21-68

卒業論文

LLVMコンパイラ基盤を用いた
ベクトル化コード生成についての検討

2022年3月

永池 晃太朗

宇都宮大学工学部
情報工学科

内容梗概

画像処理などの複数のデータに対して同じ演算を行う処理については，単一命令で複数データの処理を行う SIMD 命令によるデータ並列処理による高速化が可能である．しかし，SIMD によるデータ並列処理を行う場合，SIMD 命令は 1 命令で演算するデータ数である同時演算数が決まっている．そのため演算性能を上げるために同時演算数を変更すると機械語コードを作り直す必要がある．そこで，機械語コードの変更なく，同時演算数を変更できるスケーラブルなベクトル拡張を実現したベクトル拡張付き RISC-V が開発されている．

しかし，このベクトル拡張に対応したコンパイラがないため，ベクトル拡張付き RISC-V のアセンブリコードを生成できないという問題点がある．これに対する解決策としてベクトル拡張付き RISC-V に対応したコンパイラの実現方法を検討する．コンパイラ基盤である LLVM を用いて既に実装済みの RISC-V 向けコンパイラの機能を再利用すればコンパイラを一から開発するより容易にアセンブリコードの生成を行うことができる．また，LLVM に備わっている自動ベクトル化機能を用いることでベクトル化されたコードの生成が可能であることから，LLVM を用いることによってベクトル拡張付き RISC-V 向けにアセンブリコードを得ることができると考える．

本論文では LLVM コンパイラ基盤におけるコード生成について述べ，独自命令生成のための命令定義を行う．ベクトル拡張付き RISC-V の命令の内，ベクトル算術・論理演算命令，即値を用いるシフト命令の実装について述べる．さらに，実装した命令が正常に入力ソースコードから生成されることを確認する．C 言語で記述した配列加算等のプログラムを用いて、そのベクトル化されたアセンブリコードを生成できることを示す．また，現段階では実装できていない命令として，ベクトルロード・ストア命令等がある．その命令生成に向けて新たに必要な定義や変更点などについて述べる．

Consideration of Automatic Generation of Vectorized Code Using LLVM Compiler Infrastructure

Kotaro Nagaike

Abstract

For processing that performs the same operation on multiple data, such as image processing, it is possible to achieve higher speed by using data parallel processing with SIMD instructions that process multiple data with a single instruction. However, when using SIMD for data parallel processing, the number of simultaneous operations, which is the number of data to be processed in one instruction, is fixed for SIMD instructions. Therefore, if the number of concurrent operations is changed in order to increase the processing performance, the machine language code must be rewritten. To solve this problem, RISC-V with vector extension has been developed to realize scalable vector extension that can change the number of concurrent operations without changing the machine code.

However, there is a problem that the assembly code of RISC-V with vector extension cannot be generated because there is no compiler that supports this vector extension. As a solution to this problem, I consider how to realize a compiler that supports RISC-V with vector extensions. By using LLVM, which is a compiler infrastructure, and reusing the functions of already implemented compilers for RISC-V, it is easier to generate assembly code than developing a compiler from scratch. In addition, the automatic vectorization function of LLVM can be used to generate vectorized code, so I consider that assembly code for RISC-V with vector extensions can be obtained by using LLVM.

In this paper, I describe the code generation in the LLVM compiler infrastructure and define instructions for generating original instructions. Among the instructions of RISC-V with vector extensions, I describe the implementation of vector arithmetic and logic instructions and shift instructions using immediate values. In addition, I confirm that the implemented instructions can be successfully generated from the input source code, and show that programs such as array addition written in C can be used to generate their vectorized assembly code. I will also show that the vectorized assembly code can be generated using programs such as array addition written in C. There are instructions such as vector load/store instructions that have not been implemented at this stage. In this paper, I describe the definitions and changes

required to generate such instructions.

目 次

第1章　はじめに

FPGA (Field Programmable Gate Array) はユーザによって回路の再構成が可能な LSI (Large Scale Integration) であり，目的の処理をハードウェアとして実装可能なデバイスである．近年 FPGA の大容量化，高性能化によって大規模な回路が実現可能になった．これにより FPGA は自動運転を始めとする組込み分野での利用増加が期待されている．

FPGA を用いたハードウェア開発は HDL (Hardware Description Language) による RTL (Register Transfer Level) 設計が広く用いられている．RTL は FPGA 上に構成する回路の信号の流れや制御構造を直接設計できる一方，動作検証やデバックが難しく，短期間で複雑な処理の開発は困難である [1]．そのため，FPGA による開発期間を短縮する方法として，専用ハードウェアとプロセッサを用いたソフトウェアによる処理を組み合わせる方法が考えられる．FPGA 上のハードウェアリソースを用いて実装するプロセッサのことをソフトコアプロセッサという．ソフトコアプロセッサは柔軟性が高く，プロセッサの構成を変更することによって処理対象のアプリケーションに特化させることができる．一度作成すれば部分的な変更で異なるアプリケーションへの対応も可能なため，開発期間の短縮が期待できる．専用ハードウェアを開発する場合は，回路で実行する処理に対して最適な回路の設計を行うことによって高性能な回路を実現できるが，新たに回路を設計する必要があるため開発のコストが高い．一方，ソフトコアプロセッサでは処理をソフトウェアで記述できるため開発コストを抑えることができるがアプリケーション専用の回路としての最適化することができないため，性能は専用ハードウェアに比べて低い．組込みシステムを開発する上で高い性能が求められる処理を専用ハードウェアで行い，高い性能が必要でない処理をソフトコアプロセッサで行うことによって，すべての処理を専用ハードウェアで開発する場合と比べて開発期間を短縮でき，ソフトコアプロセッサの性能を向上させることができれば専用ハードウェアを削減でき開発コストを抑えることができる．

組込み分野では AI 技術に注目が集まっている．独立行政法人情報処理推進機構の調査によると，将来強化/新たに獲得したい技術として組込み/IoT 関連企業の 46%が AI 技術を挙げている [2]．また，近年自動運転システムへの AI 技術の応用など組み込む分野における画像認識などの画像処理を行う機会が増加している．画像処理では画像を構成する画素に対して同じ処理を行うようなものが多い．このように複数のデータに対して同じ処理を行う場合，データを 1 つずつ処理するのではなく，

データを分割して並列に処理することで処理の高速化が可能である．データ並列処理の方法として単一命令で複数データの処理を行う SIMD (Single Instruction, Multiple Data) がある [3]．SIMD によるデータ並列処理を行う場合，1 命令で複数のデータを扱う SIMD 命令によって処理を行う．ソフトコアプロセッサで SIMD 命令によるデータ並列処理を行う場合，ソフトコアプロセッサは FPGA 上で構成されているため自由にその構成を変更することができる．そのためプロセッサにおける SIMD 演算器の数を変更することによって性能の変更が可能で，目的の処理能力をもったプロセッサの実現が可能である．しかし，演算性能を上げるために演算器数を変更するとそれに応じた機械語コードを作り直す必要がある．異なる同時演算数でも同一の機械語コードを利用可能とするためには，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が必要である．スケーラブルなベクトル拡張によって機械語コードを変更することなく状況に応じ，容易に同時演算数を増やし高性能化することが可能となる．

