C-019

RISC-V プロセッサの FSL を用いた実装と評価
Evaluation of RISC-V Processors Designed and Implemented using FSL

大西 創也 † 渡邊 誠也 † 名古屋 彰 †
Soya Ohnishi Nobuya Watanabe Akira Nagoya

1. はじめに

近年，オープンな命令セットアーキテクチャであるとして RISC-V [1] が注目されており，これを命令セットアーキテクチャとしたプロセッサやマイクロコントローラの実装が行われている．

一方，我々の研究室ではハードウェア記述言語 FSL(Functional and Scalable Hardware Description Language) [2,3] を開発している．

本稿では，FSL を用いてパイプライン制御の RISC-V プロセッサを実装し，プロセッサの性能評価を行った結果と，実装を通して得られた FSL の記述上の利点について報告を行う [4].

本稿の構成は以下のとおりである．まず，第 2 章で研究背景としてハードウェア記述言語 FSL や RISC-V について述べた後，本研究の目的を述べる．第 3 章では，FSL を用いて実装した各種パイプライン構成の RISC-V プロセッサについて述べる．第 4 章ではプロセッサの動作検証方法や FPGA をターゲットデバイスとした論理合成結果を示す．また，ベンチマークプログラムを用いた各種プロセッサの評価と考察について述べる．第 5 章で FSL 記述の評価と考察を行い，第 6 章で本研究についてまとめを述べる．

2. 研究背景

本章では，ハードウェア記述言語 FSL や RISC-V について述べ，本研究の目的を示す．

2.1 ハードウェア記述言語 FSL

FSL は現在，我々が開発している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標としている．FSL はハードウェア記述言語 SFL (Structured Function Description Language) [5] の設計思想を受け継いでおり，手続き的記述でハードウェアの動作を記述可能である．2.1.1 項では SFL の概要を示す．また，FSL はプログラミング言語 Scala [6] の構文を採用しており，記述の柔軟性の向上を試みている．2.1.2 項と 2.1.3 項では FSL による記述例を示す．

2.1.1 ハードウェア記述言語 SFL

SFL は NTT で開発されたハードウェア記述言語である．記述対象のハードウェアを単相クロックに限定し，ハードウェアの動作の記述を手続き的記述のみにすることで，Verilog HDL や VHDL などの RTL (Register Transfer Level) 記述に比べ，抽象度が高く，よりソフトウェアに近い記述が可能となっている．

また，FSL が提供している構文の中には SFL が元々備えていたものを踏襲した構文も存在し，2.1.3 項で述べるステージの構文などが挙げられる．

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

```
 1  private def lessThan(a: EnvBit,
                         b: EnvBit): Boolean = {
 2    val res = alu.sub(a, b).out
 3
 4    val hash = a.msb ++ b.msb
 5    hash match {
 6      case 0b10 => true
 7      case 0b01 => false
 8      case    _ => res.msb == 0b1
 9    }
10  }
```

図 1: 関数の定義

```
 1  stage idecode(pc: EnvBit, ...) {
          ⋮
39    state decoding {
40      if(stallCtrl.idecodeStall.isRunning)
41        goto stall
42      else
43        relay execution(pc, ...)
44    }
          ⋮
58  }
```

図 2: ステージの定義

2.1.2 モジュール内での関数の記述

FSL では関数を用いてモジュール内の処理を記述する．そのため，手続き的記述でハードウェアの動作を記述可能である．図 1 は本研究で実装したプロセッサのソースコードから，比較演算を行う関数の定義を抜粋したものである．

1 行目で示すように関数 lessThan には private 修飾子が付いている．private 修飾子を付けたとき，この関数にアクセスできるのは関数を定義しているモジュール内からのみとなる．そのため，モジュール内部の処理を外部から隠蔽することが可能である．

SFL では関数と似た構文を用いて処理の定義を行う．しかし，FSL とは異なり，引数部分の端子をモジュールの定義の一部で宣言する必要がある．このため，端子が増えた場合に各端子の管理が難しくなるという問題が存在した．FSL では図 1 が示すように，引数を用いることができるためこの問題を解消している．

2.1.3 モジュール内でのステージおよびステートの記述

FSL は SFL と同様にパイプライン処理の記述を容易にするためのステージの構文および複数サイクルに渡る処理の記述を容易にするためのステートの構文を提供している．図 2 は本研究で実装したプロセッサのソースコードのうち，ステージおよびステートの構文を用いた記述を抜粋したものである．

ステートはステージの中に記述する．また，ステージは複数のステートを有することができ，41 行目では goto 文を用いて次のサイクルにおけるステートを指定している．また，43 行

Copyright © 2019 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

表 1: RISC-V の主な命令セット

略号	概要	種別
RV32I	32 ビット基本整数命令	基本命令セット
M	整数乗除算命令	拡張命令セット
A	アトミック命令	拡張命令セット
C	圧縮命令	拡張命令セット
F	単精度浮動小数点命令	拡張命令セット
D	倍精度浮動小数点命令	拡張命令セット

目では relay 文を用いて，次のサイクルで自分のステージを終了させせつつ，relay 文で指定したステージが起動されるようにしている．

SFL ではモジュール内の各ステージが使用するレジスタをモジュール定義の一部でまとめて宣言する必要があった．そのため，宣言するレジスタが増えると，どのレジスタをどのステージが利用するかなどが理解しにくくなるという問題点が存在した．FSL はこの点を改善し，図 2 に示すように，各ステージで扱うレジスタを対応するステージ定義の引数として宣言することが可能となっている．

