Sequential Circuits

NCTU-EE IC Lab Fall-2021

Lecturer: Yi-Ching Wang

Outline

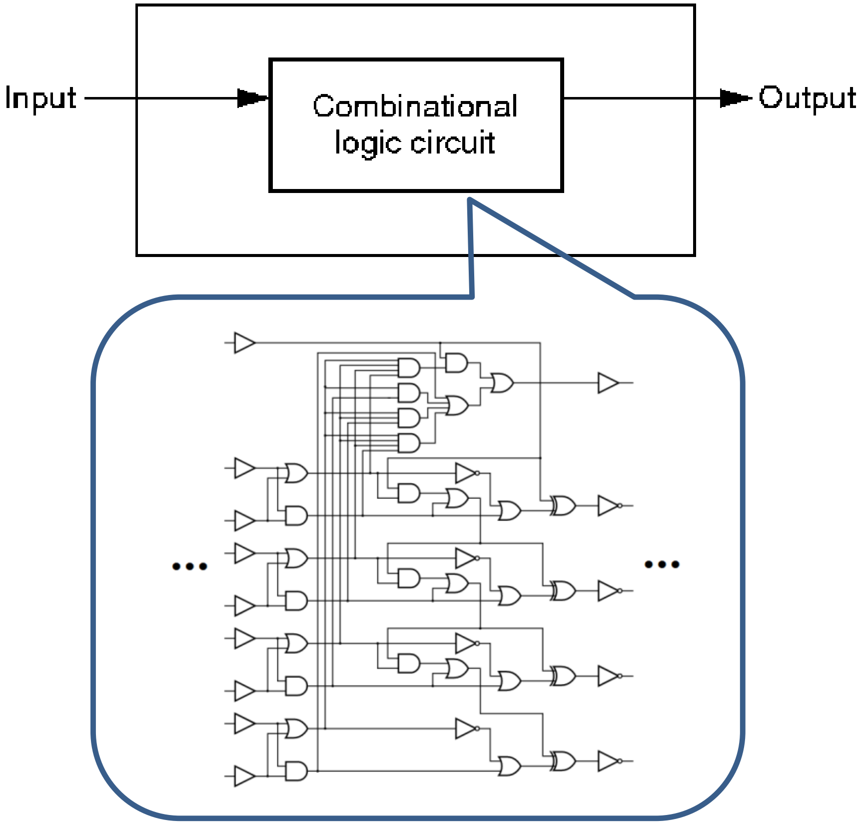
- Section 1 Sequential Circuits
- Section 2 Finite State Machine
- Section 3 Timing
- Section 4 Synthesis and Design Compiler

Outline

- ✓ **Section 1 Sequential Circuits**
  - ✓ **Introduction**
  - ✓ Syntax
  - ✓ Reset
  - ✓ Coding Style
  - ✓ Generate & For Loop

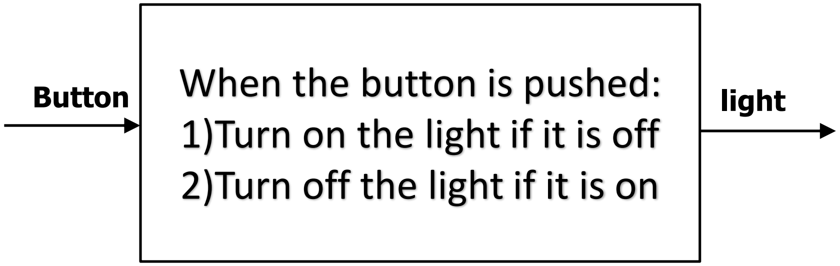
Motivation

- **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:

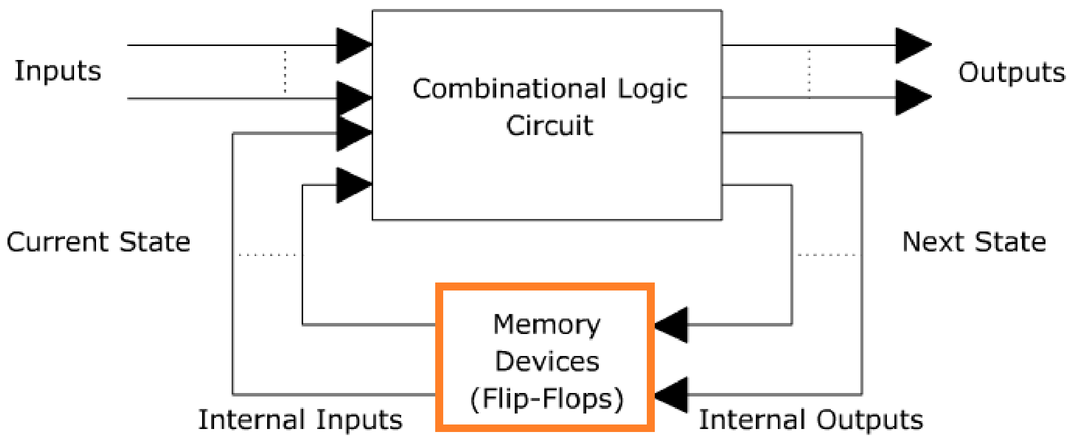


- What makes this circuit so different from we've discussed before?

"State"

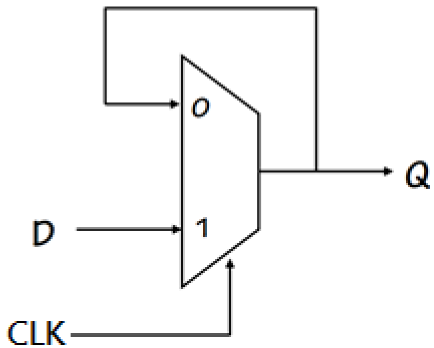
What is Sequential Circuit ?

- **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)



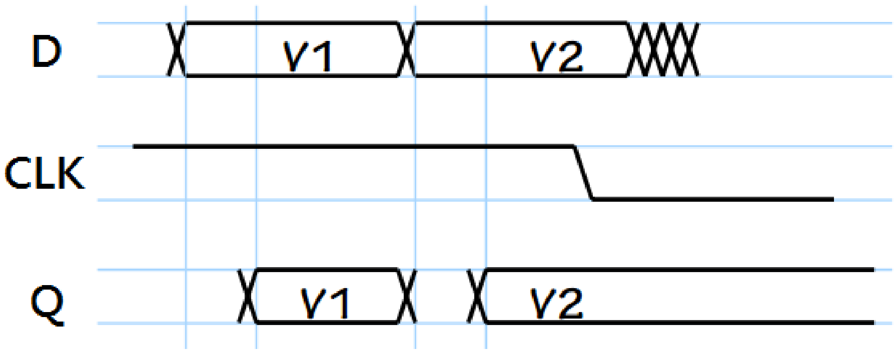
Flip-Flop Operation

✓ **Latch: level sensitive**



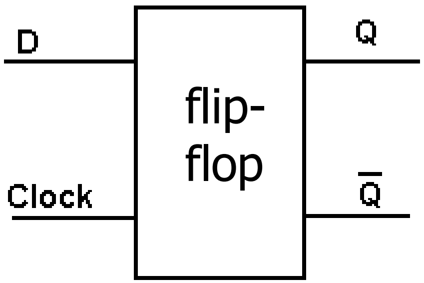
CLK	D	Q	Q'	
0	--	0	0	Q stable
0	--	1	1	
1	0	--	0	Q follows D
1	1	--	1	

