**Intra-Assignment Delay: Non-Blocking Assignment**

Here, *accum* is sampled on the rising edge of *clock* and the assignment is scheduled to occur when *a_bus* changes.

```
initial begin
   @(posedge clock)
      G <= @(a_bus) accum;
      C <= D;
end
```

*D* is also immediately sampled, and assigned to *C*.

This differs from the blocking version, which would prevent *C <= D;* from executing until the previous stmt completed its assignment.

What happens if 2 clock edges arrive before *a_bus* changes?

```
always begin
   @(posedge clock)
      G <= @(a_bus) accum;
end
```

The simulator issues a warning (queueing of values is not supported) and *G* gets the last evaluated value of *accum*.

**Intra-Assignment Delay: Non-Blocking Assignment**

How does behavior differ between blocked and non-blocked stmts which contain intra-assignment delay?

```
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
```

| t | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| 0 | x | x | x | x | x | x |
| 2 | x | x | x | x | 0 | x |
| 3 | x | x | x | x | 0 | 1 |
| 10 | 1 | x | x | 1 | 0 | 1 |
| 12 | 1 | 0 | x | 1 | 0 | 1 |
| 15 | 1 | 0 | 1 | 1 | 0 | 1 |

**Intra-Assignment Delay: Non-Blocking Assignment**

Combinational logic can be modeled by both *one-shot* and *cyclic* forms of Verilog behavior.

This one shot example is similar to the *continuous assignment* example given in Chapter 2.

```
module bit_or8_gate3(y, a, b)
   input [7:0] a, b;
   output [7:0] y;
   reg [7:0] y;
   initial begin   // Alternative in Ch. 2 does not include initial begin...
      assign y = a | b;
   end
endmodule
```

Here, the behavior is activated at t_sim = 0 and stays in effect after the behavior expires.

Although it is valid, it is not the preferred style and will **not be accepted** by a synthesis tool.

**Intra-Assignment Delay: Non-Blocking Assignment**

A second acceptable alternative uses procedural assignment.

```
module bit_or8_gate4(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
```

But not as simple as the Ch. 2 version.

Be careful when using behavioral models designed to model delay.

```
module bit_or8_gate4(y, a, b)
   input [7:0] a, b;
   output [7:0] y;
   reg [7:0] y;

   always @(a or b) begin
      #5 y = a | b;
   end
endmodule
```

**Intra-Assignment Delay: Non-Blocking Assignment**

There are two problems here.

- *y* gets old data, i.e., the values of *a* and *b* 5 time units after the activating event.
- While the delay control is blocking, i.e., waiting to assign to *y*, the event control expression cannot respond to events on *a* and *b*.

Does this model actual hardware?

Alternatively, *intra-assignment delay* can be used to get the proper values of *a* and *b*, i.e., the values at the moment they change, but the 2nd problem remains.

```
module bit_or8_gate4(y, a, b)
   input [7:0] a, b;
   output [7:0] y;
   reg [7:0] y;

   always @(a or b) begin
      y = #5 a | b;
   end
endmodule
```

**Intra-Assignment Delay: Non-Blocking Assignment**

The solution is to use a *non-blocking* assignment with *intra-assignment* delay.

```
module bit_or8_gate4(y, a, b)
   input [7:0] a, b;
   output [7:0] y;
   reg [7:0] y;

   always @(a or b) begin
      y <= #5 a | b;
   end
endmodule
```

**Simulation of Simultaneous Procedural Assignments**

The simulator needs rules to handle multiple behaviors assigning value to the same register in the same time step.

- (1) Evaluate the expressions on the RHS of all assignments to **reg** variables at that time step.
- (2) Execute the blocking assignments to registers.
- (3) Execute non-blocking assignments that can execute in the current time step (no intra-assignment timing).
- (4) Execute past procedural assignments with expired wait times.

**Simulation of Simultaneous Procedural Assignments**

- (5) Advance the simulator time ($t_{sim}$).

Verilog defines the rules using a *stratified event queue*, where the queue of pending simulation events are organized into 5 different regions.

