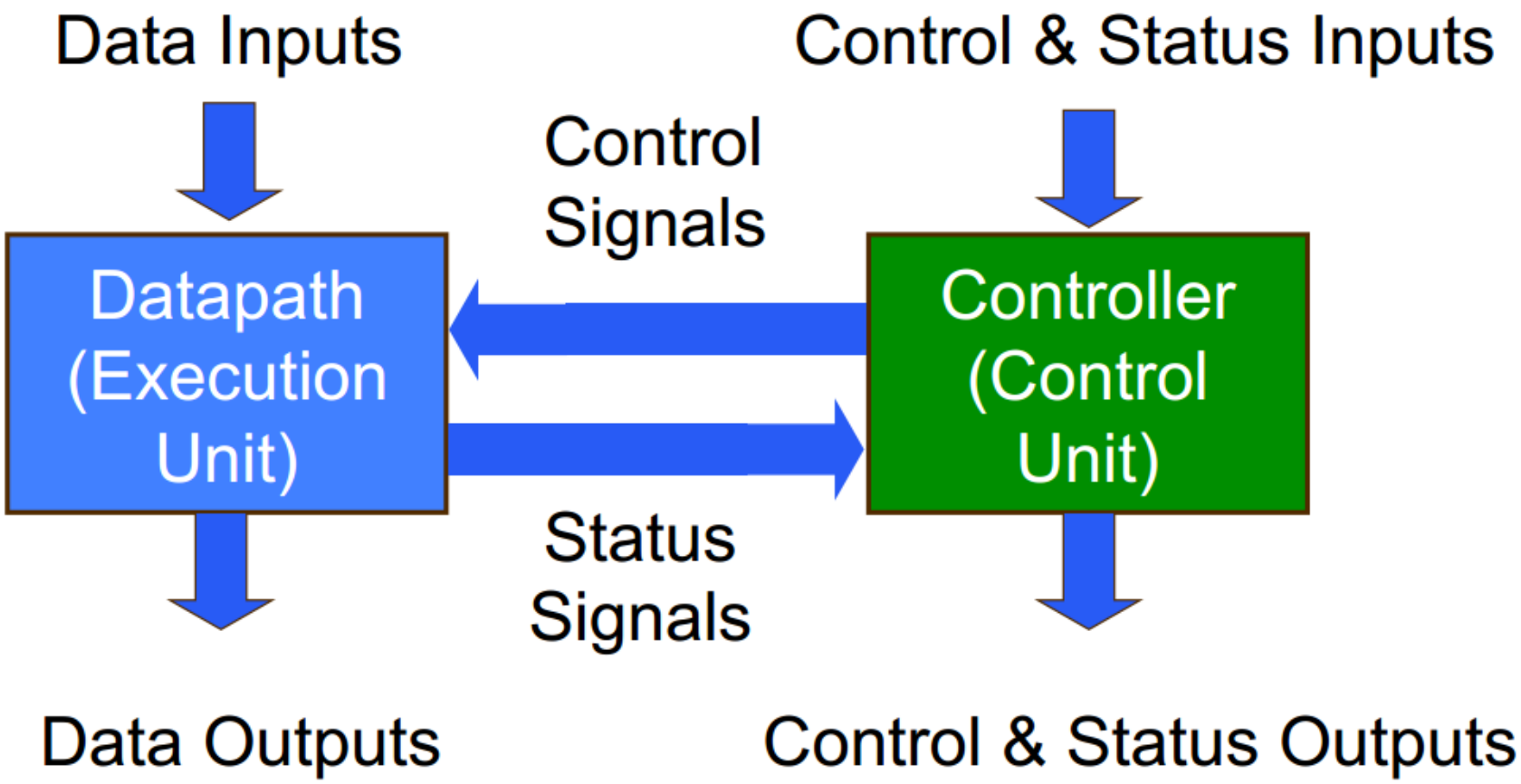
# ECE270: Embedded Logic Design (ELD)

- The material for this presentation is taken from various books, courses and Xilinx XUP resources. The instructor does not claim ownership of the material presented in this class.

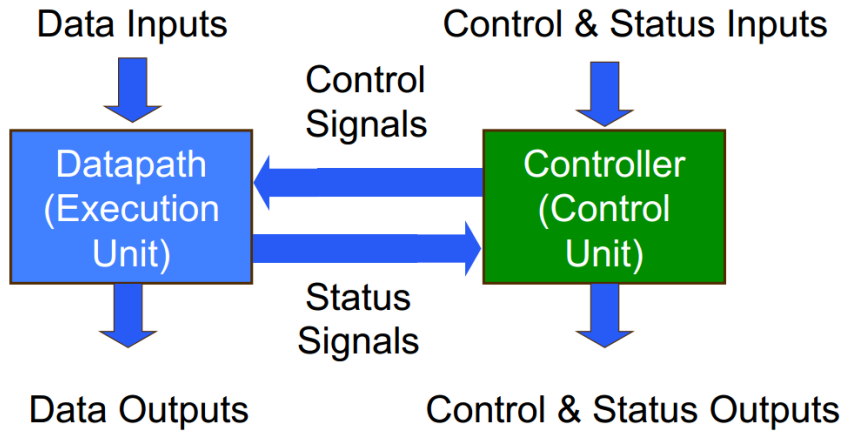
# Verilog: Greatest Common Divisor

```
module gcd (
input wire clk ,
input wire clr ,
input wire go ,
input wire [3:0] xin ,
input wire [3:0] yin ,
output reg done ,
output reg [3:0] gcd
);
reg [3:0] x, y;
reg calc;
always @(posedge clk or posedge clr)
  begin
    if(clr == 1)
      begin
        x <= 0;y <= 0;
        gcd <= 0;
        done <= 0;
        calc <= 0;
      end
    else
      begin
        done <= 0;
        if(go == 1)
          begin
            x <= xin;
            y <= yin;
            calc <= 1;
          end
        else
          begin
            if(calc == 1)
              if(x == y)
                begin
                  gcd <= x;
                  done <= 1;
                  calc <= 0;
                end
              else
                  if(x < y)
                      y <= y - x;
                  else
                      x <= x - y;
          end
      end
  end
endmodule
```

# Datapath and Controller



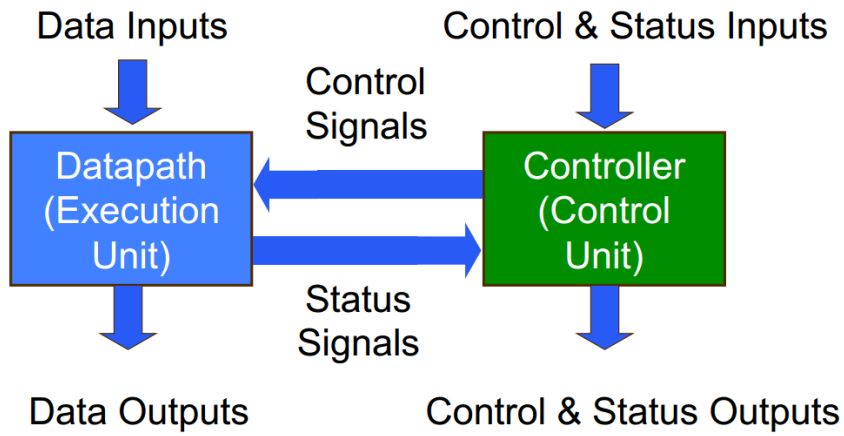
# Datapath




- Manipulates and processes data
- Performs arithmetic and logic operations, shifting/rotating, and other data-processing tasks
- Composed of registers, multiplexers, adders, decoders, comparators, ALUs, gates, etc.
- Provides all necessary resources and interconnects among them to perform specified task
- Interprets control signals from the Controller and generates status signals for the Controller

# Controller

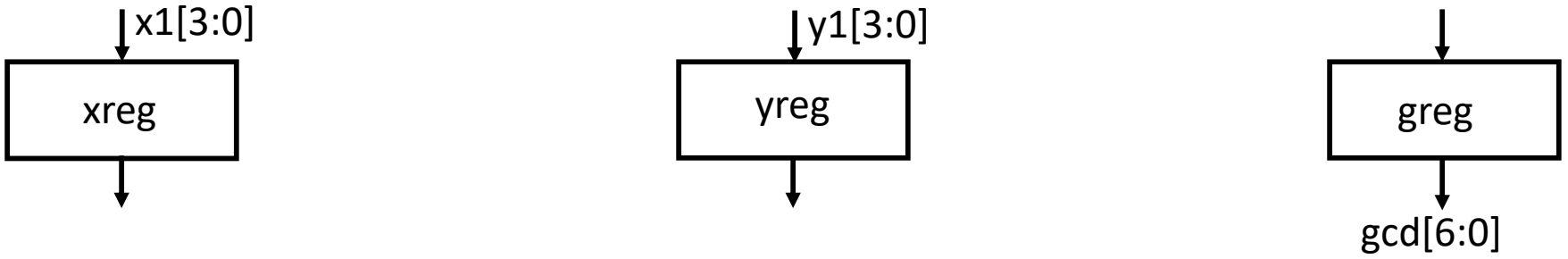
- Controls data movement in the Datapath by switching multiplexers and enabling or disabling resources: enable signal for registers, mux select
- Provides signals to activate various processing tasks in the Datapath
- Determines the sequence of operations performed by the Datapath



# GCD: Datapath

- The following steps can be used to create a datapath for implementing an algorithm:
  - Draw a register (rectangular block with input at the top and output at the bottom).
  - For example, for GCD algorithm, we need three registers for variables x, y and gcd.

