計算機アーキテクチャ　121－8
設　計　自　動　化　82－8
（1996．12．12）

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

森本 貴之† 斉藤 一志† 中村 宏§
朴 泰祐† 中澤 喜三郎‡

†筑波大学 電子・情報工学系
§東京大学 先端科学技術研究センター
‡電気通信大学 情報工学科

設計目的に合致したより高性能なプロセッサを短期間で効率良く設計するためには，構成方式や制御方式の各種の可能性の適否をシミュレーションによって評価し，その結果を設計に迅速に反映させることが重要である．しかし，一般のハードウェア記述言語は論理合成等のより下位レベルの設計への接続を強く意識し，上流工程での作業を主眼としておらず，方式レベル設計に適しているとは言えない．そこで我々は，これらの要求を満たすために，新しいハードウェア記述言語として方式レベル記述言語AIDLを提案している．

本論文では，設計の初期段階である方式レベルにおいて短期間で効率の良い設計を行なうためのハードウェア記述言語AIDLと，AIDLを用いた設計支援について述べる．

Design Assistance for Advanced Processors Using Hardware Description Language AIDL

Takayuki MORIMOTO† Kazushi SAITO† Hiroshi NAKAMURA§
Taisuke BOKU† Kisaburo NAKAZAWA‡

†Institute of Information Sciences and Electronics, University of Tsukuba
§Research Center for Advanced Science and Technology, University of Tokyo
‡Department of Computer Science, University of Electro-Communications

In order to design advanced processors best suited for design purposes in a short time, designers have to simulate their designs to select best suited architectures and implementations in the wide variety of them and reflect the results to the designs as early as possible. However, conventional hardware description languages (HDLs) are not suited for designs at the very early design stages. This is because the principal purpose of such HDLs is not the assistance of higher level design itself but the bridge to the lower level design. Then, we have proposed a new hardware description language AIDL.

In this paper, we show characteristics of AIDL and the design assistance using AIDL.

1 はじめに

プロセッサの性能が向上し，その規模が大きくなるにつれて設計はますます複雑になってきている．そのため，設計目的に合致したより高性能なプロセッサを短期間で効率良く設計するためには，設計の初期段階で，構成方式や制御方式の各種の可能性の適否をシミュレーションによって評価し，その結果を設計に迅速に反映させることが重要である．特に命令セットアーキテクチャや，パイプライン処理といったような命令実行制御方式を決定する方式レベル設計における検討は非常に重要である．ここで言う方式レベルとは，実際のハードウェアを意識した詳細なデータパス構造および回路の遅延といった詳細な動作タイミングを考慮することなく，命令セットアーキテクチャや命令の制御の設計を行なう設計レベルである．

近年ではVHDL[1]やVerilog-HDL[2]，SFL[3][4]等のハードウェア記述言語を使用したプロセッサ設計が主流となっている．ハードウェア記述言語を用いて設計を行なった場合，各言語に対して用意されているツールを用いて容易に評価を行なうことが可能で，設計仕様の変更も記述を書き換えるだけですむ利点があり，過去の設計資産の再利用も容易である．また，これらの言語では回路の自動合成も可能である．

しかし，命令セットアーキテクチャや処理方式の組合せは幾通りも考えられ，その中から短期間で目的に合致した設計を見つけ出すためには，ハードウェア記述言語は上位の設計レベルである方式レベルにおける検討に対して効果的でなければならない．すなわち，方式レベルにおけるハードウェアの記述が簡潔かつ容易であり，記述の変更も容易でなければならない．

一般によく使用されているVHDLやVerilog-HDLといったハードウェア記述言語の場合，回路の自動合成につなげることを意識しているため，記述しなければならない回路の範囲やレベルが広く，詳細な情報が必要となる．そのため，上位の設計レベルの検討に焦点を絞っても考慮し，記述しなければならない事象が多く，設計仕様変更に伴う記述の変更も多く必要となる．

また，方式レベルの記述から自動合成される回路は必ずしも品質の良いものではない．従って，我々が対象とする高性能プロセッサを設計するためには，まず方式レベルの設計を記述したとしても，その記述をレジスタトランスファレベルあるいはそれより下位レベルの記述に設計者が変更し，その後自動合成する必要がある．しかし，この記述の変更は容易ではない．

