BECE407E ASIC DESIGN

DataPath & ControlPath Design Methodologies

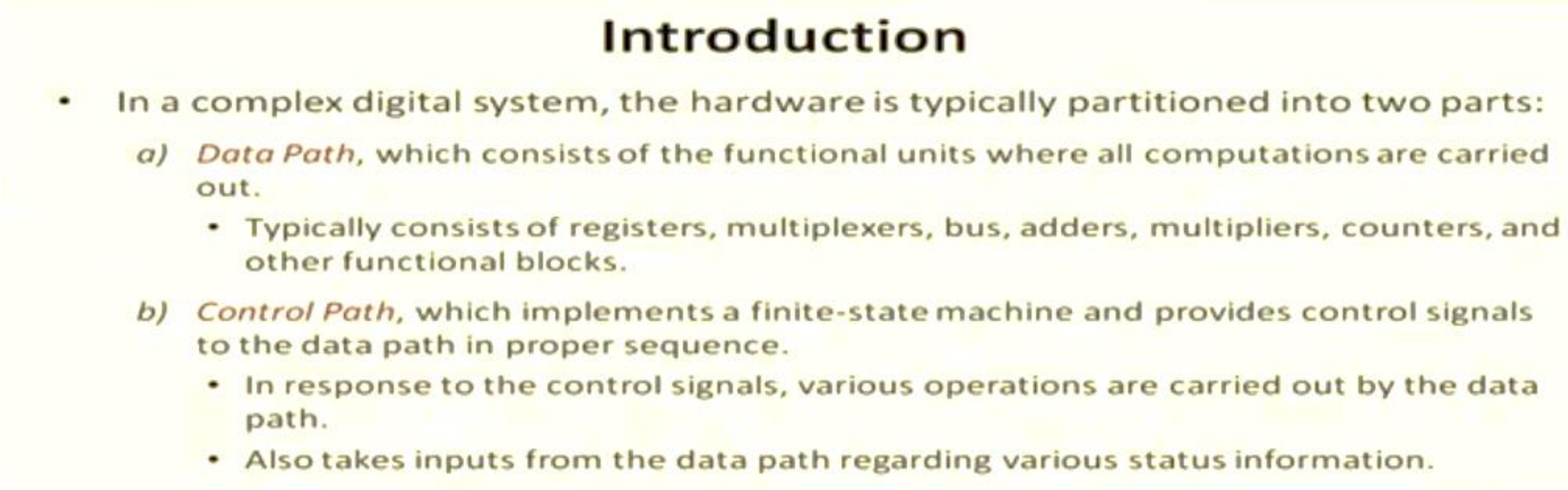
Dr. PRITAM BHATTACHARJEE

Assistant Professor (Senior Grade 2), ***Senior Member IEEE***

School of Electronics Engineering (SENSE)

Vellore Institute of Technology – Chennai

pritam.bhattacharjee@vit.ac.in, +91 8132863424



Introduction

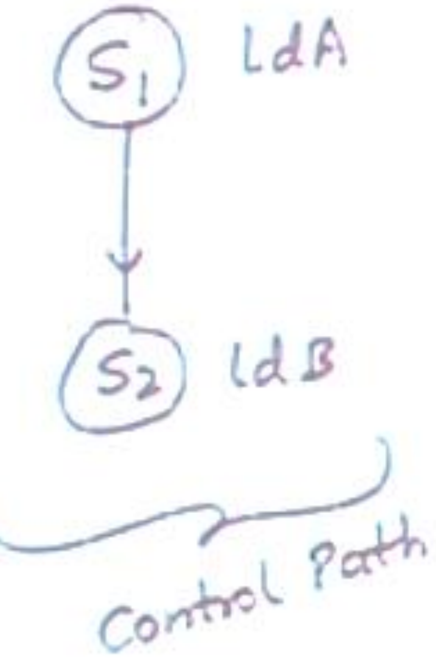
- In a complex digital system, the hardware is typically partitioned into two parts:
  - a) *Data Path*, which consists of the functional units where all computations are carried out.
    - Typically consists of registers, multiplexers, bus, adders, multipliers, counters, and other functional blocks.
  - b) *Control Path*, which implements a finite-state machine and provides control signals to the data path in proper sequence.
    - In response to the control signals, various operations are carried out by the data path.
    - Also takes inputs from the data path regarding various status information.

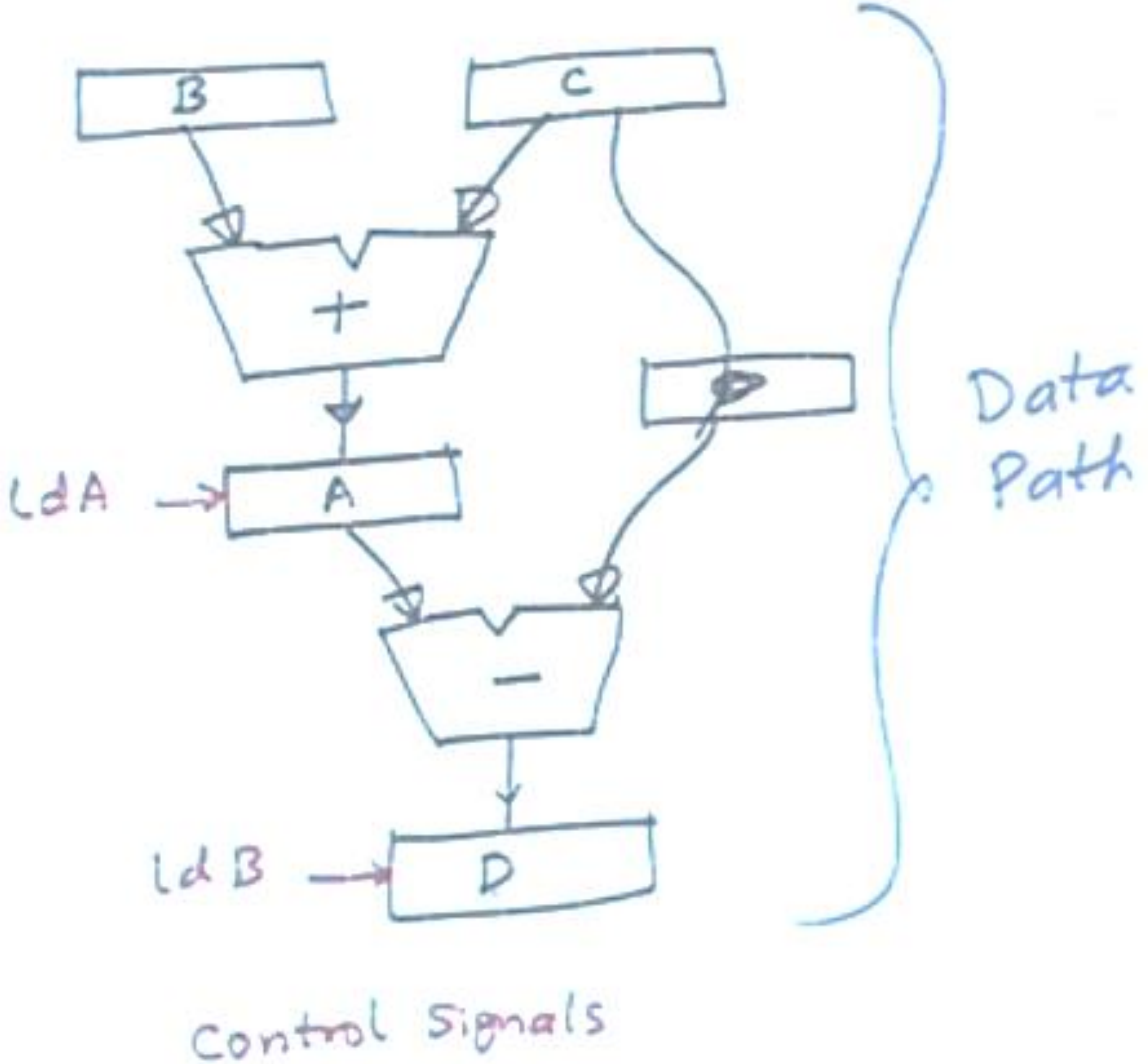
```
reg [15:0] A, B, C, D;
```

$$A = B + C$$

$$D = A - C$$

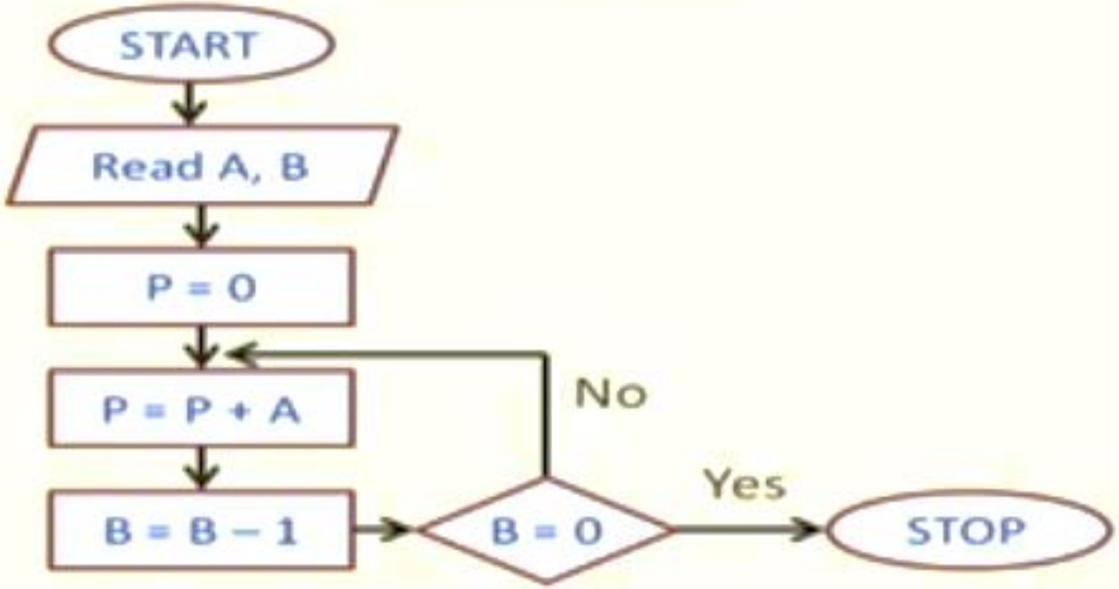
⟹





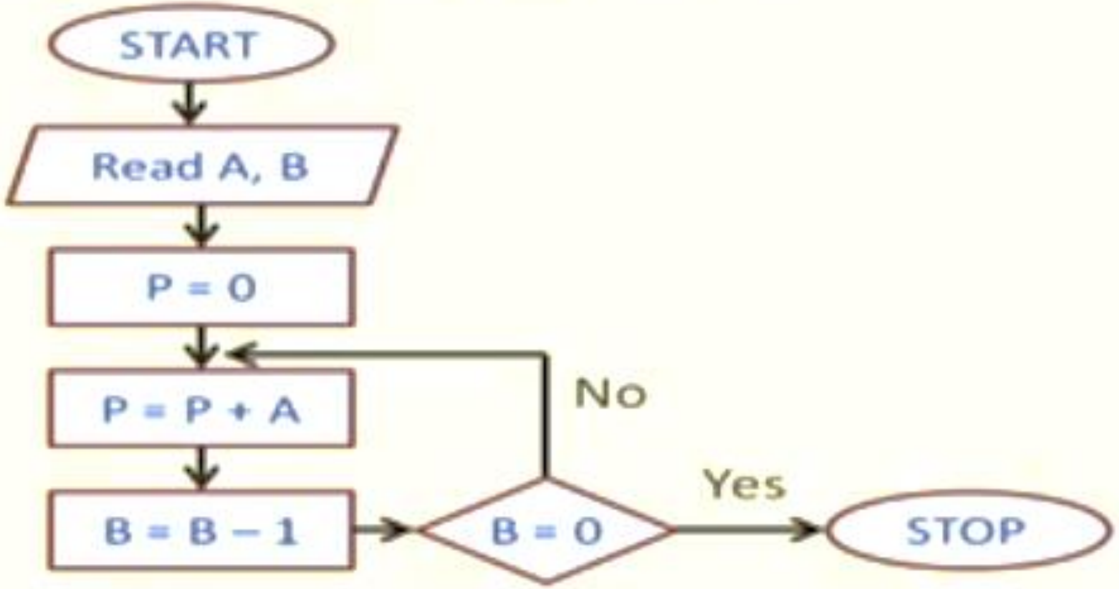
Example 1: Multiplication by Repeated Addition

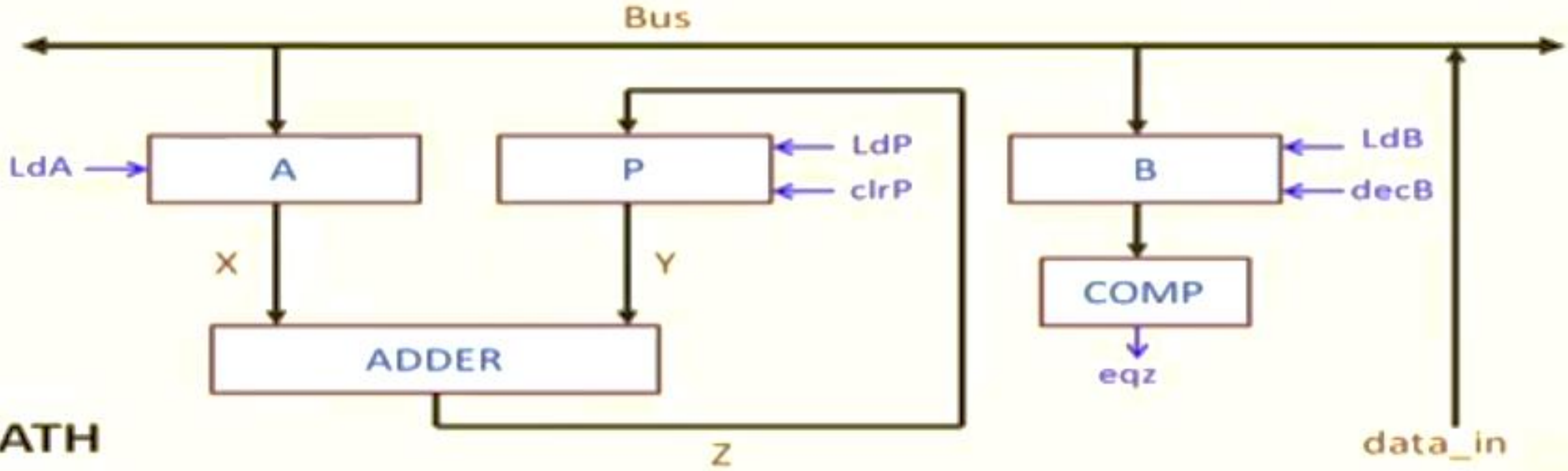
- We consider a simple algorithm using repeated addition.
  - Assume B is non-zero.



