N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

Fall 2022

Verilog/SystemVerilog (I)

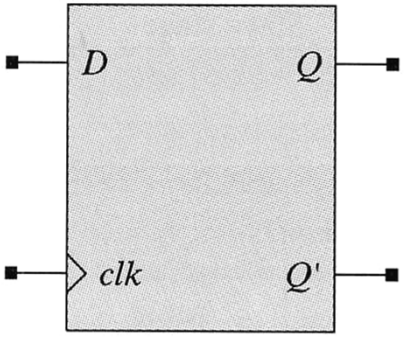
Outline

- History of Verilog
- Logic Values
- Structure Style of Modeling
- Data Structure
- Behavioral Style of Modeling
- Sequential Blocks – DFF & FSM
- Task and Function
- Assertion

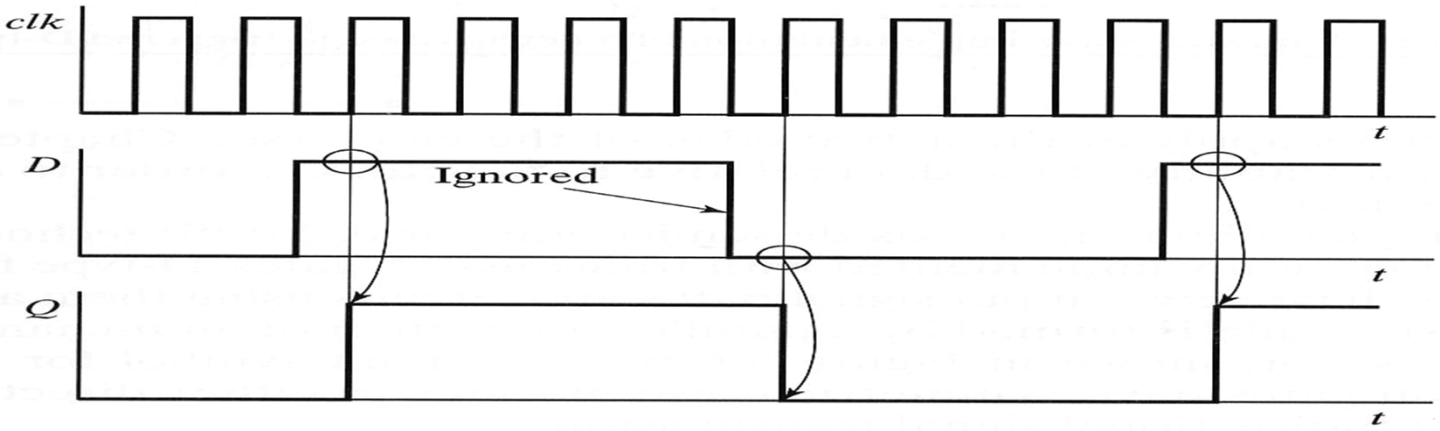
3

Sequential Basics – DFF & FSM

Edge-Triggered D Flip-flop



D	Q	Q_{next}
0	0	0
0	1	0
1	0	1
1	1	1

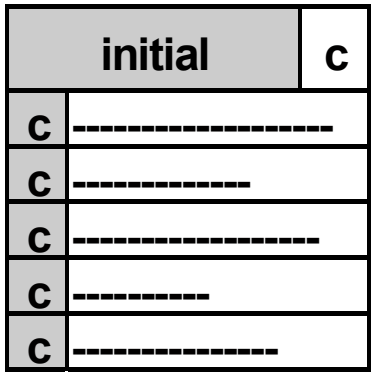


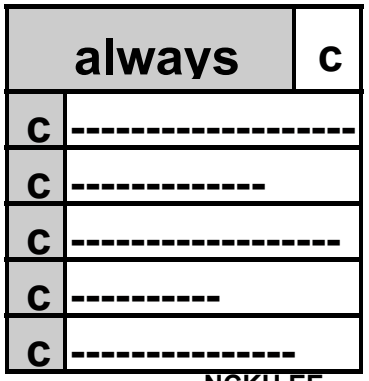
Behavioral DFF


```
module df_behav (q, q_bar, data, set, reset, clk);
input          data, set, clk, reset;
   output          q, q_bar;
   reg          q;

   assign q_bar = ~ q;

   always @   (posedge clk)   // Flip-flop with synchronous set/reset
   begin
      if (reset == 0) q <= 0;
         else if (set ==0) q <= 1;
            else q <= data;
   end
endmodule
```

