bos
SEMICONDUCTORS

Unified Test Vector Across Multiple Testbench Levels

Date: Dec 9, 2025

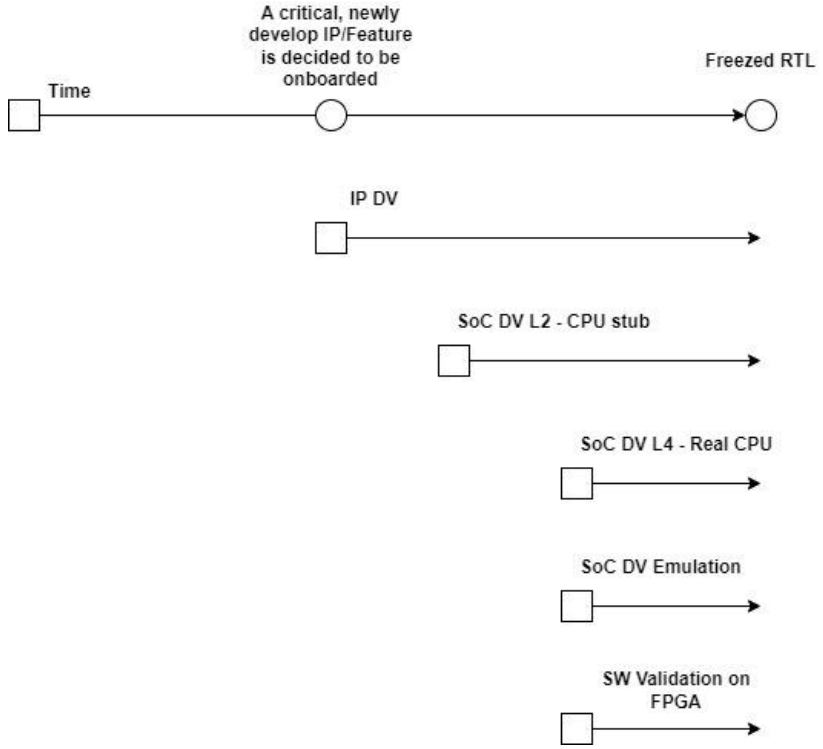
Content

1. Motivation
2. Leverage vector reusability with HOST ENV | CPU_ADAPTER

1. Motivation

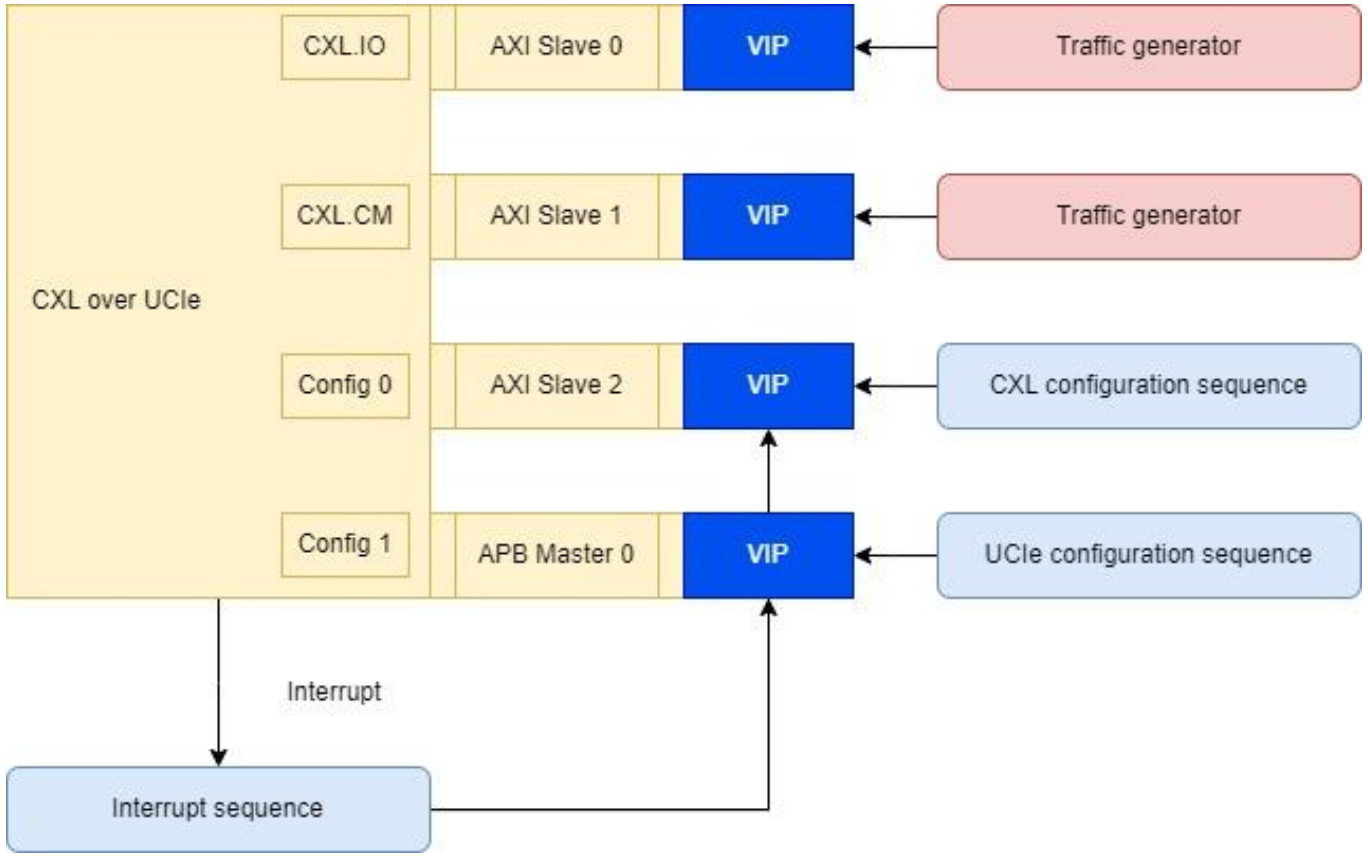
Motivation

- New IP added to the scope with an aggressive timeline; multiple testbench levels, emulation, and FPGA validation are required.
- Different test code between IP/SoC and SW teams for initialization flow and interrupt handling leads to duplicated workload.
- Multiple testbench configuration at SoC level: CPU Stub/ Real CPU



Traditional method

- IP-level testbench requires controlling multiple interface ports, resulting in multiple VIPs and different sequences.

