NATIONAL UNIVERSITY OF HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY
FACULTY OF COMPUTER ENGINEERING

CIRCUIT DESIGN WITH HDL

CHAPTER 4: BEHAVIORAL MODELING

Lecturer: Ho Ngoc Diem

Course outline

- Chapter 1: Introduction
- Chapter 2: Verilog Language Concepts
- Chapter 3: Structural modeling
- Chapter 4: Behavioral modeling
- Chapter 5: Finite State machines
- Chapter 6: Tasks and Functions
- Chapter 7: Functional Simulation/Verification (Testbench)
- Chapter 8: Synthesis of Combinational and Sequential Logic
- Chapter 9: Post-synthesis design tasks
- Chapter 10: VHDL introduction

Content

Chapter 4: Behavioral modeling – Part 2

- Boolean-equation-based behavioral model
- Continuous assignment
- Expressions in Verilog
- Cyclic behavioral models with always block
- initial & always construct
- Procedural assignment
- Blocking and non-blocking assignment
- Sequential and Parallel blocks
- Timing control mechanism in Verilog
- Conditional, Case, Loop Statements
- Related modeling techniques

Initial & Always

- Two basic components of behavioral modeling: initial, always construct.
- Each initial and always construct starts a separate activity flow in Verilog.
- initial and always can not be nested.
- In the code: **initial** and **always** start together at simulation time 0, initial executes once, always executes repetitively.
- They contain behavioral statements like: Procedural assignment, if…else, case, loop statements.

```
module behave;
reg [1:0] a, b;

initial begin
    a = 'b1;
    b = 'b0;
end
always begin
    #50 a = ~a;
end
always begin
    #100 b = ~b;
end

endmodule
```

Initial

- Variable and Net can be initialized when they are declared

```
module adder (sum, co, a, b, ci);

output reg [7:0] sum = 0; //Initialize 8 bit output sum
output reg co = 0; //Initialize 1 bit output co
input [7:0] a, b;
input ci;
…..
endmodule
```

```
module adder (output reg [7:0] sum = 0,
              output reg co = 0,
              input [7:0] a, b,
              input ci );
//it is illegal to redeclare any ports of the module
//in the body of the module
…
endmodule
```

Always

- **always** may be followed by timing control expression (**@**, **#**, **wait** -> see later)
- Statement following **always** is executed repeatedly until simulation stops. **$stop** or **$finish** to halt the simulation

```
module clock_gen (output reg clock);
//Initialize clock at time zero
initial
    clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always
    #10 clock = ~clock;
initial
    #1000 $finish;
endmodule
```

Procedural Assignment

- Assign value to variable data types

Left-hand side	**Right-hand side**
- reg, integer, time, real, realtime - Bit-select, part-select of **reg, integer**, **time** - Memory word - Concatenation of any of the above	Net, variable, function call (Any expression that evaluates a value)

- Occur in procedures: **initial**, **always, task,** and **function**
- Being thought of as "triggered" assignment
- Variable hold value until the next assignment

Examples:

```
reg[3:0] a = 4'h4;

//This is equivalent to writing
reg[3:0] a;
initial a = 4'h4;
```

Procedural assignment

- *Difference between Continuous and Procedural assignment:*

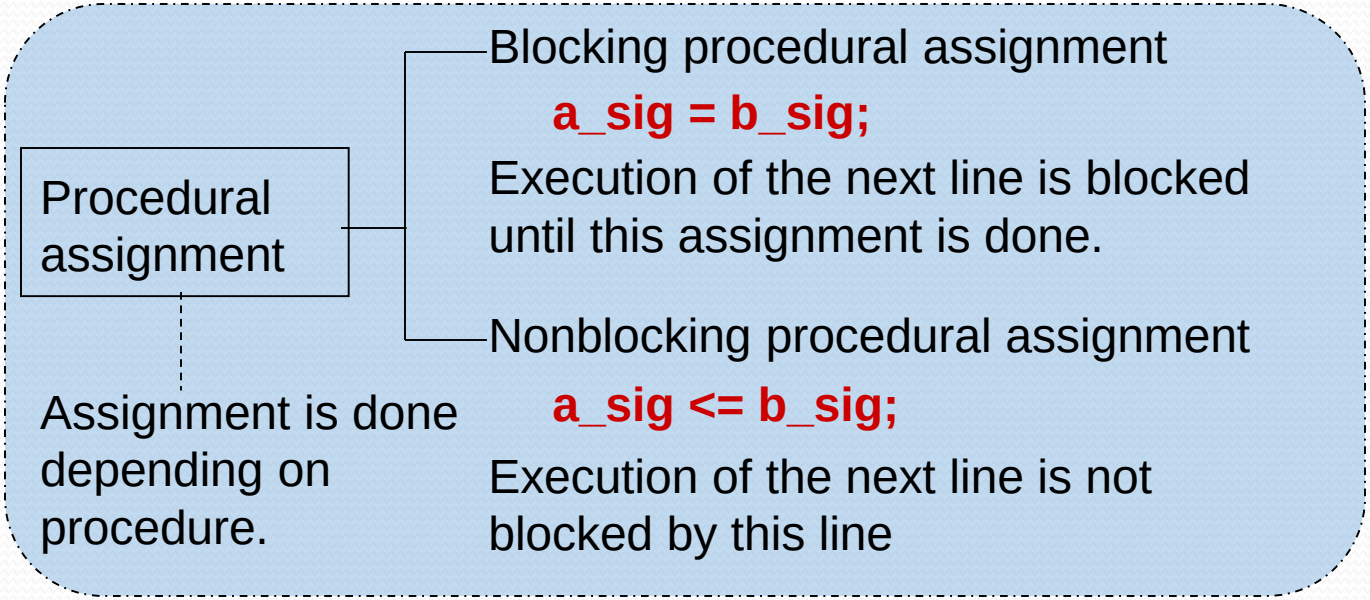
- **Continuous assignments** drive **nets**, nets are evaluated and updated whenever an input operand changes value.
- **Procedural assignments** update the value of **variables** under the control of the procedural flow constructs that surround them

