# EE671: VLSI DESIGN
# SPRING 2024/25

LAXMEESHA SOMAPPA

DEPARTMENT OF ELECTRICAL ENGINEERING

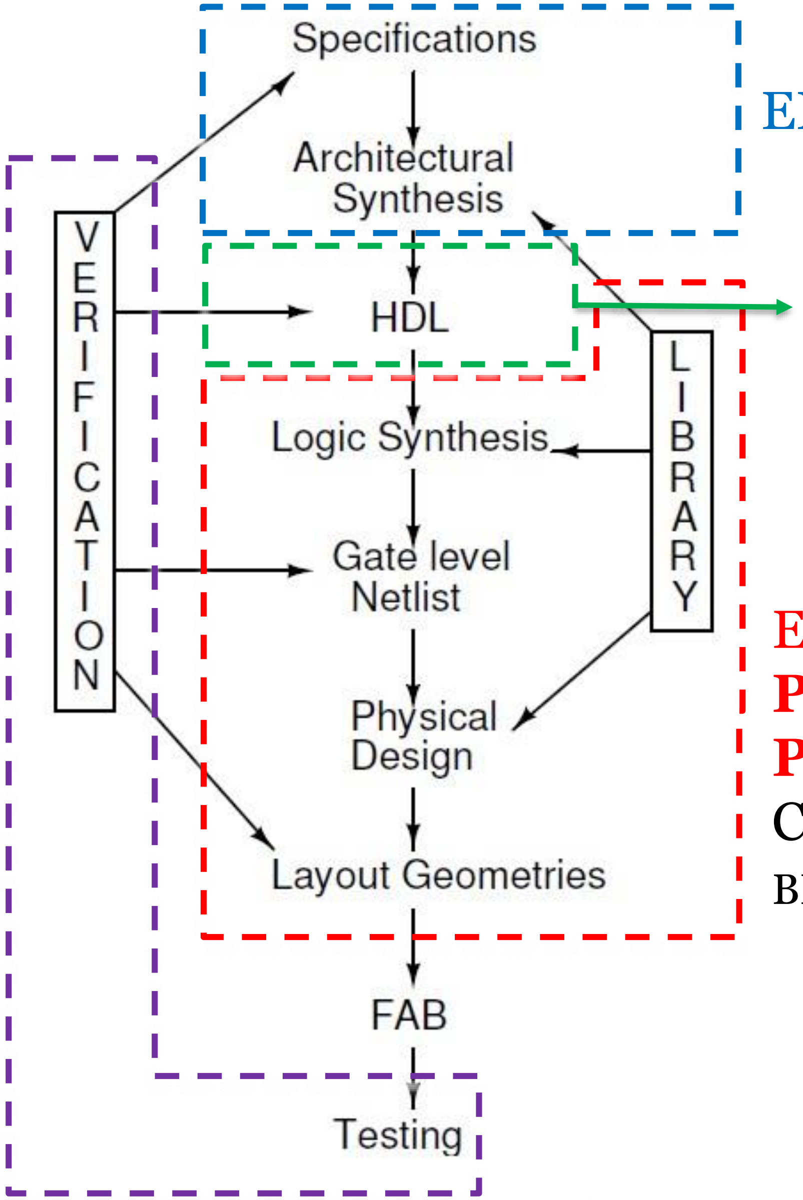
IIT BOMBAY

laxmeesha@ee.iitb.ac.in

# LECTURE – 19
# DIGITAL SYNTHESIS

laxmeesha@ee.iitb.ac.in

# What are we learning in this course?



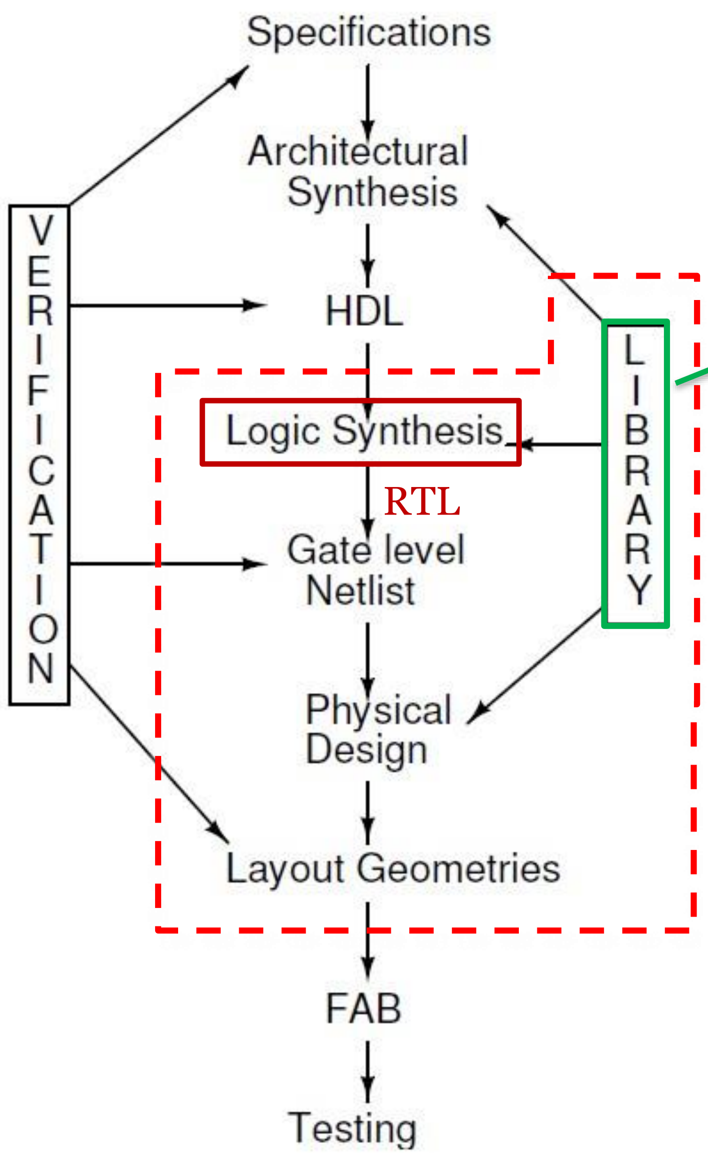
EE709:Testing & Verification

EE739: Processor Design
EE789: Algorithmic Design of Digital Systems

EE721: Hardware Description Languages

EE671 – This Course!
**Pre-Midesm:** Library Development
**Post-Midsem:** RTL to Fab
Course will not cover FPGA design – this will be taken up in EE705 (VLSI Design Lab)

## RTL to GDS and Synthesis



Standard Cell Library:
.v, .lef. Lib, .sp, .gdsII/.mag

**Logic Synthesis:**

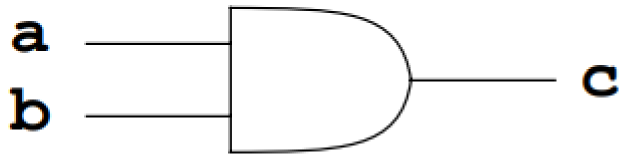
- ❑ Input: HDL (Verilog – behavioral/structural code)
- ❑ Output: RTL (Register transfer level Verilog)
- ❑ RTL: Could be or could not be mapped to a technology (Std. Cell Library)

