



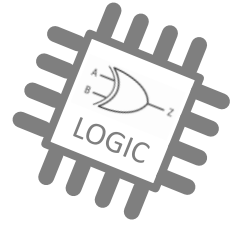
VERILOG HDL

VERILOG HDL

Hardware Description Language for Digital Circuit Design



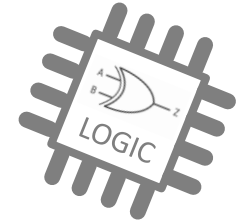
Lecturer



- BS at POSTECH
- MS at SNU
- Working for Semiconductor Company



Contents



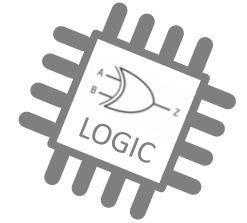
Explanation

- Introduction to Semiconductor field
- Basic concept of Verilog
- Combinational Logic
- Sequential Logic & Clock/Reset
- FSM

Coding

- Free Tools
- Declarations & Modeling
- DUT & Testbench
- Task & Function
- Lots of Practices

Contents



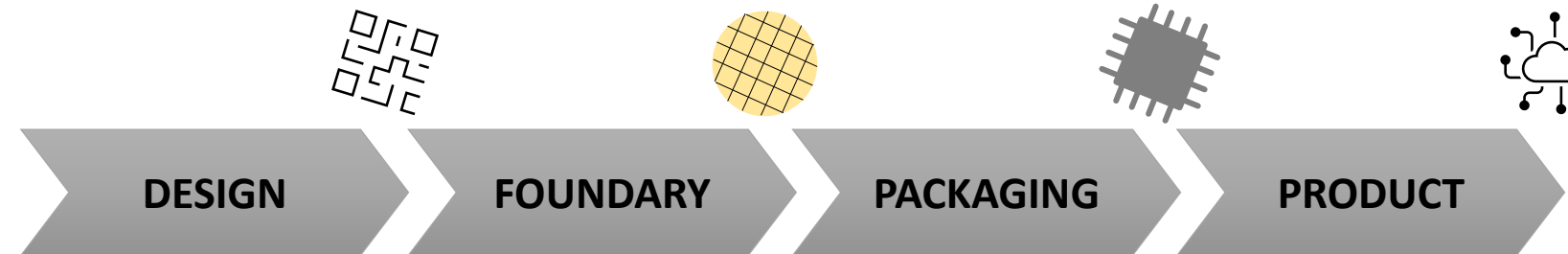
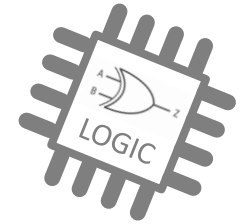
Explanation

- **Introduction to Semiconductor field**
- Basic concept of Verilog
- Combinational Logic
- Sequential Logic & Clock/Reset
- FSM

Coding

- Free Tools
- Declarations & Modeling
- DUT & Testbench
- Task & Function
- Lots of Practices

Semiconductor Ecosystem



IDM



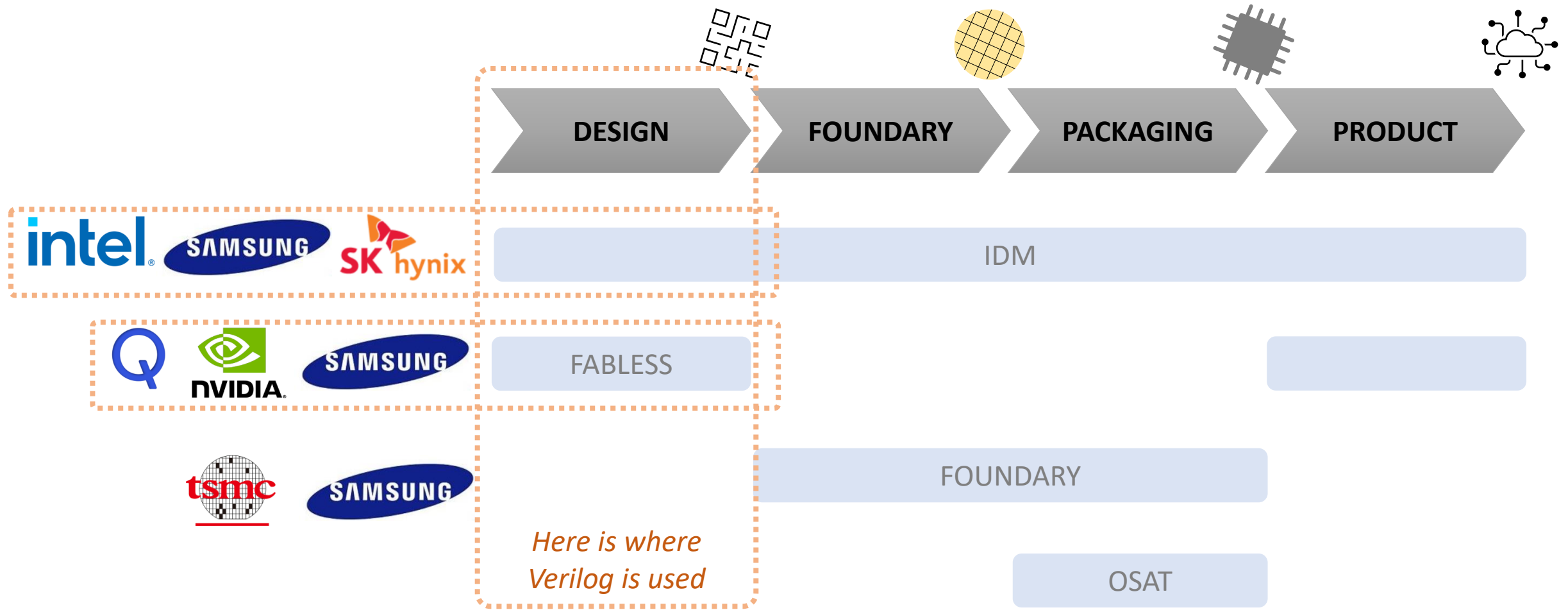
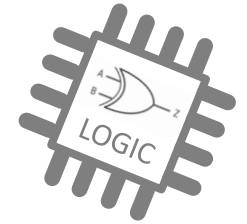
FABLESS



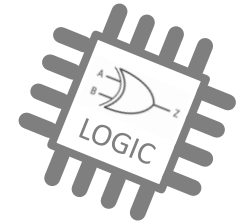
FOUNDRY

OSAT

Semiconductor Ecosystem



Semiconductor Product



MEMORY



TOSHIBA



KIOXIA

Verilog as Minor Design

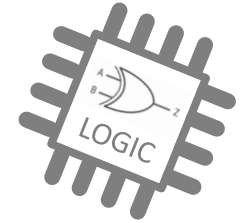
SYSTEM



*Others,
Lots of Companies*

Verilog as Major Design

Contents



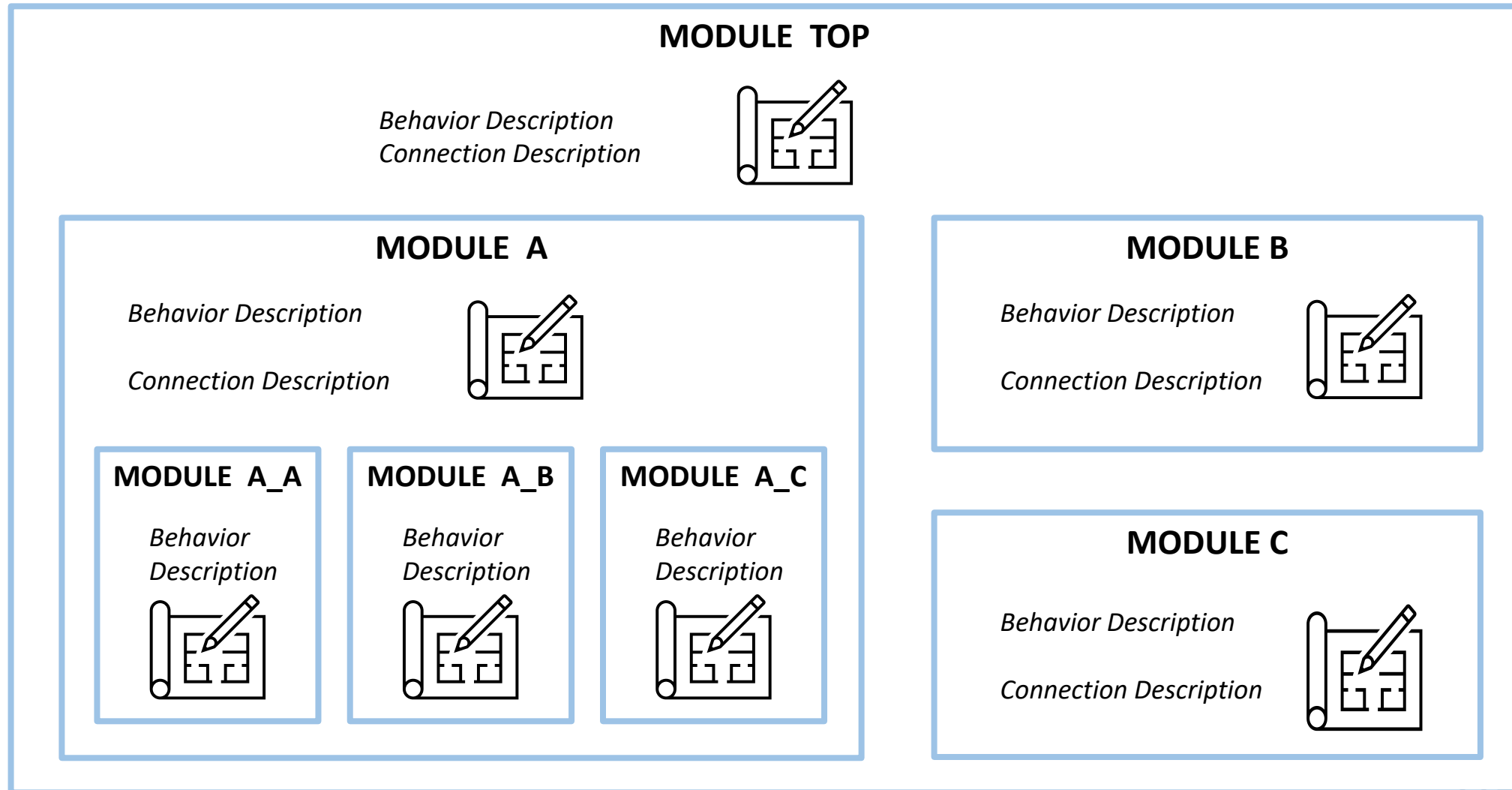
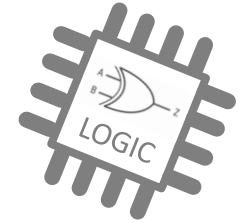
Explanation

- Introduction to Semiconductor field
- **Basic concept of Verilog**
- Combinational Logic
- Sequential Logic & Clock/Reset
- FSM

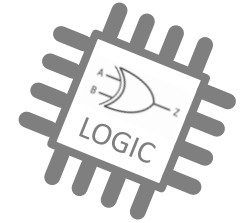
Coding

- Free Tools
- Declarations & Modeling
- DUT & Testbench
- Task & Function
- Lots of Practices

Hierarchical Language

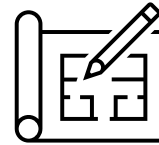


Hierarchical Language



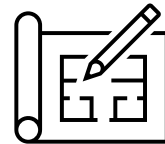
COMPUTER

Behavior Description
Connection Description



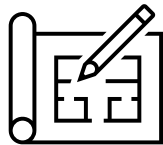
MOTHERBOARD

Behavior Description
Connection Description



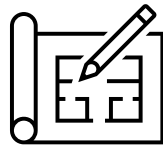
CPU

Behavior Description



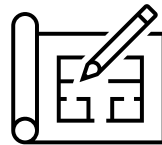
GPU

Behavior Description



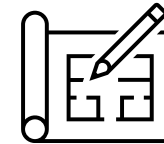
DDR

Behavior Description



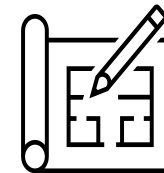
SSD

Behavior Description
Connection Description

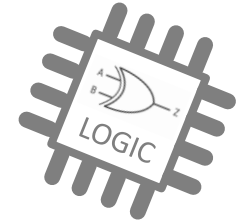


POWER SUPPLY

Behavior Description
Connection Description

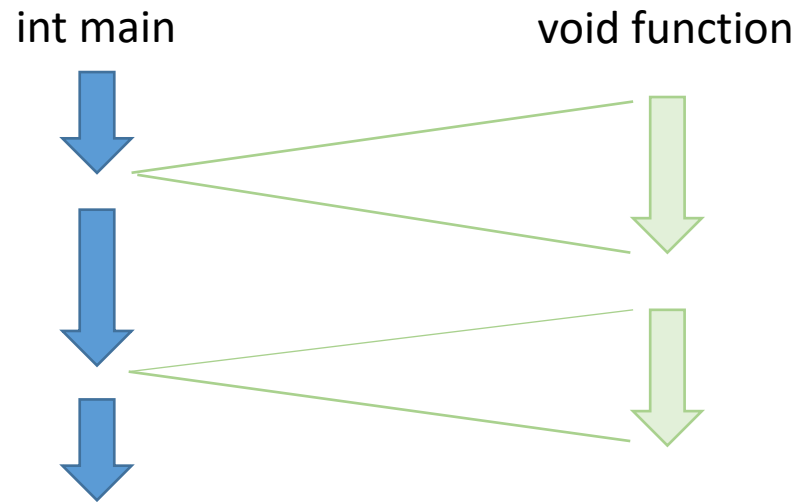


Comparison with C function



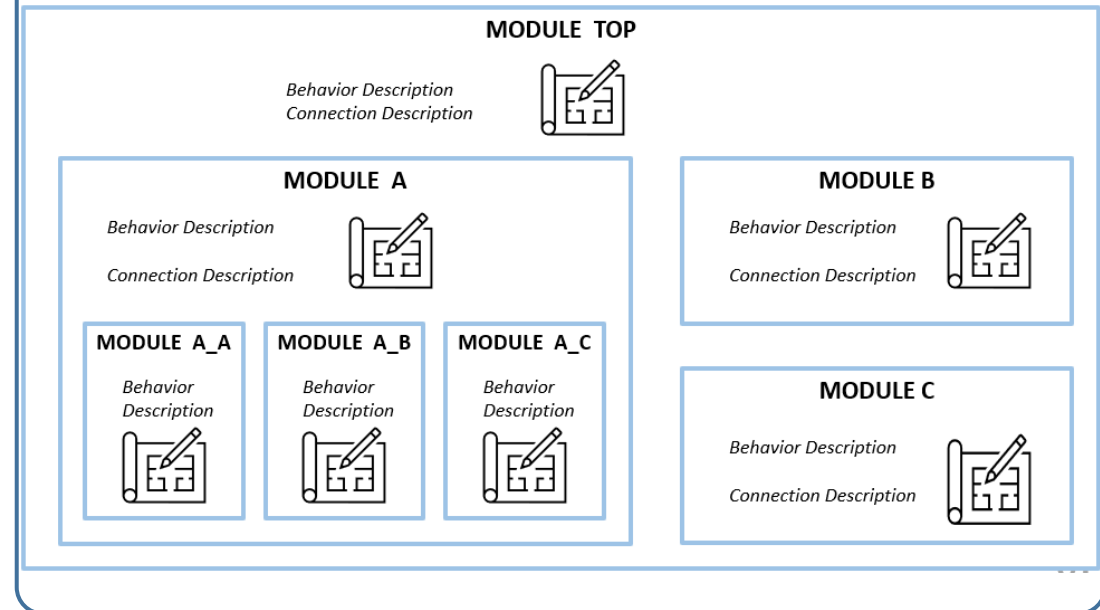
C function

- Go to function and return
- This concept is also in testbench (not design)

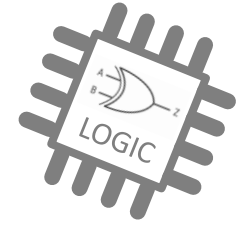


Verilog Module

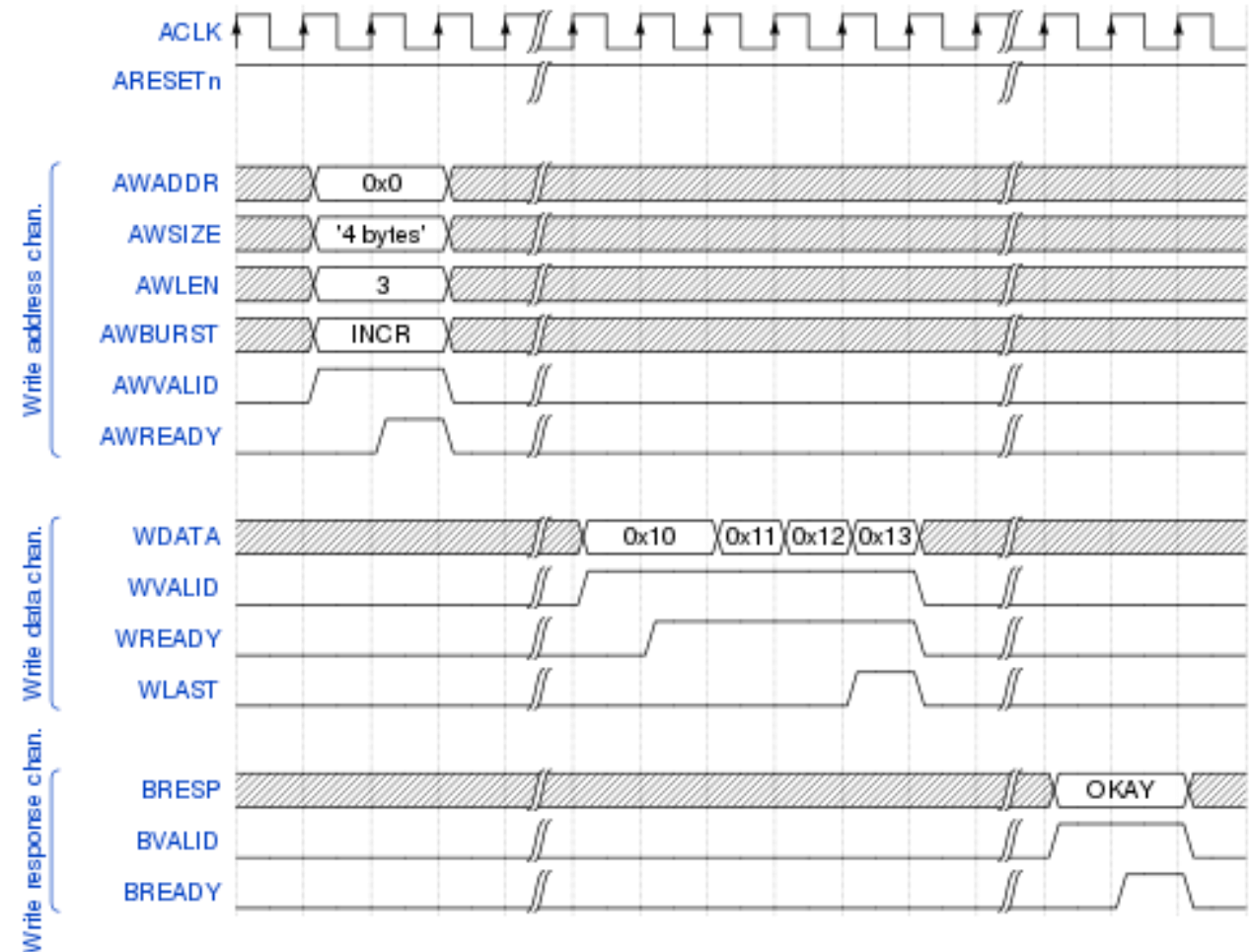
- Hierarchical in Circuit
- No concept of return



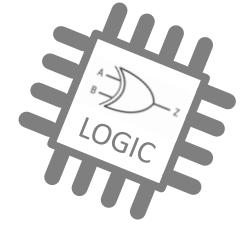
Cycle-based results



- Clock, Cycle
 - Most of Verilog design should have clock
 - At each time, depending on input, expected behavior is visible
- Parallel behavior
 - Multiple lines will be affected at the same time
cf) only 1 line should be executed in C



What is Difference ?



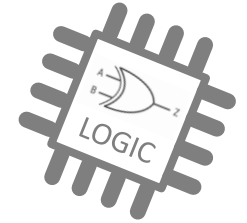
C/C++

- Language for SW
- Each line should be performed sequentially
Ex) Line 1 is followed by Line 2
- Describe which behavior should be processed
The result is action performed
Ex) "z = a + b" means "do the plus command"

Verilog/VHDL

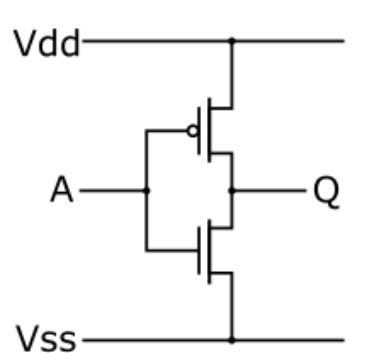
- Language for HW
- Each line should be processed at every cycle
Ex) Line 1 and 2 at same time
- Describe how the logic is implemented
The result is circuit not action
Ex) "z = a + b" means "plus logic is implemented"

What will you design ?

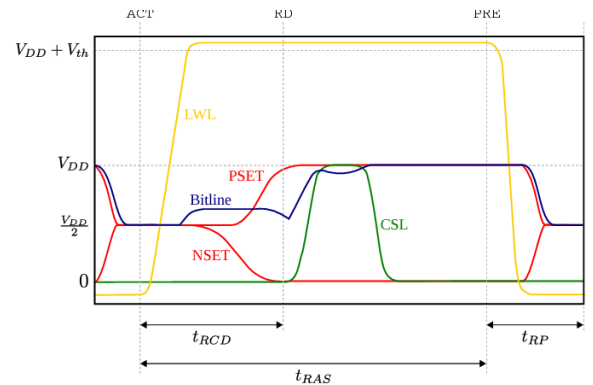


ANALOG

- Optimize for Speed, Integrity with very compact design
- Described by Schematic
- Detail view for signal
- **Appropriate for Memory !!**



by wikipedia

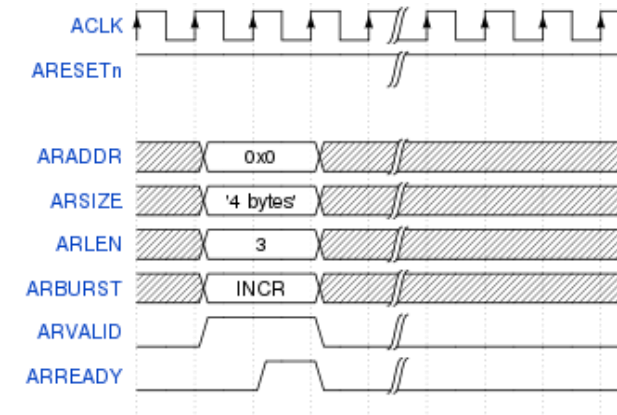


by semantic scholar

DIGITAL

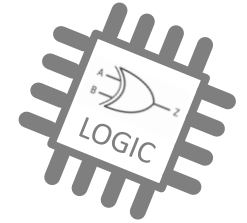
- Optimize for Complicated behavior with very large design
- Described by Language (Verilog)
- Only 0, 1 for signal
- **Appropriate for System !!**

assign out = ~in;



by wikipedia

Contents



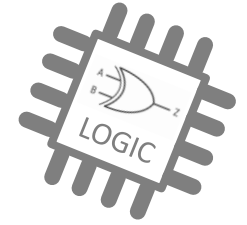
Explanation

- Introduction to Semiconductor field
- Basic concept of Verilog
- Combinational Logic
- Sequential Logic & Clock/Reset
- FSM

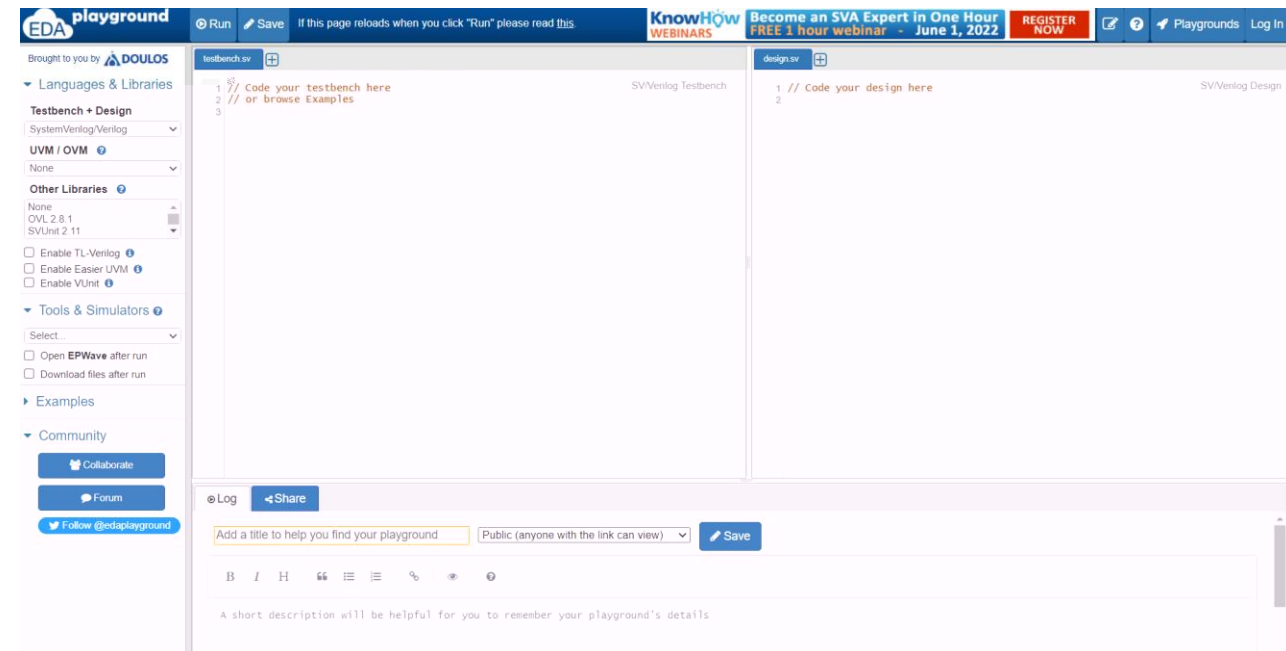
Coding

- **Free Tools**
- Declarations & Modeling
- DUT & Testbench
- Task & Function
- Lots of Practices

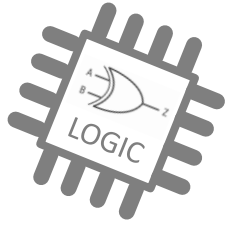
What do you need ?



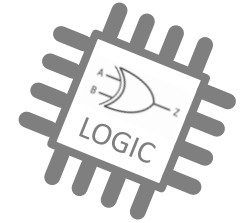
- Simulator and Debugger are essential
- Synopsys/Cadence/Siemens EDA (Mentor)
 - EDA tools are too expensive
- **Free tools !!**
 - Icarus Verilog
 - <https://www.edaplayground.com>



Practice



Contents



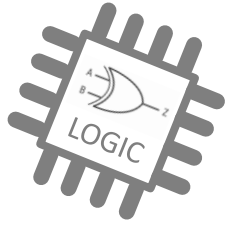
Explanation

- Introduction to Semiconductor field
- Basic concept of Verilog
- **Combinational Logic**
- Sequential Logic & Clock/Reset
- FSM

Coding

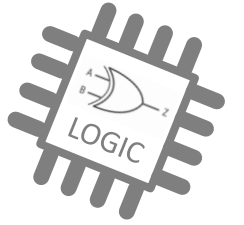
- Free Tools
- Declarations & Modeling
- DUT & Testbench
- Task & Function
- Lots of Practices

Agenda

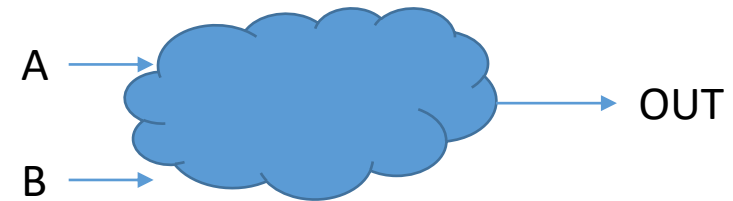


- Overview
- Gate Logic (AND, OR ...)
- Boolean Algebra
- Karnaugh Map
- Combinational Logic (Decoder, MUX, ...)
- Additional (Tri, buf, ...)

