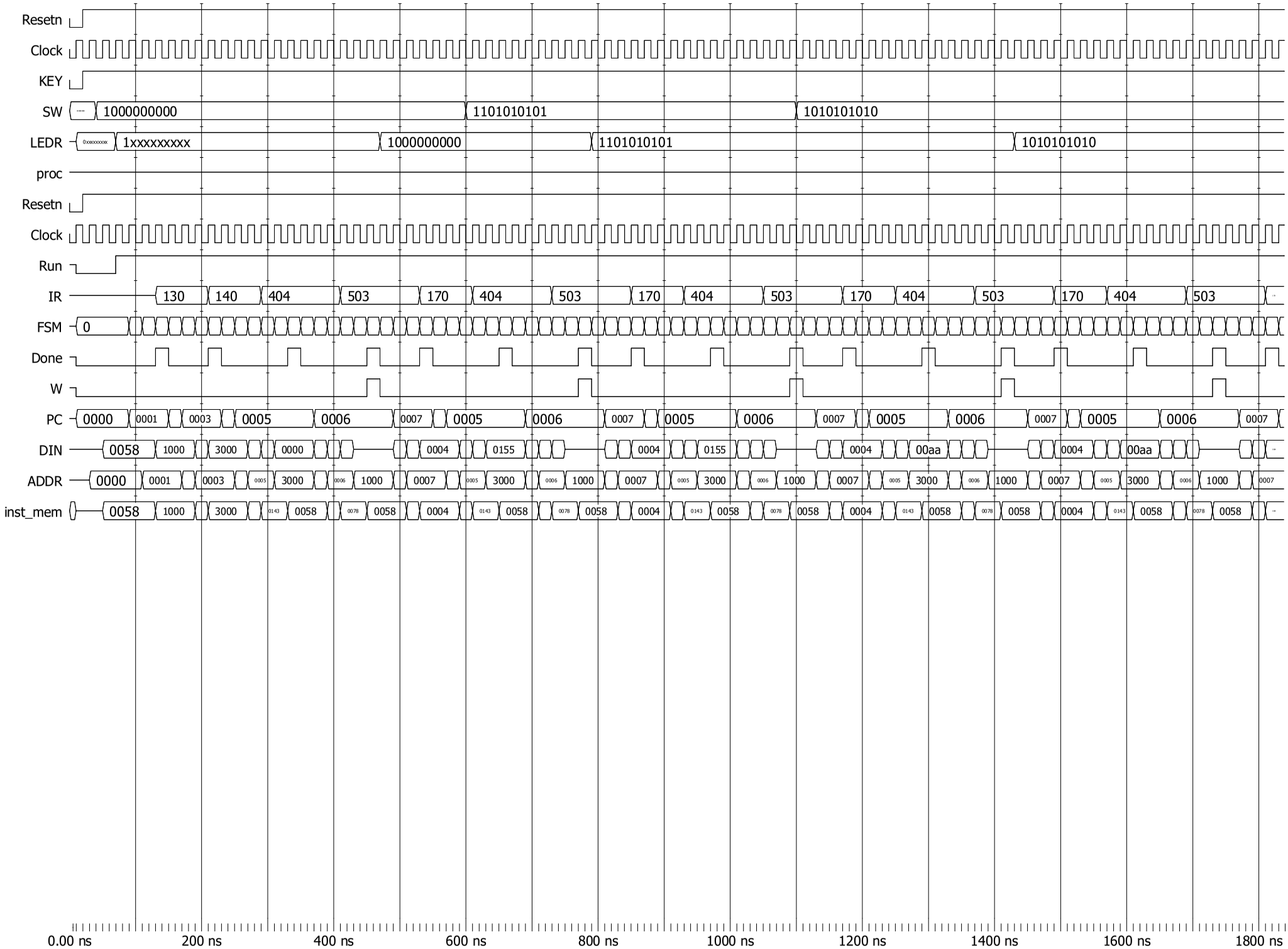
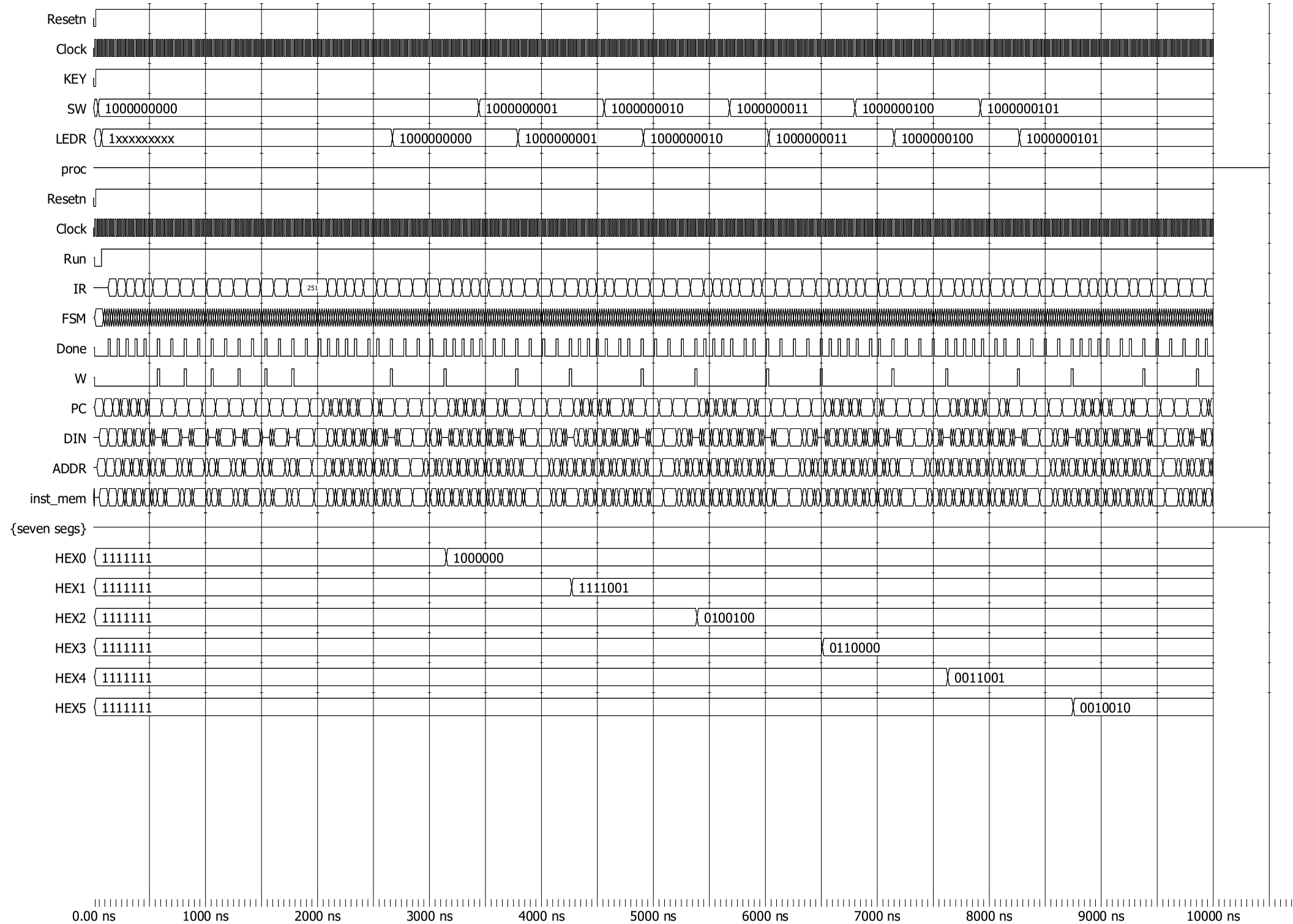
```
.define LED_ADDRESS 0x1000
.define SW_ADDRESS 0x3000
.define MAX_SPEED 0x1000

// This program displays a binary counter on the LED ports
// The speed of the counter is controlled by the switches
            mvi     r0, #0              // Used for counting
            mvi     r1, #1              // Used for add/sub 1
MAIN:       mvi     r4, #SW_ADDRESS     // Point to switches
            ld      r6, [r4]            // Read SW values
            add     r6, r1              // Add 1 for minimum delay

// Count down delay until it reaches 0
DELAY1:     mvi     r5, #MAX_SPEED      // Reset max speed delay counter
            mvi     r3, #DELAY2         // Point to inner delay loop

// Each delay counter will count MAX_SPEED times
DELAY2:     sub     r5, r1              // Count down by 1's
            mvnz    r7, r3              // Continue inner delay loop
// End of DELAY2

            sub     r6, r1              // Count down by 1's
            mvi     r3, #DELAY1         // Point to outer delay loop
            mvnz    r7, r3              // Continue outer delay loop
// End of DELAY1

            add     r0, r1              // Increment binary counter
            mvi     r4, #LED_ADDRESS    // Point to LED port
            st      r0, [r4]            // Display to LED port

            mvi     r7, #MAIN           // Endless looping

```

```
1   .define HEX_ADDRESS 0x2000
2   .define SW_ADDRESS 0x3000
3   .define MAX_SPEED 0x1000
4   .define STACK 255
5   .define COUNT_STORAGE 250
6
7   // This program displays a counter on the 7 segments
