IDeALS for all – Intelligent Detection and Accurate Localization of Stalls

Pallavi Jesrani
Advanced Micro Devices, Inc.
2485 Augustine Drive,
Santa Clara, CA 95054

ABSTRACT

Deadlocks and livelocks hold a special place in the world of pre-silicon functional verification of today's complex pipelined systems. As we stretch the limits of speculation to achieve that coveted performance gain, functional defects in these complex system designs that can result in such stall scenarios can increase exponentially. These defects or bugs are extremely challenging to debug or root cause in a post-silicon environment. It is therefore crucial to detect and fix all such bugs in a pre-silicon environment before a design tapes out. In a system level pre-silicon environment, there may be several different bugs in one or more different register transfer level (RTL) units or blocks that may result in such stall scenarios. In addition to detection, localization is therefore also important. It helps to increase the debug efficiency, results in a faster turnaround of bug fixes, and helps to give an accurate picture of the health of the units at any given stage of the design. In this paper, we present a method to detect and accurately localize stalls caused by deadlocks and livelocks. This method can be used in a pre-silicon environment as well as in an emulation environment. We have implemented this method for a microprocessor core design and have observed promising results.

I. Introduction

In any complex pipelined system that consists of several components or blocks, it is possible to encounter functional issues that block forward progress in the system. These can arise out of either Deadlock or Livelock scenarios. Deadlock can be described as a failure condition in which processing is completely stopped as all components are waiting on one another and there is no forward progress. Livelock can be described as a failure condition where the same sequence of events is continuously repeated as processing retries or loops back to an initial state without making any forward progress. For the purposes of this discussion, we shall refer to both these conditions collectively as stalls. Functional defects in a system design that result in such stall scenarios are extremely challenging to debug or root cause in a post-silicon environment. It is therefore highly desirable to detect all such defects in a pre-silicon environment. There may be several different functional defects in one or different RTL blocks that may result in such stall scenarios concurrently. Hence in addition to detection, localization is also important. Localization means narrowing down the location of the RTL bug to a specific block. This helps to give an accurate picture of the health of the different blocks at any given stage of the design and also helps to increase both the debug efficiency and productivity. In this paper, we present a method for the Intelligent Detection and Accurate Localization of Stalls (IDeALS). The IDeALS method can be useful for any complex pipelined system. We have implemented this method for a microprocessor core design in a pre-silicon core level environment as well as in an emulation environment and have observed promising results.

2. Proposal - The IDeALS approach

Consider a system consisting of 4 Blocks named B1, B2, B3 and B4 as shown in Fig.1. The major flow of information/data is indicated by the arrows from B1 to B2 and so forth. Each block can include number of pipeline stages.

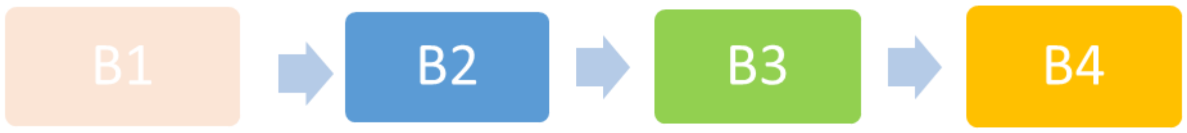

Fig. 1. Pipelined system consisting of four blocks

With the IDeALS approach, we propose to detect stalls in this system by using unit/block level timeout checks in conjunction with a system level timeout check. These checks can be implemented as assertions in the testbench. The block level timeout checks would be accompanied by further qualification as described below in sections 2.1 and 2.2 to ensure accurate localization.

2.1 Unit/Block level timeouts

The IDeALS approach treats each unit in the core engine as an entity (E) that receives information as input from one or more upstream units (UU), processes that information, and produces output for one or more downstream units (DU). Downstream units usually have a buffer in which this information is stored until they are available to process them. This is a common method used in complex microarchitectures to de-couple the pipelines of the individual units. If this buffer in a DU is full, E should not be sending any valid information to the DU. This is the correct behavior as per specification and not a bug in E. Hence, we need to account for back-pressure from DU as well to avoid false fails and incorrect localization.

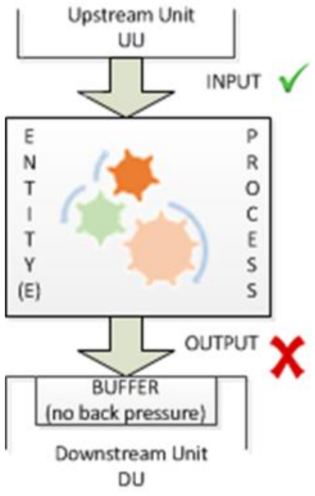

Fig 2. Graphical representation of IDeALS approach for unit timeouts

As per the IDeALS approach, a unit timeout fail should be flagged only if

- the unit has received input from upstream units,
- is not encountering any back pressure from downstream unit/s,
- but is still not producing any output for a predetermined (X) number of cycles.

