RISC-Vを用いた命令拡張のための プロセッサ開発環境の検討

中尾 怜史[1,a)]　武内 良典[2,b)]

概要：本研究では，特定用途向けプロセッサ開発の支援を行うことを目的とし，汎用プロセッサの命令拡張を効率的に行う環境を実現するための検討を行った．現在 AI 用のプロセッサや IoT には低消費電力化やセキュリティの強化などが求められており，それには特定用途に特化したプロセッサが必要とされている．しかしそのプロセッサの開発には費用や時間といったコストを要する．本報告ではプロセッサアーキテクチャとして Rocket-Chip，ISA は RISC-V を使用する．RISC-V には新規で追加可能なカスタム命令が存在する．Perl を用いて RISC-V とそのシミュレータに対し，自動でカスタム命令を定義するプログラムを作成した．

キーワード：RISC-V, Rocket-Chip

A Study of Processor Development Environment for Instruction Expansion Using RISC-V

Satoshi Nakao[1,a)]　Yoshinori Takeuchi[2,b)]

Abstract: In this study, we investigated the realization of an environment for efficient instruction expansion of general-purpose processors with the aim of supporting the development of processors for specific applications. Currently, processors for AI and the IoT are required to have low power consumption and enhanced security, which are specialized for specific applications. However, the development of such processors is costly in terms of design time and human resources. In this report, we use the Rocket-Chip processor architecture and RISC-V as the ISA, which has custom instructions that can be newly added to RISC-V. We created a program that automatically defines discriptions of custom instructions for RISC-V and its simulator using Perl.

Keywords: RISC-V, Rocket-Chip

1. 序論

近年，あらゆる機器にプロセッサを取り付け，データのやりとりを行う IoT(Internet of Things) 化が進んでいる．部屋の照明やエアコンといった家電の制御だけでなく，倉庫や工場といった製品管理をはじめ，建設現場や河川の堤防の監視など，常時電力供給が期待されない場所での運用も求められる．このため低消費電力化や，やりとりするデータの保護をするためセキュリティの強化が必要とされている．しかしこれらの機能を実現するために，毎回特定用途に特化したプロセッサの開発を行うと，膨大なコストが必要となる．

一方 RISC-V と呼ばれる ISA(Instruction Set Architecture) のプロセッサが提案されている [1], [5], [7]．RISC-V は UCB(カリフォルニア大学バークレー校) が開発したオープンな ISA である．RISC-V には 2 つの特徴があり，1 つ目はモジュール式の ISA を採用している点である．浮動小数点命令や圧縮命令など，命令セットを選択できる拡張

[1] 近畿大学大学院総合理工学研究科
[2] 近畿大学理工学部電気電子工学科
[a)] 2033340434n@kindai.ac.jp
[b)] takeuchi@ele.kindai.ac.jp

機能があり，最低限の命令のみを実装したハードウェアの開発が可能である．2 つ目の特徴は，カスタム命令が用意されていることである．カスタム命令は設計者が任意に定義できる命令であり，特定用途に向けた専用の命令を実装することが可能である．また RISC-V の実装として，Rocket-Chip が提案されている．Rocket-Chip は BSD ライセンスのフリーアーキテクチャであり，UCB と SiFive 社が開発している RISC-V を実装する SoC(System on Chip) である．Rocket-Chip には RISC-V のカスタム命令専用記述箇所や，また RoCC(Rocket Custom Coprocessor) と呼ばれるカスタム可能な外部アクセラレータが設けられており，開発したいハードウェアに合わせた回路を追加することが可能である．しかしながら，RISC-V の命令拡張を実現するためにはさまざまな拡張方法が存在し，本稿では各種拡張についての比較を行った．

ここで，本論文の構成を示す．第 2 節では RISC-V の命令と Rocket-Chip の概要について述べる．第 3 節ではカスタム命令と専用回路の実装手法を提案を行う．第 4 節で適用を行い，第 5 節で本稿のまとめを述べる．

2. RISC-V, Rocket-Chip 概要

本節では，本論文で用いる RISC-V と，これを ISA として実装するアーキテクチャ Rocket-Chip の概要について述べる．

2.1 RISC-V

RISC-V はレジスタ長として 32, 64, 128 ビットの指定を行うことが可能で，拡張機能を用いることで 16 ビットも選択できる拡張性のある ISA である [2], [3], [4]．RISC-V は従来の ISA と比べ，単純性を高めるよう設計されており，拡張命令を用いた命令の取捨選択からも，集積回路上にコンパクトな実装を見込むことができる．また，命令の単純性により CPI(命令あたりのクロックサイクル数) が少なくなり，高い性能を狙えるよう設計されている．アドレッシングモードとしては，32, 64 , 128 ビットを使用できる．

