設計自動化　68－4
（1993. 10. 28）

パイプライン制御を対象としたテストプログラム自動生成

岩下 洋哲　中田 恒夫　広瀬 文保
（株）富士通研究所　CAD 研究部
〒211 川崎市中原区上小田中 1015

あらまし

マイクロプロセッサの論理検証のための，パイプライン制御機構を対象としたテストプログラムの自動生成手法について述べている．本文では，in-order 実行, out-of-order 完了のスーパースカラプロセッサおよびスカラプロセッサを対象としている．対象とするパイプラインを抽象化して汎用的なモデルを考え，曖昧さのない簡潔な仕様項目を選定した上で，テストプログラム自動生成システムを実現した．生成されるテストプログラムは，プロセッサに様々な種類のパイプラインハザードを引き起こそうとするものである．いくつかのプロセッサに対する実験では，15 秒～50 秒程度 (SS2) でテストプログラムを生成することができた．

和文キーワード　論理検証，テストプログラム，マイクロプロセッサ，パイプライン制御

Automatic Test Program Generation for Pipeline Controllers

Hiroaki Iwashita, Tsuneo Nakata, and Fumiyasu Hirose
Fujitsu Laboratories Ltd.
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

Abstract

A method to generate test programs for verifying microprocessor pipeline controllers is described. It can be applied to superscalar pipelines with in-order execution and out-of-order completion. We defined a general pipeline model for target processors, made a form of pipeline specification, and implemented an automatic test program generation system. Our test program leads pipeline status through various pipeline hazards. Experimental results show that our system generate test programs in about fifteen to fifty seconds for several processors.

英文 key words　Design verification, Test program, Microprocessor, Pipeline controller

1 はじめに

マイクロプロセッサの機構の複雑化は，制御回路の設計誤りを増加させるとともに，その検証をより困難なものにしている．マイクロプロセッサの論理検証では人手で作成したテストプログラムを用いることが多いのが現状であるが，テストプログラムの開発効率と検証の信頼性を高めるために，我々は効果的なテストプログラムを自動生成する手法に関して研究を行なっている．

自然言語で書かれた従来の曖昧な仕様記述からは，テストプログラムを自動生成することが困難である．一方，プログラム言語によりプロセッサの仕様を洩れなく記述することは，設計者の負担の増大と仕様自体の記述誤りの増大という点で問題がある．そこでまず，プロセッサの機構を抽象化して汎用的なモデルを構築することにより，プロセッサの動作を特定するための簡潔な仕様項目を選出する．そして，そのような仕様入力からのテストプログラム自動生成を実現する．

我々はこれまでに，簡単なパイプライン制御を対象としたテストプログラム自動生成システムを開発した[1, 2]．ここでは，この手法の拡張による，より高度なパイプライン制御の取り扱いに関して報告する．

2 パイプラインモデル

2.1 対象とするプロセッサ

我々の従来のシステム[1, 2]では，同種のハードウェアユニットは1つずつしか持たず，どの命令も同じ1本のパイプラインを通るプロセッサのみを考慮していた．すなわち，対象となったプロセッサは

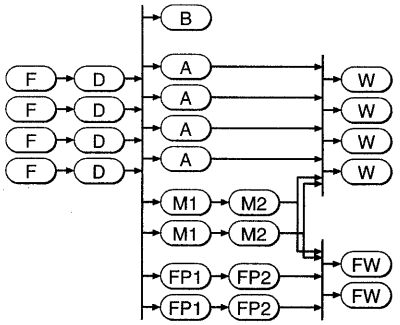
- in-order 実行，in-order 完了のスカラプロセッサ

であった．本報告では，整数演算ユニットと独立して1個以上の浮動小数点演算ユニット等を持つプロセッサや，同一サイクルに複数の命令発行が可能なプロセッサでも取り扱えるようにするためのシステムの拡張について述べる．そこで，本報告で対象とするのは

- in-order 実行，out-of-order 完了のスーパースカラプロセッサ

である．もちろん，in-order 完了のプロセッサやスカラプロセッサも取り扱うことができる．

スーパースカラプロセッサでは，リザベーションステーション等の機構により命令実行順序の入れ換えを行なって out-of-order 実行を実現することが多いが[3]，現段階では out-of-order 実行の機構は取り扱わない．しかしながら in-order 実行用に生成したテストプログラムは，ある程度 out-of-order 実行のプロセッサをテストする効果も持っている．それは，out-of-order 実行のプロセッサでも命令間の競合が起こるまでは in-order で命令実行されるためである．仮想的な in-order 実行のプロセッサ用にテストプログラムを生成して元の out-of-order 実行のプロセッサに適用すれば，in-order 実行の場合と同じ機構のテストあるいは命令実行順序の入れ換え機構のテストができる．



