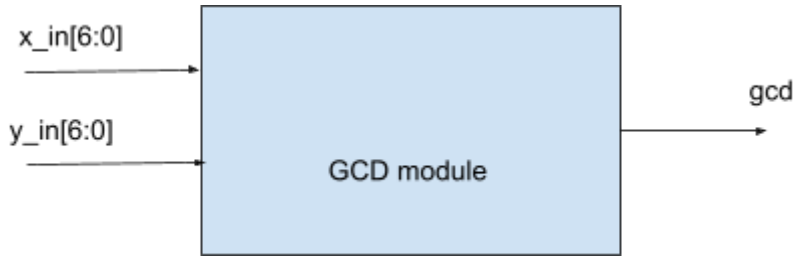
Lab 5

The task to be done in this lab: GCD code implementation on Vivado

GCD Explanation : GCD stands for Greater common divisor. Example : GCD(12,8), GCD(35,49) = 7

GCD calculation is done by subtracting larger numbers with smaller ones and replacing the larger number with the result of subtraction. Repeat this and stop when both the numbers become equal.



We would be designing a GCD module with 7-bit inputs.

One of the implementations could be thought of as the following:-

```
module gcd (
input wire [6:0] x_in ,
input wire [6:0] y_in ,
output reg gcd_o
);
reg [6:0] xs , ys;
always @(*)
    begin
      xs = x_in;
      ys = y_in;
      while(xs != ys)
        begin
            if (xs < ys)
                ys = ys - xs;
            else
                xs = xs - ys;
            end
        gcd_o = x_in;
    end
endmodule
```

But this, as discussed in the lab, won't be synthesized (won't work in hardware) since while loop is not synthesizable because of non-predetermined numbers of blocks that would be needed to synthesize it fully. While loop decides a number of iterations in runtime because of which this happens.

A solution to this is to use a for loop to do the same. This limits the dynamic range of the application that we build but for loop is synthesizable and while is not for the reason discussed.

```
module gcd (
input wire [6:0] x_in ,
input wire [6:0] y_in ,
output reg gcd_o
);
reg [6:0] xs , ys;
integer k;
always @(*)
    begin
      xs = x_in;
      ys = y_in;
      for (k=1;k<=3;k=k+1)
        begin
        if (xs < ys & xs != ys)
            ys = ys - xs;
        else if (xs != ys)
            xs = xs - ys;
        end
        gcd_o = xs;
    end
endmodule
```

Now the tool knows what the number of iterations that would be done here is. This stops after the three iterations, and hence we do have three copies of the hardware for the for loop. This has **unwrapped the for a loop.** We can notice that the dynamic range is limited since we have only three iterations available.

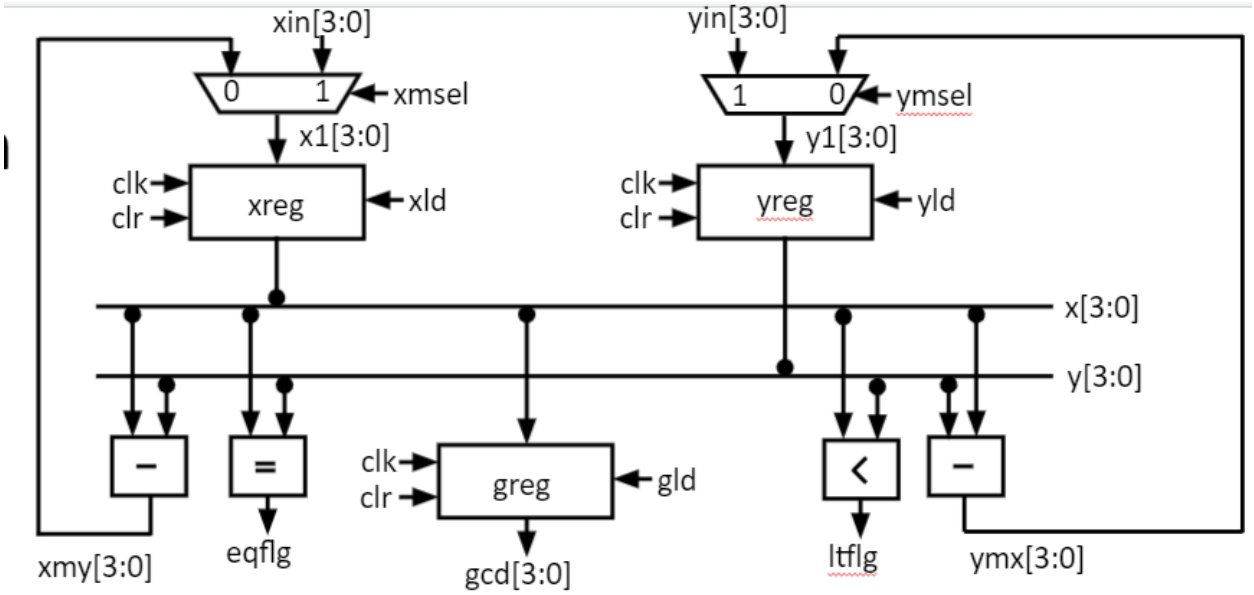
The solution for this is that we could also implement this in the form of a sequential circuit.

