A New Verification Methodology for Complex Pipeline Behavior

Kazuyoshi KOHNO
Toshiba Corporation Semiconductor Company
580-1, Horikawa-Cho, Saiwai-Ku
Kawasaki, 212-8520, JAPAN
kohno@sdel.toshiba.co.jp

Nobu MATSUMOTO
Toshiba Corporation Semiconductor Company
580-1, Horikawa-Cho, Saiwai-Ku
Kawasaki, 212-8520, JAPAN
matsumot@sdel.toshiba.co.jp

ABSTRACT

A new test program generation tool, mVpGen, is developed for verifying pipeline design of microprocessors. The only inputs mVpGen requires are pipeline-behavior specifications; it automatically generates test cases at first from pipeline-behavior specifications and then automatically generates test programs corresponding to the test cases.

Test programs for verifying complex pipeline behavior such as hazard and branch or hazard and exception, are generated. mVpGen has been integrated into a verification system for verifying RTL descriptions of a real microprocessor design and complex bugs that remained hidden in the RTL descriptions are detected.

1. INTRODUCTION

Verification time is increasing as a proportion of the design period, and there is a growing need to find ways of shortening it. There are two main ways of confirming the correctness of RTL implementations: simulation-based verification and formal verification. Simulation-based methods are widely used to verify microprocessor design, because formal verification methods cannot handle the entire designs of complex microprocessors.

All possible instruction sequences are required in order to confirm the correctness of a given microprocessor design under simulation-based verification. But it is impossible to generate and/or to simulate them in real time. One solution is to establish an efficient method of generating instruction sequences with good coverage and simulating as many of them as possible.

Various ways of generating test programs for verifying microprocessors have been developed. The simplest technique is random sequence generation, but it requires a huge number of instructions to verify complex microprocessors sufficiently. The pseudo-random sequence generation technique, respecting which randomness is controllable by probabilities given by users, has been reported [5]. But, it significantly depends on the experience of users whether implementations are sufficiently verified.

As indicated above, it is not advisable to rely solely on the use of (pseudo-)random sequence generation techniques in the verification phase; therefore, effective verification program generation techniques for some specific area of a verification space have been proposed.

An efficient test program generation method for simulation-based pipeline verification using techniques developed for formal verification is presented in [1]. Test programs are generated from a pipeline model using state enumeration techniques. The test generation method described in [1] is an excellent way to detect simple bugs respecting pipeline hazards; however, there are many complex bugs respecting which hazard and interrupt occur simultaneously [2]. It is also necessary to establish a comprehensive methodology to cover the entire verification process, which includes the reference simulation environment.

In order to solve the above problems, we have developed a new test program generation tool and verification methodology for verifying pipeline design of microprocessors and applied them for verifying a real microprocessor design.

The organization of this paper is as follows. Section 2 presents an overview of the features of mVpGen. Pipeline-behavior specifications, which are the input data of mVpGen, are explained in section 3. Methods of automatically generating test cases, pipeline models, and verification programs are also mentioned in section 3. In section 4, the results of verification for a real microprocessor design are described. Conclusions are given in section 5.

2. OVERVIEW

2.1 Program generation using BDDs

This section describes how to generate verification programs. Our test program generation tool, mVpGen, uses the Binary Decision Diagrams(BDDs) which are graph representations of logic functions. It is known that large logic functions are represented compactly by BDDs. Since a subset S of X can be represented as a kind of logic functions – characteristic function C_S, where $C_S(x)$ returns true if $x \in S$ and returns false if $x \notin S$, sets can be represented by BDDs. The test generation flow is as follows: (1) represent pipeline behavior as FSMs, (2) compute a set of reachable states, and (3) generate an instruction sequence for each reachable state. The process of test generation is implemented in BDDs since both next state functions of FSMs, which are logic functions, and sets can be represented by BDDs. To avoid state explosion, mVpGen uses Partitioned Transition Relation(PTR), which is an efficient method of calculating reachable states, presented in [4]. The way to represent pipeline behavior models in BDDs and the algorithm to generate verification programs are described in section 3.3 and 3.4, respectively.

