Q1. 定宽数组、动态数组、关联数组、队列各自的特点和使用方式。

A fixed-size array: Its size is determined at the time of definition, and it cannot be changed afterwards. It is often used to handle data sets of fixed size that will not change.

```
int array[10]; // Declare an array with 10 elements
```

A dynamic array: Its size can be dynamically changed at runtime, using the `new` operator to allocate space. The main usage scenario is when the number of elements is uncertain.

```
int dynamic_array[]; // Declare a dynamic array
dynamic_array = new[10]; // Allocate space for 10 elements
```

An associative array: It is an unordered data set, where data is stored and accessed through key-value pairs, and allows any data type to be used as an index. It is mainly used for lookup tables or dictionary-like data structures.

```
int associative_array[string]; // Declare an associative array with a string
index
associative_array["key"] = 1; // Store data with the key "key"
```

A queue: It is a data structure where elements can be inserted and removed at any position, and its size can be changed dynamically.

```
int queue[$]; // Declare a queue
queue.push_back(10); // Add 10 to the end of the queue
```

The fork...join, fork...join_any, and fork...join_none statements in SystemVerilog are used for parallel block execution. They define how and when control is transferred back from the spawned parallel blocks to the main thread.

Q2.fork...join/fork...join_any/fork...join_none 之间的异同

1. fork...join: The join keyword ensures that all blocks inside the fork...join have completed execution before control is passed back to the main thread. In other words, the main thread will wait for all forked tasks to finish. This is akin to a "barrier synchronization".

```
fork
  begin: task1
    #10;
    $display("Task 1 finished");
  end
  begin: task2
    #20;
    $display("Task 2 finished");
  end
join // Main thread waits for both tasks to finish
```

2. fork...join_any: The join_any keyword passes control back to the main thread as soon as any one of the parallel blocks completes execution. In this case, the main thread does not wait for all tasks to finish; it moves on as soon as the first task completes.

```
fork
  begin: task1
    #10;
    $display("Task 1 finished");
  end
  begin: task2
    #20;
    $display("Task 2 finished");
  end
join_any // Main thread continues as soon as any one task finishes
```

3. fork...join_none: The join_none keyword does not wait for any of the forked tasks to complete. As soon as the fork is encountered, control is passed back to the main thread. The forked tasks continue to execute independently.

```
fork
  begin: task1
    #10;
    $display("Task 1 finished");
  end
  begin: task2
    #20;
    $display("Task 2 finished");
  end
join_none // Main thread does not wait for any task to finish
```

Note: In fork...join_none, tasks continue in the background, and they can lead to race conditions if they are not properly managed. It's a best practice to use join or join_any to ensure proper synchronization and avoid unexpected behavior.

Q3.mailbox、event、semaphore 之间的异同

This question is about different mechanisms for inter-process communication in SystemVerilog.

1. **Mailbox**: Mailbox is a queue-based mechanism for inter-process communication that can be used to send and receive messages between processes. In SystemVerilog, a mailbox can be bounded or unbounded. A bounded mailbox has a fixed size, while an unbounded mailbox can receive an unlimited number of messages.

   For example:

   ```
   mailbox mb;
   initial begin
     mb = new(); // Create a new mailbox
   ```

```
  mb.put("Hello"); // Send a message to the mailbox
end

initial begin
  string msg;
  mb.get(msg); // Receive a message from the mailbox
  $display(msg);
end
```

2. **Event**: An event is a synchronization mechanism in SystemVerilog that allows a process to wait for one or more specific events to occur. A process can trigger an event by calling the `->` operator, and other processes can wait for the event using the `@` operator.

   For example:

```
event e;
initial begin
   ->e; // Trigger event e
end

initial begin
   @e; // Wait for event e to occur
   $display("Event occurred");
end
```

3. **Semaphore**: A semaphore is a synchronization mechanism that can be used to limit access to shared resources.

A semaphore in SystemVerilog can be owned by one or more processes. When a process owns a semaphore, other processes that want to get the semaphore must wait.

For example:

```
semaphore s;
initial begin
  s = new(1); // Create a new semaphore with initial value 1
  s.get(); // Get the semaphore
  // Access shared resources here
  s.put(); // Release the semaphore
end
```

Q4.@(event_handle)和 wait(event_handle.triggered)区别

Both `@(event_handle)` and `wait(event_handle.triggered)` are used to block execution until a certain event occurs, but they behave differently in SystemVerilog.

1. `@(event_handle)`: The `@` operator with an event handle in its parenthesis is a built-in SystemVerilog feature that allows one to suspend the current process until the specified event is triggered. The process resumes immediately after the event is triggered. It does not check the event's trigger condition.

```
event e1;
initial begin
  @(e1);
  $display("Event e1 triggered, resuming execution.");
end
initial begin
  #10;
  ->e1; // Trigger event e1 after 10 time units
end
```

2. `wait(event_handle.triggered)`: The `wait` statement suspends the current process until the condition in its parenthesis becomes true. In this case, the condition is `event_handle.triggered`, which will be true if the event has been triggered since the last time it was checked. Unlike `@`, the `wait` statement checks the event's trigger condition.

```
event e1;
initial begin
  wait(e1.triggered);
  $display("Event e1 triggered and checked, resuming execution.");
end
initial begin
  #10;
  ->e1; // Trigger event e1 after 10 time units
end
```

The main difference is the way they check for the event. `@(event_handle)` waits for the event trigger and then continues execution, while `wait(event_handle.triggered)` waits for the trigger and checks the condition before continuing. Therefore, if multiple triggers occur before the `wait` statement is executed, they will not be recognized by `@(event_handle)` but will be recognized by `wait(event_handle.triggered)`.

Q5.task 和 function 异同区别

omit

Q6.使用 clocking block 的好处

Using clocking blocks in SystemVerilog has several advantages, especially in terms of timing and synchronization. Here are some of them:

1. **Clearly defined timing**: A clocking block provides an unambiguous way to specify the timing of inputs and outputs with respect to a particular clock edge. It clarifies when input is sampled and when output is driven with respect to the clock.

```
clocking cb @(posedge clk);
  input a, b;
  output z;
endclocking
```
In the example above, inputs `a` and `b` are sampled and output `z` is driven at the positive edge of the `clk`.

2. **Synchronization**: A clocking block can encapsulate the synchronization of multiple signals with respect to the same clock, which can simplify the design and improve readability.
3. **Skew handling**: In simulation, clocking blocks ensure that there is no race condition between the clock edge and the data change because all activities within a clocking block are scheduled in a deterministic order.
4. **Convenience for testbench development**: In a testbench, clocking blocks can be used to create clock-driven sequences and checks in a straightforward manner. The `##` operator can be used to denote clock cycles in assertions, sequences, and properties.

```
assert property (@(cb) a |-> ##1 b);
```
In the above property, the `##1` denotes a delay of one clock cycle as per the clocking block `cb`.

5. **Improved integration with verification methodologies**: Clocking blocks are supported and extensively used in advanced verification methodologies such as UVM (Universal Verification Methodology). They allow consistent reference to timing throughout the verification environment.

Q7.同步 FIFO 和异步 FIFO 的作用和区别 (CDC)

Synchronous FIFO (First-In-First-Out) and Asynchronous FIFO are two types of FIFO memory buffers used in digital systems for data storage and transfer. Here are their purposes and the main differences between them:

Purpose:

1. **Synchronous FIFO**: It's used when the read and write operations occur at the same clock frequency, i.e., the data producer and the data consumer are operating at the same clock speed.
2. **Asynchronous FIFO**: It's used when the read and write operations happen at different clock frequencies. In other words, the data producer and the consumer operate at different clock speeds.

Differences:

1. **Clocking Scheme**: In synchronous FIFO, a single clock controls both reading and writing operations. In asynchronous FIFO, there are two separate clocks: one for reading and another for writing.
2. **Design Complexity**: Asynchronous FIFO design is more complex than synchronous FIFO because of the need to handle the data transfer between two different clock domains, which introduces additional considerations like metastability.
3. **Use Cases**: Synchronous FIFOs are suitable in applications where data rate does not vary, such as in some DSP applications. Asynchronous FIFOs are crucial in systems with varying data rates or different clock domains, such as in data communication or when interfacing between two asynchronous systems.