<u>But, the approach that we are hoing to take is to slit the solution between Data Path and Control Path</u>

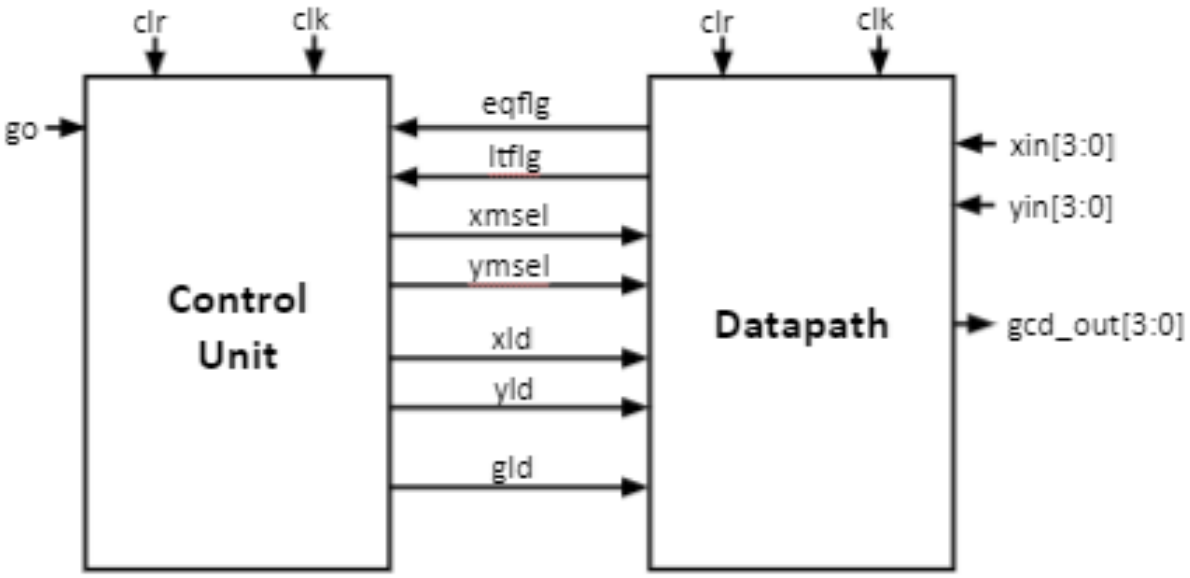
Data Path: Data Processing and operations are done here

Control Path: Gives the control logic signals and makes the Data Path do the tasks needed. The control path is the FSM.

