東京工業大学工学部

学士論文

Verilog HDLで記述する RISC-V命令セットの 教育用アウトオブオーダ実行プロセッサ

指導教員 吉瀬 謙二 准教授

平成28年2月

提出者

学科　　情報工学科
学籍番号　12_12810
氏名　　藤浪 将

指導教員認定印		
学科長認定印		

平成27年度　学士論文内容梗概

Verilog HDLで記述する RISC-V命令セットの 教育用アウトオブオーダ実行プロセッサ

指導教員　　吉瀬 謙二　准教授
情報工学科
12_12810　　藤浪 将

高性能プロセッサの実装において，アウトオブオーダ実行は重要な手法であり，プロセッサの発展のためには，本手法を学ぶことが重要である．そのため，情報工学を専攻する大学生や大学院生は，このアウトオブオーダ実行について学ぶことがある．学習時に利用する書籍では，アウトオブオーダ実行を実現する手法の記述はあるが，実装レベルの記述が少なく，アウトオブオーダ実行プロセッサの実装方法について理解するには不十分である．

実装方法を理解する手段として，HDLを用いて記述されたプロセッサのRTLのコードを読むという方法がある．初学者がHDL記述を見て学ぶ場合，ソースコードからそのプロセッサのアーキテクチャや各ハードウェアモジュールの役割を理解することは難しいため，それらを解説する十分なドキュメントが必要である．しかし，既存のHDLで記述されたアウトオブオーダ実行プロセッサには，そのようなドキュメントが公開されていないため，学習に用いるには適していない．

本論文では，教育用のアウトオブオーダ実行プロセッサ, RIDECOREの提案と評価を行う．RIDECOREは書籍*Modern Processor Design*に沿って仕様を策定するため，これを読むことでRIDECOREのアーキテクチャの概要を理解できる．また，各ハードウェアモジュールの役割やその実装について説明したドキュメントを作成し，公開することで理解を容易にする．プロセッサの実装は広く用いられているHDLであるVerilog HDLを用いて10,000行以内で記述することで実装方法の理解を容易にする．加えて，ソースコードが利用可能であることや，開発環境が整っていることも重要であるため，RIDECOREは自由に利用可能な命令セットであるRISC-Vを採用し，開発したプロセッサのソースコードを公開する．

RIDECOREの評価として，ドキュメントとソースコードの対応関係を例示することで, RIDECOREの中身を容易に理解できることを示す．次に，Verilogシミュレータを用いることでプロセッサの詳細動作をサイクル毎に確認できるため，アウトオブオーダ実行の様子を具体的に見ることができることを示す．これらの評価結果から，初学者がアウトオブオーダ実行プロセッサの実装方法について学習する教材として，RIDECOREが優れたものであることを示す．

目　次

第1章

序論

1.1 研究目的

高性能なコンピュータを実現するためには，高性能なプロセッサが必要である．高性能なプロセッサを実現する手法として，データの依存関係がない命令をプログラムに記述された順番から入れ替えて実行する，アウトオブオーダ実行が存在する．高性能プロセッサの実装においてアウトオブオーダ実行は重要な手法の一つであり，プロセッサの発展のためには，本手法を学ぶことが重要である．そのため，情報工学を専攻する大学生や大学院生は，このアウトオブオーダ実行について学ぶことがある．

初学者が学習に利用するテキストは，理解を容易にするため実装の詳細を簡略化する場合が多く，アウトオブオーダ実行プロセッサの実装方法について学習するには適していない．しかし，実装の詳細まで記述されたテキストは内容が膨大となり，初学者が読むことは難しい．また，テキストベースの学習では，そのプロセッサを実際に動作させて詳細の挙動を確認したり，改変を加えて性能を評価することはできない．以上の理由から，初学者がアウトオブオーダ実行プロセッサの実装方法やその詳細について学習する際に，テキストのみを用いることは不十分である．

一般に，ハードウェアの実装方法を理解する手段の一つとして，HDL(Hardware Description Language) を用いて記述された RTL(Register Transfer Level) のコードを読むという方法がある．ただし，初学者がこの手段で学習を行う場合，ソースコードからプロセッサのアーキテクチャや各ハードウェアモジュールの役割を理解することは難しいため，それらを解説する十分なドキュメントが必要である．また，記述に用いる HDL は言語の学習コストを考慮すると，一般的に広く利用されているものであることが望ましい．

現在，[3] や [5] など，HDL で記述されたアウトオブオーダ実行プロセッサが幾つか公開されている．しかし，プロセッサについて解説する十分なドキュメントがないことや，一般的でない HDL で記述されていることから，初学者が学習に用いるには適していない．

本論文では，教育用のアウトオブオーダ実行プロセッサ, RIDECORE(RIsc-v Dynamic Exection CORE) を提案する．初学者が用いる教材として，提案するプロセッサは以下の条件をみたすことが望ましい．

1. ソースコードが公開されていること

2. コンパイラ，アセンブラなどの開発環境が利用可能であること

3. プロセッサのアーキテクチャや，その実装方法が容易に理解できること

1., 2. の条件を満たすため，自由に利用可能な命令セットである RISC-V[1] を採用し，開発したプロセッサのソースコードを公開する．

3. の条件を満たすため，*Modern Processor Design*[2] に沿って仕様を策定し，これを読むことで RIDECORE のアーキテクチャの概要を理解できるようにする．また，各ハードウェアモジュールの役割やその実装について説明したドキュメントを作成し，公開することで理解を容易にする．さらに，プロセッサの実装は広く用いられている HDL である Verilog HDL を用いて 10,000 行以内で記述することで，実装方法の理解を容易にする．

1.2 本論文の構成

本論文の構成について述べる．第 2 章ではアウトオブオーダ実行について簡単に説明し，関連研究についても述べる．第 3 章では，提案する教育用アウトオブオーダ実行プロセッサの仕様を述べる．第 4 章では，提案したプロセッサの評価及び考察を述べる．第 5 章では，本論文のまとめ及び今後の課題について述べる．

第2章

研究の背景

この章ではまず，アウトオブオーダ実行の概要について簡単に説明し，次に関連研究について述べる．以下の議論では，プロセッサと表記した場合，スーパースカラプロセッサであることを前提とする．

2.1 アウトオブオーダ実行

2.1.1 概要

プログラムをプログラムに記述された順序通り実行する方式をインオーダ実行と呼ぶのに対し，演算結果が変わらない範囲でプログラムの実行順序を入れ替える実行方式をアウトオブオーダ実行と呼ぶ．アウトオブオーダ実行による高速化について，以下の命令列を用いて解説する．

```
命令1 R01 := Mem[R02+4]
命令2 R03 := R01 + R01
命令3 R11 := R12 * R12
命令4 R13 := R13 + R11
```

命令列より，命令2は命令1の演算結果であるR01を，命令4は命令3の演算結果であるR11をそれぞれ使用することがわかる．ここで，命令3は命令1,2の実行結果を使用しないことを考えると，命令群1,2と命令群3,4の間にはデータ依存関係がなく，これら2つの命令群は並列に実行可能であることがわかる．

しかし，この命令列をインオーダ実行のプロセッサで実行する場合，命令1,2の実行が終了するまで命令3の実行を開始することはできない．特に，命令1のメモリアクセスのように実行に時間がかかる命令が入る場合，命令開始が遅くなり，結果として全体の実行時間が遅くなる原因となる．

そこで，アウトオブオーダ実行を行う場合を考える．アウトオブオーダ実行プロセッサには，実行待機中の命令を保持しておく，リザベーションステーション(以下RSと表記する)が存在する．アウトオブオーダ実行プロセッサは

InstNo.	Operation	Opr1	OprRdy1	Opr2	OprRdy2	Ready= OprRdy1 & OprRdy2	Destination
命令1	Load	R02	1	NONE	1	1	R01
命令2	Add	命令1	0	命令1	0	0	R01
命令3	Mul	R12	1	R12	1	1	R11
命令4	Add	R13	1	命令3	0	0	R13

図 2.1: Reservation Station の例

新しく命令を受け取った際に，実行待機中の命令や現在実行中の命令との間のデータ依存関係を解析し，新しい命令を RS に入れる．上記命令群をすべて保持した場合の RS を図 2.1 に示す.

RS には，各オペランドが揃ったことを示す OprRdy と，すべてのオペランドが揃い実行可能な状態となったことを示す Ready が記録されている．Opr には OprRdy が 1 の場合はオペランドの値が格納されており，OprRdy が 0 の時はオペランドを生成する命令番号 (InstNo.) が格納されている．RS 中の命令は実行ユニットから実行結果を受け取り，自分の必要なオペランドの値であった場合はその値を Opr に保持し，OprRdy を 1 とする．プロセッサの処理が進むことで依存関係が解消され，Ready が 1 となった時に該当命令は実行可能状態となる．

図 2.1 に示す通り，アウトオブオーダ実行プロセッサで上記命令列を実行すると，命令 1,2 の実行終了を待たずに，命令 3 が実行可能状態となる．このように，アウトオブオーダ実行プロセッサは依存関係のない命令を先に実行することで，命令の並列度を向上させることが可能である．この並列度の向上により，計算を高速に実行できる.

2.2 関連研究

以下では関連研究として，HDL で実装された教育，研究用アウトオブオーダ実行プロセッサの概要について述べる.

2.2.1 UCBerkeley

RISC-V 命令セットを開発した UCBerkerey では，RISC-V 命令セットを用いた教育，研究用アウトオブオーダ実行プロセッサ, BOOM[3] の開発が進行中である．BOOM は 64bit のプロセッサであり，Linux を動作させることが可能で

ある．ASIC 及び FPGA 両方に実装可能で，ZC706 ボード [4] 上にて動作周波数 50MHz で動作する．2 命令同時発行で実装した場合，ベンチマークプログラム CoreMark での IPC は 1.26 である．

BOOM は，同大学で開発された Chisel という独自の HDL を用いて，9,000 行程度で実装されている．BOOM を教育用として用いる場合には，広く用いられていない HDL を学ぶ必要がある上に，各ハードウェアモジュールの実装について十分な解説がされていないため，初学者が学習に用いることは難しい．

2.2.2 IIT

IIT(Indian Institutes of Technology) では，RISC-V 命令セットを用いた研究用アウトオブオーダ実行プロセッサ，SHAKTI[5][6] I-Class の開発が進行中であり，現在アルファバージョンが公開されている．ASIC, FPGA の両方に実装することが可能である．FPGA に実装する場合, Artix-7 上での動作周波数は 110MHz であり，CoreMark での IPC は 1.18 である．

SHAKTI は Bluespec System Verilog を用いて，10,000 行程度で記述されている．SHAKTI に関しても，広く用いられていない HDL を学ぶ必要がある上に，各ハードウェアモジュールの実装について十分な解説がされていないため，初学者が学習に用いることは難しい．

2.2.3 東京大学

東京大学の坂井・入江研究室では，ALPHA 命令セットを用いた，研究用のアウトオブオーダ実行プロセッサ [7] を実装している．Altera 社の FPGA である Stratix デバイスに実装を行っており，最大動作周波数は 50.54MHz，同研究室で開発されたシミュレータ「鬼斬」による IPC 値は 0.839 である．ただし，ソースコードが公開されていないため，教育に用いることはできない．

第3章

教育用アウトオブオーダ実行プロセッサRIDECOREの提案

前章の関連研究で紹介したアウトオブオーダ実行プロセッサは，広く用いられていないHDLで実装されていることや，プロセッサを解説する十分なドキュメントが無いことから，教育用として用いることは難しい．本章では，これらの問題点を解決するために提案する，教育用アウトオブオーダ実行プロセッサRIDECOREの仕様について述べる．

3.1 概要

まず，基本的な仕様について述べる．RIDECOREは，アドレス，データともに32bitの，2命令同時発行を行うアウトオブオーダ実行プロセッサである．例外や割り込み処理，システムレジスタなどは未実装であるため，OSを動作させることはできない．命令セットアーキテクチャとしては，RISC-Vのサブセットである RV32IM を採用した．RV32IMとは，最も基本的な32bitの整数演算/分岐/メモリアクセス命令群に乗除算命令を追加したものである．しかし，RV32IMのすべてを実装したわけではなく，簡単のため，除算，剰余演算，1-byte/2-byteのLoad/Store命令の実装は省略した．除算及び剰余演算はnopとして処理され，1-byte/2-byteのLoad/Store命令は4-byteのLoad/Store命令に変換される．

図3.1にRIDECOREの概略図を示す．RIDECOREは6段のパイプラインとなっており，それぞれのステージ名をInstruction Fetch, Instruction Decode, Dispatch, Select and Wakeup, Execution, Completeと呼ぶ．以下，本論文ではそれぞれのステージを，IF/ID/DP/SW/EX/COMと略記する．

