Test data generation for arithmetic subsystem of CPUs MIPS64

Evgeni Kornikhin, CMC MSU

***Abstract*—The main problem is test data generation for arithmetic subsystem testing of CPUs MIPS64. A method is proposed to solve this problem. The method uses a formal description of processor functionality. A language for this description is proposed also. This language permits automatic test data generation.**

I. Problem

A computer system works correctly if all its components work correctly. One of these component is CPU. Testing is a well known method for checking of CPU work. This paper deals with testing of MIPS64 [1] CPU operations. Standard of this architecture describes a functionality for each CPU operation. Depending on parameters value an operations must work correctly and return value or throw an exception. For example, integer division must throw an exception when a divisor equals to zero. Both variants of behavior define a *test situation*. "Overflow caused" or "Result is ready" are examples of test situations.

Testing deals with a lot of different test situations. Standard of architecture MIPS64 contains 246 instructions. Among it 35 instructions are arithmetic instructions. There are another classes of instructions with similar description facilities. So whole part of these instructions reaches 90%. They produce a lot of test situations. Since large amount of test situations an automation of its test data generation is important problem.

So, the main problem is constructing a system (component) which generates values of arguments for specific test situation of CPU operation. Given test situation realizes while this CPU operation executes with these values of arguments.

II. Existence methods

Chip testing problem became an actual problem when chips appeared. The first methods of chip testing were pure electrotechnical methods. Instant software-based methods of chip testing appeared only when computers became more powerful.

Testing methods depend on chip representation. The first chip representation was a circuit diagram. Chip testing problem appeared in Richards Eldred's article [4] in 1959. He named this problem as ATPG (Automatic Test Pattern Generation). All ATPG-methods may be divided into 2 groups such as functional and structural. A functional methods work with a chart as with a function between input and output. It is a black-box method. A structural method is a white-box method. A structural method deals with wires of circuit diagram. The first ATPG-algorithm was "D-algorithm". "D-algorithm" was used for look for "constant errors" (element always generates constant when it should generate different values). "D-algorithm" spreads an error to the inputs and outputs and generates test data (on input) and output. Another methods use SAT-tools (SATisfiability), work with BDD (Binary Decision Diagram), etc. Chip testing problem has influenced even to chip design: there are requirements to chip design DFT (Design For Test) for more powerful its testing.

Another chip representation is RTL-model (Register Transfer Language). The first mention of RTL-models was in 1980 [5]. A RTL-model may be translated to the circuit diagram without functionality loss. Researchers from Corsica University proposed to select paths from RTL-model and generate a test data by constraint logical interpreter [3] applying to this path [6]. This method applied to the high-level programs also but great expressive capabilities of high-level programming languages prevented its effective using. Researchers from England proposed to use SAT for this problem [7]. Italian researchers proposed to use genetic algorithms [8].

And the third chip representation uses pseudocode for functionality description. Pseudocode is the most similar to programs on a high-level programming language. It describes chip functionality but not chip structure. This representation is used in this article. However this representations is used not so often. German researchers proposed to use integer linear programming to the verification problem [9]. But standard of architecture MIPS64 uses multiplication which is non-linear operations. Moreover integer linear programming methods don't guarantee its finishing for acceptable time.

