A BNF-Based Automatic Test Program Generator for Compatible Microprocessor Verification

LIEH-MING WU, KUOCHEN WANG, and CHUANG-YI CHIU
National Chiao Tung University

A novel Backus-Naur-form- (BNF-) based method to automatically generate test programs from simple to complex ones for advanced microprocessors is presented in this paper. We use X86 architecture to illustrate our design method. Our method is equally applicable to other processor architectures by redefining BNF production rules. Design issues for an *automatic program generator* (APG) are first outlined. We have resolved the design issues and implemented the APG by a *top-down recursive descent parsing* method which was originated from compiler design. Our APG can produce not only random test programs but also a sequence of instructions for a specific module to be tested by specifying a user menu-driven file. In addition, test programs generated by our APG have the features of no infinite loop, not entering illegal states, controllable data dependency, flexible program size, and data cache testable. Our method has been shown to be efficient and feasible for the development of an APG compared with other approaches. We have also developed a *coverage tool* to integrate with the APG. Experimental evaluation of the generated test programs indicates that our APG, with the guidance of the coverage tool, only needs to generate a small number of test programs to sustain high coverage.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Verification*

General Terms: Verification

Additional Key Words and Phrases: Advanced microprocessor, automatic program generator, BNF, compatibility verification, coverage, top-down recursive descent parsing method

1. INTRODUCTION

With rapid improvement of hardware manufacturing technologies and the help of computer-aided design (CAD) tools, superscalar microprocessors become more and more powerful and faster. Although design time can be shortened in a modern design environment, the verification effort grows exponentially as microprocessors become more complicated. That is, the total development

The work was supported in part by the National Science Council, Taiwan, ROC, under Contract No. NSC 88-2622-E-004.
Authors' addresses: L.-M. Wu and C. Y. Chiu, Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, 300, ROC; email: {wulm,takaku}@pds.cis.nctu.edu.tw; K. Wang, Department of Computer Information and Information Science, National Chiao Tung University, Hsinchy, Taiwan, 300, ROC; email: kwang@cis.nctu.edu.tw.

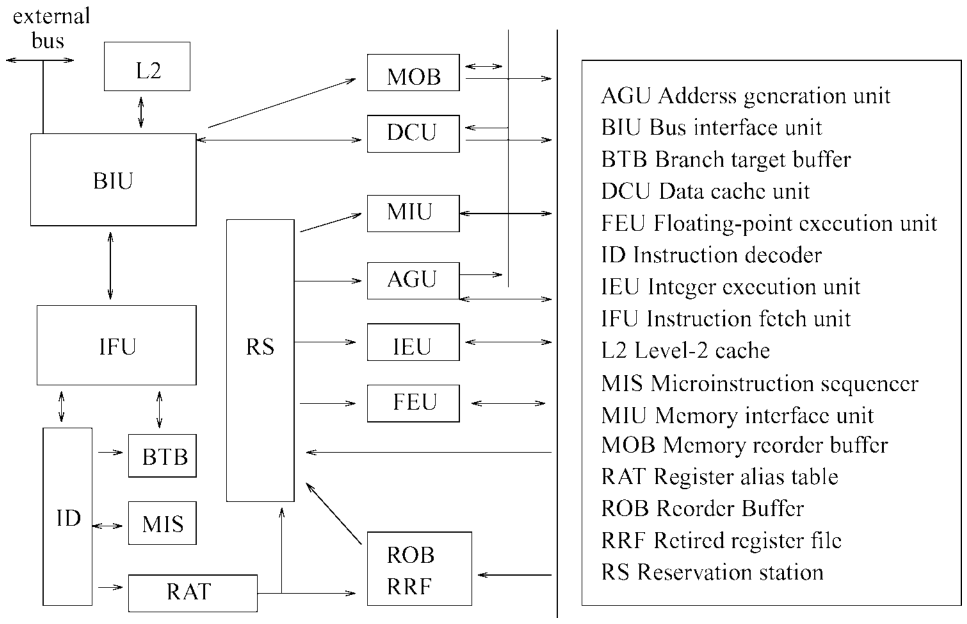


Fig. 1. Functional blocks of a Pentium Pro microprocessor.

(design and verification) cycle is still not shortened as quickly as the improvement of the design environment. In addition, an X86-compatible microprocessor needs to meet the following important characteristic: being able to run all operating systems (OSs) and applications that an Intel microprocessor can run. This makes the design and verification of an X86-compatible microprocessor more difficult. Thus compatibility verification becomes a key issue in the development of an X86-compatible microprocessor. Under time-to-market pressure, we must have a proper verification methodology for X86-compatible microprocessor development flow. Therefore, our objective is designing an APG (automatic program generator) to shorten the verification cycle.

In this paper, we use Pentium Pro as our experimental target to illustrate our design method. Our method is equally applicable to other non-X86 processor architectures. Pentium Pro is a superscalar microprocessor and is thus more complex than previous versions of X86 microprocessors [Intel Corporation 1996b]. It can decode three X86 instructions and executes five micro operations per cycle and has a long pipeline that allows high clock speed. With the help of accurate branch prediction and the outfit of ROB (reorder buffer), MOB (memory order buffer), and RS (reservation stations), Pentium Pro can execute instructions out of order. Figure 1 shows the functional blocks of Pentium Pro [Intel Corporation 1996b].

Verification of such a huge design is very costly and time-consuming. There are many test programs to be written [Thatte and Abraham 1980; Brahame and Abraham 1984]. However, it is very time-consuming to write all test programs manually. This brings about the necessity of developing an APG to speed up the verification work [Klug 1988; Savir and Bardell 1984; Al-Arian and Nordenso

1989; Thatte and Abraham 1980; Brahame and Abraham 1984]. In this paper, we present a novel Backus-Naur-form- (BNF-) based method to automatically generate test programs for X86 microprocessors. The instructions in the test programs generated by the APG must be carefully arranged to prevent the programs from reaching an illegal state [Miyake and Brown 1992, 1994]. That is, the generated test programs should be meaningful and useful for verifying a microprocessor design. Design issues for developing an APG are described first. We resolve the design issues and realize our APG by a *top-down recursive descent parsing* method [Fischer and LeBlance 1988]. The APG can produce not only random test programs but also instruction sequence-specified and module-specified test programs. Note that besides generating tests for each specified module, our APG can also generate tests to verify the interconnects between modules by specifying a user menu-driven file that involves multiple modules. We have also implemented an integrated APG & coverage tool to experiment and evaluate the quality of the APG.

The organization of the paper is as follows: Section 2 introduces the basic concept of our method. The top-level view of the pattern generation is illustrated in Section 3. Existing approaches are reviewed in Section 4. In Section 5, we discuss APG design issues and our design method. The details to implement the APG are presented in Section 6. We demonstrate an integrated debugging environment for verification in Section 7. Then, an integrated APG & coverage tool and some experimental results are presented in Section 8. Finally, Section 9 gives concluding remarks.

2. BASIC CONCEPT

We first overview the top-down recursive descent parsing method in compiler design. A BNF is simply a set of rewriting productions [Fischer and LeBlance 1988; Holub 1990]. A production is of the form $A \rightarrow B\ C\ D \cdots Z$. A is the left-hand side (LHS) of the production; $B\ C\ D \cdots Z$ is the right-hand side (RHS) of the production. A production is a rule that any occurrence of its LHS symbol can be replaced by the symbols on its RHS [Fischer and LeBlance 1988; Holub 1990]. Thus the production

$$\langle\text{program}\rangle \rightarrow \textbf{.model small}\ \langle\text{small block}\rangle$$

states that a program is required to be a small block started with a **.model small**. Two classes of symbols may be used in a BNF: *nonterminals* and *terminals*. Nonterminals are often delimited by <and>. All nonterminals must be replaced by a production with terminals. In addition, terminals are never changed. *Recursive descent parsing* is a parsing technique used in compilers. The basic concept of recursive descent parsing is that each nonterminal has a parsing procedure, which can be recursively called, that can recognize a token sequence [Fischer and LeBlance 1988; Holub 1990]. In a parsing procedure, nonterminals and terminals can be matched. Although non-context-free language constructs may be more powerful, we restrain the grammar to be context-free for parsing efficiency. Since the APG needs to generate numerous test programs, we opt for parsing efficiency. Those language constructs that cannot be generated by the context-free grammar will be expressed as macros.

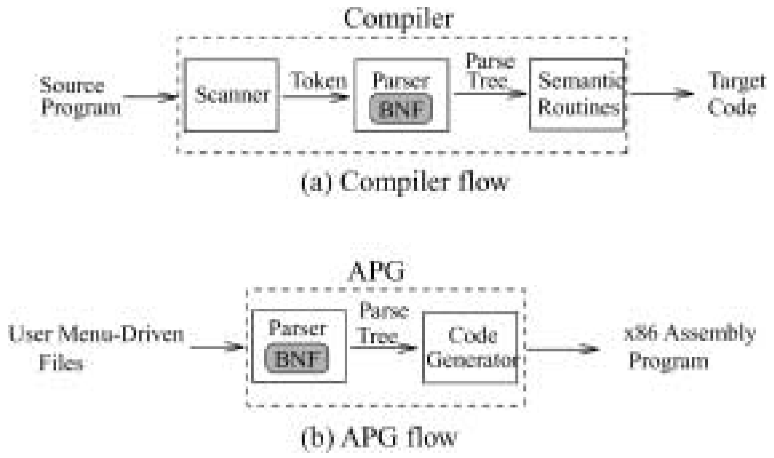


Fig. 2. Basic concept of our method.

The overall purpose of a set of productions (a BNF) is to specify what sequences of terminals (tokens) are legal. So, we can construct useful and meaningful test programs in a top-down manner by defining a BNF. The path traversed by a recursive descent parsing procedure will build up a parse tree. The selection of special parse trees will construct special test programs. That is, the features of generated test programs can be controlled. Figure 2 shows the basic concept of our method. Figure 2(a) means that we can build up a parse tree for a legal source program. Figure 2(b) means that a legal assembly program can be generated by an APG using a BNF. We can construct a test program with specific characteristics by selecting a particular parse tree.

3. TOP-LEVEL VIEW OF THE PATTERN GENERATION

Figure 3 shows the top-level view of the pattern (test program) generation. Users can control the APG by a user menu-driven file. This file will be first parsed by the APG preprocessor. The APG preprocessor will then generate some patterns to initialize the values of operands (e.g., registers). These initial patterns will be used for generating specific exceptions. The APG preprocessor will also decide some important parameters, such as the degree of data dependency and the program size based on the user menu-driven file. Besides doing those things, the APG preprocessor will generate *direction vectors* for the pattern generator with BNF kernel to generate the test programs we need. The pattern generator with BNF kernel is the main part of our APG. In the pattern generator with BNF kernel, the BNF production rules are used to generate valid test programs.

