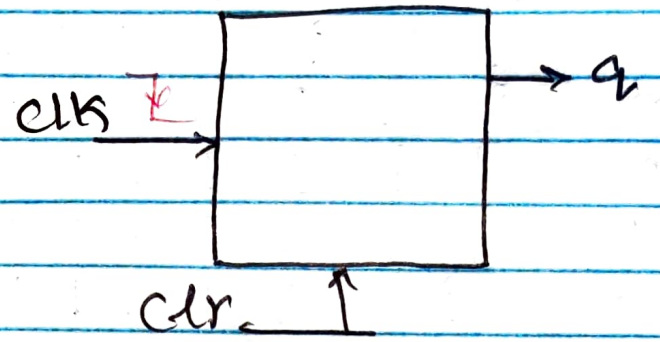


A simple sequential circuit model

A T-ff model:

```
primitive Tff (q, clk, clr);
    input clk, clr;
    output reg q;
    table
    // clk  clr   q   qnew
       ?    1   : ? : 0;
       ?   (10) : ? : -;
      (10)  0   : 1 : 0;
      (10)  0   : 0 : 1;
      (0?)  0   : ? : -;
    endtable
endprimitive
```

A ripple counter defined above can be used to design ripple counter by cascading

ex:

```
primitive ripplecounter6 (count, clk, clr);
  input clk, clr;
  output [5:0] count;
  TFF F0 (count[0], clk, clr);
  TFF F1 (count[1], count[0], clr);
  TFF F2 (count[2], count[1], clr);
```

```
TFF F3(count[3], count[2], clr);
TFF F4(count[4], count[3], clr);
TFF F5(count[5], count[4], clr);
endprimitive
```

This can also be a module with primitive instantiation.

ex: Negative-edge sensitive JK flipflop.

```
primitive JKFF(q, j, k, clk, clr);
    input j, k, clk, clr;
    output reg q;
    table
    //j  k  clk  clr  q  qnew
      ?  ?  ?    1   :  ?  :  0;  //clear
      ?  ?  ?   (10) :  ?  :  -;  // no change
      0  0  (10) 0   :  ?  :  -;  // no change
      0  1  (10) 0   :  ?  :  0;  // reset condition
      1  0  (10) 0   :  ?  :  1;  // set condition
      1  1  (10) 0   :  0  :  1;  // toggle condition
      1  1  (10) 0   :  1  :  0;  // toggle condition
      ?  ?  (01) 0   :  ?  :  -;  // no change
    endtable
endprimitive
```

ex: positive edge sensitive SR flipflop.

```
primitive SRFF(q, s, r, clk, clr);
    input s, r, clk, clr;
    output reg q;
    table
    // s  r  clk  clr  q  qnew
```

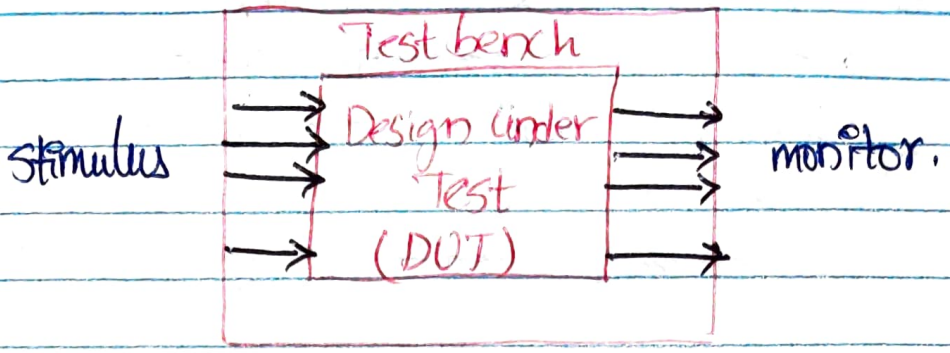
```
?   ?   ?   1   :   ?   0;
?   ?   ?   (10)  :   ?   -;
0   0   (01)  0   :   ?   -;
0   1   (01)  0   :   ?   0;
1   0   (01)  0   :   ?   1;
1   1   (01)  0   :   ?   x;
?   ?   (10)  0   :   ?   -;
endtable
endprimitive
```

Rules for primitives:

? – Not to be specified in outputs
– – No chage should be only in output
(01) – shortcut to rising edge (r)
(10) – shortcut to falling edge (f)
(*) – indicates any value change in signal.

Verilog Test Bench:
Procedural blocks which runs only once.
Used only for simulation
Generate clk, reset, other functional iputs to DUT
print values using monitor
dump values in file.

Both always & initial blocks are used but always is used only for repititive signals like clk generation.



