Subroutines

• Verilog has two types of subroutines
  – Tasks
  – Functions
• Instances of modules are not considered subroutines, though the difference is more syntactical than practical.
User-Defined Tasks and Functions in Verilog

- Both **functions** and **tasks** allow you to execute common procedures from different places in a description.

- They help to simplify the complex behavior of the design by breaking up large procedures into smaller, more manageable ones.
Tasks and Functions

- **Tasks** and **Functions** cannot contain procedural blocks (always or initial).
- Tasks and Functions contain **behavioral statements** only.
- **Functions** are called from procedural blocks (always or initial), tasks, or other functions.
- **Tasks** are enabled from procedural blocks (always or initial), or other tasks (not from functions).
## Functions

A function is unable to enable a task however functions can enable other functions.

A function will carry out its required duty in zero simulation time. (The program time will not be incremented during the function routine)

Within a function, no event, delay or timing control statements are permitted

In the invocation of a function there must be at least one argument passed.

Functions will only return a single value and can not use either `output` or `inout` statements.

## Tasks

Tasks are capable of enabling a function as well as enabling other versions of a Task

Tasks also run with a zero simulation however they can if required be executed in a non zero simulation time.

Tasks are allowed to contain any of these statements.

A task is allowed to use zero or more arguments which are of type `output`, `input` or `inout`.

A Task is unable to return a value but has the facility to pass multiple values via the `output` and `inout` statements.
Verilog Task

Key features:

- **Task** is enabled when the task name is encountered in the Verilog description.

- **Task** definition is contained in the module definition.

- The arguments passed to the task are **in the same order** as the task I/O declarations.

- You can use **timing controls** in a task.

- Tasks define a **new scope** in Verilog.

- Tasks can be **disabled** the same way as disabling a named block.
Verilog Task

• A task is typically used to simplify the debugging operations, or to describe a separate piece of hardware.

• A task can contain delays.

• A task can have input, output, and inout parameters.
Tasks and Memory Allocation

• Variables in a task are assigned suitable memory locations on host computer.
• Concurrent calls to the task will all reference these same memory locations.
• This is all but certain to cause data corruption.
• The problem is static memory allocation.
Dynamic Allocation

• Verilog 2001 added dynamic allocation to tasks when new keyword *automatic* is used.

• Automatic variables do not share storage space: memory is allocated on the fly.

• Dynamic memory allocation is not synthesizable.

• Textbook Example 8-5 (pg. 177) has automatic task example.
module Task_Example;
parameter n = 8;
reg [n:1] x, y, z;
wire [n:1] w, v;

initial
  test_task (x, y, z, w, v);

task test_task;
  output [n:1] a, b;
  inout [n:1] c;
  input [n:1] d, e;
  reg [n:1] REG1, REG2, REG3;
  begin
    REG1 = Expre1 (c, d, e); REG2 = Expre2 (c, d, e); REG3 = Expre3 (c, d, e);
    a = REG1; b = REG2; c = REG3;
  end
endtask
//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
  //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);

//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 sequence;
reg clock;
initial     init_sequence;     //Invoke the task init_sequence
always
begin
    asymmetric_sequence;       //Invoke the task asymmetric_sequence
end

task init_sequence;       //Initialization sequence
begin
    clock = l'b0;
end
endtask

task asymmetric_sequence;
begin
    #12 clock = l'b1; #5 clock = l'b0;  #3 clock = l'b1;  #10 clock = l'b0;
end
endtask
endmodule
Parallel & Serial Execution