2.2 RISC-V

RISC-V はカリフォルニア大学のバークレー校で開発され，現在は RISC-V Foundation により開発，管理が行われているオープンな命令セットアーキテクチャである．

RISC-V は命令セットを複数有し，それらは実装することが必須である基本命令セットと実装の取捨選択が可能な拡張命令セットに分類される．拡張命令セットの実装は取捨選択可能な点から，設計者は必要な命令セットのみを採用してプロセッサを実装することが可能である．表 1 は文献 [7] で定義されている主な命令セットを示したものである．

また，カスタム命令用のオペコードが定義されている．設計者はカスタム命令の処理内容を自由に決定することができるため，特定の処理の高速化を行うことが可能である．

既に RISC-V プロセッサの実装も行われており，ハードウェア記述言語である Verilog HDL や Chisel，NSL を用いたものが存在している [8–11]．Verilog HDL と Chisel を用いた実装はソースコードが GitHub 上に公開されている．

2.3 本研究の目的

2.2 節で，RISC-V プロセッサの実装例がいくつか存在することを示した．一方で，FSL を用いた RISC-V プロセッサの実装は行われていない．FSL を用いて RISC-V プロセッサを実装することで，今後の FSL を用いた研究において，拡張命令やカスタム命令の実装といったプロセッサの拡張の土台や SoC (System on a Chip) を用いた研究におけるプロセッサ部分として利用することができる．一方で，FSL の開発において，プロセッサの実装は今後の機能拡張などでより生産性の高い言語に発展させるための材料となる．

そこで本研究では，FSL を用いて RISC-V プロセッサを実装し，FPGA をターゲットデバイスとして論理合成を行い得られた結果やベンチマークプログラムの実行結果から性能評価を定量的に行いつつ，パイプラインステージ構成などの改良を進めることとした．結果的に 7 種の構成の RISC-V プロセッサを設計し，評価を実施したため，これらをまとめて第 3 章，第 4 章で説明する．また，2.1.3 項で示したステージの構文など，FSL が提供している機能について，プロセッサの実装における各機能を利用した記述の考察を第 5 章で行う．

3. 本研究で実装した RISC-V プロセッサ

本研究ではインオーダ実行のパイプライン制御 RISC-V プロセッサを FSL を用いて実装した．本章では命令セットについて述べた後，実装した 7 種のプロセッサについて示す．

3.1 命令セット

本研究では基本命令セットである RV32I を実装した．RV32I は 32 ビット幅のアドレス長を持ち，32 ビット幅の汎用レジスタを 32 本設けることを示す．

RISC-V は複数の実行モードを定義しているが，本研究ではマシンモードのみを実装し，割り込みおよび例外については，RISC-V が定義しているものの中から，マシンタイマ割り込み，不正命令例外，ブレークポイント例外，システムコール例外（マシンモード）を実装した．

3.2 実装したプロセッサ

本研究で実装した 7 種の RISC-V プロセッサはそのパイプライン構成を反映して **(1) 5 ステージ**，**(2) キャッシュレイテンシ削減**，**(3) メモリステージ分割**，**(4) 改良版 5 ステージ**，**(5) 9 ステージ**，**(6) 6 ステージ**，**(7) キャッシュレイテンシ隠蔽**とそれぞれ名付けた．この中で基本型となる **(1) 5 ステージ**の各パイプラインステージは，命令をフェッチする IF ステージ，命令のデコードとレジスタの読み込みを行う ID ステージ，演算を行う EX ステージ，キャッシュへアクセスする MEM ステージ，命令の結果をレジスタに書き込む WB ステージで構成されている．

表 2 は各プロセッサのパイプライン構成の考え方をまとめたものである．また，表 3 は各プロセッサのパイプライン段数や増分サイクル数などを示したものである．表 3 のうち，分岐予測ミス時増分サイクル数は分岐予測ミスが発生した際に，ミスが発生しなかった場合と比べて余分に消費するサイクル数を示し，データハザード時最大ストールサイクル数は，データハザードによりフォワーディングを行うステージでストールするサイクル数の最大値を示す．

各プロセッサのパイプライン制御やキャッシュ処理など，複数サイクルに渡って行われる処理の記述には，2.1.3 項で述べたステージおよびステートの構文を用いている．これらの構文を用いることで，各パイプラインステージの処理の記述をそれぞれ独立させることができ，変更や修正が容易になる．

4. 実装と評価

本章では，第 3 章 で示した 7 種の RISC-V プロセッサの FSL を用いた実装について述べた後，各プロセッサの動作検証について示す．また，FPGA をターゲットデバイスとした論理合成やベンチマークプログラムの実行結果を用いた性能評価について述べる．

4.1 FSL を用いたプロセッサの実装

本研究では，第 3 章で示した 7 種の RISC-V プロセッサを FSL を用いて実装した．

表 4 は本研究で実装したプロセッサのうち，**(1) 5 ステージ**の実装で作成したソースコードのファイル名と行数を示したものである．

表 4 で示したファイル名のうち，赤字になっているものは

Copyright © 2019 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

表 2: 各プロセッサのパイプライン構成の考え方

プロセッサ名		パイプライン構成の概要
(1)	5 ステージ	基本型となる 5 ステージパイプライン構成． データキャッシュのレイテンシと MEM ステージの構成から ロード命令は 1 サイクル以上のストールを必ず発生させる問題点が存在．
(2)	キャッシュレイテンシ削減	(1) におけるキャッシュレイテンシを 1 に削減することで， キャッシュヒット時にロード命令がストールを発生させない構成．
(3)	メモリステージ分割	(1) における MEM ステージを 2 つに分割し， データキャッシュへのアクセスとデータの受け取りを行うステージを分けることで， キャッシュヒット時にロード命令がストールを発生させない構成．
(4)	改良版 5 ステージ	(1) におけるデータキャッシュからデータを受け取るステージを MEM ステージから WB ステージに変更することで， キャッシュヒット時にロード命令がストールを発生させない構成．
(5)	9 ステージ	(1) を基にして，最大動作周波数向上のために パイプラインステージを細分化した構成．
(6)	6 ステージ	(5) を基にして，最大動作周波数を維持しつつ， パイプラインステージを統合した構成．
(7)	キャッシュレイテンシ隠蔽	(6) から，キャッシュにアドレスを渡すタイミングを変更し， キャッシュレイテンシを隠蔽した構成．