8   // The speed of the counter is controlled by the switches
9               mvi     r0, #0              // Used for counting
10              mvi     r1, #1              // Used for add/sub 1
11  MAIN:       mvi     r4, #SW_ADDRESS     // Point to switches
12              ld      r6, [r4]            // Read SW values
13              add     r6, r1              // Add 1 for minimum delay
14
15  // Count down delay until it reaches 0
16  DELAY1:     mvi     r5, #MAX_SPEED      // Reset max speed delay counter
17              mvi     r3, #DELAY2         // Point to inner delay loop
18
19  // Each delay counter will count MAX_SPEED times
20  DELAY2:     sub     r5, r1              // Count down by 1's
21              mvnz    r7, r3              // Continue inner delay loop
22  // End of DELAY2
23
24              sub     r6, r1              // Count down by 1's
25              mvi     r3, #DELAY1         // Point to outer delay loop
26              mvnz    r7, r3              // Continue outer delay loop
27  // End of DELAY1
28
29              add     r0, r1              // Increment counter
30              mvi     r3, #COUNT_STORAGE  // Store the counter because r0 is going to be
                modified
31              st      r0, [r3]
32
33  // Display the counter in decimal
34              mvi     r4, #HEX_ADDRESS    // Point to HEX port
35              mvi     r6, #6              // Used to count number of times DISPLAY must
                loop
36  DISPLAY:    mv      r5, r7              // Return address for DIV10
37              mvi     r7, #DIV10          // Call DIV10 subroutine
38
39              mvi     r5, #DATA           // Used to get display pattern
