2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Innovative Uses of SystemVerilog Bind Statements within Formal Verification

Xiushan Feng and Christopher Starr
Samsung Austin R&D Center

SAMSUNG

accellera SYSTEMS INITIATIVE

Agenda

- Introduction
  - SystemVerilog bind statement
  - Formal verification and SystemVerilog Assertions (SVAs)
- Our use cases of bind statements within formal verification
  - Sequential equivalence checking for clock gating (SEQ)
  - Data-Path Verification (DPV)
  - Formal Property Verification (FPV)

Introduction: SystemVerilog Bind

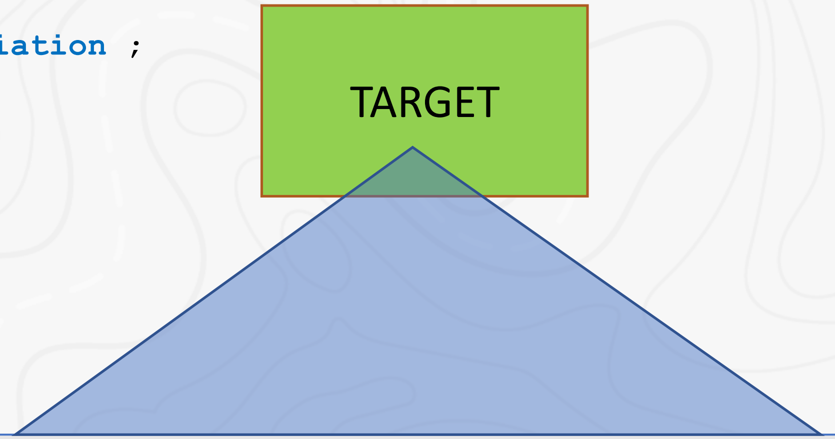
- Syntax

```
bind_directive ::=
    bind bind_target_scope [: bind_target_instance_list] bind_instantiation ;
  |  bind bind_target_instance bind_instantiation ;

bind_instantiation ::=
                        program_instantiation
                      | module_instantiation
                      | interface_instantiation
                      | checker_instantiation
```

- Usage
  - Add verification code into RTL design
  - Verification code can be separated from RTL
  - Very uncommon to be used by RTL for synthesis

scope or instance of
`module|interface`



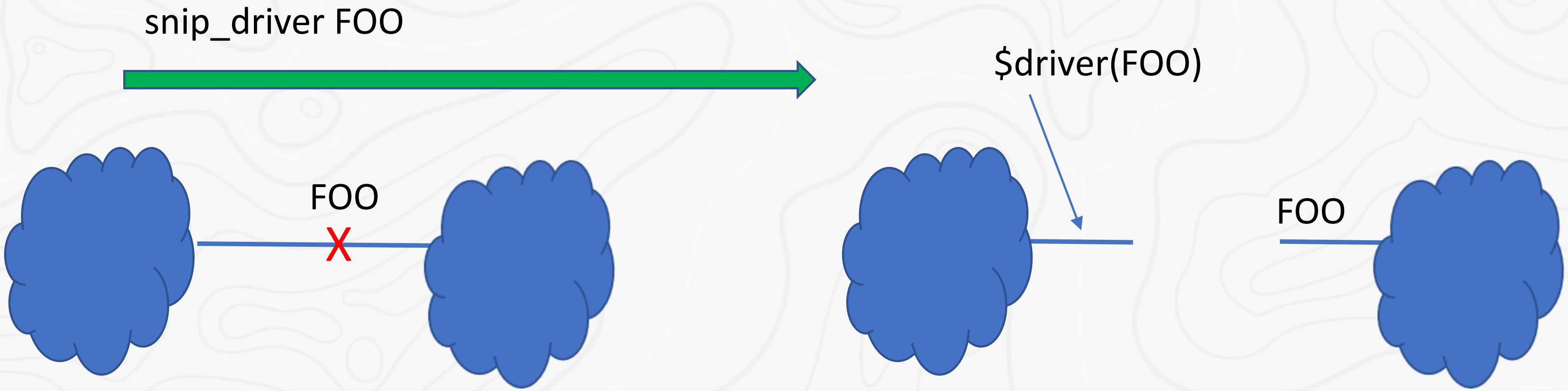

```
module|program|interface|checker FOO
…
end{module|program|interface|checker}
bind TARGET FOO #(…)  FOO_inst (.*);
```