表 3: 各プロセッサの比較

プロセッサ名		ステージ数	分岐予測ミス時 増分サイクル数	データハザード時 最大ストールサイクル数	キャッシュレイテンシ (サイクル数)	
					命令	データ
(1)	5 ステージ	5	3	1	2	2
(2)	キャッシュレイテンシ削減	5	3	1	1	1
(3)	メモリステージ分割	6	3	2	2	2
(4)	改良版 5 ステージ	5	3	2	2	2
(5)	9 ステージ	9	5	4	2	3
(6)	6 ステージ	6	4	3	2	3
(7)	キャッシュレイテンシ隠蔽	5	3	3	2	3

他のプロセッサの実装で変更を行ったファイルである．変更が必要になった主な要因として，パイプラインステージ構成の変更により管理しなければならないフラグの数が変化したことやキャッシュからデータを受け取ることができるようになるまでのタイミングを変更したことなどが挙げられる．

以降の節では FSL を用いて実装したプロセッサの動作検証や性能評価について述べる．

4.2 動作検証

ここでは実装したプロセッサの動作検証について述べる．まず，4.2.1 項で動作検証に用いる動作検証プログラムについて述べる．4.2.2 項ではこの動作検証プログラムを実装したプロセッサ上でどのようにして実行させるのかを説明し，4.2.3 項で各プロセッサの動作確認を行い，正しく実装できたことを示す．

4.2.1 検証方法

実装したプロセッサの動作検証には RISC-V Foundation が提供している riscv-tests [12] を用いる．riscv-tests には RISC-V が定義している各命令の動作検証プログラムが含まれており，RV32I で定義されている命令の動作検証プログラムも存在する．例えば，RV32I では ADD 命令が定義されているが，この命令の動作検証プログラムとして rv32ui-p-add が存在する．

各動作検証プログラムは命令のオペランドの値を変えるなどして，いくつかのパターンを試すことで動作検証を行い，最後にアドレス 0x80001000 へ値を書き込むことでテストが終了したとする．アドレス 0x80001000 に書き込まれた値が 1 の場合，すべてのパターンで正しく動作したことを示す．一方で，それ以外の値の場合はいずれかのパターンで命令が正しく動作しなかったことを示す．正しく動作しなかった場合，アドレス 0x80001000 に書き込まれた値を右に 1 ビットシフトすることで，何番目のパターンで正しく動作しなかったのかを判断できる．

4.2.2 動作検証プログラムの実行方法

実装したプロセッサ上での動作検証プログラムの実行には，プロセッサの動作をシミュレートするシミュレーションプログラムを用いる．実装したプロセッサでプログラムを実行させるまでの流れを図 3 に示す．

シミュレーションプログラムを生成するため，まず，FSL 記述ファイルを FSL Compiler [3] を用いて Verilog HDL 記述ファイルに変換する．この Verilog HDL 記述ファイルを Icarus Verilog に読み込ませ，シミュレーションプログラムを生成する．

また，動作検証プログラムをシミュレーションプログラム上で実行させるためには，動作検証プログラムをシミュレーションプログラムが読み込み可能な形式に則したテキストファイルへ変換する必要がある．まず，プログラムのソースコードを RISC-V 用の GCC (GNU Compiler Collection) [13] を用いてコンパイルし，ELF (Executable and Linkable Format) ファイルを得る．この ELF ファイルを，実行する命令の機械語とプログラムが使用するデータのみが記述されたテキストファイルへ変換する．シミュレーションプログラム実行時にこ

Copyright © 2019 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

表 4: (1) 5 ステージの実装における FSL 記述量

ファイル名	行数 (行)	ファイル名	行数 (行)	ファイル名	行数 (行)
ALU.fsl	28	BranchPredictionUnit.fsl	89	Opcodes.fsl	13
Add.fsl	11	Core.fsl	701	PrivilegeMacro.fsl	5
BarrelShifter.fsl	31	Decoder.fsl	466	StageBitMacro.fsl	21
Compare.fsl	58	ExceptionControl.fsl	108	BramControl.fsl	191
CsrController.fsl	35	FlushControl.fsl	68	BlockRam.fsl	28
Execution.fsl	97	ForwardingUnit.fsl	133	BlockRamWrapper.fsl	32
Cache.fsl	82	PCControl.fsl	97	DualPortRam.fsl	34
CacheBusyControl.fsl	18	ReturnAddressStack.fsl	106	DualPortRamWrapper.fsl	50
DataAllocater.fsl	98	StallControl.fsl	44	SinglePortRam.fsl	24
DataCacheControl.fsl	483	Top.fsl	148	SinglePortRamWrapper.fsl	32
InstAllocater.fsl	138	CacheMacro.fsl	4	CSRControl.fsl	434
InstCacheControl.fsl	276	DecodeFlagMacro.fsl	32	RegisterFile.fsl	123
PseudoLRUControl.fsl	35	DecodeType.fsl	7	PerformanceCounter.fsl	544
PseudoLRUTable.fsl	28	ExecMacro.fsl	53	Debugger.fsl	13
WriteBacker.fsl	78	MemMacro.fsl	12		
行数合計: 5108					

