

# Simulation Acceleration with ZeBu to Speed IP and Platform Verification

Hillel Miller, Wei-Hua Han – Synopsys



# Agenda

- Customer stories and ZeBu simulation acceleration technology overview (45 minutes) – Hillel Miller
- AXI example (25 Minutes) – Wei-Hua Han
- Q&A (10 Minutes)

# Customer story one

- My simulations, for my design verification environment are very slow, next generation it will be much slower, even 2x will help
- We will do anything for 2x
- My VP has given me an initiative to achieve 2x

# Customer story two

- I am developing a graphics core as third party IP vendor
- I need to do power estimation of my graphics core with Manhattan test case
- It takes more than 2 weeks to run the simulation to get power estimation numbers in my SystemVerilog environment
- I want to complete this effort in less than 24 hours

# Customer story three

- My simulations are very slow
- My Top Level Verification (TLV) environment, uses embedded C test cases running on our DSP core in RTL simulation model. The C test cases are far more successful than standalone UVM environments in finding bugs
- Some of the test cases take weeks to run

# Customer story four

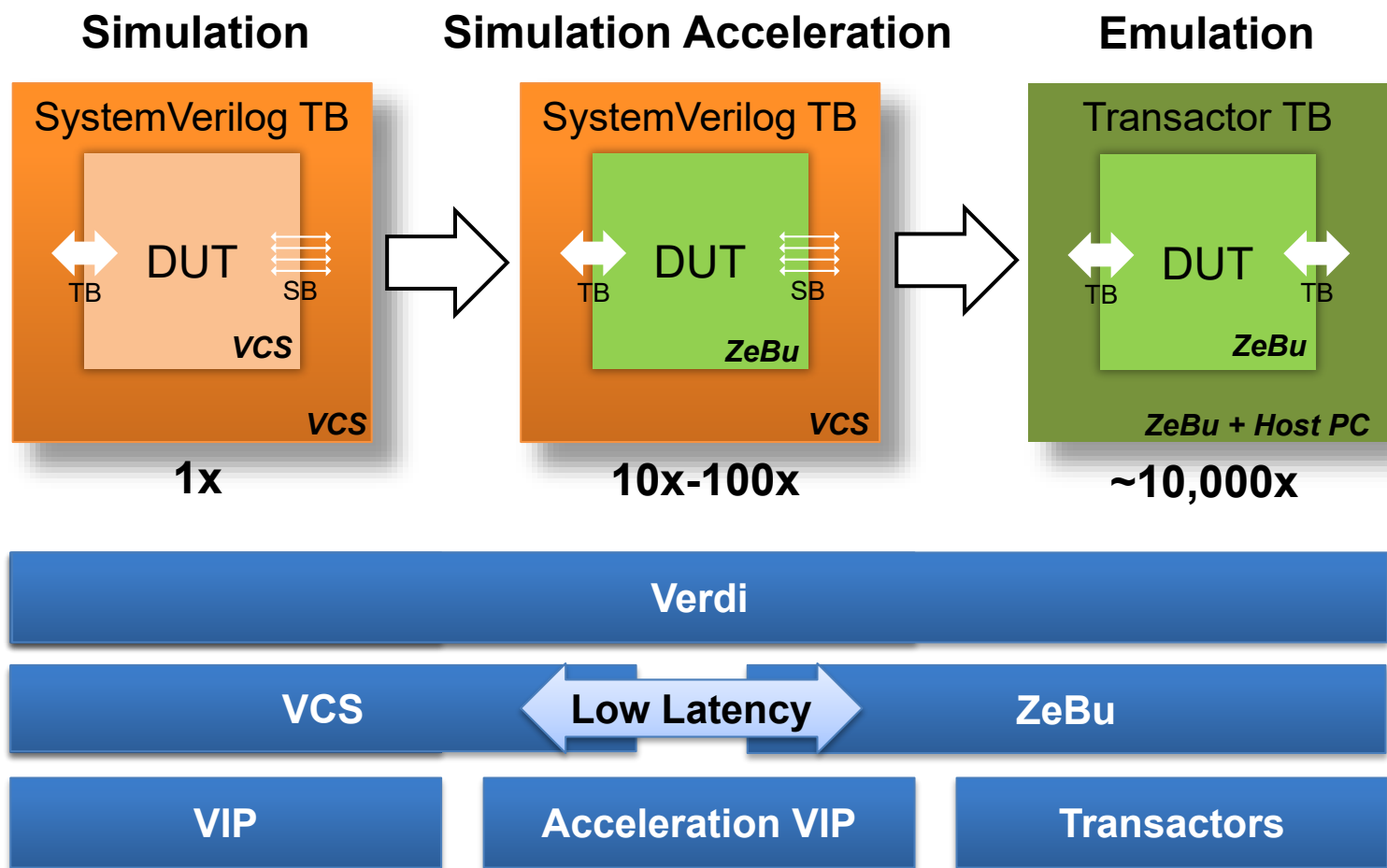
- I have an IP level UVM environment, with custom UVM VIP. My long test take days to run for number of packets in 10,000's range. My next generation IP will be 2x more complex and expect simulation performance to degrade
- We are missing bugs found by SoC
- My current regression runs on 100 machines
- I need two things
  - Run at least 100x faster so that I can match simulation regression running on 100 machines
  - Run millions of packets

# Customer story five

- My SoC test cases take days to run. My next generation will run even slower
- I have been using Synopsys tools SVT VIP, NLP, SVA, URG, Verdi for as long as can tell
- I would like to use the ZeBu emulator under the hood without any changes to speed up my simulations

# Simulation Acceleration with ZeBu

*10-100x higher performance*



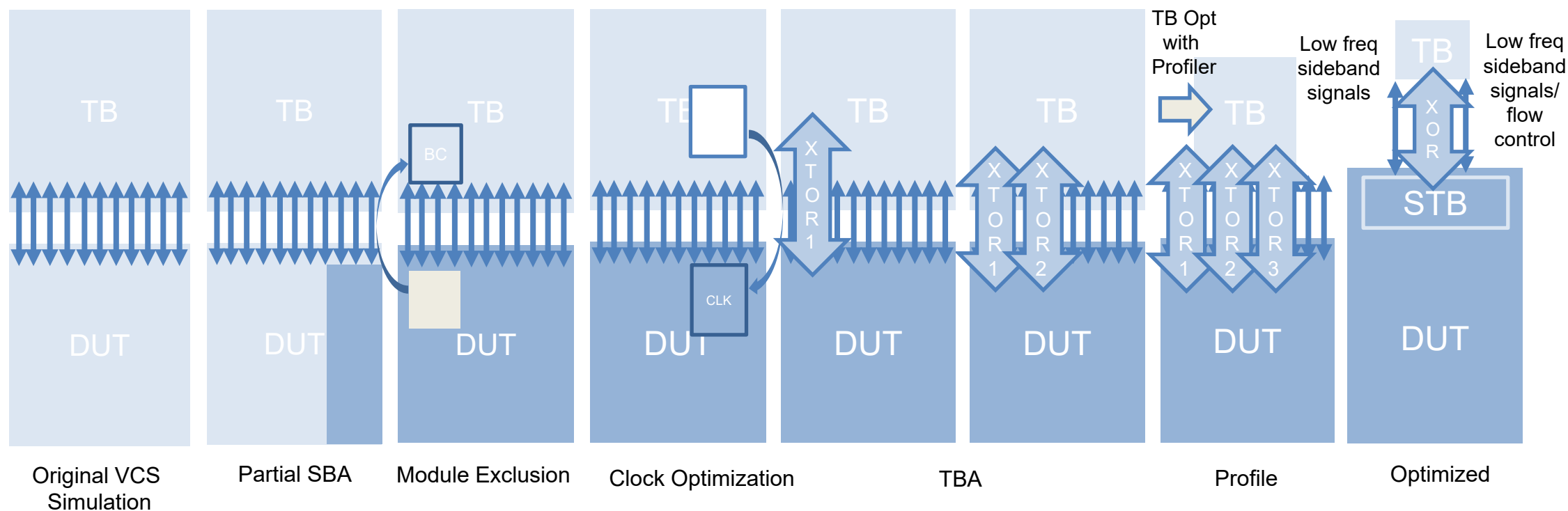
## Benefits

- Native integration with VCS
- Fully automated compile of testbench and design
- Industry leading VCS and ZeBu performance
- Integrated, interactive Verdi debug
- Coexistence of Signal and Transaction-based communication
- Low latency HW/SW interface
- Acceleration VIPs for additional acceleration



# Continuous Simulation Acceleration

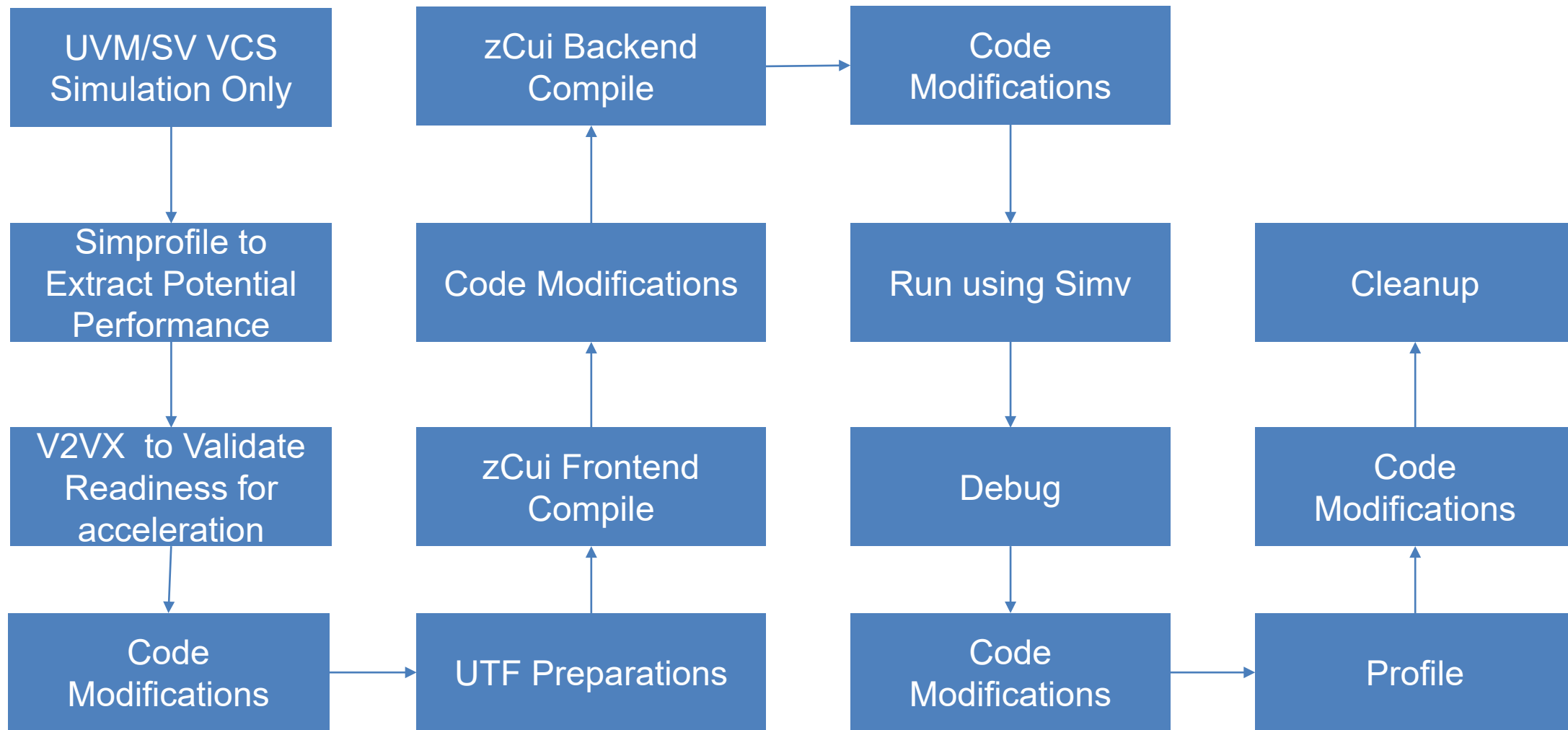
Ability to incrementally change and mix HW/SW communication capabilities