40
41              add     r5, r0              // Point to correct display pattern
42              ld      r3, [r5]            // Load display pattern
43              st      r3, [r4]            // Light up HEX display
44              add     r4, r1              // Go to next HEX display
45
46              mv      r0, r2              // Move quotient to r0 for next division
47
48              sub     r6, r1              // Decrement number of times DISPLAY still need
                to loop
49              mvi     r3, #DISPLAY        // Point to display loop
50              mvnz    r7, r3              // Keep looping until DISPLAY has looped 6
                times (r6=0)
51  // End of DISPLAY
52
53              mvi     r3, #COUNT_STORAGE  // Get the actual counter back
54              ld      r0, [r3]
55              mvi     r7, #MAIN           // Endless looping
56
57  // subroutine DIV10
58  // This subroutine divides the number in r0 by 10
59  // The algorithm subtracts 10 from r0 until r0 < 10, and keeps count in r2
60  // input: r0
61  // returns: quotient Q in r2, remainder R in r0
62  DIV10:      mvi     r1, #1
63              mvi     r3, #STACK          // Save registers on stack
64              sub     r3, r1              // save registers that are modified
65              st      r6, [r3]
```

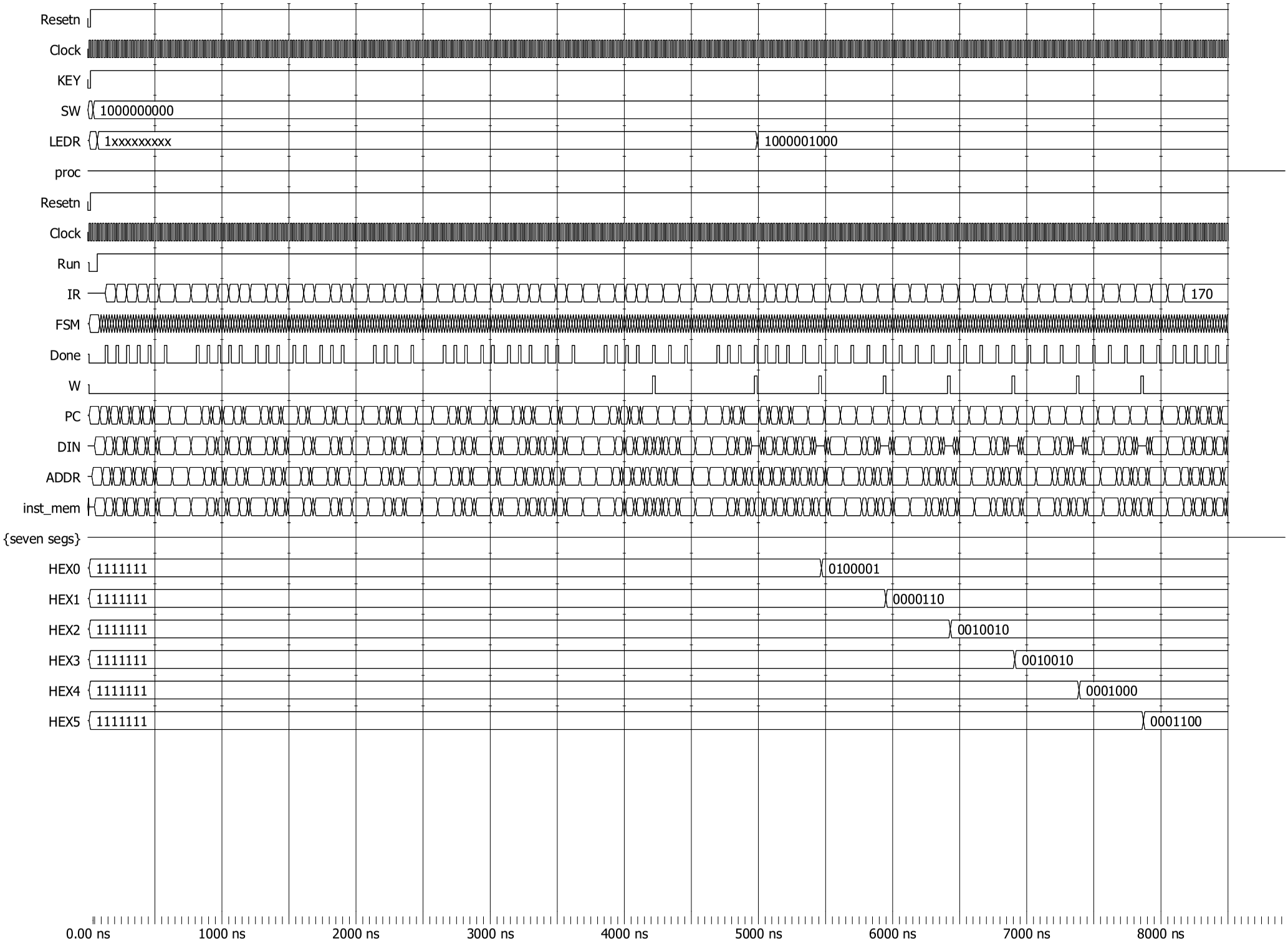
```
66                  sub     r3, r1
67                  st      r4, [r3]            // end of register saving
68                  mvi     r2, #0              // init Q
69                  mvi     r6, RETDIV          // for branching
70
71  DLOOP:          mvi     r4, #9              // check if r0 is < 10 yet
