Welcome to The Logic Design Lab!

Fall 2021
Midterm Review

Prof. Chun-Yi Lee

Department of Computer Science
National Tsing Hua University

Announcements

- Lab 3
  - Basic questions demonstration on **10/21/2021 (Thu)**
  - Advanced questions and FPGA demonstration due on **10/28/2021 (Thu)**
- Lab 4
  - Lab 4 basic questions demonstration on **11/2/2021 (Tue)**
- Midterm Exam
  - Computer-based exam on **11/4/2021 (Thu)**
- Final Project Proposal (5%)
  - Please submit your proposal (1~2 pages) by **12/13/2021 (Mon)**
  - Briefly describe your project contents
  - Be creative!

Agenda

- Gate-Level Design and Basic Concepts
- Data Flow Modeling
- Behavioral Modeling
- Sequential Circuits
- Debounce and One-Pulse Circuits
- FPGA Implementation

Today's class will help you:

1. Review the contents of this course

Primitive Logic Gates

- Basic Logic gates are provided as pre-defined primitives
  - **and**, **or**, **xor**, **nand**, **nor**, **xnor**, **not**

- The first terminal is the output and the rests are the inputs
  - How to use them? Easy!

```
wire    OUT, IN, IN1, IN2;

and     a1(OUT, IN1, IN2);
nand    na1(OUT, IN1, IN2);
or      or1(OUT, IN1, IN2);
nor     nor1(OUT, IN1, IN2);
xor     x1(OUT, IN1, IN2);
xnor    nx1(OUT, IN1, IN2);
not     n1(OUT, IN);
```

Smart Primitives

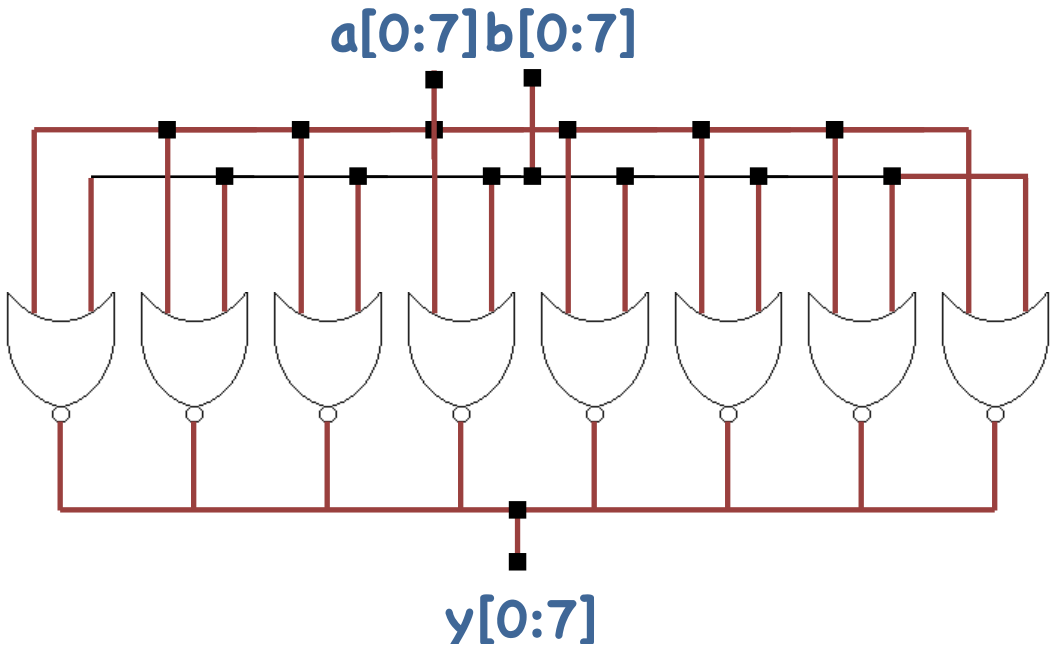
```
module      nand3 (O, A1, A2, A3);

    input   A1, A2, A3;
    output  O;

    nand    NAND1 (O, A1, A2, A3);

endmodule
```

Remember to give a name for it!

- Same primitive can be used to describe for **arbitrary number** of inputs
- Except for "**not**", **which takes ONLY ONE input**
- **Not gate can generate multiple number of outputs**
  - E.g., **not U (O1, O2, …, ON, IN),** where **O1~ON** equal the inverse of **IN**

Array of Instances

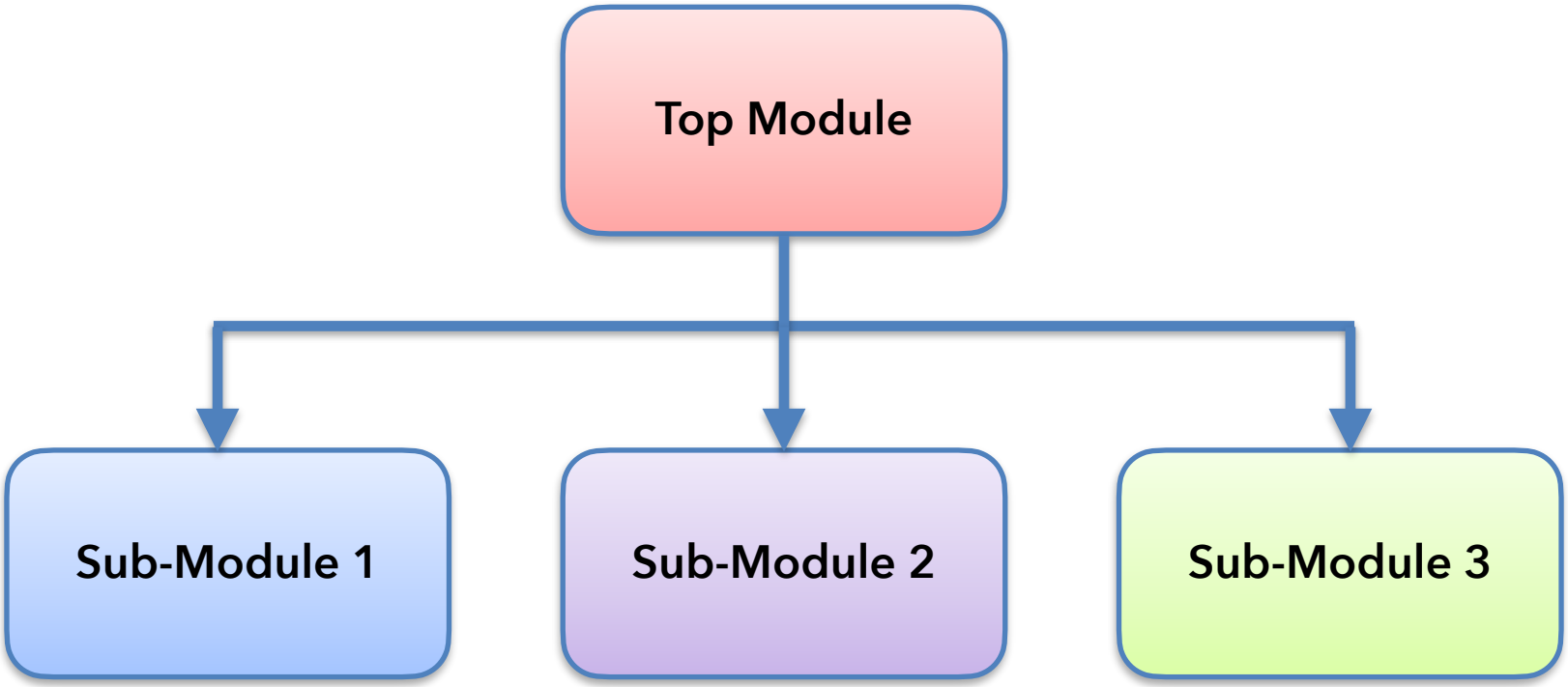
```
module array_of_nor (y, a, b);
  input   [7:0]      a, b;
  output  [7:0]      y;

  nor     Nor_Array [7:0] (y, a, b);

endmodule
```



*Pictures cited from Prof. Ting-Chi Wang's 2013 slides for education purpose only

Hierarchy of Modules (1/2)

- A typical Verilog design consists of multiple hierarchies of modules
- A module may be implemented by several sub-modules
- The outermost module is called the "**top module**"



Port Mapping

- Port mapping can be done in either of the following two ways

Connect by ordered lists

```
module red (x1, x2, y1, y2);
    input  x1, x2;
    output y1, y2;
    wire   w1, w2, w3;
    green  U1 (x1, x2, w1, w2, w3);
    blue   U2 (w1, w2, w3, y1, y2);
endmodule
```

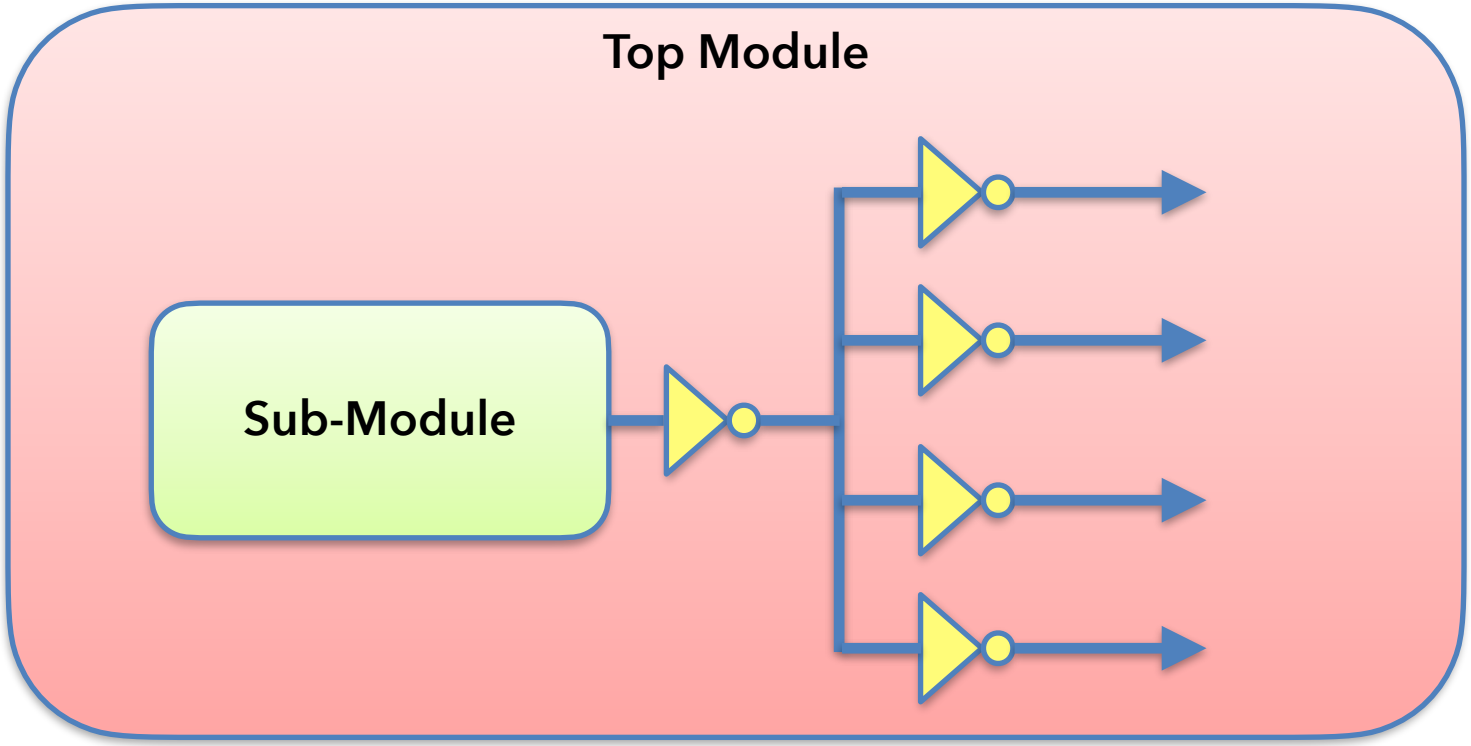
Connect by port names

```
module red (x1, x2, y1, y2);
    input  x1, x2;
    output y1, y2;
    wire   w1, w2, w3;
    green  U1 (.b(x2), .c(w1), .a(x1), .d(w2), .e(w3));
    blue   U2 (.i(y1), .j(y2), .f(w1), .g(w2), .h(w3));
endmodule
```

Port order can be random!

