2009年度　修士論文

部分マッチングを考慮し MISO(Multiple Input, Single Output)構造に対応したプロセッシングユニット最適化手法に関する研究

指導教員　戸川 望 教授

早稲田大学大学院　基幹理工学研究科
情報理工学専攻

5108B100–4

橋本 識弘

2010年2月5日

目 次

第1章

序論

1.1 本論文の背景と意義

近年，ディジタル信号処理の分野で，MPEG2 や H.264 をはじめとする高圧縮・高品質の動画像や音声データのような，情報量が多く符号化・復号化に処理時間を長く要するデータを扱うようになった．これらのデータを高速に処理するための，TI 社や Freescale 社などのアプリケーションに特化した専用プロセッサの需要が伸びており，携帯端末や車載システム，ディジタル家電など様々な場面において利用されている [3],[9]．この専用プロセッサは Intel 社や AMD 社などの汎用プロセッサと比較して小面積であり，特定のアプリケーションに対して高性能かつ低消費電力であるという利点がある．

専用処理プロセッサの実現方法は大きく分けて 2 つ挙げられる．1 つは処理ごとに IP を付加し，それを RISC や DSP で制御する手法である．この手法により高速実行が可能であるが，面積と消費電力は増大してしまう．もう 1 つは，DSP から直接操作できる専用演算器を付加することで，専用演算器の高い性能と DSP としての高い柔軟性を両立させる手法である．この専用演算器は複数の演算をまとめて実行するもので，面積の増大も最小限に抑えることができ，低消費電力を実現できる．しかし，アプリケーションごとにプロセッサや専用演算器の設計を手動で行うには時間的なコストが大きくかかってしまう．そのため，アプリケーションに特化した専用プロセッサを自動合成するシステムが必要とされている．本論文では，後者の実現方法によるプロセッサ合成を考える．

プロセッサに付加される専用演算器の生成に際し，満たすべき要件が大きく 2 つ挙げられる．1 つは生成できる専用演算器が MISO (Multiple Input, Single Output) 構造を持っていることである．MISO 構造とは，複数の入力と 1 つの出力を持つグラフ構造を取っており，グラフ中のノードは 2 つ以上の出力エッジを持つことができ，木構造と比較して複雑な構造を許容することができる．一般に，入力となるアプリケーションの CDFG は複雑な構造を持っており，木構造の専用演算器では CDFG とのマッチングが実行できない可能性があるためである．もう 1 つは，アプリケーション実行において専用演算器がより多く使用され，アプリケーション実行時間が削減されるために，専用演算器の構成とアプリケーションの CDFG のサブグラフが完全に一致している場合だけでなく，部分的に一致している場合 (以降，部分マッチングと呼ぶ) にも専用演算器で実行させることである．例えば，図 1.1 において，(a) に示した 6 入力 1 出力の専用演算器と (b) に示した DFG のマッチングを行う場合，点線で囲った演算ノードが部分的に一致しているため，専用演算器で実行される．このとき，専用演算器内で不要な演算に対して．0 もしくは 1 の定数を入力とし，演算をさせないことで，必要な出力結果を得ることができる．DFG に専用演算器を割り当てた結果を (c) に示す．

本研究室では，SIMD 型プロセッサコア向け HW/SW 協調合成システムである SPADES (System for Processor Architecture Design with Estimation - type SIMD) を構築中である．この自動合成システムは，[1] アプリケーション解析，[2] 並列化コンパイラ，[3] ハードウェア/ソフトウェア分割，[4] ハードウェア生成，[5] ソフトウェア生成からなる．入力として，C 言語で書かれたアプリケーショ

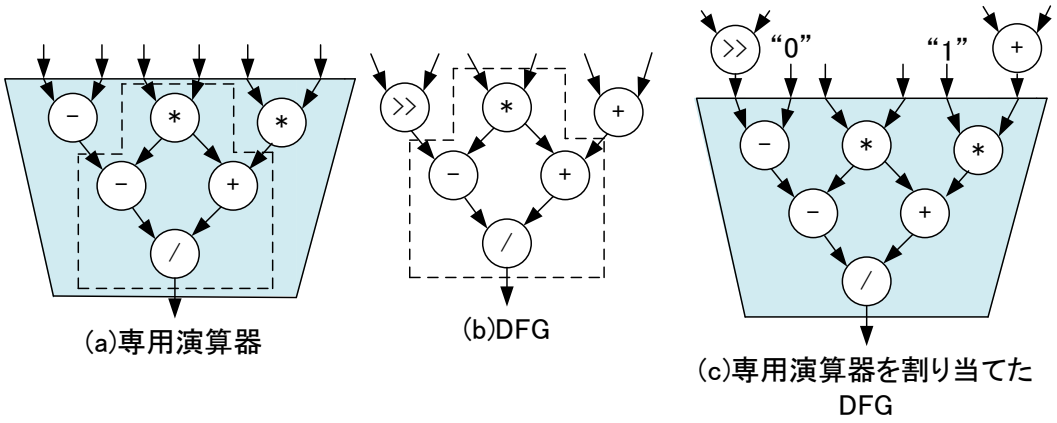

図 1.1: 部分マッチングの例

ンプログラム，アプリケーションデータおよびアプリケーション実行時間制約を与えることで，プロセッサのハードウェア記述，生成されたプロセッサで実行可能なアセンブリコードおよびコンパイラをはじめとしたソフトウェア環境を出力する．

SPADES は，画素処理の並列性を引き出すことのできる Packed SIMD 型演算を対象としており，SPADES を用いた研究として，SIMD 型演算器を持った画像処理向けプロセッサの自動合成についての文献 [5, 10, 12] が挙げられる．また，アプリケーションに特化した専用演算器であるプロセッシングユニットを自動合成することが可能であり，SPADES におけるプロセッシングユニットの自動合成に関する研究として，文献 [8] が挙げられる．しかし，文献 [8] の手法は部分マッチングを考慮しているが，木構造の専用演算器しか生成することができない．

これまでに研究されてきたアプリケーションに特化した専用演算器の合成手法や関連するツールとして，文献 [1, 2, 4, 7, 14, 15, 16] などが挙げられる．しかし，いずれの手法も専用演算器と CDFG のサブグラフが完全に一致したときにしかマッチングを実行していないため，部分マッチングが実行できない．文献 [17] ではカスタム命令セットを入力として，複数の専用演算を実行する専用演算器の合成手法を提案している．しかし，カスタム命令がすでに与えられているものとして扱っているため，設計者が手動でカスタム命令を設計する必要がある．また，あくまで複数のカスタム命令を実行できる専用演算器を生成するだけなので，単一のカスタム命令を実行する専用演算器のみを生成する場合と比べて，専用演算器で実行される回数が増えるわけではない．専用演算器が満たすべき 2 つの要件の両方を満たした専用演算器合成手法が，これまでに提案されていないのが現状である．

そこで本論文では，部分マッチングを考慮した MISO 構造を持つ専用演算器の合成手法を提案する．提案手法は，アプリケーションの CDFG，実行時間制約を入力として与え，実行時間制約を満たし面積が最小となる専用演算器の構成とアセンブリコードを出力する．専用演算器内の不要な演算に対して，0 もしくは 1 を入力とし演算を実行させないことで，専用演算器と CDFG (Control Data Flow Graph) のサブグラフが部分的に一致している場合にも，専用演算器で演算を実行させる部分マッチングを可能とした．

1.2 本論文の概要

本論文では，まず SIMD 型プロセッサコア向け HW/SW 協調合成システム SPADES について解説を行う．そして，部分マッチングを考慮し MISO 構造に対応したプロセッシングユニット最適化手法を提案し，解説する．その後，計算機実験結果から提案手法の有効性を評価する．また，既存のプロセッシングユニットの最適化手法について説明し，その他の専用演算器合成手法に関する研究を紹介する．本論文は，以下のように構成される．

第 2 章 **SIMD 型プロセッサコア向け HW/SW 協調合成システム：SPADES** では，まず対象となるプロセッサコアのアーキテクチャモデルおよびプロセッサコアに付加されるハードウェアユニットについて説明する．次に，SPADES が対象とする命令セットおよびプロセッサコアのパイプライン構成について解説する．その後，SPADES を構成するアプリケーション解析，並列化コンパイラ，ハードウェア/ソフトウェア分割，ハードウェア/ソフトウェア生成について解説する．

第 3 章**部分マッチングを考慮し MISO 構造に対応したプロセッシングユニット最適化手法**では，まずプロセッシングユニットの構成について定義する．次に，プロセッシングユニット最適化問題を定義し，プロセッシングユニットの対象となる命令について定義する．その後，プロセッシングユニットの最適化アルゴリズムを提案する．

第 4 章**計算機実験**では，第 3 章で提案したプロセッシングユニット最適化手法を計算機上に実装し，シミュレーションを行った結果を示す．そして，実験結果から，従来手法と比較して提案手法がより優れた解を出力していることを示し，本手法を評価する．

第 5 章**関連研究**では，プロセッシングユニット最適化に関する既存手法について説明する．その後，専用演算器合成手法に関するその他の研究について紹介する．

第 6 章**結論**では，本論文の内容を総括し，今後の指針について検討する．

第2章

SIMD型プロセッサコア向けHW/SW協調合成システム：SPADES

2.1 本章の概要

本章では，本研究室で構築中である SIMD 型プロセッサコア向け HW/SW 協調合成システムである SPADES について解説する．

SPADES は，(i)C 言語で書かれたアプリケーションプログラム，(ii) アプリケーションデータおよび (iii) アプリケーションの実行時間制約を入力とし，アドレッシングユニット，ハードウェアループユニット，フォワーディングユニットといった，プロセッサコアに付加されるハードウェアユニットやプロセッサの実行できる命令および内部レジスタの最適化をはじめとするプロセッサコア全体の HW/SW 協調合成最適化を行う．このとき，アプリケーション実行時間制約を満たし，プロセッサコアの面積が最小となるように探索を行うことで，アプリケーションに特化した (a) プロセッサコアの HDL 記述，(b) オブジェクトコードおよび (c) ソフトウェア環境を得る．

はじめに SPADES が対象とする SIMD 型プロセッサコアアーキテクチャモデルについて解説する．次にプロセッサコアに付加されるハードウェアユニットについて説明を行う．続いて，プロセッサコア内のリソースで実行可能な命令セットについて解説を行う．ただし，実際にはこの命令全てを含むわけではなく，アプリケーションに応じて最適な命令セットを生成する．

また，SPADES の構成要素であるアプリケーション解析，並列化コンパイラ，ハードウェア/ソフトウェア分割およびハードウェア/ソフトウェア生成について解説を行う．

2.2 SIMD 型プロセッサコアアーキテクチャモデル

本節では，現在本研究室で構築中である SIMD 型プロセッサコア向け HW/SW 協調合成システム SPADES のターゲットアーキテクチャモデルについて解説する．SPADES によって構成されるプロセッサコアのアーキテクチャを **SPADES プロセッサコア**と呼ぶ．SPADES プロセッサコアのアーキテクチャモデルは，ハードウェアユニットを制御するために必要なハードウェアから構成される**プロセッサカーネル**に，プロセッサ動作の補助を行う**ハードウェアユニット**を付加させた構成をとる．また，プロセッサカーネルは，VLIW 型のアーキテクチャを取ることで並列に命令を発行することができる．

プロセッサコアにおけるメモリのアーキテクチャは，データメモリと命令メモリを分離させたハーバードアーキテクチャをとり，外部メモリとして，1 つの命令メモリおよび X データメモリに加え Y データメモリを持つことが可能である．プロセッサコアは，命令メモリから命令を読み込み，データメモリからアプリケーションデータを読み込む．命令メモリから読み込まれた命令は，プロセッサカーネル内で処理内容がデコードされ，プロセッサカーネルは命令に従って，カーネル内部で実行可能な演算命令や制御命令はそれぞれ実行を行う．またカーネル内で実行不可能なハードウェア命令の場合は対象のハードウェアループユニットや SIMD 型演算ユニットに対して，各実行に必要なオペランドと制御信号をそれぞれのハードウェアユニットへ送出する．ハードウェアユニットは，カーネルから送られてきたデータに対して，演算処理や制御処理を行う．カーネル内の ALU や演算処理を行う SIMD 型演算器をはじめとするハードウェアユニットによって出力された結果を，内部レジスタもしくはデータメモリへ書き込むことで，アプリケーションプログラムを実行する．また，VLIW (Very Long Instruction Word) 命令方式[1]により実行を行うため，ハイパースカラ命令方式に比べプロセッサコアにスケジューラ機構を持つことなく，ハードウェア構成が簡単かつ小面積で実現できる．

SPADES システムでは，プロセッサコアに対してパイプライン段数，ハードウェアユニットの個数や種類，レジスタ数を変化させることにより，ターゲットアプリケーションに応じてアプリケーション実行時間制約を満たす中で面積が最小となる構成を出力する．

SPADES プロセッサコアの構成を図 2.1 に示す．

[1]依存関係のない複数の命令を 1 つの命令としてまとめて同時に実行することができる命令方式．同時実行される命令の数は常に一定に保たれ，規定の数に達しない場合は「NOP」命令でパディングされる．また，実行時に並列処理を判定するのではなく，コンパイラがコンパイルの時点で並列性を抽出し，命令を生成する．

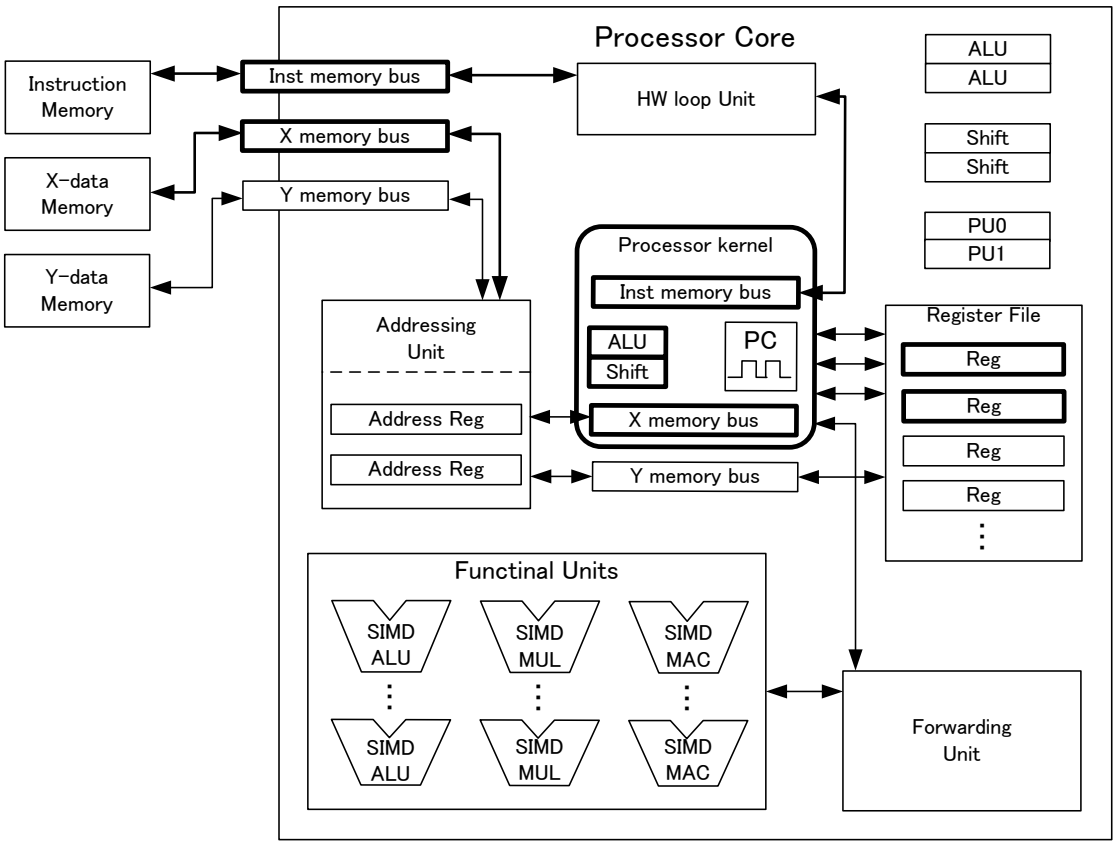

図 2.1: SPADES プロセッサコアの構成モデル

2.2.1 プロセッサカーネル

SPADES プロセッサコアがハードウェアユニットを制御するために必要なハードウェアから構成される最低限のハードウェア構成を**プロセッサカーネル**と呼ぶ．このプロセッサカーネルに，プロセッサの動作の補助を行うハードウェアユニットを付加させたものがプロセッサコアである．プロセッサカーネルの構成を以下の表 2.1 にまとめる．

表 2.1: プロセッサカーネルの構成

ハードウェア要素	必要な個数
バレルシフタ	1
ALU	1
レジスタ	2
X データメモリ用バス	1
命令データメモリ用バス	1

2.2.2 ハードウェアユニット

プロセッサカーネルには，以下のようなハードウェアユニットを付加することができる．

演算ユニット 演算ユニットとして，ALU，バレルシフタ，乗算器 (MUL)，乗加算器 (MAC) を取ることができる．演算ユニットは，ターゲットアプリケーションに応じて選択される．プロセッサカーネルに対し，付加される演算ユニットの種類および個数を変化させることができる．

SIMD 型演算ユニット プロセッサカーネルに対し，付加される SIMD 型演算器の個数を変化させることができ，複数の SIMD 型命令を実行させることが可能である．ある SIMD 型演算器が実行可能な命令のことを SIMD オプションと呼ぶ．1 つの SIMD 型演算器は複数の SIMD オプションを持つことが可能であり，持つ SIMD オプションの組み合わせによって SIMD 型演算器のハードウェア構成が決定される．各々の SIMD 型演算器が持つ SIMD オプションはアプリケーションにより決定される．SIMD 型演算器として，ALU(ADD，SUB)，乗算器，乗加算器，ビット拡張/縮小器，ビット交換器を取ることができる．SIMD 型演算器は，ハードウェアユニット生成手法を用いることでハードウェアコストおよび遅延時間の異なる複数の個数から選択可能である．