4. EXISTING APPROACHES

Three representative automatic program generators are first reviewed: the multiprocessor test generator (MPTG) [O'Krafka et al. 1995], the model-based test generator (MBTG) [Lichtenstein et al. 1994; Aharon et al. 1995], and the

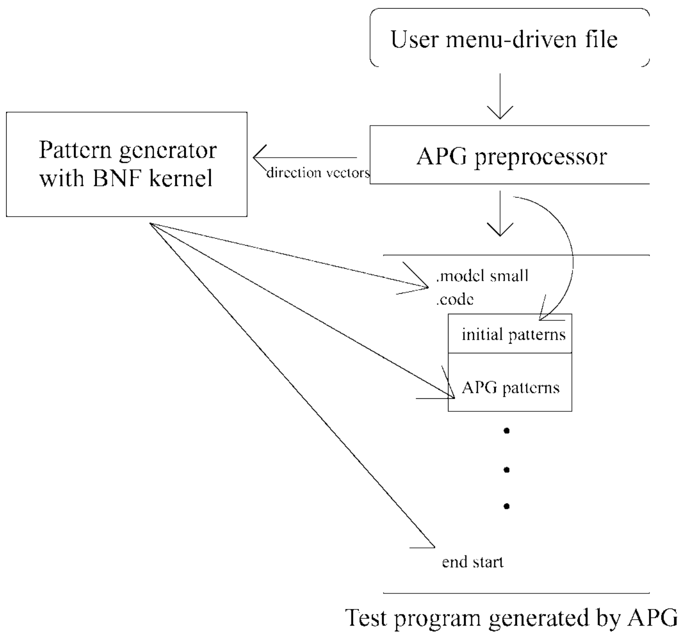


Fig. 3. Top-level view of the pattern generation.

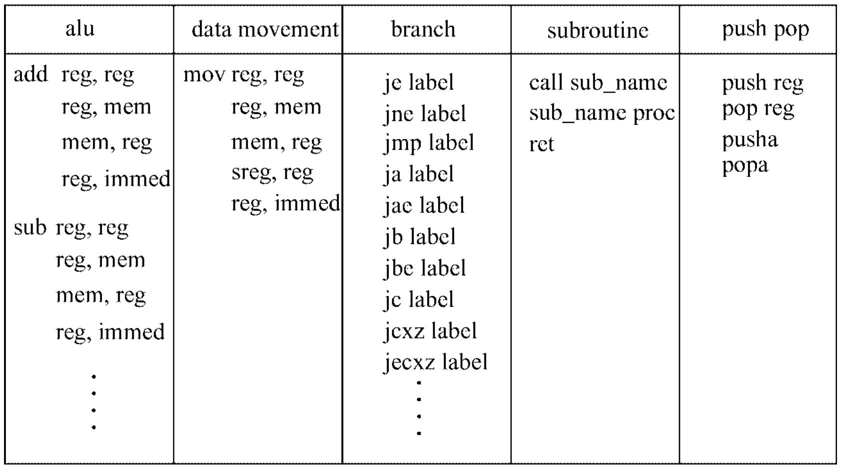
Architecture Verification Programs Generator (AVPGEN) [Chandra et al. 1995]. MPTG is a deterministic test generator [O'Krafka et al. 1995]. It generates sets of test cases that are guaranteed to cause specific events to happen. MPTG does this by employing the cache coherence protocol as its abstract machine model. This enables test specifications to directly control the occurrence of very specific sequences of cache events. MPTG is very powerful in cache verification; however, its verification scope is too limited. It needs to be coupled with a random (generally pseudorandom) test generator, such as MBTG, for exhaustive verification for most of the simulation cycles [Malik et al. 1997].

MBTG uses expert-system techniques to develop a test program generator for design verification of hardware processors [Lichtenstein et al. 1994]. The test program generator models an instruction semantically rather than syntactically. An instruction is modeled by a tree with semantic procedures at the root, operands and suboperands as internal nodes, and length, address, and data expressions as leaves. Traversing the instruction tree in a depth first order will generate instructions for test. Although MBTG is an excellent test program generator for resolving three essential difficulties in the generation process [Lichtenstein et al. 1994]—complexity, changeability, and invisibility—its ability to handle a specific instruction sequence is too weak. It was aimed toward verifying individual instructions [Malik et al. 1997]. Many exceptions such as stack overflow and page faults can only be created by executing a specific instruction sequence. And if we want to generate a specific instruction sequence to verify some functional blocks, such as a cache in a design, it is not easy to do so with MBTG. In addition, MBTG needs to maintain a large heuristic database [Lichtenstein et al. 1994]. AVPGEN [Chandra et al. 1995] uses concepts like symbolic execution, constraint solving, and biasing techniques to generate tests. A complex language called *SIGL* is provided for the user to control the generation of test programs. AVPGEN provides a very detailed way for the user to control the test pattern generation that causes it to be hard to maintain, which will be described in Section 5.7.

Next, we review the following papers, which can provide an up-to-date view of test program generation. In Aharon et al. [1991], an RTPG (random test program generator) was built, which was used from the early stages of the design until its successful completion. It provided a biasing technique to create a subset that provides high confidence in the correctness of the design. In our approach, we use the user menu-driven file and APG preprocessor to achieve the same purpose. Bin et al. [2002] dealt with the generation of test programs as a constraint satisfaction problem (CSP) and developed techniques for the problem. In contrast, our constraint-solving system for the problem is the user menu-driven file and APG preprocessor. A reloading technique was presented in Bin et al. [2002]. The drawback of the reloading techniques is that it introduces an interference into the test [Adir et al. 2001]. In order to avoid fixed code patterns, an idea of distancing the reloading instruction from the instruction that uses the resource value was shown. In our approach, we can apply appropriate BNF production rules and techniques like loop-exit code and resource locking to prevent the generated test patterns from entering illegal states. In Fournier et al. [1999] presented a methodology that relies on a verification plan. The verification plan induces sets of tests that carry out the verification tasks. Our methodology relies on appropriate BNF production rules. We can use the user menu-driven file to control the process of test pattern generation. Adir and Shurek [2002] discussed collisions for multiprocessor verification. Collisions occur when different processes access a shared resource. How the results of such collisions can be presented in test programs was described. Our APG focuses on uniprocessor verification. In Adir et al. [2002], a technique that defines unexpected events together with their alternative program specifications was proposed. When an event is detected, its corresponding alternative specification is added into the test program. Again, we use appropriate BNF production rules and techniques like loop-exit code and resource locking to prevent the generated test patterns from entering illegal states. Emek et al. [2002] presented an MBTG, *X-Gen*, targeted at systems and System-on-Chip (SoC). X-Gen provides a framework and a set of building blocks for system-level test case generation. The comparison between our BNF-based APG and the MBTG will be discussed in Section 5.7.

5. DESIGN ISSUES AND DESIGN METHOD

Our design method addresses the following issues: user-controlled APG, branch handling, test program for data dependency, instructions appearing together requirement, bounded program size, and test for data cache. At the end of this section, we will compare our APG with MBTG.



alu	data movement	branch	subroutine	push pop
add reg, reg	mov reg, reg	je label	call sub_name	push reg
reg, mem	reg, mem	jne label	sub_name proc	pop reg
mem, reg	mem, reg	jmp label	ret	pusha
reg, immed	sreg, reg	ja label		popa
sub reg, reg	reg, immed	jae label		
reg, mem		jb label		
mem, reg		jbe label		
reg, immed		jc label		
⋮		jcxz label		
		jecxz label		
		⋮		

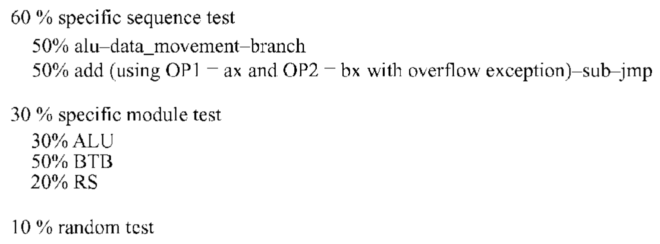
Fig. 4. Instruction classification into types.

5.1 User-Controlled APG

The types of test programs generated by an APG can be specified by users. In order to verify a design more comprehensively, three types of test programs can be defined. They are random, module-specified, and instruction sequence-specified test programs. Randomness is an effective way to generate test programs for complicated situations where even experienced functional designers can hardly figure out the details [Miyake and Brown 1994].

5.1.1 *Random Test Programs.* Random test programs, if carefully designed, can verify all combinations of instructions [Johnes et al. 1991; Hu et al. 1994; Turumella et al. 1995]. But they tend to get into illegal states (e.g., *jump* to an undefined location, too many *push* without *pop*, etc.). Besides, extremely complicated test cases, which we really want to test, occur at rare situations. Therefore, the frequency of specified test cases hit by fully random test programs is very low. Since the simulation speed of a functional model written in a hardware description language is very slow, a fully random APG may be timing-consuming for functional verification.

5.1.2 *Instruction Sequence-Specified Test Programs.* Test programs with specific sequences of instructions are needed to test a specific module. Because it is not easy to generate a specific sequence of instructions randomly, an APG should be capable of generating a specific sequence of instructions based on a user menu-driven file [Miyake and Brown 1994]. Any legal combinations of instructions should be generated by an APG. Two levels of hierarchical information (instruction type, instruction) are shown in Figure 4. We classify the instructions of X86 assembly language into five types. They are *ALU*, *data movement*, *branch*, *subroutine*, and *push/pop*. Each instruction type includes some instructions. In the figure, information about operand modes are also added to each instruction. When generating test programs, users can specify a user menu-driven file to control what sequence of instructions can be generated



```
60 % specific sequence test
   50% alu–data_movement–branch
   50% add (using OP1 = ax and OP2 = bx with overflow exception)–sub–jmp

30 % specific module test
   30% ALU
   50% BTB
   20% RS

10 % random test
```

Fig. 5. Example of a user menu-driven file.

by an APG. Figure 5 shows an example user menu-driven file. A sequence may contain instruction types (e.g., *alu*) and particular instructions (e.g., *mov*). In this menu-driven file, we can see that 60% of instructions in a generated test program are for a specific sequence test. Among them, 50% of all sequence tests are alu-data_movement-branch sequence tests, and the rest are add-sub-jmp sequence tests. We can also see that when *add* instruction is chosen, operand 1 (OP1) will be *ax* and operand 2 (OP2) will be *bx*. By carefully setting the initial values of the operands, the execution of this *add* instruction will cause an overflow exception.

