# AUTOLNKGEN: Automated Random Linker File Generation Framework for Heterogenous SoC Verification & Validation


**Yogeshwaran Shanmugam, Mohan Udayakumar, Suraj Salian, Rakesh YC, Aswin B**

**Texas Instruments PVT LTD**

Texas Instruments

# Motivation / Problem Statement

- System On Chip (SOC) designs have become progressively more complex with heterogenous combinations of CPUs in a single SoC to meet today's catalogue market needs
- Typical processor based SoCs involving CPUs have relied heavily on C-based test cases executed using set of manually written linker command files with target executable memories.
- However, this approach has several limitations:
  - **Limited Coverage**:
    - Test cases target specific scenarios related to modules/system use cases relying on default linker command file for each core to execute/access data in a fixed set of memory regions
    - Execution of same test across different memory types like RAMs, ROMs, Flash, External Memories can introduce coverage variation across latency, throughput, arbitration scenarios
  - **Risk of Silicon Bugs**:
    This leaves plausibility of potential execution gaps or design bugs which could be uncovered if executing the same test from different types of memories or address boundaries with different memory configs.
  - **Need for Comprehensive Verification**:
    To ensure high-quality silicon and minimize post-silicon debugging, it is critical to validate software execution across all regions/memories defined in the CPU memory map.
- Hence it is often necessary to steer the DV environment to generate hard-to-hit functionality scenarios. However the test plan and test cases targets on specific features / scenarios but the scenarios regressing across different memories (ROM / RAM / Flash) to execute are not exhaustively checked
- In this paper, the approach of developing memory context agnostic testcases in SoC Verification and Validation and deploying an automation to generate linker files used across cores with an constrained random approach to catch such corner case silicon issues with exhaustive execution coverage, is deployed

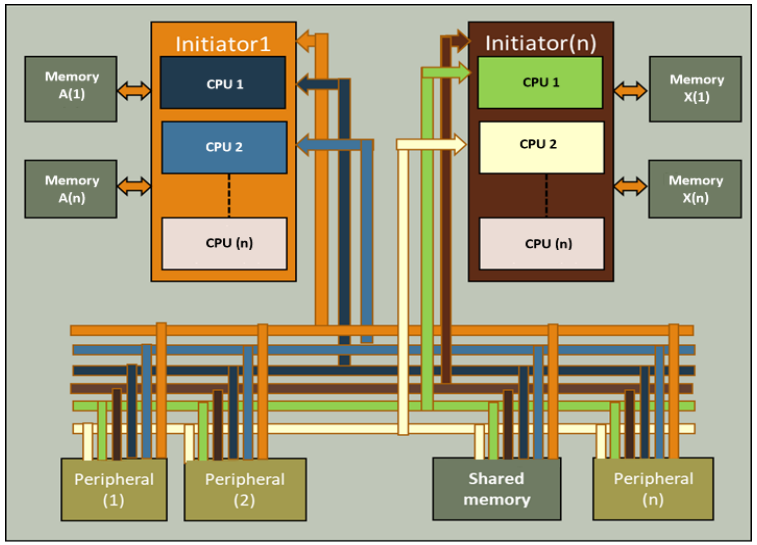



Fig.1 A typical SOC multi core design used for advanced user applications

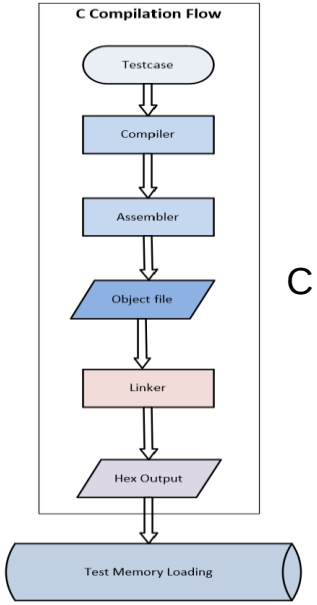



Fig. 2 State of the art:
C Based Verification Flow

# Proposed Solution: AUTOLNKGEN

- An automation framework to generate randomized linker command files by extracting executable regions from the structured "executable sheet" that specifies size and bounds for each region.
- Randomization vectors: Type of memories(latency, cache config), Start address (aligned/un-aligned) Execution range