Here's an example of a simple synchronous FIFO design:

```
module fifo #(parameter DEPTH = 16, WIDTH = 8) (
```

```
  input wire clk,
  input wire reset,
  input wire wr_en,
  input wire [WIDTH-1:0] wr_data,
  output wire [WIDTH-1:0] rd_data,
  input wire rd_en
);

  reg [WIDTH-1:0] memory [0:DEPTH-1];
  reg [WIDTH-1:0] rd_data_reg;
  integer write_pointer = 0;
  integer read_pointer = 0;

  assign rd_data = rd_data_reg;

  always @(posedge clk or posedge reset) begin
    if (reset) begin
      write_pointer <= 0;
      read_pointer <= 0;
    end else begin
      if (wr_en) begin
        memory[write_pointer] <= wr_data;
        write_pointer <= write_pointer + 1;
      end
      if (rd_en) begin
        rd_data_reg <= memory[read_pointer];
        read_pointer <= read_pointer + 1;
      end
    end
  end
endmodule
```

This FIFO is synchronous because it only has one clock `clk` that controls both the reading and writing operations.

Q8.SystemVerilog 中 OOP 的三大特性

omit

Q9.详述对于 ref 类型的理解

omit

Q10.外部约束如何使用，有哪几种方式

In SystemVerilog, constraints are used to guide the randomization process. External constraints are constraints defined outside of the class and can be used to modify the behavior of the randomization. There are mainly two ways to apply external constraints in SystemVerilog: inline constraints and constraint blocks.

1. **Inline Constraints**: They can be applied directly when calling the `randomize()` function by using the `with` keyword. This is a way to apply immediate or one-time constraints on variables during the randomization process.

Example:

```
class MyClass;
  rand bit [7:0] var1, var2;
endclass

MyClass obj = new();

// Randomize with an inline constraint
obj.randomize() with { var1 < var2; };
```

In the example above, the variables `var1` and `var2` are randomized such that `var1` is always less than `var2`.

2. **Constraint Blocks**: You can also apply external constraints by creating a constraint block using the `constraint` keyword. This can be useful for applying the same set of constraints multiple times. These constraint blocks are defined outside of the class but are associated with a class instance.

Example:

```
class MyClass;
  rand bit [7:0] var1, var2;
endclass

MyClass obj = new();

// Define an external constraint block
constraint c1 { obj.var1 < obj.var2; }

// Randomize using the constraint block
obj.randomize() with { c1; };
```

In the example above, the constraint block `c1` ensures that `var1` is always less than `var2` during randomization.

Remember that if there are any conflicts between the internal (those defined inside the class) and external constraints, the solver will fail, and the randomization will return 0 (false).

Q11.代码覆盖率、功能覆盖率、SVA 覆盖率都是衡量什么的

In the context of verification, coverage is a metric used to measure the extent to which the design or test has been exercised by a test suite. Coverage is crucial in finding holes in your testing and ensuring that your testing is comprehensive. Here is what each type of coverage measures:

1. **Code Coverage**: Code coverage measures how much of the design code has been executed during simulation. It gives an idea of how much of the design's logic or structure has been tested. The different types include line coverage, toggle coverage, branch coverage, condition coverage, path coverage, FSM state and transition coverage, and so on.
2. **Functional Coverage**: Functional coverage measures how much of the design's functionality has been covered during testing. It's user-defined and represents whether the scenarios, corner cases, and protocol-specific behaviors have been checked. Functional coverage is usually specified in the form of covergroups, coverpoints, and crosses in SystemVerilog.

```
covergroup cg_example @(posedge clk);
  coverpoint dut.signal_a;
  coverpoint dut.signal_b;
  cross signal_a, signal_b; // Cross coverage between signal_a and signal_b
endgroup
```

3. **Assertion Coverage (SVA Coverage)**: Assertion coverage measures how many assertions have been proven true or false during simulation. Assertions are used to check for conditions that should (or should not) occur during the operation of the design. SVA coverage allows us to track whether all assertions have been adequately checked during simulation.

```
// Example of an assertion
assert property (@(posedge clk) reset_n |-> !req);
```

In the example above, the assertion checks that `req` is not asserted right after the de-assertion of `reset_n`.

All these types of coverage metrics are key to a comprehensive verification strategy, as they provide different perspectives on the design's verification status. Code coverage shows what has been executed, functional coverage shows what functionality has been tested, and assertion coverage shows what assertions have been checked. Together, they can provide a holistic picture of the verification status.

Q12.为什么选择验证工作

omit

因为设计岗面不上呗…

要不是为了钱，我理你干嘛.

Q13.立即断言和并发断言的特点

In SystemVerilog, assertions are used to validate the behavior of a design, and they can be classified into two categories: immediate assertions and concurrent assertions. Here are their characteristics:

1. **Immediate Assertions**: These are evaluated at the point of their occurrence in the procedural code. An immediate assertion must complete in zero simulation time and cannot span across multiple time points. It evaluates a Boolean expression, and if the expression is false, it will immediately raise an assertion failure.

Example:

```
initial begin
  assert (reset_n == 1'b0) else $error("Reset is not asserted at start!");
end
```

In this example, the immediate assertion checks if `reset_n` is low at the start of the simulation. If not, it will immediately report an error.

2. **Concurrent Assertions (SVA - SystemVerilog Assertions)**: These are evaluated continuously over time and can span across multiple time points. They are used to express more complex temporal behaviors and relationships between signals in the design. Concurrent assertions can be used in initial blocks, always blocks, and can be directly placed in modules.

Example:

```
// Concurrent assertion
always @(posedge clk) begin
  assert property (reset_n |-> ##[1:5] !req);
end
```

In this example, the concurrent assertion checks that after `reset_n` is deasserted, `req` should not be asserted for the next 1 to 5 clock cycles. This assertion will be continuously checked at every positive edge of `clk`.

In conclusion, immediate assertions are used for point-in-time checks, while concurrent assertions are used for temporal or sequential checks in the design. Both are powerful tools for verifying the correctness of a design.

Q14.SystemVerilog 中面向对象编程的优势

omit

Q15.如何保证验证的完备性

Ensuring completeness in verification involves multiple strategies and methodologies to ensure that the design has been thoroughly tested and validated under all relevant scenarios. Here are a few strategies that can be adopted:

1. **Coverage Driven Verification**: This strategy aims to achieve high code, functional, and assertion coverage. It includes writing tests to cover all the scenarios not covered initially and refining tests until high coverage is achieved. This ensures all parts of the design and all functionalities have been exercised and verified.
2. **Assertions**: Use immediate and concurrent assertions to verify the design's behavior in both point-in-time and over time scenarios. Assertions are a powerful tool for checking complex temporal relationships in a design.
3. **Directed and Randomized Testing**: Directed tests are used to check specific behaviors and scenarios, while randomized tests are used to explore unexpected corner cases and rare conditions.
4. **Formal Verification**: For certain designs and design components, formal verification can be used to prove whether the design meets the specification under all possible conditions.
5. **Use of Verification IP and Methodologies**: Verification IPs and methodologies like UVM (Universal Verification Methodology) can be used to create comprehensive verification environments.
6. **Regression Testing**: Regular regression testing is essential to ensure any changes or updates do not introduce new bugs or errors.
7. **Peer Review and Inspection**: Regular code reviews and inspections can help catch issues that automated tools may miss.
8. **Use of Checkers and Monitors**: Checkers and monitors can be used in the verification environment to check the correct behavior of the design during simulation.

Remember that it's not enough to just write tests and run them. The key is to analyze the results, improve the test based on these results, and continuously refine and expand the test suite until high coverage is achieved. Completeness in verification requires a combination of these strategies and a robust verification plan.

Q16.权重约束中":="和":/"的区别

In SystemVerilog, the distinction between ":=" and ":/" in weighted constraints lies in their constraint solving behavior.

1. `:=`: It is a hard constraint. This means it is a requirement that must be met. If a hard constraint cannot be satisfied, the constraint solver will fail and randomization will return 0 (false).

Example:

```
class MyClass;
  rand bit [7:0] var;
  constraint c { var := 8'hA5; } // var will always be 8'hA5
endclass
```

2. `:/`: It is a soft constraint. This means it is a preference rather than a requirement. If a soft constraint cannot be satisfied, the constraint solver will still try to satisfy the other constraints.

Example: $soft var == 2;

```
class MyClass;
  rand bit [7:0] var;
  constraint c { var :/ 2; } // var will try to be 2, but it's not a requirement
endclass
```

Q17.rand 和 randc 区别

The difference between `rand` and `randc` in SystemVerilog is in the random number generation pattern they produce.