Fan-outs

- Fan-outs means the number of output wires that a signal drives
- Fan-outs may cause additional delay and power consumption
- The fan-out gates can be any types of logic gates



An example of fan-out == 4

Representation of Numbers

- Numbers are represented **UNSIGNED** by default
- Syntax:
  - **<# of bits>'<Radix><Number>**

# of bits The number of BINARY bits that the number is comprised of. (default 32 bits)

Radix Radix of the number

b or B:	Binary	E.g., 8'b1011_0011
d or D:	Decimal	E.g., 4'd5
o or O:	Octal	E.g., 3'o5
h or H:	Hexadecimal	E.g., 5'h1A

Number Any legal number in selected (**# of bits**) & (**Radix**)

Agenda

- Gate-Level Design and Basic Concepts
- Data Flow Modeling
- Behavioral Modeling
- Sequential Circuits
- Debounce and One-Pulse Circuits
- FPGA Implementation

Today's class will help you:

1. Review the contents of this course

Value Set

- The net types in Verilog can be one of these four states

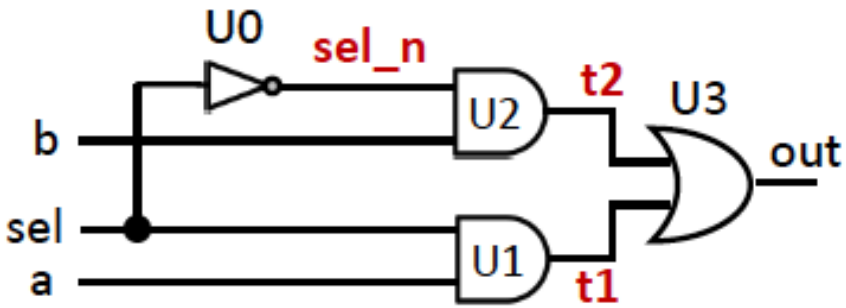
Net	Reg	Representation
0	0	Logic zero, false condition
1	1	Logic one, true condition
x		Unknown logic value
z		High impedance, floating state

- Please only set the value of a reg to be either **0** or **1**
  - Registers of real circuits do not have values of either **x** or **z**

- It is a better habit to CLEARLY specify the values in Verilog

Net Data Type

- Net type is basically the wires
  - Declare by the keyword **wire (CAUTION: NO "wire = 0")**
- Can be used to connect different modules
- No storage functionality
- Default value is **z**




```
module MUX(out, a, b, sel);

output out;
input a,b,sel;

wire sel_n,t1,t2;

not U0(sel_n,sel);
and U1(t1,a,sel);
and U2(t2,b,sel_n);
or U3(out,t1,t2);

endmodule
```

Register Data Type

- Register data type represents a data storage element **or a net**
  - Depends on how they are used in the **always** block in modules
  - Does not represent anything in a testbench
  - Declare by the keyword **reg (CAUTION: NO "reg = 0")**
- Can have storage functionality
  - If it is a data storage element
- Default value is **x**
- **reg** assignment (**=**) only occurs in either an **initial** or an **always** block

Register Data Type in Testbench

- Variables of reg data type in testbenches are not synthesized
- They are simply variables to be fed into your modules

```
`timescale 1ns / 1ps                    // Simulation Unit / Accuracy
module      test_Nand_Latch_1;          // Testbench module
reg         preset, clear;              // Inputs should be declared as reg
wire        q, qbar;                    // Outputs should be declared as wire

 Nand_Latch_1 M1 (q, qbar, preset, clear);     // Instantiate YOUR DESIGN module

 always
 begin                                         // always condition: The description always happens
   #20       clear = !clear;                   // The value of clear inverts every 20 ns
 end

 initial                                       // Initial conditions
 begin
           preset = 1'b0;   clear = 1'b1;
   #10                                         // Units of "Simulation Units".  In this case, 10ns
           preset = 1'b1;
 end

endmodule
```


*Pictures cited from Prof. Ting-Chi Wang's 2013 slides for education purpose only

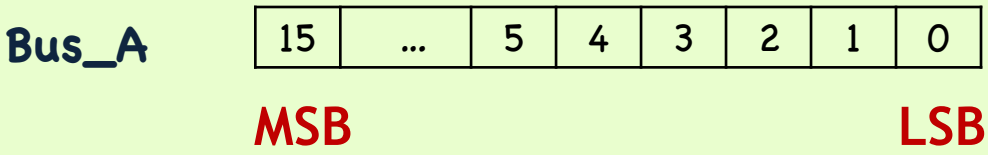
Vectors

- Both data types can be declared as vectors
  - Similar to what we have done for inputs and outputs
  - Examples
    - reg [7:0] reg_vector1, reg_vector2;
    - wire [5:0] data_bus;
- You can assign only part of a vector to another

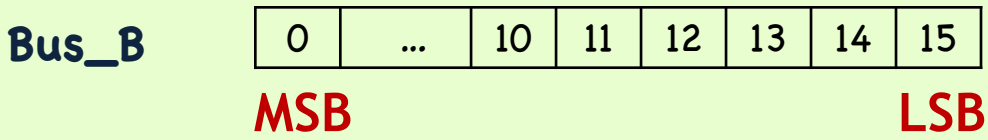
```
wire    [7:0] a, b, y;

assign y[7:4] = a[3:0];
assign y[3:0] = b[7:4];
```

MSB and LSB

- Vector definition can determine its Most Significant Bit (**MSB**) and Least Significant Bit (**LSB**)
- Important for arithmetic operations
  - E.g., **wire** [15:0] Bus_A;



  - E.g., **wire** [0:15] Bus_B;



- There is a difference between **Bus_A+1'b1** and **Bus_B+1'b1**

Verilog Modeling

- Gate level modeling

```
and     a1(OUT, IN1, IN2);
nand    na1(OUT, IN1, IN2);
or      or1(OUT, IN1, IN2);
```

- Data flow modeling

E.g., **assign** a[7:0] = b[7:0] & c[7:0]; (bitwise AND)

- Behavior modeling **(in always blocks)**

E.g.,

```
if (sel == 1'b1)    begin
                S = A;
end
else                begin
                S = B;
end
```

Data Flow Modeling

- For combinational circuits only
- Similar to logic expressions
- Easier for development
- Need to be synthesized to the gate level
- Usage
  - Continuous assignment
  - Conditional assignment
  - Operators

Continuous Assignment

- Syntax:
  - **assign** [the name of **lvalue**] = **rvalue**;
- Assignment can be done for a single wire or a bus
  - **E.g.,** **assign** a[7:0] = b[7:0];
- **lvalue** must be of type **wire** or **output** (but not **reg** type)
- **rvalue** can be an **expression**
  - **E.g.,** **assign** a[7:0] = b[7:0] & c[7:0]; (bitwise AND)
- Cascade is allowed
  - **E.g.,** **assign** {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;

Multiple Assignments

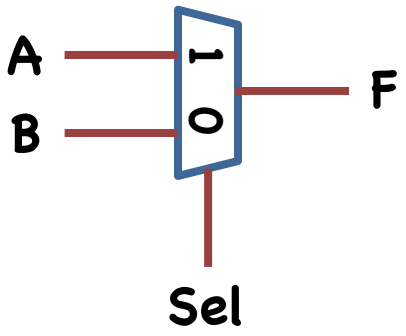
- You can do multiple assignments in a single line
  - **assign** S1 = A1 & B1, S2 = A2 | B2, S3 = A3 + B3;
  - However, it is a better habit to separate different assignment to different lines

Conditional Operator

- Evaluation depending on the value of condition
- Syntax:
  - (Condition) ? (Expression 1) : (Expression 2);
    TRUE FALSE

```
module simple_mux (F, A, B, Sel);
  input   A, B, Sel;
  output F;

  assign F = (Sel == 1'b1) ? A : B;

endmodule
```

Verilog



Schematic

Concatenation and Replication

- Assignment can be done by concatenation and replication

- Concatenation
  - Suppose **A = 1'b1**, **B = 3'b010**
  - assign **Y** = { **A, 4'hc**, **B** }; // *Result Y is 8'b1_1100_010*
  - assign **Y** = { **A**, **B[0]** }; // *Result Y is 2'b10*

- Replication
  - assign **Y** = { 4{**A**} }; // *Result Y is 4'b1111*
  - assign **Y** = { {2{**A**}}, {2{**B**}}, **A** }; // *Result Y is 9'b1_1_010_010_1*

***** Please note that **Y** should be declared with sufficient bits to accommodate **rvalue**

Agenda

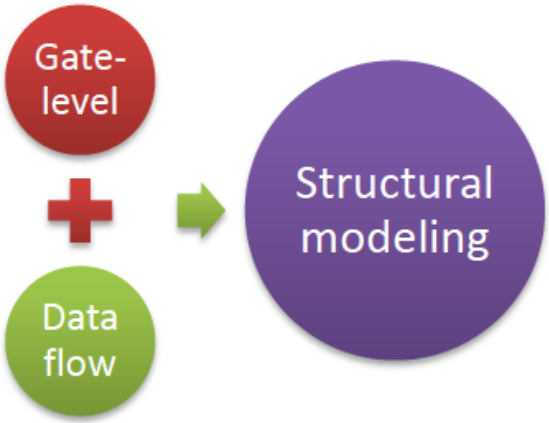
- Gate-Level Design and Basic Concepts
- Data Flow Modeling
- Behavioral Modeling
- Sequential Circuits
- Debounce and One-Pulse Circuits
- FPGA Implementation

Today's class will help you:

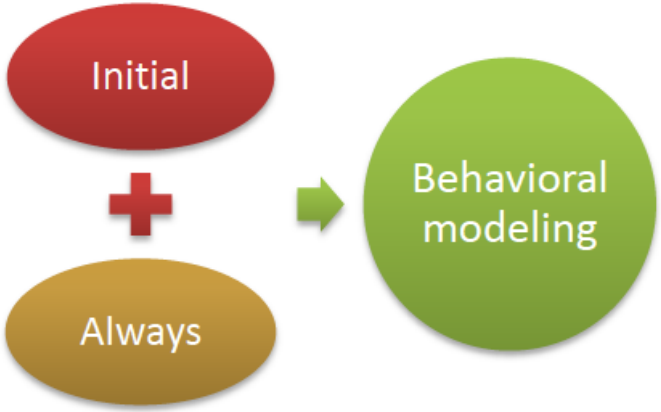
1. Review the contents of this course

Behavioral Modeling

- High level description
- Modeling a circuit by its behaviors
- Similar to C++ programming
- Behavioral modeling includes both **combinational** and **sequential** parts



