Formal Verification of an ASIC Ethernet Switch Block

B. A. Krishna
Chelsio Communications Inc.
bkrishna@chelsio.com

Anamaya Sullerey
Chelsio Communications Inc.
anamaya@chelsio.com

Alok Jain
Cadence Design Systems, Inc.
alokj@cadence.com

***Abstract*— Traditionally, validation at the ASIC block level relies primarily upon simulation based verification. Specific components that are "hot spots" are then considered as candidates for Formal Verification. Under this usage model, the hurdles to Formal Verification are intractability and poor specifications. In this paper, we outline an alternate approach, where we used Formal Verification as the "first line of defense" in the course of validating a Packet Switch. This block had several components that were complex and hard to verify, including components that required liveness guarantees, where responses are event bound, and not cycle bound. To surmount typical hurdles, an early collaboration was formed between design and verification engineer, both to influence the design as well as to identify relevant manual abstraction techniques upfront. All significant components were formally verified at the module level.**

This approach was successful in identifying most bugs during the design phase itself and drastically minimized bugs during verification/emulation phases of the project. This paper illustrates the strengths of such an approach. It describes our overall methodology and the proof techniques utilized. The overall effort yielded a total of 55 bugs found (52 during the design phase and only 3 bugs during the verification phase). No bugs were found subsequently during emulation. As a result, this block was deemed "tape out ready" 2 months prior to other blocks of similar complexity.

I. Introduction

The complexity of modern designs has been increasing at a rapid pace. Modern design blocks are made up of modules that have very complex behaviors and interactions. Verification of such blocks poses a serious challenge. The conventional approach is to verify through simulations at the block level. However, simulation has the inherent limitation that one can simulate only a limited set of patterns in any reasonable amount of time. As design sizes grow, it is becoming increasingly difficult to maintain a high level of confidence purely based on simulation coverage. A possible solution is to use Formal Verification to verify some of complex modules in your design. Formal Verification performs exhaustive verification by exploring the entire state space of the design.

In this paper, the design block under consideration is a switch with around 650k gates and with multiple ports. Most of the modules inside this design block have high complexity both in terms of the control oriented behavior and data path operations. Based on previous experiences, it was estimated that simulation based verification techniques of such designs would require more than a year for a dedicated engineer to fully verify.

In addition, the design in question also had several components for which *liveness* guarantees were required, which were not possible to validate using simulation based verification. Thus, it was therefore concluded that the most cost-effective approach would be to utilize Formal Verification techniques to prove correctness of all significant components of the design at the module level. Conventional simulation based design verification (DV) was also done, but at the block level.

Our overall approach was inspired by the following quotation from *"Mythical Man Month"* [1]:

"The use of clean, debugged components saves much more time in system testing than that spent on scaffolding and thorough component test."

Our FV efforts commenced very early during the design phase and consisted of the following methodology (which took place alongside conventional DV efforts at the block level):

1) Partition the design into minimally sized pieces and generate specifications at the module level. Use the compositional verification technique of proving properties of a system by checking the properties of its components, using "assume-guarantee" style reasoning.

2) Aim to prove "black-box" (end-to-end module level) properties and use the tractability results to both influence design re-partitioning as well as to gain insights about RTL complexity.

3) Study cones of influence in order to deduce possibilities for manual abstractions. Once identified, these abstractions were then used to replace stateful RTL components within the design.

In a few cases where all other options failed, we resorted to proving "white-box" properties (based on RTL internal state). We used this approach as a last resort since rigorous specifications of RTL internals are hard to come by, and further, such specifications often change in the course of the design cycle.

This paper will focus on the techniques used to verify two of the modules in the design, namely the *Synchronizer* and the *Page Manager* modules. The first case study is a control &

datapath block that consists of 20k gates and the second is a datapath block that consists of 25k gates.

In subsequent sections, we describe each module, the Device Under Test (DUT) operational details, the Formal Verification strategy utilized in each case as well as the verification results. Later, we also present our overall results (number of bugs found etc.) and our high level conclusions. The model checkers Incisive Formal Verifier (IFV)[2] and SMV[5] were used over the course of this project.

II. DESCRIPTION OF THE PACKET SWITCH

The design block under consideration was a packet switch with multiple ports that accepted packets, stored them in memory, and later forwarded them to various output ports, allowing for the possibilities of switching and replicating packets.