Procedural Blocks

- Procedural blocks are the basis for behavioral modeling
- Procedural blocks are of two types
  - [illegible]
  - [illegible]
- Procedural blocks have the following components
  - **Procedural assignment statements**
  - **Timing controls**
  - **High-level programming language constructs.**





always @(*)

```
// equivalent to @(a or b or c or d or f)

always @(*)
  y = (a & b) | (c & d) | myfunction(f);
```

```
// equivalent to @(a or b or c or d or tmp1 or tmp2)

always @* begin
  tmp1 = a & b;
  tmp2 = c & d;
  y    = tmp1 | tmp2;
end
```

```
// equivalent to @(b)

always @*
  @(i) kid = b;    // i is not added to @*
```


Source: C. M. Huang / SystemVerilog

always @(*)

```
// equivalent to @(a or b or c or d)

always @* begin
  x = a ^ b;
  @*                // equivalent to @(c or d)
  x = c ^ d;
end
```

```
// same as @(a or en)

always @* begin
  y    = 8'hff;
  y[a] = !en;
end
```


Source: C. M. Huang / SystemVerilog

always @(*)

```
// same as @(state or go or ws)

always @* begin
  next = 4'b0;

  case (1'b1)
    state[IDLE]: if (go)  next[READ] = 1'b1;
                 else     next[IDLE] = 1'b1;
    state[READ]:          next[DLY ] = 1'b1;
    state[DLY ]: if (!ws) next[DONE] = 1'b1;
                 else     next[READ] = 1'b1;
    state[DONE]:          next[IDLE] = 1'b1;
  endcase
end
```


Source: C. M. Huang / SystemVerilog

General Purpose Usage of always

- Modeling functions
  - Combinational logic @ RTL
  - Latched logic @ RTL
  - Sequential logic @ RTL
  - Algorithmic logic @ behavioral
- Burden on synthesis compiler
  - Compiler needs to deduce what type of hardware is represented in your code

`always_comb` and `always_latch`

- SystemVerilog provides a special `always_comb` procedure for modeling combinational logic behavior.
- SystemVerilog also provides a special `always_latch` procedure for modeling latched logic behavior.
- The `always_latch` construct is identical to the `always_comb` construct except that software tools should perform additional checks and warn if the behavior in an `always_latch` construct does not represent latched logic, whereas in an `always_comb` construct, tools should check and warn if the behavior does not represent combinational logic.

`always @*` vs `always_comb`

- `always_comb` automatically executes once at time zero, `always @*` waits until a change occurs in the inferred sensitivity list.
- Variables on the left-hand side of assignments within an `always_comb` procedure shall not be written to by any other processes, `always @*` permits multiple processes to write to the same variable.
- Statements in an `always_comb` shall not include event controls or `fork-join` statements.
- `always_comb` is sensitive to changes within the contents of a function, whereas `always @*` is only sensitive to changes to the arguments of a function.

