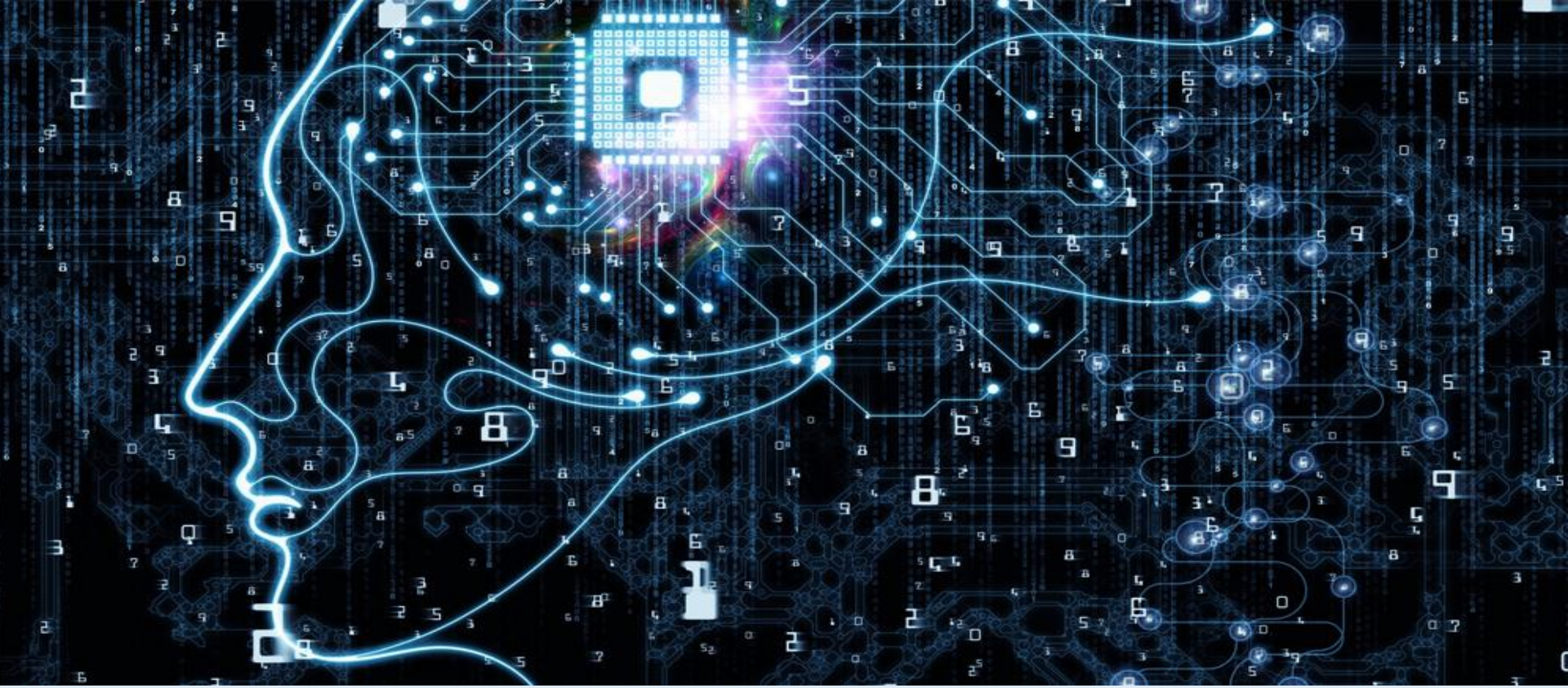


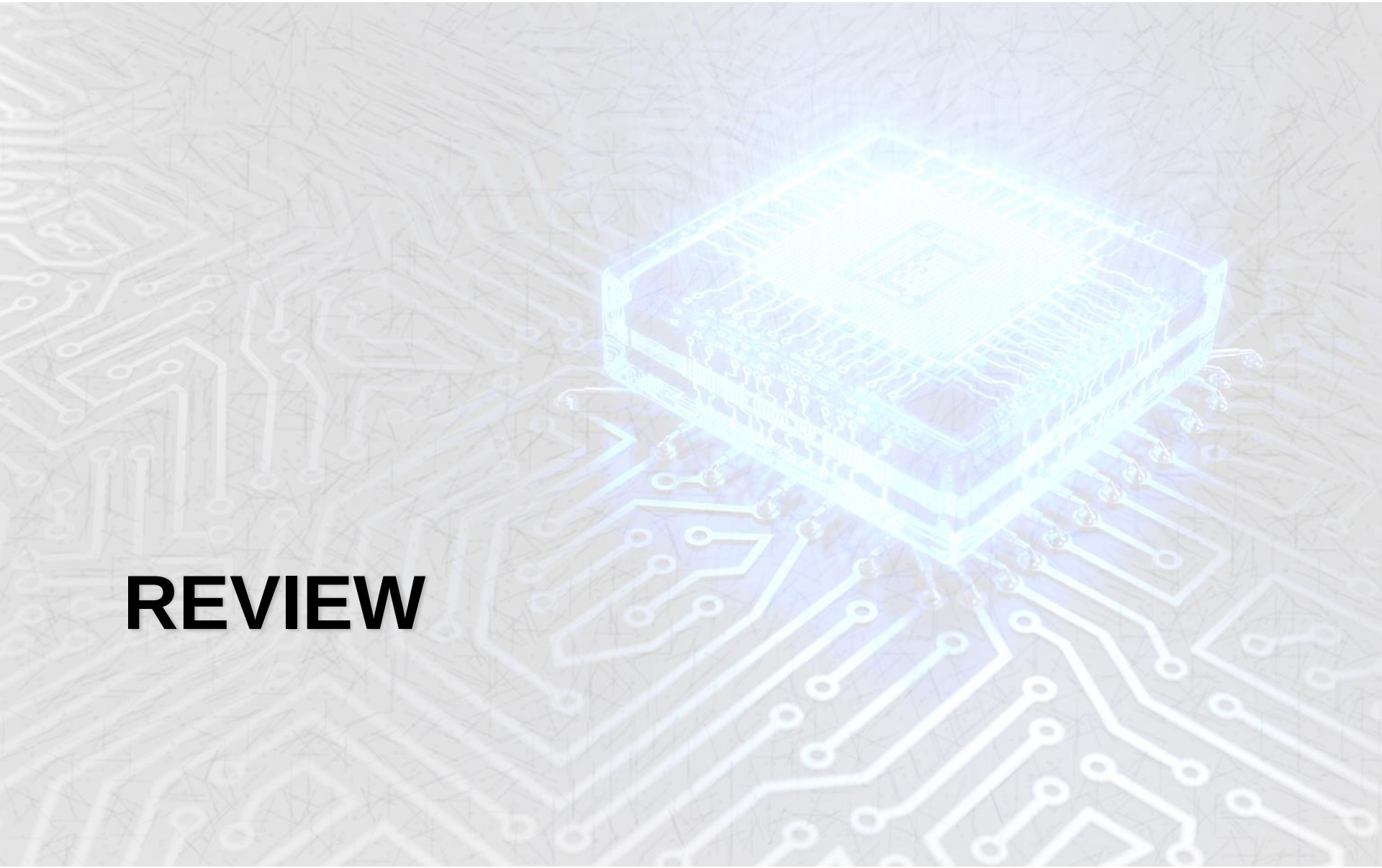
ALL PROGRAMMABLE

Hardware Description Language - Verilog

Sequential Circuits

Outline

- Review
  - Module Instantiation
  - Ports Connection
  - Number Representation
  - Assignments
  - Operators
- Sequential Circuit
  - Introduction
  - Counter
- Sequential Verilog Syntax
  - Procedural Assignment
  - Non-blocking
  - Example



REVIEW

Module Declaration

- Encapsulate structural and functional details in a module

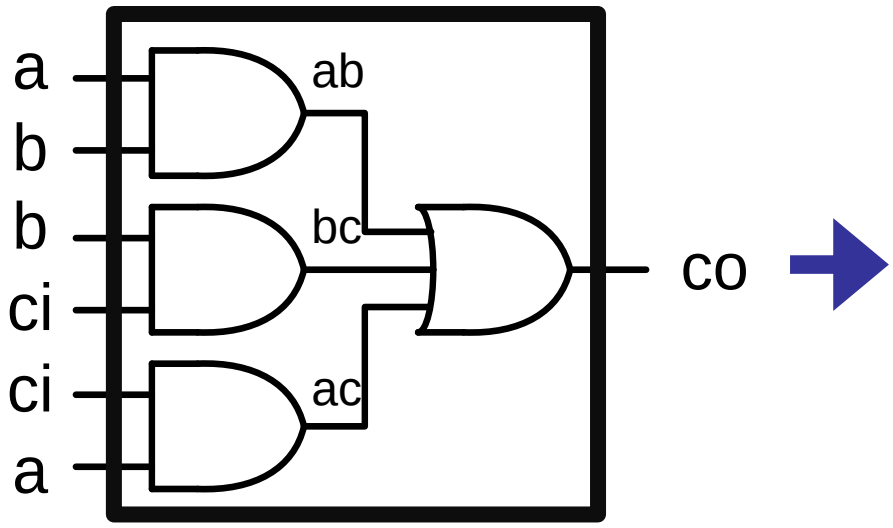
```
module <Module Name> (<PortName List>);

 // Structural part
      <List of Ports>
      <Lists of Nets and Registers>
      <SubModule List>  <SubModule Connections>

// Behavior part
      <Timing Control Statements>
            <Parameter/Value Assignments>
      <Stimuli>
      <System Task>
endmodule
```

```
module adder(out,in1,in2);
    output  out;
    input   in1,in2    ;

    assign  out=in1 + in2;
endmodule
```

