

Verification Prowess with the UVM Harness

Interface Techniques for Advanced Verification Strategies

Jeff Vance, Jeff Montesano
Verilab Inc.

October 19, 2017
Austin



Agenda

Introduction

UVM Harness Overview

Enhancing the Harness

Conclusion

Introduction



Introduction

- Key takeaways that will be demonstrated:
 - You no longer need to manage tons of assign statements and wires
 - You no longer need to adjust TB connections when DUT hierarchy changes
 - You no longer need to bother with parameterized signal widths
 - You no longer need to write code to route signal directions and resolve drivers
 - Your SV/UVM testbench will have more capabilities than ever before

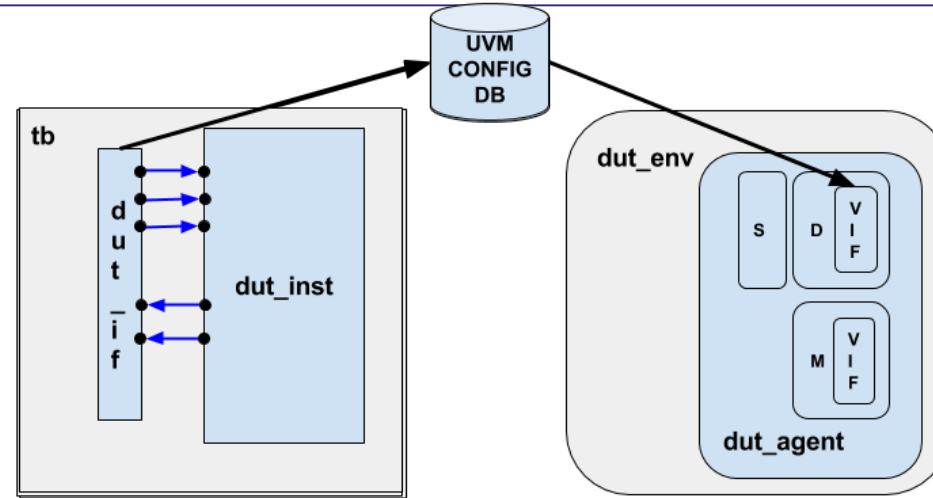
UVM Harness Overview



Traditional DUT to Testbench Connection

```
module tb;
...
if_type dut_if();
dut dut_inst(.clk(dut_if.clk),
             .rst(dut_if.rst),
             ...);
...

initial begin
    uvm_config_db#(if_type)::set(null, "*.dut_driver", "dut_vif", dut_if);
...
endmodule
```




```
class dut_driver extends uvm_driver;
    virtual if_type vif;
    function build_phase(...);
        uvm_config_db#(virtual if_type)::get(this, "", "dut_vif", vif);
        ...
    endclass
```


Interface Ports

- We must not use interface signals
- We must use interface ports of type wire

```
interface if_type();  
  logic clk;  
  logic rst;  
  logic data_in;  
  logic data_out;  
  
  modport active_mp (  
    input clk, rst, data_in,  
    output data_out);  
  
  ...  
endinterface
```



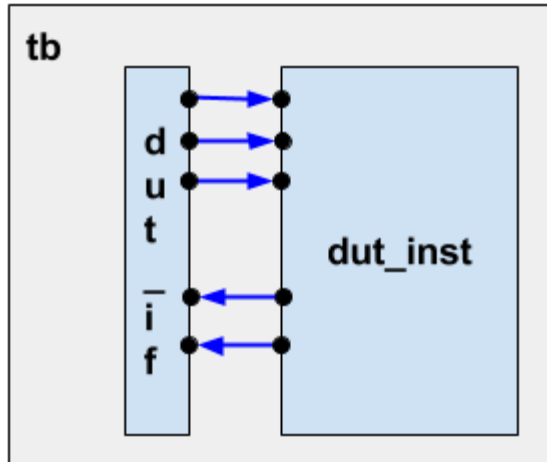
```
interface if_type(input clk,  
                  input rst,  
                  input data_in,  
                  output data_out);  
  
  ...  
endinterface
```