RIDECOREではレジスタリネーミングを行うため，リネーミングテーブルとRRF(Renamed Register File)を持つ．RRFのエントリ数はリオーダバッファと同じであり，ハードウェアとしては独立しているが，2つを合わせて1組のセットとして用いる．これにより，RRFのエントリ番号とリオーダバッファの

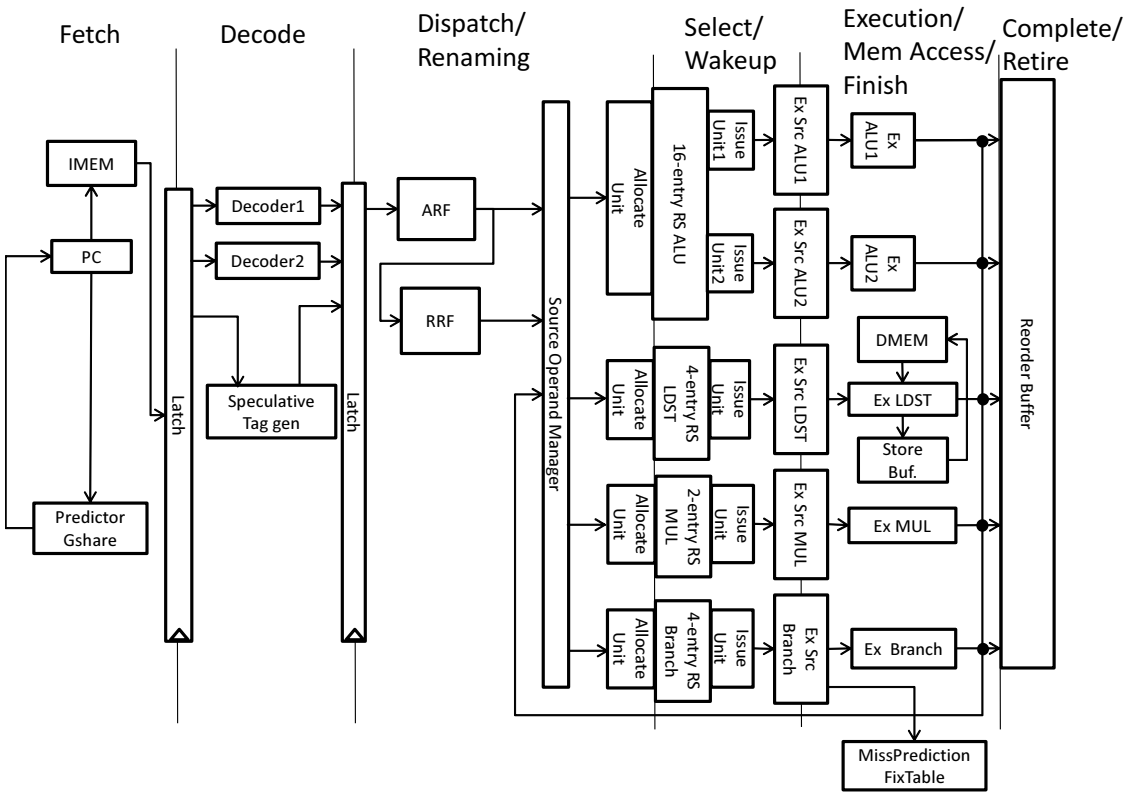


図 3.1: パイプライン概要図

エントリ番号が一致するため，管理するエントリ番号は1つでよい．

RS，演算器の仕様を表3.1に示す．RSのリードポート数を極力減らすため，RSは分散型を採用した．また，投機実行の実装を簡単にするため，分岐命令，ロード・ストア(LDST)命令を登録するRSはFIFOとし，同種の命令同士の実行順序入れ替えを禁止する．

ストアバッファは，32エントリを持つ優先度付きの連想検索が可能なメモリである．ストアバッファを用いることで，ストア命令がリオーダバッファから出る前にロード命令を実行できるため，ロード・ストア命令を高速に処理できる．

分岐予測器にはGshare方式[8]を採用する．何故なら，Gshare予測器はシンプルな構成で高い予測精度を達成する事が可能であり[9]，FPGAへの実装に適していると考えられるためである．

キャッシュは未実装であり，命令メモリ(以後IMEM)及びデータメモリ(以後DMEM)を持つ．キャッシュを実装する場合，キャッシュミスが発生した際のストールロジックなどを追加する必要がある．

表 3.1: RS, 演算器仕様

	ALU	LDST	分岐	乗算
RS エントリ数	16	4	4	2
演算器数	2	1	1	1
演算器レイテンシ	1	2	1	1

3.2 仕様詳細

以下では RIDECORE の仕様について，パイプラインの各ステージごとに述べる．

3.2.1 IF ステージ

IF(Instruction Fetch) ステージでは，IMEM から 2 つの命令のフェッチを行い，分岐予測器を用いて PC の更新を行う．IF ステージの回路を図 3.2 に示す．

IMEM は 1 エントリが 128bit である．RISC-V 命令セットの命令長は 32bit なので，1 エントリで 4 つの命令を持つことになる．PC[31:4] を用いて IMEM から 4 命令分を読みだし，PC[3:2] を用いて 4 命令から 2 命令を選択する．選択する回路を図 3.3 に示す．基本的には 4 命令から 2 命令を選択するが，PC が 8 の倍数でない場合，2 つ目の命令を無効とするために Invalid 信号を出す．例えば PC=0x2c の場合，IMEM からは 0x20/0x24/0x28/0x2c 番地にある命令が出力され，図 3.3 の回路における Insn2 の値は，0x20 番地の命令となってしまうからである．

図の複雑化を防ぐため Invalid Insn1 には常に 0 を書き込むよう記載しているが，実際には分岐予測ミスなど，IF ステージの命令を kill する必要がある時に 1 となるような回路となっている．

3.2.2 ID ステージ

ID(Instruction Decode) ステージでは，命令のデコードを行う．投機的な実行を行っている場合，投機タグ (Speculative Tag) の付与も行う．

ID ステージの回路を図 3.4 に示す．Decoder 部は後段のステージで用いる様々な値を生成している．主だった値としては，ソース/デスティネーションレジスタの番号，登録する RS の番号，ALU で行う演算の種類を表す値などである．以降図中では，Decoder で生成した値を青色で示す．

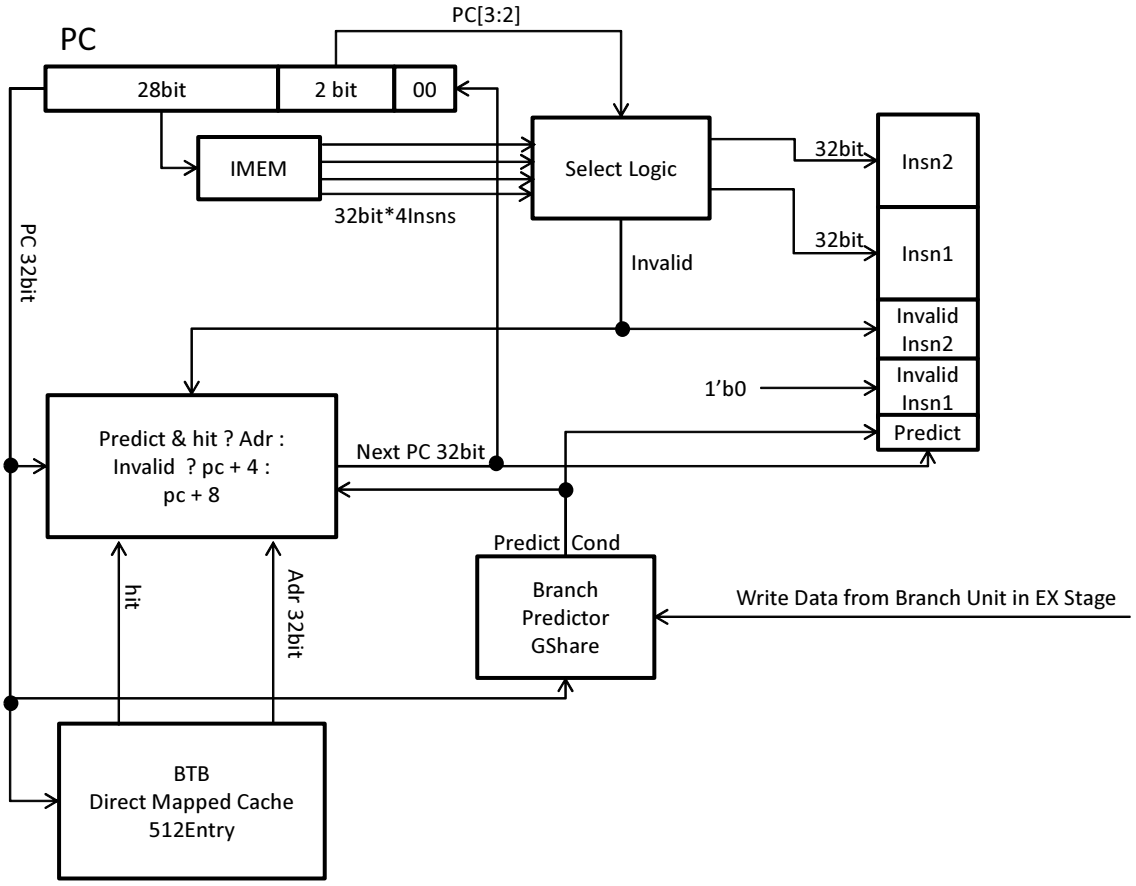


図 3.2: IF ステージ

ID ステージでは IF でフェッチした 2 命令が分岐命令であるかを確認し，分岐命令である場合にはその命令が EX ステージで実行が終了するまでの間，以降の命令を投機的に実行するために投機タグを付与する．投機タグの管理は，Speculative Tag Generator 及び Miss Prediction Fix Table で行っている．投機実行を正しく行う方法を，図 3.5 を用いて具体的に説明する．図は，分岐命令を 2 つデコードしたあとの各命令群の投機タグと Miss Prediction Fix Table を示している．投機タグは Speculative Tag Generator が，00001 から順に左ローテートしながら割り当てていく．すべての投機タグが割り当てられている場合は IF/ID をストールさせる．

Miss Prediction Fix Table には各投機タグの状態を示す 2 つの値，Value と Valid が格納されている．Valid は，該当する投機タグのついた命令が投機的であることを示す．Value は該当する投機タグの他の投機タグとの依存関係を示している．図において，branch1 で分岐予測ミスが発生した場合，該当する投機タグ:00010 の Value:00110 を読み出す．このとき，分岐予測ミスによって無効

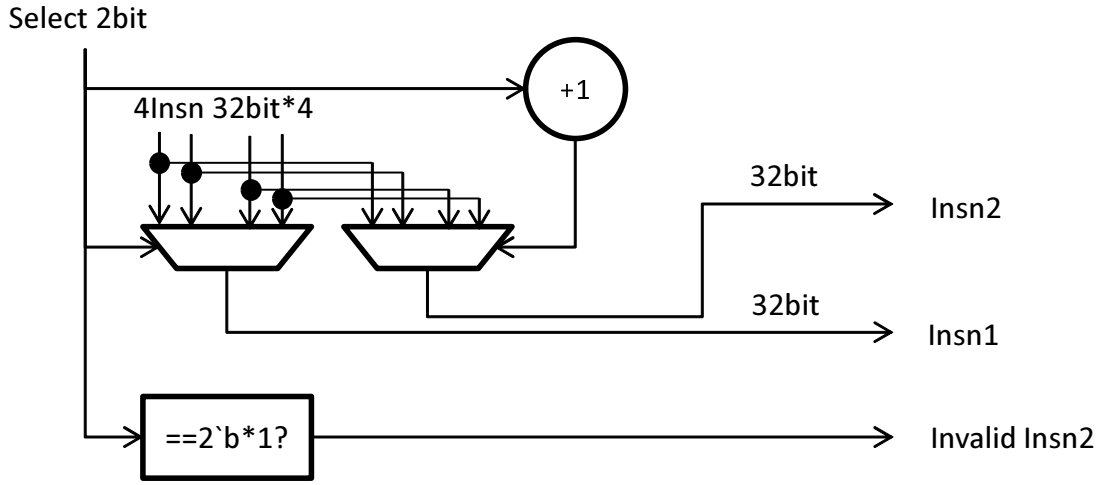


図 3.3: Select Logic

化される命令群は，Value:00110 と AND 演算を行った結果が 00000 とならない投機タグを持つ命令群となる．00110 の場合は，投機タグが 00100 または 00010 となる命令であり，branch1 以降の命令のみが無効化されることを示している．

また，1 番目の命令 Insn1 が分岐命令であり，分岐が起こると予測している場合，2 番目の命令 Insn2 は無効な命令であるため，Invalid Insn2 を 1 とする．

