# My Testbench Used to Break! Now it Bends (Adapting to Changing Design Configurations)

Jeff Vance
Jeff Montesano
Kevin Vasconcellos
Kevin Johnston

## Agenda

- Introduction
- Testbench Hierarchy Strategy
- Interface Techniques Used
- UVM Environment Topology Generation
- Examples and Applications
- Conclusion

## Introduction: Problem

Upkeep of UVM Testbenches is non-trivial

- – Design changes
- – Multiple design versions
- – Stubbed modules
- – Changing verification scope
- – Migrating to new projects

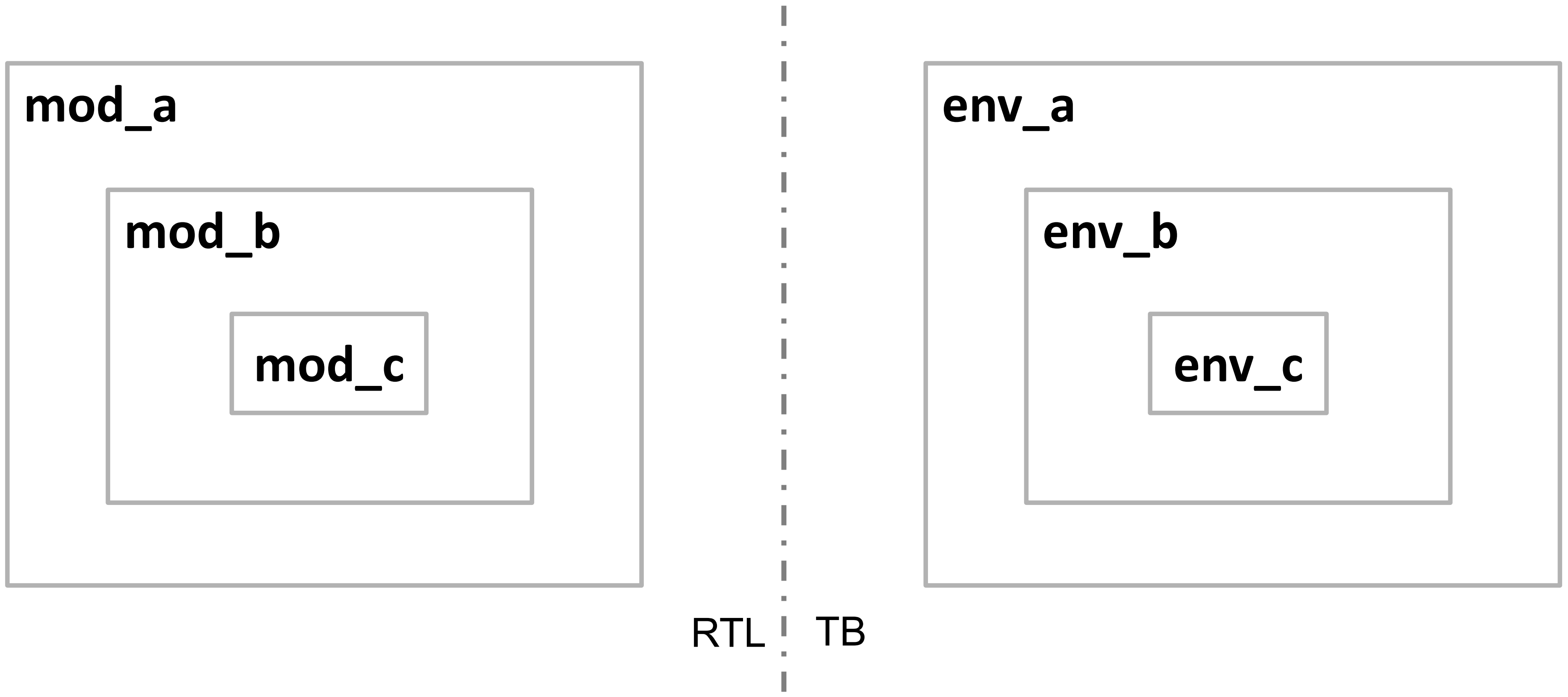
## Introduction: Solution

A single adaptable testbench using these techniques:

- UVM Environment mirrors design modules
- UVM harness with port coercion
- Auto-publishing virtual interfaces

# Mirrored Testbench Hierarchy

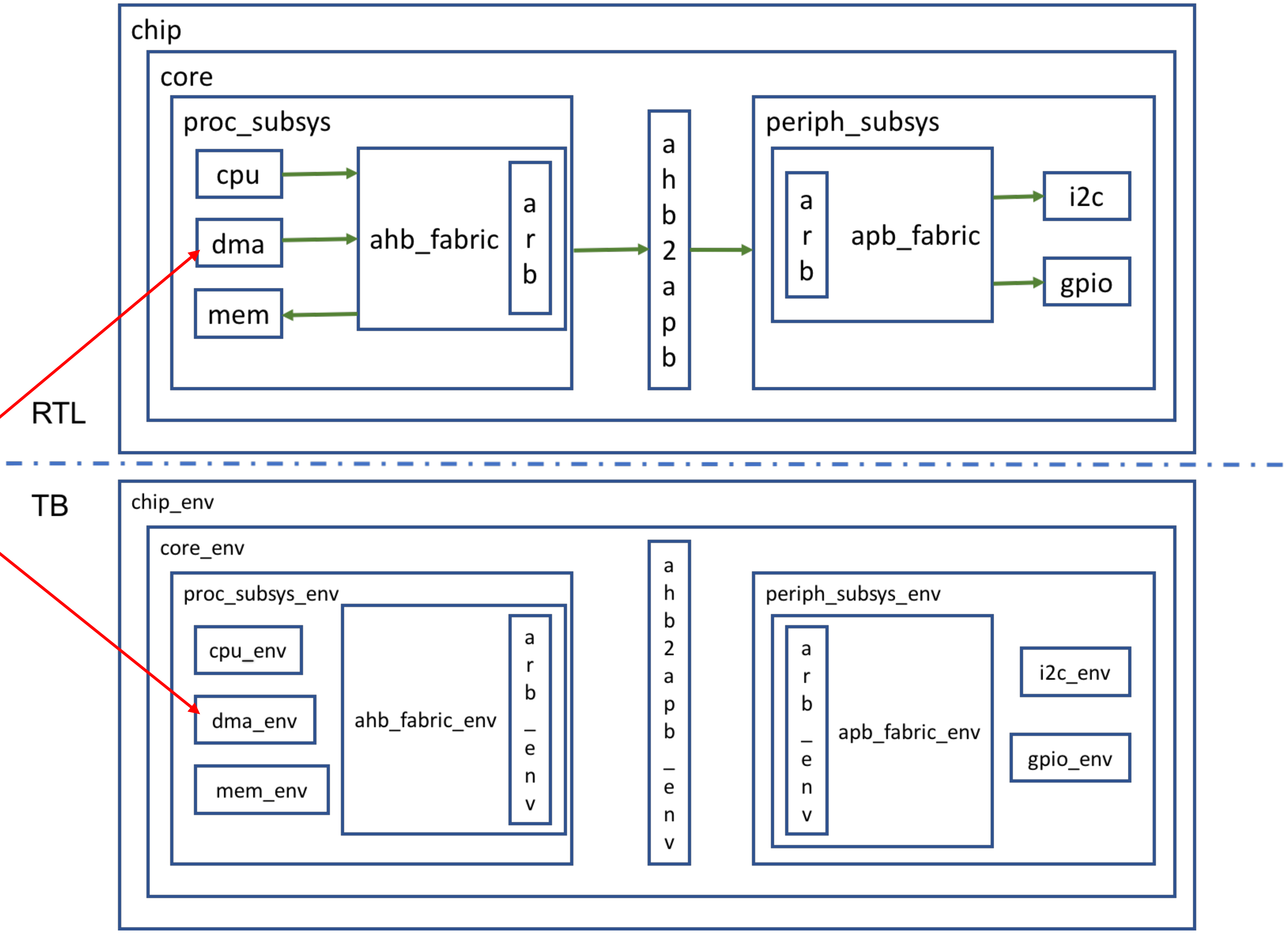
# Testbench Mirrors Design Hierarchy



Only applies to DUT Modules we want to verify

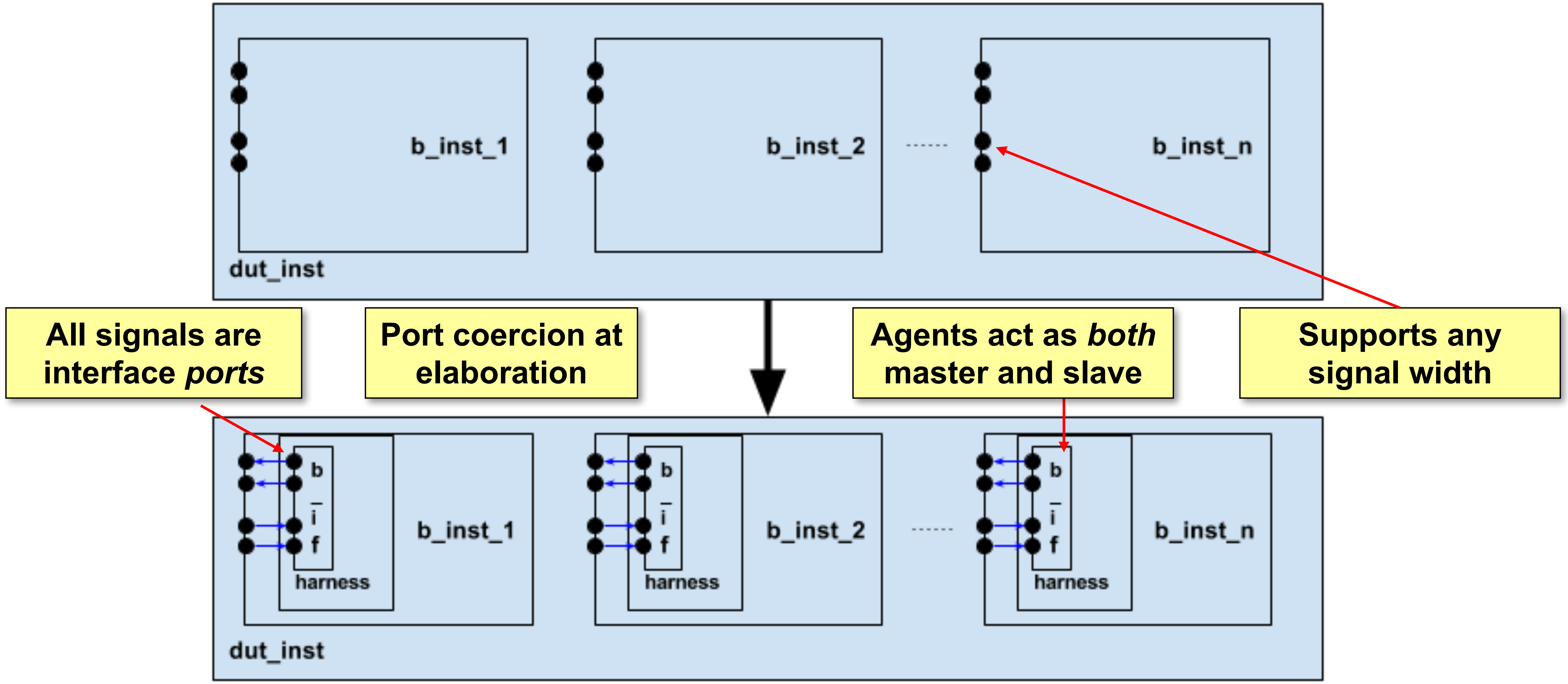
Consistent with standard UVM, but more strict