The number X needs to be configurable at runtime so that when multiple such unit timeouts are used together at a higher system level, the values for the individual timeouts can be adjusted relative to each other via a runtime command line parameter. The above approach is graphically shown in Fig. 2.

2.2 Staggering block/unit timeout values used in a higher level system environment Unit/Block level timeouts

Complex modern designs do not strictly follow a top-down linear approach. A downstream unit may also contribute with input in many cases. Attempting to account for all such cases of secondary/tertiary input in a timeout check may result in verification code shadowing the RTL to an extent that may not be feasible to implement or maintain in the long term. Hence, we incorporate the following as a second level measure to avoid any mis-categorization due to lack of modelling of secondary inputs. We do this by staggering the unit level timeout values.

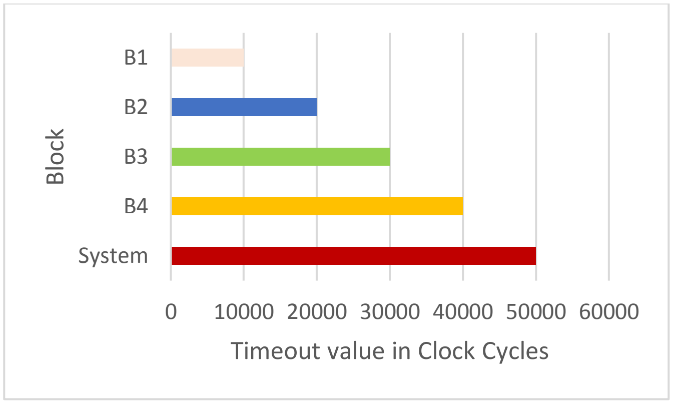

Fig. 3. Chart showing the staggered relationship among the different unit stall checks

The relationship chosen for the different unit timeouts is as seen in Figure 3, following the general flow of information in the system. The specific cycle numbers seen in Figure 3 can be changed but the relative relationship needs to be maintained in order to guarantee accurate localization.

2.3 System level timeout

In addition to the block level timeouts, we need a system level timeout to capture any fails that may potentially fall through them. Examples of this are:

- deadlock scenarios where all the units above are waiting on each other in order to make forward progress
- cases where the system is waiting upon some external trigger to make forward progress (ie i.e., waiting on an interrupt to wake up from a power management sleep state)
- any fall through cases arising from implementation limitations of unit level timeouts

2.4 Timeout Value Configuration

This section explains the factors to be considered during timeout value configuration. In the case of a functional RTL defect that results in a stall (a deadlock or a livelock), the stalled situation that does not resolve in 5,000 simulation clock cycles will not resolve in 500,000 cycles either. Hence, we need to find the sweet spot of the ideal timeout values. A good understanding of the individual blocks and system under test is invaluable for this purpose. In a system level environment, we can start from a value for the system level timeout check and work our way backwards. If the same unit level timeouts are also utilized in unit level testbenches, we can leverage from those values as well. The following factors can help in book-ending the values on the low and high side:

- If we start out with too small a timeout value, there are increased chances of false fails, that is, flagging fails where there is no deadlock or livelock but progress is slow due to legitimate reasons.
- Too big of a value, on the other hand, can result in the simulation failing with extremely generic fails like simulation or job timeout fails. It is also not efficient or preferable to flag a fail at 500,000 cycles when the same fail could be flagged in 100,000 cycles for instance, because we are then wasting precious simulation cycles.

Coming up with the ideal values is an iterative process. The runtime configurability of the values discussed in section 2.1 will be very important to help us get there and as long as the staggered relationship as discussed in section 2.2 is maintained, the localization will work.

3. Implementation

We implemented the IDeALs approach in our X86 microprocessor core design. The following sub-sections 3.1, 3.2 and 3.3 contain a description of how we applied the principles of this approach to our design. Fig. 4 shows a simplified high-level block diagram of the design under verification (DUV). The DUV is the core engine of a deeply pipelined, superscalar, out-of-order X86 microprocessor core.