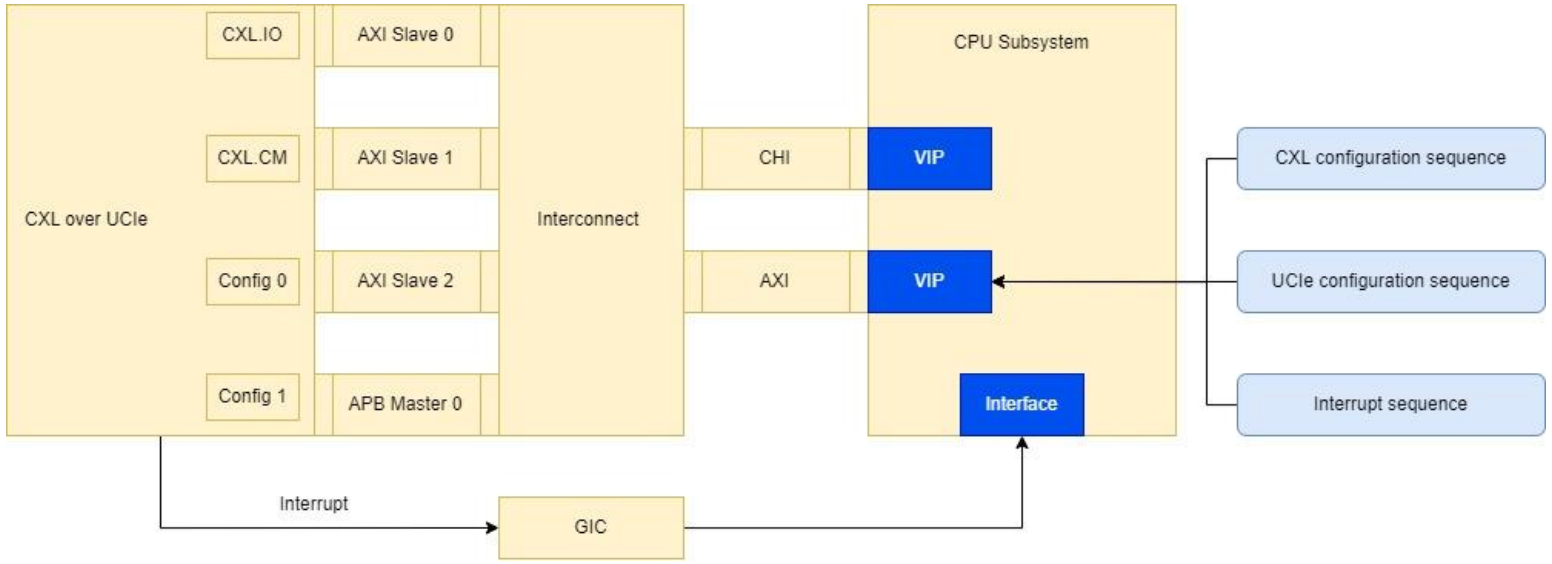


Traditional method

- L2 SoC-level testbench (CPUSTUB) merges all configuration sequences into one. Reusability is possible but requires the IP team to provide a proper, reusable environment and test vectors.

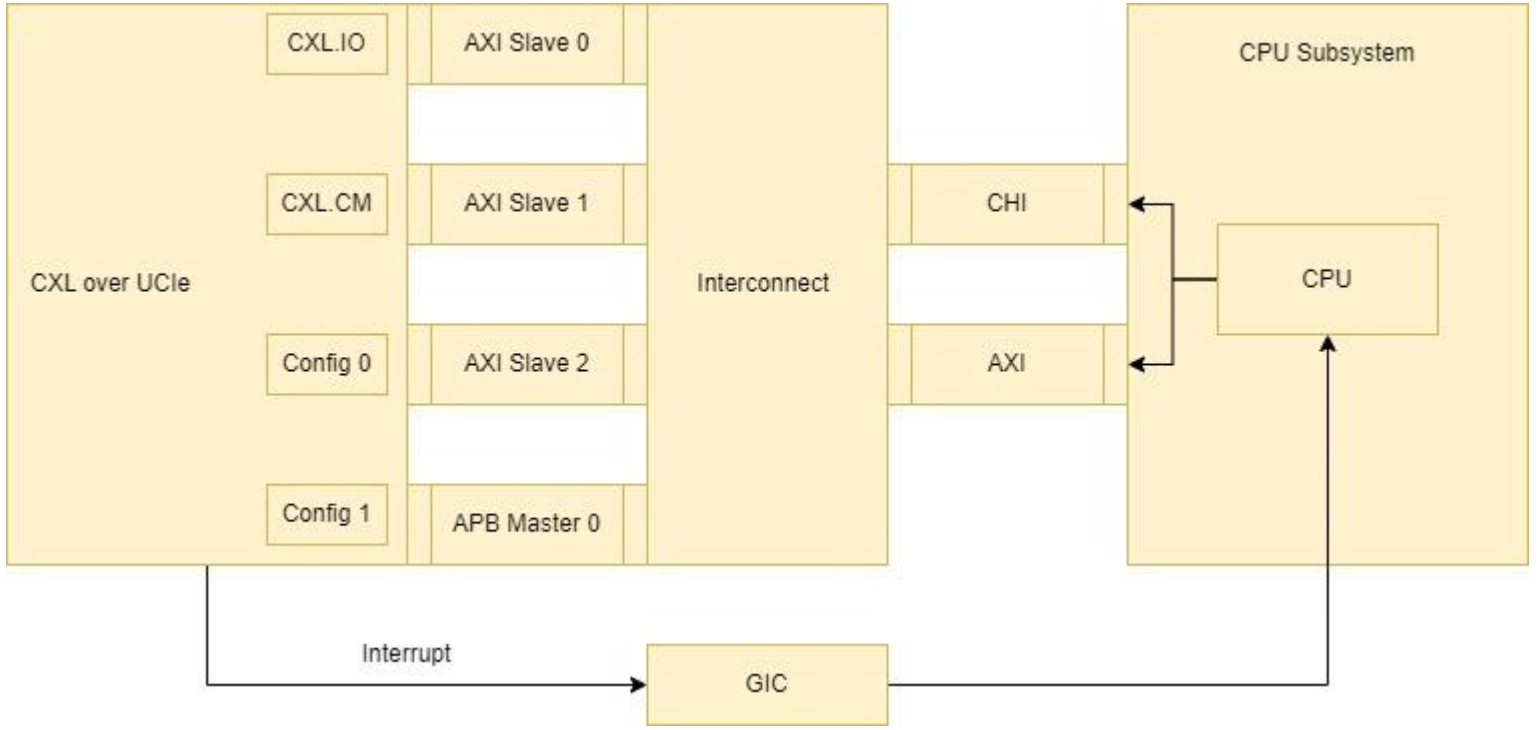
> **Caution** The [ ] is not designed to be a portable verification environment for SoC level testing. Porting the [ ] files to use as a verification component in SoC level testing or updating it to use with other controller versions requires significant [ ] changes and deep understanding of the [ ] and the controller.