SIMD 型命令を使用することにより，基準ビット幅を分割し複数のデータを同時に処理することが可能である．基準ビット幅を 32bit とすると，基準ビット幅 32 ビットを 2 つに分割し，16 ビットのデータ 2 つを格納することができる．同様に，基準ビット幅 32 ビットを 4 つに分割し，8 ビットのデータ 4 つを格納することができる．このように，基準ビット幅に対する分割方法を複数用意することによって，ビット幅の異なる複数のデータに対して並列処理が可能となる．

Y-データバス (Y データメモリに対するバス) プロセッサカーネルに対し，Y データメモリおよび付随する Y バスを付加できる．Y データメモリを使用する場合，X データメモリおよび Y データメモリを用いることで，データメモリから並列に 2 個の値をロードおよび，データメモリへ 2 個の値をストアすることができる．Y データメモリのデータバス幅およびアドレスバス幅は，X データメモリと同一とする．

アドレッシングユニット アドレッシングユニットの構成は，アドレスレジスタ，インデックスレジスタおよびモジュロレジスタを持つ．アドレッシングモードとして，アドレスレジスタ値に対して何も演算処理を加えない (i)no operation，アドレスレジスタ値をインクリメントした値をアドレスレジスタ値とする (ii)post increment，アドレスレジスタ値をデクリメントした値をアドレスレジスタとする (iii)post decrement，アドレスレジスタ値にインデックスレジスタの値を加算した値をアドレスレジスタ値とする (iv)index add，アドレスレジスタ，インデックスレジスタおよびモジュロレジスタを用いてリングカウンタを実現し加算処理を行う (v)modulo add，アドレスレジスタの上位ビットと下位ビットを交換する (vi)bit reverse のアドレス演算を

実現できる．アドレッシングユニットの構成を表 2.2 にまとめる．ターゲットアプリケーションに応じて，プロセッサカーネルに付加されるアドレッシングユニットが実行できる演算の種類 ((i)–(vi))，アドレスレジスタ数，インデックスレジスタ数およびモジュロレジスタ数を変化させられる．アドレッシングユニット内のアドレスレジスタおよびインデックスレジスタ数は同一であり，X，Y データメモリで同数とする．モジュロレジスタは，X，Y データメモリ用に高々1 個ずつ用意される．

ハードウェアループユニット ハードウェアループユニットは，ネストレジスタによって実現され，その数を変化させることができる．ループ処理の終了をハードウェアユニットによって検出することで命令の読み込みアドレスを自動的に変更する．

フォワーディングユニット フォワーディングユニットが付加されたプロセッサコアでは，連続して実行される命令間に真のデータ依存があった場合に，先に演算が終了した命令の演算結果を，次の命令が実行される演算ユニットの入力にフォワーディングすることによって，NOP 命令の挿入あるいはパイプラインストールさせることなく命令を連続して実行することが可能となる．

プロセッシングユニット プロセッシングユニットは，複数の演算をまとめて実行することができる．プロセッシングユニットは，アプリケーションごとに自動で合成される．プロセッシングユニットについては，第 3 章で詳しく解説する．

レジスタファイル プロセッサカーネルには複数の汎用レジスタを付加させることができ，レジスタファイルに含まれるレジスタの数は不変である．アプリケーションに応じてこのレジスタ数を変化させることができる．

表 2.2: アドレッシングユニットの構成

	レジスタの種類			機能
アドレッシングモード	アドレスレジスタ $addr\_reg$	インデックスレジスタ idx_reg	モジュロレジスタ mod_reg	
(i)no operation	—	—	—	—
(ii)post increment	○	—	—	$++addr\_reg$
(iii)post decrement	○	—	—	$--addr\_reg$
(iv)index add	○	○	—	$addr\_reg + idx\_reg$
(v)modulo add	○	○	○	$mod_{mod\_reg}(addr\_reg + idx\_reg)$
(vi)bit reverse	○	—	○	$\overleftarrow{addr\_reg}$

2.2.3 命令セット

SPADES プロセッサが持つことが可能な命令セットは，(1) 基本命令，(2)SIMD 型命令および (3) 複合命令の 3 種類がある．プロセッサコアが持つことが可能な命令を表 2.3 から表 2.14 に示す．

命令の中には，生成するプロセッサコアがプロセッサとして動作するために最低限必要な命令がある．この命令を**必須命令**と呼ぶ．表中の*は必須命令を表す．

基本命令 SPADES において，プロセッサカーネルはハードウェアユニットを制御するために必要なハードウェアから構成される最低限のハードウェア構成を持つ．すなわちプロセッサカーネルは X データメモリ用バス，レジスタファイルを 2 つ，バレルシフタおよび ALU を 1 つずつ持つ．このことからプロセッサの必須命令として ALU を使用した**算術演算命令**(ADD，SUB，ADDI 他) および**論理演算命令**(AND，OR 他)，**メモリアクセス命令**(LD，ST 他) および**レジスタ転送命令**(MV，IMM 他) などが挙げられる．

一方，SPADES ではプロセッサの必須命令の他に，ハードウェアユニットを付加することによって必須命令以外の命令が実行可能となっている．プロセッサにアドレッシングユニットを追加することによって，メモリアクセスとともにアドレス値に処理を加える**ロード/ストア命令**(LDIX，STIX 他) が実行可能である．またハードウェアループユニットを追加することによって，ループ命令の繰り返し回数をハードウェアによって制御しパイプラインハザードを防ぐ特殊な**ループ命令**(LOOP，RPT 他) が実行可能である．プロセッサで実行可能な基本命令セットを表 2.3 から表 2.9 に示す．

SIMD 型演算命令 SIMD 型命令とは，1 個の b ビット演算ユニットを用いて n 個の b/n ビット演算を実現する命令である．SIMD 型命令を実行する演算ユニットを SIMD 型演算ユニットと呼ぶ．SIMD 型命令セットを持つマイクロプロセッサは，SIMD 型命令を利用することでアプリケーションデータを並列に処理することができる．

SIMD 型命令は，7 つのフィールドで構成される．SIMD 型命令のフィールド構成を式に示す．ここで，n 番目のフィールドを $\#n$ とする．

$$\begin{aligned} \mathrm{OPE} \_ \{1,2,4,...\}_{\#1} \ , \ \{h,l\}_{\#2} \ , \ \{s,u\}_{\#3} \ , \\ \{l,r\}_{\#4} \ , \ \{1-32\}_{\#5} \ , \ \{w,s\}_{\#6} \ , \ \{(position)\}_{\#7} \end{aligned} \tag{2.1}$$

式 2.1 の OPE は，SIMD 型命令のオペレーションコードを表す．また，各フィールド $\#n$ に格納された値はそれぞれ以下の操作を指定する．

#1 は，プロセッサの基準レジスタビット幅の中に梱包されているデータの数を表す．

#2 は，レジスタに格納されたデータのうち上下どちらかのデータに対してビット拡張付き演算を行うかを指定する．上位ビットを処理の対象とする場合は h，下位ビットを対象とする場

合は l を格納する．

#3 は，扱うデータの符号ビットの有無を表す．符号ビット付き演算の場合は s，符号ビット無し演算の場合は u を格納する．

#4 は，精度拡張・縮小のためのビットシフト操作におけるシフト方向を表す．左シフトの場合は l，右シフトの場合は r を格納する．

#5 は，#4 でシフト処理が指定された場合のシフト量を表す．本システムのターゲットアーキテクチャにおける基準ビット幅が 32 ビットであるため，シフト量は 1〜32 となる．

#6 は，処理結果の値がオーバーフローまたはアンダーフローした場合における飽和演算処理の有無を表す．飽和演算をする場合 (saturation) は s，飽和演算をしない場合 (wrap around) は w を格納する．

#7 は，演算場所を表す．演算場所とは，内部演算並列度を下げた演算ユニットを用いて演算をする命令に対して付くパラメータであり，レジスタのどの場所を演算するかを示すパラメータである．レジスタのビット幅を b とすると，$INST\_h$ という命令は上位 $b/2$ ビットを演算し，レジスタの上位 $b/2$ ビットに結果を格納するという命令である．また，$INST\_l$ という命令は下位 $b/2$ ビットを演算し，レジスタの下位 $b/2$ ビットに結果を格納するという命令である．レジスタのビット幅や梱包数によっては，演算場所のパラメータとして，hh や hhh などが付く場合もある．hh はレジスタの最上位から $b/4$ ビットを演算する命令であり，hhh は最上位から $b/8$ ビットを演算する命令である．

以上のように，Packed SIMD 型命令は通常の命令に比べて多くのフィールドを必要とする．ただし，上記 7 つのフィールドのうち省略可能なフィールドがある場合には，そのフィールドは省略する．例えば，梱包数 2，符号無し，右 4 ビットシフト，飽和演算ありの SIMD 型乗算命令は MUL_2_ur4s と表わされる．また，SIMD 型乗算命令の上位 $b/2$ ビットを演算する命令は MUL_2_ur4s_h と表される．

プロセッサで実行可能な SIMD 型命令セットを表 2.10 から表 2.14 に示す．

複合命令 基本命令および SIMD 型命令を複数個並列に実行する命令である．基本命令および SIMD 型命令のあらゆる組み合わせを複合命令として持つのではなく，アプリケーションに応じて複合命令となる命令の組み合わせを決定する．

表 2.3: 基本命令 1 (算術命令)

ID	命令	ニモニック	動作
101*	ADD	ADD R1, R2, R3	R3 ← R1 + R2
102*	SUB	SUB R1, R2, R3	R3 ← R1 – R2
103*	SRA	SRA R1, R2, R3	R3 ← R1 >> $R2_{0..x}$ (arithmetic)
104*	SRL	SRL R1, R2, R3	R3 ← R1 >> $R2_{0..x}$
105*	SLL	SLA R1, R2, R3	R3 ← R1 << $R2_{0..x}$
106*	AND	AND R1, R2, R3	R3 ← R1 & R2
107*	OR	OR R1, R2, R3	R3 ← R1 | R2
108*	XOR	XOR R1, R2, R3	R3 ← R1 ^ R2
109	MUL	MUL R1, R2, R3	R3 $\leftarrow_{n_d}$ $(R1 * R2)_{0..(n_d-1)}$
110	DIV	DIV R1, R2, R3	R3 ← R1 / R2
111	SLT	SLT R1, R2, R3	if (R1 < R2) R3 ← 1; else R3 ← 0
112	SEQ	SEQ R1, R2, R3	if (R1 == R2) R3 ← 1; else R3 ← 0
113	SNE	SNE R1, R2, R3	if (R1 != R2) R3 ← 1; else R3 ← 0
114	COM2	COM2 R1, R3	R3 ← 2's complement of R1
115	MAC	MAC R1, R2, R3	R3 ← R1 * R2 + R3
116*	INC	INC R1	R1++
117*	DEC	DEC R1	R1--
118*	ADDI	ADDI R1, R2, imm	R2 ← R1 + imm
119*	SUBI	SUBI R1, R2, imm	R2 ← R1 – imm
120*	SRAI	SRAI R1, R2, imm	R2 ← R1 >> $imm_{0..x}$ (arithmetic)
121*	SRLI	SRLI R1, R2, imm	R2 ← R1 >> $imm_{0..x}$
122*	SLLI	SLAI R1, R2, imm	R2 ← R1 << $imm_{0..x}$
123*	ANDI	ANDI R1, R2, imm	R2 ← R1 & imm
124*	ORI	ORI R1, R2, imm	R2 ← R1 | imm
125*	XORI	XORI R1, R2, imm	R2 ← R1 ^ imm
126	MULI	MULI R1, R2, imm	R3 $\leftarrow_{n_d}$ $(R1 * imm)_{0..(n_d-1)}$
127	DIVI	DIVI R1, R2, imm	R2 ← R1 / imm

x は，n_d を汎用レジスタのビット数としたとき，$x = \lg n_d - 1$ によって与える．

表 2.4: 基本命令 2 (ロード/ストア命令)

ID	命令	ニモニック	動作
201*	LDX	LDX R1, R2, imm	R2 $\leftarrow_{n_d}$ Mx[(R1 + imm)$_{0..15}$]
202*	LDY	LDY R1, R2, imm	R2 $\leftarrow_{n_d}$ My[(R1 + imm)$_{0..15}$]
203*	STX	STX R1, R2, imm	Mx[(R1 + imm)$_{0..15}$] $\leftarrow_{n_d}$ R2
204*	STY	STY R1, R2, imm	My[(R1 + imm)$_{0..15}$] $\leftarrow_{n_d}$ R2
205*	LDRX	LDRX R1, R2	R2 $\leftarrow_{n_d}$ Mx[R1$_{0..15}$]
206*	LDRY	LDRY R1, R2	R2 $\leftarrow_{n_d}$ My[R1$_{0..15}$]
207*	STRX	STRX R1, R2	Mx[R1$_{0..15}$] $\leftarrow_{n_d}$ R2
208*	STRY	STRY R1, R2	My[R1$_{0..15}$] $\leftarrow_{n_d}$ R2
209*	LDXI	LDXI R2, imm	R2 $\leftarrow_{n_d}$ Mx[imm$_{0..15}$]
210*	LDYI	LDYI R2, imm	R2 $\leftarrow_{n_d}$ My[imm$_{0..15}$]
211*	STXI	STXI R2, imm	Mx[imm$_{0..15}$] $\leftarrow_{n_d}$ R2
212*	STYI	STYI R2, imm	My[imm$_{0..15}$] $\leftarrow_{n_d}$ R2
213	LDIX	LDIX n1, R2, DPX3	
		LDIX 0, R2, DPX3	R2 $\leftarrow_{n_d}$ Mx[DPX3]
		LDIX 1, R2, DPX3	R2 $\leftarrow_{n_d}$ Mx[DPX3]; DPX3++
		LDIX 2, R2, DPX3	R2 $\leftarrow_{n_d}$ Mx[DPX3]; DPX3--
		LDIX 3, R2, DPX3	R2 $\leftarrow_{n_d}$ Mx[DPX3]; DPX3 $\leftarrow$ DPX3 + DNX3
		LDIX 4, R2, DPX3	R2 $\leftarrow_{n_d}$ Mx[DPX3];
			DPX3 $\leftarrow$ next_circular_addr(DPX3, DNX3, DMX)
		LDIX 5, R2, DPX3	R2 $\leftarrow_{n_d}$ Mx[bit_reverse(DPX3)];
			DPX3 $\leftarrow$ DPX3 + DNX3
214	LDIY	LDIY n1, R2, DPY3	
		LDIY 0, R2, DPY3	R2 $\leftarrow_{n_d}$ My[DPX3]
		LDIY 1, R2, DPY3	R2 $\leftarrow_{n_d}$ My[DPX3]; DPY3++
		LDIY 2, R2, DPY3	R2 $\leftarrow_{n_d}$ My[DPX3]; DPY3--
		LDIY 3, R2, DPY3	R2 $\leftarrow_{n_d}$ My[DPX3]; DPY3 $\leftarrow$ DPY3 + DNY3
		LDIY 4, R2, DPY3	R2 $\leftarrow_{n_d}$ My[DPX3];
			DPY3 $\leftarrow$ next_circular_addr(DPY3, DNY3, DMY)
		LDIY5, R2, DPY3	R2 $\leftarrow_{n_d}$ My[bit_reverse(DPY3)];
			DPY3 $\leftarrow$ DPY3 + DNY3

表 2.5: 基本命令 2 (ロード/ストア命令，続き)

ID	命令	ニモニック	動作
215	STIX	STIX n1, R2, DPX3	
		STIX 0, R2, DPX3	`Mx[DPX3]` $\leftarrow_{n_d}$ R2
		STIX 1, R2, DPX3	`Mx[DPX3]` $\leftarrow_{n_d}$ R2; DPX3++
		STIX 2, R2, DPX3	`Mx[DPX3]` $\leftarrow_{n_d}$ R2; DPX3--
		STIX 3, R2, DPX3	`Mx[DPX3]` $\leftarrow_{n_d}$ R2; DPX3 $\leftarrow$ DPX3 + DNX3
		STIX 4, R2, DPX3	`Mx[DPX3]` $\leftarrow_{n_d}$ R2;
			DPX3 $\leftarrow$ next_circular_addr(DPX3, DNX3, DMX)
		STIX 5, R2, DPX3	`Mx[bit_reverse(DPX3)]` $\leftarrow_{n_d}$ R2;
			DPX3 $\leftarrow$ DPX3 + DNX3
216	STIY	STIY n1, R2, DPY3	
		STIY 0, R2, DPY3	`My[DPY3]` $\leftarrow_{n_d}$ R2
		STIY 1, R2, DPY3	`My[DPY3]` $\leftarrow_{n_d}$ R2; DPY3++
		STIY 2, R2, DPY3	`My[DPY3]` $\leftarrow_{n_d}$ R2; DPY3--
		STIY 3, R2, DPY3	`My[DPY3]` $\leftarrow_{n_d}$ R2; DPY3 $\leftarrow$ DPY3 + DNY3
		STIY 4, R2, DPY3	`My[DPY3]` $\leftarrow_{n_d}$ R2;
			DPY3 $\leftarrow$ next_circular_addr(DPY3, DNY3, DMY)
		STIY 5, R2, DPY3	`My[bit_reverse(DPY3)]` $\leftarrow_{n_d}$ R2;
			DPY3 $\leftarrow$ DPY3 + DNY3
217(*)	MV	MV m1, R2, DPX3	
*		MV 0, R2, R3	R3 $\leftarrow$ R2
		MV 1, R2, DPX3	DPX3 $\leftarrow_{16}$ $R2_{0..15}$
		MV 2, R2, DPY3	DPY3 $\leftarrow_{16}$ $R2_{0..15}$
		MV 3, R2, DNX3	DNX3 $\leftarrow_{16}$ $R2_{0..15}$
		MV 4, R2, DNY3	DNY3 $\leftarrow_{16}$ $R2_{0..15}$
		MV 5, R2, DMX	DMX $\leftarrow_{16}$ $R2_{0..15}$
		MV 6, R2, DMY	DMY $\leftarrow_{16}$ $R2_{0..15}$
218(*)	IMM	IMM m1, DPX2, imm	
*		IMM 0, R2, imm	R2 $\leftarrow$ imm
		IMM 1, DPX2, imm	DPX2 $\leftarrow_{16}$ $imm_{0..15}$
		IMM 2, DPY2, imm	DPY2 $\leftarrow_{16}$ $imm_{0..15}$
		IMM 3, DNX2, imm	DNX2 $\leftarrow_{16}$ $imm_{0..15}$
		IMM 4, DNY2, imm	DNY2 $\leftarrow_{16}$ $imm_{0..15}$
		IMM 5, DMX, imm	DMX $\leftarrow_{16}$ $imm_{0..15}$
		IMM 6, DMY, imm	DMY $\leftarrow_{16}$ $imm_{0..15}$

