Example Memory Block:
We will use a memory block example to demonstrate the verification concepts. The internals of the memory block will not be exposed here; instead we will treat it as a “black box” and examine the problem of how to verify it without knowing how it is built. Note that developing a verification test bench for the memory block is testing the logical behavior of memory block - assuming the actual underlying hardware is working. A totally different task is to develop memory test program that will test the memory for hard and soft defects that might exist in the real manufactured device due to the manufacturing process limitations.

The memory block has size of 1000 lines of 32 bits.

The behavior of the memory block is defined by the following table

<table>
<thead>
<tr>
<th>PIN VALUE</th>
<th>MODE DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME RE NAP OE</td>
<td></td>
</tr>
<tr>
<td>X X X 0</td>
<td>The output DO is in high-Z mode</td>
</tr>
<tr>
<td>X X 1 X</td>
<td>The memory is placed in low-power mode. Any activity shall wait at least 4 cycles after NAP is de-asserted for correct operation. Outputs are in high-Z mode</td>
</tr>
<tr>
<td>0 X X X</td>
<td>The memory is not active – no read or write</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>Data stored in address A provided at output DO</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>Data DI will be written into address A</td>
</tr>
</tbody>
</table>

NOTE: DATA STORED INTO ADDRESS A CAN NOT BE READ ON NEXT CYCLE
Part 1: The Verification Environment – The Testbench

The diagram shows the main elements making a verification testbench:

The following paragraphs describe key components in the Testbench environment.

**STIMULUS GENERATOR**

In order to test the model of some design, a verification program must apply test patterns to the input ports and observe the output ports over time to decide whether the inputs were transformed to the expected outputs. The generator component generates input vectors. For example, to verify a simple memory stimulus-generator generates read or write operations, an address, and data to be stored in the address if its write operation.

**Question 1:**
You are required to verify the memory ability to store data in all its addresses. Think about possible malfunctions that the DUT may have and how to catch them. Describe such a test in pseudo code.

**Answer 1:**

The following bugs may be involved in storing data in registers:

a. Register decoder does not work – write to other address or not at all
b. A change from Read to Write is taking too much time for the memory so write fails.
   c. A change of ME from 0 to 1 is taking too much time for the memory so write fails.
A test that writes unique data to each of the memory lines and then reads can verify the write was correct. Need to read after all writes were done to avoid case of same line selection. Need to run several types of cycles:

1. Write after write cycles – to verify no problem in adjacent cycles writes
2. Have intermediate cycles when ME is de-asserted – to test coming out of ME=0
3. Have intermediate cycles when RE is asserted – to test coming out of read
4. Have intermediate NAP cycles: 1 cycle NAP, 4 cycles wait, one cycle Write

# each of the tests below should start with
# Initialization sequence:
# ME=1; RE=0; CLK=0; OE=1; NAP=0
For A = 0 to MAX_ADDR:
  D=A
  CLK=1; delay
  CLK=0; delay

# Test 1: Write after write:
# ME=1; RE=0; CLK=0; OE=1; NAP=0
For A = 0 to MAX_ADDR:
  D=A
  CLK=1; delay
  CLK=0; delay
  RE=1
For A = 0 to MAX_ADDR:
  Assert (DO == A)
  CLK=0; delay

# Test 2: Write after ME=0:
# ME=1; RE=0; CLK=0; OE=1; NAP=0
For A = 0 to MAX_ADDR:
  ME=1
  D=A
  CLK=1; delay
  CLK=0; delay
  ME=0
  CLK=1; delay
  CLK=0; delay
  RE=1
  ME=1
For A = 0 to MAX_ADDR:
  CLK=1; delay
  CLK=0; delay
  NAP=0
  RD=1
  For i=1 to 4
    CLK=1; delay
    CLK=0; delay
    D=A
    RD=0
    CLK=1; delay
    CLK=0; delay
  RE=1
  ME=1
For A = 0 to MAX_ADDR:
  CLK=1; delay
  Assert (DO == A)
  CLK=0; delay

# Test 3: Write after Read:
# ME=1; RE=0; CLK=0; OE=1; NAP=0
For A = 0 to MAX_ADDR:
  RE=1
For A = 0 to MAX_ADDR:
  CLK=1; delay
  CLK=0; delay
  RE=0
  For A = 0 to MAX_ADDR:
    Assert (DO == A)
    CLK=0; delay

# Test 4: Write after NAP
# ME=1; RE=0; CLK=0; OE=1; NAP=0
For A = 0 to MAX_ADDR:
  NAP=1
  CLK=1; delay
  CLK=0; delay
  NAP=0
  For i=1 to 4
    CLK=1; delay
    CLK=0; delay
    D=A
    RD=0
    CLK=1; delay
    CLK=0; delay
  RE=1
  ME=1
For A = 0 to MAX_ADDR:
  CLK=1; delay
  Assert (DO == A)
  CLK=0; delay
DIRECTED TESTS

Knowing the bus protocol or design details a verification engineer can write tests that are directed towards testing known pitfalls of the design. That has been the only type of tests used for many years. However, with the explosion of variable and state space a more automatic and comprehensive approach was developed.

