2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Verification Methodology for Debug Unit of a Superscalar RISC-V Processor

Ajay Sharma, Afshan Anjum, Sourav Roy
NXP Semiconductors
Email: ajay.sharma_1@nxp.com, afshan.anjum@nxp.com, sourav.roy@nxp.com

ABSTRACT

When it comes to verifying a Superscalar RISC-V processor's Debug unit, the process involves not only verifying the Debug unit itself but also verifying its interactions with variety of processor configurations in addition to receiving external inputs from outside. Verification poses several challenges, including configuring the processor with different settings, mixing instructions in the pipeline, and introducing external stimuli like interrupts, debug exceptions, and bus faults. During simulation, the processor is also subjected to random debug requests. Another challenge involves testing a range of instructions in debug mode with various debug control configurations. Correctness checking is crucial, requiring synchronization of asynchronous events with the Instruction Set Simulator and generating expected outcomes for register and memory updates. The paper acknowledges the profoundness of the methodology published at DVCON US 2023 [1] in addressing these challenges in verifying a Debug unit of a Scalar RISC-V processor. However, when it comes to verifying a Debug Unit of a Superscalar RISC-V processor, it has limitations in halting the processor precisely at a desired instruction with the "State Save and Restore" technique described in [1]. The paper meticulously examines these limitations and presents "Software Breakpoint" technique in which the processor architecture guarantees precise halting at the desired instruction as part of the breakpoint implementation, irrespective of whether it is a Scalar or Superscalar processor. The paper discusses how this technique is versatile and works well for processors with Instruction cache. This technique delivers an average simulation speedup of 11% as compared to the "State Save and Restore" technique. The paper finally implements this technique for the verification of Superscalar RISC-V processor's Debug Unit demonstrating 100 percent functional and code coverage.

INDEX TERMS

Debug, RISC-V, Superscalar Processor, Verification

I. INTRODUCTION

Verification of superscalar processors is an intricate task as it concerns execution of more than one instruction simultaneously along with exceptions and external stimuli. This results in multitude of combinations of instructions, exceptions, and external stimuli which are commonly demonstrated using random testcases [2]. Debug mode in a processor is a special processor mode that halts the core and allows the user to debug the internal state of the processor, which can be different at each instruction. This makes precise halting paramount. Precise halting of a superscalar processor is quite challenging. Moreover, any scalar or superscalar processor may come with an Instruction Cache. Any methodology that aims to attain precise halting should also work on processors with Instruction Cache (while maintaining the coherency with main instruction memory). In random verification testcases, an instruction can be executed any number of times. Selective execution of debug commands on few of these instruction instances or all instances is a complex task to achieve. This paper discusses how the proposed methodology overcomes these challenges ensuring the quality of verification and how it is augmented to be swiftly applicable to all configurations of both scalar and superscalar RISC-V processors.

II. RISC-V DEBUG SPECIFICATION

Effective system bring-up and software debugging on a processor hardware requires robust debugging support integrated into the processor. RISC-V Debug specification offers various features such as Run control, Abstract Commands and Program Buffer. Various debug operations are performed by writing and reading the Debug Registers via Debug Module Interface (DMI) which is a bus master to the Debug module.

Run control feature allows debugger to halt or resume the processor. The processor can be halted at the very first instruction. It can be halted when a software breakpoint instruction is encountered. It can be halted when a trigger matches the Program Counter (PC) or a read/write address. In addition to this, debug step operation resumes the halted processor allowing it to execute one instruction before entering the halted mode again.

Provision of abstract command enables a debugger to access Debug Control and Status Registers (DCSRs), General Purpose Registers (GPRs) and Trigger CSRs. The debugger performs abstract command via writing Data and Command registers.

Program buffer support allows a debugger to execute any instruction on a halted processor. This also helps a debugger to indirectly access system memory by executing load or store instruction from the program buffer. Through this, a debugger can also access Processor CSRS, GPRS, Debug CSRS and Trigger CSRs. Though it is implementation dependent to either have program buffer using memory address space or having it physically located inside the RISC-V Debug module. Our paper, like [1] discusses the proposed methodology for implementations that incorporates program buffer physically inside the RISC-V Debug module. Further details on the RISC-V debug specification can be found in [3].

