卒業論文

題目

要求仕様に柔軟に対応できる浮動小数点演算器の設計手法に関する研究

指導教員

佐々木 敬泰 助教

2018年

三重大学 工学部 情報工学科
コンピュータアーキテクチャ研究室

米藤 智哉 (414860)

内容梗概

近年，高性能及び低消費電力を両立する手法として，異なる複数のプロセッサコアによって構成されるヘテロジニアスマルチコアプロセッサが注目されている．従来のホモジニアスマルチコアプロセッサでは，構成の同じコアを複数搭載するため，各アプリケーションに応じて全て同じリソースを割りあてるため，性能の過不足により，無駄な消費電力が発生する．ヘテロジニアスマルチコアプロセッサでは，各アプリケーションに応じて適切なコアを割りあてることにより，無駄な消費電力を抑えることが可能である．しかし，ヘテロジニアスマルチコアプロセッサでは，異なるプロセッサコア，キャッシュ，それぞれに応じたバスシステムが必要なことから，設計・検証コストがホモジニアスマルチコアプロセッサと比較して高くなり，研究・開発する上での障害となっている．この問題を解決するため，様々な構成のヘテロジニアスマルチコアプロセッサを自動設計するツールとして，FabHetero が提案されている．FabHetero では，設計者が並列度やキャッシュサイズのような要求仕様をパラメータとして与えることで，ヘテロジニアスマルチコアプロセッサを自動設計することを目指している．しかし，現状の FabHetero では，浮動小数点演算器（FPU: Floating Point Unit）の自動設計機能がない．これは，FabHetero を記述するハードウェア記述言語の System Verilog の機能である DPI-C によって大半を記述されていることに起因する．この問題は，論理合成可能な記述で FPU を実装することで解決できるが，単純な構成の FPU を実装するだけでは不十分である．なぜなら，浮動小数点演算の複雑さから，単純な FPU の構成では，パラメータの変更に対応して，最適なハードウェア規模や処理能力を提供できないためである．また，FPU の構成最適化は，設計者の再記述を要求するため，設計・検証コストの増加となってしまう．

そこで，本研究では FabHetero 上に可変設計可能な FPU を実装することで，より多様なコア構成の実現を目指す．本研究では，将来の FabHetero 上での FPU 設計機能を実装するため，可変段数パイプラインの作成，要求仕様に対するパイプライン構成の最適化支援のためにベンチマークから演算ごとの依存度の調査を行う．

Abstract

Heterogeneous multi-core processors, which are composed of processor cores different from each other, have attracted attention as a technique to achieve both high performance and low power consumption. In the conventional homogeneous multi-core processors, which are composed of the same processor cores, the same resources are assigned to each application. However, wasteful power consumption occurs due to excess or deficiency of performance. In the heterogeneous multi-core processor, it is possible to suppress unnecessary power consumption by allocating appropriate cores according to each application. But, heterogeneous multicore processors have different processor cores, caches, and corresponding bus systems, so design and verification efforts are higher than that of homogeneous multicore processors, which is an obstacle to research and development. In order to solve this problem, FabHetero has been proposed as a tool for automatically designing heterogeneous multi-core processors of various configurations. FabHetero aims to automatically design heterogeneous multi-core processors by giving required specifications such as parallel degree and cache size as parameters. However, the current FabHetero does not have the automatic design function of a floating point unit (FPU) because current FabHetero use DPI-C which is a function of System Verilog for simulation to implement an FPU. Although this problem can be solved by implementing FPU with a description that can be synthesized by logic, it is insufficient to simply implement FPU with a simple configuration. This is because, due to the complexity of floating-point arithmetic, in the configuration of a simple FPU, it is not possible to provide an optimum hardware scale or processing capacity in response to a change in parameters. Also, since FPU configuration optimization requires redescription by designers, design and verification costs increase. In this research, we aim to realize more diverse core configurations by implementing variable designable FPU on FabHetero. In order to implement FPU design function on FabHetero in the future, we will create a variable stage pipeline and investigate the dependency of each operation from the benchmark for optimizing the pipeline configuration to the

required specifications.

目 次

図 目 次

表 目 次

1　はじめに

近年，高性能と低消費電力を両立するプロセッサとして，構成の異なる複数のコアで構成されたヘテロジニアスマルチコアプロセッサが注目されている．しかし，異なる構成のコアを複数持つことから，各コア毎に個別の設計・検証が必要となるため，設計・検証コストが増加する問題がある．これを解決するため，当研究室では，ヘテロジニアスマルチコアプロセッサを自動設計するツールとして，FabHetero を提案している．これは，各コア，キャッシュ，バスシステムを自動設計することで，ヘテロジニアスマルチコアプロセッサの設計・検証コストを目指すプロジェクトである．しかし，現在の FabHetero では浮動小数点演算器（FPU: Floating Point Unit）は機能の模倣に留まり，演算部の実装には至っていないため FPU の生成は不可能である．近年では，流体シミュレーションや複雑な画像処理といった分野だけでなく，組込み機器やモバイル端末用ゲーム等，様々な分野で浮動小数点演算が必要とされている．そのため，FPU の自動設計が強く求められている．しかしながら，FPU の自動設計を行うためには，パラメタライズされた FPU 設計に加え，演算リソースやパイプライン構成に関するパラメータを自動的に最適化するツールが不可欠である．そこで，本研究ではプロセッサ設計・実装時に開発者が指定

する必要のある各種パラメータをベンチマークプログラムの実行結果や演算器のハードウェア量，要求仕様などから自動的に最適化することを目指す.

