DOI 10.31114/2078-7707-2019-2-2-7

System of Combined Specialized Test Generators for the New Generation of VLIW DSP Processors with Elcore50 Architecture

A.V. Garashchenko[1,2], A.V. Nikolaev[1], F.M. Putrya[1], S.S. Sardaryan[2]

[1]Open Joint-Stock Company Research & Development Center «ELVEES», Zelenograd, Moscow

[2]National Research University of Electronic Technology (MIET), Zelenograd, Moscow

ant.gar1@mail.ru

***Abstract* — In connection with the architectural complexity of modern multi-core structures, more than 60% of the design resources are spent on verification during the development of the processor. Automatic generation of tests is often used to increase test coverage and reduce overall test time. Therefore, the creation of verification test generators to verify the correct operation of microprocessors is becoming increasingly important.**

This paper describes the technique of development of the several tests generators used for microprocessor verification. The first one is designed for VLIW package generating. The second one is for the verification of the control flow. With the help of it sequences of assembler instructions are created to check the pipeline. Software and hardware cycles, subprogram calls, conditional and unconditional conversions are possible. The third generator is aimed at checking the cache memory of processor. It is based on the graph model of the memory subsystem built by its description.

In the suggested approach, source code of the tests is constructed by using combinatorial techniques, that is all possible combinations of instructions, situations, and dependencies are sorted taking into account the constraints that direct the tests to check certain situations and also exclude the possibility of generating infinite cycles. The generated test sequences allow for various tests.

To create more complex tests possible integration of generators into each other is considered, since the interaction of different devices of the processor generates a large number of critical situations. The proposed approach makes it possible to improve the efficiency of microprocessor testing.

***Keywords* — verification of processors, wide command word, memory subsystem, test generation, coverage.**

I. Introduction

Due to the architectural complexity of modern multi-core processors used in developing systems on a chip (SoC), more than sixty percent of design resources are spent on their verification. This is because the high combinatorial complexity of checking the correctness of the work of both single cores and the system as a whole. In addition, there is a tendency of SoC complication due to the increase of heterogeneity associated with the need to increase the speed of performing certain classes of tasks. Such computing systems consist of general-purpose and specialized computing cores. That is, the task is to check the implementations of different architectures of the computational cores that are part of the designed SoC, which may differ not only by the command system, but also by the way the commands are assembled into a VLIW package (VLIW - Very Long Instruction Word), its width, the organization of register files, caches and much more. The computational complexity of modern formal verification algorithms limits the scope of their use by small blocks (such as ALU or single elements of the memory subsystem) or separate processor properties that are easily localized, which sharply limits the use of formal methods for tests of the full processor core level [1-4]. In this way, dynamic verification is still one of the main methods for checking computational cores.

If we are concerned with the verification features of specialized microprocessors of digital signal processing (DSP), then among them the wide combinatorial space of possible situations is worth highlighting. Often, these processors have a Harvard VLIW architecture with scalar and vector executive channels, hardware cycles, and multi-channel memory. Their verification requires a huge amount of complex tests, which becomes the main problem of functional verification. However, the limiting factor is the time required to develop a complete test suite. Consequently, there is no question of writing all the tests manually. This determines the high relevance of creating new tools to check the correctness of the work of such structures.

To increase the simulation speed, and, therefore, validation checks, each processor core and even some of its modules are verified in autonomous test environments (provided that the processor design is written taking into account the requirements of the verification decomposition task [5, 6]) since the simulation speed of individual parts of the processor is much higher. However, the task of creating a comprehensive set of test sequences for the processor core remains critical. Not all subsystems can be localized as a separate unit, and even where it is possible, errors in the inter-unit communication protocol can occur. Even if you do not go into the details of the implementation of the micro-architecture, the state space of the modern core is astronomically huge. It is determined by the combination of the instruction set in the VLIW instruction, the dynamic combination of instructions and dependencies between them in the pipeline, the dynamics of memory accesses that can be executed some in the VLIW processor within the same instruction, external interruptions, and the state of the debugging subsystem. Test generators have long been serving as the main tool for covering computational kernel tests [7–9]. However, the amount of state space is such that one general-purpose generator will be excessively complex and with a rare emergence of critical situations, which will lead to an unacceptably long process of generating and running tests until the required coverage is achieved. Specialized test generators for individual subsystems or subsets of processor properties, for example, separate test generators for a set of software control commands, such as a memory system, an interrupt subsystem, allow you to create a large number of critical situations for the target subsystem and, accordingly, to achieve greater coverage. However, from the point of view of the system, specialized generators cover only a few localized subsets of the global state space, leaving large gaps between the subsets in this space. An example of an intermediate state not covered by specialized generators may be an interruption at the time of execution of a given combination of software control commands parallel to the execution of cache-addressing commands that conflict with each other when the cache line is accessed. A completely random flow of commands will create such a state through the years of modeling, and the directional specialized tests will not create this state at all since they use patterns or models that limit their behavior.