Memory section to exists in linker

Start Address of memory section

End Address of memory section

Subset Length within addr limits

Memory width as per design

Division factor for Memory Randomization

| MEMORY_ID | START_ADDR | END_ADDR | LENGTH | MEMORY_WIDTH | DIV_FAC |
|---|---|---|---|---|---|
| ATCM | 0X100 | 0XFFFF | 0xE800 | 64 | 8 |
| BTCM | 0X100000 | 0X107FFF | 0x7000 | 64 | 8 |
| LSRAM0 | 0X20000000 | 0X2003FFFF | 0x30000 | 128 | 8 |

**AUTOLNKGEN**

EXEC_SPEC.csv

**Automation Framework for Linker Generation**

- Randomize the memory start address
- Randomize the execution memory section
- Randomized memory section size fit check

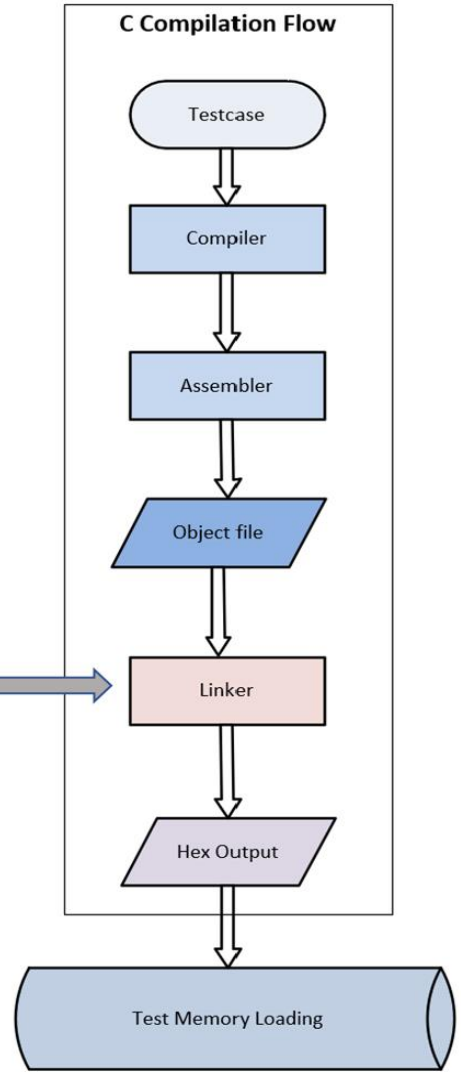
linker.cmd

Section present in linker

Section to be randomized

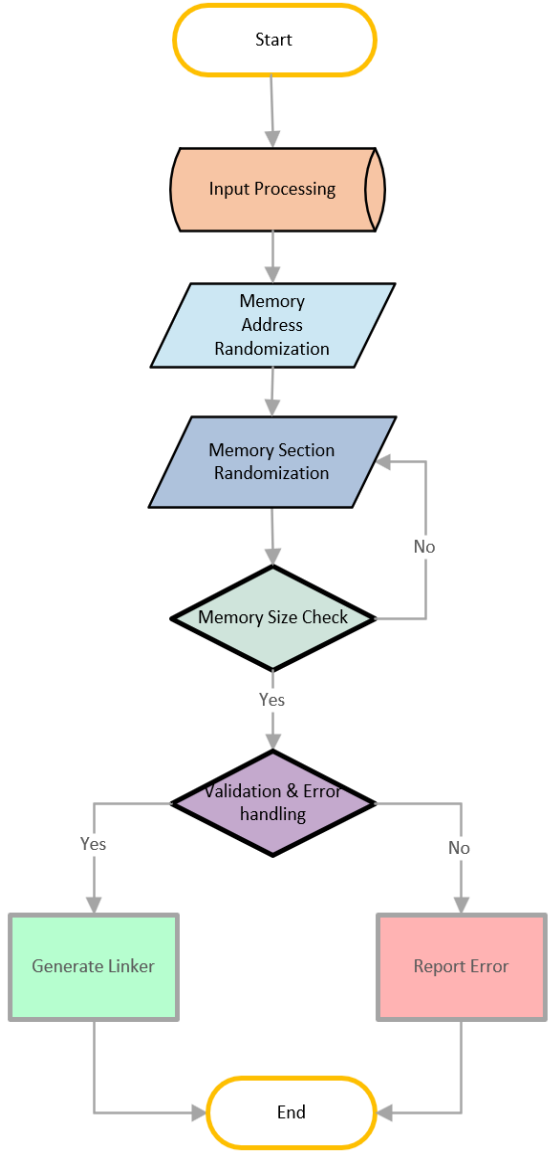
Maximum length of each section

| SECTION | SECTION_MEMORY | MAX_LENGTH |
|---|---|---|
| text.hypvectbl | default | 0x7F |
| text.startup | ATCM:LSRAM0 | 0x800 |
| code | ATCM:BTCM:LSRAM0 | 0x5000 |
| data | ATCM:BTCM:LSRAM0 | 0x2000 |



# Proposed Solution: AUTOLNKGEN

- **Input Processing**
  - ✓ AUTOLNKGEN framework parses the memory map from the executable spec to extract executable regions and linker template as inputs
  - ✓ These regions are structured into an "Executable Sheet", detailing size, bounds, and constraints for each memory section.
  - ✓ Linker template file is used for updating the memories and section after randomization.
- **Memory Randomization**
