Task and Function

Tasks and functions

- Tasks and functions are sub-programs that can be defined in Verilog
- Repeatedly used lines of code can be made into task or function.

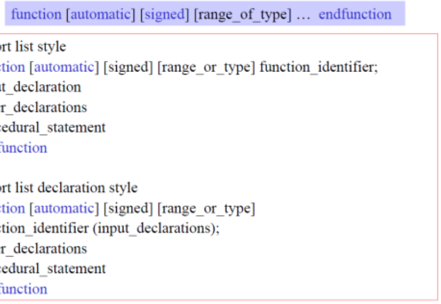
Function

- Functions are equivalent to combinatorial logic and cannot be used to replace code that contains event or delay control operators (as used in a sequential logic)
- Functions are declared within a parent module with the keywords **function** and **endfunction**

Function

- Functions are used if all of the following conditions are true:
  - There are no delay, timing, or event control constructs that are present
  - It returns a single value
  - There is at least one input argument
  - There are no output or inout argument
  - There are no non-blocking assignments

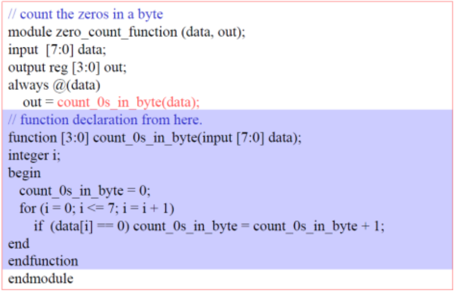
Function Definition and Calls



```
function [automatic] [signed] [range_of_type] …  endfunction
```

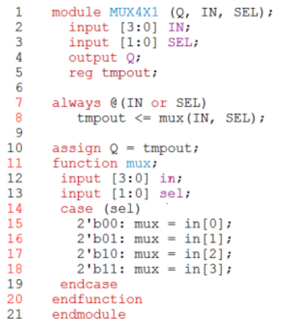
```
// port list style
function [automatic] [signed] [range_or_type] function_identifier;
input_declaration
other_declarations
procedural_statement
endfunction

// port list declaration style
function [automatic] [signed] [range_or_type]
function_identifier (input_declarations);
other_declarations
procedural_statement
endfunction
```

Example: Function and call

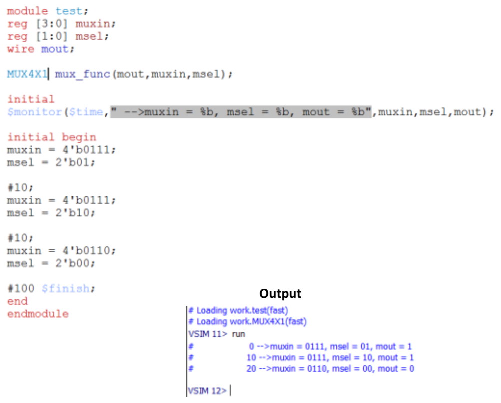


```
// count the zeros in a byte
module zero_count_function (data, out);
input  [7:0] data;
output reg [3:0] out;
always @(data)
    out = count_0s_in_byte(data);
// function declaration from here.
function [3:0] count_0s_in_byte(input [7:0] data);
integer i;
begin
   count_0s_in_byte = 0;
   for (i = 0; i <= 7; i = i + 1)
       if  (data[i] == 0) count_0s_in_byte = count_0s_in_byte + 1;
end
endfunction
endmodule
```

Function - Example



```
module MUX4X1 (Q, IN, SEL);
  input [3:0] IN;
  input [1:0] SEL;
  output Q;
  reg tmpout;

always @(IN or SEL)
   tmpout <= mux(IN, SEL);

assign Q = tmpout;
function mux;
 input [3:0] in;
 input [1:0] sel;
 case (sel)
   2'b00: mux = in[0];
   2'b01: mux = in[1];
   2'b10: mux = in[2];
   2'b11: mux = in[3];
 endcase
endfunction
endmodule
```



```
module test;
reg [3:0] muxin;
reg [1:0] msel;
wire mout;

MUX4X1 mux_func(mout,muxin,msel);

initial
$monitor($time," -->muxin = %b, msel = %b, mout = %b",muxin,msel,mout);

initial begin
muxin = 4'b0111;
msel = 2'b01;

#10;
muxin = 4'b0111;
msel = 2'b10;

#10;
muxin = 4'b0110;
msel = 2'b00;

#100 $finish;
end
endmodule
```

Output

```
# Loading work.test(fast)
# Loading work.MUX4X1(fast)
VSIM 11> run
#                  0 -->muxin = 0111, msel = 01, mout = 1
#                 10 -->muxin = 0111, msel = 10, mout = 1
#                 20 -->muxin = 0110, msel = 00, mout = 0

VSIM 12> |
```