F	命令フェッチ
D	命令デコード，レジスタフェッチ
B	分岐
A	整数／論理演算
M1, M2	メモリアクセス
FP1, FP2	浮動小数点演算
W	整数レジスタ書き込み
FW	浮動小数点レジスタ書き込み

図1　パイプラインの例

2.2 プロセッサのモデル化

パイプライン

ここで扱うプロセッサのパイプラインは，パイプラインを構成するハードウェアユニットの名前でラベル付けされた節点を持ち，ユニット間の命令の流れを有向枝としたグラフで表現することができる．ただし，1つの節点には命令を最大1つしか保持できないものとし，n 個の命令を同時に処理するユニットは並列する n 個の節点で表すものとする．浮動小数点演算ユニット等でよく見られる内部でパイプライン化されたユニットは，複数の節点を直列に並べることで表現できる．このグラフは次のような特徴を持つ．

- ループが存在しない．
- 全ての節点は，命令フェッチユニットに対応する節点から到達可能である．
- 同じラベルを持つ節点はすべて，グラフ上で等価な接続関係にある．

プロセッサのパイプラインの例を図1に示す．これは，4命令同時フェッチのプロセッサの例である．4命令が並列にフェッチ，デコードされた後，それぞれの実行ステージに処理が分かれる．このとき，分岐命令は1個のみ，整数／論理演算命令は4個同時に，メモリアクセス命令と浮動小数点演算はそれぞれ2個まで同時に実行ステージに移ることができる．また，整数レジスタへの書き込みは同時に4命令まで，浮動小数点レジスタへの書き込みは同時に2命令まで実行可能である．

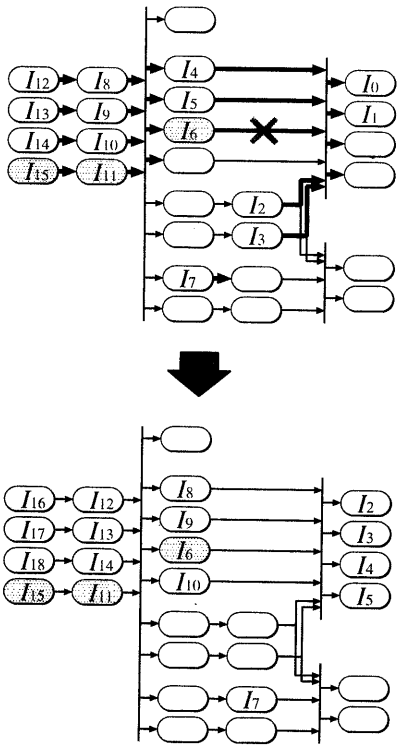


図2　Wユニットの構造ハザード

パイプラインハザード

パイプラインハザードは，命令間の競合により，パイプライン上の命令の流れが部分的に一時停止(ストール)する状態である．パイプラインハザードには次のようなものがある [4].

構造ハザード
n 個しかないハードウェア資源を同時に $n+1$ 個以上使おうとする場合.

データハザード
パイプライン処理によってデータアクセスの順序が入れ替わり，実行結果に矛盾が生じる場合.

制御ハザード
分岐命令に伴うもの.

本文のパイプラインモデルでは，パイプラインのストールは，次の節点に進むべきいくつかの命令が次のサイクルにも同じ節点に留まるという形でモデル化できる.

図1のパイプラインにおいて，Wユニットで構造ハザードが起こる例を図2に示す．この例では，I_2,....I_6 の5つの命令が同時に次のサイクルでWステージに進もうとしている．Wステージは4命令分しか用意されていないので，この状態は構造ハザードとなる．これを解消するために，5つの命令のうちの1つ，ここでは I_6 がAステージでストールする．I_6 のストールに伴って I_{11} および I_{15} もストールする.