• A Verilog Task may execute at the same time as other assignments or it may block anything else from happening.
• It depends on how it’s called.
Clock Generator Task

module taskdemo;
    parameter WIDTH = 4;
    parameter PERIOD = 10;
    reg [WIDTH - 1 : 0] COUNT, DATA;
    reg CLK, RST, LOAD;

    always @(posedge CLK or negedge RST) begin
        if (!RST) COUNT <= 'b0;
        else begin
            if (LOAD) COUNT <= DATA;
            else COUNT <= COUNT + 1;
        end
    end
endmodule
task TOGGLE;
    input integer TIMES;
    output CLK;
    reg CLK;
    repeat (TIMES) begin
      #(PERIOD / 2) CLK <= 1'b0;
      #(PERIOD / 2) CLK <= 1'b1;
    end
endtask
Calling The Task

initial TOGGLE(12, CLK); //Parallel execution

initial begin
  //TOGGLE(12, CLK); DATA = 4'ha; //Serial execution
  RST = 1'b1; LOAD = 1'b0;
  #PERIOD RST = 1'b0;
  #(2*PERIOD) RST = 1'b1;
  #(2*PERIOD) LOAD = 1'b1;
  #PERIOD LOAD = 1'b0;
end
Parallel Execution Toggle
Serial (Blocking) Execution
Different, Both Bad

- Calls to tasks produce different results, neither of which is to toggle the clock connected to the counter.
- Problem is that calling procedure does not see task variable changes.
- Only sees update when tasks terminate.
Scoping Can Solve The Problem

module taskdemo;
  parameter WIDTH = 4;
  parameter PERIOD = 10;
  reg [WIDTH - 1 : 0] COUNT, DATA;
  reg CLK, RST, LOAD;
  reg POOH;

  always @(posedge TOGGLE.CLK or negedge RST) begin
    if (!RST) COUNT <= 'b0;
    else begin
      if (LOAD) COUNT <= DATA;
      else COUNT <= COUNT + 1;
    end
  end
end
Reference TOGGLE.CLK
Scope of Variables

• Problems in previous versions were due to scope of local variables.
• Variables declared in the task are local to the task.
• Variables declared in the calling module may also be seen/used in the task.
task TOGGLE;
   input integer TIMES; //output is gone
   repeat (TIMES) begin
      #(PERIOD / 2) CLK <= 1'b0;
      #(PERIOD / 2) CLK <= 1'b1;
   end
endtask
Task is Called Without Output

initial TOGGLE(12); //No output from task

initial begin
  RST = 1'b1; LOAD = 1'b0;
  #PERIOD RST = 1'b0;
  #(2*PERIOD) RST = 1'b1;
  #(2*PERIOD) LOAD = 1'b1; DATA = 4'ha;
  #PERIOD LOAD = 1'b0;
end
CLK Is Now Shared
Task Variables Summary

• Tasks may have inputs, outputs and local variables.

• A module variable will be used by the task if the task does not have a variable by that name.

• If it does have a variable by that name, the two will not be associated: task is a new level of hierarchy.
Blocking Task

• If task is in a list of assignments, it will not allow anything else to run until the task finishes.

• Use of non-blocking assignment operator will not help.

• Solution: run tasks in parallel
  – Separate initial blocks
  – In a fork…join structure
Parallel Blocks

initial TOGGLE(12); /*This way will update clock along with other variables*/

initial begin
   //TOGGLE(12); //This will block other assignments
   RST <= 1'b1; LOAD <= 1'b0;
   #PERIOD RST <= 1'b0;
   #(2*PERIOD) RST <= 1'b1;
   #(2*PERIOD) LOAD <= 1'b1; DATA <= 4'ha;
   #(2*PERIOD) LOAD <= 1'b0;
   end
Tasks, Delays and Synthesis

- Tasks may be used in synthesizable code, if they don’t contain non-synthesizable constructs.
- Tasks may include delays.
- Explicit delays (#5 A = B & C) are ignored by synthesizers.
- Implicit delays (@(posedge CLK)) are OK.
- Always blocks are not OK in tasks.
task BEAR;
    input CLK;
    output POOH;
    reg POOH;
    @(posedge CLK) POOH <= CLK;
endtask
Illegal Event/Delay

task BEAR;
    input CLK;
    output POOH;
    reg POOH;
    always @(posedge CLK) POOH <= CLK;
endtask
Why Does It Work That Way?

