An approach for reference model implementation by predicting all possible output of design

Lirong Qiu*

School of Information Engineering, Minzu University of China, Beijing, China

Received 12 May 2014, www.tsi.lv

Abstract

In verification system, it is preferable to build reference model at transaction level which does not produce the output as the same latency as the design itself. But due to the lack of accurately modelling design's behaviour, there are some scenarios that design's output is different with reference model's output due to the different processing delay of stimulus. Scoreboard can get lots of comparison failure when it tries to do comparison between the output of reference model and design under such scenarios. In this case, neither reference nor design is wrong from functionality, but output comparison failure will mix up with the true design issue and bring trouble to the automatic check on design's behaviour. Cycle based reference model does not have such problem. But it usually takes great effort to implement cycle based reference models and maintain them. This paper provides its study on implementation style of reference model. By predicting all possible output of design, this paper presents a method for reference model to handle such stimulus competition scenarios at the transaction level. The paper also discusses the reference model's reaction effect on generator, which helps the test hit design's corner case.

Keywords: System Verilog, reference model, scoreboard, competition stimulus, coverage driven verification

1 Introduction

Verification usually consumes about 70% of the IC design's effort [1]. A lot of verification methodologies are proposed in recent years. Constrained-random stimulus is one the most important principles in IC verification methodologies [2].

Random base stimulus can only be generated automatically. For automatically generated stimulus, reference model or a scoreboard will be used to predict the results of the stimulus and compare these results with output of design in an automated way. Figure 1 shows the infrastructure of verification system with reference model. As illustrated in the Figure 1, the reference model and the design under verification are subjected to the same stimulus and their output is compared for discrepancies.

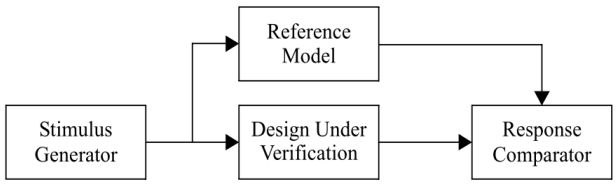

FIGURE 1 Common architecture of verification system

A reference model can be implemented at three levels of precision [3].

Reference models can provide transaction–level functionality.

Reference models can also be cycle accurate.

Reference models can provide rough justification by checking the validity of the DUT behaviour given some input and output. The DUT internals might be used for justification.

1.1 THE ISSUE OF TRANSACTION LEVEL REFERENCE MODEL

The transaction level functionality is the most commonly used reference model, so reference models are usually implemented with C, C++ and System C languages [1, 5]. It is thought that by using a common language the design and verification can proceed smoothly from system-level and architectural-specification down to detailed implementation [7]. However due to the lack of accurately modelling design's behaviour, some scenarios may make reference model and design have different output. Neither reference nor design is wrong from functionality under such circumstance, but output comparison failure will mix up with the true design issue and bring trouble to the automatic check on design's behaviour.

1.2 AN EXAMPLE WHICH HAS DIFFERENT OUTPUT BASED ON THE DELAY OF STIMULUS PROCESSING

Figure 2 shows the infrastructure of a simple design under verification and as followed its normally implemented reference model with C language. In this

* ***Corresponding author*** e-mail: qiu_lirong@126.com

example the design has 3 interfaces. Interface A is a port for data input. The input is a kind of packet with header, payload and CRC checksum. Interface B is a control interface. Controlling commands for design are injected by interface B. Interface C is for output. Design's behaviour is quite simple: it checks the input packets, discards the packets with CRC error and saves the payload of CRC correct packets into a local buffer. When a flush command is received from interface B, design will output the packets' payload in its local buffer through interface C. If local buffer is full, the input packets will be discarded.