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

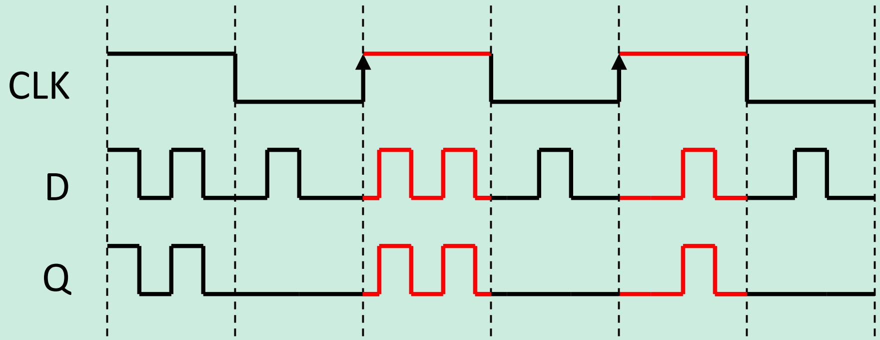
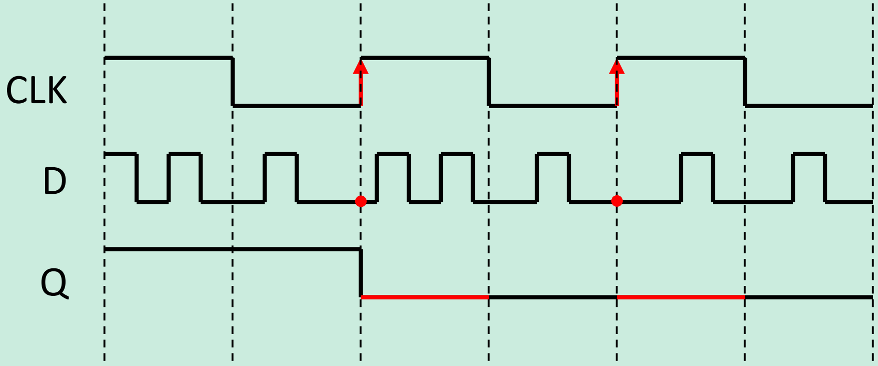


Flip-Flop Operation

- ✓ **D flip-flop: edge triggered**



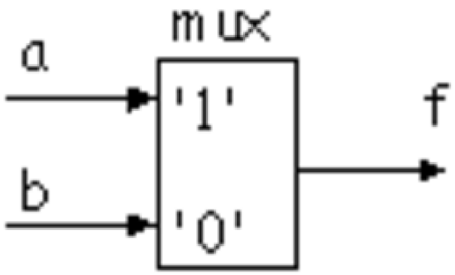
- ✓ **Positive latch v.s. positive D flip-flop**

Latch	D flip-flop
	

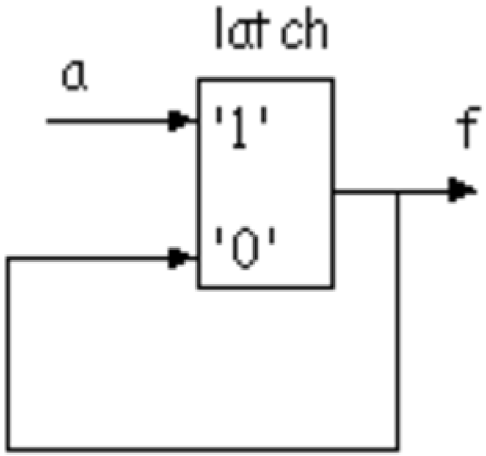
Coding Styles (1/3)

✓ **Example**

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




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



Coding Styles (2/3)

✓ **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
```

Coding Styles (3/3)

✓ Avoid combinational feedbacks

- Lead to unpredictable oscillated output
- NOT allowed

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

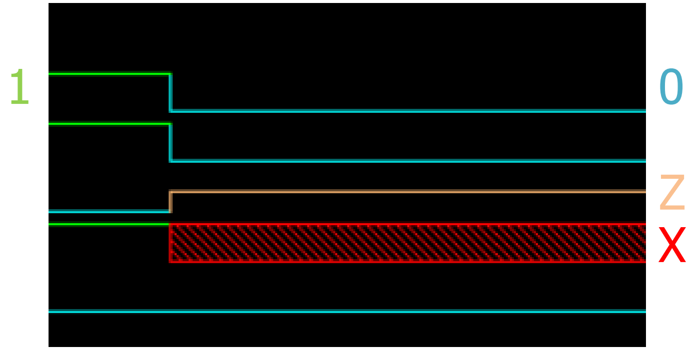
X
```
always @(*) begin
    a = a+1;
end
```

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
```

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

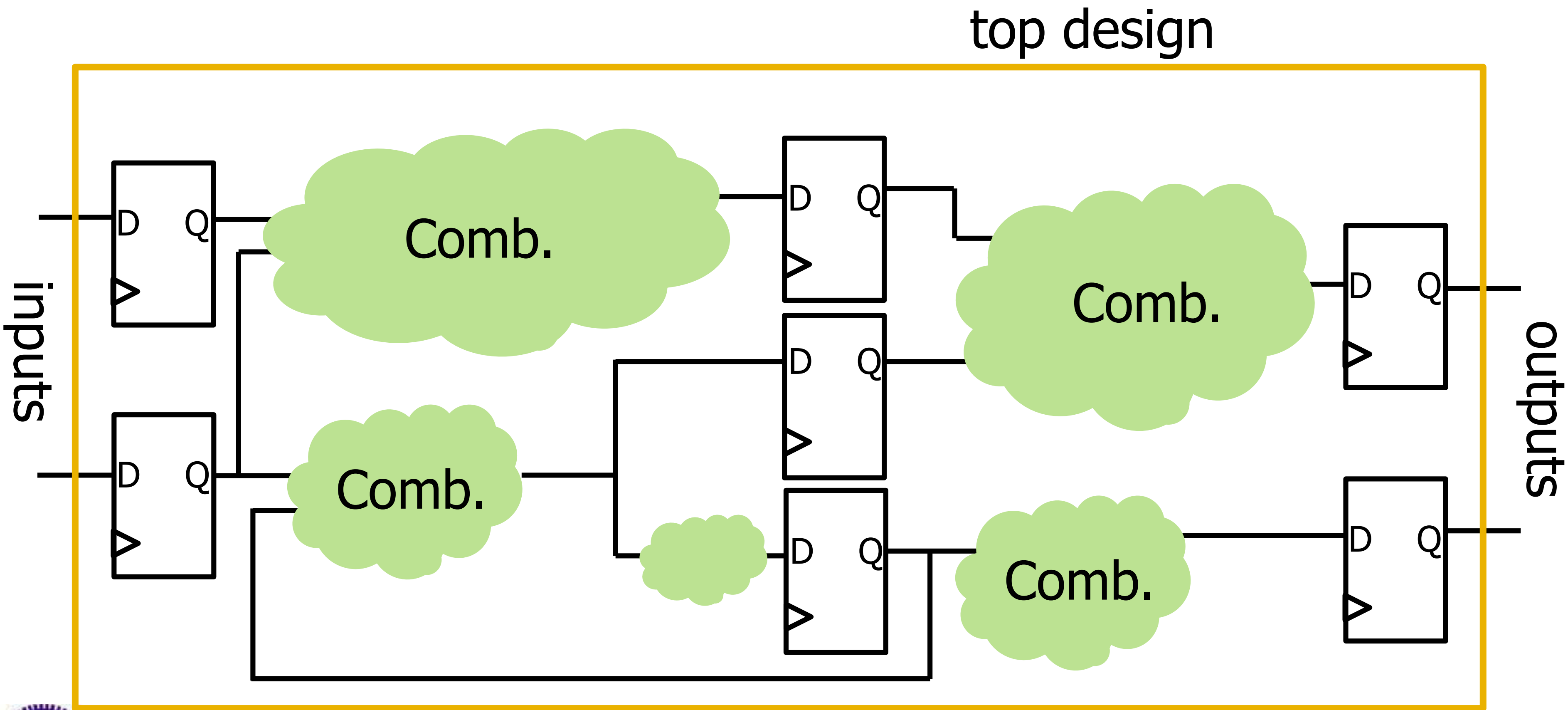
AND	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

