Vol.2010-ARC-190 No.26
2010/8/4

演算器アレイ型アクセラレータのための命令変換手法の検討

森　浩大[†1]　岩上拓矢[†1]　吉村和浩[†1]
中田尚[†1]　中島康彦[†1]

近年，メニコアプロセッサよりも電力効率が優れたリコンフィギャラブルプロセッサが数多く提案されている．しかし，これらのプロセッサ上で実行されるプログラムには多くの制約があり，各プロセッサに固有のプログラミングモデルと既存プログラムとの非互換性が問題となっている．そこで我々は，リニアアレイパイプラインプロセッサ (LAPP) を提案している．LAPP は GCC がサポートする VLIW 命令列を実行でき，プログラマビリティを備えている．しかし，LAPP の演算器アレイアクセラレータ部分で VLIW 命令列を高速実行するためには，LAPP 特有のいくつかの制約を満たす形に変換しなければならない．本稿では，この命令変換手法の検討を行った．サンプルプログラムに検討手法を適用した結果，命令変換を自動で実現できる見通しを得た．

An Instruction Translation Method for FU Array Accelerator

Kodai Mori,[†1] Takuya Iwakami,[†1]
Kazuhiro Yoshimura,[†1] Takashi Nakada[†1]
and Yasuhiko Nakashima[†1]

Recently, array based accelerators have shown its importance as an alternative low-energy solutions for many core processors. However, current accelerators have their restrictions in programmability, and thus suffer from compatibllity problem of the existing application binaries. We proposed Liner Array Pipeline Processor (LAPP). LAPP is designed to achieve backward compatibility by working on GCC generated VLIW instruction codes. However, there remains several restrictions for the VLIW codes from LAPP's ultra-high speed executions, which requires some translations prior to the program loading. This paper proposed an instruction translation method for LAPP, and verified it with a sample program. The techniques used in the proposed method also give a schematic to generate an automatic translation scheme.

1. はじめに

画像処理や科学技術計算といった膨大な計算能力が要求される場面では，プロセッサの消費電力が問題となっており，性能向上だけでなく低消費電力化が求められている． そこで我々は，低電力 ILP プロセッサの代表格である **VLIW**[1] と，軟らかいハードウェアの代表格である**リコンフィギャラブルデータパス**[2] を融合した演算器アレイ型アクセラレータとしてリニアアレイパイプラインプロセッサ（LAPP）を提案している[3]．

ソフトウェアとの親和性，および，リコンフィギャラブルデータパスによる高速性の両立を目指す仕組みに，ADRES[4),5)] や TRIPS[6)] がある．ADRES は，VLIW エンジンにて既存 VLIW 命令を実行し，アクセラレータエンジンにて専用コンパイラで生成されたループカーネルを高速実行する．TRIPS はデータフローとアレイ構造を直接記述できる EDGE 命令セット[7)] を用いることで，多数の演算器を制御することを目指している．

一方，LAPP は，ADRES や TRIPS のように新規のコンパイラや命令セットを使用せず，GNU Compiler Collection（GCC）が生成する FR-V[8)] の VLIW 命令列をそのまま利用する．演算器アレイの起動には，既存のデータプリフェッチ命令のみを使用しており，**上位互換性**だけでなく，演算器アレイで動作するロードモジュールを従来の VLIW プロセッサ上でも実行できる**下位互換性** も備えている．また，ADRES や TRIPS と異なり，アレイ実行中には，命令キャッシュを含むフロントエンド部分を完全に停止し，演算器アレイのみで VLIW 命令列と等価な演算を実行する．演算器ネットワークについてもアレイ実行中は各演算器の演算の種類と相互接続関係は固定であるので，再構成のコストを最小限に抑えられるとともに，命令が割り当てられた演算器のみに電力を供給すれば良いため，クロックゲーティング等の消費電力削減手法との相性も良い．以上の特徴により，消費電力を劇的に削減し超低電力コンピューティングを実現する．