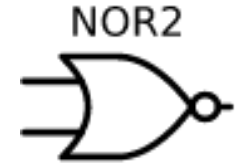
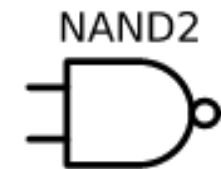
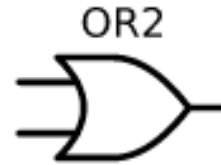
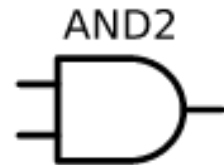
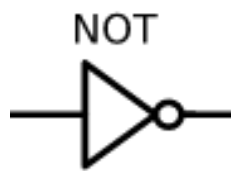
Overview



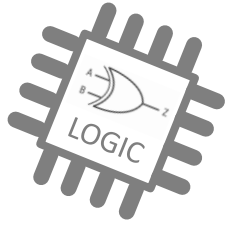
- What is Combinational Logic ?
 - The logic which output is defined by only the combination of gates and input values
 - time-independent logic



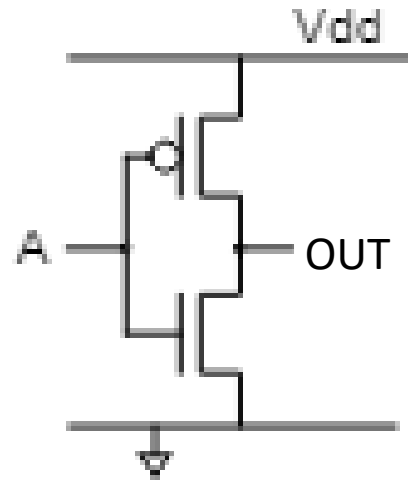
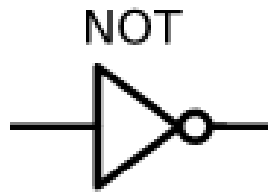
- These are basic gates which used in general



Gate Logic



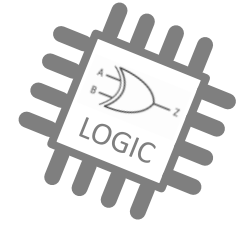
- NOT gate



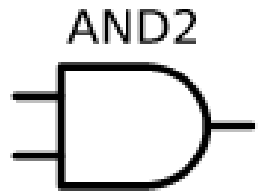
assign OUT = ~A;
assign OUT = !A;

A	OUT
0	1
1	0

Gate Logic

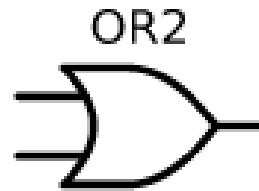


- AND/OR gate



assign OUT = A & B;

A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1



assign OUT = A | B;

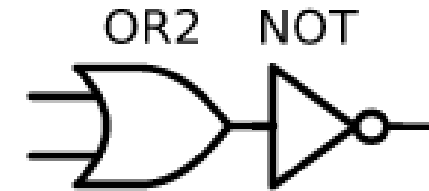
A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

- NAND/NOR gate



assign OUT = ~(A & B);

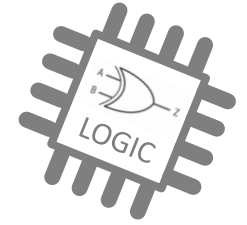
A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0



assign OUT = ~(A | B);

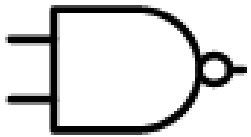
A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

Gate Logic



- NAND/NOR gate

NAND2

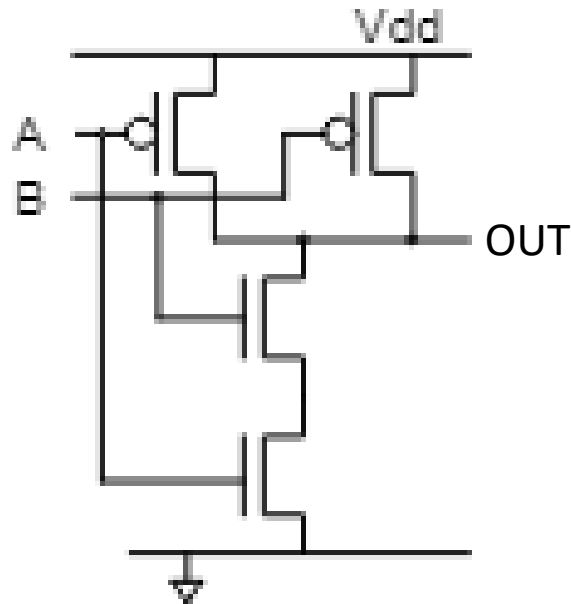


AND2 NOT



assign OUT = ~(A & B);

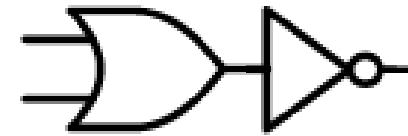
A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0



NOR2

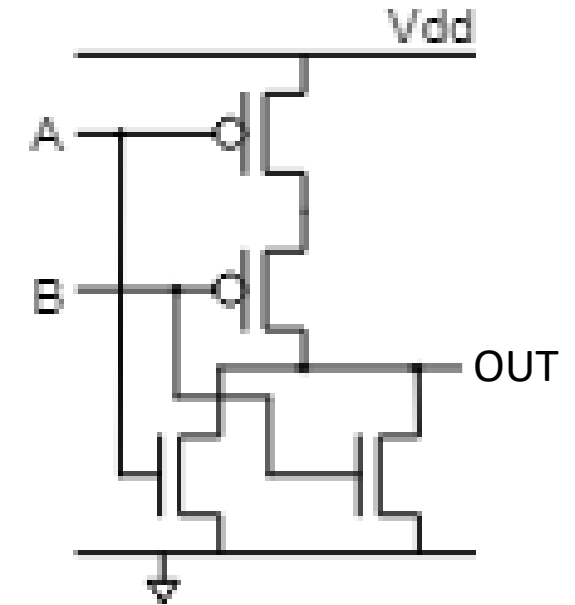


OR2 NOT

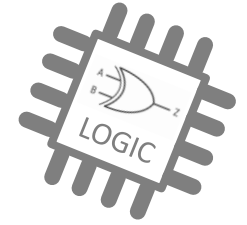


assign OUT = ~(A | B);

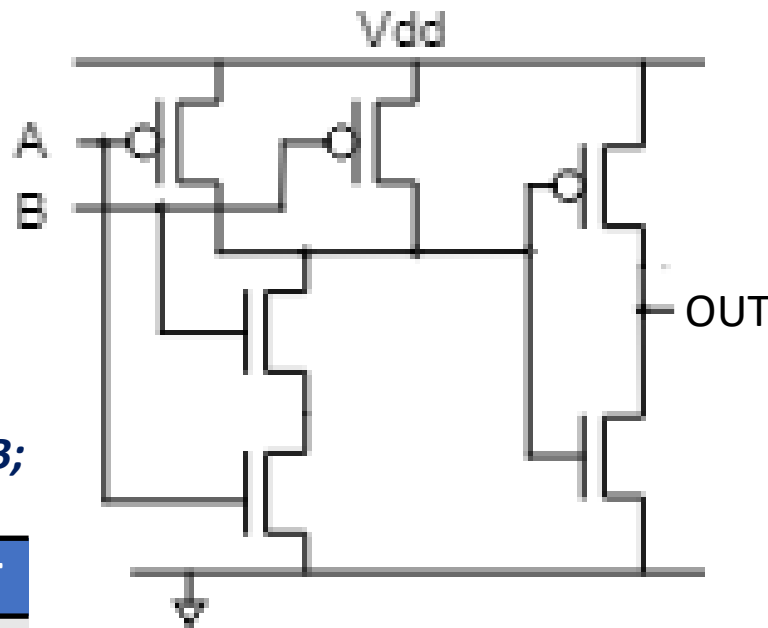
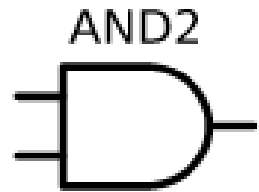
A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0



Gate Logic

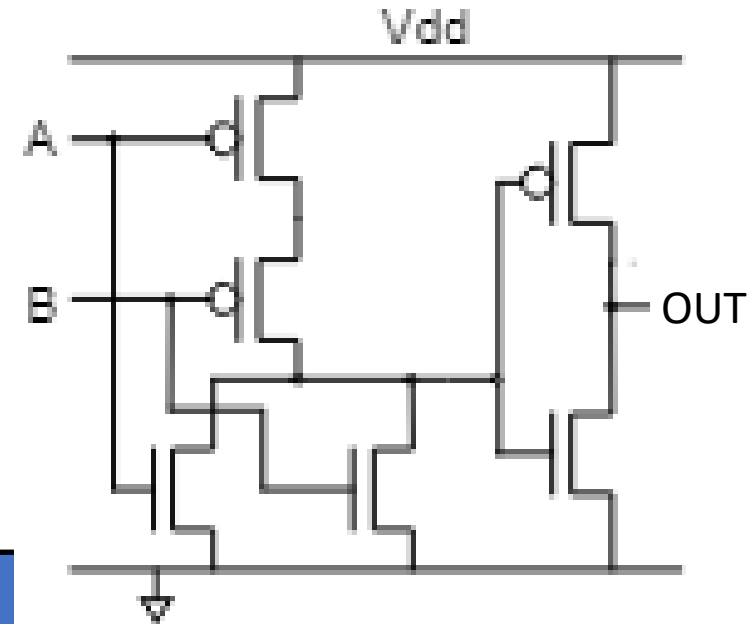
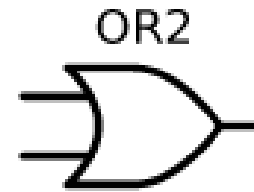


- AND/OR gate



assign OUT = A & B;

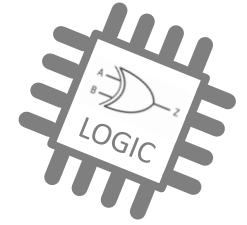
A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1



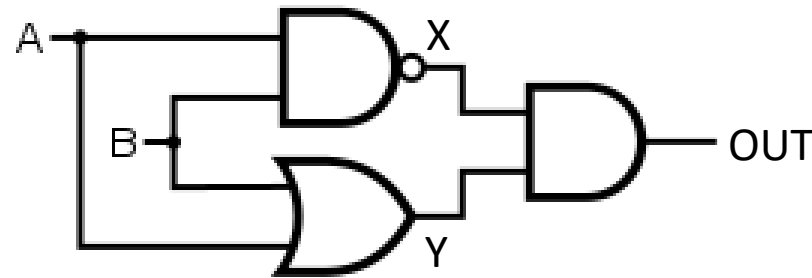
assign OUT = A | B;

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

Gate Logic



- XOR gate

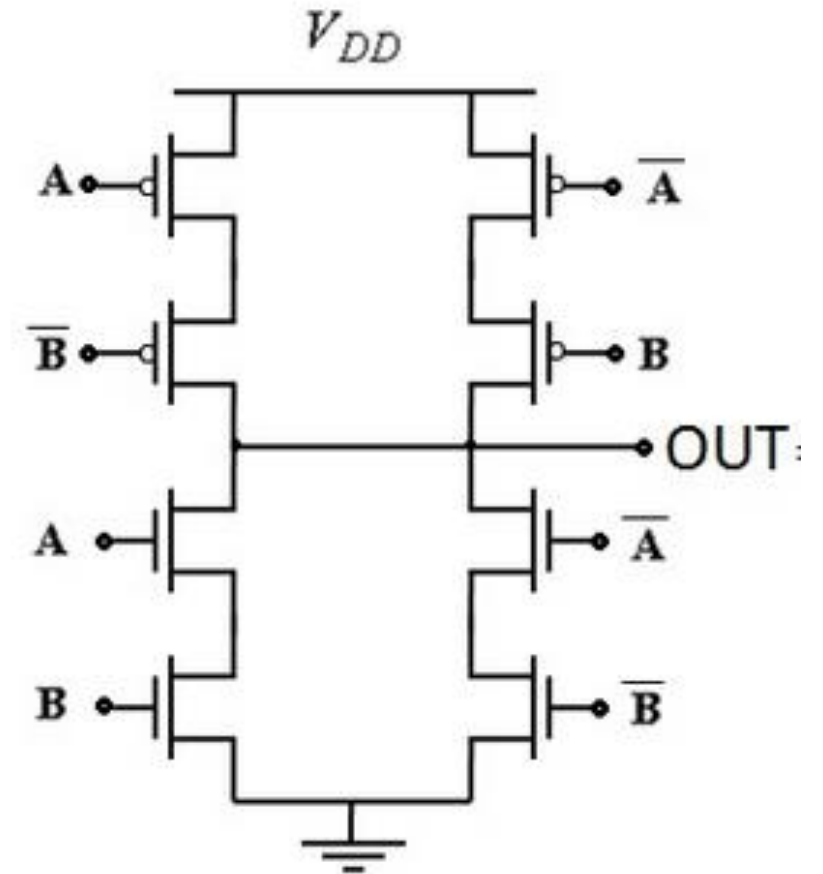


assign OUT = A ^ B;

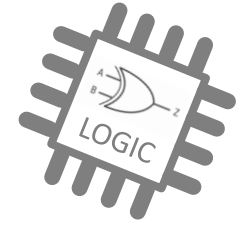
A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

assign OUT = ~(A & B) & (A | B);

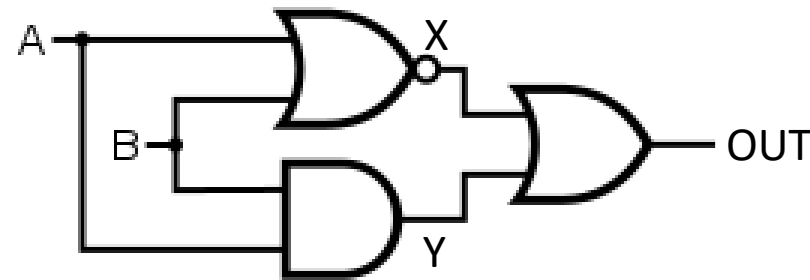
A	B	X	Y	OUT
0	0	1	0	0
0	1	1	1	1
1	0	1	1	1
1	1	0	1	0



Gate Logic



- XNOR gate

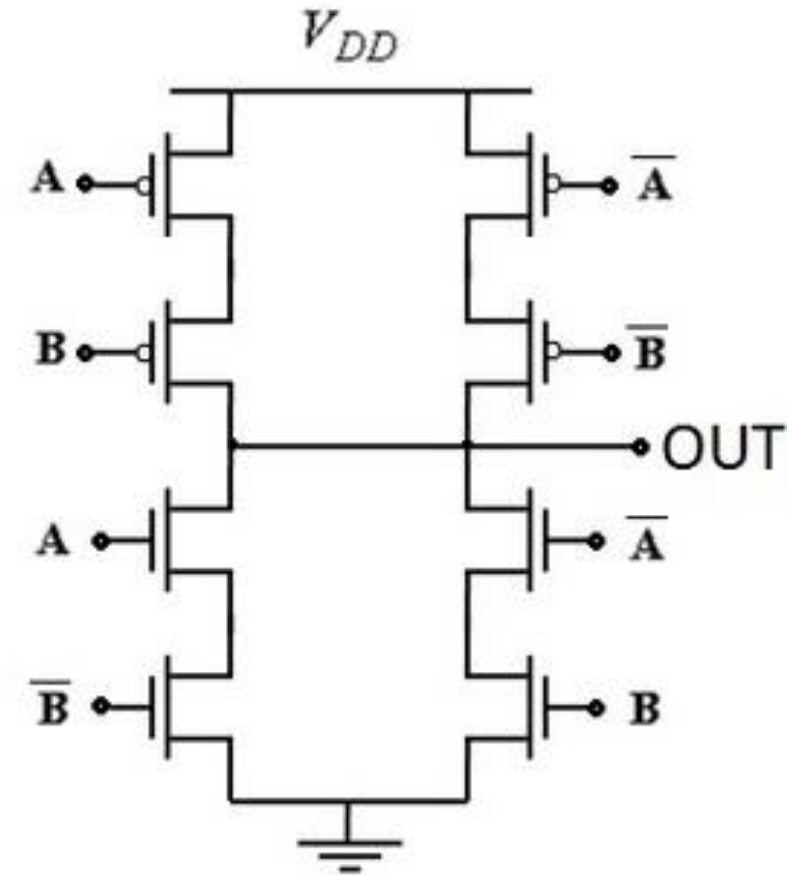


assign OUT = ~(A ^ B);

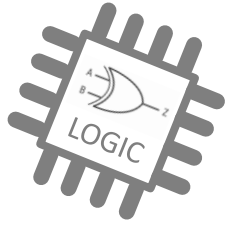
A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	1

assign OUT = ~(A | B) | (A & B);

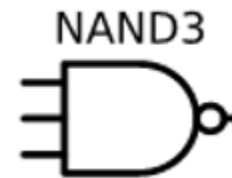
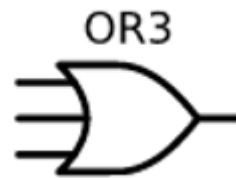
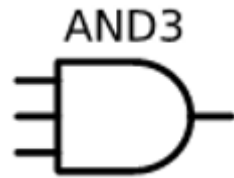
A	B	X	Y	OUT
0	0	1	0	1
0	1	0	0	0
1	0	0	0	0
1	1	0	1	1



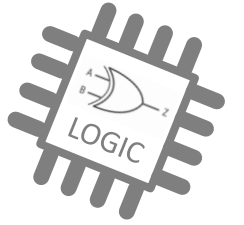
Gate Logic



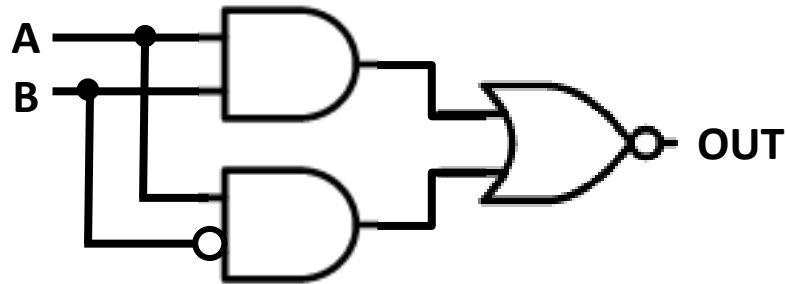
- 3-input gates have similar behavior
- Same for 4-input, 5-input, N-input



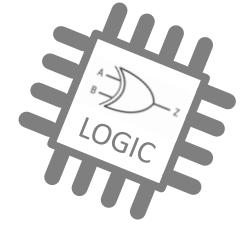
Boolean Algebra






- Why ?
 - Minimize the logic expression for same behavior
 - *Ex) Both Logics are exactly same*



Boolean Algebra



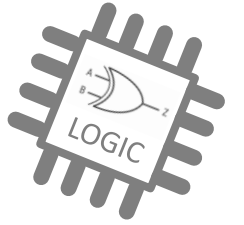
- Basic expression

- AND  : $A \cdot B, AB$
- OR  : $A + B$
- NOT  : \bar{A}

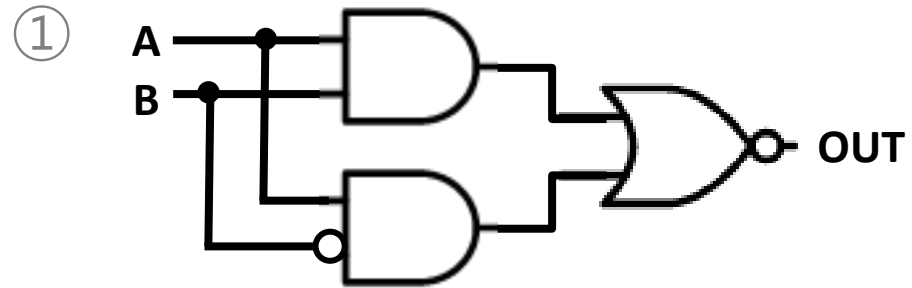
- Basic Law

- commutative law : $A + B = B + A, AB = BA$
- associative law : $(A + B) + C = A + (B + C), (AB)C = A(BC)$
- distributive law : $A(B + C) = AB + AC, A + BC = (A + B)(A + C)$
- redundancy law : $A = A \cdot 1 = A + 0, 1 = A + \bar{A} = 1 + A, 0 = A\bar{A} = A \cdot 0$
- identity law : $A = A + A = AA = A\bar{B} + AB = (A + B)(A + \bar{B})$
- De Morgan's Theorem : $\overline{A + B} = \bar{A}\bar{B}, \overline{AB} = \bar{A} + \bar{B}$

Boolean Algebra

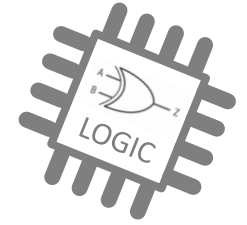


- Practice



② $AB + ABC + A\bar{B} + \bar{A}BC$

Karnaugh Map



- Why Karnaugh Map ?
 - More systematic and visualized method
 - Very easy way for up to 4 input variables

A \ B	0	1
0		
1		

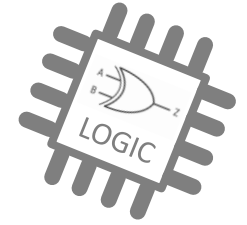
AB \ C	0	1
00		
01		
11		
10		

AB \ CD	00	01	11	10
00				
01				
11				
10				

- Need to know
 - Truth table
 - Equation
 - SOP : $F(A,B,C) = \sum m(0,3,4,7)$
 - POS : $F'(A,B,C) = \sum m(1,5)$
 $F(A,B,C) = \prod m(1,5)$
 - Don't care : $\sum d(2,6)$, $\prod d(2,6)$
 - Gray Code
 - change a bit at a time
 $00 \rightarrow 01 \rightarrow 11 \rightarrow 10$
 $100 \leftarrow 101 \leftarrow 111 \leftarrow 110$

A	B	C	OUT
0	0	0	1
0	0	1	0
0	1	0	X
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	X
1	1	1	1

Karnaugh Map



- How to use
 1. Mark each value from equations
 2. Make a rectangle with same value as big as possible (1 for SOP, 0 for POS)
(Don't care can be included to anyone)
(Bigger rectangle means less variables)
 3. Make simple expression

$$\text{Ex) } F(A,B,C,D) = \sum m(0,4,6,8,12,14) + \sum d(1,5,9,13) \rightarrow C' + BD'$$

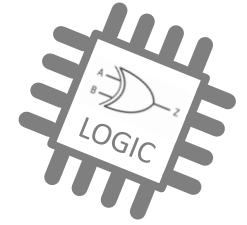
AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

AB \ CD	00	01	11	10
00	1	X	0	0
01	1	X	0	1
11	1	X	0	1
10	1	X	0	0

AB \ CD	00	01	11	10
00	1	X	0	0
01	1	X	0	1
11	1	X	0	1
10	1	X	0	0

AB \ CD	00	01	11	10
00	1	X	0	0
01	1	X	0	1
11	1	X	0	1
10	1	X	0	0

Karnaugh Map



- Practice

① $F(A,B,C,D) = \sum m(0,1,4) + \sum d(5,6,7)$

② $F(A,B,C,D) = \sum m(0,1,2,3,6,7,10,11,14) + \sum d(12,13,15)$

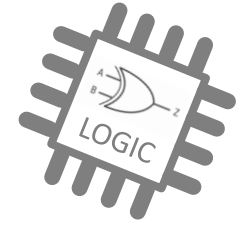
AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

AB \ CD	00	01	11	10
00				
01				
11				
10				

AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

AB \ CD	00	01	11	10
00				
01				
11				
10				

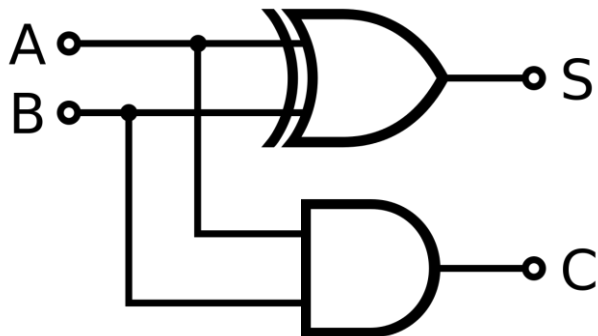
Combinational Logic



- Half-Adder

$$A+B = \text{SUM} + \text{CARRY}$$

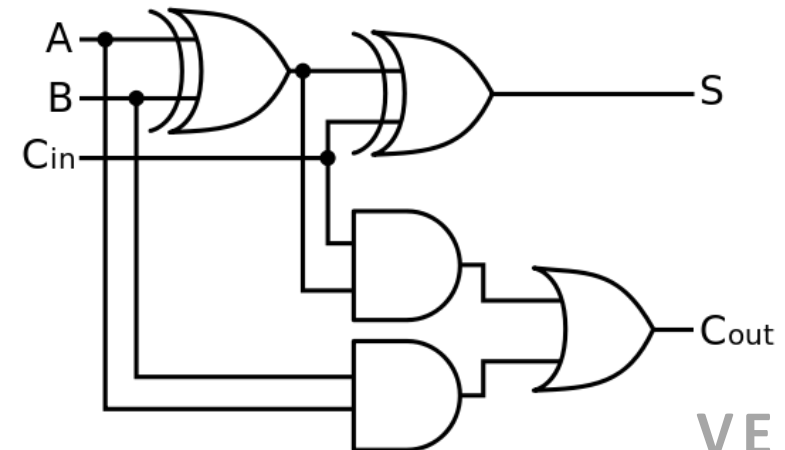
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



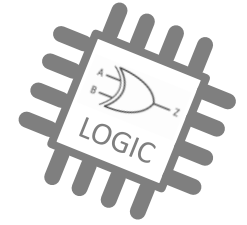
- Full-Adder

$$A+B+C_{in} = \text{SUM} + \text{CARRY}$$

Carry in makes another result

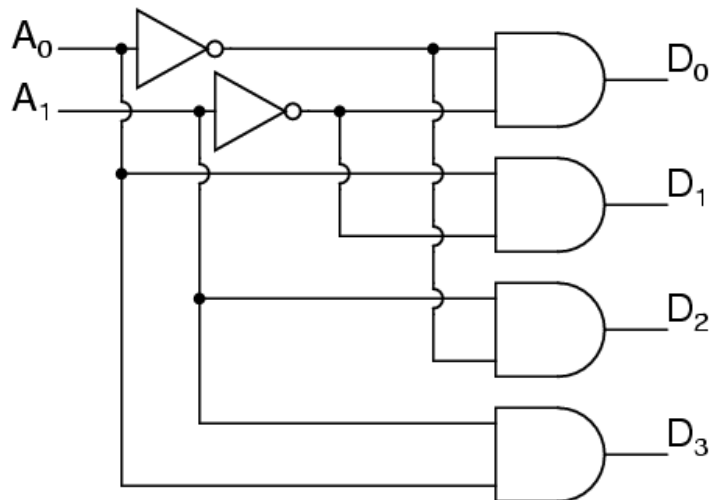


Combinational Logic



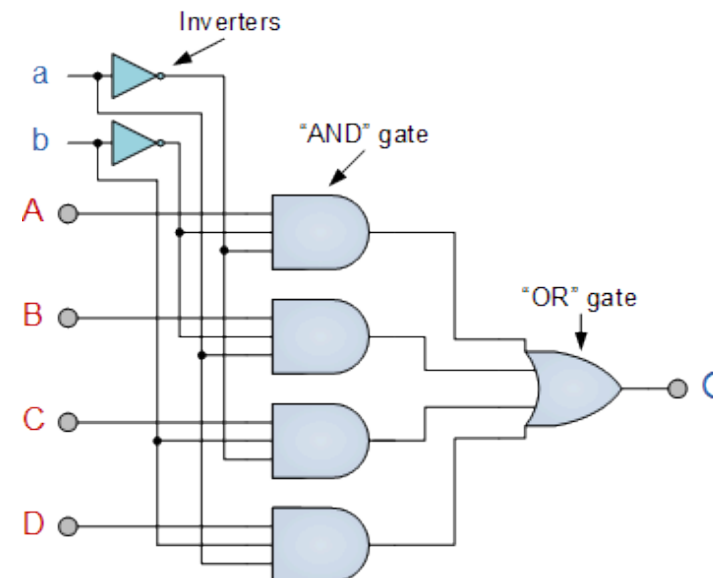
- Decoder
 - Only one output is chosen depending on input values

