Functional Verification of a Timing Co-Processor: A Case Study

Celso Brites, Cristiano Rodrigues
Brazil Semiconductor Technology Center (BSTC) - Freescale Semiconductor Inc.
{Celso.Brites, 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 and testbench that covers equally well both of these views. This paper discusses several aspects of the functional verification effort based around signal-level cycle accurate behavioral model directed self-checking and random patterns, and how each component contributed to the overall results
Index Terms— eTPU, VC Verification, timing co-processor, functional verification, simulation, testbench

1. Introduction

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 timebases, associated channel and support logic set is called an engine.

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. Some real-time system functionalities, like task scheduling and context switch, are implemented in hardware for performance sake.

The eTPU instruction set has VLIW (Very Large Instruction Width) characteristics: an instruction may contain several active fields that command parallel operations. Many operation fields directly command the channel logic. Likewise, channel events may interfere with the program flow. The timed interaction between channel logic and the microengine is an important aspect to be considered in verification.

eTPU also provides Nexus class 3 [10] hardware debug support that posed an interesting verification challenge by itself but it is out of the scope of this work.

The eTPU selected as the DUV (device under verification) is a dual-engine system, which is composed of two engines sharing code and data memories.

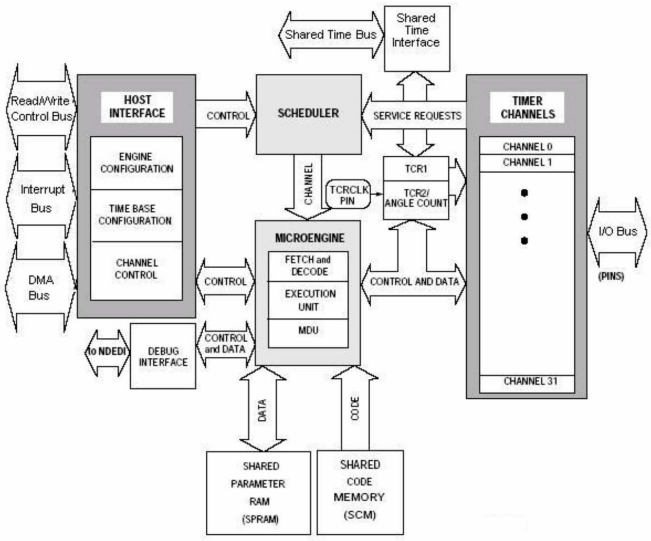

Fig. 1 eTPU Block Diagram

VC (virtual component) verification is the process of determining whether a VC fulfills a specification (spec for short) of its behavior [8]. The scope of this work is to describe the eTPU functional verification effort and its results. Due to the special nature of the eTPU that has characteristics both of a timer and processor, it required a hybrid verification strategy that partially incorporated methodologies like automated test generation [2] and transaction-based fully self-checking testbenches [3].

2. VC Verification

2.1. Verification Strategy

eTPU complexity is concentrated in two main areas which drove the verification strategy:

- The microinstruction set, with over 70 different operation fields distributed in 26 instruction formats. The level of orthogonality between those operations is relatively low, so that many instruction field value combinations configure exceptional behaviors.
- The channel logic, including the angle counters logic. Each channel can be chosen to work in either one of 13 different modes. Each of the Channel Modes configures the channel logic to respond differently to a series of input events, and to generate output signal actions.

Moreover, the two complexity factors above are combined as the channel logic behavior is affected by microinstruction intervention. Debugging support adds yet another factor to the overall system complexity. In order to cope with these challenges the eTPU verification effort followed a well known strategy [4] developed around a Response Checker and a functional model coded in C++ (Conformance Model or simply CModel). The eTPU CModel is cycle-accurate, being compared at each clock cycle with the DUT RTL (Register Transfer Level) or gate-level descriptions by a signal level Response Checker (RC). Both directed and random stimuli were created to be applied simultaneously to the CModel and DUT. Directed patterns were used to test all the basic functionality described in the eTPU spec. They target:

- Configuration register values
- Each individual microinstruction operation field, with all possible values, as well as exceptional (non-orthogonal) combinations between them.
- Each channel mode, with the event sequence responses described in the spec, not taking into account the microcode intervention into channel logic state between events.

