Packet-Data-Path Functional Verification Testbench: Stimulus Generation and User Interface

Mike Hibarger, Starent Networks
mhibarger@starentnetworks.com

Bassam Jamil Mohd, Synopsys, Inc.
bassam@synopsys.com

Jason Chen, Synopsys, Inc.
jasonc@synopsys.com

Benjamin Chin, Synopsys, Inc.
bchin@synopsys.com

ABSTRACT

The DUT receives and transmits packets to/from multiple interfaces. The testbench is able to randomly generate and self-check both valid and faulty traffic. The DUT has the flexibility to be augmented and evolved in future designs, and therefore the reusability of the testbench is essential to reducing the verification time of future projects.

The testbench was implemented using Synopsys' powerful testbench automation tool, VERA. VERA utilizes the OpenVera language to increase verification productivity. By taking advantage of an object-oriented language in OpenVera, the complexity of the testbench is encapsulated within different objects. Furthermore, with proper hierarchical organization of objects, testbench components can be easily reused.

By adopting multi-layered testbench architecture, its controllability is enriched and its usability is simplified. This allows tests with complex and different scenarios to be created with relative ease.

Keywords: Self-Check, Stimulus Generation, Directed, Pseudo-random, Fault Injection, Reuse, Multi-layered Testbench, User Interface

1 Introduction

The verification of the Packet-Data-Path presented many challenges for the verification team. Design complexity, aggressive schedule and construction of reusable testbench (TB) components were the main challenges. Building the testbench using conventional verification methods would have been painful and inefficient.

This paper discusses the implementation techniques used to build a reusable testbench, with a focus on stimulus generation and user interfaces. Due to the complexity of the design, many different traffic scenarios are required to ensure the correctness of the design. Achieving a complete set of the stimulus to test such a design becomes a major challenge. Complicated testbenches result in a steeper learning curve for writing tests. Therefore, it is desirable and essential to properly architect a testbench in order to provide an easy-to-use user interface for test writers. By using abstraction, we can free the test writers from having to be familiar with the intricacies of the testbench infrastructure. This allows the test writers to create the myriad of required tests very quickly. The additional overhead required to develop such a layered approach is quickly compensated for by the ease of test development.

This paper starts with a brief overview of the design. It is followed by a description of some requirements and challenges in building the testbench. It then discusses in detail the architecture and implementation of various mechanisms for stimulus generation in the testbench. The paper concludes with a discussion on the benefits achieved by this testbench.

2 Testbench Overview

2.1 Design-Under-Test (DUT)

The DUT is a packet processor, which routes packet cells from one port to another. Figure 2.1 shows the DUT simplified interfaces and its valid packet paths:

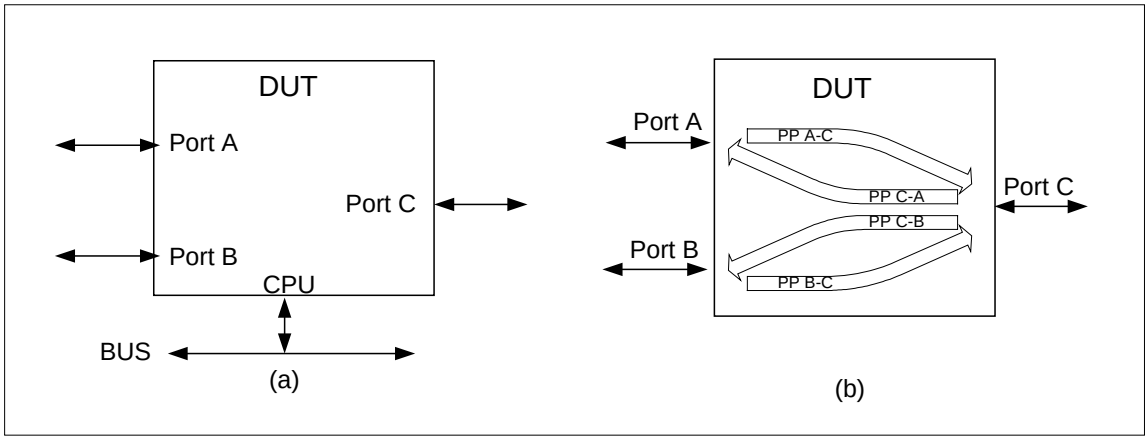

Figure 2.1 (a) DUT Interfaces, and (b) DUT Packet-Paths

- **The CPU Interface:** This interface transfers CPU commands used to configure DUT and access its internal registers.
- **Port A, B and C**: A port is an interface through which cells are transferred. Each port supports 8 logical channels carrying packets of one or more cells. Port A and B have the same protocol, and are different from port C.
- **Four Packet-Paths (PPs)**: There are only 4 valid paths where cells are transferred in the following directions:
    - PP A-C: Cells transfer from port A to port C
    - PP B-C: Cells transfer from port B to port C
    - PP C-A: Cells transfer from port C to port A
    - PP C-B: Cells transfer from port C to port B

The DUT also supports one of the following cell types:

- **Data Cells** carry packet data. Each cell has a header, payload and tail. The format of the header and tail is a function of the PP protocol.
- **Control Cells** transmit control information from the DUT to other devices. It is used in protocol supported by port A and port B.
- **IDLE Cells** carry no data. It is used in protocol supported by port A and port B.

2.2 Testbench Requirements

The following are some of the important testbench requirements:

- Testbench components:
    a. Autonomous BFMs and Transactors.
    b. Self-checking capabilities.
    c. Reusable components for future similar designs.
- Stimulus generation:
    a. Random and directed stimulus (traffic, packets) generation, to verify all features of DUT.
    b. Concurrent scenarios on interfaces generation.
- Tests:
    a. Simple and easy-to-use user interface for writing test.
    b. Easy-to-maintain test suites.

Among those requirements, we are going to focus on stimulus generation and the user interface of the testbench.

2.3 Testbench Architecture

The testbench is implemented with the OpenVera language. The OpenVera class structure is ideal for implementing testbench components because it hides complexity and has a clearly defined interface. Since testbench components are autonomous, the mailboxes and events are

very effective for communication. Packets and cells can be encapsulated inside the objects for easy data communication amongst testbench components.

In order to satisfy these requirements, a multi-layered approach was adopted when architecting the testbench. **Figure 2.2** shows a block diagram of the testbench architecture with emphasis on stimulus generation path. It organizes testbench components into different layer of abstractions.

Steps	Description
1	The only step that requires test writer intervention to define packet control and fault injection.
2	Fault injection object influences the packet generator to create cells with or without specific fault.
3	Packet control object signals packet generator when to start generating packets.
4	When a cell is generated, it mailed to RX_BFM.
5	RX_BFM forwards a copy of the cell to Scoreboard for checking later.
6	RX_BFM sends a cell to DUT.
7	TX_BFM receives a cell from DUT.
8	TX_BFM forwards a copy of the cell to Scoreboard for checking.
9	RX_BFM signals Packet Generator that it is ready to accept another cell.

