Solving the randomization challenge in CPU verification

Karthik Rajakumar, Pooja Madhusoodhanan
Texas Instruments

1. Motivation & Problem Statement (1/2)

- To accommodate the increasing demands of real-time applications, complex CPUs are being designed, which brings along the challenge of guaranteeing exhaustive verification.
- Case in study is the verification of a CPU architecture with
  - **Instruction Level Parallelism (ILP)**
  - **Very Long Instruction Word (VLIW)**
- To ensure that the design meets the specification, it is imperative to verify all operand combinations as well as sequencing of instructions.
- To achieve this vast stimulus state space coverage in a **dynamic verification environment**, **code randomization techniques** have to be adopted.
- Stimulus must also stress critical logic such as **program fetch and execution control** units. This is achieved mainly using Discontinuity Instructions **(Calls and Branches)** and **Interrupt** / Debug Events.
- When such stimulus is introduced, code execution flows to an **unpredictable location**, where there may not be any program code.
- Achieving **deterministic random code execution** in the presence of **Random Discontinuity instructions** is a big challenge **!!!**

2. Motivation & Problem Statement (2/2)
CPU Random Verification Environment

The **Random Instruction Generator** (RIG) creates test cases with random sequences of instruction packets picked from the Instruction Set Architecture (ISA)

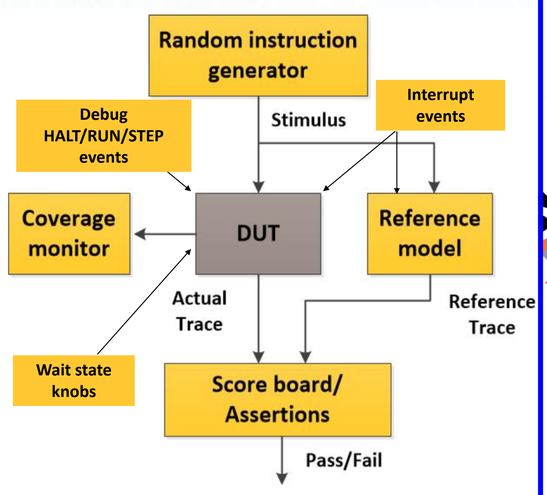
The test case runs on both the **DUT** and a Transaction Level **Reference model** (TLM)

Each test is run multiple times and the register values are randomly initialized at the start of every code run (to get maximum coverage from less number of tests)

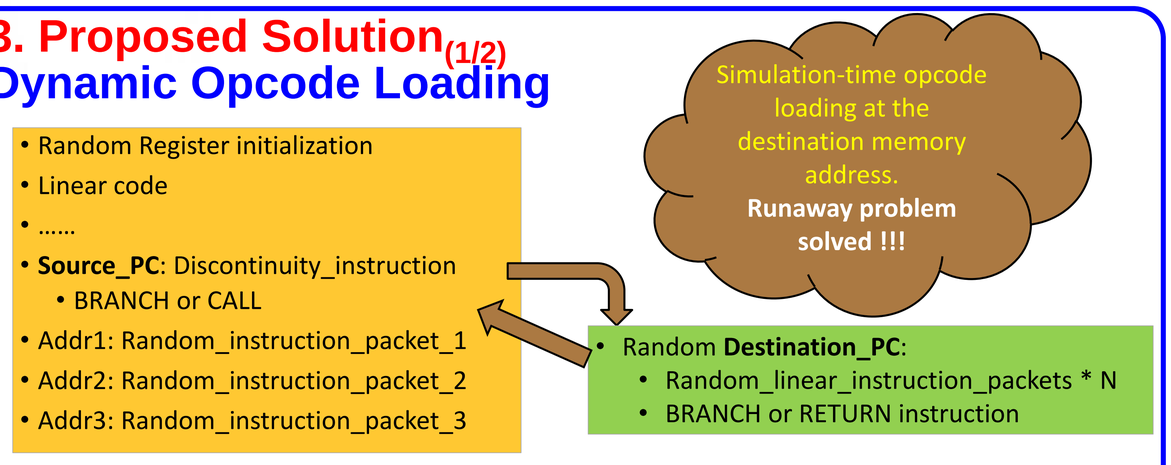
For Discontinuity instructions encountered in the test, the resolved destination Program Counter (PC) depends on a combination of register and immediate values

As the destination PC value is not deterministic at code compile time, it is not possible to initialize the destination memory location with meaningful code.