72                  sub     r4, r0
73                  mvnc    r7, r6              // if so, then return
74
75  INC:            add     r2, r1              // but if not, then increment Q
76                  mvi     r4, #10
77                  sub     r0, r4              // r0 -= 10
78                  mvi     r7, DLOOP           // continue loop
79
80  RETDIV:         ld      r4, [r3]            // restore saved regs
81                  add     r3, r1
82                  ld      r6, [r3]            // restore the return address
83                  add     r3, r1
84                  add     r5, r1              // adjust the return address by 2
85                  add     r5, r1
86                  mv      r7, r5              // return results
87
88  // DATA for 7 segments
89  DATA:           .word 0b00111111            // '0'
90                  .word 0b00000110            // '1'
91                  .word 0b01011011            // '2'
92                  .word 0b01001111            // '3'
93                  .word 0b01100110            // '4'
94                  .word 0b01101101            // '5'
95                  .word 0b01111101            // '6'
96                  .word 0b00000111            // '7'
97                  .word 0b01111111            // '8'
98                  .word 0b01101111            // '9'
```



Entity:testbench Architecture: Date: Fri Feb 01 12:27:06 EST 2019 Row: 1 Page: 1



Entity:testbench Architecture: Date: Sun Feb 03 11:04:55 EST 2019 Row: 1 Page: 1



Entity:testbench Architecture: Date: Sun Feb 03 11:20:31 EST 2019 Row: 1 Page: 1

```
// Reset with KEY[0]. SW[9] is Run.