## Logic Synthesis

- Popular logic synthesis tools:
  - Synopsys: Design Compiler
  - Cadence: Genus
  - Open Source: Yosys (will be used in this course)

- Synthesis: process of converting a gate level netlist (RTL) from a model of a circuit described in HDL
- Do we always need synthesizable codes?
  - Only if we need a hardware out of the code
  - Example: a testbench need not be synthesizable since it is not going to be translated to hardware!

## LOGIC SYNTHESIS: COMBINATIONAL LOGIC

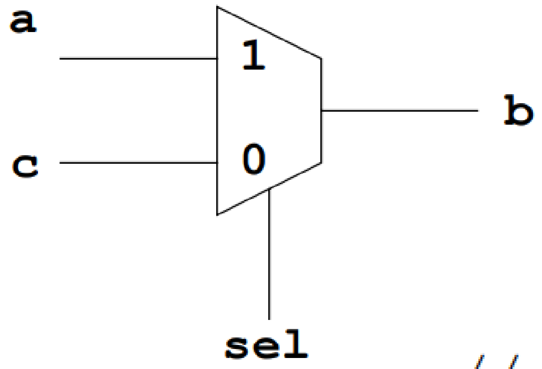


```
// Using a reg
// -----------------------------
wire a,b;
reg c;
always @ (a or b)
   c = a & b;
```

```
// Using a wire
// -----------------------------
wire a,b,c;
assign c = a & b;
```

```
// using a built in primitive (with instance name)
// -----------------------------
reg a,b;
wire c;
and u1 (c,a,b);  // output is always first in the list
```

