14:332:231 DIGITAL LOGIC DESIGN

Ivan Marsic, Rutgers University

Electrical & Computer Engineering

Fall 2013

Lecture #24: Verilog Time Dimension and Test Benches

Verilog Functions and Tasks

[behavioral style]

- Verilog **function** accepts several inputs and returns a single result

    `function` *result-type function-name* ;

    input declarations
    variable declarations
    parameter declarations

    procedural-statement

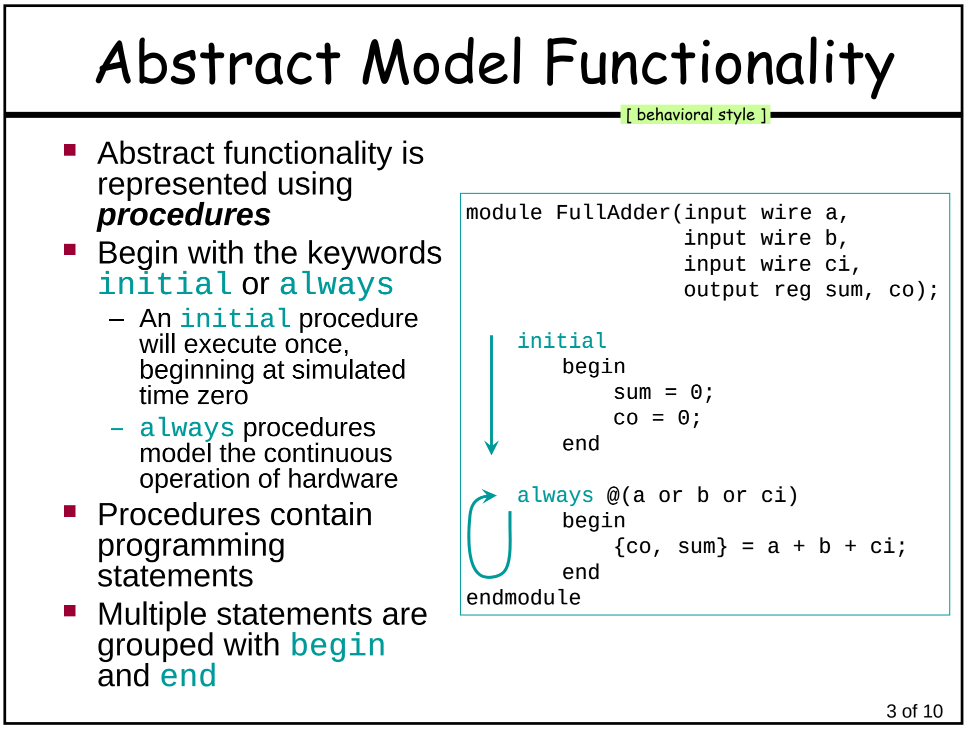
    `endfunction`

```
module VrSillierXOR ( ... );
    port-declarations

    function Inhibit ;
       input In, invIn;
          Inhibit = In & ~invIn;
       endfunction

    always @ (in1 or in2)
       begin
          inh1 = Inhibit (in1, in2);
          ...
       end
```

- Verilog **task** is similar to a function, except it does not return a result
- Built-in system tasks and functions:
    - `$display` = prints formatted signal values to “standard output” (similar to C `printf` function)
    - `$write` = similar to `$display`, but no newline char at end
    - `$monitor` = similar to `$display`, but remain active continuously and prints the listed signals whenever any one changes
    - `$time` = returns current *simulated* time



Abstract Model Functionality

[behavioral style]

