A case study for formal verification of a Timing Co-Processor

Cristiano Rodrigues
Brazil Semiconductor Technology Center (BSTC) – Freescale Semiconductor Inc.
Cristiano.Rodrigues@freescale.com

Abstract

eTPU is a state-of-the-art timing co-processor unit that aims to relief I/O processing in new advanced microcontroller units. It has characteristics of both a peripheral and a processor, which are tightly integrated, requiring a verification strategy that covers equally well both of these roles. This paper discusses the formal verification effort of some specific eTPU features. For newer versions of eTPU, some complexity increasing showed to be suitable for a formal verification approach. Formal Verification was now applied to verify recently added complex features. This approach is then compared with a simulation-only approach adopted earlier.

Index Terms— eTPU, VC Verification, timing co-processor, functional verification, formal verification

1. Introduction

The increasing complexity of designs brings challenges to engineers to fully verify and assure design meets specification requirements. In some cases, alternative functional verification approaches showed to be a good choice to reduce development cycle time, and time-to-market. The complete eTPU verification effort was previously described in [1]. In that work, the possibility of using Formal Verification techniques was identified for some eTPU sub-blocks, including eTPU channels, whose formal verification effort is discussed here.

Design complexity brings a wide range of states to be exercised, which causes a time-consuming task of generating and checking such states, and an additional effort to assure acceptable coverage metrics so the design could be considered fully verified and matching the specification.

Formal verification approach allows covering all possible states in the design since it runs a formal proof instead of simulation. Once the formal proof assures a statement as true, the negation of such statement is considered impossible to be reached.

This paper is organized as follows: after this introduction, a section which briefly describes previous works; a section which contains the formal verification environment; a section containing the case study itself; a section which shows experimental results, followed by a conclusion section.

2. Previous work

Functional verification aims to check design behavior against specification, in such a way to assure the design fully meets the specification, which contains all requirements and premises needed to have the design.[2]

2.1. Simulation-based Verification on eTPU

For the first version of eTPU, the verification effort was fully based on a reference model and a set of simulation stimuli, some of them directed and self-checking, and some additional random stimuli [1]. Particularly for channel modes verification, a set of stimuli based on the channel behavior due to sequences of events (one stimulus for each mode) was developed, in a way that all possible sequence of the four events were simulated, and channel status (flags and captured time bases) were checked after each event.

In order to ease stimuli reuse and maintenance after potential specification changes, the stimuli for channel modes verification were developed in a way which the C code was updated through a script which was fed with a table containing the specified behavior for the channels in response to any sequence of events. This approach has proved to be very effective and it achieved a high level of coverage for channel mode verification on the first version of eTPU (eTPU architecture and channel modes are going to be described in section 4).

This approach was not feasible for User Defined Channel Modes, since a set of directed stimuli to cover all possibilities would not be feasible. Random stimuli could also be a possibility, but it would require a considerable amount of simulation time and debug time over corner cases.

Consequently, the case was identified as a suitable situation to try an assertion-based formal verification (ABV)[2] approach, a formal verification technique with assertions, which has been introduced and used for functional verification in the semiconductor industry over the last years, as in [3],[4], and [5].

2.2. Assertion-based Formal Verification

This approach allows the verification engineer to improve observability and controllability over the design, and let the corner cases to be sought and identified by the formal tool, since assertions are able to capture how design should operate. It means to have the specification translated into a formal language eliminating the need for exhaustive simulation.

The formal verification tool synthesizes the block into a so-called *formal model*, and through the process of understanding the internal structure of the design, formal analysis reports whether the assertions are violated or not. This analysis provides an unbiased feedback on the design integrity. Static formal verification [6] uses mathematical techniques to prove some assertions as true (given a set of assumptions), and other assertions as false (by discovering counterexamples). A proof in this terms means that static formal verification has exhaustively explored all possible behavior regarding the assertion and has determined that it cannot be violated. A counterexample shows the specific circumstances (if any) under which the assertion is not satisfied.