Source: C. M. Huang / SystemVerilog

always @* vs always_comb

```
always_comb
  a = b;

always_comb
  a = c;
```

```
always_comb
  a = b;

always @*
  a = c;
```

```
always_comb
  begin
    #1 a = b;
  end
```

```
always_comb
  fork
    ...
  join
```

```
always @*
  a = b;

always @*
  a = c;
```

```
always_comb @(b)
  a = b;
```

```
always @*
  begin
    #1 a = b;
  end
```

```
always @*
  fork
    ...
  join
```


Source: C. M. Huang / SystemVerilog

```
// always01.sv - always @*, always_comb, and always_latch

module always01;
  integer a1, a2, a3, b, c;

  always @* begin
    a1 = b + c;
    $display("always @*    : @%g a = %d", $time, a1);
  end

  always_comb begin
    a2 = b + c;
    $display("always_comb : @%g a = %d", $time, a2);
  end

  always_latch begin
    a3 = b + c;
    $display("always_latch: @%g a = %d", $time, a3);
  end

  initial begin
    #10 b = 99;
    #15 c = 99;
    #10 $finish;
  end

endmodule
```

```
always_comb : @0 a =          x
always_latch: @0 a =          x
always @*   : @10 a =         x
always_comb : @10 a =         x
always_latch: @10 a =         x
always @*   : @25 a =       198
always_comb : @25 a =       198
always_latch: @25 a =       198
```


Source: C. M. Huang / SystemVerilog

always_ff

- The `always_ff` procedure can be used to model synthesizable sequential logic behavior.
- The `always_ff` procedure imposes the restriction that it contains one and only one event control.
- Variables on the left-hand side of assignments within an `always_ff` procedure shall not be written to by any other process.
- Software tools should perform additional checks to warn if the behavior within an `always_ff` procedure does not represent sequential logic.

Source: C. M. Huang / SystemVerilog

always_ff

```
always_ff
  a = b;
```

✗

```
always_ff @(posedge clk1 or negedge clk2)
  a = b;
```

○

```
always_ff @(posedge clk1)
  @(negedge clk2) a = b;
```

✗

```
always_ff @(posedge clk1)
  a = b;

always_ff @(posedge clk2)
  a = c;
```

✗

Source: C. M. Huang / SystemVerilog

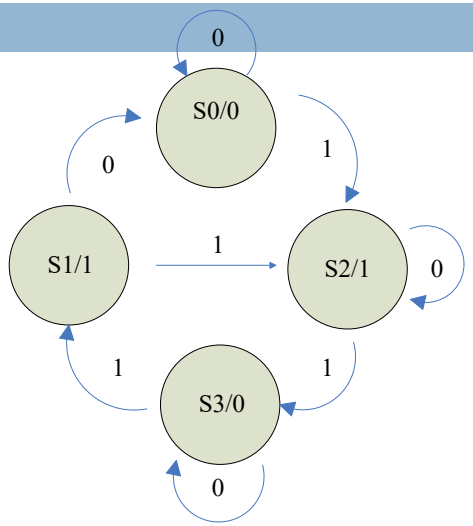
What Controller Do?

- Determine the execution order of computation
  - Load data from memory byte by byte
  - Load 9 pixels from RF (register file)
  - Find max values (or computation)
  - Output results to memory
  - ….

Moore Machine

- **Moore model**

Moore Machine



Current State	Next State		Qout
	Din=0	Din=1	
S0=00			0
S1=01			1
S2=10			1
S3=11			0

- Use binary numbers to represent each state

```
parameter [1:0] S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;
```

- Reading data input and changes state for every clock cycles
- Upon reset, machine goes back to state S0
- Next state will be determined by current state and input

Verilog Code (1/3)


```
`timescale 1ns/10ps
module moore (Qout, clk, rst, Din);
output Qout;
input clk, rst, Din;
parameter [1:0] S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;
reg Qout;
reg [1:0] CS, NS;   // CS: current state; NS: next state

always @ (posedge clk or posedge rst)
begin
if (rst==1'b1) CS=S0;
else CS = NS;
end
```

Verilog Code (2/3)


```
always @ (CS or Din)
begin
case (CS)
  S0:   begin
            Qout=1'b0;
            if (Din==1'b0) NS=S0;
            else NS = S2;
         end
  S1:   begin
            Qout=1'b1;
            if (Din==1'b0) NS=S0;
            else NS = S2;
        end
```