1. `rand`: It generates a random number every time it's called. It can produce the same number on consecutive calls, as each call is statistically independent.

```
class MyClass;
  rand bit [7:0] var;
endclass
MyClass obj = new();
obj.var.rand(); // Generates a random number
```

2. `randc`: It stands for random-cyclic. It generates a random number every time it's called, but will not repeat any value until it has generated all possible values. Once all possible values have been generated, it starts a new cycle of the same set of numbers.

```
class MyClass;
  randc bit [7:0] var;
endclass
MyClass obj = new();
obj.var.randc(); // Generates a random number, but won't repeat until all values are generated
```

In the examples above, `var` will hold a random value after the `rand` or `randc` function is called.

Q18.break、continue 和 returen 的含义用法

omit

Q19.function 中 return 语句执行之后，function 里剩下的代码语句还会执行吗

Q20.触发器和锁存器的区别

omit

Q21.为什么要使用两级触发器进行同步

omit

Q22.setup 和 hold 违例分别是什么

omit

Q23.SVA 中 and、intersect、or、throughout、$past 如何使用

The operators `and`, `intersect`, `or`, `throughout`, and `$past` in SystemVerilog Assertions (SVA) are used to create complex temporal properties. Here's how they are used:

1. `and`: Checks that two sequences are true at the same time.

Example:

```
property p; @(posedge clk) (req && grant) |-> ##1 ack; endproperty
```
In this property, both `req` and `grant` must be true for `ack` to be true on the next clock cycle.

2. `intersect`: Checks that two properties are true in the same interval.

Example:

```
property p; @(posedge clk) (a[*2] intersect b[*3]); endproperty
```
This property checks that `a` is true for 2 consecutive clock cycles at the same time that `b` is true for 3 consecutive clock cycles.

3. `or`: Checks that either one of two sequences is true.

Example:

```
property p; @(posedge clk) (req || grant) |-> ##1 ack; endproperty
```
In this property, either `req` or `grant` must be true for `ack` to be true on the next clock cycle.

4. `throughout`: Specifies that a sequence must be true throughout the duration of another sequence.

Example:

```
property p; @(posedge clk) req |-> ##[1:3] grant throughout ack; endproperty
```
This property specifies that once `req` is true, `grant` must be true and `ack` must be true throughout the next 1 to 3 clock cycles.

5. `$past`: This system function allows access to historical values of variables, expressions, or sequences. It's used to specify behavior relative to past values.

Example:

```
property p; @(posedge clk) $past(req, 2) |-> grant; endproperty
```
This property specifies that if `req` was true 2 clock cycles ago, then `grant` must be true in the current clock cycle.

~~Q24.SVA 中"->"和"=>"区别~~

~~In SystemVerilog Assertions, -> and => represent implication operators:~~

1. ~~->: It's a non-overlapping implication. The antecedent (LHS) and consequent (RHS) do not share common cycles.~~

~~Example:~~

```
property p; @(posedge clk) req |-> ##1 ack; endproperty
```
~~In this property, if `req` is true, then `ack` must be true on the next clock cycle.~~

2. ~~=>: It's an overlapping implication. The antecedent (LHS) and consequent (RHS) may share common cycles.~~

~~Example:~~

```
property p; @(posedge clk) req |=> ack; endproperty
```
~~In this property, if `req` is true, then `ack` must also be true in the same clock cycle.~~

Q25.如何关闭约束

To disable a constraint in SystemVerilog, you can use the `disable` keyword, followed by the name of the constraint to be disabled when calling the `randomize` function.

Example:

```
class MyClass;
  rand bit [7:0] var;
  constraint c1 { var < 8'h80; }
endclass
MyClass obj = new();
obj.randomize() disable obj.c1; // Disables the constraint c1
```

obj.constraint_mode(0);

obj.(constraint) constraint_mode(0);

Q26.deep copy 和 shallow copy 区别

omit

Q27.队列常用的方法有哪些

omit

Q28.local 和 protected 区别

The keywords `local` and `protected` in SystemVerilog are used to control the visibility and accessibility of class members:

1. `local`: A local data member or method is accessible only within the class where it is declared. It cannot be accessed from outside the class or from any derived classes.

```
class MyClass;
  local bit [7:0] var;
endclass
```

In this example, `var` is only accessible inside `MyClass`.

2. `protected`: A protected data member or method is accessible within the class where it is declared and in any classes derived from it, but it cannot be accessed from outside these classes.

```
class MyClass;
  protected bit [7:0] var;
endclass
```

In this example, `var` is accessible inside `MyClass` and in any classes derived from `MyClass`, but not elsewhere.

Q29.常用的 debug 方法有哪些

Debugging is an integral part of the design and verification process. Common debug methods in SystemVerilog include:

1. **Print Statements**: Using `$display`, `$write`, `$monitor`, etc. to print the values of variables, states, or conditions.
2. **Waveform Viewing**: Dumping waveforms using `$dumpfile` and `$dumpvars`, and analyzing them in a waveform viewer.
3. **Assertions**: Assertions can help catch violations of expected behavior.
4. **Using a Debugger**: Many SystemVerilog simulators come with a debugger that can single-step through code, set breakpoints, etc.
5. **Code Review**: Peer code reviews can be very effective in catching issues.
6. **Logging**: Creating a log file with timestamped events and variable values can be very useful.

Q30.亚稳态的危害

omit

Q31.二进制码、格雷码、独热码的特点

omit

Q32.packed array 和 unpacked array 的区别

omit

Q33.阻塞赋值和非阻塞赋值的区别

omit

Q34.过程性语句和连续赋值语句的区别

omit

Q35.initial 和 always 的异同

omit

Q36.FSM 有哪几种？区别是什么？

omit

Q37.数字电路中为什么要使用触发器

omit

Q38.异步复位和同步复位各自特点和区别

omit

Q39.异步复位同步释放代码实现

Here's an example of an asynchronous reset and synchronous deassertion (release) design:

```
always_ff @(posedge clk or negedge reset_n)
  if (!reset_n)
    q <= '0; // Asynchronous reset
  else
    q <= d; // Synchronous deassertion
```

In this example, `reset_n` is an active-low asynchronous reset. When `reset_n` is 0, `q` is immediately reset to 0, regardless of the clock. When `reset_n` is 1, `q` follows `d` at every rising edge of the clock.

Q40.数字电路通常分为哪两种电路

Digital circuits are generally categorized into two types:

1. **Combinational Circuits**: In these circuits, the output depends only on the current inputs. Examples include basic gates (AND, OR, NOT), decoders, multiplexers, etc.
2. **Sequential Circuits**: In these circuits, the output depends on both the current inputs and the previous state of the system. Examples include flip-flops, counters, shift registers, etc.

Q41.illegal_bins 和 ignore_bins 命中分别会怎么样？命中是否会计入覆盖率统计

In SystemVerilog, `illegal_bins` and `ignore_bins` are used to classify certain values in coverage models:

1. `illegal_bins`: Values that fall into `illegal_bins` will cause an error to be thrown during simulation if they are hit.

```
covergroup cg_example @(posedge clk);
```

```
  coverpoint dut.signal_a {
    bins legal_values = {[0:10]};
    illegal_bins illegal_values = {[11:$]};
  }
endgroup
```

In this example, if `signal_a` takes a value between 11 and `$` (maximum value), it will be considered as an illegal value and an error will be thrown.

2. `ignore_bins`: Values that fall into `ignore_bins` are ignored during coverage calculation. They do not affect coverage statistics.

```
covergroup cg_example @(posedge clk);


  coverpoint dut.signal_b {
    bins legal_values = {[0:10]};
    ignore_bins ignored_values = {[11:$]};
  }
endgroup
```

In this example, if `signal_b` takes a value between 11 and `$` (maximum value), it will be ignored in the coverage calculation.

In conclusion, hits in `illegal_bins` generate an error and do not count towards coverage, while hits in `ignore_bins` are simply ignored and also do not count towards coverage.

Q42.负数采用二进制如何表示

omit

Q43.4 值逻辑变量赋值给二值逻辑变量时，x 和 z 对应什么值

In SystemVerilog, when assigning a 4-state logic value to a 2-state logic value, the `x` (unknown) and `z` (high impedance) states map to a `0` state.