Considering the expertise and tool availability, the ABV approach chosen was to use the System Verilog Assertions (SVA) language and Cadence^{TM} IFV tool for a static formal verification environment, in which it could have the channel design sub-block instantiated with assertions connected into its interface, so the formal model could be generated and specification-based assertions could be proved.

3. Proposed Formal Verification Environment

The eTPU channel sub-block interfaces with several eTPU sub-blocks. However, for the purpose of the channel mode verification, the main focus was on the interface with two specific sub-blocks: the microengine and the time bases sub-blocks.

In this case, the module implementing the channels is checked standalone, not embedded into the eTPU module.

In formal verification terms, all module inputs for the DUV (device under verification) are handled as primary inputs by the formal analysis tool to freely drive these signals when trying to generate assertions proofs or counterexamples.

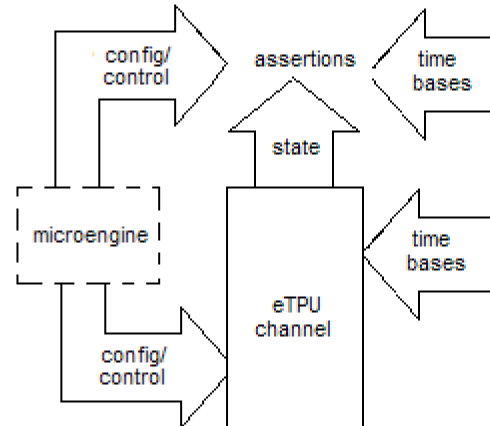

Fig 1 Primary inputs

Invalid combinations in channel inputs should be constrained so that the formal engine does not put channel into any invalid state (unreachable in a real situation). For instance, in terms of channel internal register programming, all write-enable signals should only be set active if a valid value (in terms of specification) is set to be written. In SVA terms:

```
const_ipac1_valid: assume property (@(posedge clk)
        (me_ipac1_we) |-> (me_ipac1_wdata < 6));
```

What means *me_ipac1_we* – input signal for the channel block – should be active only when *me_ipac1_wdata* – another input – has a valid value, less than 6, as stated in the specification. In the real world, the microengine would never generate a combination of signals which would violate rules as this one.

Also, some constraints were added to assure some specific configuration settings stay constant during formal proof. It does not mean different situations are not going to be covered, but it must be constant since design specification does not allow such configurations to be changed during channel operation. For example, considering a write operation in channel internal configuration registers TBS1 and TBS2 should never be performed:

#Those two lines assure TBS1/2 registers can't change
constraint -add -pin top.me_tbs1_we = 0
constraint -add -pin top.me_tbs2_we = 0

To assure all desirable states were reached, some cover statements were also added, so any available combination among channel flags should be reached. In SVA terms, being *tdlx* and *mrly* transition and match flags, respectively:

cov_tdl1: cover property (@(posedge clk) tdl1);
cov_t1_m2: cover property (@(posedge clk) (tdl1 & mrl2));

The first statement means that `tdl1` is set some time during formal analysis, and that `tdl1` and `mrl2` are set together some time during formal proof, in second case. Both signals are not directly set by the formal tool, since they are outputs from our DUV. That means formal tool should try changing its primary inputs to put our DUV in a state that obeys such conditions. A potential error is then reported if the condition cannot be obeyed.

The DUV should be put in an initial state from which formal analysis will start to explore states in order to prove assertions. IFV provides an environment to drive some initial stimulus to simulate the DUV, using TCL [7] commands. The initialization allows to provide initial values for internal signals without reset values, and so avoid X's being propagated in states to be checked by the formal analysis. Also, some input pins could be constrained to avoid undesired states, such an unexpected reset or an undesired change in an internal register in the DUV.