- Requires UVM expertise and SoC readiness to adopt the IP environment, or significant rework is unavoidable.



Traditional method

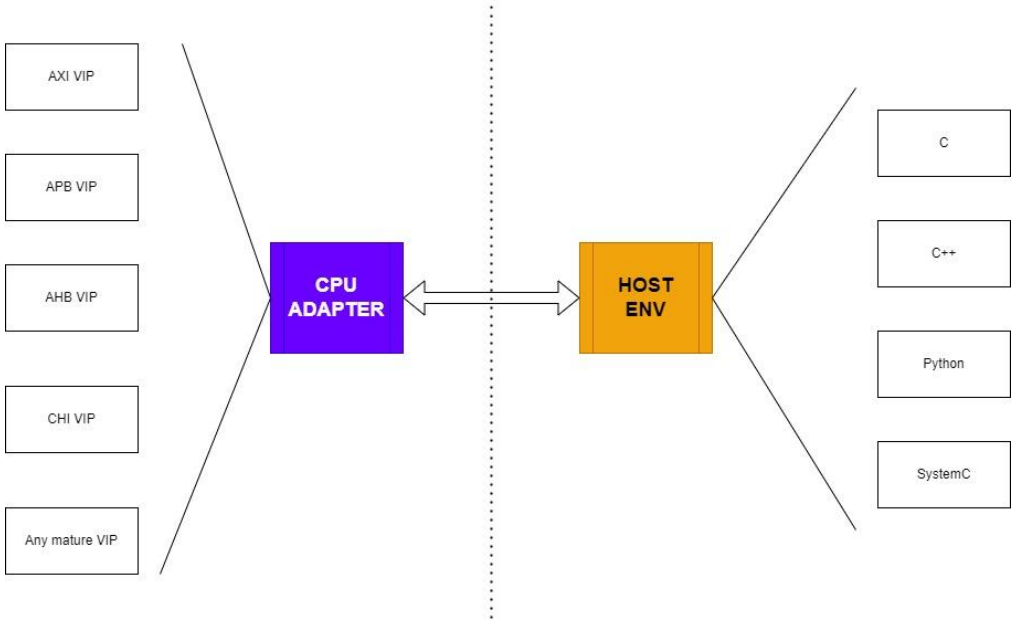
- L4 SoC-level (Real CPU) testbench, Emulation, and FPGA validation become more challenging as real CPU execution is required.