5.1.3 *Module-Specified Test Programs.* Module-specified test programs can be used to test a specific module design. In an X86-compatible microprocessor case, the test flow may be IFU, ID, RS, ALU (including IEU and FEU), and ROB. Before we test ALU, we must make sure IFU, ID, and RS are working correctly. An APG should have an option of generating test programs for a specific module of a design. Figure 5 shows that 30% of instructions in a test program are for specific module tests. Among them, 30% of specific module tests are for ALU tests, 50% for BTB tests, and 20% for RS tests. Another 10% of test programs (excluding the specific sequence test and the specific module test) are randomly generated by the APG.

From a functional test point of view, we divide the instructions into five types. These types may not fully match physical modules. Thus, there must be a mapping from instruction types to physical modules, as shown in Figure 6. For example, if we want to test BTB, we should increase the percentage of branch instructions. After the APG reads the specific module test part of a user menu-driven file (like Figure 5), it maps physical modules to instruction types (like Figure 6) and then chooses the desired instructions.

5.2 Branch Handling

Two strategies, *loop-exit code* and *resource locking*, are used to resolve the problem of infinite loop generation. The former was proposed in [Miyake and Brown 1994]. We propose the latter to overcome the limitations of the former.

5.2.1 *Loop-Exit Code.* Infinite loop generation may occur if the instruction randomly selected by an APG is a backward branch and the jump condition is not handled carefully. This is because randomly generated instructions are difficult to use to control a microprocessor state, such as condition codes for

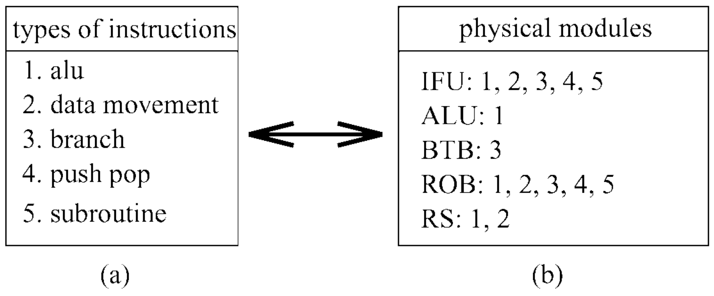


Fig. 6. Relationship between physical modules and APG-generated instruction types.

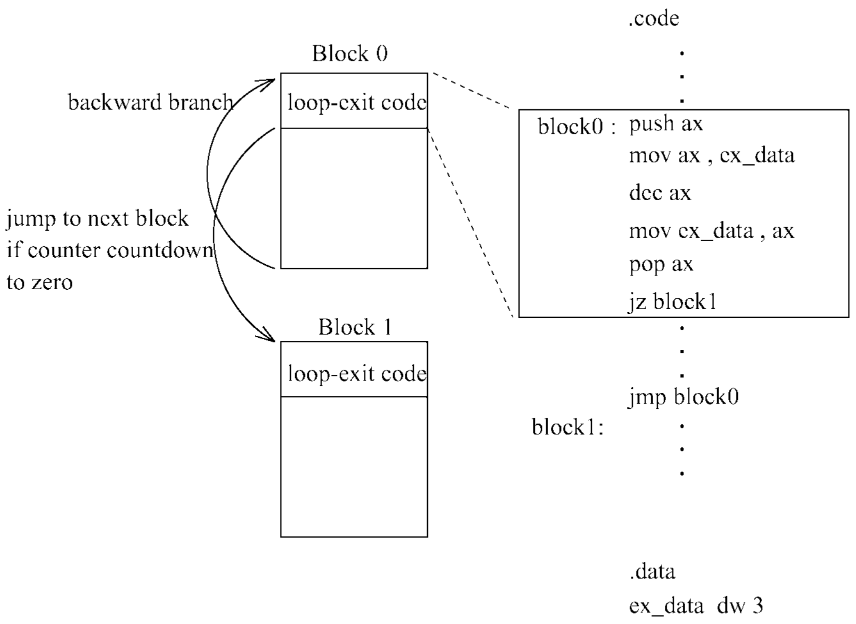


Fig. 7. Concept of loop-exit code.

branch instructions. In order to avoid infinite loops, a loop-exit code is used Miyake and Brown [1994]. Figure 7 shows the concept of the loop-exit code and an example code. A loop-exit code is generated whenever a backward branch is selected. This loop-exit code has a loop counter and jumps to the next block when the counter reaches zero. The target of the backward branch instruction is always a loop-exit code, so the test program can exit the loop. In this example code, the loop-exit code will jump to *block*1 if *ex_data* countsdown to zero.

5.2.2 *Resource Locking.* Although the generation of infinite loops is prevented by a predefined loop-exit code, it also introduces limitations on testing the whole family of loops. Therefore, we present a new approach to generate the whole family of loops without limitations. First, we deduce under what conditions infinite loops may exist. Only conditional jump instructions are illustrated here. If a backward jump instruction is always taken because the condition to

```
1  LABEL1:
2      MOV   BX, 11
3      MOV   DI, 8
4      SETNS BYTE PTR VAR[BX][DI]
5      MOV   ESI, offset VAR
6      MOV   DSI, offset VAR
7      ADD   ESI, 0ACH
8      ADD   EDI, 09EH
9      SCASW
10     MOV   BX, OFOH
11     SIDT  WORD PTR VAR[BX]
12     MOV   BP, 115
13     MOV   DI, 96
14     XOR   AX, BX
15     JNE   LABEL1
```

(a)

```
1  LABEL1:
2      MOV   CX, 11
3      MOV   DI, 8
4      SETNS BYTE PTR VAR[CX][DI]
5      MOV   ESI, offset VAR
6      MOV   DSI, offset VAR
7      ADD   ESI, 0ACH
8      ADD   EDI, 09EH
9      SCASW
10     MOV   CX, OFOH
11     SIDT  10WORD PTR VAR[CX]
12     MOV   BP, 115
13     MOV   DI, 96
14     DEC   AX
15     XOR   AX, BX
16     JNE   LABEL1
```

(b)

Fig. 8. An example of resource locking.

jump is always true or always false, we may have an infinite loop. That is, the combination of the following two elements may result in an infinite loop: a backward jump instruction and the condition that makes the jump taken. Verification is incomplete without testing backward jumps. The basic idea behind eliminating infinite loops is to prevent the loop condition from being always false or always true, and this is the basis of *resource locking*.

Figure 8 shows an example of resource locking. In Figure 8(a), the conditional jump instruction at line 15 is a backward jump. If EFLAG *ZF* is zero when the jump instruction is executed, this conditional jump is taken. To prevent *ZF* being always zero in this block, we must know which instruction is the last instruction that may change *ZF* to 1 and is executed before that backward jump. In Figure 8(a), it is *XOR* at line 14. Because we want to force the result of *XOR* changes every iteration, two things must be done: locking the operands of *XOR* and changing the values of these operands. If we didn't lock the operands of *XOR*, there might be the chance that the value of *XOR* would be increased by 4, while another instruction would decrease it by 4 in the same iteration, and thus an infinite loop would occur. Figure 8(b) shows a revised test program. In Figure 8(b), *BX* is replaced by *CX* and instruction *DEC AX* at line 14 is inserted for changing the result of *XOR* at line 15. Instructions for storing some registers temporarily are also inserted at the proper positions when there is no available register for replacement. In the following, we show how to prevent infinite loops step by step. To make it clear, we also use Figure 8 as an example.

(1) Check if the current generated instruction is a jump instruction. If it is not, it will not have an infinite loop. If it is, check if it is a backward jump instruction. If so, we may have an infinite loop in the test program. In Figure 8(a), the instruction at line 15 is a backward jump instruction that we want to find.
(2) Find out under what condition the jump instruction will be taken. In Figure 8(a), the jump instruction is taken when EFLAG *ZF* is zero.

(3) Find out which instruction is the last instruction and is executed before the jump instruction and may change the value of *ZF*. In Figure 8(a), this instruction is *XOR* at line 14.

(4) Find out which instruction is the target of the backward jump instruction. In Figure 8(a), it is *LABEL*1: at line 1.

(5) Lock the operands *AX* and *BX*, of *XOR*, at line 14. This means that all operands of instructions between *LABEL*1 at line 1 and *XOR* at line 14 cannot use *AX* and *BX* as their operands. Therefore, in Figure 8(a), *BX*s at lines 2, 4, 10, and 11 are replaced by *CX*s, which have the same size as *BX*s. We can lock the resources we want because all the resources are under our control.

(6) If there is no register with the same size available, some restoration operations are needed to free a register. In Figure 8(a), there is no need for such an operation.

(7) Insert an instruction that can change the result of *XOR* in an iteration. In Figure 8(a), *DEC* is inserted at line 14.

By these steps, a test program with no infinite loop is generated in Figure 8(b).

5.3 Test Program for Data Dependency

Data dependency may seriously affect the pipeline behavior. Many microprocessor errors occur in the pipeline design. So it is necessary to set some conditions to construct a test program with a different degree of data dependency. There are three types of data dependency. They are *read after write* (*raw*), *write after read* (*war*), and *write after write* (*waw*) [Hennessy and Patterson 1996; Hwang 1993]. The way to control the degree of data dependency is to restrict the number of available operands. The fewer the number of available operands, the more data dependency there will be in a test program. Figure 9(a) shows the three types of data dependency. In each instruction, the operator uses *operand*1 and *operand*2 as operands and stores the result back to *operand*1. For example, "*add ax, bx*" is the addition of registers, *ax* and *bx*, and the sum is stored back to *ax*. In Figure 9(a), there is a *raw* data dependency relation between operands $r1$ and $r4$, if $r1$ is the same as $r4$. Also, if $r1$ is identical to $r3$, *waw* will occur. Because $r1$ must be read when executing *op*1, *war* data dependency will occur as well. The *war* relation between $r2$ and $r4$ exists if $r2$ and $r4$ are the same. We conclude that there are three situations of data dependency in two instructions ($r1 = r4, r1 = r3, r2 = r4$), and they are represented by three arrows, as shown in Figure 9(a). Note that read-read appearances are not considered as a data dependency case.