Simple example

```
module example (A,B,C,D,E,F,Y);
    input A,B,C,D,E,F;
    output Y;
    wire t1,t2,t3,Y;
    nand #1 G1(t1,A,B);
    and  #2 G2(t2,C,~B,D);
    nor  #1 G3(t3,E,F);
    nand #1 G4(Y,t1,t2,t3);
endmodule
module testbench;
    reg A,B,C,D,E,F; wire Y;
    example DUT(A,B,C,D,E,F,Y);
    initial
        begin
        $monitor($time, "A=%b, B=%b, C=%b, D=%b, E=%b,
                         F=%b, Y=%b");
        #5 A=1; B=0; C=0; D=1; E=0; F=0;
        #5 A=0; B=0; C=1; D=1; E=0; F=0;
        #5 C=0; A=1;
        #5 F=1;
    #5 $finish;
    end endmodule
```

For synchronous sequential logic
we need clk generation logic
different ways to specify clks.
Testbench can include various simulation directives
$display, $monitor, $dumpfile, $dumpvars $finish etc

Simulator directives:

$display ("format", expr1, expr2, ...);
eg: $display ("%b", a);

$monitor ("format", expr1, expr2, ...);
eg: $monitor ($time, "%b, %d", a, b);
monitor is event driven display statement ; here expr can be only variables in monitor but can be any expression in display.

$finish is to end simulation

$dumpfile (<file_name>);
specify file in arguments to dump variables values.
has extension .vcd (value change dump) and
contains all changes in the file.

$dumpoff;
will turn off dumping into file

$dumpon;
start dumping values again.

$dumpvars (level, list of variables or modules);

ex: $dumpvars(0, module1, module2);

all values from module1 will be dumped

module2 variables will be dumped

ex: $dumpvars(1, a, b, c, d);

with level = 1 we specify variables to dump;

$dumpall;

Dump all variable values into the file irrespective of changes in their values.

$dumplimit (file size);

Specify max size of dumpfile size;

ex: 2-bit equality checker.

```
`timescale 1ns/100ps.
module comparator (x, y, z);
  input [1:0] x, y;
  output z;
  assign z = (x[0] & y[0] & x[1] & y[1]) |
             (~x[0] & ~y[0] & x[1] & y[1]) |
             (~x[0] & ~y[0] & ~x[1] & ~y[1]) |
             (x[0] & y[0] & ~x[1] & ~y[1]);
endmodule.
```

Above module compares if two bit variables are equal.

testbench:

```
`timescale 1ns/100ps;
module testbench;
  reg [1:0] x, y;
  wire z;
  comparator c2 (.x(x), .y(y), .z(z));
  initial begin
    $dumpfile("comp.vcd");
    $dumpvars(0, testbench);
    x=2'b01 ; y=2'b00;
    #10 x=2'b10; y=2'b10;
    #10 x=2'b01; y=2'b11;
  end
  initial
    begin
    $monitor("t=%d, x=%2b, y=%2b, z=%b",
                $time, x, y, z);
    end
endmodule
```

examples of testbenches for various modules.

```
module fa (s, co, a, b, c);
  input a, b, c;
  output s, co;
  assign s= a^b^c;
  assign co= (a&b) | (c&b) | (c&a);
endmodule
```

```
module testbench;
    reg a,b,c;
    wire sum, cout;
    fulladder f1 (.sum(s), .cout(c), .a(a), .b(b), .c(c));
    initial
        begin
        $monitor ($time, "a=%b, b=%b, c=%b,
                  sum=%b, cout=%b", a,b,c,sum,cout);
        #5 a=0; b=0; c=1;
        #5 b=1;
        #5 a=1;
        #5 a=0; b=0; c=0;
        #5 $finish;
      end
  endmodule
```

new version with vcd files.

```
module testbench;
    reg a,b,c; wire sum, cout;
    integer i;
    fulladder fa (sum, cout, a,b,c);
    initial begin
        $dumpfile ("full adder.vcd");
        $dumpvars(0, testbench);
      for (i=0; i<8; i=i+1);
        begin
        {a,b,c} = i; #5;
      $display ("%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b,
              $time, a,b,c, sum, cout); end
      #5 $finish;
```

```
    end
endmodule
```

Self-checking test benches:

```
module fulladder_test;
  reg a,b,c;
  wire sum, carry;
  integer correct;
  fulladder fa (a,b,c, sum, carry);
  initial
    begin
    correct=1
    #5 a=1; b=1; c=0; #5;
      if ((s!=0) || (carry!=1))
        correct=0;
    #5 a=1; b=1; c=1; #5;
      if((s!=1) ||(carry!=1))
        correct=0;
    #5 a=0; b=1; c=0; #5;
      if((s!=1) || (carry!=0))
        correct=0;
    #5 $display("%d", correct);
    end
endmodule
```

$random is used to randomly assign any values to testbench variables

used inside procedural blocks only

<seed> will be useful to generate same stimulus to debug

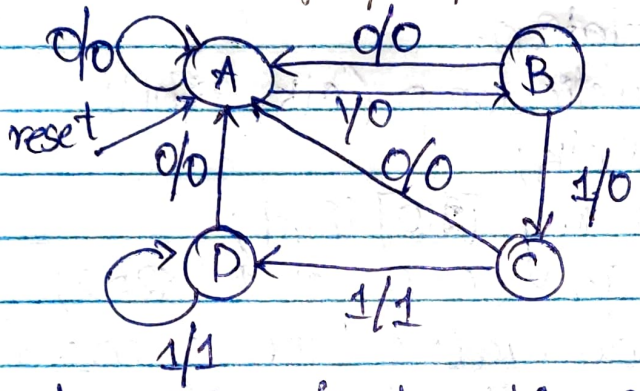
Modelling Finite state machines.