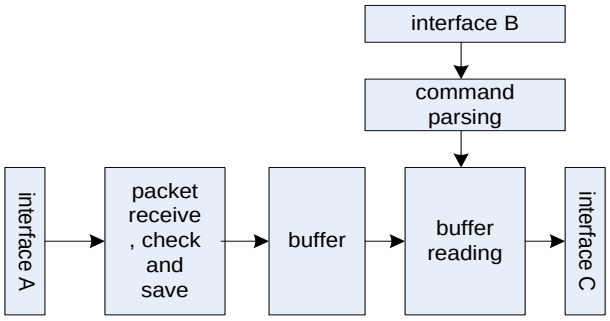

FIGURE 2 An example of design's infrastructure

Reference model for the example design is shown:

```
#include <stdio>
#define BUF_LENGH xx
typedef struct tagPacket {
    char header[4];
    char payload[1000];
    char crc[4];
    unsigned int length;
} MyPacket;
MyPacket Buffer[BUF_LENGH];
unsigned int ValidPacketNum;
void flush(MyPacket*  output, valid_length) {
  Output = Buffer;
  Valid_length = ValidPacketNum;
  ValidPacketNum = 0;
}
void packet_receive(MyPacket input) {
  Int i = 0;
  If(ValidPacketNum== BUF_LENGH){
   printf("buffer is full, packet is discarded\n");
   return;
  }
  If(!crc_check(input) {
   Printf(" CRC error is found, packet is discared\n");
   return;
  }
  for(i=0;i<4;i++)
   Buffer[ValidPacketNum].header[i]= input.header[i];
  for(i=0;i<input.length;i++)
   Buffer[ValidPacketNum].payload[i]=  input.payload[i];
  for(i=0;i<4;i++)
    Buffer[ValidPacketNum].crc[i] = input.crc[i];
  Buffer[ValidPacketNum].length = input.length;
  ValidPacketNum++;
}
```

Reference model is formed by two functions and the global variables which act as the local buffer and the valid packet number saved in the buffer. One function is to process the packets data and the other function is to process the command. As reference model is implemented at transaction level, it is not necessary to emulate the exact function of design. The data saved in the buffer can be packets type and it is not necessary to get packet header and appended CRC removed. The transaction level reference model is easy to be implemented and integrated from system level down to detailed implementation. Moreover, the simplification makes transaction level model has less bug embedded and can be entitle more confidence as a real golden model.

However, there is no timing delay for reference model to run the function task such as packet receiving, packet checking, command executing and buffer flushing. Due to the lack of timing delay, reference model may predict a different output as design does for the case that buffer is full and a flush command is coming shortly after another packet injection:

For reference model, the packet will be definitely discarded as buffer is full and flush command is not injected yet.

For design, the result depends on the delay of processing the packet and flushing command. For example, if design's behaviour is like this: when flush command is going on and the left byte is available for the new valid incoming packet, the incoming packet will be saved (not be flushed out. Chapter 4.3 will discuss the case of this packet's flushing out). Alternatively, it will be discarded.

Therefore, for real design, we get two different results based on stimuli's different processing time. If result is first one, it will be different with reference model's output. Under such circumstance we cannot say either design or reference model is wrong, because they both behave rightly according to the functionality. The most outstanding character of such scenarios is that the stimuli to be processed have competition. Who is the winner decides the processed result. In this paper, we call such scenarios as stimulus competition scenarios. In addition, the bottom of this issue is that reference model cannot process the packet or command in the same synchronization step as design. It is usually called as asynchronous issue between reference model and RTL. Stimulus competition scenarios in a verification system and asynchronous issue between reference models and designs commonly exist. Moreover, scenario of buffer flushing and packet processing under buffer full condition is also an important corner case should be covered by verification system. Although we do not care if the packet is discarded or saved, we do care if design will not hang up. This asynchronous issue must be worked around for our test target.

In fact, to work around this asynchronous issue between reference model and design, we have several choices to do: Use reference model with accurate timing

delay. Alternatively, make dedicated tests to test such scenarios and handle the output comparison specially. However, of them is perfect and neither both of them need more effort on reference model or test suites. This paper provides its study on implementation style of reference model. By predicting all possible output of design, this paper presents a method, which entitles reference model the ability of handling the timing sensitive scenarios automatically at the transaction level for testbench. The following of this paper includes:

How to implement reference model and handle stimuli competition scenarios at transaction level are discussed and an example is introduced based on the implementation method.

Several factors are introduced to optimize this method.

Moreover good test program need to provide more automation to maximize the functional coverage from each test case and reduce the time needed to create a test case [4 and 6]. So it is better to let verification system can handle such asynchronous issue automatically and hit such corner case easily. Chapter 4.2 also discuss an advantage about the method to implement reference model.