**Env component names match module instances**



## Interface Techniques Used

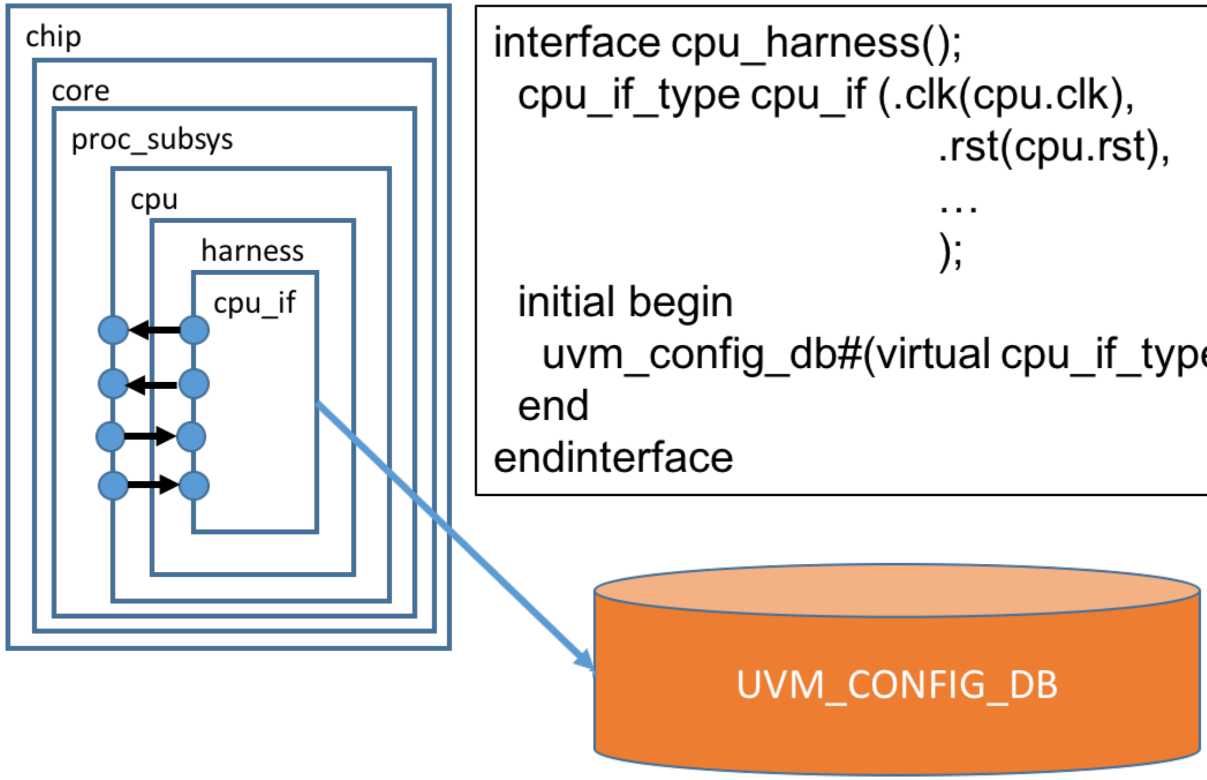
# UVM Harness



b_inst_1
b_inst_2
b_inst_n
dut_inst
All signals are interface *ports*
Port coercion at elaboration
Agents act as *both* master and slave
Supports any signal width
b_if
harness
b_inst_1
b_inst_2
b_inst_n
dut_inst

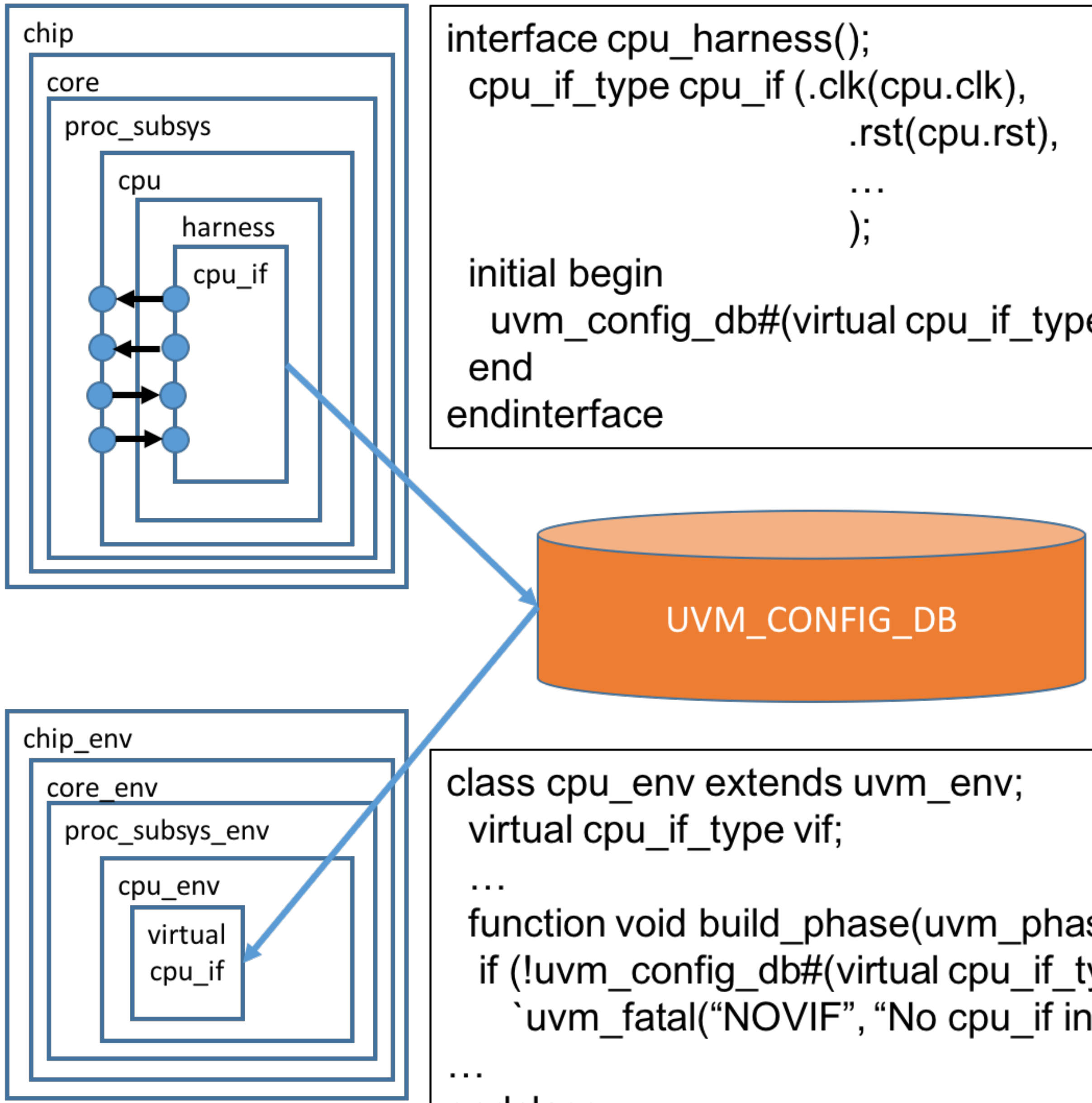
# Interface Auto-Publication

Publish to UVM configuration database using **%m**



```
interface cpu_harness();
  cpu_if_type cpu_if (.clk(cpu.clk),
                                       .rst(cpu.rst),
                                       …
                                       );
  initial begin
    uvm_config_db#(virtual cpu_if_type)::set(null, $sformatf("%m"), "cpu_if", cpu_if)
  end
endinterface
```

**Use %m to define scope for set()**



```
interface cpu_harness();
  cpu_if_type cpu_if (.clk(cpu.clk),
                                    .rst(cpu.rst),
                                    …
                                    );
  initial begin
    uvm_config_db#(virtual cpu_if_type)::set(null, $sformatf("%m"), "cpu_if", cpu_if)
  end
endinterface
```

**Scope for get() will be identical to that of set()**

```
class cpu_env extends uvm_env;
  virtual cpu_if_type vif;
  …
  function void build_phase(uvm_phase phase);
   if (!uvm_config_db#(virtual cpu_if_type)::get(this, "", "cpu_if", vif))
      `uvm_fatal("NOVIF", "No cpu_if in uvm_config_db")
…
endclass
```

## UVM Environment Topology Generation

# UVM Environment Topology Generation

- Each test configures the environment hierarchy

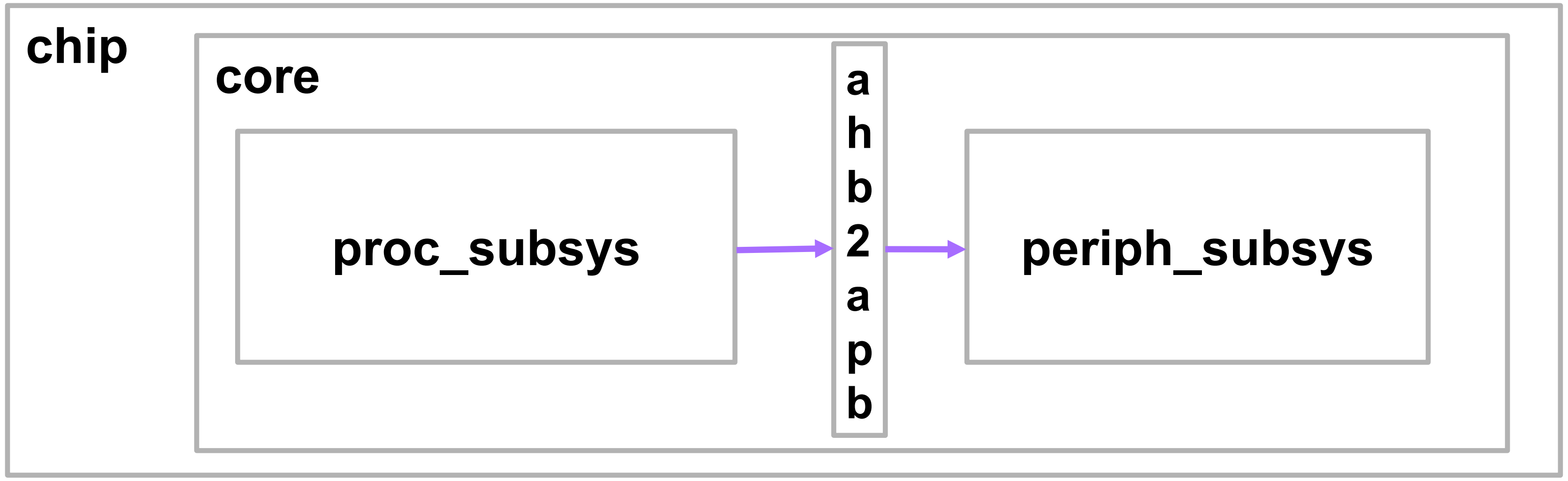
- Which env to build depends on:
  - DUT configuration
  - Presence/absence of stub modules
  - Verification objectives for that test