This paper proposes a solution based on a system of combined specialized generators that allow you to organize hierarchical calls of different generators in the process of creating a test, and, thus, to expand the covered subsets of the global space of processor states. For example, a combination of a generator for software control (generation of exceptions) with a generator for a memory subsystem (exceptions under conditions of a long blocking of the pipeline). Such tests check the system as a whole since the simultaneous operation of many parts of the processor core creates critical situations, the probability of occurrence of which, with their directed verification, is very small. To generate them, it is necessary to take into account existing methods and approaches for testing various processor units. The generated test sequences allow you to check the interaction of various devices of the processor core in one test.

II. Model of Random Instruction Flow Generator

The architecture of the model, allowing to solve the problem of generating random sequences, is shown in Fig. 1. Tests for a DSP instruction set are sequences of assembler instructions. Such testing allows you to verify these instructions for correctness from a behavioral point of view by testing random input data. For each instruction, the random data are the initial state of the used registers (both input and output), the numbers of these registers, the state, and addresses of memory cells in formats that require transfers. The generated program is designed to run on the RTL model (register transfer level) and the simulator. To use such a generator when verifying another microprocessor, minimal modification of the program is necessary, namely, to change the configuration file containing assembly instruction mnemonics and their formats.

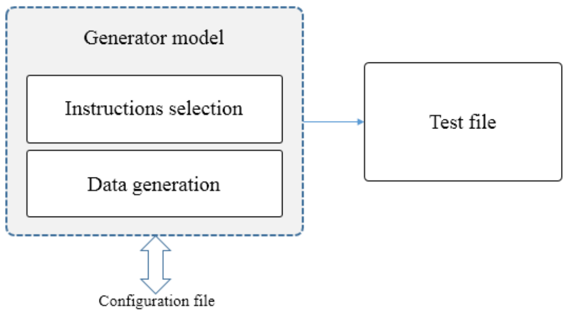


Fig. 1. Random instruction generator architecture

The generation of test data includes the random selection of the instruction being tested from the list obtained from the configuration file, the choice of a format that is possible for this instruction; selection of registers for instructions, depending on the format of transfers; initialization of the state register. For a given instruction and parameters, a reference value is calculated. When initializing the values of 64-bit and 128-bit registers, intermediate transfers are used through two 32-bit registers, therefore they are reserved and are not used as sources or receivers of computational commands and transfers.

There are two possible scenarios for using this generator model.

In the first scenario, the test consists of several subtests that are written to the PRAM memory of the processor. Each subtest consists of initializing initial data (registers and memory), executing one test command, checking the result and writing the result of the execution to memory. By default, the program generates tests for the first work scenario.

In the second scenario, the test consists of initializing the register file, address registers and the corresponding memory cells with initial arbitrary values and executing a specified number of arbitrary instructions. The number of generated instructions is limited to 4000. It supports the ability to set the proportion of basic instructions and, accordingly, extended. The result of the test is the state of the register file. The correctness of the test is checked by comparing the architectural state of the RTL and the reference processor model.

A feature of this generator is the ability to create VLIW packages containing vector and scalar instructions. The generation of packets of a given length and random (limited by the maximum) length is supported. The instructions fall into one package, taking into account the limitation of the number of commands of this type that can be executed within one VLIW package.

III. Model of control flow generator

When verifying the DSP processor control unit, one should take into account the peculiarities of its architecture. The main object of testing is the microprocessor pipeline. Basically, situations are checked that lead to various locks. Within the framework of the generator model under consideration, there is a need to cover the blocking space caused by software transitions, subroutines, hardware cycles, and exceptions; as well as locks caused by data dependencies between successive instructions.

