社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2007－ＳＬＤＭ－132 (34)
2007／11／22

VLIW型プロセッサ用リターゲッタブル・リニアアセンブラ

野垣内 聡† 石浦菜岐佐† 今井 正治††

† 関西学院大学 理工学部 〒669-1337 兵庫県三田市学園２－１
†† 大阪大学 〒565-0871 吹田市山田丘１番５号
E-mail: †{nogaito-s,ishiura}@ksc.kwansei.ac.jp, ††imai@ist.osaka-u.ac.jp

あらまし 本稿では, VLIW型カスタムプロセッサのためのソフトウェア開発ツールとして, リターゲッタブル・リニアアセンブラを提案する. リターゲッタブル・リニアアセンブラは, マイクロアーキテクチャの詳細を意識せずにコーディングができるリニアアセンブリと, ターゲット・プロセッサのアーキテクチャの記述を入力として, そのプロセッサ用に最適化されたアセンブリを出力する. 本稿では, 複数のターゲットに対応させるためのプロセッサアーキテクチャのモデル化, 及びリターゲッタブル・リニアアセンブラのスケジューリングの定式化を示す.
キーワード リターゲッタブル・リニアアセンブラ, VLIW型プロセッサ, コードスケジューリング

Retargetable Linear Assembler for VLIW Processor

Satoshi NOGAITO†, Nagisa ISHIURA†, and Masaharu IMAI††

† Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan
†† Osaka University, Yamadaoka, Suita, Osaka, 565–0871, Japan
E-mail: †{nogaito-s,ishiura}@ksc.kwansei.ac.jp, ††imai@ist.osaka-u.ac.jp

Abstract This paper proposes a *retargetable linear assembler* as a software development tool for custom VLIW processors. The retargetable linear assembler takes a *linear* assembly program, which can be coded without the detailed knowledge of the microarchitecture, and architecture description of a target processor, to generate an assembly code optimized for the processor. We present an architecture model and the formulation of the scheduling problem for the retargetable linear assembler.
Key words retargetable linear assembler, VLIW processor, code scheduling

1. はじめに

近年, ディジタル機器への搭載を目的に, 特定の応用に命令セットやハードウェア構成を最適化したカスタムプロセッサを使用することが多くなっている. 特に複数の命令を並列に実行するVLIW (Very Long Instruction Word) 型プロセッサは, 静的スケジューリングにより, 高速な演算処理や低消費電力を実現するものであり, メディア処理など性能を要求する応用に適していると考えられる.

このようなプロセッサのソフトウェアの効率的な開発には, コンパイラ, アセンブラ, リンカ, シミュレータ等のツールの開発が必須となる. カスタムプロセッサを一つ開発する度に専用のソフトウェア開発ツールを作成するのは効率が悪いため, プロセッサアーキテクチャの記述を基に上記のツールを自動生成するリターゲッタブルな開発環境が種々提案されている [1] [2].

リターゲッタブル・コンパイラ [3] はCなどの高級言語によるソースコードと, ターゲット・アーキテクチャの記述を入力とし, そのプロセッサ用のコードを生成する. Cなどの高級言語をソースコードとすると, コーディングにかかる負担は少なくなる. ただし, 特定用途のカスタム・プロセッサでは, 使用する命令やハードウェア資源が性能に大きく影響するため, 十分な質のコードが出力できない場合がある. 一方, アセンブリ言語では使用したい命令を直接指定でき, 高速なコードを記述しやすいが, VLIWなどの複雑なプロセッサでは命令の並列スケジューリングや, クラスタ分割, 及びデータ転送の挿入が必要になりコーディングが困難となる場合が多い. そのため基本的なシンタックスはアセンブリと同じであるが, 演算器やクラスタなどのハードウェアの詳細を意識せずにコーディングができるリニアアセンブリ [4] はVLIW型プロセッサでのソフトウェア開発に有用であると考えられる.