表 2.6: 基本命令 3 (ジャンプ命令他)

ID	命令	ニモニック	動作
301*	BEQ	BEQ R1, R2, imm	if (R1 == R2) PC $\leftarrow_{n_{pc}}$ PC + $\mathrm{imm}_{0..(n_{pc}-1)}$
302*	BNE	BNE R1, R2, imm	if (R1 != R2) PC $\leftarrow_{n_{pc}}$ PC + $\mathrm{imm}_{0..(n_{pc}-1)}$
303*	BZ	BZ R1, imm	if (R1 == 0) PC $\leftarrow_{n_{pc}}$ PC + $\mathrm{imm}_{0..(n_{pc}-1)}$
304*	BNZ	BNZ R1, imm	if (R1 != 0) PC $\leftarrow_{n_{pc}}$ PC + $\mathrm{imm}_{0..(n_{pc}-1)}$
305*	JP	JP imm	PC $\leftarrow_{n_{pc}}$ PC + $\mathrm{imm}_{0..(n_{pc}-1)}$
306	LOOP	LOOP R1, imm	Loop imm instructions R1 times
307	RPT	RPT imm	Repeat next instruction imm times
308*	CALL	CALL imm	Call PC + $\mathrm{imm}_{0..(n_{pc}-1)}$
309*	RET	RET	Return
310*	NOP	NOP	No operation
311*	HLT	HLT	Halt

表 2.7: 特殊複合命令 (並列ロード/ストア命令)

ID	命令	ニモニック	動作
401	LDPX	LDPX n1, DPX1, R2, R3	LDIX n1, DPX1, R2; LDIX n1, DPY1, R3
402	STPX	STPX n1, DPX1, R2, R3	STIX n1, DPX1, R2; STIX n1, DPY1, R3

表 2.8: n の値

n	演算
0	no operation
1	post increment
2	post decrement
3	index add
4	modulo add
5	bit reverse

表 2.9: m の値

m	種別	m	種別
0	R	4	DNY
1	DPX	5	DMX
2	DPY	6	DMY
3	DNX	—	

表 2.10: SIMD 型加算命令

演算名	梱包数	ビット拡張	符号演算	飽和演算	シフト演算	ニモニック
ADD	1	(none)	signed	saturation	right	ADD_1_srXXs
					left	ADD_1_slXXs
				wrap around	right	ADD_1_srXXw
					left	ADD_1_slXXw
			unsigned	saturation	right	ADD_1_urXXs
					left	ADD_1_ulXXs
				wrap around	right	ADD_1_urXXw
					left	ADD_1_ulXXw
	2	(none)	signed	saturation	right	ADD_2_srXXs
					left	ADD_2_slXXs
				wrap around	right	ADD_2_srXXw
					left	ADD_2_slXXw
			unsigned	saturation	right	ADD_2_urXXs
					left	ADD_2_ulXXs
				wrap around	right	ADD_2_urXXw
					left	ADD_2_ulXXw
		higher	signed	(none)	(none)	ADD_2h_s
			unsigned	(none)	(none)	ADD_2h_u
		lower	signed	(none)	(none)	ADD_2l_s
			unsigned	(none)	(none)	ADD_2l_u
	4	(none)	signed	saturation	right	ADD_4_srXXs
					left	ADD_4_slXXs
				wrap around	right	ADD_4_srXXw
					left	ADD_4_slXXw
			unsigned	saturation	right	ADD_4_urXXs
					left	ADD_4_ulXXs
				wrap around	right	ADD_4_urXXw
					left	ADD_4_ulXXw
		higher	signed	(none)	(none)	ADD_4h_s
			unsigned	(none)	(none)	ADD_4h_u
		lower	signed	(none)	(none)	ADD_4l_s
			unsigned	(none)	(none)	ADD_4l_u

XX はシフト演算のビット数を表す.

表 2.11: SIMD 型減算命令

演算名	梱包数	ビット拡張	符号演算	飽和演算	シフト演算	ニモニック
SUB	1	(none)	signed	saturation	right	SUB_1_srXXs
					left	SUB_1_slXXs
				wrap around	right	SUB_1_srXXw
					left	SUB_1_slXXw
			unsigned	saturation	right	SUB_1_urXXs
					left	SUB_1_ulXXs
				wrap around	right	SUB_1_urXXw
					left	SUB_1_ulXXw
	2	(none)	signed	saturation	right	SUB_2_srXXs
					left	SUB_2_slXXs
				wrap around	right	SUB_2_srXXw
					left	SUB_2_slXXw
			unsigned	saturation	right	SUB_2_urXXs
					left	SUB_2_ulXXs
				wrap around	right	SUB_2_urXXw
					left	SUB_2_ulXXw
		higher	signed	(none)	(none)	SUB_2h_s
			unsigned	(none)	(none)	SUB_2h_u
		lower	signed	(none)	(none)	SUB_2l_s
			unsigned	(none)	(none)	SUB_2l_u
	4	(none)	signed	saturation	right	SUB_4_srXXs
					left	SUB_4_slXXs
				wrap around	right	SUB_4_srXXw
					left	SUB_4_slXXw
			unsigned	saturation	right	SUB_4_urXXs
					left	SUB_4_ulXXs
				wrap around	right	SUB_4_urXXw
					left	SUB_4_ulXXw
		higher	signed	(none)	(none)	SUB_4h_s
			unsigned	(none)	(none)	SUB_4h_u
		lower	signed	(none)	(none)	SUB_4l_s
			unsigned	(none)	(none)	SUB_4l_u

XX はシフト演算のビット数を表す.

表 2.12: SIMD 型乗算命令

演算名	梱包数	ビット拡張	符号演算	飽和演算	シフト演算	ニモニック
MUL	1	(none)	signed	saturation	right	MUL_1_srXXs
					left	MUL_1_slXXs
				wrap around	right	MUL_1_srXXw
					left	MUL_1_slXXw
			unsigned	saturation	right	MUL_1_urXXs
					left	MUL_1_ulXXs
				wrap around	right	MUL_1_urXXw
					left	MUL_1_ulXXw
	2	(none)	signed	saturation	right	MUL_2_srXXs
					left	MUL_2_slXXs
				wrap around	right	MUL_2_srXXw
					left	MUL_2_slXXw
			unsigned	saturation	right	MUL_2_urXXs
					left	MUL_2_ulXXs
				wrap around	right	MUL_2_urXXw
					left	MUL_2_ulXXw
		higher	signed	(none)	(none)	MUL_2h_s
			unsigned	(none)	(none)	MUL_2h_u
		lower	signed	(none)	(none)	MUL_2l_s
			unsigned	(none)	(none)	MUL_2l_u
	4	(none)	signed	saturation	right	MUL_4_srXXs
					left	MUL_4_slXXs
				wrap around	right	MUL_4_srXXw
					left	MUL_4_slXXw
			unsigned	saturation	right	MUL_4_urXXs
					left	MUL_4_ulXXs
				wrap around	right	MUL_4_urXXw
					left	MUL_4_ulXXw
		higher	signed	(none)	(none)	MUL_4h_s
			unsigned	(none)	(none)	MUL_4h_u
		lower	signed	(none)	(none)	MUL_4l_s
			unsigned	(none)	(none)	MUL_4l_u

XX はシフト演算のビット数を表す.

表 2.13: SIMD 型乗加算命令

演算名	梱包数	ビット拡張	符号演算	飽和演算	シフト演算	ニモニック
MAC	1	(none)	signed	saturation	right	MAC_1_srXXs
					left	MAC_1_slXXs
				wrap around	right	MAC_1_srXXw
					left	MAC_1_slXXw
			unsigned	saturation	right	MAC_1_urXXs
					left	MAC_1_ulXXs
				wrap around	right	MAC_1_urXXw
					left	MAC_1_ulXXw
	2	(none)	signed	saturation	right	MAC_2_srXXs
					left	MAC_2_slXXs
				wrap around	right	MAC_2_srXXw
					left	MAC_2_slXXw
			unsigned	saturation	right	MAC_2_urXXs
					left	MAC_2_ulXXs
				wrap around	right	MAC_2_urXXw
					left	MAC_2_ulXXw
		higher	signed	(none)	(none)	MAC_2h_s
			unsigned	(none)	(none)	MAC_2h_u
		lower	signed	(none)	(none)	MAC_2l_s
			unsigned	(none)	(none)	MAC_2l_u
	4	(none)	signed	saturation	right	MAC_4_srXXs
					left	MAC_4_slXXs
				wrap around	right	MAC_4_srXXw
					left	MAC_4_slXXw
			unsigned	saturation	right	MAC_4_urXXs
					left	MAC_4_ulXXs
				wrap around	right	MAC_4_urXXw
					left	MAC_4_ulXXw
		higher	signed	(none)	(none)	MAC_4h_s
			unsigned	(none)	(none)	MAC_4h_u
		lower	signed	(none)	(none)	MAC_4l_s
			unsigned	(none)	(none)	MAC_4l_u

XX はシフト演算のビット数を表す.

表 2.14: SIMD 型交換命令・拡張命令・縮小命令

演算名	梱包数	ビット拡張	符号演算	飽和演算	シフト演算	ニモニック
EXCH	2	—				EXCH_2
	4	—				EXCH_4
	8	—				EXCH_8
EXTD	2	higher	signed	—		EXTD_2h_s
			unsigned	—		EXTD_2h_u
		lower	signed	—		EXTD_2l_s
			unsigned	—		EXTD_2l_u
	4	higher	signed	—		EXTD_4h_s
			unsigned	—		EXTD_4h_u
		lower	signed	—		EXTD_4l_s
			unsigned	—		EXTD_4l_u
EXTR	2		signed	saturation	right	EXTR_2_srXXs
					left	EXTR_2_slXXs
				wrap around	right	EXTR_2_srXXw
					left	EXTR_2_slXXw
			unsigned	saturation	right	EXTR_2_urXXs
					left	EXTR_2_ulXXs
				wrap around	right	EXTR_2_urXXw
					left	EXTR_2_ulXXw
	4		signed	saturation	right	EXTR_4_srXXs
					left	EXTR_4_slXXs
				wrap around	right	EXTR_4_srXXw
					left	EXTR_4_slXXw
			unsigned	saturation	right	EXTR_4_urXXs
					left	EXTR_4_ulXXs
				wrap around	right	EXTR_4_urXXw
					left	EXTR_4_ulXXw

XX はシフト演算のビット数を表す.

2.2.4 パイプライン構成

SPADES プロセッサコアが取りうるパイプライン構成モデルについて説明する．SPADES プロセッサコアのパイプライン段数は可変であり，様々なパイプライン構成からアプリケーションに最適なパイプライン構成を選択できる．

SPADES プロセッサコアの構成モデルにおいて，パイプライン段数 3 段から 6 段までの構成を図 2.2 に示す．図 2.2 では，上段に演算命令と分岐命令の実行工程を，下段にロード/ストア命令の実行工程を示した．パイプライン段数 6 段以降の構成では，第 2 ステージまでと最後のステージでレジスタ書き込みを行う点はパイプライン段数 5 段の構成同様で，パイプライン段数が増えるごとに演算実行のステージが 1 段ずつ増える構成となる．

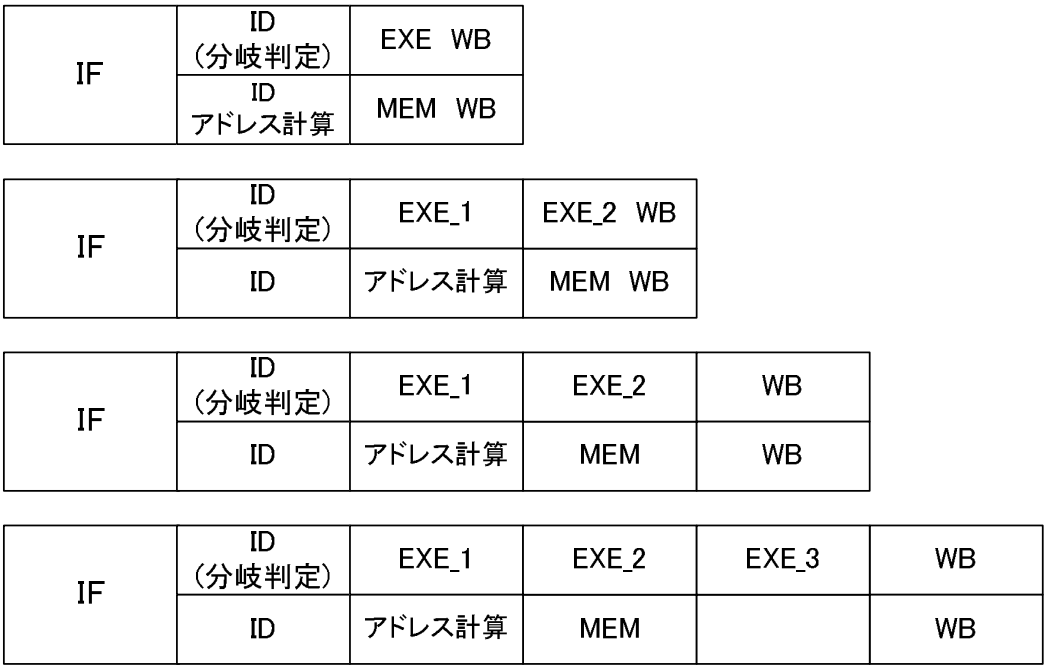

図 2.2: SPADES プロセッサコアのパイプライン構成モデル

2.3 ハードウェア/ソフトウェア協調合成システム

SPADES システムの構成図を図 2.3 に示す.

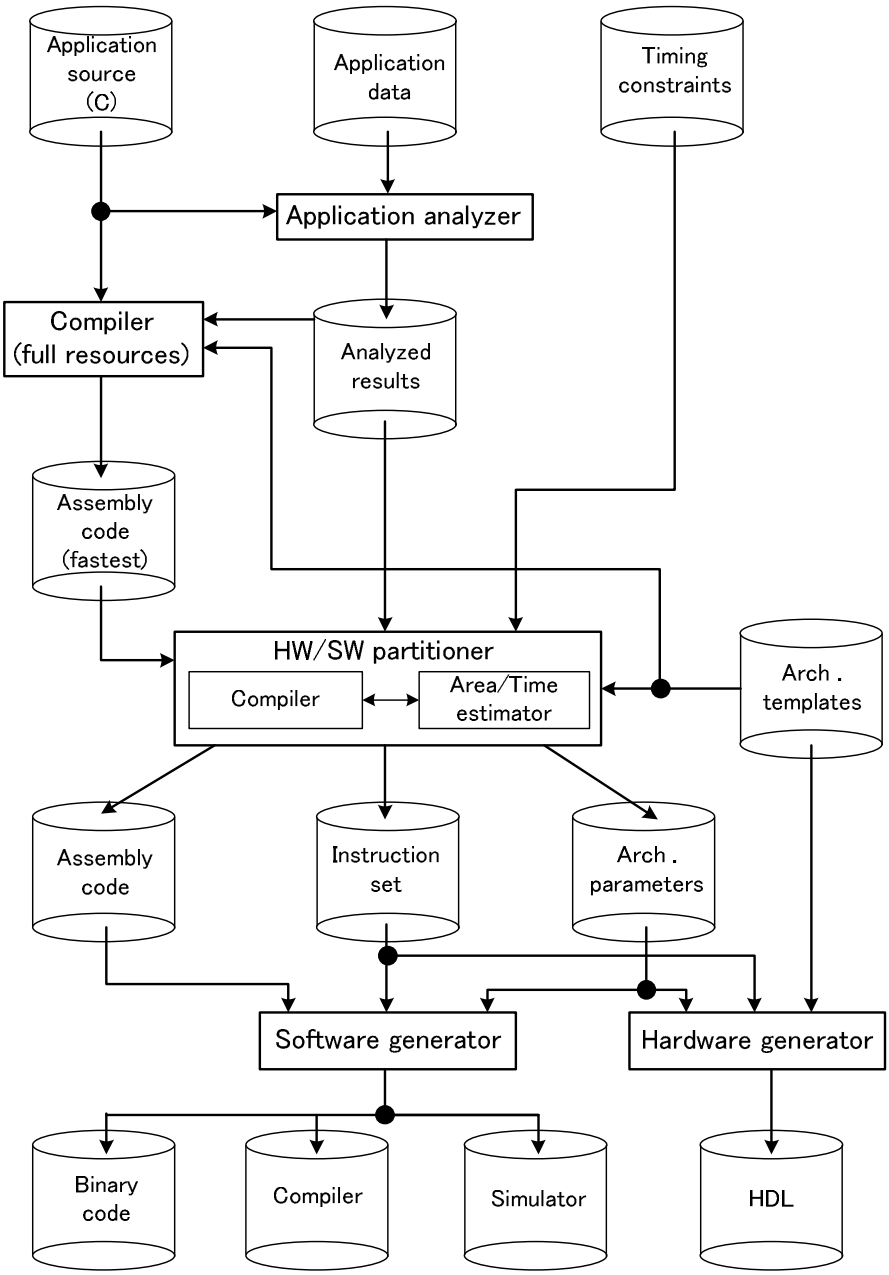

図 2.3: SPADES システム構成モデル

システムは (i)C 言語で書かれたアプリケーションプログラム，(ii) アプリケーションデータおよび (iii) アプリケーション実行時間制約を入力とし，(a) プロセッサコアの HDL 記述，(b) オブジェクトコードおよび (c) ソフトウェア環境を出力する．プロセッサ合成は，アプリケーションプログラムが実行時間制約を満たす中でハードウェア面積コストが最小となるように進められる．プロセッサコア