FSM → State table or Algorithmic State Machine Chart (ASM)

ex:

Detect 3 or more consecutive One's.

Notation: input/output



state diagram.

PS:- Present State
NS:- Next state.

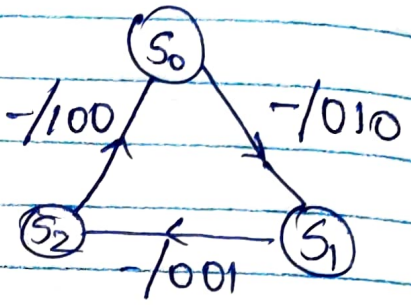
Reset	PS	input	NS	output
1	-	-	A	0
0	A	0	A	0
0	A	1	B	0
0	B	0	A	0
0	B	1	C	0
0	C	0	A	0
0	C	1	D	1
0	D	0	A	0
0	D	1	D	1

Mealy Machine: Output depends on inputs & present state

Moore Machine: Output depends on present state only.

ex: 3-state lamps RED, GREEN, YELLOW.

Assume $s_0 \rightarrow$ red
$s_1 \rightarrow$ Green
$s_2 \rightarrow$ yellow.



Three states so 2-bit vector for state representation.

```
module cyclic_lamp (clk, light);
    input clk;
    output reg [2:0] light;
    parameter S0=0, S1=1, S2=2;
    parameter red=3'b100, green=3'b010, yellow=3'b001;
     reg [1:0] State;
    always @(posedge clk)
        case (state)
            S0: begin
                light <=green; State <=S1;
            end
            S1: begin
                light <= yellow; state <=S2;
                end
            S2: begin
                light<=red ; state <=S0;
                end
          default: begin
                    light<=Red; state <=S0;
                   end
        endcase
endmodule.
```

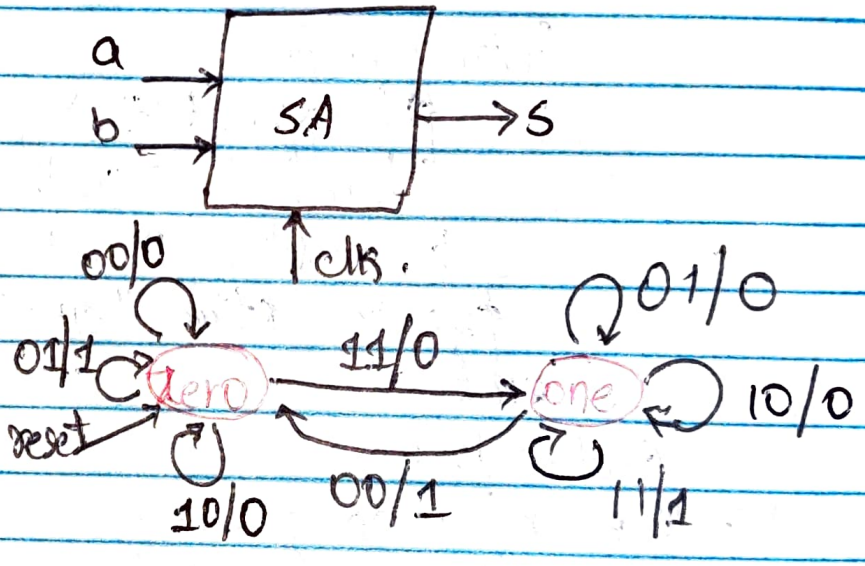
```
module test_cyclic_lamp;
    reg clk;
    wire [2:0] light;
    cyclic_lamp L1(.clk(clk), .light(light));
    always #5 clk=~clk;
    initial
      begin
        clk = 1'b0
        #100 $finish;
      end
    initial begin
        $dumpfile(0, test_cyclic_lamp);
        $dumpfile ("cyclic lights.vcd");
        $dumpvars (0, test_cyclic_lamp);
      $monitor ($time, " clk=%b, RGY=%b", clk, light);
    end
  endmodule
```

Modified and compact design

```
module cyclic_lamp (clk, light);
  input clk; output [2:0] light;
  parameter S0=0, S1=1, S2=2;
  parameter red = 3'b100, green=3'b010, yellow=3'b001;
  reg [1:0] state;
  always @(posedge clk)
      case (state)
          S0: state <=S1;
          S1: state <=S2;
          S2: state <=S0;
        default: state <=S0;
```

```
    endcase
always @(state)
case(state) S0: light = red;
            S1: light = green;
            S2: light = yellow;
        default: light = red;
    endcase
endmodule
```

Serial adder:



