Module3

Procedural Statements and Routines

System Verilog Procedural Statements

- New Operators
  - Increment and decrement operator
    - Used the same way as C

```
for (i=0; i<=31; i++)
   begin
   …
   end
```

    - Post-increment/decrement and pre-increment/decrement

```
j=i++
j=++i
j=i--;
j=-i;
```

    - May cause a race condition since ++ and -- operators behave as blocking assignments

System Verilog Procedural Statements

- Enhanced `for` loops
  - `for` loop variables are declared outside the loop in Verilog
    - Concurrent loops interfere with each other
  - SystemVerilog allows the `for` loop variables to be declared inside the loop
    - Each variable is local and unique to the loop, hence same name usage does not cause any interference
    - Local loop variables are automatic
    - The variables do not exist outside the loops

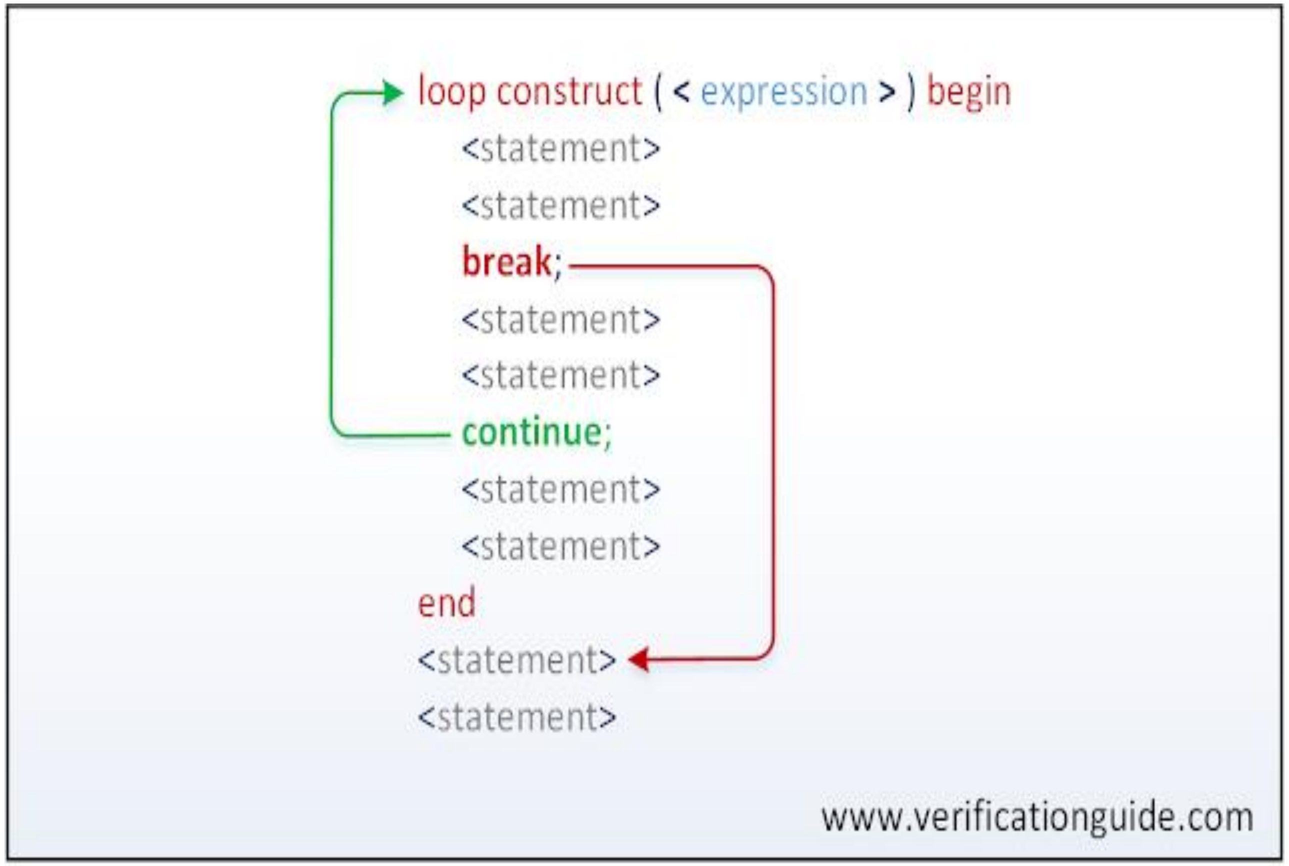
- always @(posedge clock) begin
- for(int i=1; i<=1024; i=i+1)
-   …
- end
- endmodule