Formal Verification is Assertion-Based

- M, *s*, *a* |= *p* . For all state *s* of a model M (e.g., a circuit), property *p* is true under the assumptions of *a*
- Properties/assumptions are written in assert/assume type assertions
- Not all assertions can be easily created with available design signals
  - RTL modeling is required to simplify SVAs
  - It is convenient to add some extra logic for debug
  - Complicated helper logic should be outside RTL design if possible

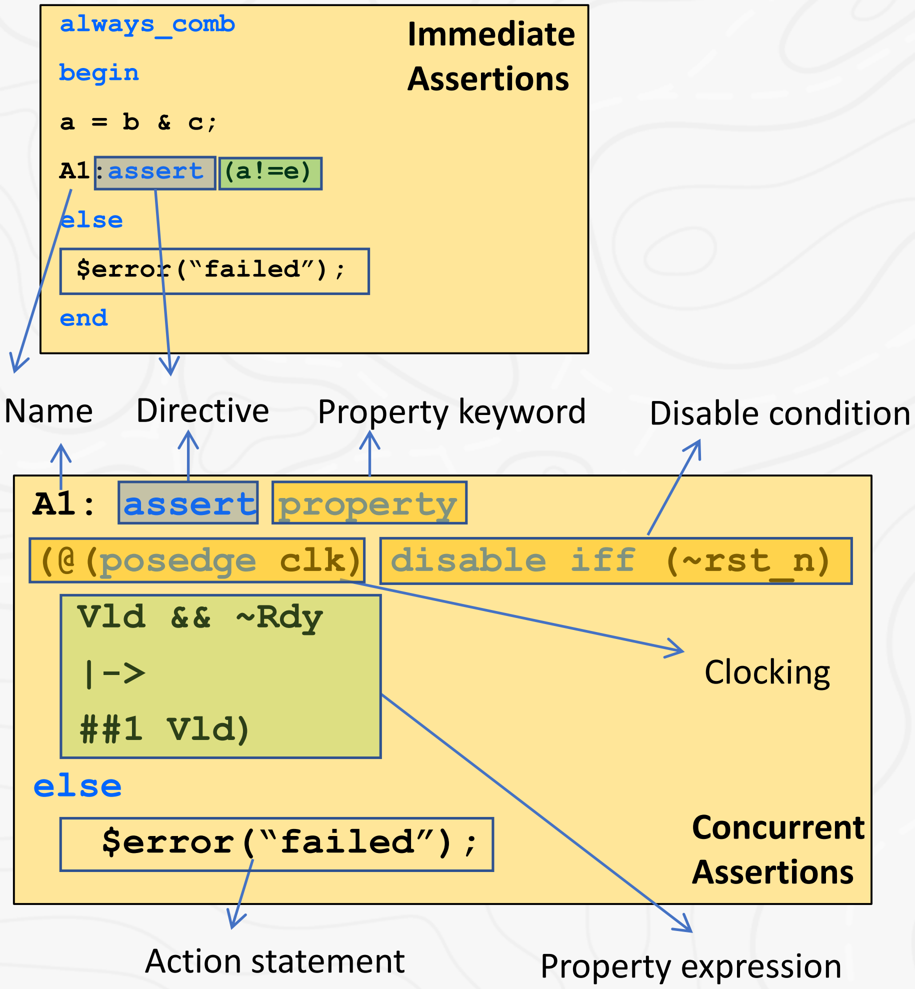
Formal Tools Can Manipulate Designs

- A snip_driver example



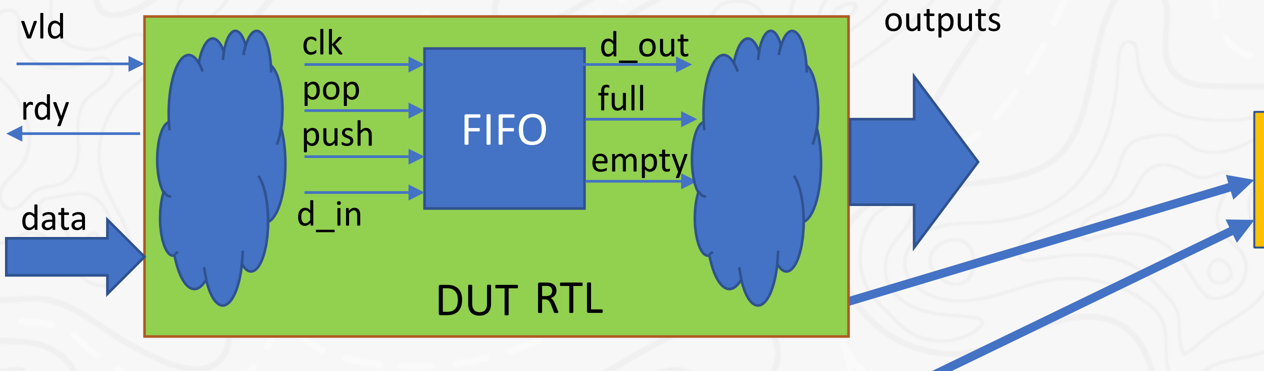
With snip_driver and assumptions, we can manipulate design logic

