ACA Verilog

Basic gates:

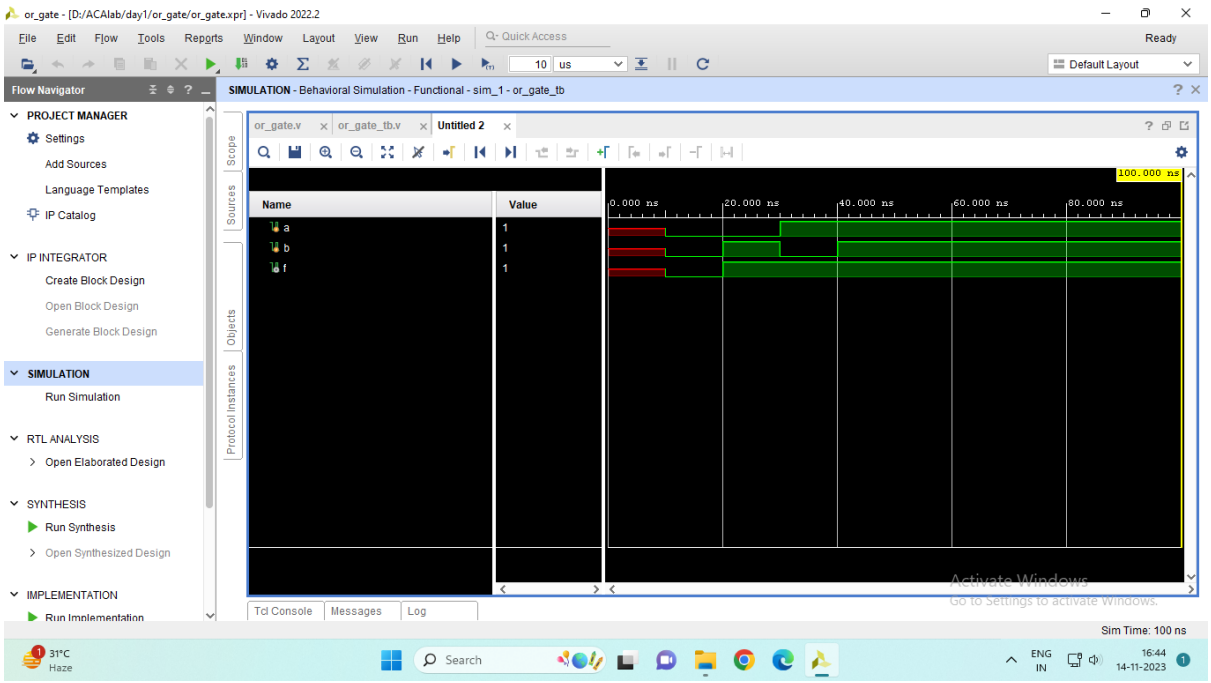
OR Gate:

```
module or_gate(
   input a, b,
   output reg f
   );
    always@(*)
   begin
   case({a,b})
   2'b01: f = 1;
   2'b10: f = 1;
   2'b11: f = 1;
   default: f = 0;
   endcase
   end
endmodule
```

Test Bench:

```
module or_tb;
   reg a, b;
   wire f;
   or_gate uut(a, b , f);
   initial begin
   #10 a=0; b=0;
   #10 a=0; b=1;
   #10 a=1; b=0;
   #10 a=1; b=1;
   #60; $finish;
   end
```

endmodule

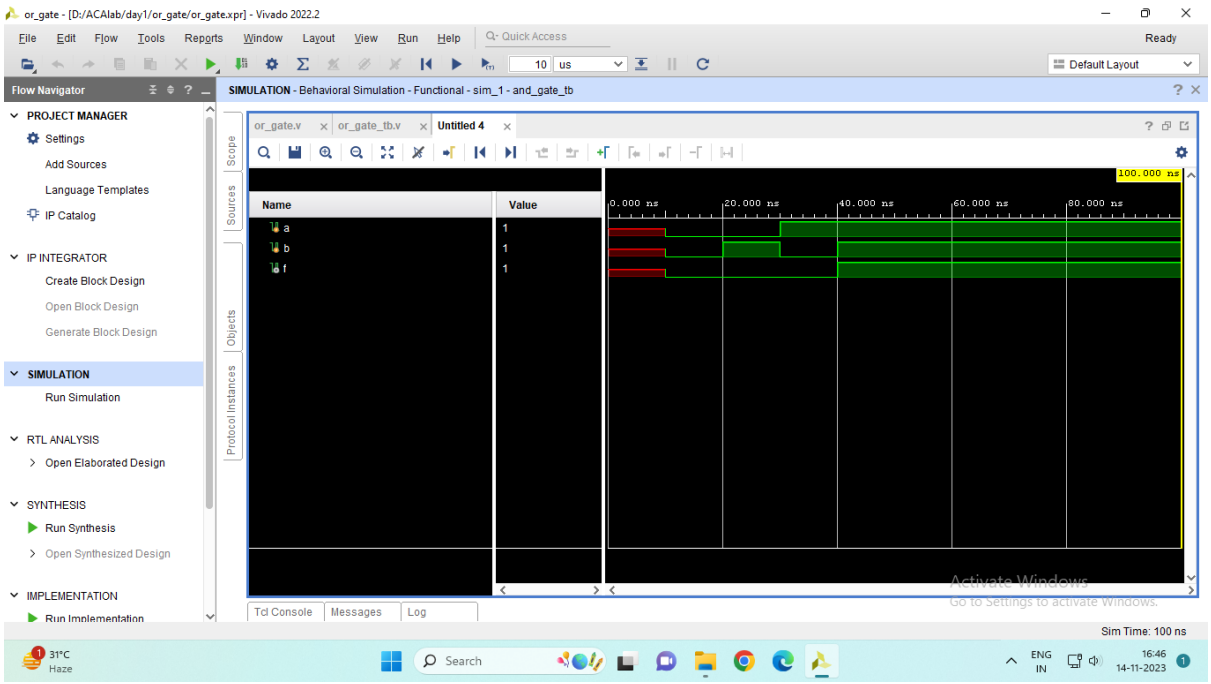


AND Gate:

```
module and_gate(
    input a, b,
    output reg f
    );
    always@(*)
    begin
    case({a,b})
    2'b00: f = 0;
    2'b01: f = 0;
    2'b10: f = 0;
    default: f = 1;
    endcase
    end
endmodule
```

Test Bench:

```
module and_tb;
    reg a, b;
    wire f;
    and_gate uut(a, b , f);
    initial begin
    #10 a=0; b=0;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #60; $finish;
    end
endmodule
```



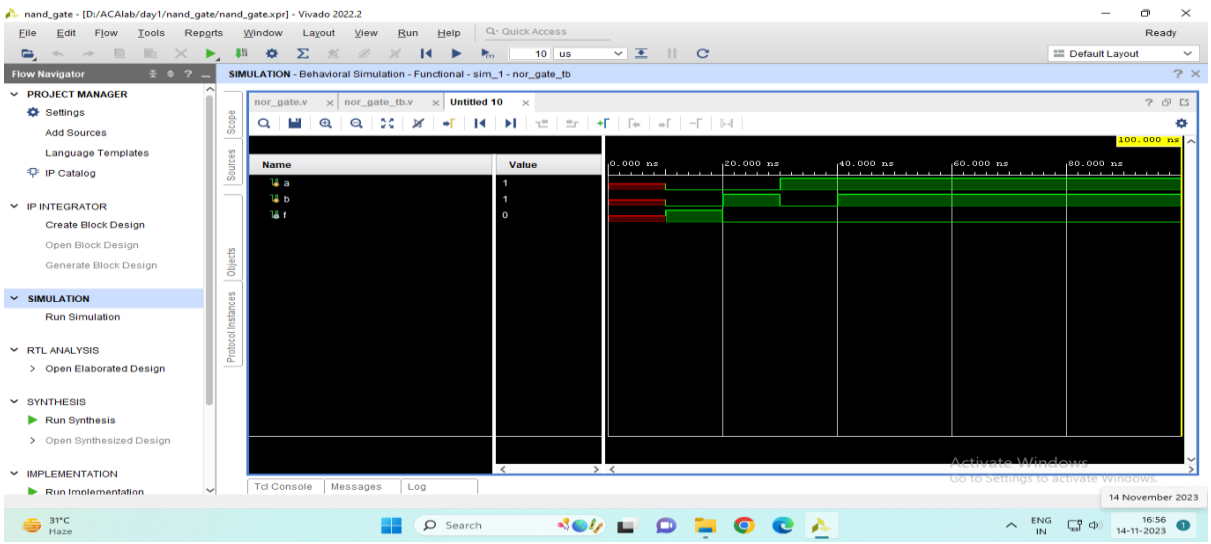
Nor Gate:

```
module nor_gate(
    input a, b,
    output reg f
    );
   always@(*)
    begin
```

```
    case({a,b})
    2'b01: f = 0;
    2'b10: f = 0;
    2'b11: f = 0;
    default: f = 1;
    endcase
    end
endmodule
```

Test Bench:

```
module nor_tb;
    reg a, b;
    wire f;
    nor_gate uut(a, b , f);
    initial begin
    #10 a=0; b=0;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #60; $finish;
    end
endmodule
```



NAND GATE:

```
module nand_gate(
    input a, b,
    output reg f
    );
    always@(*)
    begin
    case({a,b})
    2'b00: f = 1;
    2'b01: f = 1;
    2'b10: f = 1;
    default: f = 0;
    endcase
    end
endmodule
```

Test Bench:

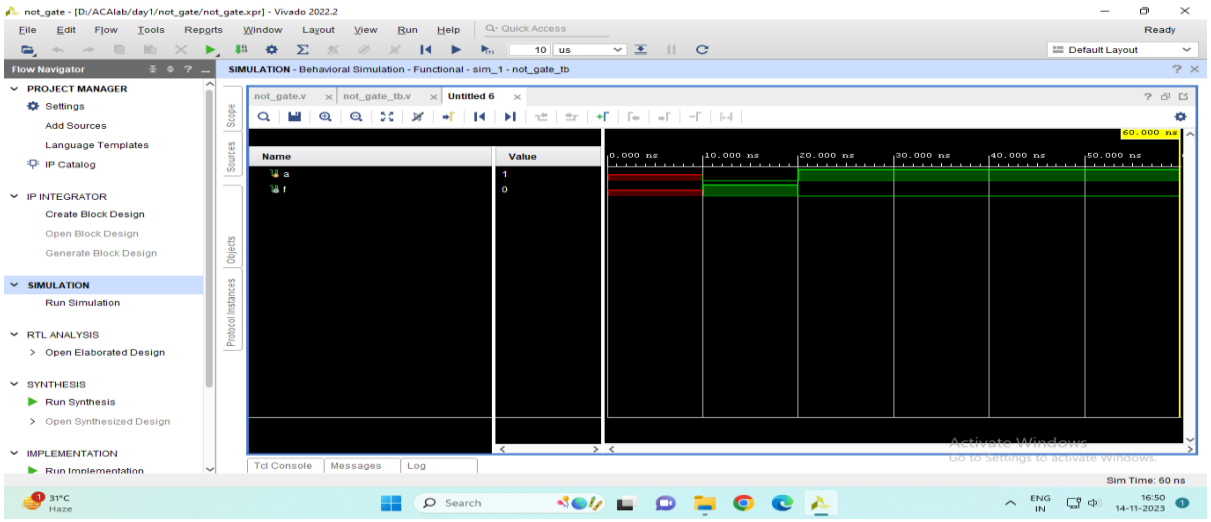
```
module nand_tb;
    reg a, b;
    wire f;
    nand_gate uut(a, b , f);
    initial begin
    #10 a=0; b=0;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #60; $finish;
    end
endmodule
```

NOT GATE:

```
module not_gate(
 input a,
    output reg f
    );
    always@(*)
    begin
    case(a)
    1'b0: f = 1;
    default: f = 0;
    endcase
    end
endmodule
```

Test Bench:

```
module not_tb;
    reg a, b;
    wire f;
    not_gate uut(a, b , f);
    initial begin
    #10 a=0;
    #10 a=1;
    #40; $finish;
    end
endmodule
```



