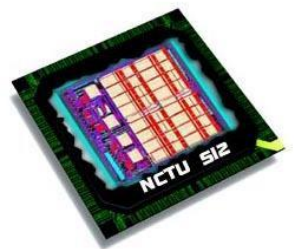


SEQUENTIAL CIRCUITS

NCTU-EE IC LAB FALL-2023



Lecturer: Yen-Teng Chuang

Outline

- ✓ **Section 1 Sequential Circuits**
- ✓ **Section 2 Finite State Machine**
- ✓ **Section 3 Timing**
- ✓ **Section 4 Synthesis and Design Compiler**
- ✓ **Section 5 Generate and for loop**



Outline

✓ Section 1 Sequential Circuits

✓ Introduction

✓ Syntax

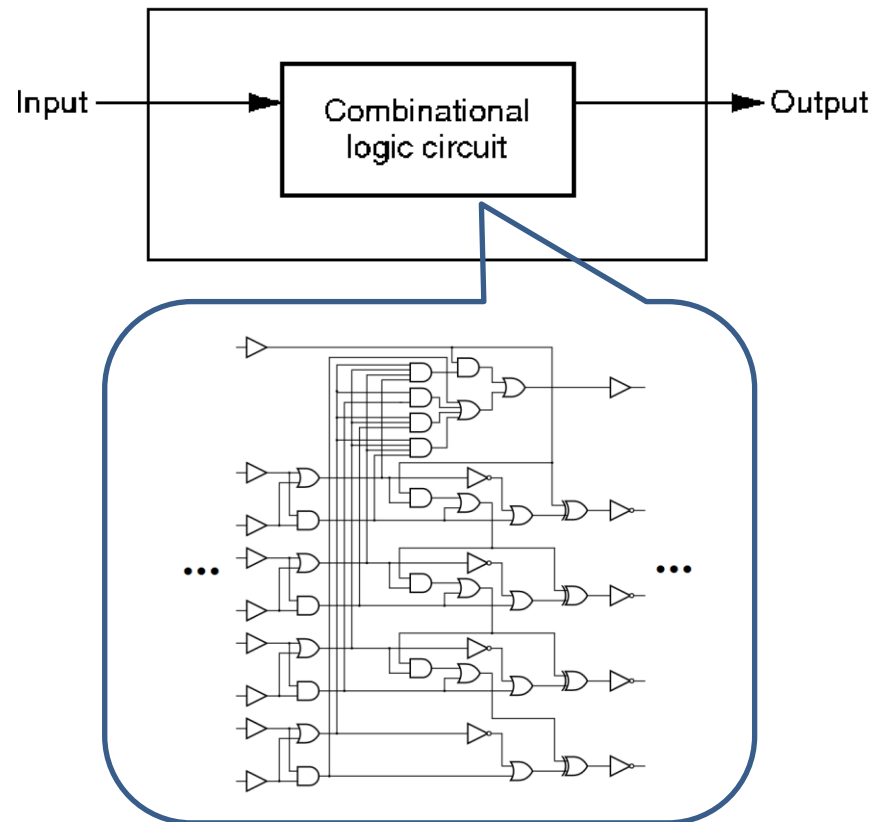
✓ Reset

✓ Coding Style



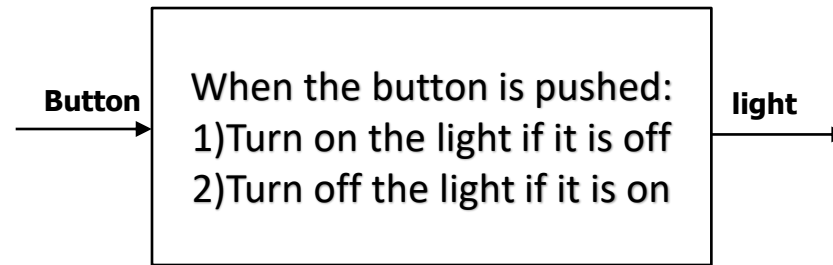
Motivation

- ✓ **Progress so far : Combinational circuit**
 - Output is only a function of the **current** input values



Motivation

- ✓ What if you were given the following design specification:



- ✓ What makes this circuit so different from we've discussed before?

“State”

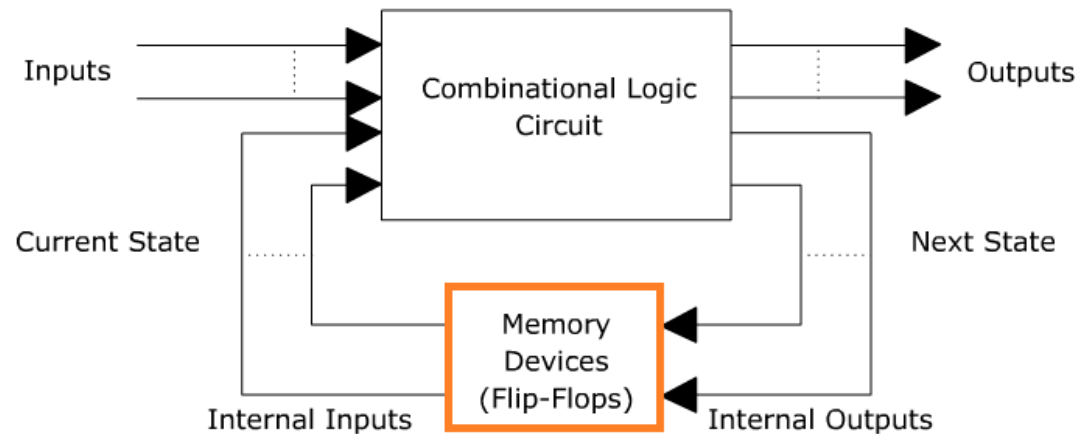
What is Sequential Circuit ?

✓ Sequential circuit

- Output depends not only on the current input values, but also on **preceding** input values
- It remembers sort of the past history of the system

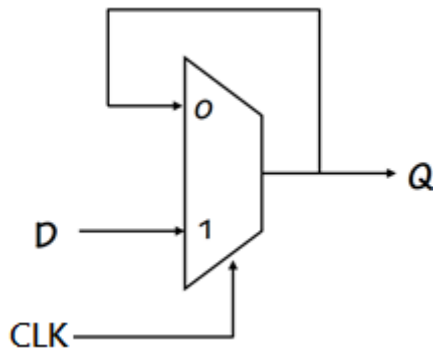
✓ How?

- Registers(Flip-Flops)



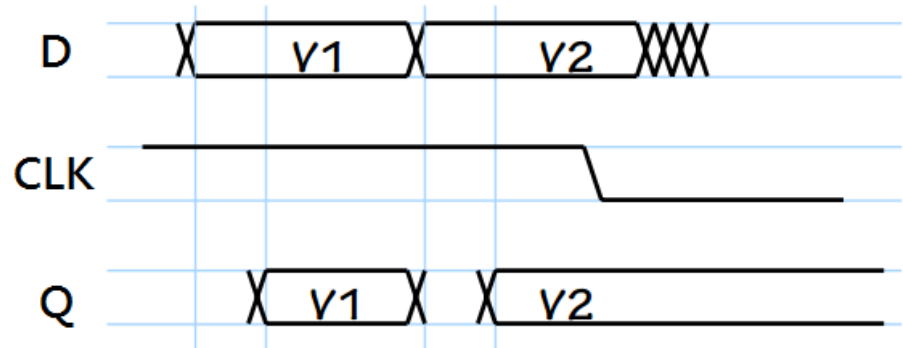
Latch Operation

✓ Latch: level sensitive



CLK	D	Q	Q'	
0	--	0	0	} Q stable
0	--	1	1	
1	0	--	0	} Q follows D
1	1	--	1	

CLK=1 : Q follows D
CLK=0 : Q holds

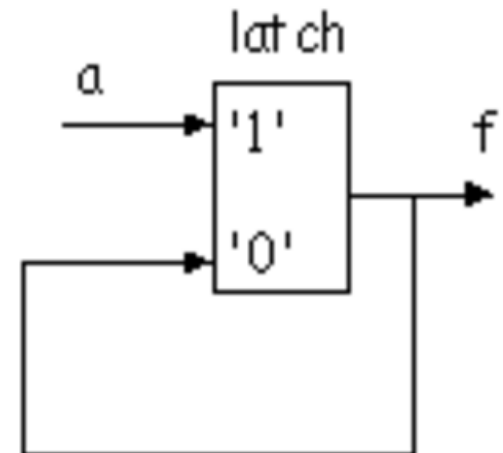
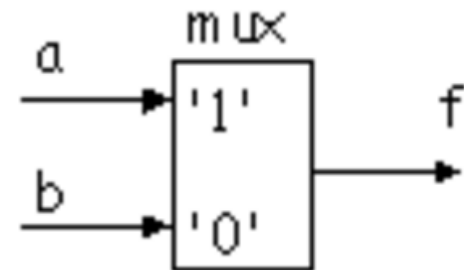


Avoid Unintentional Latch (1/2)

✓ Example

```
always @(*)  
begin  
    if(sel == 1) f = a;  
    else f = b;  
end
```

```
always @(*)  
begin  
    if(sel == 1) f = a;  
end
```



Avoid Unintentional Latch (2/2)

✓ Avoid latches in combinational circuit

- Avoid incomplete if-then-else
- Avoid incomplete case statements

X

```
if(!rst_n) out = 0;  
else if(m==3'd0) out = m0_out;  
else if(m==3'd1) out = m1_out;
```

X

```
case(mode)  
  3'd0: out = m0_out;  
  3'd1: out = m1_out;  
endcase
```

O

```
if(!rst_n) out = 0;  
else if(m==3'd0) out = m0_out;  
else if(m==3'd1) out = m1_out;  
else out = default_out;
```