3.2.3 DP ステージ

DP(DisPatch) ステージでは，ソースレジスタの値及びリネーミングテーブルへのアクセスを行い，ソース/デスティネーションレジスタのリネーミングを行う．その後，リオーダバッファへのエントリを割り当て，所定の RS に命令の登録を行う．DP ステージの回路の前半部分を図 3.6，後半部分を図 3.7 に示す．

まず前半部分 (図 3.6) について説明する．前半部分では主にリネーミングを行っている．ARF(Architected Register File) の各エントリには，COM ステージを終えて完了状態にあるレジスタの値 (Data) と，現在のリネーミング情報 (RRFTag, Busy) が格納されている．また，RRF の各エントリには，該当する命令の終了状態にあるレジスタの値 (RRFData) と，実行結果が有効であることを示す情報 (RRFValid) が格納されている．ARF，RRF を用いたリネーミング回路を図 3.8 に示す．また，ARFBusy と RRFValid から求められる，該当するレジスタの計算状況及び RS に登録する値を表 3.2 に示す．

以下，要求したレジスタのエントリを rs と書く．ARFBusy が 0 の場合は，プロセッサ内に rs をデスティネーションとする命令が存在しないことを示しており，ARF に格納された値が rs の値となることを示している．

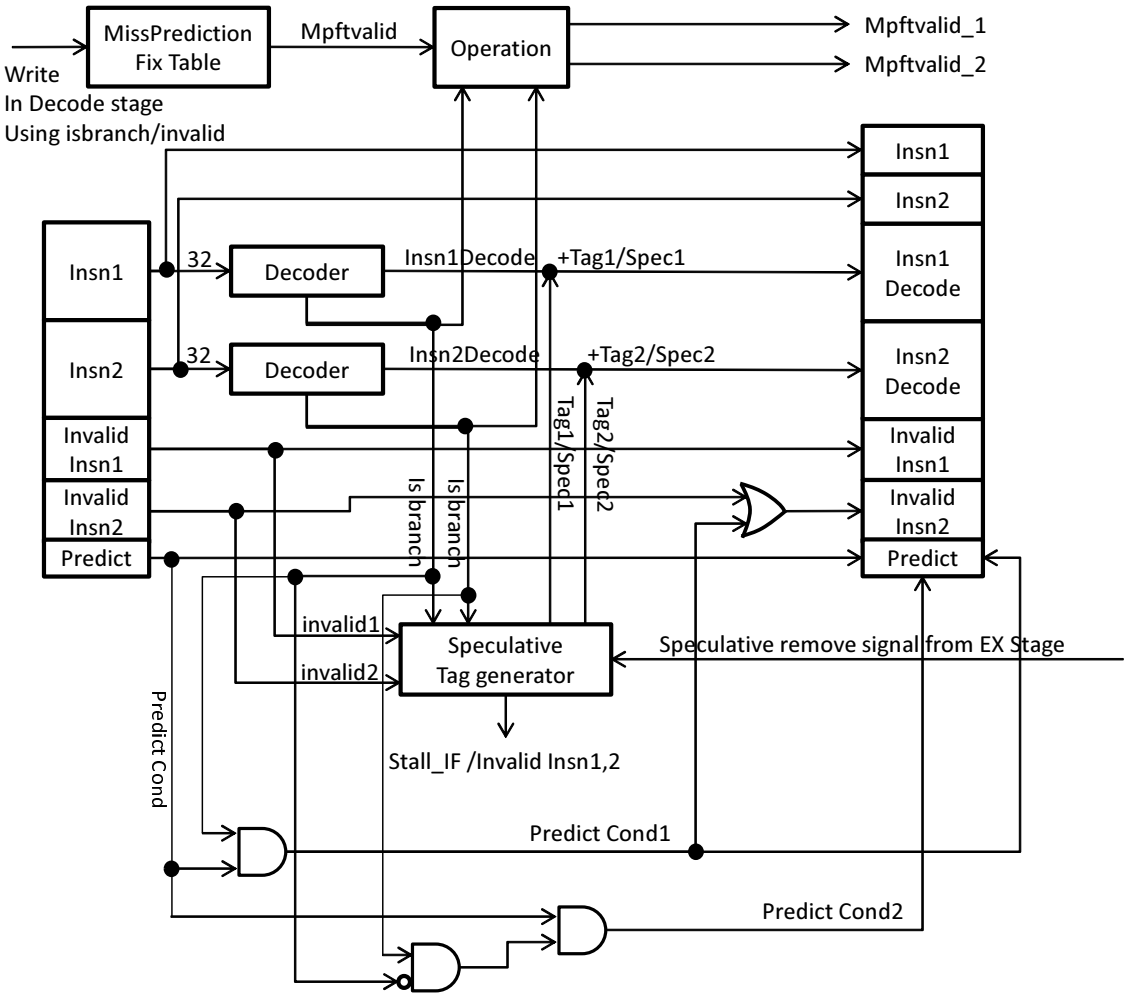
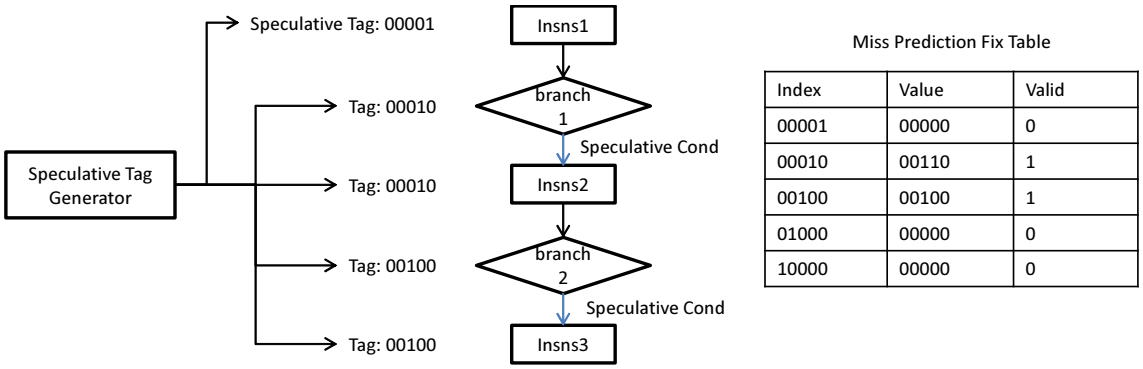


図 3.4: ID ステージ



Miss Prediction Fix Table

Index	Value	Valid
00001	00000	0
00010	00110	1
00100	00100	1
01000	00000	0
10000	00000	0

図 3.5: 投機実行

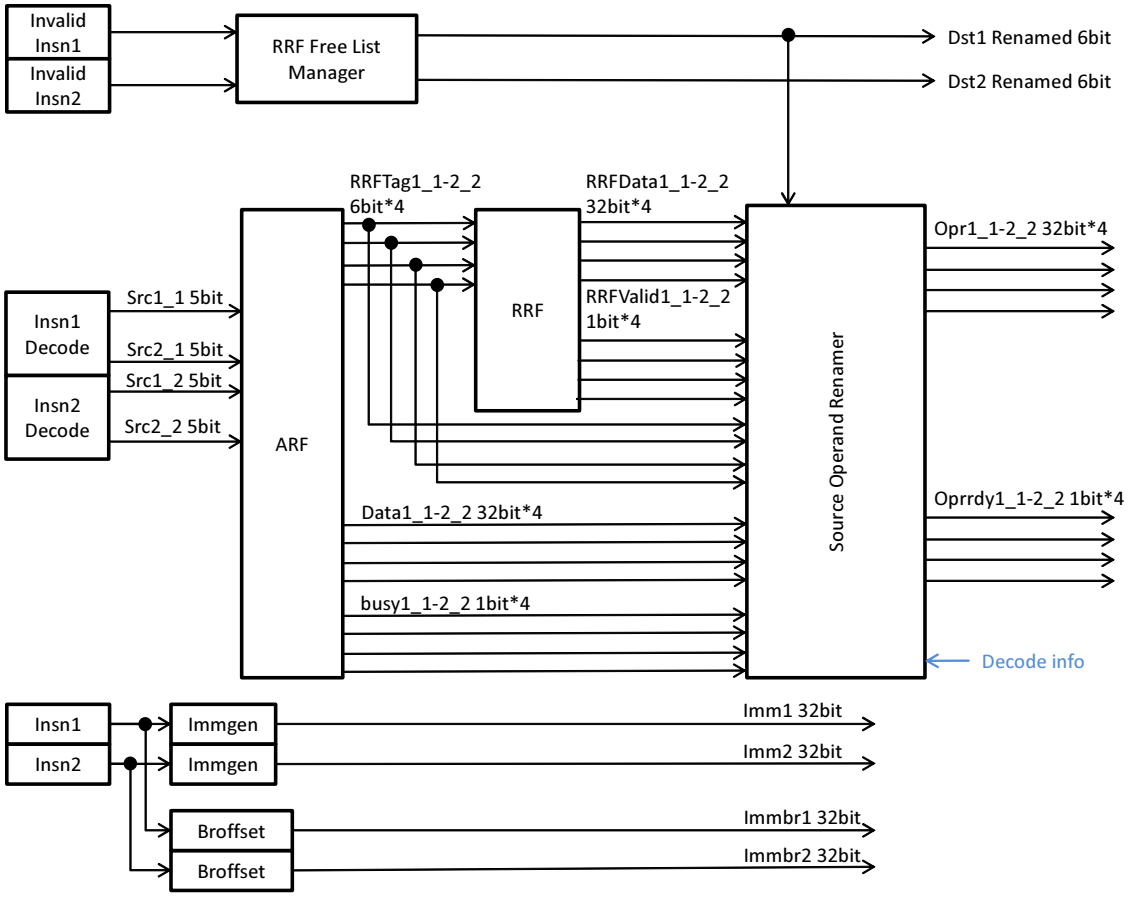


図 3.6: DP ステージ前半

表 3.2: 計算状況

ARFBusy	RRFValid	計算状況	RS 登録値
0	*	計算完了	ARF.Data
1	1	計算終了	RRF.Data
1	0	計算中	RRFTag

一方，ARFBusy が 1 の場合，プロセッサ内に rs をデスティネーションとする命令が存在することを示している．rs の RRFValid が 1 の場合，rs の値を生成する命令が EX ステージを終了し，RRF に実行結果を保持していることを示している．そのため，RRF の値を rs の値とする．RRFValid が 0 の場合，rs を生成する命令は発行されているが EX ステージを終了しておらず，値が未生成である

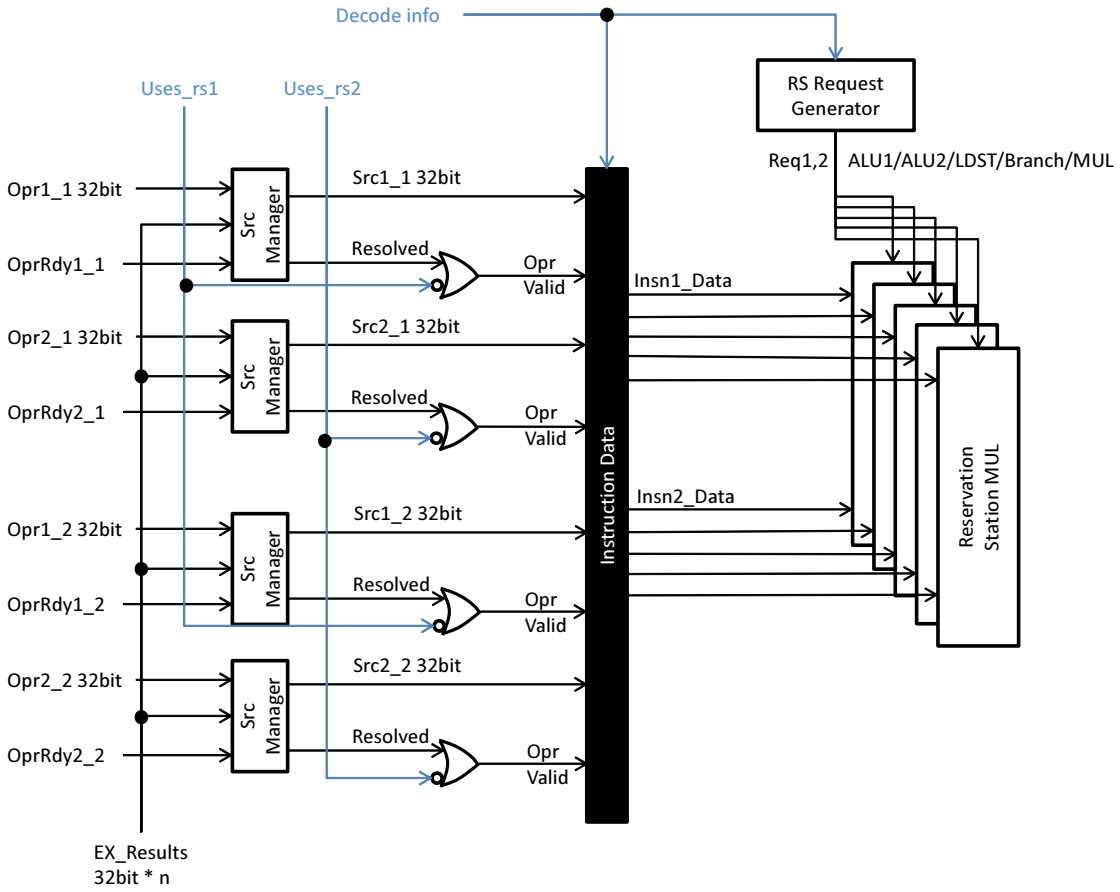


図 3.7: DP ステージ後半

ことを示している．この際，RS には RRFTag の値を登録しておき，RS は該当命令の終了時に値を取得する．

リオーダバッファと RRF が合体している設計のため，RRF が割り当て可能ならばリオーダバッファも割り当て可能であり，RRF のエントリ番号を用いてリオーダバッファにアクセス可能である．なお，RRF 及びリオーダバッファに 2 命令分以上の空きがない場合は IF/ID/DP をストールさせる．