の面積は，プロセッサカーネルおよびカーネルに付加されるハードウェアユニットの総和により求められる．システムは**アプリケーション解析，並列化コンパイラ，ハードウェア/ソフトウェア分割**および**ハードウェア/ソフトウェア生成**の構成要素から成り立つ．以下の項で，各構成要素について解説する．

2.3.1 アプリケーション解析

C 言語で記述されたアプリケーションプログラムとアプリケーションデータを入力とし，各基本ブロックの実行回数を出力する．基本ブロックとは，最初および最後を除いて制御の分岐・結合が存在しない，一連の命令列のことである．アプリケーションプログラムにアプリケーションデータを与えて，計算機上で実行することにより各基本ブロックの実行回数を計測する．

2.3.2 並列化コンパイラ

C 言語で記述されたアプリケーションプログラムと基本ブロックの実行回数を入力とし，アプリケーション実行時間が最小となるようにスケジューリングされたアセンブリコードを出力する．プロセッサカーネルに，想定される全てのハードウェアユニットを付加した仮想的なプロセッサコア上で，アプリケーションプログラムを実行することを考える．アプリケーションが持つ計算のデータ依存制約を除いて，複合命令を最大限に利用することにより，アプリケーションプログラムを並列処理したアセンブリコードを生成する．得られたアセンブリコードは，アプリケーションプログラムの実行時間は小さいがプロセッサの面積が大きくなる．

2.3.3 ハードウェア/ソフトウェア分割

ハードウェア/ソフトウェア分割では，並列化コンパイラより出力された最速アセンブリコードと，アプリケーション実行時間制約を入力とし，時間制約を満たす中で，面積が最小となるプロセッサコア構成とそのときのアセンブリコードを出力する．

まず，面積が大きくなるが，最速アセンブリコードを実行することが可能なプロセッサコアを構成する．次に，面積最大のプロセッサコア構成から，演算器やレジスタなどハードウェアユニットを削減していき，プロセッサコアの面積を削減していく．また，それに応じてソフトウェアを変更していく．一般的に，ハードウェアユニットを削減し，ソフトウェアを変更すると，アプリケーション実行時間は長くなる．実行時間制約を満たす間，ハードウェアユニットの削減を繰り返すことにより，実行制約を満たす中で，最小のプロセッサコア構成を得る．

2.3.4 ハードウェア/ソフトウェア生成

ハードウェア生成系

ハードウェア/ソフトウェア分割系から出力されるプロセッサアーキテクチャ，命令セット，アーキテクチャテンプレートを入力とし，プロセッサのハードウェア記述を出力する．HDL 言語は，VHDL を用いる．また，ハードウェア/ソフトウェア分割系において必要となるハードウェアユニットの面積・遅延値を出力する．

ソフトウェア生成系

ハードウェア/ソフトウェア分割系から出力されるアセンブリコード，プロセッサアーキテクチャ，命令セットを入力とし，プロセッサ上で動作するアプリケーションプログラムのオブジェクトコードおよびコンパイラやシュミレータ等のソフトウェアを出力する．

2.4 本章のまとめ

本章では，本研究室で構築中の SIMD 型プロセッサコア向けハードウェア/ソフトウェア協調合成システム SPADES について説明した．

2.2 節 **SIMD 型プロセッサコアアーキテクチャモデル**では，HW/SW 協調合成システム SPADES のアーキテクチャモデルについて説明した．また，SIMD 型プロセッサコアアーキテクチャの構成要素について，2.2.1 項**プロセッサカーネル**，2.2.2 項**ハードウェアユニット**，2.2.3 項**命令セット**および 2.2.4 項**パイプライン構成**でそれぞれ説明した．

2.3 節**ハードウェア/ソフトウェア協調合成システム**では，SPADES が生成するプロセッサコアが持つ可能性のある命令セットについて説明した．また，ハードウェア/ソフトウェア協調合成システムの構成要素について，2.3.1 項**アプリケーション解析**，2.3.2 項**並列化コンパイラ**，2.3.3 項**ハードウェア/ソフトウェア分割**，2.3.4 項**ハードウェア/ソフトウェア生成**でそれぞれ説明した．

第3章

部分マッチングを考慮しMISO構造に対応したプロセッシングユニット最適化手法

3.1 本章の概要

SPADES システムでは，プロセッサコアに SIMD 型演算器やアドレッシングユニット，ハードウェアユニットといったハードウェアリソースを付加させ，アプリケーション実行時間を削減することができる．しかし，市場では H.264 をはじめとした高画質・高圧縮のデータや大規模アプリケーションを扱うようなマイクロプロセッサが求められている．今後も更に扱うデータは大きくなり，アプリケーション実行時間制約を満たすのは難しくなっていくと考えられる．

そこで，アプリケーションに特化したハードウェアを付加させることで，アプリケーション実行時間制約を満たすことができる．SPADES では，アプリケーションに特化したハードウェアであるプロセッシングユニットを付加することができ，これによりアプリケーション実行時間の削減を図ることができる．しかし，必要以上にプロセッシングユニットを付加すると，面積の増加が問題となる．よって，プロセッシングユニットの最適化を図る必要がある．

これまでに提案された，プロセッシングユニット最適化に関する研究として，文献 [8] が挙げられる．文献 [8] では，木構造のプロセッシングユニットを自動合成する手法を提案しており，プロセッシングユニットとアプリケーションの CDFG のサブグラフが部分的に一致している場合も，プロセッシングユニットを割当てる部分マッチングを考慮している．しかし，アプリケーションの CDFG は一般に複雑な構造をとっており，木構造のプロセッシングユニットしか生成できないのでは，アプリケーションに柔軟に対応することができない可能性がある．特に SIMD 型演算には，交換演算のように異なる出力を複数持つ演算が存在するため，アプリケーションに柔軟に対応するためには出力エッジを複数持つ演算ノードにも対応しなくてはならない．従って，プロセッシングユニットは木構造を含む MISO(Multiple Input, Single Output) 構造をとる必要がある．MISO 構造は，内部に複数の出力エッジを持つノードを含むことができる点で，木構造と異なる．

これまでに提案された，専用演算器合成に関する研究として，文献 [1, 2, 4, 14, 15, 16, 17] があるが，部分マッチングを考慮していないか，もしくは MISO 構造に対応していない．アプリケーションに柔軟に対応し，小面積で高性能なプロセッシングユニットを設計するためには，両者の要件を満たしている必要がある．

そこで本章では，部分マッチングを考慮し MISO 構造に対応したプロセッシングユニット最適化手法を提案する．

はじめにプロセッシングユニットの構成について説明を行い，次にプロセッシングユニット最適化における問題定義を行う．さらにプロセッシングユニットが対象とする命令について説明する．加えて，プロセッシングユニット内部の構造について解説を行う．その後，プロセッシングユニット最適化手法について解説する．

3.2 プロセッシングユニットの構成

プロセッシングユニットは，ターゲットアプリケーションに特化した専用演算器であり，プロセッサカーネルに演算器として直接接続される．この手法は，プロセッサコアにバスを介して接続する手法と比較して制御が簡単であるという利点がある．しかし，多数接続した場合，面積の増加が問題となるので，最適な組み合わせの探索が必要となる．ハードウェア接続方法の特徴を表 3.1 に示す．

表 3.1: ハードウェア接続方法と特徴

	プロセッサコアに バスを介して接続する手法	プロセッサカーネルに 演算器として直接接続する方法
付加するハードウェア	バスアービタ	特になし
ハードウェアの制御	ハードウェア上で制御	プロセッサカーネル上で制御
多数接続した場合の問題	バスの競合と制御部の面積の増大	最適なハードウェアの組み合わせ
例	IP	ALU，バレルシフタ

設計方法は，複数の演算ノードをまとめて 1 つの演算器として設計する手法を採用している．この手法は，機能モジュールを一単位として設計する手法と比べて演算効率が低くなるが，ターゲットアプリケーション以外の処理を行う場合も，対象となるアプリケーションの CDFG 上でマッチする部分があれば，プロセッシングユニットを用いて演算を行うことができるので，柔軟性が高いといえる．また，アプリケーションの CDFG を与えることで，プロセッシングユニットを自動合成することができる．これにより，新しいアルゴリズムおよびシステムが提案された場合でも，設計期間に影響を与えなくて済む．ハードウェア設計方法の特徴を表 3.2 にまとめる．また，プロセッシングユニットの構成図を図 3.1 に示す．

表 3.2: ハードウェア設計方法と特徴

	機能モジュールを 一単位として設計する手法	演算ノードをまとめて 一つの演算器として設計する手法
ターゲットアプリケーション における演算処理効率	高	中
ターゲットアプリケーション以外 の処理における演算処理効率	低	中
自動合成	難易	容易
新しいシステムやアルゴリズムの設計	新規設計	自動合成による生成

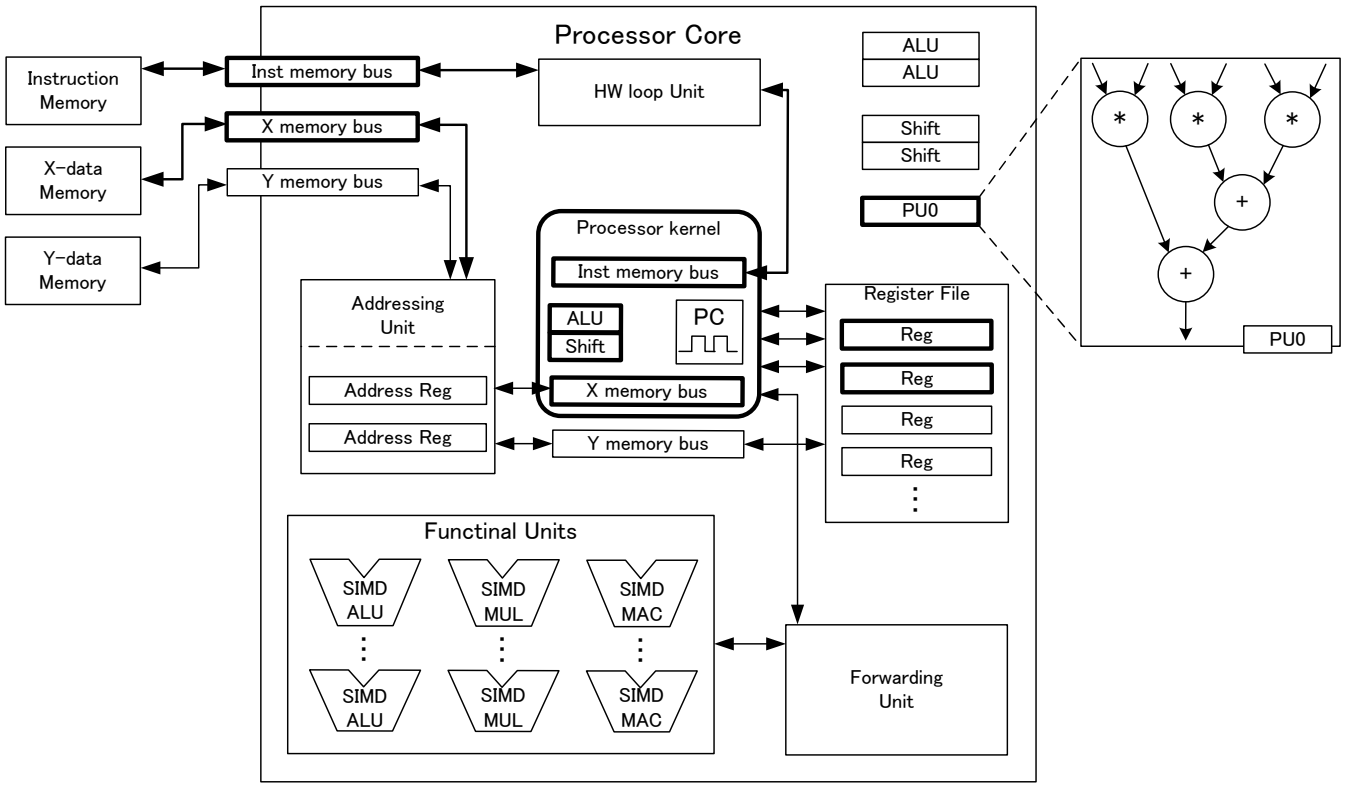

図 3.1: プロセッシングユニットの構成図

3.3 問題定義

本節では，プロセッシングユニット最適化問題を定義する．

SPADES における HW/SW 分割は，並列化コンパイラから出力されたアプリケーションの CDFG および実行時間制約を入力とし，実行時間制約を満たし面積が最小となるプロセッサコアの構成およびアセンブリコードを出力する．まず，初期スケジューリングで実行時間が最速となる初期プロセッサコアの構成および初期アセンブリコードを生成する．次に，HW/SW 分割にて初期プロセッサからレジスタやハードウェアユニットの削減を行う．プロセッシングユニットは，初期に粒度の大きいプロセッシングユニットを生成し，徐々に粒度の小さいプロセッシングユニットを生成していくことで，プロセッシングユニット最適化を行う．これらの最適化により，プロセッサコアの面積を減少させていく．

プロセッシングユニットの最適化問題について，以下のように定義する．

プロセッシングユニット最適化問題 プロセッシングユニット最適化問題とは，アプリケーションの CDFG，プロセッサの構成情報，実行時間制約 T_c が与えられたときに，アプリケーションの実行時間 T_{app} が $T_{app} \leq T_c$ を満たすプロセッシングユニットの構成を出力することである．プロセッシングユニットの面積 A_{pu} の最小化を目的とする． □

プロセッシングユニットの面積/遅延見積もりには，文献 [13] を用いる．アプリケーション実行時間 T_{app} は，アセンブリコードを実行するのに必要な総クロック数とプロセッサコアのクロック周期の乗算によって見積もる．アセンブリコードの総クロックサイクル数を N_{cycle}，プロセッサコアのクロック周期を T_{cycle} とすると，アプリケーション実行時間 T_{app} は式 3.1 により求められる．

$$T_{app} = N_{cycle} \times T_{cycle} \tag{3.1}$$

パイプライン構成で定義された全てのパイプライン構成について実行することによって，各パイプライン構成において実行時間制約を満たす中で面積最小のプロセッサコア構成を得る．得られた各構成の中から最も面積の小さい構成を選択する．

3.4 命令定義

プロセッシングユニット生成におけるクラスタリングの対象となる演算ノードは，表 3.3 の命令および SIMD 型加算命令・減算命令・乗算命令・乗加算命令・交換命令・拡張命令・縮小命令の演算ノードとする．

また，クラスタリングされた演算ノードは，入力オペランドの種類であるレジスタおよび IMM の区別がなくなり，ADDI が ADD と同じ扱いとなる．INC も ADD の第 2 オペランドに即値の”1”が入力されていると考えられ，ADD と同等の扱いになる．このように，プロセッシングユニット内部の命令は抽象度が上がる．プロセッシングユニット内部の命令をプロセッシングユニット命令 (pu_operation) と定義する．プロセッシングユニット命令の概念図を図 3.2 に示す．

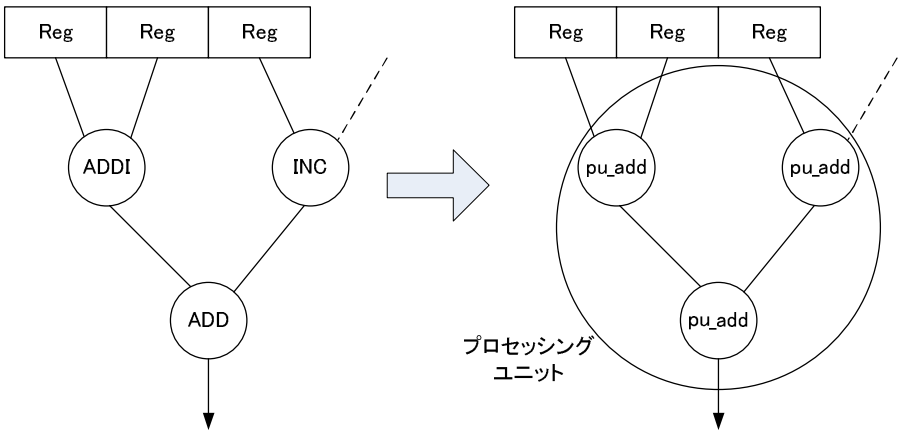

図 3.2: プロセッシングユニット命令の概念図

また，演算ノードの入力オペランドにおいて，第 1 オペランドと第 2 オペランドが入れ替わっても演算結果が同じとなる演算命令を，**可換命令**と定義する．また，第 1 オペランドと第 2 オペランドが入れ替わると演算結果が異なる演算命令を**非可換命令**と定義する．