Procedural assignment

- 2 types of procedural assignment:

  blocking and non-blocking procedural assignment

  -> specify different flow in a sequential block



Procedural assignment

- **Example**

```
//non_block1.v
module non_block1;
reg a, b, c, d, e, f;

//blocking assignments
initial begin
    a = #10 1; // a will be assigned 1 at time 10
    b = #2 0;  // b will be assigned 0 at time 12
    c = #4 1;  // c will be assigned 1 at time 16
end
//non-blocking assignments
initial begin
    d <= #10 1; // d will be assigned 1 at time 10
    e <= #2 0;  // e will be assigned 0 at time 2
    f <= #4 1;  // f will be assigned 1 at time 4
end
endmodule
```

Procedural assignment

- Blocking procedural assignment: generally used to describe the combinational logic - always @*
- Nonblocking procedural assignment: generally used to described the sequential logic -always @ (posedge/negedge clock).
- **Blocking assignment:** is executed before the execution of other statements that follow it in sequential block, but can not prevent other statements in parallel block
  - *begin…end: sequential block*
  - *fork……join: parallel block*

Procedural assignment

- The order of the execution of distinct **non-blocking assignments** to a given variable shall be preserved

```
module multiple;
reg a;

initial a = 1;
// The assigned value of the reg is determinate
initial begin
    a <= #4 0;    // schedules a = 0 at time 4
    a <= #4 1;    // schedules a = 1 at time 4
end               // At time 4, a = 1
endmodule
```

```
module multiple2;
reg a;

initial a  = 1;
initial a <= #4 0;     // schedules 0 at time 4
initial a <= #4 1;     // schedules 1 at time 4

// At time 4, a = ??
// The assigned value of the reg is indeterminate
endmodule
```

Procedural assignment

- **Continuous assignment**: assign values to nets
- **Procedural assignment**: assign values to variables
- Two additional forms: **assign/deassign** and **force/release**

  -> **Procedural continuous assignment**

	assign a = b	**force a = b**
Left-hand side	• Variable, • Concatenation of variable • ~~Memory word~~ • ~~Bit-select, part-select of variable~~	•Variable, Net, • Concatenation of any of above. • ~~Memory word~~ • ~~Bit-select, part-select of variable~~

Procedural assignment

- ***assign/deassign*** and ***force/release*** statement

-> are procedural statements that allow expression to be driven continuously onto variables or net

-> Override all procedural assignments

assign – deassign

->apply to variables

```
module dff (q, d, clear, preset, clock);
output q;
input d, clear, preset, clock;
reg q;

always @(clear or preset)
    if (!clear)
        assign q = 0;
    else if (!preset)
        assign q = 1;
    else
        deassign q;

always @(posedge clock)
    q = d;
endmodule
```

Procedural assignment

- ***force*** statement : override statements below until **release** happens:
  - ***assign*** (procedural continuous assignment)
  - gate output, module outputs, & **assign** (continuous assignment)

force – release
-> apply to net and variable

```
module test;
reg a, b, c, d;
wire e;

and and1 (e, a, b, c);

initial begin
    $monitor("%d d=%b,e=%b", $stime, d, e);
    assign d = a & b & c;
    a = 1;
    b = 0;
    c = 1;
    #10;
    force d = (a | b | c);
    force e = (a | b | c);
    #10;
    release d;
    release e;
    #10 $finish;
end
endmodule
```

Result:
0 d=0,e=0
10 d=1,e=1
20 d=0,e=0

Sequential & Parallel block

- In structured procedure, if there are more than one statement to execute, they must be grouped in one block.
  - "**begin…end**": **sequential block**, statements are executed in serial, in the order they are written
  - "**fork…join**": **parallel block**, statements are executed concurrently

Sequential & Parallel block

- Sequential block

```
parameter d = 50;     // d declared as a parameter and
reg [7:0] r;          // r declared as an 8-bit reg

begin     // a waveform controlled by sequential delay
    #d r = 'h35;
    #d r = 'hE2;
    #d r = 'h00;
    #d r = 'hF7;
    #d -> end_wave; //trigger an event called end_wave
end
```

- Parallel block

```
fork
    #50 r = 'h35;
    #100 r = 'hE2;
    #150 r = 'h00;
    #200 r = 'hF7;
    #250 -> end_wave;
join
```

Sequential & Parallel block

- **Nested blocks**: Sequential and Parallel block can be embedded within each other.

```
begin
        fork
                @Aevent;
                @Bevent;
        join
        areg = breg;
end
```

- **Named blocks**: If a block is named,

  - local variable, parameter can be declared for the block

  - the execution of the block can be disabled from anywhere in the design.