Random patterns were used to verify the intended orthogonality between unrelated features, configuration register fields, microinstruction operation fields, channels, etc. Random patterns were also used to stimulate the channel logic with event sequences not directly mentioned in the spec, as well as microcode interventions into the channel logic state. Most patterns use only one channel to run. These patterns were coded with parameterized Engine and Channel numbers,
so they can be run on any channel. The random patterns were written to provide stimulus only. The checking part was implemented by the RC, comparing the DUT state with the CModel‘s at each clock cycle.

Complementing the strategy above, Assertion Based Verification (ABV) was used in selected functional modules (namely the Arbiter and the Scheduler) to implement internal monitors and functional coverage points. The assertions were written in OVA (Open Vera Assertions) [7] and checked every cycle against the RTL in all simulation runs.

2.2. Verification Methodology

The methodology chosen to implement the verification strategy is outlined below:

- Extract testable sentences from the specification (spec tagging);
- Develop self-checking, directed patterns based on testable sentences;
- Extract corner-cases from directed patterns reviews and brainstorms with design team;
- Develop a conformance model and response checker.
- Develop random patterns to test corner-cases and orthogonality;
- Develop assertions for critical functions;
- Collect and evaluate RTL code coverage and assertion functional coverage;
- Enhance or augment pattern suite until coverage goals are fulfilled.

2.3. Verification Strategy Implementation

CModel and RC description

The eTPU CModel is a cycle-accurate, full-featured functional model. Cycle-accuracy was needed because eTPU functionality is strictly related to the timing of channel events processing.

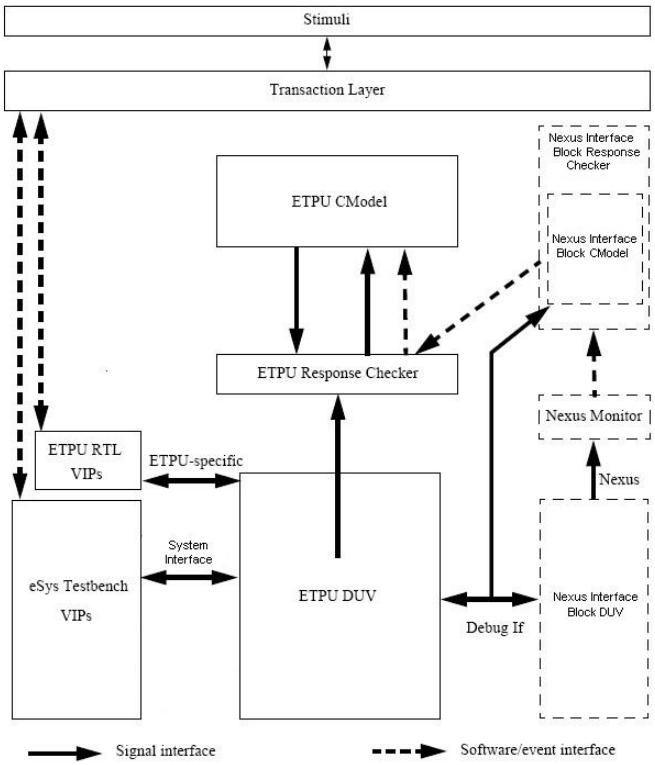

Fig.2 – eTPU Testbench Structure

The model has a hardware-like interface, consisting basically of the interface signals and buses of the eTPU itself. An API provides the call-based interface to the

stimulus. This API is the same used in all testbenches (RTL standalone, RTL + RC), allowing a pattern to be run without modification on any of them. The testbench structure is shown in fig. 2.