Combinational only



Combinational and Sequential

Structural vs. Behavioral Modeling

Structural modeling

Behavioral modeling

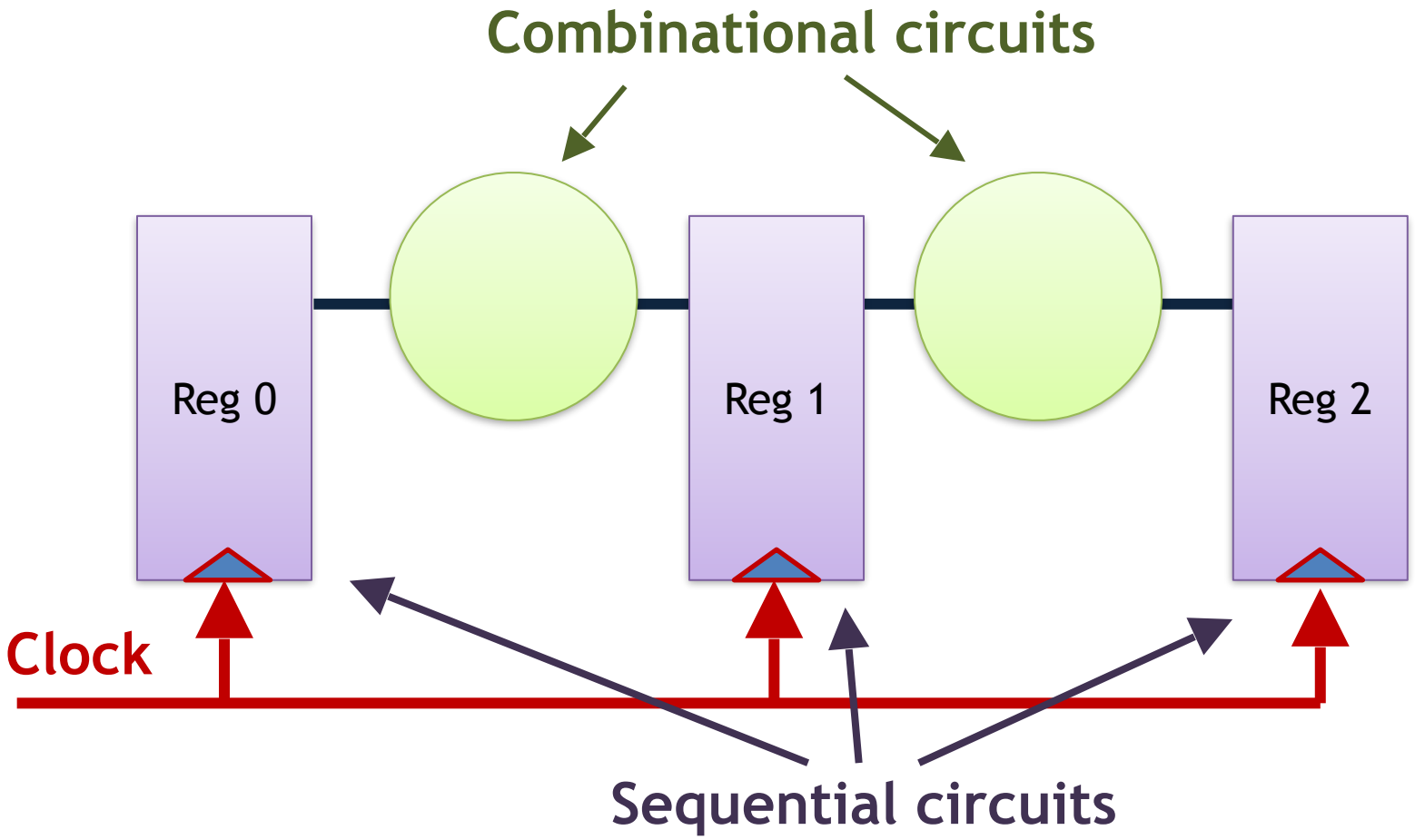
```
module My_Module(...);
  ...
  assign  O1 = A+B;          // 1. continuous assignment
  and     N1(O2, C, D);      // 2. Instantiation of a primitive
  MUX     M1(O3, Sel, F, G); // 3. Instantiation of a module

  always @ (...)             // 1. always block
    begin ... end

  initial                    // 2. initial block
    begin ... end            // initial only used in testbench

endmodule
```

Behavioral Constructs

- Two constructs: **initial** and **always**
- Similar point:
  - lvalue has to be of **reg** data type
  - Has **begin** and **end**
- **initial**:
  - Used in testbench only
  - Only run once when the testbench begins
- **always**
  - Used both in design and testbench
  - Repeated execution

Register Transfer Level

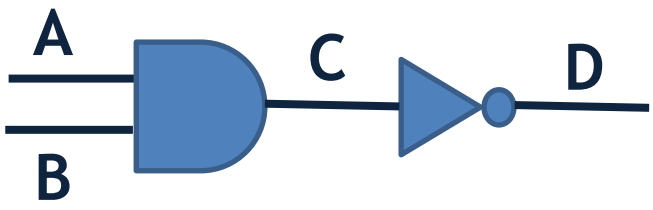
- Describes the behavior of combinational circuits between registers



Blocking and Non-blocking

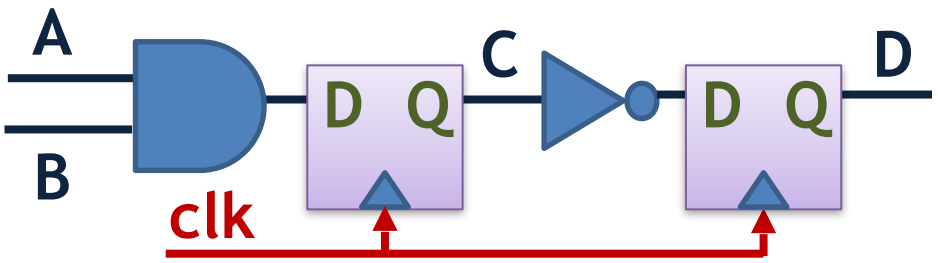
Execute in Order

```
always @ (A or B or C)
  begin

   //Blocking Assignment
        C = A & B;
        D = !C;
  end
```



Execute in Parallel

```
always @ (posedge clk)
  begin

   //Non-blocking assignment
        C <= A & B;
        D <= !C;
  end
```



- **NEVER** use blocking and non-blocking assignment in the **SAME** always block

Caution for Always Block

- Combinational circuit: use blocking assignment (=) only

```
always @ (A or B or C)
  begin
   //Blocking Assignment
          C = A & B;
          D = !C;
   end
```

- Sequential circuit: use non-blocking assignment (<=) only

```
always @ (posedge clk)
  begin

   //Non-blocking assignment
          C <= A & B;
          D <= !C;
   end
```

Sensitivity List

- Wakes up an always block and do the execution
- **For combinational circuit only**
- Separated by **or**
- Variables include:
  - Right hand side of "**=**"
  - Condition variables in **if**
  - Condition variables in **case**

```
always @ (A or B or C or Sel or Sel_Bus)
  begin
        C = A & B;

        if (Sel) begin
            D = !C;
        end

        case (Sel_Bus)
            ……
        endcase
  end
```

- No need of sensitivity list for **always @(*)**

Procedural Statements

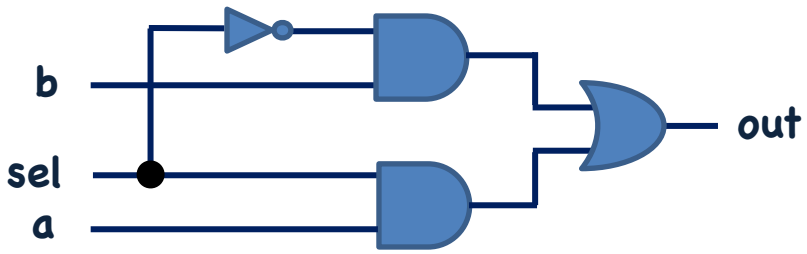
- Control operators similar to C++
- Not all of the operators can be used in your design
- Any operator used in the design must be **synthesizable**
  - However, you can use non-synthesizable operators in testbenches

Operator	Design	Testbench	Synthesized to
If-else	Yes	Yes	Mux
case	Yes	Yes	Mux or Decoder
for	No	Yes	N/A
while	No	Yes	N/A
repeat	No	Yes	N/A

If-Else Statement

```
if (condition1)
    begin
    <expression> ;
    end
else if (condition2)
    begin
    <expression> ;
    end
else
    begin
    <expression> ;
    end
```

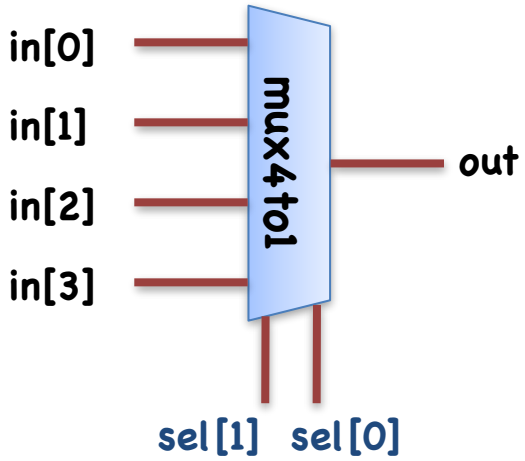
```
module MUX3(out, a, b, sel);

output    out;
input     a,b,sel;
reg       out;

always @(*) begin
    if(sel == 1'b1)
        out = a;
    else
        out = b;
end

endmodule
```



Synthesized to a **MUX**

Case Statement

```
case (condition)
  alternative1: <expression> ;
  alternative2: <expression> ;
  …
  default: <expression> ;
endcase
```



Synthesized to a **MUX**

```
module mux4to1 (out, in, sel);

output      out;
input [3:0] in;
input [1:0]  sel;
reg         out;

always @(*) begin
  case (sel)
    2'd0 :          out = in[0];
    2'd1 :          out = in[1];
    2'd2 :          out = in[2];
    default :       out = in[3];
  endcase
end

endmodule
```

Case Statement (Cont'd)

Din[1:0]	Dout[3:0]
00	0001
01	0010
10	0100
11	1000

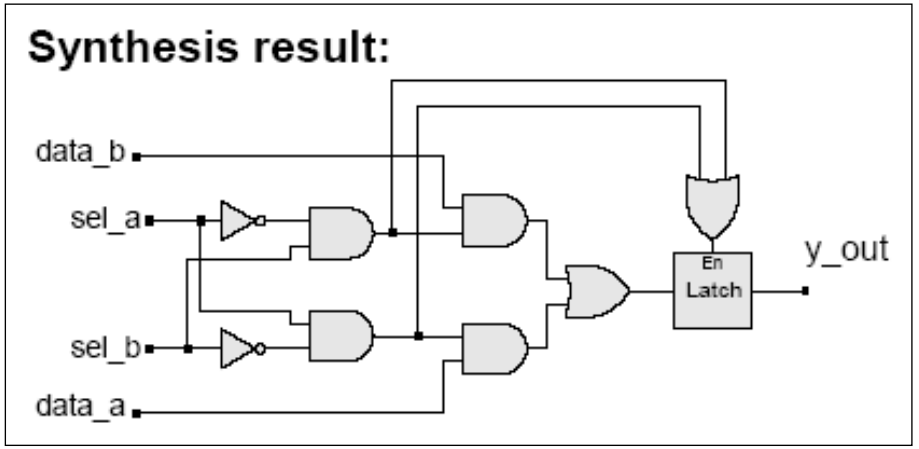
Synthesized to a **DECODER**

```
module mux4to1 (Dout, Din);

output [3:0]   Dout;
input    [1:0]   Din;
reg      [3:0]   Dout;

always @(*) begin
  case (Din)
    2'd0 :          Dout = 4'b0001;
    2'd1 :          Dout = 4'b0010;
    2'd2 :          Dout = 4'b0100;
    default :       Dout = 4'b1000;
  endcase
end

endmodule
```

Unintended Latch

- **Clearly specify** every situations in **if-else** or **case** statements
- Incomplete statements lead to **unintended latch**

```
always @ (sel_a or sel_b or
    data_a or data_b)
begin

    case ({sel_a, sel_b})
        2'b10: y_out = data_a;
        2'b01: y_out = data_b;
    endcase

end
```



*Pictures revised from Prof. Shih-Chieh Chang's 2010 slides for education purpose only

Loop Statements

- Not synthesizable
- **Used in testbench only**

For loop

```
reg [15:0] x;
integer i;

initial
begin
    x = 16'd0;
    for ( i = 0; i <=10;  i = i + 1 )
    begin
        x = x + 1'b1;
    end
end
```

Repeat loop

```
reg [15:0] x;


initial
begin
    x = 16'd0;
    repeat ( 16 )
    begin
        #2
        x = x + 1'b1;
    end
end
```

While loop

```
reg [15:0] x;


initial
begin
    x  = 16'd0;
    while ( x <= 20 )
    begin
        #2
        x = x + 1'b1;
    end
end
```

Constants

- Declare with keyword parameter
- Similar to **const** in C++
  - On the other hand, **`define** is similar to **#define** in C++
- Value does not change during simulation
- Can be used in vector declaration
  - Easier in project development and maintenance

```
parameter size = 16;
    reg [size-1:0] a;// vector declaration
parameter b = 2'b01;
parameter av_delay = (min_delay + max_delay) / 2;
```

Delay Control Operator

- # followed by units of time
- Specify the delay in terms of units specified by `timescale
- NOT synthesizable, only used in testbench
- In real circuits, delay is realized by buffers (two inverters)

```
…
always
  begin
    #0    clock = 0;
    #50   clock = 1;
    #50;
  end
…
```

```
…
always
  begin
    #clock_period/2;
    clock = ~clock;
  end
…
```

parameter

clock_period

*Pictures revised from Prof. Ting-Chi Wang's 2013 slides for education purpose only

Caution of Assignments

```
// Error version
module FullAdder(s, co, a, b, ci);

input a, b, ci;
output s, co;

s = a ^ b ^ ci;

always @(a or b or ci) begin
  assign co = (a&b)|(b&ci)|(a&ci);
end

endmodule
```

Error continuous assignment!

