Random Testcase Generation and Verification of Debug Unit for a RISC-V Processor Core

Sneha Mishra, Lu Hao, Ajay Sharma, Afshan Anjum, Lucia Franco, Sourav Roy, Jeff Scott
NXP Semiconductors
Email: sneha.mishra@nxp.com, lu.hao@nxp.com, ajay.sharma_1@nxp.com, afshan.anjum@nxp.com,
lucia.franco@nxp.com, sourav.roy@nxp.com, jeff.scott@nxp.com

Abstract

Verification of the debug unit of a processor core is different from verification of the normal core pipeline and instruction set. The main difference is the ability to verify the debug module with multitude of core configurations in addition to external stimuli coming from outside the core. The verification challenges primarily involves putting the core in different configuration with varying mix of instructions in core's pipeline along with external stimuli such as interrupts, debug exceptions, bus faults and then bombarding with debug requests randomly during simulation. The second facet of the challenge is to exercise random instruction stream within debug mode session with varying debug control configurations. And the last facet is around correctness checking, which involves synchronizing of the asynchronous events with the functional model, generating correct set of expectations in terms of registers and memory updates. The paper deep dives into the verification approach taken to verify the debug unit of a RISC-V processor core. The solution presented here tackles all the facets gracefully, which allows full-blown random verification including correctness checking at the granularity level of intermediate register and memory updates. It talks about how the RISC-V based random testcase generator Raptor [1] has been augmented to generate scenarios targeting multiple debug sessions within normal running mode, that support altering the state of the program, as compared to other solutions like RISCV-DV [2] [3]. The paper finally discusses the debug functional coverage and bug-chart that were achieved during the verification of the RISC-V core.

Index Terms

Debug, RISC-V, Random Test Generation, Verification

I. Introduction

Verification of microprocessors is a complex task as it involves exercising the entire instruction set along with exceptions. This results in innumerable combinations of instructions and exceptions, which are typically verified using random testcases [4] [5] [6]. Debug mode in a processor is a special kind of exception that halts the core and lets the user debug the internal state of the processor. Verification of the Debug module is different from verifying normal instructions as it involves external stimuli from the user that changes the state of the processor while it is executing the software testcase. The user can halt, step, resume the core. He can also add breakpoints or watchpoints that would trigger various actions (eg. halt or exception) at specific points in the program or on specific events. While in debug mode, the user can also change the state of the program by modifying registers and memory, and executing different instructions from the program buffer. This is rather complex to verify in random verification as currently available testcase generators either can not handle debug events or can not track the software state change owing to the external debug stimuli. This paper discusses how the random testcase generator and verification methodology has been augmented to handle these challenges on a RISC-V processor core.

II. RISC-V Debug Specification

It is critical to have a good debugging support built into the processor to help system bring-up and debug the software executing on the core hardware. The RISC-V Debug specification provides multiple features, some of which are mandatory while others are optional. Debug Module Interface (DMI) is the bus master to the Debug Module. It supports read and write operations to debug registers, which results in various debug operations. The debug specification provides run control feature of core in the hardware platform. It can halt the core immediately out of reset at the very first instruction. It can halt the core when software breakpoint instruction is executed. It can halt the core when a trigger matches the program counter or read/write address. It can do step operations to execute one instruction at a time. It can request the system to reset the core. The RISC-V Debug specification also provides support for the program buffer with whose help a debugger can execute instructions on a halted core. Debugger can indirectly access system memory via halted core by placing load or store instructions in the program buffer. Abstract commands provide read and write access to general purpose registers (GPRs). Additional control status registers or CSRs are accessible by writing programs to the Program Buffer. Unlike some other implementations where the program buffer uses the processor memory space, our paper discusses about the program buffer which is physically located inside the RISC-V Debug module. The verification challenges are similar for both implementations.

