システムＬＳＩ設計技術　93－10
（1999. 11. 27）

out-of-order完了可能なパイプラインプロセッサのHDL記述生成の一手法

檜垣茂明 †　伊藤 真紀子 †　佐藤 淳 ††　武内 良典 †　今井 正治 †

† 大阪大学 大学院基礎工学研究科　†† 鶴岡工業高等専門学校
Tel: 06-6850-6626　Tel: 0235-25-9086
Fax: 06-6850-6627　Fax: 0235-24-1840

E-mail: peasv@vlsilab.ics.es.osaka-u.ac.jp

あらまし
in-order 発行，out-of-order 完了型パイプライン・プロセッサの HDL 記述を自動生成する手法を提案する．out-of-order 完了型パイプライン・プロセッサは，スカラ型プロセッサであるが，複数の演算器が同時に動作するため，同時に終了する演算器間でパイプライン・ハザードが生じる．提案手法は，このパイプライン・ハザードを自動的に回避する HDL 記述を生成する．また，提案手法により MIPS R3000 の命令セットのサブセットを有する out-of-order 完了型プロセッサ生成し，同様の命令セットを持つ in-order 完了型のプロセッサと比較した結果，最大 20%の実行サイクル数減少による高速化が確認された．

キーワード　プロセッサ記述生成，VHDL，ハードウェア/ソフトウェア協調設計

Proposal of an HDL Generation Method for Pipeline Processors with Out-of-order Completion

Shigeaki Higaki †　Makiko Itoh †　Jun Sato ††　Yoshinori Takeuchi †　Masaharu Imai †

†Osaka University　††Tsuruoka National College of Technology
Tel: 06-6850-6626　Tel: 0235-25-9086
Fax: 06-6850-6627　Fax: 0235-24-1840

E-mail: peasv@vlsilab.ics.es.osaka-u.ac.jp

Abstract
In this paper, we will propose an HDL generation method for pipelined processors with out-of-order completion function. In scalar processor with out-of-order completion function, various pipeline hazards occur because it controls several components concurrently. The generated synthesizable ASIP includes HDL description to avoid these pipeline hazards automatically. The experimental results using a subset of MIPS R3000 instruction set demonstrate that a generated processor with out-of-order completion function achieves more than 20 % reduction in execution cycles compared with a processor only with in-order completion function.

Key words　processor description generation, VHDL, hardware/software co-design

1 はじめに

ASIP(Application Specific Integrated Processor, 特定用途向き集積化プロセッサ) の開発では，応用分野に応じて適切な命令セットやアーキテクチャを決定する必要がある．しかし，多様なプロセッサ・アーキテクチャの中から，短時間で適切な命令セットやアーキテクチャを決定することは困難である．そこで，命令セットの仕様に基づいて特定用途向けプロセッサを生成する研究が行われている．プロセッサ生成に関する研究は，PEAS-I[1], Satsuki[2], MetaCore[3], CASTLE[4] などのパラメタ化されたプロセッサ・コアを用いる方法と，AIDL[5], PDL[6], 文献 [7] などのプロセッサの仕様記述言語を用いる方法に大きく分けることができる．前者は，記述量が少なく，設計工数は小さいがパイプライン・ステージ数の変更や特殊な命令の追加への対応が困難である．後者は前者と比較すると記述量が多く，設計工数が大きいものの，多様なプロセッサ・アーキテクチャに対応することが可能となる．

筆者らは，プロセッサの仕様記述言語を用いたアプローチの１つとして，クロック単位の命令の動作の記述から合成可能なパイプライン・プロセッサの HDL 記述を生成する手法を提案してきた．提案した手法では，命令形式の定義，使用する演算器等のリソースの宣言，命令のクロック単位の動作記述から，データパスと制御部を生成し，論理合成可能な VHDL を記述を生成する．

提案手法では，設計するプロセッサの性能や面積に影響を及ぼすパイプライン・ステージ数，演算器構成，命令セットを容易に変更可能である．また，設計の変更が容易に可能であるため，短期間に多くのプロセッサを生成，評価でき，その評価結果の比較によって，適切な命令セットやアーキテクチャを短期間で決定することができる．

