C-008

RISC-V へのカスタム命令の実装と評価
Implementation of Custom Instructions for RISC-V

今井 信志 † 渡邊 誠也 † 名古屋 彰 †
Nobuyuki Imai Nobuya Watanabe Akira Nagoya

1. はじめに

近年，IoT デバイスが普及しており，IoT デバイス上での処理内容が高度なものとなっている．従来は IoT デバイスのソフトウェアのみの更新によって処理内容の変化に対応することが可能であった．しかし，処理内容が高度化することによって，ソフトウェアのみの更新では処理内容の変化に対応することが困難となっており，ハードウェアの更新も必要となっている．そのため，ソフトウェアとハードウェアの同時設計が重要となっている．

また，IoT デバイスなどの低消費電力で動作するデバイスへ搭載する CPU として，仕様がオープンであることから RISC-V が注目されている．RISC-V にはカスタム命令用のオペコードが定義されており，その命令の内容は設計者に委ねられている．RISC-V のカスタム命令を用いれば，特定領域に特化した RISC-V を設計することができ，ソフトウェアとハードウェアの同時設計によって，IoT デバイスでの処理の高度化に対応することができる．しかし，RISC-V へカスタム命令を実装し，定量的な評価を行った研究は少ない．

そこで，本研究では積和演算命令をカスタム命令として採用し，このカスタム命令を搭載した RISC-V をハードウェア記述言語 FSL (Functional and Scalable hardware description Language) [1] で設計した [2]．設計した RISC-V を論理合成ツールを用いて，FPGA (Field Programmable Gate Array) へと実装し，使用ハードウェアリソース量や動作周波数，消費電力などの諸量を得た．また，導入したカスタム命令を使用するように行列積演算ソフトウェアを実装し，設計した RISC-V の動作をシミュレートすることで，このソフトウェアの実行に要した命令数やクロックサイクル数を測定した．そして，これらの結果を用いて，このカスタム命令導入の効果を定量的に評価した．

本稿では，**2.** で RISC-V とカスタム命令の概要について述べ，**3.** で実装する RISC-V の構成と，カスタム命令の実装手法について述べる．**4.** で本研究での設計環境について述べ，**5.** で設計した RISC-V の評価と考察について述べる．最後に **6.** でまとめ，今後の予定を述べる．

2. RISC-V

本章では，RISC-V の概要とカスタム命令の概要について述べる．

2.1 RISC-V の概要

RISC-V はカリフォルニア大学バークレー校で開発された命令セットアーキテクチャであり，現在は RISC-V Foundation [3] により開発・管理が行われている．また，仕様書 [4] が公開されており，ロイヤリティーフリーに利用することができる．

†岡山大学大学院自然科学研究科, Graduate School of Natural Science and Technology, Okayama University

RISC-V には命令セットが複数存在し，それらは必須となる基本命令セットと実装の選択が可能な拡張命令セットの 2 種に分けられる．また，それぞれの命令セットには名称が割り当てられている．基本命令セットには，整数の加減算や論理演算を有した基本整数命令セットが存在しており，アーキテクチャのビット幅に応じて，32 ビットの RV32I，RV32I が有するレジスタの数を半減させた RV32E，64 ビットの RV64I，128 ビットの RV128I となっている．拡張命令セットには，整数の乗算・除算・剰余算を行う M，アトミックなメモリアクセスを行う A，単精度の浮動小数点演算を行う F，倍精度の浮動小数点演算を行う D などの名称が割り振られている．これらの名称を用いて，実装されている命令セットの組み合わせを表す．例えば，RV64IMA であれば，基本命令セットとして 64 ビットの基本整数命令セット，拡張命令セットとして乗除算命令群とアトミック命令群を実装していることを表す．

オープンソースの RISC-V コアの実装も行われており，カリフォルニア大学バークレー校の研究グループが開発している Rocket Chip [5] やアウトオブオーダー実行をサポートする BOOM (The Berkeley Out-of-Order Machine) [6] などが存在する．これらはいずれも Chisel [7] によって記述されている．

ソフトウェア開発に関する取り組みも行われており，GCC 7.1.0 では RISC-V が正式にサポートされた．また，フル機能の OS やリアルタイム OS などの開発も行われている．

2.2 カスタム命令の概要

RISC-V のオペコードには将来的に追加される拡張命令セット用のフィールドの他に，カスタム命令用のフィールドが定義されている．定義されているカスタム命令用のオペコードの数は，32 ビットと 64 ビットのアーキテクチャで 4 つ，128 ビットのアーキテクチャで 2 つとなっている．また，カスタム命令の内容については仕様書内では触れられておらず，設計者が自由に処理内容を決定することができる．

