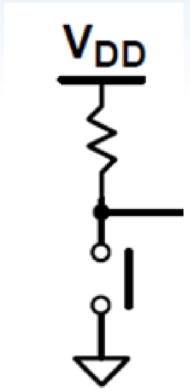
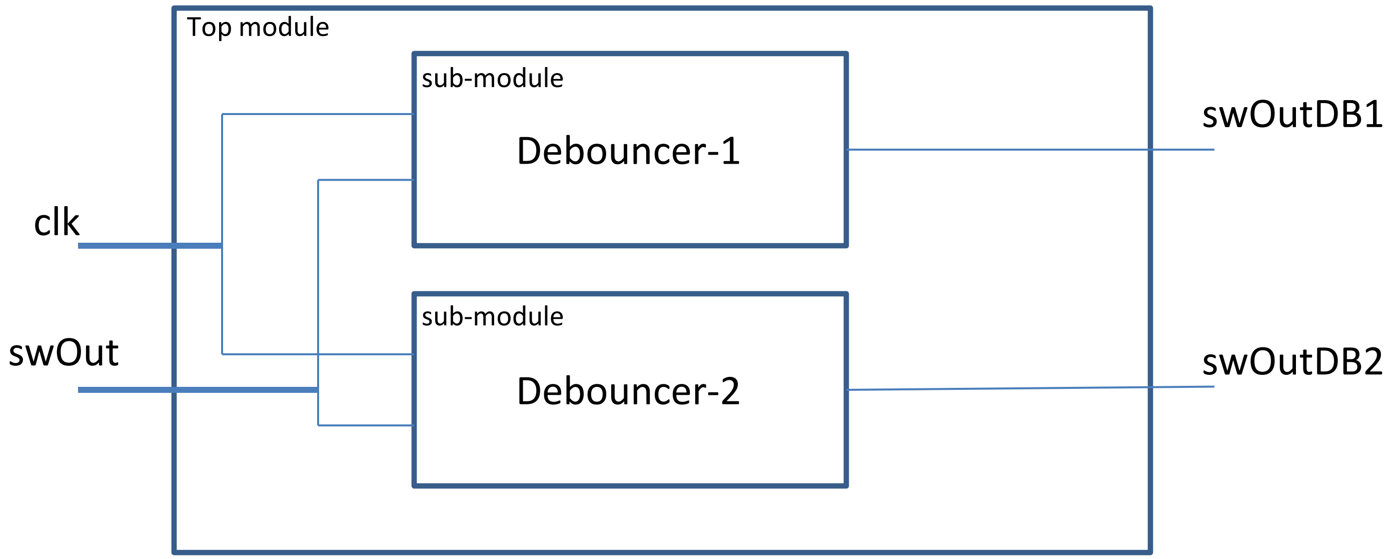
EE342 Lab-3
Instructions and Example Designs

Mehmet Çalı

Lab-3 Requirements

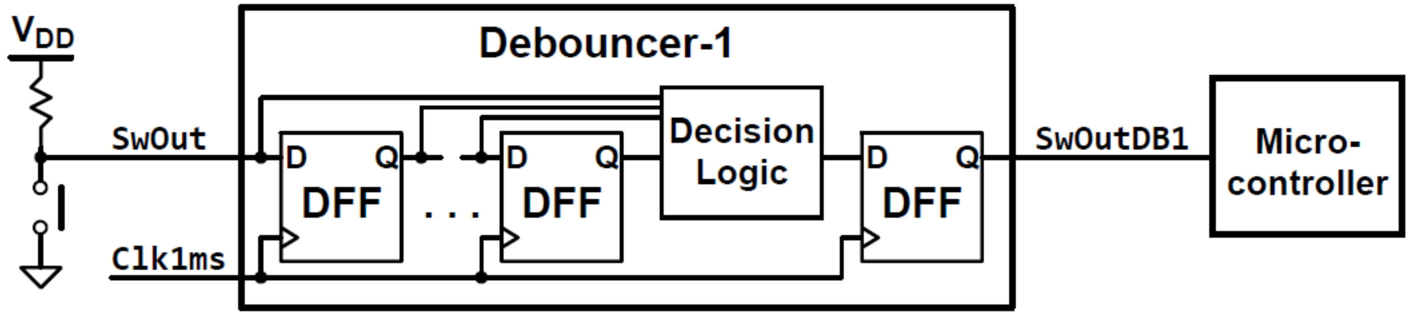
- Top Module
  - Inputs:
    - Single switch output
    - Clock
  - Outputs:
    - Debouncer-1
    - Debouncer-2