表 3.3: プロセッシングユニットの内包演算命令対象

命令	ニモニック	動作	pu_operation	可換命令
ADD	ADD R1, R2, R3	R1 + R2	pu_add	○
SUB	SUB R1, R2, R3	R3 ← R1 - R2	pu_sub	×
SRA	SRA R1, R2, R3	R3 ← R1 >> $R2_{0..x}$(arithmetic)	pu_srl	×
SRL	SRL R1, R2, R3	R3 ← R1 >> $R2_{0..x}$	pu_srl	×
SLL	SLA R1, R2, R3	R3 ← R1 << $R2_{0..x}$	pu_sll	×
AND	AND R1, R2, R3	R3 ← R1 & R2	pu_and	○
OR	OR R1, R2, R3	R3 ← R1 | R2	pu_or	○
XOR	XOR R1, R2, R3	R3 ← R1 ^ R2	pu_xor	○
MUL	MUL R1, R2, R3	R3 $\leftarrow_{n_d}$ $(R1 * R2)_{0..(n_d-1)}$	pu_mul	○
DIV	DIV R1, R2, R3	R3 ← R1 / R2	pu_div	×
MAC	MAC R1, R2, R3	R3 ← R1 * R2 + R3	pu_mac	○
INC	INC R1	R1++	pu_add	○
DEC	DEC R1	R1--	pu_sub	×
ADDI	ADDI R1, R2, imm	R2 ← R1 + imm	pu_add	○
SUBI	SUBI R1, R2, imm	R2 ← R1 - imm	pu_sub	×
SRAI	SRAI R1, R2, imm	R2 ← R1 >> $imm_{0..x}$(arithmetic)	pu_sra	×
SRLI	SRLI R1, R2, imm	R2 ← R1 >> $imm_{0..x}$	pu_srl	×
SLLI	SLAI R1, R2, imm	R2 ← R1 << $imm_{0..x}$	pu_sll	×
ANDI	ANDI R1, R2, imm	R2 ← R1 & imm	pu_and	○
ORI	ORI R1, R2, imm	R2 ← R1 | imm	pu_or	○
XORI	SORI R1, R2, imm	R2 ← R1 ^ imm	pu_xor	○
MULI	MULI R1, R2, imm	R2 $\leftarrow_{n_d}$ $(R1 * imm)_{0..(n_d-1)}$	pu_mul	○
DIVI	DIVI R1, R2, imm	R2 ← R1 / imm	pu_div	×

3.5 アルゴリズム

本節では，プロセッシングユニット最適化におけるアルゴリズムの提案を行う．

提案手法は，大きく**生成系**と**再構成系**に分かれる．生成系において，プロセッシングユニットの面積を考慮せずにアプリケーション実行時間 T_{app} が最小となるプロセッシングユニットの構成を得る．そして再構成系にてプロセッシングユニットの粒度を小さくしていき，実行時間制約 T_c に違反したら，違反する前のプロセッシングユニットの構成を解として出力する．生成系は，演算ノードをまとめて 1 つの命令にする**クラスタリング**，アプリケーションに対して新しく生成されたプロセッシングユニットを割付ける**バインディング**，生成されたプロセッシングユニットの内部構造に対してパイプラインレジスタを割付ける**プロセッシングユニット内パイプライン化**および CDFG に対してステップ数の計算およびレジスタを割付ける**スケジューリング**により成り立つ．再構成系は，現在生成されているプロセッシングユニットを削除し，より粒度の小さいプロセッシングユニットを生成する**リクラスタリング**，削除されたプロセッシングユニットが割付けられている CDFG 内のノードに対してバインディングを解除し，新たに生成されたプロセッシングユニットを割付ける**リバインディング**，**プロセッシングユニット内パイプライン化**および**スケジューリング**により成り立つ．以下に生成系および再構成系の概要について示す．また，アルゴリズムの全体の流れを図 3.3 に示す．

生成系　はじめに，ターゲットアプリケーションの全 DFG の中で，実行サイクル数と演算ノードの積が最大の DFG を対象として，対象となる DFG の接点にクラスタリングを行い，プロセッシングユニットを生成する．生成したプロセッシングユニットを PU_{gen} とする．次に，ターゲットアプリケーションの全 DFG に対して，PU_{gen} のバインディングを行う．この時点では，PU_{gen} 内部はパイプライン化されておらず，プロセッサコアのクリティカルパスとなる可能性がある．そこで，PU_{gen} 内にパイプラインレジスタを挿入し，PU_{gen} のクリティカルパスの最適化を行う．最後に，PU_{gen} をバインディングした DFG に対してリストスケジューリングを行う．また，ステップ数の割付けと汎用レジスタの割当てを行い，全 DFG の実行サイクル数の更新を行う．以上により，プロセッシングユニットを 1 つ生成する．

再構成系　はじめに，生成系において生成されたプロセッシングユニットを PU_{target} とする．PU_{target} に対して開梱を行い，再度対象となったプロセッシングユニット内の演算ノードに対してリクラスタリングを行う．この後，再生成したプロセッシングユニットを PU_{regen} とする．次に，ターゲットアプリケーションの全 DFG に対して，PU_{target} のバインディングを解除し，プロセッシングユニットを割当てる前の状態に戻す．その後，PU_{regen} のリバインディングを行う．さらに，PU_{regen} 内にパイプラインレジスタを挿入し，PU_{regen} のクリティカルパスの最適化を行う．最後に，PU_{regen} をバインディングした DFG に対してリストスケジューリングを行う．また，ステップ数の割付けと汎用レジスタの割り当てを行い，全 DFG の実行サイクル数の更新を行う．以上により，プロセッシングユニットの最適化を図る．

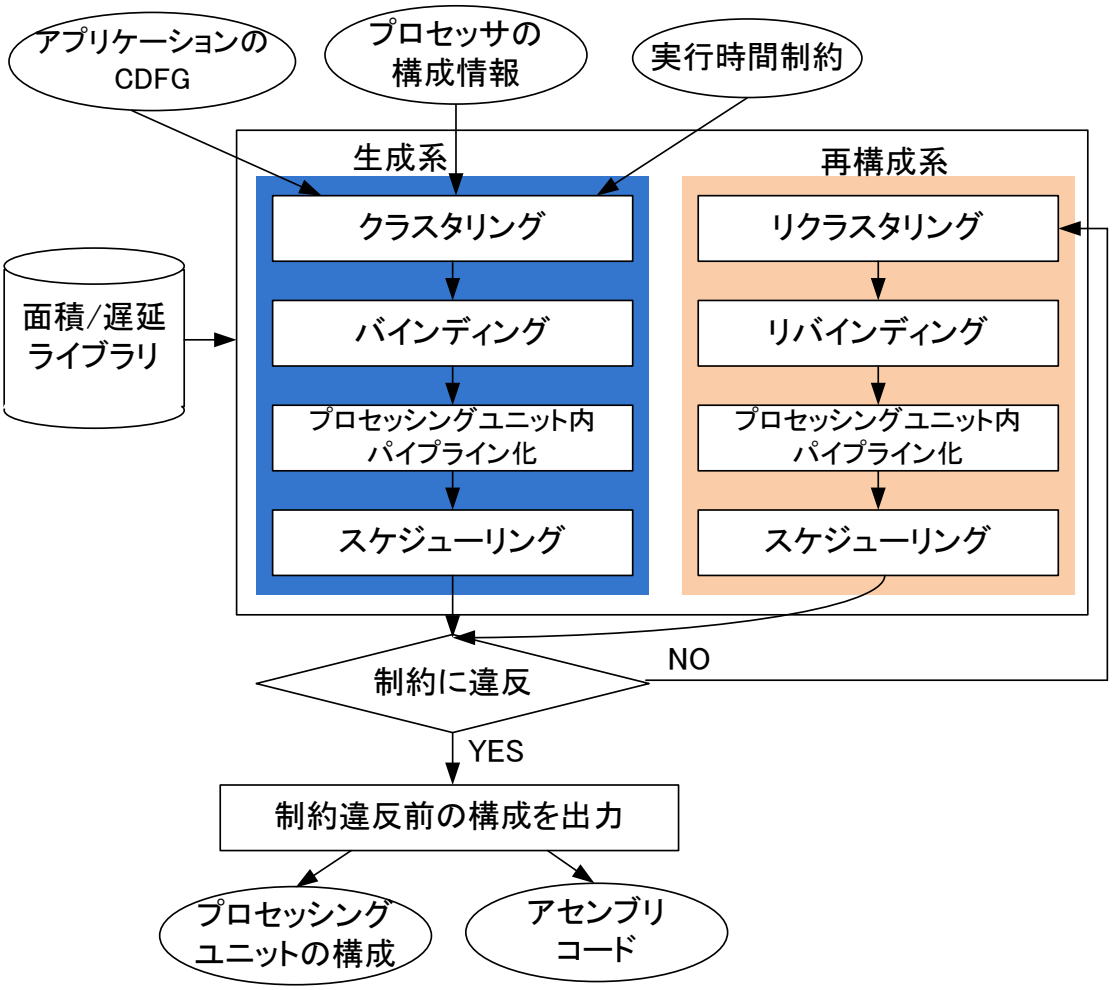

図 3.3: アルゴリズムの流れ

3.5.1 生成系

生成系では，面積を考慮せずアプリケーション実行時間 T_{app} が最小となるプロセッシングユニットを生成する．

生成系は，**クラスタリング**，**バインディング**，**プロセッシングユニット内パイプライン化**，**スケジューリング**の 4 つの工程により成り立っている．以降でそれぞれの工程について解説する．

クラスタリング

クラスタリングでは，アプリケーションの CDFG を入力とし，複数の演算ノードをまとめて実行するプロセッシングユニットを生成し出力する．プロセッシングユニット内部のグラフ構造は MISO 構造をとり，根のノードからのみ演算結果を出力する．クラスタリングの実行は大きく 4 つのステップに分かれる．

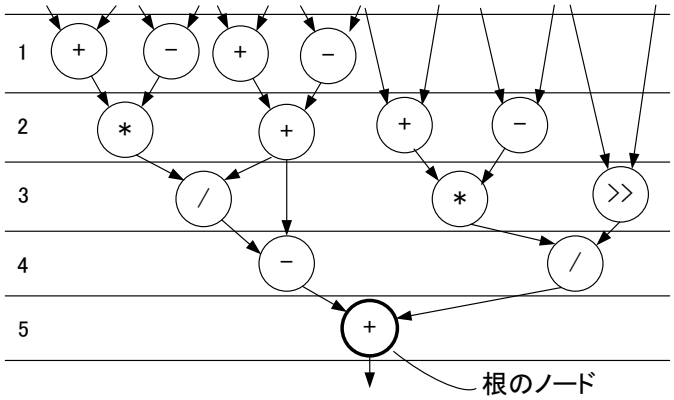

図 3.4: C-Step1 の操作

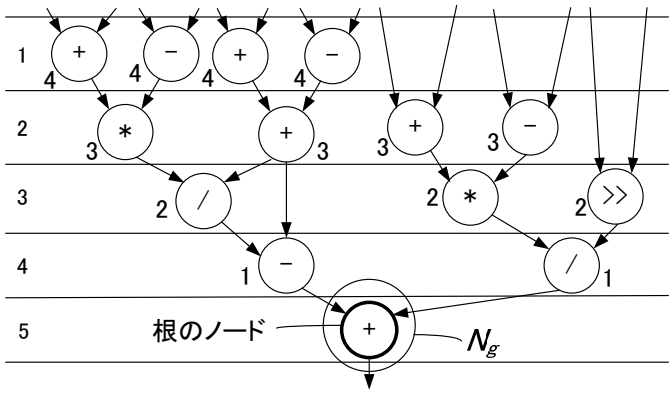

図 3.5: C-Step2 の操作

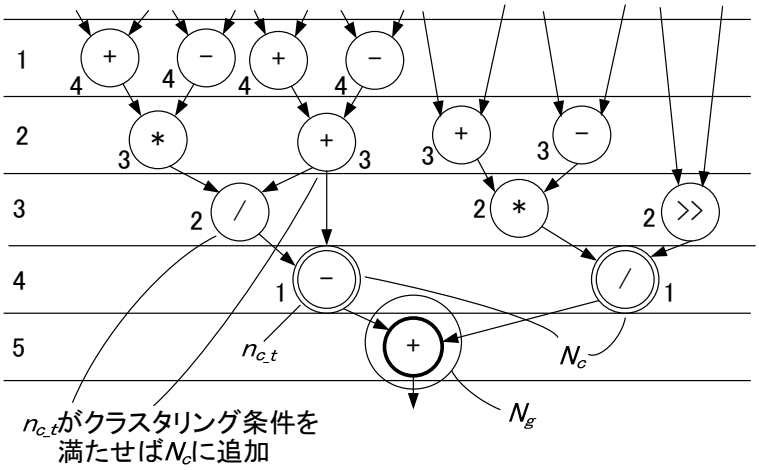

図 3.6: C-Step3 の操作 (1)

C-Step1 アプリケーションの CDFG に含まれる全 DFG の中で，ノード数が一番多い DFG の演算ノードに対し ALAP スケジューリングを行う．そして，対象の DFG の中で ALAP 値が最大の演算ノードを，プロセッシングユニットにおける根のノードとする．これらの操作を行った結果を図 3.4 に示す.

C-Step2 DFG 内の演算ノードに対し根のノードとの ALAP 値の差を n_{p_l} と定義する．DFG 内の根のノード以外の各演算ノードに対して，n_{p_l} の値を求める．ある演算ノード n_1 の入力となる演算ノード n_2 は，必ず n_{p_l} の値が n_1 より大きい値をとる．また，クラスタリングされたノード集合を N_g と定義し，N_g に根のノードを追加する．これらの操作を図 3.5 に示す.

C-Step3 クラスタリング候補を N_c と定義し，N_c に根のノードの入力となる演算ノードを追加する．N_c の中で n_{p_l} の値が最小のノードをクラスタリング対象 n_{c_t} とし，以下のクラスタリング条件を満たすかどうか判定を行う．これらの操作を図 3.6 に示す.

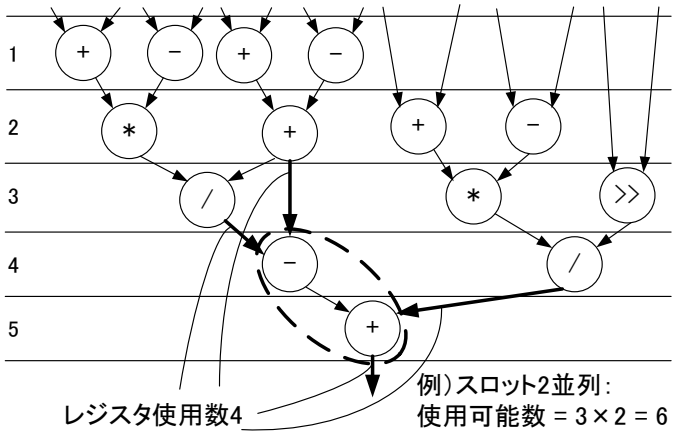

図 3.7: クラスタリング条件 (1) の判定

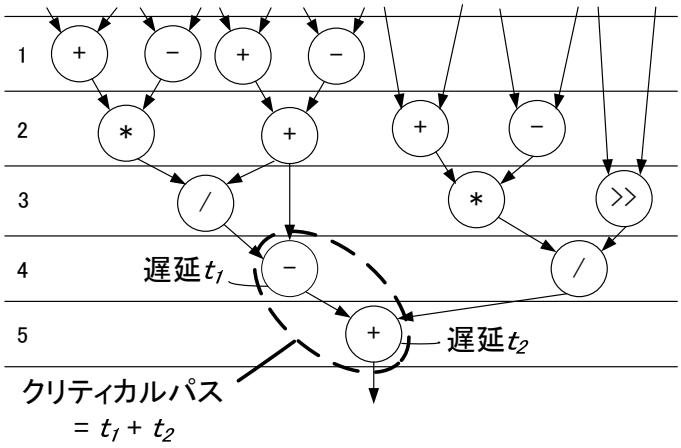

図 3.8: クラスタリング条件 (2) の判定

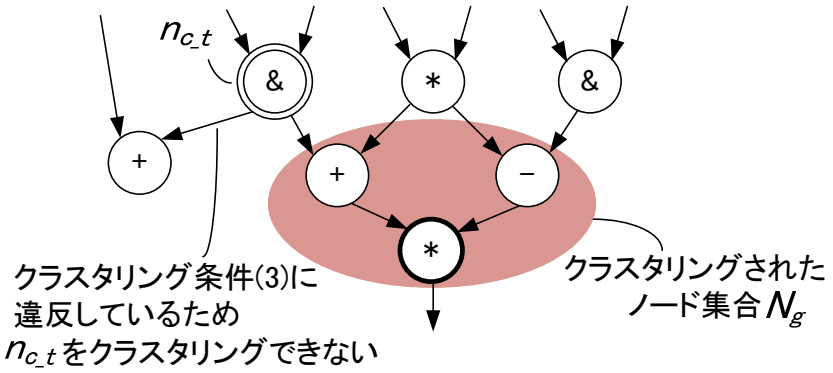

図 3.9: クラスタリング条件 (3) の判定

また，クラスタリング条件を以下の (1)〜(3) に示す．ここで，PU_{in} はプロセッシングユニットの入力数，PU_{out} はプロセッシングユニットの根のノードで異なる出力の数，$slot_{core}$ はプロセッサコアのスロット数，$core_{c\_p}$ はプロセッサコアのクリティカルパス遅延時間，PU_{c_p} はプロセッシングユニット内のクリティカルパス遅延時間である．d_m は遅延係数であり，プロセッサのパイプライン構成における実行ステージ数を e_s とすると，$1 \leq d_m \leq e_s$ の値をとる．

クラスタリング条件 (1) n_{c_t} をクラスタリングした際，レジスタ数制約 r_c を満たす (図 3.7)

$$PU_{in} \leq r_c = slot_{core} \times 3 - PU_{out}$$

クラスタリング条件 (2) n_{c_t} をクラスタリングした際，遅延制約 d_c を満たす (図 3.8)

$$PU_{c\_p} \leq d_c = core_{c\_p} \times d_m$$

クラスタリング条件 (3) n_{c_t} の出力となる演算ノードが N_g に含まれている (図 3.9)

クラスタリング条件 (3) により，プロセッシングユニットの MISO 構造への対応を可能にしている．プロセッシングユニットが根のノードのみから出力するためには，n_{c_t} の出力エッジは

