

# Digital System Design with HDL (I)

## Lecture 5

Dr. Ming Xu and Dr. Kain Lu Low  
Dept of Electrical & Electronic Engineering  
XJTLU

# In This Session

- Primitive Instantiations
- Continuous Assignments
- Procedural Statements
  - Procedural Assignment Statements

# Primitive Instantiations

A circuit can be described using predefined modules.

```
gate_type [instance_name] (output_port, input_port {,  
input_port });
```

```
gate_type # (delay) [instance_name] (output_port,  
input_port {, input_port });
```

- *gate\_type*: the type of gate
- *Instance\_name*: optional.
- *delay*: propagation delay in time units.

# Primitive Instantiations

Example: Structural specification of a full-adder

```
module fulladd (Cin, x, y, s, Cout);  
  input Cin, x, y;  
  output s, Cout;  
  wire z1, z2, z3, z4;  
  
  and And1 (z1, x, y);  
  and And2 (z2, x, Cin);  
  and And3 (z3, y, Cin);  
  or Or1 (Cout, z1, z2, z3);  
  xor Xor1 (z4, x, y);  
  xor Xor2 (s, z4, Cin);  
  
endmodule
```

```
module fulladd (Cin, x, y, s, Cout);  
  input Cin, x, y;  
  output s, Cout;  
  
  and (z1, x, y);  
  and (z2, x, Cin);  
  and (z3, y, Cin);  
  or (Cout, z1, z2, z3);  
  xor (z4, x, y);  
  xor (s, z4, Cin);  
  
endmodule
```

# Primitive Instantiations

## Gate Types

Name	Description	Usage
and	$f = (a \cdot b \cdots)$	<b>and</b> ( $f, a, b, \dots$ )
nand	$f = \overline{(a \cdot b \cdots)}$	<b>nand</b> ( $f, a, b, \dots$ )
or	$f = (a + b + \cdots)$	<b>or</b> ( $f, a, b, \dots$ )
nor	$f = \overline{(a + b + \cdots)}$	<b>nor</b> ( $f, a, b, \dots$ )
xor	$f = (a \oplus b \oplus \cdots)$	<b>xor</b> ( $f, a, b, \dots$ )
xnor	$f = (a \odot b \odot \cdots)$	<b>xnor</b> ( $f, a, b, \dots$ )
not	$f = \bar{a}$	<b>not</b> ( $f, a$ )

# Primitive Instantiations

## Gate Types

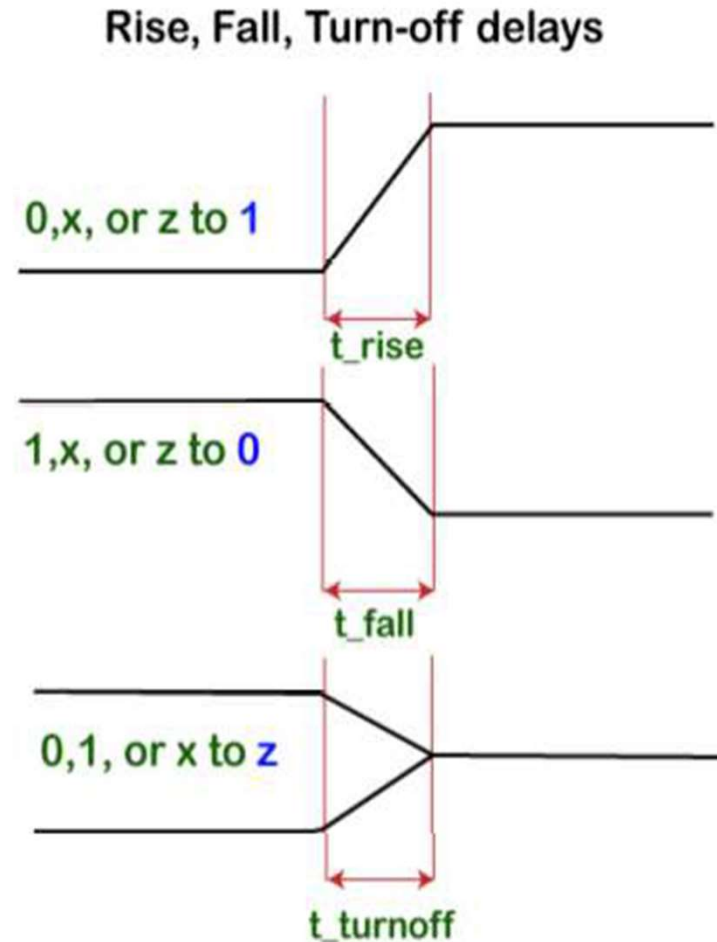
- **notif** and **bufif** gates are tri-state buffers.