  - ✓ Randomization Algorithm: AUTOLNKGEN framework randomizes the memory address regions and sections defined in the executable sheet.
  - ✓ **Memory Address Randomization**: Size and bounds of each memory region are validated to ensure compliance with the constraints.
  - ✓ **Memory Section Randomization**: Different section of linker regions are randomized as per the constraints given in the executable sheet
  - ✓ **Memory size check**: Re-Randomization Trigger: If memory sections fail to fit within the bounds, the randomization process is iteratively re-triggered.
- **Validation and Error Handling**
  - ✓ The randomized memory sections are compared against the linker template to ensure compatibility.
  - ✓ In case of mismatches between executable spec and linker template, the framework generates errors specifying the problematic sections, enabling debugging and resolution.
- **Linker File Generation**
  - ✓ After successful validation, the framework processes the linker template and incorporates the randomized memory sections.
  - ✓ The finalized linker file is output, meeting the requirements of both the executable spec and linker template.



# Proposed Solution: AUTOLNKGEN

Section present in linker

Section to be randomized

Maximum length of each section

TEST ID: 1415 →

| SECTION | SECTION_MEMORY | MAX_LENGTH |
|---|---|---|
| text.hypvectbl | default | 0x7F |
| text.startup | ATCM:LSRAM0 | 0x800 |
| code | ATCM:BTCM:LSRAM0 | 0x5000 |
| data | ATCM:BTCM:LSRAM0 | 0x2000 |

→ Ignore / Exclude bins: TEST_ID_1415 x {CTCM, LSRAM1..3, GSRAM0,..GSRAM3} → Coverage Exclusion File

