This manual contains solutions to Exercises at the end of each chapter.

Table of Contents

[1 Solution to Exercises for Chap 1 Verification Guidelines 2](#_Toc302475226)

[2 Solution to Exercises for Chap 2 Data Types 4](#_Toc302475227)

[3 Solution to Exercises for Chap 3 Procedural Statements and Routines 12](#_Toc302475228)

[4 Solution to Exercises for Chap 4 Connecting the Testbench and Design 15](#_Toc302475229)

[5 Solution to Exercises for Chap 5 Basic OOP 19](#_Toc302475230)

[6 Solution to Exercises for Chap 6 Randomization 25](#_Toc302475231)

[7 Solution to Exercises for Chap 7 Threads and Interprocess Communication 34](#_Toc302475232)

[8 Solution to Exercises for Chap 8 Advanced OOP and Testbench Guidelines 41](#_Toc302475233)

[9 Solution to Exercises for Chap 9 Functional Coverage 47](#_Toc302475234)

[10 Solution to Exercises for Chap 10 Advanced Interfaces 50](#_Toc302475235)

[11 Solution to Exercises for Chap 11 A Complete SystemVerilog Testbench 54](#_Toc302475236)

[12 Solution to Exercises for Chap 12 Interfacing with C 57](#_Toc302475237)

# Solution to Exercises for Chap 1 Verification Guidelines

1. Write a verification plan for an Arithmetic Logic Unit (ALU) with:

* Asynchronous active high input reset
* Input clock
* 4-bit signed inputs, A and B
* 5-bit signed output C that is registered on the positive edge of input clock.
* 4 opcodes
* Add: A+B
* Sub: A-B
* Bitwise invert A
* reduction OR B

*Solution: There are many solutions to this Exercise. One possible solution is:*

*Check reset value of output C.*

*Apply all permutations of max pos, max neg, and 0 to the add and subtract opcodes*

*Apply 0 and all 1’s to A for bitwise invert input A opcode. Set B to non-all 0 and non-all 1.*

*Apply 0, all 1’s, and walking 1’s to B for ReductionOR\_B opcode. Set A to a value that will yield the opposite result of B, i.e. if the expected value of B =1, set A to a value so that the bitwise invert is 0.*

*Assert reset when output C is 5’b11111.*

1. What are the advantages and disadvantages to testing at the block level? Why?

*Solution: The advantages of verifying at the block level are speed of execution due to the small amount of circuitry being simulated, ease of debug, again, due to the small amount of circuitry being simulated, and excellent control because the testbench has direct control of the I/O of the block.*

*The first disadvantage to verifying at the block level is testbench complexity if a block has many interfaces. The testbench can be very complex because all of these interfaces must be modeled. Another disadvantage is simply that the system is not being tested, just individual blocks. There is no guarantee that when all the verified blocks are integrated that they will implement the specification.*

1. What are the advantages and disadvantages to testing at the system level? Why?

*Solution: The advantage of verifying at the system level is that the final system is being tested and results can be compared directly to the specification. A system level testbench can be simpler than a block level testbench if the interfaces are few and simple and the checking mechanism is simple.*

*The disadvantages of verifying at the system level are speed of execution due to the large amount of circuitry being simulated, difficulty of debug due to the many layers of circuitry and complex interaction between circuits, and poor control because the testbench has to traverse many layers of circuitry to manipulate a particular signal.*

1. What are the advantages and disadvantages to directed testing? Why?

*Solution: The advantages to directed testing are fast testbench development time because no complex scoreboard or reference model is required, linear progress because each test takes a similar amount of time to write, and popularity of management because steady progress is being made.*

*The disadvantages to directed testing is that bugs are easy to miss because corner cases are not tested. Random testing finds corner cases that the verification engineer never thought about.*

1. What are the advantages and disadvantages to constrained random testing? Why?

*Solution: The advantages to constrained random testing are shorter total verification time due to better than linear progress, ease of maintenance because if the specification changes only the scoreboard or reference model needs to change, and a methodology that is reusable and scales to larger designs.*

*The disadvantages are that initially no verification is being done due to a more complex testbench being developed.*

# Solution to Exercises for Chap 2 Data Types

1. Given the following code sample:

byte my\_byte;

integer my\_integer;

int my\_int;

bit [15:0] my\_bit;

shortint my\_short\_int1;

shortint my\_short\_int2;

my\_integer = 32’b000\_1111\_xxxx\_zzzz;

my\_int = my\_integer;

my\_bit = 16’h8000;

my\_short\_int1= my\_bit;

my\_short\_int2 = my\_short\_int1-1;

1. What is the range of values *my\_byte* can take?

*Solution: my\_byte ranges from -128 (-(27)) to 127 (27-1) because my\_byte is of 8-bit signed type.*

1. What is the value of *my\_int* in hex?

*Solution: my\_int is 32’h0000\_0F00 because the X’s and Z’s in 4-state my\_integer resolve to 0 when assigned to 2-state my\_int.*

1. What is the value of *my\_bit* in decimal?

*Solution: my\_bit is 32768 because my\_bit is not signed*

1. What is the value of *my\_short\_int1* in decimal?

*Solution:* *my\_short\_int1 is -32768 because my\_short\_int1 is signed*

1. What is the value of *my\_short\_int2* in decimal?

*Solution: my\_short\_int2 is 32767 because it has wrapped from max\_neg to max\_pos*

*See Chap\_2\_Data\_Types/exercise1 for code.*

1. Given the following code sample:

bit [7:0] my\_mem [3];

logic [3:0] my\_logicmem [4];

logic [3:0] my\_logic;

my\_mem = '{default:8'hA5};

my\_logicmem = '{0,1,2,3};

my\_logic = 4’hF;

Evaluate in order:

1. my\_mem[2] = my\_logicmem[4];

*Solution:* *my\_mem = {A5, A5, 0} because reading from an out of bounds 4-valued type returns an X but it is resolved to a 0 when assigned to 2-value logic.*

1. my\_logic = my\_logicmem[4];

*Solution: my\_logic = 4’hxbecause reading from an out of bounds 4-valued type returns an X*

1. my\_logicmem[3] = my\_mem[3];

*Solution: my\_logicmem= {0,1,2,0} because reading from an out of bounds 2-value type returns a 0*

1. my\_mem[3] = my\_logic;

*Solution: my\_mem does not change since the write to my\_mem[3] is out of range.*

1. my\_logic = my\_logicmem[1];

*Solution: my\_logic = 1 since my\_logicmem[1] = 1*

1. my\_logic = my\_mem[1];

*Solution: my\_logic = 4’h5 since my\_mem[1] = 8’hA5 but my\_logic is only 4-bits wide*

1. my\_logic = my\_logicmem[my\_logicmem[4]];

*Solution:* *my\_logic =4’h x since my\_logicmem[4] returns an X since the index is out of bounds. my\_logicmem[x] = x.*

*See Chap\_2\_Data\_Types/exercise2 for code.*

1. Write the SystemVerilog code to:
2. Declare a 2-state array, *my\_array*, that holds four 12-bit values
3. initialize *my\_array* so that:
   * my\_array[0] = 12’h012
   * my\_array[1] = 12’h345,
   * my\_array[2] = 12’h678,
   * my\_array[3] = 12’h9AB;
4. Traverse *my\_array* and print out bits [5:4] of each 12-bit element
   1. Using a for loop
   2. Using a foreach loop

*Solution:*

module test;

bit [11:0] my\_array [4];

initial begin

my\_array = '{12'h012, 12'h345, 12'h678, 12'h9AB};

for (int i=0;i<$size(my\_array); i++) begin

$display("my\_array[%0d]=12'b%b", i, my\_array[i]);

end

$display("--------------");

foreach (my\_array[i]) begin

$display("my\_array[%0d]=12'b%b", i, my\_array[i]);

end

end

endmodule

*See Chap\_2\_Data\_Types/exercise3 for code*

1. Declare a 5 by 31 multi-dimensional unpacked array, my\_array1. Each element of the unpacked array holds a 4-state value.