Debouncer-1

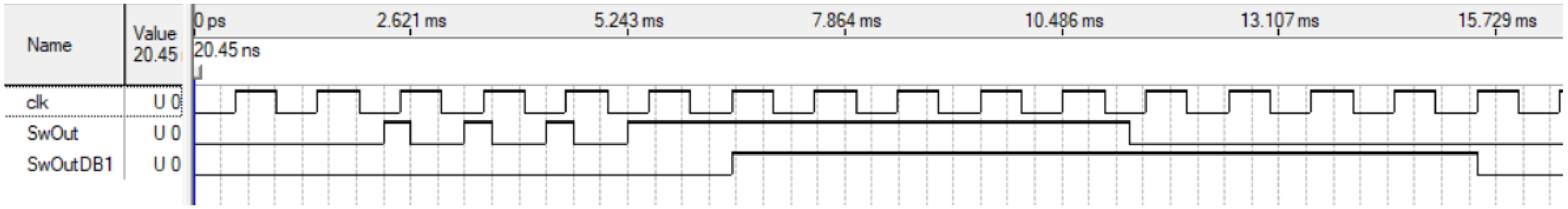
- Debouncer-1 and Debouncer-2 should filter the switch output by using a shift register and a combinational logic circuit whose schematic is given in Preliminary Work.
- Debouncer-1 should read 4 ms of steady switch output to transmit the switch output change to debouncer output.
- Although the maximum duration of random transmissions in SwOut is given to be 4 ms, this does not mean that you can just transfer the first output change with a delay of 4 ms independent of intermediate signals (4 ms of steady switch output is crucial for this module)

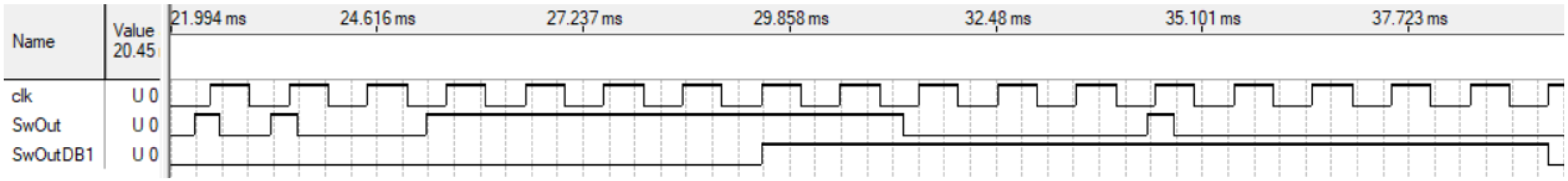


- Note that the design is almost completely specified so you should focus on implementing the design rather than trying miscellaneous solutions like creating a counter to adjust the timing
- You should observe the synthesis output from the Netlist viewers and try to fix if there is a considerable difference

Debouncer-1 Waveforms

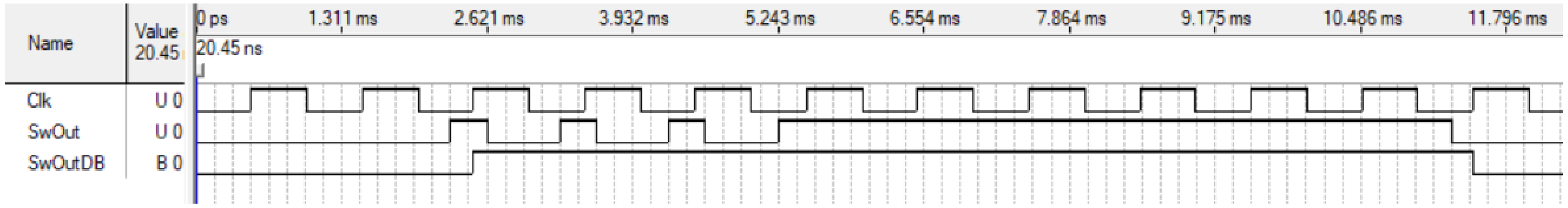
- Clock frequency should be set to 1 kHz
- Around 40-60 ms of end time should be enough (specify it according to you waveform)
- You should create a waveform that contains two press an two release to see how the output changes according to random ripples
- Example outputs for DB1:

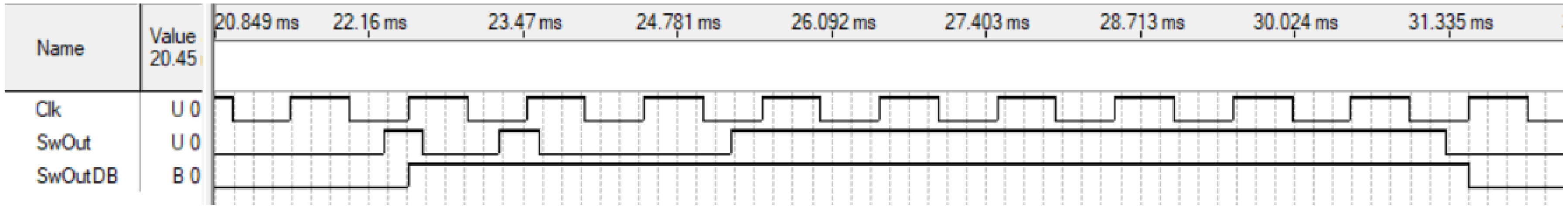




Debouncer-2

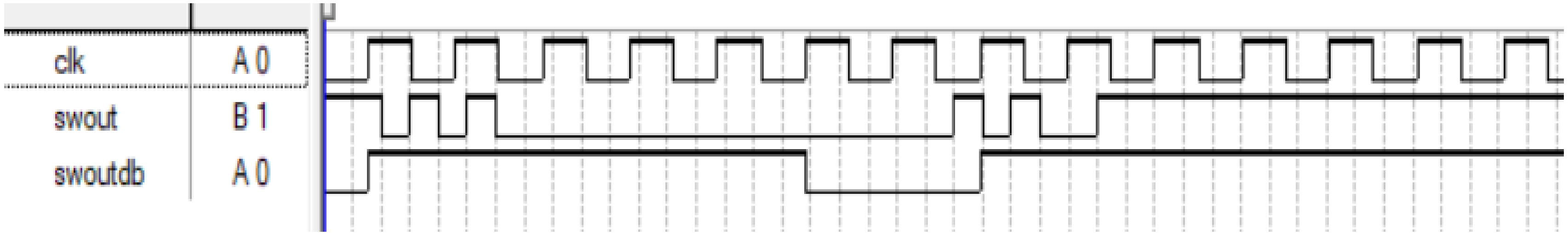
- In this module, you need to alter the output as soon as the ripples are detected and then keep the output steady for 4 ms.
- The schematic of the second module is not provided but you can again use a shift register combined with a combinational logic with different connections and logic circuit
- Example outputs of DB2:





Debouncer-2

- Debouncer-2 is trickier than the first module and wrong designs often misjudged as working due to construction of incapable waveform
- You should pay extra attention on whether the design forces the <u>debouncer output</u> to be stable for 4 ms or not
- Some wrong designs that gives misleading occasional correct results generally fails to perform accurately in the following waveform

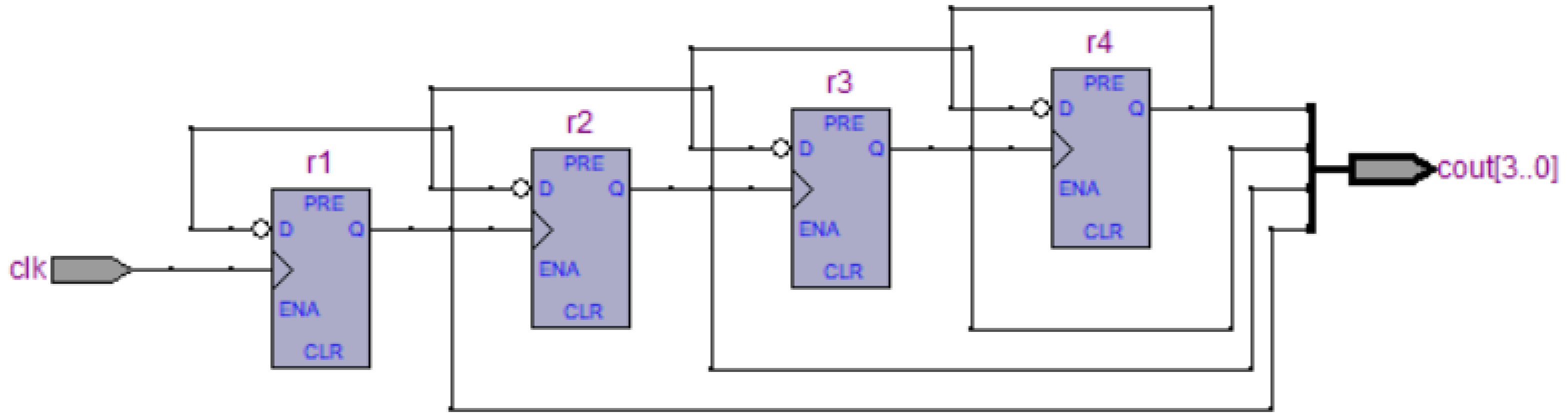


DB-2 Waveform output that belongs to an inaccurate design

- As seen in the figure above when the swout is forced to be high initially, output immediately goes to high in the next clock cycle as expected. However, after followed by two quick press and release swoutdb is observed to stay low around 2.5 ms, although the output should preserve its value for at least 4 ms.
- Your simulation file should start with the above waveform, and then you should extend the waveform with additional press and releases to test against possible design faults thoroughly