- Encapsulation makes the model available for instantiation in other modules

Ports Declaration

- Two port types
  - Input port
    - input a;
  - Output port
    - output b;
- Two net types
  - **wire (can be used for module connection)**
    - wire c;
  - reg
    - reg d;




```
module FA_co( co, a, b, ci);
        output co;
        input a, b, ci;
        wire ab, bc, ac;

        and g0 ( ab, a, b );
        and g1 ( bc, b, ci );
        and g2 ( ac, ci, a );
        or   g3 ( co, ab, bc, ac );
endmodule
```

Module Instantiation

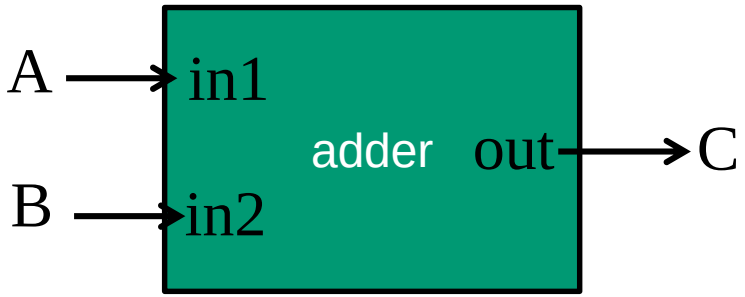
- A module provides a template from which you can create actual objects.
- When a module is invoked, Verilog creates a unique object from the template.
- Each object has its own name, variables, parameters and I/O interface.

```
module adder_tree (out0,out1,in1,in2,in3,in4);
    output  out0,out1;
    input   in1,in2,in3,in4;

    adder   add_0 (out0,in1,in2);
    adder   add_1 (out1,in3,in4);
endmodule
```