リネーミング情報の格納されるリネーミングテーブルは，投機実行を行う際に分岐予測ミスが生じた場合，分岐が起こる前の情報を復元する必要がある．そのため，リネーミングテーブルは投機タグの数だけ用意しておき，常にバックアップを取りながら実行を行う．下部に記載されている Immgen 及び Broffset は，命令中で指定する即値を抽出する回路である．

次に後半部分 (図 3.7) について説明する．後半部分では，命令を RS に登録する．図中の Src Manager で命令 1,2 の最終的なオペランド (オペランドが揃っていない場合は，RRF のエントリ番号) が生成される．Src Manager は，EX ス

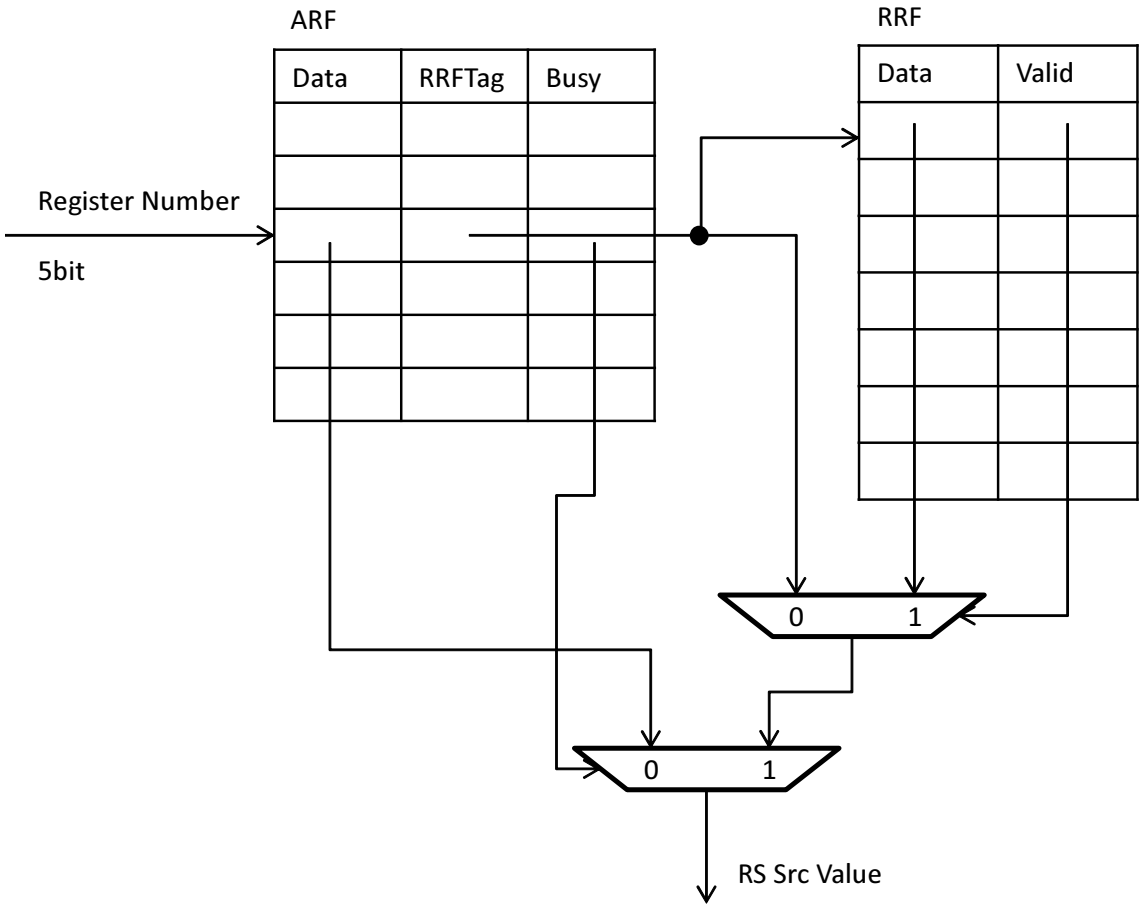


図 3.8: ARF，RRF を用いたレジスタリネーミング

テージからの実行結果を監視し，DP ステージ中の 2 命令が必要とする値ならばその値を選択する回路である．

RS は，命令 1,2 の種々のデータと Write Enable を受け取る．Write Enable は，RS Request Generator が Req1,2 として RS に出力する．RS Request Generator の図を図 3.9 に示す．ID ステージで計算した，登録する RS の種類を示す RSNum を用いて，各 RS への命令 1,2 の Write Enable である Req1,2 を生成する．

Req1,2 は RS 中の Allocate Unit が受け取る．Allocate Unit は RS 中の開いているエントリを 2 つ探し，Req があればそれを RS に書き込む．インオーダ発行を行う RS(LDST 及び分岐命令) の場合，RS はリングバッファを用いた FIFO となるため，空きエントリを探すのは容易である．しかし，アウトオブオーダ発行を行う RS の場合，発行可能になった命令から順次空きができるため空きエントリが飛び飛びになってしまい，それを探すのは容易ではない．

空きエントリを探す回路を図 3.10 に示す．まず，RS のエントリが空きである

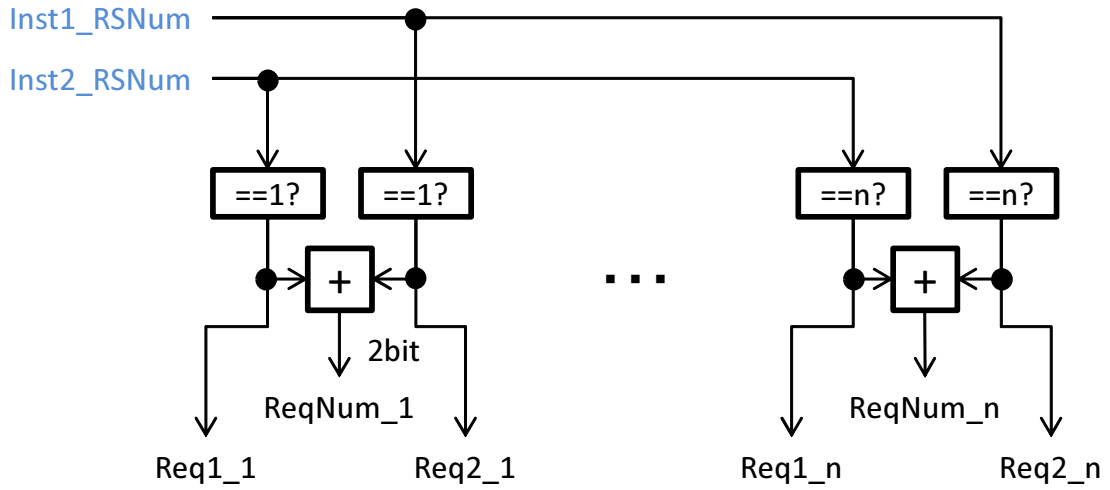


図 3.9: RS Request Generator

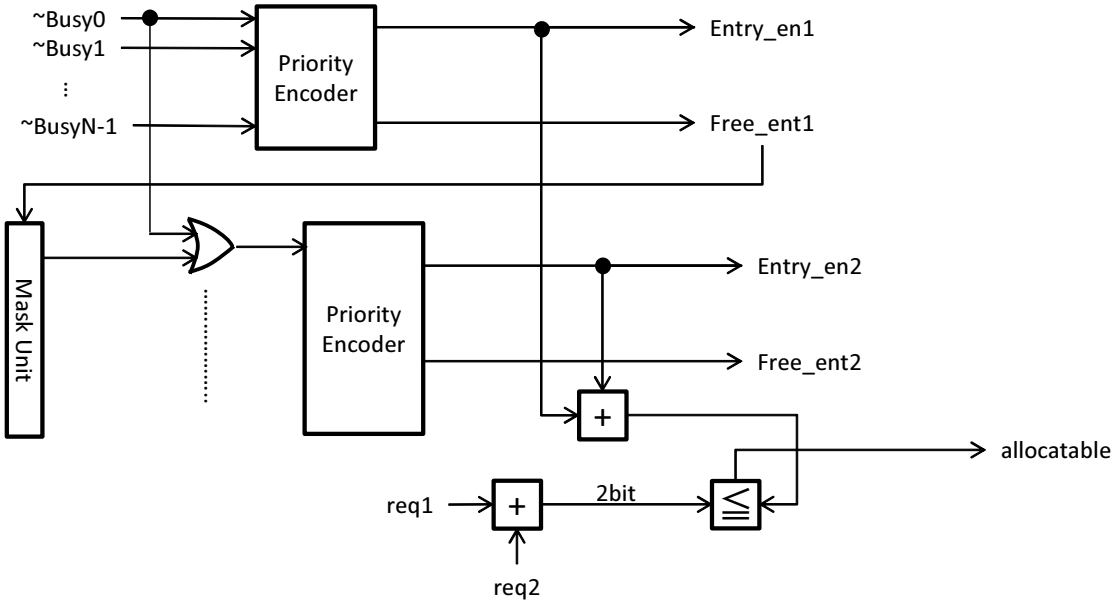


図 3.10: RS Free Entry Finder

ことを示す Busy の列を 1 段目のプライオリティエンコーダに入れる．プライオリティエンコーダは Busy が 1 であるエントリ (Free_ent1) と，出力したエントリが有効なものであることを示す (Entry_en1) を出力する．次に，求めた 1 つ目のエントリ (Free_ent1) を Mask Unit で取り除いた Busy の列を，2 段目のプライオ

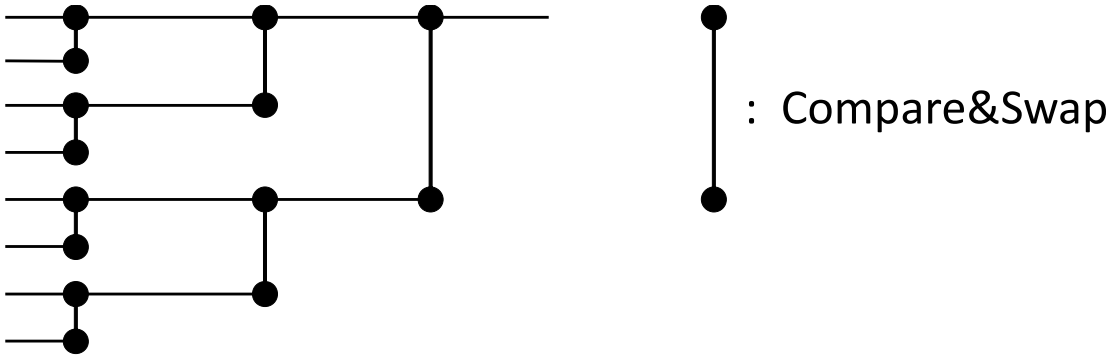


図 3.11: 最小値選択回路

リティエンコーダに入れて2つ目のエントリ (Free_ent2, Entry_en2) を求める．

このようにして，空きエントリを最大2つまで求める．Entry_en の数と Req の数を用いて，RS に Req 分の空きがある場合は Allocatable が 1 となる．全 RS で Allocatable が 1 となった時に，命令を RS に書き込むことが可能となる．

3.2.4 SW ステージ

SW(Select and Wakeup) ステージでは，RS に入っている命令の中でオペランドが揃い，実行可能な状態となった命令を1つ選択し，EX ステージで実行を開始するために命令を発行する．命令を発行した際に，RS の該当エントリは RS から消える．

インオーダ発行を行う RS の場合，発行する命令を選択する余地は無いが，アウトオブオーダ発行を行う場合は複数の選択肢が生まれる．単純かつ高いパフォーマンスを得られる選択方法として，発行可能な命令の中から一番古い命令を優先的に発行する，Oldest-First というアルゴリズムが存在する．RIDECORE ではこの Oldest-First に近い (厳密には Oldest-First でない) アルゴリズムを採用する．

その実現方法は以下のようである．RRF 及びリオーダバッファのエントリ番号 (RRFTag) は 0 から順に割り当てられるため，基本的には RRFTag の値が小さいほど古い命令である．したがって，RS 中にある発行可能 (rdy=1) な命令のうち RRFTag の値が最も小さいものを，図 3.11 のような回路を用いて選択すれば良い．この選択回路に入力する値として，RRFTag の上位に rdy の否定を結合したものを用いると，rdy=1 のものが優先して選択可能になる．しかしこれでは問題が生じる．RRF はリングバッファのため，RRFTag が上限を超える (RIDECORE の実装では 63) と再び 0 に戻る．RS に RRFTag=63 の命令が残っていて，次に RRFTag=0 の命令が RS に登録された場合，RRFTag=0 の方を古い

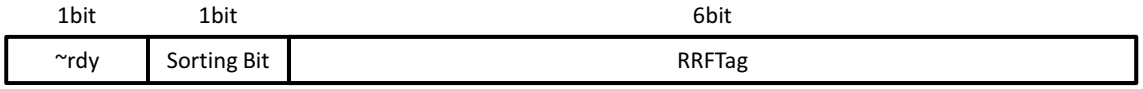


図 3.12: Sorting Data Format

命令であると計算してしまう.