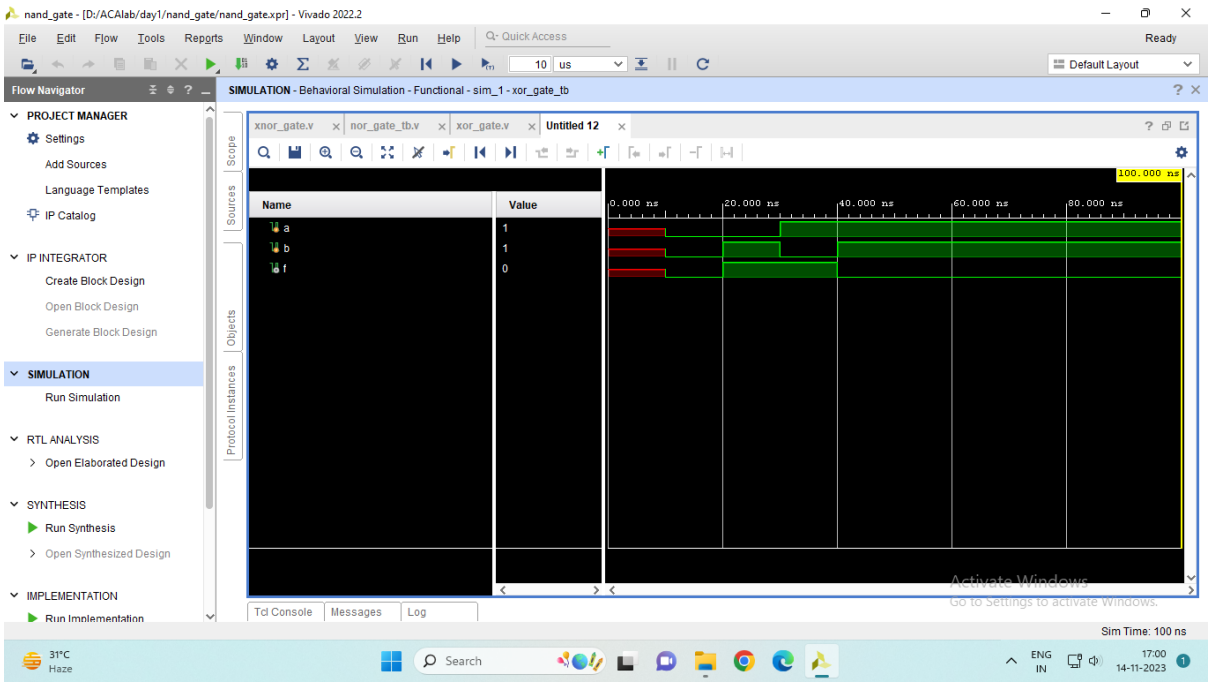
XOR GATE:

```
module xor_gate(
    input a, b,
    output reg f
    );
    always@(*)
    begin
    case({a,b})
    2'b00: f = 0;
    2'b01: f = 1;
    2'b10: f = 1;
    default: f = 0;
    endcase
    end
endmodule
```

Test Bench:

```
module xor_tb;
    reg a, b;
    wire f;
    xor_gate uut(a, b , f);
    initial begin
    #10 a=0; b=0;
```

```
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #60; $finish;
    end
endmodule
```



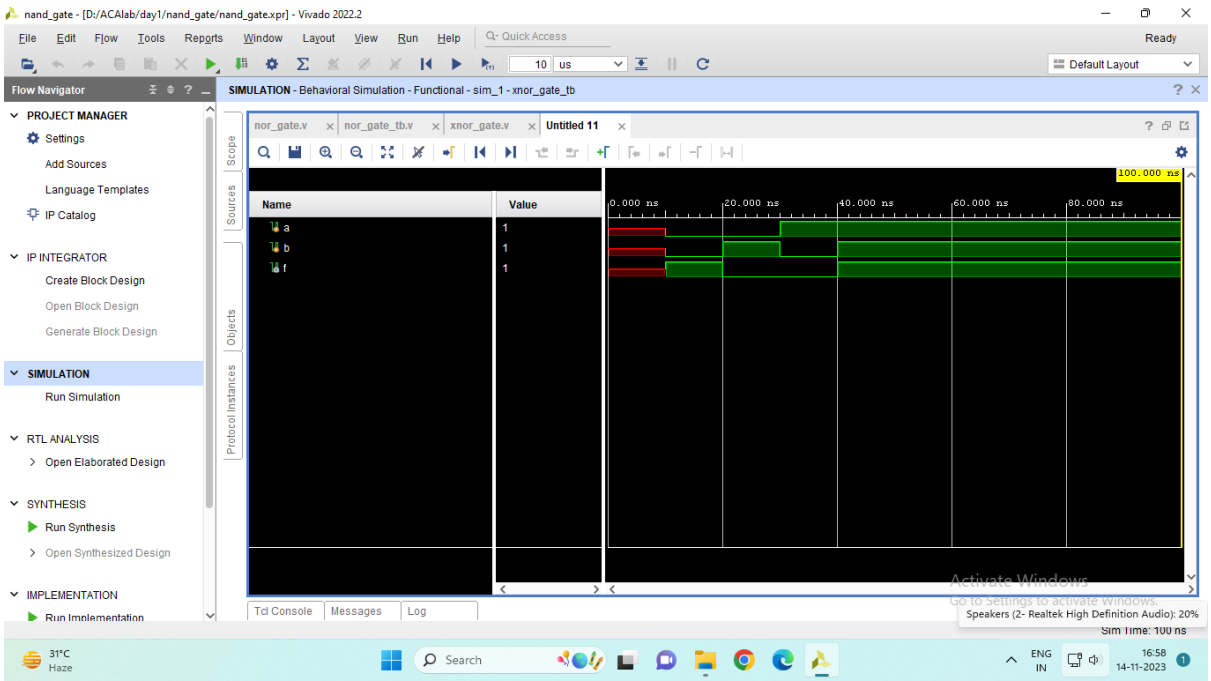
XNOR GATE:

```
module xnor_gate(
    input a, b,
    output reg f
    );
    always@(*)
    begin
    case({a,b})
    2'b00: f = 1;
    2'b01: f = 0;
    2'b10: f = 0;
    default: f = 1;
```

```
    endcase
    end
endmodule
```

Test Bench:

```
module xnor_tb;
    reg a, b;
    wire f;
    xnor_gate uut(a, b , f);
    initial begin
    #10 a=0; b=0;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #60; $finish;
    end
endmodule
```



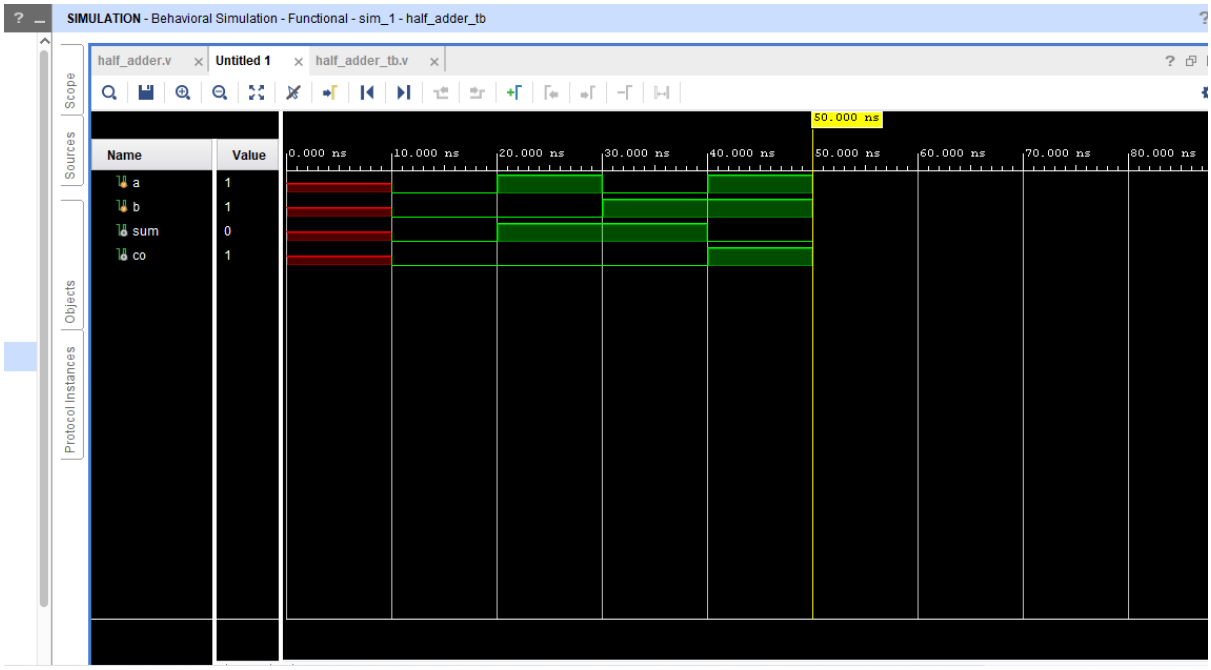
Adder:

Half Adder:

```
module half_adder(
    input a,b,
    output sum,co
    );
    assign sum =a^b;
    assign co=a&b;
endmodule
```

Test Bench:

```
module half_adder_tb;
    reg a,b;
    wire sum,co;
    half_adder uut(a,b,sum,co);
    initial begin
    a=1'b0;
    b=1'b0;
    a=1'b0;
    b=1'b1;
    a=1'b1;
    b=1'b0;
    a=1'b1;
    b=1'b1;
    end
    initial
    #100 $finish;
endmodule
```



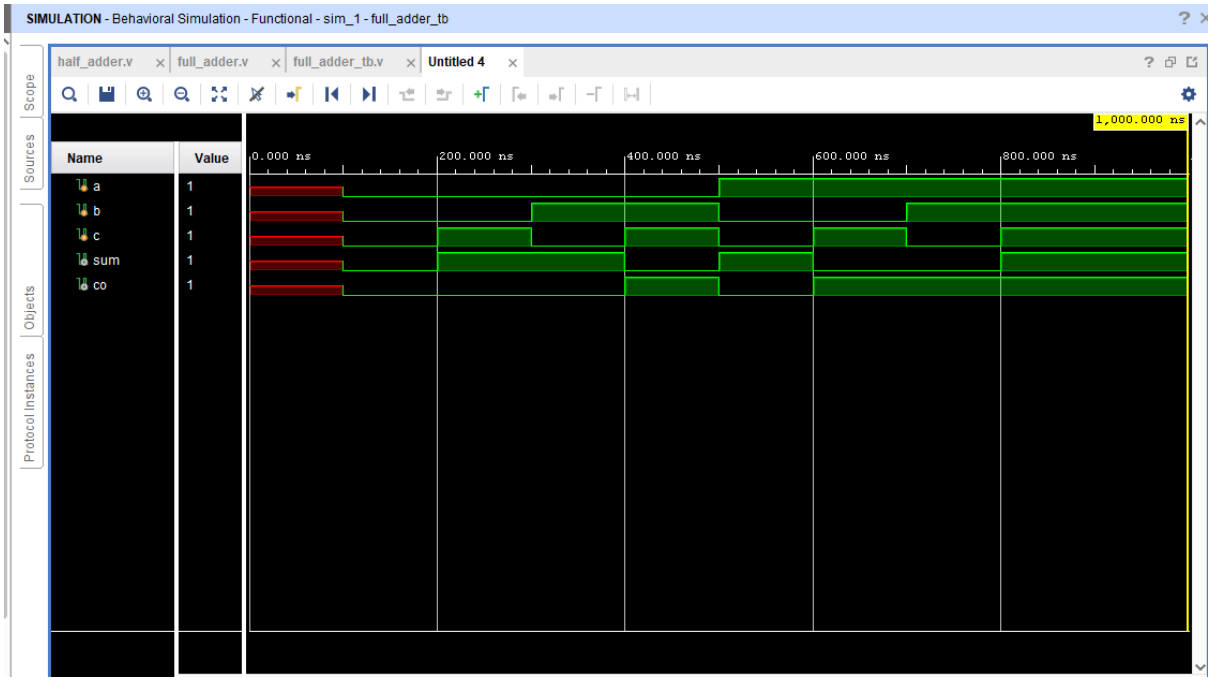
Full Adder:

```
module full_adder(
    input a, b, c,
    output sum, cout
    );
    assign sum = a^b^c;
    assign cout = (a&b) | (b&c) | (a&c);
endmodule
```

Test Bench:

```
module full_adder_tb;
    reg a, b, c;
    wire sum, cout;
    full_adder fa(a, b, c, sum, cout);
    initial begin
    a = 0; b = 0; c = 0; #100;
    a = 0; b = 0; c = 1; #100;
    a = 0; b = 1; c = 0; #100;
    a = 0; b = 1; c = 1; #100;
```

```
    a = 1; b = 0; c = 0; #100;
    a = 1; b = 0; c = 1; #100;
    a = 1; b = 1; c = 0; #100;
    a = 1; b = 1; c = 1; #100;
    end
endmodule
```