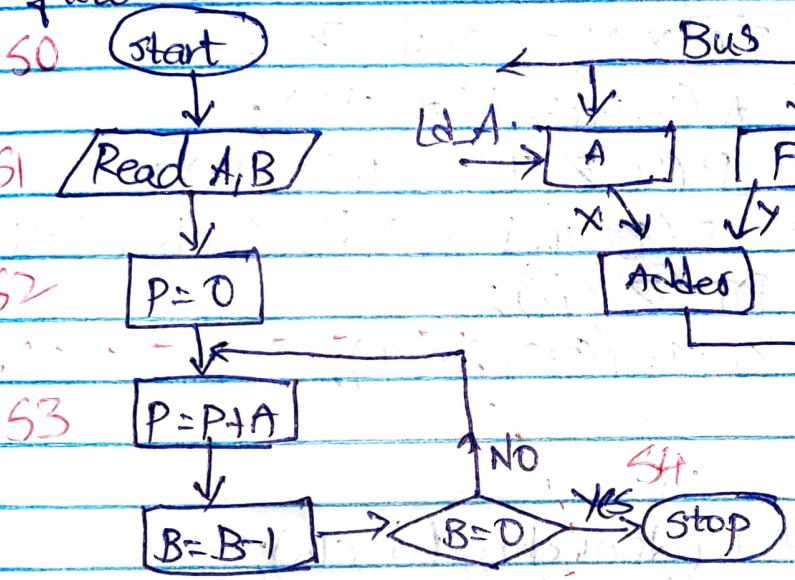
State transition diagrams are important to model any sequential circuit!

Datapath and Controller Design:

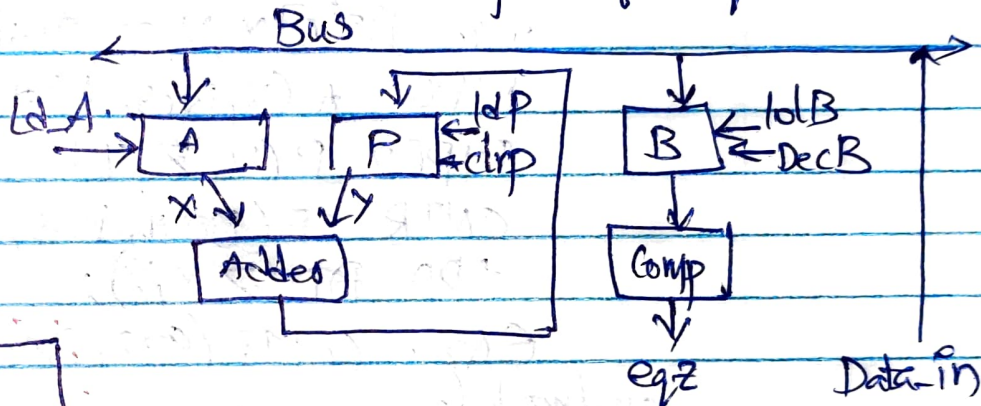
Datapath: functional blocks containing, registers, counters, adders bus, multiplexers where all computations or carried out.
Control path: Implements FSM and provides controller signals to the datapath in proper sequence. It takes inputs from datapath.

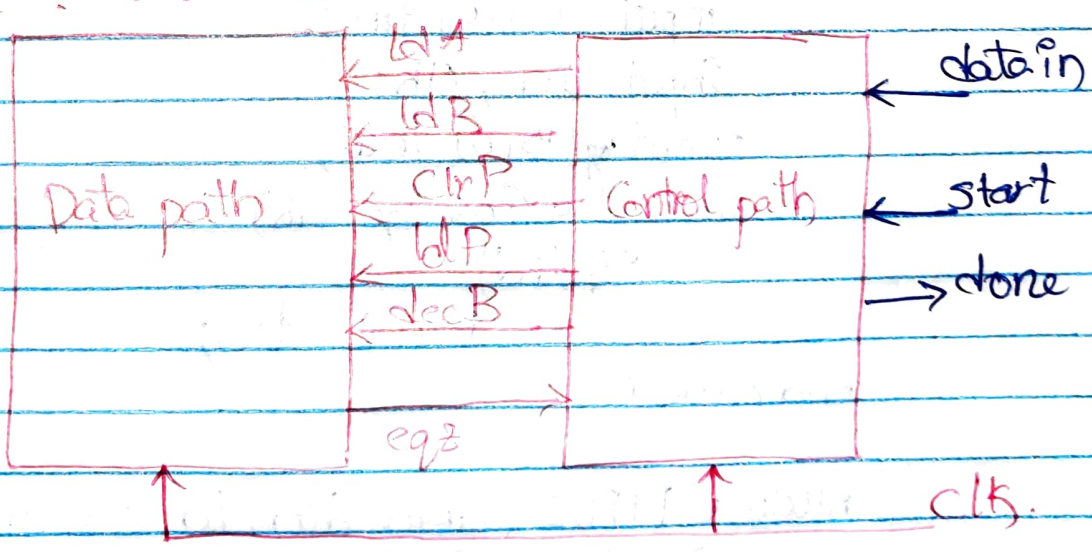
Ex: multiplication by repeated addition.

