

# Verification of the PalmDSPCore<sup>®</sup>

## Using Pseudo Random Techniques

### ABSTRACT

The advanced variable size scalable architecture has many features that present significant verification challenges to ensure correct implementation. This paper describes a functional verification system and methodology to address the requirements of correct verification of the PalmDSPCore<sup>®</sup>.

In particular, the paper presents an efficient tool and methodology to generate random assembly tests, in accordance with the dynamic change of the processor's states. It describes both random and pseudo-random scenarios and describes how to feedback this process to be sure that the Device Under Test (DUT) was adequately tested.

*Keywords:* verification, pseudo-random test generation, self-checking, coverage, PalmDSPCore, Specman Elite.

### Introduction

The complexity of modern microprocessors has made design verification a huge bottleneck for large chip designs [2]. Nowadays, verification efforts seem to consume most of the design resources. Often, there are more verification engineers than designers, making verification a Time To Market (TTM) limitation [1]. Simulation-based verification is the primary means for functional verification.

The ever-increasing frequencies, the growing amount of interacting functional modules on a chip and the increasing complexity have created new and more complicated problems. Recent publications show that pseudo-random test generation techniques have been the backbone of the verification effort in the industry[1,2,7,8].

The paper presents one of the environments that were used to verify the PalmDSPCore [5,6]. The PalmDSPCore is a family of fixed-point Digital Signal Processor (DSP) cores, which uses innovative parallel architecture. Its three

members feature high performance and variable data-width (16, 20, and 24 bits). The PalmDSPCore is a dual MAC (multiply-accumulate) and parallel functioning architecture, able to execute up to 18 operations in a single cycle.

The PalmDSPCore includes seven arithmetic units supported by three powerful instruction-sets: single, parallel and multi-parallel. The instructions are executed in a single cycle, including rich digital signal processing oriented instructions, as well as instructions suitable for control functions (CPU oriented). SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) style of instructions are also included. The instruction width is variable (16/32 bit only).

The uniqueness of the environment is that it uses the results from the previous n-1 instructions to create the instructions for state n. The paper describes both the previous verification environment and the current verification environment's architectural layout. The current verification environment uses Specman Elite [2,8]. The paper also illustrates how data flows in this environment, and its advantages over previous ways to achieve the same goal.

In later stages the paper goes into a more detailed description and shows how a user test is written, how an individual instruction gets generated and how generation feeds back as input for subsequent generation of instructions. It also discusses briefly the Specman Elite capabilities that were used to achieve these goals.

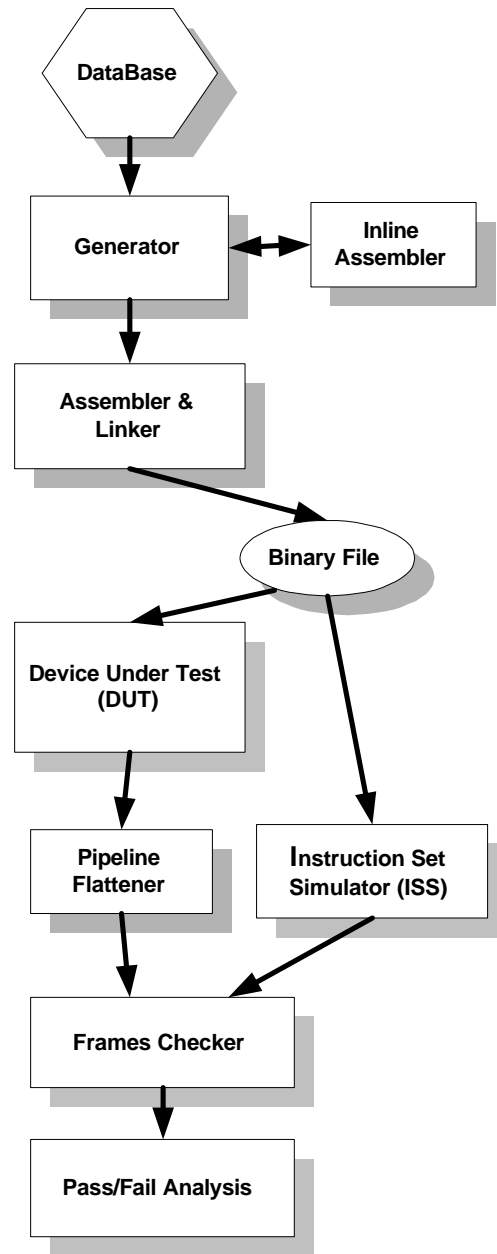
### Verification Environments

The previous verification environment was based on generating pseudo-random programs that were correct by construction. In order to create a correct by construction program and due to the large complexity of the PalmDSPCore, many preventative techniques were used, such as locked registers and memory as well as

prevention of the usage of certain instructions. Even then it was very hard to ensure that the program was correct. Checking was performed by comparing outputs from a reference model to the DUT.