FLIP FLOP:

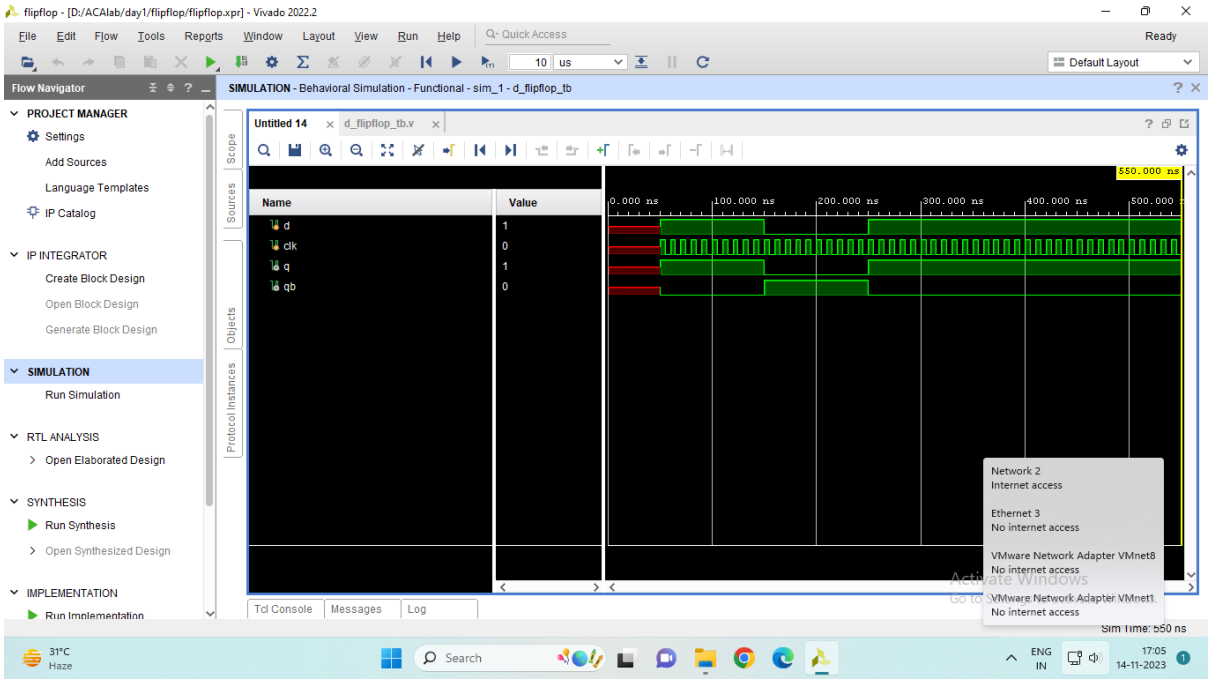
D-FLIP FLOP:

```
module d_flipflop(
    input a, clk,
    output reg q, qb
    );
    always@(posedge clk)
    begin
   if(a == 0)
    begin
   q =0; qb = ~q;
```

```
   end
   else


    begin
    q = 1; qb = ~q;
    end
    end
endmodule
```

Test Bench:

```
module d_flipflop_tb;
  reg d, clk;
    wire q, qb;
    d_flipflop uut(d, clk, q, qb);
    initial begin
    #50 clk = 0; d = 1;
    #50 d = 0;
    #50 d = 1;
    end
    always #5 clk = ~clk;
endmodule
```



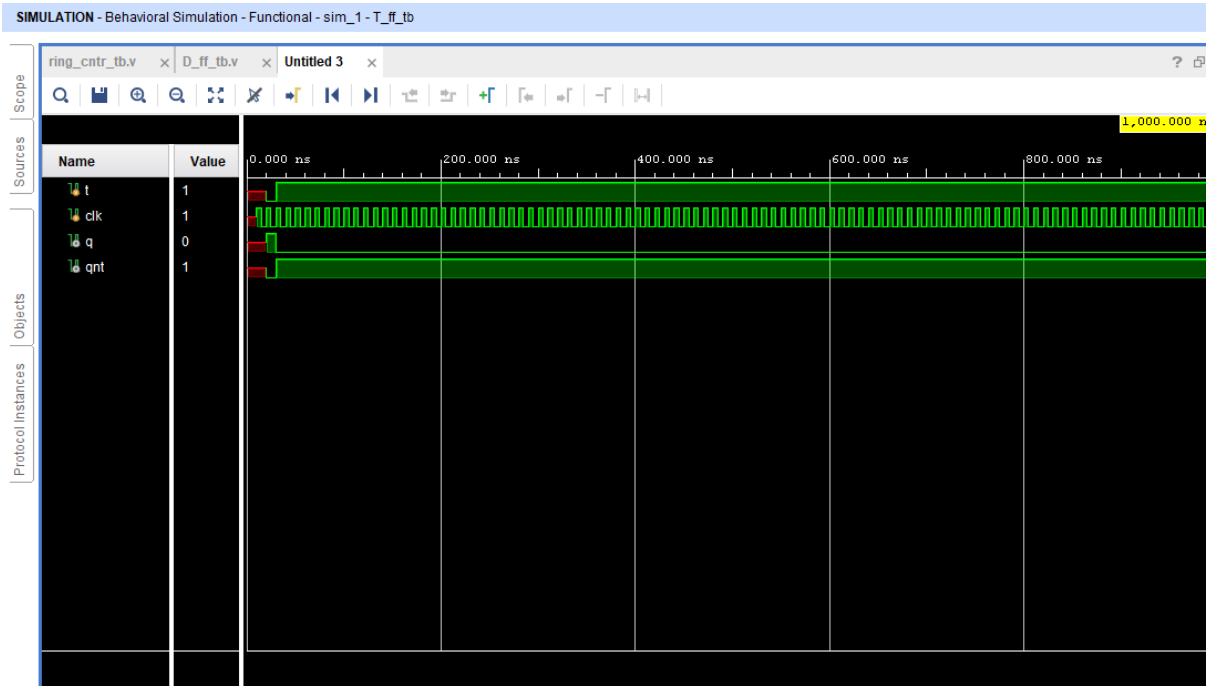
T-FLIP FLOP:

```
module t_flipflop(
    input t, clk,
    output reg q, qb
    );
      always@(posedge clk)
    begin
   if(t == 0)
    begin
   q =1; qb = ~q;
   end
   else
    begin
    q = 0; qb = ~q;
    end
    end
endmodule
```

Test Bench:

```
module t_flipflop_tb;
```

```
    reg t, clk;
    wire q, qb;
    t_flipflop uut(t, clk, q, qb);
    initial begin
    #50 clk = 0; t = 1;
    #50 t = 0;
    #50 t = 1;
    #300 $finish;
    end
    always #5 clk = ~clk;
endmodule
```



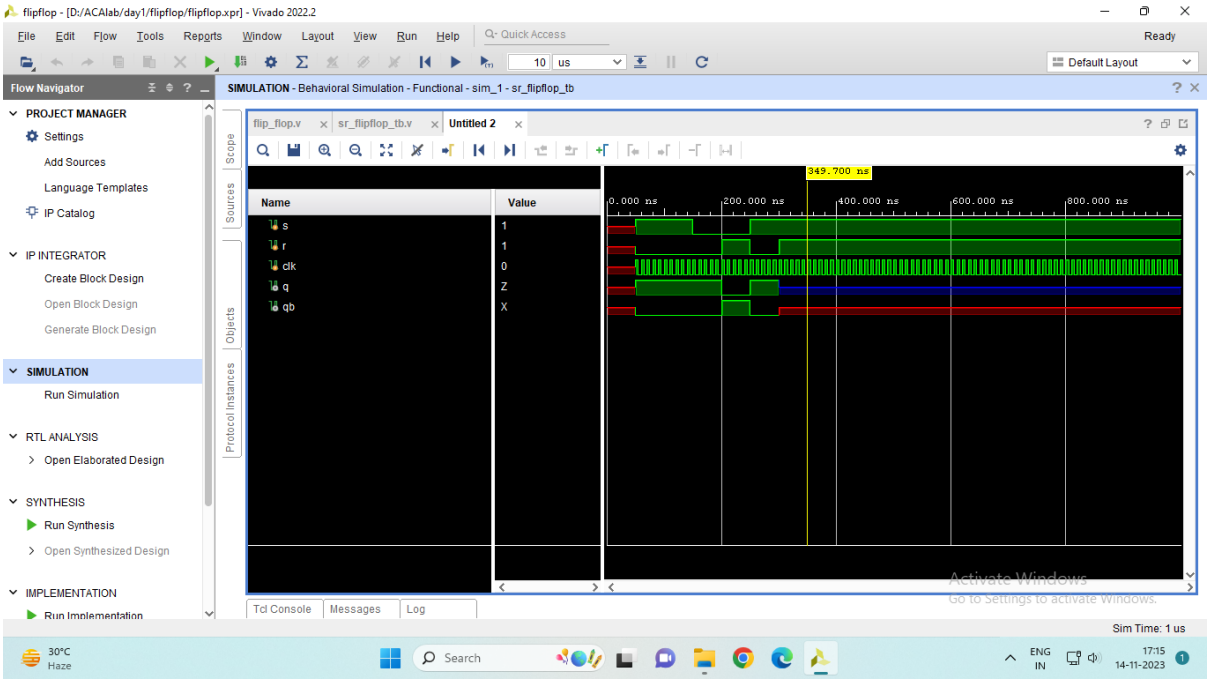
SR Flip Flop:

```
module sr_flip_flop(
    input s, r, clk,
    output reg q, qb
    );
    always@(posedge clk)
    begin
    case({s,r})
```

```
        2'b00: q = q;
        2'b01: q = 0;
        2'b10: q = 1;
        2'b11: q =1'bz;
        endcase
        qb = ~q;
    end
endmodule
```

Test Bench:

```
module sr_flipflop_tb;
    reg s, r, clk;
    wire q, qb;
    sr_flip_flop uut(s, r, clk, q, qb);
    initial begin
    #50 clk = 0; s = 1; r = 0;
    #50 s = 0;  r = 0;
    #50 s = 0;  r = 1;
    #50 s = 1;  r = 0;
    #50 s = 1;  r = 1;
    end
    always #5 clk = ~clk;
endmodule
```



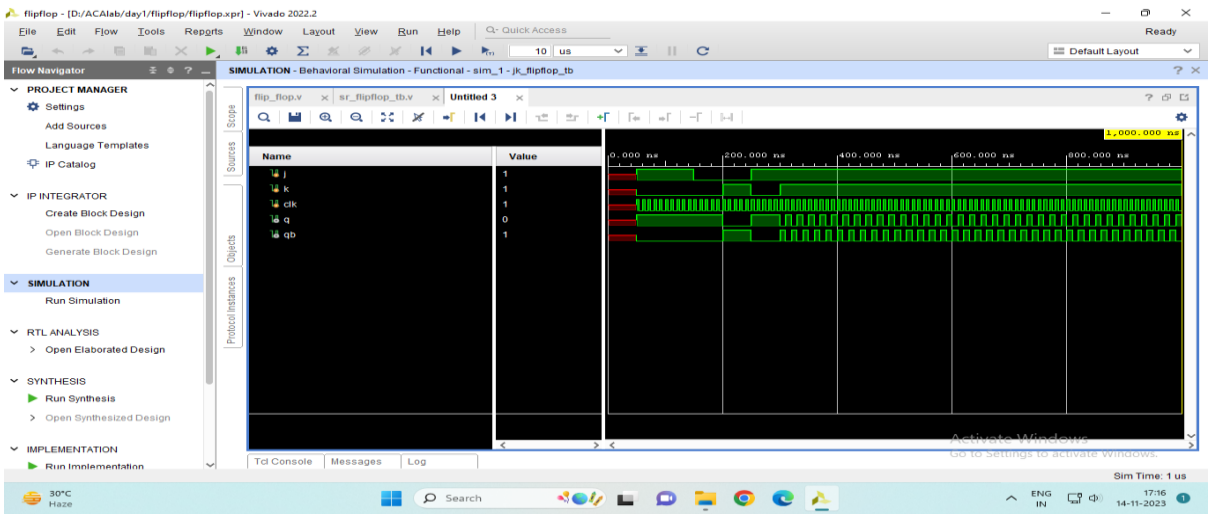
JK Flip Flop:

```
module jk_flipflop(
 input j, k, clk,
    output reg q, qb
    );
    always@(posedge clk)
    begin
    case({j,k})
        2'b00: q = q;
        2'b01: q = 0;
        2'b10: q = 1;
        2'b11: q = ~q;
        endcase
        qb = ~q;
    end
endmodule
```

