令和 5 年度

修士学位論文

# データ駆動型プロセッサのコンポジットコア化の検討

# A Study on Composite Core Architecture of Data-Driven Processor

1255117　古田雄大

指導教員　岩田誠

2024 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

# 要 旨

## データ駆動型プロセッサの<br>コンポジットコア化の検討

古田雄大

近年，IoT の急速な応用の広まりに伴い，IoT デバイスに搭載するコアには高性能と低消費電力性が求められている．そのため，それらを実現するヘテロジニアス・マルチコア (heterogeneous multicore: HMC) や強連結ヘテロジニアス・コア (tightly-coupled heterogeneous core: TCHC) 等のアーキテクチャが提案されている．TCHC の代表例であるコンポジットコア (Composite Core: CC) は単一のコア内に in-order で省電力動作する LITTLE µEngine と out-of-order で高性能に動作する Big µEngine と呼ばれる実行系を有する．そこで，エッジ・デバイス用アーキテクチャとして有望であるセルフタイム型パイプライン (Self-Timed Pipeline: STP) で動作するデータ駆動型プロセッサ (Data-Driven Processor: DDP)(以降，単に DDP) に，CC の特性を取り入れることができれば，更なる高性能化と低消費電力化が見込まれる．

DDP の CC 化を行うにあたり，従来命令セット，ベクトル・行列計算を含む提案高機能命令セットを実行制御できるパイプライン機構群をそれぞれ Little µEngine，Big µEngine として導入し，Big µEngine を実現するための専用回路を新規に考案した．

従来 DDP と提案 DDP を対象に Xilinx 社製 Zynq-7010 FPGA 回路を設計・実装し，評価を行ったところ，提案 DDP では従来 DDP と比べて大幅な性能向上と，両 µEngine を活用することで電力の削減が可能であることを確認した．

**キーワード**　IoT，強連結ヘテロジニアス・コア，コンポジットコア，データ駆動型プロセッサ，セルフタイム型パイプライン

# Abstract

## A Study on Composite Core Architecture of Data-Driven Processor

In recent years, with the rapid spread of IoT applications, high performance and low power consumption are required for the processor cores embedded in IoT devices. Therefore, architectures such as heterogeneous multi-core (HMC) and tightly-coupled heterogeneous core (TCHC) that have these features have been proposed. Composite core (CC), a representative example of TCHC, realizes both a power-efficient operation called LITTLE µEngine for in-order processing and a high-performance operation called Big µEngine for out-of-order processing within a single core.

Therefore, incorporating the characteristics of CC into a data-driven processor (DDP) operating with self-timed pipeline (STP), which is promising as an architecture for edge devices, could lead to further performance improvement and power efficiency.

In the process of transforming DDP into CC, we introduced pipeline mechanisms that can execute conventional instruction sets, as well as proposed high-performance instruction sets including vector-matrix calculations, as Little µEngine and Big µEngine, respectively. Additionally, we devised dedicated circuits to realize Big µEngine.

We designed and implemented FPGA circuits using Xilinx's Zynq-7010 for both conventional DDP and proposed DDP, and evaluated those DDPs. As a result, we confirmed significant performance improvement and power reduction compared to conventional DDP, by utilizing both µEngines.

***key words*** Internet of Things (IoT), tightly-coupled heterogeneous core (TCHC), Composite core (CC), data-driven processor (DDP), self-timed pipeline (STP)

# 目次


**第 1 章　序論** **1**

**第 2 章　背景技術** **5**

2.1　緒言 . . . . . . 5

2.2　ARM bigLITTLE . . . . . . 5

2.3　コンポジットコア (Composite Core: CC) . . . . . . 6

2.4　データ駆動型プロセッサ (Data-Driven Processor: DDP) . . . . . . 8

2.5　自己タイミング型パイプライン機構 (Self-Timed Pipeline: STP) . . . . 11

2.6　結言 . . . . . . 12

**第 3 章　DDP の CC 化** **14**

3.1　緒言 . . . . . . 14

3.2　CC 化の方策 . . . . . . 14

3.3　提案命令セットアーキテクチャ . . . . . . 15

3.4　提案アーキテクチャ構成 . . . . . . 16

3.5　提案回路構成 . . . . . . 18

3.5.1　PS1 回路の再構成 . . . . . . 18

3.5.2　EB 回路構成 . . . . . . 20

3.5.3　COPY N 回路構成 . . . . . . 20

3.5.4　vFP 回路構成 . . . . . . 23

3.5.5　BUF 回路構成 . . . . . . 27

3.6　結言 . . . . . . 29

**第 4 章　設計・評価** **31**

4.1　緒言 . . . . . . 31

4.2 FPGA 回路の設計・実装 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3 性能評価 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.4 回路規模・消費電力評価 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.5 消費電力量評価 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.6 考察 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.7 結言 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

**第 5 章 結論** **38**

**参考文献** **42**

**謝辞** **45**

**付録 A 従来 DDP 用行列計算プログラム** **46**

A.1 プログラム共通 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
A.2 スカラー倍 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
A.3 行列積 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
A.4 行列和 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

# 図目次

1.1 世界の IoT デバイス数の推移及び予測 “文献 [1] より” . . . . . . . . . . . 2

2.1 big コア (Cortex-A15) と LITTLE コア (Cortex-A7) の概要 “文献 [5] より” 6
2.2 CC アーキテクチャ概要 “文献 [8] より” . . . . . . . . . . . . . . . . . . . . 7
2.3 DDP 基本構成例 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.4 STP 構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.5 環状 STP 構成図 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.1 提案並列パイプライン構成 . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 提案環状 STP 構成図 . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3 新 PS1 回路構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.4 PS1 パケットフォーマット . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.5 CBE 回路構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.6 高機能命令用パケット複製機構 (Copy Unit for vFP: COPY N) 回路構成 21
3.7 COPY N パケットフォーマット . . . . . . . . . . . . . . . . . . . . . . 21
3.8 CCN 回路構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.9 3bit 入力 decoder(グリッチ防止回路構成) . . . . . . . . . . . . . . . . 23
3.10 vFP 回路構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.11 vFP パケットフォーマット . . . . . . . . . . . . . . . . . . . . . . . . 24
3.12 ALU Array 回路構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.13 CWRE 回路構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.14 BUF 回路構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.15 BUF パケットフォーマット . . . . . . . . . . . . . . . . . . . . . . . . 28
3.16 CF 回路構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

図目次

4.1　従来 DDP と提案 DDP のレイアウト図 “Vivado の Device 画面より” . . 32

# 表目次

2.1 DDP パケットのフィールド構成例 . . . . . . . . . . . . . . . . . . . . . . . 9

3.1 提案高機能命令セットアーキテクチャ . . . . . . . . . . . . . . . . . . . . . . . 15

4.1 追加実装するメモリ . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

4.2 従来命令セットに追加する命令 . . . . . . . . . . . . . . . . . . . . . . . . 33

4.3 従来 DDP，提案 DDP による各行列命令の処理時間 . . . . . . . . . . . . . 33

4.4 従来 DDP，提案 DDP の回路規模 . . . . . . . . . . . . . . . . . . . . . . . 33

4.5 Little µEngine，Big µEngine の回路規模 . . . . . . . . . . . . . . . . . . . 34

4.6 従来 DDP，提案 DDP の消費電力 . . . . . . . . . . . . . . . . . . . . . . . 35

4.7 従来 DDP，提案 DDP で各行列命令を実行した際の消費電力量 . . . . . . 35

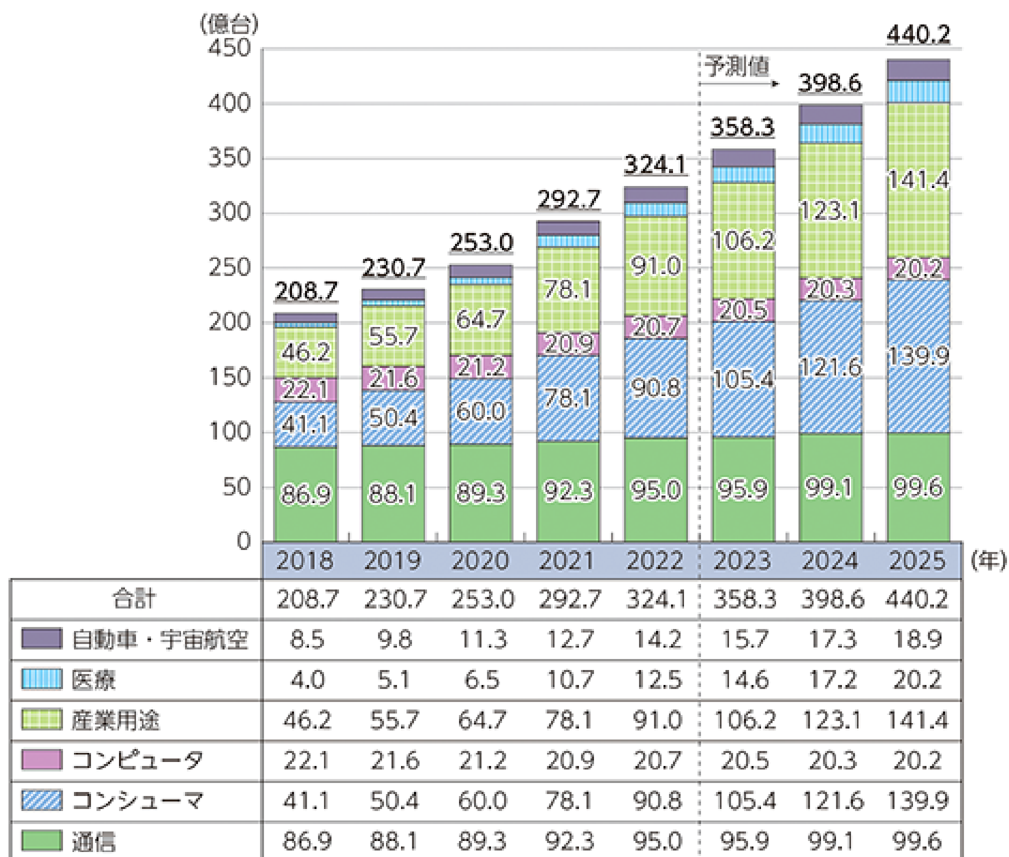
# 第 1 章

# 序論

近年，ネットワークを介してコンピュータ資源をサービスの形で提供するクラウドコンピューティング (クラウド) が世界的に普及し，オンラインであれば必要な時に必要なサービスを受けられるようになり，パソコンやスマートフォンなどのインターネット接続端末に加え，家電製品，自動車，農作機械などもネットワークにつながるようになっている．このような，身の回りのあらゆるものがネットワークにつながる仕組みである IoT の急速な応用の広まりに伴い，IoT デバイス数は増加傾向にあり，図 1.1 に示すように 2025 年には世界の IoT デバイス数はおよそ 440 億台に到達すると予測されている [1].

多くの IoT システムでは，各種センサを搭載した IoT デバイスでデータ収集を行い，それを通信回線経由でクラウドに送信して分析処理を行っている．データ収集を行うほとんどの IoT デバイスのプロセッサの性能は低く，メモリ容量も小さいため IoT デバイス側では複雑な処理をさせることができず，収集したデータをほぼそのままの状態でクラウド側に送信している．つまり，膨大な量のデータを送信してしまうということであるので，その結果，ネットワークの遅延，データの破損，コストの増大化等を招く．そこでデバイス (エッジ) 側である程度データを加工し，必要最低限のデータのみをクラウドに送信させる処理機能を持たせ，クラウドの負荷分散を実現するエッジ・コンピューティングが注目されている．そのようなクラウドの負荷分散を担うエッジ・デバイスには基本的に以下に示す要素が求められる．

- 高性能




| | 2018 | 2019 | 2020 | 2021 | 2022 | 2023 | 2024 | 2025 |
|---|---|---|---|---|---|---|---|---|
| 合計 | 208.7 | 230.7 | 253.0 | 292.7 | 324.1 | 358.3 | 398.6 | 440.2 |
| 自動車・宇宙航空 | 8.5 | 9.8 | 11.3 | 12.7 | 14.2 | 15.7 | 17.3 | 18.9 |
| 医療 | 4.0 | 5.1 | 6.5 | 10.7 | 12.5 | 14.6 | 17.2 | 20.2 |
| 産業用途 | 46.2 | 55.7 | 64.7 | 78.1 | 91.0 | 106.2 | 123.1 | 141.4 |
| コンピュータ | 22.1 | 21.6 | 21.2 | 20.9 | 20.7 | 20.5 | 20.3 | 20.2 |
| コンシューマ | 41.1 | 50.4 | 60.0 | 78.1 | 90.8 | 105.4 | 121.6 | 139.9 |
| 通信 | 86.9 | 88.1 | 89.3 | 92.3 | 95.0 | 95.9 | 99.1 | 99.6 |

図 1.1　世界の IoT デバイス数の推移及び予測 “文献 [1] より”

- 低消費電力
- 低ハードウェアコスト
- 再構成の容易さ
- 低開発コスト
- 高セキュリティ

本研究は，これら要求要素のうち高性能，低消費電力，再構成の容易さ，低開発コストに焦点を当てる．

これまで高性能と低消費電力性を持つエッジ・デバイス用マイクロプロセッサを実現する技術がいくつか提案されてきた [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] が，今回はその中から特にヘテロジニアス・マルチコア (heterogeneous multicore: HMC)，強連結ヘテ

ロジニアス・コア (tightly-coupled heterogeneous core: TCHC) に着目した．HMC はデバイスの高性能化と低消費電力化の両立を図ることを目的として，1 つの CPU に特性が異なる複数のコアを搭載している．タスクの特性に応じて使用するコアを動的にスイッチングし，その特性に特化したコアで処理を行わせることで効率よく各コアを利用する技術である．商用化された HMC の例として，ARM big.LITTLE アーキテクチャが挙げられる [4, 5, 6, 7]．一方，TCHC は非対称な複数の実行系を単一のコア内に持つ．実行系同士が密に隣接しているため，HMC よりも細粒度な実行系切り替え制御が可能となる．代表的なものとして，Lukefahr らが提案した コンポジットコア (Composite Core: CC) がある [8, 9]．しかし，これら技術が対象とするプロセッサはいずれもノイマン型プロセッサであり，クロック信号により回路制御を行う同期回路となっているため，ノイマン型プロセッサにて複数のストリームデータに対して処理を行う場合，通常割り込み処理によってこれを実現し，データの退避や処理の復帰といった命令を別途実行しなければならない．よって，プロセッサの処理速度が低下し，総処理時間が増加することが予想される．また，複数のセンサから到着した個々のデータに対して処理を行うため，プロセッサの消費電力も増大してしまう．このような特徴があることからノイマン型プロセッサはエッジ・コンピューティングに適したプロセッサとは言い難い．一方，自己タイミング型パイプライン機構 (Self-Timed Pipeline: STP) で動作するデータ駆動型プロセッサ (Data-Driven Processor: DDP) (以降，単に DDP) は，その構造から割り込み処理を行わず，多様なセンサ等から到着する異なる複数のストリームデータに対する多重処理が可能である高性能性と，クロック信号を用いず，データ入力がトリガとなり処理が実行されるため，必要な時に必要な回路のみ動作するという低消費電力性も兼ね備えていることから，エッジ・デバイス用アーキテクチャとして有望である．