2.1.1 命令フォーマット

各命令セットの説明をする前に，RISC-V の命令フォーマットについて説明する．**図 1** に命令フォーマットを示す．RISC-V の命令ビットフィールドはレジスタ長が 32, 64, 128 ビットであっても 32 ビット固定であり，そのフォーマットは R, I, S, U の 4 種類に加え，S, U を発展させた B, J の 2 種類が用意されている．これにより実装する命令が使用するレジスタ数や，即値の有無によって使い分けることができる．rs1, rs2 はソースレジスタ，rd は演算結果を格納するデスティネーションレジスタ，imm は即値，opcode はオペコードを示している．全てのフォーマットにおいて，レジスタとオペコードのビット位置を共通にすることで，デコードが容易になるよう設計されている．

RISC-V で定義されているオペコードを**図 2** に示す．オペコードは 28 種類用意されており，命令タイプによって共通のオペコードが利用されている．共通するオペコードを持つ命令どうしは，funct7, funct3 に個別の値を定義することによって区別される．また，命令ビットフィールドの [1,0] ビット目はほぼ全ての命令が 0x3 の値で設定されており，C 拡張の命令のみがそれ以外の値を持つ．

2.1.2 命令セット

2.1.2.1 基本命令

基本命令は算術命令, 論理演算命令の整数値を扱う命令を含む．これに加えロード, ストア命令やジャンプ命令といった制御系命令も定義されているため，アプリケーションを動作させるために必要最低限な命令セットとなっている．また，演算で使用するレジスタは 0 の値を固定されたレジスタと，整数値を保持する 31 個の汎用レジスタが用意されている．

2.1.2.2 拡張命令

表 1 は 2019 年 12 月 13 日の命令セットドラフトの一覧である．M 拡張は乗除算命令，A 拡張はメモリに対して不可分の操作を行うアトミック命令，F, D, Q 拡張は単精度, 倍精度, 4 倍精度の浮動小数点命令である．また，組込みシステムのようなできるだけ小さなコア設計が求められるアーキテクチャのために，基本命令はそのままに汎用レジスタ数を半分に抑えた E 拡張が存在する．この拡張は 32 ビットのみしか対応していないが，基本命令と同じ標準 ISA とされている．

これ以外にもベクトル命令を含む V 拡張，SIMD 命令を含む P 拡張なども存在し，これらは今後実装される予定となっている．RISC-V は現在も改良が続いている ISA であると言える．

2.2 Rocket-Chip

図 3 に Rocket-Chip の概略図を示す．Rocket-Chip は，RISC-V を ISA とした CPU を含む Tile と呼ばれるコア部分と，AXI バスで接続されたメモリを持つ SoC アーキテクチャである [5]．コア数などを変更し，独自の構成を作成することが可能である．また，L2 キャッシュや IO へ接続部分には，SiFive 社が定義した TileLink と呼ばれるプロトコルが使われている．これにより，複数のマスターデバイスとスレイブデバイスとのコヒーレントな接続が実現されている．対応する RISC-V の命令は，基本命令, 拡張命令の M, A, F, D, C である．アドレッシングモードは 32, 64, 128 ビットに対応している．

Rocket-Chip はハードウェア記述言語として，Chisel[6] と呼ばれるプログラミング言語 Scala の埋め込み言語を用いる．Chisel ではクロック, リセット信号を考慮する必要がなく，java などのようにクラスを定義し用いることができる．Rocket-Chip を拡張する際は Chisel の理解が必要

Tpye	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	funct7							rs2					rs1					funct3			rd					opcode						
I	imm[11:0]												rs1					funct3			rd					opcode						
S	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
B	imm[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				imm[11]	opcode						
U	imm[31:12]																				rd					opcode						
J	imm[20]	imm[10:1]										imm[11]	imm[19:12]								rd					opcode						

