Behavior Model

Simulation Control

Behavior Modeling

• It may be used as a golden reference model for the RTL model
• In Verilog, behavior statements are for controlling the simulator’s behavior
  – Behavior statements may not be synthesizable
  – This depends on the synthesis tool you are using
  – In general, just assume they are not synthesizable
• If you have a behavior model and an RTL, equivalence checking between the two can be quite challenging
• Industry is pushing a different approach for high-level design model (eg. ESL: Electronic System Level design)
Behavior Statements

- **initial**: set initial value
- **always**
  - declare action block during simulation
- **assign ... deassign**
  - assign values to regs
- **force ... release**
  - assign (force) values to nets
- **@**: at ... do ...
- **event**
  - declare an event to be used with @
What is “initial” used for

Example 7.3 The behavior encapsulated by the always keyword in clock_gen1 implements a simple clock generator and generates the waveform shown in Figure 7.3. The initial behavior assigns the initial value to clock; the always behavior delays a half cycle before toggling the value of the clock. It then re-executes indefinitely. A second initial behavior assures that the simulator terminates after 1000 time steps (10 cycles of clock).

```
module demo_2 (sig_a, sig_b, sig_c, sig_d);
output sig_a, sig_b, sig_c, sig_d;
reg sig_a, sig_b, sig_c, sig_d;

initial // An “initial” behavior
begin
  sig_a = 0; // Procedural assignments
  sig_b = 1; // execute sequentially.
  sig_c = 0;
  sig_d = 0;
end
endmodule
```

```
module clock_gen1 (clock);
parameter Half_cycle = 50;
p parameter Max_time = 1000;
output clock;
reg clock;

initial
begin
  clock = 0;
always
begin
@half_cycle clock = !clock;
end
initial
@Max_time $finish;
endmodule
```

“always”

- `always @ (some events)`
  - Some events can be
    - Signal change value
    - Clock rising or falling
    - Or your own definition

- “always” follows an event-driven simulation style of thinking
Named events

Example 7.16 A typical implementation of a named event is shown in Demo_mod_A:

```verbatim
module Demo_mod_A (...);
    ...
    event something_happens; // Declaration of an abstract event
    always begin
        ...
        -> something_happens // Triggering of an abstract event
        end
endmodule

module Demo_mod_B(...);
    always @ (Top_Module.Demo_mod_A.something_happens)
        ...
        begin
            ...
        // do something when something_happens in Demo_mod_A
            end
endmodule
```

Named events

```verbatim
module Top (clock, data, q_out);
    input clock, data;
    output q_out;
    Ttalker M1 (clock);
    Receiver M2 (q_out, data);
endmodule

module Ttalker (clock);
    input clock;
    event do_it;
    always @ (posedge clock) -> do_it;
endmodule

module Receiver (q_out, data);
    input data;
    output q_out;
    reg q_out;
    always @ (Top.M1 do_it) q_out = data;
endmodule
```
assign/deassign

• Model level-sensitive behavior of a reg
  – If you want to affect the value of a net/wire
  – Then, you use force/release

• Some synthesis tool may not support them

• assign is like “bind two ports with a physical wire”
• deassign is like “taking away the physical wire”

• In hardware, you don’t have this situation of arbitrarily putting a wire and then taking it away
  – Hence, the effects are for controlling the simulator

---

assign- a mux example

Example 7.4 The multiplexer in Figure 7.5, mux4_PCA, uses a procedural continuous assignment to bind an assignment expression (event scheduling rule) to the target register variable, just as a continuous assignment binds an expression to a net.

```verilog
module mux4_PCA (a, b, c, d, select, y_out);  
input a, b, c, d;  
input [1:0] select;  
output y_out;  
reg y_out;  

always @(select)  
if (select == 0) assign y_out = a; else  
if (select == 1) assign y_out = b; else  
if (select == 2) assign y_out = c; else  
if (select == 3) assign y_out = d; else y_out = 1'b0;  
endmodule```

---
= and \( \leq \) are different from \texttt{assign}.

- \( = \) is called blocked assignment.
- \( \leq \) is called non-blocked assignment.
- They do not represent “physical wiring” effect like the \texttt{assign} statement.
- They are for assigning the values when the corresponding statements are executed by the simulator.
  - After the statement is executed, no more future effect until the statement is executed again.

\begin{center}
\begin{itemize}
  \item Blocked assignment
    \begin{itemize}
      \item One assignment get executed after another
      \item Use “\(=\)”
      \item \( A = \#5\ B;\ C = D; \)
      \item “\(C = D\)” will not be executed after 5 time units
    \end{itemize}
  \item Non-Blocked assignment
    \begin{itemize}
      \item Assignments get executed in parallel
      \item Use “\(\leq\)”
      \item \( A \leq \#5\ B;\ C \leq D; \)
      \item “\(C = D\)” does not wait!
    \end{itemize}
\end{itemize}
\end{center}
Another example