Function Example -Parity Generator

```
module parity;
reg [31:0] addr;
reg parity;

always @(addr)
begin
   parity = calc_parity(addr);
   $display("Parity calculated = %b", calc_parity(addr) );
end

function calc_parity;
input [31:0] address;
begin
   calc_parity = ^address;
end
endfunction
endmodule
```

Function Examples - Controllable Shifter

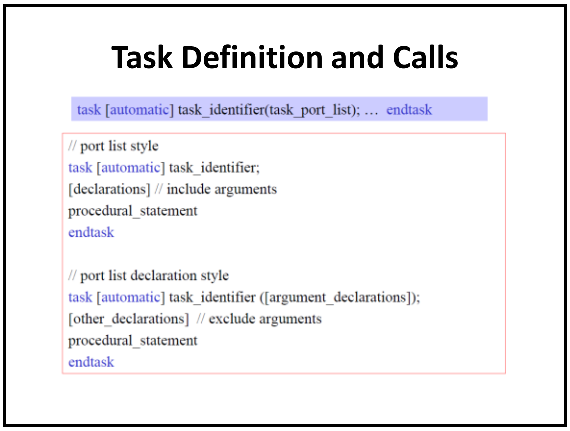
```
module shifter;
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr, right_addr;
reg control;

always @(addr)
begin
  left_addr  =shift(addr, `LEFT_SHIFT);
  right_addr =shift(addr,`RIGHT_SHIFT);
end

function [31:0] shift;
input [31:0] address;
input control;
begin
  shift = (control==`LEFT_SHIFT) ?(address<<1) : (address>>1);
end
endfunction
endmodule
```

Task

- A task is like a procedure which provides the ability to execute common pieces of code from several different places in a model.
- A task can contain timing controls, and it can call other tasks and functions.
- A task can have zero, one, or more arguments.

Task

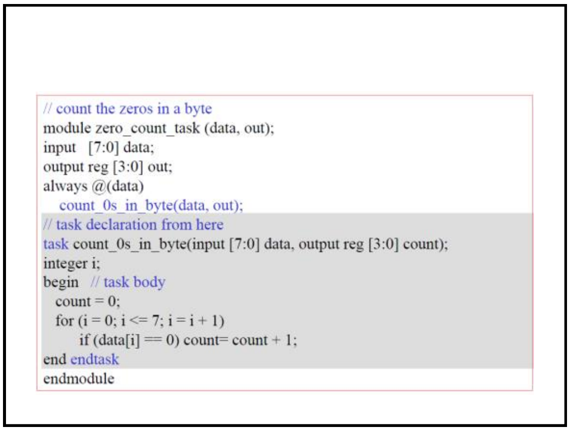
- Values are passed to and from a task through arguments.
- The arguments can be input, output, or inout.
- A task is defined, within a module definition



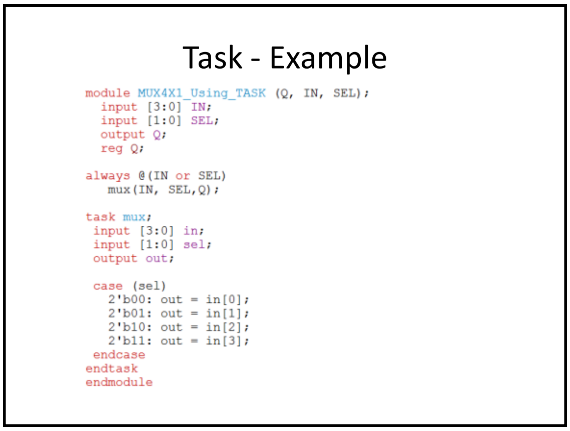
Task Definition and Calls

```
task [automatic] task_identifier(task_port_list); … endtask
```

```
// port list style
task [automatic] task_identifier;
[declarations] // include arguments
procedural_statement
endtask

// port list declaration style
task [automatic] task_identifier ([argument_declarations]);
[other_declarations]  // exclude arguments
procedural_statement
endtask
```

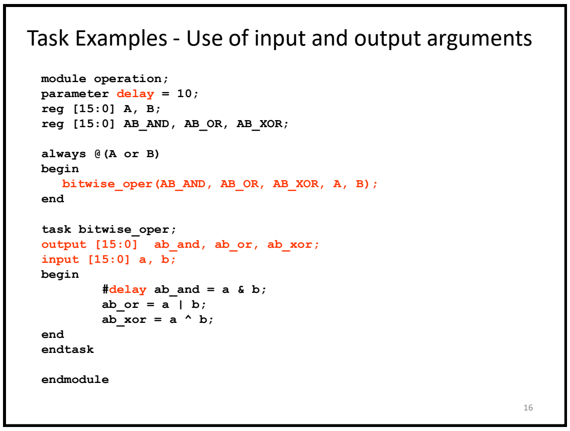


```
// count the zeros in a byte
module zero_count_task (data, out);
input   [7:0] data;
output reg [3:0] out;
always @(data)
   count_0s_in_byte(data, out);
// task declaration from here
task count_0s_in_byte(input [7:0] data, output reg [3:0] count);
integer i;
begin   // task body
  count = 0;
  for (i = 0; i <= 7; i = i + 1)
       if (data[i] == 0) count= count + 1;
end endtask
endmodule
```