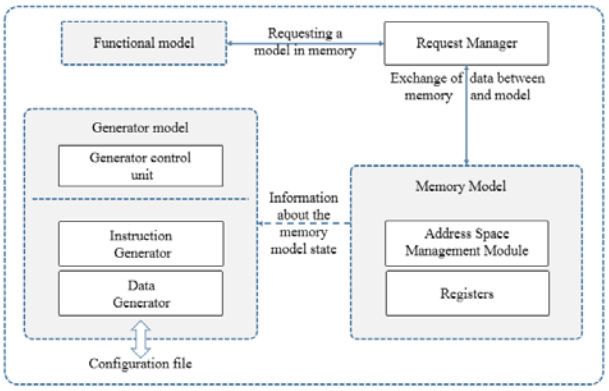


Fig. 2. Block diagram of control flow generator

The structure allowing to solve this problem is shown in Fig. 2. The generator control module sets the general test generation rules described in the configuration file. They arrive at the input of generators of instructions and data, which carry out the formation of sequences consisting of executable code, the connections between them; and the creation of dependent and independent operands for instructions, respectively. To calculate the values of registers and memory, the resulting code is run on the functional model, in addition to providing the possibility of obtaining the current state of the system under test as a whole. The request manager is responsible for processing requests for a functional model to the memory for instructions and data.

With the help of such a generator it is possible to generate tests of the form shown in fig. 3, where program conditional and unconditional jumps are available for relative (j-commands) and absolute (b-commands) addresses with delay slots, as well as preserving the return address; subroutine call, including calls from the subroutine (except recursion); nested hardware cycles [10].

The main components of the section with transitions are sequences of two types: seq_n, seq_subroutine_n. Seq is a piece of code consisting of random instructions connected by conditional or unconditional jumps. Seq_subroutine - routines with their own stack. Transitions are generated with both positive and negative offset.

The subroutine call is implemented by passing the return address, as well as arguments through registers to a function with further writing to memory (common stack). A pointer to the top of the stack is placed in memory at a known address, and this address is generated randomly, taking into account the limitations of the microprocessor memory card. When writing to memory occurs, the pointer is incremented; when retrieved, it is decremented. After executing the function body, the address is read from the stack to the register, then jump to this address takes place. It is possible to call subroutines from other subroutines, there are limitations from looping, taking into account possible indirect or direct recursion. The number of subroutine calls is limited by the size of the stack.

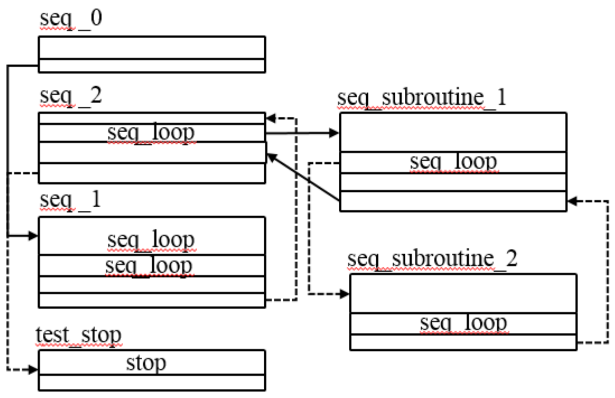


Fig. 3. Test structure

Generation of software and hardware cycles is implemented. Program cycles are made using conditional and unconditional jumps. The depth of the cycles, as well as the number of subroutine calls, is specified in the configuration file. However, it should be noted that the depth of hardware cycles is limited by hardware, therefore, when generating cycles in subroutines, their nesting before this is taken into account.

A distinctive feature of this generator is the ability to create exception handlers whose addresses are placed in a special register. In addition, three types of exceptions are generated:

1) UI (Unknown Instruction) - unknown instruction;

2) BA (Bad Address) - wrong address;

3) SYSCALL - system calls.

The first type of exceptions is created with the help of various jumps to addresses with nonexistent instructions, as well as direct recording in the register of exceptions.

The second type is generated by accessing the PRAM memory at addresses that are not allowed by the DSP processor or by Scatter-Gather accessing in a cache a single address that does not belong to the PRAM memory range.