Interface Placement

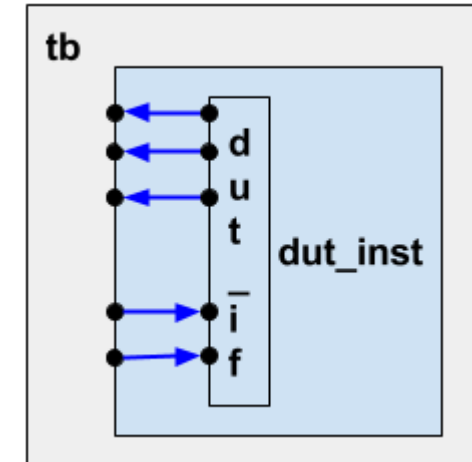
Traditional placement

- Interface is *outside* design module



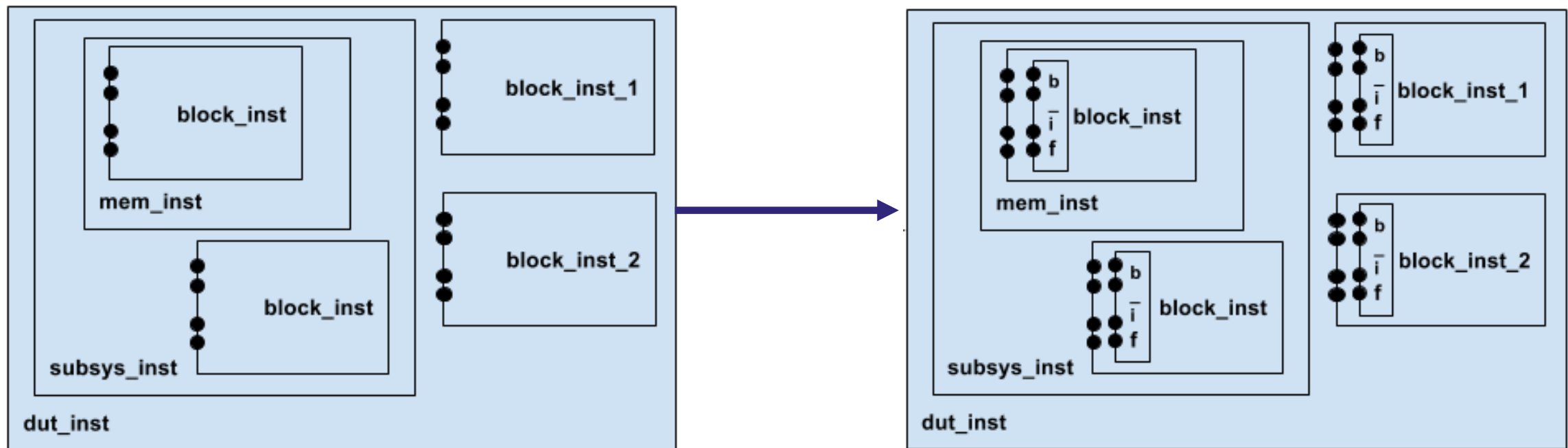
UVM Harness placement

- Interface is *inside* design module



Putting Interface Inside DUT Module

- SystemVerilog bind directive can amend a module definition
 - Syntax: **bind** <module_type> <interface_type> <interface_name>;
- Amending module definition naturally affects all instances



bind block_module_type b_if_type b_if();

UVM Harness Technique: Step-by-Step

Step 1: Define DUT interface

- Define **DUT interface** with all signals declared as ports

```
interface b_if_type(input clk,  
                    input rst,  
                    input data_in  
                    output data_out);  
endinterface
```

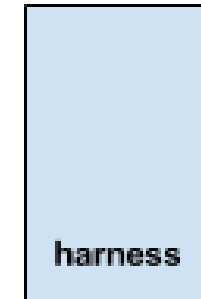


UVM Harness Technique: Step-by-Step

Step 2: Define Harness

- Define a **harness**

```
interface b_harness();  
endinterface
```

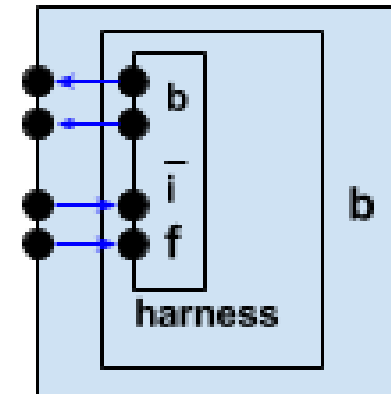


UVM Harness Technique: Step-by-Step

Steps 3 & 4: Instantiate and connect the DUT interface

- Make the harness instantiate the DUT interface
- Connect DUT interface ports to DUT using upward reference to module type

```
interface b_harness
  b_if_type b_if(.clk      (b.clk),
                 .rst      (b.rst),
                 .data_in   (b.data_in),
                 .data_out  (b.data_out)
  ...);
endinterface
```