2　マルチコアプロセッサと命令処理

提案手法を説明するのに先立ち，当研究で扱うマルチコアプロセッサの概要とコア演算の高速化手法およびFPUの説明を行う．

2.1　マルチコアプロセッサ

プロセッサコアとは、CPUの演算処理を担う部分の事である．マルチコアプロセッサとは，1つのCPUパッケージの中にプロセッサコアが複数存在するものを指す．現在，スマートフォンやゲーム機などの組み込み機器でのデジタルコンテンツの普及から，増加するアプリケーションの処理量に対応するプロセッサの性能向上が求められる．しかし，クロック周波数を上げることによるプロセッサの性能向上は，消費電力や発熱量の増大などの問題があるため困難になっている．複数のコアを持つマルチコアプロセッサでは，チップサイズの増大と引き換えに，複数の処理を分散して並列実行することで，以上の問題を回避しつつ性能を向上させることが可能である．

2.1.1　ホモジニアスマルチコアプロセッサ

ホモジニアスマルチコアプロセッサとは図2.1左のような，同じ構成のコアを複数搭載するマルチコアプロセッサである．各コアは同じ構成で

あるため，あるコアの設計・検証結果を流用することで設計・検証コストを削減することが可能である．各コアが独立して動作し，複数のアプリケーションを同時に実行することで，高い処理性能を持つ．しかし，各コアの構成が同じであるため，アプリケーションごとの特性によって要求される性能は異なることから，処理性能不足による処理時間の増加や過剰な処理性能による消費電力の増加という問題がある．

2.1.2　ヘテロジニアスマルチコアプロセッサ

ホモジニアスマルチコアプロセッサとは図 2.1 右のような，異なる構成のコアを複数搭載するマルチコアプロセッサである．各コアの構成が異なるため，アプリケーションごとの特性に応じた適切なコア割り当てを行い，高性能及び低消費電力を両立することが可能である．しかし，各コアの構成が異なることから，各コアごとに個別の設計・検証が必要となるため，ホモジニアスマルチコアプロセッサと比べ，設計・検証コストが増加する問題がある．

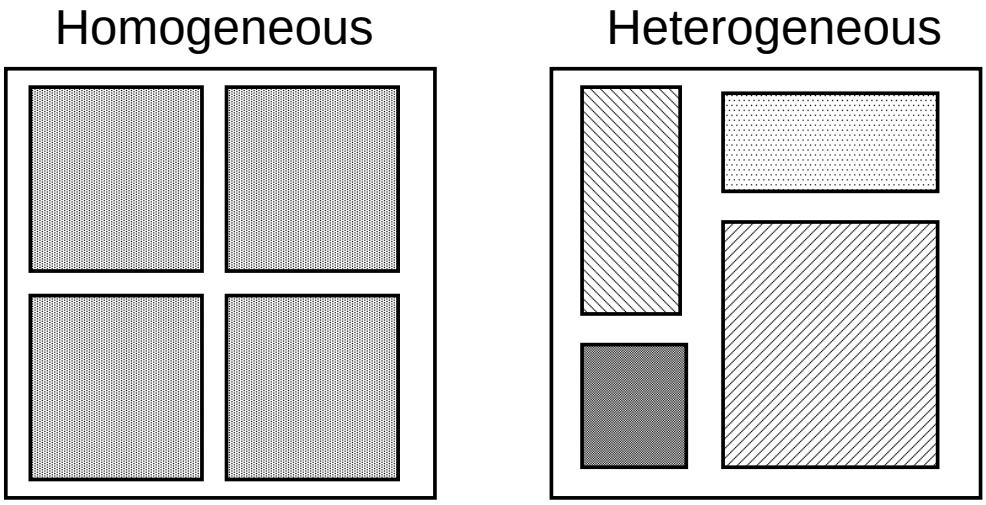

図 2.1: ホモジニアスマルチコアとヘテロジニアスマルチコア

2.2 スーパスカラと命令パイプライン

スーパスカラ及び命令パイプラインとは，図 2.2 のように命令レベルの並列性によって高速化する手法である．通常のプロセッサではプログラムを構成する命令列から一度に一命令ずつ読み込んで順番に処理を行う．スーパスカラプロセッサでは命令を解釈及び実行する回路を複数備えている．これにより一度に複数の命令を読み込み，同時実行することで高速に処理することが可能である．パイプラインとは，命令の処理を独立して実行できるステージに分割し，相互接続された各ステージを並列化することで，単位時間あたりに実行できる命令数を増やす手法である．コアの制御装置では，最も処理時間のかかるステージを基準として，

クロック信号に同期して命令を実行ユニットに送り出すことで全体の処理を高速化している．しかし，この 2 つの手法は命令列の依存により性能の制約を受ける．命令列の依存とは，例として以下のようなプログラムを挙げる．

1. $\$1 = \$2 + \$3$
2. $\$4 = \$5 \times \$6$
3. $\$1 = \$1 + \$4$

上記では，命令 1 と命令 2 は同時実行可能である．しかし，命令 3 は命令 1 と命令 2 の演算結果を使うため，前の命令が演算を終えるまで実行することができない．そのため，このような命令の依存が連続で起こるようなプログラムでは，パイプラインの使用率の低下を起こすため，プログラムによって，全体の処理性能向上はハードウェア規模増大に対して悪くなる．