// The processor executes the instructions in the file inst_mem.mif
module part3 (KEY, SW, CLOCK_50, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0, LEDR);
    input [0:0] KEY;
    input [9:0] SW;
    input CLOCK_50;
    output [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
    output [9:0] LEDR;

    wire [15:0] DOUT, ADDR;
    wire Done;
    reg [15:0] DIN;
    wire W, Sync, Run;
    wire inst_mem_cs, SW_cs, LED_reg_cs, seven_seg_cs;
    wire [15:0] inst_mem_q;
    wire [8:0] LED_reg, SW_reg; // LED[9] and SW[9] are used for Run

    // synchronize the Run input
    flipflop U1 (SW[9], KEY[0], CLOCK_50, Sync);
    flipflop U2 (Sync, KEY[0], CLOCK_50, Run);

    // module proc(DIN, Resetn, Clock, Run, DOUT, ADDR, W, Done);
    proc U3 (DIN, KEY[0], CLOCK_50, Run, DOUT, ADDR, W, Done);

    assign inst_mem_cs = (ADDR[15:12] == 4'h0);
    assign LED_reg_cs = (ADDR[15:12] == 4'h1);
    assign seven_seg_cs = (ADDR[15:12] == 4'h2);
    assign SW_cs = (ADDR[15:12] == 4'h3);
    // module inst_mem (data, wren, address, clock, q);
    inst_mem U4 (ADDR[7:0], CLOCK_50, DOUT, inst_mem_cs & W, inst_mem_q);

    always @ (*)
    if (inst_mem_cs == 1'b1)
        DIN = inst_mem_q;
    else if (SW_cs == 1'b1)
        DIN = {7'b0000000, SW_reg};
    else
        DIN = 16'bxxxxxxxxxxxxxxxx;

    // module regn(R, Rin, Clock, Q);
    regn #(.n(9)) U5 (DOUT[8:0], LED_reg_cs & W, CLOCK_50, LED_reg);
    assign LEDR[8:0] = LED_reg;
    assign LEDR[9] = Run;

    // module regn(R, Rin, Clock, Q);
    regn #(.n(9)) U7 (SW[8:0], 1'b1, CLOCK_50, SW_reg); // SW[9] is used for Run

    // 7 Segment display
    // module seg7_scroll (Data, Addr, Sel, Resetn, Clock, H5, H4, H3, H2, H1, H0);
    seg7_scroll HEX(DOUT[6:0], ADDR[2:0], seven_seg_cs & W, KEY[0], CLOCK_50, HEX5,
    HEX4, HEX3, HEX2, HEX1, HEX0);
endmodule
```

```
module proc(DIN, Resetn, Clock, Run, DOUT, ADDR, W, Done);
    input [15:0] DIN;
    input Resetn, Clock, Run;
    output wire [15:0] DOUT;
    output wire [15:0] ADDR;
    output wire W;
    output Done;

    parameter T0 = 3'b000, T1 = 3'b001, T2 = 3'b010, T3 = 3'b011, T4 = 3'b100, T5 = 3'b101;
    reg [15:0] BusWires;
    reg [0:7] Rin, Rout;
    reg [15:0] Sum;
    reg IRin, ADDRin, Done, DINout, DOUTin, Ain, Gin, Gout, AddSub;
    reg [2:0] Tstep_Q, Tstep_D;
    reg z,c; // Zero and carry flags
    reg set_z,set_c;
    wire [2:0] I;
    wire [0:7] Xreg, Yreg;
    wire [15:0] R0, R1, R2, R3, R4, R5, R6, R7 /* pc */, A;
    wire [15:0] G;
    wire [1:9] IR;
    wire [1:10] Sel; // bus selector
    reg pc_inc, W_D;

    assign I = IR[1:3];
    dec3to8 decX (IR[4:6], 1'b1, Xreg);
    dec3to8 decY (IR[7:9], 1'b1, Yreg);

    // Control FSM state table
    always @(Tstep_Q, Run, Done)
    begin
        case (Tstep_Q)
            T0: // instruction fetch
                if (~Run) Tstep_D = T0;
                else Tstep_D = T1;
            T1: // wait cycle for synchronous memory
                Tstep_D = T2;
            T2: // this time step stores the instruction word in IR
                Tstep_D = T3;
            T3: // some instructions end after this time step
                if (Done) Tstep_D = T0;
                else Tstep_D = T4;
            T4: // always go to T5 after this
                Tstep_D = T5;
            T5: // instructions end after this time step
                Tstep_D = T0;
        endcase
    end

    /* Instruction Table
    *   000: mv     Rx,Ry       : Rx <- Ry
    *   001: mvi        Rx,D        : Rx <- D
    *   010: add        Rx,Ry       : Rx <- Rx + Ry
