Welcome to The Hardware Lab!

Fall 2018
Behavioral Modeling

Prof. Chun-Yi Lee

Department of Computer Science
National Tsing Hua University

Agenda

- Announcement
- Behavioral modeling
    - Behavioral constructs
    - Procedural statements

Today's class will help you:

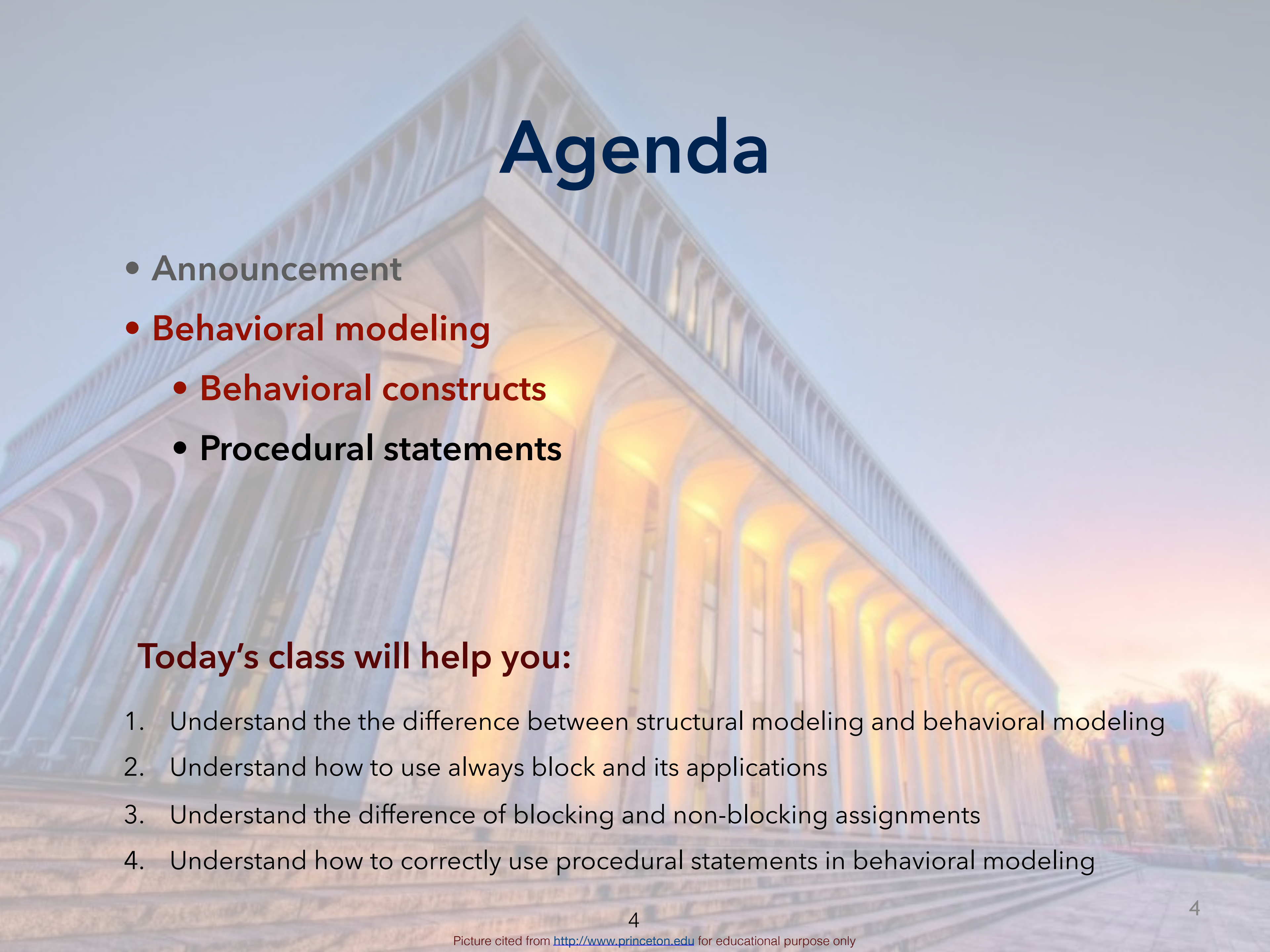
1. Understand the the difference between structural modeling and behavioral modeling
2. Understand how to use always block and its applications
3. Understand the difference of blocking and non-blocking assignments
4. Understand how to correctly use procedural statements in behavioral modeling