A1	A0	D3	D2	D1	D0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



- Mux
 - Only one value is chosen depending on the decoder

b	a	D3	D2	D1	D0	Q
0	0	0	0	0	1	A
0	1	0	0	1	0	B
1	0	0	1	0	0	C
1	1	1	0	0	0	D

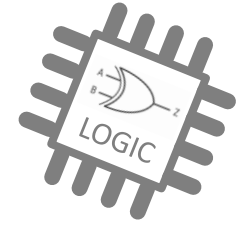


4:1 MUX



VERILOG
AEKIOG

Combinational Logic



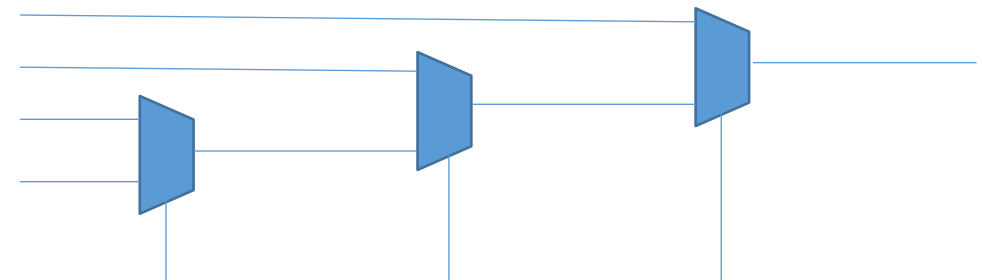
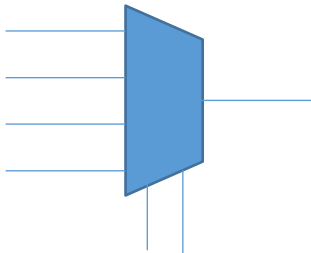
- Conditional Expression
 - case (switch-case in C)

```
case (a)
  0:    q=a;
  1:    q=b;
  2:    q=c;
  default: q=d;
```

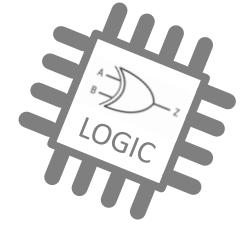
- if / else

```
if      (a==0) q=a;
else if (a==1) q=b;
else if (a==2) q=c;
else      q=d;
```

4:1 MUX

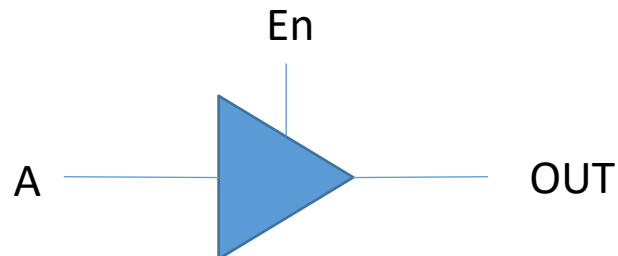


Additional Logic

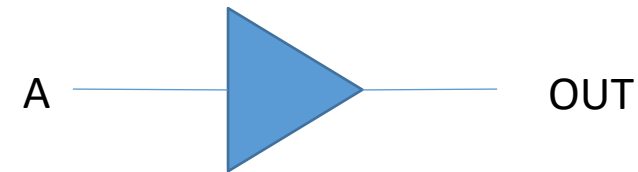


- Tri-state buffer
 - Z value (high impedance)
 - It's like a switch
 - Used for Bi-directional port

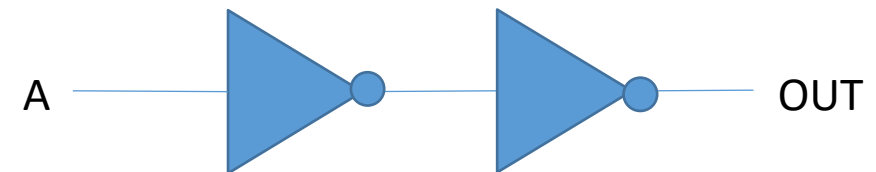
En	A	OUT
0	0	Z
0	1	Z
1	0	0
1	1	1



- Buffer
 - Nothing changes

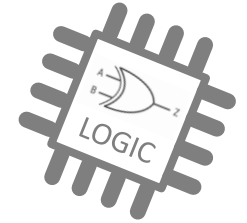


- It's like concatenation of 2 inverters



- Necessary when signal keeps its value for long distance

Contents



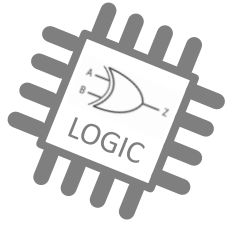
Explanation

- Introduction to Semiconductor field
- Basic concept of Verilog
- Combinational Logic
- **Sequential Logic & Clock/Reset**
- FSM

Coding

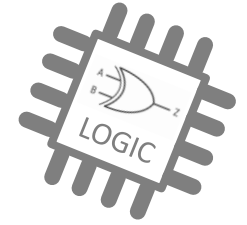
- Free Tools
- Declarations & Modeling
- DUT & Testbench
- Task & Function
- Lots of Practices

Agenda

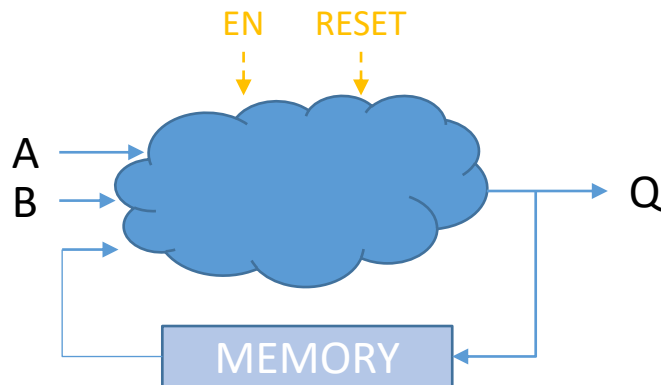


- Overview
- SR Latch, Gated SR Latch
- D Flip-Flop
- Clock & Reset
- Sequential Logic (Shift Register, Counter)

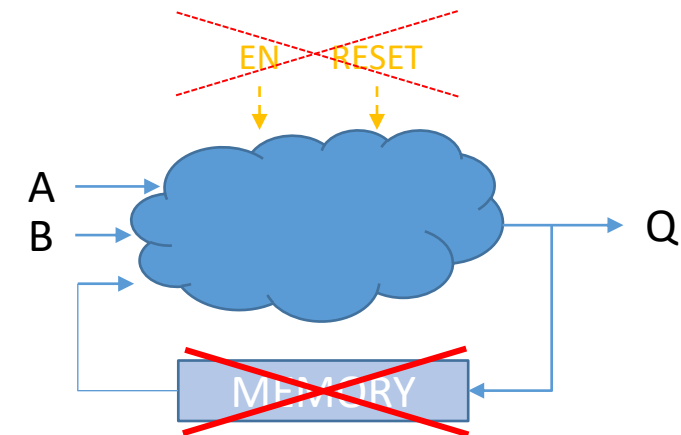
Overview



- What is Sequential Logic ?
 - The logic which output is affected by not only input values but also output states
 - time-dependent logic
 $Q(\text{next/curr})$ is affected by $Q(\text{curr/prev})$
 - EN (Clock) and Reset are usually required

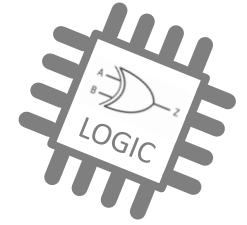


IN	Q	Q(next)
0	0	0
0	1	1
1	0	1
1	1	0

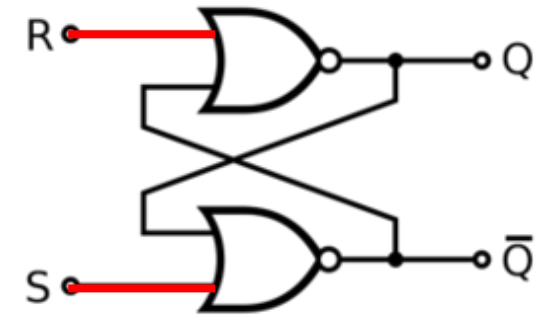
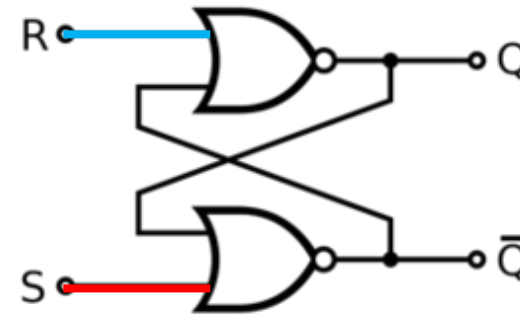
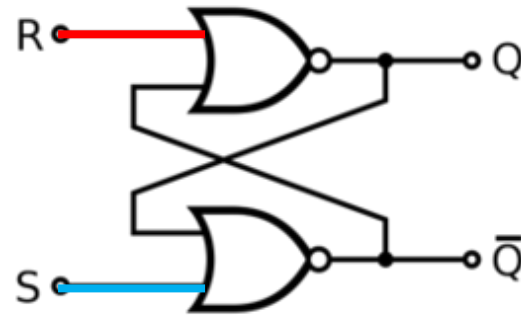
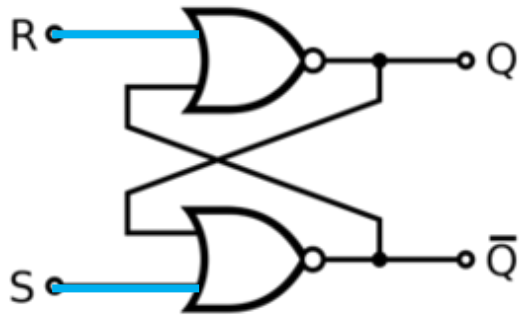


cf) Combinational Logic

SR Latch

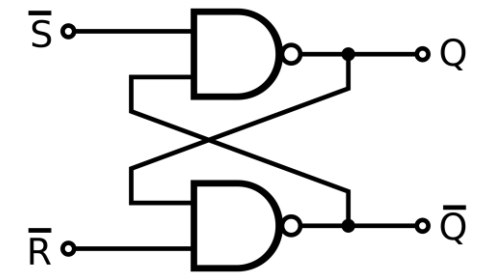


- SR NOR Latch
 - This consists of NOR gates with feedback logic



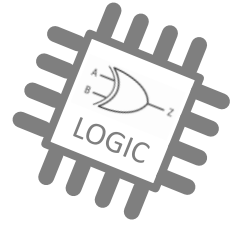
- $Q(\text{next})$ is defined by $Q(\text{curr})$ when both S and R are 0
- $Q_{\text{next}} = S + \bar{R}Q$

S	R	Q(next)
0	0	HOLD
0	1	RESET
1	0	SET
1	1	NOT ALLOWED

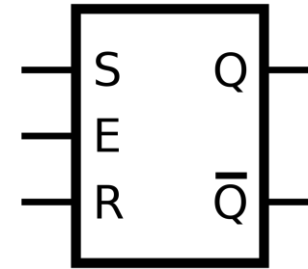
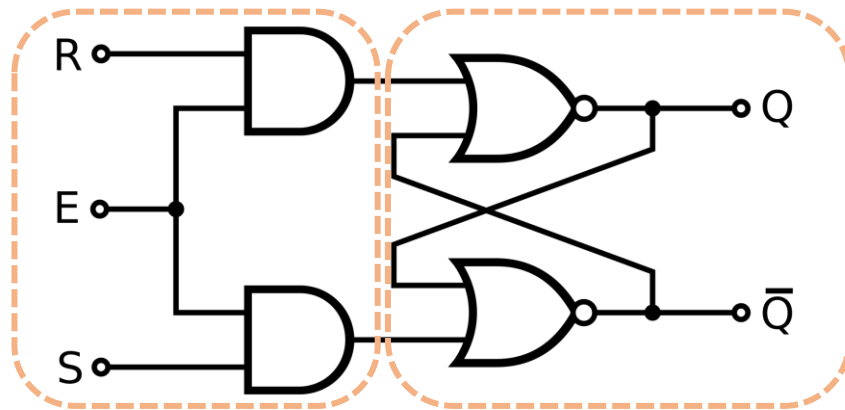


cf) SR NAND Latch

Gated Latch



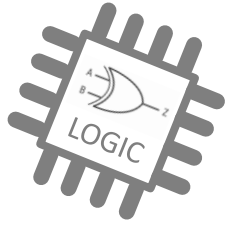
- Gated SR Latch
 - This consists of SR NOR Latch and Enable AND gates



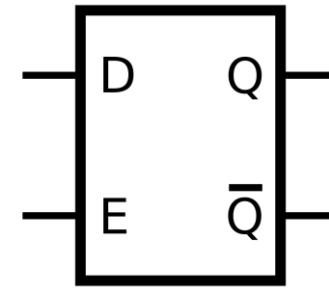
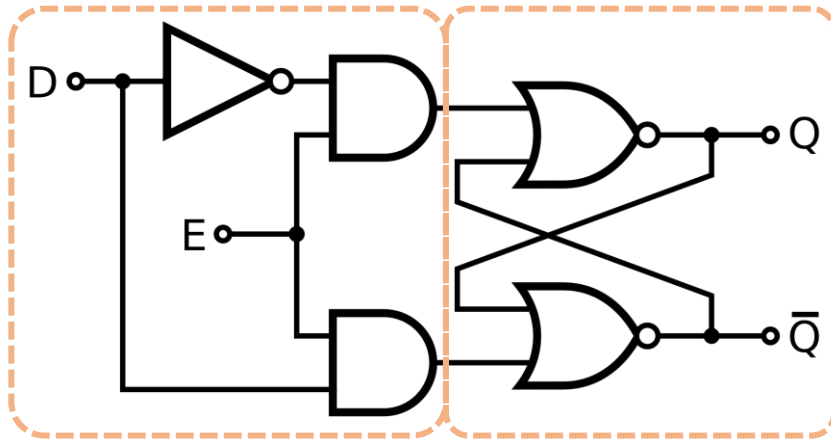
- Gated means Level-sensitive
- Enable signal is usually Clock which toggles at every period
Q can change its value when clock level is high



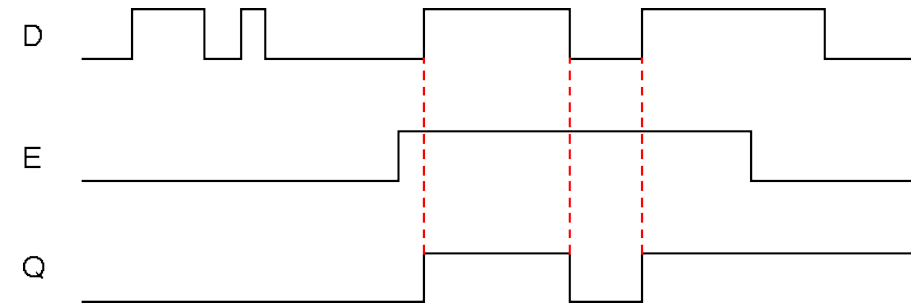
Gated Latch



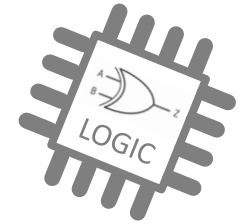
- Gated D Latch
 - Similar with Gated SR Latch, S is D while R is not D



- When E is high, D value propagates to Q
This logic can keep the value while E is low

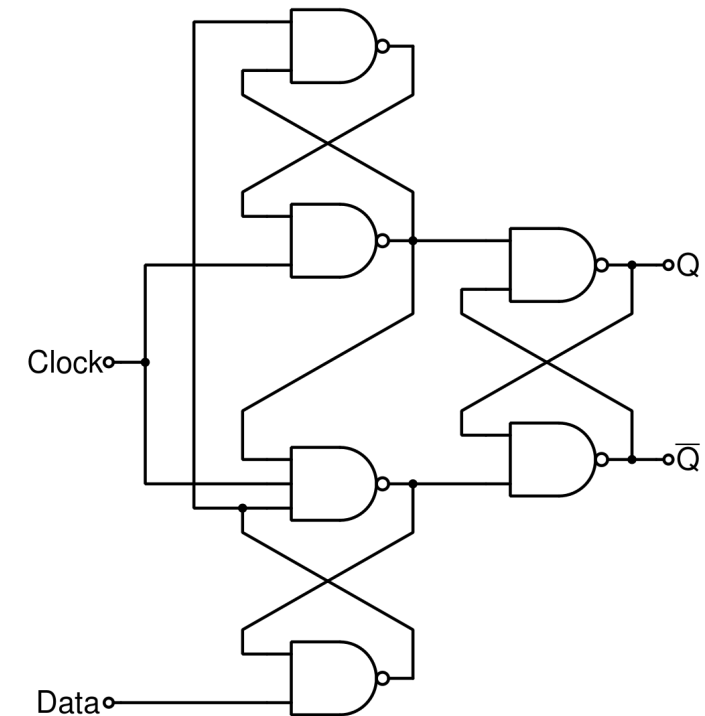


Flip-Flop

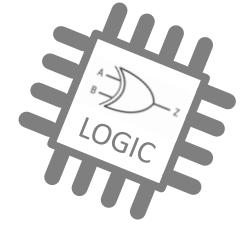


- Flip-Flop
 - Contrary to Latch, Flip-Flop can keep its value when clock (enable) is high
 - It can change Q value only when rising or falling edge of clock
- D Flip-Flop
 - Most common register in Verilog design
 - Almost sequential logic includes this one
 - D(input) value is reflected to Q(output) after clock rising edge
 - Consists of 6 NAND gates
 - Truth table is like right figure

Clock	D	Q(next)
Rising	0	0
Rising	1	1
Non-rising		Q (HOLD)

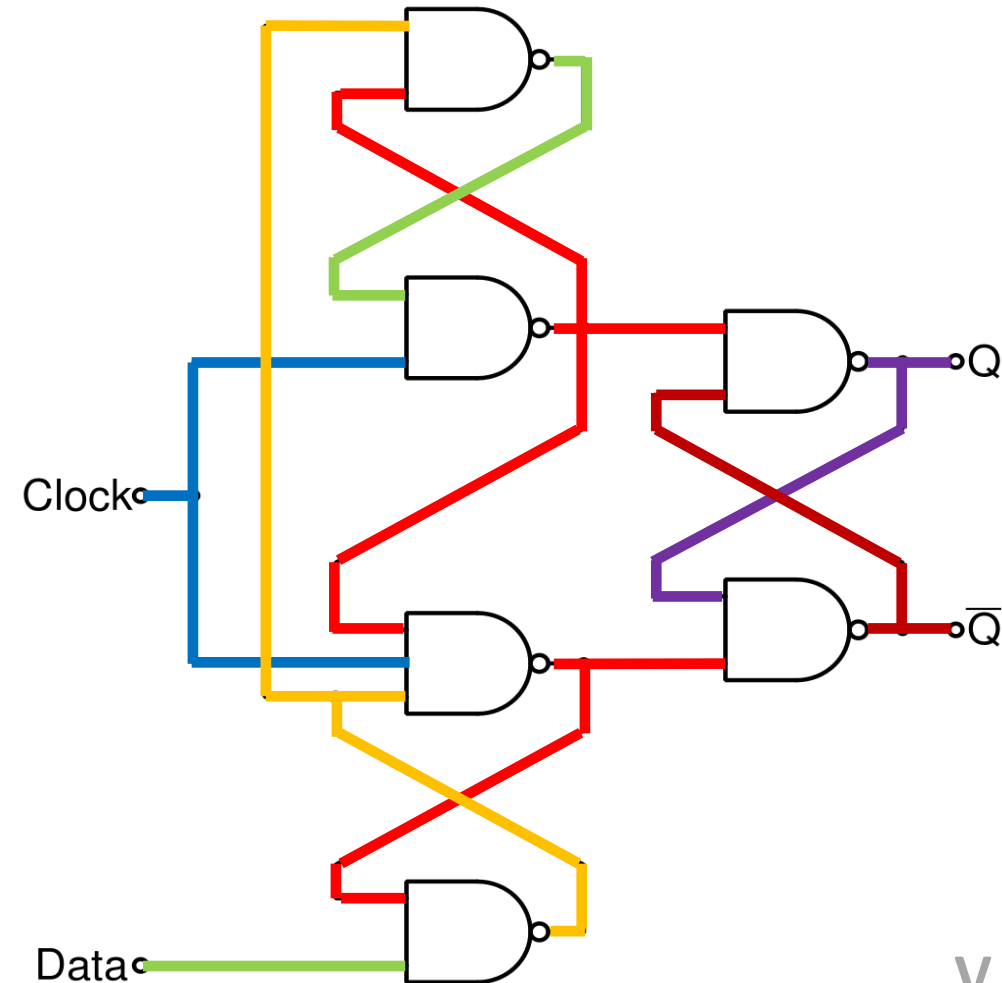


D Flip-Flop - Behavior

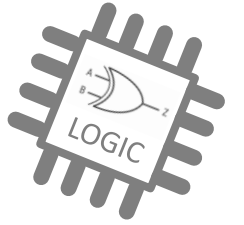


- Clock : 0
 - Clock 0 can make several NAND's output 1
 - This can make Q value keep its original one

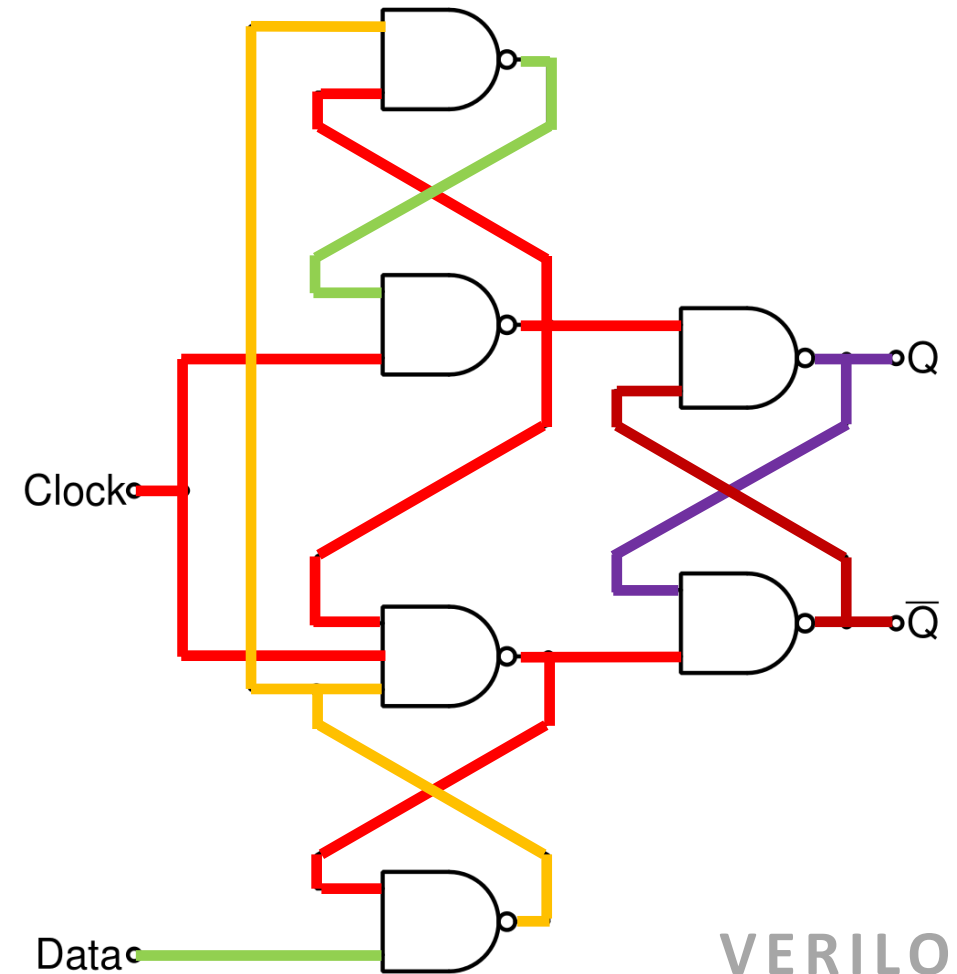
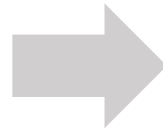
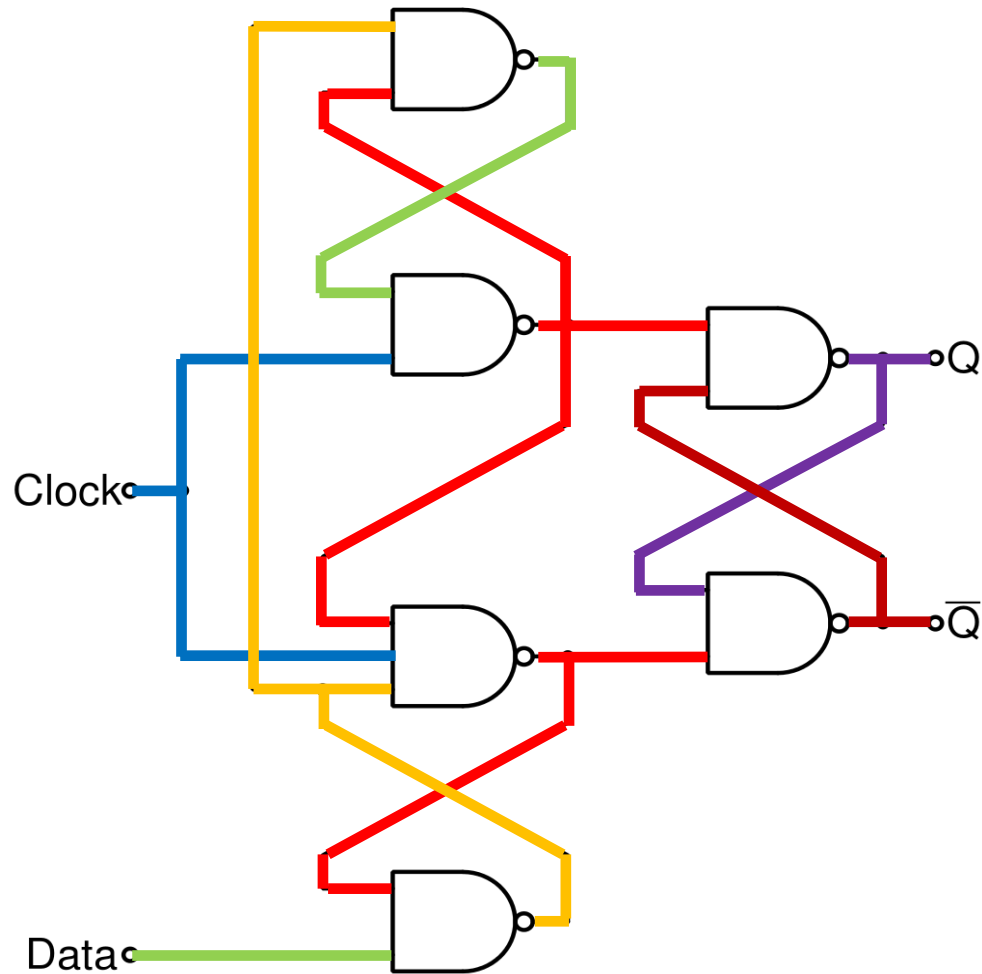
Blue is 0
Red is 1
Green is D
Yellow is D'
Purple is Q(curr)
Brown is Q'(curr)



