

FSM PRIMER

FPGA/VERILOG FROM SCRATCH

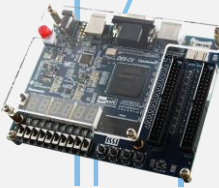
DAY 3

C3.4: FSM DESIGN WITH VERILOG HDL

- ❏ A combinational circuit produces output solely depending on the current input.
- ❏ But a sequential circuit “remembers” its previous state.
- ❏ Its output depends on present inputs and previous state.

❏ Examples:

- ❏ Latches
- ❏ Registers
- ❏ Memory
- ❏ parallel to serial / serial to parallel converters
- ❏ Counters



SUMMARY: DEFINING A MODULE

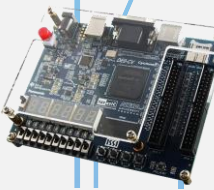
❏ A module is the main building block in Verilog

❏ We first need to declare:

❏ Name of the module

❏ Types of its connections (input, output)

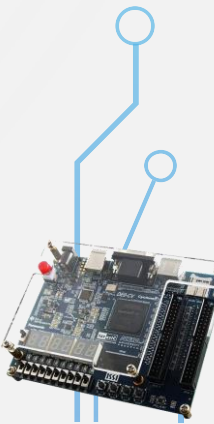
❏ Names of its connections



SUMMARY: DEFINING A MODULE



```
module example (a, b, c, y);  
    input  a;  
    input  b;  
    output y;  
  
    // here comes the circuit description  
  
endmodule
```

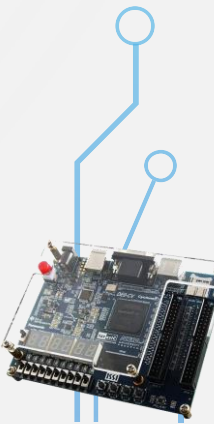


SUMMARY: WHAT IF WE HAVE BUSSES ?

 You can also define multi-bit busses.

 [range_start : range_end]

```
input  [31:0] a; // a[31], a[30] .. a[0]
output [15:8] b1; // b1[15], b1[14] .. b1[8]
output [7:0]  b2; // b2[7], b2[6] .. b1[0]
input          clk;
```



STRUCTURAL HDL EXAMPLE

```

module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;

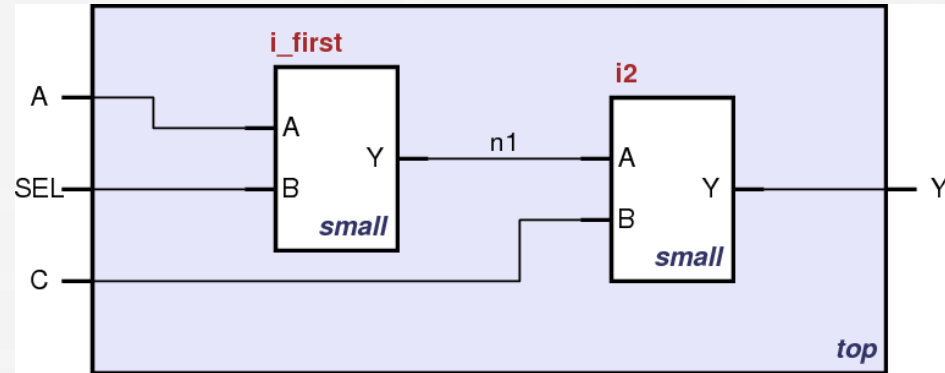
  // alternative
  small i_first ( A, SEL, n1 );

  /* Shorter instantiation,
     pin order very important */

  // any pin order, safer choice
  small i2 ( .B(C),
             .Y(Y),
             .A(n1) );

endmodule

```



```

module small (A, B, Y);
  input A;
  input B;
  output Y;

  // description of small

endmodule

