Chapter 8 Tasks and Functions

- Tasks and functions provide a mean of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions

8.1Distinctions between tasks and functions

The following rules distinguish tasks from functions:

1. A function must execute in one simulation time unit; a task can contain time-controlling statements.
2. A function cannot enable a task; a task can enable other tasks and functions.
3. A function must have at least one input argument; a task can have zero or more arguments of any type.
4. A function returns a single value; a task does not return a value.

Ex. **task called:** switch_bytes (old_word, new_word);
function call: new_word = switch_bytes (old_word);

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions do not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one input argument. They can have more than one input.	Tasks may have zero or more arguments of type input, output, or inout.
Functions always return a single value. They cannot have output or inout arguments.	Tasks do not return with a value, but can pass multiple values through output and inout argument.

8.2 Tasks and Task Enableing

The syntax for defining tasks:

```
task <name_of_task>;
    <task_declaration>;
    <statement>;
endtask
```

<task_declaration> can include: parameter_declaration, input/output declaration, register_declaration, time_declaration, integer_declaration, real_declaration, event_declaration.

Task Enabling and Argument Passing

(Ex. 9-1)

Example 9-1 : Task definition with arguments

```
task my_task;                    // my_task (v, w, x, y, z);
    input a, b;
    inout c;
    output d, e;
    reg foo1, foo2, foo3;
    begin
        <statements>     //the set of statements that
    foo1=a+b; foo2=b-c; foo3=a*c;
    c = foo1;
    d = foo2;
    e = foo3;
    end
endtask
endmodule
```

Enabling the task:

```
my_task (v, w, x, y, z);
```

The calling arguments (v, w, x, y, z) correspond to theIO arguments (a, b, c, d, e) defined by the task.At task enabling time,

v → a, w →b, x →c;

when the task completes,

c → x, d → y, e →z.

- (Ex. 9-2)

Example 9-2 : Using tasks

```
//Define a module called operation that    contains the task bitwise_oper
module operation;
--
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;
always @(A or B)    //whenever A or B changes in value
begin      //invoke the task bitwise_oper. Provide 2 input arguments A, B
           //Expect 3 output arguments AB_AND, AB_OR, AB_XOR
           //The arguments must be specified in the same order as they
           //appear in the task declaration.
          bitwise_oper (AB_AND, AB_OR, AB_XOR, A, B,);
--
end
//define task bitwise_oper
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor;      //outputs from the task
input [15:0] a, b;        //inputs to the task
begin
           #delay ab_and = a & b;
           ab_or = a | b;
           ab_xor = a ^ b;
end
endtask
--
endmodule
```

```
module traffic_lights;
    reg clock, red, amber, green;
    parameter on = 1, off = 0 , red_tics = 350,
              amber_tics = 30, green_tics = 200 ;
    // initialize colors
    initial
        red = off
    initial
        amber = off;
    initial
        green = off;
    //sequence to control the lights
    always begin
        red = on;    // turn red light on
        light (red, red_tics); //and wait.
        green = on; //turn green light on
        light (green, green_tics); //and wait.
        amber = on; //turn amber light on
        light (amber, amber_tics); //and wait.
    end
    //task to wait for 'tics' positive edge clocks
    //before turning 'color' light off
    task light;
        output color;
        input [31:0] tics;
    parameter hh=0;
        begin
            repeat (tics)
                @(posedge clock);
            color = off; //turn light off
        end
    endtask
    //waveform for the clock
    always begin
        #100 clock = 0 ;
        #100 clock = 1 ;
    end
endmodule            //traffic_lights
```

- Automatic (Re-entrant) Task

  If a task is called concurrently (in different always statements) from two places in the code, these task calls will operate on the same task variables.