  - named blocks can be referenced by hierarchical names.

Sequential &Parallel block

- **Named blocks**

```
//Example: Find the first bit with a value 1 in flag
reg [15:0] flag;
integer i; //integer to keep count
initial
begin
        flag = 16'b 0010_0000_0000_0000;
        i = 0;
        begin: block1 //The main block inside while is named block1
                while(i < 16)
                begin
                if (flag[i])
                     disable block1; //disable block1 because you found true bit.
                i = i + 1;
                end
        end
end
```

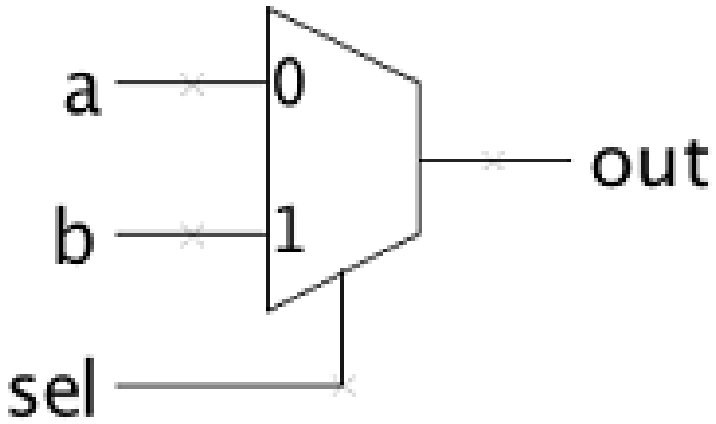
Behavioral Statements

- Conditional statement

  if - else

(combinational circuit)

```
module comb(input a, b, sel,
            output reg out);
  always @(*) begin
    if (sel) out = b;
    else out = a;
  end
endmodule
```



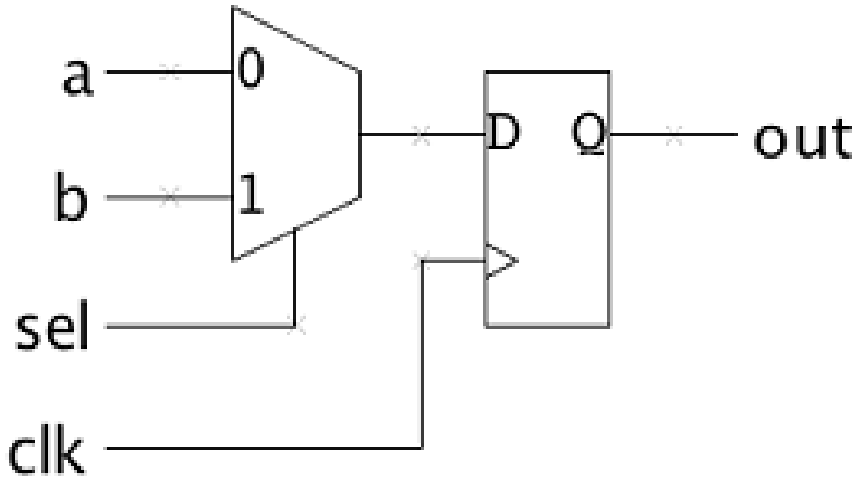
Behavioral Statements

- Conditional statement

  if - else

(sequential circuit)

```
module seq(input a, b, sel, clk,
           output reg out);
  always @(posedge clk) begin
    if (sel) out <= b;
    else out <= a;
  end
endmodule
```



Statements

- Conditional statement

  if-else-if

```
// declare regs and parameters
reg [31:0] instruction, segment_area[255:0];
reg [7:0] index;
reg [5:0] modify_seg1,
    modify_seg2,
    modify_seg3;
parameter
    segment1 = 0,  inc_seg1 = 1,
    segment2 = 20, inc_seg2 = 2,
    segment3 = 64, inc_seg3 = 4,
    data = 128;

// test the index variable
if (index < segment2) begin
    instruction = segment_area [index + modify_seg1];
    index = index + inc_seg1;
end
else if (index < segment3) begin
    instruction = segment_area [index + modify_seg2];
    index = index + inc_seg2;
end
else if (index < data) begin
    instruction = segment_area [index + modify_seg3];
    index = index + inc_seg3;
end
else
    instruction = segment_area [index];
```