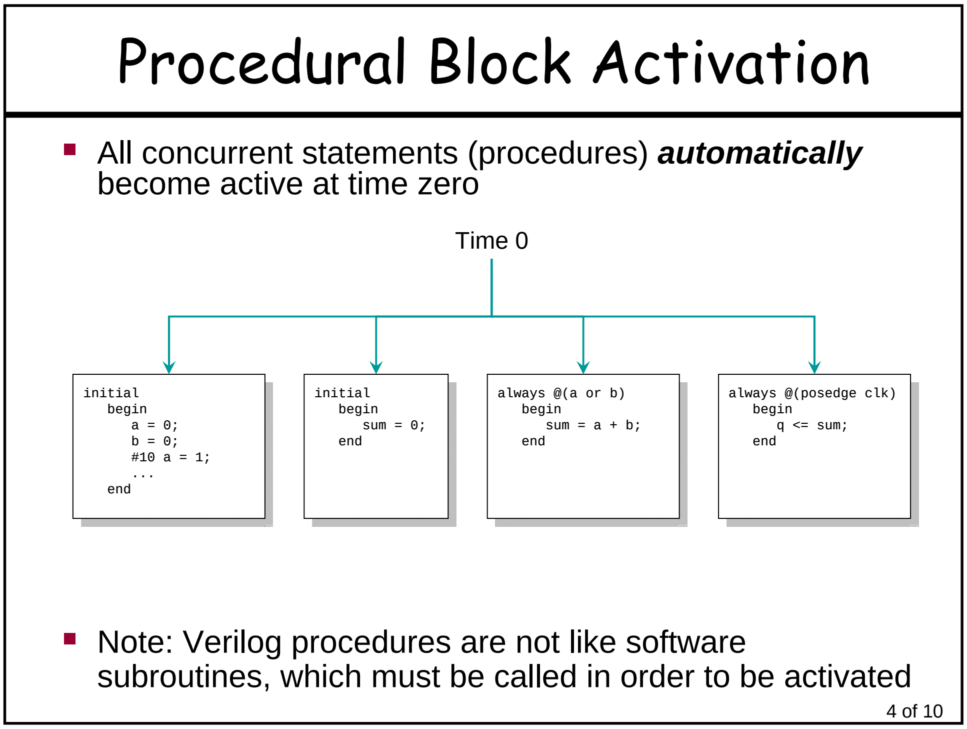
- Abstract functionality is represented using ***procedures***
- Begin with the keywords `initial` or `always`
  - An `initial` procedure will execute once, beginning at simulated time zero
  - `always` procedures model the continuous operation of hardware
- Procedures contain programming statements
- Multiple statements are grouped with `begin` and `end`

```
module FullAdder(input wire a,
                 input wire b,
                 input wire ci,
                 output reg sum, co);

   initial
      begin
         sum = 0;
         co = 0;
      end

   always @(a or b or ci)
      begin
         {co, sum} = a + b + ci;
      end
endmodule
```

3 of 10



Procedural Block Activation

- All concurrent statements (procedures) ***automatically*** become active at time zero

Time 0

```
initial
   begin
      a = 0;
      b = 0;
      #10 a = 1;
      ...
   end
```

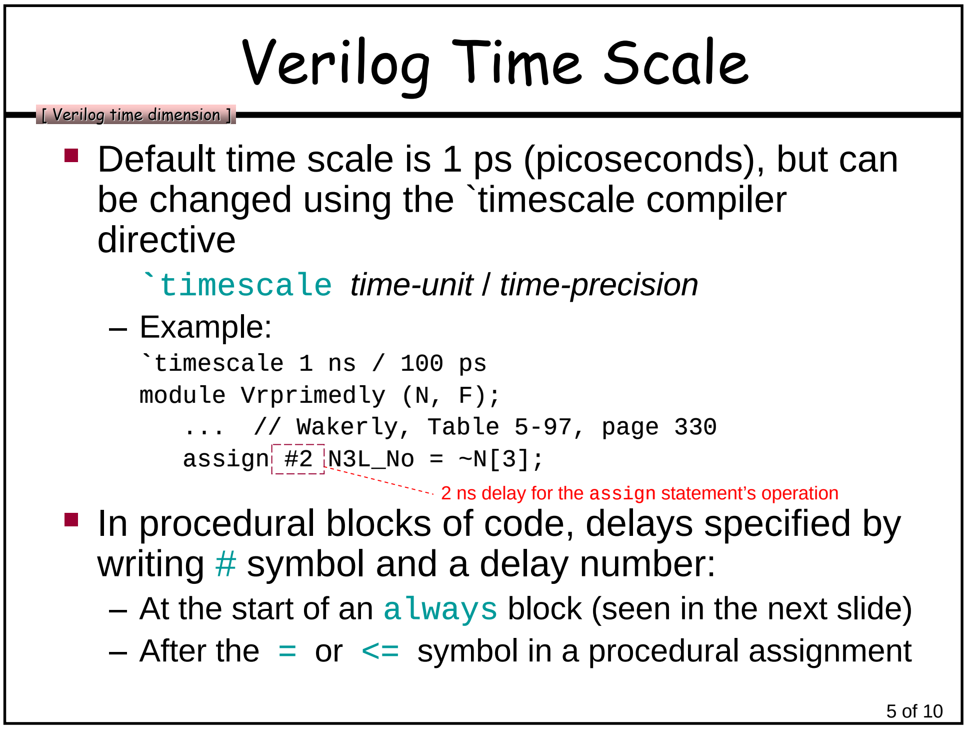
```
initial
   begin
      sum = 0;
   end
```

```
always @(a or b)
   begin
      sum = a + b;
   end
```

```
always @(posedge clk)
   begin
      q <= sum;
   end
```