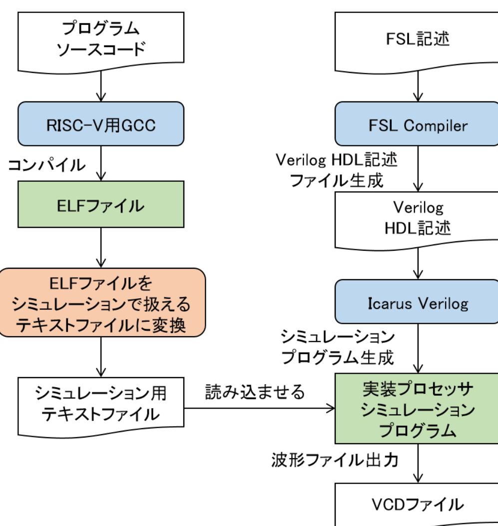

図 3: 実装したプロセッサでのプログラム実行手順

のテキストファイルを読み込ませ，実装したプロセッサ上でのプログラムの実行をシミュレートする．

4.2.3 実装したプロセッサの動作確認

本研究で実装した RISC-V プロセッサの動作検証には ECALL 命令，EBREAK 命令および CSR 操作命令を除いて rv32ui-p-“命令名” のプログラムを用いた．また，ECALL 命令は rv32mi-p-scall，EBREAK 命令は rv32mi-p-sbreak，CSR 操作命令は rv32mi-p-csr を用いて動作検証を行った．

動作検証プログラムを実行した結果，すべてアドレス 0x80001000 に 1 を書き込んだため，RISC-V プロセッサを正しく実装できたとした．

4.3 論理合成による評価

ここでは，論理合成結果を用いて 3.2 節で示した各プロセッサの評価を行う．まず 4.3.1 項で論理合成の実行環境について述べ，4.3.2 項で論理合成結果を用いた各プロセッサの評価を行う．

4.3.1 論理合成環境

本研究では，Intel 社の FPGA である Cyclone IV EP4CE115F29C7N [14] をターゲットデバイスとして論理合成を行う．このデバイスは Cyclone IV デバイス・アーキテクチャにおけるロジックの最小単位である LE (Logic Element) を 114,480 個持つ．LE は LUT (Look Up Table) とレジスタを内包しているため，論理合成結果の LE 数には使用レジスタ数が含まれる．また，メモリブロックを 432 個内蔵しており，最大 3,981,312 ビットの利用が可能である．デバイスのその他の構成要素として，DSP (Digital Signal Processor) が存在するが，本研究の実装では使用しない．

論理合成には Intel 社の FPGA 開発ソフトウェアである Quartus Prime を用いる．Quartus Prime は FSL 記述ファイルを扱えないため，FSL Compiler を用いて FSL 記述ファイルを Verilog HDL 記述ファイルへ変換し，これを Quartus Prime への入力とした．

4.3.2 論理合成結果

各プロセッサの論理合成結果を表 5 に示す．また，表 5 のうち，各プロセッサの LE 数と最大動作周波数をグラフ化したものを図 4 と図 5 にそれぞれ示す．

ステージ数が増加するほど LE 数とレジスタ数が増加していることが分かる．これはステージ数が増加することで，パイプラインレジスタが増加するためである．一方で，**(6) 6 ステージ**の LE 数とレジスタ数が同じステージ数である **(3) メモリステージ分割**より大きい値になっている．これは，**(6) 6 ステージ**ではキャッシュ内部の処理におけるパイプラインステージ数が 2 から 3 に増加したことで，パイプラインレジスタの数が増えたためである．

また，最大動作周波数の向上を図ったプロセッサである **(5) 9 ステージ**と **(6) 6 ステージ**の最大動作周波数はそれぞれ **(1) 5 ステージ**の 1.23 倍，1.22 倍となっている．そのため，パイプラインステージを細分化した効果が現れていることが分かる．一方で **(6) 6 ステージ**を基にした **(7) キャッシュレイテンシ隠蔽**の最大動作周波数は **(1) 5 ステージ**の 1.08 倍に留まっている．この原因として，**(7) キャッシュレイテンシ隠蔽**は次の命令のアドレスを一旦 PC に格納せず，直接命令キャッシュへ渡すため，クリティカルパスの配線遅延が **(5) 9 ステージ**や **(6) 6 ステージ**に比べ増加したことが挙げられる．

(2) キャッシュレイテンシ削減は LE 数とレジスタ数が他より多い．これは，表 2 に示した構成の変更により，他のプロセッサはデータキャッシュの構成要素がメモリブロックで実

Copyright © 2019 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

表 5: 各プロセッサの論理合成結果

プロセッサ名		LE 数 (使用率)		レジスタ数	メモリブロック数	最大動作周波数 (MHz)
(1)	5 ステージ	8,708	(8%)	4,867	28	66.93
(2)	キャッシュレイテンシ削減	108,452	(95%)	80,312	18	38.62
(3)	メモリステージ分割	8,709	(8%)	4,958	28	63.98
(4)	改良版 5 ステージ	8,656	(8%)	4,872	28	64.35
(5)	9 ステージ	10,089	(9%)	5,828	28	84.42
(6)	6 ステージ	9,228	(8%)	5,330	28	84.07
(7)	キャッシュレイテンシ隠蔽	8,803	(8%)	5,149	28	72.47