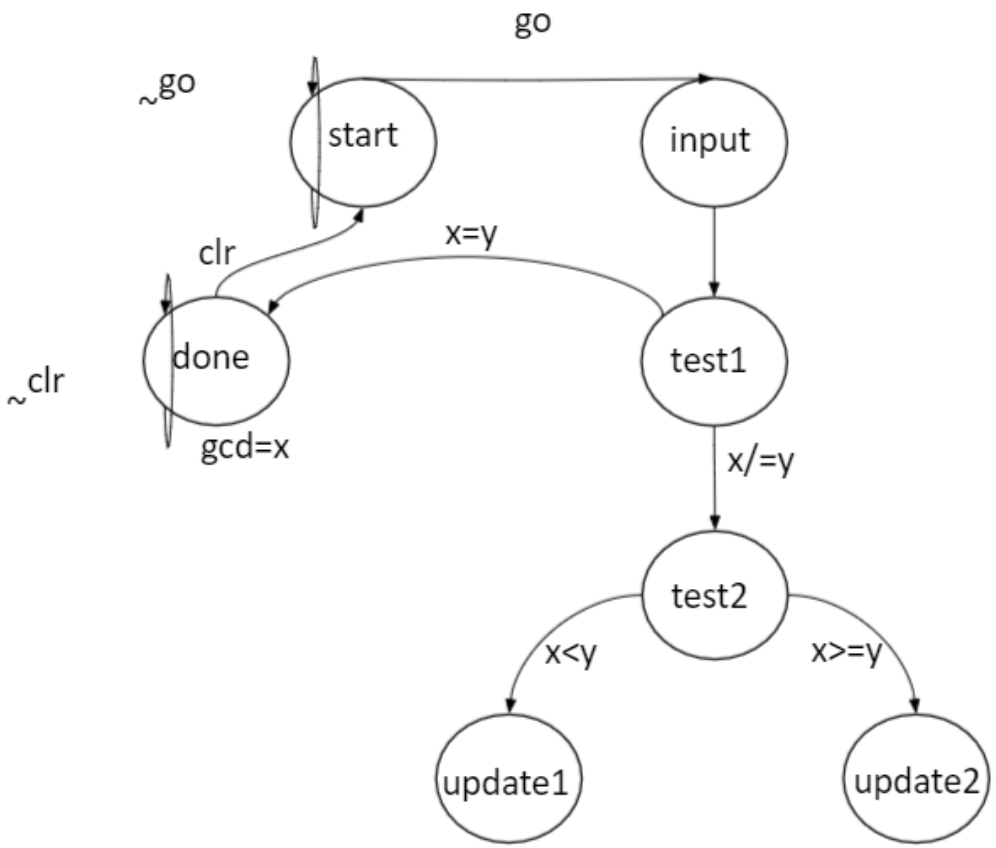
1. We have the registers for both inputs and outputs in the code. They are declared as xreg, yreg and greg. The former two are 3 bits wide, and the latter are one bit wide.
2. Connection of the register is given to the logical and arithmetic operations.
3. Following is the block diagram of the code that we are going to implement in Verilog.



4. To decide the values of xld, yld, clk, clr and gld, we will also make a control unit.



5. The above figure shows the overall control and data transfer
    a. When go is one then data xin and yin will be loaded into the data path
    b. If x=y then the controller makes the gld = 1 and x passed to the output
    c. if x<y then y = y-x will be loaded to the yreg and the other way around for the condition of x>y
6. Following is the FSM that we will be implementing for the GCD controller block:-



Creating the project :

1. We will first add the gcd.v file to the design source, and this works as the top module in the project.

```
`timescale 1ns / 1ps
module gcd (

input wire clk ,
input wire clr ,
input wire go ,
input wire [3:0] xin ,
input wire [3:0] yin ,
output wire [3:0] gcd_out
);

wire eqflg, ltflg, xmsel, ymsel;
wire xld, yld, gld;


gcd_datapath U1
    (.clk(clk),
    .clr(clr),
    .xmsel(xmsel),
    .ymsel(ymsel),
    .xld(xld),
    .yld(yld),
    .gld(gld),
    //data
    .xin(xin),
    .yin(yin),
    .gcd(gcd_out),
    // flags to determine the weather we obtained the GCD or shoukld we keep on going.
     .eqflg(eqflg),
     .ltflg(ltflg)
);

gcd_control U2
     (.clk(clk),
     .clr(clr),
     .go(go),
     // flags to determine the weather we obtained the GCD or shoukld we keep on going.
     .eqflg(eqflg),
     .ltflg(ltflg),
     //select line for loading data into reg through mux
     .xmsel(xmsel),
     .ymsel(ymsel),
     //load signals
     .xld(xld),
     .yld(yld),
     .gld(gld)
);

endmodule
```

2. Data path and Control path have been instantiated here, and we need to add files gcd_datapath.v and gcd_controlpath.v in the project.

gcd_controlpath.v

```
module gcd_control (
input wire clk ,
input wire clr ,
input wire go ,
input wire eqflg ,
input wire ltflg ,
output reg xmsel = 0 ,
output reg ymsel = 0,
output reg xld = 0,
output reg yld = 0,
output reg gld = 0
);

reg[2:0] present_state = 0, next_state;

parameter start = 3'b000, input1 =3'b001, test1 = 3'b010,
          test2 = 3'b011, update1 = 3'b100, update2 = 3'b101,
          done = 3'b110;  // states

// State registers
always @(posedge clk or posedge clr)
  begin
    if (clr == 1)
            present_state <= start;
    else
            present_state <= next_state;
  end

// C1 module
always @(*)
  begin
     case(present_state)
        start: if(go == 1)
                next_state = input1;
            else
                next_state = start;
        input1: next_state = test1;
        test1: if(eqflg == 1)
                next_state = done;
            else
                next_state = test2;
        test2: if(ltflg == 1)
                next_state = update1;
            else
                next_state = update2;
        update1: next_state = test1;
        update2: next_state = test1;
        done: next_state = done;
          default next_state = start;
     endcase
  end

 // C2 module
 always @(*)
```

```
    begin
      xld = 0; yld = 0; gld = 0;
      xmsel = 0; ymsel = 0;
      case(present_state)
         input1:
            begin xld = 1; yld = 1;
                  xmsel = 1; ymsel = 1;
            end
         update1: yld = 1;
         update2: xld = 1;
         done: gld = 1;
           default ;
      endcase
   end

endmodule
```