O

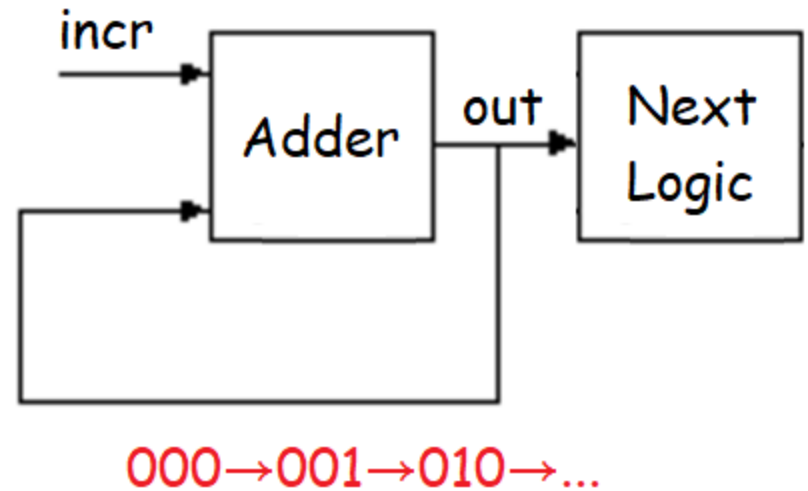
```
case(mode)  
  3'd0: out = m0_out;  
  3'd1: out = m1_out;  
  default:  
    out = default_out;  
endcase
```



Avoid Combinational Feedback (1/2)

✓ Example

```
assign out=out+1;
```



Avoid Combinational Feedback (2/2)

✓ Avoid combinational feedbacks

- Lead to unpredictable oscillated output
- NOT allowed

X

```
assign a=a+1;
```

X

```
always @(*) begin
    a = a+1;
end
```

X

```
always @(*) begin
    if(in_a) a = c;
    else a = a;
end
```

X

```
assign out_value=out;
always @(*) begin
    case(mode)
        3'd0: out = m0_out;
        3'd1: out = m1_out;
        default:
            out = out_value;
    endcase
end
```

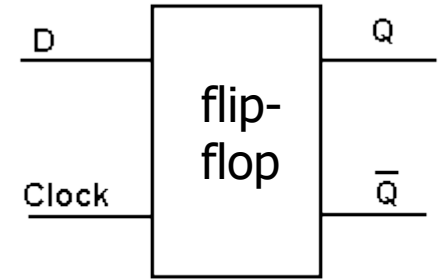


Avoid Latch Summary

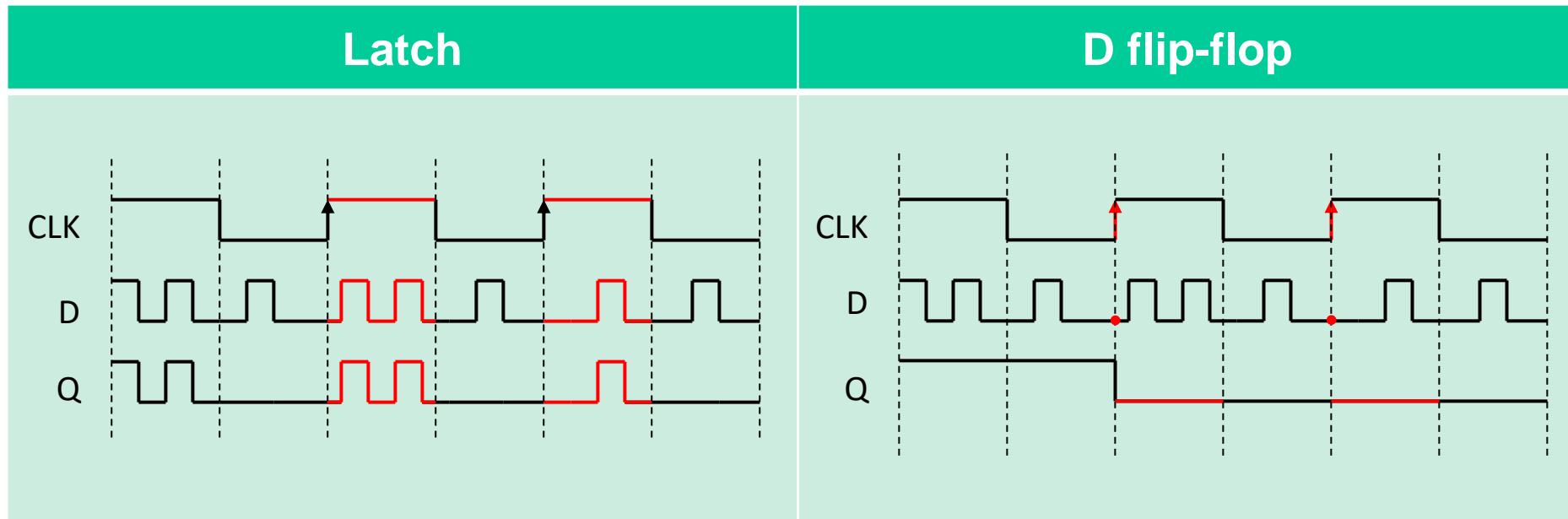
- ✓ **In a sequential circuit -- with clk control**
 - It is a flip-flop so there is not a latch problem.
- ✓ **In a combinational circuit -- without clk control**
 - If some net needs to keep its data, DC will synthesize a latch.
- ✓ **How to avoid?**
 - Conditional statement : must be full cases
 - Otherwise it will produce latches.
 - if – else work together or add default value
 - Ex: `if (a == b) c = 1 ;`
 - Case statement : remember default value
 - Ex: `case (a) 1'b0: c = b; endcase`
 - Avoid combinational feedback
- ✓ **Notice**
 - In a combinational circuit, no information will be stored, so latches are not allowed.
- ✓ **Latch is a memory storage device**
 - It will cause the problems of timing analysis .
 - That's why we recommend to avoid latches here!!

Flip-Flop Operation

✓ D flip-flop: edge triggered



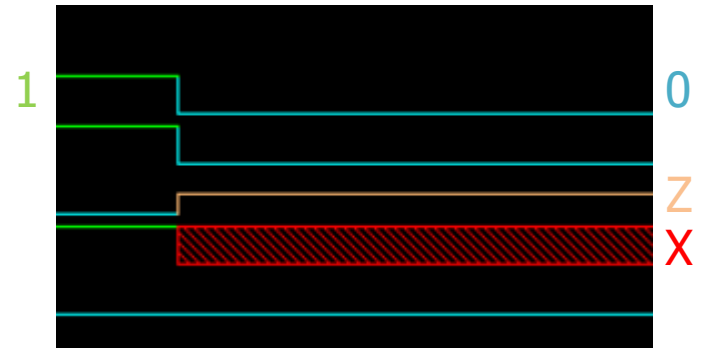
✓ Positive latch v.s. positive D flip-flop



Flip-Flop Data Type

✓ Flip-flop: data storage element with 4 states (0,1, X, Z)

- **0**: logic low
- **1**: logic high
- **X**: unknown, may be a 0,1, Z, or in transition
- **Z**: high impedance, floating state



✓ Operations on the 4 states

- Example: AND, OR, NOT gate

AND	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

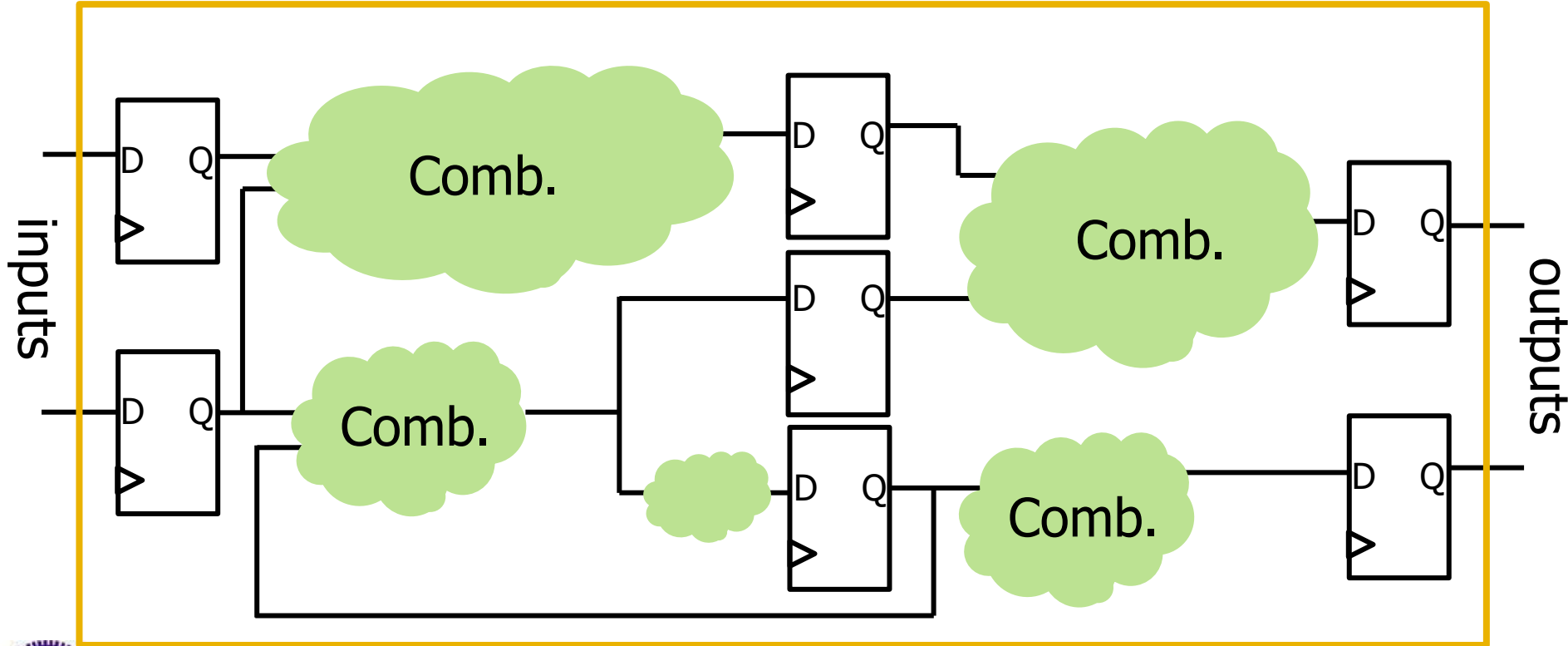
OR	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

