ECE 351
Verilog and FPGA Design

Week 8_1: SystemVerilog functions and tasks
Review Roy's midterm exam solution

Roy Kravitz
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science

Questions about Homework #3?

Write-up: \ece351sp21_hw3_release_r1_0\docs\hw3_write_up.pdf

Functions and tasks

Sources:

- Mark F. and Roy's lecture material from other courses

Functions and tasks

- SystemVerilog equivalent to programming language subroutines (Tasks) and functions (Functions)
  - Used to implement the same functionality at many places…or to structure/organize the code
  - Allows the designers to abstract SystemVerilog code (ex: breaking large behavioral designs into smaller pieces)
- Value passing
  - Tasks and functions can have `input`, `output` and `inout` arguments
- Tasks and functions are included in the design hierarchy
  - Can be addressed by hierarchical names like anything else in the hierarchy
- Tasks and functions are synthesizable and result in hardware for every place they are invoked

Functions and tasks (cont'd)

Functions	Tasks
Can invoke another function but not another task	Can invoke other tasks and functions
Execute in 0 simulation time	May execute in non-zero simulation time
Must not contain any delay, event, or timing control statements	May contain delay, event, or timing control statements
May have zero or more arguments of type `input`, `output`, or `inout`	May have zero or more arguments of type `input`, `output`, or `inout`
May return a single value or no value (void)	Do not return a value, but can pass multiple values though output or inout arguments

Functions

```
always @(posedge clock)
  result <= divide(b, a);

function int divide (input int numerator,
                                denominator);
  if (denominator == 0) begin
    $display("Error! divide by zero");
    return 0;
  end
  else
    return numerator / denominator;
endfunction

// SystemVerilog style function call
always @(posedge clock)
  result <= divide(.denominator(b),
                   .numerator(a) );
```

- Declared with the keywords `function` and `endfunction`
- Parameters can be passed by name or by position
- SystemVerilog functions can be of type **`void`**
- SystemVerilog functions can have zero arguments
- SystemVerilog functions can have **`input`**, **`output`** and **`inout`** formal arguments
  - Default is **`input`**
- SystemVerilog allows empty function definitions (stubs)

Default arguments

- SystemVerilog allows specification of default formal argument values

```
function void print_checksum(ref bit [31:0] a[],
                             input bit [31:0] low = 0,
                             input int high = -1);
  bit [31:0] checksum = 0;

  if (high == -1 || high >= a.size())
    high = a.size()-1;

  for (int i=low; i<=high; i++)
    checksum += a[i];
  $display("The array checksum is %0d", sum);
endfunction
print_checksum(a);        // Checksum a[0:size()-1] - default
print_checksum(a, 2, 4); // Checksum a[2:4]
print_checksum(a, 1);     // Start at 1
print_checksum(a,, 2);    // Checksum a[0:2]
print_checksum();         // Compile error: a has no default
```

Functions: Example 1

```
// define the parity calculation function
function calc_parity;
        input [31:0] address;
        begin
                // set the output value appropriately. Use the
                // implicit internal register calc_parity.
                // Return the xor of all address bits.
                calc_parity = ^address;
        end
endfunction
```

...or using ANSI-style declarations

```
//define the function using ANSI C Style arguments
function calc_parity (input [31:0] address);
        begin
                // set the output value appropriately. Use the
                // implicit internal register calc_parity.
                // Return the xor of all address bits.
                calc_parity = ^address;
        end
endfunction
```

Functions: Example 1 (cont'd)


```
// define a module that contains the function calc_parity
module parity;
...
reg [31:0] addr;
reg parity;

//Compute new parity whenever address value changes
always @(addr) begin
        // first invocation of calc_parity()
        parity = calc_parity(addr);
        // second invocation of calc_parity()
        $display("Parity calculated = %b", calc_parity(addr) );
end
...
...
endmodule
```

Automatic (recursive) functions

- Functions are normally used non-recursively
  - Only a single copy of the return variable and any local variables
  - If a function is called concurrently from two locations…or, if a function calls itself (i.e. is recursive) results are non-deterministic
- Functions can be made automatic (recursive) by adding the keyword `automatic` to the functional declaration
  - Ex: `function automatic integer factorial;`
  - All function declarations are allocated dynamically for each recursive call
  - Each call to an automatic function operates in an independent variable space
  - Automatic function items cannot be accessed (even w/ hierarchical references) but the automatic function can be invoked through its hierarchical name

Example 2: Automatic function

```
// define a factorial with a recursive function
module top;
...
// define the function
function automatic integer factorial;
input [31:0] oper;
integer i;
begin
        if (operand >= 2)
        factorial = factorial (oper -1) * oper; //recursive call
        else
        factorial = 1 ;
end
endfunction
...
...
```

Source: HDL Chip Design by Douglas Smith

Example 2: Automatic function (cont'd)

```
…
…
// call the function
integer result;

initial begin
        result = factorial(4); // call the factorial of 4
        $display("Factorial of 4 is %0d", result); //displays 24
end
...
...
endmodule
```