- Legal task has an implicit delay (wait for rising edge of clock), then does something.
- Illegal task has an implicit procedural block, would run forever.
- A task is not the same as a module. It may not contain functional (always, initial) blocks. It must return after some specified time or event.
module test(PAddr, PWrite, Psel, PRData, RST, CLK);
//Port declarations go here, omitted for slide legibility
initial begin
RST <= 1'b0;
#100 RST <= 1'b1;
@(posedge CLK)
    PAddr <= 16'h50;
PWData <= 32'h50;
PWrite <= 1'b1;
PSel <= 1'b1;
//Toggle Penable
 @(posedge CLK)     PEnable <= 1'b1;
 @(posedge CLK)     Penable <= 1'b0;
//Check the results
if (top.mem.memory[16'h50] == 32'h50)
    $display("Success");
else
    $display("Error, wrong value in memory");
$finish;
end
endmodule
task write(reg [15:0] addr, reg [31:0] data);

// Drive Control Bus
@ (posedge CLK)
    PAddr <= addr;
PData <= data;
PWrite <= 1'b1;
PSel <= 1'b1;

// Toggle Penable
@ (posedge CLK) PEnable <= 1'b1;
@ (posedge CLK) PEnable <= 1'b0;
endtask
module test(PAddr, PWrite, Psel, PRData, RST, CLK);
initial begin
    reset(); // task reset not shown
    write(16’h50, 32’h50); // input values for task write

    // Check the results
    if (top.mem.memory[16’h50] == 32’h50)
        $display(“Success”);
    else
        $display(“Error, wrong value in memory”);
$finish;
end
endmodule
Verilog Function

- A function is typically used to create a new operation, or to represent combinational logic.

- A function cannot contain any delays. Functions happen in zero simulation time.

- A function has only input parameters and returns a single value.
Verilog Function

Key features:

• A function definition cannot contain any timing-control statements.

• It must contain at least one input and does not contain any output or inout port.

• A function returns only one value, which is the value of the function itself.

• The arguments passed to the function are in the same order as the function input parameter declarations.

• The function definition must be contained within the module definition.

• A function cannot enable a task. However, a task can enable a function.

• Functions define a new scope in Verilog.
Function Definition

```plaintext
function [range] function_name;
  input_declaration
  other_declaration
  procedural_statements
endfunction
```

* A functions could be either a scalar or a vector. If no range is specified, then a 1-bit function is assumed.
module parity_gen;
    reg [31:0] addr;
    reg parity;
    //Compute new parity whenever address value changes
    always @(addr)
    begin
        parity = calc_parity(addr);   //First invocation of calc_parity
        $display("Parity calculated = %b", calc_parity(addr));
    end                                              //Second invocation of calc_parity end

    //define the parity calculation function
    function calc_parity;
        input [31:0] address;
        begin
            calc_parity = ^address;  //Returns the xor of all address bits.
        end   // Returns one for odd number of ones
    endfunction
endmodule
Functions In Expressions

• One advantage functions have over tasks is that functions may be used in expressions and tasks may not be.

• Example:

```
if (parity(DBUS[7:0]) != DBUS[8]) PFLG <= 1'b1;
```
Test Plans

• A test plan is an integral part of each lab report.
• You must state what tests you are running and why they are adequate to verify your design.
• See lab manual pages 5-6 and 12-16.
• Verification is typically a bigger job (3x bigger) than design.
Exam II

- Tuesday, November 10
- Open text.
- Covers all material from first day of class through the day of the exam.
- Bring a blue book.
- Don’t bring a telephone.
Homework 3

- Homework 3 is on the web site.
- No due date. Will not be collected or graded.
module cpu_iface (<ports>);
  // I/O declarations go here
reg [16:1] IR, PC, address;
event read_complete;

always @(posedge sys_clk)
begin
  if (read_request == 1)
    begin: read_block
      read_mem(IR, PC);
      ->read_complete;
      IR = swap_bits (IR) ;
    end