III. Current results

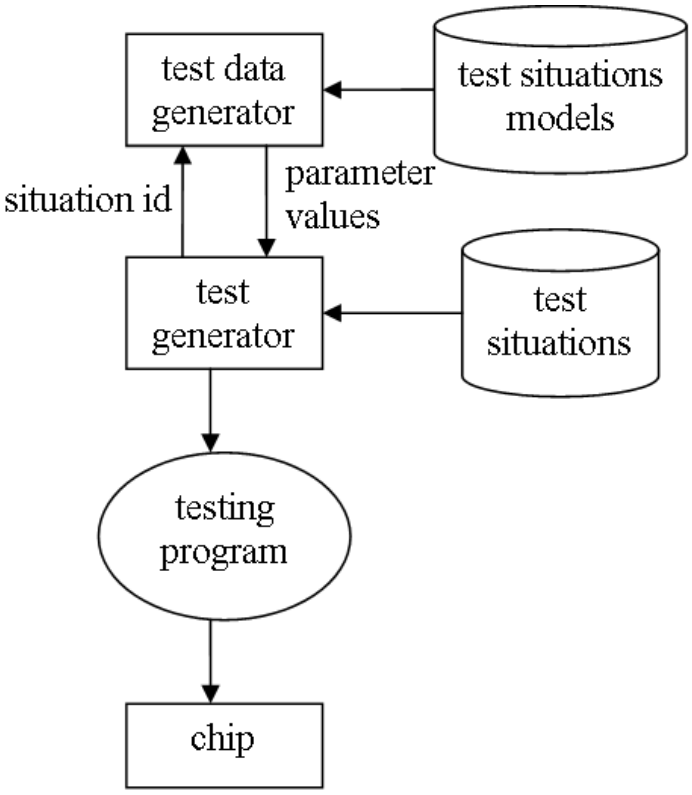

Fig. 1. Testing system structure

Fig. 1 shows a structure of test system. The central component is the test generator. It generates testing programs for executing on a chip. But the goal of this research is another component – the test data generator (its the rest is given here only for illustration). The test data generator takes a test situation identifier (name) from the test generator and gives generates values for arguments of operation which corresponds to taken test situation. The model of test situation must be constructed for each test situation manually. The test data generator is based on this model when creates a test data. The model of test data is the text on a proposed language (see III-B). This language is very similar to pseudocode used in MIPS64 standard [1] for definition of operations functionality. The sequence of actions including a model creation and its using is the proposed method of test data generation.

A. Method of test data generation

Using proposed method it is possible to construct code with test data by formal or semi-formal operation description. The method is based on constraint logical programming [3].

1) find a formal or semi-formal description of given operation behavior
2) find arguments of given operation
3) find test situations corresponding to given operation
4) determine ways of test situation obtaining
5) make up a test situation model by proposed language (see III-B)
6) run proposed test data generator (for example, by API); it creates a file with intermediate representation if it doesn't exist

The language of test situation description is very similar to pseudocode. So in the most cases the manual creating of a test situation model is come to standard rewriting.

B. Language of test situation description

A language of test situation description contains all operations of pseudocode from standard [1]:

- get bit by its index (for example, `x[7]` is 7th bit of `x`)
- get range of bits by indexes of bounds (for example, `x[8..5]` is range of bits started at 8th and finished at 5th including both bound bits)
- concatenation (for example, `x.y` is number which binary notation started with binary notation of `x` and continued with binary notation of `y`)
- bit power (itself concatenation by required times) (for example, `x^5` is bit power of `x`)
- regular arithmetic operations (addition, subtraction, multiplication)
- regular comparison operations (greater, less than)
- regular logical operations `AND` and `OR`
- assignment operator (for example, `x := 4;`)
- assertion operator (for example, `ASSERT x = 5;` is assertion about value of `x`: it must be equal to 5)

The language doesn't contain a conditional operation and loop operation because all MIPS64 test situations may be described without them.

The language doesn't contain a logical operator `NOT`. This operator conflicts with the resolution method which is base of CLP. Moreover all MIPS64 test situations may be described without `NOT`. Therefore the language contains all comparison operations, extra versions of functions (with invected `NOT`). For example, the language contains `WordValue(x)` because `NOT(NotWordValue(x))` is used in test situations.

C. Test data generator

The test data generator requires a file with test data model. The generator translates it to the intermediate representation if this representation doesn't exists. Then the generator executes the intermediate representation. And finally it analyzes results of execution and builds values of operations parameters as the output.

The intermediate representation is a logical program. It is executed by open-source constraint logical interpreter ECLiPSe [2]. Constraints is key technique of proposed method. Each operator may be translated to a set of constraints (predicates) and logical interpreter tries to find true values of variables (all constraints are true with these values of variables). Variables are represented by SSA-form [10].

The generator always finishes because the language doesn't contains a loop operator and call-mechanism (to define recursive calculations).

The generator tries to create as many as possible different sets of test data. This feature allows to get as many as possible executions of testing program on chip (and possibly to reveal hiding errors).