```



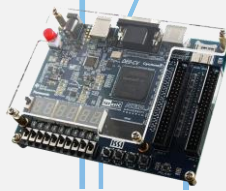
SUMMARY: BITWISE OPERATORS

```
module gates(input  [3:0] a, b,
              output [3:0] y1, y2, y3, y4, y5);
```

```
/* Five different two-input logic
   gates acting on 4 bit busses */
```

```
assign y1 = a & b;    // AND
assign y2 = a | b;    // OR
assign y3 = a ^ b;    // XOR
assign y4 = ~(a & b); // NAND
assign y5 = ~(a | b); // NOR
```

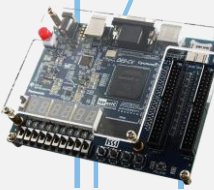
```
endmodule
```



SUMMARY: CONDITIONAL ASSIGNMENT

 **? :** is also called a ternary operator because it operates on 3 inputs

```
module mux2(input  [3:0] d0, d1,  
             input      s,  
             output [3:0] y);  
  
    assign y = s ? d1 : d0;  
    // if (s) then y=d1 else y=d0;  
  
endmodule
```

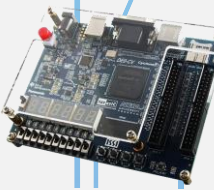


SEQUENTIAL LOGIC IN VERILOG

- ❏ **Define blocks that have memory**
 - ❏ Flip-Flops, Latches, Finite State Machines

- ❏ **Sequential Logic is triggered by a 'CLOCK' event**
 - ❏ Latches are sensitive to level of the signal
 - ❏ Flip-flops are sensitive to the transitioning of clock

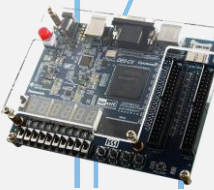
- ❏ **Combinational constructs are not sufficient**
 - ❏ We need new constructs:
 - ❏ **always**
 - ❏ **initial**



ALWAYS STATEMENT, DEFINING PROCESSES

 Whenever the event in the sensitivity list occurs, the statement is executed

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



EXAMPLE: D FLIP-FLOP

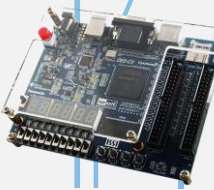
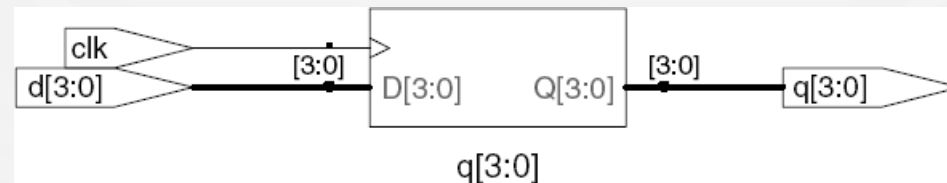
```

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

    always @(posedge clk)
        q <= d;                                // pronounced "q gets d"

endmodule

```



EXAMPLE: D FLIP-FLOP

```

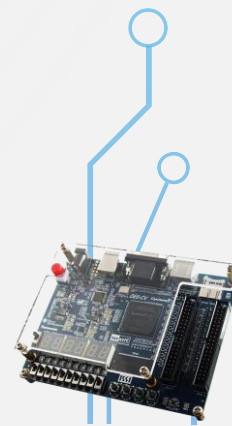
module flop(input          clk,
            input  [3:0] d,
            output reg [3:0] q);

    always @(posedge clk)
        q <= d; // pronounced "q gets d"

endmodule

```

- ❏ The **posedge** defines a rising edge (transition from 0 to 1).
- ❏ This process will trigger only if the **clk** signal rises.
- ❏ Once the clk signal rises: the value of **d** will be copied to **q**