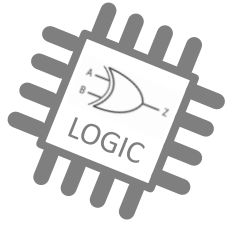
D Flip-Flop - Behavior



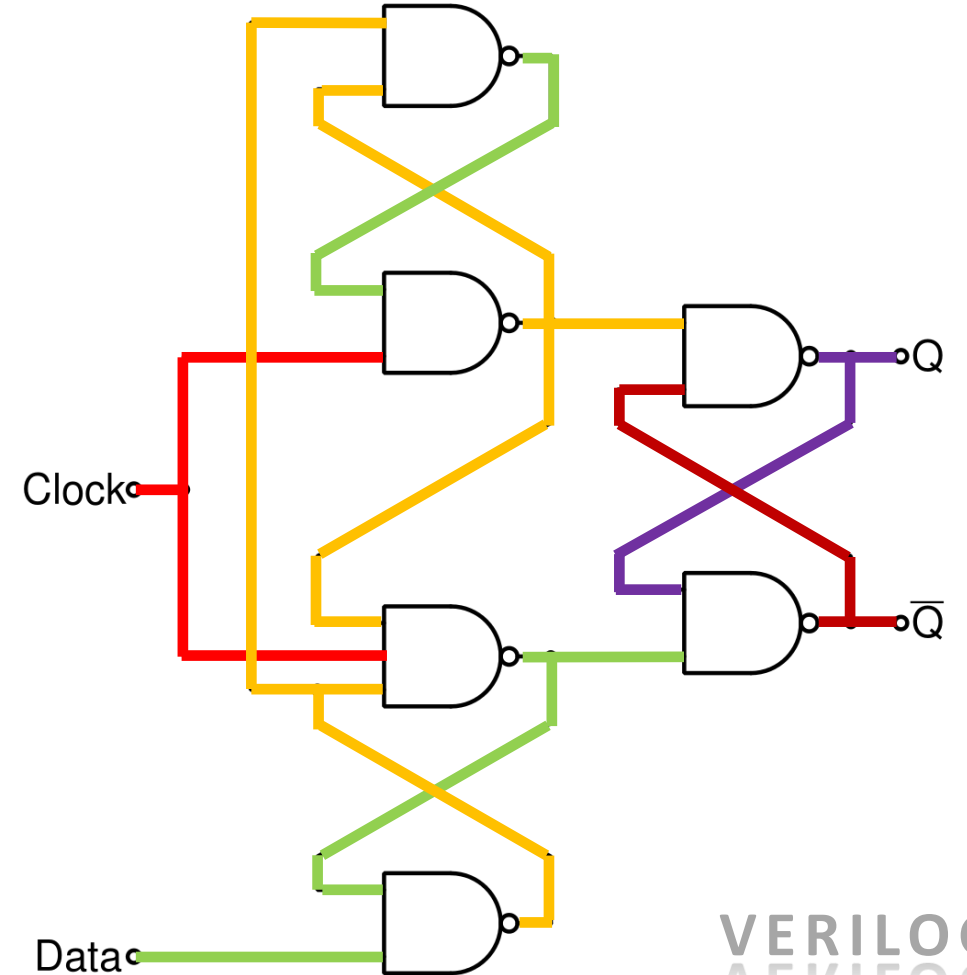
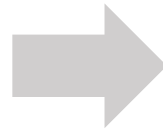
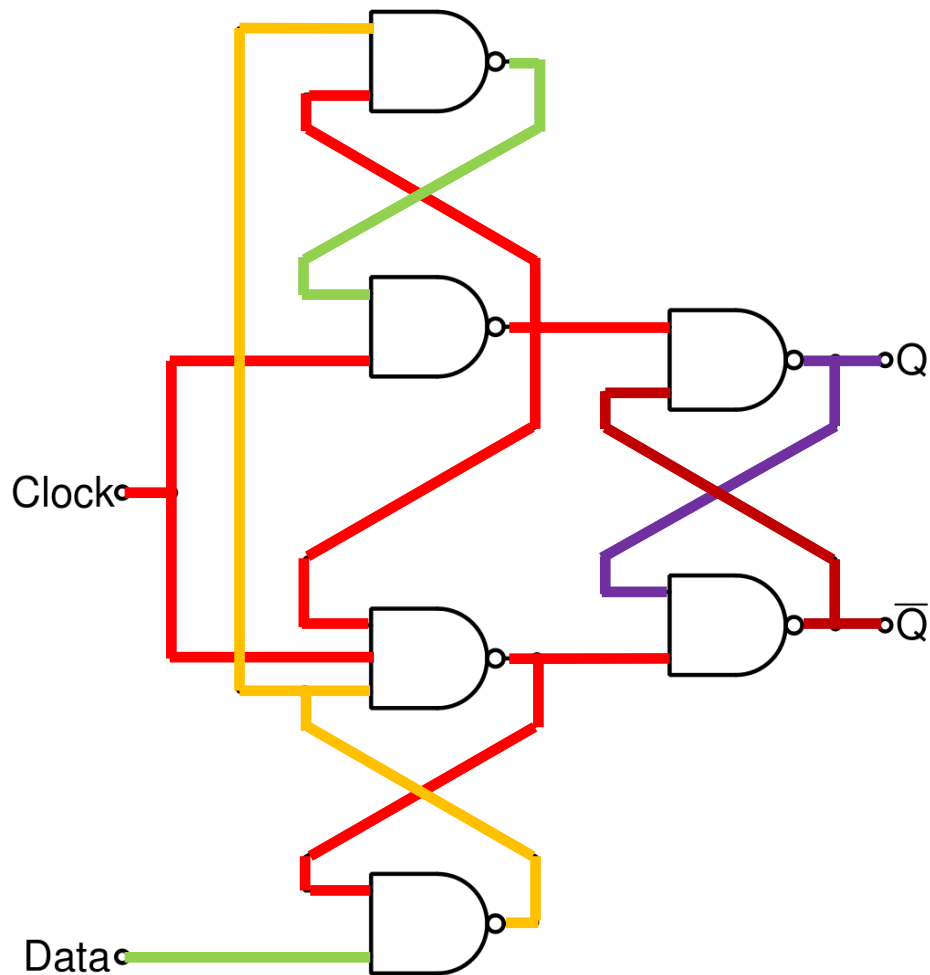
- Clock : 0 \rightarrow 1



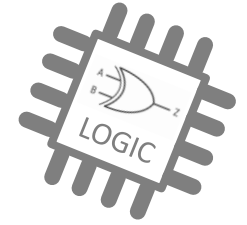
D Flip-Flop - Behavior



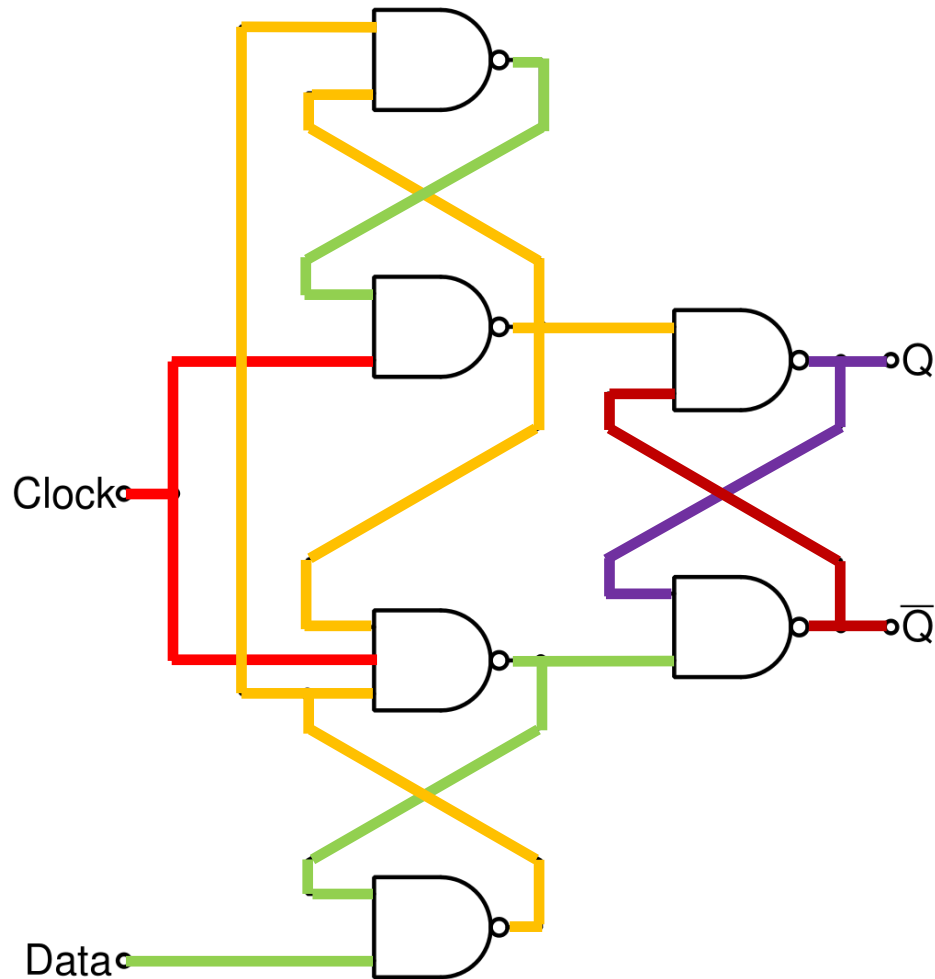
- Clock : 0 \rightarrow 1, the others are changing



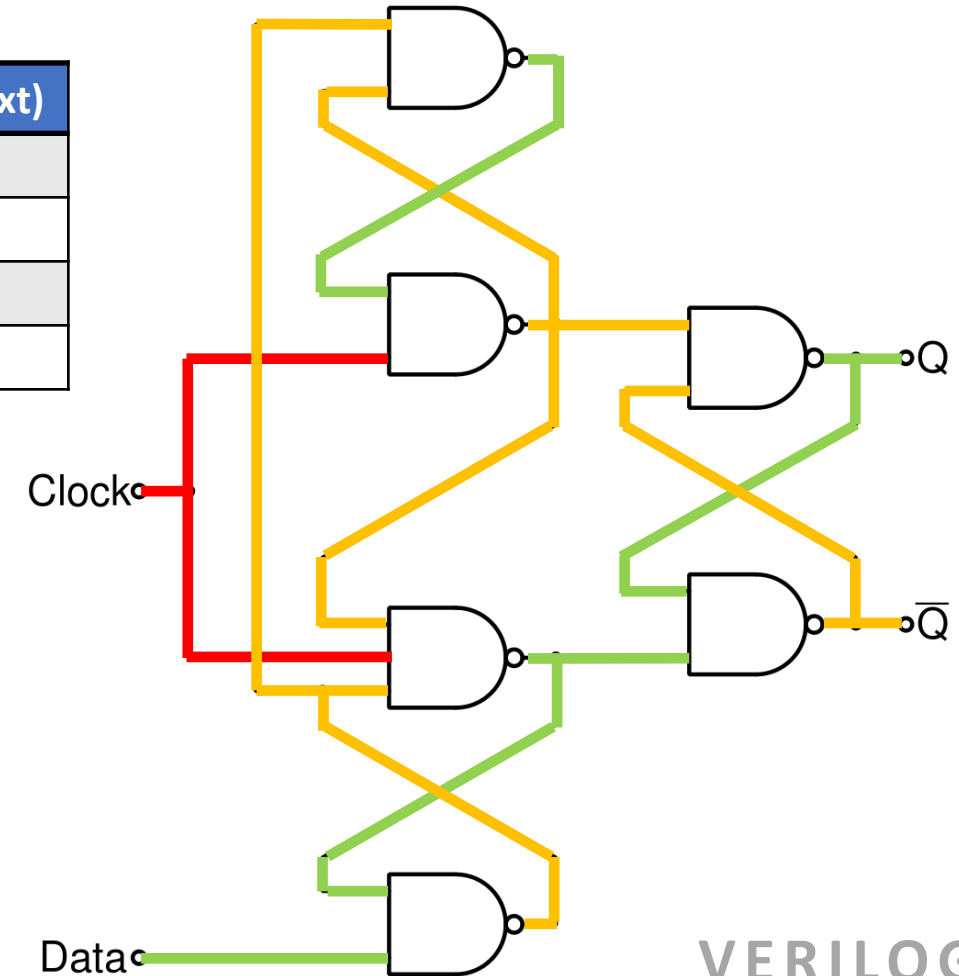
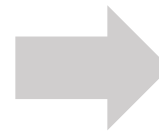
D Flip-Flop - Behavior



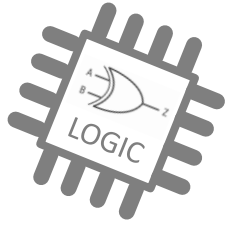
- Clock : 0 \rightarrow 1, Q value changes



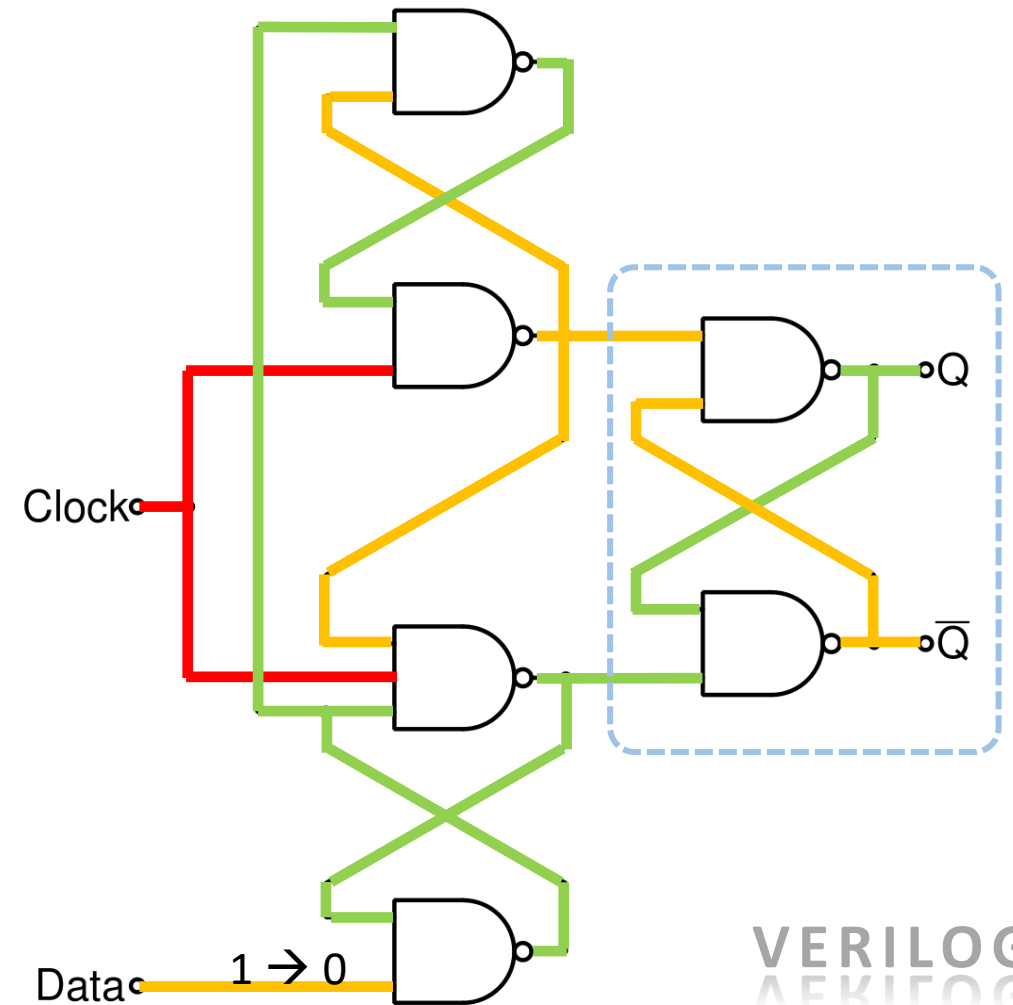
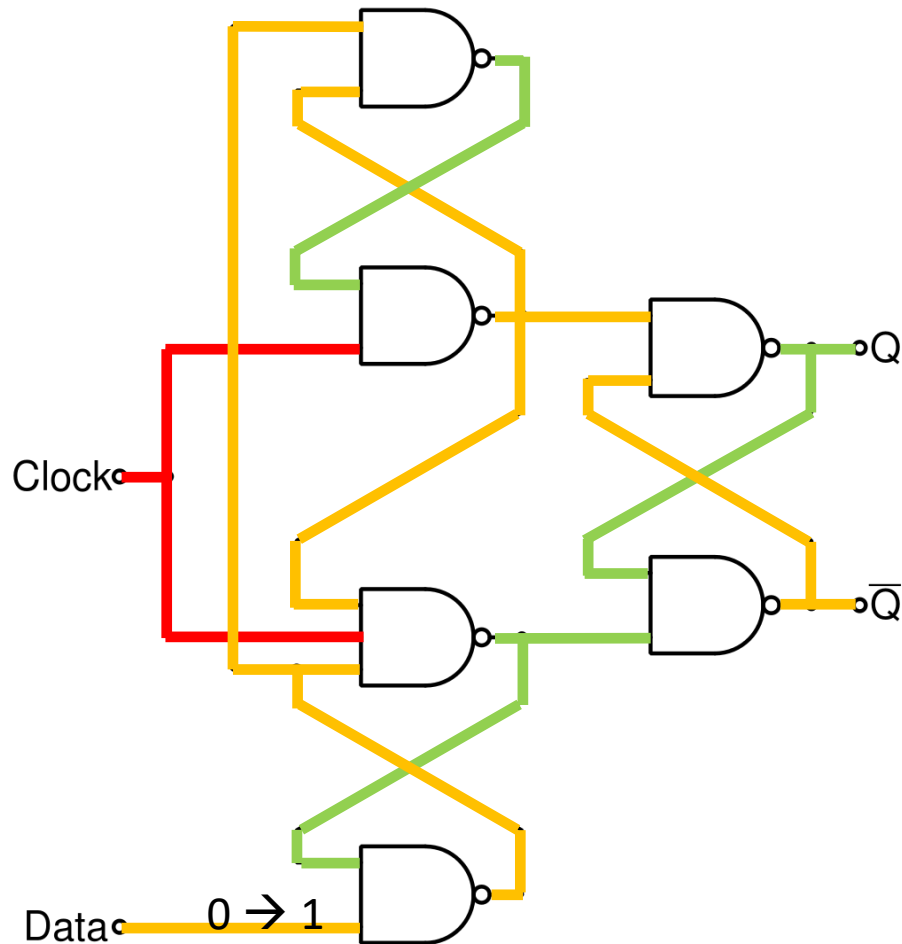
D	Q	Q(next)
0	0	0
0	1	0
1	0	1
1	1	1



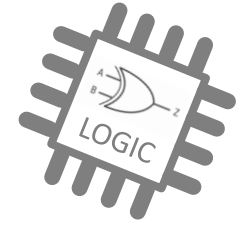
D Flip-Flop - Behavior



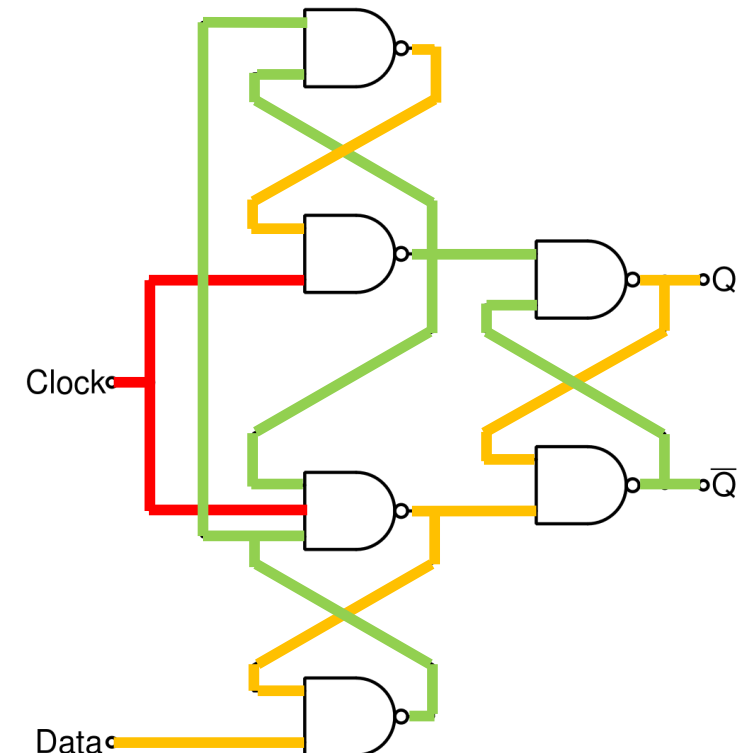
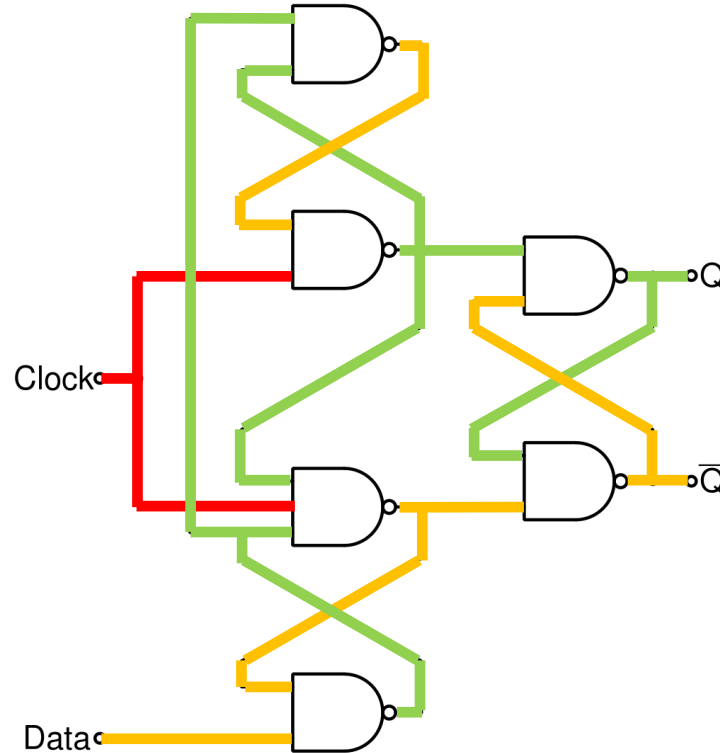
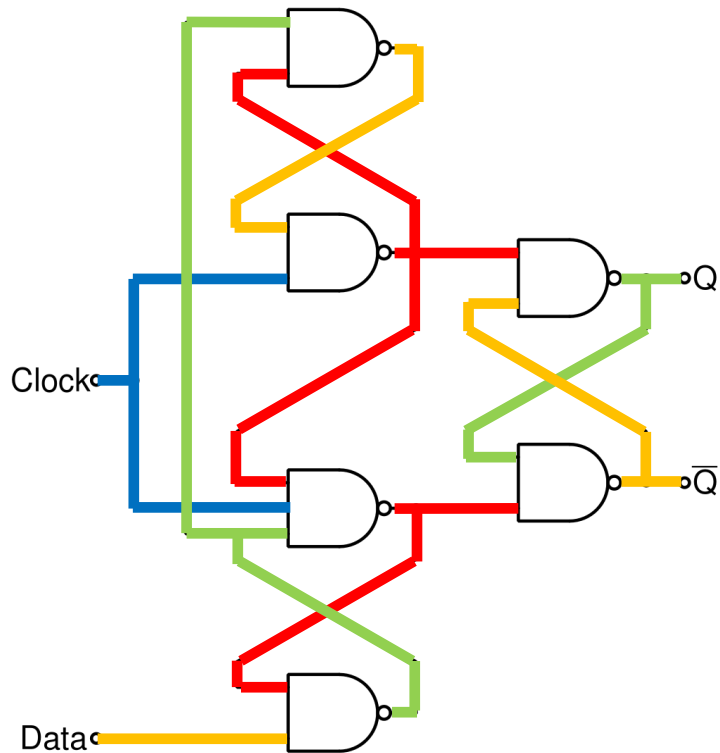
- Clock : 1, Stable, Q value doesn't change



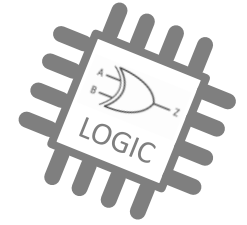
D Flip-Flop - Behavior



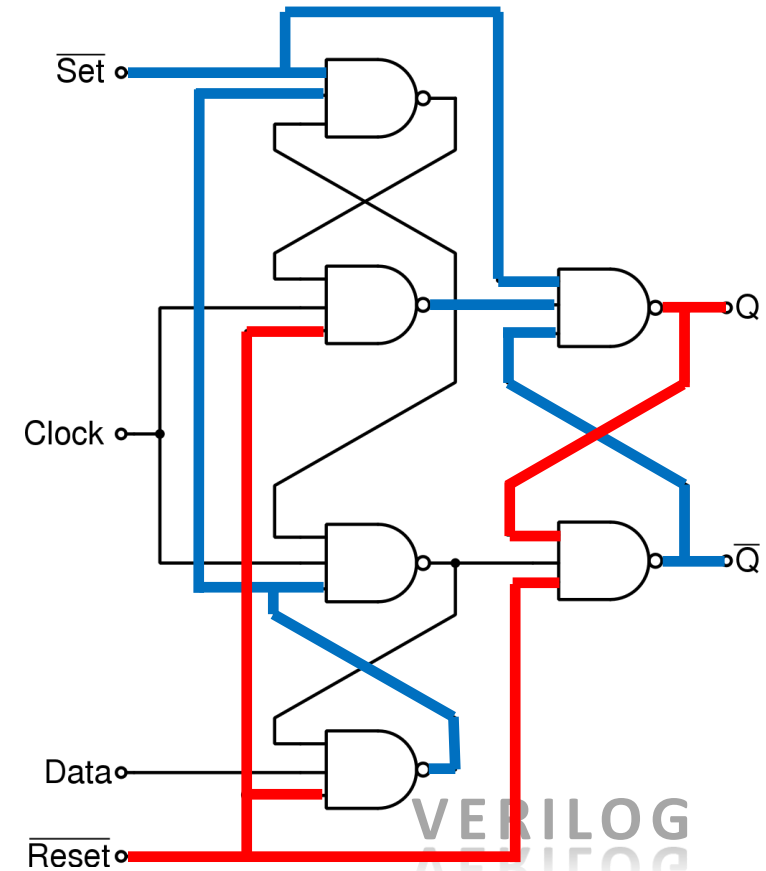
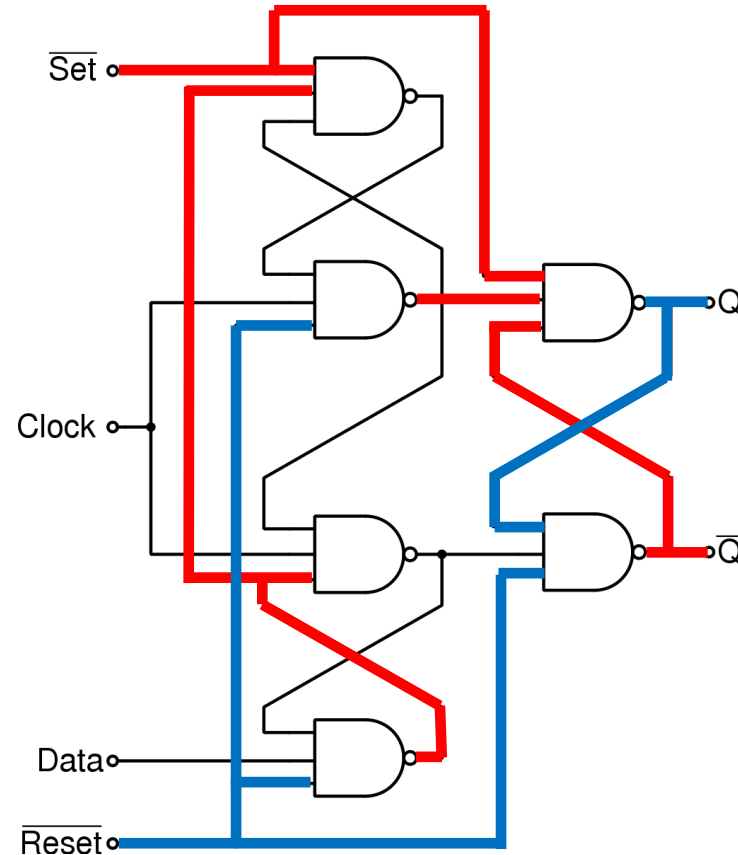
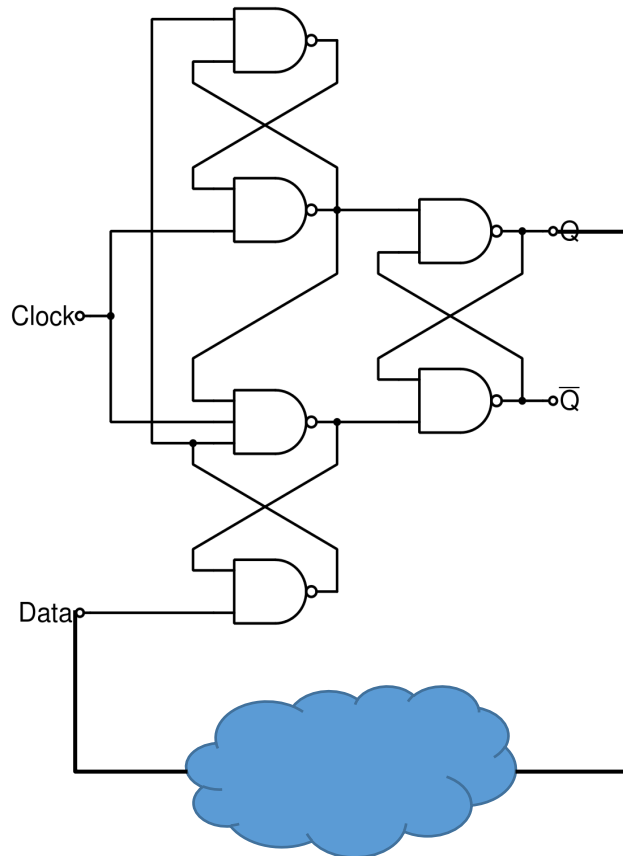
- Clock : $1 \rightarrow 0 \rightarrow 1$
 - Q value can change at rising edge



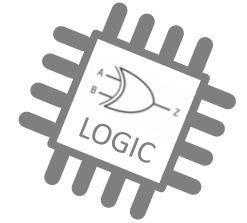
D Flip-Flop + Comb



- Sequential Logic
 - D can be combination of Q and logic (feedback)
 - Normally, DFF needs initial value (Reset)

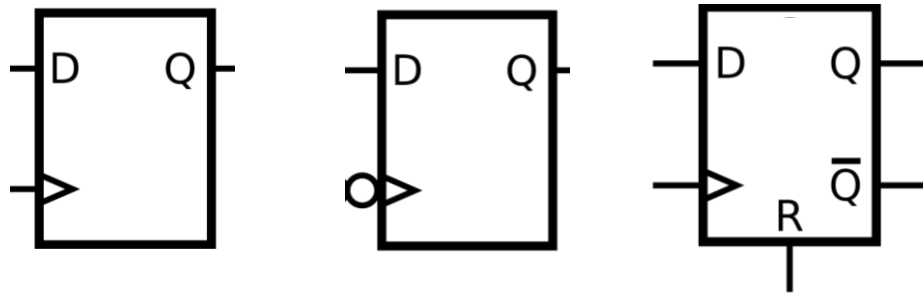


D Flip-Flop

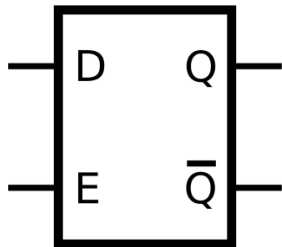


- Symbol

- DFF's symbol is similar to Latch's one triangle means edge-triggered (clock)
- Reset is normal while set is not