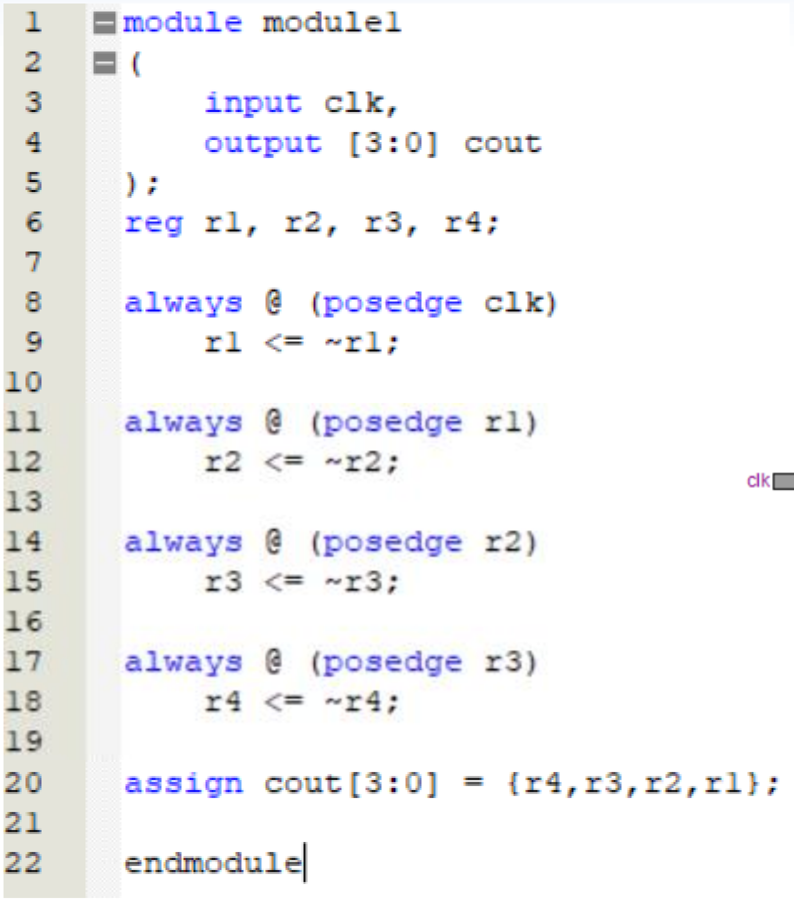
Asynchronous Counter

- We want to design a 4 bit asynchronous (ripple) counter, in other words a counter that does not have a common clock signal
- Most basic schematic of an asynchronous counter can be shown as figure below

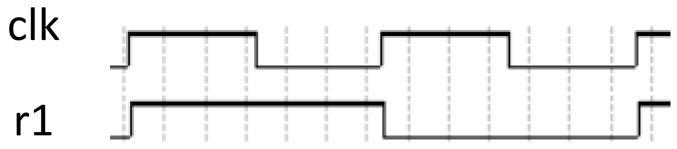


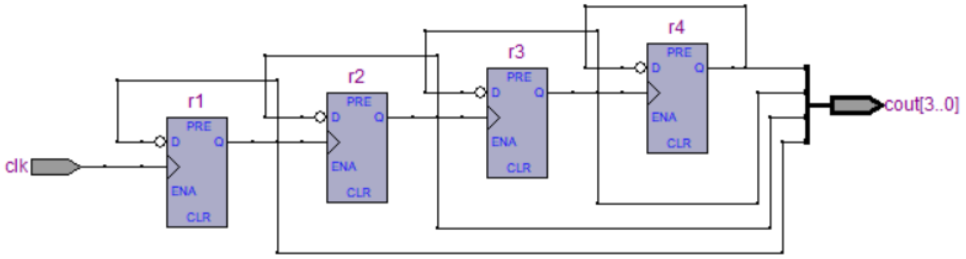
Asynchronous Counter

- We can make use of concatenated clock dividers (frequency divider) for our purpose