# Simulation Acceleration

Customer Design		Acceleration factor	Type
Multimedia	Sub-System	25x	TBA
	Sub-System	80x	SBA
Multimedia	Block	18x	SBA
	SoC	14.5x	SBA
Multimedia	Sub-System	1.5x	SBA
	Sub-System	4.8x	SBA
	Sub-System	1.5x	SBA
Processor	Sub-System	6.0x	SBA
	Sub-System	8.0x	SBA
Processor	CPU Sub-System	6x	SBA
GPU	Sub-System	28x, 64x	SBA
	Sub-System	104x, 2091x	TBA
Networking	Block	62x	TBA

# Deployment Steps for UVM/SV



# VCS Options – V2VX mode

- Uses only VCS for both HW and SW compilation. Used for pipe cleaning before moving to ZeBu compile
- VCS Compile option to enable V2VX mode: -Xhwcosimtest=v2vx
- VCS Compile option to specify HW top level instance: -Xhdl\_cosim\_dut <hierarchical path to the HW instance> If module is compiled as a top-level module, then it will be the name of the module
- Run using existing simv options
- Debug using Verdi

# UTF Options – V2Z mode

- Uses ZeBu for HW compilation and VCS for SW compilation
- UTF files is a list of commands specifying at a high level how to execute the compile
- ZeBu zCui tool will do the full compile with a UTF file as input and the UTF file pointing to the VCS compile, using the UTF command VcsCommand
- Compile generates a zcui.work directory which is used at runtime to enable VCS to co-simulate with the emulator
- For compile add the following UTF options and VCS compile options:
  - UTF command to enable acceleration: `simxl –enable true`
  - UTF command to specify HW top level instance
    - `simxl_set_hwtop -instance <hierarchical path to the HW instance>`
    - All logic from that instance and below will go into HW. If module is compiled as a top-level module than it will be the name of the module.
- ZCUI compilation GUI is opened using: `zCui -u project.utf`
  - Push the Make Target button in the ZCUI GUI to start the compile process

# VCS Compile/Runtime – V2Z mode

- VCS additional compile option to generate diagnostic data in current run directory
- -Xhwcosim=diag
- Run using SIMV additional runtime option to point to zcui.work for HW database: +zcui.work=<zcui.work directory generated by zCui>

# Structural Signal Communication

- Signal communication using the following primitive data types for signals:
  - wire, wand, wor, tri, etc.
  - reg
  - logic
  - enum
  - byte
  - integer
  - int
  - shortint
  - longint
  - Signed modifier can be applied to any of the above data types.
- Signal communication using the following aggregate data types:
  - Packed arrays.
  - Unpacked arrays.
  - Multi-dimensional arrays.
  - Packed structs.
  - Unpacked structs.
  - Unions.
  - Any composition of the above data types.
- Signal communication using interface ports:
  - Interface ports containing signals with any of the above types.

# Structural Signal Communication: Clocking

```
// Running in ZeBu
module HW(input wire clock, input logic reset);
endmodule

// Running in VCS
interface ift;
    logic clock;
    logic reset;
    assign HW.clock = clock;
    assign HW.reset = reset;
    clocking cb @(posedge clock);
        output clock;
        output reset;
    endclocking
endinterface
module SW;
    ift m_if();
Endmodule
```



# Structural Signal Communication: Datatype

```
typedef struct {  
    integer source;  
    integer destination;  
} descriptor;  
module HW(input wire clock, input logic reset,  
input wire descriptor d[100]);  
endmodule  
interface ift;  
    descriptor m_d[100];  
    genvar i;  
    for (i=0; i<100; i++)  
        assign HW.d[i] = m_d[i];  
endinterface  
module SW;  
    ift m_if();  
endmodule
```

# Behavioral Signal Communication

```
module hw(input a);  
  reg b;  
endmodule  
module sw;  
  reg c;  
  initial  
    hw.b = c;  
endmodule
```

# Force/Release HW from SW

```
module hw(input a);  
  reg b;  
endmodule  
module sw;  
  reg c;  
  initial begin  
    force hw.b = c;  
    wait(my_event.triggered);  
    release hw.b  
  end  
endmodule
```

# Assertion and Coverage in HW

To enable functional coverage and assertions at compile time, use the following UTF commands:

```
coverage -enable true // Enables covergroups specified in HW  
assertion_synthesis -enable ALL // Enables assertions and coverage  
properties specified in HW
```

To activate functional coverage and assertions at runtime, add the following options to the simv command:

```
-simx1=enable_dut_fcov,enable_dut_sva
```

Coverage data is generated in the simv.zebu.vdb file.

No special options are required to enable functional coverage and assertions in SW.

To merge coverage data from HW with coverage data from SW, use the following URG command:

```
urg -dir simv.vdb simv.zebu.vdb
```

# Preloading HW memory using SW

```
interface ift;
    logic [0:255] mem [4096];
endinterface
module dut;
endmodule
module HW;
    ift m_if();
    dut dut();
endmodule
module SW;
initial
    begin
        $readmemh("data.hex",
            HW.m_if.mem);
        $writememh("data_copy.hex",
            HW.m_if.mem);
    end
endmodule
```

# Export Subroutine Call

```
interface ift;
    task task1(output o1);
        begin
            o1 = 1;
        end
    endtask
    function integer calculate();
        return(0);
    endfunction
endinterface
module HW();
    ift m_if();
endmodule
module SW();
    logic a;
    initial
        begin
            HW.m_if.task1(a);
            $display("A: %d, CALC: %d", a,
                HW.m_if.calculate());
        end
endmodule
```

# Import Subroutine Call

```
class C;  
  function void observe(input integer i1);  
    $display("I: %d", i1);  
  endfunction  
endclass  
interface ift;  
  bit configure_done = 0;  
  C m_handle;  
  function void configure(C handle);  
    m_handle = handle;  
    configure_done = 1;  
  endfunction  
  initial begin  
    wait (configure_done == 1);  
    m_handle.observe(4);  
  end  
endinterface  
module HW();  
  ift m_if();  
endmodule
```

```
module SW();  
  C c1 = new();  
  initial  
  begin  
    HW.m_if.configure(c1);  
  end  
endmodule
```

# Moving HW module to SW

- Reasons to moving HW module to SW
  - HW module has constructs that are non-synthesizable
  - Instance of HW module is being accessed from SW in a non-supported way\
  - HW module is causing a compile bottleneck
  - Instance of HW module is causing runtime failures
- Add the following VCS option for V2VX flow

```
-Xhdl_cosim_etb <module name>
```

Must be specified for each module that is moved from HW to SW

- Add the following UTF command for V2Z flow:  

```
simx1_move_to_tb -module {<module name list>}
```

All Modules specified above will execute in SW, using their HW context.



# Moving clocks from SW to HW

- Reasons for moving clocks from SW to HW
  - Clocks toggling in SW increase HW/SW interaction slowing down performance.
  - #-delays executing in HW avoid all kinds of issues related to semantics, synchronization, race conditions, etc.
- Changes required to support clocks in HW
  - Add new wrapper module that will instantiate the design and clocks.
  - Wrapper module will be instantiated instead of DUT and should have same ports as DUT except for clocking ports. All XMRs will need to be updated to add wrapper segment.
  - Create clock generator module that will contain implementation of a clock using #-delay. See example below.
  - Add following commands to UTF file

```
clock_delay <name of clock generator module> -tolerance auto
clock_config -accuracy 32
// Needed when there is not enough bits to scale all clock frequencies
```
  - Try maintain single timescale throughout the parsed database.
- Validate changes by running representative test cases in simulation only.

# Guidelines: Time synchronization

- Usage of #-delays in SW
  - #-delays should not be used in SW
  - Price of usage is very high as HW needs to sync with SW every event clock
  - There is a feature in VCS that identifies all #-delays
    - Compile option for VCS is: -simprofile.
    - Runtime option for SIMV is: -simprofile delay
  - Event clock granularity is determined by smallest precision of timescales in HW
    - For example, if SW has #1 for timescale 1ns/1ns and HW has timescale 1ns/1fs, then there will be 1, 000, 000 synchronization points.
  - Use events that are originated from HW to control synchronization on SW side

# Guidelines: Time synchronization

- Simplify RTL or SW clocks
  - Try keep clocks with whole unit ratio's between clocks
  - Complex clocks in SW will slow down simulation significantly
  - Do not use decimal point for clock implementations
  - Minimize the number of clocks used, eliminate all unnecessary clocks (e.g. test clocks, random delayed clocks, etc)
  - For RTL clocks use tolerance feature to reduce number of clock events
    - Clock events are synchronization events that are used to determine the next time stamp.

# Guidelines: Time synchronization

- Simplify usage of timescale
  - Do not use decimal point in #-delays which requires needing timescale precision
    - Waveform calculation uses timescale finest precision as calculate rate. Finest precision may be an overkill
  - Coordinate single usage of timescale directive across the project
  - Validate changes to timescale using simulation test case
- Waiting for HW clock cycles on SW side
  - Do not call export task to wait for clock cycles
    - The same export cannot be called simultaneously during acceleration
  - Use an XMR to the clock in the HW side and repeat statement to wait for a finite number of clocks cycles (e.g. repeat @(posedge vif.clock))

# General Guidelines

- Avoid explicit initialization of memory or registers to Zero
  - Many wasted cycles, those elements are initialized to zero by default
  - Use +vcs+initreg+random at vcs elaboration time and +vcs+initreg+0 at runtime, for VCS simulation only
- Avoid calling tasks from always block for cycle update of variables.
  - Inline the update
  - Synthesis doesn't handle this well
- Avoid doing 100k's backdoor to explicit memory addresses or registers from SW
  - This is not as fast as VCS simulation
  - Updates should be done in HW through Transactors
    - May contain a lot of additional overhead in gate count.
- Bring up in the shortest running test available
  - Full vision is usually only achievable with a short run
  - Identify partial test and run till first point of failure

# General Guidelines (Cont'd)

- Initiate all actions from HW as much as you can.
  - Bind transactors to HW that initiate transactions.
  - Try to make actions non-time consuming.
  - Avoid scheduling on SW side, to reduce SW overhead.
  - HW synchronization is better over synchronization in SW.
- Implement transactor as BFM as much as you can
  - Runs on HW, needs to be synthesizable and not behavioral code, to achieve maximum performance.
  - Use a thin layer of behavioral code.

