Structured Procedures

- There are two structured procedure statements in Verilog: always and initial.
- These statements are the two most basic statements in behavioral modeling.
- All other behavioral statements can appear only inside these structured procedure statements.
- Verilog is a concurrent programming language unlike the C programming language, which is sequential in nature.
- Activity flows in Verilog run in parallel rather than in sequence.
- Each always and initial statement represents a separate activity flow in Verilog.
- Each activity flow starts at simulation time 0.
- The statements always and initial cannot be nested.

initial Statement

- All statements inside an initial statement constitute an initial block
- An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again
- If there are multiple initial blocks, each block starts to execute concurrently at time 0
- Each block finishes execution independently of other blocks
- Multiple behavioral statements must be grouped, typically using the keywords begin and end
- If there is only one behavioral statement, grouping is not necessary

Example 1 initial Statement

```
1. module stimulus;
2. reg x,y, a,b, m;
3. initial
4. m = 1'b0; //single statement; does not need to be grouped
5. initial
6. begin
7. #5 a = 1'b1; //multiple statements; need to be grouped
8. #25 b = 1'b0;
9. end
10. initial
11. begin
12. #10 x = 1'b0;
13. #25 y = 1'b1;
14. end
15. initial #50 $finish;
16. endmodule
```

- In the above example, the three initial statements start to execute in parallel at time 0

- If a delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time.

- Thus, the execution sequence of the statements inside the initial blocks will be as follows.

```
time                    statement executed
0                       m = 1'b0;
5                       a = 1'b1;
10                      x = 1'b0;
30                      b = 1'b0;
35                      y = 1'b1;
50                      $finish;
```

- The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run

- The following subsections discussion how to initialize values using alternate shorthand syntax

- The use of such shorthand syntax has the same effect as an initial block combined with a variable declaration

Example 2 Initial Value Assignment

- //The clock variable is defined first
- reg clock;
- //The value of clock is set to 0
- initial clock = 0;
- //Instead of the above method, clock variable can be initialized at the time of declaration
- //This is allowed only for variables declared at module level
- reg clock = 0;

Combined Port/Data Declaration and Initialization

- The combined port/data declaration can also be combined with an initialization
- Example 3 shows such a declaration

❑Example 3 Combined Port/Data Declaration and Variable Initialization

```
1. module adder (sum, co, a, b, ci);
2. output reg [7:0] sum = 0; //Initialize 8 bit output sum
3. output reg co = 0; //Initialize 1 bit output co
4. input [7:0] a, b;
5. input ci;
6. --
7. --
8. endmodule
```

- **Example 4 Combined ANSI C Port Declaration and Variable Initialization**

```
1. module adder (output reg [7:0] sum = 0, //Initialize 8 bit output
2. output reg co = 0, //Initialize 1 bit output co
3. input [7:0] a, b,
4. input ci
5. );
6. --
7. --
8. endmodule
```

- All behavioral statements inside an always statement constitute an always block
- The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion
- This statement is used to model a block of activity that is repeated continuously in a digital circuit
- An example is a clock generator module that toggles the clock signal every half cycle
- In real circuits, the clock generator is active from time 0 to as long as the circuit is powered on

```
1. module clock_gen (output reg clock);
2. //Initialize clock at time zero
3. initial
4. clock = 1'b0;
5. //Toggle clock every half-cycle (time period = 20)
6. always
7. #10 clock = ~clock;
8. initial
9. #1000 $finish;
10. endmodule
```

- In Example 5, the always statement starts at time 0 and executes the statement clock = ~clock every 10 time units

- Notice that the initialization of clock has to be done inside a separate initial statement

- If we put the initialization of clock inside the always block, clock will be initialized every time the always is entered

- Also, the simulation must be halted inside an initial statement. If there is no $stop or $finish statement to halt the simulation, the clock generator will run forever

- Procedural assignments update values of reg, integer, real, or time variables
- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value
- The syntax for the simplest form of procedural assignment is shown below
- assignment ::= variable_lvalue = [delay_or_event_control] expression

- The left-hand side of a procedural assignment <lvalue> can be one of the following:
  - A reg, integer, real, or time register variable or a memory element
  - A bit select of these variables (e.g., addr[0])
  - A part select of these variables (e.g., addr[31:16])
  - A concatenation of any of the above
- The right-hand side can be any expression that evaluates to a value

- Blocking assignment statements are executed in the order they are specified in a sequential block
- A blocking assignment will not block execution of statements that follow in a parallel block
- The = operator is used to specify blocking assignments

- Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block

- A <= operator is used to specify nonblocking assignments. Note that this operator has the same symbol as a relational operator, less_than_equal_to

- The operator <= is interpreted as a relational operator in an expression and as an assignment operator in the context of a non-blocking assignment

- Having described the behavior of non-blocking assignments, it is important to understand why they are used in digital design
- They are used as a method to model several concurrent data transfers that take place after a common event
- Consider the for example where three concurrent data transfers take place at the positive edge of clock

1. always @(posedge clock)
2. begin
3. reg1 <= #1 in1;
4. reg2 <= @(negedge clock) in2 ^ in3;
5. reg3 <= #1 reg1; //The old value of reg1
6. end

❑At each positive edge of clock, the following sequence takes place for the non-blocking assignments

1. A read operation is performed on each right-hand-side variable, in1, in2, in3, and reg1, at the positive edge of clock. The right-hand-side expressions are evaluated, and the results are stored internally in the simulator

2. The write operations to the left-hand-side variables are scheduled to be executed at the time specified by the intra-assignment delay in each assignment, that is, schedule "write" to reg1 after 1 time unit, to reg2 at the next negative edge of clock, and to reg3 after 1 time unit

- The write operations are executed at the scheduled time steps. The order in which the write operations are executed is not important because the internally stored right-hand-side expression values are used to assign to the left-hand-side values

- For example, note that reg3 is assigned the old value of reg1 that was stored after the read operation, even if the write operation wrote a new value to reg1 before the write operation to reg3 was executed

- //Illustration 1: Two concurrent always blocks with blocking statements

1. always @(posedge clock)
2. a = b;
3. always @(posedge clock)
4. b = a;

- //Illustration 2: Two concurrent always blocks with nonblocking statements

1. always @(posedge clock)
2. a <= b;
3. always @(posedge clock)
4. b <= a;

Example Implementing Nonblocking Assignments using Blocking Assignments

- //Emulate the behavior of nonblocking assignments by using temporary variables and blocking assignments

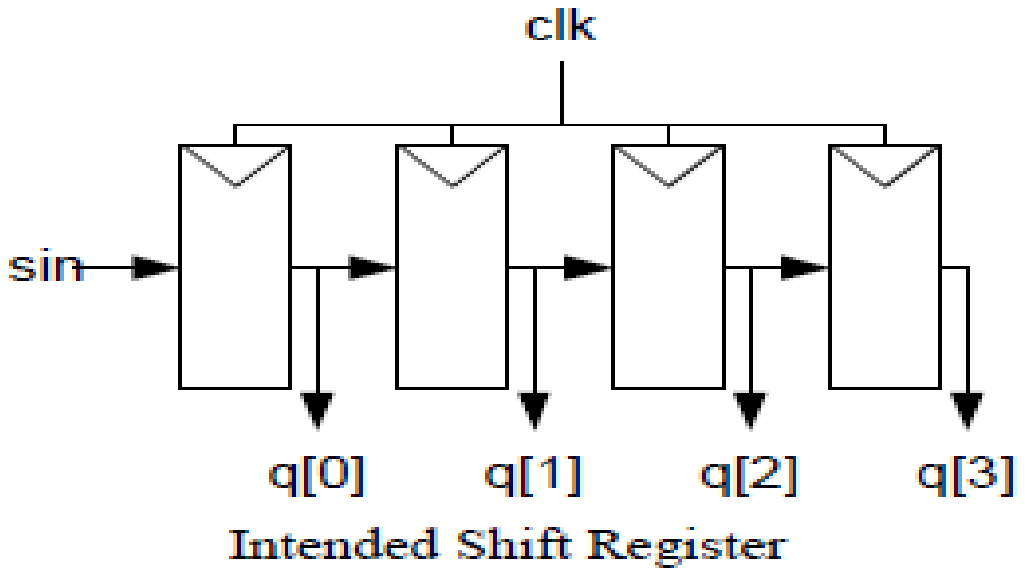
```
1. always @(posedge clock)
2. begin
3. //Read operation store values of right-hand-side expressions in temporary variables
4. temp_a = a;
5. temp_b = b;
6. //Write operation Assign values of temporary variables to left-hand-side variables
7. a = temp_b;
8. b = temp_a;
9. end
```

Blocking and Nonblocking Ex-2

- Verilog supports two types of assignments inside an always block. *Blocking assignments* use the = statement. *Non-blocking assignments* use the <= statement
- Do not confuse either type with the assign statement, which cannot appear inside always blocks at all
- A group of blocking assignments inside a begin/end block are evaluated sequentially, just as one would expect in a standard programming language
- A group of non-blocking assignments are evaluated in parallel
- All of the statements are evaluated before any of the left hand sides are updated. This is what one would expect in hardware because real logic gates all operate independently rather than waiting for the completion of other gates.

