Digital Design with Verilog

Verilog

Lecture 9: Behavioral Modeling – Part 1

Introduction

- Designers need to be able to evaluate the tradeoffs of various architectures and algorithms before they decide on the optimum architecture and algorithm to implement in hardware.

- Verilog provides designers the ability to describe design functionality in an algorithmic manner.

- Behavioral modeling represents the circuit at a very high level of abstraction.

- Design at this level resembles C programming more than it resembles digital circuit design.

Learning Objectives

- Understanding Behavioral Modeling
- Structured procedures “always” and “initial”.
- Define blocking and non-blocking procedural assignments.
- Describe event-based timing control mechanism in behavioral modeling.
- Use level-sensitive timing control mechanism in behavioral modeling.

Procedural Blocks

- All procedural signal assignments must be enclosed within a procedural block.
- Verilog has two types of procedural blocks: initial and always
- Each always and initial statement represents a separate activity flow in Verilog.
- Each activity flow starts at simulation time 0.
- The statements always and initial cannot be nested.

initial

- An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again.
- If there are multiple initial blocks, each block starts to execute concurrently at time 0.
- Each block finishes execution independently of other blocks.
- Multiple behavioral statements must be grouped, typically using the keywords begin and end.

initial

- An initial block is not used to model synthesizable behavior.
- It is instead used within testbenches to either set the initial values of repetitive signals.
- Typically used for initialization, monitoring, waveforms and to execute one-time executable processes.

Example: initial

```
module stimulus;
reg x,y, a,b, m;

initial
    m = 1'b0;
    //single statement; does not need to be grouped
initial
    begin
        #5 a = 1'b1;
        //multiple statements; need to be grouped
        #25 b = 1'b0;
    end
initial
    begin
        #10 x = 1'b0;
        #25 y = 1'b1;
    end
initial
    #50 $finish;
endmodule
```

always

- Model a block of activity that is repeated continuously in a digital circuit
- An always block will execute forever, or for the duration of the simulation. Equivalent to an infinite loop
- Stopped only by $finish (power-off) or $stop (interrupt)
- An always block can be used to model synthesizable circuits in addition to non-synthesizable behavior in test benches.

Example: always

```
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

- Verilog uses procedural assignment to model signal assignments that are based on an event.
- An event is most commonly a transition of a signal.
- This provides the ability to model sequential logic circuits such as D-flip-flops and finite state machines by triggering assignments of a clock edge.
- A procedural assignment can also be used to model combinational logic circuits by making signal assignments when any of the inputs to the model change.

Procedural Assignment

- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.

- Procedural assignment also supports standard programming constructs such as if-else decisions, case statements, and loops.

- These are unlike continuous assignments, where one assignment statement can cause the value of the right-hand-side expression to be continuously placed onto the left-hand-side net.

Procedural Assignments Cont'd

- The left-hand side of a procedural assignment <lvalue> can be one of the following:
  - A reg, integer, real, or time register variable or a memory element
  - A bit select of these variables (e.g., addr[0])
  - A part select of these variables (e.g., addr[31:16])
  - A concatenation of any of the above

- The right-hand side can be any expression that evaluates to a value.

- There are two types of procedural assignment statements: blocking and nonblocking.

Blocking Assignments

- A blocking assignment is denoted with the = symbol and the evaluation and assignment of each statement takes place immediately.

- When this behavior is coupled with a sensitivity list that contains all the inputs to the system, this approach can model synthesizable combinational logic circuits.

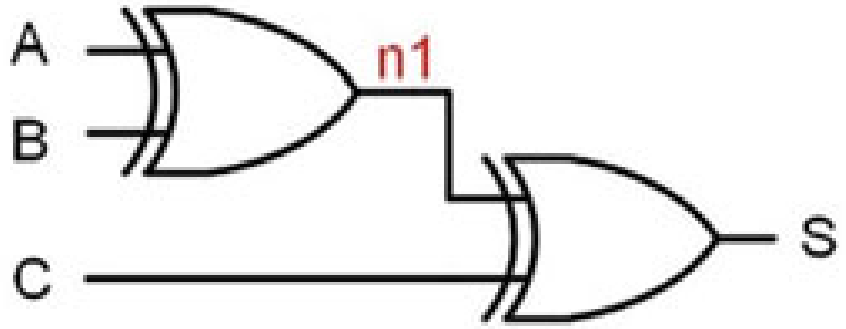
Blocking Assignments

Why should we use blocking assignments instead of continuous assignments to model combinational logic?

- Blocking statements have more advanced programming constructs are supported within Verilog procedural blocks.

Blocking Assignments