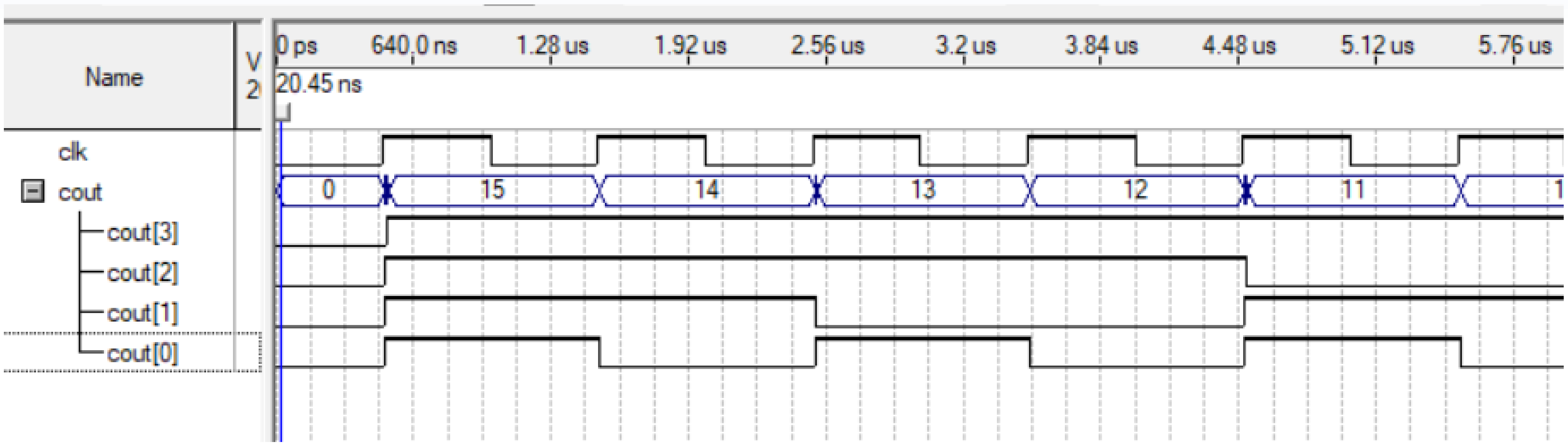
```verilog
module module1
(
    input clk,
    output [3:0] cout
);
reg r1, r2, r3, r4;

always @ (posedge clk)
    r1 <= ~r1;

always @ (posedge r1)
    r2 <= ~r2;

always @ (posedge r2)
    r3 <= ~r3;

always @ (posedge r3)
    r4 <= ~r4;

assign cout[3:0] = {r4,r3,r2,r1};

endmodule
```





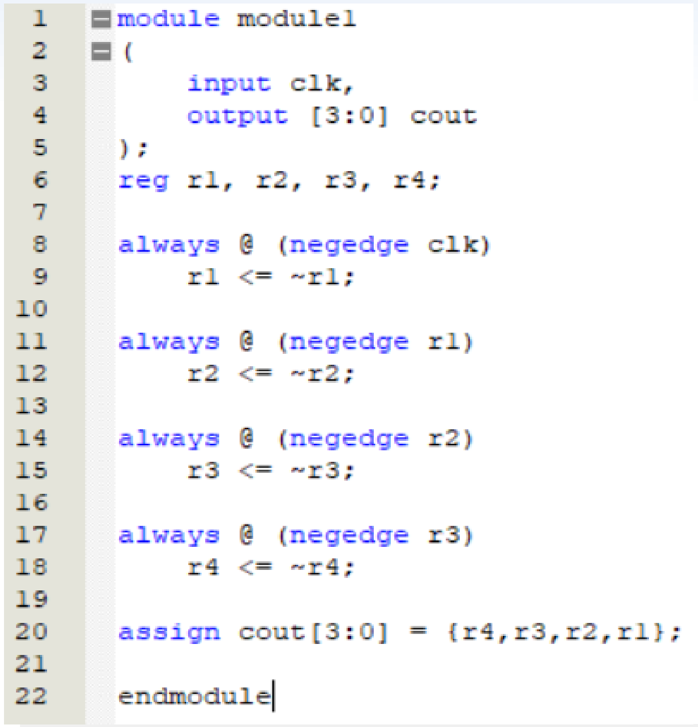
Asynchronous Counter

- Output Waveform:

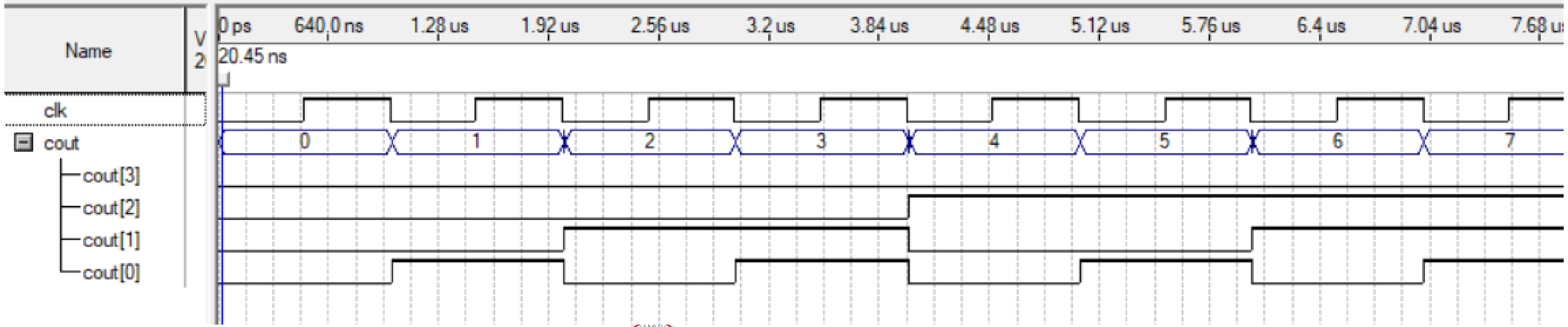


- How can we convert it to up-counter?

Asynchronous Counter

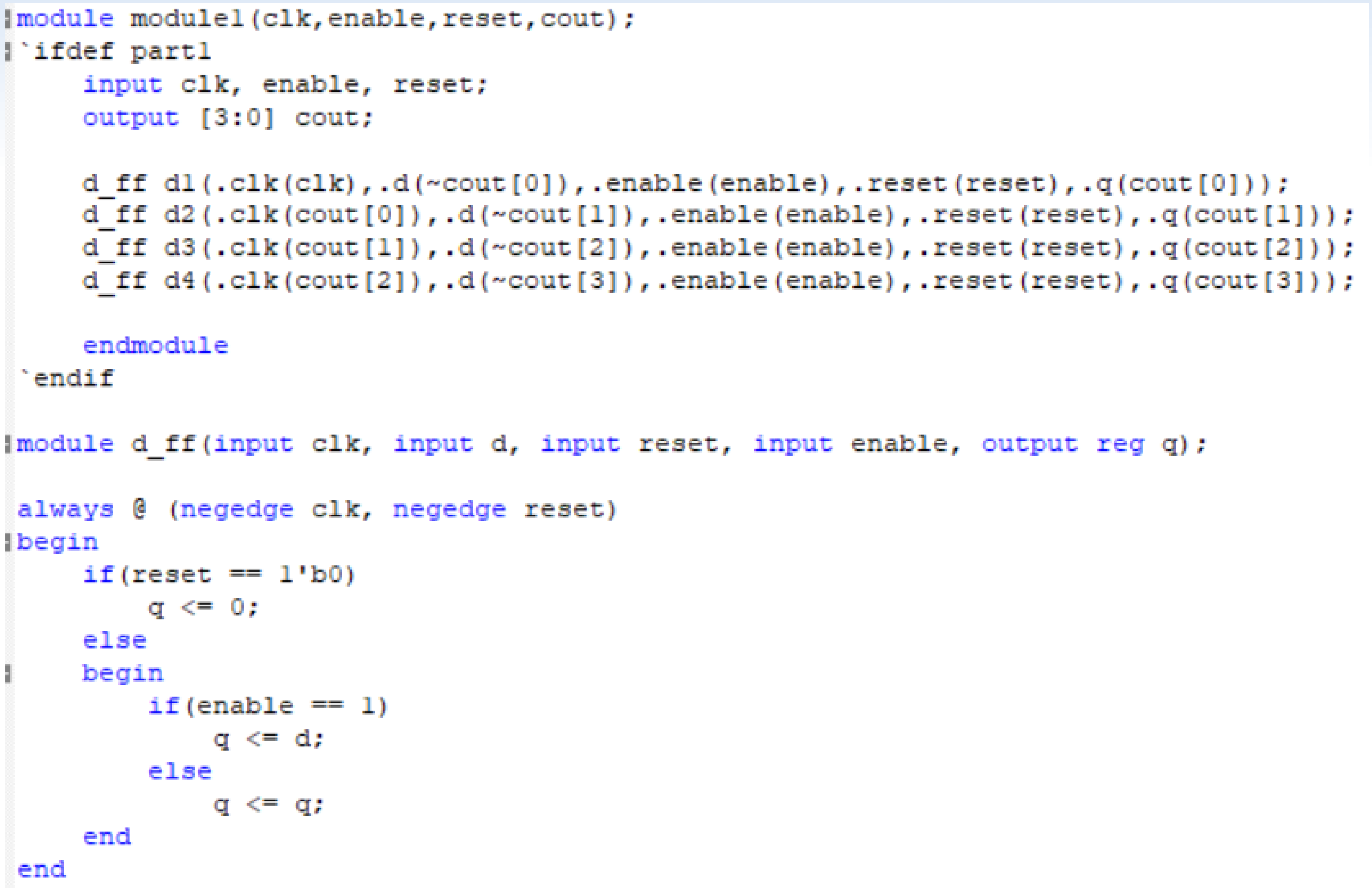


```verilog
module module1
(
	input clk,
	output [3:0] cout
);
reg r1, r2, r3, r4;

always @ (negedge clk)
	r1 <= ~r1;

always @ (negedge r1)
	r2 <= ~r2;

always @ (negedge r2)
	r3 <= ~r3;

always @ (negedge r3)
	r4 <= ~r4;

assign cout[3:0] = {r4,r3,r2,r1};

endmodule
```