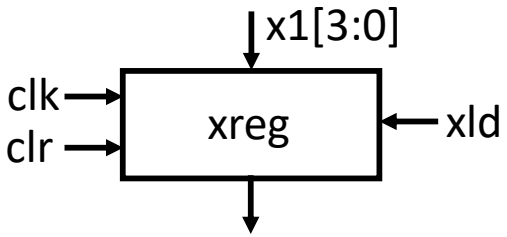
```
module gcd1 (
input wire [3:0] x ,
input wire [3:0] y ,
output reg [3:0] gcd
);
reg [3:0] xs , ys;

always @(*)
    begin
      xs = x;
      ys = y;
      while(xs != ys)
        begin
          if (xs < ys)
           ys = ys - xs;
        else
           xs = xs - ys;
        end
      gcd = xs;
    end

endmodule
```

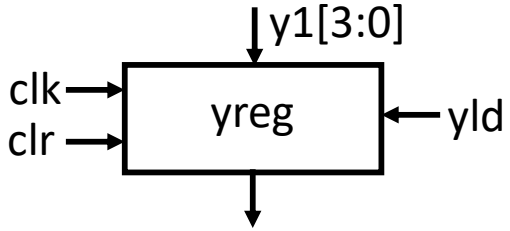


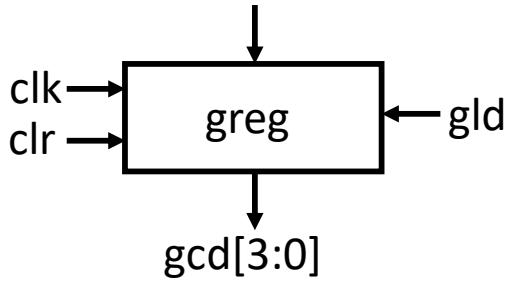
# GCD: Datapath

- The following steps can be used to create a datapath for implementing an algorithm:
  - Draw a register (rectangular block with input at the top and output at the bottom).
  - For example, for GCD algorithm, we need three registers for variables x, y and gcd.
  - Each register will also have clear, clock and load inputs. When clear is high, the output of the register will be a predetermined initial value. If load signal is high, then on the next rising edge of the clock, the input value will be loaded into the register and appears on the output.

# GCD: Datapath

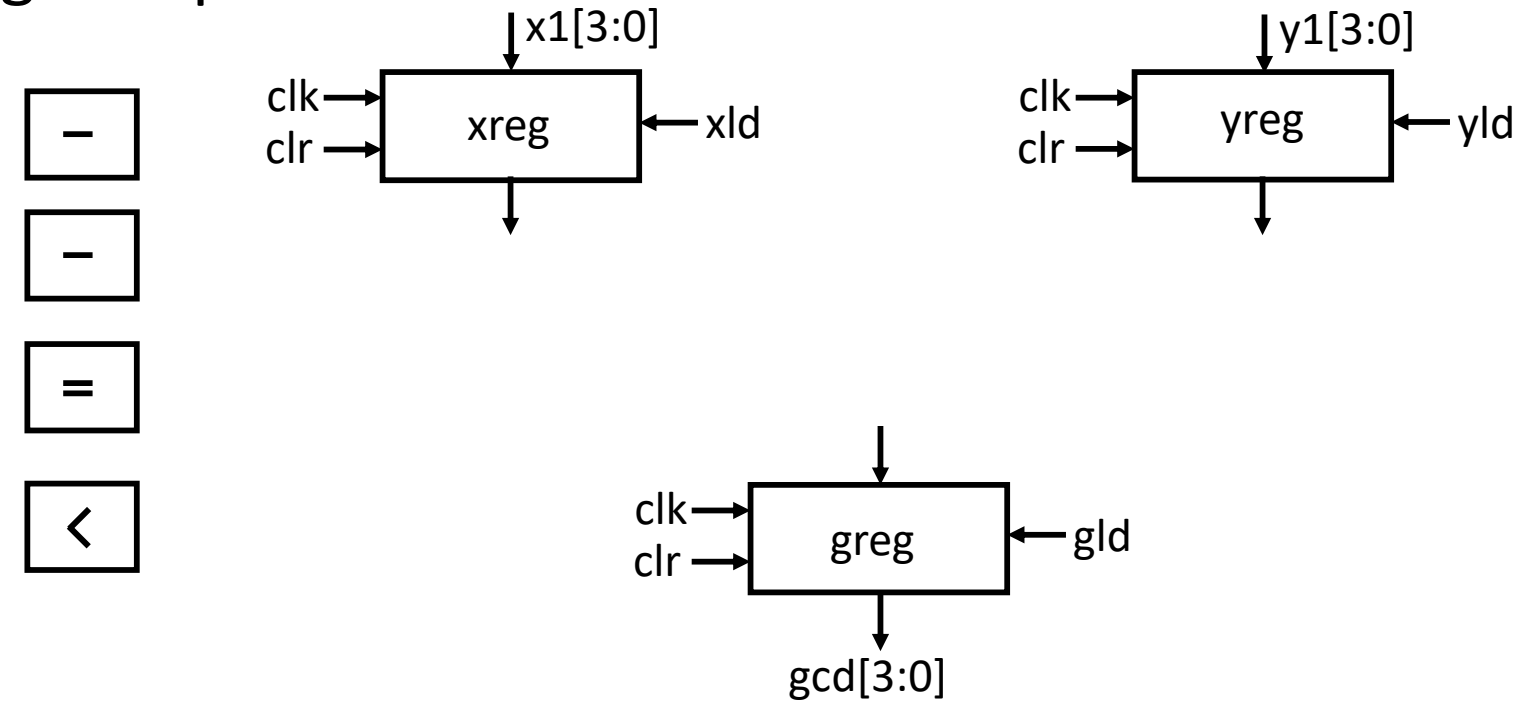


# GCD: Datapath

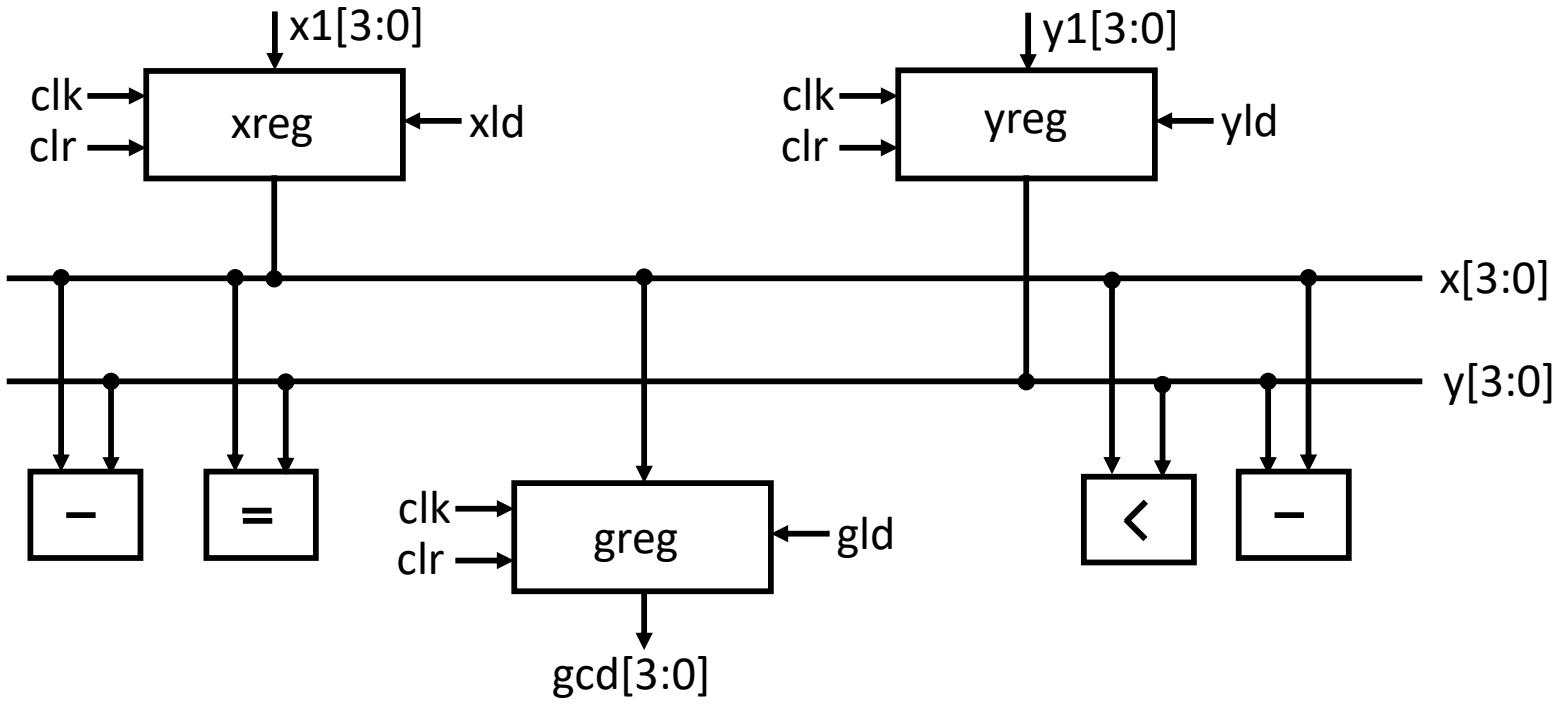
- Define combinational block to implement any necessary arithmetic or logical operation.




```
module gcd1 (
input wire [3:0] x ,
input wire [3:0] y ,
output reg [3:0] gcd
);
reg [3:0] xs , ys;

always @(*)
    begin
      xs = x;
      ys = y;
      while(xs != ys)
        begin
          if (xs < ys)
            ys = ys - xs;
          else
            xs = xs - ys;
        end
      gcd = xs;
    end

endmodule
```