Source: HDL Chip Design by Douglas Smith

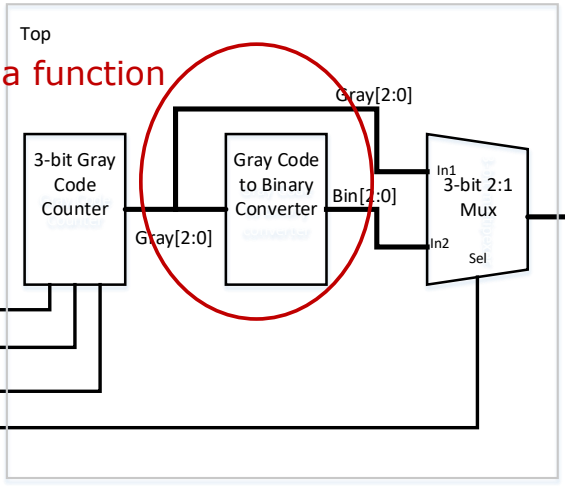
Signed functions

- SystemVerilog functions can return a signed result

```
module top;
…
// signed function declaration
// returns a 64 bit signed value
function signed [63:0] compute_signed(input [63:0] vector);
…
…
endfunction
…
//Call to the signed function from the higher module
if(compute_signed(vector) < -3)
begin
…
end
…
endmodule
```

Let’s write some code for a function

Gray code counter/mux revisited



Problem statement:

Implement a gray-code counter w/ a selectable binary and gray code output

When Conv == 0 you want Out[2:0] to be the unconverted Gray Code from the counter. When Conv == 1 you want the Out[2:0] to be the Binary code from the converter. The 3-bit Up/Down Gray Code counter will be implemented as a FSM.

Gray Code -> Binary

Gray Code			Binary Code		
Gray[2]	Gray[1]	Gray[0]	Bin[2]	Bin[1]	Bin[0]
0	0	0	0	0	0
0	0	1	0	0	1
1	0	1	0	1	0
1	0	0	0	1	1
1	1	0	1	0	0
1	1	1	1	0	1
0	1	1	1	1	0
0	1	0	1	1	1

Review Roy's midterm exam solution

Roy's solutions: ..\..\exams\midterm_exam\solution_code

Functions and tasks (cont'd)

Sources:

- Mark F. and Roy's lecture material from other courses

Arrays, structs, unions as arguments

- SystemVerilog allows arrays, structures, and unions to be passed in/out of tasks and functions
  - For structures and unions use **typedef**
  - Packed arrays treated like vectors; if size doesn't match they are truncated or expanded
  - Unpacked arrays must match length exactly

```
typedef struct {
  logic        valid;
  logic [ 7:0] check;
  logic [63:0] data;
} packet_t;

function void fill_packet (
  input logic [7:0] data_in [0:7], // array arg
  output packet_t data_out );  // structure arg

  for (int i=0; i<=7; i++) begin
    data_out.data[(8*i)+:8] = data_in[i];
    data_out.check[i] = ^data_in[i];
  end
  data_out.valid = 1;
endfunction
```

Passing arguments by reference

- When task/function is called, inputs are copied into the task/function where they become local values Upon return, outputs are copied to the caller
- Task/function can reference signals not passed in as arguments (synthesizable)
- However, this makes the task/function less portable/re-usable as the referenced signal is hardcoded in the task/function
- SystemVerilog adds pass by reference using ref keyword (only in automatic tasks/functions)
- Careful: semantics of ref arguments means that, unlike other arguments, changes made to them in the task/function will immediately affect the signal outside the task/function
- `ref` arguments may not be synthesizable…check the supported features for your synthesis tool

Passing arguments by reference (cont'd)

```
module chip (...);

  typedef struct {
    logic        valid;
    logic [ 7:0] check;
    logic [63:0] data;
  } packet_t;

  packet_t  data_packet;
  logic [7:0] raw_data [0:7];

  always @(posedge clock)
    if (data_ready)
      fill_packet (.data_in(raw_data),
                   .data_out(data_packet) );

  function automatic void fill_packet (
    ref logic [7:0] data_in [0:7],  // ref arg
    ref packet_t data_out );        // ref arg

    for (int i=0; i<=7; i++) begin
      data_out.data[(8*i)+:8] = data_in[i];
      data_out.check[i] = ^data_in[i];
    end
    data_out.valid = 1;
  endfunction
  ...
endmodule
```

Passing arguments by reference (cont'd)


```
function automatic void fill_packet (
  const ref logic [7:0] data_in [0:7],
  ref   packet_t data_out );
  ...
endfunction
```

Read-only reference argument

Named task/function ends

```
function int add_and_inc (int a, b);
  return a + b + 1;
endfunction : add_and_inc


task automatic check_results (
  input packet_t sent,
  ref   packet_t received,
  ref   logic    done );
  static int error_count;
  ...
endtask: check_results
```

Next Time

- Topics:
  - Implicit FSM's
  - Writing testbenches
- You should:
  - Be working on HW #3
- Homework, projects and quizzes
  - Homework #3 has been released. Due to D2L by 10:00 PM on Mon, 24-May
    - No late submissions after Noon on Mon, 01-Jun
  - Graded in-class exercise is schedule for Thu, 27-May