## Logic Synthesis: Combinational Logic

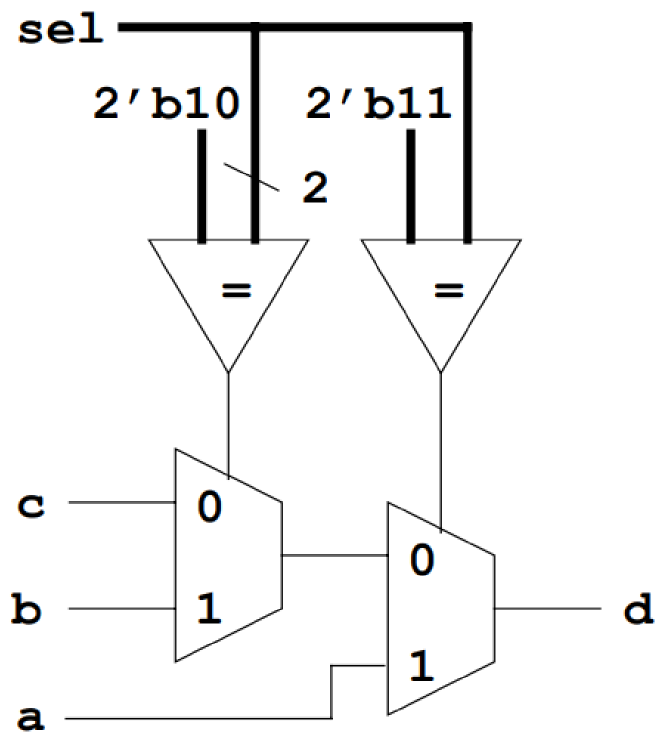


```
// 1. using an always
always@(a or b or sel)
    if (sel == 1'b1)
        c = a;
    else
        c = b;
```

```
// 2. using the ternary operator
wire c = sel ? a : b;
```

```
// 3. using the case statement
always @ (a or b or sel)
    case (sel)
        1'b1: c = a;
        1'b0: c = b;
    endcase
```

# LOGIC SYNTHESIS: COMBINATIONAL LOGIC



```
always @ (sl or a or b or c)
    case (sel)
        2'b11:  d = a;
        2'b10:  d = b;
        default: d = c;
    endcase
```

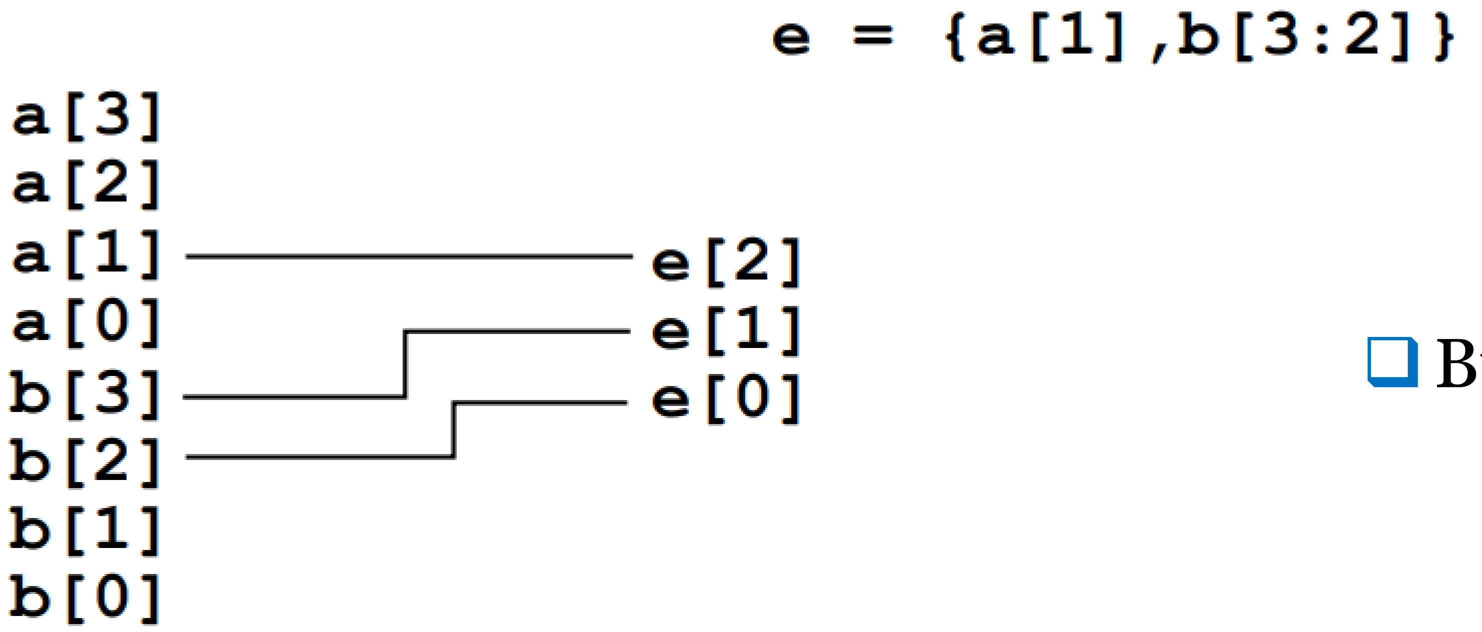
```
always @ (sl or a or b or c)
    if (sel == 2'b11)
        d = a;
    else if (sel ==2'b10)
        d = b;
    else
        d = c;
```