We are going to derive an expression to estimate the *degree of data dependency* ($D(n, k)$), where n is the number of instructions and k is the number of operands. It means the occurrence probability of data dependency. The combinatorics is to express the *relative* degree of data dependency in terms of number of instructions and number of operands. Without loss of generality, we make some assumptions. First, we assume the operands of instructions are a two-column format, as shown in Figure 9(a). Second, we estimate the degree of data

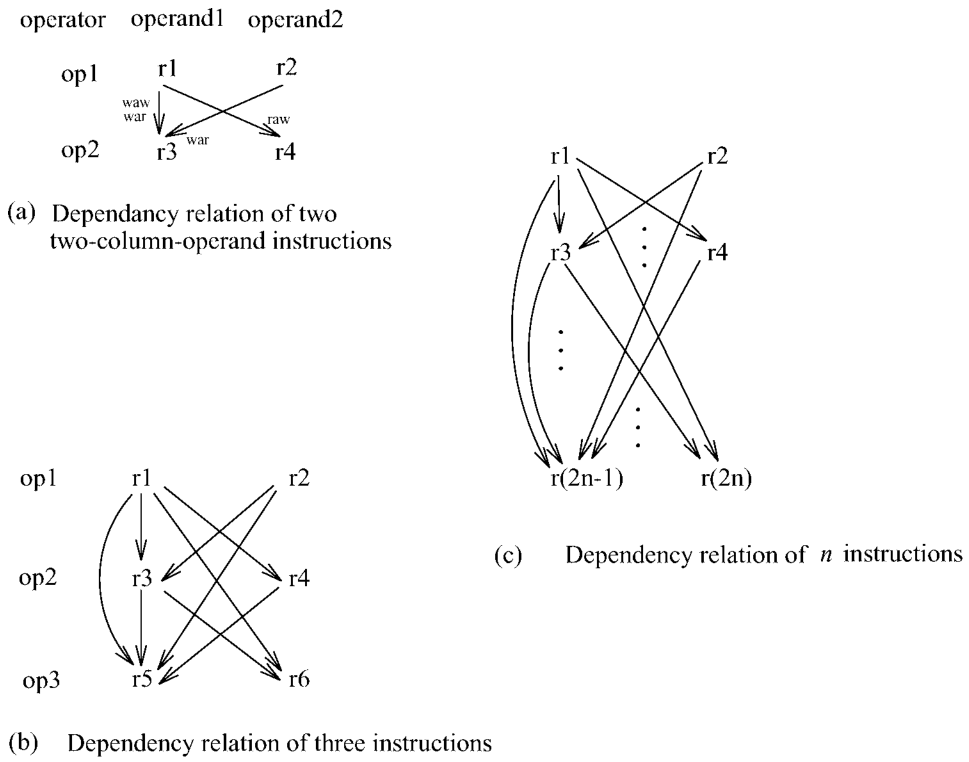

operator
operand1
operand2
op1
r1
r2
waw
war
war
raw
op2
r3
r4
(a) Dependancy relation of two two-column-operand instructions
op1
r1
r2
op2
r3
r4
op3
r5
r6
(b) Dependency relation of three instructions
r1
r2
r3
r4
r(2n-1)
r(2n)
(c) Dependency relation of n instructions

Fig. 9. Data dependency graphs.

dependency in a *basic block*. A basic block is a linear sequence of instructions that contains no branch except at the very end [Fischer and LeBlance 1988]. No branch in the middle of a basic block is permitted. Every programs can be represented as a series of basic blocks, linked together by branch instructions [Fischer and LeBlance 1988]. Four requirements for $D(n, k)$ are shown, as follows:

(1) $D(n_1, k) < D(n_2, k)$ $\quad$ *if* $n_1 < n_2$;
(2) $D(n, k_1) > D(n, k_2)$ $\quad$ *if* $k_1 < k_2$;
(3) $D(n, k) = 1$ $\quad$ *if* $k = 1$ *and* $n \rightarrow \infty$;
(4) $D(n, k) = 0$ $\quad$ *if* $k \rightarrow \infty$.

Figure 9(b) shows the data dependency relation of three instructions. There are nine arrows in the dependency graph of three instructions ($(\binom{3}{2}) + 3\times(3-1) = 9$). The equation means that there are $\binom{3}{2}$ arrows between $r1$, $r3$, and $r5$ and $3 \times (3 - 1)$ for the rest. Figure 9(c) shows the data dependency relation of n instructions. The number of arrows in n instructions is

$$\binom{n}{2} + n \times (n - 1) = \frac{3}{2} \times (n) \times (n - 1). \tag{1}$$

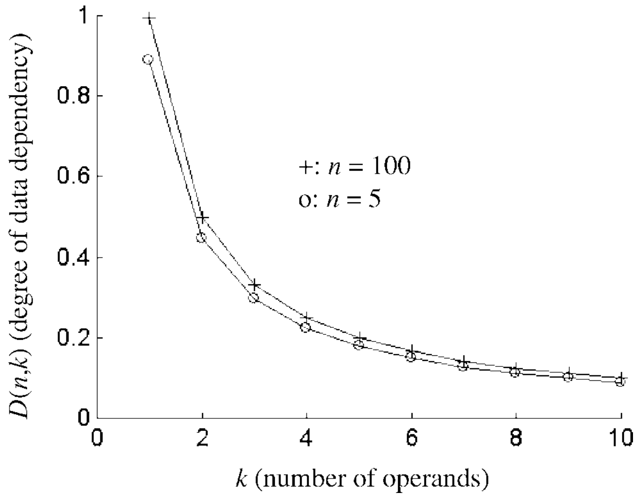


Fig. 10. Degree of data dependency for $n = 5$ and $n = 100$.

If there are k operands, the average number of arrows in n instructions is

$$\frac{\frac{3}{2} \times (n) \times (n-1)}{k}. \tag{2}$$

The total number of possible arrows considering data dependency and no data dependency is

$$\binom{2n}{2}. \tag{3}$$

Thus, we can define the degree of data dependency, $D(n, k)$, as

$$D(n,k) = \frac{4}{3} \times \frac{\frac{\frac{3}{2} \times (n) \times (n-1)}{k}}{\binom{2n}{2}} \tag{4}$$

$$= \frac{2}{k} \times \frac{(n-1)}{(2n-1)}. \tag{5}$$

$\frac{4}{3}$ in Equation (4) is a scale factor and is used to adjust $D(n, k)$ to 1 as $k = 1$ and $n \to \infty$. Figure 10 shows the data dependency for $n = 5$ and $n = 100$, respectively. In real cases, the operands can be chosen from general registers, segment registers, and memory locations. In our experimental results, if $D(n, k)$ is larger than 0.5, the generated test program has heavy data dependency. Decreasing the number of operands will increase $D(n, k)$.

5.4 Instructions Appearing Together Requirement

Some instructions always appear together. For example, *push/pop*, *call/ret/ subroutine definition*, *jump to label/label definition*, *memory access/memory*

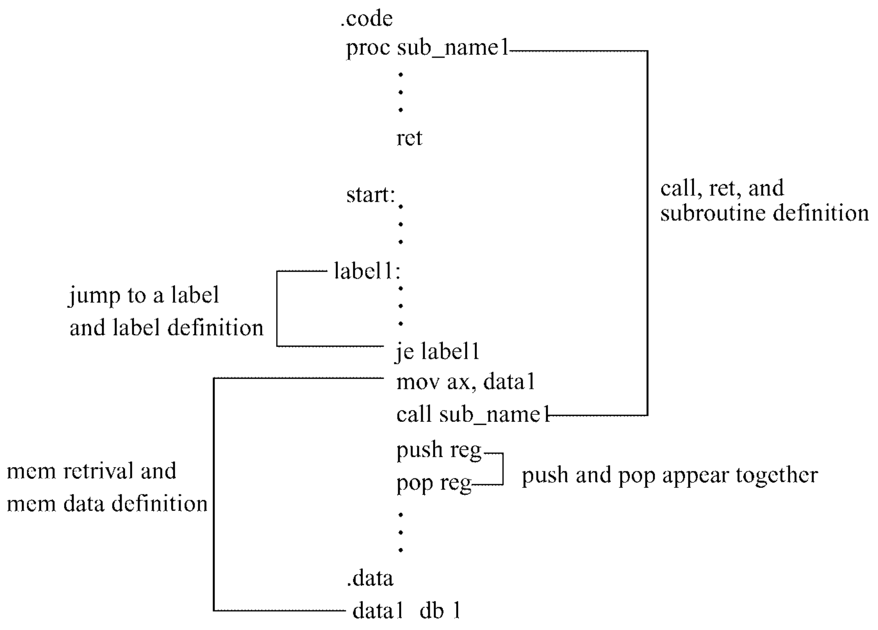

.code
proc sub_name1
ret
start:
label1:
je label1
mov ax, data1
call sub_name1
push reg
pop reg
.data
data1 db 1
call, ret, and subroutine definition
jump to a label and label definition
mem retrival and mem data definition
push and pop appear together

Fig. 11. Instructions appearing together requirement.

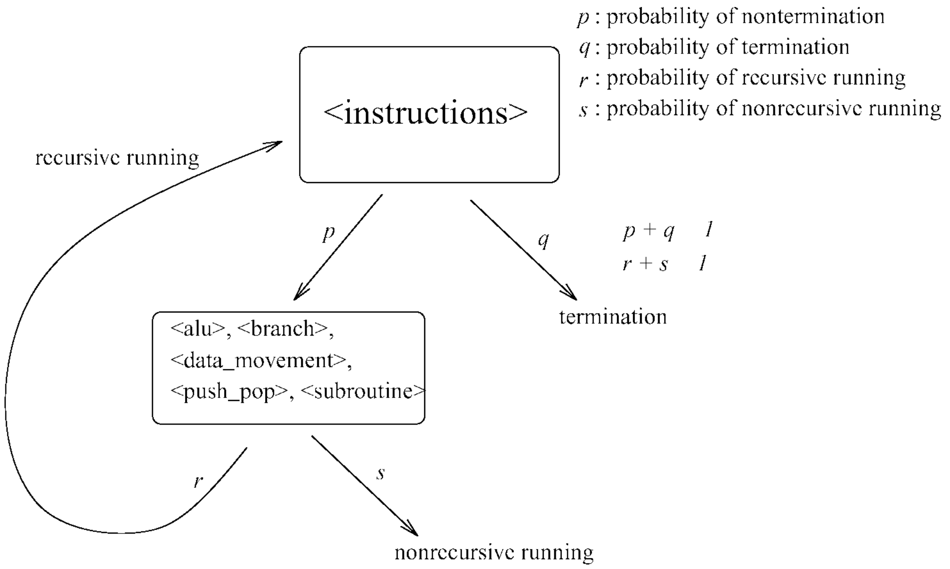
data definition, etc. Figure 11 shows an example of the instructions appearing together requirement. When the *je label*1 instruction is selected, we must decide on the location of *lable*1. In addition, if *lable*1 is a backward branch location, we also have to insert a loop-exit code, as discussed above. The second operand in the *mov* instruction is a memory data location (*data*1). If the operand type is memory data, we must define it in the data segment. When generating a *call* instruction, the corresponding subroutine must be defined. Also, a *push* instruction must appear together with a *pop* instruction.

5.5 Bounded Program Size

The size of a test program generated by an APG should be bounded in a range, but not fixed in size. For example, when generating a *call* instruction, a subroutine definition should be completed before the test program generation terminates. All instructions included in the instructions appearing together requirement discussed above have the same restriction. The program size problem can be resolved naturally in our approach by setting different probabilities, as shown in Figure 12. In the following, we derive the mean of a program size, N. The parameter, l, is used to represent the average number of generated instructions for different instruction types. Therefore,

$$N = p \times s \times (l) + p \times r \times p \times s \times (2l) + \cdots + p^k \times r^{(k-1)} \times s \times (kl) + \cdots \quad (6)$$

$$= \sum_{k=1}^{\infty} p^n \times r^{(k-1)} \times s \times (kl). \quad (7)$$



<instructions>
p : probability of nontermination
q : probability of termination
r : probability of recursive running
s : probability of nonrecursive running
recursive running
p
q
p + q 1
r + s 1
termination
<alu>, <branch>, <data_movement>, <push_pop>, <subroutine>
r
s
nonrecursive running

Fig. 12. Controlling program size by setting different probabilities.

