UVM Based Generic Interrupt Service Routine (gISR)

Ritesh Mehta
Design Verification Engineer, Google
riteshmehta@google.com

Nakul Sharma
Senior Design Verification Engineer, Google
nakulssharma@google.com

Abstract

Complex System-on-Chip (SoC) designs extensively rely on intricate interrupt mechanisms for event handling. Traditional hardware interrupt verification uses unstandardized, manual, and error-prone methods that are difficult to scale or reuse, especially dealing with complex interrupt aggregation points. This paper introduces a novel, reusable UVM-based framework designed to overcome the aforementioned issues. We present a solution that models interrupts as a hierarchical object tree, featuring a central manager for building these hierarchies and an intelligent monitor that automates runtime source identification, reporting and end-of-simulation checks. The architecture's key innovation is its clear separation of DUT-specific integration from the generic, scalable, reusable logic. We detail the design, demonstrate its application on a complex Multimedia Processing Unit (MPU) and discuss its benefits for achieving scalable and robust interrupt verification.

Index Terms

UVM, Interrupt Verification, SoC, Hierarchy, Monitor, Reusability, SystemVerilog, Functional Verification.

I Introduction

MODERN System-on-Chip (SoC) designs often have complex interrupt structures due to the integration of multiple peripherals and processors. These interrupt paths are rarely simple, often involving intricate hierarchical controllers that aggregate, prioritize, and route events from many sources. Verifying this fabric is critical to ensure system resilience, data integrity, and overall stability, as errors can lead to severe system-level issues like race conditions, data corruption, or crashes.

Consider a typical SoC where a CPU receives interrupts from a DMA, UARTs, and custom peripherals. Each peripheral's internal events are aggregated into a block-level interrupt, which then feeds a system-level controller, ultimately asserting a single CPU IRQ pin. This tree-like structure, coupled with diverse interrupt behaviors (level-sensitive, pulse-sensitive) and varied status register clearing policies (e.g., Write-1-to-Clear (W1C), Read-to-Clear (RTC)), poses significant verification challenges that traditional, ad-hoc methods struggle to address comprehensively. Key among these are:

- Accurately identifying the precise root cause of an aggregated interrupt.
- Verifying that all expected interrupts are actually generated.
- Detecting spurious or uncleared *ghost* interrupts.

Traditional interrupt verification methods suffer from significant drawbacks due to their reliance on ad-hoc, manual tests. This approach provides limited coverage, failing to guarantee the generation of all expected interrupts and effectively missing spurious or *ghost* interrupts that fall outside a test's narrow scope. Simultaneously, limited visibility into the interrupt tree makes identifying the root cause of an aggregated interrupt a time-consuming manual debug task. These fundamental flaws are compounded by a lack of standardization across projects, poor scalability with growing design complexity, and low reusability of the tailor-made test cases, leading to suboptimal verification quality.

This paper introduces a generic Interrupt Service Routine (gISR) for HW - a novel, reusable, and scalable Universal Verification Methodology (UVM)-based framework for hierarchical interrupt verification. Our approach adapts the proven concept of software-based interrupt handling to solve complex hardware verification challenges. The framework is built on three interconnected classes— *generic_interrupt_properties*, *generic_interrupt_manager* and *generic_interrupt_monitor* —that together form a powerful and extensible verification solution.

II Related Work and Motivation

The landscape of interrupt verification in SoC environments has traditionally been fragmented, with most methodologies being highly manual, non-reusable, and project-specific. While modern UVM-based verification has matured for standard interfaces and protocols, interrupt verification has lagged behind in standardization, tooling, and automation.

A. Traditional Verification Approaches

Historically, interrupt verification often relied on direct signal monitoring and Verilog-based testbenches. These methods typically involved manual observation of interrupt pins and status registers using waveform viewers or simple `always` blocks. Such approaches are highly manual, error-prone, and lack the flexibility, reusability, and scalability required for modern complex designs.

B. UVM-Based Interrupt Handling

Some prior UVM-based testbenches implemented interrupt sequencing via RAL blocks or signal monitors tied to interrupt pins. While moderately reusable at the block level, these approaches falter at SoC scale due to:

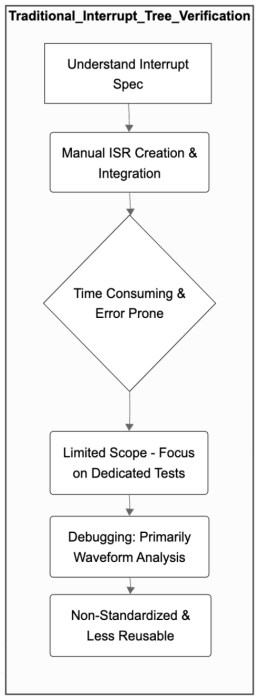
- Lack of centralized interrupt hierarchy models.
- Inability to trace multi-level interrupt propagation paths.
- No automated runtime checks or end-of-simulation reporting.

C. Our Contribution

To address these shortcomings, we propose the gISR framework — a reusable, self-contained UVM package that:

- Models interrupt trees hierarchically via generic node objects.
- Allows rapid integration of DUT-specific interrupt registers via configuration APIs.
- Automates runtime interrupt tracing, status register decoding, and ghost interrupt detection.
- Provides structured reporting, end-of-sim analysis, and plug-and-play reuse across IPs and SoCs.

By aligning interrupt verification with how real software ISRs process and clear interrupts, gISR offers a scalable and standardized solution for a long-standing gap in verification methodology.



Aspect	Traditional Method	gISR Framework
Standardization	**Inconsistent**; lacks a unified, repeatable approach across projects.	**Consistent**; enforces a standardized methodology for all interrupt verification.
Automation	**Manual**; relies on time and effort-intensive test case creation.	**Automated**; provides an efficient and faster verification flow through its core engine.
Scope & Coverage	**Limited**; often misses complex corner cases, leading to potential bug escapes.	**Holistic**; provides comprehensive coverage of the entire interrupt tree structure.
Debugging	**Slower**; heavily focused on manual, time-consuming waveform analysis.	**Faster**; relies on focused, automated logging for quick root cause identification.
Reusability	**Low**; leads to redundant work as tests are tailor-made for specific designs.	**High**; enables efficient reuse of the verification framework across different projects.
Scalability	**Difficult**; faces significant challenges with increasing design complexity.	**Easy**; handles complexity well and scales effectively with larger and deeper interrupt trees.

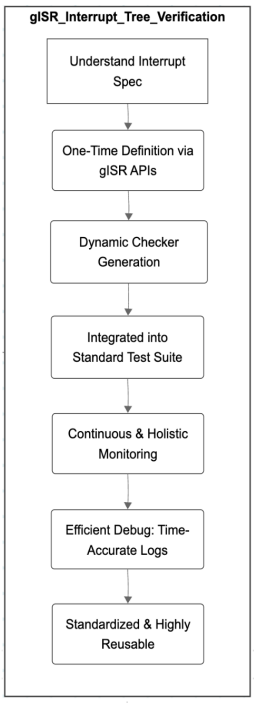


Fig. 1: Comparative Analysis.