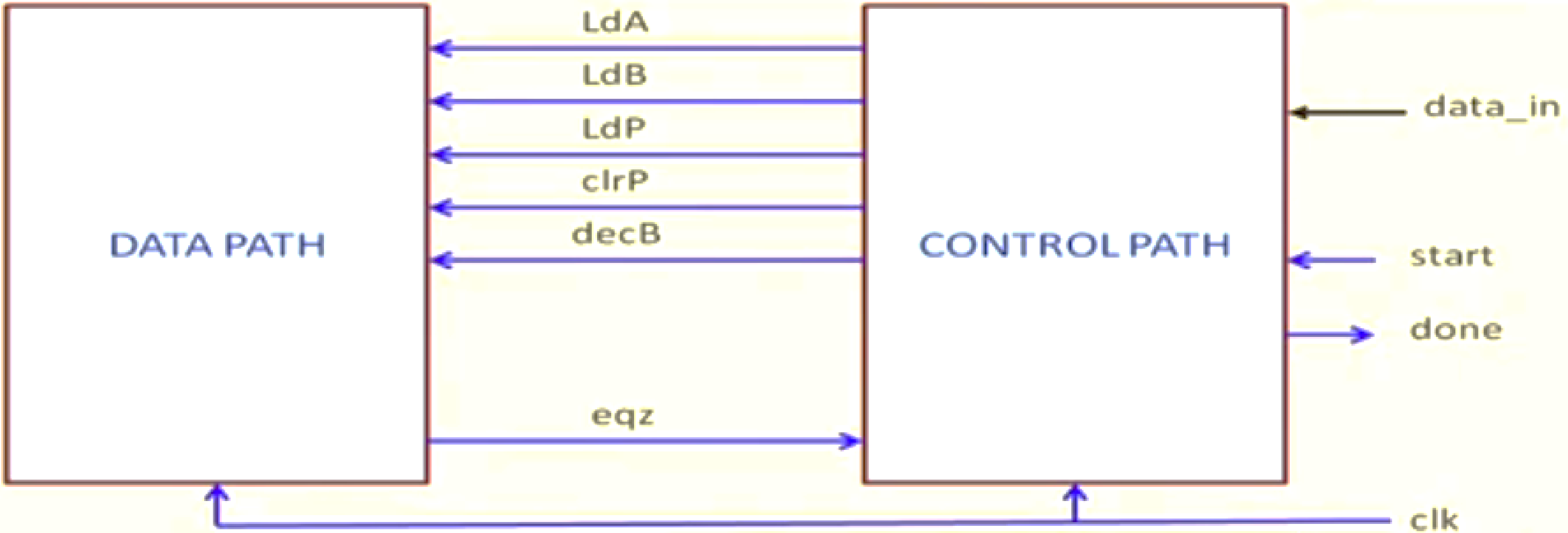
Example 1: Multiplication by Repeated Addition

- We consider a simple algorithm using repeated addition.
  - Assume B is non-zero.
- We identify the functional blocks required in the data path, and the corresponding control signals.
- Then we design the FSM to implement the multiplication algorithm using the data path.

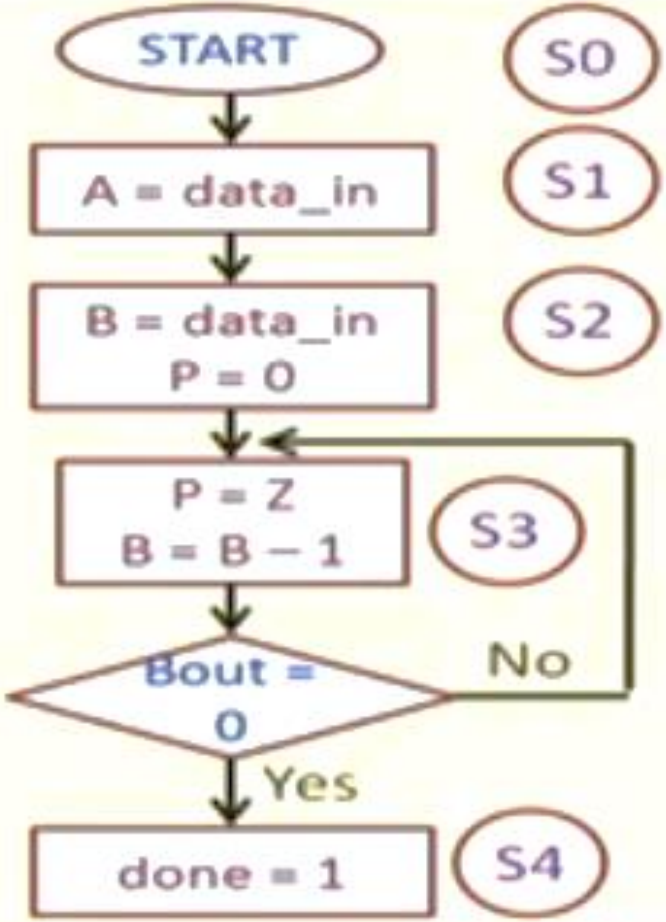


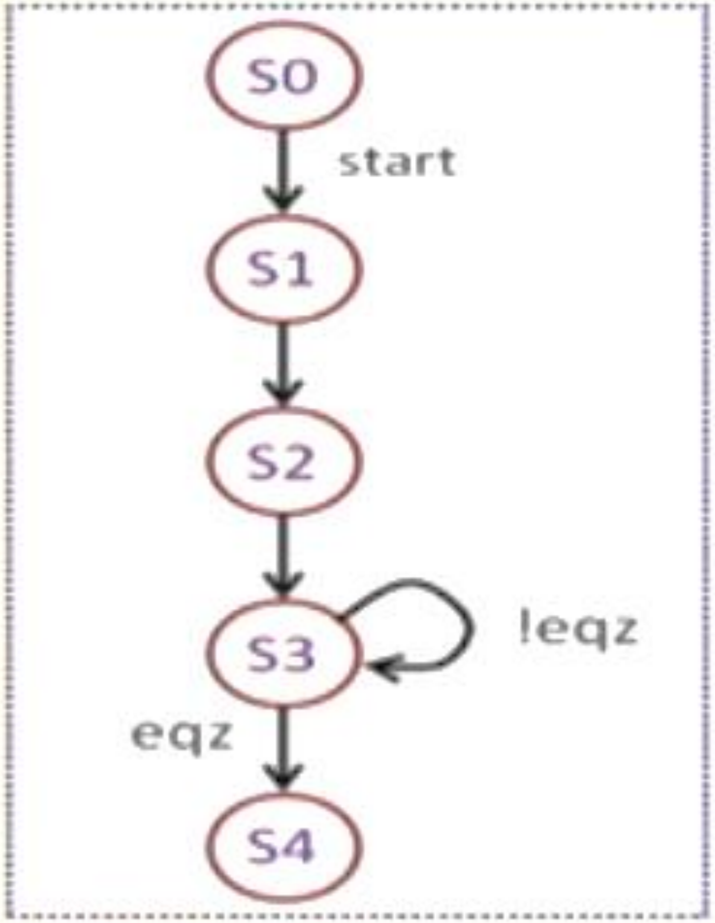


DATA PATH

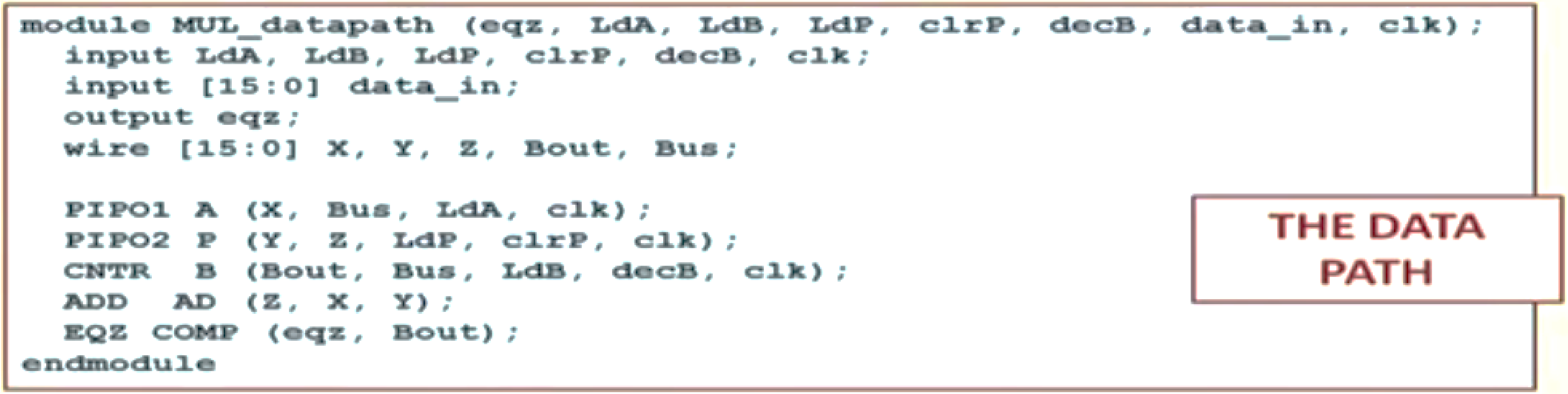

LdA
LdB
LdP
clrP
decB
eqz
DATA PATH
CONTROL PATH
data_in
start
done
clk

CONTROL PATH

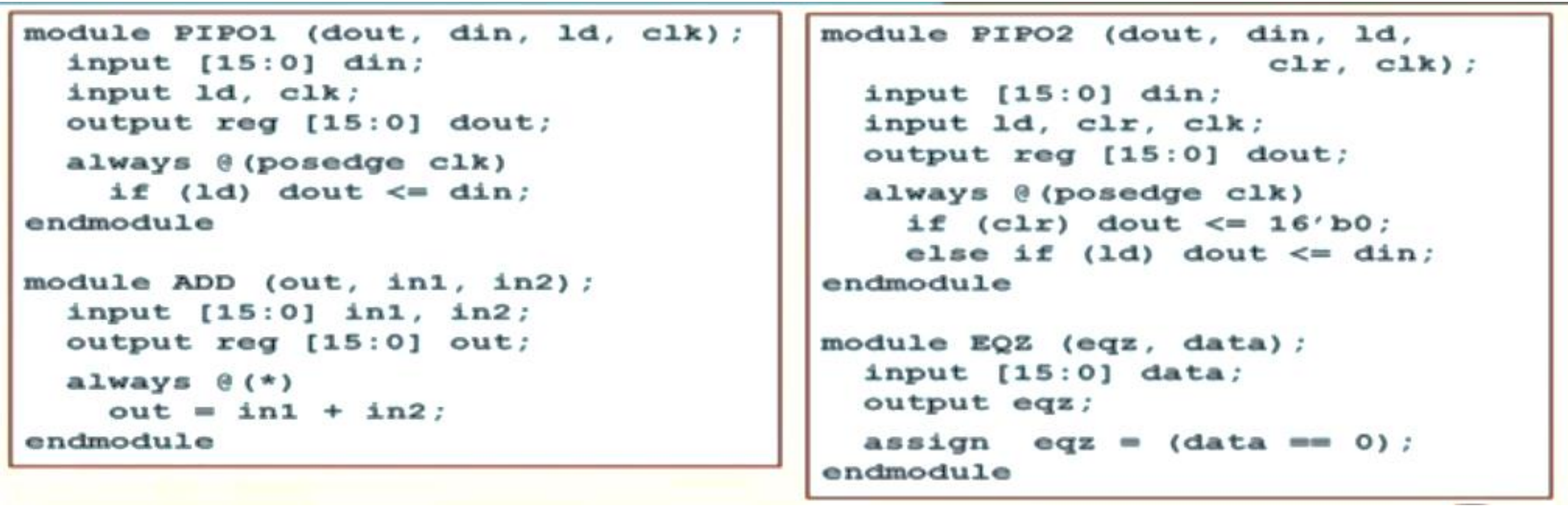




THE DATA PATH

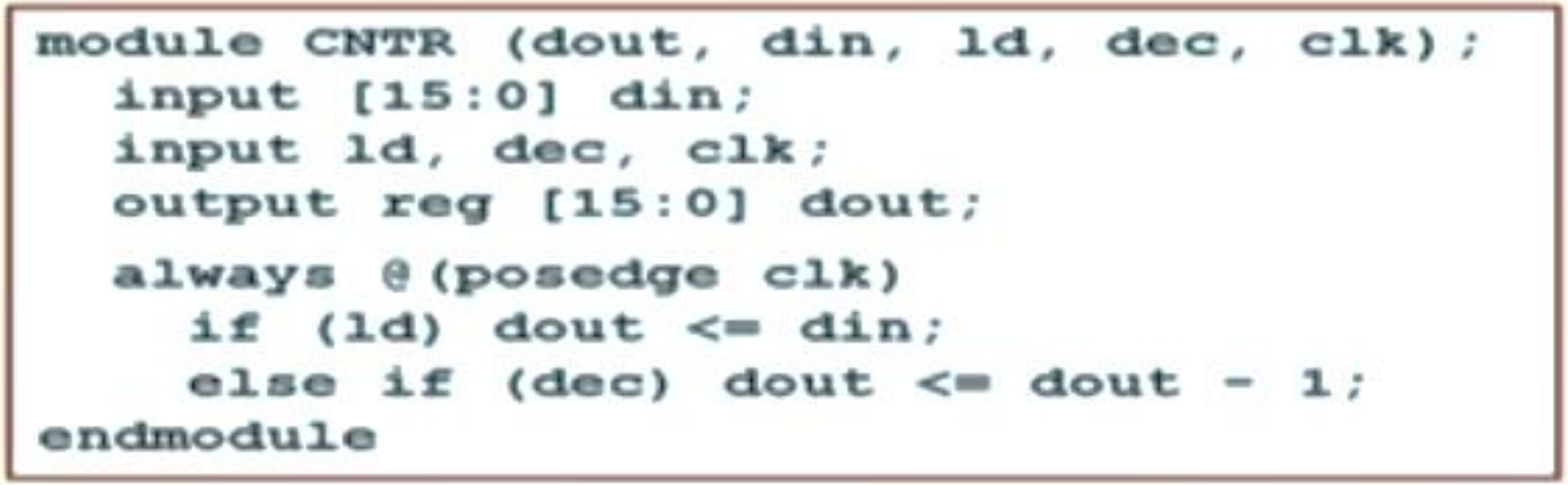


```
module MUL_datapath (eqz, LdA, LdB, LdP, clrP, decB, data_in, clk);
  input LdA, LdB, LdP, clrP, decB, clk;
  input [15:0] data_in;
  output eqz;
  wire [15:0] X, Y, Z, Bout, Bus;

  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
```