Error procedural assignment!

```
// Correct version
module FullAdder(s, co, a, b, ci);

input a, b, ci;
output s, co;
reg co;

assign s = a ^ b ^ ci;

always @(*) begin
  co = (a&b)|(b&ci)|(a&ci);
end

endmodule
```

lvalue of procedural assignment must be reg

*Pictures revised from Prof. Shih-Chieh Chang's 2010 slides for education purpose only

Combinational Circuits (1/2)

- You have **three ways** to model a combinational circuit
  - You will be good if you remember them

1. **Gate-level modeling**
   - **not** N1(**out, in**);
   - **nand** N2(**out, in1, in2**);
   - **or** N3(**out, in1, in2, in3**);
   - etc.

2. **Continuous assignment**
   - Declare your **lvalues** as **wire** data type
   - **assign lvalue** = (your logic expression);

Combinational Circuits (2/2)

3. **Use always block**
   - Declare your **lvalues** as **reg** data type
   - Only use "**=**" in your assignment expressions
   - Avoid using "**<=**" in your expressions

Correct

```
reg lvalue;

always @ (*)
  begin
        lvalue = (your expression);
  end
```

Wrong

```
wire lvalue;

always @ (*)
  begin
        assign lvalue <= (your expression);
  end
```

1. lvalue should be reg
2. Can't use assign
3. Avoid "<="

Agenda

- Gate-Level Design and Basic Concepts
- Data Flow Modeling
- Behavioral Modeling
- Sequential Circuits
- Debounce and One-Pulse Circuits
- FPGA Implementation

Today's class will help you:

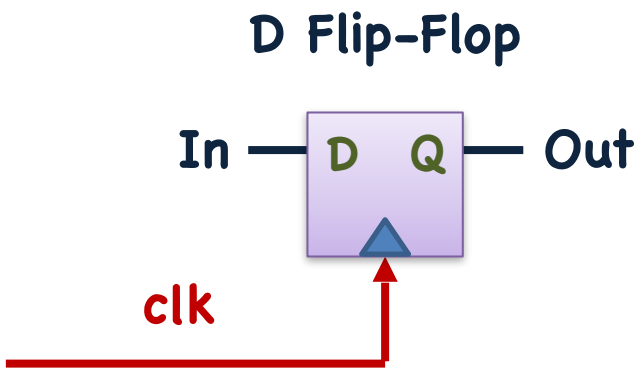
1. Review the contents of this course

What is Sequential Circuit?

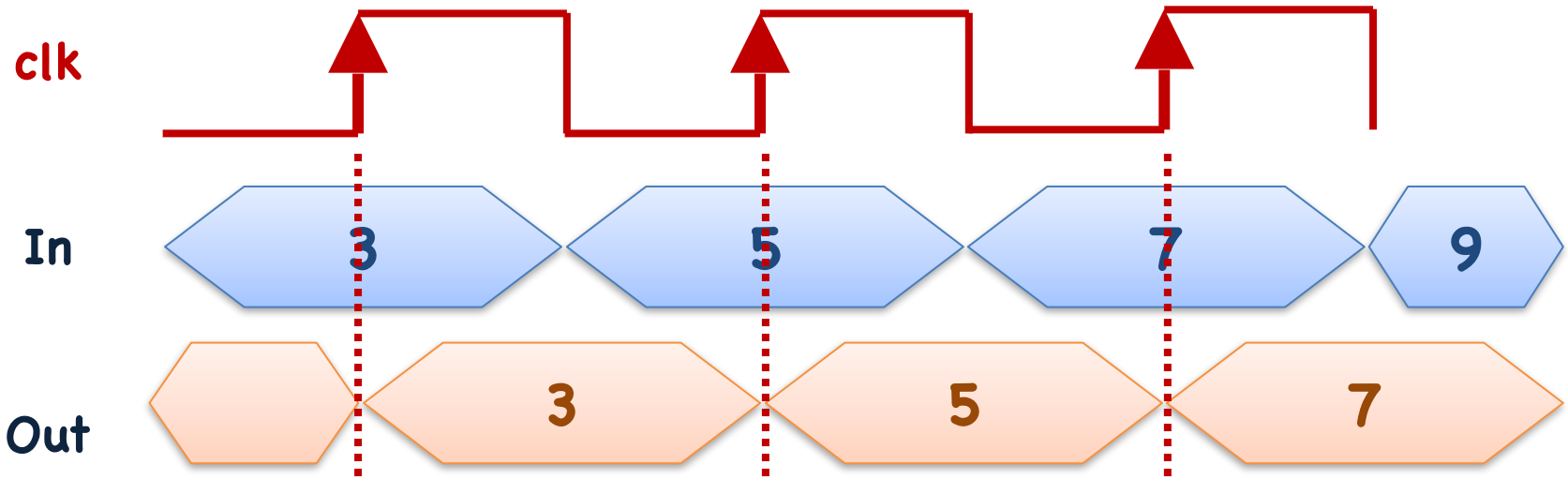
	Sequential	Combinational
Clock	Yes	No
Memory elements	Yes	No
Always block assignment	Non-blocking <=	Blocking =
Continuous assignment	Not available	assign lvalue = rvalue

- Memory element
  - Stores “1” or “0”
  - Latch and Flip-Flop
  - Usually triggered by clock signals

Basic Sequential Circuit

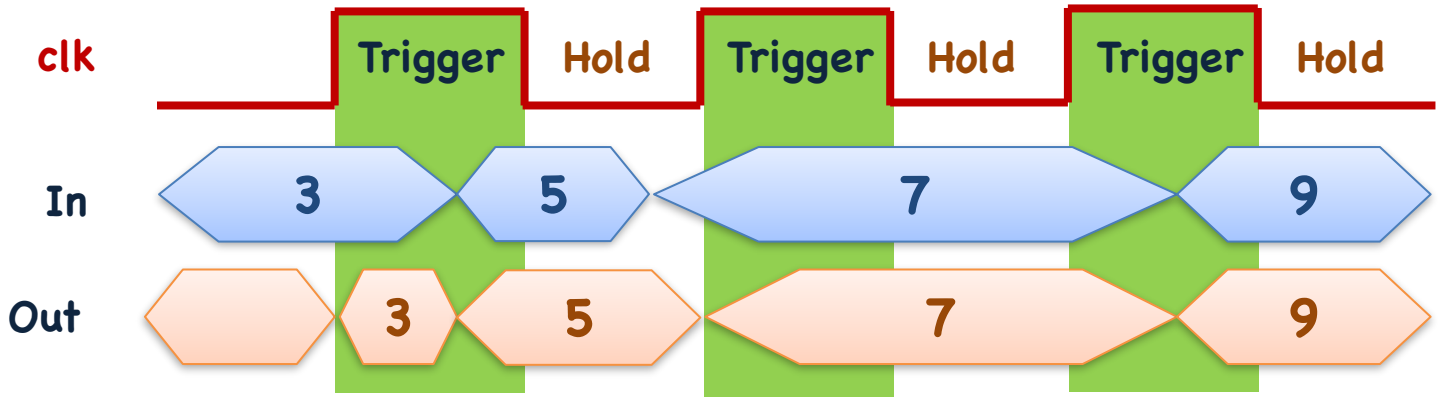



```
always @ (posedge clk)
  begin

  //Non-blocking assignment
        Out <= In;
  end
```

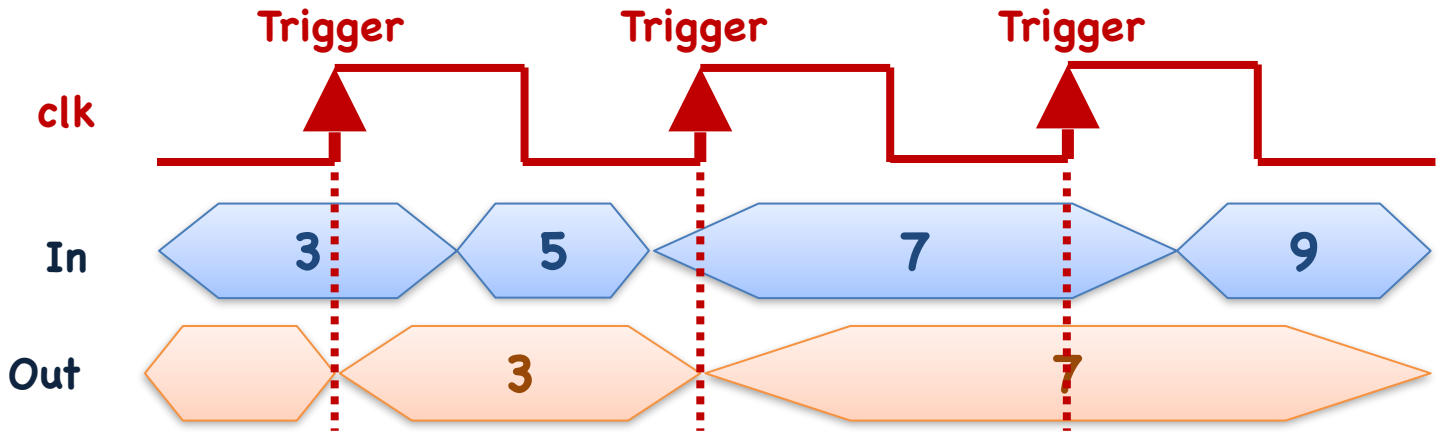


Latch and D Flip-Flop

- **Latch**: triggered by **level**



- **D Flip-Flop**: triggered by **clock edge**



D Flip Flop

- Changes its value at the clock edges

Positive-edge triggered D Flip-Flop Truth table

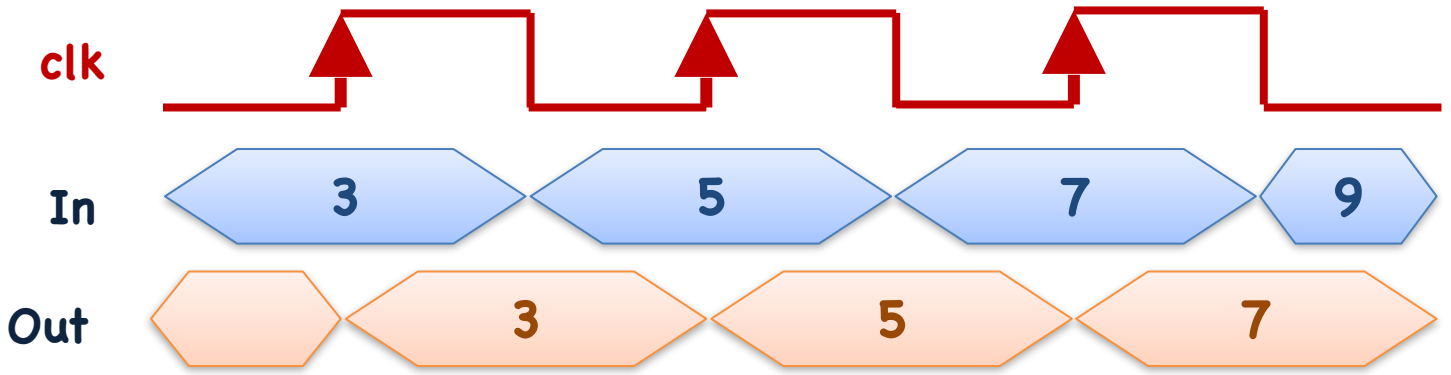
Clk	D	Q(t+1)
↑	0	0
↑	1	1
0	–	Q(t)
1	–	Q(t)