しかし，演算器アレイで高速実行される VLIW 命令列は，LAPP の実行モデルおよびモジュール構成に起因するいくつかの制約を満たす必要があり，現時点では，GCC が生成する VLIW 命令列をそれらの制約を満たす VLIW 命令列に，プログラマが手動で変換しなければならず，高いプログラマビリティを備えたとは言い難い．

本稿では，演算器アレイで高速実行するために求められる制約を明らかにし，GCC が生

†1 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

Vol.2010-ARC-190 No.26
2010/8/4

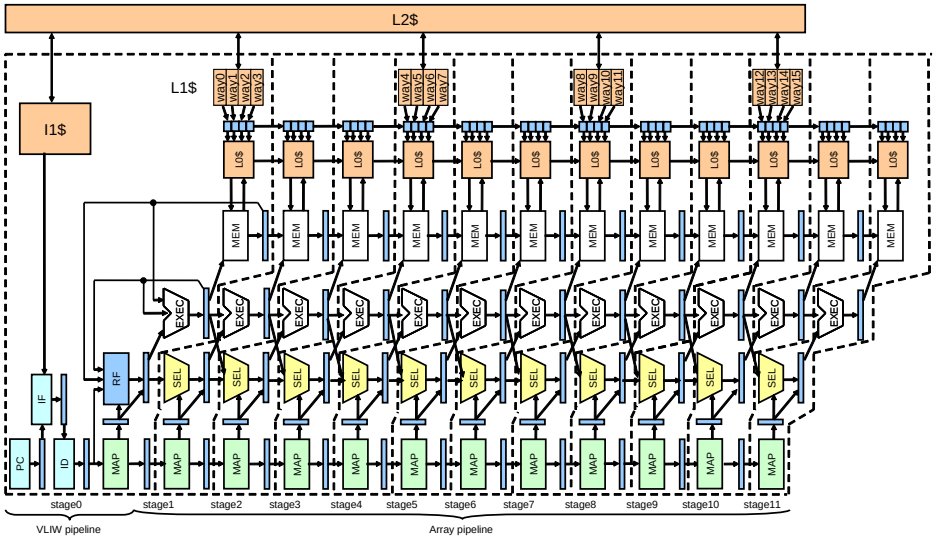

図 1 LAPP のモジュール構成

成した VLIW 命令列からアレイ実行可能な VLIW 命令列への命令変換手法を検討する．この命令変換手法と GCC を組み合わせることにより，アレイ実行可能な VLIW 命令列を自動生成することができ，高いプログラマビリティをプログラマに提供する．以降，2 章では，LAPP の概要とアレイ実行可能な VLIW 命令列が満たすべき制約について述べる．3 章では，サンプルプログラムを用いて，求められる制約を満たす命令変換手法のアルゴリズムを検討する．4 章では，検討した命令変換手法について考察を述べ，5 章では，本稿のまとめを述べる．

2. LAPP

本章では，LAPP によりアレイ実行可能な VLIW 命令列を生成するために，演算器アレイが変換手法に与える制約について明らかにする．まず，LAPP のモジュール構成と実行モデルを説明し，その後，アレイ実行可能な VLIW 命令列を生成するためのプログラミング制約について述べる．

2.1 LAPP のモジュール構成と実行モデル

LAPP のモジュール構成を**図 1** に示す．LAPP の構成は，大きく初段（VLIW pipeline）と段数が可変のアレイ部分（Array pipeline）に分かれている．この例は初段（stage0）に加えてアレイ部分が 11 段（stage1〜stage11）の合計 12 段構成であるが，段数は対象とするプログラムに応じて設計時に決定する．初段は，従来の VLIW プロセッサと同様に，2 次キャッシュ(L2$)，1 次データキャッシュ（L1$），命令キャッシュ（I1$），プログラムカウンタ（PC），命令フェッチユニット（IF），命令デコーダ（ID），レジスタファイル（RF），演算器（EXEC），メモリアクセスユニット（MEM）のパイプライン構造である．そして，演算器やメモリアクセスユニットから演算器およびレジスタファイルへとデータをフォワーディングする演算器バイパスを備えている．一方，アレイ部分は従来プロセッサの演算器ネットワークを線形アレイ状に拡張したパイプライン構造である．異なる段の演算器間は，レジスタファイルを介することなく，パイプラインレジスタを介して接続される．格段は，初段データキャッシュの代わりとして L0 キャッシュ（L0$），初段レジスタファイルの代わりとなる伝播レジスタとセレクタ（SEL）および命令マッパ（MAP）から構成される命令スケジューラ[9] を備えている．L1$は 4way 毎に分散配置されており，それぞれ 1way が読み書き用（way0, way4, way8, way12），3way が読み込み専用（way1-3, way5-7, way9-11, way13-15）となる．way0-3 のみを利用する場合には全段からすべてのデータにアクセスすることが可能であるが，way4-7 を利用する場合には way4-7 が接続されている段以降では，way0-3 のデータにアクセスすることが出来ない．以降，way8-11，way12-15 についても同様である．ただし，書き込みについては常に全段から行える．

