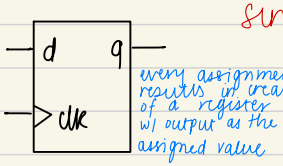
Sequential Verilog

- small blocks can be integrated into larger designs using instantiation
- we usually describe synchronous circuits (edge-triggered components)
- the synthesis tools will generally convert designs to D-type flip-flop / registers

↓

at the rising edge of the clock, input is passed to output



every assignment results in creation of a register w/ output as the assigned value

flip-flop

single bit reg.

```
module simplereg ( input d, clk,
                   output reg q);
```

synchronous always block

assignment within the block only occurs when pos(itive) edge of clk

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

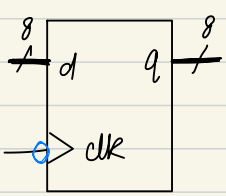
d is assigned to q

instruction to synthesis tool

different from combinational assignment

```
endmodule
```

8 individual registers




```
module multireg ( input [7:0] d, input
                  clk, output reg [7:0] q);
always @ (posedge clk)
  q <= d ;
```

falling-edge trigger:
always @ (negedge clk)

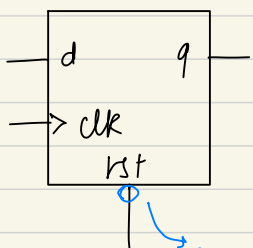
```
endmodule
```

- resetting values in a register

asynchronous : whenever reset input is asserted, the contents are set to the reset value

synchronous : at rising edge, if reset is asserted, then contents are set to the reset value

↑ preferred

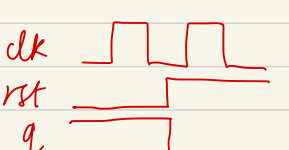
- 


```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);

always @(posedge clk)
begin
  if(!rst)
    q <= 8'b0000_0000;
  else
    q <= d;
end

endmodule
```

sometimes rst is active low

or posedge rst

recommended use a synchronous reset



- asynchronous reset :

```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);
always @(posedge clk or posedge rst)
begin
   if (!rst)
      q <= 8'b0000_0000;
   else
      q <= d ;
end

endmodule
```

neg if active low

- multiple registers

```
module multireg (input [7:0] a, b, c,
                 input clk, rst,
                 output reg [7:0] q, r, s);
always @(posedge clk)
begin
   if (!rst)
   begin
      q <= 8'b0000_0000;
      r <= 8'b0000_0000;
      s <= 8'b0000_0000;
   end
   else begin
      q <= a;
      r <= b;
      s <= c;
   end
end
endmodule
```

reset and clk may be left out in the diagrams. but they should be there

Assignment

- `<=` : non-blocking assignment

combinational : `=` , order matters
synchronous : `<=` , order doesn't matter

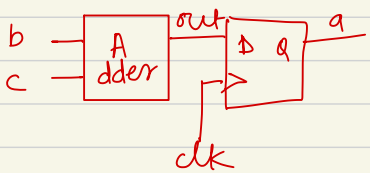
↓ always for always@*

↑ always for always@ (posedge clk)

↳ look into this

↓ but timing matters
it is one after the other

↳ results in a register :
eg. `a <= b+c;`



Counters

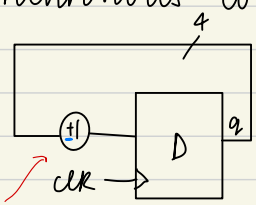
binary counters

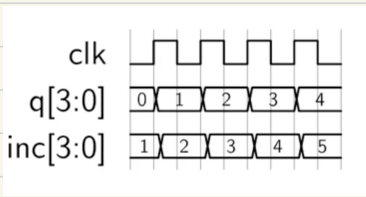
- increasing output value in each cycle

2 bit · 00 → 01 → 10 → 11 and repeat

counters can count down also

- synchronous counters





combinational increment

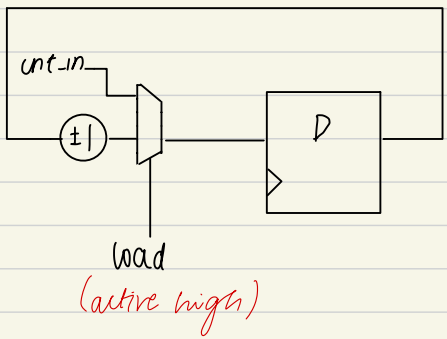
down

```
module simplecnt (input clk, rst, output reg [3:0] q);
  always @ (posedge clk)
  begin
    if (rst)
      q <= 4'b0000
    else
      q <= q + 1'b1;
  end
endmodule
```

old value

```
else
  if (down)
    q <= q - 1'b1;
  else
    q <= q + 1'b1;
```

- input custom value into counter




```
module simplecnt (input clk, rst,
                  input down, load,
                  input [3:0] cnt_in,
                  output reg [3:0] q);
always @(posedge clk)
begin
  if (rst)
    q <= 4'b0000;
  else
    if (load)
      q <= cnt_in;
    else
      if (down)
        q <= q - 1'b1;
        q <= q + 1'b1;
end
endmodule
```

don't mix rst is not part of the body of the counter while load is