Test Bench:

```
module jk_flipflop_tb;
    reg j, k, clk;
```

```
    wire q, qb;
    jk_flipflop uut(j, k, clk, q, qb);
    initial begin
    #50 clk = 0; j = 1; k = 0;
    #50 j = 0;  k = 0;
    #50 j = 0;  k = 1;
    #50 j = 1;  k = 0;
    #50 j = 1;  k = 1;
    end
    always #5 clk = ~clk;
endmodule
```



Counters:

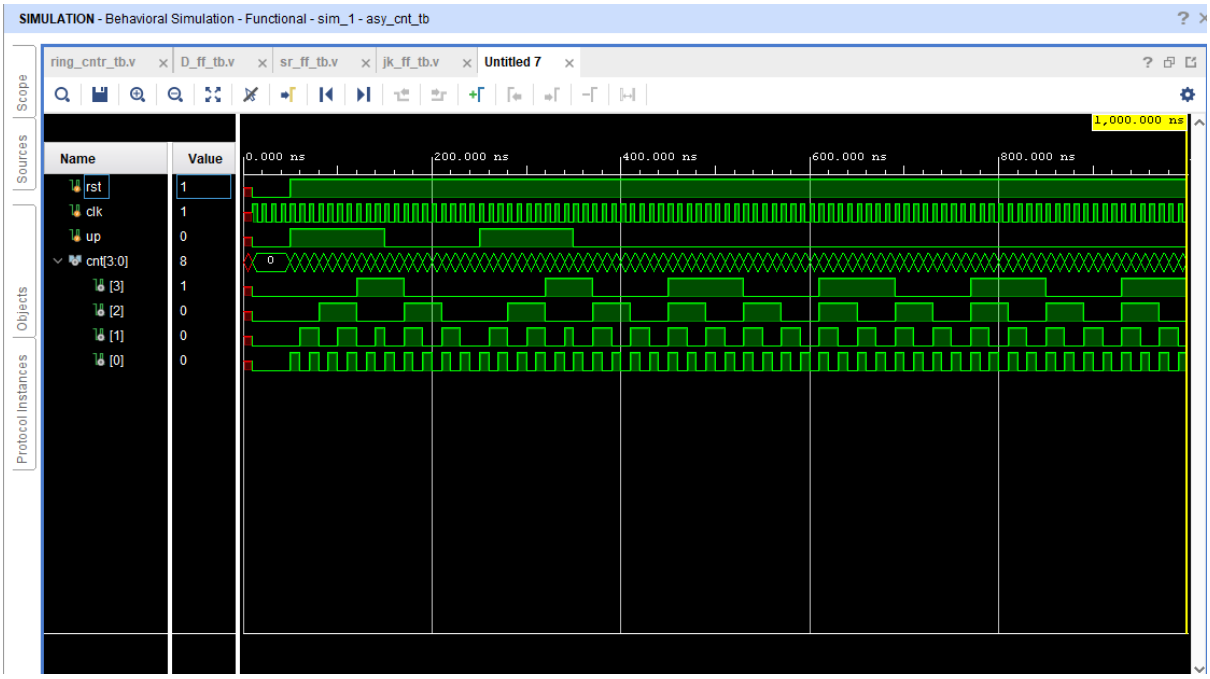
Asynchronous Up Counter and Down Counter:

```
module async_counter(
    input up, clk, rst,
    output reg[3:0] count
    );
    always@(posedge clk or negedge rst)
    begin
    if(rst == 0)
    count = 0;
```

```
    else
    begin
    if(up)
        count = count + 1;
        else if(!up)
        count = count - 1;
     end
     end
endmodule
```

TestBench:

```
module async_counter_tb;
    reg up, clk, rst;
    wire [3:0] count;
    async_counter uut(up, clk, rst, count);
    initial begin
    #50 clk = 0 ; rst = 0; up = 0; #10 rst = 1; up = 1;
    #500 rst = 1; up = 0; #100 rst = 1; up = 1;
    end
    always #5 clk = ~clk;
endmodule
```

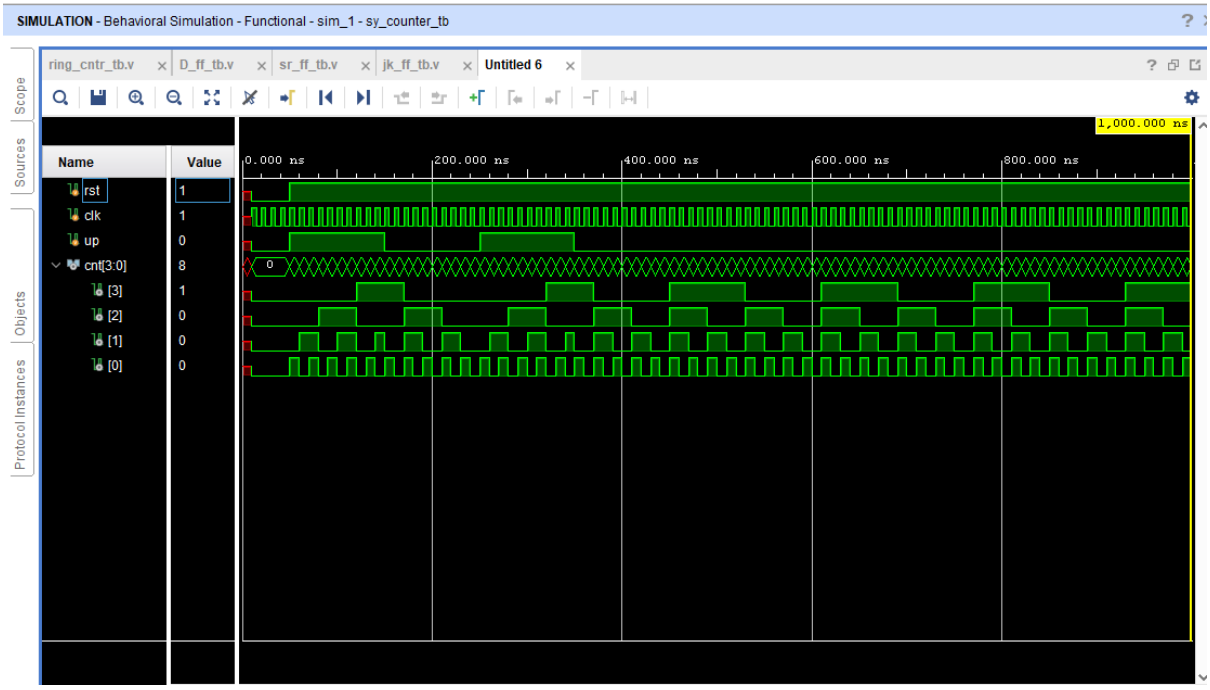


Synchronous Counter Up and down:

```
module sync_counter(
    input clk, rst, up,
    output reg[3:0] count
    );
    always@(posedge clk)
    begin
    if(rst == 0)
    count = 0;
    else
    begin
    if(up)
        count = count + 1;
        else if(!up)
        count = count - 1;
     end
     end
endmodule
```

Test Bench:

```
module sync_counter_tb;
    reg up, clk, rst;
    wire [3:0] count;
    sync_counter uut(clk, rst, up, count);
    initial begin
    #50 clk = 0 ; rst = 0; up = 0;
    #10 rst = 1; up = 1;
    #500 rst = 1; up = 0;
    #100 rst = 1; up = 1;  end
    always #5 clk = ~clk;
endmodule
```



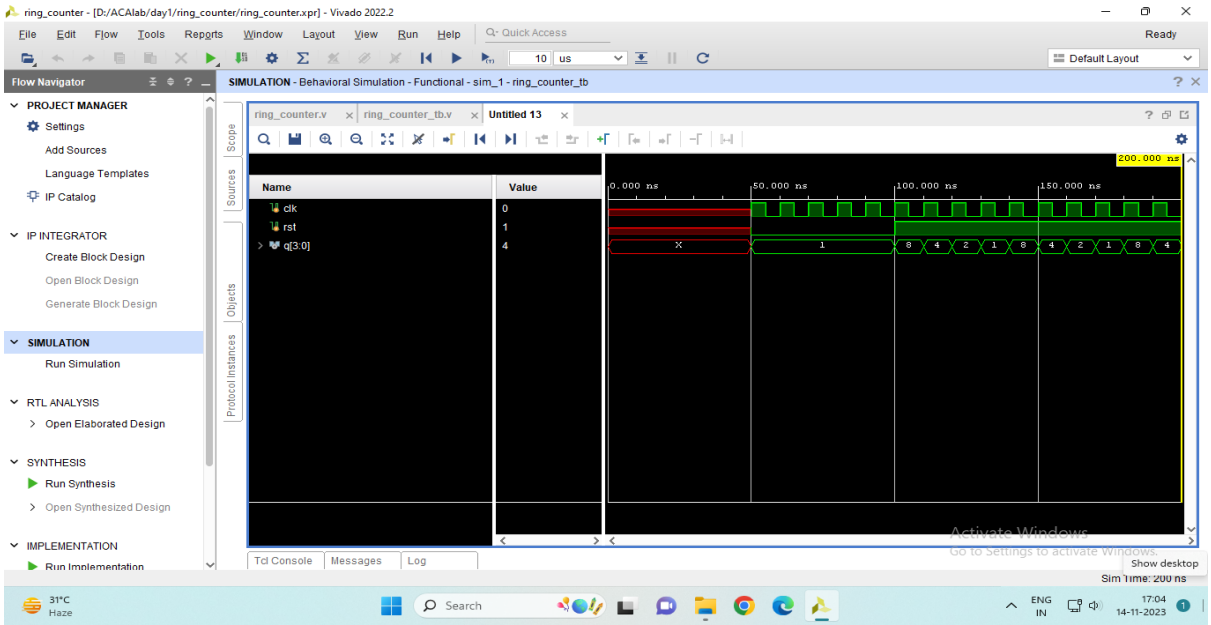
Ring Counter:

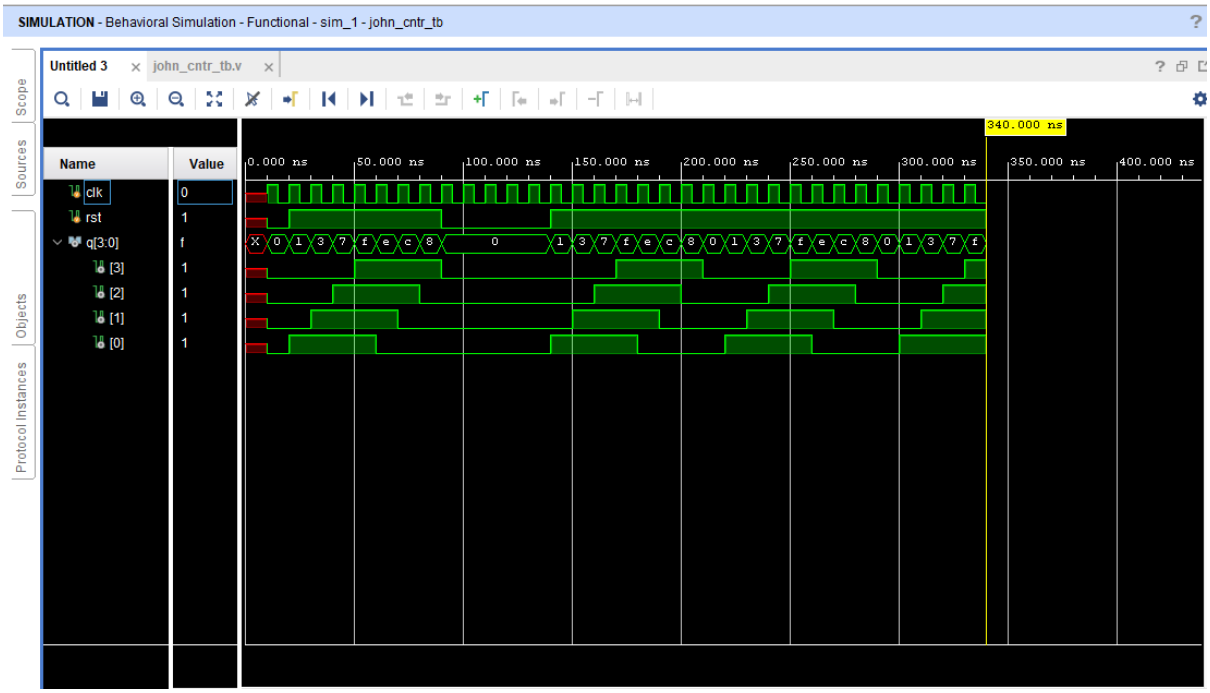
```
module ring_counter(
    input clk, rst,
    output reg [3:0]q
    );
    always@(posedge clk)
        if(!rst)
        q = 4'b0001;
```

```
        else
        begin
        q[0] <= q[1];
        q[1] <= q[2];
        q[2] <= q[3];
        q[3] <= q[0];
        end
endmodule
```