そこで本稿では, リニアアセンブリとアーキテクチャ記述から最適化されたアセンブリを出力するリターゲッタブル・リニアアセンブラを提案する. リターゲッタブル・リニアアセンブラはアーキテクチャの記述を入力とすることで複数のターゲッ

```
 1:     MVK .S1 50,A2
 2:   ||ZERO.L1 A6
 3:   ||ZERO.L2 B6
 4:L1: LDW.D1 *A3++,A5
 5:   ||LDW.D2 *B3++,B5
 6:     NOP        4
 7:     MPY .M1 A5,B5,A6
 8:   ||MPYH.M2 A5,B5,B6
 9:     ADD .L1 A6,A7,A7
10:   ||ADD .L2 B6,B7,B7
11:     ADD .L1 A7,B7,A4
12:[A2]SUB .L1 A2, 1,A2
13:[A2]  B .S1 L1
```

(a) アセンブリ

```
 1:   MVK  50,i
 2:   ZERO s0
 3:   ZERO s1
 4:L1:LDW  *a++,v1
 5:   LDW  *b++,v2
 6:   MPY  v1,v2,p1
 7:   MPYH v1,v2,p2
 8:   ADD  p1,s0,s0
 9:   ADD  p2,s1,s1
10:   ADD  s0,s1,A4
11:[i]SUB   i, 1, i
12:[i]  B   L1
```

(b) リニアアセンブリ

図 1 TMS320C62x のアセンブリとリニアアセンブリの例

トに対応し, またリニアアセンブリをソースコードとすることで, コーディングの負担を減らしながらも性能の良いコードを出力することを目的とする. 1 つの命令セットアーキテクチャに対して複数のマイクロアーキテクチャを開発する場合に, その差異を吸収してソースコードの移植性を維持するのにも有用である. また, リニアアセンブリを中間表現とすれば, リターゲッタブル・コンパイラのバックエンドとして使用することも出来る. 本稿ではリターゲッタブル・リニアアセンブラが扱うアーキテクチャのモデル, スケジューラの行う処理の定式化を示す.

2. リニアアセンブラ

図 1 (a) は Texas Instrumental 社製の VLIW 型プロセッサ TMS320C62x [7] のアセンブリの例である. TMS320C62x はそれぞれがレジスタファイルと 4 種類の演算器を持つ二つのクラスタから構成されており, アセンブリもそれらを考慮した記述となる. コード中の||はその行の命令を直前の命令と並列に実行する指示であるが, ユーザは命令レベルの並列化を陽に指定しなければならない. 6 行目の NOP は 4, 5 行目のロード命令の完了を待つためのものである. ニーモニックの後の.S1 などは使用する演算器を, オペランドの A2, B6 などはそれぞれクラスタ A, クラスタ B のレジスタを指定するものであり, 命令のクラスタへの割り当てや演算器の選択もユーザが行わなければならない. このようにプロセッサの持つ演算器, クラスタ, パイプライン構成を熟知している必要がある.

このアセンブリに対応するリニアアセンブリが図 1 (b) である. リニアアセンブリは命令が逐次実行されるセマンティクスであるため, ||や NOP を指定する必要は無い. また, 命令を実行する演算器やクラスタの指定も不要であり, オペランドのレジスタは変数を用いて記述できる. このようにリニアアセンブリはアーキテクチャが持つ演算器, レジスタ, クラスタなどを意識せずにコーディングを行うことができる.

リニアアセンブラは, リニアアセンブリを入力として, 命令スケジューリングや演算器/レジスタの割り当て, スピル/リロード命令の追加などを行いアセンブリを出力する. 単体で使用する以外にコンパイラのバックエンドとしての使用も考えられる.

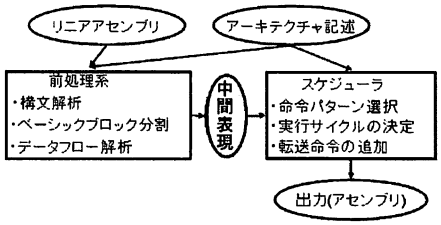

図 2 リターゲッタブル・リニアアセンブラの構成

3. リターゲッタブル・リニアアセンブラ

リターゲッタブル・リニアアセンブラは図 2 のように, リニアアセンブリとターゲットアーキテクチャの記述を入力とし, 最適化されたアセンブリを出力する. すなわち, アーキテクチャ記述を用意すれば, そのプロセッサ用のリニアアセンブラが得られることになる.