- Each env has a config object
- Base test builds top environment
- Extended tests build sub-environments

**Each test can define a different hierarchy!**

**Next Steps:**
1) **Environments**
2) **Config object**
3) **Base test**
4) **Extended tests**
5) **Env build phase**

# Step 1: Environments



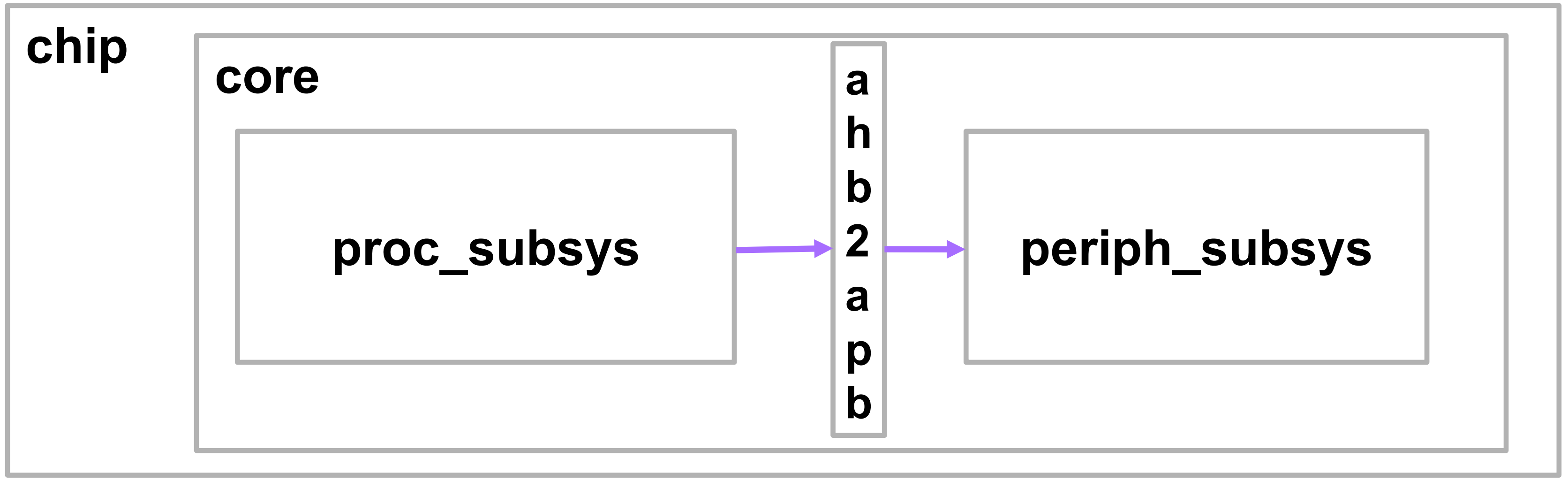
**Arrays of sub-envs**

**Array index is *module* instance name**

**Repeat for all modules**

```
class verilab_core_env extends uvm_env;
  proc_subsys_env     proc_subsys_envs   [string];
  periph_subsys_env   periph_subsys_envs[string];
  ahb2apb_env         ahb2apb_envs       [string];

  verilab_core_env_cfg           cfg;
  verilab_core_pharness_base     harness;
  verilab_core_vseqr             vseqr;
```

# Step 2: Configuration Objects



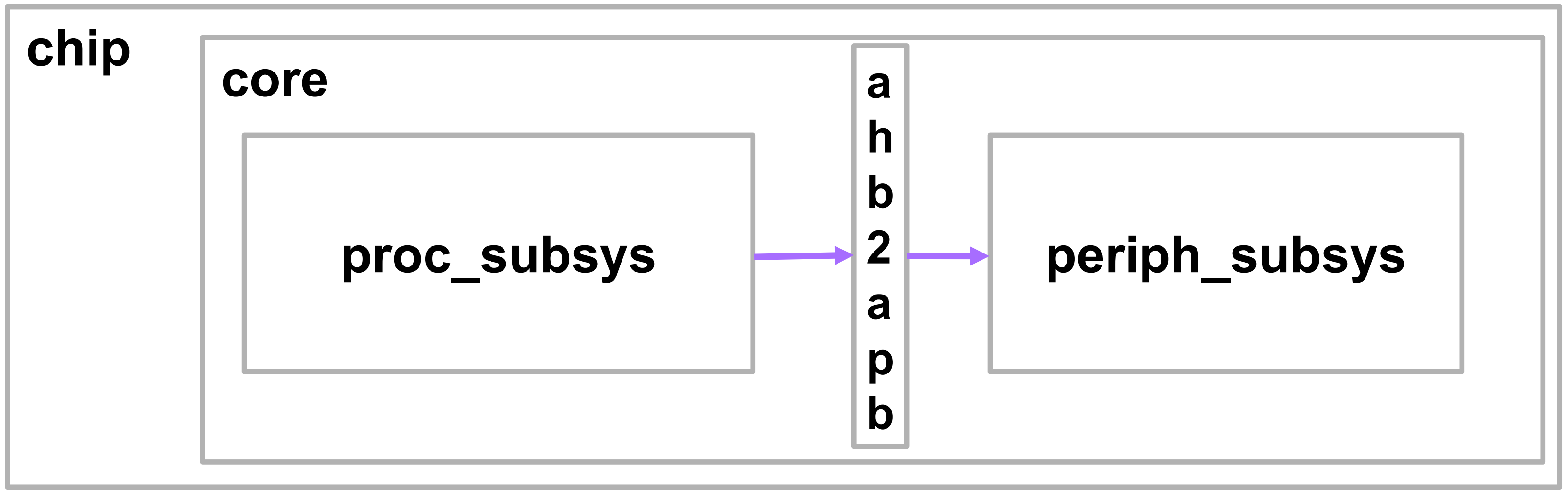
**Arrays of sub-cfgs**

**String index is *module* instance name**

**Repeat for all modules**

```
class verilab_core_env_cfg extends uvm_object;
  proc_subsys_env_cfg    proc_subsys_env_cfgs   [string];
  periph_subsys_env_cfg  periph_subsys_env_cfgs[string];
  ahb2apb_env_cfg        ahb2apb_env_cfgs       [string];
  ...
endclass
```

# Step 3:  Base Test

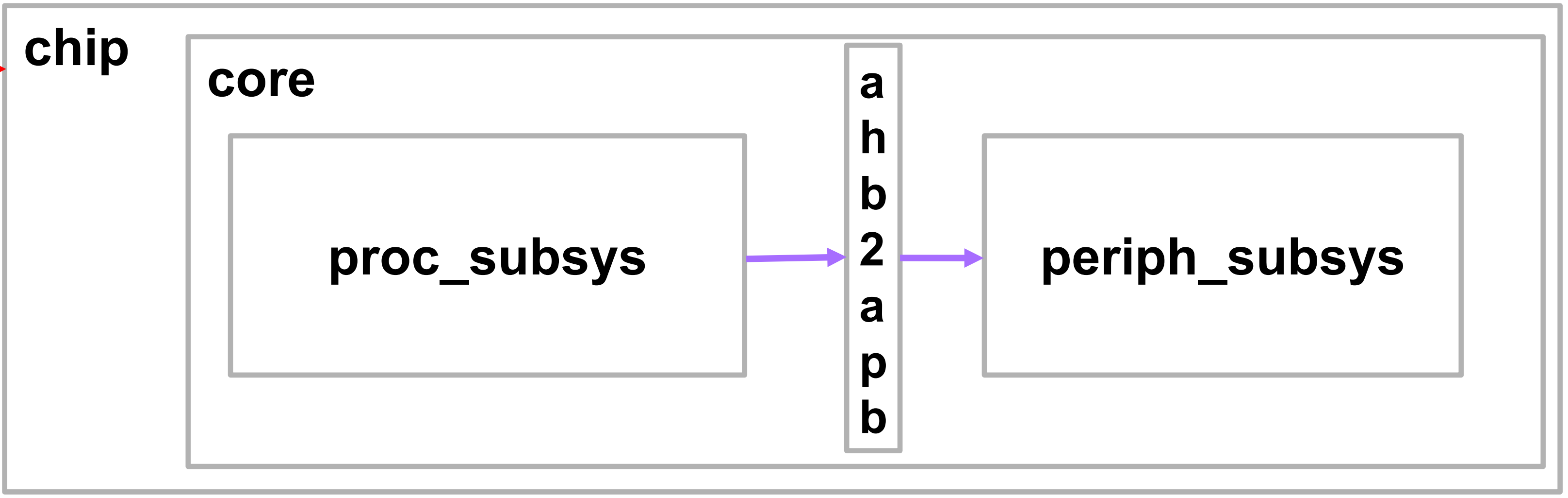


Has the *top* config object and env

```
class verilab_chip_base_test extends uvm_test;
  verilab_chip_env_cfg   verilab_chip_env_cfgs[string];
  verilab_chip_env       verilab_chip_envs    [string];
  …
 virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);

   verilab_chip_env_cfgs["dut"] = verilab_chip_env_cfg::type_id::create("dut");
   verilab_chip_envs["dut"] = verilab_chip_env::type_id::create("dut", this);
```