スケーラブルなベクトル拡張を実現したものとしてオープンな命令セットアーキテクチャである RISC-V[4] をベクトル拡張したベクトル拡張付き RISC-V が開発されている [5]．ベクトル拡張付き RISC-V は組込み機器に広く用いられている ARM (Advanced RISC Machine) のベクトル拡張である ARM SVE (Scalable Vector Extension)[6] の命令セットを参考に組み込み向けに RISC-V に拡張したものである．しかし，ベクトル拡張付き RISC-V に対応したコンパイラが存在していない．そのため，ベクトル拡張付き RISC-V のアセンブリコードを得るためには直接アセンブリコードの記述を行う必要があり，目的の処理を行うためのアセンブリコードを得ることが容易ではない．

そこで解決策としてベクトル拡張付き RISC-V のベクトル命令のアセンブリコードを得るためのコンパイラの実現方法を検討した．ベクトル拡張付き RISC-V に対応したコンパイラを実現できれば，C 言語等の高水準言語からベクトル命令によるアセンブリコードを得ることができ，ソフトウェアの開発が行いやすくなる．ベクトル拡張付き RISC-V のアセンブリコードを得るコンパイラの実現のためにコンパイラ基盤である LLVM[7] を用いる．LLVM はコンパイラの各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なためコンパイラのすべての機能を開発する必要がない．また，ベクトル拡張付き RISC-V のアセンブリコードを得るためにはソースコードのループ処理をベクトル命令に対応した形式に変換する必要がある．LLVM にはループを自動でベクトル命令に対した形式に変換する自動ベクトル化機能が既に備わっている．この機能を利用してベクトル拡張付き RISC-V の命令の生成を行う．

本論文では，第 2 章で現在のベクトル拡張付き RISC-V プロセッサについて述べる．第 3 章でベクトル拡張付き RISC-V 命令の生成のために利用したコンパイラ基盤である LLVM について述べる．第 4 章で実際に命令生成のための実装について述べる．第 5 章では実際のソースコードからアセンブリコードの出力を行った結果について述べる．そして最後に第 6 章で本論文のまとめと課題について述べる．

第2章　　ベクトル拡張付きRISC-Vプロセッサ

本章ではベクトル処理機能を持つソフトコアプロセッサであるベクトル拡張付きRISC-Vプロセッサについて述べる．

2.1　RISC-V

RISC-Vはカルフォルニア大学バークレイ校が新たに開発したRISCの設計思想に基づく命令セットアーキテクチャ (Instruction Set Architectures) である．RISC-Vはオープンな ISAであり，使用料のかからないオープンソースライセンスで提供されている．

従来の ISAでは，後方バイナリ互換性を維持するために過去に拡張した命令すべてを実装する必要がある．しかし，過去の拡張命令すべてを実装してしまうと，命令数が増加し複雑になる．そこでRISC-Vではモジュール式 ISAを採用している [8]．RISC-Vのシステムの機能を独立したモジュールとして分け，アプリケーションに応じて拡張機能を組み込むかを選択できる．RISC-Vには必ず組み込まなければならない基本命令セット (RV32I ,RV64I) の他に，主な拡張機能として乗算及び除算 (RV32M ,RV64M)，単精度浮動小数点 (RV32F ,RV64F)，倍精度浮動小数点 (RV32D ,RV64D) 等がある．

RISC-Vの基本命令は6つの命令フォーマットで表すことができる．基本命令の命令フォーマットを図 2.1 に示す．

R形式は2つのソースレジスタを扱う形式，I形式は12ビットの即値を扱う命令，S形式はスト

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

図 2.1　基本命令の命令フォーマット

ア命令，B 形式は条件分岐命令，U 形式は 20 ビットの即値を扱う命令，J 形式は無条件ジャンプ命令の形式である．命令形式が単純であるため命令のデコーディングが単純化される．

2.2 ベクトル拡張付き RISC-V プロセッサ概要

ベクトル拡張付き RISC-V プロセッサとは，RISC-V コアにベクトル演算機能を拡張したソフトコアプロセッサである．図 2.2 にベクトル拡張付き RISC-V プロセッサの全体構成を示す．ベクトル拡張付き RISC-V プロセッサの RISC-V コアプロセッサは RISC-V プロセッサを使用している．このプロセッサは Verilog-HDL で記述された 5 段パイプラインプロセッサで必要最低限の基本命令セットである RV32I で規定された命令を実装している．

命令のフェッチおよびデコードは RISC-V コアプロセッサ上で行い，デコード結果がベクトル拡張命令であったときはベクトル処理ユニットで動作させる．

メモリアクセスはプロセッサからメモリコントローラを介して SDRAM にアクセスする．メモリアクセス要求がプロセッサから発行されたときは，メモリコントローラが SDRAM を制御してメモリアクセスを実現する．また，ベクトルメモリアクセスに関してもメモリコントローラにて行う．SDRAM コントローラのデータバス幅は 128 ビットであるから，プロセッサとメモリコントローラ間のデータバス幅は 128 ビットとなっている．

RISC-V コアプロセッサは命令フェッチ (IF)，命令デコード (ID)，実行 (EX)，メモリアクセス (MA)，ライトバック (WB) の 5 段のパイプラインで構成されている．ベクトル処理ユニットと RISC-V コアの双方で動作する必要のある命令に対応するためベクトル処理ユニットは RISC-V コアと同じく 5 段パイプライン構成となっており，RISC-V コアと協調動作する構成となっている．なお，命令フェッチ部分と命令デコード部分は RISC-V コアと共通となっている．

ベクトル拡張付き RISC-V プロセッサによってスケーラブルなベクトル拡張を実現し，同時演算数の変更によって機械語コードの変更の必要がなくなったが，ベクトル拡張付き RISC-V プロセッサが対応している命令セットに対応しているコンパイラがないため，ベクトル化アセンブリコードを得ることができない．そのためコンパイラを作る必要がある．