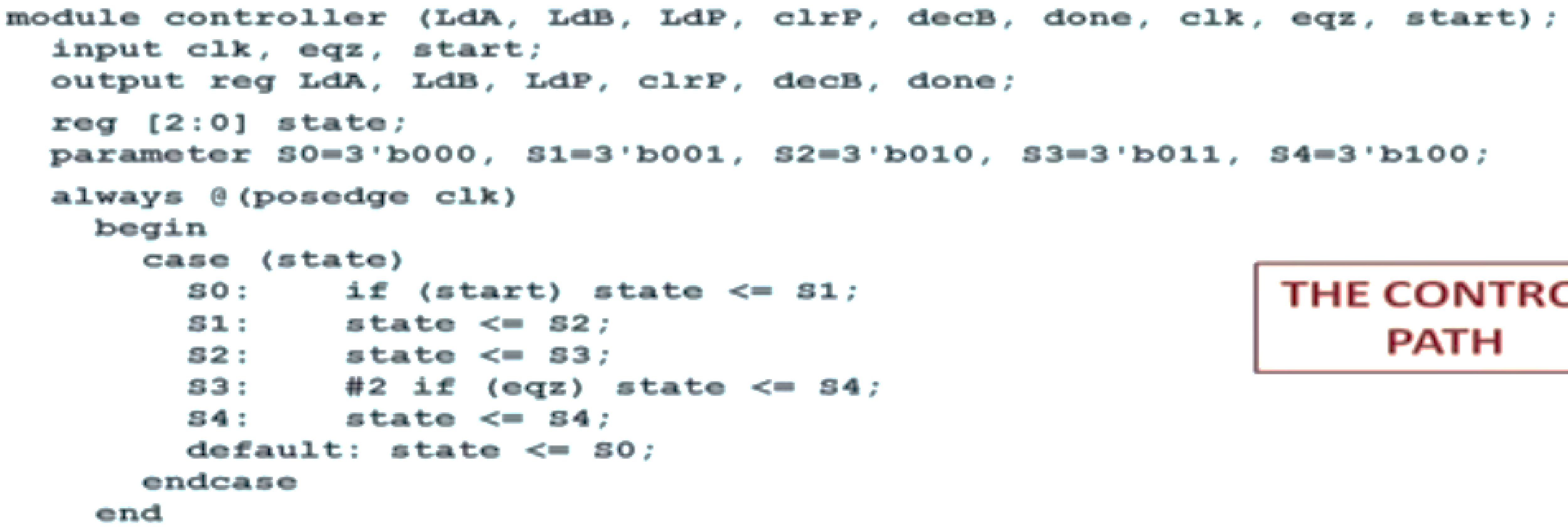


```
module PIPO1 (dout, din, ld, clk);
  input [15:0] din;
  input ld, clk;
  output reg [15:0] dout;
  always @(posedge clk)
    if (ld) dout <= din;
endmodule

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

```
module PIPO2 (dout, din, ld,
                          clr, clk);
  input [15:0] din;
  input ld, clr, clk;
  output reg [15:0] dout;
  always @(posedge clk)
    if (clr) dout <= 16'b0;
    else if (ld) dout <= din;
endmodule

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



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

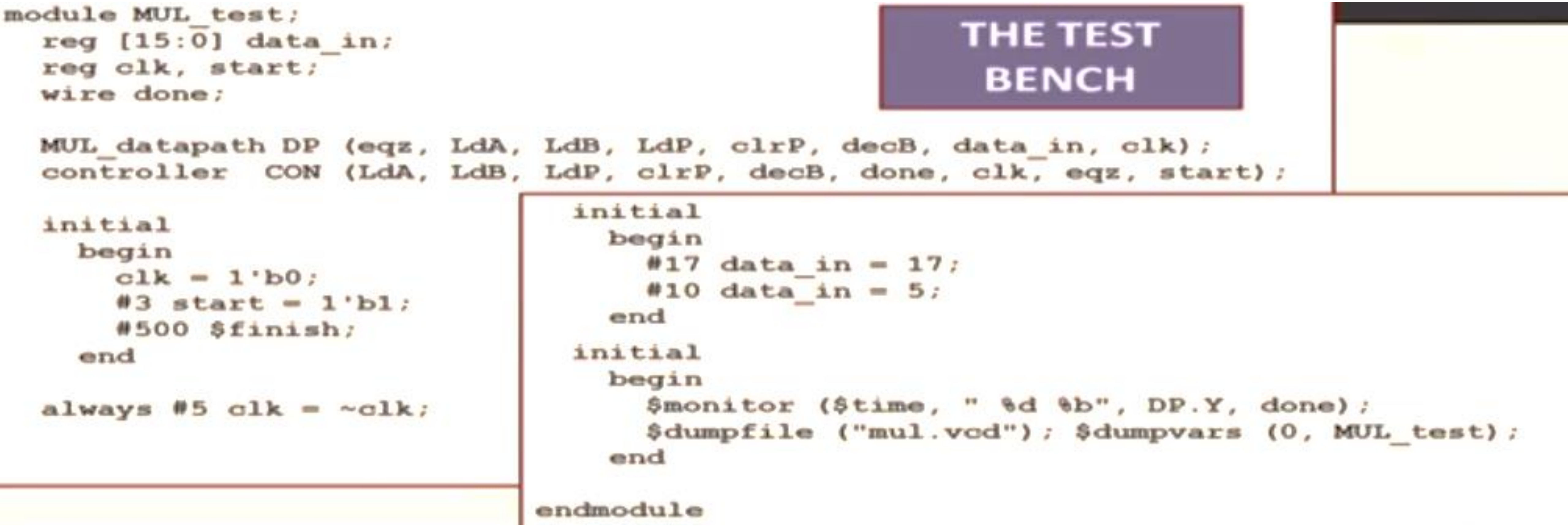


```
module controller (LdA, LdB, LdP, clrP, decB, done, clk, eqz, start);
  input clk, eqz, start;
  output reg LdA, LdB, LdP, clrP, decB, done;
  reg [2:0] state;
  parameter S0=3'b000, S1=3'b001, S2=3'b010, S3=3'b011, S4=3'b100;
  always @(posedge clk)
    begin
      case (state)
        S0:     if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

THE CONTROL PATH

```
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
```

THE TEST BENCH



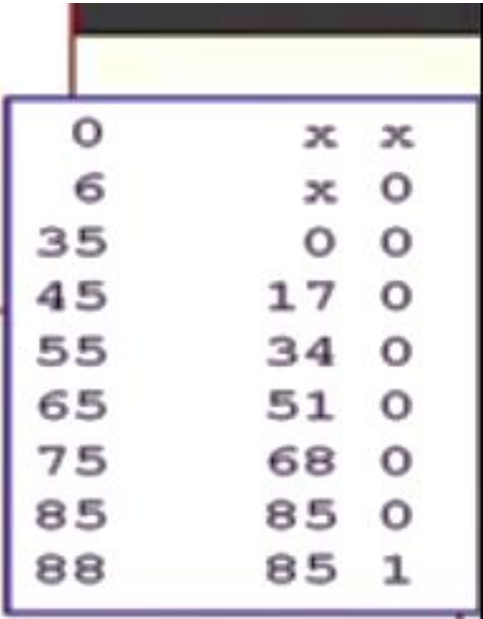
```
module MUL_test;
  reg [15:0] data_in;
  reg clk, start;
  wire done;

  MUL_datapath DP (eqz, LdA, LdB, LdP, clrP, decB, data_in, clk);
  controller  CON (LdA, LdB, LdP, clrP, decB, done, clk, eqz, start);

  initial
    begin
      clk = 1'b0;
      #3 start = 1'b1;
      #500 $finish;
    end

  always #5 clk = ~clk;
```

```
  initial
    begin
      #17 data_in = 17;
      #10 data_in = 5;
    end

  initial
    begin
      $monitor ($time, " %d %b", DP.Y, done);
      $dumpfile ("mul.vcd"); $dumpvars (0, MUL_test);
    end

endmodule
```

THE TEST BENCH

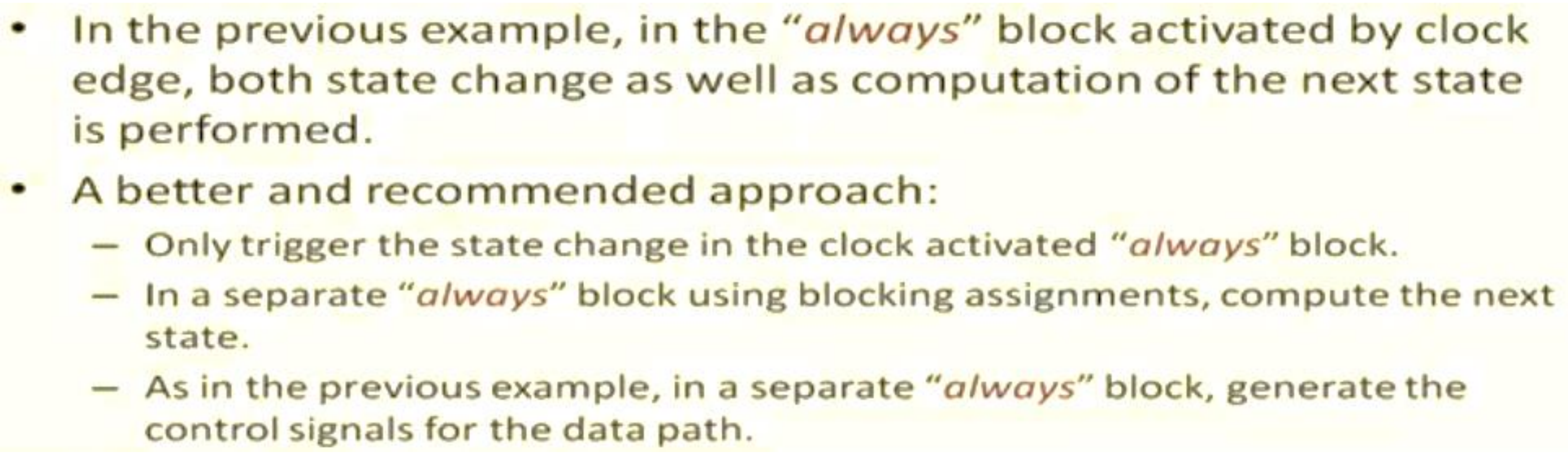
```
module MUL_test;
  reg [15:0] data_in;
  reg clk, start;
  wire done;

  MUL_datapath DP (eqz, LdA, LdB, LdP, clrP, decB, data_in, clk);
  controller  CON (LdA, LdB, LdP, clrP, decB, done, clk, eqz, start);

  initial
    begin
      clk = 1'b0;
      #3 start = 1'b1;
      #500 $finish;
    end

  always #5 clk = ~clk;
```

```
   initial
     begin
       #17 data_in = 17;
       #10 data_in = 5;
     end
   initial
     begin
       $monitor ($time, " %d %b", DP.Y, done);
       $dumpfile ("mul.vcd"); $dumpvars (0, MUL_test);
     end