同一の命令セットであっても VLIW のスロット数, ディスパッチモデル, レジスタ構成を変えることにより, 性能, ハードウェア量, 消費電力のバランスを大きく変化させることができる. リターゲッタブル・リニアアセンブラはこのようなマイクロアーキテクチャの探索にも有用と考えられる. また, リターゲッタブル・リニアアセンブラの行う命令スケジューリングやレジスタ割り当てなどの処理は, コンパイラのバックエンドが行う処理と同じであるため, これをリターゲッタブル・コンパイラのバックエンドとして利用することもできる.

3.1 アーキテクチャ記述

本稿でターゲットとする VLIW プロセッサのモデルは, 大阪大学で開発された特定用途向プロセッサ開発ツール ASIP Meister [5] のモデルに基づくものであり, TMS320C62x, 富士通の FR400 シリーズ [8], PHILLIPS 社の TriMedia [9] などの VLIW 型プロセッサを扱うことができる.

プロセッサは, 任意の数のスロットを持ち, 命令のパイプライン段数は任意とする. 以下, 一度にディスパッチされる命令を「長命令」, リニアアセンブリ中の命令を「リニアアセンブリ命令」, プロセッサの個々のスロットが実行する命令を「マイクロ命令」と呼称する.

アーキテクチャ記述は, リソースの情報, リニアアセンブリ命令の情報, マイクロ命令の情報, ディスパッチ情報, データ転送表から成る.

(1) リソース情報

プロセッサが持つレジスタやレジスタファイル, 演算器などの情報を記述する. それぞれのレジスタファイルの容量や, レジスタのデータ幅, レジスタペアを構成するレジスタなどの情報を記述する.

(2) リニアアセンブリ命令の情報

各リニアアセンブリ命令に対し, 命令のシンタックス, オペラン

ド名とその使用法, 命令を実行可能なマイクロ命令の集合を記述する. 図 3 にレジスタを使用する加算命令 ADD_Register の例を示す. シンタックスは正規表現で記述する. $reg はレジスタオペランドにマッチし, 括弧で囲まれたものが左から 1 番目, 2 番目, …のオペランドとなる. オペランド情報は, それが順に src1, src2, dst に対応し, それぞれ read, read, write の属性を持つ. 「マイクロ命令」はこのリニアアセンブリ命令をマッピング可能なマイクロ命令の集合である. ADD_L1_AAA は, L1 を使用し, レジスタファイル GRA に, 3 つのオペランドそれぞれがアクセスするマイクロ命令を表わす.

ID		ADD_Register
シンタックス		\s*ADD\s+($reg)\s*,\s*($reg)\s*,\s*($reg)
オペランド	1	src1(read)
	2	src2(read)
	3	dst(write)
マイクロ命令		ADD_L1_AAA, ADD_L1_ABA, ADD_L1_BAA, ADD_L2_BBB, ADD_L2_BAB, ADD_L2_ABB, ADD_S1_AAA, ADD_S1_ABA, ADD_S2_BBB, ADD_S2_BAB, ADD_D1_AAA, ADD_D2_BBB

図 3 リニアアセンブリ命令情報の例

(3) マイクロ命令情報

各マイクロ命令の使用するスロットやパイプライン・ステージ毎の動作を記述する. 図 4 に, レジスタファイル GRA と M1 を使用する乗算を行うマイクロ命令 MPY_M1_AAA の例を示す. このマイクロ命令は"slot3"からディスパッチされる. 実行ステージの 1 ステージ目でレジスタファイル GRA にポート 0, ポート 1 を使用して read を行い, 演算器 M1 を使用して乗算を行っている. 2 ステージ目では結果をレジスタファイル GRA のレジスタに格納している. 使用する演算器, アクセスするレジスタファイルが異なるものはそれぞれ別個のマイクロ命令となる. また, それぞれのマイクロ命令をディスパッチできるスロットは一意に決まるものとする.

