INF5430

SystemVerilog code examples

Spear & Tumbush

SystemVerilog for Verification

Chapter 2 and 3

Chapter 2: Data Types
2.1.1: The Logic Type

- Verilog has 2 data types
  - wire
  - reg
- SystemVerilog introduces the `logic` data type
  - Still 4-value
  - Replaces both wire and reg
  - Cannot have multiple drivers
  - For a bidirectional bus use the `wire` type.

```
logic reset;
logic [15:0] data;
```

2.1.2: 2-State Data Types

- Why?
  - Faster simulation and less memory than 4-state logic
  - Easier to interface to C/C++ functions
- `bit byte shortint int longint`

```
bit [15:0] my_reg;
shortint my_reg2;
```

- Caveats
  - bit is not signed
  - byte, shortint, int, longint are signed
  - Initial value is 0
  - X's and Z's resolve to a 0

2.1.2: 2-State Data Types (cont)

- 4-state logic types are x at beginning of simulation
- 2-state logic types are 0 at beginning of simulation
- Can assign 4-state to 2-state.
- Can use $isunknown to check for x or z

```
if ($isunknown(iport) == 1)
    $display("@%0t: 4-state value detected on
              iport %b", $time, iport);
```

2.4 Queues

- Provides easy sorting and searching
- Can add and remove elements from anywhere
- Can dynamically grow and shrink. No new[].
- Can copy contents of fixed or dynamic arrays to the queue

```
string coworkers[$];
reg [7:0] address[$];
reg [7:0] j;
coworkers = {"Willy", "Rob"};
address = {8'h3F, 8'hA5};
coworkers.insert(1, "Holger"); //{"Willy", "Holger", "Rob"};
address.push_back(8'h00); // {8'h3F, 8'hA5, 8'h00};
j = address.pop_front(); // {8'hA5, 8'h00}, j = 8'h3F
j = address.size(); // j = 2
address.delete();
```

2.8 Creating new types with *typedef*

- Useful if you have a common bit width or type

Example: Create a unsigned 2-state `byte`

```
typedef bit [7:0] ubyte_t;
```

- Useful for avoiding bus width errors. For example, if all the busses in your system are 13-bits wide define it.

```
typedef logic [12:0] global_bus_t;
```

- Usage:

```
ubyte_t my_data
global_bus_t my_bus;
```

2.9 Creating User-Defined Structures

- A structure is a collection of variables and/or constants that can be accessed separately or as a whole
- Why?
  - Great for enumerating system resources
  - Allows grouping of data

```
typedef struct {
                reg [7:0] data_in;
                reg [3:0] address;
                } mem_bus;
mem_bus my_mem_bus = `{8'hA5, 4'hC}; //Initialize

initial begin
  my_mem_bus.data_in = 8'h5A; //Change all bits
  my_mem_bus.address = 4'h3;  //Change all bits
end
```

2.10 Packages

- Where to put all this stuff? In a package!
- Packages reduce the need for `` `include ``

```
package ABC;
   parameter int abc_data_width = 32;
   typedef logic [abc_data_width-1:0] abc_data_t;
   parameter time timeout = 100ns;
   string message = "ABC done";
endpackage // ABC
```

Importing Packages

- Recall that package ABC defined `abc_data_width, abc_data_t, timeout, and message`

```
module test;
   import ABC::*;
   abc_data_t data;
   string message = "Test timed out";
   initial begin
      #(timeout);
      $display("Timeout - %s", message);
      $finish;
   end
endmodule
```

- Use `ABC::message` to use the message variable in package ABC.

2.11 Type Conversion

Static cast converts between 2 types ⟵ No bounds checking

```
int i;
real r;
byte b;

initial begin
   i=int'(10.0-0.1);
   $display("i = 0d%0d", i);
   r=real'(42);
   $display("r = %f", r);
   b=byte'(256);
   $display("b = 0d%0d", b);
end
```

```
# i = 0d10
```

```
# r = 42.000000
```

```
# b = 0d0
```

2.13 Enumerations

- Create list of constant names

```
enum {CFG, ADC_REG, CTRL} reg_e;
enum {CFG=5, ADC_REG=6, CTRL} reg2_e;
```

- Easier than:

```
localparam CFG = 2'b00,
           ADC_REG = 2'b01,
           CTRL = 2'b10;
```