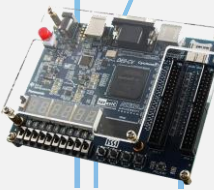
EXAMPLE: D FLIP-FLOP

```
module flop(input          clk,  
            input  [3:0] d,  
            output reg [3:0] q);  
  
    always @(posedge clk)  
        q <= d;           // pronounced "q gets d"  
  
endmodule
```

❗ 'assign' statement is not used within always block

❗ The <= describes a 'non-blocking' assignment

❗ We will see the difference between 'blocking assignment' and 'non-blocking' assignment in a while



EXAMPLE: D FLIP-FLOP

```
module flop(input          clk,  
            input          [3:0] d,  
            output reg [3:0] q);  
  
    always @(posedge clk)  
        q <= d;                // pronounced "q gets d"  
  
endmodule
```

- Assigned variables need to be declared as reg
- The name reg does not necessarily mean that the value is a register. (It could be, it does not have to be).



D FLIP-FLOP WITH ASYNCHRONOUS RESET



```

module flop_ar (input          clk,
                 input          reset,
                 input [3:0] d,
                 output reg [3:0] q);

    always @(posedge clk, negedge reset)
    begin
        if (reset == '0') q <= 0;    // when reset
        else               q <= d;    // when clk
    end
endmodule

```

 In this example: two events can trigger the process:

-  A rising edge on clk
-  A falling edge on reset



D FLIP-FLOP WITH ASYNCHRONOUS RESET

```

module flop_ar (input          clk,
                input          reset,
                input  [3:0] d,
                output reg [3:0] q);

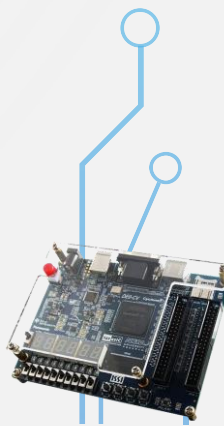
    always @(posedge clk, negedge reset)
    begin
        if (reset == '0') q <= 0;    // when reset
        else               q <= d;    // when clk
    end
endmodule

```

🔧 For longer statements a begin end pair can be used

🔧 In this example it was not necessary

🔧 The always block is highlighted



D FLIP-FLOP WITH ASYNCHRONOUS RESET

```

module flop_ar (input          clk,
                input          reset,
                input [3:0] d,
                output reg [3:0] q);

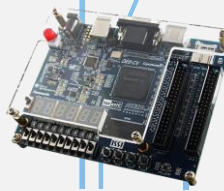
    always @(posedge clk, negedge reset)
    begin
        if (reset == '0') q <= 0; // when reset
        else                q <= d; // when clk
    end
endmodule

```

🖨️ First reset is checked, if reset is 0, q is set to 0.

🖨️ This is an 'asynchronous' reset as the reset does not care what happens with the clock

🖨️ If there is no reset then normal assignment is made



D FLIP-FLOP WITH SYNCHRONOUS RESET

```

module flop_sr (input          clk,
                 input          reset,
                 input  [3:0] d,
                 output reg [3:0] q);

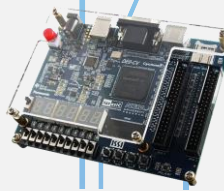
    always @(posedge clk)
    begin
        if (reset == '0') q <= 0;    // when reset
        else               q <= d;    // when clk
    end
endmodule

```

❏ The process is only sensitive to clock

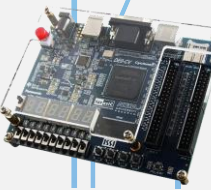
❏ Reset only happens when the clock rises. This is a 'synchronous' reset

❏ A small change, has a large impact on the outcome



SUMMARY: SEQUENTIAL STATEMENTS SO FAR

- Sequential statements are within an 'always' block
- The sequential block is triggered with a change in the sensitivity list
- Signals assigned within an always must be declared as reg
- We use \leq for (non-blocking) assignments and do not use 'assign' within the always block.