Ex. 8-5

```
//There are two clocks. clk2 runs at twice the frequency of clk and is synchronous with clk
module top;
    reg [15:0] cd_xor, ef_xor;      ‘’variables in module top
    reg [15:0] c, d, e, f;          //variables in module top

    task automatic bitwise_xor;
        output [15:0]    ab_xor;        //output from the taskl
        input [15:0] a, b;      //inputs to the task
        begin
            #delay    ab_and = a & b;
            ab_or = a | b;
            ab_xor = a ^ b;
        end
    endtask
    …
    always @(posedge clk)   bitwise_xor (ef_xor, e, f);
    ….
    always @(posedge clk2) bitwise_xor (cd_xor, c, d);
    …
endmodule
```

Verilog-Xl allows multiple copier of a task to execute concurrently, but it does not copy or preserve the task arguments or local variables.

```
module mark1Task;       // execute Instructions
reg [15:0] m [0:8191]; // 8192 x 16 bit memory
reg [12:0] pc; // 13 bit program counter
reg [12:0] acc; // 13 bit accumulator
reg ck; // a clock signal
always begin:
    reg [15:0] ir; // 16 bit instruction register
    @(posedge ck) ir = m [pc];
    @(posedge ck)
        case (ir [15:13])
            …….
            3'b011 : xor (acc, m [ir [12:0]]);
            3'b100 : add (acc, m [ir [12:0]]);
            3'b101 : subtract (acc, m [ir [12:0]]);
            3'b110 : divide (acc, m [ir [12:0]]);
            3'b111 : multiply (acc, m [ir [12:0]]);
        endcase
    pc = pc + 1;
end

task multiply;
inout [12:0] a;
input [5:0] b;
    begin: serialMult
        reg [5:0] mcnd, mpy;//multiplicand and multiplier
        reg [12:0] prod;//product
        mpy = b[5:0];
        mcnd = a[5:0];
        prod = 0;
        repeat (6)
            begin
                if (mpy[0])
```

```
                    prod = prod + {mcnd, 6'b000000};
                prod = prod >> 1;
                mpy = mpy >> 1;
            end
        a = prod;
    end
endtask
endmodule
```

8.3 Functions and Function Calling

- The syntax for function

  function <range_or_type> <name_of_function>;
  <tf_declaration>+
  <statement_or_null>
  endfunction

  The <range_or_type> item is optional.
- (Ex. 9.3)

Example 9-3 : A function definition using range

```
function [7:0] getbyte;
    input [15:0] address;
    reg[3:0] result_expression;
        begin
            //<statements> code to extract low-order
          result_expression=address*5 // byte from addressed word
          getbyte = result_expression;
        end
endfunction
```

- Calling a Function

  word = control ? {getbyte (msbyte), getbyte (lsbyte)} : 0;
  msbyte→address lsbyte→address
- Function Rules

  Functions are more limited than tasks. The following four rules govern their usage:

  1. A function definition cannot contain any time controlled

statements (#, @, or wait).

2. functions cannot enable tasks.
3. A function definition must contain at least one input argument.
4. A function definition must include an assignment of the function result value to the internal variable that has the same name as the function.

- Ex. 8.7 Parity Calculation

```
module parity;
    …
    reg [31:0]    addr;      reg parity;
    …
    always @ (addr)    begin
            parity = calc_parity (addr);     //first invocation of calc_parity
            $display (“parity calculated = %b, calc_parity (addr));
    end
    …
    //define the parity calculation function
    function calc_parity;
            input    [31:0]    address;
            begin
             calc_parity = ^address;
            end
    endfunction
    …
endmodule
```

Ex. 8-9 Lift/Right Shifter

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

  always @(addr)    begin
      left_addr = shift(addr, `LEFT_SHIFT);
      righ_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
```

8.3.3 Automatic (Recursive) Functions

Functions are normally used non-recursively. If a function is called concurrently from two locations, the results are non-deterministic because both calls operate on the same variable space.

Ex. 8-10

```
module top;
    …
    // define the function
    function automatic integer factorial;
        input [31:0] oper;
        integer i;
        begin
            if (iperand >=2)
```

```
            factorial = factorial (oper – 1) * oper;   //recursive call
        else    factorial = 1;
    endfunction