# Incremental Compilation

- Incremental compilation is implemented in the simulation acceleration flow
  - -Xhwcosim=incr\_comp
- VCS detects if the second compilation interface database is subset of previous database
  - Avoid new ZeBu compilation
  - Examples
    - Changes are completely local to testbench
    - Removing port connections or SW to HW XMRs from second compilation
- User can force TB only compilation if he is sure there is no change in HW and SW/HW communications
  - UTF command: `simxl_tb_compile_only -enable true`
- Incremental compilation diagnostics
  - -Xhwcosim=incr\_diag
  - “IncrDiag.txt” dumped with info about if TB and DUT communications changed or not

# INTERACTIVE DEBUG WITH VERDI



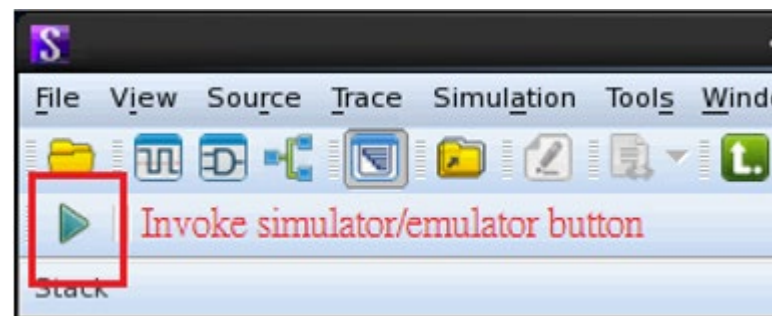
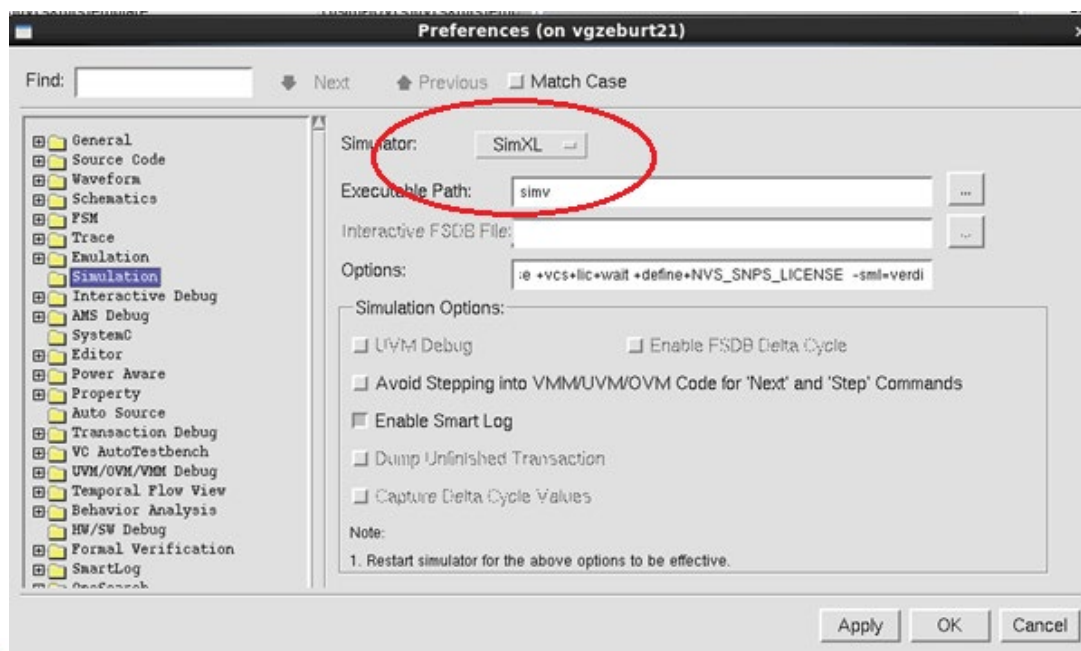
# Invoking Verdi

Invoke simulation acceleration option in Verdi command line

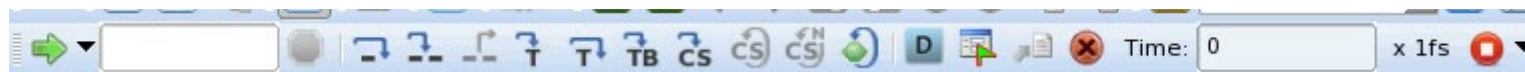
```
verdi -simBin simv -i -emulation -lca -simXL --root (ztdbHwRoot) -simDelim  
+zcui.work=zcui.work/zebu.work +zebu.verbose +vcs+lic+wait  
+define+NVS_SNPS_LICENSE
```

- (Green part is Verdi option; blue part is runtime option; use “-simDelim” to separate them)

Invoke simulation acceleration using preference dialog in Verdi



# Verdi Interactive Debug Toolbar



- Run
- Stop
- Next
- Step in
- Step out
- Step in thread
- Next in thread
- Step in TB
- Step in Constraint Solver
- Restart
- Dump emulation file
- Breakpoint manager
- Go to active file/line
- Quit simulation

# Verdi Interactive Debug Views

- Instance/Declaration/Signal List
- Stack/Local
- Class/Object/Member
- Source
- Watch
- Dump
- Waveform
- Console

# Verdi Interactive Debug Views

**Instance**

Hierarchy	Module
cosimTFTemplate	cosimTFTemplate
cosimTimeTemplate	cosimTimeTemplate
cosimZcei	cosimZcei
dummy_zcei_top	dummy_zcei_top
dut_top	dut_top
u_addr	addr
u_cnt	cnt
u_axi_if	axi_lite_interface
ZFWC_DFLT_GRP	
top	top
u_addr	tb_addr
u_cnt	tb_cnt
uvm_custom_install_recording	uvm_custom_install_rec...
unnamed\$\$_0	
getenv	
set_recording_detail_file	
vcs_begin_tr	
vcs_check_handle_kind	
vcs_create_stream	

**Signal\_List**

Signal	Value	Type
dut_top		
ack	0(Dumped)	bit
aresetn	0(Dumped)	bit
cnt1[7:0]	0(Dumped)	wire
cnt2[7:0]	0(Dumped)	wire

Instance Declaration Stack Class Object

Signal\_List Local Member

**Stack**

Call Stack	Thread	File:Line
top	0	top.sv:24
top	17	top.sv:38
uvm_config_db#(axi_lite_interfa...)	17	uvm_config_db.svh:164

**Local**

Name	Value	Type
cntxt	null	uvm_component (Class)(Pc)
cs	null	uvm_coreservice_t (Class)
curr_phase	null	uvm_phase (Class)
exists	0	bit
field_name	"axi_vif"	string(Port In)
inst_name	"uvm...top"	string(Port In)
lookup	""	string
p	null	process (Class)
pool	null	uvm_pool#(string,uvm_res
r	null	uvm_resource#(T) (Class)
rstate	""	string
uvm_pkg	NV	package

Instance Declaration Stack Class Object

Signal\_List Local Member

**Class**

Name	Module
axi_lite_common	axi_lite_master_agent_pkg
axi_lite_common	\$unit
axi_lite_common	\$unit
axi_lite_delay_vars	axi_lite_master_agent_pkg
get_t	uvm_pkg
m_uvm_tr_stream_cfg	uvm_pkg
m_uvm_waiter	uvm_pkg
msglog	uvm_custom_install_recording
msglog_msgname	uvm_custom_install_recording
sev_id_struct	uvm_pkg
uvm_callback_iter#(uvm...	uvm_pkg
uvm_callback_iter#(uvm...	uvm_pkg
uvm_callback_iter#(uvm...	uvm_pkg

**Member**

Name	Type	Attribute/Value
Methods		
shift_data_left	Task	Static, Public
shift_data_r...	Task	Static, Public
Variables		
Constraints		

Instance Declaration Stack Class Object

Signal\_List Local Member

**Object**

Hierarchy	Class Type	Object Id	Create Time
dut_top			
u_axi_if			
uvm_custom_install_recording			
_vcs_catcher	vcs_smartl...	@1	0
clp	uvm_cmdli...	null	
cs	uvm_cores...	null	
msglog			
uvm_vcs_tr_stream			
vcs_db	uvm_vcs_t...	null	
uvm_custom_install_verdi_r...			
uvm_pkg			
build_ph	uvm_phase	@2	0
check_ph	uvm_phase	@8	0
connect_ph	uvm_phase	@3	0

**Member**

Name	Type	Attribute/Value
Information		
Create Thre...	int	10
Create Time	time	0
Size	byte	674
Methods		
Variables		
m_end_node	Class uvm...	null
m_executin...	bit[*]	Size: 0
m_imp	Class uvm...	@1
m_jump_bkwd	bit	0
m_jump_fwd	bit	0
m_jump_ph...	Class uvm...	null
m_num_pro...	int	0
m_parent	Class uvm...	@1
m_phase_h...	\mailbox#(...	Size: 64

Instance Declaration Stack Class Object

Signal\_List Local Member

# Verdi Source and Watch

- View HW and SW source code
- Set breakpoint
- Add signal to watch
- Add signal to dump file

```
*Src1:top/slowfs/vgverdirnd1/sbchen/simXL/simXL_AXI_lite_UTF/common/hdl/top/top.sv
29
30
31 wire [7:0] tb_cnt1;
32 wire [7:0] tb_cnt2;
33
34 tb_addr u_addr(.tb_Ai(tb_cnt1[7:0]),.Bi(tb_cnt2[7:0]));
35 tb_cnt u_cnt(.rst_n(tb_aresetn),.aclk(tb_aclk),.cnt1(tb_cnt1[7:0]),.cnt2(tb_cnt2[7:0]))
36
37
38 initial begin
39     uvm_config_db #(virtual axi_lite_interface)::set(null, "uvm_test_top", "axi_vif",
40 dut_top.u_axi_if);
41     run_test ();
42 end
43
44 initial begin
45     $hw_read(dut_top.u_cnt.cnt2);
46     $hw_read(dut_top.u_cnt.cnt1);
47     $hw_read(dut_top.u_addr.Co);
48     $hw_force(dut_top.u_addr.Co);
49     $hw_force(dut_top.u_cnt.cnt1);
50     $hw_force(dut_top.u_cnt.cnt2);
51     tb_aresetn = ~dut_top.aresetn;
52     $fsdbDumpvars("+fsdbfile+test.fsdb");
53 end
54
55 initial begin
56     tb_aclk = 1'b0;
57     forever begin
58         #10000;
59     end
60 end
```

Watch			
Filter String			
Search String			
Watch 1			
Name	Value	Type	Scope
tb_Ai[7:0]	'b00000000	Wire(Port In)	Static(top.u_addr)
cnt2[7:0]	'bxxxxxxx	reg(Port Out)	Static(dut_top.u_cnt)
cnt1[7:0]	'bxxxxxxx	reg(Port Out)	Static(dut_top.u_cnt)
aclk	0	Wire(Port In)	Static(top.u_cnt)
uvm_pkg.uvm_build_phase@1	@1	uvm_b..lass	Object(uvm_pkg.uvm_build_phase@1)
rst_n	NF	Wire(Port In)	Static(dut_top.u_cnt)
aclk	NF	Wire(Port In)	Static(dut_top.u_cnt)

# Dump Emulation File

- No inter.fsdb dump by default, use emulation dump dialog to dump ztdb file.

