Using ER Models for Microprocessor Functional Test Coverage Evaluation

Authors: Benso A., Di Carlo S., Prinetto P., Savino A., Scionti A.,

Published in the Proceedings of the IEEE 11th International Biennial Baltic Electronics Conference (BEC), 6-8 Oct. 2008, Tallin, EE.

DOI: 10.1109/BEC.2008.4657498

Using ER Models for Microprocessor Functional Test Coverage Evaluation

Alfredo Benso, Stefano Di Carlo, Paolo Prinetto, Alessandro Savino, Alberto Scionti
Politecnico di Torino Dipartimento di Automatica e Informatica Torino, Italy
e-mail: {alfredo.benso, stefano.dicarlo, paolo.prinetto, alessandro.savino, alberto.scionti}@polito.it

***Abstract*—Test coverage evaluation is one of the most critical issues in microprocessor software-based testing. Whenever the test is developed in the absence of a structural model of the microprocessor, the evaluation of the final test coverage may become a major issue. In this paper, we present a microprocessor modeling technique based on entity-relationship diagrams allowing the definition and the computation of custom coverage functions. The proposed model is very flexible and particularly effective when a structural model of the microprocessor is not available.** [1]

I. Introduction

The complexity of modern microprocessors makes software-based self-testing techniques very attractive w.r.t costly hardware BIST architectures [1] [2]. Software-based self-testing starts with a list of faults composed of either structural faults (e.g., stuck-at faults, delay faults, etc.), or high level functional faults, and proposes a set of software routines for their detection. Test routines are generated using different methodologies (e.g., random generation, graph based, genetic algorithms, etc.), and the fault detection mechanism relies on the use of microprocessor instructions only [3] [4], [5].

The evaluation of the test routines final fault coverage strongly depends on the type of faults the test is designed for, i.e., structural faults or functional faults. When dealing with structural faults, a structural model of the target microprocessor is usually available, and the fault coverage can be easily computed by means of fault simulation. The situation is more complex when considering functional faults. The definition of functional faults is less formal than the one of structural faults. Moreover, functional faults are usually considered when, for IP protection, the internal structure of the microprocessor is not available and only high level functional models are provided. In this context, the evaluation of the fault coverage is more complex and requires the definition of new metrics and models [6], [7].

This paper addresses the problem of evaluating the functional test coverage of software microprocessor test programs using an high-level microprocessor model more efficient in terms of complexity w.r.t. previously proposed ones [8]. In particular, the proposed methodology is based on the use of an entity-relationship (ER) diagram [9] to represent the microprocessor under test (MUT), its instruction set architecture (ISA) and the target test program. The functional coverage is then computed resorting to a set of interrogations on the database implementing the ER model.

The paper is organized as follow: Section II summarizes the basic steps required to set up the proposed test coverage evaluation methodology. Section III proposes a model for the target microprocessor and test program, whereas Section IV maps these models to an ER diagram. Section V introduces the rules to define new functional coverage metrics based on the ER diagram. Section VI applies the proposed methodology to a commercial microprocessor and finally, Section VII concludes the paper and proposes future extensions.

II. Test Coverage Evaluation Flow

The methodology proposed in this paper is organized in the following steps:

1) *Microprocessor and test program modeling*: the microprocessor under test (MUT), the instruction set architecture (ISA), and the target test program need to be described according to an high level model highlighting the main information required to define functional coverage metrics. The main requirement of the proposed model is the possibility of working even if only user level information about the MUT is available;
2) *Entity-Relationship (ER) model definition*: the model created in the previous step is translated into an entity-relationship diagram suited for the definition of functional coverage metrics;
3) *Coverage Metrics Definition*: functional coverage metrics are defined resorting to Structured Query Language (SQL) queries.

III. Microprocessor and Test Program Modeling

This section proposes the different models used to represent the MUT and the target test program. They represent the basis for the proposed test coverage estimation methodology.

A. Microprocessor Model

This subsection introduces our microprocessor model, called Units-Paths Model (UPM). It tries to represent the microprocessor at a very high level resorting to information usually available, through the microprocessor documentation, to every class of users. According to [4], the first step for the definition of an high-level model for microprocessor software testing

[1]This work has been partially funded by Regione Piemonte under grant #E22 Software-based Solutions for the dependability of digital systems in space applications

is the identification of the main microprocessor functional components (e.g., ALU, Register Files, etc.).

The UPM represents a microprocessor as a set of *Units* connected through a set of *Interconnections*, and working on a set of *Resources*.

Each Unit U identifies a *functionality* of the microprocessor not necessarily associated with an hardware block. It can be formally defined as a 3-tuple $U =< unit\_id, f\_set, r\_set >$, where $unit\_id$ univocally identifies the Unit, f_set is the set of functionalities implemented by the unit (e.g., arithmetic operations, memory transfers, logical operations, etc.), and r_set is the set of resources (e.g., registers) involved during the execution of the functionalities the Unit is designed for.