Name	Description	Usage
buf	$f = a$	<b>buf</b> ( $f, a$ )
notif0	$f = (!e ? \bar{a} : 'bz)$	<b>notif0</b> ( $f, a, e$ )
notif1	$f = (e ? \bar{a} : 'bz)$	<b>notif1</b> ( $f, a, e$ )
bufif0	$f = (!e ? a : 'bz)$	<b>bufif0</b> ( $f, a, e$ )
bufif1	$f = (e ? a : 'bz)$	<b>bufif1</b> ( $f, a, e$ )

# Primitive Instantiations

## Delays

- Up to three delays can be defined.
- Rise delay: the output changes to 1.
- Fall delay: the output changes to 0.
- Turn-off delay: the output changes to z.



# Primitive Instantiations

## Delays

- If no delays are defined, the default delays are 0.
- If only one delay is defined, it is for all.
- If two delays are defined, they are rise and fall delays.
- If three delays are defined, they are rise, fall and turn-off delays, which can only be used for buffers.

## Examples

and #(2) o1 (out1, a, b);

and #(2, 3) o2 (out2, a, b);

bufif0 #(5, 6, 7) o3 (out3, a, b);



# Continuous Assignments

- Continuous assignments model combinational logic.
- Each time a signal on the right-hand side changes, the net on the left-hand side is re-evaluated.

## **Explicit** Continuous Assignment

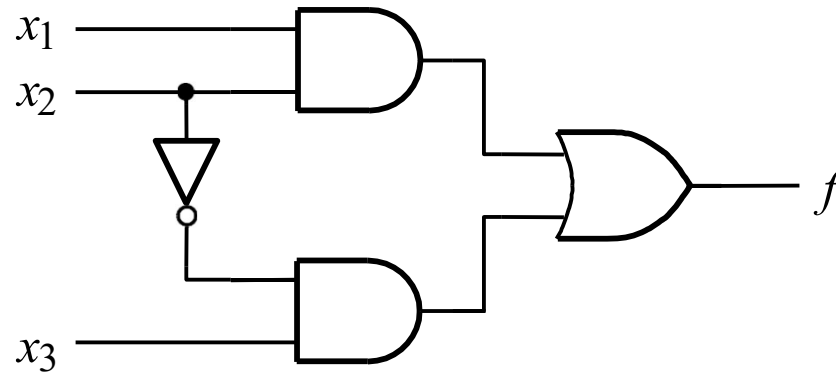
```
net_type [size] net_name;  
assign net_name = expression;
```

## **Implicit** Continuous Assignment

```
net_type [size] net_name = expression;
```

# Continuous Assignments

Example



```
module example3 (x1, x2, x3, f);  
    input x1, x2, x3;  
    output f;  
  
    assign f = (x1 & x2) | (~x2 & x3);  
  
endmodule
```

# Continuous Assignments

- Multiple assignments can be specified in one assign statement (separated by commas)

**assign** Cout = (x & y) | (x & Cin) | (y & Cin),  
          s = x ^ y ^ z;

- Multiple assignments can be combined with a net (wire, tri) declaration.

**wire** s = x ^ y,  
      c = x & y;

- Continuous assignments are **concurrent** statements; their ordering in the code does not matter.

# Procedural Statements

- Procedural statements are evaluated in the order in which they appear.
- Procedural statements must be contained in **always** (or **initial**) blocks, or **function** or **task** blocks that are called only from inside **always** (or **initial**) blocks.
- **always** procedural blocks process statements *repeatedly*.
- **initial** procedural blocks process statements *one time*.

# always Block

**always** @(sensitivity\_list)

**[begin]**

[procedural assignment statements]

[programming constructs]

**[end]**

- If the value of a signal in the *sensitive list* changes, all the statements are evaluated sequentially.
- The signals are separated by keyword **or** or comma in Verilog 2001.