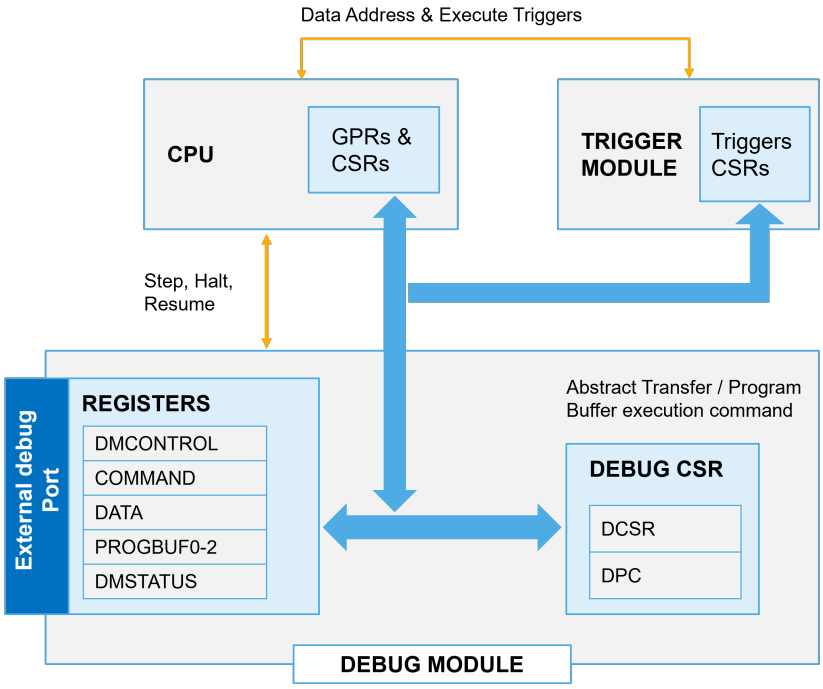

Figure 1. RISC-V Debug: Example Block Diagram

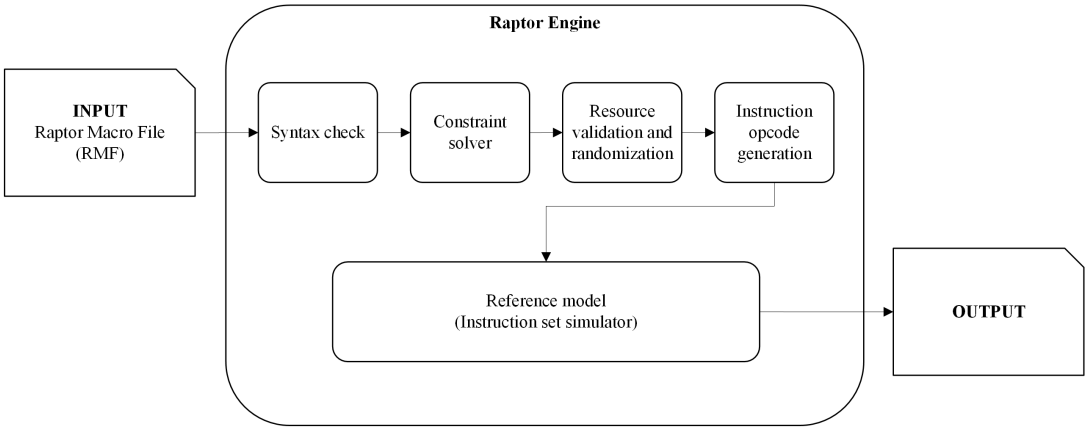

Figure 2. Raptor Engine Overview

III. Raptor – The Random Test Generator

The Raptor tool is a biased-random assembly-level test case generator for microprocessors. It is a stand-alone and pre-simulation tool independent of any specific test bench; Raptor generates test cases for different architectures, but we will limit our discussion to the RISC-V ISA in this paper. This tool can generate from purely random code to very contrived and algorithmic code in which processor state, memory state, and instructions are all fixed; and between random and fixed generation exists biased generation. Biased generation allows the user to specify some or all of the arguments required in the test case or to describe what type of events are desired.