Each resource R can also be represented as a 3-tuple $R =< resource\_id, access, type >$ where $resource\_id$ is an identifier for the given resource (e.g., the register name), $access$ identifies whether the resource is readable, writable or both of them, and $type$ identifies whether the resource is accessible through a microprocessor instruction or hidden. Information about connections of resources, timing, etc. are not considered in our model since strictly related to the microprocessor structural model and not always available to the user.

The *Interconnections* are defined as a typed high-level composition of physical connections. Each interconnection needs to be modeled by specific information:

- The direction of the interconnection: a *source* (unitS) and a *destination* (unitD) Unit are identified;
- The type of the interconnection (e.g. data, address or instruction).

Each interconnection can be defined as $In =< unitS, unitD, type >$. Bidirectional interconnections are modeled as two distinct interconnections.

B. Instruction Set Architecture and Test Program Model

The microprocessor instructions represent the only instrument available for the test program to access the internal elements, i.e., Units, Resources, Interconnections (see Section III-A) of the microprocessor. The Instruction Set Architecture needs therefore to be modeled in order to have a direct connection from the software domain to the microprocessor hardware architecture.

We distinguish among Instruction Types (IT) and Instructions. An Instruction Type represents a collection of instructions performing the same type of operation (e.g., an addiction, a memory store, etc.). Each IT can be therefore represented by the operation Op it performs, the set of possible source operands SSO used as input for the computation, and the set of possible destination operands SDO used to store the results of the computation: $IT =< SSO, SDO, Op >$. SSO and SDO are taken from the set of microprocessor resources tagged as accessible by the user.

An Instruction $I =< IT, SO, DO >$ is an instance of an IT where $SO \in SSO$ and $DO \in SDO$ represent specific values of the source and destination operands.

At the programmer's level, instructions are atomic operations but, going into more detail each instruction or more in general each instruction type can be further split into a sequence of *micro-operations*. Each micro-operation can be defined as a 4-tuple $MO =< SR, DR, Ins, Op >$ where Op defines the performed operation, SR is the set of source resources and DR is the set of destination resources. SR and DR in this case include also hidden microprocessor resources. Finally, at this level Ins is the set of Interconnections activated by the micro-operation. This second level of description is not mandatory since it usually requires structural information. In the most general case each IT can be mapped to a single micro-operation.

The availability of the MO description allows for each IT to:

- Distinguish between internal and external resources: since SO and DO are a subset of SR and DR used by the MOs, we can easily distinguish between internal and external resources;
- Inherit a list of activated interconnections which are defined as the union of the interconnections used by the MOs composing the instruction.

To conclude, a given test program can be defined as a set of *test routines* where each routine is defined as a *sequence of instructions*: $TestRoutine = \{I_1, \cdots, I_i, \cdots, I_n\}$. The instructions are identified by their position (i.e., a unique sequence position number i).

IV. ER Model Definition

This section describes how the models introduced in Section III can be translated into an Entity Relationship (ER) diagram used to easily define and compute different test coverage metrics. Actually the presented models define a set of different entities (e.g., units, interconnections, instruction types, etc.) and identify the relationships existing among entities. An ER is able to capture this information in an very effective abstract model. The translation may be automated if proper languages are used for the model descriptions.

Figure 1 shows the ER schema representing all the entities involved in the microprocessor, ISA, and test program models as well as the relationships among these entities.

The ER diagram can be easily converted into a relational database filled with the information regarding the MUT and the test program. It can then be used to evaluate test coverage metrics defined by means of Structured Query Language (SQL) queries.

The resulting database will contain two set of information:

1) *Microprocessor & ISA information*: loaded only once for a given MUT. It is thus a *static* information never updated once the MUT and ISA models are correctly defined;
2) *Test program information*: information concerning the execution of the target test program. It is a *dynamic* information since it requires the actual execution of the test program and the collection of execution traces. This