Table 2.1 Brief Description of Each Steps Functionality

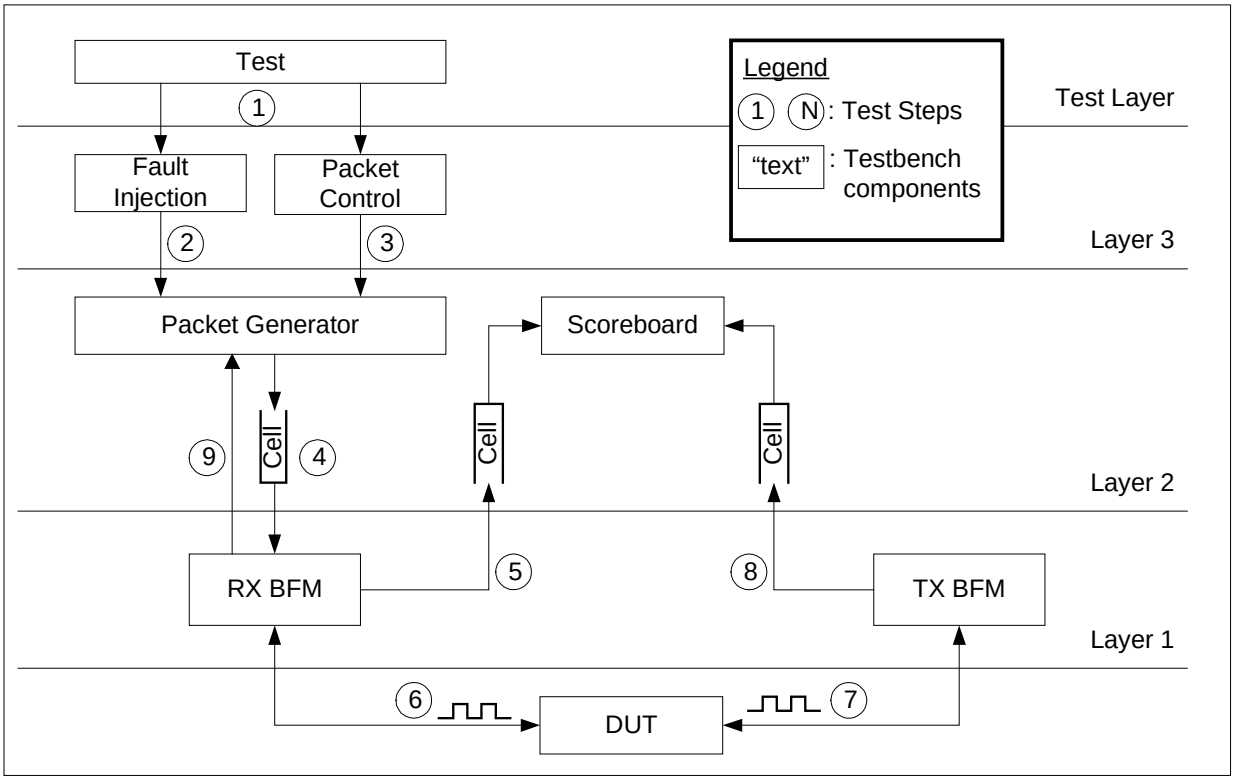

Figure 2.2 Simplified Block Diagram of a Partial Testbench

In the diagram, nine test steps are labeled. These steps are associated with the generation and the checking of a packet. They are referred to at various places throughout this article. Table 2.1 summarizes functionality of each of these steps.

The following is a brief description of each of the components in the block diagram starting from the lower layer of abstraction:

- **BFMs** (Bus Functional Model) are used to interact with the DUT. As shown in Figure 2.2, each BFM contains routines to transmit/receive data to/from the DUT. To transmit a cell, the cell is sent to the BFM via a mailbox (step 4). Once received, the BFM exercises the DUT I/Os to transfer the cell to the DUT (step 6). On the receiving end, when the BFM receives a cell from DUT (step 7), it mails the cell to the scoreboard via another mailbox (step 8).
- **The scoreboard** is a self-checking component, which receives cells from the BFMs (step 5 and 8). The scoreboard reconstructs the received cells into packets, and it then compares constructed packets to those generated by the packet generator. Error message will be reported in the case of mismatch.
- **Packet generator** generates packet cells and sends them to the BFMs to be transmitted to the DUT (step 4).
- **Packet controller** allows the test writer to configure the characteristics of the channel traffic. The configuration includes the specification of the number of cells, the maximum or minimum number of bytes in the cell payload, the channel priority, etc.
- **Fault injection** allows the test writer to specify the types of faults to be injected per channel, if any. A more detailed description of those components will be presented in the next section.

In Figure 2.2, the components are organized into three layers. Higher the layer of abstraction, higher is the level of intelligence built into the testbench to help hide complexity among different testbench components. As a result, the test writer (step 1) is shielded from the complex details of lower levels and writes tests at a higher level of abstraction that in turn, simplifies test development.

3 Stimulus Generation

From the testbench requirements, stimulus generation is a major challenge in this project. There are many different scenarios and types of stimuli that need to be created. The following subsections illustrate the different types of stimuli required to verify the DUT, and the implementation using OpenVera.