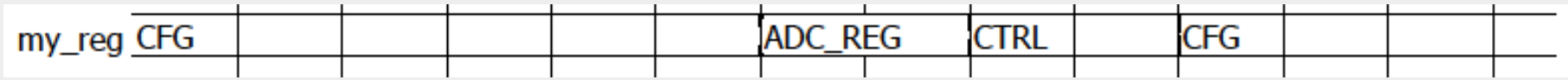
- Easier to add/delete registers
- Register name is not visible in waveform
- Usage

```
bit [1:0] my_reg;
my_reg = CFG;
```

2.13 Enumerated Types

- Creates user defined type of the enumeration
- Value of register is visible in waveform

```
typedef enum {CFG, ADC_REG, CTRL} reg_e;
reg_e my_reg;
initial begin
   my_reg = CFG;
   #30ns;
   my_reg = ADC_REG;
   #10ns;
   my_reg = CTRL;
   #10ns;
   my_reg = CFG;
end
```



2.13.2 Routines for enumerated types

```
typedef enum {ST0, ST1, ST2} state_e;
state_e state;
initial begin
  $display("first = 0x%0h", state.first);
  $display("next(1) = 0x%0h", state.next(1));
  $display("prev(1) = 0x%0h", state.prev(1));
  $display("num = 0x%0h", state.num);
  $display("name = %s", state.name);
  $display("last (hex) = 0x%0h", state.last);
  $display("last (string)=%s", state.last);
  $finish;
end
```

```
# first = 0x0
# next(1) = 0x1
# prev(1) = 0x2
# num = 0x3
# name = ST0
# last (hex) = 0x2
# last (string)=ST2
```

Stepping through all enumerated members

```
typedef enum {RED, BLUE, GREEN} color_e;
for (color_e color=color.first; color!=color.last; color= color.next)
      $display("Color = %0d/%s", color, color.name);
```

```
# Color = 0/RED
# Color = 1/BLUE
```

```
typedef enum {RED, BLUE, GREEN} color_e;
color_e color;
color = color.first;
do
   begin
   $display("Color = %0d/%s", color, color.name());
   color = color.next;
   end
while (color != color.first);
```

```
# Color = 0/RED
# Color = 1/BLUE
# Color = 2/GREEN
```

2.13.3 Converting to/from enum types

From enum type to non-enum type with simple assignment

```
typedef enum {Add, Sub, Not_A, ReductionOR} opcode_e;
opcode_e opcode;
int i;
i = Not_A;
$display("i=%0d", i);
```

```
# i=2
```

2.12.3 Converting to/from enum ... (cont)

From non-enum type to enum type requires static cast or $cast

```
i=3;
if (!$cast(opcode,i))
   $display("Cast failed for i=%0d", i);
$display("opcode=%s", opcode);                  # opcode=ReductionOR
i=4;
if (!$cast(opcode,i))
   $display("Cast failed for i=%0d", i);        # Cast failed for i=4
$display("opcode=%s", opcode);                  # opcode=ReductionOR
opcode = opcode_e'(i);
$display("opcode=%s", opcode);                  # opcode=4
```

2.15 Strings

Why?

- Much simpler string manipulation
- No more packing characters into a reg variable

```
string s;
s="AMI Semiconductor";
$display(s.toupper());          # AMI SEMICONDUCTOR
$display(s.substr(3,7));        # Semi

s = {"ON ", s.substr(s.len()-13, s.len()-1)};
$display(s);                    # ON Semiconductor
s={s, " 2008"};
$display(s);                    # ON Semiconductor 2008
```

$random & $urandom

- Available since Verilog 1995.
- $random creates a 32-bit signed random number.
- If your range is a power of 2 use implicit bit selection

```
logic [3:0] addr;
addr = $random;
```

- If your range is not a power of 2 use modulus (%).
  Example: generate a random addr between 0 and 5

```
addr = $unsigned($random) % 6;
```

- New to SystemVerilog is $urandom and $urandom_range

```
$urandom_range(5,0);
```

- Unless your code changes the random generation will be the same. Use a seed to change the generation

```
integer r_seed = 2;
addr = $unsigned($random(r_seed)) % 6;
```

Chapter 3: 3.1 Procedural Statements

- post increment (`i++`), pre increment (`++i`)
- post decrement (`i--`), pre decrement (`--i`)
- Loop enhancements
  - locally defined index (`for (int i=0;....)`)
  - continue
  - Break