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

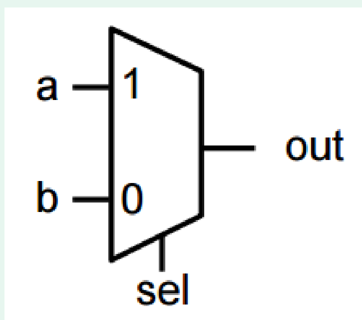
- Most computations are done by combinational circuit
- Sequential elements are used for storage

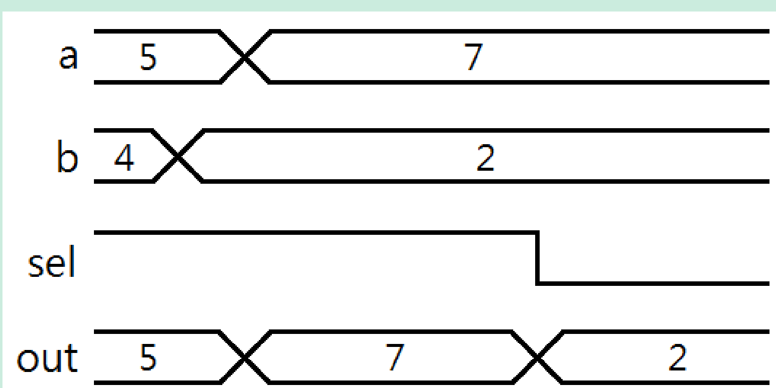


Combinational v.s. Sequential

Combinational

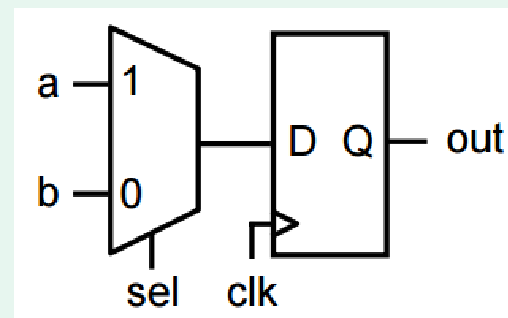
```
always@(*)
begin
   if(sel) out = a;
   else    out = b;
end
```

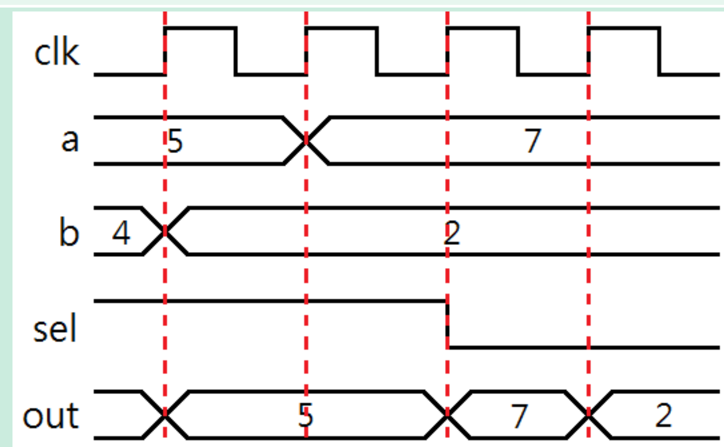




Sequential

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





Outline

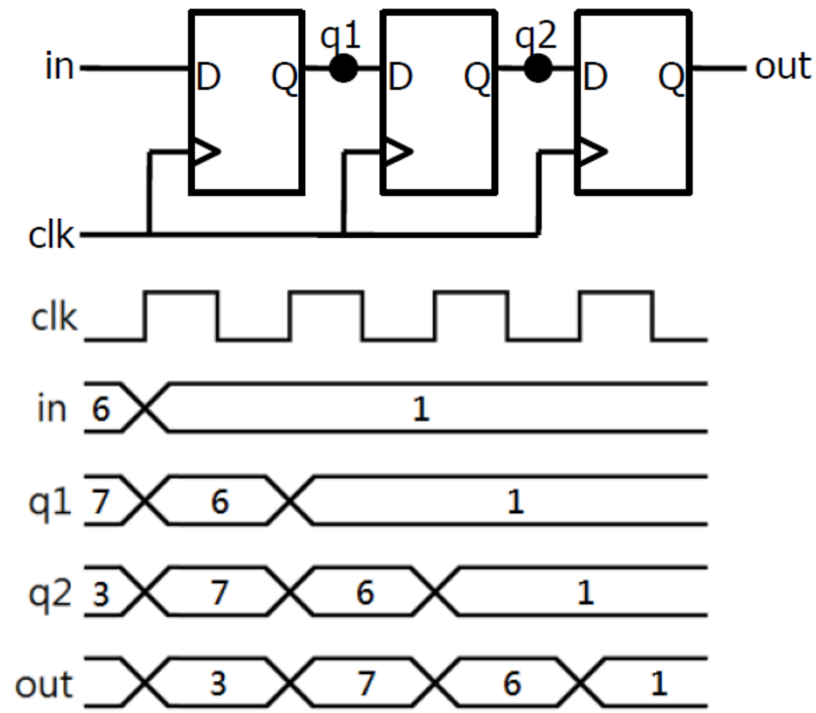
- Section 1 Sequential Circuits
  - Introduction
  - **Syntax**
  - Reset
  - Coding Style
  - Generate & For Loop

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
```



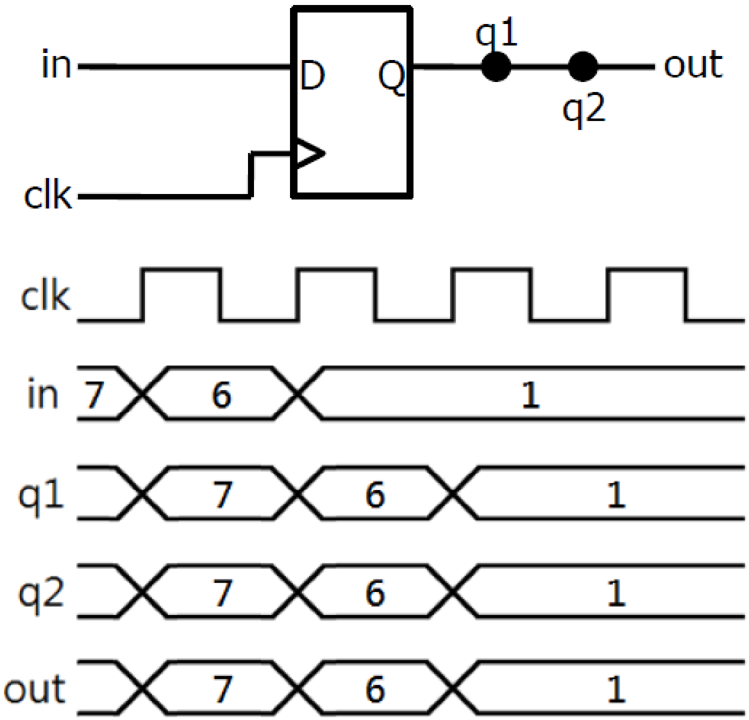
Assignment in Sequential Circuit

✓ 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
```



