Verilog

Harris & Harris, Digital Design and Computer Architecture

Ch. 4 - Hardware Description Languages

- 4.2.10 Delays
- 4.4 SEQUENTIAL LOGIC
- 4.5 MORE COMBINATIONAL LOGIC

Visualize a circuit

- To visualize the circuit, we need a sequence of commands that we run with **yosys**
- The sequence can be in a file having extension .ys

```
$ yosys synth.ys
```

```
# example of synth.ys
# read design
read -sv example.v
# elaborate design hierarchy
hierarchy -top example
# convert processes (always blocks)to netlist
# elements and perform some simple optimizations
proc; opt
# translate netlist to gate logic and perform
# some simple optimizations
techmap; opt
# write design netlist to a json file
write_json example.js
```

Visualize a circuit

- From file `example.js` produced by `yosys`, a file `example.svg` (Scalable Vector Graphics format) can be produced using the command

```
$ netlistsvg example.js -o example.svg
```

- File `example.svg` can be finally seen in a graphical window using the command `ristretto` `(picture.viewer)`

```
$ ristretto example.svg
```

Visualize a circuit

- Let us consider the module for the full-adder, stored in a file with extension (for example `esempioFA.v`)

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;   // internal nodes
  assign p = a ^ b;
  assign g = a & b;
  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

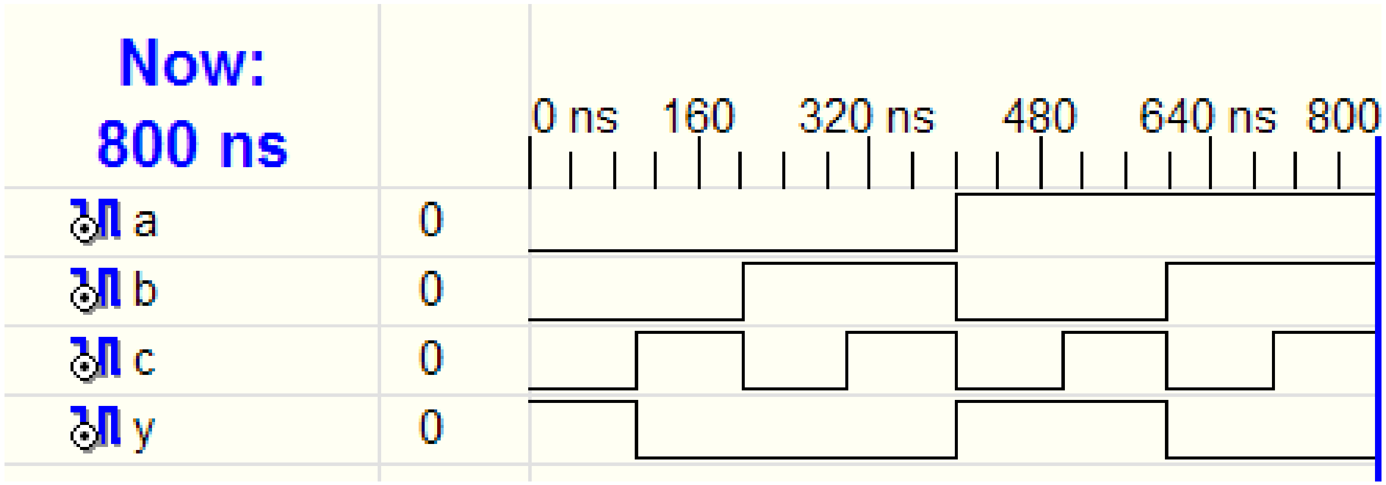
Give the following commands to visualize the circuit

```
$ yosys synth-esempioFA.ys
$ netlistsvg esempioFA.js -o esempioFA.svg
$ ristretto esempioFA.svg
```