The current verification environment is a dynamic-feedback instruction-generator. The instruction created for the program for instruction n is affected by all the results that were stable after the execution of the preceding n-1 instructions on a reference model. Checking of the expected results is performed after each basic block of instructions was executed on the DUT.

## Previous Verification Environment



**Figure 1** Block Diagram of Previous Environment

## **Components in the Previous Environment**

*DataBase*: A database of all PalmDSPCore instructions, registers and operands.

*Inline Assembler*: An inline assembler compiler that compiles a given single instruction and produces its encoding (after checking that the syntax is correct and has not violated simple architectural restrictions).

*Generator*: A pseudo-random static generator that generates correct assembly tests, using the *Inline Assembler*.

*ISS*: An Instruction Set Simulator – simulates the core on instruction by instruction basis, producing accurate registers and memory locations. This is not a cycle accurate model.

*Assembler & Linker*: An assembler compiler that compiles a given assembly file and produces its encoding. The Linker links all pieces of code into the memory.

*DUT*: Device Under Test - the RTL model and all associated environment.

*Pipeline Flattener*: A special component that flattens the pipeline so that the pipeline can be viewed on instruction-by-instruction basis (as the ISS does).

*Frames Checker*: a simple diff utility that tries to match the frames from the *DUT* and the *ISS*.

## **Data Flow in the Previous Environment**

1. The generator generates a full pseudo-random program on a static basis. A complete test is generated before actually executing it (thus the test is generated “statically”).
2. Once a test is generated (assembly file):
  - The test is translated into a binary file (using the *Assembler* and *Linker* tools).
  - The test is loaded to the *DUT* and *ISS* memories.
  - The test is executed on the *ISS* and on the *DUT*.
  - All register values and values written to memory are collected after each instruction (called “status frames”). On the *DUT* side, a *Pipeline Flattener* is

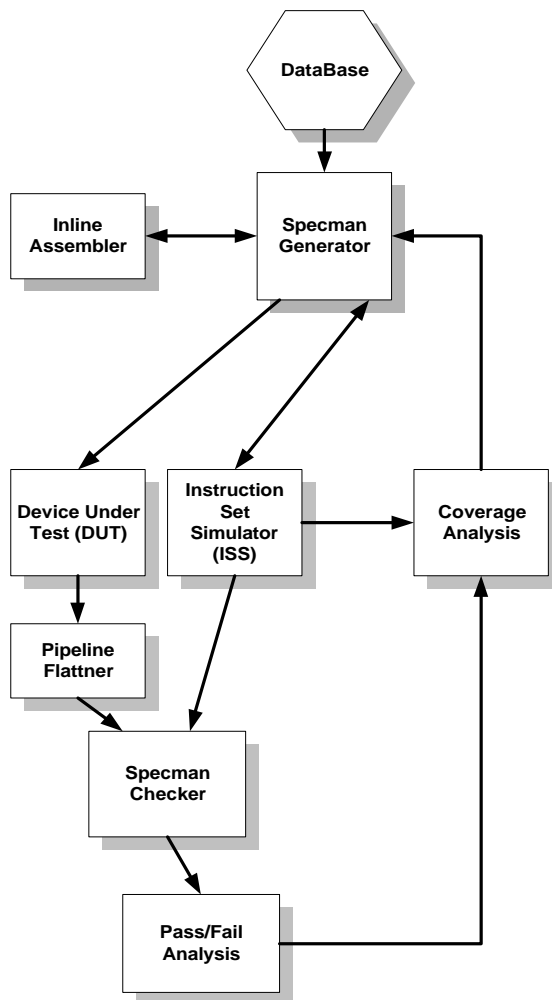
used to do this. Status frames are written into text files (separately for the *ISS* and for the *DUT*), later these text files are compared to provide a pass/fail indication (using a “diff” mechanism).