ID	MPY_M1_AAA
slot	slot3
stage 1	source0 = GRA.read0(src1) source1 = GRA.read1(src2) result = M1.mul(source0, source1)
stage 2	GRA.write1(dst, result)

図 4 マイクロ命令情報の例 (フォワーディング無)

プロセッサがフォワーディングを行う場合は, それをフォワーディングユニットへのアクセスの形で記述する. 本稿で扱うフォワーディングは, [6] のモデルを使用する. 図 5 に図 4 にフォワーディングを追加したマイクロ命令の例を示す. 1 ステージ目において, フォワーディングユニット FWU0, FWU1 は受け取るデータがフォワーディングされていれば, フォワーディングユニットに write されたデータを受け取り, そうでなければレジスタからデータを転送する. 1, 2 ステージ目では演算結果を FWU0, FWU1 にフォワーディングしている.

ID	MPY_M1_AAA
slot	slot3
stage 1	tmp0 = GRA.read0(src1) tmp1 = GRA.read1(src2) source0 = FWU0.read(src1, tmp0) source1 = FWU1.read(src2, tmp1) result = M1.mul(source0, source1) FWU0.write1(dst, result) FWU1.write1(dst, result)
stage 2	GRA.write1(dst, result) FWU0.write2(dst, result) FWU1.write2(dst, result)

図 5 マイクロ命令情報の例 (フォワーディング有)

(4) ディスパッチ情報

長命令としてディスパッチ可能なマイクロ命令の組み合わせを記述する. マイクロ命令単位での記述は効率が悪いので, 複数のマイクロ命令を集めたものをマイクロ命令グループとし, ディスパッチできるマイクロ命令グループの組み合わせを記述する. 図 6 にディスパッチ情報の例を示す. 各行は長命令で実行可能なマイクロ命令グループの組を表わす.

slot1	slot2	slot3	…	slot8
P_{ALU_L1}	P_{ALU_S1}	P_{MUL_M1}	…	P_{ALU_S1}
P_{ALU_L1}	P_{SHIFT_S1}	P_{MUL_M1}	…	P_{ALU_S1}
P_{ALU_L1}	P_{SHIFT_S1}	P_{MUL_M1}	…	P_{SL_S1}
P_{ALU_L1}	P_{ALU_S1}	P_{MUL_M1}	…	P_{SL_S1}
…	…	…	…	…

図 6 ディスパッチ情報の例

(5) データ転送表

レジスタからレジスタ, レジスタからメモリ, メモリからレジスタへの転送を行う際に使用するマイクロ命令と, そのオペランドの指定法を記述する. 図 7 にデータ転送表の例を示す. 例えば, 1 行目はレジスタファイル GRA から GRB への転送は, マイクロ命令 ADD_L1_AIB の 2 番目のオペランドを 0 にして行うことを示している.

転送元	転送先	マイクロ命令列
GRA	GRB	(ADD_L2_AIB %src1, 0, %dst)
GRB	GRA	(ADD_L1_BIA %src1, 0, %dst)
GRA	MEM	(STW_D1_AA %base, %offset, %src)
GRB	MEM	(STW_D2_BB %base, %offset, %src)
MEM	GRA	(LDW_D1_BB %base, %offset, %dst)
MEM	GRB	(LDW_D2_BB %base, %offset, %dst)

図 7 データ転送表の例

3.2 リニアアセンブリとアセンブリ

本稿で扱うリニアアセンブリとそこから出力されるアセンブリの例をそれぞれ図 8 (a), (b) に示す. リニアアセンブリのオペランドは図 8(a) 6 行目のように基本的には変数を記述するが 8, 9 行目のように A2, B2 のようなプロセッサが実際に持つレ

```
 1:.reg a,b,c,d,e,f,g
 2:.rf GRB h
 3:.base A1
 4:.temp 24:72
 5:
 6:    SUB g,g,g
 7:L1:
 8:    LDW *A2++,a
 9:    LDW *B2++,b
10:    MPY a,b,c
11:    ADD c,g,g
12:    SUB h,1,h
13:[h]  B L1
```