2. Leverage vector reusability with HOST ENV | CPU_ADAPTER

HOST ENV | CPU ADAPTER

- **BVM package on top of UVM:**
  - Provides common APIs, handshakes, and methods.
- **Key components:**
  - CPU_ADAPTER: Bypasses VIP-specific sequences and protocol differences.
  - HOST_ENV: Acts as a hub to connect the UVM testbench with test vectors/models written in other languages.



SYSTEM VERILOG

```
m_cpu["DS"].cpu_write32(`BOS_UCIE_CORE_APP_BASE_ADDR, 'h260);
```

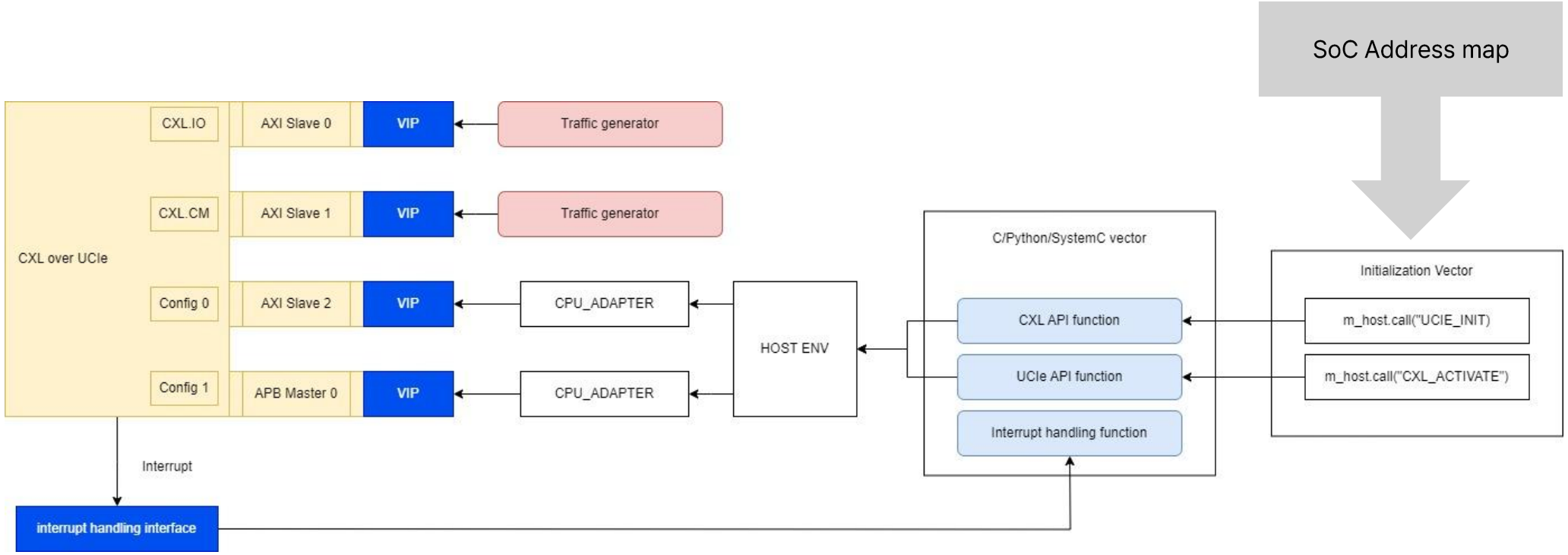
C

```
reg_read(ucie_info_map[ucie_idx].base_addr + 0x0190, &data, host_idx);
data = EXTRACT_BITS(data, 5, 0);
```

No VIP dependency with above code. Can be reuse for any testbench level or project.