図 1 命令フォーマット
Fig. 1 Instruction format

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD_FP	**custom0**	MISC_MEM	OP_IMM	AUIPC	OP_IMM_32	48b
01	STORE	STORE_FP	**custom1**	AMO	OP	LUI	OP_32	64b
10	MADD	MSUB	NMSUB	NMADD	OP_FP	**reserved**	**custom2/rv128**	48b
11	BRANCH	JALR	**reserved**	JAL	SYSTEM	**reserved**	**custom3/rv128**	>80b

図 2 オペコード
Fig. 2 Opecode

表 1 命令リスト
Table 1 Instruction List

Base	Version	Status
RV32I	2.0	Ratified
RV64I	2.1	Ratified
RV32E	1.9	Draft
RV128I	1.7	Draft
Extension	**Version**	**Status**
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
L	0.0	Draft
B	0.0	Draft
J	0.0	Draft
T	0.0	Draft
P	0.2	Draft
V	0.7	Draft

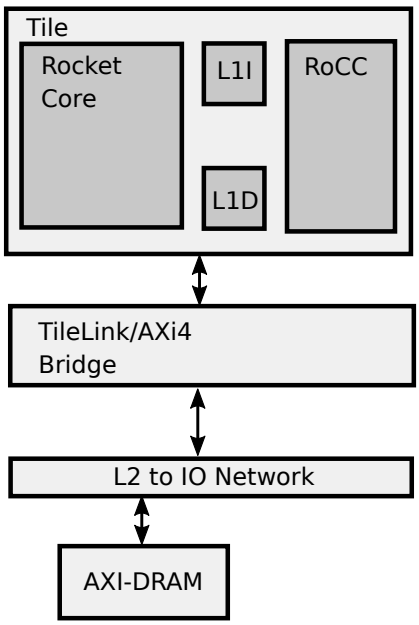

図 3 Rocket-Chip outline
Fig. 3 Outline of Rocket-Chip

となるが，Verilog 言語を埋め込みとして用いることもできる．

3. 実装手法

本稿ではこれら拡張性を持つ RISC-V と Rocket-Chip を用いることで，特定用途に特化したプロセッサの開発環境の検討を行った．**図 4** は提案手法で命令拡張にともない，修正が必要な部分を示したものである．ソフトウェア (SW) 部では RISC-V のアセンブラと，カスタム命令の動作確認に用いるシミュレータに手を加える．またハードウェア (HW) 部では，カスタム命令を Rocket Core 内部もしくは，アクセラレータとして実装するように追加記述する．

3.1 ソフトウェア部への追加

SW 部への追加は，C と C++のアセンブラを含む riscv-gnu-toolchain と，その動作確認を行うシミュレータ spike を含む riscv-tools と呼ばれるレポジトリに対しカスタム命令の定義を行った．spike へは，カスタム命令の動作記述と，命令の特定に使用する MATCH, MASK, DECLARE のマクロと，それらを組み合わせ作成した定義文を追加し