# Base Test Creates Top Environment

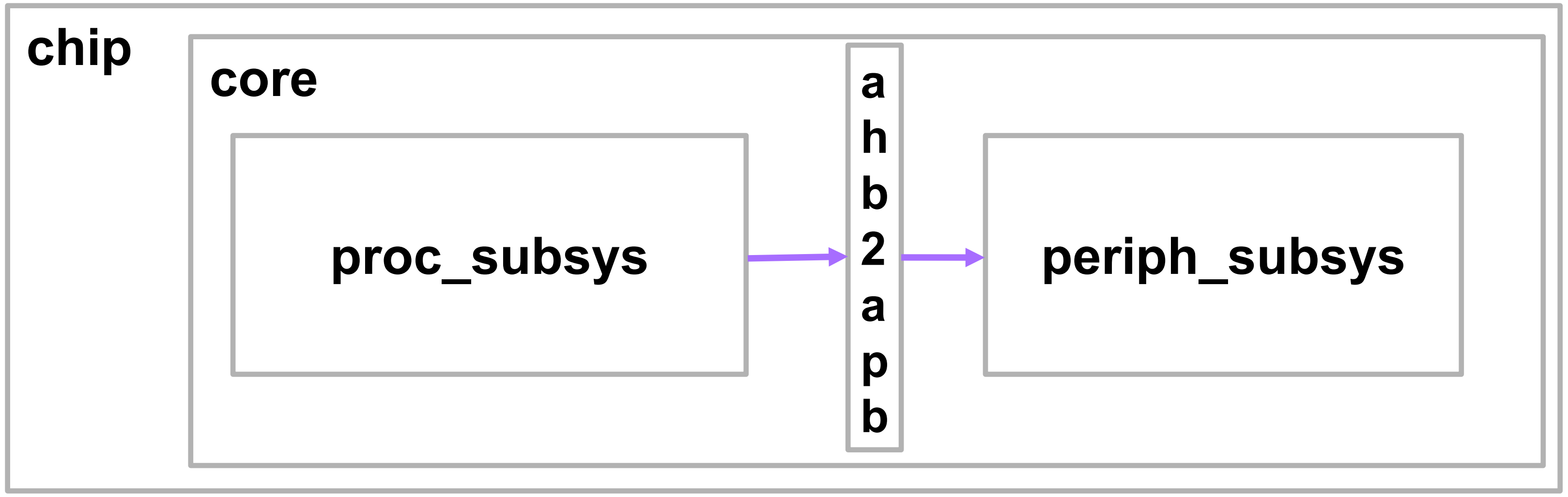
Instance name = “dut”



Index, component, and instance names match

```
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    verilab_chip_env_cfgs[“dut”] = verilab_chip_env_cfg::type_id::create(“dut”);
    verilab_chip_envs[“dut”] = verilab_chip_env::type_id::create(“dut”, this);

    verilab_chip_envs[“dut”].cfg = verilab_chip_env_cfgs[“dut”];
```

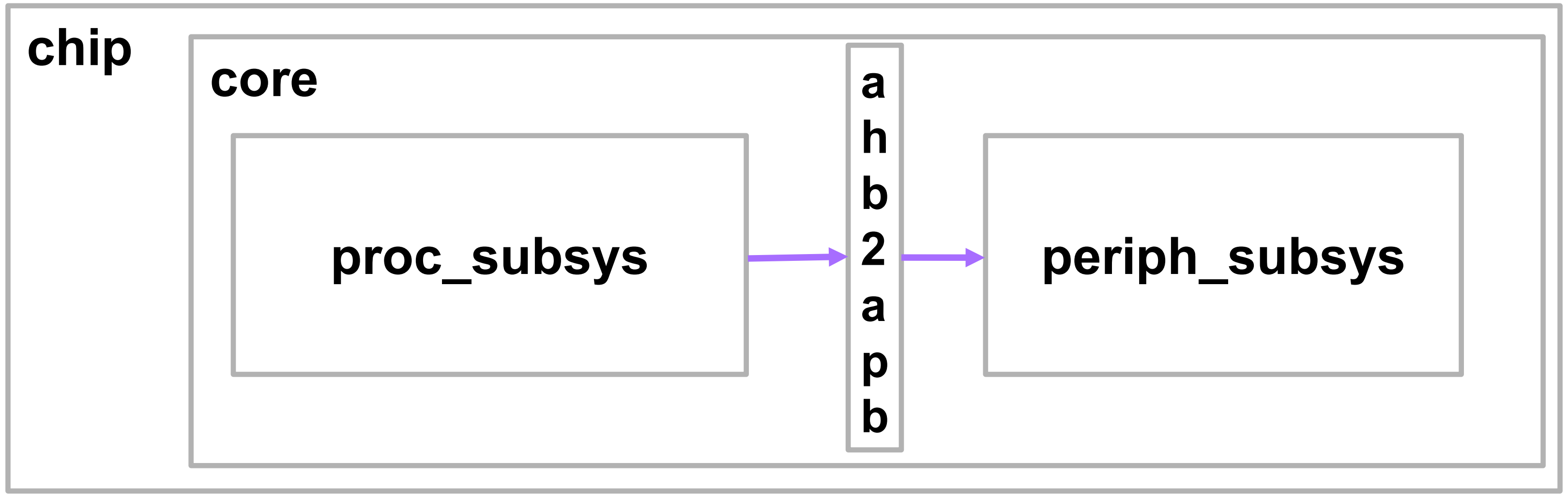
# Step 4: Extended Test



```
class verilab_core_test extends verilab_chip_base_test;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    verilab_chip_env_cfgs["dut"].verilab_core_env_cfgs["verilab_core_0"]
                             =  verilab_core_env_cfg::type_id::create("verilab_core_0");
```

Creates "chip_env_cfg"

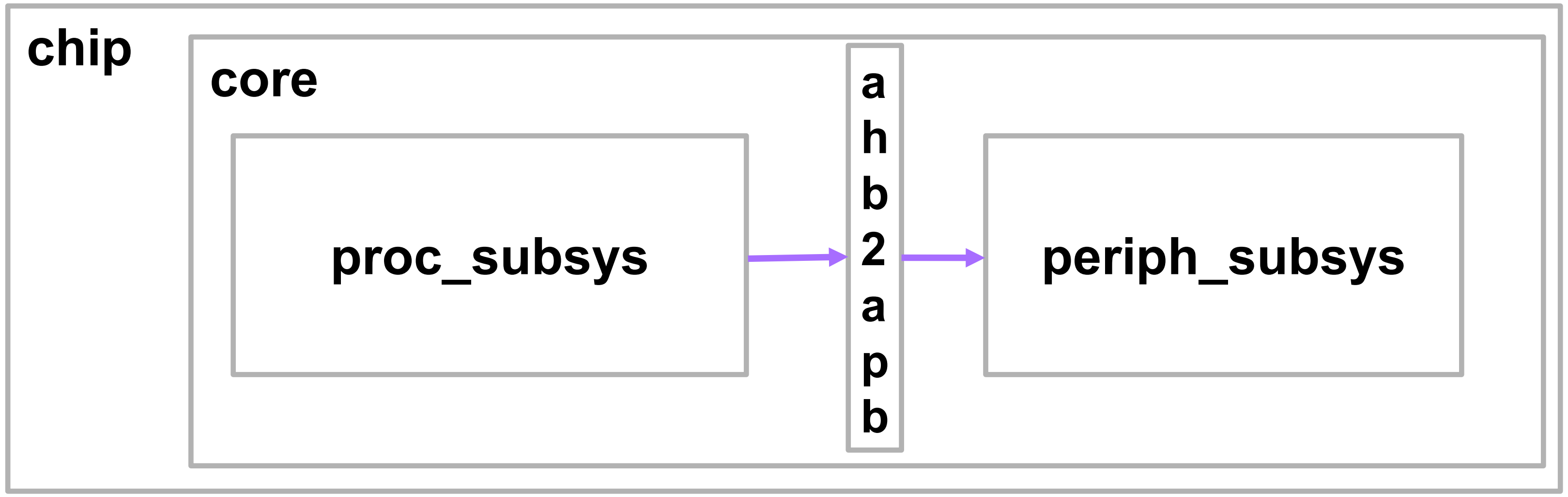
# Create Lower Configuration Objects



```
verilab_chip_env_cfgs[“dut”].verilab_core_env_cfgs [“verilab_core_0”]
                             .proc_subsys_env_cfgs   [“proc_subsys_0” ]
                           =  proc_subsys_env_cfg::type_id::create(“proc_subsys_0”);

verilab_chip_env_cfgs[“dut”].verilab_core_env_cfgs  [“verilab_core_0” ]
                             .periph_subsys_env_cfgs [“periph_subsys_0”]
                        =  periph_subsys_env_cfg::type_id::create(“periph_subsys_0”);
```

# Step 5: Sub-Environments

Arrays of agents

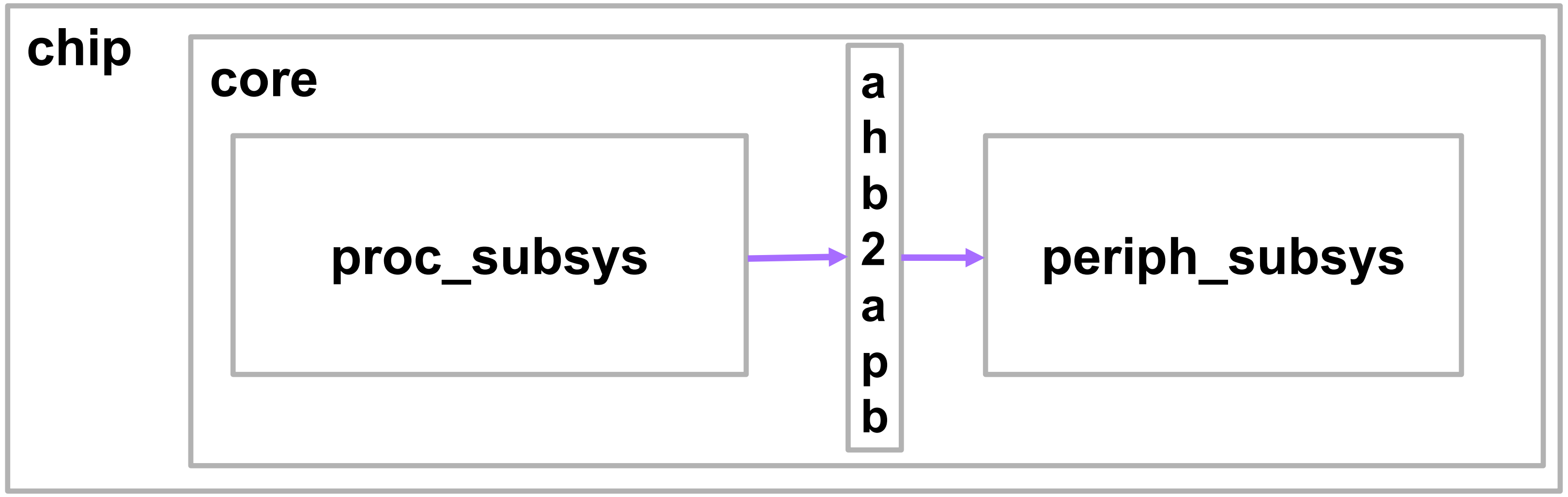


```
class verilab_core_env extends uvm_env;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    clk_rst_agents["clk_rst_if"] = clk_rst_agent::type_id::create("clk_rst_if", this);
    gpio_agents   ["gpio_if"   ] = gpio_agent::type_id::create   ("gpio_if",    this);
    i2c_agents    ["i2c_if"    ] = i2c_agent::type_id::create    ("i2c_if",     this);

    foreach (cfg.proc_subsys_env_cfgs[s]) begin
    proc_subsys_envs[s]       = proc_subsys_env::type_id::create(s, this);
```