Question 2:
Assuming there is a counter in the memory that implements low-power mode. It counts 4 cycles after NAP de-asserted to enable normal read/write operation. Propose a test in pseudo code that will verify getting into and out of low-power mode.

RANDOM TEST GENERATION

Modern generators generate random, biased, and valid stimuli. The randomness is important to achieve a high distribution over the huge space of the available input stimuli. To this end, users of these generators intentionally under-specify the requirements for the generated tests. It is the role of the generator to randomly fill this gap. This mechanism allows the generator to create inputs that reveal bugs not being searched for directly by the user. Generators also bias the stimuli toward design corner cases to further stress the logic. Biasing and randomness serve different goals and there are tradeoffs between them, hence different generators have a different mix of these characteristics. Since the input for the design must be valid and many targets should be maintained, many generators use the Constraint Satisfaction Problem technique to solve the complex testing requirements. SystemVerilog, Vera, SystemC and Specman (products that support advance test benches) supports "constraints" to specify the legality of the design inputs. The model-based generators use this model to produce the correct stimuli for the target design. The stimulus generator should be intelligent and easily controllable.

Question 3:
Write a set of constraints that will allow generating randomly valid tests for the memory block.

Eitan,

Shouldn't it be:
not([Read] && PrevCycleWasWrite) or (......) ?

You wrote "write"..

Answer 3:
There are basically just 2 validity constraints:
1. Address range must match the memory size
2. After write to an address it cannot be used for read on the next cycle

Constraint 1:
A < 1000
Constraint 2:
Assuming we keep PrevWriteAddress the address of previous write and PrevCycleWasWrite flag.
not(WriteRead && PrevCycleWasWrite) or (PrevWriteAddress != Address)
**BUS FUNCTIONAL vs. TRANSACTIONAL MODELS**

The Bus Functional Model (BFM) for a device interacts with the DUT by both driving and sampling the DUT signals. A bus functional model is a model that provides a task or procedural interface to specify certain bus operations for a defined bus protocol. For a memory DUT, transactions usually take the form of read and write operations only. However, buses like PCIEx or DDR2 present much higher complexity both in the number of signals, involved protocol and semantic richness. Bus functional models have to follow the timing protocol of the DUT interface. BFM describes the functionality and provides a cycle accurate interface to DUT. It models external behavior of the device. For reusability, the implementation of the BFM functionality should be kept as independent of the communication to the BFM as it can be. The main drawback of BFM is in their performance. The need to drive all the bus signals with cycle accuracy places a huge burden on the logic simulator.

Bus Transactional Models are used for reducing the performance penalty in cycle accuracy. Their interfaces hide the details of the bus signals and present the protocol level only. Writing tests for that level is many times much faster and focus on the complex relationships of the bus state and activity.
**Question 4:**
Re-write your random test code using “nap”, “read” and “write” functions corresponding to the higher level events the model supports.

Answer 4:

```c
# init
For (a=0; a < 1000; a++) ExpectedRead[a]= -1;
cycleFromNap = 5;
prevWasWrite = 0;
prevWriteAddr = 0;
While (1) do:
    Type = random(Read, Write, NoOp, Nap);
    A = random(0 ... 1000-1)
    Case (type):
    Read:
        prevWasWrite = 0
        if (cycleFromNap < 5)
            Then cycleFromNap++;
        Else
            If (not prevWasWrite OR (A != prevWriteAddr) then
                D = Read(A)
                # Check Assert ((ExpectedRead[A] < 0) OR (ExpectedRead[A] == D)
    Write:
        if (cycleFromNap < 5)
            Then cycleFromNap++;
        Else
            D = random(0... 2^32-1)
            ExpectedRead[A] = D
            prevWasWrite = 1
            prevWriteAddr = A
    NoOp:
        cycleFromNap++;
        prevWasWrite = 0
        nop() # need to out the ME for once cycle to 0
    Nap:
        cycleFromNap = 0;
        prevWasWrite = 0
        nap()
```
 DRIVER:

Driver is a type of BFM. The driver translates the stimuli produced by the generator into the actual inputs for the design under verification. Generators create inputs at a high level of abstraction; namely, as transactions, like “read-operation” or “write-operation”. The drivers convert this input into actual design low level inputs (bits), as defined in the specification of the designs interface.