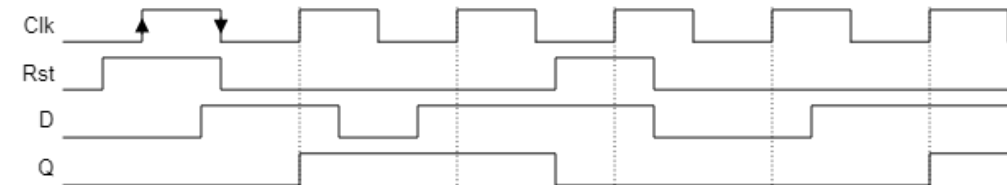


- *cf) Latch's symbol*

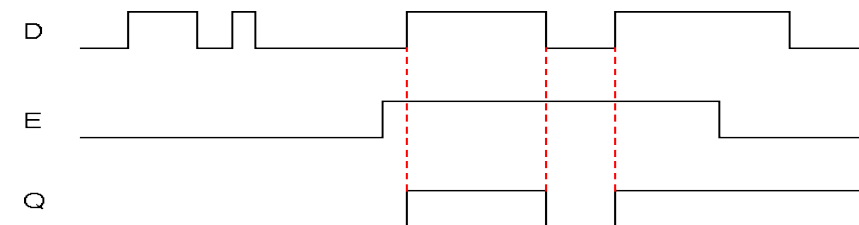


- Waveform

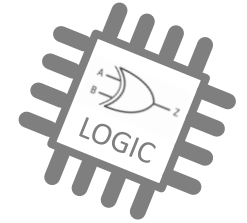
- DFF can reflect value when clk is rising
- DFF is 0 when rst(rstn) is(de) asserted



- *cf) Latch can reflect value when E is high*

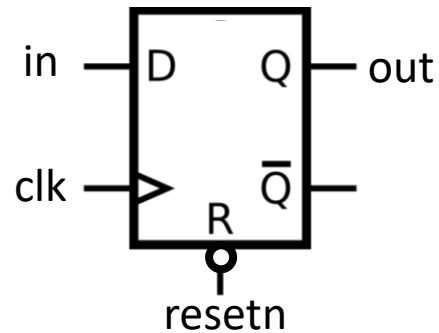


D Flip-Flop



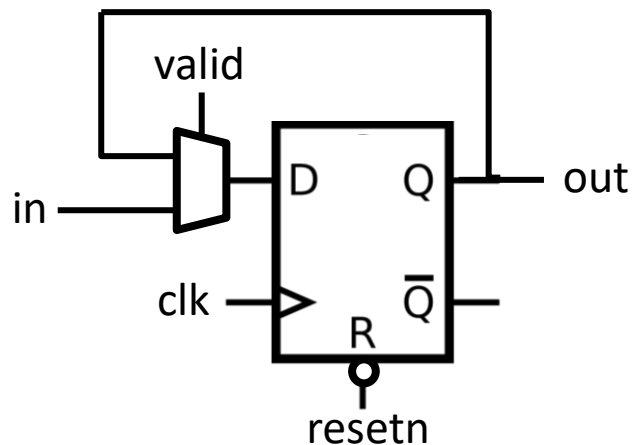
- Verilog coding

- Basic DFF



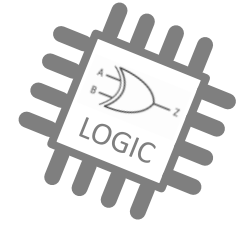
```
always @(posedge clk, negedge resetn)
if (!resetn) out <= 0;
else      out <= in;
```

- D is reflected to Q only when D is valid. Otherwise, DFF keeps its value
MUX can be translated to conditional expression



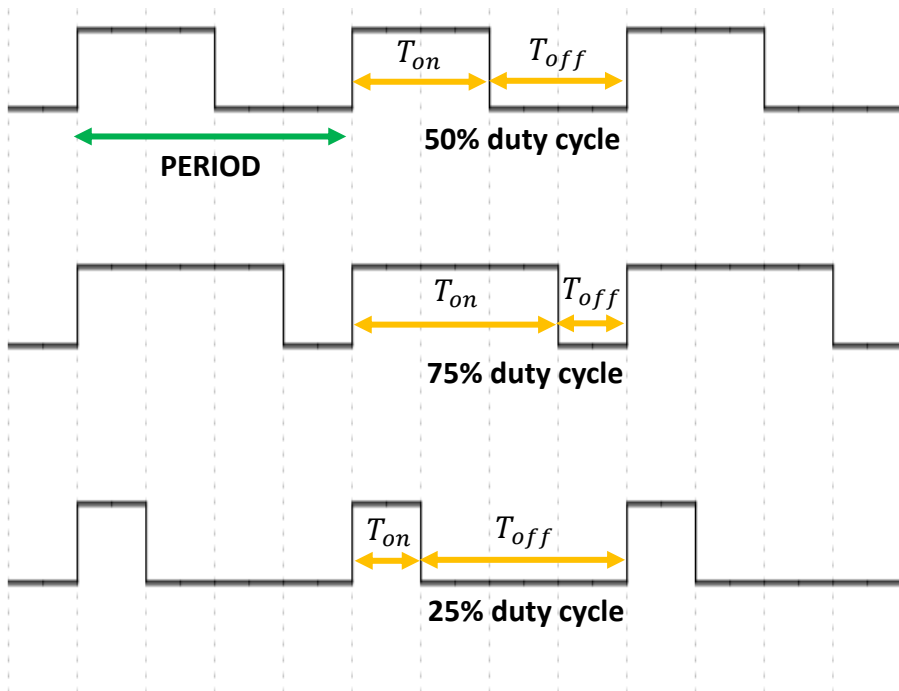
```
always @(posedge clk, negedge resetn)
if (!resetn) out <= 0;
else if (valid) out <= in;
```

Clock

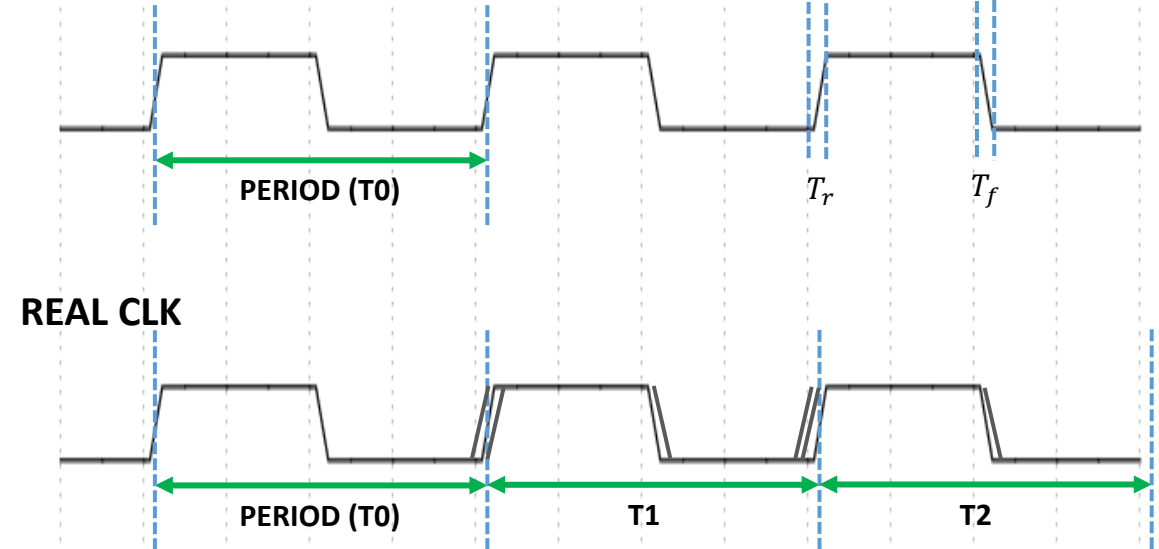


- Clock properties
 - frequency, duty cycle
 - jitter

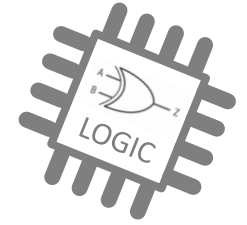
Frequency = 1 (sec) / PERIOD , cf) 200MHz = 5ns period



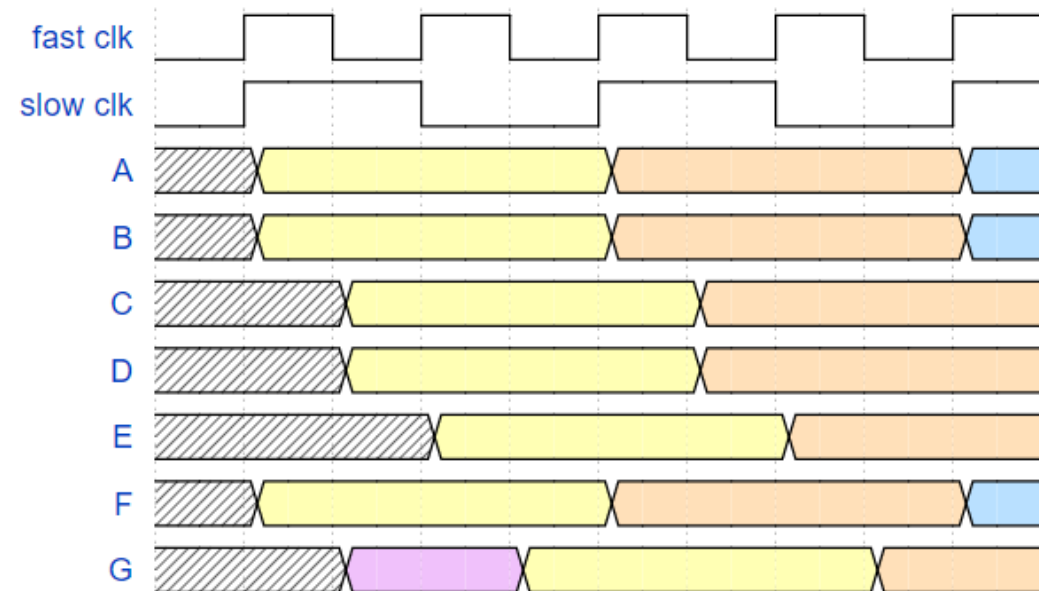
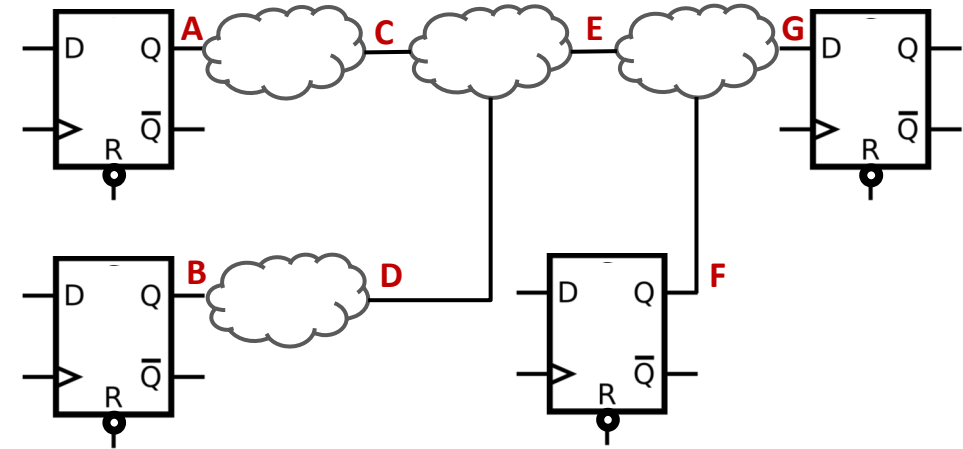
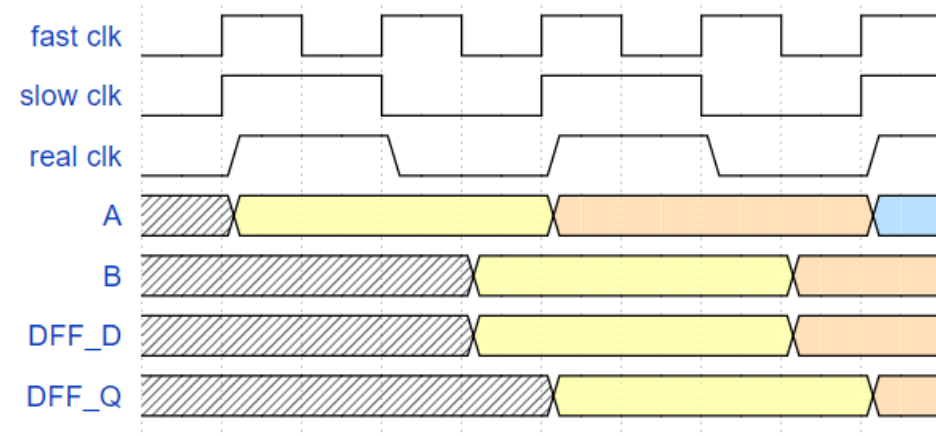
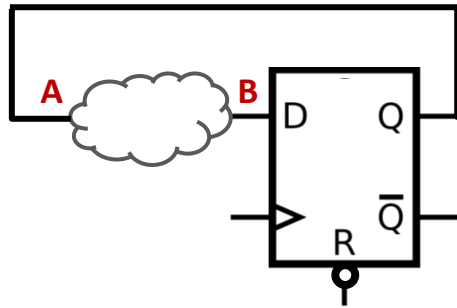
IDEAL CLK



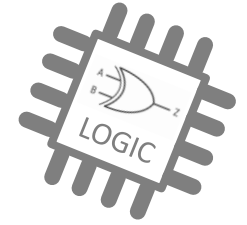
Clock



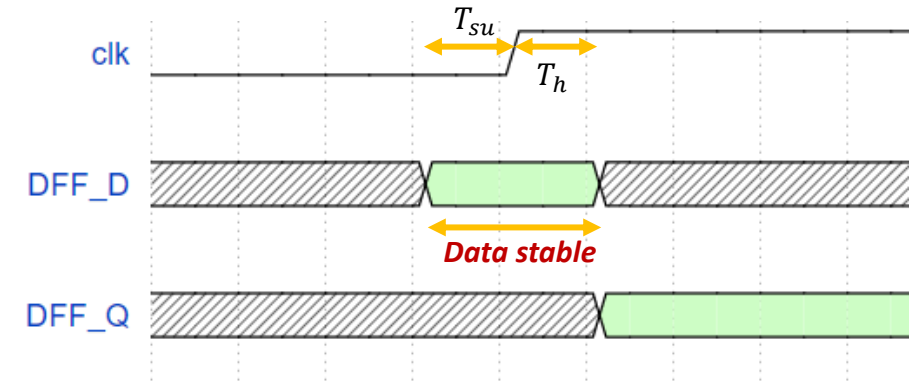
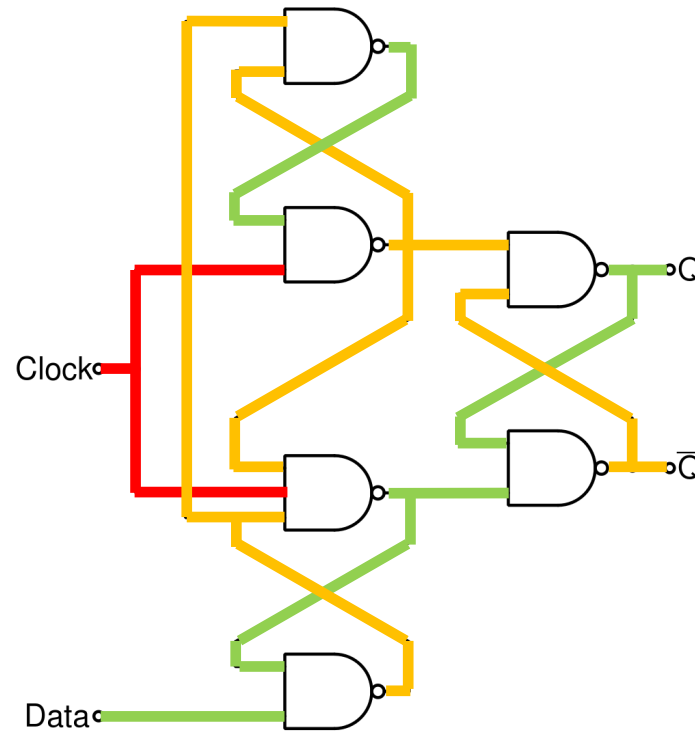
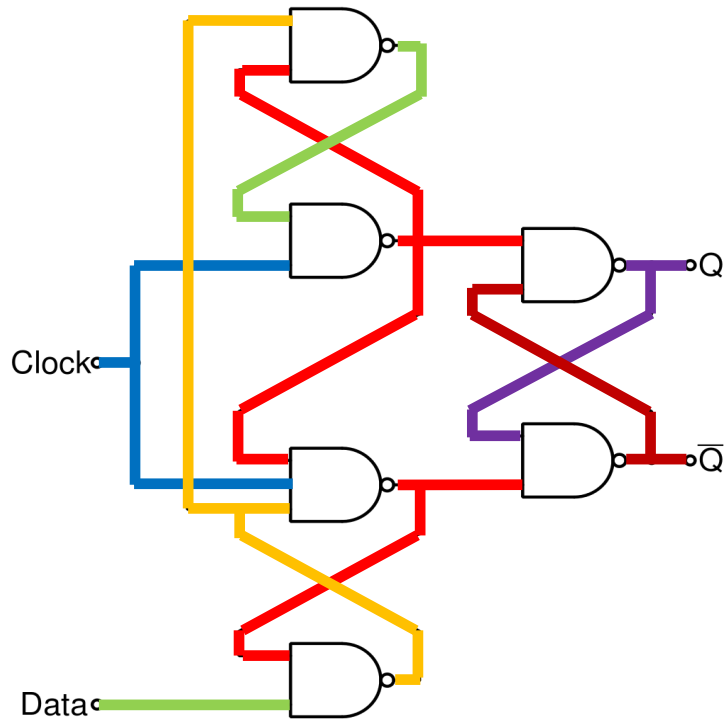
- How to decide clock frequency
 - $\text{Period} > \text{Gate delay} + \text{Rising time} + \text{Jitter}$



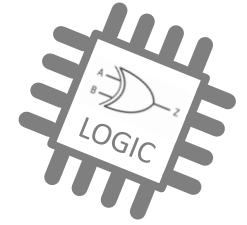
Clock



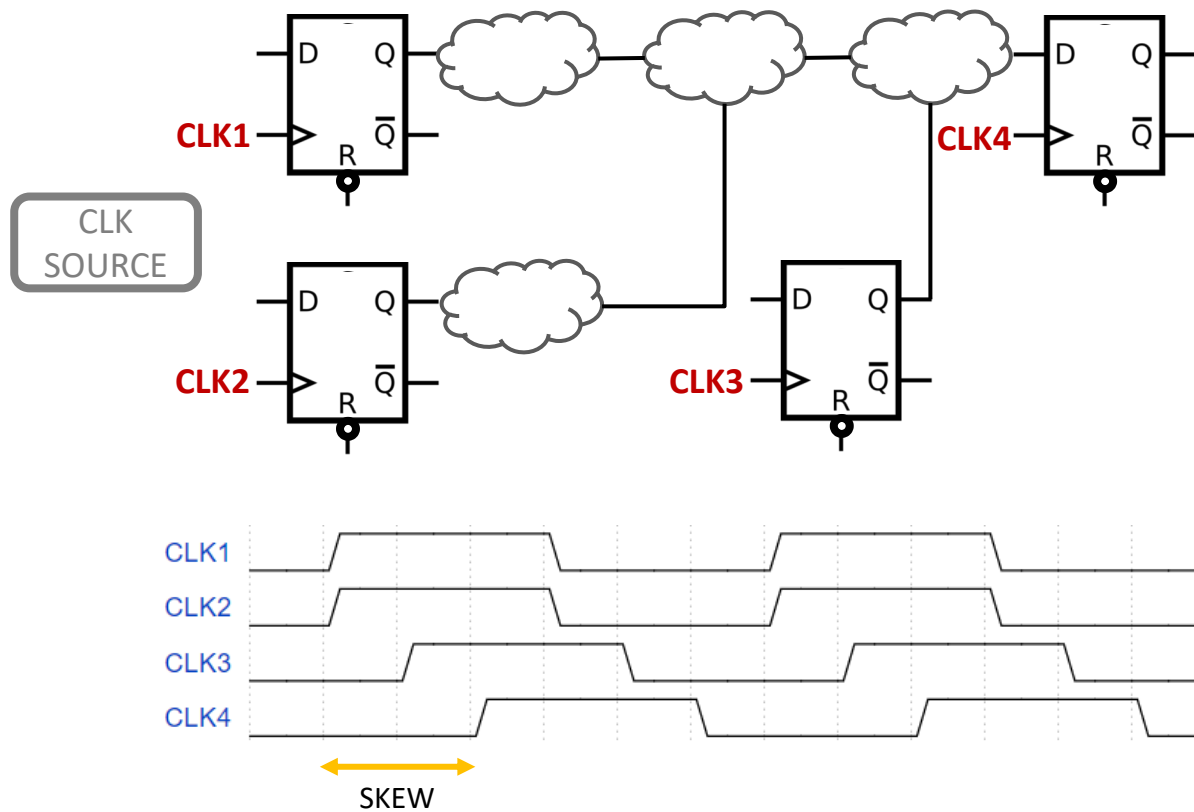
- How to decide clock frequency
 - $\text{Period} > \text{Gate delay} + \text{Rising time} + \text{Jitter} + \text{Setup time} + \text{Hold time}$
 - Data is stable before and while clock rising



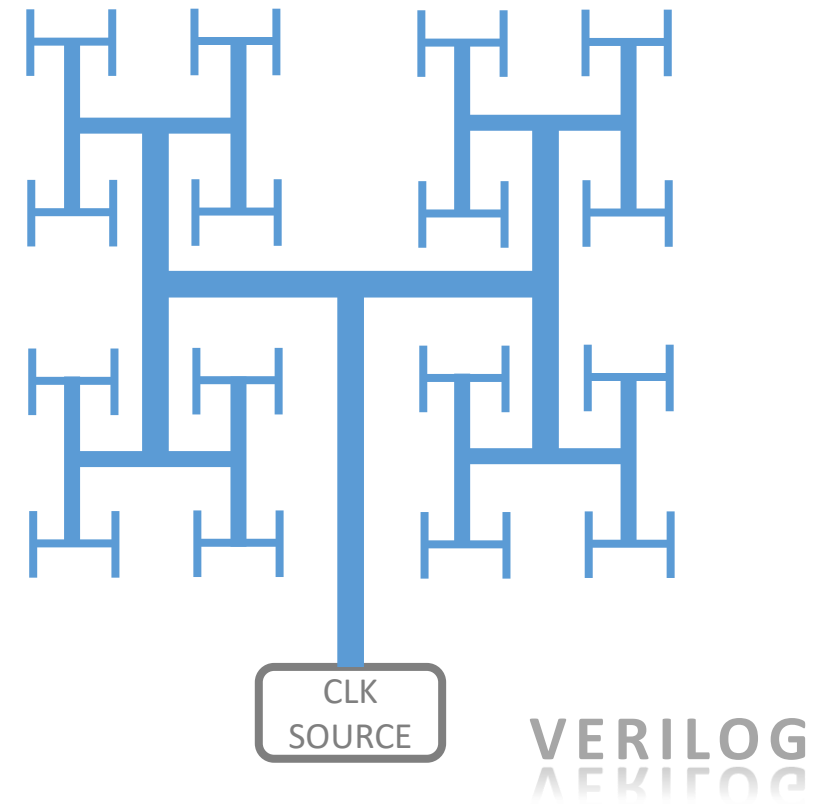
Clock



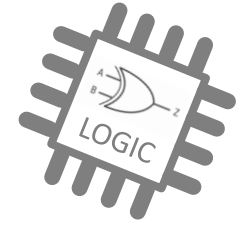
- Clock skew
 - We have lots of DFF which needs clock
 - $\text{Period} > \text{Gate delay} + \text{Rising time} + \text{Jitter} + \text{Setup time} + \text{Hold time} + \text{Skew}$
 - **H tree** is one of the solution to reduce clock skew



H tree for clock distribution

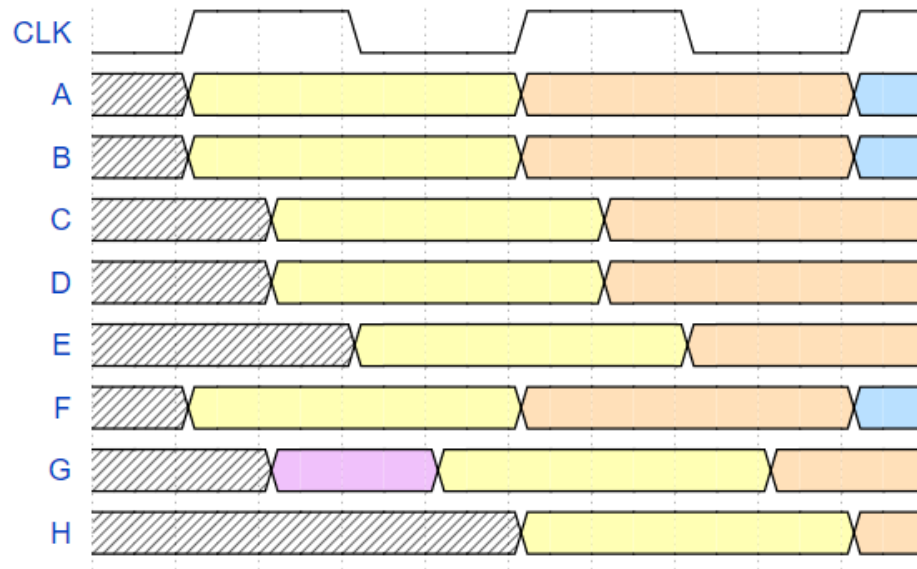
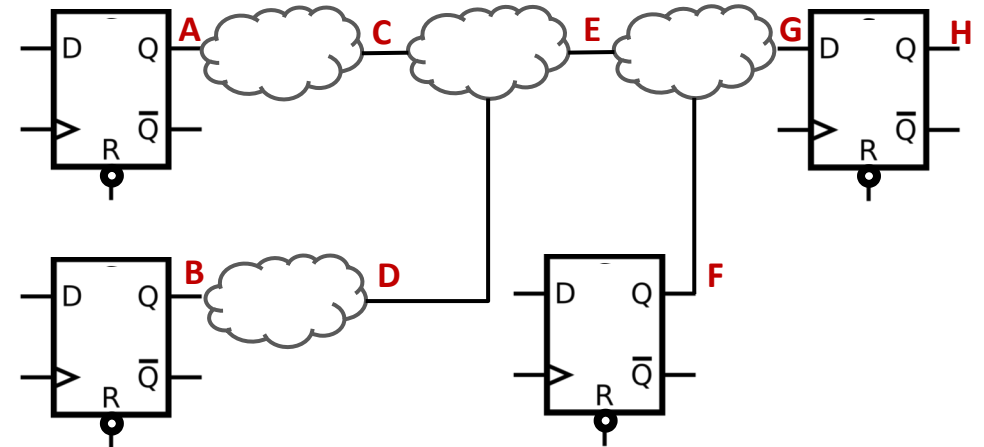


Clock

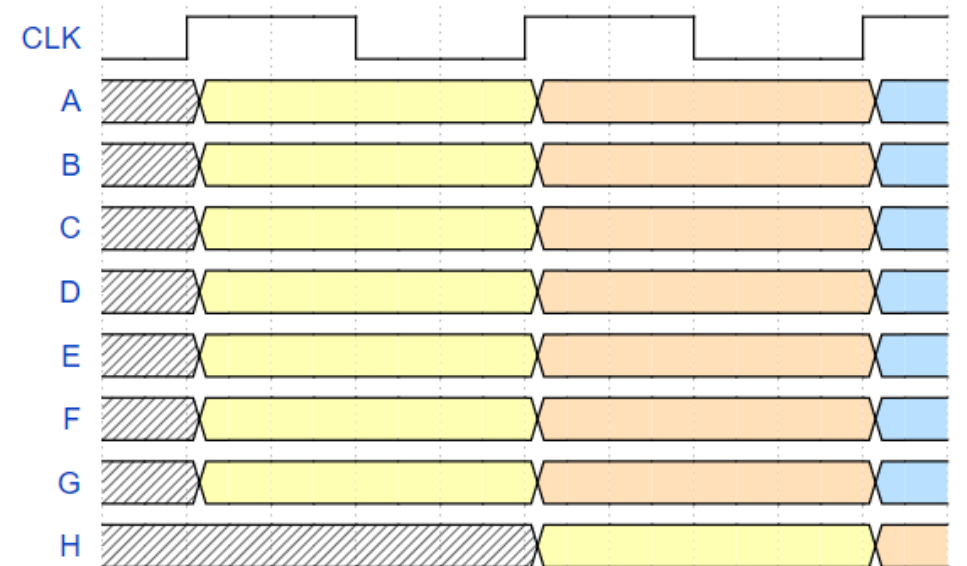


- Waveform in simulation
 - Simulation is ideal situation
 - $\text{Period} > \text{Gate delay} + \text{Jitter} + \text{Setup time} + \text{Hold time} + \text{Skew}$