3.1 Valid Stimulus

3.1.1 What to generate?

A valid packet consists of single or multiple cells. Each cell consists of header and data fields. One could generate a directed or random packet. Information that can be randomized includes:

- Randomize different header and data fields in a cell.
- Randomize packet size.
- Randomize cells sequences.
- Randomize gap between cells.

The test for these types of stimulus should have a lighter traffic going through the DUT; in other words, neither the DUT nor the BFM will issue traffic control commands. Also, cells transferred to the DUT contain no fault.

3.1.2 How to implement?

Referring to Figure 2.2, a detailed packet flow is as follows: the packet controller will first signal the packet object to start creating packet cell(s) (step 3). When the cell is created, the cell object is mailed to the BFM (step 4). In OpenVera, a mailbox is a mechanism to exchange messages or data between processes. Data can be sent into a mailbox by one process, and retrieved by another process [3]. Once the bus is available, the BFM transmits the cell to the DUT (step 6). Next, the BFM sends an indication to the packet object that the cell is transmitted and it is ready for the next cell (step 9). The mechanism is implemented by OpenVera's trigger and sync constructs. When the "sync" is called inside the packet generator object, the process to generate the next cell is blocked until the process in BFM object sends a trigger to unblock it. [3]

Based on the requirement, we need to build a packet generator to support randomized and directed tests. When instructed, it needs to randomize different fields or parameters as mentioned before. The packet generator also supports directed stimulus generation, where a user can create some very specific packets to send across the DUT. These kind of basic tests are very useful to flush out all the data integrity errors. For example, a simple packet is good for testing the data integrity for the first time. Other more specific packets are useful to verify certain corner cases.

3.2 Fault Injected Stimulus

3.2.1 What to generate?

To verify the error recovery mechanisms implemented in the DUT, the test writer can inject faults in the transmitted cells. Injected faults are of three types: packet-based, cell-based, and protocol-based.

- Packet-based faults affect the cell sequencing.
- Cell-based faults affect the rest of the header and data fields in the cell, e.g. payload, CRC etc.
- Protocol-based faults affect the framing of cells at the interface level, e.g. signal toggling, framing errors etc.

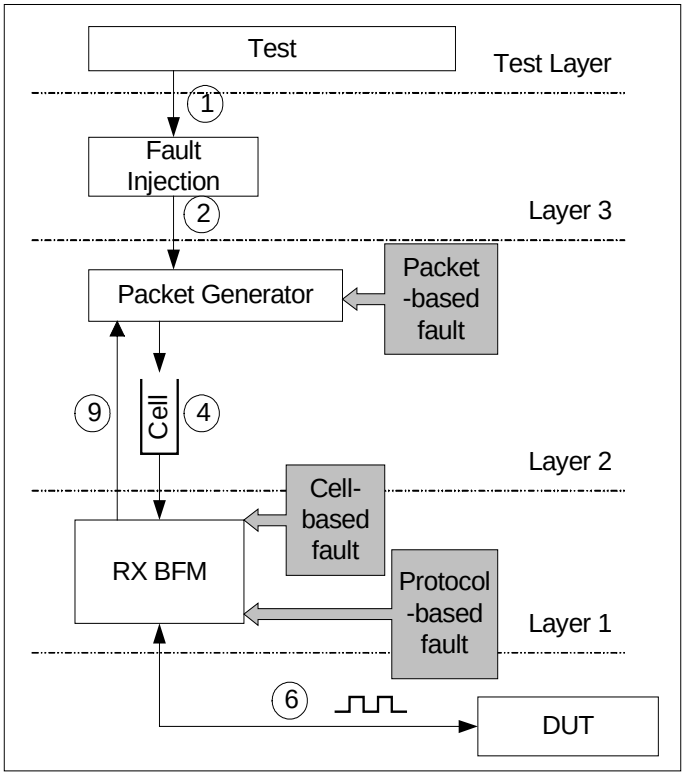

Figure 3.1 Fault Injected Stimulus Generation

3.2.2 How to implement?

Figure 3.1 shows the flow of fault injected stimulus generation. For each packet object, there is a fault object associated with it and influences the packet object fault injection (step 2).