The eTPU RC acts at the same time as a driver to the eTPU CModel and as a monitor for the DUT (RTL or gate-level). It also advances the CModel's simulation clock. At each clock cycle, the RC collects the values of a set of signals from the DUT, in either a black-box (where only external output signals are checked) or a grey-box mode (where internal signals, registers, flags and FSM states are also checked). Mismatches are accused, ultimately causing the simulation to fail. The eTPU RC also takes care of handling the implementation differences between the CModel and the DUT, converting CModel signals to their DUT equivalents when necessary. For instance, the microengine state machines are different (albeit functionally equivalent) in the CModel and the RTL. The Response Checker creates an internal state register equivalent to the RTL one based on the CModel microengine state and other signals. The comparison is then made, each cycle, between this internal state register and the RTL one.

The CModel and RC were built upon a custom C++ framework. The framework uses a 4-level logic scheme where signals can assume the values 0 (logic zero), 1 (logic one), X (undetermined) and “don’t care”. The “don’t care” value was an important feature to make the RC signal comparison simple and ultimately viable: the overloaded ‘==’ C++ operator returns true when a signal with “don’t care” value is compared to any other value. The CModel assigns “don’t care” to any net or bus when their value is not significant from the functional point of view, usually when their qualifier control signal is not active. For instance, when a “read enable” signal is not valid, the read data bus is assigned a “don’t care”. Therefore, a direct comparison of this bus with its equivalent in the RTL can be made at all times regardless of the “read enable” control signal, avoiding any false mismatches. This greatly simplified the development and maintenance of the Response Checker.

Verification Patterns Development

Directed self-checking patterns were specified with the process known as "Spec-Tagging" [9], which consists in associating each functional sentence in the spec to a directed pattern that covers a specific functionality. A tool automatically generates a text file for each pattern and lists all functional statements associated with it (features tested) on a standard header. The pattern header also contains a section to document the strategy to cover the function as well as the expected DUT response. The strategy has enough detail to drive coding, to document the pattern objectives and to assist on pattern debug. All pattern headers were formally reviewed and corrections were made to the spec-tagging and strategies as necessary.

Patterns were then coded in C++ using a transaction-level API following the rules and guidelines in the Semiconductor Reuse Standards - SRS [1]. Patterns were manually debugged, meaning that stimulus at the transaction level were checked for their correct behavior.

Microcode for the eTPU itself was coded in simplified eTPU assembly language, which was developed to set directly all possibly fields in eTPU microinstructions, providing a very low level coding scheme suitable for automatic code generation. PHP [5] was used as a code generation language for both directed and random patterns, mainly in repetitive/random coding structures.

2.4. Verification Effort

The testbench evolved as the CModel, patterns and Response checker were developed. Table 1 relates pattern groups with the testbenches where they were run. The verification effort progressed through pattern/testbench combinations numbered I to V, as shown in Table 1.

Testbench	Cmodel	RTL	RC
Directed Self-Check Patterns	I	II	III
Random Patterns	used for pattern debug only		IV, V

Table 1 - Testbenches

The first activity was the development of the CModel, delivered as a conformance model to Ashware Inc, which used it to check its own eTPU simulator as part of an eTPU IDE suite. Bug reports were exchanged during the early stages of both model’s development.

When the first version of the CModel was released, the verification effort started with spec-tagging, after which a set of 107 directed patterns were listed to cover each testable line in the spec. The coding and debugging of these patterns used a CModel only testbench (I). CModel errors were also found and fixed. Many specification errors and omissions were uncovered during CModel development and debugging. When the directed pattern suite was fully debugged on the CModel, a functional version of the RTL became available, so the pattern debug started on an RTL standalone testbench (II). Most of the reported errors corresponded to real bugs in RTL, only a few being related to pattern coding or CModel. This first set of directed patterns was used as a basis for a pattern regression to be run regularly upon new RTL releases, uncovering new bugs introduced or uncovered by other bugs fixes. The Response Checker became available and started running with directed patterns (III), so all internal signals were being compared in RTL and CModel. This step increased the observability with exactly the same stimuli in the previous steps, uncovering mismatches between RTL and CModel that can be classified in 4 categories:

1. Spec ambiguities or just different interpretations from RTL or CModel designers;
2. Implementation differences on functional details not addressed by the specification, not considered bugs on either the RTL or the CModel. Usually the CModel was modified to adjust to the RTL behavior.
3. RTL bugs, not caught by the directed patterns.