As shown in figure 2, a Raptor Macro File (RMF) is written by the user as the input using Raptor Macro Language (RML). This file will specify the instruction generation directions such as biasing information, resource initialization, and instruction generation flow. RML is a full-featured language that gives users great flexibility and control when writing a test specification. It offers well-known control flow structures (i.e., if, while, foreach, etc.) and supports querying generator and processor state, thus allowing users to dynamically bias or change the test flow during runtime based on current conditions. When Raptor receives an input file, it will first analyze the instructions specified by the user and the constraints requests. Then the Raptor

engine will execute a syntax check followed by the resource validation where all required randomization takes place to obtain the corresponding instruction opcodes.

To simulate the test case instructions as they are generated, Raptor will communicate with an external reference model for the microprocessor to obtain an output; such output is a user-friendly ASCII text file containing the initial machine state (memory, registers, etc.), instructions to execute, and expected results (both intermediate and final). The Raptor output file (called a UVP file) can be used as a stimulus for RTL simulations or to compare results with the RTL execution. Raptor provides several features that will enable the user to manipulate their test case flow, the most used features are the following:

- Mix of 16-bit and 32-bit instructions generation.
- Automatic calculation of valid branch target address.
- Memory region configuration (i.e., data and instruction memory).
- Automatic branch insertion to a bigger instruction area in case there is not enough space for the current instruction flow.
- Interrupt insertion to mimic SoC pins.
- Randomized invalid opcode generation to ensure the RTL handles it correctly.
- Label creation to associate the current program counter or effective address.

IV. Debug Verification with Raptor

Raptor has been augmented to generate scenarios targeting multiple debug sessions within normal running mode for debug unit verification. The following sections describe the details.

A. Program Buffer

Raptor provides users several ways to place instructions in the program buffer. For example:

a) progBuff(”progbuf0”, 0x123452b7); // direct opcode
b) progBuff(”progbuf1”, ‘call # LH(realAddr=0x9000)); // biased-random instruction: load from address 0x9000
c) progBuff(”progbuf2”, instr()); // function call contains biased instructions
d) progBuffs(instrs()); // function call containing multiple biased instructions

For cases c) and d), the function call instr() and instrs() contain one or multiple biased-random instructions like the LH (Load Half-Word) instruction in case b). For case d), Raptor will put all the instructions in instrs() function one by one in each program buffer that’s implemented in the design, it’s up to the user to have the correct number of instructions in instrs() that matches the number of implemented program buffers.

Raptor can generate instructions for the program buffer similar to the way it generates normal instructions for a testcase. However, if the program buffer is a physical buffer in the debug module, as assumed in our work, there are some differences in the behavior of program buffer instructions as compared to normal instructions. Branch instructions from the program buffer are treated as NOPs. Exceptions while executing any program buffer instruction terminates the program buffer execution itself. These are taken care in the functional model that interact with Raptor.

B. Abstract Commands

Further, we extended the progBuff() macro to support abstract commands in Raptor. The user will need to provide the exact register value and Raptor will write it to the “command” register. e.g. progBuff(”command”, 0x00240000). Raptor has been augmented for the following command types:

a) Command that executes the program buffer – see paragraph “program buqffer”
b) Automatically incremented Register index via aarpostincrement field configuration – for each time the index is incremented, Raptor automatically initialize the GPR value.
c) progbufx or datax register read triggers – Raptor implemented pseudo registers “reg_read_progbufx” and “reg_read_datax” to mimic the behavior by writing the same value back to the register.

C. Data and Instruction triggers

Raptor has implemented trigger registers such as “tdata1”, ”tdata2” and “tselect”. They are being accessed via CSR instructions internally and by debugger externally.