```
module BlcokingExample ( output reg S,
                         input wire A, B, C);


    reg n1;

    always @(A, B, C)
        begin
            n1 = A ^ B; // statement 1
            S = n1 ^ C; // statement 2
        end
endmodule
```



Blocking Assignments Cont'd

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
x = 0; y = 1; z = 1; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //initialize vectors
#15 reg_a[2] = 1'b1; //Bit select assignment with delay
#10 reg_b[15:13] = {x, y, z} //Assign result of concatenation
to
// part select of a vector
count = count + 1; //Assignment to an integer (increment)
end
```

Blocking Assignments Cont'd

- Assignments can block execution in another concurrent construct also

```
always @(posedgeclk)
begin
    word[15:8] = word[ 7:0];
    word[ 7:0] = word[15:8];
end
//swap bytes in word???
```

```
always @(posedgeclk)
begin
    word[ 7:0] = word[15:8];
    word[15:8] = word[ 7:0];
end
//swap bytes in word???
```

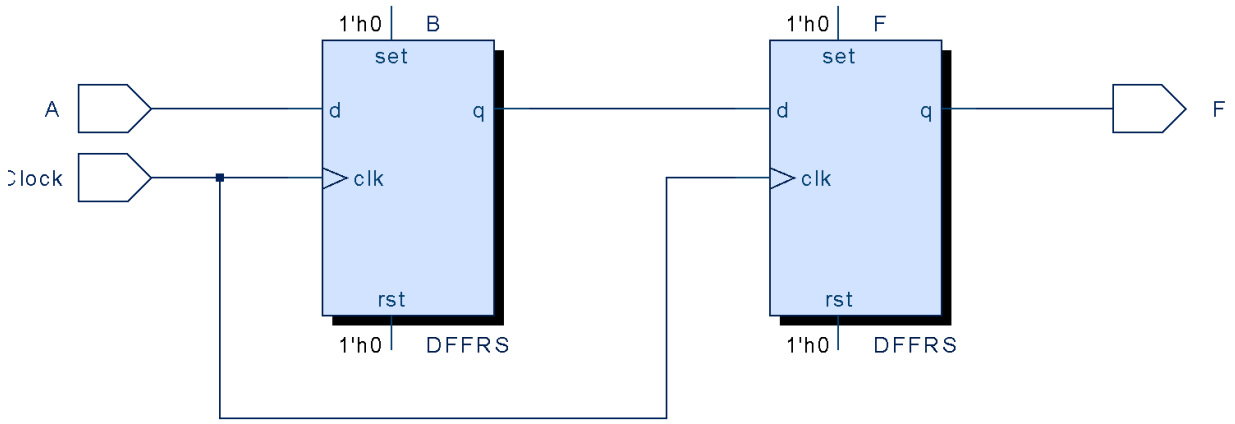
Non-blocking Assignments

- Evaluated and assigned in two steps:
  - The right-hand side is evaluated immediately
  - The assignment to the left-hand side is postponed until other evaluations in the current time step are completed
  - Execution flow within the procedure continues until a timing control is encountered (flow is not blocked)
  - Operator is <=

```
always @(posedgeclk)
begin
    word[15:8] <= word[ 7:0];
    word[ 7:0] <= word[15:8];
end
//can swap the order of statements
```

Non-blocking Assignments

```
module NonBlocking ( output reg F,
                     input wire A,
                     input wire Clock);

        reg B;

        always@(posedge Clock)
        begin
                B <= A;  //statement 1
                F <= B;  //statement 2
        end
endmodule
```



Non-blocking Assignment Cont'd.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
x = 0; y = 1; z = 1; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //Initialize vectors
reg_a[2] <= #15 1'b1; //Bit select assignment with delay
reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation
//to part select of a vector
count <= count + 1; //Assignment to an integer (increment)
end
```

Application of Non-Blocking Assignments

- They are used as a method to model several concurrent data transfers that take place after a common event.