III The Idea: Building the Interrupt Hierarchy

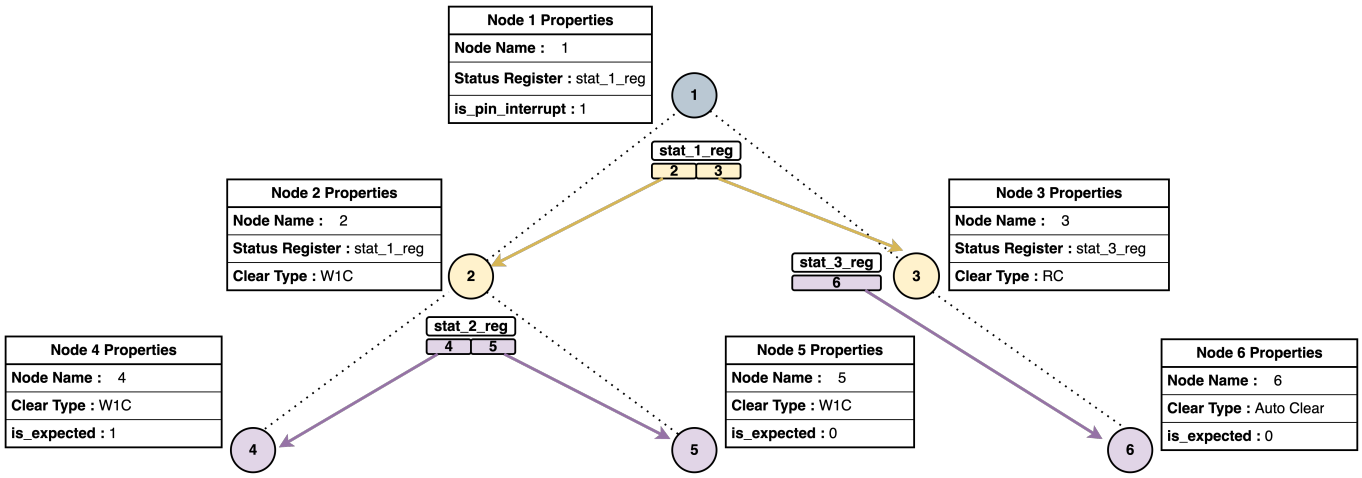


Fig. 2: Simplified block diagram of an interrupt node structure.

The core idea behind representing an interrupt hierarchy involves modeling each interrupt source as a class-based node. This node, a logical interrupt entity, encapsulates its relevant characteristics and can link to both its parent and child nodes, thereby constructing a complete interrupt tree hierarchy from the top-level pin down to the deepest individual event sources.
As illustrated in Figure 2, the following steps outline the creation of a conceptual interrupt tree hierarchy:

1) **Top-Level Node Creation (Node 1):** The process begins by creating Node 1, representing a top-level interrupt pin. This node is assigned a unique identifier, *1*, and is associated with a status register named *stat_1_reg*.
   - Crucially, *stat_1_reg* contains two logical fields, *2* and *3*. These fields indicate the next level of child interrupt nodes for Node 1, thus linking Node 2 and Node 3 as child interrupts to Node 1.
2) **Intermediate Node Creation (Nodes 2 and 3):** Following the top node, its child interrupt nodes, Node 2 and Node 3, are created.
   - For Node 2, its properties include its name, *2*, a defined clearing mechanism of Write-1-to-Clear (W1C), and an associated status register, *stat_2_reg*.
     - *stat_2_reg* further consists of logical fields *4* and *5*, which in turn define the child interrupt nodes for Node 2.
   - Similarly, Node 3 is created with its name, *3*, a clearing mechanism of Read-to-Clear (RC), and an associated status register, *stat_3_reg*.
     - *stat_3_reg* contains logical field *6*, which defines the child interrupt node for Node 3.
3) **Leaf Node Creation (Nodes 4, 5, and 6):** The process continues recursively until leaf-level interrupt nodes are created. Nodes 4, 5, and 6 are examples of such leaf nodes. They are characterized by not having any further associated status registers, indicating they are the deepest sources in their respective branches.
   - Each leaf node includes a flag - *is_expected* indicating if this specific interrupt source was expected to be asserted during the test.
   - Node 4 has a Write-1-to-Clear (W1C) clearing mechanism and is marked as expected.
   - Node 5 has an automatic clearing mechanism and is marked as not expected.
   - Node 6 has a Write-1-to-Clear (W1C) clearing mechanism and is marked as not expected.

This systematic, recursive process ensures that the entire interrupt network, from top-level pins to individual event sources, is accurately modeled within the verification environment.

IV The Algorithm: Interrupt Hierarchy Tree Traversal

The interrupt traversal algorithm is a recursive, post-order process designed to systematically identify and service every active interrupt source. As illustrated in Figure 3, the process consists of two main phases: a **Descent** to pinpoint all active leaf-level sources, followed by an **Ascent** to clear the sources and their corresponding status bits up the hierarchy. This ensures that an interrupt is not cleared at a higher level until its specific root cause has been fully serviced.
The following steps detail this process, corresponding to the numbered actions in the diagram.

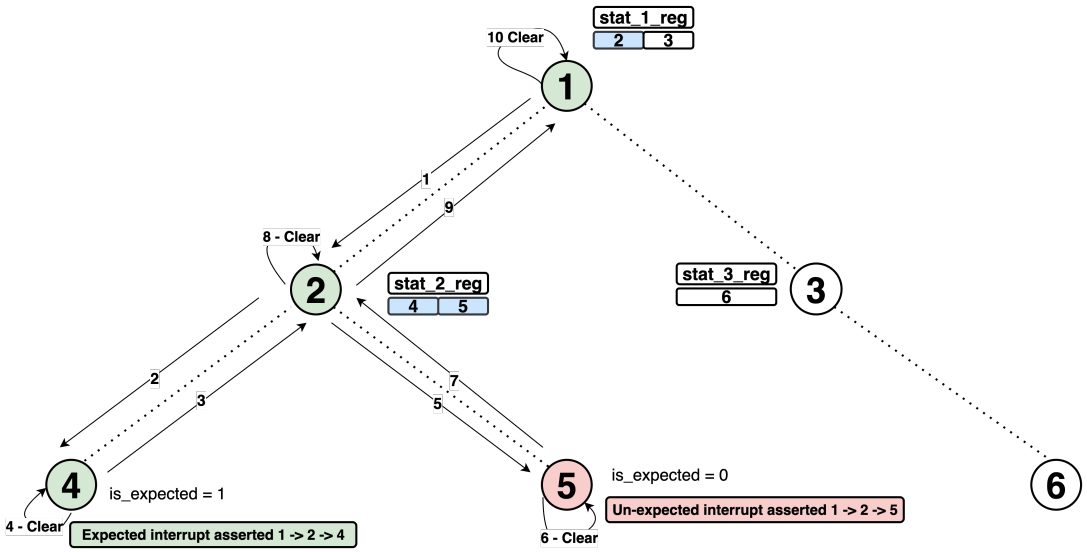


Fig. 3: Simplified block diagram of an interrupt tree traversal.

Phase 1: Recursive Descent for Source Identification The first phase involves traversing down the tree by reading status registers to find the root cause(s) of the interrupt.

1) **Descend from Root to Intermediate Node :** The process begins at the top-level **Node 1**. The monitor reads its status register, `stat_1_reg`, finds the bit for **Node 2** is active, and makes a recursive call to process Node 2.
2) **Descend to First Leaf :** Now at **Node 2**, the algorithm reads `stat_2_reg` and finds the bit for **Node 4** is active. It makes another recursive call, descending to this leaf node.
3) **Descend to Second Leaf :** The algorithm continues checking `stat_2_reg` and finds the bit for **Node 5** is also active. A separate recursive call descends to this second leaf. The descent phase is now complete, as all active leaf sources (**4** and **5**) have been reached.

Phase 2: Post-Order Ascent for Clearing The second phase begins after the recursion reaches the leaf nodes. The algorithm processes each source and works its way back up the tree.

