Parameters and Generate Blocks

Objectives of this topic

- Parameter declaration and use
- How to *dynamically* generate Verilog code

Parameters

- Similar to `const` in C
  - But can be overridden for each module at compile-time
- Syntax:

```
parameter <const_id> = <value>;
```

- Gives flexibility
  - Allows to customize the module
- Example:

```
parameter port_id = 5;
parameter cache_line_width = 256;
parameter bus_width = 8;
parameter signed [15:0] WIDTH;

wire [bus_width-1:0] bus;
```


Parameter Example

```
module hello_world;
  parameter id_num = 0;

  initial
       $display("Displaying hello_world id number = %d", id_num);
endmodule

//define top-level module
module top;
  defparam w1.id_num = 1, w2.id_num = 2;
  hello_world w1();
  hello_world w2();
endmodule
```


Better Coding Style

```
module hello_world;
  parameter id_num = 0;

  initial
       $display("Displaying
   hello_world id num = %d",
   id_num);
endmodule

module top;
  defparam w1.id_num = 1,
     w2.id_num = 2;
  hello_world w1();
  hello_world w2();
endmodule
```

```
module hello_world
  #(parameter id_num = 0);

   initial
      $display("Displaying
   hello_world id num = %d",
   id_num);
endmodule


module top;
  hello_world #(1) w1();
  hello_world #(.id_num(2)) w2();
endmodule
```


Parameters (cont'd)

- `localparam` keyword

```
localparam state1 = 4'b0001,
           state2 = 4'b0010,
           state3 = 4'b0100,
           state4 = 4'b1000;
```

Operations for HDL simulation

- Compilation/Parsing
- Elaboration
  - Binding modules to instances
  - Build hierarchy
  - Compute parameter values
  - Resolve hierarchical names
  - Establish net connectivity
- Simulation

Generate Block

- Dynamically generate Verilog code at elaboration time
  - Usage:
    - Parameterized modules when the parameter value determines the module contents
  - Can generate
    - Modules
    - User defined primitives
    - Verilog gate primitives
    - Continuous assignments
    - `initial` and `always` blocks

Generate Loop

```
module bitwise_xor #(parameter N = 32)(output [N-1:0] out, input
   [N-1:0] i0, i1);
  genvar j; // This variable does not  exist during simulation

  generate for (j=0; j<N; j=j+1) begin: xor_loop
       //Generate the bit-wise Xor with a single loop
     xor g1 (out[j], i0[j], i1[j]);
    end
  endgenerate //end of the generate block

/* An alternate style using always blocks:
 reg [N-1:0] out;
 generate for (j=0; j<N; j=j+1) begin: bit
     always @(i0[j] or i1[j]) out[j] = i0[j] ^ i1[j];
   end
 endgenerate
endmodule  */
```

xor_loop is the name given to this begin-end block

If N=3, the instance names will be
xor_loop[0].g1
xor_loop[1].g1
xor_loop[2].g1

Example 2: Ripple Carry Adder

```
module ripple_adder #(parameter N=4) (output co, output [N-1:0] sum, input
   [N-1:0] a0, a1, input ci);
   wire [N:0] carry;
   assign carry[0] = ci;

   genvar i;
   generate for (i=0; i<N; i=i+1) begin: r_loop
        wire t1, t2, t3;
        xor g1 (t1, a0[i], a1[i]);
        xor g2 (sum[i], t1, carry[i]);
        and g3 (t2, a0[i], a1[i]);
        and g4 (t3, t1, carry[i]);
        or  g5 (carry[i+1], t2, t3);
    end
   endgenerate //end of the generate block
   assign co = carry[N];
endmodule
```

Note: hierarchical instance names

Generate Conditional

```
module multiplier (output [product_width -1:0] product, input [a0_width-1:0] a0,
          input [a1_width-1:0] a1);
    parameter       a0_width = 8;
    parameter       a1_width = 8;

    localparam      product_width = a0_width + a1_width;

    generate
          if (a0_width <8) || (a1_width < 8)
                cla_multiplier  #(a0_width, a1_width)   m0 (product, a0, a1);
          else
                tree_multiplier  #(a0_width, a1_width)   m0 (product, a0, a1);
    endgenerate

endmodule
```


Generate Case

```
module adder (output co, output [N-1:0] sum,
       input [N-1:0] a0, a1, input ci);

    parameter N = 4;

    // Parameter N that can be redefined at instantiation time.
    generate
          case (N)
                1:        adder_1bit        adder1(c0, sum, a0, a1, ci);
                2:        adder_2bit        adder2(c0, sum, a0, a1, ci);
                default: adder_cla   #(N)   adder3(c0, sum, a0, a1, ci);
          endcase
    endgenerate

endmodule
```