In order to accomplish this functionality, the block had various types of complex components, components that were responsible for storing incoming packets to memories, components that were responsible for managing pages in memory over which packets were stored, components that maintained caches, etc.

The goal here was to a) design specifically with Formal Verification in mind (keep modules small, keep interfaces crisp) as well as to b) formally verify as many elements of the design as possible. In total, 14 modules of the design were formally verified. The design consisted of 18 modules in its entirety.

The following design principles were utilized to ensure FV tractability:

- Careful design partitioning with exhaustive invariants of module interfaces.
- Isolation of modules that exhibit FIFO-ness.
- Significant parameterization of modules, to allow abstraction/reduction of bus widths, etc.
- Significant reuse of common modules, e.g., arbiters, aligners, etc.
- Decomposition of all architectural invariants into micro-architectural invariants.

III. FORMAL VERIFICATION OF THE SYNCHRONIZER

The synchronizer module has two inputs, a) packet data is sent across *in_{valid,sop,eop,data[63:0]}* , where *sop* and *eop* are start/end packet delimitors and b) address of a valid page is specified across *in_addr, in_addr_valid*. Its purpose is to place the arriving data, which arrives in units of 8 bytes, into various slots within the specified page. The interface for this module is shown in Fig 1.

The input packet data bus adheres to the following protocol: *in_valid* is asserted whenever there is new data presented across the input. During the first 8 byte data chunk within a packet, *in_sop* will be asserted, and during the last 8 byte chunk, *in_eop* will be asserted.

Each page is of size 128 bytes, which is broken down into 16 x 8 byte slots. This module receives an input, *sync_cnt[3:0]*, which is an external counter that increments every cycle. The output consists of: *rf_write, rf_write_sop, rf_write_eop, rf_write_data[63:0]*. If, at any point in time, we see *rf_write==1* and *sync_cnt==i (where i:=0...15)*, then it means that *rf_write_data[63:0]* is being written into slot *i* within the page.

The rules determining when/what data is written into a particular slot in a page are described in the *Operational Details* section. All data arriving over *in_data* goes into an internal *skid_fifo*. The data that is at the head of the *skid_fifo* is written out into a page only when various design rules are satisfied.

This module is called the *synchronizer* because it synchronizes when and where an incoming data segment is written into a page. It is part of a larger system that is responsible for accumulating various 8 byte chunks of data within a register file so that it can later generate an atomic memory write operation for a half page worth of data.

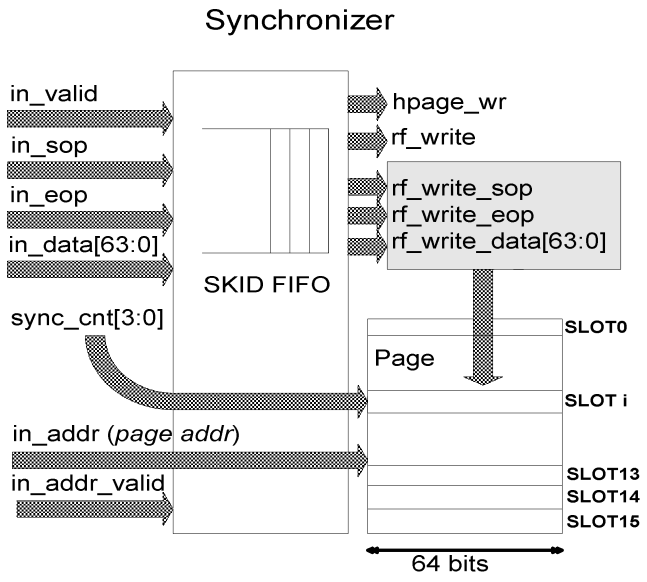

Figure 1 – Block Diagram of the Synchronizer

IV. OPERATIONAL DETAILS OF THE SYNCHRONIZER

Following are rules governing the *Synchronizer* module:

- Across the datapath between *in_{valid,sop,eop,data}* & *rf_write,rf_write_{sop,eop,data}*, FIFO-ness needs to hold. Note that the input bus has no backpressure capability (i.e., the input interface should *always* be able to sink data and cannot throttle the input bus).

- For a given page (presented on: *in_addr), rf_writes* should occur to *slot=0....15* in a monotonically increasing fashion.