D. Example

Consider the method at the operation ADD. There is description of this operation in standard [1] on page 36. For the beginning let's looking for its arguments. Read: "Description : $rd \leftarrow rs + rt$". So, rt and rs are arguments of this operation. Then let's looking for test situations. Read an operation pseudocode from standard:

```
  if NotWordValue(GPR[rs]) or
NotWordValue(GPR[rt]) then
      UNPREDICTABLE
  endif
  temp ← (GPR[rs]31||GPR[rs]31..0) +
(GPR[rt]31||GPR[rt]31..0)
  if temp32 ≠ temp31 then
      SignalException(IntegerOverflow)
  else
      GPR[rd] ← sign_extend(temp31..0)
  endif
```

`SignalException` is used for denoting an exception. One of test situations is `IntegerOverflow`, i.e. overflow when summation is executed. Let's looking for a way of execution a pseudocode to call `SignalException(IntegerOverflow)`. At first calculation of "NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])" mustn't get a true value (otherwise execution of ADD is unpredictable because pseudocode contains `UNPREDICTABLE` in case of calculation "NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])" get true value). Then an assignment to $temp$ must be executed. And finally values of 32nd and 31st bits of $temp$ mustn't be equal. This is only one way to call `SignalException(IntegerOverflow)`. Let's write this way by language proposed in III-B:

```
VAR rs : 32;
VAR rt : 32;

ASSERT WordValue(rs) AND WordValue(rt);

temp :=
   rs[31].rs[31..0] + rt[31].rt[31..0];

SITUATION temp[32] # temp[31]
    IS IntegerOverflow .
```

`NOT( NotWordValue(GPR[rs]) OR NotWordValue(GPR[rt]) )` was replaced by `WordValue(GPR[rs]) AND WordValue(GPR[rt])` and using of `GPR` was removed for simplicity. So, expression is transformed to `WordValue(rs) AND WordValue(rt)`.

It is enough to run the test data generator and get test data (values of rs and rt).

IV. Future works

The article has proposed the method of systematical test data generation for testing of arithmetic CPUs MIPS64 subsystem. The language for test situtions description and necessary tools have been proposed also.

The future plans are analyzing another architecture standards. The goal is production a language applicable to the wider CPU architectures. The generator prototype will require a revision. The future plans are also wider approbation of proposed tools. And another plan is test data generation for non-arithmetic CPUs MIPS64 subsystem.

References

[1] MIPS64 Architecture For Programmers VolumeII: The MIPS64 Instruction Set. Document Number: MD00087. Revision 2.00. June 9, 2003.

[2] K.Apt, M.Wallace. Constraint Logic Programming using Eclipse. Cambridge University Press, 2007.

[3] K.Marriott, Peter.J.Stuckey. Programming with Constraints. MIT Press, 1998.

[4] Richard D. Eldred: Test Routines Based on Symbolic Logical Statements. J. ACM 6(1): 33-37 (1959)

[5] Davidson and Fraser; The Design and Application of a Retargetable Peephole Optimizer; ToPLaS v2(2) 191-202 (April 1980)

[6] C.Paoli, M.-L. Nivet, J.-F.Santucci, A.Campana. Electronic Design, Test and Applications, 2002. Proceedings. The First IEEE International Workshop on Volume , Issue, 2002 Page(s):382 - 386

[7] F.Fallah, S.Devadas, K.Keutzer. Functional Vector Generation For HDL Models Using Linear Programming and 3-Satisfiability // in Proceedings of the Design Automation Conference, pp. 528-533, June 1998.

[8] F.Corno, A.Manzone, A.Pincetti, M.Sonza Reorda, G.Squillero. Automatic Test Bench Generation for Validation of RT-level Descriptions: an Industrial Experience // DATE2000: Design, Automation and Test in Europe, Paris (F), March 2000, pp. 385-389.

[9] R.Brinkmann, R.Drechsler. RTL-datapath verification using integer linear programming // In IEEE VLSI Design'01 & Asia and South Pacific Design Automation Conference, Bangalore, pages 741–746, 2002.

[10] R.Cytron, J.Ferrante, B.K.Rosen, M. N.Wegman, F.K.Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph // ACM Transactions on Programming Languages and Systems, vol.13, no.4, ACM Press, pp.451–490, 1991.