そこで，我々は方式レベルに焦点を絞ったハードウェア記述言語AIDLを提案し，AIDLを用いた高性能プロセッサの設計支援を効率良く行なうことを提案する．図1はAIDLを用いた設計フローである．このように我々の設計手法では，方式レベルより下位のレベルにおける設計への接続には，AIDL記述を他の論理合成可能なハードウェア記述言語によるレジスタトランスファレベル記述への変換・合成系を用いることを想定している．本論文ではレジスタトランスファレベルをハードウェアリソースやそれらの接続といった詳細なデータパス，および制御系とデータパス系間の動作のタイミングを考慮した設計レベルと定義する．この変換・合成系を用いることによって，AIDLを用いた際に必要なハードウェア記述言語の変更そのものに要する期間を取り除くことができる．従って，我々の提案する設計手法を用いて上位レベルにおける設計期間を短縮することは，全体の設計期間の短縮につながるものである．

2章では我々の提案する設計支援の方針を示し，3章ではハードウェア記述言語AIDLについて述べる．続く4章では我々の設計手法で用いる変換・合成系について示し，5章ではVHDLとAIDLの設計期間の比較と変換・合成系の実験について述べる．最後の6章ではまとめと今後の課題について述べる．

2 設計支援の方針

上位レベル設計支援の方法として，ハードウェアとソフトウェアの両者を含めたシステム全体の設計を行なう手法が提案されている[5][6]．これらの研究では，与えられた情報を元にシステムがハードウェア記述言語を用いた設計を行なうことができる．しかしながら，生成される設計における自由度はそれほど広いものではない．例えばレジスタファイルのレジスタ数といったようなハードウェア情報などは設計者が自由に選択することができるが，命令パイプライン構成の変更といったようなことはできない．また，文献[5]の設計手法では与えられる応用プログラムの特徴を抽出してハードウェアを生成するため，入力プログラムにかなり依存したハードウェアを生成する．我々の対象と

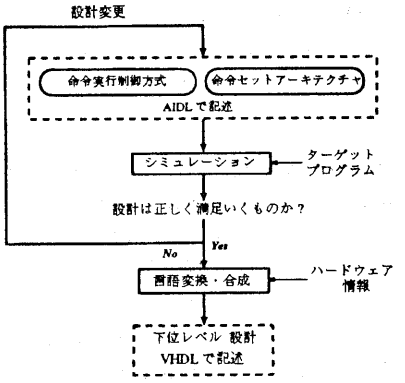


図1: AIDLを用いた設計フロー

する高性能プロセッサ設計においては，命令パイプライン構成の変更といったような，大幅なハードウェアの変更が要求される可能性が非常に高い．さらに，完全に特定のアプリケーションに特化するとも限らない．従って，これらの設計手法は我々の考える高性能プロセッサ設計にはあまり適しているとはいえない．

また，我々の設計方針と同様にレジスタトランスファレベルより上位のレベルでの設計支援を行なう手法も幾つか提案されている [7][8][9]．例えば，文献 [7] はハイレベル合成を行なうものだが，信号処理やネットワークといったものが設計対象であり，我々の対象とする高性能プロセッサに適応するには十分ではない．また，文献 [8] では，ユーザーの記述する，より上位レベルの記述とレジスタトランスファレベル記述のギャップを埋めるための定義記述を用いる方法が示されている．文献 [9] は，従来から使用されているハードウェア記述言語に状態遷移等を付け加えることで，より上位レベルでの設計を行ない，ツールを用いて合成を行なうことで既存の言語のツールを使用可能とする方法を示している．しかしながら，文献 [8] のような方法では，上位レベルの記述と定義記述の両者が必要であり，高性能プロセッサ設計においては定義記述が複雑なものになってしまう可能性も高い．さらに定義記述の書き方によっては，上位レベルの記述変更の影響が定義記述に及ぶことも考えられる．また，文献 [9] のような方法では，元となるハードウェア記述言語自体の持つ制約を受ける可能性がある．

以上に述べた設計手法に対して，我々の提案する手法は，設計対象をプロセッサに，記述レベルを方式レベルに絞り，設計者が詳細な動作タイミングやハードウェアリソース，ハードウェア間の接続といった事象をそれほど考慮することなく，容易に各種の命令セットアーキテクチャや命令実行制御方式を記述可能とする新しいハードウェア記述言語を提案し，この言語を使用することにより設計サイクルの短縮を計るものである．

