**SV Lab-1**

Declare variables of following types and perform operations as defined below:

bit, 8-bit, 8-bit signed logic, int, integer, unsigned integer, byte, shortint and longint.

* Print default value of all variables.
* Fill all locations of every variable with 1, print result and then perform arithmetic right shift and then print result again.
* Assign value 8’b10X1\_ZX10 to 8-bit variable and observe the result.
* Verify whether we can assign values using continuous assignment.
* Declare a 2-D (4 x 8) packed array of bits, initialize it with random value. Try to access this array in different combinations and store results in temporary variables.
* Define a dynamic array where each element is capable of storing 10-bits. Allocate 20 memory locations to this array. Randomize values at all these locations and print array size. Now double the size of array by keeping contents of initial 20 locations same as previous. Assign value to few new locations and then print content and size of array.

**SV Lab-2**

* Declare an array of int which has 10 locations. Initialize the array with data of your choice, write an algorithm to sort this array and compare your result with predefined sort method.
* Declare an integer queue. Initialize it with five elements (0 to 4). Perform following operations on queue and print content after every operation:
* Insert an element (value 19) at beginning without any method.
* Insert an element (value 29) at beginning using predefined method.
* Insert an element (value 39) at the end using predefined method.
* Insert an element (value 49) at index 4.
* Get 1st element in queue and compare with value 29.
* Get last element in queue and compare with value 39.
* Try to get value at index 3 without removing it from queue.
* Delete an element at index 4 in queue.
* Design a synchronous LIFO (Last In First Out) using bound queue. Queue has following specification:
* Inputs: clk, Data\_in, write\_en and read\_enable.
* Outputs: empty, full, Data\_out.
* Size: 16 locations each capable of storing 5 bits.
* Declare an associative array, assign 50 random integer values at random locations between 1 and 100 and perform following operations:
* Check whether value exists at location 3, 23, 75 and 80.
* Print the value at first index along with index number.
* Print the value at last index along with index number.
* Print the size of array.
* Delete elements at 5th, 10th, 15th and 100th index if they exist.
* Print the size of array again.
* Declare an associative array of int with index of type string. Use following string index’s to store their equivalent ASCII value (Decimal). “0”- “9”, “a”- “h”, “P”- “Z”
* Print all value of array along with its index.
* Check whether value exists at following index “x”, “A”, “p” and “T”.
* Print Num of elements in the array.
* Delete first 3 and last five elements of array.
* Print content of array and size of array.

**SV Lab-3**

Declare an enum of calendar with all months as its element. Assign value 5 to March and value 15 to September. Declare a variable of defined enum type and try following operations and print element name and its value after every operation :

* Try assigning integer value to enum-variable (try in range and out of range). Understand application of casting.
* Assign value June to enum-variable.
* Declare an int type variable and update it with value enum - variable + 3.
* Declare a string and initialize it with your complete name. Perform following operations:
* Print the size of string.
* Use substr method to print your Last Name.
* Convert string to upper case.
* Get 4rd character of the string and display both character and string.
* Put a character at 5th position and re-display the string.
* Delete 7th character present in the string.
* Try to print binary value of all the characters present in the string.
* Declare a string “12578\_123\_12abcd” try to extract 1257812312 value form string and store in some integer.
* Take an integer initialize it with 5678. Now store equivalent string representation of the value in a user defined string.

**SV Lab-4**

* Write a function that accepts a 1-D dynamic array of bytes and returns a queue. The function should process the array and should return a queue containing unique elements present in the array.
* Modify the above function two include two more arguments min\_limit and max\_limit. If only min\_limit is provided then it should return unique elements which have value more then min\_limit. If only max\_limit is provided then it should return unique elements which have value less then max\_limit. If both are provided then it should return unique elements whose value lies between min\_limit and max\_limit.
* Write a function (function name: add) and task (task name: add) that accepts two integer arguments and returns their sum. Try to define and call them in same code and analyze the problem. Once problem has been analyzed, use concept of package to perform this subprogram overloading.
* Design a 8x1 mux using logic and unique case statement, don’t include default statement. Use inside operator to print unknown selection inputs.
* Design a parameterized counter that accepts data type along with maximum count as a parameter. Counter has a load and enable pin. Verify you design by counting following:
* Months from (Jan to July).
* MOD 7 counter made of bits.
* Even numbers till 30. (Try).
* Write two tasks called as Increment1 and Increment2. Define two variables count1 (static) and count2 (automatic). Increment1 and Increment2 task contains for loop to increment count1 and count2 variable 8 times respectively. Use $display to display value on every count. Call these tasks several times to observe difference between static and automatic variables. Observe if there is any difference in result if these variables were defined inside their respective tasks instead of defining in module.

**SV Lab-5**

* Write a Test Bench and design for a 8-bit PISO shift register with following ports:
* 1-bit clock input.
* 8-bit parallel input.
* 1-bit serial output.
* 1-bit load input.
* 1-bit enable input.

Instantiate both test bench and PISO in the top module. Generate clock in the top module (using always block) and random data for all other inputs (using initial forever block). Use $display in test bench to display current data in PISO. Observe what difference it makes if test bench is defined using program block.

* Design a 5-bit counter with following ports:
* 1-bit clock input.
* 1-bit asynchronous reset.
* 1-bit load input
* 5-bit data input.
* 5-bit data output.