- For a given page, if a non-EOP data word was written into *slot=i*, then the next data word for this packet *must* be written into *slot=i+1*.

- If we are at the lower half of a page (*slot=7*) and a) there's an *rf_write* or b) we are not within a packet and have seen an *rf_write* in the past to the lower half of this page, then at the next cycle *hpage_wr* will be asserted and not otherwise.

- If we are at the upper half of a page (*slot=15*) and a) there's an *rf_write* or b) we are not within a packet and have seen an *rf_write* in the past to the upper half of this page, then at the next cycle *hpage_wr* will be asserted and not otherwise.

V. DESIGNER'S INVARIANTS FOR THE *SYNCHRONIZER*

Apart from the rules that were identified by the verification engineer, we also proceeded to prove the following invariants put forth by the designer. The intent here was to prove invariants that emerged after interface study by the verification engineer, as well as those that were deemed important by the designer.

- If there is an *rf_write* to some slot *x (where x=0…15)*, then there will be no write to slot *y (y<=x)* until there is an assertion of output signal *hpage*_wr.

VI. SYNCHRONIZER VERIFICATION STRATEGY

We could visually establish that this module demonstrated *data independence*. The circuit accepted data and shuffled it around, but no control signals were derived from data. This could be done relatively easily, by examining the fan-out cone associated with the data-path elements.

Further, the design also dealt exclusively in terms of 8 byte (64 bit chunks) and didn't reorder data bytes within each incoming double word. In order to prove that the unit fulfilled the specification of a FIFO, it was possible to utilize *Wolper's Theorem* [3], abstract the data width to just 2 bits, inject a regular expression consisting of *A*BA*CA** over the input data interface and expect that the data showing up at the output also conformed to this regular expression.

A packet generator was written to inject packets that a) conformed to SPI4 framing conventions and b) had a minimum length of 64 bytes, over the input bus: *in_{valid,sop,eop,data}*. This packet generator data words consisting of just 4 types: *{A,B,C,D}, where A=64'h0, B=64'h1, C=64'h2, D=64'h3.* A auxiliary fsm was written to monitor the outputs: *rf_write,rf_write_{sop,eop,data}*.

Three critical proofs, pertaining to packet data-integrity and framing, were then cast using the packet generator and auxiliary FSM.

Proof Obligation1: To prove data integrity across the FIFO's data-path.

This proof asserted that if we injected packets conforming to the regular expression *A*BA*CA** over *in_data[1:0]*, then we are guaranteed to see outputs that also conform to the regular expression *A*BA*CA** over *rf_write[1:0].* Note that this regular expression is injected and expected across all valid input and output data words This proves that no input data word is dropped, duplicated or reordered.

Proof Obligation2: To prove that SOPs are preserved intact across the internal FIFO.

For this proof, the regular expression *A*BA*CA** was injected into *in_data[1:0]* for SOP input words, and *D* was injected into *in_data[1:0]* for non-SOP input words. The expectation was that the regular expression *A*BA*CA** will always be seen on *rf_write[1:0]*, for SOP output words and *D* will always be seen on *rf_write[1:0]*, for non SOP output words.

Any corruption of an input SOP word (with data values: *{A,B,C}*) into an output non-SOP word, would result in an output non-SOP with a value of *{A,B,C}*, which will be detected as a violation of *Proof Obligation2*.

Any corruption of an input non-SOP word (with data value: *D*) into an output SOP word, would result in an output SOP word with a value of *D*, which will be detected as a violation of *Proof Obligation2*.

Proof Obligation3: To prove that EOPs are preserved intact across the internal FIFO.

For this proof, the regular expression *A*BA*CA** was injected into *in_data[1:0]* for EOP input words, and *D* was injected into *in_data[1:0]* for non-EOP input words. The expectation was that the regular expression *A*BA*CA** will always be seen on *rf_write[1:0]*, for EOP output words and *D* will always be seen on *rf_write[1:0]*, for non EOP output words.

Any corruption of an input EOP word (with data values: *{A,B,C}*) into an output non-EOP word, would result in an output non-EOP with a value of *{A,B,C}*, which will be detected as a violation of *Proof Obligation3*.

Any corruption of an input non-EOP word (with data value: *D*) into an output EOP word, would result in an output EOP word with a value of *D*, which will be detected as a violation of *Proof Obligation3*.