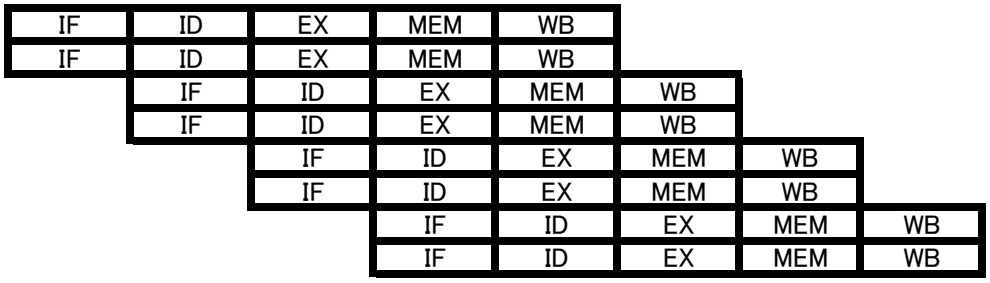

図 2.2: スーパスカラなパイプラインの例

2.3 FPU

FPU とは，浮動小数点演算を専門に行う処理装置である．数値データの表現形式の一つである浮動小数点数の計算に特化しており，表現できる数値の範囲が広く，幅広い研究分野で扱われる．浮動小数点数処理の一部または全部が図 2.3 のような，IEEE 754 という規格に統一されている．IEEE 754 での浮動小数点演算では，符号部 (S)，指数部 (E)，仮数部 (F) と 3 つで構成される．加減算を例に挙げると，符号部の判定による加算・減算の変更，指数部による桁合わせ，仮数部の計算のように最上位ビット側からの順番の処理になる．通常の整数演算命令のみを使って浮動小数点演算を行うと、多くの命令と処理時間が必要になる．そのため，整数演算器とは別にコアに内蔵されているか，コプロセッサとして搭載されている．当研究では，後の章で述べる FabHetero への FPU の実装を

目的とするとともに，要求仕様に対応した構成変更機能と構成最適化を提案する．

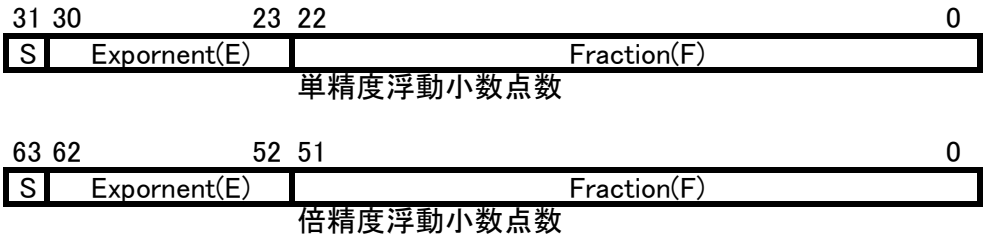

図 2.3: IEEE 754 規格

3　FabHeteroプロジェクト

FabHeteroとは，当研究室で提案されているヘテロジニアスマルチコアプロセッサを自動設計するツールである．ヘテロジニアスマルチコアプロセッサの設計・検証には多大なコストが掛かる．FabHeteroは，それらのコストを低減するため，パラメータを指定することでヘテロジニアスマルチコアプロセッサを自動設計することを目的とする．設計者は要求仕様をパラメータとして指定することで，様々な構成のヘテロジニアスマルチコアプロセッサを自動設計し，設計及び検証コストを低減することができる．FabHeteroによって設計されるヘテロジニアスマルチコアプロセッサの例を図3.4に示す．FabHeteroは，図3.4のようにコアを設計するFabScalar[2]，キャッシュを設計するFabCache[3]，バスを設計するFabBus[4]の3つの構成で成り立つツールである．しかし，現在のFabHeteroではFPUの自動設計機能が実装されていない．ヘテロジニアスマルチコアプロセッサでは異なる構成のコアを持つため，FPUもそれぞれの設計データを持つ．しかし，浮動小数点演算の複雑さから，各コアでのFPUの設計・検証は大きなコストとなる．後の章では，FPU自動設計機能の実装対象であるFabScalarの概要及び実装に伴う問題点を挙げる．

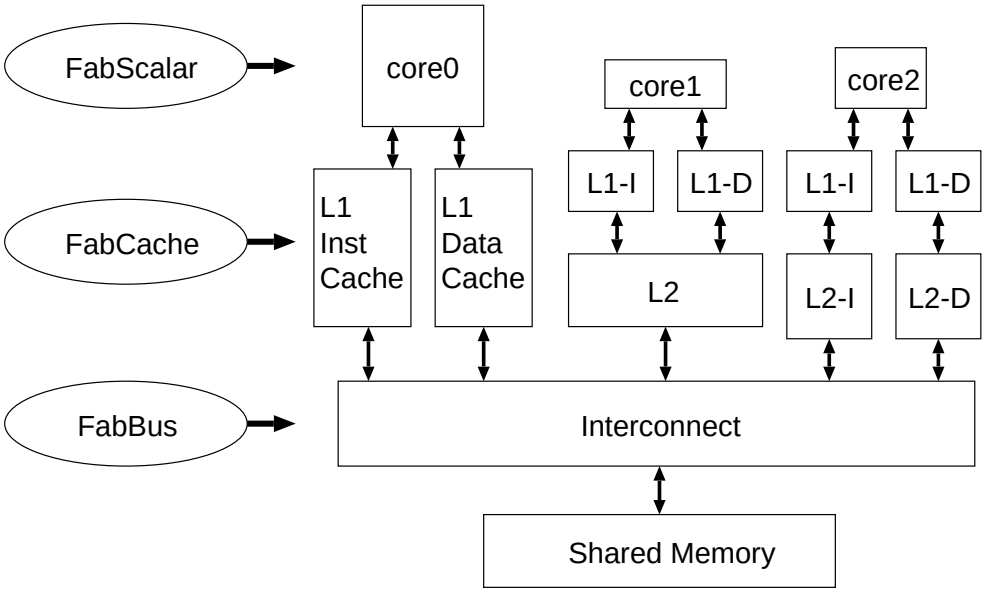

図 3.4: FabHetero

3.1 FabScalar

ノースカロライナ州立大学により，スーパースカラプロセッサ自動設計ツール FabScalar が提案されている．また，FabScalar の FPU に対応したパイプラインの構成 [5] は A. V. Shastri らによって提案されている．これは，図 3.5 のような FPU を持つパイプラインステージの生成を目的とする．浮動小数点命令は上流 4 ステージを整数命令と共有し，以降のステージでは，整数命令は左の整数演算器 (IU: Integer Unit) へ，浮動小数点命令は右の FPU へ分類される．Reg-Read ステージから WriteBack ステージは，命令を演算処理する実行レーンである．IU，FPU の共通の実行レーンは、整数及び浮動小数点のロードとストアの処理を行い，整数と浮動小数点の間の相互変換もサポートする．FabScalar では様々なパラメータ，例えばフェッチ幅や発行幅，コミット幅，発行キューのエントリ数，リオーダバッファのエントリ数などを指定することで異なる構成のスーパスカラコアを自動設計し，設計及び検証にかかる時間を大きく短縮することを可能とする．

