CESE4005: Hardware Fundamentals

2024-2025, lecture 5

Introduction to Verilog: Part 2

Computer Engineering Lab

Faculty of Electrical Engineering, Mathematics & Computer Science

2024-2025

Anteneh Gebregiorgis

TU Delft

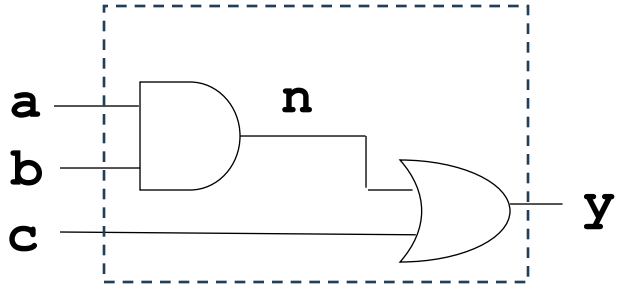
Delft University of Technology

Overview

- Recap
- Usage of the always statement
- More on blocking and nonblocking assignments
- Demo circuits
- D Flip-flop
- Finite State Machines
- Counters
- Adders
- Decoders

Recap

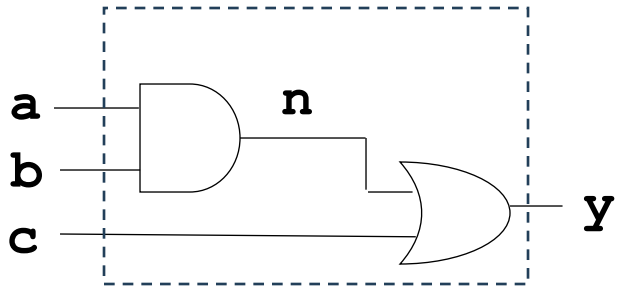
```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule
```



Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule
```

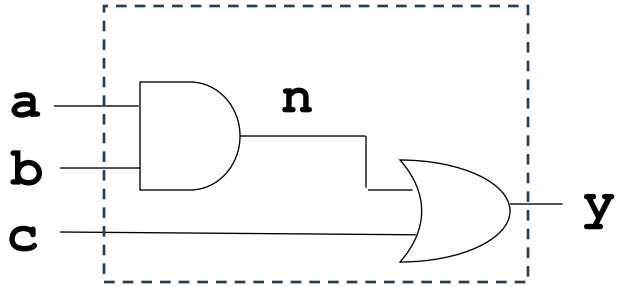
Order is not important



Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

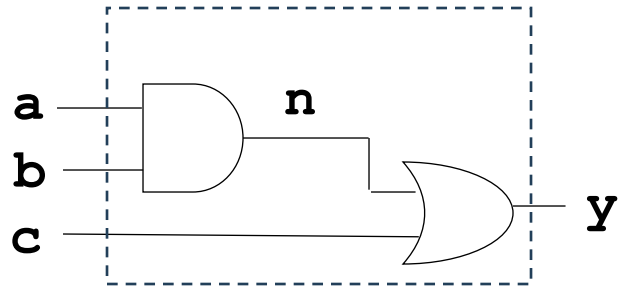
Order is not important



Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

Order is not important



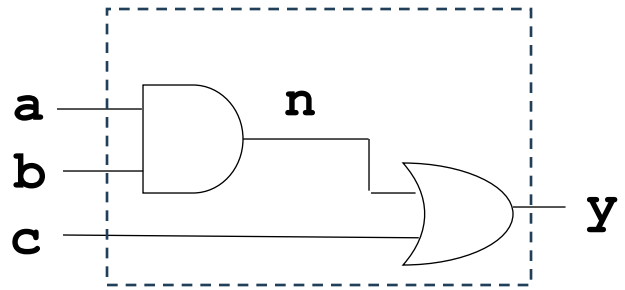
Here, order is important. Starting from begin, statements are executed one after another (with delays)

Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

Order is not important

Simulation with testbench



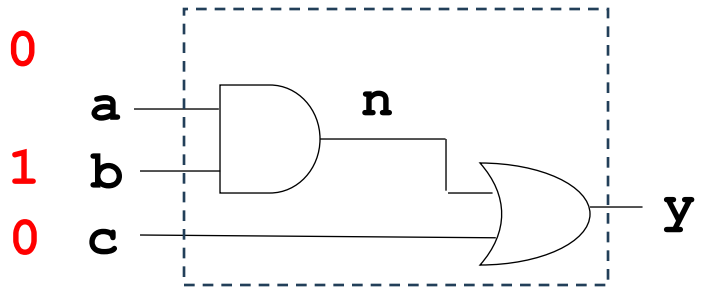
Here, order is important.
Starting from begin, statements are executed one after another (with delays)

Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

Order is not important

Simulation with testbench



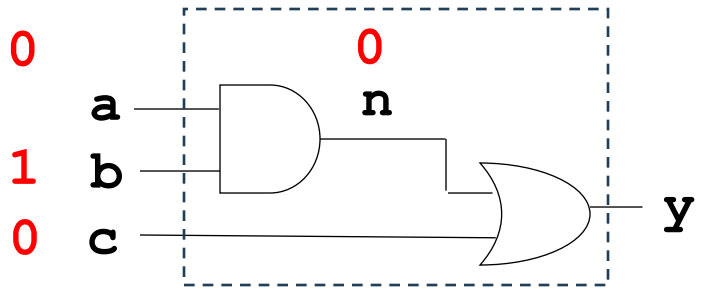
Here, order is important.
Starting from begin, statements are executed one after another (with delays)

Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

Order is not important

Simulation with testbench



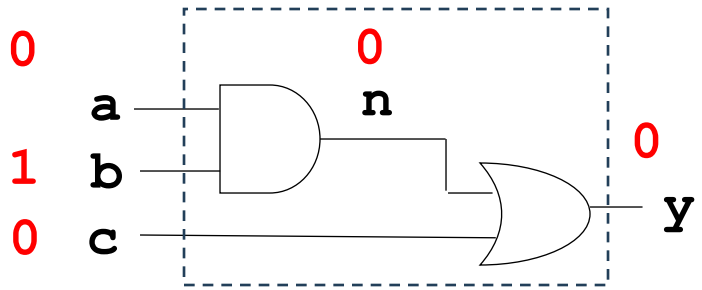
Here, order is important.
Starting from begin, statements are executed one after another (with delays)

Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

Order is not important

Simulation with testbench



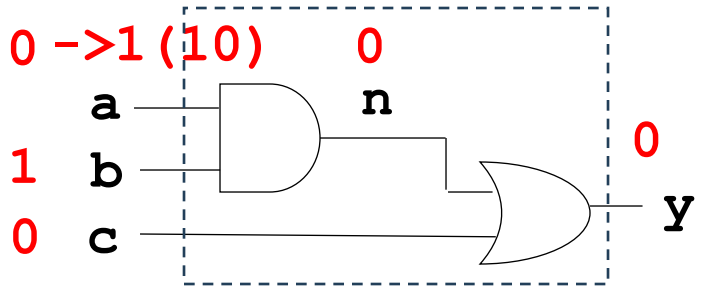
Here, order is important.
Starting from begin, statements are executed one after another (with delays)

Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

Order is not important

Simulation with testbench



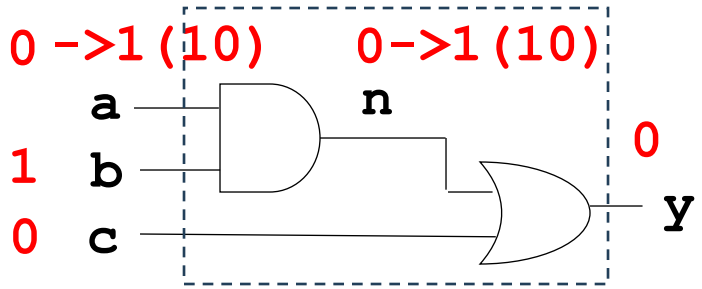
Here, order is important.
Starting from begin, statements are executed one after another (with delays)

Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

Order is not important

Simulation with testbench



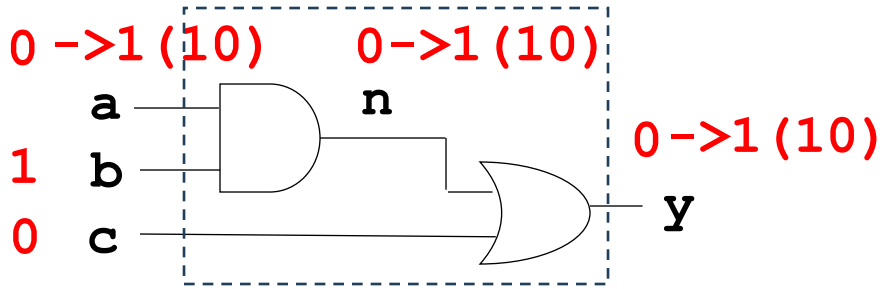
Here, order is important.
Starting from begin, statements are executed one after another (with delays)

Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

Order is not important

Simulation with testbench



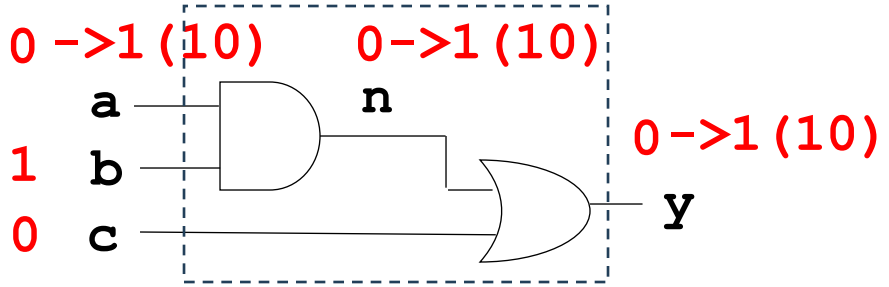
Here, order is important.
Starting from begin, statements are executed one after another (with delays)

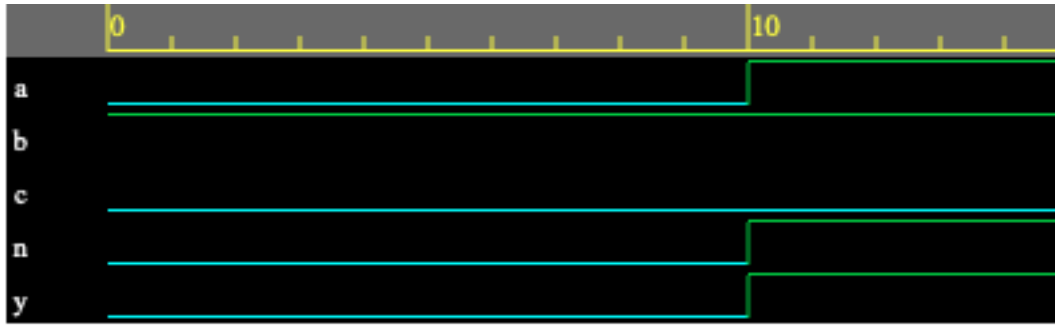
Recap

```
module func1(input a, b, c,
             output y);
   wire n;

   assign y = n | c;
   assign n = a & b;
endmodule

module testfunc1();
   reg a, b, c;
   wire y;

   func1 dut(a, b, c, y);

   initial begin
       a = 0; b = 1; c = 0;
       #10; a = 1;
   end
endmodule
```

Order is not important

Simulation with testbench





Here, order is important.
Starting from begin,
statements are executed one after another (with delays)

Using the always block

In general **`always`** block can also be used with/without sensitivity list

When it has no sensitivity list, the block is evaluated whenever one of the module inputs change.

Using the always block

In general **`always`** block can also be used with/without sensitivity list

When it has no sensitivity list, the block is evaluated whenever one of the module inputs change.

When it has a sensitivity list, the block is evaluated whenever one of the signal in the sensitivity list changes value.

```
always @(a, b, cin)
```

Using the always block

In general **`always`** block can also be used with/without sensitivity list

When it has no sensitivity list, the block is evaluated whenever one of the module inputs change.

When it has a sensitivity list, the block is evaluated whenever one of the signal in the sensitivity list changes value.

```
always @(a, b, cin)
```

Where signal names in the sensitivity list may be preceded by **`posedge`** or **`negedge`** to indicate that evaluation is triggered by only a rising edge or falling edge event .

Using the always block

In general **`always`** block can also be used with/without sensitivity list

When it has no sensitivity list, the block is evaluated whenever one of the module inputs change.

When it has a sensitivity list, the block is evaluated whenever one of the signal in the sensitivity list changes value.

```
always @(a, b, cin)
```

Where signal names in the sensitivity list may be preceded by **`posedge`** or **`negedge`** to indicate that evaluation is triggered by only a rising edge or falling edge event **.**

For flip-flops (registers), latches and combinational circuits the usage of respectively **`always_ff`, `always_latch`** and **`always_comb`** is preferred.

Using always block to describe a combinational circuit

An **`always_comb`** block can be used to describe a combinational circuit.

Using always block to describe a combinational circuit

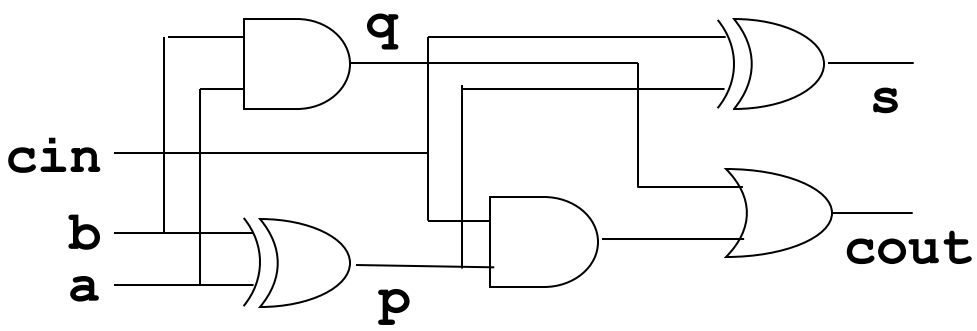
An **`always_comb`** block can be used to describe a combinational circuit.

```
module FA(input a, b, cin,
                output s, cout);
wire p,q;
always_comb
begin
   p = a ^ b;
   g = a & b;
   s = p ^ cin;
   cout = g | (p & cin);
end
endmodule
```

Using always block to describe a combinational circuit

An **always_comb** block can be used to describe a combinational circuit.

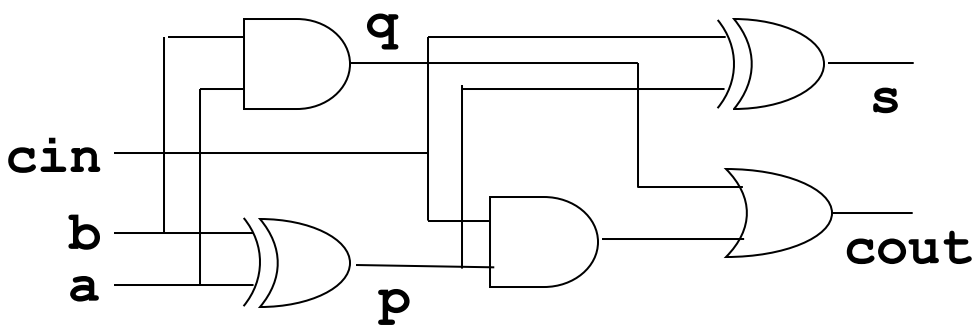
```
module FA(input a, b, cin,
                output s, cout);
wire p,q;
always_comb
begin
   p = a ^ b;
   g = a & b;
   s = p ^ cin;
   cout = g | (p & cin);
end
endmodule
```



Using always block to describe a combinational circuit

An **`always_comb`** block can be used to describe a combinational circuit.

```
module FA(input a, b, cin,
                output s, cout);
wire p,q;
always_comb
begin
   p = a ^ b;
   g = a & b;
   s = p ^ cin;
   cout = g | (p & cin);
end
endmodule
```

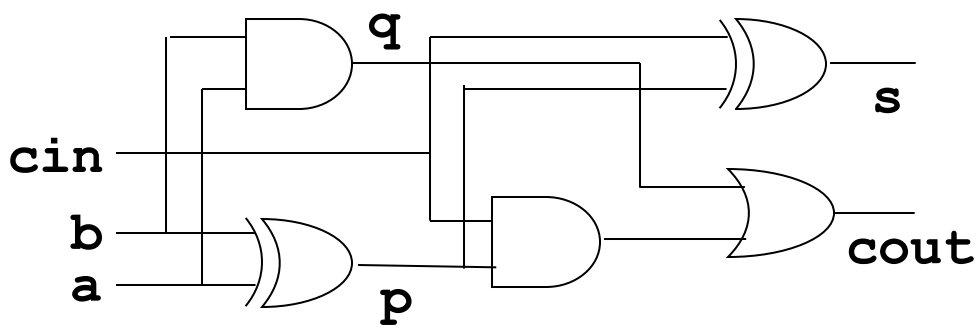


The block is evaluated whenever one of the inputs (`a, b or cin`) change.

Using always block to describe a combinational circuit

An **`always_comb`** block can be used to describe a combinational circuit.

```
module FA(input a, b, cin,
                output s, cout);
wire p,q;
always_comb
begin
   p = a ^ b;
   g = a & b;
   s = p ^ cin;
   cout = g | (p & cin);
end
endmodule
```



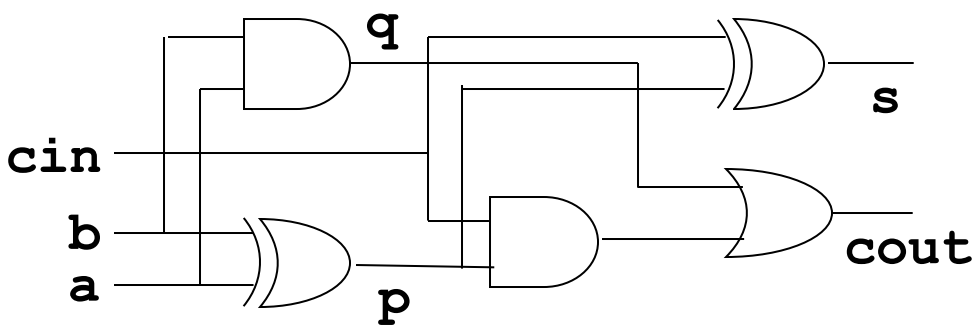
The block is evaluated whenever one of the inputs (`a, b or cin`) change.

It is important that you specify the value of all outputs in all cases.
Otherwise, unwanted latches may be created during synthesis.

Using always block to describe a combinational circuit

An **always_comb** block can be used to describe a combinational circuit.

```
module FA(input a, b, cin,
                output s, cout);
wire p,q;
always_comb
begin
   p = a ^ b;
   g = a & b;
   s = p ^ cin;
   cout = g | (p & cin);
end
endmodule
```



The block is evaluated whenever one of the inputs (a, b or cin) change.

It is important that you specify the value of all outputs in all cases.
Otherwise, unwanted latches may be created during synthesis.

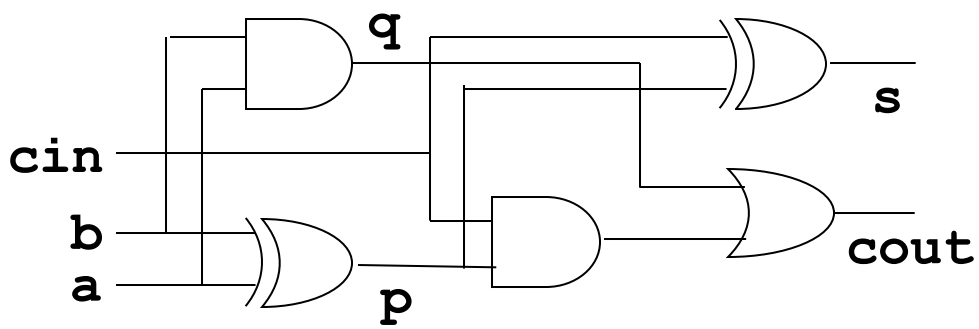
```
always_comb begin
   if (s = 1)begin
   y = x1;
   else y = x0;
   end
end
```

ok

Using always block to describe a combinational circuit

An **always_comb** block can be used to describe a combinational circuit.

```
module FA(input a, b, cin,
                output s, cout);
wire p,q;
always_comb
begin
   p = a ^ b;
   g = a & b;
   s = p ^ cin;
   cout = g | (p & cin);
end
endmodule
```



The block is evaluated whenever one of the inputs (a, b or cin) change.

It is important that you specify the value of all outputs in all cases.
Otherwise, unwanted latches may be created during synthesis.

```
always_comb begin
   if (s = 1)begin
   y = x1;
   else y = x0;
   end
end
```

ok

```
always_comb begin
   if (s = 1)begin
    y = x1;
   end
end
```

wrong

Blocking vs. Nonblocking Assignment

In an always block, there is an important difference between using a <= or a = in an assignment.

Blocking vs. Nonblocking Assignment

In an always block, there is an important difference between using a <= or a = in an assignment.

- <= is a **nonblocking** assignment: **do not block** the execution of the next statements. The order of the assignments is not important.
- **=** is a **blocking** assignment. **blocks** the execution of the next statements. The order of the assignments is important.

Blocking vs. Nonblocking Assignment

In an always block, there is an important difference between using a <= or a = in an assignment.

- <= is a **nonblocking** assignment: **do not block** the execution of the next statements. The order of the assignments is not important.
- **=** is a **blocking** assignment. **blocks** the execution of the next statements. The order of the assignments is important.

```
// nonblocking assignments
module nonblockeg(input clk,d,
                  output q);
   wire n1;
   always_ff @(posedge clk)
   begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
   end
endmodule
```

```
// blocking assignments
module blockeg(input clk, d,
                  output q);
   wire n1;
   always_ff @(posedge clk)
   begin
      n1 = d;  // blocking
      q  = n1; // blocking
   end
endmodule
```

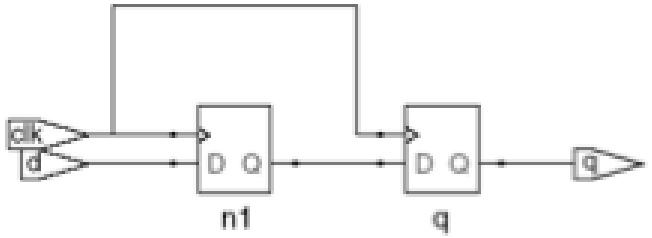
Blocking vs. Nonblocking Assignment

In an always block, there is an important difference between using a <= or a = in an assignment.

- <= is a **nonblocking** assignment: **do not block** the execution of the next statements. The order of the assignments is not important.
- **=** is a **blocking** assignment. **blocks** the execution of the next statements. The order of the assignments is important.