instance name IO interface

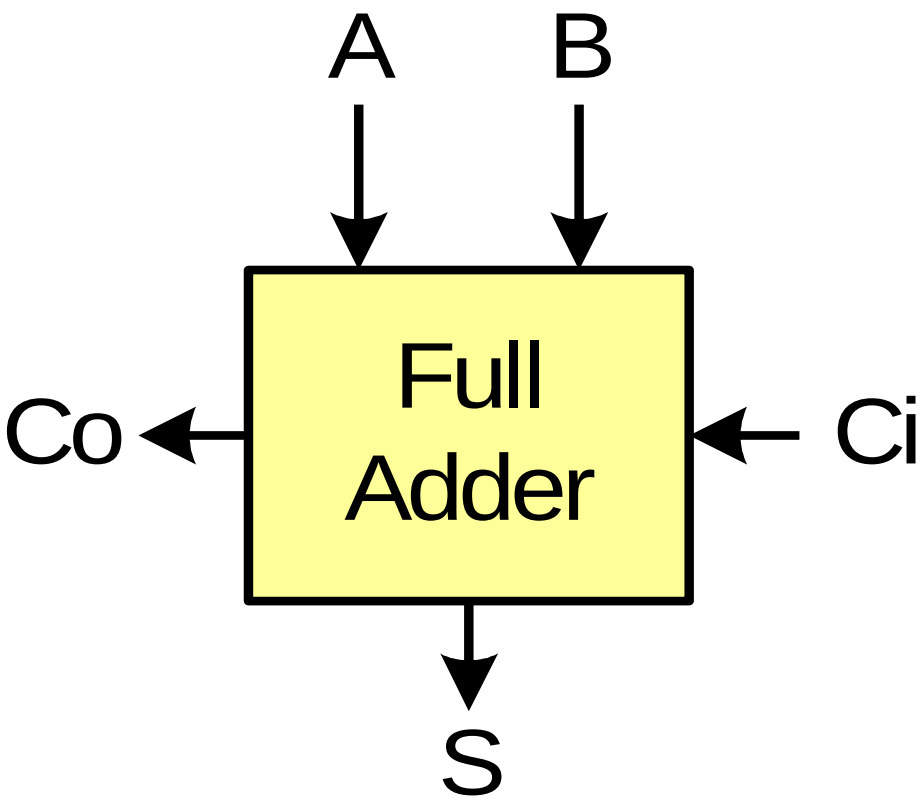
Ports Connection

```
module adder (out,in1,in2);
  output out;
  input  in1 , in2;

  assign out = in1 + in2;
endmodule
```



- Connect module ports by *name* **(Recommended!!!!!)**
  - Usage: .PortName (NetName)
  - `adder adder_0 ( .out(C) , .in1(A) , .in2(B) );`
- Connect module ports by order list
  - `adder adder_1 ( C , A , B );  // C = A + B`
- Not fully connected
  - `adder adder_2 ( .out(C) , .in1(A) , .in2() );`

Case Study: 1-bit Full Adder (1/4)

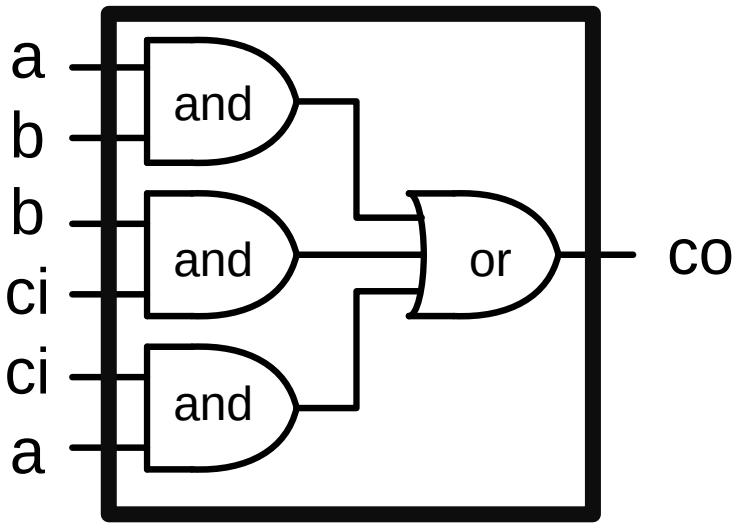


Ci	A	B	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Case Study: 1-bit Full Adder (2/4)

- co = (a • b) + (b • ci) + (ci • a);

Be careful the gate name is different from lib.v in Lab2. Your gate name should be based on lib.v !!!!



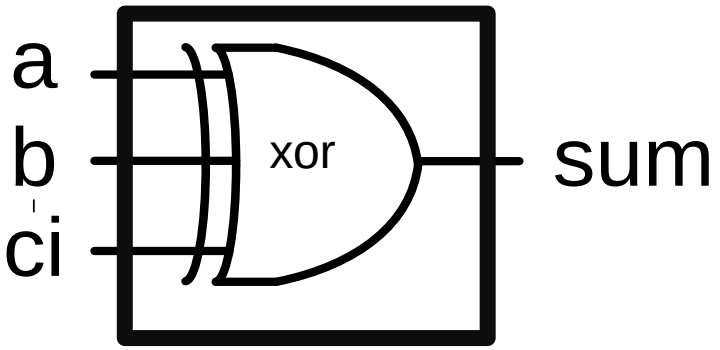

```
30
31 module FA_co ( co, a, b, ci );
32
33   input    a, b, ci;
34   output   co;
35   wire     ab, bc, ca;
36
37   and g0( ab, a, b );
38   and g1( bc, b, c );
39   and g2( ca, c, a );
40   or  g3( co, ab, bc, ca );
41
42 endmodule
43
```

instance name IO interface

Case Study: 1-bit Full Adder (3/4)

- sum = a ⊕ b ⊕ ci

Be careful the gate name is different from lib.v in Lab2. Your gate name should be based on lib.v !!!!




```
44 module FA_sum ( sum, a, b, ci );
45
46   input    a, b, ci;
47   output   sum, co;
48
49   xor g1( sum, a, b, ci );
50
51 endmodule
52
```