The fault injection object uses OpenVera random, weighted-case construct to determine when to inject a fault. Once it has determined that, the fault object mails the information (of the desired fault) to the packet generator. The packet generator marks the currently created cell with the intended fault.

Once the faulty object is transferred to the DUT (step 6), the faulty object is sent to scoreboard (step 5) for result checking. The scoreboard will expect the DUT behavior according to the injected fault type.

Based on the above flow, different types of faults are introduced at different stages. A packet-based fault is introduced during cell type selection in the packet generator of layer-2, for example, replacing a “start-of-packet” cell type with a “middle-of-packet” cell type.

A cell-based fault is most likely introduced during the composition of a cell, in this case is the RX BFM model in layer-1 interfacing with layer-2. For example, a CRC error is injected when calculating the CRC value for the cell.

A protocol-based fault is at a lower level. It is usually introduced during the translation of the cell information into signals before transmitting them to the DUT, in this case is the RX BFM model in layer-1 interfacing with DUT. For example, one way of creating a framing error is by corrupting the special-character “start-of-frame” signals sending to DUT.

3.3 Traffic and Channel Control

3.3.1 What to generate?

For this project, several types of traffic and channel controls are required to represent a more realistic traffic in the network. The generation of traffic consists different streams of packets

creating interesting scenarios, and could be viewed as a higher-level stimulus. The following list several types of traffic scenarios which is interesting for the DUT:

- Backpressure traffic: Heavy traffic is sent to the DUT. Either the DUT or the BFMs will use flow control commands to control the traffic transmission.
- Channel characteristics: burst rate, arbitration modes, gap between cells of different channels, and other controls.

3.3.2 How to implement?

The backpressure scenario can be orchestrated in one channel or simultaneously in several channels using OpenVera fork/join construct. The following describes a sequence of controls of the testbench in order to create a backpressure scenario:

1. The destination BFM does not accept any DUT cells.
2. Next, the source BFM starts sending cells.
3. Once the DUT FIFOs are at the "almost-full-mark", the DUT asks the source BFM to cease sending cells using the Flow-Control-Stop command.
4. To emulate system response latency, the source BFM sends extra N cells after receiving the stop command, and then stops sending cells,
5. After sometime, the destination BFM informs the DUT it is ready to accept more cells.
6. When the DUT FIFOs are drained enough, the DUT sends Flow-Control-Resume-Tx command to source BFM,
7. The source BFM resumes sending cells until the test is completed,
8. During the test, the checker makes sure that the DUT does not discard or re-order channel cells.

The channel characteristics of a packet-path can be configured using a dedicated packet control in layer-3. Configuration parameters include, for example:

1. Number of packets per channel,
2. Maximum and minimum packet length (in Bytes), per channel,
3. Maximum and minimum cell length (in Bytes), per channel,
4. Channel arbitration: Since there are 8 channels (per PP) connected to one DUT port, channel arbitration is required to determine which channel can transmit cells. There are three arbitration modes:
    a. Full Random,
    b. Round-Robin,
    c. Constrained Random: Random selection based channel weights,
5. Maximum and Minimum burst (in cells): this determines how many cells a channel can transmit once it is granted transmitting cells on the port,
6. Maximum and Minimum gap between cells of channel,

The packet control will then control the packet generation using events (step 3 in Figure 2.2).

3.4 Benefits

- Directed stimulus is very useful in early debugging stages, as well as creating a hard to reach corner cases.
- Randomized stimulus greatly improved coverage.
- A thorough set of fault-injected stimulus will enhance the DUT error handling mechanism, and hence improves the overall design robustness.
- Traffic control could simulate system traffic behavior, and hence enables a more realistic system performance measurement.

4 User Interface

In order to generate many different types of stimuli and scenarios very quickly, a variety of switches are created to control the different components within the testbench. These switches are defined at a high level of abstraction and are designed from a system level perspective. By defining the switches at a high level, the test writer can easily control specific component(s) buried within the testbench hierarchy.

The following lists some requirements for controlling the testbench better:

- Testbench should be easy to learn and use.
- Testbench should provide good and meaningful control handles for test writer, in order to create flexible and interesting scenario.