V. Verification Methodology

The verification flow as shown in figure 3 is a 2-pass flow using the functional model or the ISS (Instruction Set Simulator). It first creates a random testcase using Raptor and the functional model. This test is simulated on the RTL to generate a output trace and an events file. The testcase is then run on the functional model for the second time but with the events file as a second input. This produces a trace that is then compared with the RTL trace for every instruction and event. This is similar to using co-simulation of the design and model in lock-and-step mode. The co-simulation approach terminates a failing test immediately, whereas using the model with the events file post the RTL simulation provides more control to re-order events

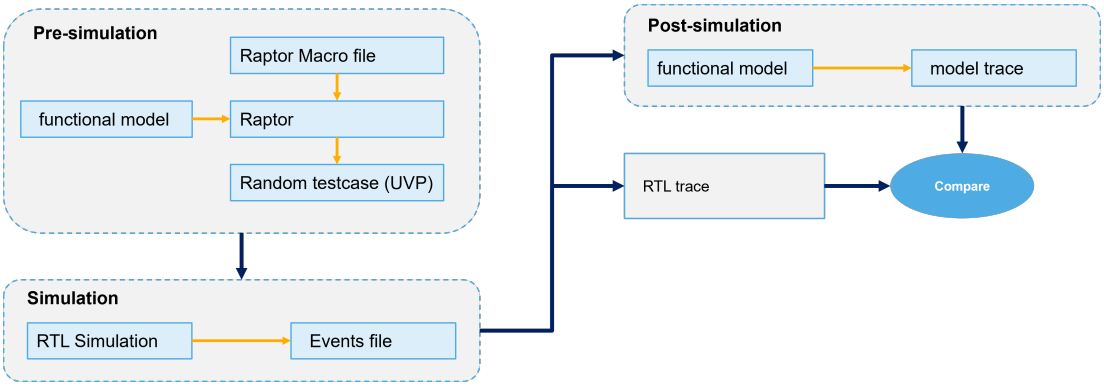

Figure 3. Verification Environment and Methodology

in complex scenarios. The events file lets us insert external stimuli into the functional model based on the RTL simulations events. But the external stimuli have to recoverable in nature, like an interrupt with a recoverable handler code, that returns from the handler code without modifying the state (i.e. registers and memory) of the processor. A simple external debug session consisting of halt, step and resume is also recoverable. But if a register is modified during the debug session, the state of the processor is modified. The simulation would then deviate from the intended test flow and would produce a failure. The following sections describe how we handle this case by incorporating non-recoverable debug events into the test creation flow in Raptor.

A. Debug Verification Environment

The verification environment can be primarily categorized into following phases :

1) Pre-Simulation phase : User writes raptor macro file (RMF) which is a constraint file containing register initializations and weights associated with different types of instructions. Raptor generates testcase either in a binary (ELF format) or executable trace (UVP). The UVP-Parser parses the address-opcode pair and loads them into testbench memory.
2) Simulation phase : Tarmac module hooked within core generates RTL trace and an event file. The event file contains the footprints of total number of completed instructions and external events asserted throughout the simulation as follows,

```
events {
  step 4                  //Number of instructions completed
  async mei=1          //MEI interrupt asserted
  step 8                  //Number of instructions completed
  async mei=0         //MEI interrupt  deasserted
  step 5                 //Number of instructions completed
}
```

3) Post simulation : Event file and UVP are further passed to functional model. We have an event synchronizer utility that synchronizes external async events with the functional model and generates model trace output (which is now inclusive of all async events that were not part of static UVP). Since the model is untimed, asynchronous events are passed to it via un-architected registers to assert the change of program flow and generate correct set of expectations. Model trace and RTL trace are compared at the granularity level of intermediate registers and memory updates, including final result checking as well.

The primary features of RISC-V debug are:

1) RUN Control: Halt, resume, single-step
2) Halt groups, Resume groups and External Triggers
3) Data and instruction triggers
4) Abstract commands: register access and program buffer execution

The following sections deep dive into the modular approach taken to verify the above features.