**Linker Randomization Coverage Metrics**

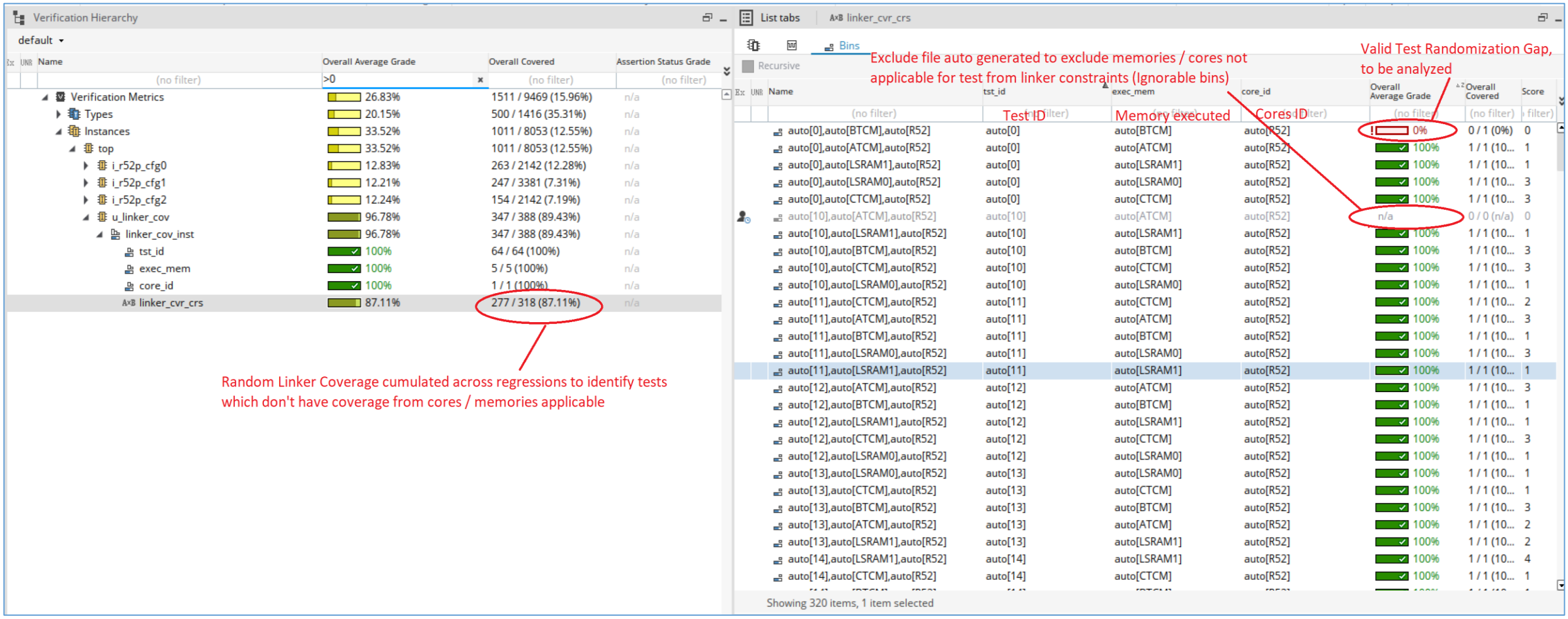
As linkers are randomly generated in runtime with constraints, we need metrics to analyze the random coverage:

- ✓ Functional Coverage defined to analyze cross coverage (Test ID x Memories Exercised x Processor Core Instances)
- ✓ These coverpoints are parsed as plusargs to functional coverage model (Test ID & Core from Test Plan, Memory from Randomization)
- ✓ Randomization Vector XLS used per test is used to auto generate exclusion list to exclude/ignore invalid combinations
- ✓ Across Regressions this functional coverage is enabled (Since it is one time coverage capture, runtime impact is miniscule)

```
// Memories in the SoC
typedef enum int {
                    ATCM,
                    BTCM,
                    CTCM,
                    LSRAM0,
                    LSRAM1,
                    LSRAM2,
                    LSRAM3,
                    LSRAM0,
                    LSRAM1,
                    GSRAM0,
                    GSRAM1,
                    GSRAM2,
                    GSRAM3

} executable_mem_e;

//Core instances in the SoC
typedef enum int {
                    R52_C0,
                    R52_C1,
                    R52_C2,
                    R52_C3,
                    C29_C0,
                    C29_C1,
                    C29_C2,
                    C29_C3,
                    M33_C0,
                    M4_C0,
                    M0_C0

} test_core_e;
```

```
module linker_cov;
  import uvm_pkg::*;
  typedef uvm_enum_wrapper#(executable_mem_e) exec_mem_args;
  string code_sec;
  string core_sec;
  logic [5:0] test_id;
  executable_mem_e code_sec_mem;
  test_core_e test_core;

  initial begin
    if ($value$plusargs ("TEST_ID=%0d", test_id)) //Test ID from test plan
      $display ("TEST_ID for testcase %d", test_id);
    if ($value$plusargs ("CODE_SEC=%s", code_sec))begin //Memory executed
      $display ("CODE_SEC is %s for testcase %d",code_sec, test_id);
      exec_mem_args::from_name(code_sec, code_sec_mem);
    end
    if ($value$plusargs ("CORE=%s", core_sec))begin //Core executed
      $display ("CORE_INST is %s for testcase %d",core_sec test_id);
      exec_mem_args::from_name(core_sec, test_core);
    end
    @(posedge top.nreset) linker_cov_inst.sample(); //Sampling
  end

  covergroup linker_cg;
    option.per_instance =1;
    tst_id: coverpoint test_id;
    exec_mem: coverpoint code_sec_mem;
    core_id: coverpoint test_core;

    linker_cvr_crs: cross tst_id, exec_mem, core_id; //Cross Coverage
  endgroup
  linker_cg linker_cov_inst = new();

endmodule
```