III. SUPERSCALAR PROCESSOR

A superscalar processor can execute more than one instruction per clock cycle by dispatching multiple instructions to appropriate functional units within a processor [4]. This parallelism is achieved through deciding what instruction can run in parallel based on data dependencies. Fig. 1 shows Dual-Issue Superscalar processor simple pipeline which can fetch, decode, and execute two instruction per clock cycle.

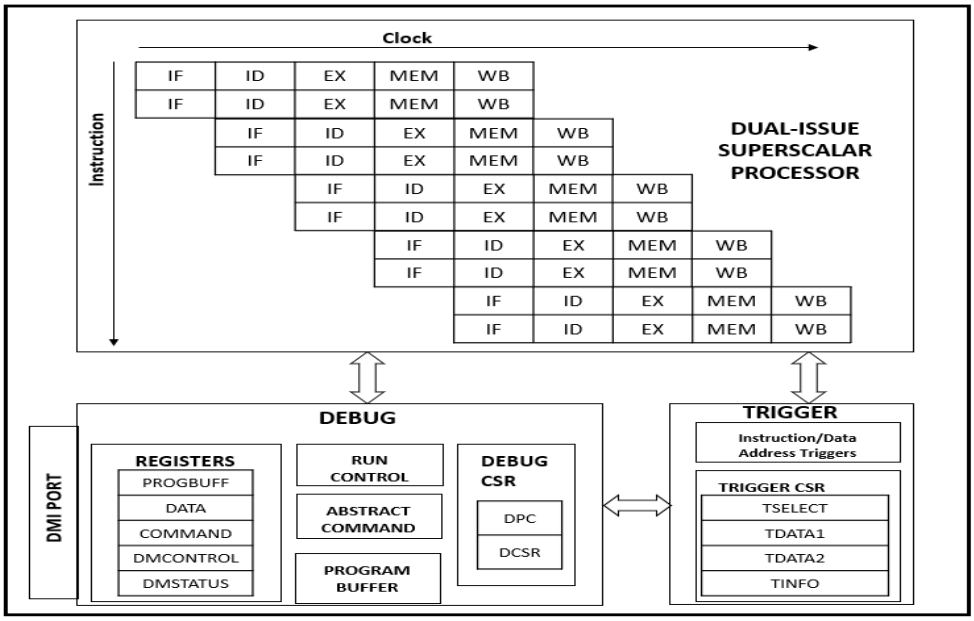

Figure 1. Debug unit of a Dual-Issue Superscalar RISC-V Processor Block Diagram

IV. RELATED WORK

The methodology published at DVCON US 2023 [1] is profoundly successful in verifying Debug Unit of a Scalar Processor. However, when it comes to verifying Debug Unit of a Superscalar processor, it has limitations. The methodology has verification challenges in precisely halting the processor at a desired instruction completion count prescribed by the Random Instruction Generator Tool. Due to this, many a times, the debug session gets skid off by few instructions which results in deviation of the simulation from the desired test. To manage this limitation, the methodology uses "State Save and Restore" technique which saves the processor state (CSRs, GPRs, memory snapshot, branch target buffer) at the desired instruction completion count and restores it just after entering the skidded debug session. Since, Superscalar processor unlike Scalar processor can execute multiple instructions per cycle, therefore, the processor state at the desired instruction completion count may not exist. For instance, let us take an example of Dual-Issue Superscalar processor which can execute two instructions per cycle. Let us suppose Random Instruction Generator Tool prescribes to halt the processor at an instruction completion count=100 and the processor executes 100th and 101st instruction in a single cycle. Therefore, no valid processor state exists at an instruction completion count=100. In other words, there is a valid processor state at an instruction completion count=99 and 101 but not at 100. So, "State Save and Restore" technique is unable to halt the Superscalar processor precisely at an instruction completion count=100. Nevertheless, the methodology in [1] has proposed a solution to handle this problem by halting the processor to the nearest state and then stepping (debug step operation) into the desired state. However, this solution again has verification challenges in precisely halting at the nearest state. Here also, processor halting can get skid off by few instructions. Therefore, the methodology must save the processor state at the nearest state and restore the state when the processor gets halted after skidding off few instructions from the nearest state; thereon debug step operation must be used to take the processor to the desired state. This solution limits external stimuli randomness between the nearest state and the desired state because the processor forward progresses in a step fashion through multiple intermediate debug sessions to reach the desired state. Saving and restoring the state of the processor, requires probing of many internal state level signals. This solution is highly pipeline dependent for tracking the instruction completion count, requires incremental changes even with the changes in the pipeline implementation as well as addition of new features. As we move from Scalar to Superscalar processor, efforts of managing this limitation with this methodology increases significantly.

