Vol.2019-ARC-235 No.12
Vol.2019-SLDM-187 No.12
Vol.2019-EMB-50 No.12
2019/3/17

RISC-Vを基本とする構成可変プロセッサのための ハードウェア開発環境の検討

竹谷 凌[1,a)]　武内 良典[2,b)]

概要：本研究では，RISC-V を基本とする応用に特化した処理に適したアーキテクチャの構成変更が容易であるプロセッサのハードウェア (HW) 開発環境を提案する．RISC-V はオープンな命令セットアーキテクチャ (ISA) として注目度が高まっている．現在は Internet of Things (IoT) 時代となり，機器の少量多品種化が進んでいる．また，これらの機器には，マイクロプロセッサが搭載されている．IoT 機器では，それぞれの応用に特化した処理が求められるようになっている．したがって，プロセッサのアーキテクチャを容易に変更できる構成可変プロセッサへの期待が高まっている．そこで，本稿では，RISC-V の ISA を基本とした構成可変であるプロセッサの HW 開発環境の構築を行った．また，生成されたプロセッサの網羅的な確認を行い，HW の性能評価を行った．また，命令拡張を行う際の従来手法との設計工数の比較を行った．

キーワード：RISC-V, 構成可変プロセッサ

1. 序論

近年，Internet of Things (IoT) 時代を迎えており，あらゆる機器にマイクロプロセッサが搭載されるようになってきている．また，IoT 機器は少量多品種であり，それぞれの機器の応用に特化した処理が求められる．したがって，IoT 機器のためのプロセッサとして，命令セットアーキテクチャ (Instruction Set Architecture;ISA) を容易に変更できる構成可変プロセッサへの期待が高まっている．

一方，オープンな ISA としてカリフォルニア州立大学バークレー校により提案された RISC-V [1] の注目度が高まっている．RISC-V は従来の ISA である MIPS や x86 に比べ，複雑な命令が存在せず，シンプルな命令セットとなっている．また，RISC-V の命令セットシミュレータ，コンパイラ等のソフトウェア (SW) 開発環境である riscv-tools [2] は，ISA の拡張が容易であり，命令のビヘイビア記述とオペランド，オペコードを定義することで命令の追加が可能である．しかし，RISC-V の ISA を持つハードウェア (HW) 開発環境は，主に Rocket-Chip [3]，BOOM [4] などが提案されているが，ISA の拡張には，ハードウェア記述言語 (Hardware Description Language;HDL) に変更を加える必要がある．HDL を直接修正追加しての ISA 拡張では，命令の動作記述，オペランド，オペコードだけでなく，ステートマシンやファンクションブロックなど多くの記述追加が必要になる．記述量が多いと，命令拡張にかかる工数が多くなり，複数箇所に記述を加えるため，矛盾のない拡張が困難になる．また，プロセッサ全体の記述を理解した上でのコードの解析が必要になる．

そこで，本研究では，シンプルで拡張性の高い ISA である RISC-V を基本とした構成可変プロセッサのための HW 開発環境を提案する．そのための HW 開発環境の生成のためのツールとして ASIP Meister [5] を用いる．ASIP Meister は特定用途向き命令セットプロセッサの対話型開発環境である．独自のプロセッサ仕様記述からプロセッサの制御論理等を自動的に決定し，論理合成可能なプロセッサの HDL を自動生成することが可能である．したがって，ASIP Meister を用いることで HW 開発環境への命令拡張をより簡単に行うことができる．

本稿では，ASIP Meister を用いて RISC-V の ISA を持ったプロセッサの実装を行った．また，設計したプロセッサに対して設計品質の計測，正しい挙動の網羅的な確認，HDL に命令追加する場合との必要記述量の比較を行った．

本稿の構成は次の通りである．2 節では，RISC-V の ISA と開発環境についてとプロセッサ自動設計手法について述べる．3 節では，HW 開発環境の実装方法について述べる．4 節では，設計した HW 開発環境の評価について述べる．5 節では，まとめと今後の課題について述べる．