```
// nonblocking assignments
module nonblockeg(input clk,d,
                  output q);
   wire n1;
   always_ff @(posedge clk)
   begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
   end
endmodule
```

```
// blocking assignments
module blockeg(input clk, d,
                  output q);
   wire n1;
   always_ff @(posedge clk)
   begin
      n1 = d;  // blocking
      q  = n1; // blocking
   end
endmodule
```

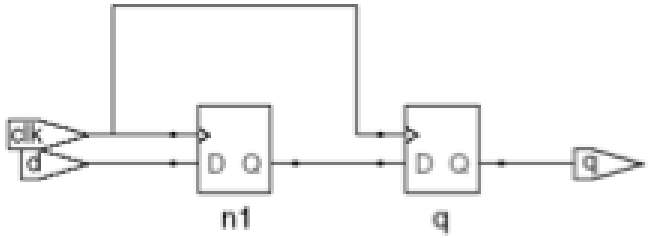


Blocking vs. Nonblocking Assignment

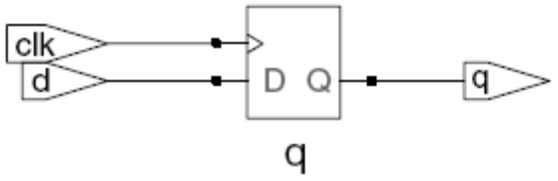
In an always block, there is an important difference between using a <= or a = in an assignment.

- <= is a **nonblocking** assignment: **do not block** the execution of the next statements. The order of the assignments is not important.
- **=** is a **blocking** assignment. **blocks** the execution of the next statements. The order of the assignments is important.

```
// nonblocking assignments
module nonblockeg(input clk,d,
                  output q);
   wire n1;
   always_ff @(posedge clk)
   begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
   end
endmodule
```




```
// blocking assignments
module blockeg(input clk, d,
                  output q);
   wire n1;
   always_ff @(posedge clk)
   begin
      n1 = d;  // blocking
      q  = n1; // blocking
   end
endmodule
```



Blocking vs. Nonblocking Assignment

For **`always_com`** blocks it is ok to use blocking assignments:

Blocking vs. Nonblocking Assignment

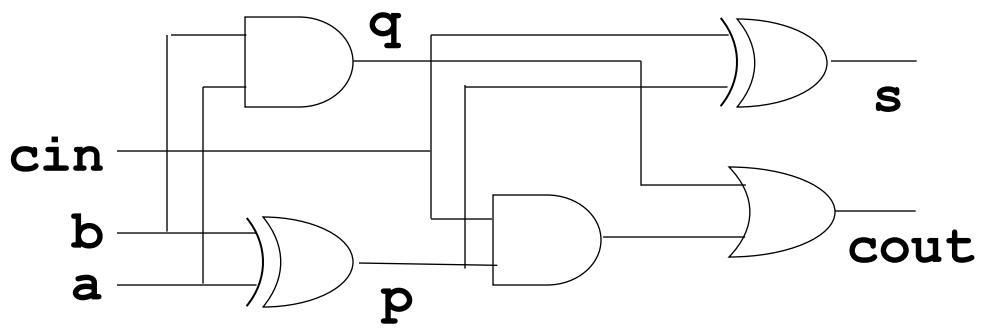
For **`always_com`** blocks it is ok to use blocking assignments:

```
module FA(input a, b, cin,
                output s, cout);
wire p,q;
always_comb
begin
   p = a ^ b;
   g = a & b;
   s = p ^ cin;
   cout = g | (p & cin);
end
endmodule
```

Blocking vs. Nonblocking Assignment

For **always_com** blocks it is ok to use blocking assignments:

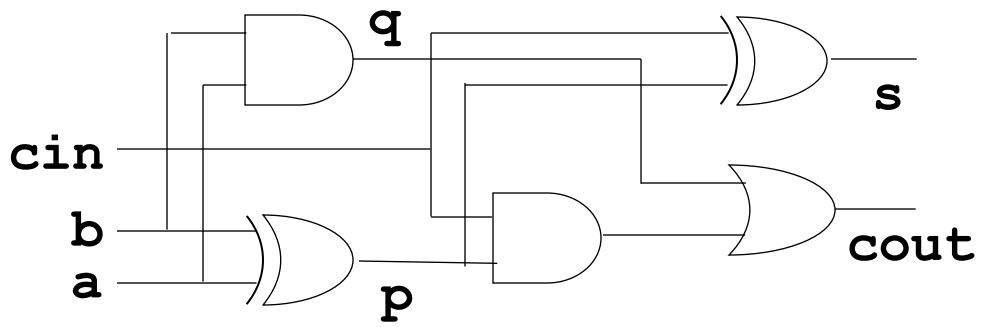
```
module FA(input a, b, cin,
                output s, cout);
wire p,q;
always_comb
begin
   p = a ^ b;
   g = a & b;
   s = p ^ cin;
   cout = g | (p & cin);
end
endmodule
```



Blocking vs. Nonblocking Assignment

For **always_com** blocks it is ok to use blocking assignments:

```
module FA(input a, b, cin,
                output s, cout);
wire p,q;
always_comb
begin
   p = a ^ b;
   g = a & b;
   s = p ^ cin;
   cout = g | (p & cin);
end
endmodule
```



Best practice:

➔ Only use non-blocking assignment for sequential logic (e.g., FF)

➔ Mainly use blocking assignment for combinational logic

Rules for Signal Assignment

- **Synchronous sequential logic:** use `always_ff @(posedge clk)` and nonblocking assignments (`<=`)

Rules for Signal Assignment

- **Synchronous sequential logic:** use `always_ff @(posedge clk)` and nonblocking assignments (`<=`)

```
always_ff @(posedge clk)
  q <= d; // nonblocking
```

Rules for Signal Assignment

- **Synchronous sequential logic:** use `always_ff @(posedge clk)` and nonblocking assignments (`<=`)

```
always_ff @(posedge clk)
  q <= d; // nonblocking
```

- **Simple combinational logic:** use continuous assignments (`assign`…)

```
assign y = a & b;
```

Rules for Signal Assignment

- **Synchronous sequential logic:** use **`always_ff @(posedge clk)`** and nonblocking assignments (**`<=`**)

```
always_ff @(posedge clk)
  q <= d; // nonblocking
```

- **Simple combinational logic:** use continuous assignments (**`assign`**…)

```
assign y = a & b;
```

- **More complicated combinational logic:** use **`always_comb`** and blocking assignments (**`=`**)
- In an **`always_comb`** block, assign a value to *each* output for *all* input combinations.

Rules for Signal Assignment

- **Synchronous sequential logic:** use **`always_ff @(posedge clk)`** and nonblocking assignments (**`<=`**)

```
always_ff @(posedge clk)
  q <= d; // nonblocking
```

- **Simple combinational logic:** use continuous assignments (**`assign`**…)

```
assign y = a & b;
```

- **More complicated combinational logic:** use **`always_comb`** and blocking assignments (**`=`**)
- In an **`always_comb`** block, assign a value to *each* output for *all* input combinations.
- Assign a signal in *only one* **`always`** statement or continuous assignment statement. ➔ ***avoids conflicting signal assignment***

Demo circuits: seven segment display

A combinational circuit for a seven-segment display decoder that uses a case statement

Demo circuits: seven segment display

A combinational circuit for a seven-segment display decoder that uses a case statement

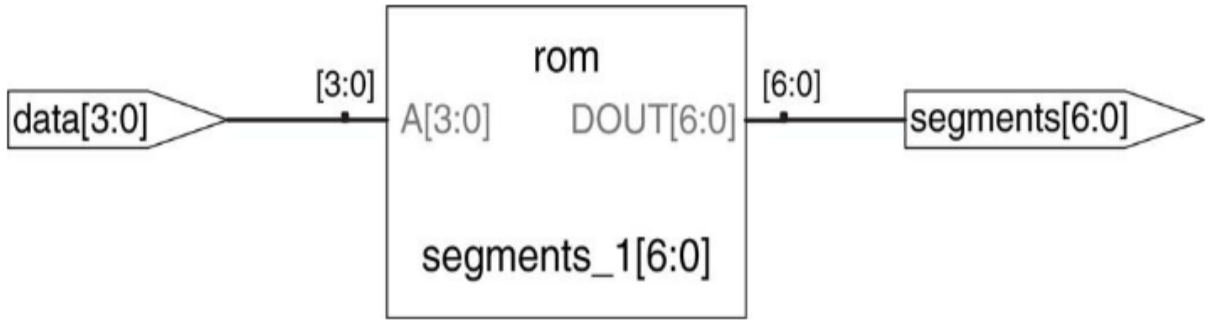
```
module sevenseg(input [3:0] data,
                output [6:0] segments);
always_comb
begin
   case(data)
        0:      segments = 7'b111_1110;
        1:      segments = 7'b011_0000;
        2:      segments = 7'b110_1101;
        3:      segments = 7'b111_1001;
        4:      segments = 7'b011_0011;
        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;
   endcase
end
endmodule
```

Demo circuits: seven segment display

A combinational circuit for a seven-segment display decoder that uses a case statement

```
module sevenseg(input [3:0] data,
                output [6:0] segments);
always_comb
begin
   case(data)
        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;
   endcase
end
endmodule
```

Synthesis:



Demo circuits: priority circuit

A priority circuit that uses a nested if-else statement

Demo circuits: priority circuit

A priority circuit that uses a nested if-else statement

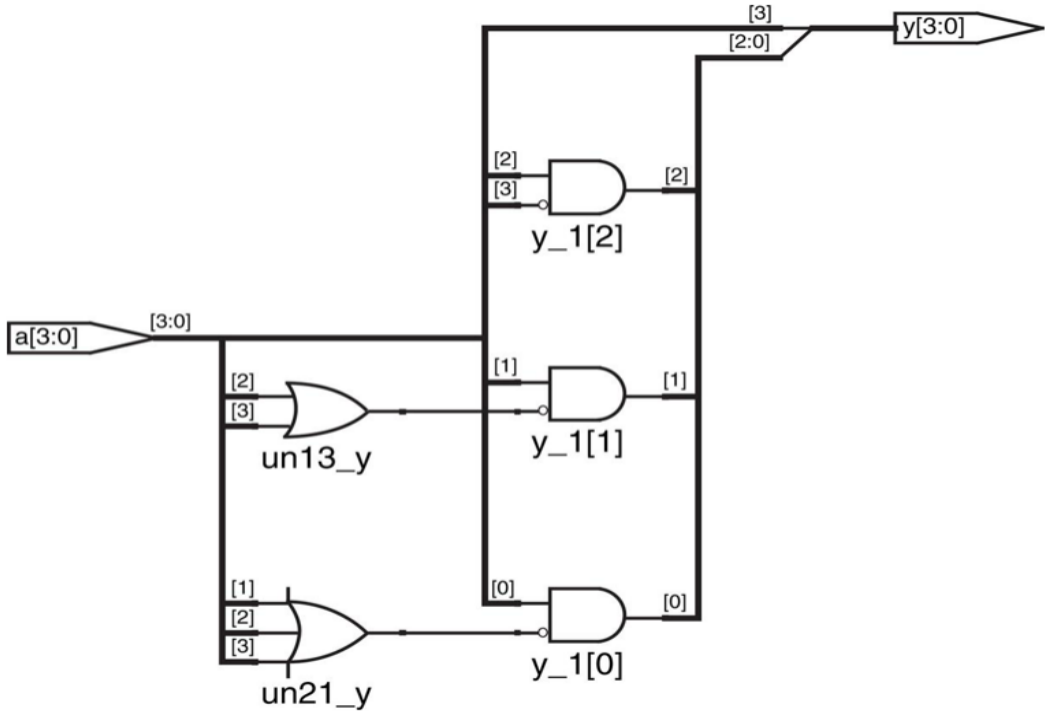
```
module priorityckt(input [3:0] a,
                   output [3:0] y);

   always_comb
begin
      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;
end
endmodule
```

Demo circuits: priority circuit

A priority circuit that uses a nested if-else statement

```
module priorityckt(input [3:0] a,
                   output [3:0] y);

   always_comb
begin
      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;
end
endmodule
```

Synthesis:



Demo circuits: priority circuit with don’t cares

- Truth tables may include don’t cares to allow more logic simplification.
- The following shows how to describe the previous priority circuit with a case statement, which allows don’t cares to be used.

Demo circuits: priority circuit with don't cares

- Truth tables may include don't cares to allow more logic simplification.
- The following shows how to describe the previous priority circuit with a case statement, which allows don't cares to be used.

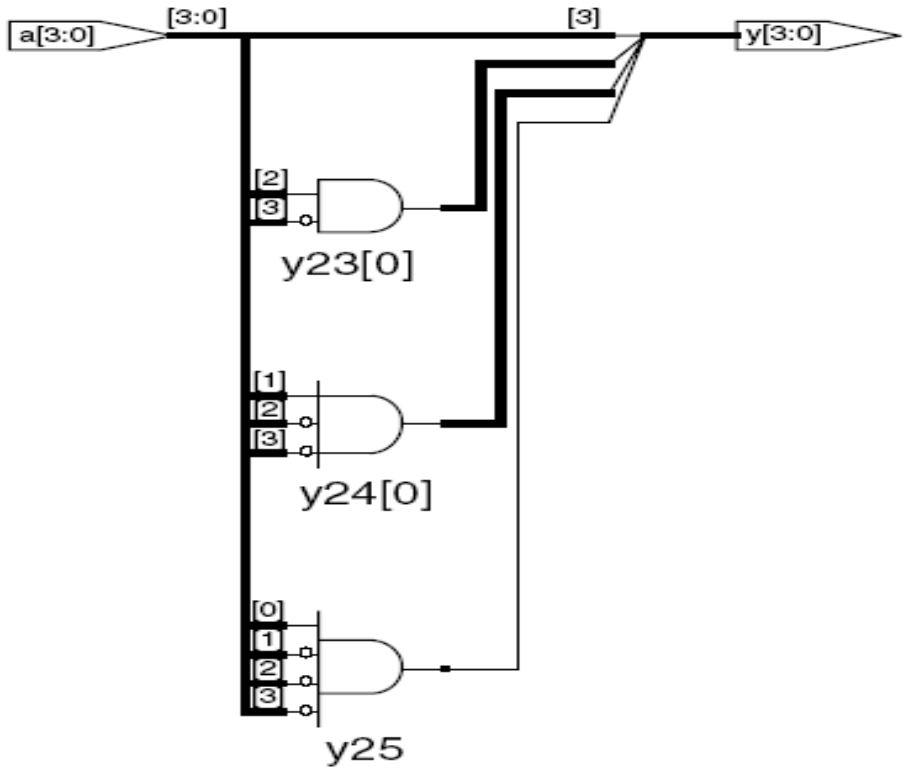
```
module priority_case(input [3:0] a,
                     output [3:0] y);
  always_comb
begin
    case(a)
      4'b1???: y = 4'b1000; // ? = don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
   endcase
end
endmodule
```

Demo circuits: priority circuit with don't cares

- Truth tables may include don't cares to allow more logic simplification.
- The following shows how to describe the previous priority circuit with a case statement, which allows don't cares to be used.

```
module priority_case(input [3:0] a,
                     output [3:0] y);
  always_comb
begin
    case(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
end
endmodule
```

Synthesis:

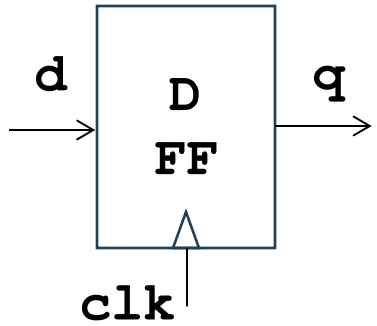


Describing a D flip-flop

```
module dff(input clk,
           input d,
           output q);

   always_ff @(posedge clk)   // on a rising clock edge
      q <= d;                 // q gets the value of d
endmodule
```

Describing a D flip-flop

```
module dff(input clk,
           input d,
           output q);

   always_ff @(posedge clk)   // on a rising clock edge
      q <= d;                 // q gets the value of d
endmodule
```



Describing a D flip-flop with reset

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

Describing a D flip-flop with reset

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

with synchronous reset, the reset
is done only on rising clock edge

```
always_ff @(posedge clk)
   if (reset = 1) q <= 0;
   else q <= d;
```

Describing a D flip-flop with reset

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

with synchronous reset, the reset is done only on rising clock edge

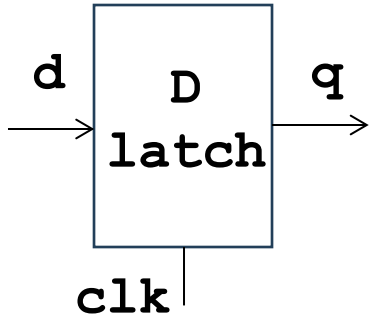
```
always_ff @(posedge clk)
   if (reset = 1) q <= 0;
   else q <= d;
```

with asynchronous reset, the reset is done any moment reset becomes 1

```
always_ff @(posedge clk, posedge reset)
   if (reset = 1) q <= 0;
   else q <= d;
```

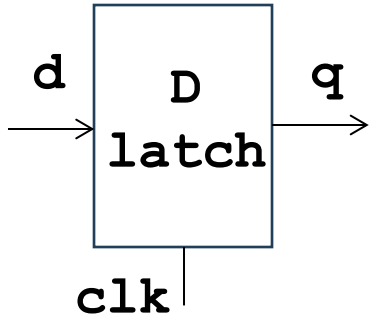
Describing a latch

The block **`always_latch`** is used to describe a latch



Describing a latch

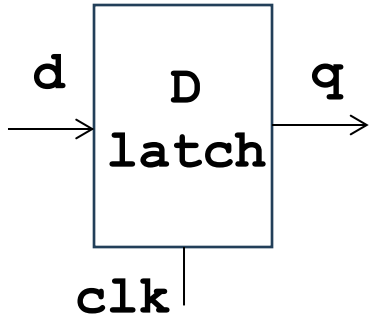
The block **always_latch** is used to describe a latch




```
always_latch
   if (clk) q <= d;    // when clk = 1, q gets value of d
                       // when clk = 0, q remembers its value
```

Describing a latch

The block **always_latch** is used to describe a latch

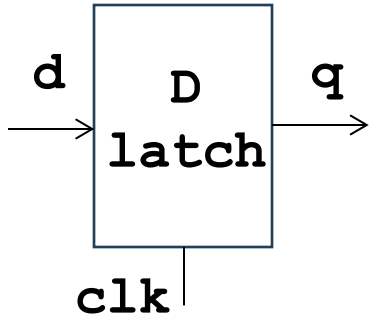



```
always_latch
   if (clk) q <= d;    // when clk = 1, q gets value of d
                       // when clk = 0, q remembers its value
```

The block **always_latch** is evaluated whenever one of the inputs (**clk** or **d**) changes value

Describing a latch

The block **always_latch** is used to describe a latch




```
always_latch
   if (clk) q <= d;    // when clk = 1, q gets value of d
                       // when clk = 0, q remembers its value
```

The block **always_latch** is evaluated whenever one of the inputs (**clk** or **d**) changes value

Normally, it is not a good idea to use latches in your circuit: they are transparent as long as clk = 1, so problematic combinational feedback loops may occur.

Registers

The following example shows a 4-bit register with asynchronous reset and enable.

It retains its old value if both reset and en are FALSE.

Registers

The following example shows a 4-bit register with asynchronous reset and enable.

It retains its old value if both reset and en are FALSE.

```
module register4bit(input clk, reset, en, [3:0] d,
                    output [3:0] q);

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

endmodule
```

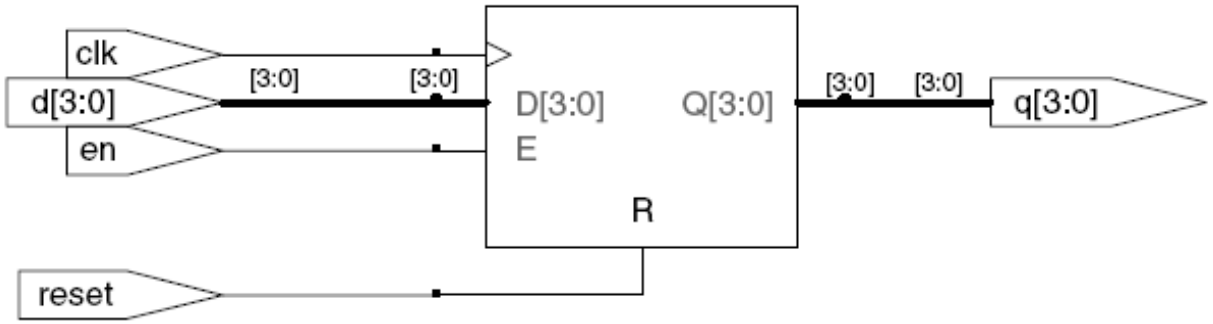
Registers

The following example shows a 4-bit register with asynchronous reset and enable.

It retains its old value if both reset and en are FALSE.