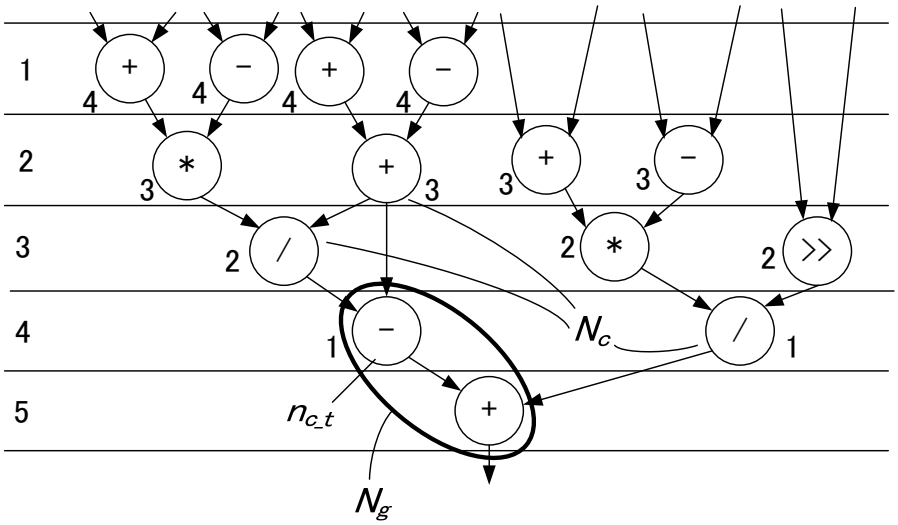

図 3.10: C-Step3 の操作 (2)

全てプロセッシングユニット内のノードに向いていなければならない．n_{p_l} の値が小さいノードから順にクラスタリングを行っているので，n_{c_t} の出力先のノードは全てクラスタリングされている必要がある．この様子を図 3.9 に示す．

クラスタリング条件を全て満たしていたら，n_{c_t} に対して N_g とクラスタリングを行い，n_{c_t} の入力となる演算ノードを N_c に追加する．また，クラスタリング条件を満たしている場合もそうでない場合も，n_{c_t} を N_c から取り除く．これらの操作を行った結果を図 3.10 に示す．

C-Step4 N_c の中で n_{p_l} が最小のノードを n_{c_t} とし，C-Step3 を繰り返す．N_c を全てクラスタリングするか，もしくはクラスタリング条件を満たさなくなったら終了する．クラスタリングが終了したら，現在の根のノードの次に ALAP 値が大きいノードを新たな根のノードとして C-Step2 に戻る．

N_g の中で，ノードの梱包数が最大かつ占有しているスロット数でノード数を割った値が最大である N_g をプロセッシングユニットとして生成する．

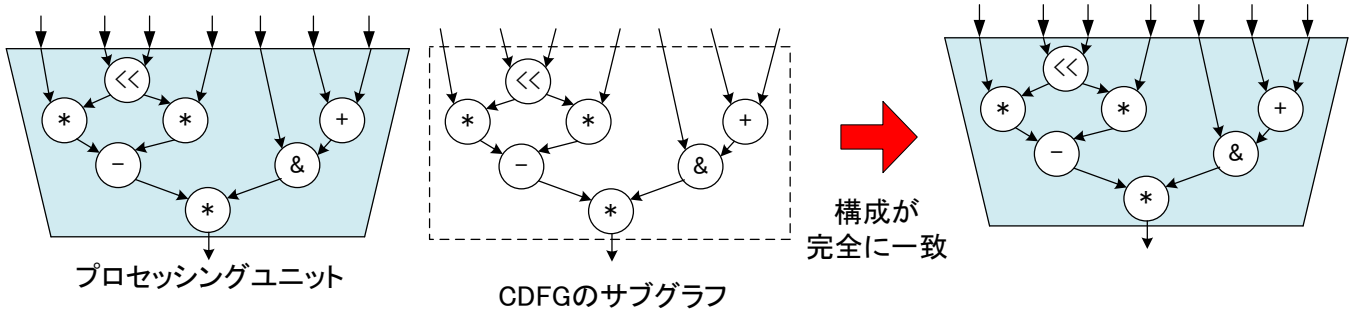

図 3.11: 完全マッチング

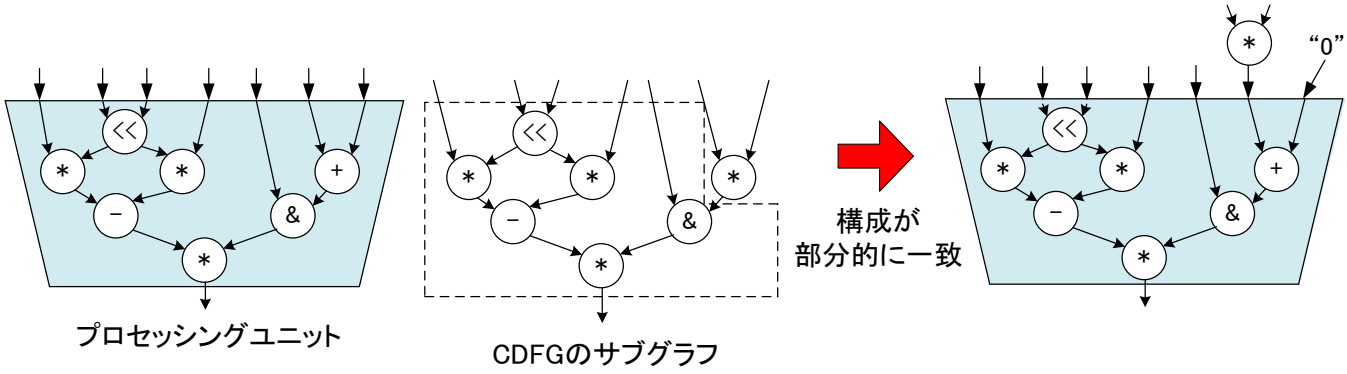

図 3.12: 部分マッチング (a)

バインディング

バインディングでは，MISO 構造のプロセッシングユニットとアプリケーションの CDFG を入力として，プロセッシングユニットが割付けられた CDFG を出力する．まず，バインディングのマッチングタイプを定義する．マッチングタイプは，プロセッシングユニットと CDFG のサブグラフの構成がまったく同じである完全マッチングと，プロセッシングユニットの部分的なリソースを使用し演算を実行することのできる部分マッチングに分けられる．プロセッシングユニットと完全マッチングしている DFG のサブグラフは図 3.11 のようになる．また，部分マッチングは更に以下の (a)〜(d) に分類される．

(a) プロセッシングユニット内部で，実行されない演算ノードが存在する．図 3.12 では，プロセッシングユニット内部の加算ノードが実行されないので，入力の片方に定数 0 を入力することで演算を行わせない．

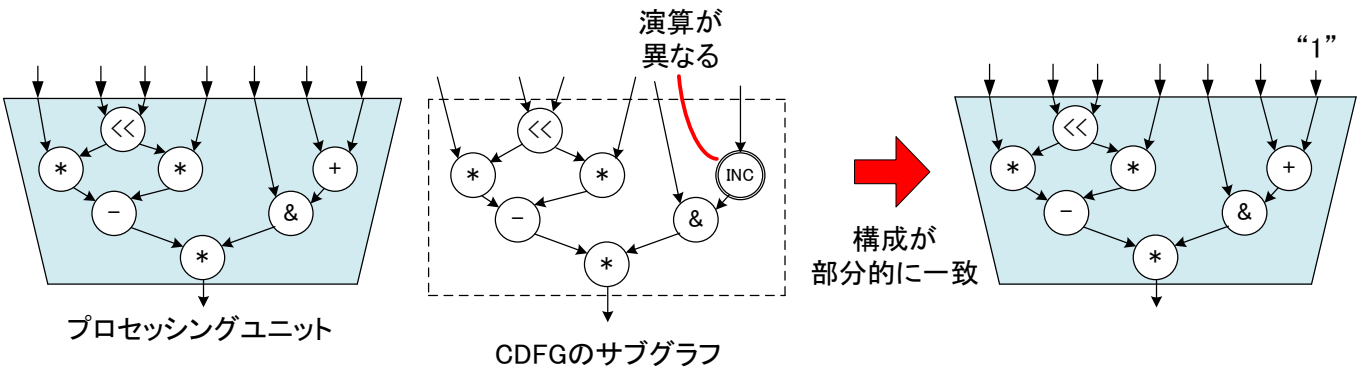

図 3.13: 部分マッチング (b)

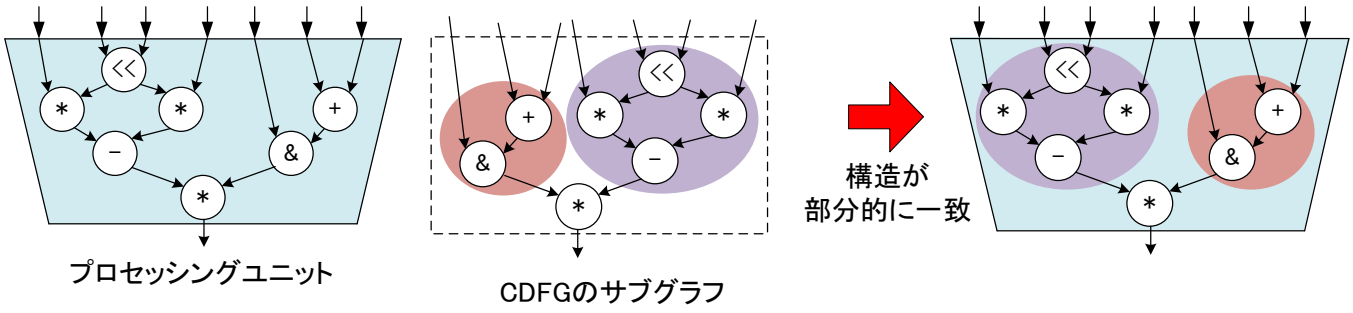

図 3.14: 部分マッチング (c)

(b) 実行される演算の種類が異なるが，プロセッシングユニットで実行可能である．図 3.13 では，プロセッシングユニット内部の加算ノードに対して，DFG のサブグラフの演算ノードは入力に 1 を加えて出力する INC 演算である．しかし，加算ノードの入力の片方に 1 をとることで，プロセッシングユニットで INC 演算を実行することができる．

(c) 入力の位置が異なるが，可換演算である．図 3.14 では，プロセッシングユニットと DFG のサブグラフで，根のノードへの入力の位置が異なるが，根のノードが乗算ノードで可換演算のため，入力の位置が異なっていてもプロセッシングユニットで実行することができる．

(d) 以上 (a)〜(c) の複合構造を持っている．

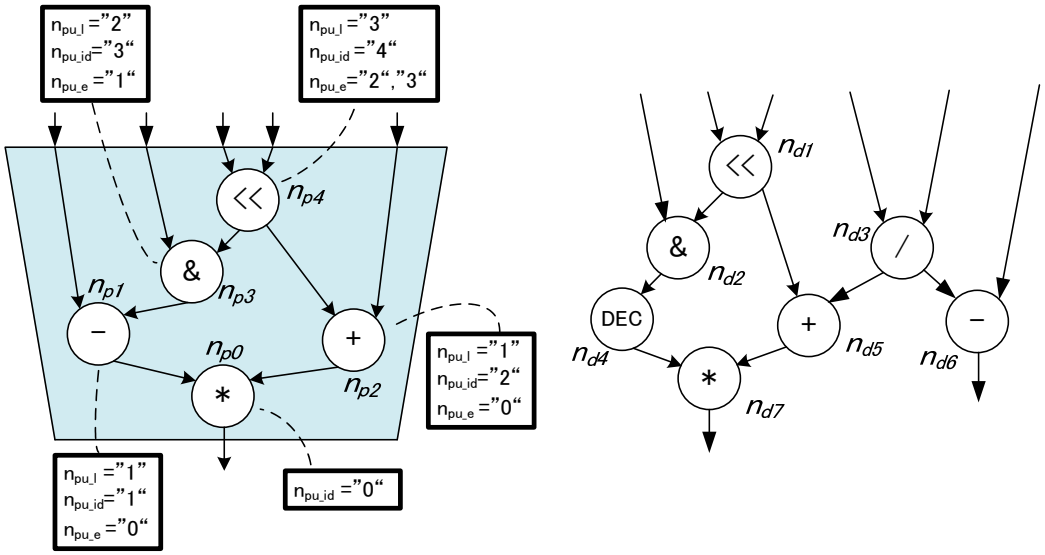

図 3.15: B-Step1 の操作

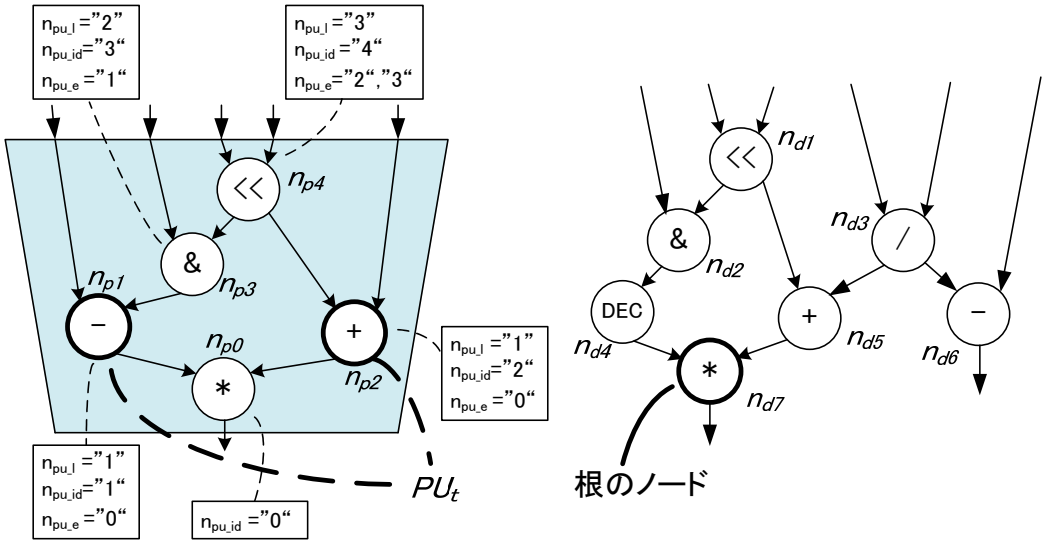

図 3.16: B-Step2 の操作

バインディングの実行は大きく 5 つのステップに分かれる．ここでは，図 3.15 左のプロセッシングユニットを右の DFG にバインディングすることを考える．

B-Step1 プロセッシングユニット内部の根のノードである n_{p0} 以外の演算ノードに対して，n_{p0} との最長距離 n_{pu_l} を求める．また，n_{pu_l} が小さい順に判別番号 n_{pu_id} を割付ける．そして，n_{p0} 以外のノードに対して，出力ノードの n_{pu_id} を表す n_{pu_e} を割付ける．これらの操作を行った結果を図 3.15 に示す．図 3.15 において，n_{p4} が n_{pu_e} を 2 つ以上持っているので，このプロセッシングユニットは木構造でない MISO 構造である．

B-Step2 プロセッシングユニットのバインディング対象のノード集合を PU_t と定義する．$n_{pu\_l} = 1$ である n_{p1} と n_{p2} を PU_t に追加する．n_{p0} と同じ乗算ノードである n_{d7} を，バインディングする根のノードとする．この操作を図 3.16 に示す．

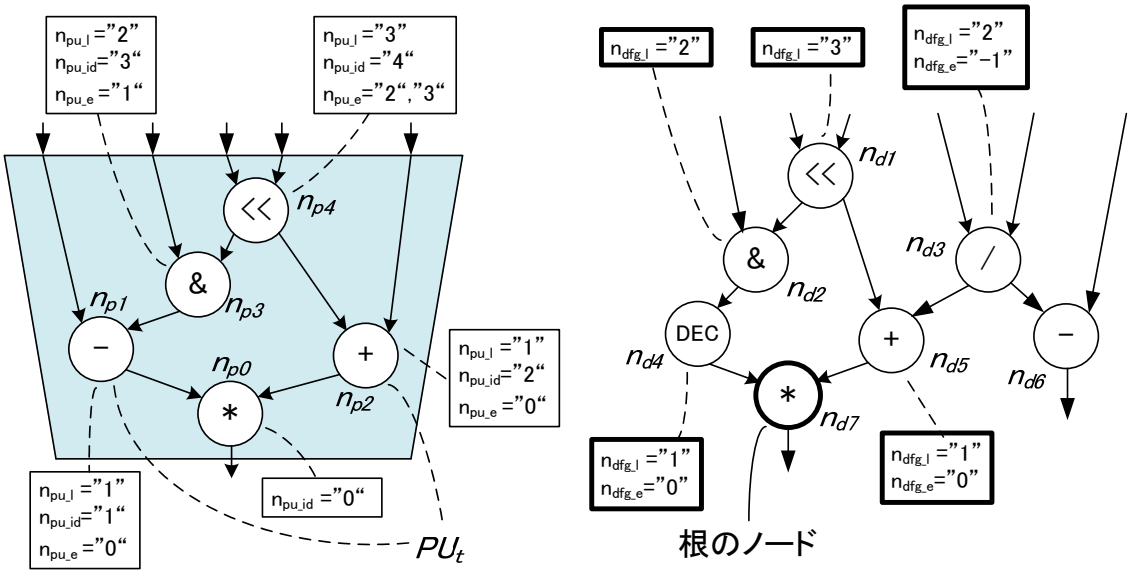

図 3.17: B-Step3 の操作

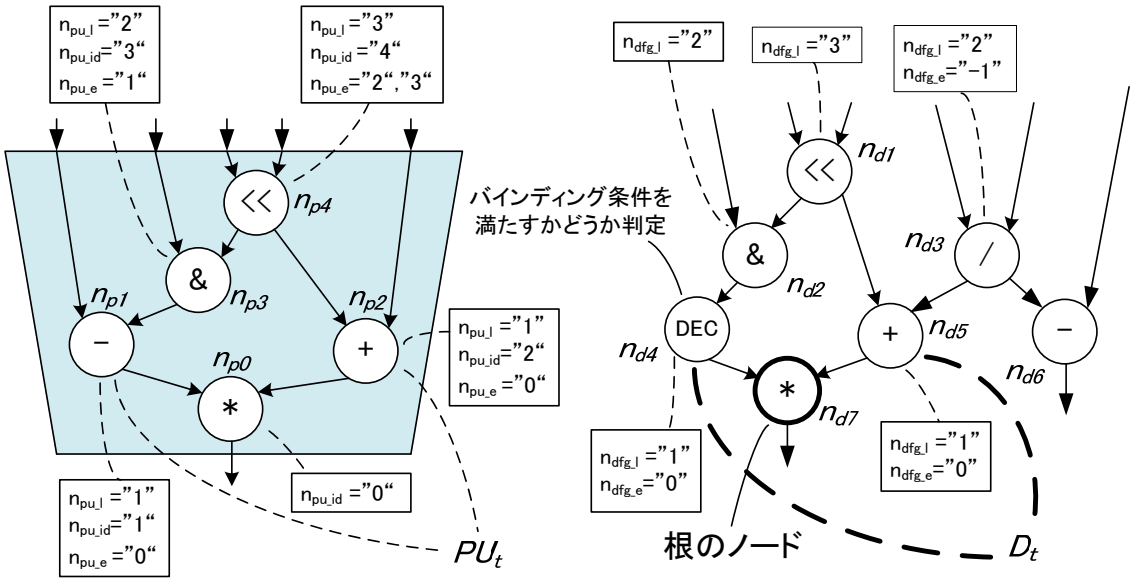

図 3.18: B-Step4 の操作 (1)