Coding Styles

- ✓ **Sequential blocks should only use "<=" assignments**

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

- ✓ **Combinational blocks should only use "=" assignments**

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

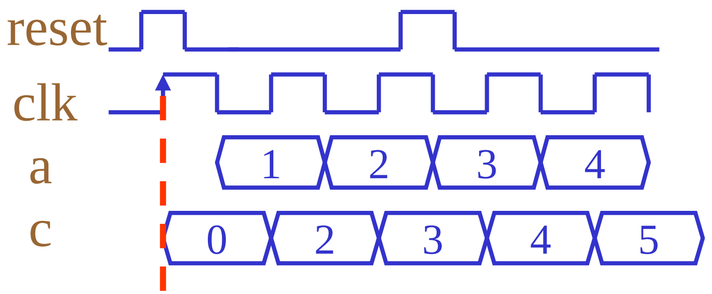
Outline

- **Section 1 Sequential Circuits**
  - Introduction
  - Syntax
  - **Reset**
  - Coding Style
  - Generate & For Loop

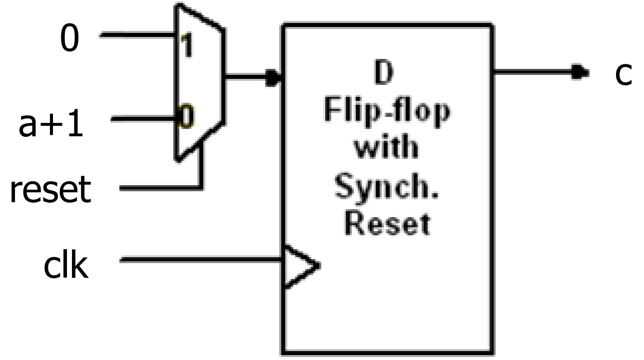
Synchronous Reset (1/2)

- **Register with synchronous reset**
  - Syntax: `always@(posedge clk)`

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

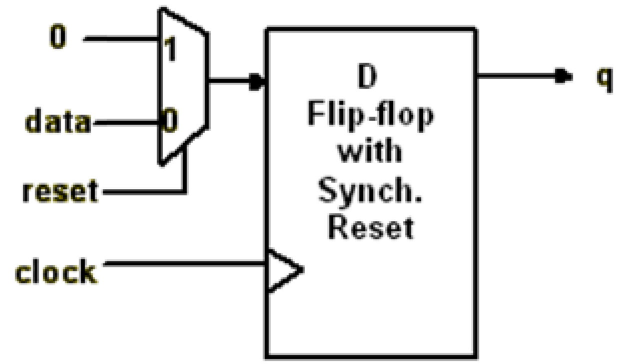


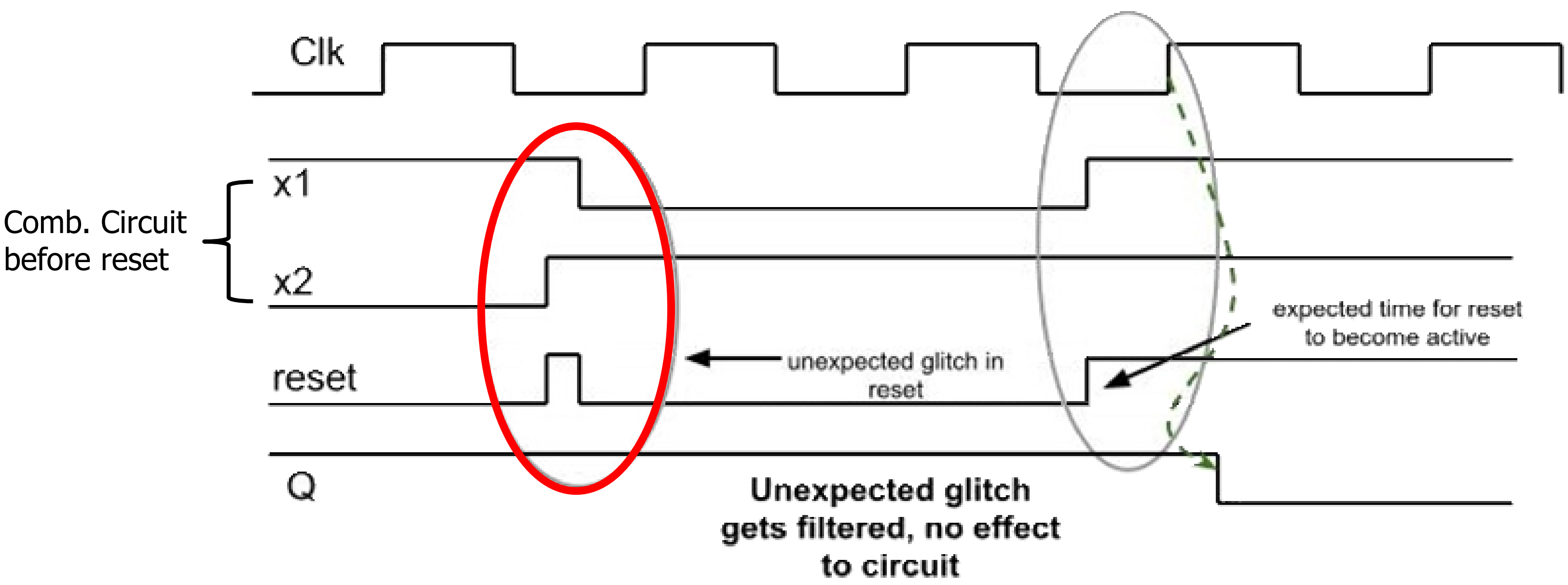
- **Advantages**
  - Glitch filtering from reset combinational logic
- **Disadvantages**
  - Can't be reset without clock signal
  - May need a pulse stretcher
    - Guarantee a reset pulse wide enough
  - Larger area
  - Increasing critical path



Synchronous Reset (2/2)