```
module register4bit(input clk, reset, en, [3:0] d,
                    output [3:0] q);

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

endmodule
```

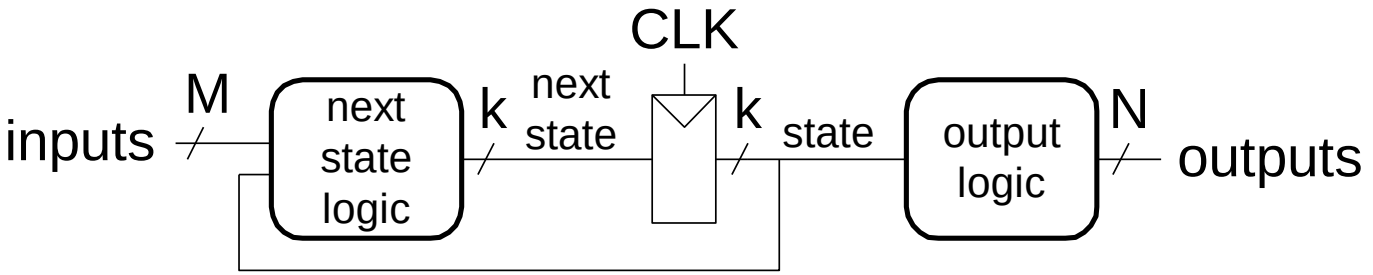
Synthesis:



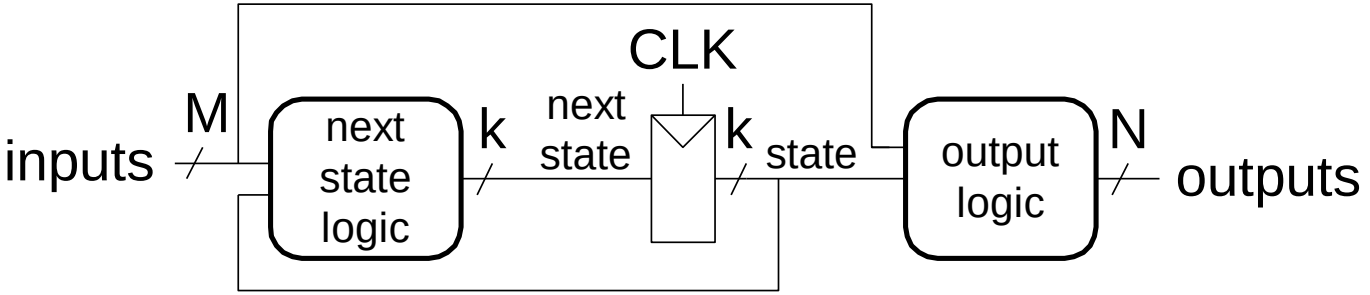
Finite State Machines

- **Three blocks:**
  - next state logic
  - state register
  - output logic

Moore FSM

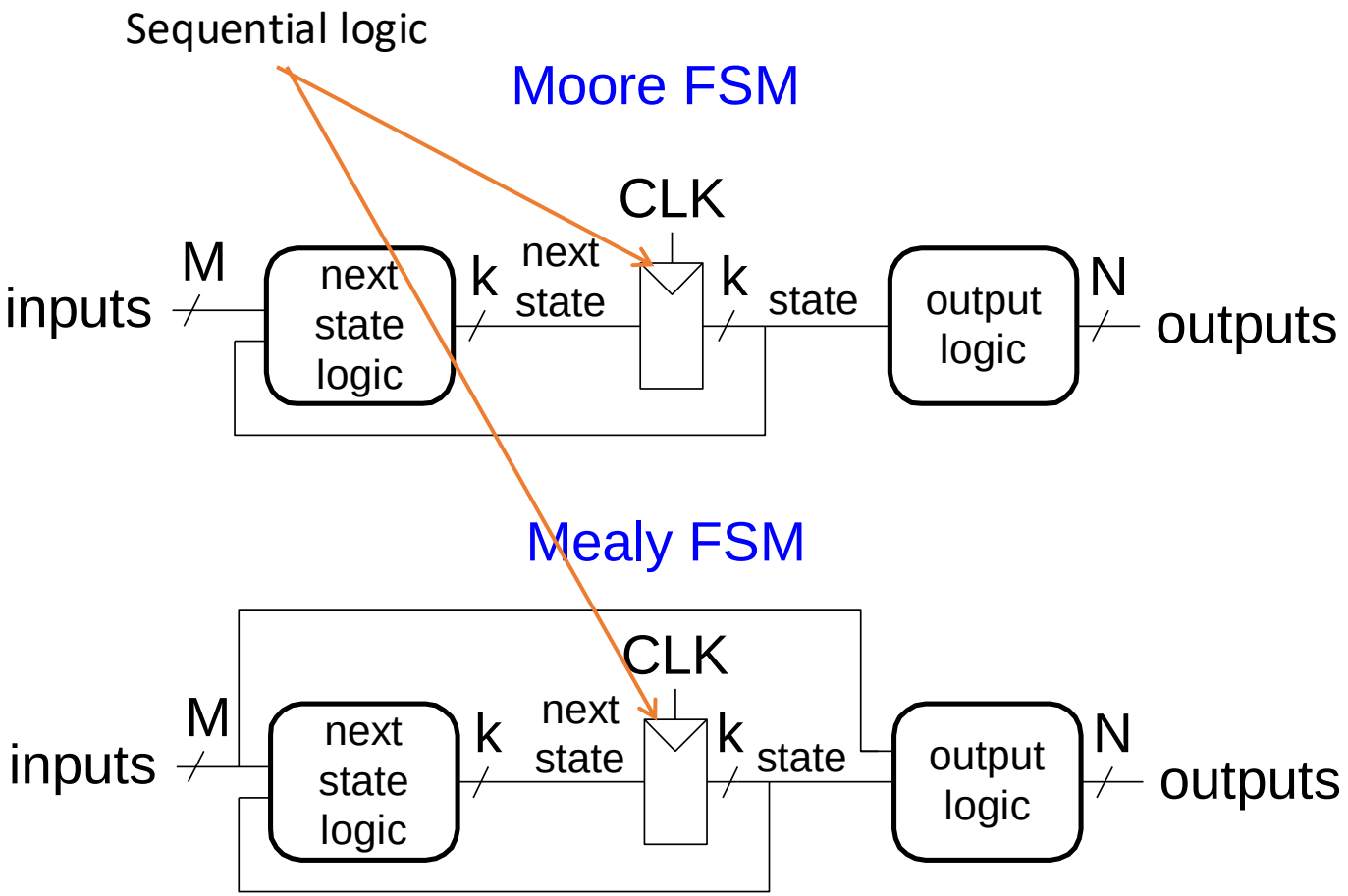


Mealy FSM



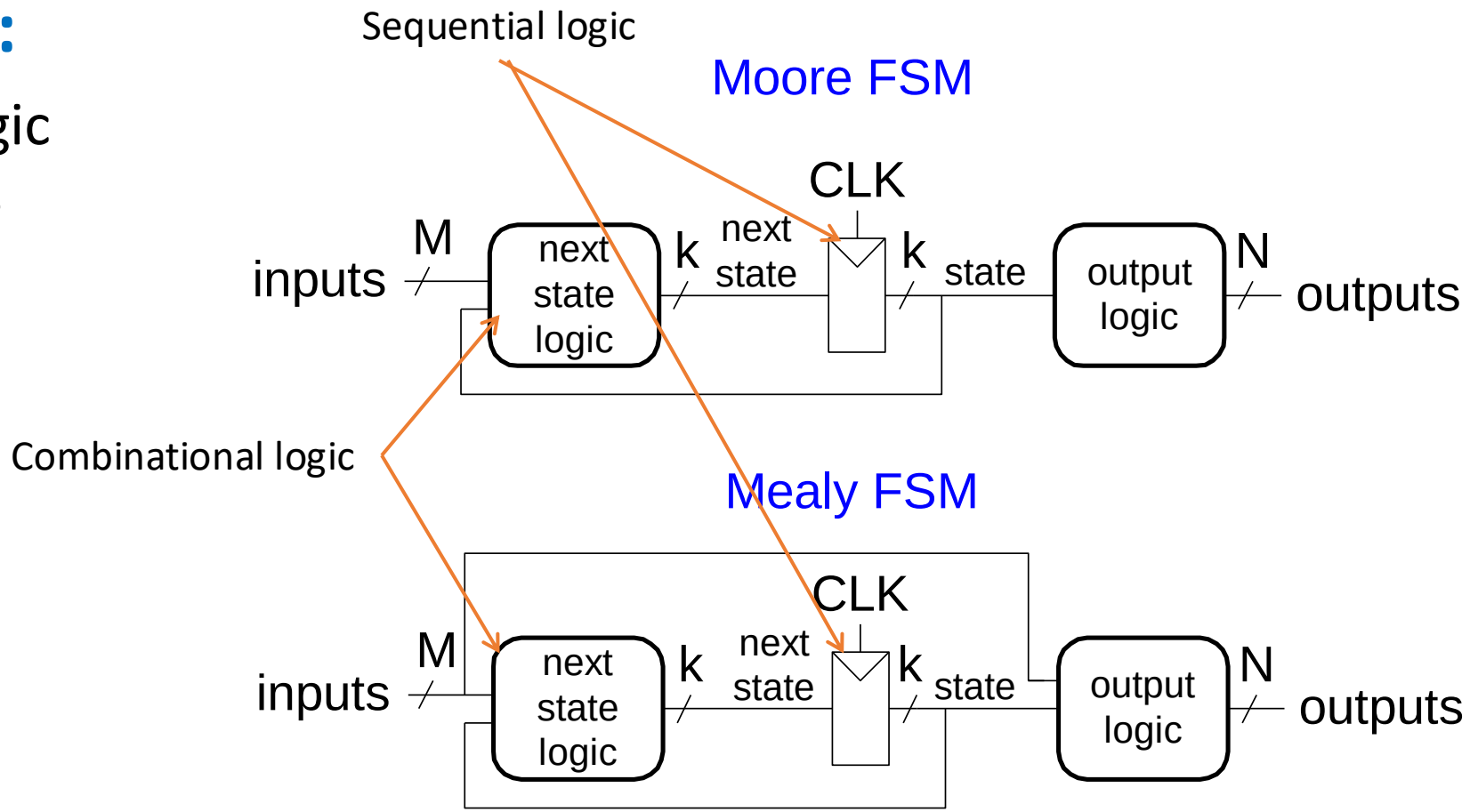
Finite State Machines

- **Three blocks:**
  - next state logic
  - state register
  - output logic



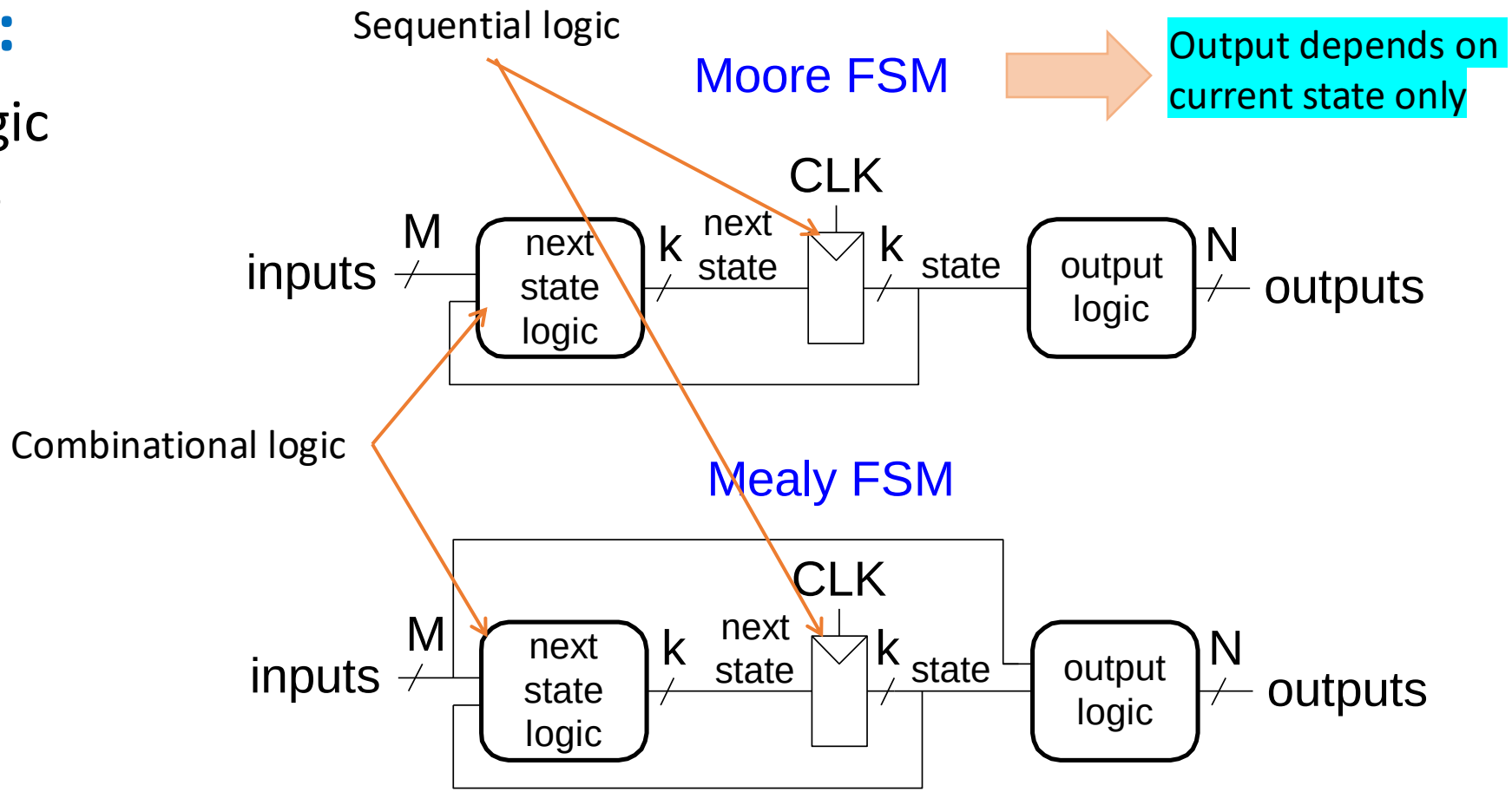
Finite State Machines

- **Three blocks:**
  - next state logic
  - state register
  - output logic



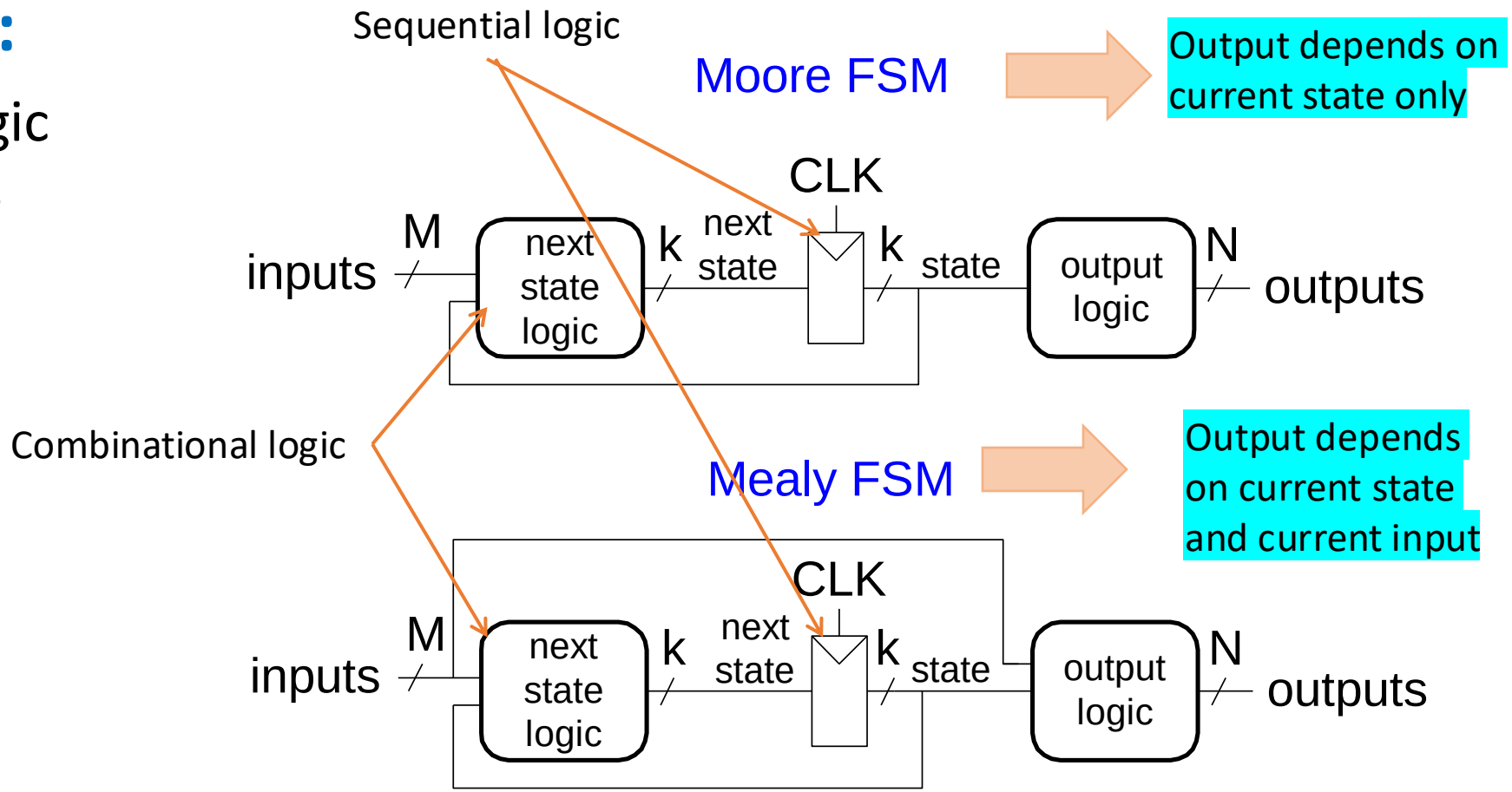
Finite State Machines

- **Three blocks:**
  - next state logic
  - state register
  - output logic



Finite State Machines

- **Three blocks:**
  - next state logic
  - state register
  - output logic



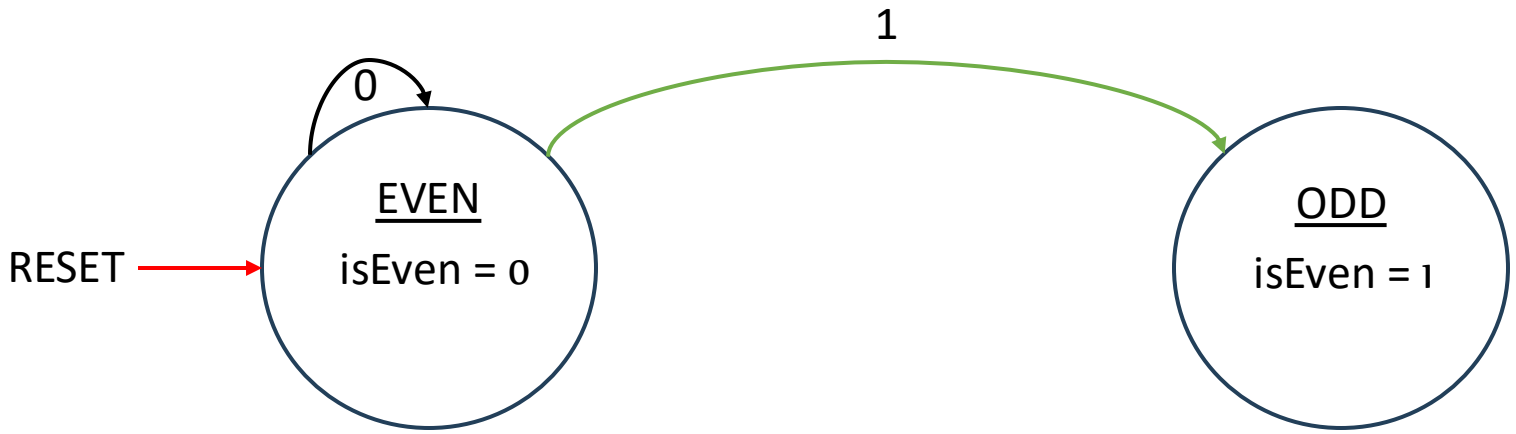
FSM Example 1: Even/Odd FSM

Time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Input	0	0	1	1	1	0	1	1
State	EVEN	EVEN	EVEN	ODD	EVEN	ODD	ODD	EVEN
Output	0	0	0	1	0	1	1	0

FSM Example 1: Even/Odd FSM

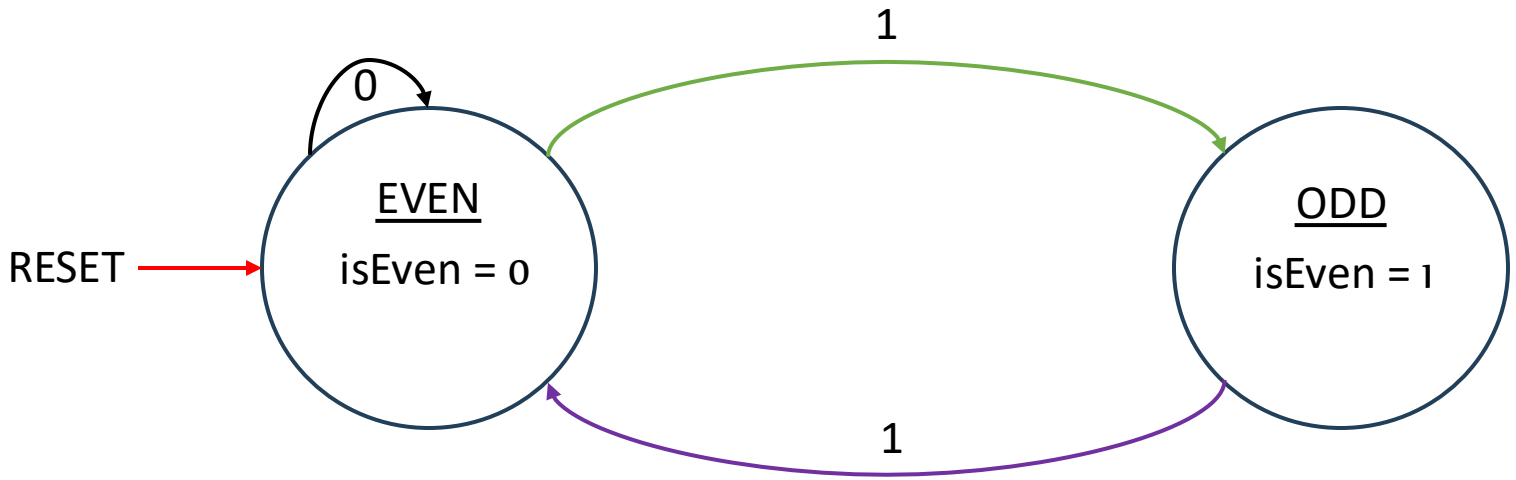
Time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Input	0	0	1	1	1	0	1	1
State	EVEN	EVEN	EVEN	ODD	EVEN	ODD	ODD	EVEN
Output	0	0	0	1	0	1	1	0

FSM Example 1: Even/Odd FSM



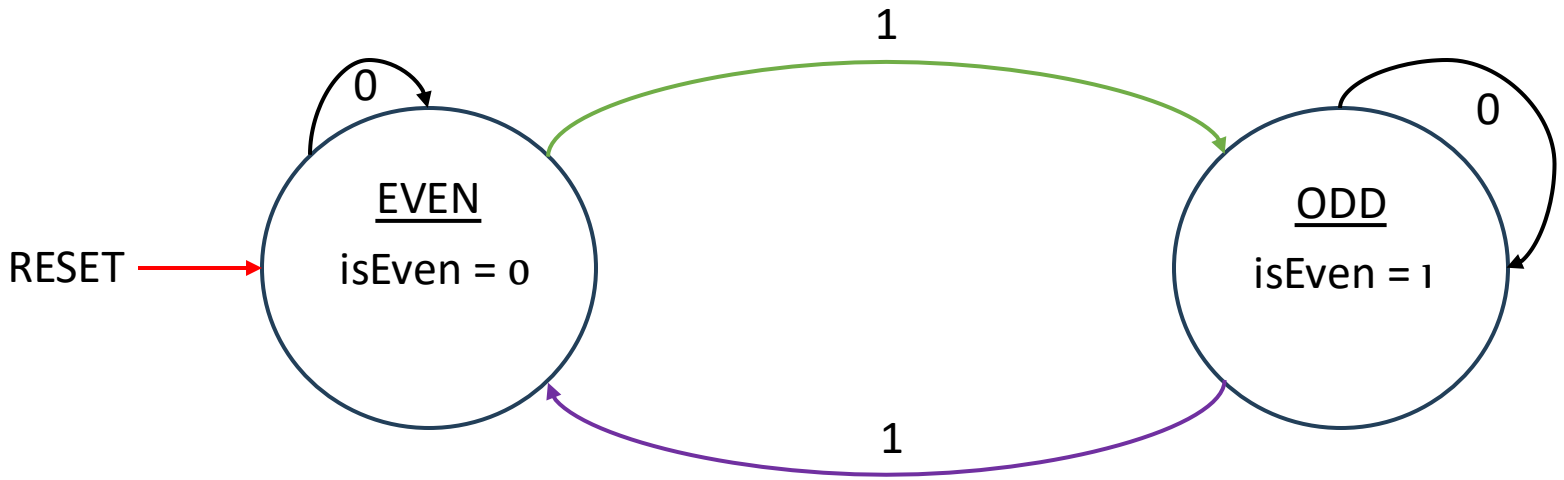
Time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Input	0	0	1	1	1	0	1	1
State	EVEN	EVEN	EVEN	ODD	EVEN	ODD	ODD	EVEN
Output	0	0	0	1	0	1	1	0

FSM Example 1: Even/Odd FSM



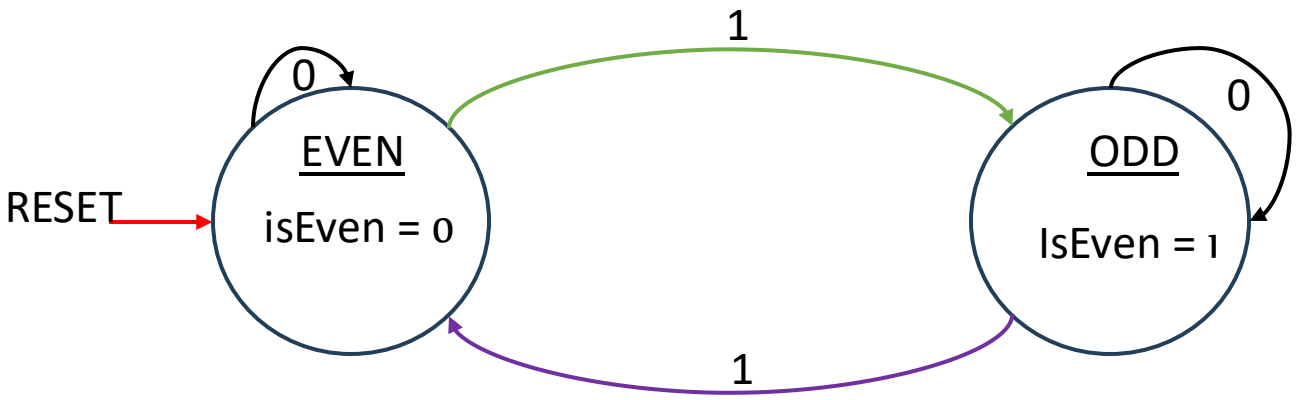
Time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Input	0	0	1	1	1	0	1	1
State	EVEN	EVEN	EVEN	ODD	EVEN	ODD	ODD	EVEN
Output	0	0	0	1	0	1	1	0

FSM Example 1: Even/Odd FSM



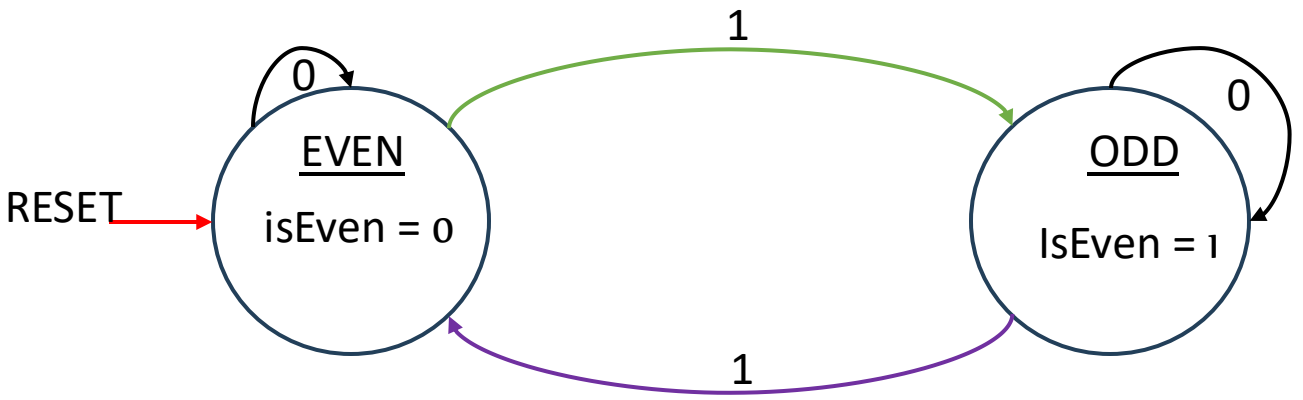
Time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Input	0	0	1	1	1	0	1	1
State	EVEN	EVEN	EVEN	ODD	EVEN	ODD	ODD	EVEN
Output	0	0	0	1	0	1	1	0

FSM Example 1: Even/Odd FSM



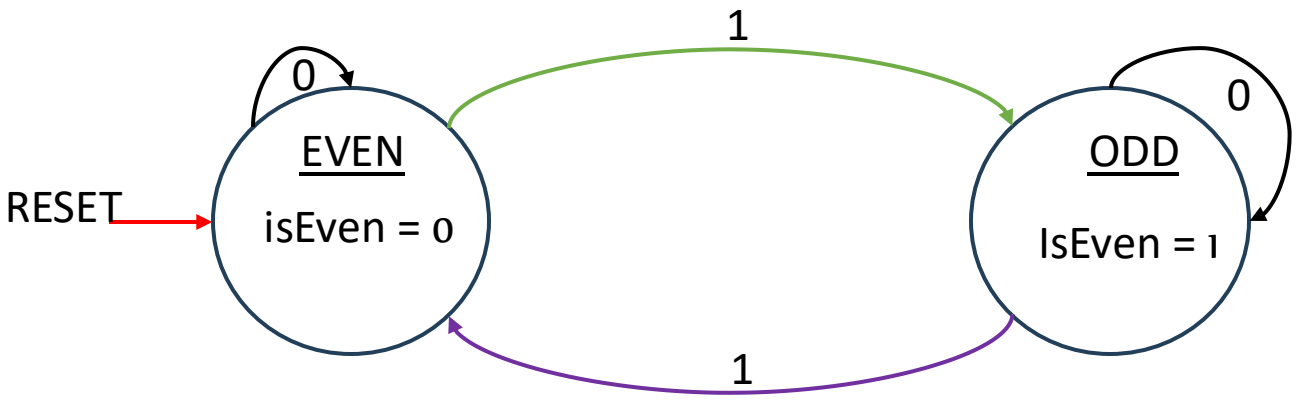
FSM Example 1: Even/Odd FSM