そのためには上位レベル設計，特に命令セットアーキテクチャや命令実行制御方式等を検討する方式レベルにおける設計で，ハードウェア記述言語に要求されることは大きく分けて以下の 2 点であると考える．

(1) 各種の命令セットアーキテクチャや命令実行制御方式が容易に記述可能であること

(2) 設計仕様の変更を記述に容易に反映可能であること

命令セットアーキテクチャや命令実行制御方式の組合せは幾通りも考えられ，その組合せの中から目的に合致した設計を短期間で効率良く見つけるためには (1) の要求が必要不可欠である．またシミュレーション，その結果に基づく設計仕様変更という設計サイクルが何度となく繰り返されるため，そのターンアラウンドタイムを短くするためには (2) の要求が必要不可欠である．

そこで我々は，従来より方式レベルにおける支援に焦点を当て，多種多様な命令セットアーキテクチャと命令実行制御方式を簡潔に記述することを目的にした方式レベル記述言語 AIDL (Architecture and Implementation Description Language)[10] を提案している．

AIDL は方式レベルにおける設計支援を目的としているため，レジスタトランスファレベル等のより下位の設計レベルには論理合成可能な他のハードウェア記述言語を用いた設計への接続を必要とする．そこで我々は，変換・合成系を用意することでその橋渡しを行なうことを考える．

3 方式レベル記述言語 AIDL

我々の研究目的は，設計目的に最も合致した高性能プロセッサを短期間で効率良く設計することである．そのため，パイプライン制御はもとより data forwarding, delayed branch, multi-functional unit, superscalar, out-of-order completion, out-of-order issue といった各種の命令実行制御方式が容易に記述可能でなければならない [10][11]．従って，2 章でも述べたように使用するハードウェア記述言語は以下の 2 つの要求を満たすことが重要である．

(1) 各種の命令セットアーキテクチャや命令実行制御方式が容易に記述可能であること

(2) 設計仕様の変更を記述に容易に反映可能であること

これら 2 つの要求を満たし，多種多様な命令実行制御方式を正確かつ簡潔に記述するために，AIDL は

- 複数のパイプラインステージの動作間の，並列性や逐次性といった時間関係
- パイプライン処理における命令実行の阻害 (データ依存，制御依存等) が発生した際の処理といった動作間の因果関係

が記述できるようになっている．

以降の 3.1 節と 3.2 節では時間関係と動作の因果関係を表現するための AIDL の特徴について述べ，3.3 節で簡単な AIDL 記述例を示す．

3.1 時間関係

AIDL には時間関係を容易にかつ正確に記述するための 2 つの特徴がある．1 つは “Interval Temporal Logic”[12] に基づいた時間関係の概念である．AIDL はプロセッサの方式レベル設計支援が目的であるため，記述対象は同期回路である．同期回路の動作タイミングを容易に記述可能とするため，取り扱う時間系は時間軸上で定義された離散時間で，動作は離散時間に基づいたインターバルで定義および実行される．

```
stage <stage 名>(起動条件){
  <フラグ設定部処理> //
  block(処理時間 1, 起動条件, 保留時間){
    <本体部処理> } //
  <フラグ設定部処理> //
  block(処理時間 2, 起動条件, 保留時間){
    <本体部処理> } //
                  :
}
```

図 2: stage の構成

もう 1 つの特徴は，動作の並列性と逐次性の明確な定義である．AIDL には複数の動作をまとめる単位として stage があり，これはパイプライン処理のステージに対応する．図2に stage の構成を示す．各 stage は毎クロック起動条件のチェックを行なうが，同時刻に同じ stage は複数起動しない (複数個の同一の stage を同時に起動したい場合には別の stage に分けて記述する)．もし複数の stage がそれぞれ起動条件を満たすと，それらの stage は並列に実行される．1 つの stage はフラグ設定部処理と本体部処理から構成される．操作子 "//" は逐次性を表現し，"A // B" は "A" の処理が終了してから "B" の処理が実行されることを表す．従って，ある stage のフラグ設定部処理と本体部処理は逐次的に実行される．