- ✓ **Advantage: glitch filtering**

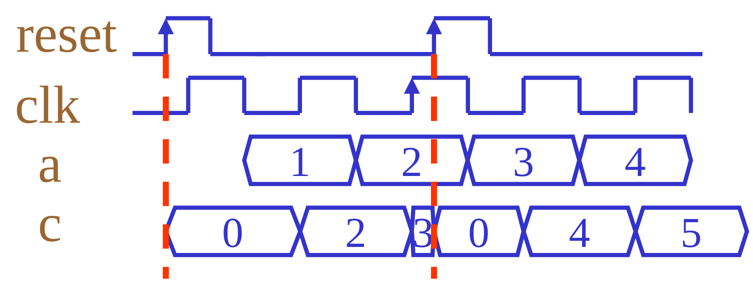




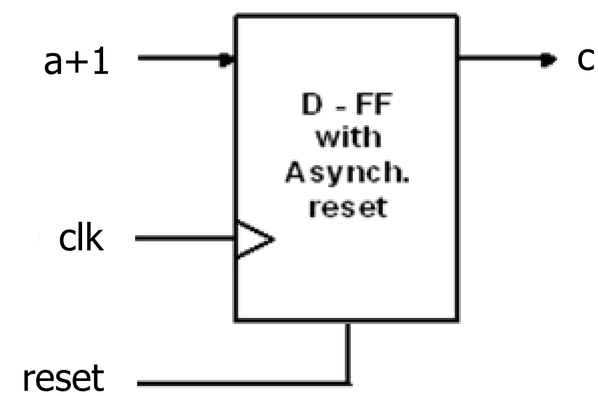
Asynchronous Reset

- ✓ **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

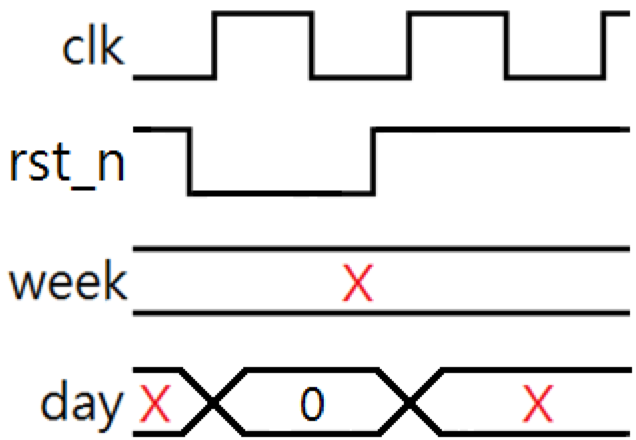


Coding Styles

- Reset all signals to avoid unknown propagation

X

```
always @(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
```



- Avoid conditional resets

X

```
always @(posedge clk or posedge reset or posedge a) begin
    if (reset || a) q <= 0;
    else …
    …
end
```

Outline

- **Section 1 Sequential Circuits**
  - Introduction
  - Syntax
  - Reset
  - **Coding Style**
  - Non-synthesizable code

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

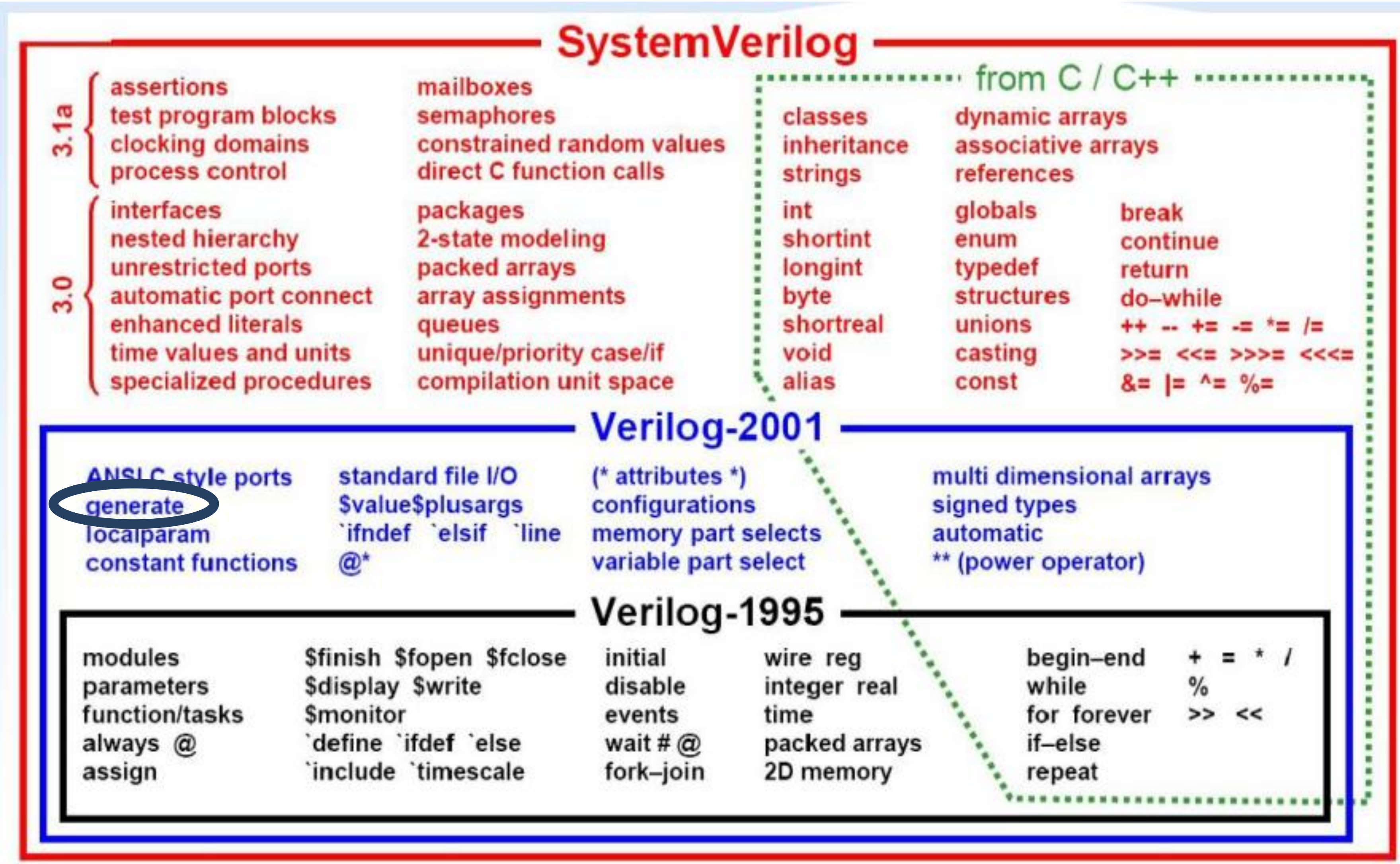
```
always @(posedge CLK) begin
   q2 <= in;
end
always @(posedge CLK) begin
   if(sel==0) out <= q2;
   else if(sel==1) out <= q3;
   else out <= out;
end
```