Negative-edge triggered D Flip-Flop Truth table

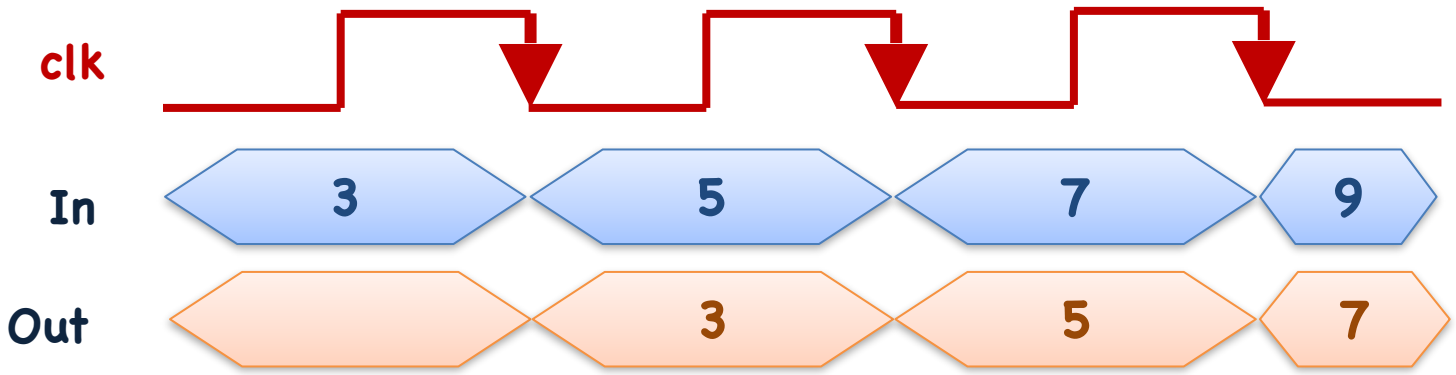
Clk	D	Q(t+1)
↓	0	0
↓	1	1
0	–	Q(t)
1	–	Q(t)

Edge-Trigger in Always Block

- **Positive-edge trigger**: use **posedge** in your always block

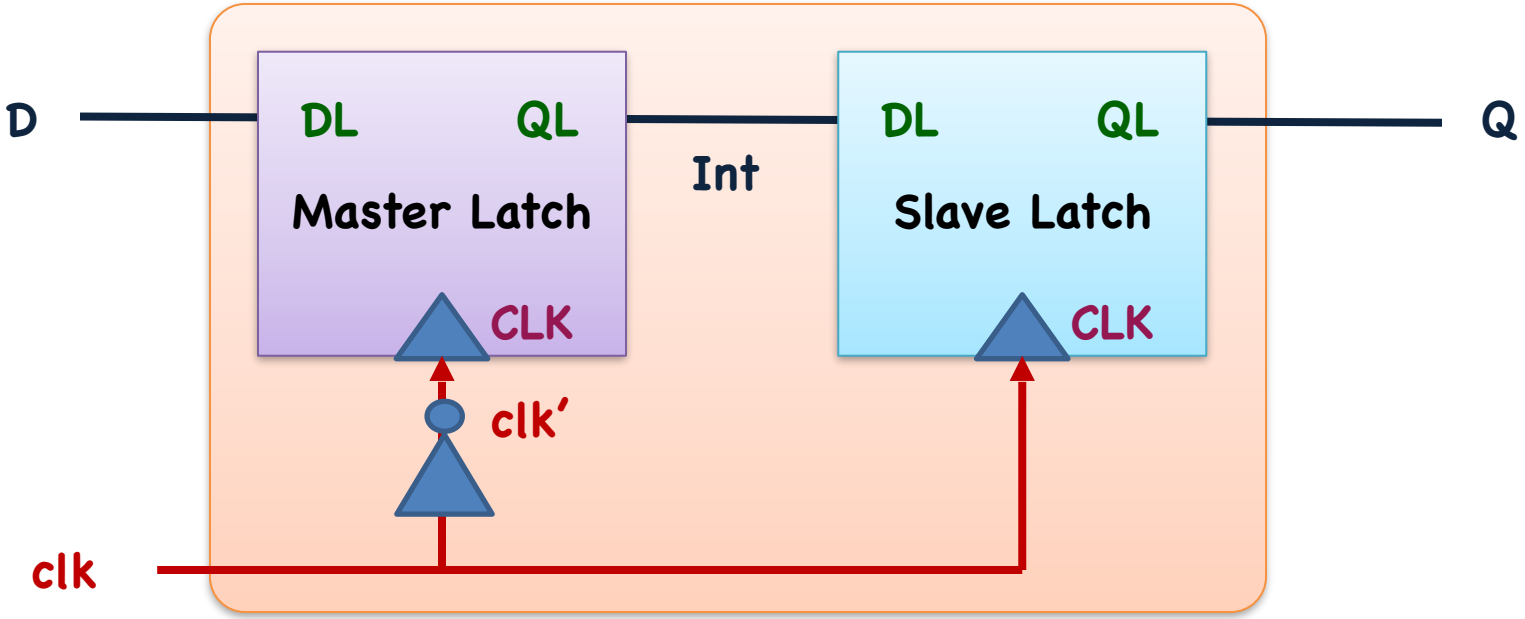


- **Negative-edge trigger**: use **negedge** in your always block



Latches in a D Flip-Flop

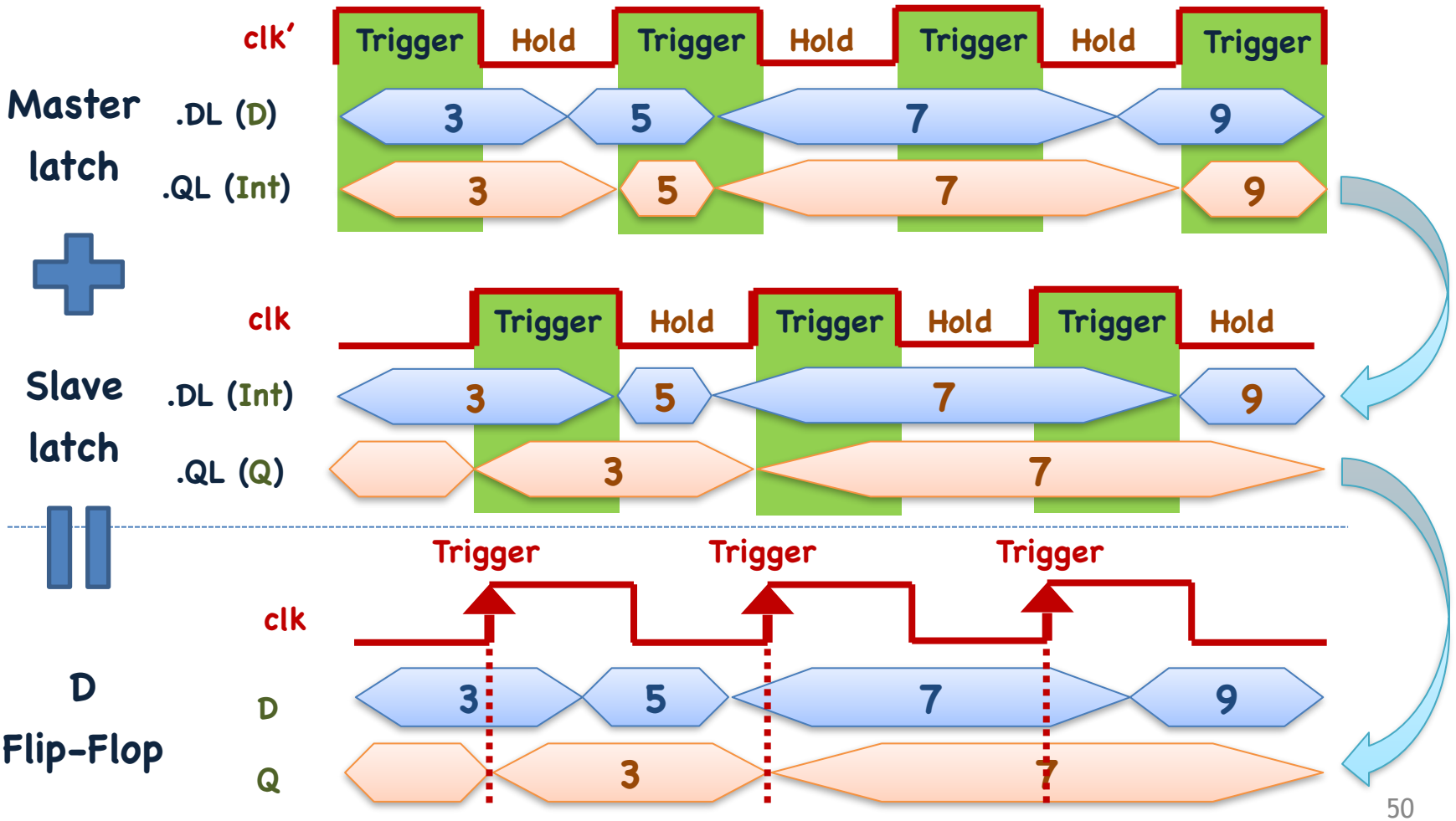
- A positive-edge triggered D Flip-Flop consists of two latches
  - The master latch is triggered by **clk'**
  - The slave latch is triggered by **clk**
- Two level triggered latches form an edge-triggered Flip-Flop



D Flip-Flop

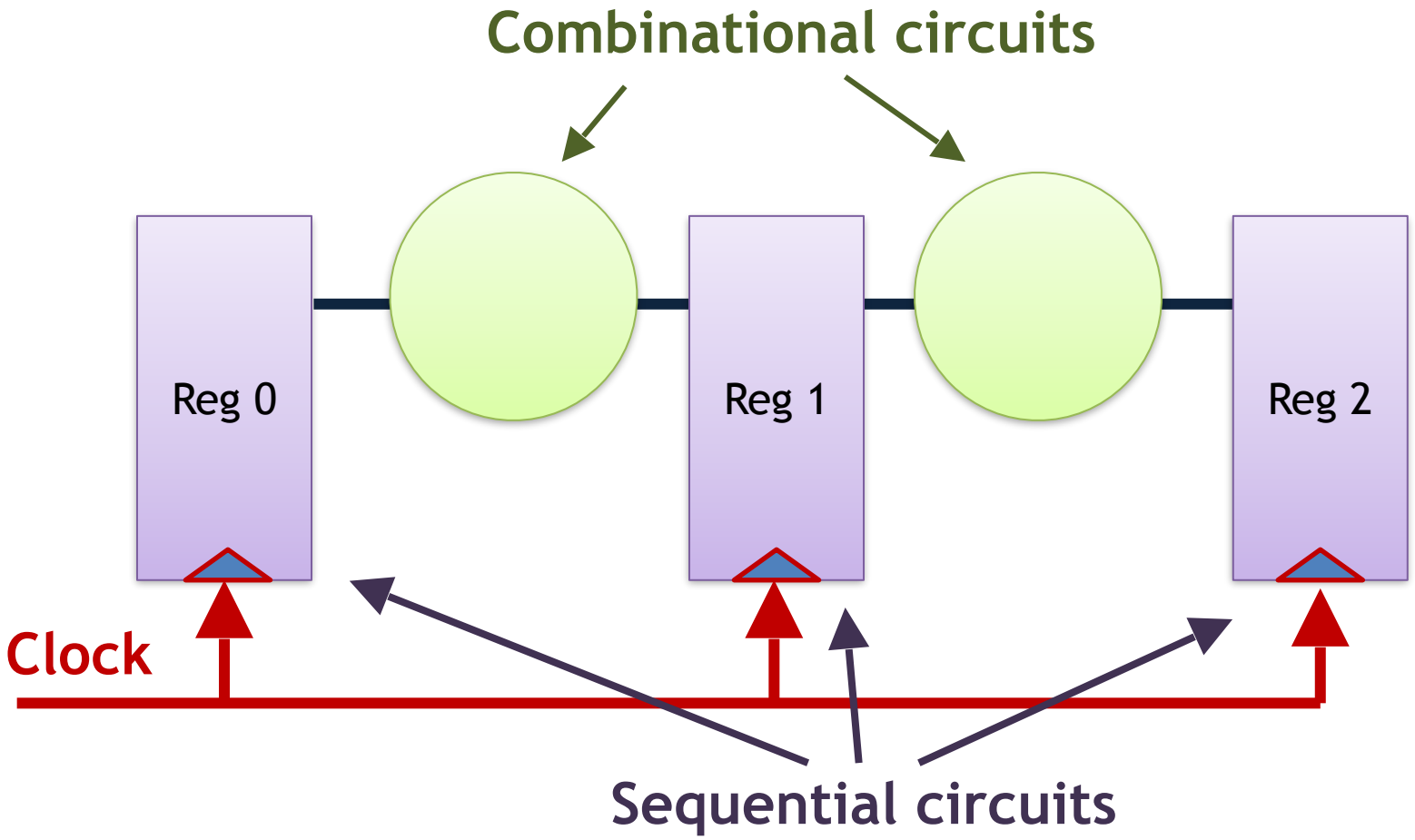
Flip-Flop Timing Diagram

- A timing diagram from the master latch to slave latch



Register Transfer Level

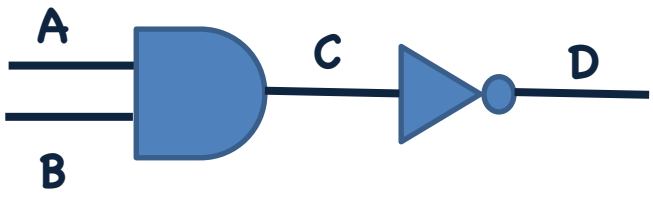
- Describes the behavior of combinational circuits between registers