# GCD: Datapath

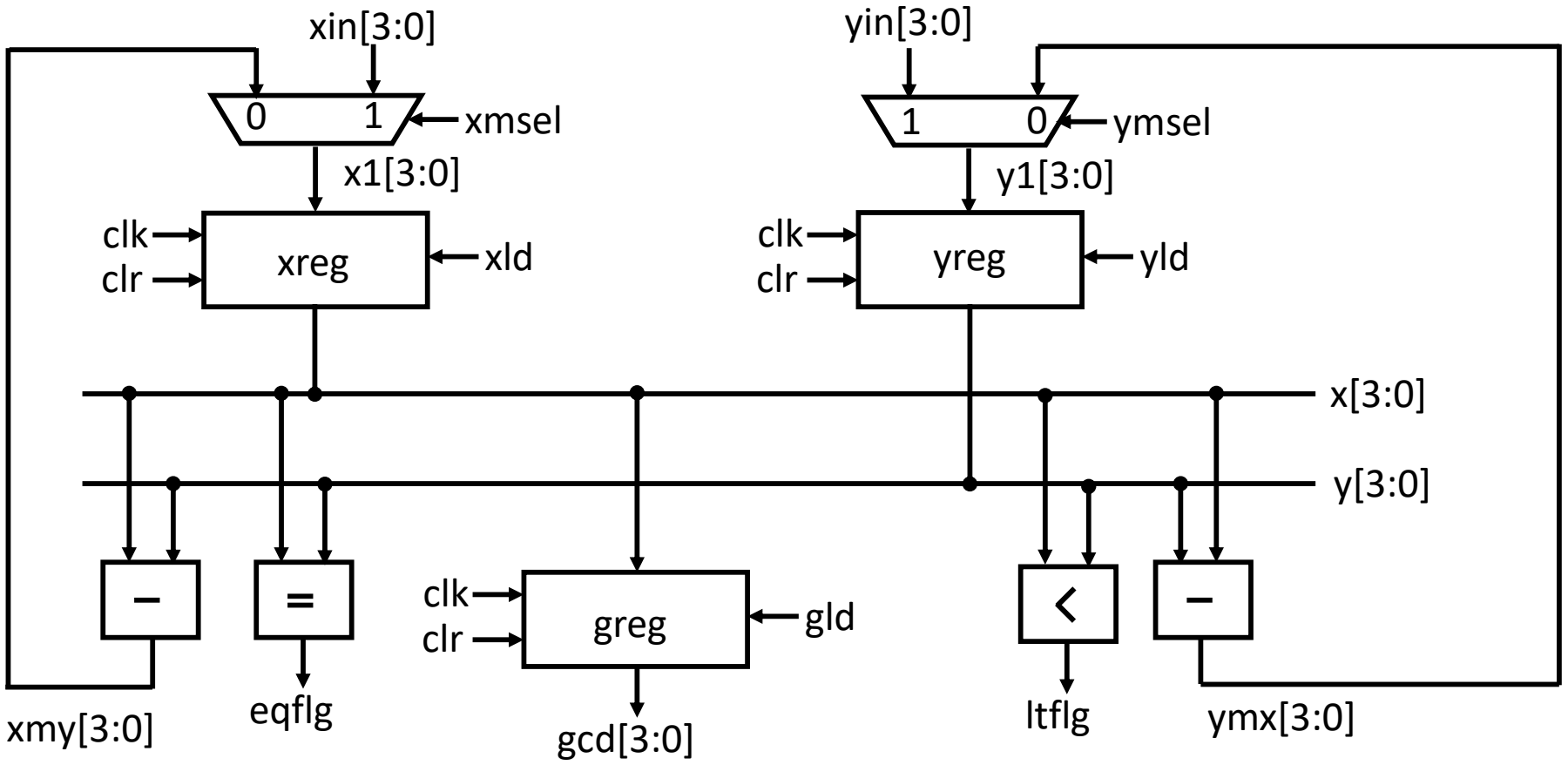
- Connect the output of registers to the inputs of the appropriate arithmetic and logical operations.




```
module gcd1 (
input wire [3:0] x ,
input wire [3:0] y ,
output reg [3:0] gcd
);
reg [3:0] xs , ys;

always @(*)
    begin
      xs = x;
      ys = y;
      while(xs != ys)
        begin
          if (xs < ys)
            ys = ys - xs;
        else
            xs = xs - ys;
        end
      gcd = xs;
    end

endmodule
```

# GCD: Datapath

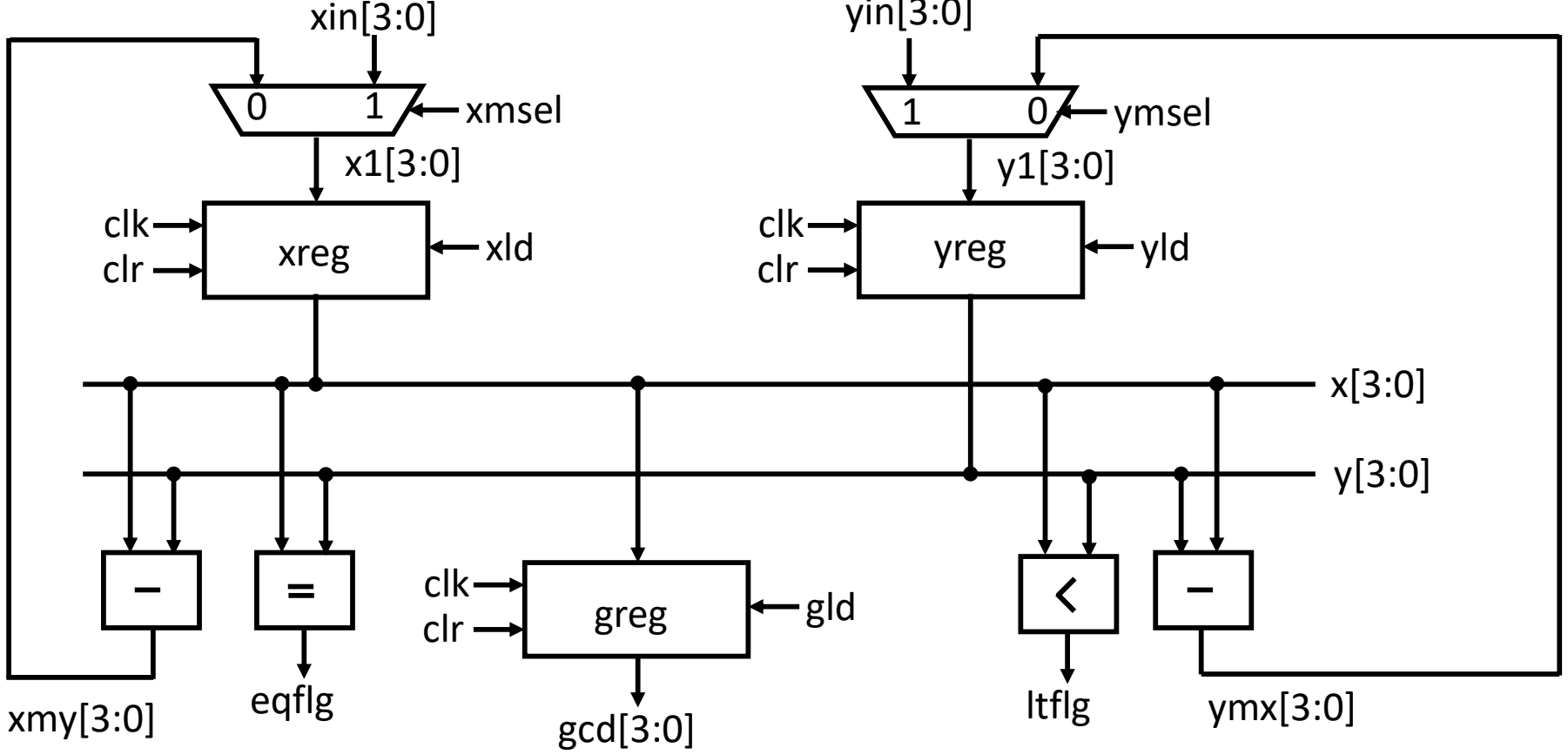
- Connect outputs of the arithmetic and logical operations to the appropriate registers. Multiplexers can be used if the input to register can come from more than one source.




```
module gcd1 (
input wire [3:0] x ,
input wire [3:0] y ,
output reg [3:0] gcd
);
reg [3:0] xs , ys;

always @(*)
    begin
      xs = x;
      ys = y;
      while(xs != ys)
        begin
          if (xs < ys)
            ys = ys - xs;
          else
            xs = xs - ys;
        end
      gcd = xs;
    end

endmodule
```