この問題を解決するため，図3.12のようにSorting bit[11]と呼ばれる1bitの値をRRFTagの上位に結合する．このbitはRSに登録する際には1として書き込むが，RRFTagが上限を超えた際にRSに通知を行い，RS内のすべてのSorting bitを0とする．この操作により，古い命令を正確に求めることが可能となる．

RIDECOREの仕様ではRRFTagを1サイクルに最大2つ割り当てるため，RRFTag番号63と0の命令を割り当てる際に，RRFTag=0のSorting bitも0となってしまい，厳密にはOldest-First ではないが，限定的な問題である．

3.2.5 EXステージ

EX(EXecution)ステージでは，SWステージで発行された命令を実行する．実行が終了した命令は，実行結果を該当するRRFに書き込み，RRFValidを1とした上でリオーダバッファに終了通知を行う.

実行ユニットはALUが2つ，乗算器が1つ，ロード・ストアユニットが1つ，分岐ユニットが1つである．ALUの回路図を図3.13, ロード・ストアユニットの回路図を図3.14, 分岐ユニットの回路図を図3.15 に示す．乗算器についてはALUと同じのため略する．また，図中のKill Genの回路を図3.16に示す．この回路は分岐予測に失敗した際に，現在演算中の計算が破棄すべき命令かを判定する.

いずれの回路も，リオーダバッファへの終了通知(ROB WE)及びそのエントリ(RRF Write Addr)，RRFに書き込む実行結果(RRF Write Data)を最終的な出力としている．これらの他に追加の操作を行うロード・ストアユニット及び分岐ユニットについて説明する.

まず，ロード・ストアユニットについて説明する．ロード・ストアユニットは内部が2段のパイプラインとなっている．RSから受け取った命令がロード命令の場合，まず1段目でDMEMとストアバッファにアクセスする．ストアバッファは，命令は終了したがメモリに書き込んでいない未ストアのデータとそのアドレスのペアを持ち，アドレスを与えると連想検索を行い，最新の未ストアデータを返す．2段目においてDMEMからデータを受け取る．未ストアのデータがある場合はストアバッファから受け取ったデータ，ない場合はDMEMから受け取ったデータを実行結果としてRRFに書き込む.

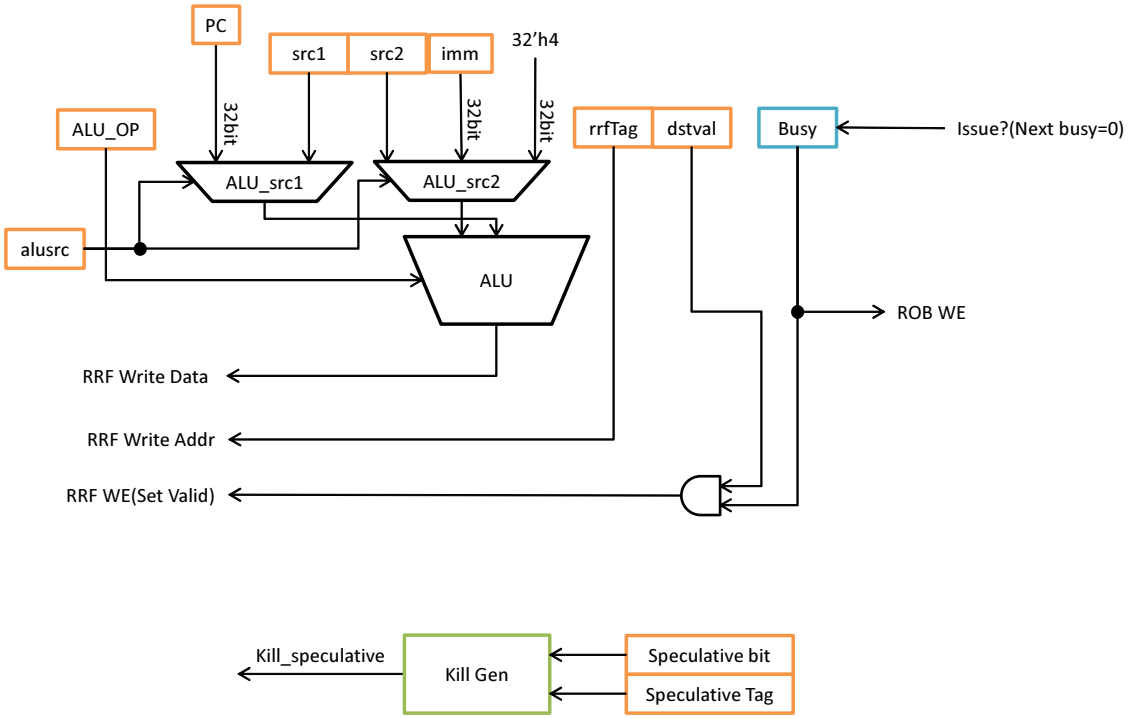


図 3.13: ALU

RS から受け取った命令がストア命令の場合，1 段目でストアバッファにアドレス，データを書き込み，2 段目でリオーダバッファに終了通知を行う．COM ステージを経てストア命令が完了した際に，該当する命令はストアバッファからメモリに書き込みを行うことが許可され，ロード命令が実行されていない間に DMEM に書き込みを行う．

次に，分岐ユニットについて説明する．分岐ユニットは条件に従って変わる分岐先アドレスを計算し，IF ステージで予測した分岐先との比較を行う．分岐予測が成功した場合，Miss Prediction Fix Table の更新及び投機実行のための情報の破棄を行う．分岐予測が失敗した場合，プロセッサを分岐予測を行った直前の状態まで復元する．いずれの場合も情報を復元するためにストールが発生する．なお，COM ステージで分岐予測器に情報を通知するため，リオーダバッファには終了通知に加えて分岐先アドレスと分岐条件の情報を書き込む．

3.2.6 COM ステージ

COM(COMplete) ステージでは，リオーダバッファに保存されている実行が終了した命令をインオーダに完了する．完了する命令数は最大で 2 つである．完了の際，RRF に書き込まれていた実行結果を，リネーミングテーブルを見て

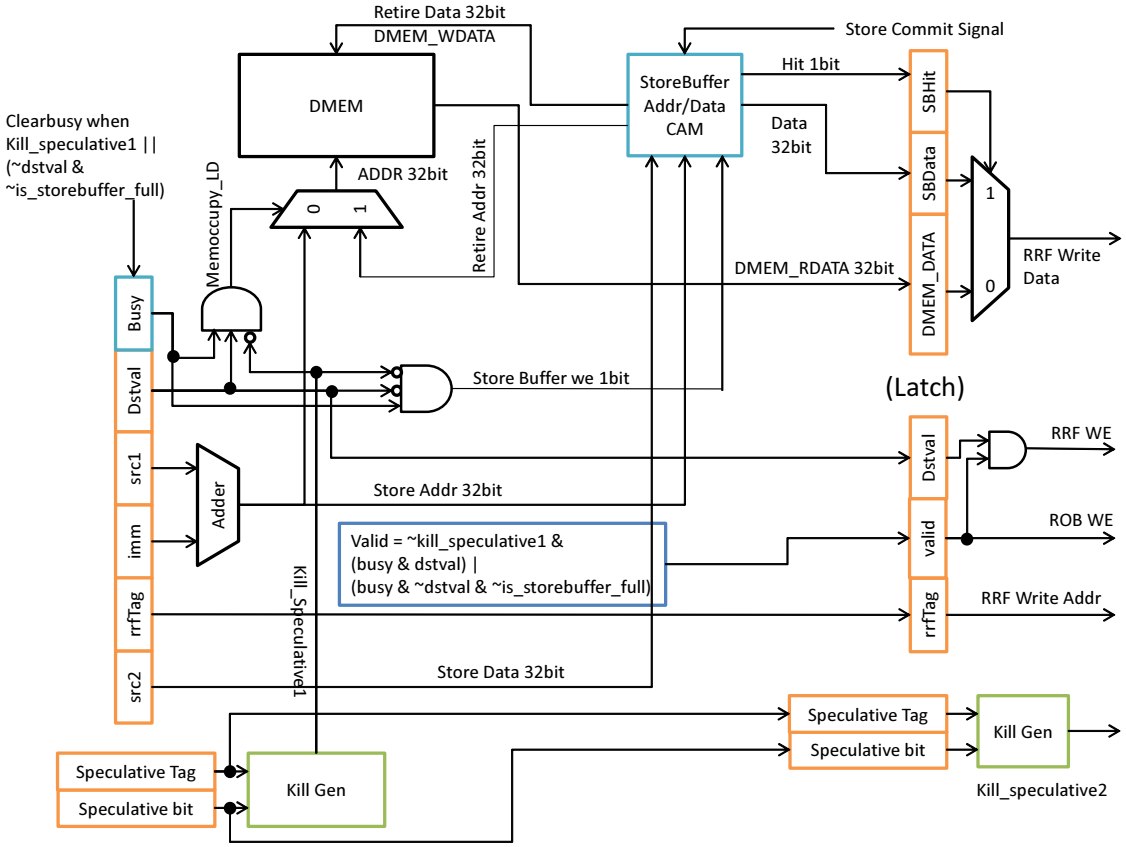


図 3.14: ロード・ストアユニット

から必要に応じて ARF に移動する.