これら技術はエッジ・デバイスの高性能化，低消費電力化が急務となっている近年において．この分野の技術向上に欠かせないものとなっている．最新技術としては，3 種の CPU コアを利用して高性能と低消費電力性の両立を図っているものもある [14, 15]．これらのアーキテクチャを組み合わせ，各アーキテクチャの利点を取り入れることがで

きれば，エッジ・デバイスの更なる高性能化と低消費電力化につながるのではないかと考えられるため，その組み合わせ方法に関して検討することには意義がある．実際，筆者は以前 DDP に big.LITTLE の特性を取り入れる研究を行ったことがある [16]．本研究では，DDP の CC 化を提案する．

高性能化，低消費電力化が議論される傍ら，情報処理を担う集積回路は微細化の限界，発熱 (または消費電力の増大) による性能低下に直面しており，従来の集積回路では更なる電力効率の向上が困難になっている．そこで，特定のアプリケーションに特化した専用集積回路 (Application Specific Integrated Circuit: ASIC) が用いられるが，その構造から処理能力，電力効率が高いものの，柔軟性 (またはプログラマビリティ) に欠ける，開発コストが高額である等の課題がある．一方，FPGA (Field Programmable Gate Array) は製造後に購入者や設計者が内部回路をプログラムし，アプリケーションに合わせた論理回路構成を回路上に何度でも再構築できる集積回路であるため，これを利用することでエッジ・デバイスが求める再構成の容易さ，低開発コストを実現できる．

これ以降本稿において，第 2 章では，ARM bigLITTLE，CC の構成と動作特性を説明し，技術的課題を述べる．その後，従来 DDP アーキテクチャと，STP の動作原理について説明する．

第 3 章では，DDP の CC 化の方策を述べた後，提案命令セットアーキテクチャ，提案 DDP アーキテクチャ構成，提案回路構成について説明することで提案 DDP を示す．

第 4 章では，従来 DDP と提案 DDP を対象として FPGA 回路を設計・実装し，性能，消費電力の観点からの評価を行い，提案 DDP のエッジ・デバイス用アーキテクチャとしての有効性を測る．

第 5 章では，本研究で提案した DDP の CC 化についてまとめ，今後の課題を述べ，本論文を総括する．

# 第 2 章

# 背景技術

## 2.1　緒言

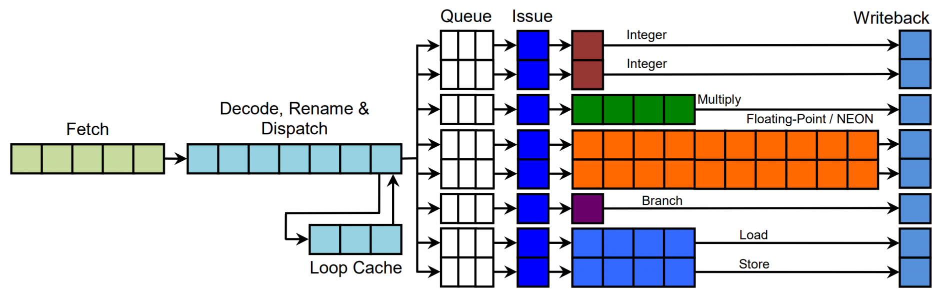
エッジ・デバイス用マイクロプロセッサの高性能化，低消費電力化のため，HMC，TCHC，DDP 等のエッジ・デバイス用アーキテクチャがこれまで提案されてきた．これらの特徴をうまく組み合わせることで，新たなアーキテクチャ構成を考えていくが，そのためにはまず既存アーキテクチャの技術的特徴・課題を整理する必要がある．

本章では，本稿で取り上げる ARM bigLITTLE と CC の技術的特徴を説明し，技術的課題を述べる．続いて，DDP アーキテクチャの動作特性，パイプラインステージ構成について説明し，パケットのフィールド構成例を示す．最後に，STP の動作原理とそれを構成する C 素子の種類と働きについて説明し，DDP アーキテクチャに合わせた環状 STP 構成図を示す．

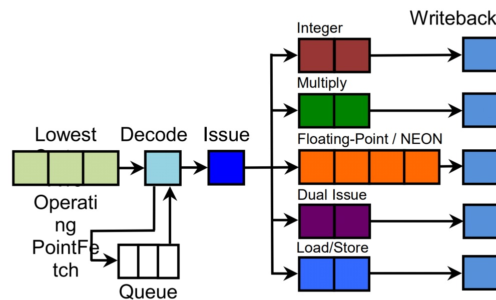
## 2.2　ARM bigLITTLE

ARM bigLITTLE は HMC の代表例である．高性能で動作する big コアと低消費電力で動作する LITTLE コアを 1 チップ上に備えており，これらコアをプログラムのフェーズに応じて使い分けることでプロセッサのエネルギー効率を向上させている．big コアと LITTLE コアの概要を図 2.1(a)，図 2.1(b) に示す．

しかし，HMC では各コアでそれぞれ独立したキャッシュや分岐予測器を持つため，コア間でのコンテキストスイッチング時のオーバーヘッドが非常に大きくなってしまうと




(a) bigコア (Cortex-A15)




(b) LITTLEコア (Cortex-A7)

図 2.1 big コア (Cortex-A15) と LITTLE コア (Cortex-A7) の概要 “文献 [5] より”

いう問題がある．そのため，コアの切り替えは 100M 命令程度の比較的長い間隔で行われることが多い．

## 2.3 コンポジットコア (Composite Core: CC)

HMC の切り替え間隔を軽減するために TCHC が提案されているが，その代表例として CC がある．CC は単一のコア内に in-order で低消費電力で動作する Little µEngine と out-of-order で高性能に動作する Big µEngine と呼ばれる実行系を有している．CC アーキテクチャ概要を図 2.2 に示す．

Little µEngine が動作するモードを low-power(LP) モード，Big µEngine が動作するモードを high-performance(HP) モードと呼び，これら実行モードを切り替えながら命令を実行する．キャッシュ，命令フェッチ機構は共通しており，デコード以降の処理を

### 2.3　コンポジットコア (Composite Core: CC)

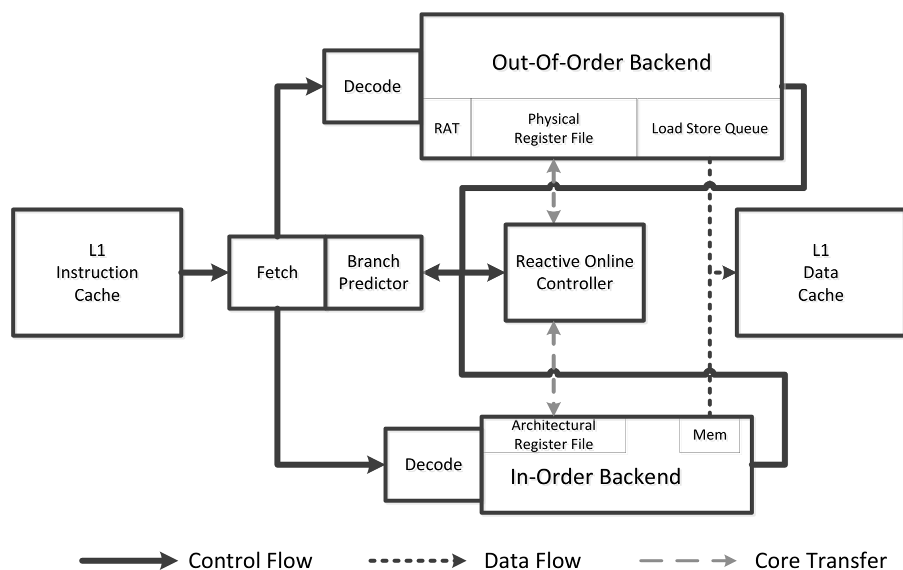



図 2.2　CC アーキテクチャ概要 “文献 [8] より”

各 µEngine が担当する．レジスタファイルは各 µEngine がそれぞれ有しており，実行モードが切り替わる際には以下の手順を踏みコヒーレンスをとる．この動作にかかる時間が実行モード切り替えペナルティとなる．

1. 命令フェッチ停止後，アクティブな実行系から全命令がリタイアされるまで待機
2. 命令フェッチの停止と並行して，それぞれの実行系が有するレジスタ・ファイル間で値の転送を投機的に実行
3. 全命令のリタイア後，転送に失敗したレジスタ値を再転送
4. 再転送終了後，切り替え先の実行系で処理を開始

実行モード切り替え自体のペナルティは HMC のコア切り替え時に生じるペナルティと比べ非常に短くなっているが，細粒度で実行系の切り替え制御を行った場合，その分ペナルティが無視できないほど増加してしまう．そのため，切り替え粒度は最低でも 100 命令程度に制限される．

## 2.4　データ駆動型プロセッサ (Data-Driven Processor: DDP)

DDP は待ち合わせ機構にて必要なデータ (パケット) が揃い次第演算を実行するプロセッサである．DDP の基本構成例を図 2.3，DDP パケットのフィールド構成例を表 2.1 に示す．パケットが外部から入力されるとその対となるパケットが到着するまで待ち合わせを行い，パケットが揃えば命令を読み出し，その命令に応じた演算を実行する．また，演算を終えると，次の命令に備えるため再度内部への入力，またはプログラム終了のため外部への出力を行う環状パイプライン構成となっているため，環状パイプライン上で周回するパケットが詰まらない程度であればデータ依存関係に基づき多重並列実行が可能である．

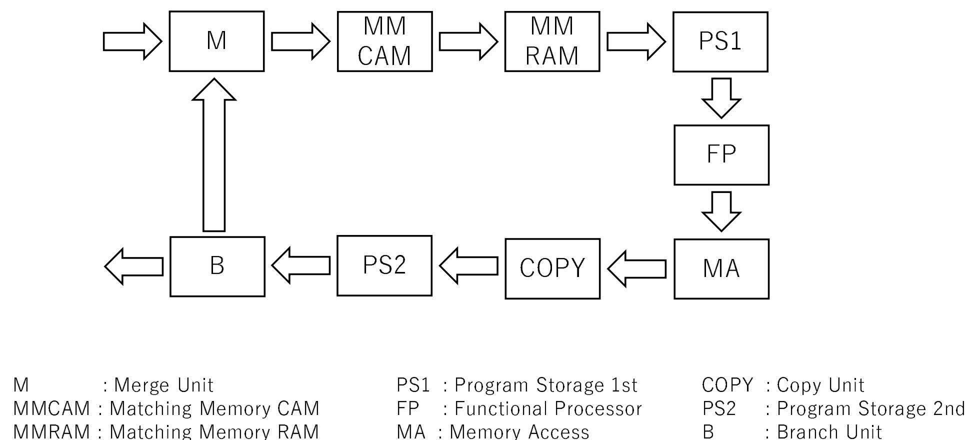



図 2.3　DDP 基本構成例

本研究で扱う DDP は 9 つのパイプラインステージで構成される．各パイプラインステージの詳細を以下に示す．

- 合流機構 (Merge Unit: M)
  DDP は環状パイプラインを流れるパケットに対して処理を行うため，パケットが外部から入力される経路と内部から入力される経路が存在するが，それら経路を流れるパケットの合流を調停する．

表 2.1　DDP パケットのフィールド構成例

| フィールド名 | 名称 | 情報 | ビット数 |
|---|---|---|---|
| color | Color | タスクの識別情報 | 3bit |
| gen | Generation | パケットの計算順序識別情報 | 8bit |
| dest | Destination | パケットの宛先情報 | 7bit |
| LR | Left or Right | パケットの左右識別情報 | 1bit |
| MF | Maching flag | 待ち合わせの有無識別情報 | 1bit |
| OPC | Operation Code | 命令コード | 6bit |
| C | Carry flag | 桁上げ有無識別情報 | 1bit |
| Z | Zero flag | 計算結果が 0 であるかを示す情報 | 1bit |
| Data | Data | 演算に使用するデータ | 16bit |
| CPY | Copy flag | パケットの複製要否識別情報 | 1bit |
| BR | Branch flag | パケット行先情報 | 1bit |

- パケット待ち合わせ機構 (Matching Memory CAM: MMCAM)
  パケット同士の演算を行う場合，その対となるパケットが到着する (発火する) まで待ち合わせをする．内部の連想メモリにパケット情報 (color, gen, dest, LR) を一時的に保存し，発火すると，それを後段機構に伝える．
- パケットデータ保持・定数読み出し機構 (Matching Memory RAM: MMRAM)
  DDP における 2 項演算は 2 パケットが保有するオペランド同士の演算として実行されるため，2 項演算を行う場合，dest をアドレスとして RAM(Random Access Memory) にデータ (C, Z, Data) を一時的に保存する．発火を検出すると保存されたデータを読み出し，それを対となるパケットに付加して出力する．
  定数演算を行う場合，dest をアドレスとして Constant Memory からデータを読み出し，それをパケットに付加して出力する．
- 第 1 命令フェッチ機構 (Program Storage 1st: PS1)

dest をアドレスとして Program Storage からパケット更新情報 (CPY, OPC) を読み出し，それをパケットに付加する．

- 演算機構 (Functional Processor: FP)
  一つ目のパケットの Data(DataL) を第一オペランド，二つ目のパケットの Data(DataR) を第二オペランドとして OPC に応じた演算 (加算，減算，乗算，論理演算，シフト演算，条件分岐 (dest の更新)，ロード命令に必要なアドレスの計算，識別子変換 (color，gen の更新) など) を行う．
- データメモリアクセス機構 (Memory Access: MA)
  必要に応じて Data Memory にアクセスする．OPC がストア命令を示す場合, DataL をアドレスとして DataR を書き込む．OPC がロード命令を示す場合，FP で計算したアドレスをもとに Data Memory から値を読み出し，それをパケットの Data に反映する．
- パケット複製機構 (Copy Unit: COPY)
  必要に応じてパケットをコピーする．パケットをコピーするかどうかは CPY の値で判断する．複製されたパケットは複製元のパケットと区別するため dest を+1 する．
- 第 2 命令フェッチ機構 (Program Storage 2nd: PS2)
  dest をアドレスとして Program Storage からパケット更新情報 (dest, LR, MF, BR) を読み出し，それをパケットに付加・反映する．
- 分流機構 (Branch Unit: B)
  BR の値に応じてパケットを内部へ入力，または外部へ出力させる．

## 2.5　自己タイミング型パイプライン機構 (Self-Timed Pipeline: STP)

本研究で扱う DDP は自己タイミングでデータ処理を行う非同期式回路である STP で動作する．STP は大域的なクロック信号を必要とせず，隣接するパイプライン段間において，ハンドシェイク信号のみを使ったデータの受け渡しができるパイプライン処理システムを構成できる回路構成法として有望である [17]．STP の構成を図 2.4 に示す．