The rules for execution order are fairly complex (see text) and some are implementation dependent.

**Repeated Intra-Assignment Delay**

The *event_expression* in intra-assignment delay can be repeated a specified number of times.

reg_a = **repeat** (5) @ (**negedge** clock) reg_b;

The assignment to *reg_a* will be made after 5 falling edges of the clock.

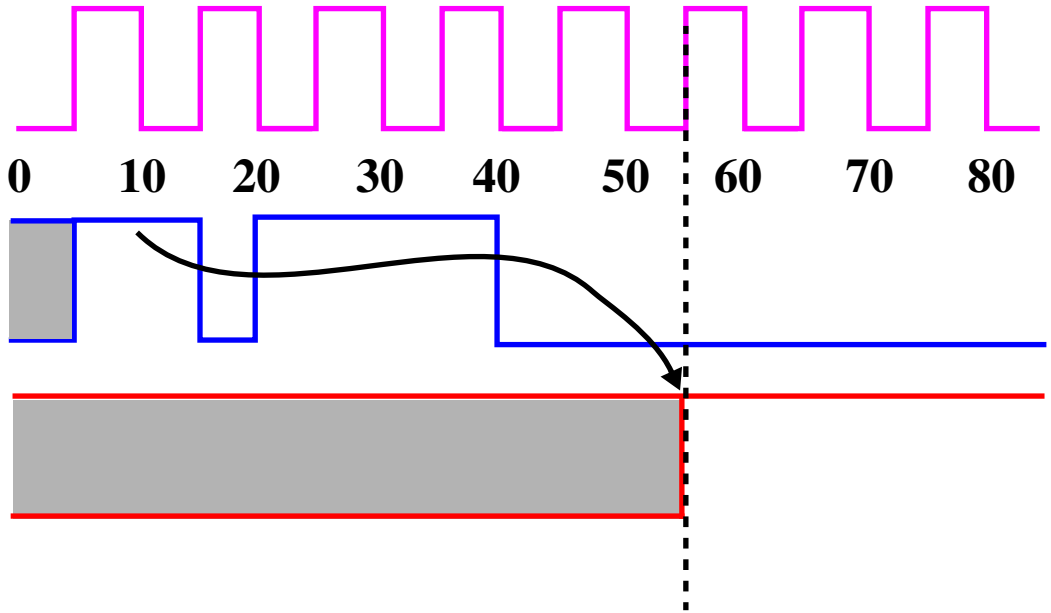
```
begin                          // The above is equivalent to.
   temp = reg_b;
   @ (negedge clock); @ (negedge clock); @ (negedge clock);
   @ (negedge clock); @ (negedge clock);
   reg_a = temp;
end
```

## Repeated Intra-Assignment Delay

Another 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
```

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



**Other Examples**

Using non-blocking assignments with intra-assignment delay to create a schedule of assignments to a target register variable.

```
module multiple_no_block_1;
  reg wave;
  reg [2:0] i;

  initial begin
    for (i = 0; i <= 5; i = i+1)
      wave <= #(i*10) i[0];
    end
endmodule
```

All RHS are sampled at the same time, but the value depends on *i* (the for loop executes in 0 time).

```
module multiple_no_block_2;
  reg wave1, wave2;

  initial begin
    #5 wave1 = 0;
       wave2 = 0;
       wave1 <= #5 1;
       wave2 <= #10 1;
       wave2 <= #20 0;
   #10 wave1 = 1;
       wave1 <= #5 0;
    end
endmodule
```

Draw the waveforms associated with these modules.

**Other Examples**

```
module non_block(sig_a, sig_b, sig_c);
   reg sig_a, sig_b, sig_c;
   initial
      begin                          // Non-overlapping wfms generated
         sig_a = 0;                  // (sig_a and sig_b) from a clock signal
         sig_b = 1;                  // sig_c
         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
```

You should be able to draw the wfms from these descriptions.

**Constructs for Activity Flow Control**

These types of statements modify the activity flow 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
```

**Constructs for Activity Flow Control: Case stmt**

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

case_stmt ::= **case** (*expression*) case_item {case_item} **endcase** |
**casex** (*expression*) case_item {case_item} **endcase** |
**casez** (*expression*) case_item {case_item} **endcase**

case_item ::= expression {, expression}: stmt_or_null |
**default** [:] stmt_or_null

Correct implementation of a MUX but inefficient because it reacts to all activities independent of the selection.

```
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 = 1'bx;
endcase endmodule
```

**Constructs for Activity Flow Control: Case stmt**

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

The *case_item* expression are evaluated in the order listed, and if a match is found, the other cases are not examined or executed.