```
module EvenOddFSM(input clk, reset, in
                  output reg isEven);
reg state, nextstate;
parameter EVEN == 0;
Parameter ODD = 1;
initial begin
          state = EVEN;
end
always_ff @(posedge clk, posedge reset) begin
    if (reset) state <= EVEN; // non-blocking <=
   else       state <= nextstate;
end
   // next state logic
   always_comb
    begin
     case (state)
        EVEN: begin
              isEven = 0;
              if(in) nextstate = ODD;
              else nextstate = EVEN;
           end
           ODD: begin
              isEven = 1;
              if(in) nextstate = EVEN;
              else nextstate = ODD;
           end
           default: begin
              isEven = 0'bx;
              nextstate = 0'bx;
           end
    endcase
end
endmodule
```



FSM Example 1: Even/Odd FSM

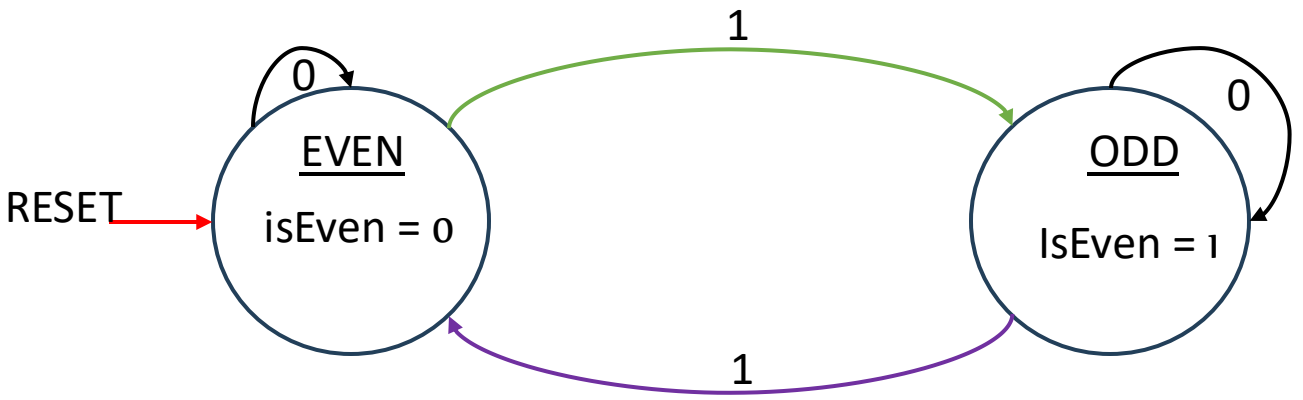
```
module EvenOddFSM(input clk, reset, in
                  output reg isEven);
reg state, nextstate;
parameter EVEN == 0;
Parameter ODD = 1;
initial begin
          state = EVEN;
end
always_ff @(posedge clk, posedge reset) begin
    if (reset) state <= EVEN; // non-blocking <=
   else       state <= nextstate;
end
   // next state logic
   always_comb
    begin
     case (state)
        EVEN: begin
              isEven = 0;
              if(in) nextstate = ODD;
              else nextstate = EVEN;
           end
           ODD: begin
              isEven = 1;
              if(in) nextstate = EVEN;
              else nextstate = ODD;
           end
           default: begin
              isEven = 0'bx;
              nextstate = 0'bx;
           end
    endcase
end
endmodule
```




```
always_ff @(posedge clk) begin
       if (reset) state <= EVEN;
   else       state <= nextstate;
end
```

FSM Example 1: Even/Odd FSM

```
module EvenOddFSM(input clk, reset, in
                  output reg isEven);
reg state, nextstate;
parameter EVEN == 0;
Parameter ODD = 1;
initial begin
          state = EVEN;
end
always_ff @(posedge clk, posedge reset) begin
    if (reset) state <= EVEN; // non-blocking <=
   else       state <= nextstate;
end
   // next state logic
   always_comb
    begin
     case (state)
        EVEN: begin
              isEven = 0;
              if(in) nextstate = ODD;
              else nextstate = EVEN;
           end
           ODD: begin
              isEven = 1;
              if(in) nextstate = EVEN;
              else nextstate = ODD;
           end
           default: begin
              isEven = 0'bx;
              nextstate = 0'bx;
           end
    endcase
end
endmodule
```

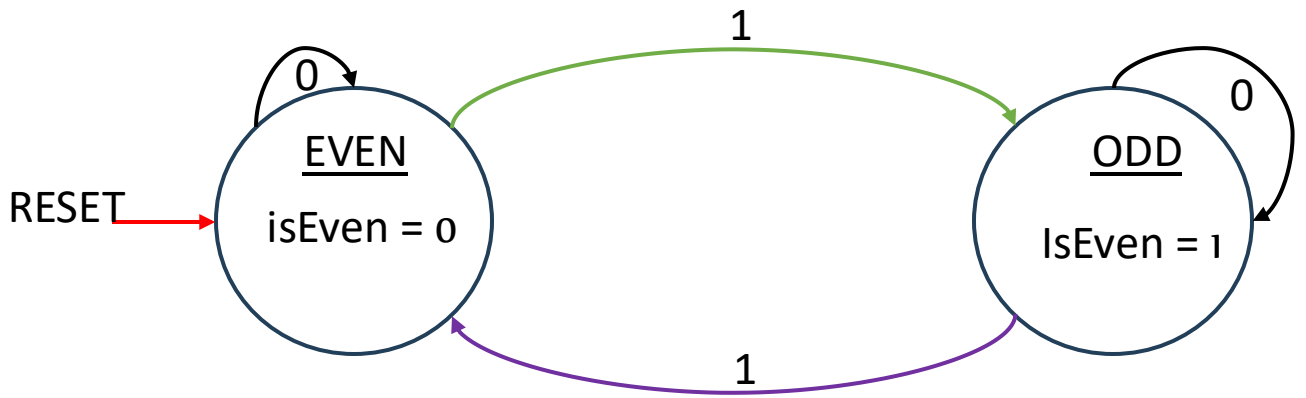


What is the difference?

```
always_ff @(posedge clk) begin
      if (reset) state <= EVEN;
   else       state <= nextstate;
end
```

FSM Example 1: Even/Odd FSM

```
module EvenOddFSM(input clk, reset, in
                  output reg isEven);
reg state, nextstate;
parameter EVEN == 0;
Parameter ODD = 1;
initial begin
          state = EVEN;
end
always_ff @(posedge clk, posedge reset) begin
    if (reset) state <= EVEN; // non-blocking <=
   else       state <= nextstate;
end
   // next state logic
   always_comb
    begin
     case (state)
        EVEN: begin
                isEven = 0;
                if(in) nextstate = ODD;
                else nextstate = EVEN;
            end
            ODD: begin
                isEven = 1;
                if(in) nextstate = EVEN;
                else nextstate = ODD;
            end
            default: begin
                isEven = 0'bx;
                nextstate = 0'bx;
            end
    endcase
end
endmodule
```



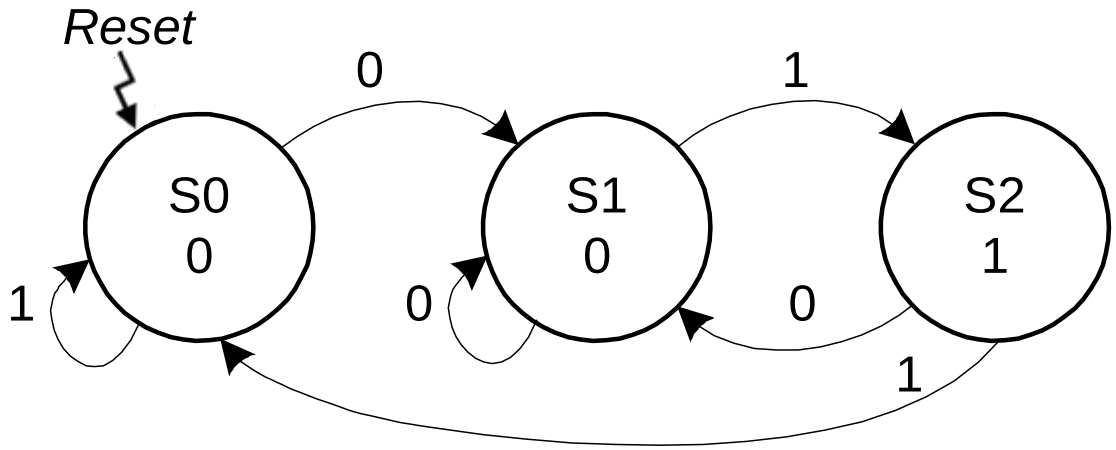
What is the difference?

```
always_ff @(posedge clk) begin
      if (reset) state <= EVEN;
   else       state <= nextstate;
end
```

Note that a default case is used in the combinational block, to be sure that output **`nextstate`** *receives a value in all cases*.
- The initial blocks sets the FSM to start at EVEN state

FSM Example 2: Sequence Detector

Moore FSM

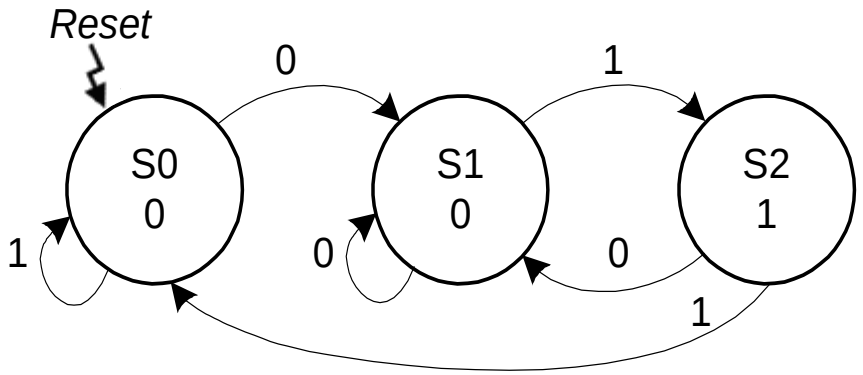


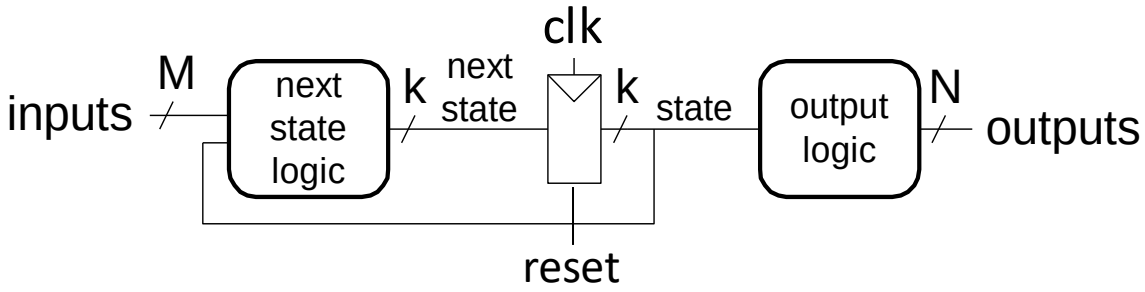
Which sequence will be detected?

Sequence Detector FSM: Moore

```
module seqDetectMoore(input clk, reset, a,
                      output smile);

   parameter S0 = 00;
   parameter S1 = 01;
   parameter S2 = 10;
   reg state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;
   // next state logic
   always_comb
      case (state)
        S0:      if (a) nextstate = S0;
                 else   nextstate = S1;
        S1:      if (a) nextstate = S2;
                 else   nextstate = S1;
        S2:      if (a) nextstate = S0;
                 else   nextstate = S1;
        default:        nextstate = S0;
      endcase
   // output logic
   assign smile = (state == S2);
endmodule
```

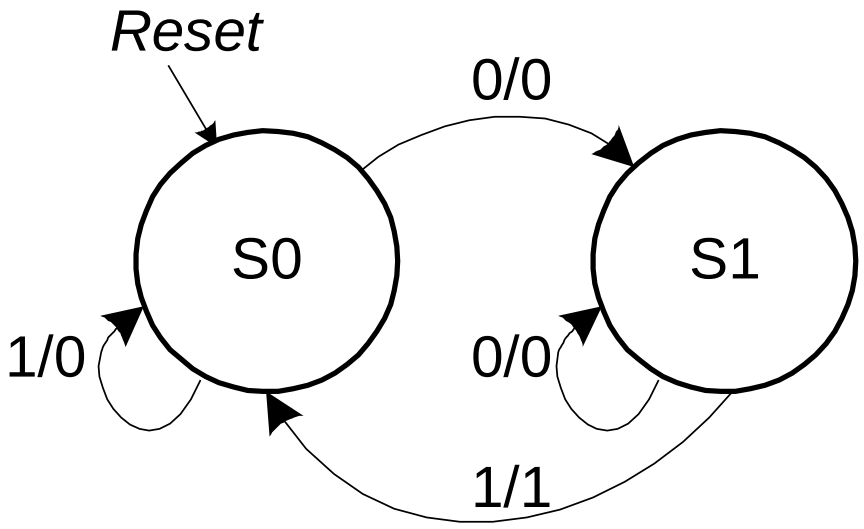
Moore FSM





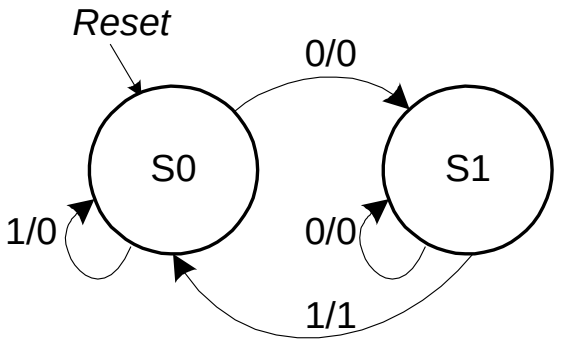
FSM Example 3: Sequence Detector

Mealy FSM

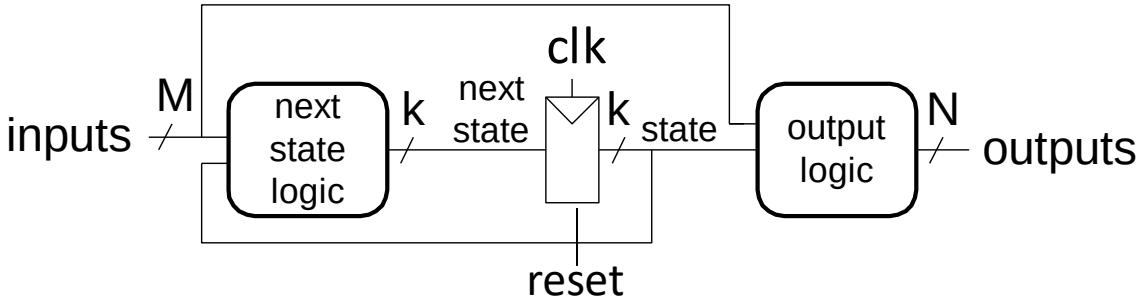


Sequence Detector FSM: Mealy

Mealy FSM



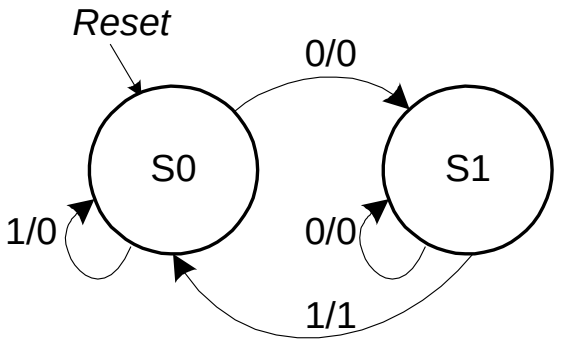
Mealy FSM



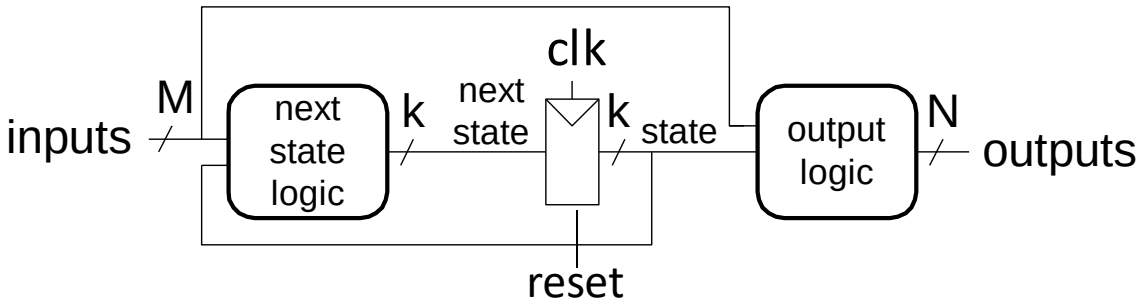
Sequence Detector FSM: Mealy