B. Verification Target – Run control and External Triggers

The main idea is to put core in varying configurations with respect to its pipeline and then randomly bombarding with debug requests, external triggers. This is shown in figure 4. We have developed raptor macros generating combinations of different instructions in core's pipeline which run in conjunction with external stimuli like interrupts, bus faults etc. While this runs

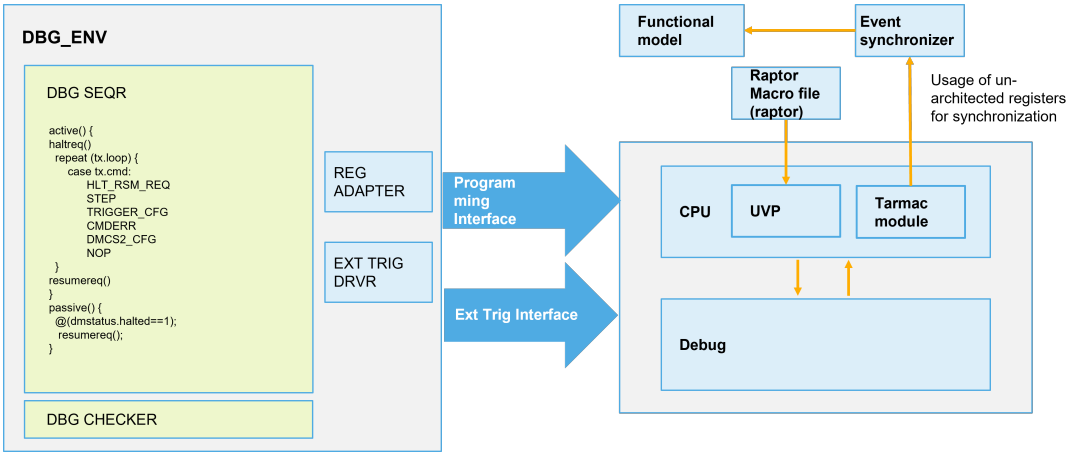

Figure 4. Verification methodology for debug run control

on core, UVM sequence has been developed to exercise the debug interface in parallel. As depicted in the pseudo code, it primarily consists of following tasks:

- active(): This is invoked randomly during simulation. Each command caters to generating specific debug verification scenario such as single stepping, data and instruction trigger configuration (debug and exception actions), random register configurations for debug CSRs like DMCS2 and DCSR. These commands eventually translate to driving debug programming interface or external trigger interface with relevant debug control register configurations
- passive() : This mode is invoked after active() mode finishes. Sequence remains in passive mode and monitors core's debug status. It generates resume request if core enters into debug mode due to internal events like address trigger hits, ebreak etc. Event synchronizer synchronizes all the external events and generates model trace that includes debug activities such as DATA0, ABSTRACTCS register updates due to abstract commands, address and external trigger status updates, DCSR updates etc.

C. Verification Target – Data and Instruction Triggers

Generic macros have been developed, which are called in conjunction with other raptor macros to allow trigger set randomly on different types of instructions and control configurations in terms of action, match, dmode, size etc. (tdata1).

1) Instruction trigger: Address trigger data register (tdata2) is programmed with random offset relative to the current instruction address, so that when core proceeds it may hit the trigger and take the anticipated action. Address control configuration register (tdata1) is programmed randomly to exercise all trigger match scenarios.

```
IAR = current instruction address;
tdata2 = IAR + random_range(0,0xFFF);
tdata1 = random_range(0,0xFFFFF);
repeat(count) {
call any_inst(); //select instruction randomly from instruction list
}
```

2) Data trigger: The idea is to randomly program tdata2 register within a specified address range and then generate load/store instructions within that range. Here the target address is also programmed in a general purpose register (say, gpr10) and then passed as an operand to the instruction.

```
tdata2 = random_range(0xFFFFF000,0xFFFFFFFF);
tdata1 = random_range(0,0xFFFFF);
gpr10 = tdata2;
repeat(count) {
call ld_st(target address = gpr10); //select instruction randomly from loads, stores
call any_inst(); //select instruction randomly from instruction list
}
```