SUMMARY: BASICS OF ALWAYS STATEMENTS

 You can have many always blocks

```

module    example (input          clk,
                    input    [3:0] d,
                    output reg [3:0] q);

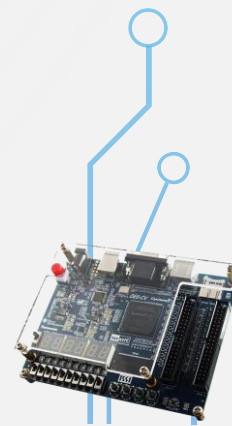
    wire    [3:0] normal;           // standard wire
    reg     [3:0] special;          // assigned in always

    always @(posedge clk)
        special <= d;               // first FF array

    assign normal = ~ special; // simple assignment

    always @(posedge clk)    q
        <= normal;           // second FF array
endmodule

```



SUMMARY: BASICS OF ALWAYS STATEMENTS

 **Assignments are different within always blocks**

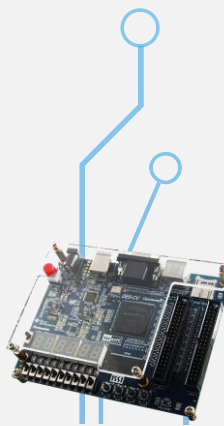
```
module example (input          clk,
                 input  [3:0] d,
                 output reg [3:0] q);

  wire [3:0] normal;           // standard wire
  reg  [3:0] special;          // assigned in always

  always @(posedge clk)
    special <= d;              // first FF array

  assign normal = ~ special;   // simple assignment

  always @(posedge clk) q
    <= normal;                 // second FF array
endmodule
```



WHY DOES AN ALWAYS STATEMENT MEMORIZE?

```
module flop (input          clk,  
             input    [3:0] d,  
             output reg [3:0] q);  
  
    always @(posedge clk)  
    begin  
        q <= d;    // when clk rises copy d to q  
    end  
endmodule
```

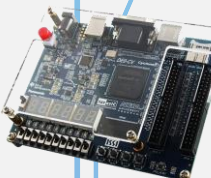
- ❏ This statement describes what happens to signal q
- ❏ ... but what happens when clock is not rising?



WHY DOES AN ALWAYS STATEMENT MEMORIZE?

```
module flop (input          clk,  
             input    [3:0] d,  
             output reg [3:0] q);  
  
    always @(posedge clk)  
    begin  
        q <= d;    // when clk rises copy d to q  
    end  
endmodule
```

- ❏ This statement describes what happens to signal q
- ❏ ... but what happens when clock is not rising?
- ❏ **The value of q is preserved (memorized)**



WHY DOES AN ALWAYS STATEMENT MEMORIZE?

```

module comb (input          inv,
              input    [3:0] data,
              output reg [3:0] result);

    always @(inv, data)           // trigger with inv, data
    if (inv) result <= ~data;    // result is inverted data
    else      result <= data;    // result is data

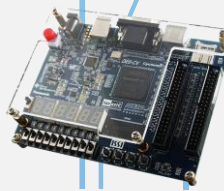
endmodule

```

🔧 This statement describes what happens to signal result

🔧 When inv is 1, result is ~data

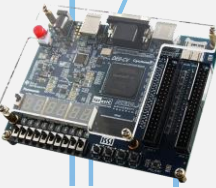
🔧 What happens when inv is not 1 ?



WHY DOES AN ALWAYS STATEMENT MEMORIZE?

```
module comb (input          inv,  
             input    [3:0] data,  
             output reg [3:0] result);  
  
    always @(inv, data)          // trigger with inv, data  
    if (inv) result <= ~data; // result is inverted data  
    else      result <= data; // result is data  
  
endmodule
```

- ❏ This statement describes what happens to signal result
 - ❏ When inv is 1, result is ~data
 - ❏ When inv is not 1, result is data
- ❏ Circuit is combinational (no memory)
 - ❏ The output (result) is defined for all possible inputs (inv data)



ALWAYS BLOCKS FOR COMBINATIONAL CIRCUITS

❏ If the statements define the signals completely, nothing is memorized, block becomes combinational.