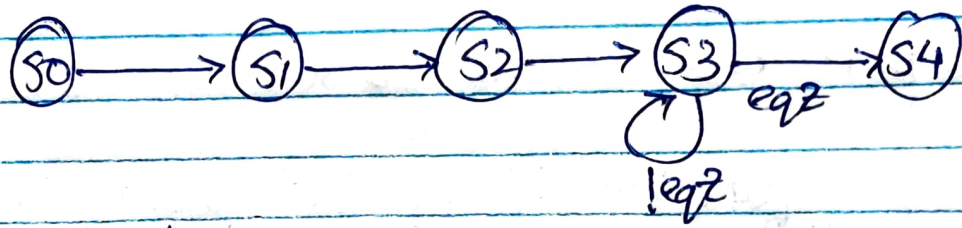
flow chart



Datapath from flowchart







Verilog code:

```
module multipathdatapath (eqz, ldA, ldB, ldP, clrP, decB,
        data in, clk);
    input ldA, ldB, ldP, clrP, decB, clk;
    input [15:0]data_in;
    output eqz;
    wire [15:0] X, Y, Z, Bout, Bus;
    PIPO1 A (X, Bus, ldA, clk);
    PIPO2 P (Y, Z, ldP, clrP, clk);
    CNTR  B(Bout, Bus, ldB, decB, clk);
    ADD AD(Z, X, Y);
    EQZ. COMP (eqz, Bout);
endmodule
// individual module behavioral description.
module PIPO1 (dout, din, ld, clk);
    input [15:0] din;
    input ld, clk;
    output reg [15:0] dout;
    always @(posedge clk)
        if (ld)
            dout <= din;
endmodule

module PIPO2 (dout, din, ld, clr, clk);
    input [15:0] din;
    input clk, clr, ld;
```

```
    output [15:0] dout;
    always @(posedge clk)
        if (clr)
            dout <= 16'b0;
        else if (ld)
            dout <= din;
endmodule

module CNTR(dout, din, clk, ld, dec);
    input [15:0] din;
    input clk, ld, dec;
    output [15:0] dout;
    always @ (posedge clk)
        if (ld)
            dout <= din ;
        else if (dec)
            dout <= dout-1;
end module

module ADD (z,x,y);
    input [15:0] x, y;
    output reg [15:0] z;
    always @ (*)
        z = x+y;
endmodule

module EQZ (out, data);
    input [15:0] data;
    output out;
    assign out = (data == 0);
endmodule
```

//Next step is to code FSM for controller.

```
module controller(ldA, ldB, ldP, clrP, decB, done, clk, eq2, start);
  input clk, eq2, start;
  output reg ldA, ldB, ldP, clrP, decB, done;
  reg [2:0] state;
  parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010,
            S3 = 3'b011 S4 = 3'b100 ;
  always@(posedge clk)
    begin
      case (State)
        S0: if(start)
              state <= S1;
        S1: state <= S2;
        S2: state <= S3;
        S3: #2 if (eq2)
              state <= S4;
        S4: state <= S4;
        default: state <= S0;
      endcase
      end
endmodule
always@(state)
  begin
    case (state)
    S0: begin #1 ldA=0; ldB=0; ldP=0;
          clrP=0 ; decB=0; end
    S1: begin #1 ldA=1; end
    S2: begin #1 ldA=0; ldB=1 ; clrP=1; end
    S3: begin #1 ldB=0; ldP=1; clrP=0;
          decB=1; end
```

```
        S4 : begin #1 done =1 ; ldB=0; ldP=0; decB=0; end
    default: begin #1 ldA=0; ldB=0; ldP=0; clrP=0;
                    decB=0; end.
    endcase
  end
endmodule.
//test bench;
module multest;
    reg [15:0] data_in;
    reg clk, start;
    wire done;
  multipathdatapath DP(eqz, ldA, ldB, ldP, clrP, decB, data_in, clk);
  Controller  con(ldA, ldB, ldP, clrP, decB, done, clk, eqz, start);
  initial
      begin                                  $dumpvars(0, multest);
        clk = 1'b0;                      $monitor($time, "%d %b",
        #3 start = 1'b1;                          DP.Y, done);
        #1500 $finish;                   end
      end                                endmodule

  always #5 clk= ~clk;
 initial begin
      #17 data_in=17;
      #10 data_in=15;
   end
 initial
    begin
    $dumpfile("Multiple.vcd");
```

Example 2: GCD computation with repeated subtraction.
GCD is largest integer which divides both given numbers.

ex: 26, 65 ∴ Answer is 13.