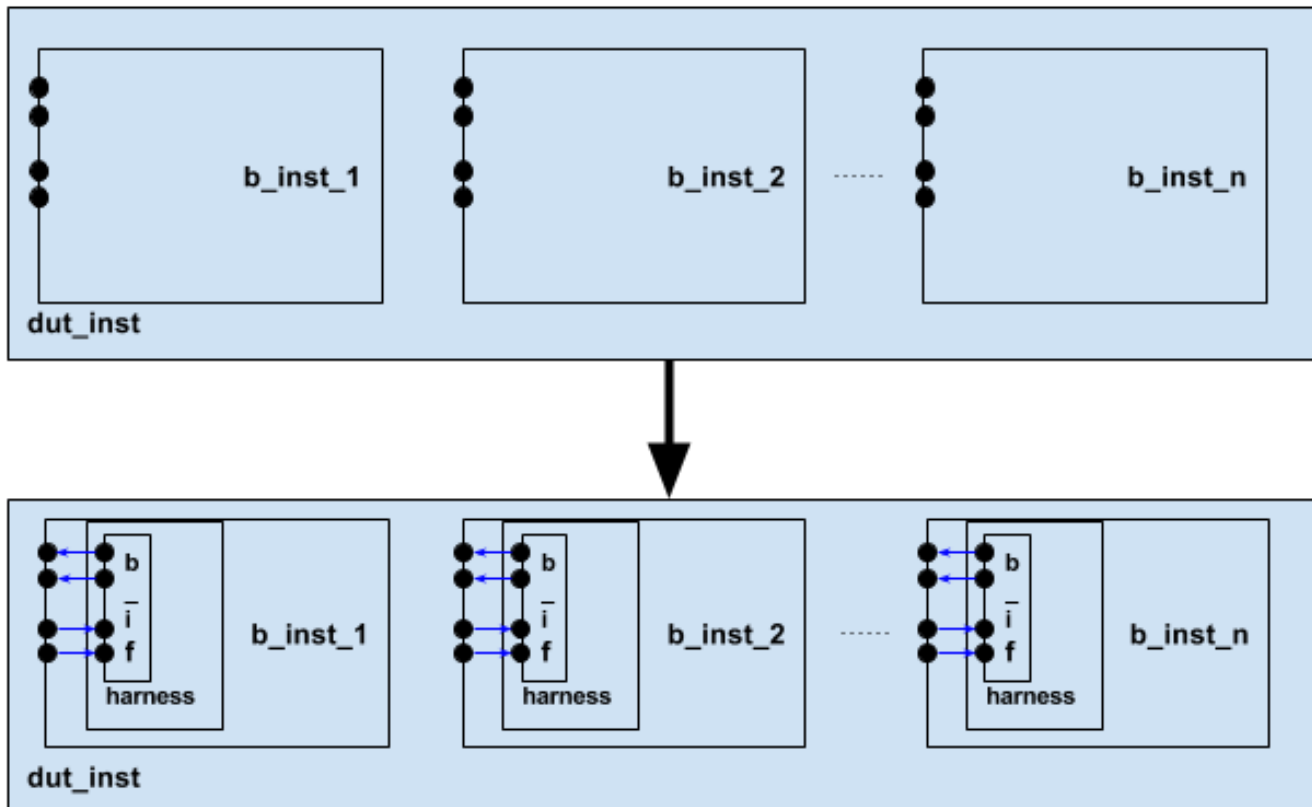
“b” is the module type

UVM Harness Technique: Step-by-Step

Step 5: Bind the harness

- Bind the harness to the DUT module definition

```
// bind <module_type> <interface_type> <interface_name>";  
bind b b_harness harness;
```



UVM Harness Technique: Step-by-Step

Step 6: Add set_vif method

- Add set_vif() function to harness

```
interface b_harness();  
    b_if_type b_if(.if_clk(dut_clk),  
                  .if_rst(dut_rst),  
                  ...);  
    function void set_vif(string path);  
        uvm_config_db#(b_if_type)::set(null, path, "b_vif", b_if);  
    endfunction  
endinterface
```