2 The approach to handle stimulus competition scenarios at transaction level

In previous chapter, we have pointed out that due to asynchronous issue, stimulus competition scenarios may make reference model and design have different output. The main reason is transaction level reference usually takes no simulation time to process a stimulus while RTL design cannot. Cycle accurate level reference does not have such issue, as it emulate each time cycle of design. To keep the transaction level feature, a good work around is that reference model emulate several important time stages of stimulus's processing instead of each time cycle.

2.1 EMULATE STIMULUS' PROCESSING AT STIMULUS' START AND END.

Cycle accurate reference model can totally emulate every time step of stimulus' processing. However, it is not what we want, as the effort is almost like to re-write a RTL design and such model is quite difficult to maintain when real design suffers a little change. In fact, what the reference needs is just to emulate some important time points of stimulus' processing. It is not necessary to care about every state of stimulus' processing at each cycle.

Two of the most important timing points in stimulus' processing are stimulus' start points and end points. At beginning reference model is in known state, we set it verified state VS. Once a stimulus is injected to reference model, we can look the injection operation as the start point of the stimulus and put it into a timing uncertain stimulus set, we mark it as U_SET. This uncertain stimulus set is bind to VS state. We mark this set it as U_SETVS. In addition, with simulation going on, VS state may evolve into several possible states due to stimulus competition scenarios. We mark the possible state as PS and manage all uncertain stimuli in an independent set, U_SET_ALL. U_SET_ALL include all stimuli, which are bind to different PS. We set all new added stimuli in U_SET_ALL as processing state for later use. Figure 3 shows the flow chart of managing injected stimulus by U_SETs (Different U_SET is bind to different PS or VS) and U_SET_ALL. To make it a more common solution, we assume reference model's starting state is in several possible states (PS). Each possible state (PS_i) has a U_SET bonded on it and marked as U_SET_{PSi}.

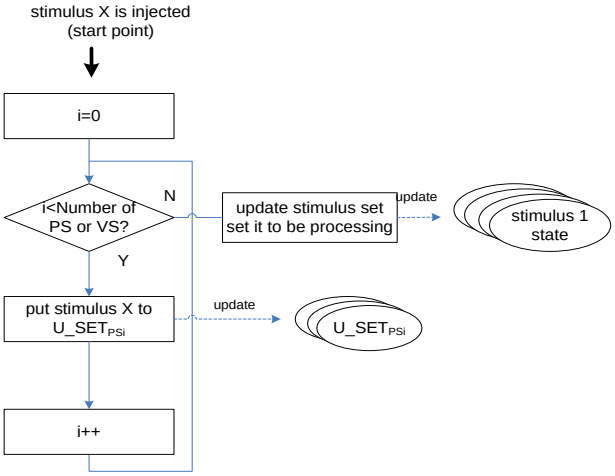

FIGURE 3 Manage input stimulus with U_SET_{PSi} and U_SET_ALL

When an event of design is monitored by TB, we make evaluation to see if any stimulus end point is relative to this event. A good case is we are sure some stimuli are definitely relative to this event. We mark these stimuli as processed in U_SET_ALL. However, a more complicated case is we just know some stimuli are possibly relative to this event. We will handle this complicated case directly. Reference model should try to process the U_SETVS (or each U_SET_{PS}). The process step is as followed.

Step 1: figure out all possible sequences combination in U_SETVS (or each U_SET_{PS})

Some combinations maybe illegal and should be excluded: for example, if two stimuli come from same interface, one must be injected later than the other. The sequence combination can be managed by a stimulus queue and the queue should be processed with its bind state VS as reference model's initial state. If reference model is in several possible states, each U_SET, which is bind to its possible state (PS) can make a bunch of queues. We managed this bunch of queues with a queue set marked as $queue\_set_{PS}$. For easy use of next step, we put all these queues into a big queue set ($queue\_set_{all}$). Each queue will be attached with a reference model's initial state $PS_{init,i}$. $PS_{init,i}$ is equal to queue's bind state (PS_i or VS) at start. We also attach $J_{veri\text{-}end}$ and $J_{min\text{-}end}$ to each

queue and set the queue as processing state for later use. The meaning of $J_{veri\text{-}end}$ and $J_{min\text{-}end}$ will be explained later. Figure 4 shows the flow chart of this step.