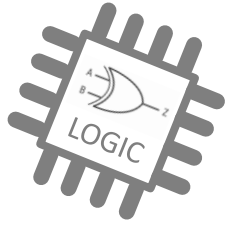
No gate delay, No Jitter, No Setup time, No Hold time
No Rising time, No Skew



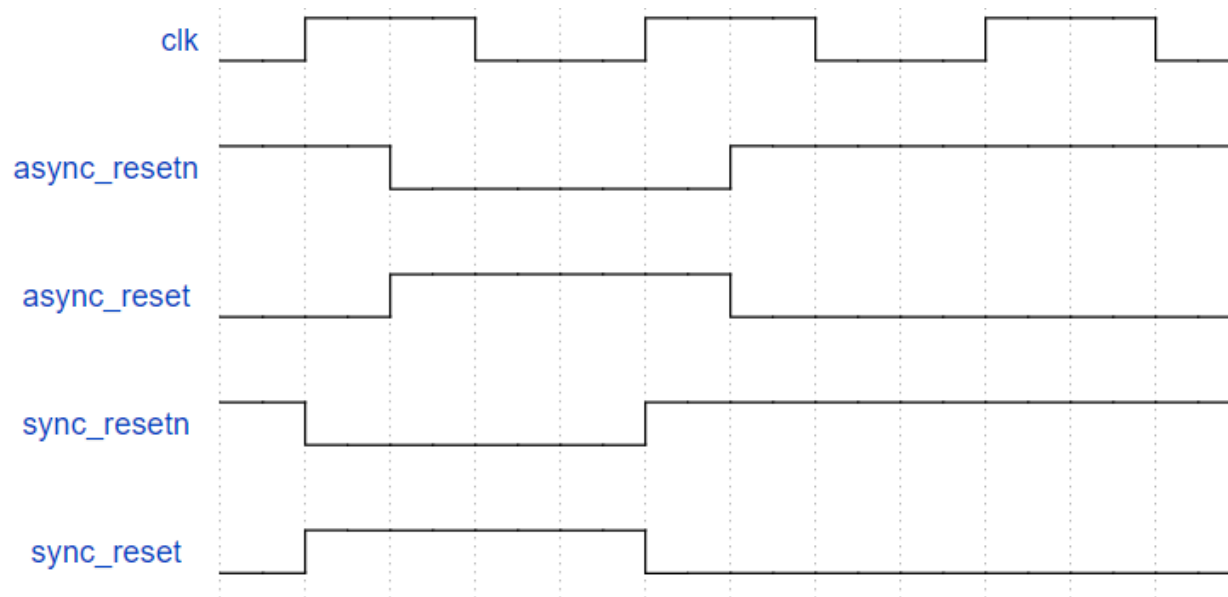
Simulation



Reset



- Reset
 - sync / async
 - active low / active high



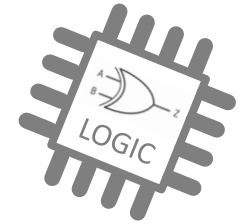
```
always @(posedge clk, negedge async_resetn)
if (!async_resetn) out <= 0;
else out <= in;
```

```
always @(posedge clk, posedge async_reset)
if (async_reset) out <= 0;
else out <= in;
```

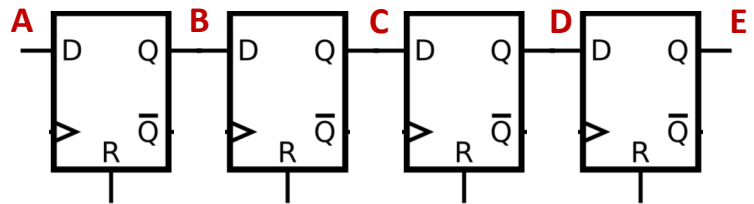
```
always @(posedge clk)
if (!sync_resetn) out <= 0;
else out <= in;
```

```
always @(posedge clk)
if (sync_reset) out <= 0;
else out <= in;
```

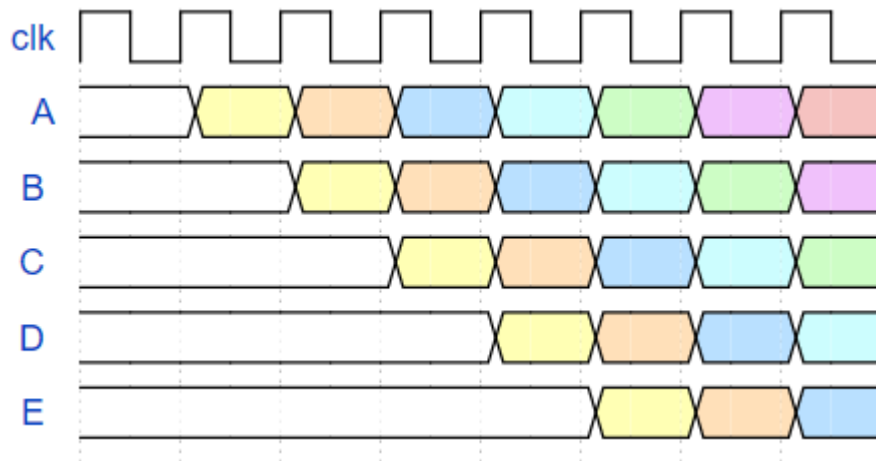
Sequential Logic



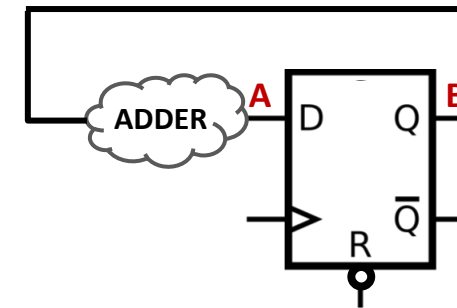
- Shift register
 - *save data or delay data*



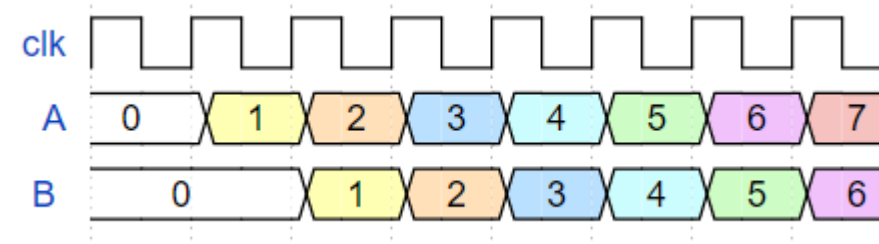
```
always @(posedge clk, posedge async_reset)
if (async_reset) {E,D,C,B} <= 0;
else {E,D,C,B} <= {D,C,B,A};
```



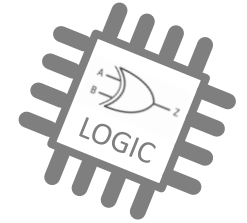
- Counter
 - *counting number*



```
always @(posedge clk, posedge async_reset)
if (async_reset) B <= 0;
else B <= B+1;
```



Contents



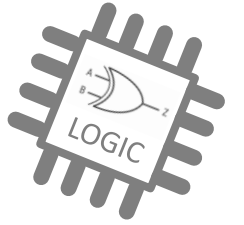
Explanation

- Introduction to Semiconductor field
- Basic concept of Verilog
- Combinational Logic
- Sequential Logic & Clock/Reset
- FSM

Coding

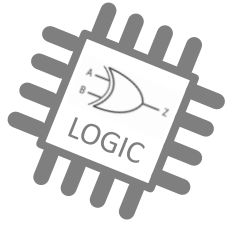
- Free Tools
- **Declarations & Modeling**
- DUT & Testbench
- Task & Function
- Lots of Practices

Agenda



- Data type
- Module & Port & Instance
- Operator
- Modeling
- Behavior Modeling

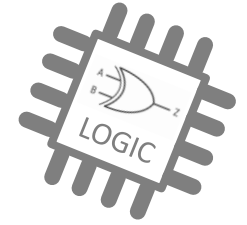
How to print



- ***\$display** have a role as similar as printf in C*
- ***\$monitor** is a little bit different. It is active only when variables change*

```
1 `timescale 1ns/1ns
2 module hello_world;
3     int test = 0;
4
5     initial begin
6         #10 $display("time %t : hello world %d ",$time,test++);
7         #10
8         $display("time %t : hello world %d ",$time,test++);
9         $display("time %t : hello world %d ",$time,test++);
10
11     $finish;
12 end
13
14     initial begin
15         $monitor("monitor test %d",test);
16     end
17 endmodule
```

How to debug

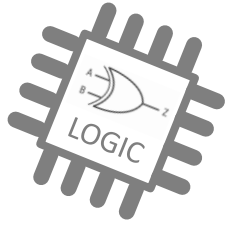


- ***\$dumpfile** define output files which can save waveform (*.vcd)*
- ***\$dumpvars** define the range of waveform dump*

```
1 module dump_example;
2     reg a, b;
3
4     initial begin
5         a=0; b=0;
6         #10 a=1; b=0;
7         #10 a=0; b=1;
8         #10 a=1; b=1;
9         $finish;
10    end
11
12    initial begin
13        $dumpfile("dump_example.vcd");
14        $dumpvars(1, dump_example);
15        $display("test waveform dump");
16    end
17 endmodule
```

- *Let's go to “edaplayground.com”*

Data type



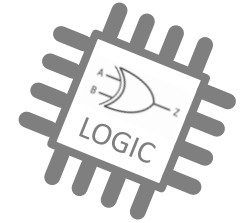
- Value Set
 - **0 : logic 0, false**
 - **1 : logic 1, true**
 - x : unknown
 - z : high-impedance, open connection
- Format
 - Binary : 'b
 - Decimal : ('d)
 - Hexa : 'h

 - *Ex) a = 'b1010; a = 'd10; a = 10; a = 'hA;*
 - *Ex) a = 4'b1010; a = 4'd10; a = 10; a = 4'hA;*

 - *cf) \$display("binary %b, decimal %d, hexa %h",bin_a,dec_b,hex_c);*

 - *cf) a = '0; a = '1; a = 'x; a = 'z;*

Data type



- Nets

- physical connection between entities
- Do not store values
This data type is always affected by driving circuit
- Normally used in combinational logic
- **wire**, tri, wand, wor, supply0, supply1,

- *assign **wire_a** = ... ;
module_a inst_a (**wire_a**)*



- *module_b inst_b (**wire_b**);
module_c inst_c (**wire_b**);*

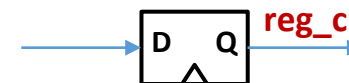


- Variables

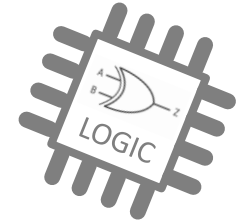
- used in procedural blocks
- Store values
- Normally used in not only sequential logic but also combinational logic
- **reg**, int, real, time

- *initial begin reg_a = ...; end*
- *always @(*) begin reg_b = ...; end*
- *reg_a, reg_b keeps its value before next line*

- *always @(posedge clk) begin reg_c <= ...; end*



Data type



- Vectors

- nets, variables are usually 1 bit type
- We can define bit width “N” like

```
wire [N-1:0] wire_a;  
reg  [N-1:0] reg_a;
```

```
wire      wire_1bit;  
wire [7:0] wire_8bit;  
  
reg      reg_1bit;  
reg [7:0] reg_8bit;  
  
assign    wire_1bit = 1'b0;  
assign    wire_8bit = 8'hFF;  
  
always @* begin  
    reg_1bit = 1'b1;  
    reg_8bit = 8'hF0;  
end  
  
wire [7:0] wire_8bit_2 = 'hF;
```

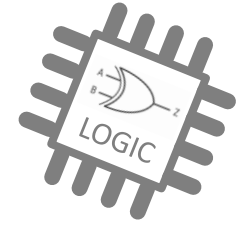
- Array

- Data type can have its dimensions
- $[N-1:0]$ after data type keyword means bit width=N (vector) while $[M-1:0]$ after variable name means its depth=M (scalar)

```
wire [N-1:0] array_a [M-1:0];  
reg  [N-1:0] array_b [M-1:0];
```

```
reg [3:0]    vector_a;           //4-bit width reg vector  
reg          scalar_a [3:0];    //scalar reg array of depth 4, each 1-bit  
  
reg [3:0]    array_1D [7:0];     //4-bit reg vector with depth 8, 1D array  
reg [3:0]    array_2D [1:0][7:0]; //2D array with 2 rows, 8 columns, each 4-bit wide  
  
always @* begin  
    vector_a      = 'h0;  
  
    //scalar_a    = 'h0;        //illegal  
    scalar_a[0]   = 'h0;  
    scalar_a[1]   = 'h0;  
    //...  
  
    array_1D[0]   = 'h0;  
    array_2D[0][0] = 'h0;  
end
```

Module & Port



- Module
 - Design Unit in general
 - Usually, one module in a file with same name



```
module DESIGN_A (  
    clk, rstn,  
    in_a, in_b,  
    out_a, out_b  
);  
    input  clk;  
    input  rstn;  
    input  in_a;  
    input  in_b;  
    output out_a;  
    output out_b;  
  
    //Code for DESIGN_A  
    //....  
    //....  
  
endmodule
```

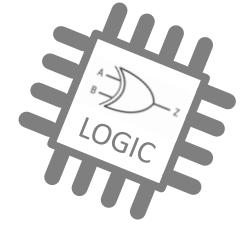
```
module DESIGN_A (  
    input  clk,  
    input  rstn,  
    input  in_a,  
    input  in_b,  
    output out_a,  
    output out_b  
);  
  
    //Code for DESIGN_A  
    //....  
    //....  
  
endmodule
```

- Ports
 - input & output & (inout) ,
 - interface between modules
 - We can express bit width as data type



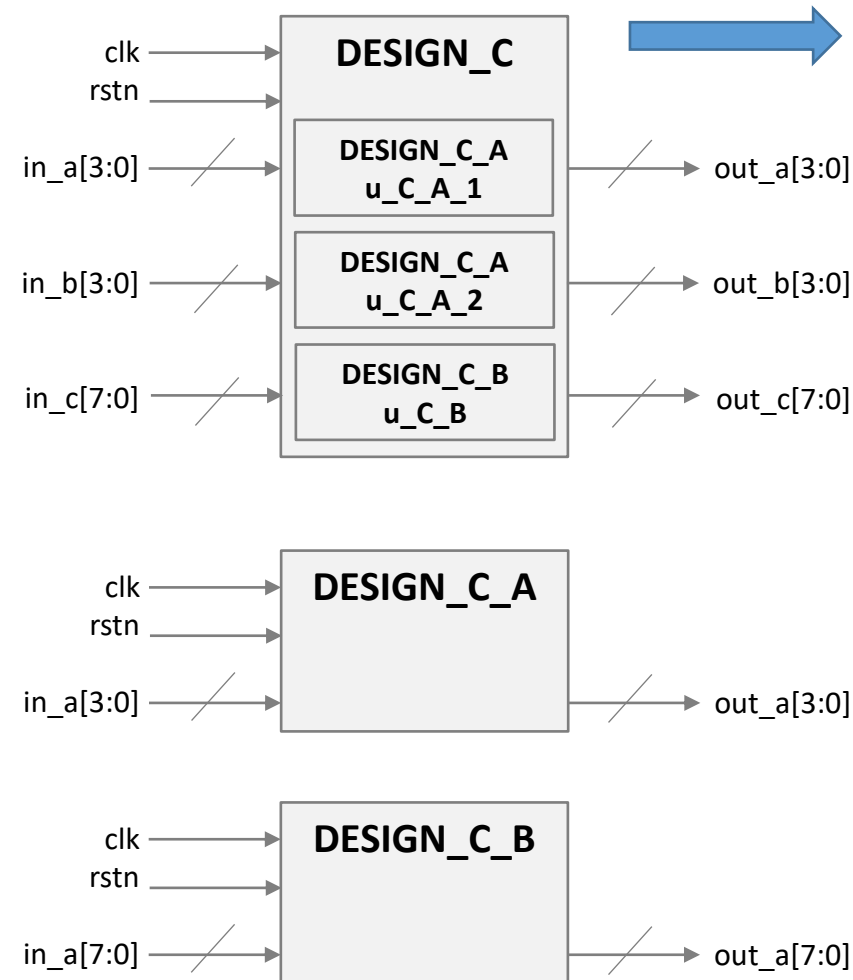
```
module DESIGN_B (  
    input  clk,  
    input  rstn,  
    input [3:0] in_a,  
    input [7:0] in_b,  
    output [3:0] out_a,  
    output [7:0] out_b  
);  
  
    //Code for DESIGN_B  
    //....  
    //....  
  
endmodule
```

Instantiation



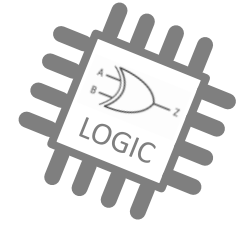
- Instantiation
 - Call sub-modules below module
 - Relationship (hierarchy) between modules

```
module DESIGN_C_A (  
    input    clk,  
    input    rstn,  
    input [3:0] in,  
    output [3:0] out  
);  
    //Code for DESIGN_C_A  
  
endmodule  
  
module DESIGN_C_B (  
    input    clk,  
    input    rstn,  
    input [7:0] in,  
    output [7:0] out  
);  
    //Code for DESIGN_C_B  
  
endmodule
```

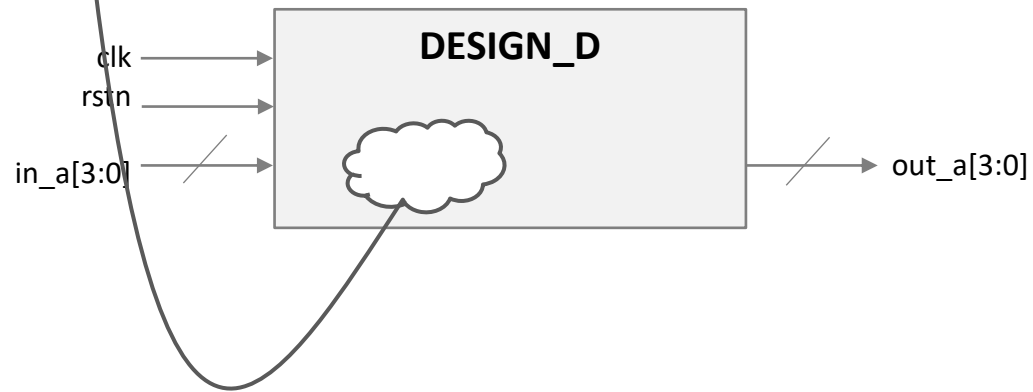


```
module DESIGN_C (  
    input    clk,  
    input    rstn,  
  
    input [3:0] in_a,  
    input [3:0] in_b,  
    input [7:0] in_c,  
  
    output [3:0] out_a,  
    output [7:0] out_b,  
    output [7:0] out_c  
);  
    //Instantiation for DESIGN_C  
    DESIGN_C_A u_C_A_1 (clk, rstn, in_a, out_a);  
  
    DESIGN_C_A u_C_A_2 (  
        .clk    (clk    ),  
        .rstn   (rstn   ),  
        .in     (in_b   ),  
        .out    (out_b  )  
    );  
  
    DESIGN_C_B u_C_B (  
        .clk    (clk    ),  
        .rstn   (rstn   ),  
        .in     (in_c   ),  
        .out    (out_c  )  
    );  
  
    //Code for DESIGN_C  
    //....  
    //....  
  
endmodule
```

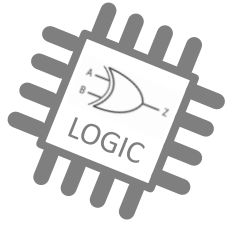
Module Design



- Module design with Data type
 - out is Flip-Flop's output with +1



PARAMETER

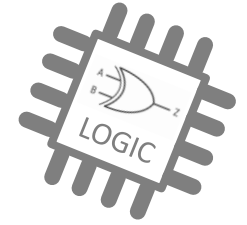


- What is parameter
 - Bit width for vector or array can be various depending on situation
 - for same function with different bit width, it's too inefficient to make lots of modules for each bit width
 - We can use parameters for variant number. We don't need to fix bit width which is variant
 - parameterized design is very recommended in field because it is very good to reuse

```
module PARAMETER_EX1(  
    input    clk,  
    input    rstn,  
    input [3:0] in,  
    output [3:0] out  
);  
    parameter N=4;  
  
    wire [N-1:0] a;  
    reg  [N-1:0] out;  
  
    assign      a = in + 1;  
  
    always @(posedge clk, negedge rstn)  
        if (!rstn) out <= 0;  
        else      out <= a;  
  
endmodule
```

```
module PARAMETER_EX2 #(  
    parameter N=4 //,  
) (  
    input    clk,  
    input    rstn,  
    input [N-1:0] in,  
    output [N-1:0] out  
);  
  
    wire [N-1:0] a;  
    reg  [N-1:0] out;  
  
    assign      a = in + 1;  
  
    always @(posedge clk, negedge rstn)  
        if (!rstn) out <= 0;  
        else      out <= a;  
  
endmodule
```

PARAMETER



- parameter override
 - Several bit width should be used at multiple instance in hierarchy
 - We can control parameter's value in upper hierarchy
 - defparam, parameter argument are the way
- localparam
 - This acts like parameter in a module but can't be overridden by upper hierarchy

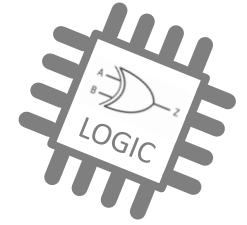
```
module PARAMETER_EX3 #(
    parameter N=4,
    parameter M=8
) (
    input clk,
    input rstn,

    input [N-1:0] in_1,
    input [M-1:0] in_2,
    output [N-1:0] out_1,
    output [M-1:0] out_2,
);
    defparam u_EX3_1.N = N; //parameter 4
    PARAMETER_EX3_1 u_EX3_1 (in_a, out_a);

    PARAMETER_EX3_1 #(.N(M)) u_EX3_2 (in_a, out_a); //parameter 8
endmodule
```

```
module PARAMETER_EX3_1 #(
    parameter N=8
) (
    input [N-1:0] in,
    output [N-1:0] out
);
    localparam K=32;
    //Code for PARAMETER_EX3_1
endmodule
```


DEFINE



- Usage

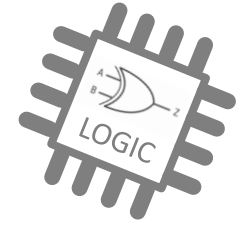
- This role is similar with language C
- Sometimes, several lines should be compiled or not depending on situation
define is frequently used with ifdef/else for covering this case
- *``define`, ``ifdef`, ``ifndef`, ``else`* in Verilog
- Do not use too much because it can make worse readability

```
`define SEQ_LOGIC
module DEFINE_EX #(
    parameter    N=8
) (
    `ifdef SEQ_LOGIC
        input        clk,
        input        rstn,
    `endif
        input [N-1:0] in,
        output[N-1:0] out
    );

    `ifndef SEQ_LOGIC
        assign out = in;
    `else
        reg [N-1:0] out;

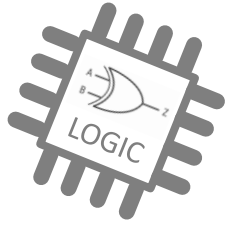
        always @(posedge clk, negedge rstn)
            if (!rstn) out <= 0;
            else      out <= in;
    `endif
endmodule
```

Operator



- Arithmetic
 - `a+b; a-b; a*b; a/b; a%b; a**b;`
- Relational
 - `a<b; a>b; a<=b; a>=b;`
- Equality
 - `a===b; a!==b; a==b; a!=b;`
- Logical
 - `a && b; a || b; !a;`
- Bitwise
 - `~a; a & b; a | b; a ^ b;`
- Unary
 - `&a; |a; ^a;`
- Shift
 - `a >> 3; a << 3; a >>> 3; a <<< 3;`

Operator



- **Concatenation**

- $a = \{b, c\};$



- $\{b, c\} = a;$

- Replication
 $a = \{2\{b\}, c\};$

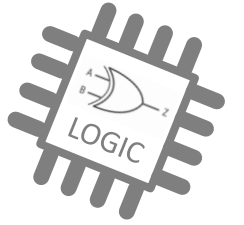


- Bit width of each variables are very important

- **Conditional**

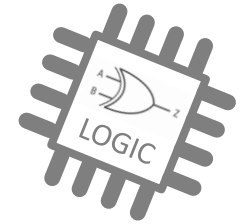
- $a = b ? c : d;$

Modeling

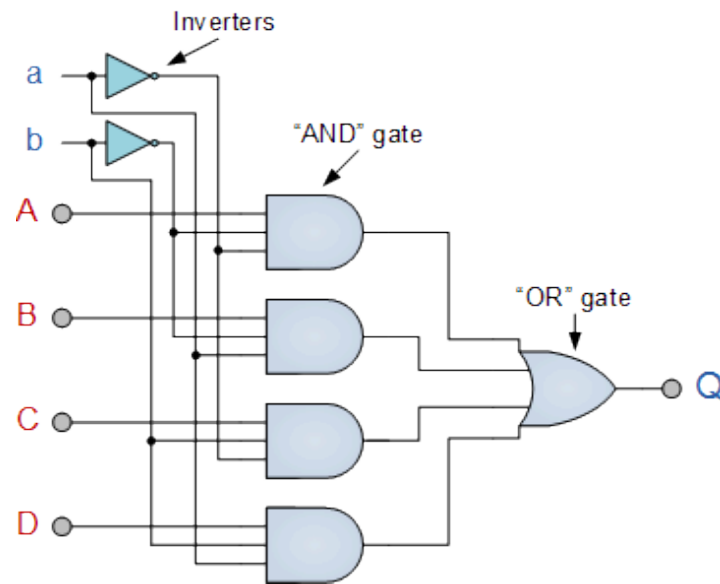


- 3 types of Modeling
 - Gate-level (Not much usage in real coding)
 - Dataflow
 - Behavioral

Gate Level Modeling

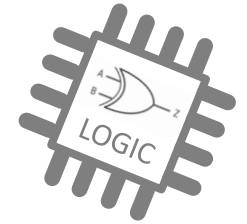


- Gate Level Modeling
 - Verilog can provide gate primitive cell which can be instantiated
 - Describe each gate with each instance including connection

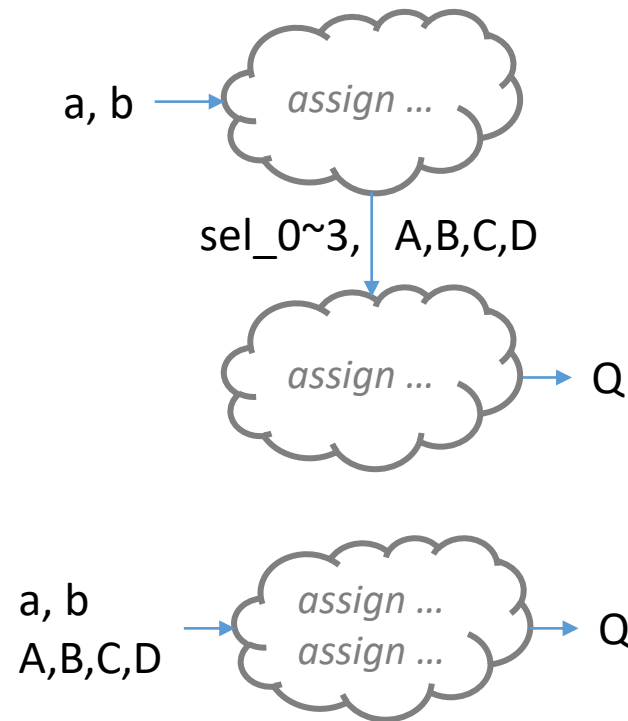
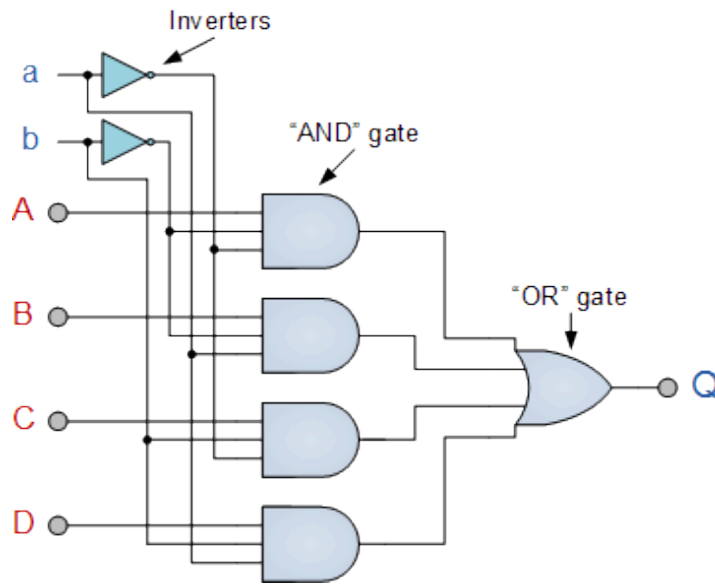


```
module mux_gatelevel(  
    input  a, b,  
    input  A, B, C, D,  
    output Q  
);  
  
    wire  a_not, b_not;  
    wire  A_pick, B_pick, C_pick, D_pick;  
  
    not(a_not , a);  
    not(b_not , b);  
  
    and(A_pick, a_not, b_not, A);  
    and(B_pick, a      , b_not, B);  
    and(C_pick, a_not, b      , C);  
    and(D_pick, a      , b      , D);  
  
    or (Q, A_pick, B_pick, C_pick, D_pick);  
  
endmodule
```