*Solution:*

*logic my\_array1 [5] [31];*

1. Which of the following assignments are legal and not out-of-bounds?
2. my\_array1[4][30] = 1'b1;

*Solution: This assignment is legal because only assignments of a single bit are allowed to be made to an unpacked array of type logic. The assignment is in-bounds because bit [30] of my\_array1[4] is defined.*

1. my\_array1[29][4] = 1'b1;

*Solution: This assignment is legal because only assignments of a single bit are allowed to be made to an unpacked array of type logic. The assignment is out-of-bounds because bit [4] of my\_array1[29] is not defined.*

1. my\_array1[4] = 31'b1;

*Solution: The value 31’b1 is a packed type and assigning a packed type to an unpacked type is illegal.*

1. Draw *my\_array1* after the legal assignments complete.

*Solution: All locations not marked with a 1 have the value X.*



*See Chap\_2\_Data\_Types/exercise4 for code*

1. Declare a 5 by 31 multi-dimensional packed array, my\_array2. Each element of the packed array holds a 2-state value.

*Solution:*

bit [4:0] [30:0] my\_array2;

* 1. Which of the following assignments are legal and not out-of-bounds?
     1. my\_array2[4][30] = 1'b1;

*Solution:* *This assignment is legal because a specific bit of a packed array may be assigned to. The assignment is in-bounds because bit [30] of my\_array2[4] is defined.*

* + 1. my\_array2[29][4] = 1'b1;

Solution: *This assignment is legal because a specific bit of a packed array may be assigned to. The assignment is out-of-bounds because bit [4] of my\_array2[29] is not defined.*

* + 1. my\_array2[3] = 31'b1;

*Solution: This assignment is legal because a 31-bit packed value can be written to a specific 31-bit location of a packed array.*

* 1. Draw my\_array2 after the assignment statements complete.

*Solution: All locations not marked with a 1 have the value 0.*



*See Chap\_2\_Data\_Types/exercise5 for code*

1. Given the following code, determine what will be displayed.

`default\_nettype none

module test;

string street[$];

initial begin

street = {"Tejon", "Bijou", "Boulder"};

$display("Street[0] = %s", street[0]);

street.insert(2, "Platte");

$display("Street[2] = %s", street[2]);

street.push\_front("St. Vrain");

$display("Street[2] = %s", street[2]);

$display("pop\_back = %s", street.pop\_back);

$display("street.size = %0d", street.size);

end

endmodule // test

Solution:

* $display("Street[0] = %s", street[0]); will display **Street[0] = Tejon** because “Tejon” is the first string in the queue.
* $display("Street[2] = %s", street[2]); will display **Street[2] = Platte** because the “Platte” was previously inserted in location 2 of the queue. The queue now consists of {"Tejon", "Bijou", “Platte”, "Boulder"};
* $display("Street[2] = %s", street[2]); will display **Street[2] = Bijou** because "St. Vrain" was pushed to the front of the queue which pushed “Bijou” to location 2 of the queue. The queue now consists of {"St. Vrain", "Tejon", "Bijou", “Platte”, "Boulder"};
* $display("pop\_back = %s", street.pop\_back); will display **pop\_back = Boulder** because the last string in the queue is “Boulder” and it was popped. The queue now consists of {"St. Vrain", "Tejon", "Bijou", “Platte”};
* $display("street.size = %d", street.size); will display **street.size = 4** because the number of elements in the queue is 4.

*See Chap\_2\_Data\_Types/exercise6 for code*

1. Write code for the following problems.
2. Create memory using an associative array for a processor with a word width of 24 bits and an address space of 220 words. Assume the PC starts at address 0 at reset. Program space starts at 0x400. The ISR is at the maximum address.
3. Fill the associated array with the following instructions:

* 24'hA51000; // Jump to location 0x400 for the main code
* 24'h123456; // Instruction 1 located at location 0x400
* 24'h789ABC; // Instruction 2 located at location 0x401
* 24'h0F1E2D; // ISR = Return from interrupt

1. Print out the elements and the number of elements in the array.

*Solution: See Chap\_2\_Data\_Types/exercise7 for code*

1. Create the SystemVerilog code for the following requirements

a. Create a 3-byte queue and initialize it with 1, -1, and 127

b. Declare an integer called total

c. Print out the sum of the queue

d. Print out the min and max values in the queue

e. Sort all values in the queue and print out the resulting queue

f. Print out the index of any negative values in the queue

g. Print out the positive values in the queue

h. Reverse sort all values in the queue and print out the resulting queue

*Solution: See Chap\_2\_Data\_Types/exercise8 for solution.*

1. Define a user defined 7-bit type and encapsulate the fields of the following packet in a structure using your new type. Lastly, assign the header to 7’h5A;



Solution:

module test;

typedef bit [6:0] bit7\_t;

typedef struct {

bit7\_t header;

bit7\_t cmd;

bit7\_t data;

bit7\_t crc;

} packet;

packet my\_packet;

initial begin

my\_packet.header = 7'h5A;

end

endmodule // test

*See Chap\_2\_Data\_Types/exercise9 for code*

1. Create the SystemVerilog code for the following requirements:
   1. Create a user defined type, nibble, of 4 bits
   2. Create a real variable, *r*, and initialize it to 4.33
   3. Create a short int variable, *i\_pack*
   4. Create an unpacked array, *k*, containing 4 elements of your user defined type nibble and initialize it to 4’h0, 4’hF, 4’hE, and 4’hD
   5. Print out *k*
   6. Stream *k* into *i\_pack* right to left on a bit basis and print it out
   7. Stream *k* into *i\_pack* right to left on a nibble basis and print it out
   8. Type convert real *r* into a nibble, assign it to *k[0]*, and print out *k*

*Solution: See Chap\_2\_Data\_Types/exercise10 for solution.*

1. An ALU has the following opcodes

|  |  |
| --- | --- |
| **Opcode** | **Encoding** |
| Add: A+B | 2’b00 |
| Sub: A-B | 2’b01 |
| Bitwise invert A | 2’b10 |
| Reduction OR B | 2’b11 |

Write a testbench that performs the following tasks

1. Create an enumerated type of the opcodes: *opcode\_t*
2. Create a variable, *opcode*, of type *opcode\_t*
3. Loop through all the values of variable *opcode* every 10ns
4. Instantiate an ALU with one 2-bit input *opcode*

*Solution: See Chap\_2\_Data\_Types/exercise11 for solution.*

# Solution to Exercises for Chap 3 Procedural Statements and Routines

1. Create the SystemVerilog code for the following requirements:
   1. Create a 512 element integer array
   2. Create a 9-bit address variable to index into the array
   3. Initialize the last location in the array to 5
   4. Call a task, *my\_task()*, and pass the array and the address
   5. Create *my\_task()* that takes 2 inputs, a constant 512-element integer array passed by reference, and a 9-bit address. The task calls a function, *print\_int()*,and passes the array element, indexed by the address, to the function, pre-decrementing the address.
   6. Create *print\_int()* that prints out the simulation time and the value of the input. The function has no return value

*Solution: See Chap\_3\_Procedural\_Statements\_and\_Routines/exercise1 for solution.*

1. For the following SystemVerilog code, what is displayed if task *my\_task2()* is automatic?

int new\_address1, new\_address2;

bit clk;

initial begin

fork

my\_task2(21, new\_address1);

my\_task2(20, new\_address2);

join

$display("new\_address1 = %0d", new\_address1);

$display("new\_address2 = %0d", new\_address2);

end

initial

forever #50 clk = !clk;

task my\_task2(input int address, output int new\_address);

@(clk);

new\_address = address;

endtask

*Solution: new\_address1 = 21 and new\_address2 = 20 will be displayed.*

*See Chap\_3\_Procedural\_Statements\_and\_Routines/exercise2 for code.*

1. For the same SystemVerilog code in Exercise 2 what is displayed if the task *my\_task2()* is not automatic?

*Solution: If the call to my\_task2(20, new\_address2) is scheduled last* ***new\_address1 = 20*** *and* ***new\_address2 = 20*** *will be displayed. If the call to my\_task2(21, new\_address1) is scheduled last* ***new\_address1 = 21*** *and* ***new\_address2 = 21*** *will be displayed.*

*See Chap\_3\_Procedural\_Statements\_and\_Routines/exercise3 for code.*

1. Create the SystemVerilog code to specify that the time should be printed in ps (picoseconds), 2 digits to the right of the decimal point, and use as few characters as possible.