筆者らがこれまで提案した手法 [8] では，割り込み，遅延分岐方式への対応を行ってきた．また，単サイクルで演算を終了する演算器だけでなく，マルチサイクル演算器への対応を行い，使用できる演算器の範囲を広げることで様々な演算器構成を持ったプロセッサの設計を可能にした．しかし，ここでのマルチサイクル演算への対応は，HW インタロックにより in-order 完了を実現するものであり，マルチサイクル演算器を用いる所要サイクル数の多い命令は，その性能を制限されるという問題が発生していた．また，データ・ハザードへの対応は行われていなかった．

本稿では，マルチサイクル演算の性能を引き出す out-of-order 完了型のプロセッサに対応するために，パイプライン・プロセッサ・モデルおよび生成手法を拡張する．また，out-of-order 完了の実現により，in-order 完了の場合と比較して多くのパイプライン・ハザードの発生が予測されるが，生成されるプロセッサの制御部には，次クロックに発生するパイプライン・ハザードを予測し，ハザードを回避する制御信号を含む HDL 記述が生成される．

out-of-order 完了型のプロセッサの自動生成の実現により，設計可能なプロセッサの性能を向上できる．パイプライン・ハザードへの対応によって，より広い範囲のプロセッサを生成可能となる．生成される HDL 記述は論理合成可能であるため，面積や動作周波数を評価することが可能であり，短期間で用途に応じた，より高性能なプロセッサの設計が可能となる．

本稿は次のように構成されている．まず，2 節では，従来のパイプライン・プロセッサのモデルを拡張し，out-of-order 完了を実現する方法について述べる．3 節では，発生するパイプライン・ハザードの予測と回避の方法について述べる．4 節では，out-of-order 完了を実現し，パイプライン・ハザードを回避する制御部の生成，およびプロセッサ・モデル生成について述べる．5 節では，MIPS 社 R3000 の命令セットのサブセットを持つプロセッサを生成し，out-of-order 完了型パイプライン・プロセッサによる効果を確認した．6 節では，まとめと今後の課題を述べる．

2 プロセッサ・モデル

本節では，out-of-order 完了型プロセッサとして生成されるパイプライン・プロセッサのモデルについて述べる．また，提案手法で考慮するパイプライン・ハザードの種類について述べる．

2.1 パイプライン・モデル

提案手法では，生成されるプロセッサのパイプライン・ステージ数や各ステージで実行する処理は，設計者の記述する設計仕様記述により自由に定義可能である．設計者の設計仕様記述に柔軟に対応するために，プロセッサ・モデルは図 1 に示すパイプライン・ステージの集合で構成される．1 パイプライン・ステージは，そのステージで扱うリソース “resource” やパイプライン・レジスタ “pipeline register” の有向枝による接続から構成されるデータパスと，ステージ制御部 “ctrl” から

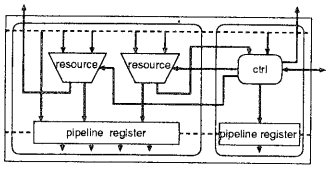

図 1: パイプライン・ステージ

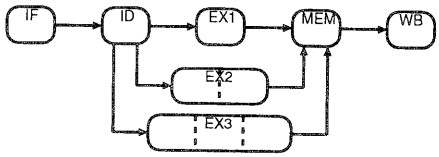

図 2: out-of-order 完了型プロセッサのステージ構成例

構成されている．プロセッサ生成では，必要なステージ数のパイプライン・ステージを連結する事で，設計者の意図するステージ数をもつプロセッサの生成を行う．なお，最終ステージを構成するパイプライン・ステージについては，パイプライン・レジスタを含まない．