```
module seqDetectMealy(input clk, reset, a,
                      output smile);

parameter S0 = 0;
parameter S1 = 1;
reg state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;
   // next state and output logic
   always_comb begin
      // make sure smile receives value in all cases.
      smile = 1'b0;
      case (state)
         S0:    if (a)   nextstate = S0;
                else     nextstate = S1;
         S1:    if (a) begin
                         nextstate = S0;
                         smile = 1'b1;
                        end
                else     nextstate = S1;
         default:        nextstate = S0;
     endcase
   end
endmodule
```

Mealy FSM



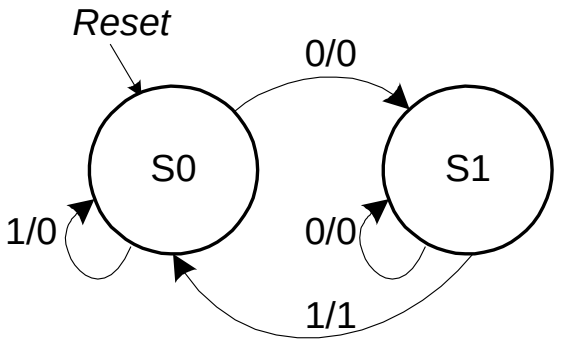
Mealy FSM



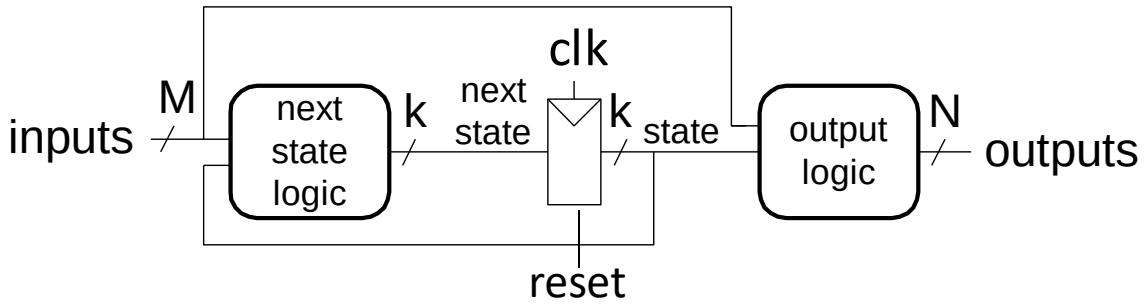
Sequence Detector FSM: Mealy

```
module seqDetectMealy(input clk, reset, a,
                      output smile);

parameter S0 = 0;
parameter S1 = 1;
reg state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;
   // next state and output logic
   always_comb begin
      // make sure smile receives value in all cases.
      smile = 1'b0;
      case (state)
         S0:    if (a)   nextstate = S0;
                else     nextstate = S1;
         S1:    if (a) begin
                         nextstate = S0;
                         smile = 1'b1;
                        end
                else     nextstate = S1;
         default:        nextstate = S0;
      endcase
   end
endmodule
```

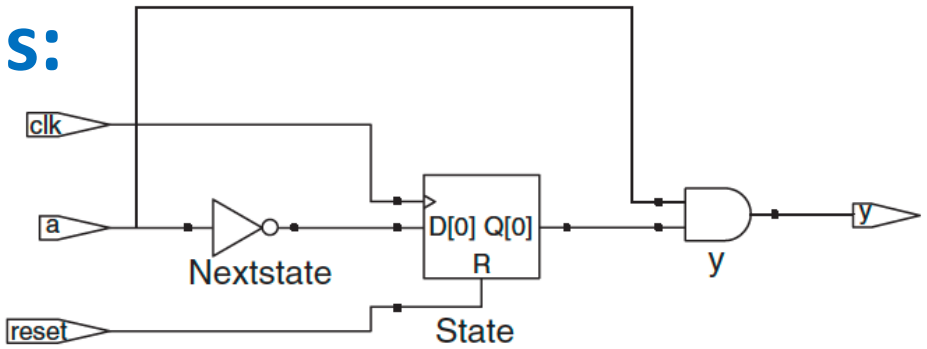
Mealy FSM



Mealy FSM



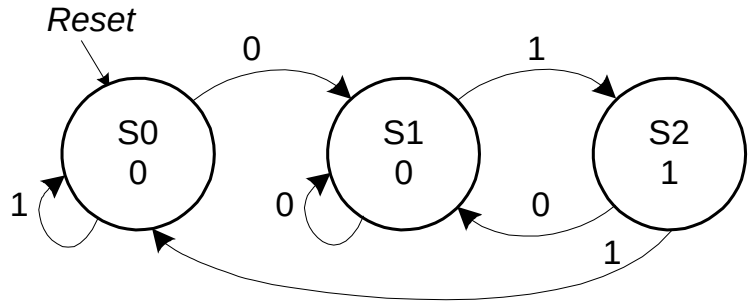
Synthesis:



FSM Testbench

```
`timescale 1ns/1ps

module seqDetectMoore_tb();

   reg clk, reset, a;
   wire smile;

   seqDetectMoore dut (clk, reset, a, smile);

   initial
      clk = 0;
   always
      #10 clk = ~clk;

   initial begin
            reset = 1; a = 0;
      #20; reset = 0;
      #20;             a = 1;
   end

endmodule
```

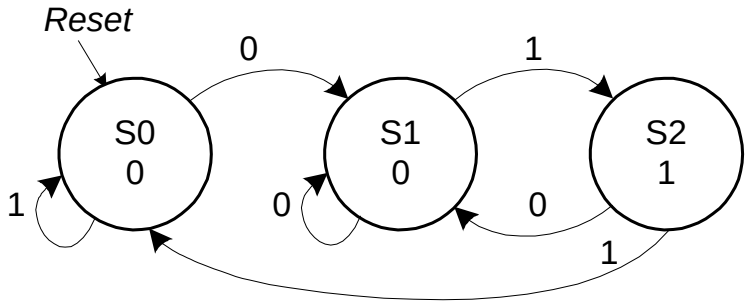
Moore FSM

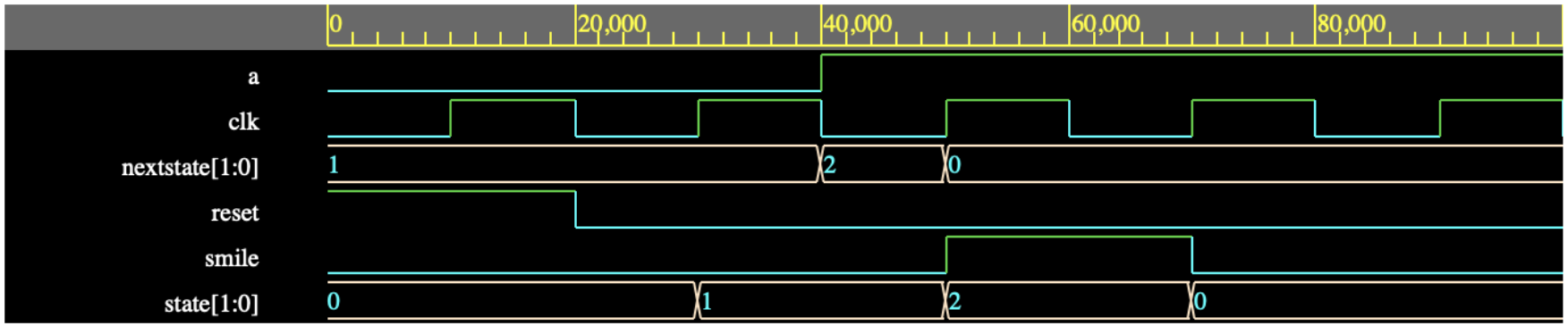


FSM Testbench

```
`timescale 1ns/1ps

module seqDetectMoore_tb();

   reg clk, reset, a;
   wire smile;

   seqDetectMoore dut (clk, reset, a, smile);

   initial
      clk = 0;
   always
      #10 clk = ~clk;

   initial begin
            reset = 1; a = 0;
      #20; reset = 0;
      #20;             a = 1;
   end

endmodule
```

Moore FSM

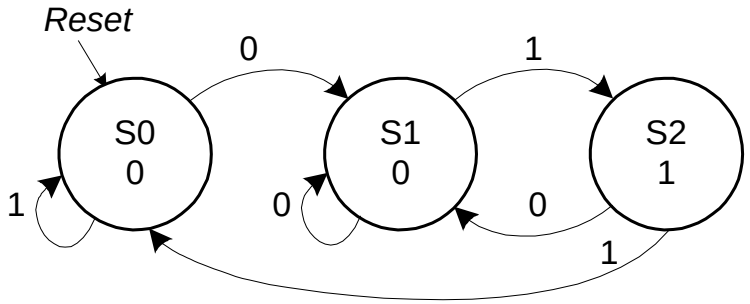


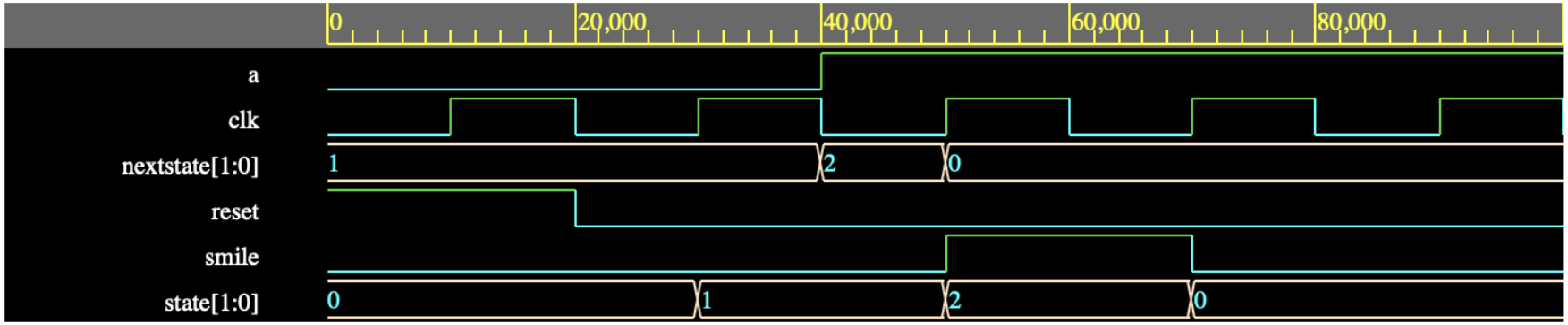


FSM Testbench

```
`timescale 1ns/1ps

module seqDetectMoore_tb();

   reg clk, reset, a;
   wire smile;

   seqDetectMoore dut (clk, reset, a, smile);

   initial
      clk = 0;
   always
      #10 clk = ~clk;

   initial begin
            reset = 1; a = 0;
      #20; reset = 0;
      #20;             a = 1;
   end

endmodule
```

Moore FSM





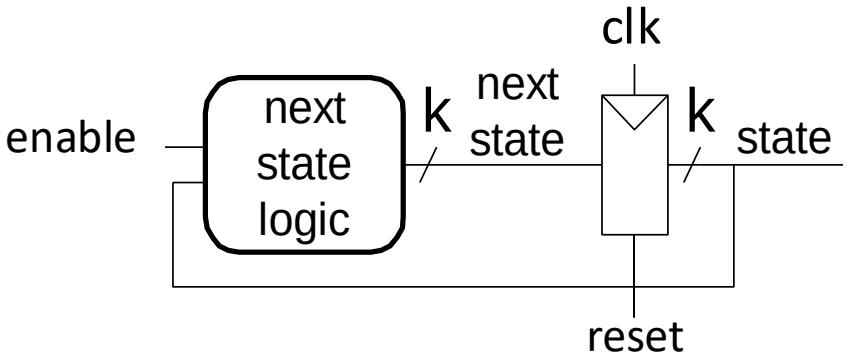
Important: to avoid setup/hold time violations, do not change input signals on the rising clock edge. Instead, do it on the negative clock edge.

Counter

A counter can be considered as a FSM that adds 1 to its state in its next state logic.
Usually there is no output logic.

Counter

A counter can be considered as a FSM that adds 1 to its state in its next state logic. Usually there is no output logic.

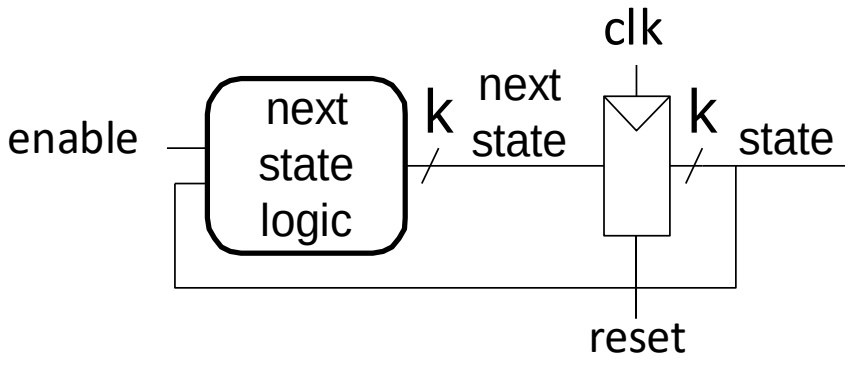


state ≡ count

Counter

A counter can be considered as a FSM that adds 1 to its state in its next state logic. Usually there is no output logic.

```
module counter(input clk, reset, enable,
               output [7:0] count);

   reg [7:0] next_count;

   // register
   always_ff@(posedge clk)
   begin
      if (reset) count <= 0;
      else count <= next_count;
   end

   // next state logic
   always_comb
   begin
      if (enable) next_count <= count + 1;
      else next_count <= count;
   end
endmodule
```



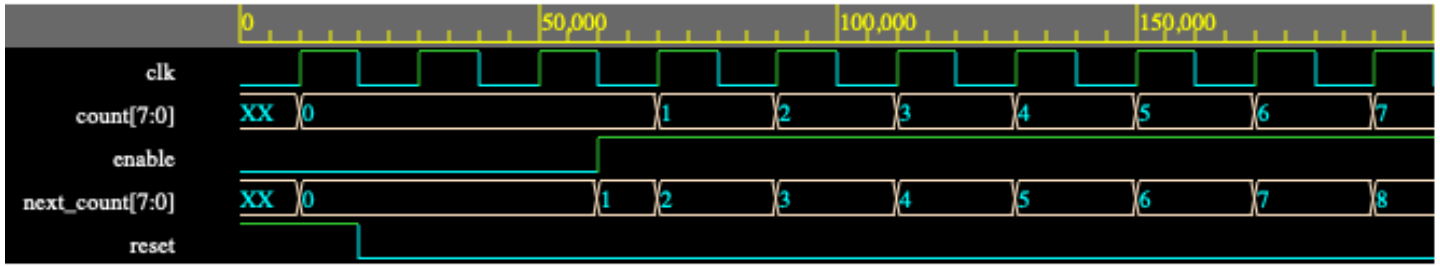
state ≡ count

Counter simulation

```
module counter_tb();

    reg clk;
    reg reset;
    reg enable;
    wire [7:0] count;

    counter dut (clk, reset, enable, count);

    always
        #10 clk = ~clk;
    initial
        clk = 0;

    initial begin
            reset = 1; enable = 0;
        #20; reset = 0;
        #40;           enable = 1;
    end

endmodule
```

Counter simulation

```
module counter_tb();

    reg clk;
    reg reset;
    reg enable;
    wire [7:0] count;

    counter dut (clk, reset, enable, count);

    always
        #10 clk = ~clk;
    initial
        clk = 0;

    initial begin
            reset = 1; enable = 0;
        #20; reset = 0;
        #40;           enable = 1;
    end

endmodule
```



Adder design in verilog

Half Adder

- Two 1-bit inputs
- Two 1-bit outputs

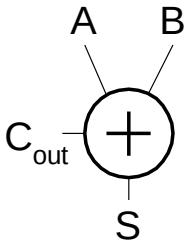
Full Adder

- Three 1-bit inputs
- Two 1-bit outputs

Adder design in verilog

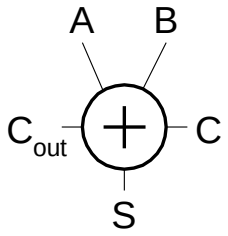
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



Full Adder

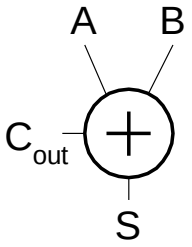
- Three 1-bit inputs
- Two 1-bit outputs



Adder design in verilog

Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



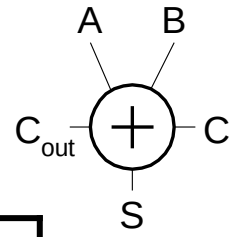
$S =$

$C_{out} =$

A	B	C_o	S
0	0		
0	1		
1	0		
1	1		

Full Adder

- Three 1-bit inputs
- Two 1-bit outputs



$S =$

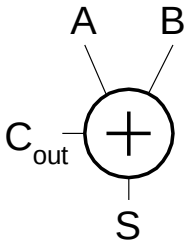
$C_{out} =$

A	B	C	C_o	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Adder design in verilog

Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



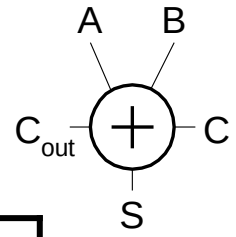
$$S = A \oplus B$$

$$C_{out} = AgB$$

A	B	C_o	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full Adder

- Three 1-bit inputs
- Two 1-bit outputs



$$S = A \oplus B \oplus C$$

$$C_{out} = MAJ(A, B, C)$$

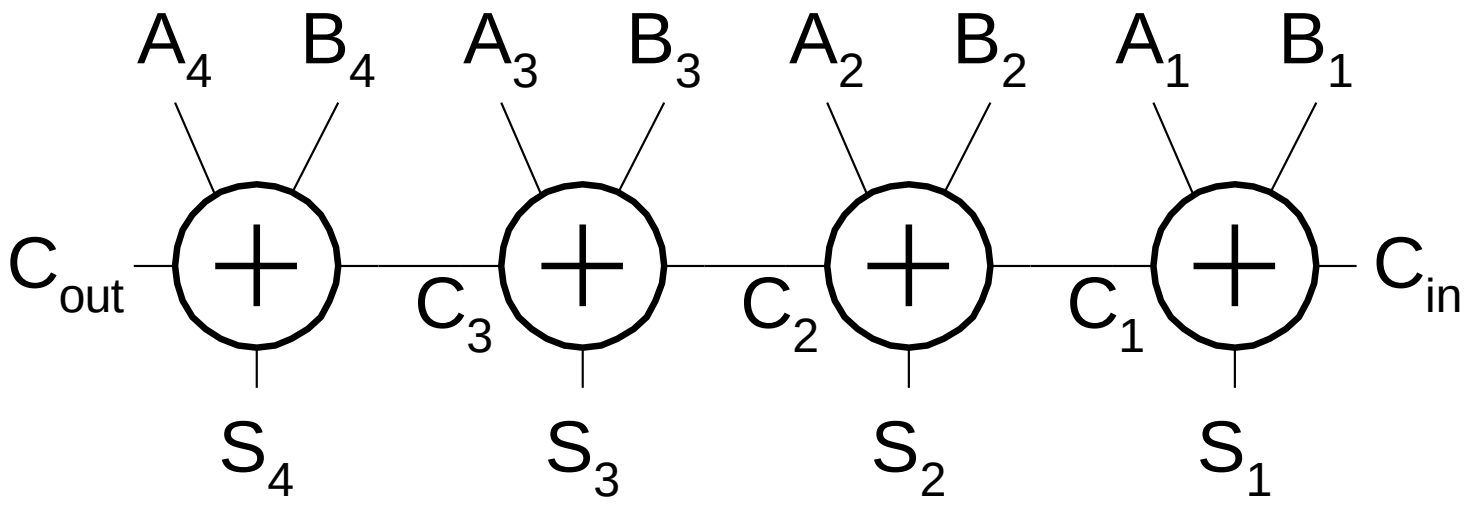
A	B	C	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Simple multi-bit adder

- Simplest design: cascade full adders
  - Critical path goes from C_{in} to C_{out}
  - Design full adder to have fast carry delay

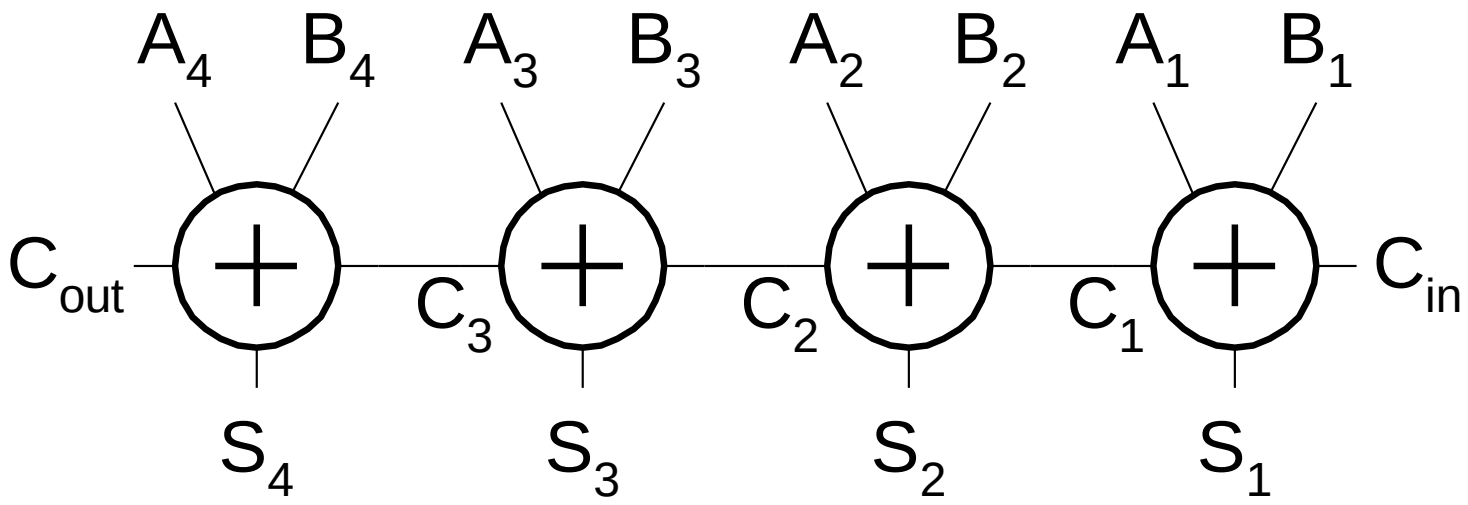
Simple multi-bit adder

- Simplest design: cascade full adders
  - Critical path goes from C_{in} to C_{out}
  - Design full adder to have fast carry delay



Simple multi-bit adder

- Simplest design: cascade full adders
  - Critical path goes from C_{in} to C_{out}
  - Design full adder to have fast carry delay

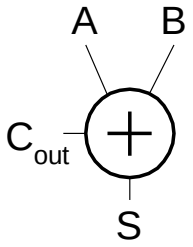


- Commonly known as ripple-carry adder
  - Named from its carry-chain structure

Adder circuit implementation

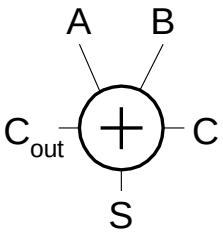
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



Full Adder

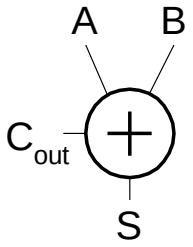
- Three 1-bit inputs
- Two 1-bit outputs



Adder circuit implementation

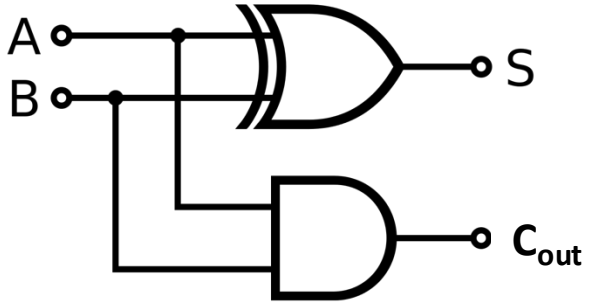
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



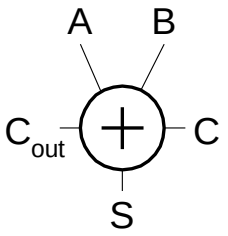
$$S = A \oplus B$$

$$C_{out} = AgB$$



Full Adder

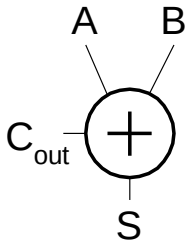
- Three 1-bit inputs
- Two 1-bit outputs



Adder circuit implementation

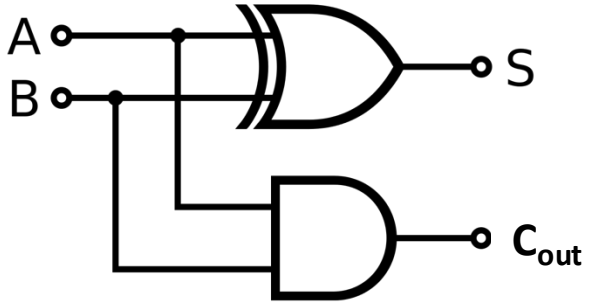
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



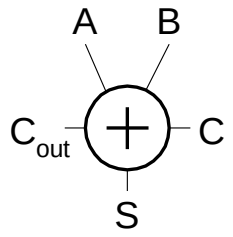
$$S = A \oplus B$$

$$C_{out} = AgB$$



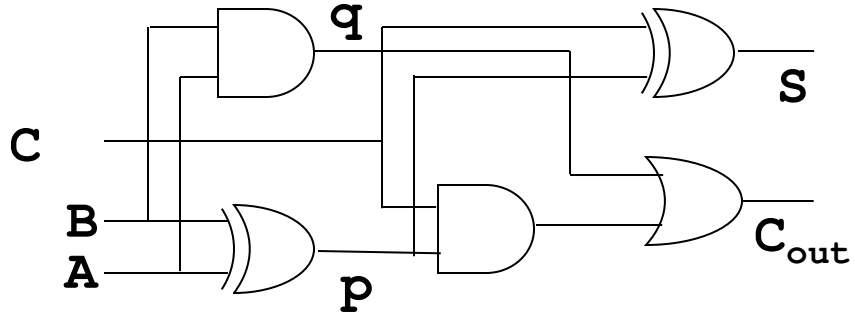
Full Adder

- Three 1-bit inputs
- Two 1-bit outputs



$$S = A \oplus B \oplus C$$

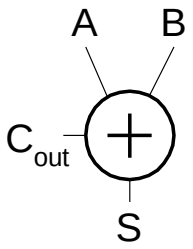
$C_{out} = A.B + B(A \oplus B)$



Adder circuit implementation

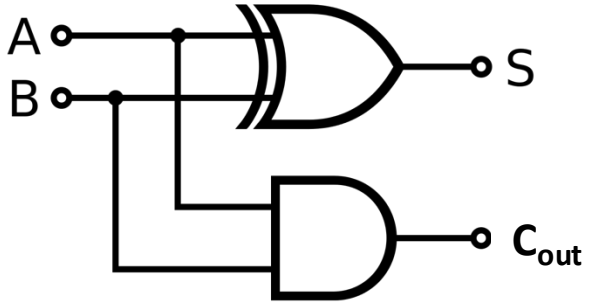
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



$$S = A \oplus B$$

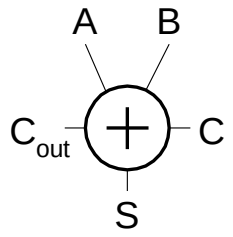
$$C_{out} = AgB$$