Test Bench:

```
module ring_counter_tb;
    reg clk, rst;
    wire [3:0] q;
    ring_counter uut(clk, rst, q);
    initial begin
    #50 clk = 0 ; rst = 0;
    #10 rst = 1;
end
    always #5 clk = ~clk;
endmodule
```





MULTIPLEXER:

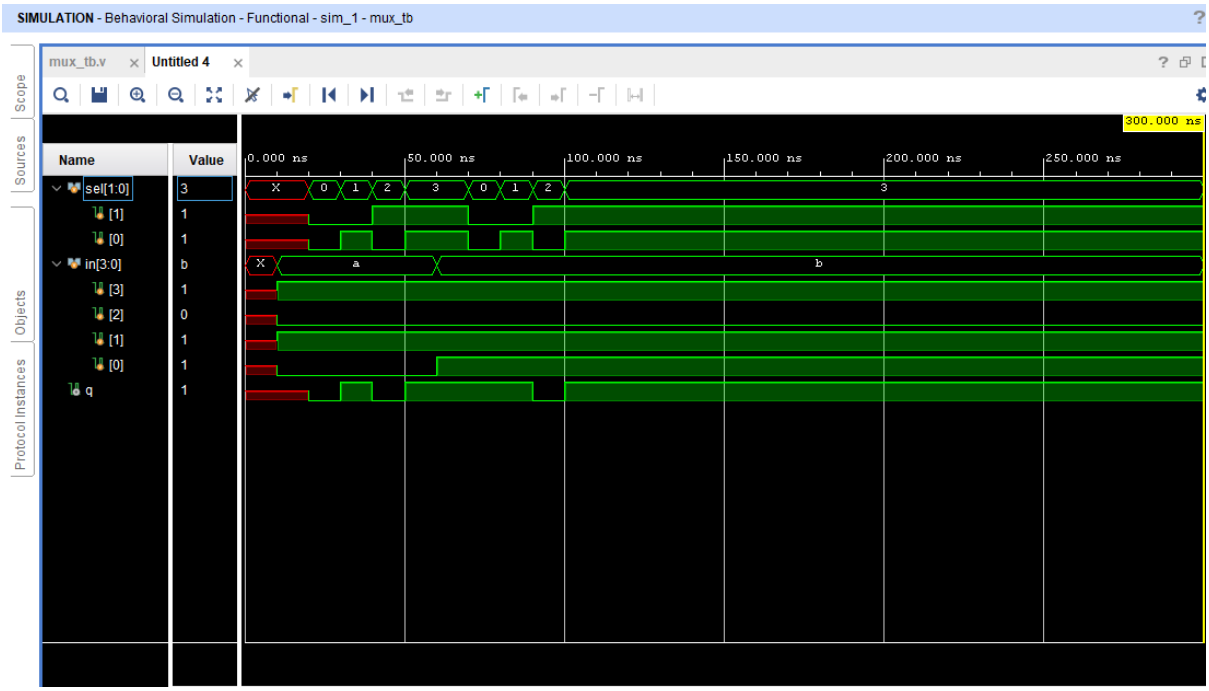
4:1 MUX:

```
module mux_4_1(
   input [1:0]sel, [3:0]i,
   output reg q
   );
   always@(*)
   begin
      case(sel)
        2'b00: q = i[0];
        2'b01: q = i[1];
        2'b10: q = i[2];
     default: q = i[3];
      endcase
   end
endmodule
```

Test Bench:

```
module mux_4_1_tb();
   reg [1:0]sel;
```

```
    reg [3:0]i;
    wire y;
    mux_4_1 uut(sel, i,y);
    initial begin
    #10 i = 4'b0100;
    #10 sel = 2'b00;
    #10 sel = 2'b01;
    #10 sel = 2'b10;
    #10 sel = 2'b11;
    #10 i = 4'b0101;
    #10 sel = 2'b00;
    #10 sel = 2'b01;
    #10 sel = 2'b10;
    #10 sel = 2'b11;
    #10 i = 4'b1100;
    #10 sel = 2'b00;
    #10 sel = 2'b01;
    #10 sel = 2'b10;
    #10 sel = 2'b11;
    #10 i = 4'b0101;
    #10 sel = 2'b00;
    #10 sel = 2'b01;
    #10 sel = 2'b10;
    #10 sel = 2'b11;
    #200; $finish;
    end
endmodule
```



8:1 MUX:

```
module mux8_1(
   input [2:0]sel, [7:0]i,
   output reg q
   );
   always@(*)
   begin
     case(sel)
       3'b000: q = i[0];
       3'b001: q = i[1];
       3'b010: q = i[2];
       3'b011: q = i[3];
       3'b100: q = i[4];
       3'b101: q = i[5];
       3'b110: q = i[6];
       3'b111: q = i[7];
     endcase
   end
endmodule
```

Test Bench:

```
module mux8_1_tb();
    reg [2:0]sel;
    reg [7:0]i;
    wire y;
    mux8_1 uut(sel, i,y);
    initial begin
    #50 i = 8'b01000100;
    #50 sel = 3'b000;
    #50 sel = 3'b001;
    #50 sel = 3'b010;
    #50 sel = 3'b011;
    #50 sel = 3'b100;
    #50 sel = 3'b101;
    #50 sel = 3'b110;
    #50 sel = 3'b111;
    #200; $finish;
    end
endmodule
```

Encoder:

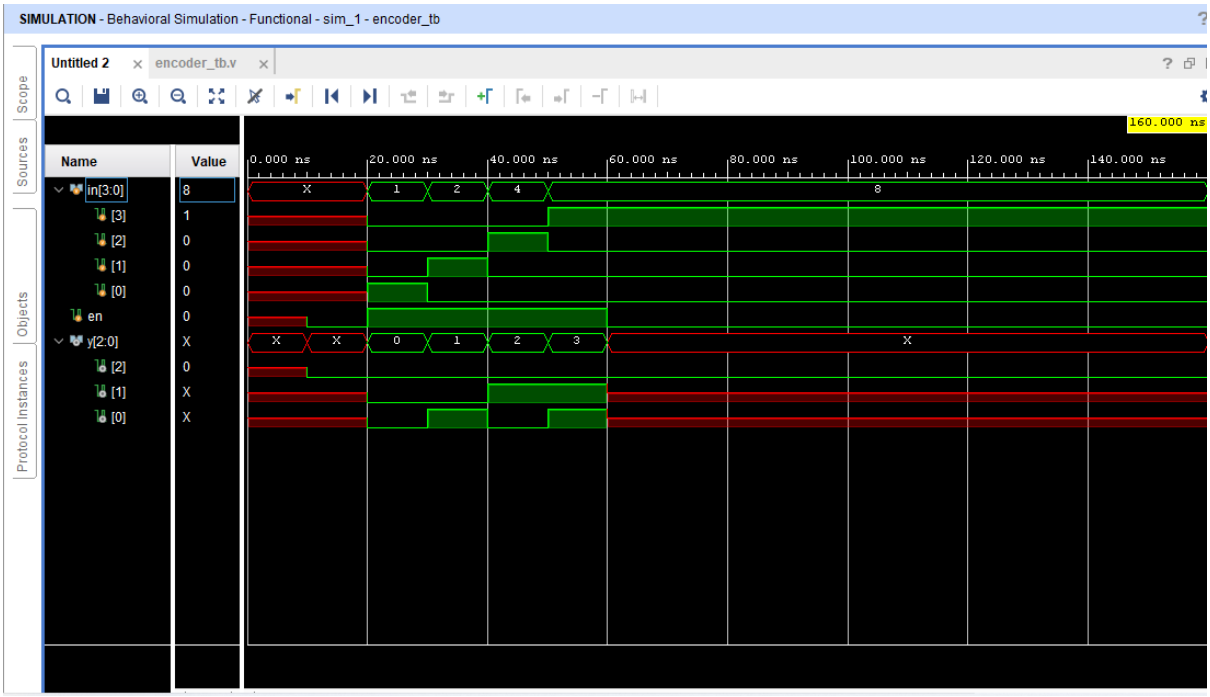
4:2 Encoder:

```
module encoder(
    input [3:0]y,
    input en,
    output reg [1:0]a
    );
    always@(*)
    begin
    if(en)
    case(y)
    4'b0001: a = 2'b00;
    4'b0010: a = 2'b01;
    4'b0100: a = 2'b10;
    4'b1000: a = 2'b11;
    endcase
    else;
    end
```

endmodule

Test Bench:

```
module encoder_4_2_tb;
 reg en;  reg [3:0]y;
 wire [1:0]a;
  encoder uut(en, y, a);
  initial   begin
 #20  en =0; a =0;
  #20 en=1; a = 4'b0001;  #20  a = 4'b0010;
  #20;  a = 4'b0100; #20;  a = 4'b1000;
  #20 $finish;
  end endmodule
```



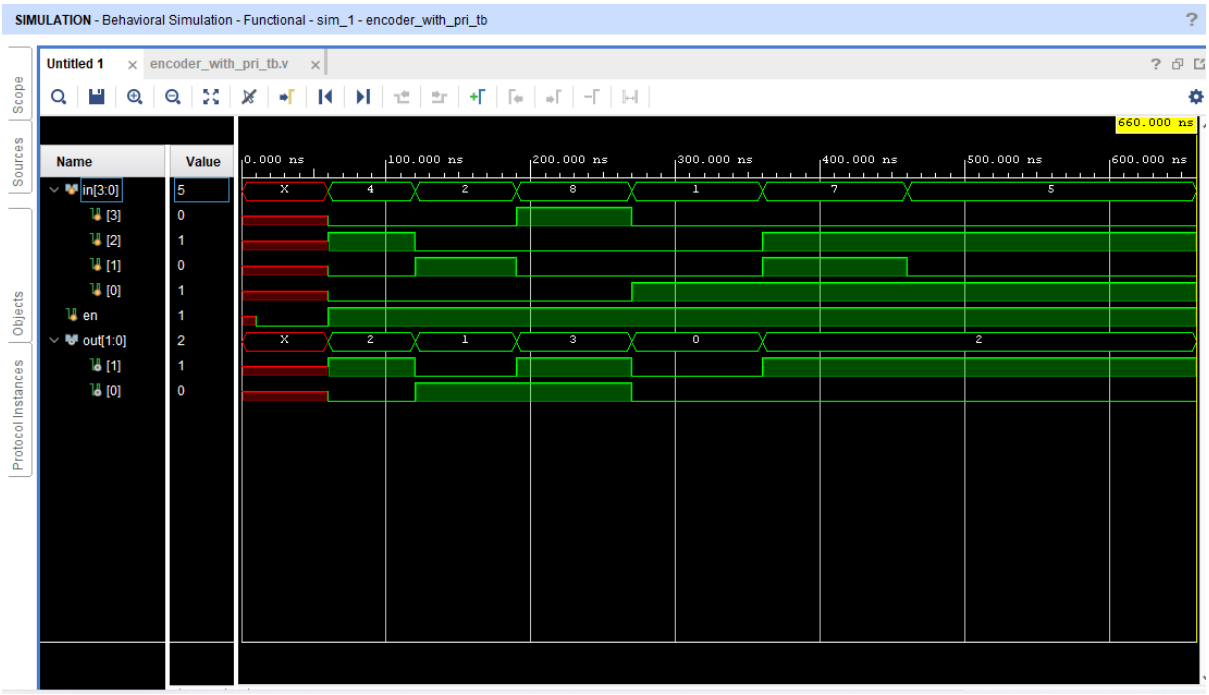
Encode with priority:

```
module encoder_w_p(
   input en,
   input [3:0] a,
   output reg [1:0] y
   );

   always@(en,a)
   if(en==1)
   begin
```

```
    casex(a)
    4'b0001:y=2'b00;
    4'b001x:y=2'b01;
    4'b01xx:y=2'b10;
    4'b1xxx:y=2'b11;
    default:y=2'bxx;

    endcase
    end
    else
    y=2'bzz;
endmodule
```

Test bench

```
module encoder_tb;
reg en;
reg [3:0]a;
wire [1:0]y;

encoder_w_p uut(en,a,y);

initial begin

en=0;a=0;
#10 en=1;a=4'b0001;
#10 a=4'b0010;
#10 a=4'b0111;
#10 a=4'b1010;
#100 $finish;
end
endmodule
```



