独自ベクトル処理機能を備えたプロセッサ向け 自動ベクトル化コンパイラの開発

永池　晃太朗 † 大津　金光 †† 横田　隆史 †† 小島　駿 ††
† 宇都宮大学工学部情報工学科　†† 宇都宮大学大学院地域創生科学研究科

1　はじめに

画像認識等のデータ並列性の高い処理の高速化にはSIMD 命令による処理が効果的である．しかし，SIMD 命令による処理は同時演算数を変更する場合，同時に機械語コードを作り直す必要がある．そこで我々は機械語コードの変更なしに同時並列演算数を変更することができる，データ並列処理のためのスケーラブルなベクトル処理機能を備えたベクトル拡張付き RISC-V を提案している [1].

我々のベクトル拡張付き RISC-V は，ARM のベクトル拡張である ARM SVE[2] を参考に組込み向けに RISC-V[3] をベクトル拡張したものである．これにより，機械語コードが同時演算数に依存しないスケーラブルなベクトル拡張を実現したが，ベクトル拡張付き RISC-V に対応したコンパイラがない．

この問題に対して，解決策として既存の RISC-V コンパイラに変更を加えることによって，ベクトル拡張付き RISC-V アセンブリコードを得ることのできるコンパイラを開発する手段を検討した．

2　ベクトル拡張付き RISC-V

ベクトル拡張付き RISC-V は RISC-V が有する 4 つのカスタム命令のためのオペコード領域のうち 2 つを利用しており，1 つをベクトルロード，ストア命令，もう一方をベクトル演算命令とベクトル制御命令に使用している．また，ベクトル拡張付き RISC-V の命令フォーマットは RISC-V に従う．ベクトルロード，ストア命令のアドレスはベースとなるスカラレジスタの値にオフセットを加えることで計算する．オフセットはスカラレジスタ，ベクトルレジスタ，即値の 3 種類を指定できる．オフセットにベクトルレジスタを指定することでギャザー，スキャターによる非連続なロード，ストアを行う．

ベクトル演算命令は，プレディケートあり演算，プレディケートなし演算命令，即値による演算命令に分けられる．演算命令は，基本的な算術論理演算命令，乗除算命令，ベクトルレジスタの各要素の総和を求める命令やアドレス計算の際に用いるインデックスレジスタを生成する命令からなる．プレディケートあり演算命令ではプレディケートレジスタによるベクトルマスク制御を行う．

Development of automatic vectorization compiler for processors with original vector processing function.
†Kotaro Nagaike, ††Kanemitsu Ootsu, ††Takashi Yokota, ††Shun Kojima,
Department of Information Science, Faculty of Engineering, Utsunomiya University (†)
Graduate School of Regional Development and Creativity, Utsunomiya University (††)

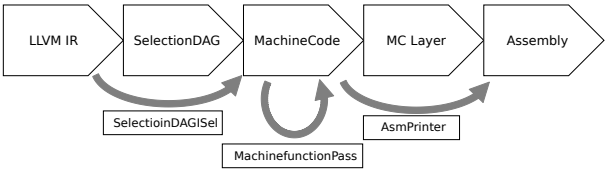

図 1: コード生成の形式変換の流れ

ベクトル制御命令はプレディケート演算命令とベクトル長操作命令に分けられる．プレディケート演算命令は，プレディケートレジスタ同士の論理演算を行う．ベクトル長操作命令は，スカラレジスタに対しベクトル長を足す等の命令がある．

レジスタの構成は，v0 から v31 のベクトルレジスタ，vp0 から vp15 のプレディケートレジスタ，ベクトル長レジスタ，x0 から x31 の汎用レジスタとなっている．また，ベクトル拡張付き RISC-V の基本命令は RV32I の命令を組み込んでいる．

3　LLVM コンパイラ基盤

コンパイラの開発はコンパイラ基盤である LLVM[4] を用いて行う．コンパイラ基盤はコンパイラの機能がモジュール化されており，既存機能の再利用が可能になっている．そのため，コンパイラの開発の際にコンパイラ基盤を用いることによって独自部分のみの開発で済む．前述した通りベクトル拡張付き RISC-V は RISC-V を拡張したものであることから，LLVM の RISC-V コンパイラとしての機能を再利用してコンパイラの開発を行う．

なお，使用する LLVM のバージョンは 13.0.0 である．

LLVM は C 言語などのソースコードから中間表現である LLVM IR に変換を行うフロントエンド，アセンブリコード生成を行うバックエンドからなる．本研究では独自命令生成のためにバックエンドに対して変更を加える．