```
module HA( input A, B,
output S,Cout );

assign S = a ^ b;
assign Cout = a & b;

endmodule
```

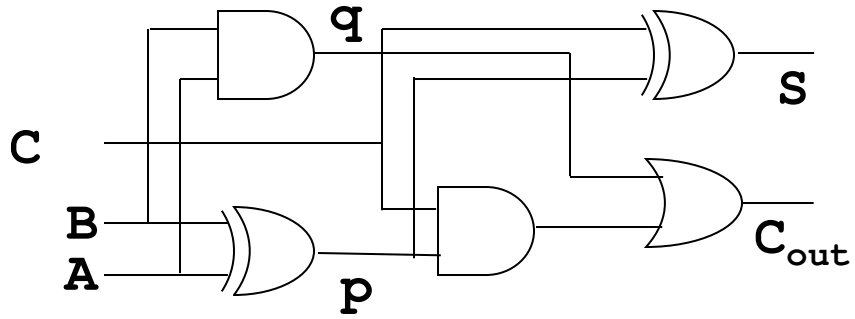
Full Adder

- Three 1-bit inputs
- Two 1-bit outputs



$$S = A \oplus B \oplus C$$

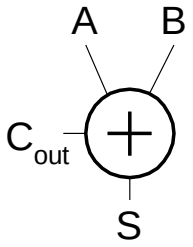
C_{out} = A.B + B(A⊕B)



Adder circuit implementation

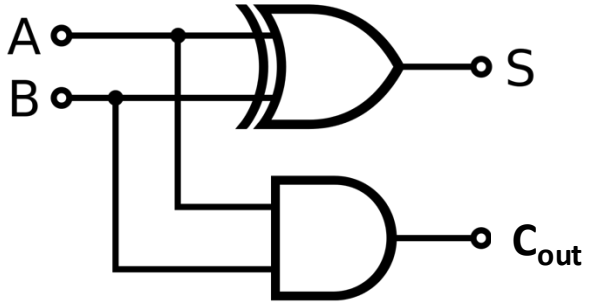
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



$$S = A \oplus B$$

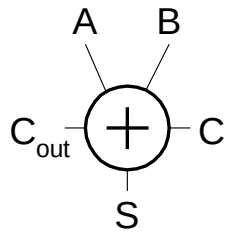
$$C_{out} = AgB$$




```
module HA( input A, B,
output S,Cout );

assign S = a ^ b;
assign Cout = a & b;

endmodule
```

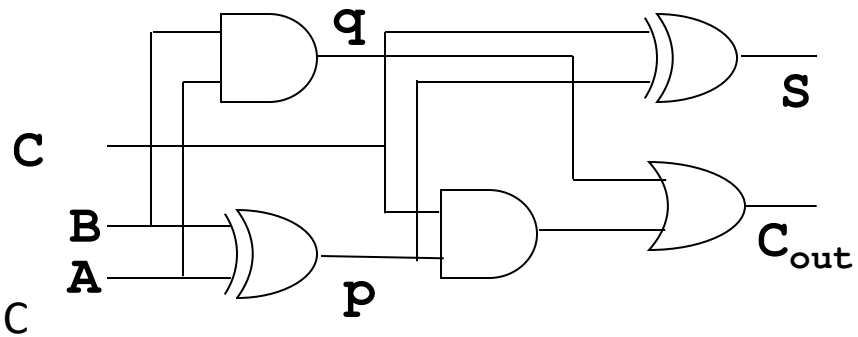
Full Adder

- Three 1-bit inputs
- Two 1-bit outputs



$$S = A \oplus B \oplus C$$

$C_{out} = A.B + B(A \oplus B)$




```
module FA( input A, B, C
output S, Cout );
Wire p,q;

assign p = B ^ A;
Assign q= A & B;
assign S = p ^ C;
assign Cout = q | (p & C);
endmodule
```

Full adder impementation in verilog

Behavioural verilog implemntation

```
module FA( input A, B, C
output S, Cout );
Wire p,q;

assign p = B ^ A;
Assign q= A & B;
assign S = p ^ C;
assign Cout = q | (p & C);
endmodule
```

Structural verilog implementation

Full adder impementation in verilog

Behavioural verilog implemntation

```
module FA( input A, B, C
output S, Cout );
Wire p,q;

assign p = B ^ A;
Assign q= A & B;
assign S = p ^ C;
assign Cout = q | (p & C);
endmodule
```

Equivalent

```
module FA_Behavioral_( input A, B, C, output S, Cout );

reg[1:0] temp;

always @(*)
begin
temp = {1'b0,A} + {1'b0,B}+{1'b0,C};
end
assign S = temp[0];
assign Cout = temp[1];
endmodule
```

Structural verilog implementation

Full adder impementation in verilog

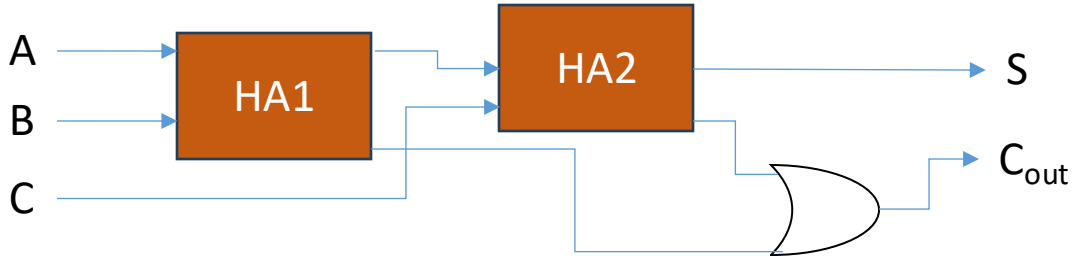
Behavioural verilog implemntation

```
module FA( input A, B, C
output S, Cout );
Wire p,q;

assign p = B ^ A;
Assign q= A & B;
assign S = p ^ C;
assign Cout = q | (p & C);
endmodule
```

Equivalent

```
module FA_Behavioral_( input A, B, C, output S, Cout );

reg[1:0] temp;

always @(*)
begin
temp = {1'b0,A} + {1'b0,B}+{1'b0,C};
end
assign S = temp[0];
assign Cout = temp[1];
endmodule
```

Structural verilog implementation



Full adder impementation in verilog

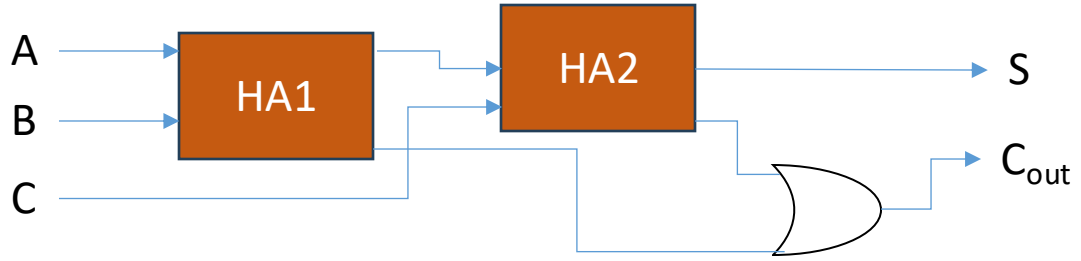
Behavioural verilog implemntation

```
module FA( input A, B, C
output S, Cout );
Wire p,q;

assign p = B ^ A;
Assign q= A & B;
assign S = p ^ C;
assign Cout = q | (p & C);
endmodule
```

Equivalent

```
module FA_Behavioral_( input A, B, C, output S, Cout );

reg[1:0] temp;

always @(*)
begin
temp = {1'b0,A} + {1'b0,B}+{1'b0,C};
end
assign S = temp[0];
assign Cout = temp[1];
endmodule
```

Structural verilog implementation




```
module HA( input A, B,
output S,Cout );

assign S = a ^ b;
assign Cout = a & b;

endmodule
```

Full adder impementation in verilog

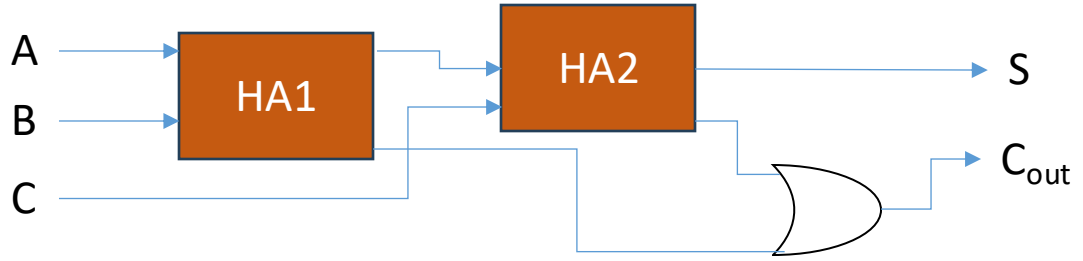
Behavioural verilog implemntation

```
module FA( input A, B, C
output S, Cout );
Wire p,q;

assign p = B ^ A;
Assign q= A & B;
assign S = p ^ C;
assign Cout = q | (p & C);
endmodule
```

Equivalent

```
module FA_Behavioral_( input A, B, C, output S, Cout );

reg[1:0] temp;

always @(*)
begin
temp = {1'b0,A} + {1'b0,B}+{1'b0,C};
end
assign S = temp[0];
assign Cout = temp[1];
endmodule
```

Structural verilog implementation




```
module HA( input A, B,
output S,Cout );

assign S = a ^ b;
assign Cout = a & b;

endmodule
```

```
module FA( input A, B, C,
output S, Cout );
wire c0,c1,so;
HA ha0(A, b, So, c0);
HA ha1(C, so, S,c1);
assign carry = c0 | c1 ;
endmodule
```

Verilog for Ripple Carry Adder (RCA)

```
module nbit_RCA( input [7:0] a, [7:0] b, cin,
                 output [7:0] sum, cout);
wire [7:1] carry; /* transfers the carry between bits */

   FA a0(a[0],b[0],cin,sum[0],carry[1]);
   FA a1(a[1],b[1],carry[1],sum[1],carry[2]);
   FA a2(a[2],b[2], carry[2],sum[2],carry[3]);
   FA a3(a[3],b[3],carry[3],sum[3],carry[4]);
   FA a4(a[4],b[4], carry[4],sum[4],carry[5]);
   FA a5(a[5],b[5],carry[5],sum[5],carry[6]);
   FA a6(a[6],b[6], carry[6],sum[6],carry[7]);
   FA a7(a[7],b[7],carry[7],sum[7], cout);
endmodule
```

Verilog for Ripple Carry Adder (RCA)

```
module nbit_RCA( input [7:0] a, [7:0] b, cin,
                 output [7:0] sum, cout);
wire [7:1] carry; /* transfers the carry between bits */

   FA a0(a[0],b[0],cin,sum[0],carry[1]);
   FA a1(a[1],b[1],carry[1],sum[1],carry[2]);
   FA a2(a[2],b[2], carry[2],sum[2],carry[3]);
   FA a3(a[3],b[3],carry[3],sum[3],carry[4]);
   FA a4(a[4],b[4], carry[4],sum[4],carry[5]);
   FA a5(a[5],b[5],carry[5],sum[5],carry[6]);
   FA a6(a[6],b[6], carry[6],sum[6],carry[7]);
   FA a7(a[7],b[7],carry[7],sum[7], cout);
endmodule
```

Carry propagation time is a problem!!

Verilog for Ripple Carry Adder (RCA)

```
module nbit_RCA( input [7:0] a, [7:0] b, cin,
                 output [7:0] sum, cout);
wire [7:1] carry; /* transfers the carry between bits */

   FA a0(a[0],b[0],cin,sum[0],carry[1]);
   FA a1(a[1],b[1],carry[1],sum[1],carry[2]);
   FA a2(a[2],b[2], carry[2],sum[2],carry[3]);
   FA a3(a[3],b[3],carry[3],sum[3],carry[4]);
   FA a4(a[4],b[4], carry[4],sum[4],carry[5]);
   FA a5(a[5],b[5],carry[5],sum[5],carry[6]);
   FA a6(a[6],b[6], carry[6],sum[6],carry[7]);
   FA a7(a[7],b[7],carry[7],sum[7], cout);
endmodule
```

Carry propagation time is a problem!!

Reduce the carry propagation time.

Verilog for Ripple Carry Adder (RCA)

```
module nbit_RCA( input [7:0] a, [7:0] b, cin,
                 output [7:0] sum, cout);
wire [7:1] carry; /* transfers the carry between bits */