    *   011: sub        Rx,Ry       : Rx <- Rx - Ry
    *   100: ld     Rx,[Ry] : Rx <- [Ry]
    *   101: st     Rx,[Ry] : [Ry] <- Rx
    *   110: mvnz   Rx,Ry       : Rx <- Ry, if G != 0
    *   111: mvnc   Rx,Ry       : Rx <- Ry, if carry-out != 1
    *   OPCODE format: III XXX YYY , where
    *   III = instruction, XXX = Rx, and YYY = Ry. For mvi,
    *   a second word of data is read in the following clock cycle
    */
    // R7 is the program counter
    parameter
        mv = 3'b000, mvi = 3'b001, add = 3'b010, sub = 3'b011, ld = 3'b100, st = 3'b101, mvnz = 3'b110, mvnc = 3'b111;
    // Control FSM outputs
    always @(*)
```

```
68      begin
69          Done = 1'b0; Ain = 1'b0; Gin = 1'b0; Gout = 1'b0; AddSub = 1'b0;
70          IRin = 1'b0; DINout = 1'b0; DOUTin = 1'b0; ADDRin = 1'b0; W_D = 1'b0;
71          Rin = 8'b0; Rout = 8'b0; pc_inc = 1'b0; set_z = 1'b0; set_c = 1'b0;
72          case (Tstep_Q)
73              T0: // fetch the instruction
74                  begin
75                      Rout = 8'b00000001; // R7 is program counter (pc)
76                      ADDRin = 1'b1;
77                      pc_inc = Run; // to increment pc
78                  end
79              T1: // wait cycle for synchronous memory
80                  // in case the instruction turns out to be mvi, read memory
81                  begin
82                      Rout = 8'b00000001; // R7 is program counter (pc)
83                      ADDRin = 1'b1;
84                  end
85              T2: // store instruction on DIN in IR
86                  begin
87                      IRin = 1'b1;
88                  end
89              T3:   //define signals in T3
90                  case (I)
91                      mv: // mv Rx,Ry
92                      begin
93                          Rout = Yreg;
94                          Rin = Xreg;
95                          set_z = 1'b1;
96                          Done = 1'b1;
97                      end
98                      mvi: // mvi Rx,#D
99                      begin
100                         // data is available now on DIN
101                         DINout = 1'b1;
102                         Rin = Xreg;
103                         pc_inc = 1'b1;
104                         Done = 1'b1;
105                     end
106                     add, sub: //add, sub
107                     begin
108                         Rout = Xreg;
109                         Ain = 1'b1;
110                     end
111                     ld: // ld Rx,[Ry]
112                     begin
113                         Rout = Yreg;
114                         ADDRin = 1'b1;
115                     end
116                     st: // st Rx,[Ry]
117                     begin
118                         Rout = Yreg;
119                         ADDRin = 1'b1;
120                     end
121                     mvnz: // mvnz rX,rY
122                     begin
123                         if(!z)
124                         begin
125                             Rout = Yreg;
126                             Rin = Xreg;
127                             Done = 1'b1;
128                         end
129                     end
130                     mvnc: // mvnc rX,rY
131                     begin
132                         if(!c)
133                         begin
134                             Rout = Yreg;
135                             Rin = Xreg;
136                             Done = 1'b1;
```

```
                        end
                    end
                    default: ;
