DOI: 10.31114/2078-7707-2020-3-22-27

Use of Formal Methods to Resolve Actual Problems of ASIC Design Verification

A.A. Sokhatski

Cisco Systems Inc., asokhats@cisco.com

***Abstract* — There are some critical problems of SoC Design Verification (DV) related to growing functional complexity including**

- **Time to Market: full enough verification takes too much time;**
- **Verification Quality: costly chips re-spins take place; bugs are escaping detection by traditional verification method.**

The paper describes:

1) **How Formal Verification could help to resolve Time to Market problem by doing "left shift" of design – verification timeline involving designers to use Formal tool to initially clean up design**
2) **How Formal Verification helps to improve verification quality and avoid re-spins by detection of simulation – resistant bugs [1]**
3) **Example of Lookup Table Block, corner case bugs close to simulation – resistant bug concept detected by Formal Verification which will be difficult to catch by simulation; Formal Verification strategy for the block and used Formal techniques**

***Keywords* — SoC, Design Verification, Formal Verification, SystemVerilog, SVA.**

I. INTRODUCTION

While simulation keeps being the main method of functional Design Verification of chips[2], it is not able to adequately address challenges related to growing complexity of the chips. Two major challenges are time to market and quality of verification – ability to avoid or reduce number of chip re-spins. The challenges require to use other methods, first of all, emulation and Formal verification.

Traditionally application of Formal methods was started from checking equivalence between RTL and gate-level netlist. Then Formal methods started to be used to prove particular properties represented as assertions (Formal Property Verification [3,4,5]).

Now bunch of Formal tool applications are available. They simplify Formal usage for particular functional aspects, for example, Connectivity Checks, Formal Coverage Analysis for detection of uncoverable code.

In addition to verification of particular aspects, Formal Verification could be used for overall sign-off of design blocks. Formal tools provide support for Formal coverage analysis to ensure Formal verification quality [].

Paper will look how Formal Verification could help to address verification challenges and why it is difficult to do with just simulation. Example will be used based on the author experience.

II. FORMAL VERIFICATION TO REDUCE TIME TO MARKET

A. Comparison with simulation

Rough typical sequence of design and verification steps for target block includes:

- Block design by design engineer;
- Development of verification environment by verification engineer:
- Development and running tests and regressions; at that time first bug and majority of bugs are found
- Running more random regressions, collecting coverage and developing more tests

In that flow block needs to be designed, passed to verification engineer, simulation environment developed before first bug is found.

B. How Formal could help

Formal verification helps to reduce total development and verification time by

- doing "left shift": find first bug earlier and complete earlier;
- reducing time in the last phase: no need to run random regression more and more; Formal coverage analysis is still required though.

The following could be done to do "left shift":

- Use Formal method by designers at early stage for initial cleaning of the block from bugs
- Use of Formal Applications which do not require setting environments, e.g. Automatically generated checks / Super lint, X-Propagation checks.

Designer could start from generation and analysis of waveforms with use of Formal tool. It could be done by

defining cover properties and proving them. Formal tool will generate waveform how to reach coverage point.

Formal tools have now special features to assist initial bring up, for example, Formal Navigator which is part of VC Formal. In simple case user could just select signal, value and Formal tool will use it as cover property and will generate waveform for that. It also assists in creation of more complex cover properties.

After exercising design with cover properties designer could put assertions and try to prove them. It makes sense to start from simple assertions for particular cases.

Designer could also apply Formal Apps mentioned above. We had the case when designer found arithmetic overflow case which potentially could be an issue.

C. *What needs to be done in advance in company / project scope*

Note that unreal scenarios could be generated for cover properties and false failures could be detected for assertions. In that case constraints needs to be added, in particular constraints for input interfaces. For industry standard protocols it makes sense to use assertion IP and for standard protocols at company / project scope it makes sense to develop such assertion IP[6]. For input interfaces assertions will be used as constraints (assumptions).

Another group of assertions which could come "for free" for designer are assertions from design reuse modules, for example, FIFOs, credit control modules. At the company / project scope it makes sense to fill up such reuse modules with SVAs for error conditions. Even simple assertions for FIFO / credit overflow could help to detect issues with usage of that components at upper scope.