Verilog Code (3/3)


```
  S2:  begin
            // fill the rest of the code
       end
  S3:  begin
            // fill the rest of the code
       end
   endcase
end // always(cs or din)
endmodule
```

FSM with Enumerated Types (Poor)


```
module traffic_light (……)
enum {RED, GREEN, YELLOW} State, Next;

always_ff @(posedge clk, negedge resetN)
  if (!resetN) State <= RED; //Reset to red light
  else              State <= Next;

always_comb begin: set_next_state
  Next = State; // the default for each branch below
  unique case (State)
    RED:            if (sensor)                    Next= Green;
    GREEN:          if (green_downcnt==0)          Next = YELLOW;
    . . .
  endcase
end: set_next_state

always_comb begin: set_outputs
  {green_light, yellow_light, red_light} = 3'b000;
  unique case (State)
    RED:            red_light = 1'b1;
    GREEN:          green_light = 1'b1;
    YELLOW:         yellow_light = 1'b1;
   endcase
  end:  set_outputs
endmodule
```

- Default type of enum is int, a 2-state type
- Lead to mismatches in RTL simulation between gate-level implementation.

One-hot Encoding w/ Enumerated Types


```
module traffic_light2 (……) //
enum logic [2:0] {RED = 3'b001, GREEN = 3'b010, YELLOW = 3'b100} State, Next; //explicit

always_ff @(posedge clk, negedge resetN)
  if (!resetN) State <= RED; //Reset to red light
  else         State <= Next;

always_comb begin: set_next_state
  Next = State; // the default for each branch below
  unique case (State)
    RED:    if (sensor)                  Next= Green;
    GREEN:  if (green_downcnt==0)        Next = YELLOW;
    . . .
  endcase
end: set_next_state

always_comb begin: set_outputs
  {green_light, yellow_light, red_light} = 3'b000;
  unique case (State)
    RED:            red_light = 1'b1;
    GREEN:          green_light = 1'b1;
    YELLOW:         yellow_light = 1'b1;
   endcase
  end:  set_outputs
endmodule
```

- logic is a 4-state type
- Help to compare pre- and post-Synthesis model functionality.

Block Names

- Both sequential and parallel blocks can be named by adding `:name_of_block` after the keywords `begin` or `fork`. A named block creates a new hierarchy scope. The naming of blocks serves the following purposes:
  - It allows local variables, parameters, and named events to be referenced hierarchically, using the block name.
  - It allows the block to be referenced in statements such as the `disable` statement
- A matching block name may be specified after the block `end`, `join`, `join_any`, or `join_none` keyword, preceded by a colon. It shall be an error if the name at the end is <u>different</u> from the block name at the beginning.

```
begin: blockB
  ...
end: blockB
```


Source: C. M. Huang / SystemVerilog

Statement Labels

- A label can be specified before any procedural statement (any non-declaration statement that can appear inside a begin-end block).
- A statement label is used to identify a single statement. The label name is specified before the statement, followed by a colon.

```
labelA: statement
```

- A begin-end or fork-join block is considered a statement, and can have a statement label before the block. Specifying a statement label before a `begin` or `fork` keyword is equivalent to specifying a block name after the keyword, and a matching block name may be specified after the block `end`, `join`, `join_any`, or `join_none` keyword.

```
labelB: fork     // label before the begin or fork
  ...
join_none : labelB
```


Source: C. M. Huang / SystemVerilog

```
// named_blobk01.sv - named block

module named_blobk01();

  reg clk = 0;

  initial
    FIRST_BLOCK : begin
      $display ("This is the first block");
    end

  initial begin : SECOND_BLOCK
     $display ("This is the second block");

     fork : FORK_BLOCK
       #1 $display ("Inside fork with delay 1");
       #2 $display ("Inside fork with delay 2");
     join_none

     FORK_NONE : fork
       #4 $display ("Inside fork with delay 4");
       #5 $display ("Inside fork with delay 5");
     join_none

     #10 $finish;
  end

  always begin : THIRD_BLOCK
    #1 clk = ~clk;
  end : THIRD_BLOCK

endmodule
```

```
This is the first block
This is the second block
Inside fork with delay 1
Inside fork with delay 2
Inside fork with delay 4
Inside fork with delay 5
```

28

Task and Function

Subroutines: Task and Function

- There are two types of sub-programs that encapsulate and organize a Verilog description: tasks and functions.
- A `task` is typically used to behaviorally describe hardware, or to perform debugging operations.
- A `function` is typically used to perform a computation (expression), or to represent combinational logic.
- Tasks and functions encourage the readability, portability, and maintainability of codes.