命令グループ

現実のプロセッサでは，RISCのものでもかなり多くの種類の命令が用意されている．例えばSPARC-V9[5] の場合，演算の精度やコンディションコードレジスタの取り扱い，アドレシングモード等の命令バリエーションを考慮すれば，命令の種類は数百種にもなる．それらを全て別のものとして取り扱うのは効率的でないため，ここでは，パイプライン処理の観点から命令のグループ化を行なう.

以下の点が全て一致する命令は，パイプライン処理の観点では同一の種類の命令とみなし，同じ命令グループに分類する.

- パイプラインの経路
- 各ステージのレイテンシ
- 各ステージで占有する資源
- 各ステージで読み書きするレジスタ，メモリ

例えば，SPARCのレジスタ-レジスタ間ADD命令とレジスタ-レジスタ間SUB命令は同じ命令グループに分類し，レジスタ-即値間ADD命令は読み出すレジスタの数が異なるため，これらとは別の命令グループに分類する.

オペランド

取り得るオペランド値の全てについて総当たり的に取り扱うことは，現実的な方法ではない．パイプライン処理の観点では，オペランドのレジスタ番号やメモリアドレスの値そのものは重要な意味を持たないため，ある番号のレジスタが特殊な役割を持つような場合を除いて，複数の命令間でオペランドの示す記憶場所が一致するかどうかのみを考慮する.

3　パイプライン仕様記述

テストプログラム自動生成のための入力となるプロセッサのパイプライン仕様記述は，次の要求を満たさなければならない.

- プロセッサのパイプライン動作を特定してテストプログラムの自動生成を可能にするための，十分な情報が含まれていること.
- 設計者が手数をかけず正確に記述できるよう，十分に簡潔であること.

我々は，仕様項目を以下のように選定した．これらの情報を得ることができれば，次節の方法によるテストプログラム自動生成が可能である.

- ハードウェア資源の種類および数

(a) ハードウェア資源の種類および数

占有される資源 (数量)	F	(4)
	D	(4)
	B	(1)
	A	(4)
	M1	(2)
	M2	(2)
	FP1	(2)
	FP2	(2)
	W	(4)
	FW	(2)
データ記憶資源	GPR	
	FPR	
	CC	
	PC	

(b) 命令グループ毎のパイプライン処理

命令グループ	クロックサイクル 0	1	2	3	4
NOP	F PC/R	D			
BR	F PC/R	D CC/R	B PC/W		
ADD	F PC/R	D GPR[RS1]/R GPR[RS2]/R	A	W GPR[RD]/W	
ADDcc	F PC/R	D GPR[RS1]/R GPR[RS2]/R	A	W GPR[RD]/W CC/W	
MOVcc	F PC/R	D GPR[RS2]/R CC/R	A	W GPR[RD]/W	
LD	F PC/R	D GPR[RS1]/R GPR[RS2]/R	M1	M2	W GPR[RD]/W
FLD	F PC/R	D GPR[RS1]/R GPR[RS2]/R	M1	M2	FW FPR[RD]/W
FADD	F PC/R	D FPR[RS1]/R FPR[RS2]/R	F1	F2	FW FPR[RD]/W

図3　パイプライン仕様の例

- パイプライン処理の過程で占有される資源の種類および数
- レジスタ，メモリ等のデータ記憶資源の種類

• 命令グループ毎のパイプライン処理

- 各サイクルに占有する資源
- 各サイクルに読み書きするデータ記憶資源

ハードウェア資源の種類および数については，パイプラインハザードを起こす可能性のあるもののみについて記述すればよい．常に同時に使用されるハードウェアのグループがあれば，グループ全体を1つの資源として定義することができる．

命令グループ毎のパイプライン処理は，ハザードが起こらない場合の理想的なパイプライン処理のタイミングについて記述されていればよい．また，この情報は前節の命令グループ化の基準として述べた4つの項目から導くこともできる．

図1のプロセッサのパイプライン仕様は，図3のようなものになる．ここでは例を簡単にするために，命令グループの一部やメモリアクセスに関する記述等を省略している．GPR, FPR, CC, PC はそれぞれ汎用レジスタ，浮動小数点レジスタ，コンディションコードレジスタ，プログラムカウンタを表しており，RS1, RS2, RD はレジスタアドレスを指定する命令オペランドである．仕様の (b) において，資源名の後の /R および /W はそれぞれその資源のデータを読むことおよび書くことを示し，どちらの表記もない場合はその資源を占有することを示している．

