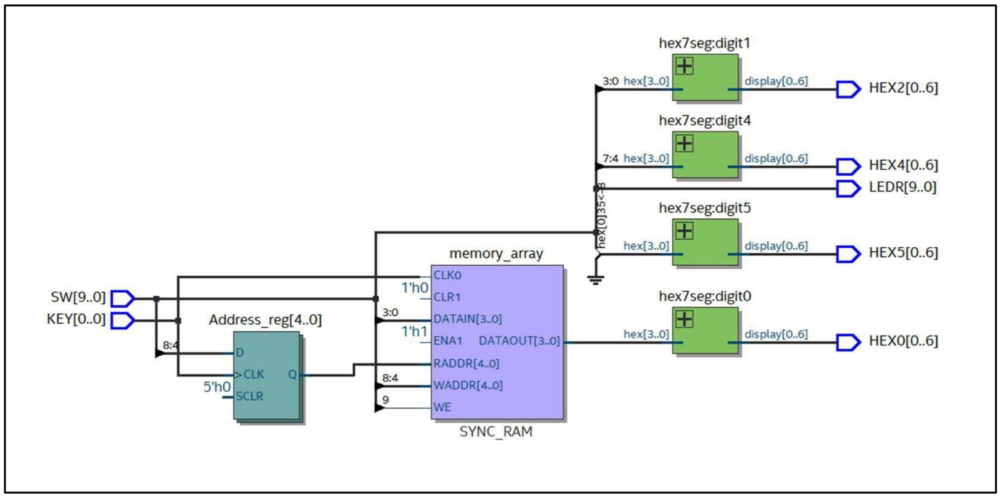
<u>PART 3</u>

Program:

```
//Name: R Yashswini , USN: 1MS21EE043
// This code instantiates a 32 x 4 memory
//
// inputs: KEY0 is the clock, SW3-SW0 provides data to write into memory.
// SW8-SW4 provides the memory address, SW9 is the memory Write input.
// outputs: 7-seg displays HEX5-4 show the memory address, HEX2
// displays the data input to the memory, and HEX0 show the contents read
// from the memory. LEDR shows the status of the SW switches.
module part3 (KEY, SW, HEX5, HEX4, HEX2, HEX0, LEDR);
        input [0:0] KEY;
        input [9:0] SW;
        output [0:6] HEX5, HEX4, HEX2, HEX0;
        output [9:0] LEDR;

        wire Clock, Write;
        wire [4:0] Address;
        wire [3:0] DataIn, DataOut;

        assign Clock = KEY[0];
        assign Write = SW[9];
        assign DataIn = SW[3:0];
        assign Address = SW[8:4];

        reg [3:0] memory_array [31:0];
        reg [4:0] Address_reg;

        // infer RAM module
        always @(posedge Clock)
        begin
                if (Write)
                        memory_array[Address] <= DataIn;
                Address_reg <= Address;
  end
  assign DataOut = memory_array[Address_reg];


        // display the data input, data output, and address on the 7-segs
        hex7seg digit0 (DataOut[3:0], HEX0);
        hex7seg digit1 (DataIn[3:0], HEX2);
        hex7seg digit5 ({3'b0, Address[4]}, HEX5);
        hex7seg digit4 (Address[3:0], HEX4);

        assign LEDR[3:0] = DataIn;
        assign LEDR[8:4] = Address;
        assign LEDR[9] = Write;
endmodule

//------------------------------------------------------------------
module hex7seg (hex, display);
        input [3:0] hex;
        output [0:6] display;

        reg [0:6] display;

        always @ (hex)
```

```
                case (hex)
                        4'h0: display = 7'b0000001;
                        4'h1: display = 7'b1001111;
                        4'h2: display = 7'b0010010;
                        4'h3: display = 7'b0000110;
                        4'h4: display = 7'b1001100;
                        4'h5: display = 7'b0100100;
                        4'h6: display = 7'b0100000;
                        4'h7: display = 7'b0001111;
                        4'h8: display = 7'b0000000;
                        4'h9: display = 7'b0000100;
                        4'hA: display = 7'b0001000;
                        4'hb: display = 7'b1100000;
                        4'hC: display = 7'b0110001;
                        4'hd: display = 7'b1000010;
                        4'hE: display = 7'b0110000;
                        4'hF: display = 7'b0111000;
                endcase
endmodule
```

RLT View:



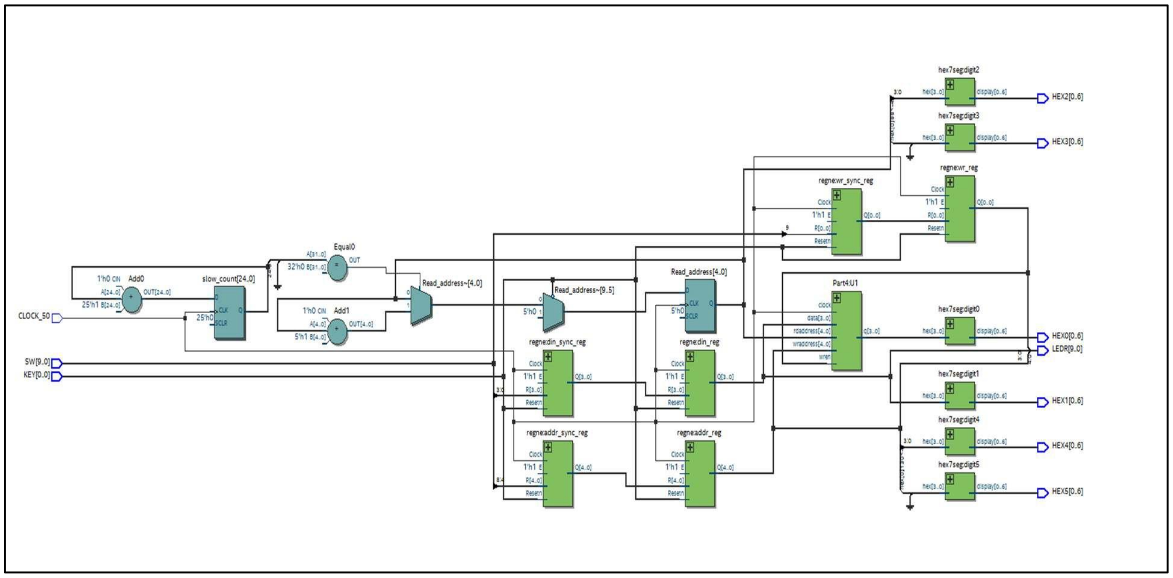
PART 4

Program:

```
// This code implements a simple dual-port memory
//
// inputs: CLOCK_50 is the clock, KEY0 is Resetn, SW3-SW0 provides data to