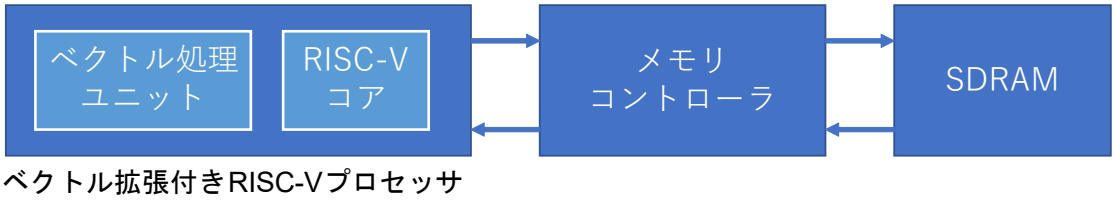

図 2.2　ベクトル拡張付き RISC-V プロセッサの構成

2.3 ベクトル拡張付き RISC-V 命令セット

ベクトル拡張付き RISC-V は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 ISA であり，既存の RISC-V を ARM SVE を参考にベクトル拡張したものである．ベクトル拡張付き RISC-V ではスケーラブルなベクトル拡張を行うためにプレディケートによるループ制御がある．プレディケートを用いたループ処理はループカウンタと処理する全データ数を比較してループカウンタが全データ数より小さい間対応するプレディケートレジスタを True にする．この命令によってベクトル処理で余りの要素がある場合，余りの要素の部分に対応するプレディケートの要素のみを True にする．これによって余りの要素が変化してもすべてのデータ数分ベクトル処理を行うことができ，同時演算数に依存しない機械語コードが実現できる．プレディケートレジスタの一例を図 2.3 に示す．図 2.3 の (a) ではループカウンタがデータ数より小さい場合であり，プレディケートの要素が全て True になっている．図 2.3 の (b) では端数要素の処理を行う場合であり，ループカウンタがデータ数より小さいときのみ要素が True となっている．

ベクトル命令はベクトルロード，ストア命令，ベクトル演算命令，ベクトル制御命令の 3 つに分けられる．RISC-V にはカスタム命令用にオペコード領域が 4 つ用意されているがそのうちの 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．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．ベクトル演算命令は，プレディケートあり演算命令，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．浮動小数点命令は必要なハードウェア資源が多いためサポートしていない．ベクトル拡張付き RISC-V は基本命令に関して RV32I のみ組み込んでいる．

レジスタ構成はベクトルレジスタ v0-v31，ベクトルマスク制御に用いるためのプレディケートレジスタ vp0-vp7，プレディケートレジスタ同士の論理演算に用いるプレディケートレジスタ vp8-

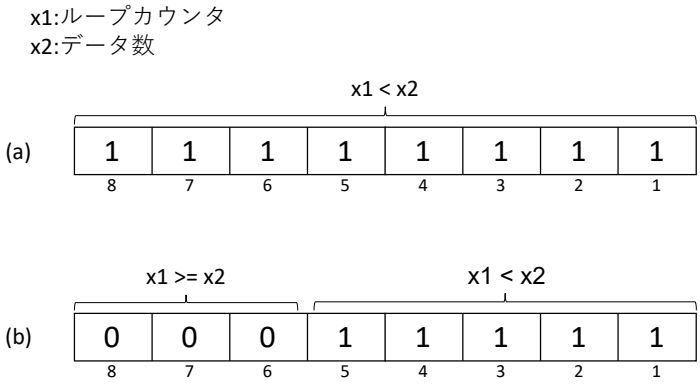

図 2.3　プレディケートレジスタの一例

vp15，RISC-V の汎用レジスタ x0-x31，プログラムカウンタとなっている．ベクトルレジスタの長さは $128\times2^n(0 \leq n \leq 4)$ ビットで表され，汎用レジスタの幅は 32 ビットとなっている．

命令のフォーマットは RISC-V の命令形式である R 形式に従ったものとなっている．また，デコーダの単純化のために同じフィールドにはできるだけ同じ機能をもたせている．

第3章　　LLVM

本章ではベクトル命令拡張付き RISC-V のアセンブリコードを得るためのコンパイラの開発に用いたコンパイラ基盤である LLVM について述べる．

3.1　LLVM 概要

LLVM は 2000 年にイリノイ大学で開発が開始されたコンパイラ基盤である．コンパイラ基盤とはコンパイラに必要となるモジュールをまとめたもので，コンパイラを開発するためのフレームワークである．LLVM の構成を図 3.1 に示す．LLVM はソースコードを LLVM の中間表現である LLVM IR に変換するフロントエンド，LLVM IR に対して最適化等の操作や LLVM IR から機械語やアセンブリコードへの変換を行うバックエンドに分かれている．中間表現である LLVM IR は階層構造のドメイン固有言語である．LLVM ではこれらの機能がモジュール化されており，独自機能を実装する以外は既存のものを再利用することができる．例えば新たなアーキテクチャ向けのコンパイラを開発する際はバックエンドのみ実装を行い，フロントエンドについては再利用することができる．

LLVM には既に RISC-V を対象としたコード生成のためのバックエンドが実装されている．本研究では RISC-V を独自にベクトル拡張したベクトル拡張付き RISC-V の命令の生成を目的としているため，この RISC-V 向けのバックエンドに対して変更を加えることによって独自命令の生成を行う．