```
initial begin
  A = 1;
  B = 0;
  ...
  A <= B; // Uses B = 0
  B <= A; // Uses A = 1
end
```

The result of simulating the behaviors is that A and B swap values, leaving A=0 and B=1. The order of the non-blocking statements has no effect. On the other hand, if the "<=" operator is replaced by the "=" operator in the behaviors, the results depend on the order of the statements. The two behaviors below differ only in the order in which the non-blocking assignments are listed.

```
initial begin
  A = 1;
  B = 0;
  ...
  A = B; // Uses B = 0
  B = A; // Uses A = 1
end
```

Another example

```
module nb1;
  reg a, b, c, d, e, f;

  // blocking assignments
  initial begin
    a = #10 1;
    b = #2 0;
    c = #3 1;
  end

  // non-blocking assignments
  initial begin
    d <= #10 1;
    e <= #2 0;
    f <= #3 1;
  end
endmodule
```

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Mixed with delays (get complicated)

module bit_or8_gate6 (y, a, b);
  input [7:0] a, b;
  output [7:0] y;
  reg [7:0] y;
  always @(a or b) begin
    y = a | b;
  end
endmodule

Two things are wrong. First, the assignment of value to y is determined by
old data (i.e., the values of a and b at five time steps after the activating event). The
second issue is that while the delay control
is blocking the procedural assignment to y,
the event control expression does not
respond to event of a or b. Physical
combinational parts do exhibit this behavior.

Non-overlapping clock modeling

module non_block (sig_a, sig_b, sig_c);
  reg sig_a, sig_b, sig_c;
  initial begin
    sig_a = 0;
    sig_b = 1;
    sig_c = 0;
  end
  always sig_c = #5 ~sig_c;
  always @(posedge sig_c)
  begin
    sig_a <= sig_b;
    sig_b <= sig_a;
  end
endmodule

Figure 7.24: Signal generator formed by non-blocking assignments

\[
\begin{array}{ccccccc}
\text{sig}_c & & & & & & \\
5 & 10 & 15 & 20 & 25 & 30 & t_{\text{wn}} \\
\text{sig}_a & & & & & & \\
5 & 10 & 15 & 20 & 25 & 30 & t_{\text{wn}} \\
\text{sig}_a & & & & & & \\
5 & 10 & 15 & 20 & 25 & 30 & t_{\text{wn}}
\end{array}
\]
Other Behavior Statements

- case
- ?
- repeat
- Flow control
  - for
  - while
  - wait
  - forever
  - disable
- task and function

---

case

Example 7.36: The description of a four-channel multiplexer in Figure 7.27 has a single behavior using a case statement with a case expression consisting of the select word. An event on a datapath or select causes y_out to be computed. This model is correct, but inefficient in simulation—it reacts to all channel activity independently of the channel selection.

```verilog
module mux4_case (a, b, c, d, select, y_out);
input    a, b, c, d;
input [1:0] select;
output   y_out;
reg      y_out;

always @ (a or b or c or d or select)
begin
  case (select)
    0: y_out = a;
    1: y_out = b;
    2: y_out = c;
    3: y_out = d;
    default y_out = 'bx;
  endcase
endmodule
```

---

Other Behavior Statements

- case
- ?
- repeat
- Flow control
  - for
  - while
  - wait
  - forever
  - disable
- task and function

---

case

Example 7.36: The description of a four-channel multiplexer in Figure 7.27 has a single behavior using a case statement with a case expression consisting of the select word. An event on a datapath or select causes y_out to be computed. This model is correct, but inefficient in simulation—it reacts to all channel activity independently of the channel selection.