Example:

```
logic [3:0] four_state = 4'b1zx1;
bit [3:0] two_state = four_state; // will be 4'b1001
```

In this example, `x` and `z` in `four_state` get mapped to `0` in `two_state`.

Q44.类中 this 是什么

Q45.子类中 super 是什么?

omit

Q46.在 IC 验证中，我们一般对哪些内容进行随机化

In Integrated Circuit (IC) verification, the following components are commonly randomized:

1. **Inputs to the Design Under Test (DUT)**: Input stimuli are randomized to exercise the DUT under various conditions and corner cases.
2. **Timing parameters**: To verify the DUT under different speed conditions and clock frequencies.
3. **Operational modes**: If the DUT supports various operational modes, these modes are randomized to test the DUT's ability to switch between modes and operate correctly in each mode.
4. **Configuration parameters**: These are randomized to test the DUT's behavior under different configurations.

Q47.通过函数返回数组有哪些方法

In SystemVerilog, you can return arrays from a function in a few ways:

1. **Returning dynamic arrays and associative arrays**: You can return dynamic arrays and associative arrays directly from a function.

```
function int[] return_array();
  int array[] = {1, 2, 3};
  return array;
endfunction
```

2. **Returning fixed-size arrays**: For fixed-size arrays, you need to wrap the array inside a `typedef` or a `struct` or `class`, because SystemVerilog does not allow returning packed arrays from a function.

```
typedef bit [7:0] array_t[3];
function array_t return_array();
  array_t array = '{8'h1, 8'h2, 8'h3};
  return array;
endfunction
```

Q48.什么是 clocking block 的 skew

omit

Q49.并发断言的主要组成有哪些

Concurrent assertions in SystemVerilog are composed of several key components:

1. **Sequence**: A sequence defines a series of events that occur in specific temporal order.
2. **Property**: A property is a condition that a sequence must meet.
3. **Assertion**: An assertion specifies a property that must hold true. If the property is violated, an error or warning is triggered.

Example:

```
sequence s; @(posedge clk) req ##1 ack; endsequence
property p; @(posedge clk) s |-> ##1 grant; endproperty
assert property(p); // Assertion
```

Q50.如何检查随机化是否成功

In SystemVerilog, you can check if a randomization was successful by checking the return value of the `randomize` method. It returns a `0` if randomization fails, and `1` if it succeeds.

Example:

```
class MyClass;
  rand bit [7:0] var;
endclass
MyClass obj = new();
if (!obj.randomize()) $display("Randomization failed!"); // Check if
randomization succeeded
```

In this example, a message is displayed if randomization of `obj` fails.

Q51.什么时候 randomize()失败

The randomize() function in SystemVerilog can fail in the following situations:

- When the randomization constraints are conflicting or cannot be met.
- When the object being randomized is declared as "const".

Example:

```
class MyClass;
  rand bit [7:0] data;

  constraint c_data { data < 10; data > 20; } // conflicting constraints
endclass

MyClass my_obj = new();
if (!my_obj.randomize()) $display("Randomization failed");  // This will display
"Randomization failed"
```

Q52.黑盒验证、灰盒验证、白盒验证

omit

Q53.竞争与冒险是什么

- Race condition: In concurrent computing, a race condition occurs when two or more threads can access shared data and they try to change it at the same time, leading to non-deterministic outcomes.
- Hazards: In digital logic, hazards refer to a situation where changes in input variables do not change the output correctly, mainly due to the propagation delays.

Q54.虚接口有什么好处

In SystemVerilog, virtual interfaces provide a level of abstraction that allows the same testbench to be reused with different DUTs (Design Under Test). It also facilitates the use of the same testbench code for different interface instances.

Q55.接口的使用有什么优势

omit

Q56.$cast 在句柄转换时如何使用

$cast is used in SystemVerilog for dynamic casting. It attempts to cast the object handle to the specified type and returns true if successful, false otherwise.

Example:

```
class Base;
endclass

class Extended extends Base;
endclass

Base base_handle;
Extended extended_handle = new;

if (!$cast(base_handle, extended_handle)) $display("Cast failed");
```

Q57.为什么要进行后仿真

- Post-simulation is performed to verify that the design behaves as expected after synthesis and place & route, with the applied back-annotated delays from the actual layout.
- It's done by taking the design netlist (post-synthesis or post-place & route), applying back-annotated delays, and then using the same testbench to simulate the design.
- It's essential because the synthesis and place & route processes can introduce changes and delays that weren't there in the RTL (Register Transfer Level) design. Post-simulation ensures that the final implementation of the design is correct.

Q58.如何进行后仿真

omit

Q59.什么是后仿真

omit

Q60.当 task 的通过 ref 传递数据时，如果 task 内部对数据进行了修改，task 外部是否立即可以看到数据被修改了还是要等到 task 执行完才能看到

When a task in SystemVerilog receives data via a `ref` argument, it receives a reference to the actual variable instead of a copy of the value. Therefore, if the task modifies the data, the changes are seen immediately outside the task, even before the task has finished execution.

Here's an example to illustrate this:

```
task my_task(ref int data);
  data = 10;
  #5; // delay
  data = 20;
endtask

initial begin
  int data = 0;
  fork
    my_task(data);
    begin
      #2; // delay
      $display(data); // displays 10, even though my_task has not finished execution
    end
  join
end
```

In this example, `my_task` receives `data` as a `ref` argument and modifies it. The `initial` block that calls `my_task` sees the modification to `data` before `my_task` has finished execution.

Q61.使用 packed struct 定义下面数据包：

31:24 23:16 15:4 3:0 p1 p2 p3 p4

In SystemVerilog, packed structs can be used to aggregate multiple different data types into a single data type. Here's how you can define the given data packet using a packed struct:

```
typedef struct packed {
  bit [31:24] p1;
  bit [23:16] p2;
  bit [15:4]  p3;
  bit [3:0]   p4;
} my_packet_t;

// Then you can declare a variable of this type:
```

```
my_packet_t packet;
```

Q62.随机化的优势是什么？是不是意味着不再需要定向 case 了

omit

Q63.randomize with{....}中的约束与 class 中的约束是什么关系

omit

Q64.如何基于随机化的验证环境写定向测试

omit

Q65.为什么数字电路系统中要使用二进制

omit

Q66.数字电路中可能存在的风险问题有哪些?

omit

Q67.什么是虚方法

omit

Q68.低功耗方法你了解哪些

omit

Q69.使用 FSM 设置序列检测器（序列：110110）

omit

Q70.描述你对数字集成电路设计流程的认识

omit

Q71.虚接口是什么

A virtual interface in SystemVerilog is a handle that points to an actual interface instance. It provides a way to reference and use actual interface instances in testbench code, adding a layer of abstraction that allows for greater reusability and configurability.

Example:

```
interface my_if(input logic clk);
  logic data;
  modport source(output data);
  modport sink(input data);
endinterface

class my_class;
  virtual my_if.vif; // Declare a virtual interface handle in a class
endclass
```

Q72.预定义的随机方法有哪些

In SystemVerilog, the predefined random methods are `randomize()` and `std::randomize()`. They are used to randomize the values of variables and can be used with constraints for more controlled randomization.

The `randomize()` method cannot be overloaded directly, but constraints can be used to modify its behavior.

The execution order of randomization is from parent class to child class, which means constraints of parent class are solved first. If `randomize()` fails, it returns 0, else it returns 1.

```
class MyClass;
  rand bit [7:0] data;
endclass
```

```
MyClass my_obj = new();
if (!my_obj.randomize()) $display("Randomization failed");
```

Q73.预定义的随机方法是否可以重载

omit

Q74.预定义的随机方法执行顺序和执行情况

omit

Q75.package 用途是什么

namespace —— { }

Packages in SystemVerilog are used to encapsulate definitions of data types, functions, tasks, and other items. They are used to avoid naming collisions and make code more modular, allowing code reuse.

Example:

```
package my_package;
  typedef enum {RED, GREEN, BLUE} color_t;
endpackage : my_package

module my_module;
  import my_package::*; // importing the package

  my_package::color_t my_color; // using a typedef from the package
endmodule
```

Q76.package 如何使用

omit

Q77.如何在子类中调用父类中的方法

To call a method of a parent class in a subclass, use the `super` keyword:

```
class Base;
  function void my_function();
    $display("Base class function");
```

```
  endfunction
endclass

class Child extends Base;
  function void my_function();
    super.my_function(); // calling the parent class function
    $display("Child class function");
  endfunction
endclass
```

Q78.bit[7:0]和 byte 有什么区别

omit

Q79.类中的方法和类外的方法有什么区别

Methods inside a class are typically related to the data of the class and can access this data directly. They need an object of the class to be invoked.