4　テストプログラム自動生成

4.1　ハザード発生ケースの列挙

パイプライン仕様から全てのハザード発生ケースを求める手続きを以下に示す．

(1) 構造ハザード

パイプライン処理の過程で占有される資源の全てついて，その資源を S, 同時に使用可能な S の数を N_S として以下の手続きを繰り返す．

(1-1) パイプライン仕様から，クロックサイクル t に資源 S を占有する命令 (命令グループ) の集合

$$\boldsymbol{I}_S^t = \{I \mid \text{命令 } I \text{ はサイクル } t \text{ に資源 } S \text{ を占有する.}\}$$

を各サイクル $t = 1, 2, \ldots, T$ について作る．ここで，T は仕様に書かれた最も遅いクロックサイクルとする．

(1-2) サイクル t と命令の集合 $\boldsymbol{I}_S^t$ の組のうち，$\boldsymbol{I}_S^t$ が空集合でないものの集合

$$\boldsymbol{X}_S = \{(t, \boldsymbol{I}_S^t) \mid 1 \le t \le T,\ \boldsymbol{I}_S^t \neq \phi\}$$

を作る．これは，命令が資源 S を占有する全てのケースを表している．

(1-3) $\boldsymbol{X}_S$ から重複を許して $N_S + 1$ 個の要素を取り出す組合せを列挙することにより，資源 S で構造ハザードが発生する全てのケース

$$\boldsymbol{H}_S = \{\{X_1, \ldots, X_{N_S+1}\} \mid X_1, \ldots, X_{N_S+1} \in \boldsymbol{X}_S\}$$

を得る．

(2) データハザード

パイプライン処理の過程で読み書きされるレジスタ，メモリ等のデータ記憶資源の全てついて，その資源を D として以下の手続きを繰り返す．

(2-1) パイプライン仕様から，クロックサイクル t で資源 D のデータを読み出す命令 (命令グループ) の集合

$$\boldsymbol{I}^t_{D/R} = \{I \mid \text{命令 } I \text{ はサイクル } t \text{ で } D \text{ のデータを読む.}\}$$

を各サイクル $t = 1, 2, \ldots, T$ について作る．

(2-2) サイクル t と命令の集合 $\boldsymbol{I}^t_{D/R}$ の組のうち，$\boldsymbol{I}^t_{D/R}$ が空集合でないものの集合

$$\boldsymbol{R}_D = \{(t, \boldsymbol{I}^t_{D/R}) \mid 1 \le t \le T,\ \boldsymbol{I}^t_{D/R} \neq o\}$$

を作る．これは，命令が資源 D のデータを読み出す全てのケースを表している．

(2-3) 同様にして，命令が資源 D にデータを書き込む全てのケース

$$\boldsymbol{W}_D = \{(t, \boldsymbol{I}^t_{D/W}) \mid 1 \le t \le T,\ \boldsymbol{I}^t_{D/W} \neq o\}$$

を作る．ここで，$\boldsymbol{I}^t_{D/W}$ はサイクル t で資源 D にデータを書き込む命令 (命令グループ) の集合である．

(2-4) $\boldsymbol{R}_D$ と $\boldsymbol{W}_D$ から要素を 1 個ずつ選ぶ場合および $\boldsymbol{W}_D$ から重複を許して 2 個の要素を選ぶ場合を全て列挙し，資源 D でデータハザードを起こす全てのケース

$$\boldsymbol{H}_D = \{\{R, W\} \mid R \in \boldsymbol{R}_D,\ W \in \boldsymbol{W}_D\} \cup \{\{W_1, W_2\} \mid W_1 \in \boldsymbol{W}_D,\ W_2 \in \boldsymbol{W}_D\}$$

を得る．

制御ハザードは，プログラムカウンタ上のデータハザードとみなせば，上の手続きを適用することができる．

この手続きにより得られるハザード発生ケースは，クロックサイクル t と命令の集合 $\boldsymbol{I}$ の組 $(t, \boldsymbol{I})$ の集合である．$(t, \boldsymbol{I})$ は $\boldsymbol{I}$ に含まれる命令が t サイクルめの処理を実行しようとする状態を示している．