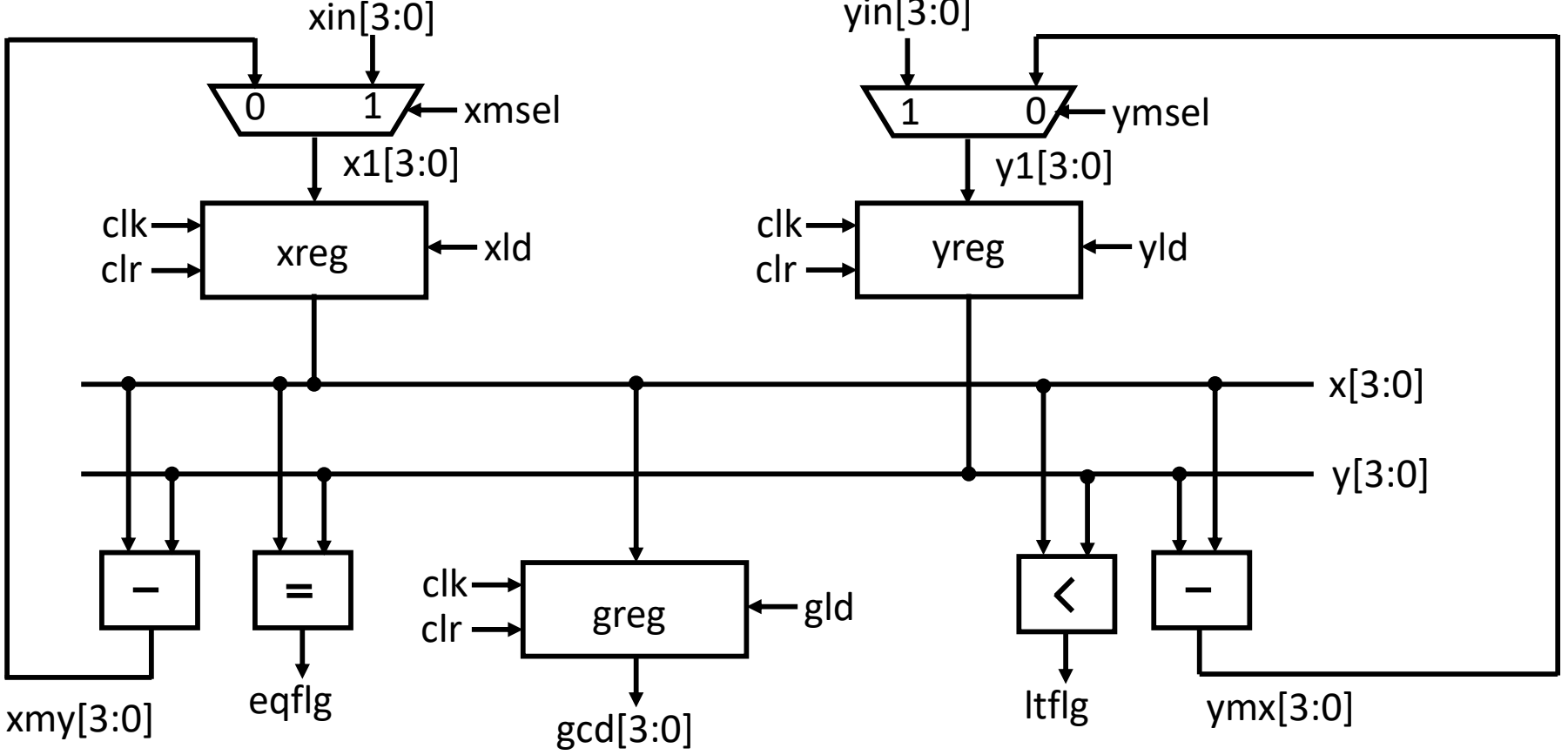
# GCD: Datapath

```
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 ,
output reg ltflg
);
```



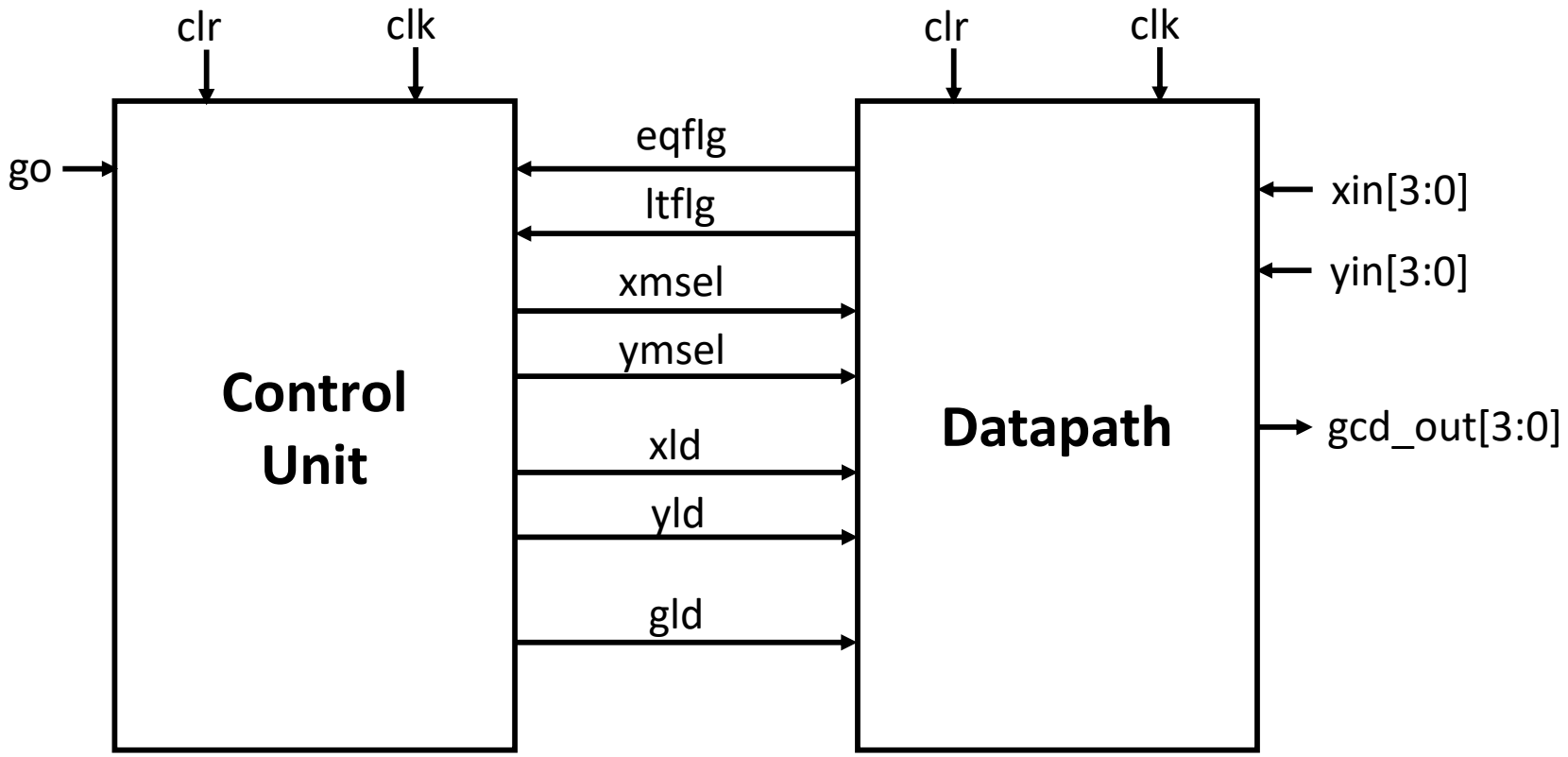
# GCD: Datapath

```
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
```