// write into memory.
// SW8-SW4 provides the memory address for writing, SW9 is the memory Write input.
// outputs: 7-seg display HEX5-4 displays the write address, and HEX3-2 shows the read
// address. HEX1 displays the write data and HEX0 shows the read data.
// LEDR shows the status of the SW switches.
module part4 (CLOCK_50, KEY, SW, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0, LEDR);
	input CLOCK_50;
	input [0:0] KEY;
	input [9:0] SW;
	output [0:6] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
	output [9:0] LEDR;

	wire Clock, Resetn, Write, Write_sync;
	wire [4:0] Write_address, Write_address_sync;
	wire [3:0] DataIn, DataIn_sync, DataOut;

	assign Resetn = KEY[0];
	assign Clock = CLOCK_50;

	// synchronize all asynchronous inputs to the clock
	regne #(.n(1)) wr_sync_reg(SW[9], Clock, Resetn, 1'b1, Write_sync);
	regne #(.n(1)) wr_reg(Write_sync, Clock, Resetn, 1'b1, Write);
	regne #(.n(5)) addr_sync_reg(SW[8:4], Clock, Resetn, 1'b1, Write_address_sync);
	regne #(.n(5)) addr_reg(Write_address_sync, Clock, Resetn, 1'b1, Write_address);
	regne #(.n(4)) din_sync_reg(SW[3:0], Clock, Resetn, 1'b1, DataIn_sync);
	regne #(.n(4)) din_reg(DataIn_sync, Clock, Resetn, 1'b1, DataIn);

	// one second cycle counter
	parameter m = 25;
	reg [m-1:0] slow_count;
	reg [4:0] Read_address; // cycles from addresses 0 to 31 at one second per address

	// Create a 1Hz 5-bit address counter
	// A large counter to produce a 1 second (approx) enable
	always @(posedge Clock)
		slow_count <= slow_count + 1'b1;
	// the read address counter
	always @ (posedge Clock)
		if (Resetn == 0)
			Read_address <= 5'b0;
		else if (slow_count == 0)
			Read_address <= Read_address + 1'b1;

	// instantiate memory module
	// module ram32x4 (clock, data, rdaddress, wraddress, wren, q);
	ram32x4 U1 (Clock, DataIn, Read_address, Write_address, Write, DataOut);

	// display the data input, data output, and addresses on the 7-segs
	hex7seg digit5 ({3'b0, Write_address[4]}, HEX5);
	hex7seg digit4 (Write_address[3:0], HEX4);
	hex7seg digit3 ({3'b0, Read_address[4]}, HEX3);
	hex7seg digit2 (Read_address[3:0], HEX2);
	hex7seg digit1 (DataIn[3:0], HEX1);
	hex7seg digit0 (DataOut[3:0], HEX0);

	assign LEDR[3:0] = DataIn;
	assign LEDR[8:4] = Write_address;
	assign LEDR[9] = Write;
endmodule

module regne (R, Clock, Resetn, E, Q);
	parameter n = 7;
	input [n-1:0] R;
	input Clock, Resetn, E;
	output [n-1:0] Q;
	reg [n-1:0] Q;

	always @(posedge Clock)
		if (Resetn == 0)
			Q <= {n{1'b0}};
		else if (E)
			Q <= R;
endmodule

//------------------------------------------------------------

module hex7seg (hex, display);
	input [3:0] hex;
	output [0:6] display;

	reg [0:6] display;

	always @ (hex)
		case (hex)
			4'h0: display = 7'b0000001;
			4'h1: display = 7'b1001111;
			4'h2: display = 7'b0010010;
			4'h3: display = 7'b0000110;
			4'h4: display = 7'b1001100;
			4'h5: display = 7'b0100100;