Delays

- HDL statements may be associated with **delays**, helpful during simulation to predict how fast a circuit will work and for debugging purposes (but ignored during synthesis)
- Let's consider a previous example and its simulation waveforms

```
module example(input  logic a, b, c,
               output logic y);
assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```



Delays

- We now add delays to the previous function
- We assume:
  - **inverters** have a delay of **1 ns**,
  - **three-input AND gates** have a delay of **2 ns**,
  - **three-input OR gates** have a delay of **4 ns**

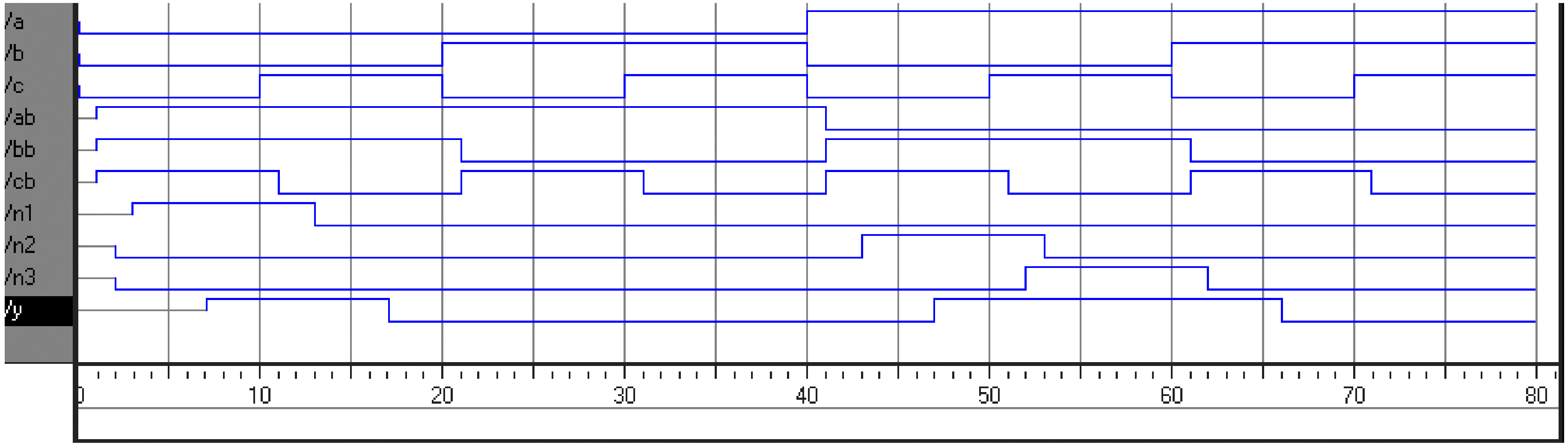
```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

In SystemVerilog, a **# symbol** is used to indicate the number of units of delay

