Novel Adaptive CPU Scoreboard Methodology for a Multi-language environment

Pooja Madhusoodhanan
Texas Instruments, Bangalore, India
pooja.m@ti.com

Saya Goud Langadi
Texas Instruments, Bangalore, India
saigoud@ti.com

Labeeb K
Texas Instruments, Bangalore, India
labeeb@ti.com

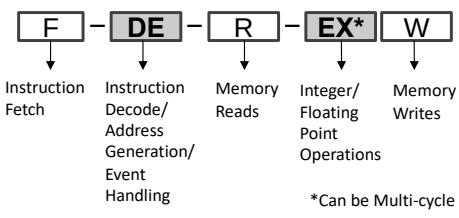
***Abstract*- Modern CPU design is taking in more instructions per cycle while supporting numerous optimization methods to push its performance. This makes CPU verification one of the most complex challenges for a DV engineer because of the various corner cases that are introduced. Simulation based approach using an accurate reference model, is the usual practice, to verify sequencing of instructions. However, CPU performance improvement being a continuous process, complex modifications in the design are difficult to replicate in the reference model without being error prone or at risk of imitation.**

This paper proposes an adaptive scoreboard methodology for truly functional verification of CPU core, with tolerance for performance variation between the DUT and the reference model. The goal is to minimize the complexity in the reference model, in terms of external variations as well as accuracy, while being able to use it for verification. This is done by tracking the key aspects of the pipelined transactions, as the CPU under discussion has a multi-stage protected pipeline that can support 600+ instructions and can execute many of them of them per cycle.

I. Introduction

A. CPU Architecture

The CPU design under consideration has a multi-stage fully protected pipeline that can be simplified as in Fig. 1, illustrating pipeline protection between I2 evaluating in EX phase, and I3 evaluating in DE phase. The hyphenation in the pipeline stages denotes decoupling, i.e., the pipeline can partially move forward if there is an instruction available in that phase and the next phase is able to receive it.



CYCLE	DE	R	EX	W
1	I1			
2	I2	I1		
3	I3	I2	I1	
4	I3		I2	I1
5	I3			
6		I3		

Fig. 1 CPU Pipeline Phase Schematic

Also, in order to facilitate the program flow, CPU has the following resources available internally.

- Ax registers used for address computation. These are mostly sampled/updated in the DE phase, except for the case of loading Ax from a memory location which happens in EX phase.
- Dx registers used to carry out ALU operations. These are always sampled/updated in the EX phase.
- Mx registers used to carry out advanced ALU operations. These are always sampled/updated in the EX phase.
- Status registers, used for capturing status flags during the D2 or EX phase operations, and event flags.
- Other miscellaneous registers like Program Counter (PC), Stack Pointer (SP), Return Program Counter (RPC) etc.
- For some complex operations, there are multi-cycle instructions which update the Dx/Mx registers.

As the CPU has a VLIW architecture, the packing of multiple instructions into a single packet is possible. The

assembler makes sure that overall size of the packet is within maximum permitted limit and honors the constraints on availability of limited resources (registers/ sub-blocks). If in case there is a pipeline hazard due to phase difference in handling of a resource, the DE phase of pipeline will be stalled until the hazard is resolved.

B. Testbench Architecture

The multi-language testbench built around this CPU is depicted in Fig. 2, with a System C reference model which was adopted from the architecture team, and a System Verilog wrapper encompassing the DUT. Both the TOP modules have their own Program memories preloaded with Instruction opcodes, derived from the assembly which is either manually coded or randomly generated. They also have separate data memories for read-write operations. Several external triggers affecting the functioning of the DUT are introduced, for example, debug events such as HALT, RUN, STEP, power up-downs, interrupt events such as NMI and INT, and memory stalls during fetch as well as read and write operations.

CPU performance improvement being a continuous process, complex modifications in the design are difficult to replicate in the reference model without being error prone or at risk of imitation. Hence, the reference model and the DUT are not designed to match cycle by cycle. In several scenarios, the DUT would perform a smidge better than the reference model.