*Solution:*

$timeformat(-12, 2, "ps", 0);

*See Chap\_3\_Procedural\_Statements\_and\_Routines/exercise4 for code.*

1. Using the formatting system task from Exercise 4, what is displayed by the following code.

timeunit 1ns;

timeprecision 1ps;

parameter real t\_real = 5.5;

parameter time t\_time = 5ns;

initial begin

#t\_time $display("1 %t", $realtime);

#t\_real $display("1 %t", $realtime);

#t\_time $display("1 %t", $realtime);

#t\_real $display("1 %t", $realtime);

end

initial begin

#t\_time $display("2 %t", $time);

#t\_real $display("2 %t", $time);

#t\_time $display("2 %t", $time);

#t\_real $display("2 %t", $time);

end

*Solution:*

*In the first initial block the following will displayed because $realtime does not round to the nearest timeunit of 1ns.*

#t\_time $display("1 %t", $realtime); will display **1 5000.00ps**

#t\_real $display("1 %t", $realtime); will display **1 10500.00ps**

#t\_time $display("1 %t", $realtime); will display **1 15500.00ps**

#t\_real $display("1 %t", $realtime); will display **1 21000.00ps**

*In the second initial block the following will displayed because $time does round to the nearest timeunit of 1ns.*

#t\_time $display("2 %t", $time); will display **2 5000.00ps**

#t\_real $display("2 %t", $time); will display **2 11000.00ps**

#t\_time $display("2 %t", $time); will display **2 16000.00ps**

#t\_real $display("2 %t", $time); will display **2 21000.00ps**

*See Chap\_3\_Procedural\_Statements\_and\_Routines/exercise5 for code.*

# Solution to Exercises for Chap 4 Connecting the Testbench and Design

1. Design an interface and testbench for the ARM Advanced High-performance Bus (AHB). You are provided a bus master as verification IP that can initiate AHB transactions. You are testing a slave design. The testbench instantiates the interface, slave, and master. Your interface will display an error if the transaction type is not IDLE or NONSEQ on the negative edge of HCLK. The AHB signals are described below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Width** | **Direction** | **Description** |
| HCLK | 1 | output | Clock |
| HADDR | 21 | output | Address |
| HWRITE | 1 | output | Write flag. 1=write, 0=read |
| HTRANS | 2 | output | Transaction type. 2’b00 = IDLE, 2’b10 = NONSEQ |
| HWDATA | 8 | output | Write data |
| HRDATA | 8 | input | Read data |

*Solution: See Chap\_4\_Connecting\_the\_Testbench\_and\_Design/exercise1 for solution.*

1. For the following interface, add the following code
2. A clocking block that is sensitive to the negative edge of the clock, and all I/O that are synchronous to the clock.
3. A modport for the testbench called *master*, and a modport for the DUT called *slave*
4. Use the clocking block in the I/O list for the master modport.

interface my\_if(input bit clk);

bit write;

bit [15:0] data\_in;

bit [7:0] address;

logic [15:0] data\_out;

endinterface

*Solution:*

interface my\_if(input bit clk);

bit write;

bit [15:0] data\_in;

bit [7:0] address;

logic [15:0] data\_out;

clocking cb @(negedge clk);

output write, data\_in, address;

input data\_out;

endclocking;

modport slave (input clk, write, data\_in, output dat\_out);

modport master(clocking cb);

endinterface

*See Chap\_4\_Connecting\_the\_Testbench\_and\_Design/exercise2\_3 for code.*

1. For the clocking block in Exercise 2, fill in the *data\_in* and *data\_out* signals in the following timing diagram.



**Solution:**



*See Chap\_4\_Connecting\_the\_Testbench\_and\_Design/exercise2\_3 for code.*

1. Modify the clocking block in Exercise 2 to have
2. output skew of 25ns for outputs write and address
3. input skew of 15ns
4. restrict *data\_in* to only change on the positive edge of the clock

*Solution:*

clocking cb @(negedge clk);

output #25ns write;

output #25ns address;

output posedge data\_in;

input #15ns data\_out;

endclocking;

*See Chap\_4\_Connecting\_the\_Testbench\_and\_Design/exercise4\_5 for code.*

1. For the clocking block in the Exercise 4, fill in the following timing diagram. Assume the period of test/reg\_bus/clk is 100ns.



*Solution:*

**

*See Chap\_4\_Connecting\_the\_Testbench\_and\_Design/exercise4\_5 for code.*

# Solution to Exercises for Chap 5 Basic OOP

1. Create a class called *MemTrans* that contains the following members, then construct a *MemTrans* object in an initial block.
2. An 8-bit *data\_in* of logic type
3. A 4-bit *address* of logic type
4. A void function that prints out the value of *data\_in* and *address*

*Solution:*

class MemTrans;

logic [7:0] data\_in;

logic [3:0] address;

function void print;

$display(“Data\_in = 0x%0h, address = 0x%0h”, data\_in, address);

endfunction

endclass

initial begin

MemTrans MyMemTrans;

MyMemTrans = new(); // or MemTrans MyMemTrans = new();

end

*See Chap\_5\_Basic\_OOP/exercise1 for code.*

1. Using the MemTrans class from Exercise 1, create a custom constructor, the *new* function, so that *data\_in* and *address* are both initialized to 0.

*Solution:*

// Define function in class MemTrans

function new;

data\_in = 8’h0;

address = 4’h0;

endfunction

*See Chap\_5\_Basic\_OOP/exercise2 for code.*

1. Using the MemTrans class from Exercise 1, create a custom constructor so that *data\_in* and *address* are both initialized to 0 but can also be initialized through arguments passed into the constructor. In addition, write a program to perform the following tasks.
2. Create two new *MemTrans* objects
3. Initialize *address* to 2 in the first object, passing arguments by name
4. Initialize *data\_in* to 3 and *address* to 4 in the second object, passing arguments by name.

*Solution:*

// Define function in class MemTrans

function new(logic [7:0] data\_init = 0, logic [3:0] address\_init = 0);

data\_in = data\_init;

address = address\_init;

endfunction

initial begin

MemTrans mt1, mt2; // Create 2 new objects of class MemTrans

mt1 = new(.address\_init(2)); // Initialize address to 2 by name in the 1st object

mt2 = new(.data\_init(3),.address\_init( 4)); // Initialize *data\_in* to 3 and address to 4 in the

// 2nd object by name

end

*See Chap\_5\_Basic\_OOP/exercise3 for code.*

1. Modify the solution from Exercise 3 to perform the following tasks.
2. After construction, set the address of the first object to 4'hF.
3. Use the *print* function to print out the values of *data\_in* and *address* for the two objects.
4. Explicitly deallocate the 2nd object.

*Solution:*

initial begin

…….

m1.address = 4’hF; // Assign the address of the first object to 4’hF

// Use the print function to print out the values of *data\_in* and *address* for the 2 objects.

mt1.print;

mt2.print;

mt2 = null; // Explicitly deallocate the 2nd object.

end

*See Chap\_5\_Basic\_OOP/exercise4 for code.*

1. Using the solution from Exercise 4, create a static variable *last\_address* that holds the initial value of the address variable from the most recently created object, as set in the constructor. After allocating objects of class *MemTrans* (done in Exercise 4) print out the current value of *last\_address.*

*Solution:*

// Add static member variable last\_address to class MemTrans declaration

static logic [3:0] last\_address;

// Add to custom contructor to keep track of address of most recently created object

MemTrans::last\_address = address;

// Add to initial block

$display("last\_address is %h", MemTrans::last\_address);

// or

$display("last\_address is %h", mt1.last\_address);

// or

$display("last\_address is %h", mt2.last\_address);

*See Chap\_5\_Basic\_OOP/exercise5 for code.*

1. Using the solution from Exercise 5, create a static method called *print\_last\_address()* that prints out the value of the static variable *last\_address*. After allocating objects of class *MemTrans*, call the method *print\_last\_address()* to print out the value of *last\_address.*

*Solution:*

// Add static member function print\_last\_address to class MemTrans declaration

static function void print\_last\_address;

$display("last\_address is %h", last\_address);

endfunction

// Add to initial block call to print\_last\_address

MemTrans::print\_last\_address;

// or

mt1.print\_last\_address;

// or

mt2.print\_last\_address;

*See Chap\_5\_Basic\_OOP/exercise6 for code.*

1. Given the following code, complete the function *print\_all* in class *MemTrans* to print out *data\_in* and *address* using the class *PrintUtilities*. Demonstrate using the function *print\_all*.

class PrintUtilities;

function void print\_4(input string name, input [3:0] val\_4bits);

$display("%t: %s = %h", $time, name, val\_4bits);

endfunction

function void print\_8(input string name, input [7:0] val\_8bits);

$display("%t: %s = %h", $time, name, val\_8bits);

endfunction

endclass

class MemTrans;

bit [7:0] data\_in;

bit [3:0] address;

PrintUtilities print;

function new();

print = new();

endfunction

function void print\_all;

// Fill in function body

endfunction

endclass

*Solution:*

function void print\_all;

print.print\_4("data\_in", data\_in);

print.print\_8("address", address);

endfunction

// Add to initial block to demonstrate usage

mt1.print\_all;

// or

mt2.print\_all;

*See Chap\_5\_Basic\_OOP/exercise7 for code.*

1. Complete the following code where indicated by the comments starting with // .

program automatic test;

import my\_package::\*;

initial begin

// Declare an array of 5 Transaction handles

// Call a generator task to create the objects

end

task generator(…. // Complete the task header

// Create objects for every handle in the array

// and transmit the object.

endtask

task transmit(Transaction tr);

.......

endtask // transmit

endprogram

*Solution:*

program automatic test;

import my\_package::\*;

initial begin

Transaction tarray[5]; // Declare an array of Transaction handles

generator(tarray); // Call a generator task to create the objects

end

// Complete the task header

task generator(ref Transaction gen\_array[5]);

// Create objects for every handle in the array and transmit the object.

foreach (gen\_array[i]) begin

gen\_array[i] = new();

transmit(gen\_array[i]);

end

endtask

*See Chap\_5\_Basic\_OOP/exercise8 for code.*

1. For the following class create a *copy* function and demonstrate its usage. Assume the *Statistics* class has its own *copy* function.

package automatic my\_package;

class MemTrans;

bit [7:0] data\_in;

bit [3:0] address;

Statistics stats;

function new();

data\_in = 3;

address = 5;

stats = new();

endfunction

endclass;

endpackage

*Solution:*

// Add member function copy to class MemTrans declaration

function MemTrans copy();

copy = new();

copy.address = address;

copy.data\_in = data\_in;

copy.stats = stats.copy();

endfunction

// Add to initial block to demonstrate usage

MemTrans mt3; // Declare new handle to point to copied object

mt3 = mt1.copy();

*See Chap\_5\_Basic\_OOP/exercise9 for code.*

# Solution to Exercises for Chap 6 Randomization

1. Write the SystemVerilog code for the following items:
2. Create a class *Exercise1* containing two random variables, 8-bit *data* and 4-bit *address*. Create a constraint block that keeps *address* to 3 or 4.
3. In an *initial* block, construct an *Exercise1* object and randomize it. Check the status from randomization.