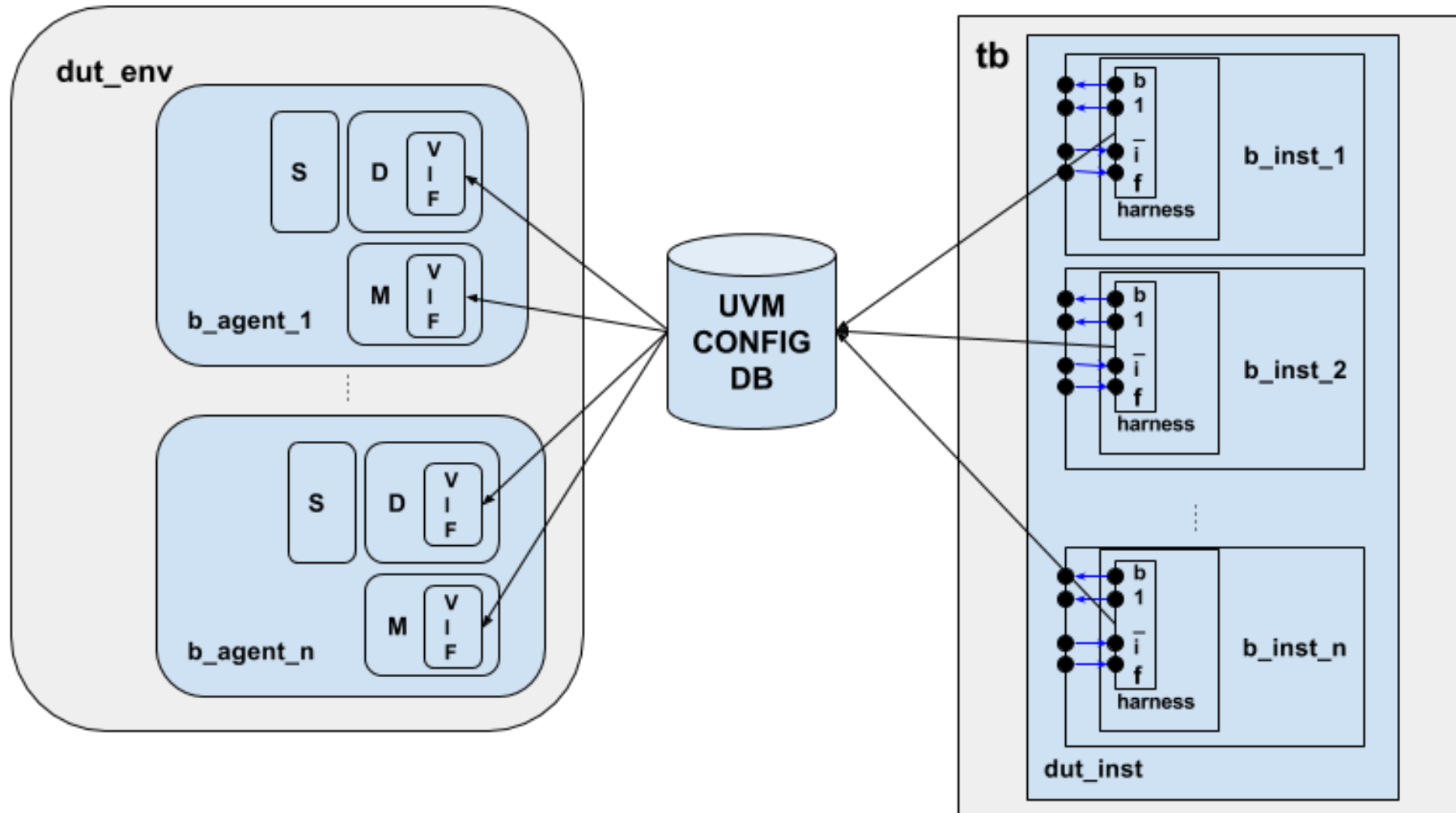

UVM Harness Technique: Step-by-Step

Step 7: call set_vif method

- Call set_vif() from top testbench module

```
module tb;
  dut dut_inst();
  ...
  initial begin
    dut_inst.b_inst_1.harness.set_vif("*.env.b_agent_1.driver");
    dut_inst.b_inst_1.harness.set_vif("*.env.b_agent_1.monitor");
    ...
    dut_inst.b_inst_n.harness.set_vif("*.env.b_agent_n.driver");
    dut_inst.b_inst_n.harness.set_vif("*.env.b_agent_n.monitor");
  end
endmodule
```

UVM Harness Technique



UVM Harness Technique Benefits

- Key benefits of basic UVM Harness technique:
 - Testbench automatically adapts to changes in DUT hierarchy
 - DUT-interface connections are encapsulated in harness
 - Testbench module file is greatly simplified