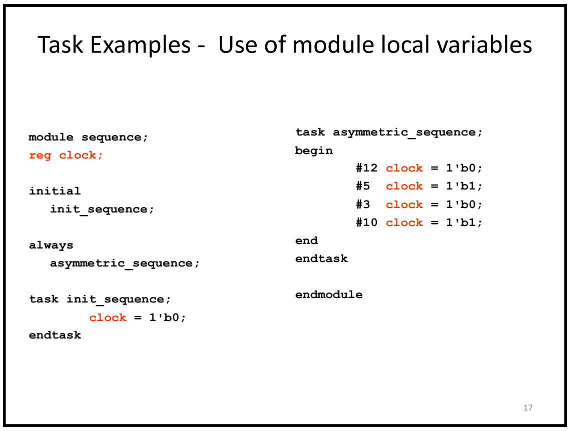
Task - Example

```
module MUX4X1_Using_TASK (Q, IN, SEL);
  input [3:0] IN;
  input [1:0] SEL;
  output Q;
  reg Q;

always @(IN or SEL)
   mux(IN, SEL,Q);

task mux;
 input [3:0] in;
 input [1:0] sel;
 output out;

 case (sel)
   2'b00: out = in[0];
   2'b01: out = in[1];
   2'b10: out = in[2];
   2'b11: out = in[3];
 endcase
endtask
endmodule
```



Task Examples - Use of input and output arguments

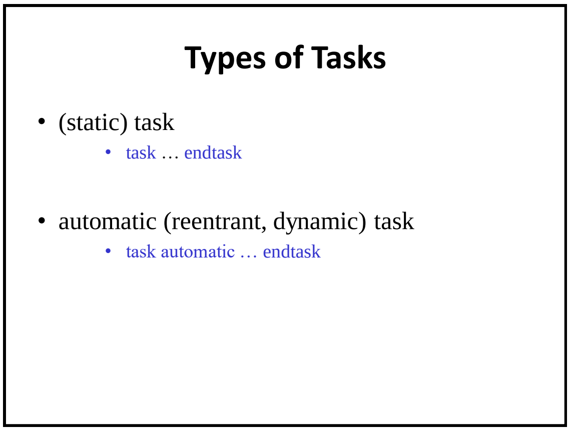
```
module operation;
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

always @(A or B)
begin
   bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
end

task bitwise_oper;
output [15:0]  ab_and, ab_or, ab_xor;
input [15:0] a, b;
begin
        #delay ab_and = a & b;
        ab_or = a | b;
        ab_xor = a ^ b;
end
endtask

endmodule
```



Task Examples - Use of module local variables

```
module sequence;
reg clock;

initial
   init_sequence;

always
   asymmetric_sequence;

task init_sequence;
        clock = 1'b0;
endtask
```

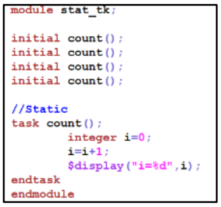
```
task asymmetric_sequence;
begin
        #12 clock = 1'b0;
        #5  clock = 1'b1;
        #3  clock = 1'b0;
        #10 clock = 1'b1;
end
endtask

endmodule
```



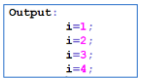
Types of Tasks

- (static) task
  - task … endtask
- automatic (reentrant, dynamic) task
  - task automatic … endtask

- Static Task
  - Member variables will be shared across different invocations of the same task that has been launched to run concurrently

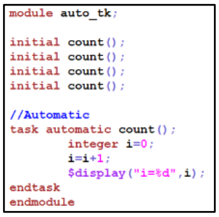


```
module stat_tk;

initial count();
initial count();
initial count();
initial count();

//Static
task count();
        integer i=0;
        i=i+1;
        $display("i=%d",i);
endtask
endmodule
```

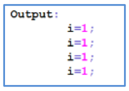


```
Output:
        i=1;
        i=2;
        i=3;
        i=4;
```

- Automatic Task
  - Re-entrant task
  - All items inside are allocated dynamically for each invocation and not shared between invocations of the same task running concurrently



```
module auto_tk;

initial count();
initial count();
initial count();
initial count();

//Automatic
task automatic count();
        integer i=0;
        i=i+1;
        $display("i=%d",i);
endtask
endmodule
```



```
Output:
        i=1;
        i=1;
        i=1;
        i=1;
```

Task and function differences

Task	Function
May execute on non-zero simulation time	Executes on zero simulation time
May have delay, event or timing control constructs	Not possible as it executes on zero simulation time
Cannot return a value	Always return a single value
Pass values (can be multiple) through output or inout arguments	Cannot have output or inout arguments
Can enable other functions and tasks	Can enable other functions and not task
Can have input, output or inout	Must have atleast one input. Cannot have output or inout