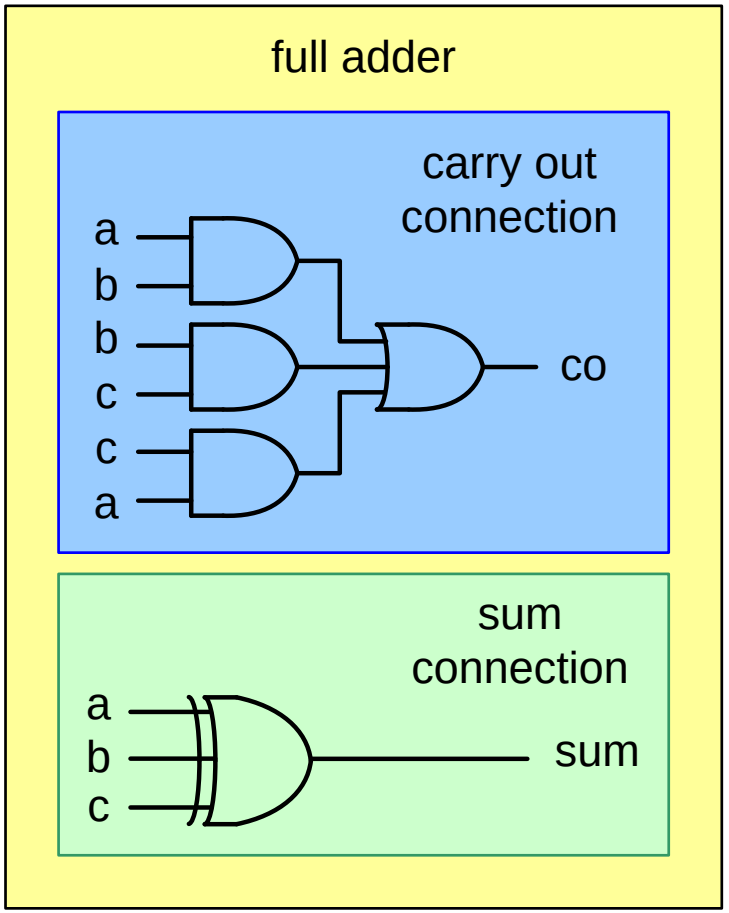
Instance name IO interface

Case Study: 1-bit Full Adder (4/4)

- Full Adder Connection
  - Instance *ins_c* from FA_co
  - Instance *ins_s* from FA_sum

```
20
21 module FA_gatelevel( sum, co, a, b, ci);
22
23   input    a, b, ci;
24   output   sum, co;
25
26   FA_co    ins_c( co, a, b, ci );
27   FA_sum   ins_s( sum, a, b, ci );
28
29 endmodule
30
```

instance name IO interface



Number Representation

- Format: <size>′<base_format><number>
  - <size> - decimal specification of number of bits
  - <base format> - ' followed by arithmetic base of number
  - <number> - value given in base of <base_format>
- Examples:
  - 6′b010_111 gives 010111
  - 8′b0110 gives 00000110
  - 4′bx01 gives xx01
  - 16′H3AB gives 0000_0011_1010_1011
  - 24 gives 0...0011000
  - 5′O34 gives 11_100
  - 16′Hx gives xxxxxxxxxxxxxxx
  - 8′hz gives zzzzzzzz

Values Assignment

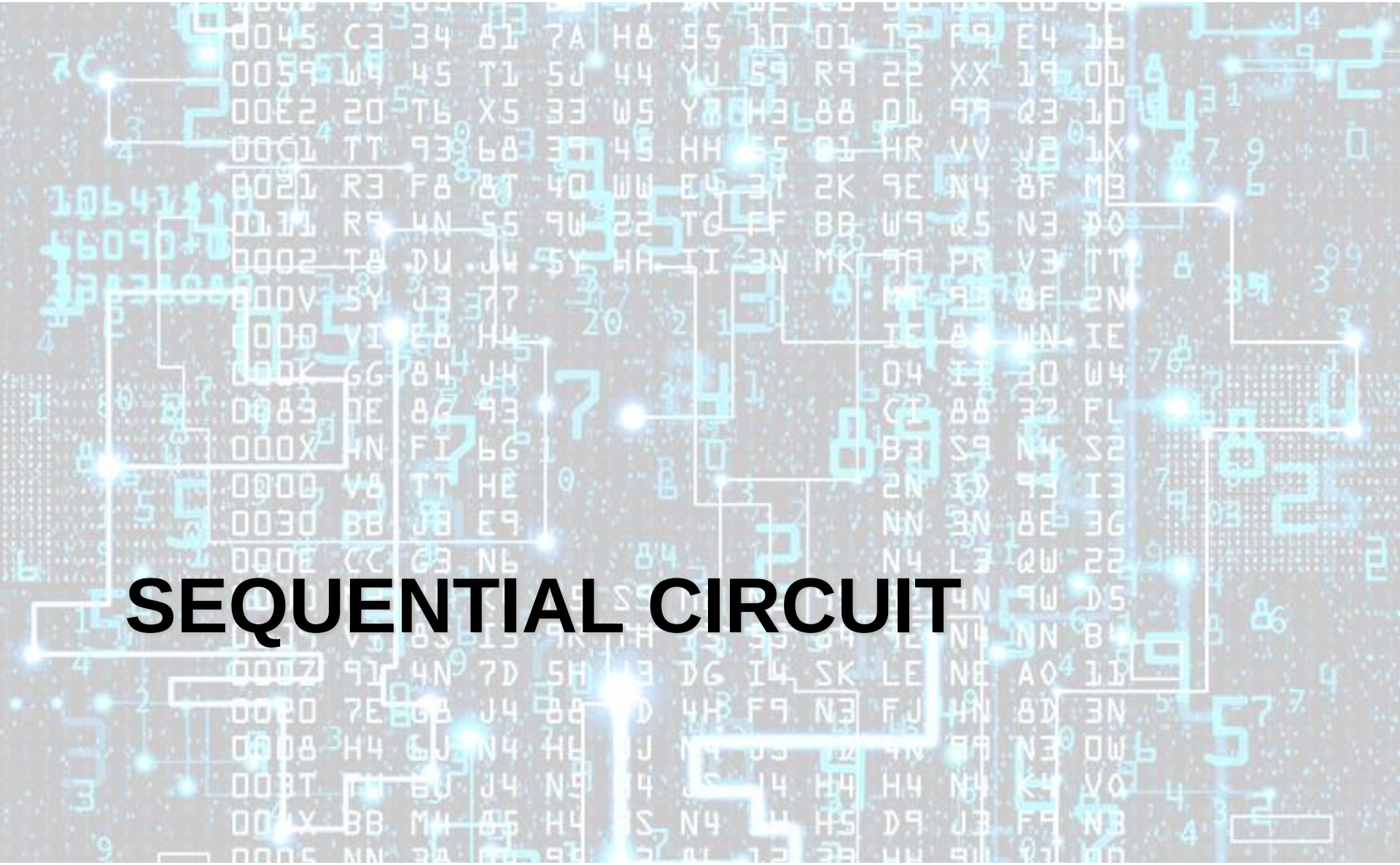
- Assignment: Drive value onto nets and registers
- There are two basic forms of assignment
  - continuous assignment, which assigns values to *wire* type
  - procedural assignment, which assigns values to *reg* type
- Basic form