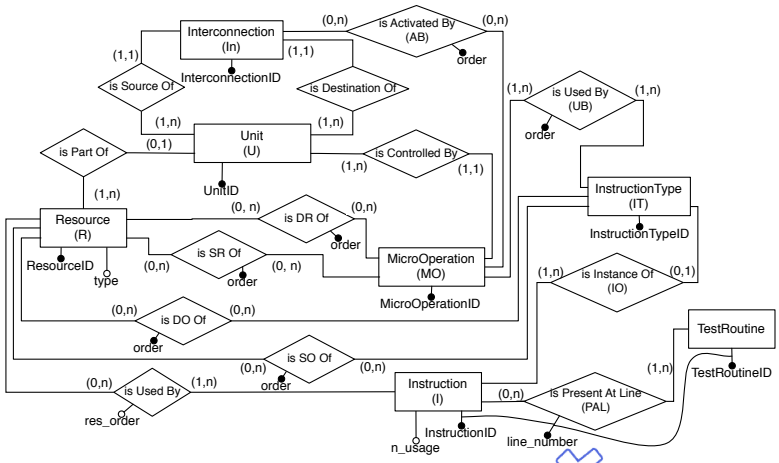

Fig. 1. Microprocessor ER Schema

TABLE I
SINGLE UNIT ACTIVATION COVERAGE METRIC

1. SELECT	COUNT(DISTINCT UnitID)
2. FROM	U, MO, I, UB
3. WHERE	I.n_usage $>$ 0
4.	AND UB.InstructionTypeID = I.InstructionTypeID
5.	AND MO.MicroOperationID = UB.MicroOperationID
6.	AND MO.UnitID = U.UnitID

set of information needs to be recreated every time the test program is modified.

V. COVERAGE FUNCTIONS DEFINITION

The ER model defined in Section IV can be used to define different types of functional test coverage metrics. Since we are working with a very high level model of both the MUT and the test program, the type of metrics we can define are not correlated to classical structural faults but in general are able to measure the amount of microprocessor elements (e.g., units, interconnections, resources, instructions, etc.) actually activated by the given test program. Nevertheless, more traditional functional tests, e.g., functional patterns, can be mapped getting rid of the necessary information in the DB.

Test coverage metrics are defined by means of SQL queries performed on the proposed ER model. In order to understand this concept, we provide two simple examples of coverage metrics. Table I proposes a coverage metric evaluating the percentage of microprocessor units activated by the target test program.

The query counts how many different units (`SELECT COUNT(DISTINCT UnitID)`) are activated by the Instructions executed in the test program. Executed instructions are selected (row 3, Table I), and the corresponding operations identified (rows 4-5, Table I). The operations are finally used to identify the units involved in their execution (row 6, Table I).

Another example of coverage function is the percentage of interconnections $In_{activated}$ activated during the test over the total set of interconnections In_{all}. Table II shows the SQL queries required to compute this coverage function.

TABLE II
INTERCONNECTIONS ACTIVATION COVERAGE METRIC

$In_{activated}$	SELECT	COUNT(DISTINCT InterconnectionID)
	FROM	In
$In_{operation}$	SELECT	UB.MicroOperationID
	FROM	I, UB
	WHERE	n_usage $>$ 0 AND I.InstructionTypeID = UB.InstructionTypeID
	GROUP BY	I.MicroOperationID
In_{all}	SELECT	COUNT(DISTINCT InterconnectionID)
	FROM	AB
	WHERE	AB.MicroOperationID IN ($In_{operation}$)
	$coverage_{interconnections} = \frac{In_{activated}}{In_{all}}$	

It is very important to highlight that the use of SQL queries allows the computation of several coverage metrics just selecting the required information from the database. Moreover, the SQL also allows to obtain, if required, a detailed list of undetected functional faults (e.g., list of not activated units) that can be used to improve the given test program.

VI. EXPERIMENTAL RESULTS

This section shows the application of the proposed methodology to evaluate the functional fault coverage of a test program designed for the functional self-test of the Motorola MPC7457 microprocessor [10]. The microprocessor has been described according to the model of Section III-A and III-B by simply resorting to the information contained in the microprocessor user manual. It is composed of the following elements:

- 25 Units:
  - 9 Execution units (e.g., Integer Unit, etc.)
  - 8 Control units (e.g., Branch Processing Unit)
  - 7 Hidden units
- 51 Interconnections:
  - 22 Data Interconnections.
  - 11 Address Interconnections.
  - 18 Instruction Interconnections.
- 132 Resources: all programmer accessible registers.

The target test program has been designed by clustering groups of units based on their functionalities according to the rules defined in [11]:

TABLE III
COVERAGE METRIC FOR INSTRUCTION USAGE

Used Instructions (I_1)	SELECT COUNT(DISTINCT InstructionID) FROM IO, I WHERE IO.InstructionID = I.InstructionID AND I.n_usage $>$ 0
Used Instruction Test Routine (I_2)	SELECT TestRoutineID, COUNT(DISTINCT IO.InstructionTypeID) FROM IO, I, PAL WHERE PAL.InstructionID = I.InstructionID AND I.InstructionID = IO.InstructionID AND I.n_usage $>$ 0 GROUP BY TestRoutineID
Total Instructions (I_3)	SELECT COUNT(InstructionTypeID) FROM IT
	$cov_{ISA} = \frac{I_1}{I_3}$
	$cov_{ISA_{test\ routine}} = \frac{I_2}{I_3}$

- Branch Prediction Unit (BPU);
- Special Purpose Register (SPR) File;
- Integer Unit (IU);
- General Purpose Register (GPR) File;
- Instruction Cache (ICache);
- Data Cache (DCache);
- Load-Store Unit (LSU);
- Pipeline.