- Question: What is the implemented logic?

## Logic Synthesis: Bus Logic

```
wire [2:0] e = {a[1],b[3:2]};
```

```
reg [2:0] e;
always @ (a or b)
   e = {a[1],b[3:2]};
```

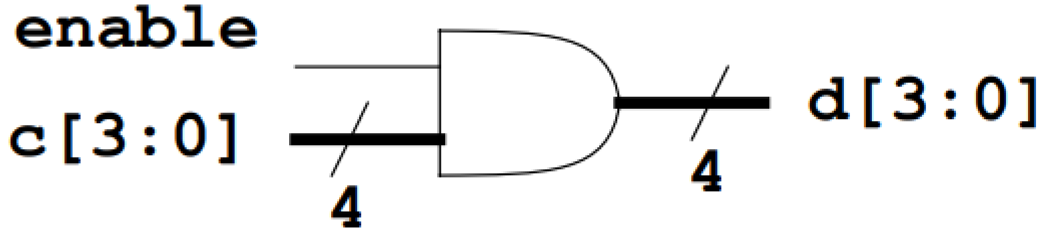


- Bus concatenation

## LOGIC SYNTHESIS: BUS LOGIC

```
wire [3:0] d = ({4{enable}} & c);
```

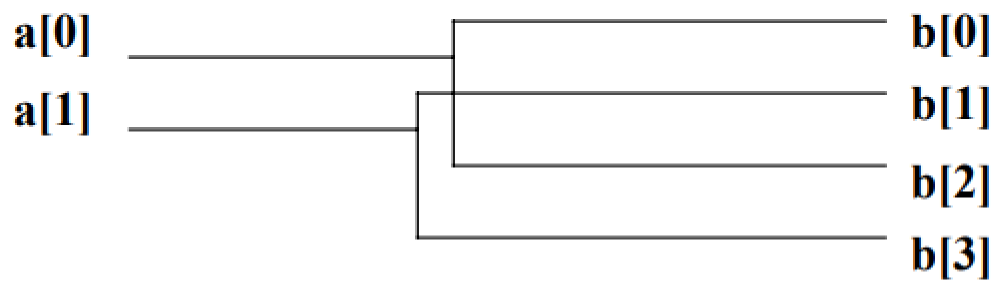
```
reg [3:0] d;
always @ (c or enable)
   d = c & {4{enable}};
```



- Bus Enabling

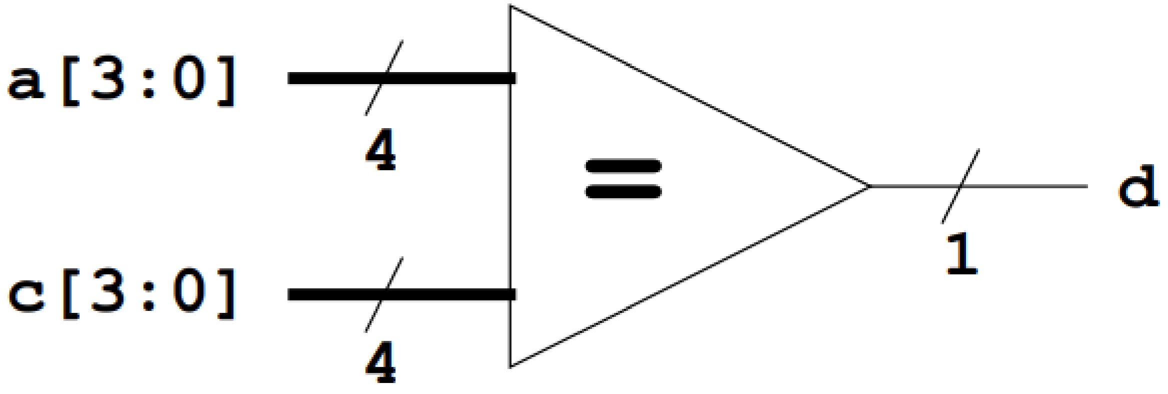
# Logic Synthesis: Bus Logic

```
wire [1:0] a;

wire [3:0] b;

assign b = {2{a}};
```