Enhancing the Harness

Improve Flexibility and Power



Signal Width Problem

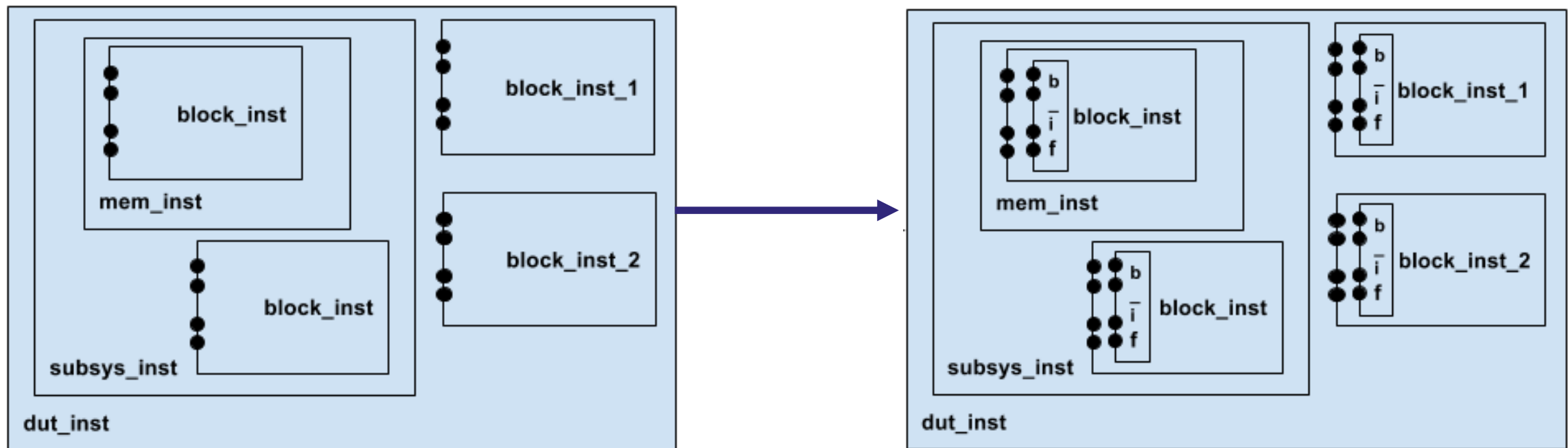
Common Issues for any Testbench

- Interfaces have different widths
 - 32 vs 64-bit data
 - Address bus widths
 - Device ID and sideband signal widths
- Changes to signal widths break simulations
- Limits reuse of interfaces between projects
- Parameterized interfaces adds complexity and maintenance

Signal Width Problem

Significant for Harness

- A single bind statement connects *all* interface instances
- Each instance may have different width signals

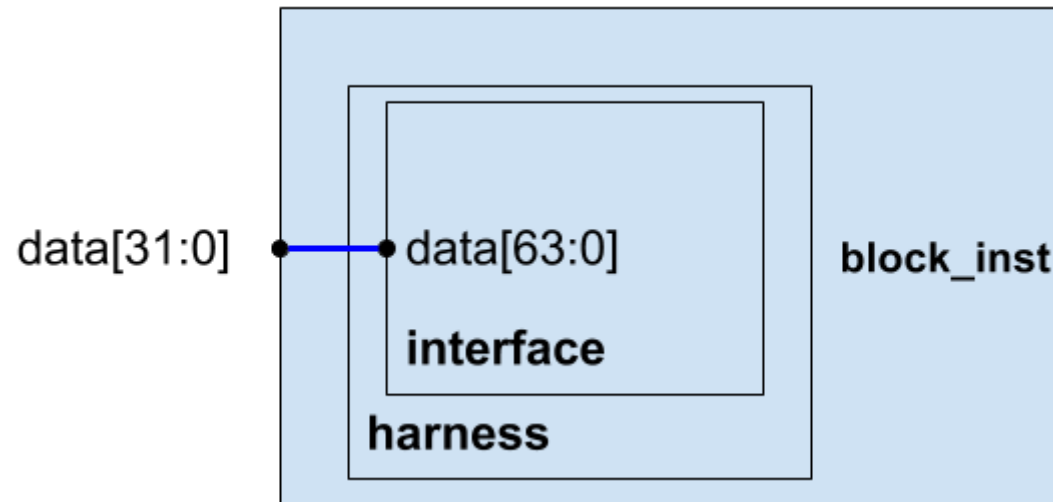


```
bind block_module_type b_if_type b_if();
```


Variable Width Support

Common Solution

- Use maximum width for all interface signals
- We can connect signals of width less than the max
- Changes to width don't impact connection (unless max width changes).



Signal Direction Problem

Common Issues for any Testbench

- Active vs Passive Roles
 - Active agents use both output and input signals
 - Passive agents treat everything as inputs
- Master vs Slave Roles
 - Masters drive request signals as outputs
 - Slaves monitor the same signals as inputs
- Common solutions are not ideal
 - Different master and slave interface definitions
 - Assignment logic to manipulate directions (using compiler directives)