[1] 大阪大学 大学院情報科学研究科
[2] 近畿大学 理工学部
[a)] r-taketn@ist.osaka-u.ac.jp
[b)] takeuchi@ele.kindai.ac.jp

Vol.2019-ARC-235 No.12
Vol.2019-SLDM-187 No.12
Vol.2019-EMB-50 No.12
2019/3/17

2. 研究背景

本節では，RISC-V の ISA について説明し，オープンである SW 開発環境と HW 開発環境について説明する．また，プロセッサの自動生成手法について説明する．

2.1 RISC-V

2.1.1 RISC-V の命令セット

命令セットは，基本命令と拡張命令によって構成されている．基本命令は，RV32I，RV64I，RV128I が存在し，それぞれ 32，64，128 ビットのデータ幅の整数命令セットである．RV32I の命令フォーマットを図 1 に示す．図 1 内の funct3，funct7，opcode はオペコードであり，命令ごとに固定値である．rs1，rs2 はソースレジスタ，rd はディスティネーションレジスタである．imm は即値である．

R-type は，2 つのソースレジスタとディスティネーションレジスタを扱う命令に用いる．I-type は，ソースレジスタと即値とディスティネーションレジスタを扱う命令に用いる．S-type は，2 つのソースレジスタと即値を用いる命令に用いる．U-type は，即値とディスティネーションレジスタを用いる命令に用いる．B，J-type は，それぞれ S，U-type の即値フィールドの符号ビットを最上位と最下位に割り当てることで符号ビットを別にデコードすることを可能にしている．また，参照するアドレスは最低の 16 ビットのアドレス幅でも 2 の倍数になるので，即値の最下位ビット imm[0] は 0 に固定し，命令には割り当てていない．その分，imm[12] を用いることができ，より大きいアドレスへの参照を可能にしている．B-type は条件分岐命令，J-type は無条件分岐命令に用いられている．また，6 つのフォーマット全体で命令内でのレジスタ，オペコードのフィールド位置が固定されている．これにより，命令をデコードするためのマルチプレクサの数を減らすことができる．

次に，RV32I の命令表を表 1 に示す．表 1 中の Mem はデータメモリで，pc はプログラムカウンタである．ロードストア命令は 8，16，32 ビットのデータをロード，ストアする命令が存在する．加減算，論理演算，シフト演算，比較，条件分岐，ジャンプの命令が存在する．RISC-V の基本命令は，MIPS や x86 等の過去の ISA に比べ，複雑な命令が省略されており，シンプルで独立性の高い命令セットとなっている．

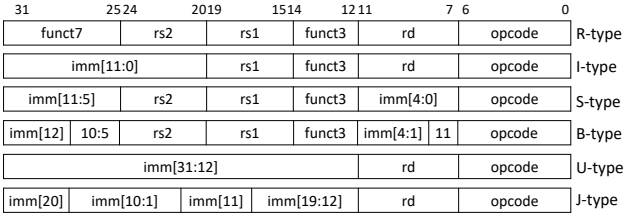

図 1: RV32I の命令フォーマット

表 1: RV32I の命令セット

Inst. name	Format	Behavior
L(B|H|W)(U)	I	rd = Mem[rs1+imm]
S(B|H|W)(U)	I	Mem[rs1+imm] = rs2
SLL(I)	R|I	rd = rs1 << (rs2|imm)
SR(L|A)(I)	R|I	rd = rs1 >> (rs2|imm)
ADD(I)	R|I	rd = rs1 + (rs2|imm)
SUB	R	rd = rs1 - rs2
LUI	U	rd = imm[31:12]&12b0
AUIPC	U	rd = pc + [imm[31:12]&12b0]
XOR	R|I	rd = rs1 xor (rs2|imm)
OR	R|I	rd = rs1 or (rs2|imm)
AND	R|I	rd = rs1 and (rs2|imm))
SLT(U)(I)	R	rd = (Unsigned) rs1 < (rs2|imm)
BEQ	B	if(rs1=rs2) pc=pc+imm
BNE	B	if(rs1≠rs2) pc=pc+imm
BLT(U)	B	if((Unsigned) rs1<rs2) pc=pc+imm
BGE(U)	B	if((Unsigned) rs1≥rs2) pc=pc+imm
JAL	J	rd=pc，pc=pc+imm
JALR	I	rd=pc，pc=rs1+imm