4) **Clear First Leaf Source :** The recursive call for **Node 4** executes. It identifies the node as an expected interrupt (`is_expected = 1`) and performs the necessary clear operation.
5) **Process Second Leaf Source :** The recursive call for **Node 5** executes. The algorithm reads the node's properties and identifies it as an unexpected interrupt (`is_expected = 0`), which would be logged as an error.
6) **Clear Second Leaf Source :** Following the check, the clear operation for **Node 5** is performed.
7) **Return to Parent Node :** With all its active leaves serviced, the recursive calls for nodes 4 and 5 complete, and control returns to the parent, **Node 2**.
8) **Clear Status Bits in Parent :** Now in the post-order phase for **Node 2**, the algorithm clears the bits corresponding to '4' and '5' within `stat_2_reg`, acknowledging that its children have been serviced.
9) **Clear Status Bit in Root :** The call for Node 2 completes, and control returns to the root. The algorithm clears the bit for '2' within `stat_1_reg`, acknowledging that the entire branch is now serviced.
10) **Clear Top-Level Interrupt :** Finally, with all underlying sources and status bits cleared, the top-level interrupt at **Node 1** is de-asserted, completing the entire service routine.

This recursive, post-order algorithm provides an effective and reusable solution that is straightforward to implement and debug for any hierarchical interrupt design.

V Exclusive Verification Checks

While the core of gISR focuses on modeling and tracing interrupts hierarchically, its value lies equally in the suite of automated checks it provides during runtime and end-of-simulation phases. These exclusive checks ensure correctness, coverage, and debug visibility across the interrupt fabric.

A. Initialization Checks

- **Interrupt Tree Sanity Check:** At the start of simulation, the monitor performs a structural validation of the interrupt tree. Each node is verified to have valid parent-child associations and correctly linked status register fields.

B. Runtime Checks

- **Hierarchical Assertion Consistency:** Whenever a parent node is asserted, at least one of its child nodes must also be asserted. If a parent is active without an asserted child, a potential misrouting or false aggregation is flagged.
- **Leaf Node Expectation Validation:** For all leaf nodes marked as `expected = 1`, their assertion is tracked. If such a node never gets triggered, a warning or error is logged, indicating testbench coverage gaps or RTL issues.
- **Auto-Clear and Re-Assertion Verification:** If an interrupt is configured with automatic clearing (e.g., pulse or auto-reset behavior), gISR validates that it de-asserts correctly after detection. Re-assertions are separately tracked to avoid false positives.
- **Status Register Monitoring:** For all active interrupts, the status registers of the corresponding nodes are continuously monitored. If a register remains uncleared despite servicing, gISR re-initiates traversal and highlights possible RTL clearing logic bugs.

C. End-of-Simulation Analysis

- **Expected Assertion Verification:** At simulation end, all interrupts marked as expected (`expected = 1`) are verified to have been asserted at least once. This guarantees that the functional stimulus has exercised all configured scenarios.
- **Global Register Cleanup Check:** All interrupt-related status registers are checked for final cleared state. Any residual bits are flagged to catch incomplete servicing, dangling events, or status register RTL mismatches.

These runtime and end-of-sim checks are critical for scalable, reusable environments where hundreds of interrupts must be validated across multiple integration levels. All check results are standardized through UVM messaging and can optionally be tied to scoreboard coverage or waveform viewers.

VI Proposed UVM Architecture

Our framework provides a structured and reusable solution built upon three core UVM base classes and a well-defined integration layer for the Device Under Test (DUT). The architecture's key recipe lies in its clear separation of the generic, reusable logic from the DUT-specific implementation details. This ensures that the core verification engine remains unchanged across different projects, requiring only a small, specific adapter layer for each new DUT.

The three base classes work in concert to model and verify the interrupt system:

- **`generic_interrupt_properties`:** A UVM object that acts as the data model. Each instance of this class represents a single interrupt source as a **node** in the hierarchy, containing all of its static properties.
- **`generic_interrupt_manager`:** A UVM object that serves as a factory and central database. It is responsible for **building** the complete interrupt tree from the individual node objects at the start of the simulation.
- **`generic_interrupt_monitor`:** A UVM component that contains the active verification engine. It uses the model created by the manager to autonomously **trace and validate** interrupt behavior against the rules defined in the properties.

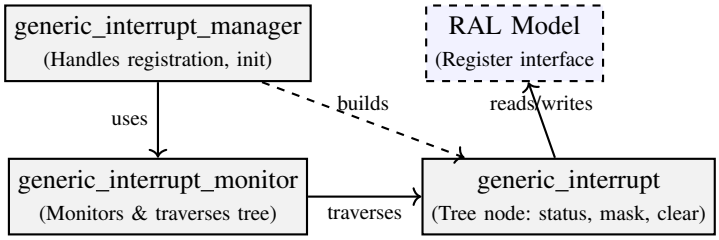


Fig. 4: Class Architecture of gISR Framework

A. The Interrupt Verification Challenge: An MPU Example

To illustrate the complexities, we use a Multimedia Processing Unit (MPU) as a reference. The MPU integrates Audio, Video, and Image processing blocks, each generating events that funnel up to a single CPU interrupt pin.

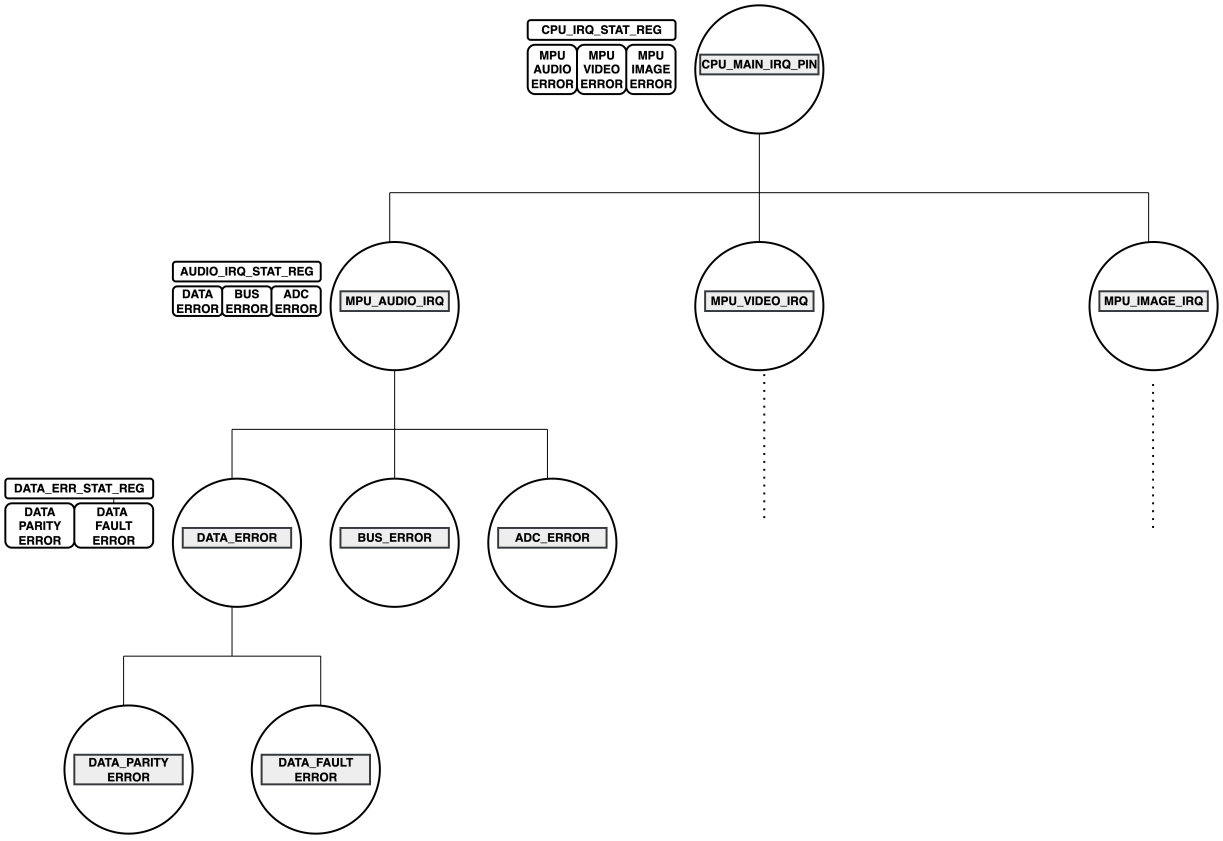


Fig. 5: Simplified block diagram of the MPU interrupt hierarchy.