# Proposed Solution: AUTOLNKGEN




**Sample Regressions Coverage Metrics**

# Issue 1: Safe CPU Reset Anomaly: False Uncorrectable Error

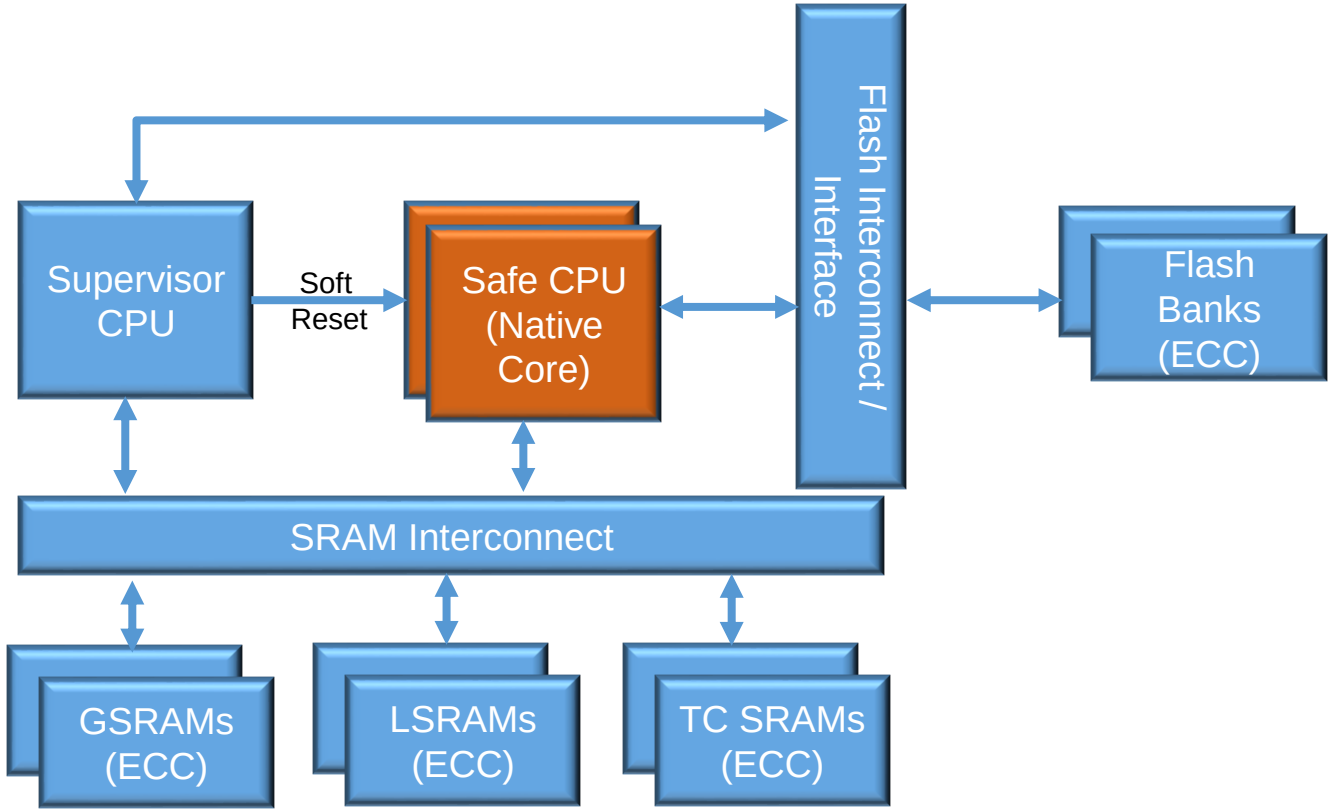
**Bug Description:** Supervisor core can soft-reset application cores which are safe CPUs with FuSa capabilities. System components (supervisor CPU, interconnects, flash, memories) have system reset, CPUs have additional soft reset. DV covered these soft reset tests of application core with both linear and discontinuous instruction execution from SRAMs and all passed. When the same test were randomized to execute from Flash using AUTOLNKGEN flow, random failures were observed