Assignments	Left Hand Side	Example
Continuous Assignment	wire	wire a; assign a = 1'b1;
Procedural Assignment	reg	reg a; always@(*) a = 1'b1;

P.S. Left hand side (LHS) = Right hand side (RHS)

Operators

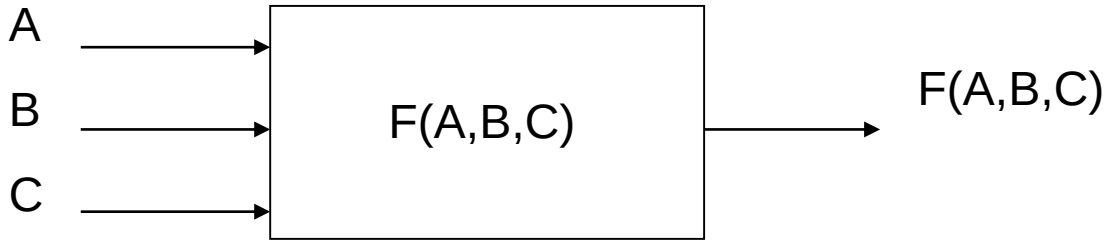
Operator	Sign	Example
Concatenation and replications	{,}	{{8{byte[7]},byte}, {a[1:0],b[0]}
Negation	**! , ~**	!2'01 //0 , ~2'b01 // 10
Bitwise	**~ , & , | , ^**	2'b01 | 2'b11 // 2'b11
Arithmetic	**+ , - , * , / , %**	3%2 //1
Shift	>> , <<	3'b011 >> 2 // 3'b000
Relational	< , <= , > , >=	
Equality	**== , !=**	
Conditional	**? :**	assign out = sel ? 1'b1 : 1'b0



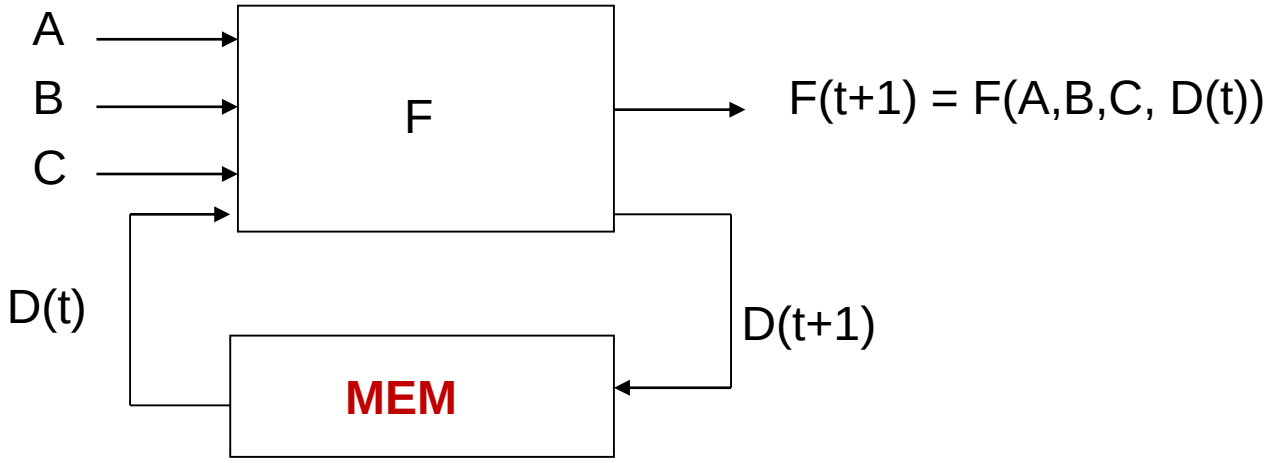
SEQUENTIAL CIRCUIT

Introduction

- Combinational Circuits (without memory)



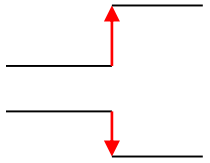
- Sequential Circuit (with memory)

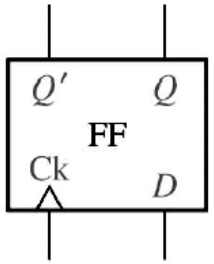


Edge-triggered D Flip-Flop

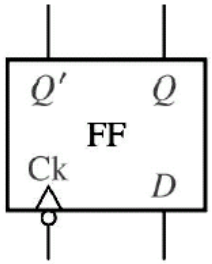
- Positive (Rising edge) trigger
- Negative (Falling edge) trigger