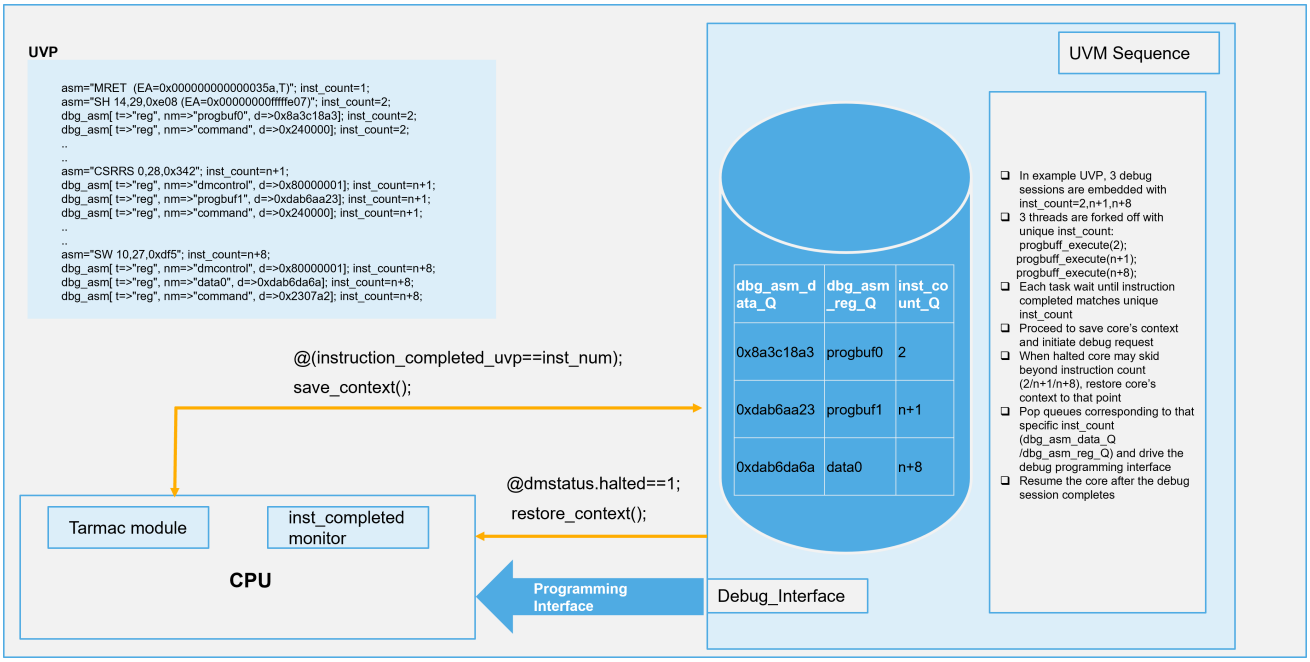

Figure 5. CPU state save-restore during program buffer execution

D. Verification Target – Abstract command and Program Buffer execution

Program buffer execution potentially alter core's context in terms of CSR, GPR contents, privilege level transitions etc. It required enhancements in Raptor to generate random scenarios containing instructions in both normal mode and debug modes as described in section IV. With the augmented Raptor, UVP contains random instructions and debug mode aware instructions embedded into it statically. A separate hardware semantics has been devised in Raptor to differentiate the debug mode instructions. Each instruction in UVP has a unique number associated with it. Parser parses through the UVP and places address opcode pair in testbench memory. It stores the debug session instructions in testbench queues, along with its associated unique number.