*Solution:*

class Exercise1;

rand bit [7:0] data;

rand bit [3:0] address;

constraint address\_c {

address > 2;

address < 5;

// or

((address==3) || (address==4));

// or

address inside {[3:4]};

}

Endclass

initial begin

Exercise1 MyExercise1;

MyExercise1 = new;

assert (MyExercise1.randomize()) else $fatal(0, “MyExercise1.randomize failed”);

end

*See Chap\_6\_Randomization/exercise1\_3 for complete solution.*

1. Modify the solution for Exercise 1 to create a new class *Exercise2* that:
2. *data* is always equal to 5
3. Probability of *address* *== 0* is 10%
4. Probability of *address* being between [1:14] is 80%
5. Probability of *address* *== 15* is 10%

*Solution:*

class Exercise2;

rand bit [7:0] data;

rand bit [3:0] address;

constraint data\_c{data == 5;}

constraint address\_dist {

address dist{0:=10, [1:14]:/80, 15:=10};

}

endclass

*See Chap\_6\_Randomization/exercise2\_3 for complete solution.*

1. Using the solution to either Exercise 1 or 2, demonstrate its usage by generating 20 new *data* and *address* values and check for success from the constraint solver.

*Solution if Exercise* 1 was used:

initial begin

Exercise1 MyExercise1;

repeat (20) begin

MyExercise1 = new;

assert (MyExercise1.randomize())

else $fatal(0, "MyExercise1::randomize failed");

MyExercise1.print\_all();

end

end

*See Chap\_6\_Randomization/exercise1\_3 for complete solution.*

*Solution if Exercise* 2 was used:

initial begin

Exercise2 MyExercise2;

repeat (20) begin

MyExercise2 = new;

assert (MyExercise2.randomize())

else $fatal(0, "MyExercise2::randomize failed");

MyExercise2.print\_all();

end

end

*See Chap\_6\_Randomization/exercise2\_3 for complete solution.*

1. Create a testbench that randomizes the Exercise2 class 1000 times.
   1. Count the number of times each address value occurs and print the results in a histogram. Do you see an exact 10% / 80% / 10% distribution? Why or why not?
   2. Run the simulation with 3 different random seeds, creating histograms, and then comment on the results. Here is how to run a simulation with the seed 42.

VCS: > simv +ntb\_random\_seed=42

NCV: > irun exercise4.sv

Questa: > vsim -sv\_seed 42

*Solution:*

*Plotted using gnuplot with script “histogram.plt”*



I do not see an an exact 10% / 80% / 10% distribution because to achieve a perfect random distribution many more samples need to be generated. Different seeds produce a different random result.

*See Chap\_6\_Randomization/exercise4 for complete solution.*

1. For the code in Sample 6-4, describe the constraints on the *len, dst*, and *src* variables.

*Solution: The constraints limit* ***len*** *to between 1 and 999 inclusive. If* ***congestion\_test*** *is 0,* ***dst*** *is unconstrained and* ***src*** *can take on values 0, 2 to 10 inclusive, and 100 to 107 inclusive. If* ***congestion\_test*** *is 1,* ***src*** *= 42 and* ***dst*** *is constrained to be between 32 (42-10) and 52 (42+10) inclusive.*

*See Chap\_6\_Randomization/exercise5 for complete solution.*

1. Complete the table below for the following constraints:

class MemTrans;

rand bit x;

rand bit [1:0] y;

constraint c\_xy {

y inside{[x:3]};

solve x before y;

}

endclass

Solution:

|  |  |  |  |
| --- | --- | --- | --- |
| **Solution** | **x** | **y** | **Probability** |
| A | 0 | 0 | ½\* ¼ = 1/8 |
| B | 0 | 1 | ½\* ¼ = 1/8 |
| C | 0 | 2 | ½\* ¼ = 1/8 |
| D | 0 | 3 | ½\* ¼ = 1/8 |
| E | 1 | 0 | ½ \* 0 = 0 |
| F | 1 | 1 | ½ \*1/3=1/6 |
| G | 1 | 2 | ½ \*1/3=1/6 |
| H | 1 | 3 | ½ \*1/3=1/6 |

*See Chap\_6\_Randomization/exercise6 for complete solution.*

1. For the following class, create
2. A constraint that limits read transaction addresses to the range 0 to 7, inclusive.
3. Write behavioral code to turn off the above constraint. Construct and randomize a MemTrans object with an in-line constraint that limits read transaction addresses to the range 0 to 8, inclusive. Test that the in-line constraint is working.

class MemTrans;

rand bit rw; // read if rw = 0, write if rw = 1

rand bit [7:0] data\_in;

rand bit [3:0] address;

endclass // MemTrans

*Solution to part is to add the following code to the class:*

constraint valid\_rw\_address {(rw == 0)->(address inside {[0:7]});}

*Solution to part is to add the following code to the initial block in the testbench:*

MyMemTrans.valid\_rw\_address.constraint\_mode(0);

assert (MyMemTrans.randomize() with {(rw == 0)->(address inside {[0:8]});}) else $fatal(0, "MyMemTrans::randomize failed");

*See Chap\_6\_Randomization/exercise7 for complete solution.*

1. Create a class for a grapics image that is 10x10 pixels. The value for each pixel can be randomized to black or white. Randomly generate an image that is, on average, 20% white. Print the image and report the number of pixels of each type.

*Solution:*

package my\_package;

parameter int HEIGHT = 10;

parameter int WIDTH = 10;

parameter int PERCENT\_WHITE = 20;

typedef enum bit {BLACK, WHITE} colors\_t;

class Screen;

rand colors\_t pixels [HEIGHT][WITDH];

constraint colors\_c {foreach (pixels[i,j]) pixels[i][j] dist {BLACK :=100-PERCENT\_WHITE, WHITE := PERCENT\_WHITE};}

endclass

endpackage

# '{BLACK, BLACK, BLACK, WHITE, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK}

# '{BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, WHITE}

# '{BLACK, BLACK, WHITE, BLACK, BLACK, WHITE, BLACK, BLACK, BLACK, BLACK}

# '{WHITE, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, WHITE, BLACK, BLACK}

# '{WHITE, BLACK, BLACK, WHITE, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK}

# '{BLACK, BLACK, BLACK, WHITE, WHITE, BLACK, BLACK, BLACK, BLACK, BLACK}

# '{BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK}

# '{BLACK, WHITE, BLACK, BLACK, WHITE, WHITE, BLACK, WHITE, BLACK, BLACK}

# '{BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, WHITE, BLACK, BLACK}

# '{BLACK, WHITE, BLACK, WHITE, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK}

# Num of pixels=WHITE = 17

*See Chap\_6\_Randomization/exercise8 for complete solution.*

1. Create a class, *StimData*, containing an array of integer samples. Randomize the size and contents of the array, constraining the size to be between 1 and 1000. Test the constraint by generating 20 transactions and reporting the size.