→to align with clock edges





(a) Rising-edge trigger



(b) Falling-edge trigger

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

$Q^+ = D$

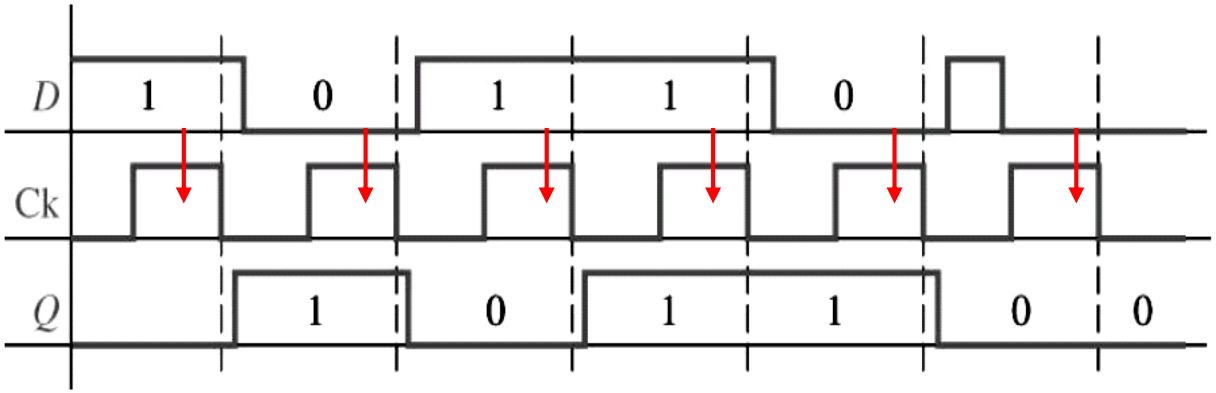
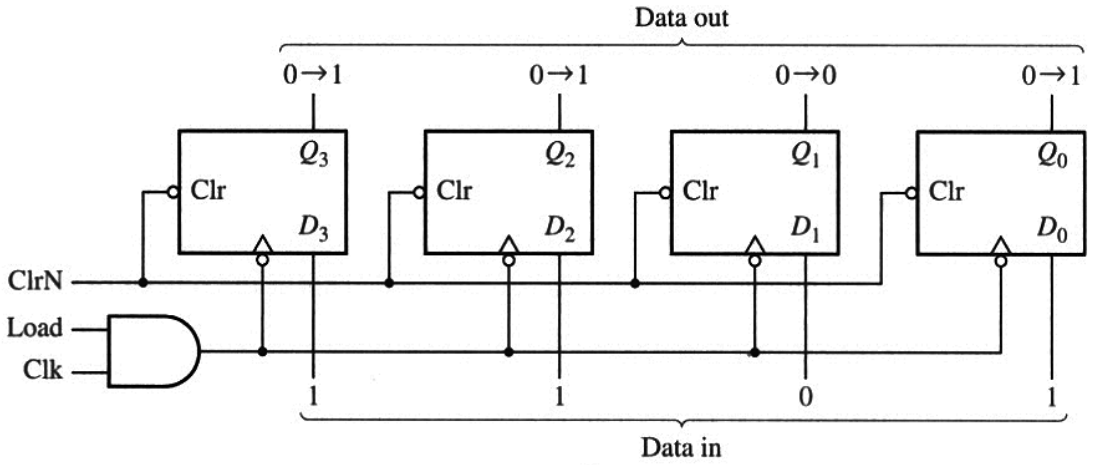
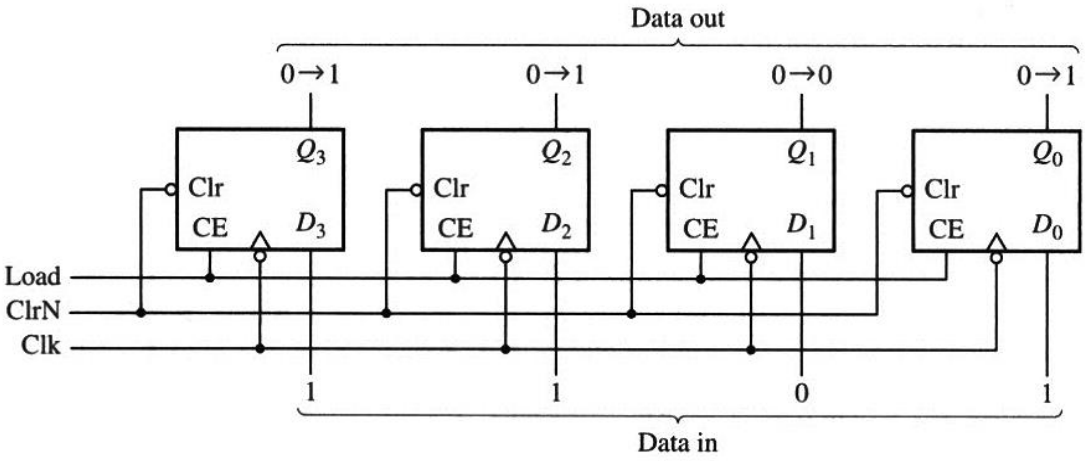


Figure 11-18 Timing for D Flip-Flop ***(Falling-Edge Trigger)***

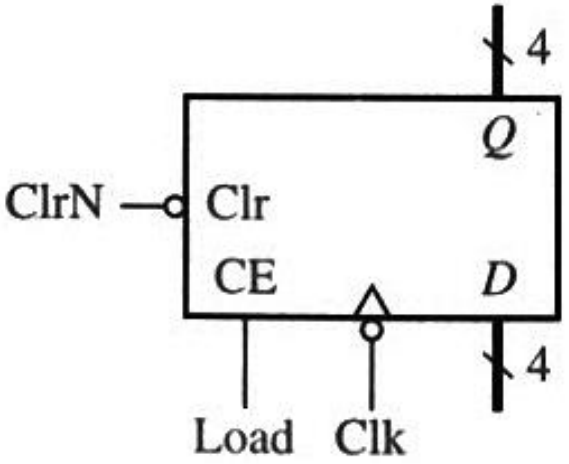
4-bit D Flip-Flop Registers



(a) Using gated clock



(b) With clock enable



(c) Symbol

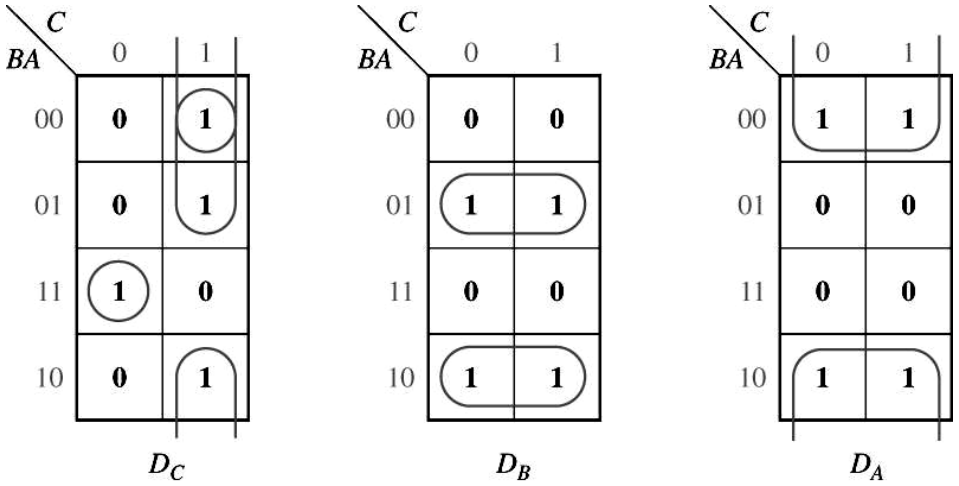
Binary Counters using D F/F (1/2)