LAPP は，通常実行モード，アレイ実行モードおよびアレイ設定モードの 3 つのモードを備える．通常実行モードでは，従来 VLIW プロセッサとして初段のみが動作し，データプリフェッチ命令を契機に，アレイ演算器を起動し，アレイ設定モードに遷移する．アレイ設定モードでは，2 次キャッシュから 1 次データキャッシュへデータのプリフェッチが実行されると同時に，データプリフェッチ命令の次命令からループの先頭に戻る無条件分岐命令までをループカーネルとみなし，ループカーネル内のすべての VLIW 命令列を演算器へ割り当て，演算器間ネットワークを設定する．命令スケジューラは順次流し込まれるループカーネル内の VLIW 命令列を演算器アレイへと写像するとともに，演算器間ネットワークの設定を行う．ループカウンタを更新する命令を演算器に割り当てるには，自身の出力を自身の入力へバイパスする自己ループを設定する．また，アドレス計算に相当する命令を演算器に割り当てるには，自己ループの設定に加えて，先行ロードストア命令のアドレス計算に対してソースオペランドを反映する．命令スケジューリングの詳細については文献 9) を参照されたい．各演算器の命令割り当て，演算器ネットワーク設定およびデータプリフェッチが完

了するとアレイ実行モードに遷移する．アレイ実行モードでは，初段のレジスタファイルと1次データキャッシュから入力データを流し込み，伝播レジスタを介して途中結果を後段の演算器へと伝えていく．演算結果であるストアデータは，最終段から1次データキャッシュへ書き戻され，ループ脱出用の条件付分岐命令の成立を検出すると通常モードへ復帰する．

2.2 プログラム

本節では，2.1節で述べたLAPPのハードウェアによる制約を満たし，アレイ実行可能なVLIW命令を生成するために必要となるプログラミング制約について述べる．本稿の目的はGCCが生成するVLIW命令列からアレイ実行可能なVLIW命令列への変換であるため，ハードウェア資源に関する制約は考慮しない．つまり，ループカーネル内のVLIW命令列を実行するために必要なアレイ段数と伝播レジスタ数はそれぞれ必要とする数よりも多いことを前提とする．

(1) 自己更新命令
2.1節で述べたように，自己ループが設定されるループカウンタやロードストアアドレスを更新する命令を自己更新命令と呼ぶ．アレイ実行モードにおいて，各段は異なるイタレーションを実行しているため，自己更新命令の結果が正しく反映される命令は，ソースオペランドを正しく反映できる当該自己更新命令の先行命令の一部（addi/subi/ldi/ldfi等）と，伝播レジスタを介して演算結果を供給できる当該自己更新命令の後続命令である．

(2) LOAD-USE レイテンシ
一般的にロード命令には算術命令と比較して長いレイテンシが必要であるため，ロード命令の直後でその結果を使うことができない．通常のプロセッサではストール等により常に正しい実行が保証されるが，演算器アレイの入力データは連続的に供給されるため，アレイ実行中にストールできず，不正な実行となる．したがってLOAD-USEレイテンシに応じて，命令間の距離を離しておかなければならない．

(3) 分岐命令
2.1節で述べたように，演算器アレイで実行されるVLIW命令列はループカーネルであり，終端をループ先頭に戻る無条件後方分岐命令にしなければならない．また，アレイ実行モードの終結はループ脱出用の条件付前方分岐命令でなければならない．