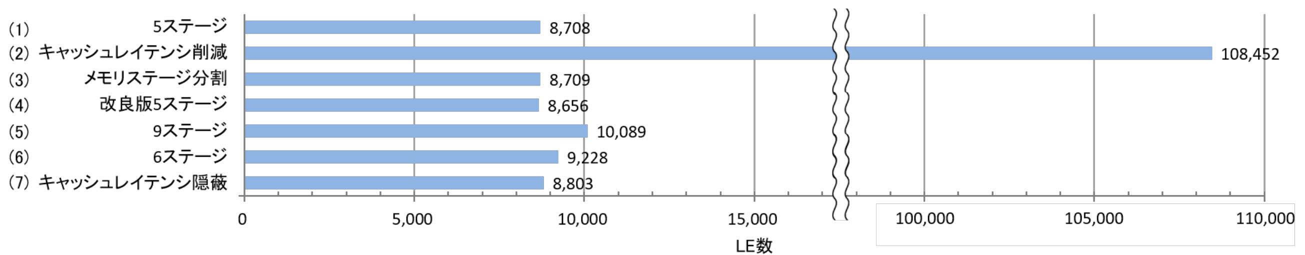

図 4: 各プロセッサ LE 使用量

装された一方で，**(2) キャッシュレイテンシ削減**ではレジスタで実装されたためである．また，**(2) キャッシュレイテンシ削減**の最大動作周波数は **(1) 5 ステージ**の 0.58 倍に低下している．この原因として，レジスタ数の増加により配線の自由度が減り，配線遅延が大きくなったことが挙げられる．その他に，他のプロセッサがデータキャッシュ内部の処理を複数サイクルかけて実行している一方で，**(2) キャッシュレイテンシ削減**では 1 サイクルで処理を完了させる必要があるため，論理段数が深くなったことが考えられる．

4.4 ベンチマークプログラムの実行結果を用いた評価と考察

ここではベンチマークプログラムの実行結果を用いた各プロセッサの評価と考察を行う．まず，4.4.1 項でベンチマークに使用するプログラムについて述べる．また，4.4.2 項と 4.4.3 項ではそれぞれベンチマークプログラムの実行により得られた CPI と平均命令実行時間を用いて，各プロセッサの評価を行う．最後に 4.4.4 項でベンチマークプログラムの実行結果を用いた考察を行う．

4.4.1 ベンチマークに用いるプログラム

本研究では riscv-tests に含まれているベンチマークプログラムを用いてベンチマークを行う．ベンチマークプログラムを実行する際は 4.2.2 項 で示した方法を用いる．本研究で用いるベンチマークプログラムを表 6 に示す．各プログラムが使用するデータのうち，qsort と rsort は同じデータを用いる．また，towers を除いて各プログラムのデータはシード値を固定して生成した乱数を用いる．

表 6: 各ベンチマークプログラムの概要

プログラム名	アルゴリズム	データ数
qsort	クイックソート	2,048
rsort	基数ソート	2,048
median	中央値算出	400
multiply	シフトと加算を用いた乗算	100
vvadd	ベクトル加算	300
towers	ハノイの塔	（ディスク数） 7

riscv-tests には表 6 で示されていないベンチマークプログラムがいくつか存在している．しかし，それらは本研究で実装した命令セットである RV32I のみではコンパイルが出来ないプログラムや，本研究で実装できていない実時間を測る機能が必要なプログラムであるため除外した．

4.4.2 CPI による比較

各プロセッサの性能について表 6 で示したベンチマークプログラムの実行結果から得た CPI を用いて比較を行う．CPI を測定する際には測定用のカウンタが必要となるため，ベンチマークプログラム実行時のみ，CPI を測定するためのカウンタを追加で設ける．CPI は各ベンチマークプログラム終了時に得られたカウンタの値を用いて算出する．また，性能低下を引き起こしている主要な原因を特定するため，複数ある性能低下の原因毎にカウンタを用意する．