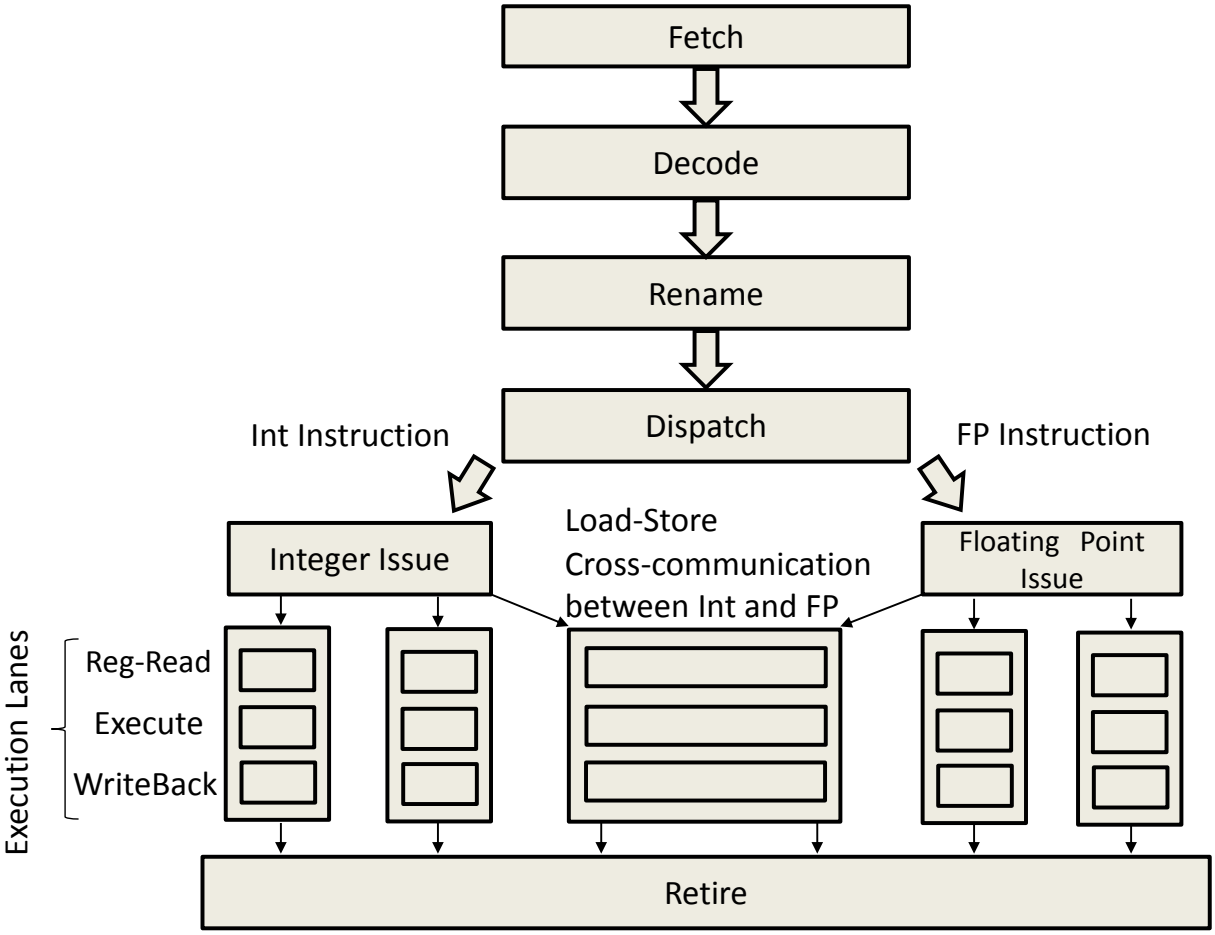

図 3.5: FabHetero

3.2 FabScalar における FPU の問題点

FPU で扱われる浮動小数点演算は一般的にレイテンシの長い命令演算である．そのため，浮動小数点演算ではパイプラインステージが多いほど性能向上が期待できる場合が多い．しかし，FabScalar の FPU は FabHetero を記述するハードウェア記述言語の System Verilog の機能である DPI-C（DPI-C: Direct Programming Interface - C）によって大半を記述されている．DPI-C とは，System Verilog と C 言語で記述した機能モデルとの接続を行うインターフェースである．そのため，現状の FabScalar では，機能シミュレーションのみが提供され，ハードウェアの記述はなされていないため，論理合成による FPU の自動設計は不可能である．また，FPU で扱われる各演算リソース数や実行ステージの段数はパラメータによって変更することができない．浮動小数点演算は整数演算と比較してハードウェア記述言語による記述量が多い．ヘテロジニアスマルチコアプロセッサのような異なるコア構成を持つ場合，それぞれで異なる記述が必要なため設計・検証に非常に時間が掛かる．よって，FabScalar に FPU の自動設計する機能を追加する必要がある．しかし，単純な設計で行ったパラメータ指定のできない固定的な FPU の構成では問題が生じる．また，現在の FabScalar で提案される FPU は，全ての浮動小数点命令は同

一種のパイプラインで処理される．そのため，全ての浮動小数点命令は最も処理時間の掛かる浮動小数点命令によって性能が低下する．よって，命令ごとにパイプラインを分けることで命令ごとの性能を引き出す必要がある．しかし，命令ごとにパイプラインを専門化すると問題が生じる．