(4) 入出力データサイズ
アレイ実行モードでは，プリフェッチによりキャッシュヒットを保障しているため，アレイ実行で利用可能な入出力データサイズは1次データキャッシュのサイズより小さくなければならない．

(5) データプリフェッチ命令
データプリフェッチ命令はループカーネルの先頭を表すだけでなく，アレイ実行モードでの読み出しwayと書き込みwayに対するプリフェッチを指示する．データプリフェッチ命令の詳細については次節で述べる．

(6) ロード命令の配置制限
アレイ実行モードではキャッシュのwayが分散して配置されており，入力データが含まれているwayに応じてVLIW命令列内のロード命令の位置が制限される．

2.3 データプリフェッチ命令

本節ではLAPPのデータプリフェッチ命令（DCPL: Data Cache Pre-Load）について説明する．2.1節で述べたように，キャッシュは複数のwayから構成されており，アレイ実行モードでは，way0/4/8/12が読み書き先，way1-3/5-7/9-11/13-15が読み出し元となる．そのため，アレイ設定モードにて，これらのwayに対してプリフェッチの指示を与えなければならない．しかし，一般的な32bitのデータプリフェッチ命令では，これらの情報を表すにはフィールドが不足するため，プリフェッチ命令を含むVLIW命令内の他命令で各wayに対するプリフェッチを指示する．プリフェッチに必要な情報を以下に示す．

(1) プリフェッチ動作の有無
(2) プリフェッチ対象wayの選択
(3) プリフェッチの先頭アドレスの要素数と方向

図2にDCPL命令を含むVLIW命令の形式を示す．DCPLを含むVLIW命令はDCPLを除き，最大3命令までを実行可能である．DCPLを除いた3命令は，ADDI命令もしくはSUBI命令のみを指定可能とする．“第0命令”の“`#a`”はwayを指示し，`#a`が0，1，2，3であればそれぞれway0-3，way4-7，way8-11，way12-15を示す．“`Rx`”はロードストア先の先頭アドレスを示す．“`Rp`”はプリフェッチ長などの詳細情報が格納されたレジスタである．“`#lk`”は0か1を指定し，0の場合は後続命令を引き続き実行し，1の場合はアレイ実行を開始する．“第1から第3命令”の`ADDI`命令または`SUBI`命令が，“`Ry + #offset`”で先頭アドレスを示し，それぞれ正方向と負方向のプリフェッチを示す．また，“`R0`”はゼロレジスタであり，演算器アレイを持たない既存プロセッサで実行しても副作用が発生しないようにする．

Vol.2010-ARC-190 No.26
2010/8/4

```
第 0 命令      DCPL, #a, Rx, Rp, #lk
第 1〜3 命令   {ADDI, SUBI}, Ry, #offset, R0
```

図 2 DCPL 命令を含む VLIW 命令の構成

3. 命令変換

本章では，GCC が生成する VLIW 命令列をアレイ実行可能な VLIW 命令列に変換するために，命令変換の概要を述べた後，求められる制約を明らかにしながら，前者から後者に変換するために必要なアルゴリズムについて述べる．

3.1 アプローチ

本節では，GCC が生成したアセンブリコードをアレイ実行可能なアセンブリコードに変換するアルゴリズムを検討する．提案アルゴリズムの流れを**図 3** に示す．2.2 節で述べたプログラミング制約に従って書かれた手書きアセンブリコード（Hand-coded ASM source）は演算器アレイで実行できる．しかし，GCC のコンパイラによって最適化され，生成されたアセンブリコード（ASM source without DCPL）はこのプログラミング制約に従っているとは限らず，さらにデータプリフェッチ命令が挿入されていないため，通常実行は可能であるがそのままでは演算器アレイで実行できない．そこで，本命令変換手法は GCC のコンパイラが最適化したアセンブリコードから変換系（Translator）を介して，プログラミング制約に従う形に変換し，アセンブリコードからメモリアクセス情報を抽出してデータプリフェッチ命令を生成・挿入することにより，アレイ実行可能なアセンブリコード（ASM source with DCPL）を出力する．