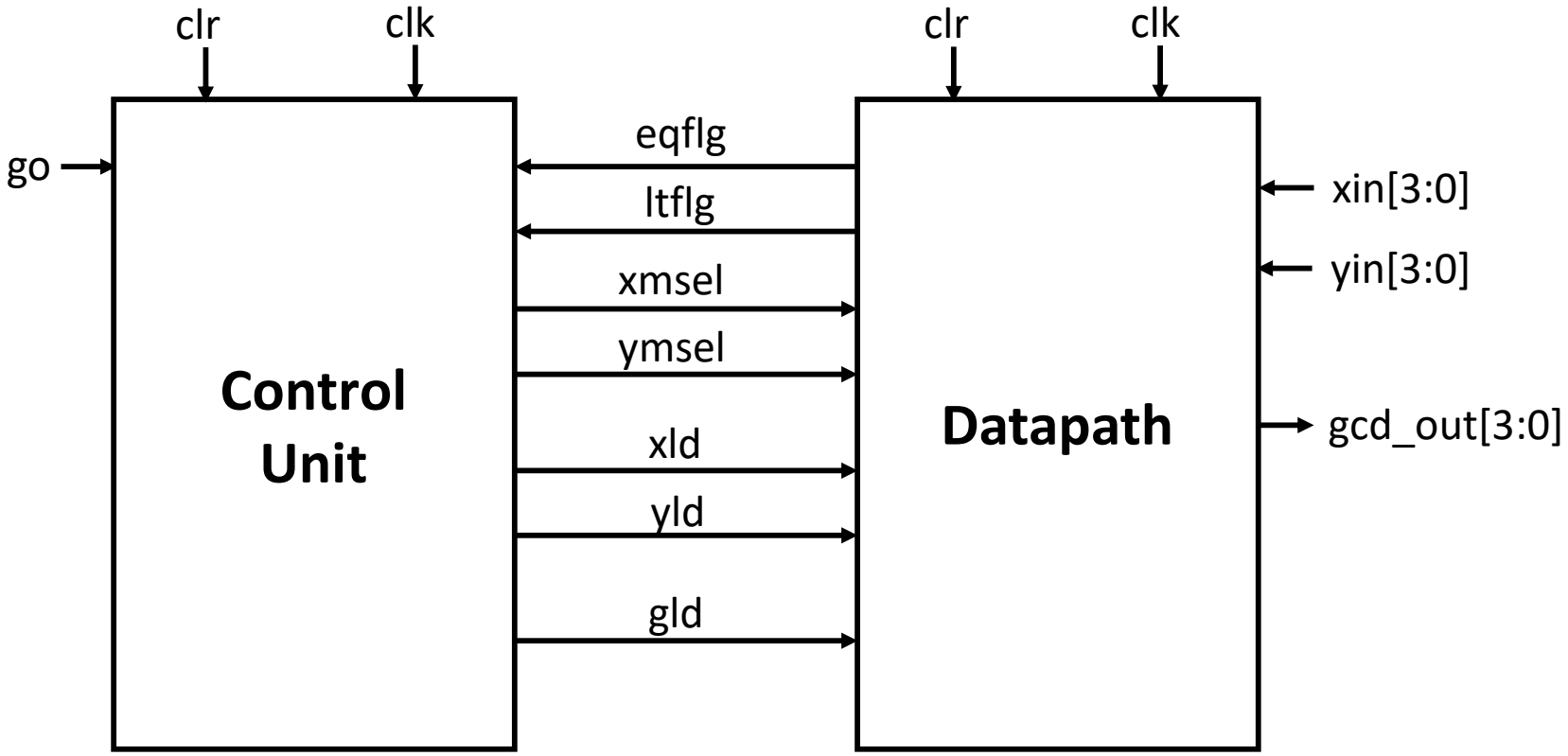

- For rest, you may write code for 4-bit 2:1 MUX and use it two time via module instantiation
- Same thing can be done for registers

# GCD: Datapath and Controller

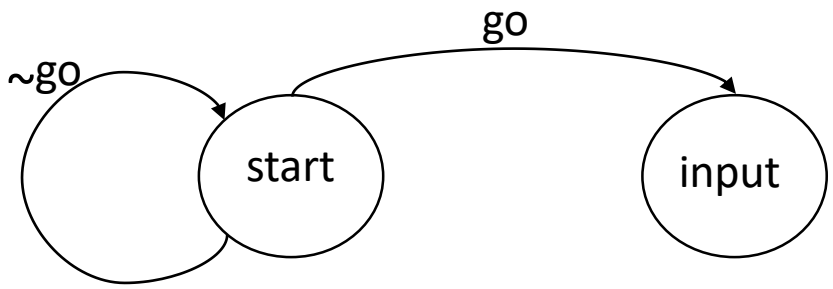


# GCD: Datapath and Controller

- Controller will be implemented using FSM.



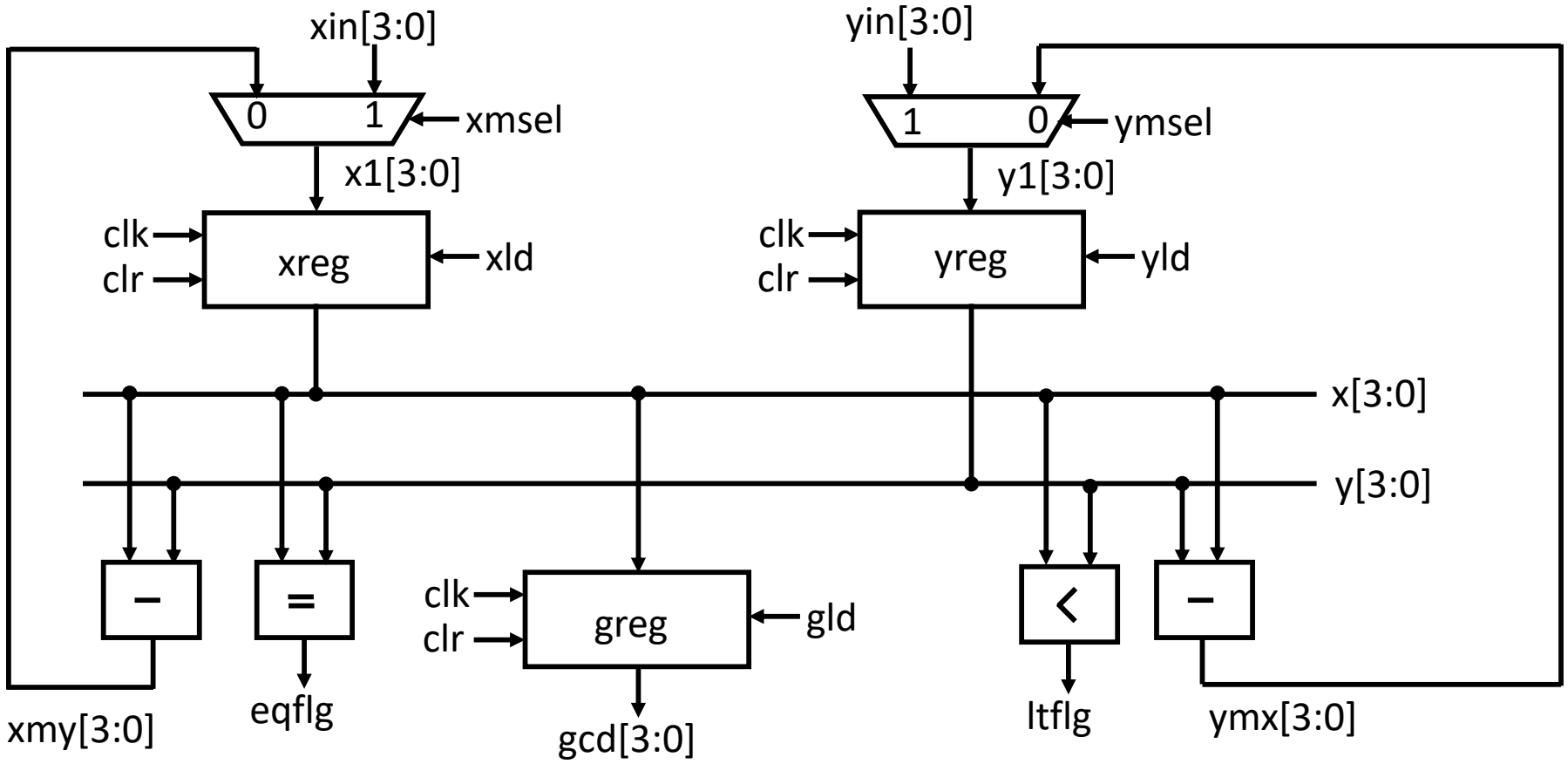
# GCD: Datapath and Controller




- Controller will be implemented using FSM.
- Algorithm will wait in the start state until the go signal goes high.
- In the input state, xmsel, ymsel, xld and yld will be set to 1 so that the two inputs, xin and yin, will be loaded into the registers, xreg and yreg, on the next rising edge of the clock.