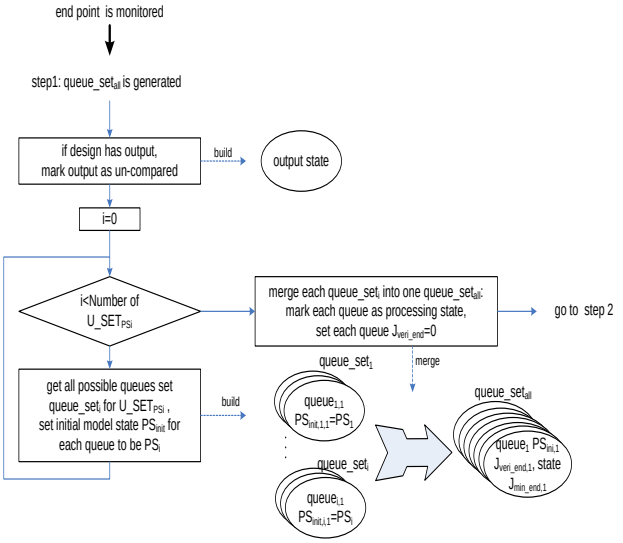

FIGURE 4 Generate the queues for all possible stimulus sequence combination

Step 2: Process all queues in queue_set$_{all}$ with their attached state as reference model's initial state.

For queue i, we process these stimuli one by one and from start to end. As a stimulus may influence design's output, we can compare reference model's output with design's to make sure if this stimulus's end point happens or has been processed by design. A variable $J_{veri\text{-}end,i}$ is used to record the position of latest stimulus whose end point is verified to happen. In addition, we need a variable TEMP_PS to record model's state for next stimulus's processing. It is updated after a stimulus's processing. According to comparison result, we get two routines depending on the comparing result: If reference model's output match with part or whole of design's output, evaluate this stimulus to see if current reference model's output is influenced by it. If the influence exists, update $J_{veri\text{-}end,i}$ to the stimulus's position and update reference model's state $PS_{init,i}$ to TEMP_PS. We also update this stimulus's state as processed in U_SET_ALL for later use. Then continue the next stimulus processing. One thing should be noted that to check if output is influenced by current stimulus is a case-by-case problem and should be carefully figured out during implementation.

If reference model's output does not match with any of design's output, stop this queue's processing and start next queue's processing.

During one queue's processing, we can check if design's output is all compared off. If it is, that will mean design's behaviour is right. If not and all queues are processed, that means some error happens and we need to check design or model in further. Figure 5 shows this step's flow chart.

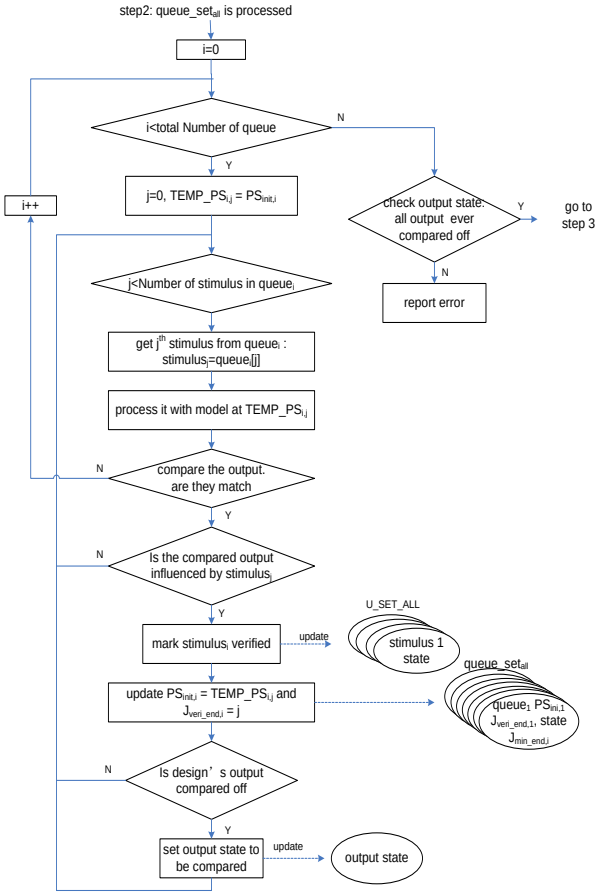

FIGURE 5 Process all queues in queue_setall to queue's end

Step 3: Process all queues in queue_set$_{all}$ to real $J_{min\text{-}end}$.