3.2.1　パイプラインの段数の問題

パイプラインの段数とは，パイプラインのステージ数を指す．パイプラインを命令ごとに分割した場合，クロック周波数の変更に応じて各命令パイプラインの再構成が必要となる．パイプライン段数が多ければ，高クロック周波数に対応したコアにすることが可能であるが，高クロック周波数化は消費電力の増大に繋がる他，ステージごとに挟まれるパイプラインレジスタによるハードウェア規模・消費電力の増加が伴う．固定的な設計では，低面積や省電力といった要求仕様，高クロック周波数化に対して対応することができないため，パラメータごとにパイプライン段数を変更する必要がある．

3.2.2　パイプラインの本数の問題

例として図 3.6 で示される FPU の構成を挙げる．図では，命令を一時的に溜めておく場所 (Reservation Station) を作り，命令順序を入れ替え

て実行するアウト・オブ・オーダー実行を想定している．この構成では3命令までが同時に発行可能であるとしており，加減算器2つと乗除算器1つを持っている．発行とは，Reservation Station 中の命令を適当な実行ユニットに対して割り当てることを指す．ここで，加算命令1つと乗算命令2つを発行しようと考える．しかし，この3命令では，加算命令と乗算命令1つずつならば発行可能であるが，下段の乗算器の有無によって命令間の依存に関わらず2つ目の乗算命令は発行が行えないため，命令の発行を遅らせる必要がある．このように，各命令の演算器の有無が処理性能に影響を与えることがある．しかし，並列実行の可能性を上げるためにパイプラインの本数を増やした場合，その分だけレジスタに接続するポート数は増加する．レジスタファイルはSRAMで構成されており，1命令あたり，2つのリードポートと1つのライトポートを必要とする．SRAMの回路面積は，ポート数の2乗に比例するため，レジスタファイルの回路面積は非常に大きいものとなる．これは，製造コストや消費電力，アクセス時間の増大といった問題を引き起こす．また，第2.2節の命令列の依存によるパイプライン構成の問題のように命令列に依存が連続して発生すると，パイプラインの使用率が低下するため，ハードウェア規模増加に見合う性能が出せないことがある．

そこで，第 3.2.1 節と第 3.2.2 節で挙げる問題を解決し，ハードウェア量を抑えつつ高性能を実現するためには，1) 命令パイプラインごとの段数変更によるクロック周波数変更への対応，2) 同時に使用する可能性の低い演算器，かつパイプライン段数が同程度の演算器をまとめ，レジスタファイルの書き込みポート数を削減が必要がある．

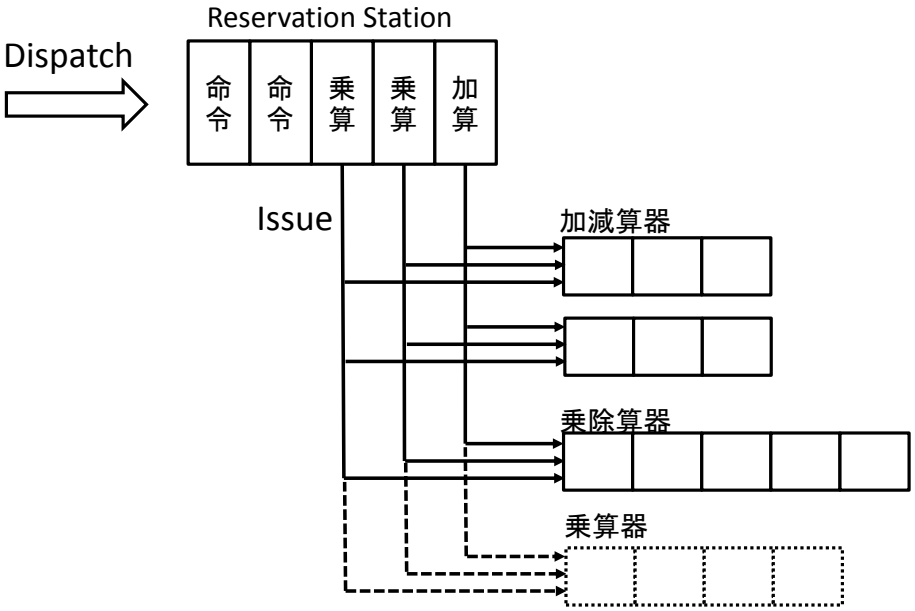

図 3.6: パイプライン構成の例

4　提案手法

本章では，第 3.2 節で述べた 2 つの問題点に対応する手法として，クロック周波数に応じた可変段数パイプラインの提案，パイプラインの統合及び分割によるパイプライン構成自動最適化のための命令依存の調査について述べる．

4.1　可変段数パイプライン

第 3.2.1 節に述べた問題に対応するため可変段数パイプラインを提案する．可変段数パイプラインは，各ステージ間に存在するパイプラインレジスタの有無によって段数を変更する．System Verilog は設計データ作成する際，モジュールという単位で構成を変更することができる．各演算器内で処理を行うパイプラインステージとステージ間のパイプラインレジスタをモジュールとして別途定義して，パラメータごとに下位モジュールとして生成することが可能である．例として図 4.7 に加減算器の例を示す．図 4.7 では，桁合わせを行う align ステージ，実際に加算を行う calc ステージ, 丸めや例外処理を行う norm ステージの 3 ステージを基準として，各ステージ間のパイプラインレジスタの生成の有無をパラメータに応じて変更することで 1 から 3 ステージの加減算器の設計を可能とする．

図 4.7 では，align ステージ，calc ステージ間のパイプラインレジスタを生成しないことで 2 段のパイプラインとなっている．パラメータによるクロック周波数の変更に応じて，設計時に動的にパイプラインの段数を変更可能とすることで，各パイプラインの再記述が不要なため設計コストが減少する．