2.2 Features

mVpGen automatically generates test programs for verifying pipeline behavior; moreover, it has the following features:

- generating test cases from pipeline-behavior specifications
- generating a pipeline-behavior model from pipeline-behavior specifications
- generating complex test cases, in which hazard and interrupt/exception, hazard and branch occur simultaneously

These features are explained in section 3.

3. METHOD FOR TEST PROGRAM GENERATION AND VERIFICATION

A procedure for verifying pipeline behavior for resolving pipeline hazards is as follows : (1) generate instruction sequences which cause a pipeline hazard for the verification, and (2) simulate using the generated instruction sequences and investigate the behavior for resolving hazard by bypassing or stall. mVpGen generates test programs according to the flow described below:

Step 1 coding a pipeline-behavior specification description

Step 2 test case generation

Step 3 pipeline model generation

Step 4 template generation

Step 5 template derivation

Step 6 test program generation

Step 7 verification by comparison

The procedure of test program generation is composed of Step 3 to Step 6, where BDD techniques are used at Step 3 and Step 4. At Step 5, templates for verifying complex pipeline behavior are derived form templates generated at Step 4.

3.1 Coding a pipeline-behavior specification description (Step 1)

Instructions are categorized according to pipeline behavior. **Pipeline-behavior specification descriptions**, which are formal descriptions representing pipeline behavior of microprocessors, are composed of **pipeline definitions** for instruction categories and **explicit stall definitions**. For each instruction category, pipeline stages, specification of data hazard, specification of bypassing, and resource usages are defined as a **pipeline definition**. Stall specifications which cannot be represented in pipeline definitions are written as **explicit stall definitions**.

Figure 1 shows an example of pipeline-behavior specification description for a simple vliw architecture processor, where pipeline behavior of the instruction categories, `alu`, `ldcp`, `calu`, and `cmac`, are defined. Here, `GPR`s, `CPR`s, and `ACC`s correspond to general-purpose registers of the core processor, general-purpose registers of the coprocessor, and accumulators of the coprocessor, respectively.

All instructions which belong to category `alu` go through stages from `Fcore` to `W`, as shown in (1).

$$\mathtt{Fcore} \rightarrow \mathtt{Dcore} \rightarrow \mathtt{E} \rightarrow \mathtt{M} \rightarrow \mathtt{W} \tag{1}$$

Data hazards arise when an instruction depends on the results of a previous instruction [8]. "`Dcore check(GPR)`" in `alu` pipeline definition shows that read after write hazard(RAW) is checked at `Dcore` stage. "`E bypass(GPR)`" in `alu` pipeline definition means that the result of instruction which is written back to `GPR` is bypassed at `E` stage or later.

If some combination of instructions cannot be accommodated because of resource conflicts, the machine is said to have a **structural hazard** [8]. "`X0(MAC)`" and "`X1(MAC)`" in the pipeline definition of `cmac` show that resource `MAC` is used at both `X0` and `X1` stage. If two instructions which both belong to `cmac` category are issued successively, then the second instruction stalls at `T` stage to resolve structural hazard caused by `MAC` (Figure 2).

"`stall :  M -> T`" represents that pipeline-stall at `M` stage causes stall at `T` stage.

```
cmac |Fcop|Dcop| T  | X0 | X1 |
cmac      |Fcop|Dcop| Ts | T  | X1 |
```

Figure 2: Pipeline behavior of cmac → cmac

Pipeline-behavior specification descriptions are used to generate test cases and pipeline behavior models.

3.2 Test case generation (Step 2)

This section describes how to generate test cases from a pipeline-behavior specification.

The following test cases are automatically generated from pipeline-behavior specifications : (1) data hazard, (2) structural hazard, and (3) combination of data or structural hazards. At first, we explain how to generate test cases respecting data hazard from the pipeline-behavior specification description shown in Figure 1. Test cases of data hazard are pipeline states in which two or more instructions access the same register.