                endcase
            T4:   //define signals T4
                case (I)
                    add: // add
                    begin
                        Rout = Yreg;
                        Gin = 1'b1;
                        set_c = 1'b1;
                    end
                    sub: // sub
                    begin
                        Rout = Yreg;
                        AddSub = 1'b1;
                        Gin = 1'b1;
                        set_c = 1'b1;
                    end
                    ld: // ld Rx,[Ry]
                        ; // wait cycle for synchronous memory
                    st: // st Rx,[Ry]
                    begin
                        Rout = Xreg;
                        DOUTin = 1'b1;
                        W_D = 1'b1;
                    end
                    default: ;
                endcase
            T5:   //define T5
                case (I)
                    add, sub: //add, sub
                    begin
                        Gout = 1'b1;
                        Rin = Xreg;
                        set_z = 1'b1;
                        Done = 1'b1;
                    end
                    ld: // ld Rx,[Ry]
                    begin
                        DINout = 1'b1;
                        Rin = Xreg;
                        Done = 1'b1;
                    end
                    st: // st Rx,[Ry]
                    begin
                        Done = 1'b1;
                    end
                    default: ;
                endcase
            default: ;
        endcase
    end

    // Control FSM flip-flops
    always @(posedge Clock)
        if (!Resetn)
            Tstep_Q <= T0;
        else
            Tstep_Q <= Tstep_D;

    // Registers for z and c flags
    always @(posedge Clock)
    begin
        // z flag
        if(!Resetn) // Active low reset
            z <= 1'b1;
        else if(set_z) // The bus wires will either be G or rY
            z <= (BusWires==0);
```

```
206
207         // c flag
208         // If a carryout is generated then the sum is less than
209         // either value due to the concatenated carry out digit
210         // A borrow is required when A-B goes negative
211         if(!Resetn) // Active low reset
212             c <= 1'b1;
213         else if(set_c & !AddSub) // Addition
214             c <= (A > A+BusWires);
215         else if(set_c & AddSub) // Subtraction
216             c <= (A < BusWires);
217     end
218
219     regn reg_0 (BusWires, Rin[0], Clock, R0);
220     regn reg_1 (BusWires, Rin[1], Clock, R1);
221     regn reg_2 (BusWires, Rin[2], Clock, R2);
222     regn reg_3 (BusWires, Rin[3], Clock, R3);
223     regn reg_4 (BusWires, Rin[4], Clock, R4);
224     regn reg_5 (BusWires, Rin[5], Clock, R5);
225     regn reg_6 (BusWires, Rin[6], Clock, R6);
226
227     // R7 is program counter
228     // module pc_count(R, Resetn, Clock, E, L, Q);
229     pc_count pc (BusWires, Resetn, Clock, pc_inc, Rin[7], R7);
230
231     regn reg_A (BusWires, Ain, Clock, A);
232     regn reg_DOUT (BusWires, DOUTin, Clock, DOUT);
233     regn reg_ADDR (BusWires, ADDRin, Clock, ADDR);
234     regn #(.n(9)) reg_IR (DIN[8:0], IRin, Clock, IR);
235
236     flipflop reg_W (W_D, Resetn, Clock, W);
237
238     //  alu
239     always @(AddSub or A or BusWires)
240         begin
241         if (!AddSub)
242             Sum = A + BusWires;
243        else
244             Sum = A - BusWires;
245         end
246
247     regn #(.n(16)) reg_G (Sum, Gin, Clock, G);
248
249     // define the internal processor bus