- ✓ **Use FSM (Finite State Machine)**

Outline

- Section 1 Sequential Circuits
  - Introduction
  - Syntax
  - Reset
  - Coding Style
  - 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
3.0
interfaces
nested hierarchy
unrestricted ports
automatic port connect
enhanced literals
time values and units
specialized procedures
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
++ -- += -= *= /=
>>= <<= >>>= <<<=
&= |= ^= %=
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;
integer i;
always @(posedge clk) begin
  for (i = 0; i < 3 ; i = i + 1) begin: for_name
    temp[i] <= 1'b0;
  end
end
```

=

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

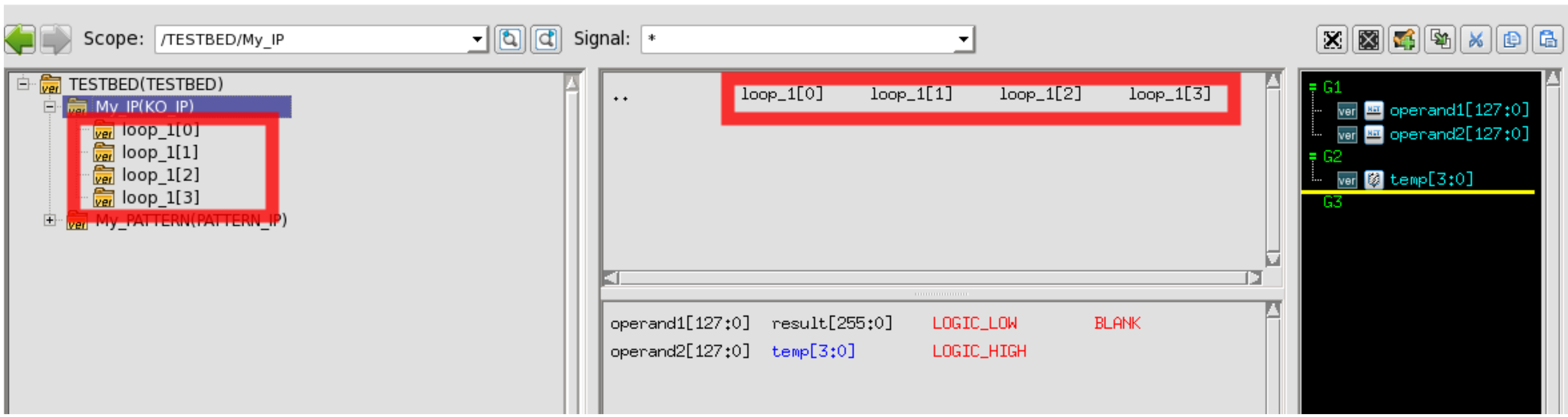
Regular for loop

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

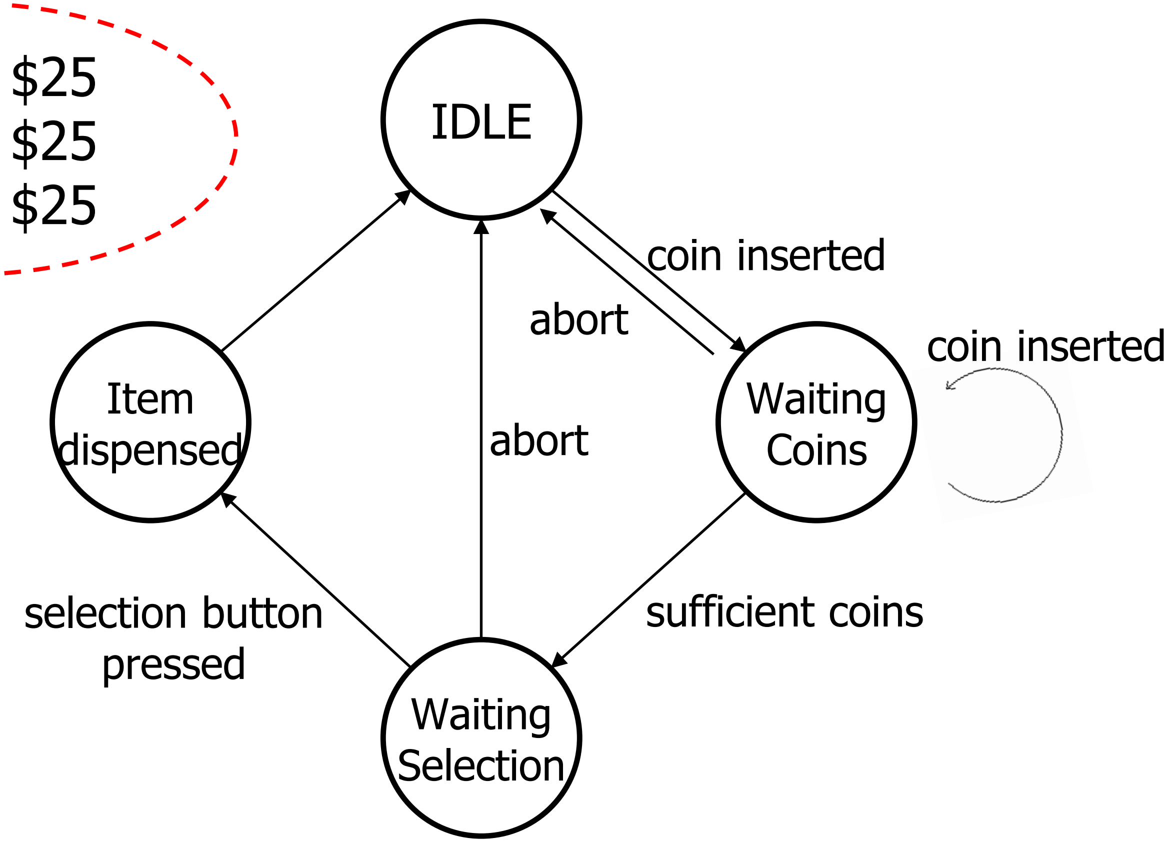
Outline

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

Finite State Machine

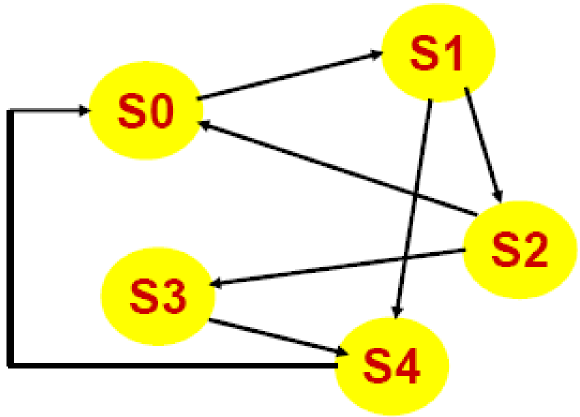
✓ **Example: Vending machine**

Coke $25
Pepsi $25
Sprite $25



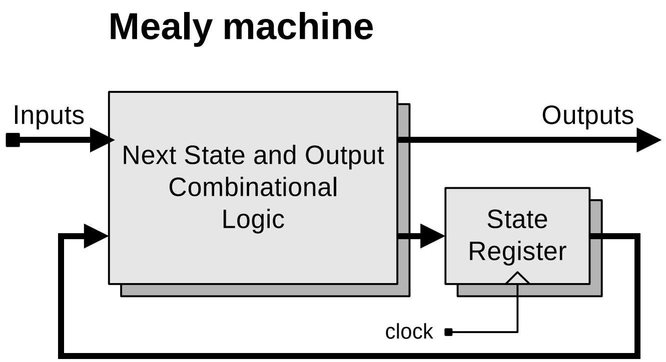
Finite State Machine

- **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**

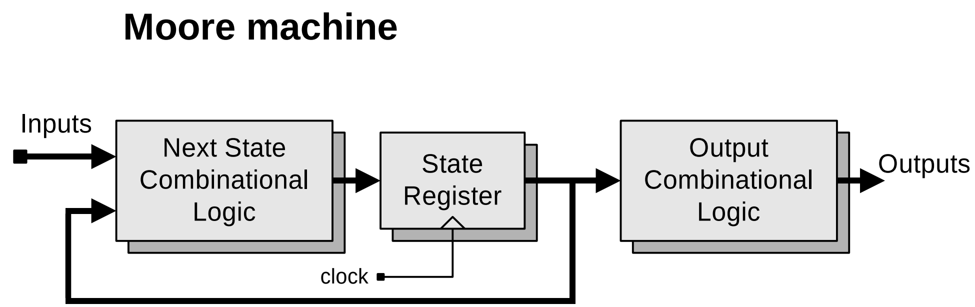


Mealy and Moore Machines

- **Mealy machine**
  - The outputs depend on the current state and inputs
- **Moore machine**
  - The outputs depend on the current state only



output = f (In, CS)



output = f (CS)

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

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

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
```