			4'h6: display = 7'b0100000;
			4'h7: display = 7'b0001111;
			4'h8: display = 7'b0000000;
			4'h9: display = 7'b0000100;
			4'hA: display = 7'b0001000;
			4'hb: display = 7'b1100000;
			4'hC: display = 7'b0110001;
			4'hd: display = 7'b1000010;
```

```
                    4'hE: display = 7'b0110000;
                    4'hF: display = 7'b0111000;
            endcase
endmodule
```

RLT View:



EXP-6
Part-3

```
// Implements an unsigned multiplier P = A x B.
// inputs:	SW11-8 = A, SW3-0 = B
// outputs:		HEX5-4 shows the product P, HEX2 shows A, HEX0 shows B
module part3 (SW, HEX5, HEX4, HEX2, HEX0);
		input [7:0] SW;
		output [0:6] HEX5, HEX4, HEX2, HEX0;

		wire [3:0] A, B;
		wire [7:0] P;
		wire [3:1] C_b1;  // carries for row that ANDs with B1
		wire [5:2] PP1;   // partial products from row that ANDs with B1
		wire [3:1] C_b2;  // carries for row that ANDs with B2
		wire [6:3] PP2;   // partial products from row that ANDs with B2
		wire [3:1] C_b3;  // carries for row that ANDs with B3

		assign A = SW[7:4];
		assign B = SW[3:0];
		assign P[0] = A[0] & B[0];

		// module fa (a, b, ci, s, co);
		fa b1_a0 (A[1] & B[0], A[0] & B[1], 1'b0,    P[1],   C_b1[1]);
		fa b1_a1 (A[2] & B[0], A[1] & B[1], C_b1[1], PP1[2], C_b1[2]);
		fa b1_a2 (A[3] & B[0], A[2] & B[1], C_b1[2], PP1[3], C_b1[3]);
		fa b1_a3 (1'b0,        A[3] & B[1], C_b1[3], PP1[4], PP1[5]);


		// module fa (a, b, ci, s, co);
		fa b2_a0 (PP1[2], A[0] & B[2], 1'b0,    P[2],   C_b2[1]);
		fa b2_a1 (PP1[3], A[1] & B[2], C_b2[1], PP2[3], C_b2[2]);
		fa b2_a2 (PP1[4], A[2] & B[2], C_b2[2], PP2[4], C_b2[3]);
		fa b2_a3 (PP1[5], A[3] & B[2], C_b2[3], PP2[5], PP2[6]);

		// module fa (a, b, ci, s, co);
		fa b3_a0 (PP2[3], A[0] & B[3], 1'b0,    P[3], C_b3[1]);
		fa b3_a1 (PP2[4], A[1] & B[3], C_b3[1], P[4], C_b3[2]);
		fa b3_a2 (PP2[5], A[2] & B[3], C_b3[2], P[5], C_b3[3]);
		fa b3_a3 (PP2[6], A[3] & B[3], C_b3[3], P[6], P[7]);

		// drive the displays through a 7-seg decoders
		hex7seg digit_5 (P[7:4], HEX5);
		hex7seg digit_4 (P[3:0], HEX4);
		hex7seg digit_3 (A, HEX2);
		hex7seg digit_2 (B, HEX0);

endmodule

module fa (a, b, ci, s, co);
		input a, b, ci;
		output s, co;

		wire a_xor_b;

		assign a_xor_b = a ^ b;
		assign s = a_xor_b ^ ci;
		assign co = (~a_xor_b & b) | (a_xor_b & ci);
endmodule

module hex7seg (hex, display);
		input [3:0] hex;
```

```
	output [0:6] display;

	reg [0:6] display;

	always @ (hex)
		case (hex)
			4'h0: display = 7'b0000001;
			4'h1: display = 7'b1001111;
			4'h2: display = 7'b0010010;
			4'h3: display = 7'b0000110;
			4'h4: display = 7'b1001100;
			4'h5: display = 7'b0100100;
			4'h6: display = 7'b0100000;
			4'h7: display = 7'b0001111;
			4'h8: display = 7'b0000000;
			4'h9: display = 7'b0000100;
			4'hA: display = 7'b0001000;
			4'hb: display = 7'b1100000;
			4'hC: display = 7'b0110001;
			4'hd: display = 7'b1000010;
			4'hE: display = 7'b0110000;