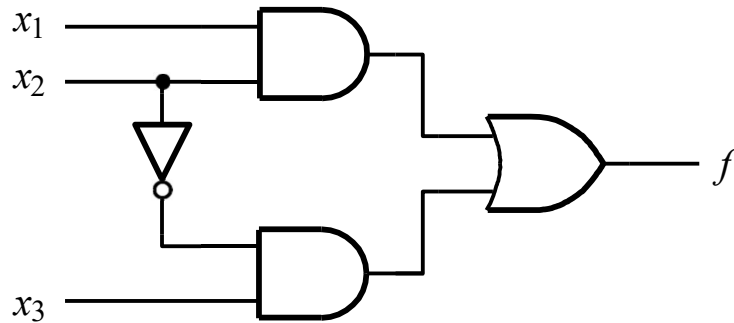
# always Block

- Level-sensitive **always** block
  - **always @** (signal1 **or** signal2)
  - Used for combinational circuits and latches
- Edge-sensitive **always** block
  - **always @** (**posedge** clk)
    - Response to positive edge of clk signal
  - **always @** (**negedge** clk)
    - Response to negative edge of clk signal
  - Used for sequential circuits and flip-flops

# always Block

## Example:

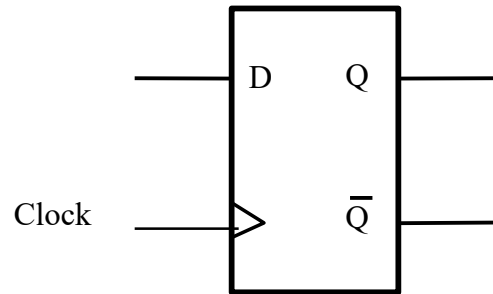
a combinational logic circuit



```
module example5 (x1, x2, x3, f);  
  input x1, x2, x3;  
  output f;  
  reg f;  
  
  always @(x1 or x2 or x3)  
    if (x2 == 1)  
      f = x1;  
    else  
      f = x3;  
  
endmodule
```

# always Block

**Example:** D-type flip-flop



```
module flipflop (D, Clock, Q);  
  input D, Clock;  
  output Q;  
  reg Q;  
  
  always @(posedge Clock)  
    Q = D;  
  
endmodule
```



# Procedural Assignment Statements

- **=**  
**Blocking assignments** – evaluate *in order*  
    **begin**  
        Q1 = D;  
        Q2 = Q1; // new Q1 goes to Q2.  
    **end**
- **<=**  
**No-blocking assignments** – evaluate *in parallel*  
    **begin**  
        Q1<= D;  
        Q2<= Q1; // old Q1 goes to Q2  
    **end**

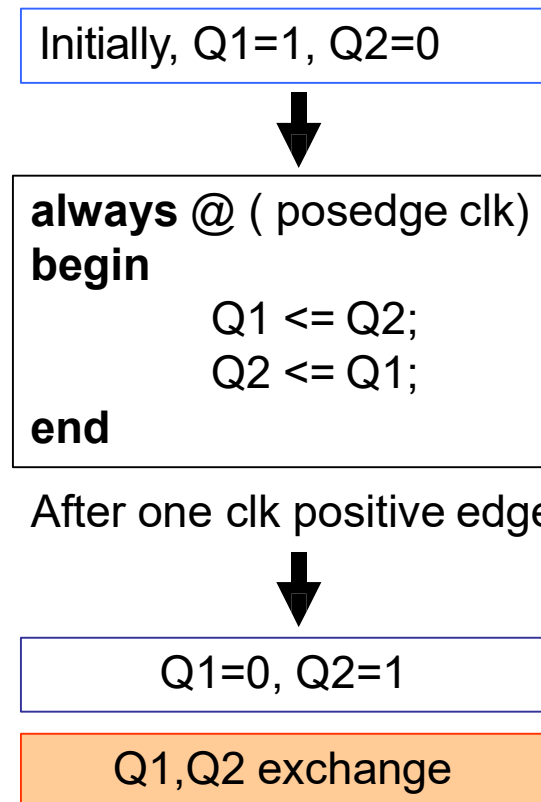
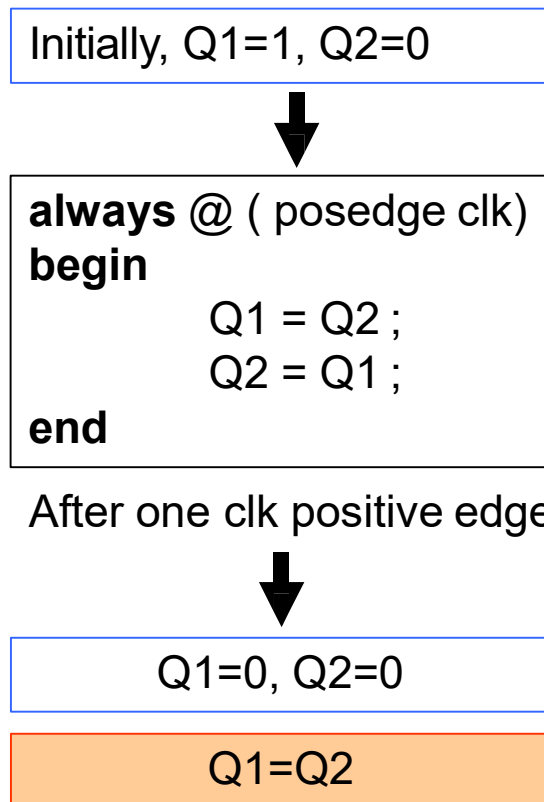
The order of statements doesn't matter