Present State			Next State			Flip-Flop Inputs		
C	B	A	C^+	B^+	A^+	D_C	D_B	D_A
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

State Table → *As Function of (A,B,C)!*

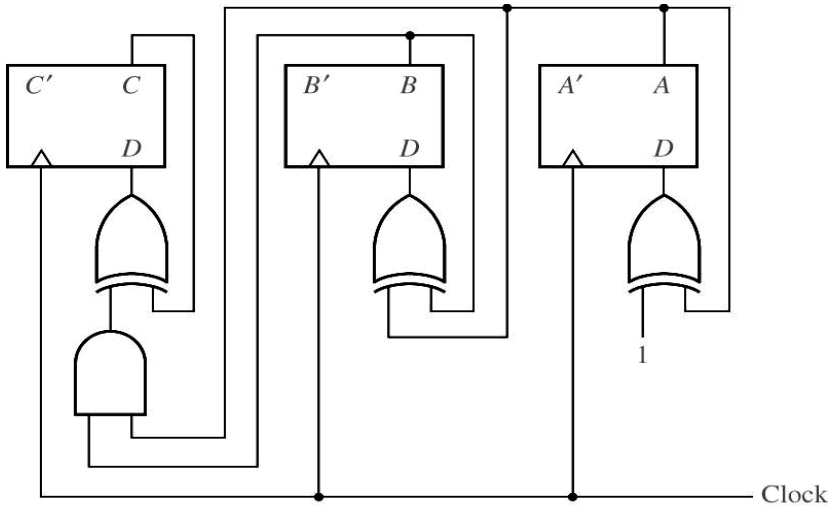
Binary Counters using D F/F (2/2)

- K-map



$D_C = AB \oplus C$ $D_B = A \oplus B$ $D_A = A'$

- Circuit Implement



SEQUENTIAL VERILOG SYNTAX

Procedural Assignment

- Procedure blocks
  - `always` block
- Only ***reg*** type can be LHS in procedure blocks
  - It's syntax. Not relevant to whether it's a flip-flop or a metal wire!
  - RHS is not restricted.
- Variable is updated by *procedural assignment*
  - For combinational circuit
    - Pure logic circuit
    - Use blocking assignment (=)
    - Use always @ (*)
  - **For sequential circuit**
    - D-FF circuit
    - Use non-blocking assignment (<=)
    - Edge trigger of clock or reset signal

```
//============= Sequential =============
always@(posedge clk or posedge rst) begin
    if(rst)
        cnt <= 0;
    else
        cnt <= nxt_cnt;
end
```

Blocking or Non-Blocking?

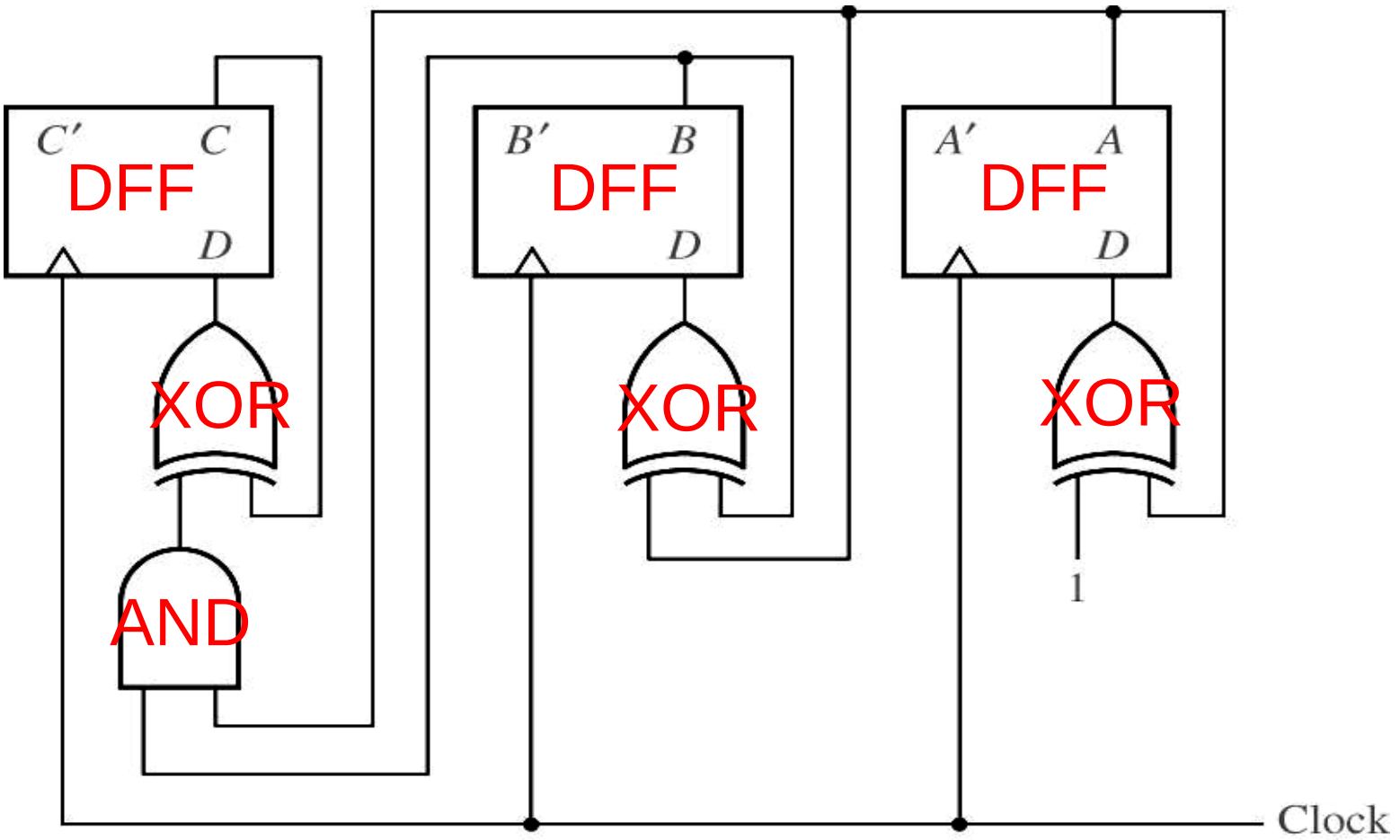
- Blocking assignment (=) Combinational !!!!
  - Evaluation and assignment are immediate

```
always @ (a or b or c)
begin
  x = a | b;          1. Evaluate a | b, assign result to x
  y = a ^ b ^ c;      2. Evaluate a^b^c, assign result to y
  z = b & ~c;         3. Evaluate b&(~c), assign result to z
end
```