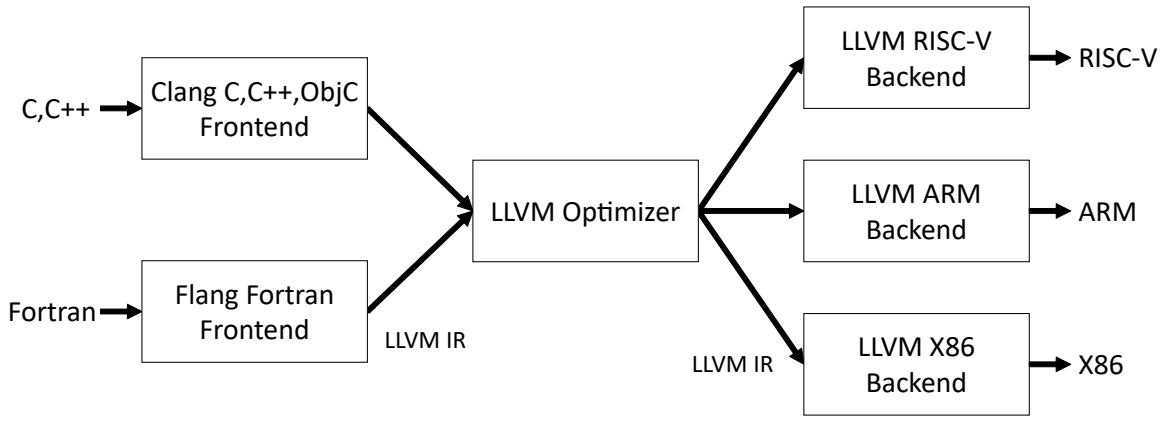

図 3.1　LLVM の構成

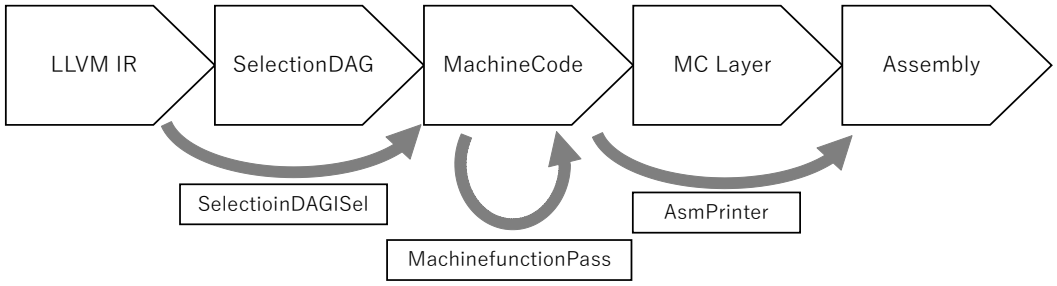

図 3.2　LLVM バックエンド

LLVM バックエンドにおけるコード生成の流れを図 3.2 に示す．LLVM バックエンドでは Pass によって処理が行われる．図 3.2 では LLVM バックエンドにおけるデータフォーマットの変化と実行される Pass を表している．

バックエンドでは LLVM IR から DAG (Directed Acyclic Graph) である SelectionDAG へフォーマットを変換する．SelectionDAG は LLVM IR をグラフ形式で表したもので，各命令やデータの依存関係を表現する．SelectionDAG への変換は SelectionDAGISel パスで行われ，SelectionDAG は最終的に SelectionDAGISel パスによって MachineCod 形式に変換される．SelectionDAGISel では Lower, Combine，Legalize，Select，Schedule のフェーズから構成される．

Lower は LLVM IR から SelectionDAG に変化させるフェーズである．このフェーズでは LLVM IR から SelectionDAG のノードへと一対一の対応を行う．この段階ではターゲットマシンでは利用できない命令やデータ形式を含んでる不正 (illegal) な状態である．

Combine フェーズではパターンマッチングによる置き換えで最適化を行い，命令の単純化を行う．

Legalize はターゲットマシンではサポートされていない命令やデータ形式を他のものに置き換えるフェーズである．このフェーズによって不正な状態であった SelectionDAG ノードが正当 (legal) な状態となる．

Select フェーズでは SelectionDAG のノードをターゲットマシンの命令を含んだ MachineNode へと変換する．

Schedule フェーズでは構築されたグラフの依存関係を元に命令をスケジューリングするフェーズである．

SelectionDAGISel パスではこれらのすべてのフェーズが終わった後に MachineCode を出力する．

Machinefunction パスでは MachineCode という形式を扱う．この形式は LLVM IR と似た構成になっているが，より機械語に近い表現になっている．MachineCode が SelectionDAGISel によって生成された直後はまだ命令で扱うレジスタは無限個あると仮定した仮想レジスタや phi 関数を含んだ SSA (Static Single Assignment form) 形式で表現されている．MachinefunctionPass ではレジスタの割当や

phi 命令の削除を行い，MachineCode を非 SSA 形式へと変換する．

AsmPrinter パスは MachineCode を MC Layer 形式へと変換した後にアセンブリコードなどを出力するパスである．MC Layer 形式は MachineCode 形式のような階層構造がない形式であり，アセンブリコードへの変換だけでなく，オブジェクトファイルへの変換のための形式である．

3.2 LLVM による自動ベクトル化機能

LLVM には自動ベクトル化機能が備わっている．自動ベクトル化とは配列の演算処理などのループによる繰り返し処理をベクトル命令の形式に置き換える機能である．LLVM による自動ベクトル化機能はソースコードにおけるループをベクトル化された LLVM IR へと変換が行われる．LLVM による自動ベクトル化の例を図 3.3 に示す．LLVM IR においてベクトル命令はベクトル型を用いて表現される．図 3.3 にて <128 x i32> となっている箇所がベクトル型の指定を行っており，これは 128 個の 32 ビット整数の演算を行うベクトル型の命令となっている．

また，LLVM IR では対象の全データに演算を行うための制御を行っている．例えば対象データ要素数が 400 とし，一度にベクトル演算を行う個数を 128 とする．この場合，128 個を対象としたベクトル演算を 3 回行う．するとベクトル演算では 384 個の要素を演算することができるが，16 個の要素が余っている．このあまりに関しては逐次処理によってスカラ演算を 16 回繰り返す処理を行う．これによってすべてのデータの演算を可能にしている．

LLVM ではベクトル化された LLVM IR からベクトル命令を生成することが可能であるが，RISC-V の V 拡張命令の生成については現在実験段階であり，完全なアセンブリコードを得ることはできない．

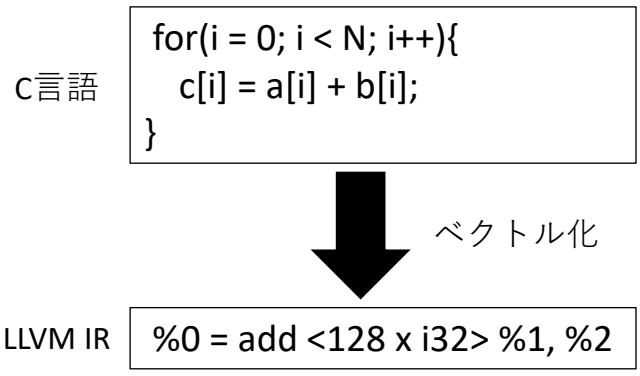

図 3.3 LLVM による自動ベクトル化例

第4章　ベクトル拡張付きRISC-Vコンパイラ

本章ではベクトル拡張付きRISC-Vのベクトル命令をLLVMによって生成するための命令実装手法と，実際に実装した命令の定義について述べる．

4.1　LLVMバックエンドにおける独自命令実装手法

LLVMにおいて特定のターゲットマシン命令の生成は3.1で述べたSelectionDAGISelパスによるSelectフェーズにてLLVM IRの命令からターゲットマシン命令に変換されることによって行われる．この変換はSelectionDAGのノードに対してパターンマッチングを行い，特定のパターンにターゲット命令を対応させる手法で行われる．

このパターンマッチングに用いるパターンはLLVMの固有ドメイン言語であるTableGenによって行われる．TableGenはターゲットマシンの命令やレジスタ等の情報を記述するために用いられる．TableGenでは命令生成のためにパターンの定義だけでなく，アセンブリコード等の出力のためにニーモニックの定義や命令フォーマットの定義も行っている．

SelectionDAGのパターンマッチングの例を図4.1に示す．図4.1では加算命令の例を示している．図4.1上にパターン定義クラスPatがある．Patクラスの第一引数と変換前のSelectionDAGのノードが一致した場合，Patクラスの第二引数で指定したノードに変換する．