An instruction completion monitor tracks the completed instruction during simulation and once it hits the number associated with the debug mode instructions, UVM sequence initiates halt requests through the external debug interface. The entire scheme requires the core to be put in debug mode at precise point, which is always not possible to achieve. Since halting takes multiple cycles and core may skid and likely to deviate from the intended programming region, we have introduced the concept of saving and restoring core's context to achieve debug mode entry at specific points in time dictated by the UVP itself. Core's context (CSRs, GPRs, memory snapshot, branch target buffer) is saved when instruction completed matches with the unique number associated with the debug mode instruction. Once the core enters debug mode, possibly at a later point, its context is restored to that precise point. Debug's DPC and DCSR[PRV] registers are manipulated such that it resumes from the exact same PC and privilege level. The restoration involves writing into the DUT registers via backdoor mechanism and also the testbench memory. This is easy to do since the core is halted at this time and no activity is going on in the DUT. All the debug instructions are then popped from the queues and driven along the external interface to execute random instruction stream via program buffer execution in debug mode. This is depicted in figure 5. Note for superscalar processors, the debug entry point in the UVP may not be a valid state, since multiple instructions complete in a cycle. For such processors, the state is restored to the nearest valid state, and then step operation(s) is used to reach the intended point in the program, if required. In this way, the methodology is scalable from low-end to high-end cores.

Pseudo code depicting the above UVM sequence implementation is shown in figure 6.

E. Debug checkers

Apart from trace comparison with functional model, we have developed debug checkers to do exhaustive checking of signal-level protocol that are not always traceable such as:

- CPU and debug interactions
- Debug state transitions (halt request, resume ack, halton-reset request, and hart reset)
- External notification on the output port for non-debug trigger actions
- Number of steps during single stepping

```
//Task to parse UVP and store debug aware instructions in queues dbg_asm_data_Q/dbg_asm_reg_Q
task progbuff_instr_Q();
  foreach (core._instr_Q[i]) begin
      if (inst.debug_progbuf_Q_data.size() > 0) begin
        dbg_asm_data_Q.push_back(inst.debug_progbuf_Q_data[0].data);
        dbg_asm_reg_Q.push_back(inst.debug_progbuf_Q_data[0].reg_name);
        inst_count_Q.push_back(inst.inst_count);
      end
  end
endtask

//wrapper task to fork off multiple threads of progbuff_execute() with unique inst_count associated with each debug session
 task progbuff_sequence:: progbuff_parse;
 uniqueNum_Q = inst_count_Q.unique();
  foreach (uniqueNum_Q[i]) begin
    progbuff_execute(uniqueNum_Q[i]);
  end
 endtask:progbuff_parse
```

```
//Each thread monitors instruction count completion and drives external debug programming interface
task progbuff_sequence:: progbuff_execute(int inst_num);
fork
  SAVE : begin
      @(dbg_vif.instructions_completed_uvp == inst_count ) begin
        state.save(); //CPU state saved
        haltreq();     //halt request initiated
        ->progbuff_start;
      end
  end
  RESTORE : begin
    @(progbuff_start);
    wait_for_halted_dmstatus();
    state.restore(0);   //CPU state restored
    foreach(inst_count_Q[i]) begin
      if (inst_num == inst_count_Q[i]) begin
        data = dbg_asm_data_Q[i];
        regname = dbg_asm_reg_Q[i];
        case (regname)
            "abstractauto":  regmodel.ABSTRACTAUTO.write(status,opcode);
             ....
        end
      end
```

Figure 6. UVM sequence during program buffer execution

Any change on input ports of the Debug module are captured and pushed back in internal queues. For any transition in debug state, or on output boundary, the relevant queue is popped and mapped back to the source of impact. If no credible source is found, an error is flagged. At the end of the simulation, all internal queues are checked if they are empty, or else flags an error, citing that few transitions remain unexplained without an impact.

VI. Results and Coverage

Applying the random testcase generation and verification techniques described in the previous sections we were able to achieve high coverage of the debug module. Several interesting scenarios were also unearthed that were not thought of during conventional directed verification of the debug module. Examples include processing of interrupts and exceptions during debug step, program buffer execution of various categories of instructions in different privilege modes, etc. The highlights are as follows:

- Multiple corner case bugs and interesting architectural issues around custom instructions reported.
- Debug features crossed with other external stimuli like interrupts, bus faults etc.
- Debug features crossed with decode class exceptions (ecall, mis-aligned faults etc.)
- Debug features crossed with low power mode stimuli
- External debug and internal debug mode cross scenarios
- Program buffer execution and register accesses
- 100% functional coverage of debug module was achieved. Examples of functional coverage are coverage of triggers and instructions executed from the program buffer. Figure 7 shows how the latter improved over time.
- 100% code coverage was also achieved utilizing random verification and Unreachability (UNR) analysis with formal tools.