In previous step, we can figure out which stimulus's end point happens and already mark it as processed state in U_SET_ALL. Now we should process each queue again and update their attached state forward until latest stimulus which is marked as processed in this queue. For queue$_i$, the process can be start with $J_{veri\text{-}end,i+1}$ and initialized state $PS_{init,i}$, as $PS_{init,i}$ has been updated with $J_{veri\text{-}end}$ stimulus's processing in previous step. According to the state of each stimulus in whole uncertain stimulus set U_SET_ALL, we can find out the latest stimulus whose end point happens in queue$_i$. We mark its position as $J_{min\text{-}end,i}$. It is a similar flow as step 2: after one stimulus is processed, by comparing the output with design's, we get two routines depending on the comparing result.

If reference model's output match with part or whole of design's output, Update $J_{veri\text{-}end,i}$ to the stimulus position and update reference model's state $PS_{init,i}$. Then continue the next stimulus processing.

If reference model's output doesn't match with any of design's output, stop this queue's processing and mark this queue as discarded. And then start next queue's processing.

After all queues are processed, check the state of all queues. If they are all in discarded state, some error

happens and we need to check design or model in further. The flow chart is showed in Figure 6:

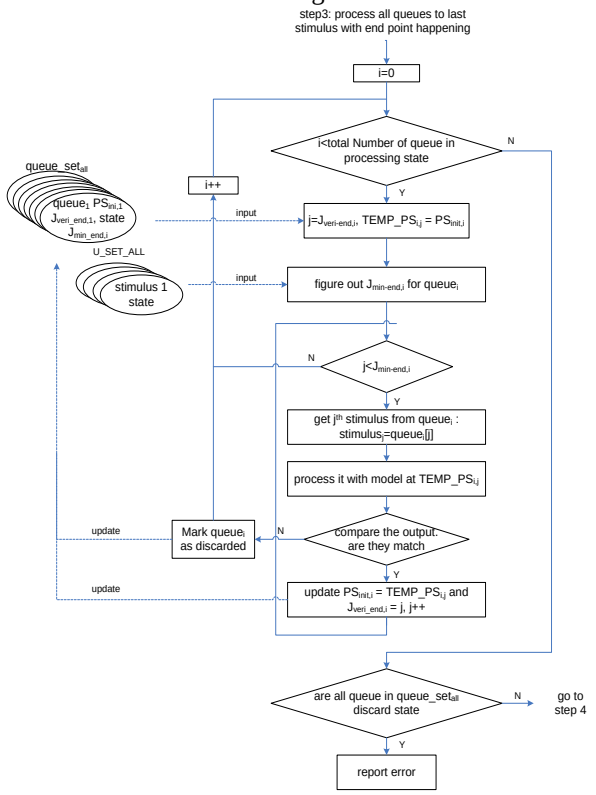

FIGURE 6 Process all queues in queue_set$_{all}$ to real $J_{min\text{-}end}$

Step 4: merge the left queues.

After step 3, for queue$_i$, reference model's state is updated to a new $PS_{init,i}$ and there are some stimuli (index from $J_{veri\text{-}end,i+1}$ to end) are left as new uncertain stimulus set which is bind to $PS_{init,i}$, $U\_SET_{PSinit,i}$. We can compare queue$_i$ with all other queues. If some of them are equal (their left stimuli are equal and binding state $PS_{init,i}$ are equal too), they can be merge to one U_SET_{PSi}. Then we get several new possible states, PS_i and new stimulus set U_SET, which is bind to it and is ready for next around processing.

2.2 AN APPLICATION FOR THE EXAMPLE DESIGN IN CHAPTER 2

Following this method, it is not hard to figure out a solution for the example mentioned chapter 2. Assuming current verified state VS is buffer full, packet A is coming and then a flush command B follows shortly after. The uncertain stimulus set U_SET_{VS} will have packet A and command B. So does U_SET_ALL. Due to flush command B, design will output packet payload in buffer and mark flush command B's end point. Therefore, we can set flush command B as processed. To make the competition stimulus scenario happen easily, we assume buffer can store two packets' payload.

Two possible sequences combination in U_SETVS are got: flush command B -> packet A and packet A -> flush command B. they build up queue_set$_{all}$ too.