Solution: Port Coercion

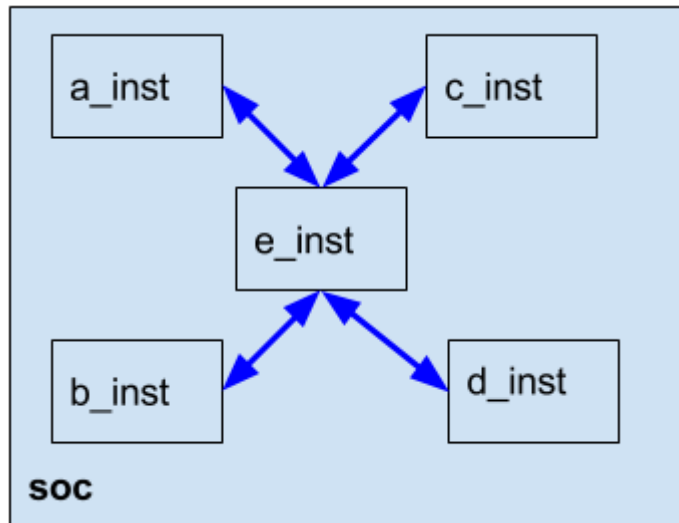
- Define **one** universal interface
 - Same interface for both master and slave agents
 - Same interface for both active and passive agents
- Don't use **inout** ports
 - max-width interfaces don't work with inout ports
 - VCS Log: *"Error-[IOPCWM] Inout port connection width mismatch"*
- Declare all interface ports as **input**
 - **port coercion** changes ports to *inout* at elaboration
 - Any driver with a driving statement via **virtual interface** will coerce the port
 - VCS Log: *"Notice: Ports coerced to inout, use -notice for details"*

Stub Modules

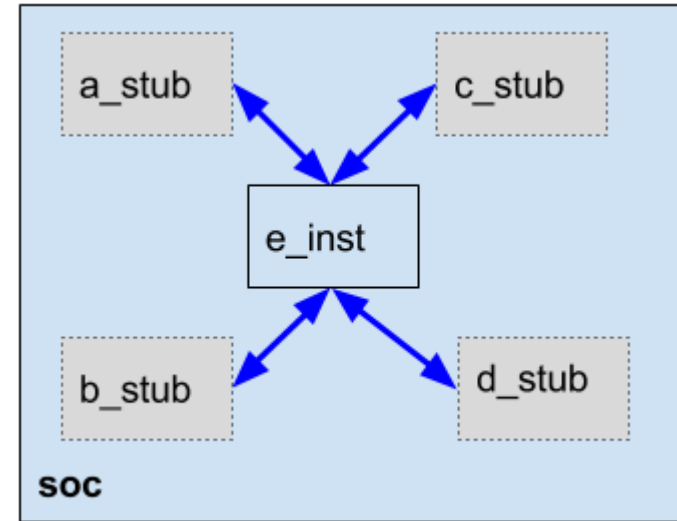
Improve Testbench Capabilities with Stubs

- What is a stub module?
 - Same ports and parameters as design module
 - No functionality
- Why use stub modules?
 - Performance benefits
 - Advanced verification strategies
- Common problem with stubs:
 - Agents change roles
 - passive and active
 - master and slave