```verilog
module mux4_case (a, b, c, d, select, y_out);
input    a, b, c, d;
input [1:0] select;
output   y_out;
reg      y_out;

always @ (a or b or c or d or select)
begin
  case (select)
    0: y_out = a;
    1: y_out = b;
    2: y_out = c;
    3: y_out = d;
    default y_out = 'bx;
  endcase
endmodule
```
used in an assignment

Example 7.35  The conditional operator in \texttt{mux\_behavior} implements a selection between two operations on datapaths \texttt{a} and \texttt{b}.

\begin{verbatim}
module mux\_behavior (y\_out, clock, reset, sel, a, b);
    input    clock, reset, sel;
    input    [15:0]  a, b;
    output   [15:0]  y\_out;
    reg      [15:0]  y\_out;

    always @(posedge clock or negedge reset)
        if (reset == 0) y\_out = 0; else y\_out = (sel) ? a + b : a \* b;
endmodule
\end{verbatim}

repeat

- \texttt{Reg\_a=repeat (5) @ (negedge clock) Reg\_b;}
- is equivalent to
  - \texttt{begin}
    - \texttt{temp = Reg\_b;}
    - \texttt{@ (negedge clock);}
    - \texttt{@ (negedge clock);}
    - \texttt{@ (negedge clock);}
    - \texttt{@ (negedge clock);}
    - \texttt{@ (negedge clock);}
    - \texttt{Reg\_a = temp;}
  - \texttt{end}
repeat in an example

```
module repeater;
  reg clock;
  reg reg_a, reg_b;

  initial
    clock = 0;
  initial begin
    #5 reg_a = 1;
    #10 reg_a = 0;
    #5 reg_a = 1;
    #20 reg_a = 0;
  end
  always
    #5 clock = ~ clock;
  initial
    #100 $finish;
  initial begin
    #10 reg_b = repeat (5) @ (posedge clock) reg_a;
  end
endmodule
```

Flow control: for

**Example 7.43** In this example, the for loop is used to assign value to the two fields of bits within a register after it has been initialized to "x". The results of executing the loop are shown in Figure 7.31.

```
reg [15:0] demo_register;
integer K;

for (K = 4; K = K+1)
  begin
    demo_register [K+10] = 0;
    demo_register [K+2] = 1;
  end
```

![Figure 7.31 Register contents after execution of for loop](image)
Flow control: while

Example 7.46 The behavior in count_of_1s counts the number of ‘1’s within a register by counting the number of times that a ‘1’ is observed in the LSB as the word is shifted repeatedly to the right. (Recall that the shift operator shifts a ‘0’ in the position emptied by the shift operation.)

```verilog
begin count_of_1s
    reg [7:0] temp_reg;
    count = 0;
    temp_reg = reg_a; // load a data word
    while (temp_reg)
        begin
            if (temp_reg[0]) count = count + 1;
            temp_reg = temp_reg >> 1;
        end
    end
```

An alternative implementation is shown below:

```verilog
begin count_of_1s
    reg [7:0] temp_reg;
    count = 0;
    temp_reg = reg_a; // load a data word
    while (temp_reg)
        begin
            count = count + temp_reg[0];
            temp_reg = temp_reg >> 1;
        end
    end
```

The **wait** statement

- Suspend (not terminate) statements until something comes true

Example 7.19 When the activity flow of the behavior within example_of_wait reaches the wait statement, the enable is evaluated to determine whether it is TRUE or FALSE. If it is TRUE the activity flow continues. Otherwise, it suspends until enable becomes TRUE.

```verilog
module example_of_wait();

always begin
    wait (enable) register_a = register_b;
    #10 register_c = register_d;
end
endmodule
Flow control: **forever**

**Example 7.48** Clocks and pulsetrains in testbenches are easily described with a *forever* loop. The code below produces the waveforms in Figure 7.37 under simulation. This example also illustrates how the activity of an initial behavior may continue for the duration of the simulation.

```vhdl
parameter half_cycle = 50;

initial
begin: clock_loop
    clock = 0;
    forever
    begin
        #half_cycle clock = 1;
        #half_cycle clock = 0;
    end
end

initial
#350 disable clock_loop;
```

---

Flow control: **disable**

```vhdl
module find_first_one (A_word, trigger, index_value);
input [15:0] A_word;
input trigger;
output [3:0] index_value;
reg [3:0] index_value;

always @(trigger)
begin
    index_value = 0;
    for (index_value = 0; index_value <= 15; index_value = index_value + 1):
        if (A_word [index_value] == 1) disable;
end
endmodule
```
Task (procedure in a module)

Example 7.53 A task that receives a binary word and returns the number of ones in the word is declared in Bit Counter. The environment in which the task is invoked passes the value of reg_a to count_ones_in_word and receives count when the task completes execution.