4. CModel bugs not caught by the directed patterns.

The following step was random pattern development. These are stimulus only (not self-checking), so a testbench with the RC was necessary to uncover bugs. A brainstorming meeting was held with both design and verification teams in order to find corner case scenarios, situations not yet exercised which could be critical, such as channel action sequences not explicitly described in the spec, or microcode intervention simultaneous with channel events. This resulted in 17 high-level test descriptions of constrained random patterns to be developed. This new set of patterns (IV), uncovered a new wave of mismatches between RTL and CModel, included in the same categories above.

The final step was to simulate eTPU together with the Nexus debug support block (V). This module had already been verified in standalone mode with a very similar simulation environment, also using its own CModel and Response Checker, but with random stimulus only. The challenge was to completely verify the integration interface, since the two blocks were going to be delivered together, wrapped into a single bigger block. A new set of stimulus-only patterns was developed, mixing directed (not self-checking) and random patterns that uncovered a number of interface issues.

Assertions were introduced to check some specific features mainly related to scheduler and they were used to tune some constrains to cover situations not exercised.

3. Results

After all the verification effort reported here, the VC containing both the eTPU and the Nexus was delivered to be implemented in the MPC5554 microcontroller and the MCF523X family of microcontrollers. So far, a single bug not caught during verification was found on silicon.

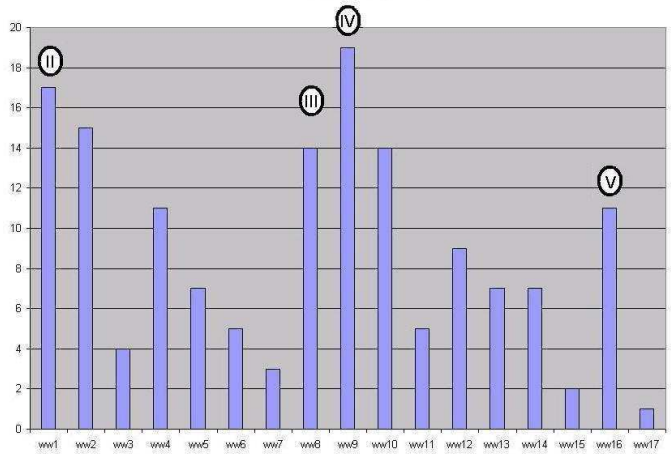

Fig. 3 – Bugs Found x Working Week

By analyzing the bug rate during each step of the verification effort - see fig. 3 -, it can be seen that a new bug rate peak appeared at the beginning of each new phase: directed patterns on RTL standalone (II), directed patterns on RTL with Response Checker (III), random patterns with Response Checker (IV) and debug interface patterns (V). It is interesting to note that the bug rate rose when the response checker was introduced with directed patterns (III). Around 25% of all bugs caught when running directed self-checking patterns were reported by the Response Checker, not by the pattern checking itself. It shows clearly how the Response Checker improved the testbench in terms of observability, when compared to the same set of patterns run on RTL only, having the same controllability.

4. Possible Improvements

The testbench currently does not allow two patterns to run in parallel in the C side, stressing ETPU behavior in a system situation closer to real life. This limitation is minimized by the parallelism of the eTPU itself, allowing microcode threads to run in a time-sharing basis or in actual parallelism on a dual engine system.