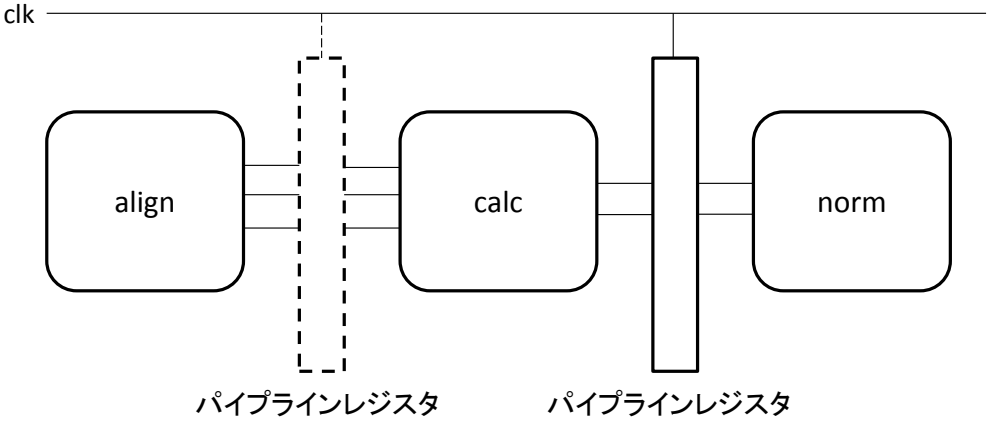

図 4.7: FADD パイプライン

4.2 パイプラインの統合及び分割のための命令依存の調査

第 3.2.2 節に述べた問題に対応するため，設計時に浮動小数点演算パイプラインの統合及び分割を行う．しかし，パイプラインの統合及び分割はプログラムから生成されるアセンブリコードによって最適な構成に差が生じる．例えば，画像処理では，ある領域ごとに同じ命令を連続して実行するような命令が多いため，同一の命令の実行ユニットを複数搭載す

るか，命令パイプラインの専門化などで高速化できる．ただし，性能に見合うハードウェア規模の増大であるかは，実際に命令が並列に実行されているかによって変わるため，どれだけ依存状態で実行されてない命令があるか依存度を調べる必要がある．ヘテロジニアスマルチコアプロセッサでは，コアごとに扱うプログラムの分野を専門化することが多いため，コアごとにパイプラインの構成を変える必要があり，扱うプログラムごとに適した構成を検討するには時間的コストが掛かる．そのため，コアで扱うプログラムの分野ごとにベンチマークから構成の指標となる命令の依存関係を調べることで，パイプライン構成の自動最適化を図る．

命令の依存度は，ベンチマークの実行履歴を記録したトレースデータとベンチマークの逆アセンブルテキストから計測する．命令の依存度とは，各命令ごとに後続の命令で依存した命令数を命令別にカウントしたものを指す．命令は単精度，倍精度を含めた四則演算，浮動小数点レジスタを使うロードストア命令を対象とし，各命令が結果を出すまでのサイクルを指定したとき，どの命令がどれだけの回数だけ依存状態であったかをカウントして，依存度によって今後のパイプラインの統合及び分割の指標とする．例として，図 4.8 を挙げる．1 番目の乗算命令 mul.d の処理に必要なサイクル中に実行される以降の命令から依存関係を検出す

る．以降の命令 add.d 及び mul.d ではともに$f2 の演算結果待ちであるため依存が存在する．また，3 番目の命令では，ディスティネーションレジスタが 1 番目の命令と同じであるため，以降の命令では 1 番目の命令と依存関係を持つことはないので当該命令との依存検出をする必要はない．

以下に依存検出の擬似コードを示す．

```
fill the Count array with 0
fill the InstSeq array with assembly instruction sequence
fill the ReqCycle array with required cycle of each
instruction
while length(InstSeq) do
  for i in range(1,ReqCycle[InstSeq[0]])
    if OpDistination(InstSeq[0]) == OpSource(InstSeq[i]) or
    OpDistination(InstSeq[0]) == OpTarget(InstSeq[i]) then
      Count[InstSeq[0]][InstSeq[i]] += 1
    end if
    if OpDistination(InstSeq[0]) ==
    OpDistination(InstSeq[i]) then
      exit
    end if
  end for
  pop(InstSeq)
end while
```

命令間で依存度が高いほど演算結果待ちの状態が続くため，その命令のパイプラインが使用されない時間が増えることでハードウェア規模の増加に対する性能向上が悪くなる．そのため，依存度の高い命令同士を統合することで性能低下を抑えつつ，ハードウェア規模を縮小する．

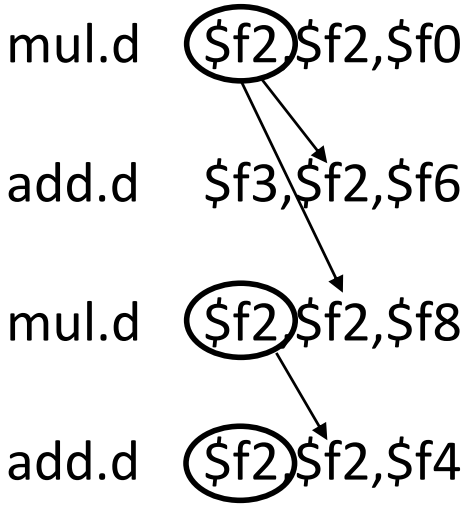

図 4.8: 命令間の依存

5　評価と考察

今回，ベンチマークの一種であるSPLASH2[7]の高速フーリエ変換（FFT: Fast Fourier Transform），RADIXソート及び姫野ベンチマーク[6]のプログラムから浮動小数点演算同士の依存関係を調査した．表5.2では，各命令の演算に想定した必要なサイクル数を示す．各ベンチマークの依存度の図では，各命令が命令パイプライン上にある時，どれだけ後続の命令が依存状態であったかを示す．例として，図5.9の一番左のFADDでは，加減算命令が命令パイプラインで処理されている際，3サイクル中の後続の命令で加減算命令は19481回，乗算命令は2283回，ストア命令は14336回依存状態にあり，処理できない状態であったことを示す．図5.9より，FFTの傾向として，加減算と乗算が依存しやすく，除算は他の命令に依存しにくいことが分かる．実際に実行されるアセンブリコードより，命令自体の実行回数を理由に，加減算命令，乗算命令は実行回数の多い命令と依存しやすく，除算命令では使用するレジスタをほぼ専有状態にして依存度を下げていることなどが分かる．また，表5.2と図5.10に見られる除算命令が無く，依存関係がそもそも存在しえない場合や，図5.11の加減算のように，命令は存在するが，依存関係がない場合もある．今回のデータより，重要項目を以下に挙げ，パイプラインの構成を考える．