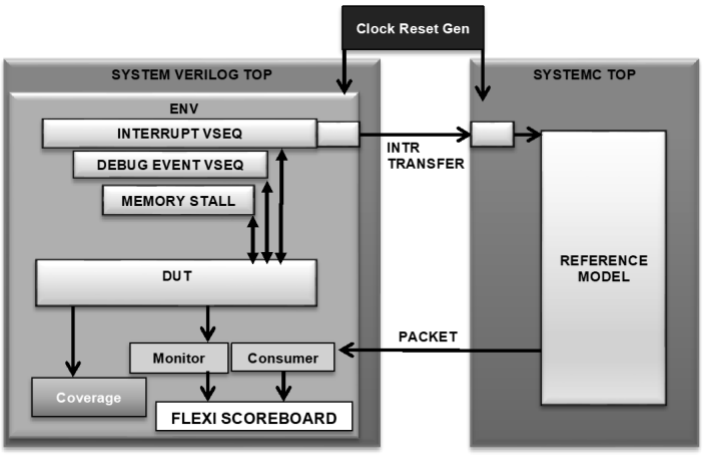


Fig. 2 Testbench Schematic

C. Interrupt Architecture

Interrupts result in a change of context as shown in Fig. 3, where the code execution branches into the ISR routines, and returns after restoring resources to their original state. Interrupt entry is guarded by several configurations and pipeline protections, depending on whether it is an NMI or INT. Nesting of Interrupts is supported up to a limit.

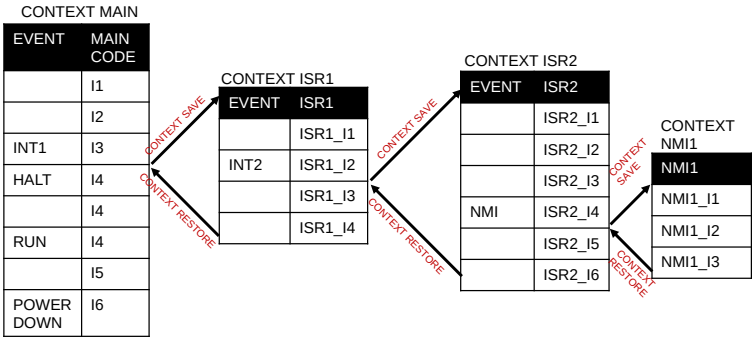


Fig. 3 Context Transition due to Interrupt events

II. CHALLENGES

A. *Resource disintegration*

Since the reference model and DUT are not cycle matching, any comparison which uses a single snapshot of all the registers will not work. As the pipeline is decoupled at multiple points, it is possible that two registers that are updated together in one case, are updated a cycle apart in the other. For example, if the fetch of I3 (MV A1,0x4) is delayed, the updates due to instruction I3 will happen after I1 (LD A5,0x3) has completed execution, as shown in Case 1 and 2 of Fig. 4.