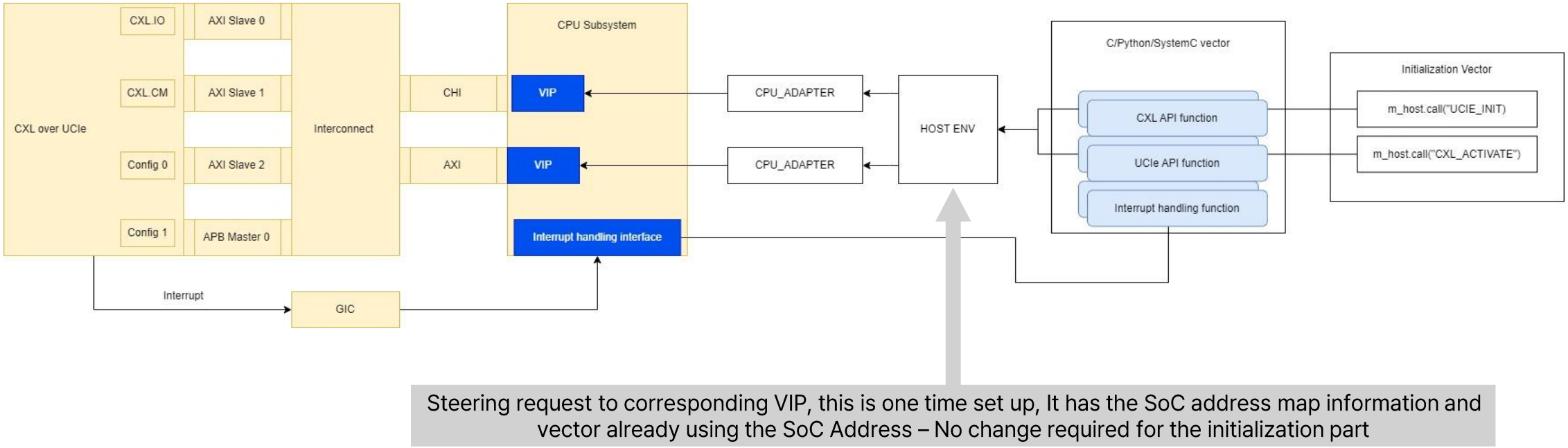
How is IP level bringing up?

- Unified test vectors across IP/SoC: all setup sequences and interrupt handling are written as protocol-independent API functions, using the same SoC address map for IP and SoC, with no test code change needed at the SoC level.
- No reconfiguration, setup, or porting of the IP environment package is required in SoC Level.



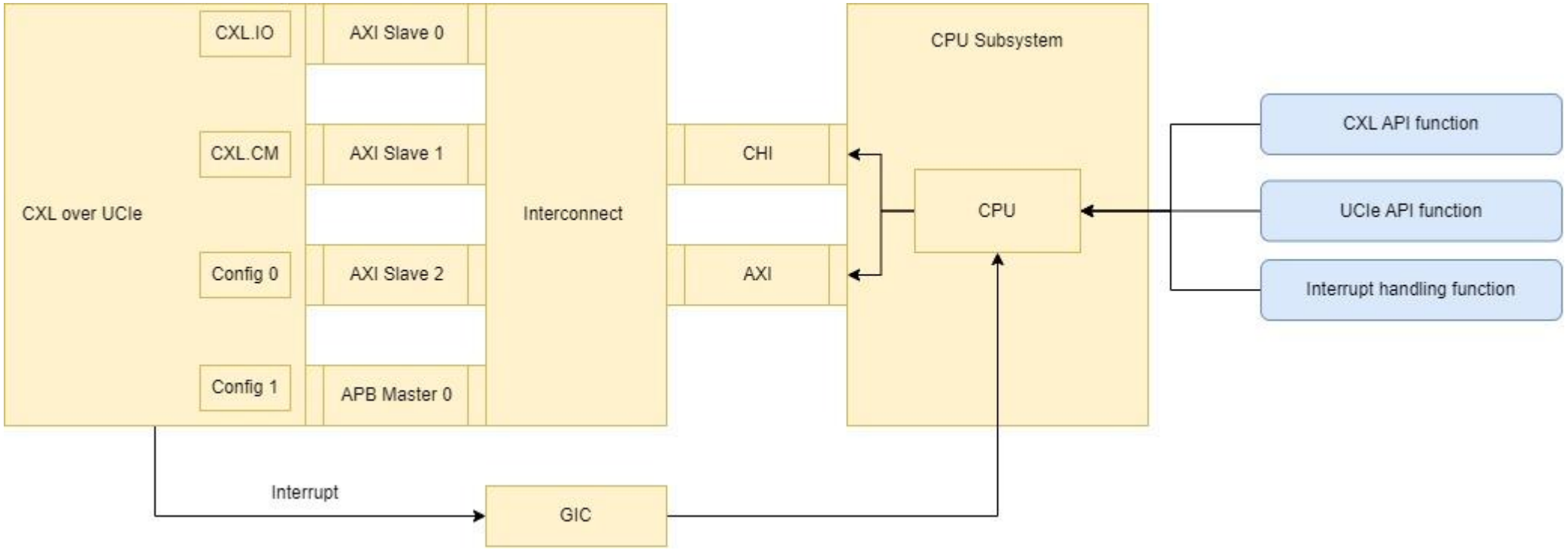
How does it help the SoC – L2?