構造ハザードの発生ケース $\{(t_1, \boldsymbol{I}_1), (t_2, \boldsymbol{I}_2), \ldots\} \in \boldsymbol{H}_S$ の場合は，それらが同時に起こる時にハザードとなる．一方，データハザードの発生ケース $\{(t_1, \boldsymbol{I}_1), (t_2, \boldsymbol{I}_2)\} \in \boldsymbol{H}_D$ では，2 つの状態の起こる順序がプログラム順にならない時にハザードとなる．

図 1 のプロセッサがパイプラインの W ユニットで構造ハザードを起こすケースの 1 つを図 4 に示す．

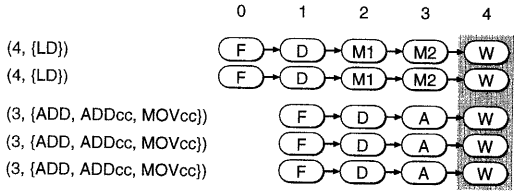


図 4　W ユニットで構造ハザードを起こすケース

4.2　命令列の生成

ここでは，パイプラインが一度もストールすることなく処理が進行した後にハザードが発生するような命令列の生成を考える．

ハザードが発生するクロック時刻を t_h とする．時刻 t_h より前にはストールが起こらないという仮定から，時刻 t_h より前では仕様記述に与えられた通りのタイミングで処理が進行することが保証される．したがって，状態 $(t, \boldsymbol{I})$ を実現するためには時刻 $t_h - t$ に $\boldsymbol{I}$ に含まれる命令をフェッチすれば良いことになる．

プロセッサが 1 サイクルに発行できる命令の数を N，最初の N 個の命令をフェッチする時刻を 0 とすれば，各時刻に対してそれぞれ N 個の命令位置を対応させることができる．

ハザード発生ケースに対応する命令列を生成する手続きを以下に示す．

(1) 構造ハザード

ハザード発生ケース $\{(t_1, \boldsymbol{I}_1), (t_2, \boldsymbol{I}_2), \ldots\} \in \boldsymbol{H}_S$ それぞれについて，以下の手続きを繰り返す．

(1-1) ハザード発生ケースを構成する全ての状態 $(t_k, \boldsymbol{I}_k)$ のうち，最も大きい t_k の値を t_h とする．

(1-2) それぞれの状態 $(t_k, \boldsymbol{I}_k)$ について，以下の手続きを繰り返す．

(1-2-1) $\boldsymbol{I}_k$ の命令フェッチ時刻 $t_f = t_h - t_k$ を求める．

(1-2-2) 時刻 t_f に対応する命令位置から空き位置を 1 つ選んで，その位置に $\boldsymbol{I}_k$ に含まれる命令を 1 つ割り当てる．

(1-3) 命令が割り当てられていない命令位置を NOP 命令で埋める．

(1-4) 完成した 1 セットの命令列を出力する．

(2) データハザード

それぞれのハザード発生ケース $\{(t_1, \boldsymbol{I}_1), (t_2, \boldsymbol{I}_2)\} \in \boldsymbol{H}_D$ (ただし $t_1 \le t_2$) について，以下の手続きを繰り返す．(実行順序が逆になるのはプログラム上の順序が $\boldsymbol{I}_2 \to \boldsymbol{I}_1$ の場合．)

(2-1) $\boldsymbol{I}_1$ を $\boldsymbol{I}_2$ と同時にフェッチし時刻 t_1 にハザードが発生する場合の命令列を，次の手続きにより生成する．

(2-1-1) 時刻 0 に対応する命令位置から命令位置を 2 つ選んで，その位置に $\boldsymbol{I}_1$ および $\boldsymbol{I}_2$ に含まれる命令をこの順で一つずつ割り当てる．

(2-1-2) 命令が割り当てられていない命令位置を NOP 命令で埋める．

(2-1-3) 完成した 1 セットの命令列を出力する．

(2-2) $d = 1, \ldots, t_2 - t_1$ について，$\boldsymbol{I}_1$ を $\boldsymbol{I}_2$ の d サイクル後にフェッチし時刻 $t_1 + d$ にハザードが発生する場合の命令列を，次の手続きにより生成する．

(2-2-1) 時刻 d および 0 に対応する命令位置をそれぞれ 1 つ選び，それらに $\boldsymbol{I}_1$ および $\boldsymbol{I}_2$ に含まれる命令をそれぞれ 1 つずつ割り当てる．