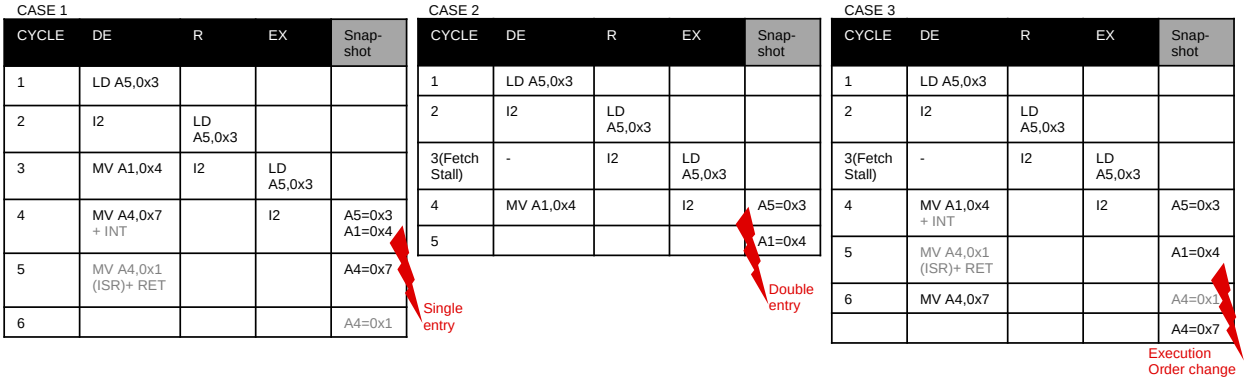


Fig. 4 Snapshot mis-comparison

To address this, the scoreboard would have to look at each register standalone and ensure it has been updated with the correct value, at the correct time, with respect to the instruction that caused the update.

B. *Event disintegration*

The effect of events such as HALT, Interrupts, etc, depends on the state of the DUT or model at the time. Their acknowledgment is guarded by several conditions, as previously mentioned. Since it is not guaranteed for the DUT and reference model to be in the same state at any given time, a simultaneous event trigger may be immediately acknowledged by one, whereas the other may consume instructions before it acknowledges an interrupt. Thus, the course of execution cannot be compared linearly. This is illustrated in Case 1 and 3 of Fig. 4, where the Interrupt (INT) is fired on the same cycle in both cases. Case 1 takes the interrupt after I4 (MV A4, 0x7), whereas the Case 3 takes the interrupt before I4.

C. *Simulation approach*

To simplify the reference modelling effort and synchronization of external factors, we took the following steps:

- Developed an adaptive scoreboard that can compare the state of the CPU despite cycle differences. In our case, the state is determined by the contents of all the internal resources.
- Apart from interrupts, triggered all events such as HALT, RUN, STEP, power up-downs, correctible errors, and memory stalls, only to the DUT. The effect of such events on the CPU is similar to a performance delay and doesn't modify the course of execution.
- Introduced interrupts in both setups but built the adaptive scoreboard to adjust to different acknowledgement times and consider the current context, as interrupts cannot be triggered in a manner that the execution course of both setups be the exact same.
- Setup a performance check with margin of error, which accounts for the delay caused by the different events introduced.

III. METHODOLOGY

A. *Principal Framework*

The Adaptive Scoreboard collects the contents of the registers and memory buses, throughout the CPU execution. A queue is maintained per register, adding an entry every time a resource is written, compared between the DUT and model. Register and bus updates are detected not only by content change, but also by directly probing the write enables of the Resources. This ensures that duplicate writes are also detected along with content change. At the end of test, queues are emptied and checked for size differences.

While this ensures data correctness, it raises the question of whether the register has been updated at the correct time. A single instruction updating 2 registers must not update one before the other. To address this, the PC is chosen as a reference point to track all the register updates. For each update, the phase that caused the update is determined, and the PC associated with it is backtracked to DE phase and linked with the register queue. Typically, the phase is determined based on the port from where the update was triggered, which in turn depends on which instruction caused it, as illustrated in Fig. 5.

Fig. 5 PC monitoring based on phase of update

The PC comparison indirectly checks if multiple updates due to a single instruction packet are synchronized. While backtracking the PC, we need to consider if there were any pipeline stalls. In case one of them is delayed, or is not consistent with the pipeline stall controls, there would be a null or incorrect PC at the point in the history, indicating that the timing is not right. To enable this backtracking, a PC history is maintained by the scoreboard up to the maximum multi-cycle instruction that is supported by the architecture.

In addition to PC, the time of update is also queued up for debuggability and reporting. Fig. 6 depicts the multi-dimensional queue structure required for keeping track of all the aforementioned information.

