


EE461 Verilog-HDL

Week 7 Task & Function

Alex Yang, Engineering School, NPU

Week 7 Outlines

- Function
  - Usage:
    - Mathematical Manipulation (Non-synthesizable); Combinational Logic Sequences (synthesizable); Conversions of Data (Non-synthesizable)
  - Syntax
    - Function Block
    - Function Invocation
    - Examples
- Task
  - Syntax
    - Task Block
    - Task Invocation
    - Examples
- Difference Between Function & Task
- Interesting Questions about Function &Task

Function

- Syntax
  - Function Block

```
File Name: myfunc.v
module myfunc();
   function  myfunction;
      input a, b, c, d;
      begin
         myfunction = ((a+b) + (c-d));
     end
  endfunction
endmodule
```

```
module  top (a, b, c, d, e, f);
    input a, b, c, d, e ;
    output f;
    wire f;
    `include "myfunc.v"
    assign f =  (myfunction (a,b,c,d)) ? e :0;

endmodule
```

```
module tryfact;
   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
  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
```

Function

- Syntax
  - Function Block
    - Position: is created only in module, not in "assign", "initial" & "always" block.
    - Function itself
      1. data type: must be reg, integer, real or time, no more. However, by default it is reg and don't need to be declared. Otherwise, you will get error.

         e.g. function reg[7:0] foo; ... ... endfunction
      2. function's value: is x by default unless a value is assigned to it by statements within function block.
      3. name: can't have the same name as a variable or a task/function.

Function

- Syntax
  - Function Block
    - Input variables:
      1. at least 1 input.
      2. data type: must be reg, integer, real or time, no more. By default, it is reg.
      3. value is copied by order from function invocation, no passing parament by reference like pointer.
      4. scope: local

```
function integer foo;
    input    a, b, c;
 // data type: reg 1 bit if no more declaration
endfunction
```

```
... ... ;
assign A = foo(1,0,1);

Note: a = 1; b = 0; c = 1
```

Function

- Syntax
  - Function Block
    - Output: (no inout in function block)
      1. can't declare output, no output parameter in function.
      2. function's return value is output.
      3. function (itself) return value could be assigned by different data types. Of course, it must be reg, integer, real or time.
      4. function individual bit return value assignment could be allowed

```
function time foo;
    input    a, b, c;
 // data type: reg 1 bit if no more declaration
   foo[1] = a
endfunction
```

Function

- Syntax
  - Function Block
    - Internal variables:
      1. scope: local. It is different from variables with the same names outside function block.
      2. data type: must be reg, integer, real or time.
    - Statements in block:
      1. must start with "begin ... end" for more than one statement
      2. can't include "always"&"initial" blocks
      3. can't have nonblocking statements.
      4. can't have event, delay or timing control statements like #, @ and wait().
      5. can't have "disable" statement.
      6. can't call task, but system task like $display and other funcitons.
      7. can have "assign", but don't recommend

Function

- Syntax
  - Function Block
    - Statements in block:
      8. can include if-else, if-else if- else, case, (or casex, casez: don't recommand), blocking(=), for, while and repeat loop (don't recommend), functions/system tasks.
      9. carry out its required duty within zero simulation time
      10. Can't have forever loop
      11. synthesizable if no non-synthesizable statements.
  - Function Invocation
    - Must have input parameters
    - Allows recursive calling function by "automatic" keyword
    - Allows parameter variable passed to function
    - Not necessary to have the same data type to copy arguments from caller.
    - Can be called in "assign", "always" & "initial" blocks

Function

- Examples:

```
module testFunc(a,b,c,d);
    input a,b,c;
    output d, e;
    reg e;
    assign d = a & b & c;
    always@(a, b)begin
      e= a | b;
      function  foo1;    //Error
              input a, b, c;
              foo1 = a & b & c;
     endfunction
    end
    initial begin
      function  foo2;    //Error
              input a, b, c;
              foo2 = a | b | c;
     endfunction
    end
endmodule
```

```
module testFunc(a,b,c,d,e);
    input a,b,c;
    output d,e;
   reg e;
    assign d = a & b & c;
    always@(a,b,c)begin
      e= foo1(a,b,c);
    end
   function reg foo1;  //Error
              input a, b, c;
              foo1 = a & b & c;
   endfunction

    function  foo2;
              input a, b, c;
              foo2 = a | b | c;
     endfunction
endmodule
```

```
module testFunc(a,b,c,d,e);
    input a,b,c;
    output d,e;
    reg e;
    assign d = a & b & c;
    always@(a,b,c)begin
      e= foo1(a,b,c);// e=?
    end
   function reg foo1;  //Error
              input a, b, c;
              reg d;
              wire e;   //Error
              d = a & b & c;
    endfunction
     function  foo2;
              input a, b, c;
              foo2 = a | b | c;
      endfunction
endmodule
```