## **Disadvantages of the Previous Environment**

1. While generating the test, the generation system cannot know the exact processor status. Hence, in order to create correct by construction random assembler programs, several assumptions must be made, for example:
  - Locking certain values in registers, in order to make sure that these values are not changed. These registers are “locked out” of being used by the generator.
  - Locking access to certain memory banks.
  - In order to ensure that the right values, modes of operation and other information within the processor fit the generated instruction, special preparations (in the form of assembler instructions) are inserted prior to that instruction (thus biasing machine state).
2. The static method of test generation yields many uncovered spots (without any ability to know about them), due to the great complexity and different modes of operation of the PalmDSPCore.
3. This verification method also produced some non-tested elements, such as external events like interrupts and reset. These elements had to be fully covered using other methodologies.

The generated assembly file cannot hold any information on external events. Those events are not instruction-driven, so the *DUT* and the *ISS* could not possibly acquire information on events using this method.

## Current Verification Environment (Specman Elite)



**Figure 2** Block Diagram of Current Environment

### Components in the Current Environment

The *ISS*, *Inline Assembler*, and *Pipeline Flattener* are the same as before.

Additional components are:

*Specman Elite*: A tool for high-level verification methodology encapsulating the environment and all the other components (*DUT*, *Inline Assembler* and *ISS*).

*Specman Elite Generator*: A generator built in the Specman tool to create correct PalmDSPCore assembler programs.

*Specman Elite Checker*: A checker analyzer of test results.

*Specman Elite Coverage Analysis*: The component that collects high-level coverage information, and enhances the generation process with unexplored areas.

### Data Flow in the Current Environment

The following steps occur for each instruction

1. Instructions are generated according to the test file themes requested by the user (see example test file below).
2. Each instruction is compiled by the *Inline Assembler* program to yield an encoding that is dynamically loaded into the *ISS* memory and executed.
3. Each instruction results are recorded in a special data structure (the “Status Frame”).
4. The machine state is updated and serves as an input for the next generated instruction.

After a sequence (“block”) of instructions is generated, and executed on the *ISS*:

1. Encoding (for the whole block) is loaded into the *DUT* memory and executed. Instruction results are recorded in Status Frames using the pipeline flattener.
2. Instruction Status Frames are compared and mismatches are reported.

After a certain amount of tests were executed, coverage information is collected to be used in order to enhance subsequently generated tests (i.e., put more weight on uncovered areas of the instructions).

### Advantages of the Current Environment

In addition to fulfilling all the previous environment’s shortcomings mentioned in the previous section, the current verification environment offers the following advantages:

1. Error localization. Only the sequences of mismatched instructions need to be investigated in order to find the cause

(possibly a bug). In the previous method, the whole test failed/passed with no particular indication.

2. Corner cases can be reached based on previous state analysis, yielding better coverage of the instruction set.
3. There are metrics presenting the amount of coverage achieved by the generated tests. These metrics enhance later runs of the generation process by putting extra weights on uncovered areas.

## **Pseudo Random Generator**

For each instruction that the pseudo random generator creates, there are three levels of restrictions that must be adhered to:

- *Syntax*: The generated (pseudo random) instruction must be correct in terms of assembly language syntax and usage of operands, switches etc.
- *Run-time Restrictions*: There are some run time restrictions, such as memory banks of the register pointers, that must be handled in run-time.
- *Special Instructions*: A correct special instruction sequence must be created when special instructions are generated, such as FFT support instructions or nesting of interrupts.

The Specman Elite built-in generator has the methodologies to solve all the above levels of constraints and also to assign higher weight to the areas that were not covered (in terms of instructions, sequences, processor state, etc').

## **Self Checking**

Each group of instructions is called a block. Every block is generated-executed on the *ISS*, instruction by instruction, each instruction creates a data structure of all the registers, and data memory writes. This structure is maintained within Specman Elite (using its built in listing capabilities).

Afterwards, the instructions are loaded into the *DUT's* memory and executed. When they terminate their execution, they are stored inside Specman Elite.

The next stage is to compare the instructions one by one. If a mismatch occurs, the mismatching

structure is displayed as well as an assembly test file demonstrating the mismatch.