Tests should be easy to create, and be as self-documenting as possible, to increase productivity. A self-documenting test makes the test code readable and the test scenario more readily understood. For example, by going through the test code, one could easily identify and understand all the settings and traffic configurations for a particular packet path.

4.1 How to implement?

All stimulus checking and generation in the testbench are done automatically. Once the test is run, the test writer does not have to manually investigate the results. Hence, the test writer can focus more effectively on controlling parameters of stimulus generation.

Referring to **Figure 2.2**, an additional layer, the layer-3, is created to hide the test writer from the need to know about the lower layers of the testbench that is more complicated and likely not the primary interest of the test writer. In this project, there are two objects in the layer-3: the fault injection object and the packet control object.

The fault injection object provides the necessary control to all three levels of fault-injections, the protocol, cell-based, and packet-based faults. The packet control object provides traffic control parameters, such as the cell size, the minimum and packet size, and the burst length. Effectively, the test writer describes the traffic characteristics by defining a simple table that is encapsulated in the packet control and fault injection objects (step 1 in **Figure 2.2**).

```
class packet_control {
  // public vars for channels
  // Channel arbitration weights, used when channel
  // arbitration is random
   integer   CHAN_ARBIT; // 0:Round-Robin (default)
                         // 1:Random
   integer   CHAN_ARB_W       [8] ;
   integer   MAX_BURST_CH     [8] ;
   integer   MIN_BURST_CH     [8] ;
    …

  // private vars
   rand  local integer rand_burst_length;
         local integer rand_burst_length_min,
                       rand_burst_length_max;
  …

  // Constraints
  constraint burst_length_const {
    rand_burst_length >= rand_burst_length_min;
    rand_burst_length <= rand_burst_length_max;
  }
  …

  // Private Methods
  local task  send_burst_cell( integer current_channel);
  local function integer channel_burst_length (integer
current_channel);
  …
}// class packet_control
```

Figure 4.1 Example code of packet_control class

```
program testbench
{
  …
//packet control for port A
  packet_control pc_rx_a;
  …
  test();
  …
 }

//////////////////////////
task test()
{
…
// CONFIGURE THE TRAFFIC
pc_rx_a.CHAN_ARBIT = 0 ;
//SETUP CHANNELS
// CHANNEL 0
pc_rx_a.MAX_BURST_CH[0] = 5;
pc_rx_a.MIN_BURST_CH[0] = 3;
…

}
```

Figure 4.2 Example code of testbench and test.

4.1.1 Example

Figure 4.1 shows a part of the source code for the packet_control class. The data member of the class consists of all the parameters for traffic shaping. In this example, it shows the control for channel arbitration, and channel burst length. The burst length of a channel is represented by a random variable, rand_burst_length, of type integer, and it is the private data within the packet control class.

As in a typical object oriented language, OpenVera classes have both public and private data and code. When a data or code is declared as “private”, it is inaccessible directly by program outside the class [3]; hence, the private data or code is protected. In this case, the value of the private data, rand_burst_length, can only be assigned and accessed by class packet_control. Its value cannot be corrupted by any methods outside of the class.

Another attribute to variable rand_burst_length is the randomization capability. The values of random variables can be controlled by constraint blocks. Constraint blocks limit the set of legal values that random variables can assume. These constraints are applied when randomize() method is called [3]. From the example above, burst_length_const is the constraint block for the random variable rand_burst_length.

Figure 4.2 shows the instantiation of the packet control object, pc_rx_a, in the testbench, and how a user controls the packet control object in a test. At the beginning of a test, a table that

describes the port and traffic characteristics for each channel is constructed. The example code illustrates how the arbitration, minimum and maximum burst length of a channel are set up. The channel arbitration scheme is round robin, and the burst length of a channel is a random integer between 3 and 5.

4.2 Benefits

- Easy-to-use user interface: The easier the user interface is, the higher the productivity of test development task. The test writer deals with only two types of control parameters that enable the construction of all kinds of traffic.
- Tests reuse and low maintenance: In the event when a new protocol replaces and supercedes the old one, the tests can be reused as there is no reference to the protocol-level components in these tests. The end results of reuse are to save time and reduce chances to make mistakes.