(2-2-2) 命令が割り当てられていない命令位置を NOP 命令で埋める．

(2-2-3) 完成した 1 セットの命令列を出力する．

図 4 のケースでは時刻 0 に LD 命令グループの命令を 2 つ，時刻 1 に ADD, ADDcc, または MOVcc 命令グループの命令を 3 つフェッチするように命令を割り当てれば良い．このプロセッサでは 1 サイクルに 4 命令を同時にフェッチするため，4 つの命令が 1 サイクルに対応する．したがって，時刻 0 の位置には 2 個の LD グループの命令の他に 2 個の NOP 命令を生成する．同様に，時刻 1 の位置には ADD, ADDcc, MOVcc の命令グループの命令の中から 3 つの命令を選び，残りの位置に NOP 命令を 1 つ入れる．時刻 2 から 時刻 4 の位置までは全て NOP 命令で埋める．以上により，例えば次のような 20 個の命令の列を得ることができる．

$$\texttt{LD NOP LD NOP MOVcc ADDcc NOP ADD } \underbrace{\texttt{NOP} \ldots \texttt{NOP}}_{12}$$

(1-2-2) または (2-1-1) で命令位置に空きがない場合は，そのケースに対応する命令列は生成できない．これは，パイプラインが一度もストールすることなく処理が進行した後にそのハザード発生ケースに至るような命令列が存在しない場合である．

命令のオペランドについては，ハザードを起こす資源がアドレス付きの場合は同じアドレスとなるように，その他のオペランド値はそれぞれ異なる値となるように選ぶ．

4.3 無効な命令列の除去

前節では，パイプラインが一度もストールすることなく処理が進行した後にハザード発生ケースに至るという条件の下で命令列の生成を考えたが，生成された命令列が目的のハザードを起こす時刻 t_0 以前に別のハザードを起こさないという保証については触れていない．

時刻 t_0 以前に別のハザードを起こす場合とは，生成した命令列が t_0 以前に別のハザード発生ケースの条件を満たしてしまう場合である．したがって，目的のハザード発生ケースに属する命令列の集合と t_0 以前にハザードを起こすハザード発生ケースに属する命令列の集合の包含関係を調べることにより，目的のハザードを確実に起こす命令列の集合を求めることができる．

図 4 のケースは，CC レジスタのデータハザードを起こすケースの 1 つ {(1,{MOVcc}), (3,{ADDcc})} を満たす可能性がある．それは，時刻 1 でフェッチする 4 つの命令が ADDcc, MOVcc という命令の組合せをこの順序で含んでいる場合である．そのような命令列を実行すると，ADDcc, MOVcc をフェッチしてから 1 クロック後 (時刻 2) に CC

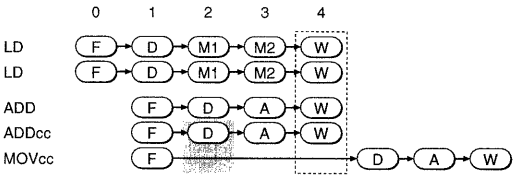


図 5　目的のハザードが発生しない例

レジスタのデータハザードが発生することになり，MOVcc 命令の実行が遅れる．結果として，その後に起こるはずの W ステージの構造ハザードが発生しないことになる．図 5 はこの場合のパイプライン動作を示すものである．ここでは，CC レジスタの演算結果をフォワーディングしない場合を仮定している．

図 4 のケースから ADDcc, MOVcc の命令列および，同様に別のハザードを起こす ADDcc, ADDcc の命令列を除くと，命令列の候補として次の 3 ケースの和集合が残る．なお，ここではこの順序を崩さず命令を生成するものとする．

1. (4,{LD})
   (4,{LD})
   (3,{ADD,MOVcc})
   (3,{ADD,MOVcc})
   (3,{ADD,ADDcc,MOVcc})

2. (4,{LD})
   (4,{LD})
   (3,{ADD,MOVcc})
   (3,{ADDcc})
   (3,{ADD})

3. (4,{LD})
   (4,{LD})
   (3,{ADDcc})
   (3,{ADD})
   (3,{ADD})

図 4 のケースは $1 \times 1 \times 3 \times 3 \times 3 = 27$ 通りの命令列の選び方を表している．これに対し，1. は $1 \times 1 \times 2 \times 2 \times 3 = 12$ 通り，2. は $1 \times 1 \times 2 \times 1 \times 1 = 2$ 通り，3. は $1 \times 1 \times 1 \times 1 \times 1 = 1$ 通りで，合計 15 通りの選び方に削減されたことになる．