3. 実装する RISC-V

本章では，実装する RISC-V の構成と，実装するカスタム命令について述べる．

3.1 RISC-V の構成

本研究で実装する RISC-V の命令は RV32IM をベースとし，32 ビット基本整数命令である RV32I から例外，割り込み，CSR (Control and Status Registers) 関連の命令を除いた命令と，乗除算関連の命令セットである M から除算系，剰余算系の命令を除いたものとした．表 1 に実装する命令の一覧を示す．表 1 中の [U] が添えられているものは，符号付きで実行する命令と符号なしで実行する命令の 2 種が存在することを表している．例えば，BLT[U] は符号付きで実行する BLT と，符号なしで実行する BLTU の 2 つの命令を表す．また，[U] と [S] が添えられている MULH[[S]U] は，MULH，MULHU，MULHSU の 3 種の命令を表す．MULH は符号付き整数同士

表 1: 実装する命令

種類	命令
Arithmetic	ADDI，SLTI[U]
	ADD，SUB，SLT[U]，MUL，MULH[[S]U]
Logical	ANDI，ORI，XORI
	AND，OR，XOR
Shift	SLL，SRLI，SRAI
	SLL，SRL，SRA
Immediate	LUI，AUIPC
Branch	BEQ，BNE，BLT[U]，BGE[U]
Store	SB，SH，SW
Load	LB[U]，LH[U]，LW

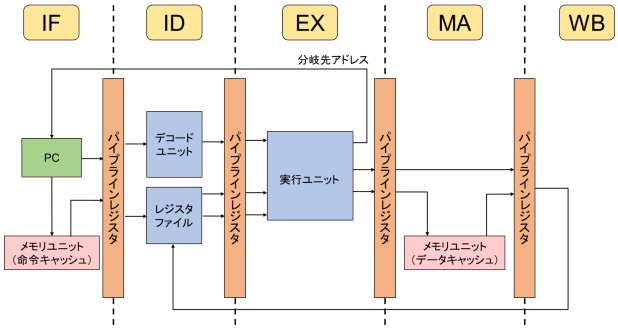

図 1: 実装する RISC-V のパイプライン構成

の乗算，MULHU は符号なし整数同士の乗算，MULHSU は符号付き整数と符号なし整数の乗算となっている．

また，図 1 に実装する RISC-V のパイプライン構成を示す．パイプライン構成はインオーダ実行の 5 段パイプライン構成とし，パイプラインステージは命令フェッチ（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）となっている．

3.2 カスタム命令

実装するカスタム命令の概要とその実装手法について述べる．

3.2.1 実装するカスタム命令の概要

近年，機械学習分野，特にニューラルネットワークにおける演算では，積和演算の高い処理能力が要求される．そこで，本研究では RISC-V へ実装するカスタム命令として，積和演算処理を行うものを採用した．

実装するカスタム命令を表 2 に示す．`madd` 命令は，3 つのオペランドの積和を行い，結果を指定された *rd* レジスタへと格納する．`madd44` 命令は，`a0`（x10）レジスタから `a7`（x17）レジスタまでの固定的に定めた 8 つのレジスタの値を用いて積和を行い，結果を指定された *rd* へと格納する．`madd` 命令，`madd44` 命令ともに演算に用いる数値は 32 ビット整数とする．

以降，カスタム命令を搭載しないコアを C1，`madd` 命令を搭載するコアを C2，`madd44` 命令を搭載するコアを C3 と呼ぶ．

3.2.2 カスタム命令の実装手法

本研究では，カスタム命令を実現するために，ベースとした RISC-V のパイプライン上へ機能を実装した．主にレジスタファイルと実行ユニットの変更を行った．図 2 に各カスタム命令の実装手法を示す．

`madd` 命令

通常の命令ではソースレジスタは 2 つのみであるが，`madd` 命令では 3 つのレジスタを用いて積和演算を行うため，*rd* レジスタ用にレジスタファイルのポートを追加した．

また，積和演算については実行ユニット内の既存の乗算器と加算器を組み合わせて実装を行った．実行（EX）ステージのパイプライン化は行っておらず，演算は 1 クロックサイクルで行われる．

表 2: 実装するカスタム命令

命令書式	処理内容
`madd` *rd, rs1, rs2*	$rd = rd + rs1 * rs2$
`madd44` *rd*	$rd = a0 * a1 + a2 * a3 + a4 * a5 + a6 * a7$