- 命令の実行回数
- 命令間の依存度
- 命令実行に必要なパイプライン段数

例えば，FFT の例では，レジスタの書き込みポートが 4 本の場合は，加減算，乗算，除算，ロードをそれぞれ個別のパイプラインに，3 本の場合は加減算と乗算は依存しやすいため，この 2 つを統合して，加減乗算，除算，ロード，2 本の場合は実行回数の少ない除算とロードを統合して，加減乗算，ロード除算，1 本の場合はすべてを統合すればよいことがわかる．このように，命令の依存関係には偏りがあるため，依存の多い命令，つまり同時に発生することが多い命令用のパイプラインは統合しても性能に影響が少ないと考えられる．各コアに対して処理させる分野に応じたベンチマークの命令依存度を与えることで適したパイプラインの構成が可能である．従って，各演算器の取り得るパイプライン段数，各命令の依存度，利用可能な物理レジスタのポート数を与えることで最適な構成を求められることがわかった．現状，構成の定式化には至っていないが，最適化の手順案として以下を挙げる．

1. 動作クロック周波数から各ステージが動作可能な最も細分化された

各命令パイプラインを基準とする．

2. 許容可能なレジスタサイズ，ポート数を元に，パイプラインの本数を決定する．

3. 実行回数が多く，他命令に依存しない命令は優先して，単一命令パイプラインとする．

4. 依存関係の強い命令同士は，パイプライン段数の近いもの同士から統合を行う．

5. 実行回数の少ない命令は，パイプライン段数の近いものと統合を行う．

依存度による構成の最適化は，パイプライン統合・分割によって性能の変化がどれだけ大きくなるかが考慮されていないため，統合・分割の優先度は決定出来ない．例えば，実行回数の少ない除算命令を乗算命令と同じパイプラインとする場合，除算パイプラインと乗算パイプラインでは大きなステージ数の差がある．乗除算パイプラインはボトルネックとなる除算パイプラインと同じ段数となるため，乗算命令の必要サイクル数が増えるためステージ数の差だけ性能が低下する．このように単純に依存があるため統合を行う場合，命令の性能が大きく下がる場合が考え

られる．そのため，定式化を行うにあたって，依存度，実行回数の他に段数変化に伴う性能変化から重み付けを行う必要がある．

	加減算	乗算	除算	ロード
必要サイクル数	3	5	24	7

表 5.1: 各演算の必要サイクル

	加減算	乗算	除算	ロード	ストア
FFT	70159	60175	2114	82546	46544
RADIX ソート	6029322	12058637	0	35	34
姫野ベンチマーク	4074878	2522527	262	38	35