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

**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.

**Constructs for Activity Flow Control: *if else* stmt**

Alter the normal sequential activity flow within a behavior.

if_stmt ::= **if** (expression) stmt_or_null [**else** stmt_or_null]
stmt_or_null ::= stmt | ;

As always, *stmt_or_null* can be a single or block stmt. Null stmts must terminate with a semicolon.

```
(a) if (A < B) some_register = some_value + 1;
(b) if (C < D);                                    // null stmt
(c) if (k == 1)
        begin : A_Block
           sum_out = sum_reg(4);
           c_out = c_reg(2);
        end
```

The value of the Boolean expression evaluated in the **if** stmt is treated as false if it has numerical value of *0*, or the values *x* or *z*

The meaning of the **else** clause conforms to the notion you've learned for other programming languages.

**Constructs for Activity Flow Control: Loops**

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

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

repeat_loop ::= **repeat** (expression) stmt

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

If *expression* evaluates to *x* or *z*, the result is treated as 0 and no loop iterations occur.

Otherwise, the loop executes unless terminated by a **disable** stmt.

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

**Constructs for Activity Flow Control: Loops**

The **for** loop semantics are identical to those in other prog. languages.

for_loop ::= **for** ( reg_assignment; expression; reg_assignment ) stmt

*register* variable must be an **integer** or **reg**

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

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

reg [3:0] K;                                   // Beware, loops forever...
for ( K = 0; K <= 15; K = K + 1) ...
```

See carry look-ahead example in text.

**Constructs for Activity Flow Control: Loops**

The **while** loop is also semantically identical to its definition in C.

while_loop ::= **while** (Bool_expr) stmt

Loop while *Bool_expr* remains true.

```
begin: count_of_1s
reg [7:0] temp_reg;

count = 0;
temp_reg = reg_a;
while (temp_reg)
   begin
      if (temp_reg[0]) count = count + 1;
      temp_reg = temp_reg >> 1;
   end
end
```

Caution -- the following usage will choke simulator:

```
module Asking_for_trouble(some_external_input);
input some_external_input;
   always begin
      while (some_external_input); //Wait for external variable
   end
endmodule
```

**Constructs for Activity Flow Control: Loops**

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

forever_loop ::= **forever** stmt

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

```
parameter helf_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;
```

**Constructs for Activity Flow Control: Loops**

Do not confuse **always** and **forever**.

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 flow.

The **disable** stmt is used to prematurely terminate a named block or task.

When executed, activity flow is transferred to the stmt immediately following the named block.

```
module find_first_one(A_word, trigger, ind_val);
  input [15:0] A_word;
  input trigger;
  output [3:0] ind_val;
  reg [3:0] ind_val;
  always @ trigger
    begin
      ind_val = 0;
      for (ind_val = 0; ind_val <= 15; ind_val = ind_val + 1)
        if (A_word[ind_val] == 1) disable;
    end
endmodule
```

**Constructs for Activity Flow Control: *fork...join***

The **fork...join** construct is NOT supported by synthesis tools but is useful to generate wfms in testbenches.

It creates parallel threads of activity, each executing concurrently with the others.

```
...
fork      //t_sim = 0
   #50 sig_wave = 'b1;      // Order of stmts here is not important
   #100 sig_wave = 'b0;
   #150 sig_wave = 'b1;
   #300 sig_wave = 'b0;     // Executes at t_sim = 300
join                        // Resynchronize all parallel threads here
```