Behavioral Descriptions: initial and **always**

General rules distinguishing nets and registers:

- A net or register variable can be **referenced** anywhere in a module.
- A net variable can NOT be assigned to within a behavior; a register variable can NOT be assigned to outside a behavior.
- A net variable can be an **input**, **output** or **inout** port, a register variable can ONLY be used in an **output** port.

initial and **always**

Enable the designer to describe design at a high level of abstraction, i.e., as a sequence of operations, in a recipe-type fashion.

Contain *procedural statements* that execute sequentially, controlling activity flow and making assignments to register variables.

Unlike *continuous assignment*, a procedural statement is NOT sensitive to the activity in the circuit.

Procedural statements affect the register variables ONLY when control is passed to them.

Behavioral Descriptions: **initial** and **always**

initial and **always**

initial declares a *one-shot* sequential activity flow activated at $t_{sim} = 0$.

Typically used by designers to initialize a simulation.

always declares a *cyclic behavior* also activated at $t_{sim} = 0$.

Once the last stmt executes, control is passed again to the first stmt.

initial and **always** blocks execute concurrently with other structural and *continuous assignment* stmts in the module.

There are three types of assignments within behaviors:

- *Procedural assignment* (=),
- *Non-blocking assignment* (<=)
- *Procedural continuous assignment* (**assign**)

Procedural assignments using '=' operator execute sequentially and are called *blocking assignments*.

Behavioral Descriptions: Assignment

Verilog also provides a *non-blocking* procedural assignment construct, <=, which does NOT block the execution of stmts that follow.

Non-blocking assignments execute in two steps

- First the RHS is evaluated and the simulator schedules the assignment at a time determined by an optional intra-assignment delay or event control.

- At the end of the designated future time step, the actual assignment is carried out.

Non-blocking stmts evaluate *concurrently* and their order is irrelavant.

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

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

Behavioral Descriptions: Assignment

In contrast

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

Non-blocking assignments are preferred in synthesis.

Warning: Synthesis tools do NOT support a mixture of *blocking* and *non-blocking* assignments within the same behavior.

A ***procedural*** *continuous assignment* (PCA) creates a *dynamic binding* to a register variable when the statement executes.

It uses "=" as in procedural assignment with the keyword **assign**.

WARNING: The Xilinx synthesis engine does not accept this Verilog construct

Behavioral Descriptions: Event Control Operator

Synchronizes the execution of a procedural statement(s) to a change in value of a variable in the sensitivity list.

We've seen these for sequential circuits synchronized to clock edges using **posedge** and **negedge**, e.g., a D-flipflop, with synchronous set/reset.

```
module sync_df(data, clk, q, q_bar, set, reset)
   input data, clk, set, reset;
   output q, q_bar;
   reg q;

   assign q_bar = ~q;

   always @(posedge clk)
      begin
         if (reset == 0 ) q = 0;
         else if (set == 0) q = 1;
         else q = data;
      end
endmodule
```

WARNING: Do NOT assign within the behavior to the variable in the sensitivity list.

Behavioral Descriptions: Event Control Operator

Asynchronous set/reset can be introduced easily using **or**.

```
module asynch_df(data, clk, q, q_bar, set, reset)
  input data, clk, set, reset;
  output q, q_bar;
  reg q;
  assign q_bar = ~q;
  always @(negedge set or negedge reset or posedge clk)
    begin
      if (reset == 0 ) q = 0;
      else if (set == 0) q = 1;
      else q = data;
    end
endmodule
```

For latches.