B. `generic_interrupt_properties`: The Hierarchical Node Model

This UVM object represents a logical interrupt node and holds its properties within the tree structure. Below is a sample of key methods and configuration fields.

```
class generic_interrupt_properties#(type INTR_NODE = generic_interrupt_properties) extends uvm_object;

  string interrupt_name;
  string top_interrupt_name;
  INTR_NODE interrupt_parent_node;
  string status_registers_q[$];
  string mask_registers[string]; // mask register maps to one of the status registers
  clear_type_enum status_reg_clear_type[string]; // clear type register maps to one of the status register's fields
  bit expected_interrupt;
  bit interrupt_asserted;
  INTR_NODE interrupt_child_node[string];

  // --- Helper functions to set and get interrupt node properties ---

 // Adds a child interrupt node to the hierarchy under a specified parent within this subtree.
  function void add_interrupt_child_node(
    string top_interrupt_name = "",
    string parent_src_name = "",
    string interrupt_name = "",
    string status_reg[$] = {},
    string mask_reg[] = '{}
  );
    INTR_NODE parent_node;
    // Find the parent node within the hierarchy rooted at this.
    parent_node = this.find_interrupt_node_by_name(parent_src_name);
   if(parent_node != null) begin
       parent_node.add_node(/*child node properties*/);
  endfunction

endclass
```

Listing 1: Partial implementation of interrupt node property class

Each node also tracks whether it is a leaf or intermediate interrupt, status register names, clear policies, and severity or any other properties if extended.

```
class mpu_interrupt_properties extends generic_interrupt_properties#(mpu_interrupt_properties);

  mpu_register_access_type_e mpu_reg_acc_type[string];
  mpu_register_severity_type_e mpu_reg_everity_type[string];

  ....

endclass
```

Listing 2: MPU interrupt properties extending from `generic_interrupt_properties` to add it's custom properties

C. `generic_interrupt_manager`: The Central Coordinator

This manager acts as the single point for building and storing the entire interrupt hierarchy.

```
class generic_interrupt_manager#(type INTR_NODE = generic_interrupt_properties)  extends uvm_object;

  // Associative array storing only the TOP-LEVEL interrupt nodes
  INTR_NODE top_interrupt_node[string];

  // This unction creates the interrupt node for top pin level interrupts and adds them to associative array `
     top_interrupt_node`
  extern virtual function void create_top_interrupt_node(string node_name, string status_reg_q[$]);

  extern virtual function void add_child_interrupt(string top_interrupt_name = "", string parent_interrupt_name = "", string
     interrupt_name = "", string status_reg[$] = {}, string mask_reg[] = '{});

    INTR_NODE top_node = get_top_interrupt_node(top_interrupt_name);

    top_node.add_interrupt_child_node(
     // Child node properties i.e node_name, status_registers, parent_node_name etc. ...
    );
  endfunction

  // This function finds and returns the interrupt node handles based on the node_name inside the specified top interrupt
  extern virtual function INTR_NODE get_interrupt_node(string top_interrupt_name, string node_name);
endclass
```

Listing 3: partial implementation of `generic_interrupt_manager`

This `generic_interrupt_manager` is extended by the user to create `mpu_interrupt_manager` which instantiates and creates the complete interrupt hierarchy tree.

```
class mpu_interrupt_manager extends generic_interrupt_manager#(type INTR_NODE = mpu_interrupt_properties);

  function void add_mpu_top_interrupt_node(string name);

    create_top_interrupt_node(
      .node_name("CPU_MAIN_IRQ_PIN"),
      .status_reg_q({"CPU_IRQ_STAT_REG"})
    );
  endfunction

  function void add_mpu_child_interrupt_nodes();

    // ---- First Level child interrupt nodes ----
    add_child_interrupt(
      .top_interrupt_name("CPU_MAIN_IRQ_PIN"),
      .parent_interrupt_name("CPU_MAIN_IRQ_PIN"),
      .interrupt_name("MPU_AUDIO_IRQ"),
      .status_reg("AUDIO_IRQ_STAT_REG"),
    );
    // Similarly for MPU_VIDEO_IRQ and MPU_IMAGE_IRQ ...

    // ----- Second level interrupt nodes ,Child nodes of MPU_AUDIO_IRQ -> DATA_ERROR ------
    add_child_interrupt(
      .top_interrupt_name("CPU_MAIN_IRQ_PIN"),
      .parent_interrupt_name("MPU_AUDIO_IRQ"), // parent node as MPU_AUDIO_IRQ
      .interrupt_name("BUS_ERROR")
    );
    // Similarly for BUS_ERROR and ADC_ERROR ...

    // ---- Third level interrupt nodes ,Child nodes of MPU_AUDIO_IRQ -> DATA_ERROR -> DATA_PARITY_ERROR ----
    add_child_interrupt(
      .top_interrupt_name("CPU_MAIN_IRQ_PIN"),
      .parent_interrupt_name("DATA_ERROR"), // parent node as DATA_ERROR
      .interrupt_name("DATA_PARITY_ERROR")
    );
    // Similarly for DATA_FAULT_ERROR ...
  endfunction

endclass
```

Listing 4: `mpu_interrupt_manager` extending `generic_interrupt_manager` to build the interrupt hierarchy