NOT	output
0	1
1	0
X	X
Z	X



Concept of Sequential Circuit

- ✓ Most computations are done by combinational circuit
 - ✓ Sequential elements are used for storage
- top design



Outline

✓ **Section 1 Sequential Circuits**

✓ Introduction

✓ **Syntax**

✓ Reset

✓ Coding Style



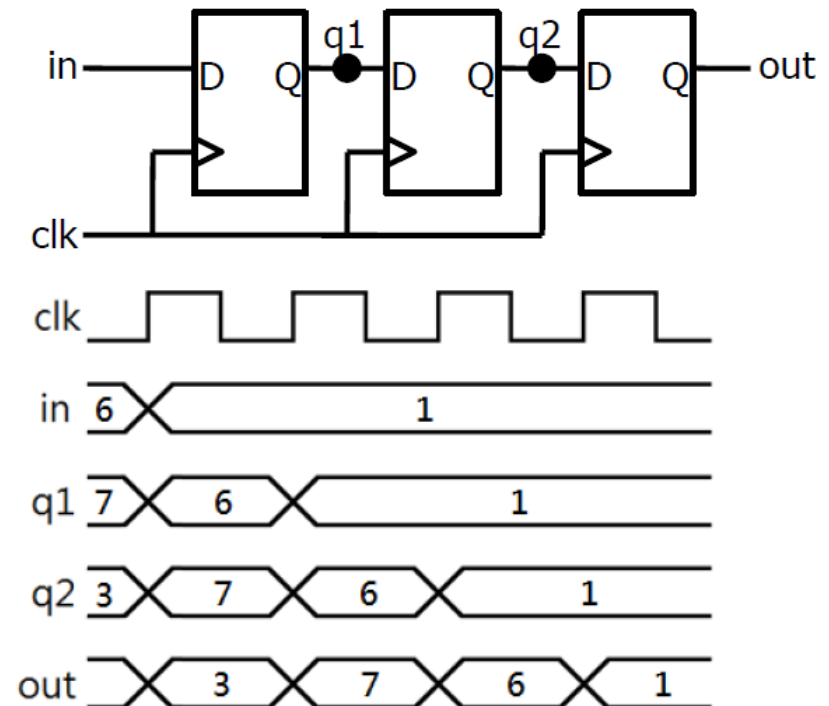
Assignment in Sequential Circuit

✓ Non-blocking assignment

- Evaluations and assignments are executed **at the same time without regard to orders or dependence upon each other**
- Syntax : **<variable> <= <expression>;**

✓ Example

```
always @(posedge clk)
begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
end
```



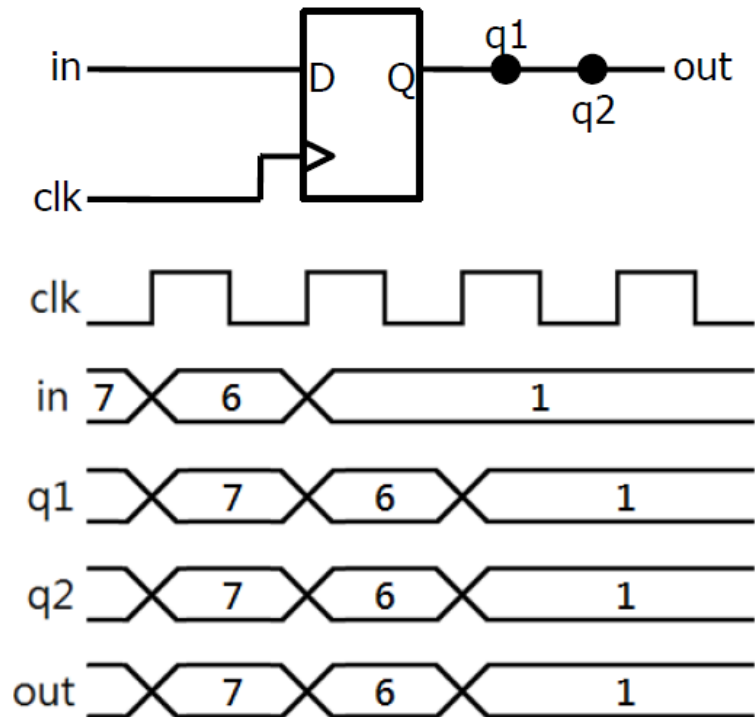
Assignment in Sequential Circuit

✓ Blocking assignment

- Evaluations and assignments are **immediate** and **in order**
- Syntax : **<variable> = <expression>;**

✓ Example

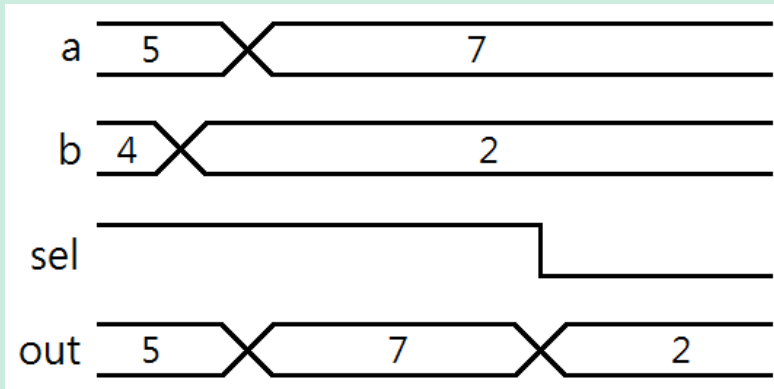
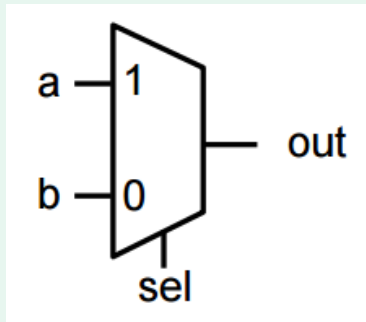
```
always @(posedge clk)
begin
    q1 = in;
    q2 = q1;
    out = q2;
end
```



Combinational v.s. Sequential

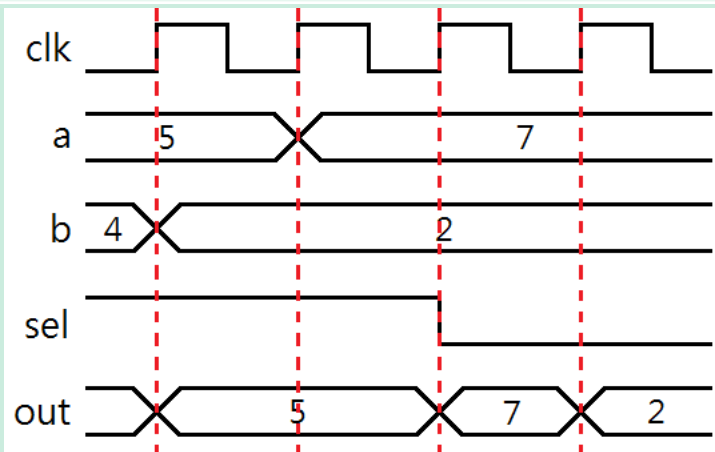
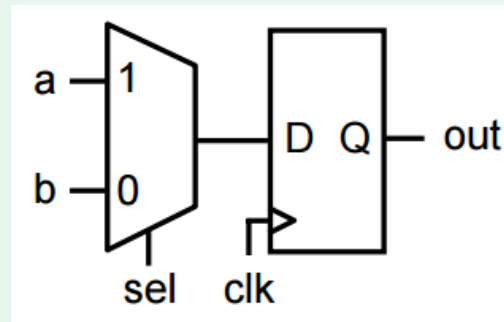
Combinational

```
always@ (*)  
begin  
    if(sel) out = a;  
    else    out = b;  
end
```



Sequential

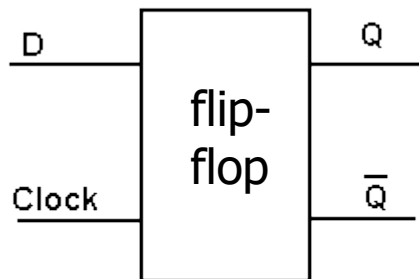
```
always@(posedge clk)  
begin  
    if(sel) out <= a;  
    else    out <= b;  
end
```



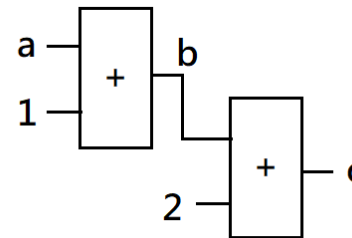
Sequential Circuit

- ✓ **Sequential block**
 - use **non-blocking** assignments
- ✓ **Combinational block**
 - use **blocking** assignments
- ✓ **Comb./Seq. logic should be separated**

```
always@(posedge clk)
begin
    Q <= D;
end
```



```
always@*
begin
    b = a + 1;
    c = b + 2;
end
```