Methods outside a class, i.e., tasks or functions, are not directly associated with a class and don't need an object to be invoked. However, they cannot directly access the data of a class.

To define a method outside a class, simply define a task or function outside of the class scope.

```
task my_task;
  // do something
endtask

class MyClass;
  // class definition
endclass
```

Q80.如何将类中的方法定义在类外

omit

Q81.modport 的用途是什么

omit

Q82.struct 和 union 的异同

omit

Q83.$rose 和 posedge 区别

omit

Q84.如何在 fork...join 结构中 kill 进程

omit

Q85.什么是覆盖率驱动的验证

Coverage-driven verification (CDV) is a technique used in the field of semiconductor device verification. The goal is to ensure the design has been fully tested and there are no functional errors. The two key elements of CDV are coverage collection and constraint random verification.

Q86.如何检查句柄是否指向有效对象

In SystemVerilog, you can check if a handle points to a valid object by comparing it with null. If the handle is null, it does not point to a valid object.

```
class MyClass;
endclass

MyClass my_obj;

if (my_obj == null) $display("my_obj does not point to a valid object");
```

Q87.semaphore 用处是什么

omit

Q88.为什么要使用断言

omit

Q89.如何在 clocking block 中声明异步信号

Asynchronous signals are not associated with any clock signal. In a clocking block, all signals are implicitly associated with the clock. Hence, you cannot directly declare an asynchronous signal inside a clocking block. However, you can access the asynchronous signals outside the clocking block.

In SystemVerilog, a clocking block is primarily associated with a specific clock signal that dictates the clocking edge and skews for all signals within the clocking block. This means that you cannot directly declare asynchronous signals inside a clocking block.

However, you can still declare asynchronous signals outside the clocking block and use them inside the clocking block. This can be done by directly referring to the external signals.

For example:

```
interface intf(input logic clk, input logic reset, inout logic [7:0] data);
  clocking cb @(posedge clk); // Clocking block
  default input #1ns output #1ns; // Set skews
  inout data;
endclocking
endinterface : intf

module tb;
  logic clk, reset, [7:0] data;
  intf i_intf(.clk(clk), .reset(reset), .data(data)); // Instantiate the
interface

  initial begin
    // Drive the asynchronous reset signal
    reset = 1;
    #5ns reset = 0;
  end

  initial begin
    // Use the asynchronous reset signal in a procedural block
    @(negedge reset) data = 8'hAA; // Do something when reset is asserted
  end
endmodule : tb
```

In this example, `reset` is an asynchronous signal that is declared outside the clocking block and used inside the clocking block. Inside the clocking block, you can modify the `data` signal based on changes to the `reset` signal.

Q90.代码覆盖率和功能覆盖率的关系

omit

Q91.什么是验证计划，应该包含哪些部分

omit

Q92.类中的静态方法使用注意事项有哪些

The following precautions should be taken when using static methods in SystemVerilog classes:

- Static methods can be invoked using the class name, without the need for any object of that class.
- Static methods cannot access non-static class properties or methods directly, as they are not associated with any specific object instance.
- You should make sure the static method does not try to modify any static variable in a way that could cause problems when multiple instances of the class exist.

Example:

```
class MyClass;
  static function void static_func();
    // do something
  endfunction
endclass

// calling the static method
MyClass::static_func();
```

Q93.initial 和 final 的区别

In SystemVerilog, `initial` and `final` are procedural blocks that are used to specify actions at the start and end of simulation respectively.

- `initial` blocks begin execution at time 0 in simulation and only execute once.
- `final` blocks execute after all other activity in the simulation has finished, just before termination.

Q94.建模存储器，使用什么类型的数组

omit

Q95.如何避免测试平台和 dut 之间的竞争冒险

To avoid race conditions and hazards, you can:

- Use handshake signals or protocols to ensure that both the testbench and DUT are ready before they start interaction.
- Use non-blocking assignments in always blocks to avoid race conditions.
- Use clocking blocks in testbenches to control the timing of signals.

Q96.logic、bit、wire 区别

omit

Q97.什么是抽象类

omit

Q98.always@*与 always_comb 区别

In SystemVerilog, `always@*` and `always_comb` are similar in functionality, both used to describe combinational logic.

- `always@*` block infers sensitivity list automatically. It includes all the variables that are on the RHS of assignments in the block.
- `always_comb` is similar to `always@*` but it includes some additional compile-time checks to ensure it behaves as intended, such as prohibiting usage of certain system tasks or disabling of the block.

Q99.简述验证结构

A typical SystemVerilog verification environment consists of multiple components:

- Testbench: It contains the DUT and test scenarios.
- Driver: It drives the inputs to the DUT.
- Monitor: It observes the outputs from the DUT.
- Scoreboard: It checks that the output is as expected.
- Sequencer/Sequence: They are used to generate stimulus to the DUT.
- Coverage collector: It collects coverage information.

Q100.parameter、define 和 typedef 之间区别

In SystemVerilog:

- `parameter`: It's a constant within modules

, interfaces, or packages. It cannot be changed once it's defined.

- `define`: It's a preprocessor macro. It's text replacement performed before actual compilation.
- `typedef`: It's used to define a new data type name, making code more readable and maintainable.

Example:

```
`define SIZE 8 // Preprocessor macro
typedef bit [7:0] byte_t; // typedef

module my_module;
  parameter WIDTH = `SIZE; // parameter

  byte_t data; // using the typedef
endmodule
```

Q101.new()和 new[]的区别

omit

Q102.solve...before 如何使用

The `solve...before` construct in SystemVerilog is used to direct the randomization engine to solve for the values of certain variables before others during randomization. It's especially useful in cases where there are dependencies between the variables.

Example:

```
class MyClass;
  rand bit [7:0] a, b;
  constraint c { solve a before b; b > a; }
endclass
```

In this example, `a` is solved before `b` due to the `solve a before b` directive.

Q103.mailbox 和队列的异同

Both mailbox and queue in SystemVerilog are used to store data. The main differences are:

- A mailbox provides a mechanism for interprocess communication and synchronization, where one process can wait for another process to send data to the mailbox. A queue, on the other hand, is simply a data structure with no such synchronization mechanism.
- A mailbox can be bounded or unbounded, whereas a queue is always unbounded.

Q104.什么是静态变量

omit

Q105.什么是生命周期

omit

Q106.交叉覆盖率的优点

Cross coverage in SystemVerilog is a method for measuring the coverage of a combination of variables, not just single variables. The advantages of cross coverage are:

allow to check coverpoints simutaneously

- It allows us to verify that different combinations of inputs have been tested.
- It provides more detailed coverage data that can help to uncover corner cases.

Q107.pass_by_value 和 pass_by_ref 区别

omit

Q108.$display 和$write 区别

Both `$display` and `$write` are system tasks in SystemVerilog used for displaying information.

- `$display`: It outputs the specified values and automatically adds a newline at the end.
- `$write`: It outputs the specified values but does not add a newline.

Q109.同一个作用范围内使用枚举类型需要注意什么

When using enumeration types in the same scope in SystemVerilog, you should ensure that there are no overlapping enumeration identifiers as this can lead to ambiguities and conflicts.

Example:

```
typedef enum {RED, GREEN, BLUE} color_e;
typedef enum {ORANGE, BLUE, YELLOW} fruit_e; // This will cause a conflict with the 'BLUE' in color_e
```

Q110.敏感信号列表信号缺失会如何

If a signal is missing from a sensitivity list in SystemVerilog, the always block may not execute when expected. It can lead to incorrect simulation results as changes to the missing signal will not trigger the always block.

Q111.covergroup 在类中使用和类外分别如何使用

A `covergroup` in SystemVerilog can be defined either inside a class or outside of it. The location does not affect its functionality. However, when defining a `covergroup` inside a class, the `covergroup` is associated with an instance of the class, allowing you to collect coverage data on a per-instance basis. When defining a `covergroup` outside of a class, the `covergroup` is associated with the module or interface in which it is defined.