In order to prove that writes within a page were to monotonically increasing slots, a tracking FSM was written. This FSM did the following: Every time a new page was presented over *in_addr, in_addr_valid*, it recorded the slot into which it first saw an *rf_write*, storing both the value of *sync_cnt* into *last_wr_ptr* as well as *rf_write_{sop,eop}* into *last_wr_{sop,eop}*.

Properties were then written to monitor the behavior of *rf_write*. The two most important assertions were:

1. If we are performing an *rf_write* to some slot=*sync_cnt* and if this is *not* the first write to the page, then *sync_cnt* will be greater than *last_wr_ptr*.

2. If we are performing an *rf_write* and if this is *not* the first write to the page and if the previous write was a non-EOP data word (i.e., *last_wr_ptr=i && last_wr_eop=0*), then this write *will* be to slot=*(i+1)*.

This tracking FSM also monitored writes to upper/lower halves of a page such that later, when *sync_cnt={7,15}* (i.e., write pointer is at the upper/lower half boundaries), if any writes had occurred to a half, the output *hpage_wr* would be asserted.

VII. SYNCHRONIZER VERIFICATION RESULTS

A critical bug was found in the implementation of *hpage_wr*. The failing counterexample consisted of a scenario where there was a write to the upper half of a page for which there was a valid *hpage_wr* assertion. However, this signal continued to be asserted for 8 extra cycles indicating a write to the lower half of the page inspite of the fact that the lower half was not written into. This was found very early in the design stage.

Another critical bug was found in the FIFO size required. The property corresponding to *Proof Obligation1* failed. Our analysis showed us that the minimum FIFO depth should have been 18 and not 16. The depth had to account for the internal FIFO latency. This bug was found very early in the design stage. While *sync_cnt* is a primary input to this module, it is an internal signal within the larger block. Since conventional simulation based DV was being performed at the block level, precise control over this signal is difficult to realize in simulation, making this bug an improbable event within block level DV. The designer estimates that debugging this issue would have required ~ 2 hours within a block level verification test failure, but within the module level FV framework, this debugging took just a few minutes.

VIII. FORMAL VERIFICATION OF THE *PAGE MANAGER* MODULE

The *Page Manager* module's block diagram is shown in Figure 2. It is responsible for managing all pages on the receive path of our Ethernet Switch. This module's interface supports four types of requests: *Allocate, Enqueue, Dequeue* and *Dealloc*. It also has an output bus, *Page Free*.

IX. PAGE MANAGER OPERATIONAL DETAILS

Data passing through the switch from input to output ports is stored in pages. A list of pages defines a packet. The *Page Manager* maintains the state of the page, from the time it is allocated until the time it is relinquished. Internally, the *Page Manager* consists of a) *Free List Manager* and b) *Life Count Memory*. These two sub-units together maintain the state of a page, which consists of its allocation state as well its reference count (i.e., the number of packets utilizing that page).

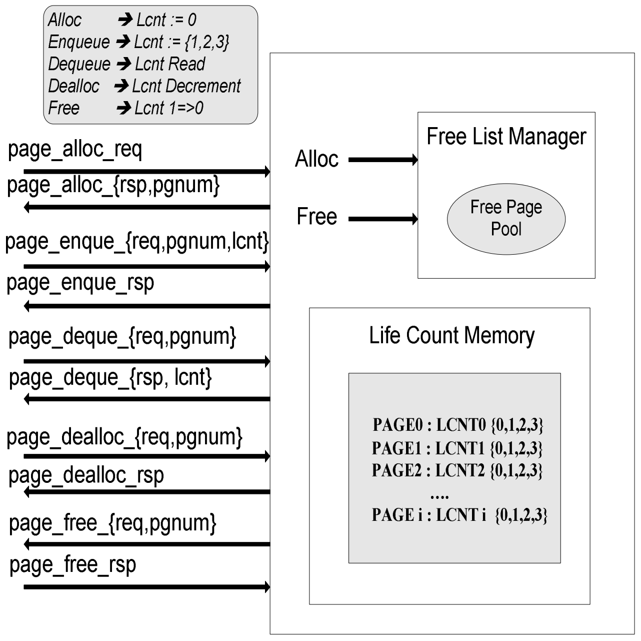

Figure 2 – Block Diagram of Page Manager

The *Free List Manager* sub-unit maintains a list of free pages and its interface allows pages to be allocated and freed. The *Life Count Memory* sub-unit maintains a reference count (also called *Life count* or *lcnt*) on a per-page basis, representing the number of packets present on a single page. The legal *lcnt* values are: 0...3.