As $k \to \infty$, we have

$$N = \frac{s \times p \times l}{(1 - pr)^2} \tag{8}$$

$$= \frac{p \times (1 - r) \times l}{(1 - pr)^2}, \tag{9}$$

where $s = 1 - r$.

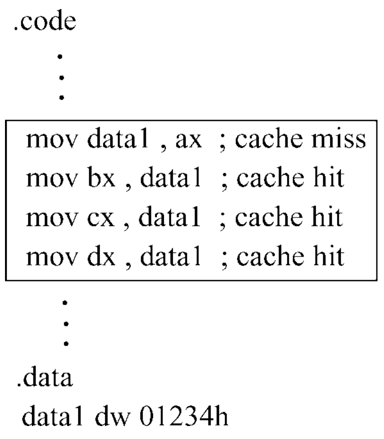
Thus, we can get a simple expression to estimate the program size:

$$N \approx \frac{l}{(1 - r)} \quad \text{for} \quad p \approx 1. \tag{10}$$

The usefulness of program size estimation will be clear in Figure 28.

5.6 Test for Data Cache

Since a data cache is prone to design errors, we must pay special attention to the testing of the data cache. Three operations should be verified for data cache testing. They are *cache hit*, *cache miss*, and *cache replacement*. Figure 13 shows a test for cache hit and cache miss. In this instruction sequence, it will cause some cache hits and cache misses. Figure 14 shows a test for cache replacement. The cache architecture is assumed to be four-way set-associative. We use a *base pointer*, *bp*, to locate the base address and the offset addresses. These memory addresses all map to the same cache set. For example, in Figure 14, the *base pointer* points to the address of *data*1. The number of sets in the cache is k. The data located in the addresses $[bp]$, $[bp] + k \times 1$, $[bp] + k \times 2, \ldots,$ and $[bp] + k \times i$ (where i is an integer constrained by memory size) will all map to the same set, *set* 0. Thus, the fifth memory access instruction addressed to these locations will cause a cache replacement.



```
.code
  •
  •
  •
mov data1 , ax  ; cache miss
mov bx , data1  ; cache hit
mov cx , data1  ; cache hit
mov dx , data1  ; cache hit
  •
  •
  •
.data
data1 dw 01234h
```

Fig. 13. Test for cache hit/miss.

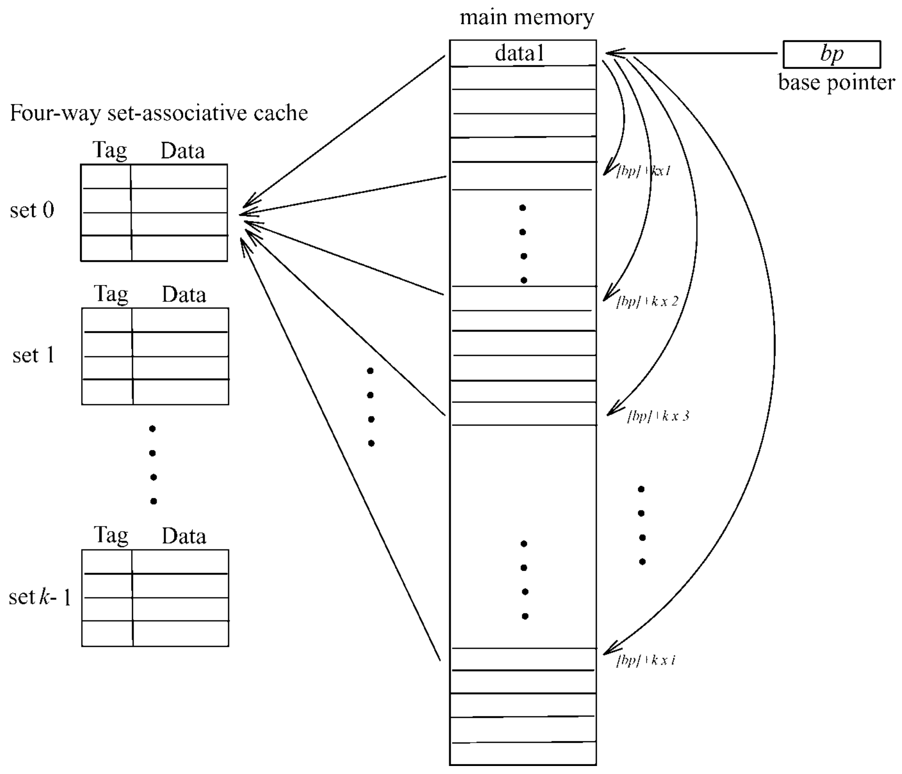
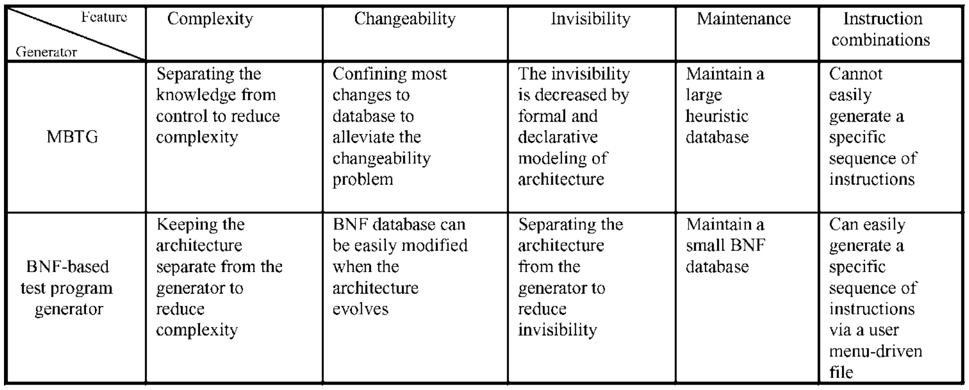


Fig. 14. Test for cache replacement.

5.7 BNF-Based APG Compared with Other Approaches

Since MPTG is a deterministic test generator, we only compare our BNF-based APG with MBTG [Lichtenstein et al. 1994] and AVPGEN [Chandra et al. 1995]. Figure 15 summarizes the similarities and differences between our APG and MBTG in five aspects. The two approaches are comparable in terms of



Feature / Generator	Complexity	Changeability	Invisibility	Maintenance	Instruction combinations
MBTG	Separating the knowledge from control to reduce complexity	Confining most changes to database to alleviate the changeability problem	The invisibility is decreased by formal and declarative modeling of architecture	Maintain a large heuristic database	Cannot easily generate a specific sequence of instructions
BNF-based test program generator	Keeping the architecture separate from the generator to reduce complexity	BNF database can be easily modified when the architecture evolves	Separating the architecture from the generator to reduce invisibility	Maintain a small BNF database	Can easily generate a specific sequence of instructions via a user menu-driven file

Fig. 15. A comparison between our BNF-based APG and MBTG.

complexity, *changeability*, and *invisibility*. However, our method is superior to MBTG in terms of *maintenance* and *instruction combinations*. MBTG needs to maintain a large heuristic database and cannot easily generate a specific sequence of instructions [Lichtenstein et al. 1994; Malik et al. 1997], as addressed in Section 4. Our BNF-based APG only needs to maintain a small BNF database, as described in the following. There are 345 instructions for the Pentium Pro processor, and we provide each instruction with a production rule. We also provide some production rules for operands and addressing modes selection. The total size of the BNF developed is about 550 production rules. In addition, our APG can generate a specific sequence of instructions easily via the specification of a user menu-driven file.

In MBTG, a test knowledge base is added to increase the probability to test corner cases [Lichtenstein et al. 1994]. The knowledge base represents test engineers' expertise. This means that test engineers must know what corner cases are and what kinds of instruction sequences may test corner cases. In our approach, if test engineers know what kinds of instruction sequences may test corner cases, they specify them in a user menu-driven file to increase the probability of generating the corner cases. We also allow test engineers to add macros to the test programs generated by the APG. A macro is a hand-written program segment, which is an instruction sequence not easily generated by the APG. The APG will include the macros in the generated test programs.

Comparing the BNF-based APG with IBM's AVPGEN [Chandra et al. 1995], we conclude that the BNF-based APG can do the things done by the AVPGEN, like operands initialization and exception control. However, our BNF-based APG is easier to maintain than the AVPGEN. The AVPGEN needs to carefully design four blocks: *refinement*, *dispatching*, *solving*, and *pattern generation* [Chandra et al. 1995] in order to generate specific exceptions and to prevent the test program from entering invalid states, while our APG only needs to maintain the APG preprocessor and BNF production rules, as shown in Figure 3. The user menu-driven file is easy to prepare and the BNF-based pattern generator can guarantee that the generated test programs are always in valid states.

```
1. <asm_program>::=<program> #dump_s_pool #dump_i_pool #dump_d_pool
2. <program>::= .model small <small_block>
3. <small_block>::=.code start: mov ax,@data mov ds, ax
                   <instructions> mov ah, 4ch int 21h .stack end start
4. <instructions>::= <alu_inst> | <branch_inst> | <subroutine_inst>|
                   <data_movement_inst> | <push_pop_inst>
5. <alu_inst>:=add <reg> , <reg> | add <reg>, <mem>
6. <reg>::= ax | bx | cx | dx
7. <mem>::= mem8 | mem16 #create mem
8. <branch_inst>::=<forward_branch> | <backward_branch>
9. <forward_branch>::= jmp label <instructions> #create label
10. <backward_branch>::= #create label #add loop-exit code <instructions> jmp label
11. <subroutine_inst>::= call subroutine_name #creare subroutine
12. <subroutine>::=subroutine_name proc <subinstructions> ret
13. <subinstructions>::=<alu_inst>| <branch_inst> | <data_movement_inst> | <push_pop_inst>
14. <push_pop_inst>::= push <reg> <instructions> pop <reg>
```

Fig. 16. A sample BNF.

6. APG IMPLEMENTATION

Figure 16 shows a sample BNF. Action symbols (starting with #) are used to specify particular tasks. Production numbers are added for ease of discussion. There are 14 productions in this BNF. Figure 17 is an example assembly test program generated according to this BNF. There are 20 lines in the test program. Note that we have used the X86 processor architecture to illustrate our design method. However, our method can be easily extended to other processor architectures (e.g., instructions with three operands). We only need to redefine BNF production rules, as shown in Figure 16, based on other processor architectures. In the following, we will trace this BNF and discuss how our APG uses this BNF to efficiently construct the test program.