拡張命令の例としては，乗除算命令の RV32M，ベクトル演算命令の RV32V，浮動小数点演算命令の RV32F などが存在する．基本命令の上に必要な拡張命令を適宜載せて扱う構成となっている．また，アドレッシングモードとしては，32，64，128 ビットが存在する．レジスタ数は 32 であり，演算命令はレジスタを介して行われる．また，拡張命令を実装するフィールドが残っており，ユーザーが自由に命令拡張を行うことができる．

2.1.2 RISC-V の SW 開発環境

RISC-V の SW 開発環境は多数公開されており，主要なものは riscv-tools [2] というレポジトリにまとめられている．SW 開発環境には，GCC 等のコンパイラや OS への対応 SW，命令セットシミュレータの環境が存在し，自由に使用することができる．

riscv-tools の命令セットに命令拡張を行う手順を次に示す．まず，命令のオペコードとオペランドの位置とオペコードのビット値を定義する．次に，命令のビヘイビア記述を追加する．以上の追加を行い，開発環境をビルドし直すことで命令拡張が可能である．

以上より，RISC-V の SW 開発環境は，容易に命令拡張が可能な開発環境である．

2.1.3 RISC-V の HW 開発環境

次に，オープンに公開されている RISC-V の HW 実装の中でも主要な Rocket-Chip，BOOM，VexRiscV について説明する．

2.1.3.1 Rocket-Chip [3]

Rocket-Chip は，Rocket という RISC-V の ISA を持つ CPU のコアをチップ化した HW 実装である．RISC-V の ISA のほぼ全てを実装している，5 ステージのパイプライン IN-Order 実行のアーキテクチャである．実装は，Chisel [6]

Vol.2019-ARC-235 No.12
Vol.2019-SLDM-187 No.12
Vol.2019-EMB-50 No.12
2019/3/17

という Scala をベースとした HDL で記述されている．

実装に拡張を加える際に，Chisel のコードの理解が必要になる．また，HDL に命令拡張を行う場合は，命令の動作記述だけでなく，パイプラインなどのステートマシン，ファンクションブロックにも拡張を行う必要がある．したがって，難解なコードの解析と HDL の理解が必要になり，命令拡張における設計工数も非常に多くなる．

2.1.3.2 BOOM [4]

BOOM は，アウトオブオーダー実行のアーキテクチャであり，Rocket に比べ，複数命令を同時に発行できるため，高性能で高い動作周波数で動作するアーキテクチャとなっている．また，面積や遅延時間も Rocket より高く，高性能な HW 実装となっている．

実装は，Rocket と同じく Chisel で書かれている．しかし，実装が非常に複雑で解析することが難しく，命令拡張をする際にも，多くのコード解析の時間と記述量が求められる．

2.1.3.3 VexRiscV [7]

VexRiscV は，RV32I と RV32M の ISA を持つプロセッサの HW 開発環境であり，FPGA の開発環境も含め，オープンに公開されている．5 ステージのパイプラインで，基本的な演算器，メモリを用いており，シンプルな構造となっている．実装は，SpinalHDL という HDL を用いて記述されている．したがって，命令拡張を行うには同様に，HDL の解析を行う必要がある．

本研究では，より命令拡張が少ない記述量かつ容易に行うことが出来る HW 開発環境を提案する．そのために，プロセッサ自動生成環境を用いて RISC-V の HW 開発環境の構築を行う．