この様にSelectionDAGのパターンと命令ごとに定義されているパターンが一致したときに対応した命令にノードが変換される．LLVMではRISC-VのV拡張命令のためのパターン定義が既に実装されており，そのパターンと一致した際に変換する命令を独自のベクトル拡張付きRISC-Vの命令に定義しなおすことによってベクトル拡張付きRISC-V命令の生成を実現する．

LLVMにおける命令の定義は命令フォーマットの定義と命令の定義に分かれる．命令フォーマットはTableGenによって命令の種類ごとにフォーマットのクラスを定義を行う．LLVMにおける命令フォーマットは基本クラスRVInstを継承する形で行われる．RVInstでは32ビットのフィールドInstや命令のニーモニックを格納するAsmStringを定義している．このRVInstを継承して異なるフォーマットを定義していく．

命令の定義は命令フォーマットのクラスをインスタンス化する形で行われるが，そのために命令フォーマットを更に継承したクラスを定義する．このクラスは似たような種類の命令を定義するに

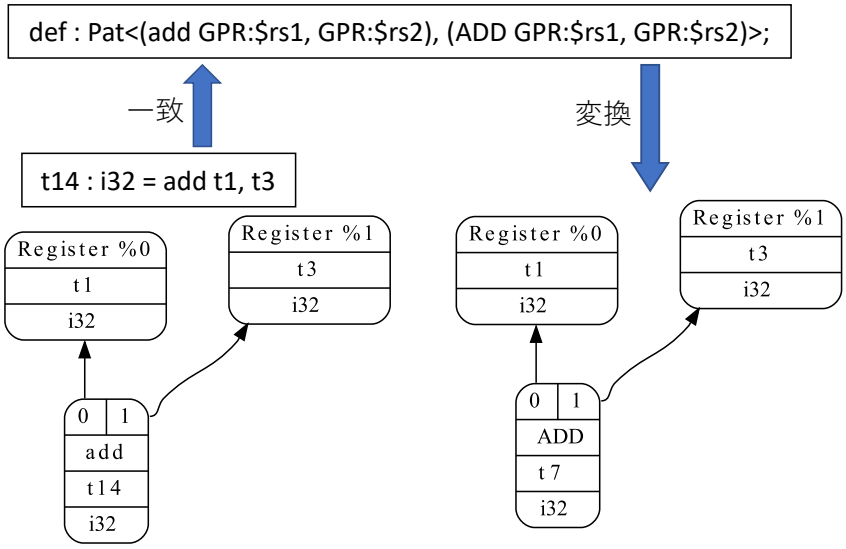

図 4.1 SelectionDAG の命令変換

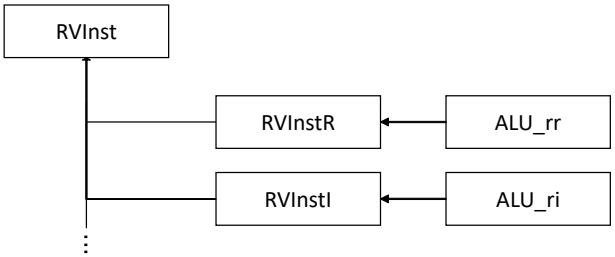

図 4.2 基本命令フォーマットクラス継承関係

当たって同じような定義を何度も繰り返さないように定義する．

命令フォーマット，命令定義のクラスの継承関係を図 4.2 に示す．基本クラスである RVInst を継承して基本命令のフォーマットの定義を行っている．クラス ALU_rr，ALU_ri はそれぞれ算術演算命令のためのクラスである．ALU_rr は汎用レジスタ同士の加算や減算の算術演算の定義に用いられ，ALU_ri は即値を用いた算術演算の命令定義に用いられる．

LLVM では上記のように命令の定義が行われており，この定義方法に従った形式で我々のベクトル拡張付き RISC-V のベクトル命令の定義を行う．

4.2 ベクトル拡張付き RISC-V 命令の定義

4.1 で述べた LLVM の RISC-V バックエンドでの命令生成のための命令定義クラスについて，ベクトル拡張付き RISC-V のベクトル命令のための命令フィールドと命令の定義を行った．

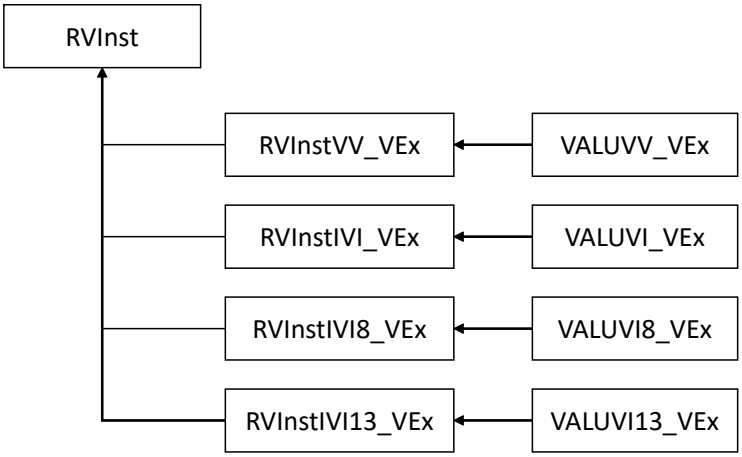

図 4.3　定義したクラスの継承関係

図 4.3 に定義した命令フィールド定義クラスと命令定義のためのクラスの継承関係を示す.

本研究ではベクトル拡張付き RISC-V の命令のうち，ベクトル演算命令のプレディケートレジスタを用いない命令を実装した．プレディケートレジスタを用いない命令はオペランドで用いるレジスタや命令の種類が類似しているため，プレディケートレジスタを用いている命令より比較的容易に実装が可能である.

実装した命令とそのフォーマットを図 4.4 にまとめる.

図 4.4 で vadd などの命令フォーマットの 26-25 ビットにて指定されている size はベクトル要素のサイズを指定するためのフィールドであり，この値に応じたサフィックスが命令の末尾に追加されるものである．本研究では整数要素の処理の場合のみを想定し，この size の値は 32 ビットサイズを表すために'0b10' で固定している．また，即値を用いた算術演算命令である vaddi ,vsubi の命令フィールドのうち 14 ビットの sh は 24-20 ビットで指定した即値を 8 ビットシフトするかを決定するためのフィールドである．このフィールドが 0 である場合はシフトを行わず，1 であるときは 8 ビットのシフトを行う．この sh の値は vaddi 等の命令のオペランドにて指定を行うが，本研究ではシフトを行わない 0 で値を固定している.