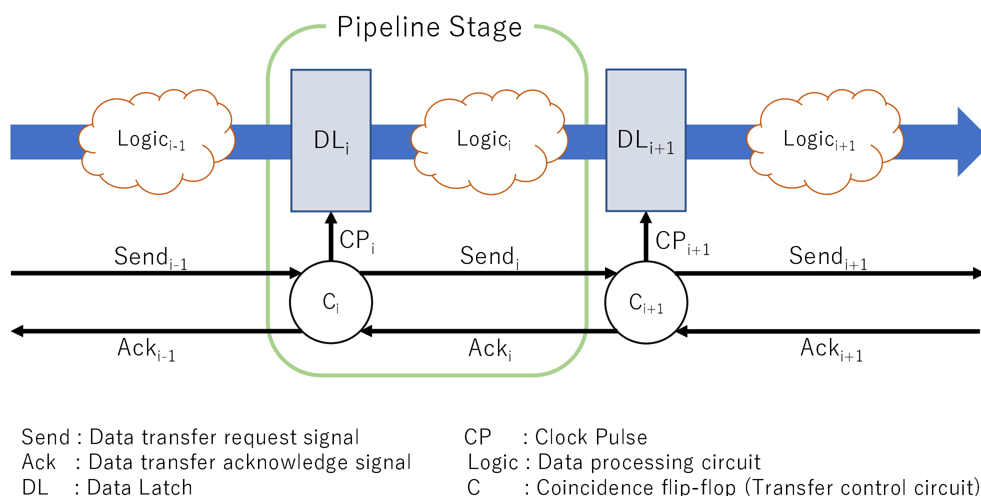



図 2.4　STP 構成

1 パイプラインステージは，ステージロジック (Logic)，データラッチ (DL)，転送制御回路 (C 素子) により構成される．前段の Logic での処理が終了したパケットが DL に取り込まれると同時に，C 素子が前段の C 素子とデータ転送要求信号 (Send 信号)，データ転送許可信号 (Ack 信号) によるハンドシェイク通信を行い，その後，データラッチ開放信号 (CP 信号) を立ち上げる．この CP 信号の立ち上がりを検知した DL はパケットを開放し，Logic にデータを転送する．このように STP においては，パケットがパイプラインステージ間を移動する際に局所的に転送制御が行われていくため，電力の消耗を最小限に抑えることができる．

C 素子にはいくつか種類があり，各パイプラインステージで行う処理に応じた C 素子

を割り当てている．既存の各 C 素子について以下に示す．

- 通常 C 素子
  特殊な動作が必要なければこれを基本的に使用する．
- 2 入力 1 出力用 C 素子 (C Merge: CM)
  外部および内部からの Send 信号と Ack 信号を受け，パケットの合流の調停を実現する．
- パケット消去機能付き C 素子 (C Erase: CE)
  DDP 内部を巡回している不要なパケットを削除するための C 素子．パケットの待ち合わせ時や ABSORB 命令実行時に使用される．DEL 信号を外部から入力し，その値をもとにパケットを削除するかどうかを決定する．
- パケット複製機能付き C 素子 (C Copy: CC)
  パケットを複製するための C 素子．CPY 信号を外部から入力し，その値をもとにパケットを複製するかどうかを決定する．
- 1 入力 2 出力用 C 素子 (C Branch: CB)
  外部および内部へ Send 信号と Ack 信号を渡し，パケットの分流を実現する．

図 2.3 の DDP 基本構成例に合わせて上記 C 素子を環状につなげると図 2.5 のようになる．パケットが外部に出力する際，パケットはもう 1 つ DL を通過するため，その DL への CP 信号を出力する C 素子を最後尾につなげている．

## 2.6　結言

本章では，本稿で取り上げる ARM bigLITTLE と CC の技術的特徴を説明し，技術的課題を述べた．続いて，DDP アーキテクチャの動作特性，パイプラインステージ構成について説明し，パケットのフィールド構成例を示した．最後に，STP の動作原理とそれを構成する C 素子の種類と働きについて説明し，DDP アーキテクチャに合わせた環状 STP 構成図を示した．

## 2.6　結言

図 2.5　環状 STP 構成図

次章では，DDP の CC 化の方策，提案命令セットアーキテクチャ，提案アーキテクチャを説明し，それを実現するための具体的な回路構成を示す.

# 第 3 章

# DDP の CC 化

## 3.1　緒言

本研究では，第 2 章にて述べた DDP の動作特性と，CC の HMC より回路規模を節約できる点に着目し，DDP の動作をベースとしながらも CC の特性を併せ持つ IoT 向きプロセッサ構成を提案する．

本章では，DDP の CC 化のための方策を述べた後，提案命令セットアーキテクチャ，提案アーキテクチャ構成を述べ，それを実現するための詳細な提案回路構成を示す．

## 3.2　CC 化の方策

まずは，DDP の動作特性に目を向けてみる．DDP はデータ駆動であり，内部に命令メモリやレジスタ・ファイルを有しない．そのため，仮に DDP に Little µEngine，Big µEngine に相当する機構を CC のように導入したとしても，実行系切り替え時の特別な処理 (命令のリタイア，レジスタ・ファイル間の転送・再転送) は不要となる．よって，実行系切り替え粒度は CC ほど制限されることはない．

CC の特性を従来 DDP に取り入れると，以下の特性を持つアーキテクチャとなることが予想された．

- 【CC 特性】実行系の切り替えにより高性能かつ低消費電力動作が実現できる
- 【CC 特性】実行系切替え時のオーバーヘッドを相対的に低減できる
- 【DDP 特性】両実行系ともに多重並列実行が可能である

- 【DDP 特性】処理の実行にクロック信号を必要としない

このアーキテクチャ実現のために DDP 内部に CC の Little µEngine，Big µEngine に相当する実行系を導入する方法について検討する．

## 3.3 提案命令セットアーキテクチャ

CC の Little µEngine，Big µEngine に相当する実行系の導入前に，まずは，各 µEngine で実行する命令セットを定義する必要がある．方法としては従来命令セットを据え置いたとして，その命令セットの低機能化，または高機能化を図るなどして命令セットの差別化を行わなければならないが，DDP の従来命令セットは，既に汎用的な処理 (算術演算，論理演算，条件分岐等) の集合であったため，従来命令セットは据え置き，新たなより高機能な命令セットの導入が望ましいと考えた．高機能な命令としては，例えば，科学技術計算やデジタル信号処理において必須の処理となる浮動小数点演算や積和演算，一つの命令を複数のデータに適用して並列処理を行う SIMD(Single Instruction, Multiple Data) 演算等が考えられた．本研究では，その中でも近年注目されている深層ニューラルネットワーク等で多用されるベクトル・行列計算を含む高機能命令セットを定義し，従来命令セット，提案高機能命令セットを実行制御できるパイプライン機構群をそれぞれ DDP 用 Little µEngine，Big µEngine として導入する．

手始めに，ベクトル・行列計算命令としては，スカラー倍，行列積，行列和を採用する．提案高機能命令セットアーキテクチャを表 3.1 に示す．

表 3.1 提案高機能命令セットアーキテクチャ

| Instruction | OPC | Operation |
|---|---|---|
| スカラー倍 | 000 001 | 行列 $A$ にスカラー $a$ をかける |
| 行列積 | 000 010 | 行列 $A$ に行列 $B$ をかける |
| 行列和 | 000 011 | 行列 $A$ に行列 $B$ をたす |

従来命令セットに続きベクトル・行列計算命令を新たに追加していくとその総命令数はOPCの従来ビット幅(6bit)で表現できる範囲を超えてしまうことが予想されたが，従来ビット幅のまま，既存の命令との重複に構わず，任意の値に命令を割り当てる．OPCの重複の問題については，3.4節で示す提案アーキテクチャ構成により解消される．

## 3.4　提案アーキテクチャ構成

ベクトル・行列計算命令の実行はポインタの授受，ベクトルレジスタ，ベクトル演算器の導入により実現する．ベクトル・行列計算命令はおおまかに以下の手順で実行する．ただし，パケットはオペランドの代わりに読み込み用ポインタをDataとして保持する．

1. 命令コードの読み出しと同じタイミングで書き込み用ポインタを読み出してパケットに付加
2. 読み込み用ポインタをもとに対象のベクトル(オペランド)をベクトルレジスタから読み出す
3. ベクトル演算器で命令コードに応じた演算を実行
4. 書き込み用ポインタをもとに演算結果をベクトルレジスタに書き込む.
5. 書き込み用ポインタを読み込み用ポインタとして，次の命令に備えるため再度内部へ入力するか，プログラムを終了するため外部へ出力する

以上を考慮し，提案アーキテクチャは図2.3に示す従来DDPの演算機構FP (Functional Processor) とメモリアクセス機構MA (Memory Access) を，図3.1に示すLittle µEngineとBig µEngineを並列に配置した並列パイプライン構成に置換することで実現する．

sFP (Scalar FP) とする既存のFPと既存のMAを併せてLittle µEngineとし，新規に導入した高機能命令用パケット複製機構COPY N (Copy Unit for vFP) とベクトル・行列演算機構vFP (Vector FP) を併せてBig µEngineとする．各µEngineでの実行を切り替えるために， 既存の分流機構Bおよび合流機構MをそれぞれEB (Execution Branch

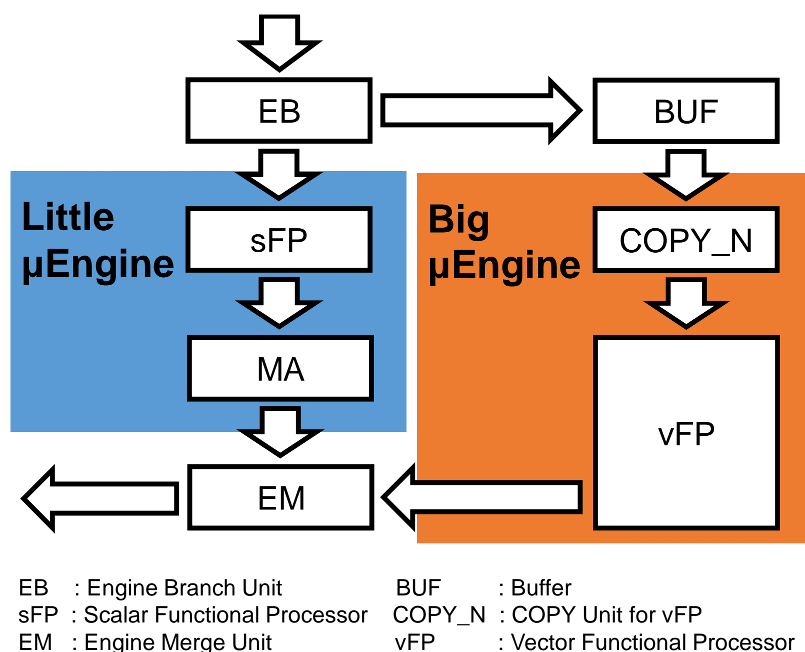



図 3.1　提案並列パイプライン構成

Unit) と EM (Execution Merge Unit) として追加している．EB に到達したパケットは新たに付加した識別情報を元に各 µEngine に振り分け，処理が終了した各 µEngine 内のパケットは EM で合流させる．

Big µEngine での処理時間が長いとパケットが STP 内に溢れ，その間 Little µEngine での処理も不可能となる．そのため，あらかじめ処理に時間がかかることを見越して一時的にパケットをいくつか溜めるためのバッファー機構 (Buffer: BUF) を Big µEngine 前に配置する．

図 3.1 の提案並列パイプライン構成導入後の環状 STP 構成を図 3.2 に示す．

今回新規に導入する各 C 素子の概要を以下に示す．

- パケット消去機能付き CB (C Branch & Erase: CBE)

  CB と CE の仕組みを併せ持つ．

- バッファー構成用 C 素子 (C FIFO: CF)

  RAM とともにリングバッファーを構成する．

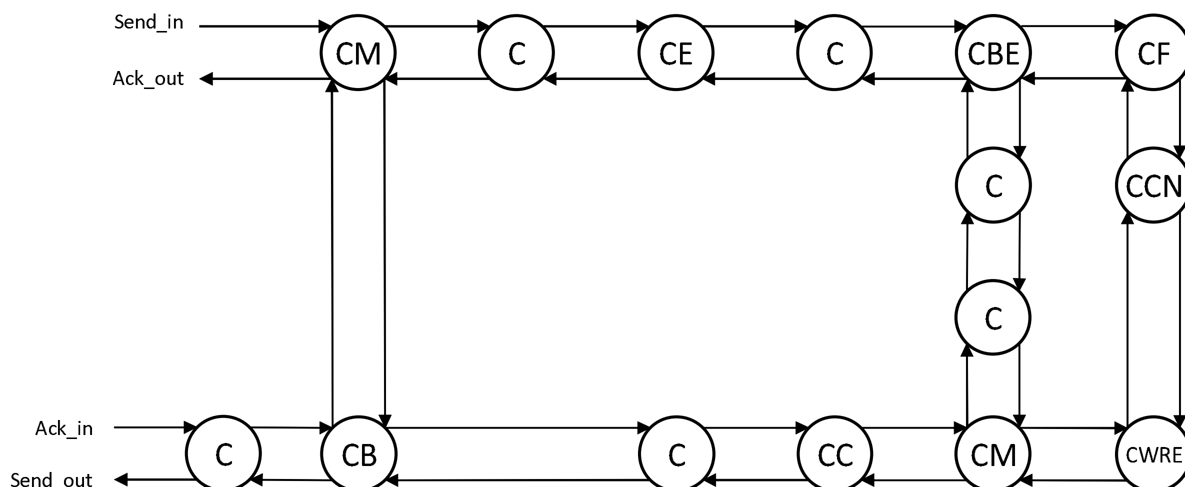



図 3.2　提案環状 STP 構成図

- 高機能命令用パケット複製機能付 C 素子 (C Copy N: CCN)

  CC の拡張版．パケットを N-1 回複製する．

- パケット消去機能付読込･書込信号生成 C 素子 (C Write after Read & Erase: CWRE)

  一度のパケット通過でベクトルレジスタへの読み込みと書き込みを可能とする．

## 3.5　提案回路構成

3.4 節で述べた提案アーキテクチャを実現するために，一部既存回路の再構成と新規回路の導入が必要となる．以降，その具体的な回路構成を述べる．

### 3.5.1　PS1 回路の再構成

新 PS1 回路構成を図 3.3，パケットフォーマットを図 3.4 に示す．

行列計算を実行する場合，オペランド読み込み用の 2 ポインタに加え，演算結果書き込み用のポインタが必要となる．読み込み用 2 ポインタは従来パケットの DataL(16bit の内，下位 8bit)，DataR(16bit の内，下位 8bit) にそれぞれ割り当て，書き込み用ポインタは，PS1 内に新しく配置する ROM(Assignment Pointer Table: APT) から，dest に