Delays

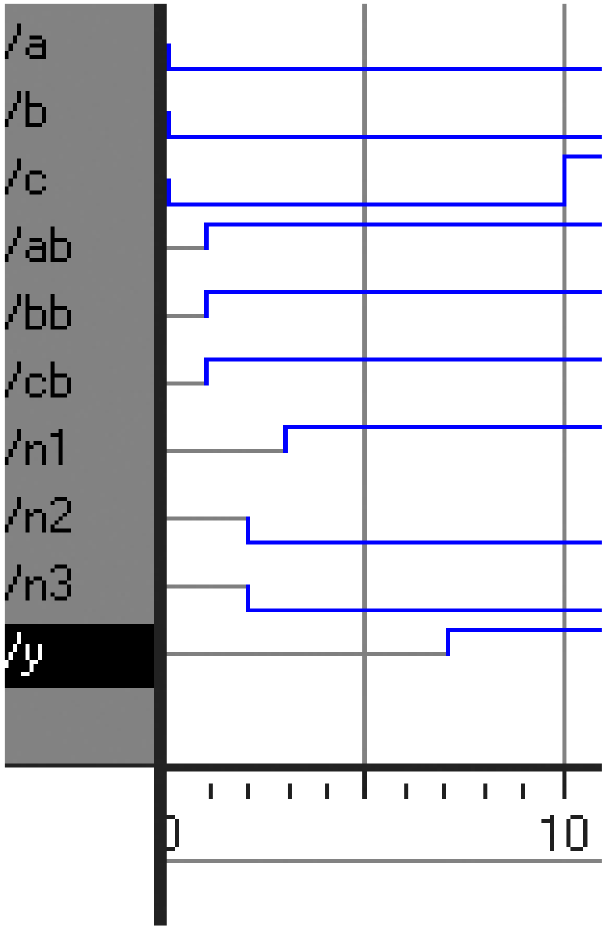
```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Let's analyze the simulation waveform



Delays

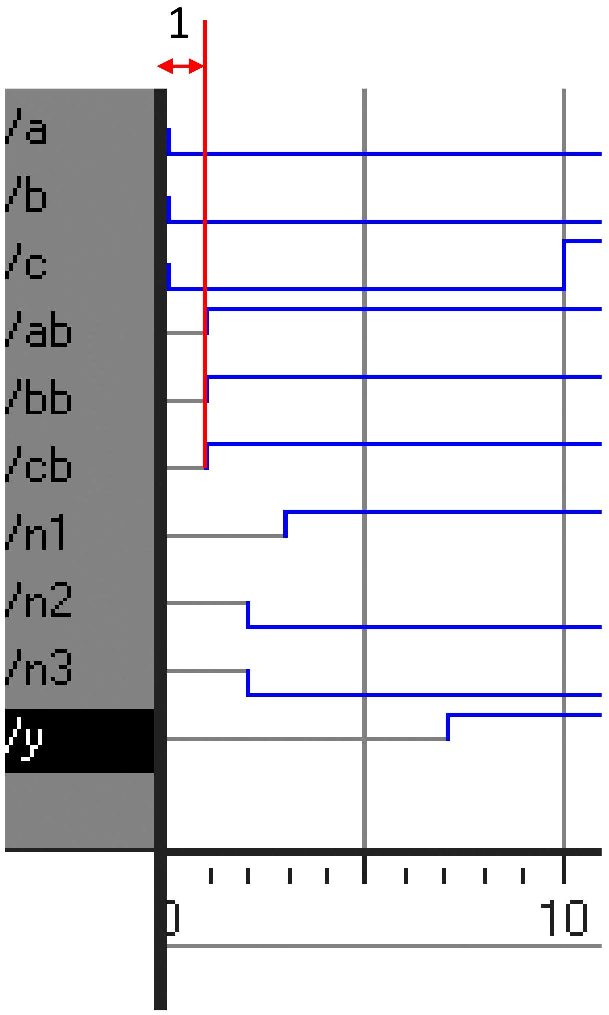
```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



Note that the beginning of the simulation only the variables **a, b, c** are known.
In particular y is initially unknown