- Bus replication

## Logic Synthesis: Bus Logic

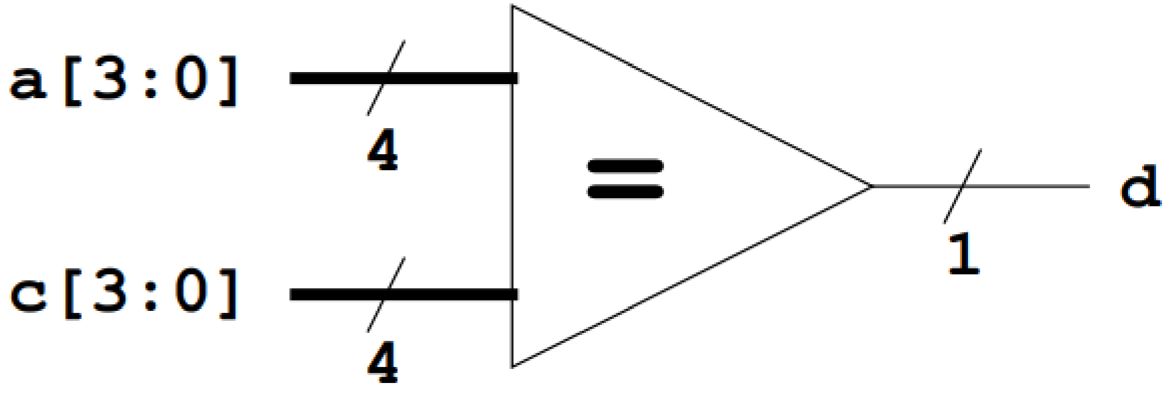


```
wire d;
assign d = (a == c);
```

```
reg d;
always @ (a or c)
    d = (a == c);
```

- Comparator

## LOGIC SYNTHESIS:



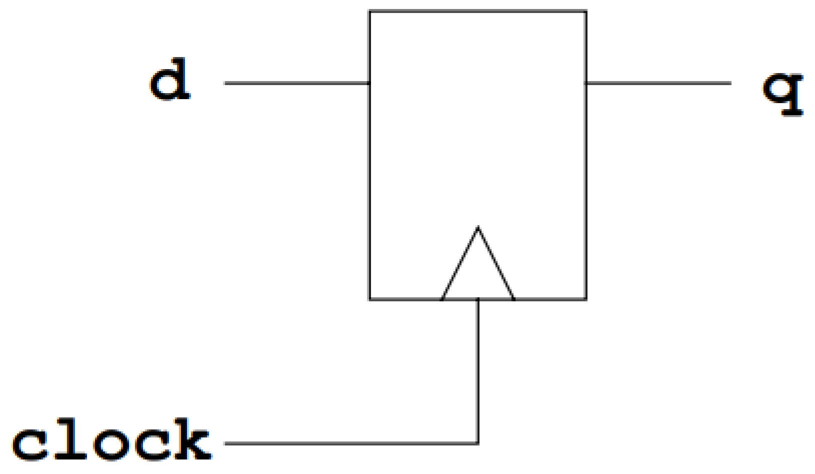
```
wire d;
assign d = (a == c);
```

```
reg d;
always @ (a or c)
   d = (a == c);
```