V. METHODOLOGY

A. Software Breakpoint Technique

The proposed methodology overcomes the above-mentioned limitations by precisely halting the processor at a desired instruction address through "Software Breakpoint" technique. In this technique the processor architecture guarantees precise halting at the desired instruction as part of the breakpoint implementation, irrespective of whether it is a scalar or super-scalar processor. RISC-V Unprivileged ISA [5] provides Breakpoint instructions. EBREAK and C.EBREAK are 32-bit and 16-bit RISC-V Breakpoint instructions respectively, which can be used to halt the running processor precisely. Fig. 2 shows verification testbench, where we have random instruction generator tool, which also generates multiple debug sessions within normal processor running mode. A Debug session consists of debug commands which can execute any instruction on a halted processor, set hardware breakpoints, access GPRs, etc. Fig. 3 shows the sequence flow chart. The UVM based sequence parses the output file (execution traces) of random instruction generator tool. At 0 time, through backdoor mechanism, the sequence loads the instructions in the instruction memory of the processor. It then figures out from the tool's output file all the instruction addresses of those instructions which are to be preceded by debug session. It replaces each of these instructions with either EBREAK if the size of the replaced instruction is 32-bit or with C.EBREAK if the size of the replaced instruction is 16-bit. It also enables the processor halting feature due to software breakpoints by initializing dcsr.ebreakm, dcsr.ebreaks and dcsr.ebreaku fields to one through backdoor mechanism. All this happened at 0 time. Then the reset of the processor is lifted. The sequence now waits for the processor to get halted due to a software breakpoint. Once the processor halts, the sequence replaces the EBREAK/C.EBREAK instruction of the current debug session with the actual instruction through backdoor mechanism. Then the sequence executes the debug session as prescribed by the random instruction generator tool's output file. Here execution of debug session means decoding the debug

session and invoking the desired debug commands by configuring the Debug registers through register interface. Then the sequence executes the Debug Resume Request command which resumes the processor to execute its instructions in the running mode. The sequence now checks if all the debug sessions prescribed by the tool have now got executed. If the debug sessions are pending, then sequence again waits for the processor to halt due to a software breakpoint and then repeats the above-mentioned tasks. If all debug sessions are completed, then sequence before getting ended waits for the processor to complete execution of all remaining instructions. Post simulation, both instruction and debug traces from the RTL are compared with the expected traces generated from the Golden reference instruction set simulator. Since, "Software Breakpoint" technique makes use of RISC-V Breakpoint instructions provided by RISC-V Unprivileged ISA [5], therefore this methodology can work with Debug unit of any RISC-V Superscalar processor.

B. Debug Unit of a RISC-V Processor with Instruction Cache

The proposed methodology of utilizing "Software Breakpoint" technique via a UVM based sequence, alters the state of Main Instruction Memory of processor once during the reset phase and then in the desired Debug session. When it comes to verifying a processor with Instruction Cache (I-Cache), there can be coherency issues (between I-Cache memory and main instruction memory) due to memory variance. The proposed methodology strategically eliminates such coherency issues by using "UPDATE" or "INVALIDATE" mechanism on the I-Cache lines containing breakpoint instructions.