endmodule
```



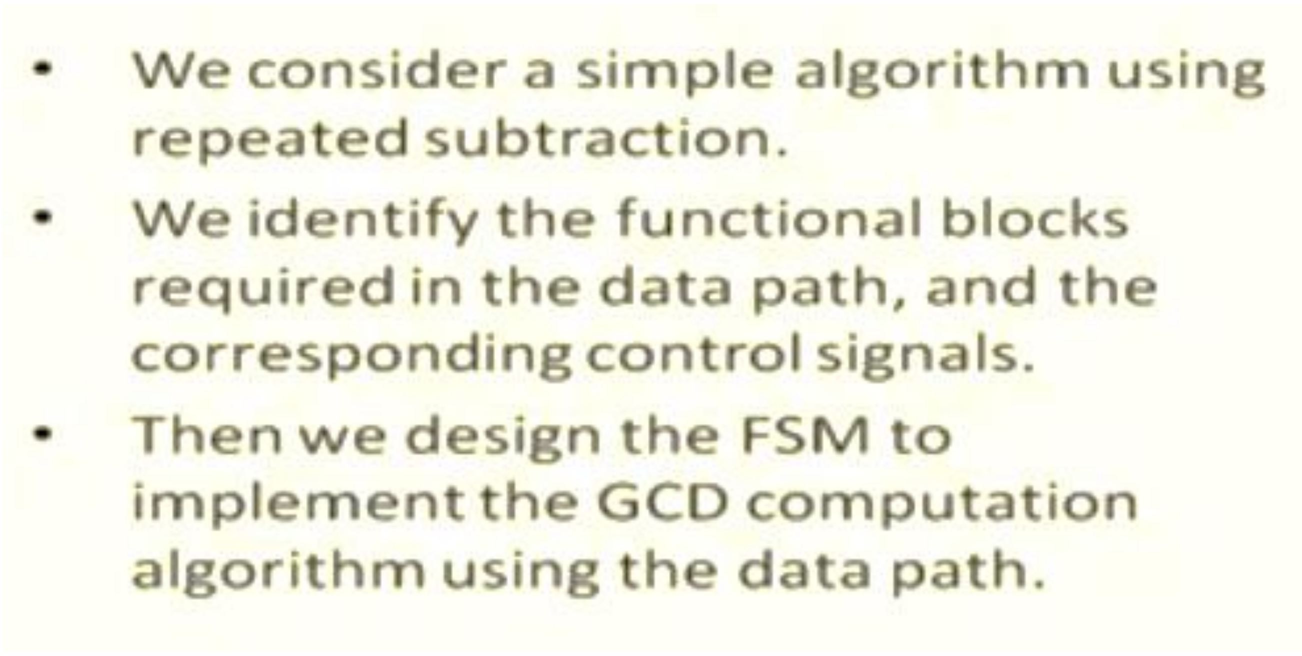
```
   0        x x
   6        x 0
  35        0 0
  45       17 0
  55       34 0
  65       51 0
  75       68 0
  85       85 0
  88       85 1
```

A Better Style of Modeling Data/Control Path

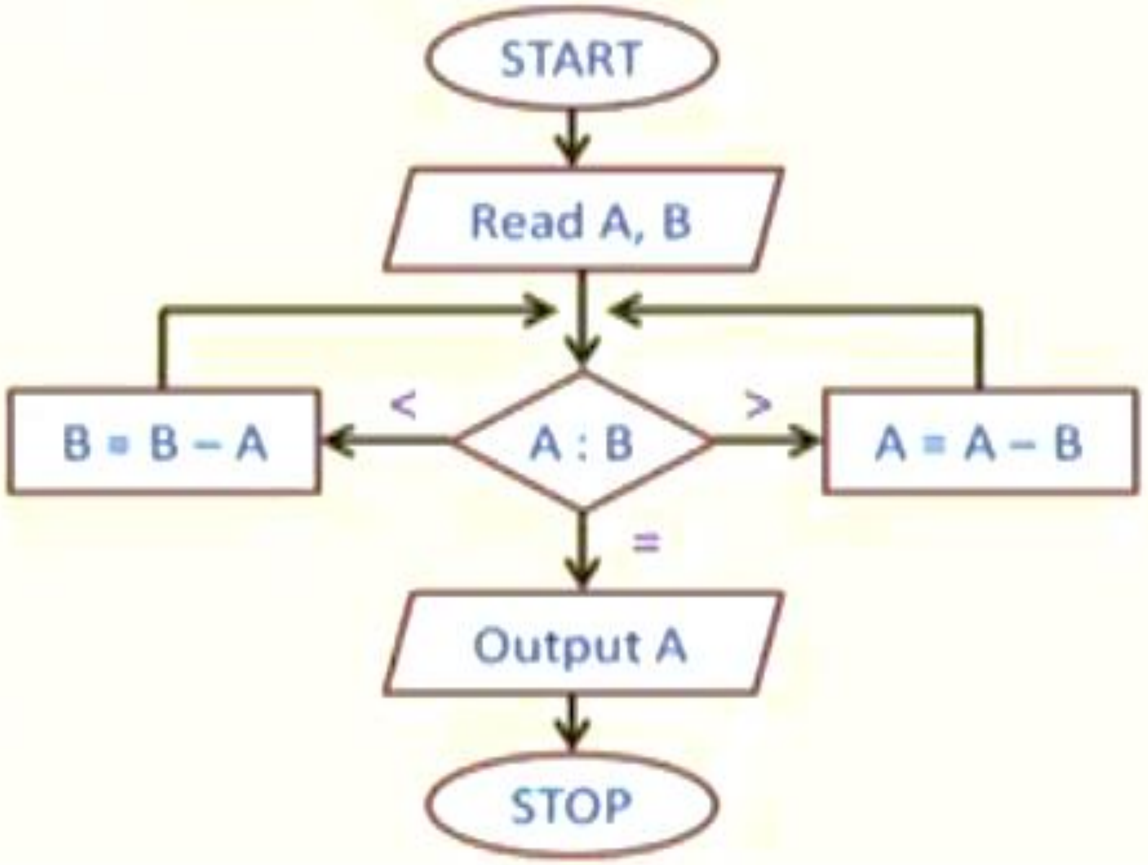


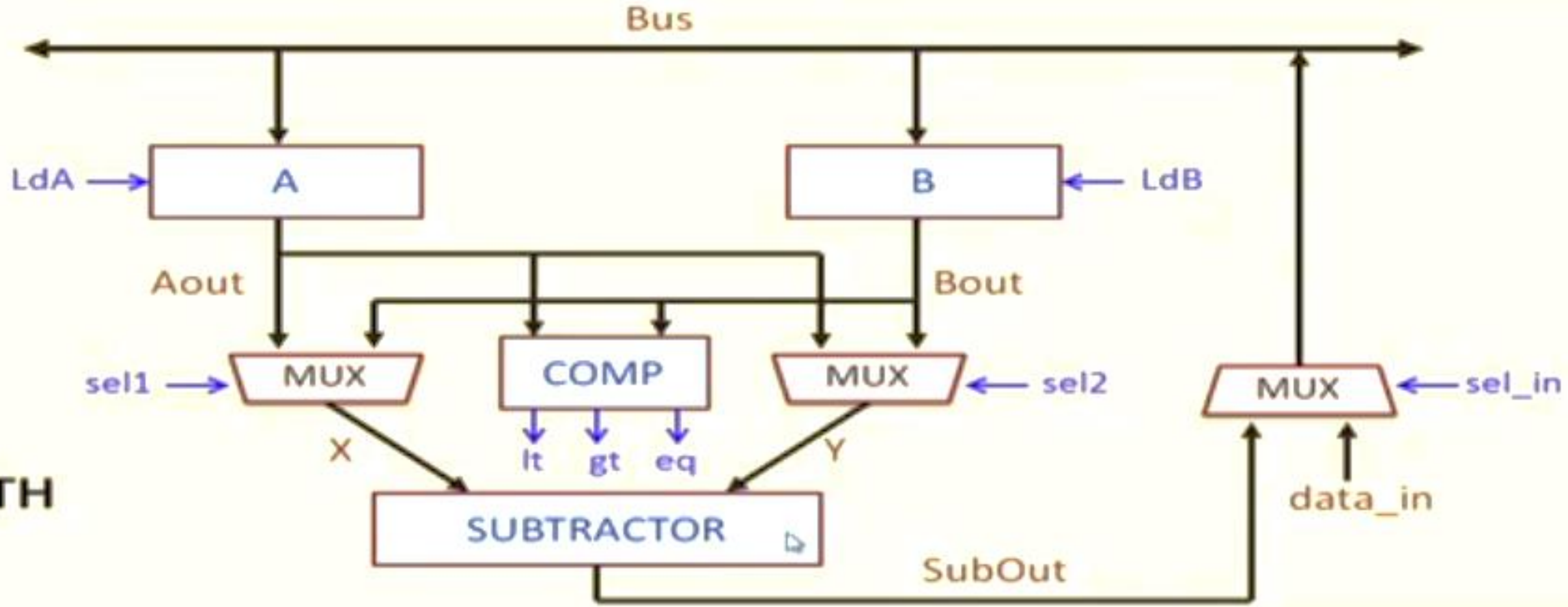
- In the previous example, in the "*always*" block activated by clock edge, both state change as well as computation of the next state is performed.
- A better and recommended approach:
  - Only trigger the state change in the clock activated "*always*" block.
  - In a separate "*always*" block using blocking assignments, compute the next state.
  - As in the previous example, in a separate "*always*" block, generate the control signals for the data path.

Example 2: GCD Computation

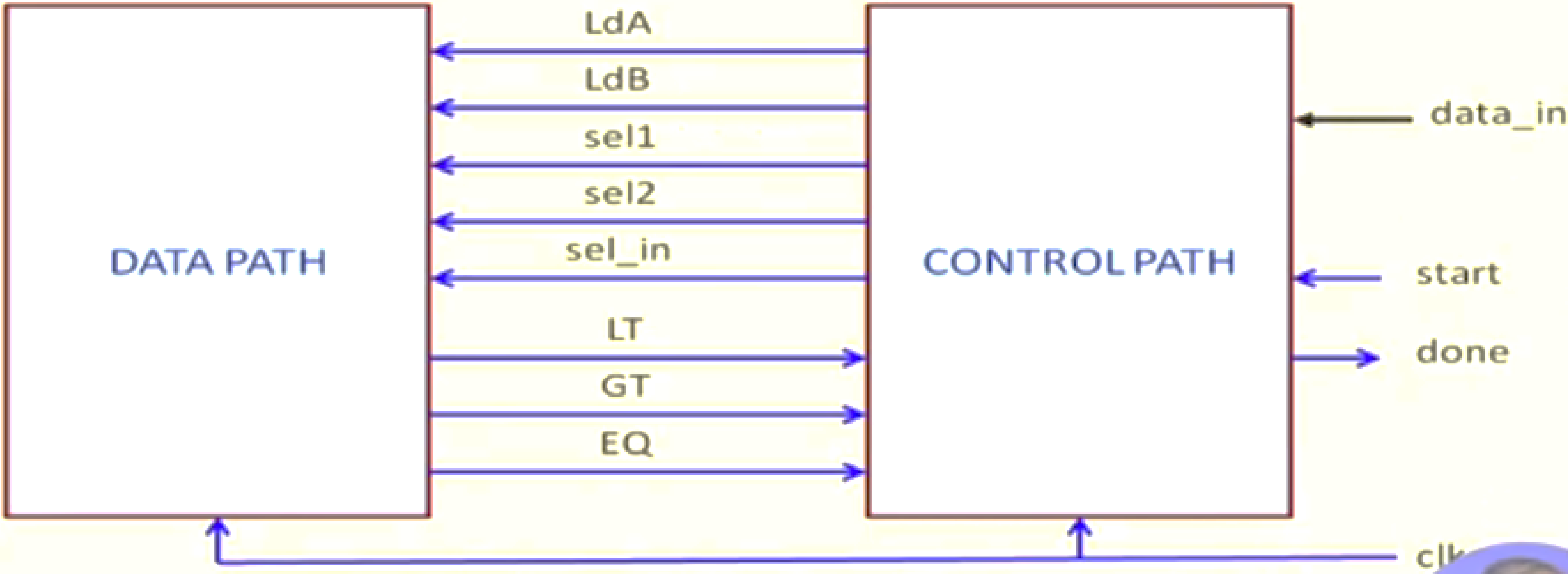


- We consider a simple algorithm using repeated subtraction.
- We identify the functional blocks required in the data path, and the corresponding control signals.
- Then we design the FSM to implement the GCD computation algorithm using the data path.

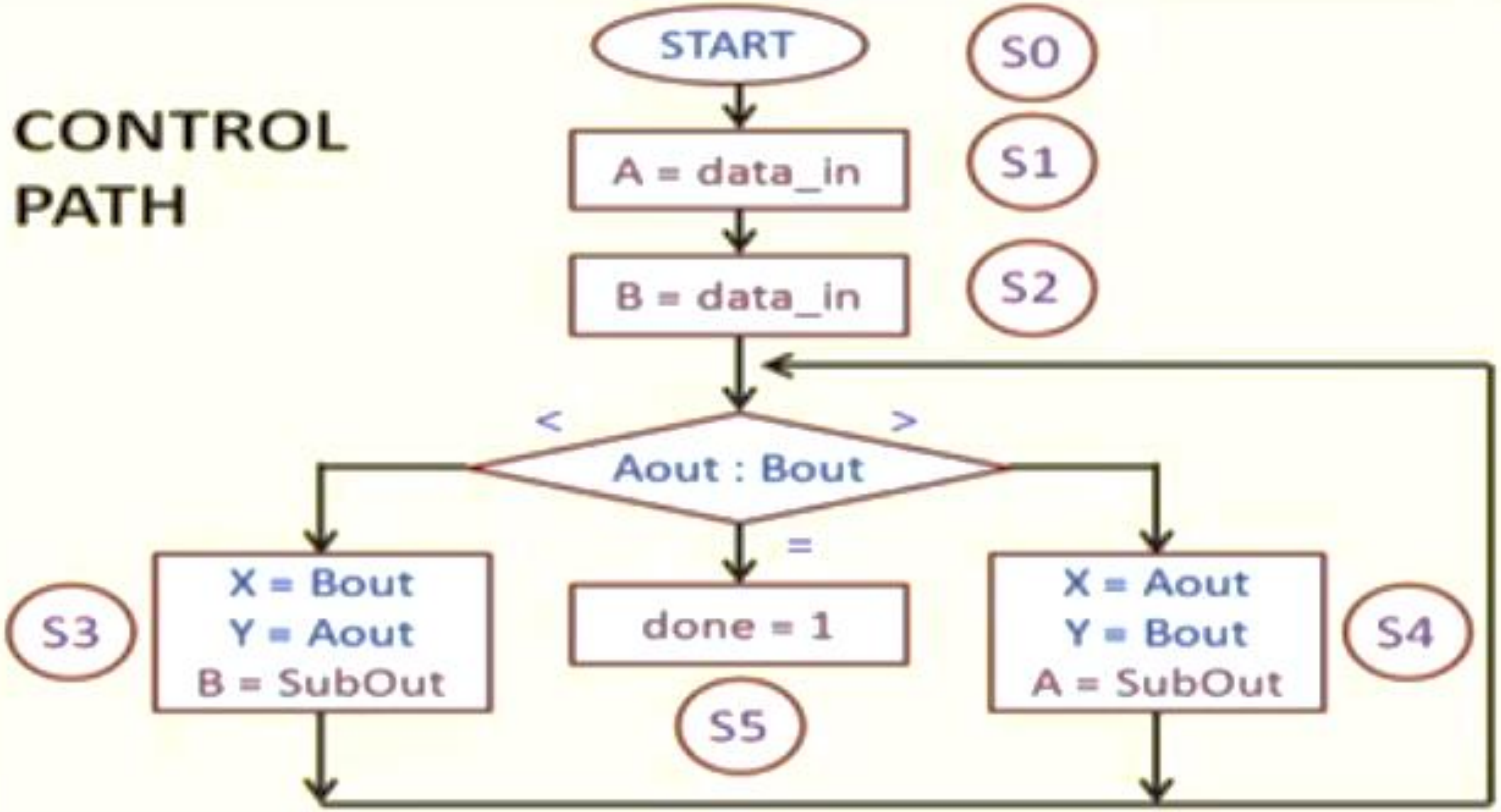


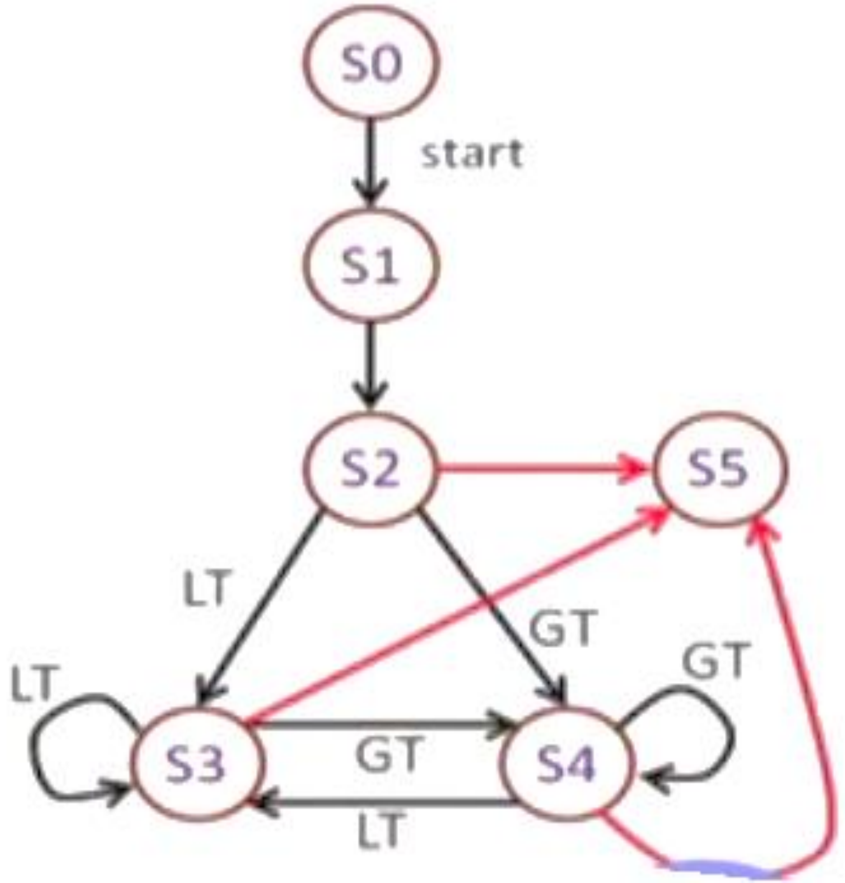

Bus
LdA
A
B
LdB
Aout
Bout
sel1
MUX
COMP
MUX
sel2
MUX
sel_in
X
lt
gt
eq
Y
data_in
SUBTRACTOR
SubOut

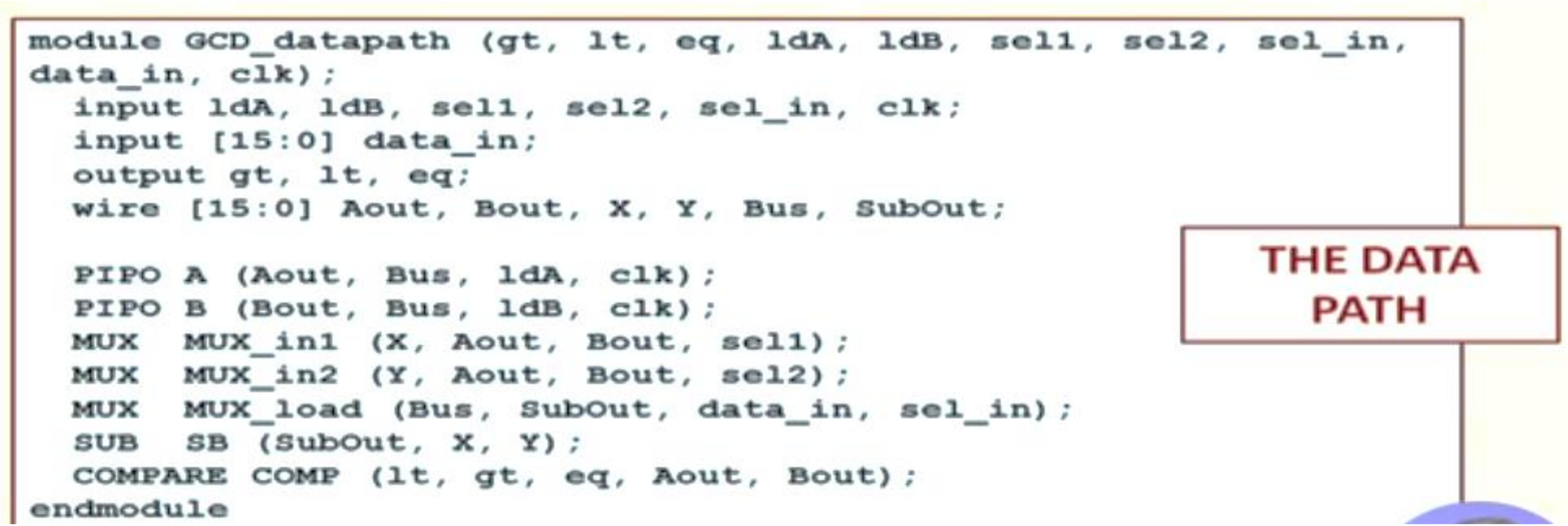
DATA PATH


LdA
LdB
sel1
sel2
sel_in
LT
GT
EQ
DATA PATH
CONTROL PATH
data_in
start
done
clk

CONTROL PATH

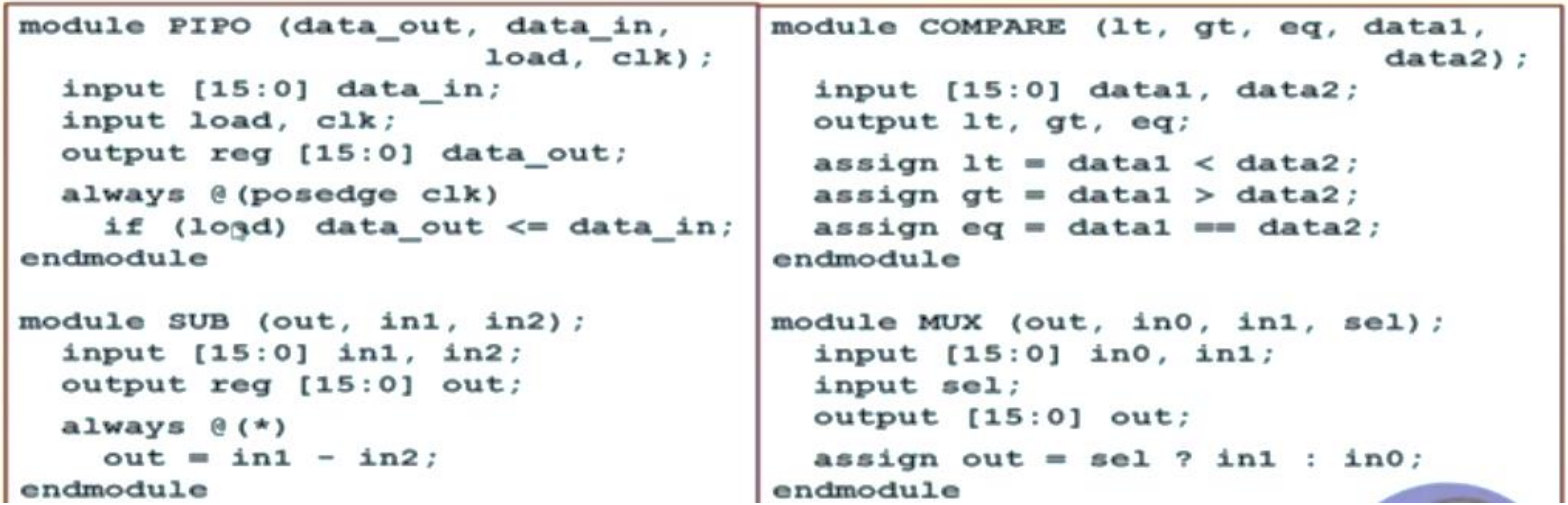