Data Flow Modeling

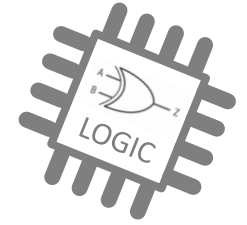


- Data Flow Modeling
 - assign statement is used with operators which is more effective to describe than gates
 - Between assign statements, there is no sequence but dependency
 - We can know how data flows in this modeling (which variables follow which operations)

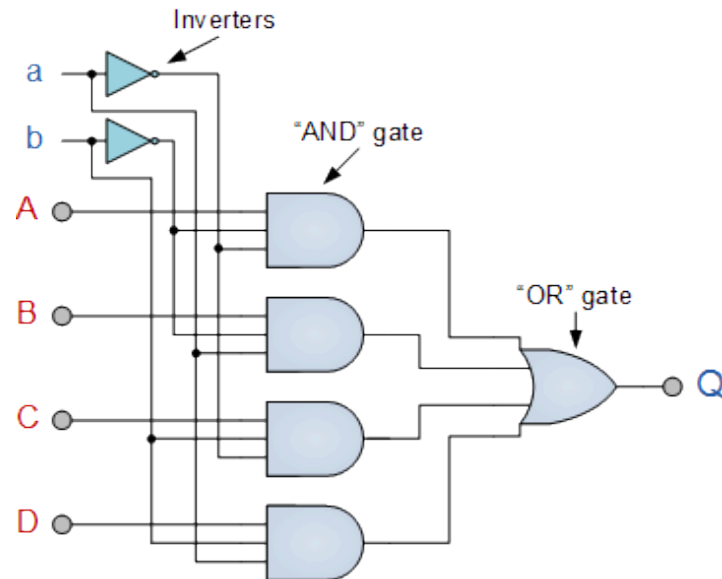


```
module mux_dataflow(  
    input  a, b,  
    input  A, B, C, D,  
    output Q  
);  
    wire  sel_0, sel_1, sel_2, sel_3;  
  
    assign sel_0 = ~a & ~b;  
    assign sel_1 = a & ~b;  
    assign sel_2 = ~a & b;  
    assign sel_3 = a & b;  
  
    assign Q = sel_0 ? A :  
               sel_1 ? B :  
               sel_2 ? C :  
               sel_3 ? D : 0;  
  
    /*  
    assign Q = (~a & ~b) ? A :  
               ( a & ~b) ? B :  
               (~a & b) ? C :  
               ( a & b) ? D : 0;  
    */  
endmodule
```

Behavioral Modeling

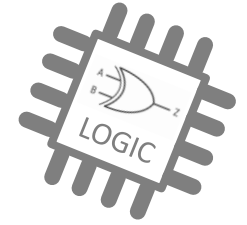


- Behavioral Modeling
 - Describe behavior in procedure blocks
 - This is usually in “*always*” block
 - This is normal way to implement sequential logic
We can not save the value in variables with dataflow modeling (except tri-reg)
 - Let's deep-dive into behavioral modeling after practice

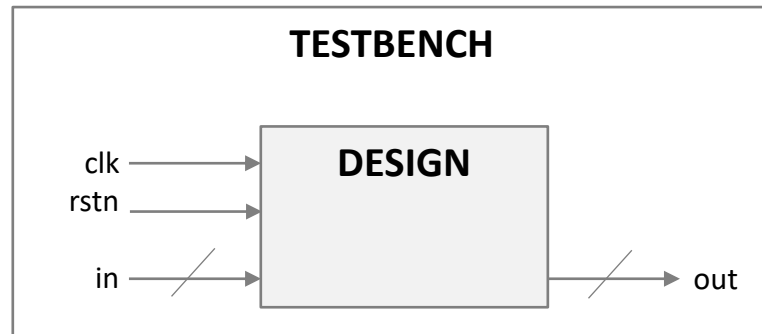


```
module mux_behavior(  
    input  a, b,  
    input  A, B, C, D,  
    output Q  
);  
    reg    Q;  
  
    always @* begin  
        case({b,a})  
            'b00 : Q=A;  
            'b01 : Q=B;  
            'b10 : Q=C;  
            'b11 : Q=D;  
            default : Q=0;  
        endcase  
    end  
endmodule
```

cf) Stimulus



- Stimulus
 - We have only logic which is not triggered
 - If there is no value on input port, we can't see anything
 - to simulate, we need a stimulus logic called *testbench* (we will focus on next chapter)
- Let's try on EDA playground



```
module mux_testbench;
    reg [3:0] a, b;
    reg [3:0] A, B, C, D;
    wire [3:0] Q;

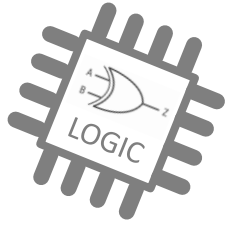
    mux_gatelevel u_mux_gatelevel(a[0], b[0], A[0], B[0], C[0], D[0], Q[0]);
    mux_dataflow u_mux_dataflow(a[1], b[1], A[1], B[1], C[1], D[1], Q[1]);
    mux_behavior u_mux_behavior(a[2], b[2], A[2], B[2], C[2], D[2], Q[2]);

    initial begin
        A='0; B='1; C='1; D='0;

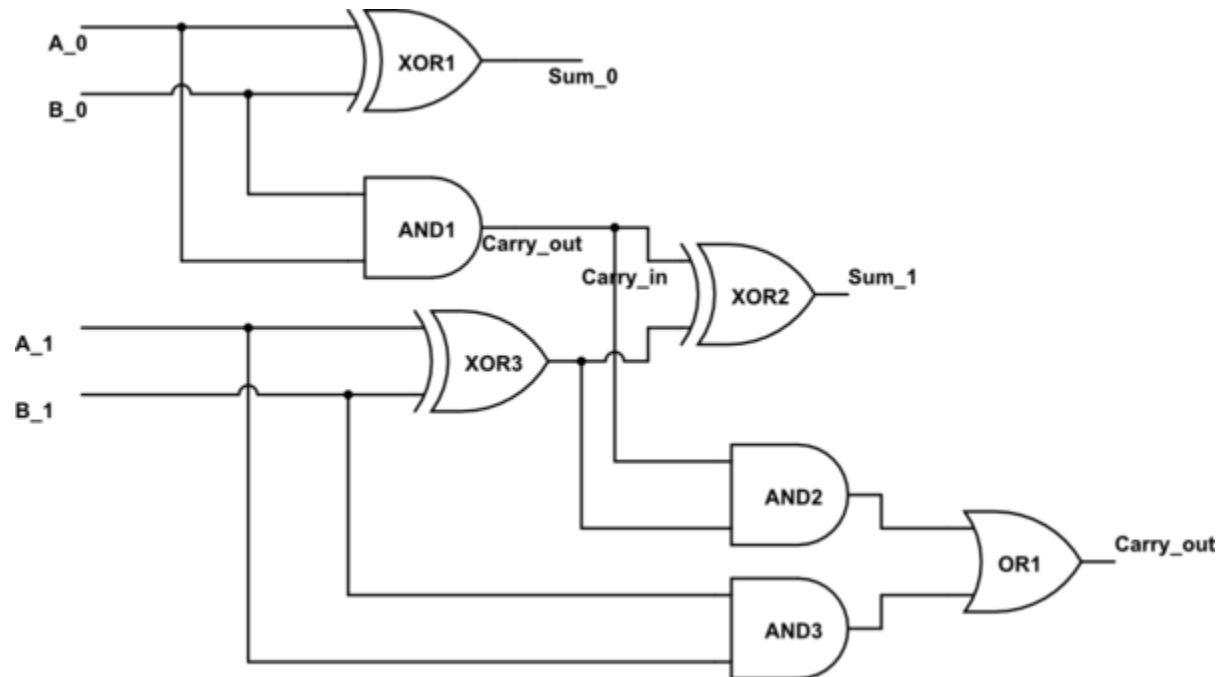
        a='0; b='0;
        #10 a='1; b='0;
        #10 a='0; b='1;
        #10 a='1; b='1;
    end

    initial begin
        $monitor("gatelevel : a %d, b %d, Q %d",a[0],b[0],Q[0]);
        $monitor("dataflow : a %d, b %d, Q %d",a[1],b[1],Q[1]);
        $monitor("behavior : a %d, b %d, Q %d",a[2],b[2],Q[2]);
    end
endmodule
```

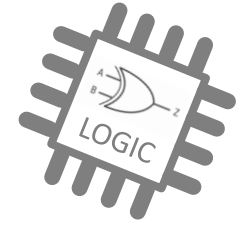

MODELING PRACTICE



- Describe and simulate *2-bit full adder* in 3 modeling ways



Behavior Modeling



- Procedure block
 - *initial* : execute only once at time zero
 - *always* : execute over and over, with condition defined at @

```
initial begin
    a = 0;
    a = ~a;
end

always @(posedge clk, negedge rstn)
if (!rstn) a <= 0;
else      a <= ~a;
```

- *begin-end* : sequential executes
- *fork-join* : parallel executes

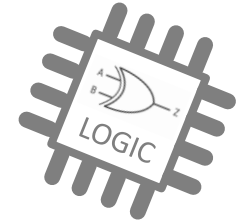
- net type can not be used in procedure block
- Every procedure block executes concurrent.

```
initial begin
    a=0; b=0; c=0;

    #10 a=1;    //time 10: a=1
    #20 b=1;    //time 30: b=1
    #30 c=1;    //time 60: c=1
end

initial begin
    a=0; b=0; c=0;
    fork
        #10 a=1;    //time 10: a=1
        #20 b=1;    //time 20: b=1
        #30 c=1;    //time 30: c=1
    join
end
```

Behavior Modeling



- Blocking assignment

- executes in order, sequential
- previous line blocks next line.
it is why they are called blocking
- $a=b;$

```
initial begin
    #10 a=0;
    #20 a=1;
end

initial begin
    #10 b<=0;
    #20 b<=1;
end

initial begin
    c = #10 0;
    c = #20 1;
end

initial begin
    d <= #10 0;
    d <= #20 1;
end
```

- Non-Blocking assignment

- executes in parallel
- previous line doesn't block next line.
it is why they are called non-blocking
- $a<=b;$
- *Do not mix blocking & non-blocking in a procedure*

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

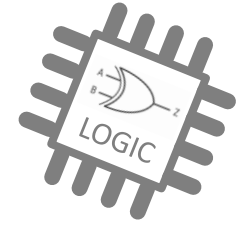
initial begin
    c=0; d=1;
    #11
    c = d;
    d = c;
end
```

```
initial begin
    e=0; f=1;
    #12
    e <= f;
    f = e;
end

initial begin
    g=0; h=1;
    #13
    g = h;
    h <= g;
end

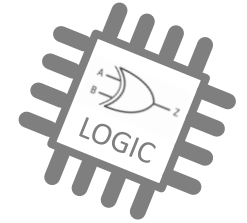
initial begin
    h=0; i=1;
    #14
    h <= i;
    #1
    i <= h;
end
```

Behavior Modeling



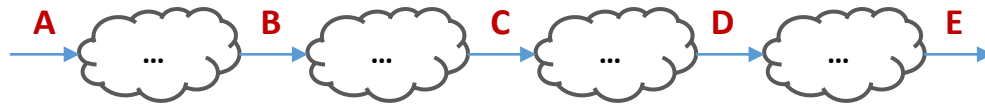
- *always @(sensitivity list)*
 - always block executes when sensitivity list events
- *always @(a, b)* means this procedure executes when a or b changes
- *always @(*)* : * means any signal event who affects output of this procedure
- *always @(posedge clk)* means this procedure excutes when clk is rising
- *always @(negedge rstn)* means this procedure excutes when rstn is falling
- *always @* , always @(posedge clk, negedge rstn)* are most common usage

Behavior Modeling



- *always @**

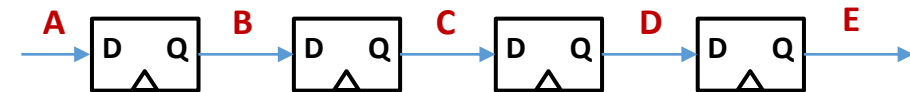
- This procedure executes when input signal of this block changes
- *It's like combinational logic !!*



- *Blocking assignment is highly recommended*
Because each signal should be followed by next one

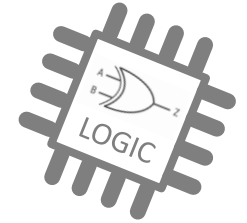
- *always @(posedge clk, negedge rstn)*

- This procedure executes when clock is rising (or rstn is falling)
- *It's like sequential logic (FF) !!*



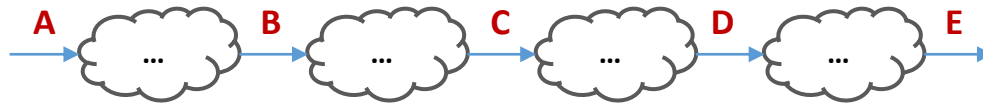
- *Non-blocking assignment is highly recommended*
Because "Q" should be affected by current "D", not follow previous signal

Behavior Modeling



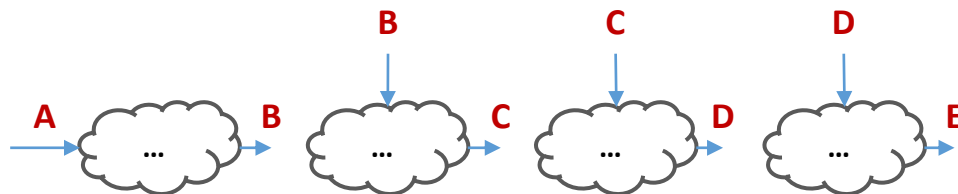
- *always @**
 - Non-blocking assignment makes unexpected results (Wrong implementation)
 - The connection can be cut off with non-blocking assignment
 - **Avoid Non-blocking assignment in “always @(*)”**

GOOD (blocking)



```
initial #10 a = 1;  
always @(*) begin  
    b = a+1;  
    c = b+1;  
    d = c+1;  
    e = d+1;  
end
```

```
initial #10 a = 1;  
always @(a) begin  
    b = a+1;  
    c = b+1;  
    d = c+1;  
    e = d+1;  
end
```

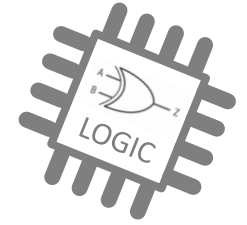


```
initial #10 a = 1;  
always @(*) begin  
    b <= a+1;  
    c <= b+1;  
    d <= c+1;  
    e <= d+1;  
end
```

```
initial #10 a = 1;  
always @(a) begin  
    b <= a+1;  
    c <= b+1;  
    d <= c+1;  
    e <= d+1;  
end
```

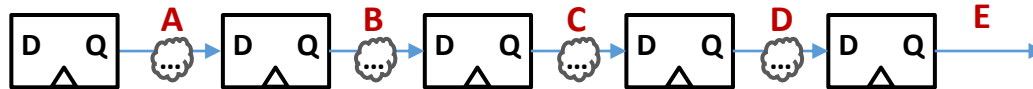
BAD (non-blocking)

Behavior Modeling



- *always @(posedge clk, negedge rstn)*
 - Blocking assignment makes that output is affected by the value being calculated, not the current state of input (what FF does)
 - Blocking assignment makes absence of Flip Flop (Wrong implementation)
 - **Avoid Blocking assignment in “always @(posedge clk)”**

GOOD (non-blocking)



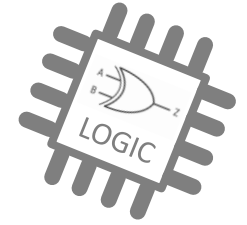
```
always @(posedge clk, negedge rstn)
if (!rstn) begin
    {a,b,c,d,e} <= 0;
end else begin
    a <= 1;
    b <= a+1;
    c <= b+1;
    d <= c+1;
    e <= d+1;
end
```

BAD (blocking)



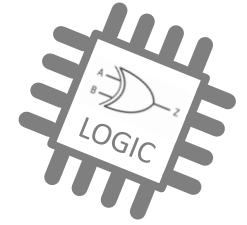
```
always @(posedge clk, negedge rstn)
if (!rstn) begin
    {a,b,c,d,e} = 0;
end else begin
    a = 1;
    b = a+1;
    c = b+1;
    d = c+1;
    e = d+1;
end
```

Conditional statement



- if-else
 - Role is as same as C language
 - begin-end grammar instead of { ... }
- ```
if (sel==0) begin
 Q=A;
end else if (sel==1) begin
 Q=B;
end else if (sel==2) begin
 Q=C;
end else begin
 Q=D;
end
```
- case
  - Role is as same as C language's switch-case
  - case-endcase grammar instead of switch{ ... }
- ```
case (sel)
    2'b00:    Q=A;        //begin-end for over 2 lines
    2'b01:    Q=B;
    2'b10:    Q=C;
    default:  Q=D;
endcase
```
- when there are lots of selections,
case statement is preferred to if-else statement
- Case will be translated to **N:1 Mux**
while if-else will be translated to N-1's **2:1 Mux**

Looping statement



- ***for***

- Same as C language
- grammar : `for (...) begin ... end`
- `for (i=0;i<8;i++) begin`
 `a[i] <= 0;`
 `end`
- *Only this Loop is normally used in Design (for reset initialization)*

- ***while***

- Same as C language
- grammar : `while (...) begin ... end`
- `while (ready == 0) begin`
 `valid <= 1;`
 `end`

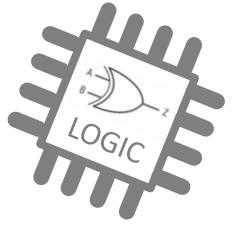
- ***forever***

- This loop never ends and executes continually
- It should be used with timing construct
- Normally, this is used for clock generation in TB
- `initial begin`
 `clk = 0;`
 `forever #5 clk = ~clk;`
 `end`

- ***repeat***

- Executes fixed number of times
- `initial begin`
 `x=0;`
 `repeat (10) #5 x++;`
 `end`

cf) CLK, RST

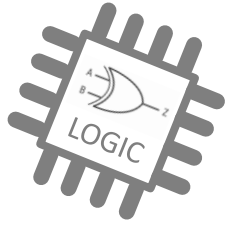


- Design who implements Sequential Logic needs clock and reset from TestBench
- In this case, TestBench should have not only signal stimulus but also clock & reset generator
- Clock is usually generated by forever statements. Reset is usually generated in initial statements

- *initial begin*
 clk = 0;
 forever #5 clk = ~clk;
 end

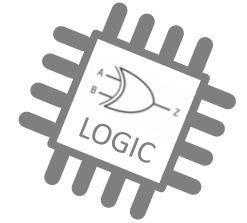
- initial begin*
 rstn = 1;
 #10 rstn = 0;
 #20 rstn = 1;
 end

PRACTICE



- 1. Design and simulate 4-bit counter
- 2. Design and simulate a clock who marks hour and minute when 1 cycle is corresponding to 1 minute

Contents



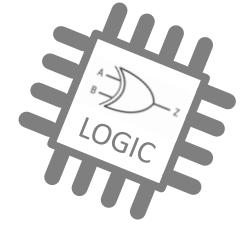
Explanation

- Introduction to Semiconductor field
- Basic concept of Verilog
- Combinational Logic
- Sequential Logic & Clock/Reset
- **FSM**
- ~~FSM~~

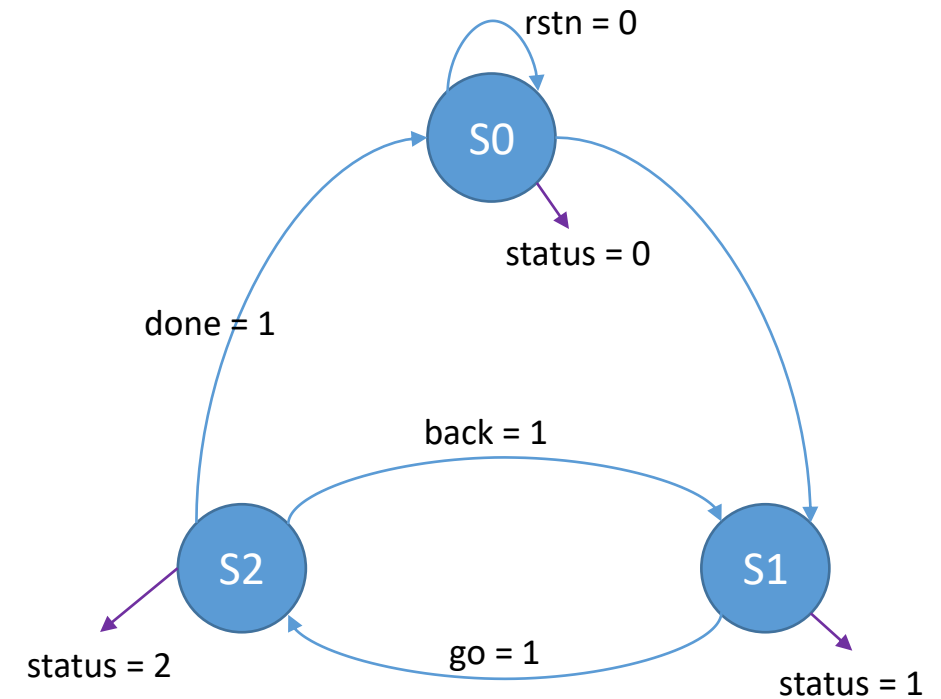
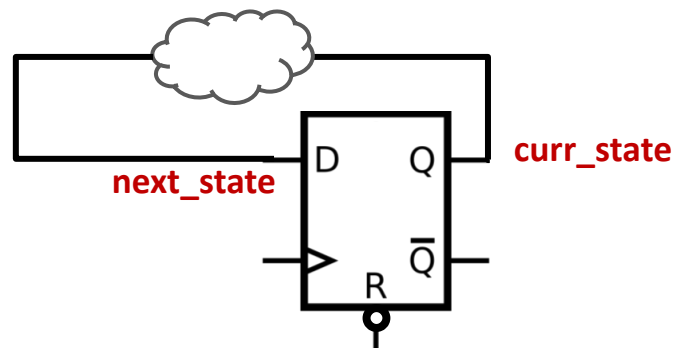
Coding

- Free Tools
- Declarations & Modeling
- ~~DUT & Testbench~~
- DUT & Testbench
- Task & Function
- Lots of Practices

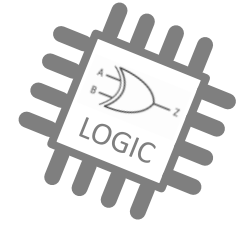
What is FSM



- Finite State Machine
 - for Sequential Logic, output depends on not only input but also current stored information
 - Current stored information is called states,
When system has finite number of states, we can call it **F**inite **S**tate **M**achine
 - Useful way to control and check the progressing status of Design
 - System behaves differently depending on state
 - Each state can go to another state with certain condition
Each state can have its own output
 - State Machine is used for deciding other signal's behavior

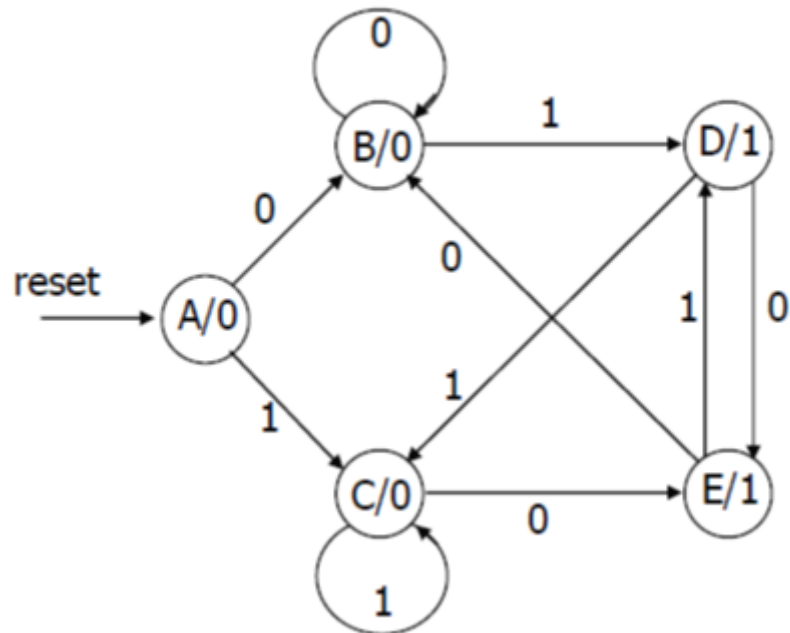


Moore vs Mealy



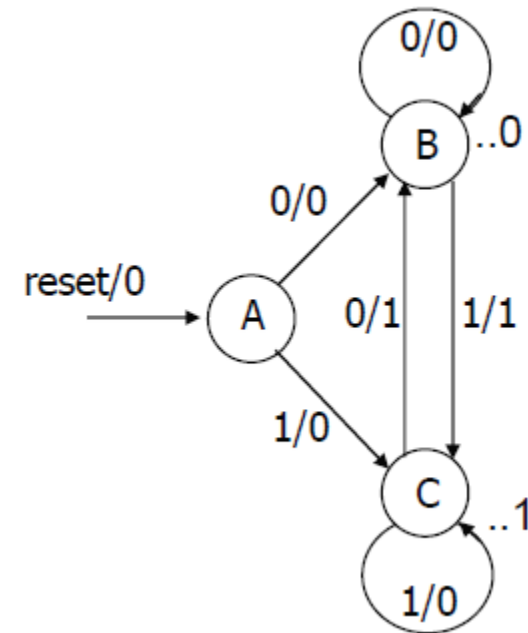
- Moore machine

- output depends on only current state
- output will change at only clock edge
- safer than mealy machine because there is no asynchronous issue

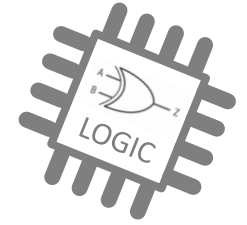


- Mealy machine

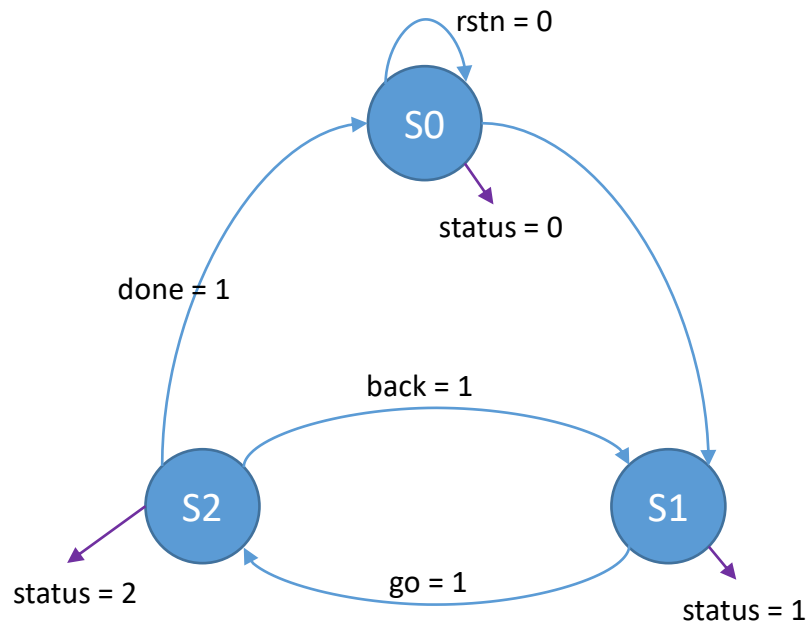
- output depends on both current state and input
- react faster than moore machine because it doesn't have to wait for 1 clock
- can have less state than moore machine



FSM Coding



- Coding style
 - next_state is implemented by case statements, combinational logic
 - curr_state is F/F output of next_state
 - Above 2 things can be combined into one procedure block
 - localparam (parameter) is used for state encoding