(a) リニアアセンブリ

```
 1:    SUB_L1_AAA A3,A3,A3
 2:L1:
 3:    LDW_D1_AA *A2++,A4
 4:|| LDW_D2_BB *B2++,B4
 5:|| SUB_L2_BIB B5,1,B5
 6:    NOP
 7:    NOP
 8:    NOP
 9:    NOP
10:    MPY_M1_ABA A4,B4,A5
11:    NOP
12:    ADD_L1_AAA A5,A3,A3
13:    CB_S1      L1,B5
```

(b) アセンブリ

図 8 リニアアセンブリと出力

ジスタも指定できる. 図 8(a) 1 から 4 行目の "." から始まる記述はディレクティブであり, 以下のことをユーザーが指定する.

(1) **.reg** (1 行目)
変数がレジスタに格納されることを宣言する.

(2) **.rf** (2 行目)
変数が指定したレジスタファイルのレジスタに格納されることを宣言する. この例は, 変数 **h** を GRB に割り当てることを指定している.

(3) **.base, .offset** (3, 4 行目)
データ転送命令でデータを格納するアドレスを得るための, ベースレジスタとオフセットを指定する. スケジューラがスピル/リロードコードを追加する際にメモリ中にデータを格納できるアドレスをユーザが指定するために記述する.

関数の呼び出し規約に関しては, リニアアセンブラは特別な処理を行わず, ユーザが必要な処理 (引数の受け渡しやレジスタの保存など) をリニアアセンブリで直接コーディングするものとする.

4. コード生成の方式

リターゲッタブル・リニアアセンブラの処理の流れを図 2 に示す. リニアアセンブリ入力に対して前処理系が構文解析を行い, スケジューラが扱う中間表現に変換する. スケジューラは基本ブロックの中間表現毎に, アーキテクチャ記述に基づきコードの最適化を行う.

4.1 前処理系

リニアアセンブリ入力を, 分岐命令やラベルの位置で分割し, 基本ブロック分割を行う. 命令が分岐命令かどうか, またそれが条件付かどうかはマイクロ命令記述から判定する.

4.2 スケジューラ

アーキテクチャ記述を基に, スケジューラは前処理系が出力する中間表現中の各命令に対して, 実行にかかるサイクル数がなるべく少なくなるように, 命令の実行開始サイクルとマイクロ命令を決定する. 各命令のオペランドにはレジスタを割り当てるが, 必要がある場合にはスピル/リロードなどの転送命令の追加を行う.

図 9 にスケジューリングの例を示す. 2 つの乗算に対してすでにマイクロ命令は割り当てられ, それぞれ 1 サイクル目から実行されているとする. この 2 つの命令は依存関係がなく, またマイクロ命令が同じ資源を使用していないので並列にスケジューリングできる. 次の加算に対して割り当てるマイクロ命令の候補として ADD1 と ADD2 があるとする. ADD2 を割り当てた場合, 前の 2 つの命令と依存関係があるため 5 サイクル目から実行することになる. ADD1 を割り当てた場合, 前の命令がそれぞれフォワーディングユニットである FWU0, FWU1 にフォワーディングしたデータを受け取ることができるため命令間依存距離が縮まり, 3 サイクル目に実行することができる.

クラスタ化されたプロセッサの場合, ある変数に対してそれを定義する命令と使用する命令が別々のクラスタに割り振られる場合が生じる. TMS320C62x の「クロスパス」のように専用のパスを使用してそれぞれのクラスタにアクセスできるマイクロ命令が存在すれば, そのマイクロ命令を割り当てればよいが, そうでない場合はレジスタ間のデータ転送を行う命令を挿入して, マイクロ命令を割り当てられるようにする. また, レジスタの数が足りない場合はスピルコードを追加する. 追加する命令は, アーキテクチャ記述中のデータ転送表から選択し, 他の命令と共にスケジューリングする.