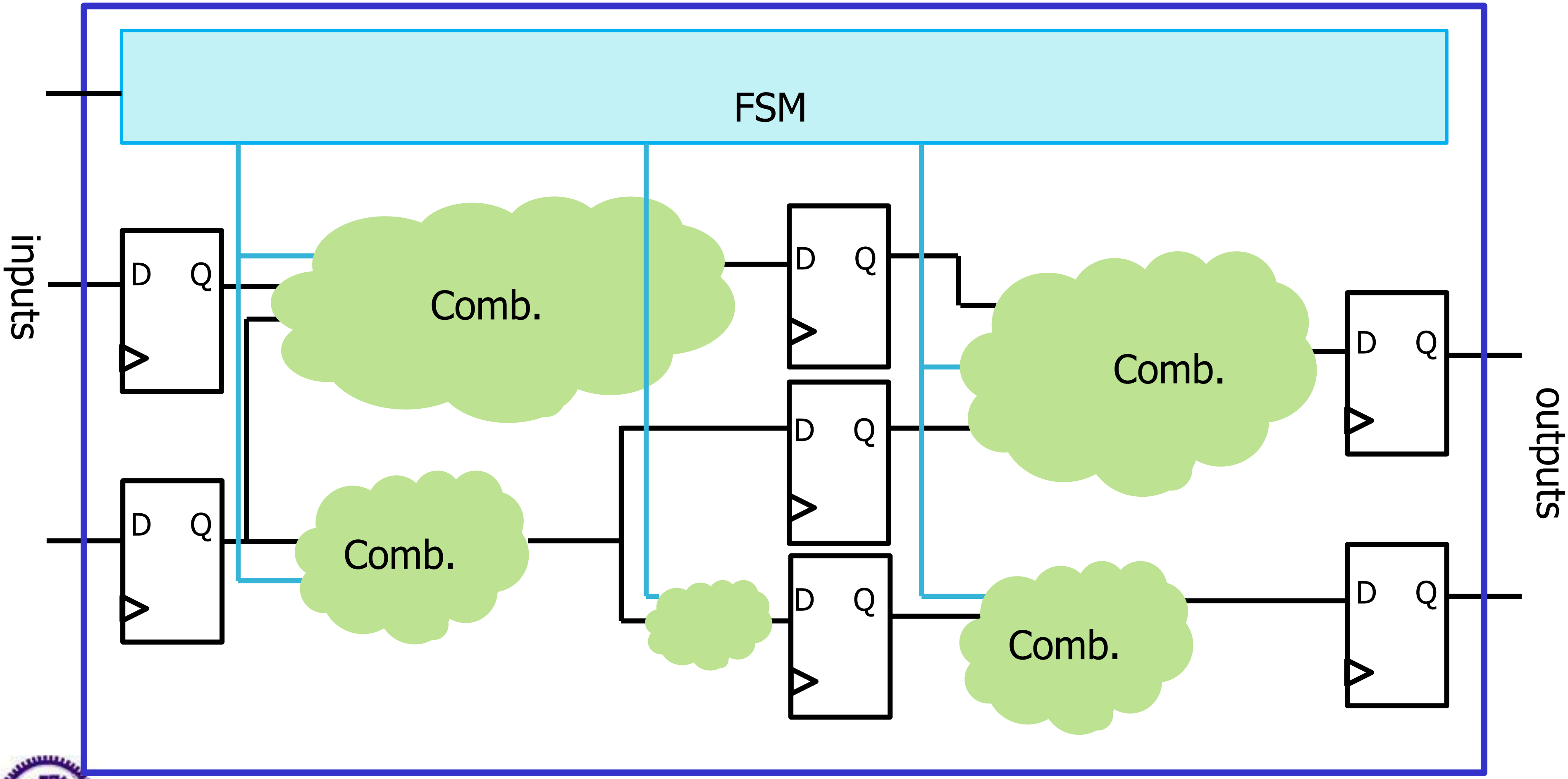
If it's not full case and without default case, latch would be incurred!

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**



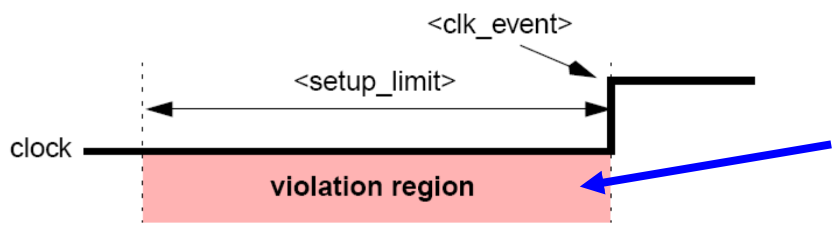
Outline

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

Timing Check (1/3)

✓ Setup time check

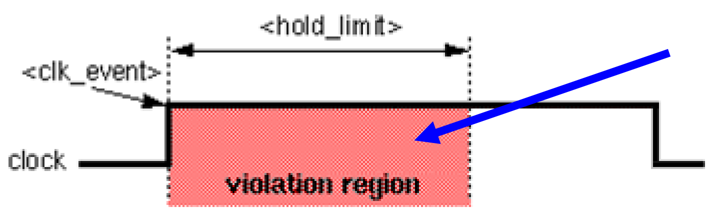
- The $setup system task determines whether a data signal remains stable for a minimum specified time before a transition in an enabling, such as a clock event.



data can’t be changed
in this red region

✓ Hold time check

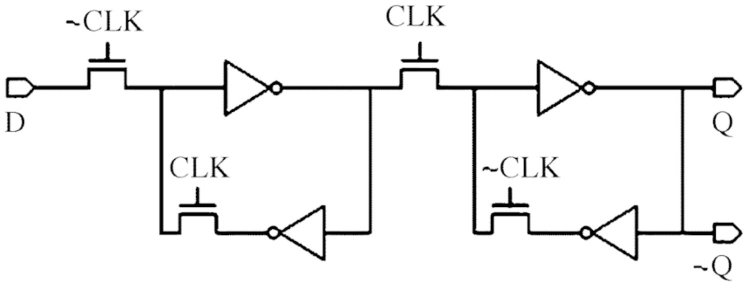
- The $hold system task determines whether a data signal remains stable for a minimum specified time after a transition in an enabling signal, such as a clock event.