*Solution:*

package my\_package;

class StimData;

rand int data[];

constraint c {data.size() inside {[1:1000]}; }

endclass

endpackage

# Created an 804 element array

# Created an 699 element array

# Created an 66 element array

# Created an 843 element array

# Created an 13 element array

# Created an 858 element array

# Created an 46 element array

# Created an 253 element array

# Created an 136 element array

# Created an 972 element array

# Created an 497 element array

# Created an 914 element array

# Created an 358 element array

# Created an 299 element array

# Created an 761 element array

# Created an 179 element array

# Created an 384 element array

# Created an 284 element array

# Created an 8 element array

# Created an 349 element array

*See Chap\_6\_Randomization/exercise9 for complete solution.*

1. Expand the *Transaction* class below so back-to-back transactions of the same type do not have the same address. Test the constraint by generating 20 transactions.

package my\_package;

typedef enum {READ, WRITE} rw\_e;

class Transaction;

rw\_e old\_rw;

rand rw\_e rw;

rand bit [31:0] addr, data;

constraint rw\_c{if (old\_rw == WRITE) rw != WRITE;};

function void post\_randomize;

old\_rw = rw;

endfunction

function void print\_all;

$display("addr = %d, data = %d, rw = %s", addr, data, rw);

endfunction

endclass

endpackage

*Solution:*

*Add to class Transaction the following code. Note that a constraint for* ***rw****=WRITE is not required because back to back writes are already not allowed.*

bit [1:0] old\_addr;

constraint addr\_c{if (old\_rw == rw) old\_addr != addr;}

// or

constraint addr\_c{if (old\_rw == READ) old\_addr != addr;}

*Add to the post\_randomize function in the class the following code:*

old\_addr = addr;

*See Chap\_6\_Randomization/exercise10 for complete solution.*

1. Expand the *RandTransaction* class below so that back-to-back transactions of the same type do not have the same address. Test the constraint by generating 20 transactions.

class Transaction;

rand rw\_e rw;

rand bit [31:0] addr, data;

endclass

class RandTransaction;

rand Transaction trans\_array[];

constraint rw\_c {foreach (trans\_array[i])

if ((i>0) && (trans\_array[i-1].rw == WRITE))

trans\_array[i].rw != WRITE;}

function new();

trans\_array = new[TESTS];

foreach (trans\_array[i])

trans\_array[i] = new();

endfunction;

endclass

*Solution:*

*Add to class RandTransaction the following code. Note that a constraint for* ***rw****=WRITE is not required because back to back writes are already not allowed.*

constraint addr\_c {foreach (trans\_array[i])

if ((i>0) && (trans\_array[i-1].rw == READ)) trans\_array[i-1].addr != trans\_array[i].addr;}

*See Chap\_6\_Randomization/exercise11 for complete solution.*

# Solution to Exercises for Chap 7 Threads and Interprocess Communication

1. For the following code determine the order and time of execution for each statement if a *join* or *join\_none* or *join\_any* is used. Hint: the order and time of execution between the *fork* and *join/join\_none/join\_any* is the same, only the order and execution time of the statements after the join are different.

initial begin

$display("@%0t: start fork...join example", $time);

fork

begin

#20 $display("@%0t: sequential A after #20", $time);

#20 $display("@%0t: sequential B after #20", $time);

end

$display("@%0t: parallel start", $time);

#50 $display("@%0t: parallel after #50", $time);

begin

#30 $display("@%0t: sequential after #30", $time);

#10 $display("@%0t: sequential after #10", $time);

end

*join or join\_any or join\_none*

$display("@%0t: after join", $time);

#80 $display("@%0t: finish after #80", $time);

end

*Solution: Note that if execution occurs in parallel it will have the same execution order number.*

*Same regardless of whether a join/join\_none/join\_any is used:*

|  |  |  |
| --- | --- | --- |
| **Execution Order** | **Execution Time** | **Statement** |
| 1 | 0 | $display("@%0t: start fork...join example", $time); |
| 3 | 20 | #20 $display("@%0t: sequential A after #20", $time); |
| 5 | 40 | #20 $display("@%0t: sequential B after #20", $time); |
| 2 | 0 | $display("@%0t: parallel start", $time); |
| 6 | 50 | #50 $display("@%0t: parallel after #50", $time); |
| 4 | 30 | #30 $display("@%0t: sequential after #30", $time); |
| 5 | 40 | #10 $display("@%0t: sequential after #10", $time); |

*For join*

|  |  |  |
| --- | --- | --- |
| **Execution Order** | **Execution Time** | **Statement** |
| 7 | 5 | $display("@%0t: after join", $time); |
| 8 | 130 | #80 $display("@%0t: finish after #80", $time); |

*For join\_none*

|  |  |  |
| --- | --- | --- |
| **Execution Order** | **Execution Time** | **Statement** |
| 2 | 0 | $display("@%0t: after join", $time); |
| 7 | 80 | #80 $display("@%0t: finish after #80", $time); |

*For join\_any*

|  |  |  |
| --- | --- | --- |
| **Execution Order** | **Execution Time** | **Statement** |
| 3 | 0 | $display("@%0t: after join", $time); |
| 7 | 80 | #80 $display("@%0t: finish after #80", $time); |

*See Chap\_7\_Threads\_and\_Interprocess\_Communication/exercise1 for complete solution.*

1. For the following code what would the output be with and without a *wait fork* inserted in the indicated location?

initial begin

fork

transmit(1);

transmit(2);

join\_none

fork: receive\_fork

receive(1);

receive(2);

join\_none

// What is the output with/without wait fork inserted here?

#15ns disable receive\_fork;

$display("%0t: Done", $time);

end

task transmit(int index);

#10ns;

$display("%0t: Transmit is done for index = %0d", $time, index);

endtask

task receive(int index);

#(index \* 10ns);

$display("%0t: Receive is done for index = %0d", $time, index);

endtask

*Solution:*

*With wait fork execution only resumes when all child threads are complete:*

*# 10: Receive is done for index = 1 // Order is indeterminate*

*# 10: Transmit is done for index = 1 // Order is indeterminate*

*# 10: Transmit is done for index = 2 // Order is indeterminate*

*# 20: Receive is done for index = 2*

*# 35: Done*

*Without wait fork the transmit and receive processes are all spawned at time 0 and execution continues to the disable #15ns receive\_fork line.*

*# 10: Receive is done for index = 1 // Order is indeterminate*

*# 10: Transmit is done for index = 1 // Order is indeterminate*

*# 10: Transmit is done for index = 2 // Order is indeterminate*

*# 15: Done*

*See Chap\_7\_Threads\_and\_Interprocess\_Communication/exercise2 for complete solution.*

1. What would be displayed with the following code? Assume that the events and task *trigger* is declared inside a program declared as automatic.

event e1, e2;

task trigger(event local\_event, input time wait\_time);

#wait\_time;

->local\_event;

endtask // void

initial begin

fork

trigger(e1, 10ns);

begin

wait(e1.triggered());

$display("%0t: e1 triggered", $time);

end

join

end

initial begin

fork

trigger(e2, 20ns);

begin

wait(e2.triggered());

$display("%0t: e2 triggered", $time);

end

join

end

*Solution:*

*# 10: e1 triggered*

*# 20: e2 triggered*

*See Chap\_7\_Threads\_and\_Interprocess\_Communication/exercise3 for complete solution.*

1. Create a task called *wait10* that for 10 tries will wait for 10ns and then check for 1 semaphore key to be available. When the key is available, quit the loop and print out the time.

*Solution:*

task wait10;

int i= 10;

do begin

#10ns;

i--;

end

while ((i!= 0) && !sem.try\_get); // Didn’t get key, keep trying

$display("%0t: Done at i=%0d", $time, i);

endtask // wait10

*See Chap\_7\_Threads\_and\_Interprocess\_Communication/exercise4\_5 for complete solution.*

1. What would be displayed with the following code that calls the task from Exercise 4?

initial begin

fork

begin

sem = new(1);

sem.get(1);

#45ns;

sem.put(2);

end

wait10();

join

end

*Solution:*

*# 50: Done at i=5*

*See Chap\_7\_Threads\_and\_Interprocess\_Communication/exercise4\_5 for complete solution.*

1. What would be displayed with the following code?

`default\_nettype none

program automatic test;

mailbox mbx;

int value;

initial begin

mbx = new(1);

$display("mbx.num()=%0d", mbx.num());

$display("mbx.try\_get= %0d", mbx.try\_get(value));

mbx.put(2);

$display("mbx.try\_put= %0d", mbx.try\_put(value));

$display("mbx.num()=%0d", mbx.num());

mbx.peek(value);

$display("value=%0d", value);

end

endprogram

*Solution:*

*# mbx.num()=0 // Nothing put in mailbox*

*# mbx.try\_get= 0 // Nothing put in mailbox so 0 returned but execution continues*

*# mbx.try\_put= 0 // No room to put so 0 returned but excution continues*

*# mbx.num()=1 // 1 integer in mailbox*

*# value=2*

*See Chap\_7\_Threads\_and\_Interprocess\_Communication/exercise6 for complete solution.*

1. Look at Figure 7-8 “Layered testbench with environment” and create the Monitor class. You can make the following assumptions.
2. The *Monitor* class has knowledge of class *OutputTrans* with member variables *out1* and *out2*.
3. The DUT and *Monitor* are connected with an interface called *my\_bus* with signals *out1* and *out2*.
4. The interface *my\_bus* has a clocking block *cb*.
5. On every active clock edge, the *Monitor* class will sample the DUT outputs, *out1* and *out2*, assign them to an object of type *OutputTrans*, and place the object in a mailbox.