```
module GCD_datapath (gt, lt, eq, ldA, ldB, sel1, sel2, sel_in,
data_in, clk);
  input ldA, ldB, sel1, sel2, sel_in, clk;
  input [15:0] data_in;
  output gt, lt, eq;
  wire [15:0] Aout, Bout, X, Y, Bus, SubOut;

  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
```

THE DATA PATH

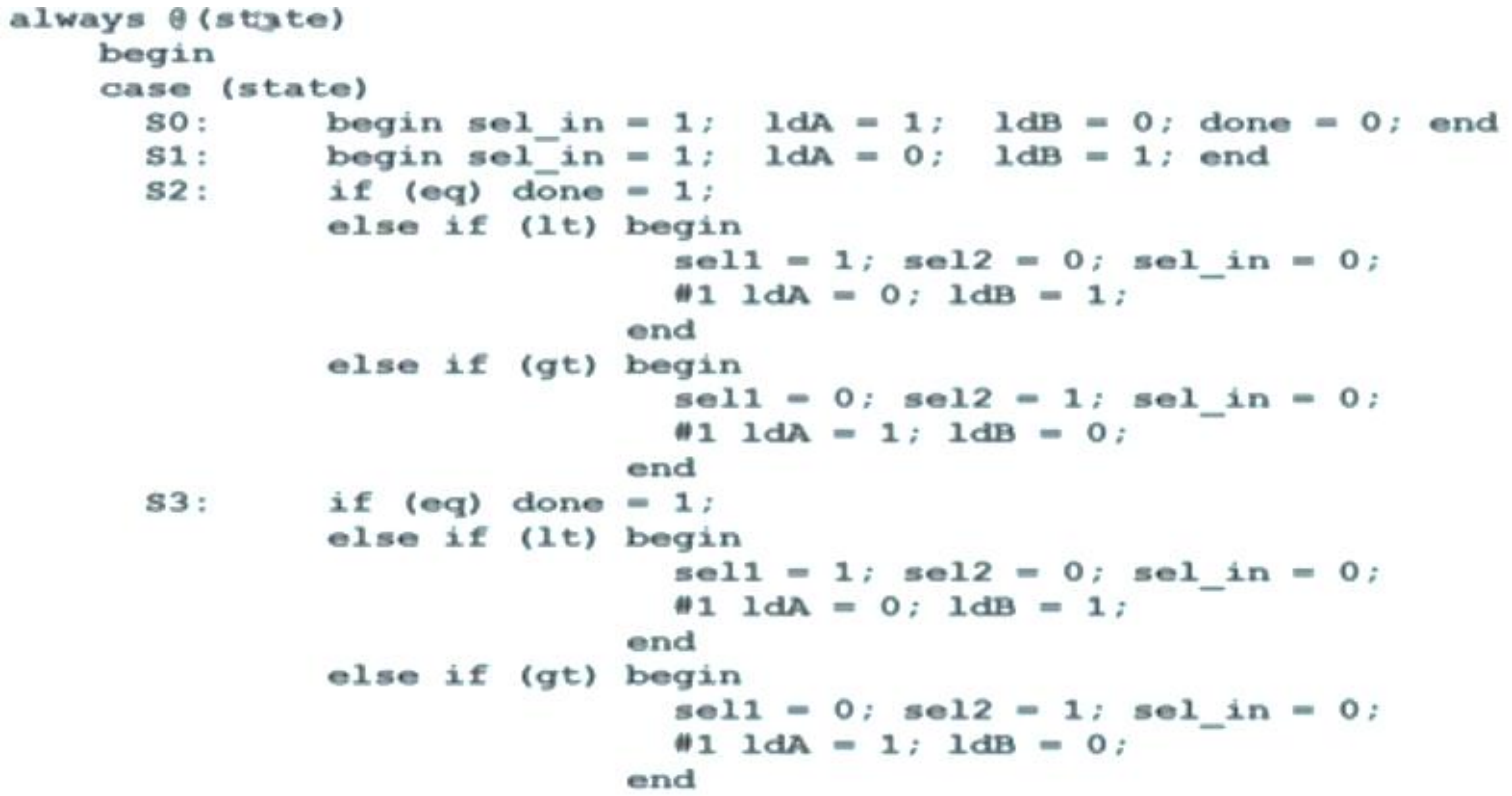


```
module PIPO (data_out, data_in,
                      load, clk);
  input [15:0] data_in;
  input load, clk;
  output reg [15:0] data_out;
  always @(posedge clk)
    if (load) data_out <= data_in;
endmodule

module SUB (out, in1, in2);
  input [15:0] in1, in2;
  output reg [15:0] out;
  always @(*)
    out = in1 - in2;
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 : in0;
endmodule
```

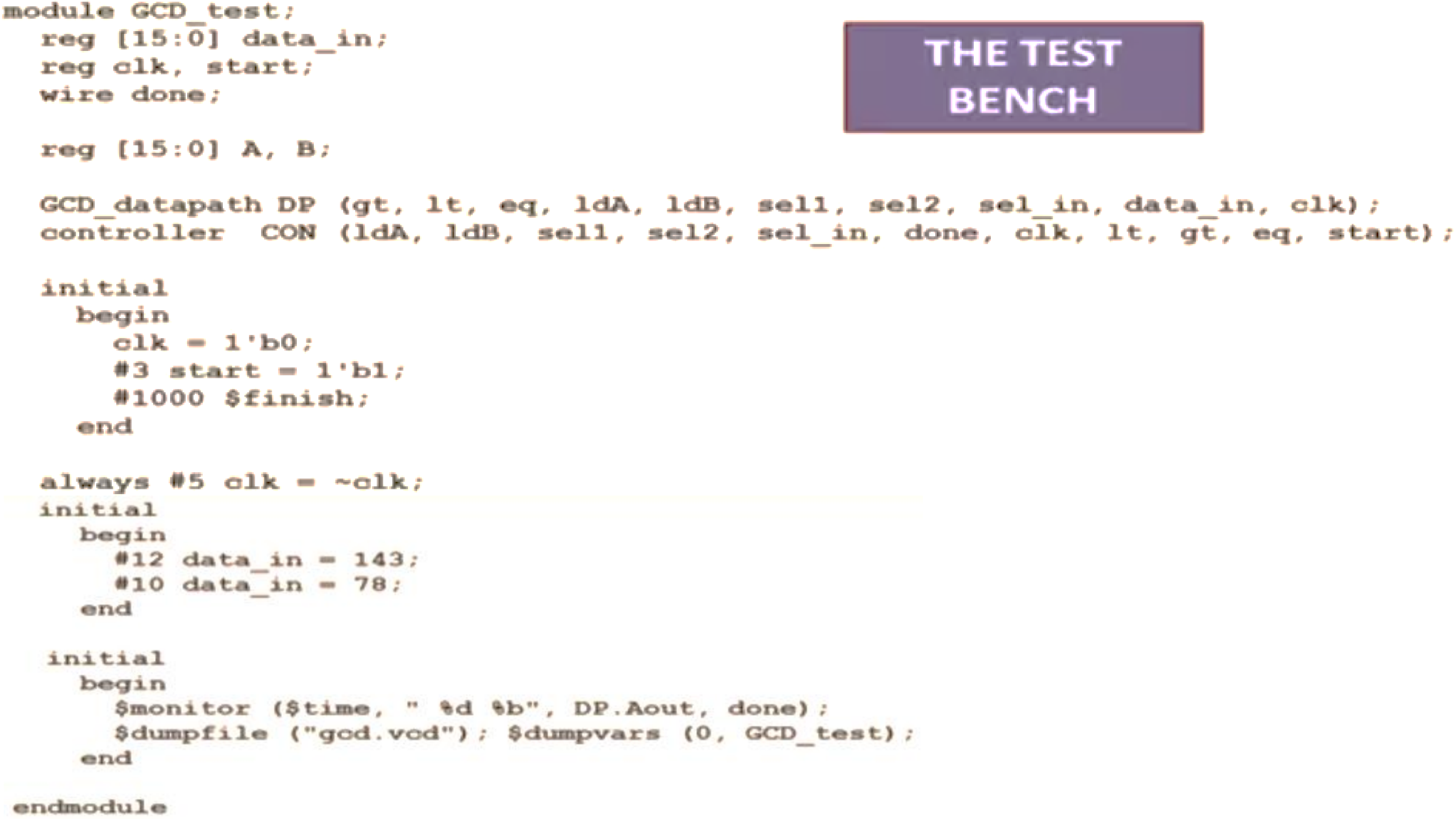
```
module controller (ldA, ldB, sel1, sel2, sel_in, done, clk, lt, gt, eq, start);
  input clk, lt, gt, eq, start;
  output reg ldA, ldB, sel1, sel2, sel_in, done;
  reg [2:0] state;
  parameter S0=3'b000, S1=3'b001, S2=3'b010, S3=3'b011, S4=3'b100, S5=3'b101;
  always @(posedge clk)
    begin
    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