# Environment Creates Sub-Envs

Foreach loops create all configured envs

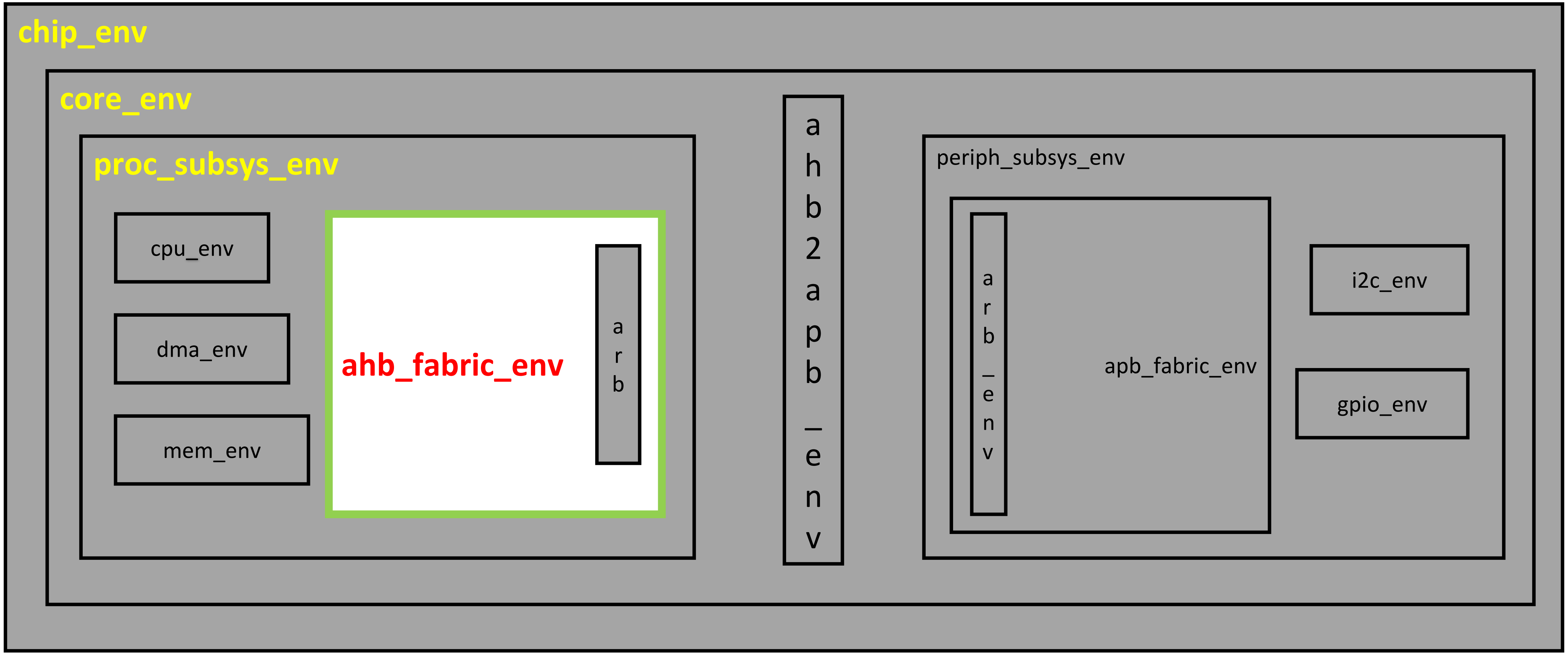
Lowest env has empty array



```
class verilab_core_env extends uvm_env;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    clk_rst_agents["clk_rst_if"] =clk_rst_agent::type_id::create("clk_rst_if", this);
    gpio_agents["gpio_if"]       = gpio_agent::type_id::create("gpio_if", this);

    foreach (cfg.proc_subsys_env_cfgs[s]) begin
      proc_subsys_envs[s]     = proc_subsys_env::type_id::create(s, this);
      proc_subsys_envs[s].cfg = cfg.proc_subsys_env_cfgs[s];
    end

    // ... (foreach loops for other sub-environments)
```