The life cycle of any page consists of the following event sequence:

- The unit first receives a *Page Allocate*. This request is fielded by the *Free List Manager*, and a free page is handed to out to the requestor. Coincident with that, the page's *lcnt* is initialized to 0 in the *Life Count Memory*..

- Once a page has been successfully allocated, an *Enqueue* reque*st* will be received along with a specified initial *lcnt*. The legal values for *lcnt* are: {0,1,2,3}. This information is then stored alongside the page within the LCNT complex.

- After a page has been *Enqueue'ed*, it will then receive (at arbitrary points in time), various *Page Dealloc* requests.

During each *Dealloc* request, this page's *lcnt*, will be decremented in the *Life Count Memory* complex.

- The design assumes that once a page has been *Enqueue'ed* with some *lcnt* (1,2 or 3), it will only field those many *Dealloc* requests. After the last *Dealloc* request (in the course of which a particular page's *lcnt* goes from 1 to 0), the *Free List Manager* should free the relevant page and the*Page Free* output signal will be asserted.

- Between the time a particular page has been *Enqueue'ed*, and the time it is freed up, its *lcnt* can be read any number of times over the *Page Dequeue* interface. Each *Dequeue* request extracts the *lcnt* and return this value in the *Dequeue* response.

X. PAGE MANAGER VERIFICATION STRATEGY

The design was responsible for managing a total of 1024 pages. When an attempt was made to cast proofs against the DUT, it was found that the proofs did not converge due to *state space explosion*. The biggest contributor to the state space was the *Free List Manager* (with 1024 state bits).

The *Free List Manager*'s interface definition is shown in Table I. This module has a page allocation interface *alloc_{srdy,drdy,num}* as well as a page free interface *dlloc_{srdy,drdy,num}*.

Table I (Free List Manager Interface)

```
/*
 *     alloc_srdy => alloc page available
 *     alloc_drdy => alloc page consumed by client
 *     alloc_num => alloc page number
 *     dlloc_srdy => dlloc page requested by client
 *     dlloc_drdy => dlloc page request accepted
 *     dlloc_num => dlloc page number
 */
module fl_mgr(
  Clk,
  Rst_,
  alloc_srdy,
  alloc_drdy,
  alloc_num,
  dlloc_srdy,
  dlloc_drdy,
  dlloc_num
);
input              Clk;
input              Rst_;
output             alloc_srdy;
input              alloc_drdy;
output [9:0]       alloc_num;
input              dlloc_srdy;
output             dlloc_drdy;
input [9:0]        dlloc_num;
...
endmodule
```

Our abstraction reasoning hinged on a single observation: *If you focus on the life of a single page, every other page's activity (and state) should be orthogonal to this page's life.* We utilized this observation in constructing a manual abstraction for the *Free List Manager* that maintains state only for a single page of interest thereby cutting down the size of the cone-of-influence significantly. This technique is based on the *Refinement strategy* described in [4].

The *Free List Manager* abstraction had the following characteristics:

- It was aware of the address of a *magic page* and maintained state only for that page.

- It operates in two modes, depending upon whether this *magic page* is allocated or not:
  - If the *magic page* was already allocated, during subsequent allocation requests, it would non-deterministically allocate a page whose address!= *magic page*.

  - If the *magic* page wasn't already allocated, during subsequent allocation requests, it would non-deterministically allocate any page (including one whose address == *magic page*).

This *Free List Manager* abstraction SMV code is shown in Table II. This abstraction was coded in both SMV (for abstraction soundness proofs) as well as in verilog (for the *Page Manager* proofs, which were run within IFV).

As can be seen in the abstraction's code, a single state variable, *magicPageAllocated*, was used to record whether or not the *magic page* was allocated, and this state is then used in determining the page handed out during allocation requests.