6.1 Code Generation

The first production defines that the generation of an assembly program can be divided into three parts. They are *subroutine*, *main body*, and *memory data*. In Figure 17, lines 3 to 5 are a subroutine, lines 17 to 18 are memory data, and the other lines are the main body. The second production says that an assembly program starts with a string **.model small**. Production 3 defines that a small block is constructed with some instructions and delimited by some required codes (lines 2, 6, 7, 15, 16, 19, and 20 in Figure 17). Production 4 defines that there are five types of instructions in our test programs. They are *alu*, *branch*, *subroutine*, *data movement*, and *push/pop*. Production 5 says that **add** is an *alu* type instruction and the operands can be *register-register* or *register-memory*. Lines 8 and 13 in Figure 17 are constructed according to this rule. Production 6

```
1.          .model small
2.          .code
3.    sub1   proc            ; subroutine, created by #creat_subroutine
4.           mov dx, ax
5.           ret             ; ret appears together with call and proc
6.   start:  mov ax, @data
7.           mov ds, ax
8.           add ax, bx
9.           mov dx, mem1    ; memory access, invoke #create_mem16
10.          call sub1       ; call subroutine, invoke #create_subroutine
11.          jump label1     ; jump label, invoke #create_label to define a label
12.  label1  push ax         ; label1, defined by #create_label
13.          add dx, dx
14.          pop bx          ; pop appears together with push
15.          mov ah, 4ch
16.          int 21h
17.         .data
18.   mem1   db '1234'       ; mem data defined by #create_mem16
19.         .stack
20.          end start
```

Fig. 17. An example assembly program.

says that four registers can be chosen (*ax, bx, cx,* and *dx*). Production 7 means that there are two kinds of memory data, *byte* and *word*, and that they are represented as *mem*8 and *mem*16. The action symbol, *create_mem*, will define a memory datum in the data segment. Line 9 in Figure 17 uses a memory location as the second operand, and line 18 defines the memory datum. Production 8 says that branch instructions can be divided into *forward branch* and *backward branch*. Forward branch is defined in production 9 and backward branch is defined in production 10. We can see that a backward branch creates a label before a branch operator and adds a loop-exit code after the label definition. Between a branch operator and the corresponding label, we can insert other instructions. There is no loop-exit code for forward branch. Line 11 in Figure 17 is a forward branch, and line 12 defines a label. Production 11 defines a subroutine call. The action symbol *create_subroutine* will use productions 12 and 13 to generate a subroutine. We can see that no more subroutine call is allowed in a subroutine. In Figure 17, lines 3, 4, 5, and 10 are generated by these two productions. Production 14 defines a push/pop structure. Between *push* and *pop*, we can insert other instructions. Lines 12 to 14 are generated by this production.

6.2 Structure of a Test Program

Figure 18(a) shows the structure of a test program. We can see that there are other push/pop structures inside a push/pop. Branch instructions may have the same structure. These two types of instructions must be carefully considered to prevent the test program from entering illegal states. For example, Figure 18(b)

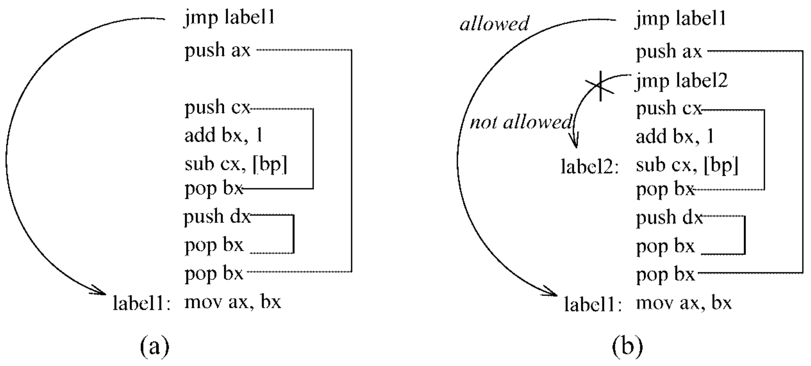


Fig. 18. Structure of a test program.

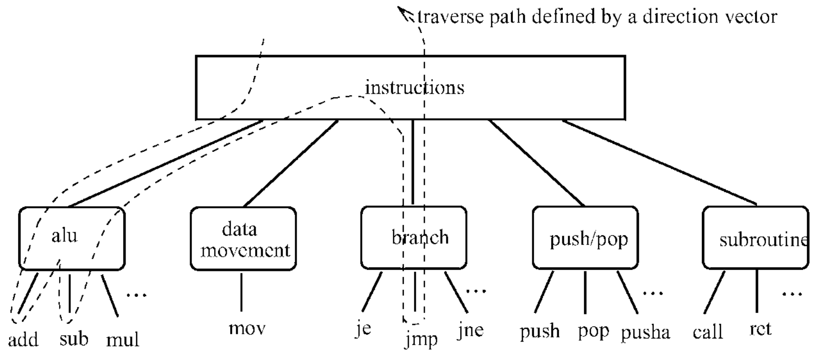


Fig. 19. Using a direction vector to guide the traverse path of a parse tree.

shows an illegal test program. After the instruction, **jmp label2**, is executed, the instruction sequence jumps to the middle of a push/pop structure. This causes a pop being executed without a push first. If this situation occurs too many times, the stack may overflow and the verification task may be crushed. In our *top-down recursive descent parsing* method, this structure problem can easily be resolved. By productions 9 and 14 in Figure 16, it is not possible to generate a test program like Figure 18(b). The two productions emphasize that a *push* must appear together with a *pop*, and this structure will never be broken by branch instructions.

6.3 User-Controlled Part

We have discussed (in Sections 5.1.2 and 5.1.3) that the generation of instruction sequence-specified test programs and module-specified test programs can be controlled by user menu-driven files. We use a *direction vector* to define the traverse path of a parse tree and to generate a specific sequence of instructions. That is, we force the APG to go through a particular path so that the required instructions can be generated. Figure 19 shows that, by defining a direction vector, we can obtain a specific sequence of test instructions (*add-sub-jmp*). A direction vector can be derived according to a user menu-driven file (e.g., as in Figure 5). Thus, we can generate instruction sequence-specified and

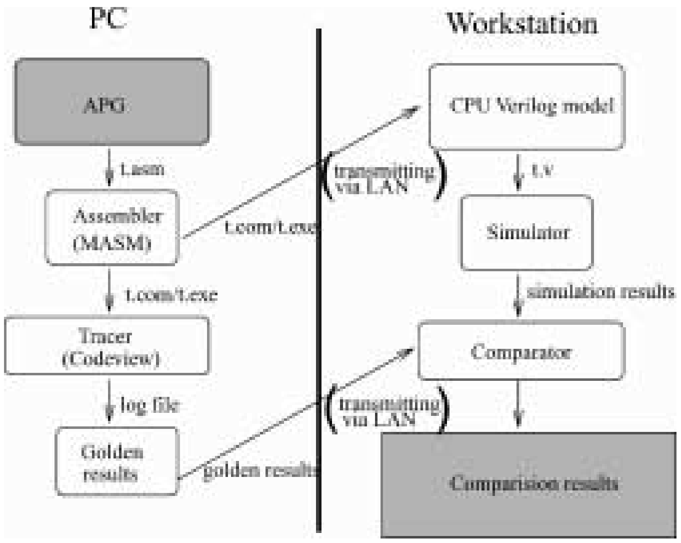


Fig. 20. Verification flow between PC and workstation.

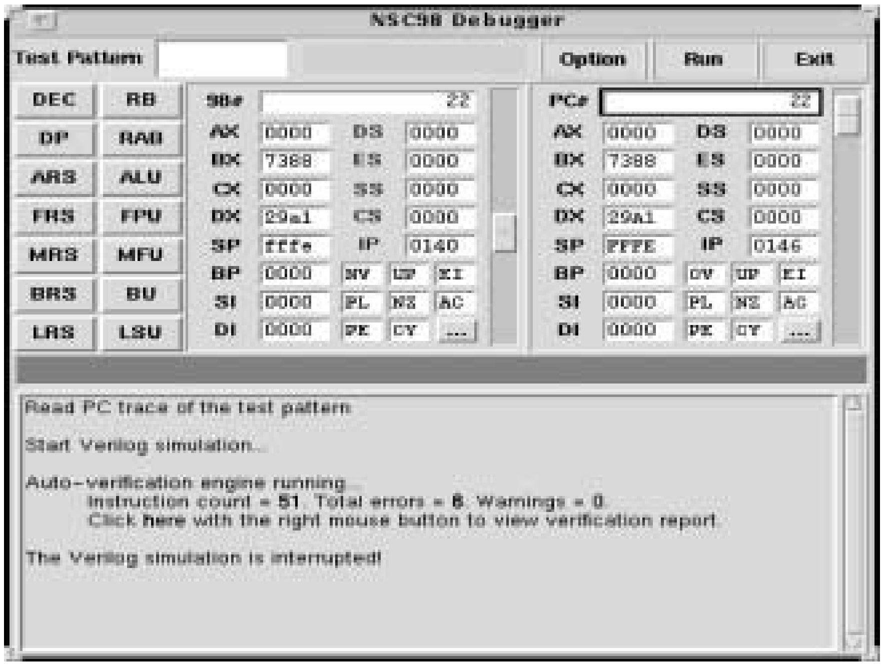
module-specified test programs by user menu-driven files. If we want to generate test programs with high data dependency, we may confine the traverse path of a parse tree to use a subset of operands. In addition, the test engineers need to cooperate with the design engineers to get the information about instruction sequence generation based on microarchitecture knowledge. After that, the test engineers can write a user menu-driven file to guide the APG to automatically generate an instruction sequence based on microarchitecture knowledge.

6.4 Other Types of Test Programs Not Included in Our APG

There are other types of test programs that are not suitable to be automatically generated by an APG, for example, the X86 test programs in protected mode. Protected mode testing is one of the most difficult parts of X86-compatible microprocessor verification. Some special actions must be taken—for example, initialization—in order to switch a PC into the protected mode. Thus, these kinds of test programs are not feasible to be generated by an APG. The resolution of generating these kinds of test programs is to use the *macro-include* method. That is, we prepare some macros and include them in the test programs generated by the APG.