表 5.2: 各演算の実行回数

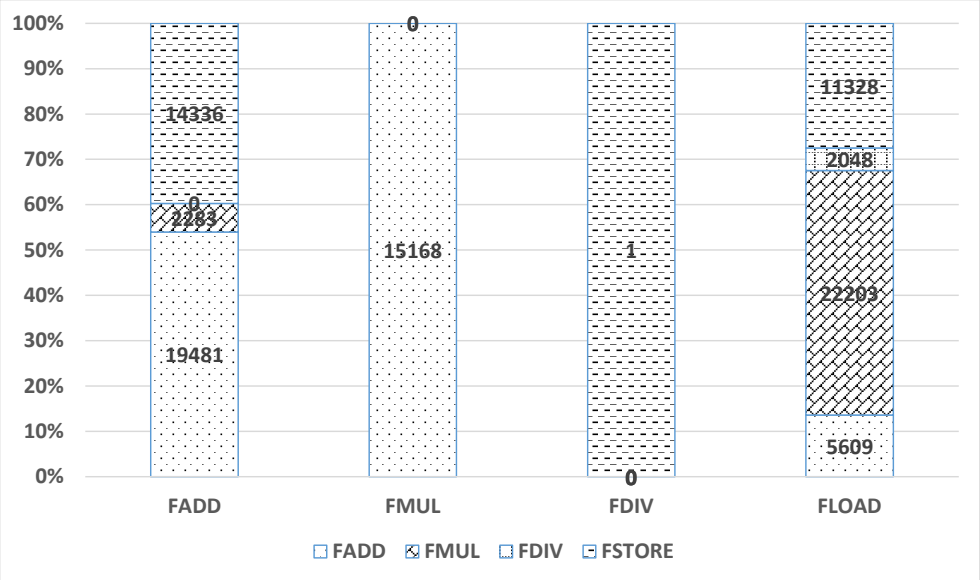

図 5.9: FFT における依存度

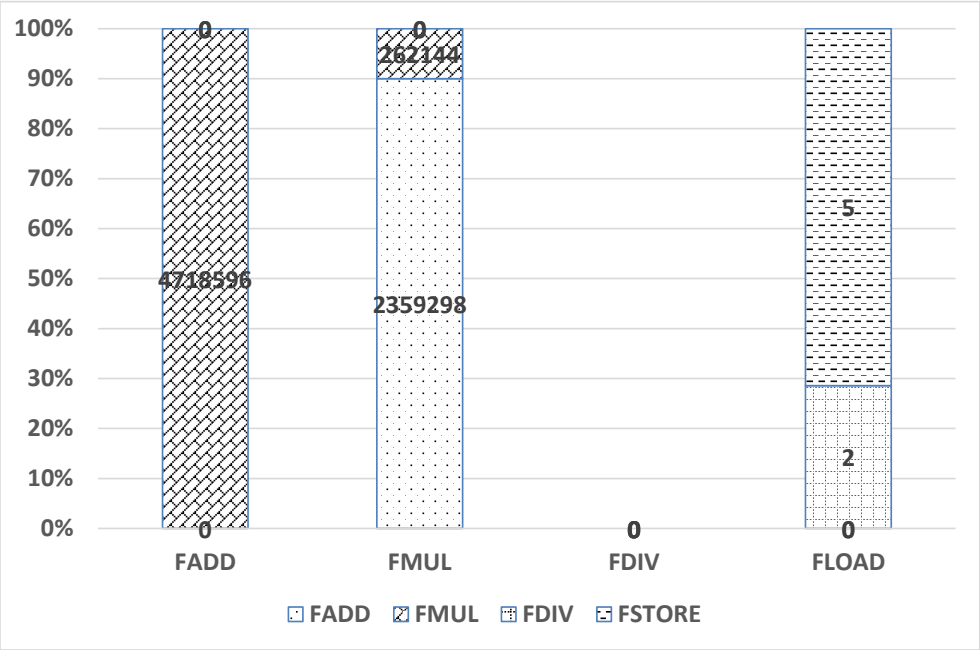

図 5.10: RADIX ソートにおける依存度

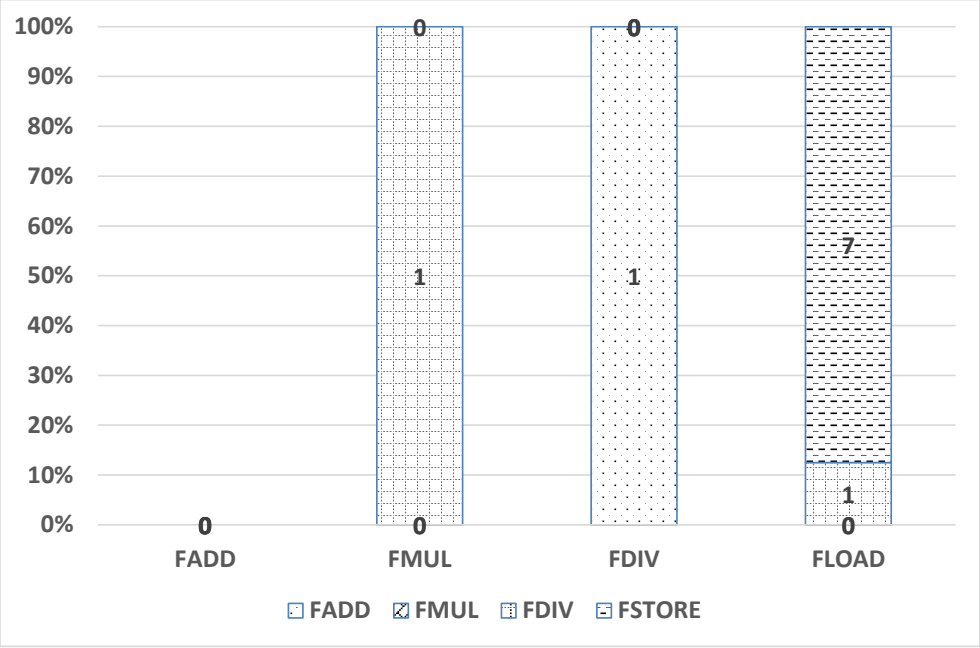

図 5.11: 姫野ベンチマークにおける依存度

6　おわりに

本研究では，FabScalar の FPU 設計機能の追加をさせるために，FPU の可変段数パイプライン化及びパイプライン統合に向けてのベンチマークの各命令依存の関係を調査した．しかし，FabScalar への可変段数パイプラインの追加及び構成最適化の定式化は行えていない．今後の展望として，まだ調査していない命令の命令依存度の調査及び FabScalar への FPU の可変段数パイプラインの追加及び統合機能の追加を行い，実際にハードウェア上でのこの手法の有効性及び性能変化を調査する必要がある．

謝辞

本研究を行うにあたり，多数のご指導を頂きました近藤敏夫教授，佐々木敬泰助教，並びに深澤研究員に深く感謝いたします．また，様々な局面にてお世話になりましたコンピュータアーキテクチャ研究室の皆様に深く感謝いたします．

参考文献

[1] T. Nakabayashi, et. al.“FabHetero: An Environment for Developing Diverse Heterogeneous Multi-core Processors. ISSEMU, Nov 2012.”

[2] N. K. Choudhary，et. al.”FabScalar: Composing. Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template ”. ISCA-38, pp. 11-22, June 2011.

[3] T. Okamoto，T.Sasaki and T.Kondo. FabCache: Cache Design Au-tomation for Heterogeneous Multi-core Processors. CANDAR-2013,pp.602-606, Dec 2013.

[4] Y. Seto, T. Nakabayashi, T. Sasaki, and T. Kondo. FabBus: A Bus Framework for Heterogeneous Multi-core processor. 28th International Technical Conferench on Circuits/Systems, Computers and Communications (ITC-CSCC2013), pp. 254-257, July 2013.

[5] SHASTRI, Ashlesha Vijay, et al. ”Microarchitectural Implementation of the MIPS Floating-point ISA in FabScalar-generated Superscalar Cores.”， 2012.

[6] 姫野龍太郎，姫野ベンチマーク，¡http://accc.riken.jp/supercom/himenobmt/¿

[7] S. C. Woo, et. al.:'The SPLASH-2 programs: Characterization and methodological considerations' , ACM SIGARCH Computer Architecture News. Vol. 23. No. 2. ACM, 1995