```
always @(posedge clock)
begin
reg1 <= #1 in1;
reg2 <= @(negedge clock) in2 ^ in3;
reg3 <= #1 reg1; //The old value of reg1
end
```

Non-blocking Assignments Cont'd

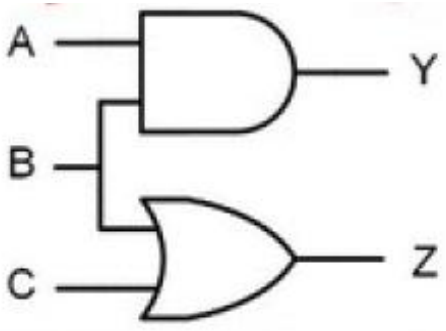
```
//Illustration 1: Two concurrent always blocks with blocking
//statements
always @(posedge clock)
a = b;
always @(posedge clock)
b = a;
//Illustration 2: Two concurrent always blocks with nonblocking
//statements
always @(posedge clock)
a <= b;
always @(posedge clock)
b <= a;
```

Blocking Vs Non-Blocking

- The difference between blocking and non-blocking assignments is subtle and is often one of the most difficult concepts to grasp.

```
module blocking ( output reg Y, Z,
                  input wire A, B, C);


        always@(A, B, C)
        begin
                Y = A & B;
                Z = B | C;
        end
endmodule
```

```
module nonblocking (output reg Y, Z,
                    input wire A, B, C);

        always@(A, B, C)
        begin
                Y <= A & B;
                Z <= B | C;
        end
endmodule
```



Blocking Vs Non-Blocking

- Both blocking and non-blocking assignments *can* produce the same results when they either contains a single assignment or a list of assignments that don't have any signal interdependencies.

- A *signal interdependency* refers to when a signal that is the target of an assignment (i.e., on the LHS of an assignment) is used as an argument (i.e., on the RHS of an assignment) in subsequent statements.

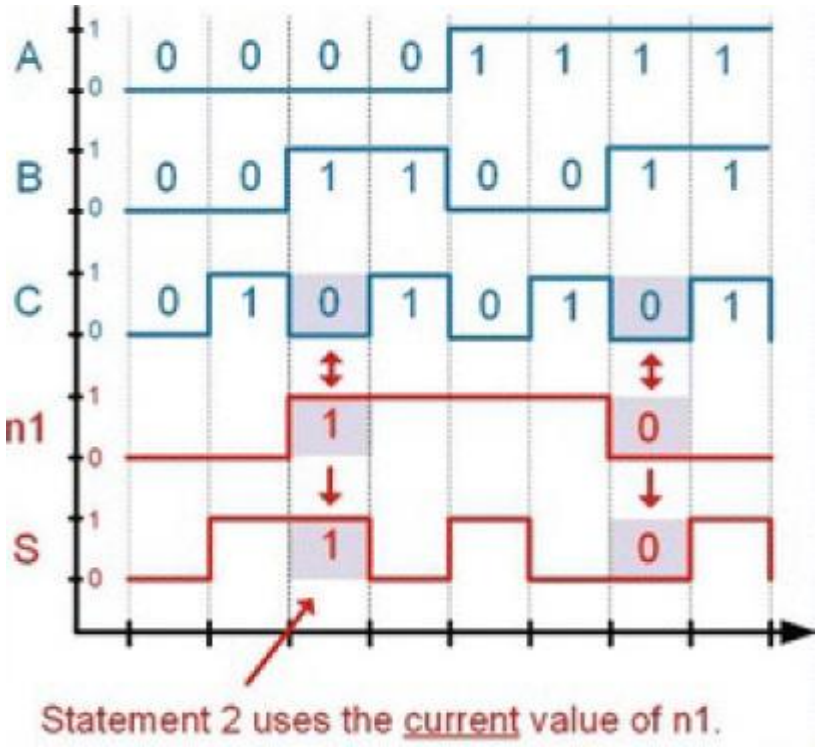
Blocking Vs Non-Blocking