Source: C. M. Huang / SystemVerilog

Task Example

```
module Bit_Counter(data_word, bit_count);
  input  [7:0] data_word;
  output [3:0] bit_count;
  reg    [3:0] bit_count;

  always @(data_word)
    count_ones_in_word(data_word, bit_count);

  task count_ones_in_word;
    input  [7:0] reg_a;
    output [3:0] count;
    reg    [3:0] count;
    reg    [7:0] temp_reg;

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

endmodule
```

referenced within behavior

declared within module

arguments

local variable

Function Example

```
module MULT(m1, m2, a, b);
  input  [7:0]  a, b;
  output [15:0] m1, m2;
  wire   [15:0] m1;
  reg    [15:0] m2;

  assign m1 = mult(a, b);

  always @(a or b)
    m2 = mult(a, b);

  function [15:0] mult;
    input  [7:0] x, y;

    mult = x * y;

  endfunction

endmodule
```

referenced with continuous assignment

referenced with procedural assignment

declared within module

arguments

Source: C. M. Huang / SystemVerilog

Distinction between Task and Function

- Tasks and functions provide the ability to execute common procedures from several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions.
- The following rules distinguish tasks from functions:
  - The statements in the body of a function shall execute in one simulation time unit; a task may contain time-controlling statements.
  - A function cannot enable a task; a task can enable other tasks and functions.
  - A nonvoid function shall return a single value; a task or void function shall not return a value.
  - A nonvoid function can be used as an operand in an expression; the value of that operand is the value returned by the function.

Source: C. M. Huang / SystemVerilog

Static and Automatic Variables

- [Verilog-1995] All variables are static
- [Verilog-2001] Variables in a task or function can be defined as automatic
  - the variable storage is dynamically allocated when required and deallocated when vanished
  - Intended for representing verification routines
  - Allow coding recursive function calls
- [SystemVerilog] Allow any variables to be explicitly declared as either static or automatic

Recursive Call using automatic


```
function automatic int b_add (int lo, hi);
  int mid = (lo + hi +1) >>1;
  if (low + 1 != hi)
          return (b_add(lo, (mid-1)) + b_add(mid, hi));
  else
          return (array[lo] + array[hi]);
endfunction
```

lo and *hi* will be initialized when the function was entered because of variables defined as *automatic*.

Static Variable Initialized Only Once


```
function int count_ones (input [31:0] data);
  logic [31:0] count = 0;   // initialized once
  logic [31:0] temp = data; // initialized once

  for (int i=0; i<=32; i++) begin
     if (temp[0] count++;
     temp >>=1;
  end
  return (count);
endfunction
```

- count is initialized to 0 for the first time and will not be re-initialized the next time it is called.
- Resulting an error in count

Automatic Variable Initialized per Call


```
function int count_ones (input [31:0] data);
  automatic logic [31:0] count = 0;  // initialized once
  automatic logic [31:0] temp = data; // initialized once

  for (int i=0; i<=32; i++) begin
     if (temp[0] count++;
     temp >>=1;
  end
  return (count);
endfunction
```

- count is initialized to 0 each time it is called.
- A variable declared in an automatic function or task will be automatic by default.