B-Step3 n_{d7} から入力側にあるノードに対して，n_{d7} との最長距離 n_{dfg_l} を求める．また，n_{d7} への出力エッジを持つノードに対して，出力エッジ n_{dfg_e} に 0 を割付け，n_{dfg_l} が割付けられていないノードへの出力エッジを持つノードに対して，バインディングが不可能であることを示す -1 を n_{dfg_e} に割付ける．この操作を図 3.17 に示す．

B-Step4 DFG のノードのバインディング候補を D_t とする．図 3.18 では n_{d7} の入力ノードである n_{d4} と n_{d5} を D_t とする．D_t の中で n_{dfg_l} が最小のノードをバインディング対象 n_{b_t} とする．n_{d4} と n_{d5} は共に $n_{dfg\_l} = 1$ なので，ここでは n_{d4} を n_{b_t} とする．n_{b_t} に対して，バインディング条件を満たすかどうか判定を行う．この操作を図 3.18 に示す．

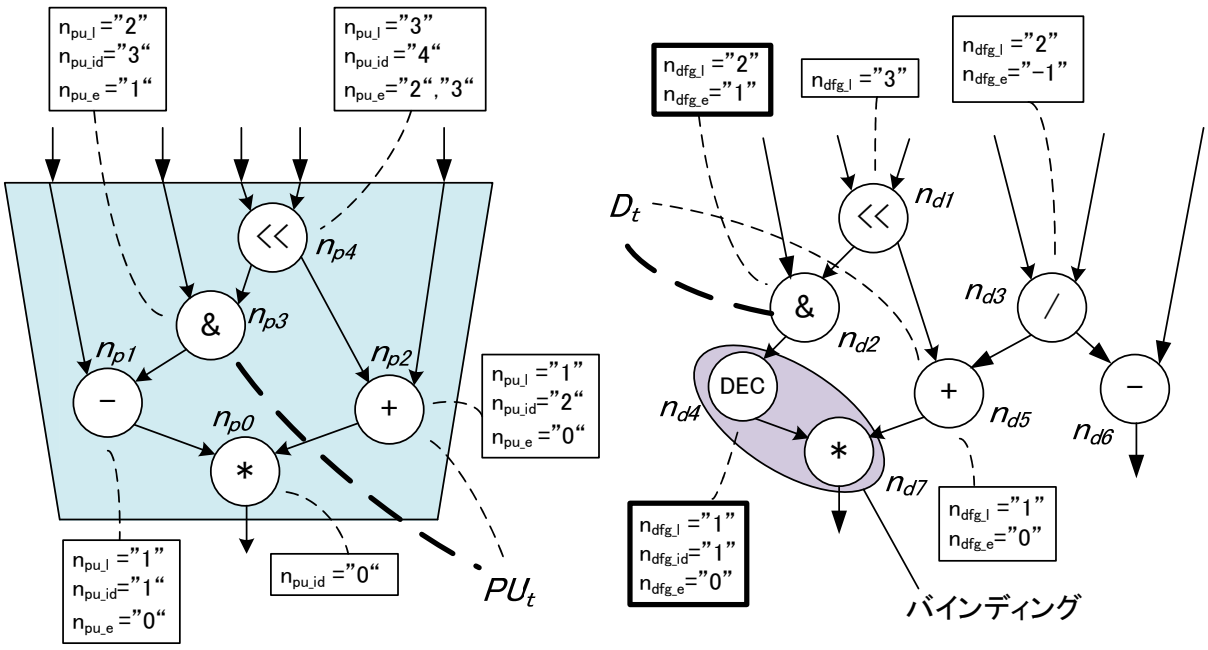

図 3.19: B-Step4 の操作 (2)

バインディング条件を以下に示す．なお，PU_t の中で n_{pu_l} の値が最小のノードを n_{pu_t} とする．

バインディング条件 (1) n_{b_t} と n_{pu_t} が同じ種類の演算ノードである．即値演算は変数の演算に置き換えて比較する

バインディング条件 (2) n_{b_t} の n_{dfg_l} が n_{pu_t} の n_{pu_l} と同じ値である

バインディング条件 (3) n_{b_t} の n_{dfg_e} が n_{pu_t} の n_{pu_e} と同じ値である

バインディング条件 (4) n_{pu_t} の出力ノードが非可換演算の場合，n_{b_t} の出力ノードへの入力の位置が n_{pu_t} と同じである

PU_t で n_{pu_l} の値が最小である n_{p1} と n_{d4} を比較すると，n_{p1} は減算ノードであるのに対し n_{d4} は入力から 1 引いて出力する DEC 演算ノードである．しかし，減算ノードの 2 つ目の入力に定数 1 をとることで DEC 演算の実行が可能なのでバインディング条件 (1) を満たす．これにより図 3.13 の部分マッチング (b) を実現している．また，n_{p1} の n_{pu_l} と n_{d4} の n_{dfg_l} は共に 1 であり，バインディング条件 (2) を満たす．そして，n_{p1} の n_{pu_e} と n_{d4} の n_{dfg_e} が共に 0 なので，バインディング条件 (3) も満たす．n_{p1} の出力ノードである n_{p0} は乗算ノードで可換なので，バインディング条件 (4) を考慮する必要はない．n_{d4} が全てのバインディング条件を満たしているので，バインディングを行う．そして，n_{d4} を D_t から除き，n_{d4} の入力ノードである n_{d2} を新たに D_t に追加する．同様に，n_{p1} を PU_t から除き，n_{p1} の入力ノードである n_{p3} を新たに PU_t に追加する．また，n_{d4} の n_{dfg_id} に n_{p1} の n_{pu_id} である 1 を割付け，n_{d4} への出力を持つ n_{d2} の n_{dfg_e} に 1 を割付ける．これらの操作を行った結果を図 3.19 に示す．

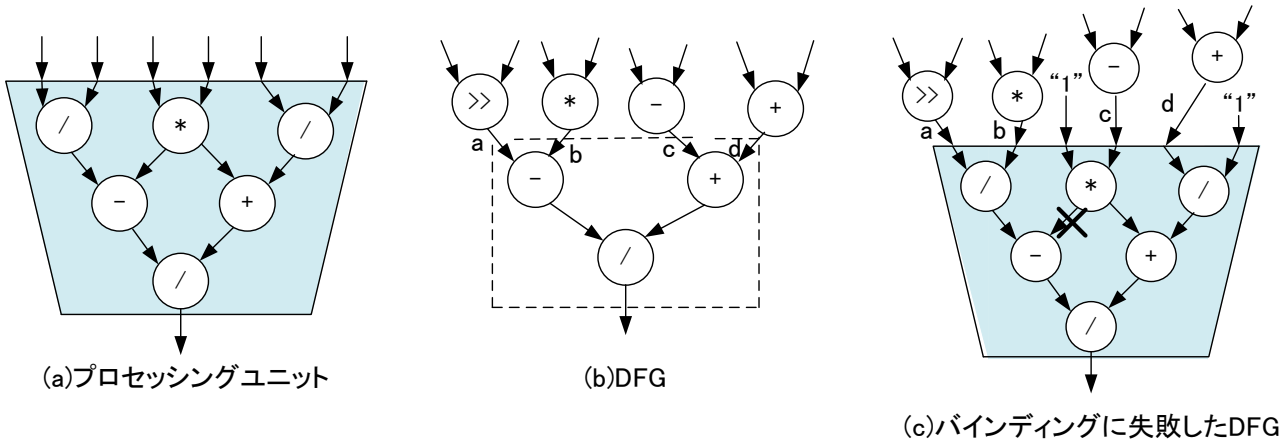

図 3.20: バインディングできない DFG の例

B-Step5 D_t の中で n_{dfg_l} が最小である n_{d5} を n_{b_t} とし，B-Step4 の操作を行う．D_t の中で n_{pu_t} とのバインディング条件を満たす演算ノードがなければ，n_{p0} と同じ種類の演算ノードを DFG から探し，B-Step3 に戻る．

ただし，出力エッジが複数あるノードのバインディングが行われなかった場合は，その根のノードに関して全体のバインディングが失敗したとみなし，部分マッチングも行わないことにする．これは，図 3.12 の部分マッチング (a) において，実行しないノードに対して定数を入力する場合，出力エッジを複数持つノードが正しい値を出力しなくなるためである．図 3.20 では，$c+d$ を正しく実行させようと，プロセッシングユニットの乗算ノードに定数 1 を入力させた結果，$a-b$ が正しく実行できなくなっている．

プロセッシングユニット内パイプライン化

プロセッシングユニット内パイプライン化では，プロセッシングユニットの構成とプロセッサコアのパイプライン構成における実行ステージ数を入力として，パイプライン化させたプロセッシングユニットの構成を出力する．

複数の演算をまとめて実行するプロセッシングユニットは，他の演算器と比較して遅延時間が長くなり，プロセッサコアにおけるクリティカルパスとなる可能性がある．そこで，プロセッシングユニット内部にパイプラインレジスタを挿入し，プロセッシングユニットのクリティカルパス PU_{c_p} を分割する．プロセッシングユニットを n 分割すると，理想的には $\frac{PU_{c\_p}}{n}$ になるが，現実的には分割対象の演算ノードが均一の遅延を持っていないので，等分割を行うことはできない (図 3.21)．

そこで，文献 [6] の手法を用いて，パイプラインステージにおける遅延値のリバランスを行う．文献 [6] では，各演算を最小モジュールに分割することで，パイプライン化において演算ノード単位ではなく最小モジュール単位でパイプライン化を行う手法を提案している．この手法を用いてパイプラインレジスタを挿入し，遅延が均一になるようにプロセッシングユニットのクリティカルパスを分割

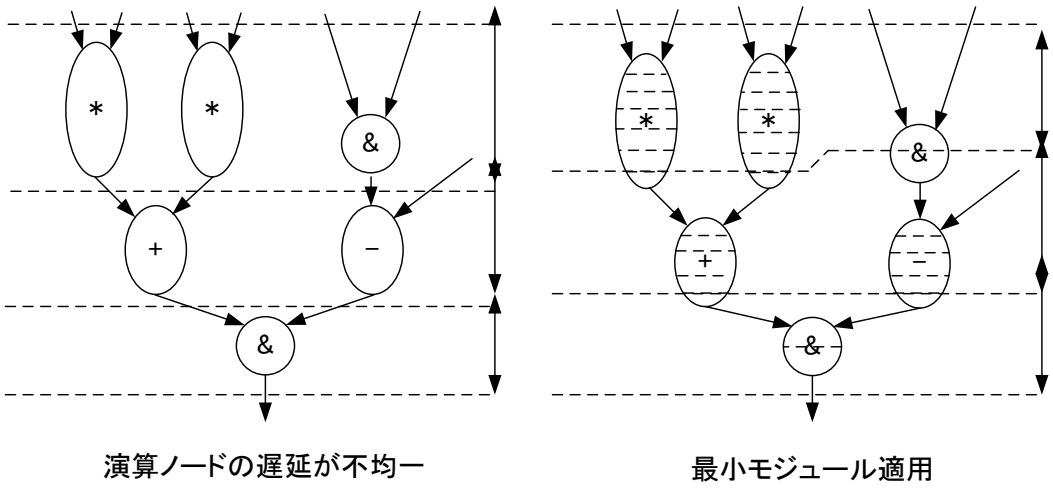

図 3.21: プロセッシングユニット内の遅延のばらつき

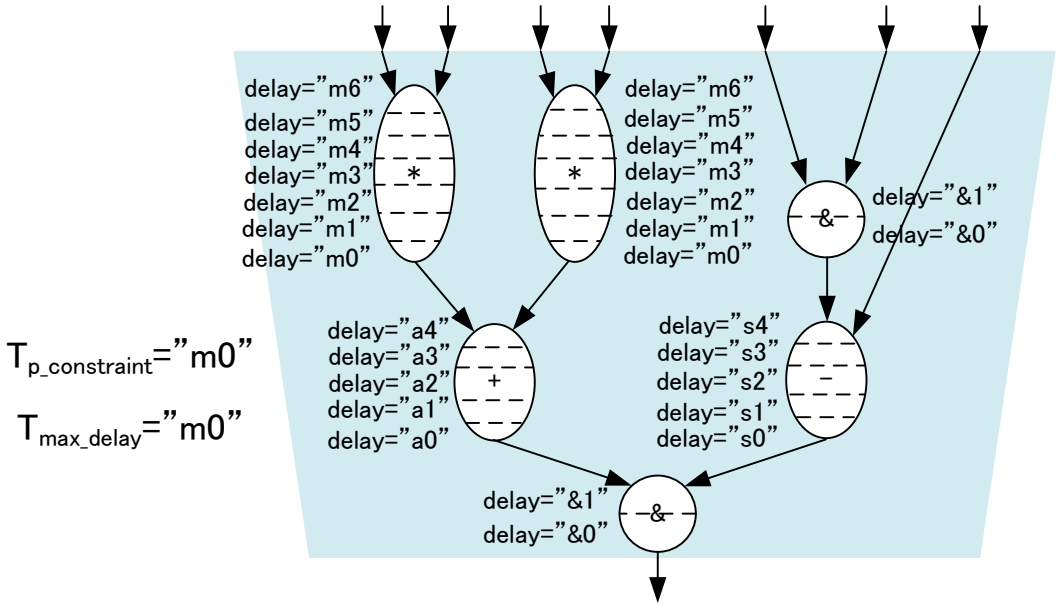

図 3.22: P-Step1 の操作

する．

プロセッシングユニット内パイプライン化の実行は大きく 4 つのステップに分かれる．

P-Step1 まず，対象となるプロセッシングユニット内部の全最小モジュールのうち，最大遅延を持つ最小モジュールの遅延を T_{max_delay} とする．ここで，パイプラインステージあたりの遅延制約を $T_{p\_constraint}$ とし，初期の $T_{p\_constraint}$ を T_{max_delay} とする．また，遅延最小制約違反値を $T_{min\_violation}$ とする．この操作を図 3.22 に示す．

P-Step2 プロセッシングユニットの根のノードにおいて，最後に実行される最小モジュールを PU_{root} とし，PU_{root} のパイプラインステージを”1”とする．PU_{root} の前に実行される最小モジュール

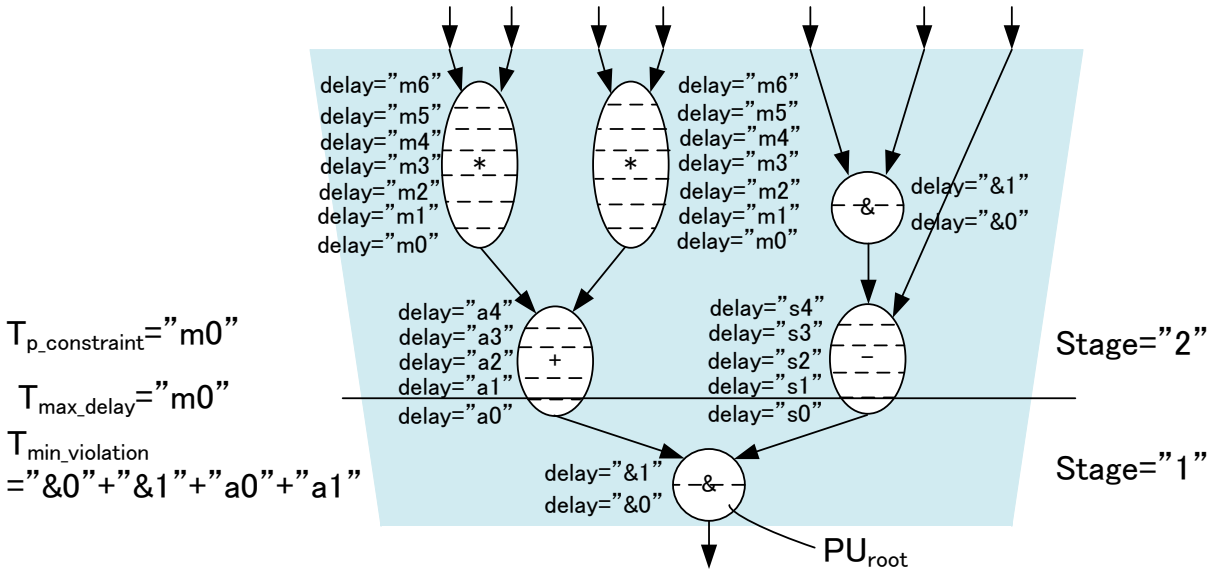

図 3.23: P-Step2 の操作

に対して，その最小モジュールと PU_{root} の遅延時間の和が $T_{p\_constraint}$ 以下であればパイプラインステージ”1”を割付ける．PU_{root} までの最小モジュールの遅延時間の和が $T_{p\_constraint}$ を超えない間，パイプラインステージ”1”を割付けていき，$T_{p\_constraint}$ を満たさなくなったら，満たさない最小モジュールに対してパイプラインステージ”2”を割付ける．また，$T_{p\_constraint}$ を満たさないもののうち，最小の遅延値を $T_{min\_violation}$ とする．この操作を図 3.23 に示す.