Function

- Examples:

```
module testFunc(a,b,c,d);
    input a,b,c;
    output d;
    reg e;
    assign d = a & b & c;
    always@(a, b, c)begin
      e= foo1(a,b,c);
    end
    function  foo1;
          input a, b, c;
          output d; // Error
         begin   //a=a,b=b,c=c
          foo1 = a & b & c;
          d = a | b | c;
         end
    endfunction
endmodule
```

Note: different a in module and function block

```
module testFunc(a,b,c,d,e);
    input a,b,c;
    output d,e;
    reg e;
    assign d = a & b & c;
    always@(a,b,c)begin
      e= foo1(a,b,c)
     end
     function foo1;
              input a, b, c;
              foo1 = a & b & c;
    endfunction
    function  foo2;  //Error
              foo2 = 1'b0;
    endfunction
endmodule
```

Note: at least 1 input in function block

```
module testFunc(a,b,c,d,e);
    input a,b,c;
    output d,e;
    assign d = a & b & c;
    assign e = foo(a,b,c)

     function  foo;
              input a, b, c;
              integer a,b,c;
              reg d;
                 foo= a | b | c;

    endfunction

endmodule
```

Note:1 OK! a(integer)=a(wire)
2 foo(reg) =a|b|c (integer)

Function

- Examples:

```
module callFunc(a, b,c);
  input a, b ;
 output c;
 wire c;
 function foo;
    input a, b;
   begin
     always@(a,b) begin
              foo=a+b;
     end
  end
 endfunction
 assign c = foo(a,b);
endmodul
```

Note: can't include always and initial block

```
module callFunc(a, b,c);
  input a, b ;
 output c;
 wire c;
 function foo;
    input a, b;
     reg c
   begin
       wait(a==1) c=a+b;
        #5 foo=c;
  end
 endfunction
 assign c = foo(a,b);
endmodul
```

Note: can't include wait statement.

```
module callFunc(a, b,c);
  input a, b ;
 output c;
 wire c;
 function foo;
    input a, b;
     reg c
   begin
        c <= a+b;
        foo <= c;
  end
 endfunction
 assign c = foo(a,b);
endmodul
```

Note: can't have non-blocking assignment

Function

- Examples:

```
module callFunc(a, b,c);
  input a, b ;
 output c;
 wire c;
 function foo;
    input a, b;
   begin
      assign foo=a+b;
     $display("Testing");
  end
 endfunction
 assign c = foo(a,b);
endmodul
```

Note: allow to use assign or call other functions or system task, but can't call user-defined task.

```
module multAcc(
      input [7:0] ina, inb;
      input clk, clr;
      output reg[15:0] macO
);
  wire[15:0] multO,adderO;
  reg[15:0] o;
   parameter set=10;
   parameter hld=20;
   assign adderO=multO+o;
always@(posedge clk)begin
   if(clr)  o<=16'h0000;
   else    o<=adderO;
... ...
assign multO=mult(ina,inb);
... ...
```

```
function [15:0] mult;
  input[7:0] a, b;
  reg[15:0] r;
 integer i;
 begin
      if(a[0]==1) r=b;
      else r=0;
      for(i=1,i<=7;i=i+1)begin
            r=r+b<<1;
      end
      mult=r;
 end
endfunction
endmodule
```

Note: within begin-end block, allow to use if/case/for loop etc.