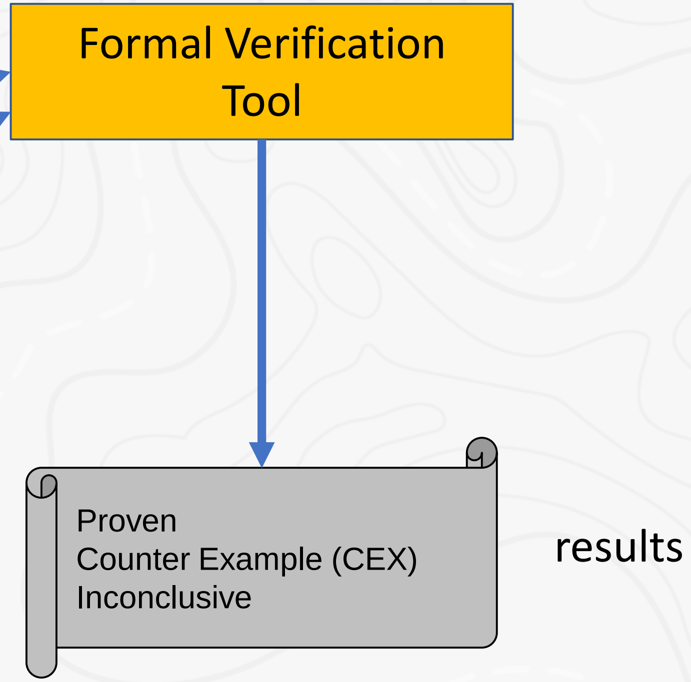
SystemVerilog Assertions



Type	Immediate	Concurrent
Expression Sampling	Simulation event semantics. Glitches will cause issues	Clock semantics (Preponed region) Sample once for one clock event
Evaluation Region	Active/inactive regions. If used inside combo logic, assert #0 to avoid races	Observed region
Intention	Mainly for simulation	Simulation/formal/emulation
Location	Inside **procedural statement** (initial/always/final/task/function)	Inside **module/checker/interface/…**
Assertion Expression	Boolean	Temporal (behavior spans over time)
Other diffs	No “disable iff” No “property” keyword	

