UMBC

AN HONORS UNIVERSITY IN MARYLAND

Department of Computer Science and Electrical Engineering

CMPE 415

Parameterized Modules and Simulation

Prof. Ryan Robucci

Parameters

Modules may include parameters that can be overridden for tuning behavior and facilitating code reuse

```
module pdff(q,d,clk,arst);
  parameter size=1, resetval=0;
  output [size-1:0] q;
  input [size-1:0] d;
  input  clk,arst;


  always @(posedge clk, posedge arst)
    if (arst)
      q<=resetval;
    else
      q <= d;
endmodule
```

Overriding the parameters in the module instantiation is optional and may be provided by an ordered list or named list:

```
pdff I1 (.q(ur1q),
         .d(u1d),
         .clk(clk),.arst(rst));

pdff #(.size(8),.resetval(128)) I2 (.q(ur8q),
                                    .d(u8d),
                                    .clk(clk),
                                    .arst(rst));

pdff #(8,128) I3 (.q(ur8q),
                  .d(u8d),
                  .clk(clk),
                  .arst(rst));

pdff #(.size( ),.resetval(1) ) I4 (.q(ur1q),
                                   .d(u1d),
                                   .clk(clk),.arst(rst));
```

Overriding the parameters can also be done with the keyword **defparam** and the hierarchical name.

```
pdff I2 (.q(ur8q), .d(u8d), .clk(clk), .arst(rst));
defparam  I2.size = 8;
defparam  I2.resetval = 128;
```

This can also be useful for overriding parameters for simulation from the testbench:

```
defparam topDUT.comModule1.uart2.BAUD_RATE = 100;
```

Local parameters cannot be overridden directly from the instantiation, but they can depend on other parameters. Use keyword **localparam**

```
module pdff(q,d,clk,arst);
  parameter size=1, resetval=0;
  localparam msb=size-1;
  output [msb:0] q;
  input [msb:0] d;
  input  clk,arst;
  reg [msb:0] out;

  always @(posedge clk, posedge arst)
    if (arst)
      q<=resetval;
    else
      q <= d;
endmodule
```

Parametrized Simulation Models

Parametrization modules are useful for creating simulation for simulation that takes much less time to simulate, involves minimal differences from the code used for hardware, and maintains high functional test coverage.

Ex 1: delay, size

Module:

```
module ROM(q,addr);
  parameter delay = 1000;
  parameter memorysize = 1024*1024*1024;
  wire #delay = q_int;
```

…

In testbench:

```
ROM (10,1024) I1(q,addr)
```

Ex 2: cycle counts

```
module clk(strobe_out,addr,clk);
  parameter cycles = 1024*1024;
  …
  if (counter == cycles) begin
   strobe_out<=1; counter<=0;
  end else begin
   strobe_out<=0; counter=counter+1;
```

Parametrized Simulation Models

- You should always consider the options to accelerate simulation without compromising the integrity of the simulation
- A common example of modified models for simulation is an asynchronous communication module called a UART Universal Asynchronous Receiver Transmitter.
  - The baud rate is set by parameter for generality (easily implement multi rates for synthesis) and to aid simulation.

```
module uart(…);
…
// clock rate (50Mhz) / (baud rate (9600) * 4)
parameter CLOCK_DIVIDER = 1302;
…
endmodule
```

Consider the significance:

Simulating a 1kB transfer of data would require over 400 Millon Clock cycles! For complex systems this can be prohibitive or impossible, and managing the output data is difficult.

A properly coded module could allow the divider to be set to a smaller divider value such as 8 for simulation

Functions and Tasks

Bulleted points, text, examples, and tables
taken from book: The Verilog Hardware Description Language

- Like modules, functions and tasks allow us to break up a design into more-manageable parts, but unlike modules do not imply structural connections.
- Avoid unnecessary code repetition
- Easier to update fewer task and functions then many similar pieces of code
- Increase code re-ability, no need to clutter high-level code with low-level details

Functions and Tasks

- Tasks are similar to macros in C, and are used in procedural code.
- Functions are used to return a single result based on input variables and may be used in procedural code and continuous assignments.