D. generic_interrupt_monitor: The Verification Engine

This UVM component implements the main interrupt tracking algorithm. It tracks all top-level interrupt signals and recursively traces down the hierarchy using status registers.

```
class generic_interrupt_monitor#(type INTR_NODE = generic_interrupt_properties, type INTERRUPT_MANAGER_OBJECT =
     generic_interrupt_manager) extends uvm_monitor;

  `uvm_component_utils(generic_interrupt_monitor)

  INTERRUPT_MANAGER_OBJECT interrupt_manager;

  // This task MUST be implemented by User in extended interrupt_monitor
  extern virtual task read_status_reg(string a_status_reg_name, INTR_NODE a_interrupt,
     ref uvm_reg_data_t a_status_register_val[string]);

  // This task MUST be implemented by User in extended interrupt_monitor
  extern virtual task read_status_reg_field(string a_status_reg_name, string a_status_reg_field, INTR_NODE a_interrupt,
     output uvm_reg_data_t a_rd_data);

  // This task MUST be implemented by User in extended interrupt_monitor
  extern virtual task write_status_reg_field(string a_status_reg_name, string a_status_reg_field,
     INTR_NODE a_interrupt,uvm_reg_data_t a_wr_data);

  // This task is to be implemented in the extended class where it should wait for the trigger of every individual top interurpt
  extern virtual task wait_for_interrupt_trigger(string a_interrupt_source_name, output bit a_is_interrupt_asserted);

  task run_phase(uvm_phase phase);
    super.run_phase(phase);

    check_interrupt(); // Checks the structural validity of the interrupt heirarchy
    monitor_interrupt();
  endtask: run_phase


  task monitor_interrupt();
    // based on interrupt type , foreach top interrupt
    process_pulse_interrupt(interrupt_name);
  endtask: monitor_interrupt

  task process_level_interrupt(string a_interrupt_source_name);
    // partial implemetation to show that we wait for the pin/top interrupt and proceed to process it
    wait_for_interrupt_trigger(a_interrupt_source_name);
    process_interrupt_source(a_interrupt_source_name);
  endtask: process_level_interrupt

  task process_interrupt_source(string node_name);
    generic_interrupt_properties node = mgr.get_interrupt_node(node_name);

    // Check if this is a leaf interrupt
    if (node.interrupt_child_node.num() == 0) begin
      report_leaf_interrupt_detected(node_name);
      clear_status_reg(node_name);
    end
    else begin
      foreach (node.interrupt_child_node[child_name]) begin
        bit active;
        read_status_reg(child_name, active);
        if (active)
          process_interrupt_source(child_name); // recursive call
          // Check for interrupt to be cleared
          // Check if the interrupt was re-asserted , if yes then call process_interrupt_source again
      end
    end
  endtask
endclass
```

Listing 5: Partial implementation of generic_interrupt_monitor

This generic_interrupt_monitor is extended by the user to create mpu_interrupt_monitor which provides the necessary extended APIs like to read and write registers as per user's environment.

```
class mpu_interrupt_monitor extends generic_interrupt_monitor#(mpu_interrupt_properties, mpu_interrupt_manager);

  `uvm_component_utils(mpu_interrupt_monitor)

  virtual interface mpu_intr_if mpu_if;

  extern task read_status_reg(string a_status_reg_name, INTR_NODE a_interrupt, ref uvm_reg_data_t a_status_register_val[string]);
  extern task read_status_reg_field(string a_status_reg_name, string a_status_reg_field, INTR_NODE a_interrupt, output
     uvm_reg_data_t a_rd_data);
  extern task write_status_reg_field(string a_status_reg_name, string a_status_reg_field, INTR_NODE a_interrupt,uvm_reg_data_t
     a_wr_data);
  extern task wait_for_interrupt_trigger(string a_interrupt_source_name, output bit a_is_interrupt_asserted);


endclass: mpu_interrupt_monitor
```

```
  task mpu_interrupt_monitor::read_status_reg(string a_status_reg_name, INTR_NODE a_interrupt, ref uvm_reg_data_t
     a_status_register_val[string]);

    uvm_reg mpu_reg;
    uvm_reg_field reg_fields[$];

    mpu_reg = get_mpu_register_by_name(a_status_reg_name);
    mpu_reg.read();
    mpu_reg.get_fields(reg_fields);

    foreach (reg_fields[i]) begin
      a_status_register_val[reg_fields[i].get_name()] = reg_fields[i].get();
    end
  endtask

  task mpu_interrupt_monitor::read_status_reg_field(string a_status_reg_name, string a_status_reg_field, INTR_NODE a_interrupt,
     output uvm_reg_data_t a_rd_data);

    // Same as above
    a_rd_data = reg_fields["a_status_reg_field"].get();
  endtask

  task mpu_interrupt_monitor::write_status_reg_field(string a_status_reg_name, string a_status_reg_field, INTR_NODE
     a_interrupt,uvm_reg_data_t a_wr_data);
    uvm_reg_data_t wr_data;
    ....
    // Read register create data and update relevant field of wr_data with a_wr_data
    mpu_reg.write(wr_data);
  endtask

  task mpu_interrupt_monitor::wait_for_interrupt_trigger(string a_interrupt_source_name, output bit a_is_interrupt_asserted);

    if(a_interrupt_source_name == "CPU_MAIN_IRQ_PIN") begin
      wait(mpu_if.cpu_main_irq == 1);
      a_is_interrupt_asserted = 1;
    end
  endtask: wait_for_interrupt_trigger
```

Listing 6: `mpu_interrupt_monitor` extending `generic_interrupt_monitor`

E. Recursive Traversal Algorithm

The core of the monitor's intelligence is the `process_interrupt_source` method, a recursive algorithm that autonomously pinpoints the precise root cause of any top-level interrupt. The process is analogous to a manager finding the source of a fire alarm in a large building: they start at the main panel and follow the active signals down to the specific room before resetting the system back up the chain.

```
task process_interrupt_source(INTR_NODE current_node);

  // Determine if the node is a leaf or a branch in the tree.
  if (current_node.is_a_leaf_node()) begin
    // Check expectations for the leaf source.
    check_expected_interrupt(current_node);
  end
  else begin
    bit sources_still_asserted;
    do begin
      uvm_reg_data_t status_data[string][string];

      // a. Read all associated status registers from the DUT.
      read_all_status_registers(current_node, status_data);

      // b. Find asserted children, recurse, and clear.
      foreach (status_data[reg_name][field_name]) begin
        if (status_data[reg_name][field_name] != 0) begin
          INTR_NODE child_node;

          child_node = get_child_node_for_field(current_node, field_name);
          if (child_node != null) begin
            // Recurse into the child node to process the next level.
            process_interrupt_source(child_node);
            // Clear the source based on its defined policy (e.g., W1C).
            clear_interrupt_source(reg_name, field_name, child_node.get_clear_type());
            // Verify that the register field has been cleared in the DUT.
            check_if_interrupt_source_is_cleared(reg_name, field_name);
          end
        end
      end
      // c. Re-read registers to handle any new or persistent interrupts.
      sources_still_asserted = check_if_sources_still_asserted(current_node);
    end while (sources_still_asserted);
  end

  current_node.set_interrupt_asserted(1);
endtask: process_interrupt_source
```

Listing 7: Pseudocode for Recursive Interrupt Traversal

The algorithm executes in two distinct phases:

1) **Recursive Descent for Source Identification**
   When a top-level interrupt is detected (e.g., at `CPU_MAIN_IRQ_PIN`), the traversal begins:
   a) The monitor reads the status register of the current node.
   b) For each active bit in the register, it identifies the corresponding child node and makes a **recursive call** to itself with that child node as the new input.
   c) This process repeats, efficiently navigating down the tree by following only the active branches. Inactive branches like `MPU_AUDIO_IRQ_N` are instantly ignored, minimizing simulation overhead.

```
CPU_MAIN_IRQ_PIN
   |-- MPU_AUDIO_IRQ_N      [inactive]
   |-- MPU_IMAGE_IRQ_N      [inactive]
   |-- MPU_VIDEO_IRQ_N      [active]
         |-- Video_Decode_Error [active] -> Source Found!
```

Fig. 6: Recursive descent follows only active paths to efficiently locate the asserted source.

2) **Post-Order Ascent for Clearing and Logging**
   The actual verification logic occurs during the return path, after recursion reaches a leaf node:
   a) **At the Leaf:** When a call reaches a leaf node (e.g., `Video_Decode_Error`), it is identified as a root cause. The monitor logs the finding, checks it against test expectations, and applies the clearing policy defined in its properties (e.g., W1C).
   b) **Returning to the Parent:** As each recursive call completes, control returns to the parent node. The algorithm clears the corresponding status bit in the parent's register, acknowledging that the child's interrupt has been serviced.