Example:

```
class MyClass;
```

```
  bit [7:0] data;

  covergroup cg;
    coverpoint data;
  endgroup

  function new();
    cg = new();
  endfunction
endclass

module my_module;
  MyClass mc = new();

  covergroup cg;
    coverpoint mc.data;
  endgroup

  initial cg = new();
endmodule
```

In this example, `MyClass` has a `covergroup` that collects coverage on `data`, and `my_module` also has a `cover

group`that collects coverage on`mc.data`.

Q112.简述回调机制

omit

Q113.三段式状态机是哪三段（状态转移、组合逻辑描述状态转移规律、电路输出）

The three stages of a state machine usually refer to:

- Current state: where the machine stores the status information.
- State transition: which provides the rules for moving from one state to another.
- Output function: it provides the rules for determining the output of the machine based on the current state.

Q114.什么是虚接口，为什么要使用虚接口

Q115.Verilog 中 for 能不能综合

Yes, the 'for' loop can be synthesized in Verilog, but the loop iteration count must be a compile-time constant. The synthesizer unrolls the loop to generate the hardware.

Q116.举例常见的单 bit 同步机制

- Single bit synchronization: Flip-flops can be used for single-bit synchronization. An example would be a shift register, which can pass a bit from one flip-flop to another.

```
always @(posedge clk or negedge reset) begin
  if (!reset)
    q <= 1'b0;
  else
    q <= d;
end
```

- Multi-bit synchronization: For multi-bit synchronization, multiple flip-flops can be used, one for each bit. A synchronization mechanism can include a series of flip-flops for each bit to be synchronized.

```
genvar i;
generate
for (i=0; i<WIDTH; i=i+1) begin : multi_bit_sync
  always @(posedge clk or negedge reset) begin
    if (!reset)
      q[i] <= 1'b0;
    else
      q[i] <= d[i];
  end
end
endgenerate
```

Q117.举例常见的多 bit 同步机制

omit

Q118.SystemVerilog 中##n 表示什么

In SystemVerilog, the `##n` is used in sequence expressions to denote a delay of 'n' clock cycles. For example, `a ##3 b` would mean that event 'b' happens 3 clock cycles after event 'a'.

Q119.UVM 指的是什么？具有哪些特点，为什么要使用 UVM?

omit

Q120.简介工厂机制（factory）

The factory mechanism in UVM is used to create and configure UVM objects and components. The factory provides a central location for object creation, which enables more flexible testbench development, including the ability to override the object type, which is essential for building configurable and reusable testbenches.

Q121.简介事务级建模

Transaction-Level Modeling (TLM) is a high-level approach to modeling digital systems where the focus is on the flow of data transactions rather than the implementation details of the digital system. This makes it possible to abstract away many details of the system and can significantly speed up simulation.

Q122.uvm_component 和 uvm_object 的区别

The main difference is that `uvm_component` is derived from `uvm_object` and includes additional features necessary for hierarchical structure, like parent-child relationships, phases, and TLM interfaces.
Both `uvm_component` and `uvm_object` are base classes used in UVM from which other classes are derived.

Q123.UVM 中 run_phase 和 main_phase 的区别

The `run_phase` is a top-level phase that is used to execute the test. It consists of several sub-phases, one of which is the `main_phase`. The `main_phase` is typically where the bulk of the test functionality is coded. So the `run_phase` contains the `main_phase` as well as other phases like the pre_main_phase and post_main_phase.

Q124.为什么要使用 phase 机制

The phase mechanism in UVM provides a structured way to organize and control the execution of a test. The phases ensure that certain operations happen in a specific order, like build, connect, end of test, etc., making the tests more predictable and manageable.

Q125.m_sequencer 和 p_sequencer 区别

`m_sequencer` and `p_sequencer` are both handles to the sequencer driving a UVM agent or sequence item. `p_sequencer` is typically used in sequences to access the user-defined sequencer, and it must be cast to the appropriate type. `m_sequencer` is an integral part of UVM, automatically set to point to the sequencer running the sequence, and does not require casting.

Q126.top-down phase、bottom-up phase 有哪些

Top-down and bottom-up phases refer to the order in which phases are executed in the UVM testbench hierarchy.

Top-down phases are: build_phase, ~~connect_phase~~, end_of_elaboration_phase, start_of_simulation_phase, and run_phase.

Bottom-up phases are: extract_phase, check_phase, report_phase, final_phase.

Q127.为什么 build_phase 是 top-down phase，connect_phase 是 bottom-up phase

The build_phase is a top-down phase because it starts from the highest level of the hierarchy (i.e., the test) and works its way down, allowing the lower-level components to be aware of their parents and siblings during construction. This is useful for setting configuration values in lower-level components.

The connect_phase is a bottom-up phase because connections are often made from a lower-level component (like a driver or monitor) to a higher-level component (like an agent or a scoreboard). Therefore, it is easier to make these connections after all lower-level components have been constructed and any necessary configuration has been applied.

Q128.$size 用于 packed array 和 unpacked array 分别得到的什么

$size in SystemVerilog returns the number of elements in an unpacked array and the number of bits in a packed array.

Q129.class 和 struct 的异同

Both class and struct are used to group related variables and functions together. The differences are:

- Class: A class is a dynamic data type, meaning objects of the class are created dynamically at runtime. Classes support inheritance and polymorphism, allowing for more complex and flexible data structures.
- Struct: A struct is a static data type, meaning its size is fixed at compile time. It does not support inheritance or polymorphism.

Q130.class 和 module 的异同

Both classes and modules are used to encapsulate and manage related data and behavior. The differences are:

- Class: Classes are typically used to model data structures or transactions in a testbench. They are not synthesizable and can support dynamic object-oriented features like inheritance and polymorphism.
- Module: Modules are used to model design entities and can be synthesized into hardware. They are static and do not support dynamic features.

Q131.对象创建的初始化顺序

omit

Q132.子类和父类中是否可以定义相同名称的成员变量和方法（非虚方法）

但不可定义重命变量.

Yes, a subclass can define a member or method with the same name as in the parent class. However, the subclass's version will shadow the parent's version, which means that when accessed from the subclass, it will refer to the version defined in the subclass.

Q133.为什么需要随机

omit

Q134.线程间通信控制共享资源的原因是什么

Inter-thread communication is used to control shared resources in order to prevent race conditions and ensure that resources are used in a coordinated manner. This helps to maintain data integrity and avoid unexpected behavior.

Q135.uvm_transaction 和 uvm_seq_item 的关系

uvm_seq_item is a subclass of uvm_transaction. uvm_transaction is the base class for all transaction-level models. uvm_seq_item adds additional functionality that's required when a transaction is used in a sequence, like the ability to keep track of the sequence and sequencer that generated it.

Q136.p_sequencer 是什么?

p_sequencer is a handle to the user-defined sequencer that's driving an agent or sequence item. It is typically cast to the appropriate type in the sequence.

Q137.m_sequencer 是什么?

m_sequencer is a handle that's automatically set to point to the sequencer that's currently running the sequence. It is an integral part of UVM and does not need to be cast to a particular type.

Q138.new()和 create 有什么区别

new() is a constructor method that's used to create an instance of a class. create is a method in UVM factory that's used to create an object or component. The main advantage of using create over new is that create allows for factory overrides, which is useful for creating flexible and reusable testbenches.

Q139.如何启动 sequence

To start a sequence, you can call the `start` method on the sequence, passing in a handle to the sequencer. For example:

```
my_sequence.start(my_sequencer);
```

Q140.copy 和 clone 的区别

Both copy and clone are used to create a duplicate of an object, but they behave differently:

- The `clone()` method creates a new object that is a mirror image of the object it was called on, including any dynamic data. The new object is the same type as the original object.
- The `copy()` method copies the state of the object it was called on to the caller object. It doesn't create a new object, and the caller object can be a different type from the original object.

Q141.Agent 中的 Active mode 和 Passive mode 区别

Active and Passive modes refer to the modes of operation of an agent in UVM:

- Active mode: In this mode, the agent contains a driver to drive stimulus onto the DUT and a monitor to monitor the DUT's response. This mode is typically used for stimulus generation.
- Passive mode: In this mode, the agent contains only a monitor to monitor the DUT's response. This mode is typically used for observation and doesn't generate any activity on the interface.

Q142.在 UVM 的工厂机制中，为什么要使用注册机制

The registration mechanism is used in UVM's factory to allow objects to be created by their type name as a string. This enables more flexible and configurable testbenches since objects can be created and configured dynamically at runtime based on the type name.