- Nonblocking assignment (<=) Sequential !!!!!
  - All assignment deferred until all right-hand sides have been evaluated (end of the virtual timestamp)

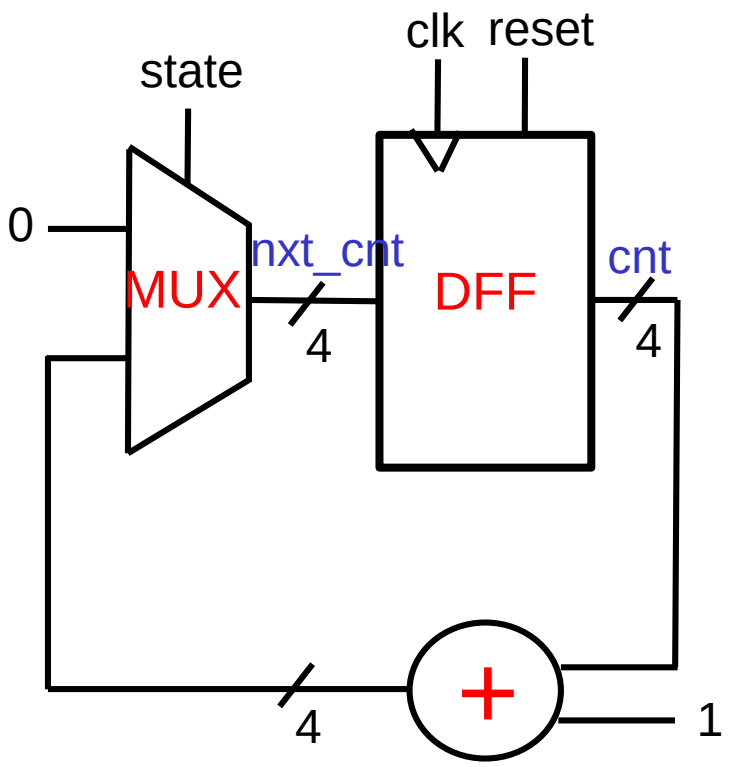
```
always @ (posedge clk)
begin
  x <= a | b;         1. Evaluate a | b but defer assignment of x
  y <= a ^ b ^ c;     2. Evaluate a^b^c but defer assignment of y
  z <= b & ~c;        3. Evaluate b&(~c) but defer assignment of z
end                   4. Assign x, y, and z with their new values
```

3-bits Counter – Gate Level



4-bits Counter – RTL Level

```
//=========  Reg/Wire Declaration ======
    reg [3:0]    cnt, nxt_cnt;

//============= Combinational ===========
    always@(*) begin
        if(state == STATE_CNT) begin
            nxt_cnt = cnt + 1;
        end
        else begin
            nxt_cnt = 0;
        end
    end
//============== Sequential =============
    always@(posedge clk or posedge rst) begin
        if(rst)
            cnt <= 0;
        else
            cnt <= nxt_cnt;
    end
```



```
/*=======================================
Author: Yu Chuan, Chuang
Module: Counter
Description:
When getting start_i signal, counter starts
to count from 0 to 15.
=======================================*/
module counter (
    input           clk,
    input           rst,
    input           start_i,
    output  [3:0]   count_o
);

//============  Parameter ===============
    localparam  STATE_IDLE  = 1'b0;
    localparam  STATE_CNT   = 1'b1;

//=========  Reg/Wire Declaration ======
    reg         state, nxt_state;
    reg [3:0]   cnt, nxt_cnt;
//========= Finite State Machine =======
    always@(posedge clk or posedge rst) begin
        if(rst)
            state <= STATE_IDLE;
        else
            state <= nxt_state;
    end

    always@(*) begin
        case(state)
            STATE_IDLE: begin
                if(start_i)
                    nxt_state = STATE_CNT;
                else
                    nxt_state = STATE_IDLE;
            end
            STATE_CNT: begin
                if(cnt == 4'd15)
                    nxt_state = STATE_IDLE;
                else
                    nxt_state = STATE_CNT;
            end
            default: nxt_state = STATE_IDLE;
        endcase
    end

//============ Combinational ===========
    assign count_o = cnt;

    always@(*) begin
        if(state == STATE_CNT) begin
            nxt_cnt = cnt + 1;
        end
        else begin
            nxt_cnt = 0;
        end
    end

//============= Sequential =============
    always@(posedge clk or posedge rst) begin
        if(rst)
            cnt <= 0;
        else
            cnt <= nxt_cnt;
    end

endmodule
```

Header/Comment
Module instantiation
Input/output declaration
Parameter
Number Representation
Reg/WIre
FSM
Procedure Block
If statement
case statement
Combinational
Continuous Assignment
Sensitivity list
Operator
Procedure Assignment
Sequential

Thanks! Feel free to ask me any questions!