# Procedural Assignment Statements

- For *blocking assignments*, the values of variables in each statement are the new values set by any preceding statements in the **always** block.
- For *non-blocking assignments*, the values of variables are the values at the beginning of the **always** block.
- Although the statements inside each **always** block are evaluated in order, each entire **always** block is still like a concurrent statement.

# Procedural Assignment Statements

## Blocking and non-blocking assignments

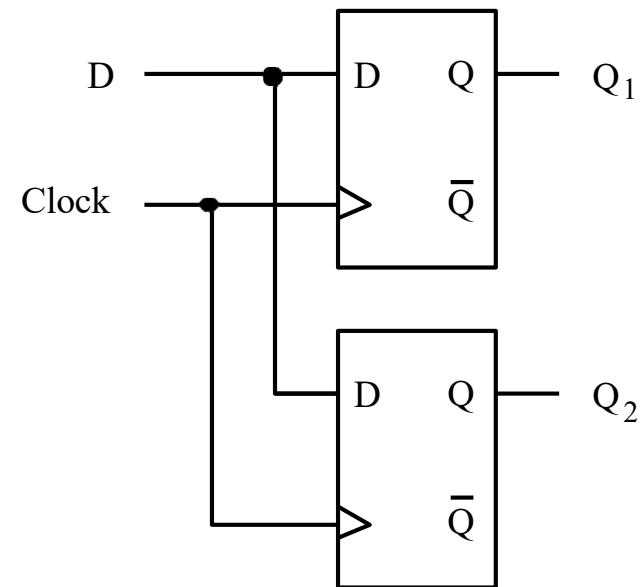


What happens  
if we change the  
order of two  
statements?

# Procedural Assignment Statements

## Example: Blocking assignments

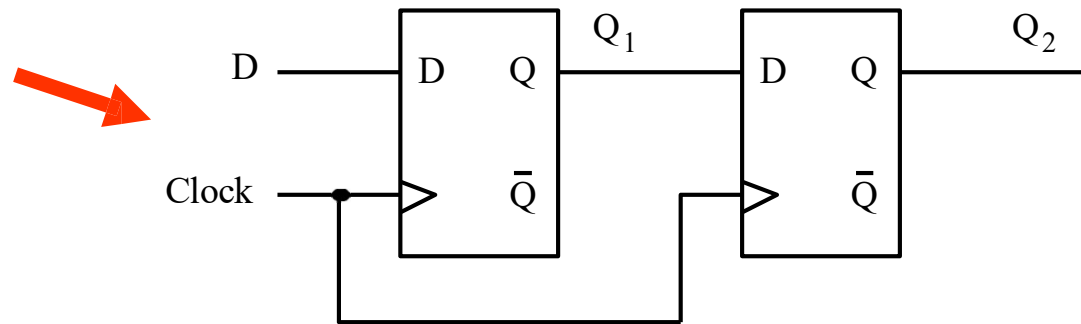
```
module example7_3 (D, Clock, Q1, Q2);  
  input D, Clock;  
  output Q1, Q2;  
  reg Q1, Q2;  
  
  always @(posedge Clock)  
  begin  
    Q1 = D;  
    Q2 = Q1;  
  end  
  
endmodule
```



# Procedural Assignment Statements

**Example:** No-blocking assignments

```
module example7_4 (D, Clock, Q1, Q2);  
  input D, Clock;  
  output Q1, Q2;  
  reg Q1, Q2;  
  
  always @(posedge Clock)  
  begin  
    Q1 <= D;  
    Q2 <= Q1;  
  end  
  
endmodule
```



# Procedural Assignment Statements

## Recommendations

- It is better to use **blocking** assignments when describing **combinational** circuits
- It is better to using **no-blocking** assignments to describe **sequential** circuits