  // Task definition goes here

function [16:1] swap_bits;
  input [16:1] in_vec;
  integer i;
  for ( i = 0; i <= 15 ; i = i+1 )
    swap_bits[i+1] = in_vec[ 16 – i ] ;
endfunction

endmodule
Parameters and Functions

• There’s no way to override a parameter declared in a function.
• Thus one will act like a localparam.
• A parameter declared in the host module may be used by a function without re-declaring it in the function.
module testfunc(DATA, PARITY);
    parameter WIDTH = 32;
    input [WIDTH - 1 : 0] DATA;
    output reg PARITY;

    // #8 is a delay, not a parameter redefinition
    always @(DATA) PARITY = #(8) calc_parity(DATA);

    function calc_parity; // Uses module parameter
        input reg [WIDTH - 1 : 0] DATA; // Width is 32 here too
        calc_parity = ^DATA;
    endfunction

endmodule
module testfunc(DATA, PARITY);
  parameter WIDTH = 32;
  input [WIDTH - 1 : 0] DATA;
  output reg PARITY;

always @(DATA) PARITY = #(32) calc_parity(DATA);

//Local declaration of WIDTH will *not* be over-ridden
function calc_parity;
  parameter WIDTH = 8;
  input reg[WIDTH - 1 : 0] DATA; //Now WIDTH is 8.
  calc_parity = ^DATA; //Parity calculated only over 8 bits
endfunction
endmodule
Recursion

- Tasks and functions can be made recursive by use of the keyword “automatic.”
- Verilog 2001 enhancement.
- Synthesizer does not care either way about keyword “automatic.”
- That does not mean recursive functions are synthesizable. They are not and never will be.
Recursive Function Example

function automatic integer factorial;
input [31:0] operand;
  if (operand >= 2)
    factorial = factorial (operand * (operand – 1));
  else
    factorial = 1;
endfunction
Integers In Design Units

• Text example of automatic function uses an ill-defined integer for returned variable.
• Current synthesizer treats integers as signed 32 bit quantities. This was not always so and may not always be so in the future.
• Use vectors in circuit descriptions:
  – function automatic [31:0] factorial
• Still won’t be synthesizable, as recursion is inherently unsynthesizable.
Automatic Functions and Memory

• Standard Verilog functions allocate memory at compilation.
• Can not have multiple copies running concurrently: data from one would overwrite data from another.
• Automatic functions remove this restriction.
• This is a simulation concept. Memory refers to host computer memory. It has no meaning for circuit design.
Practical Functions

• Examples in the book are all too simple to be at all useful. They are all more trouble than they are worth, as the actual function is no longer than the call to the function.

• Functions really do have uses though. For example, Verilog has no built-in trig functions.
Sine Function, Abbreviated Version

function real sine;
    input x;
    real x;
    real y, y2, y3, y5, y7;
begin
    y = x*2/3.14159;
y2 = y*y;
y3 = y*y2;
y5 = y3*y2;
y7 = y5*y2;
sine = 1.570794*y - 0.261799*y3 + 0.0130899*y5 - 0.000311655*y7;
end
endfunction

*Does not handle sign. Complete version does.
Use sin function to make cos

function real cos;
    input x;
    real x;
    begin
        cos = sin(x + 3.14159265/2.0);
    end
endfunction
Followed by tan

function real tan;
    input x;
    real x;
    begin
        tan = sin(x)/cos(x);
    end
endfunction
Square Root Function

function real rootof2;
  input n;
  integer n;
  real power;
  integer i;

begin
  power = 0.82629586;
  power = power / 10000000.0;
  power = power + 1.0;
  i = -23; // 23 bits

  if (n >= 1)
    begin
      power = 2.0;
      i = 0;
    end
    for (i = i; i < n; i = i + 1)
      begin
        power = power * power;
        end
  rootof2 = power;
end
endfunction
Power Operator

- Verilog 2001 also adds a power operator, **
- Example:
  ```verilog
  always @(posedge CLOCK)
  RESULT <= BASE ** EXPONENT;
  ```
Generate