2.2 プロセッサ自動生成環境

2.2.1 ASIP Designer

独自の記述方式を用いてプロセッサの仕様を記述し設計する手法として Synopsys 社の ASIP Designer がある．ASIP Designer は，元々ドイツの Aachen 工科大学で研究されていたプロセッサ生成ツール LISA を元に Synopsys 社が商用化したものである．ASIP Designer はプロセッサを自由に設計することができ，HDL だけでなく，コンパイラ，命令セットシミュレータまで自動生成することができる．しかし，使用する HW リソースは定義されておらず，自分で実装しなければならない．また，マイクロ動作記述だけでなく，パイプラインの論理制御も自分で定義する必要がある．

2.2.2 ASIP Meister [5]

ASIP Meister は，特定用途向けプロセッサを設計するための対話型環境である．

ASIP Meister を用いたプロセッサ設計は次の 7 工程からなる．各工程について説明する

アーキテクチャ・パラメータの設定 パイプラインの段数，各ステージの情報を設定する．

リソース宣言 プロセッサが用いる演算器，ストレージ等のリソースを宣言する．ASIP Meister は HW リソースモデルとしてパラメタライズされたリソースモデルの Flexible Hardware Model (FHM) を用いている．FHM はリソースのビット幅などを容易にパラメータ化している HW リソースであり，ASIP Meister は，FHM のデータベースを読み込んで，パラメータのみを設定することで，使用することができる．FHM は必要になるリソースがあれば新たに定義してデータベースに追加することが可能である．

ストレージ仕様定義 レジスタ，レジスタファイル，メモリなどのストレージのビット幅，用途，レジスタ数等の仕様を設定する．

入出力インターフェースの定義 プロセッサのエンティティと入出力ポートの設定を行う．

命令タイプ・命令セット・例外の定義 命令タイプと命令セットを定義する．命令ごとにオペコード，オペランドを定義する．

アセンブラの生成 ASIP Meister はどのような命令体系にも対応可能なメタアセンブラが付属している．メタアセンブラに渡すためのプロセッサのアセンブラ記述を生成する．アセンブラ記述は，評価用のアセンブリプログラムを実行する際に用いる．

マイクロ動作記述の定義 全命令のマイクロ動作記述をパイプラインステージごとに記述する．マイクロ動作記述はプロセッサ記述言語 (Processor Description Language) [8] を用いて記述する．

HDL の生成 以上の情報から論理合成可能な HDL 記述を生成する．

以上の手順で，任意のプロセッサの HDL の生成が可能である．命令拡張における設計工数の減少度の評価も 4 節にて行っている．

2.2.3 ビヘイビア記述からのマイクロ動作記述生成手法

また，ASIP Meister のマイクロ動作記述を命令のビヘイビア記述から生成する手法 [9] が提案されている．図 2 に ADD 命令のビヘイビア記述とマイクロ動作記述の生成例を示す．ビヘイビア記述は命令の挙動を示す記述でパイプラインステージごとの記述，使用するリソース，ファンクションの記述を必要としない記述である．本生成手法では，命令のビヘイビア記述と HW リソースのデータベースを入力として与える．ビヘイビア記述を構文解析し，演算子と被演算子等に分ける．それぞれの演算子，被演算子ごとに使用する HW リソースとファンクションが自動で決定される．最後に，パイプラインの構成に応じて，マイクロ動作記述を生成することができる．本手法を用いることで，マイクロ動作記述より更に少ない記述量で命令拡張を