フラグ設定部処理の実行はδ時間のインターバルで定義されている．全ての stage でこの値は一定であり，フラグ設定部処理の終了は異なる stage であっても同時である．本体部処理の実行は "処理時間" のインターバルで定義される．この "処理時間" は設計者によって与えられる整数値である．フラグ設定部処理と本体部処理の内部は並列に実行され，フラグ設定部処理では stage 制御のための変数の設定，本体部処理では実際のデータ転送が実行される．変数値の設定およびデータの転送は操作子 "<-" によって表現される．"A <- B" は，インターバルの開始時の "B" の値を，インターバルの終了時に "A" に代入することを表す．

3.2 動作間の因果関係

AIDL の変数にはレジスタ型変数，動作間の制御を行なうフラグ型変数，及びこれらの変数の論理組合せからなる中間信号変数がある．

レジスタ型変数はデータパス中のレジスタに相当し，本体部処理でのみ値を代入することができる．また，フラグ型変数は制御回路のラッチに相当し，stage 間の因果関係を表現するのに使用され，"TRUE" または "FALSE" の値をとる．さらに，各フラグ型変数は優先値を持ち，同時に異なる値が代入される場合には指定された優先値の値が代入される．

3.3 AIDL の記述例

AIDL によるパイプライン処理のステージ遷移およびステージの stall と再開の例を図3に示す．

図3-(A) は AIDL による記述で，この記述では stage "decode" から stage "execute" への遷移と，フラグ型変数 datahazard によるパイプラインの stall と再開の処理を行なう部分を示したものである．下線部が各 stage の起動条件を示し，3，13 行目がフラグ設定部処理，5〜10，15〜18 行目が本体部処理である．また，本体部処理のうち，8，15 行目はそれぞれパイプラインステージの stall と再開処理のための条件文である．decode_start，execute_start 及び datahazard は各 stage の起動制御に使用されるフラグ型変数で，counter は datahazard による stage 起動制御のために使用される情報を持つ 4 ビットのレジスタ型変数である．

図3-(B) は図3-(A) の記述の実行例で，変数の値とステージの流れを示す．時刻 t ではフラグ型変数 decode_start は "TRUE"，execute_start は "FALSE"，datahazard は "FALSE" で，4 ビットの register 型変数 counter は "0000" であるとすると，stage "decode" の起動条件のみが成立し，stage "decode" が起動される．起動された stage "decode" ではまずフラグ設定部処理 (3 行目) がおこなわれ，decode_start の値が "FALSE" となる．時刻 t では stage "decode" のみが実行されているので，続いて stage "decode" の本体部処理のみが並列に実行される．その結果，8 行目の条件は満たされず (変数 counter は "0000" で，第 3 ビットは "0")，各変数の値は時刻 t+1 に示される値となる．次に，時刻 t+1 では stage "decode" と同時に stage "execute" も起動され，フラグ設定部処理 (3 行目と 13 行目) と本体部処理 (5〜10 行目と 15〜18 行目) がそれぞれ並列に実行される．また，時刻 t+1 では 8 行目の条件文 (変数 counter の第 3 ビットは "1") が成立し，フラグ型変数 datahazard は "TRUE" となる．時刻 t+2 では起動条件が成立しないため，stage "decode" は起動されない．しかし，時刻 t+2 では stage "execute" の 15 行目の条件文が成立するため 16，17 行目の処理が実行され，その結果時刻 t+3 では stage "decode" が再び起動される．

このように AIDL では時間関係を容易に記述することが可能であり，さらに動作間の因果関係も容易にかつ正確に表現することができる．

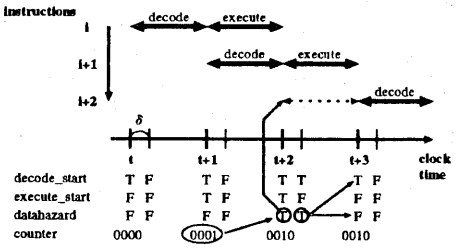
また，この AIDL 記述を入力とするシミュレータがすでに開発されており，このシミュレータを用いることで記述の動作検証ならびに性能評価を行なうことが可能である．