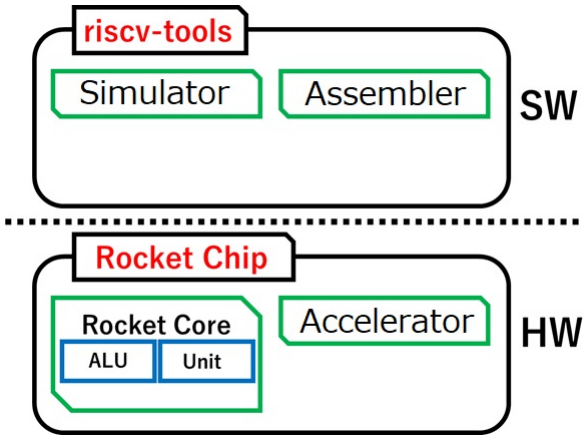

図 4 提案手法の概要
Fig. 4 Outline of Proposed Method

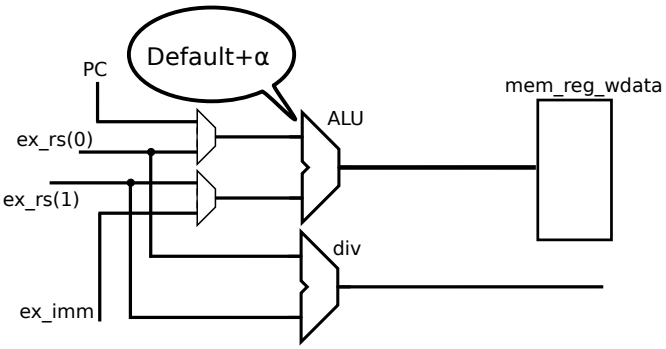

図 5 ALU の拡張
Fig. 5 Extension of ALU

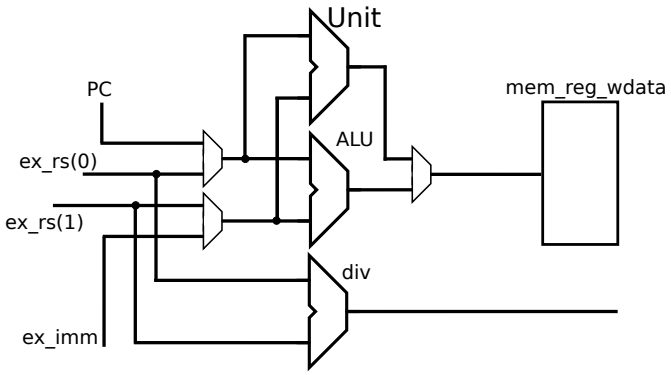

図 6 専用 Unit
Fig. 6 Dedicated Unit

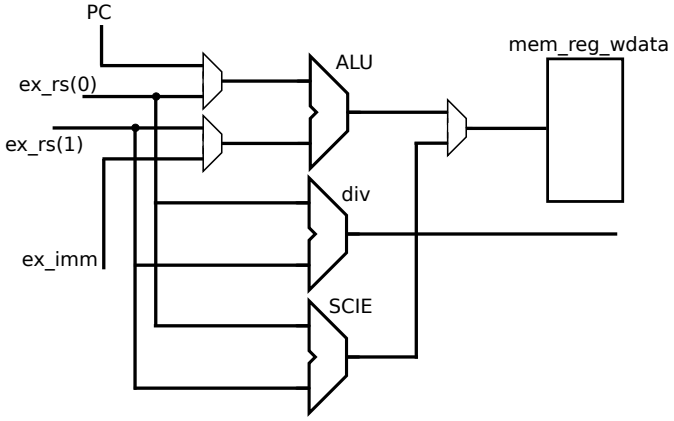

図 7 SCIE の拡張
Fig. 7 Extension of SCIE

た．またアセンブラへは，命令特定のマクロと，カスタム命令の定義文を追加した．

3.2 ハードウェア部への追加

HW 部への追加について述べる．Rocket-Chip への処理記述の追加には，1:ALU の拡張，2:専用 Unit を作成，3:SCIE の拡張，4:RoCC へ追加の 4 通りを検討する．また，共通する手順として，カスタム命令の名前と所属拡張の登録が必要である．

3.2.1 ALU の拡張

1 つ目は ALU を拡張する方法である．**図 5** に ALU の概略図を示す．ALU への処理構造を追加するには，扱える命令の名の追加と処理記述で，他の方法と比べ非常に容易である．

3.2.2 専用 Unit の作成

2 つ目は新しく専用 Unit を作成する方法である．**図 6** に専用 Unit の概略図を示す．I/O と処理記述を含むモジュールを定義し，ALU への入力信号を接続すればよい．また，メモリステージへの出力は，ALU との出力信号をマルチプレクサで選択させることで実装する．

3.2.3 SCIE の拡張

3 つ目は SCIE(Simple Custom. Instruction Extension) を拡張する方法である．**図 7** に SCIE の概要図を示す．SCIE は 2019 年頃に追加されたカスタム命令専用の処理構造であり，ALU の出力とマルチプレクサで選択されメモリステージへ出力される．実装可能な構造は，パイプライン，非パイプラインの 2 種類が可能である．実装するには，変数を変更し SCIE を使用可能状態にする必要がある．また記述には，Chisel を知らない人でも設計できるよう Verilog を使えるクラスを継承しているため，Verilog を用いて記述できる．

3.2.4 RoCC

最後に RoCC を用いる方法である．**図 8** に RoCC の概要を示す．RoCC はアクセラレータであり，複雑な処理を実行することで全体を高速処理することを目的として設計されている．RoCCInterface を介して CPU コアから，RS1, RS2 の値などを含むデータ cmd を受取り，演算結果