Function

- Examples:

```
module testInput();
  function [7:0] test;
     input integer num;
    begin
        assign test = num+1;
    end
  endfunction
  initial begin //Error
$display ("test: %0d", test());
    #1 $finish;
end
endmodule
```

Note: # of input parameters must match that in function block.

```
module factorial(
   input [7:0] n,
   output [15:0] result
);
 assign result = fact(7);
function automatic[15:0] fact;
   input [7:0] N;
   if (N == 1) fact = 1;
  else fact = N * fact(N - 1);
endfunction
initial begin
  $display ("result:%d", result);
end
endmodule
```

Note: Allows recursive funciton call by "automatic"

```
module testRec(
            a,b,cin,sum,co);
  input a,b,cin;
  output sum,co;
  assign {co,sum}=a+b+cin;
//Illegal to have recursive module
      testRec u0(
                  .a (a),
                  .b (b),
                  .cin (cin),
                  .sum (sum),
                  .co  (co)
      );
endmodule
```

Note: don't allow recursive module

Function

- Examples: Constant function

```
module register_file (…);
  parameter NUM_ENTRIES=64;
  localparam NUM_ADDR_BITS=ceil_log2(NUM_ENTRIES);

  function [31: 0] ceil_log2(input [31: 0] value);
    reg sticky;
    reg [31:0] temp; //reg variable is OK as for-loop index
    begin
                        sticky = 1'b0;
                        for (temp=32'd0; value>32'd1; temp=temp+1) begin
                                if((value[0]) & (|value[31:1]))
                                        sticky = 1'b1;
                                value = value>>1;
                        end
                        clogb2 = temp + sticky;
            end
  endfunction
... ...
endmodule
```

Task

- Syntax
  - Task Block
    - Position: is created only in module, not in "assign", "initial" & "always" block.
    - Task itself
      1. data type: can't declare any date types.Otherwise, you will get error.
      2. value: can't assign return value to it.
      3. name: can't have the same name as a variable or a function/task.

```
task foo;    //Error:  task reg foo;
   input    a, b, c;
   output    d;
   ... ...
endtask
```

```
reg e,f;
initial begin
   foo(1,0,1,e); // "f = foo(1,0,1,e)" =>Error
end
```

Note: a = 1; b = 0; c = 1; e=d

Task

- Syntax
  - Task Block
    - Input variables:
      1. 0 or 1 or more inputs, and inout is allowed.
      2. data type: must be reg, integer, real or time, no more. By default, it is reg.
      3. value is copied by order from function invocation, no passing parament by reference like pointer, could be copied from different data type like wire.
      4. scope: local. Variable name is different from the same name outside task.
      5. can't be timing control signals, such as clock.

Task

- Syntax
  - Task Block
    - Input variables:

```
wire x, y;
task foo;
    inout    a, b;  //no inputs, by dafault, is reg
    output   c;
             c = x + y; // OK, if x&y are from outside
endtask
```

```
reg a;
task foo;
    input        a, b, c; //different from above a
   output        foo;
                 foo = a + b;
endtask
initial begin
                 foo(0,1,0,a)
end   // a=0,b=1,c=0;  a=foo(output)
```

```
input  clk;
reg    a;        //if a is wire,   error after calling task
task foo;
   input         a, b;  //by dafault, is reg
   output        c;
            c = a + b;
endtask
 initial begin
                 foo(clk,1,a);        // clk is input. Avoid it
end              // a = clk, b=1, a = c(output of task)
```

```
reg     x, y;
task foo;
   input         a, b;
   output        c;
          wait(x==1)  c = a + b;
endtask
 initial begin
                 foo(0,1,y);
end  // x is from outside and a timing control signal
```

Task

- Syntax
  - Task Block
    - Output:
      1. 0 or 1 or more output, and inout could be allowed.
      2. variable name could be the same as task name, but task itself doesn't have this return value.
      3. individual bit output value assignment in task could be allowed.
      4. output variables could be from outside of task block. But it must be reg, integer, real or time.

```
task foo;
    input    a, b, c;
             x = a + b;   //If a = 1, b=1, x=?
endtask // x is global variable. x=0, It's OK
```

```
task foo;
    input    a, b, c;
   output    foo;
             foo = a + b;
endtask // foo(output) is different from foo(task)
```

Task

- Syntax
  - Task Block
    - Internal variables:
      1. scope: local. It is different from variables with the same names outside task block.
      2. data type: must be reg, integer, real or time.
      3. variable from outside in LHS must be reg. RHS don't care
    - Statements in block:
      1. must start with "begin ... end" for more than one statement
      2. CAN'T include "always"&"initial" blocks
      3. can have nonblocking statements.
      4. can have event, delay or timing control statements like #, @ and wait(), but non-synthesizable
      5. can have "disable" statement.
      6. can call task,system task like $display and other funcitons.
      7. can have "assign", but don't recommend.