Statements

- Case statement

```
case (op)
    2'b00: y = a + b;
    2'b01: y = a – b;
    2'b10: y = a ^ b;
    default: y = 'hxxxx;
endcase
```

case	:compare each bit
casez	:treat z or ? as do-not-care value (? is a shorthand for z)
casex	:treat x ,z,? as do-not-care value

```
reg [7:0] ir;

casez (ir)
    8'b1???????: instruction1(ir);
    8'b01??????: instruction2(ir);
    8'b00010???: instruction3(ir);
    8'b000001??: instruction4(ir);
endcase
```

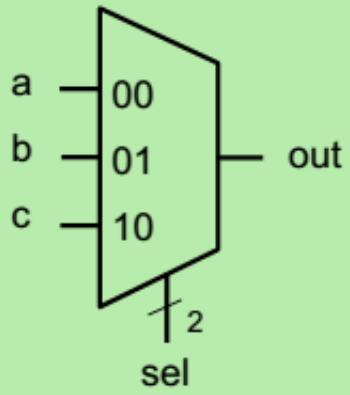
```
reg [7:0] r, mask;

mask = 8'bx0x0x0x0;
casex (r ^ mask)
    8'b001100xx: stat1;
    8'b1100xx00: stat2;
    8'b00xx0011: stat3;
    8'bxx010100: stat4;
endcase
```

What is the result if r = 8'b01100110?

Behavioral Statements

- **Incomplete Specification**

Goal:



3-to-1 MUX
('11' input is a don't-care)

Proposed Verilog Code:

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

  always @(a or b or c or sel)
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

Is this a 3-to-1 multiplexer?

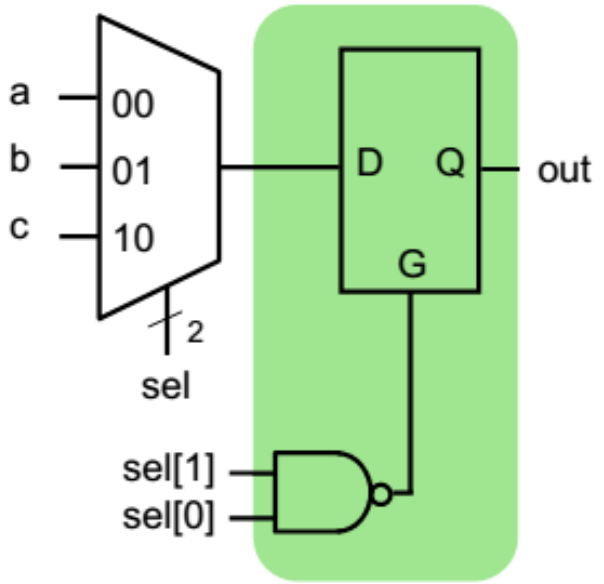
Behavioral Statements

- **Incomplete Specification infers LATCHs**

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

  always @(a or b or c or sel)
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

if `out` is not assigned during any pass through the always block, then the previous value must be retained!

Synthesized Result:



- Latch memory "latches" old data when G=0 (we will discuss latches later)
- In practice, we almost *never* intend this

Behavioral Statements

- **Avoiding Incomplete Specification**

- Precede all conditionals with a default assignment for all signals assigned within them…

```
always @(a or b or c or sel)
  begin
    out = 1'bx;
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

```
always @(a or b or c or sel)
 begin
 case (sel)
     2'b00: out = a;
     2'b01: out = b;
     2'b10: out = c;
     default: out = 1'bx;
   endcase
  end
endmodule
```

- …or, fully specify all branches of conditionals and assign all signals from all branches
  - For each `if`, include `else`
  - For each `case`, include `default`

Statements

- Loop statement: *forever, repeat, while, for*

  Control the execution of a statement zero, one, or more times

- **Forever**, **repeat**, **while**: In many cases, these are not synthesizable or there may be heavy tool dependency. Therefore, avoid using these statements in a synthesizable code.

Statement

- **For (var_assign; expression; var_assign) statement**

```
//an increasing sequence of values on an output
reg [3:0] i, output;

for ( i = 0 ; i <= 15 ; i = i + 1 ) begin
      output = i;
      #10;
end
```

- If loop instructions are used in a module which has to be synthesized, care must be taken.
- Be aware what structure will be generated with loop instructions. (They are different from C programming language.)

Statements

- **while (expression) statement:**
  execute statement while expression is true

```
begin : count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = rega;
    while (tempreg) begin
        if (tempreg[0])
            count = count + 1;
        tempreg = tempreg >> 1;
    end
end
```

-> Count the number of logic 1 values in reg a

Statements

- **repeat (expression) statement:**

  execute statement a fixed number of times

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;

begin : mult
    reg [longsize:1] shift_opa, shift_opb;
    shift_opa = opa;
    shift_opb = opb;
    result = 0;
    repeat (size) begin
        if (shift_opb[1])
            result = result + shift_opa;
        shift_opa = shift_opa << 1;
        shift_opb = shift_opb >> 1;
    end
end
```

-> Implement a multiplier

Statements

- **forever statement:** execute statement forever

Example 1: Clock generation , use forever loop instead of always block

```
reg clock;
initial begin
    clock = 1'b0;
    forever #10 clock = ~clock; //Clock with period of 20 units
end
```

Example 2: Synchronize two register values at every positive edge of lock

```
reg clock;
reg x, y;
initial
    forever @(posedge clock) x = y;
```

Procedural timing control

- How does the behavioral model advance time?
  - **Delay control #**: delay a specific amount of time
  - **Event control @**:delay until an event occurs (“posedge”, “negedge”, or any change)

    - edge-sensitive timing control
  - **Statement wait**: delay until an event occur

    - level-sensitive timing control
- In Verilog, if there are no timing control statements, the simulation time does not advance.