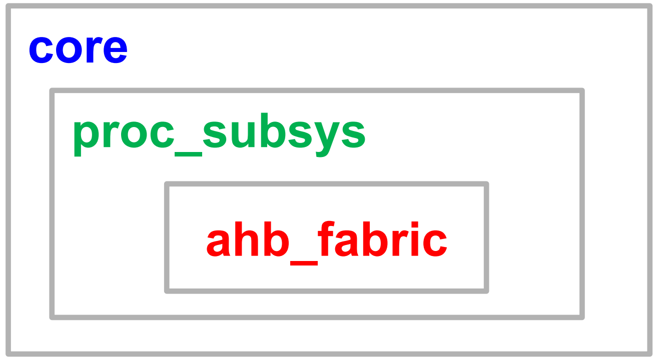
## Examples and Applications

# Block-Level Verification



# Extended Test: build_phase

## Block-level Verification

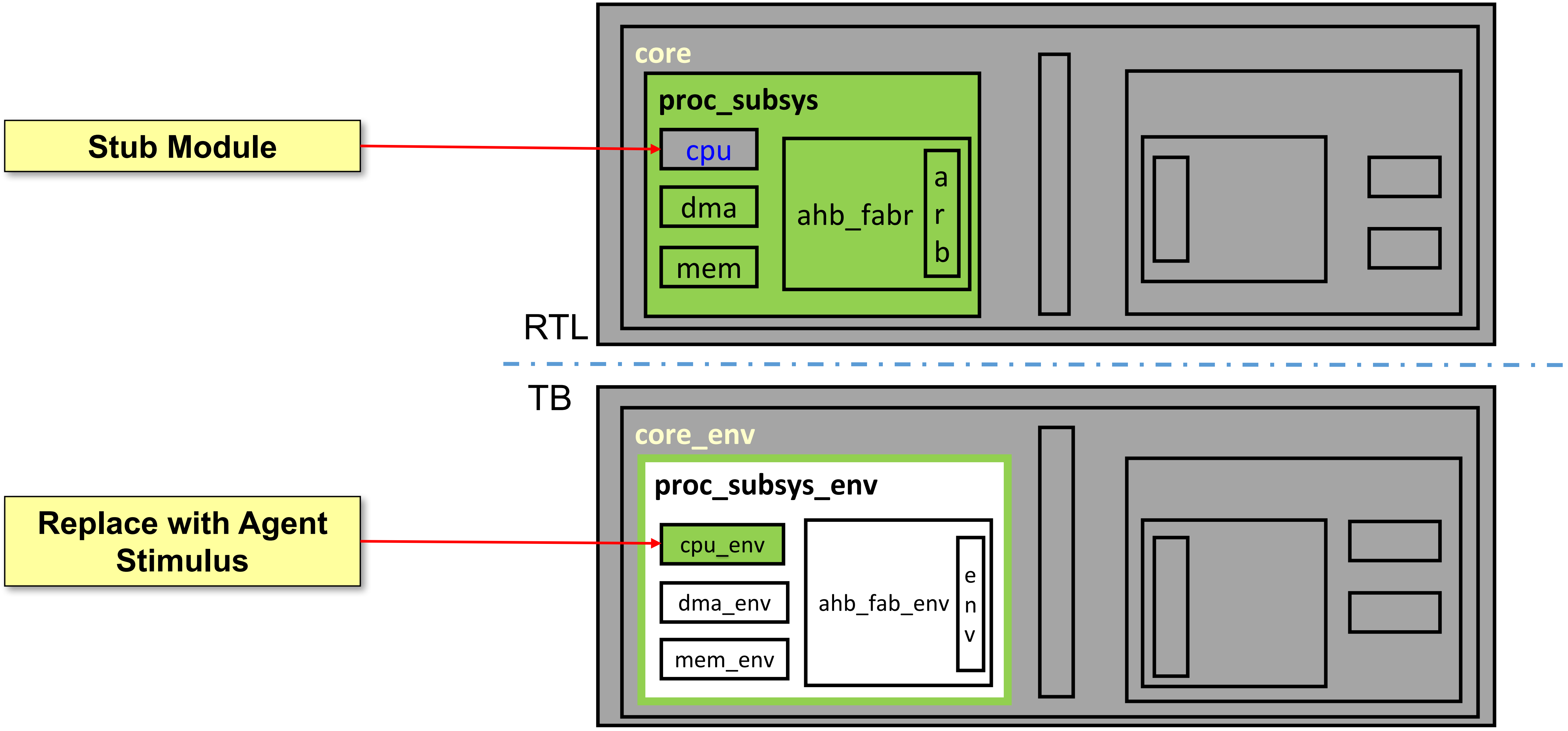


Test creates entire hierarchy of config objects

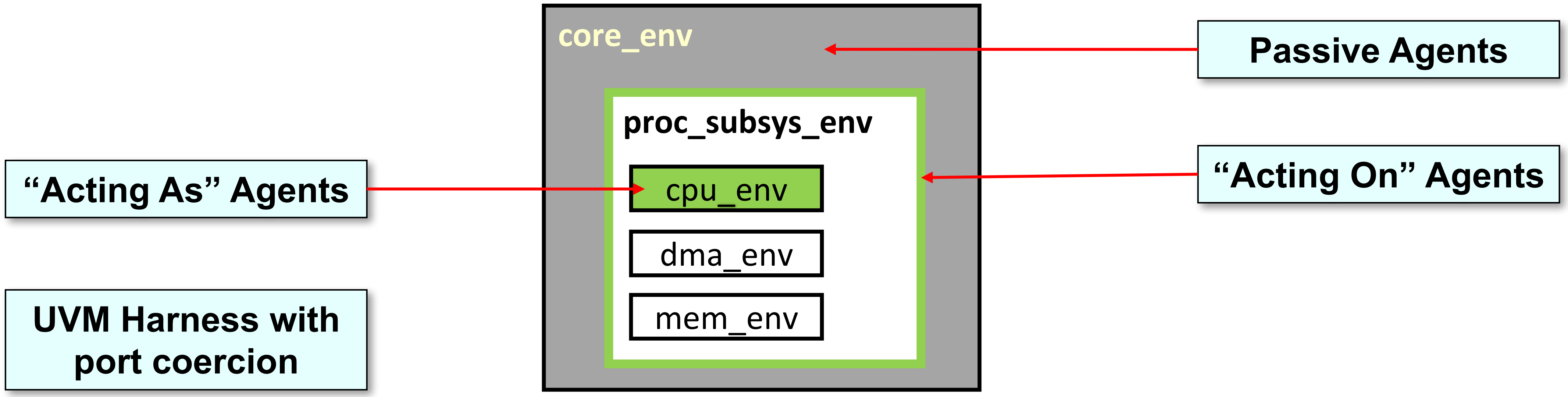
```
verilab_chip_env_cfgs["dut"].verilab_core_env_cfgs ["verilab_core_0"]
                          =  verilab_core_env_cfg::type_id::create("verilab_core_0");

verilab_chip_env_cfgs["dut"].verilab_core_env_cfgs ["verilab_core_0"]
                            .proc_subsys_env_cfgs  ["proc_subsys_0" ]
                          =  proc_subsys_env_cfg::type_id::create("proc_subsys_0");

verilab_chip_env_cfgs["dut"].verilab_core_env_cfgs ["verilab_core_0"]
                            .proc_subsys_env_cfgs  ["proc_subsys_0" ]
                            .ahb_fabric_env_cfgs   ["ahb_fabric_0"  ]
                          =  ahb_fabric_env_cfg::type_id::create("ahb_fabric_0");
```

Sequence started on block-level sequencer

## System-level Verification



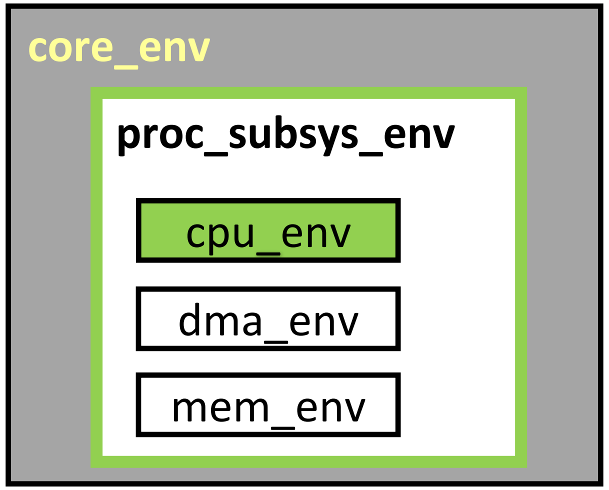
# Environment Roles



- **Env Role:** a combination of master / slave agents
- Env is ***acting as*** the stubbed module
- Env verifying a module is ***acting on*** that module

# Set Environment Roles

## System-level Verification



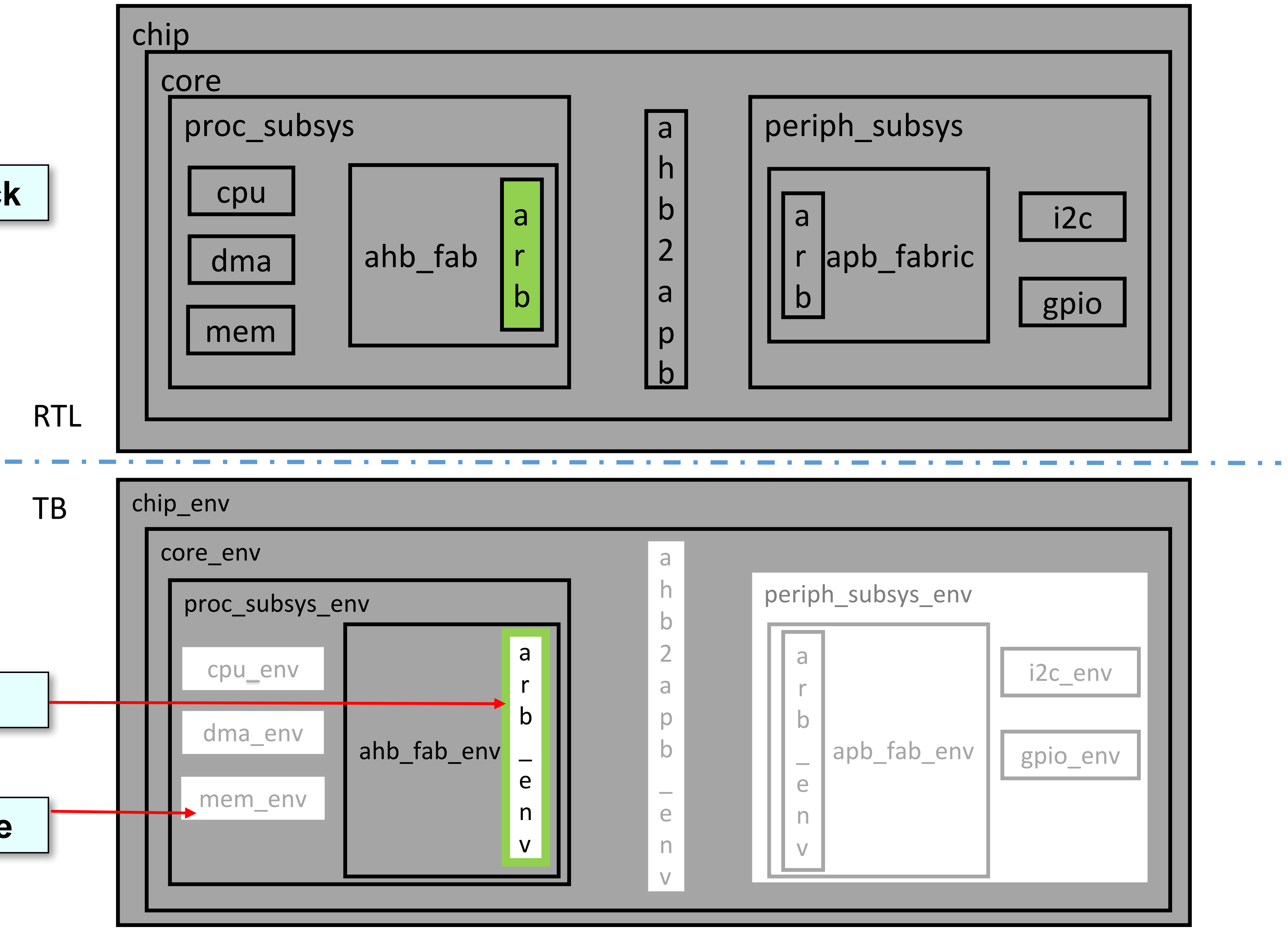
Configure topology down to cpu_env

proc_subsys_env *acting on* proc_subsys module

```
class proc_subsys_xcpu_xdma_test extends verilab_chip_base_test;
  virtual function void build_phase(uvm_phase phase);
    …
    verilab_chip_env_cfgs["dut"].verilab_core_env_cfgs ["verilab_core_0"]
                                 .proc_subsys_env_cfgs  ["proc_subsys_0" ].role
                               =  proc_subsys_pkg::ACTING_ON;

    verilab_chip_env_cfgs["dut"].verilab_core_env_cfgs ["verilab_core_0"]
                                 .proc_subsys_env_cfgs  ["proc_subsys_0" ]
                                 .cpu_env_cfgs          ["cpu_0"         ].role
                               =  cpu_pkg::ACTING_AS;
```