- Note: Verilog procedures are not like software subroutines, which must be called in order to be activated

4 of 10



Verilog Time Scale

[Verilog time dimension]

- Default time scale is 1 ps (picoseconds), but can be changed using the `timescale compiler directive

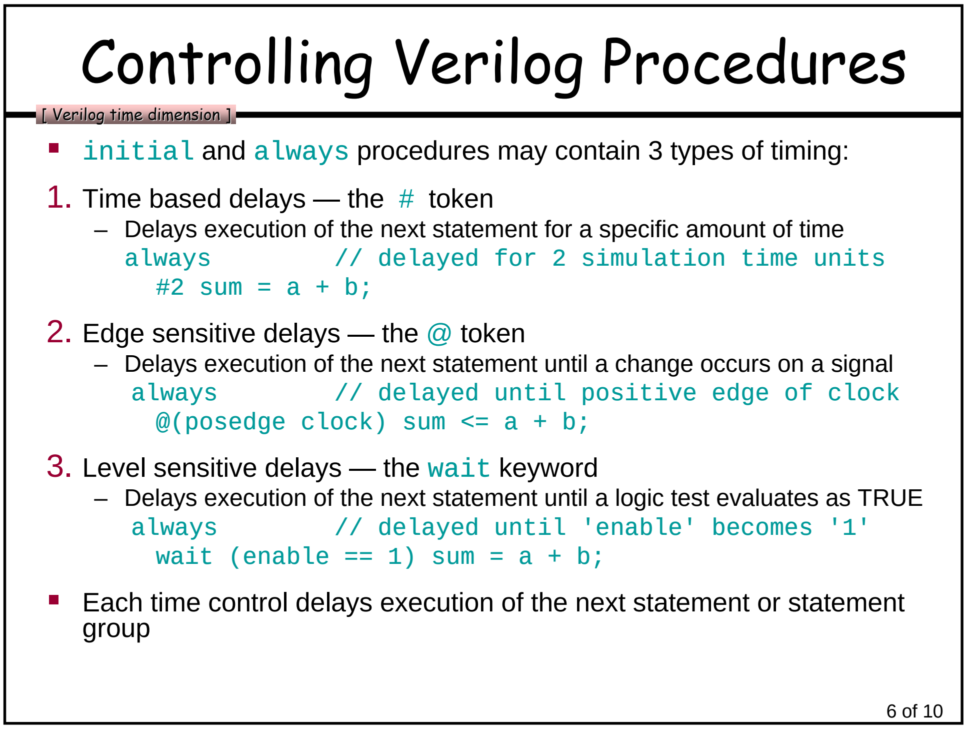
  `` `timescale `` *time-unit* / *time-precision*

  – Example:

```
`timescale 1 ns / 100 ps
module Vrprimedly (N, F);
   ...  // Wakerly, Table 5-97, page 330
   assign #2 N3L_No = ~N[3];
```

2 ns delay for the `assign` statement's operation

- In procedural blocks of code, delays specified by writing `#` symbol and a delay number:
  - At the start of an `always` block (seen in the next slide)
  - After the `=` or `<=` symbol in a procedural assignment



Controlling Verilog Procedures

[Verilog time dimension]

- `initial` and `always` procedures may contain 3 types of timing:

1. Time based delays — the `#` token
   – Delays execution of the next statement for a specific amount of time

```
always         // delayed for 2 simulation time units
  #2 sum = a + b;
```

2. Edge sensitive delays — the `@` token
   – Delays execution of the next statement until a change occurs on a signal

```
always         // delayed until positive edge of clock
  @(posedge clock) sum <= a + b;
```