III. Formal Verification to avoid Simulation Resistant bugs

Despite help of Formal tools to reduce time to market is important Formal benefit, the main advantage and goal of Formal method from the beginning is exhaustive proof of design properties.

Especially it is important for designs which could potentially contains so called "simulation - resistant bugs", which could be very difficult to find via simulation.

A. *Why simulation-resistant bugs could escape*

Simulation – resistant bugs could escape because of

- High parallelism of the design, several processing threads crossing inside design;
- Big number of configuration inputs / registers required verification for various combinations; each combination might require separate simulation run
- Big number of input combinations, for example, for packet alignments, sizes, etc.

Two last factors could be mitigated and addressed by running random regressions for a long time. The first one is most critical. The reasons why bugs could escape from simulation for blocks with high parallelism include:

- It is difficult to cover all critical timing between events;
- Sequence of critical events could contain several consecutive events;
- Critical events could be deep inside design, not obvious and not controllable by simulation;

B. *Blocks with potential simulation – resistant bugs*

Signatures of such blocks include

- Several input interfaces affecting same internal state / output;
- Dynamic structures supporting add, delete, update operations
- Bypass logic, cache, arbitration logic

OSKI Technology is collecting list of such blocks in different areas [1] which includes for example:

- For CPU & GPU: Instruction Fetch Queue, Load Store unit, L2 Cache, Coherency manager, Resource Manager;
- For Networking, Ethernet, Wireless: Forwarding Engines, Linked-list controllers, Quality of Service units, Buffer managers, Block aligner, Packet Encoder/Decoder, Bypass cache and forwarding logic.

There are some examples of simulation-resistant bugs in the next section.

Formal method should catch simulation resistant bugs by proving assertions for ANY sequences satisfied constraints. For that certainly set of checks should be sufficient which could be confirmed by Formal coverage analysis.

IV. Example

A. *Design*

Here is example of Lookup Block with conceptual diagram below.

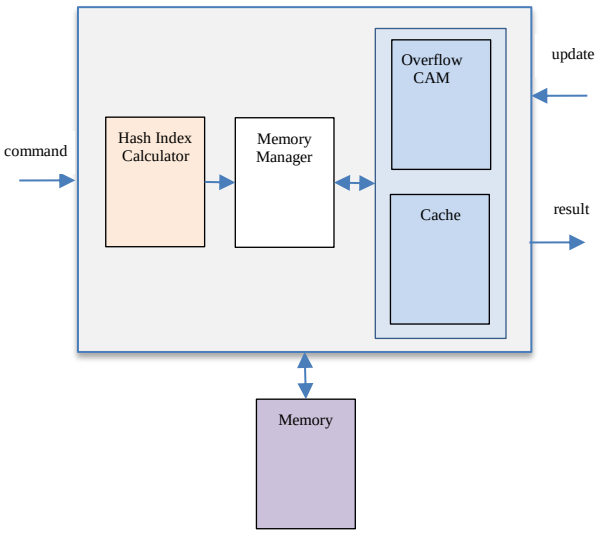

Figure 1. Diagram of Lookup Table Block

Block takes command which could be

- Lookup data for certain key;
- Scan lookup table: read and optionally remove entry

Block sends downstream lookup result and takes update for the lookup table.

Inside block has:

- Hash Index Calculator to select memory index for lookup key;
- Memory Manager controlling access to lookup table in external memory and communication with other blocks;
- Cache based bypass logic to accommodate memory latency as long as lookup commands could go back-to-back and update from previous one should contain lookup result for the next;
- Overflow CAM which contains lookup entries which cannot be placed in the memory, it contains internal bypass logic.

The Lookup Table block was selected for Formal Verification because it has several activities working in parallel: lookup, scan, update, it contains cache / bypass logic and dynamic data.

The most critical sub-module where actually simulation – resistant bugs were found is Overflow CAM block. Here we have dynamic location for key inside CAM: key could be added, removed, updated which is creating extra complexity and potential for simulation – resistant bugs.