実際に定義した命令フォーマットクラスの一覧を図 4.5 に示す．RVInstVV_VEx は vadd ,vsub ,vor ,vxor ,vand のための命令フォーマットで．RVInstIVI_VEx は 5 ビットの即値を用いる vsra ,vsrl ,vsll 用の命令フォーマット，RVInstIVI8_VEx は 8 ビットの即値を用いる vaddi ,vsubi 用の命令フォーマット，RVInstIVI13_VEx は 13 ビットの即値を用いる vori ,vxori ,vandi 用の命令フォーマットの定義である.

また，実際の TableGen による定義の例として RVInstVV_VEx の定義を図 4.6 に示す．図 4.5 で示したフォーマットとなるように命令フィールド Inst のどのフィールドにどの値が格納されるかを指

	31–27	26–25	24–20	19–15	14–12	11–7	6–0
vadd	00001	size	vs2	vs1	000	vd	0001011
vsub	00001	size	vs2	vs1	001	vd	0001011
vor	00001	size	vs2	vs1	100	vd	0001011
vxor	00001	size	vs2	vs1	101	vd	0001011
vand	00001	size	vs2	vs1	110	vd	0001011
vsra	00010	size	imm5	vs1	000	vd	0001011
vsrl	00010	size	imm5	vs1	001	vd	0001011
vsll	00010	size	imm5	vs1	011	vd	0001011

	31–27	26–25	24–23	22–15	14	13–12	11–7	6–0
vaddi	00101	size	00	imm8	sh	00	vd	0001011
vsubi	00101	size	00	imm8	sh	01	vd	0001011

	31–28	27–15	14–12	11–7	6–0
vori	0011	imm13	000	vd	0001011
vxori	0011	imm13	001	vd	0001011
vandi	0011	imm13	010	vd	0001011

図 4.4　実装した命令

	31–27	26–25	24–20	19–15	14–12	11–7	6–0
RVInstVV_VEx	funct5	size	vs2	vs1	funct3	vd	opcode
RVInstIVI_VEx	funct5	size	imm5	vs1	funct3	vd	opcode

	31–27	26–25	24–23	22–15	14	13–12	11–7	6–0
RVInstIVI8_VEx	funct5	size	funct2	imm8	sh	funct2	vd	opcode

	31–28	27–15	14–12	11–7	6–0
RVInstIVI13_VEx	0011	imm13	010	vd	0001011

図 4.5　実装した命令フォーマットクラス

定する．

命令の定義を図 4.7 に示す．図 4.7 の ALUVV_VEx はベクトル同士の演算を行うための命令の定義用に定義の繰り返しを防ぐためのクラスである．ALUVV_VEx では出力レジスタと入力レジスタの指定を行う．このクラスを定義することで例えば “vadd.w vd vs1 vs2” と “vsub.w vd vs1 vs2” のようにオペランドが同じ命令の定義を行う場合，ALUVV_VEx をインスタンス化する際に入出力レジスタの定義を繰り返しを防ぐことができる．

命令のインスタンス化も図 4.7 で行っている．ベクトル算術・論理演算命令のインスタンス化を行っている．引数では命令の文字列と命令選択のための値を指定している．

TableGen による命令の定義は以上の様に行われ，残りの命令フォーマットや命令についても同様

```
1  class RVInstVV_VEx<bits<3> funct3, dag outs, dag ins, string opcodestr, string argstr>
2      : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
3    bits<5> vs2;
4    bits<5> vs1;
5    bits<5> vd;
6
7    let Inst{31-27} = 0b00001;
8    let Inst{26-25} = 0b10;
9    let Inst{24-20} = vs2;
10   let Inst{19-15} = vs1;
11   let Inst{14-12} = funct3;
12   let Inst{11-7} = vd;
13   let Inst{6-0} = 0b0001011;
14 }
```

図 4.6　TableGen による RVInstVV_VEx の定義

```
1  class VALUVV_VEx <bits<3> funct3, string opcodestr>
2     : RVInstVV_VEx <funct3, (outs VR:$vd), (ins VR:$vs2, VR:$vs1),
3              opcodestr, "$vd, $vs2, $vs1"> {
4  }
5
6  def VADD_VV : VALUVV_VEx <0b000, "vadd.w">,
7              Sched<[WriteVIALUV, ReadVIALUV, ReadVIALUV]>;
8
9  def VSUB_VV : VALUVV_VEx <0b001, "vsub.w">,
10             Sched<[WriteVIALUV, ReadVIALUV, ReadVIALUV]>;
```

図 4.7　命令の定義

に定義を行った.

第5章　　検証と課題

本章では実装した命令の生成を実際にプログラムから生成されるかを検証し，課題として未実装である命令について述べる．

5.1　アセンブリコードの生成検証

実装した命令が正しく出力されるか，図 5.1 に示した配列加算のプログラムを入力としてアセンブリコードの出力を行った．

図 5.2 は配列 a,b の各要素の加算を配列 c に格納する関数 add のアセンブリコードで，データ数 999 個に対して一度にベクトル演算を行う配列要素を 4 として生成を行ったものである．25 行目にあるベクトル加算のための ‘vadd.w’ が今回実装した我々のベクトル拡張付き RISC-V の命令である．その他の命令は RISC-V で定義されている命令である．処理の流れはラベル ‘.LBB0_2’ にてベクトル演算を行っており，ラベル ‘LBB0_4’ にて余りの要素の処理を行っている．

図 5.1 の配列要素加算部分を変更し，他の定義した命令の生成を検証した．

ベクトル減算命令である vsub の生成を図 5.3 に，ベクトル論理積，ベクトル論理和，ベクトル排他的論理和命令である vand,vor,vxor の生成を図 5.4，図 5.5，図 5.6 に示す．

また，即値を用いたベクトル演算命令の生成も検証した．即値によるベクトル加算・減算命令の生成を図に示す．定数を指定して配列要素への加算を行うプログラムを入力することで即値による加算命令となる．

同様に即値を用いる論理演算命令についても vandi を図 5.8 に vori を図 5.9 に vxori を図 5.10 に示す．

```
1  void add(int *a, int *b, int *c){
2          for (int i=0;i<N;i++){
3                  c[i] = a[i] + b[i];
4          }
5  }
```