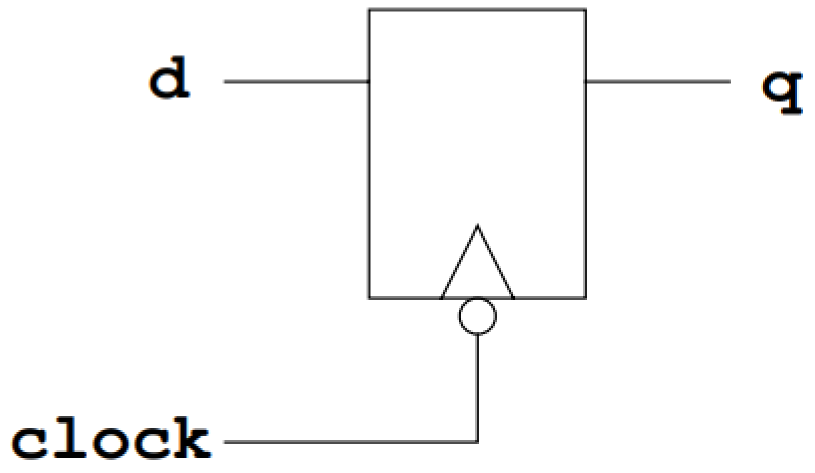
- Comparator

## Logic Synthesis: Sequential Logic

```
always @ (posedge clock)
    q <= d;
```



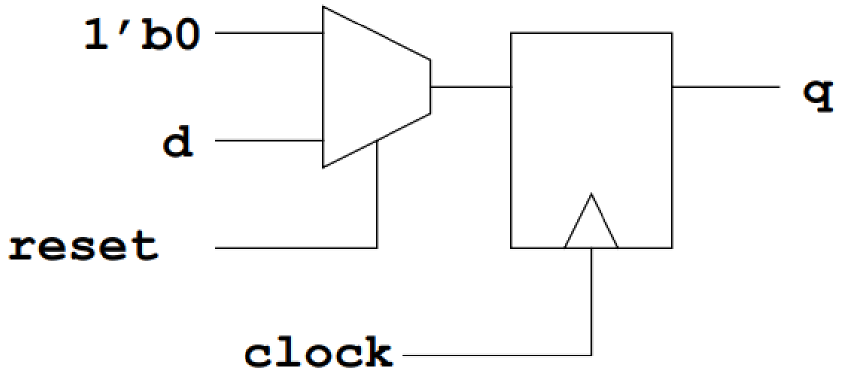
```
always @ (negedge clock)
    q <= d;
```



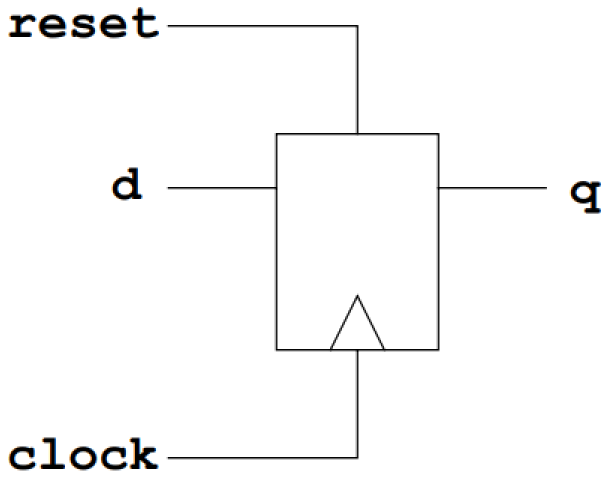
- Use non-blocking assignments ( <= )

## LOGIC SYNTHESIS: SEQUENTIAL LOGIC

```
always @ (posedge clock)
    if (reset)
        q <= 1'b0;
    else
        q <= d;
```



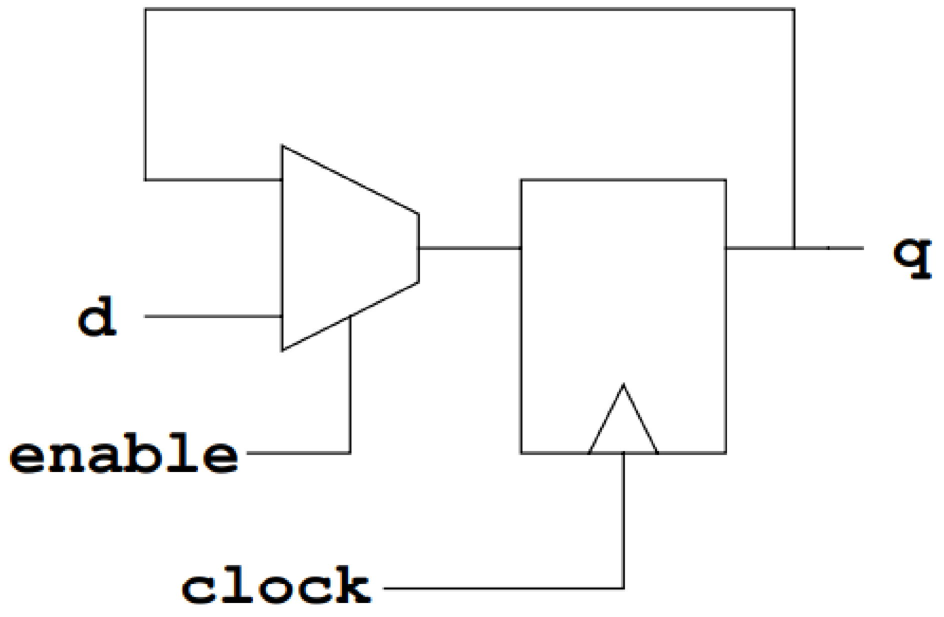
```
always @ (posedge clock or posedge reset)
    if (reset)
        q <= 1'b0;
    else
        q <= d;
```