Nesting

- Generate blocks can be nested
  - Nested loops cannot use the same `genvar` variable

Some System Tasks and Compiler Directives

Compiler Directives

- Instructions to the *Compiler* (not *simulator*)
- General syntax:
  `` `<keyword> ``
- `` `define ``
  - similar to `#define` in C
  - `` `<macro_name> `` to use the macro defined by `` `define ``
- Examples:

```
`define WORD_SIZE 32
`define S $stop

`define WORD_REG reg [31:0]
`WORD_REG a_32_bit_reg;
```


Compiler Directives (cont'd)

- `` `undef ``
  - `Undefine a macro`
- Example:

```
`undef BUS_WIDTH
```

- `` `include ``
  - Similar to `#include` in C
- Example:

```
`include header.v
...
<Verilog code in file design.v>
...
```


Compiler Directives (cont'd)

- `` `ifdef ``, `` `ifndef ``, `` `else ``, `` `elsif ``, and `` `endif ``.

```
`define TEST
`ifdef TEST //compile module test only if macro TEST is defined
   module test;
        ...
        ...
   endmodule
`else //compile the module stimulus as default
   module stimulus;
        ...
        ...
   endmodule
`endif //completion of 'ifdef directive
```


System Tasks

- System Tasks: standard routine operations provided by Verilog
  - Displaying on screen, monitoring values, stopping and finishing simulation, etc.
- All start with `$`
- Instructions for the *simulator*

System Tasks (cont'd)

- `$display`: displays values of variables, strings, expressions.

```
$display(p1, p2, p3, …, pn);
```

  - `p1,…, pn` can be quoted string, variable, or expression
  - Adds a new-line after displaying `pn` by default
  - Format specifiers:
    - `%d, %b, %h, %o`: display variable respectively in decimal, binary, hex, octal
    - `%c, %s`: display character, string
    - `%e, %f, %g`: display real variable in scientific, decimal, or whichever smaller notation
    - `%v`: display strength
    - `%t`: display in current time format
    - `%m`: display hierarchical name of this module

`$display` examples

```
$display("Hello Verilog World!");
```
Output: `Hello Verilog World!`

```
$display($time);
```
Output: `230`

```
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h",
  $time, virtual_addr);
```
Output: `At time 200 virtual address is 1fe000001c`

`$display` examples (cont'd)

```
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
```
Output: `ID of the port is 00101`

```
reg [3:0] bus;
$display("Bus value is %b", bus);
```
Output: `Bus value is 10xx`

```
$display("Hierarchical name of this module is %m");
```
Output: `Hierarchical name of this module is top.p1`

```
$display("A \n multiline string with a %% sign.");
```
Output:
```
A
      multiline string with a % sign.
```


`$monitor` System Task

- `$monitor`: monitors signal(s) and displays them when their value changes

```
$monitor(p1, p2, p3, …, pn);
```

  - `p1,…, pn` can be quoted string, variable, or signal names
  - Format specifiers similar to `$display`
  - Continuously monitors the values of the specified variables or signals, and displays the entire list whenever any of them changes.
  - `$monitor` needs to be invoked only once (unlike `$display`)
    - Only one `$monitor` (the latest one) can be active at any time
    - **`$monitoroff`** to temporarily turn off monitoring
    - **`$monitoron`** to turn monitoring on again

`$monitor` Examples

```
initial
  $monitor($time, "Value of signals clock=%b,
     reset=%b", clock, reset);
initial
  begin
   clock=0;
   reset=1;
   #5 clock=1;
   #10 clock=0; reset=0;
  end
```

- Output:

```
0 value of signals clock=0, reset=1
5 value of signals clock=1, reset=1
15 value of signals clock=0, reset=0
```


`$stop` System Task

- `$stop`: stops simulation
  - Simulation enters interactive mode
  - Most useful for debugging
- `$finish`: terminates simulation

- Examples:

```
initial
begin
  clock=0;
  reset=1;
  #100 $stop;
  #900 $finish;
end
```

Useful System Tasks: File I/O

Useful Modeling Techniques

Opening a File

- Opening a file

```
<file_handle> = $fopen( "<file_name>" );
```

  - `<file_handle>` is a 32 bit value, called *multi-channel descriptor*
  - Only 1 bit is set in each descriptor
  - Standard output has a descriptor of 1 (Channel 0)

```
//Multichannel descriptor
integer handle1, handle2, handle3; //integers are 32-bit values

//standard output is open; descriptor = 32'h0000_0001 (bit 0 set)
initial
begin
   handle1 = $fopen("file1.out"); //handle1=32'h0000_0002 (bit 1 set)
   handle2 = $fopen("file2.out"); //handle2=32'h0000_0004 (bit 2 set)
   handle3 = $fopen("file3.out"); //handle3=32'h0000_0008 (bit 3 set)
end
```