7. INTEGRATED DEBUGGING ENVIRONMENT FOR VERIFICATION

Figure 20 shows the verification flow between a PC and a workstation. In the PC, the APG generates a *t.asm* assembly program and then the assembler (*MASM*) assembles the assembly program. After successfully assembling the *t.asm* file, the assembler produces a *t.com* or *t.exe* file. Then the tracer traces the *t.com* or *t.exe* file and saves the golden results in a log file. In the workstation, the microprocessor Verilog model runs the executable file (*t.com* or *t.exe*) using a simulator from the PC and saves the simulation results. After



```
SB#  22                          PC#  22
AX  0000   DS  0000              AX  0000   DS  0000
BX  7388   ES  0000              BX  7388   ES  0000
CX  0000   SS  0000              CX  0000   SS  0000
DX  29a1   CS  0000              DX  29A1   CS  0000
SP  fffe   IP  0140              SP  FFFE   IP  0146
BP  0000                         BP  0000
SI  0000                         SI  0000
DI  0000                         DI  0000
```

```
Read PC trace of the test pattern

Start Verilog simulation...

Auto-verification engine running...
        Instruction count = 51. Total errors = 6. Warnings = 0.
        Click here with the right mouse button to view verification report.

The Verilog simulation is interrupted!
```

Fig. 21. Integrated debugging environment for verification.

comparing the results from the PC and the workstation, we can check if the test program generated by the APG is correctly executed by the microprocessor Verilog model. In this way, the microprocessor Verilog model can be verified and debugged.

Figure 21 shows an integrated debugging environment for verification. This environment contains a smart debugger to compare the execution (expected) results of test programs from the PC and the microprocessor Verilog model. If a mismatch occurs, which instructions have caused this error should be figured out. In this test case, the user menu-driven file of this test program is shown in Figure 5. We can see that when executing instruction 22, the *overflow* flag on the PC is not equal to the one on the microprocessor Verilog model. The bug is then reported along with this test program for further debugging.

8. INTEGRATED APG & COVERAGE TOOL AND EXPERIMENTAL RESULTS

In this section, we present our integrated APG & coverage tool and demonstrate some experimental results.

8.1 Our Integrated APG & Coverage Tool

Our APG has been integrated with a coverage tool, a test program pool, and a simulator, *TR* [Liu 1999], as shown in Figure 22. We explain each component as follows:

—*Automatic test program generator*: At the first run, no coverage report is available. The APG generates a test program by following the directions of

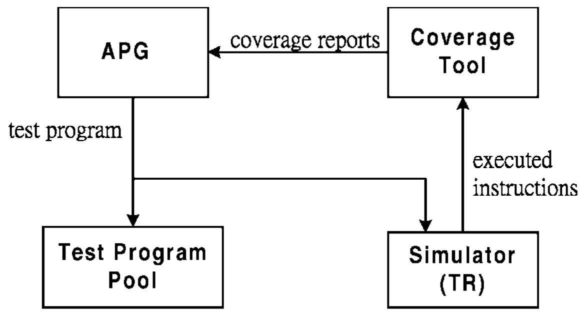


Fig. 22. System configuration of our integrated APG & coverage tool.

the user menu-driven file. This generated program is then sent to the test program pool and the simulator, *TR*. On subsequent runs, coverage reports about the previously generated test programs are collected. Based on the coverage reports, the APG will try to generate instructions that rarely appeared or were not generated before. The generation process is terminated when a specified coverage is achieved.

—*Test program pool*: Whenever the APG generates a new test program, it sends the test program to this pool. This component collects all generated test programs and forms a test suite. This test suite can be used to test compatible microprocessor designs.

—*Simulator*: The simulator, *TR* [Liu 1999], is used to simulate the execution of a generated test program. It can record all executed instructions. This information is then sent to the coverage tool for analysis.

—*Coverage tool*: By knowing all executed instructions, the coverage tool can analyze the coverage of a single instruction, instruction combinations, data dependency, and data cache access. If a specified coverage has been achieved, the coverage tool will notify the APG. If not, the coverage tool will send a coverage report to the APG for the next generation process.

Figure 23 shows a coverage tool used to refine and evaluate an APG. The coverage tool keeps track of all signals, states, and exceptions of a design. If there are some signals that are never touched, some states that are never reached, or some exceptions that never happen, we then want to change user menu-driven files to generate test programs to cover those situations. However, if there are some cases that the APG cannot generate, we should refine the APG to cover these cases. If such cases are really not feasible to be generated by an APG, manual coding test programs (or macros) is necessary. Thus, a faithful coverage tool [Fine and Ziv 2003] is very important in developing an efficient APG.

8.1.1 *Coverage Reports.* The coverage reports in Grinwald et al. [1998] and Wang and Liu [1998] were of numeric meanings only. We need more information to guide the APG. Therefore, each instruction is associated with a 3×3 coverage matrix. Figure 24(b) shows an example coverage matrix for *SHLD* shown in Figure 24(a). There are three addressing modes and three operands for each instruction. The addressing modes are register, immediate,

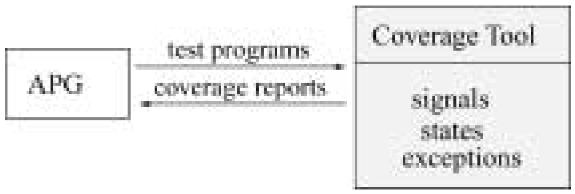


Fig. 23. Refining an APG by a coverage tool.

SHLD AX, BX, 5H

(a)

operand size / addressing mode	8 bits	16 bits	32 bits
Register	NotGenerated	Generated	NotGenerated
Immediate	Generated	NotUsed	NotUsed
Memory	NotGenerated	NotGenerated	NotGenerated

(b)

Fig. 24. (a) A generated instruction *SHLD*, (b) the 3 × 3 coverage matrix for *SHLD* reported by the coverage tool.

and memory (direct, indirect, base, index, and base index) [Intel Corporation 1996a, 1997b]. Operand sizes are 8, 16, and 32 bits [Intel Corporation 1997a]. Each entry in Figure 24(b) stands for a combination of an addressing mode and an operand size, and may have one of the three values: NotUsed, NotGenerated, and Generated. If the operand of one instruction cannot be an 8-bit register, the corresponding entry is marked as NotUsed. If a combination is generated in test programs, it is marked as Generated. Otherwise, entries are marked as Not-Generated. For example, if the instruction in Figure 24(a) is executed, a 3 × 3 coverage matrix of *SHLD* will be reported by the coverage tool, as shown in Figure 24(b), and the coverage of *SHLD* is $\frac{2}{7}$.

8.1.2 *Flowchart of the Integrated APG and Coverage Tool.* The flowchart of the integrated APG and coverage tool is shown in Figure 25, and we describe each step as follows:

(1) If there is no instruction that is not analyzed yet in the file returned by the *TR*, go to step 2. Otherwise, go to step 3.
(2) Evaluate the coverage of test programs generated. If the desired coverage has been achieved, notify the APG of this information. Then, the control returns to the APG.
(3) Read an executed instruction from the file and identify the instruction type it belongs. If this instruction has any operand, go to step 4. Otherwise, go to step 5.
(4) Update the 3 × 3 coverage matrix that is associated with the instruction. Then, go to step 1.
(5) Mark this instruction as 100% covered. Then, go to step 1.

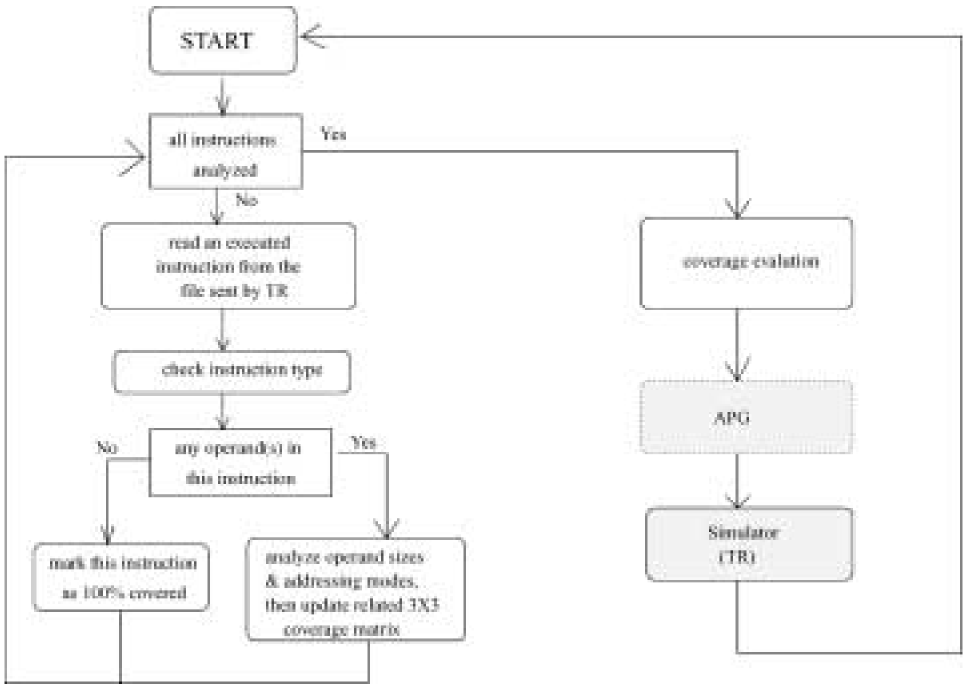

START
all instructions analyzed
Yes
No
read an executed instruction from the file sent by TR
check instruction type
any operand(s) in this instruction
No
Yes
mark this instruction as 100% covered
analyze operand sizes & addressing modes, then update related 3X3 coverage matrix
coverage evalution
APG
Simulator (TR)

Fig. 25. Flowchart of the integrated APG and coverage tool.

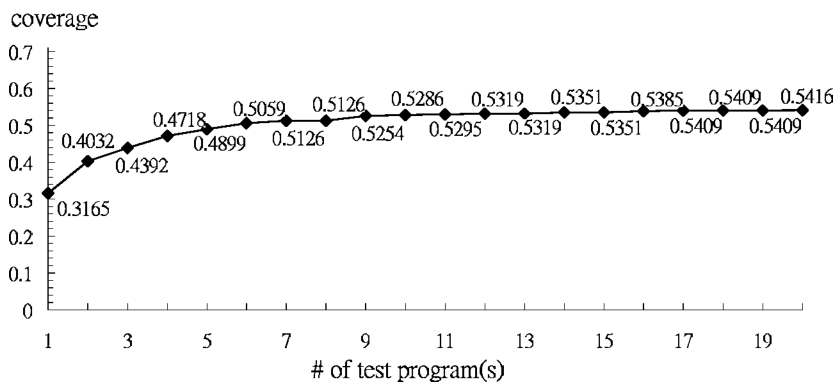


Fig. 26. The overall coverage of generated test programs without coverage feedback.