## **Generation Feedback**

There is a major advantage in the generator being able to look both into the architectural model and the *DUT* in that the current instruction executed can be taken into account in the generation of the next instruction.

This methodology also lets you "grade" your current generated tests, and generate the consequent instructions by leveraging on this grading information. This process is called "coverage grading" in Specman Elite terminology.

## **Sequences and Test Writing**

A test file is a test that an individual user executes. In order to recreate the exact test and run it, the user needs to save the following:

- The *e* test file.
- The random seed that the test was executed with Specman Elite

This ensures that the test runs the same.

Following is a sample test file (in *e* language):

```
<
--
-- All these are instructions that I do
-- not want to test in this test
--
extend palm_inst {
    keep code not in ["mov...", "add",
"mpy..."];
};
--
-- Enabling interrupts
--
extend sys {
    post_generate() is also {
        interrupt_enabled = TRUE;
    };
};
--
-- Only one hardware loop type is enabled
--
extend sequence {
    when bkrep sequence {
        keep          bkrep_type      ==
bkrep_immediate;
    };
};
--
--
-- Interrupts have a chance of 1-10 to
-- appear.
--
```

```

extend interrupt {
  keep soft i_kind == select {
    9000 : 0 ;
    1000 : [1..255];
  };
  -- All interrupt routines are
  -- function blocks.
  keep for each in seq_list {
    it == call_ret;
  };
};

```

## Debugging Environment

The debugging environment is based on the fact that the random seed together with the *e* test file determine how to run the test. Hence, reproduction of a specific test is very easy.

Each sequence of code (“block”) that is generated maintains its waveform database. If a mismatch is detected, the waveform displayer is used to debug it. There is no need to rerun that specific test (which might take a long time) in order to start debugging it. In addition, an assembly file that represents the generated test is produced, enabling easier debugging and reconstruction of the problem.

Another very affective debugging method can be the built-in Specman Elite debugger that can probe certain signals and events.

## Bug Statistics

Following are bug statistics of a period of 20 weeks in four-week resolution. The Y-axis is the amount of cycles that were executed in order to find a bug and the X-axis is the workweek (1-20). The chart demonstrates the number of cycles that must be executed in order for the environment to find the next bug in the *DUT*. In order to avoid weekly peaks in the cycle per bug ratio, the resolution used for calculating these numbers is of four weeks.

Until week eight, many bugs were found but after week 6 there is a constant improvement. Between weeks 16-18 there is a setback due to the introduction of support for interrupts.

From week 20 on, the bug rate is significantly improving, reaching 100 Million cycles per bug in week 26 (not shown).

## Specman Elite

Specman Elite provides a comprehensive environment for all aspects of verification:

- Automatic generation of functional tests
- HDL simulation control
- Data and temporal checking
- Functional coverage analysis

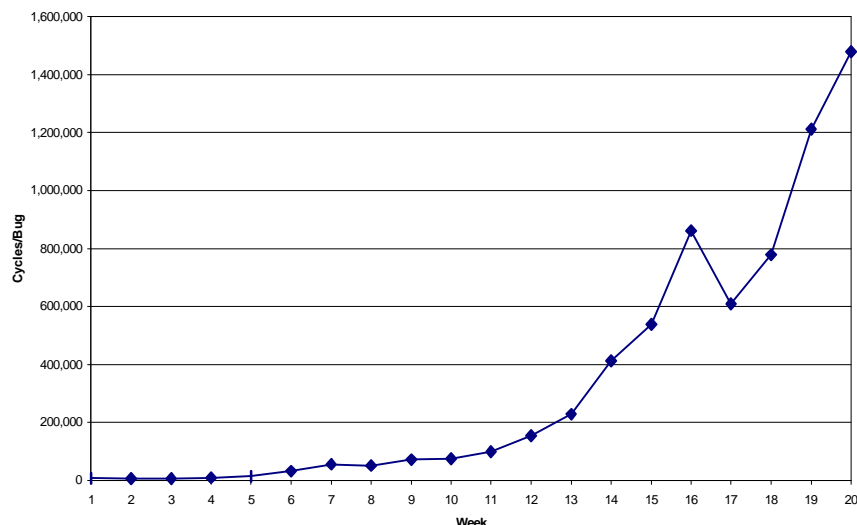
These components are integrated in a complete solution for the functional verification of complex electronic systems and ASIC/ICs.