Vol.2019-ARC-235 No.12
Vol.2019-SLDM-187 No.12
Vol.2019-EMB-50 No.12
2019/3/17

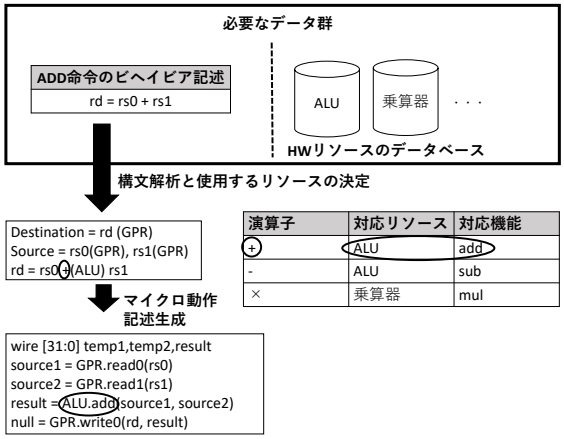

図 2: ADD 命令のビヘイビア記述からのマイクロ動作記述生成

行うことが可能である．したがって，従来のプロセッサ設計手法と比べてもより少ない設計工数で命令拡張ができる．

3 節では，実装する構成可変プロセッサ開発環境についてと ASIP Meister を用いて実装した RV32IM の HW 開発環境の仕様と実装内容について説明する．

3. 提案手法

本節では，提案する構成可変プロセッサ開発環境について説明する．

3.1 実装するプロセッサ開発環境の概要

2 節の背景から，本研究で提案するプロセッサ開発環境の全体図を図 3 に示す．SW 開発環境は，容易に命令拡張が行える riscv-tools を用いる．それに対して，HW 開発環境は，ASIP Meister を用いて新たに構築を行う．そして，ASIP Meister と riscv-tools の命令定義の同期をとり，HW・SW 開発環境に同時に矛盾ない命令拡張が行えるプロセッサ開発環境を構築する．本研究では，RISC-V の ISA を基本としたプロセッサの HW 開発環境の構築を ASIP Meister を用いて行った．

3.2 HW 開発環境の仕様

ASIP Meister を用いて，プロセッサの構築を行った．実装した ISA は RISC-V の 32 ビット基本命令である RV32I と乗除算命令 RV32M である．5 ステージのパイプライン

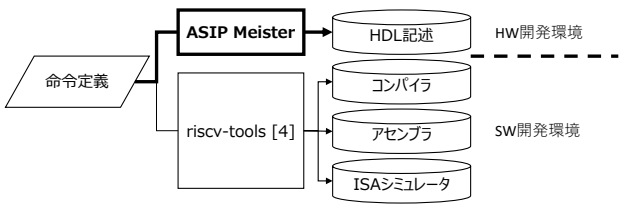

図 3: プロセッサ開発環境全体図

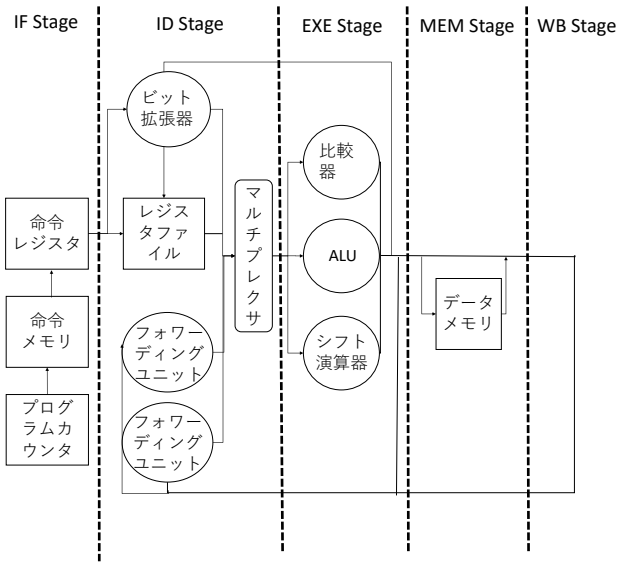

図 4: HW 概要図

で実装を行った．HW 実装の概要図を図 4 に示す．パイプラインの各ステージはそれぞれ IF，ID，EXE，MEM，WB ステージである．IF ステージでは，命令フェッチを行う．ID ステージでは，レジスタ，即値のデコードを行う．EXE ステージでは，演算処理を行う．MEM ステージでは，メモリの読み書きを行う．WB ステージでは，レジスタへの書き込みを行う．また，パイプラインハザードを防ぐために，フォワーディングユニットを 2 つ実装している．