Some functional aspects of the eTPU seem particularly suitable to formal verification: the arbiter, the scheduler and the channel logic. Assertions were already developed for the first two.

5. Conclusions

The early building of a conformance model was a valuable activity in itself, making the verification team to have an in-depth knowledge of the specification, also helping to identify omissions and obscure points in the spec. Early model delivery, combined with a single stimulus API used for all testbenches, allowed the verification team to start writing and debugging patterns independently of the design team.

The cycle-accurate nature of the conformance model allowed RC implementation at the signal level, making it relatively simple and quick to deploy. A near-optimum level of observability was achieved by comparing the model and RTL inputs, outputs, internal registers and states at each clock cycle.

Timing critical patterns previously written and debugged on the model could be run on the RTL with little or no hanges. On the down side, timing mismatches between the model and the RTL were frequent, almost always in cases were the particular implementation was not relevant from the functional point of view, demanding a considerable workload. The design and verification team should inform each other and reach an early agreement whenever the specification gives margin to diverging implementations. These implementation details did not have necessarily to be back annotated into the specification, but could for instance be recorded and followed up in a bug tracking system. Working very closely with the design team was very important for an effective debug process.

Constrained random patterns proved to be an outstanding resource to find corner case bugs. A relatively small set of random patterns caught a large amount of bugs. A numeric comparison of the results/effort ratio between random and directed self-checking patterns is difficult, considering the workload taken to develop the model and the conformance testbench, and that directed self-checking patterns were part of their debug effort. However, it became very clear that random patterns exercised a large number of cases not covered by the much larger directed pattern set.

PHP usage for automatic eTPU assembly code generation, both for directed and random patterns, allied to the power and flexibility of the language has greatly enhanced the coding productivity.

A stimulus API was defined early on in the project and, although evolving and incorporating new required calls, was consistently and exclusively used along the project in all patterns, directed or random. This allowed the patterns to run unmodified in all simulation testbenches, and it also facilitated porting simulation patterns to run on the MPC5554 evaluation board.

A structured testbench with a well defined-class hierarchy also contributed to the reuse of patterns across testbenches. The three-layer API, however, added to the maintenance effort. Testbuilder [6] was used only at the lower (signal wiggling) layer to avoid tool limitations.

OVA assertions were used to improve the functional coverage for some higher risk sub-blocks. Functional coverage assertions identified uncovered cases that demanded adjustments in the random stimuli constraints. The benefit of property checking assertions was less clear because it started late in the flow and it overlapped with the conformance checking by the eTPU CModel. All bugs reported by OVA were also reported by the RC.

Acknowledgment

The authors would like to thank the eTPU verification and design teams at Freescale's Brazil Semiconductor Technology Center (BSTC), who made this work possible; César Dueñas, for his valuable contributions; Mike Pauwels, Jeff Loeliger and Richard Soja from the Freescale's TECD application teams in Oak Hill, TX and East Kilbride, Scotland; and Andrew Klumpp at Ashware Inc.

References

[1]Semiconductors Reuse Standards – http://www.freescale.com – search for SRS

[2]Baray F, Conognet P., Diaz D. and Michel H., "Validation of functional processor descriptions by test generation"

[3]Zhang E. and Yogev, E., "Functional Verification with Completely Self-Checking Tests", Verilog HDL Conference, 1997., IEEE International , 31 March-2 April 1997

[4]Monaco, J.; Holloway, D.; Raina, R., "Functional verification methodology for the PowerPC 604 microprocessor", Design Automation Conference Proceedings 1996, 33rd , 3-7 June 1996

[5]PHP – http://www.php.net

[6]Testbuilder – http://www.testbuilder.net

[7] Synopsys, "OpenVera™ Language Reference Manual: Assertions, Version 1.4", Synopsys.

[8] VSIA, "Taxonomy of Functional Verification For Virtual Component Development and Integration, Version 1.2"

[9]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.

[10] Nexus 5001 Forum – http://www.nexus5001.org