```
localparam [1:0] S0=0,S1=1,S2=2;

always @(posedge clk, negedge rstn)
if (!rstn) curr_state <= 0;
else      curr_state <= next_state;

always @* begin
    next_state = curr_state;
    case(curr_state)
        S0:
        S1: if (go==1) next_state = S2;
        S2: if (back==1) next_state = S1;
            else if (done==1) next_state = S0;
    endcase
end

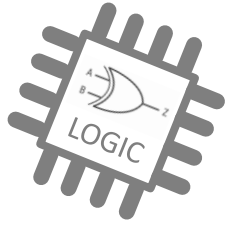
always @* begin
    status = 0;
    case (curr_state)
        S0: status = 0;
        S1: status = 1;
        S2: status = 2;
    end
end
```

```
localparam [1:0] S0=0,S1=1,S2=2;

always @(posedge clk, negedge rstn)
if (!rstn) curr_state <= 0;
else
    case(curr_state)
        S0: curr_state <= S1;
        S1: if (go==1) curr_state <= S2;
        S2: if (back==1) curr_state <= S1;
            else if (done==1) curr_state <= S0;
    endcase
end

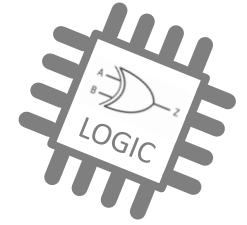
always @* begin
    status = 0;
    case (curr_state)
        S0: status = 0;
        S1: status = 1;
        S2: status = 2;
    endcase
end
```

PRACTICE

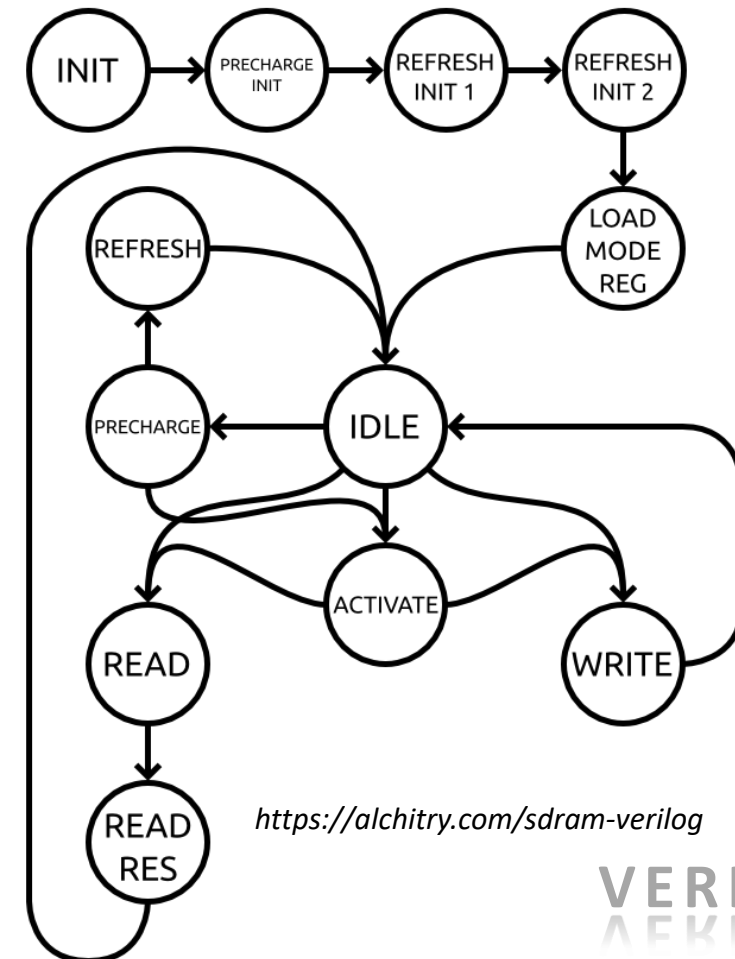


- Implement traffic light with FSM
 - Red will go to Yellow after 30 cycles
 - Yellow will go to Green after 5 cycles
 - Green will go to Red after 20 cycles
 - The light color will be output

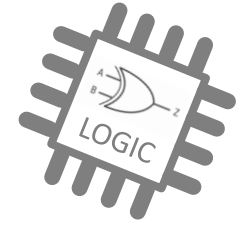
FSM usage



- Usage in field
 - FSM is normally used for controlling behavior from a large perspective
Ex) Initialization step of system, whole progress of device
 - Because this is used for controlling lots of signals,
The FSM should be implemented very well



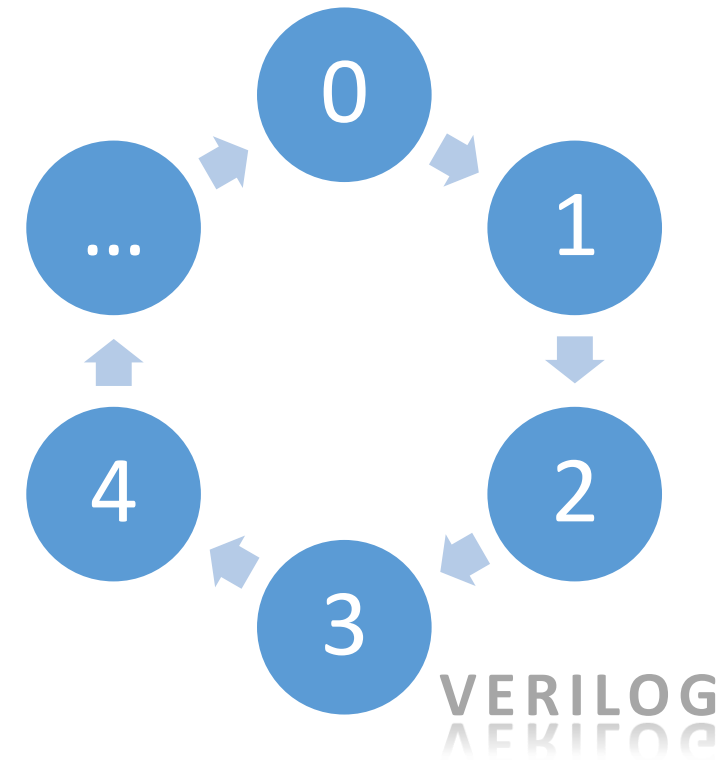
FSM usage



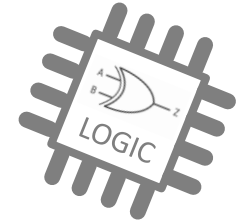
- When not to use
 - We don't have to use FSM when the state can be implemented very simple like "one loop", "zero or one control input"
 - Counter is one example. This is free running without any input
It can be also implemented by FSM but not effective

```
always @(posedge clk, negedge rstn)
if (!rstn) cnt <= 0;
else      cnt <= cnt + 1;
```

```
always @(posedge clk, negedge rstn)
if (!rstn) cnt <= 0;
else
  case(cnt)
    0: cnt <= 1;
    1: cnt <= 2;
    2: cnt <= 3;
    .....
    default: cnt <= 0;
  endcase
```



Contents



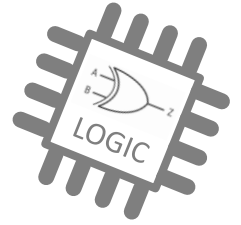
Explanation

- Introduction to Semiconductor field
- Basic concept of Verilog
- Combinational Logic
- Sequential Logic & Clock/Reset
- FSM

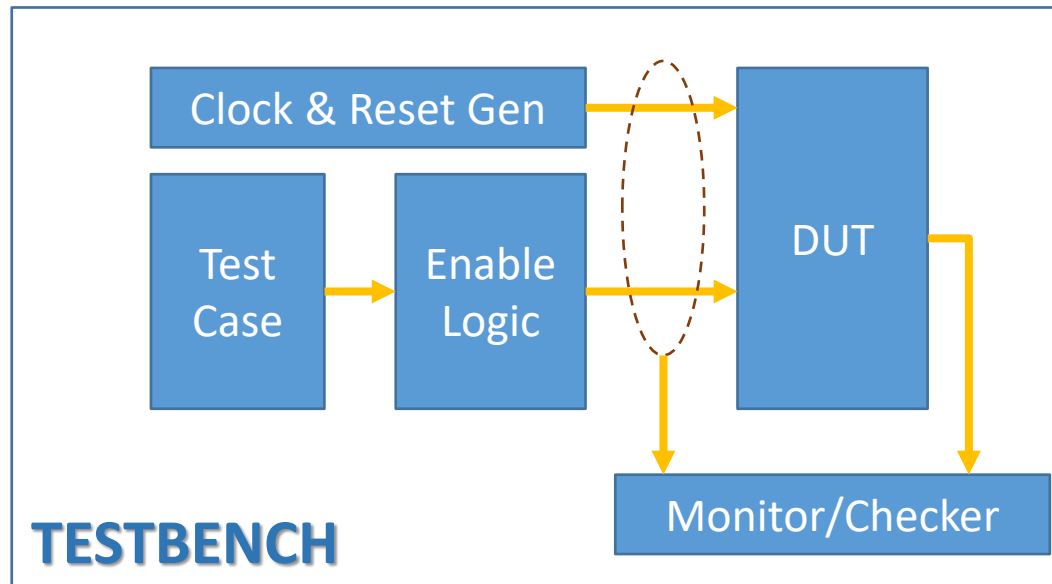
Coding

- Free Tools
- Declarations & Modeling
- **DUT & Testbench**
- Task & Function
- Lots of Practices

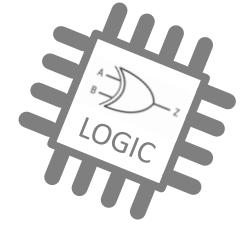
TB Environment



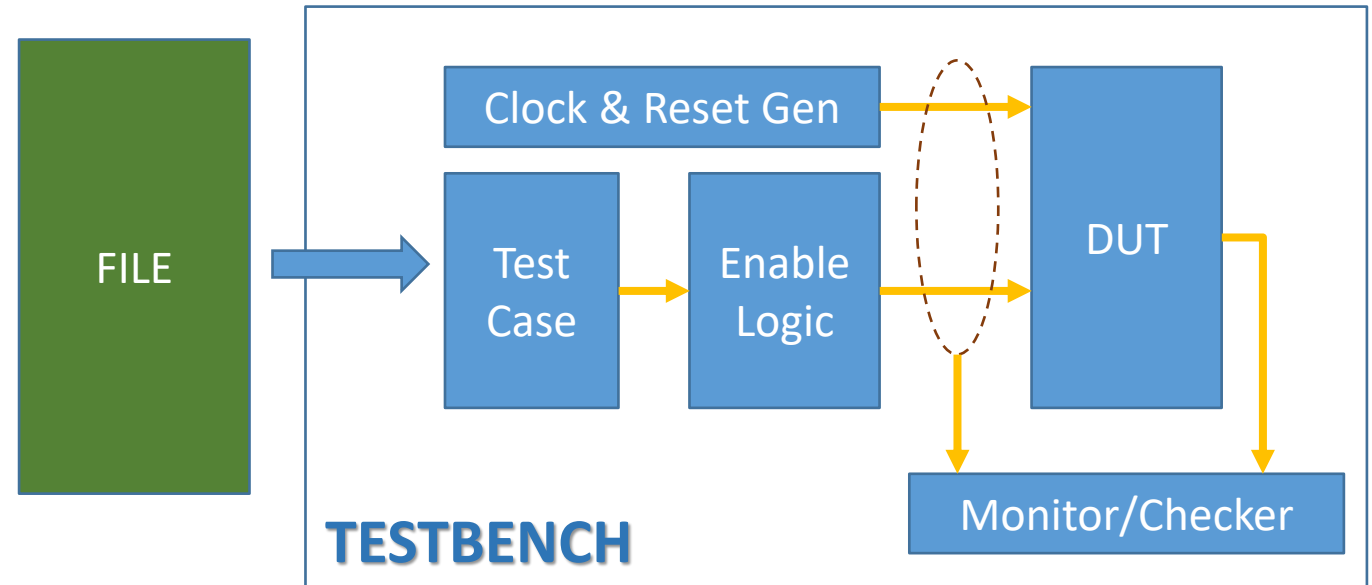
- DUT
 - Design Under Test
 - The grammar is more strict because EDA tool can synthesize this part
- TestBench (TB)
 - Who provides an environment for verifying DUT
 - It's as important as DUT
It's as complex as DUT
 - We need more test case for larger design
 - Nowadays, System-Verilog is recommended for TB



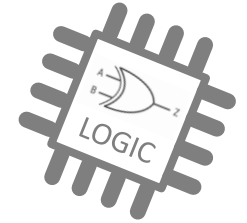
File i/o



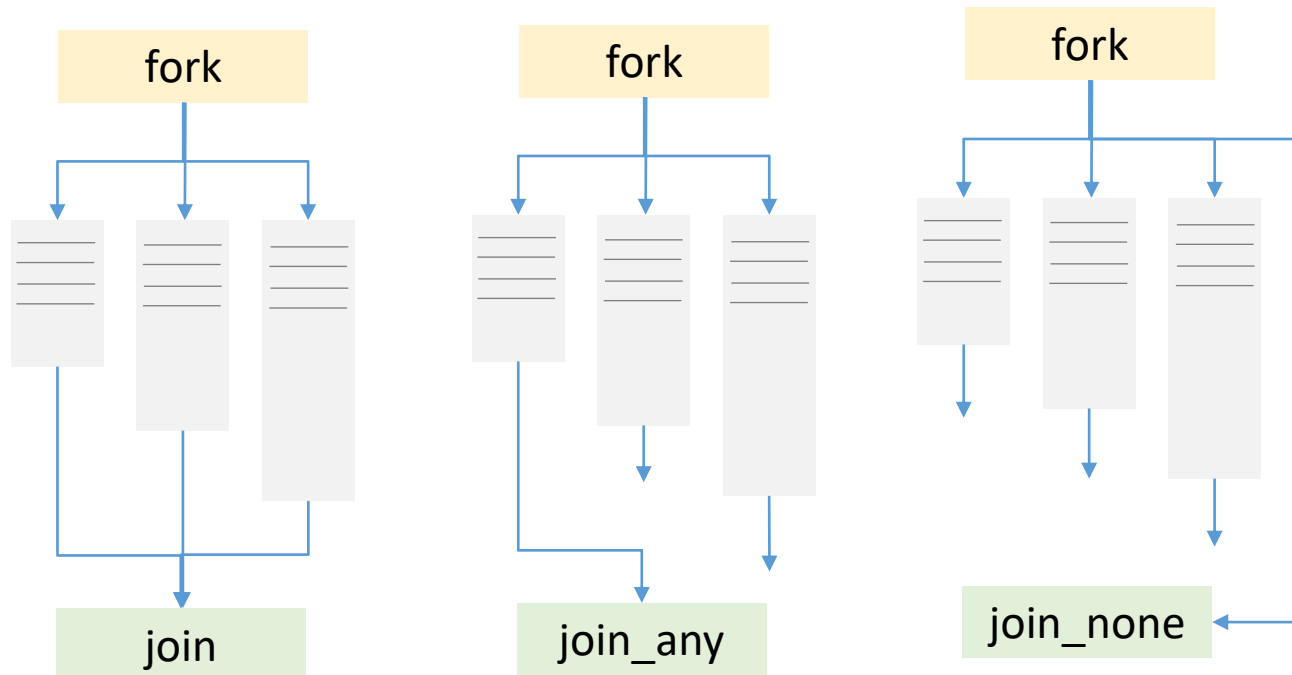
- `$fopen`
 - binary type is supported
 - ```
int fin, fout;
fin = $fopen("file_in.txt","r");
fout = $fopen("file_out.txt","w");
```
- `$fscanf`
  - ```
reg [7:0]  num, data;  
$fscanf(fin, "%d %h", num, data);
```
- `$fdisplay`
 - ```
$fdisplay(fout, "%d %h", num, data);
```
- `$fclose`
  - ```
$fclose(fin);  
$fclose(fout);
```



fork-join



- fork-join
 - There are 3 ways to implement fork statement
 - all the parallel statements finish before *join*
 - any of the parallel statements finishes before *join_any*
 - none of the parallel statements need to finish before *join_none*

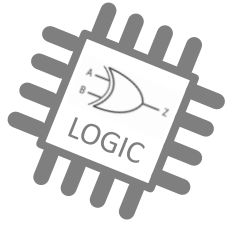


```
initial begin
    $display("before fork");
    fork
        #10 $display("parallel 1");
        #20 $display("parallel 2");
        #30 $display("parallel 3");
    join
    #40 $display("after join");    //70ns
end

initial begin
    $display("before fork");
    fork
        #10 $display("parallel 1");
        #20 $display("parallel 2");
        #30 $display("parallel 3");
    join_any
    #40 $display("after join");    //50ns
end

initial begin
    $display("before fork");
    fork
        #10 $display("parallel 1");
        #20 $display("parallel 2");
        #30 $display("parallel 3");
    join_none
    #40 $display("after join");    //40ns
end
```

Event, Wait



- event
 - This is another variable type for testbench
 - event happens after fixed time
 - User can detect when event happens
- we can use posedge, negedge instead of event
- wait
 - Testbench can wait for a signal until when it gets certain value

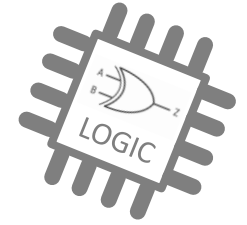
```
initial begin
    #20 -> event_a;
    $display("%3t: event_a", $time);
end

initial begin
    $display("%3t: before event_a", $time);
    @(event_a);
    $display("%3t: after event_a", $time);
end
```

```
initial begin
    @(negedge rstn);
    repeat(5) @(posedge clk);
    $display("%3t: after rstn, after 5 clk", $time);
end
```

```
initial begin
    wait(a==1);
    $display("a gets value 1");
end
```

Force, Release



- force
 - We can force a certain value regardless of existing operation
 - The other signals which depends on forced signal are also affected
 - We can use hierarchical reference
- release
 - User can release forced signal
 - The signal will be affected by existing operation after release

```
module tb_counter;
    reg clk, rstn;

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        rstn = 1;
        #10 rstn = 0;
        #20 rstn = 1;
    end

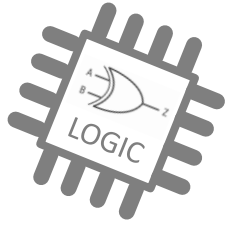
    initial begin
        #50 force    u_counter.en = 0;
        #50 release u_counter.en;
        #50 $finish;
    end

    counter u_counter(clk, rstn, 1);
endmodule

module counter(
    input clk, rstn,
    input en
);
    reg [3:0] cnt;

    always @(posedge clk, negedge rstn)
        if (!rstn) cnt <= 0;
        else if (en) cnt <= cnt + 1;
endmodule
```


Etc



- \$random, \$random (seed)
 - Making random value with seed (optional)
- \$readmemh, \$readmemb
 - copy file's txt or data to variables via hex or bin type
 - similar as \$fscanf but much faster and simple for initializing memory
 - we need to match vector size with txt
 - start address, end address can also be defined

```
int rand_num_0, rand_num_1
initial rand_num_0 = $random % 32;
initial rand_num_1 = $random(100) % 32;
```

```
reg [7:0] mem_8bit [7:0];
initial $readmemh("8bit.mem",mem_8bit);

/* 8bit.mem
01 23 45 67
89 ab cd ef
*/

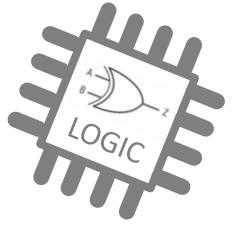
reg [15:0] mem_16bit [3:0];
initial $readmemh("16bit.mem",mem_16bit);

/* 16bit.mem
0123 4567
89ab cdef
*/

reg [3:0] mem_4bit [7:0];
initial $readmemb("4bit.mem",mem_4bit);

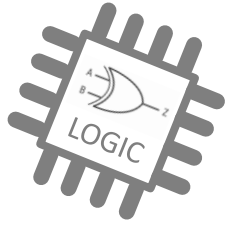
/* 4bit.mem
0000 0001 0010 0011
0100 0101 0110 0111
*/
```

Other technique



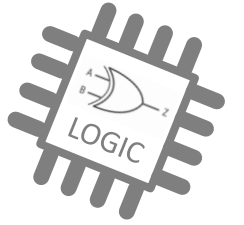
- Advanced skill for Testbench with System-verilog
 - assertion
 - queue
 - associative memory
 - class
- Advanced skill for DUT with System-verilog
 - Interface
 - Struct

Practice



- Design timers and Simulate with 3 different settings
 - They must start after 50ns (*use event*)
 - timer_a counts 10 cycles, timer_b counts 15 cycles, timer_c counts 20 cycles (*use parameter*)
 - Simulate will finish when anyone of 3 timers finish its job (*use fork*)

Contents



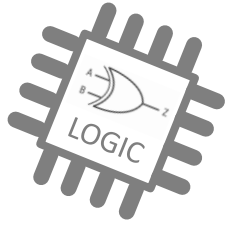
Explanation

- Introduction to Semiconductor field
- Basic concept of Verilog
- Combinational Logic
- Sequential Logic & Clock/Reset
- FSM

Coding

- Free Tools
- Declarations & Modeling
- DUT & Testbench
- **Task & Function**
- Lots of Practices

Task



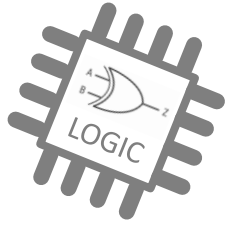
- Task
 - This is subroutines of procedure block. This is usually called in initial block (testbench)
 - Task has its own input/output like module
 - *task ... endtask*
- Task can have a concept of time
- It is convenient when executing cycle-consuming instructions

```
@(posedge clk);  
#10;
```

- we can't detect the change of output at real-time

```
/*  
task my_task;  
    input [3:0] in_a, in_b;  
    output[3:0] out_a, out_b;  
*/  
task my_task(  
    input [3:0] in_a, in_b,  
    output[3:0] out_a, out_b  
);  
begin  
    out_a = in_a + 1;  
    #1 out_b = in_b + 1;  
  
    @(posedge clk);  
    out_a = out_a + 1;  
    out_b = out_b + 1;  
  
    @(posedge clk);  
    out_a = out_a + 1;  
    out_b = out_b + 1;  
end  
endtask
```

Task



- Task
 - Task can call another task
 - task automatic can make each subroutine has its own independent area without automatic, the task is static

```
module tb_display;
    initial #10 my_display();
    initial #10 my_display();
    initial #10 my_display();

    initial #20 my_auto_display();
    initial #20 my_auto_display();
    initial #20 my_auto_display();

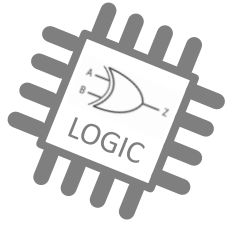
    task my_display;
        int i = 0;
        $display("static integer i=%2d", i++);
        display();
    endtask

    task automatic my_auto_display;
        int i = 0;
        $display("auto   integer i=%2d", i++);
        display();
    endtask

    task display;
        int j = 0;
        $display("sub    integer j=%2d", j++);
    endtask

endmodule
```

Task

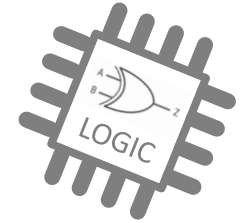


- Task
 - Task can use both blocking and non-blocking assignment
 - Even they look like same, the detailed behavior is a little bit different
 - Be careful when you need to choose one of them

```
task my_task(  
    input [3:0] in_a, in_b,  
    output[3:0] out_a, out_b  
);  
begin  
    out_a = in_a + 1;  
    #1 out_b = in_b + 1;  
  
    @(posedge clk);  
    out_a = out_a + 1;  
    out_b = out_b + 1;  
  
    @(posedge clk);  
    out_a = out_a + 1;  
    out_b = out_b + 1;  
end  
endtask
```

```
task my_task(  
    input [3:0] in_a, in_b,  
    output[3:0] out_a, out_b  
);  
begin  
    out_a <= in_a + 1;  
    #1 out_b <= in_b + 1;  
  
    @(posedge clk);  
    out_a <= out_a + 1;  
    out_b <= out_b + 1;  
  
    @(posedge clk);  
    out_a <= out_a + 1;  
    out_b <= out_b + 1;  
end  
endtask
```

Function



- Function

- This is subroutines of procedure block or data-flow.
- Function has its own input, function name is only output
- *function ... endfunction*

- Function can't have a concept of time
#10 or *@(posedge clk)* will make a compile error
- It is convenient when repeating complex calculation
- automatic is for avoiding static function
- function can call another function

- This can be synthesized in combinational logic which means that blocking assignment is recommended
- It is also used frequently at DUT (task is rarely used at DUT)

```
module tb_function;
    reg [4:0] out_a;
    wire [4:0] out_b;

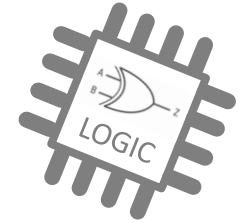
    initial out_a = my_function(1, 2);
    //always @(posedge clk) out_a = my_function(1, 2);
    assign out_b = my_function(1, 2);

    function [4:0] my_function(
        input [3:0] in_a, in_b
    );
        reg [4:0] sum;
        reg [4:0] shift;

        sum = in_a + in_b;
        my_function = sum << 1;

        $display("result is %2d",my_function);
    endfunction
endmodule
```


Contents



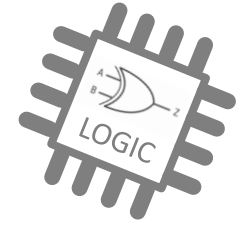
Explanation

- Introduction to Semiconductor field
- Basic concept of Verilog
- Combinational Logic
- Sequential Logic & Clock/Reset
- FSM

Coding

- Free Tools
- Declarations & Modeling
- DUT & Testbench
- Task & Function
- **Lots of Practices**

Common mistakes



- `always @(posedge clk)`
 - Avoid multiple driver

```
always @(posedge clk) begin  
  if (a == 0) a <= 1;  
  if (b == 0) a <= 0;  
end
```



```
always @(posedge clk) begin  
  if (a == 0) a <= 1;  
  else if (b == 0) a <= 0;  
end
```

- `always @*`
 - See if all cases are allocated without exception
 - Define default value at the first line in `always @*` block

```
always @* begin  
  case(a)  
    0: b = 1;  
  endcase  
end
```



```
always @* begin  
  b = 0;  
  case(a)  
    0: b = 1;  
  endcase  
end
```

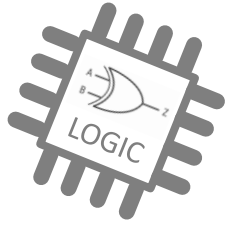
- Combinational Loop
 - Combinational Loop can make simulator hang

```
assign a = a+1;  
always @* b = b+1;
```

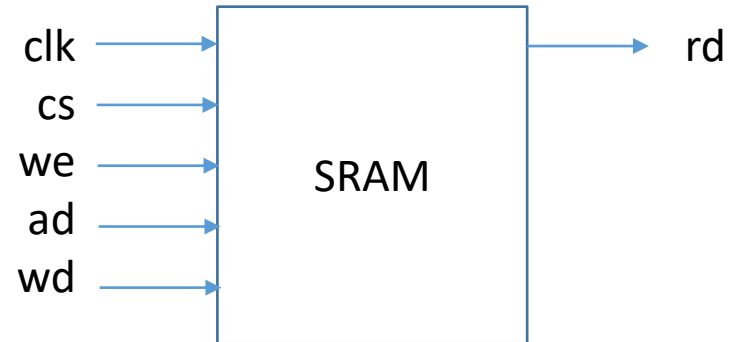


```
always @(posedge clk) a <= a+1;  
always @(posedge clk) b <= b+1;
```

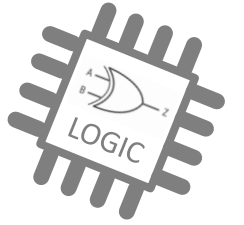
Practice



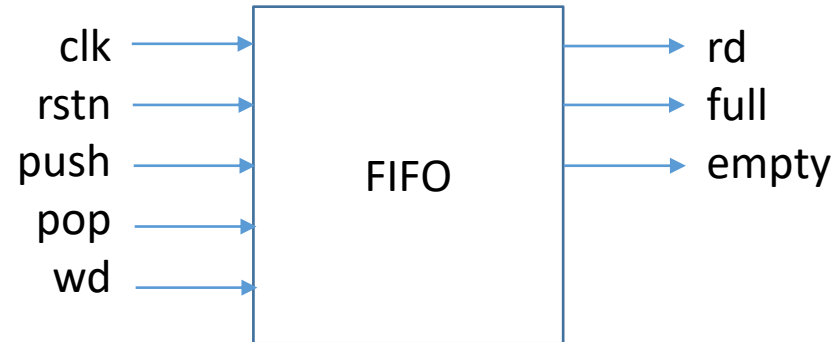
- Design SRAM and simulate



Practice



- Design FIFO and simulate



THANK YOU

THANK YOU