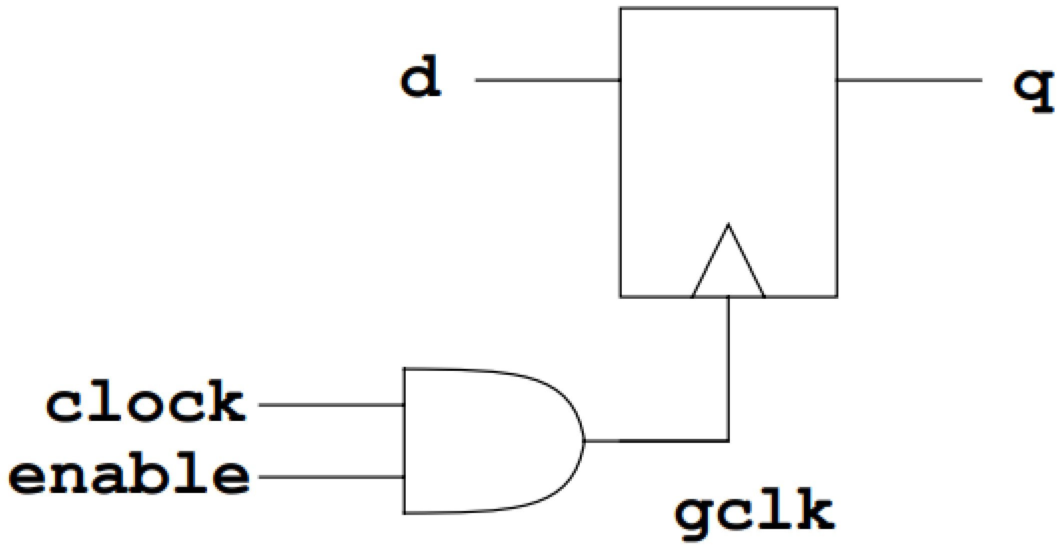


## Logic Synthesis: Sequential Logic

```
always @ (posedge clock)
   if (enable)
      q <= d;
```

```
wire gclk = (clock && enable);
always @ (posedge gclk)
     q <= d;
```