本手法を適用するサンプルプログラムとして，SPEC ベンチマーク swim の calc1 関数を用いた．swim の calc1 関数のループ処理を**図 4** に示す．calc1 は 3 配列（`P`，`U`，`V`）を入力とし，4 配列（`CU`，`CV`，`Z`，`H`）を出力する関数である．プログラム内の “`FSDX`” および “`FSDY`” は定数であり，“`I`” および “`J`” はそれぞれループ変数である．以降，swim を FR-V 向けクロスコンパイル環境（frv-elf-gcc 4.1.2）を用いて，-O2 オプションでコンパイルしたアセンブリコードを例として用いる．

3.2 アルゴリズム

本節では 2.2 節で述べた (1)〜(5) のプログラミング制約を満たすためのアルゴリズムを提案する．

(1) 自己更新命令

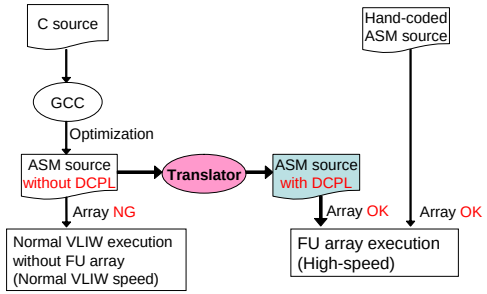

図 3 命令変換の概要

```
CU[J][I+1]  = 0.5 * (P[J][I+1]+P[J][I]) * U[J][I+1];
CV[J+1][I]  = 0.5 * (P[J+1][I]+P[J][I]) * V[J+1][I];
Z[J+1][I+1] = (FSDX * (V[J+1][I+1]-V[J+1][I])-FSDY * (U[J+1][I+1]
              -U[J][I+1])) / (P[J][I]+P[J][I+1]+P[J+1][I+1]+P[J+1][I]);
H[J][I]     = P[J][I]+0.25 * (U[J][I+1] * U[J][I+1]+U[J][I] * U[J][I]
              +V[J+1][I] * V[J+1][I]+V[J][I] * V[J][I]);
```

図 4 swim calc1 関数のループ内

変換例を**図 5** に示す．変換前のコードではアドレスのインクリメントレジスタに初期値 0 をセットし，ループの終端でインクリメントしている．そこで制約 (1) に従い，ベースアドレスのインクメントを検索し，ループ処理の先頭に移動する．また，インクリメントを先頭に移動することでループ中で参照するアドレスが一つずれるため，ループに入る前のループ変数初期化において，一つ前のアドレスから始める必要がある．そこで，“`#0`” を “`#-4`” に書き換える．

(2) LOAD-USE レイテンシ

制約に従い，ループ中のすべてのロード命令に対して直後の命令で同一レジスタを使用している場合はその間に nop を挿入する．

(3) 分岐命令

変換例を**図 6** に示す．変換前のコードはループの終端でループ回数をインクリメントし，回数比較を行い，ループ先頭への条件付後方分岐（`bne`）をしているのがわかる．そこで制約 (3) に従い，ループ回数を数えるインクリメント処理とループ回数

```
        :
    setlos #0, gr6
        :
.LOOP:
    ldf.p  @(gr6,gr16), fr8
    add.p  gr16,gr6,gr10
    add    gr15,gr6,gr11
    ldfi.p @(gr10,4), fr5
    add.p  gr23,gr6,gr12
    add    gr17,gr6,gr10
        :
    addi   gr6,#4,gr6
        :
```

(a) 変換前

```
        :
    setlos #-4, gr6
        :
.LOOP:
    addi   gr6,#4,gr6
    ldf.p  @(gr6,gr16), fr8
    add.p  gr16,gr6,gr10
    add    gr15,gr6,gr11
    ldfi.p @(gr10,4), fr5
    add.p  gr23,gr6,gr12
    add    gr17,gr6,gr10
        :
        :
```