図 5.1　配列加算プログラム

```
1  add:
2        lui    a3, 1
3        addi a3, a3, -100
4        add  a4, a2, a3
5        add  a5, a0, a3
6        add  a6, a1, a3
7        sltu  a5, a2, a5
8        sltu  a3, a0, a4
9        and  a3, a5, a3
10       sltu  a5, a2, a6
11       sltu  a4, a1, a4
12       and  a4, a5, a4
13       or    a3, a3, a4
14       mv   a6, zero
15       bnez a3, .LBB0_3
16       addi a6, zero, 996              #ベクトル処理対象の要素数を計算
17       mv   a7, a0
18       mv   a5, a1
19       mv   a3, a2
20       addi a4, zero, 996              #ループカウンタの設定
21 .LBB0_2:
22       vsetivli    zero, 4, e32, m1, ta, mu
23       vle32.v    v25, (a7)            #ベクトルロード
24       vle32.v    v26, (a5)            #ベクトルロード
25       vadd.w    v25, v26, v25         #ベクトル加算
26       vse32.v    v25, (a3)            #ベクトルストア
27       addi a4, a4, -4                 #ループカウンタをデクリメント
28       addi a3, a3, 16                 #アドレス更新
29       addi a5, a5, 16                 #アドレス更新
30       addi a7, a7, 16                 #アドレス更新
31       bnez a4, .LBB0_2
32 .LBB0_3:
33       addi a4, a6, -999               #余りの要素数を計算
34       slli   a3, a6, 2                #処理済要素分のアドレス計算
35       add  a2, a2, a3                 #アドレス更新
36       add  a1, a1, a3                 #アドレス更新
37       add  a0, a0, a3                 #アドレス更新
38 .LBB0_4:
39       lw    a6, 0(a0)                 #ロード
40       lw    a5, 0(a1)                 #ロード
41       mv   a3, a4                     #カウンタをセット
42       add  a4, a5, a6                 #加算
43       sw    a4, 0(a2)                 #ストア
44       addi a4, a3, 1                  #カウンタをインクリメント
45       addi a2, a2, 4                  #アドレス更新
46       addi a1, a1, 4                  #アドレス更新
47       addi a0, a0, 4                  #アドレス更新
48       bgeu       a4, a3, .LBB0_4      #余りがまだあったら再度計算
49       ret
```

図 5.2　生成されたアセンブリコード

```
#define N 999

void add(int a[], int b[], int c[]){

        for (int i=0;i<N;i++)
        {
                c[i] = a[i] - b[i];
        }
}
```

```
[…]
.LBB0_2:
        vsetivli  zero, 4, e32, m1, ta, mu
        vle32.v   v25, (a7)
        vle32.v   v26, (a5)
        vsub.w    v25, v25, v26
        vse32.v   v25, (a3)
        addi      a4, a4, -4
        addi      a3, a3, 16
        addi      a5, a5, 16
        addi      a7, a7, 16
        bnez      a4, .LBB0_2
[…]
```

図 5.3　vsub 命令の生成

```
#define N 999

void add(int a[], int b[], int c[]){

        for (int  i=0;i< N;i++)
        {
                c[i] = a[ i] & b[ i];
        }
}
```

```
[…]
.LBB0_2:
        vsetivli  zero, 4, e32, m1, ta, mu
        vle32.v   v25, (a7)
        vle32.v   v26, (a5)
        vand.w    v25, v26, v25
        vse32.v   v25, (a3)
        addi      a4, a4, -4
        addi      a3, a3, 16
        addi      a5, a5, 16
        addi      a7, a7, 16
        bnez      a4, .LBB0_2
[…]
```

図 5.4　vand 命令の生成

即値によるベクトルシフト演算命令について，ベクトル論理左シフト命令である vsll を図 5.11 に示す．ベクトル算術右シフト命令である vsra を図 5.12 に示す．ベクトル論理右シフト命令である vsrl を図 5.13 し示す．ベクトル論理右シフト命令を出力するために C 言語に変更を加えており，演算対象の配列の型宣言を int 型から unsigned int に変更している．

5.2　課題

本研究ではベクトル拡張付き RISC-V の命令の内，プレディケートなしのベクトル算術・論理演算命令，即値によるシフト命令を実装している．

図 5.1 のアセンブリコードでもベクトルロード・ストア命令等の命令が RISC-V の V 拡張の命令のままであり，ベクトル拡張付き RISC-V 命令の内，ベクトルロード・ストア命令，プレディケート付きベクトル演算命令，ベクトル制御命令について未実装である．これらの命令実装には本研究で行ったように命令の実装のみならず，新たなレジスタの定義に加えて LLVM IR への変更が必要であると考えられる．

```
#define N 999

void add(int a[], int b[], int c[]){

        for (int i=0;i<N;i++)
        {
                c[i] = a[i] | b[i];
        }
}
```

```
[…]
.LBB0_2:
        vsetivli   zero, 4, e32, m1, ta, mu
        vle32.v    v25, (a7)
        vle32.v    v26, (a5)
        vor.w      v25, v26, v25
        vse32.v    v25, (a3)
        addi       a4, a4, -4
        addi       a3, a3, 16
        addi       a5, a5, 16
        addi       a7, a7, 16
        bnez       a4, .LBB0_2
[…]
```

図 5.5　vor 命令の生成

```
#define N 999

void add(int a[], int b[], int c[]){

        for (int i=0;i<N;i++)
        {
                c[i] = a[i] ^ b[i];
        }
}
```

```
[…]
.LBB0_2:
        vsetivli   zero, 4, e32, m1, ta, mu
        vle32.v    v25, (a7)
        vle32.v    v26, (a5)
        vxor.w     v25, v26, v25
        vse32.v    v25, (a3)
        addi       a4, a4, -4
        addi       a3, a3, 16
        addi       a5, a5, 16
        addi       a7, a7, 16
        bnez       a4, .LBB0_2
[…]
```

図 5.6　vxor 命令の生成

```
#define N 999

void add(int a[], int c[]){

        for (int i=0;i<N;i++)
        {
                c[i] = a[i] - 10;
        }
}
```

```
[…]
.LBB0_2:
        vsetivli   zero, 4, e32, m1, ta, mu
        vle32.v    v25, (a3)
        vsubi.w    v25, v25, 10
        vse32.v    v25, (a4)
        addi       a5, a5, -4
        addi       a4, a4, 16
        addi       a3, a3, 16
        bnez       a5, .LBB0_2
[…]
```

図 5.7　vsubi 命令の生成

```
#define N 999

void add(int a[],   int c[]){

        for (int  i=0;i< N;i++)
        {
                c[i] = a[ i] & 0b1010;
        }
}
```

```
[…]
.LBB0_2:
        vsetivli   zero, 4, e32, m1, ta, mu
        vle32.v    v25, (a3)
        vandi.w    v25, 10
        vse32.v    v25, (a4)
        addi       a5, a5, -4
        addi       a4, a4, 16
        addi       a3, a3, 16
        bnez       a5, .LBB0_2
[…]
```

図 5.8　vandi 命令の生成

```
#define N 999

void add(int a[], int c[]){

        for (int i=0;i<N;i++)
        {
                c[i] = a[i] | 0b1010;
        }
}
```

```
[…]
.LBB0_2:
        vsetivli   zero, 4, e32, m1, ta, mu
        vle32.v    v25, (a3)
        vori.w     v25, 10
        vse32.v    v25, (a4)
        addi       a5, a5, -4
        addi       a4, a4, 16
        addi       a3, a3, 16
        bnez       a5, .LBB0_2
[…]
```