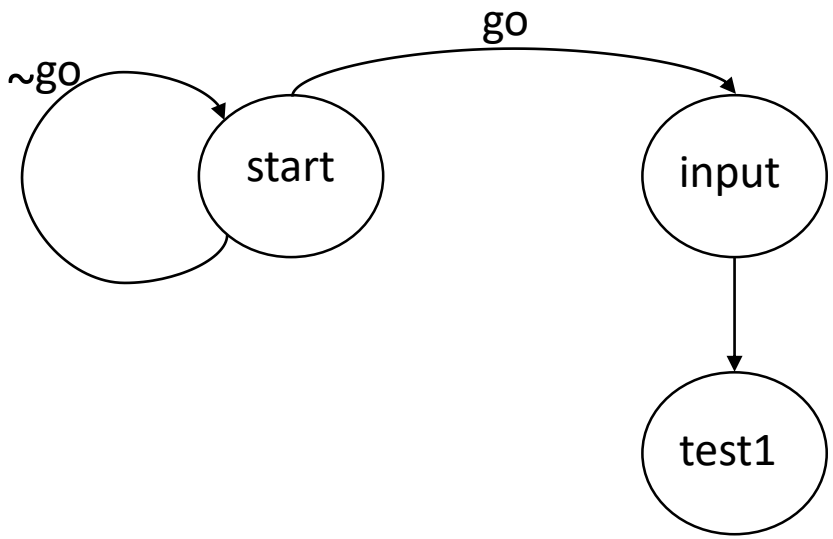
x=x-y

# GCD: Datapath



# GCD: Datapath and Controller

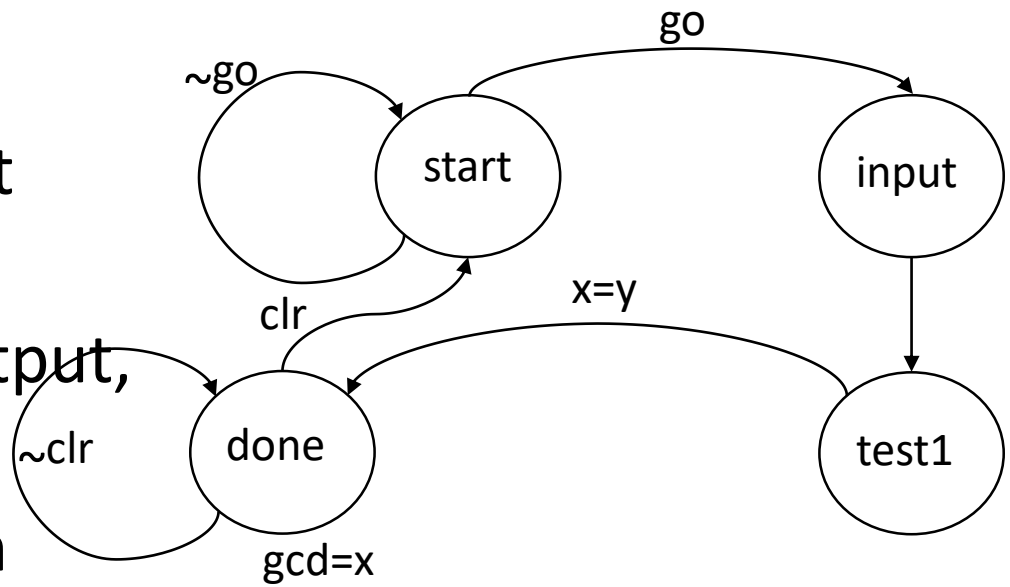
- The state diagram then moves to next state, test1.




x=x-y

# GCD: Datapath and Controller

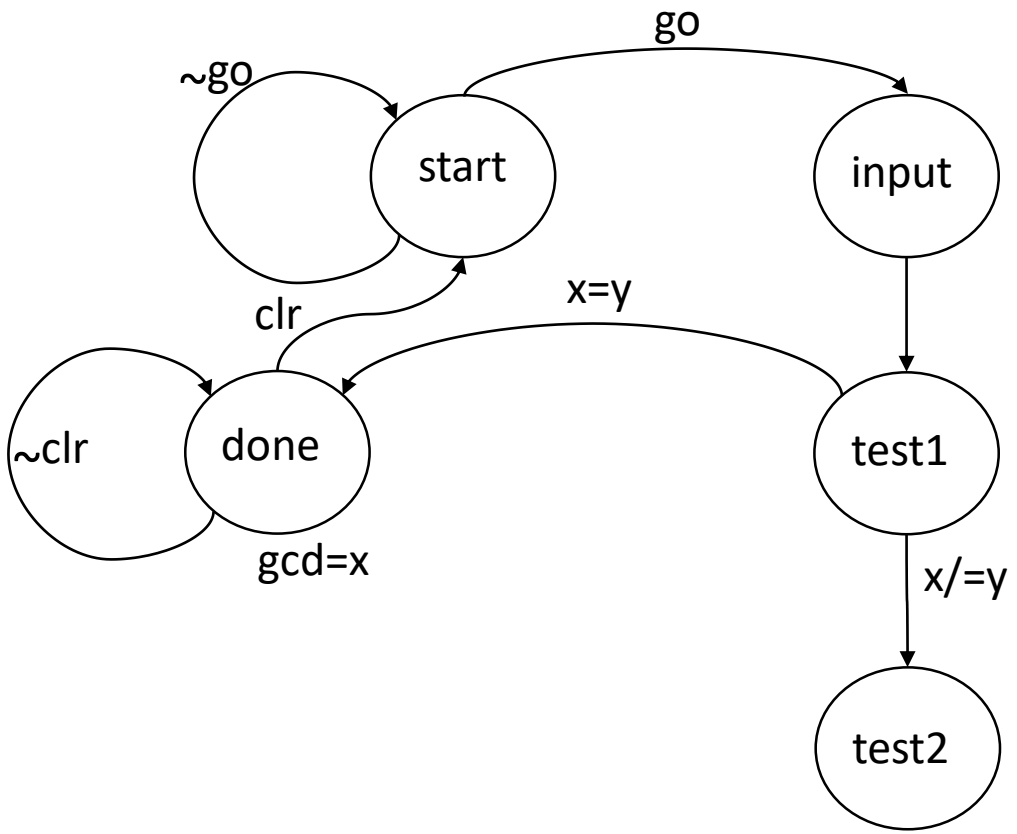
- The state diagram then moves to next state, test1.
- In this case, the value of datapath output, eqflag will determine the next state.
- If eqflag=1 that means that x=y which means that next state is done.
- When algorithm gets to the done state, it will stay there and continuously load the final value of x into the gcd output register by setting gld=1.