Procedural timing control

- Examples

```
//delay-based timing control
#10 rega = a+b;
reg a = #10 a+b;  //intra-assignment delay

//event-based timing control
```

- @(clock) q = d; //q = d is executed whenever signal clock changes value
- @(posedge clk) q=d;
- q = @(posedge clock) d;
- @(a, b, c) y = (a & b) | (b & c) | (a & c); //equivalent @(a or b or c)
- @(*) // or @*, equivalent to @ (a or b or c or d or f)
  y = (a & b) | (c & d) | myfunction(f);

```
//wait control
wait(!enable) #10 a = b;
```

Procedural timing control

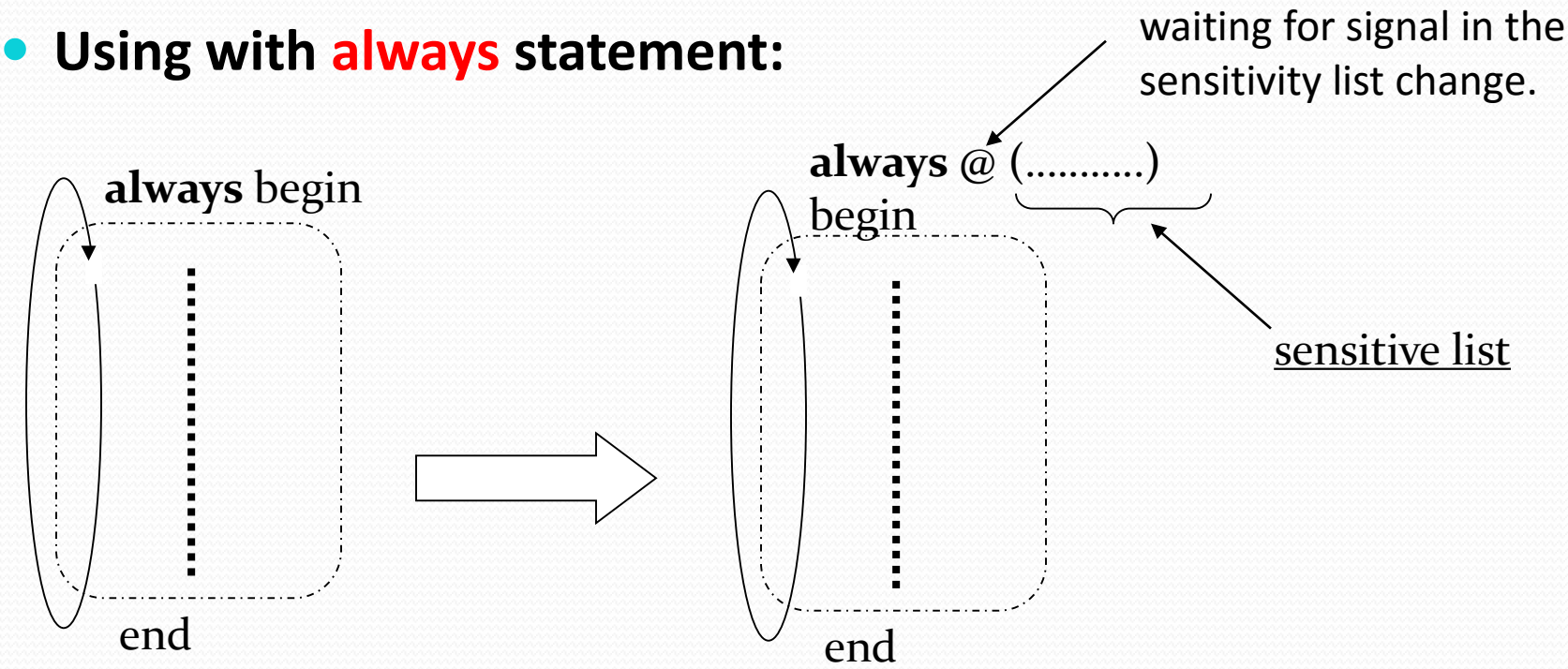
- Examples

```
fork
      begin: event_expr
            @ev1;  //named-event
            repeat(3) @trig;
            #d action (areg, breg);
      end
      @reset disable event_expr;
join
```

- The **event_expr** block waits for one occurrence of event **ev1** and three occurrences of **event trig**; plus a delay of **d** time units, the task **action** executes.
- When event **reset** occurs, regardless of events within the sequential block, the **fork-join** block terminates

Procedural timing control

- **Using with always statement:**

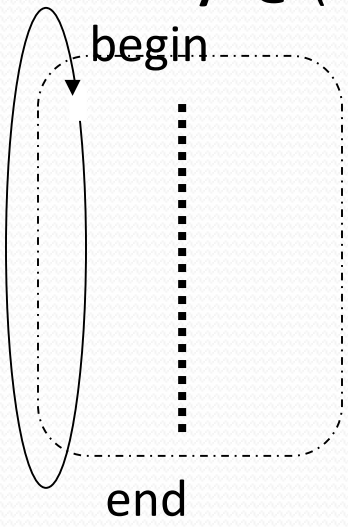


always creates an infinite loop

By adding "**@** ( )" to **always** statement: means "wait for change of signals listed in the sensitivity list" and when the change takes place, sequential block (from begin to end) is executed.