2.5.1 Tasks

A Verilog *task* is similar to a software procedure. It is called from a calling statement and after execution, returns to the next statement. It cannot be used in an expression. Parameters may be passed to it and results returned. Local variables may be declared within it and their scope will be the task. Example 2.9 illustrates how module Mark-1 could be rewritten using a task to describe a multiply algorithm.

A task is defined within a module using the *task* and *endtask* keywords. This task is named **multiply** and is defined to have one inout (**a**) and one input (**b**). This task is called from within the always statement. The order of task parameters at the calling site must correspond to the order of definitions within the task. When **multiply** is called, **acc** is copied into task variable **a**, the value read from memory is copied into **b**, and the task proceeds. When the task is ready to return, **prod** is loaded into **a**. On return, **a** is then copied back into **acc** and execution continues after the task call site.

Task Example

```
module mark1Task;
reg [31:0]      m [0:8191];   // 8192 x 32 bit memory
reg [12:0]      pc;           // 13 bit program counter
reg [31:0]      acc;          // 32 bit accumulator

    always
        begin: executeInstructions
            reg  [15:0]     ir;   // 16 bit instruction register

            ir = m [pc];
            case (ir [15:13])
                // other case expressions as before
                3'b111 :   multiply (acc, m [ir [12:0]]);
            endcase
            #1 pc = pc + 1;
        end

    task multiply;
    inout    [31:0]  a;
    input    [31:0]  b;

        begin: serialMult
            reg      [15:0] mcnd, mpy; //multiplicand and multiplier
            reg      [31:0] prod;      //product

            mpy = b[15:0];
            mcnd = a[15:0];
            prod = 0;
            repeat (16)
                begin
                    if (mpy[0])
                        prod = prod + {mcnd, 16'h0000};
                    prod = prod >> 1;
                    mpy = mpy >> 1;
                end
            a = prod;
        end
    endtask
endmodule
```

Start of definition of task

Aliases to use inside "macro" as inputs and outputs

Variables to operate on/with

End of definition

Functions

2.5.2 Functions

A Verilog function is similar to a software function. It is called from within an expression and the value it returns will be used in the expression. The function has one output (the function name) and at least one input. Other identifiers may be declared within the function and their scope will also be the function. Unlike a task, a function may not include delay(#) or event control (@, wait) statements. Although not illustrated here, a function may be called from within a continuous assignment. Functions may call other functions but not other tasks. During the execution of the function, a value must be assigned to the function name; this is the value returned by the function.

Function Example

```
module mark1Fun;
reg [31:0]      m [0:8191];    // 8192 x 32 bit memory
reg [12:0]      pc;            // 13 bit program counter
reg [31:0]      acc;           // 32 bit accumulator

always
   begin: executeInstructions
         reg  [15:0]     ir;   // 16 bit instruction register

         ir = m [pc];
         case (ir [15:13])
              //case expressions, as before
              3'b111:    acc = multiply(acc, m [ir [12:0]]);
         endcase
         #1 pc = pc + 1;
   end

function [31:0] multiply;
input    [31:0]  a;
input    [31:0]  b;

   begin: serialMult
         reg      [15:0]      mcnd, mpy;

         mpy = b[15:0];
         mcnd = a[15:0];
         multiply = 0;
         repeat (16)
              begin
                   if (mpy[0])
                        multiply = multiply + {mcnd, 16'h0000};
                   multiply = multiply >> 1;
                   mpy = mpy >> 1;
              end
   end
endfunction
endmodule
```

Start definition

Aliases of input values to reference inside function

1 or more values to pass into function

Single-variable output is referenced using the name of the function

End definition

Functions

- Comb. Only…represent instantaneous calculation in simulation
- Single output, may not directly manipulate several values
  - Though you could hack this by packing several outputs into a single output
- No timing (delay, events, or wait)
- May not invoke a task
- May call other functions, but not recursively
- Return a single value, no declared output or inout ports
- Must have 1 or more inputs and be invoked with inputs