THE CONTROL PATH



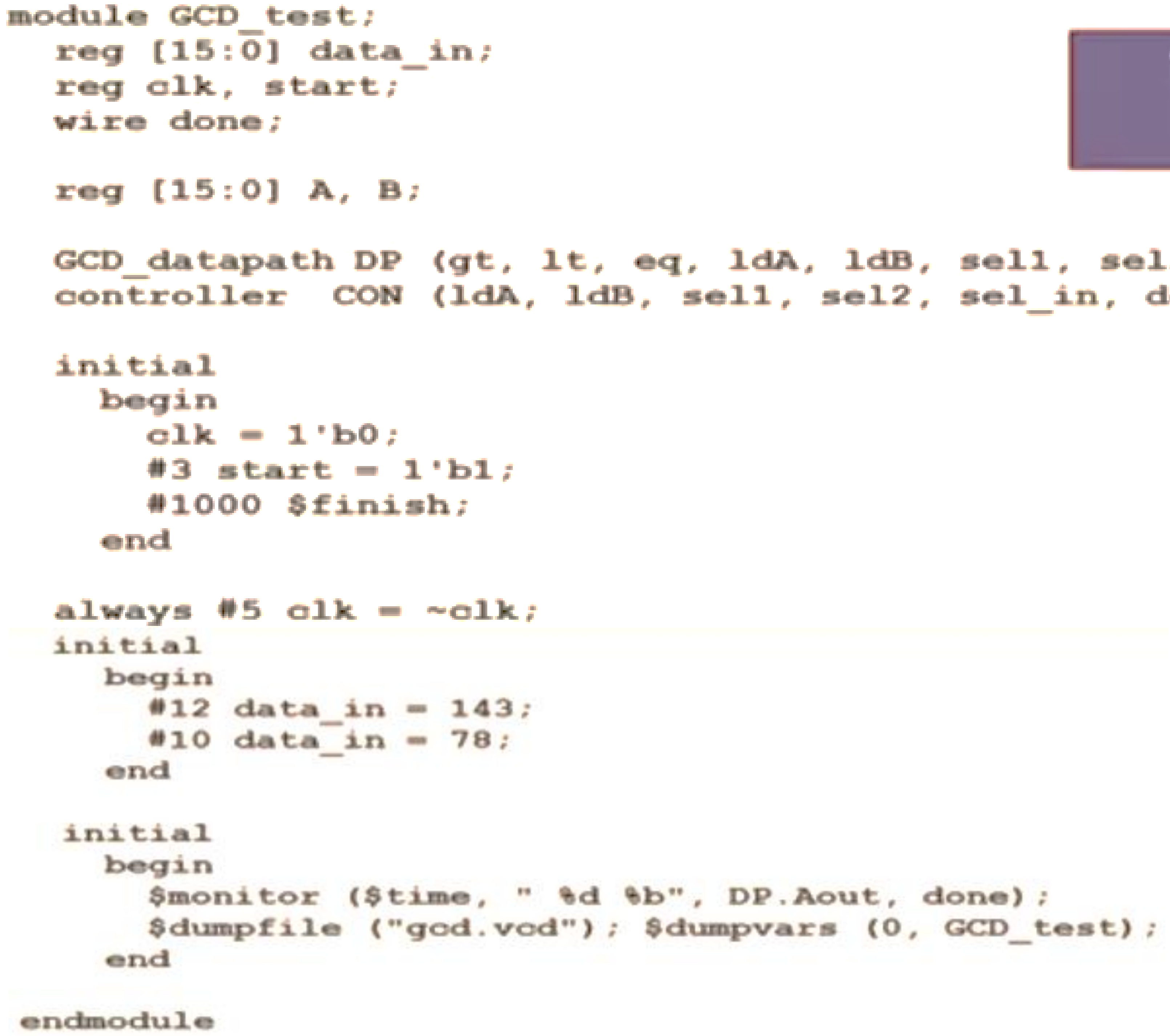
```
always @(state)
    begin
    case (state)
      S0:      begin sel_in = 1;  ldA = 1;  ldB = 0; done = 0; end
      S1:      begin sel_in = 1;  ldA = 0;  ldB = 1; end
      S2:      if (eq) done = 1;
               else if (lt) begin
                              sel1 = 1; sel2 = 0; sel_in = 0;
                              #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; ldA = 0;
                ldB = 0;
              end
       default: begin ldA = 0; ldB = 0; end
    endcase
    end

endmodule
```

THE TEST BENCH

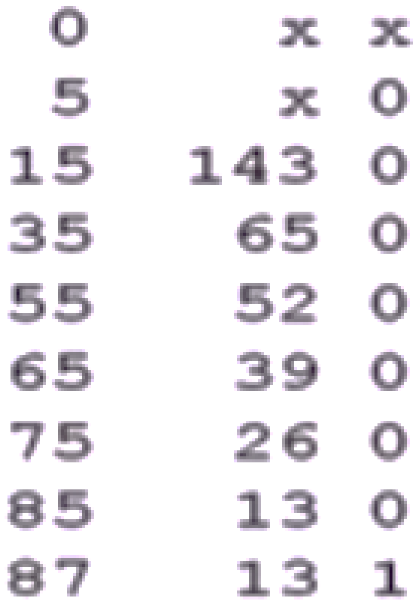


```
module GCD_test;
  reg [15:0] data_in;
  reg clk, start;
  wire done;

  reg [15:0] A, B;

  GCD_datapath DP (gt, lt, eq, ldA, ldB, sel1, sel2, sel_in, data_in, clk);
  controller  CON (ldA, ldB, sel1, sel2, sel_in, 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", DP.Aout, done);
      $dumpfile ("gcd.vcd"); $dumpvars (0, GCD_test);
    end

endmodule
```

THE TEST BENCH

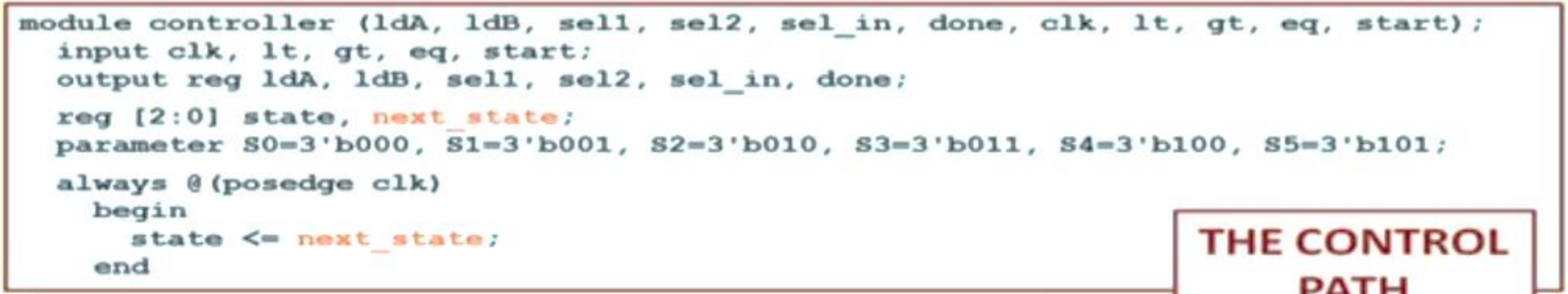


```
module GCD_test;
  reg [15:0] data_in;
  reg clk, start;
  wire done;

  reg [15:0] A, B;

  GCD_datapath DP (gt, lt, eq, ldA, ldB, sel1, sel2, sel_in, data_in, clk);
  controller  CON (ldA, ldB, sel1, sel2, sel_in, 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", DP.Aout, done);
      $dumpfile ("gcd.vcd"); $dumpvars (0, GCD_test);
    end

 endmodule
```



```
 0     x  x
 5     x  0
15   143  0
35    65  0
55    52  0
65    39  0
75    26  0
85    13  0
87    13  1
```

MODELING THE CONTROL PATH USING THE ALTERNATE APPROACH

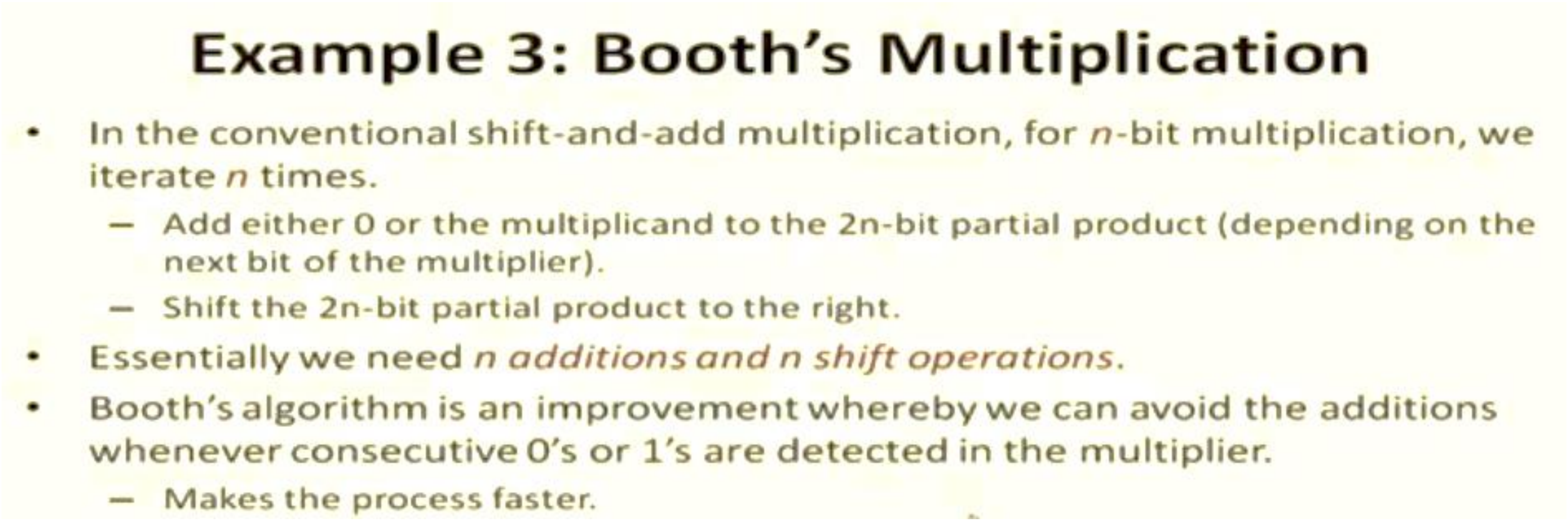


```
module controller (ldA, ldB, sel1, sel2, sel_in, done, clk, lt, gt, eq, start);
  input clk, lt, gt, eq, start;
  output reg ldA, ldB, sel1, sel2, sel_in, done;
  reg [2:0] state, next_state;
  parameter S0=3'b000, S1=3'b001, S2=3'b010, S3=3'b011, S4=3'b100, S5=3'b101;
  always @(posedge clk)
  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

THE CONTROL PATH

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

```
       S4:     if (eq) begin done = 1; next_state = S5; end
               else if (lt) begin
                              sel1 = 1; sel2 = 0; sel_in = 0; next_state = S3;
                              #1 ldA = 0; ldB = 1;
                            end
               else if (gt) begin
                              sel1 = 0; sel2 = 1; sel_in = 0; next_state = S4;
                              #1 ldA = 1; ldB = 0;
                            end
        S5:    begin
                 done = 1; sel1 = 0; sel2 = 0; ldA = 0;
                 ldB = 0; next_state = S5;
               end
        default: begin ldA = 0; ldB = 0; next_state = S0; end
     endcase
     end