Blocking and Non-blocking

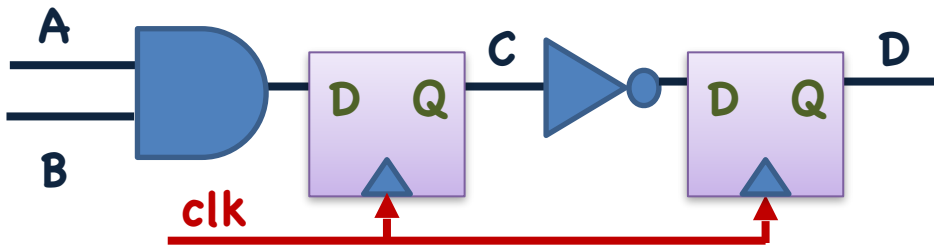
Execute in Order

```
always @ (A or B or C)
  begin

   //Blocking assignment
        C = A & B;
        D = !C;
  end
```



Execute in Parallel

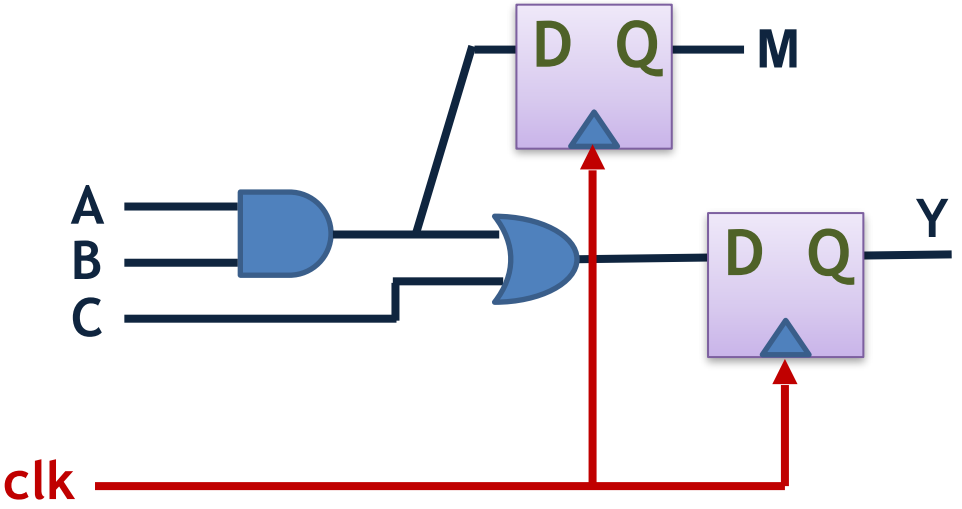
```
always @ (posedge clk)
  begin

   //Non-blocking assignment
        C <= A & B;
        D <= !C;
  end
```



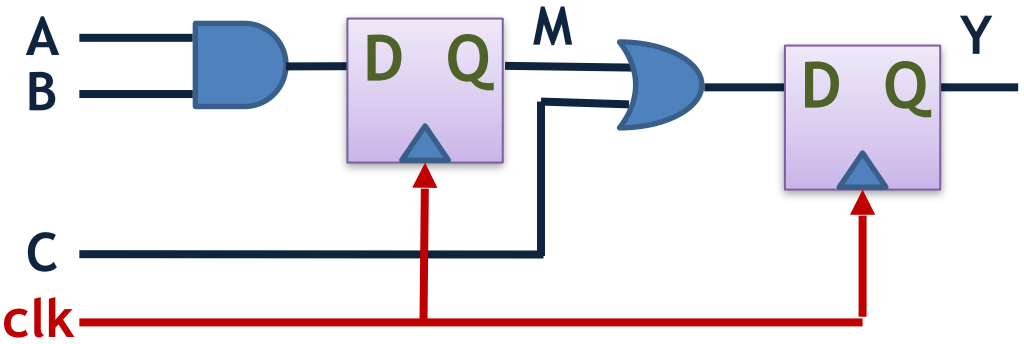
- **NEVER** use blocking and non-blocking assignment in the **SAME** always block

Bad and Good Examples

```
// blocking assignment
always @(posedge clk) begin
  M  = A & B;
  Y  = M | C;
end
```



```
// non-blocking assignment
always @(posedge clk) begin
  M  <= A & B;
  Y  <= M | C;
end
```



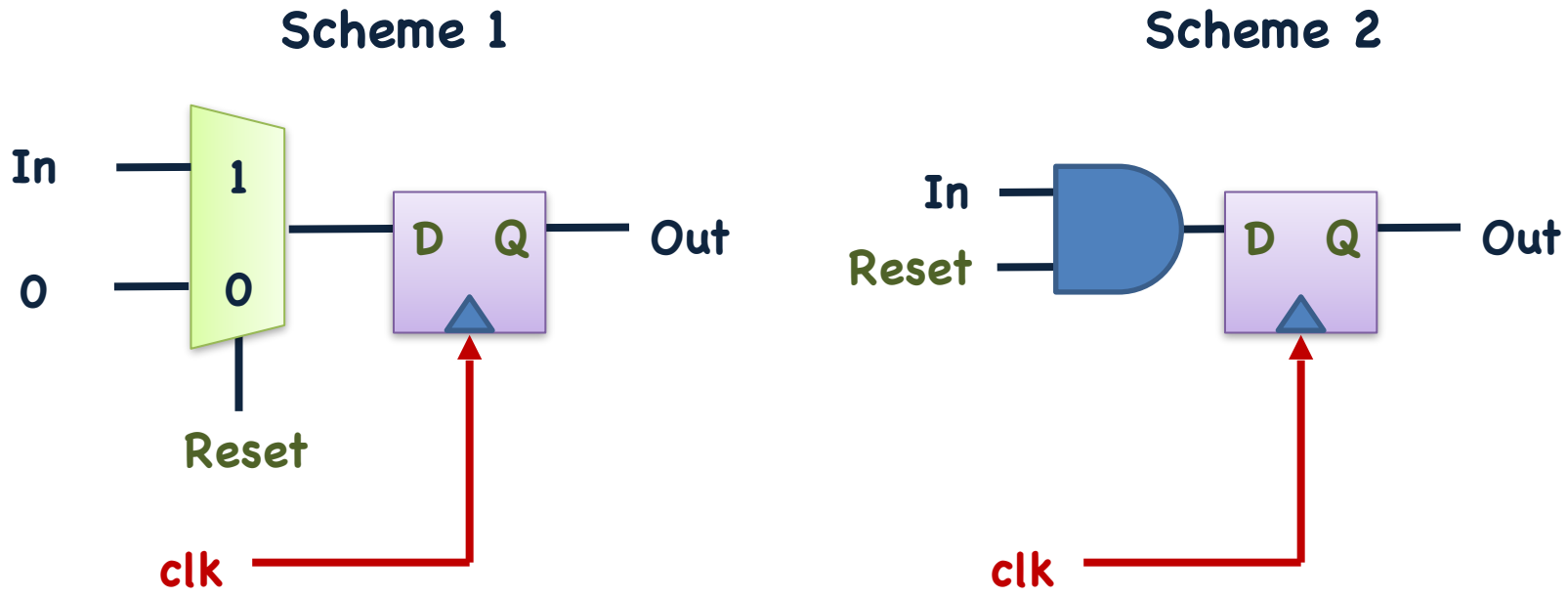
*Pictures revised from Prof. Shih-Chieh Chang's 2010 slides for education purpose only

Initialize Flip-Flops

- For combinational circuits, we initialize the inputs of modules by initial blocks in the testbench
- For sequential circuits, a **reset signal** is necessary
  - Initialize the reset signal (i.e. your module) first in your testbench
- Two reset methods
  - Synchronous reset
  - Asynchronous reset

Synchronous Reset

- Reset the value stored in a D Flip-Flop to 0
- Triggered by clock edges
- Assume that **Q = 1'b0** when Reset is set to **1'b0**



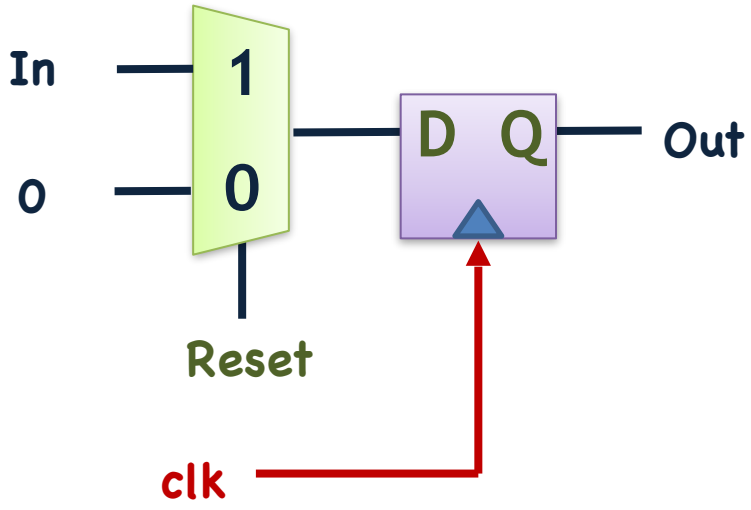
Synchronous Reset in Always Block

- **Prioritize** your reset signal over the rest of the inputs
- In your testbench, **initialize your reset signal first**
- You **shall not** use any initial block in your design modules

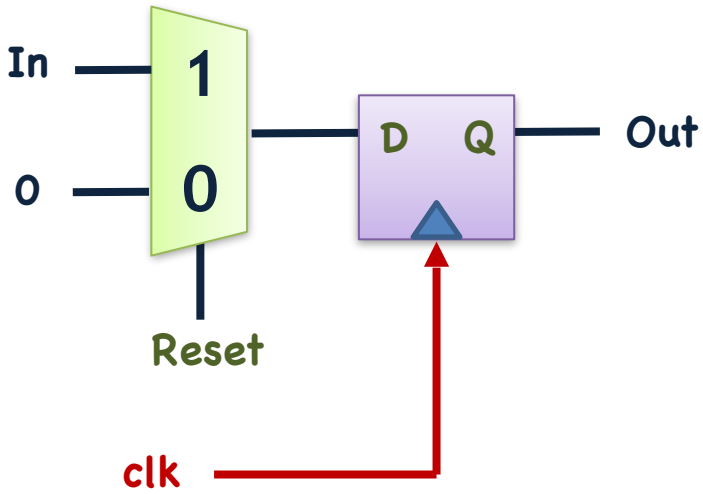
```
module My_Test_Flip_Flop (Out, In, clk, Reset);
  input    In, clk, Reset;
  output   Out;
  reg      Out;

always @ (posedge clk)
  begin
    if (Reset == 1'b0)
    else
  end
endmodule
```