3.3 命令拡張方法

実装した HW 開発環境への ASIP Meister を介した命令拡張の方法について説明する．

例として，ABSO (Absolute Value of Subtraction Result) 命令を追加した．ABSO 命令は，2 つのソースレジスタの値の差の絶対値をディスティネーションレジスタに格納する命令である．ASIP Meister での ABSO 命令の追加方法の概要図を図 5 に示す．ABSO 命令のビヘイビアは，rs1-rs2 が負であれば rs2-rs1 を，正であれば rs1-rs2 を rd に格納する．まず，使用する命令タイプを選び，オペコードの値をバイナリで記述する．オペコードの値は，他の命令と被りがないようにしなければならない．次に，パイプラインステージごとのマイクロ動作記述を生成する．IF ステージでは，命令レジスタ，プログラムカウンタを用いて，命令フェッチを行っている．ID ステージでは，ソースレジスタ rs1，rs2 の値をデコードする．このとき，直前の命令のレジスタ書き込みが反映されていない場合，フォワーディングユニットからの値を保存する．EXE ステージでは，演算処理を行う．まず，ALU を用いて 2 つのソースの値の減算を行い，temp2 に格納している．次に，reverse に temp2 の符号反転させた値を格納している．result に，temp2 の最上位ビットである符号ビットが負なら temp2

Vol.2019-ARC-235 No.12
Vol.2019-SLDM-187 No.12
Vol.2019-EMB-50 No.12
2019/3/17

```
if ( ( rs1 - rs2) < 0 )
     rd = rs2 - rs1
else
     rd = rs1 - rs2
```

ABSO命令のビヘイビア記述

1. 命令定義 : 命令フォーマット，オペランド，オペコードの指定

Format: **ABSO rd, rs1, rs2**

Inst. type	MSB	LSB	Field Type	Field Attr	Value
R	31	25	opcode	binary	**0000000**
I	24	20	operand	name	rs2
S	19	15	operand	name	rs1
U	14	12	opcode	binary	**000**
B	11	7	operand	name	rd
J	6	0	opcode	binary	**1010111**

2. マイクロ動作記述定義 : 各パイプラインステージごとの動作記述を定義

ステージ名	マイクロ動作記述
VARIABLE	wire [31:0] source0; wire [31:0] source1; wire [31:0] result;
IF	FETCH()

ID	wire[31:0] temp0;
	wire[31:0] temp1;
	temp0 = GPR.read0(rs1);
	temp1 = GPR.read1(rs2);
	source0 = FWU1.forward(rs1,temp0);
	source1 = FWU2.forward(rs2,temp1);

EXE	wire [3:0] flag;
	wire [31:0] temp2;
	wire[31:0] reverse;
	<temp2, flag> = ALU.sub(source0, source1);
	reverse = ~temp2;
	result = (temp2[31]) ? temp2 : reverse;
	null = FWU1.forward1(rd.result);
	null = FWU2.forward1(rd,result);

MEM	

WB	null = GPR.write0(rd, result);
	null = FWU1.forward3(rd.result);
	null = FWU2.forward3(rd,result);

図 5: ABSO 命令の命令追加手順

を，正なら reverse を格納している．最後に，WB ステージで，rd に演算結果を格納している．以上の手順で，ABSO 命令の命令追加を実現できる．

実装した HW に対しての性能評価について 4 節で述べる．

4. 評価実験

本節では，実装したプロセッサの性能の評価内容と方法，評価結果について述べる．

4.1 評価内容

実装したプロセッサに対して評価する内容は次の通りである．まず，ASIP Meister を用いることで減少した命令を追加するときに必要な変更記述量を計測する．設計品質に関しては HW の面積，遅延時間を評価する．最後に，生成されたプロセッサの網羅的な確認を行う．