At the stage where `GPR` hazard is checked for each instruction category, instruction categories which write to `GPR`s and instruction categories which read `GPR`s are required in order to generate `GPR` hazard test cases.

The following information is drawn from the pipeline-behavior specification shown in Figure 1:

- data hazard of `GPR` is checked at `Dcore` stage
- `alu` and `ldcp` read data from `GPR`
- `alu` assigns data to `GPR`

Test cases of `GPR` hazard generated from the above information are shown in Figure 3. Test case "`((Dcore,alu), (E,alu), GPR)`" shown in Figure 3 represents a pipeline state such that an instruction of category `alu` at stage `Dcore` and an instruction of category `alu` at `E` causes `GPR` hazard. Test cases for `CPR` hazard shown in Figure 4 are also generated in the same way.

```
{((Dcore,alu),(E,alu),GPR), ((Dcore,alu),(M,alu),GPR),
 ((Dcore,alu),(W,alu),GPR), ((Dcore,ldcp),(E,alu),GPR),
 ((Dcore,ldcp),(M,alu),GPR),((Dcore,ldcp),(W,alu),GPR)}
```

Figure 3: Test cases of GPR hazard

```
// core pipes
pileline : alu  = {Fcore(),Dcore() check(GPR),E() bypass(GPR),M(),W()            };
pipeline : ldcp = {Fcore(),Dcore() check(GPR),E(),            M(),W() bypass(CPR)};
// copro pipes
pipeline : calu = {Fcop(),Dcop(),T() check(CPR),X0() bypass(CPR),S()};
pipeline : cmac = {Fcop(),Dcop(),T() check(CPR),X0(MAC),         X1(MAC) bypass(ACC)};
// explicit stall
stall : M -> T;
stall : T -> E;
stall : Dcore -> Dcop;
```

Figure 1: Pipeline-behavior specification

```
{((T,calu),(M,ldcp),CPR),  ((T,calu),(W,ldcp),CPR),
 ((T,calu),(X0,calu),CPR), ((T,calu),(X0,cmac),CPR),
 ((T,calu),(X1,cmac),CPR), ((T,calu),(S,calu),CPR),
 ((T,calu),(S,cmac),CPR),  ((T,calu),(S,cmac),CPR)}
```

Figure 4: Test cases of CPR hazard

Next, we show how to generate test cases respecting structural hazard. Test cases of structural hazard are represented as pipeline states in which two or more instructions access the same resources. These test cases are obtained from the following information: pipeline stage transition and resources used at each stage, which are all described in pipeline-behavior specifications. Test cases shown in Figure 5 are automatically generated from the pipeline-behavior specification in Figure 1.

```
{((T,cmac),(X0,cmac))}
```

Figure 5: Test cases of structural hazard

3.3 Pipeline model generation (Step 3)

This section describes how to generate pipeline behavior models automatically from pipeline-behavior specification descriptions explained in section 3.1.

The models are represented as finite state machines(FSMs) whose inputs are instructions fed into fetch stages at each cycle and states are combinations of instructions in pipeline stages. Suppose s is a fetch stage, the next state of s is represented as expression (2). Otherwise, the next state of s is represented as expression (3), where $ps_i \in Prev(s)$.

$$NextInst(s) = Stall(s)\ \&\ Inst(s)\ |\ \overline{Stall(s)}\ \&\ Inst(IN) \tag{2}$$

$$\begin{aligned} &NextInst(s) = \\ &\quad Stall(s)\ \&\ Inst(s) \\ &|\quad \overline{Stall(s)}\ \&\ \textstyle\sum_{ps_i}(\overline{Stall(ps_i)}\ \&\ Next(ps_i, Inst(ps_i)) \equiv S)\ \&\ Inst(ps_i) \\ &|\quad \overline{Stall(s)}\ \&\ \textstyle\prod_{ps_i}(Stall(ps_i)\ |\ Next(ps_i, Inst(ps_i)) \neq S)\ \&\ Bubble \end{aligned} \tag{3}$$

$Next(s,i)$ is a function which returns the next stage of i, where i is the current instruction at stage s. Function $Prev(s)$ returns a set of stages whose next stage set contains s. These functions are easily derived from pipeline-behavior specifications.