設計仕様記述のクロック単位の命令の動作の記述により，各命令の各ステージにおいての動作が定義される．動作定義は，データの転送，リソースを用いた演算などの処理，演算・データ転送の実行の条件式の 3 つの要素を用いて記述され，各パイプライン・ステージにおける制御部では，命令の種類，条件式に応じて，データの転送やリソースを用いた処理を行う制御信号を出力する．以上のように制御を実現されたパイプライン・ステージのモデルは，各サイクルで伝搬された有効な命令を処理する．ステージ制御部では，次ステージへの命令の伝搬についての制御も行う．

提案手法では，パイプライン・ステージの連結によりパイプライン・プロセッサの生成を行う．out-of-order 完了型のパイプライン・プロセッサでは，マルチサイクル演算を行うために，in-order 発行であっても演算は並列に実行される．上記のパイプライン・ステージは 1 ステージで 1 命令を実行するので，並列に行われる可能性がある演算器については，その演算器を含むステージを，演算実行のステージから分割し新たに並列に設けることで，演算器を並列に動作させることを可能にする．図 2 に 5 ステージの in-order 発行，out-of-order 完了型パイプライン・プロセッサの構成例を示す．図 2 では，演算に 2 サイクル必要な乗算器と演算に 3 サイクル必要な除算器をそれぞれ単サイクルの演算器のあるステージ (EX1) から切り離し，それぞれ乗算ステージ (EX2) と除算ステージ (EX3) を構成している．本稿では，命令デコードを行うステージ (以下，ID ステージ) より後にあるマルチサイクル演算器についての，out-of-order 完了の実現を考慮する．

2.2 ステージ制御

各パイプライン・ステージにおける命令実行制御方式について述べる．ステージの実行制御は，ステージ st の状態を表す $valid(st)$ 信号，ステージ st からステージ st' への遷移に関してインタロックが発生していることを表す $lock(st, st')$ 信号，ステージ st からステージ st' への遷移を表す $go(st, st')$ 信号，の 3 種類の信号を用いて行うことが可能である [9]．各信号の制御論理の詳細は 4.2 節で述べる．

- $valid(st)$:
  ステージ st で処理中の命令が有効である場合 1 となり，無効な場合 0 となる信号．
- $lock(st, st')$:
  次クロックでのステージ st でステージ st' への遷移に関して，インタロックが発生した場合に 1 となり，発生していない場合に 0 となる信号．$lock(st, st')$ はパイプライン・ハザードの回避のため，ステージの遷移をインタロックする際や，複数の遷移先をもつステージにおいて，次クロックの遷移先でないステージへの遷移に対してインタロックする際などに用いられる．
- $go(st, st')$:
  次クロックにおいてステージ st からステージ st' に遷移する場合に 1 となり，遷移しない場合に 0 となる信号．

また，ステージ制御は，以上の制御信号と，各ステージで処理中の命令から行われる．ステージ制御に関する集合を以下のように定義する．

ST	:	ステージの集合
ST_m	:	マルチサイクル演算を行うステージの集合
I_{st}	:	ステージ st で処理される命令の集合
V	:	$valid$ 信号の集合
L	:	$lock$ 信号の集合
G	:	go 信号の集合

集合 L および G は，ステージ st からステージ st' への遷移の可能性のあるものについて，$lock(st, st')$, $go(st, st')$ のみを含むものとする．図2の例では，各集合は，以下のように定義される．

$$ST = \{IF, ID, EX1, EX2, EX3, MEM, WB\}$$
$$ST_m = \{EX2, EX3\}$$
$$I_{EX2} = \{mul(\text{乗算命令})\}$$
$$I_{EX3} = \{div(\text{除算命令})\}$$
$$V = \{valid(IF), valid(ID), ...\}$$
$$L = \{lock(IF, ID), lock(ID, EX1), ...\}$$
$$G = \{go(IF, ID), go(ID, EX1), ...\}$$

2.3 パイプライン・ハザード

生成されるプロセッサはパイプライン・プロセッサであるため，命令を適切なサイクルで実行できないパイプライン・ハザードが発生する．パイプライン処理の性能を低下させるパイプライン・ハザードは以下の3種類に分類できる [11].