4.1.1 命令追加時の変更記述量の評価

追加した命令は ABSO 命令と SWAP 命令である．ABSO 命令は，3 節で述べた 2 つの値の差の絶対を格納する命令である．

SWAP 命令の概要図を図 6 に示す．あるメモリのデータのエンディアンを反転させる命令である．まず，メモリから 32 ビットのデータを読み込む．8 ビットごとにデータを

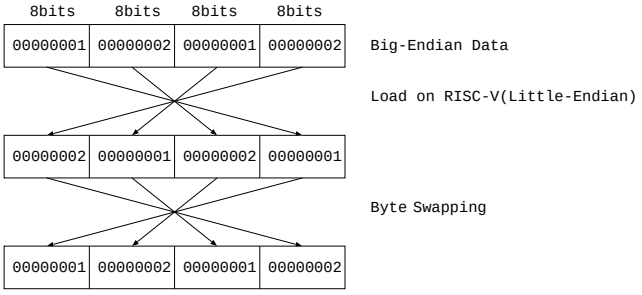

図 6: SWAP 命令の動作概要

分割し，順番を反転させてレジスタに格納する命令である．

以上の命令を追加した際に必要な変更記述量と ASIP Meister が生成した HDL 記述の変更記述量の比較を行った結果を表 2 に示す．変更記述量は，行数で ABSO 命令は 93.5 %，SWAP 命令では 94.5 %削減できている．以上より，命令追加による設計工数は大幅に削減できる．

4.2 設計品質の評価

実装したプロセッサの HDL に対して，Synopsys 社の Design Compiler と Nangate 社の Nangate 45nm Open Cell Library を使用し，ゲートレベルで論理合成することで，面積と遅延時間の計測を行った．また，クロック周波数の制約を 200MHz (5ns/clock)，400MHz (2.5ns/clock) とした．また，同じく RV32I と RV32M の実装である VexRiscv に対しても同様に計測を行った．計測結果を表 3 に示す．面積は 1.3 倍，遅延時間は 1.4 倍である．

4.3 プロセッサの正しい挙動の検証

最後に，実装した HW の正しい挙動の網羅的な確認を行った．評価環境を図 7 に示す．

テストプログラムはアセンブリプログラムを用いる．プログラムは，実装した全命令を繰り返し実行するものと，ランダムに命令を実行するものである．また，参照レジスタもランダムに扱う．プログラム数は 1 命令につき 20 種の連続で同じ命令を実行するものとランダムに実行するプログラムが 50 種である．

HW での実行方法としては，PAS というツールを用い

表 2: 命令の追加前と追加後の記述量の比較

	ASIP Meister の変更行数	HDL 記述の変更行数
ABSO 命令	22	341
SWAP 命令	35	615

表 3: 設計品質の評価結果

プロセッサ	動作制約	面積 [μm^2]	ゲート数	遅延時間 [ns]
ASIP Meister	200MHz	52,632	28,037	4.92
VexRiscv	200MHz	37,245	19,841	3.70
ASIP Meister	400MHz	53,272	28,378	2.42
VexRiscv	400MHz	37,332	19,887	1.50