2

Announcement

- Lab
  - Lab 2 Verilog submission due on **10/4/2016** (**Thu**)
  - Lab 2 FPGA demonstration due on **10/4/2016** (**Thu**)

- Assistance:
  - During the lab hours
  - TA hour: after the lectures (19:00~21:00) on Tuesdays
  - Please make a reservation in advance
    https://goo.gl/forms/MfI2Y1erpw6b71Wf2



Agenda

- Announcement
- **Behavioral modeling**
  - **Behavioral constructs**
  - Procedural statements

Today's class will help you:

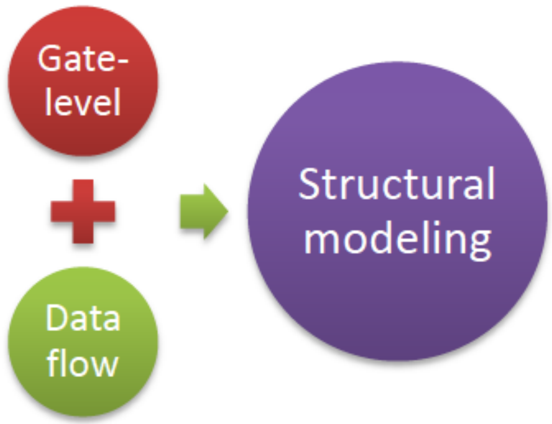
1. Understand the the difference between structural modeling and behavioral modeling
2. Understand how to use always block and its applications
3. Understand the difference of blocking and non-blocking assignments
4. Understand how to correctly use procedural statements in behavioral modeling