   FA a0(a[0],b[0],cin,sum[0],carry[1]);
   FA a1(a[1],b[1],carry[1],sum[1],carry[2]);
   FA a2(a[2],b[2], carry[2],sum[2],carry[3]);
   FA a3(a[3],b[3],carry[3],sum[3],carry[4]);
   FA a4(a[4],b[4], carry[4],sum[4],carry[5]);
   FA a5(a[5],b[5],carry[5],sum[5],carry[6]);
   FA a6(a[6],b[6], carry[6],sum[6],carry[7]);
   FA a7(a[7],b[7],carry[7],sum[7], cout);
endmodule
```

Carry propagation time is a problem!!

Reduce the carry propagation time.

How to do it?

Complex adders with circuitry to detect carry generation/completion

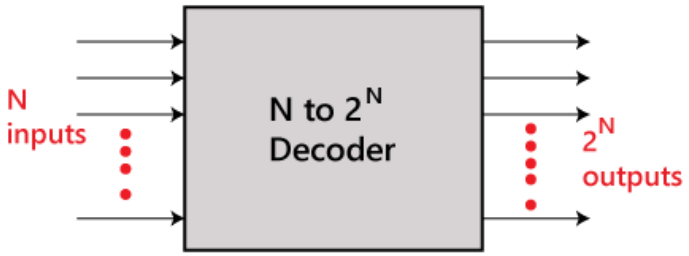
Not part of this lecture

Decoders

- Decoder is a combinational circuit that change the binary information into 2^N output lines
- Performs the reverse operation of encoder

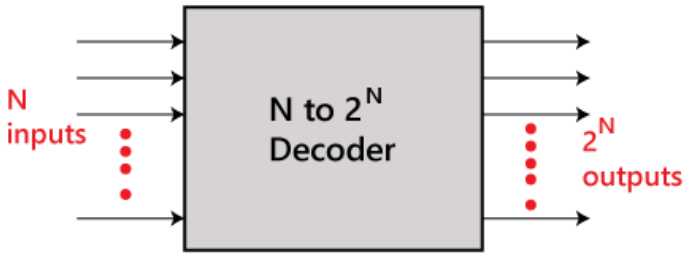
Decoders

- Decoder is a combinational circuit that change the binary information into 2^N output lines
- Performs the reverse operation of encoder



Decoders

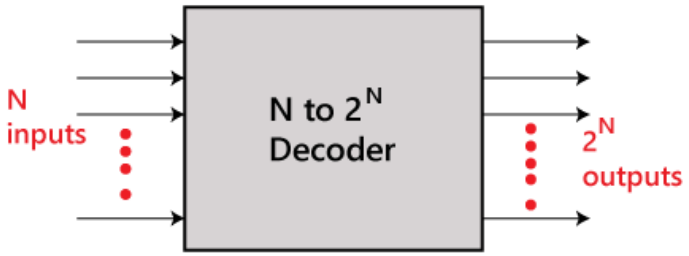
- Decoder is a combinational circuit that change the binary information into 2^N output lines
- Performs the reverse operation of encoder



Example: Three-to-eight decoder

Decoders

- Decoder is a combinational circuit that change the binary information into 2^N output lines
- Performs the reverse operation of encoder



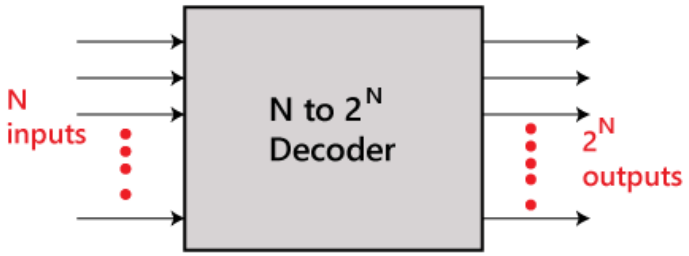
Example: Three-to-eight decoder

Truth table of a three-to-Eight (3:8) decoder

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Decoders

- Decoder is a combinational circuit that change the binary information into 2^N output lines
- Performs the reverse operation of encoder



Example: Three-to-eight decoder

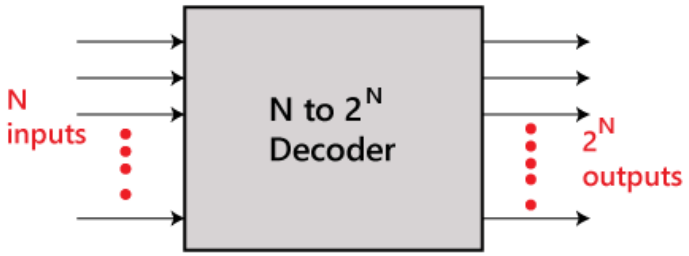
Truth table of a three-to-Eight (3:8) decoder

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1

Decoders

- Decoder is a combinational circuit that change the binary information into 2^N output lines
- Performs the reverse operation of encoder



Example: Three-to-eight decoder

Truth table of a three-to-Eight (3:8) decoder

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1

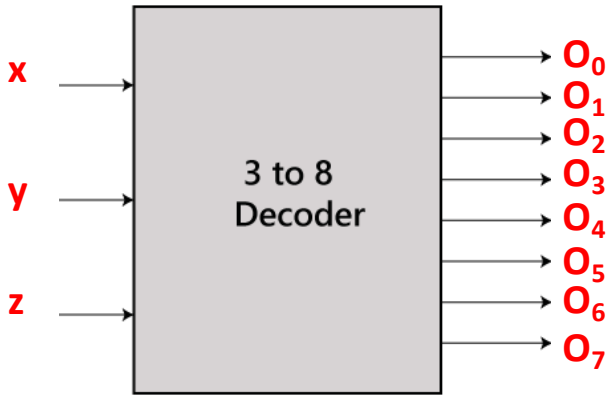
Converts binary information from *n* input lines to 2^n unique output

Or the circuit takes a binary number and converts it to an octal number

Decoder hardware implementation

Three-to-eight decoder

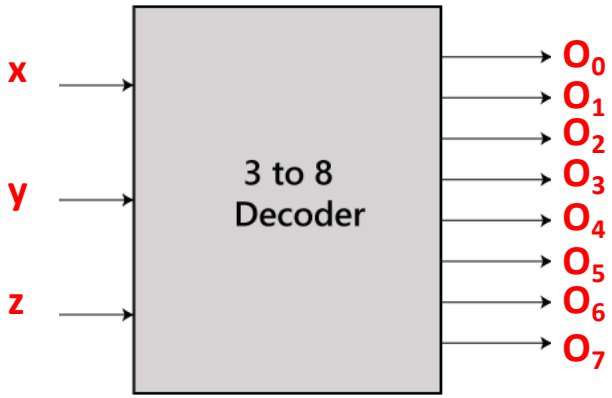
Block diagram



Decoder hardware implementation

Three-to-eight decoder

Block diagram



Logical expression

$O_0 = x'.y'.z'$

$O_1 = x'.y'.z$

$O_2 = x'.y.z'$

$O_3 = x'.y.z$

$O_4 = x.y'.z'$

$O_5 = x.y'.z$

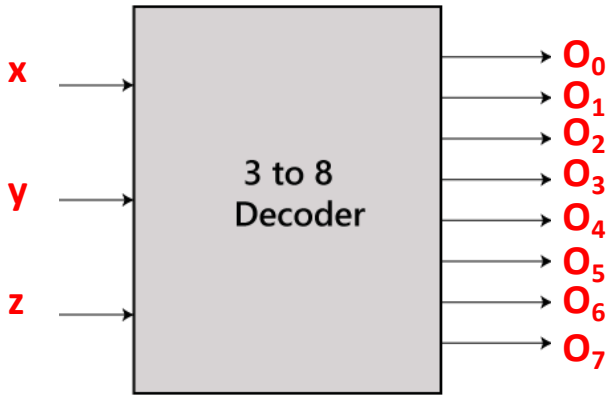
$O_6 = x.y.z'$

$O_7 = x.y.z$

Decoder hardware implementation

Three-to-eight decoder

Block diagram



Logical expression

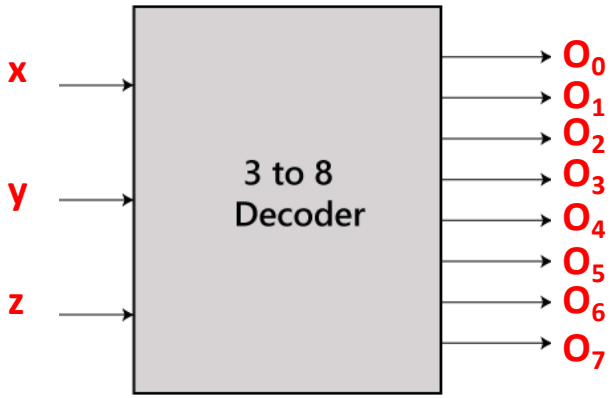
$O_0 = x'.y'.z'$
$O_1 = x'.y'.z$
$O_2 = x'.y.z'$
$O_3 = x'.y.z$
$O_4 = x.y'.z'$
$O_5 = x.y'.z$
$O_6 = x.y.z'$
$O_7 = x.y.z$

Circuit implementation

Decoder hardware implementation

Three-to-eight decoder

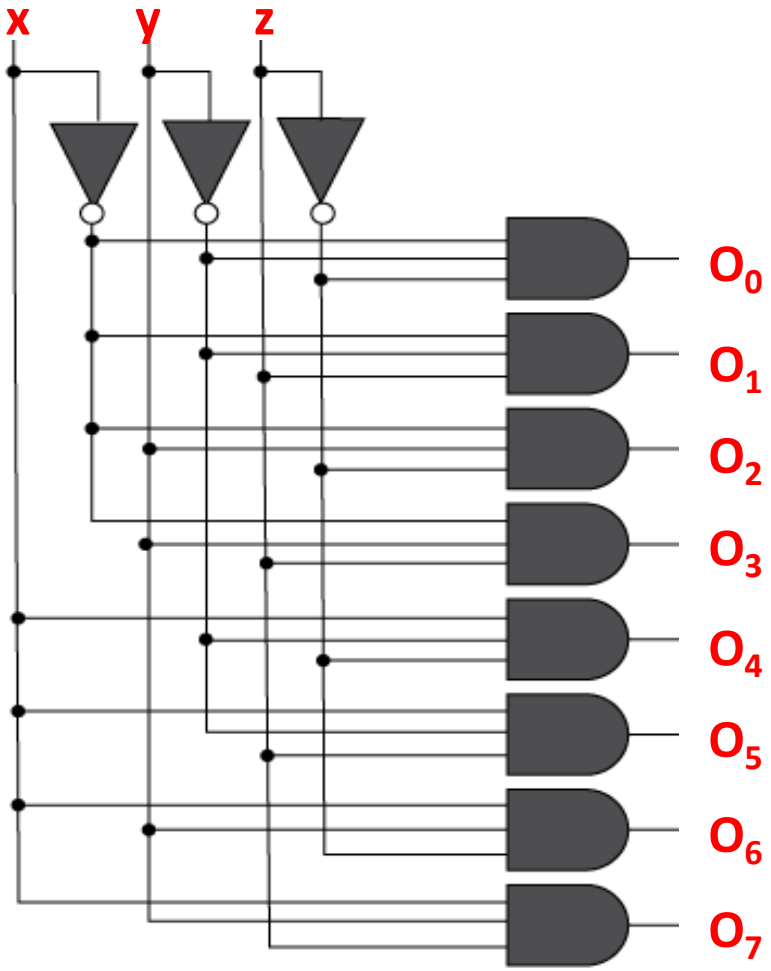
Block diagram



Logical expression

$O_0 = x'.y'.z'$
$O_1 = x'.y'.z$
$O_2 = x'.y.z'$
$O_3 = x'.y.z$
$O_4 = x.y'.z'$
$O_5 = x.y'.z$
$O_6 = x.y.z'$
$O_7 = x.y.z$

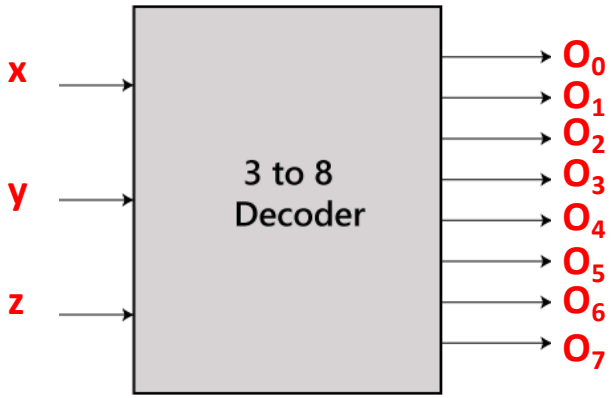
Circuit implementation



Decoder hardware implementation

Three-to-eight decoder

Block diagram



Logical expression

$O_0 = x'.y'.z'$

$O_1 = x'.y'.z$

$O_2 = x'.y.z'$

$O_3 = x'.y.z$

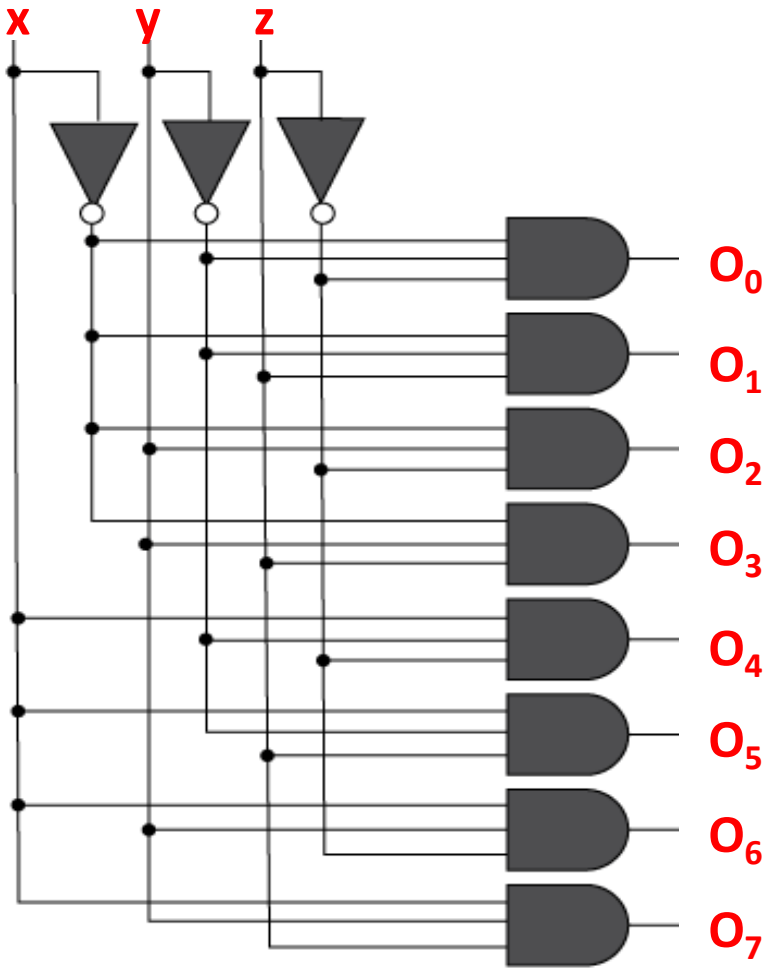
$O_4 = x.y'.z'$

$O_5 = x.y'.z$

$O_6 = x.y.z'$

$O_7 = x.y.z$

Circuit implementation



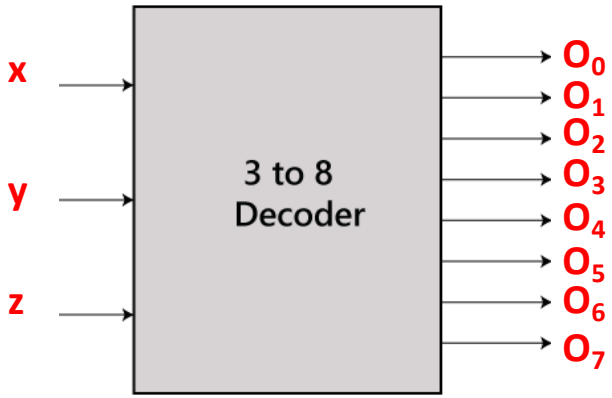
4 to 16 decoder

Can be built using 3 to 8 decoder

Decoder hardware implementation

Three-to-eight decoder

Block diagram



Logical expression

$O_0 = x'.y'.z'$

$O_1 = x'.y'.z$

$O_2 = x'.y.z'$

$O_3 = x'.y.z$

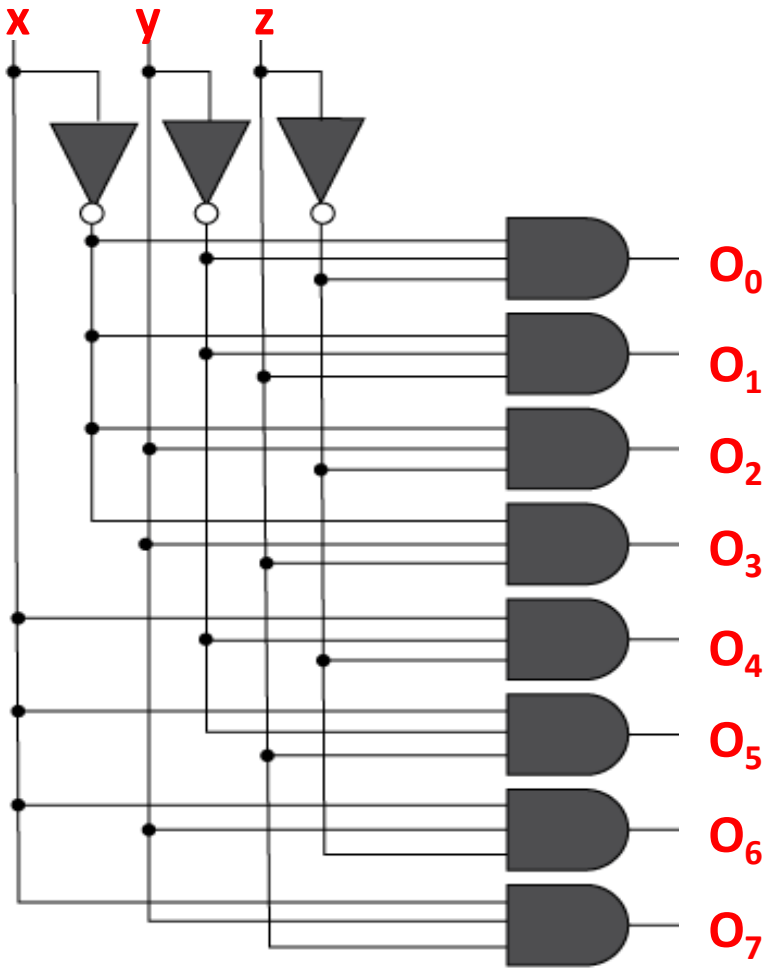
$O_4 = x.y'.z'$

$O_5 = x.y'.z$

$O_6 = x.y.z'$

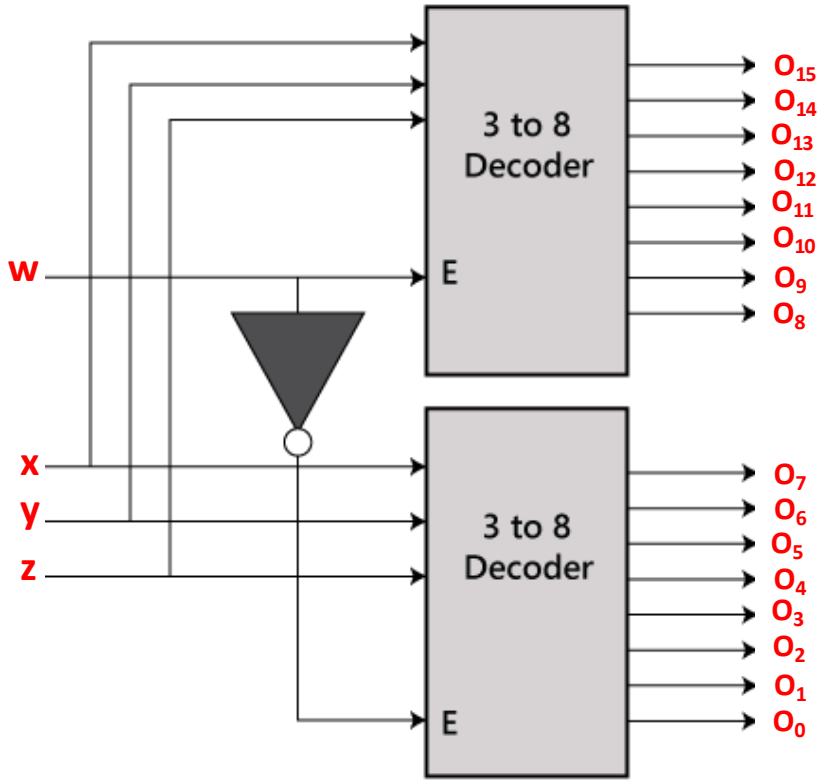
$O_7 = x.y.z$

Circuit implementation



4 to 16 decoder

Can be built using 3 to 8 decoder

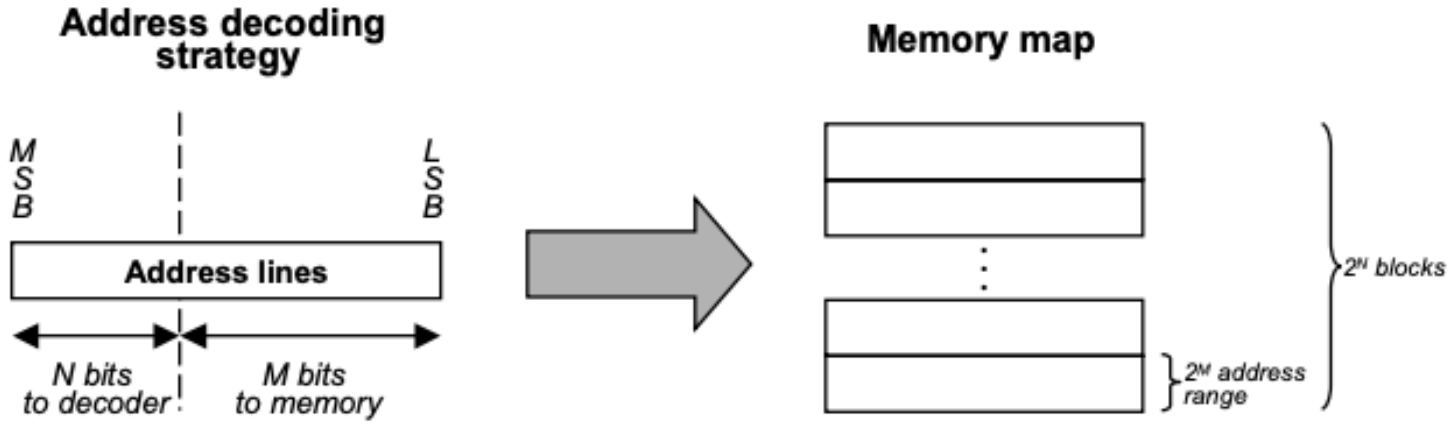


Decoder application: Memory address decoder

- Different portions of memory are used for different purposes: RAM, ROM, I/O devices
- Address decoding is the process of generating chip select (CS*) signals from the address bus for each device in the system

Decoder application: Memory address decoder

- Different portions of memory are used for different purposes: RAM, ROM, I/O devices
- Address decoding is the process of generating chip select (CS*) signals from the address bus for each device in the system
- The address bus lines are split into two sections
  - The N most significant bits are used to generate the CS* signals for the different devices
  - The M least significant signals are passed to the devices as addresses to the different memory cells or internal registers



Decoder application: Memory address decoder

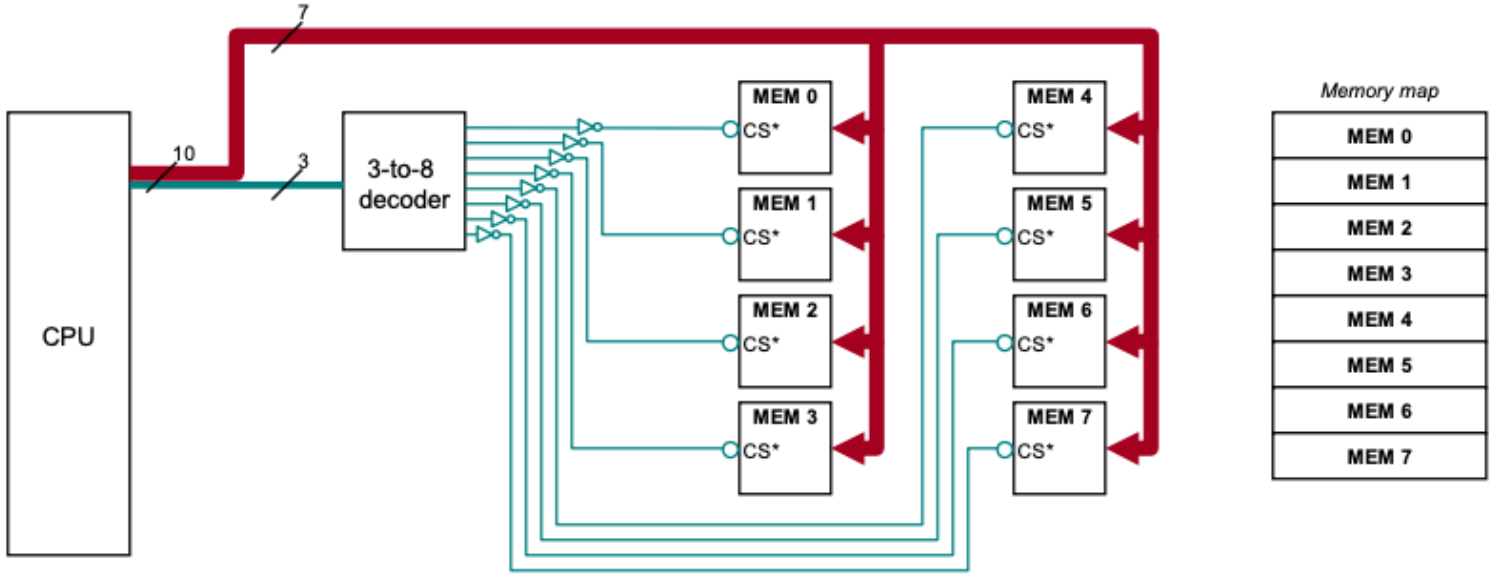
- Let’s assume a very simple microprocessor with 10 address lines (1KB memory)
- Let’s assume we wish to implement all its memory space and we use 128x8 memory

Decoder application: Memory address decoder

- Let’s assume a very simple microprocessor with 10 address lines (1KB memory)
- Let’s assume we wish to implement all its memory space and we use 128x8 memory
- Solution
  - We will need 8 memory chips (8x128=1024)
  - We will need 3 address lines to select each one of the 8 chips
  - Each chip will need 7 address lines to address its internal memory cells

Decoder application: Memory address decoder

- Let’s assume a very simple microprocessor with 10 address lines (1KB memory)
- Let’s assume we wish to implement all its memory space and we use 128x8 memory
- Solution
  - We will need 8 memory chips (8x128=1024)
  - We will need 3 address lines to select each one of the 8 chips
  - Each chip will need 7 address lines to address its internal memory cells



Decoder application: Full adder implementation

3-to-8 decoder as a Full adder

- Truth table

Decoder application: Full adder implementation

3-to-8 decoder as a Full adder

- Truth table

row	x	y	z	C	S
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

Decoder application: Full adder implementation

3-to-8 decoder as a Full adder

- Truth table

row	x	y	z	C	S
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

$S(x, y, z) = \sum(1,2,4,7)$

$C(x, y, z) = \sum(3,5,6,7)$

Decoder application: Full adder implementation

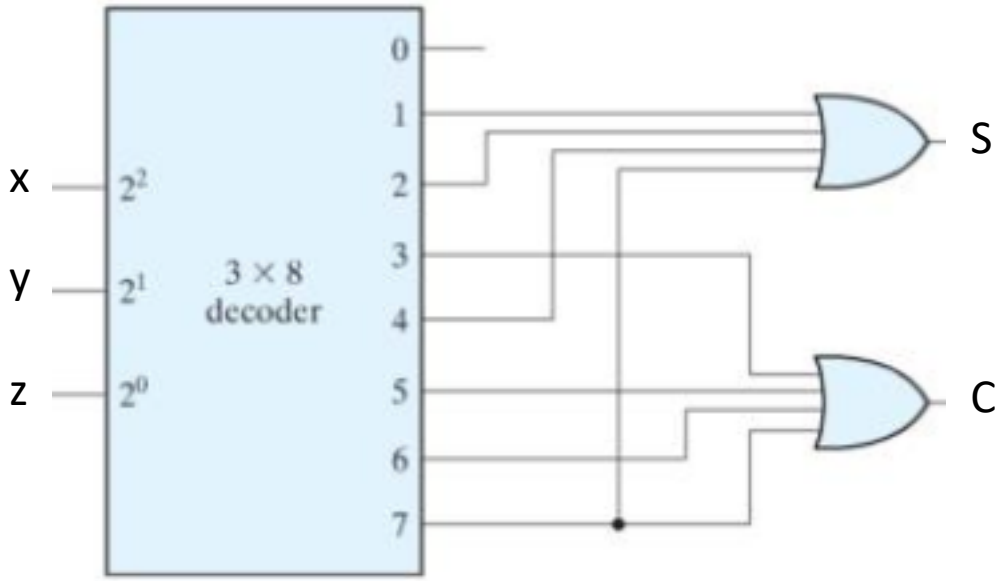
3-to-8 decoder as a Full adder

- Truth table

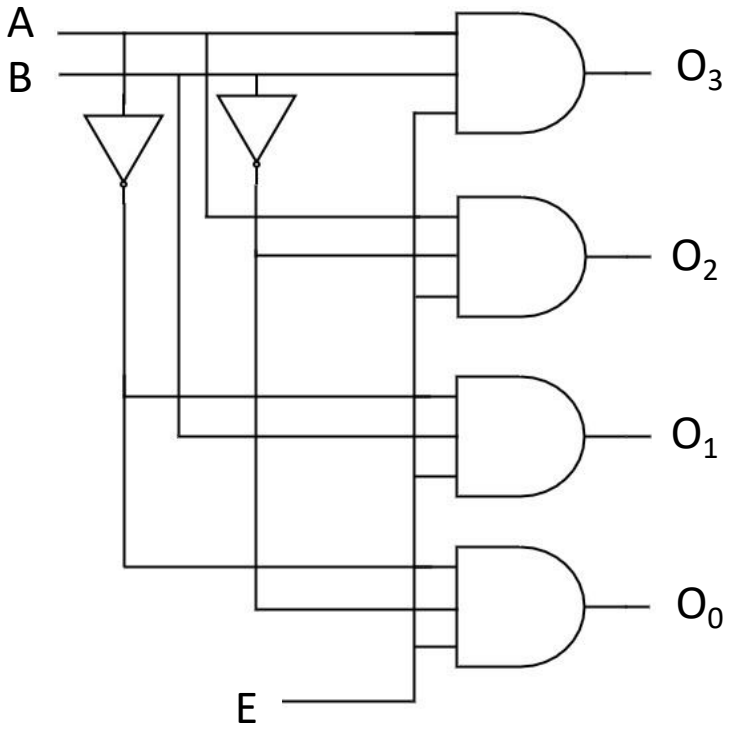
row	x	y	z	C	S
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

$S(x, y, z) = \sum(1,2,4,7)$

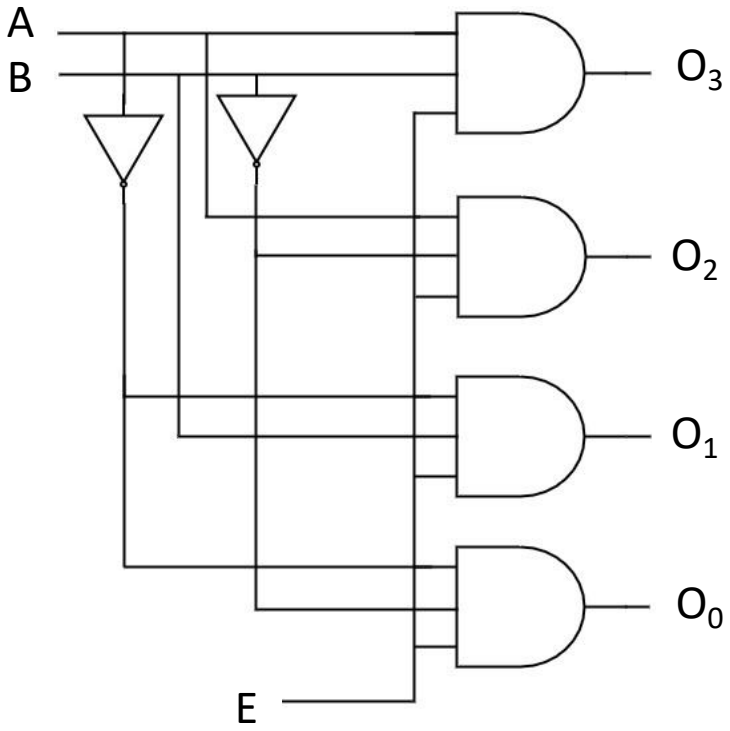
$C(x, y, z) = \sum(3,5,6,7)$



Decoder modeling in Verilog: 2-to-4 decoder

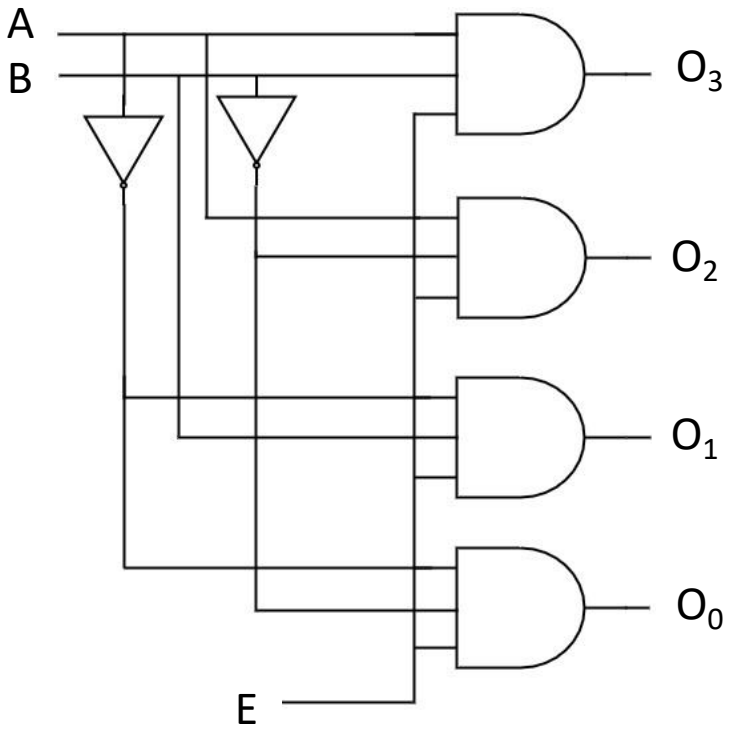


Decoder modeling in Verilog: 2-to-4 decoder



E	A	B
0	x	x
1	0	0
1	0	1
1	1	0
1	1	1

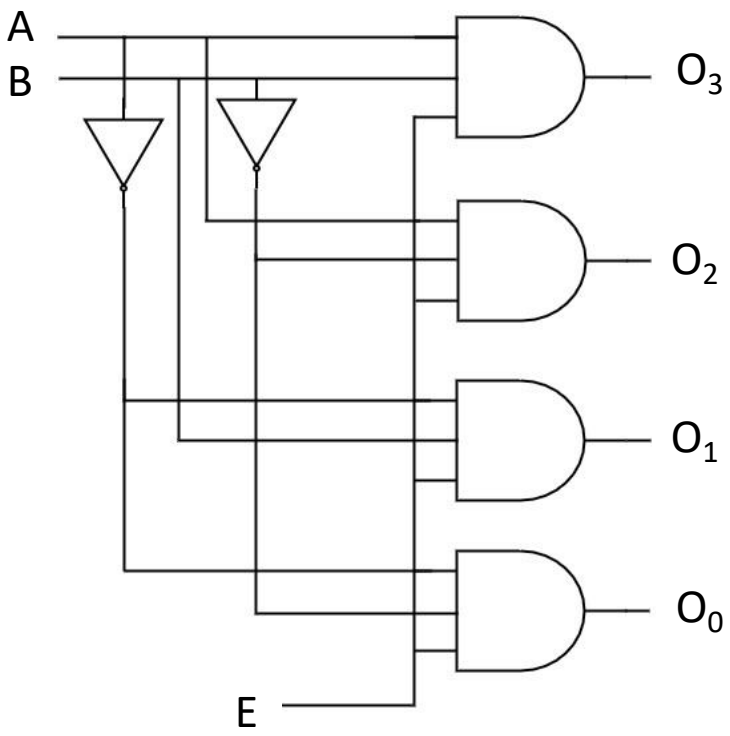
Decoder modeling in Verilog: 2-to-4 decoder



E	A	B
0	x	x
1	0	0
1	0	1
1	1	0
1	1	1

O_0	O_1	O_2	O_3
0	0	0	0
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Decoder modeling in Verilog: 2-to-4 decoder



E	A	B
0	x	x
1	0	0
1	0	1
1	1	0
1	1	1

O_0	O_1	O_2	O_3
0	0	0	0
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

