1998 International Computer Symposium
Workshop on Computer Architecture
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Coverage Evaluation for Test Programs of X86 Compatible Microprocessors

Kuochen Wang and Sanjin Liu

Department of Computer and Information Science
National Chiao Tung University
Hsinchu, Taiwan 30050, ROC
Email: kwang@cis.nctu.edu.tw

Abstract

We propose a coverage tool for X86 compatible microprocessors that can evaluate the coverage of test programs according to a coverage model we built. Some existing coverage evaluation tools use the *design code tracing method*. This method needs the behavior model of a target design. We evaluate the coverage of test programs by analyzing test programs directly using the *test program tracing method*. Our approach just needs to know the architecture of a microprocessor, such as its instruction set, function units, pipeline stages, and data cache organization, etc. Experimental results show that using our coverage evaluation tool can use a smaller number of test programs to achieve higher coverage.

1 Introduction

The complex of VLSI designs is in close cooperation with the rapid improvement in VLSI manufacturing technologies. As a result, design verification becomes a very important task in VLSI designs. The development time of a design can be shortened by a good verification method. There are two types of verification methods: formal verification [1] and code simulation. In the formal verification, theorem proving and model checking [2] are used to verify the logical correctness of a design. However, theorem proving and model checking are not feasible for a complex design. In the code simulation method, a test suite needs to be developed. A good test suite that is efficient is very desirable in the design verification. By it, designers can find out errors of their designs quickly. Thus, how to generate a good test suite for a design is the main challenge. There are two ways to generate a test suite: automatic test program generation and manual generation. Manual generation is to manually generate those cases that are not easy to be generated by automatic test program generation. In order to evaluate a test suite, we have developed a coverage tool for X86 compatible microprocessors. By this tool, we can know the coverage of test programs according to a coverage model we built. In [3], the automatic test program generator (ATPG) generates test programs based on four levels of hierarchical information (instruction, instruction type, sequence, and sequence type). But how can we know the test programs that an ATPG generated cover which portions of a design? Therefore, a coverage tool is required to analyze the coverage of the test programs generated by an ATPG.

There are several methods to evaluate the coverage of test programs. One is to use an FSM (finite state machine) to evaluate coverage [4]. In [4], it uses an extracted control flow machine for estimation of functional coverage. The extracted control flow machine is an FSM and it is extracted from the control unit of a design. This method can easily measure the coverage of a test suite for the control unit. But this method may not be convenient for a huge design, like a microprocessor. This is because transferring a microprocessor design to an FSM is not an easy thing. Another approach to estimate functional coverage is to define a functional fault model [5] and to determine the coverage by computing the fraction of all possible such errors detected by the tests. In our approach, we analyze a test program directly for coverage measurement. We first built our coverage model according to the instruction aspect. Then, we built a coverage tool based on the coverage model.

This paper is organized in the following manner. Section 2 presents the details of the coverage model and the architecture of our coverage tool. The implementation of our coverage tool is described in section 3. Section 4 shows some experimental results. Finally, we make a few concluding remarks in section 5.

2 Design Approach

2.1 Coverage Model

Before we develop our coverage tool, we need to build a coverage model. Our coverage model, as shown in Table 1, focuses on six aspects. In the following, we present each separately.

Table 1: Coverage model.

Coverage type	Coverage item	
Single instruction	Covered instructions	ALU instructions
		FPU instructions
		MMX instructions
		System instructions
	Addressing modes	Register addressing mode
		Immediate addressing mode
		Memory addressing mode
	Operand size	8, 16, and 32 bits
Instruction combinations	2 instructions	From the same instruction group
		From different instruction groups
	n same instructions from each instruction group (n = 3, 4, 5)	
Data dependence	Data dependence degree	
	Dependent instructions coverage	
	n consecutive dependent instructions coverage (n = 3, 4, 5)	
Control transfer	Branches	Unconditional branches, conditional branches, loops, and nested loops
	Procedure calls	Nested calls, interrupts, and exception handling procedures
Data cache access	Read miss, read hit, write miss, and write hit for each cache set	
EFLAGS setting	Related flags setting for each instruction group	

2.1.1 *Single Instruction*

According to the instruction classification, we analyze each instruction in a test program. We find out each instruction belonging to which instruction type and group to determine the covered instructions. Two more coverage items are considered: addressing mode and operand size. We consider these two coverage items only in each instruction group.

2.1.2 *Instruction Combinations*

Two coverage items of instruction combinations are considered: combinations of two instructions and three up to five same instructions from each instruction group. We just consider the instruction combinations for the same instruction type. This is because instructions with different instruction types are executed in different functional units and they are independent. Then we divide the combinations of two instructions into two parts: the instructions picked up from the same instruction group and from different instruction groups. For the coverage of instruction combinations, we find out all possible combinations according to the coverage model we defined. In the other coverage items, we consider three up to five same instructions from each instruction group.