			4'hF: display = 7'b0111000;
		endcase
endmodule
```

PART-4

```
// Implements an unsigned multiplier P = A x B.
// inputs:	SW7-0, SW9 select A, SW8 selects B, KEY[0] is reset, KEY[1] is clock
// outputs:		HEX3-0 shows the product P
module part4 (SW, KEY, LEDR, HEX3, HEX2, HEX1, HEX0);
	input [9:0] SW;
	input [1:0] KEY;
	output [9:0] LEDR;
	output [0:6] HEX3, HEX2, HEX1, HEX0;

	wire Resetn, Clock;
	reg [7:0] A, B;
	reg [15:0] P_reg;
	wire [15:0] PP0, PP1, PP2, PP3, PP4, PP5, PP6;
	wire [15:0] P;

	assign Resetn = KEY[0];
	assign Clock = KEY[1];
	always @(posedge Clock)
	begin
		if (~Resetn)
		begin
			A <= 8'b0; B <= 8'b0; P_reg <= 16'b0;
		end
		else
		begin
			if (SW[9])		A <= SW[7:0];
			if (SW[8])		B <= SW[7:0];
			P_reg <= P;
		end
	end

	assign LEDR[7:0] = SW[9] ? A : (SW[8] ? B : 8'b0);
	assign LEDR[9:8] = SW[9:8];

	assign PP0 = { 8'd0, A & {8{B[0]}} };
	assign PP1 = PP0 + { 7'd0, A & {8{B[1]}}, 1'd0 };
```

```
            assign PP2 = PP1 + { 6'd0, A & {8{B[2]}}, 2'd0 };
            assign PP3 = PP2 + { 5'd0, A & {8{B[3]}}, 3'd0 };
            assign PP4 = PP3 + { 4'd0, A & {8{B[4]}}, 4'd0 };
            assign PP5 = PP4 + { 3'd0, A & {8{B[5]}}, 5'd0 };
            assign PP6 = PP5 + { 2'd0, A & {8{B[6]}}, 6'd0 };
            assign P =   PP6 + { 1'd0, A & {8{B[7]}}, 7'd0 };

            // drive the display through a 7-seg decoder
            hex7seg digit_3 (P_reg[15:12], HEX3);
            hex7seg digit_2 (P_reg[11:8], HEX2);
            hex7seg digit_1 (P_reg[7:4], HEX1);
            hex7seg digit_0 (P_reg[3:0], HEX0);
endmodule

module hex7seg (hex, display);
            input [3:0] hex;
            output [0:6] display;

            reg [0:6] display;

            always @ (hex)
                        case (hex)
                                    4'h0: display = 7'b0000001;
                                    4'h1: display = 7'b1001111;
                                    4'h2: display = 7'b0010010;
                                    4'h3: display = 7'b0000110;
                                    4'h4: display = 7'b1001100;
                                    4'h5: display = 7'b0100100;
                                    4'h6: display = 7'b0100000;
                                    4'h7: display = 7'b0001111;
                                    4'h8: display = 7'b0000000;
                                    4'h9: display = 7'b0001100;
                                    4'hA: display = 7'b0001000;
                                    4'hb: display = 7'b1100000;
                                    4'hC: display = 7'b0110001;
                                    4'hd: display = 7'b1000010;
                                    4'hE: display = 7'b0110000;
                                    4'hF: display = 7'b0111000;
                        endcase
endmodule
```

PART-5

```
// Implements an unsigned multiplier P = A x B.