Guidelines for Static and Automatic Variables

- In a procedural block,
  - Use static variables if there no in-line initialization
  - Use automatic variables otherwise
- In a task or function,
  - Use automatic variables if recursively call
  - Use static if representing the behavioral of a single piece of hardware

Tasks

- A task declaration has the formal arguments either in parentheses or in declarations and directions.

```
task mytask1 (output int x, input logic y);
  ...
endtask

task mytask2;
  output x;
  input  y;
  int    x;
  logic  y;
  ...
endtask
```

```
input     // copy value in at beginning
output    // copy value out at end
inout     // copy in at beginning and out at end
ref       // pass reference
```


Source: C. M. Huang / SystemVerilog

Tasks (Synthesizable if no timing constructs)

- There is a default direction of input if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments `a` and `b` default to inputs, and `u` and `v` are both outputs:

```
task mytask3(a, b, output logic [15:0] u, v);
  ...
endtask
```

- An array can be specified as a formal argument to a task.

```
// the resultant declaration of b is input [3:0][7:0] b[3:0]
task mytask4(input [3:0][7:0] a, b[3:0], output [3:0][7:0] y[1:0]);
   ...
endtask
```


Source: C. M. Huang / SystemVerilog

```
// task01.sv foreach loop with task

module task01;

  logic [7:0] a [];

  task print(input logic [7:0] t []);       // inout ???
    foreach (t[i])
      begin
        t[i] = $random;
        $display("t[%0d] = %b", i, t[i]);
      end
  endtask

  initial begin

    a = new[3];
    print(a);
    foreach (a[i])
      $display("a[%0d] = %b", i, a[i]);

    a = new[5];
    print(a);
    foreach (a[i])
      $display("a[%0d] = %b", i, a[i]);

  end

endmodule
```

```
t[0] = 00100100
t[1] = 10000001
t[2] = 00001001
a[0] = xxxxxxxx
a[1] = xxxxxxxx
a[2] = xxxxxxxx

t[0] = 01100011
t[1] = 00001101
t[2] = 10001101
t[3] = 01100101
t[4] = 00010010
a[0] = xxxxxxxx
a[1] = xxxxxxxx
a[2] = xxxxxxxx
a[3] = xxxxxxxx
a[4] = xxxxxxxx
```

```sv
// task01.sv foreach loop with task

module task01;

  logic [7:0] a [];

  task print(input logic [7:0] t []);
    foreach (t[i])
      begin
        t[i] = $random;
        $display("t[%0d] = %b", i, t[i]);
      end
  endtask

  initial begin
    a = new[3];
    print(a);
    foreach (a[i])
      begin
        a[i] = $random;
        $display("a[%0d] = %b", i, a[i]);
      end

    a = new[5];
    print(a);
    foreach (a[i])
      begin
        a[i] = $random;
        $display("a[%0d] = %b", i, a[i]);
      end
  end

endmodule
```

```
t[0] = 00100100
t[1] = 10000001
t[2] = 00001001
a[0] = 01100011
a[1] = 00001101
a[2] = 10001101

t[0] = 01100101
t[1] = 00010010
t[2] = 00000001
t[3] = 00001101
t[4] = 01110110
a[0] = 00111101
a[1] = 11101101
a[2] = 10001100
a[3] = 11111001
a[4] = 11000110
```

Functions (Synthesizable)

- The primary purpose of a function is to return a value that is to be used in an expression. A `void` function can also be used instead of a task to define a subroutine that executes and returns within a single time step.
- A function declaration has the formal arguments either in parentheses or in declarations and directions:

```
function logic [15:0] myfunc1(int x, int y);
  ...
endfunction

function logic [15:0] myfunc2;
  input int x;
  input int y;
  ...
endfunction
```


Source: C. M. Huang / SystemVerilog

Return Values and void Functions

- The function definition shall implicitly declare a variable, internal to the function, with the same name as the function. This variable has the same type as the function return value.
- Function return values can be specified in two ways, either by using a `return` statement or by assigning a value to the internal variable with the same name as the function.

```
function [15:0] myfunc1 (input [7:0] x,y);
  myfunc1 = x * y - 1;       // return value assigned to function name
endfunction

function [15:0] myfunc2 (input [7:0] x,y);
  return x * y - 1;          //return value is specified using return statement
endfunction
```