- **構造ハザード** (structural hazard) :
  オーバラップ実行する命令の全組み合わせをハードウェアがサポートしていない場合，資源競合が原因で生じる．
- **データ・ハザード** (data hazard) :
  ある命令の実行がパイプライン内の先行命令の実行結果に依存している場合に生じる．
- **制御ハザード** (control hazard) :
  プログラム・カウンタを変更する分岐命令等をパイプライン処理する際に生じる．

また，データ・ハザードには以下の3種類に分類できる．

- **RAW** ハザード (read after write) :
  先行命令があるレジスタに値を書き込む前に，後続命令が当該レジスタを読み出そうとする．後続命令は誤って古い値を得るハザード．
- **WAR** ハザード (write after read) :
  先行命令があるレジスタに値を読み出す前に，後続命令が当該レジスタを書き込もうとする．先行命令は誤って新しい値を得るハザード．
- **WAW** ハザード (write after write) :
  先行命令があるレジスタに値を書き込む前に，後続命令が当該レジスタを書き込もうとする．最終的には先行命令が書き込んだ結果となるハザード．

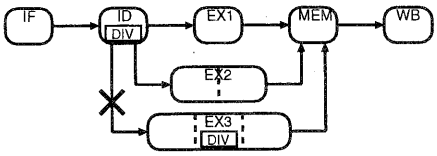

図 3: 構造ハザードの発生例1

制御ハザードはプログラム・カウンタへのデータ・ハザードと考えることができるので，以下では構造ハザードとデータ・ハザードの回避について述べる．提案手法では，パイプライン・ハザードの発生を予測し，回避する制御を実現する．次節では，パイプライン・ハザードの回避方法について述べる．

3 パイプライン・ハザードの予測と回避

提案手法では，発生しうるパイプライン・ハザードの予測を行い，ハザードを回避する制御を行うプロセッサを生成する．以下では，ハザードの予測と回避について述べる．

3.1 構造ハザードの予測と回避

発生する可能性のある構造ハザードは大きく分けて2種類ある．1つはマルチサイクル演算器がパイプライン演算器ではなく，かつ，マルチサイクル演算器の演算中に，別の命令が同じ演算器を使用しようとした場合に発生する．図3では，EX3ステージで除算の演算中に，さらに除算命令が発行され除算器の競合による構造ハザードが発生している．もう1つは複数のステージが同時に同一ステージに遷移しようとした場合に発生する．図4では，EX2ステージの乗算命令とEX3ステージの除算命令が同時にMEMステージにアクセスする事による構造ハザードが発生している．

マルチサイクル演算器へのアクセスの競合による構造ハザードについては，マルチサイクル演算を行うステージを $st1$ とすると，以下の全ての条件を満たす $st2$, $st3$ が存在した場合，次クロックで構造ハザードが発生すると予測される．

- $valid(st1) = 1$
- $go(st1, st2) = 0$ $(go(st1, st2) \in G)$

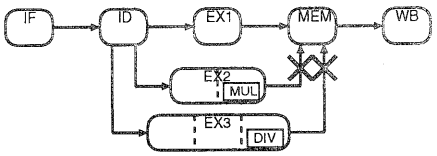

図 4: 構造ハザードの発生例 2

- $go(st3, st1) = 1$ $(go(st3, st1) \in G)$

図 3 の例では，$st1 = \mathrm{EX3}$，$st2 = \mathrm{MEM}$，$st3 = \mathrm{ID}$ と $st2$，$st3$ が存在しているため，構造ハザードの発生が予測される．

この構造ハザードを回避の方法として，提案手法では，ハザードが解消されるマルチサイクル演算の終了まで後続命令の遷移をインタロックするものとする．そこで，$st3$ から $st1$ への遷移をインタロックするため，$lock(st3, st1) = 1$ $(lock(st3, st1) \in L)$ となる．図 3 の例では，
$lock(ID, EX3) = 1$ となる．