## 3.5　提案回路構成

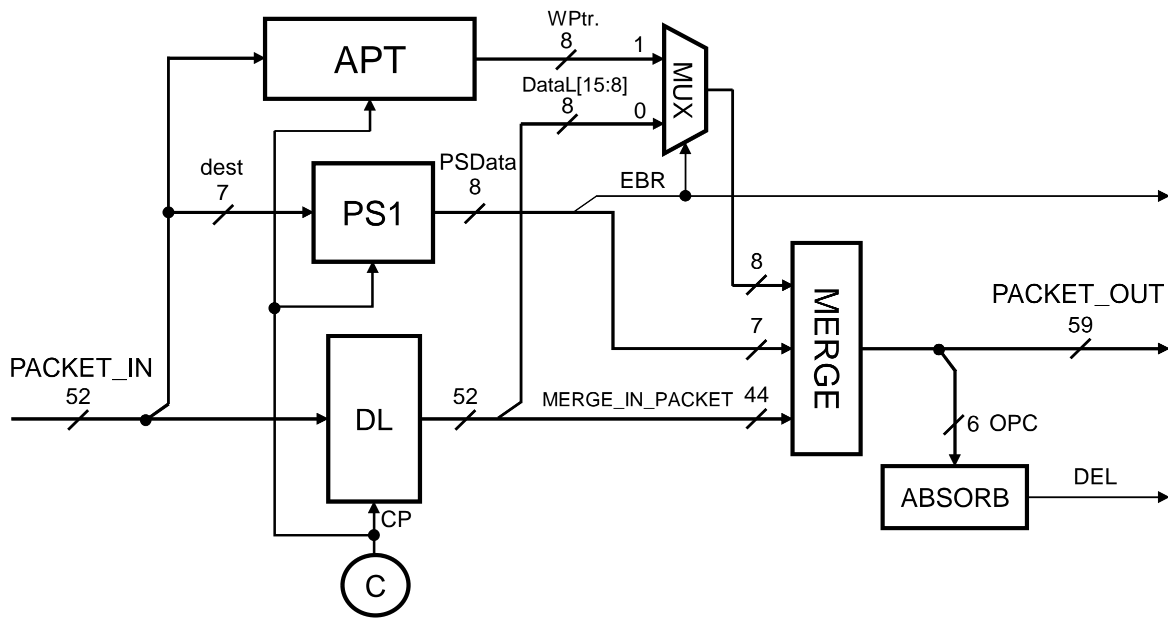



図 3.3　新 PS1 回路構成

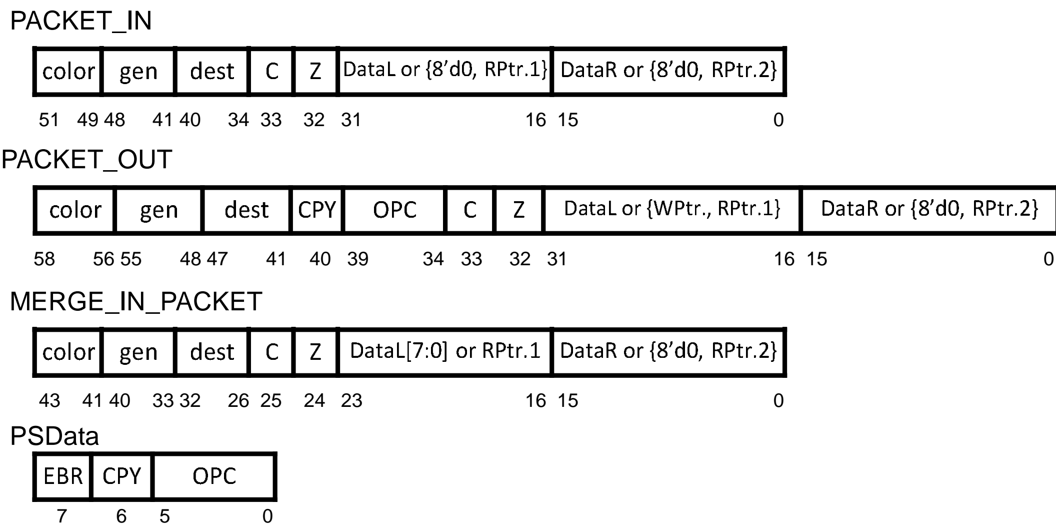



図 3.4　PS1 パケットフォーマット

応じて読み出し，パケットの DataL(16bit の内，上位 8bit) に付加するようにする．また，PS1 メモリに µEngine 用パケット行先情報 (µEngine Branch Flag: EBR) を CPY, OPC に加えて格納し，読み出し後は後段機構 (EB) で使用する．EBR は DataL の上位 8bit を決定するためのマルチプレクサの SEL 信号としても使用される．

### 3.5.2　EB 回路構成

図 3.1 に示す並列パイプライン構成導入前は PS1 の後段機構は FP であった．この時，FP を構成する C 素子は ABSORB 命令の実行を考慮した CE であり，PS1 から受け取る DEL 信号をもとにパケットを削除するかどうかを判断していた．同様に，今回は後段機構である EB にてパケットの削除ができるようにする必要があるが，EB には CB の割り当てを想定しており，CE を割り当てられない．そこで，CB にパケット削除機能を付加することを考えた．パケット削除機能付き CB(C Branch & Erase: CBE) の回路構成を図 3.5 に示す．

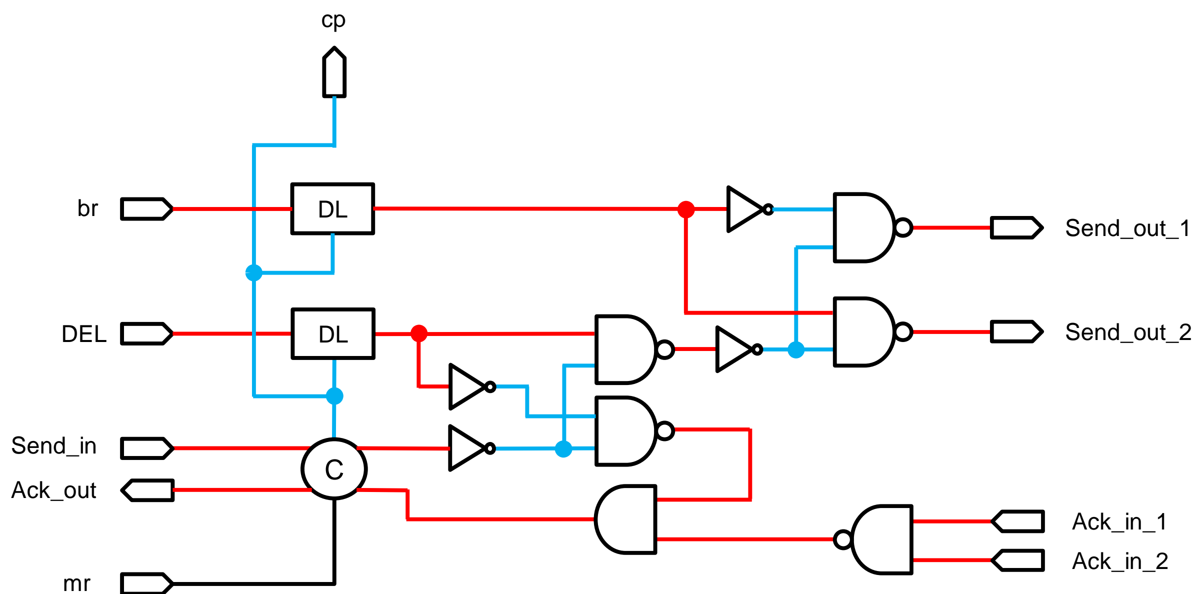



図 3.5　CBE 回路構成

### 3.5.3　COPY N 回路構成

COPY N は，パケットが vFP に転送される前にパケットを N-1 回複製し，それらパケットで一つの命令を実行するための高機能命令用パケット複製機構である．この機構の存在により，従来 DDP の構成的な問題やその演算の特徴等により，一度のパケット通過で処理を終えることが難しい命令 (例えば，行列積) を考慮できる．COPY N の回

路構成を図 3.6，パケットフォーマットを図 3.7 に示す．

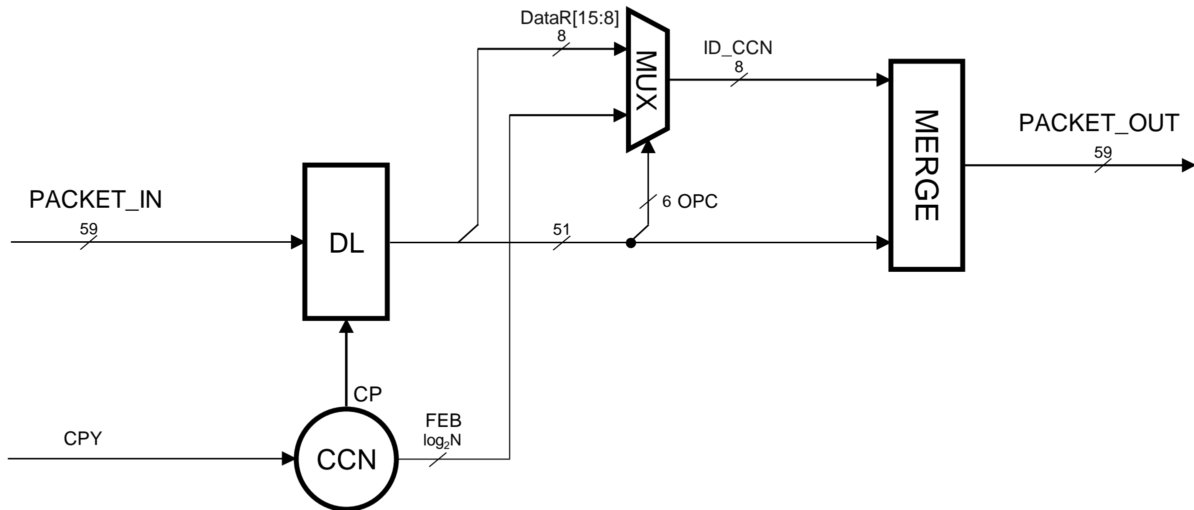



図 3.6　高機能命令用パケット複製機構 (Copy Unit for vFP: COPY N) 回路構成

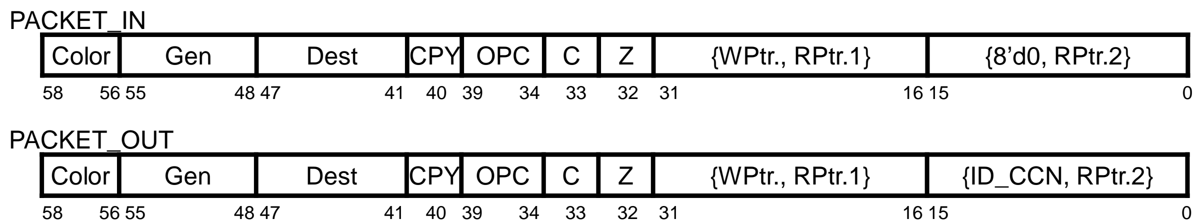



図 3.7　COPY N パケットフォーマット

既存のパケット複製機能付き C 素子である CC は，パケットの複製を 1 度だけ行う仕組みとなっていたため，複製を複数回行えるように回路を拡張する必要があった．パケットの複製を N-1 回行う高機能命令用パケット複製機能付き C 素子 (C CopyN: CCN) を図 3.8 に示す．

CCN は，パケットが保持する OPC をもとに生成された CPY を見てパケットを複製するかどうかを判断する．複製しない場合は，通常の C 素子と同じ振る舞いをする．複製する場合は，CP 信号を通常の出力分に加え N-1 回出力させる．複製用 CP 信号の出力時，内部に置かれたカウンタが作動しその値が複製パケットの識別情報 (ID_CCN) としてパケットに付加される．CCN を構成する各要素の詳細な働きを以下に示す．

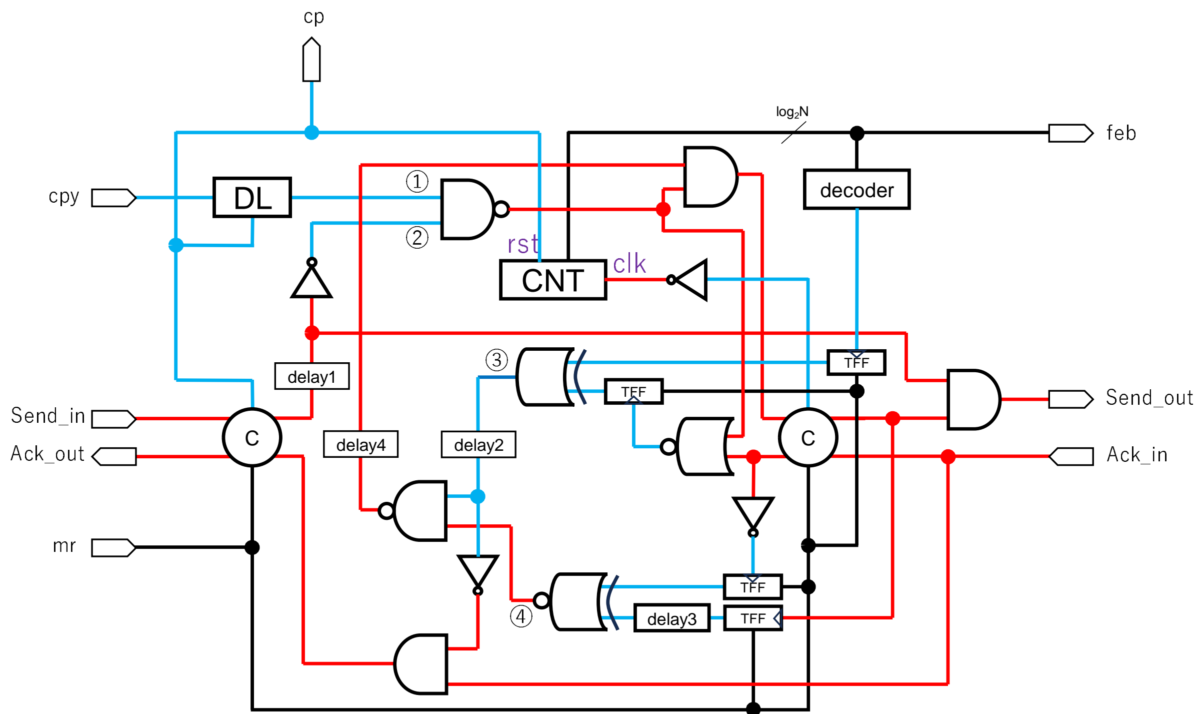



図 3.8　CCN 回路構成

- CNT

  左 C 素子の CP 信号 (rst) の立ち上がりで出力 (feb) を 0 に初期化し，右 C 素子の CP 信号 (clk) の立上がりで feb をインクリメントする．

- decoder

  feb が 3'b111(3bit カウンタの場合) の時は 1 を出力し，それ以外なら 0 を出力する．

- delay1 ～ 4

  後述する制約を守るために配置する．

decoder にて，3'b011 → 3'b100 や 3'b101 → 3'b110 のような複数桁の値が変化するようなカウンタ値更新時にグリッチが出てしまう．TFF の入力信号としてこの Decoder 出力を使用しているため，このままではそれ以降の回路で不具合が生じることが予想された．そのためグリッチが出ないよう図 3.9 のように回路に工夫を施す．