This causes an untamed random code runaway during the simulation…

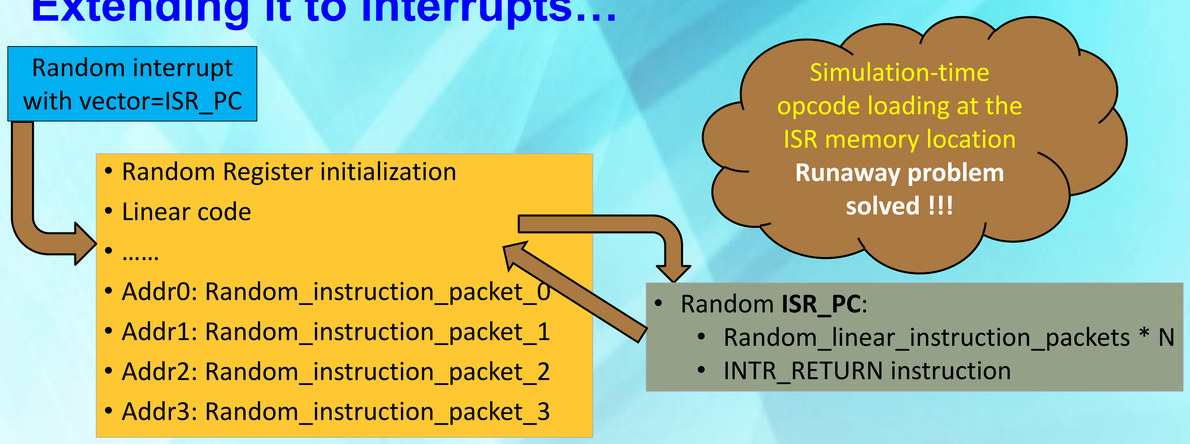


3. Proposed Solution (1/2)
Dynamic Opcode Loading



- On detecting each discontinuity, the testbench logic **on-the-fly** initializes the destination memory location with random code.
  - This is done identically for the DUT and Model program memories.
  - Because of the wait state differences and debug events, the dynamic loading is also done at a **transaction level** per discontinuity.
- The initialized code comprises **of N random linear instruction packets of random packet sizes**, followed by an instruction to bring back the execution PC to the main code, for deterministic code execution.
- The type of the “bring back” instruction depends on the type of the forward discontinuity instruction
  - **“CALL #Destination_PC”** gets back with a **“RETURN”** (uses Addr1 saved on the stack as the return address)
  - **“BRANCH #Destination_PC”** gets back with a **“BRANCH #Addr1”**

4. Proposed Solution (2/2)
Extending it to interrupts…



- The method of handling random Destination_PC can be extended to interrupts.
- On detecting an interrupt discontinuity, the testbench logic **on-the-fly** initializes the **Interrupt Service Routine** (ISR) memory location with random code.
  - This is done identically for the DUT and Model program memories (adopting a transaction level handling approach).
- The initialized code comprises of N random linear instruction packets followed by the INTR_RETURN instruction to bring back the execution PC to the main code for deterministic code execution.
- **INTR_RETURN** uses Addr1 saved on the stack as the return address

5. Evidence and Results

Activity	Traditional Approach*	RIG Approach
Test Case Development Time	1-2 months	< 1 week
Critical Bugs Found/Verification Quality	Low	High
Coverage closure effort	Very High (may not cover all functional cover points)	Low

Deterministic code execution in a totally unconstrained random environment

- ✓ **Support for completely random Destination_PC for Call/Branch instructions and random ISR_PC for Interrupts**
  - ✓ The entire 2^N code address space can be explored without any constraints (N = address bus width)
- ✓ **Code memory context separation for the dynamically loaded code**
  - ✓ Takes care of the cases where the Destination_PC/ISR_PC could coincide with and corrupt the main code
- ✓ **Data memory context separation for the dynamically loaded code**
  - ✓ To ensure the integrity of the main code in the presence/absence of interrupts
- ✓ **Completely random asynchronous interrupt generation support**
  - ✓ Provides interrupt coverage across different instruction boundaries (including discontinuity instructions)
- ✓ **Facilitates verification of the DUT with randomized code/data memory wait states and random Debug Events**

6. Conclusion

- Methodology has been deployed in the dynamic random verification of a next-gen CPU
- Has significantly improved time and quality of CPU verification
- Takeaways of this solution can be applied to other random verification areas to obtain unconstrained state space coverage.

Acknowledgements

- **Vinod Paparaju, Sai Langadi, Bhavya Dasari and Devi A** of Texas Instruments for their support in this work