Asynchronous Counter with enable and reset

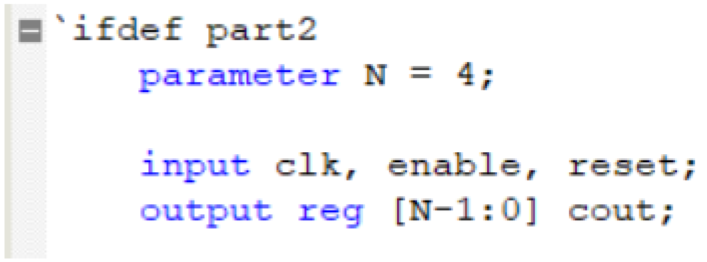
- Now, lets add enable and reset input and define a dff module and instantiate it, instead of reconstructing from the beginning
- New requirements are:
  - Synchronous enable
  - Asynchronous active low reset
  - 4 bit up counter



```
module module1(clk,enable,reset,cout);
`ifdef part1
    input clk, enable, reset;
    output [3:0] cout;

    d_ff d1(.clk(clk),.d(~cout[0]),.enable(enable),.reset(reset),.q(cout[0]));
    d_ff d2(.clk(cout[0]),.d(~cout[1]),.enable(enable),.reset(reset),.q(cout[1]));
    d_ff d3(.clk(cout[1]),.d(~cout[2]),.enable(enable),.reset(reset),.q(cout[2]));
    d_ff d4(.clk(cout[2]),.d(~cout[3]),.enable(enable),.reset(reset),.q(cout[3]));

    endmodule
`endif

module d_ff(input clk, input d, input reset, input enable, output reg q);

always @ (negedge clk, negedge reset)
begin
    if(reset == 1'b0)
        q <= 0;
    else
    begin
        if(enable == 1)
            q <= d;
        else
            q <= q;
    end
end
```