For queue 1, after flush command B is executed, reference model will output data in buffer. TEMP_PS is buffer empty. $PS_{init,1}$ is updated to TEMP_PS, buffer empty state. Then for packet A, it will be saved according to TEMP_PS. However, design should not have packet A's payload as output. So Packet A is still in processing state and $J_{veri\text{-}end,1}$ keeps to be 1. For queue 2, packet A is discarded and reference model will output data in buffer after flush command B. Flush command B can be verified. $PS_{init,2}$ is updated to buffer empty state, $J_{veri\text{-}end,2}$ is set to 2.

By checking U_SET_ALL, we can figure out $J_{min\text{-}end,1}$ is 1 and $J_{min\text{-}end,2}$ is 2. So step 3 can be ignored. Now two queue's state is as followed: buffer empty and packet A is left in queue; buffer empty and no stimuli. They cannot be merged. Therefore, we get U_SET_{PS1} and U_SET_{PS2}.

Then come packet C, packet D and flush command E, which may compete with packet D. The new packet input is also an event of end point of previous packet. Therefore, packet A and packet C can be marked as processed. Here we skip the end event of packet A and packet C to ignore the procedure we do not care. For U_SETPS1 we have two possible queues: packet A-> packet C ->packet D->flush command E and packet A-> packet C -> flush command E -> packet D. For U_SETPS2 we have two possible queues too: packet C ->packet D->flush command E and packet C -> flush command E -> packet D. Now we can merge them into one big queue_set.

Queue 1: packet A-> packet C ->packet D->flush command E, $PS_{init,1}$ is buffer empty.

Queue 2: packet A-> packet C -> flush command E -> packet D, $PS_{init,2}$ is buffer empty.

Queue 3: packet C -> packet D -> flush command E, $PS_{init,3}$ is buffer empty.

Queue 4: packet C -> flush command E -> packet D, $PS_{init,4}$ is buffer empty.

For queue 1, packet A, packet C will be saved; Packet D is discarded due to buffer is full again; Packet A and packet C will be flush out due to flush command E. Now we can get sure state about packet A by comparison result with design's output. If two results match, packet A and packet C are set to processed state. Queue 1 is updated to the state that buffer is empty and no stimulus is left in queue. If results don't match, queue 1 will not be updated.

For queue 2, packet A, packet C will be saved and then be flushed out. Packet D is saved. By comparison result with design's output. If two results match, packet A and packet C are set to processed state. Queue 2 is updated to buffer empty and packet D is left in queue. If results do not match queue 2 will not be updated.

For queue 3, packet C and packet D are saved and packet C will be flush out.

For queue 4, packet C and packet D are saved and flushed out. As the output is different, we can make sure if packet D is flush out or not.

If packet A is not discarded in previous stimulus competition scenario, queue 3 and 4 will be marked as discarded state in step 3. Queue 1 and queue 2's states are quite like the states of two queues in previous competition scenario. If packet A is discarded, queue 1 and queue 2 will be marked as discarded state in first stimulus's processing and comparing. Queue 3 and queue 4 will make a duel based packet D's comparison. So only one of them survives and bring reference model to a determined state.

3 The consider factor in reference model's implementation

Sometimes there are too many stimuli in the U_SET, which will make too many PS states. Based on these new PS states, a new big U_SET may be got again. After processing these new U_SET, more second stage of PSs may be got. Chapter 3.2 has shown us an example, 4 queues and two PSs are got in second stimulus competition scenario. To record and maintain these PSs and U_SET is a complicated job. So better to do some optimization or trade off during reference model's implementation in a verification system.

3.1 REDUCE THE AMBIGUOUS TIME

For stimulus competition scenarios which may generate different result due to different time delay of stimulus' processing, time between start point and end point of the stimulus can be regarded as an ambiguous time. We do not know design behaviour definitely. If the ambiguous time is reduced, the number of stimuli in U_SET will be reduced and the number final possible state PS after all possible queues are processed will be reduced too. To do this, we need to mark the stimulus start point as later as possible and mark stimulus end point as early as possible. For example, set stimulus start point when the stimulus are totally injected to DUT if we are sure that design output can only be affected after the whole stimulus are totally taken in by design.

If internal signal of design is available to verification engineer, checking the internal signal of design is another good method. Although internal signal can be changed, some important signals are usually preserved if functionality of design is not modified a lot. Moreover checking internal signal means we can let reference model sync to design's timing step and can reduce ambiguous time to 0. This method is a trade-off. If too many internal signals are monitored, to maintain these signals will be another burden for testbench. A worst case is sometimes RTL is encrypted if verification system is developed by third part agent.