オペランドの中に read, write の両方が行われている場合は, read と write を別個のオペランドとして扱う. 例えば, **LDW *p++, a** というリニアアセンブリ命令はオペランド **p** に割り当てられるレジスタからデータを読み, ポストインクリメントを行っているが, この **p** の read, write それぞれを別オペランドとして扱う.

プロセッサの中には複数のレジスタを使用して一つのデータを扱うレジスタペアを持つものがある. プロセッサによってアセンブリ中でのレジスタペアのシンタックスは異なり, ペアを構成するレジスタすべてを記述するものと, ペアの一部のみを記述するものが存在する. 一部のみを記述する場合, ペアを構成するオペランドがどれかを判断しにくいため, 本稿で扱うリニアアセンブリではペアを構成するオペランドすべてを記述するものとする. それぞれの変数がペアであることは命令の情報に記述し, ペアを構成できるようにレジスタ割り当てを行う.

4.3 スケジューリング問題の定式化

リニアアセンブリ命令の集合を A とする. 各命令はオペレーション, オペランド, 依存の情報を持つ. ある命令 $a \in A$ を実行可能なマイクロ命令の集合を $P(a)$ とする. i 番目のオペランドが read であるか, write であるかを $rw(a,i)$, オペランドに使用している変数を $v(a,i)$, オペランドに対して依存関係のある命令と, そのオペランド番号のペアの集合を $D(a,i)$ とする.

リニアアセンブリ命令中の変数の集合を V とし, $v \in V$ に対して, $r(v,t)$ を, 時刻 t において変数 v のデータが格納されている記憶要素とする.

レジスタなどの記憶要素 (レジスタなど) の集合を S とし, $s \in S$ が所属する記憶要素クラス (レジスタファイルなど) を

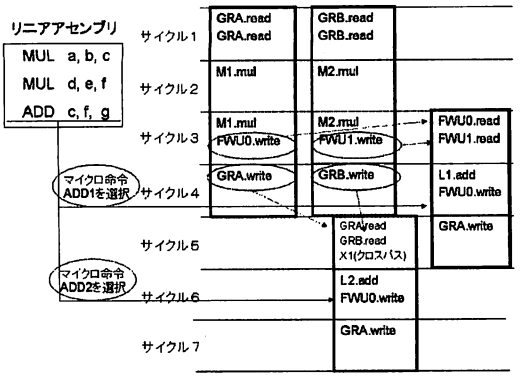

図 9　命令スケジューリングの例

$b(s)$ とする.

マイクロ命令の集合を P とする. $p \in P$ がの i 番目のオペランドの記憶要素にアクセスするステージを $e(p,j)$, アクセスする記憶要素クラスを $s(p,i)$ とする. p が k ステージ目で使用する資源の集合を $R(p,k)$ とする. p を実行するスロットを slot(p) とし, slot(p) は $R(p,1)$ に含むものとする. p が所属する命令グループを $g(p)$ とし, 長命令としてディスパッチ可能なマイクロ命令グループの組の集合を L とする.

プロセッサがフォワーディングを行う場合, フォワーディングの情報も必要になる. p の i 番目のオペランドが read するフォワーディングユニットを $fwr(p,i)$, i 番目のオペランドがフォワーディングユニットからデータを read するステージを $fwrs(p,i)$ とする. p の i 番目のオペランドが write するフォワーディングユニットとステージのペアの集合を $FWW(p,i)$ とする.

スケジューラはリニアアセンブリの各命令 $a \in A$ に対してその命令を実行するマイクロ命令 $p(a)$ と, 実行開始サイクル $c(a)$ を決定し, a の各オペランド i に対して割り当てる記憶要素 $r(a,i)$ を決定する. $p(a)$ は $P(a)$ から選択する.