図 5.9　vori 命令の生成

```
#define N 999

void add(int a[], int c[]){

        for (int i=0;i<N;i++)
        {
                c[i] = a[i] ^ 0b1010;
        }
}
```

```
[…]
.LBB0_2:
        vsetivli   zero, 4, e32, m1, ta, mu
        vle32.v    v25, (a3)
        vxori.w    v25, 10
        vse32.v    v25, (a4)
        addi       a5, a5, -4
        addi       a4, a4, 16
        addi       a3, a3, 16
        bnez       a5, .LBB0_2
[…]
```

図 5.10　vxori 命令の生成

```
#define N 999

void add(int a[], int c[]){

        for (int i=0;i<N;i++)
        {
                c[i] = a[i] << 2;
        }
}
```

```
[…]
.LBB0_2:
        vsetivli   zero, 4, e32, m1, ta, mu
        vle32.v    v25, (a3)
        vsll.w     v25, v25, 2
        vse32.v    v25, (a4)
        addi       a5, a5, -4
        addi       a4, a4, 16
        addi       a3, a3, 16
        bnez       a5, .LBB0_2
[…]
```

図 5.11　vsll 命令の生成

```
#define N 999

void add(int a[], int c[]){

        for (int i=0;i<N;i++)
        {
                c[i] = a[i] >> 2;
        }
}
```

```
[…]
.LBB0_2:
        vsetivli   zero, 4, e32, m1, ta, mu
        vle32.v    v25, (a3)
        vsra.w     v25, v25, 2
        vse32.v    v25, (a4)
        addi       a5, a5, -4
        addi       a4, a4, 16
        addi       a3, a3, 16
        bnez       a5, .LBB0_2
[…]
```

図 5.12　vsra 命令の生成

```
#define N 999

void add(unsigned int a, unsigned int c[]){

        for (int i=0;i<N;i++)
        {
                c[i] = a[i] >> 2;
        }
}
```

```
[…]
.LBB0_2:
        vsetivli   zero, 4, e32, m1, ta, mu
        vle32.v    v25, (a3)
        vsrl.w     v25, v25, 2
        vse32.v    v25, (a4)
        addi       a5, a5, -4
        addi       a4, a4, 16
        addi       a3, a3, 16
        bnez       a5, .LBB0_2
[…]
```

図 5.13　vsrl 命令の生成

新たなレジスタの定義についてだが，これは未実装命令で用いられているプレディケートレジスタの定義が必要である．ベクトル拡張付き RISC-V ではプレディケートレジスタを用いたベクトル処理を行うことによってスケーラブルなベクトル拡張を実現している．そのためプレディケートレジスタが必須であるが RISC-V はプレディケートレジスタを有していない．そのため，新たにプレディケートレジスタを定義する必要がある．また，定義したプレディケートレジスタをそれを用いる命令のオペランドに割りあてられるようにするなど，命令とレジスタの定義のみでは実装が難しい．

また，LLVM による自動ベクトル化が行われた LLVM IR ではベクトル処理がベクトル演算の繰り返しと余りの要素の処理で行われているのに対して，我々のベクトル拡張付き RISC-V ではプレディケートレジスタを用いたベクトル処理を行っており，余りの要素の処理はプレディケートレジスタを用いた計算を行っている．そのため，現在の LLVM IR から我々のベクトル拡張のプレディケートレジスタを用いる命令を効果的に生成することができないため，ベクトル化された LLVM IR について変更が必要である．

第6章　おわりに

本論文では，スケーラブルなベクトル処理を実現したベクトル拡張付き RISC-V 向けのアセンブリコードを得るための手法について検討した．

まずベクトル拡張付き RISC-V プロセッサ，ベクトル拡張付き RISC-V の概要について確認し，コンパイラ基盤である LLVM におけるコード生成について確認した．

LLVM では入力ソースコードを LLVM 独自の中間表現である LLVM IR に変換し，LLVM IR から SelectionDAG，MachineCode，MCLayer を経てアセンブリコードを生成する．LLVM は機能の再利用ができるため，実装済みの RISC-V 向けのコンパイラ機能を再利用して我々のベクトル拡張付き RISC-V のアセンブリコードの生成を目指した．

また，LLVM では入力ソースコードにおける繰り返し処理をベクトル化された LLVM IR に変換する自動ベクトル化機能を持っている．ベクトル化された LLVM IR ではベクトル演算の繰り返しと余剰要素の演算によるベクトル処理を行っている．

LLVM バックエンドではドメイン固有言語である TableGen によって命令やレジスタの情報が定義され，その定義に従ってアセンブリコードなどの生成が行われるため，我々のベクトル拡張付き RISC-V の命令を LLVM の RISC-V 向けバックエンドに定義した．ベクトル拡張付き RISC-V の命令の内，プレディケートレジスタを用いないベクトル命令としてベクトル算術論理演算命令である vadd,vsub,vand,vor,vxor と即値による演算命令である vaddi,vsubi,vandi,vori,vxori,vsll,vsra,vsrl を実装した．

また，実装した命令のアセンブリコードが正しく生成されるかを実際に配列加算等の命令を入力として生成を検証した．

今後の課題としては，現時点では未実装である命令の実装を行い，動作可能なアセンブリコードを得ることを確認することが挙げられる．

謝 辞

本研究の機会を与えていただき，また，日頃から貴重な御意見，御指導いただいた，大津 金光教授，横田 隆史教授，小島 駿助教に深く感謝致します．そして，本研究において多大な御力添えを頂いた，研究室の方々に感謝致します．

参考文献

[1] 三好健文 : “FPGA 向けの高位合成言語と処理系の研究動向,” コンピュータソフトウェア. Vol.30, No.1, pp.76-84, 2013.

[2] 独立行政法人情報処理推進機構 社会基盤センター: “2020 年度組込み/IoT 産業の動向把握等に関する調査,” 2021.

[3] デイビッド・A・パターソン, ジョン・L・ヘネシー著 成田 光彰 訳: “コンピュータの構成と設計 第 5 版,” 日経 BP 社. 2017.

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

[5] 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.

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

[7] 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.75-86, 2004.

[8] デイビッド・パターソン (著), アンドリュー・ウォーターマン (著), 成田 光彰 (訳): “RISC-V 原典オープンアーキテクチャのススメ,” 日経 BP 社, 2018.

[9] 平石康祐, 橋本瑛大, 大津金光, 大川猛, 横田隆史: “SIMD 拡張ソフトコアプロセッサのための効率的なメモリシステムの検討,” 情報処理学会第 80 回全国大会講演論文集, Vol.1, pp.115-116, 2018.