Procedural timing control

- **Using with always statement:**

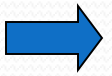


sensitive list

```
negedge: transition from 1 to x, z, or 0,
         and from x or z to 0
posedge: transition from 0 to x, z, or 1,
         and from x or z to 1
```

description	timing
always @ (posedge clk)	when signal clk rises
always @ (negedge rst_n)	when signal rst_n falls
always @ (a or b)	when signal a or b change

Do not mix single-edge (posedge / negedge) and double-edge event in an event control.



~~always @ (a or posedge clk)~~

Examples

- **4 to 1 mux**

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

output out;
input i0, i1, i2, i3;
input s1, s0;
//output declared as register
reg out;

//All input signals that cause a recomputation of out to
//occur must go into the always @(...) sensitivity list.
always @(s1 or s0 or i0 or i1 or i2 or i3) begin
        case ({s1, s0})
                2'b00: out = i0;
                2'b01: out = i1;
                2'b10: out = i2;
                2'b11: out = i3;
                default: out = 1'bx;
        endcase
end
endmodule
```

Examples

- **4-bit counter**

```
module counter(Q , clock, clear);

output [3:0] Q;
input clock, clear;
//output defined as register
reg [3:0] Q;

always @( posedge clear or negedge clock)
begin
        if (clear)
        Q <= 4'd0; //Nonblocking assignments are recommended
                        //for creating sequential logic such as flipflops
        else
        Q <= Q + 1; // Modulo 16 is not necessary because Q is a
                                // 4-bit value and wraps around.
end
endmodule
```

Summary

- Behavioral model describes a circuit in terms of the algorithm it implements
- **always**: basic structures in behavioral modeling
- **Procedural assignment**: assign value to variables
- 3 ways of **timing control** in Verilog:

  - Delay-based

  - Event-based (edge-sensitive)

  - Wait statement (level-sensitive)

Summary

- All procedures in Verilog HDL are specified within one of these statement:

- **initial** construct
- **always** construct
- **Task**
- **Function**

(Tasks & Functions are discussed later)

Summary: **Tips from UC San Diego Univ.**

- Think of hardware when writing RTL
- Understand "concurrency"
- Write proper combinational procedural assignments to avoid latches
  - Sensitivity list: all signals (wires, regs) on the RHS of "=";
  - Fully describe the behaviors of all signals:
    - if/else: always have an "else";
    - case: always use "default" clause;
- Properly use sequential process
  - always reset a FF: async reset vs. sync reset
  - signals can not appear in multiple sequential process
- Assignments
  - Use blocking assignments (=) in combinational procedural assignments
  - Use non-blocking assignments (<=) in sequential procedural assignments
- use wire/reg data type
  - wire: signals assigned using continuous assignment; port connections;
  - reg: signals assigned using procedural assignments (either comb. or seq.)
- One module in one file, module name same as file name Use lower-case for signals, upper-case ONLY for parameters

Content

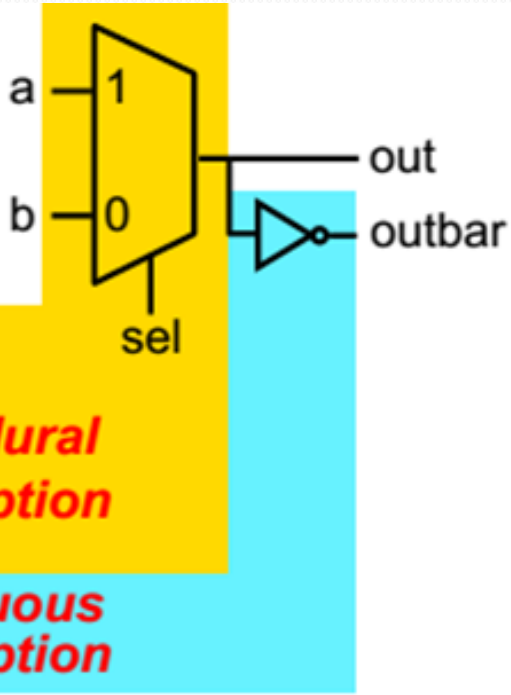
Chapter 4: Behavioral modeling – Part 2

- Boolean-equation-based behavioral model
- Continuous assignment
- Expressions in Verilog
- Cyclic behavioral models with always block
- initial & always construct
- Procedural assignment
- Blocking and non-blocking assignment
- Sequential and Parallel blocks
- Timing control mechanism in Verilog
- Conditional, Case, Loop Statements
- Related modeling techniques

Mixed model

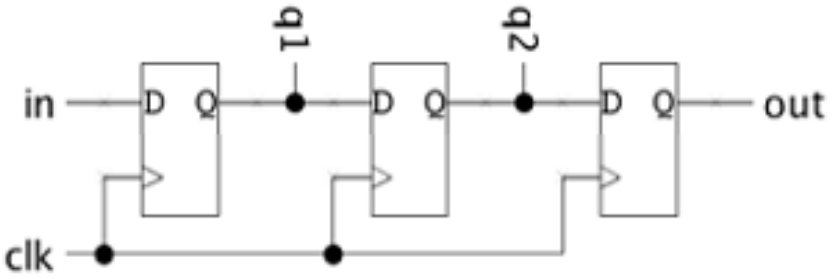
- *Procedural* and *continuous assignment* usually co-exist within a module
- Different modeling level (structural, behavioral) can be mixed in a Verilog module.

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

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

  assign outbar = ~out;

endmodule
```