命令パターンを決定した際, 命令の実行開始サイクルはデータ依存関係, 資源制約を考慮し決定する.

(1)　資源制約

すべての時刻において, 同じ資源は高々1 回までしか使用できない. $a_1, a_2 \in A$ に対して, k_1, k_2 を任意のステージとしたとき,

$$c(a_1)+k_1 = c(a_2)+k_2 \text{ならば } R(p(a_1),k_1) \cap R(p(a_2),k_2) = \phi$$

でなければならない.

(2)　データ依存制約

ある命令 $a_1 \in A$ の実行開始サイクルを決定する際, すべての依存 $(a_2,j) \in D(a,i)$ に対して

$$c(a_2) + e(p(a_2),j) < c(a) + e(p(a),i)$$

を満たす必要がある.

フォワーディングを行う場合は依存の種類により, 次の制約を満たす必要がある. 扱うフォワーディングモデルは完全フォワーディング [6] とする.

(2-a)　RAW 依存の場合

依存 $(a_2,j) \in D(a_1,i)$ に対して, $rw(a_1,i) = read, rw(a_2,j) = write$ の時, この依存は RAW である. 依存関係のある命令 a_2 が write するフォワーディング $(fwu, fws) \in FWW(p(a_2),j)$ の中に, $fwr(p(a_1),i) = fwu$ を満たすものがある時, a_1 は

$$c(a_2) + fws < c(a_1) + fwrs(p(a_1),i)$$

を満たすサイクルに実行開始できる.

(2-b)　WAR 依存の場合

依存 $(a_2,j) \in D(a_1,i)$ に対して, $rw(a_1,i) = write, rw(a_2,j) = read$ の時, この依存は WAR である. 命令 a_1 が write するフォワーディング $(fwu, fws) \in FWW(p(a_1),i)$ の中に, $fwr(p(a_2),j) = fwu$ を満たすものがある時, 命令 a_1 が fwu に最も早くデータをフォワーディングするステージを $x = \min fws(fwu)$ としたとき, a_1 は

$$c(a_2) + fwrs(p(a_2),j) < c(a_1) + x$$

を満たす必要がある.

(2-c)　WAW 依存の場合

依存 $(a_2,j) \in D(a_1,i)$ に対して, $rw(a_1,i) = write, rw(a_2,j) = write$ の時, この依存は WAW である. 命令 a_1 が write するフォワーディング $(fwu_1, fws_1) \in FWW(p(a_1),i)$, 命令 a_2 が write するフォワーディング $(fwu_2, fws_2) \in FWW(p(a_2),j)$ の中に, $fwu_1 = fwu_2$ を満たすものがある時, 命令 a_1 が fwu_1 に最も早くデータをフォワーディングするステージを $x_1 = \min fws(fwu_1)$ 命令 a_2 が fwu_2 に最も早くデータをフォワーディングするステージを $x_2 = \min fws(fwu_2)$ としたとき, a_1 は

$$c(a_2) + x_2 < c(a_1) + x_1$$

を満たす必要がある.

(3)　ディスパッチ制約

すべての時刻 t に対して, $A_t = \{a \in A \mid c(a) = t\}$ とした時に,

$$\{g(p(a)) \mid a \in A_t\} \in L$$

を満たす必要がある.

(4)　アクセスする記憶要素クラスの制約

任意のリニアアセンブリ命令 $a_1, a_2 \in A$ の i,j 番目のオペランドについて

$$v(a_1,i) = v(a_2,j) \text{ ならば } (p(a_1),i) = s(p(a_2),j)$$

である必要がある. $s(p(a_1),i) \neq s(p(a_2),j)$ であるならば, $s(p(a_1),i)$ から $s(p(a_2),j)$ への転送命令を追加する必要がある.

(5)　記憶要素の割り当ての制約命令 $a \in A$ の i 番目のオペランドに記憶要素を割り当てる際に, 以下の制約を満たす必要がある.

(5-a)　記憶要素クラスの制約

割り当てる記憶要素の所属する記憶要素クラスと, マイクロ命令のアクセスする記憶要素クラスが一致しなければならない.