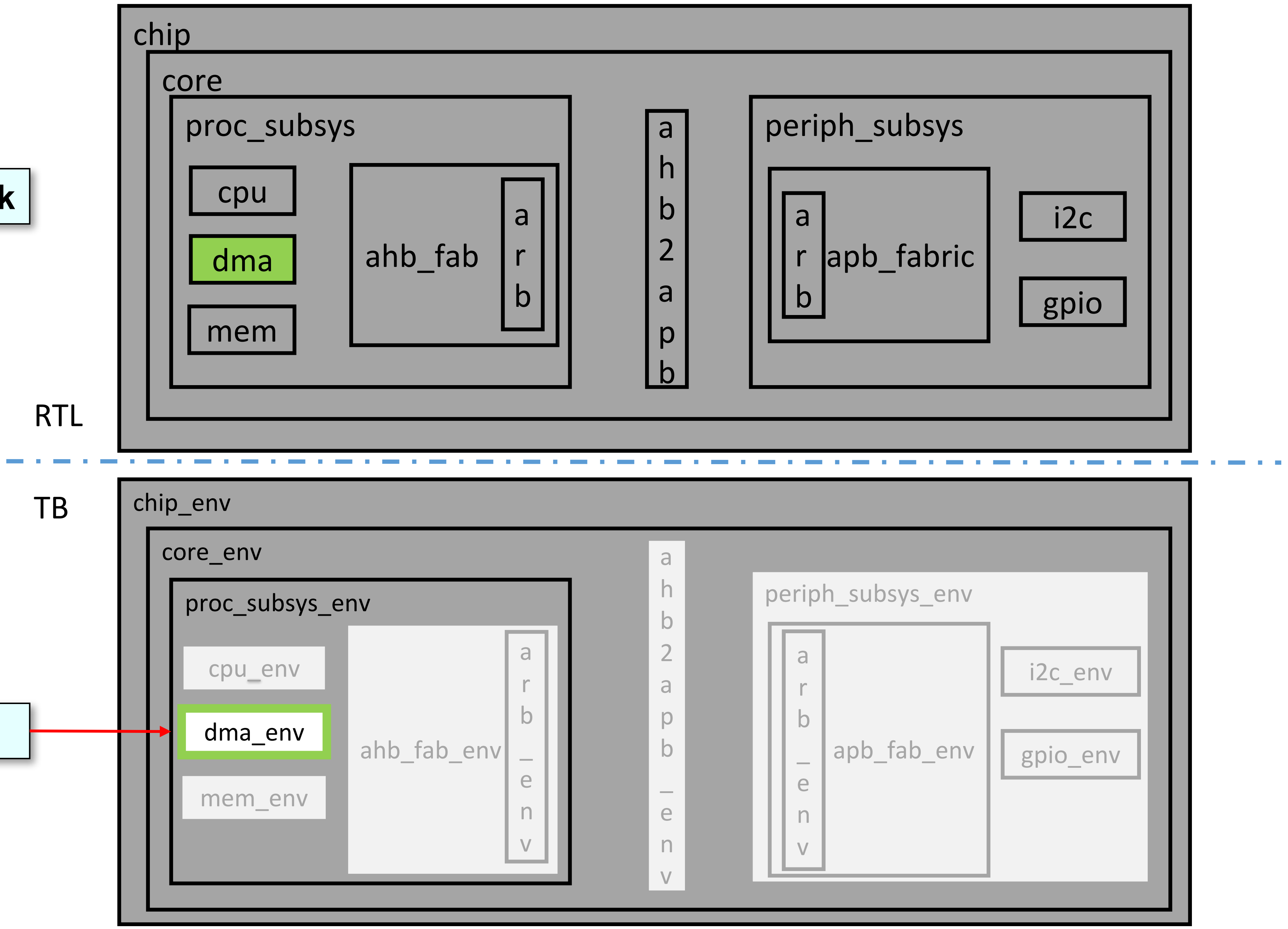
cpu_env *acting as* cpu module

## What Else Can we Do?

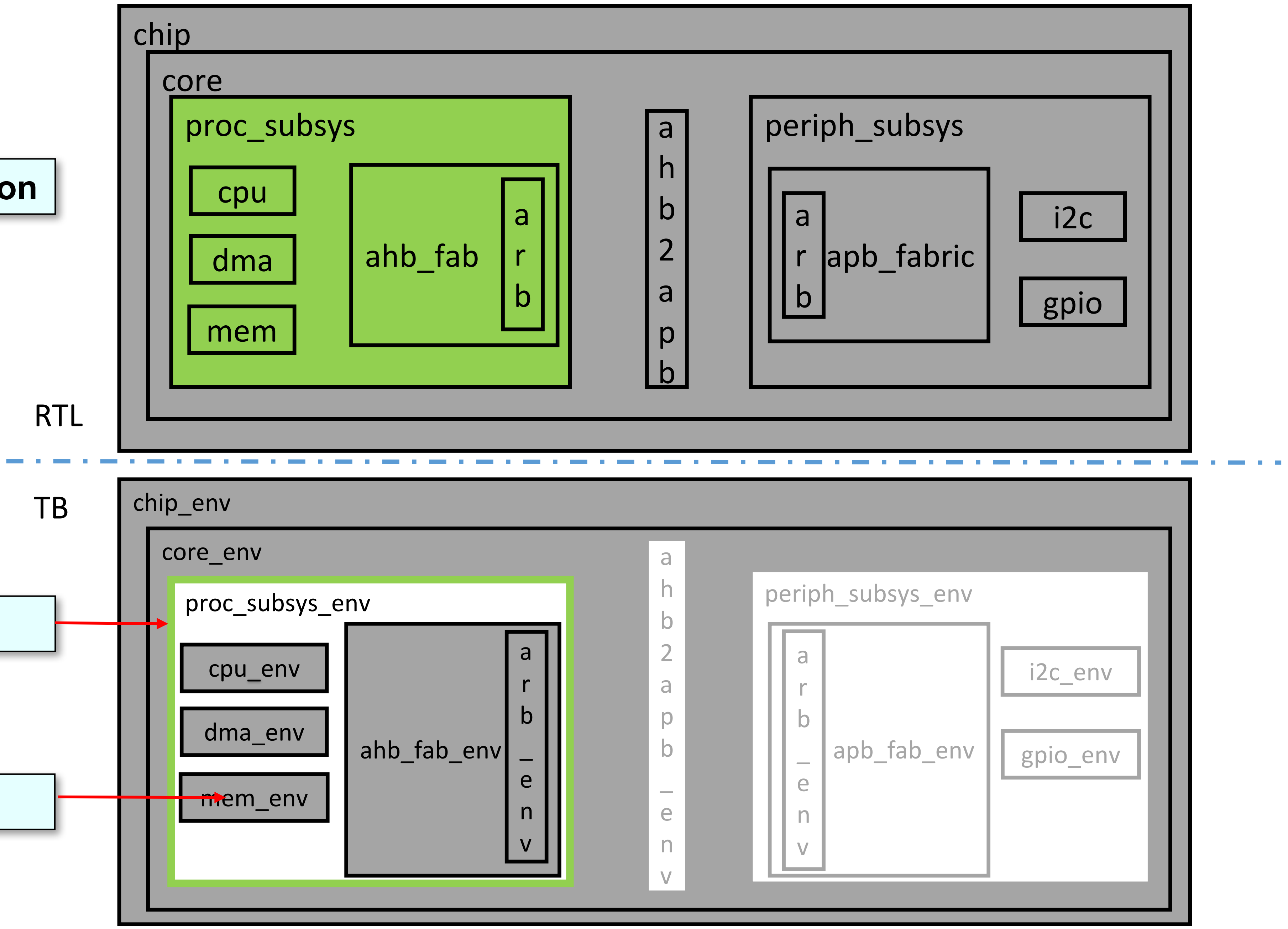

Lowest Block
chip
core
proc_subsys
cpu
dma
mem
ahb_fab
arb
ahb2apb
periph_subsys
arb
apb_fabric
i2c
gpio
RTL
TB
chip_env
core_env
proc_subsys_env
cpu_env
dma_env
mem_env
ahb_fab_env
arb_env
ahb2apb_env
periph_subsys_env
arb_env
apb_fab_env
i2c_env
gpio_env
Acting On
Don’t Create