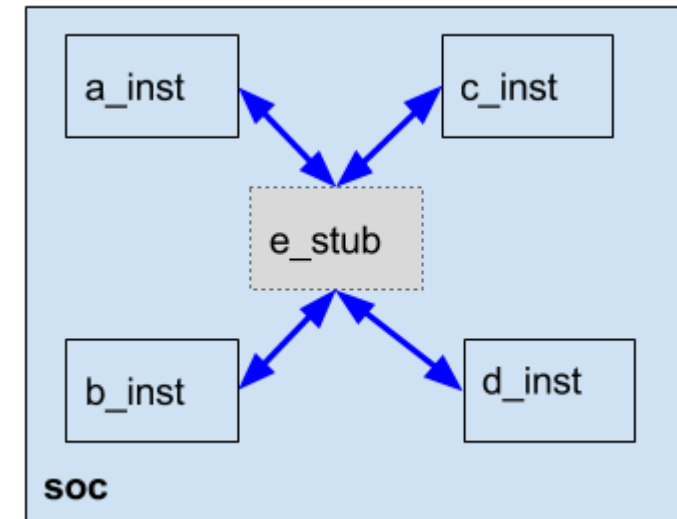
Stubbing Strategies



stub out

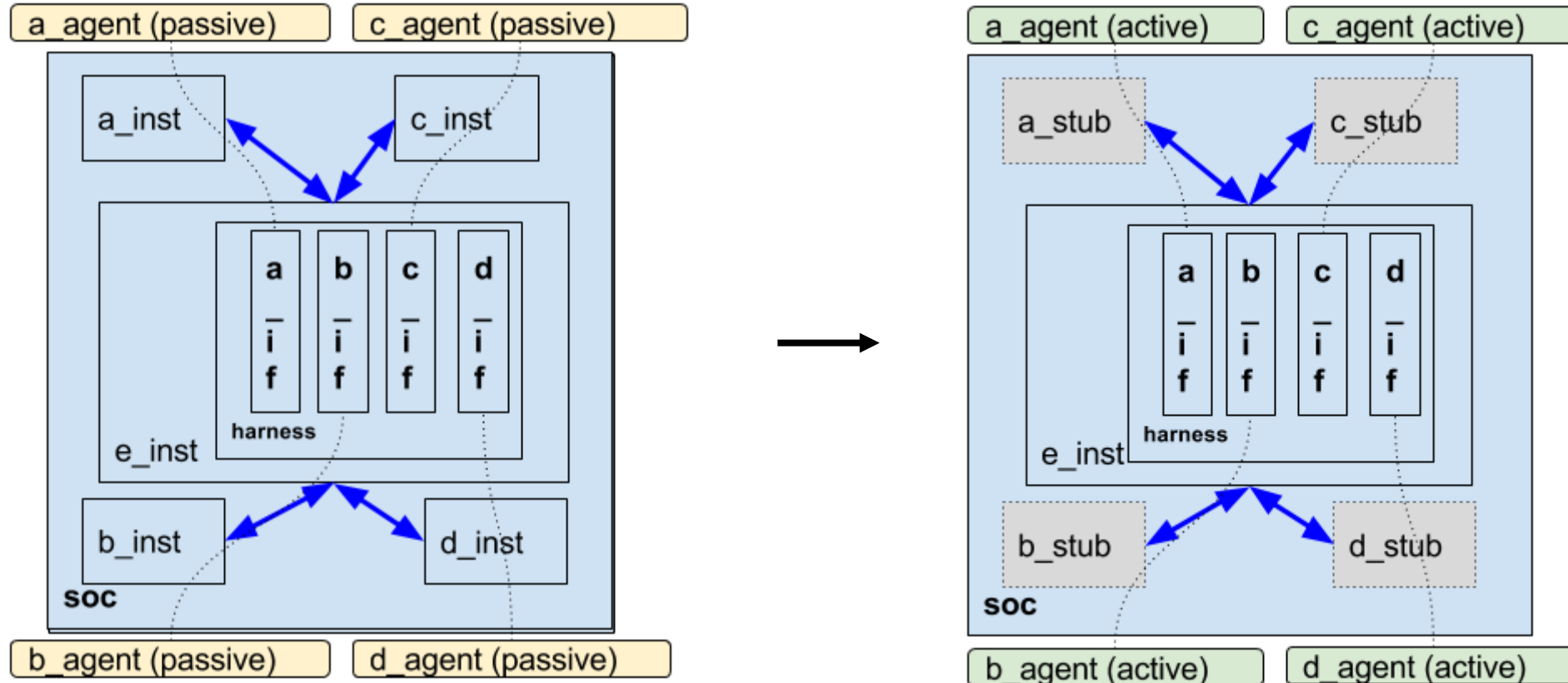


stub in



Block-level Verification with System TB

“Stub out” all modules except one under test



Block Level Verification with System TB

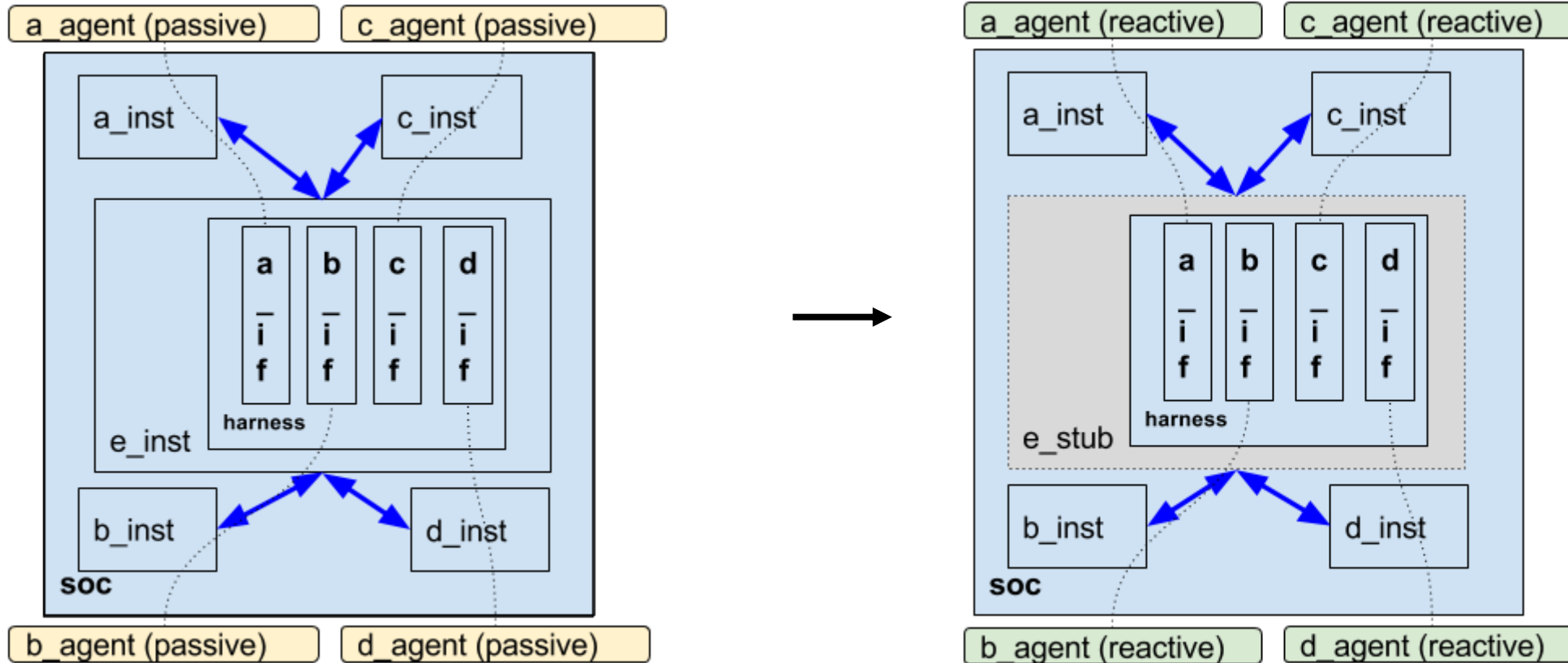


Benefits

- Maintain a single testbench instead of many block-level ones
 - We usually need an SoC testbench anyway
 - Equivalent to a combinatorial number of testbenches -- **for free**
- Verify multiple design versions: 2-core, 4-core, 8-core, etc.
- Block-level tests get accurate system-level clocks and resets
- Bug isolation is easier
 - Recreate failing system test scenarios for isolated blocks

System-level Verification

“Stub in” testbench stimulus for an internal module



Encapsulate Methods in the Harness

- Harness has DUT scope
 - Use “upward” reference to module type
 - Continue references “downward” with instance names
 - `<module type>.<instance name>.<signal name>`
- Agents/sequences/assertions could make use of a harness API to:
 - Preload memories from files
 - Force signals
 - Declare assertions between signals in different interfaces
- Outside code (agents/sequences/assertions) needs access to harness API
 - Ideally harness would be published to **uvm_config_db** as virtual interface
 - **Problem:** LRM restrictions prevent this

Encapsulating Methods in the Harness

Solution: Polymorphic Harness

- Step 1: Define an API Class
 - Create class inside harness
 - Make harness class methods call harness methods
 - **Problem:** class is not visible to code outside of harness

```
interface dut_harness();

    task harness_force_sig1(int data);
        force dut.sub_mod.sig1 = data; // "dut" is a module name
    endtask

    class harness_api;
        task force_sig1(int data);