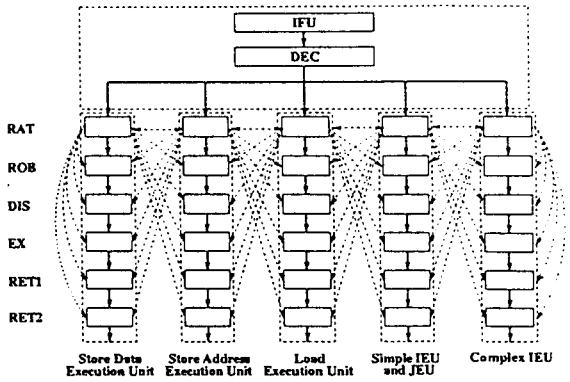

Figure 1: Possible pipeline hazards in Pentium Pro.

2.1.3 *Data Dependence*

In [8], it proposes an approach to generate a test program for pipeline interlock validation. We use the approach in our coverage tool. Consider the following two instructions:

mov eax, ebx
add ecx, eax

The first instruction move the data to register *eax* which is used as a source operand by the second instruction. This is known as a read-after-write (RAW) data dependence [9]. Each RAW pipeline hazard has three components: a producing instruction, a consuming instruction and a register that is a dependent operand. We know that all possible pipeline hazards must be exercised for verification. Figure 1 shows the possible pipeline hazards in Pentium Pro [8]. When a pipeline hazard occurs, the consuming instruction (dependent instruction) must be in the RAT (register alias table and allocator) stage, waiting for a dependent register value. The producing instruction can be in one of the other stages. The dashed arrows shown in Figure 1 means that there can be a pipeline hazard between the instructions in any two stages. Any two functional units in the microprocessor can have pipeline hazards. For example, if a producing instruction is in the DIS (dispatch) stage of the store address execution unit and a consuming instruction is in the RAT stage of the store data execution unit, there will be a pipeline hazard. According to these observations, we measure the data dependence by analyzing the pipeline hazards of a test program. We define three coverage items for data dependence. They are data dependence degree, dependent instructions coverage and consecutive dependent instructions coverage. The data dependence degree is the percentage of dependent instructions in a test program. In the second coverage item, we want to evaluate the coverage of all instructions that can be dependent instructions. In other words, we find out all instructions that may be dependent instructions, and check which instruction is a dependent instruction in a test program. In the third coverage item, we check if a test program has three up to five consecutive dependent instructions

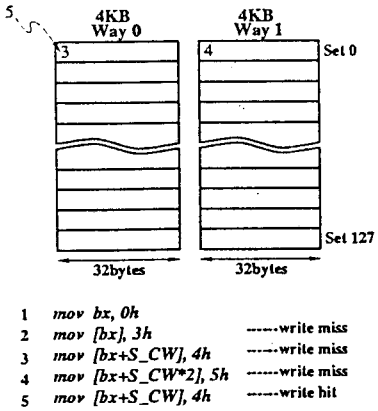

Figure 2: An example of data cache access.

for each instruction group. This item can check the coverage of the out of order execution mechanism.

2.1.4 *Control Transfer*

There are two types of control transfer: branches and procedure calls. As to branches, there are two subtypes: unconditional branches and conditional branches. Control dependence originates from the conditional branch instruction. It may cause pipeline stalls and severely affect microprocessor performance. Incorporating a branch prediction mechanism to a microprocessor design can improve the performance of the microprocessor. There are the other two coverage items of branches: loops and nested loops. In the coverage evaluation of branches, we check if all jump instructions are covered in a test program, and these jump instructions are taken or not taken. When a call instruction is taken, some stack operations should be performed to record the contents of some registers, such as the EFLAGS register. Also the return address should be stored in the EIP (instruction pointer) register. For a nested call, it may cause stack overflow. This is because the stack size assigned by a program may be too small to store all the contents of registers for a deep nested call. The other subtypes of procedure calls are interrupts and exceptions. An interrupt is an asynchronous event that is typically triggered by an I/O device [6]. An exception is a synchronous event that is generated when the microprocessor detects one or more predefined conditions while executing an instruction [6].