Emulation Dump (on vgzebut21)

File

File:  Type: fwc

Signal

☒ Value Set: ZFWC\_DFLT\_GRP

☐ Signal:

☐ Instance:  Depth: All

Current File: fwc\_file.ztdb

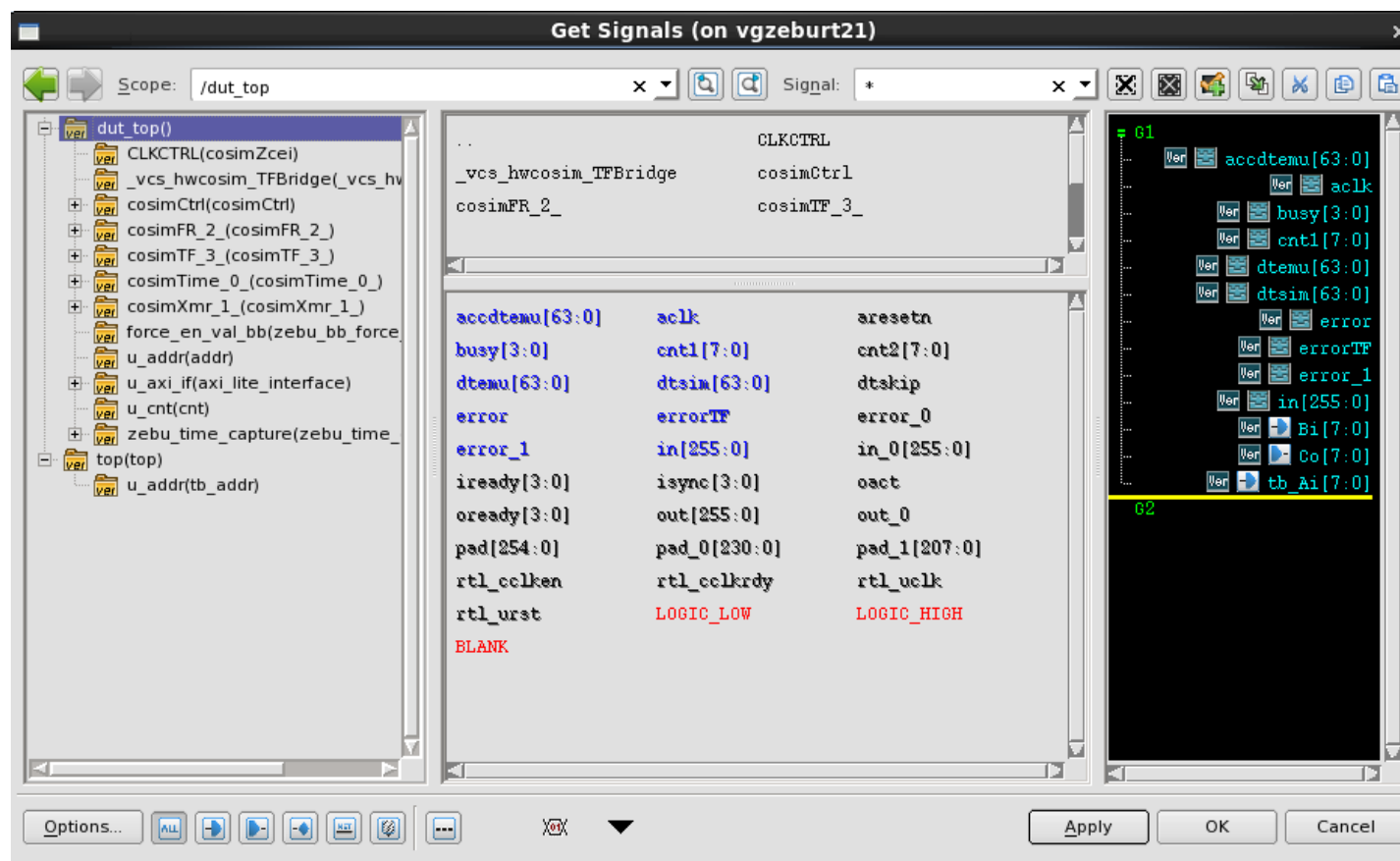
File List

ID	Type	File	Status	Dump On	Finish
ZTDB0	fwc	fwc_file.ztdb	on	<input checked="" type="checkbox"/>	<input type="button" value="X"/>

- Add emulation Dump file with type
- Add hardware value set into file
- Add software signal/instance into file
- Dump on/off
- Dump close

# Open ZTDB file in Waveform

Unify dumping. HW and SW signals are in one ztdb file





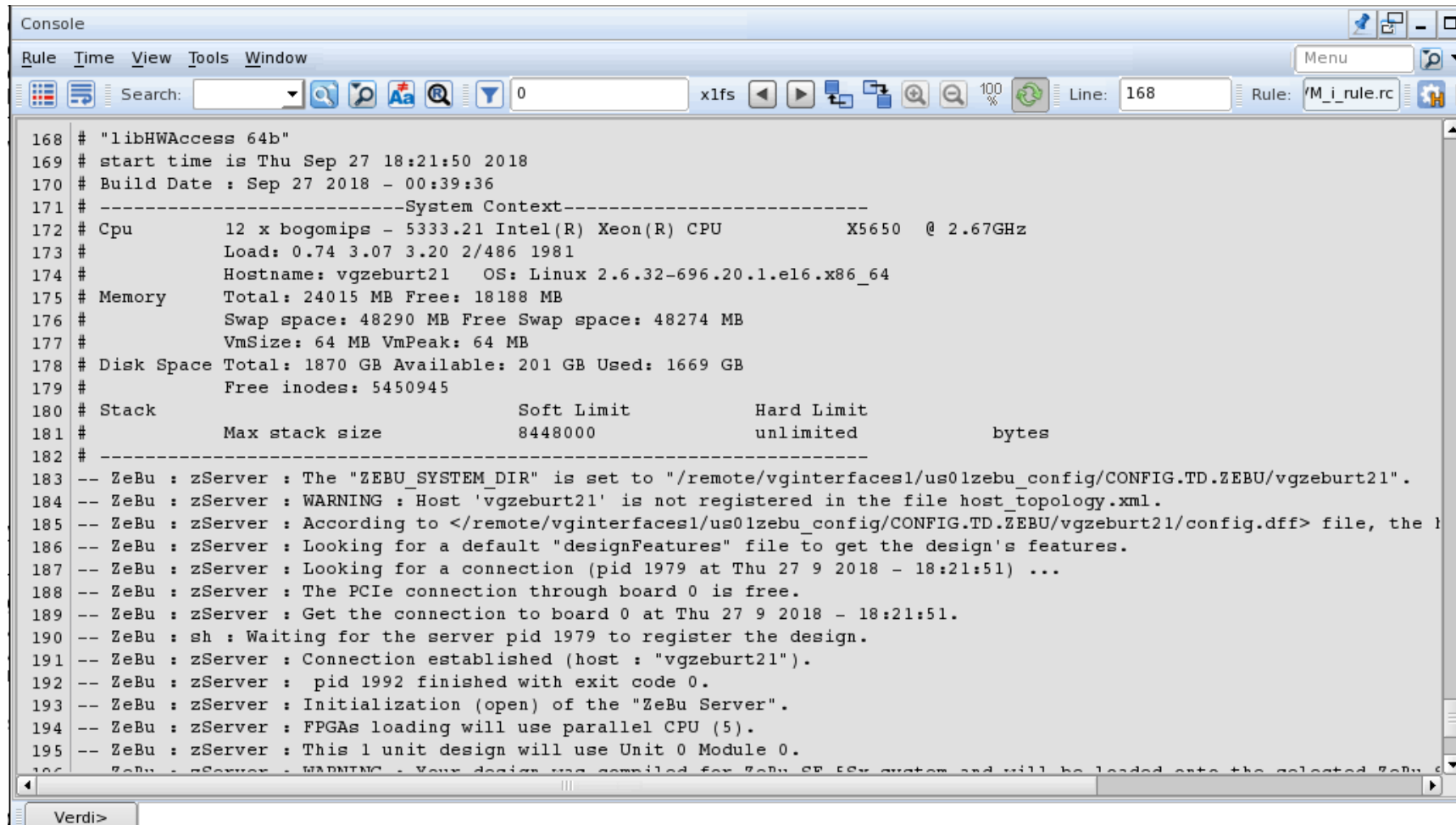
# Open ZTDB file in Waveform (cont'd)

The screenshot displays the VeriTraceMain application interface. The top window shows the project hierarchy and source code for the file `master_bfm_inst`. The bottom window shows the waveform for the same file, with a time scale of 0 to 8,000,000 ps. The waveform displays various signals, including `ack`, `aresetn`, `vdata`, `wlast`, `wready`, `wstrb`, `vvalid`, `awaddr`, `awburst`, `awlen`, `awready`, `awsize`, `awvalid`, and `rdata`. The signals are color-coded and show their state over time. The `Message` pane at the bottom shows the selected file `dut_top.master_bfm_inst.vvalid`.