System Verilog Procedural Statements

- `do…while` loop
  - A `while` loop in Verilog might not execute at all
    - If the test of control values is false the first time the loop is encountered in the execution flow
  - SystemVerilog adds a `do..while` loop (similar to C)
    - A do while loop executes atleast once
    - Control of the loop is tested at the end of each pass of the loop
      - All logic for setting the outputs of the loop can be placed inside the loop
      - Makes code more concise and intuitive

```
do <statement or statement block>
while (<condition>);
```

break and continue statements



System Verilog Procedural Statements

- **break and continue:**
- The execution of a break statement leads to the end of the loop.
- break shall be used in all the loop constructs (while, do-while, foreach, for, repeat and forever).
- Execution of continue statement leads to **skip** the execution of statements followed by continue and jump to next loop or iteration value

Example for break statement

```
module break_in_loop;

  initial begin
    $display("-------------------------------------------------------------");

    for(int i=0;i<8;i++) begin
      $display("\tValue of i=%0d",i);
      if(i == 4) begin
        $display("\tCalling break,");
        break;
      end
    end

    $display("-------------------------------------------------------------");
  end
endmodule
```

- when the loop value equals 4, the break is called this leads to the end of the loop.

Example for continue statement

```
module continue_in_loop;

  initial begin
    $display("-----------------------------------------------------------------");

    for(int i=0;i<8;i++) begin

      if((i > 2) && (i < 7))begin
        $display("\t\tCalling continue,");
        continue;
      end

      $display("\t\tAfter Continue\t:: Value of i=%0d",i);
    end

    $display("-----------------------------------------------------------------");

  end

Endmodule
```

- when ever the loop value is with in 3 to 6, continue statement will be executed, this leads to skip the execution of display statement after the continue.

Tasks and functions

- Tasks and Functions provide a means of splitting code into small parts.
- A Task can contain a declaration of parameters, input arguments, output arguments, in-out arguments, registers, events, and zero or more behavioral statements.
- **Static tasks**
- Static tasks share the same storage space for all task calls.
- **Automatic tasks**
- Automatic tasks allocate unique, stacked storage for each task call.

- Task declration can be as in verilog 1995/2001 or can be declared as in C or C++. In SystemVerilog following rules hold good for any Task declaration
- **Default Port Direction :** Any port is seen as **input**, unless declared as other types. Following are port types
- **input :** copy value in at beginning
- **output :** copy value out at end
- **inout :** copy in at beginning and out at end
- **ref :** pass reference

- **Default Data TYpe :** Unless declared, data types of ports are of **logic** type.
- **begin..end :** There is **no need** to have begin, end, when more then one statement is used.
- **return :** A task can be terminated before endtask, by usage of **return** statement.
- **Variables :** Systemverilog allows to have local static, or local dynamic variables.
- **life time :** SystemVerilog allows a task to static or automatic.
- **wire :** Wire data type can not be used in port list;

task arguments in parentheses

```
module sv_task;
  int x;

  //task to add two integer numbers.
  task sum(input int a,b,output int c);
    c = a+b;
  endtask

  initial begin
    sum(10,5,x);
    $display("\tValue of x = %0d",x);
  end
endmodule
```

task arguments in declarations and mentioning directions

```
• module sv_task;
•   int x;
•
•   //task to add two integer numbers.
•   task sum;
•     input int a,b;
•     output int c;
•     c = a+b;
•   endtask
•
•   initial begin
•     sum(10,5,x);
•     $display("\tValue of x = %0d",x);
•   end
• endmodule
```

Functions

- A Function can contain declarations of range, returned type, parameters, input arguments, registers, and events.
- A function without a range or return type declaration returns a one-bit value
- Any expression can be used as a function call argument
- Functions cannot contain any time-controlled statements, and they cannot enable tasks
- Functions can return only one value

- Function declration can be as in verilog 1995/2001 or can be declared as in C or C++. In SystemVerilog following rules hold good for any Function declaration
- **Default Port Direction :** Any port is seen as **input**, unless declared as other types. Following are port types
- **input :** copy value in at beginning
- **output :** copy value out at end
- **inout :** copy in at beginning and out at end
- **ref :** pass reference