2.1.5 *Data Cache Access*

There are three types of data caches: direct mapped cache, n-way set-associative cache, and fully set-associative cache [9]. The test of a data cache can be done by issuing load/store instructions for data in different addresses. As to the coverage evaluation of a data cache, we compute the data cache's hit rate and miss rate of a test program. The data cache size of Pentium Pro is 8 KB and the cache is a 2-way set-associative cache [7]. Figure 2 shows an example of data cache access. We use the assembly code in Figure 2 to illustrate

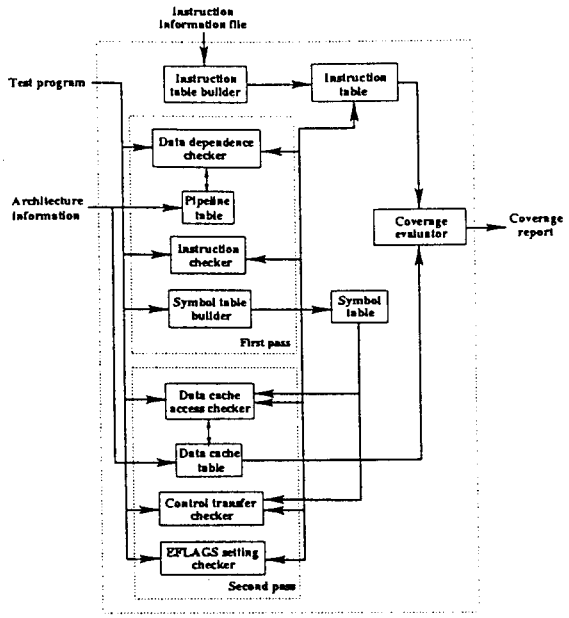

Figure 3: The architecture of our coverage tool.

how to evaluate the coverage of data cache access.

S_CW is the size of one cache way, and it is 4 KB in Pentium Pro. The instructions 2, 3, and 4 will cause cache write misses. And the cache write miss of instruction 4 will cause a cache replacement. The last instruction will cause a cache write hit. In this way, we can compute the coverage of cache misses, hits, and replacements for test programs.

2.1.6 *EFLAGS Setting*

The EFLAGS register of X86 compatible microprocessors contains eleven flags. Each instruction has it's own influence on respective flags. It will test, modify, reset or set related flags before or after an instruction is finished. Note that the flags record the status of a microprocessor. When running a program, certain instructions will test or modify some flags. If it is a branch instruction, the microprocessor will decide if the branch instruction is taken or not according to some flag, such as ZF (zero flag). That is, the EFLAGS will affect the instruction flow and most of the control unit's actions are affected by the EFLAGS. So, a full coverage of EFLAGS setting for each instruction implies a full test of most of the control unit.

2.2 The Architecture of Our Coverage Tool

Figure 3 shows the architecture of our coverage tool. There are two passes in our tool. The first pass has three main modules: *instruction checker*, *data dependence checker*, and *symbol table builder*. In the instruction checker, we compare each instruction with the instruction table created by the instruction table builder based on the instruction information file. This table records all the instruction information and

has some flags to record the coverage for each instruction. In the data dependence checker, while scanning an input program, we record the destination operand in the pipeline table if an instruction has a destination operand. We check the source operand of an instruction to see if it is in the pipeline table. If it is in the table, it means a pipeline hazard has occurred. A symbol table is built for the second pass. It records the data, labels, and procedure calls declaration in a test program. The second pass contains three main modules: *data cache access checker*, *control transfer checker*, and *EFLAGS setting checker*. In the data cache access checker, we create a data cache table to simulate the cache's contents during the execution of a test program. The next module is the control transfer checker. This module is divided into two parts, branch checker and procedure call checker. The last module, EFLAGS setting checker, is used for the checking of flags setting. Finally, after completing the test program evaluation, the coverage tool records all the coverage results and generates a coverage report for evaluation. The coverage report is useful to guide an ATPG to generate specific test programs. In this way, the ATPG can generate fewer test programs with higher coverage.

3 Implementation

Our coverage tool is implemented in Microsoft Windows using Borland C++Builder. The inputs of our coverage tool are listed as follows:

1. The architecture information of Pentium Pro.
2. Instruction information file.
3. A test program.

The overall coverage items are shown as follows:

1. The coverage of single instructions.
2. The coverage of instruction combinations.
3. The coverage of data dependence.
4. The coverage of control transfer instructions.
5. The coverage of data cache access.
6. The coverage of EFLAGS setting.
7. The overall coverage.

The definition of each coverage item has been described in section 2. The coverage tool will generate a detailed coverage report for users. We now show how we implement the coverage tool in Figure 3. The main tasks of first pass are described as follows:

1. Compare each instruction with the instruction table and set the instruction status, such as covered or uncovered, etc.