- A Cold Cache based simulation always starts with a MISS for every new address. A desired debug session at an instruction address, say "X" occurs due to presence of software breakpoint instruction at this address in the main instruction memory (introduced by the UVM sequence during reset phase). The line with address "X" if cacheable, is allocated to I-Cache memory, which has software breakpoint instruction. In the desired debug session, once the processor halts, sequence replaces the software breakpoint instruction of the current debug session (address "X") with the actual instruction through backdoor mechanism. The debugger utilizes program buffer execution to either "INVALIDATE" or "UPDATE" the allocated SET for instruction address "X". The proposed methodology uses invalidation for ease of implementation. There is a possibility that I-Cache is in the process of fetching the line to be invalidated from main Instruction Memory or may have it in a buffer and is not yet allocated to I-Cache memory. Hence, it is not advised to use a backdoor mechanism to invalidate the cache as it will require some signal tapping or glue logic to check for status of cache line. Rather, to ensure proper invalidation, program buffer execution is employed to execute a fence instruction followed by appropriate instruction for invalidation. This ensures a clean invalidation operation. Once the sequence executes a Debug Resume Request and the processor begins running, the instruction address "X" is again fetched from the Main instruction memory and not from I-Cache memory. The processor executes the actual instruction for instruction address X. The Line with address "X" if cacheable, is allocated to I-Cache Memory, with actual instruction, effectively maintaining the coherency between Main Instruction Memory and I-Cache Memory.
- In a Warm-Cache based simulation, the I-Cache memory is first invalidated and then preloaded with instructions from the main-memory during the reset phase via a backdoor mechanism. The UVM based sequence updates the main instruction memory with software breakpoint instruction at desired instruction addresses for debug session, say "X, Y and Z" after the warm-up of I-Cache memory completes. To Maintain coherency between the two memories, sequence invalidates the cache lines with addresses "X, Y and Z" in the I-Cache memory using a backdoor mechanism during the reset phase. The instruction address "X" becomes a MISS and is fetched from the Main Memory. Processor enters debug session due to the presence of software breakpoint instruction at "X" in Main Memory. The line with instruction address "X" if cacheable, is allocated to I-Cache memory, which has software breakpoint instruction. Once the processor halts at "X" the sequence replaces Software Breakpoint instruction of the current debug session with the actual instruction through backdoor mechanism. The debugger utilizes program buffer execution to "INVALIDATE" the associated SET for instruction address X. Once sequence completes the debug session and executes the Debug Resume Request, the processor starts running. Instruction address "X" is fetched from the Main instruction memory and not from I-Cache memory. The processor executes the actual instruction for instruction address X. The Line

with address X, if cacheable, is allocated to Instruction Cache Memory, with actual instruction, effectively maintaining the coherency between Main Instruction Memory and I-Cache Memory.
The sequence keeps waiting for a debug session at address "Y" or "Z" and repeats the same procedure when the debug session occurs at these addresses.

C. Debug on Repeated Instructions

The proposed methodology caters to have debug session on repeated instructions whose execution from the same instruction address occur multiple times due to control flow mechanisms like loop and function calls.
The Random Instruction Generator Tool which generates execution traces assumes halt on all the instances of repeated instructions or none. For all the halted instances of the repeated instructions, the tool generates random debug sessions. Some of these debug sessions may be empty which do not execute any debug commands such as executing any instruction on a halted processor, setting hardware breakpoints, accessing GPRs, etc. When the intention is to execute debug commands on some specific instances of the repeated instruction, the debug session of all other instances will just have empty debug session.
As shown in Fig. 3, the sequence parses the output file (execution traces) generated by the Random Generator Tool. The sequence figures out what all instructions getting repeated are having debug session. After loading memory, GPRs and CSRs, the sequence replaces the repeated instruction with the Breakpoint instruction before lifting off the reset. When the processor halted at the repeated instruction address due to a Breakpoint instruction, the sequence replaces the Breakpoint instruction with the actual repeated instruction. Then the sequence executes debug session of that instance of the repeated instruction. This debug session can have random debug commands or may be empty depending upon the random debug session prescription of the Random Generator Tool. After this, sequence performs the Debug step operation which temporarily resumes the processor to execute one instruction. The sequence again replaces the last instruction executed by step operation with the Breakpoint instruction to ensure that processor must get halted at the next instance of the repeated instruction. After this, the sequence performs the Debug Resume operation resuming the processor to start executing instruction from the instruction address pointed by Debug Program Counter (DPC). The sequence now checks if all the debug sessions prescribed by the tool have now got executed. If the debug sessions are pending, then sequence again waits for the processor to halt due to a software breakpoint and then repeats the above-mentioned tasks.