File Output and Closing

- Writing to files
  - `$fdisplay, $fmonitor, $fstrobe`
  - `$strobe, $fstrobe`
    - The same as `$display, $fdisplay`, but executed after all other statements schedule in the same simulation time
  - Syntax:

```
$fdisplay(<handle>, p1, p2,…, pn);
```

- Closing files

```
$fclose(<handle>);
```


Example: Simultaneously writing to multiple files

```
//All handles defined in Example 9-7
//Writing to files
integer desc1, desc2, desc3; //three file descriptors
initial
begin
   desc1 = handle1 | 1; //bitwise or; desc1 = 32'h0000_0003
   $fdisplay(desc1, "Display 1");//write to files file1.out & stdout

   desc2 = handle2 | handle1; //desc2 = 32'h0000_0006
   $fdisplay(desc2, "Display 2");//write to files file1.out & file2.out

   desc3 = handle3 ; //desc3 = 32'h0000_0008
   $fdisplay(desc3, "Display 3");//write to file file3.out only
end
```


Random Number Generation

- Syntax:

```
$random;
$random(<seed>);
```

- Returns a 32 bit random value

```
//Generate random numbers and apply them to a simple ROM
module test;
integer r_seed;
reg [31:0] addr;//input to ROM
wire [31:0] data;//output from ROM
…
…
ROM rom1(data, addr);

initial
   r_seed = 2; //arbitrarily define the seed as 2.

always @(posedge clock)
   addr = $random(r_seed); //generates random numbers

…
<check output of ROM against expected results>
…
…
endmodule
```


Useful System Tasks Initializing Memory from File

- Keywords:
  - `$readmemb, $readmemh`
- Used to initialize memory (`reg [3:0] mem[0:1023]`)
- Syntax:

```
$readmemb("<file_name>", <memory_name>);
$readmemb("<file_name>", <memory_name>, <start_addr>);
$readmemb("<file_name>", <memory_name>, <start_addr>,
   <finish_addr>);
```

- The same syntax for **`$readmemh`**

```
module test;

reg [7:0] memory[0:7]; //declare an 8-byte memory
integer i;

initial
begin
  //read memory file init.dat. address locations given in memory
  $readmemb("init.dat", memory);
  //display contents of initialized memory
  for(i=0; i < 8; i = i + 1)
    $display("Memory [%0d] = %b", i, memory[i]);
end

endmodule
```

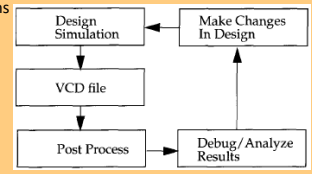
```
@02
11111111 01010101
00000000 10101010

@06
1111zzzz 00001111
```

```
Memory [0] = xxxxxxxx
Memory [1] = xxxxxxxx
Memory [2] = 11111111
Memory [3] = 01010101
Memory [4] = 00000000
Memory [5] = 10101010
Memory [6] = 1111zzzz
Memory [7] = 00001111
```

31

Useful System Tasks
Value Change Dump (VCD) File

- ASCII file containing information on
  - Simulation time
  - Scope and signal definitions
  - Signal value changes
- Keywords
  - `$dumpvars`
  - `$dumpfile`
  - `$dumpon`
  - `$dumpoff`
  - `$dumpall`




```
//specify name of VCD file. Otherwise,default name is
//assigned by the simulator.
initial
      $dumpfile("myfile.dmp"); //Simulation info dumped to myfile.dmp

//Dump signals in a module
initial
   $dumpvars; //no arguments, dump all signals in the design
initial
   $dumpvars(1, top); //dump variables in module instance top.
```

```
              //Number 1 indicates levels of hierarchy. Dump one
             //hierarchy level below top,i.e.,dump variables in top,
            //but not signals in modules instantiated by top.
initial
   $dumpvars(2, top.m1);//dump up to 2 levels of hierarchy below top.m1
initial
   $dumpvars(0, top.m1);//Number 0 means dump the entire hierarchy
                        // below top.m1

//Start and stop dump process
initial
begin
   $dumpon;            //start the dump process.
   #100000 $dumpoff;  //stop the dump process after 100,000 time units
end

//Create a checkpoint. Dump current value of all VCD variables
initial
   $dumpall;
```

33