B. Simulation resistant bugs

Here are few examples of corner case bugs which are illustrating concept and close to simulation – resistant bugs. Paper describes sequence of events required for bug detection. Bugs description and correspondent sequences are simplified to pass the concept. Actual sequences are more difficult to envision during test planning.

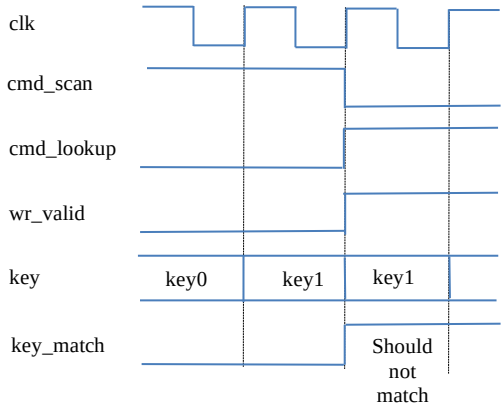

Figure 2. Waveform for 1st bug detection

There is the following sequence of 2 events when 1st bug detected:

- Scan command with certain *key1* presented at *key* signal; note that *key* is don't care signal for scan command (scan command is accompanied with memory / CAM index) but as long as simulation done at upper level it is not primary input and is not driven to 'X' during simulation
- Lookup command with the same key1 as for scan command

Bypass logic has key_match signal to check condition when lookup done for the same key as in previous cycle

```
wire key_match = wr_valid && fn_key_compare
   (key, ket_d, …);
```

However due to bug it missed command check. Should be

```
wire key_match = wr_valid && cmd_lookup_d &&
   fn_key_compare (key, ket_d, …);
```

This bug could be detected if we drive 'X' at *key* port when doing simulation at the scope of the CAM overflow module. Depending on the coding of fn_key_compare it might require activation of X-Prop simulation mode.

However as long as simulation done at upper level without control of the *key* signal it is very difficult to hit case when "occasionally" same *key* set for two commands especially as long as at upper level scan and lookup are in independent threads and timing between scan and lookup commands is not controllable. Note that simulation coverage will not detect missing the issue in that case. Formal method easily detected the bug.

Another example is related to dynamic modification: installation new key and invalidation of entry.

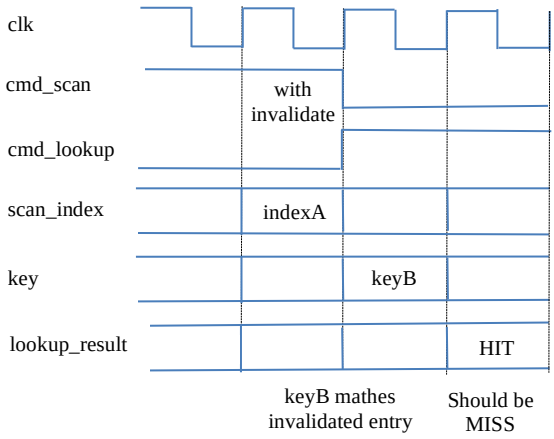

Figure 3. Waveform for 2nd bug detection

Here back-to-back coming commands:

- Scan command which invalidates entry with *indexA*
- Lookup command for *keyB* which matches contents of entry under *indexA*

Due to bug lookup gives HIT result but should give MISS one which requires installation of new entry.

It's difficult to hit such sequence via random simulation testing but it could be hit with dedicated test. However at the upper scope above Lookup Table Block it is much more

difficult to hit such case.

Another example is similar when we have no space available in the CAM but some entry just invalidated in previous cycle and available for installation of new key.

There is also case when bug detection requires sequence of 3 events in the row which is very difficult to hit or even to predict when putting cover properties for internal points.

Sequence here contains

- 2 scan commands: one causes update of the entry and the other does not

- lookup command which hits the entry

Due to the bug bypass logic take info from the wrong scan command.