    // call the function
    integer result;
    initial begin
        result = factorial (4);
        $desplay(“factorial of 4 is %od”, result);
    end
    …
endmodule
```

(Ex. 9-4)
Example 9-4 : Defining and calling a function

```
module tryfact;
    // define function
        function [31:0] factorial;
            input[3:0] operand;
            reg [3:0] index;
            begin
                factorial = operand ? 1 : 0;

        for (index = 2; index <= operand; index = index + 1)
                        factorial = index * factorial;
        end
     endfunction

   // Test the function
  reg[31:0] result;
  reg[3:0] n;
  initial
      begin
          result = 1;
          for (n = 2; n<= 9; n = n+1)
              begin
                  $display("Partial result n = %d result=%d", n, result);
                  result = n * factorial(n) / ((n * 2) + 1);
              end
          $display("Final result=%d", result);
      end
endmodule //tryfact

module mark1Fun;
      reg [15:0] m [0:8191]; // 8192 x 16 bit memory
      reg [12:0] pc; // 13 bit program counter
      reg [12:0] acc; // 13 bit accumulator
      reg ck; // a clock signal
      always
```

```
        begin: executeInstructions
            reg [15:0] ir; // 16 bit instruction register
            @(posedge ck)    ir = m [pc];
            @(posedge ck)
                case (ir [15:13])
                    //case expressions, as before
                    3'b111: acc = multiply(acc, m [ir [12:0]]);
                endcase
            pc = pc + 1;
        end
function [12:0] multiply;
    input [12:0] a;
    input [15:0] b;
        begin: serialMult
            reg [5:0] mcnd, mpy;
            mpy = b[5:0];
            mcnd = a[5:0];
            multiply = 0;
            repeat (6) begin
                if (mpy[0])
                    multiply = multiply + {mcnd, 6'b000000};
                multiply = multiply >> 1;
                mpy = mpy >> 1;
                end
        end
    endfunction
endmodule
```

System Tasks and Functions

F.1 Display and Write Tasks

$display(“Some text %d and maybe some more:%h.”,a,b);

h or H display in hexadeximal
d or D display in decimal
o or O display in octal
b or B display in binary
c or C display ASCII character
v or V display net signal strength(see Table 10.4)
m or M display hierarchical name
s or S display string

\n is the new line character
\t is the tab character
\\ is the \ character
\” is the “ character
\ddd is the character specified in up to 3 octal digits

$display(“Hello world\n”);

F.2 Continuous Monitoring

$monitor(parameters as used in the $display task);
$monitor($time,,”regA = ”,regA);

F.4 File Output

$fdisplay(descriptor,parameters as in the display command);
$fwrite(descriptor,parameters as in the write command);
$fmonitor(descriptor,parameters as in the monitor command);
$fstrobe(descriptor,parameters as in the strobe command);

The descriptor is a 32-bit value
$fopen(“name of file”);
$fclose(descriptor);

F.5 Simulation Time

$time is a function that returns the current time as a 64-bit value.$stime will return a 32-bit value.

$monitor($time,,,"regA = ",regA);

F.6 Stop and Finish

The $stop and $finish tasks stop simulation.They differ in that $stop returns control back to the simulator's command interpreter,while $finish returns back to the host operating system.

$stop;
$stop(n);
$finish;
$finish(n);

Parameter Value	Diagnostics
0	Prints nothing
1	Gives simulation time and location
2	Same as 1,plus a few lines of run statistics

F.7 Random

Parameter SEED = 33;
Reg [31:0] vector;

Always @(posedge clock)
    Vector = $random(SEED);

F.8 Reading Data From Disk Files

The $readmemb and $readmemh system tasks are used to load information stored in disk files into Verilog memories.

$readmemx("filename",<memname>,<<start_addr> <,<finish_addr>>?>?);

Exercises 8 p.185 1, 2, 3, 4, 5 Function & task

Exercises 14 p.340 2, 3, 4, 5 Logical Synthesis