**Behavioral Descriptions**

We have already discussed 3 types of abstract (non-structural) behaviors in Verilog.

- continuous assignments

  Implement *implicit* combinational logic through static bindings of expressions and target nets.

- **initial** and **always**

  Declare a description of functionality in computational activity flows, modeling the relationship between the I/O ports of the module.

  **initial** declares a *one-shot* sequential activity flow while **always** declares a cyclic sequential activity flow.

```
module demo_1 (sig_a)
  output sig_a;
  reg sig_a;
  initial
     sig_a = 0;
endmodule
```

Here, *sig_a* is assigned 0 at $t_{sim} = 0$, which it retains indefinitely.

## Behavioral Descriptions

The statements implementing a declared behavior (within **initial** and **always**) will be referred to as *procedural statements*.

As indicated, procedural assignment can be made only to register variables, i.e., **reg**, **integer**, **real**, **realtime** and **time**.

initial_construct ::= **initial** statement
always_construct ::= **always** statement
statement ::=
blocking_assignment; |
non_blocking_assignment; |
  procedural_assignment; |
  procedural_timing_control_stmt |
  conditional_statement |
  case_statement |
  loop_statement |
  wait_statement |
  disable_statement |
  event_trigger |
  seq_block |
  par_block |
  task_enable |
  system_task_enable;

**Behavioral Descriptions**

A simple clock generator:

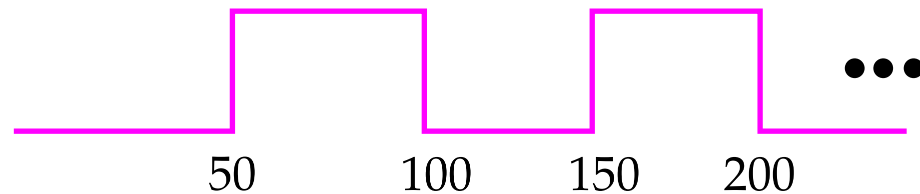
```
module clock_gen1 (clock)
    parameter Half_cycle = 50;
    parameter Max_time = 1000;
    output clock;
    reg clock;

    initial
      clock = 0;

    always
      begin
        #Half_cycle clock = ~clock;
      end

    initial
      #Max_time $finish;

endmodule
```




The #*Half_cycle* introduces 50 units of delay.

The simulation finishes after 10 clock cycles.

**Behavioral Descriptions**

The statements with an **initial** and **always** behavior support sequential computations that manipulate the values of data objects in memory.

However, Verilog behaviors also implicitly govern the activity flow of a simulation by influencing whether and when events are scheduled.

The procedural constructs can be organized into several categories:

- Assignment
    - Conditional (? ... :)
    - Procedural Assignment (=)
    - Procedural-continuous (**assign** ... **deassign**, **force** ... **release**)
    - Non-blocking assignment (<=)
- Code Management
    - Function calls
    - Task calls
    - Prog. lang interface (PLI)

**Behavioral Descriptions**

- Timing and Synchronization
  - Assignment delay control
  - Intra-assignment delay
  - Event control
  - Wait
  - Named Events
  - Pin-pin delay
- Flow Control
  - Conditional (**if**)
  - Case
  - Branching
  - Loops
  - Parallel activity (**fork** ... **join**)

**Procedural Assignment**

A statement that assigns value to a register variable is called *procedural assignment.*

**Procedural Assignment**

There are three types of *procedural assignments*, as indicated in the previous listing:

- *Procedural assignment* (=),
- *Procedural continuous assignment* (**assign** or **force** ... **release**)
- *Non-blocking assignment* (<=)

Bear in mind that assignment to register variables obey different rules than assignments to nets.

When the input to a primitive or continuous assignment statement changes, the output is evaluated and scheduled to change in the future.

For procedural assignments, assignment occurs *only if* control is passed to it and the statement executes. Assignment is immediate.

Therefore, the mere appearance of a statement in a process *does not guarantee* that the target register variable will ever be assigned to.

See text p. 167 for summary of rules for nets and registers.

**Procedural Continuous Assignment**

In order to emulate level sensitive behavior within the hardware, e.g., a latch, Verilog defines a *procedural continuous assignment* (PCA).

*Continuous assignment* establishes a *static binding* of the RHS expression and LHS net variable, and can only be defined for nets, not registers.

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

There are two forms, one can be used only with register variables, while the 2nd can be used on registers or nets.

- **assign** ... **deassign** is similar to a continuous assignment to a net, but the binding here can be removed dynamically.

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

  It is used to model the *level-sensitive* behavior of combinational logic, transparent latches and asynchronous control of sequential parts.