C. Formal Verification Strategy for Lookup Table Block

Due to complexity it is difficult to achieve Formal prove at the block scope for all properties. Instead considered the following environments:

- Cache bypass logic; it is simple and rather standard environment; it was encoded very fast and found bug (not simulation resistant though);
- Overflow CAM; all corner-case simulation resistant bugs found here;
- Top-level environment which checks memory management using memory abstraction model and checks integration of other parts

Verification strategies for the environments are presented below illustrated with code examples for verification techniques good to know:

- Cache verification using symbolic memory index;
- Floating pulse method to simplify checks and improve performance by tracking only one transaction
- Memory abstraction

D. Formal Verification Cache – Bypass Environment

Basic strategy to verify cache – bypass logic:

- keep track of ANY but only one memory location;
- keep sign of presence data for that location in the cache and last written data
- compare read data with last written data if data is in the cache.

Here is some illustration code:

```
// symbolic variable which takes ANY value but
stable
// during formal prove
wire [INDEX_SZ-1:0] sym_mem_idx;
sym_mem_idx_stable: assert property (
  ##1 sym_mem_idx == $past(sym_mem_idx)
);

// check if selected index detected & keep status
of
// presence in the cache
wire wr_mem_idx_matches_sym = wr_mem_row_vld &&
  (wr_mem_idx == sym_mem_idx);
reg [CACHE_SZ-1:0] wr_mem_idx_sym_shift;
always @(posedge clk)
  wr_mem_idx_sym_shift <=
    {wr_mem_idx_sym_shift[CACHE_SZ-2:0],
     wr_mem_idx_matches_sym};
wire
wr_mem_idx_sym_in_cache=|wr_mem_idx_sym_shift;

// keep last data written for symbolic index
reg [DATA_SZ-1:0] wr_mem_data_last_sym;
always @(posedge clk)
  if (wr_mem_idx_matches_sym)
    wr_mem_data_last_sym <= wr_mem_data;

// check data from cache
rd_data_from_cache: assert property (
  @(poseged clk) disable iff (!rstn)
  rd_mem_idx_matches_sym &&
wr_mem_idx_sym_in_cache
    |-> rd_data == wr_mem_data_last_sym
```

E. Formal Verification TCAM Based Lookup Environment

Used verification strategy:

- check either lookup or bypass command at a time; use symbolic variable to select command type;
- for lookup command keep track of any but only one symbolic lookup key;
- for scan command keep track of any but only one symbolic CAM location;
- keep sign of presence data for symbolic key / location in the CAM and last written data;
- keep valid bit for all locations in the CAM;
- compare read data with last written data if data is in the CAM;
- use reset abstraction for CAM contents:

Reset CAM memory abstraction allow to start from any CAM state (contents) instead of starting from “empty” CAM state. That is why Formal tool does not need to go through sequence of transactions to reach interesting states including CAM almost full / full states. It will dramatically reduce proof time.

Reset abstraction could be done manually by cutting connection of reset to CAM entries but assigning entries to unconnected wires, see reset abstraction example here [3]. On the other hand, now days Formal tools could help with memory reset abstraction by using dedicated commands.

F. Formal Verification Top-level lookup environment with Memory Abstraction Model

Used verification strategy:

- It is verified that correct data from memory passed to the output for lookup or scan command and memory properly updated from update input on the right (see Figure 1.)
- Any arbitrary but only one transaction is selected using floating pulse method (see below) and tracked;
- Symbolic key for lookup command / symbolic index for

scan command assumed to be on the input interface when command selected; they are used to sync with abstraction models (see below);

- Hash index calculator contains some arithmetic calculation logic which is not friendly for Formal Property verification; it is not verified in this environment but replaced with abstraction model;
- Hash index calculator abstraction model returns symbolic index for symbolic key and random data for any other keys;
- Memory is represented by abstraction model which supports writing, storing and reading data only for one entry at symbolic address but returns random data for any other accesses; this way access with wrong address will be easily detected;
- Overflow CAM could be replaces with similar abstraction model as well but for the sake of time and because it is located at the boundary of the block, passing right data is checked at the Overflow CAM ports and Overflow CAM excluded (black-boxed) from DUT;
- Cache module is left inside DUT RTL; it was found that it does not significantly affect performance, no need in abstraction