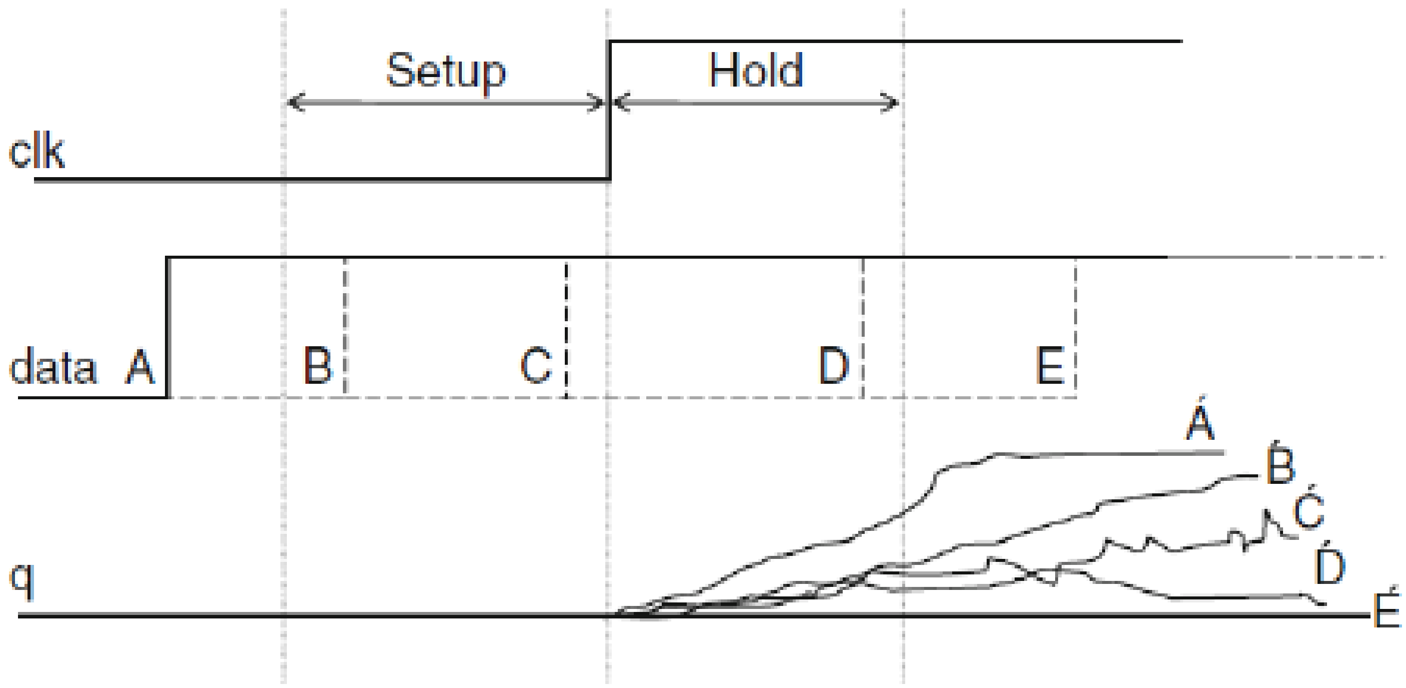


data can’t be changed
in this red region

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**

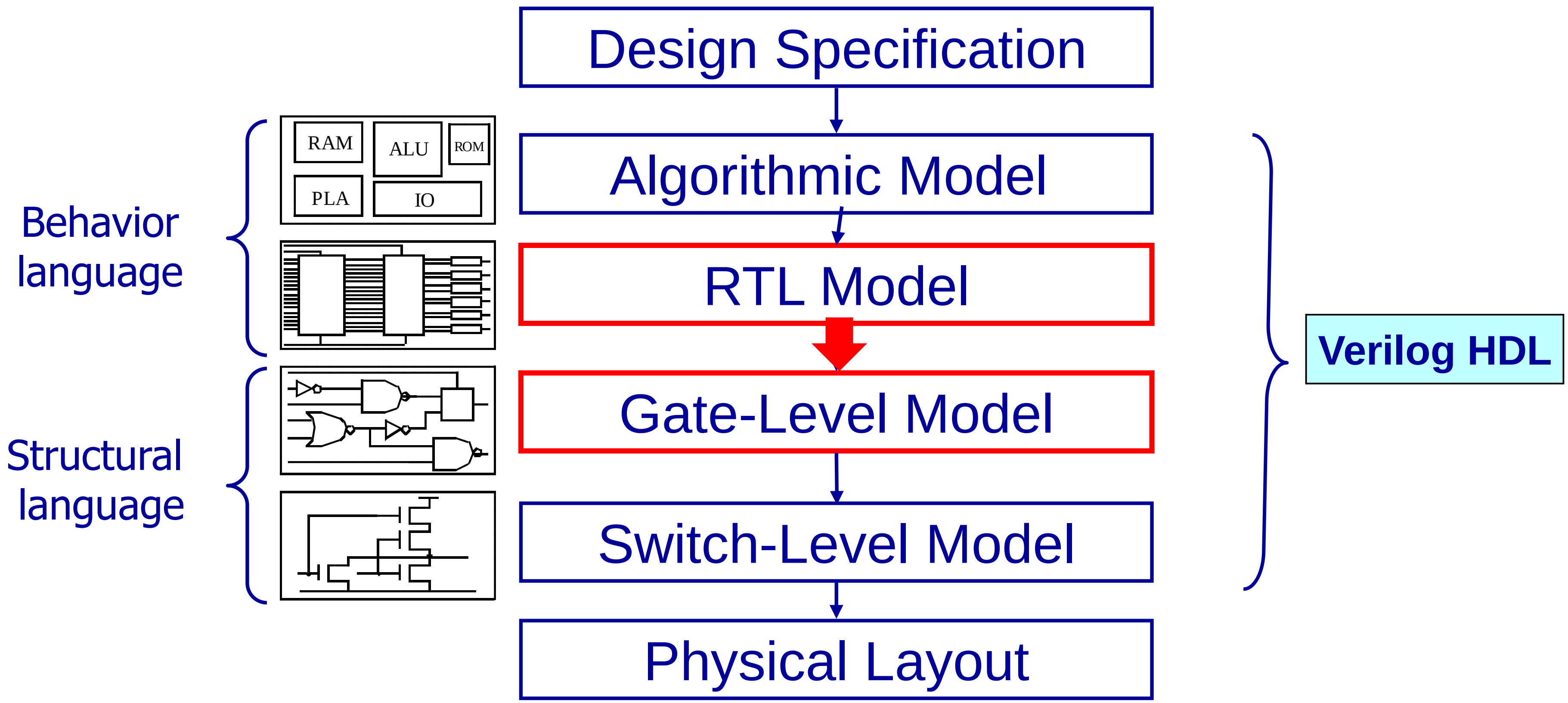
Slacks should be MET!
(non-negative)

```
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**

Recall: Design Flow



Logic Synthesis

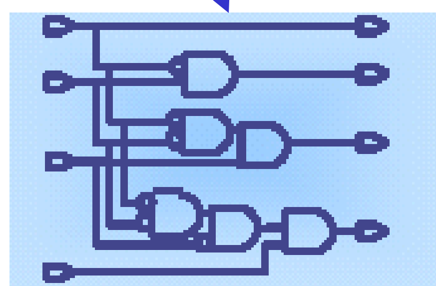
✓ Logic synthesis

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

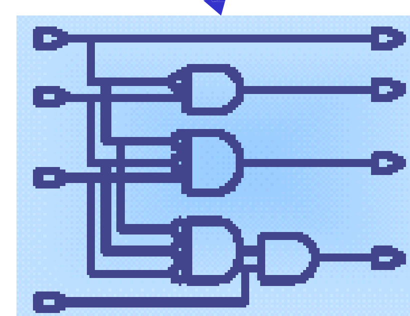
```
assign avg=sum/total;
always_ff @(posedge clk)
begin
    sum=sum+score*weight;
end
```

HDL Source

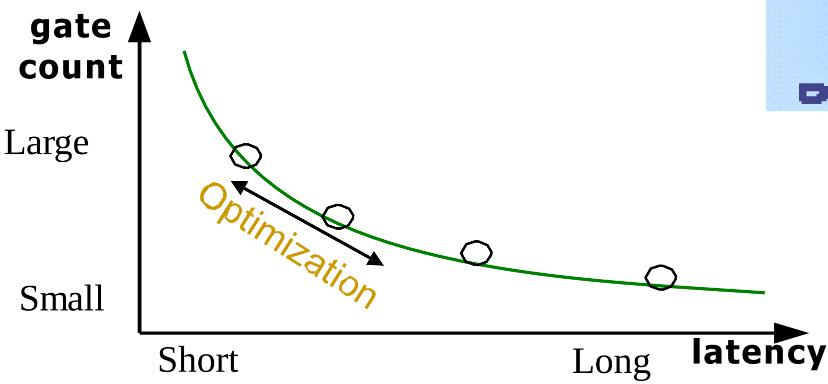
Translate



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.