5 実験結果

我々は，テストプログラム自動生成システムを約 2,000 行の Perl[6] プログラムで実現した．このシステムは，図 6 のようなパイプライン仕様記述ファイルを入力とする．これは，図 3 で示したパイプライン仕様の例と同じ内容を記述したものである．この仕様記述を入力としてハザード発生パターンを列挙し，無効な命令列を取り除いた結果の一部を図 7 に示す．

SPARC-V9 アーキテクチャに基づいて，次の 3 種類のパイプライン構成で仕様を記述した．

```
#
# example.spec
#

%RESOURCE   GPR[], FPR[], CC, PC

%STAGE      F * 4
%LATENCY    1,1
%FUNCTION
<0>
*:          @R PC

%STAGE      D * 4
%LATENCY    1,1
%FUNCTION
<0>
read_rs1:   @R GPR[RS1]
read_rs2:   @R GPR[RS2]
read_fs1:   @R FPR[RS1]
read_fs2:   @R FPR[RS2]
read_cc:    @R CC

%STAGE      B * 1
%LATENCY    1,1
%FUNCTION
<0>
take_br:    @W PC

%STAGE      A * 4
%LATENCY    1,1

%STAGE      M * 2
%LATENCY    1,2

%STAGE      FP * 2
%LATENCY    1,2

%STAGE      W * 4
%LATENCY    1,1
%FUNCTION
<0>
write_rd:   @W GPR[RD]
write_cc:   @W CC

%STAGE      FW * 2
%LATENCY    1,1
%FUNCTION
<0>
write_fd:   @W FPR[RD]

%PIPELINE
nop_pipe:   F  ->  D
br_pipe:    F  ->  D  ->  B
alu_pipe:   F  ->  D  ->  A  ->  W
mem_pipe:   F  ->  D  ->  M  ->  W
fmem_pipe:  F  ->  D  ->  M  ->  FW
fpu_pipe:   F  ->  D  ->  FP ->  FW

%INSTRUCTION
NOP:        nop_pipe
BR:         br_pipe(read_cc, take_br)
ADD:        alu_pipe(read_rs1, read_rs2, write_rd)
ADDcc:      alu_pipe(read_rs1, read_rs2, write_rd, write_cc)
MOVcc:      alu_pipe(read_cc, read_rs2, write_rd)
LD:         mem_pipe(read_rs1, read_rs2, write_rd)
FLD:        fmem_pipe(read_rs1, read_rs2, write_fd)
FADD:       fpu_pipe(read_fs1, read_fs2, write_fd)
```

図 6　仕様記述ファイル

```
#    NBAAMLFF
#    ORDDODLA
#    P DDV DD
#       cc  D
#       cc
#   (11111111)  <0.0>  @X F
#   (11111111)  <1.0>  @X D
#   (01000000)  <2.0>  @X B
#   (00111000)  <2.0>  @X A
#   (00000110)  <2.0>  @X M
#   (00000001)  <2.0>  @X FP
#   (00111000)  <3.0>  @X W
#   (00000100)  <4.0>  @X W
#   (00000011)  <4.0>  @X FW
#   (00110110)  <1.0>  @R GPR[RS1]
#   (00111110)  <1.0>  @R GPR[RS2]
#   (00111000)  <3.0>  @W GPR[RD]
#   (00000100)  <4.0>  @W GPR[RD]
#   (00000001)  <1.0>  @R FPR[RS1]
#   (00000001)  <1.0>  @R FPR[RS2]
#   (00000011)  <4.0>  @W FPR[RD]
#   (01001000)  <1.0>  @R CC
#   (00010000)  <3.0>  @W CC
#   (11111111)  <0.0>  @R PC
#   (01000000)  <2.0>  @W PC

#1
  Structural hazard on M  (2 x 2 x 2 = 8)
  2 (00000110)
  2 (00000110)
  2 (00000110)

#2
  Structural hazard on FP  (1 x 1 x 1 = 1)
  2 (00000001)
  2 (00000001)
  2 (00000001)

#5-35-35
  Structural hazard on W  (1 x 1 x 2 x 2 x 3 = 12)
  4 (00000100)
  4 (00000100)
  3 (00101000)
  3 (00101000)
  3 (00111000)

#5-35-35-38
  Structural hazard on W  (1 x 1 x 2 x 1 x 1 = 2)
  4 (00000100)
  4 (00000100)
  3 (00101000)
  3 (00010000)
  3 (00100000)

#5-35-35-38-38
  Structural hazard on W  (1 x 1 x 1 x 1 x 1 = 1)
  4 (00000100)
  4 (00000100)
  3 (00010000)
  3 (00100000)
  3 (00100000)

#6-35-35-35
  Structural hazard on W  (1 x 2 x 2 x 2 x 3 = 24)
  4 (00000100)
  3 (00101000)
  3 (00101000)
  3 (00101000)
  3 (00111000)
```