Declaration of Synchronous Behavior

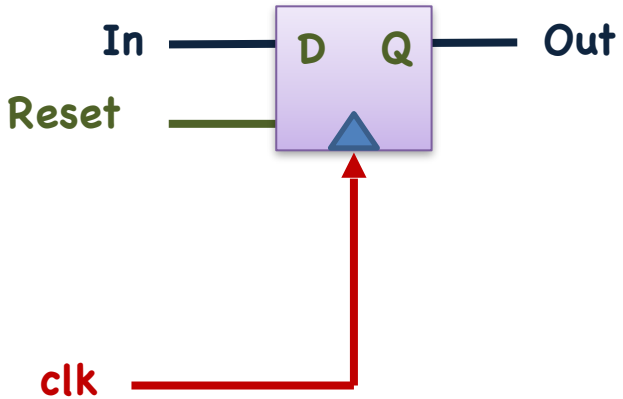
```
Out <= 1'b0;
Out <= In;
```



Synchronous vs. Asynchronous Reset



Synchronous Reset

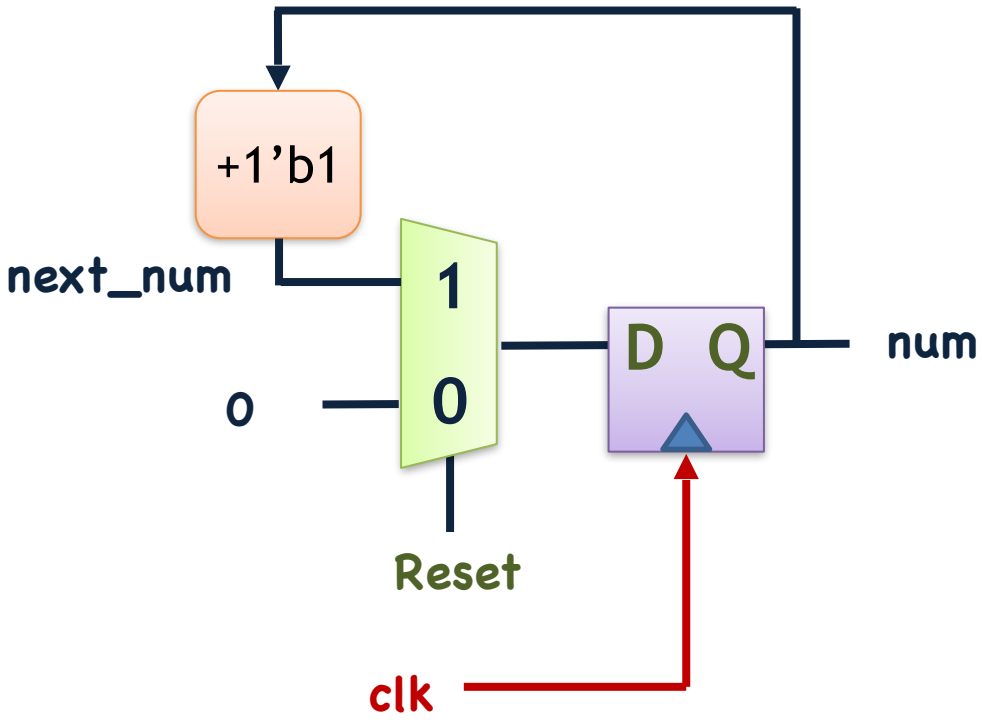


Asynchronous Reset

	Synchronous Reset	Asynchronous Reset
Clock	Reset at clock edges	Regardless of clock
Always block	always @ (posedge clk)	always @ (posedge clk or **nededge Reset**)
Logic	Additional Mux or AND gate	Additional reset input port in DFF

Example: 3-Bit Counter

```
module Counter_3Bit(clk, Reset, num);
input       clk, Reset;
output      [2:0] num;
reg         [2:0] num;
wire        [2:0] next_num;

always @(posedge clk) begin
    if (!Reset)
        num <= 1'b0;
    else
        num <= next_num;
end

assign      next_num = num + 1'b1;

endmodule
```



Agenda

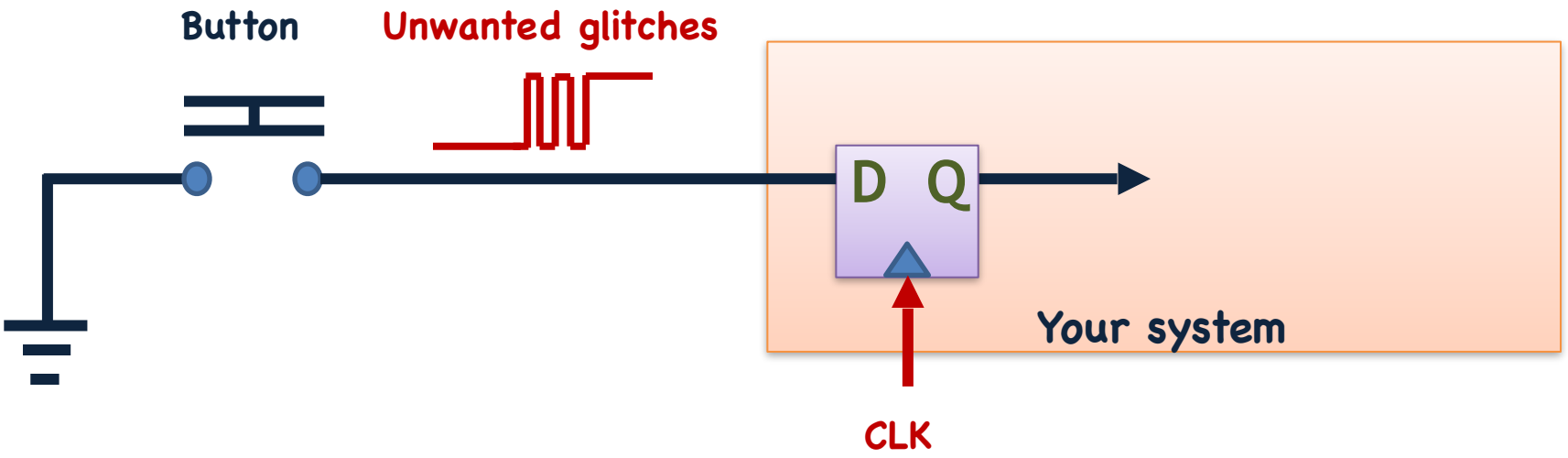
- Gate-Level Design and Basic Concepts
- Data Flow Modeling
- Behavioral Modeling
- Sequential Circuits
- Debounce and One-Pulse Circuits
- FPGA Implementation

Today's class will help you:

1. Review the contents of this course

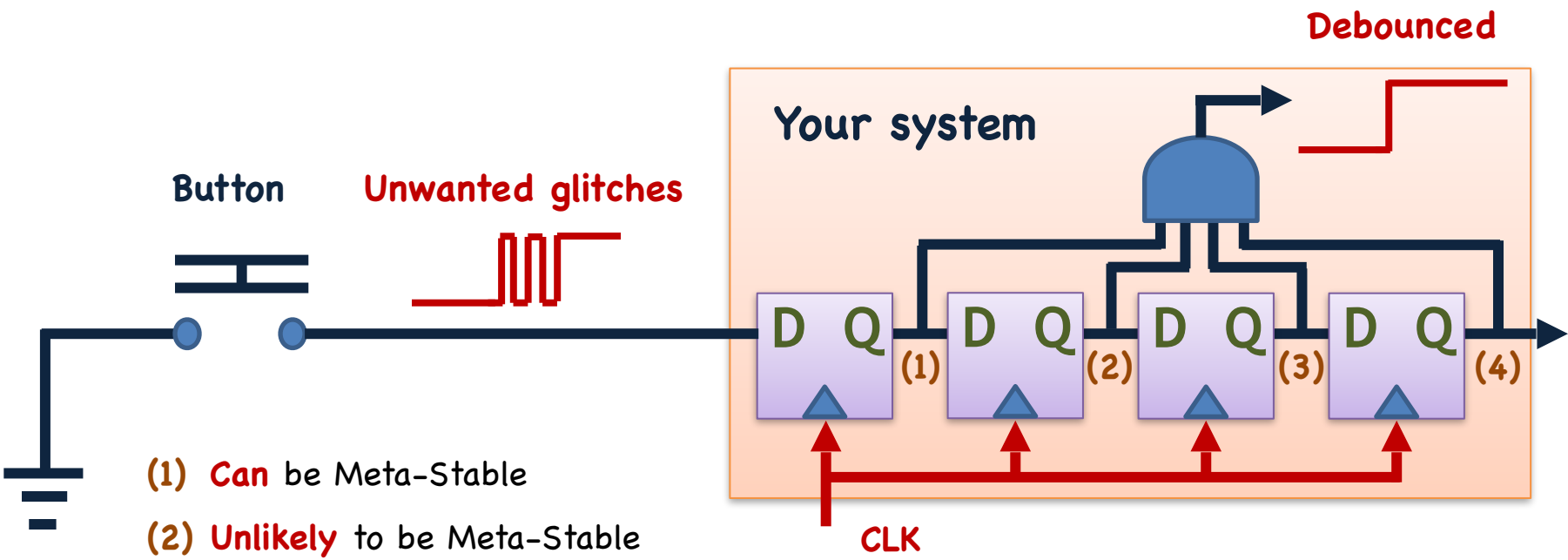
Push Buttons

- Several push buttons are available as inputs on FPGA
- Map push buttons to your inputs using .xdc file
- Need to handle **meta-stability** (**glitches**) problem



Handling Meta-Stability

- Debounce circuit
- Shift registers to allow time for signals to stabilize
- Detect if DFF chains are all ones or zeros

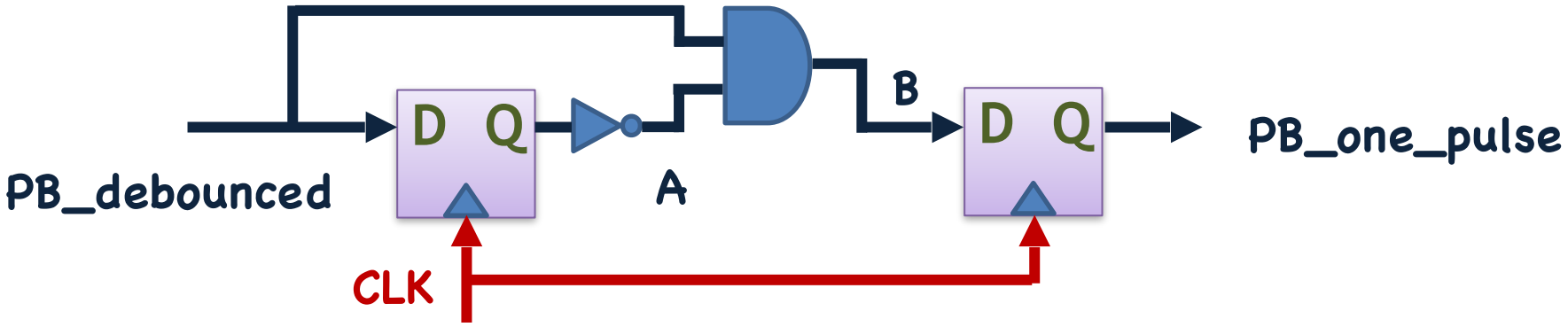


Verilog Code for Debounce Circuit

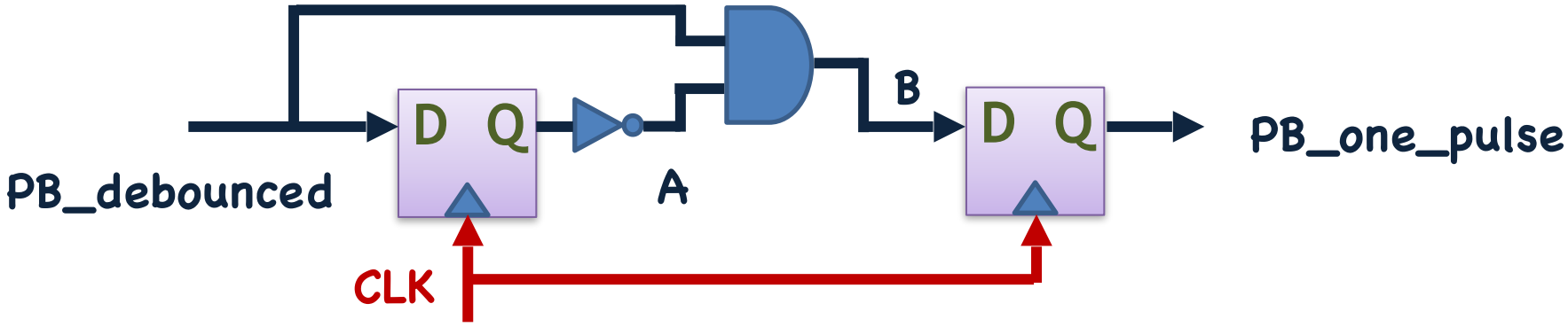
```
module debounce (pb_debounced, pb, clk);

  output    pb_debounced;      // signal of a pushbutton after being debounced
  input     pb;                // signal from a pushbutton
  input     clk;

  reg [3:0] DFF;               // use shift_reg to filter pushbutton bounce

  always @(posedge clk)
    begin
      DFF[3:1] <= DFF[2:0];
      DFF[0]   <= pb;
    end

  assign pb_debounced = &DFF;

endmodule
```