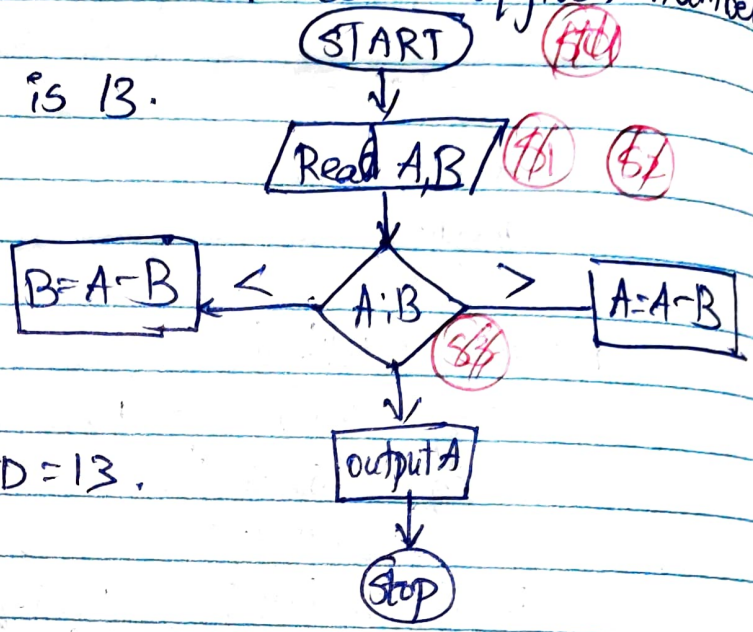
A=26 B=65

B=39

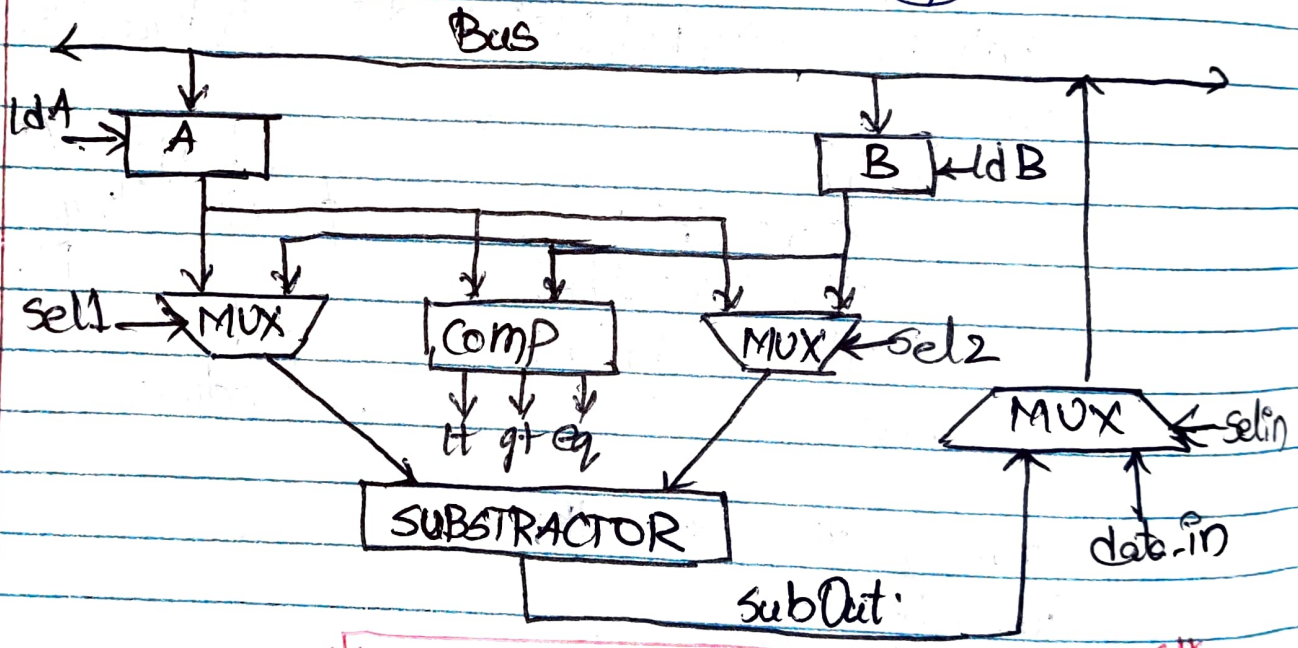
B=13