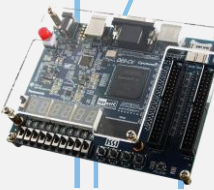
❏ Care must be taken, it is easy to make mistakes and unintentionally describe memorizing elements (latches).

❏ Always blocks allow powerful statements

❏ If

❏ Case

❏ Use always blocks only if it makes your job easier



ALWAYS STATEMENT IS NOT ALWAYS PRACTICAL...

```
reg [31:0] result;
wire [31:0] a, b, comb;
wire      sel,

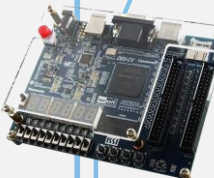
always @(a, b, sel)    // trigger with a, b, sel
if (sel) result <= a; // result is a
else    result <= b; // result is b

assign comb = sel ? a : b;

endmodule
```

Both statements describe the same multiplexer

In this case, the always block is more work



SOMETIMES ALWAYS STATEMENTS ARE GREAT

```

module sevensegment (input [3:0] data,
                     output reg [6:0] segments);

    always @(*) // * is short for all signals
    case (data) // case statement
        0: segments = 7'b111_1110; // when data is 0
        1: segments = 7'b011_0000; // when data is 1
        2: segments = 7'b110_1101;
        3: segments = 7'b111_1001;
        4: segments = 7'b011_0011;
        5: segments = 7'b101_1011;
        // etc etc
        default: segments = 7'b000_0000; // required
    endcase

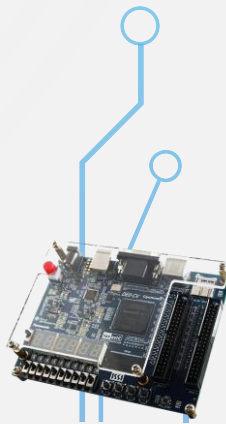
endmodule

```



THE CASE STATEMENT

- 🔧 Like **if.. then ..else** can only be used in **always** blocks
- 🔧 The result is combinational only if the output is defined for all cases
 - 🔧 Did we mention this before ?
- 🔧 Always use a **default** case to make sure you did not forget a case (which would infer a latch)



NON-BLOCKING AND BLOCKING STATEMENTS

Non-blocking

```
always @(a)
begin
    a <= 2'b01;
    b <= a;
    // all assignments are made here
    // b is not (yet) 2'b01
end
```

🖨 Values are assigned at the end of the block.

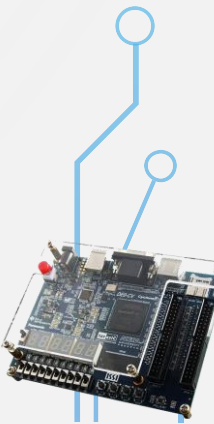
🖨 All assignments are made in parallel, process flow is **not-blocked**.

Blocking

```
always @(a)
begin
    a = 2'b01;
    // a is 2'b01
    b = a;
    // b is now 2'b01 as well
end
```

🖨 Value is assigned immediately.

🖨 Process waits until the first assignment is complete, it **blocks** progress.