4 変換・合成系

AIDL を用いた設計手法では，方式レベルより下位の設計レベルへ接続するためには，AIDL 記述を他の論理合成可能なハードウェア記述言語の記

```
 1 stage decode(decode_start == TRUE
 2                && datahazard != TRUE){
 3   decode_start <- FALSE;//
 4   block(1,TRUE,1){
 5     execute_start <- TRUE;
 6     decode_start <- TRUE;
 7     counter<0:3> <- counter<0:3> + 'b0001;
 8     if (counter<3> == 'b1) then
 9       datahazard <- TRUE;
10     endif;
11 }}
12 stage execute(execute_start == TRUE){
13   execute_start <- FALSE;//
14   block(1,TRUE,1){
15     if (datahazard == TRUE) then
16       datahazard <- FALSE;
17       decode_start <- TRUE;
18     endif;
19 }}
```

(A): AIDL 記述



(B): 記述 (A) の実行例

図 3: パイプライン記述例

述に合成する必要がある．そこで図1に示されるように，AIDL 記述とデータパス構成といったハードウェア情報を入力とし，他の論理合成可能なハードウェア記述言語によるレジスタトランスファレベル記述を出力とする変換・合成系を用いることを考える．その変換・合成系の出力言語としては VHDL を選択する．

変換・合成系では以下の 4 つの点を考慮する必要がある．(1)〜(3) は AIDL と VHDL のセマンティクスの差異によるものであり，(4) は方式レベルからレジスタトランスファレベルの設計を合成するためのものである．

(1) 明示的なクロックの定義

(2) フラグ設定部と本体部の逐次性

(3) フラグ型変数の同時代入処理

(4) 詳細なデータパス構成

AIDL ではクロックは暗示的であるため，VHDL では (1) に示すように明示的なクロックを表現する必要がある．そこで，VHDL 記述では図 4 のように明示的なクロックを定義する必要がある．この表現では，クロックは "clk" で定義され，各パイプラインステージの動作は "clk" の立ち上がりに同期する．

VHDL で (2) を表現するために，各フラグ型変数を 2 つに分割する．本体部処理で参照され，フラグ設定部処理で代入される変数には従来の変数名の接頭に "flag_" をつける．同様に，フラグ設定部処理で参照され，本体部処理で代入される変数には "data_" をつける．これら 2 つの分割された変数をそれぞれフラグ設定部処理と本体部処理で使用することで，逐次性を表現する．

```
pipeline : process(clk)
begin
  if (clk'event and clk = 1) then
    if (ステージの起動条件) then
      ----- ステージ内処理 -----
    end if;
    if (ステージの起動条件) then
      ----- ステージ内処理 -----
    end if;
      ⋮
  end if;
end process pipeline;
```

図 4: クロックの定義とパイプラインステージ

(3) は AIDL のフラグ型変数の特徴によるものであり，優先回路のついた RS フリップフロップと同様の振舞いを表現する VHDL 記述に変換することで対処することができる．

(4) は方式レベルからレジスタトランスファレベルへの合成を行なうためには，設計者がハードウェアのイメージ情報を与える必要がある．このハードウェアイメージにはリソース数やその配置，I/O 接続といったものがある．

現在，変換・合成系のうち，変換部のプロトタイプが完成している．このプロトタイプではレジスタトランスファレベル記述への合成処理の実装がまだなされておらず，現在の出力は方式レベルの VHDL 記述である．

5 実験

我々の提案する AIDL と AIDL を用いた設計手法は高性能プロセッサの設計期間の短縮を目的としている．しかし，この設計手法では言語の変換に伴う設計の品質の劣化が考えられる．そこで，AIDL と AIDL を用いた設計手法の有効性を明らかにするために，2 つの実験を行なう．1 つは方式レベルにおける設計期間に関する実験で，もう 1 つは設計の品質に関する実験である．

5.1 実験 1 (設計期間の比較)

本実験では AIDL を用いた方式レベル設計に要する期間と，VHDL を用いた方式レベル設計に要する期間を比較し，設計期間の短縮の度合を基に AIDL を用いた設計の有効性の検討を行なう．

比較のための設計対象プロセッサは 3 つで，それぞれ "basic pipeline"，"data forwarding"，"out-of-order completion" と呼ぶ．これらのプロセッサの命令セットアーキテクチャは PA-RISC 1.1[13] をベースとし，23 個の命令を実装する．全てのプロセッサはデータキャッシュ(容量 : 1 KB，ラインサイズ : 32 B，writeback，write allocate) を持つ．