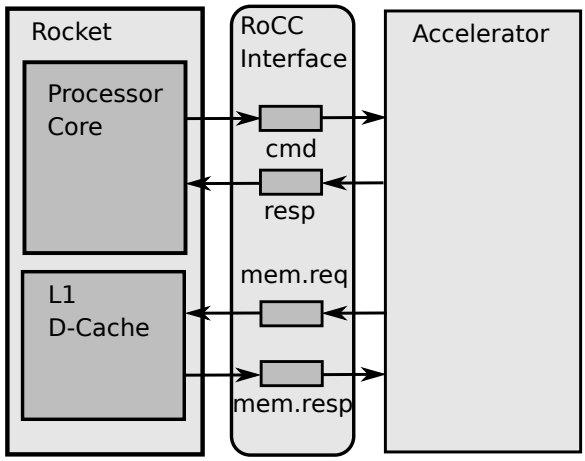

図 8 RoCC の概要
Fig. 8 Outline of RoCC

と RD のアドレスを含むデータ resp を返す．データのやり取りは，valid, ready, busy 信号を用いて制御されている．また，RoCC から Rocket コアのキャッシュメモリや FPU に接続し，値を送受信して演算処理を行わせることも可能である．

RoCC は，カスタム命令用のオペコード 1 つにつき 1 つの RoCC を実行することができる．Rocket-Chip では，既にサンプルの RoCC が実装されているので，記述を上書きして実装する必要がある．また RoCC は，status レジスタの mstatus_xs に 0 以外の値を設定し，カスタム命令を実行することで動作する． **図 9** は RoCC で動作する命令のフォーマットである．RoCC で動作する命令のフォーマットは R タイプのみである．funct3 はカスタム命令が使用できるレジスタを表す．また，funct7 を用いることで処理の選択が可能であり，1 つのオペコードでも複数の回路を実装することができる．

3.3 実装例

以下では 3.1, 3.2 で紹介した提案手法を，abs 命令を例に説明する．RS1 の絶対値を計算する．funct7=0x01, fucnt3=0x07 で custom-0 のオペコードを使用する整数値のみを扱う命令である．

3.3.1 ソフトウェア部の記述

3.3.1.1 spike

spike への abs 命令の動作記述は以下のように定義した．

```
sreg_t tmp = sext_xlen(RS1);
if(tmp < 0) WRITE_RD(sext_xlen(-tmp));
else WRITE_RD(sext_xlen(tmp));
```

まず, 符号付整数レジスタ tmp に RS1 の値を符号付整数として格納する．次に tmp の値を比較し，負数なら正数を変換し，それ以外なら値をそのままに符号付整数として RD レジスタに書き込む．

次に命令特定用マクロを作成する．**図 10** に abs 命令のビットフィールドと作成したマクロを示す．MATCH, MASK のマクロは，命令ビットフィールドを 16 進数に変換した値を並べたコードとなっている．MATCH はデコードに使用するビットフィールドの情報を示したものである．funct7, funct3, opecode の値をそのまま，それ以外を 0 とした値となる．abs 命令を例にすると 0x0200700B となる．

一方 MASK は，デコードで使用するビット位置を特定するためのもので，funct7, funct3, opcode の全てのビットを 1 とし，それ以外を 0 とした時の値となる．abs 命令では 0xFE00707F となる．

DECLARE は MATCH, MASK と命令の名前を引数に定義するマクロで, abs 命令を例にすると以下の記述となる．

```
DECLARE_INSN(abs,MATCH_ABS,MASK_ABS)
```

3.3.1.2 アセンブラ

次にアセンブラへのカスタム命令の追加をする．命令特定用のマクロは，spike で実装した 3 種類を用いた．

アセンブラへの abs 命令の定義は以下のように記述した．

```
{"abs", "I", "d,s,t", MATCH_ABS, MASK_ABS,
match_opcode, 0 }
```

順に命令の名前, 所属する拡張命令名，使用するオペランド, 参照する MATCH などの命令を特定する要素，pinfo 情報となっている．使用するオペランドはマクロを用いて指定する．また，pinfo 情報は 0 と INSN_ALIAS のいずれかの値を取る．

3.3.2 ハードウェア部の記述

3.3.2.1 ALU の拡張

図 11 に ALU に追加した abs 命令の処理記述を示す．ALU で扱う命令の数を 5 ビットへと拡張し，値が 16 に abs 命令登録している．

処理記述では，rs1 の MSB を全てのビットと排他的論理和を行い，0 か 1 を加算する．出力は他演算の出力をマルチプレクサを通して選択させている．

3.3.2.2 専用 Unit の追加

図 12 に abs 命令のみを処理する Unit の記述を示す．ALU の拡張記述に加えて，モジュールと I/O を新しく定義している．

3.3.2.3 SCIE の拡張

図 13 に SCIE へ追加した abs 命令の拡張記述を示す．verilog で処理記述を追加している．

3.3.2.4 RoCC

図 14 に RoCC へ追加した abs 命令の記述を示す．RoCC の IO は，定義されたモジュールの継承で使用する．割り込み処理や，Dcache は使用していない．