Aside from this state, the notion of *magic page* was maintained within a *rigid variable* that was set non-deterministically by the external environment at the time of reset, and kept constant during each path. By virtue of maintaining just 1 bit of state (*magic page*'s allocation state), the number of bits of state was reduced by 1023 bits within the cone of influence. This abstraction was then used to replace the *Free List Manager* instance within the DUT.

The intent here, in the construction of the *Free List Manager* abstraction, was to provide ourselves with a light-weight stub that allowed completely non-deterministic allocation and freeing of pages, with arbitrary latencies, with a single restriction that it would never reallocate the *magic page*, if someone else already have it allocated – which are characteristics required for this abstraction to be "sound".

Table II (Free List Manager Abstraction)

```
layer abstract : {
  alcVld                      : boolean;
  dlcVld                      : boolean;
  magicPageAllocated          : boolean;
  magicPageAllocatedNxt       : boolean;

  alcVld := (alloc_srdy & alloc_drdy);
  dlcVld := (dlloc_srdy & dlloc_drdy);

  init  (magicPageAllocated) := 0;
  next (magicPageAllocated) := magicPageAllocatedNxt;

  /* magicPageAllocatedNxt generation */
  default {
    magicPageAllocatedNxt := magicPageAllocated;
  } in {
    if (~Rst_)
      magicPageAllocatedNxt := 0;
    else {
      if (alcVld & ~dlcVld){
        /* Only Alloc */
        if ((alloc_num=magicPage) | magicPageAllocated)
          magicPageAllocatedNxt := 1;
      }
      else
      if (~alcVld & dlcVld){
        /* Only Dlloc */
        if (magicPageAllocated & dlloc_num=magicPage)
          magicPageAllocatedNxt := 0;
      }
      else
      if (alcVld & dlcVld){
        /* Both Alloc & Dlloc */
        if (alloc_num=magicPage)
          magicPageAllocatedNxt := 1;
        else
        if (dlloc_num=magicPage)
          magicPageAllocatedNxt := 0;
      }
    }
  }

  /* alloc_num generation */
  default {
    /* any page whatsoever */
    alloc_num := {0..MAX_NPAGES-1};
  } in {
    if (alloc_drdy & magicPageAllocated){
      /* any page other than magicPage */
      alloc_num := { i : i=0..MAX_NPAGES-1, i~=magic Page };
    }
  }
```

The FV framework additionally maintained an auxiliary non-deterministic "tracking state" FSM (*trkState*) to both exhaustively generate requests sequences while tracking the life of the *magic page* as well as to help predict the DUT's responses. This FSM's state diagram is shown in Figure 3

The *trkState* FSM starts off in IDLE state and transitions into ALCD state if *magic page* is allocated. Once it is in ALCD state, it non-deterministically generates an *Enqueue* request with *lcnt={1,2,3}* and transitions to states LCNT1, LCNT2, LCNT3 respectively. After it moves into an LCNT state, it then non-deterministically generates as many *Dealloc* requests as is permissable.

During the last *Dealloc* request generation (which occurs while in LCNT1) state, this FSM expects to see a *Page Free* event for the *magic page*. If this event occurs, the FSM transitions to IDLE. On the other hand, during this last *Dealloc*, a *Page Free* event is not observed for the *magic page*, it transitions to and forever remains in ERROR state. In addition, any unexpected output event also caused a transition to ERROR state.

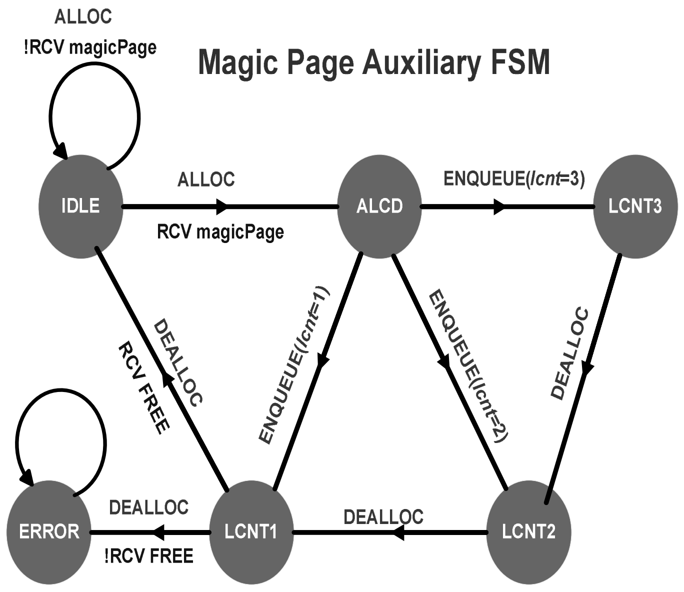

Figure 3 – trkState FSM state diagram

There are two modes of operation within the FV framework, based on whether or not *magicPageAllocated* is set:

1. If *magicPageAllocated* is 0, the *trkState* FSM will be in IDLE and the FV framework will non-deterministically generate requests (for any page), to the DUT.