また図 3 の例のように，ID ステージからの遷移をインタロックすると命令の発行を停止する．インタロックの対象となる命令の後続の命令に，実行可能な命令が存在するのであれば，インタロックの対象となる命令を一時退避する事で，後続の命令を実行可能となり，より効率の良い命令実行が期待できる．そのため，提案手法では，ID ステージにキューを配置する事で，ハザードの発生のため実行できない命令を一時的に退避する事が可能である．キューへの命令の退避の条件は，ID ステージにインタロックを発生させるハザードの発生とキューに空きがある事である．キューに空きがなければ，キューを配置しない場合と同様にインタロックする．

また，複数のステージが同一ステージに遷移しようとしていることによる構造ハザードについては，複数ステージから同時に遷移されようとしているステージを $st1$ とすると，以下の条件を満たす $st2$ が複数存在した場合，次クロックで構造ハザードが発生すると予測される．

- $go(st2, st1) = 1$ $(go(st2, st1) \in G)$

図 4 では $st1 = \mathrm{MEM}$，$st2 = \mathrm{EX2,EX3}$ と $st2$ が複数存在するので，構造ハザードの発生が予測される．

この構造ハザードの回避の方法として，提案手法では，$st2$ にあたるステージの中から，最も所要サイクルの多いステージを優先して遷移を行い，他のステージからの遷移をインタロックするものとする．そこで，$st2$ の中からもっとも所要サイクルの多いステージでないステージを $st2'$ とすると，$lock(st2', st1) = 1$ $(lock(st2', st1) \in L)$ とする．図 4 の例では，$lock(EX2, MEM) = 1$ とする．

また図 4 の例のように，マルチサイクル演算器の出力値の遷移をインタロックすると，マルチサイクル演算器が値を保持できない場合，出力値を一時的に退避する必要がある．そのため，提案手法では，マルチサイクル演算を行うステージと並列するステージであり，単サイクルのステージでなく，かつ，並列するステージの中で最も所要サイクル数が多いステージでないステージの出力値と命令を格納するキューを配置する．図 2 では，EX2 がそのステージにあたる．キューへの値，命令の退避の条件は，キューが配置されているステージからの遷移にインタロックが発生した場合であり，キューの中に値，命令が退避されている場合は，そのキュー内の値の処理を優先的に行う．

3.2 データ・ハザードの予測と回避

データ・ハザードの発生の予測は，ID ステージの後続の各ステージのパイプラインレジスタ内に含まれる，書き込みレジスタ番号と参照レジスタ番号，および，デコードした命令の書き込みレジスタ番号，参照レジスタ番号の情報から，データ・ハザード予測機構がデータ・ハザードの発生を判断する．ID ステージの後続の各ステージの書き込みレジスタ番号の集合を W_p，参照レジスタ番号の集合を R_p，デコードした命令の書き込みレジスタ番号を w_i，参照レジスタ番号を r_i とすると，各データ・ハザードは，以下の条件を満たす w_p,w_i が存在した場合に発生すると予測することができる．

$$\begin{array}{ll} \mathrm{RAW} & : \quad r_i = w_p (w_p \in W_p) \\ \mathrm{WAW} & : \quad w_i = w_p (w_p \in W_p) \\ \mathrm{WAR} & : \quad w_i = r_p (r_p \in R_p) \end{array}$$

このデータ・ハザードを回避する方法として，提案手法では，ハザードが解消されるまでの間，命令の発行をインタロックするものとする．ハザード予測機構がハザードの発生を予測すると，ID ステージからのステージの遷移をインタロックするため，$lock(st_{id}, st') = 1$ $(lock(st_{id}, st') \in L)$ とする．ここで，st_{id} は ID ステージを表す．

また，データ・ハザードが発生した際も命令の発行を停止する．3.1 節で述べたように，ID ステージにキュー

を配置し，実行可能な後続命令を実行することで，効率の良い命令実行が期待できる．IDステージにキューを配置する場合は，データ・ハザード予測機構はキューの中の命令との依存関係も調べる必要がある．