4. 検討したカスタム命令

本節では，第 3 節で提案した手法を用いて実装を行い，比較し検討する．**表 2** に実装したカスタム命令を示す．実装した命令は全て整数命令として実装した．また fact, pow

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct7							rs2					rs1					xd	xs1	xs2	rd					opcode							
roccinst[6:0]							rs2					rs1					1	1	1	rd					custom0/1/2/3							

図 9 RoCC で動作する命令フォーマット
Fig. 9 RoCC's Instruction Format

ビット番号	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12
ABS	0	0	0	0	0	0	1	rs2					rs1					1	1	1
MATCH	0				2			0					0				7			
MASK	F				E			0					0				7			

ビット番号	11	10	9	8	7	6	5	4	3	2	1	0
ABS	rd				0	0	0	1	0	1	1	
MATCH	0				0			B				
MASK	0				7			F				

図 10 abs 命令のビットフィールド
Fig. 10 abs Instruction's Bit field

```
val SZ_ALU_FN = 5
def FN_X    = BitPat("b?????")
def FN_ABS  = UInt(16)
def isAbs(cmd: UInt) = cmd === FN_ABS

val abs = (io.in1 ^ Fill(xLen, io.in1(xLen-1))) +
io.in1(xLen-1)
val logic_abs = Mux (isAbs (io.fn), abs, shift_logic)
val out = Mux(io.fn === FN_ADD || io.fn === FN_SUB,
io.adder_out, logic_abs)
```

図 11 ALU の拡張記述
Fig. 11 Extended description of ALU

```
val SZ_ALU_FN = 5
def FN_X    = BitPat("b?????")
def FN_ABS  = UInt(16)
def isAbs(cmd: UInt) = cmd === FN_ABS

class AbsUnit(implicit p: Parameters) extends
CoreModule()(p) {
  val io = new Bundle {
    val dw = Bits(INPUT, SZ_DW)
    val fn = Bits(INPUT, SZ_ALU_FN)
    val in1 = UInt(INPUT, xLen)
    val out = UInt(OUTPUT, xLen)   }
//ABS
val abs = (io.in1 ^ Fill(xLen, io.in1(xLen-1))) +
io.in1(xLen-1)   io.out := abs
if (xLen > 32) {
    require(xLen == 64)
    when (io.dw === DW_32) { io.out := Cat(Fill(32,
abs(31)), abs(31,0)) }
 }
}
```

図 12 専用 Unit の記述
Fig. 12 Description of Dedicated Unit

命令は，**図 15** に示すカウンタを用いてループ処理をさせることで実装した．また SW 部へのカスタム命令の追加は，3.2 の処理を自動で行う Perl プログラムを作成した使用した．

```
setInline("SCIEUnpipelined.v",
  s"""
    |module SCIEUnpipelined #(parameter XLEN = 32) (
    |   input  [${SCIE.iLen-1}:0] insn,
    |   input  [XLEN-1:0] rs1,
    |   input  [XLEN-1:0] rs2,
    |   output [XLEN-1:0] rd);
    |
    | /* Determine if the custom instruction is 0 or 1 */
    | wire custom0 = !insn[5];
    |
    | /* Mux the operands. */
    | wire signed [XLEN-1:0] rhs = rs1;
    |
    | /* Perform the computation. */
    | wire signed [XLEN-1:0] abs = rhs >= 0 ? rhs :
~rhs+1;
    | wire [XLEN-1:0] result = custom0 ? abs : 0;
    |
    | /* Drive the output. */
    | assign rd = result;
    |
    |endmodule
  """.stripMargin)
```

図 13 SCIE の拡張記述
Fig. 13 Extended description of SCIE

表 2 実装命令
Table 2 Implement Instruction

name	behavior
abs	$rd = \|rs1\|$
inc	if$(rs1 = 0) : rd = rs1 + 1$,else: $rd = rs1 + rs2$
dec	$if(rs1 = 0) : rd = rs1 - 1, else : rd = rs1 - rs2$
pow	$rd = rs1\hat{}rs2$
fact	$rd = rs1!$