Cycle	PC	Update Phase	Register Update	
1	0x20	DE	A1=0x1	DE[PC] = 0x20
2	0x22	EX		
3	0x26	EX		
4			A1=0xa4	DE[PC] = 0x22
5			A1=0x30	DE[PC] = 0x26
6	0x58	DE	A1=0xc1	DE[PC] = 0x58

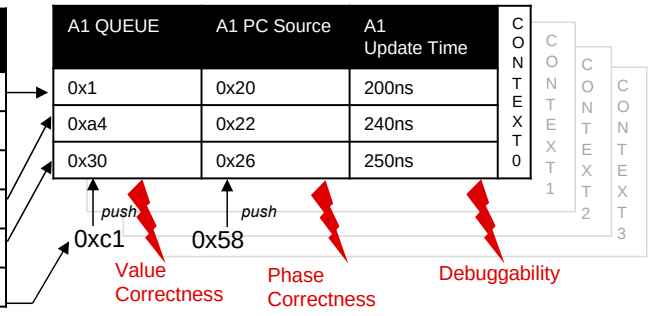


Fig. 6 Scoreboard Queue Structure Illustration

B. *Interrupt Context Change Extension*

As previously discussed, interrupts can cause execution order to change, due to CPU and model being in different states. This means that resource updates coming from interrupts cannot be linearly entered in the queue and be expected to match. Hence, for each interrupt, we create a new parallel context queue, identical to a call and return. This queue will be used only for that interrupt and will not be overwritten once the interrupt has returned. This is done because nesting equivalence cannot be assumed. For example, the same interrupt may be nested in one scenario and may not be nested in another. If we reuse the queues on interrupt return, Context1 queue will be filled in the non-nested case, and Context2 queue will be filled in the nested case, thus resulting in a mismatch. With our proposal, this can be avoided. Thus, we have an array of queues to compare, each having same checks as main queue, i.e., ‘Context

0-N' shown in Fig. 6, where '1-N' queues correspond to the 'N' Interrupts that occurred.

During the Interrupt Context Save and Restore, the values written to the registers may not match between the two setups, since there is no synchronization between CPU and model. Thus, Context Save and Restore period will be a blanked-out window for the above scoreboard. No entries will be made to any of the queues in this window. The instructions within the interrupt service routine will behave identically, as the context is cleared before re-enabling the scoreboard.

A compensation for the blanked-out window is required though, as we are not checking whether what is saved is restored. To resolve this, a separate Context switch stack is maintained as shown in Fig. 7, which compares the saved and restored values for equivalence. Along with this, some custom checks are added, to ensure sideband signals correctly reflect the context. For example, at every context entry, an acknowledgement must be sent.

CONTEXT SAVE QUEUE	CONTEXT SAVE POINTER
A1 = 0xc1 …. M32 = 0xa5	1
A1 = 0x3a … M32 = 0x40	2
A1 = 0x20 … M32 = 0x1c	

CONTEXT RESTORE QUEUE	CONTEXT RESTORE POINTER
A1 = 0xc1 …. M32 = 0xa5	1
A1 = 0x3a … M32 = 0x40	2
Waiting for restore…	

Fig. 7 Scoreboard Context Queue Structure Illustration

D. Limitations

While this adaptive scoreboard has simplified many of the complexities in our CPU verification, some performance leniencies have been taken. Some examples of items that cannot be verified accurately by the scoreboard include interrupt response time, precision of HALT, RUN and interrupt entry, quantitative impact of memory stalls, etc. These are compensated by running Benchmarks, use of supplementary assertions and margin-of-error based performance checks in the testbench. In addition, some dedicated tests are also developed to check the same.

Another limitation of the scoreboard is that it is quite intrusive. It looks at a large number of signals to arrive at its decisions, which have to be guaranteed using alternate means. Some of these signals are guaranteed by Formal Verification of sub-units, whereas others are guaranteed by a combination of assertions and design of the scoreboard.

Lastly, having an advanced scoreboard such as ours impacts the simulation time, as it is data heavy and does extensive post processing on every write access to every resource of the CPU. As the CPU scales across generations with more resources, this impact will also rise manifold.