The testbench below processes a list of commands:

```
file = $fopen("commands.txt", "r");
while (!$feof(file)) begin
  c = $fscanf(file, "%s", cmd);
  case (cmd)
     ""     : continue; //Blank line; skip it and cont.
     "done": break;    //Drop out of loop
      ........
  endcase
end
$fclose(file);
```

3.1 Procedural Statements (cont)

- Verilog 1995 supported while loops
- A while loop might never execute (if count is 128 or greater ….)

```
while (count < 128) begin
  $display("Count = %d", count);
  count = count + 1;
end
```

- SystemVerilog adds a `do..while` loop (similar to C)
- A `do..while` loop always executes at least once
- Control of the loop is tested at the end of each pass of the loop

```
do begin
  $display("Count = %d", count);
  count = count + 1;
end while (count < 128);
```

3.1 Procedural Statements (cont)

A case item can now be a range of values.

```
case (address) inside
   [0:16'h1FFF]: $display("Address in ROM");
   [16'h2000:16'h7FFF]: $display("Address in SRAM");
   [16'h8000:16'hFFFF]: $display("Address in DRAM");
endcase
```

3.2 Tasks, Functions, and Void Functions

- In Verilog 2001 functions must specify a return value

```
function int func_2001(input integer new_int);
   func_2001 = new_int;
endfunction
```

- System Verilog defines the `void` function
- For functions that have no return value the return is void

```
function void func_sv(input int new_int);
   $display("new_int = %d", new_int);
endfunction
```

- Can ignore a tasks return value

```
void'(func_2001(1));
```

3.3 & 3.4 Routine Arguments

- In Verilog 2001
  - begin/end required
  - `input/output/inout` to tasks/functions copied to a local variable.
  - Arrays could not be passed
- SystemVerilog
  - begin/end not required
  - Argument to tasks/functions can be passed by reference (requires automatic!)
  - Arrays can be passed (best with “ref” or “const ref” to avoid copy)

```
function automatic void my_func(ref int func_array[1023:0]);
    $display("func_array[0] = %h", func_array[0]);
  endfunction
```

or

```
function automatic void my_func(const ref int func_array[1023:0]);
    $display("func_array[0] = %h", func_array[0]);
  endfunction
```

3.4.4 Default value for Arguments

- Suppose we have the following task which is called 100's of times

```
task compare(input bit [7:0] expected, input logic [7:0] actual)
if (expected !== actual)
  $display("Error: Expect of %h != actual of %h", expected, actual);
endtask
```

- Now we want to pass in a string of what is being compared.
- Use default values

```
task compare(input bit [7:0] expected, input logic [7:0] actual,
                              input  string compared = "NULL");
  if (expected !== actual)
     $display("Error: For %s expect of %0h != actual of %0h",
                          compared, expected, actual);
endtask

compare(8'h8, 8'h5);
compare(8'h8, 8'h5, "my_reg");
```

```
# Error: For NULL expect of 8 != actual of 5
# Error: For my_reg expect of 8 != 5
```

3.4.5 Passing Arguments by name

- Just like modules, specify task/function arguments by name instead of position.
- Only pass those arguments that don't match default

```
task many(input int a=1, b=2, c=3, d=4);
  $display("%0d  %0d  %0d  %0d", a, b, c, d);
endtask

initial begin
  many(.d(6),.c(7),.b(8),.a(9));
  many();
  many (.c(5));
  many(, 6, .d(8));
end
```

```
# 9 8 7 6
# 1 2 3 4
# 1 2 5 4
# 1 6 3 8
```

3.5 Returning from a Routine

- With Verilog-2001 no way to end a task/function early
- SystemVerilog adds the `return` statement

```
task automatic my_task(ref int my_array[]);
   if (my_array.size() == 0)
      return;
   $display("my_array = %p", my_array);
endtask
```

- Value returned by a function can be specified by `return`

```
function [8:0] increment(input [8:0] address);
      return ++address; // Equivalent
      // increment = ++address; // Equivalent
endfunction
```

3.6 Local Data Storage

- With Verilog-1995 space for a task is allocated once.
- All calls to a task use this single memory space.
- Concurrent calls to a task will clobber each other.

```
int new_address1, new_address2;
initial begin
   my_task(5, new_address1);
   $display("new_address1 = %0d", new_address1);
end
initial begin
   my_task(6, new_address2);
   $display("new_address2 = %0d", new_address2);
end
task my_task(input int address, output int new_address);
   #5ns;
   new_address = address;
endtask // my_task
```

```
# new_address1 = 6
```

```
# new_address2 = 6
```