また，CCN の正常な動作のためには以下の制約を守る必要がある．

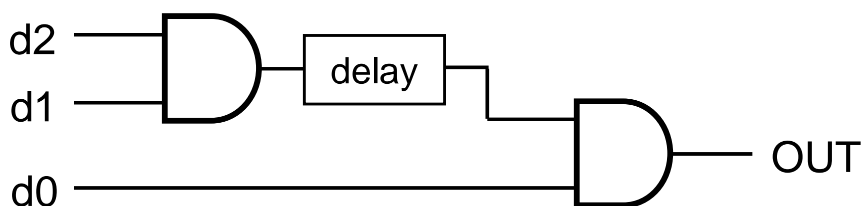



図 3.9　3bit 入力 decoder(グリッチ防止回路構成)

- ②よりも①が先に NAND に到達する必要があるため遅延回路 delay1 を挿入する
- ③が立ち上がるよりも先に④が立ち下がる，かつ④が立ち上がるよりも先に③が立ち下がる必要があるので遅延回路 delay2，delay3 を挿入する
- 右 C 素子に send 信号が入力され CP 信号が立ち上がる前に，次の send 信号が入力されてはいけないので遅延回路 delay4 を挿入する

### 3.5.4　vFP 回路構成

vFP は内部にベクトル・レジスタ，ALU Array を有することを大きな特徴としたベクトル・行列演算機構である．vFP 回路構成を図 3.10，パケットフォーマットを図 3.11 に示す．

ベクトル・レジスタは複数個の RAM で実現し，その各 RAM から 2 ベクトルを読み出すことで一度に行列単位で 2 オペランドを読み出し，ALU Array で演算し，その演算結果をベクトルレジスタに書き込む仕組みとした．vFP にパケットが到達するとパケットが保持するポインタ情報がベクトル・レジスタにセットされ，その情報をもとにオペランドを読み出し，ALU Array に渡す．ただし，第一オペランドは，行列積を計算する場合，パケット識別子の値を見てその値に応じたベクトルが選択される．このパケット識別子は，COPY N にてパケットの複製がされた際，その各パケットに割り振られるものであるが，この値が何番目の RAM を対象としているのかつまり，行列の何行目を対象としているのかを指す．第二オペランドは行列命令の種類によってそれを転置したものや Scalar をベクトルの要素数分連結したものにもなる．

## 3.5 提案回路構成

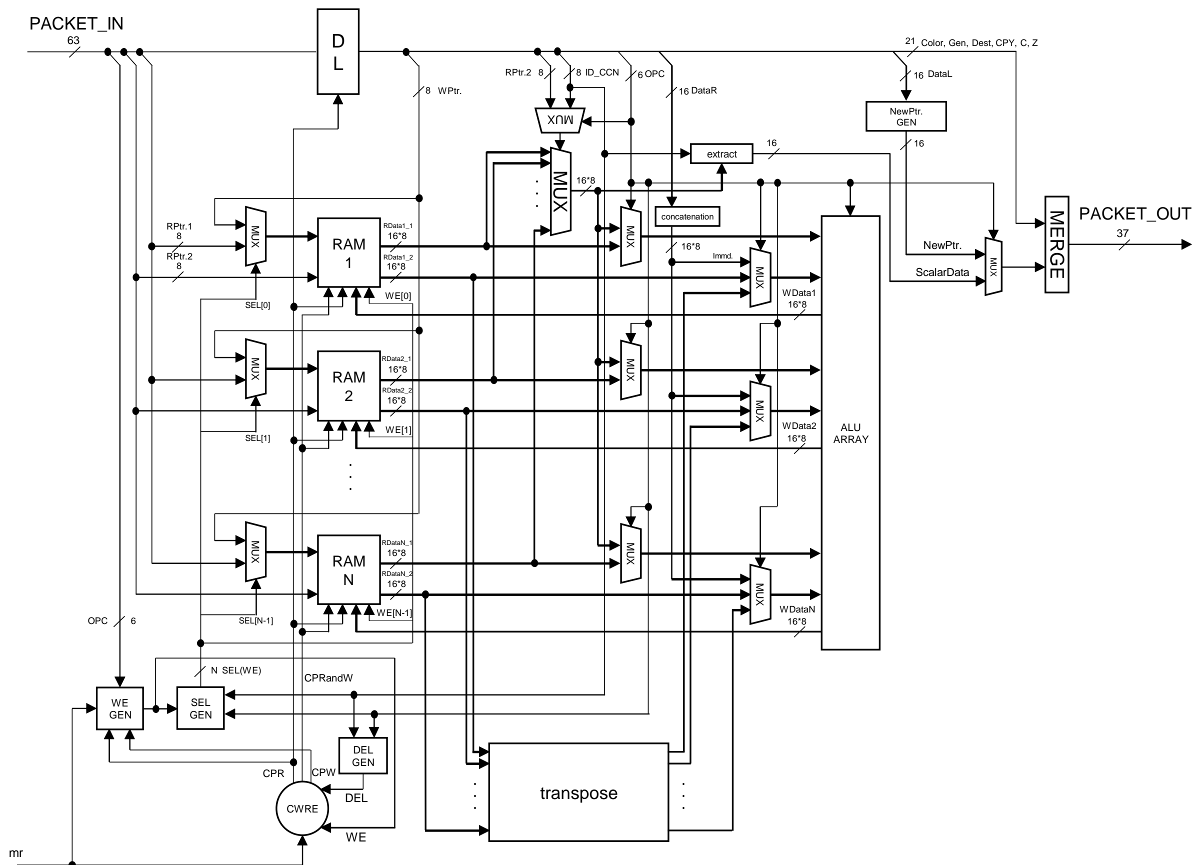



図 3.10　vFP 回路構成

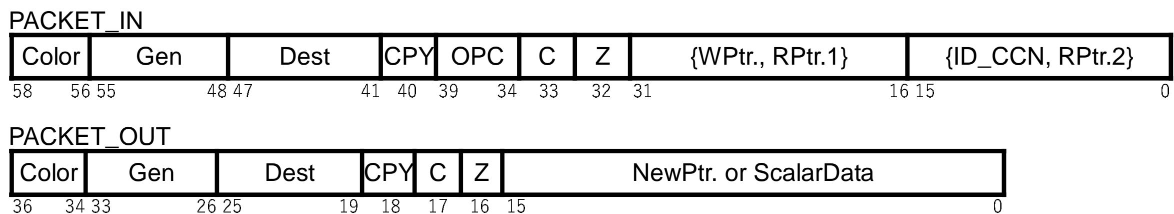



図 3.11　vFP パケットフォーマット

ALU Array は行列単位での演算ができるように内部に二次元に多段に連結した ALU を備えており，行列の要素一つ一つに対して同時に処理できる．ALU Array の回路構成を図 3.12 に示す．

ALU Array を構成する各要素の詳細な働きを以下に示す．

- add 1 ～ N

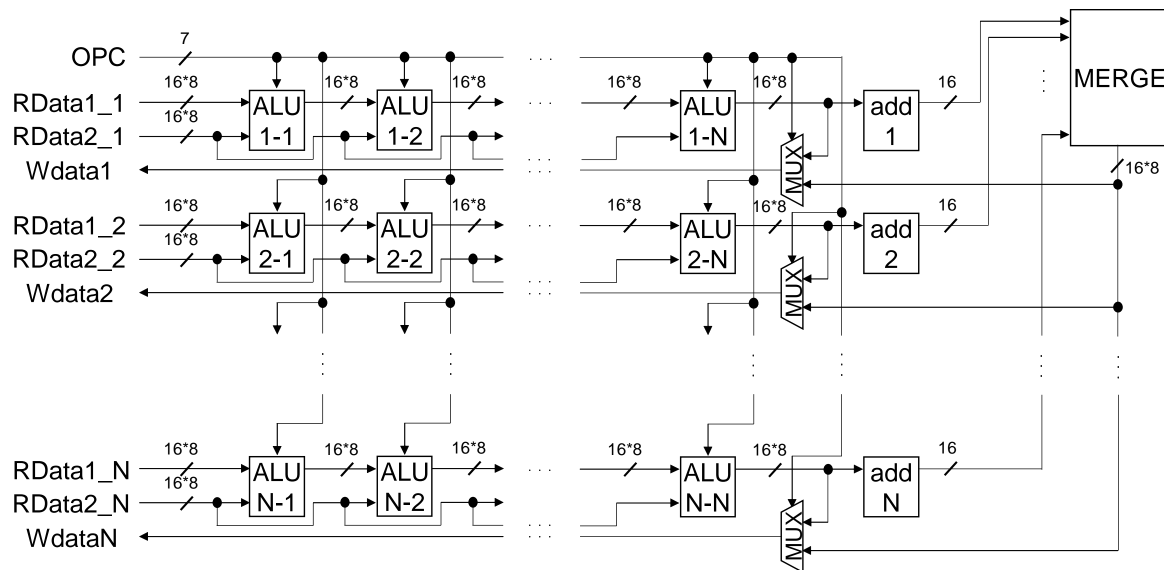



図 3.12　ALU Array 回路構成

ベクトルの内積を計算する．

- MERGE

  全ベクトルの内積を連結する．

ベクトル・レジスタを DDP に導入しただけでは，その構成上から一度のパケット通過でオペランドの読み出しに加え，演算結果の書き込みまでを行うことができなかった．そのため，今回はパケット消去機能付き読出・書込信号生成 C 素子 (C Write after Read & Erase: CWRE) を用意した．CWRE の回路構成を図 3.13 に示す．

CWRE は，通常 (ベクトル・レジスタからのオペランド読み出し用) の CP 信号の出力に加え，演算結果書き込み用の CP 信号も出力できる．ベクトル・レジスタへの書き込みを行わない命令を実行する際は通常の C 素子と同様の振る舞いをする．COPY N で複製したパケットは vFP 以降必要ないため，それらを消去するための機能も有している．制約として，WE が決定するまでに Send 信号の入力があるとグリッチが出てしまうため，遅延回路 delay1 を挿入する．

その他の vFP を構成する各要素の詳細な働きを以下に示す．

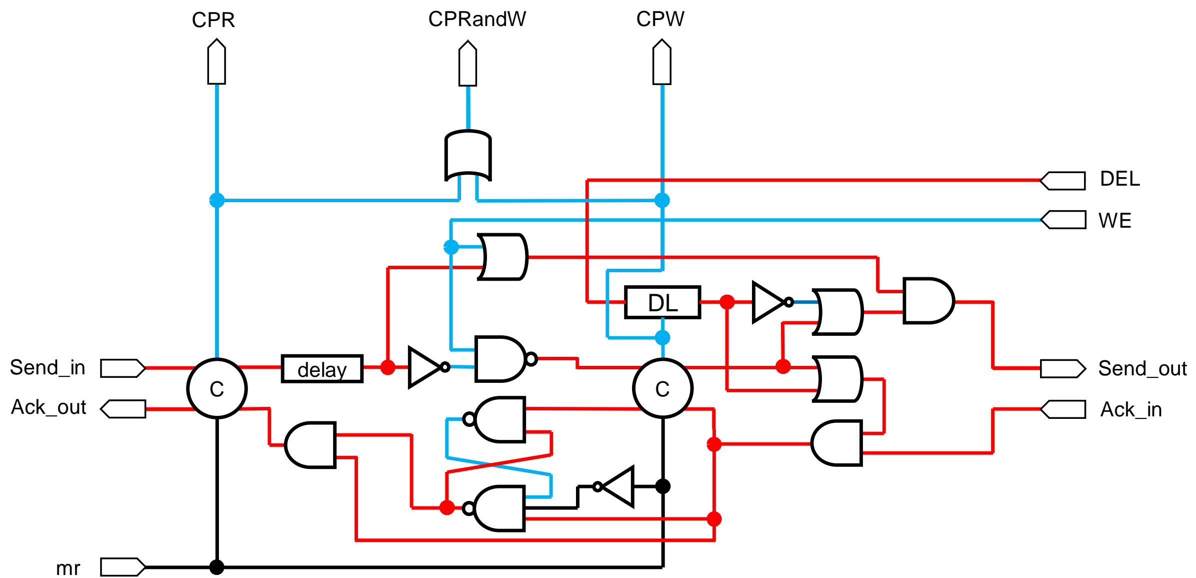



図 3.13　CWRE 回路構成

- transpose

  第二オペランド (行列) を転置する．

- extract

  読み出した対象のベクトルから ID_CCN をもとに 1 要素 (スカラー) を抜き出す．

- concatenation

  パケットが保持する DataR(即値) をベクトルの要素数分連結する．

- NewPtr. GEN

  書き込み用ポインタを読み込み用ポインタに変換する．DataL の下位 8bit に割り当てられている読み込み用ポインタに対して，上位 8bit に割り当てられている書き込み用ポインタで上書きし，空いた上位 8bit は 0 で埋める．

- DEL GEN

  パケットが COPY N で複製されているかどうかを OPC をもとに判定し，さらに，複製されているのであればそれが削除対象 (ID_CCN=1 ～ $N-1$) であるかどうかも判定し，その判定結果を反映した Delete 信号 (DEL) を生成する．

- WE GEN

演算結果の書き込みを必要とする命令であるかどうかを OPC をもとに判定し，演算結果の書き込みを必要とする命令であれば，CPR が立ち下がった時に Write Enabel 信号 (WE) を 1 にする．CPW が立ち下がった時に 0 に戻す．

- SEL GEN

  演算結果の書き込みを必要とする命令であるかどうかを OPC をもとに判定し，演算結果の書き込みを必要とする命令であれば，そのパケットが何番目であるかどうかを ID_CCN をもとに判定し，その判定結果を反映した信号 (SEL) を生成する．SEL は RAM の Ptr. 用ポートへの入力値を決めるマルチプレクサの Select 信号として使用される．また，RAM の Write Eneble 信号としても使用される．ただし，WE GEN からの WE が 0 である場合，SEL は機能しない (0 のまま).