## Constraint-Driven Test Generation

Specman Elite offers a powerful constraint-based approach that lets you direct the test generator to areas of interest rather than tediously composing tests. The generation process is based on the use of constraints. Constraints are used for two purposes:

- Specification constraints define the legal ranges (which are sometimes context dependent).
- Test constraints direct the generation to specific scenarios of interest.

These constraints can access signals from the HDL, so the generation is taking into account the current state of the *DUT* at any given time, allowing for the automatic capture of even hard-to-reach corner cases. Specman Elite's powerful constraint solver dynamically resolves all the specified constraints and then generates values that are within the target scenario.



## **Data and Temporal Checking**

Specman Elite facilitates the creation of data and temporal checks. Checks are always active as an integral part of the environment, reducing chances of unnoticed failures. With Specman Elite it is easy to automate self-checking, thus eliminating the need for post-run inspections. Both reference-based and rules-based checking methodologies are supported. Since the environment is integrated (test generation, checking, coverage), the checking methodologies supported are robust and efficient. For example, since checking and generation can be intermingled, the following sequence is possible:

- Generate a packet
- Send to simulator
- Wait  $n$  cycles
- Execute checks
- Generate another packet

## **Simulation Control**

Specman Elite is integrated with all of the leading VHDL and Verilog simulators and offers 100 percent controllability and observability into the HDL. Internal signals of the *DUT* can be sampled and driven by Specman Elite. The interface to the simulator is straightforward and easy to specify.

## **Functional Coverage Analysis**

Specman Elite's functional coverage capabilities gives a meaningful metric measuring and directing the verification process. This measures the progress of verification and automatically identifies holes (uncovered spots) in test coverage.

The combination of functional coverage analysis and constraint-driven generation is what ensures the most efficient verification process. Goals are defined and results are quantified. Once holes are uncovered, simple test constraints target the generator to eliminate these holes. This combination ensures that every test adds coverage and avoids wasted simulation cycles. Any user scenario can be covered including state machines, data, transitions, multi-cycle events and context-dependent events. Functional coverage analysis also allows tracking of progress and makes verification schedules more predictable.

## **Conclusions**

The paper presented two verification environments used to verify the powerful and complex PalmDSPCore. The first method was a static type of random tests generation. It was used to verify the PalmDSPCore at the early verification stages, where no special and complicated tool was required. This environment has a few drawbacks, most of which are based on the method of generation.

Main disadvantages of the previous environment were the non-fully-random generation, no way to invoke external events (e.g. interrupts) and no coverage accumulating.

The upgrade to the new environment using Specman Elite was required at a certain stage, where the old method failed to find bugs in a reasonable rate. Specman Elite succeeded to fill all gaps, mainly due to its native generation capabilities. Its dynamic pseudo random generated tests proved to be much less restrictive, fully covering the processor's behavior and modes of operation.

The coverage information, accumulated for all tests throughout the verification stage was very helpful in terms of progress estimation, as well as uncovered spots.

In addition, the new environment succeeded to re-use most of the previous environment's components, saving a great amount of development time and relying on already well proven components.

## **Acknowledgements**

We would like to thank all the members of the SDT group in DSP Group for their continued support, especially Menachem Stern, Yair Siegel and Yagil Gantz.

## **References**

- [1] "Spec Based Verification: A new methodology for Functional Verification of Systems/ASICS", white paper, Verisity design Web page: [www.verisity.com/html/technical\\_papers.html](http://www.verisity.com/html/technical_papers.html).
- [2] Verifying Large Scale Microprocessors Using an Abstract Verification Environment,

Denis Abts, Mike Roberts, DAC 1999 (session 10-1)

[3] “*e* Language LRM (Language Reference Manual)” Verisity Design.

[4] Verilog XL Reference Material, Cadence Design Inc.

[5] PalmDSPCore Architectural Definition.

[6] PalmDSPCore Software Development Tools User Manuals.

[7] D. Geist, G. Biran, T. Arons, M. Slavkin, Y. Nustov, M. Farkas, K. Holtz, A. Long, D. King, S. Barret, “A Methodology For the Verification of a ‘System on Chip’,” DAC, 1999.

[8] C. Hanoach, “High Level Verification Automation: A New Methodology For Functional Verification of Systems/ ASICs”, DesignCon, 1998