4 out-of-order完了型プロセッサ生成

本章では，2節で述べたプロセッサ・モデルと3節で述べたパイプライン・ハザード回避方法に基づいて，out-of-order完了型プロセッサを生成する方法について述べる．データパスの生成については，追加するコンポーネントについて述べ，パイプライン・ステージの制御信号については，out-of-order完了を実現する制御論理について述べる．

4.1 データパス生成の拡張

提案手法によるout-of-order完了の実現，および，パイプライン・ハザードの回避のため，データパス部では文献[8]に加えて以下の拡張が必要となる．

1. マルチサイクル演算ステージの生成
2. データ・ハザード予測機構の追加
3. キューの配置

マルチサイクル演算を行うステージの生成は，設計仕様記述のクロック単位の命令の動作により，各ステージにおけるマルチサイクル演算の存在を確認し，マルチサイクル演算ステージの生成を行う．データ・ハザードの予測機構は，生成されたプロセッサのIDステージ以降の各ステージのパイプライン・レジスタから，書き込み・参照レジスタの番号を，IDステージのパイプライン・レジスタから，デコードされた命令の書き込み・参照レジスタの番号を得て，IDステージのステージ制御部にデータ・ハザードの有無を表す信号を生成するものである．キューは，演算終了後に演算結果を保持できないマルチサイクル演算を行うステージで，かつ，並列するステージの中で所要サイクル数が最大でないステージに配置される．また，IDステージで発行できなかった命令を退避するためのキューを配置することも可能である．

4.2 ステージ制御部生成

2.2節で述べた3種類の制御信号を用いて，ステージ制御は行われる．本節では，$valid$, $lock$, go 3種類の制御信号の制御論理について述べる．

ステージ st で処理中の命令が有効となるのは，以下の条件のいずれかが成立した場合である．

- 有効な命令をもっており，かつ，次クロックでステージ st に遷移するステージが存在する場合．
- ステージ st の命令が有効であり，かつ，次クロックにおいてステージ st から後続のいずれのステージにも遷移が発生しない場合．

各ステージの状態を表す信号 $valid$ は次式で求められる．

$$valid(st) = (\bigcup_{s \in ST} valid(s) \cdot go(s, st)) + (valid(st) \cdot \bigcup_{t \in ST} \overline{go(st, t)})$$

ただし，パイプラインの先頭のステージは，割り込みの処理中でなければ，常に命令をフェッチし続け，割り込み処理中であれば，命令はフェッチしない．したがって，パイプラインの先頭のステージを st_{1st} とすると，$valid(st_{1st})$ は，外部割り込みの処理中は0，処理中以外は1となる．

また，次クロックにおいてステージ st からステージ st' に遷移するのは，以下の条件が全て満たされた場合である．

- ステージ st の命令が有効である．
- ステージ st からステージ st' への遷移に関して，インタロックが発生しない．
- ステージ st' の命令が無効，または ステージ st' からその後続のステージ へ命令が遷移する．

したがって，ステージの遷移を表す信号 go は次式で与えられる．

$$go(st, st') = valid(st) \cdot \overline{lock(st, st')} \cdot (\overline{valid(st')} + \bigcup_{s \in ST} go(st', s))$$

ただし，パイプラインの各最終ステージを st_{end} とすると，$go(st_{end})$ は，次式で与えられる．

$$go(st_{end}) = valid(st_{end}) \cdot \overline{lock(st_{end})}$$

パイプラインの遷移のインタロックは，$lock$ 信号を用いて行う．$lock$ 信号は以下の場合に 1 となる信号である．

- パイプライン・ハザードを回避するため，パイプラインの遷移をインタロックする必要がある場合．
- マルチサイクル演算中に演算結果の次ステージへの遷移をインタロックする場合．
- 次クロックにおける遷移先を複数もつステージで処理している命令が，次クロックで遷移しないステージがある場合．