# Console View

- Display simulation acceleration output in console window



The screenshot shows the Verdi Console window with a menu bar (Rule, Time, View, Tools, Window) and a toolbar. The console displays the following output:

```

168 # "libHWAccess 64b"
169 # start time is Thu Sep 27 18:21:50 2018
170 # Build Date : Sep 27 2018 - 00:39:36
171 # -----System Context-----
172 # Cpu      12 x bogomips - 5333.21 Intel(R) Xeon(R) CPU          X5650   @ 2.67GHz
173 #         Load: 0.74 3.07 3.20 2/486 1981
174 #         Hostname: vgzeburt21   OS: Linux 2.6.32-696.20.1.el6.x86_64
175 # Memory   Total: 24015 MB Free: 18188 MB
176 #         Swap space: 48290 MB Free Swap space: 48274 MB
177 #         VmSize: 64 MB VmPeak: 64 MB
178 # Disk Space Total: 1870 GB Available: 201 GB Used: 1669 GB
179 #         Free inodes: 5450945
180 # Stack
181 #         Max stack size      Soft Limit      Hard Limit      bytes
182 # -----
183 -- ZeBu : zServer : The "ZEBU_SYSTEM_DIR" is set to "/remote/vginterfaces1/us01zebu_config/CONFIG.TD.ZEBU/vgzeburt21".
184 -- ZeBu : zServer : WARNING : Host 'vgzeburt21' is not registered in the file host_topology.xml.
185 -- ZeBu : zServer : According to </remote/vginterfaces1/us01zebu_config/CONFIG.TD.ZEBU/vgzeburt21/config.dff> file, the
186 -- ZeBu : zServer : Looking for a default "designFeatures" file to get the design's features.
187 -- ZeBu : zServer : Looking for a connection (pid 1979 at Thu 27 9 2018 - 18:21:51) ...
188 -- ZeBu : zServer : The PCIe connection through board 0 is free.
189 -- ZeBu : zServer : Get the connection to board 0 at Thu 27 9 2018 - 18:21:51.
190 -- ZeBu : sh : Waiting for the server pid 1979 to register the design.
191 -- ZeBu : zServer : Connection established (host : "vgzeburt21").
192 -- ZeBu : zServer : pid 1992 finished with exit code 0.
193 -- ZeBu : zServer : Initialization (open) of the "ZeBu Server".
194 -- ZeBu : zServer : FPGAs loading will use parallel CPU (5).
195 -- ZeBu : zServer : This 1 unit design will use Unit 0 Module 0.
196 -- ZeBu : zServer : WARNING : Your design was compiled for ZeBu SE 5.5x system and will be loaded onto the selected ZeBu s

```

Verdi>

# Debugging/Profiling

- Simulation acceleration uses many Zebu debug methodologies
  - zRun can be used to run and dump waveforms and control Zebu normally
    - zRun -testbench “./simv <Run\_Time\_Switches>” ...
- Functional Debug using UCLI preferred to zRun
  - UCLI for VCS is enhanced to create simulator like interface for Verification Engineers
  - The UCLI will provide following functionality
    - Dumping waveforms using dynamic\_probe/FWC/QIWC
    - Able to force/deposit/watch/get signal in HW.
    - Able to set breakpoint on signal change values in HW.
    - Dumping/Reading from Zebu Memory.
- Performance Debug
  - VCS Profiler to optimize time spend in testbench.
    - ./simv -simprofile <other switches>
    - profrpt -view time\_all -format all simprofile\_dir/ -dut hw\_top
    - profileReport/TimeSummary.txt
      - User readable profile file is generated( shows DUT% & other TB components)
  - Simulation acceleration profiler
    - For SIGNAL acceleration It will be primarily used to find communication overhead.
    - It can help identify active channels which can benefit from AVIPs.

# Compiling for Debug (UTF file)

- Enabling UCLI signal control from UTF file
  - `zforce -rtlname <Zebu_mapped_signal>`
    - This will enable force/release from command line
  - `zinject -rtlname < Zebu_mapped_signal>`
    - This UTF command will make transformations to enable deposit signal values.
  - `probe_signals -type dynamic -rtlname < Zebu_mapped_signal>`
    - This UTF command will allow Reading/Watching signal values in UTF commands.
- Value Sets for QIWC/FWC are defined similar to Zebu Flow
  - `(*fwc*) $dumpvars(0, hw_top.DUMP.GRP2);`
  - `(*qiwc*) $dumpvars(0, hw_top.DUMP.GRP6);`
- `zSelectProbes`
  - `zSelectProbes` is used to select signals for dynamic probing(ReadBack)
  - No change from regular Zebu flow

# UCLI Commands

- Simulation acceleration supports TestBench/DUT waveform dumping at same time.(New capability)
  - HW Waveforms are captured in ZTDB and Testbench in fsdb file concurrently.
  - Verdi provide ways to view the waveforms together with a aligned timestamp.
- `dump -file <FILE> -type fwc|dynamic_probe -driverClk`
  - It will create a waveform database directory named <FILE> and also create a test bench side waveform simxl.fsdb inside <FILE>.
  - Without `-driverClk` option, It uses `tickClk` for sampling.
- `dump -add <list_of_nids> -fid <FID>`
  - Add signals into test bench side simxl.fsdb when the given FID is ZTDB FID
- `dump -add_value_set <value_set> -fid <FID>`
  - Add <value\_set> to a given <FID>. Multiple value set can be added
- `dump -enable/-disable -fid <FID>`
  - Enable/Disable dumping
- `dump -close <FID>`
  - Close the Dump file
- `dump -flush <FID>`
  - Flush data to dump file
- `dump -load_selection`
  - Loads the `zSelectProbes` before enabling dump for readback.

# UCLI Commands (cont'd)

- Command: run
  - Supports running for relative or absolute time.
    - run 20ns
    - run –absolute 5us
  - Supports running till a HW/SW trigger.
    - run –change tb\_signal
      - No special compile time directive required
    - run –change dut\_signal
      - Must be enabled using probe\_signals command.
- Command : show
  - show -value tb\_signal
    - No special compile time directive required
  - show -value hw\_signal
    - Must be enabled using probe\_signals command at compile time in UTF file.

# UCLI Commands (cont'd)

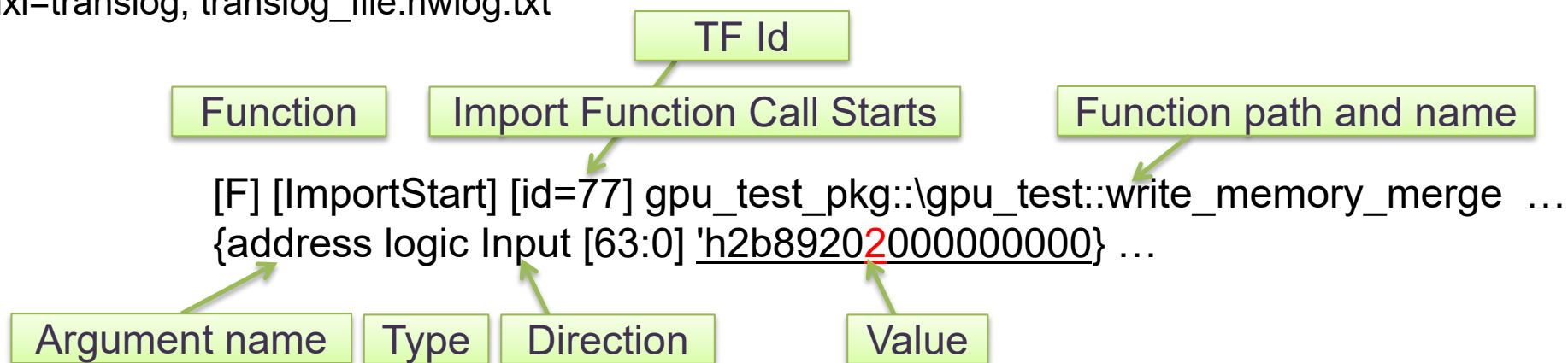
- `force <sw_signal> <value>`
  - No special directive required for forcing SW signal.
- `force <hw_signal> <value>`
  - HW force must be enabled by “zforce” UTF command.
- `release <sw_signal>`
  - No special directive required for releasing SW signal.
- `release <hw_signal>`
  - HW force/release must be enabled by “zforce” UTF command
- `force –deposit <sw_signal> <value>`
  - No special directive required for depositing SW signal.
- `force –deposit <hw_signal> <value>`
  - HW deposit must be enabled by “zinject” UTF command.

# UCLI Commands (cont'd)

- `get u_duv.ctrl -radix b`
  - Get Signal with `-radix b`(binary)/`d`(decimal) or `h`(hex)
- `memory -read|-write -file <fname> [-radix <radix>] [-start start_address] [-end end_address]`
  - Load/write memory values from/to files, or initialize memory with given value
- `restart`
  - Restart tool execution; UCLI will return to time zero
- `senv value_sets`
  - returns all value sets existing in design compilation.
- `senv driver_clk_frequency`
  - returns driver clock frequency in kHz

# Profiler and Translog Dump

- VCS communication profiler
  - ./simv -simxl=profile <Other options>
  - Enables communication profiling and dumps simxlProfile.txt at the end of test.
  - It reports Elapsed Time/Driver Clock Cycles/Active Signals/Memory Read-Write Calls/TimeStamp Synchronization/DPI-Verilog task calls.
  - The purpose is to reduce synchronizations between TB-DUT by moving towards a TBA(Transaction Based Acceleration) and identify active interactions.
- Communication overhead in transferring signal/memory values etc.
- Enabling dumping of detailed continuous I/O communication between HW/SW.
- ./simv -simxl=translog, translog\_file:hwlog.txt





# Increasing Acceleration Performance

- Usage of HW clock signals in SW can be optimized under the hood to reduce communication overhead.
  - When waiting for a fixed number of cycles from many different places.
  - For example, usage of 50 difference places in testbench of (repeat (N) @(posedge top.chip.clk))
  - Even though communication is at signal level implementation is optimized to communicate when exceeding the repeat count.
- Signals that change infrequently, should be kept at signal communication over communicating them through a code implementation.
  - Signals between HW and SW are only communicated when they change their values. Therefore it is worth keeping the communication at signal level and let the tool handle when appropriate to communicate

# AXI EXAMPLE

# A Typical UVM Environment

- Signal based connection (through virtual interfaces) in UVM
  - Driving and sampling functionalities are usually implemented in bus functional model (BFM) tasks inside drivers or monitors