Outline

✓ Section 1 Sequential Circuits

✓ Introduction

✓ Syntax

✓ **Reset**

✓ Coding Style

✓ Generate & For Loop

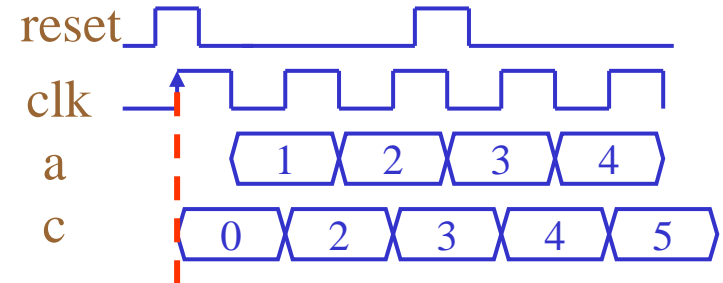


Synchronous Reset (1/2)

✓ Register with synchronous reset

- Syntax: **always@ (posedge clk)**

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

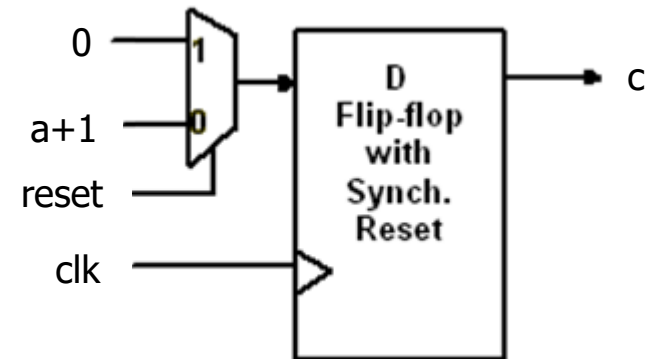


✓ Advantages

- Glitch filtering from reset combinational logic

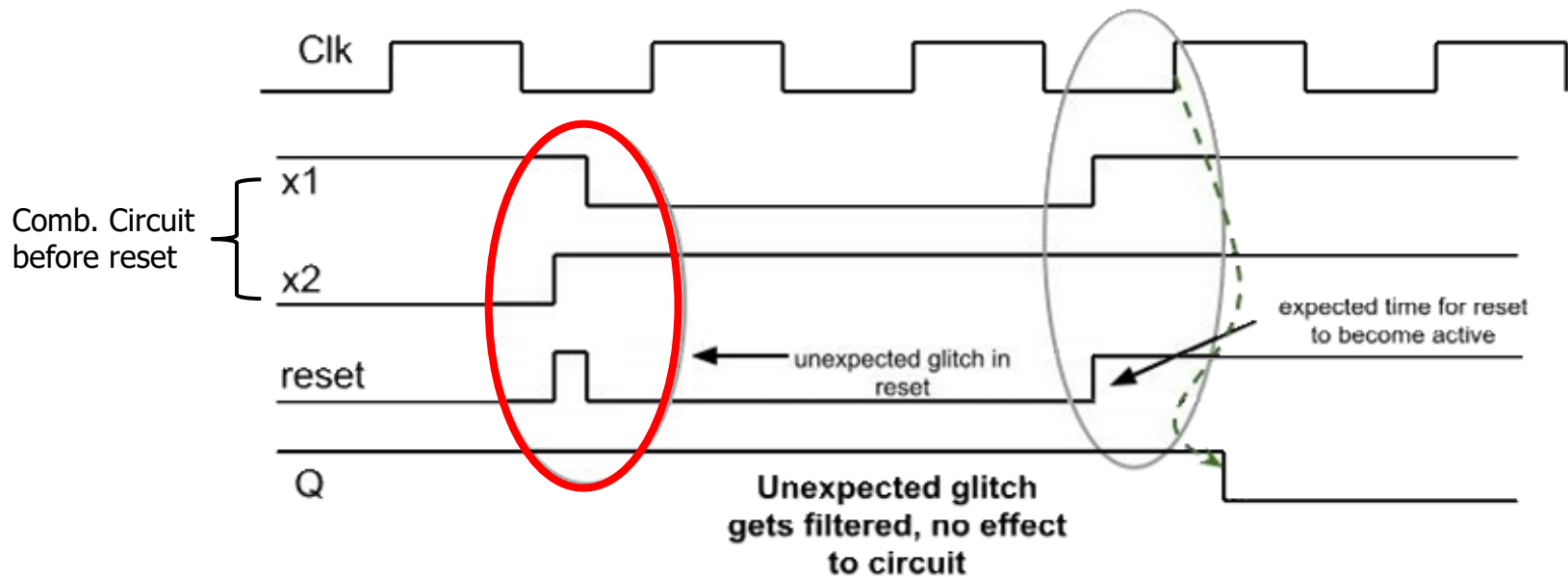
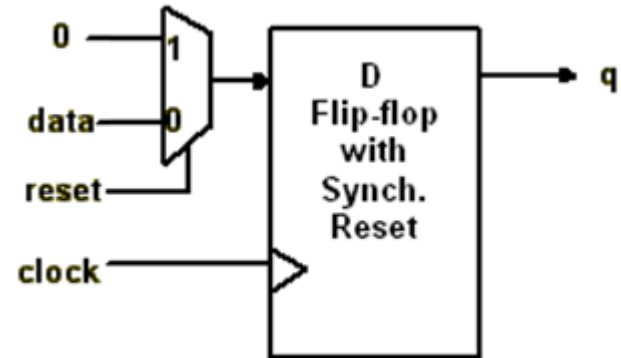
✓ Disadvantages

- Can't be reset without clock signal
- May need a pulse stretcher
 - Guarantee a reset pulse wide enough
- Larger area
- Increasing critical path



Synchronous Reset (2/2)

✓ Advantage: glitch filtering

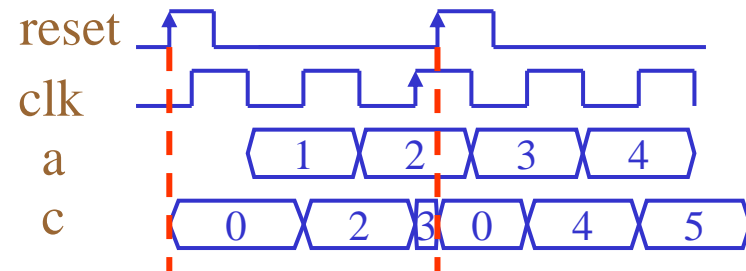


Asynchronous Reset

✓ Register with asynchronous reset

- Syntax: **always @(posedge clk or posedge reset)**

```
always @(posedge clk or posedge reset)
begin
    if (reset) c <= 0;
    else c <= a+1;
end
```

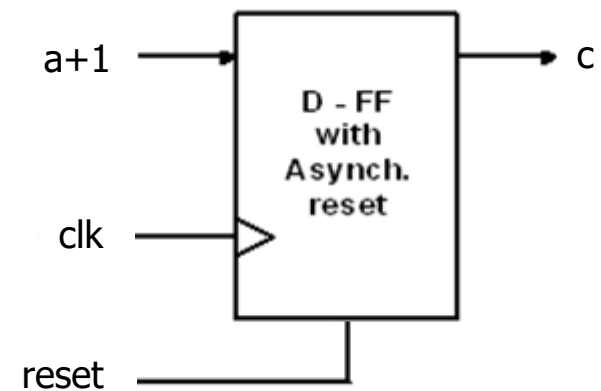


✓ Advantages

- Reset is independent of clock signal
- Reset is immediate
- Less area

✓ Disadvantages

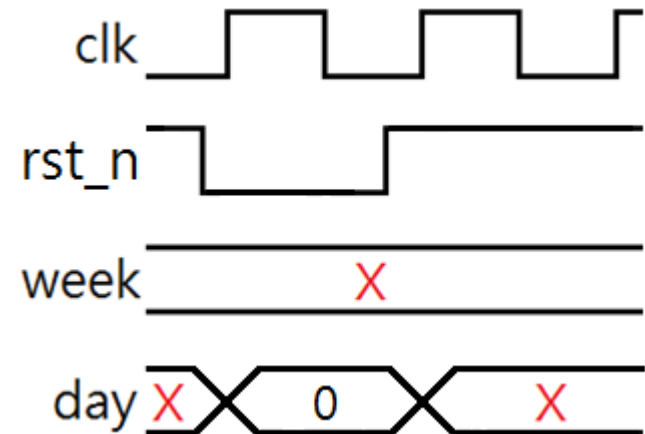
- Noisy reset line could cause unwanted reset
- Metastability



Avoid Unknown

- ✓ Reset all signals to avoid unknown propagation

Xalways @(posedge clk) begin
// if(!rst_n) week <= 0;
week <= week+1;
end
always @(posedge clk) begin
if(!rst_n) day <= 0;
else day <= week * 7;
end



AND	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

OR	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

NOT	output
0	1
1	0
X	X
Z	X



Outline

✓ **Section 1 Sequential Circuits**

✓ Introduction

✓ Syntax

✓ Reset

✓ **Coding Style**



Coding Styles (1/2)

✓ Naming should be readable

✓ Synthesizable codes

- assign, always block, called sub-modules, if-then-else, cases, parameters, operators

✓ Data has to be described in one always block

- Multiple source drive is not valid

X

```
always @(posedge clk) begin
    out <= out+1;
end
always @(posedge clk) begin
    out <= a;
end
```

✓ Always block can't exist both blocking and non-blocking assignment

X

```
always @(posedge clk) begin
    if(reset) out = 0;
    else out <= out+in;
end
```



Coding Styles (2/2)