表 1: 設計期間

設計	設計期間 [時間]	
	VHDL	AIDL
basic pipeline	83	18
basic pipeline → data forwarding	24	19
data forwarding → out-of-order completion	49	29

表 2: 論理合成結果

設計	ゲート数	クロックサイクル時間 [ns]
AIDL を用いた設計 (図 7)	127	6.21
VHDL のみを用いた設計 (図 6)	102	5.93

また，命令キャッシュは持たないが，命令フェッチは 1 クロックサイクルで行なえると仮定している．

"basic pipeline" は read after write ハザードが発生すると，解消されるまでパイプラインはストールする単純なパイプライン処理を行なうプロセッサである．"data forwarding" は "basic pipeline" に演算結果およびロードされたデータのデータフォワーディング機能を追加したものである．"out-of-order completion" は "data forwarding" の命令パイプラインを変更し，命令の out-of-order completion を許すように改良したものである．

表 1に設計期間の結果を示す．それぞれの値は，"basic pipeline" の設計に要した時間，"basic pipeline" から "data forwarding" への改良に要した時間，"data forwarding" から "out-of-order completion" への改良に要した時間である．

以上の結果から，方式レベル設計においては AIDL は VHDL と比較して，初期設計に要する時間で 22%程度，改良設計に要する時間でも約 59 ～79%で済むことがわかる．

5.2 実験 2 (設計の品質の比較)

我々の設計手法では一度 AIDL で記述した後，他の論理合成可能なハードウェア記述言語への言語変換およびレジスタトランスファレベルへの合成を行なう必要がある．そのため，2 つの言語を用いることによる設計の品質の劣化が問題となる．この設計の品質の差を調べるため，本実験では

- 高性能プロセッサ設計においては，方式レベル記述からレジスタトランスファレベル等の下位レベル記述への変更は人手で実行
- 現在完成しているプロトタイプの変換・合成系の出力は方式レベルの VHDL 記述

という 2 点をふまえて，AIDL で設計した後にプロトタイプの変換・合成系によって生成された方式レベルの VHDL 記述から合成した論理回路と，VHDL のみを用いて設計した方式レベル記述から合成した論理回路の比較を行なう．5.1 節で設計したプロセッサは，図 1に示すハードウェア情報が設計品質に大きな影響を与えるような設計である．このような設計は，方式レベル記述で合成を行なっても，AIDL からの言語変換に伴う品質劣化の評価には適していない．また，AIDL 記述から VHDL 記述への変換において回路の品質に影響を及ぼす原因は，4 章で述べたように (1)～(3) の 3 つがある．このうち (1) は関係ないので，(2) のフラグ設定部と本体部の逐次性と (3) のフラグ型変数の同時代入処理の 2 つが問題となる．以上のことから，本実験では同時代入処理を含んだ制御系を中心にした簡単な 2 段のパイプライン処理でカウントアップを行なう記述を用いる．しかしながら，この実験結果から得られる傾向は，高性能プロセッサ設計においてもある程度の目安とすることができると考えられる．

図5は AIDL による記述の抜粋で，図6は VHDL のみを用いて行なった記述の抜粋である．図7は図 5 を元に変換を行なった VHDL 記述の抜粋である．図 6と図 7の記述の論理合成を行なった結果を表 2 に示す．論理合成には Synopsys 社の論理合成ツール "design_analyzer" と附属のテスト用ライブラリを用いた．

AIDL を用いた設計は VHDL のみを用いた設計と比較して，ゲート数で約 1.25 倍多く，クロックサイクル時間で約 0.3 ns 長い．これは主に制御論理部分におけるフラグ型変数の同時代入の処理 (図 5と図 7の下線部) に必要な回路によるものである．実際のプロセッサでは，ゲート数やクロックサイクル時間を決める大部分はデータパスによるものであるが，この実験例は簡単なパイプライン処理の制御を記述したもので，プロセッサ設計で要求されるような複雑かつ大規模なデータパスは含まれていない．従って，ゲート数に関してはこの程度の増加は無視することができると考えられる．また，クロックサイクル時間に関しても同様に考えられることができる．確かに，クリティカルパスの部分に同時代入処理を必要とする制御系の信号が含まれていると，クリティカルパスが長くなると考えられる．しかし，付加される遅延時間はデータパス系の部分の遅延時間に比べてかなり短いと考えられる．なぜなら，設計が複雑で，同時代入処理を判断する条件が複雑な場合や同時代入処理を行なう信号が複数箇所で使用されている場合でも，同時代入処理の条件判断そのものは並列に処理さ

れることが可能だからである.