8.2 Experimental Results

Figures 26 and 27 show the overall coverage of generated test programs without coverage feedback and with coverage feedback, respectively. The instructions generated belong to Integer Instructions [Intel Corporation 1996a]. The average program size in our experiment is 800 lines and a total of 100 test programs are generated by the APG. Note that the average program size is determined by simulation. Figure 28 shows the coverage reaches the maximum when the program size exceeds 800 lines. With this information, we can deduce the optimal

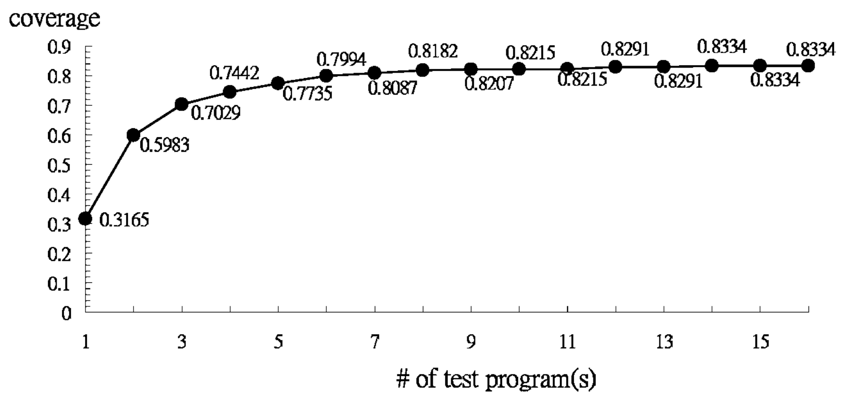


Fig. 27. The overall coverage of test programs with coverage feedback.

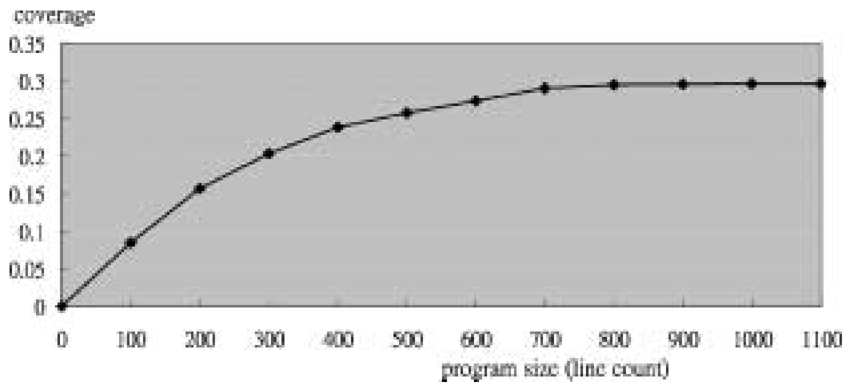


Fig. 28. Coverage versus program size.

value of the *r* variable in the program size expression (Equation (10)), and use the value in the test program generation process. In Figure 26, after the number of test programs exceeds six, the coverage increases slowly. Even with 20 test programs, the coverage is only 0.5416. This demonstrates a situation that generated test programs with low coverage is a serious problem of the APG without coverage feedback, and it implies that the APG may generate some instructions repeatedly.

Comparing Figure 27 with Figure 26, we found that the APG with coverage feedback from the coverage tool can generate higher coverage (0.8334 vs. 0.5416, 60% improved) test programs with a smaller number (20 vs. 16) of test programs than the APG without coverage feedback.

9. CONCLUSIONS

In this paper, a new BNF-based method to develop an APG has been presented. We have discussed design issues and have resolved them by our method. An APG generates test programs according to some predefined production rules. These rules should be general enough to have high coverage and restricted enough to prevent the test programs from reaching illegal states. We have used BNF to define the production rules and build the APG by the *top-down recursive descent parsing* method. By this method, the test programs generated by our APG have the features of no infinite loop, not entering illegal states, controllable data dependency, flexible program size, and being data cache testable. These features are usually hard to realize but can be easily resolved by our BNF-based method. According to the experience of practical implementation and evaluation, our method has been shown to be efficient and feasible for the development of an APG compared with other approaches. We have also incorporated a coverage tool to refine our APG. Experimental results show that our integrated APG & coverage tool only needs to generate a small number of test programs to sustain high coverage.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments and suggestions, which helped enhance the quality of this paper.

REFERENCES

ADIR, A., MARCUS, E., AND EMEK, R. 2002. Adaptive test program generation: Planning for the unplanned. *High-Level Design Validation and Test Workshop* (Oct. 2002). 83-88.

ADIR, A., MARCUS, E., RIMON, M., AND VOSKOBOYNICK, A. 2001. Improving test quality through resource reallocation. *High-Level Design Validation and Test Workshop* (Nov. 2001). 64–69.

ADIR, A. AND SHUREK, G. 2002. Generating concurrent test-programs with collisions for multiprocessor verification. *High-Level Design Validation and Test Workshop* (Oct. 2002). 77–82.

AHARON, A., BAR-DAVID, A., DORFMAN, B., GOFMAN, E., LEIBOWITZ, M., AND SCHWARTZBURD, V. 1991. Verification of the IBM RISC System/6000 by a dynamic biased pseudo-random test program generator. *IBM Syst. J. 30*, 4, 527–538.

AHARON, A., GOODMAN, D., LEVINGER, M., LICHTENSTEIN, Y., MALK, Y., METZGER, C., MOLCHO, M., AND SHUREK, G. 1995. Test program generation for functional verification of PowerPC processors in IBM. In *Proceedings of the 32nd Design Automatic Conference* (June 1995). 274–285.

AL-ARIAN, S. A. AND NORDENSO, M. 1989. FUNTEST: A functional automatic test pattern generator. In *Proceedings of the 1989 International Test Conference* (Aug. 1989). 945–946.

BIN, E., EMEK, R., SHUREK, G., AND ZIV, A. 2002. Using constraint satisfaction formulations and solution techniques for random test program generation. *IBM Syst. J. 41*, 3, 386–402.

BRAHAME, D. AND ABRAHAM, J. A. 1984. Functional testing of microprocessors. *IEEE Trans. Comput, C-33*, 6 (June), 475–485.

CHANDRA, A., IYENGAR, V., JAMESON, D., JAWALEKAR, R., NAIR, I., ROSEN, B., MULLEN, M., YOON, J., ARMONI, R., GEIST, D., AND WOLFSTHAL, Y. 1995. AVPGEN—A test generator for architecture verification. *IEEE Trans. VLSI Syst. 3*, 2 (June), 188–200.

EMEK, R., JAEGER, I., NAVEH, Y., ALONI, G., BERGMAN, G., DOZORETS, I., FARKASH, M., GOLDIN, A., AND KATS, Y. 2002. X-Gen: A random test-case generator for systems and SoC. *High-Level Design Validation and Test Workshop* (Oct. 2002). 145–150.

FINE, S. AND ZIV, A. 2003. Coverage directed test generation for functional verification using Bavesian networks. *Design Automaion Conference* (June 2003). 286–291.

FISCHER, C. N. AND LEBLANCE, R. J. JR. 1988. *Crafting a Compiler*. Benjamin Cummings, San Francisco, CA, 26–43.

FOURNIER, L., ARBETMAN, Y., AND LEVINGER, M. 1999. Functional verification methodology for microprocessors using the Genesys test-program generator—application to the x86 microprocessors family. *Design, Automation and Test in Europe Conference and Exhibition* (March 1999). 434–441.

GRINWALD, R., HAREL, E., ROGAD, M., UR, S., AND ZIV, A. 1998. User defined coverage—a tool supported methodology for design verification. In *Proceedings of Design Automatic Conference* (June 1998). 159–163.

HENNESSY, J. N. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, San Francisco, CA.

HOLUB, A. I. 1990. *Compiler Design in C*. Prentice-Hall, Englewood Cliffs, NJ.

HU, E., YEH, B., AND CHAN, T. 1994. A methodology for design verification. In *Proceedings of the IEEE ICC* (Sep. 1994). 236–239.

HWANG, K. 1993. *Advanced Computer Architecture*. McGraw-Hill, New York, NY, 312–313.

INTEL CORPORATION. 1996a. *Pentium Pro Family Developer's Manual, Vol. 2: Programmer's Reference Manual*. Intel, Santa Clara, CA.

INTEL CORPORATION. 1996b. *Pentium Pro Processor User's Manual, Volume 3: Architecture and Programming Manual*. Intel, Santa Clara, CA.

INTEL CORPORATION. 1997a. *Intel Architecture Software Developer's Manual, Vol. 2: Instruction Set Reference*. Intel, Santa Clara, CA.

INTEL CORPORATION. 1997b. *Intel Architecture Software Developer's Manual, Vol. 3: System Programming Guide*. Intel, Santa Clara, CA.

JOHNES, D., YANG, R., KWANG, M., AND HARPER, G. 1991. Verification techniques for a MIPS compatible embedded control processor. In *Proceedings of the IEEE ICC* (Oct. 1991). 329–332.

KLUG, H. P. 1988. Microprocessor testing by instruction sequences derived from random patterns. In *Proceedings of the 1988 International Test Conference* (Sep. 1988). 73–80.

LICHTENSTEIN, Y., MALKA, Y., AND AHARON, A. 1994. Model based test generation for processor verification. In *Proceedings of the Sixth Innovative Applications of Artificial Intelligence Conference*. 83–94.

LIU, T. T. 1999. Super TRacer Program. Available online at http://www.netease.com/ayliutt/eng.htm.

MALIK, N., ROBERTS, S., PITA, A., AND DOBSON, R. 1997. Automaton: An autonomous coverage-based multiprocessor system verification environment. In *Proceedings of the 8th IEEE International Workshop on Rapid System Prototyping: Shortening the Path from Specification to Prototype* (June 1997). 168–172.

MIYAKE, J. AND BROWN, G. 1992. Functional testing of modern microprocessors. In *Proceedings of the European Conference on Design Automation* (March 1992). 350–354.

MIYAKE, J. AND BROWN, G. 1994. Automatic test generation for functional verification of microprocessors. In *Proceedings of the Third Asian Test Symposium* (Nov. 1994). 292–297.

O'KRAFKA, B., MANDYAM, S., KREULEN, J., RAGHAVAN, R., SAHA, A., AND MALIK, N. 1995. MPTG: A portable test generator for cache-coherent multiprocessors. In *Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference*. 38–44.

SAVIR, J. AND BARDELL, P. H. 1984. On random pattern test length. *IEEE Trans. Comput. C-33*, 6 (June), 467–474.

THATTE, S. M. AND ABRAHAM, J. A. 1980. Test generation for microprocessors. *IEEE Trans. Comput. C-29*, 6 (June), 429–441.

TURUMELLA, B., KABAKIBO, A., AND BOGADI, M. 1995. Design verification of a super-scalar RISC processor. In *Proceedings of the IEEE ICC* (June 1995). 472–477.

WANG, K. C. AND LIU, S. J. 1998. Coverage evaluation for test programs of X86 compatible microprocessors. In *Proceedings of the 1998 International Computer Symposium*. 60–64.

Received March 2000; revised June 2003; accepted October 2003