Details and Comparison of Tasks and Functions

Table 2.1 Comparison of Tasks and Function

Category	Tasks	Functions
Enabling (calling)	A task call is a separate procedural statement. It cannot be called from a continuous assignment statement.	A function call is an operand in an expression. It is called from within the expression and returns a value used in the expression. Functions may be called from within procedural and continuous assignment statements.
Inputs and outputs	A task can zero or more arguments of any type.	A function has at least one input. It does not have inouts our outputs.
Timing and event controls (#, @, and wait)	A task can contain timing and event control statements	Functions may not contain these statements
Enabling other tasks and functions	A task may enable other tasks and functions	A function can enable other functions but not other tasks.
Values returned	A task does not return a value. However, values written into its inout or output ports are copied back at the end of the task execution.	A function returns a single value to the expression that called it. The value to be returned is assigned to the function identifier within the function.

Procedural Timing Controls

- Delay control: delay between encountering the expression and when it executes.
  - Introduced by simple #
- Event control: delay until event
  - Explicit Events are named events that allow triggering from other procedures
    - A named event may be declared using the keyword **event**
    - The event control operator @ can be used to hold procedural code execution
    - operator -> triggers the event

```
event triggerName;
```

```
@(triggerName);
```

```
-> triggerName;
```

  - Implicit events are responses to changes in variables
    - The event control operator @ can be provided a sensitivity list with variables and on optional event selectivity using keywords **posedge** and **negedge**

Event Control Operator

The event control operator may be provided with variable multiple variables using comma separation. (older syntax is to us **or**)

@(a,b,c) @(a or b or c)

Negedge and posedge restrict sensitivity to the following transitions

@(**posedge** clk or **negedge** edge en)

Negedge: $1 \rightarrow 0$
$1 \rightarrow$ x or z
x or z $\rightarrow 0$

Posedge: $0 \rightarrow 1$
x or z $\rightarrow 1$
$0 \rightarrow$ x or z

When posedge and negedge modify a multi-bit operand, only the lsb is used to detect the edge

Level-sensitive event control using **wait**

- The wait statement suspends execution until a condition is true.
- It can be considered as a **level-sensitive** control since it doesn't wait for a transition edge.

`@(posedge clk)` if clk is already true, wait for next rising edge

`wait (clk);` if clk is already true, produce without delay

Example to change data on the falling clock edge that follows a variable exceeding the value 10

```
wait(a>10);
@ (negedge clk);
data=data+1;
```

Repeat

Any timing control may be modified so as to be repeated/multiplied, by using the keyword **repeat**

```
repeat (count) @ (event expression)
```

If count is positive, repeat the timing control that number of time. If count is equal or less than 0, skip

Wait fot 10 clk rising edges before proceeding execution:

```
repeat (10) @ (posedge clk);
```

Delay an assignment by 5 clk edges

```
a <= repeat(5) @(posedge clk) data;
```

initial and always

The **initial** construct is used to denote code to be executed once at the beginning of the simulation.

The **always** construct causes code to run repeatedly in an infinite loop. It is only useful with a delay or control construct, otherwise will create a zero delay infinite loop that can block time progression in simulation

```
always x=a&b;
```
This would run infinitely