すべての $a \in A$ について,

$$b(r(a,i)) = s(p(a),i).$$

(5-b) 記憶要素の制約
異なる2つの変数の $v_1, v_2 \in V, v_1 \neq v_2$ について, 同じサイクル t で同じ記憶要素が割り当てられていてはならない.

$$\forall t : r(v_1,t) \neq r(v_2,t).$$

(5-c) 同一命令中のオペランドの制約
同一命令の複数オペランドが同じ変数であったならば, 同じレジスタを割り当てなければならない.

$v(a,i) = v(a,j)$ であれば, $r(a,i) = r(a,j)$.

(5-d) レジスタペア
レジスタペアを扱う場合は, S の要素 s に対して使用する記憶要素の集合を $U(s)$ とする. レジスタペアの記憶要素 $s_1 \in S$ を割り当てる時には $U(s_1)$ の要素である記憶要素すべてが使用可能でなければならない.

5. プロトタイプの実装

リターゲッタブル・リニアアセンブラのプロトタイプを Cygwin 上で Perl5.8.7 で実装した. プロトタイプのスケジューラはリストスケジューリングを行い, 転送命令はマイクロ命令を割り当てる際に, アクセスするレジスタファイルの不一致やレジスタの不足が生じたときのみ追加するとした. TMS320C62x の一部約 50 のマイクロ命令を含む約 2500 行のアーキテクチャ記述を作成した. プロトタイプにこのアーキテクチャ記述と 20 行程度のアセンブリを入力とした場合に出力を得るのに要した時間は約 0.5 秒であった (Core2 T5500 1.66GHz, メモリ 1GB).

6. むすび

本稿ではリターゲッタブル・リニアアセンブラとその構成法を提案した. 今後, TriMedia TM1300 など他のアーキテクチャをターゲットとすることを目標に, これらのターゲットに合わせてスケジューラが行う処理を改良して行く予定である. 今回開発したスケジューラは, 定式化の検証を主目的に極力簡易な実装としたが, 出力するコードの質の向上のために, 最適化機能を強化したスケジューラを開発していくことも課題である. また, 現在のスケジューラは基本ブロックごとにスケジューリングを行っているが, 大きなスーパーブロックやハイパーブロックに定式化を拡張することも今後の課題に挙げられる.

謝辞 本研究を進めるにあたり, 御助言, 御指導を頂きました大阪大学今井研究室の関係諸氏に感謝致します. 種々のアドバイスを頂きました, 日本電気株式会社の池川将夫氏, 久村孝寛氏に感謝致します. また, スケジューリング問題の定式化についてアドバイスを頂いた益井勇気氏, アーキテクチャ記述の作成を行って頂いた浅井圭悟氏をはじめ, 支援と助言を頂いた石浦研究室の諸氏に感謝致します.

文 献

[1] http://archc.sourceforge.net/.
[2] O. Schliebusch, A. Hoffman, A. Nohl, G. Braun, H. Meyr: "Architecture Implementation Using the Machine Description Language LISA," in *Proc. ASP-DAC 2002*, pp. 239–244 (Jan. 2002).
[3] R. Leupers and P. Marwedel: *Retargetable Compiler Technology for Embedded Systems*, Kluwer Academic Publishers (2001).
[4] Texas Instruments: *TMS320C6000 Optimizing Compiler User's Guide* (Mar. 2000).
[5] 武内良典, 小林真輔, 今井正治: "特定用途向きインストラクションセットプロセッサ開発環境 ASIP Meister と DSP アプリケーションへの応用," 信学技報, CAS2002-61 (Sept. 2002).
[6] 平岡佑介, 石浦菜岐佐, 今井正治: "プロセッサ仕様記述からの命令依存距離抽出," 信学技報, VLD2004-119/CPSY2004-85 (Jan. 2005).
[7] http://www.ti.com/.
[8] http://jp.fujitsu.com/.
[9] http://www.nxp.com/.