### 3.5.5　BUF 回路構成

BUF は RAM と C 素子を主軸として構成される機構である．BUF の回路構成を図 3.14，パケットフォーマットを図 3.15 に示す．

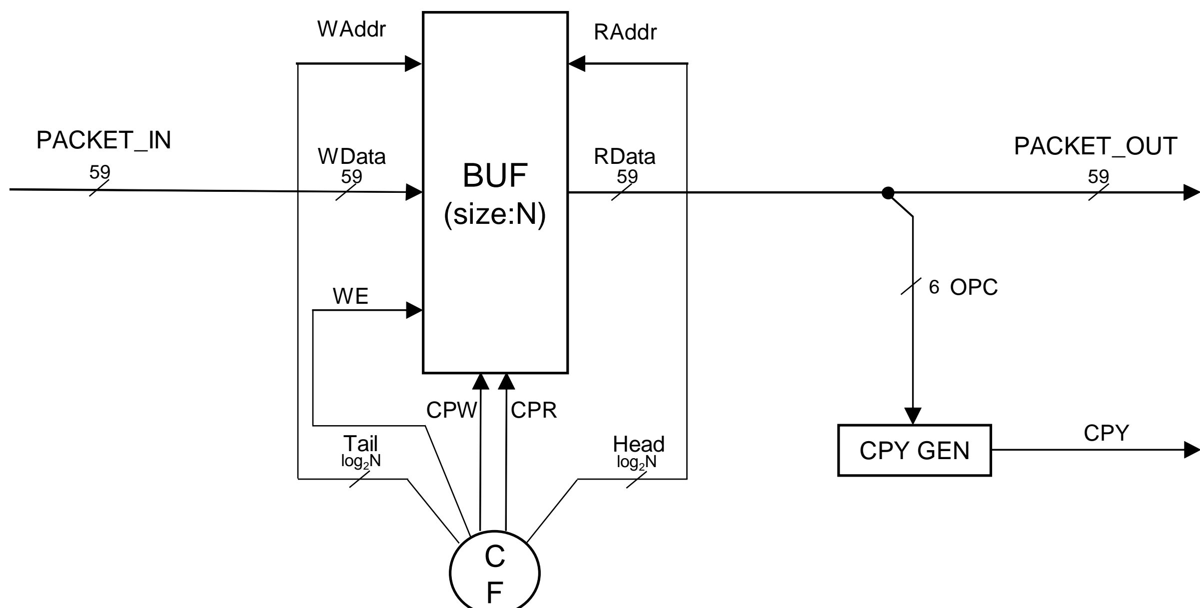



図 3.14　BUF 回路構成

## 3.5 提案回路構成

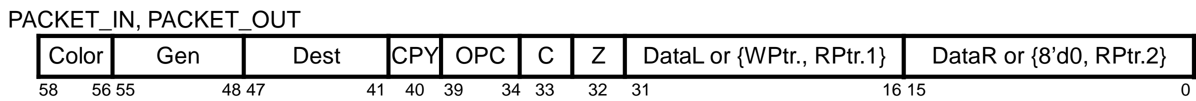



図 3.15　BUF パケットフォーマット

バッファー構成用 C 素子 (C FIFO: CF) は，BUF を構成する RAM に必要な情報 (Head，Tail，読み出し用 CP 信号，書き込み用 CP 信号，Write Enable 信号) を提供する．CF の回路構成を図 3.16 に示す．

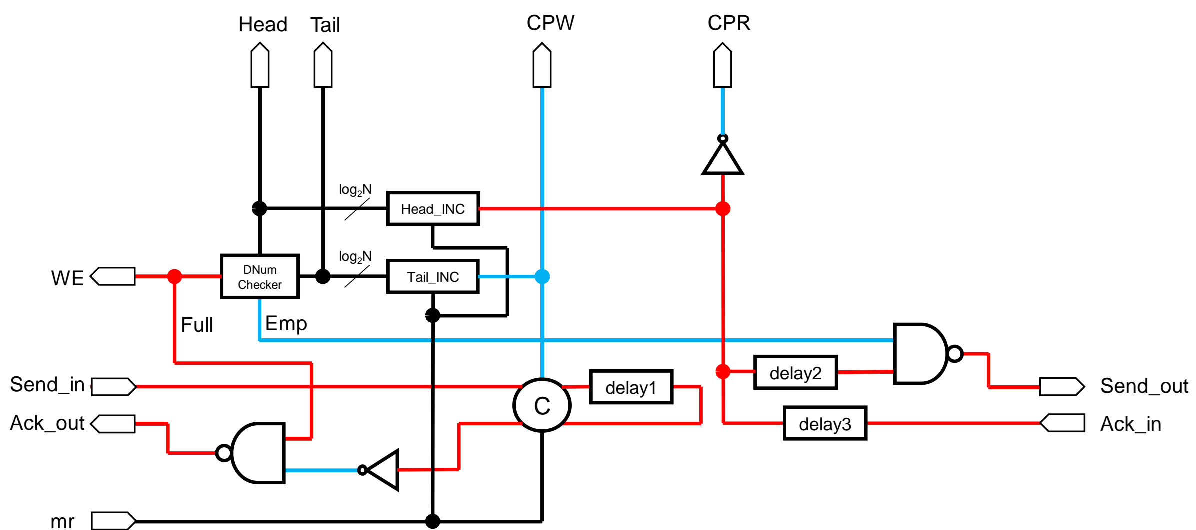



図 3.16　CF 回路構成

メモリに空きが無い場合，前段の C 素子への Ack 信号を返さず，メモリ内に有効なデータが無い場合，後段の C 素子への Send 信号を送らないよう動作する．CF を構成する各要素の詳細な働きを以下に示す．

- Head INC

  パケット読み出しアドレスである Head を出力する．後段機構へデータが送信されると Head の値をインクリメントする．

- Tail INC

  パケット書き込みアドレスである Tail を出力する．前段機構からデータが到着すると

Tail の値をインクリメントする．

- DNum Checker

  Head と Tail の値をもとに，Full 信号，Emp 信号を出力する．

  「Tail = Head-1」の時，「Full = 0」(空き無)，それ以外の時，「Full = 1」(空き有).

  「Tail == Head」の時，「Emp = 0」(有効パケット無)，それ以外の時，「Emp = 1」(有効パケット有).

- delay 1 ∼ 3

  後述する制約を守るために配置する．

また，CF の正常な動作のためには以下の制約を守る必要がある．

- tail がインクリメントされる前に CPW をトリガとして RAM へパケットを書き込む
- head がインクリメントされる前に CPR をトリガとして RAM からパケットを読み出す
- Full 信号が確定してから Ack out が出力される必要があるので遅延回路 (delay1) を挿入する
- Emp 信号が確定してから Ack in が入力され，Send out を出力する必要があるので遅延回路 (delay2) を挿入する
- 後段の C 素子の CP 信号が立ち上がる前に Send out を出力してはいけないので遅延回路 (delay2) を挿入する
- RAM にパケットが完全に書き込まれてから，CPR を立ち上げ読み込む必要があるため，CPW の立ち上げと CPR の立ち上げの間隔を空けるための遅延回路 delay3 を挿入する

## 3.6　結言

本章では，DDP の CC 化のための方策を述べた後，提案命令セットアーキテクチャ，提案アーキテクチャ構成を述べ，それを実現するための詳細な提案回路構成を示した．

具体的には，DDP の CC 化のために DDP 内部に CC の Little µEngine，Big µEngine に相当する実行系を導入する方法について検討した．そして，従来命令セットを実行制御できる Little µEngine と提案高機能命令セットを実行制御できる Big µEngine の導入を実現する並列パイプラン構成を含む提案アーキテクチャ構成を示し，それに準じて必要となる再構成および新規追加するステージロジック，C 素子回路について詳細に述べるまでを行った．

Big µEngine を構成するベクトル・行列演算機構 vFP は，現状，一部の行列計算命令のみの実行を想定しているため，行列 × 行列 の演算しか実行できない．今後，ベクトル・行列計算を含む高機能命令セットの充実や，それに準じて ベクトル × 行列 等の演算も可能とする回路構成を考案するなど改善の余地がある．

次章では，従来 DDP と本章で述べた提案 DDP の FPGA 回路の設計・実装を行い，性能および消費電力の観点からの評価結果について述べる．

# 第 4 章

# 設計・評価

## 4.1　緒言

これより，第 3 章にて述べた提案 DDP のエッジ・デバイス用アーキテクチャとしての有効性を測るため従来 DDP と比較し評価を行う．

本章では，Xilinx 社が提供する FPGA 向け設計環境 Vivado を用いて設計・実装した従来 DDP と提案 DDP を対象として，論理シミュレーション結果や使用した回路資源データをもとに性能および消費電力の観点から評価した結果を述べる．

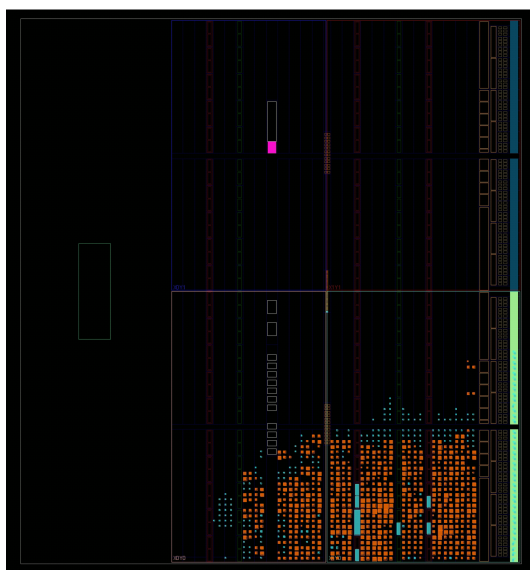
## 4.2　FPGA 回路の設計・実装

提案 DDP を設計するにあたり，新たに Assignment Pointer Table，バッファー，ベクトル・レジスタが必要となるため，それらを構成するメモリを Vivado の機能である IP Customize により追加実装する．今回，想定する行列サイズを $8 \times 8$ として評価を行うため，ベクトル・レジスタを構成する RAM は 8 個とする．それに伴い，COPY N ではパケットを 7 個複製するよう構成し，BUF を構成する RAM のサイズも 8 とする．追加実装するメモリについて表 4.1 に示す．また，EBR を追加で格納できるよう PS1 内の Program Storage のデータ幅を 1bit 拡張する．

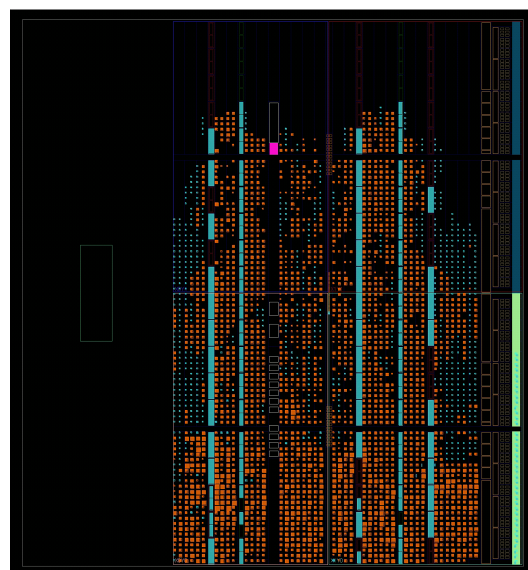
今回は DDP の実装先として，Xilinx 社製 Zynq-7010 搭載の Zybo Z7-10 開発ボードを想定しており，それに対して配置配線を行った．従来 DDP と提案 DDP のレイアウトをそれぞれ図 4.1(a)，図 4.1(b) に示す．

表 4.1　追加実装するメモリ

| memory name | memory size | memory type |
|---|---|---|
| Assignment Pointer Table | $8bit \times 128words$ | Single Port ROM |
| Buffer | $59bit \times 8words$ | Simple Dual Port RAM |
| RAM for Vector Register | $16bit \times 256words$ | True Dual Port RAM |



(a) 従来DDPのレイアウト図



(b) 提案DDPのレイアウト図

図 4.1　従来 DDP と提案 DDP のレイアウト図 “Vivado の Device 画面より”

## 4.3　性能評価

設計回路の論理シミュレーションにより，従来 DDP と提案 DDP を対象にそれぞれで各行列命令 (スカラー倍，行列積，行列和) の処理時間を測定する．従来 DDP においては，行列計算を実行するために，表 4.2 に示す CMPGEN 命令を従来命令セットに新たに追加している．また，従来 DDP で行列計算を行うために使用したプログラムは付録 A に示す．

従来 DDP，提案 DDP による各行列命令の処理時間を表 4.3 に示す．

表 4.2　従来命令セットに追加する命令

| Instruction | OPC | Operation |
|---|---|---|
| CMPGEN | 010 100 | gen が右 Data と一致すると dest を+1 する命令 |

表 4.3　従来 DDP，提案 DDP による各行列命令の処理時間

| | 従来 DDP | 提案 DDP |
|---|---|---|
| スカラー倍 [µs] | 179.02 | 0.848 |
| 行列積 [µs] | 1696.37 | 1.13 |
| 行列和 [µs] | 185.15 | 0.848 |

提案 DDP による各行列命令の処理時間は従来 DDP のそれと比べ，スカラー倍は 211.15 倍，行列積は 1502.41 倍，行列和は 218.37 倍の速度向上を達成した．

## 4.4　回路規模・消費電力評価

従来 DDP，提案 DDP に使われる回路資源 (LUT 数，Register 数，BRAM サイズ) を測定し回路規模を示す．そして，消費電力と回路規模はおおよそ比例すると仮定して，それら回路規模をもとに消費電力を算出する．

従来 DDP，提案 DDP の回路規模を表 4.4 に示す．

表 4.4　従来 DDP，提案 DDP の回路規模

| | 従来 DDP | 提案 DDP |
|---|---|---|
| LUT [個] | 1613 | 8119 |
| Register [個] | 1735 | 2041 |
| BRAM [KB] | 11.25 | 162 |
| Total*1 | 3359.25 | 10322 |

次に，提案 DDP 内の Little µEngine，Big µEngine の回路規模を表 4.5 に示す．

表 4.5 Little µEngine，Big µEngine の回路規模

| | Little µEngine | Big µEngine |
|---|---|---|
| LUT [個] | 420 | 6257 |
| Register [個] | 97 | 129 |
| BRAM [KB] | 2.25 | 144 |
| Total[*1] | 519.25 | 6530 |

両 µEngine の消費電力 $P_E$ は Little µEngine，Big µEngine の稼働率をそれぞれ $\alpha$，$1-\alpha$，消費電力をそれぞれ $P_L$，$P_B$ で表すと，(4.1) 式で求められる．

$$P_E = \alpha \times P_L + (1-\alpha) \times P_B \qquad (0 \leq \alpha \leq 1) \tag{4.1}$$

消費電力と回路規模はおおよそ比例すると仮定すると，Little µEngine，Big µEngine の消費電力 $P_L$ と $P_B$ は、それぞれ回路資源 $R_L$ と $R_B$ で代替できるため，$P_E$ は (4.2) 式でも求められる．

$$P_E = \alpha \times R_L + (1-\alpha) \times R_B \tag{4.2}$$

表 4.5 より，$P_L = R_L = 519.25$，$P_B = R_B = 6530$ である．この時, $\alpha$ を 80% と仮定すると，$P_E = 0.8 \times 519.25 + 0.2 \times 6530 = 2136.80$．これは $\alpha$ を 0% つまり，Big µEngine のみを動作させた時 ($P_E = P_B = 6530$) と比べ，67.28% 電力を削減できることを示唆している．

また，従来 DDP の消費電力 $P_N$ は，表 4.4 より，$P_N = 3359.25$．提案 DDP の消費電力 $P_P$ は，表 4.4，表 4.5，$P_E$ より，$P_P = 10322 - (519.25 + 6530) + 2136.80 = 5409.55$．以上を表 4.6 にまとめた．