Connect this design to TB using concept of interface and mod ports. Monitor counter data inside interface block. Once verified create a copy of this code and perform following modifications and re-verify the design:

* Replace 5-bit data input and 5-bit data output by 5-bit data (inout port)
* Add 1-bit dir (input port), to specify the direction of data.
* Design a 4-bit counter with load, en and asynchronous reset pin. Write a clocking blocking to drive TB outputs 2ns after clock and sample TB inputs 1ns before clock.
* Include load and count in clocking block.
* Include load, en and count, sample load at negative edge of the clock.

Also verify use of @ and ## constructs of the clocking block.

**SV Lab-6**

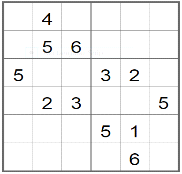
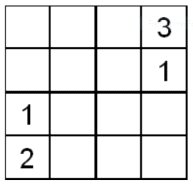
* Design a 4-bit magnitude comparator. Connect that to test bench with help of an interface. Write one task that generates random inputs for DUT and one which generates directed inputs for DUT. Call both the tasks in fork-join and control the flow in which inputs are fed. First provide random inputs for 100 time units (at interval of 3 time unit) then provide directed inputs for 20 time units (at interval of 2 time unit) and then switch back to random inputs (**using fine grain control**). Finish simulation at 1000 time units and use final block to print “end of simulation” banner along with time.
* Declare two events fork after a finite time. In one thread, assign value to first variable randomly, trigger ev\_1, and implement separate initial blocks for them. In one initial block, trigger the event after some delay and wait for the event triggering on consecutive code lines using @ event control. In second initial block, use wait() instead of @. Print the time using $display at the end in each of these initial blocks and observe the output. Add non-blocking event triggering to the same example and observe the output.
* Declare two int variables, two events ev\_1 and ev\_2 and two forever threads in your fork join\_none block. Disable wait for ev\_2 and print the value of second variable, while in another thread, assign value to second variable randomly, trigger ev\_2, wait for ev\_1 and print the value of first variable Observe the output using relevant $display statements while triggering events and waiting for events. Give different delay values in both the threads.
* Design a 4-bit synchronous Full Adder (inputs a, b (4-bits) and c (1-bit), output sum (5-bits)). Define interface with modports DUT (for design), TB (for Test Bench) and MON (use for monitoring inputs, predicting result and comparing with current output). In top module instantiate all three codes (i.e. TB, DUT and MON). Generate and drive random data to DUT using a TB. A new data will be generated on every clock cycle (each combination of a, b and c should be unique). Generate 300 such unique combinations. In MON write a forever loop which samples value of a, b and c on every positive clock edge. MON predicts output based on sampled value and compares it with output generated by DUT. Use a task to perform comparison of predicted and obtained output (sum). Make sure MON works for every sampled data (use $time to verify it). Realize the difficulty and check if use of any forked process can be useful to solve the above problem.

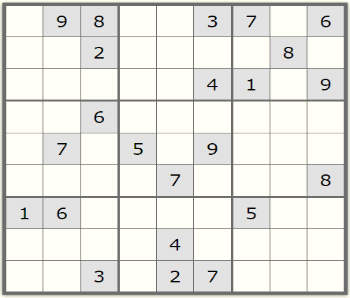
**SV Lab-7**

* Define a parameterized class called as packet. Type of data is a parameter input. Class contains following properties: length (no. of data), saddr (2-bits source address), daddr (2-bits destination address) and data (based on type and length). Class contains following methods: random (use to randomize data), set\_addr (use to set source and destination address), new constructer (accept length as an argument) and display (to display saddr, daddr, length and data in the given order). Add a check to make sure that source address and destination address are not same, if same display a message and randomize destination address (should be different than source address). Create packet objects of type int, 7-bits and 4-bits.
* Inheritence
* Local, and protected
* Static and this
* Abstraction, check for pure (search for example)
* Polymorphism (search for example)
* Mailbox
* Semaphore (Arbiter).

**SV Lab-8**

* Use parameterized class and concept of constraints to solve following 3 Sudoku and print their solution. The class parameter denotes the size of Sudoku (N x N). Here N can be multiple of 2 or 3.





**SV Lab-9**

* Design an Arbiter, which accepts 4 requests (req1, req2, req3, req4) and generates 4 grants (gnt1, gnt2, gnt3, gnt4) respectively. Arbitration is done based on fixed priority (req1>req2>req3>req4). Arbiter can generate grant anytime between 3 to 7 clock cycles (selection is random) once a request is received. At a given time only single grant can be high and can be high for maximum 2 clock cycles. New requests should not come until one of the grants has gone from high to low. Use assertions to perform all checks.
* Design a state machine to detect sequence “10110” left to right Non-Overlapping. Use one hot encoding to design this state machine. Write assertions to make sure that:
* State machine is not going into any invalid state i.e. all states should be one-hot encoded.
* All state transitions are valid with respect to current input.
* There is no invalid state transition irrespective of inputs.
* Output is high only if the entire sequence is detected.
* Output should not be high for more than one clock cycle.

**SV Lab-10**

* Refer to question 1 of Lab 9 and write coverage to cover following scenarios:
* Code Coverage.
* Toggling of all requests and grants.
* All possible combinations of request.
* Each grant was high for both one and two clock cycles.
* Each grant was generated 3 to 7 clock cycles after request.
* Generating request when grant has not gone from high to low.
* And other scenarios.
* Refer to question 2 of Lab 9 and write coverage to cover following scenarios:
* Code Coverage.
* All valid states have been covered at least 10 times.
* All valid state transitions have been covered at least 2 times.
* Illegal states.
* Illegal state transitions.
* Transition from each valid state to Idle (Reset) state when reset is applied.
* Both combinations of input i.e. 0 and 1 while performing state transitions.
* And other scenarios.