gcd_datapath.v

```
module gcd_datapath (

input wire clk ,
input wire clr ,
input wire xmsel ,
input wire ymsel ,
input wire xld ,
input wire yld ,
input wire gld ,
input wire [3:0] xin ,
input wire [3:0] yin ,
output wire [3:0] gcd ,
output reg eqflg = 0,
output reg ltflg = 0

);

wire [3:0] xmy, ymx, gcd_out;
wire [3:0] x, y, x1, y1;

assign xmy = x - y;
assign ymx = y - x;

always @(*)
    begin
        if(x == y)
            eqflg = 1;
```

```
        else
            eqflg = 0;
    end

always @(*)
    begin
        if(x < y)
            ltflg = 1;
        else
            ltflg = 0;
    end

mux2g #(
    .N(4))
M1 (.a(xmy),
    .b(xin),
    .s(xmsel),
    .y(x1)
);

    mux2g #(
    .N(4))
M2 (.a(ymx),
    .b(yin),
    .s(ymsel),
    .y(y1)
);

register #(
     .N(4))
R1 (.load(xld),
     .clk(clk),
     .clr(clr),
     .d(x1),
     .q(x)
);

register #(
     .N(4))
R2 (.load(yld),
     .clk(clk),
     .clr(clr),
     .d(y1),
     .q(y)
);

register #(
     .N(4))
R3 (.load(gld),
     .clk(clk),
     .clr(clr),
     .d(x),
     .q(gcd_out)
);

assign gcd = gcd_out;

endmodule
```

3. Notice that we have used some register and multiplexer modules in the datapath code. These are being used for the storage of the data and decision making, respectively.

Registers store the data, and multiplexers pass one of the inputs to output based upon the select line inputs.

4. Register code is the be implemented as follows.:-

```
2  module register
3  #(parameter N = 8)
4  (input wire load ,
5   input wire clk ,
6   input wire clr ,
7   input wire [N-1:0] d ,
8   output reg [N-1:0] q = 0
9  );
.0
.1 always @(posedge clk or posedge clr)
.2      if(clr == 1)
.3          q <= 0;
.4      else if(load == 1)
.5          q <= d;
.6
.7 endmodule
.8
```

5. Mux in verilog would be implemented as follows:-

```
module mux2g
#(parameter N = 4)
(input wire [N-1:0] a,
 input wire [N-1:0] b,
 input wire s,
 output reg [N-1:0] y = 0
);

always @(*)
if(s == 0)
        y = a;
    else
        y = b;

endmodule
```