3.2 CONSTRAIN THE STIMULUS'S GENERATION

The PS or VS of reference model can be a good feedback to generator. When competition stimulus scenario happens and several possible results are got. By checking the state of reference model, the generator can be forced to generate stimulus, which can bring definitely determined state to reference model. For example in chapter 3.2, if generator finds that reference model is not sure about buffer's state as packet A may be discarded or saved, another flush command can be injected. Then by comparing flush output, packet A's state will be determined quite soon. However, with such constraint, some scenarios cannot be produced. Therefore, verification engineers should judge if lost scenarios are important for the design's function verification.

Another advantage of constraining stimulus's generation based on reference model's state is the expected corner case is easier to hit than normal randomized generation. For the example design given in chapter 2, buffer full can be a corner case. By checking buffer state in reference model, we can constraint to generate more packets and less flush command when buffer is nearly full. Then buffer full condition can be easily achieved.

3.3 SUBTRACT THE EVENT FOR STIMULUS'S START POINT AND END POINT

Subtract the right event for stimulus's start point and end point is another important factor for the method presented in this paper. In fact, from a broad concept, polling design's internal signal is a way to subtract the event of stimulus's start point and end point. However, by analysing design behaviour, some event can be subtracted just based on the input and output.

You will find that solution in chapter 3 will not work if we modify the design's behaviour like this: Interface C must be in a stable speed if data flush is ongoing. Due to buffer and speed of interface A, C are well defined, when the buffer is only flushed a little, available byte cannot afford the incoming packet, the packet will be discarded; when the buffer is flushed too much, to flushing out incoming packet may face the risk that left byte in buffer is flushed out but incoming packet is not fully received. Under such condition, design may not keep a stable flushing speed for interface C. So incoming packet will be just saved; when the buffer is flushed neither so less nor so much, the incoming packet can be flushed out.

The answer for such design change is not hard: if we subtract a stimulus like "enough byte is flushed out for incoming packet" which is after the flush command and adapt it to the solution, the problem can be solved again.

4 Summary

The method to implement reference model presented by this paper is to find all possible states and results during ambiguous time and figure out the final state and result after comparison with design's output. To do this, reference model will record all stimuli whose end points are not coming. By process all possible sequence for

these stimuli, design's possible state and output will be figured out. From 4.2 we also find an extra advantage to record these states, which help with the test to cover the specified corner case easily. This is another back up to implement reference model by using this method.

References

[1] Bergeron J 2006 *Writing testbenches using system Verilog* Springer.
[2] Fitzpatrick T, Salz A, Rich D, Sutherland S 2006 *System Verilog for Verification* Springer
[3] Rosenberg S, Meade K A 2010 A practical guide to Adopting the universal verification methodology (UVM). Cadence Design Systems.
[4] Haque F, Michelson J 2001 *Art of Verification with VERA* Verification Central
[5] Jindal R, Jain K 2003 Verification of transaction-level SystemC models using RTL testbenches. Formal Methods and Models for Co-Design, 2003 *MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on, IEEE (Mont Saint-Michel, France, 24-26 June)* 199-203
[6] Rowen C 2002 Reducing SoC simulation and development time *Computer* **35**(12) 29-34
[7] Clouard A, Mastrorocco G, Carbognani F, Perrin A, Ghenassia F 2002 Towards bridging the precision gap between SoC transactional and cycle-accurate levels *Proc. Design, Automation and Test in Europe (DATE, Paris, France, 4-8 March, 2002)*
[8] Carbognani F, Lennard C K, Ip C N, Cochrane A, Bates P 2003 Qualifying precision of abstract systemc models using the systemc verification standard. Design, Automation and Test in Europe *Conference and Exhibition, 2003, IEEE (Munich, Germany, 3-7 March)* 88-94

Acknowledgments

Our work is supported by the National nature science foundation of China (NO.61103161) and the Program for New Century Excellent Talents in University (NO. NCET-12-0579).

Author

Lirong Qiu, born on August 28, 1978, China

Current positions, grades: full professor of computer sciences at Information Engineering Department, Minzu University of China
University studies: M.Sc. in Computer Sciences (2004) and PhD in Information Sciences (2007) from Chinese Academy of Science.
Scientific interests: different aspects of natural language processing, artificial intelligence and distributed systems.