各マルチサイクル演算の実行条件，終了条件を求めることから，各ステージでマルチサイクル演算中を表す信号 $multi_{st}(st \in ST_m)$ を生成する [8]．$multi_{st}$ は，演算中である場合 1 を出力，演算を行っていない場合 0 を返す関数である．また，データ・ハザード予測機構から ID ステージに送られる，データ・ハザード予測信号を $hazard_d$ とする．$hazard_d$ は，ハザードの発生が予測されたとき，1 を出力，予測されない時，0 を出力する．ステージ st の所要サイクル数を返す関数 $count(st)$ を定義すると，構造ハザードの回避，データ・ハザードの回避の手法から，以下の条件のいずれかを満たす時 $lock(st, st')$ は 1 となる．

- $multi_{st} = 1$ の場合
  (マルチサイクル演算中)
- $st = ID$ かつ $hazard_d = 1$ の場合
  (データ・ハザード)
- $st' \in ST_m$ かつ $multi_{st'} = 1$ の場合
  (構造ハザード)
- $go(s, st') = 1$ かつ $count(s) > count(st)$ を満たす $s(s \in ST)$ が存在する場合
  (構造ハザード)
- $st = ID$ かつ $go(ID, s) \in G - \{go(ID - st')\}$ を満たすステージ s のいずれかについて，$inst \in I_s$ の場合

$inst$ は ID ステージで処理中の命令である．

また，キューが配置されたステージについては，$lock$ 信号の発生により，遷移がインタロックされた場合に，キューに値，命令を退避する．

表 1: 実験結果

	プロセッサ A	プロセッサ B	減少率
8 次 FIR	1317	1225	7%
16 次 FIR	2653	2416	9%
16 点 FFT	1437	1311	9%
8 点 IDCT	5639	4506	20%
8 点 DCT	5483	4393	20%

(単位:Cycle)

表 2: 生成されたプロセッサの諸元

	プロセッサ A	プロセッサ B
面積	61211	62765
動作周波数	47.6	43.3

(単位は，面積:Gate，動作周波数:MHz)
(VLSI テクノロジー社 VSC753d
(CMOS 0.5μm) ライブラリ)

5 実験

提案手法より生成される out-of-order 型パイプライン・プロセッサによる効果を示すため，MIPS 社の R3000[10] 命令セットのサブセット 52 命令をもつプロセッサの HDL 記述を，文献 [8] の手法により生成された in-order 完了型のプロセッサ A と，提案手法により生成された out-of-order 完了型のプロセッサ B の比較により評価を行った．生成されたプロセッサは，どちらも乗算器，除算器として 34 サイクルのマルチサイクル演算器を 1 つずつ含んでいる．

生成された 2 つのプロセッサ A，B それぞれに対し，乗除算を数多く含む FIR フィルタ，FFT，DCT などのアプリケーションプログラムを動作させ，実行サイクル数を比較した．実験の結果を表 1 に示す．また，生成されたプロセッサの HDL 記述を，Synopsys 社の Design Compiler を使用して論理合成を行った結果を表 2 に示す．合成用ライブラリとして VLSI テクノロジー社の VSC753d (0.5μm CMOS) ライブラリを用いた．

out-of-order 完了の効果により，最大 20%の実行サイクル数の削減が達成されることが，表 1 の結果より確認できる．パイプライン・ハザード予測機構等の追加により，若干の面積の増加，動作周波数の低下が表 2 より確認されるが，実行サイクル数の削減により，十分に補えるものと考えられる．

また，提案手法では，ID ステージにキューを配置し

```
MULT  $1,$2        (HI,LO <- $1×$2)
MFLO  $3           ($3    <- LO)
MULT  $4,$5        (HI,LO <- $4×$5)
MFLO  $6           ($6    <- LO)
```

図 5: 依存関係を持つ命令列

て，発行できない命令を一時的に退避することも可能である．今回の実験では，命令列の最適化を十分に行ったため，キューの配置による効果は，プロセッサ B における実行サイクル数と比較して，数クロック減少するにとどまったが，実行命令列が図 5 のようにデータ・ハザードを頻繁に発生するものであれば，キューを配置しない場合の out-of-order 完了型プロセッサの実行サイクル数が大きく増加する一方，キューを配置した場合の out-of-order 完了型プロセッサの実行サイクル数には大きな変化は起こらない．