Identify the data stream of 1010 from given series of data:

(Melay)

```
module sequence(
   input wire clk,reset,
   input wire input_bit,
   output reg[1:0] state,
   output reg out
   );
   always@(posedge clk or posedge reset)
   begin
      if(reset == 0)
      begin
         state <= 2'b00;
         out <=0;
      end
      else
      begin
      case(state)
       2'b00 :begin if(input_bit) state = 2'b01; else state = 2'b00; end
```

```
        2'b01 :begin if(!input_bit) state = 2'b10;  else state = 2'b00; end
         2'b10 :begin if(input_bit) state <= 2'b11;  else state = 2'b00; end
       2'b11 :begin if(!input_bit) begin
         state <= 2'b00;
           out <= 1; end
        out <= 1; end
       endcase
       end
       end
endmodule
```

Test Bench:

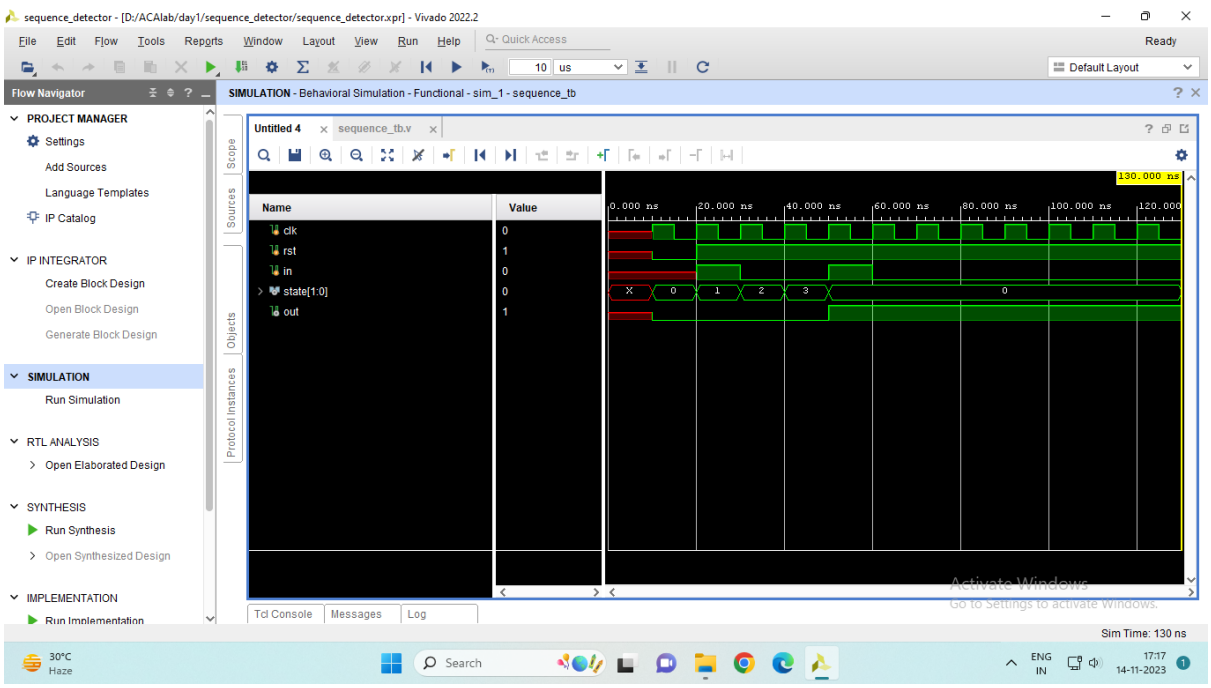
```
module sequential_tb;
  reg clk, reset;
  reg input_bit;
  wire [1:0]state;
  wire out;

  sequence uut(clk, reset, input_bit,state, out);
  initial begin
  #10  reset = 0; clk=0;
  #10 reset = 1; input_bit = 1;
 #10 input_bit = 0;
 #10 input_bit = 1;
 #10 input_bit = 0;
 #70 $finish;
  end
  always #5 clk = ~clk;
endmodule
```

Identify the data stream of 1001 from given series of data: (Melay)

```
module sequence(
   input wire clk,reset,
   input wire input_bit,
   output reg[1:0] state,
   output reg out
   );

   always@(posedge clk or posedge reset)
   begin
      if(reset == 0)
      begin
         state <= 2'b00;
         out <=0;
      end
      else
      begin
      case(state)
         2'b00 :begin if(input_bit) state = 2'b01; else state = 2'b00; end
         2'b01 :begin if(!input_bit) state = 2'b10;  else state = 2'b00; end
         2'b10 :begin if(!input_bit) state <= 2'b11;  else state = 2'b00; end
         2'b11 :begin if(input_bit) begin
         state <= 2'b00;
         out <= 1; end
         out <= 1; end
      endcase
      end
      end
endmodule
```

Test Bench:

```
module sequential_tb;
   reg clk, reset;
   reg input_bit;
   wire [1:0]state;
   wire out;
   sequence uut(clk, reset, input_bit,state, out);
   initial begin
   #10  reset = 0; clk=0;
#10 reset = 1; input_bit = 1;
 #10 input_bit = 0;
 #10 input_bit = 0;
#10 input_bit = 1;
#10 input_bit = 0;
#70 $finish;
   end
   always #5 clk = ~clk;
endmodule
```



Following is the Verilog code for an 4-bit shift-left register with a positive-edge clock, serial in and serial out.

```
module shift(
   input clk, si,rst,
   output so,
   output reg [3:0] tmp
   );
   always@(posedge clk)
   begin
   if(rst == 0)
      tmp = 4'b0000;
    else
      begin
      tmp <= tmp << 1;
      tmp[0] <= si;
      end
   end
      assign so = tmp[3];
endmodule
```

Test Bench:

```
module shift_tb;
reg clk, si, rst;
wire so;
wire[3:0] tmp;
shift uut(clk,si,rst,so,tmp);
initial begin
#10 clk = 0; rst = 0;#10 rst = 1; si = 1;
#10 si = 0; #10 si = 1;#10 si = 1;#10 rst = 0;
#70 $finish; end always #5 clk = ~clk;
endmodule
```

Following is the Verilog code for an 4-bit shift-left register with a negative-edge clock, a clock enable, a serial in and a serial out.

```
module neg_shift(
   input clk, ce, si, rst,
   output reg so,
   output reg[3:0]tmp
   );
   always @(posedge rst or negedge clk)
   begin
      if(ce == 1)
      begin
         if(rst == 0)
             tmp = 4'b1111;
         else
          begin
             tmp <= tmp << 1;
             tmp[0] <= si;
          end
       end
   so = tmp[3];
   end
endmodule
```

Test Bench:

```
module neg_shift_tb;
reg clk, si,rst,ce;
wire so;
wire[3:0] tmp;
neg_shift uut(clk, ce, si, rst,so,tmp);
initial begin
#10 clk = 0;ce = 1; rst = 0;
#10  rst = 1; si = 1;
```

#10 si = 0;

#10 si = 1;

#10 si = 1;

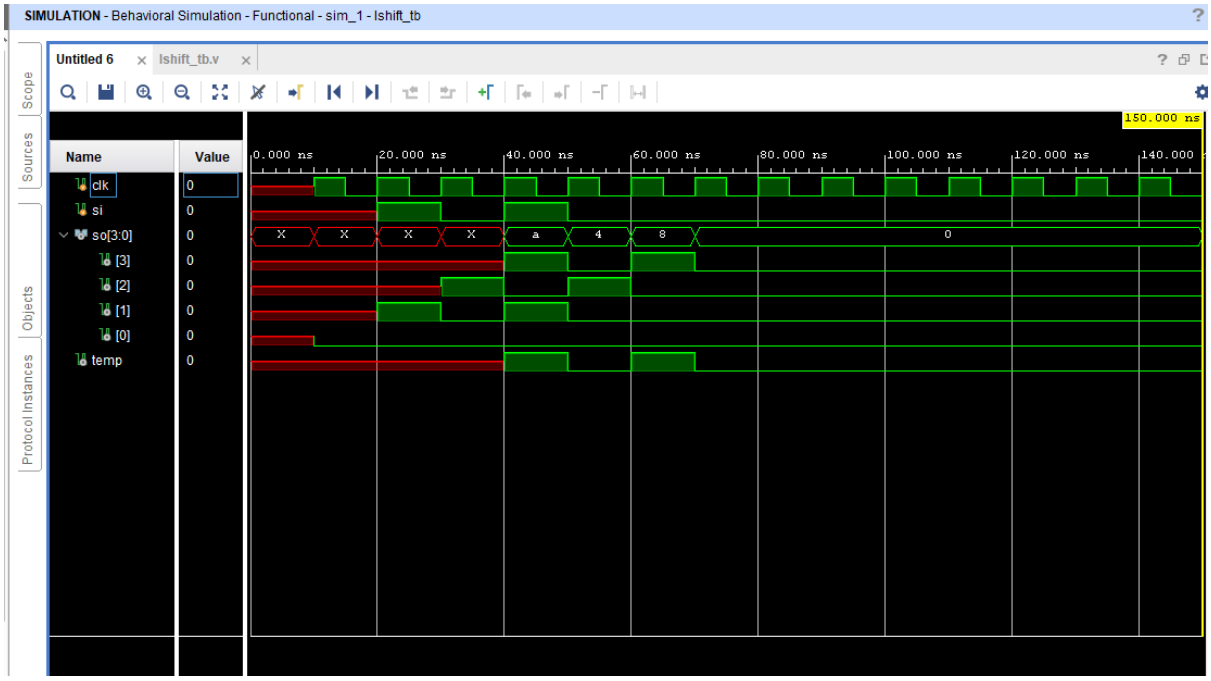
#10 rst = 0;

#150 $finish;

end

always #5 clk = ~clk;

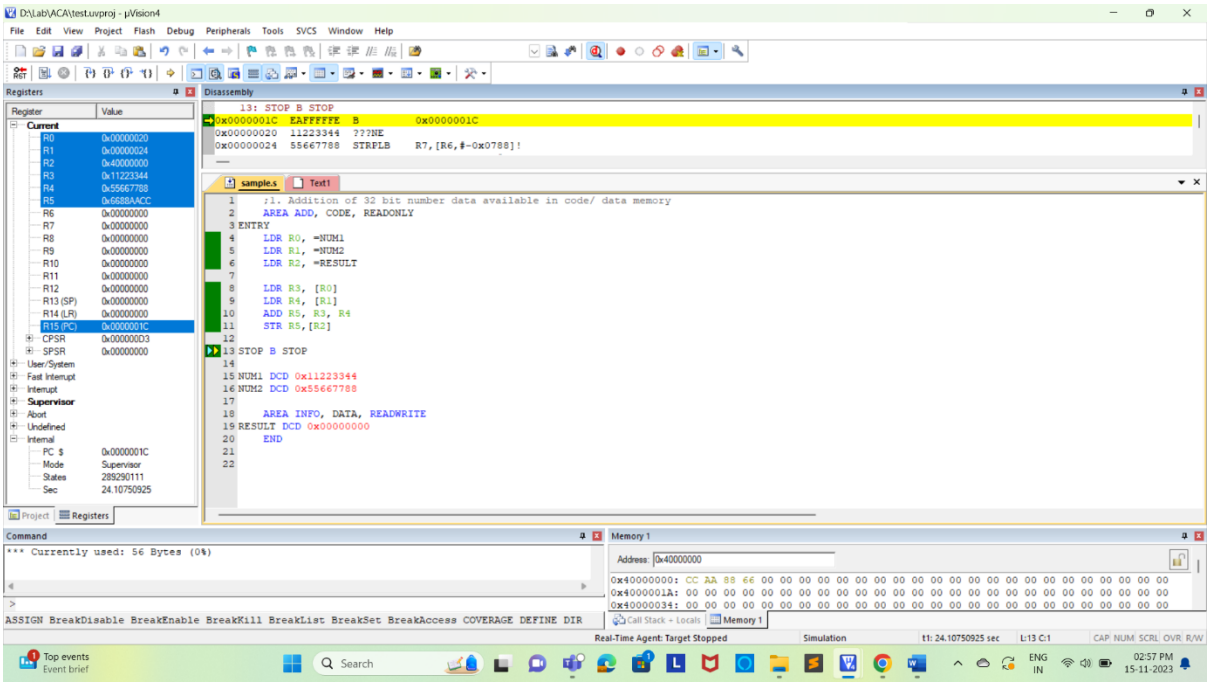
endmodule



Assembly Code:

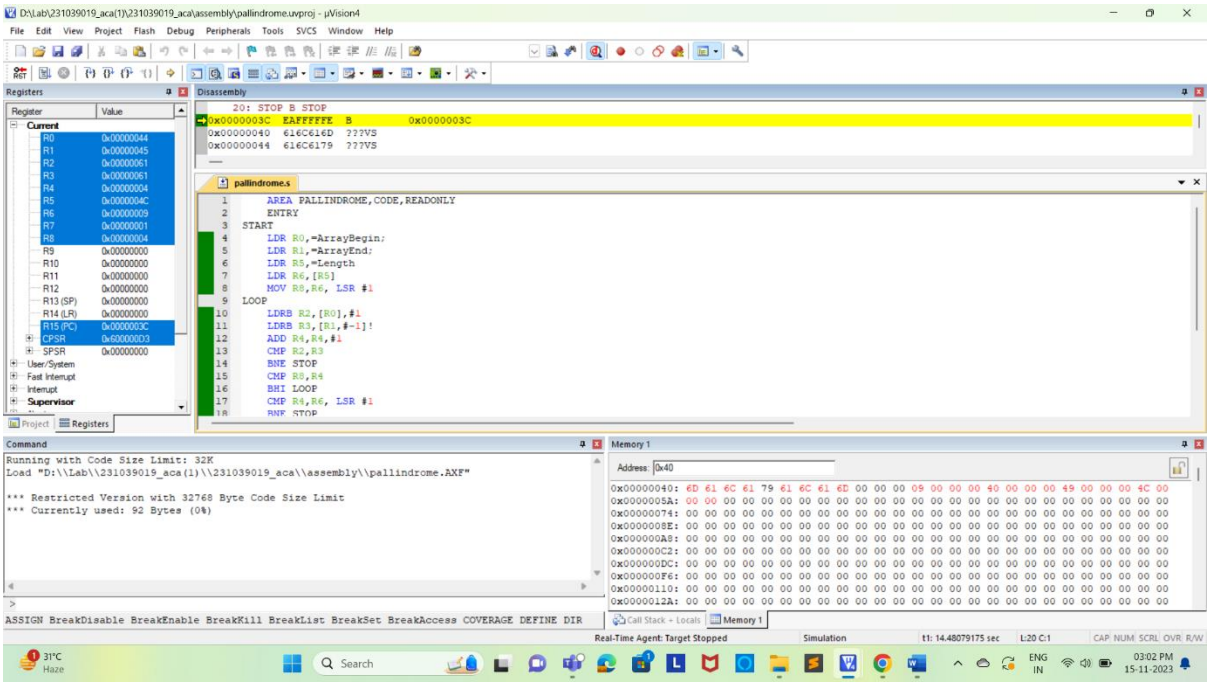
1. **Addition of 32 bit number data available in code/ data memory**

```
        ;1. Addition of 32 bit number data available in code/ data memory
        AREA ADD, CODE, READONLY
ENTRY
        LDR R0, =NUM1
        LDR R1, =NUM2
        LDR R2, =RESULT

        LDR R3, [R0]
        LDR R4, [R1]
        ADD R5, R3, R4
        STR R5,[R2]

STOP B STOP

NUM1 DCD 0x11223344
NUM2 DCD 0x55667788

        AREA INFO, DATA, READWRITE
RESULT DCD 0x00000000
        END
```



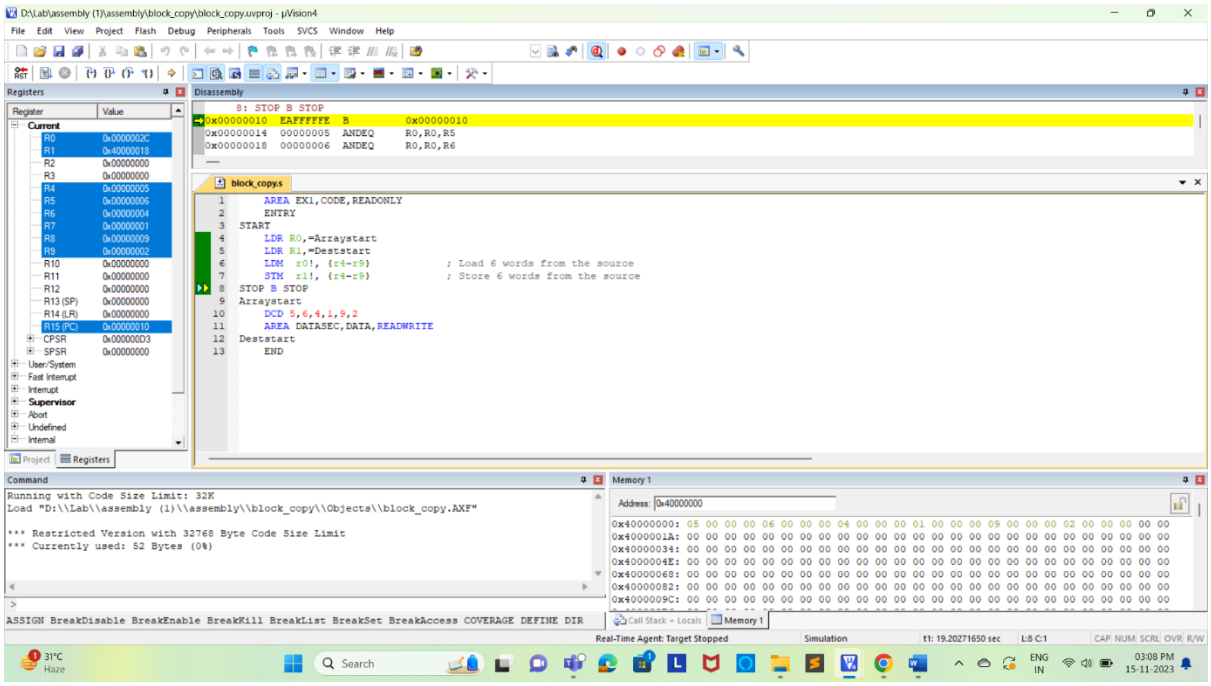
2. Check whether the string is palindrome or not

```
    AREA PALINDROME,CODE,READONLY
    ENTRY
START
   LDR R0,=ArrayBegin;
    LDR R1,=ArrayEnd;
    LDR R5,=Length
    LDR R6,[R5]
    MOV R8,R6, LSR #1
LOOP
    LDRB R2,[R0],#1
    LDRB R3,[R1,#-1]!
   ADD R4,R4,#1
    CMP R2,R3
    BNE STOP
    CMP R8,R4
    BHI LOOP
    CMP R4,R6, LSR #1
    BNE STOP
    MOV R7,#1
STOP B STOP
ArrayBegin
    DCB "malayalam"
ArrayEnd
   ALIGN
Length
   DCD 9
    END
```



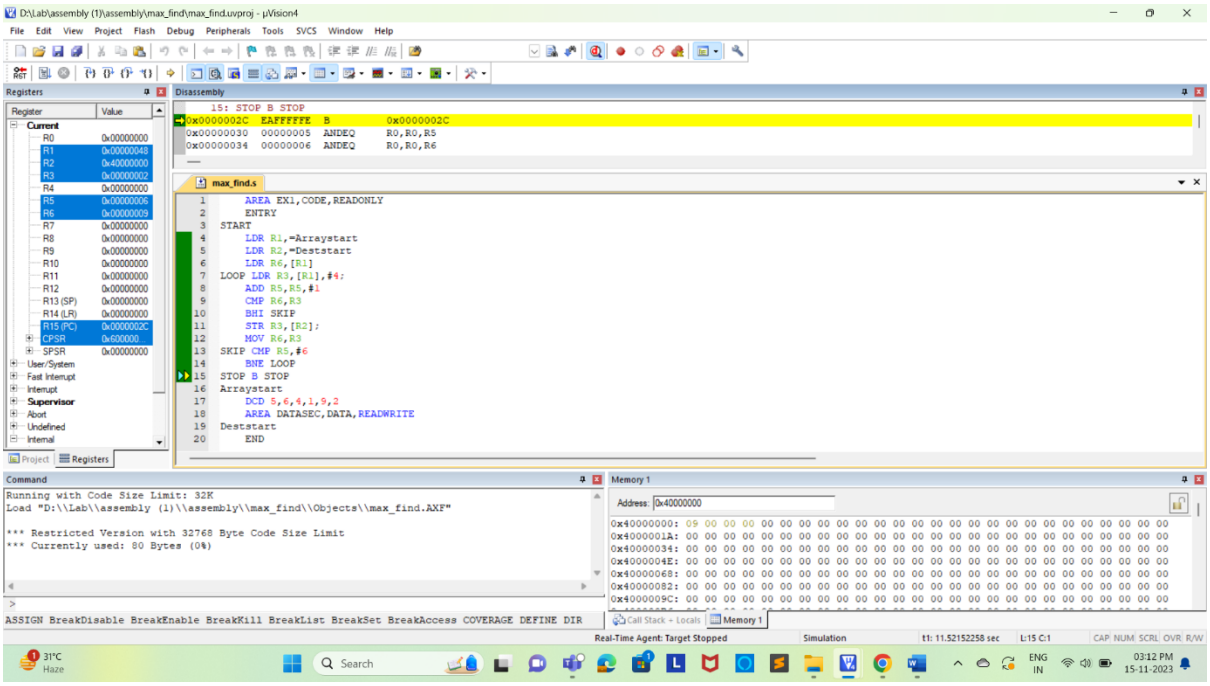
3. Block of data moving from one location to another location

```
        AREA EX1,CODE,READONLY
        ENTRY
START
        LDR R0,=Arraystart
        LDR R1,=Deststart
      LDM  r0!, {r4-r9}          ; Load 6 words from the source
      STM  r1!, {r4-r9}          ; Store 6 words from the source
STOP B STOP
Arraystart
      DCD 5,6,4,1,9,2
        AREA DATASEC,DATA,READWRITE
Deststart
      END
```



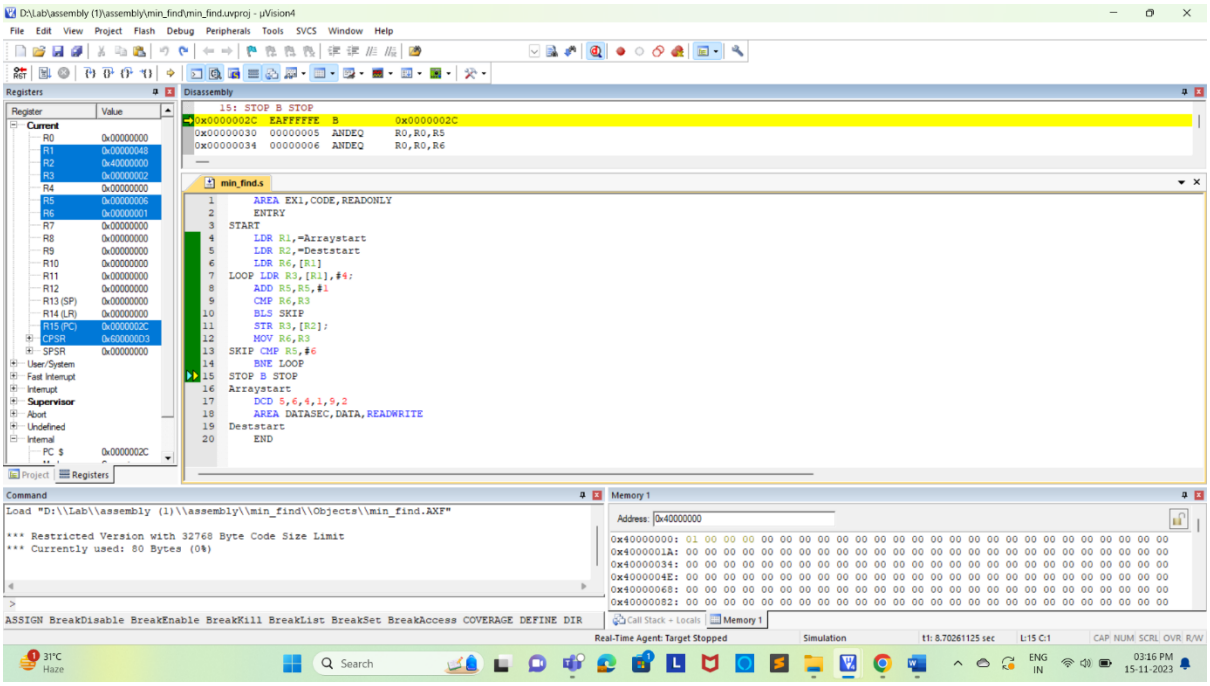
4. Find the maximum number in a given series

```
        AREA EX1,CODE,READONLY
        ENTRY
START
        LDR R1,=Arraystart
        LDR R2,=Deststart
        LDR R6,[R1]
LOOP LDR R3,[R1],#4;
        ADD R5,R5,#1
        CMP R6,R3
        BHI SKIP
        STR R3,[R2];
        MOV R6,R3
SKIP CMP R5,#6
        BNE LOOP
STOP B STOP
Arraystart
   DCD 5,6,4,1,9,2
        AREA DATASEC,DATA,READWRITE
Deststart
   END
```