x=x-y

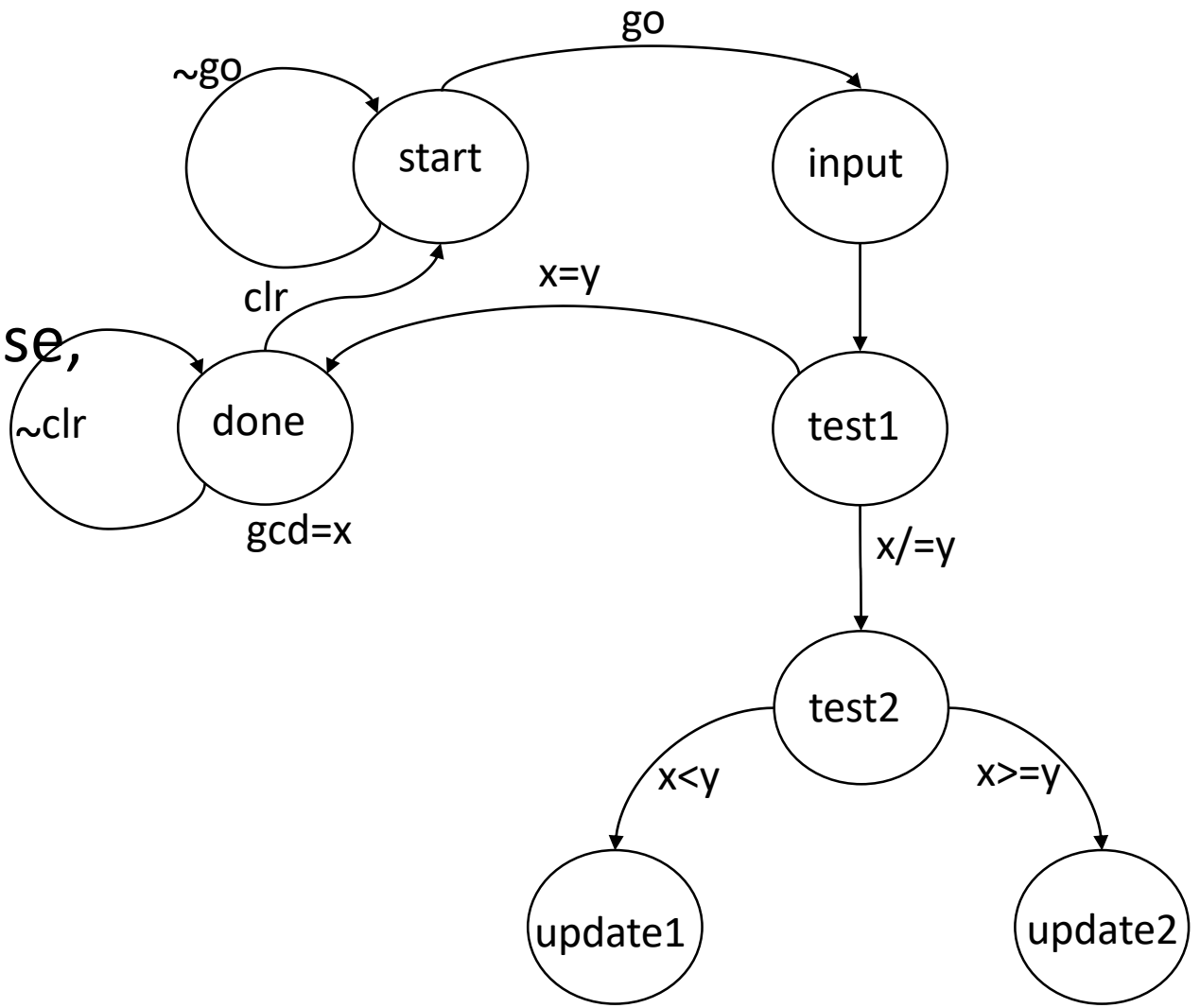
# GCD: Datapath and Controller

- If eqflag=0, then next state is test2.

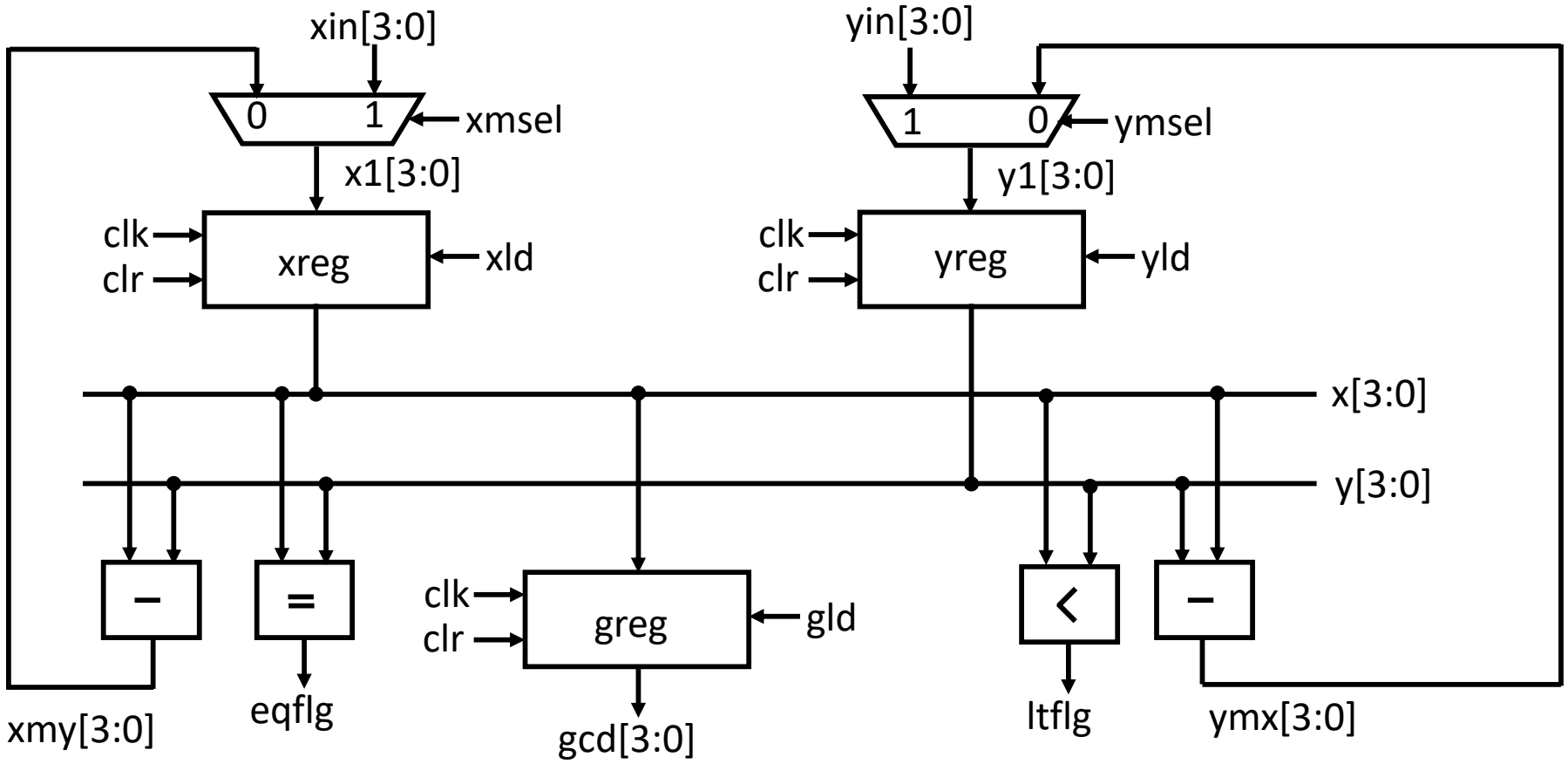


# GCD: Datapath and Controller

- If eqflag=0, then next state is test2.
- If ltflg=1 which means that x<y and hence, next state is update1. Otherwise, next state is update2.