In the example above, a part of the generated test is presented, consisting of one sequence of the seq type, in which there are two calls to the subroutine seq_subroutine_1. In the seq_subroutine_1 subroutine, the return address is first saved, then the stack pointer is shifted up 4 units, after which the subprogram seq_subroutine_2 is called in the nested loop. Commands are separated by randomly generated instructions. At the end of seq_subroutine_0, the return address to register 31 is read, the pointer to the top of the stack is shifted down 4 units and a transition occurs at the address in register 31.

Example:

```
seq_subroutine _1:
   stl r31, (r0)
   addl 4, r0, r0
   do 4 llsub_11_end
   xor r12, r1, r3
   minl r7.l, r4.l, r6.l
      do 6, llsub_12_end
   bs seq_subroutine _2
      llsub_12_end:
   subl 4, r0, r0
   llsub_11_end:
   ldl (r0), r31
   b r31
```

```
seq _0:
   subl r22, r12, r15
   addl r22, r17, r19
   bs seq_subroutine _1
   bs seq_subroutine _1
   stl r0, (r=0x1200000)
   j test_stop
```

Despite the presence of a functional model in the generator, to test the microprocessor core under test, the subjects of the discrepancy between the behavior of the processor under test and the reference model mainly use the mode of comparison the RTL and simulator (TLM model). The advantage of this approach is that the verification of the correctness of the behavior of the components is performed automatically.

IV. Model of Cache Hierarchy Generator

The construction of tests with this generator is carried out by constructing a graph model of the cache memory hierarchy, the vertices of which are cache states (Fig. 4), and the edges are state transitions (instructions written in a metalanguage).

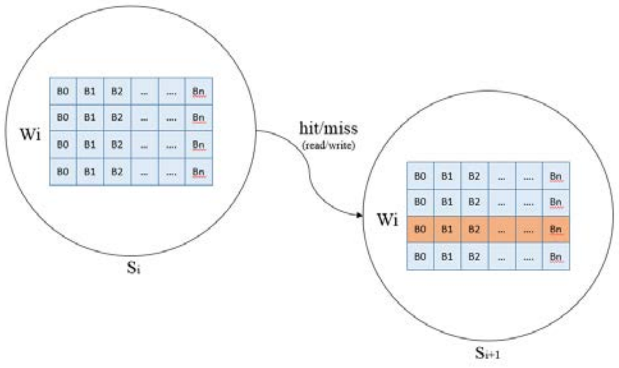


Fig. 4. Graph model of cache memory hierarchy

The metalanguage makes it possible to work with the microprocessor's memory at a more abstract level, decoupling the tests from a specific assembler, which allows you to transfer tests to verify processors with another architecture. It contains read functions and write functions. Vector-block work with memory is provided. A special translator has been developed for converting metalanguage into assembler commands, which supports no more than 16 vector-block memory accesses.

The following transitions are possible between the vertices:

1) hit(Cache L, int mode) – hit in the cache L by reading or writing. The mode parameter determines which address will be used: if it is 0, a random cached address is used; in the event that mode is 1, the address of the last executed command is used.

2) miss(Cache L) – miss by reading or writing in the cache L.

3) parallel_hit_miss(Cache L, int _mode=0) – parallel operations of hit and miss on reading or writing in the L cache on several ports. Parameter mode works as in the case of a hit.

4) replace(Cache L) – evicting the line used in the last executed instruction in the L cache.

5) next_line(Cache L) – reading or writing in two adjacent lines from the line used in the last executed command, but there is no access to the line itself.

6) next_way(Cache L) – reading or writing in a random associative way for the string used in the last executed command, while there is no access to the line itself.

7) nop() – nop instruction.

8) gather_scatter(int addr_banks_mode, int command) – addresses are sent to one bank or to random banks with the ability to set the type (read /write) of command.

9) reset_mem_model() – graph memory model reset function.

At generation of all transitions, the address, the type of circulation, and the size of the transaction are set randomly. An example of a graph for a row of one cache and two associative paths is shown in Figure 5.