            harness_force_sig1(data);
        endtask
    endclass

    harness_api api; //instance of API class
endinterface
```

Encapsulating Methods in the Harness

Solution: Polymorphic Harness

- Step 2: Class Visibility Workaround
 - Harness class extends a base class
 - Put base class in a package: it can be imported anywhere
 - Pass *the base class* type as uvm_config_db type.

```
interface dut_harness();  
    import abs_pkg::*; // abs_pkg has class harness_api_abstract  
  
    class harness_api extends harness_api_abstract;  
        task force_sig1(int data);  
            harness_force_sig1(data);  
        endtask  
    endclass  
  
    harness_api api; //instance of API class  
  
    function void set_vif(string path);  
        api = new("harness_api");  
        uvm_config_db#(harness_api_abstract)::set(null,"*",  
                                                    "harness_api", api);  
    endfunction  
endinterface
```

Additional Topics Discussed in Paper

- Using SystemVerilog **force** as an alternative to stubs
- Handling VIPs that use signal-based interfaces
- Accessing DUT parameters
- References to original UVM Harness paper, Polymorphic Harness paper
- Future considerations

Conclusion

- **Simplify**
 - Eliminate lists of extra wires and assign statements
 - Remove dependencies on DUT hierarchy
 - Remove dependencies on signal widths and directions
- **Save Time**
 - Reduce maintenance for design changes
 - Reduce the number of independent testbenches
 - Improve reuse between projects
- **Harness the Power**
 - Encapsulate methods and access RTL signals
 - Stub any combination of design modules
 - Control internal stimulus to reach special scenarios

Start using the
harness now!

Thanks to author
Kevin Johnston!

Thank You