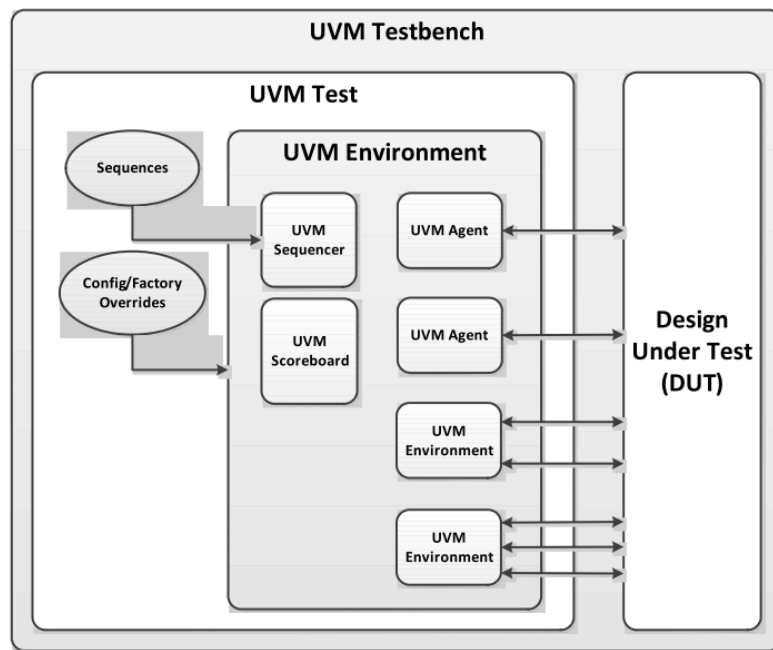


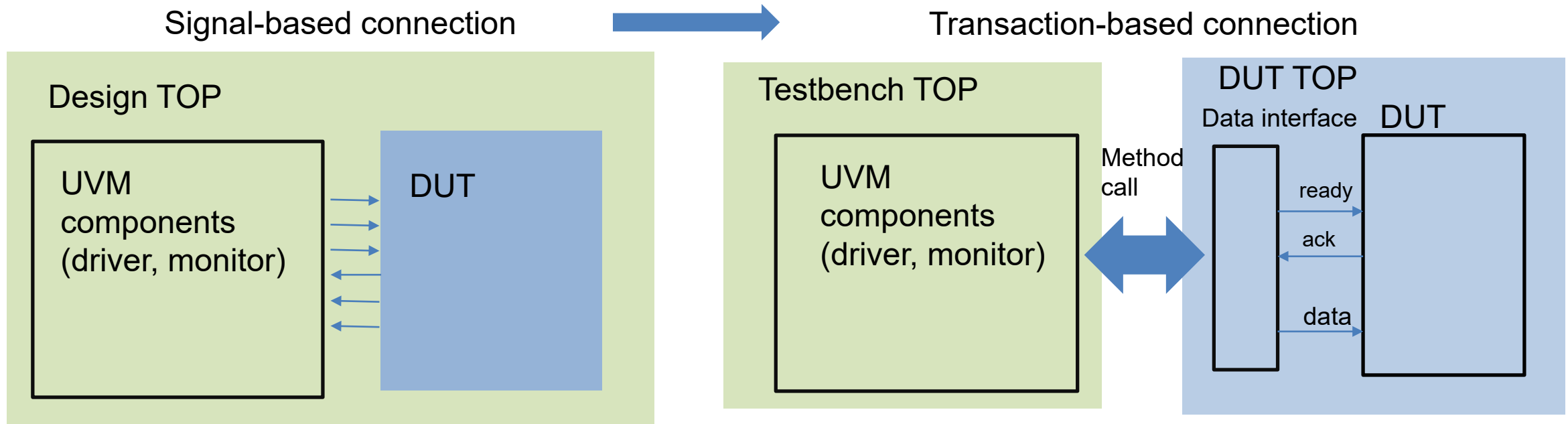
Figure 1—Typical UVM Testbench Architecture

```
class xbus_master_driver extends uvm_driver #(xbus_transfer);

    // The virtual interface used to drive and view HDL signals.
    virtual xbus_if xmi;
    // get_and_drive
    virtual protected task get_and_drive();
    @(negedge xmi.sig_reset);
    forever begin
        @(posedge xmi.sig_clock);
        seq_item_port.get_next_item(req);
        ...
        xmi.sig_addr <= trans.addr;
        ...
        xmi.sig_size <= 2'bz;
        ...
    end
endtask : get_and_drive
```

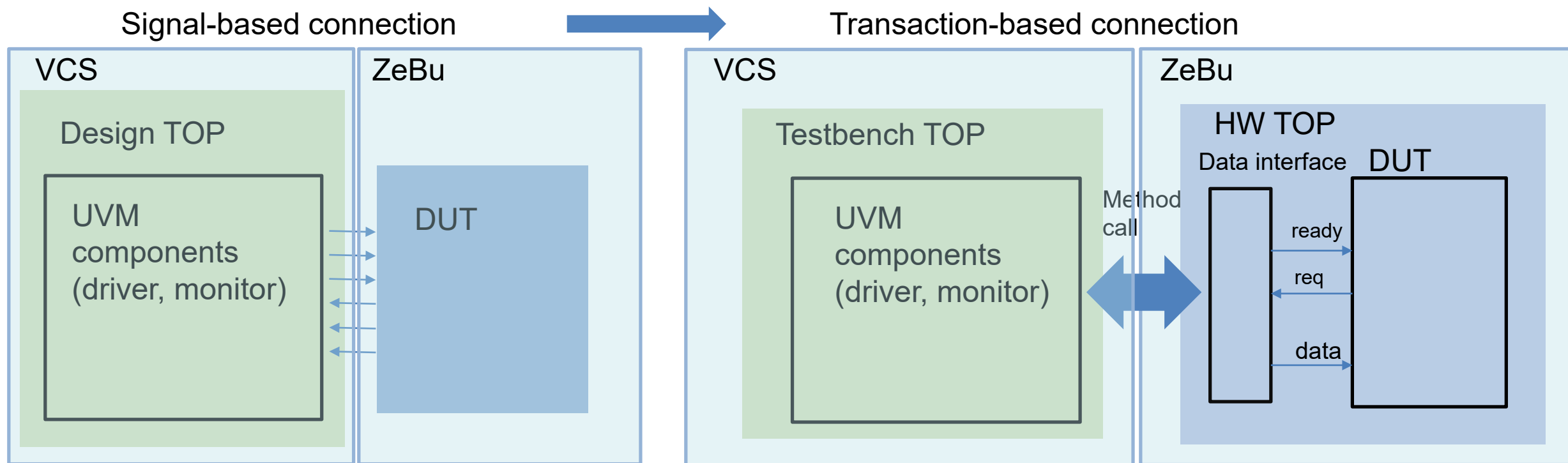
# Moving BFM to Interfaces/Modules

- Isolating transaction generation/post-processing (scoreboards, etc) from transaction driving/sampling
- BFMs can be behavioral level code (for simulation) or RTL code (for synthesis)
- Transactions are passed between testbench and interfaces using the method calls



# Signal-based vs. Transaction-based

- Simulation acceleration supports both usages
- Transaction based connection provides better performance
  - Less communication events between simulator and emulator



# Method Call: Export/Import

- Export call
  - testbench calls interface through virtual interface
    - `vif.write(transaction)`
- Import call
  - interface calls testbench (UVM component) methods through object handle
    - `master_handle.get(transaction)`

# Data Class: Transaction or Configuration

- For objects to be passed between VCS and ZeBu we need to convert them to a struct type.

```
typedef struct {  
    logic[3:0] in1;  
    logic[3:0] in2;  
    logic[4:0] out1;  
} data_s;  
class data_c;  
    rand logic[3:0] in1;  
    rand logic[3:0] in2;  
    rand logic[4:0] out1;  
  
    function void copy_to_item(input data_s data_t);  
        this.in1 = data_t.in1;  
        this.in2 = data_t.in2;  
        this.out1 = data_t.out1;  
    endfunction  
    function void copy_to_struct(output data_s data_t);  
        data_t.in1 = this.in1;  
        data_t.in2 = this.in2;  
        data_t.out1 = this.out1;  
    endfunction  
    function void print();  
        $display($stime,,"in1: %x  in2: %x",this.in1,this.in2);  
    endfunction  
endclass
```

- The driver/monitor is running inside VCS but the object handle will be passed to data interface so data interface can call the driver/monitor method
  - import call

# Driver/Monitor Class

```
class mst_c;
    data_c data;
    virtual data_itf vif;
    function void connect();
        vif.xtor_register(this); //register this xtor to
data interface
    endfunction
    //data interface will call this task to get a
transaction
    task get(output data_s data_t);
        seq_item_port.get_next_item(req_t);
        ...
        req_t.copy_to_struct(data_t);
    endtask
    task put(input data_s _rsp);
        data_c rsp_t;
        ...
        rsp_t.copy_to_item(_rsp);...
        seq_item_port.item_done(rsp_t);
    endtask
endclass
```



# Data Interface

- The data interface gets/puts transactions from/to UVM driver/monitor
  - As in the typical UVM environment, the driver still pulls transactions from UVM sequence and the monitor still puts the transactions to analysis components
- The interface has other methods for the communication with testbench, for example to register a driver/monitor to this interface instance
  - xtor\_register

```
interface data_itf(  
    input logic clk,  
    input logic reset,  
    output logic data_ready,  
    output data_s data,  
    input logic data_req  
);  
    mst_c mst;    //xtor handle  
    function void xtor_register(input mst_c  
mst_t);  
        mst = mst_t;  
    endfunction  
    always @(posedge clk) begin  
        if(reset) begin  
            data_ready <= 1'b0;  
            next_state <= ST_GET_DATA;  
        end  
        else begin  
            case(current_state)  
                ST_GET_DATA:  
                    begin  
                        mst.get(data);  
                        data_ready <= 1'b1;  
                        if(data_req) begin  
endinterface
```

# BFM Module

- BFM module is a synthesizable module to drive the transaction to DUT or collect the transaction from DUT
  - BFM is optional since users can also implement driving/ monitoring methods inside data interface
- Data interface puts/gets the data to/from BFM module
  - Same data interface instance will be connected to driver/monitor virtual interface

# BFM Module

```
module mst_bfm(  
    input logic data_ready,  
    output logic data_req,  
    input data_s data,  
    //DUT  
    itf itf_p  
);  
always @(posedge itf_p.clk) begin  
    if(itf_p.reset) begin  
        next_state <= ST_DRIVE;  
        data_req <= 1'b0;...  
    end  
    else begin  
        case (current_state)  
            ST_DRIVE:  
                begin  
                    if(data_ready) begin //data ready, drive data to DUT  
                        itf_p.valid <= 1'b1;  
                        itf_p.in1 <= data.in1;  
                        itf_p.in2 <= data.in2;  
                        if(itf_p.ack) begin //dut ack, get next data  
                            next_state <= ST_DRIVE;  
                        end  
                    end  
                end  
        end  
    end  
end
```

# Behavioral Compilation

- UVM driver/monitor BFM tasks are usually implemented with “behavioral code”
- Behavioral Compilation
  - Hardware synthesis of behavioral code

```
class xbus_master_driver extends uvm_driver #(xbus_transfer);  
  
    // The virtual interface used to drive and view HDL signals.  
    virtual xbus_if xmi;  
    // get_and_drive  
    virtual protected task get_and_drive();  
    @(negedge xmi.sig_reset);  
    forever begin  
        @(posedge xmi.sig_clock);  
        seq_item_port.get_next_item(req);  
        ...  
        xmi.sig_addr <= trans.addr;  
        ...  
        xmi.sig_size <= 2'bz;  
        ...  
    end  
endtask : get_and_drive
```

# Behavioral Compilation

- Initial block and # delay
- Bounded and Unbounded loops

```
while (1) begin
    @(posedge clk1)
    c <= c + 1;
end

for (i=0;i<128;i=i+1) begin
    @(posedge clk);
    mem[i] = 0;
end
```

```
bit aclk;
bit aresetn;
initial begin
    aresetn = 1'b0;
    #5000;
    aresetn = 1'b1;
end
initial begin
    aclk <= 1'b0;
    forever begin
        aclk <= 1'b1;
        #1000;
        aclk <= 1'b0;
        #1000;
    end
```

# Behavioral Compilation

- Multiple clocks or edge expressions in the same process
- Wait statement
- Named events

```
always @(posedge clk) begin
    a = 0;
    @(negedge reset);
    a = 1;
    @(negedge clk);
    a = 2;
end

initial begin
    wait(reset==1'b0);

    event my_event;
    initial -> my_event;
    always begin
        @(my_event);
        a <= b;
    end
```