Vol.2019-ARC-235 No.12
Vol.2019-SLDM-187 No.12
Vol.2019-EMB-50 No.12
2019/3/17

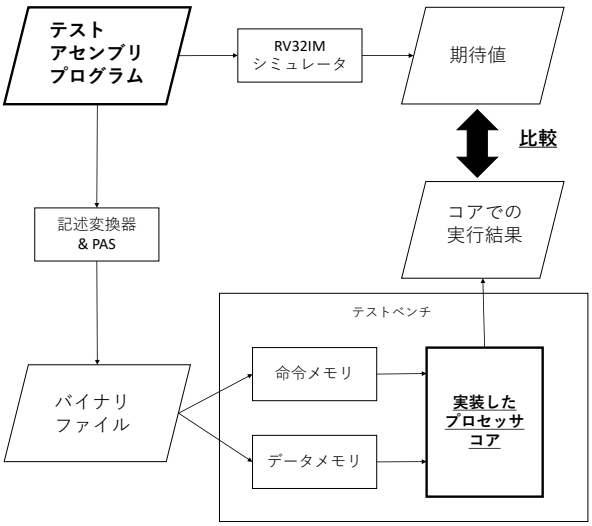

図 7: 評価環境

て，バイナリファイルへの変換を行う．そして，バイナリファイルの命令を命令メモリ，データをデータメモリに与えてシミュレーションを行う．

期待値は，アセンブリプログラムをを riscv-tools のシミュレータ Spike を用いて実行を行う．Spike の実行結果と HW での実行結果の汎用レジスタ 32 個の値を全て比較することで正当性の網羅的な確認を行う．

評価結果として，全てのテストプログラムにおいて正しい結果が得られた．

5. 結論

本稿では ASIP Meister を用いて RISC-V の HW 開発環境を実装し，RISC-V の開発環境全体を検討した．実装した HW 開発環境に対して多数のアセンブリプログラムを実行し，全てのプログラムに対して正しい結果が得られた．これにより，プロセッサとして正しい挙動を行うことを網羅的に確認した．また，実際に命令追加を行い，HDL に命令拡張を行った場合との必要変更記述量の比較を行い，約 6%の記述量で命令追加が可能であることを確認した．

今後の課題として，まず，性能向上が挙げられる．同じ命令セットを持つ VexRiscV と比較した結果，面積が約 1.3 倍，遅延時間が約 1.4 倍となった．同等の面積と遅延時間を持つプロセッサが生成できれば有用性が高くなる．

また，現在の実装は RISC-V の 32 ビットの基本命令と乗除算命令のみである．ベクトルアーキテクチャである RV32V や Compressed 命令セットである RV32C も実装目標として挙げられる．また，RISC-V の SW 開発環境である riscv-tools とビヘイビア記述からのマイクロ動作記述生成手法 [9] を用いて，より簡単で矛盾のない単一の命令定義から HW・SW 開発環境への命令拡張が可能な開発環境の構築が挙げられる．

謝辞

本研究の成果の一部は，JSPS 科研費 JP17K00077 の助成によるものである．本研究は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] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[2] “GitHub - riscv/riscv-tools: RISC-V Tools (GNU Toolchain, ISA Simulator, Tests).” [Online]. Available: https://github.com/riscv/riscv-tools

[3] K. Asanovic, J. B. Rimas Avizienis, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, and J. Koenig, “The rocket chip generator,” Electrical Engineering and Computer Sciences Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016.

[4] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovi, “BOOMv2: an open-source out-of-order RISC-V core,” Electrical Engineering and Computer Sciences Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, 2017.

[5] M. Imai, Y. Takeuchi, K. Sakanushi, and N. Ishiura, “Advantage and possibility of application-domain specific instruction-set processor (ASIP),” *IPSJ Transactions on System LSI Design Methodology*, vol. 3, pp. 161–178, 2010.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, john Wawrzynek, and K. Asanović, “Chisel: constructing hardware in a scala embedded language,” in *Proceedings of Design Automation Conference*, 2012, pp. 1212–1221.

[7] “GitHub - SpinalHDL/VexRiscv: a FPGA friendly 32 bit RISC-V CPU Implementation.” [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[8] P. Mishra and N. Dutt, *Processor Description Languages Applications and Methodologies*. Morgan Kaufmann Publishers, 2008.

[9] T. Shiro, M. Abe, K. Sakanushi, Y. Takeuchi, and M. Imai, “A processor generation method from instruction behavior description based on specification of pipeline stages and functional units,” in *Proceedings of Asia and South Pacific Design Automation Conference*, 2007, pp. 286–291.