**System Setup:**
- Supervisor core can soft reset Safe CPU (application core)
- Safe CPU executing from flash banks with prefetch enabled

**Error Sequence:**
- Soft reset initiated during discontinuity instruction execution
- CPU outputs, including address bus, reset to safe state (zeroes)
- Flash interface performs ECC check with zeroed address
- ECC mismatch triggers false uncorrectable error
- Safe CPU enters hard fault state

**Root Cause:**
- Flash interconnect IP bug: Failed to handle request removal specifically during discontinuity fetch scenario when prefetch and code cache are enabled

**Impact:**
- High impact bug to silicon was detected in Pre-Silicon
- Potential Re-PG until such scenarios are thought thoroughly during verification and validation plan




Example SOC Architecture

# Issue 2: Flash Register Read Issue From Flash Code

**Bug Description (Legacy Devices Customer Feedback):** While executing from flash banks, reads to flash powerup status / config register (FBFALLBACK) returned zeroes (Which means that flash bank was powered down). The same reads were working fine , when read while code execution is from SRAMs. *This bug was the trigger for this methodology to brainstorm / evolve.* In Verification, Flash Register checks were run from SRAM and not from the same flash bank.

**System Setup:**
- FBFALLBACK register: Configures flash bank powerup state
- Initial setting: FBFALLBACK = 0x3 (Flash Bank is active)
- Code execution: From flash bank

**Error Sequence:**
- Code execution from flash initiates
- FBFALLBACK read returns unexpected 0x0
- Code execution switched to SRAM
- Subsequent reads show correct value (0x3)

**Root Cause:**
- On Flash read/fetch access, flash controller triggers bank "wake up" mechanism by which design internally writes '11' to FBALLBACK to activate the bank (Auto wakeup)
- This write occurs even if bank is already active (Design Issue)
- For simultaneous read/write, design prioritizes write to register but read returned false ‘0x0’ as return data as it got dropped off

**Impact:**
- Customer was aware that they have to jump to SRAM and execute to update flash bank configurations in runtime
- Now read APIs also follows same approach as workaround

**Table 6-6. FBFALLBACK Register Field Descriptions**

| Bit | Field | Type | Reset | Description |
|---|---|---|---|---|
| 31-16 | RESERVED | R | 0h | Reserved |
| 15-6 | RESERVED | R | 0h | Reserved |
| 5-4 | BNKPWR2 | R/W | 0h | Fall Back power mode<br>00 Sleep (Sense amplifiers and sense reference disabled)<br>01 Standby (Sense amplifiers disabled, but sense reference enabled)<br>10 Reserved<br>11 Active (Both sense amplifiers and sense reference enabled)<br>Reset type: SYSRSn |
| 3-2 | BNKPWR1 | R/W | 0h | Fall Back power mode<br>00 Sleep (Sense amplifiers and sense reference disabled)<br>01 Standby (Sense amplifiers disabled, but sense reference enabled)<br>10 Reserved<br>11 Active (Both sense amplifiers and sense reference enabled)<br>Reset type: SYSRSn |
| 1-0 | BNKPWR0 | R/W | 0h | Fall Back power mode<br>00 Sleep (Sense amplifiers and sense reference disabled)<br>01 Standby (Sense amplifiers disabled, but sense reference enabled)<br>10 Reserved<br>11 Active (Both sense amplifiers and sense reference enabled)<br>Reset type: SYSRSn |