```
module blocking ( output reg S,
                  input wire A, B, C);

reg n1;

always@(A, B, C)
begin
        n1 = A ^ B; //statement 1
        S  = n1 ^ C; //statement 2
end
endmodule
```

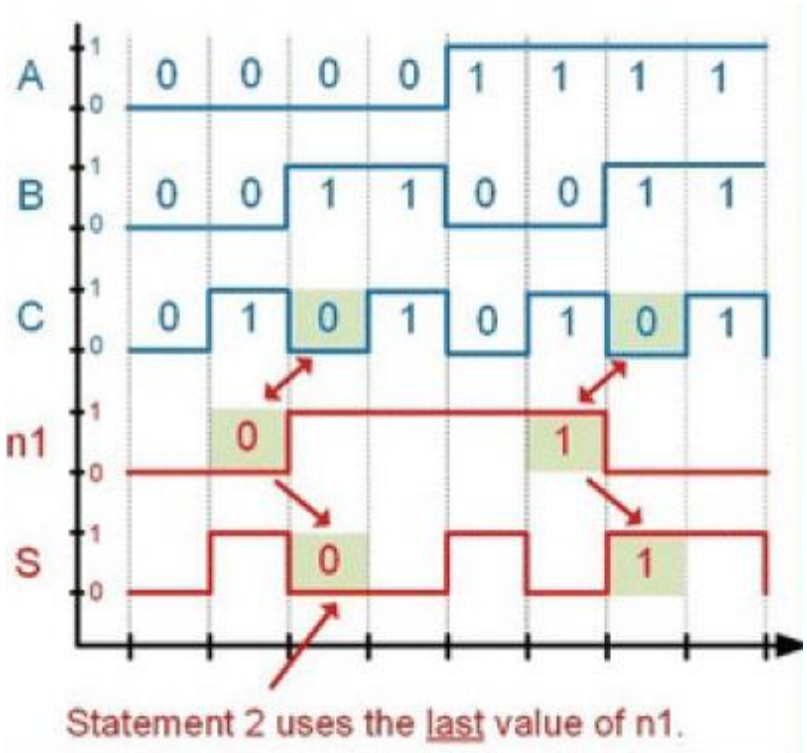
```
module nonblcoking (output reg S,
                    input wire A, B, C);

reg n1;