Delays

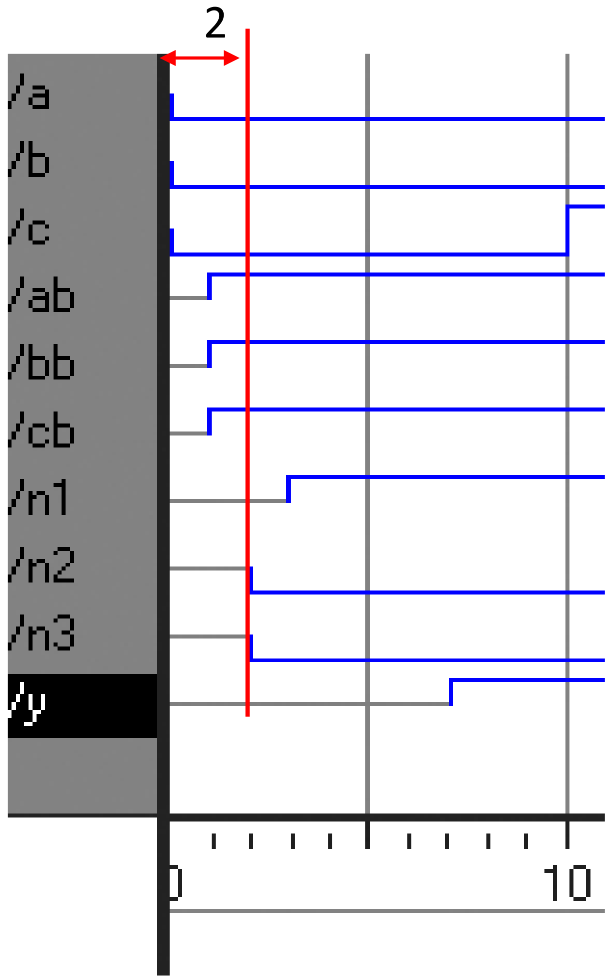
```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                 ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



After 1 ns inverters produce their outputs

Delays

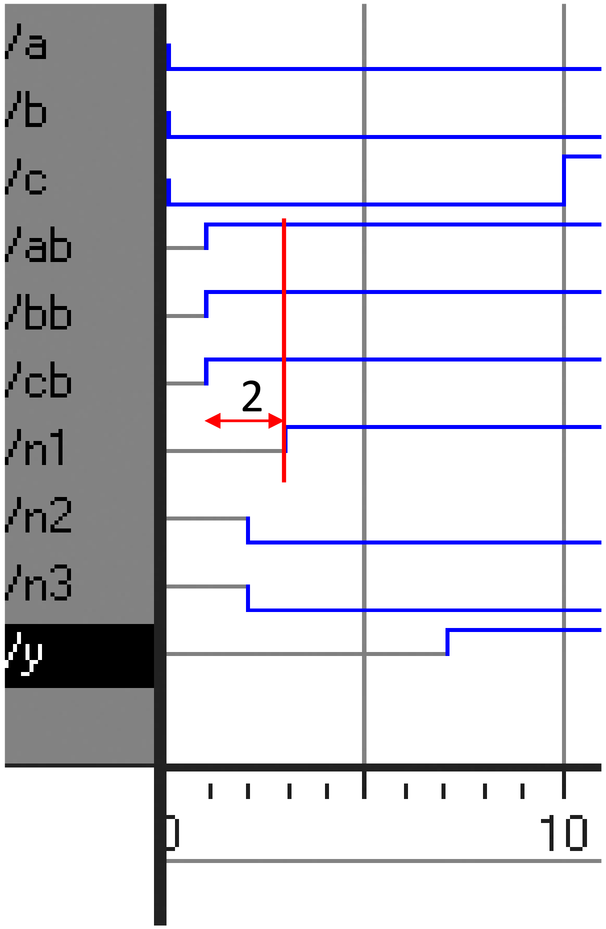
```
module example(input  logic a, b, c,
                      output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                         ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



After 2 ns *and* gates produce their outputs

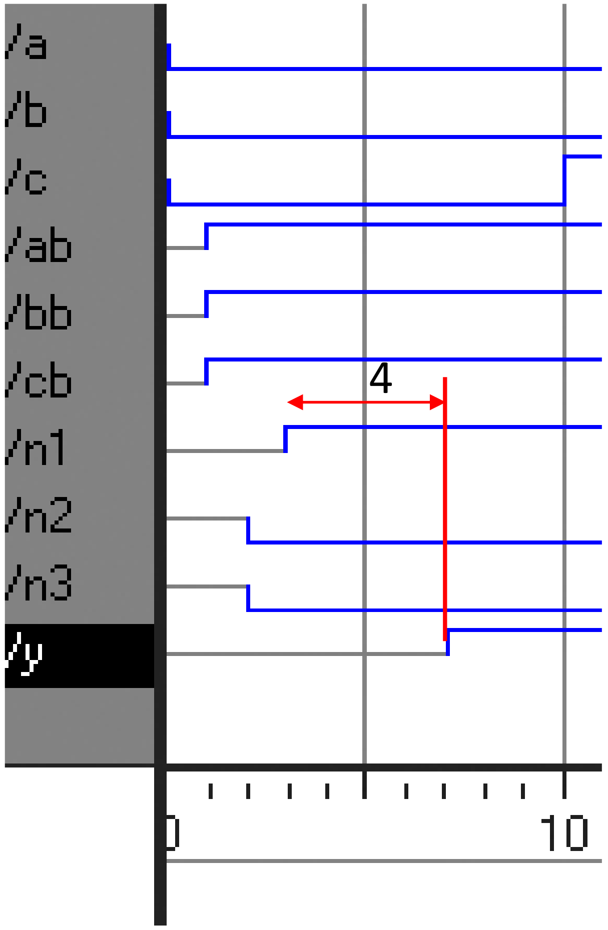
Delays

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                    ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