IV. RESULTS

Using this methodology, we saved about 6MM (Man Months) of time in modelling, synchronization, and Benchmarking, as shown in in Fig. 6. We were able to avoid the need for fine-tuning the reference model to be an exact cycle replica of the CPU, ensure that there is a fixed reference which is not subject to excessive variations, which in turn reduced the Benchmark qualification effort. The DUT alone was subjected to all variations such as debug event triggers, power saving modes, program and data memory wait states and errors, which were changing on the fly, thus eliminating the need for continuous synchronization across multi-language boundary.

Effort Saved	Activity	Benefits
3 MM	Modelling	Events not modelled: (Introduced on-the-fly) ECC errors Debug events Memory Stalls (On-the-fly and static) Power down and Wake-up
2 MM	Synchronization and TB development	Drivers developed only for Interrupts Multi-language testbench enhancement effort minimized
1 MM	Benchmark Qualification of model	Benchmarks qualification was only done on Zero wait-state error-free memories

Fig. 8 Effort reduction analysis

Critical bugs found using this solution include:

- Instructions executing during a pipeline stall condition
- Missing pipeline protection for pipeline hazards
- Unintended updates to a resource
- Dual execution of instruction during HALT and RUN
- Instruction intent not met during the presence of correctible errors
- De-railed course of execution due to incorrect context save and restore

The use of this scoreboard has significantly improved debuggability, to the effect that any bugs can be easily root-caused. Every comparison is logged and reported. Value and cause of errors are specifically indicated as shown in Fig. 9.

```
--- SPLIT INTS:0----->comparing reg_A15 in RTL: 0 from PC: 20, @346.000000ns with SC: 0 from PC: 20, @346.000000ns<----
--- SPLIT INTS:0----->comparing reg_DSTS in RTL: 7f90000 from PC: 20, @346.000000ns with SC: 7f90000 from PC: 20, @346.000000ns<---
--- SPLIT INTS:2[0]----->comparing reg_M2 in RTL: 720000 from PC: f8, @946.000000ns with SC: 720000 from PC: f8, @946.000000ns<----
--- SPLIT INTS:2[0]----->comparing reg_ISTS in RTL: e9 from PC: 10c, @946.000000ns with SC: 0 from PC: 10c, @946.000000ns<----
```

Fig. 9 Sample Reports

V. Conclusion

In this paper, we discussed the development and deployment of Adaptive Scoreboard. This solution enabled us to verify our CPU which directly impacts an entire generation of devices. The key advantage of the scoreboard is that it is flexible and responsive to context changes and stalls, making the cycle accuracy of the verification model a nonrequirement. The scoreboard is currently being used to qualify thousands of regressed test cases, where triggers and errors are introduced without much hassle. It has also led to several critical bug findings, which would have been difficult to detect by any other means. Ultimately, we were able to achieve our goal of minimizing the complexity in the reference model while being able to use it for verification. This approach also allows us to commission a massive debug effort to the scoreboard, focusing our efforts only on quality of the DUT. The methodology has been presented in an adaptable manner which can be applied to any CPU in future.

References

[1] Milos Becvar, Greg Tumbush, “Design and Verification of an Image Processing CPU using UVM”, DVCON US, 2013
[2] Chang-Yong Heo, Kyu-Baik Choi, In-Pyo Hong, Yong-Surk Lee, “An implementation of scoreboarding mechanism for ARM-based SMT processor”, International Conference on ASIC, 2003
[3] Abdelfattah Munir, Mina Magdy, Samer Ahmed, Sherouk Nasr, Sameh El-Ashry, Ahmed Shalaby, “Fast Reliable Verification Methodology for RISC-V Without a Reference Model”, International Workshop on Microprocessor Test and Verification (MTV), 2018
[4] Ofer Shacham, Megan Wachs, Alex Solomatnikov, Amin Firoozshahian, Stephen Richardson, Mark Horowitz,“Verification of chip multiprocessor memory systems using a relaxed scoreboard”, IEEE/ACM International Symposium on Microarchitecture, 2008
[5] Abhineet Bhojak, Stephen Hermann, “ UVM Based Methodology for Processor Verification”, DVCON EUROPE, 2015