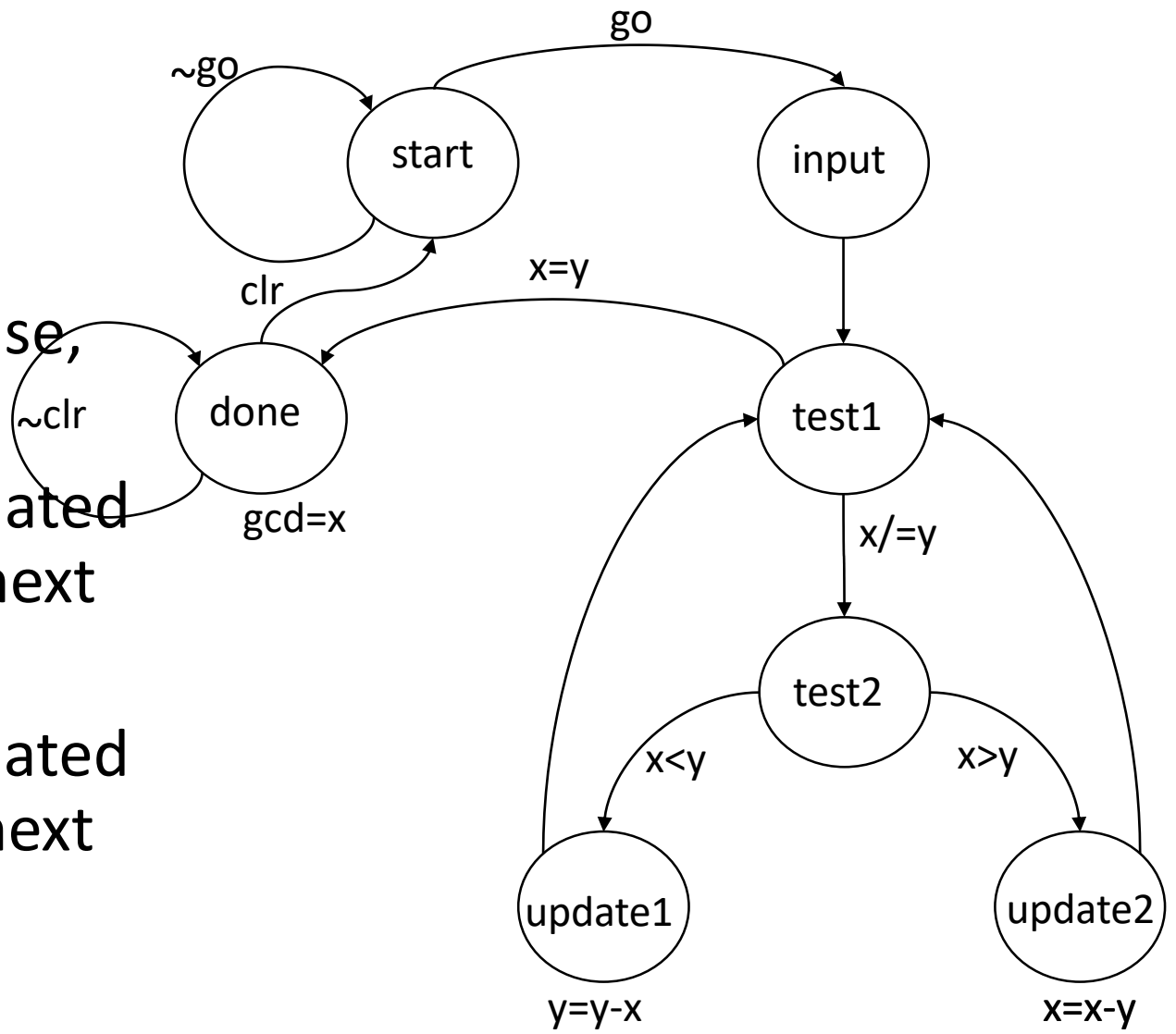


# GCD: Datapath

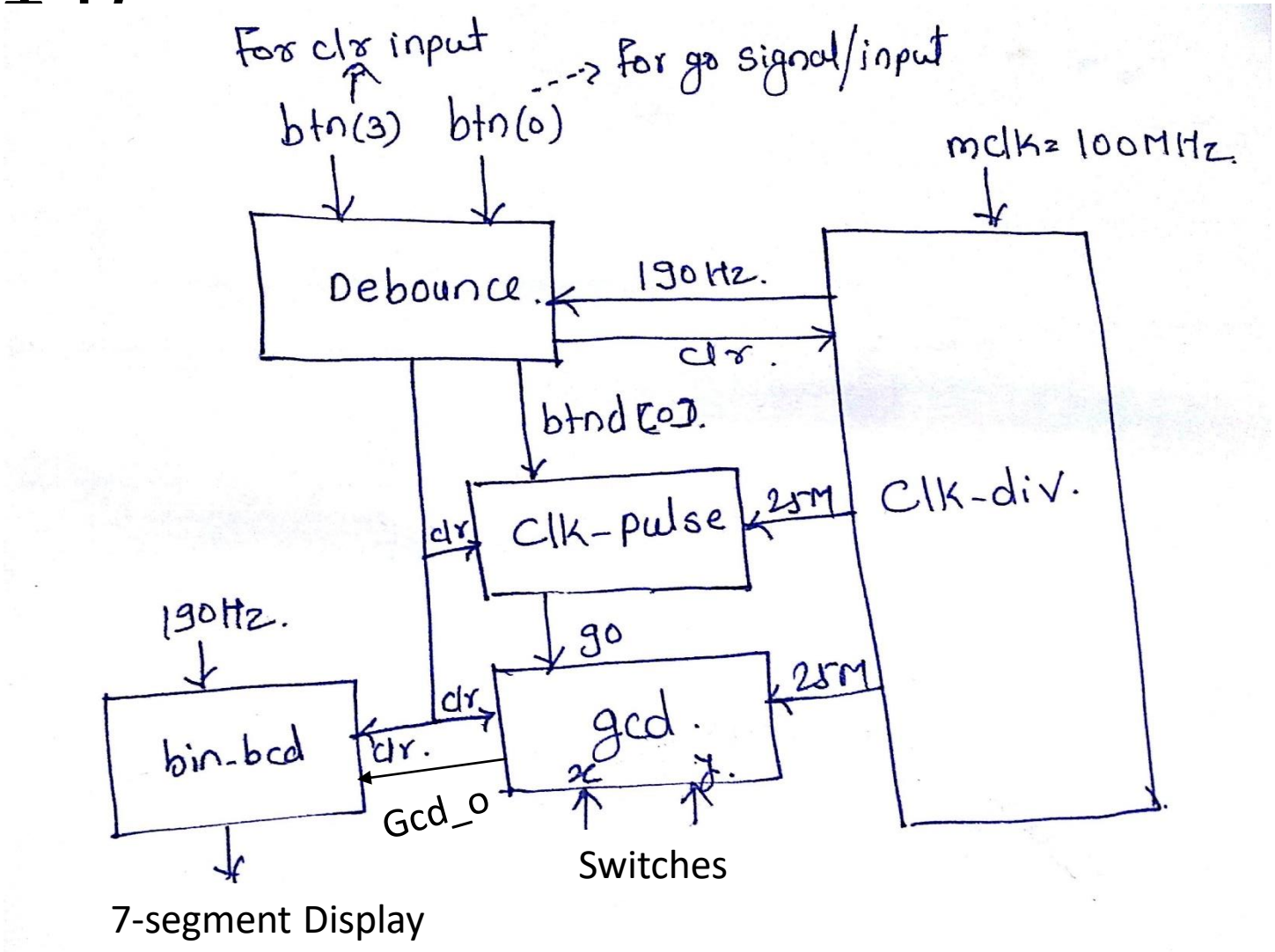


# GCD: Datapath and Controller

- If eqflag=0, then next state is test2.
- If ltflg=1 which means that x<y and hence, next state is update1. Otherwise, next state is update2.
- In state update1, the value of y is updated by setting yld=1 and ymsel=0. Then, next state is test1.
- In state update2, the value of x is updated by setting xld=1 and xmsel=0. Then, next state is test1.



# Lab 11 and 12: Demo Nov. 15 (Backpack: Nov. 14)



# Submission Requirements

- Write separate verilog code for control
- Write separate Verilog code for datapath
- Write top file (name it as gcd_top)
- Give the input via switches and display the results on 7-segment in BCD format (same as previous lab)
- Use 25 MHz clock for gcd and 190Hz clock for led-to-bcd.
- Validate the design on FPGA
- Include the option to read the inputs stored in block RAM (The switches will act as Block RAM address)

x=x-y

```
module isr(
    input [7:0] a,
    output reg [3:0] sqroot
    );

    reg [7:0] as,square,delta;

    always @(*)
        begin
           as = a;
          square = 1;
          delta = 3;
           while(square <= as)
            begin
              square = square + delta;
              delta = delta + 2;
            end
          sqroot = {0,delta[7:1]}-1;
        end

endmodule
```

x=x-y

# Integer Square Root Algorithm

```
module isr(
    input [7:0] a,
    output reg [3:0] sqroot
    );

    reg [7:0] as,square,delta;

    always @(*)
        begin
           as = a;
          square = 1;
          delta = 3;
           while(square <= as)
            begin
              square = square + delta;
              delta = delta + 2;
            end
          sqroot = {0,delta[7:1]}-1;
        end

endmodule
```

| n | Square | Delta | (Delta/2)-1 |
|---|---|---|---|
| 0 | 0 | | |
| 1 | 1 | 3 | |
| 2 | 4 | 5 | 1 |
| 3 | 9 | 7 | 2 |
| 4 | 16 | 9 | 3 |
| 5 | 25 | 11 | 4 |
| 6 | 36 | 13 | 5 |
| 7 | 49 | 15 | 6 |
| 8 | 64 | 17 | 7 |
| 9 | 81 | 19 | 8 |
| 10 | 100 | 21 | 9 |
| 11 | 121 | 23 | 10 |
| 12 | 144 | 25 | 11 |

# Integer Square Root Algorithm

```
module isr(
    input [7:0] a,
    output reg [3:0] sqroot
    );

    reg [7:0] as,square,delta;

    always @(*)
        begin
            as = a;
          square = 1;
          delta = 3;
           while(square <= as)
            begin
              square = square + delta;
              delta = delta + 2;
            end
          sqroot = {0,delta[7:1]}-1;
        end

endmodule
```