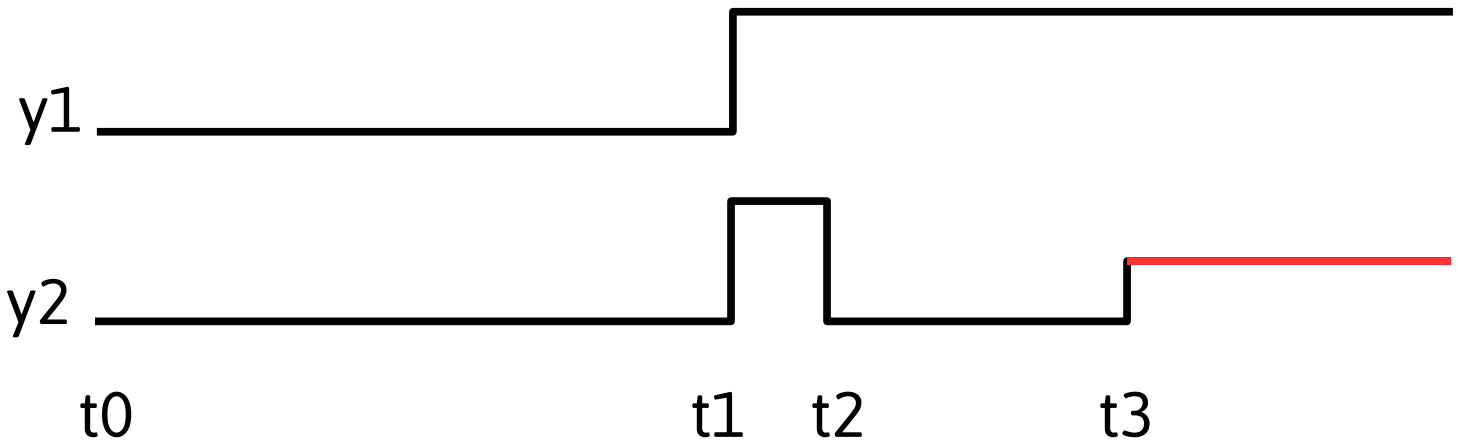
This prints endlessly in the beginning of the simulation:

```
always begin
  $display("hello %0t");
end
```

```
0 : hello
0 : hello
0 : hello
0 : hello
0 : hello ...
```

Value Change Dumpfile

- Rather than store values of several variables at EVERY timestep, only record changes.
- Each entry is a time along with a list of changes



Time	Value Changes
0	y1=0, y2=0
73ns	y1=1, y2=1
75ns	y2=0
90ns	y2=x

Value Change Dumpfile

Ref http://www.verilog.renerta.com/mobile/source/vrg00056.htm

- Commands: $**dumpfile**, $**dumpvar**, $**dumpon**, $**dumpoff**, $**dumpall**
- Syntax Examples
  - $**dumpfile**("filename.vcd")
  - $**dumpvars**
    - selects all variables in the design
  - $**dumpvar**(0, top)
    - dumps all the variables in module top and all levels in the hierarchy below
  - $**dumpvar**(n, top)
    - selects all the variables in module top and ,skipping n levels below, selects all variables at level n+1 and below
  - $**dumpvar**(varname1,varname2,....)
    - selects individual variables based on name. Allows hierarchical names e.g. DUT.I1.process1.u
  - $**dumpvar**(n,top,varname1, varname1,....)
    - selects individual variables based on name in addition to all variables in top and and n+1 levels below

More VCD Control

Ref http://www.verilog.renerta.com/mobile/source/vrg00056.htm

- $**dumpall**
  - creates a checkpoint with the value of all selected variables
- $**dumpoff**
  - stop dumping and create a checkpoint in the file with all variables dumped as x for that time
- $**dumpon**
  - resumes dumping and creates checkpoint in the file with all variables dumped
- $**dumpflush**
  - updates the dumpfile allowing other processes to read updated contents

VCD command usage example

Ref http://www.verilog.renerta.com/mobile/source/vrg00056.htm

```
initial …
$dumpfile("test.txt")
;
   $dumpvars(1,a,y);
   #200;
   $dumpoff;
   #200;
   $dumpon;
   #20;
   $dumpall;
   #10;
   $dumpflush;
```

The dumpfile will contain only changes of 'a' and 'y' variables. After 200 time units, dumping will be suspended for 200 time units. Next, dumping will start again and after 20 time units, all variables will be dumped.

VCD command usage example 2

```
module dump;
  event do_dump; //declare event that can trigger dumping
  initial $dumpfile("verilog.dump");
  initial @do_dump $dumpvars; //dump variables in the design
  always @do_dump begin //to begin the dump at event do_dump
    $dumpon;
    //no effect the first time through
    repeat (500) @(posedge clock); //dump for 500 cycles
    $dumpoff;
    //stop the dump
  end

  //regularly report ALL variables
  initial @(do_dump) forever #10000 $dumpall;
endmodule
```