Task

- Syntax
  - Task Block
    - Statements in block:
      8. can include if-else, if-else if- else, case, (or casex, casez: don't recommand), blocking(=), for, while repeat loop (don't recommend), functions/system tasks.
      9. carry out its required duty within non zero simulation time
      10. Can't have forever loop
      11. synthesizable if no timing control & non-synthesizable statements.
  - Task Invocation
    - May have 0 input or 0 output parameter or none of them
    - Recommend: use "task automatic foo" for multi invocations
    - Allows parameter variable passed to task
    - Not necessary to have the same data type to copy arguments from caller.
    - Can be called in "always" & "initial" blocks, not assign.
    - Be careful about multi-times calling task with timing control.

Task

- Examples:

```
module zero_count_task(data,
out);
    input [7:0] data;
    output reg [3:0] out;
   always @(data)
      countZero(data, out);
   task countZero(
     input [7:0] data, output
                reg[3:0] count);
     integer i;
    begin
        count = 0;
        for (i = 0; i <= 7; i = i + 1)
          if (data[i] == 0) count= count + 1;
    end
  endtask
endmodule
```

Note: out(module)must be reg

```
module taskParam();
    parameter b=1;
    reg[1:0] a;
        task test;
          input[1:0] a;
          output[1:0] test;
          begin
             test=a;
           $display("Hello World!");
          end
        endtask
        initial begin
              test(b,a);
           $display("a=%0b",a);
        end
endmodule
```

Note: parameter variable could be input. task could call system task.

```
module testTask();
   task disp;
      input integer a;
      input integer d;
     begin
 #(d) $display("%t,d is %0d,a is %0d", $time,d,a);
      end
   endtask
   initial
       #10 disp(10,14);
   initial
       #14 disp(23,4);
   initial
       #4 disp(11,14);
   initial
       #100 $finish;
endmodule
```

What are results? and why?

18 d is 4,a is 23

18 d is 4,a is 23

24 d is 4,a is 23

Task

- Examples:

```
module top;
   reg clk, a, b;
   DUT u1(out,a,b,clk);
   always #5 clk=!clk;
   task negClk;
      input[31:0] numEdges;
      repeat(numEdges)
         @(negedge clk);
   endtask
   initial begin
            clk=0; a=1; b=1;
            negClk(3);
            a=0;
            negClk(5);
            b=0;
   end
endmodule
```

Note: no output

```
module mult(clk,a,b,out,
enMult);
   input clk, enMult;
   input[3:0] a,b;
   output[7:0] out;
   always@(posedge clk)
            multMe(a,b,out);
   task multMe;
      input[3:0] xMe, toMe;
      output[7:0] result;
      wait(enMult)
            result=xMe*toMe;
   endtask
endmodule
```

Note: timing control signal from outside involved

```
module top(clk, ... ...);
   reg[15:0] cdXOR, efXOR;
   reg[15:0] c,d,e,f;
task automatic bitwiseXOR;
   output[15:0] abXOR;
   input[15:0] a,b;
   begin
      abXOR= a ^ b;
   end
endtask
always@(posedge clk)
      bitwiseXOR(efXOR,e,f);
always@(posedge clk)
      bitwiseXOR(cdXOR,c,d);
endmodule
```