2. If *magicPageAllocated* is 1, the *trkState* FSM will generate legal/exhaustive requests (for *magic page*) while other input constraints non-deterministically generate requests (for any page other than *magic page*).

In addition to generating exhaustive and legal inputs, the purpose of the FSM's state variable was to predict the DUT's responses while in various states.

We now describe some important assertions governing the DUT's behavior (These were coded in System Verilog):

- While in non-IDLE states (i.e., *magic page* has already allocated), the DUT should not reallocate *magic page* to any other requesting agent.

- After the Allocate phase, during the *Enqueue* phase for the *magic page,* the specified *lcnt* should be initialized.

- After the *Enqueue* phase for the *magic page,* during each *Dealloc* phase, its *lcnt* should be properly decremented in the LCNT memory.

- The output *Page Free* should be generated for the *magic page* if and only if the last *Dealloc* request has been issued for this page.

- While in non-IDLE states, for any *Dequeue* request, the response *lcnt* should match what we expect based on the FSM state (0 if in ALCD, 1 if in LCNT1, 2 if in LCNT2, 3 if in LCNT3).

Table III (Example SV Assertions)

```
/*
 *  If we're in non-IDLE state, magic page is already in use and
 * should not be reallocated to any other requestor
 */
assert_page_no_realloc: assert property(
   @(posedge Clk) disable iff  (!Rst_)(
     (trkState!=IDLE) |-> !(page_alloc_req && page_alloc_rsp
&& page_alloc_pgnum==magic_page)
   )
);

/*
 *  If  in LCNT1 state and there is a dealloc of the magic page,
 * then we should see a freeing of the magic page
 */
assert_page_free_valid: assert property(
   @(posedge Clk) disable iff  (!Rst_)(
     (trkState==LCNT1 && page_dealloc_req &&
page_dealloc_rsp && page_dealloc_pgnum==magic_page) |->
     (page_free_req && page_free_pgnum==magic_page
   )
);

/*
 * If  in !(LCNT1 state and there is a dealloc of the magic page),
 * then we should not see a freeing of the magic page
 */
assert_page_free_invalid: assert property(
   @(posedge Clk) disable iff  (!Rst_)(
     ! (trkState==LCNT1 && page_dealloc_req &&
page_dealloc_rsp && page_dealloc_pgnum==magic_page) |->
     ! (page_free_req && page_free_pgnum==magic_page
   )
);
```

We provide some example SV assertions in Table III. The first property, *assert_page_no_realloc*, asserts that if *trkState* is not IDLE, that is if the *magic page* is already allocated, it will not be reallocated to any other requestor.

The second and third properties that are shown here, *assert_page_free_{valid,invalid}*, describe the necessary and sufficient condition required for the *magic page* to be freed ("*magic page should be freed if and only if trkState is in LCNT1 and magic page is deallocated*").

By maintaining a rigid variable that determined *magic page* and by having a *Free List Manager* abstraction that maintained state for just this one page, the design was rendered tractable. The properties outlined earlier were all proven against the life of this single *magic page*, and since this page address was non-deterministically generated (to have any page address), the proofs hold for all pages.

In the interest of completeness, the *Free List Manager* was separately formally verified within an SMV framework. Two properties were proven against the actual *Free List Manager*:

- A page, once allocated, will never be reallocated until it is deallocated *(safety property)*
- All page allocation requests will eventually be fulfilled *(liveness property)*

It is worth noting that the last property mentioned above required the following *fairness* constraint: *"Every allocated page will always eventually be relinquished"* in order to eliminate invalid counter-examples.

In addition, the soundness of (an SMV version of) the *Free List Manager* abstraction was also proven within this framework.