$Stall(s)$, a logic function which returns true if the instruction at stage s stalls at current cycle, can be represented by the logic sum of four functions shown in expression (4). The meaning of each function is described in Table 1.

$$\begin{aligned} Stall(s) \quad = \quad & StructuralHazardStall(s)\ |\ DataHazardStall(s) \\ & |\ ImplicitStall(s)\ |\ ExplicitStall(s) \end{aligned} \tag{4}$$

Let i be an instruction at stage s and R be the set of resources used by i at stage $Next(s,i)$. Logic function $StructuralHazardStall(s)$ returns true if there exists no instruction i' which is issued before i and resource set used by i' at stage $Next(s',i')$, where s' is the current stage of i', does not intersect R. Other functions in Table 1 can be generated from pipeline-behavior specifications.

3.4 Template generation (Step 4)

Sequences of instruction categories with register constraints, called **templates**, are generated in the template generation phase. The procedure of template generation is implemented by the state enumeration technique based on transition relations [7] described below. Let F be an FSM which represents a pipeline behavior model, i be the number of input variables of F, m be the number of state variables of F, and $n = m + i$. The next function of F is represented by m logic functions $f = [f_1, \dots, f_m]$, where $f_i : B^n \to B$. Let $x = (x_1, \dots, x_n)$ be a set of input variables of f, and $y = (y_1, \dots, y_m)$ be a set of output variables of f. Then, the characteristic function of transition relation of f is represented as:

$$F(x,y) = \Pi_{1 \le i \le m}(y_i \equiv f_i(x)).$$

The image of $A \subset B^n$ by f is calculated as follows:

$$f(A)(y) = S_x(F(x,y) \cdot A(x)).$$

where S_x designates the smoothing operation by x [7]. All reachable states from an initial state are obtained by applying the above image calculation iteratively [1].

Figure 6 shows the way to generate sequences of instruction categories. S_0, S_i, $S$and T correspond to initial state, the set of reachable states within i-th cycle, the set of reachable states, and the set of test cases, respectively. A dot represents a state and an arrow represents a state transition caused by some instruction.

A template corresponding to a test case exists if the test case is a member of the reachable state set calculated from the initial state. The template can be generated by tracing the path from the initial state to the test case.

Figure 7 shows an example of a template. "`ldcp assign (cpr0:CPR, gpr0:GPR)`" shows that `ldcp` writes values to both CPR named `cpr0` and GPR named `gpr0`. "`cmac refer (cpr0:CPR)`" means that `cmac` reads GPR named cpr0. The template of Figure 7 is used to verify the case in which two data hazards occur simultaneously.

3.5 Template derivation (Step 5)

Several templates are automatically derived from one template generated in section 3.4 in order to generate test programs

logic function	synopsis
$StructuralHazardStall(s)$	return true if the instruction at stage s is stalled by structural hazard
$DataHazardStall(s)$	return true if the instruction at stage s is stalled by data hazard
$ImplicitHazardStall(s)$	return $Stall(Next(s,i))$, where i is the instruction at stage s
$ExplicitHazardStall(s)$	return true if there exists an explicit stall definition for stage s "`stall :  t -> s`" and $Stall(t)$ is true