- Zero compilation errors—no more 'null object/component' issues or missing interface problems
- Just bring the **C**/Python/etc library into the SoC testbench, connect the host env into the VIP
- After initial set up, all IP can be brought up into the SoC with the same flow, zero environment set up required.



How does it help the SoC – L4 + Emulation?

- CPU runs using C code, and the library is already in C. This approach has been proven to work in L4 and Emulation environments.



Example – IP Level set up

- CPU Adapter set up:

```
//CPU Adapter
bvm_cpu_adapter          m_cpu[string];
```

```
m_cpu["HOST_AXI0"].set_sequencer(.sequencer(m_host_axi_vip["HOST_AXI0"].master.sequencer), .adapter(m_host_axi_adapter["HOST_AXI0"]));
m_cpu["HOST_AXI1"].set_sequencer(.sequencer(m_host_axi_vip["HOST_AXI1"].master.sequencer), .adapter(m_host_axi_adapter["HOST_AXI1"]));
m_cpu["HOST_APB0"].set_sequencer(.sequencer(m_host_apb_vip["HOST_APB0"].master.sequencer), .adapter(m_host_apb_adapter["HOST_APB0"]));
```

- HOST set up:

```
//HOST
bvm_host_c                m_host[`NUM_HOST];
```

```
m_host[`LOCAL_DIE_IDX].add_master("HOST_AXI0", m_cpu["HOST_AXI0"]);
m_host[`LOCAL_DIE_IDX].add_master("HOST_AXI1", m_cpu["HOST_AXI1"]);
m_host[`LOCAL_DIE_IDX].add_master("HOST_APB0", m_cpu["HOST_APB0"]);
```

- Register API function:

```
void dummy_func(uint32_t host_idx) {
  uint32_t rdata = 0;

  HOSTC_DEBUG(__func__, "Start the function");
  reg_read(DUMMY_ADDR, &rdata, host_idx);
  rdata = SET_BITS(rdata, 7, 4, trace_type);
  rdata = SET_BITS(rdata, 0, 0, 1);
  reg_write(DUMMY_ADDR, rdata, host_idx);
}
REGISTER_FUNC("dummy_func", dummy_func)
```

- Invoke the function in the test, seq:

```
m_env.m_host[`LOCAL_DIE_IDX].call("dummy_func");
```

Example – SoC level set up

- CPU Adapter set up:

```
//HOST
bvm_host_c                  m_host[`NUM_HOST];
```

```
m_cpu["HOST_AXI0"].set_sequencer(.sequencer(m_host_axi_vip["HOST_AXI0"].master.sequencer), .adapter(m_host_axi_adapter["HOST_AXI0"]));
```

- HOST set up: address map information is required

```
//HOST-C
bvm_host_c                  m_host[`NUM_HOSTC];
```

```
m_host[`LOCAL_DIE_IDX].add_master("HOST_AXI0", m_cpu["HOST_AXI0"]);
```

- Move the C library from IP into SoC:
- Invoke the function in the test, seq:

```
m_env.m_host[`LOCAL_DIE_IDX].call("dummy_func");
```

~ Zero compilation errors
One-time HOST/CPU adapter setup for all IPs
Direct reuse of IP-level API functions

bos
SEMICONDUCTORS

Thank You