✓ Do not put many variables in one `always` block

- Except shift registers or registers with similar properties

bad

```
always @(posedge CLK) begin
    q2 <= in;
    if(sel==0) out <= q2;
    else if(sel==1) out <= q3;
    else out <= out;
end
```

suggested

```
always @(posedge CLK) begin
    q2 <= in;
end
always @(posedge CLK) begin
    if(sel==0) out <= q2;
    else if(sel==1) out <= q3;
    else out <= out;
end
```

✓ Use FSM (Finite State Machine)



Outline

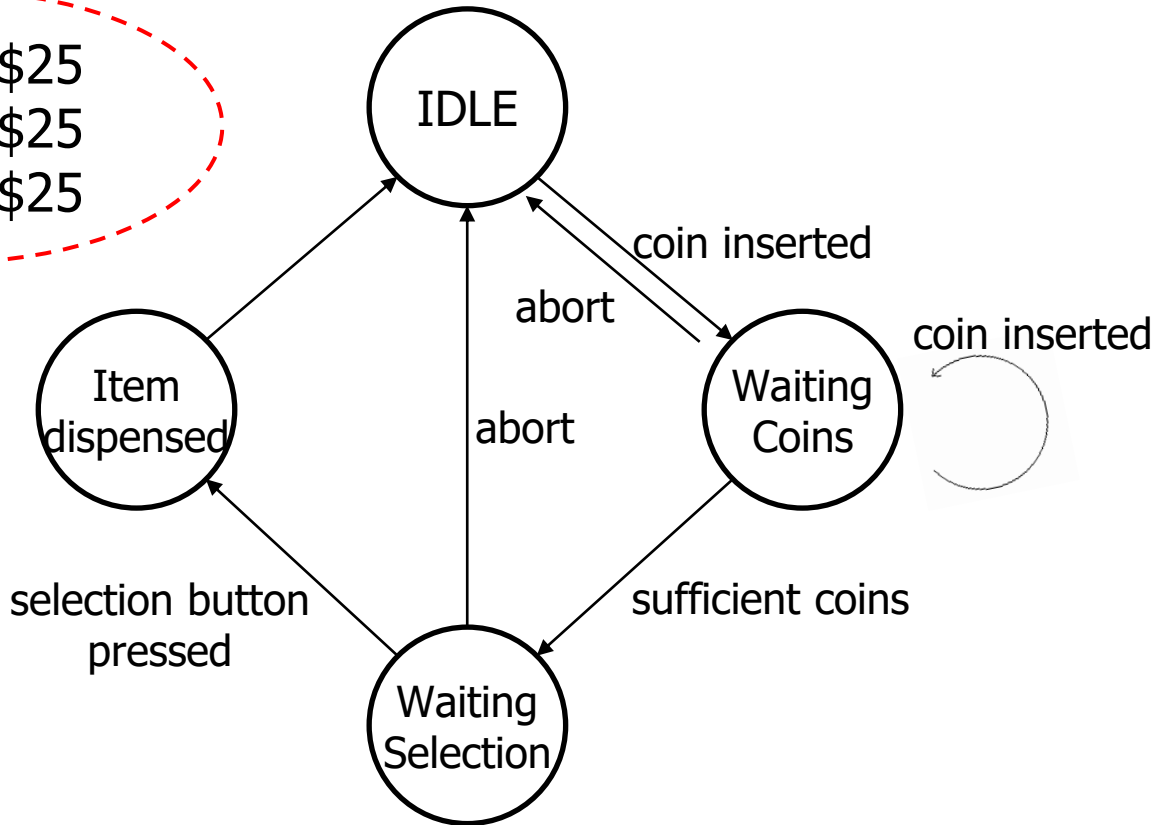
- ✓ Section 1 Sequential Circuits
- ✓ **Section 2 Finite State Machine**
- ✓ Section 3 Timing
- ✓ Section 4 Synthesis and Design Compiler
- ✓ Section 5 Generate & for loop



Finite State Machine

✓ Example: Vending machine

Coke \$25
Pepsi \$25
Sprite \$25

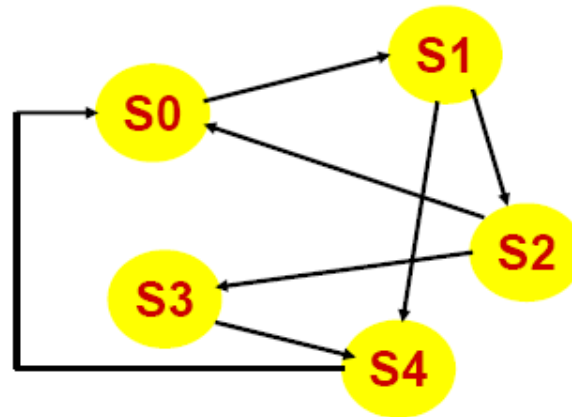


Finite State Machine

✓ Finite state machine

- Powerful model for describing a sequential circuit
- Divide a sequential circuit operation into finite number of states.
- A state machine controller can output results depending on the input signal, control signal and states.
- As different input or control signal changes, the state machine will take a proper state transition.

✓ State diagram



Mealy and Moore Machines(1/3)

✓ Mealy machine

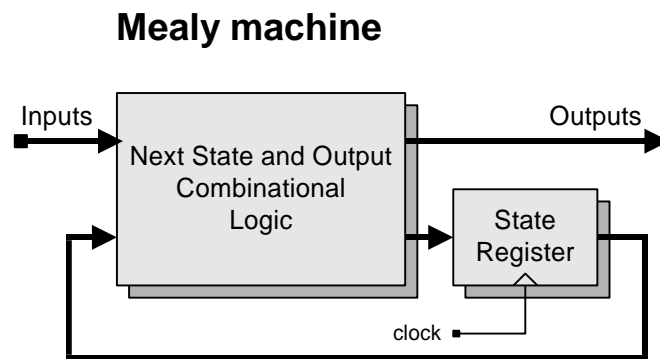
- The outputs depend on the current state and inputs
- If input changes, output also changes

✓ Advantages

- Less number of states are required

✓ Disadvantages

- More hardware requirements for circuit implementation



$$\text{output} = f(\text{In}, \text{CS})$$

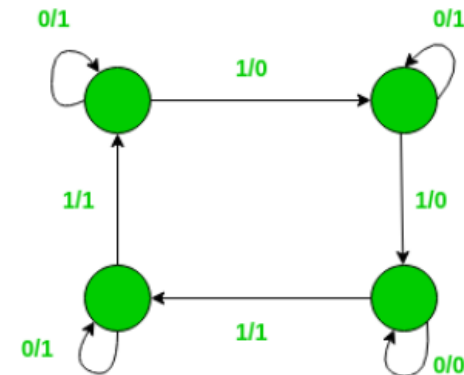


Figure - Mealy machine



Mealy and Moore Machines(2/3)

✓ Moore machine

<https://www.geeksforgeeks.org/difference-between-mealy-machine-and-moore-machine/>

- The outputs depend on the current state only
- Inputs affect outputs but not immediately

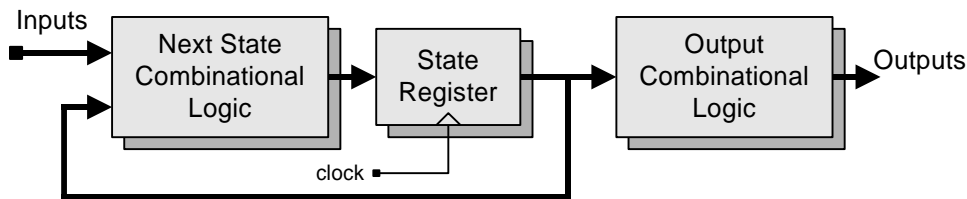
✓ Advantage

- Safer. Outputs change at clock edge

✓ Disadvantage

- More states are required

Moore machine



$$\text{output} = f(\text{CS})$$

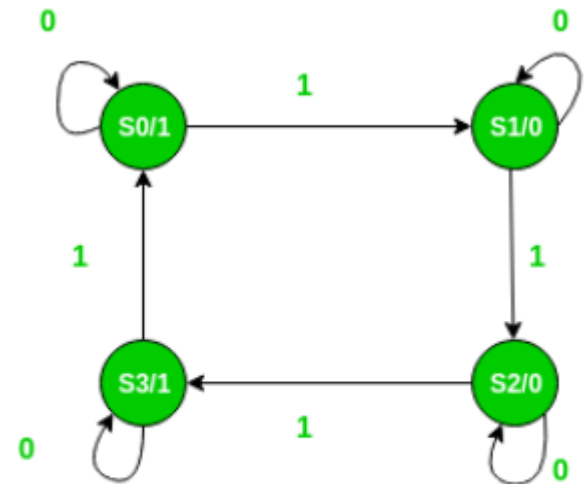


Figure - Moore machine



Mealy and Moore Machines(3/3)

- FSM coding style
 - Separate CS, NS and OL

Current
State

```
always @(posedge clk)
    current_state<=next_state;
```

Next
State

```
always @(current_state or In)
case (current_state)
state_0: case(In)
            In0: next_state = state_value1;
            In1: next_state = state_value2;
            .....
        endcase
        .....
    default : .....
endcase
```

If it is not full case and without
default case ,latch will be incurred!