*Solution: See code in Chap\_7\_Threads\_and\_Interprocess\_Communication/exercise7*

# Solution to Exercises for Chap 8 Advanced OOP and Testbench Guidelines

1. Given the following class, create a method in an extended class *ExtBinary* that multiplies *val1* and *val2* and returns an integer.

class Binary;

bit [3:0] val1, val2;

function new(input bit [3:0] val1, val2);

this.val1 = val1;

this.val2 = val2;

endfunction

virtual function void print\_int(input int val);

$display("val=0d%0d", val);

endfunction

endclass

*Solution:*

class ExtBinary extends Binary;

function new(input bit [3:0] val1, val2);

super.new(val1, val2);

endfunction

function int mult();

mult = val1\*val2;

endfunction

endclass

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise1\_2 for complete solution.*

1. Starting with the solution to Exercise 1, use the *ExtBinary* class to initialize *val1=15, val2=8*, and print out the multiplied value.

*Solution:*

ExtBinary mc;

initial begin

mc = new(15, 8);

mc.print\_int(mc.mult());

end

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise1\_2 for complete solution.*

1. Starting with the solution to Exercise 1, create an extended class *Exercise3* that constrains *val1* and *val2* to be less than 10.

*Solution: Add the following constraint to class ExtBinary from Exercise 1.*

constraint less\_than\_10 {

val1 < 10;

val2 < 10;

}

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise3\_4 for complete solution.*

1. Starting with the solution to Exercise 3, use the *Exercise3* class to randomize *val1* and *val2*, and print out the multiplied value.

*Solution:*

ExtBinary mc;

initial begin

mc = new(15, 8);

assert(mc.randomize());

mc.print\_int(mc.mult());

end

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise3\_4 for complete solution.*

1. Given the classes in Exercise 1 and 2, and the following declarations, what will be the result of executing code snippets a-d?

ExtBinary mc, mc2;

Binary b;



mc = new(15, 8);

b = mc;



b = new(15, 8);

mc = b;



mc = new(15, 8);

b = mc;

mc2 = b;



mc = new(15, 8);

b = mc;

mc2 = b;

if(!$cast(mc2, b))

$display("ERROR: cannot assign");

else

$display(“Success”);

*Solution:*

1. *Handle* ***b*** *will point to object* ***mc.*** *Handle* ***mc2*** *will point to null.*
2. *Compile error*
3. *Compile error*
4. *Handle* ***b*** *and* ***mc2*** *will point to object* ***mc****. The string Success will print.*

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise5 for complete solution.*

1. Given the class in Exercise 1 and the following copy function for class *Binary* create a copy function for the extended class.

virtual function Binary copy();

copy = new(15,8);

copy.val1 = val1;

copy.val2 = val2;

return copy;

endfunction

*Solution:*

virtual function Binary copy();

ExtBinary mc;

mc = new(15,8);

mc.val1 = val1;

mc.val2 = val2;

return mc;

endfunction

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise6\_7 for complete solution.*

1. From the solution to Exercise 6, use the copy function to copy the object pointed to by the extended class handle *mc* to the extended class handle *mc2*.

*Solution: Note that* ***mc2 = mc.copy()*** *will result in a compile error because mc.copy returns a handle to an object of type* ***Binary****, not* ***ExtBinary****.*

if (!$cast(mc2, mc.copy()))

$display("ERROR: cannot assign");

else

$display("Success");

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise6\_7 for complete solution.*

1. Using code Samples 8-26 to 8-28 in section 8.7.1 and 8.7.2 of the text add the ability to randomly delay a transaction between 0 and 100ns.

*Solution:*

*Define the following class:*

class Driver\_cbs\_delay extends Driver\_cbs;

virtual task pre\_tx(ref Transaction tr, ref bit drop);

#($urandom\_range(0,100));

endtask

endclass

*Add the following to the begin/end block inside the initial in the program:*

Driver\_cbs\_delay delay\_cb = new();

env.drv.cbs.push\_back(delay\_cb);

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise8 for complete solution.*

1. Create a class that can compare any data type using the case equality operator, ===. It contains a compare function that returns a 1 if the two values match, 0 otherwise. By default it compares two 4-bit data types.

*Solution:*

class comparator #(type T=bit[3:0]);

function bit compare(input T expected, input T actual);

compare=1;

if (expected !== actual) begin

$display("Expected=0x%0h != actual=0x%0h", expected, actual);

compare=0;

end

endfunction

endclass

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise9\_10 for complete solution.*

1. Using the solution from Exercise 9, use the comparator class to compare two 4-bit values, *expected\_4bit* and *actual\_4bit*. Next, compare two values of type *color\_t, expected\_color* and *actual\_color*. Increment an error counter if an error occurs.

*Solution:*

comparator compare\_4bit;

comparator #(color) compare\_color;

initial begin

compare\_4bit = new();

compare\_color = new();

if (!(compare\_4bit.compare(expected\_4bit, actual\_4bit))) error++;

if (!(compare\_color.compare(expected\_color, actual\_color))) error++;

end

*See Chap\_8\_Advanced\_OOP\_and\_Testbench\_Guidelines/exercise9\_10 for complete solution.*

# Solution to Exercises for Chap 9 Functional Coverage

1. For the class below, write a covergroup to collect coverage on the test plan requirement “All opcode’s of the ALU must be tested”. Assume the opcodes are valid on the positive edge of signal *clk*.

typedef enum {ADD, SUB, MULT, DIV} opcode\_e;

class Transaction;

rand opcode\_e opcode;

rand byte operand1;

rand byte operand2;

endclass

Transaction tr;

*Solution:*

covergroup CovCode@(posedge clk);

coverpoint tr.opcode;

endgroup

*See Chap\_9\_Functional\_Coverage/exercise1 for complete solution.*

1. Expand the solution to Exercise 1 to to cover the test plan requirement, “Operand1 shall take on the values maximum negative (-128), zero, and maximum positive (127).” Define a coverage bin for each of these values as well as a default bin. Label the coverpoint *operand1\_cp*.

*Solution:*

covergroup CovCode @ifc.cb;

operand1\_cp: coverpoint tr.operand1{

bins max\_neg = {-128};

bins zero = {0};

bins max\_pos = {127};

bins misc = default;

}

endgroup

*See Chap\_9\_Functional\_Coverage/exercise2 for complete solution.*

1. Expand the solution to Exercise 2 to cover the following test plan requirements:
   1. “The opcode shall take on the values ADD or SUB” (hint: this is 1 coverage bin).
   2. “The opcode shall take on the values ADD followed by SUB” (hint: this is a second coverage bin).

Label the coverpoint *opcode\_cp*

*Solution: Add to the covergroup the following coverpoint.*

opcode\_cp: coverpoint tr.opcode{

bins add\_sub = {ADD, SUB};

bins add\_then\_sub = (ADD=>SUB);

}

*See Chap\_9\_Functional\_Coverage/exercise3\_4 for complete solution.*

1. Expand the solution to Exercise 3 to cover the test plan requirement, “Opcode must not equal DIV” (hint: report an error using *illegal\_bins*).

*Solution: Add to the covergroup the following coverpoint.*

coverpoint tr.opcode{

illegal\_bins no\_div = {DIV};

}

*See Chap\_9\_Functional\_Coverage/exercise3\_4 for complete solution.*

1. Expand the solution to Exercise 4 to collect coverage on the test plan requirement, “The opcode shall take on the values ADD or SUB when operand1 is maximum negative or maximum positive value.” Weight the cross coverage by 5.

Solution: *Add to the covergroup the following coverpoint.*

// Cross of coverpoints opcode\_cp with operand1\_cp will create 6 bins:

// <add\_sub, max\_neg>, <add\_sub, max\_pos>, <add\_sub, zero>,

// <add\_then\_sub, max\_neg>, <add\_then\_sub, max\_pos>, <add\_then\_sub, zero>

opcode\_operand1: cross opcode\_cp, operand1\_cp {

// Remove bins <add\_sub, zero> and <add\_then\_sub, zero>

ignore\_bins operand1\_zero = binsof(operand1\_cp.zero);

// or

// ignore\_bins operand1\_zero = binsof(operand1\_cp) intersect{0};

// Remove bins <add\_then\_sub, max\_neg> and <add\_then\_sub, max\_pos>,

ignore\_bins opcode\_add\_then\_sub = binsof(opcode\_cp.add\_then\_sub);

// Only bins <add\_sub, max\_neg> and <add\_sub, max\_pos> remain

option.weight = 5;

}

*See Chap\_9\_Functional\_Coverage/exercise5 for complete solution.*

1. Assuming that your covergroup is called *CovCode* and the instantiation name of the covergroup is *ck* expand Exercise 4 to:
   1. Display the coverage of coverpoint *operand1\_cp* referenced by the instantiation name.
   2. Display the coverage of coverpoint *opcode\_cp* referenced by the covergroup name.