図6のVHDL記述ではprocess文内の記述の順序性を利用することで,この多重代入処理を図7と比べて簡潔に記述できている.これは実験に用いた設計例が高々2段の簡単なパイプラインステージ構成のため,VHDL記述ではprocess文の特徴を容易に利用できたことによるものである.しかしながら,より現実的なプロセッサ設計においては,パイプラインステージ構成や動作間の因果関係はより複雑なものであり,設計者は意図して図7のように記述しなければならない場合も考えられる.

高性能プロセッサの設計においては,たとえVHDLであっても方式レベルからより下位レベルの記述に設計者が変更する必要がある.従って,変換・合成系の出力する記述は,VHDLのみを用いた設計の場合と比べてそれほど遜色なく,高性能プロセッサ設計に十分適応できるものであると考えられる.

6 おわりに

我々は方式レベルに焦点を絞ったハードウェア記述言語AIDL及び,AIDLを用いた高性能プロセッサの設計支援を提案している.

現在,シミュレータとプロトタイプの変換・合成系が完成している.このシミュレータを用いることで設計したプロセッサの動作検証及び性能評価を行なうことができる.また,このプロトタイプは簡単なAIDL記述を方式レベルのVHDL記述に変換することができ,AIDLによる設計をより下位レベルの設計に接続することができる.5.2節の実験で示したように,AIDLからVHDLへ変換された記述はVHDLのみを用いて設計された記述と比較して制御系部分に余計なゲート及び遅延時間が現れる.しかしながら,我々が設計対象とする高性能プロセッサでは,このようなゲート数や遅延時間の差は無視できる程度のものである.従って,AIDLを用いることによる方式レベルの設計期間の短縮は,全体の設計期間の短縮に有効であると言える.また本論文ではAIDLとVHDLを用いて3つのプロセッサの方式レベルにおける設計を行ない,設計期間を比較した.その結果,AIDLはVHDLと比較して,初期設計で約22%,設計改良で約59〜79%の設計期間で済むことがわかった.以上のことから,我々の提案するAIDLとAIDLを用いた設計支援は高性能プロセッサの設計期間の短縮に有効であると言える.

今後の課題としては,さらなるAIDLの有効性の検討のために,より複雑なプロセッサを記述することが挙げられる.また,変換・合成系の改善が挙げられる.図1に示したように変換・合成系は最終的にはレジスタトランスファレベル記述を出力とするが,現在の出力は方式レベル記述である.この変換・合成系の改善によってさらなる設計期間の短縮が図れると考えられる.

謝辞

本研究に関し貴重な御意見を頂いた,筑波大学西川博昭助教授ならびに筑波大学アーキテクチャ研究室グループ諸氏に深く感謝します.

参考文献

[1] R.Lipsett, C.Schaefer and C.Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989

[2] E.Sternheim, R.Singh, and Y.Trivedi, *Digital Design with Verilog HDL*, Automata Publishing Company, 1990

[3] Y.Nakamura, *An Integrated Logic Design Environment Based on Behavioral Description*, IEEE Trans. on CAD, CAD-6, No.3, pp322-336, 1987

[4] 中村行宏, 小野定康, ULSIの効果的な設計法, オーム社.

[5] 中田武治, 佐藤淳, 塩見彰睦, 今井正治, 引地信之, ASIP向きハードウェア/ソフトウェア・コデザインシムテム PEAS-Iにおけるハードウェア生成手法, 情報処理学会研究報告, vol.93, No.111, 93-ARC-103, 93-DA-69, 1993.

[6] 中村秀一, 安浦寛人, ハードウェア/ソフトウェア同時協調設計のためのSoft-Core Processor, 情報処理学会研究報告, vol.93, No.111, 93-ARC-103, 93-DA-69, 1993.

[7] D.W.Knapp, *Behavioral Synthesis : Digital System Design Using the Synopsys Behavioral Compiler*, Prentice Hall, 1996.