Table 1: Stall functions

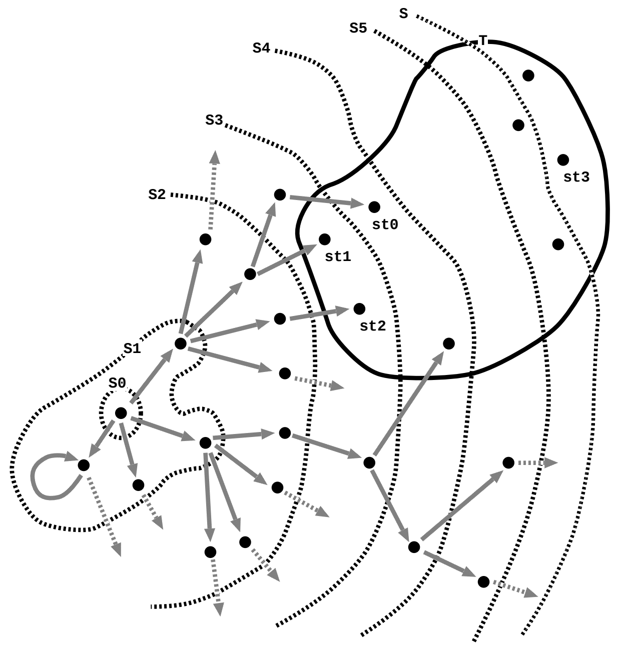

Figure 6: Set of reachable states

```
ldcp assign(cpr0:CPR,gpr0:GPR) + cnop                ;
nop                            + cmac refer(cpr0:CPR);
alu refer(gpr0:GPR)            + calu                ;
```

Figure 7: An example of a template

for verifying the following complex test cases: (1) hazard and branch, (2) hazard and interrupt/exception, and (3) hazard, branch and interrupt/exception.

3.5.1 Hazard and branch

Templates for test cases of hazard and branch are derived from templates generated in section 3.4, inserting a branch instruction just before or among the sequence of instructions. The templates in Figure 8 are derived from the template shown in Figure 7.

3.5.2 Hazard and interrupt/exception

Templates for test cases respecting hazard and interrupt/exception are automatically derived from templates generated in section 3.4 and *3.5.1*. A template is derived from an original template in order to generate interrupt/exception when each instruction which constructs the original template is executed. Interrupt or exception signals are generated by controlling a circuit added for verification by mean of memory-mapped I/O. The outline of the derived template is shown in Figure 9.

```
// branch derivation 1
branch                         + cnop                ;
ldcp assign(cpr0:CPR,gpr0:GPR) + cnop                ;
nop                            + cmac refer(cpr0:CPR);
alu refer(gpr0:GPR)            + calu                ;

// branch derivation 2
ldcp assign(cpr0:CPR,gpr0:GPR) + cnop                ;
branch                         + cmac refer(cpr0:CPR);
alu refer(gpr0:GPR)            + calu                ;
```

Figure 8: After branch derivation

```
         instNum = 3; // the number of instructions
                      // of original template
         base = TBEGIN;
         offset = 0;
LBEGIN : generate an interrupt/exception signal at base + offset;
TBIGIN : ldcp assign(cpr0:CPR,gpr0:GPR) + cnop                ;
         nop                            + cmac refer(cpr0:CPR);
         alu refer(gpr0:GPR)            + calu                ;
         ++offset;
         if (offset < instNum) goto LBEGIN;
```

Figure 9: Interrupt/exception derivation

3.6 Test program instantiation and verification by comparison (Step 6 and Step 7)

In the program generation phase, address, data, branch condition and instruction in templates are instantiated and preprocessing and postprocessing are added.

RTL implementations of the microprocessor are verified by comparing the results of RTL simulation and pipeline simulation by feeding the same test programs into RTL simulator and pipeline level simulator.

4. VERIFICATION RESULTS

We applied this verification system for verifying the RTL implementations of our media-processor, MeP. MeP is a newly developed microprocessor which is designed particularly for multimedia applications [6]. It has 32-bit RISC architecture fundamentally, but is flexible so that users can customize architecture and make it fit their own applications. Verification flow is depicted below:

Step 1 basic verification

Step 2 single instruction verification

Step 3 exception verification

Step 4 complex verification

Step 5 random verification

At first, basic behavior,namely initial values of registers, fetch behavior, basic instructions, etc., was verified using hand-coded test programs.

Next, each instruction was verified using Architecture Verification Programs(AVPs), which are automatically generated by the in-house tool, aVpGen. Basic behavior of interrupt/exception was verified in parallel.