```
module decoder2to4(input A, B, E,
                   output O0, O1, O2, O3);
   assign O0 = ((~A) & (~B) & (E));
   assign O1 = ((~A) & (B) & (E));
   assign O2 = ((A) & (~B) & (E));
   assign O3 = ((A) & (B) & (E));
endmodule
```

Decoder modeling in Verilog: 2-to-4 decoder

```
module decoder2to4_tb();
    reg A, B, E;
    wire O0, O1, O2, O3;
    decoder2to4(.A(A), .B(B), .E(E, .O0(O0), .O1(O1), .O2(O2), .O3(O3));
    initial begin
```

Time =0, A=x, B= x, E=x ➔ O_0=x, O_1 =x, O_2 =x, O_3 =x

Decoder modeling in Verilog: 2-to-4 decoder

```
module decoder2to4_tb();
    reg A, B, E;
    wire O0, O1, O2, O3;
    decoder2to4(.A(A), .B(B), .E(E, .O0(O0), .O1(O1), .O2(O2), .O3(O3));
    initial begin
     A = 0; B = 0; E = 0; #10; //apply inputs, wait 10ns
```

Initialization➔

Time =0, A=x, B= x, E=x ➔ O_0=x, O_1 =x, O_2 =x, O_3 =x

Time =0, A=0, B= 0, E=0 ➔ O_0=0, O_1 =0, O_2 =0, O_3 =0

Decoder modeling in Verilog: 2-to-4 decoder

```
module decoder2to4_tb();
    reg A, B, E;
    wire O0, O1, O2, O3;
    decoder2to4(.A(A), .B(B), .E(E, .O0(O0), .O1(O1), .O2(O2), .O3(O3));
    initial begin
     A = 0; B = 0; E = 0; #10; //apply inputs, wait 10ns
                   E = 1; #10; //change the value of c, wait 10ns
```

Initialization➔

Time =0, A=x, B= x, E=x ➔ O_0=x, O_1 =x, O_2 =x, O_3 =x

Time =0, A=0, B= 0, E=0 ➔ O_0=0, O_1 =0, O_2 =0, O_3 =0

Time =10, A=0, B= 0, E=1 ➔ O_0=1, O_1 =0, O_2 =0, O_3 =0

Decoder modeling in Verilog: 2-to-4 decoder

```
module decoder2to4_tb();
    reg A, B, E;
    wire O0, O1, O2, O3;
    decoder2to4(.A(A), .B(B), .E(E, .O0(O0), .O1(O1), .O2(O2), .O3(O3));
    initial begin
     A = 0; B = 0; E = 0; #10; //apply inputs, wait 10ns
                   E = 1; #10; //change the value of c, wait 10ns
            B = 1;        #10; //change the values of b, wait 10ns
```

Initialization ➔

Time =0, A=x, B= x, E=x ➔ O_0=x, O_1 =x, O_2 =x, O_3 =x

Time =0, A=0, B= 0, E=0 ➔ O_0=0, O_1 =0, O_2 =0, O_3 =0

Time =10, A=0, B= 0, E=1 ➔ O_0=1, O_1 =0, O_2 =0, O_3 =0

Time =20, A=0, B= 1, E=1 ➔ O_0=0, O_1 =1, O_2 =0, O_3 =0

Decoder modeling in Verilog: 2-to-4 decoder

```
module decoder2to4_tb();
    reg A, B, E;
    wire O0, O1, O2, O3;
    decoder2to4(.A(A), .B(B), .E(E, .O0(O0), .O1(O1), .O2(O2), .O3(O3));
    initial begin
     A = 0; B = 0; E = 0; #10; //apply inputs, wait 10ns
                    E = 1; #10; //change the value of c, wait 10ns
            B = 1;         #10; //change the values of b, wait 10ns
     A = 1; B = 0;         #10;
```

Initialization ➔

Time =0, A=x, B= x, E=x ➔ O_0=x, O_1 =x, O_2 =x, O_3 =x

Time =0, A=0, B= 0, E=0 ➔ O_0=0, O_1 =0, O_2 =0, O_3 =0

Time =10, A=0, B= 0, E=1 ➔ O_0=1, O_1 =0, O_2 =0, O_3 =0

Time =20, A=0, B= 1, E=1 ➔ O_0=0, O_1 =1, O_2 =0, O_3 =0

Time =30, A=1, B= 0, E=1 ➔ O_0=0, O_1 =0, O_2 =1, O_3 =0

Decoder modeling in Verilog: 2-to-4 decoder

```
module decoder2to4_tb();
    reg A, B, E;
    wire O0, O1, O2, O3;
    decoder2to4(.A(A), .B(B), .E(E, .O0(O0), .O1(O1), .O2(O2), .O3(O3));
    initial begin
     A = 0; B = 0; E = 0; #10; //apply inputs, wait 10ns
                    E = 1; #10; //change the value of c, wait 10ns
            B = 1;         #10; //change the values of b, wait 10ns
     A = 1; B = 0;         #10;
            B = 1;        #10;
    end


endmodule
```

Initialization ➔

Time =0, A=x, B= x, E=x ➔ O_0=x, O_1 =x, O_2 =x, O_3 =x

Time =0, A=0, B= 0, E=0 ➔ O_0=0, O_1 =0, O_2 =0, O_3 =0

Time =10, A=0, B= 0, E=1 ➔ O_0=1, O_1 =0, O_2 =0, O_3 =0

Time =20, A=0, B= 1, E=1 ➔ O_0=0, O_1 =1, O_2 =0, O_3 =0

Time =30, A=1, B= 0, E=1 ➔ O_0=0, O_1 =0, O_2 =1, O_3 =0

Time =40, A=1, B= 1, E=1 ➔ O_0=0, O_1 =0, O_2 =0, O_3 =1

Verilog restirictions

- Always blocks: always blocks may be used in testbenches, or to build registers.

Verilog restirictions

- Always blocks: always blocks may be used in testbenches, or to build registers.
  - always blocks for registers must be of the form always @ (posedge Clock) and must use nonblocking assignment <=

Verilog restirictions

- Always blocks: always blocks may be used in testbenches, or to build registers.
  - always blocks for registers must be of the form always @ (posedge Clock) and must use nonblocking assignment <=
  - You may not put any combination logic in a register
  - In always @ (posedge Clock) block may not contain even a simple counter

Verilog restirictions

- Always blocks: always blocks may be used in testbenches, or to build registers.
  - always blocks for registers must be of the form always @ (posedge Clock) and must use nonblocking assignment <=
  - You may not put any combination logic in a register
  - In always @ (posedge Clock) block may not contain even a simple counter
- Variable index:
  - Variable-indexed shifts (<< or >>) and bit-selects (x[y] or x[y:z]) are not allowed

Verilog restirictions

- Always blocks: always blocks may be used in testbenches, or to build registers.
  - always blocks for registers must be of the form always @ (posedge Clock) and must use nonblocking assignment <=
  - You may not put any combination logic in a register
  - In always @ (posedge Clock) block may not contain even a simple counter
- Variable index:
  - Variable-indexed shifts (<< or >>) and bit-selects (x[y] or x[y:z]) are not allowed
- Looping:
  - The keywords forever, repeat, while, for, fork and join may appear only in testbenches

Verilog restirictions

- Always blocks: always blocks may be used in testbenches, or to build registers.
  - always blocks for registers must be of the form always @ (posedge Clock) and must use nonblocking assignment <=
  - You may not put any combination logic in a register
  - In always @ (posedge Clock) block may not contain even a simple counter
- Variable index:
  - Variable-indexed shifts (<< or >>) and bit-selects (x[y] or x[y:z]) are not allowed
- Looping:
  - The keywords forever, repeat, while, for, fork and join may appear only in testbenches
- Macros:
  - The keyword ‘timescale must appear in every testbench, and nowhere else

Verilog common pitfalls

- If you program sequentially, the synthesizer may add a lot of hardware to try to do what you say
- If you program in parallel (multiple "always" blocks), you can get non-deterministic execution
  - Which "always" happens first?
- You create lots of state that you didn't intend

```
if (x == 1) out = 0;
if (y == 1) out = 1;    // else out retains previous state? R-S latch!
```

- You don't realize how much hardware you're specifying
  - x = x + 1 can be a LOT of hardware
- Slight changes may suddenly make your code "blow up"
  - A chip that previously fit suddenly is too large or slow

Verilog common pitfalls

- Several common pitfalls which trap the novice Verilog programs

Verilog common pitfalls

- Several common pitfalls which trap the novice Verilog programs
- Unlike Java or even C, Verilog does not check the types or widths of signals very seriously.

Verilog common pitfalls

- Several common pitfalls which trap the novice Verilog programs
- Unlike Java or even C, Verilog does not check the types or widths of signals very seriously.
- In the below cases, the compiler may not warn you of your mistake

Verilog common pitfalls

- Several common pitfalls which trap the novice Verilog programs
- Unlike Java or even C, Verilog does not check the types or widths of signals very seriously.
- In the below cases, the compiler may not warn you of your mistake

➢ Case1: Undeclared wires:
  - ➢ Any undeclared signal is automatically treated as a 1-bit wire
  - ➢ Assigning to undeclared signals from an always or initial block ➔ results in compiler errors
  - ➢ Forgetting to declare a bus or bit-vector will result in only the the 0^{th} being connected properly

Verilog common pitfalls

- Case2: width mismatch (small wires):
  - Connecting a small wire (e.g. 1-bit) to a large port (e.g., 4-bit) ➔ cause port width mismatch
  - Some of the signals intended for the bus will be lost
  - Only the lowest few bits will appear at the port

Verilog common pitfalls

- Case2: width mismatch (small wires):
  - Connecting a small wire (e.g. 1-bit) to a large port (e.g., 4-bit) ➔ cause port width mismatch
  - Some of the signals intended for the bus will be lost
  - Only the lowest few bits will appear at the port

- Case3: width mismatch (large wires):
  - Connecting a large wire (e.g., 4-bit) to a small port (e.g., 1-bit) ➔ cause port width mismatch
  - Many waveforms will appear in blue ➔ denoting undriven signal

Verilog vs system Verilog

Major areas of System Verilog improvement over Verilog

Verilog vs system Verilog

Major areas of System Verilog improvement over Verilog

1. SVD – System Verilog for Design ➔Enhancements to design constructs
   - E.g., more data types unum, struct, class etc

Verilog vs system Verilog

Major areas of System Verilog improvement over Verilog

1. SVD – System Verilog for Design ➔Enhancements to design constructs
   - E.g., more data types unum, struct, class etc
2. SVTB - System Verilog for Testbench ➔ biggest enhancement
   - Support all testbench modeling
   - Object oriented support
   - Creating constrained random stimulus, semaphore, concurrent process etc

Verilog vs system Verilog

Major areas of System Verilog improvement over Verilog

1. SVD – System Verilog for Design ➔Enhancements to design constructs
   - E.g., more data types unum, struct, class etc
2. SVTB - System Verilog for Testbench ➔ biggest enhancement
   - Support all testbench modeling
   - Object oriented support
   - Creating constrained random stimulus, semaphore, concurrent process etc
3. SVA - System Verilog Assertions ➔ features for temporal and concurrent assertions

Verilog vs system Verilog

Major areas of System Verilog improvement over Verilog

1. SVD – System Verilog for Design ➔Enhancements to design constructs
   - E.g., more data types unum, struct, class etc
2. SVTB - System Verilog for Testbench ➔ biggest enhancement
   - Support all testbench modeling
   - Object oriented support
   - Creating constrained random stimulus, semaphore, concurrent process etc
3. SVA - System Verilog Assertions ➔ features for temporal and concurrent assertions
4. SVDPI - System Verilog Direct Program Interface ➔ features for better C/C++ integration

Verilog vs system Verilog

Major areas of System Verilog improvement over Verilog

1. SVD – System Verilog for Design ➔Enhancements to design constructs
   - E.g., more data types unum, struct, class etc
2. SVTB - System Verilog for Testbench ➔ biggest enhancement
   - Support all testbench modeling
   - Object oriented support
   - Creating constrained random stimulus, semaphore, concurrent process etc
3. SVA - System Verilog Assertions ➔ features for temporal and concurrent assertions
4. SVDPI - System Verilog Direct Program Interface ➔ features for better C/C++ integration
5. SVAPI - System Verilog Application Program Interface ➔ features for better integration of APIs

Thank you