**6.12 Procedure to Change the Flash Control Registers**

During Flash configuration, no accesses to the Flash or OTP can be in progress. This includes instructions still in the CPU pipeline, data reads, and instruction prefetch operations. To be sure that no access takes place during the configuration change, follow the procedure shown below for any code that modifies the Flash control registers.

1. Start executing application code from RAM/Flash/OTP.
2. Branch to or call the Flash configuration code (that writes to Flash control registers) in RAM. This is required to properly flush the CPU pipeline before the configuration change. The function that changes the Flash configuration cannot execute from the Flash or OTP and must reside in RAM.
3. Execute the Flash configuration code (located in RAM) that writes to Flash control registers like FRDCNTL, FRD_INTF_CTRL, and so on.
4. At the end of the Flash configuration code execution, wait eight cycles to let the write instructions propagate through the CPU pipeline. This must be done before the return-from-function call is made.
5. Return to the calling function that resides in RAM or Flash/OTP and continue execution.

# Issue 3: Cache Coherency Issue Across Initiators w/ Posted Writes

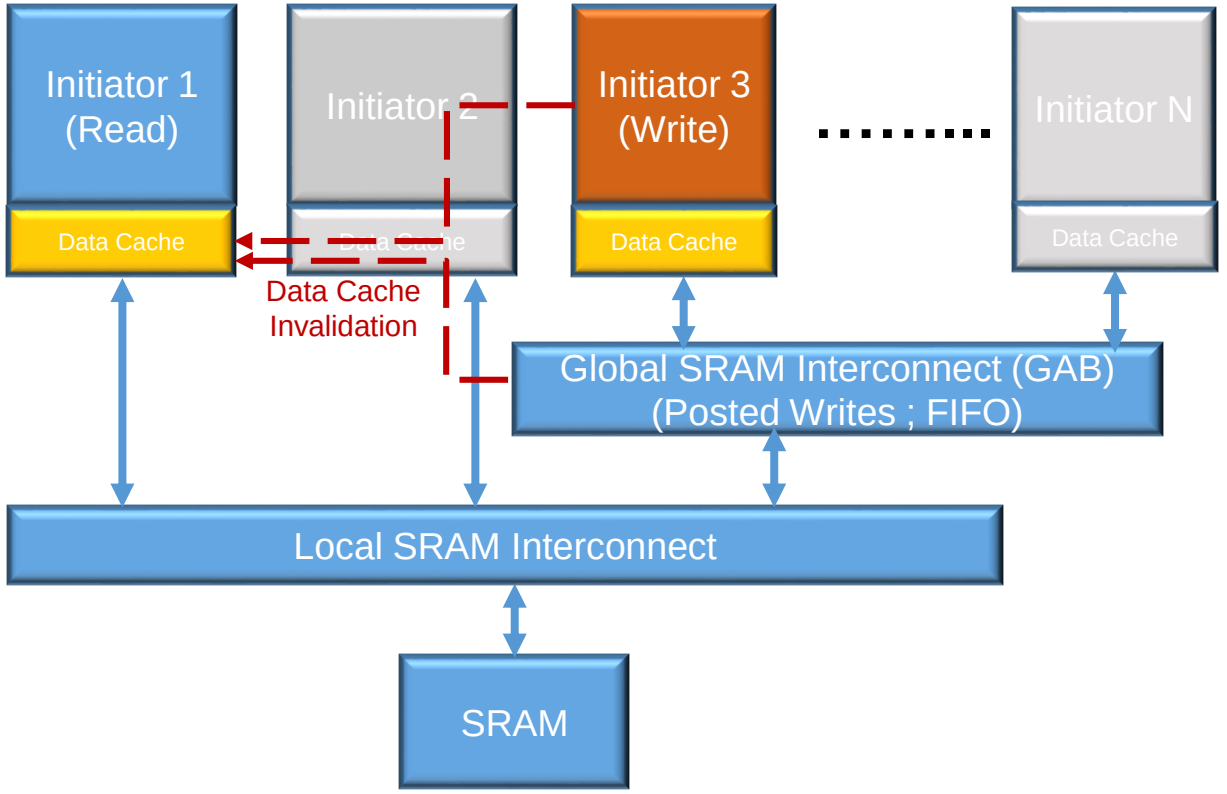
**Bug Description:** Initiator1 tries to poll (read continuously) on a particular SRAM address and there is simultaneous write from Initiator 3 to same SRAM address. (Typical cross core synchronization method using SRAM read /write operations other than MSGRAMs / IPC handshake). Randomly AUTOLNKGEN regression failed, the polling from Initiator 1 hangs forever with data from data cache representing old data. The write to memory happened when read via debugger, but it did not flush the data cache once the memory got written with ready handshake.