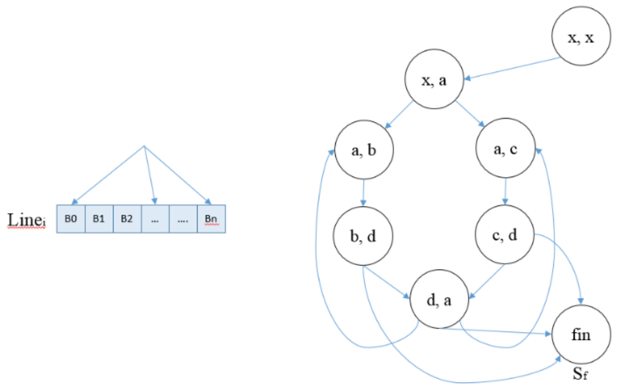
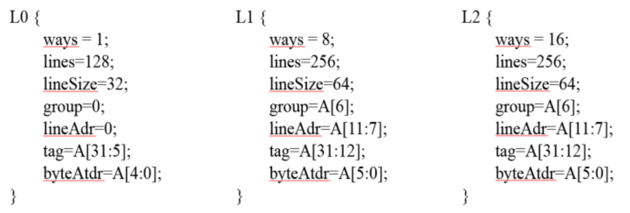


Fig. 5. Graph model for a single cache line

According to the description given in the configuration file (Fig. 6), a graph model is built. Each transition in the graph is a separate test. First, a preamble is generated, consisting of sequences of metalanguage commands that bring the cache memory system to the desired state; followed by a block of memory dump; then the instruction responsible for the transition; the block of checking the memory dump with the reference model.



```
L0 {                        L1 {                        L2 {
     ways = 1;                   ways = 8;                   ways = 16;
     lines=128;                  lines=256;                  lines=256;
     lineSize=32;                lineSize=64;                lineSize=64;
     group=0;                    group=A[6];                 group=A[6];
     lineAdr=0;                  lineAdr=A[11:7];            lineAdr=A[11:7];
     tag=A[31:5];                tag=A[31:12];               tag=A[31:12];
     byteAtdr=A[4:0];            byteAtdr=A[5:0];            byteAtdr=A[5:0];
}                           }                           }
```

Fig. 6. Cache hierarchy description

Possible critical situations for the cache (without taking into account the coherence) created by the generator in question: chains read-write, write-write-read-read; accessing by addresses after the data cached by them has been evicted; eviction of a line with further reading from it; writing or reading data crossing the edge of one line in one associative way to suppress two lines in it at once; writing or reading data crossing the edge of one associative way in order to evict two lines at once in two different ways. Generating invalidation commands allows creating additional load on the cache system. In the future, it is intended to impose all this on the coherence mechanism.

To generate tests for a bunch of L1 and L2 caches, we need to additionally specify the following points in the configuration file: whether L1 is inclusive; how the displacement from L1 and the displacement directly from L1 and L2 takes place; how the miss in L0 with the eviction in L1 takes place.

A distinctive feature of the generator is the support for the generation of test scenarios for multi-channel access mode in a cache memory. The default option is a preamble consisting of a sequence of commands of the metalanguage on one channel with a simultaneous transition to different states on all valid channels and checking data by comparing with the reference model. Also, random instructions for accessing external and internal memory have been added to test generation.

V. External interrupt generation

An external interrupt, generally coming asynchronously to the command execution flow, should result in the exit into the handler, its execution, and return to the execution process of the main control flow in an initial state of the pipeline. External generation of interrupts for the RTL model is easy to organize. However, there are two problems.

The first problem is to combinatorically go through the states of the core at the moment the interrupt arrives. For these purposes, the program generated by all the generators and their combinations described above is used as the victim program.

The second problem is the validation of the entrance to the handler and returns from it. The classic method of verification is the comparison of traces and states at the control points of the RTL and TLM models. However, it is extremely difficult to implement the submission of interruptions at the same time points from the point of view of the pipeline of models of different levels of abstraction. It would be possible to achieve tact precision TLM-model and use information about the internal state of the pipeline to generate an interrupt event, however, this approach requires too much labor. At this stage, it was decided to use a simplified comparison mechanism. The fact of the exit to the handler and the correctness of its operation is checked by a separate monitor (no coincidence of the traces is required). In turn, the traces are compared for all instructions, except those that are executed in the interrupt handler. The similarity of the main program is provided by the context recovery procedure when exiting the handler.

VI. Generator integration

To create more complex tests, the integration of generators into each other is necessary, since the interaction of different microprocessor devices, including the DSP processor, generates a huge variety of possible situations.