4.1 動作確認

カスタム命令の動作確認は，riscv-tests レポジトリと C 言語から作成したベンチマークと，RTL シミュレータ verilator を用いた．シミュレータの出力ファイルから，カスタム命令の発行と演算結果を確認した．

RoCC による命令実行は，波形データを出力させ GTKWave を用いた確認も行った．**図 16** に fact 命令で 10!を実行した時の波形データを示す．valid 信号が真の間 CPU

```
class AbsRoCCModuleImp(outer: AbsRoCC)(implicit p:
Parameters) extends LazyRoCCModuleImp(outer)
 with HasCoreParameters {
 val cmd = Queue(io.cmd)
 val funct = cmd.bits.inst.funct
 val doAbs = funct === UInt(0)
 val doRead = funct === UInt(1)

 val abs = (cmd.bits.rs1 ^ Fill(xLen, cmd.bits.rs1 >>
(xLen-1))) + (cmd.bits.rs1 >> (xLen-1))

 val doResp = cmd.bits.inst.xd
 val stallResp = doResp && !io.resp.ready
 cmd.ready := !stallResp
 io.resp.valid := cmd.valid && doResp

 io.resp.bits.rd := cmd.bits.inst.rd
 io.resp.bits.data := abs
 io.busy := cmd.valid || busy.reduce(_||_)
 io.interrupt := Bool(false)
 io.mem.req.valid := Bool(false)
}
```

図 14 RoCC の記述
Fig. 14 Description of RoCC

```
when(stateReg === process){
 when(count === 0.U){
  stateReg := output
 } .otherwise{
  fact := fact * count
  count := count - 1.U
 }
}
```

図 15 カウンタの記述例
Fig. 15 Counter description example

コアから RS1 の値 10 が渡され，stateReg が 0x01 の間演算を実行していることが確認できた．

4.2 比較

表 3, **表 4**, **表 5**, **表 6**, **表 7** に Chisel,Verilog 記述での各文字, 行数を示す．ALU を拡張する方法は，内部の拡張のみとなるため他の方法と比べても最も少ない記述量であった．専用 Unit を作成する方法は，新しくモジュールを作成し配線を接続する必要があるため，Chisel を用いた記述では 2 番目に記述量が多くなった．SCIE を拡張する方法では，予め用意されたカスタム命令専用のモジュールのため，専用 Unit を作成する方法と比較すると，Chisel 記述は少なくなる．しかし,Verilog での記述量は RoCC への記述量に次いで 2 番目に多い．これは SCIE は直接 Verilog 記述しており，他の実装法とは違い Chisel からの変換ではないからと考えられる．RoCC を用いる方法では，RocketCore との制御信号やレジスタのアドレスなどの記述により両言語の記述量は共に最も多くなった．Chisel と Verilog との比較では，全体的に文字, 行数は Chisel の方が少なることがわかる．

5. まとめ

RISC-V への命令追加は，マクロを組み合わせ定義文を記述することで追加することができた．拡張したプロセッサの動作の確認は C,C++で作成したプログラムを使用でき新たな HDL の記述等は不要であり，動作確認が効率的に行える．またベクトル命令など拡張命令のバリエーションの増加から，RISC-V は開発する対象の ISA として採用しやすい．

これら Rocket-Chip と RISC-V へのカスタム命令の実装の難度から，独自のプロセッサへの開発環境手段としては大きいと感じた．また Chisel を用いることで，Verilog よりも約 1/10 ほど記述量を抑えることができるため，短い時間で開発することがで可能である．

今後の方針として，RISC-V へ追加したカスタム命令の自動実装プログラム拡張し，その処理回路も生成，実装するプログラムを作成する．また SCIE の有効性として，パイプライン構造を含む回路を実装し RoCC へ実装しパターンと比較を行う．

謝辞 本研究は JSPS 科研費 JP20H00606 の助成を受けている。また，東京大学大規模集積システム設計教育研究センターを通し、メンターグラフィクス株式会社、シノプシス株式会社の協力で行われたものである．