# Behavioral Compilation

- Clocking blocks

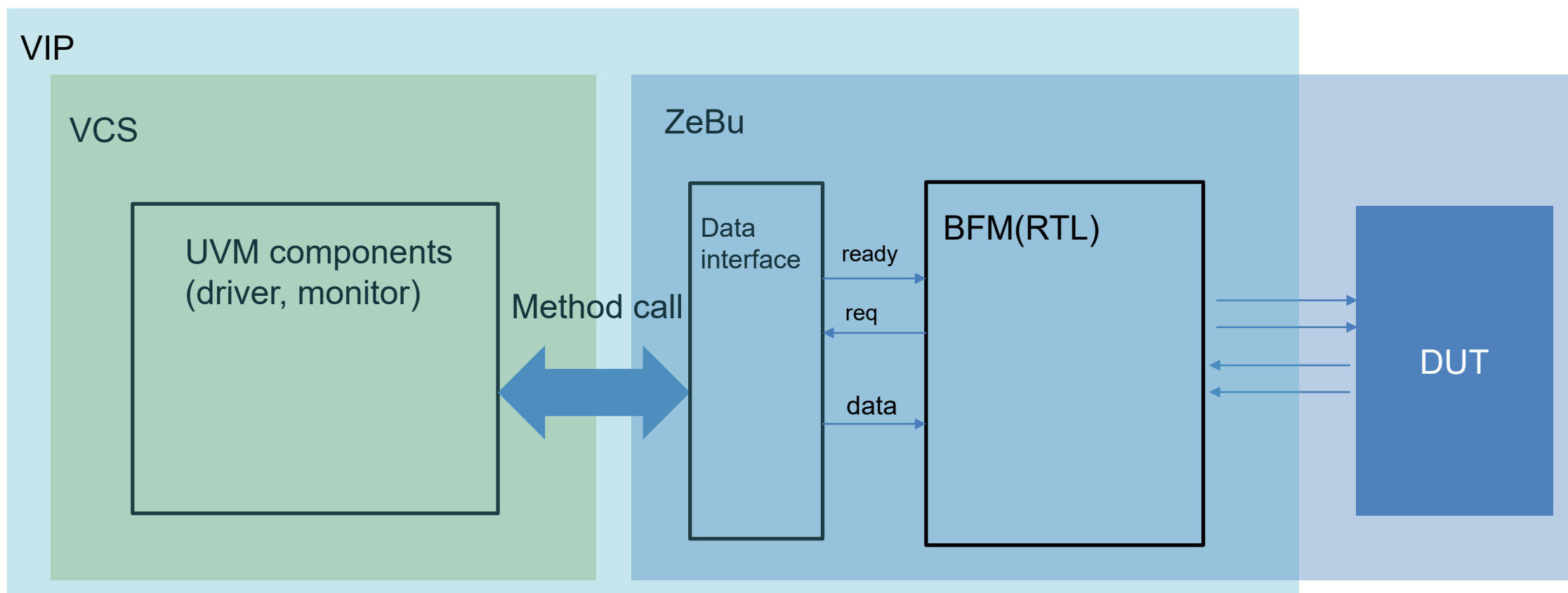
```
clocking cb @(posedge aclk);  
  input  aresetn ;  
  input  awaddr ;  
endclocking  
initial begin  
  @(cb);  
  A <= cb.awaddr;  
end
```

- Fork/Join
  - fork/join
  - fork/join\_none
  - fork/join\_any

```
fork  
  do_A();  
  do_B();  
join
```

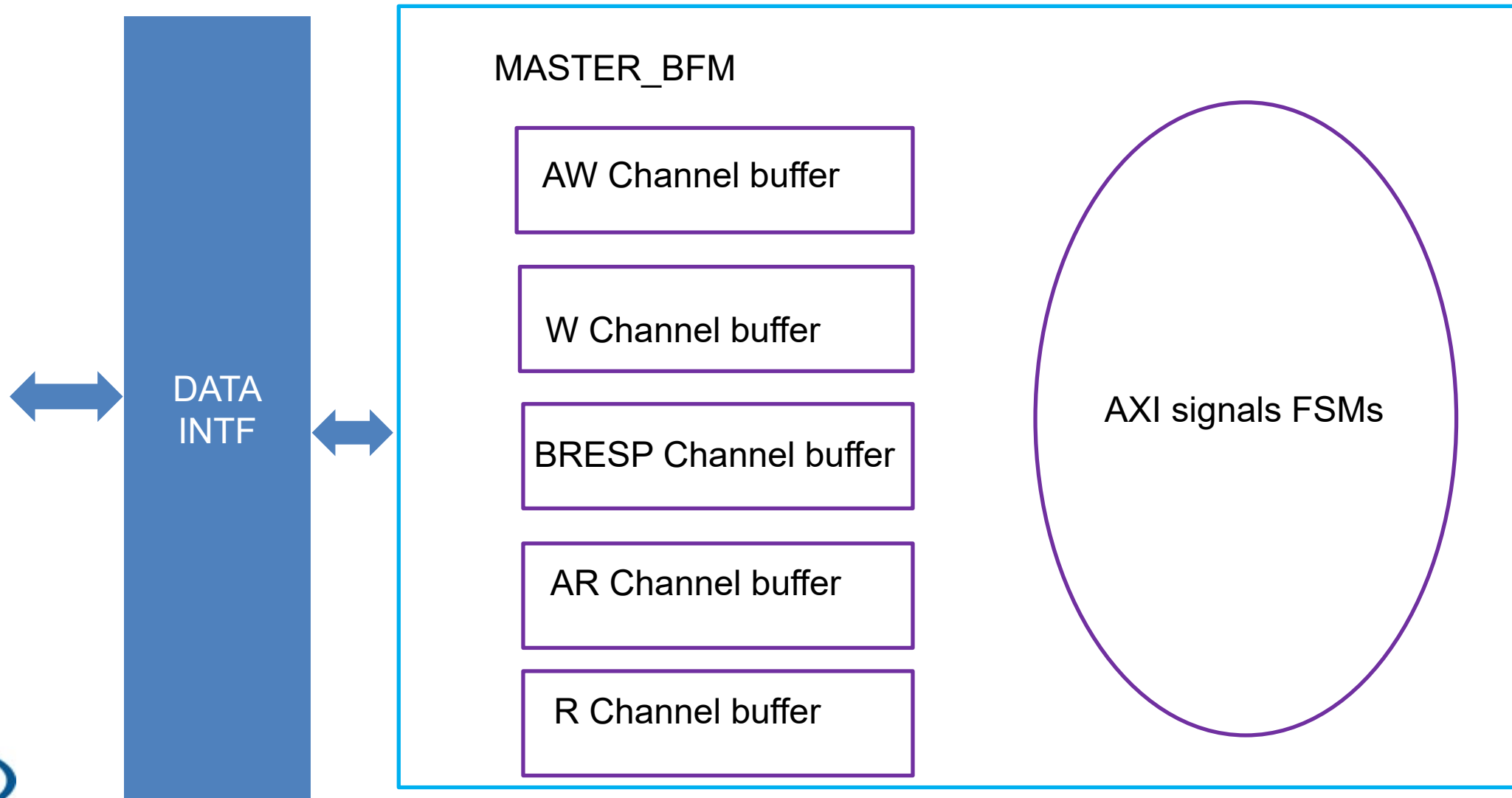
# AXI 3/4 VIP Environment

- Transaction based connection in a UVM based environment
  - UVM testbench (running with VCS)
  - Interface and BFM (running with ZeBu)
  - UVM drivers/monitors communicate with ZeBu through method call

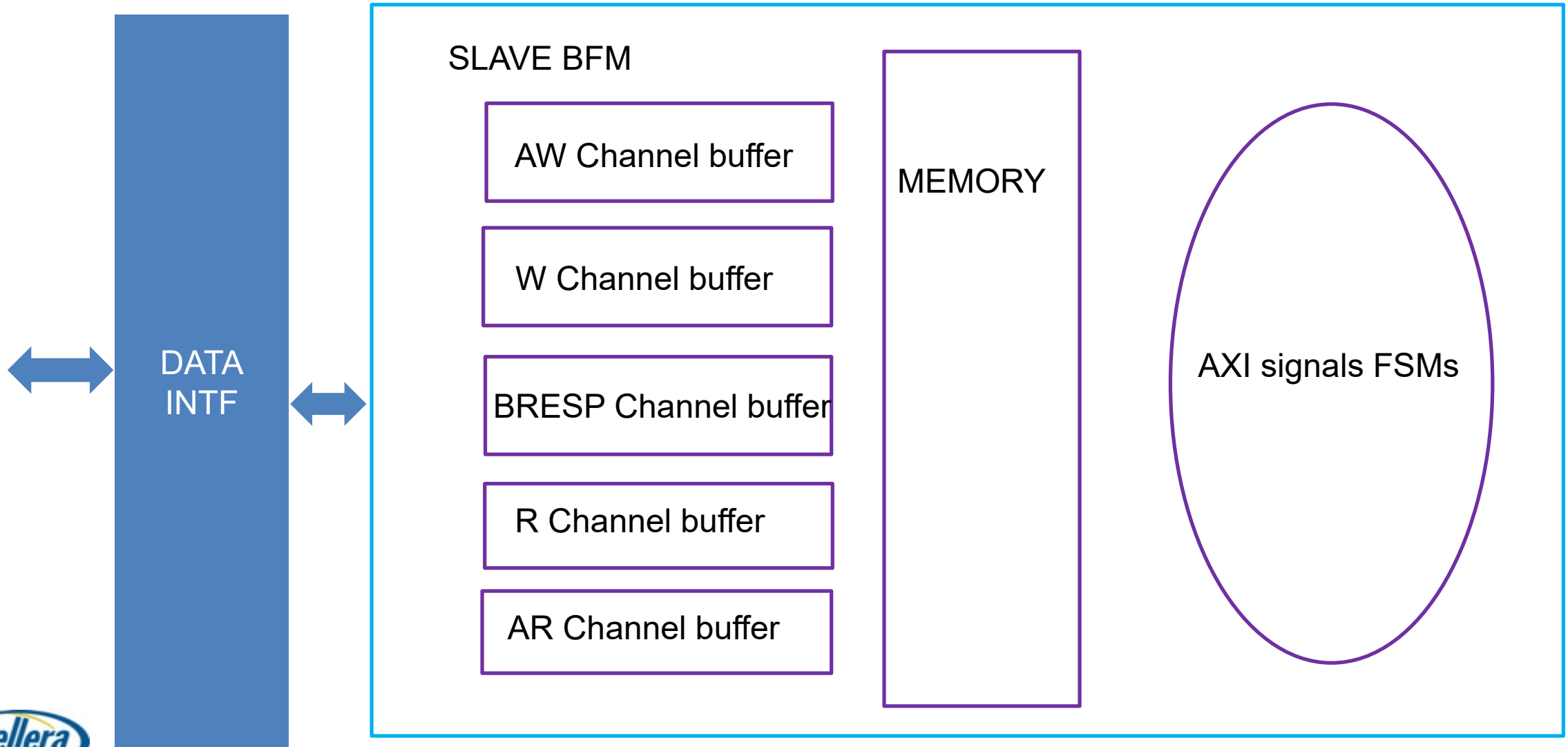




# AXI Master/Monitor



# AXI Slave



# Data Interfaces

- Get AXI transactions from AXI UVM master component
- Put AXI transactions to AXI UVM slave or monitor components
- When needed, testbench code can also pass the configuration data to data interface through the xtor\_register call

```
//slave data interface
function void xtor_register(input axi_slave_drv slv_t,input axi_common_pkg::slave_cfg
_slave_cfg_data );
    slv = slv_t;
    slave_cfg_data = _slave_cfg_data;
endfunction
always @(posedge clk) begin
    if(reset!=0) begin
        if (write_mem_tran_in_done) slv.put(write_mem_tran_in,0);
    ...
end

//master data interface
always @(posedge clk) begin
    ...
    else begin
        if(data_req==1'b1) begin
            mst.get(data_is_valid,data_new);
```