Formal Verification with Bind Statements



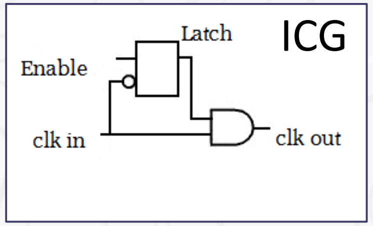


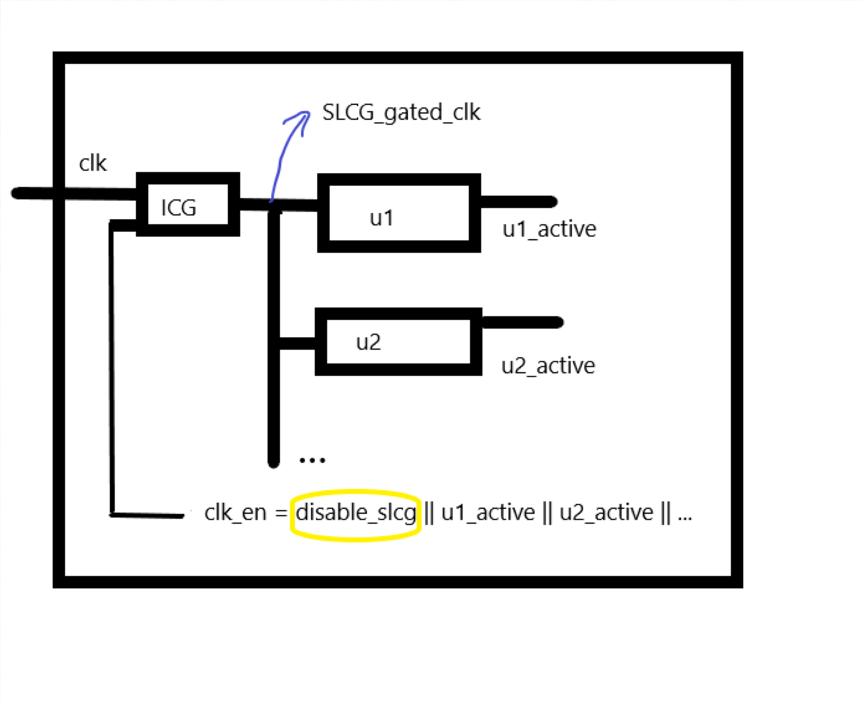

```
module ABV_chk(input vld, rdy, data, …);
default clocking main_clk @(posedge clk); endclocking
wire local_push;
// build RTL logic for local_push
A1: assert property (full  |->!push);
A2: assert property (empty |->!pop);
A3: assert property (local_push == push);
C1: assume property (vld & !rdy |-> ##1 vld);
C2: assume property (vld & !rdy |-> ##1 $stable(data));
endmodule
bind DUT ABV_chk abv_inst (.*);
```