always@(A, B, C)
begin
        n1 <= A ^ B;  //statement 1
        S  <= n1 ^ C; //statement 2
end
endmodule
```

Blocking Vs Non-Blocking

Timing Waveforms:
Blocking Assignment



Non-Blocking Assignment

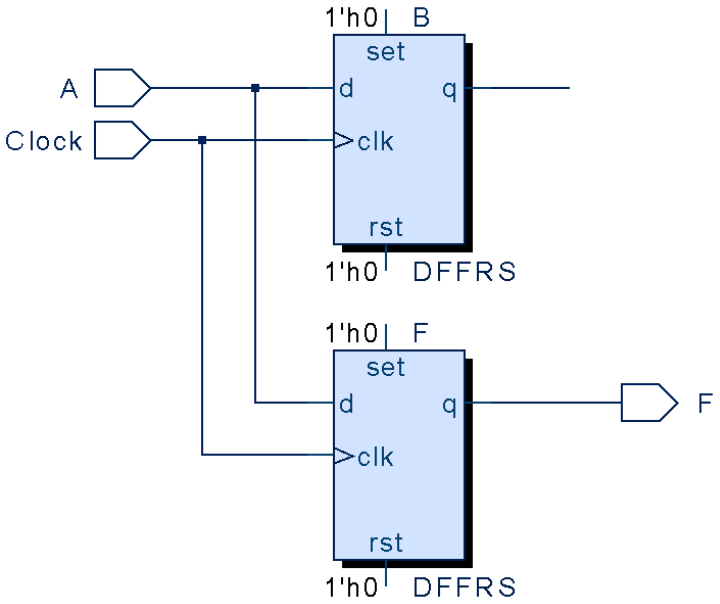


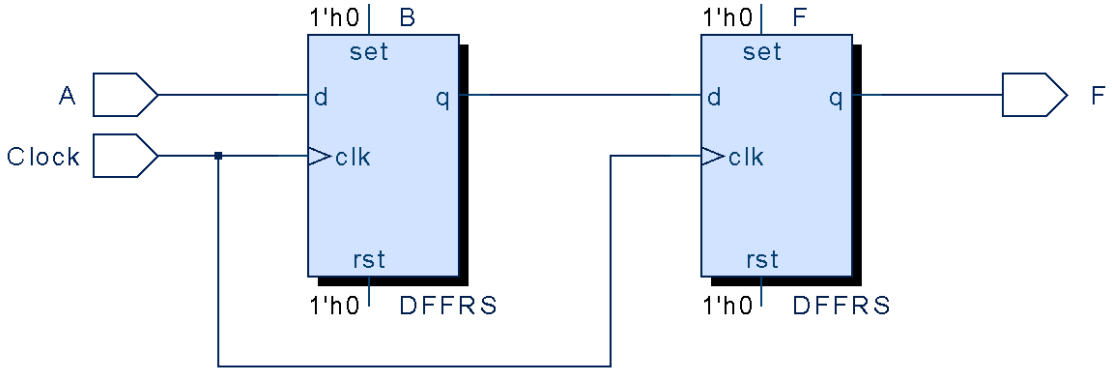
Blocking Vs Non-Blocking

```
module blcoking (output reg F,
                 input wire A,
                 input wire Clock);

reg B;

always@(posedge Clock)
begin
        B = A;
        F = B;
end
endmodule
```

```
module nonblcoking (output reg F,
                    input wire A,
                    input wire Clock);

reg B;