This post-order traversal continues upward to the root, ensuring that aggregated interrupts are not cleared until all of their children are handled. The recursive design enables the monitor to dynamically trace all asserted sources and independently resolve simultaneous or cascaded interrupts.

Overall, this recursive traversal strategy forms the core of the monitor's intelligence—transforming interrupt verification from signal-level polling into a scalable, structure-aware checking methodology.

F. Integration into the UVM Testbench

Integrating the generic interrupt framework into a DUT-specific UVM testbench is a straightforward process handled during the standard UVM build and connect phases. The goal is to link the generic monitor's logic to the specific DUT's signals and to the interrupt hierarchy model built by the manager.
The integration follows three primary steps:

1) **Build Components:** In the UVM environment's 'build_phase', the user instantiates their DUT-specific monitor (e.g., `mpu_monitor`) and the corresponding manager (e.g., `mpu_interrupt_manager`). The manager's hierarchy-building methods are then called to construct the complete interrupt tree model for the DUT.
2) **Share the Manager Handle:** The handle to the populated interrupt manager is passed to the monitor using the 'uvm_config_db'. This gives the monitor access to the complete interrupt map.
3) **Connect Interfaces:** In the 'connect_phase', the final connections are made. The monitor's handles to the DUT's virtual interface (for observing physical interrupt pins) and the RAL model (for register access) are assigned.

The code snippet below shows a typical implementation of the 'connect_phase', where the monitor ('intr_mon') is retrieved from the configuration database and linked to the manager ('mgr') and the DUT's virtual interface ('vif').

```
virtual task my_env::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  // Get the monitor handle from the config database
  if (!uvm_config_db#(generic_interrupt_monitor)::get(this, "", "interrupt_mon", intr_mon))
    `uvm_fatal("CFG", "Failed to get interrupt monitor")

  // Link the monitor to the manager, ral model and the DUT interface
  intr_mon.set_interrupt_manager(mgr);
  intr_mon.set_ral_model(env_ral_model);
  intr_mon.dut_vif = vif;
endtask
```

Listing 8: Monitor integration in the UVM environment's connect_phase.

Once these steps are complete, the components interact seamlessly at runtime, as shown in Figure 7. The monitor observes the DUT. When an interrupt is triggered, it uses the manager to get the node's properties and then reads the DUT's status registers to begin its recursive tracing algorithm.

The key advantage of this architecture is its clean separation of concerns. The manager holds the static model of the hierarchy, while the monitor contains the dynamic, reusable verification logic. This allows each leaf source to be detected automatically and the appropriate policies (like W1C or auto-clear) to be executed based on the metadata in its node, making the entire process robust and scalable.

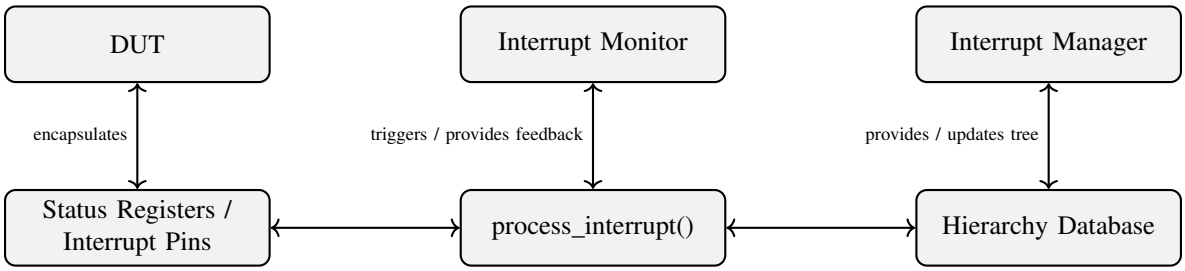


Fig. 7: Interrupt processing flow in the gISR framework.

The following results highlight the effectiveness of the proposed framework across multiple real-world SoC programs. Each scenario demonstrates how gISR reduced manual effort, enabled scalable reuse, and improved interrupt validation quality. Quantitative and qualitative metrics from production use-cases are discussed next.

VII Use Cases and Verification Scenarios

The architecture supports a wide range of interrupt verification scenarios, which are programmed in UVM sequences that follow a clear pattern of **Expect -> Stimulate -> Verify**. The sequence first informs the framework of an expected interrupt, then applies stimulus to the DUT to trigger it, and finally waits for the monitor to confirm that the specific interrupt was correctly processed.

```
class audio_codec_done_test_seq extends uvm_sequence;
  `uvm_object_utils(audio_codec_done_test_seq)

  // Handles to the manager and monitor, obtained from uvm_config_db
  generic_interrupt_manager m_intr_mgr;
  my_mpu_monitor            m_intr_mon;

  virtual task body();
    // 1. EXPECT: Inform the framework that this interrupt is expected.
    m_intr_mgr.set_expected_interrupt("Audio_Codec_Done", 1, "MPU_CPU_IRQ");

    // 2. STIMULATE: Trigger the event in the DUT.
    start_audio_codec_transaction();

    // 3. VERIFY: Block until the monitor confirms the interrupt was processed.
    // This is an optional step if the user intends to check for an interrupt event ,
    // Otherwise interrupt monitor checks for all of the expected interrupts to be asserted at the End of Simulation.
    m_intr_mon.wait_for_interrupt("Audio_Codec_Done", 1000ns);
    `uvm_info(get_type_name(), "Interrupt verified.", UVM_MEDIUM)
  endtask
endclass
```

Listing 9: Example of a UVM sequence testing a specific interrupt.

A. Supported Verification Scenarios

The monitor and hierarchy model support several interrupt use cases out of the box:

- **Basic Interrupt Assertion:**
  - Test: Triggers an event (e.g., MPU finishes Audio Codec processing).
  - Test: `m_intr_mgr.set_expected_interrupt("CPU_MAIN_IRQ_PIN", "Audio_Codec_Done", 1);`
  - Monitor: Detects the assertion `CPU_MAIN_IRQ_PIN`, reads `MPU_AGG_STATUS_REG`, then `AUDIO_STATUS_REG`, finds the `CODEC_DONE` bit set, recursively processes, and marks the `Audio_Codec_Done` node as asserted.
  - End-of-Sim Check: Verifies that the `Audio_Codec_Done` node was asserted as expected.
- **Interrupt Masking and Unmasking:**

  - Test: Unmasks `MPU_VIDEO_IRQ_N` by writing to a mask register, then triggers a video error.
  - Monitor: Verifies the full interrupt assertion path.
  - Test: Masks `MPU_VIDEO_IRQ_N`, then triggers another video error.
  - Monitor: Verifies that `CPU_MAIN_IRQ_PIN` *does not* assert. If it does, a `UVM_ERROR` is reported for an unexpected interrupt.
- **Concurrent Interrupts:**
  - Test: Triggers multiple events causing `Audio_Buffer_Underrun` and `Video_Frame_Sync_Loss` to assert near-simultaneously.
  - Monitor: The main monitor loop forks parallel processing threads for each top-level interrupt. The recursive ‘process_interrupt_source‘ task naturally handles each branch of the hierarchy in parallel.
- **Unexpected (Spurious) Interrupts:**
  - Test: Does not set any expected interrupts. A glitch in the DUT causes `CPU_MAIN_IRQ_PIN` to assert.
  - Monitor: Immediately reports a `UVM_ERROR` for an *Unexpected interrupt asserted* and then traces down to the deepest unexpected source it can find.
- **Missing Expected Interrupts:**
  - Test: Sets `m_intr_mgr.set_expected_interrupt("CPU_MAIN_IRQ_PIN", "Image_Capture_Timeout", 1);`, but a bug in the DUT prevents the event from ever firing.
  - Monitor: At the ‘check_phase‘ of the simulation, the framework traverses the entire hierarchy and finds that the `Image_Capture_Timeout` node was expected but its ‘m_interrupt_asserted‘ flag is still 0. It reports a `UVM_ERROR` for a ”Missing expected interrupt,” catching a silent failure.
- **Multi-level Hierarchy Tests:**
  - Validate tracing through 4–5 levels of aggregation.