(b) 変換後

図 5 アドレスインクリメントの移動

を比較する演算をループの先頭に挿入する．条件付後方命令（bne）を，ループ脱出用の条件付前方分岐命令（beq）とループ先頭への無条件後方分岐命令（bra）用いたものに変換し，条件付前方分岐（beq）を比較演算（cmp）の後に，無条件後方分岐命令（bra）をループ終端に挿入する．また，この変換を行うことでループ回転回数が 1 回少なくなってしまうために，ループ直前のループ回数初期化において “-1” をする必要がある．

(4) 入出力データサイズ

swim では入出力データサイズが 1 次データキャッシュサイズよりも小さいため，変更は不要である．

(5) データプリフェッチ命令

DCPL 命令を記述するためにはロード命令やストア命令のベースアドレスが必要になる．calc1 に含まれるロードストア命令について**表 1** にまとめる．表中の “P，U，V，CU，CV，H，Z” は配列の先頭アドレス，“gr6” はオフセットである．ループ内のロード命令，ストア命令を検索し，ベースレジスタを決定し，ベースレジスタを最後に更新した命令を辿り．そこから実質的なベースレジスタを決定する．検出したベースレジスタとベースアドレスを基に，前述した DCPL 命令の仕様に従ってデータプリフェッチ命令を生成する．calc1 の例ではロードアドレスが “gr16，gr15，gr17，gr28，gr24，gr19，gr7” となり，ストアアドレスが “gr23，gr22，gr21，gr20”

```
        :
    setlos #0, gr14
        :
.LOOP
    ldf.p  @(gr6,gr16), fr8
    add.p  gr16,gr6,gr10
    add    gr15,gr6,gr11
    ldfi.p @(gr10,4), fr5
        :
    addi   gr14,#1,gr14
        :
    fadds  fr8,fr3,fr8
    stf.p  fr8, @(gr6,gr20)
    cmp    gr25,gr14,icc0
    bne    icc0,2,.LOOP
.LOOP_exit
```

(a) 変換前

```
        :
    setlos #-1, gr14
        :
.LOOP
    addi  gr14,#1,gr14
    cmp   gr25,gr14,icc0
    beq   icc0,0,.LOOP_exit
    ldf.p @(gr6,gr16), fr8
    add.p gr16,gr6,gr10
    add   gr15,gr6,gr11
        :
    fadds fr3,fr0,fr3
    fadds fr8,fr3,fr8
    stf.p fr8, @(gr6,gr20)
    bra   .LOOP
.LOOP_exit
```

(b) 変換後

図 6 分岐命令の変換

```
dcpl.p #0,gr23,gr30,#0;   addi.p gr16,#0,gr0;   addi gr7, #0,gr0
dcpl.p #1,gr22,gr30,#0;   addi.p gr15,#0,gr0;   addi gr19,#0,gr0
dcpl.p #2,gr21,gr30,#0;   addi.p gr18,#0,gr0;   addi gr17,#0,gr0
dcpl   #3,gr20,gr30,#1
```

図 7 swim calc1 のデータプリフェッチ命令

となる．前者が図 2 の “Ry” にあたり，後者が “Rx” にあたる．表 1 を基に作成した calc1 のデータプリフェッチ命令を**図 7** に示す．“.p” が命令のパックを示すため，1 行が 1 つの VLIW 命令に相当する．“dcpl” 命令にストアアドレスを，“addi” 命令にロードアドレスをそれぞれ指定している．“gr30” はプリフェッチ長などの詳細情報を指示するレジスタである．最後の “dcpl” 命令の “#1” がアレイ実行開始を示す．

(6) ロード命令の配置制限

本稿では（5）で生成したデータプリフェッチ命令に従いキャッシュの way を考慮し，ロード命令のスケジューリングを手動で行った．

GCC が出力した swim プログラムのアセンブリに本稿の変換アルゴリズムを手動で適用させたところ，LAPP シミュレータ上でアレイ実行が正しく行われていることを確認した．