Delays

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



Sequential Logic

- SystemVerilog uses **idioms** to describe latches, flip-flops and FSMs

- Other coding styles may simulate correctly but produce incorrect hardware

Always Statement

- The majority of modern commercial systems are built with registers using positive edge-triggered D flip-flops
- In SystemVerilog `always` statements signals keep their old value until an event in the **sensitivity list** takes place that explicitly causes them to change

```
always @(sensitivity list)
  statement;
```

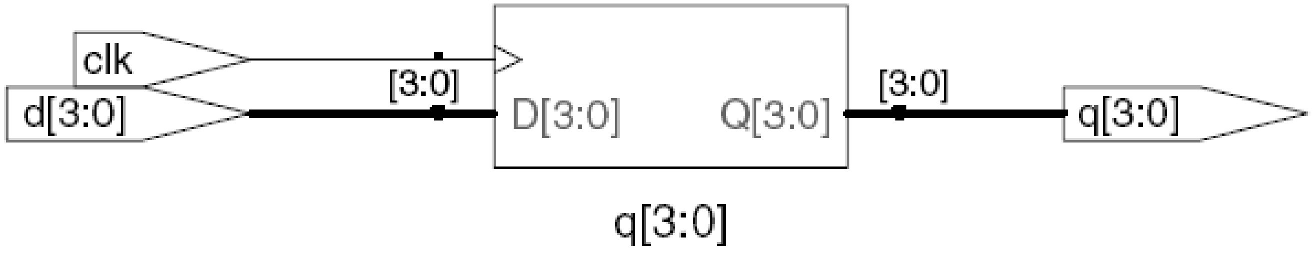
- Whenever the event in **sensitivity list** occurs, `statement` is executed
- Hence, code using `always`, with appropriate sensitivity lists, can be used to describe **sequential circuits**

D Flip-Flop

- For example, the flip-flop includes only **clk** in the sensitive list
- It remembers its old value of q until the next rising edge of the **clk**, even if d changes in the interim

```
module flop(input  logic clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff@(posedge clk)
    q <= d;              // pronounced “q gets d”
endmodule
```

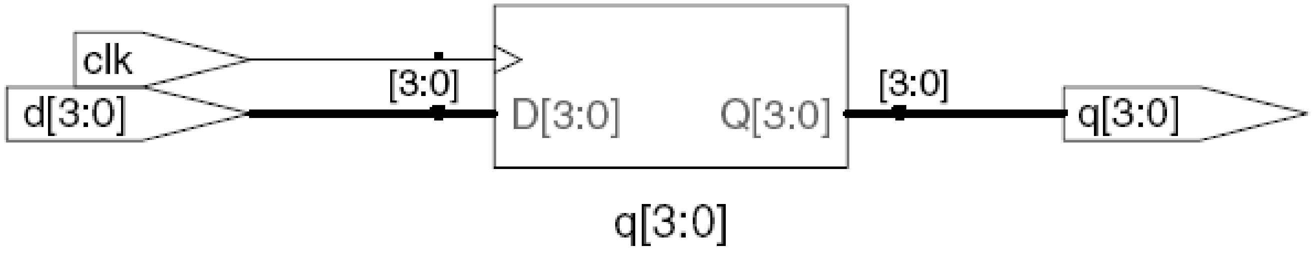


D Flip-Flop

- <= is called a **nonblocking assignment**
- <= is used instead of assign inside an always statement

```
module flop(input  logic clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff@(posedge clk)
    q <= d;                // pronounced "q gets d"
endmodule
```



D Flip-Flop

- `always` statements can be used to imply **flip-flops**, **latches**, or **combinational logic**, depending on the sensitivity list and statement

- To reduce the risk of common errors, SystemVerilog introduces `always_ff`, `always_latch`, and `always_comb`

- `always_ff` behaves like `always` but is used exclusively to imply flip-flops and allows tools to produce a warning if anything else is implied