The resulting test program is thus composed of 8 test routines (one for each testable unit) and it counts about 30,000 assembly instructions. While the entire test design process required several months, the definition of the UPM and ISA models, and the population of the database was performed in only two weeks with a limited amount of automation available.

For the proposed test program we defined three different coverage metrics:

1) cov_{ISA}: the percentage of ISA instruction types activated by the test program (see Table III);
2) $cov_{ISA_{test\ routine}}$: the percentage of ISA instruction types activated by each test routine (see Table III);
3) $cov_{Resources\ Usage}$: for each test routine the percentage of usage of microprocessor resources (see Table IV).

Table V, summarizes the results obtained by computing the three defined coverage metrics on the proposed test-bench. Column 1 reports the name of the different MUT Units, column 2 indicates the number of instructions composing the test routine of each MUT Unit, column 3 reports the $cov_{ISA_{test\ routine}}$, and finally column 4 shows the $cov_{Resources\ Usage}$.

TABLE IV
COVERAGE METRIC FOR RESOURCES USAGE

Used Resources Test Routine (I_4)	SELECT TestRoutineID, COUNT(DISTINCT UB.ResourceID) FROM I, PAL, UB WHERE I.n_usage $>$ 0 AND PAL.InstructionID = I.InstructionID AND I.InstructionID = UB.InstructionID GROUP BY TestRoutineID
All Available Resources (I_5)	SELECT COUNT(DISTINCT ResourceID) FROM R
	$cov_{ISA} = \frac{I_4}{I_5}$

TABLE V
COVERAGE METRICS EVALUATION RESULTS

Unit	Length (# instructions)	$cov_{ISA_{test\ routine}}$	$cov_{Resources\ Usage}$
BPU	3953	14%	40%
SPR File	5523	10%	45%
IU	2923	64%	46%
GPR File	4260	10%	47%
ICache	8380	13%	43%
DCache	1561	10%	45%
LSU	679	28%	44%
Pipeline	1001	9%	47%
Total	28280	85% ($coverage_{ISA}$)	

VII. CONCLUSION

This paper presented a new methodology to define and compute functional coverage metrics for microprocessor software test. It is based on the definition of an ER diagram to model both the microprocessor under test and the target test program.

The main advantage of the proposed approach is its effectiveness even when a complete structural model of the microprocessor is not available. The modeling technique was efficiently applied to the Motorola MPC7457 showing its applicability on a real microprocessor. Experimental results show the flexibility achieved in defining different coverage metrics.

Future works involve the validation of the proposed approach by analyzing the correlation of the defined coverage functions w.r.t. traditional structural fault coverage measures. We also plan to explore the mapping between our models and a functional description of the microprocessor.

REFERENCES

[1] S. Thatte and J. Abraham, "Test generation for microprocessors," *IEEE Trans. Comput.*, vol. C-29, no. 6, pp. 429–441, 1980.

[2] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 3, pp. 369–380, 2001.

[3] D. Brahme and J. Abraham, "Functional testing of microprocessors," *IEEE Trans. Comput.*, vol. C-33, no. 6, pp. 475–485, 1984.

[4] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Trans. Comput.*, vol. 54, no. 4, pp. 461–475, 2005.

[5] E. Sanchez, E. Sanchez, M. S. Reorda, G. Squillero, and M. Violante, "Automatic generation of test sets for sbst of microprocessor ip cores," in *Proc. th Symposium on Integrated Circuits and Systems Design*, M. S. Reorda, Ed., 2005, pp. 74–79.

[6] D. Moundanos, D. Moundanos, J. Abraham, and Y. Hoskote, "Abstraction techniques for validation coverage analysis and test generation," *IEEE Trans. Comput.*, vol. 47, no. 1, pp. 2–14, 1998.

[7] P. Mishra, N. Dutt, N. Krishnamurthy, and M. Ababir, "A top-down methodology for microprocessor validation," *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 122–131, 2004.

[8] D. Mathaikutty, S. Kodakara, A. Dingankar, S. Shukla, and D. Lilja, "Mmv: A metamodeling based microprocessor validation environment," *IEEE Trans. VLSI Syst.*, vol. 16, no. 4, pp. 339–352, 2008.

[9] R. W. Scamell, *Data Modeling and Database Design*. Course Technology Ptr, 2007.

[10] *MPC7450 RISC Microprocessor Family Reference Manual*, Motorola, Jan. 2005, rev. 5.

[11] A. Benso, A. Bosio, P. Prinetto, and A. Savino, "An on-line software-based self-test framework for microprocessor cores," in *Proc. International Conference on Design and Test of Integrated Systems in Nanoscale Technology DTIS 2006*, 2006, pp. 394–399.