Solution:

$display("%t: Coverpoint ck.operand1\_cp coverage is %f", $time, ck.operand1\_cp.get\_coverage());

$display("%t: Covergroup CovCode::opcode\_cp is %f", $time, CovCode::opcode\_cp.get\_coverage());

*See Chap\_9\_Functional\_Coverage/exercise6 for complete solution.*

# Solution to Exercises for Chap 10 Advanced Interfaces

1. Complete the following code, as indicated by the comments.

class Driver;

...

// Declare a virtual interface for the DUT

function new(input mailbox #(Instruction) agt2drv,

/\* complete the argument list \*/ );

this.agt2drv = agt2drv;

// Save the virtual interface argument in the class-level variable

endfunction

endclass

class Environment;

Driver drv;

.....

drv = new(agt2drv, /\* complete the argument list \*/);

.....

endclass

*Solution:*

class Driver;

.......

virtual risc\_spm\_if risc\_bus; // Declare a virtual interface for the DUT

function new(input mailbox #(Instruction) agt2drv, input virtual risc\_spm\_if risc\_bus); // complete the argument list

this.agt2drv = agt2drv;

// Save the virtual interface argument in the class-level variable

this.risc\_bus = risc\_bus;

endfunction

endclass

class Environment;

Driver drv;

…..

drv = new(agt2drv, risc\_bus); // complete the argument list

……

endclass

*See Chap\_10\_Advanced\_Interfaces/exercise1\_2 for complete solution*

1. Using the solution to Exercise 1, complete the following code where indicated by the comments.

program automatic test(risc\_spm\_if risc\_bus);

import my\_package::\*;

Environment env;

initial begin

// Create object referenced by env handle

end

endprogram

*Solution:*

program automatic test(risc\_spm\_if risc\_bus);

import my\_package::\*;

Environment env;

initial begin

env = new(risc\_bus); // Create object referenced by env handle

end

endprogram

*See Chap\_10\_Advanced\_Interfaces/exercise1\_2 for complete solution*

1. Modify the following program declaration to use cross module references (XMR). Assume the top module that contains the interface is named *top*.

program automatic test(risc\_spm\_if risc\_bus);

...

endprogram

Modify the following instantiation of program *test* to use cross module references (XMR).

`include "risc\_spm\_if.sv"

module top;

....

test t1(risc\_bus);

....

endmodule

*Solution:*

program automatic test(); // Do not need to pass in interface

....

virtual risc\_spm\_if risc\_bus = top.risc\_bus; // Interface is known to program with XMR.

endprogram

`include "risc\_spm\_if.sv"

module top;

....

test test(); // Do not need to pass in interface

....

endmodule

*See Chap\_10\_Advanced\_Interfaces/exercise3 for complete solution*

1. Expand the solution to Exercise 3 to create *NUM\_RISC\_BUS* environments and create *NUM\_RISC\_BUS* interfaces.

*Solution: See Chap\_10\_Advanced\_Interfaces/exercise4 for solution*

1. Expand the solution to Exercise 3 to use a *typedef* for the virtual interface.

Solution:

typedef virtual risc\_spm\_if risc\_spm\_if\_v;  
program automatic test();

....

risc\_spm\_if\_v risc\_bus = top.risc\_bus;

....

endprogram

*See Chap\_10\_Advanced\_Interfaces/exercise5 for complete solution*

1. Modify the following interface to use a parameter, *ADDRESS\_WIDTH*. By default the addressing space supports 256 words.

interface risc\_spm\_if (input bit clk);

bit rst;

bit [7:0] data\_out;

logic [7:0] address;

logic [7:0] data\_in;

logic write;

modport DUT (input clk, data\_out,

output address, data\_in, write);

endinterface

*Solution:*

interface risc\_spm\_if #(ADDRESS\_WIDTH=8) (input bit clk);

bit rst;

bit [7:0] data\_out;

logic [ADDRESS\_WIDTH-1:0] address;

logic [7:0] data\_in;

logic write;

modport DUT (input clk, data\_out,

output address, data\_in, write);

endinterface

*See Chap\_10\_Advanced\_Interfaces/exercise6 for complete solution*

# Solution to Exercises for Chap 11 A Complete SystemVerilog Testbench

1. In Sample 11-2, why is *clk* not passed into the port list of program *test*?

*Solution: Signal* ***clk*** *not passed into the port list of program test because the testbench only uses clocking blocks in the interfaces to synchronize the DUT, not low level clocks.*

1. In Sample 11-6, could *numRx* be substituted for *Rx.size()?* Why or why not?

*Solution: Yes, because the size of array* ***Rx*** *is defined in module top to be of size 0:****numRx****-1.*

1. For the following code snippet from Sample 11-6, explain what is being created for each statement.

function void Environment::build();

cpu = new(mif, cfg);

gen = new[numRx];

drv = new[numRx];

gen2drv = new[numRx];

drv2gen = new[numRx];

scb = new(cfg);

cov = new();

foreach(gen[i]) begin

gen2drv[i] = new();

gen[i] = new(gen2drv[i], drv2gen[i],cfg.cells\_per\_chan[i], i);

drv[i] = new(gen2drv[i], drv2gen[i], Rx[i], i);

end

…..

*Solution: In this code snippet the following objects, dynamic arrays, mailboxes, and events are being created:*

1. *Statement* ***cpu = new(mif, cfg)*** *creates an object of class* ***CPU\_driver****. The* ***mif*** *interface and* ***cpu*** *object handle are passed to the custom constructor. Handle* ***cpu*** *points to the object.*
2. *Statement* ***gen = new[numRx]*** *allocates* ***numRx*** *locations in dynamic array* ***gen*** *for handles to objects of class* ***UNI\_generator****.*
3. *Statement* ***drv = new[numRx]*** *allocates* ***numRx*** *locations in dynamic array* ***drv*** *for handles to objects of class* ***Driver****.*
4. *Statement* ***gen2drv = new[numRx]*** *allocates* ***numRx*** *locations in dynamic array* ***gen2drv*** *for handles to mailboxes.*
5. *Statement* ***drv2gen = new[numRx]*** *allocates* ***numRx*** *locations in dynamic array* ***drv2gen*** *for events.*
6. *Statement* ***scb = new(cfg)*** *creates an object of class* ***Scoreboard*** *pointed to by handle* ***scb****. Handle* ***cfg*** *to an object of class* ***Config*** *is passed to the custom contructor.*
7. *Statement* ***cov = new()*** *creates an object of class* ***Coverage*** *pointed to by handle* ***cov****.*
8. *Statement* ***foreach(gen[i])*** *begin loops through dynamic array* ***gen****. For every element in the dynamic array:*
   1. *Statement* ***gen2drv[i] = new()*** *creates a mailbox pointed to by handle* ***gen2drv[i]****.*
   2. *Statement* ***gen[i] = new(gen2drv[i], drv2gen[i],cfg.cells\_per\_chan[i], i)*** *creates an object of class* **UNI\_generator** *pointed to by handle* ***gen[i].*** *Handles to mailboxes(gen2drv[i]), handles to an events(drv2gen[i]), the number cells per channel randomly generated in the* ***cfg*** *object(cfg.cells\_per\_chan[i]), and the* ***PortID*** *(i) is passed to the custom constructor.*
   3. *Statement* ***drv[i] = new(gen2drv[i], drv2gen[i], Rx[i], i);*** *creates an object of class* **Driver** *pointed to by handle* ***drv[i].*** *Handles to mailboxes(gen2drv[i]), handles to events(drv2gen[i]), handles to interfaces(Rx[i]) , and the* ***PortID*** *(i) is passed to the custom constructor.*
9. In Sample 11-9 what coverage object does the handle *cov* point to?

*Solution: Handle* ***cov*** *points to an object of class* ***Coverage*** *created in the* ***Environment*** *class.*

1. In Sample 11-17, function *UNI\_cell::copy* assumes that the handle to the object *UNI\_cell* points to an object of class *UNI\_cell* as depicted in the following drawing. Draw what object the handle *dst* points to for the following function calls.



* 1. *copy()*
  2. *copy(handle);*

Solution:

1. *A call to function copy with no arguments will create a new object of class* ***UNI\_cell*** *pointed to by handle* ***dst*** *as depicted in the following drawing:*



1. *A call to function copy with an argument will simply point* ***dst*** *to the same object pointed to as pointed to by <handle to object UNI\_cell> as depicted in the following drawing:*



1. In Sample 11-18, why is the *$cast()* required?

*Solution: Cast is required because the return type of* ***copy*** *is* ***BaseTr****. The type of* ***cell*** *is* ***UNI\_cell*** *which is an extended class of* ***BaseTr****. If cast was not used a compiler error would result.*

1. In Sample 11-19 and 11-20, why is the *typedef* declarations needed?

Solution: Answer, because Driver\_cbs requires knowledge of the Driver class and the driver class requires knowledge of the Driver\_cbs class.