**System Setup:**
- Two initiators accessing same memory address
- Involves Global Interconnect / bridge for write path
- Direct Local RAM interconnect for read path

**Error Sequence:**
- Initiator1 polls SRAM and Initiator 3 writes same SRAM address
- Handshake should continue without hang

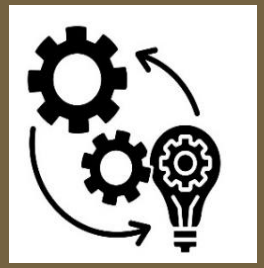
**Root Cause:**
- The data cache of Initiator 1, gets invalidated/flushed due to writes of other initiator to same address tag
- For particular initiators, Global SRAM interconnect (GAB) does arbitration and it does posted writes with internal FIFO
- Data Cache invalidation took care GAB as an initiator to invalidate, but since it was posted write (2 cycle pipelined), the flushing happened prior actual write to memory, and the read latched old data in cache

**Impact:**
- Need to lose data cache feature leading to performance loss / Re-PG



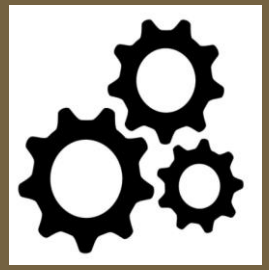

Example SOC Architecture

# Configurability, Scalability, Portability & Quality



**Generation Time Configurations of Linker file Components:**

- Specifying the memory region and its limit gives the flexibility to randomize bounds of memories
- Aligned and unaligned section combination creations for test-code execution
- Memory section configuration provides the flexibility to select which region to be used for different section of linker file



**Scalability & Portability across multiple and different CPU's in an SoC :**

- Separate executable sheet for each CPU helps in scaling this methodology can be easily expanded for multicore SoC
- Since the linker template is given as one of the input for the automation framework, this methodology can be expanded for different CPU like MCU, MPU and custom DSP

**Silicon Quality:**

- **This methodology helps in achieving exhaustive memory coverage, minimizing validation escapes, and improving the overall quality of SoC DV**
- **Achieves first pass success of rates by uncovering potential silicon bugs earlier in the verification cycle, ultimately leading to more robust and reliable SoC designs.**

# Summary & Results

**Evidence:**

- This randomization framework was deployed in a recent MCU SoC verification with heterogenous CPUs (including custom DSPs and ARM cores) and helped in identifying design issues in flash read interface and memory controllers for certain SoC use cases which could have slipped to silicon

**Results:**

- Usable across CPU based SoC verification / validation platforms
- Early hit of corner case silicon bugs
- Scalable and reusable framework to enable module owners to focus in creating context-agnostic module/system tests, while regression of these tests across SoC context (Memories) is random generated via the framework

**Summary:**

- Improves memory coverage and SoC Verification quality
- Enhances first-pass success rates by uncovering potential silicon bugs