Mealy
machine

Moore
machine

Output
Logic

```
always @(current_state or In)
    Z = values;
```

```
always @(current_state)
    Z = values;
```

FSM Coding Style

✓ Separate current state, next state and output logic

Current State

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) current_state <= IDLE;
    else current_state <= next_state;
end
```

Use parameters for readability

Next State

```
always @(*) begin
    if(!rst_n) next_state=IDLE;
    else begin
        case(current_state)
            STATE_1: begin
                if (in==in_1) next_state=STATE_2;
                else next_state=current_state;
            end
            STATE_2: .....
            ....
            default: next_state=current_state;
        endcase
    end
end
```

If it's not full case and without default case, latch would be incurred!

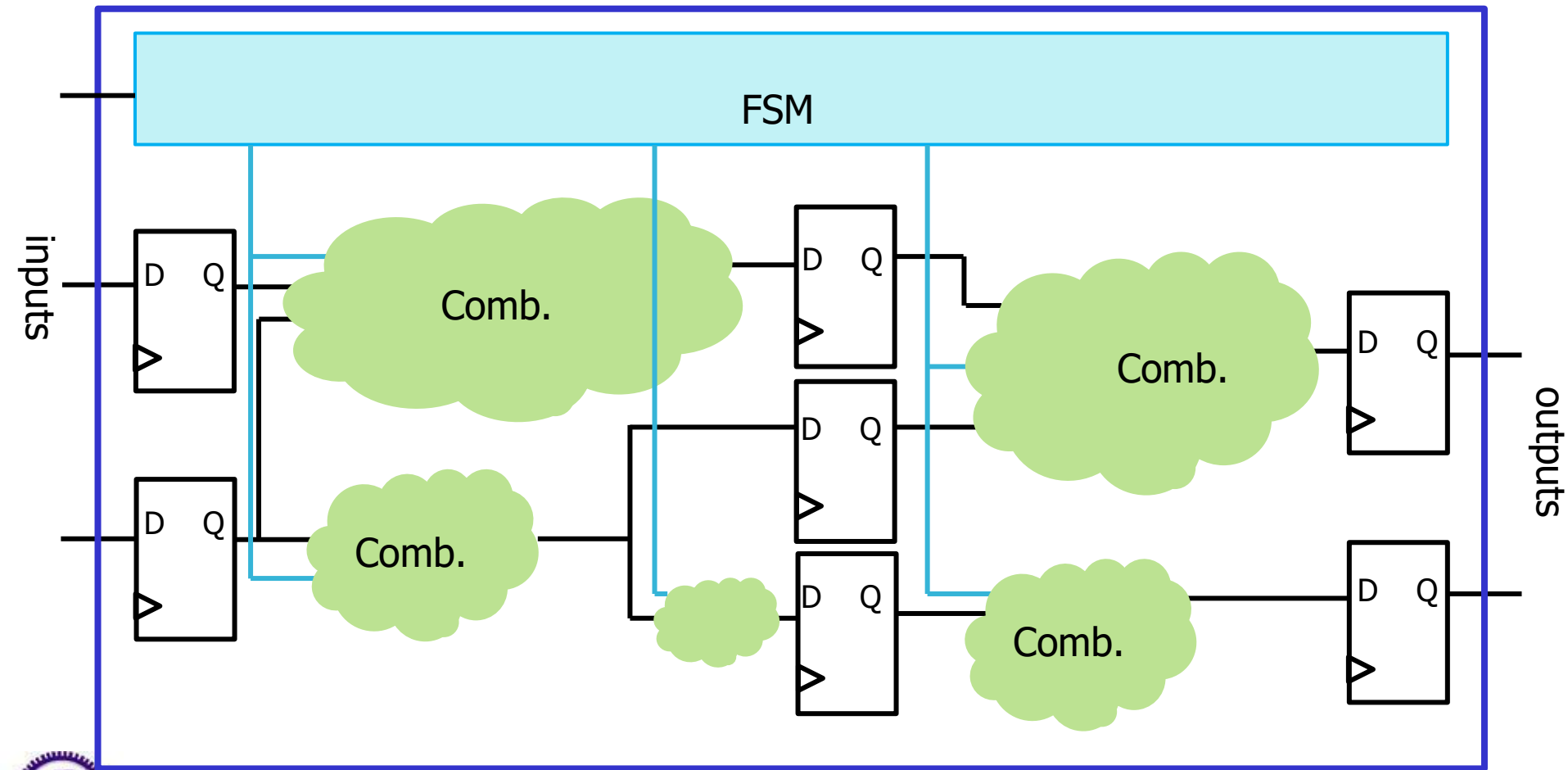
```
parameter IDLE      = 2'd0;
parameter STATE_1   = 2'd1;
parameter STATE_2   = 2'd2;
parameter STATE_3   = 2'd3;
```

Output Logic