Picture cited from http://www.princeton.edu for educational purpose only

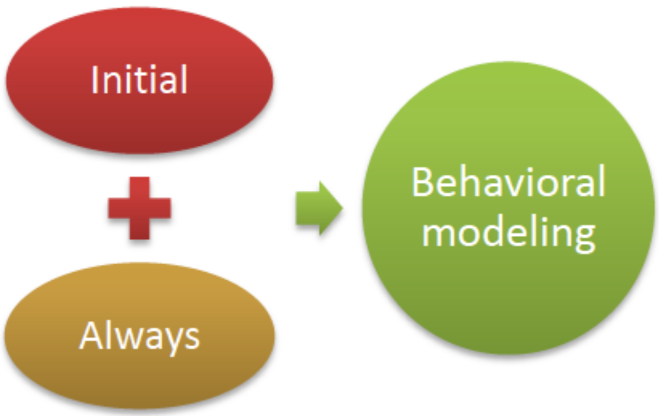
4

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

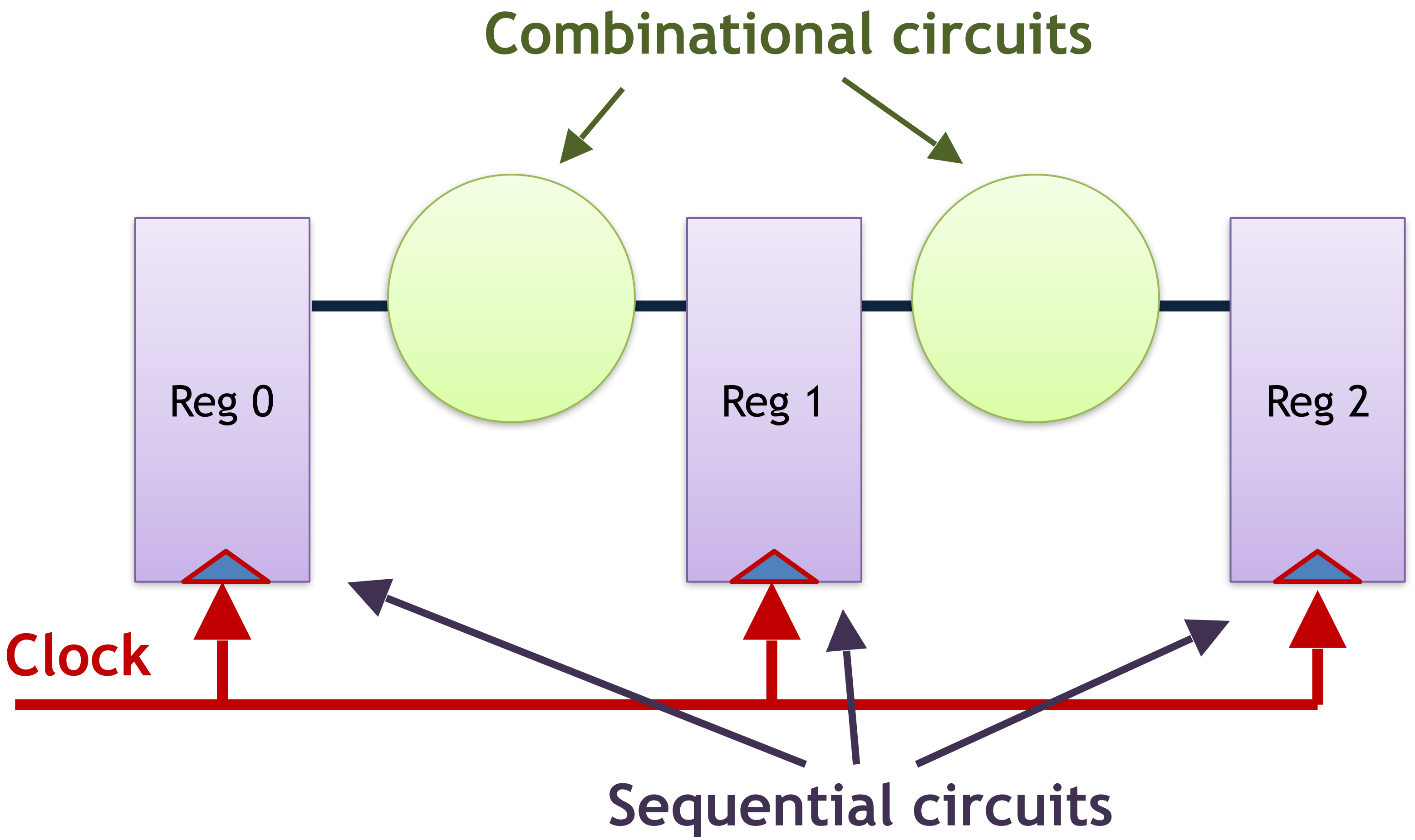
Initial Block

- In your **testbench only**
- **NOT** synthesizable
- Run once in the beginning (**time 0**)

```
...
initial                          // An "initial" behavior
    begin
        A = 1'b0;                // Procedural assignments
        B = 1'b1;                // execute sequentially
        #20
        C = 2'b11;
    end
...
```

Register Transfer Level

- Describes the behavior of combinational circuits between registers



Always Block

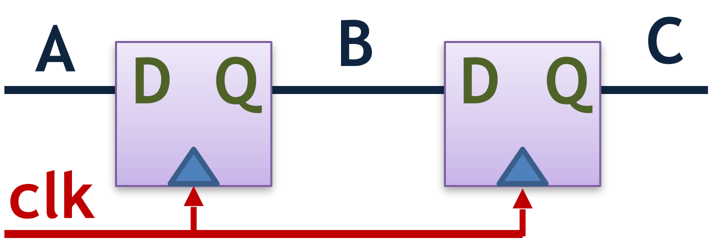
- Two types of always block

Combinational circuit

```
reg A, B, C;

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

Sequential circuit

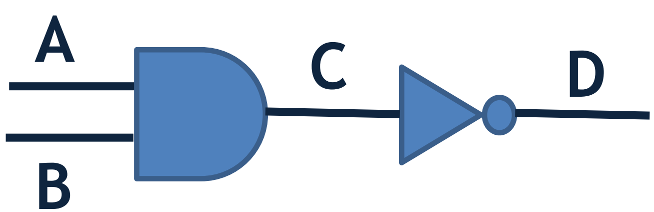
```
reg A, B, C;