- For example, consider two attempts to describe a shift register.

- On each clock edge, the data at sin should be shifted into the first flop. The first flop shifts to the second flop. The data in the second flop shifts to the third flop, and so on until the last element drops off the end.



Intended Shift Register

Blocking and Nonblocking Assignment Ex-2 Cont'd

```
1.module shiftreg(clk, sin, q);
2.input clk;
3.input sin;
4.output [3:0] q;
5.// This is a correct implementation using nonblocking assignment
6.reg [3:0] q;
7.always @(posedge clk)
8.begin
9.q[0] <= sin; // <= indicates nonblocking assignment
10.q[1] <= q[0];
11.q[2] <= q[1];
12.q[3] <= q[2];
13.// it would be even more better to write q <= {q[2:0], sin};
14.end
15.endmodule
```

Blocking and Nonblocking Assignment Ex-2 Cont'd

- The non-blocking assignments mean that all of the values on the right hand sides are assigned simultaneously.

- Therefore, q[1] will get the original value of q[0], not the value of sin that gets loaded into q[0]. This is what we would expect from real hardware. Of course all of this could be written on one line for brevity

- Blocking assignments are more familiar from traditional programming languages, but inaccurately model hardware.

- Consider the same module using blocking assignments. When clk rises, the Verilog says that q[0] should be copied from sin. Then q[1] should be copied from the new value of q[0] and so forth. All four registers immediately get the sin value.

Blocking and Nonblocking Assignment Ex-2 Cont'd

```
1.module shiftreg(clk, sin, q[3:0]);
2.input clk;
3.input sin;
4.output [3:0] q;
5.// This is a bad implementation using blocking assignment
6.reg [3:0] q;
7.always @(posedge clk)
8.begin
9.q[0] = sin; // = indicates blocking assignment
10.q[1] = q[0];
11.q[2] = q[1];
12.q[3] = q[2];
13.end
14.endmodule
```

- The moral of this illustration is to always use non-blocking assignment in always blocks when writing structural Verilog. With sufficient cleverness, such as reversing the orders of the four commands, one might make blocking assignments work correctly, but they offer no advantages and great risks.

- Finally, note that each always block implies a separate block of logic. Therefore, a given reg may be assigned in only one always block. Otherwise, two pieces of hardware with shorted outputs would be implied.

- For digital design, use of non-blocking assignments in place of blocking assignments is highly recommended in places where concurrent data transfers take place after a common event
- In such cases, blocking assignments can potentially cause race conditions because the final result depends on the order in which the assignments are evaluated
- Non-blocking assignments can be used effectively to model concurrent data transfers because the final result is not dependent on the order in which the assignments are evaluated
- Typical applications of non-blocking assignments include pipeline modeling and modeling of several mutually exclusive data transfers
- On the downside, non-blocking assignments can potentially cause a degradation in the simulator performance and increase in memory usage

Sequential and Parallel Blocks

- The keywords begin and end are used to group statements into sequential blocks
- The statements in a sequential block are processed in the order they are specified
- A statement is executed only after its preceding statement completes execution[except for non-blocking assignments with intra-assignment timing control).
- If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

1. reg x,y;
2. reg [1:0]z,w;
3. initial
4. begin
    1. x=1’b0; //competes at simulation time 0
    2. #5 y=1’b1; //completes at simulation time 5
    3. #10 z={x,y}; //competes at simulation time 15
    4. #20 w={y,x}; //complete at simulation time 35
5. end

- Parallel blocks, specified by keyword fork and join
- Statements in a parallel block are executed concurrently
- Ordering of statements is controlled by the delay or event control assignment to each statement
- If delay or event control is specified, it is relative to the time the block was entered
- The keyword fork splits a single flow into independent flows
- The keyword join joins the independent flows back into single flow

1. reg x,y;
2. reg [1:0]z,w;
3. initial
4. fork
    1. x=1’b0; //competes at simulation time 0
    2. #5 y=1’b1; //completes at simulation time 5
    3. #10 z={x,y}; //competes at simulation time 10
    4. #20 w={y,x}; //complete at simulation time 20
5. join

- We discuss three special features available with block statements:
  - Nested blocks
  - Named blocks, and
  - disabling of named blocks