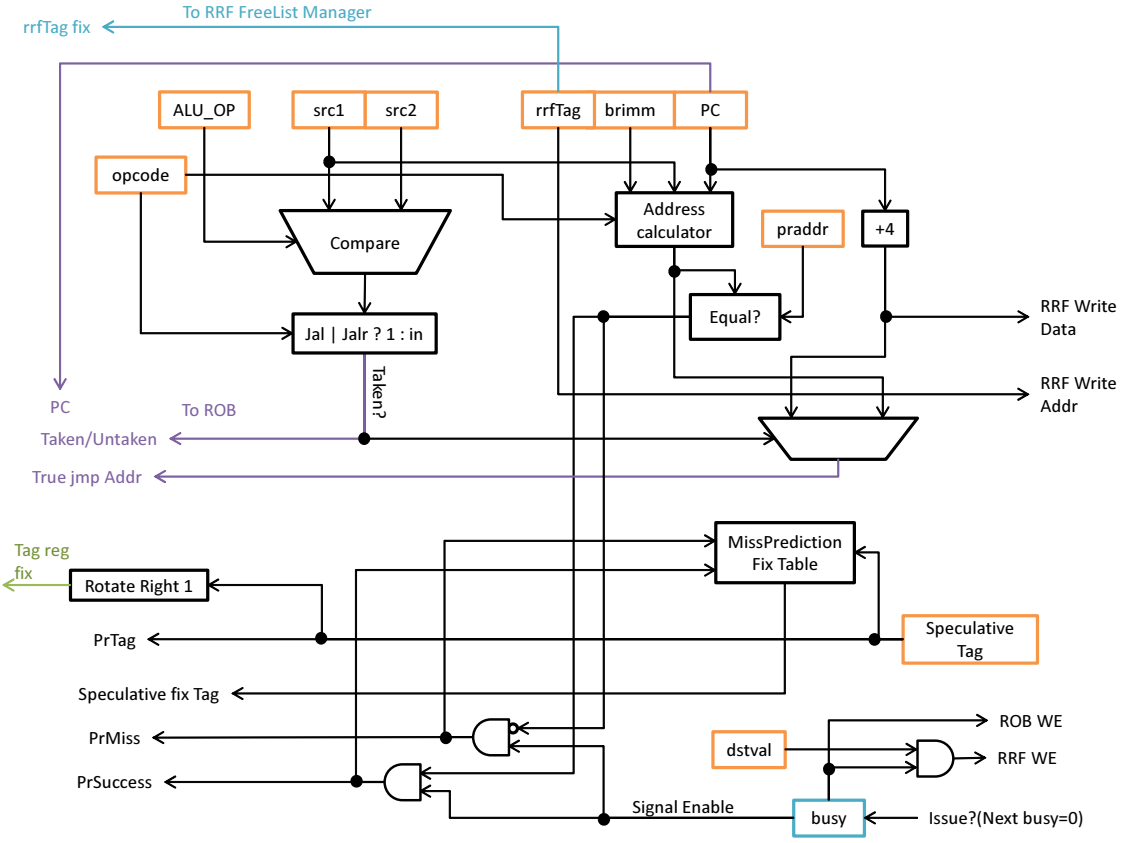


図 3.15: 分岐ユニット

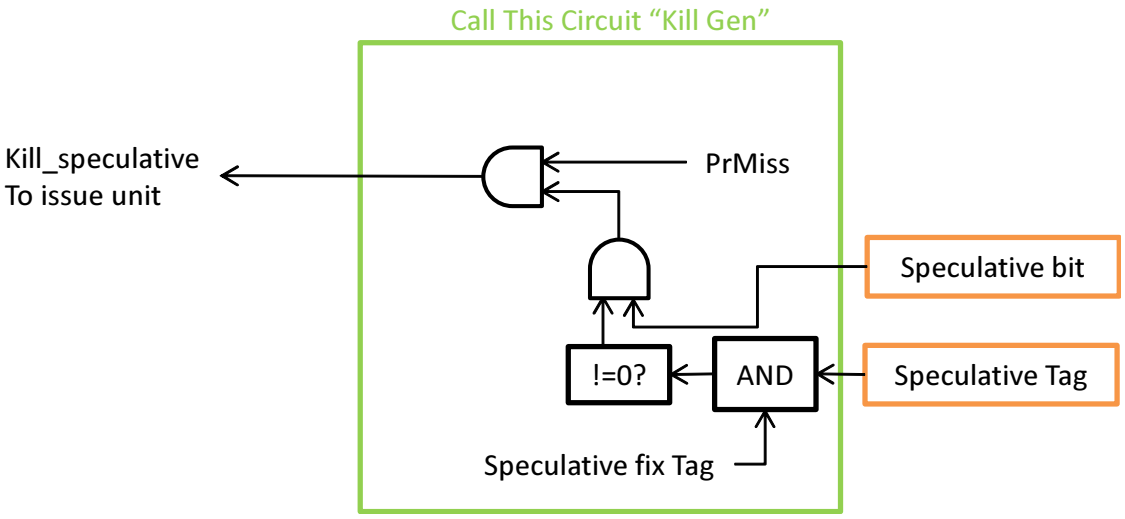


図 3.16: Kill Gen

第4章

評価

本章ではまず，前章で述べたRIDECOREをFPGAに実装して評価する環境を提示する．次に，RIDECOREを性能や有用性の観点から評価する．

4.1 評価環境

アウトオブオーダ実行プロセッサはその機能の複雑さから回路規模が大きくなる傾向にある．そのため，大規模回路向けのFPGA評価ボード，VC707[12]に実装を行う．RIDECOREを用いたシステムの概要を図4.1に示す．

命令メモリ及びデータメモリの初期化は，ホストコンピュータからUART通

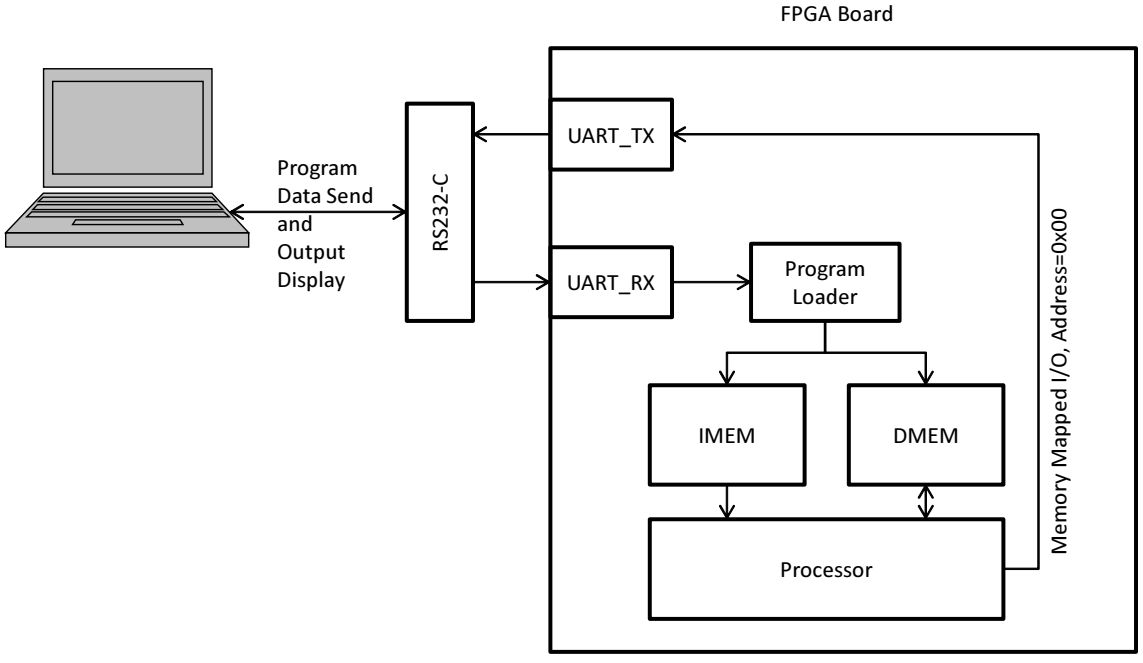


図 4.1: VC707上に実装するRIDECOREシステムの概要図

信でFPGAに初期化用データを送信し，Program Loaderが受信したデータを両メモリに書き込むことで実現する．また，RIDECOREの文字出力については，メモリアドレス0番地に文字出力用のバッファが用意されており，バッファからUARTを通じてホストコンピュータにデータを送信する．後述する最大動作周波数の制限から，RIDECOREに用いるクロックの周波数は50MHzとする．

4.2 正常な動作の検証

一般的に，HDLで記述した回路が正しく動作していることを確認するためには，シミュレータなどを用いてすべてのレジスタや信号線の値を毎サイクル監視し，意図した設計通りに動作していることを確認する必要がある．プロセッサの場合，動かすプログラムによってその動作が変わるため，すべてのプログラムのパターンについて動作が正しいことを確認しなければならず，上述する検証方法を用いることは難しい．

そこでRIDECOREが正しく実装されていると確認するテスト方法として，アプリケーションの実行結果が正しいならば，実装が正しいとすることにした．以下では検証方法を詳細に述べる．

アプリケーションはC言語を用いて作成し，RISC-V向けgcc(ver. 5.2.0)を用いてコンパイルする．なお，テストに用いるアプリケーションは，自作した5つのアプリケーション(fib, matmul, sort_3, hanoi, stencil)，及びC言語による最新アルゴリズム辞典[13]に掲載されている6つのアプリケーション(acker, cprimes, combinat, komachi, stirling, tarai)の入出力部分を，実装するプロセッサ向けに書き換えたものとする．

作成したC言語のアプリケーションを，一般のコンピュータで実行した結果をAとする．次にFPGA上に実装し，アプリケーションを実行することによって得られた結果をBとする．上述した11のアプリケーションすべてにおいて，実行結果AとBが一致した．

この事実をもって，FPGA上に実装したRIDECOREが正しく実装されているとする．

4.3 リソース使用量及び性能

ここでは，RIDECOREを，単体でFPGA上に実装した場合のリソース使用量と，IPC(Instruction Per Cycle)ベースでの性能を述べる．ここで述べるIPCとは，プログラムの実行終了までに実行を完了した命令数を，実行サイクル数で割ったものである．比較対象とするのは，RISC-V命令セットの3段パイプラインを備えたインオーダ実行スカラプロセッサ, vscale[10]のメモリバス(AMBA)

表 4.1: リソース使用量及び動作周波数

	vscale	RIDECORE
Slice LUTs	2402	48488
Slice Registers	1073	14196
BRAM	0	2
DSP	0	16
Frequency[MHz]	136.4	53.46

を単純な回路に置き換えたものとする．論理合成についてはVivado2015.2を使用し，IMEM及びDMEMを取り除いて行った．

vscale及びRIDECOREのリソース使用量及び動作周波数を表4.1に示す．前述した通り，RIDECOREは回路規模が非常に大きく，実装可能なFPGAは限られる．動作周波数については，使用しているFPGAは異なるものの，関連研究で紹介した[3]や[7]と同程度である．

次に各アプリケーションのIPCを比較する．IPCの測定はverilogシミュレータ上で行った．vscale及びRIDECOREのIPCと，相対性能(RIDECOREのIPCとvscaleのIPCの比)を表4.2に示す．表中のAVERAGEについては幾何平均を用いた．また，図4.2に各アプリケーションのIPCをグラフとして示す．

vscaleとRIDECOREの性能を比較すると，RIDECOREのほうが幾何平均で37%程度性能が高く，2命令発行であることや，アウトオブオーダ実行を行うことで高速化を達成できていると言える．

matmul及びkomachiでは2倍以上の差があるが，これは乗算命令の割合が多いことに起因する．vscaleはシフターとアダーを用いて乗算器を実装しており，実行結果を得るまでに32サイクルを要するのに対し，RIDECOREはFPGA上のDSPブロックを用いて乗算を計算することを想定しており，1サイクルで計算が終了するためである．これら2つのアプリケーションを除いて性能比較を行うと，性能向上率は幾何平均で20%程度となる．

各アプリケーションに注目してIPCを比較すると，cprimesやstirlingなど，RIDECOREにて性能が高くない項目がいくつか存在する．これは分岐予測ミスによるものと考えられる．表4.3に各アプリケーションの分岐予測のミス数，ミス率を示す．また，図4.3に各アプリケーションのミス率をグラフとして示す．図表より，IPCが低いアプリケーションは分岐予測ミス率が高いことがわかる．アウトオブオーダ実行プロセッサはパイプライン段数が深いため，予測ミスによるペナルティが大きく，分岐予測ミスが性能に与える影響は大きい．

分岐予測ミスが多い原因として考えられるのは，分岐予測器が正しく実装されていないことである．今回行ったRIDECOREのテストでは，アプリケーショ

表 4.2: 各アプリケーションの IPC

Application	vscale	RIDECORE	relative
fib	0.868	0.995	1.147
matmul	0.428	1.053	2.463
sort_3	0.835	0.880	1.054
hanoi	0.832	1.083	1.300
stencil	0.853	1.123	1.316
cprimes	0.617	0.592	0.959
acker	0.892	1.095	1.228
combinat	0.789	0.903	1.143
komachi	0.338	0.914	2.707
stirling	0.537	0.796	1.497
tarai	0.877	1.070	1.219
AVERAGE	0.683	0.941	1.378