Source: C. M. Huang / SystemVerilog

Return Values and void Functions

- Functions can be declared as type `void`, which do not have a return value. Function calls may be used as expressions unless of type `void`, which are statements:

```
a = b + myfunc1(c, d); // call myfunc1 as an expression

myprint(a);            // call myprint as a statement

function void myprint (int a);
  ...
endfunction
```

- The function can be used as a statement and the return value discarded without a warning by casting the function call to the `void` type.

```
void'(some_function());
```


Source: C. M. Huang / SystemVerilog

Pass by Value

- Pass by value is the default mechanism for passing arguments to subroutines.
- This argument passing mechanism works by copying each argument into the subroutine area. If the subroutine is automatic, then the subroutine retains a local copy of the arguments in its stack.
- If the arguments are changed within the subroutine, the changes are not visible outside the subroutine.

Source: C. M. Huang / SystemVerilog

Pass by Value

- When the arguments are large, it can be undesirable to copy the arguments. Also, programs sometimes need to share a common piece of data that is not declared global.
- For example, calling the function below copies 1000 bytes each time the call is made.

```
function automatic int crc( byte packet [1000:1] );
  for( int j= 1; j <= 1000; j++ ) begin
    crc ^= packet[j];
  end
endfunction
```


Source: C. M. Huang / SystemVerilog

Pass by Reference

- Arguments passed by reference are not copied into the subroutine area, rather, a reference to the original argument is passed to the subroutine. The subroutine can then access the argument data via the reference.
- Arguments passed by reference shall be matched with equivalent data types. No casting shall be permitted.
- To indicate argument passing by reference, the argument declaration is preceded by the `ref` keyword. It shall be illegal to use argument passing by reference for subroutines with a lifetime of static.

Source: C. M. Huang / SystemVerilog

Pass by Reference

```
function automatic int crc( ref byte packet [1000:1] );
  for( int j= 1; j <= 1000; j++ ) begin
    crc ^= packet[j];
  end
endfunction

...

byte packet1[1000:1];
int k = crc( packet1 ); // pass by value or by reference: call is the same
```

- When the argument is passed by reference, both the caller and the subroutine share the same representation of the argument; therefore, any changes made to the argument, within either the caller or the subroutine, shall be visible to each other.

Source: C. M. Huang / SystemVerilog

```
// task03.sv - pass by reference

module task03;

  integer a;

  task print(integer t);
    t = t + 1;
    $display("print: %g", t);
  endtask

  task automatic print_ref(ref integer t);
    t = t + 1;
    $display("print_ref: %g", t);
  endtask

  initial begin
    a = 100;
    print(a);
    $display("a = %g", a);
    print_ref(a);
    $display("a = %g", a);
  end

endmodule
```

```
print: 101
a = 100
print_ref: 101
a = 101
```


Source: C. M. Huang / SystemVerilog

```
// task01a.sv foreach loop with task

module task01a;

  logic [7:0] a [];

  task automatic print(ref logic [7:0] t []);
    foreach (t[i])
      begin
        t[i] = $random;
        $display("t[%0d] = %b", i, t[i]);
      end
  endtask

  initial begin

    a = new[3];
    print(a);
    foreach (a[i])
      $display("a[%0d] = %b", i, a[i]);

    a = new[5];
    print(a);
    foreach (a[i])
      $display("a[%0d] = %b", i, a[i]);

  end

endmodule
```

```
t[0] = 00100100
t[1] = 10000001
t[2] = 00001001
a[0] = 00100100
a[1] = 10000001
a[2] = 00001001

t[0] = 01100011
t[1] = 00001101
t[2] = 10001101
t[3] = 01100101
t[4] = 00010010
a[0] = 01100011
a[1] = 00001101
a[2] = 10001101
a[3] = 01100101
a[4] = 00010010
```


Source: C. M. Huang / SystemVerilog