- Blocks can be given names
- Local variables can be declared for the named block
- Named blocks are a part of the design hierarchy. Variables in a named block can be accessed by using hierarchical name referencing
- Named blocks can be disabled, i.e., their execution can be stopped

Example 1 Named Blocks

```
//Named blocks
module top;
initial
begin: block1 //sequential block named block1
integer i; //integer i is static and local to block1  can be accessed by hierarchical name, top.block1.i
...
...
end
initial
fork: block2 //parallel block named block2
reg i; // register i is static and local to block2
// can be accessed by hierarchical name, top.block2.i
...
...
join
```

- Blocks can be nested. Sequential and parallel blocks can be mixed, as shown in Example 2
- **Example 2 Nested Blocks**
- //Nested blocks

```
1. initial
2. begin
3. x = 1'b0;
4. fork
5. #5 y = 1'b1;
6. #10 z = {x, y};
7. join
8. #20 w = {y, x};
9. end
```

Disabling named blocks

- The keyword disable provides a way to terminate the execution of a named block
- disable can be used to get out of loops, handle error conditions, or control execution of pieces of code, based on a control signal
- Disabling a block causes the execution control to be passed to the statement immediately succeeding the block
- For C programmers, this is very similar to the break statement used to exit a loop
- The difference is that a break statement can break the current loop only, whereas the keyword disable allows disabling of any named block in the design
- The while loop can be recoded, using the disable statement as shown in Example 3. The disable statement terminates the while loop as soon as a true bit is seen

Example 3 Disabling Named Blocks

- //Illustration: Find the first bit with a value 1 in flag (vector variable)

```
1.  reg [15:0] flag;
2.  integer i; //integer to keep count
3.  initial
4.  begin
5.  flag = 16'b 0010_0000_0000_0000;
6.  i = 0;
7.  begin: block1 //The main block inside while is named block1
8.  while(i < 16)
9.  begin
10. if (flag[i])
11. begin
12. $display("Encountered a TRUE bit at element number %d", i);
13. disable block1; //disable block1 because you found true bit.
14. end
15. i = i + 1;
16. end
17. end
18. end
```

Verilog HDL - Loops

- There are four types of looping statements in Verilog: while, for, repeat, and forever
- The syntax of these loops is very similar to the syntax of loops in the C programming language
- All looping statements can appear only inside an initial or always block
- Loops may contain delay expressions

1. while
2. for
3. repeat
4. forever

while loop

- The keyword while is used to specify this loop
- The while loop executes until the while-expression becomes false

- increment count from 0 to 127. And exit at count 128 and display the count variable

```
1. integer count;
2. initial begin
3. count = 0;
4. while (count < 128)    //exit at count 128
5. begin
6. $display("Count=%d", count);
7. count=count+1;
8. end
9. end
```

- Find the first bit with a value 1 in flag

```
1.  reg[15:0]flag;
2.  integer i;
3.  reg continue;
4.  initial
5.  begin
6.  flag=16'b0010_0000_0000_0000;
7.  i=0;
8.  continue=1;
9.  while(i<16 && continue) // Multiple conditions using operators.
10. begin
11. if(flag[i]) begin
12. $display("Encountered a TRUE bit at element number %d", i);
13.  continue = 0;
14. end
15. i=i+1;
16. end
17. end
```

for loop

- The keyword for is used.
- The for loop contains three parts:
  - An initial condition
  - A check to see if the terminating condition is true
  - A procedural assignment to change value of the control variable

1. integer count;
2. initial
   3. for (count=0; count < 128; count=count+1)
      4. $display(“Count = %d”, count);

for loop –Cont'd

Initialize an array or memory

```
`define MAX_STATES 32
integer state[0: `MAX_STATES-1]; // Integer array state with elements 0:31
integer i;
initial
begin
    for(i=0;i<32;i=i+2) //initialize all even locations with 0
        state[i]=0;
    for(i=1;i<32;i=i+2)
        state[i]=1;
        end
```

repeat loop

- The keyword repeat is used for this loop
- The repeat construct executes the loop a fixed number of times
- A repeat construct cannot be used to loop a general logical expression

Increment and display count from 0 to 127 integer

```
1.  integer count;

2.  initial
3.  begin
4.     count=0;
5.     repeat(128)
6.     begin
7.           $display(Count=%d“, count);
8.           count = count +1;
9.    end
10. end
```

forever loop

- The keyword forever is used
- The loop does not contain any expression and executes forever until
- The $finish task and disable statement is encountered

- clock generation

```
1. reg clock;
2. initial
3. begin
        1. clock=1'b0;
        2. forever #10 clock=~clock; //clock with period of 20 units
4.  end
```