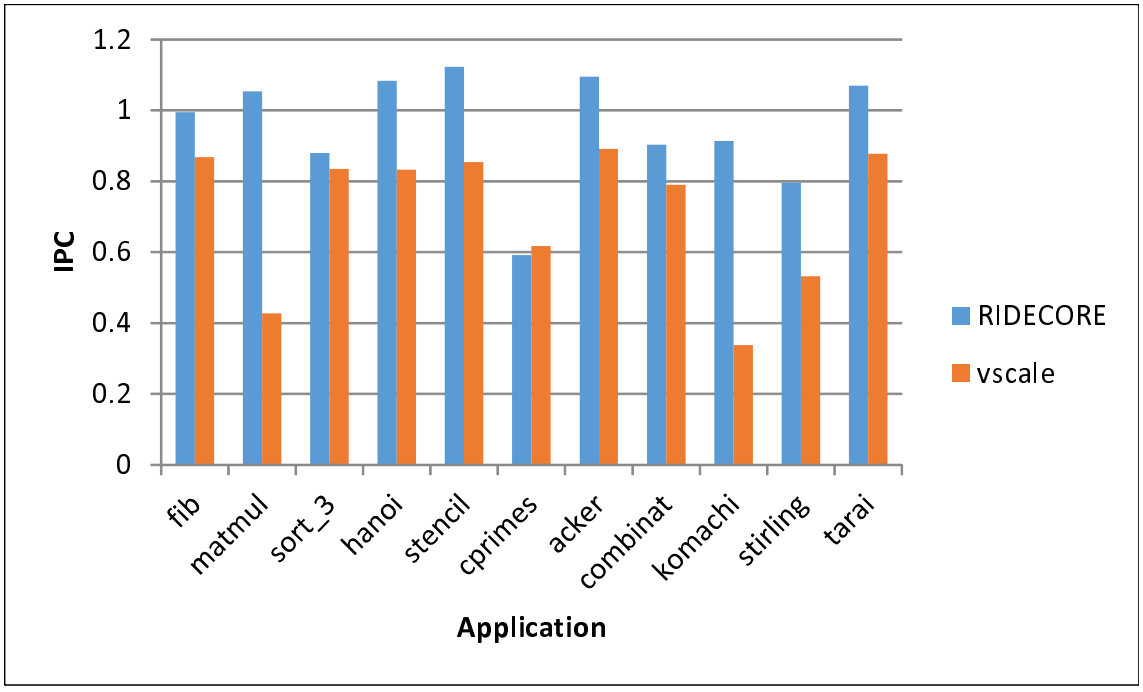


図 4.2: IPC グラフ

ンが正しく実行できることのみ確認している．このテスト方法では，分岐予測

表 4.3: 分岐予測ミス情報

Application	分岐総数	ミス数	ミス率
fib	1035	332	0.321
matmul	374	70	0.187
sort_3	976	299	0.306
hanoi	1599	136	0.085
stencil	620	63	0.102
cprimes	19450	8623	0.443
acker	7273	2445	0.336
combinat	9715	2384	0.245
komachi	296480	144640	0.488
stirling	8133	3487	0.429
tarai	38824	12615	0.325

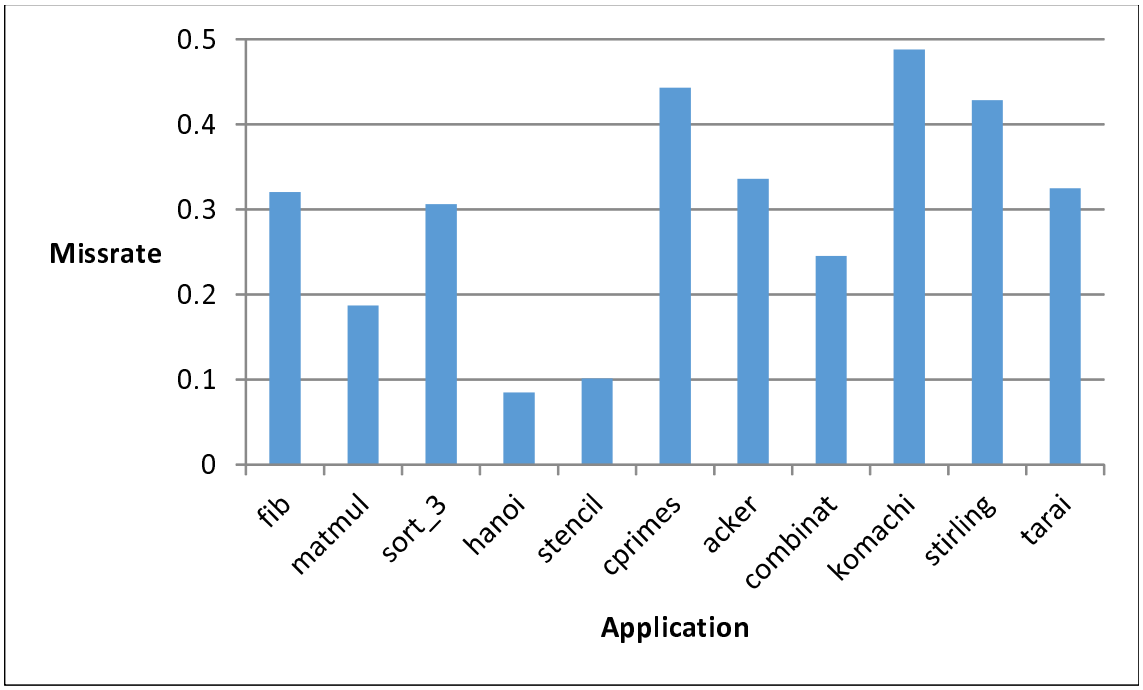


図 4.3: ミス率グラフ

器が誤った予測をしても最終的に実行される命令列には違いがなく，アプリケーションは正しく動作してしまい，分岐予測器のバグを検知できない．

次に原因として考えられるのは，アプリケーションの特性上の問題である．条件分岐の条件式の真偽に関して規則性が低い場合，分岐を予測するのが困難であり，分岐予測ミス率は必然的に高くなる．

4.4 有用性

ここでは，初学者がアウトオブオーダ実行プロセッサの実装方法について学ぶ教材として，RIDECOREが有用であることを示す．

4.4.1 HDL記述

RIDECOREは広く用いられているHDLである, Verilog HDLを用いて記述されているため，ソースコードの理解にかかるコストは低いと言える．ソースコード行数はコメントも含め8558行であり，リファクタリングを行うことで行数を削ることも可能であるため，学習者が一人で把握可能なコード量であるといえる．

4.4.2 ドキュメント

アウトオブオーダの実装方法には多様性があり，公開にあたっては十分なドキュメントが提供されることが望ましい．RIDECOREは*Modern Processor Design*[2]に沿って仕様を策定したため，これを読むことでRIDECOREのアーキテクチャの概要を理解できるようにする．また，仕様策定時に第3章で提示したような図を作成しており，その図に沿ってソースコードを記述した．RIDECOREのドキュメントにそれらの図を掲載し，ソースコードと対比して説明することで理解を容易にする．

ドキュメントの作成例として，IDステージで用いられている投機タグの生成器(Speculative Tag Generator)を用いて提示する．

*****説明例*****

Tag Generatorの回路図を図4.4に示す．また，以下にTag Generatorモジュールの Verilog HDL記述を示す．

```
module tag_generator(
  input wire                      clk,
  input wire                      reset,
  input wire                      branchvalid1,
  input wire                      branchvalid2,
```

```
  input wire                      prmiss,
  input wire                      prsuccess,
  input wire                      enable,
  input wire [`SPECTAG_LEN-1:0]   tagregfix,
  output wire [`SPECTAG_LEN-1:0]  sptag1,
  output wire [`SPECTAG_LEN-1:0]  sptag2,
  output wire                     speculative1,
  output wire                     speculative2,
  output wire                     attachable,
  output reg [`SPECTAG_LEN-1:0]   tagreg
  );

  reg [`BRDEPTH_LEN-1:0]         brdepth;

  assign sptag1 = (branchvalid1) ?
                  {tagreg[`SPECTAG_LEN-2:0], tagreg[`SPECTAG_LEN-1]}
                  : tagreg;
  assign sptag2 = (branchvalid2) ?
                  {sptag1[`SPECTAG_LEN-2:0], sptag1[`SPECTAG_LEN-1]}
                  : sptag1;
  assign speculative1 = (brdepth != 0) ? 1'b1 : 1'b0;
  assign speculative2 = ((brdepth != 0) || branchvalid1) ?
                        1'b1 : 1'b0;
  assign attachable = (brdepth + branchvalid1 + branchvalid2)
                      > (`BRANCH_ENT_NUM + prsuccess) ? 1'b0 : 1'b1;

  always @ (posedge clk) begin
    if (reset) begin
      tagreg <= `SPECTAG_LEN'b1;
      brdepth <= `BRDEPTH_LEN'b0;
    end else begin
      tagreg <= prmiss ? tagregfix :
                ~enable ? tagreg :
                sptag2;
      brdepth <= prmiss ? `BRDEPTH_LEN'b0 :
                 ~enable ? brdepth - prsuccess :
                 brdepth + branchvalid1 + branchvalid2 - prsuccess;
    end
  end

endmodule // tag_generator
```

Tag Generator は，内部にレジスタとして，現在割り当てている投機タグ (tagreg) と現在の分岐の深さ (brdepth) を持つ．2 つの命令 (Insn1, Insn2) がそれぞれ有効な分岐命令であることを示す信号線，branchvalid1 と branchvalid2, 及び命令の発行が可能であることを示す enable を受け取る．それら信号に応じて投機タグ (sptag1, sptag2) 及び自身が投機命令であることを示すフラグ (speculative1, speculative2) を生成し，tagreg, brdepth の更新を行う．

投機タグは，00001 から順番に 00010，00100... というように左ローテートを行いながら割り当てられる．割り当てられるタグがなくなってしまった場合は，

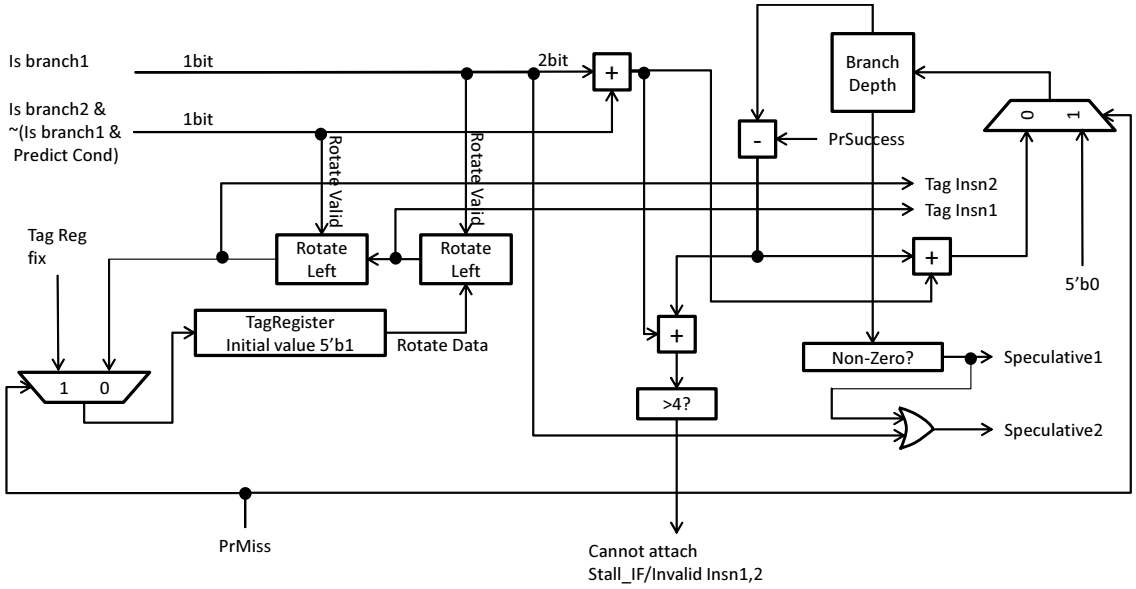


図 4.4: Speculative Tag Generator

タグ割り当て可能フラグ (attachable) に 0 を出力し，ストールを発生させる．

分岐予測に成功した場合 (prsuccess=1)，投機タグの開放を行うため，brdepth をデクリメントする．一方分岐予測に失敗した場合 (prmiss=1)，tagrag 及び brdepth を分岐予測前の状態に戻す必要がある．そのため，tagreg の修復情報，tagregfix を分岐命令実行ユニットから受け取り，修復を行う．brdepth は 0 としているが，分岐命令はインオーダ実行を行うため，分岐予測前の状態に復元した際，その時点での投機命令は必ず 0 個となるためである．

*****説明例終了*****

このようにして各モジュールの説明を記述すると，理解が容易であることがわかる．

4.4.3 詳細動作の理解

RIDECORE は Verilog シミュレータを用いて，各ハードウェアモジュールの詳細動作をサイクル単位で追うことが可能であり，アウトオブオーダ実行の様子を具体的に見ることができる．シミュレーション結果の一部を，GTKwave を用いて表示した様子を図 4.5 に示し，プロセッサ内における，シミュレーションで監視したデータの流れを図 4.6 に示す．また，シミュレーションに用いたプログラムのディスアセンブルを以下に示す．

```
00000000 <_start>:
   0:   00000013                nop
   4:   00000033                add     zero,zero,zero
   8:   000000b3                add     ra,zero,zero
   c:   00000133                add     sp,zero,zero
  10:   000001b3                add     gp,zero,zero
  14:   00000233                add     tp,zero,zero
  18:   000002b3                add     t0,zero,zero
  1c:   00000333                add     t1,zero,zero
  20:   000003b3                add     t2,zero,zero
  24:   00000433                add     s0,zero,zero
  28:   000004b3                add     s1,zero,zero
  2c:   00000533                add     a0,zero,zero
  30:   000005b3                add     a1,zero,zero
  34:   00000633                add     a2,zero,zero
  38:   000006b3                add     a3,zero,zero
  3c:   00000733                add     a4,zero,zero
  40:   000007b3                add     a5,zero,zero
  44:   00000833                add     a6,zero,zero
  48:   000008b3                add     a7,zero,zero
  4c:   00000933                add     s2,zero,zero
  50:   000009b3                add     s3,zero,zero
  54:   00000a33                add     s4,zero,zero
  58:   00000ab3                add     s5,zero,zero
  5c:   00000b33                add     s6,zero,zero
  60:   00000bb3                add     s7,zero,zero
  64:   00000c33                add     s8,zero,zero
  68:   00000cb3                add     s9,zero,zero
  6c:   00000d33                add     s10,zero,zero
  70:   00000db3                add     s11,zero,zero
  74:   00000e33                add     t3,zero,zero
  78:   00000eb3                add     t4,zero,zero
  7c:   00000f33                add     t5,zero,zero
  80:   00000fb3                add     t6,zero,zero
  84:   00002137                lui     sp,0x2
  88:   f0010113                addi    sp,sp,-256
  8c:   1740006f                j       200 <main>
  90:   00000013                nop

00000200 <main>:
 200:   01400093                addi    ra,zero,20
 204:   00a00113                addi    sp,zero,10
 208:   002081b3                add     gp,ra,sp

0000020c <$L1>:
 20c:   00200423                sb      sp,8(zero)
 210:   ffdff06f                j       20c <$L1>
```