**Question 5:**
Implement in pseudo code the “read”, “write” and “nap” functions.

**Read(A)**
```
DUT.A = A
DUT.CLK = 1
Delay
D = DUT.DO
DUT.CLK = 0
return(D)
```

**Write(A, D)**
```
DUT.A = A
DUT.D = D
DUT.CLK = 1
Delay
DUT.CLK = 0
```

**Nap()**
```
DUT.NAP = 1
DUT.CLK = 1
Delay
DUT.CLK = 0
DUT.NAP = 0
```

**RECIIVER:**

Receiver is also a type of BFM, which collects the output of the DUT. The output of the DUT is available in a low level format. For example, consider a packet protocol, where the interface has "start of the packet" and "end of packet" signals to indicate the packet arrival. The receiver starts collecting the packet looking at the signal "start of packet" and does this job until the "end of the packet".
**PROTOCOL MONITOR:**

Protocol monitors do not drive any signals, monitor the DUT outputs, identifies all the transactions and report any protocol violations. The monitor converts the state of the design and its outputs to a transaction abstraction level so it can be stored in a ‘score-boards’ database to be checked later on. Again, let’s take a packet protocol. The monitor extracts the information from the packet, like: length of the packet, address of the packet etc.

**Question 6:**
What kind of high level events can be observed on our memory model?

Answer:
The events that can be observed are:
ValidRead, ValidWrite, NAP, OutputDisable, MemoryDisable (inverse of enable)

**SCORE-BOARD:**

Score-board is sometimes referred to as storage structure. The stimulus generator generated the random vectors. These are derived to the DUT. These stimuli are to be stored in scoreboard until the output comes out of the DUT.
For the packet delivery example, let’s consider a packet switch: the score-board may need record the packets going into the switch and their destination port such that when they eventually come out they can be verified. Score-board may also implement logic to compute the “expected” outputs of the DUT.

**Question 7:**
Describe what storage the score-board should include for the memory test.

Answer 7:
The scoreboard must store enough data to remember the state of the memory which includes the actual memory content, the number of cycles from last NAP. It also need to remember the state of the output enable.

**CHECKER:**

Checker is a part of the score-board. The checker validates that the contents of the ‘score-boards’ are legal. There are cases where the generator creates expected results, in addition to the inputs. In these cases, the checker must validate that the actual results match the expected ones.
**Question 8:**
Write down the specific checks that comprise the whole checker of our memory test.

**Answer 8:**

ScoreBoardChecks(LastOperationType, A, D, cycleFromNap, prevWasWrite, prevWriteAddr, ExpectedRead) {

Case (LastOperationType):
  Read:
    if (cycleFromNap > 5)
      If (not prevWasWrite OR (A != prevWriteAddr) then
        Assert ((ExpectedRead[A] < 0) OR (ExpectedRead[A] == D))

Write:
  # nothing to check on write
NoOp:
  # nothing to check on No Op

Nap:
  # nothing to check on NAP
}


Part 2: Coverage – or how good are the tests we run?

Coverage is defined as the percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project in order to make sure enough simulation cycles spent in verifying a design.

 Broadly speaking, there are two types of coverage metrics: those that can be automatically extracted from the design code, such as code coverage, and those that are specified by the user in order to tie the verification environment to the design intent or functionality.

Since in most designs it is impractical to write a single test that can sensitize all possible scenarios, it is common that the verification effort will include running thousands or even millions of random and directed tests. So the coverage information is collected and accumulated for multiple runs of the verification to represent the total effectiveness of the tests run.

**FUNCTIONAL COVERAGE:**

Functional coverage is a user defined metric that measures how much of the design specification, as enumerated by features in the test plan, has been exercised. It can be used to measure whether interesting scenarios, corner cases, specification invariants, or other applicable design conditions captured as features of the test plan have been observed, validated, and tested.

**Question 9:**
Define a comprehensive set of functional events for the memory model.

**Answer 9:**
ValidRead, ValidWrite, NAP, InValidRead, InValidWrite, OutputDeSelect, NoMERead, NoMEWrite, ReadAfterWrite, ReadDuringNap, WriteDuringNap, ReadInNapRecover, WriteInNapRecover

**CODE COVERAGE:**

Code coverage, in short, is all about how thoroughly your tests exercise your code base. Taken further, code coverage can be considered as an indirect measure of quality, defining the degree to what tests cover the code, or simply, the quality of tests. In other words, code coverage is not about verifying the end product’s quality.
Statement/Line coverage:
This is the easiest understandable type of coverage. It is defined as the ratio of the number of code lines executed to the total number of lines of code and according to the applied stimulus. For example in the following verilog RTL code when exercised with input where \( a > b \) only statements 1, 2, 3, 4 and 5 are evaluated.

```verilog
always @(posedge clk)
begin
  if (a > b) // statement 1
    begin
      y = a and b; // statement 2
      z = a or b; // statement 3
    end
  if (a < b) // statement 4
    begin
      y = a xor b;
      z = a xnor b;
    end
  if (a == b) // statement 5
    begin
      y = not b;
      z = a % b;
    end
end
```

As can be seen in the example, the only statements which will execute are those whose condition is satisfied. Statement coverage will only consider these statements.