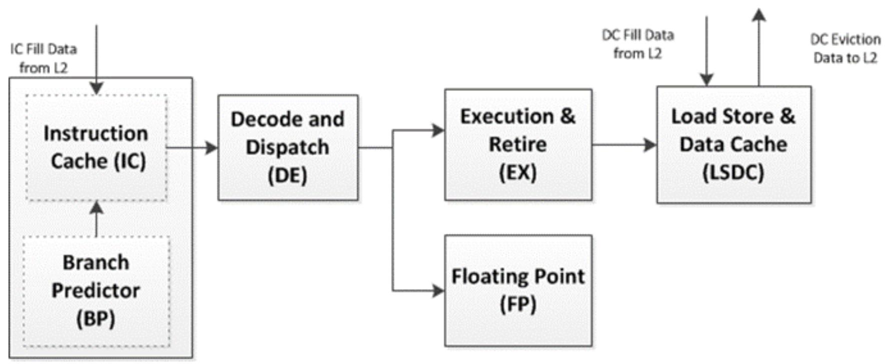

Fig. 4. Simplified high-level block diagram of the DUV

The different units in Fig. 4 are briefly described below. Branch Prediction (BP) predicts a sequence of fetch addresses and the specific bytes to fetch at each address and sends this information to the Instruction Cache (IC). IC gets page translation input from LSDC, fetches the specific bytes from the level 2 cache (L2), and sends them downstream to the decode and dispatch unit (DE). DE decodes the bytes from IC into instructions, converts them into micro operations and dispatches them into integer or floating point units downstream for execution. Execution and Retire (EX) handles integer execution and also includes the scheduling and retire logic. Floating point (FP) unit handles floating point operations. The load-store and data cache (LSDC) unit has the main task to service memory requests from the core. These memory requests (either loads or stores) are serviced either from the local data cache or are forwarded to the L2 cache. Microcode (not shown in above block diagram) implements various features of the x86 architecture in order to reduce hardware complexity.

3.1 Unit/Block level timeouts

Let us consider how we applied the rules of section 2.1 to the IC block as an example. With the IDeALs approach, An IC stall would be flagged only if:

- IC was receiving input from BP in the form of predictions and not waiting on getting a response from upstream units like a level 2 cache (L2) or a page translation (TLB) response from LSDC
- The Instruction byte buffer in downstream DE unit is not full
- But the IC still does not produce output in the form of valid fetch packets to send downstream to DE in X number of cycles.

A sample Verilog code snippet that reflects this is shown below:

```
IcTimeoutMax <= 10000;
IcTimeoutCounter <= 0;
always @ (posedge clk) begin
    if (ValidFetchFromIcToDe) begin
//resetting stall counter with every valid IC->DE transaction
            IcTimeoutCounter <= 0;
      end

    if (!ValidFetchFromIcToDe  && ValidBpPrediction && !WaitForL2Response &&
        !WaitForTlbResponse && !IbbFull) begin
            IcTimeoutCounter <= IcTimeoutCounter + 1;
      end

    assert (IcTimeoutCounter < IcTimeoutMax) else $error ("IC Stall detected");
end
```

3.2 Staggering block/unit timeout values used in the core environment

Applying the rules of section 2.2, the relationship chosen for the different unit timeouts is as seen in Fig. 5, following the general flow of information in the core engine.

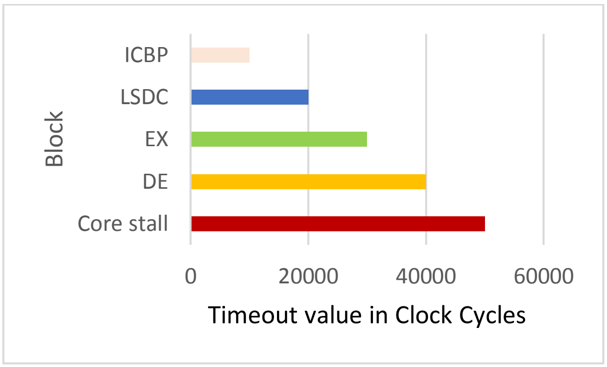

Fig.5. Staggered block/unit timeout values used in core environment

The exception here is the DE dispatch timeout that has a higher timeout limit than the retire timeouts seen by LSDC and EX. The reason for this is the dual role of DE in performing Decode and Dispatch functions. While the Decode logic requires input from IC, the Dispatch logic requires downstream machine resources to be available. This is only guaranteed if ops are retiring regularly and freeing up the resources. Therefore, before flagging a Dispatch timeout, we need to ascertain that the Retire logic is operating correctly.
The values needed to program each unit timeout in a higher level environment may need to go through an initial iterative process and the ability to configure them at runtime without any code changes will be invaluable to find the right mix.

3.3 System level timeout

This was implemented as a Core level instruction stall.