まず，プログラムの解説を行う．0-90 番地の命令群はレジスタの初期化を行い，main 関数を呼び出す命令列である．main では ra に 0+20=20(0x14) の値を，sp に 0+10=10(0x0A) の値を代入し，最後に ra+sp(=30, 0x1E) を計算して gp に

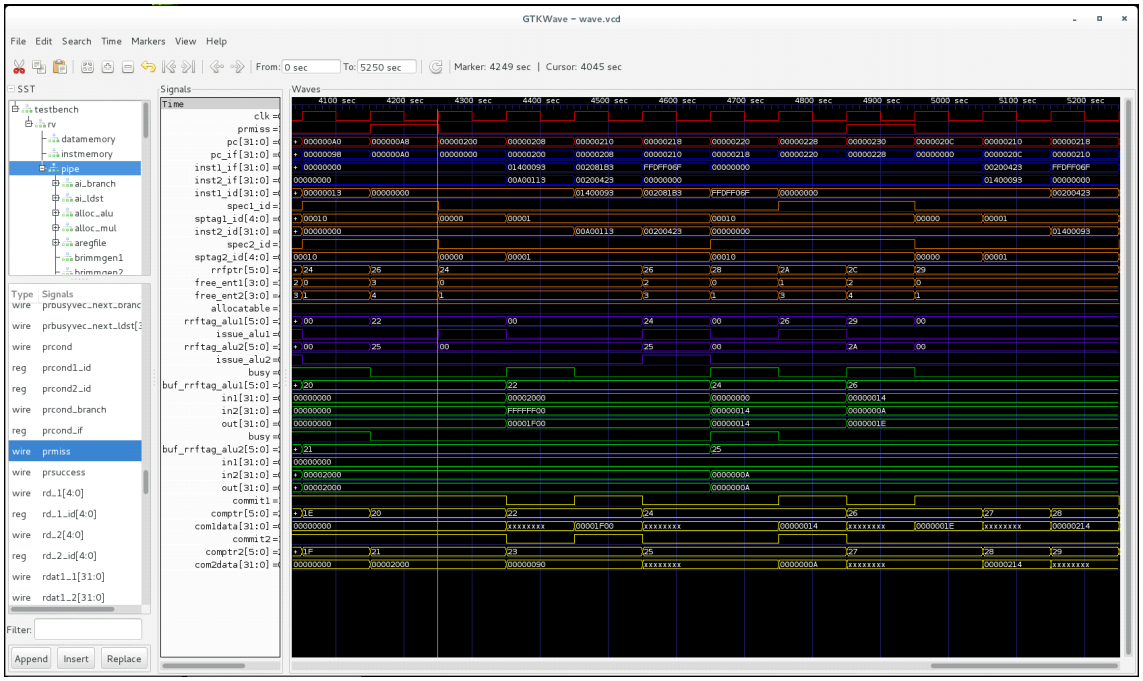


図 4.5: シミュレーション

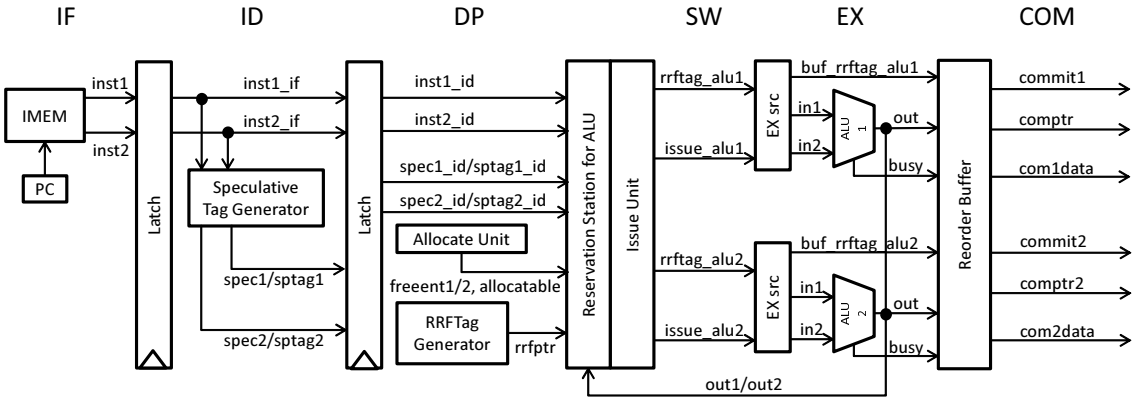


図 4.6: データフロー

代入する．シミュレーションの終了は，メモリの 8 番地に値を書き込んだ時と定義しており，$L1 行では終了処理を行っている．

次にシミュレーション結果について説明する．波形を表 4.4 のように，パイプラインステージで色分けするが，例外として，clk, prmiss は赤で示す．

まず，IF ステージでフェッチした 2 命令 (inst1_if, inst2_if) を ID ステージに送り，ID ステージでデコードした情報が DP ステージに送られる．DP ステージ

表 4.4: 色分け

ステージ	色
IF	赤
ID	青
DP	橙
SW	紫
EX	緑
COM	黄

では, inst1_id, inst2_id それぞれに rrftag(書き込む RRF 及びリオーダバッファのエントリ番号) を割り当てる．割り当てる番号は rrfptr, rrfptr+1 である．次に, Allocate Unit が RS の空きエントリ (free_ent1, 2) を探して，2 命令の実行に必要な値を RS に登録する．

SW ステージでは，計算に必要なオペランドが揃った命令 (rrftag_alu1, 2) を選び，実行ユニット (ALU1, 2) に発行 (issue_alu1, 2) する．発行した命令 (buf_rrftag_alu1, 2) は EX ステージで計算され (計算中は busy が 1 となる)，計算結果 (out) を RS に通知し，リオーダバッファ中の RRF に書き込み，終了通知を行う．COM ステージではリオーダバッファの中から終了した命令を，命令の発行順で 1 つまたは 2 つ (comptr, comptr2) 選び，ARF に実行結果 (com1data, com2data) の書き込みを行い，命令の実行を完了 (commit1, 2) する．

プログラムとシミュレーション結果を並べて比較すると，上述した動作が実現できていることがわかる．例えば，208 番地の加算命令 (rrftag=26) が実行される様子を見る．200 番地及び 204 番地の加算命令 (rrftag=24, 25) の終了後 (EX ステージ参照)，RS に値 (0x14, 0x0A) が通知される．通知された次のサイクルで，RS 内にある rrftag=26 の加算命令のオペランドが揃うため，命令が ALU1 に発行される (SW ステージ参照)．その次のサイクルで，ALU1 で 0x14+0x0A が実行され，実行結果として 0x1E を出力する．

このように，サイクル単位の動作を知りたい場合はシミュレーションを行い，各信号線やレジスタの値を監視することで理解することができる．

4.4.4 有用性についてのまとめ

4.4 節では，以下の 3 つの理由から，大学生と大学院生がアウトオブオーダ実行プロセッサの実装方法を学ぶ教材として，RIDECORE が有用であることを示した．

1. Verilog HDL を用いて 10,000 行以内で記述したためソースコードを理解するコストが低いこと
2. RIDECORE について十分に解説したドキュメントを用意すること
3. Verilog シミュレータを用いることでアウトオブオーダ実行の様子をサイクル単位で追えること

第5章

結論

5.1 まとめ

本論文では，情報工学を専攻する学生向けの教育用アウトオブオーダ実行プロセッサ, RIDECOREを提案し，教育用としての有用性を明らかにした．

RIDECOREは, *Modern Processor Design*[2] に沿って仕様を策定し，これを読むことでRIDECOREのアーキテクチャの概要を理解できる．また，各ハードウェアモジュールの役割やその実装について説明したドキュメントを用意することで，理解を容易にする．RIDECOREは広く用いられているVerilog HDLを用いて記述し，コード行数を10,000行以内に抑えることで実装方法の理解を容易にする．加えて，コンパイラ，アセンブラなどの開発環境が簡単に用意できること，プロセッサのソースコードを公開できることから，命令セットとして自由に利用可能なRISC-Vを採用した．

5.2 今後の課題

今後の課題として，以下が考えられる．

- オープンソースとして公開するための，ソースコードのリファクタリングやRIDECOREに関するドキュメントを完成させること
- 分岐予測器を含め，内在する可能性のあるバグを取り除くこと
- 回路規模を減らし，より回路規模の小さなFPGAへの実装を可能にすること
- 除算，浮動小数点演算などの命令をサポートすること
- プロセッサの性能を向上させること
- 例外や割り込みに対応するためのハードウェア機構を実装し，OSの動作を保証すること

謝辞

本研究をすすめるにあたり，適切なご指導を賜りました指導教員の吉瀬謙二先生に深く感謝いたします.

本研究で提案したプロセッサの仕様策定及び設計の際には大いに助けて頂き，本論文や全国大会の原稿の添削も熱心に行っていただいた眞下達先輩に深く感謝いたします.

修士論文で忙しい中，様々な知識を教えて頂いたり，論文の構成に関してアドバイスをして頂いた森悠先輩に感謝いたします．また，卒業論文のテーマを決定する際に様々な選択肢を提示して頂いたり，プロセッサをFPGAで動作させる際に助けて頂いた小林諒平先輩に感謝いたします.

さらに，種々の相談にのって頂いた松田裕貴先輩，ゼミの際に多くのアドバイスを頂いたThiem Van Chu先輩，臼井琢真先輩，便利なソフトウェアについて教えて頂いた奥村開里先輩，隣席で研究室を賑やかな雰囲気にしてくださった小川愛理先輩，味曽野智礼先輩に感謝いたします.

最後に，いろいろなことを教え合ったり他愛のない雑談にのったりしてくれた，同期のPaniti Achararit君，川井博斗君，大谷伸吾君に感謝いたします.

参考文献

[1] RISC-V, http://riscv.org/ (2016/01/20)

[2] John Paul Shen, Mikko H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, 2013.

[3] Christopher Celio, David A. Patterson and Krste Asanovi: The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor, EECS Department University of California, Berkeley Technical Report (Jun. 2015)

[4] ZC706, http://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html (2016/02/01)

[5] SHAKTI, http://rise.cse.iitm.ac.in/shakti.html (2016/01/20)

[6] ORCONF2015, http://openrisc.io/orconf/ (2016/01/20)

[7] 杉本 健，入江 英嗣，五島 正裕，坂井 修一：Out-of-Order スーパスカラプロセッサの FPGA への実装，先進的計算基盤システムシンポジウム SACSIS, pp. 196-197 (2007).

[8] McFarling, Scott: Combining branch predictors, Technical Report TN-36, Digital Western Research Laboratory, (1993)

[9] 吉瀬謙二，片桐孝洋，本多弘樹，弓場敏嗣：高性能プロセッサのための代表的な分岐予測器の実装と評価，Technical Report UEC-IS-2003-2, 電気通信大学大学院情報システム学研究，(2003)

[10] vscale, https://github.com/ucb-bar/vscale (2016/01/07).

[11] Buyuktosunoglu A., El-Moursy A., and Albonesi D. H.: An oldest-first selection logic implementation for non-compacting issue queues [microprocessor power reduction]. In 15th Annual IEEE International ASIC/SOC Conference (pp. 31-35). (2002).

[12] VC707, http://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html (2016/02/01)

[13] 奥村晴彦：C 言語による最新アルゴリズム辞典，事典技術評論社，1991

発表文献

1. 藤浪　将，眞下　達，吉瀬　謙二：　Verilog HDLで記述するRISC-V命令セットのアウトオブオーダ実行プロセッサ，第78回IPSJ全国大会，(2016/03/12予定)