An example of tcl commands sequence to put DUV in a 'safe' state is the following:

 # Initialize reset
force top.ipg_hard_sync_reset_b 0
force top.me_er1_wdata 24'b000000000000000000000000
force top.me_er2_wdata 24'b000000000000000000000000

#Run 10 clks to propagate reset
run 10

force top.ipg_hard_sync_reset_b 1
…
force top.me_tbs1_we 1
force top.me_tbs2_we 1
force top.me_tbs1_wdata 3'h0
force top.me_tbs2_wdata 3'h7
run 5

The first block would generate a reset, and force some non-initialized signals to a known state. It runs 10 time units so the DUV could effectively reset. The second block will release reset signal and then provide a safe write to registers which won't change during formal analysis, and make it run for 5 time units more. This will be enough to put the DUV in a safe and known initial state.

4. Case Study: eTPU

The enhanced Time Processing Unit (eTPU) is an intelligent, semi-autonomous co-processor designed for I/O processing with timing control. Operating in parallel with the main microcontroller CPU, the eTPU processes instructions and real-time input events, performs output waveform generation, and accesses shared data without CPU intervention. Consequently, for each timed I/O event, the CPU setup and service times are minimized or eliminated. The I/O events are first processed by a configurable hardware logic named a channel. There is one channel for each I/O signal pair. A dedicated, Harvard architecture CPU (hereafter called microengine) processes requests that come from the channels. The microengine serves up to 32 channels, which also share a pair of time-base counters used for input event timing and output timed event generation (see block diagram in Fig 1). The module formed by a microengine, the time bases, associated channel and support logic set is called an engine.

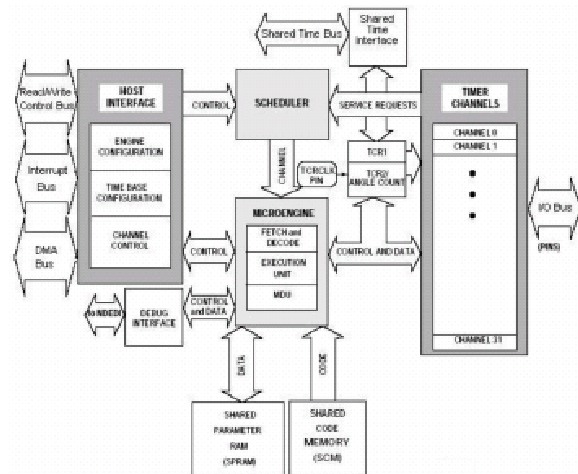

Fig. 2 eTPU engine block diagram

The eTPU works much like a typical real-time system: it runs microengine code from instruction memory to handle specific events while accessing data memory for parameters and application data. Events may originate from I/O Channels (due to pin transitions and/or time base matches), CPU requests or inter-channel requests. Events that call for local eTPU processing activate the microengine by issuing a

service request, which is a request for the microengine to execute some specific pre-loaded code. Some real-time system functionalities, like task scheduling and context switch, are implemented in hardware for performance sake.

4.1 eTPU channels

eTPU channels comprise hardware support for input digital signal processing and output digital signal generation. Each channel is associated with one input and one output signal. Channels are capable of dual action, meaning that each channel logic can handle two events at different times and/or cause two separated actions - these actions and events can be mutually dependent (with the first either blocking or enabling the other), or both independent, depending on the programmed channel mode[8].

Each channel contains event logic containing two event register sets, each set supporting one input and/or output action, the pair implementing dual action support. Each event register set contains two 24-bit registers: Match and Capture. The Match register holds the pending match value which is compared against one of the two time bases by an equal-only/greater-equal comparator. The Capture register captures one of the two time bases as a result of a Match or Transition detection. Requests to run microcode (service requests) are issued on particular combination of match and capture events, defined by the selected Channel Mode.

In the context of the eTPU channels an **event** could be either a **match** or a **transition**. A **match** is a comparison between a time base value and a channel match register value. A **transition** is a change in the input signal, which can be programmed to be detected or ignored. The dual-action characteristic of the channel defines then 4 different events: Match1, Match2, Transition1, and Transition2 (referred from now on as M1, M2, T1, and T2 for short).