```
always@(posedge clk or negedge rst_n) begin
    if (!rst_n) out <= 0;
    else if (current_state==STATE_3) out <= output_value;
    else out <= out;
end
```

Why FSM?

- ✓ FSM can be referred to as the controller and status of the whole module



Outline

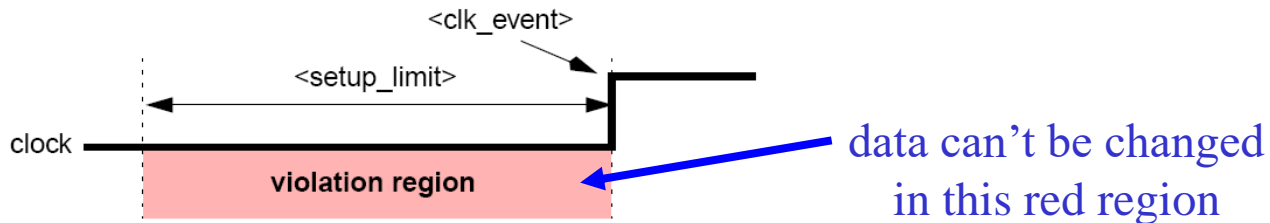
- ✓ Section 1 Sequential Circuits
- ✓ Section 2 Finite State Machine
- ✓ **Section 3 Timing**
- ✓ Section 4 Synthesis and Design Compiler
- ✓ Section 5 Generate & for loop



Timing Check (1/3)

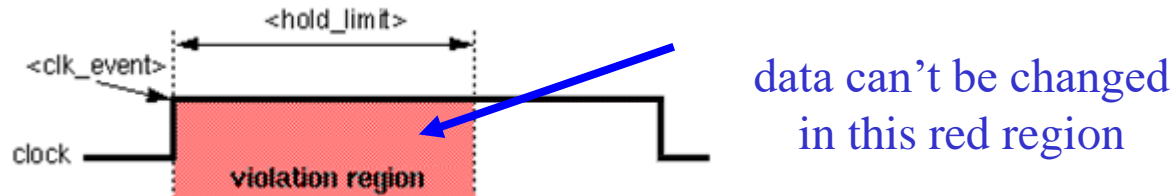
✓ Setup time check

- The `$setup` system task determines whether a data signal remains stable for a minimum specified time before a transition in an enabling, such as a clock event.



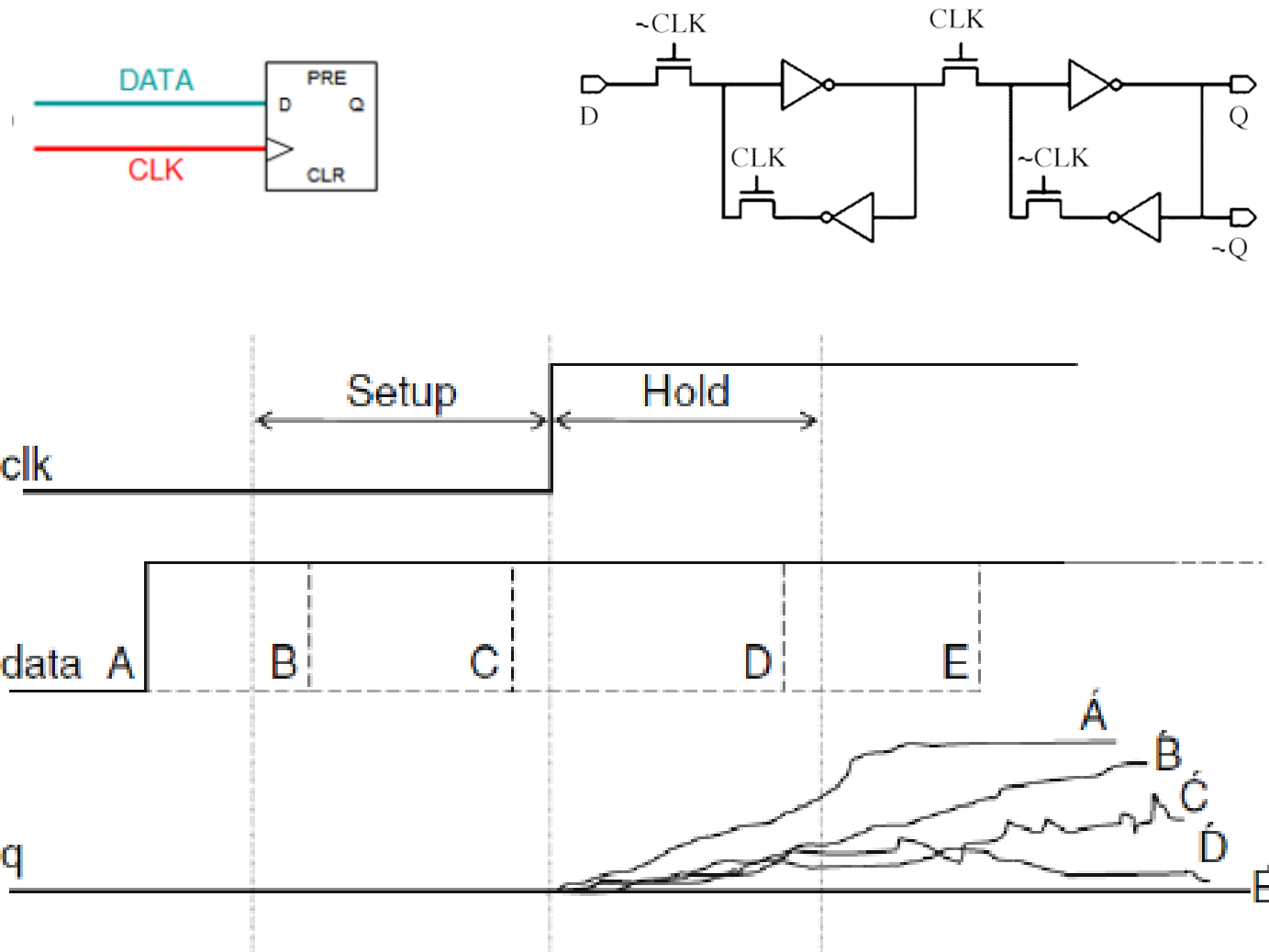
✓ Hold time check

- The `$hold` system task determines whether a data signal remains stable for a minimum specified time after a transition in an enabling signal, such as a clock event.



Timing Check (2/3)

✓ Metastability



Timing Check (3/3)

✓ Timing report: setup time

clock CLK_1 (rise edge)	2.00	2.00
clock network delay (ideal)	2.00	4.00
clock uncertainty	-0.50	3.50
IN_A_reg[0]/CK (EDFFXL)	0.00	3.50 r
library setup time	-0.42	3.08
data required time		3.08

data required time		3.08
data arrival time		-3.08

slack (MET)		0.00

✓ Timing report: hold time

Slacks should be **MET**!
(non-negative)

clock CLK_2 (rise edge)	0.00	0.00
clock network delay (ideal)	4.00	4.00
clock uncertainty	1.00	5.00
IN_B_reg[20]/CK (EDFFXL)	0.00	5.00 r
library hold time	-0.19	4.81
data required time		4.81

data required time		4.81
data arrival time		-4.82

slack (MET)		0.01

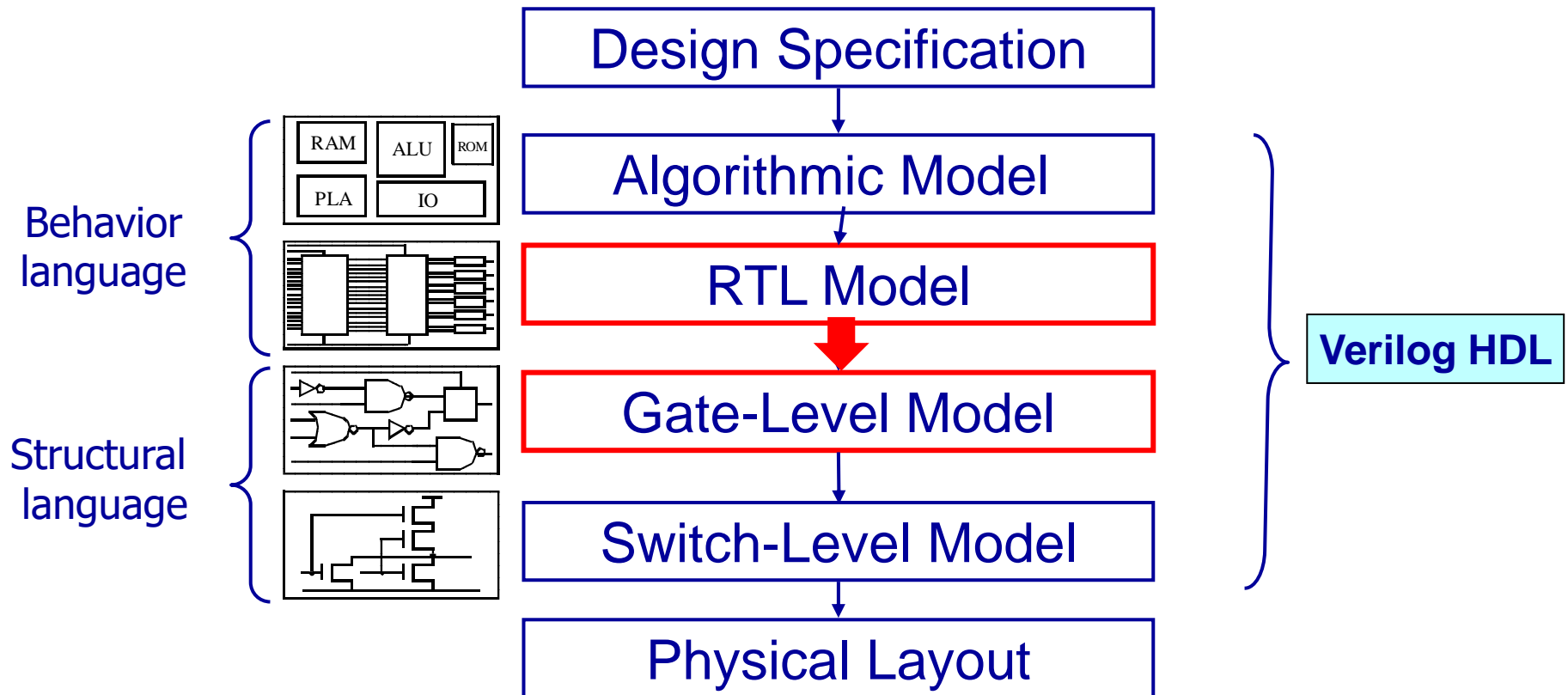


Outline

- ✓ Section 1 Sequential Circuits
- ✓ Section 2 Finite State Machine
- ✓ Section 3 Timing
- ✓ **Section 4 Synthesis and Design Compiler**
- ✓ Section 5 Generate & for loop



Recall: Design Flow



Logic Synthesis

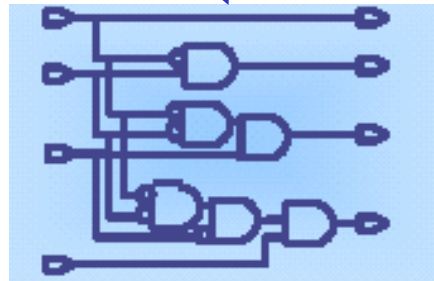
✓ Logic synthesis

- A process by which behavioral model of a circuit is turned into an implementation in terms of logic gates
- Synthesis = **Translation+Mapping+Optimization**

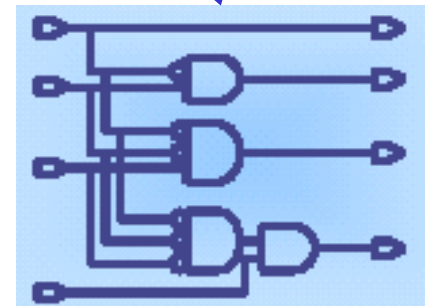
```
assign avg=sum/total;  
always_ff @(posedge clk)  
begin  
    sum=sum+score*weight;  
end
```

HDL Source

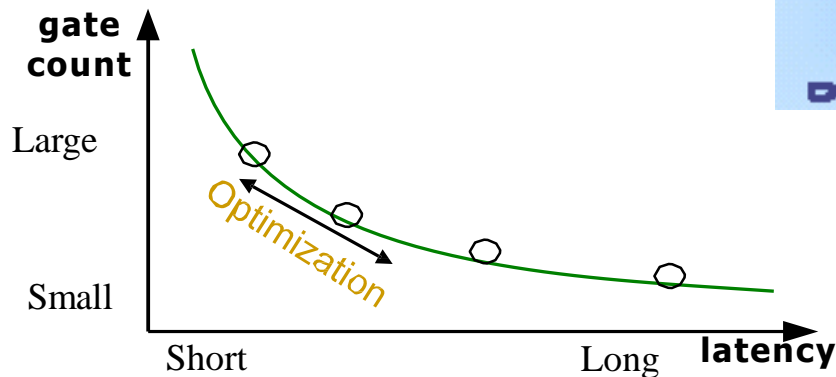
Translate



Map+Optimize



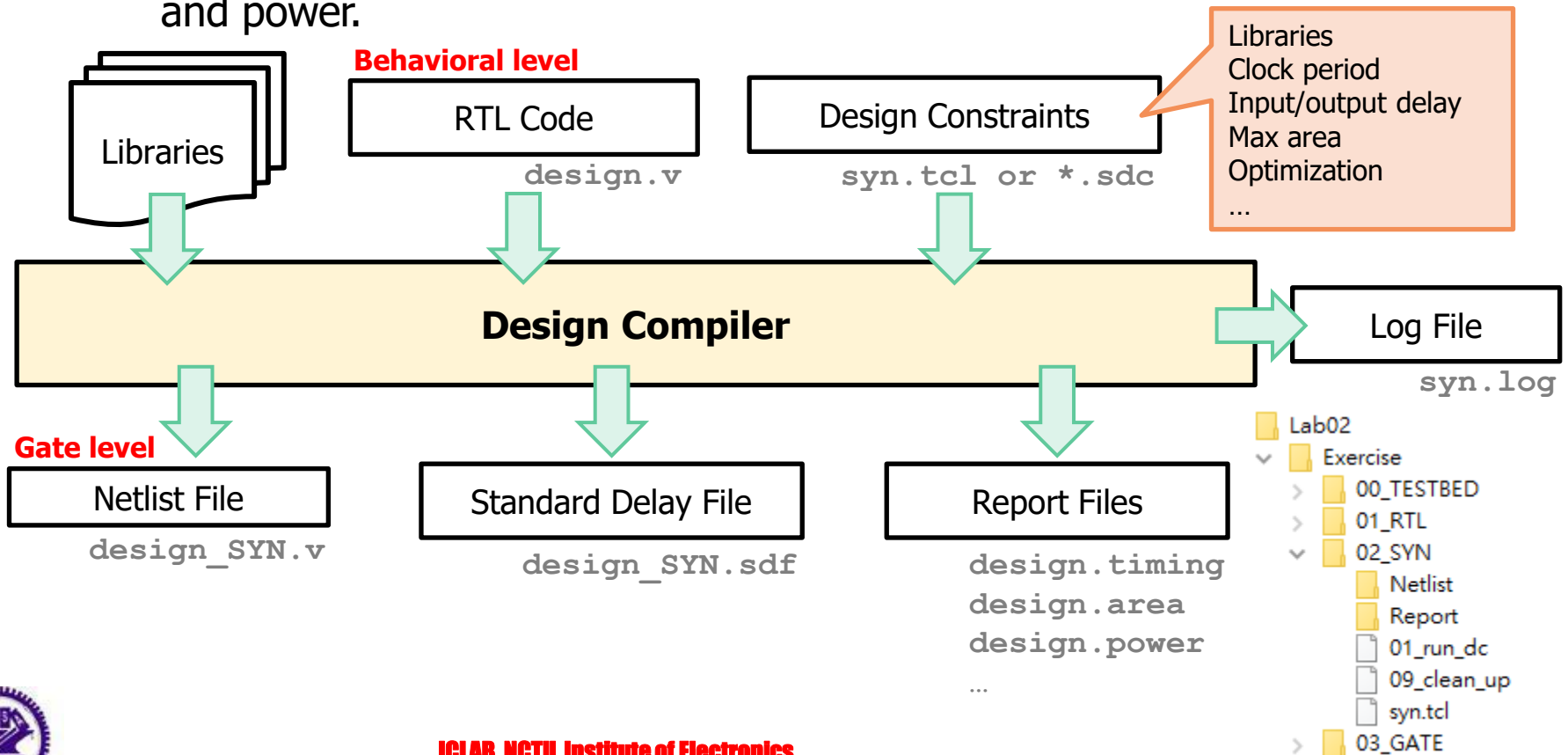
Target Technology



Design Compiler

✓ Design compiler

- A tool by Synopsys, Inc. that synthesizes your HDL designs (**Verilog**) into optimized technology-dependent, **gate-level** designs.
- It can optimize both combinational and sequential designs for speed, area, and power.



.lib

- ✓ Cell name
- ✓ Drive strength
- ✓ Area
- ✓ Pin
- ✓ Leakage power
- ✓ Timing for each pin
- ✓ Internal power



Same information as .db file