WHY USE (NON)-BLOCKING STATEMENTS

- There are technical reasons why both are required

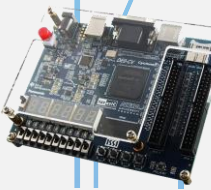
- It is out of the scope of this course to discuss these

- Blocking statements allow sequential descriptions

- More like a programming language

- If the sensitivity list is correct, blocks with non-blocking statements will always evaluate to the same result

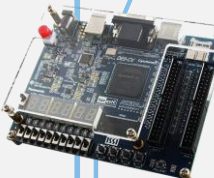
- It may require some additional iterations



EXAMPLE: BLOCKING STATEMENTS

 Assume all inputs are initially '0'

```
always @ ( * )  
begin  
    p    = a ^ b ;           // p    = 0  
    g    = a & b ;           // g    = 0  
    s    = p ^ cin ;         // s    = 0  
    cout = g | (p & cin) ; // cout = 0  
end
```



EXAMPLE: BLOCKING STATEMENTS

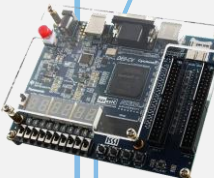
Now **a** changes to '1'

```
always @ ( * )  
begin  
    p    = a ^ b ;           // p    = 1  
    g    = a & b ;           // g    = 0  
    s    = p ^ cin ;         // s    = 1  
    cout = g | (p & cin) ;    // cout = 0  
end
```

The process triggers

All values are updated in order

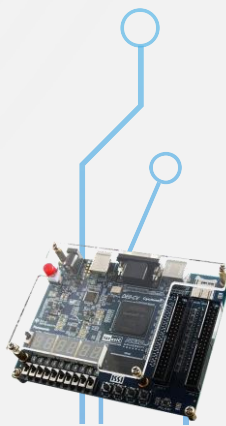
At the end, **s = 1**



SAME EXAMPLE: NON-BLOCKING STATEMENTS

 Assume all inputs are initially '0'

```
always @( * )  
begin  
    p    <= a ^ b ;           // p    = 0  
    g    <= a & b ;           // g    = 0  
    s    <= p ^ cin ;        // s    = 0  
    cout <= g | (p & cin) ; // cout = 0  
end
```



SAME EXAMPLE: NON-BLOCKING STATEMENTS

Now **a** changes to '1'

```
always @( * )  
begin  
    p    <= a ^ b ;           // p    = 1  
    g    <= a & b ;           // g    = 0  
    s    <= p ^ cin ;         // s    = 0  
    cout <= g | (p & cin) ;    // cout = 0  
end
```

The process triggers

All assignments are concurrent

When **s** is being assigned, **p** is still 0, **result** is still 0



SAME EXAMPLE: NON-BLOCKING STATEMENTS

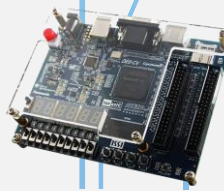
After the first iteration **p** has changed to '1' as well

```
always @( * )
begin
    p    <= a ^ b ;           // p    = 1
    g    <= a & b ;           // g    = 0
    s    <= p ^ cin ;         // s    = 1
    cout <= g | (p & cin) ;    // cout = 0
end
```

Since there is a change in **p**, process triggers again

This time **s** is calculated with **p=1**

The result is correct after the second iteration



RULES FOR SIGNAL ASSIGNMENT

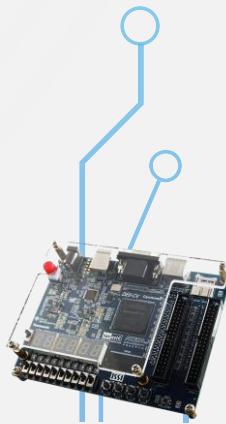
🖨️ Use **always@(posedge clk)** and non-blocking assignments (**<=**) to model synchronous sequential logic

always @ (posedge clk)

```
q <= d;      // nonblocking
```

🖨️ Use continuous assignments (**assign ...**) to model simple combinational logic.

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



RULES FOR SIGNAL ASSIGNMENT (CONT)

🔧 Use always @ (*) and blocking assignments (=) to model more complicated combinational logic where the always statement is helpful.

🔧 Do not make assignments to the same signal in more than one always statement or continuous assignment statement

Adapted from Verilog for Sequential Circuits, Srdjan Capkun, Design of Digital Circuits 2014