always @ (posedge clk)
  begin
   //Non-blocking assignment
         B <= A;
         C <= B;
   end
```



Blocking and Non-blocking

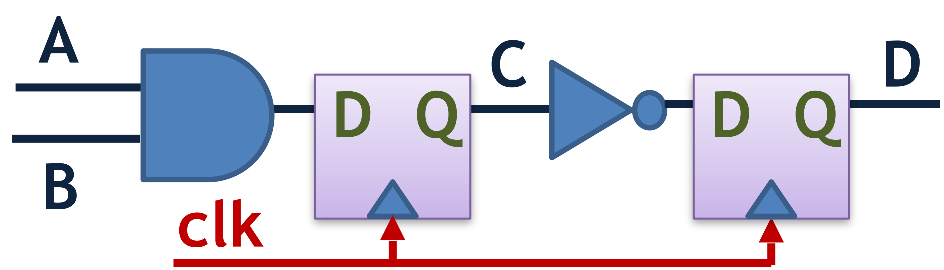
Execute in Order

```
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

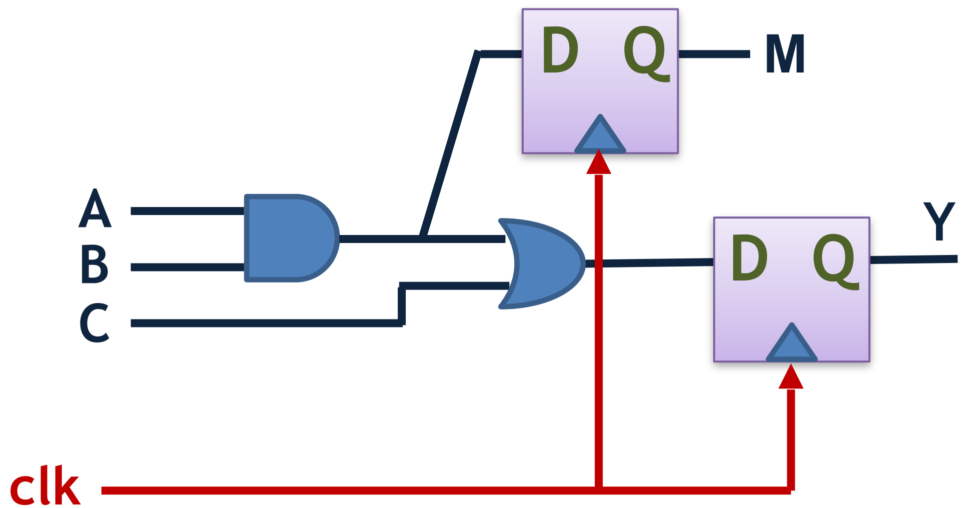
```
always @ (A or B or C)
  begin
   //Blocking Assignment
         C = A & B;
         D = !C;
   end
```

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

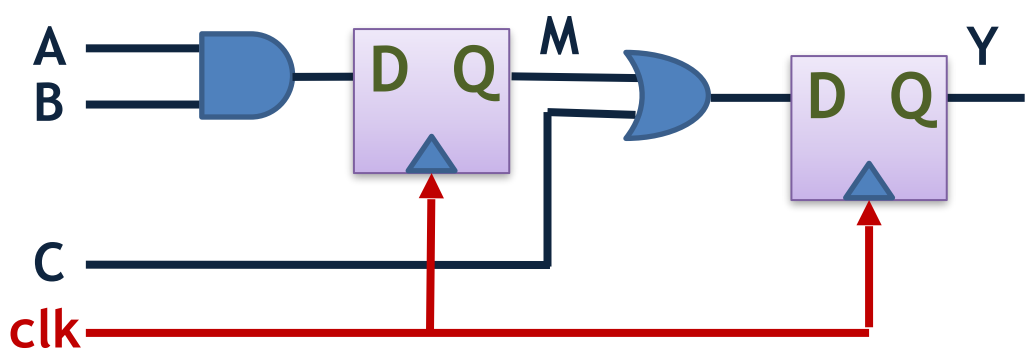
```
always @ (posedge clk)
  begin

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

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

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 @(*)**

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

Agenda

- Announcement
- **Behavioral modeling**
    - Behavioral constructs
    - **Procedural statements**

Today's class will help you:

1. Understand the the difference between structural modeling and behavioral modeling
2. Understand how to use always block and its applications
3. Understand the difference of blocking and non-blocking assignments
4. Understand how to correctly use procedural statements in behavioral modeling

Picture cited from http://www.princeton.edu for educational purpose only

Procedural Statements

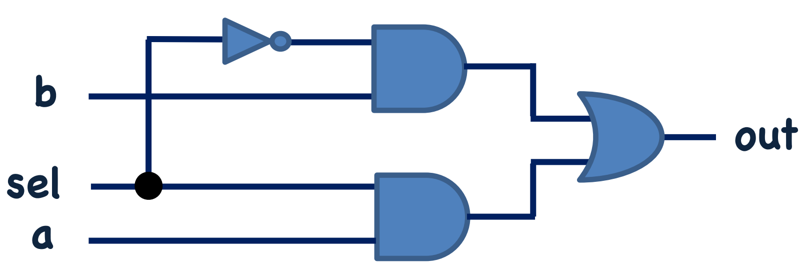
- Control operators similar to C++
- Not all of the operators can be used in your design
- Any operator used in the design must be **synthesizable**
    - However, you can use non-synthesizable operators in testbenches

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
```