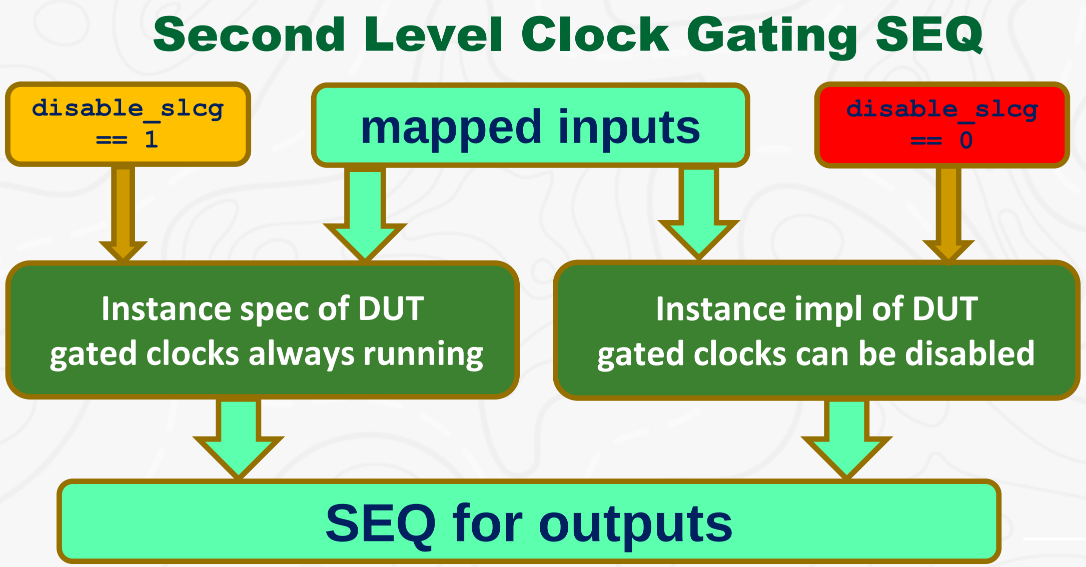
Assertions

Formal Clock Gating Verification -SEQ

SEQ: SEQuential equivalence checking



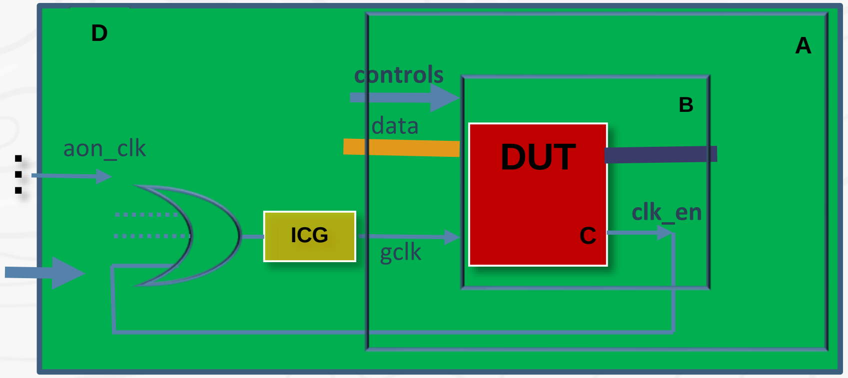




- spec.out_foo == impl.out_foo
  or spec.out_valid && impl.out_valid |-> spec.out_foo == impl.out_foo
- For all valid input combinations, there is no output mismatch up to certain cycles
- If possible, full proofs for all cycles

C1: Sub-unit Level Clock Gating Verification

- Divide-conquer is the key for full convergence (e.g., module based)
- Sub-modules with clock enable logic can be easily proven
- At top level SEQ, sub-modules can be abstracted (e.g., black boxed)
- This can be done before top level RTL is completed



cg_constraints.sv

```
bind C ICG fv_icg(.aon_clk(),
                  .oclk(),
                  .clk_en(clk_en));
```

control.tcl:

```
elaborate_seq -same_design -spectop C  -impltop C ...
map_by_name -exclude gclk
create_clock {spec.fv_icg.aon_clk spec.gclk} -period 100
fvassume impl.gclk == impl.fv_icg.oclk -env
```

C2: Temporary RTL Fixes without Touching RTL

- Multiple bugs filed with different fixes
- Need to verify RTL fixes before submitting
- One formal run to disable and enable each fix without re-building RTL files or re-compiling bench

```
module foo (
….
clk_en = f1 || f2 ;
…
```

RTL fix

```
module fv_foo (
….
clk_en = f1 || f2 || f3 ;
…
endmodule
bind foo fv_foo foo_fixed (.*);
```