B. Recursive Debug Log Trace

The monitor provides debug logs that recursively trace each interrupt trigger path. These logs are extremely useful for debug and waveform correlation. Below is an example log of a video decoding error:

```
UVM_INFO @100ns [gmon] Interrupt detected at CPU_MAIN_IRQ_PIN
UVM_INFO @100ns [gmon] Tracing: CPU_MAIN_IRQ_PIN -> MPU_VIDEO_IRQ_N
UVM_INFO @100ns [gmon] Tracing: CPU_MAIN_IRQ_PIN -> MPU_VIDEO_IRQ_N -> Video_Decode_Error
UVM_INFO @100ns [gmon] Leaf interrupt asserted: Video_Decode_Error
UVM_INFO @100ns [gmon] Clearing interrupt source: Video_Decode_Error using W1C
```

Fig. 8: Trace log showing recursive detection from top pin to leaf interrupt.

C. End-of-Simulation Assertion Checks

A special mechanism performs an end-of-test validation to report all **expected but unasserted** interrupts. This helps detect silent failures or improperly connected sources.

D. Debug Time Reduction

Thanks to recursive tracing, root cause identification time is drastically reduced. Engineers no longer need to inspect waveforms manually or hardcode register offsets in the monitor.

VIII Results and Effectiveness

To validate the scalability and robustness of the gISR framework, it was applied to a production-scale SoC subsystem that integrated over 150 interrupt sources. These sources originated from both internally developed IPs and third-party cores, organized across 5+ levels of interrupt aggregation.

A. Deployment Metrics

- Over **150 unique interrupt sources** were modeled and verified using less than 300 lines of hierarchical configuration code.
- The interrupt monitor dynamically identified and resolved interrupt propagation paths with zero DUT-specific code changes.
- End-of-simulation analysis reliably caught missing interrupts and stale status bits across all simulations.

B. Bugs Caught During Integration

The following representative issues were caught early in the simulation cycle using gISR's runtime and post-processing checks:

- **Unexpected Interrupt Assertion:** A controller issued a surprise error interrupt during a reset sequence, violating the expected assertion conditions. gISR flagged the event as unexpected and traced it back to a third-party IP behavior.
- **Unconnected Interrupt Node:** Certain memory-related interrupt lines were left unconnected in the RTL netlist. These were caught due to their persistent unasserted state despite being marked `expected = 1`.
- **Incorrect Interrupt Routing:** One interrupt source was mistakenly connected to an unrelated aggregation point, resulting in an incorrect top-level interrupt pin assertion. gISR's tracing clearly identified the mismatch.
- **Interrupt Not Cleared:** An interrupt path feeding through a legacy interface did not properly support register clearing. gISR detected persistent assertion in status bits even after all child interrupts had been serviced.
- **Status Register Attribute Mismatch:** A mismatch between RTL and register description (RDL) files caused certain fields to be incorrectly marked as RW instead of RO. gISR's final status clearing checks flagged these as stale.

C. Debug and Traceability Improvements

Each of the above issues was accompanied by:

- Recursive logs tracing the propagation of interrupts from leaf nodes to top-level pins.
- Error messages pinpointing missing child assertions or violated clearing mechanisms.
- Auto-generated coverage reports showing which expected interrupts were exercised.

These results demonstrate gISR's effectiveness not just as a monitoring tool, but as an early-stage debug accelerator, reducing manual waveform triage and significantly boosting confidence in interrupt subsystem correctness.

D. Feature-wise Comparative Analysis

Table I summarizes the advantages of the gISR framework over conventional interrupt verification methods, including flat uvm monitors and custom testbench-based approaches. This comparison highlights gISR's strengths in modularity, automation, and end-to-end completeness.

TABLE I: Comparison of gISR with Other Interrupt Verification Techniques

Feature	Manual Testbench	Flat UVM Monitor	gISR (Proposed)
Hierarchical Interrupt Modeling	✗	**Partial**	✓
Status/Mask Register Decoding	**Partial** Manual	**Partial** Manual	✓Auto-decoded
Recursive Source Tracing	✗	✗	✓
Runtime Ghost Interrupt Detection	✗	✗	✓
Support for Multiple Clear Policies	✗	**Partial**	✓
End-of-Simulation Expected Check	✗	✗	✓
Plug-and-Play Reuse	✗	**Partial**	✓
Config via API (No RTL Change)	✗	**Partial**	✓
UVM Messaging Integration	**Partial** Custom	**Partial** Custom	✓Structured

IX Conclusion

Interrupt verification in complex SoCs presents significant challenges due to deep hierarchies, multiple status/mask/clear interactions, and runtime configurability. Traditional flat, DUT-bound approaches often lack flexibility, leading to duplication, limited reuse, and delayed debug.

The proposed **generic Interrupt Service Routine (gISR)** framework offers a scalable, reusable solution by abstracting interrupt definitions as hierarchical class-based models and enabling recursive traversal of interrupt trees. It integrates seamlessly with UVM RAL models, supports runtime status/mask checking, and facilitates expectation-driven validation with minimal coding overhead.
Through deployment across multiple SoC programs, gISR has demonstrated:

- **Testbench simplification**, with interrupt intent modeled in the env and reused across verification levels,
- **Faster bring-up**, due to early visibility into unexpected or missing interrupt conditions,
- **Improved debug productivity**, thanks to rich runtime logging and error traceability.

The architecture encourages reuse across IPs and subsystems, making it an ideal fit for verification teams that work across diverse programs and interrupt schemes. Its modularity, abstraction, and post-simulation introspection capabilities help standardize & streamline interrupt validation across verification flows, ultimately aid in identifying potential issues early in the design cycle,

improving overall design quality and reducing the risk of bugs.

Looking ahead, the gISR approach can serve as a template for other hierarchical protocol monitors, particularly where recursive decoding and multi-level status correlation are required. With further enhancements such as automated tree generation, interrupt coverage, and domain-aware modeling, gISR stands to become a key component in interrupt-aware verification environments.

X Future Scope

While the current implementation of gISR already delivers scalable, reusable, and production-tested interrupt verification infrastructure, several enhancements are being explored to further improve automation, usability, and coverage metrics.

A. Automated Configuration from Spreadsheets or Text Files

Manual configuration of hundreds of interrupts, though simplified by gISR's APIs, can still be error-prone. Future iterations of the framework will support parsing of structured specification files, such as:

- Google Sheets or Excel-based register and hierarchy definitions.
- JSON/YAML-based interrupt map exports from design tools.

This feature will allow automatic instantiation of interrupt nodes, saving engineering effort and eliminating config mismatches.

B. Enhanced Interrupt Coverage Analytics

A dedicated interrupt coverage collector is planned to track:

- Assertion frequency and timing of each interrupt node.
- Whether each expected interrupt fired at least once.
- Hierarchical traversal coverage statistics.

These can be exported in tabular or graphical formats to support verification closure and review dashboards.

C. Pulse and Timing-Aware Validation

The current implementation verifies level- and pulse-based interrupts through assertion/de-assertion tracking. Future support will include:

- Pulse-width validation against user-configured min/max duration.
- Pulse glitch detection for timing-sensitive modules.