250     assign Sel = {Rout, Gout, DINout};
251
252     always @(*)
253     begin
254         if (Sel == 10'b1000000000)
255             BusWires = R0;
256     else if (Sel == 10'b0100000000)
257             BusWires = R1;
258         else if (Sel == 10'b0010000000)
259             BusWires = R2;
260         else if (Sel == 10'b0001000000)
261             BusWires = R3;
262         else if (Sel == 10'b0000100000)
263             BusWires = R4;
264         else if (Sel == 10'b0000010000)
265             BusWires = R5;
266         else if (Sel == 10'b0000001000)
267             BusWires = R6;
268         else if (Sel == 10'b0000000100)
269             BusWires = R7;
270         else if (Sel == 10'b0000000010)
271             BusWires = G;
272     else BusWires = DIN;
273     end
274 endmodule
```

```
275
276
277  module pc_count(R, Resetn, Clock, E, L, Q);
278      input [15:0] R;
279      input Resetn, Clock, E, L;
280      output [15:0] Q;
281      reg [15:0] Q;
282
283      always @(posedge Clock)
284          if (!Resetn)
285              Q <= 9'b0;
286          else if (L)
287              Q <= R;
288          else if (E)
289              Q <= Q + 1'b1;
290  endmodule
291
292  module dec3to8(W, En, Y);
293      input [2:0] W;
294      input En;
295      output [0:7] Y;
296      reg [0:7] Y;
297
298      always @(W or En)
299      begin
300          if (En == 1)
301              case (W)
302                  3'b000: Y = 8'b10000000;
303             3'b001: Y = 8'b01000000;
304                  3'b010: Y = 8'b00100000;
305                  3'b011: Y = 8'b00010000;
306                  3'b100: Y = 8'b00001000;
307                  3'b101: Y = 8'b00000100;
308                  3'b110: Y = 8'b00000010;
309                  3'b111: Y = 8'b00000001;
310              endcase
311          else
312              Y = 8'b00000000;
313      end
314  endmodule
315
316  module regn(R, Rin, Clock, Q);
317      parameter n = 16;
318      input [n-1:0] R;
319      input Rin, Clock;
320      output [n-1:0] Q;
321      reg [n-1:0] Q;
322
323      always @(posedge Clock)
324          if (Rin)
325              Q <= R;
326  endmodule
327
```

```
// Data written to registers R0 to R5 are sent to the H digits
module seg7_scroll (Data, Addr, Sel, Resetn, Clock, H5, H4, H3, H2, H1, H0);
    input [6:0] Data;
    input [2:0] Addr;
    input Sel, Resetn, Clock;
    output [6:0] H5, H4, H3, H2, H1, H0;

    reg [5:0] enable; // Enable signal to decide which 7 segment to write to

    // One hot encode the 7 segment to be written to
    always @(*)
    begin
        case(Addr)
            0: enable = 6'b000001;
            1: enable = 6'b000010;
            2: enable = 6'b000100;
            3: enable = 6'b001000;
            4: enable = 6'b010000;
            5: enable = 6'b100000;
            default: enable = 6'b000000;
        endcase
    end

    // Registers for the data written to each 7 segment display
    regne HEX0(.R(~Data),.Clock(Clock),.Resetn(Resetn),.E(enable[0] & Sel),.Q(H0));
    regne HEX1(.R(~Data),.Clock(Clock),.Resetn(Resetn),.E(enable[1] & Sel),.Q(H1));
    regne HEX2(.R(~Data),.Clock(Clock),.Resetn(Resetn),.E(enable[2] & Sel),.Q(H2));
    regne HEX3(.R(~Data),.Clock(Clock),.Resetn(Resetn),.E(enable[3] & Sel),.Q(H3));
    regne HEX4(.R(~Data),.Clock(Clock),.Resetn(Resetn),.E(enable[4] & Sel),.Q(H4));
    regne HEX5(.R(~Data),.Clock(Clock),.Resetn(Resetn),.E(enable[5] & Sel),.Q(H5));

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'b1}};
        else if (E)
            Q <= R;
endmodule
```