Q143.简述 UVM 的工厂机制

The factory in UVM is a mechanism that is used for creating and configuring UVM objects and components. The factory provides a central location for object creation, which allows for object type overrides and more flexible and reusable testbench development.

Q144.UVM 中的 RAL 什么，可以用来干什么?

RAL stands for Register Abstraction Layer in UVM. It is used to create an abstract model of the registers in the DUT. This abstract model can be used for generating register accesses in tests, checking the results of register accesses against the model, and predicting the behavior of the DUT based on register accesses.

Q145.简述系统级、子系统级和模块级验证

These levels of verification refer to the granularity at which the verification is performed:

- System-level verification: This involves verifying the entire system as a whole, including all its subsystems and components. This is typically where integration and use-case testing occur.
- Subsystem-level verification: This involves verifying a subsystem of the system, which could be a group of modules that work together to perform a particular function.
- Module-level verification: This involves verifying an individual module, focusing on the functionality of that module.

Q146.IP 和 VIP 分别指的是什么

IP stands for Intellectual Property and in the context of verification, it refers to a reusable unit of logic or design (like a module or subsystem) that can be used across multiple projects. VIP, or Verification IP, is a reusable unit of verification environment that is used to verify the functionality of IP.

Q147.set_config_*和 uvm_config_db 区别

Both set_config_* methods and uvm_config_db are used to pass configuration information from one component to another. The difference is:

- set_config_* methods: These are methods in uvm_component used to set a configuration value for a particular field. The value can be retrieved later using get_config_* methods.
- uvm_config_db: This is a database that stores configuration information in a central location, allowing it to be accessed by any component in the testbench.

Q148.$stop、$finish 和 final 如何使用

$stop, $finish and final are system tasks used to control simulation flow:

- $stop: This system task stops the simulation and leaves the simulator in an interactive mode, allowing the user to inspect the state of the design.
- $finish: This system task stops the simulation and exits the simulator.
- final: This is a special system task that is automatically executed at the end of the simulation, after $finish. It's often used for tasks that need to be done after the simulation has completed, such as reporting coverage.

Q149.简述 virtual sequence 和 virtual sequencer 作用

Virtual sequences and virtual sequencers are used in UVM to control and coordinate the operation of multiple sequences and sequencers:

- Virtual sequence: A virtual sequence is a sequence that can start other sequences on multiple sequencers, allowing for coordinated stimulus generation across multiple interfaces.
- Virtual sequencer: A virtual sequencer is a sequencer that doesn't drive any interface itself, but serves as a placeholder for a virtual sequence to run on.

1. how to use tri-state buffer and not gate to implement all gate?
2. give you an inifinite sequence, you every 1 bit every cycle, write the state matchine if the current number can be divided by 5? What if MSB coming first? what if LSB coming first? And explain the code.
3. how to write a fix-priority arbiter, how to write a round robind arbiter (use kill chain). How do you verify it? And explain the code.
4. how to write a CAM? And explain the code.
5. how to design HW linked list? And explain the code.

---

1. Implementing all gates using tri-state buffers and NOT gate:

This is an example of how you can create a NAND gate using a tri-state buffer and a NOT gate. NAND gate is universal, and all gates can be created using a combination of NAND gates.

```
module NAND_GATE(input a, b, output wire y);
  wire not_a, not_b;
  assign not_a = ~a;
  assign not_b = ~b;

  tri1 tri_state_buf1(.y(y), .a(1'b1), .e(not_a));
  tri1 tri_state_buf2(.y(y), .a(1'b1), .e(not_b));
endmodule
```

2. State machine to check if a number can be divided by 5:

This can be done using a simple finite state machine (FSM) that moves through 5 states, with each state representing the remainder when divided by 5. Here's an example when the MSB is coming first:

```
module div_by_5(input wire clk, reset, bit_in, output reg [2:0] state);
  always @(posedge clk or posedge reset) begin
    if (reset) state <= 3'd0;
    else state <= (state << 1 | bit_in) % 5;
  end
  assign div_by_5 = (state == 0);
endmodule
```

If the LSB is coming first, you can shift the state to the right instead of the left.

3. Fixed priority arbiter:

A fixed-priority arbiter gives the highest priority to the highest numbered request. A round-robin arbiter, on the other hand, selects the next highest request from the last one served. Here's an example of a fixed-priority arbiter:

```
module arbiter #(parameter N = 4)(input wire clk, reset, wire [N-1:0] req, output
reg [N-1:0] grant, output reg [N-1:0] ack);
  always @(posedge clk or posedge reset) begin
    if (reset) grant <= 4'b0;
    else begin
      grant <= req & ~ack;
      ack <= (ack << 1) | (req & ~ack);
```

tri buffer

```
module tri (input x, input en, output y);
    assign y = en ? x : 1'bz;
endmodule
```

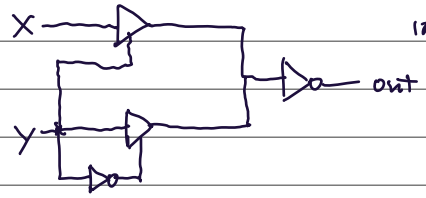
x	en	y
0	0	z
0	1	0
1	0	z
1	1	1

1. NAND

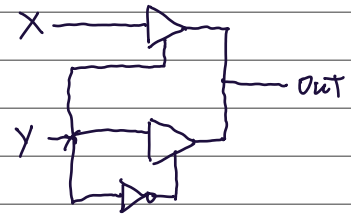
x	y	out
0	0	1
0	1	1
1	0	1
1	1	0

希望输出 0, 0, 0, 1
再用非门转成 1, 1, 1, 0

当前为 z, 0, z, 1
希望为 0, 0, 0, 1
以两端口 y 为 1 时，控制正常 x 作为输出
当 y 为 0 时，输出为 z，希望输出为 0，
则需要让 y 同之作为输出

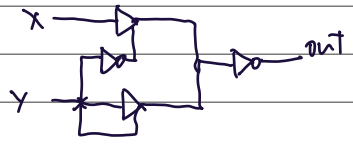


则 AND Gate：

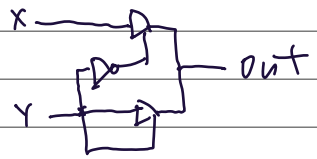


NOR:

x	y	exp	out	实际输出
0 √	0	1	z	0
0	1 √	0	0	1
1 √	0	0	z	1
1	1 √	0	1	1

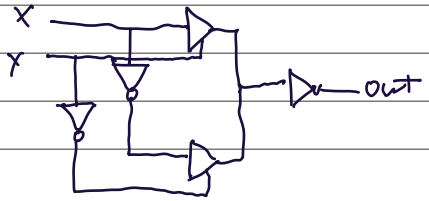


则 OR 为



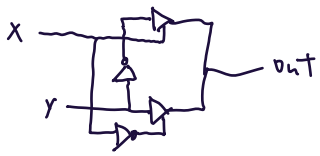
XOR

x	y	exp	out	
0 △	0	0	z	1
0 √	1	1	0	0
1 △	0	1	z	0
1 √	1	0	1	1

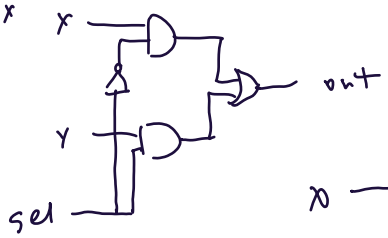


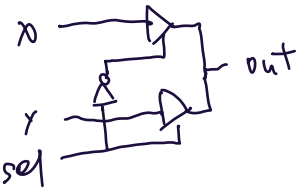
XOR

x	y	out
0	0	0
0	1	1
1	0	1
1	1	0



Mux



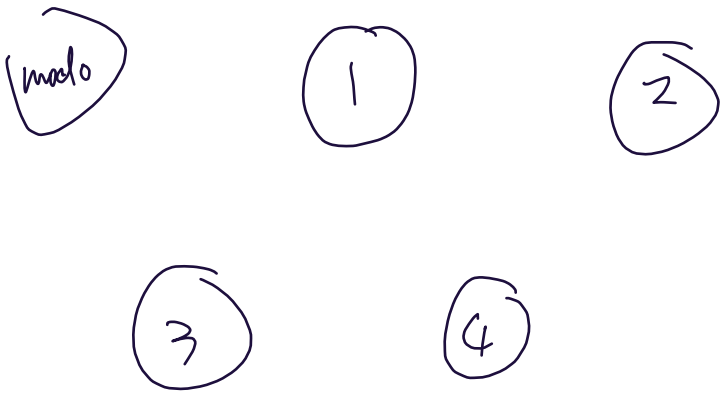


x	y	sel	out
0	0	0	x
		0	x
		1	y
		1	y

	1	2	4	8	16	32	64	128
mod 5	1	2	4	3	1	2	4	3

如果 MSB 先来

当前余数 ×2 + 新余数



0

1

2

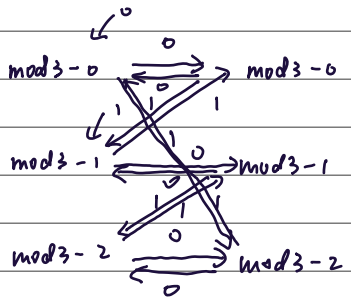
3

4

出来看是否能被3整除.