endmodule
```

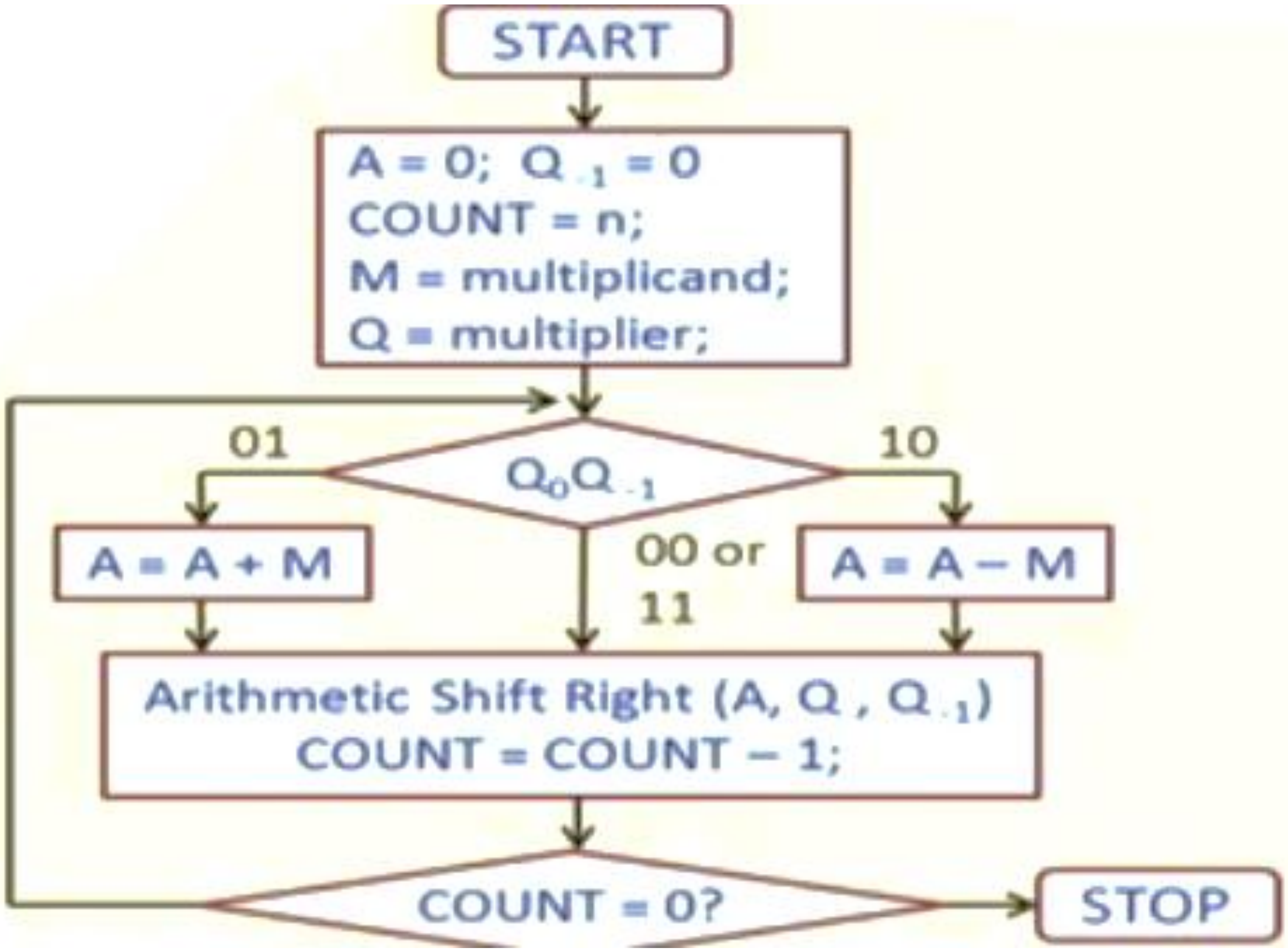


Example 3: Booth's Multiplication

- In the conventional shift-and-add multiplication, for *n*-bit multiplication, we iterate *n* times.
  - Add either 0 or the multiplicand to the 2n-bit partial product (depending on the next bit of the multiplier).
  - Shift the 2n-bit partial product to the right.
- Essentially we need *n additions and n shift operations*.
- Booth's algorithm is an improvement whereby we can avoid the additions whenever consecutive 0's or 1's are detected in the multiplier.
  - Makes the process faster.

Basic Idea Behind Booth's Algorithm

- We inspect two bits of the multiplier ($Q_{i,}$ Q_{i-1}) at a time.
  - If the bits are same (00 or 11), we only shift the partial product.
  - If the bits are 01, we do an addition and then shift.
  - If the bits are 10, we do a subtraction and then shift.
- Q_{-1} is assumed to be equal to 0.
- Significantly reduces the number of additions / subtractions.



START

$A = 0$; $Q_{-1} = 0$
COUNT = n;
M = multiplicand;
Q = multiplier;

Q_0Q_{-1}

01

10

00 or 11

$A = A + M$

$A = A - M$

Arithmetic Shift Right (A, Q , Q_{-1})
COUNT = COUNT – 1;

COUNT = 0?

STOP

M: n-bit multiplicand

Q: n-bit multiplier

A: n-bit temporary register

Q_{-1}: 1-bit flip-flop

Example 1: (-10) x (13)

Assume 5-bit numbers.

M: $(1\,0\,1\,1\,0)_2$
-M: $(0\,1\,0\,1\,0)_2$
Q: $(0\,1\,1\,0\,1)_2$

Product = -130
= $(1\,1\,0\,1\,1\,1\,1\,1\,1\,0)_2$

A	Q	Q_{-1}		
0 0 0 0 0	0 1 1 0 1	0	Initialization	
0 1 0 1 0	0 1 1 0 1	0	A = A - M	Step 1
0 0 1 0 1	0 0 1 1 0	1	Shift	
1 1 0 1 1	0 0 1 1 0	1	A = A + M	Step 2
1 1 1 0 1	1 0 0 1 1	0	Shift	
0 0 1 1 1	1 0 0 1 1	0	A = A - M	Step 3
0 0 0 1 1	1 1 0 0 1	1	Shift	
0 0 0 0 1	1 1 1 1 0	1	Shift	Step 4
1 0 1 1 1	1 1 1 0 0	1	A = A + M	Step 5
1 1 0 1 1	1 1 1 1 0	0	Shift	

Example 2:

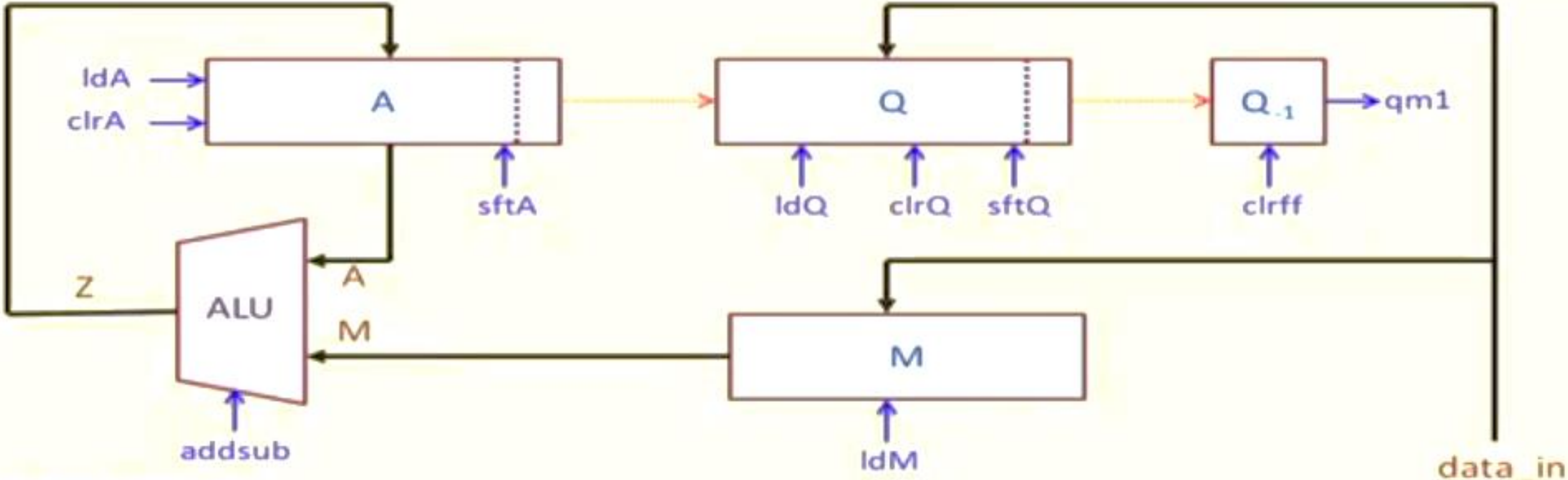
(-31) x (28)

Assume 6-bit numbers.

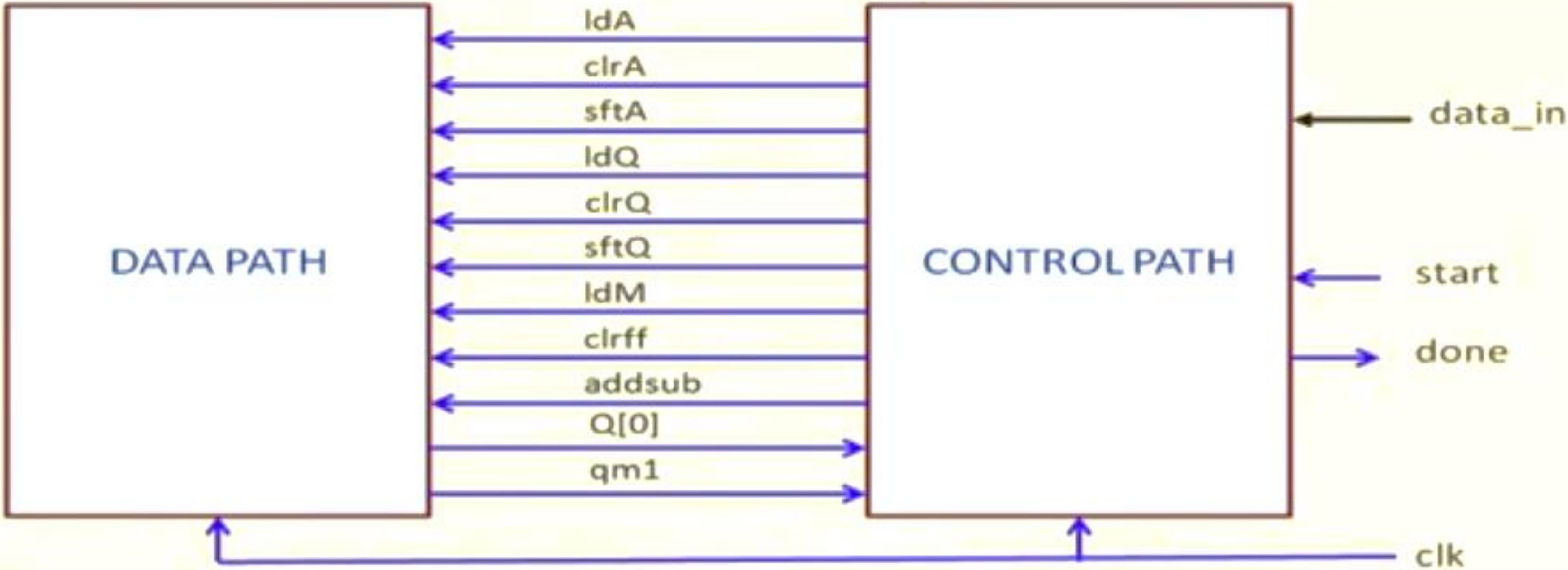
M: $(1\,0\,0\,0\,0\,1)_2$
-M: $(0\,1\,1\,1\,1\,1)_2$
Q: $(0\,1\,1\,1\,0\,0)_2$

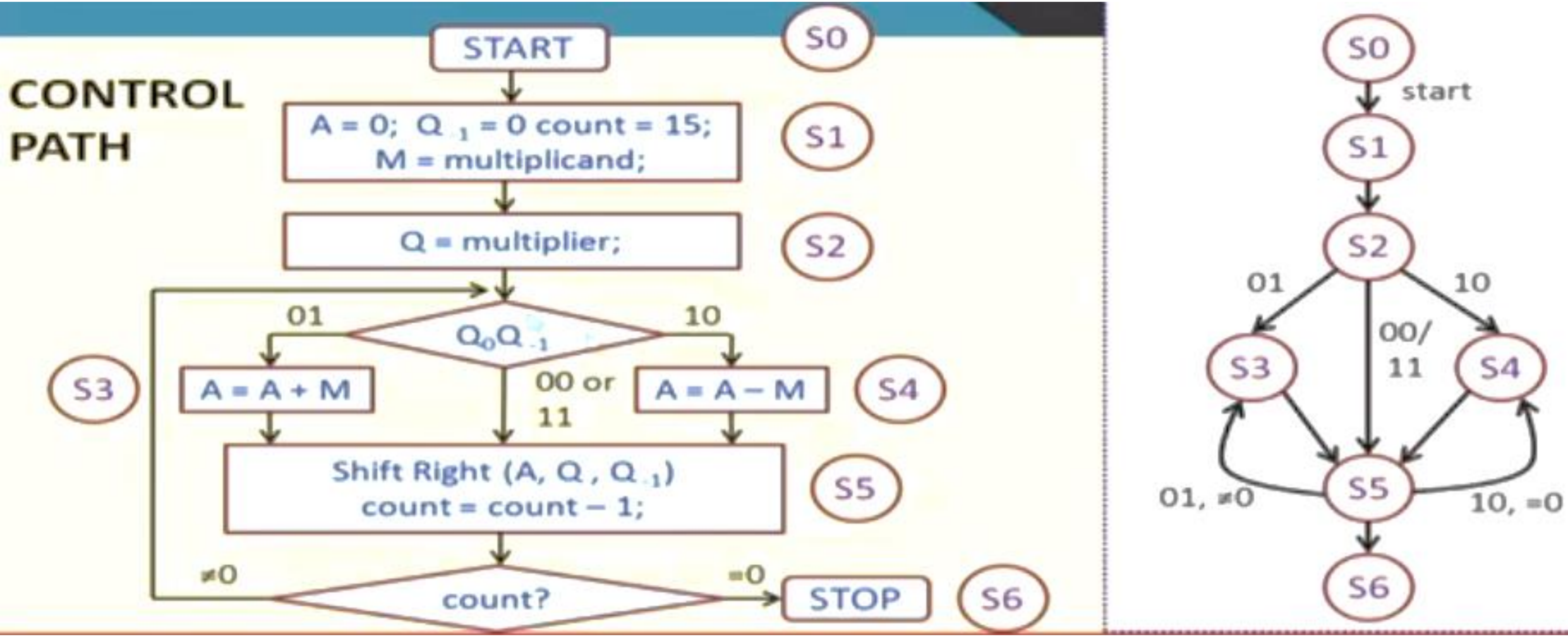
Product = -868
= $(1\,1\,0\,0\,1\,0$
$0\,1\,1\,1\,0\,0)_2$

A	Q	Q_{-1}		
0 0 0 0 0 0	0 1 1 1 0 0	0	Initialization	
0 0 0 0 0 0	0 0 1 1 1 0	0	Shift	Step 1
0 0 0 0 0 0	0 0 0 1 1 1	0	Shift	Step 2
0 1 1 1 1 1	0 0 0 1 1 1	0	A = A – M	Step 3
0 0 1 1 1 1	1 0 0 0 1 1	1	Shift	
0 0 0 1 1 1	1 1 0 0 0 1	1	Shift	Step 4
0 0 0 0 1 1	1 1 1 0 0 0	1	Shift	Step 5
1 0 0 1 0 0	1 1 1 0 0 0	1	A = A + M	Step 6
1 1 0 0 1 0	0 1 1 1 0 0	0	Shift	


ldA
clrA
A
sftA
Q
ldQ
clrQ
sftQ
Q_{-1}
qm1
clrff
Z
ALU
A
M
addsub
M
ldM
data_in

DATA PATH


DATA PATH
CONTROL PATH
ldA
clrA
sftA
ldQ
clrQ
sftQ
ldM
clrff
addsub
Q[0]
qm1
data_in
start
done
clk



CONTROL PATH

START

S0

$A = 0$; $Q_{-1} = 0$ count = 15; M = multiplicand;

S1

Q = multiplier;

S2

Q_0Q_{-1}

01

10

00 or 11

A = A + M

S3

A = A − M

S4

Shift Right (A, Q , Q_{-1}) count = count − 1;

S5

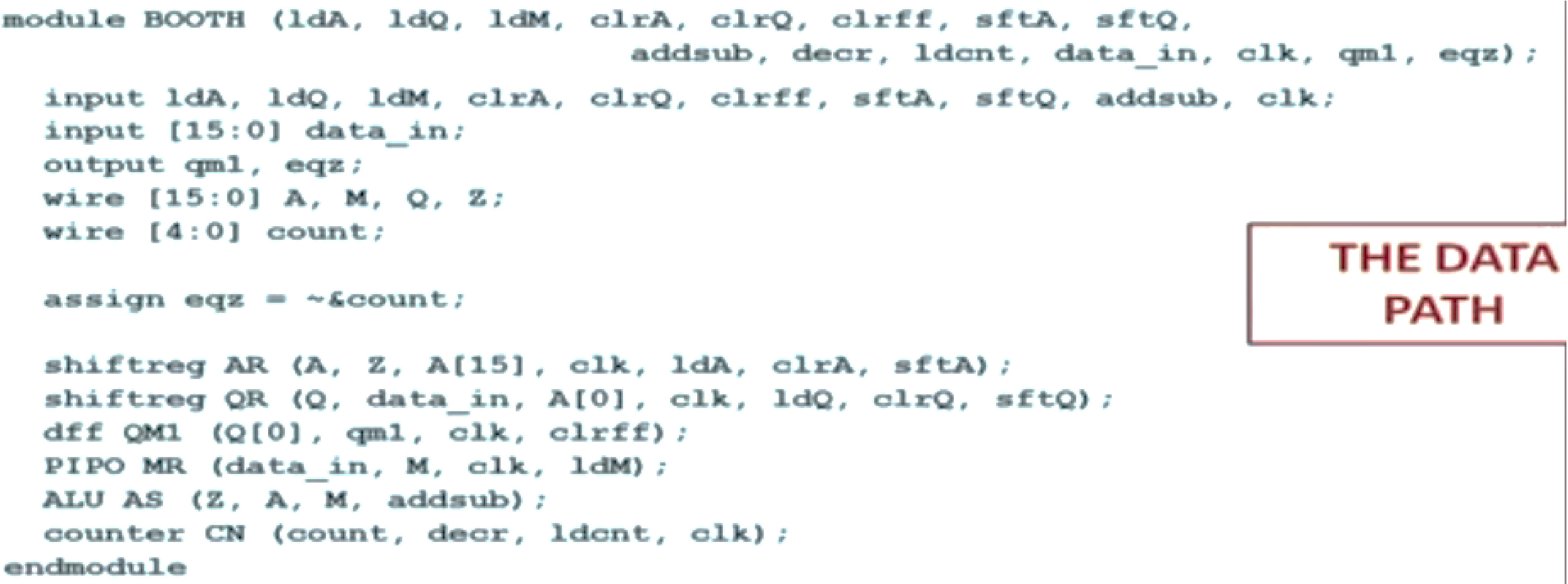
count?

≠0

=0

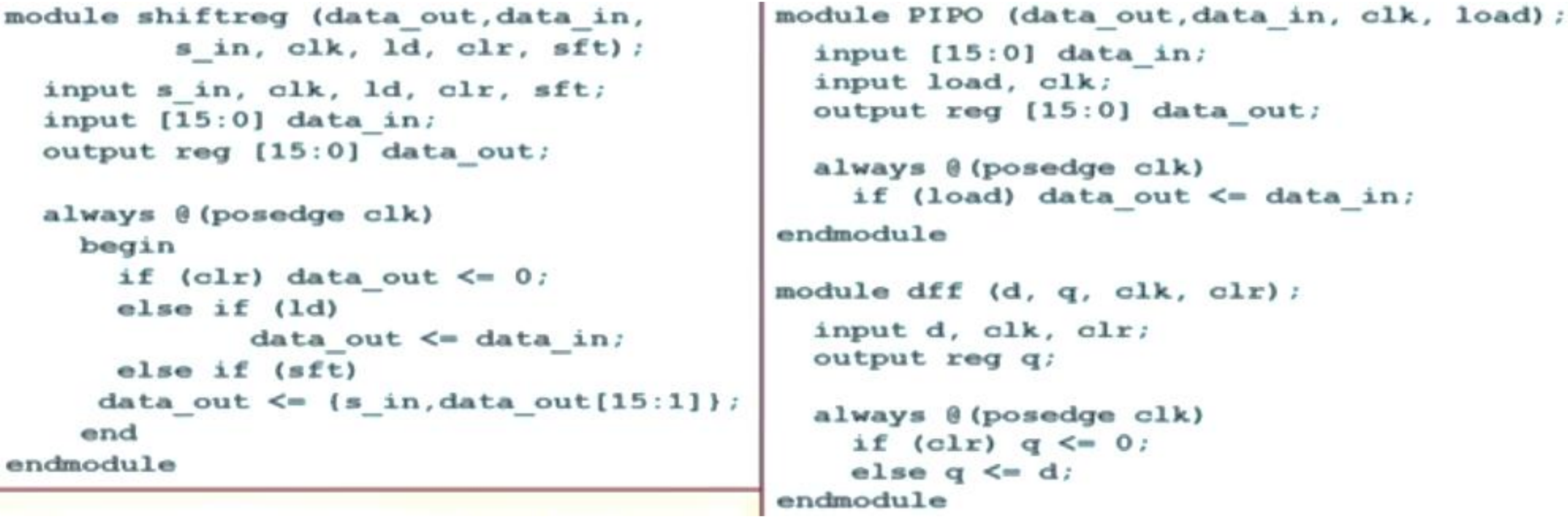
STOP

S6



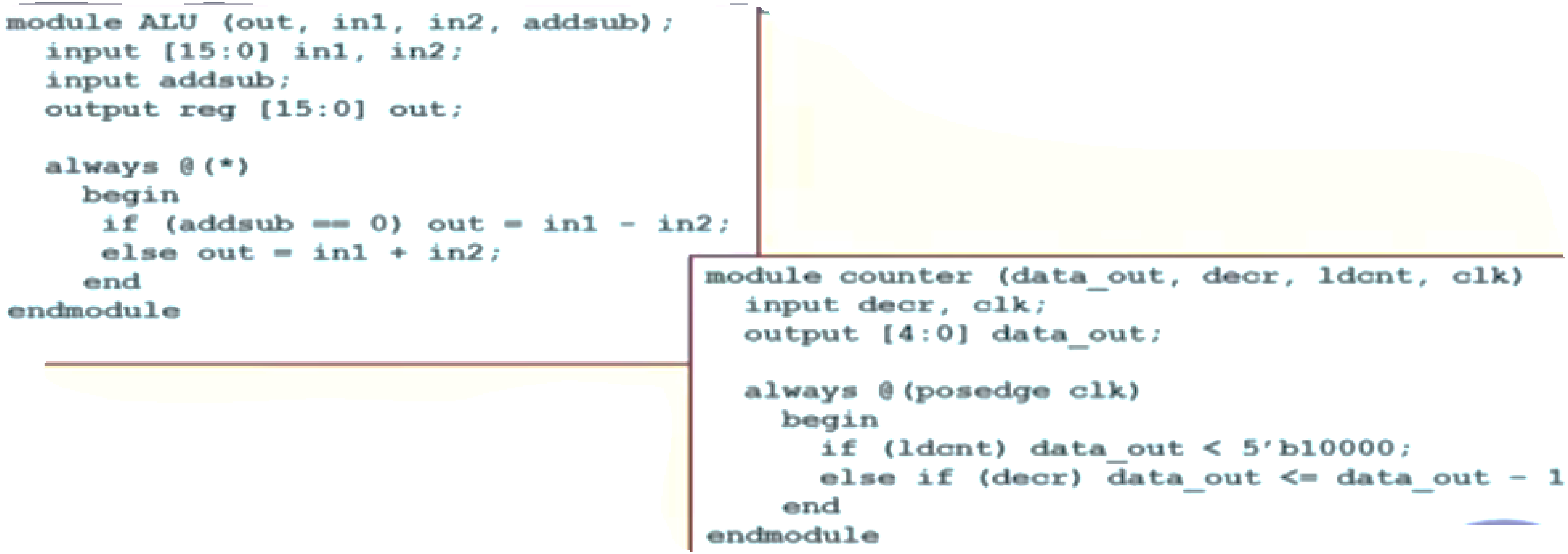
```
module BOOTH (ldA, ldQ, ldM, clrA, clrQ, clrff, sftA, sftQ,
                                addsub, decr, ldcnt, data_in, clk, qm1, eqz);
  input ldA, ldQ, ldM, clrA, clrQ, clrff, sftA, sftQ, addsub, clk;
  input [15:0] data_in;
  output qm1, eqz;
  wire [15:0] A, M, Q, Z;
  wire [4:0] count;