**Procedural Continuous Assignment**

Bear in mind that **some** synthesis tools do NOT support this construct.

```
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 = c; else y_out = 1'bx;
endmodule
```

Here, the procedural assignment statement binds an assignment expression (event scheduling rule) to a target register variable.

Similar to a continuous assignment binding an expression to a net.

The assignment *takes effect* when executed and *stays in* effect until another procedural assignment is made or a **deassign** statement is made to the **reg**.

**Procedural Continuous Assignment**

While a PCA is in effect, it **overrides all** procedural assignments to the target variable.

It models behavior in which an *asynchronous control signal*, the set-reset signal of a FF, must override a synchronous signals, such as the clock.

It also effectively models a latch, which responds to input signal changes when enabled (clk is high for example) but ignore them when disabled.

```
module Flop_PCA (preset, clear, q, qbar, clock, data)
  input preset, clear, clock, data;
  output q, qbar;
  reg q;
  assign qbar = ~q;
  always @(negedge clock)
    q = data;                          // overridden if clear/preset unset
  always @(clear or preset)
    begin
      if (!clear) assign q = 0;
      else if (!preset) assign  q = 1;
      else deassign q;
  end...
```

**Procedural Continuous Assignment**

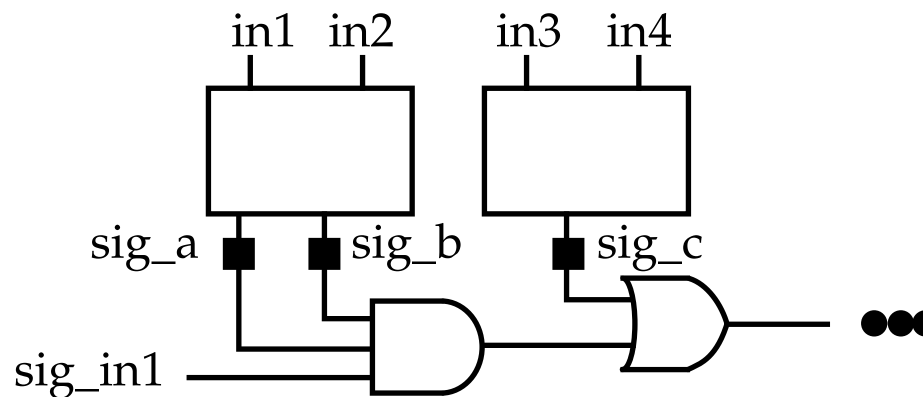
- **force** ... **release** is similar to **assign** ... **deassign** but applies to registers or nets.

  When **force** is applied to a net, the expression forced overrides all other drivers until a **release** is executed.

  It can override a primitive driver, a continuous assignment, a procedural assignment and an **assign** .. **deassign** PCA to a register.

  Used in test benches mainly -- don't expect the synthesis tool to support this one.

```
force sig_a = 1;
force sig_b = 1;
force sig_c = 0;
sig_in1 = 0;
#5 sig_in1 = 1;
#5 sig_in1 = 0;
//  other code
release sig_a;
release sig_b;
release sig_c;
```




A test to sensitive a path

**Procedural Timing Controls and Synchronization**

Verilog provides 4 mechanisms to provide explicit control over time of execution of a procedural statement.

- Delay control
- Event control
- Named events
- **wait** construct

We already saw an example of *delay control* in the clock generator example.

*Event control*, *named events* and **wait** are event-sensitive mechanisms, that synchronize activity within and between behaviors.

When a behavior executes, it continues until it encounters a *delay control* operator (#), an *event control* operator (@) or **wait** construct.

When it suspends, other processes can execute.

**Delay Control Operator**

Suspends the activity fl ow within a behavior by postponing the execution of a procedural statement.

delay_control ::= # delay_value | #(expression)

The actual delay can be expressed as a number, an identifier (with implicit numeric value) or an expression.

If expression evaluates to #0 (no delay), the stmt executes at the end of the current simulation cycle.

If # *delay_value* proceeds an assignment statement, the assignment **is not performed** until *after* the specified time elapses.

Also, all statements following it are suspended.

```
initial   //Note: at time 0, IN3, IN4 and IN5 are initialized to value 'x'
  begin
    #0 IN1 = 0; IN2 = 1;        // Executes at t_sim = 0
    #100 IN3 = 1;               // Executes at t_sim = 100
    #100 IN4 = 1, IN5 = 1;      //Executes at t_sim = 200
    #400 IN5 = 0;               // Executes at t_sim = 600
  end
```

**Event Control Operator**

Synchronizes the execution of a procedural statement(s) to a change in the value of either an identifier or an expression.

