Sharif University of Technology
Department of Computer Engineering

Digital System Design
Tasks and Functions

Siavash Bayat-Sarmadi

Introduction

- Tasks and functions
  - Behavioral modelling
  - Same functionality, different places
    - Invoking routines instead repeating code
  - Can be addressed by hierarchical names

Functions Vs. Tasks

Functions

- Can invoke other functions but not other tasks
- Executed in 0 simulation time
  - No delay or timing control acceptable

Tasks

- Can invoke both other functions and tasks
- Can execute in non-zero simulation time
  - Delays and timing controls acceptable

Functions Vs. Tasks (Cont'd.)

Functions

- At least one input argument
  - inout not allowed
- Always return a single value
  - No output argument
- Non-blocking not allowed

Tasks

- May have zero or more input or inout arguments
- Do not return a value
  - output or inout arguments must be used when necessary
- Non-blocking allowed

Similarities

- Must be defined inside a module
  - Local to that module
- Behavioral statements only
  - E.g., no wires
- No always or initial blocks allowed inside

Functions

- Implicit actions
  - A variable with function name declared
    - Default: 1-bit reg
  - The value of this variable is returned
    - Placed where the function was invoked

Example 1: Parity Calculation


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

    always @(addr)
    begin
        //First invocation
        parity = calc_parity(addr);
        //Second invocation
        $display("Parity calculated = %b",
            calc_parity(addr) );
    end
```

Example 1: Parity Calculation (Cont'd.)


```
    function calc_parity;
    input [31:0] address;
    begin
        //Implicit internal reg
        calc_parity = ^address;
    end
    endfunction

endmodule
```

Example 1: Parity Calculation (Cont'd.)

- ANSI C-style declaration

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

Example 2: Left/Write 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
```

Example 2: Left/Write Shifter (Cont'd.)


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

Recursive Functions

ordinary functions

- Calls operate on the same variable space
- Two concurrent calls
  - Non-deterministic results

automatic functions

- Variables allocated separately for each call
- Recursive functions possible

Example: Factorial


```
module top;
    function automatic integer factorial;
    input [31:0] operand;
    begin
        if (operand >= 2)
            factorial = factorial
                (operand -1) * operand;
        else
            factorial = 1 ;
    end
    endfunction
```

Example: Factorial (Cont'd.)


```
    integer result;
    initial
    begin
            result = factorial(4);
            $display("Factorial of 4 is %0d",
                result);
    end
endmodule
```

Constant Functions

- Ordinary declaration and invocation
  - ‘constant’, no such keyword exists
- Can be used to compute a value at compile time
  - If all inputs are constants

Example: Computing ceil ($\log_2()$) at compile time


```
module ram (...);
    parameter RAM_DEPTH = 256;
    input [clogb2(RAM_DEPTH)-1:0] addr_bus;

    function integer clogb2(input integer depth);
    begin
        for(clogb2=0; depth >0; clogb2=clogb2+1)
            depth = depth >> 1;
    end
    endfunction
endmodule
```

Signed Functions

- Allowing signed operations on return values
  - E.g.

```
module top;
function signed [63:0]compute_signed
  (input [63:0] vector);
    …
endfunction
…
if(compute_signed(vector) < -3)
begin
  …
end
endmodule
```

Tasks

- More general than functions
- Arguments
  - input, output and inout
  - For passing values to/from tasks
  - Rather than connecting signals to modules

Example 1


```
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
```

Example 1 (Cont'd.)


```
    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
```

Example 1 (Cont'd.)

- ANSI C-style declaration

```
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
```

Example 2

- Directly operating on variables declared in module

```
module sequence;
    reg clock;

    initial
        init_sequence;
    always
        asymmetric_sequence;
```

Example 2 (Cont'd.)


```
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
```

Re-entrant Tasks

- Similar to the issue in recursive functions
  - All declared items shared across calls
  - Concurrent calls may lead to incorrect results
- Use atomatic for these cases
  - E.g.

```
task automatic bitwise_xor;
    …
endtask
```