Verification of complex cases began after almost all the bugs detected by AVPs had been fixed. Test programs generated by mVpGen and hand-coded test programs were used in the verification phase of complex behavior. mVpGen automatically generates test programs for pipeline behavior. Test programs for multiple interrupt/exceptions and corner cases of cache behavior are coded by hand.

4.1 Verification method

A pipeline-level simulator implemented in C++ is used as a reference model. RTL implementations of the microprocessor are verified by comparing results of RTL simulation and pipeline simulation by feeding the same test programs into RTL simulator and pipeline-level simulator. The following values are compared:

- register values
- cycle times
- values of program counter at execution stage

4.2 Results

The details of detected bugs are shown in Table 2. Forty-three bugs related to pipeline behavior can be detected by the test program generated by mVpGen; 17 complex pipeline-related bugs among them can be detected by test programs generated from derived templates. Among the bugs detected by random programs, one bug is related to pipeline behavior. About 98% of pipeline-related bugs can been detected by mVpGen.

category	#bugs	test programs
basic	27	hand-coding
single instruction	70	aVpGen
interrupt/exception	39	hand-coding
blocks outside core	25	hand-coding
simple pipeline	26	mVpGen
complex pipeline	17	mVpGen
others	6	random

Table 2: Details of bugs

We describe the bugs detected by the test programs generated from mVpGen.

4.2.1 Bugs respecting simple pipeline behavior

The resolving of data hazard is basic pipeline behavior. There are three ways to resolve data hazard: stall, bypassing, and both stall and bypassing. Bugs related to resolving data hazard such as wrong bypassing, deadlock, and illegal stall are detected by the test programs generated from templates obtained in section 3.4.

4.2.2 Bugs respecting complex pipeline behavior (1)

A branch instruction called `REPEAT` is defined in ISA of MeP. Branch penalty of normal branch instructions of MeP is two cycles. `REPEAT` can iterate instruction sequences called **REPEAT block** with no penalty; therefore, implementation of `REPEAT` is different from those of the other branch instructions. mVpGen derives templates for `REPEAT` as follows. Figure 10 shows templates derived from the template in Figure 7.

```
// repeat derivation 1
       repeat
       ldcp assign(cpr0:CPR,gpr0:GPR) + cnop                ;
       nop                            + cmac refer(cpr0:CPR);
REND : alu refer(gpr0:GPR)            + calu                ;

// repeat derivation 2
       repeat
       nop                            + cmac refer(cpr0:CPR);
       alu refer(gpr0:GPR)            + calu                ;
       ldcp assign(cpr0:CPR,gpr0:GPR) + cnop                ;

// repeat derivation 3
       repeat
       alu refer(gpr0:GPR)            + calu                ;
       ldcp assign(cpr0:CPR,gpr0:GPR) + cnop                ;
       nop                            + cmac refer(cpr0:CPR);
```

Figure 10: After REPEAT derivation

The following bugs were detected by test programs instantiated for the templates mentioned above.

- If the second instruction from the last of a REPEAT block stalls at D stage, then the processor becomes deadlocked.
- If the last instruction of a REPEAT block stalls at D stage, then the processor becomes deadlocked.

4.2.3 Bugs respecting complex pipeline behavior (2)

There are two behavior modes, core operation mode and vliw operation mode in MeP. One core instruction and one coprocessor instruction are executed in parallel in vliw instruction mode. Instructions that transfer data from memory to some `CPR`, **coprocessor load instructions**, and instructions that transfer data from some `CPR` to memory, **coprocessor store instructions**, are defined as core instructions. Also, arithmetic instructions using `CPR`s are defined as coprocessor instructions. Therefore, it is possible for data hazard to occur between a core instruction and a coprocessor instruction.

Bugs are reported respecting which an interrupt occurs when the instruction stalls to resolve the data hazard that occurs between a core instruction and a coprocessor instruction. It is difficult for a human to deal with the test case described above. Our tool can generate test programs for the above test cases by automatically deriving instruction sequences which generate interrupt/exception when stalls are resolved.