These directly contribute to improved quality and productivity.

5 Conclusion

Hierarchical, reusable testbenches provide many benefits. In addition to reuse and ease of use, they also provide greater accuracy in reducing the design development schedule. However, this does not come for free either. A well-architected testbench requires careful planning. All facets of the verification process should be taken into consideration before the testbench environment is implemented. The way the tests are written, the way new features are added, and the way the testbench is organized and stored should be thought through before undertaking the larger task of implementation.

A comprehensive solution results in the creation of a testbench that has appropriate levels of abstraction. Each component has the required scope for its task and allows the test writer to be insulated from complexities at lower levels. It also allows the lower levels to be uncluttered with features and controls desired by higher levels.

In a testbench with inadequate effort allocated to planning, the code becomes unwieldy and it becomes difficult to enhance it and maintain it. **Figure 5.1** shows a typical verification project schedule and compares the verification efforts with and without advance planning. The upper bar represents the project schedule with minimum effort in planning. In such a testbench, only some simple models in layer-1 and layer-2 have been implemented. These types of testbenches require more detailed and complex test writing and therefore elongate the test-writing phase of the project.

The lower bar represents a project schedule with advance planning. Although one needs to spend more time planning and developing more high-level testbench components and interfaces, it will save a lot of time during the test-writing phase. It will also save time as new features and requirements appear. Overall, the time spent on verification project with advance planning is reduced.

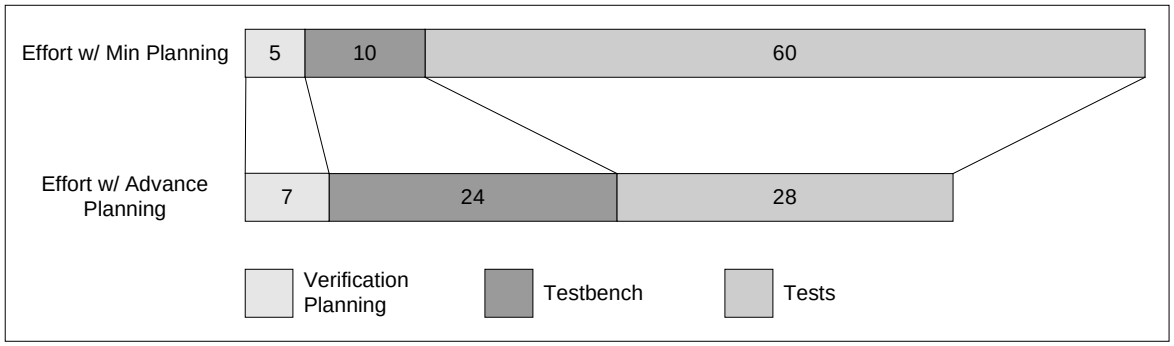

Figure 5.1 Verification Schedule

Also, providing the necessary hooks in the architecture to easily add routines to test complex corner case functions will allow the complete test plan to be achieved before the design under test hits the lab. Problems found in the simulation environment are easier to identify, fix and verify. If the problems are found in the lab, an unpredictable amount of time can be spent trying to capture and identify the problem, and then recreating and testing the problem in the lab becomes a secondary problem once problem has been fixed. Hence, the design verification environment should systematically sweep through all possible conditions before embarking on such an effort as seen in the lab.

By investing time and effort into developing a comprehensive test plan and carefully architecting the testbench, the design verification time is reduced as well as the time spent in the lab. This decreases the overall product design cycle time, and thus justifies the efforts of the design verification engineer.

Enabling and encouraging code and testbench reuse will jumpstart future projects and further decrease the design time of the testbench environment.

6 Acknowledgements

We would like to thank Parvez Khan and Alex Wakefield for their helpful and constructive feedback through the planning and implementation phases of the testbench.

7 References

[1] Faisal Haque, Jonathan Michelson, Khizar Khan, “The Art of Verification with Vera”, Verification Central, September 2001.

[2] Synopsys Professional Service, “Reference Verification Flow User Manual”, Synopsys Inc., May 2002.

[3] “VERA User Guide version 5.1.0”, Synopsys Inc., May 2002.