  assign eqz = ~&count;

  shiftreg AR (A, Z, A[15], clk, ldA, clrA, sftA);
  shiftreg QR (Q, data_in, A[0], clk, ldQ, clrQ, sftQ);
  dff QM1 (Q[0], qm1, clk, clrff);
  PIPO MR (data_in, M, clk, ldM);
  ALU AS (Z, A, M, addsub);
  counter CN (count, decr, ldcnt, clk);
endmodule
```

THE DATA PATH



```
module shiftreg (data_out,data_in,
         s_in, clk, ld, clr, sft);
  input s_in, clk, ld, clr, sft;
  input [15:0] data_in;
  output reg [15:0] data_out;

  always @(posedge clk)
    begin
      if (clr) data_out <= 0;
      else if (ld)
             data_out <= data_in;
      else if (sft)
     data_out <= {s_in,data_out[15:1]};
    end
endmodule
```

```
module PIPO (data_out,data_in, clk, load);
  input [15:0] data_in;
  input load, clk;
  output reg [15:0] data_out;

  always @(posedge clk)
    if (load) data_out <= data_in;
endmodule

module dff (d, q, clk, clr);
  input d, clk, clr;
  output reg q;

  always @(posedge clk)
    if (clr) q <= 0;
    else q <= d;
endmodule
```

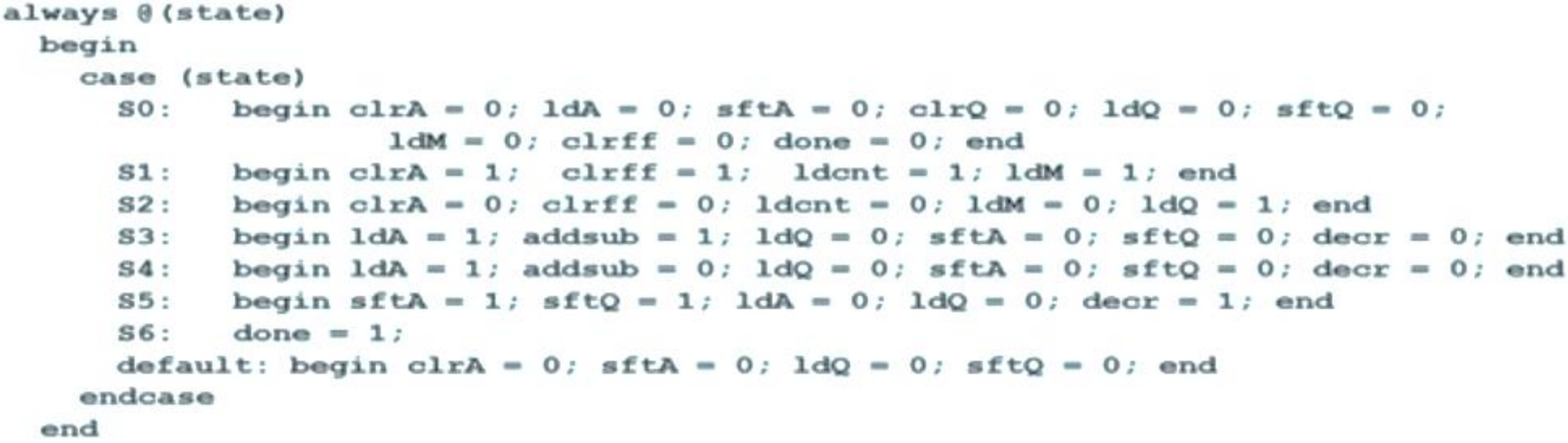


```
module ALU (out, in1, in2, addsub);
  input [15:0] in1, in2;
  input addsub;
  output reg [15:0] out;

  always @(*)
    begin
     if (addsub == 0) out = in1 - in2;
     else out = in1 + in2;
    end
endmodule
```

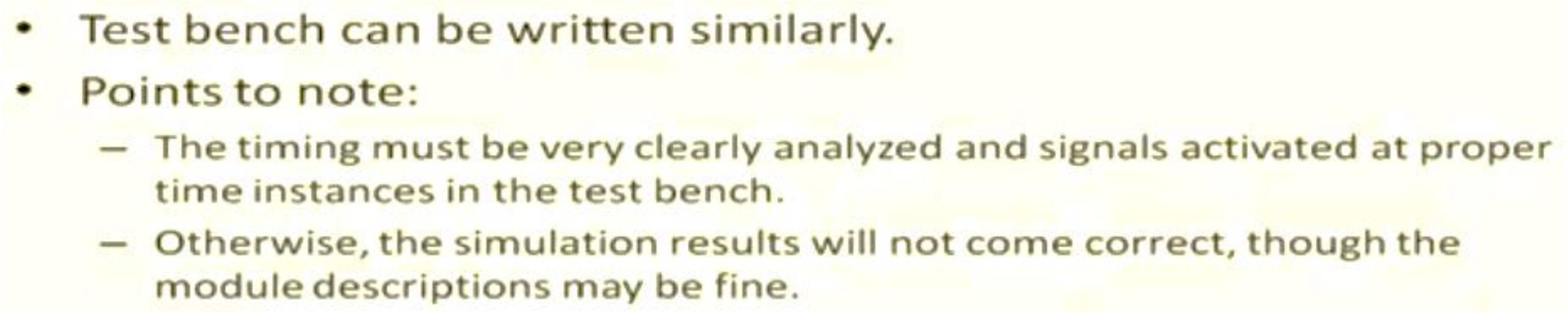
```
module counter (data_out, decr, ldcnt, clk)
  input decr, clk;
  output [4:0] data_out;

  always @(posedge clk)
    begin
      if (ldcnt) data_out < 5’b10000;
      else if (decr) data_out <= data_out - 1;
    end
endmodule
```

```
module controller (ldA, clrA, sftA, ldQ, clrQ, sftQ, ldM, clrff, addsub, start,
                            decr, ldcnt, done, clk, q0, qm1);
  input clk, q0, qm1, start;
  output reg ldA, clrA, sftA, ldQ, clrQ, sftQ, ldM, clrff, addsub,decr, ldcnt, done;
  reg [2:0] state;
  parameter S0=3'b000,S1=3'b001,S2=3'b010,S3=3'b011,S4=3'b100,S5=3'b101,S6=3’b110;
  always @(posedge clk)
    begin
      case (state)
        S0:    if (start) state <= S1;
        S1:    state <= S2;
        S2:    #2 if ({q0,qm1}==2’b01) state <= S3;
               else if ({q0,qm1}==1’b10) state <= S4;
               else state <= S5;
        S3:    state <= S5;
        S4:    state <= S5;
        S5:    #2 if (({q0,qm1}==2’b01) && !eqz) state <= S3;
               else if (({q0,qm1}==2’b10) && !eqz) state <= S4;
               else if (eqz) state <= S6;
        S6:    state <= S6;
        default: state <= S0;
      endcase
    end
```

THE CONTROL PATH



```
always @(state)
  begin
    case (state)
      S0:   begin clrA = 0; ldA = 0; sftA = 0; clrQ = 0; ldQ = 0; sftQ = 0;
                  ldM = 0; clrff = 0; done = 0; end
      S1:   begin clrA = 1;  clrff = 1;  ldcnt = 1; ldM = 1; end
      S2:   begin clrA = 0; clrff = 0; ldcnt = 0; ldM = 0; ldQ = 1; end
      S3:   begin ldA = 1; addsub = 1; ldQ = 0; sftA = 0; sftQ = 0; decr = 0; end
      S4:   begin ldA = 1; addsub = 0; ldQ = 0; sftA = 0; sftQ = 0; decr = 0; end
      S5:   begin sftA = 1; sftQ = 1; ldA = 0; ldQ = 0; decr = 1; end
      S6:   done = 1;
      default: begin clrA = 0; sftA = 0; ldQ = 0; sftQ = 0; end
    endcase
  end
```



- Test bench can be written similarly.
- Points to note:
  - The timing must be very clearly analyzed and signals activated at proper time instances in the test bench.
  - Otherwise, the simulation results will not come correct, though the module descriptions may be fine.