4.2.4 Bugs which are hard to be detected

The bug described below is hard to be detected by test programs generated from mVpGen. Coprocessor store instruction `SWCP` which follows coprocessor load instruction `LWCP` does not stall though data hazard occurs between the two instructions. Whether the above bug occurs or not depends on the displacement value of `SWCP`. Figure 11 shows instruction codes of some instructions. Bit patterns `nnnn` and `mmmm` in Figure 11 are the binary expressions of the register number of `(C)Rn` and `(C)Rm`, respectively. Bit pattern `dddd_dddd_dddd_dddd` in Figure 11 is the binary expression of the displacement value of `SWCP`. Suppose `LWCP` and some instruction `Inst` whose instruction code is `code[31:0]` are issued successively. The implementation of RTL design is as follows: if the register number of `LWCP` is equal to that of `Inst` and expression 5 is true, then `Inst` stalls. According to the above implementation, `SWCP` whose displacement value is `16'h0000` stalls, but `SWCP` whose displacement value is `16'h0002` does not stall.

$$\{\texttt{code}[16{:}13],\texttt{code}[3{:}0]\}\,!\!=8'\texttt{hf1}\ \&\&\ \texttt{code}[1]==1'\texttt{b1} \tag{5}$$

```
CMOV  CRn, Rm         1111_nnnn_mmmm_0111 1111_0000_0000_0000
CMOV  Rm, CRn         1111_nnnn_mmmm_0111 1111_0000_0000_0001
CMOVC CCRn, Rm        1111_nnnn_mmmm_0111 1111_0000_0000_0010
CMOVC Rm, CCRn        1111_nnnn_mmmm_0111 1111_0000_0000_0011
SWCP  CRn, disp16(Rm) 1111_nnnn_mmmm_1100 dddd_dddd_dddd_dddd
```

Figure 11: Instruction codes

We intend to improve mVpGen to generate test programs that can detect the bugs described above.

5. CONCLUSIONS

A test program generation tool, mVpGen, has been developed for verifying pipeline behavior of microprocessors. The only inputs mVpGen requires are pipeline-behavior specifications; it automatically generates test cases at first from pipeline-behavior specifications and then automatically generates test programs corresponding to the test cases. As well as test-program generation, verification environment including reference simulation models is also constructed automatically from the pipeline-behavior specification. Unlike conventional test-program generation tools, mVpGen can handle complex test cases, in which data hazard, branch, and interrupt occur simultaneously. mVpGen and the verification environment described in this paper have been adopted for our media-processor design. As a result, about 98% of pipeline-related bugs can be detected by mVpGen, which greatly contributes to improvement of RTL verification efficiency.

6. REFERENCES

[1] Hiroaki Iwashita et al., “Automatic Test Program Generation for Pipeline Processors,” IEEE/ACM International Conference on Computer-Aided Design, pp580-538, 1994.

[2] Richard C. Ho et al., “Architecture Validation for Processors,” Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp404-413, 1995.

[3] R. E. Bryant et al., “Graph Based Algorithms for Boolean Function Manipulation,” IEEE Trans. on Computers, C-35(8):677-691, August 1986.

[4] Ramin Hojati et al., “Early Quantification and Partitioned Transition Relations,” pp12-19, 1996.

[5] Jiro Miyake et al., “Automatic Test Generation for Functional Verification of Microprocessors,” Proceedings of the Third Asian Test Symposium, pp292-297, 1994.

[6] Yoshihisa Kondo et al., “4GOPS 3Way-VLIW Image Recognition Processor based on a Configurable Media-Processor,” ISSCC 2001, pp148-149, 2001.

[7] Abhijit Ghosh et al., “SEQUENTIAL LOGIC TESTING AND VERIFICATION,” Kluwer Academic Publishers, 1992.

[8] David A. Patterson, John L. Hennessy, “Computer Architecture: A Quantitative Approach Second Edition,” Morgan Kaufmann Publishers,Inc., 1996.