Here is code illustrating Floating pulse method to select any but only one transaction

```
wire fl_pulse;
reg fl_pulse_done;
always @ (posedge clk) begin
   if(!rstn) begin
      fl_pulse_done <= 1'b0;
   end else if(fl_pulse) begin
      fl_pulse_done <= 1'b1;
   end
end
no_fl_pulse_when_done_model: assume property(
   fl_pulse_done |-> (!fl_pulse)
);
```

Here is simplified code of Memory Abstraction model. Here we assume unknown output if read happen when write is in progress or vice versa.

```
module fv_mem_one_entry_abs#(
   parameter WIDTH = 1,
   parameter ADDR_SZ = 1,
   parameter RD_DELAY = 1,
   parameter WR_DELAY = 1
)(
   // DUT inputs
   input clk,
   input rst,
   input rd_en,
   input [ADDR_SZ-1:0] rd_addr,
   input wr_en,
   input [ADDR_SZ-1:0] wr_addr,
   input [WIDTH-1:0] wr_data,
   // DUT output
   input [WIDTH-1:0] rd_data
);

default clocking @(posedge clk); endclocking
default disable iff rst;

// symbolic address - only one address is tracked
wire [ADDR_SZ-1:0] sym_addr;
// symbolic initial data contents
wire [WIDTH-1:0] sym_init_mem_data;
// keep only one data for symbolic address
reg [WIDTH-1:0] mem_data;
// read pipe for matched valid
reg [RD_DELAY-1:0] rd_pipe_valid_sym;

wire rd_en_sym; // when address matches symbolic
wire rd_data_sym;
wire [WIDTH-1:0] rand_rd_data; // when no match
wire [WIDTH-1:0] fv_rd_data; // calculated output
reg [$clog2(WR_DELAY):0] wr_in_progress_cnt;
wire wr_en_sym;
wire wr_in_progress;

assign wr_en_sym = wr_en && (wr_addr ==
sym_addr);
always @(posedge clk) begin
   if (rst) begin
      mem_data <= sym_init_mem_data;
   end else if (wr_en_sym) begin
      mem_data <= wr_data;
      wr_in_progress_cnt <= WR_DELAY;
   end else if (wr_in_progress_cnt != 0) begin
      wr_in_progress_cnt--;
   end
end
assign wr_in_progress = wr_en_sym ||
   (wr_in_progress_cnt > 0);

assign rd_en_sym = rd_en && (rd_addr ==
sym_addr);
always @(posedge clk) begin
   if (rst) begin
      rd_pipe_valid_sym <= 0;
   end else if (wr_en_sym) begin
      rd_pipe_valid_sym <= 0};
   end else if (rd_en_sym && !(wr_in_progress))
   begin
      rd_pipe_valid_sym <=
        {rd_pipe_valid_sym[RD_DELAY-2:0], 1'b1};
   end else begin
      rd_pipe_valid_sym <=
        {rd_pipe_valid_sym[RD_DELAY-2:0], 1'b0};
   end
end

assign rd_data_is_sym =
   rd_pipe_valid_sym[RD_DELAY-1];
assign fv_rd_data = rd_data_is_sym? mem_data :
   rand_rd_data;

// Assign output through constraint
output_drive__rd_data: assume property (
   rd_data == fv_rd_data
);

endmodule
```

Memory abstraction model is instantiated inside testbench connecting to correspondent ports. Symbolic variables need to be synced with symbolic variables defined inside Formal testbench, for example:

```
// memory instantiation
fv_mem_one_entry_abs #(…)
) lkup_mem
(
   .clk(clk),
   .rst(!rstn),
    …
);
// symbolic variable for memory address
wire [ADDR_SZ-1:0] sym_lkup_mem_addr;
sym_lkup_mem_addr_stable: assume property (
  ##1 sym_lkup_mem_addr ==
$past(sym_lkup_mem_addr)
);
// should be synced with hash index calculator
and
// memory abstraction model, e.g.
lkup_mem_sync_model: assume property (
   (lkup_mem.sym_addr == sym_lkup_mem_addr)
);
```

V. Conclusions

Formal method could help to solve two critical functional Design Verification issues:

- Reduce Time to Market when designers start to apply Formal techniques earlier in the project;
- Improve verification quality, try to avoid chip re-spin by detecting simulation - resistant bugs,

It is illustrated in example which shows some bugs close to simulation – resistant concept, verification strategy and Formal techniques.