**Mid-Level Block**

**Acting On**



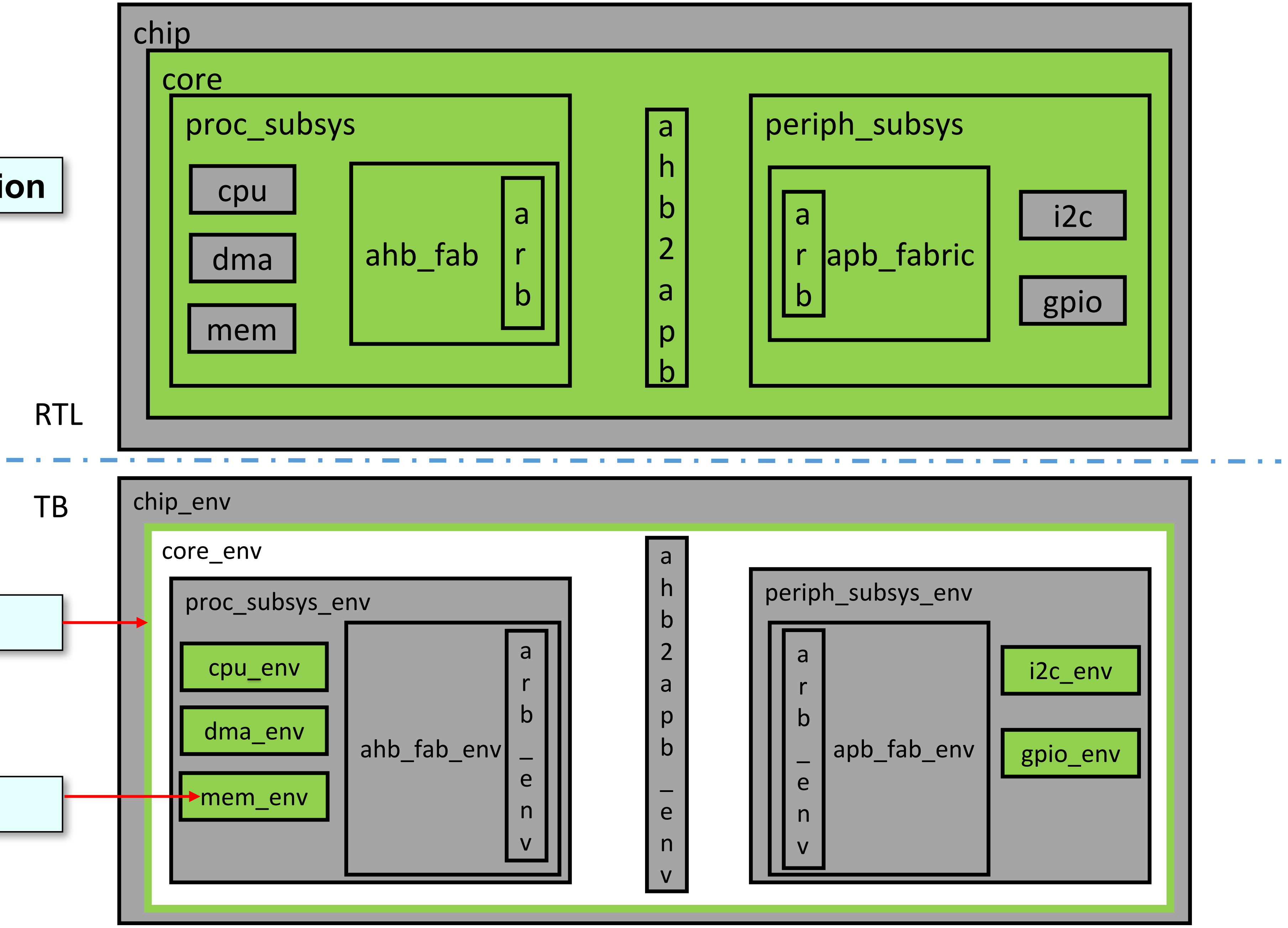
## Block Integration



Acting On

Passive

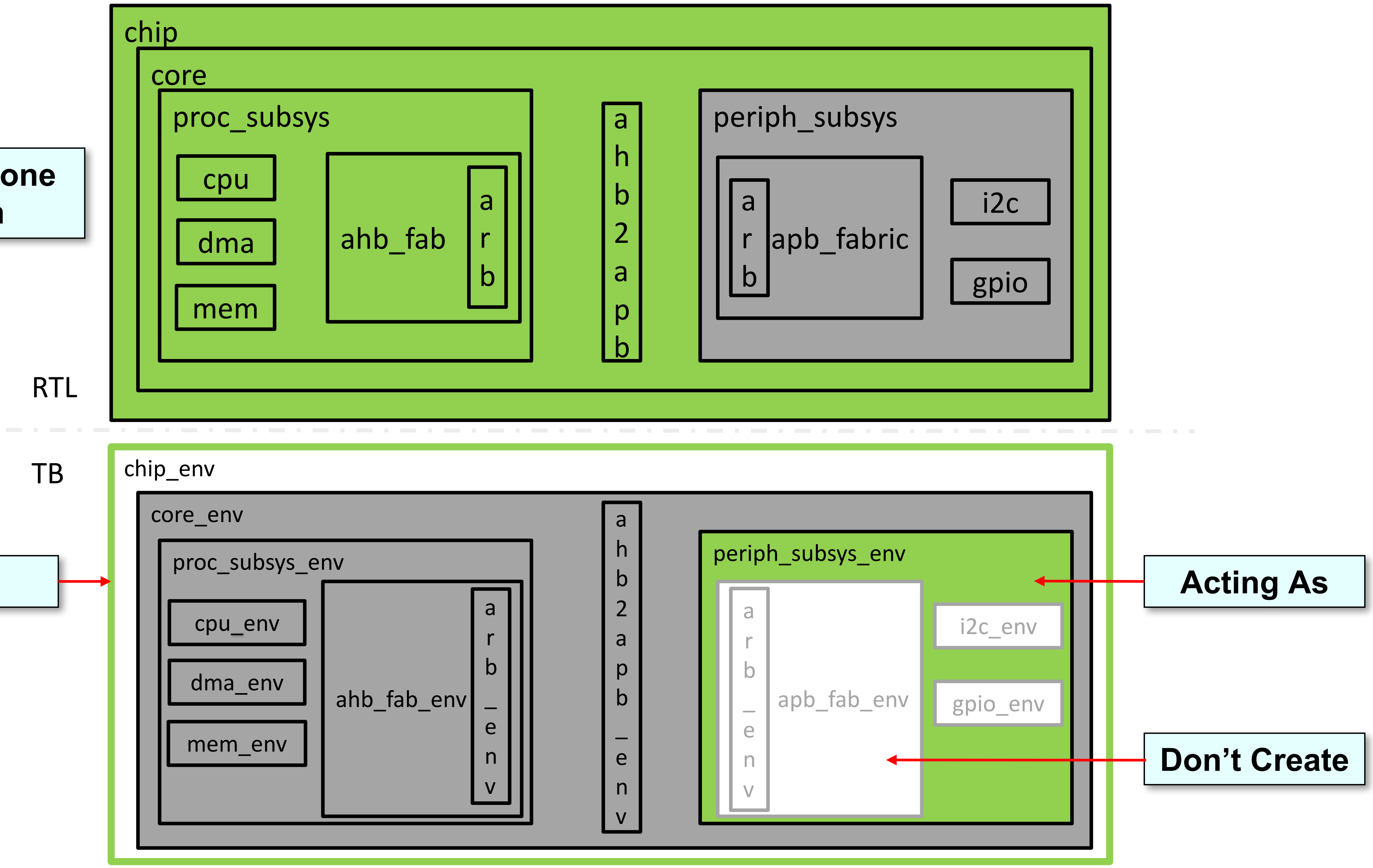
## Fabric Integration



**Acting On**

**Acting As**

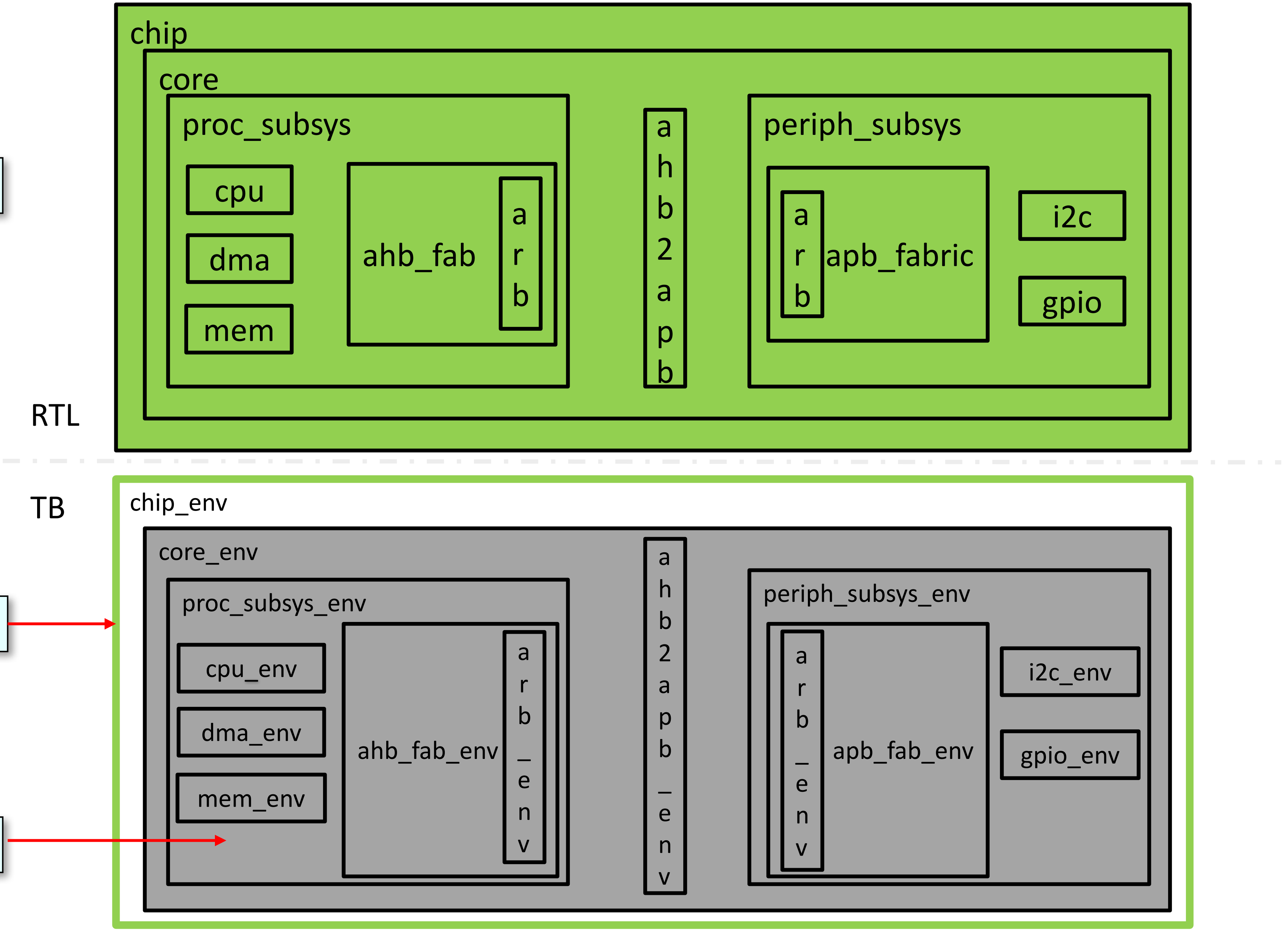
## Integration of one subsystem



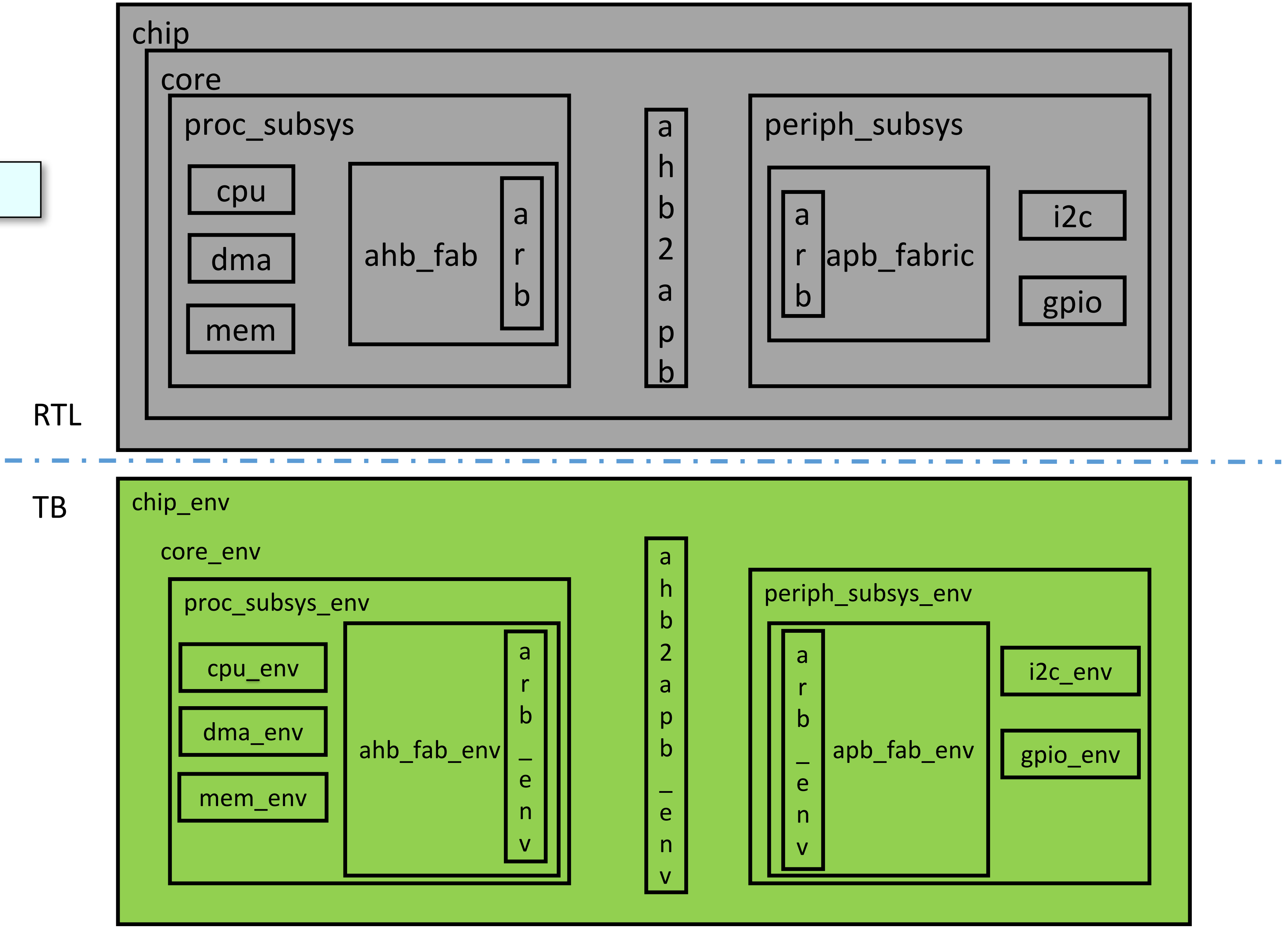
**Full Chip**

**Acting On**

**Passive**



No RTL!



RTL
chip
core
proc_subsys
cpu
dma
mem
ahb_fab
arb
ahb2apb
periph_subsys
arb
apb_fabric
i2c
gpio
TB
chip_env
core_env
proc_subsys_env
cpu_env
dma_env
mem_env
ahb_fab_env
arb_env
ahb2apb_env
periph_subsys_env
arb_env
apb_fab_env
i2c_env
gpio_env

## Conclusion

- Build a testbench that bends without breaking
  - Adapts to multiple design versions
  - Adapts to changes in design hierarchy
  - Simulate subsystems in isolation
  - Error-proof connectivity to the DUT

- Save time and effort on projects and follow-on projects

- Demo code available

http://www.verilab.com/resources/source-code/