[*1] BRAM 1Byte ≒ LUT 1 個 ≒ Register 1 個 と仮定して換算

表 4.6 従来 DDP，提案 DDP の消費電力

| | 従来 DDP | 提案 DDP |
|---|---|---|
| 消費電力 [W] | 3359.25 | 5409.55 |

## 4.5 消費電力量評価

4.4 節で概算した $P_N$，$P_P$ は単位時間あたりの消費電力であるので，4.3 節で求めた各行列命令の実行時間をかけることで，各行列命令を実行した際にかかる消費電力量も概算し評価する．

表 4.3，表 4.6 に示す値をもとに概算した各消費電力量を表 4.7 に示す．

表 4.7 従来 DDP，提案 DDP で各行列命令を実行した際の消費電力量

| | 従来 DDP | 提案 DDP |
|---|---|---|
| スカラー倍 [Ws] | 0.601 | 0.00459 |
| 行列積 [Ws] | 5.70 | 0.00611 |
| 行列和 [Ws] | 0.622 | 0.00459 |
| Total | 6.92 | 0.0153 |

提案 DDP により各行列命令を実行した際の消費電力量は従来 DDP によるそれと比べ，スカラー倍は 131.12 倍，行列積は 932.97 倍，行列和は 135.61 倍もの消費電力量を削減できている．

## 4.6 考察

行列計算命令の内，行列積は，ベクトル・レジスタを構成する RAM の数が 1 つ増えるごとに「1 パケットの複製にかかる時間+vFP での処理時間」分処理時間が長くなってしまう．そのため，扱う行列サイズを大きくすることで性能を向上させようとしても，行列積計算時は，大幅に性能が低下してしまう可能性がある．また，用意できる RAM の

上限は今回で言えば Zynq-7010 にて使用できる RAM の数にあたる．扱う FPGA 開発ボードによって，行列サイズ，性能が制限されるため，それを考慮し実装先を選定する必要がある．回路構成によっても処理時間は大きく変化する．STP は 2.5 節で述べた動作を行うためには，C 素子間には遅延量が設定されており，パイプラインステージ間でのデータの受け渡しが終了するまで後段 C 素子への Send 信号の到達を遅延させる必要がある．今回はこの遅延量を十分に設定しているためその分処理時間が増えてしまっている．また，表 4.3 に示してあるように行列命令の処理速度を測定したが，付録 A に示してある処理時間の測定に使用したプログラムは逐次的なプログラムになっており，多重並列処理が可能である DDP の特性を十分に生かしきれていない．そのため，今回算出した処理時間は DDP の真の処理時間とは言えない．しかし，多重にプログラムを実行させる場合，多重の度合いにもよるが，現状よりも PS の容量を多く必要とするため，性能を測る上でも性能と回路規模のトレードオフを意識する必要がある．

本来，(4.1) 式で求められる消費電力の値は消費電力と回路規模はおおよそ比例するとした仮定のもと，(4.2) 式で算出している．例えば C 素子の遅延回路は，Vivado での回路設計時，Xilinx FPGA の LUT から成る 1 入力バッファを多段に連結させることで実現している．評価時，消費電力と回路規模はおおよそ比例すると仮定し消費電力を算出しているため，C 素子の遅延量が最適化されていないのであれば，これもまた真の消費電力を算出できたとは言えない．また，BRAM 1Byte 分が LUT，Register でいう 1 個分と仮定して換算しているため，本来の回路規模も算出できていない．正規の手法を用いた消費電力，回路規模の算出が今後の課題として残っている．

表 4.7 に示している通り，提案 DDP において各行列命令を実行した際の総消費電力量が従来 DDP におけるそれと比べ 45277.96% 少ないため，同一のバッテリを用いた場合，45277.96% 長寿命になることが予想される．

## 4.7　結言

本章では，Xilinx 社が提供する FPGA 向け設計環境 Vivado を用いて設計・実装した従来 DDP と提案 DDP を対象として，論理シミュレーション結果や使用した回路資源データをもとに性能および消費電力の観点から評価した結果を述べた．

今回は，従来 DDP および提案 DDP において各行列命令の処理時間を測定することで性能を見積もり，回路規模測定結果から消費電力を概算した．また，それらをもとに消費電力量も概算した．しかし，本研究では，回路設計・実装に重きを置いたため特に C 素子の遅延回路の最適化を行えていない．それによって余分な遅延量が設定されているのであれば，その分処理時間，そして回路規模に影響を及ぼしかねない．今後，真の性能・消費電力・消費電力量評価のために，回路全体の最適化が残されている．

次章では，本研究で提案した DDP の CC 化についてまとめ，今後の課題を述べ，本論文を総括する．

# 第 5 章

# 結論

近年，IoT の急速な応用の広まりに伴い，IoT デバイス数は増加傾向にあり，2025 年には世界の IoT デバイス数はおよそ 440 億台に到達すると予測されている．しかし，ほとんどの IoT デバイスのプロセッサの性能は低く，メモリ容量も小さいため IoT デバイス側では複雑な処理をさせることができずにクラウド側に送信している．その結果，クラウドへの処理負荷の増加が問題となっている．そこで，クラウドの負荷分散を実現するエッジ・コンピューティングが注目されているが，そんなクラウドの負荷分散を担うエッジ・デバイスにはいくつかの課題がある．本研究では，そのうち高性能，低消費電力，再構成の容易さ，低開発コストに着目した．

これまでエッジ・デバイス用マイクロプロセッサを実現する技術がいくつか提案されている．デバイスの高性能化と低消費電力化の両立を図ることを目的として，1 つの CPU に特性が異なる複数のコアを搭載するヘテロジニアス・マルチコア (heterogeneous multicore: HMC) がある．しかし，HMC では各コアでそれぞれ独立したキャッシュや分岐予測器を持つため，コア間でのコンテキストスイッチング時のオーバーヘッドが非常に大きくなる問題がある．そんな HMC の切り替えペナルティを軽減するために強連結ヘテロジニアス・コア (tightly-coupled heterogeneous core: TCHC) も提案されている．TCHC は HMC とは異なり，非対称な複数の実行系を単一のコア内に持つ．代表的なものとして，コンポジットコア (Composite Core: CC) がある．

自己タイミング型パイプライン機構 (Self-Timed Pipeline:STP) を用いて動作するデータ駆動型プロセッサ (Data-Driven Processor: DDP) (以降，単に DDP) は，その構造か

ら割り込み処理を行わず，多様なセンサ等から到着する異なる複数のストリームデータに対する多重並列処理が可能である高性能と，クロック信号を用いず，データ入力がトリガとなり処理が実行されるため，必要な時に必要な回路のみ動作するという省電力性も兼ね備えていることから，エッジ・デバイス用のアーキテクチャとして有望である．

これら技術はエッジ・デバイスの高性能化，低消費電力化が急務となっている近年において．この分野の技術向上に欠かせないものとなっている．そんなこれらアーキテクチャを組み合わせ，各アーキテクチャの利点を取り入れることができれば，エッジ・デバイスの更なる高性能化と低消費電力化につながるのではないかと考えられるため，その組み合わせ方法に関して検討することには意義がある．本研究では，DDP の CC 化を提案した．

TCHC である CC は単一のコア内に，in-order で省電力動作する Little μ Engine と，out-of-order で高性能に動作する Big μ Engine という実行系が備えられている．そのため，HMC と比べて細粒度で実行系の切替制御が可能である．この特性を従来 DDP に取り入れると，次の特性を持つアーキテクチャとなることが予想された．

- 【CC 特性】実行系の切り替えにより高性能かつ低消費電力動作が実現できる
- 【CC 特性】実行系切替え時のオーバーヘッドを相対的に低減できる
- 【DDP 特性】両実行系ともに多重並列実行が可能である
- 【DDP 特性】処理の実行にクロック信号を必要としない

このアーキテクチャ実現のために DDP 内部に CC の Little μEngine，Big μEngine に相当する実行系を導入する方法について検討した．

本研究では，近年注目されている深層ニューラルネットワーク等で多用されるベクトル・行列計算を含む高機能命令セットを定義し，従来命令セット，提案高機能命令セットを実行制御できるパイプライン機構群をそれぞれ Little μEngine，Big μEngine とした．具体的には，従来 DDP の演算機構 FP と既存のメモリアクセス機構 MA を，FP，MA を併せた Little μEngine と新規に導入する高機能命令用パケット複製機構 COPY N，ベ

クトル・行列演算機構 vFP を併せた Big µEngine を並列に配置した並列パイプライン構成に置換した．これには，各 µEngine での実行を切り替えるための EB および EM，Big µEngine での処理に時間がかかることを見越していつくかパケットを溜めるためのバッファーを含む．また，この DDP の CC 化に伴い，書き込み用ポインタを格納するための APT，CB にパケット削除機能を追加した CBE，BUF を構成するリングバッファーを実現する CF，COPY N を構成する N-1 回のパケット複製を実現する CCN，vFP を構成するベクトル・レジスタと ALU Array と一度のパケット通過でベクトルレジスタへの読み込み・書き込みと複製パケットの消去を行うことができる CWRE を導入した．

提案 DDP の評価を行うために，Xilinx 社が提供する FPGA 向け設計環境 Vivado を用いて，従来 DDP と提案 DDP の FPGA 回路の設計・実装を行い，性能および消費電力の観点からの評価を行った．設計回路の論理シミュレーションにより，従来 DDP と提案 DDP を対象にそれぞれで各行列命令 (今回は，スカラー倍，行列積，行列和) の処理時間を測定したところ，提案 DDP による各行列命令の処理時間は従来 DDP のそれと比べ，スカラー倍は 211.15 倍，行列積は 1502.41 倍，行列和は 218.37 倍の速度向上を達成した．また，消費電力と回路規模はおおよそ比例すると仮定すると，Little µEngine，Big µEngine の各消費電力はそれぞれの回路資源で代替できるため，両 µEngine に使われる回路資源 (LUT 数，Register 数，BRAM サイズ) を測定しそれを消費電力算出のために利用した．Little µEngine での稼働率を 80% と仮定すると，両 µEngine の消費電力は 2136.80 となった．これは Big µEngine のみを動作させた時の消費電力と比べ，67.28% 電力を削減できることを示唆していた．さらに，各行列命令の処理時間と消費電力をもとに消費電力量を概算したところ，提案 DDP により各行列命令を実行した際の消費電力量は従来 DDP によるそれと比べ，スカラー倍は 131.12 倍，行列積は 932.97 倍，行列和は 135.61 倍もの消費電力量が削減できることが確認できた．

以下に本研究においての今後の課題を示す．

- 行列命令の処理速度測定時に使用したプログラムは逐次的なプログラムになっており，

多重処理が可能である DDP の特性を十分に生かしきれていなかったため，今後 DDP に適したプログラムでの処理時間の測定も試みる．

- 消費電力の値を，消費電力と回路規模はおおよそ比例するとした仮定のもと算出している．また，BRAM 1Byte 分が LUT，Register でいう 1 個分と仮定して換算しているため，真の消費電力を算出できていない．正規の手法を用いることでより厳密な消費電力の算出を行う．
- 今回は回路設計に重きをおいたため，パイプラインステージ間でのデータの受け渡しのために設定されている C 素子間の遅延量を最適化できていない．最適化後，真の処理時間，回路資源を測定する．

今後，以上の課題を解決することで，提案 DDP の更なる性能の向上と消費電力，消費電力量の削減が見込まれる．

# 参考文献


[1] 総務省, “令和 5 年版情報通信白書 世界の IoT デバイス数の推移及び予測,” https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/r05/html/datashu.html#f00212, 2023 年 12 月 29 日閲覧.

[2] Tulika Mitra, “Heterogeneous Multi-core Architectures,” IPSJ Transactions on System LSI Design Methodology Vol.8, pp. 51–62, Aug. 2015.

[3] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, “Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction,” Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO), 2003.

[4] ARM Ltd., “big.LITTLE Technology: The Future of Mobile Making very high performance available in a mobile envelope without sacrificing energy efficiency,” WHITE PAPER, 2013.

[5] P. Greenhalgh, “Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7,” WHITE PAPER, 2011.

[6] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, “Power-performance Modeling on Asymmetric Multi-cores,” Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 15:1–15:10, 2013.

[7] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, “Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE),” Proceedings of the 39th Annual International Symposium on Computer Architecture, pp. 213–224, 2012.

[8] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch,

and S. Mahlke, “Composite Cores: Pushing Heterogeneity Into a Core,” Proceedings of the 45th Annual International Symposium on Microarchitecture (MICRO), pp. 317–328, 2012.

[9] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, “Trace Based Phase Prediction for Tightly-coupled Heterogeneous Cores,” Proceedings of the 46th Annual International Symposium on Microarchitecture (MICRO), pp. 445–456, 2009.

[10] 塩谷亮太, 地代康政, 出岡宏二朗, 五島正裕, 安藤秀樹, “低電力モードを備えるプロセッサとモード切り替えアルゴリズムによる電力効率の向上,” 情報処理学会研究報告, Vol. 2017-ARC-226, No. 17, 2017.

[11] 三輪忍, 塩谷亮太, 佐々木広, “回路資源の投入により電力効率を改善するプロセッサ・アーキテクチャ,” 情報処理学会研究報告, Vol. 2014-ARC-212, No. 12, 2014.

[12] S. Navada, N. K. Choudhary, S. V. Wadhavkar, and E. Rotenberg, “A Unified View of Non-monotonic Core Selection and Application Steering in Heterogeneous Chip Multiprocessors,” Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, pp. 133–144, 2013.

[13] J. Sampson, M. Arora, N. Goulding-Hotta, G. Venkatesh, J. Babb, V. Bhatt, S. Swanson, and M. B. Taylor, “An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors,” 21th International Conference on Field Programmable Logic and Applications, pp. 24–29, 2011.

[14] 小島郁太郎, “Intel が PC 向け次期 MPU を 12 月発売構成チップレットの 4 分の 3 が TSMC 製 CPU コアが 3 種類に，AI ビジョン処理用のコアも内蔵,” 日経エレクトロニクス, pp. 34–37, Dec. 2023.

[15] W. Gomes, S. Morgan, B. Phelps, T. Wilson, and E. Hallnor, “Meteor Lake and Arrow Lake Intel Next-Gen 3D Client Architecture Platform with Foveros,” Hot Chips 34 Symposium (HCS), 2022.

[16] 古田雄大, “データ駆動型ヘテロジニアス・マルチコアにおける高性能コアの一検

討," 高知工科大学, 学士学位論文, 2020.

[17] 三宮秀次, 大森洋一, 酒居敬一, 岩田誠, "自己タイミング型パイプラインシステムの性能見積りモデル," 電子情報通信学会論文誌, Vol.J92-A, No.7, pp. 477–486, 2017.

