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)

A behavior view of a 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

```
module test (i, o);
    reg i, o, g1, g2;
    assign g1 = (i == 1) ? 0 : 1;
    assign g2 = (g1 == 0) ? 0 : 1;
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:

```vhdl
module Demo_mod_A ( ... );
    event something_happens: // Declaration of an abstract event
        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

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, `mux_FCA`, 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.

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

  always @ (select)
    case (select)
      0 : assign y_out = a;    // Blocked assignment
      1 : assign y_out = b;    // Non-blocked assignment
      2 : assign y_out = c;    // Blocked assignment
      3 : assign y_out = d;    // Non-blocked assignment
    endcase

endmodule
```

= and <= are different from assign

- `=` is called blocked assignment
- `<=` is called non-blocked assignment
- They do not represent “physical wiring” effect like the `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

Blocked vs. Non-Blocked

- Blocked assignment
  - One assignment get executed after another
  - Use `=`
  - `A = #5 B; C = D;`
  - “C = D” will not be executed after 5 time units
- Non-Blocked assignment
  - Assignments get executed in parallel
  - Use `<=`
  - `A <= #5 B; C <= D;`
  - “C = D” does not wait!
Another example

```
module n0; begin
  a, b, c, d, e, f;
  // blocking assignments
  initial begin
    a := #10 1;
    b := #2 0;
    c := #8 1;
    end
  // non-blocking assignments
  initial begin
    d := #10 1;
    e := #2 0;
    f := #3 1;
    end
  endmodule
```

Mixed with delays (get complicated)

```
module bit_quant(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
```

```
module bit_quant7(y, a, b);
  input [7:0] a, b;
  output [7:0] y;
  reg [7:0] y;
  always @(a or b) begin
    y <= (a << 1) + b;
  end
endmodule
```
Non-overlapping clock modeling

Other Behavior Statements

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

Example 7.38: The description of a four-channel multiplexer in Figure 7.27 has a single behavior using a case statement with a debug expression mentioning of the select word. An event on a datapath or select causes y_out to be computed. This module is corrected using in simulation it reacts to all channel activity independently of the channel selection.

```
module mux_case (a, b, c, d, select, y_out);
input a, b, c, d;
output [1:0] 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 = 1'b0;
  endcase
endmodule
```
used in an assignment

Example 3.36 The conditional operator in `mux_behavior` implements a selection between two operations on variables `a` and `b`.

```verilog
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 == 1) y_out = 0; else y_out = (sel) ? a : b;
  endmodule
```

repeat

- `Reg_a=repeat (5) @(nedge clock)Reg_b`;
- is equivalent to
  - begin
    - temp = Reg_b;
    - @(nedge clock);
    - @(nedge clock);
    - @(nedge clock);
    - @(nedge clock);
    - @(nedge clock);
    - Reg_a = temp;
  - end

repeat in an example

```verilog
module repeat_count;
  reg clock;
  reg reg_a, reg_b;
  initial
    clock = 0;
  initial begin
    reg_a = 0;
    reg_b = 0;
    reg_a = 1;
    reg_b = 1;
  end
  always
    reg_a = reg_a + 1;
  endmodule
```

Figure 3.20 Waveforms for repeated into-assignment delay.

- Figure 3.20 Waveforms for repeated into-assignment delay.
Flow control: for

Example 7.43 In this example, the for loop is used to assign values to the four fields of intr 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 = 0; K <= 3; K++)
begin
  demo_register[K*16] = 0;
  demo_register[K*16+1] = 1;
end
```

Figure 7.31 Register contents after execution of for loop

Flow control: while

Example 7.44 The behavior is count_of to count the number of '1's within a register by counting the number of times that a '1' is observed in the 16-bit register. A count operation sets a '1' in the position occupied by the count operation.

```
begin count_of, reg [15:0], tmp_reg;
  count = 0;
  tmp_reg = reg_a;
  while (tmp_reg)
    begin
      tmp_reg &= (tmp_reg - 1);
      count = count + 1;
    end
end
```

An alternative implementation is shown below.

```
begin count_of, reg [15:0], tmp_reg;
  count = 0;
  tmp_reg = reg_a;
  while (tmp_reg)
    begin
      tmp_reg = (tmp_reg - 1);
      count = count + 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.

```
module example_of_wait();
    always begin
        wait (enable) register_a = register_b;
        if (enable) register_c = register_d;
    end
endmodule
Flow control: **forever**

Example 7.48: Clocks and pulsatins in testbenches are easily described with a *forever* loop. The code below produces the waveforms in Figure 7.57 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 = 1;
    clockCycle = 0;
  end
end
initial
r(300 disable clock_loop);
```

Flow control: **disable**

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

always @(trigger)
begin
  index_value = 0;
  for (index_value = 0; index_value <= 16; index_value = index_value + 1):
    if (A_word[index_value] == 1) disable;
end
endmodule
```

Task (procedure in a module)

Example 7.49: A task that returns a delay word and updates the number of cycles the task is executed in 48 (Cycle). The number of cycles in which the task is executed is stored in the value of reg 4_count, which also enables control when the task completes execution.

```vhdl
module task_cycle (data_word, delay_word);
input data_word;
input delay_word;
output reg [3:0] count;

always @ (data_word or delay_word)
begin
  count = 0;
  if (data_word) begin
    delay_word_req = 1;
    end
  end
end
```
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

```verilog
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 == 'h0) 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 == 'h0) state = start_state, else
    state <= next_state;
endmodule

Style #3

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 == 'h0) state = start_state, else
    begin
      state <= next_state;
      outputs <= some_value (inputs, next_state);
    end
endmodule
An example

Different styles

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