always@(posedge Clock)
begin
        B <= A;  //statement 1
        F <= B;  //statement 2
end
endmodule
```

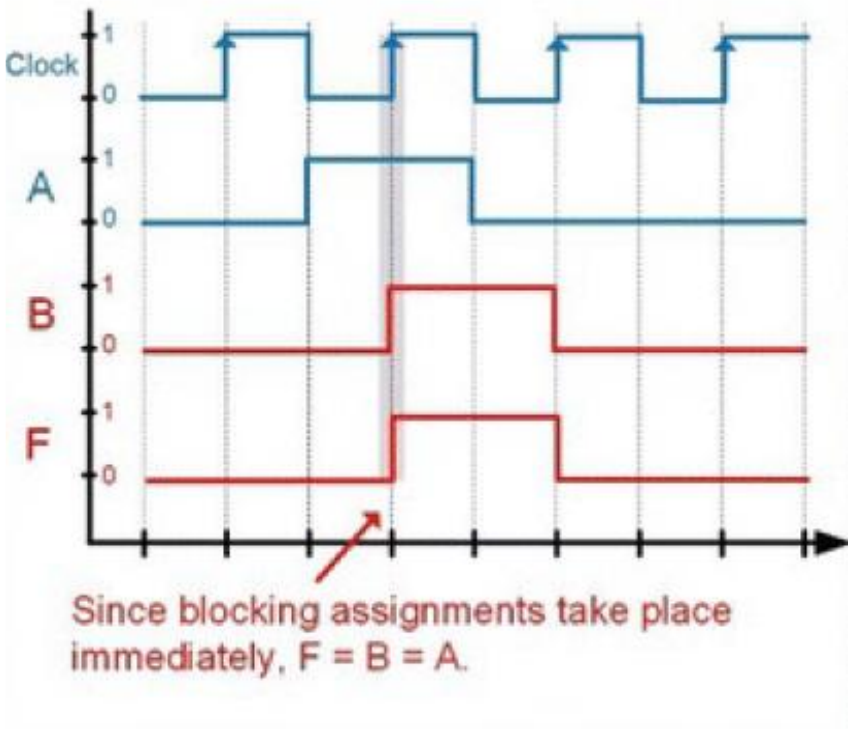
Blocking Vs Non-Blocking



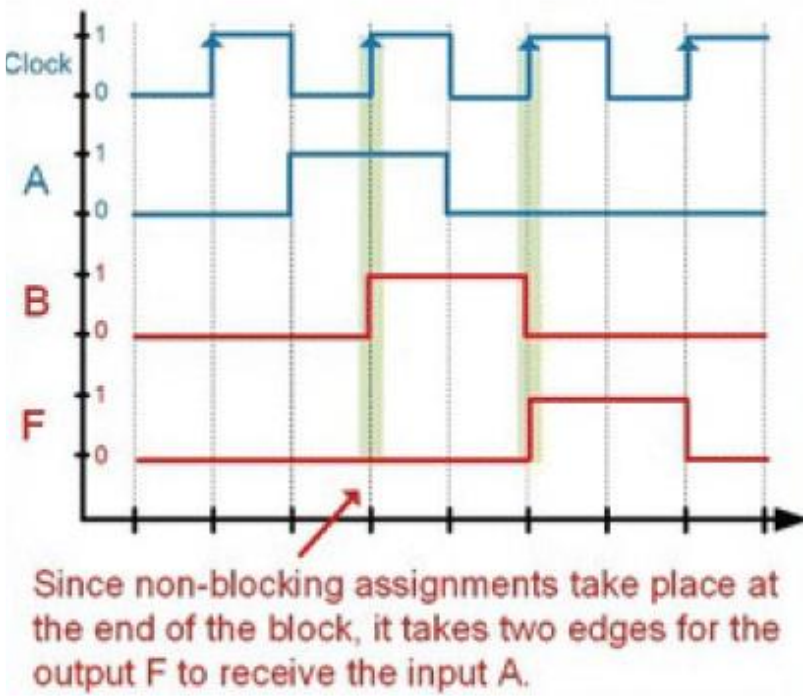


Blocking Vs Non-Blocking

Timing Waveforms: Blocking Assignment



Non-Blocking Assignment



Blocking Vs Non-Blocking

- There are two design guidelines that can make creating accurate, synthesizable models straightforward.

- They are:

  - When modeling **combinational logic**, use blocking assignments and list every input in the sensitivity list.

  - When modeling **sequential logic**, use *non-blocking assignments* and *only list the clock and reset lines* (if applicable) in the sensitivity list.

Timing Controls

Timing Controls

- Timing controls provide a way to specify the simulation time at which procedural statements will execute.

- There are three methods of timing control:
  - Delay-based timing control,
  - Event-based timing control, and
  - Level-sensitive timing control

Delay-Based Timing Control

- Delay based control in an expression specifies the time duration between when the statement is encountered and when it is executed.

- Regular delay control

  - Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment.

Delay-Based Timing Control

Regular delay control

```
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;
initial
begin
x = 0; // no delay control
#10 y = 1; // delay control with a number. Delay execution of
// y = 1 by 10 units
#latency z = 0; // Delay control with identifier. Delay of 20
units
#(latency + delta) p = 1; // Delay control with expression
#y x = x + 1; // Delay control with identifier. Take value of
y.
#(4:5:6) q = 0; // Minimum, typical and maximum delay values.
//Discussed in gate-level modeling chapter.
end
```

Delay-Based Timing Control

Intra-assignment delay control