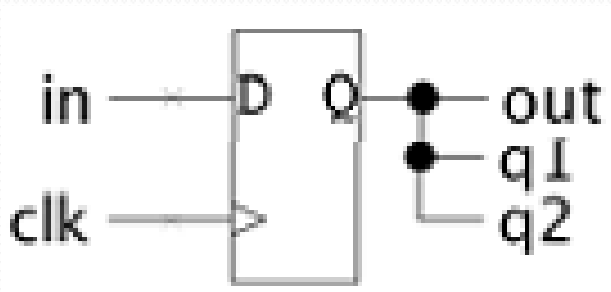
Procedural assignment

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



"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."

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



"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1*.
After that, *out* = *q2*.
Therefore *out* = *in*."

Model combinational logic

- Using always with procedural assignment

  - **The rules for specifying combinational logic using procedural statements**
    - Every element of the input set must be in the sensitivity list
    - The combinational output must be assigned in *every* control path

```
module mux (f, sel, b, c);
    output   f;
    input    sel, b, c;
    reg      f;

    always @ (sel or b or c)
             if (sel == 1)
                  f = b;
             else
                  f = c;
endmodule
```

So, we're saying that if any input changes, then the output is re-evaluated. — That's the definition of combinational logic.

Model combinational logic

- If you don't follow the rules...? … you're dead meat
  --> Create LATCH circuit

```
module blah (f, g, a, b, c);
    output   f, g;
    input    a, b, c;
    reg      f, g;

    always @ (a or b or c)
            if (a == 1)
                f = b;
            else
                g = c;
endmodule
```

This says: as long as a==1, then f follows b. (i.e. when b changes, so does f.) But, when a==0, f remembers the old value of b.

Combinational circuits don't remember anything!

What's wrong?

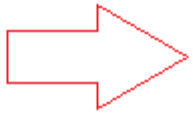
f doesn't appear in *every* control path in the always block (neither does g).

Model combinational logic

```
always @(coke or cola) begin
      if (coke)
            blah1 = 1;
      else if (cola > 2'b01)
            blah2 = coke;
      else if ( …
      …


end
```

```
always @(coke or cola) begin
      blah1 = 0;
      blah2 = 0;
      if (coke)
            blah1 = 1;
      else if (cola > 2'b01)
            blah2 = coke;
      else if ( …
      …

end
```

```
  always @(a or b or c or sel)
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```



```
always @(a or b or c or sel)
 begin
 case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    default: out = 1'bx;
   endcase
 end
endmodule
```

Model combinational logic

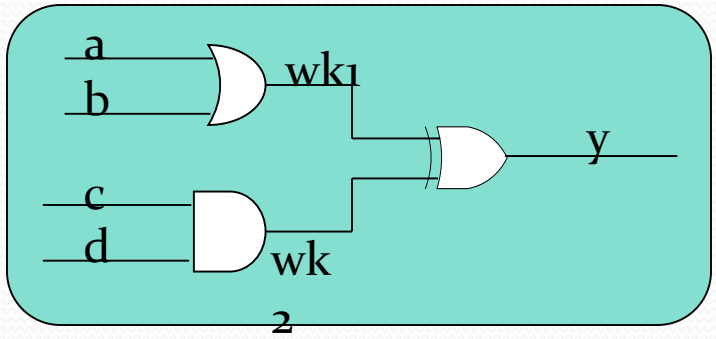
- **Your Verilog for combination stuff will look like this:**

```
module blah (<output names>, <input names>);
    output   <output names>;
    input    <input names>;
    reg      <output names>;

    always @ (<names of all input vars>)
        begin
            < LHS = RHS assignments>
            < if ... else  statements>
            < case statements >
        end
endmodule
```

Recommended: **always @(*)**

Model combinational logic

To generate combinational logic as shown on the right, Style 1 is recommended compared to Style 2 because of the missing inputs in sensitive list.



Style 1

```
reg a, b, c, d, y;
reg wk1, wk2;

always @ ( *)
   begin
      wk1 = a | b;
      wk2 = c & d;
      y = wk1 ^ wk2;
  end
```

Style 2