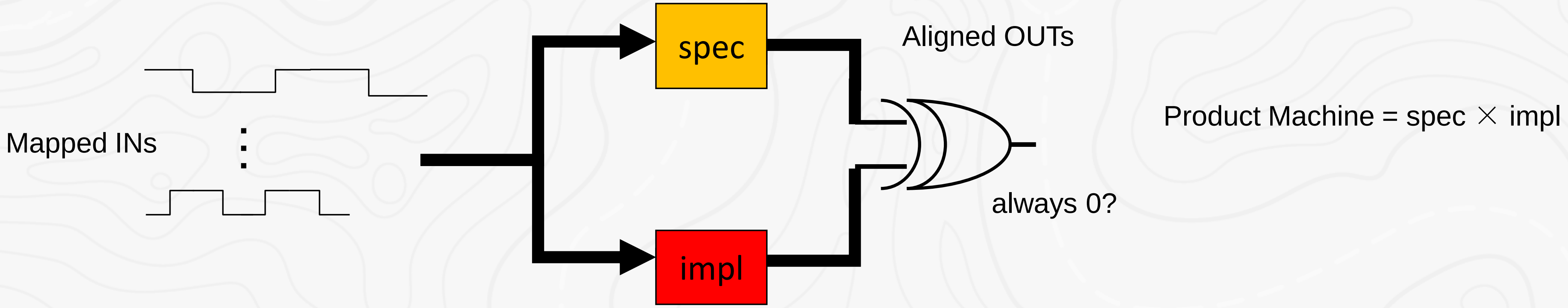
control.tcl:

```
snip_driver -env -seqmap -exact spec.foo.clk_en
fvassert driver_foo_clk_en -expr {$drive(spec.foo.clk_en) == $drive(impl.foo.clk_en)}
fvassume BUG_xxx_fix -expr {impl.foo.clk_en == impl.foo_fixed.clk_en} -env


…
## disable RTL fix for BUG_xxx
snip_driver -remove {spec.foo.clk_en impl.foo.clk_en}
fvdisable -usage assert {driver_foo_clk_en}
fvdisable -usage assume {BUG_xxx_fix}
```

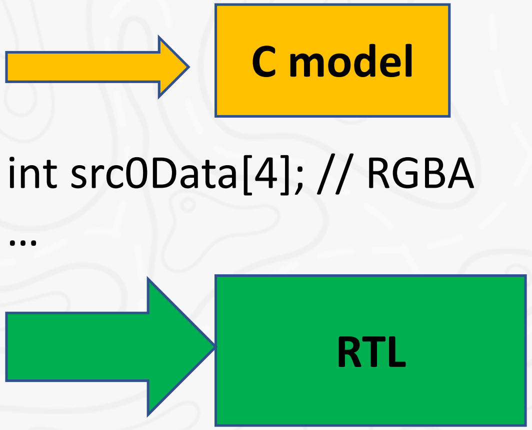
Data-Path Verification (DPV)

- Increasing use of data path (arithmetic) logic requires formal proofs
  - GPU/CPUs
  - Machine learning accelerators
- Specification vs. implementation
  - Specification: high-level programming language (e.g., C++)
  - Implementation: RTL
- Challenges
  - Mappings IOs between two models (data type, encoding, ….)
  - Convergence



C3: Constrain Inputs in DPV

- Bit to bit mapping between C model and RTL is not easy
- TCL control file cannot easily access text macros inside RTL
- Helper RTL logic with bind statements can simplify IO mappings
- Create more conditions for better case-splitting to achieve full proofs



Color Blender
4(2x2) pixels

int src0Data[4]; // RGBA
…

input [3:0] [37:0] src0_color;
input [3:0] src0_color_fmt; // ARGB/RGBA/BGRA …..
…;

```
module map_bits #(`include "cb_params.vh") ( …);
…
always_comb begin
 for (pix_num=0;pix_num<NUM_PIXELS;pix_num=pix_num+1) begin: pixel
….
 // build the logic to match C model for each color format
     newData[…] = inData[…]…;
 // build valid logic for each pixel for a given color format
     valid_pixel[pix_num] = ….;
 end : pixel
 assign inDataValid = &valid_pixel;
…
endmoule
bind color_blender map_bits src0 (.inData(src0_color),
                                  .color_fmt(src0_color_fmt), …);
```

control.tcl

```
assume spec.src0Data (1) == impl.src0.newData(1)
assume impl.color_blend.src0.inDataValid (1)
##Case-splitting on internal signals of map_bits module.
….
```