The "@" symbol implements this control.

event_control ::= @ *event_identifier* stmt_or_null |
@ (*event_expression*) stmt_or_null

stmt_or_null ::= statement | ;

Here, the assignment is carried out when *Signal_1* changes:

```
begin
  ...
  @ Signal_1 register_A = register_B;
  ...
end
```

Activity control suspends at the @ symbol and the simulator **monitors** *Signal_1* for an event.

NOTE: Activity control MUST be suspended at the @ symbol in order for the simulator to monitor changes in it.

**Event Control Operator**

If *event_A* has occurred but *event_B* has not, control will be suspended at *event_B* and further events on *event_A* are ignored.

```
...
  @ (event_A) begin
    ...
      @ (event_B) begin
```

For synchronous sequential circuits that synchronize on clock edges, Verilog provides edge qualifiers **posedge** (0 --> 1, 0 --> *x*, *x* --> 1) and **negedge**.

```
always @(posedge clock) #10 q_register = data_path;
```

*q_register* gets the value of *data_path* 10 times units after the positive edge of *clock*.

The *event_expression* and *event_identifier* used in event control must reference a net or register (not a parameter, which is a constant).

Also, the register referred to in the control expression can **not** be assigned to within the behavior that it synchronizes.

**Event Control Operator**

A better model of a D-flipflop, with synchronous set/reset.

```
module df_behav (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
```

Verilog also allows "event OR-ing", the use of disjunction to form complex *event_expressions*.

```
always @(Signal_1 or Signal_2) register_A = register_B;
```

**Event Control Operator**

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

```
module asynch_df_behav (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
```

**Named Events**

Provides a high-level mechanism of communication and synchronization within and between modules in Verilog

A *named event* or *abstract event* provides interprocess communication without the details of the physical implementation (nothing is passed in portlist).

```
module Demo_mod_A(...)
...
   event event_a;   // variable of type event is declared in one module.
   always
      begin
         ...
         -> event_a   // Event trigger operator -- triggers an abstract event.
      end
endmodule

module Demo_mod_B(...)
   always @(Top_Module.Demo_mod_A.event_a) // Event monitor
      begin
      ...             // do something when event is triggered.
      end
endmodule
```

**Wait Construct**

Models *level-sensitive* behavior by suspending (not terminating) activity fl ow until an *expression* is TRUE.

If true when evaluated, no suspension occurs.

If false, the simulator suspends the activity thread and sets up a monitor.

wait_statement ::= **wait** (expression) stmt_or_null

stmt_or_null ::= statement | ;

For example:

```
module wait_example(...)
  ...
  always
    begin
      ...
      wait (enable) register_a = register_b;
      #10 register_c = register_d;
    end
endmodule
```

**Intra-Assignment Delay -- Blocked Assignments**

When a timing control operator (# or @) precedes a procedural stmt, the delay is referred to as a *blocking* delay.

All stmts following a *blocked* stmt are also suspended.

Verilog supports another form of delay, *intra-assignment delay*, where the timing control is placed on the righthand side (in RHS) in an assignment stmt.

Here, the RHS is evaluated **immediately** but the assignment doesn't take place until the future.

```
...
A = #5 B;                // A = @(event_expression) var; can also be used.
C = D;
...
```

Here, *B* is sampled but *A* is not assigned the value of *B* for 5 more time units.

This separates referencing and evaluation from the actual assignment.

The stmt, *C* = *D*; does not execute until the assignment is made 5 time units in the future.

**Non-Blocking Assignments**

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

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

non_blocking_assignment ::= reg_lvalue <= [delay_or_event_control] expr;

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.

Note, during the actual time step, the non-blocking assignments are usually performed last (if other assignments are also being made) to prevent races.

## Non-Blocking Assignments

```
initial
   begin
   A = 1;
   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
```

The non-blocking stmts evaluate *concurrently and independently* of their order, once evaluated, the assignments are made.

The results of either code sequence are identical (swaps the values).

In contrast

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

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

**Non-Blocking Assignments**

Non-blocking assignments are useful in modeling *concurrent transfers of data* in sequential circuits.

Synthesis tools recommend non-blocking assignments for this purpose.

Here, the *order* of the execution of these stmts is *not* important, and it models the concurrent assignment to a set of register variables in the same time step.

Normally, all RHS are evaluated before any assignments are made and therefore order doesn’t matter.

However, if 2 non-blocking assignments assign to the same register variable, Verilog uses the last assignment in the ordered list.

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

Even though Verilog does.