Resettable registers

- When simulation begins or power is first applied to a circuit, the output of a flop or register is **unknown**
- This is indicated with ***x*** in SystemVerilog

- It is good practice to use **resettable registers** so that on powerup you can put your system in a **known state**

- The reset may be either **asynchronous** or **synchronous**:
  - *asynchronous* reset occurs immediately
  - *synchronous* reset clears the output only on the next rising edge of the clock

Resettable D Flip-Flop

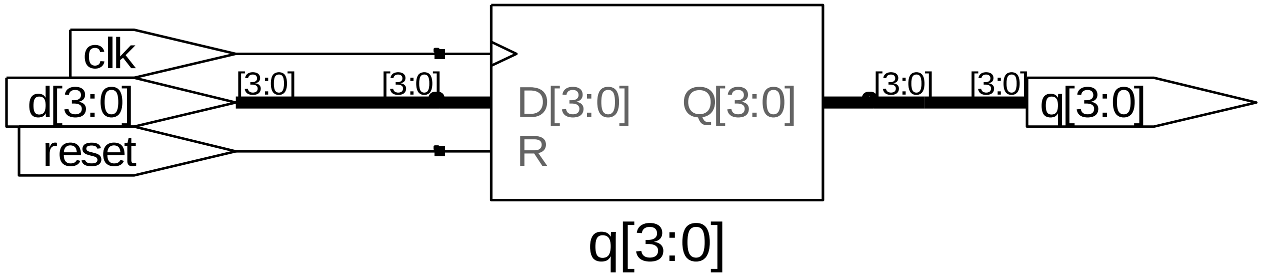
```
module flopr(input  logic clk,
             input  logic reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```

Posedge clk is alone in the sensitivity list thus:

- **synchronously** resettable flops respond to reset **only on the rising edge of the clock**



Resettable D Flip-Flop

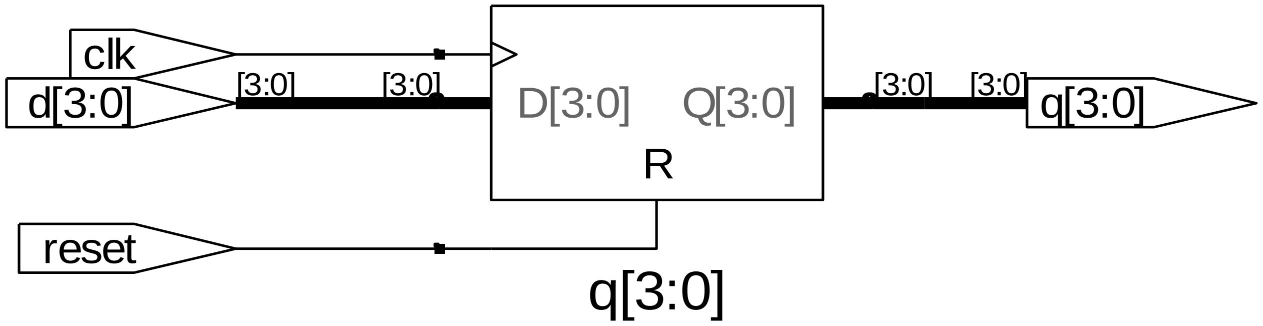
```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```

Posedge reset is in the sensitivity list only on the *asynchronously resettable flop*, thus:

- **asynchronously** resettable flops **immediately** respond to a rising edge on **reset**