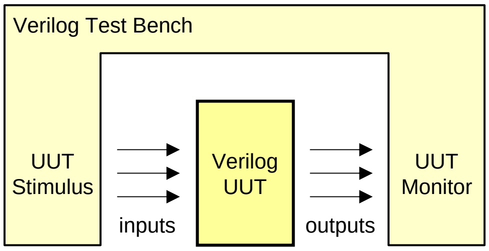
3. Level sensitive delays — the `wait` keyword
   – Delays execution of the next statement until a logic test evaluates as TRUE

```
always         // delayed until 'enable' becomes '1'
  wait (enable == 1) sum = a + b;
```

- Each time control delays execution of the next statement or statement group

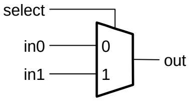
Verilog Test Benches

- ***Unit under test* (UUT)** = the entity/module being tested
  - Also called Device under test (DUT)
- *Verilog Test Bench* consists of:
  - UUT
  - UUT stimulus, to provide inputs to the UUT
  - UUT monitor, to capture and analyze the UUT output



Example Verilog Test Bench (1)

- Unit under test: mux2
  (described in Lecture #23)



```
/* 2-input multiplexor test bench #1 */
`timescale 1 ns / 100 ps
module mux2_tb1 ( );
    wire m_out;
    reg m_sel, m_in0, m_in1;

    mux2 m2_uut (m_in0, m_in1, m_sel, m_out);

    initial begin
        m_in0 = 1'b0;
        m_in1 = 1'b0;
        m_sel = 1'b0;
        $display ("time: %d, output: %d", $time, m_out);
        #5      // wait 5 ns before continuing
        m_in0 = 1'b1;
        m_sel = 1'b1;
        $display ("time: %d, output: %d", $time, m_out);
        $finish; // task call ends simulation
    end
endmodule // mux2_tb1
```

```
/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
    input in0,in1,select;
    output out;
    wire s0,w0,w1;
    not (s0, select);
    and (w0, s0, in0),
            (w1, select, in1);
    or (out, w0, w1);
endmodule // mux2
```

Two concurrent statements:

- Instance statement
- `initial` procedure

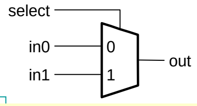
Both automatically become active at time zero

The `initial` procedure changes the input values for UUT as it runs *continuously*

Note *blocking* assignments

Example Verilog Test Bench (2)

- Generating Test Vectors



```
/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
    input in0,in1,select;
    output out;
    wire s0,w0,w1;
    not (s0, select);
    and (w0, s0, in0),
            (w1, select, in1);
    or (out, w0, w1);
endmodule // mux2
```

```
/* 2-input multiplexor test bench #2 */
`timescale 1 ns / 100 ps
module mux2_tb2 ( );
    wire m_out;
    reg [2:0] test_vectors; // 3-bit wide test vector
    integer i;

    mux2 m2_uut (.in0(test_vectors[2]), .in1(test_vectors[1]),
         .select(test_vectors[0]), .out(m_out) );

    initial begin // initialize all variables
         test_vectors = 3'b000;
    end

    initial begin
         for (i=0; i<7; i=i+1) begin
             #5   // wait 5 ns before continuing
             test_vectors = test_vectors + 1;
             $display ("time: %d, output: %d", $time, m_out);
         end
    end
endmodule // mux2_tb2
```

- Now, all the data is stored in "test_vectors."
- The most significant bit, is assigned to "in0", the next to "in1" and the last to "select".
- In the second `initial` block, we generate all the test vectors, 000 through 111, in a `for` loop.
- Note that the #5 waits 5 ns before going to the next test vector.

Self-Checking Test Bench

- SystemVerilog `assert` statement checks if specified condition is true; if not, it executes the `else` statement
- The `$error` system task prints and error message describing the assertion failure

```
/* 2-input multiplexor test bench #3 */
`timescale 1 ns / 100 ps
module mux2_tb3 ( );
    wire m_out;
    reg m_sel, m_in0, m_in1;

    mux2 m2_uut (m_in0, m_in1, m_sel, m_out);

    initial begin
        m_in0 = 1'b0;
        m_in1 = 1'b0;
        m_sel = 1'b0;
        assert ( m_out === 0 ) else $error("000 failed");
        #5      // wait 5 ns
        m_in0 = 1'b1;
        m_sel = 1'b1;   // selects m_in1, which is 1'b0
        assert ( m_out === 0 ) else $error("101 failed");
        $finish;
    end
endmodule // mux2_tb3
```