バックエンドにおけるコード生成について，処理の流れとパスを図 1 に示す．SelectionDAG フェーズでは SelectionDAGISel パスによって LLVM IR を DAG 形式へと変換し，DAG のパターンマッチングによる命令の単純化，共通部分削除などの最適化，ターゲット命令への変換の順に行う．SelectionDAGISel は命令変換を行った後に MachineCode 形式を出力する．MachineCode 形式はターゲットが使用する実際の命令を持った形式であり，MachinefunctionPass によって SSA ベースの最適化を行い，命令への物理レジスタの割当

```
class RVInstVVMIQS<bits<5> funct5, bits<3> opv,
             dag outs, dag ins,string opcodestr, string argstr>
   : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
  bits<5> vs2;
  bits<5> vs1;
  bits<5> vd;
  bit vm;

  let Inst{31-27} = funct5;
  let Inst{26-25} = 0b10;
  let Inst{24-20} = vs2;
  let Inst{19-15} = vs1;
  let Inst{14-12} = opv;
  let Inst{11-7} = vd;
  let Opcode = 0b0001011;
}
```

図 2: 命令フォーマットの定義

```
class VALUVVMIQS<bits<5> funct5, bits<3> opv, string opcodestr>
     : RVInstVVMIQS<funct5, opv, (outs VR:$vd),
                    (ins VR:$vs2, VR:$vs1),
                    opcodestr, "$vd, $vs2, $vs1"> {
}

multiclass VALU_IV_V_X_I_MIQS<string opcodestr, bits<3> opv> {
          def V : VALUVVMIQS<0b00001, opv, opcodestr # ".w">,
                  Sched<[WriteVIALUV, ReadVIALUV, ReadVIALUV]>;
          […]
          def I : VALUVI8MIQS<0b00101, opv, opcodestr # "i.w">,
                  Sched<[WriteVIALUI, ReadVIALUV]>;
}

defm VADD_V : VALU_IV_V_X_I_MIQS<"vadd", 0b000>;
```

図 3: 命令の定義

を行う．レジスタの割当を行った後は非 SSA 形式となる．MC Layer は AsmPrinter パスでアセンブリコード，オブジェクトコードを出力するための命令形式であり，関数などの構造がない．

以上の LLVM バックエンドでの処理はレジスタ情報，命令フォーマットや命令などのターゲット情報に従って行われ，ターゲット情報は LLVM におけるターゲット情報記述のためのドメイン固有言語である TableGen を用いて記述する．

LLVM では RISC-V の V 拡張用に TableGen によるベクトルレジスタ，ベクトル命令の定義が既に存在している．本研究ではベクトルレジスタはそのまま使用し，ベクトル命令については既存の定義に従った形式でベクトル拡張付き RISC-V 命令の定義を行う．

また，ベクトル命令生成のための機能として LLVM には自動ベクトル化機能が備わっている．この機能はフロントエンドにて入力ソースコードの繰り返し処理をベクトル化された LLVM IR に変換する．ベクトル化された LLVM IR からパターンマッチングにより各ターゲットへのベクトル命令への変換などが行われる．RISC-V の V 拡張用のパターンマッチングについても実装済みのものを再利用する．

4 LLVM バックエンドにおける独自命令の生成機能の実装

ベクトル拡張付き RISC-V 命令の定義のために，図 2 のようなクラスを定義した．クラス RVInstVVMIQS は 2 つのベクトル要素を入力に持ち，ベクトル要素を出力する命令の命令フォーマットである．各命令フォーマットは 32bit の命令フィールドを定義している基本クラス RVInst を継承する形で命令によって異なるフォーマットを定義する．

命令の定義は命令フォーマットのクラスを継承し，エンコードの値やニーモニックを指定して行う．ベクトル拡張付き RISC-V 命令の定義を図 3 に示す．図 3 の VALUVVMIQS クラスはベクトル算術演算用の命令定義であり，入力レジスタと出力レジスタの指定を行う．また，図 3 にて命令のインスタンス化も行っており，ここで命令の文字列と命令選択に用いるための 12-14 ビットの値の指定を行っている．

現在ベクトル拡張付き RISC-V のベクトル命令の内，ベクトル演算命令のプレディケートなし，即値による演算命令の実装が完了している．

5 おわりに

本研究では，自動でベクトル化された独自のベクトル拡張付き RISC-V 命令アセンブリコードを得るための LLVM バックエンドの実装を行った．

今後の課題として，未実装の命令生成の実現が挙げられる．現在実装済みの命令は命令の定義等で実装が可能であった．しかし，未実装の命令については現在のベクトル化された LLVM IR からの生成が困難であると考えられる．そのため，レジスタの定義に加え，LLVM IR の生成を行うフロントエンドの変更が必要である．

謝辞

本研究は一部 JSPS 科研費 20K11726 の援助による．

参考文献

[1] Yoshiki Kimura,et al: “Proposal of Scalable Vector Instruc-tion Set for Embedded RISC-V Processor,” Proc. 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW), Vol.1, pp.435-439, 2019.

[2] Nigel Stephens, et al: “The ARM Scalable Vector Extension,” IEEE Micro, Vol.37, No.2, pp.26-39, 2017.

[3] Andrew Waterman, Krste Asanovi: “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA,Document Version 2.2,” 2017.

[4] Chris Lattner, Vikram Adve: “LLVM: A Compilation Frame-work for Lifelong Program Analysis Transformation,” Proc. 2004 International Symposium on Code Generation and Optimization (CGO ’04), pp.75, 2004.