Channel modes configure how channel handles with event recognition and related actions, such as service requests, time base capture, and output pin action, or event detection enabling/disabling. In first released version of eTPU, in Freescale MPC5554 and reused in some other Freescale microcontrollers, there were 13 pre-defined channel modes which could be described in terms of 8 internal signals, each one regarding a different channel behavior. They were:

- MSR (2 bits), which defines which match generates a service request;
- DTM (1 bit), defines time base capture and service request generated by transitions;
- MCAP (1 bit), defines time base captures issued by matches;
- M1ET (1 bit), defines if M1 enables transitions;
- M1EM2 (1 bit), defines if M1 enables M2;
- M1BM2 (1 bit), defines if M1 blocks M2;
- M2BM1 (1 bit), defines if M2 blocks M1;
- M2BT (1 bit), defines if M2 blocks transitions.

In all modes, transitions are always ordered, what means that T1 always enables T2. Obviously, not all combinations of signals above are reachable in only 13 channel modes. Verification of all these modes was then handled by a set of directed self-checking patterns, as will be described later. Also, besides the channel mode selection, there are others configuration items which could interfere on pin action and detection and also service request blocking, but these features were covered by a different set of directed patterns, not regarding channel mode handling.

On latest eTPU implementations, a more flexible configuration scheme, named User Defined Channel Modes (UDCM), which allows each of the signals listed above to be set independently, and also adding some signals (TSR and TCAP replaced DTM bit described above):

- TSR (1 bit), which defines which transition generates a service request;
- TCAP (1 bit), which defines time base captures by transition detection;
- T1BM1 (1 bit), which defines if T1 blocks M1;
- T2BM1 (1 bit), which defines if T2 blocks M1;
- TBM2 (1 bit), which defines which transition blocks M2;
- T1ET2 (1 bit), which defines transition ordering;

As they could be set independently, we have potentially 2^{13} different combinations, although some combinations make no sense (e.g., M1EM2 and M1BM2 set simultaneously), a lot of potential states and corner cases to be verified, therefore those new features were identified as candidates for formal verification.

5. Experimental Results

Assertions are, by definition, predicates (i.e. true-false statements) which translate the properties specified for the design in such a way to describe the behavior. SVA is a System Verilog superset Property Specification Language [9], [10].

In our formal verification environment, assertions were developed focusing on internal signals which

characterize the channel mode. During assertions coding and development, the debug effort was mainly focused on the creation and adjusting of constraints, since the assertions were almost copied directly from signals specification. For example, if signal `M1EM2` is set, it means that M2 can't be set if M1 was not set before. In SVA terms:

```
assign no_mrl = !(chns_mrl1 || chns_mrl2); // aux code

a_m1em2:   assert property (@(posedge clk)
disable iff ((full_progr_mode===1'b1) & (m1em2!==1'b1))
                                no_mrl |=> !chns_mrl2);
```

This assertion would be disabled if `m1em2` is not set in full programmable mode. If not in full programmable mode, the channel mode is in one of pre-defined modes, so this assertion is added only for modes which `m1em2` is set, no disabling needed. The assertion itself is very simple, self-explained: if no `mrl` is set, `mrl2` should never be set in the following clock cycle. It excludes situations where a sequence of events set `mrl1`, then `mrl2` and then MRL1 is cleared. It leaves design on a valid state, but it may not be understood by the assertions if instead of '`no_mrl`' we put '`no_mrl`1', for example.

The '`disable iff`' clause may be difficult to debug in some cases as was observed during the assertion debug phase. Assertions appeared to be false, but analysis of the counterexample showed that it was a situation with a correct behavior, but the assertion was not protected from a specific situation, such as a microcode intervention concurrently with event detection, for instance. In this case a condition had to be added.