表 1 ベースレジスタ

load, store	直前の命令	ベース レジスタ	格納命令	先頭 アドレス
ldf @(gr6,gr16), fr8	---	gr16	set (P), gr16	P
ldfi @(gr10,4), fr5	add gr16, gr6, gr10	gr16	set (P), gr16	P+4
ldfi @(gr11,4), fr6	add gr15, gr6, gr11	gr15	set (U), gr15	U+4
ldf @(gr6,gr17), fr0	---	gr17	set (V+2052), gr17	V+2052
ldf @(gr6,gr18), fr7	add gr17,gr28,gr18	gr17 gr28	set (V+2052), gr17 set (-2052), gr28	V
ldfi @(gr10,4), fr2	add gr17,gr6,gr10	gr17	set (V+2052), gr17	V+2052+4
ldf @(gr6,gr7), fr4	add.p gr16,gr24,gr7	gr16 gr24	set (P), gr16 set (2052), gr24	P+2052
ldfi @(gr4,4), fr1	add gr19,gr6,gr4	gr19	set (U+2052), gr19	U+2052+4
ldfi @(gr11,4), fr0	add gr7,gr6,gr11	gr7	---	P+2052+4
ldf @(gr6,gr15), fr0	---	gr15	set (U), gr15	U
stfi fr0, @(gr12,4)	add gr23,gr6,gr12	gr23	set (CU), gr23	CU+4
stf fr0, @(gr6,gr22)	---	gr22	set (CU+2052), gr22	CV+2052
stfi fr2, @(gr5,4)	add gr21,gr6,gr5	gr21	set (Z+2052), gr21	Z+2052+4
stf fr8, @(gr6,gr20)	---	gr20	set (H), gr20	H

4. 考察・課題

本稿ではアレイ段数を制限せずに命令変換アルゴリズムを検討した．しかし，規模の大きなプログラムを扱う際は，実際のアレイ段数に収まらない場合やデータキャッシュの容量の不足が発生すると予想される．これを解決するためには，ループを分割する必要があるがこれは今後の課題である．また，今回はアルゴリズムを手動で実行したが，自動で行われるプログラムを完成させるのも今後の課題とする．

5. むすび

本稿では，既存プログラムを演算器アレイ型アクセラレータである LAPP 上で，高速実行させるための命令変換手法を検討した．検討したアルゴリズムを，手動でサンプルプログラムに適用したところ，正常なアレイ実行を確認できた．その結果，命令変換を自動で行える見通しを得ることができた．命令変換を自動で行うことで，C プログラムを超高速でアレイ実行可能となり，プログラム互換性の更なる向上が期待できる．今後は，検討したアルゴリズムを自動化するプログラムを構築し，大規模なプログラムに対応できるようにする予定である．

謝　辞

本研究の一部は科学研究費補助金 (若手研究 (B) 課題番号 22700053) による．

参考文献

1) Shiota, T. et al.: A 51.2GOPS, 1.0GB/s-DMA Single-Chip Multi-Processor Integrating Quadruple 8-Way VLIW Processor, *ISSCC*, pp.194–195 (2005).
2) Becker, J. and Hübner, M.: Run-time reconfigurabilility and other future trends, *the 19th annual symposium on Integrated circuits and systems design*, pp. 9–11 (2006).
3) 中田　尚，上利宗久，中島康彦：画像処理向け線形アレイ VLIW プロセッサ，先進的計算基盤システムシンポジウム SACSIS2009，pp.293–300 (2009).
4) Bouwens, F.J. et al.: Architecture Enhancements for the ADRES Coarse-Grained Reconfigurable Array, *HiPEAC'08*, pp.66–81 (2008).
5) Mei, B. et al.: Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study, *DATE*, pp.1224–1229 (2004).
6) Sankaralingam, K. et al.: Distributed Microarchitectural Protocols in the TRIPS Prototype Processor, *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, pp.480–491 (2006).
7) Burger, D. et al.: Scaling to the End of Silicon with EDGE Architectures, *Computer*, Vol.37, No.7, pp.44–55 (2004).
8) 富士通株式会社：*FR550 Series Instruction Set Manual Ver.1.1* (2002).
9) 吉村和浩，上利宗久，中田　尚，中島康彦：演算器アレイ型プロセッサのための命令スケジューラの設計と評価，信学技報 CPSY2009-94，Vol.109, No.474, pp.511–516 (2010).