module Bit_Counter (data_word, bit_count);
input [7:0] data_word;
output [3:0] bit_count;
reg [3:0] bit_count;
always @(data_word);
  count_ones_in_word (data_word, bit_count);

  task count_ones_in_word;
  input [7:0] reg_a;
  output [3:0] count;
  reg [7:0] temp_reg;
  begin
    count = 0;
    temp_reg = reg_a; // load a data word
    while (temp_reg)
      begin
        count = count + temp_reg[0];
        temp_reg = temp_reg >> 1;
      end
  end
endtask
endmodule

function

- Used inside a module
- Implement combinational behavior only
  - No recursion
  - No call to a task
  - No event/delay control

- Output is a reg variable by default
function in a module

module word_aligner (w_in, w_out);
input [7:0] w_in;
output [7:0] w_out;

assign w_out = align (w_in);

function [7:0] align;
input [7:0] word;
begin
    align = word;
    if (align != 0)
        while (align[7] == 0)
            align = align << 1;
end
endfunction
endmodule

FSM Modeling
Style #1

module FSM_style1 (...);
  input ... ;
  output ... ;
  parameter size = ... ;
  reg [size-1; 0] state, next_state;

assign the_outputs = ...
  // a function of state and inputs
assign next_state = ...
  // a function of state and inputs

always @ (negedge reset or posedge clk)
  if (reset == 1'b0) state = start_state; else
    state <= next_state;
endmodule
Style #2

module FSM_style2 (...);
...
assign the_outputs = ...  // a function of state and inputs

always @ (state or the_inputs)
begin
   // decode for next_state with "case" or "if"
end

always @ (negedge reset or posedge clk)
if (reset == 'b0) state = start_state; else
   state <= next_state;
endmodule
Style #3

```verilog
module FSM_style3 (...);

always @ (state or the_inputs)
begin
  // decode for next_state with "case" or "if"
  end

always @ (negedge reset or posedge clk)
if (reset == 1'b0) state = start_state; else
begin
  state <= next_state;
  outputs <= some_value (inputs, next_state);
end
endmodule
```
An example

module Up_Down_Explicit (count, up_down, clock, reset);
output [2:0] count;
input [1:0] up_down;
input clock, reset;
reg [2:0] count, next_count;
always @ (negedge clock or negedge reset)
    if (reset' == 0) count' = 3'b0; else count' = next_count;

always @ (count or up_down) begin
    case (count)
        0: case (up_down)
            0.3: next_count = 0;
                1: next_count = 1;
                2: next_count = 3'b111;
                default next_count = 0; endcase
        1: case (up_down)
            0: next_count = 1;
            1: next_count = 2;
            2: next_count = 0;
            default next_count = 1; endcase
        2: case (up_down)
            0.3: next_count = 2;
                1: next_count = 3;
                2: next_count = 1;
                default next_count = 2; endcase
        3: case (up_down)
            0.3: next_count = 3;
                1: next_count = 4;
                2: next_count = 2;
                default next_count = 3; endcase
    endcase
endmodule

Different styles

module Up_Down_Implicit (count, up_down, clock, reset);
output [2:0] count;
input [1:0] up_down;
input [1:0] clock, reset;
reg [2:0] count;
always @ (negedge clock or negedge reset)
    if (reset' == 0) count' = 3'b0; else
        if (up_down == 2'b00 || up_down == 2'b11) count' = count; else
        if (up_down == 2'b01) count' = count + 1; else
        if (up_down == 2'b10) count' = count - 1;
endmodule

module Up_Down_ Implicit2 (count, up_down, clock, reset);
output [2:0] count;
input [1:0] up_down;
input clock, reset;
reg [2:0] count, next_count;
always @ (negedge clock or negedge reset)
    if (reset' == 0) count = 3'b0; else count = next_count;
always @ (count or up_down) begin
    if (up_down == 2'b00 || up_down == 2'b11) next_count = count; else
    if (up_down == 2'b01) next_count = count + 1; else
    if (up_down == 2'b10) next_count = count - 1; else
        next_count = count;
endmodule
Behavior modeling

• Care only about input/output behavior
• Is to be interpreted by a simulator
• Does not reveal the actual implementation
  – What cells to use
  – Where to put the flip-flops
  – What is the encoding scheme for FSM
• In general, may not be synthesizable
  – Typically, a synthesis tool can’t take a behavior model and synthesize it automatically
  – However, ad hoc solution may be available, i.e. synthesis of certain styles or usage of behavior statement may be possible
• Much more efficient in simulation than an RTL model (good for simulation, not synthesis)
• Equivalence checking between a behavior model and a low-level model such as RTL or gate can be challenging