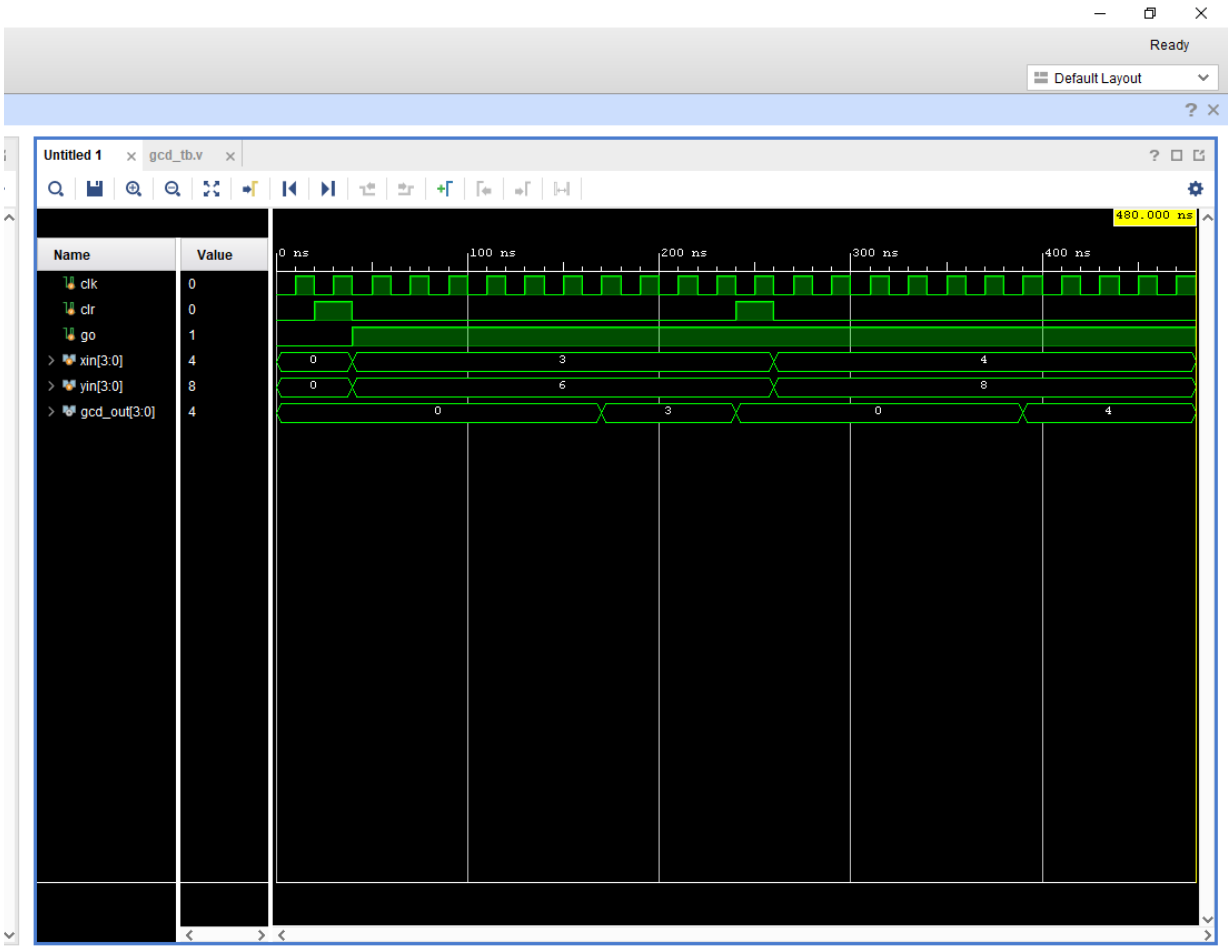
6. To check the functionality of the design implemented, We would need to design a testbench and confirm if we are getting the correct expected output or not.

```
22
23  module gcd_tb(
24      );
25
26  reg clk, clr, go;
27  reg [3:0] xin, yin;
28  wire [3:0] gcd_out;
29
30  //wire eqflg, ltflg, xmsel, ymsel;
31  //wire xld, yld, gld;
32
33  gcd U4(.clk(clk),
34      .clr(clr),
35      .go(go),
36      .xin(xin),
37      .yin(yin),
38      .gcd_out(gcd_out)//,
39  //  .eqflg(eqflg),
40  //  .ltflg(ltflg),
41  //  .xmsel(xmsel),
42  //  .ymsel(ymsel),
43  //  .xld(xld),
44  //  .yld(yld),
45  //  .gld(gld)
46  );
47
48  initial begin
49  clk = 1'b0;
```

```
clr = 1'b0;
go = 1'b0;
xin = 4'b0000;
yin = 4'b0000;
end

always #10 clk =~clk;


initial begin
@(negedge clk);
clr = 1;

@(negedge clk);
clr=1'b0;
go = 1'b1;
xin = 4'b0011;
yin = 4'b0110;
@(negedge clk);
@(negedge clk);
@(negedge clk);
@(negedge clk);
@(negedge clk);
@(negedge clk);
@(negedge clk);
@(negedge clk);
@(negedge clk);
```

```
77  @(negedge clk);
78  clr=1'b1;
79  @(negedge clk);
80  clr=1'b0;
81  go = 1'b1;
82  xin = 4'b0100;
83  yin = 4'b1000;
84  @(negedge clk);
85  @(negedge clk);
86  @(negedge clk);
87  @(negedge clk);
88  @(negedge clk);
89  @(negedge clk);
90  @(negedge clk);
91  @(negedge clk);
92  @(negedge clk);
93  @(negedge clk);
94  @(negedge clk);
95  $finish;
96
97  end
98
99  endmodule
100
```

7. Following are the simulation waveforms:-


Ready
Default Layout
Untitled 1
gcd_tb.v
480.000 ns
Name
Value
clk
0
clr
0
go
1
xin[3:0]
4
yin[3:0]
8
gcd_out[3:0]
4
0 ns
100 ns
200 ns
300 ns
400 ns
0
3
4
0
6
8
0
3
0
4