図 6 は各プロセッサにおいて，6 つのベンチマークプログ

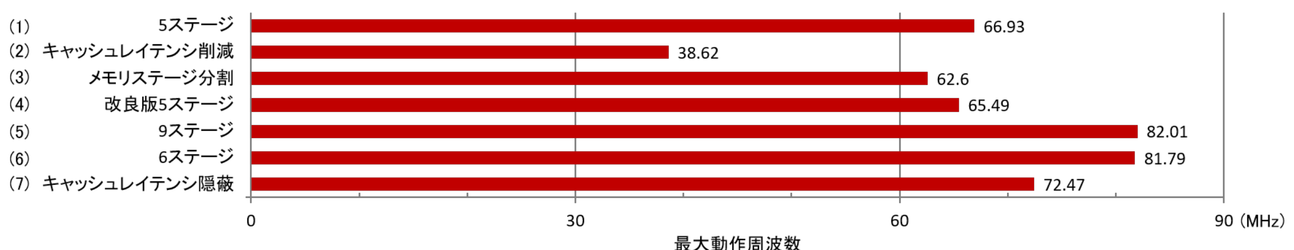

図 5: 各プロセッサ最大動作周波数

Copyright © 2019 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

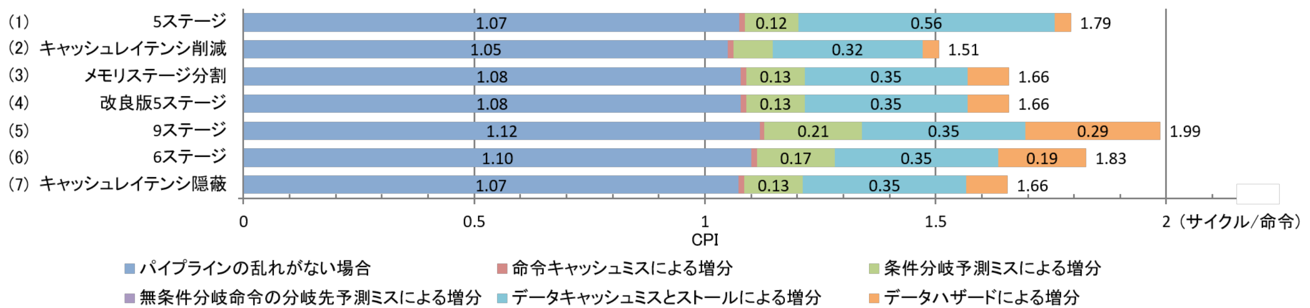

図 6: ベンチマーク実行結果による各プロセッサの CPI

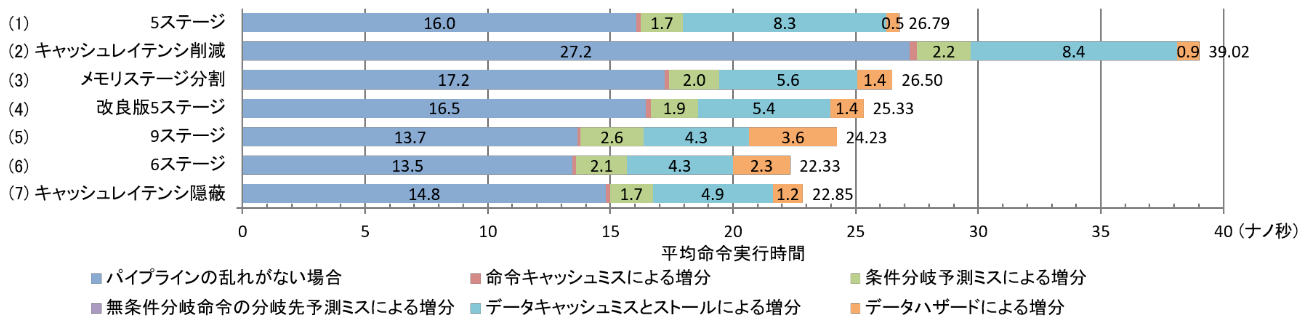

図 7: ベンチマーク実行結果による各プロセッサの平均命令実行時間

ラムの実行結果から得られた CPI の平均値をそれぞれ示したものである．また，図 6 の積み上げグラフ上の値は CPI が増加する各原因毎の増分値を示す．

(1) 5 ステージは「データキャッシュミスとストールによる増分」の値が，他のプロセッサの平均値の 1.63 倍である．これは，表 2 で示したように，**(1) 5 ステージ**はロード命令を実行した際，MEM ステージで必ず 1 サイクル以上のストールを発生させるためである．

また，**(2) キャッシュレイテンシ削減**の CPI が最も小さく，**(1) 5 ステージ**の 0.84 倍である．これは，分岐予測ミスにより正しい命令をフェッチし直す際に，他のプロセッサよりも，キャッシュから少ないレイテンシで命令を受け取ることが可能である点が理由として挙げられる．また，**(1) 5 ステージ**を除く他のプロセッサは，ロード命令によるキャッシュヒット時のストールを防止するため，フォワーディングユニットが存在するステージとロード命令がキャッシュからデータを受け取るステージの間のステージ数が **(2) キャッシュレイテンシ削減**のプロセッサよりも多い．そのため，**(2) キャッシュレイテンシ削減**はデータハザードによる性能の低下が他より小さいことがその他の理由として挙げられる．

一方で，**(5) 9 ステージ**と **(6) 6 ステージ**はそれぞれ CPI が **(1) 5 ステージ**の 1.11 倍，1.02 倍に増加している．これは分岐予測の成否を判定するステージやロード命令がキャッシュからデータを受け取るステージが他のプロセッサよりも深い位置に存在していることで，分岐予測ミスやデータハザードによる性能低下が大きくなったためである．

4.4.3 平均命令実行時間による比較

各プロセッサの性能について，ベンチマークの実行結果から得た平均命令実行時間を用いて比較を行う．平均命令実行時間は，4.4.2 項で示した CPI の値と 4.3.2 項で示した最大動作周波数の値を用いて算出できる．

図 7 は各プロセッサにおいて，6 つのベンチマークプログラムの実行結果から得られた平均命令実行時間の平均値をそれぞれ示したものである．また，図 7 の積み上げグラフ上の値は平均命令実行時間が増加する各原因毎の増分値を示す．

図 7 から CPI が最も小さかった **(2) キャッシュレイテンシ削減**は，平均命令実行時間が **(1) 5 ステージ**の 1.46 倍に増加していることが分かる．これは 4.4.2 項で述べたように，最大動作周波数が **(1) 5 ステージ**の 0.58 倍に低下したためである．

また，最大動作周波数を向上させることで性能の改善を図ったプロセッサのうち，**(5) 9 ステージ**と **(6) 6 ステージ**は 4.4.2 項で示したように CPI は他のプロセッサに比べ大きい値であったが，平均命令実行時間はそれぞれ **(1) 5 ステージ**の 0.90 倍，0.83 倍になっている．そのため，ステージ数を増やし，CPI が増大したことで生じた性能低下を，最大動作周波数上昇による性能向上が上回ったことが分かる．

一方で，**(6) 6 ステージ**が基となっている **(7) キャッシュレイテンシ隠蔽**は CPI が **(6) 6 ステージ**の 0.88 倍であったにも関わらず，平均命令実行時間では **(6) 6 ステージ**の 1.02 倍と劣っている．このことから，CPI が改善されたとしてもそれに応じて最大動作周波数が低下してしまった場合，性能改善は期待できないことが分かる．