Using a random instruction generator, VLIW packets are created that are randomly placed in the body of subroutines, loops, exception handlers, and inserted between program jumps and memory access commands.

The combination of a generator for control flow with a generator for a memory subsystem allows creating critical situations for the microprocessor. For example, exceptions under long-term pipeline blocking conditions, such as locks associated with memory accesses, which, in turn, are divided into arbitration locks caused by conflicts when accessing memory blocks, and locks caused by delayed response from the memory (space for cache misses, as well as non-cacheable accesses to external memory).

To ensure the combinability of generators in the generated code that checks a particular subsystem, special sections are provided into which integration of the code of the stream of arithmetic commands or memory access tests of orthogonal in terms of resources used to the test program of the upper level is allowed.

VII. Conclusion

The paper analyzes the problems that arise when automating the generation of pseudo-random tests for computational cores. Ways to solve them are described with the formation of recommendations for optimizing the corresponding architectural implementations by the example of developing specialized generators, such as a flow of random instructions, control flow tests, cache hierarchy tests, and external interrupt tests. The developed generators were applied when checking the correctness of the operation of the VLIW DSP processor with the Elcore50 architecture. An approach to combining them was also proposed, which allowed to test the interaction of different processor units and identify several critical errors in the operation of the DSP core.

After a year, using the described models, a test generation system was created for the developed processor core with the VLIW architecture, solving the complex task of checking all core subsystems and their interaction by combining the already existing and newly developed specialized generators for separate subsystems of the core.

REFERENCES

[1] Vigyan Singhal, Starting Formal Right from Formal Test Plannin, Oski Technology, Verification Academy at DAC-2015. S. 1–10.

[2] David M. Russinoff. Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover. In Computer-Aided Reasoning: ACL2 Case Studies, chapter 13. Kluwer Academic Publishers. 2000. S. 1–8.

[3] Kamkin A.S., Petrochenkov M.V. Sistema podderzhki verifikacii realizacij protokolov kogerentnosti s ispol'zovaniem formal'nyh metodov (System for supporting the verification of coherence protocol implementations using formal methods) // Voprosy radioehlektroniki. 2014. T. 5. № 2. S. 5–17.

[4] Karpov YU.G. Model Checking. Verifikaciya parallel'nyh i raspredelennyh programmnyh sistem (Verification of parallel and distributed software systems) SPb.: BHV-Peterburg, 2010. 560 s.

[5] Bening, Lionel, Foster, Harry D. Principles of Verifiable RTL Design: A functional coding style supporting verification processes in Verilog Hardcover, Springer – May 31, 2001 g., S. 12–17.

[6] Greene B. and McDaniel M. The Cortex-A15 Verification Story // DVClub, Austin, december 7, 2011 g., S. 1–7.

[7] Kamkin A.S., Kocynyak A.M., Smolov S.A., Tatarnikov A.D., CHupilko M.M., Sortov A.A. Sredstva funkcionalnoj verifikacii mikroprocessorov (Tools for Functional Verification of Microprocessors) / Sb. trudov Instituta sistemnogo programmirovaniya RAN T. 26. 2014 S. 149–206.

[8] Meshkov A.N., Ryzhov M.P., SHmelev V.A. Razvitie sredstv verifikacii mikroprocessora «Elbrus-2S» (The development of the verification tools of the "Elbrus-2s" microprocessor) // Voprosy radioehlektroniki. 2014. T. 4. № 3. S. 5–17.

[9] Putrya F.M. Primenenie generatorov sluchajnyh programm i sluchajnyh fonovyh vozdejstvij pri funkcional'noj verifikacii mnogoyadernyh sistem na kristalle (Application of generators of random programs and random background influences in the functional verification of multi-core systems on a chip): materialy sed'moj mezhdunarodnoj konferencii "Avtomatizaciya proektirovaniya diskretnyh sistem". 16–17 noyabrya 2010 g., Minsk, S. 234–241.

[10] Garashchenko A.V, Gagarina L.G., Fedotova E.L., Vysochkin A.V., Zajcev V.V. Razrabotka generatora verifikacionnyh testov dlya mnogoyadernyh struktur (Development of a verification test generator for multi-core structures) // Informatizaciya i svyaz'. 2017. №4. S. 20–25.