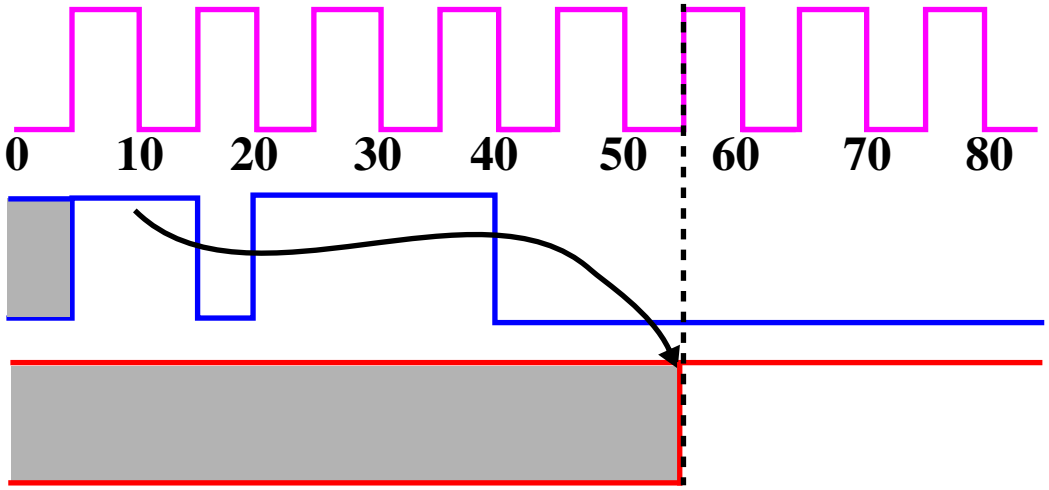
```
module t_latch(q_out, enable, data)
  input enable, data;
  output q_out;
  reg q_out;
  always @(enable or data)
    begin
      if (enable) q_out = data;
    end
endmodule
```

Behavioral Descriptions: Event Control Operator

The *event_expression* can be repeated a specified number of times:

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

  initial
    clk = 0;

  initial begin
    #5 reg_a = 1;
    #10 reg_a = 0;
    #5 reg_a = 1;
    #20 reg_a = 0;
  end

  always
    #5 clk ~= clk;

  initial
    #100 $finish;
  initial begin
    #10 reg_b = repeat (5) @(posedge clk) reg_a;
  end
endmodule
```

Note that t_sim = 55, *reg_b* get the value *reg_a* had at t_sim = 10, not the value at t_sim = 55.




```
// temp = reg_b;
//@ (posedge clk); @ (posedge clk); @ (posedge clk);
//@ (posedge clk); @ (posedge clk);
// reg_a = temp;
```

Behavioral Descriptions: Activity Flow Control

These types of statements modify the activity fl ow within a behavior

- **?...:, case, if** (conditional)
- **repeat, for, while, forever** (loop)
- **wait** (suspend)
- **fork ... join** (branch)
- **disable** (terminate)

Conditional Operator (? ... :)

Discussed previously when used with continuous assignment stmts -- can also be used with *procedural stmts*.

```
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
```

Behavioral Descriptions: Activity Flow Control

The *case expression* is evaluated in Verilog's 4-value logic system.

The **case** stmt requires an exact bitwise match.

The other two variants of the **case** treat *don't care* situations.

casex ignores values in those bit positions of the case *expression* or *case_item* that have the value "x" or "z" -- matches anything, 0, 1, *x* or *z*.

casez ignores any bit position of the case *expression* or *case_item* that have value "z". It also uses "**?**" as an explicit *don't care*.

```
always @(decode_pulse)
  casez (instruction_word)
    16'b0000_????_????_????:; // Null stmt for no-op.
```

For simulation, the **default** case is optional.

For synthesis, be sure to cover all possible combinations of the *expression* in the *case_item* list (or use **default**) to avoid unwanted latches.

Behavioral Descriptions: Activity Flow Control

Verilog has 4 loops mechanisms, **repeat**, **for**, **while** and **forever**

for and **while** work as they do for std programming languages.

Repeat executes a stmt or block a specified number of times.

When reached, *expression* is evaluated to determine the number of iterations.

```
...
word_address = 0;
repeat (memory_size)
   begin
   memory[word_address] = 0;
   word_address = word_address + 1;
   end
...
```

The **forever** loop is unconditional and is terminated via a **disable** stmt.

disable can also be used to terminate a **for**, **repeat** or **while** loop.

Behavioral Descriptions: Activity Flow Control

Clocks and pulse-trains in testbenches are easily implemented using **forever** loops:

```
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;
```

NOTE: do NOT confuse **always** and **forever** stmts

The **always** construct declares a concurrent behavior, that can NOT be nested and becomes active at the beginning of simulation.

The **forever** loop is a computational activity, that can be nested and does not execute until it is reached within an activity fl ow.