1. In Sample 11-19, why is *peek()* used first and then later a *get()?*

*Solution: This is as a method of synchronization. An object in the mailbox is looked at and used but not removed until the transmit is complete. This method is described further in Section 7.6.4 of the text.*

1. In Sample 11-23, is the error message *“ …cell not found…”* in the function *check\_actual* printed every time it is called? Why or why not?

*Solution: The error report is not executed every time function* ***check\_actual*** *is called because the* ***foreach*** *loop causes a return from the function using the* ***return*** *statement if a match is found.*

1. Why do classes *Environment, Scoreboard*, and *CPU\_driver* all define a handle to class *Config*? Are 3 objects of class *Config* created?

*Solution: Every class defines a handle to class Config so that every class is aware of the configuration. The handle points the single configuration object created in class Environment.*

# Solution to Exercises for Chap 12 Interfacing with C

1. Create a C function, *shift\_c*, that has two input arguments: a 32-bit unsigned input value *i* and an integer for the shift amount *n*. The input *i* is shifted *n* places. When *n* is positive, values are shifted left, when *n* is negative, shifted right, and when *n* is 0, no shift is performed. The function returns the shifted value. Create a SystemVerilog module that calls the C function and tests each feature. Provide the output.

*Solution:*

#include <stdio.h>

#include <svdpi.h>

int shift\_c(const svBitVecVal \*i, int n)

{

if (n < 0)

return (\*i >> (n \*-1));

else if (n > 0)

return (\*i << n);

else

return \*i;

}

module shift;

import "DPI-C" context function int shift\_c(input bit[31:0] i, input int n);

reg [31:0] val;

int shift;

reg [31:0] shifted;

initial begin

val = 1; shift = 1;

shifted = shift\_c(val, shift);

$display("0d%0d shifted by 0d%0d = 0d%0d", val, shift, shifted);

val = 2; shift = -1;

shifted = shift\_c(val, shift);

$display("0d%0d shifted by 0d%0d = 0d%0d", val, shift, shifted);

val = 2; shift = 0;

shifted = shift\_c(val, shift);

$display("0d%0d shifted by 0d%0d = 0d%0d", val, shift, shifted);

$finish;

end

endmodule

*Output:*

# 0d1 shifted by 0d1 = 0d2

# 0d2 shifted by 0d-1 = 0d1

# 0d2 shifted by 0d0 = 0d2

*See Chap12\_Interfacing\_with\_C/exercise1 for complete solution.*

1. Expand Exercise 1 to add a third argument to *shift\_c*, a load flag *ld*. When *ld* is true, *i* is shifted by *n* places and then loaded into an internal 32-bit register. When *ld* is false, the register is shifted *n* places. The function returns the value of the register after these operations. Create a SystemVerilog module that calls the C function and tests each feature. Provide the output.

*Output:*

# loading

# 0d1 shifted by 0d1 = 0d2

# 0d2 shifted by 0d-1 = 0d1

# 0d2 shifted by 0d0 = 0d2

# Not loading

# 0d255 shifted by 0d2 = 0d8

# 0d1023 shifted by 0d-1 = 0d4

# 0d2047 shifted by 0d0 = 0d4

*See Chap12\_Interfacing\_with\_C/exercise2 for complete solution.*

1. Expand Exercise 2 to create multiple instances of the *shift\_c* function. Each instance in C needs a unique identifier, so use the address where the internal register is stored. Print this address along with the arguments when the function *shift\_c* is called. Instantiate the function twice, and call each instance twice. Provide the output.

*Output:*

# loading

# Instance address 156031392: 1 shifted by 1 = 2, load = 1

# Instance address 156029328: 2 shifted by -1 = 1, load = 1

# Instance address 156031392: 2 shifted by 0 = 2, load = 1

# Instance address 156029328: 2 shifted by 2 = 8, load = 1

# Not loading

# Instance address 156031392: 255 shifted by 1 = 4, load = 0

# Instance address 156029328: 511 shifted by -3 = 1, load = 0

# Instance address 156031392: 1023 shifted by 0 = 4, load = 0

# Instance address 156029328: 2047 shifted by 1 = 2, load = 0

*Solution: See Chap12\_Interfacing\_with\_C/exercise3*

1. Expand the C code from Exercise 3 to display the total number of times the *shift\_c* function has been called, even if the function is instantiated more than once.

Solution:

*Output:*

# loading

# Instance address 156031392, function call 0: 1 shifted by 1 = 2, load = 1

# Instance address 155781216, function call 0: 2 shifted by -1 = 1, load = 1

# Instance address 156031392, function call 1: 2 shifted by 0 = 2, load = 1

# Instance address 155781216, function call 1: 2 shifted by 2 = 8, load = 1

# Not loading

# Instance address 156031392, function call 2: 255 shifted by 1 = 4, load = 0

# Instance address 155781216, function call 2: 511 shifted by -3 = 1, load = 0

# Instance address 156031392, function call 3: 1023 shifted by 0 = 4, load = 0

# Instance address 155781216, function call 3: 2047 shifted by 1 = 2, load = 0

*Solution: See Chap12\_Interfacing\_with\_C/exercise4*

1. Expand Exercise 4 to provide the ability to initialize the stored value at instantiation.

*Output:*

# Instance address 156005928: initialized internal\_reg to 512

# Instance address 156031512: initialized internal\_reg to 2048

# Not loading

# Instance address 156005928, function call 0: 512 shifted by 1 = 1024, load = 0

# Instance address 156031512, function call 0: 2048 shifted by -3 = 256, load = 0

# Instance address 156005928, function call 1: 1024 shifted by 0 = 1024, load = 0

# Instance address 156031512, function call 1: 256 shifted by 1 = 512, load = 0

*Solution: See Chap12\_Interfacing\_with\_C/exercise5*

1. Expand Exercise 5 to encapsulate the *shift\_c* function in a class.

*Output:*

# Instance address 156035112: initialized internal\_reg to 512

# Instance address 156035016: initialized internal\_reg to 2048

# Not loading

# Instance address 156035112 function call 0: 512 shifted by 1 = 1024, load = 0

# Instance address 156035016 function call 0: 2048 shifted by -3 = 256, load = 0

# Instance address 156035112 function call 1: 1024 shifted by 0 = 1024, load = 0

# Instance address 156035016 function call 1: 256 shifted by 1 = 512, load = 0

*Solution: See Chap12\_Interfacing\_with\_C/exercise6*

1. For the code in Sample-24 and Sample 12-25 what is returned by the following open array methods?

svLeft(h, 1));

svLeft(h, 2));

svRight(h, 1));

svRight(h, 2));

svSize(h, 1));

svSize(h, 2));

svDimensions(h));

svSizeOfArray(h));

Solution:

svLeft(h, 1)); returns 6

svLeft(h, 2)); returns 8

svRight(h, 1)); returns 1

svRight(h, 2)); returns 3

svSize(h, 1)); returns 6

svSize(h, 2)); returns 6

svDimensions(h)); returns 2

svSizeOfArray(h)); returns 144

*Solution: See Chap12\_Interfacing\_with\_C/exercise7 for complete solution.*

1. Modify Exercise 1 so that instead of shifting the value in C, the function calls an exported SystemVerilog void function named *shift\_sv* that does the shifting.

*Solution: See Chap12\_Interfacing\_with\_C/exercise8 for complete solution.*

1. Expand Exercise 8 to call the SystemVerilog function *shift\_sv* for two different SystemVerilog objects as demonstrated in Section 12.8.4 of the text. Assume the SystemVerilog function *shift\_build* has been exported to the C code.

*Solution: See Chap12\_Interfacing\_with\_C/exercise9 for complete solution.*

1. Expand Exercise 8 to:
   1. Create a SystemVerilog class *Shift* containing the function *shift\_sv* that stores the result in a class-level variable, and a *shift\_print* function that displays the stored result.
   2. Define and export SystemVerilog function *shift\_build*.
   3. Support the creation of multiple *Shift* objects with the handles to these objects stored in a queue.
   4. Create a testbench that constructs multiple *Shift* objects. Demonstrate that each object holds a separate result after performing calculations.

*Output:*

# 1 shifted by 1 = 2

# 2 shifted by -1 = 1

# 2 shifted by 0 = 2

# 512 shifted by 1 = 1024

# 1024 shifted by -1 = 512

# 2048 shifted by 0 = 2048

*Solution: See Chap12\_Interfacing\_with\_C/exercise10 for complete solution.*