Using parameters and module arrays

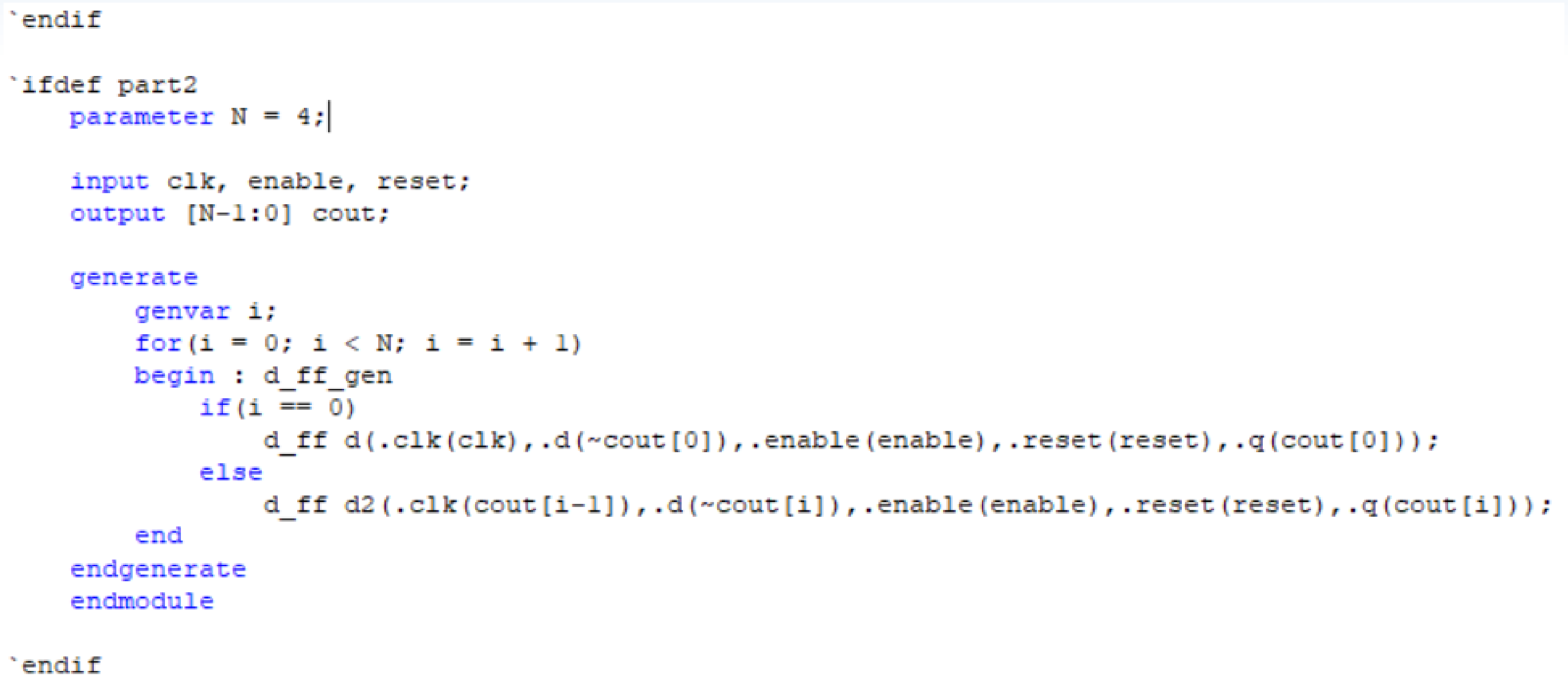
Parameters can be thought as constant data types, they usually defined at the beginning and they can be used to specify the size of the busses or module arrays



```
`ifdef part2
    parameter N = 4;

    input clk, enable, reset;
    output reg [N-1:0] cout;
```

If we have a large counter size instantiating them one by one will be impractical, so lets try to adjust our code to work with any number of counter size

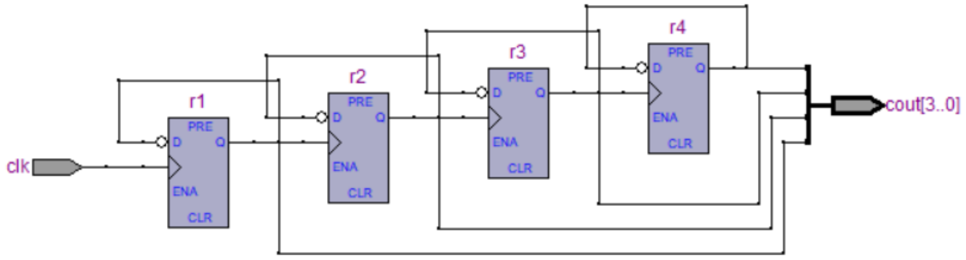
Generate block



```
`endif

`ifdef part2
    parameter N = 4;

    input clk, enable, reset;
    output [N-1:0] cout;

    generate
        genvar i;
        for(i = 0; i < N; i = i + 1)
        begin : d_ff_gen
            if(i == 0)
                d_ff d(.clk(clk),.d(~cout[0]),.enable(enable),.reset(reset),.q(cout[0]));
            else
                d_ff d2(.clk(cout[i-1]),.d(~cout[i]),.enable(enable),.reset(reset),.q(cout[i]));
        end
    endgenerate
    endmodule

`endif
```

Using arrays of instances

```
`endif
`ifdef part3
    parameter N = 4;

    input clk, enable, reset;
    output [N-1:0] cout;

    d_ff d[N-1:0](.clk({cout[N-2:0],clk}),.d(~cout[N-1:0]),.enable(enable),.reset(reset),.q(cout[N-1:0]));

    endmodule
`endif

module d_ff(input clk, input d, input reset, input enable, output reg q);

always @ (negedge clk, negedge reset)
begin
    if(reset == 1'b0)
        q <= 0;
    else
    begin
        if(enable == 1)
            q <= d;
        else
            q <= q;
    end
end
endmodule
```



clk
r1
r2
r3
r4
PRE
D
Q
ENA
CLR
cout[3..0]