6 おわりに

本稿では，out-of-order 完了を実現すると同時に，パイプライン・ハザードを回避を考慮したプロセッサ・モデルの HDL 記述生成手法を提案した．

out-of-order 完了の実現により，マルチサイクル演算器がその性能を制限されることがなくなる．評価実験により，様々なアプリケーションにおいてその必要サイクル数の減少が確認できたことから，提案手法による効果が示され，生成されるプロセッサの性能の向上が確認された．

提案手法では，パイプライン・ハザードの回避制御，キューの配置により，どのような命令列にも対応可能となった．ID ステージにキューを配置する事による効果は，アーキテクチャの異なるプロセッサにおいても，すでに存在するバイナリコードを効率を落とすことなく使用できることである．out-of-order 完了型のプロセッサによる実行サイクル数は，in-order 完了型と比較して増加する事はないが，実行する命令列に依存する所が大きい．このキューの存在により，その依存を軽減することも可能であるが，キューの大きさだけの面積が必要であるので，トレードオフが発生する．このキューの有効利用法と合わせて，生成されたプロセッサへの最適なコンパイラ・スケジューラの生成が今後の課題となる．

謝辞

本研究を進めるにあたり貴重なコメントを頂いた，大阪大学今井研究室の諸氏に深謝する．なお，本研究の一部は (株) 半導体理工学研究センターとの共同研究による．

参考文献

[1] Sato,J., Alomary,A.Y., Honma,Y., Nakata,T., Shiomi,A., Hikichi,N. and Imai,M., "PEAS-I:A Hardware/Software Codesign System for ASIP Development", IEICE Trans. Fundamentals, Vol.E77-A, No.3, pp.483-491 1994

[2] Shackleford,B., Yasuda,M., Okushi,E., Koizumi,H., Tomiyama,H. and Yasuura,H., "An Integrated Processor Synthesis and Compiler Generation System", IEICE Trans. Inf.&Syst., Vol.E79-D, No.10, pp.1373-1381 1996

[3] Yang,J.-H., Kim,B.W.,Nam,S.-J., Cho,J.-H., Seo,S.-W., Ryu,C.-H. et al, "MetaCore: An Application Specific DSP Development System", 35th DAC, pp.800-803 1998

[4] Campasano,R. and Wilberg,J., "Embedded System Design", Design Automation for Embedded Systems, Vol.1, No.Nos.1-2 pp.5-50 1996

[5] 森本 貴之, 斎藤 一志, 中村 宏, 朴 泰祐, 中澤 喜三郎, "方式レベル記述言語 AIDL を用いた高性能プロセッサ設計支援" 情処研報 DA 82-9, pp.57-64 1996

[6] 鶴田 三敏, 安部 公輝, "マイクロプロセッサ記述言語 PDL に基づく設計支援システム" 情処研報 DA 82-9, pp.65-72 1996

[7] 濱辺 雅哉, 能勢 敦, 戸川 望, 柳澤 政夫, 大附 辰夫, "パイプラインプロセッサのハードウェア生成記述生成手法" 信学技報 VLD 97-117 pp.33-40 1997

[8] 伊藤 真紀子, 檜垣 茂明, 塩見 彰睦, 佐藤 淳, 武内 良典, 今井 正治, "パイプライン・ハザードを考慮した合成可能なプロセッサの HDL 記述生成手法の提案" DA シンポジウム'99 論文集 pp.201-206 1999

[9] 古渡 聡, 岩下 洋哲, 中田 恒夫, 広瀬 文保, "パイプラインプロセサの制御論理自動合成" 信学技報 VLD 94-41 pp.17-24 1994

[10] Gerry Kane, "mips RISC アーキテクチャ — R2000 / R3000 —" 共立出版 1992

[11] John L. Hennessy and David A. Patterson, "Computer Architecture A Quantitative approach" MORGAN KAUFMANN PUBLISHERS 1990