Note: automatic=dynamic
2 groups of local variables:
G1: a=e,b=f; G2 a=d,b=d

Task

- Examples:

```
module alu(func,a,b,c);
  input[1:0] func;
  input[3:0] a,b;
  output[3:0] c;
  reg[3:0] c;
  task myAnd;
    input[3:0] a,b;
    output[3:0] andOut;
    integer i;
    begin
      for(i=3;i>=0;i=i-1)
        andOut[i]=a[i]&b[i];
    end
  endtask
  always@(*)begin
    case(func)
      2'b00: myAnd(a,b,c);
      2'b01: c=a|b;
      2'b10: c=a-b;
      default: c=a+b;
  end
endmodule
```

```
module trafficLights;
  reg clock, red, amber, green;
  parameter on = 1, off = 0
  redTics = 350,amber_Tics = 30,
  green_Tics = 200;
  initial
    red = off;
  initial
    amber = off;
  initial
    green = off;
  always begin
    red = on;
    light(red, red_tics);
    green = on;
    light(green, green_tics);
    amber = on;
    light(amber, amber_tics);
  end
```

```
  always begin
    red = on;
    light(red, red_tics);
    green = on;
    light(green,green_tics);
    amber = on;
    light(amber, amber_tics);
  end
  task light;
    output color;
    input [31:0] tics;
    begin
      repeat (tics)
        @(posedge clock);
      color = off;
    end
  endtask
  always begin
    #100 clock = 0;
    #100 clock = 1;
  end
endmodule
```

Task

- Examples:

```
module taskInternal();
        reg[1:0] a;
        reg b;
        reg[1:0] x;
        task test;
                input[1:0] a;
                begin
                        always@(a or b)begin //Error: can't have always block
                                x[0]=a+b;
                        end
                        $display("Hello World!");
                end
        endtask
        initial begin
                b=1;
                test(2);
                $display("x=%0b",x);
        end
endmodule
```

Diff Between Func. & Task

- Difference Between Function & Task

Function	Task
Function itself: reg by default	Task itself: can't declare data type
Function Name: can't be the same as global variable or other functions/Task	Task Name: can't be the same as global variable or other functions/Task
Function Value: X or retun value	Task value: no value
# of Input variables: at least 1	# of Input variables: 0 or 1 or more
scope of input variable: local	scope of input variable: local
input variable: register type, not timing control.	input variable: register type, not timing control.
internal variables: local or global in module	internal variables: local or global in module
LHS internal variables type: global in module=>reg	LHS internal variables type: global in module=>reg

Diff. Between Func. & Task

- Difference Between Function & Task

Function	Task
RHS internal variables type: global in module=>reg/wire	RHS internal variables type: global in module=>reg/wire
Statements: no timing control	Statements: timing control
Statements: =(blocking)	Statements:<=(non-blocking)
Statements: no "disable"	Statements: could have "disable"
Statements: can't call task, but function	Statements: call task/function
Simulation: zero	Simulation: non-zero
Synthesizable	Synthesizable(w/o delay)
No forever loop	No forever loop

Diff. Between Func. & Task

- Difference Between Function & Task

Function	Task
Modeling combo logic	Modeling combo/sequential logic
Invocation in expression	Single statement invocation
Passing value by value	Passing value by value
Can drive global reg variables	Can drive global reg variables
0 or 1 output	0 or 1 or more outputs
Static: multi-invocations=>share one local variable	Static: multi-invocations=>share one local variable
Automatic: multi-invocations=> more local variables & Recursive	Automatic: multi-invocations=>more local variables
Constant function	Constant task

Interesting Questions

- Why a function cannot call a task?
  - As functions does not consume time, it can do any operation which does not consume time. Mostly tasks are written which consumes time. So a task call inside a function blocks the further execution of function until it finished. But it's not true. A function can call task if the task call consumes zero time, but the IEEE LRM (Language Reference Manual) doesn't allow.
- Why tasks are not synthesized?
  - Wrong question! Tasks can be synthesized if it doesn't consume time.

Interesting Questions

- Why a function should return a value?
  - There is no strong reason for this in Verilog. This restriction is removed in SystemVerilog.
- Why a function should have at least one input?
  - There is no strong reason for this in verilog. This restriction is not removed in SystemVerilog. Some requirements where the inputs are taken from the global signal, those functions don't need any input. A work around is to use a dummy input.

Interesting Questions

- Why a task cannot return a value?
  - If tasks can return values, then Lets take a look at the following example.

    A=f1(B)+f2(C);

    and f1 and f2 had delays of say 5 and 10. When would B and C be sampled, or global variable inside f1 and f2 be sampled? How long does then entire statement block? This is going to put programmers in a bad situation. So languages gurus made that tasks can't return.
- Why a function cannot have delays?
  - The answer is same as above. But in OpenVera, delays are allowed in function. A function returns a value and therefore can be used as a part of any expression. This does not allow any delay in the function.

Interesting Questions

- Why disable statements are not allowed in functions?
  - If disable statement is used in function, it invalids the function and its return value. So disable statements are not allowed in function.