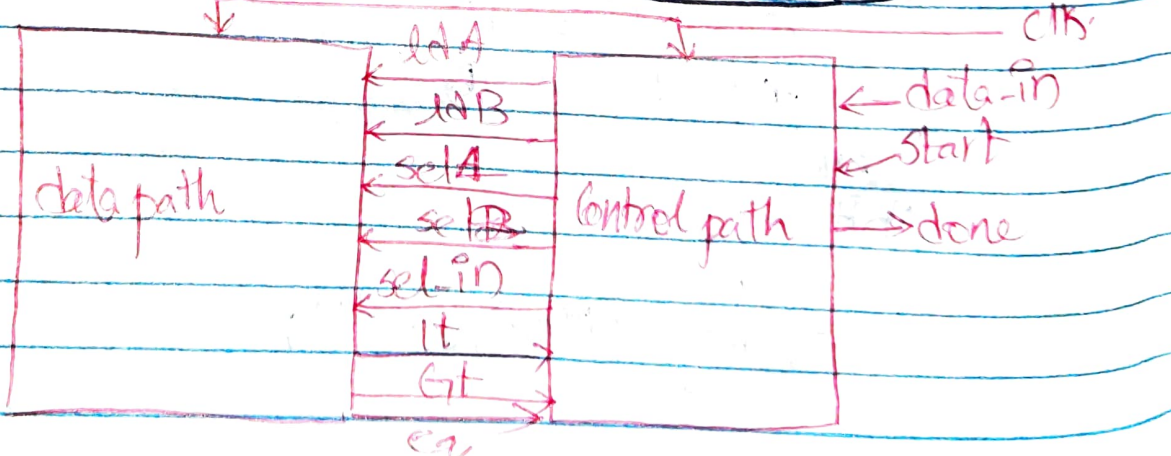
A=13

∴ A=B=13 ⇒ GCD=13.

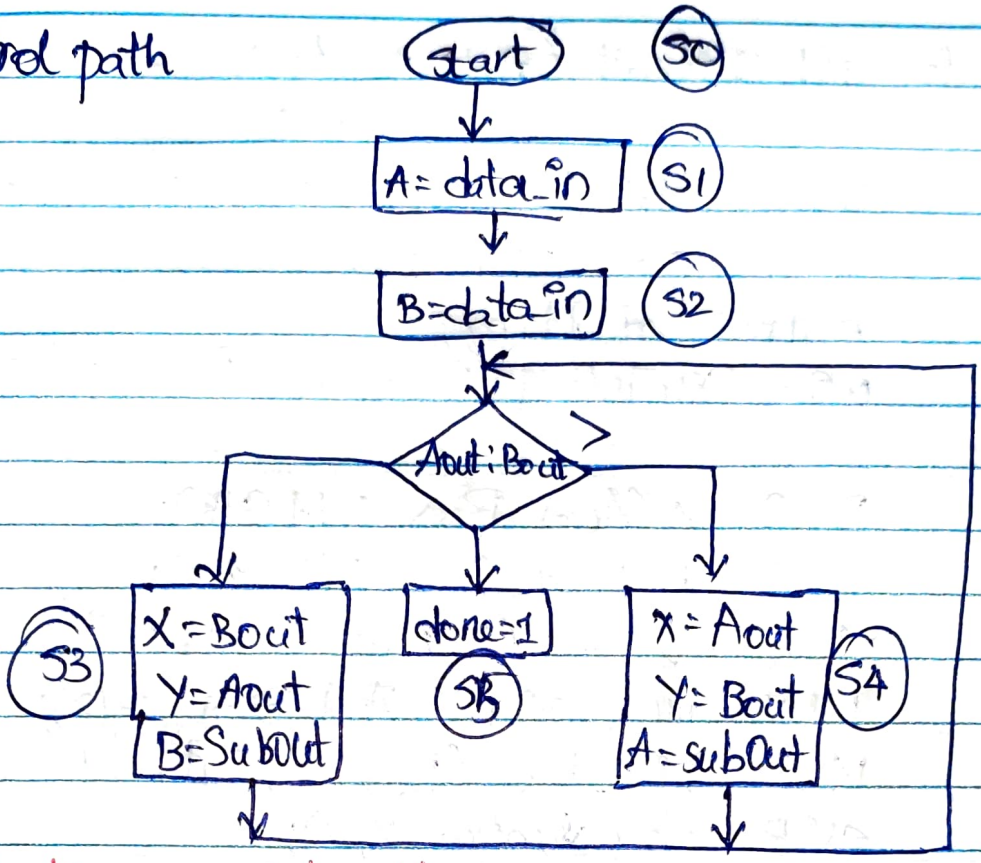


Data Path.