```
//define register variables
reg x, y, z;
//intra assignment delays
initial
begin
x = 0; z = 0;
y = #5 x + z; //Take value of x and z at the time=0, evaluate
//x + z and then wait 5 time units to assign value
//to y.
end
//Equivalent method with temporary variables and regular delay control
initial
begin
x = 0; z = 0;
temp_xz = x + z;
#5 y = temp_xz; //Take value of x + z at the current time and
//store it in a temporary variable. Even though x and z
//might change between 0 and 5,
//the value assigned to y at time 5 is unaffected.
end
```

Intra-assignment delay control cont'd.

- Assigning delay to the right of the assignment operator
- The intra-assignment delay computes the right-hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable.
- Equivalent to regular delays with a temporary variable to store the current value of a right-hand-side expression

Delay-Based Timing Control

Zero delay control

- Procedural statements inside different always-initial blocks may be evaluated at the same simulation time.

- The order of execution of these statements in different always-initial blocks is nondeterministic and might lead to race conditions.

Delay-Based Timing Control

Zero delay control

- Zero delay control ensures that a statement is executed last, after all other statements in that simulation time are executed.

- This can eliminate race conditions, unless there are multiple zero delay statements, in which case again nondeterminism is introduced.

Zero delay control cont'd

```
initial
begin
x = 0;
y = 0;
end
initial
begin
#0 x = 1; //zero delay control
#0 y = 1;
end
```

Event-Based Timing Control

- An event is the change in the value on a register or a net.
- Events can be utilized to trigger execution of a statement or a block of statements.
- There are four types of event-based timing control:
  - regular event control,
  - named event control,
  - event OR control, and
  - level sensitive timing control.

Regular Event Control

- The @ symbol is used to specify an event control.
- Statements can be executed on changes in signal value or at a positive or negative transition of the signal value.

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does
//a positive transition ( 0 to 1,x or z,
// x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does
//a negative transition ( 1 to 0,x or z,
//x to 0, z to 0)
q = @(posedge clock) d; //d is evaluated immediately and assigned
//to q at the positive edge of clock
```

Named event control

- Verilog provides the capability to declare an event and then trigger and recognize the occurrence of that event.
- The event does not hold any data.
- A named event is declared by the keyword event. An event is triggered by the symbol ->.
- The triggering of the event is recognized by the symbol @.

Named event control

```
//This is an example of a data buffer storing data after the
//last packet of data has arrived.
event received_data; //Define an event called received_data
always @(posedge clock) //check at each positive clock edge
begin
if(last_data_packet) //If this is the last data packet
->received_data; //trigger the event received_data
end
always @(received_data) //Await triggering of event received_data
//When event is triggered, store all four
//packets of received data in data buffer
//use concatenation operator { }
data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```

Event OR control

- Sometimes a transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements. This is expressed as an OR of events or signals.

```
//A level-sensitive latch with asynchronous reset
always @( reset or clock or d)
//Wait for reset or clock or d to
change
begin
if (reset) //if reset signal is high, set q to 0.
q = 1'b0;
else if(clock) //if clock is high, latch input
q = d;
end
```

Example: Sensitivity List & Comma Operator

```
//A level-sensitive latch with asynchronous reset
always @( reset, clock, d)
//Wait for reset or clock or d to
change
begin
if (reset) //if reset signal is high, set q to 0.
q = 1'b0;
else if(clock) //if clock is high, latch input
q = d;
end
//A positive edge triggered D flipflop with asynchronous falling
//reset can be modeled as shown below
always @(posedge clk, negedge reset) //Note use of comma operator
if(!reset)
q <=0;
else
q <=d;
```

Example: Use of @* Operator

```
//Combination logic block using the or operator
//Cumbersome to write and it is easy to miss one input to the block
always @(a or b or c or d or e or f or g or h or p or m)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
//Instead of the above method, use @(*) symbol
//Alternately, the @* symbol can be used
//All input variables are automatically included in the
//sensitivity list.
always @(*)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
```

Sensitivity List

Level-Sensitive Timing Control

- Verilog allows level-sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed.

```
always
wait (count_enable) #20 count = count + 1;
```

- This monitors the variable count_enable and if it is zero, it will not enter. When it becomes one, then after 20 units of time it will increment the variable count.

- If count_enable stays at 1, then count is incremented every 20 units

References

- Chapter 7, Verilog HDL by Samir Palnitkar, Second Edition.

- Various online resources ☺

Thank you, done for today !!!