XI. OVERALL VERIFICATION RESULTS

During this project, 14 modules within this block were formally verified by a single FV engineer, over a period of 6 months. A total of 55 bugs were found during this effort; 52 bugs were found in the design phase and 3 bugs were found in the verification phase. It is also worth noting that during the verification phase, 3 other bugs slipped through FV and were found in block level simulation (2 were due to missing properties and 1 was due to an overly tight constraint).

The 3 bugs found in simulation were recreated within FV by adding new properties and correcting an overly constrained input. In addition, the fixes were formally verified.

During emulation, this formally verified block was the first to successfully withstand data integrity type testing. As a consequence, this block was deemed *tape-out ready* two months prior to other blocks, of similar complexity that exclusively underwent simulation based verification.

During ASIC "bring-up", no issues were found in any of the design components that were formally verified.

XII. CONCLUSIONS

Based on our experience, we come to the conclusion that it is possible to significantly address block level verification needs by breaking down the design into minimally sized modules and then formally verifying each of them.

Our methodology also helped yield the following benefits over the course of this project:

- Overcoming state space explosion during proof runs within the model checker.
- Generating rigorous specifications upfront at the module level, something that is often overlooked while embarking on “block level” DV.
- Providing SVA assertions and assumptions which could also be used in simulation.
- Creating FV frameworks within which we could verify design changes/bug fixes with a high degree of confidence alleviating the need to rerun all simulation tests.

While re-partitioning of design based on FV tractability can sometimes lead to added design latency, this tradeoff was worthwhile overall because the more minimally sized design modules were easier to maintain.

We also observed that debugging of counter-examples was very efficient since we specified a large number of module level invariants that helped isolate root-causes fairly quickly.

We believe there is value in some amount of overlap between FV efforts and conventional simulation based verification. Such a parallel/overlapping approach reduces the risks posed by overly tight constraints and inadequate (or missing) properties. This overlapping effort is justified by the fact that almost all bugs were found in the design phase itself and the FV proof frameworks provided us with a vehicle within which the fixes could be formally verified.

While the techniques outlined here, to render modules tractable under FV, are well known in the research world, they are seldom applied in the course of ASIC formal verification efforts and are hence worth emphasizing.

XIII. LIMITATIONS AND FUTURE WORK

Our approach relies on the verification engineer using design insights to come up with the right manual abstractions. This approach does risk bias particularly in light of the fact that commercial model checkers (that we know of) lack the means to prove *soundness* of abstractions or the means to express *refinement maps* (as can be done with SMV[5]).

To alleviate this risk, we made a deliberate attempt to keep our abstractions very simple (less than half a screen worth of verilog code per abstraction), and as a result have a high degree of confidence in our abstractions’ soundness.

For the specific case of the *Free List Manager* abstraction, we reimplemented this abstraction within an SMV “layer” and proved its soundness, ensuring that for every path taken within the RTL component replaced, there exists at least one identical path within the abstract definition.

Most commercial model checkers do not possess the ability to verify data-independence in any automated way. We look forward to such features so that we can utilize them in the interest of completeness.

However, to put these concerns into practical perspective, we observe that these risks are no worse than other concerns, such as ensuring that DUT inputs are not over-constrained, ensuring that assertions correctly capture the specification’s intent, etc.

ACKNOWLEDGMENT

We are very grateful to Ásgeir Eiríksson, the CTO of Chelsio Communications, for providing us with valuable insights pertaining to abstractions and to Jon Michelson for sharing his numerous detailed observations. We are also grateful to Vigyan Singhal, Prashant Arora and Paul Everhardt for providing feedback regarding various drafts of this paper.

REFERENCES

[1] Frederick P. Brooks, Jr. *The Mythical Man-Month*, Addison-Wesley Publications, 1995.

[2] *Incisive Formal Verifier User Guide*, Cadence Design Systems.

[3] Pierre Wolper. *Expressing interesting properties of program in propositional temporal logic*. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 184-193.* ACM Press, 1986.

[4] Ásgeir Eiríksson. *The Formal Design of 1M-gate ASICs*. In *Formal Methods in System Design, Vol 16, Issue 1 (Jan 2000), Special issue on formal methods for computer-aided system design.* Kluwer Academic Publishers.

[5] K. L. McMillan, *Getting started with SMV*, Cadence Berkeley Labs, 1999.