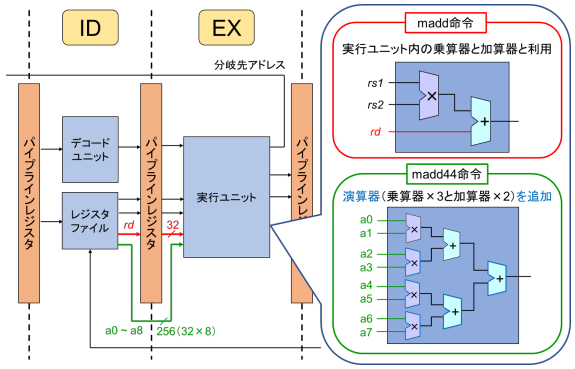

図 2: カスタム命令の実装手法

`madd44` 命令

`madd` 命令とは異なり，`madd44` 命令では積和演算に用いられるレジスタが固定されているため，レジスタファイルのポート数を増加させるのではなく，8 つのレジスタの出力を直接実行ユニットへと接続した．

積和演算については実行ユニットへ乗算器を 3 つと加算器を 2 つ追加し，実行ユニット内の既存の乗算器，加算器と組み合わせて実装を行った．`madd` 命令と同様に実行（EX）ステージのパイプライン化は行っておらず，演算は 1 クロックサイクルで実行される．

4. 設計環境

本研究は，RISC-V の設計記述言語として FSL を用いることに特徴がある．そこで本章では，まずハードウェア記述言語 FSL について述べ，その後に FSL を用いた設計の流れについて述べる．

4.1 ハードウェア記述言語 FSL

FSL は我々の研究室で開発中のハードウェア記述言語であり，生産性の高いハードウェア設計環境の実現を目指している．

FSL は NTT で開発されたハードウェア記述言語 SFL (Structured Function description Language) [8] の設計思想を受け継いでおり，Verilog HDL や VHDL などの RTL 記述と比較して抽象度が高く，ソフトウェアにより近い記述が可能である．また，プログラミング言語 Scala [9] の構文を採用しており，柔軟で簡潔にハードウェアを記述することができる．

RISC-V の設計に FSL を用いることで，本研究で実施するカスタム命令の追加やパイプライン構成の変更などを容易に行うことができる．また，FSL による RISC-V の記述例を蓄積しておくことで，後にハードウェア設計者が用途に応じた構成で RISC-V を利用したいという際に役立てることもできる．

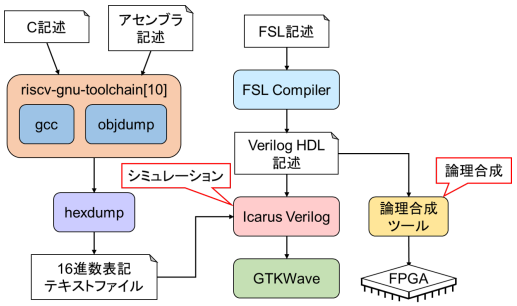

図 3: FSL を用いた設計フロー

4.2 設計の流れ

FSL を用いた設計のフローを図 3 に示す．FSL 記述から FSL Compiler を用いて Verilog HDL 記述に変換し，Icarus Verilog を用いてシミュレーションを行うことによって，動作検証とベンチマークを行う．論理合成を行う場合は，FSL Compiler が出力した Verilog HDL 記述を論理合成ツールへと入力する．

動作検証とベンチマークを行うプログラムは C 言語やアセンブラ言語を用いて記述し，公開されているツール群（riscv-gnu-toolchain [10]）を用いてコンパイル，アセンブルを行い，Icarus Verilog で読み取れる形式へと変換する．

なお，本研究ではカスタム命令に対応するようにツール群を改変している．

5. 評価と考察

本章では，実装した RISC-V について，FPGA への論理合成とベンチマークから得られた結果を用いた評価と考察について述べる．

5.1 論理合成

本研究では，Xilinx 社の FPGA である Artix-7 (XC7A35TICSG324-1L) をターゲットデバイスとして論理合成を行った．表 3 に Artix-7 のリソース量を示す．また，論理合成ツールには Xilinx 社の Vivado を用いた．

表 4 に実装した各 RISC-V コアの論理合成結果を示す．表 4 内の使用率は Artix-7（XC7A35TICSG324-1L）のリソース使用率を表している．また，消費電力の見積もりには，レジスタ，LUT，DSP にトグル率の設定を行い，値は全て 12.5%とした．