4 RESULTS

Following are the results of our implementation of the IDeALs approach.

4.1 Accuracy:

Table 1 shows the results observed so far in the format of a confusion matrix. Each row represents the bugs found by a specific unit timeout check. Columns represent the different units. The diagonal elements indicate the percentage of bugs where the localization was correct. (i.e., the unit that flagged the timeout/stall assertion was indeed the unit that had the RTL bug)

Table 1

		Unit with RTL bug			
		ICBP	**LS**	**EX**	**DE**
TIMEOUT CHECK	ICBP stall	88%	7%	4%	1%
	LS stall	0	96%	2%	2%
	EX retire stall	0	40%	50%	10%
	Dispatch stall	25%	0	0	75%

- For the most part, localization was accurate. This is evident from the green boxes along the diagonal. The exceptions are explained as below:
- EX retire stall check had a limitation whereby it was not accounting for the lack of input coming in from the LS and hence ended up finding some LS bugs in the process. This is the reason for 40% LS bugs found by this check.

4.2 Effectiveness

Fig. 6 below shows the effectiveness of the checks - that is, the number of RTL bugs found by a check versus the number of verification bugs (bugs in the check itself).

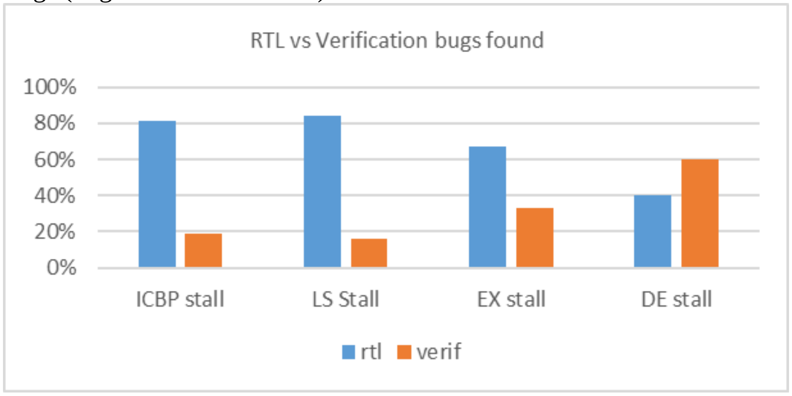

Fig. 6. RTL vs Verification bugs found by the unit timeout checks

- The DE stall check was not able to account for inputs coming from other units in the form of availability of tokens needed for dispatch due to the complexity of the logic. This resulted in the higher number of verification bugs.
- The Ex stall check encountered false positives due to the limitation described in section 4.1

5 IN PROGRESS AND FUTURE WORK

5.1 Dynamic scaling of timeouts to minimize false positives Effectiveness

There may be legitimate, non-defective situations in which forward progress is very slow but not entirely stalled. For example, an instruction that includes one or more loads from a non-cacheable memory location may just take a long time due to the long latency of accessing the main memory. The firing of a timeout check for such a condition constitutes a false positive.

It is essential that we eliminate or minimize the false positives because they hurt debug productivity and reduce the effectiveness of the timeout checks. One of the ways to do this would be to dynamically scale the relevant timeout check on detection of such a scenario. This information also needs to be passed on to the downstream timeout checks to prevent them for firing falsely as well.

5.2 Fine Grained

So far, we have localized the bug resulting in a stall fail to the specific unit of origin. It is possible to further refine this approach to a specific flow inside the unit. For example, the LSDC timeout check could be enhanced by adding more information about the stalled operation, such as whether it is a load or a store operation, with a cacheable or uncacheable memory type [1] and much more. The EX retire stall can be enhanced by differentiating between a floating point operation or an integer operation. The core instruction stall can include the instruction name in the error message.

5.3 Use in an Emulation environment

The unit timeout checks described in Section 2.1 can be implemented in the form of synthesizable System Verilog checks. This would allow them to be used in an emulation environment where the observability and visibility is very limited to begin with.

6 CONCLUSION

In this paper, we present the IDeALS approach for detection and localization of stall fails. The results from our implementation of this approach are promising.

This methodology has helped to redirect fails to the relevant experts, minimize cross-unit shuttling of bugs, and thus improve debug efficiency. It has also provided additional visibility into the health of the design and thus made it easier to detect pain points at a glance even before the fails have been debugged.

ACKNOWLEDGMENT

The results presented in this paper have been obtained by the combined effort of many talented engineers across multiple AMD locations.

REFERENCES

[1] AMD64 Architecture Programmers Manual Volume 2: System Programming, AMD, 2019.