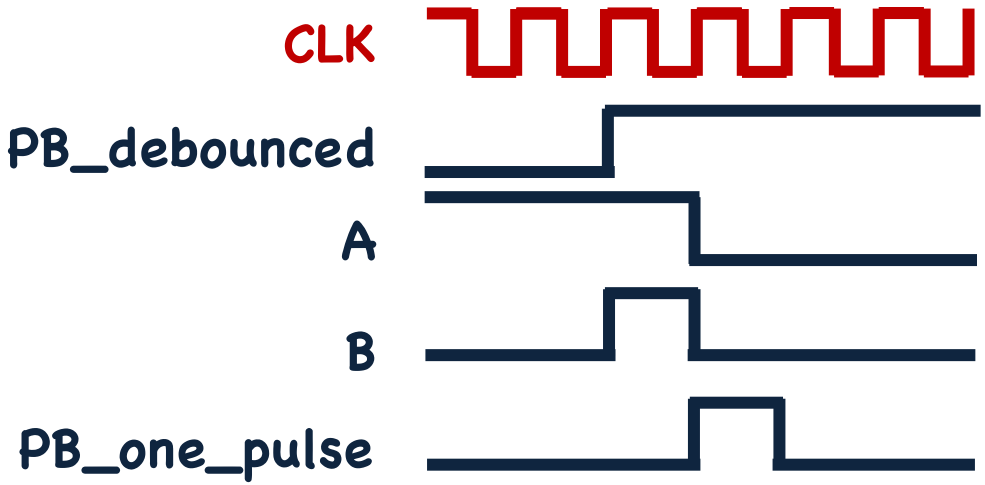
One Pulse Circuit

- The one pulse circuit generates a one-clock-cycle pulse when the push button is pressed
  - A push button is usually pressed for many clock cycles
- The circuit is used when we want to trigger only once



One Pulse Circuit (Cont'd)





Verilog Code

```
// One-pulse circuit
module onepulse (PB_debounced,  CLK, PB_one_pulse);
  input           PB_debounced;
  input           CLK;
  output          PB_one_pulse;
  reg             PB_one_pulse;
  reg             PB_debounced_delay;

  always @(posedge CLK) begin
      PB_one_pulse <= PB_debounced & (! PB_debounced_delay);

      PB_debounced_delay <= PB_debounced;
  end
endmodule
```

Agenda

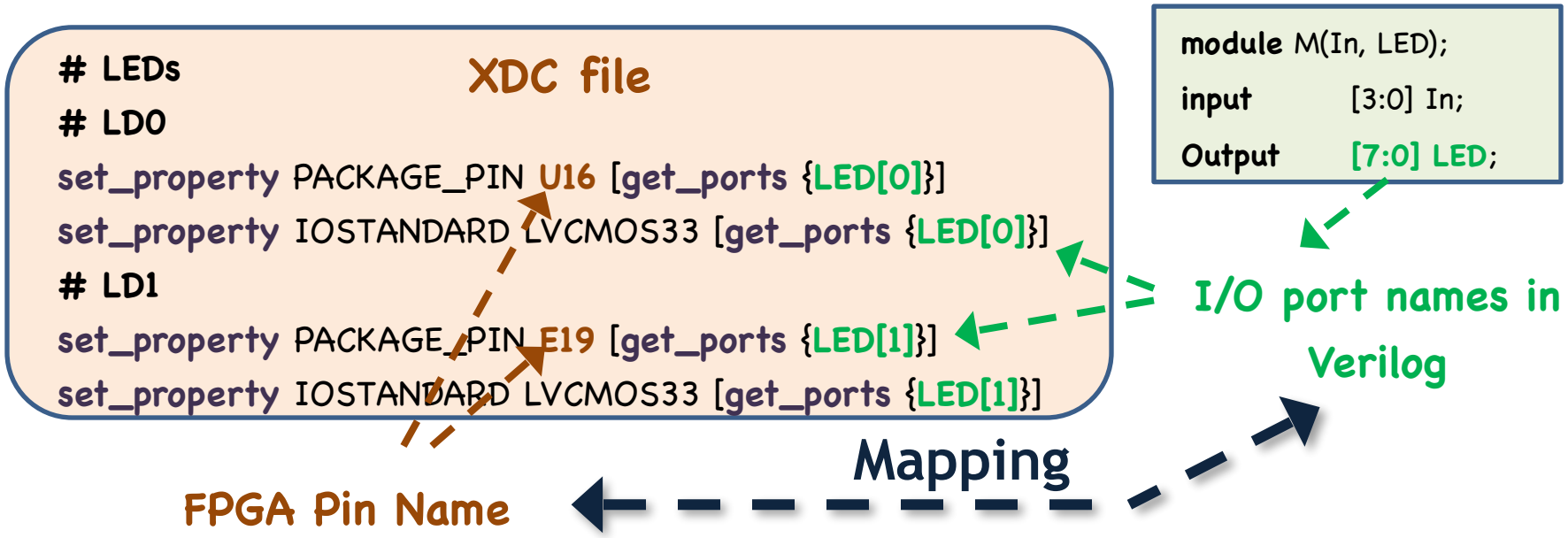
- Gate-Level Design and Basic Concepts
- Data Flow Modeling
- Behavioral Modeling
- Sequential Circuits
- Debounce and One-Pulse Circuits
- FPGA Implementation

Today's class will help you:

1. Review the contents of this course

Port Mapping

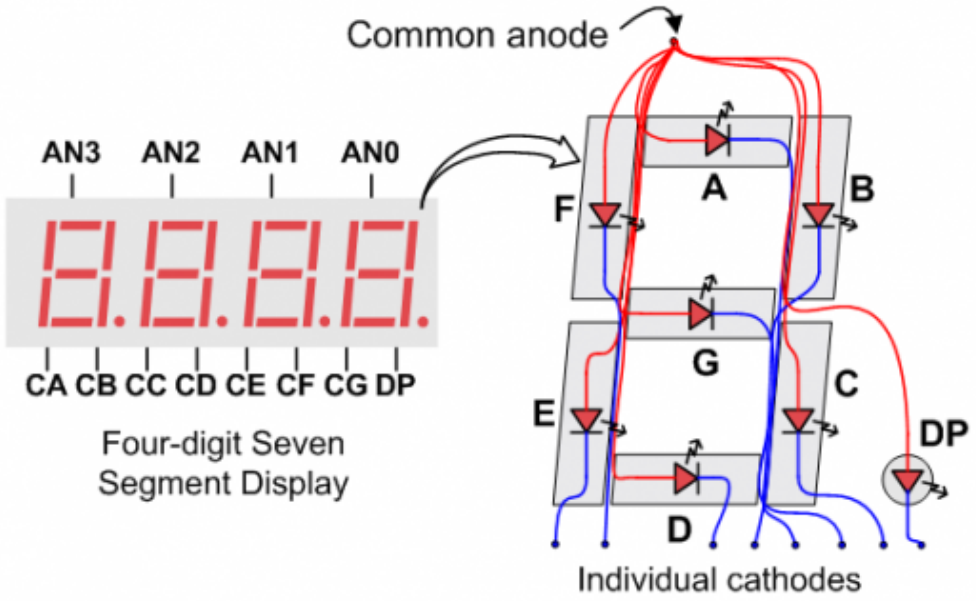
- Maps your inputs and outputs to FPGA pins
  - One-to-one mapping
- Defined in an **XDC** (Xilinx Design Constraints) file



7-Segment Display Control (1/2)

- Four enable signals for the four display units
  - **AN[3:0]** correspond to **the four digits**
  - **AN** is used to **ENABLE** one of **the four digits**
  - Set one bit of AN[3:0] to **LOW** to illuminate one digit

AN Bit	FPGA Pin
3	W4
2	V4
1	U4
0	U2

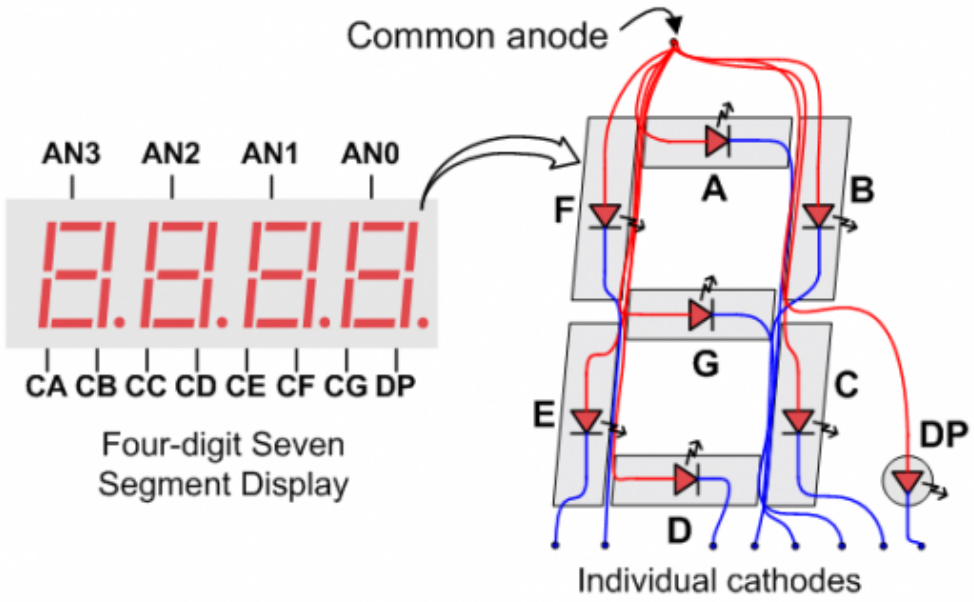


Digit	AN[3:0]
3	0111
2	1011
1	1101
0	1110

*Pictures cited from www.xilinx.com for education purpose only

7-Segment Display Control (2/2)

- Eight signals for the segments and dot
  - **seg[0:6]** in the XDC file correspond to **segments A~G** shown below
  - **dp** in the XDC file corresponding to **DP (dot)**
  - To illuminate a segment, set its value to **LOW**

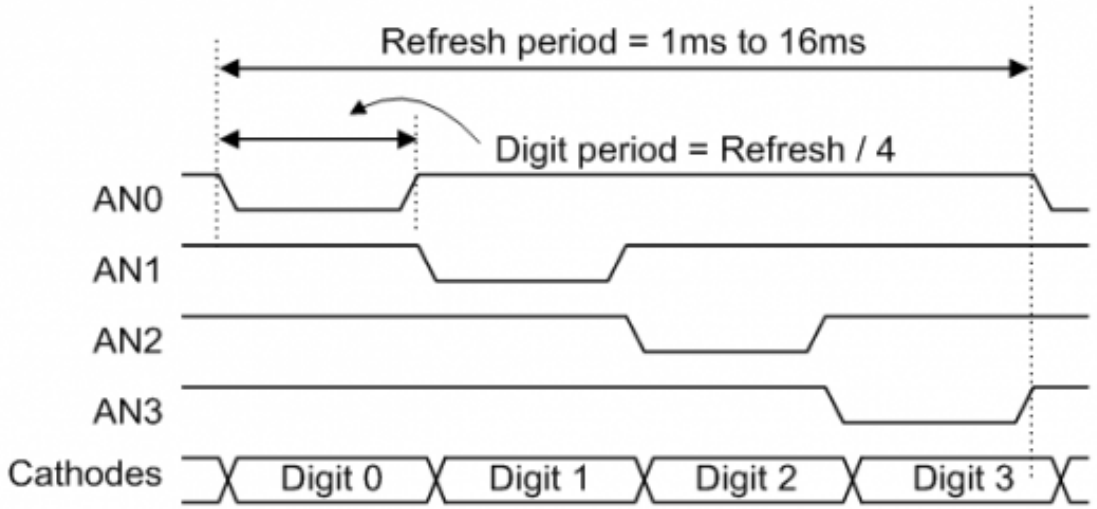


Symbol	FPGA Pin
CA	W7
CB	W6
CC	U8
CD	V8
CE	U5
CF	V5
CG	U7
DP	V7

*Pictures cited from www.xilinx.com for education purpose only

Display Four Digits Concurrently

- Time multiplexing
- Use a **mux** and **clock divider**
  - **Mux** to select the digit to display (i.e. set which **AN** bit to low)
  - **Clock divider** to display at the right refresh frequency (**1ms to 16ms**)
- The clock provided by Basys 3 is **100MHz**
  - That is, **10ns** per clock cycle
  - Generate a clock with a frequency of **$1/2^{17}$ clk**



*Pictures cited from www.xilinx.com for education purpose only

*Taken at Yosemite Valley Village at Yosemite, California, USA.
This picture is taken by Chun-Yi Lee himself, who is also a fan of photography