参考文献

[1] 金森拓斗, 宮崎広夢, 吉瀬謙二,「組み込みシステムに適した RISC-V ソフトプロセッサの設計と実装」, 情処究報、pp. 1–8, Vol.2020-EMB-54, 2020.
[2] A.Waterman, K.Asanovic, SiFive Inc.,「The RISC-V Instruction Set Manual Volume I:Unprivileged ISA Document Version 20191213-draft」, University of California, Berkeley(2021).
[3] David A.Patterson, A.Waterman,「RISC-V 原典 オープンアーキテクチャのススメ」, 日経 BP(2018).
[4] David A.Patterson, John L.Hennessy,「COMPUTER ORGANIZATION AND DESIGN THE HARDWARE/SOFTWARE INTERFACE RISC-V EDITION」, Morgan Kaufmann(2017).
[5] Krste Asanović, et al.,「The Rocket Chip Generator」, Technical Report No. UCB/EECS-2016-17 (2016).
[6] Jonathan Bachrach, et al.,「Chisel: Constructing Hardware in a Scala Embedded Language」, DAC Design Automation Conference 2012, p1216–1225 (2012).
[7] Christopher Celio, et al.,「BOOM v2: an open-source out-of-order RISC-V core」, Technical Report No. UCB/EECS-2017-157 (2017).

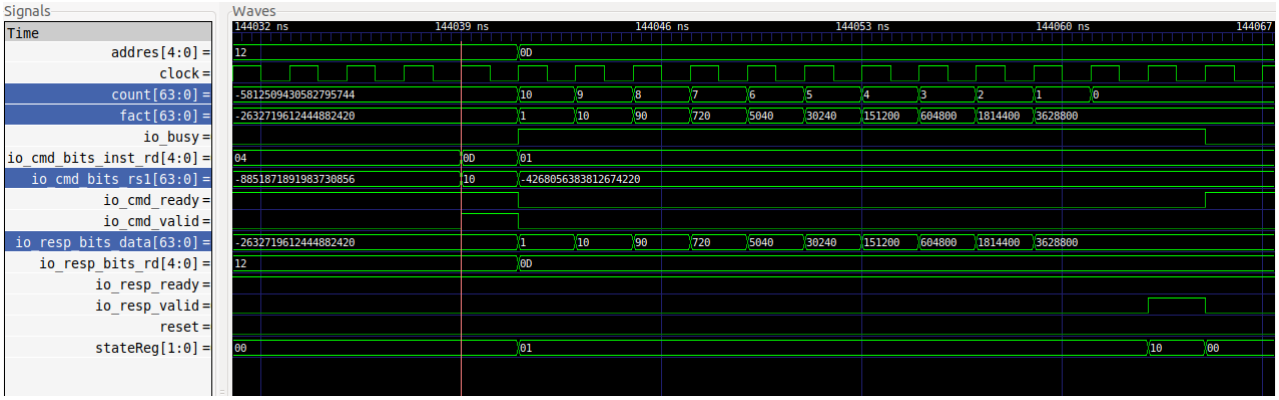

図 16 RoCC に実装した fact 命令の波形データ
Fig. 16 Description of RoCC

表 3 abs 命令における文字行数比較
Table 3 Comparison of the number of characters and lines in the abs instruction

行数/文字数	Base	ALU	Unit	SCIE	RoCC
Chisel	46511/1844003	46522/1844661	46541/1845205	46536/1844862	46549/1845185
Verilog	246692/16964301	246720/16967397	246747/16969446	246960/16977016	248157/17073546

表 4 inc 命令における文字行数比較
Table 4 Comparison of the number of characters and lines in the inc instruction

行数/文字数	Base	ALU	Unit	SCIE	RoCC
Chisel	46511/1844003	46522/1844663	46542/1845238	46537/1844893	46567/1845603
Verilog	246692/16964301	246712/16966475	246733/16967896	246961/16977055	248217/17072174

表 5 dec 命令における文字行数比較
Table 5 Comparison of the number of characters and lines in the dec instruction

行数/文字数	Base	ALU	Unit	SCIE	RoCC
Chisel	46511/1844003	46522/1844664	46542/1845240	46537/1844899	46567/1845603
Verilog	246692/16964301	246740/16969558	246761/16970982	246973/16978333	248217/17072174

表 6 pow 命令における文字行数比較
Table 6 Comparison of the number of characters and lines in the pow instruction

行数/文字数	Base	RoCC
Chisel	46511/1844003	46577/1845843
Verilog	246692/16964301	248209/17183549

表 7 fact 命令における文字行数比較
Table 7 Comparison of the number of characters and lines in the fact instruction

行数/文字数	Base	RoCC
Chisel	46511/1844003	46577/1845863
Verilog	246692/16964301	248209/17183549