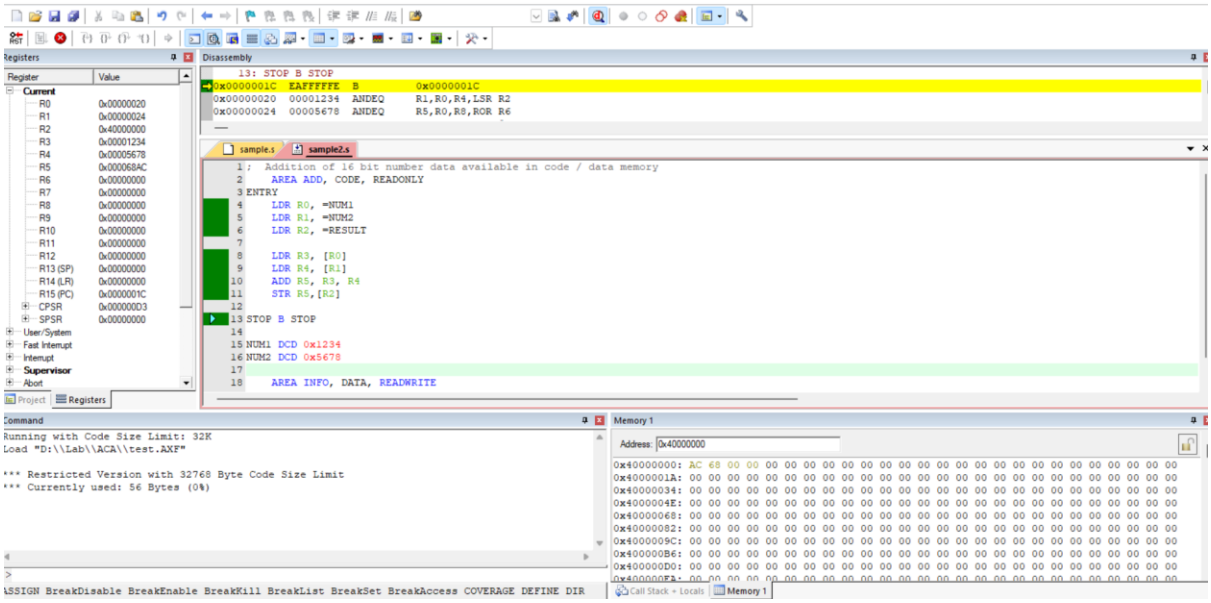
5. **Find the minimum number in a given series**

```
        AREA EX1,CODE,READONLY
        ENTRY
START
        LDR R1,=Arraystart
        LDR R2,=Deststart
        LDR R6,[R1]
LOOP LDR R3,[R1],#4;
        ADD R5,R5,#1
        CMP R6,R3
        BLS SKIP
        STR R3,[R2];
        MOV R6,R3
SKIP CMP R5,#6
        BNE LOOP
STOP B STOP
Arraystart
   DCD 5,6,4,1,9,2
        AREA DATASEC,DATA,READWRITE
Deststart
   END
```



6. Addition of 16 bit number data available in code / data memory

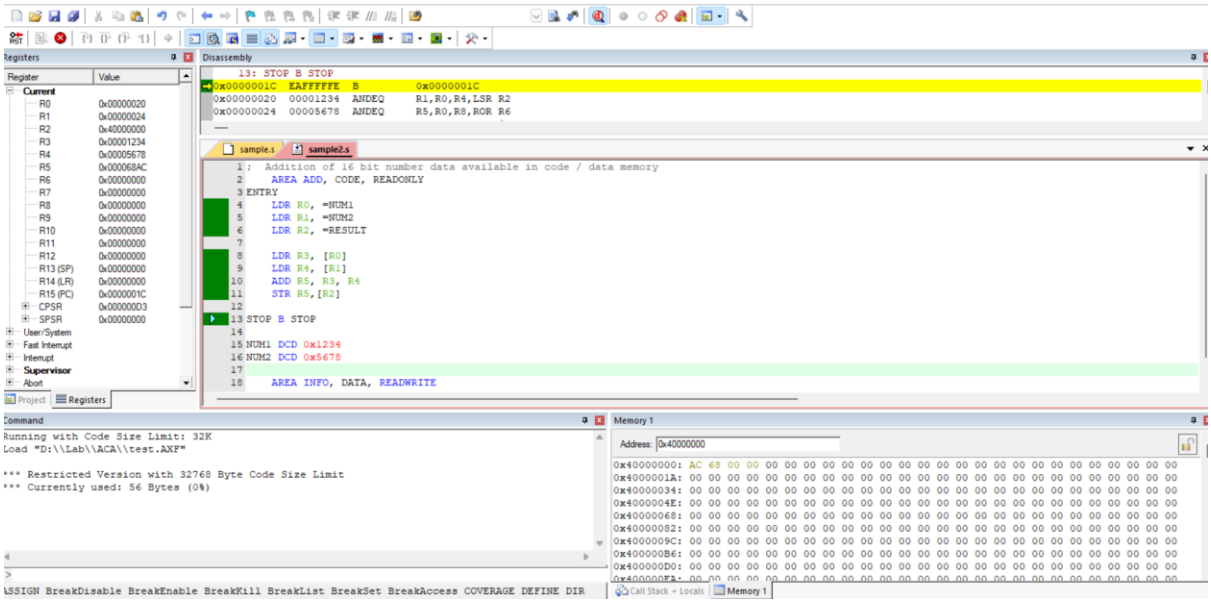
```
; Addition of 16 bit number data available in code / data memory
        AREA ADD, CODE, READONLY
ENTRY
        LDR R0, =NUM1
        LDR R1, =NUM2
        LDR R2, =RESULT

        LDR R3, [R0]
        LDR R4, [R1]
        ADD R5, R3, R4
        STR R5,[R2]

STOP B STOP

NUM1 DCD 0x1234
NUM2 DCD 0x5678

        AREA INFO, DATA, READWRITE
RESULT DCD 0x00000000
        END
```



7. Addition of 64 bit number data available in in code/ data memory

;Addition of 64 bit number data available in in code/ data memory

```
        AREA    add64, CODE, READONLY
ENTRY
MAIN
      LDR    R0, =Value1       ;pointer to first value
      LDR    R1, [R0]          ;load first part of value1
      LDR    R2, [R0, #4]      ; load lower part of value1
      LDR    R0, =Value2       ;pointer to second  value
      LDR    R3, [R0]          ;load upper part of value2
      LDR    R4, [R0, #4]      ; load lower part of value2

      ADDS    R6, R2, R4       ;add lower 4 bytes  and set carry flag
      ADC    R5, R1, R3        ;add upper 4 bytes  including carry

      LDR    R0, =Result       ;pointer to result

      STR    R5, [R0]          ;store upper part of result
      STR    R6, [R0, #4]      ;store lower part of result

      SWI    &11

Value1    DCD    &12A2E640, &F2100123
Value2    DCD    &001019BF, &40023F51

Result    DCD    0

      END
```


Registers
Register
Value
Current
R0 0x00000020
R1 0x00000024
R2 0x40000000
R3 0x00001234
R4 0x00005678
R5 0x000068AC
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x00000000
R14 (LR) 0x00000000
R15 (PC) 0x0000001C
CPSR 0x000000D3
SPSR 0x00000000
User/System
Fast Interrupt
Interrupt
Supervisor
Abort
Project
Registers
Disassembly
13: STOP B STOP
0x0000001C EAFFFFFE B 0x0000001C
0x00000020 00001234 ANDEQ R1,R0,R4,LSR R2
0x00000024 00005678 ANDEQ R5,R0,R8,ROR R6
sample.s
sample2.s
1 ; Addition of 16 bit number data available in code / data memory
2 AREA ADD, CODE, READONLY
3 ENTRY
4 LDR R0, =NUM1
5 LDR R1, =NUM2
6 LDR R2, =RESULT
7
8 LDR R3, [R0]
9 LDR R4, [R1]
10 ADD R5, R3, R4
11 STR R5,[R2]
12
13 STOP B STOP
14
15 NUM1 DCD 0x1234
16 NUM2 DCD 0x5678
17
18 AREA INFO, DATA, READWRITE
Command
Running with Code Size Limit: 32K
Load "D:\\Lab\\ACA\\test.AXF"
*** Restricted Version with 32768 Byte Code Size Limit
*** Currently used: 56 Bytes (0%)
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE DEFINE DIR
Memory 1
Address: 0x40000000
0x40000000: AC 68 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x4000001A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000034: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x4000004E: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000068: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000082: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x4000009C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000B6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Call Stack + Locals
Memory 1

8. ARMSTRONG

```
        AREA ARMSTRONG,CODE,READONLY
        ENTRY
START
        LDR R0,=Dest
        LDR R4,=Source
        LDR R1,[R4]
        MOV R2,#10
LOOP CMP R1,R2
        BLT RESULT
        SUB R1,R1,R2
        ADD R3,R3,#1
        B LOOP
RESULT
        STR R1,[R0],#4
        ADD R6,R6,#1
        MOV R1,R3
        MOV R3,#0
        CMP R1,#0
        BNE LOOP

        LDR R0,=Dest
        MOV R4,R6
NEXT MOV R1,R6    ;number of digits
   LDR R2,[R0]
        MOV R3,#1
MULTIPLY MUL R3,R2,R3
        SUB R1,R1,#1
        CMP R1,#0
        BNE MULTIPLY
        STR R3,[R0],#4 ;multiplication result storage
        SUB R4,R4,#1
        CMP R4,#0
        BNE NEXT

        LDR R0,=Dest ;add multiplied results
        MOV R2,#0
ADDITION    LDR R1,[R0],#4
        ADD R2,R2,R1
        SUB R6,R6,#1
        CMP R6,#0
        BNE ADDITION
STOP B STOP

Source
```

```
    DCD 153
        AREA DATAMEM,DATA,READWRITE
Dest

        END
```

9. Kelwin Palindrome

```
        AREA RESET,CODE,READONLY
        ENTRY

        LDR R0,=my_string;
        MOV R2,R0; Keep a copy of start address of string in R2
Start LDRB R1,[R0];
        CMP R1,CR;
        BEQ CheckPalin; // found the END of String
        ADD R0,R0,#0x01;
        B Start;

CheckPalin SUB R0,#1; Interate from last
        LDRB R1,[R2]; load first char
        LDRB R3,[R0]; load last char
        ADD R2,R2,#0x01;
        CMP R1,R3;
        BNE NotPalindrome;
        CMP R0,R2;verilog
        BLS Palindrome;
        B CheckPalin;
NotPalindrome MOV R0,#0;
        B STOP;
Palindrome MOV R0,#1;
STOP B STOP;
```

```
my_string
        DCB "MADAM",CR
        ALIGN
CR EQU 0x0d
        END
```

10. Display hello world on console

```
        AREA EX1,CODE,READONLY
        ENTRY

START LDR R0,=SOURCE
        LDR R1,=DEST

LOOP LDRB R2,[R0],#1
        STRB R2,[R1],#1
        CMP R2,#0x0000000A
        BNE LOOP

STOP B STOP
SOURCE
        DCB "HELLO WORLD", 0x0A

        AREA EX2,DATA,READWRITE
DEST
        END
```

Block code copy from one table to another table

```
        AREA BLOCKCOPY,CODE,READONLY
SWI_WriteC   EQU &0;
SWI_Exit     EQU &11;
        ENTRY

        ADR R1,TABLE1
        ADR R2,TABLE2
        ADR R3,T1END

LOOP1 LDR R0,[R1],#4
        STR R0,[R2],#4
        CMP R1,R3
        BLT LOOP1

        ADR R1,TABLE2
LOOP2 LDRB R0,[R1],#1
        CMP R0,#0
        SWINE SWI_WriteC
        BNE LOOP2
        SWI SWI_Exit

TABLE1 =            "This is the world" ,&a ,&d, 0
T1END
        ALIGN
TABLE2 =            "This is the wrong strong", 0
        END
```