3.6 Local Data Storage (cont)

- With Verilog 2001 memory space can be allocated for each task call
- Deallocated after each task call
- **Use the `automatic` keyword**

```
int new_address1, new_address2;
initial begin
   my_task(5, new_address1);
   $display("new_address1 = %0d", new_address1);
end
initial begin
   my_task(6, new_address2);
   $display("new_address2 = %0d", new_address2);
end
task automatic my_task(input int address, output int
new_address);
   #5ns;
   new_address = address;
endtask // my_task
```

```
# new_address1 = 5
```

```
# new_address2 = 6
```

3.6.2 Variable initialization - bug

Local variables are initialized before the start of simulation.

```
program initialization;
   task check_bus();
      repeat (5) @(posedge clock);
      if (bus_cmd === READ) begin
         logic [7:0] local_addr = addr<<2;
         $display("Local Addr = %h", local_addr);
      end
   endtask
endprogram
```

3.6.2 Variable initialization - solutions

Solution 1: Declare program as automatic

```
program automatic initialization;
...
endprogram
```

Solution 2: Separate variable declaration from assignment.

```
program initialization;
   task check_bus();
      repeat (5) @(posedge clock);
      if (bus_cmd === READ) begin
         logic [7:0] local_addr;
         local_addr = addr<<2;
         $display("Local Addr = %h", local_addr);
      end
   endtask
endprogram
```

3.7.2 Time Literals

- For Verilog 1995/2001 `timescale determined the units of #

```
#5;  // ns, ps, fs ???
```

- SystemVerilog allows specification of units.

Units: `fs ps ns us ms s`

- Use the Verilog 2001 `$timeformat` to specify how time is displayed
- Syntax is `$timeformat(units_number, precision_number, suffix_string, minimum_field_width) ;`

```
timeunit 1ns; timeprecision 1ps;
initial begin
   $timeformat(-9, 3, "ns", 8);
   #1     $display("%t", $realtime); // # 1.000ns
   #2ns   $display("%t", $realtime); // # 3.000ns
   #0.1ns $display("%t", $realtime); // # 3.100ns
   #41ps  $display("%t", $realtime); // # 3.141ns
end
```

3.7.3 Time and Variables

- Time values are scaled according to the current timeunit and precision
- `time` type is a 64-bit integer

```
timeunit 1ns;
timeprecision 100ps;
initial begin
   realtime rtdelay = 849ps;
   time tdelay = 800ps;
   $timeformat(-12, 0, "ps", 5);
   #rtdelay;
   $display("@%0t, rtdelay = %t, %f", $realtime, rtdelay, rtdelay);
   #tdelay;
   $display("@%0t, tdelay = %t, %0d", $realtime, tdelay, tdelay);
end
```

```
# @800ps, rtdelay = 800ps, 0.800000
# @1800ps, tdelay = 1000ps, 1
```

3.7.4 $time vs $realtime

- System task $time returns an integer scaled to the time unit
- $realtime returns a real number scaled to the time precision

```
timeunit 1ns; timeprecision 1ps;
initial begin
   $timeformat(-9, 3, "ns", 8);
   #1     $display("%t", $realtime);   # 1.000ns
   #2ns   $display("%t", $realtime);   # 3.000ns
   #0.1ns $display("%t", $realtime);   # 3.100ns
   #41ps  $display("%t", $realtime);   # 3.141ns
end
initial begin
   #1     $display("%t", $time);       # 1.000ns
   #2ns   $display("%t", $time);       # 3.000ns
   #0.1ns $display("%t", $time);       # 3.000ns
   #41ps  $display("%t", $time);       # 3.000ns
end
```

3.7.1 Time units and Precision

- For Verilog 1995/2001 `` `timescale `` determines timescale/precision
- Applies for files compiled after `` `timescale `` is encountered
- SystemVerilog introduces the `timeunit` and `timeprecision` declarations

```
module test;
  timeunit 1ns;
  timeprecision 1ps;
endmodule
```

- **Because these are local to a module must be declared in every module that has delay statements.**
- **Overrides `` `timescale ``**