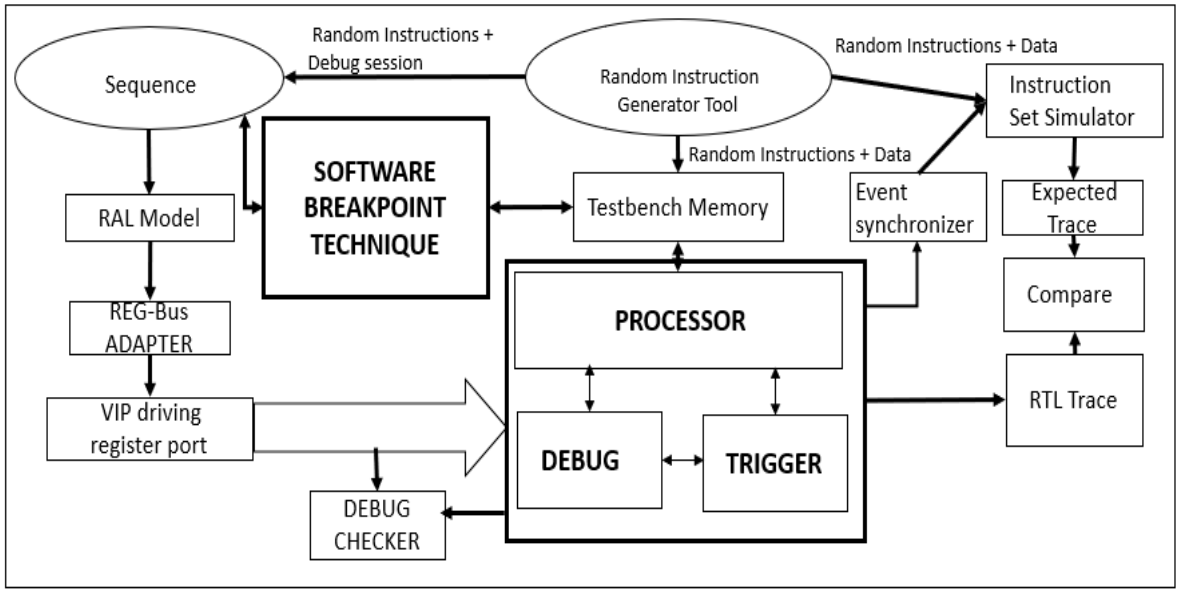

Figure 2. Verification Testbench

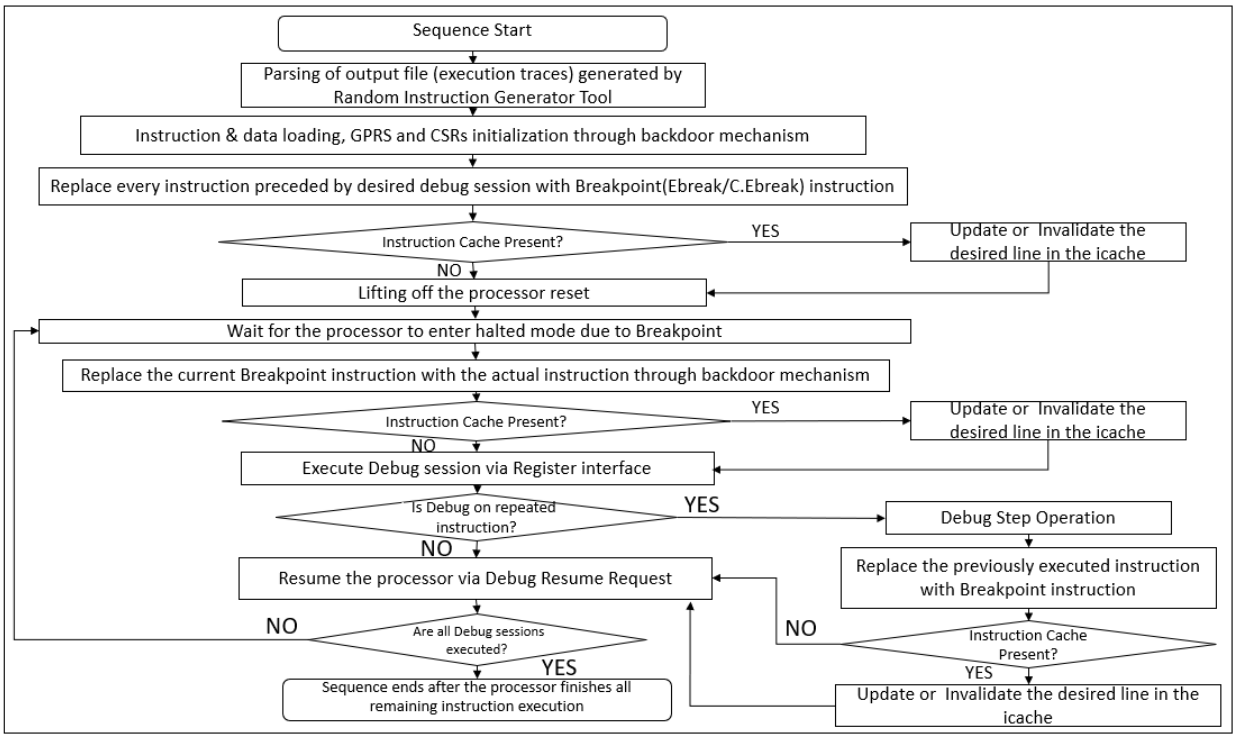

Figure 3. Sequence Flow Chart

```
// Replacing every instruction preceded by desired debug session with Breakpoint(EBREAK/C.EBREAK) instruction
foreach (instNum_ebrk_queue[i])
begin
  if(instSize_ebrk_queue[i] == 32) // 32 bit instr
  begin
    loadWord(instAddr_ebrk_queue[i],32'h00100073,instSize_ebrk_queue[i]); // Replacement with EBREAK(32'h00100073)
    instr_data = 'h00100073;
    byte_mask = 'hF;
    $display("mem_init_debug: addr=%x, random data=%x, byte_mask=%b @time %t",instAddr_ebrk_queue[i],instr_data,byte_mask,$time);
  end
  else // 16 bit instr
  begin
    loadWord(instAddr_ebrk_queue[i],32'h00009002,instSize_ebrk_queue[i]); // Replacement with C.EBREAK(32'h00009002)
    instr_data = 'h00009002;
    byte_mask = 'h3;
    $display("mem_init_debug: addr=%x, random data=%x, byte_mask=%b @time %t",instAddr_ebrk_queue[i],instr_data,byte_mask,$time);
  end
end
```

Figure 4. Pseudo code snippet- Actual Instruction replacement with Breakpoint Instruction during reset

```
wait_for_halted_brkp(); // Wait for processor to enter halted mode due to Breakpoint

// Replace the current Breakpoint Instruction with the actual instruction
foreach (instNum_ebrk_queue[i])
begin
  if((inst_num + 1) == instNum_ebrk_queue[i]) // (inst number + 1) is ebreak instr number which is supposed to be replaced with the actual instruction
  begin //  Actual instr restore via backdoor
      loadWord(instAddr_ebrk_queue[i],instData_ebrk_queue[i],instSize_ebrk_queue[i]);
      instr_data = instData_ebrk_queue[i];
      if(instSize_ebrk_queue[i] == 32)
        byte_mask = 'hF;
      else
        byte_mask = 'h3;

      $display("mem_init_debug: addr=%x, random data=%x, byte_mask=%b @time %t",instAddr_ebrk_queue[i],instr_data,byte_mask,$time);
  end
end
```