mod3	2	1	2	1	
	8	4	2	1	
	1	1	0	0	对.
	1	1	1	1	对
		1	1	0	对

① 先给 LSB



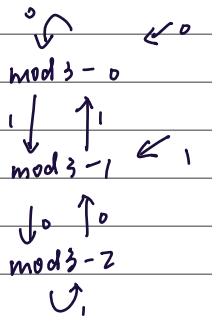
分数: 1, 2, 1, 2

1×2%3 2×2%3

↓ ↓

2 1

② 先给 MSB 当前分数 <= 当前分数 ×2 + 新来分数



reg[3:0] mod_remain [4] = '{ 1, 2, 4, 3};

=1

1×2%5 = 2

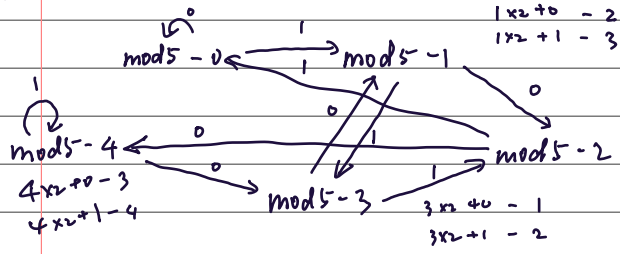
2×2%5 = 4

4×2%5 = 3

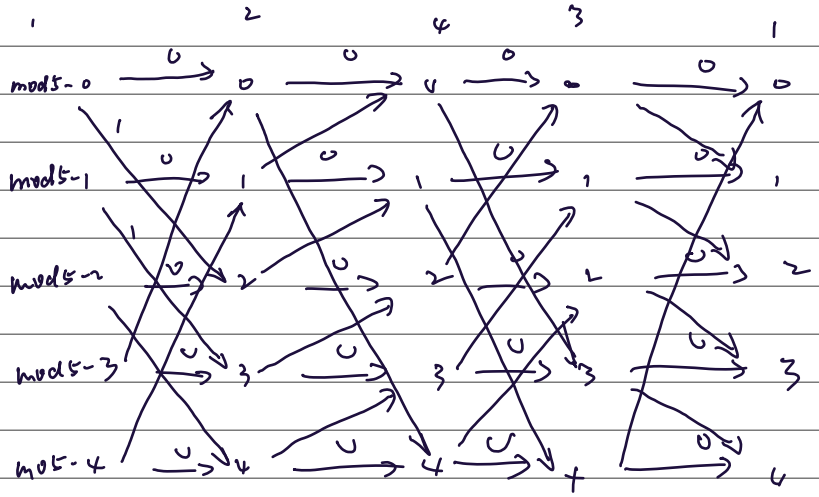
3×2%5 = 1

situation mod5:

先给 MSB



先给 LSB



```
module fixed_priority_arbiter (
        input  [7:0] req,
        output [7:0] grant
);
        assign grant = req & (~req +1);
endmodule

module rotater (
     input  [7:0] in,
     input  clk,
     output [7:0] out
);
    reg [7:0] shift_count;
    initial shift_count <= 0;
    always @ (posedge clk) begin
        if (shift_count == 0)
            out <= in;
        else
            out <= {in [7 - shift_count : 0], in [7 : 8 - shift_count]};
        shift_count <= shift_count + 1;
    end
endmodule
```

减号用于反向旋转.

控制优先级的核心就是控制 shift_count

若需要 non-preemptive，则可设置 if (最高位为1) 则不变

```
    end
  end
endmodule
```

4. Content Addressable Memory (CAM):

A CAM is a special type of memory where the memory unit is accessed based on the content rather than the address. In a CAM, you supply the content, and the CAM returns the address where the content is stored.

```
module CAM #(parameter WIDTH = 32, DEPTH = 256)(input wire [WIDTH-1:0] data_in,
input wire [WIDTH-1:0] search_data, input wire clk, input wire write_en, input
wire search_en, output wire match);
  reg [WIDTH-1:0] mem [DEPTH-1:0];
  integer i;
  always @(posedge clk) begin
    if (write_en) mem[i] <= data_in;
  end

  assign match = (search_en) ? (mem[i] == search_data) : 1'b0;
endmodule
```

5. Hardware Linked List:

A linked list in hardware can be implemented using a block RAM with two ports, one for reading and one for writing. The RAM can store the data of the linked list nodes and the index of the next node in the list.

```
module LinkedList #(parameter WIDTH = 32, DEPTH = 256)(input wire [WIDTH-1:0]
data_in, input wire write_en,

 input wire clk, output wire [WIDTH-1:0] data_out);
  reg [WIDTH-1:0] mem [DEPTH-1:0];
  reg [WIDTH-1:0] next_index [DEPTH-1:0];
  integer head, tail;

  always @(posedge clk) begin
    if (write_en) begin
      mem[tail] <= data_in;
      next_index[tail] <= head;
      head <= tail;
    end
    data_out <= mem[head];
  end
endmodule
```

For all the above examples, the code can be tested and verified by creating testbenches and applying different stimulus. You can also verify the designs using formal methods. Please note that the examples are simple and for illustration purposes only. Real-world designs may require additional features and more complexity.

```
module CAM #(WIDTH = 8, ADDR = 4 ) (
        input [WIDTH-1:0] data_in,
        input [ADDR-1:0 ] addr,
        input [WIDTH-1:0] search_data,
        output [2** ADDR-1 :0] out,
        output matched,
        input clk );
      reg [WIDTH-1:0] mem [2** ADDR-1 :0];
      always @ (posedge clk) begin
         if (wr_en) begin
             mem [ addr [mem_cache [addr]]] <=0 ;
             mem [ addr [data_in] ] <=1 ;
             mem_cache [addr] <= data_in ;
         end else if (search_en) begin
             genvar i ;
             generate
                 for (i=0 ; i< 2**ADDR; ++i) begin
                     out [i] <= mem[i][i];
                 end
             endgenerate
      end
endmodule
```

[$clog2 (WIDTH)-1:0]

```
integer :
for ( i=0 ; i < WIDTH ; ++i)
    if (onehot[i])   bin = i ;
```

```
module linked_list (input [7:0] data_in, append,
```

关于硬件链表，我认为若是实际做出"链表"两功能的话，例可实现

→ 插入（插入在某 index 节点后）

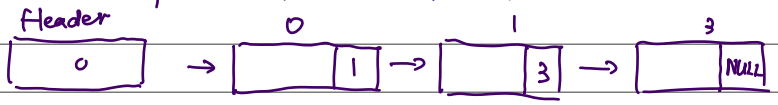
→ 删除（删除某 index 的节点）

，其它功能，如边界检查等可在实际实现。

Header	Data field	Next (ptr field)
0		1
1		3
2		NULL
3		NULL

Free list	
0	0
1	0
2	1
3	0

此链表含数据域和指针域，Header 中只有头节点下标



① 插入：若要在节点 i 后插入，　insert_index 为 i

(1) 从 free_list 中获取一个 free_index

(2) mem[insert_index].next <= free_index

mem[free_index].next <= mem[insert_index].next

mem[free_index].data <= data_in

② 删除，需注意，此种简单链表需遍历才能找到 prev_node.

特例是个问题点，解决方案：(1) 增加一个指针域用于保存 prev_node.

(2) 如果需删中头需特是删除第一个节点，由于第一个节点的 prev_node 总是 header，因此需额外做特判。

③ 也许，为了解决边界问题，加入 trailer 节点可能会更省资源。