4.4.4 ベンチマーク結果の考察

ベンチマークの結果から，パイプラインステージの細分化により最大動作周波数が上昇し性能が向上したことで，分岐予測ミスなどの影響増加による性能低下を打ち消すことが分かった．そのため，処理速度を上げる場合は，仮に CPI が増加したとしてもパイプラインステージの細分化を行い，最大動作周波数を向上させた方が良いと考えられる．

一方，同じステージ数である **(3) メモリステージ分割**と **(6) 6 ステージ**について，**(6) 6 ステージ**の平均命令実行時間は **(3) メモリステージ分割**の 0.84 倍となっている．これは表 5 に示すように，**(6) 6 ステージ**の最大動作周波数が **(3) メモリステージ分割**の 1.31 倍である点が大きい．ここから，パイプラインステージを細分化したとしても，必ずしも最大動

Copyright © 2019 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

```
1   module DualPortRam(dataWidth = 8,
                       addrWidth = 8,
                       depth     = 256) {
2     input readAddr: Bit(addrWidth)
3     input re:       Boolean
4
5     input writeAddr: Bit(addrWidth)
6     input writeData: Bit(dataWidth)
7     input we:        Boolean
8
9     output out: Bit(dataWidth)
10
11    mem[Bit(dataWidth)] ram(depth)
                :
33  }
34
35  module BranchPredictionUnit {
                :
45    val branchRAM = new DualPortRam(32, 8, 256)
46    val infoRAM   = new DualPortRam(27, 8, 256)
                :
121 }
```

図 8: パラメータによるモジュールの再利用例

作周波数が増加するわけではなく，最大動作周波数上昇による性能向上を狙うには，パイプラインレジスタを挟む位置を慎重に検討しなければならないことが分かる．

5. FSL を用いた記述の評価と考察

ここでは，プロセッサの実装における FSL の記述について，FSL が提供している機能の評価を行なった後，FSL 記述に関する考察を述べる．

5.1 ステージの構文

FSL はステージの構文を用いてパイプライン処理の記述が可能であることを 2.1.3 項で述べた．ステージの構文を用いた場合，各サイクルで行う処理がブロック内の記述でまとまるため，可読性が高く，処理の改変も容易になる．

また，2.1.3 項で示したようにステージの構文は引数を宣言でき，これらはレジスタとして扱われる．このため，各ステージが使用するレジスタが明確になり，可読性が向上する．

その他のステージを用いる利点として，ステージの引数として扱われるレジスタのスコープは各ステージの内部で閉じているため，そのステージの外からレジスタにアクセスするには “ステージ名.引数名” と記述する．このため，レジスタ名の競合が発生しにくくなることやレジスタ名の記述ミスによるバグの発生を抑えることが期待できる．

以上の理由から，本研究において複数のプロセッサを記述する際，パイプラインステージの構成の改変を容易に行うことができた．

5.2 パラメータ

FSL では，パラメータを用いてモジュールの定義を一般化することが可能である．パラメータを用いることで，1 つのモジュールの定義から，異なるビット幅の端子や異なる深さのメモリを扱うモジュールを利用することが可能となる．そのため，同じソースコードを再利用できる範囲が広がり，記述の複製防止や修正の手間の減少に繋がる．

```
1   private inline def getTag(addr: EnvBit) =
      addr(31, 12)
2
3   private def isHit(addr: EnvBit):
                  (HitStatus, WayBit) = {
4     val tag = getTag(addr)
              :
22  }
23
24  private def hitHandler(addr: EnvBit) {
25    val tag = getTag(addr)
              :
42  }
```

図 9: インライン関数の使用例

図 8 は実際にパラメータを利用した例である．1 行目で記述されている dataWidth，addrWidth，depth の 3 つの変数が DualPortRam モジュールが有するパラメータである．これらのパラメータは端子のビット幅やメモリの深さの指定に用いられている．45 行目と 46 行目では DualPortRam モジュールがパラメータの指定を行った上でインスタンス化されている．パラメータがなければ，インスタンス化される 2 つのモジュールのために，類似のコードを記述する必要がある．

5.3 インライン関数

FSL は関数を用いて処理を定義することを 2.1.2 項で示した．しかし，同一サイクル内で複数の処理が同じ関数を呼び出した場合，関数の引数の値が競合を起こし，正しい処理を行えなくなる．そのため，同一サイクル内で複数の処理から同じ関数の呼び出しが発生する可能性がある場合，処理を複製した名前だけが異なる関数を新たに定義する必要がある．

この問題の解決方法の一つとして，コンパイラ側で関数を呼び出した数だけ複製する方法がある．しかし，この方法は同一サイクル内で呼ばれないことが分かっている関数も複製してしまい，ハードウェアリソースを余計に使用する原因となる．

他の解決方法としてインライン関数を用いる方法が挙げられる．FSL ではこのインライン関数が用意されている．インライン関数を用いた場合，各関数呼び出しの部分で関数のインライン展開が行われるため，同一サイクル内で呼び出されたとしても引数の競合は発生しない．また，同一サイクル内で呼び出されないことが分かっている関数はインライン関数にしないことで，関数の余計な複製が抑えられる．そのため，ハードウェアリソース使用量の節約に繋がる．