[8] 小田原都子, 栗山和則, 関根麻子, Archimedesにおけるメタ記述を用いたHigh Level Simulationのアプローチ, DAシンポジウム, pp93-96, 1993.

[9] F.Vahid, S.Narayan, and D.D.Gajski, *SpecCharts: A VHDL Front-End for Embedded Systems*, IEEE Trans. on Computer-Aided Design, vol.14, No.6, pp694-706, 1995.

[10] H.Nakamura, M.Ito, H.Imori, and K.Nakazawa, *Architecture and Implementation Description Language for Advanced Processor Design*, Proc. of APCCAS'92, pp213-218, 1992

[11] T.Morimoto, K.Yamazaki, H.Nakamura, T.Boku, and K.Nakazawa, *Superscalar Processor Design with Hardware Description Language AIDL*, Proc. of APCHDL'94, pp51-58, 1994.

[12] B.Moszkowski, *A Temporal Logic for Multi-Level Reasoning about Hardware*, Proc. of CHDL'83, 1983.

[13] Hewlett Packard, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual (Third Edition)* 1994.

```
stage fetch(f_start == TRUE){
  f_start <- FALSE;//
  block(1,TRUE,1){
    count <- count + 'b01;
    f_start <- TRUE;
    d_start <- TRUE;
  }
}
stage decode(d_start == TRUE){
   d_start <- FALSE;//
  block(1,TRUE,1){
    if (count == 'b11) then
      f_start <- FALSE;
      hazard <- TRUE;
      A <- A + 'b01;
    endif;
    if (hazard == TRUE) then
      hazard <- FALSE;
      f_start <- TRUE;
    endif;
    if (A == 'b11) then
      all_ready <- FALSE;
    endif;
  }
}
```

図 5: AIDL によるパイプライン記述抜粋

```
process(clk,rst)
begin
if (rst = '1') then
  all_ready <= '1';
  hazard <= '0';
  f_start <= '1';
  d_start <= '0';
  A <= "00";
  count <= "00";
elsif (clk'event and clk = '1') then
  if (all_ready /= '0') then
------ fetch stage ------
    if (f_start = '1') then
      count <= count + "01";
      f_start <= '1';
      d_start <= '1';
    end if;
------ decode stage ------
    if (d_start = '1') then
      if (count = "11") then
        f_start <= '0';
        hazard <= '1';
        A <= A + "01";
      end if;
      if (hazard = '1') then
        hazard <= '0';
        f_start <= '1';
      end if;
      if (A = "11") then
        all_ready <= '0';
      end if;
    end if;
  end if;
end if;
end process;
```

図 6: VHDL によるパイプライン記述抜粋

```
architecture behavior of TEST is
          :
begin
          :
  flag_all_ready <= data_all_ready;
  flag_hazard <= data_hazard;
  flag_f_start <= flag_f(data_f_start
                  ,fetch_exe);
  flag_d_start <= flag_f(data_d_start
                  ,decode_exe);
  data_f_start <= comb_f(true_data_f_start
   ,false_data_f_start,pre_data_f_start);
  process(clk,rst)
  begin
  if (rst = '1') then
    data_all_ready <= '0';
    data_hazard <= '0';
    true_data_f_start <= '0';
    false_data_f_start <= '0';
    pre_data_f_start <= '0';
    data_d_start <= '0';
  elsif (clk'event and clk = '1') then
    data_all_ready <= flag_all_ready;
    data_hazard <= flag_hazard;
    pre_data_f_start <= flag_f_start;
    data_d_start <= flag_d_start;
    true_data_f_start <= '0';
    false_data_f_start <= '0';
------ fetch stage ------
    if (fetch_exe = '1') then
      count <= count + "01";
      true_data_f_start <= '1';
      data_d_start <= '1';
    end if;
------ decode stage ------
    if (decode_exe = '1') then
      if (count = "11") then
        false_data_f_start <= '1';
        data_hazard <= '1';
        A <= A + "01";
      end if;
      if (flag_hazard = '1') then
        data_hazard <= '0';
        true_data_f_start <= '1';
      end if;
      if (A = "11") then
        data_all_ready <= '0';
      end if;
    end if;
  end if;
  end process;
end behavior;
```

図 7: VHDL によるパイプライン記述抜粋 (図 5 を変換)