Figure 8 shows an example chart for the simulation statistics related to the debug module.

VII. Conclusion

Current state-of-the-art random verification methodologies can handle external stimuli using testbench handshake mechanism. But they cannot handle external debug stimuli that change the state of the processor, like abstract command and program buffer execution. This paper, on the other hand, describes a 2-pass flow using the events file from RTL simulation to synchronize the functional model with external stimuli. Raptor has been augmented to generate the external debug stimuli during test creation phase itself. This allows any state-altering random sequence to run in the debug mode, covering the full range of random scenarios and complete trace comparison. A special technique of state save-and restore is used to force the processor to halt at the intended debug point. Applying this methodology we are able to hit 100% coverage and unearth very corner case bugs around debug mode functionality on an industrial quality RISC-V processor.

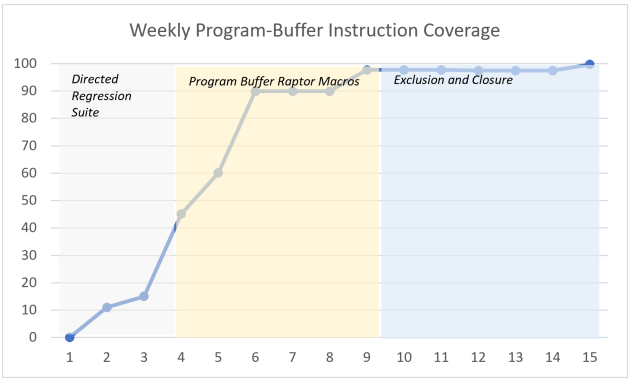

Figure 7. Program Buffer coverage

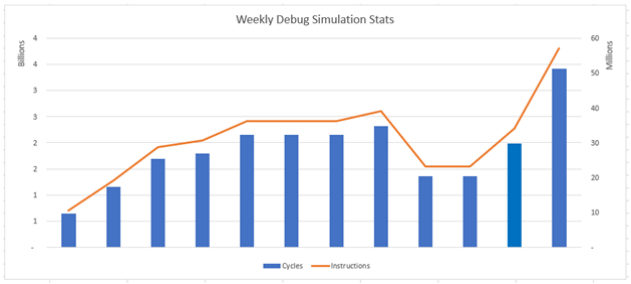

Figure 8. Debug Simulation statistics

ACKNOWLEDGMENT

The authors would like to thank several of their colleagues at NXP who contributed to this work. Special thanks to Nikhil Jain, Ravinder Dasila, Navaneet Kumar, Mayuri Gadewar and Mark Johnstone.

REFERENCES

[1] D. M. Farkash, "Cache Coherency Verification," https://www.coursehero.com/file/33451611/13-2pdf, 2018.
[2] "RISCV-DV Documentation," https://htmlpreview.github.io/?https://github.com/google/riscv-dv/blob/master/docs/build/singlehtml/index.html#L110.
[3] T. Liu, R. Ho, and U. Jonnalagadda, "Open Source RISC-V Processor Verification Platform," in *RISC-V Summit*, 2019.
[4] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey, "Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor—the DEC Alpha 21264 microprocessor," in *Proceedings of the 35th annual Design Automation Conference*, 1998.
[5] K.-D. Schubert *et al.*, "Functional verification of the IBM POWER7 microprocessor and POWER7 multiprocessor systems," *IBM Journal of Research and Development*, vol. 55, May-June 2011.
[6] J. M. Ludden *et al.*, "Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems," *IBM Journal of Research and Development*, vol. 46, January 2002.