表 4 より，C2 のみで FF 数が増加していることがわかる．これは *rd* レジスタの値を引き渡すパイプラインレジスタを追加したためであると考えられる．また，C3 のみで DSP 数が増加している．これは，`madd` 命令では乗算器を追加しなかったのに対し，`madd44` 命令では 32 ビット乗算器を 3 つ追加したためであると考えられる．

設計した各 RISC-V コアのクリティカルパスを調査したところ，全てのコアで実行（EX）ステージがクリティカルパスとなっていた．また，最大動作周波数は C1 に対して，C2 では 3%，C3 では 4%の低下となり，最大動作周波数に大きな差は生じなかった．これは積和演算を行う部分の構成によるものだと考えられる．図 2 で示した通り，`madd` 命令では乗算器の出力を加算器へ接続している．また，`madd44` 命令では乗算器は並列に配置し，それぞれの出力を加算器へと接続している．このように乗算器の出力をさらに乗算器へと接続するというような構成にはなっていない．遅延時間の大きな乗算器が 1 つもしくは並列に配置されているため，最大動作周波数に大きな差が生じなかったと考えられる．

表 3: Artix-7 (XC7A35TICSG324-1L) のリソース量

LUT 数	FF 数	DSP 数
20,800	41,600	90

消費電力の面では，C3 が最大となった．DSP の消費電力が他のリソースよりも大きいことが原因であると考えられる．

5.2 ベンチマーク

ベンチマークの概要と得られた結果について述べる．

5.2.1 ベンチマークの概要

本研究ではカスタム命令として積和演算を採用したため，積和演算が多用される行列積演算プログラムをベンチマークソフトウェアとして採用した．

行列は $N \times N$ の正方行列で，各要素は 32 ビット整数とし，実装は主に C 言語で行った．行列積演算の実装は 3 重のループ文で行い，カスタム命令を利用する場合は，最内ループで行われる積和演算部をそれぞれのカスタム命令が使用できるように書き換えた．

各カスタム命令を用いた実装は，`madd` 命令ではインラインアセンブラを用いて記述を行い，`madd44` 命令では `madd44` 命令を使用した関数をアセンブリ言語で記述し，それを関数呼び出しする形で記述した．`madd44` 命令のソフトウェアを関数呼び出しの形式で記述したのは，関数の呼び出し手続きにより，コンパイラが関数の引数の値を `a0` レジスタから順に `a7` レジスタまでへと割り当てていくためである．

また，行列サイズが 4×4 と 8×8 の場合で動作検証を行い，シミュレーション結果とソフトウェアの実行結果が一致することを確認した．

5.2.2 ベンチマーク結果の比較

要した命令数とクロックサイクル数により，積和演算を行うカスタム命令導入の効果を比較する．

命令数の比較

実装したベンチマークソフトウェアをディスアセンブルし，解析したものを表 5 に示す．なお，表 5 内の N は行列の一辺の大きさを示している．表 5 からわかるように，カスタム命令を用いることで，実行命令数を削減することができている．理論的には，`madd` 命令では 4%，`madd44` 命令では 10%の命令数削減となった．

クロックサイクル数と実行時間の比較

行列サイズごとに，要したクロックサイクル数を測定した．クロックサイクル数の測定は，RISC-V をシミュレートする際のテストベンチに測定用のカウンタを設け，それを用いて行った．図 4 に C1 を基準とした各コアのクロックサイクル数と実行時間の比を示す．なお，メモリアクセスのレイテンシはキャッシュにヒットしたと仮定して，1 とした．また，実行時間は要したクロックサイクル数と表 4 内の最大動作周波数から算出したものである．

C2 では，実行命令数の削減率は 4%であるが，クロックサイクル数の削減率は約 2%にとどまり，動作周波数は C1 から約 3%低下しているため，最終的に実行時間は約 1%増加してしまった．通常の命令ではオペランドの数は 2 つであるが，`madd` 命令では 3 つとなっている．これにより，通常の命令では生じなかった，演算を行う命令と直前のロード命令の依存関係によるパイプラインストールが発生してしまい，クロックサイクル数の削減率が低下したと考えられる．