# 謝辞

本研究に際し，日頃より懇切丁寧かつ熱心に御指導，御鞭撻を賜りました岩田誠教授に心より感謝の意を表すとともに，ここで厚く御礼申し上げます．御多忙な中，貴重なお時間を割いて相談に乗って頂いたおかげで，本研究を進めることができました．長いようで短かった 4 年間でしたが，研究以外のことでも大変お世話になりました。

本研究の論文の副査をお引き受け下さりました，松崎公紀教授，並びに吉田真一教授に心より感謝申し上げます．ご多忙な中お時間頂き，大変貴重なご意見を頂きました．

既にご卒業されましたが，時間を惜しむことなく回路設計のノウハウ等研究に関して相談に乗っていただいた長野寛司氏，井上聡氏，そして同期として一緒に切磋琢磨した岡野秀平氏に心より感謝致します．研究に行き詰った際には，親身にご助力頂いたおかげで研究を進めることができました。

そして，同研究室メンバとして日頃から暖かいご支援ご協力を頂きました，張震氏，Valeeprakhon Tamnuwat 氏，髙橋龍一氏，植元陸氏，坂口白磨氏，松坂拓海氏，市ノ木一希氏，伊藤雅俊氏，岡村健勝氏，山下拓巳氏，大﨑綾斗氏，奥平舜理氏，片岡拓心氏，門屋陽丈氏，山﨑浩正氏に心より感謝致します．引き続き研究室に在籍される皆様におかれましては，研究活動に際しこれから益々のご活躍とご健勝をお祈り致します．

最後になりましたが，日頃からご支援頂きました関係者の皆様に心より御礼申し上げます．

# 付録 A

# 従来 DDP 用行列計算プログラム

## A.1 プログラム共通

ソースコード A.1 DMEM

```
1    0000000000000001
2    0000000000000001
3    ...
64   0000000000000001 // 64個（1行列分）用意する
65   0000000000000010
66   0000000000000010
67   ...
128  0000000000000010 // 64個（1行列分）用意する
129  0000000000000011
130  0000000000000011
131  ...
192  0000000000000011 // 64個（1行列分）用意する
193  ...
```

## A.2 スカラー倍

ソースコード A.2 Packet

```
1  // color gen dest LR MF C Z Data
2  000_00000000_0000000_0_0_0_0_0000000000000000 // &A
3  000_00000000_0000001_0_0_0_0_0000000000000011 // &A'
```

ソースコード A.3 CMEM

## A.2 スカラー倍

```
1  // Data
2  0000000000000110
3  0000000000000110
4  0000000001000000
5  0000000001000000
6  0000000000000000
7  0000000000000000
8  0000000000000000
9  0000000000000000
10 0000000000000000
11 0000000000000001
12 0000000000000000
13 0000000000000101
14 0000000000000000
15 0000000000000000
16 0000000000000000
17 0000000000000000
18 0000000000000001
19 0000000000111111
20 0000000000000110
```

ソースコード A.4 PS1

```
1  // CPY OPC
2  0_001000 // SHL
3  0_001000 // SHL
4  0_101010 // ADDGEN
5  0_101010 // ADDGEN
6  1_000000 // NOP_ADD
7  0_111111 // ABSORB
8  0_101100 // DECGEN
9  0_010100 // CMPGEN
10 0_000000 //
11 0_000000 // ADD
12 0_100000 // LDM
13 0_000111 // MUL
14 0_100001 // STM
15 0_101100 // DECGEN
16 0_010100 // CMPGEN
```

```
17  0_000000 //
18  0_000000 // ADD
19  0_000010 // SUB
20  0_001001 // SHR
```

ソースコード A.5　PS2

```
1   // dest LR MF BR2
2   0000010_0_0_0 // to ADDGEN L -- input A
3   0000011_0_0_0 // to ADDGEN L -- input A'
4   0000100_0_0_0 // to NOP_ADD L
5   0001100_1_1_0 // to STM R
6   0000110_0_0_0 // to DECGEN L
7   0001010_0_0_0 // to LDM L
8   0000111_0_0_0 // to CMPGEN L
9   0001001_0_0_0 // to ADD L
10  0000101_0_0_0 // to ABSORB
11  0000100_0_0_0 // to NOP_ADD L
12  0001011_0_0_0 // to MUL L -- A * a = A'
13  0001100_0_1_0 // to STM L
14  0001101_0_0_0 // to DECGEN L
15  0001110_0_0_0 // to CMPGEN L
16  0010000_0_0_0 // to ADD L
17  0010001_0_0_0 // to SUB L
18  0001100_1_1_0 // to STM R
19  0010010_0_0_0 // to SHR L
20  0011011_0_0_1 // END
```

## A.3　行列積

ソースコード A.6　Packet

```
1 // color gen dest LR MF C Z Data
2   000_00000000_0000000_0_0_0_0_0000000000000011 // &A'
3   000_00000000_0000001_0_0_0_0_0000000000000101 // &A''
4   000_00000000_0000010_0_0_0_0_0000000000000001 // &B
```

## A.3 行列積

ソースコード A.7 CMEM

```
// Data
0000000000000110
0000000000000110
0000000000000110
0000000000001000
0000000000000000
0000000000000000
0000000000000001
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000001000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000111111
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000001
0000000000000000
```

```
38 0000000000000000
39 0000000000111111
40 0000000000000000
41 0000000000000000
42 0000000000001000
43 0000000000000000
44 0000000000000000
45 0000000000000000
46 0000000000000000
47 0000000000000000
48 0000000000001000
49 0000000000000000
50 0000000000000000
51 0000000000000000
52 0000000000001000
53 0000000000000000
54 0000000000000000
55 0000000000000000
56 0000000000111111
57 0000000000000000
58 0000000000000000
59 0000000000000000
60 0000000000001000
61 0000000000000000
62 0000000000000000
63 0000000000000111
64 0000000000000000
65 0000000000000000
66 0000000000000000
67 0000000000000000
68 0000000000000000
69 0000000000000110
```

ソースコード A.8 PS1

```
1 // CPY OPC
2 0_001000 // SHL
3 0_001000 // SHL
4 0_001000 // SHL
```

```
5   0_101010 // ADDGEN
6   1_000000 // NOP_ADD
7   0_111111 // ABSORB
8   0_000000 // ADD
9   0_101100 // DECGEN
10  0_010100 // CMPGEN
11  0_000000 //
12  0_010000 // BZ
13  0_000000 //
14  0_000010 // SUB
15  1_000000 // NOP_ADD
16  0_000000 //
17  0_010000 // BZ
18  0_000000 //
19  1_000100 // AND
20  0_000000 //
21  1_000000 // NOP_ADD
22  0_000000 //
23  1_000000 // NOP_ADD
24  0_000000 //
25  0_100000 // LDM
26  0_000111 // MUL
27  0_000000 // ADD
28  0_101100 // DECGEN
29  0_010100 // CMPGEN
30  0_000000 //
31  1_000000 // NOP_ADD
32  0_000000 //
33  0_100001 // STM
34  0_010000 // BZ
35  0_000000 //
36  0_000000 // ADD
37  0_010000 // BZ
38  0_000000 //
39  0_000010 // SUB
40  1_000000 // NOP_ADD
41  0_000000 //
42  0_101010 // ADDGEN
```

```
43  0_000111 // MUL
44  0_010000 // BZ
45  0_000000 //
46  0_010000 // BZ
47  0_000000 //
48  0_101010 // ADDGEN
49  1_000000 // NOP_ADD
50  0_000000 //
51  0_100000 // LDM
52  0_000000 // ADD
53  0_101100 // DECGEN
54  0_010100 // CMPGEN
55  0_000000 //
56  1_000010 // SUB
57  0_000000 //
58  0_010000 // BZ
59  0_000000 //
60  0_000010 // SUB
61  0_010000 // BZ
62  0_000000 //
63  1_000100 // AND
64  0_000000 //
65  1_000000 // NOP_ADD
66  0_000000 //
67  1_000000 // NOP_ADD
68  0_000000 //
69  0_001001 // SHR
```

ソースコード A.9　PS2

```
1  // dest LR MF BR1
2  0000011_0_0_0 // to ADDGEN L -- input A'
3  0100110_0_0_0 // to NOP_ADD L -- input A''
4  0101110_0_0_0 // to ADDGEN L -- input B
5  0000100_0_0_0 // to NOP_ADD L
6  0000110_0_0_0 // to ADD L
7  0010111_0_0_0 // to LDM L
8  0000111_0_0_0 // to DECGEN L
9  0001000_0_0_0 // to CMPGEN L
```

```
10 0000100_0_0_0 // to NOP_ADD L
11 0001010_0_1_0 // to BZ L
12 0001100_0_0_0 // to SUB L
13 0001101_0_0_0 // to NOP_ADD L
14 0000011_0_0_0 // to ADDGEN L
15 0001111_0_1_0 // to BZ L
16 0010001_0_0_0 // to AND L
17 0000011_0_0_0 // to ADDGEN L
18 0000101_0_0_0 // to ABSORB
19 0010011_0_0_0 // to NOP_ADD L
20 0010101_0_0_0 // to NOP_ADD L
21 0001111_1_1_0 // to BZ R
22 0100011_1_1_0 // to BZ R
23 0101100_1_1_0 // to BZ R
24 0111011_1_1_0 // to BZ R
25 0011000_0_1_0 // to MUL L
26 0011001_0_1_0 // to ADD L
27 0011010_0_0_0 // to DECGEN L
28 0011011_0_0_0 // to CMPGEN L
29 0011001_1_1_0 // to ADD R
30 0011101_0_0_0 // to NOP_ADD L
31 0011111_0_1_0 // to STM L
32 0101010_0_1_0 // to BZ L
33 0100000_0_1_0 // to BZ L
34 0100010_0_0_0 // to ADD L
35 0100011_0_1_0 // to BZ L
36 0011111_1_1_0 // to STM R
37 0100010_0_0_0 // to ADD L
38 0100101_0_0_0 // to SUB L
39 1000011_0_0_0 // to SHR L
40 0101000_0_0_0 // to ADDGEN L
41 0011111_1_1_0 // to STM R
42 0101001_0_0_0 // to MUL L
43 0011001_1_1_0 // to ADD R
44 0101000_0_0_0 // to ADDGEN L
45 0101100_0_1_0 // to BZ L
46 0101000_0_0_0 // to ADDGEN L
47 1000100_0_0_0 // to ABSORB
```

```
48    0101111_0_0_0 // to NOP_ADD L
49    0110001_0_0_0 // to LDM L
50    0110010_0_0_0 // to ADD L
51    0011000_1_1_0 // to MUL R
52    0110011_0_0_0 // to DECGEN L
53    0110100_0_0_0 // to CMPGEN L
54    0101111_0_0_0 // to NOP_ADD L
55    0110110_0_0_0 // to SUB L
56    0111000_0_1_0 // to BZ L
57    0111101_0_0_0 // to AND L
58    0101110_0_0_0 // to ADDGEN L
59    0111010_0_0_0 // to SUB L
60    0111011_0_1_0 // to BZ L
61    0101110_0_0_0 // to ADDGEN L
62    1000100_0_0_0 // to ABSORB
63    0111111_0_0_0 // to NOP_ADD L
64    1000001_0_0_0 // to NOP_ADD L
65    0111000_1_1_0 // to BZ R
66    0101010_1_1_0 // to BZ R
67    0001010_1_1_0 // to BZ R
68    0100000_1_1_0 // to BZ R
69    1000100_0_0_1 // END
```

## A.4　行列和

ソースコード A.10　Packet

```
1    // color gen dest LR MF C Z Data
2    000_00000000_0000000_0_0_0_0_0000000000000101 // &A''
3    000_00000000_0000010_0_0_0_0_0000000000000110 // &A'''
4    000_00000000_0000001_0_0_0_0_0000000000000100 // &C'
```

ソースコード A.11　CMEM

```
1    // Data
2    0000000000000110 // 6
3    0000000000000110 // 6
```

```
4  0000000000000110 // 6
5  0000000001000000 // 64
6  0000000000000000 // 0
7  0000000000000000 //
8  0000000000000000 // not use
9  0000000000000000 // 0
10 0000000000000000 //
11 0000000000000001 // 1
12 0000000000000000 // 0
13 0000000000000000 // not use
14 0000000000000000 // not use
15 0000000000000000 // not use
16 0000000000000000 // 0
17 0000000000000000 //
18 0000000000000001 // 1
19 0000000000111111 // 63
20 0000000001000000 // 64
21 0000000001000000 // 64
22 0000000000000000 // 0
23 0000000000000000 //
24 0000000000000000 // 0
25 0000000000000000 // not use
26 0000000000000000 // 0
27 0000000000000000 //
28 0000000000000001 // 1
29 0000000000000110 // 6
```

ソースコード A.12 PS1

```
1  // CPY OPC
2  0_001000 // SHL -- input A''
3  0_001000 // SHL -- input A'''
4  0_001000 // SHL -- input C'
5  0_101010 // ADDGEN
6  1_000000 // NOP_ADD
7  0_111111 // ABSORB
8  0_101100 // DECGEN
9  0_010100 // CMPGEN
10 0_000000 //
```

```
11 0_000000 // ADD
12 0_100000 // LDM
13 0_000000 // ADD -- A'' + C'
14 0_100001 // STM
15 0_101100 // DECGEN
16 0_010100 // CMPGEN
17 0_000000 //
18 0_000000 // ADD
19 0_000010 // SUB
20 0_101010 // ADDGEN
21 0_101010 // ADDGEN
22 1_000000 // NOP_ADD
23 0_000000 //
24 0_100000 // LDM
25 0_101100 // DECGEN
26 0_010100 // CMPGEN
27 0_000000 //
28 0_000000 // ADD
29 0_001001 // SHR
```

ソースコード A.13 PS2

```
1 // dest LR MF BR1
2 0000011_0_0_0 // to ADDGEN L -- input A''
3 0010010_0_0_0 // to ADDGEN L -- input A'''
4 0010011_0_0_0 // to ADDGEN L -- input C'
5 0000100_0_0_0 // to NOP_ADD L
6 0000110_0_0_0 // to DECGEN L
7 0001010_0_0_0 // to LDM L
8 0000111_0_0_0 // to CMPGEN L
9 0001001_0_0_0 // to ADD L
10 0000101_0_0_0 // to ABSORB
11 0000100_0_0_0 // to NOP_ADD L
12 0001011_0_1_0 // to ADD L ----------
13 0001100_0_1_0 // to STM L
14 0001101_0_0_0 // to DECGEN L
15 0001110_0_0_0 // to CMPGEN L
16 0010000_0_0_0 // to ADD L
17 0010001_0_0_0 // to SUB L
```

```
18  0001100_1_1_0 // to STM R
19  0011011_0_0_0 // to SHR L
20  0001100_1_1_0 // to STM R
21  0010100_0_0_0 // to NOP_ADD L
22  0010110_0_0_0 // to LDM L
23  0010111_0_0_0 // to DECGEN L
24  0001011_1_1_0 // to ADD R -----------
25  0011000_0_0_0 // to CMPGEN L
26  0011010_0_0_0 // to ADD L
27  0001010_0_0_0 // to ABSORB
28  0010100_0_0_0 // to NOP_ADD L
29  0011101_0_0_1 // END
```