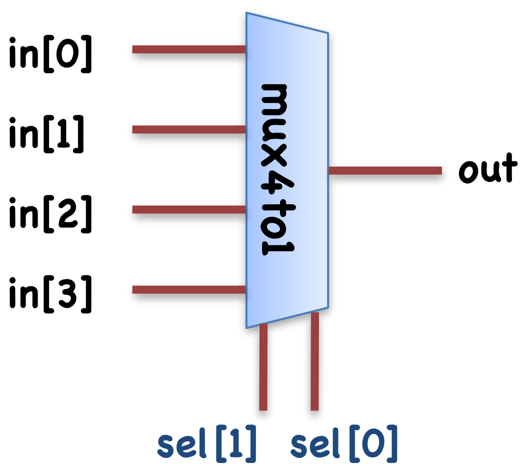
```
module MUX3(out, a, b, sel);

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

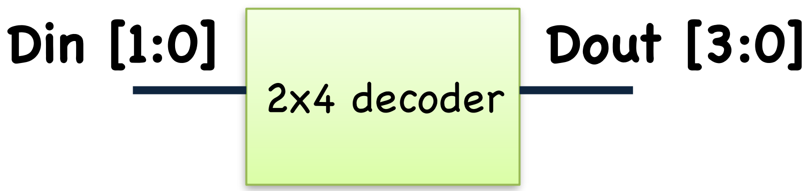
```
case (condition)
  alternative1: <expression> ;
  alternative2: <expression> ;
  …
  default: <expression> ;
endcase
```



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

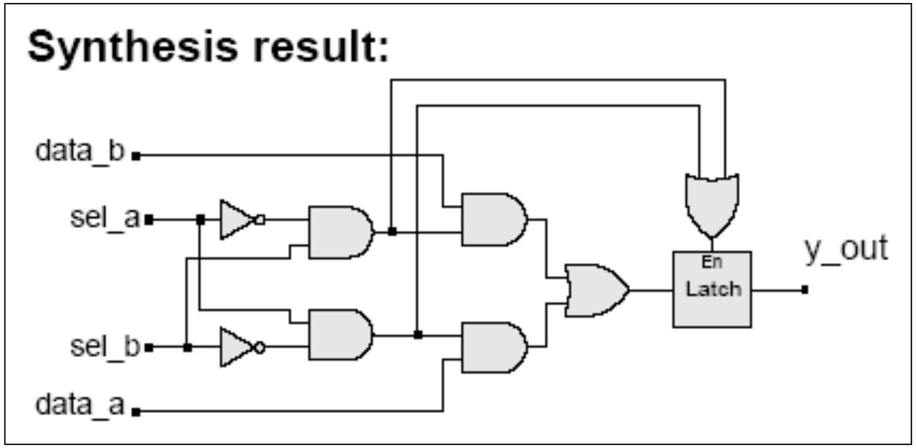
Synthesized to a **DECODER**

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

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[i+1] = 16'd0;
        x[i+2] = 16'd1;
    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

Thank you for your attention!

*Reno Air Balloon Festival taken at Reno, Nevada, USA.
This picture is taken by Chun-Yi Lee himself, who is also a fan of photography