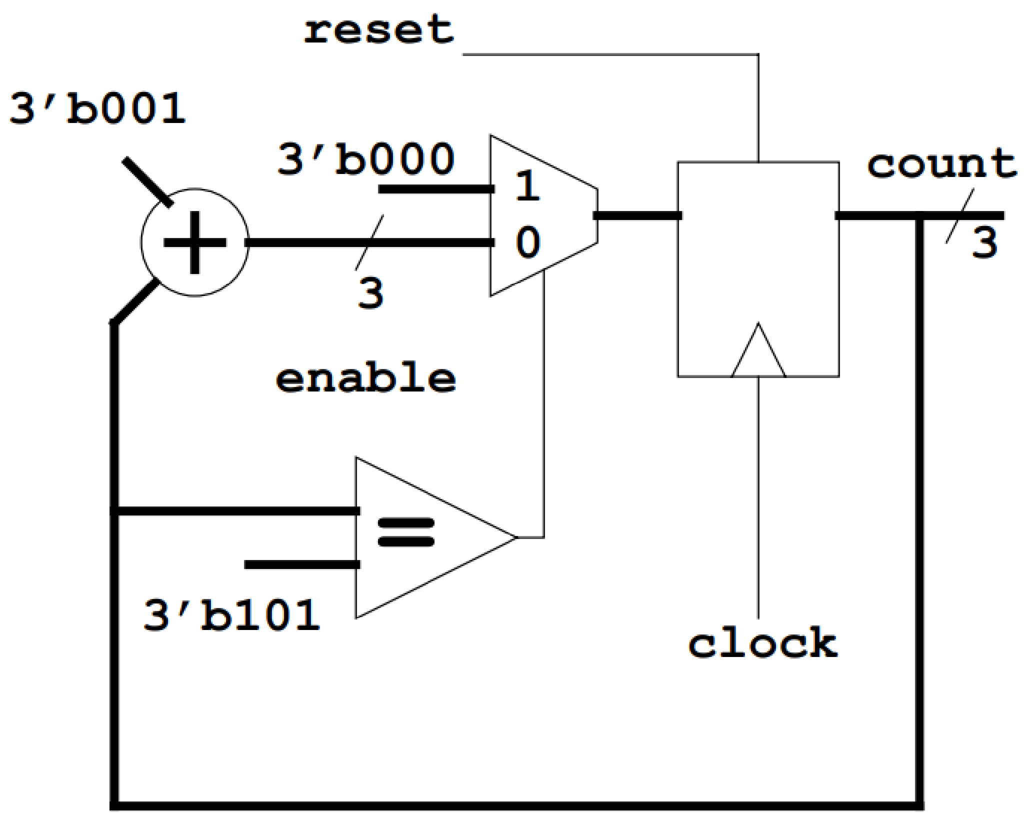




- Data enabled FF
- Clock gated FF (EN must not have any glitches!)

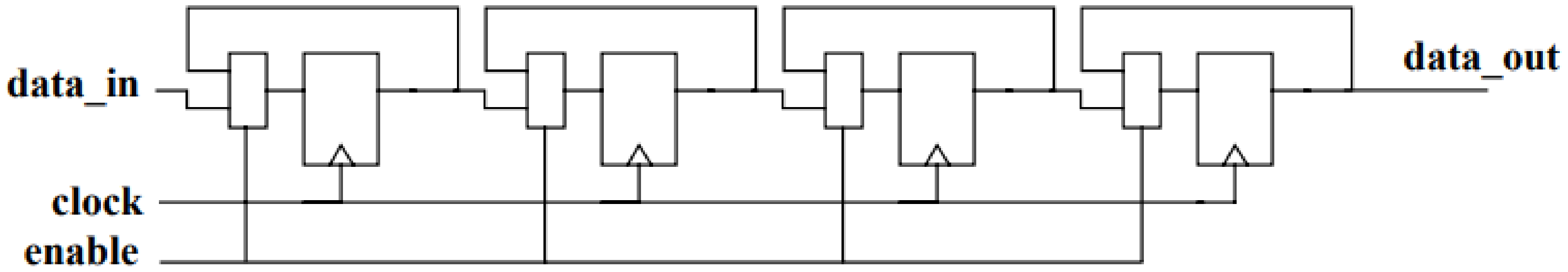
# Logic Synthesis: Sequential Logic

```
// 3 bit asynchronously resettable
// partial range counter
always @ (posedge clock or posedge reset)
   if (reset)
      count <= 3’b0;
   else
      if (count == 3’b101)
         count <= 3’b0;
      else
         count <= count + 3’b001;
```



## Logic Synthesis: Sequential Logic

```
module enabled_shift_reg (clock,enable,data_in,data_out);
input clock;
input enable;
input [3:0] data_in;
output [3:0] data_out;

reg [3:0] data_out;
reg [3:0] shift_reg_1;
reg [3:0] shift_reg_2;
reg [3:0] shift_reg_2;


always @ (posedge clock)
   if (enable)
      begin
      shift_reg_1 <= data_in;
      shift_reg_2 <= shift_reg_1;
      shift_reg_3 <= shift_reg_2;
      data_out <= shift_reg_3;
      end

endmodule
```



- Bus and FFs are 4-bit wide

# Logic Synthesis: Coding Guidelines

- Use non-blocking assignments (<=) in clocked procedures. Don’t use blocking assignments (=).

```
always @ (posedge clock)
   q <= d;
```

- Use blocking assignments (=) in combinational procedures:

```
always @ (a or b or sl)
   if (sl)
      d = a;
   else
      d = b;
```

- Make sure that the event lists are complete

```
always @ (a or b) // this event list is missing signal sl
   if (sl)
      d = a;
   else
      d = b;
```

## Logic Synthesis: Coding Guidelines

- Use named port mapping when instantiating.

```
state_machine u1 (
   .sm_in       (in1),
   .sm_clock    (clk),
   .reset       (reset),
   .sm_out      (data_mux)
   );
```

- Take care of indentation. Develop your own identation guidelines and stick to them. Make sure others can read them. It helps readability and debugging greatly if it is done properly.
- Comment code properly. The theory about good commenting is that you should be able to remove all functional code and the comments remaining should almost document the block you are designing.

- Don’t make the code any more complicated than it needs to be. Your priorities should be correctness, then readability and finally code efficiency.