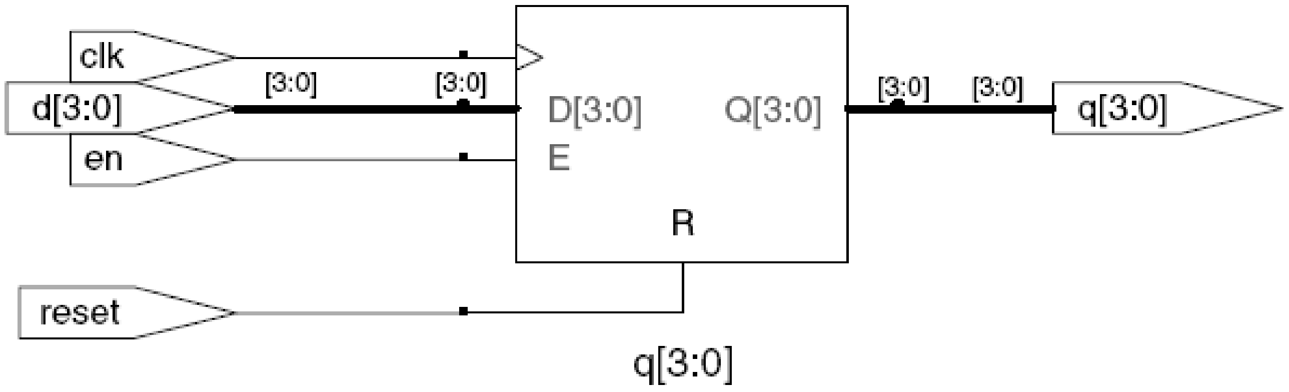


D Flip-Flop with Enable

```
module flopren(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // enable and asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if       (reset) q <= 4'b0;
    else if (en)     q <= d;

endmodule
```



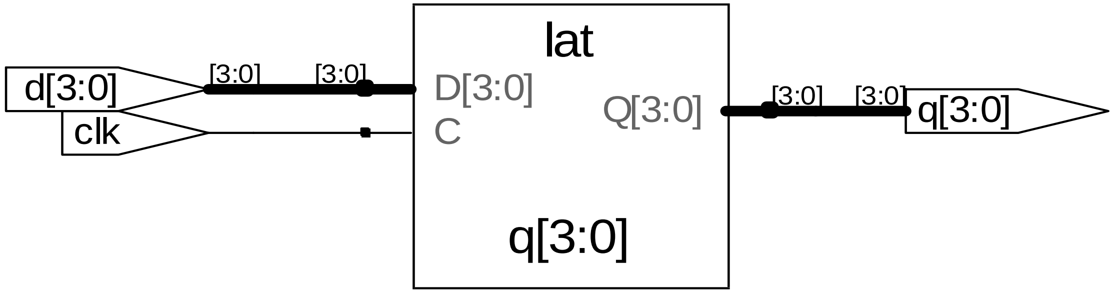
- **Enabled registers** respond to the clock **only** when the **enable is asserted**
- Asynchronously resettable enabled registers retain their old value if **both** **`reset`** **and** **`en`** **are FALSE**

Latch

```
module latch(input  logic clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;

endmodule
```



always_latch is equivalent to **always@(clk,d)**:
if clk is HIGH, d flows through to q (positive level sensitive)
otherwise, q keeps its old value

It is the preferred idiom for describing a SystemVerilog latch: a warning is generated if the always_latch block is not a latch

More combinational logic

- SystemVerilog **`always`** statements are used to describe sequential circuits, because they remember the old state when no new state is prescribed

- We used assignment statements to describe combinational logic behaviorally

- **`always`** statements can also be used to describe combinational logic behaviorally, but:
  - the **sensitivity list** MUST respond **to changes in all of the inputs**
  - all **outputs** should have **default values** or must be **assigned for every input combination**, otherwise a latch will be generated to hold the current value

Blocking vs. Nonblocking Assignment

- SystemVerilog supports **blocking** and **nonblocking** assignments in an always statement

- = is **blocking** assignment:
  - A group of **blocking** assignments are **evaluated in the order** in which they appear in the code, just as one would expect in a standard programming language

- <= is **nonblocking** assignment:
  - A group of **nonblocking** assignments are evaluated **concurrently**
  - all of the statements are evaluated before any of the signals on the left hand sides are updated

Blocking vs. Nonblocking Assignment

```
module inv(input  logic [3:0] a,
              output logic [3:0] y);
 always_comb
   y = ~a;
endmodule
```

- `always_comb` reevaluates the statements inside the `always` statement any time any of the signals on the right hand side of `<=` or `=` in the `always` statement change
- In this case, it is equivalent to `always @(a)`, but is better because it avoids mistakes if signals in the always statement are renamed or added
- If the code inside the always block is not combinational logic, SystemVerilog will report a warning