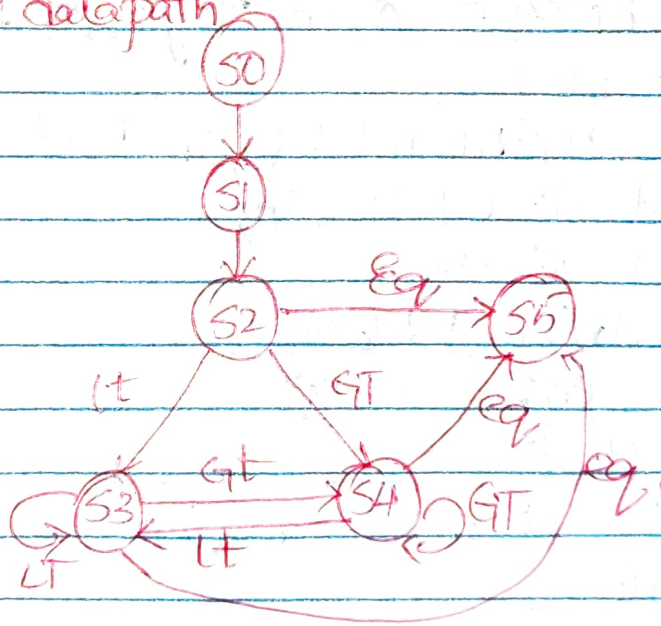




Control path



State diagram: datapath



```
module gcd_datapath(gt, lt, eq, ldA, ldB, sel1, sel2, sel_in,
                    data_in, clk);
    input ldA, ldB, sel1, sel2, sel_in, clk;
    input [15:0] data_in;
    output gt, lt, eq;
    wire [15:0] Aout, Bout, X, Y, Bus, subOut;

    PIPO A(Aout, Bus, ldA, clk);
    PIPO B(Bout, Bus, ldB, clk);
    MUX mux_in1(X, Aout, Bout, sel1);
    MUX mux_in2(Y, Aout, Bout, sel2);
    MUX Mux_load(Bus, subOut, data_in, sel_in);
    SUB sb(subOut, X, Y);
    COMPARE comp(lt, gt, eq, Aout, Bout);
endmodule
module PIPO(data_out, data_in, load, clk);
  input [15:0] data_in;
  input ld, clk;
  output reg [15:0] data_out;
  always @(posedge clk)
    if(load)
        data_out <= data_in;
endmodule

module COMPARE(lt, gt, eq, data1, data2);
    input [15:0] data1, data2;
    output lt, gt, eq;
```

```
        assign lt = data1 < data2;
        assign gt = data1 > data2;
        assign eq = data1 == data2;
    endmodule

module MUX (out, in0, in1, sel);
    input [15:0] in0, in1;
    input sel;
    output [15:0] out;
    assign out = sel ? in1 : in2;
endmodule

module SOB (out, in1, in2);
    input [15:0] in1, in2;
    output reg [15:0] out;
    always @ (*)
        out = in1 - in2;
endmodule
```

Control path :

```
module controller (ldA, ldB, sel1, sel2, sel_in, done, clk,
                   gt, lt, eq, start);
    input clk, gt, lt, eq, start;
    output reg ldA, ldB, sel1, sel2, sel_in, done;
    reg [2:0] state;
    parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011,
              S4 = 3'b100, S5 = 3'b101;
    always @ (posedge clk)
        begin
```

```
Case (State)
   S0: if (start)
          State<=S1;
   S1: State <=S2;
   S2: #2 if (eq) State<=S5;
       else if (lt) state <=S3;
       else if (gt) state <=S4;
   S3: #2 if (eq) State<=S5;
       elseif (lt) State <=S3;
       elseif (gt) State <=S4;
   S4: #2 if (eq) State<=S5;
       elseif (lt) State <=S3;
       elseif (gt) State <=S4;
   S5: State <=S5;
 default: State <=S0;
endcase
end
always @ (state)
   begin
      Case (state)
         S0: begin
               sel_in=1; ldA=1;
               ldB=0; done=0;
             end
         S1: begin
               sel_in=1; ldA=0;
               ldB=1; done=0;
             end
```

```
S2: if(eq) done=1;
    else if (lt) begin
                   sel1=1; sel2=0; sel_in=0;
                   #1 ldA=0; ldB=1;
                 end
    else if (gt) begin
                   sel1=0; sel2=1; sel_in=0;
                   #1 ldA=1; ldB=0;
                 end
S3: if(eq) done=1;
    else if (lt) begin
                   sel1=1; sel2=0; sel_in=0;
                   #1 ldA=0; ldB=1;
                 end
    else if (gt) begin
                   sel1=0; sel2=1; sel_in=0;
                   #1 ldA=1; ldB=0;
                 end
S4: if (eq) done=1;
    else if (lt) begin
                   sel1=1; sel2=0; sel_in=0;
                   #1 ldA=0; ldB=1;
                   end
    else if (gt) begin
                   sel1=0; sel2=1; sel_in=0;
                   #1 ldA=1; ldB=0;
                 end
S5: begin
    done=1; sel1=0; sel2=0; sel_in=0; ldB=0
```

```
            end
        default: begin
                    ldA=0; ldB=0;
                 end
    endcase
  end
endmodule
// testbench for gcd calculator
module gcdtest;
  reg clk, start;
  reg [15:0] data_in;
  wire done;
  reg [15:0] A, B;
  GCD_datapath gcddata(gt, lt, eq, ldA, ldB, sel1, sel2,
                       selin, data_in, clk);
  Controller Controlpath(ldA, ldB, sel1, sel2, selin, done,
                         clk, lt, gt, eq, start, );
  initial begin
      clk=1'b0;
    #3 start=1'b1;
    #1000 $finish;
  end
  always #5 clk=~clk;
  initial begin
    #12 data_in=143;
    #10 data_in=78;
  end
  initial begin
    $monitor($time, "%d : %b", gcddata.Aout, done);
    $dumpfile("gcd.vcd");
    $dumpvars(0, gcdtest);
  end
endmodule
```