Block Coverage
It is similar to statement / line coverage, but measures the coverage in terms of number of “begin – end” blocks which were covered.

Branch coverage
Branch coverage will report the true or false of the branch like if-else, case and the ternary operator (?:) statements. In bellow example, the execution of different branches of “case” statement depends upon the implementation of stimulus. The default branch in case statement in RTL is never exercised because the design guidelines insist usually to mention all the branches of the case statement.

```verilog
case (state)
  idle : .casez (bus_req)
  4'b0000 : next = idle;
  4'b1?? : next = grant1;
  4'b01?? : next = grant2;
  4'b001? : next = grant3;
  4'b0001 : next = grant4;
  default : next = idle;
endcase
```
Path Coverage
Sometimes interdependency between code blocks will expose a bug only when a specific path is executed. Path coverage reports the % of all possible paths executed by the tests run on the DUT. The following diagram shows multiple different paths that may occur in the simple example:

![Diagram showing multiple paths]

Expression Coverage
Examines the internals of a Boolean equation and counts for each cube the number of times it was evaluated to 1 and the number of times it was evaluated to 0. This coverage is important to make sure that effect of certain signals is not always masked by others.

Toggle Coverage
This is the simplest type of coverage report. It simply counts low-to-high and high-to-low transitions on every signal in the design.

FSM Coverage
Finite State Machines are a key design pattern in modern designs. Several different metrics are used to evaluate FSM coverage:

- State Coverage: Counts the number of times each state is visited.
- Transition Coverage: This type of check reports which arcs in the state diagram have been exercised.
- Sequence Coverage: Similarly to the Path Coverage, this report is used to verify that all paths through the state diagram have been tested.
Question 10:

a. Define using a state-diagram and/or Verilog code and/or other way a synchronous FSM with input “D”, which asserts its output “O” one cycle each time it recognizes the pattern 1110 in its input stream.

b. Write down an input stream for your FSM. Point out which are the first and the last bits of the stream.

c. Instrument your code with “Transition Coverage” reporting. Prove the FSM transitions through all possible arcs.

Part 3: ASSERTIONS or how to catch the bug before it grows

Assertions are small fragments of code that describe relations between design signals that are always required to be true. When the design is simulated these assertions are checked to evaluate to “true” and if not an error is generated.

So an assertion specifies a behavior of the system. Assertions are primarily used to validate the behavior of a design. In addition, assertions can be used to provide functional coverage and generate input stimulus for validation.

Modern verification environments use assertions to pass the information to the test bench and allow the test bench to react to the status of assertions without requiring a separate application programming interface (API) of any kind.

The main advantages of assertions are:

- **Capture the designer intents – by documenting assumptions**
  Large designs are hierarchical and many times involve not only interface between sub-blocks but also between different designers implementing these blocks. Many of these interfaces carry some assumptions regarding the possible combinations of the interface signals – during a single cycle or maybe even more complicated relations across multiple cycles. Assertions can be used to formalize these restrictions on the interface signals. A classic example would be an ALU interface with one-hot encoding of the current operation: ADD, SUBTRACT, etc. The control block which issues these signals will probably guaranty only one of them will be asserted. This behavior can be formalized in an assertion.

- **Improving bugs observability – especially since test time is finite**
  A bug in inconsistent behavior at some internal design point may take many clock cycles to propagate and cause real impact on the signals being observed by the test bench (it may be unit level or chip level test bench). On large designs the simulation is limited in time and thus the bug will never reach the test bench due to the limited number of clock cycles. An assertion based on internal signals of the block – deep in the hierarchy – will react immediately to the false condition – stopping the simulation at the cycle the problem occurred and not when it reached unit or chip interface.
• Bugs can be found earlier and are more isolated
  This argument is similar to the one above, but stresses the difference between debugging the “cause” and “effect”. A bug that cause some machine or signals to misbehave may cause some much more convoluted behavior when observed after hundreds of clocks. When inspected in the context of actual bug it is much easier to pin-point the cause of misbehavior.

Assertions can be used to capture the information about various levels of properties.
• Conceptual: can be used to verify system level properties which are more architectural level.
• Design: express unit level properties

Several kinds of tests can be applied:
• Conditional: It checks the some condition to be true using Boolean expressions.
• Sequential: Checks whether the properties are true using temporal expression (looking for sequence of events).
• Signal: Checks for signal types like ‘X’s in the design.
• Encoding types: Checks whether the encoding is violated (like in one-hot or gray-code relations between signals).

**Question 11:**
Propose at least one assertion that could be coded as part of the memory RTL code.