# Parameterized Buffer Size

- BFM module has a parameter to specify how many outstanding transactions can be processed in parallel

```
module slave_bfm #(
    parameter bit[3:0] depth=1    //how many transactions can be
    processed in same time
    ...
)

endmodule

module hw_top;
    slave_bfm #(buffer_size,...)
    slave_bfm_inst(.slave_cfg_port(slave_cfg_data),.*);
    ...
endmodule
```

# Single and Burst Read/Write Sequence Tasks

```
virtual task read(input bit [`AXI_MAX_AW-1:0]    addr,
                  output logic [`AXI_MAX_DW-1:0] data,
                  output logic [1:0]              rresp,
                  input                          uvm_sequencer_base seqr,
                  input bit[(`AXI_ID_WIDTH - 1):1] arid=0,
                  input int                       tr_size_in_bytes =
`AXI_MAX_DW/8,
                  input bit[1:0]                  burst_type = 2'b01,
                  input                          uvm_sequence_base parent =
null);
virtual task read_burst(input bit [`AXI_MAX_AW-1:0]    addr,
                        output logic [`AXI_MAX_DW-1:0] data [],
                        input int                       burst_length,
                        output logic [1:0]              rresp [],
                        input                          uvm_sequencer_base seqr,
                        input bit[(`AXI_ID_WIDTH - 1):1] arid=0,
                        input int                       tr_size_in_bytes =
`AXI_MAX_DW/8,
                        input bit[1:0]                  burst_type = 2'b01,
                        input                          uvm_sequence_base parent
= null);
```

# Single and Burst Read/Write Sequence Tasks

```
virtual task write(input bit  [`AXI_MAX_AW-1:0]    addr,
                      input logic [`AXI_MAX_DW-1:0] data,
                      output logic [1:0]            bresp,
                      input                                uvm_sequencer_base seqr,
                      input bit[(`AXI_ID_WIDTH - 1):1] awid=0,
                      input axi_delay_vars_struct    delay_vars=axi_delay_vars_struct'{default:0},
                      input int                        tr_size_in_bytes = `AXI_MAX_DW/8,
                      input bit[1:0]                  burst_type = 2'b01,
                      input                                uvm_sequence_base parent = null);

virtual task write_burst(input bit  [`AXI_MAX_AW-1:0]    addr,
                        input logic [`AXI_MAX_DW-1:0] data [],
                        input int                        burst_length,
                        output logic [1:0]              bresp,
                        input                                uvm_sequencer_base seqr,
                        input bit[(`AXI_ID_WIDTH - 1):1] awid=0,
                        input axi_delay_vars_struct    delay_vars=axi_delay_vars_struct'{default:0},
                        input int                        tr_size_in_bytes = `AXI_MAX_DW/8,
                        input bit[1:0]                  burst_type = 2'b01,
                        input                                uvm_sequence_base parent = null);
```

# Write and Read Test Example

```
wr_seq = axi_master_write_seq::type_id::create("wr_seq");
rd_seq = axi_master_read_seq::type_id::create("rd_seq");

//SIMPLE WRITE READ, one byte each transfer
for(int i = 0; i < 16; i++) begin
    wr_seq.write(32'h00000100+(i*bytelane_num), i+1, resp, env.u_axi_master_agt.sqr,1,,1);
end
repeat(30) @(posedge master_vif.clk);
for(int i = 0; i < 16; i++) begin
    rd_seq.read(32'h00000100+(i*bytelane_num), rd_data_single, resp, env.u_axi_master_agt.sqr,1,1);
end
repeat(30) @(posedge master_vif.clk);

//BURST WRITE and READ
begin
    burst_length=5;
    wr_data = new[burst_length];
    foreach(wr_data[ii]) begin
        wr_data[ii]=ii+5;
    end
    wr_seq.write_burst(32'h1000, wr_data, 5,bresp, env.u_axi_master_agt.sqr,1,,2);
    repeat(10) @(posedge master_vif.clk);
    rd_seq.read_burst(32'h1000, rd_data,5, rresp, env.u_axi_master_agt.sqr,1,2);
    repeat(10) @(posedge master_vif.clk);    //unaligned address
    wr_seq.write_burst(32'h1001, wr_data, 5,bresp, env.u_axi_master_agt.sqr,1,,2);
    repeat(30) @(posedge master_vif.clk);
    rd_seq.read_burst(32'h1001, rd_data,5, rresp, env.u_axi_master_agt.sqr,1,2);
end
```

# Memory Access Debug Message

- AXI slave BFM module puts the received write transaction (both AW channel info and WDATA channel info), received read transaction (AR channel info) and transaction after reading memory (byte enable for each transfer, data read from memory) to AXI slave UVM driver/monitor for the debug purpose.

```
//slave_data_if
always @(posedge clk) begin
    if(reset!=0) begin
        if (write_mem_tran_in_done) slv.put_transaction(write_mem_tran_in,0);
        if (read_mem_tran_in_done) slv.put_transaction(read_mem_tran_in,1);
        if (read_mem_tran_out_done) slv.put_transaction(read_mem_tran_out,2);
    end
end
end
//class axi_slave_drv
function void put_transaction(input axi_seq_item_struct tran_struct, input bit[1:0] which_tran);
    read_mem_tran_out=axi_seq_item::type_id::create("read_mem_tran_out");
    read_mem_tran_out.copy_to_item(tran_struct);
    `uvm_info(get_full_name(),$sformatf("%p",read_mem_tran_out),UVM_DEBUG)
endfunction
```

```
UVM_INFO /slowfs/vgzebucae10/whan/MYAXI/0507/myaxi_0628/src/dv/axi_slave/axi_slave_drv.sv(47) @
50315000: uvm_test_top.env.u_axi_slave_agt.drv [uvm_test_top.env.u_axi_slave_agt.drv] ...
m_leaf_name:"read_mem_tran_out"
addr:'he0000150, data:('{h7, 'hc, 'h11, 'h16, 'h1b} , burst_length:5, burst_type:1,
byte_en:('{hf, 'hf, 'hf, 'hf, 'hf} , tr_size_in_bytes:4,
```



# ZeBu Model Instantiation

- Hardware top module

```
module hw_top();  
    master_data_itf  
master_data_itf_inst(aclk,aresetn,data_ready,data,data_req,...);  
    master_bfm#(buffer_size) master_bfm_inst(data_ready, data, data_req,  
wr_data_valid,..., .*);  
    slave_dut slave(.*);    //AXI signal connections  
    initial begin  
        aclk = 1'b0;  
        forever #5 aclk = ~aclk;  
    end  
  
    initial begin  
        aresetn = 1'b0;  
        repeat(10) @(posedge aclk);  
        aresetn <= 1'b1;    //reset deassertion should be synchronous on the  
        rising edge of aclk  
    end  
endmodule
```

# AXI Example Capabilities

- Different burst types, transfer size, burst lengths
- Separate address/control, data and response phases. Separate read and write channels.
- Support for burst-based transactions with only start address issued.
- Write strobe support to enable sparse data transfer on the write data bus
- Narrow transfer support
- Unaligned address access support.
- Ability to issue multiple outstanding transactions.
- Out of order transaction completion support.
- Support for Write data phase before Write address phase (negative AWVALID to WVALID delay)
- Write data and read data interleaving support.
- Configurable write and read interleave depth.
- Slave and Master support fine grain control of response per address or per transaction (OKEY or SLVERR)

# AXI Example Profile

	Time Spent	%time
SimXL Elapsed Time	455.387 s	100.0
Test Bench Time	438.024 s	96.2
Communication Time	17.363 s	3.8
Synchronization	0.000 ms	0.0/0.0
PIO data packing	17.363 s	100.0/3.8

	Time Spent	%time
SimXL Elapsed Time	265.108 s	100.0
Test Bench Time	264.930 s	99.9
Communication Time	177.462 ms	0.1
Synchronization	0.000 ms	0.0/0.0
TF data packing	177.462 ms	100.0/0.1

## Data Transfer Summary:

Type	Total calls	Total data	Average calls	Average data
Input	14599876	3.481 Gb	1.106	283.152 b
Output	14341377	3.419 Gb	1.086	278.139 b
Force	0	0 b	0.000	0.000 b
SW->HW call	0	0 b	0.000	0.000 b
HW->SW call	0	0 b	0.000	0.000 b
Memory Read	0	0 b	0.000	0.000 b
Memory Write	0	0 b	0.000	0.000 b

SBA

## Data Transfer Summary:

Type	Total calls	Total data	Average calls	Average data
Input	0	0 b	0.000	0.000 b
Output	0	0 b	0.000	0.000 b
Force	0	0 b	0.000	0.000 b
SW->HW call	3	290 b	0.000	0.000 b
HW->SW call	600090	1001.699 Mb	0.045	79.573 b
Memory Read	0	0 b	0.000	0.000 b
Memory Write	0	0 b	0.000	0.000 b

TBA

# AXI Example Profile

- Other profile information

Instance Name: dut\_top.master\_data\_itf\_inst

Total Calls	Direction	Input Size	Output Size	Task ID	Task/Func name	
200000	HW->SW	96 b	1.616 Kb	1	axi_master_pkg::^Gxi_master_drv::get_req	*
99998	HW->SW	1.709 Kb	0 b	5	axi_master_pkg::^Gxi_master_drv::put_tran	*
99998	HW->SW	1.709 Kb	0 b	6	axi_master_pkg::^Gxi_master_drv::put_tran	*
97	HW->SW	96 b	1.616 Kb	0	axi_master_pkg::^Gxi_master_drv::get_req	*
1	HW->SW	96 b	1.616 Kb	3	axi_master_pkg::^Gxi_master_drv::get_req	*
1	SW->HW	64 b	32 b	7	this.vif.xtor_register	
0	HW->SW	96 b	1.616 Kb	2	axi_master_pkg::^Gxi_master_drv::get_req	*
0	HW->SW	96 b	1.616 Kb	4	axi_master_pkg::^Gxi_master_drv::get_req	*

Details of signal changes:

Total Changes	Width	ID	Signal name

Inputs:

13199868	1 b	1	dut_top.bfm_top_inst.aclk
200001	1.614 Kb	3	dut_top.bfm_top_inst.data_in_bits
3	1 b	4	dut_top.bfm_top_inst.data_ready
2	1 b	2	dut_top.bfm_top_inst.aresetn
1	2 b	16	dut_top.bfm_top_inst.slave_cfg_data_bits

# Summary

- Simulation acceleration provides flow for VCS users to take advantage of ZeBu for accelerating their UVM regressions
- VCS supports signal-based and transaction-based acceleration
- By moving UVM driver/monitor BFM tasks to interfaces/modules we can setup a UVM-based emulation friendly environment