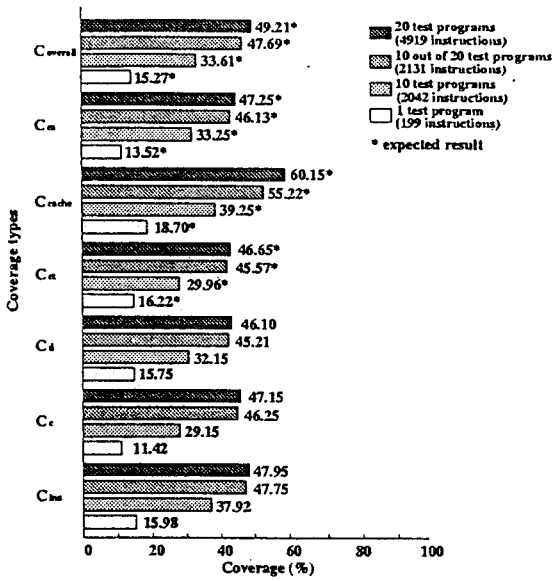

Figure 4: A coverage report using our coverage tool.

2. Check data dependence using five pipeline tables.
3. Create a symbol table for the tasks of the second pass.

The main tasks of the second pass are described as follows:

1. Simulate the execution of the test program.
2. Evaluate the coverages of branch instructions, procedure calls and EFLAGS setting.
3. Generate a coverage report.

4 Experimental Results

To evaluate our coverage tool, we use test programs generated by an ATPG in [10] as our input test programs. Now, we illustrate a coverage report, as shown in Figure 4, using our coverage tool. The abscisca is the coverage for each coverage item (ordinate). There are four situations shown in Figure 4: (1) 1 test program, (2) 10 test programs, (3) 10 test programs picked up from 20 test programs, and (4) 20 test programs. We can see the coverage of 10 test programs picked up from 20 coverage evaluated test programs is higher than that of 10 test programs without coverage evaluation first. We also compare the coverage of 20 test programs with that of 10 test programs picked up from 20 coverage evaluated test programs. We can see the coverage of the later is just slightly lower than that of the former. Although evaluating 20 test programs first will require more time, the third situation can have higher coverage with fewer instructions than the second situation.

5 Conclusions

We have developed a new and efficient approach to evaluate the coverage of test programs for X86 compatible microprocessors. Although our approach can not promise a full coverage as the FSM method, the latter is not feasible for complex microprocessor designs. Our approach does not need the RTL code or the behavior model of a design. Our coverage tool can be integrated with an ATPG to help the ATPG generate a small number of test programs with high coverage.

6 Acknowledgement

This research was supported in part by the National Science Council, ROC under Grant NSC86-2622-E-009-009.

References

[1] E. M. Clarke and R. P. Kurshan, "Computer-Aided Verification," *IEEE Spectrum*, Vol. 33, Iss. 6, pp. 61-67, Jun. 1996.

[2] A. K. Chandra, V. S. Iyengar, R. V. Jawalekar, M. P. Mullen, I. Nair, and B.K. Rosen, "Architectural Verification of Processors Using Symbolic Instruction Graphs," *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 454-459, Oct. 1994.

[3] Jiro Miyake, Gary Brown, Masahiko Ueda, and Tamotsu Nishiyama, "Automatic Test Generation for Functional Verification of Microprocessors," *Proceedings of the Third Asian Test Symposium*, pp. 292-297, Nov. 1994.

[4] Yatin V. Hoskote, Dinos Moundanos, and Jacob A. Abraham, "Automatic Extraction of the Control Flow Machine and Application to Evaluate Coverage of Verification Vectors," *Proceedings of International Conference on Computer Design*, pp. 532-537, Oct. 1995.

[5] M. Abadir, J. Ferguson, and T. Kirkland, "Logic Design Verification via Test Generation," *IEEE Transactions on CAD*, Vol. 7 No. 1, pp. 138-148, Jan. 1988.

[6] Intel Corporation, "Pentium Pro Family Developer's Manual," Vol. 2: *Programmer's Reference Manual*, 1996.

[7] Tom Shanley, "Pentium Pro Processor System Architecture," *MindShare, Inc.*, 1996.

[8] Trung A. Diep and John Paul Shen, "Systematic Validation of Pipeline Interlock for Superscalar Microarchitectures," *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pp. 100-109, Jun. 1995.

[9] J. N. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," *Morgan Kaufman Publishers*, 1996.

[10] Kuochen Wang and Leih-Ming Wu, "Automatic Test Program Generator for X86 Compatible Microprocessor Verification," *Proceedings of 1998 International Conference on Computer Systems Technology for Industrial Applications - Chip Technology*, pp. 100-105, Apr. 1998.