D. Support for Counter-Based Interrupts

Some subsystems use counter registers to trigger interrupts after N events. Planned support includes:

- Threshold configuration and violation checks.
- Cross-verification of counter increments vs. interrupt assertions.

E. ML-based Anomaly Detection

Apply machine learning models to post-process interrupt logs and detect subtle behavioral patterns.
Potential anomalies include:

- High-frequency spurious assertions
- Long latency between stimulus and IRQ
- Non-deterministic clearing behavior across simulations

F. Integration with Structural Test Flows

Structural test patterns often rely on notify/test bits in interrupt registers.
gISR will support:

- Notify-bit checking logic
- Signature-based verification of test-mode interrupt servicing

Appendix A
Common API Reference

The gISR framework exposes a modular API surface to configure, control, and traverse the interrupt tree. The APIs are logically grouped into those relevant for individual nodes and those handled by the manager.

A. Node APIs

TABLE II: Node-Level Public APIs in gISR

API Name	Description / Purpose
`add_status_reg(reg)`	Adds a status register to this node.
`add_mask_reg(status_reg, mask_reg)`	Adds a mapping between a status and corresponding mask register.
`set_interrupt_asserted(val)`	Sets internal flag for runtime observation. Used by monitor.
`check_if_expected_interrupt_not_asserted()`	Performs end-of-test validation for expected-but-unseen interrupts.
`print_interrupt_tree(prefix)`	Recursively dumps node and its descendants for debug.
`add_interrupt_child_node(...)`	Creates and attaches a new child node to the current one.

B. Manager APIs

TABLE III: Manager-Level Public APIs in gISR

API Name	Description / Purpose
`create_top_interrupt_node(name, status_reg_q[$])`	Initializes and registers a new top-level interrupt node.
`add_child_interrupt(...)`	Adds a child node to the hierarchy with register info and clear policy.
`get_interrupt_node(string name, top_name)`	Retrieves node handle for querying, update, or debug.
`set_expected_interrupt(string name, bit expected)`	Sets whether a node is expected to fire during this test.
`(set/get)_interrupt_property(...)`	Accessor methods to retrieve or update any node-level metadata.

C. Monitor APIs

TABLE IV: Manager-Level Public APIs in gISR

API Name	Description / Purpose
`set_checker_enable(bit enable)`	**[Control API]** Enables or disables the entire interrupt monitor at runtime.
`wait_for_interrupt_monitor_idle()`	**[Test Sequence API]** A synchronization task that blocks until the monitor has finished processing all currently active interrupts.
`wait_for_interrupt(name, timeout)`	**[Test Sequence API]** Allows a test to block until a specific logical interrupt has been fully detected, traced, and processed.
`wait_for_interrupt_trigger(name, &asserted)`	**[User-Implemented]** Blocks until a physical DUT interrupt pin asserts. Connects the framework to DUT signals.
`read_status_reg(name, )`	**[User-Implemented]** Bridges to the RAL model to read an entire status register.
`write_status_reg_field(name, field, data)`	**[User-Implemented]** Bridges to the RAL model to write to a field, typically for clearing an interrupt source (e.g., W1C).
`get_status_register_reg_fields(name, fields_q)`	**[User-Implemented]** Provides the monitor with a list of all relevant field names within a given status register.
`process_interrupt_source(node)`	The core internal recursive engine that traverses the interrupt tree to identify the root cause of a detected interrupt.
`run_phase() / check_phase()`	Standard UVM phases that implement the main monitor loop and the final end-of-test validation checks for missed interrupts.

These APIs collectively enable a scalable and fully configurable framework for interrupt modeling and runtime introspection, without requiring intrusive RTL modifications.

Appendix B
Sample Monitor Output Log

Below is an excerpt of UVM logs automatically emitted by the gISR monitor for a test involving a two-level interrupt assertion:

```
UVM_INFO gic_monitor.sv(107) @ 1000:
  [INTERRUPT_TRAVERSAL] Starting interrupt descent from CPU_MAIN_IRQ

UVM_INFO gic_monitor.sv(115) @ 1002:
  [STATUS_DECODE] Node 'AUDIO_IRQ' active in status_reg[0x100]: bit 2 set

UVM_INFO gic_monitor.sv(130) @ 1003:
  [LEAF_ASSERTED] Node 'AUDIO_BUFFER_OVERFLOW' is expected and asserted.

UVM_ERROR gic_monitor.sv(145) @ 1003:
  [UNEXPECTED_INTERRUPT] Node 'AUDIO_PARITY_ERROR' was not expected but asserted!

UVM_INFO gic_monitor.sv(200) @ 1050:
  [CLEARING_STATUS] Cleared bits in AUDIO_IRQ and CPU_MAIN_IRQ
```

Listing 10: Runtime UVM log for a test with expected and unexpected interrupts

Appendix C
Example Interrupt Hierarchy Printout

For debug, the manager can dump the complete constructed hierarchy:

```
CPU_MAIN_IRQ
|-- AUDIO_IRQ
|    |-- AUDIO_BUFFER_OVERFLOW
|    |-- AUDIO_PARITY_ERROR
|-- VIDEO_IRQ
     |-- FRAME_DROP
```

Listing 11: Printed Interrupt Tree Hierarchy from gISR

References

[1] Accellera Systems Initiative, "Universal Verification Methodology (UVM) 1.2 User's Guide," 2014. [Online]. Available: https://verificationacademy.com/topics/uvm-universal-verification-methodology/

[2] T. Prasad, S. Damani, "Plug-n-play UVM Environment for Verification of Interrupts in an IP," *Design-Reuse.com*, 2021. [Online]. Available: https://www.design-reuse.com/article/60437-plug-n-play-uvm-environment-for-verification-of-interrupts-in-an-ip/

[3] "How to Handle Interrupt in UVM," *The Art of Verification*, 2020. [Online]. Available: https://theartofverification.com/how-to-handle-interrupt-in-uvm/

[4] S. Vagaggini, R. Ciardi, M. Trafeli, and L. Fanucci, "Development of highly reliable UVM-based Verification Environment for SpaceWire Codec," in *2022 IEEE 9th International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, Pisa, Italy, 2022, pp. 1-6.

[5] A. Dimanic, N. Stevanovic, Y. Furman, I. Henigsberg, "Automate Interrupt Checking with UVM Macros and Python," in *DVCon Europe 2022*, Munich, Germany, 2022. [Online]. Available: https://www.thevtool.com/wp-content/uploads/2023/09/Automate_Interrupt_Checking_with_UVM_Macros_and_Python_DVCon_2022.pdf

[6] C. Rumpler, A.W. Rath, S. Simon, H. Endres "A UVM-based Approach for Rapidly Verifying Digital Interrupt Structures," in , *DVCon United States 2016*. [Online]. Available: https://dvcon-proceedings.org/wp-content/uploads/a-uvm-based-approach-for-rapidly-verifying-digital-interrupt-structures.pdf

[7] A. Tutov, A. Kamkin, and A. Smolov, "Difficulties in UVM Interrupt Handling, Especially in Multi-Processor Systems," in *2020 Ivannikov Ispras Open Conference (ISPRAS)*, Moscow, Russia, 2020, pp. 58-64.

[8] H. Fagih, "A Recipe for Verification IP - The Role of Methodology," *Design-Reuse.com*, 2020.

[9] D. Borza and L. Cucu, "Generic System Verilog Universal Verification Methodology based Reusable Verification Environment," *arXiv preprint arXiv:1301.2858*, 2013.