```
cell (NANDX1) {
  pin(A1) {
    direction : input;
    capacitance : 0.00683597;
  }
  pin(A2) {
    direction : input;
    capacitance : 0.00798456;
  }
  pin(ZN) {
    direction : output;
    capacitance : 0.0;
    internal_power() {
      timing() {
        cell_rise(table10){
          values ("0.020844,0.02431,0.030696,0.039694,0.048205,0.072168,0.10188",\
                  "0.024677,0.027942,0.035042,0.045467,0.054973,0.082349,0.11539",\
                  "0.032068,0.035394,0.042758,0.055361,0.065991,0.090936,0.13847",\
                  "0.046811,0.049968,0.057164,0.064754,0.086481,0.11676,0.15744",\
                  "0.073919,0.078805,0.080873,0.091007,0.11655,0.1579,0.21448",\
                  "0.13162,0.13363,0.1383,0.14793,0.1685,0.22032,0.30054",\
                  "0.24661,0.24835,0.25294,0.26221,0.282,0.32417,0.42783");
          out_capacitance
          input_transition_time
        }
      }
    }
  }
}

lu_table_template(table10){
  variable_1 : total_output_net_capacitance;
  variable_2 : input_transition_time;
  index_1 ("0.001400,0.003000,0.006200,0.012500,0.025100,0.050400,0.101000");
  index_2 ("0.0208,0.0336,0.06,0.1112,0.2136,0.4192,0.8304");
}
```

Outline

- ✓ Section 1 Sequential Circuits
- ✓ Section 2 Finite State Machine
- ✓ Section 3 Timing
- ✓ Section 4 Synthesis and Design Compiler
- ✓ **Section 5 Generate & for loop**



Generate

SystemVerilog

3.1a { assertions test program blocks clocking domains process control		mailboxes semaphores constrained random values direct C function calls		from C / C++ classes inheritance strings dynamic arrays associative arrays references	
		packages 2-state modeling packed arrays array assignments queues unique/priority case/if compilation unit space		int shortint longint byte shortreal void alias globals enum typedef structures unions casting const break continue return do-while ++ -- += -= *= /= >>= <<= >>>= <<<= &= = ^= %=	
3.0 { interfaces nested hierarchy unrestricted ports automatic port connect enhanced literals time values and units specialized procedures					

Verilog-2001

ANSI C style ports
 generate
 localparam
 constant functions

standard file I/O
 \$value\$plusargs
 `ifndef `elsif `line
 @*

(* attributes *)
 configurations
 memory part selects
 variable part select

multi dimensional arrays
 signed types
 automatic
 ** (power operator)

Verilog-1995

modules
 parameters
 function/tasks
 always @
 assign
 \$finish \$fopen \$fclose
 \$display \$write
 \$monitor
 `define `ifdef `else
 `include `timescale

initial
 disable
 events
 wait # @
 fork-join
 wire reg
 integer real
 time
 packed arrays
 2D memory

begin-end
 while
 for forever
 if-else
 repeat
 + = * /
 %
 >> <<

For Loop

- For loop in Verilog
 - Duplicate same function
 - Very useful for doing reset and iterated operation
 - Unrolling

```
reg [3:0] temp[0:2];
integer i;
always @(posedge clk) begin
    for (i = 0; i < 3 ; i = i + 1) begin: for_name
        temp[i] <= 4'b0;
    end
end
```

=

```
always @(posedge clk) begin
    temp[0] <= 4'b0;
    temp[1] <= 4'b0;
    temp[2] <= 4'b0;
end
```

```
reg [3:0] temp[0:2];
reg [3:0] data;
integer i;
always @(posedge clk) begin
    for (i = 0; i < 3 ; i = i + 1) begin: for_name
        temp[i+1] <= temp[i];
    end
    temp[0] <= data;
end
```

=

```
always @(posedge clk) begin
    temp[0] <= data;
    temp[1] <= temp[0];
    temp[2] <= temp[1];
    temp[3] <= temp[2];
end
```


Generate

- How to use for loop with generate?
 - For loop in generate : four always blocks
 - Regular for loop : one always block

```
reg [3:0] temp;  
genvar i;  
generate  
for (i = 0; i < 4 ; i = i + 1) begin: for_name  
    always @(posedge clk) begin  
        temp[i] <= 1'b0;  
    end  
end  
endgenerate
```

Generate block

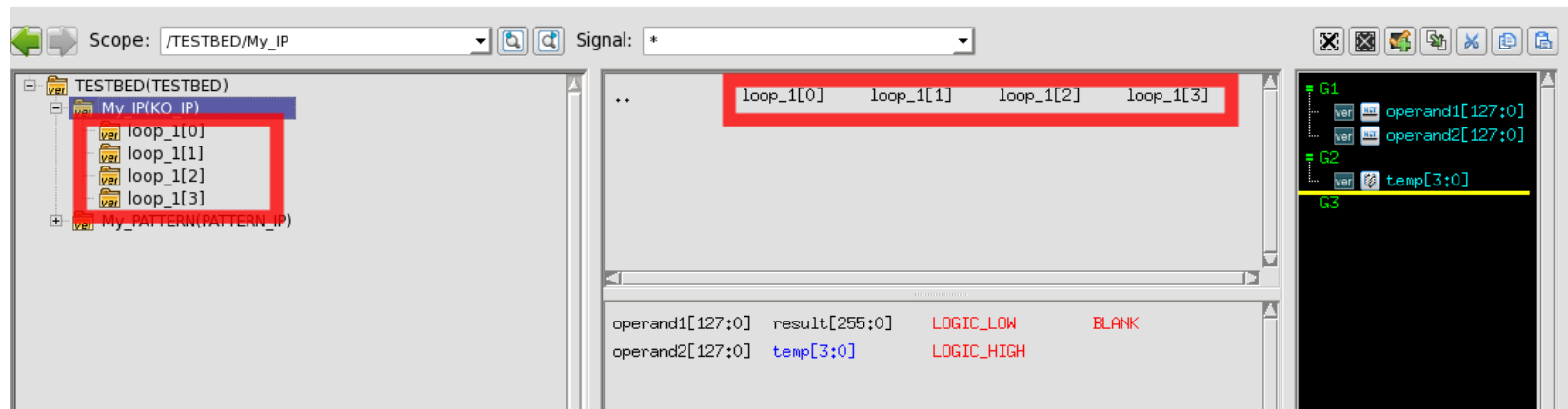
```
reg [3:0] temp;  
integer i;  
always @(posedge clk) begin  
    for (i = 0; i < 4 ; i = i + 1) begin:  
        temp[i] <= 1'b0;  
    end  
end
```

Regular for loop

Generate

```
reg [3:0] temp;  
  
genvar i;  
generate  
for (i=0 ; i <4; i = i+1)begin loop_1  
    always@(*)begin  
        temp[i] = operand1[i] & operand2[i];  
    end  
end  
endgenerate
```

always block in for loop with
genvar

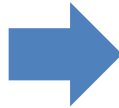


4 always block
instance

For Loop/Generate Example

- ✓ **Example**
 - Copy a module for 3 times
- ✓ **Generate:**

```
module A();  
endmodule  
  
module B();  
  genvar i;  
  generate  
    for(i=0; i<3; i=i+1) begin  
      A uA(...)  
    end  
  endgenerate  
endmodule
```



```
module A();  
...  
endmodule  
  
module A();  
...  
endmodule  
  
module A();  
...  
endmodule
```

<https://www.chipverify.com/verilog/verilog-generate-block>



For Loop/Generate Example

- ✓ **Example**
 - Copy a module for 3 times
- ✓ **Generate:**

```
module A();  
endmodule  
  
module B();  
for(i=0; i<3; i=i+1) begin  
  A uA(...)  
end  
endmodule
```