図 9 は実際の RISC-V プロセッサの実装において，キャッシュ内の処理で用いるアドレスからタグを得る処理をインライン関数にしたものである．3 行目で定義されている関数 isHit と 24 行目で定義されている関数 hitHandler は同一サイクル内で呼び出される可能性のある関数である．それらの関数の中では関数 getTag が呼び出されている．この関数 getTag は 1 行目で示すようにインライン関数なので同一サイクル内で呼び出されても問題がない．

アドレスからタグを得る処理などは同一サイクル内で頻繁に行うため，インライン関数化を行っている．関数化することで記述の複製を抑えることが可能となる．また，関数名によりどのような処理なのか明確になる．

Copyright © 2019 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

```
1 val (rs1Addr, rs2Addr, _) = decode.regAddr
2 val rs1Data = regfile.readRs1(rs1Addr).rs1Data
3 val rs2Data = regfile.readRs2(rs2Addr).rs2Data
```

図 10: タプル展開例

5.4 タプルを用いた複数の値の統合

FSL では，関数から値を返す際に複数の値を返すなど，複数の値を同時に扱いたい場合，複数の値の組を表すタプルを用いてこれを実現する．また，タプルの値を取り出す場合，1 行で複数の値を取り出すことが可能である．

図 10 は実際にタプルを用いた例である．1 行目の右辺の `decode.regAddr` は 3 つの値を保持するタプルであり，左辺でこのタプルの展開を行っている．このうち，1 つ目の値は `rs1Addr` に，2 つ目の値は `rs2Addr` に割り当て，2 行目と 3 行目で関数の引数としてそれぞれ用いている．一方，3 つ目の値を `_` に割り当てているような記述は値を捨てることを示す．値を捨てることが可能なため，その場で使用しない値の変数名を考える必要が無くなる．また，他の変数名との競合が発生しにくくなる．

5.5 FSL 記述の考察

5.1 節で示したように，ステージの構文を用いることでパイプライン処理の記述の可読性が高くなり，処理の把握やステージ構成の改変が容易となる．本研究でもステージの構文を用いることができたために，あるプロセッサを基にした改善案の適用および修正を容易に行うことができた．本研究ではパイプライン制御のプロセッサの実装を行ったが，パイプライン制御を行うハードウェアはプロセッサだけではない．そのため，FSL はプロセッサの記述だけでなく，パイプライン制御を行うハードウェアの記述に適していると考えられる．

また，パラメータやインライン関数によってモジュールや処理の一般化を行うことで，記述の複製を抑えることができる点から，修正が容易になることや複製によるバグの伝搬を抑えることが期待できる．

以上から，複数サイクルに渡る処理を要する記述であっても高い可読性を維持することができ，記述の複製を抑制する機能を備えているため，FSL によるハードウェア記述は高い生産性を達成できると考えられる．

6. おわりに

本稿では，ハードウェア記述言語 FSL を用いて実装した RISC-V プロセッサの評価と FSL の記述について述べた．

プロセッサの実装では，パイプラインステージ構成の異なる 7 種のプロセッサを実装した．その結果，**(6) 6 ステージ**が平均命令実行時間で最も短いプロセッサとなり，**(1) 5 ステージ**に比べ 0.83 倍となった．

また，FSL の記述については，ステージの構文やインライン関数などの機能を用いることで，記述の変更や修正が容易になることや，記述の複製を防止することが可能であることを述べた．

今後の課題として，RISC-V プロセッサの実装に関しては，分岐予測方式やキャッシュ構成の見直しによる性能の変化の調査などが考えられる．さらに，これらの設計を通じて，FSL 記述の評価を更に進めることや不足している機能の発見などを行うことが挙げられる．

参考文献

[1] RISC-V Foundation, RISC-V: The Free and Open RISC Instruction Set Architecture, `https://riscv.org/`.

[2] 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, Vol. 155, No. 228, pp. 27–32, Sept. 2015.

[3] 渡邊 誠也, 名古屋 彰, “SFL の設計思想と言語機能を受け継いだ新たなハードウェア記述言語 FSL,” 第 41 回パルテノン研究会資料集, pp. 15–22, Dec. 2015.

[4] 大西 創也, 渡邊 誠也, 名古屋 彰, “RISC-V プロセッサの FSL による実装と評価,” 第 44 回パルテノン研究会資料集, pp. 33–42, Dec. 2018.

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

[6] École polytechnique fédérale de Lausanne (EPFL), The Scala Programming Language, `http://www.scala-lang.org/`.

[7] A. Waterman and K. Asanovic (eds.), *The RISC-V Instruction Set Manual Volume I: User-Level ISA (Document Version 2.2)*, RISC-V Foundation, May 2017.

[8] UC Berkeley Architecture Research, vscale, `https://github.com/ucb-bar/vscale` (accessed Nov. 2018).

[9] freechipsproject, rocket-chip, `https://github.com/freechipsproject/rocket-chip` (accessed Sept. 2018).

[10] 冨山 修平, 清水 尚彦, “NSL で記述した RISC-V CPU とその演習環境の開発,” 第 43 回パルテノン研究会資料集, pp. 19–22, Dec. 2017.

[11] 宗藤 誠治, “RISC-V ベースのエッジ向けプロセッサーを開発 (前編),” 日経エレクトロニクス no. 1198, pp. 77–83, Nov. 2018.

[12] RISC-V Foundation, riscv-tests, 2018, `https://github.com/riscv/riscv-tests` (accessed Jun. 2018).

[13] RISC-V Foundation, riscv-gcc, 2018, `https://github.com/riscv/riscv-gcc` (accessed Jun. 2018).

[14] Intel, CYCLONE IV FPGAS FEATURES, `https://www.intel.com/content/www/us/en/products/programmable/fpga/cyclone-iv/features.html`.

Copyright © 2019 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.