// inputs:SW7-0, SW9 select A, SW8 selects B, KEY[0] is reset, KEY[1] is clock
// outputs:           HEX3-0 shows the product P
module part5 (SW, KEY, LEDR, HEX3, HEX2, HEX1, HEX0);
            input [9:0] SW;
            input [1:0] KEY;
            output [9:0] LEDR;
            output [0:6] HEX3, HEX2, HEX1, HEX0;

            wire Resetn, Clock;
            reg [7:0] A, B;
            reg [15:0] P_reg;
            wire [15:0] PP0, PP1, PP2, PP3, PP4, PP5, PP6;
            wire [15:0] P;

            assign Resetn = KEY[0];
            assign Clock = KEY[1];
            always @(posedge Clock)
            begin
                        if (~Resetn)
                        begin
```

```
				A <= 8'b0; B <= 8'b0; P_reg <= 16'b0;
			end
			else
			begin
				if (SW[9])		A <= SW[7:0];
				if (SW[8])		B <= SW[7:0];
				P_reg <= P;
			end
		end

		assign LEDR[7:0] = SW[9] ? A : (SW[8] ? B : 8'b0);
		assign LEDR[9:8] = SW[9:8];

		assign PP0 = { 8'd0, A & {8{B[0]}} } + { 7'd0, A & {8{B[1]}}, 1'd0 };
		assign PP1 = { 6'd0, A & {8{B[2]}}, 2'd0 } + { 5'd0, A & {8{B[3]}}, 3'd0 };
		assign PP2 = { 4'd0, A & {8{B[4]}}, 4'd0 } + { 3'd0, A & {8{B[5]}}, 5'd0 };
		assign PP3 = { 2'd0, A & {8{B[6]}}, 6'd0 } + { 1'd0, A & {8{B[7]}}, 7'd0 };
		assign PP4 = PP0 + PP1;
		assign PP5 = PP2 + PP3;
		assign P =   PP4 + PP5;

		// drive the displays through 7-seg decoders
		hex7seg digit_3 (P_reg[15:12], HEX3);
		hex7seg digit_2 (P_reg[11:8], HEX2);
		hex7seg digit_1 (P_reg[7:4], HEX1);
		hex7seg digit_0 (P_reg[3:0], HEX0);
endmodule

module hex7seg (hex, display);
		input [3:0] hex;
		output [0:6] display;

		reg [0:6] display;

		always @ (hex)
			case (hex)
				4'h0: display = 7'b0000001;
				4'h1: display = 7'b1001111;
				4'h2: display = 7'b0010010;
				4'h3: display = 7'b0000110;
				4'h4: display = 7'b1001100;
				4'h5: display = 7'b0100100;
				4'h6: display = 7'b0100000;
				4'h7: display = 7'b0001111;
				4'h8: display = 7'b0000000;
				4'h9: display = 7'b0001100;
				4'hA: display = 7'b0001000;
				4'hb: display = 7'b1100000;
				4'hC: display = 7'b0110001;
				4'hd: display = 7'b1000010;
				4'hE: display = 7'b0110000;
				4'hF: display = 7'b0111000;
			endcase
endmodule
```