Figure 5. Pseudo code snippet- Breakpoint Instruction replacement with Actual Instruction during debug session

VI. SCOPE

The "Software Breakpoint" technique makes use of RISC-V Breakpoint instructions provided by RISC-V Unprivileged ISA [5], therefore this methodology can work with Debug unit of any Superscalar RISC-V processor.

VII. RESULTS

The following are the main results-

- Uncovered architectural issues around custom extension of Debug.
- Verification closure of Debug Module for Superscalar RISC-V processor (with and without I-Cache) by writing new test suite as well as utilizing the existing testcases for verification targets covered in the methodology published at DVCON US 2023 [1].
- Achieved 100% functional coverage of Debug Module. This includes features like:
  - RUN Control: Halt, resume, Single-Step
  - Data and Instruction triggers, External Triggers
  - Abstract commands: program buffer execution and register access
- Achieved 100% Code Coverage through Directed and Random verification utilizing this methodology and Unreachability (UNR) analysis via formal tools.
- "State Save and Restore" technique in [1] makes use of saving and restoring the CPU state which can be time consuming and resource intensive. Moreover, skidding debug session problem in this technique consumes additional simulation cycles. So, this technique leads to overall reduced simulation speed. Fig. 6 shows average simulation speedup of 11% with the "Software Breakpoint" technique, as compared to the "State Save and Restore" technique. The data shown here has been captured by running ten tests with both techniques.
- Fig. 7 shows bug chart for Debug Unit of the Superscalar Processor. Seventeen interesting bugs were discovered. All got resolved except one which was too corner case and had simple workaround in software.