表 4: 論理合成結果

コアの名称 （カスタム命令）	LUT 数 （使用率）	FF 数 （使用率）	DSP 数 （使用率）	最大動作周波数 [MHz] （C1 比）	動的消費電力 [mW]	静的消費電力 [mW]
C1 （カスタム命令なし）	2,121 (10.20%)	1,432 (3.44%)	10 (11.11%)	51.55 (基準)	29	60
C2 （madd 命令）	2,778 (13.36%)	1,464 (3.52%)	10 (11.11%)	50.19 (0.97)	36	60
C3 （madd44 命令）	2,626 (12.63%)	1,432 (3.44%)	19 (21.11%)	49.59 (0.96)	38	60

表 5: 命令数の比較

コアの名称	ループ部の命令数	ループ回数	行列積演算全体の実行命令数	実行命令数比
C1 （カスタム命令なし）	25	N^3	$25 \times N^3$	1.00 (基準)
C2 （madd 命令）	24	N^3	$24 \times N^3$	0.96
C3 （madd44 命令）	90	$N^3/4$	$22.5 \times N^3$	0.90

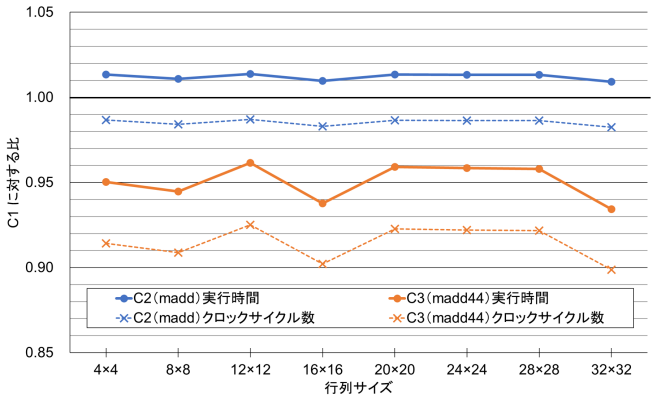

図 4: クロックサイクル数と実行時間の C1 比

C3 では，実行命令数の削減率は 10%であったが，クロックサイクル数は約 9%削減することができ，実行時間も約 5%削減することができた．madd44 命令を用いたことにより，ループ回数を他の実装の 4 分の 1 に削減することができ，分岐した際に発生するパイプラインフラッシュの回数も同様に削減できたことが，クロックサイクル数削減の要因であると考えられる．

6. おわりに

本稿では，積和演算を行う madd 命令と madd44 命令を搭載した RISC-V を設計し，FPGA への実装と行列積演算プログラムによるベンチマークを行い，それらの結果を用いて定量的に評価を行った．結果として，madd 命令では，実行命令数を約 4%削減できたが，パイプラインストールなどの影響により，実行時間は約 1%増加してしまった．madd44 命令では，実行命令数を約 10%削減することができ，実行時間も約 5%削減することができた．課題として，演算器のパイプライン化も含めたパイプライン構成全体の見直しや利用用途に対して高い効果をもたらすカスタム命令の仕様の追求などが挙げられる．

また，本研究ではカスタム命令を RISC-V のパイプライン上の実行（EX）ステージに実装したが，カスタム命令専用のパイプラインを設けて実装することも可能である．今後は，必要に応じて，カスタム命令専用のパイプラインを有した RISC-V の実装も検討する予定である．

参考文献

[1] 渡邊 誠也，名古屋 彰，“オブジェクト指向/関数型言語をベースとするハードウェア記述言語 FSL の設計，” 電子情報通信学会技術研究報告，RECONF2015-37，pp. 27–32，Sept. 2015.

[2] 今井 信志，渡邊 誠也，須山 敬之，名古屋 彰，“RISC-V へのカスタム命令実装手法の検討，” 電子情報通信学会 2019 年総合大会 情報・システムソサエティ特別企画 学生ポスターセッション予稿集，ISS-P-036，pp. 96，Mar. 2019.

[3] RISC-V Foundation，RISC-V Foundation | Instruction Set Architecture (ISA)，https://riscv.org/.

[4] A. Waterman and K. Asanovic，The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2，RISC-V Foundation，2017.

[5] freechipsproject，rocket-chip，https://github.com/freechipsproject/rocket-chip (accessed Dec. 2018).

[6] RISC-V BOOM，https://boom-core.org/.

[7] Chisel: Constructing Hardware in a Scala Embedded Language，https://chisel.eecs.berkeley.edu/.

[8] PARTHENON HOME PAGE，http://www.parthenon-society.com/archive/NTT/.

[9] The Scala Programming Language，https://www.scala-lang.org/.

[10] RISC-V Foundation，riscv-gnu-toolchain，https://github.com/riscv/riscv-gnu-toolchain (accessed Sept. 2018).