```
reg a, b, c, d, y;
reg wk1, wk2;

always @ ( a or b or c or d or wk1 or wk2 )
   begin
      wk1 <= a | b;
      wk2 <= c & d;
      y <= wk1 ^ wk2;
  end
```

Model combinational logic

Using @ to model combinational logic

Best style

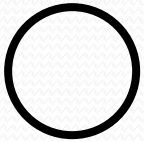
```
always @ (*)
  begin
    wk1 = a | b;
    wk2 = c & d;
    y = wk1 ^ wk2;
  end
```

Not recommended style (use non-blocking)

```
always @ ( a or b or c or d or wk1 or wk2)
  begin
    wk1 <= a | b;
    wk2 <= c & d;
    y <= wk1 ^ wk2;
  end
```

Acceptable style

```
always @ ( a or b or c or d )
  begin
    wk1 = a | b;
    wk2 = c & d;
    y = wk1 ^ wk2;
  end
```



Wrong / buggy style (use non-blocking and miss wk1, wk2)

```
always @ ( a or b or c or d)
  begin
    wk1 <= a | b;
    wk2 <= c & d;
    y <= wk1 ^ wk2;
  end
```

Model sequential logic

- Latch
- Flip-flop
- Register
- Counter
- Timing generator
- …..

How would you model D-latch and D flip-flop?

- Asynchronous reset D-type flip-flop
- Synchronous reset D type flip-flop
- D-latch

Model sequential logic

Some points:

- Most often modeled with always block
- Verilog code in **initial** block *can not be synthesized*
- Using @ construct
  - Sensitivity list contains **single *edge*** or ***double edge*** events
  - Reset optional.

Model sequential logic

- How would you model a D flip-flop and D latch?

```
// asynchronous reset D-type flip-flop
module DFF_async_reset (clk, reset_n, d, q);
input clk, reset_n, d;
output reg q;
always @(posedge clk or negedge reset_n)
   if (!reset_n) q <= 0;
   else q <= d;
endmodule
```

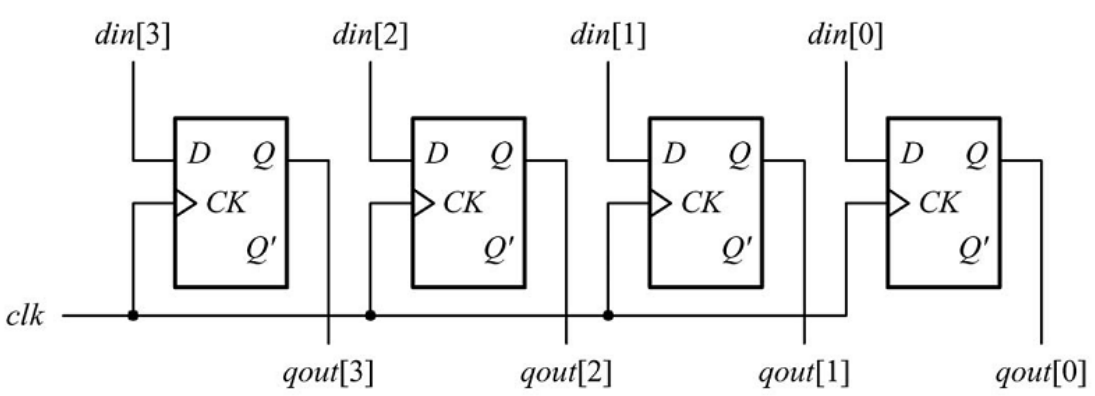
```
// synchronous reset D-type flip-flop
module DFF_sync_reset (clk, reset, d, q);
input clk, reset, d;
output reg q;
always @(posedge clk)
    if (reset) q <= 0;
    else q <= d;
endmodule
```

```
//D latch
module D_latch (enable, d, q);
input enable, d;
output reg q;
always @(enable or data)
    if (enable) q  = data;
endmodule
```

Model sequential logic

- Data register

```
// an n-bit data register
module register(clk, din, qout);
parameter N = 4; // number of bits
input [N-1:0] din;
output reg [N-1:0] qout;

always @(posedge clk)  qout <= din;
endmodule
```



Model sequential logic

- Data register

```
// an N-bit data register with synchronous load and
// asynchronous reset
module register_4 (clk, load, reset_n, din, qout);
parameter N = 4; // number of bits
input clk, load, reset_n;
input [N-1:0] din;
output reg [N-1:0] qout;

always  @(posedge clk or negedge reset_n)
        if (!reset_n)    qout <= {N{1'b0}};
        else if (load)   qout <= din;
endmodule
```

Synthesizable coding

- Always keep in mind what sort of implementation your design could map to.

  - No **case** statements *without* ***default*** case → create Latch in combinational model.

  - No **if** statements *without an* ***else*** case → create Latch in combinational model

- Limited loops
- No initial blocks
- Limited operators
  - **+** and **–** are the only arithmetic operators allowed
  - Try to avoid relational operators (>, ==) in favor of simpler logic
- Use **assign** statements when possible