- **Default Data TYpe :** Unless declared, data types of ports are of **logic** type.
- **begin..end :** There is **no need** to have begin, end, when more then one statement is used.
- **return :** A function can be terminated before endfunction, by usage of **return** statement.
- **Variables :** Systemverilog allows to have local static, or local dynamic variables.
- **life time :** SystemVerilog allows a function to static or automatic.
- **Wire :** Wire data type can not be used in port list;
- **void :** SystemVerilog allows functions to be declared as type void

function arguments in parentheses

```
•  module sv_function;
•    int x;
•    //function to add two integer numbers.
•    function int sum(input int a,b);
•      sum = a+b;
•    endfunction
•
•    initial begin
•      x=sum(10,5);
•      $display("\tValue of x = %0d",x);
•    end
•  endmodule
```

function arguments in declarations and mentioning directions

```
•  module sv_function;
•    int x;
• 
•    //function to add two integer numbers.
•    function int sum;
•      input int a,b;
•      sum = a+b;
•    endfunction
•    initial begin
•      x=sum(10,5);
•      $display("\tValue of x = %0d",x);
•    end
•  endmodule
```

function with return value with the return keyword

```
module sv_function;
  int x;

  //function to add two integer numbers.
  function int sum;
    input int a,b;
    return a+b;
  endfunction

  initial begin
    x=sum(10,5);
    $display("\tValue of x = %0d",x);
  end
endmodule
```

Void function

```
•  module sv_function;
•    int x;
•    //void function to display current simulation time
•    function void current_time;
•      $display("\tCurrent simulation time is %0d",$time);
•    endfunction
•
•    initial begin
•      #10;
•      current_time();
•      #20;
•      current_time();
•    end
```

discarding function return value

- The function return value must be assigned to a variable or used in an expression.
- Calling a function without return value assigned to a variable can result in a warning message. SystemVerilog void data type is used to discard a function’s return value without any warning message.

```
module sv_function;
  int x;
   //function to add two integer numbers.
  function int sum;
    input int a,b;
    return a+b;
  endfunction

  initial begin
    $display("Calling function with void");
    void'(sum(10,5));
  end
 endmodule
```

function call as an expression

```
•  module sv_function;
•    int x;
•    //function to add two integer numbers.
•    function int sum;
•      input int a,b;
•    return a+b;
•    endfunction
•    initial begin
•      x = 10 + sum(10,5);
•      $display("\tValue of x = %0d",x);
•    end
•  endmodule
```

SystemVerilog Tasks and Functions

- Return Statement
  - SystemVerilog adds a return statement
    - If a return statement is executed that value is returned else the last value assigned to the function name is the return value
    - A return statement can be used to exit a task or a function

```
function int add_and_inc (input [31:0] a,b);
   add_and_inc=a+b+1;
endfunction
```

```
function int add_and_inc (input [31:0] a,b);
   add_and_inc=a+b;
   return ++add_and_inc;
endfunction
```

SystemVerilog Tasks and Functions

- Void functions
  - Void functions do not return a value
  - Output and inout formal arguments allow a void function to propagate changes to the scope
  - A void function can be called like a task but must adhere to the restriction of function contents

SystemVerilog Tasks and Functions

- Passing argument values by reference instead of copy
  - Values are passed to tasks and functions by copy (most often)
  - SystemVerilog has explicit pass by reference task/function arguments
    - Use keyword `ref` (instead of input, output or inout)
    - Only automatic tasks and functions can have ref arguments
  - Pass by reference can be read-only
    - Allows the task/function to reference the information in the calling scope
    - Prohibits the task or function from modifying the information
  - Task ref arguments are sensitive to change
  - Ref arguments can read current values
  - Ref arguments can propagate changes immediately

Example code

```
•  typedef struct{
•       logic       valid;
•       logic [7:0] check;
•       logic [63:0] data;
•  } packet_t;
•  packet_t data_packet;
•  logic [7:0] raw_data [0:7];

•  always@(posedge clock)
•  if(data_ready)
•  fill_packet (.data_in(raw_data), .data_out(data_packet));

•  function automatic void fill_packet (
•       ref logic [7:0] data_in [7:0], ref packet_t data_out);
•       for(int i=0; i<=7; i++) begin
•       data_out.data[(8*i)+:8]=data_in[i]; end
•  endfunction
•  endmodule
```