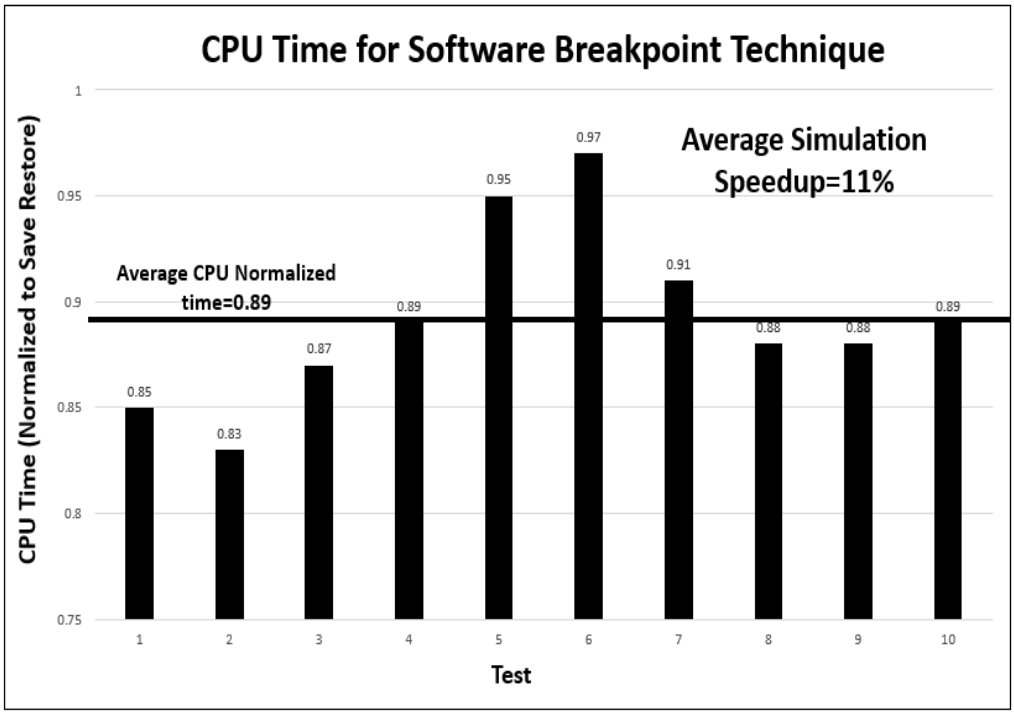

Figure 6. Simulation Speedup

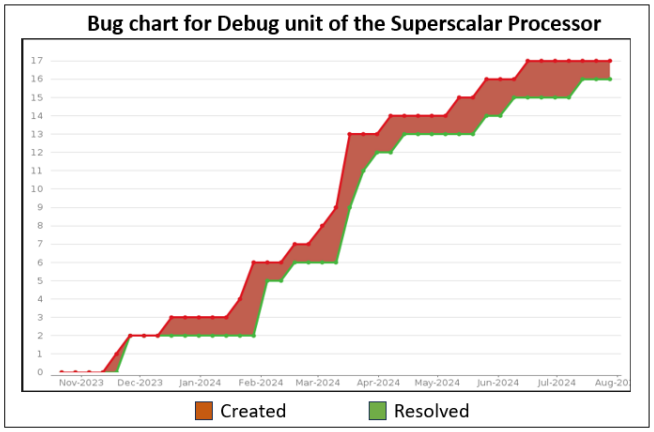

Figure 7. Bug Chart

VIII. POTENTIAL LIMITATION

The proposed methodology has limitations when applied to Self-Modifying program, which can modify its own code during execution. The methodology during processor reset inserts Breakpoint instructions in the instruction memory for executing debug session at desired instruction addresses. When the processor start executing the Self-Modifying program, then it can replace Breakpoint instructions with other instructions. Due to this, debug session at desired instruction addresses can get missed which may result in deviation of the simulation from the desired test. This limitation is not substantial as Self-Modifying programs are rarely used in software development.

IX. CONCLUSION

The methodology published at DVCon US 2023 [1] has limitations with Debug unit of a Superscalar RISC-V Processor due to its "State Save and Restore" technique; the methodology described in this paper, an extension of the former [1], overcomes these limitations by using the "Software Breakpoint" technique. This technique makes use of RISC-V Breakpoint instructions provided by RISC-V Unprivileged ISA [5], therefore this methodology can work with Debug unit of any RISC-V Superscalar processor. This methodology akin to approach in [1], supports debug session changing the state of the processor, offers a clear advantage over external state-of-the-art solution provided by RISCV-DV [6], which lacks this feature. This extended methodology not only supports Debug Unit of a RISC-V Superscalar processor but also Debug Unit of a Scalar processor. Above all, it also supports processors with Instruction cache. "Software Breakpoint" technique in this methodology delivers an average simulation speedup of 11% as compared to the "State Save and Restore" technique in [1]. 100 percent functional and code coverage has been achieved along with discovering seventeen interesting bugs by implementing this methodology on Debug unit of a Superscalar RISC-V processor.

REFERENCES

[1] S. Mishra et al., "Random Testcase Generation and Verification of Debug Unit for a RISC-V Processor Core," in *DVCon U.S. 2023*, https://dvcon-proceedings.org/wp-content/uploads/1110-Random-testcase-generation-and-Verification-of-Debug-Unit-for-a-RISCV-Processor-Core.pdf.
[2] S. Taylor et al., "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.*
[3] "RISC-V Debug Specification," https://github.com/riscv/riscv-debug-spec/releases.
[4] J.Smith and G.Sohi, "The Microarchitecture of Superscalar Processors," *in Proceedings Of the IEEE, VOL. 83, NO. 12, DECEMBER 1995.*
[5] "RISC-V Volume1, Unprivileged Specification ," https://riscv.org/technical/specifications.
[6] "RISCV-DV Documentation,"https://htmlpreview.github.io/?https://github.com/google/riscv-dv/blob/master/docs/build/singlehtml/index.html#L110.