Acknowledgements

Author thanks OSKI Technology team who delivered training on Formal Verification few years ago and continue to guide Formal at conference and seminar events, introduced simulation – resistant bug concept and left shift idea with Formal help. Author thanks his managers and colleagues from Cisco Systems for support.

References

[1] Oski Technology. High-Risk Blocks Formal Sign-Off Available at http://www.oskitechnology.com (accessed 01.07.2020)

[2] Sokhatski .A. Practical Aspects of Design Verification of Complex chips // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2016. Proceedings / edited by A. Stempkovsky, Moscow, IPPM RAS, 2016. Part2. P. 16-21.

[3] Sokhatski .A. Practical Aspects of Formal Verification of Networking chips // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2018. Proceedings / edited by A. Stempkovsky, Moscow, IPPM RAS, 2018. Part2. P. 16-22.

[4] Seligman E., Schubert T., Kumar A.K. Formal Verification: An Essential Toolkit for Modern VLSI Design. Waltham, MA, USA: Elsevier, 2015, 352P.

[5] Murphy B., Pandey M., Safarpour S., Finding Your Way Through Formal Verification. Danville, CA, USA: SemiWiki LLC, 2018, 133P

[6] Tatarnikov Y., Labib K. Next step of Formal Verification utilization Available at https://www.synopsys.com/community/snug/snug-silicon-valley/location-proceedings-2018.html (accessed 03.05.2018)

УДК 519.714

Использование формальных методов для решения актуальных проблем верификации проектов СБИС

А.А. Сохацкий

Сиско Системс Инк., asokhats@cisco.com

***Аннотация* — Моделирование остается основным методом функциональной проверки проектов СБИС и систем на кристалле. Однако этот метод не справляется с проблемами, связанными с растущей функциональной сложностью систем и их блоков. К числу основных проблем относятся:**

- **Жесткие требования к срокам разработки, проверки и выпуска, нарушение этих сроков**
- **Неполнота функциональной проверки проектов; необнаруженные ошибки проектов приводят к необходимости перепроектирования и повторного изготовления**

Применение метода и инструментов формальной верификации может помочь в решении этих проблем. В статье рассматриваются следующие вопросы:

1) Как применение метода формальной верификации может помочь в решении проблемы сокращения сроков разработки и проверки проектов путем использования инструментов формальной верификации разработчиками блоков для начальной проверки с использованием специальных отладочных режимов формальных инструментов и с применением формальных приложений автоматически создающих утверждения (assertions);

2) Как применение метода формальной верификации позволяет обнаружить ошибки проекта, которые сложно обнаружить путем моделирования, так называемые "simulation resistant bugs";

3) Рассматривается пример блока поиска и для него примеры ошибок проекта, которые сложно обнаружить путем моделирования; также рассматривается стратегия формальной проверки и используемые подходы, включая использование символических переменных, символического выбора элемента последовательности, абстрактной модели памяти; приводится код на языке SystemVerilog.

***Ключевые слова* — СБИС, система на кристалле, формальная верификация, моделирование, RTL, SystemVerilog, SVA.**

Литература

[1] Oski Technology. High-Risk Blocks Formal Sign-Off Available at http://www.oskitechnology.com (accessed 01.07.2020)

[2] Сохацкий А.А. Практические аспекты верификации проектов СБИС // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2016. №2. С. 16-23.

[3] Сохацкий А.А. Практические аспекты формальной верификации проектов сетевых СБИС // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2018. №2. С. 16-22.

[4] Seligman E., Schubert T., Kumar A.K. Formal Verification: An Essential Toolkit for Modern VLSI Design. Waltham, MA, USA: Elsevier, 2015, 352P.

[5] Murphy B., Pandey M., Safarpour S., Finding Your Way Through Formal Verification. Danville, CA, USA: SemiWiki LLC, 2018, 133P

[6] Tatarnikov Y., Labib K. Next step of Formal Verification utilization Available at https://www.synopsys.com/community/snug/snug-silicon-valley/location-proceedings-2018.html (accessed 03.05.2018).