Blocking vs. Nonblocking Assignment

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
 logic p, g;
 always_comb
  begin
   p = a ^ b;              // blocking
   g = a & b;              // blocking
   s = p ^ cin;            // blocking
   cout = g | (p & cin); // blocking
  end
endmodule
```

- In this case, `always@(a,b,cin)` would have been equivalent to `always_comb` but `always_comb` is better because it avoids missing signals in the sensitivity list
- this example uses **blocking assignments**, first computing `p`, then `g`, then `s`, and finally `cout`, that are better for combinational logic

Other Behavioral Statements

- `case` and `if` statements are convenient for modeling more complicated combinational logic
- `case` and `if` statements MUST appear within `always` statements

- `case` statement
  - **performs different actions depending on the value of its input**
  - implies **combinational** logic **only if** all possible input combinations described
  - need to use `default` statement, **otherwise** it implies **sequential** logic, because the output will keep its old value in the undefined cases

Combinational Logic using `case`

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case (data)
      //                   abc_defg
      0:  segments =       7'b111_1110;
      1:  segments =       7'b011_0000;
      2:  segments =       7'b110_1101;
      3:  segments =       7'b111_1001;
      4:  segments =       7'b011_0011;
      5:  segments =       7'b101_1011;
      6:  segments =       7'b101_1111;
      7:  segments =       7'b111_0000;
      8:  segments =       7'b111_1111;
      9:  segments =       7'b111_0011;
      default: segments = 7'b000_0000; //required
    endcase
endmodule
```

- The **`case`** statement checks the value of data and performs the action after the colon
- The **`default`** clause is used to define the output for all cases not explicitly listed, guaranteeing combinational logic

Other Behavioral Statements

- `always` statements may also contain `if` **statements**
- The if statement may be followed by an `else` statement
- If **all possible input** combinations are handled, the statement implies **combinational** logic; **otherwise**, it produces **sequential** logic

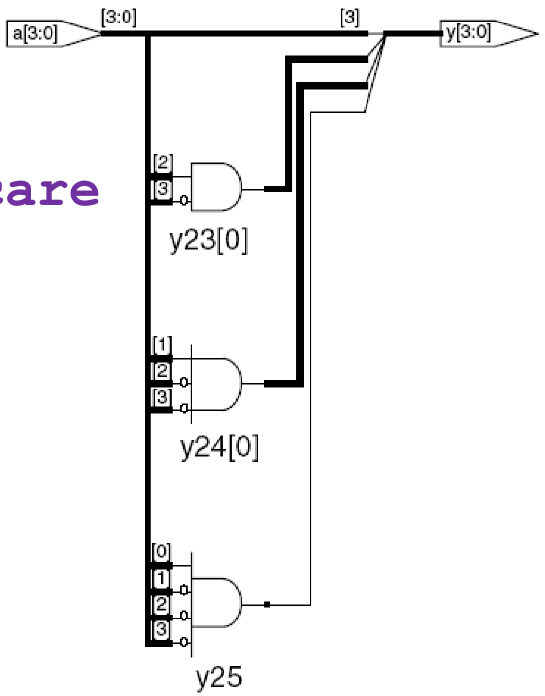
```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);
 always_comb
  if (a[3]) y = 4'b1000;
  else if (a[2]) y = 4'b0100;
  else if (a[1]) y = 4'b0010;
  else if (a[0]) y = 4'b0001;
  else y = 4'b0000;
endmodule
```

priority circuit: N-input priority circuit sets the output bit TRUE that corresponds to the most significant input that is TRUE

Combinational Logic using casez

- Truth tables may include **don't care**'s to allow simplification
- A priority circuit can be defined using don't cares
- The **casez** statement acts like a case statement but it also **recognizes ? as don't care**

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000; // ?=don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
   endcase
endmodule
```


a[3:0]
[3:0]
[3]
y[3:0]
[2]
[3]
y23[0]
[1]
[2]
[3]
y24[0]
[0]
[1]
[2]
[3]
y25