• Verilog 2001 has also added a Generate construct.
• Can be used to selectively generate logic elements.
• Theoretically useful for synthesis, maybe it will be some day.
• Now supported for synthesis.
module multiplier (a, b, product);
  parameter a_width = 8, b_width = 8; 
  localparam product_width = a_width + b_width;
  input [a_width-1:0]    a;
  input [b_width-1:0]    b;
  output [product_width-1:0] product;

  generate
    if((a_width < 8) || (b_width < 8))
      CLA_multiplier #(a_width, b_width)
        ul (a, b, product);
    else
      WALLACE_multiplier #(a_width, b_width)
        ul (a, b, product);
  endgenerate
endmodule
module Nbit_adder (co, sum, a, b, ci);
    parameter SIZE = 4;
    output [SIZE-1:0] sum;
    output co;
    input [SIZE-1:0] a, b;
    input ci;
    wire [SIZE:0] c;

    genvar i;
    assign c[0] = ci;
    assign co = c[SIZE];

    generate
        for(i=0; i<SIZE; i=i+1)
            begin:addbit
                wire n1,n2,n3;    //internal nets
                xor g1 ( n1, a[i], b[i]);
                xor g2 (sum[i], n1,  c[i]);
                and g3 ( n2, a[i], b[i]);
                and g4 ( n3, n1,   c[i]);
                or  g5 (c[i+1], n2, n3);
            end
    endgenerate
endmodule
Question:
Is there an easy way of creating multiple objects within a module in Verilog? Is this supported in HDL Compiler (Presto Verilog)?

Answer:
Beginning with Design Compiler version U-2003.03, generate statements are supported by Presto Verilog as defined by the IEEE Verilog Standard, 1364-2001. The Verilog generate statement is similar to the VHDL generate statement and can be used to create multiple objects within a module. Most objects (for example, always blocks, module instantiations, nets, variables, and functions tasks) can be placed inside generate blocks. The Verilog generate statements provide concise and readable code. The following sample code is for gray to binary conversion with and without generate statements.
Verilog 1995 (without generate statement)

assign bin[7] = gray[7];
...
assign bin[0] = bin[1] ^ gray[0];
Verilog 2001 (with generate statement)
---------------------------------------
generate
    genvar i;
    assign bin[7] = gray[7];
    for (i=6; i>=0; i=i-1)
        begin: gray_to_bin_conv
            assign bin[i] = bin[i+1] ^ gray [i];
        end
endgenerate
Behavioral Code and Generate

- Generate statements are a great way to make a large number of instances.
- They are a way to make an indeterminate (until compile time) number of instances.
- Could be done with parameter and array of instances, too.
- They are not terribly useful when what goes in the generate loop is nothing but behavioral statements.
A Gray to Binary Converter

Note: this is a serial (ripple) process, much slower than B to G conversion.
module gray2bin(bin, gray);

    parameter width = 8;
    input [width - 1 : 0] gray;
    output [width - 1 : 0] bin;
    reg [width - 1 : 0] bin;
    integer i;

    always @(gray) begin
        bin[width - 1] = gray[width - 1];
        for (i = width - 2; i >= 0; i = i - 1)
            bin[i] = gray[i] ^ bin[i + 1];
    end
endmodule

Same function, n-bit scalable, but no need to use a generate loop.
module tb_gray2bin();

    parameter width = 8;
    reg [width - 1 : 0] gray;
    wire [width - 1 : 0] bin;
    reg [width - 1 : 0] count;
    integer i, j;

    gray2bin  uut(bin, gray);

    initial begin
        for (i = 0; i <= 255; i = i + 1)
            #10 count = i;
    end

    always @(count) begin
        for (j = 0; j <= width - 2; j = j + 1)
            gray[j] = count[j] ^ count[j + 1];
    end
endmodule
Synthesized 8-bit G2B
More Verilog 2001 Confusion

Continuous Assignment:

wire xbit = 1'b0; xbit will always be driven to 0

Verilog 2001 reg initialization:

reg xbit = 1'b0

Is equivalent to:

reg xbit;

initial xbit = 1'b0;

Obviously (?) this is only for simulation and will never be for synthesis.