About 44 property assertions were written to cover all possible situations to be checked regarding channel state, such as event flags, service request generation and time bases capturing. Additionally, 12 assertions were written to cover some specific situations which involve not only channel modes, but also some additional configuration registers which sets output pin actions due to events. There are two internal registers which handle output pin action, one for M1/T1 and other for M2/T2. When two simultaneous events take place, and each one of them is configured to a different pin action conflicting with each other, some kind of rule must take place to decide which action should prevail over the other. This rule is defined clearly in the specification, so additional assertions were written to check it.

5.1. Bugs found and verification effectiveness

The debug phase took most of the verification effort time. The assertion debug process had some iterations in order to refine the formal analysis, by tightening and loosing constraints – and eventually adding 'disable iff'clauses – as needed to achieve both coverage metrics goals and assertions proofing.

The formal verification effort payed off, finding one bug within a reasonable time window, before the eTPU IP was released. The bug was about output pin action prevailing depending on channel mode configuration.

The formal tool was able to generate a sequence of events/signals which took the design into a state which violates one of the assertions got from the specification. The tool provides a waveform which such sequence, but probing only signals which are relevant to generate the sequence. This saves a lot of debugging effort as well.

Overall, the task to have channel mode implementation fully verified took about 40% less effort than the estimated random simulation-based approach, considering that it would include the reference model update and debug to have the environment able to start running.

6. Conclusions and future work

Formal verification was found to be very effective for situations where increasing complexity causes simulation-based approach to be unfeasible. Assertions allow to easily translate specification into checking code, and additionally can provide coverage metrics to assure verification effectiveness. In the future, other eTPU sub-blocks like the microengine will also be considered for assertion-based formal verification.

Scenarios which may bring an unmanageably large, uncountable set of states to be checked are potential candidates for static formal analysis as applied in this work. ALUs, on the other hand, may not be suitable for formal verification. Some interesting investigation and valuable contribution would be a benchmark to properly identify circuits which may be more appropriate to be verified with formal techniques.

7. Acknowledgements

The author would like to thank the microcontroller verification team at Freescale's Brazil Semiconductor Technology Center, in special to Cesar Dueñas and Celso Brites, verification team manager and eTPU verification team leader, respectively, for the opportunities of trying innovative approaches; Walter

Encinas and Victor Miranda for their valuable contributions to this work; Chris Komar, from Cadence Design Systems Inc., for his always useful support during this case execution and evaluation.

8. References

[1] Brites C, Rodrigues C, "Functional Verification of a Timing Co-Processor: A Case Study", 6th IEEE Latin American Test Workshop, March 2005, Digest of Papers pp 129-133

[2] Encinas Jr. W.S, Duenas M. C.A, "Functional Verification in 8-bit Microcontrollers: A Case Study", Microelectronic Technology and Device, 2001, Symposium on, Brazilian Microelectronics Society, 2001.

[3] Sen, A.; Ogale, V.; Abadir, M.S.; "Predictive runtime verification of multi-processor SoCs in SystemC", Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE 8-13 June 2008 Page(s):948 – 953

[4] Kai-Hui Chang; Wei-Ting Tu; Yi-Jong Yeh; Sy-Yen Kuo; "A simulation-based temporal assertion checker for PSL", Circuits and Systems, 2003. MWSCAS '03. Proceedings of the 46th IEEE International Midwest Symposium on; Volume 3, 27-30 Dec. 2003 Page(s):1528 – 1531

[5] Roy, S.K.; "Top Level SOC Interconnectivity Verification Using Formal Techniques", Microprocessor Test and Verification, 2007. MTV '07. Eighth International Workshop on 5-6 Dec. 2007 Page(s):63 - 70

[6] Yeoung P., "4 Pillars of ABV", Euro DesignCon 2004

[7] TCL Developer Site, http://www.tcl.tk/

[8] Freescale, Inc., "ETPU Reference Manual" [online], available at <http://www.freescale.com>

[9] Property Specification Language (PSL) Reference Manual. [online], available: <http://www.eda.org>.

[10] System Verilog language reference manual, IEEE STD 1800-2005.