図 7　ハザード発生パターン (一部)

Single

1サイクル1命令発行の単純な5段パイプラインのプロセッサで，全てのユニットのレイテンシを1と仮定した場合．

Normal

1サイクル1命令発行で，一般命令の実行ユニットとは別に浮動小数点演算用のユニット (FPU) を持つプロセッサ．命令により FPU 演算結果のレイテンシが異なるが，FPU は内部でパイプライン化されている．

Superscalar

1サイクル4命令発行で in-order 実行のスーパースカラプロセッサ．命令により実行ステージのレイテンシが異なるが，実行ステージは全てパイプライン化されている．

これらのプロセッサについて，SPARCstation2 上で実験を行なった結果を表1に示す．

表1　実験結果

	Single	Normal	Superscalar
命令グループ数	65	65	65
命令発行数／サイクル	1	1	4
実行ユニットの種類	1	2	4
実行ユニットの総数	1	2	9
仕様記述行数	194	199	211
ハザード発生ケース	34	143	156
命令列の選び方	7174	8292	3054107
テストプログラム長	104	796	2327
実行時間 (sec.)			
ケースの列挙	8.1	8.7	8.5
無効な命令列の除去	0.7	6.4	18.9
命令／オペランド生成	5.9	17.6	21.4
実行時間合計	14.7	32.7	48.8

実験では，ハザード発生ケース1つにつき命令列1種類ずつを生成した．実際の使用では，異なる命令／オペランドを選んで複数の命令列を生成することも考えられる．この場合，テストプログラム長および命令／オペランド生成時間はその数に比例して大きくなる．

6　おわりに

本稿では，パイプライン制御に対するテストプログラム自動生成システムの，in-order 実行，out-of-order 完了のスーパースカラプロセッサへの対応について述べた．

我々は，対象とするパイプラインを抽象化して汎用的なモデルを考えた上で，それに基づいて曖昧さのない簡潔な仕様項目を選定した．テストプログラム自動生成のためには，ハードウェア資源の種類および数と，それぞれの命令が各サイクルで使用する資源の記述を仕様とすればよい．

テストプログラム生成では，まずパイプラインハザードを起こすケースを全て列挙し，次にそれぞれに対応する命令列を生成していく．本稿では，全てのハザード発生ケースを列挙する手続きと，パイプラインが一度もストールすることなく処理が進行した後にハザードが発生するような命令列を生成する手続きについて述べた．このとき，実際には目的のハザードが発生しないような無効な命令列を判定して除去する方法についても述べた．

いくつかのプロセッサについて仕様を記述し，それらについては十分簡潔に記述可能であることを確認した．それらの入力に対してテストプログラム自動生成システムを適用した結果，15秒 ～ 50秒程度 (SS2) でテストプログラムを生成することができた．

今後，ハザード解消の過程で2次的に発生するハザードに対応すること，および，リザベーションステーション等の特殊な機構の一般化を検討して現在のモデルに取り入れることにより本格的な out-of-order 実行のスーパースカラ対応へ発展させることが課題となっている．

参考文献

[1] H. Iwashita, T. Nakata, and F. Hirose, "Behavioral Design and Test Assistance for Pipelined Processors," *First Asian Test Symposium*, pp. 8–13, 1992.

[2] H. Iwashita, T. Nakata, and F. Hirose, "Integrated Design and Test Assistance for Pipeline Controllers," *IEICE Transactions on Information and Systems*, pp. 747–754, vol. E76-D, no. 7, 1993.

[3] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.

[5] SPARC Version 9 Draft 1.0.3, SPARC International, 1992.

[6] L. Wall and R. L. Schwarts, *Programming Perl*, O'Reilly & Associates, 1991.