C4: Build RTL Wrapper with Bind - A Decompressor

- Bind statement can be used to build a RTL wrapper to connect pieces of RTL from different places
- Need to verify CVD (Coarse Visibility Decompressor). C model of it cannot be consumed directly
- CVC (Coarse Visibility Compressor) has been verified through C2RTL
- CVC and CVD are from different clusters of the GPU
- RTL designers of CVC/CVD tried to build a wrapper circuit for formal – very time-consuming task

Prove: InputPrimitive == OutputPrimitive

```
bind COMP DECOMP decomp_inst (.sclk(sclk), .rst(rst));
```

```
set_cutpoint COMP.decomp_inst.next_data_buffer_p0
…//compile and elaborate design
cutpoint DECOMP_indata_cut = COMP.decomp_inst.next_data_buffer_p0 (1)
assume DECOMP_dataInput = {DECOMP_indata_cut[275:127] == COMP.code_final[149:1](1)}
```

TCL control file

Formal Property Verification (FPV)

- Besides the normal usage of bind statements for assertions and helper logic, we can have another usage model for bind statements
- Two cases are shown
  - Concurrent SVAs for combinational modules
  - Parameterized RTL

C5: Concurrent SVAs for combinational modules

- Avoid immediate assertions for formal verification
- What about a combinational module where there is no clock/reset?
- Conditional bind under `ifdef for different scopes

```
module fv_checker (input clock, reset, ..);
// concurrent assertions
endmodule
`ifdef TOP_DUT
   bind DUT fv_checker fv_inst (.clock(), .reset(), .*);
`else
   bind DUT fv_checker fv_inst (.clock(FOO.clock),
.reset(FOO.reset), .*);
`endif
```

```
analyze ... +define+TOP_DUT
create_clock {DUT.fv_inst.clock}
create_reset {DUT.fv_inst.reset}
```

TCL control file

FOO is a module scope above DUT with clock/reset defined

C6: Parameterized RTL Designs – ECC

- 10+ configurations for Error Correction Code (ECC) in a project
- Each configure can be verified with only one formal bench

```
module ecc_chk #(parameter DWIDTH=256,
                           ECC_WIDTH=10)(
...//helper logic and SVAs
endmodule
bind ecc ecc_chk #(.DWIDTH(DWIDTH),
                   .ECC_WIDTH(ECC_WIDTH)
ecc_chk_inst;
```

```
if {[info exists ::env(DWIDTH)] && [info exists ::env(ECC_WIDTH)]}{
   set p_str "-pvalue+ecc.DWIDTH=$::env(DWIDTH)
              -pvalue+ecc.ECC_WIDTH=$::env(ECC_WIDTH)"
} else {
   set p_str "-pvalue+ecc.DWIDTH=256 -pvalue+ecc.ECC_WIDTH=10"
}
elaborate  -top ecc  -vcs " … $p_str"
```

control.tcl

```
#!/bin/bash
…
export DWIDTH =128; export ECC_WIDTH=9
vcf -f ecc_check.tcl -out_dir ecc_${DWIDTH}_${ECC_WIDTH}
export DWIDTH =256; export ECC_WIDTH=10
vcf -f ecc_check.tcl -out_dir ecc_${DWIDTH}_${ECC_WIDTH}
…
```

regress.sh

Conclusion

- Formal verification is Assertion-Based Verification (ABV)
- Helper logic and assertions are heavily used
- Bind statement is very powerful together with TCL controls in formal
- 6 use cases are given in this paper for popular formal applications
- Any innovative use of bind statements can
  - Simplify benches and cut the cost of future maintenance effort
  - Verify clock-gating before RTL is complete
  - Help formal convergence

Questions?