P-Step3 パイプラインステージ”2”を割付けた最小モジュールの前に実行される最小モジュールに対して，$T_{p\_constraint}$ を満たす間クラスタリングを行う．$T_{p\_constraint}$ を満たさなくなったら，満たさない最小モジュールに対してパイプラインステージ”3”を割付ける．さらに，$T_{p\_constraint}$ を満たさないもののうち最小の遅延値が，$T_{min\_violation}$ より小さければ，$T_{min\_violation}$ を更新する．この操作を図 3.24 に示す.

P-Step4 パイプラインステージ”3”以降についても同じ操作を繰返し，全ての最小モジュールに対してパイプラインステージを割付ける．パイプラインステージが対象とするプロセッサコアの実行ステージ数以上になった場合，$T_{min\_violation}$ を $T_{p\_constraint}$ とし，P-Step1 に戻る．

パイプラインステージが，対象とするプロセッサコアの実行ステージ数以内になれば終了する．これにより，プロセッシングユニット内のパイプライン化を実現し，クリティカルパスの最適化を行う．プロセッサコアの実行ステージ数が 3 だった場合の．プロセッシングユニット内パイプライン化を行った結果を図 3.25 に示す。

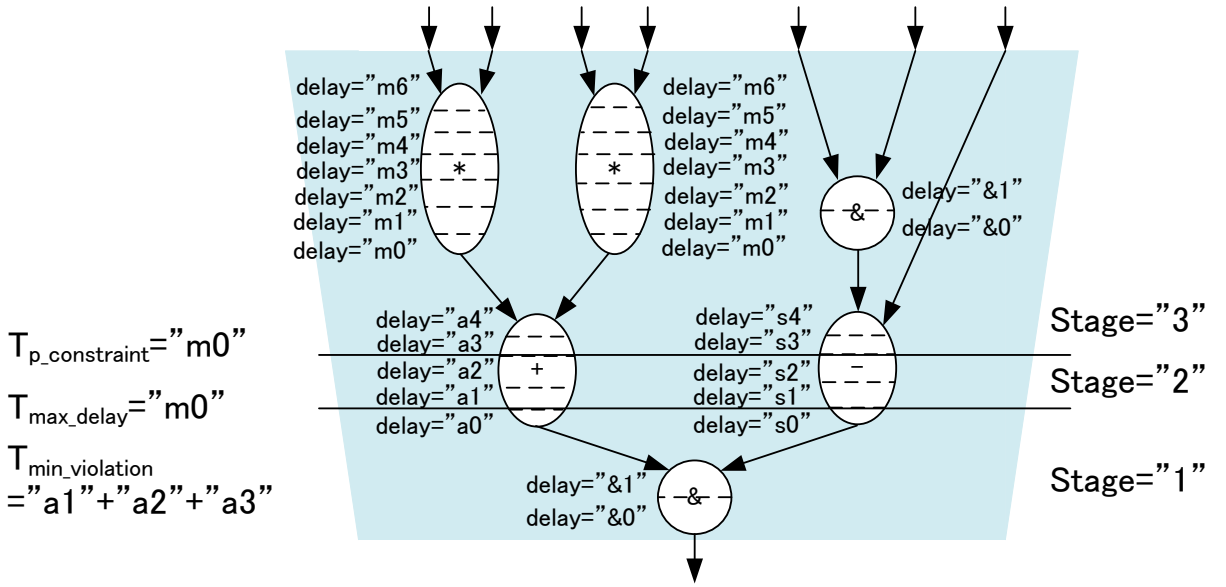

図 3.24: P-Step3 の操作

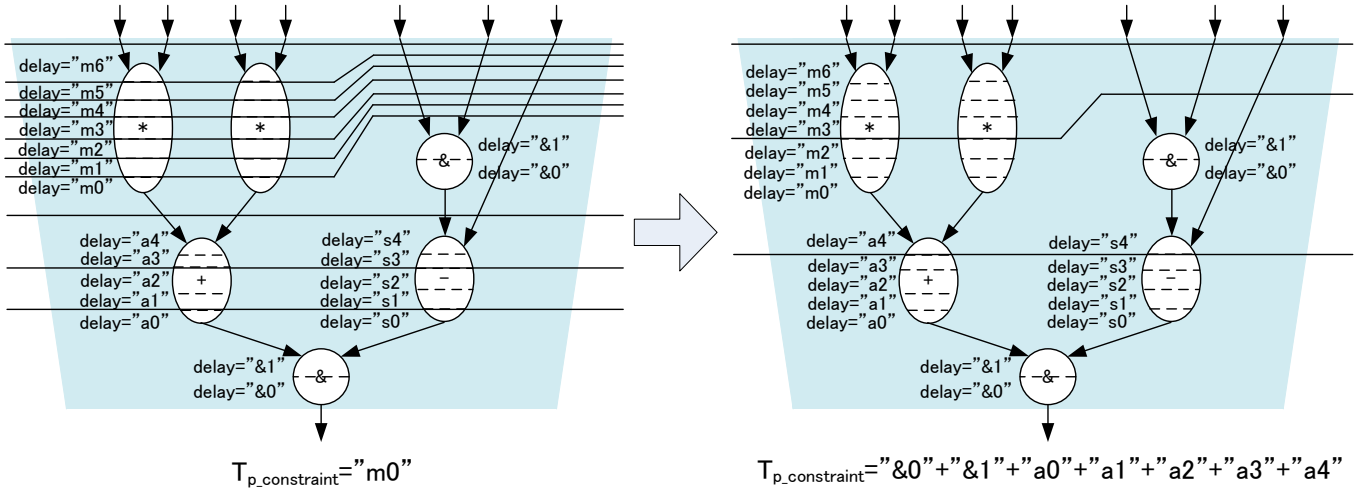

図 3.25: プロセッシングユニット内パイプライン化の様子

スケジューリング

スケジューリングでは，プロセッシングユニットが割付けられた DFG に対してリストスケジューリングを行い，DFG の実行にかかるコントロールステップ数を算出する．また，汎用レジスタを割付け，アプリケーション実行にかかるサイクル数を更新する．

3.5.2 再構成系

再構成系では，生成系にて生成されたプロセッシングユニットを削除し，新たに粒度の小さいプロセッシングユニットを生成する．アプリケーション実行時間制約 T_c に違反するまで最構成系の実行を繰返し，制約に違反したら，制約に違反する前に生成されたプロセッシングユニットの構成を解として出力する．

再構成系は，リクラスタリング，リバインディング，プロセッシングユニット内パイプライン化，スケジューリングの 4 つの工程により成り立っている．プロセッシングユニット内パイプライン化，スケジューリングについては生成系と同じであるため，リクラスタリング，リバインディングについて以降で解説していく．

リクラスタリング

リクラスタリングでは，現在生成されているプロセッシングユニットを開梱する．そして，削除されたプロセッシングユニットのノード梱包数から 1 引いた値を，次に生成するプロセッシングユニットのノード梱包数の上限とし，クラスタリング条件に追加する．その後，開梱されたプロセッシングユニットに対して，クラスタリングを行う．

リバインディング

リバインディングでは，アプリケーションの CDFG の中で削除されたプロセッシングユニットが割付けられているノードに対して，バインディングを解除し元の CDFG に戻す．その後，新たに生成されたプロセッシングユニットのバインディングを行う．

3.6 本章のまとめ

本章では，部分マッチングを考慮し MISO 構造に対応したプロセッシングユニット最適化手法について解説した．

3.2 節**プロセッシングユニットの構成**では，対象となるプロセッシングユニットの構成について解説した．

3.3 節**問題定義**では，プロセッシングユニット最適化における問題定義を行った．

3.4 節**命令定義**では，プロセッシングユニットが対象とする命令の定義を行った．

3.5 節**アルゴリズム**では，提案手法におけるアルゴリズムを解説した．提案手法を構成する各工程について，3.5.1 項**生成系**，3.5.2 項**再構成系**で説明を行った．

第4章

計算機実験

4.1 本章の概要

本章では，3 章で提案した，部分マッチングを考慮し MISO 構造に対応したプロセッシングユニット最適化手法について，計算機実験を行い評価する．まず，実験に用いる計算機環境や実験を行う際の条件について説明する．ターゲットアプリケーションには，アルファブレンドを使用する．その後，アプリケーションの CDFG に対して各条件でプロセッシングユニットを生成し，既存手法との結果を比較する．また，実験結果と提案手法に対する考察を行い，今後の課題について述べる．

4.2 実験方法

提案手法を評価するために，計算機実験を行った．計算機実験に用いた計算機環境は，OS に Debian sarge，CPU に Intel Core2 1.86GHz，コンパイラに gcc(version 3.3.5)，メモリを 1GB 使用した．入力アプリケーションとしてアルファブレンドを与えた場合において，MISO 構造で部分マッチングを考慮した場合，MISO 構造で部分マッチングを考慮しない場合，木構造で部分マッチングを考慮した場合 (文献 [8]) でそれぞれプロセッシングユニットを 1 個生成し，プロセッシングユニットの面積およびアプリケーション実行時間を比較した．なお，今回の実験では特にアプリケーション実行時間制約は与えず，生成系において実行時間が最速となるプロセッシングユニットを生成し，その後再構成系において粒度の小さいプロセッシングユニットを生成していった．プロセッシングユニットを付加する前のアプリケーション実行時間は 8.1ms だった．また，プロセッシングユニットを付加する前のプロセッサコアにおける演算器面積の合計は $102553\mu m^2$ だった．プロセッサのパイプライン構成における実行ステージ数は 3 に設定し，遅延係数 $d_m = 3$ とした．合成したプロセッシングユニットの面積/遅延見積もりには，文献 [13] を用いた．

表 4.1: 実験結果

	MISO/部分マッチング○			MISO/部分マッチング×			tree/部分マッチング○ (文献 [8])		
ノード梱包数	A_{pu} [μm^2]	T_{app} [ms]	T_e (%)	A_{pu} [μm^2]	T_{app} [ms]	T_e (%)	A_{pu} [μm^2]	T_{app} [ms]	T_e (%)
6	24401	6.27	29.4	24401	7.00	15.8	–	–	–
5	24345	6.64	22.2	24345	7.37	10.0	12313	7.37	10
4	24289	7.00	15.8	24289	7.37	10.0	12257	7.74	4.8
3	24232	7.74	4.8	24232	7.37	10.0	13400	7.74	4.8
2	12144	7.74	4.8	12144	7.74	4.8	113	7.74	4.8

A_{pu}：プロセッシングユニットの面積，T_{app}：アプリケーション実行時間，T_e：実行時間削減率 $\left(\frac{T_b - T_a}{T_a}\right)$

4.3 実験結果

プロセッシングユニットの面積 $A_{pu}[\mu m^2]$，アプリケーション実行時間 T_{app}[ms]，実行時間削減率 $T_e = \left(\dfrac{T_b - T_a}{T_a}\right)$ をまとめたものを表 4.1 に示す．ただし，T_b はプロセッシングユニット付加前のアプリケーション実行時間，T_a はプロセッシングユニット付加後のアプリケーション実行時間である．MISO 構造のプロセッシングユニットに関しては，ノード梱包数が 6 のものは生成系において生成され，その後再構成系において，ノード梱包数を 1 ずつ減らしたものを生成した．木構造のプロセッシングユニットに関しても同様に，ノード梱包数が 5 のプロセッシングユニットを生成系にて生成した後，再構成系で粒度の小さいプロセッシングユニットを生成した．

提案手法では，最大で 29.4%の実行時間削減率を実現し，MISO 構造で部分マッチングを考慮しなかった場合 (15.8%)，木構造で部分マッチングを考慮した場合 (10%) と比べて，それぞれ 86%，194%の改善が見られた．

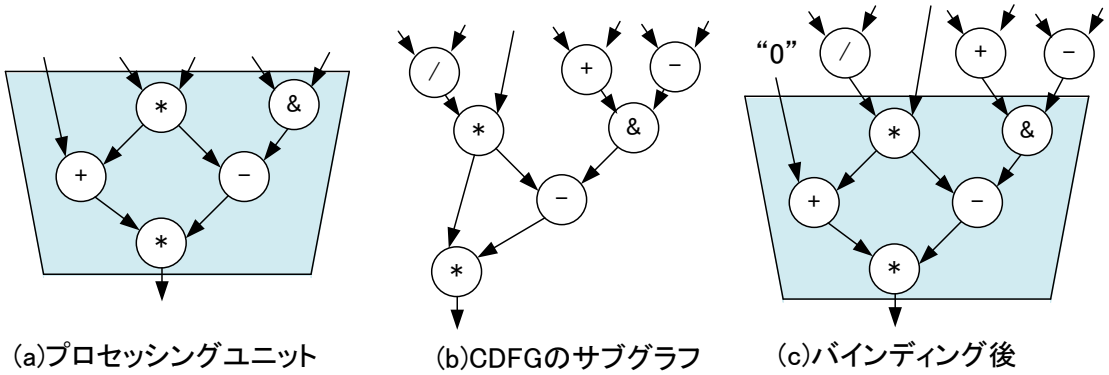

図 4.1: 部分マッチング可能な場合

4.4 考察

既存手法と比較して，提案手法は実行時間削減率で上回っており，提案手法の有効性を示すことができた．しかし，ノード梱包数が 3 のとき，MISO 構造で部分マッチングを考慮した場合とそうでない場合を比較すると，部分マッチングを考慮していない方が実行時間の削減が大きくなった．これは，部分マッチングにより実行時間が増大してしまったためだと考えられる．部分マッチングでは，プロセッシングユニットと CDFG のサブグラフのノードが 2 つ以上一致している場合，バインディングを行う．しかし，部分マッチングの場合も，そのノードの実行にかかるレイテンシはプロセッシングユニットの全体のレイテンシと同じになるので，汎用演算器で実行した場合よりも実行時間が長くなってしまう可能性がある．この場合，この演算ノードが実行されるコントロールステップにおいて，汎用演算器が他の演算ノードに割り当てられていなければ，部分マッチングを行わずに汎用演算器で実行したほうが望ましい．

提案手法では，プロセッシングユニットの中に出力エッジが複数ある演算ノードを含む場合，そのノードがバインディング条件を満たさなければ，バインディング全体が失敗したとみなし，部分マッチングも行わないことにしている．ただし，実際には複数の出力エッジを持つノードがバインディング条件を満たさなくても，部分マッチングによるバインディングが可能な場合がある (図 4.1)．今後の課題として，バインディング条件を満たさなくてもバインディングが可能なケースを考慮し，そのようなサブグラフを探索する必要がある．

またクラスタリングにおいて，プロセッシングユニットの候補となるパターンの中から，できるだけ演算ノードの梱包数が多いプロセッシングユニットを採用し，梱包数が同じ場合はスロットの占有率が低いものを優先しているが，スロットの占有率が低いプロセッシングユニットが，必ずしもアプリケーション実行時間の削減により大きく貢献するとは限らない．図 4.2 に示したノード集合がプロセッシングユニット候補として存在した場合，どちらがより実行時間の削減につながるかどうかを考慮していない．

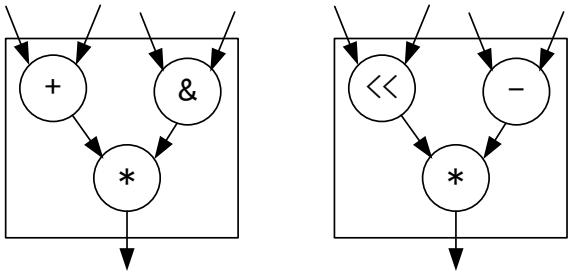

図 4.2: ノード梱包数，スロット占有率が等しいノード集合

また，提案手法では生成するプロセッシングユニットの個数を 1 つに限定していたが，今後の課題として，複数のプロセッシングユニットを生成する場合を考慮し，実行時間制約を満たし面積が最小となるプロセッシングユニットのセットを探索する必要がある．ただし，複数あるプロセッシングユニットのうちどちらでもバインディングが可能な DFG のサブグラフが存在する可能性が考えられ，解の探索が困難になることが予想される．

4.5 本章のまとめ

本章では，提案手法の評価実験を行い，結果について考察した．

4.2 節**実験方法**では，計算機実験に用いた計算機環境について述べた．また，実験を行う際の条件や他の手法との比較方法などについて説明した．

4.3 節**実験結果**では，実験結果を表に示した．また，他の手法でプロセッシングユニットを生成したときと比較して，提案手法によって生成されたプロセッシングユニットの方がアプリケーション実行時間の削減が大きいことを示した．

4.4 節**考察**では，4.3 節で示した実験結果について考察した．また，提案手法の問題点についてまとめ，今後の課題について述べた．

第5章

関連研究

5.1 本章の概要

本章では，専用演算器合成に関する既存手法について紹介する．

はじめに，文献 [8] による，SPADES におけるプロセッシングユニット最適化に関する既存手法について紹介する．その後．その他の専用演算器合成に関する既存手法 [1, 2, 4, 14, 15, 16, 17] について解説する．

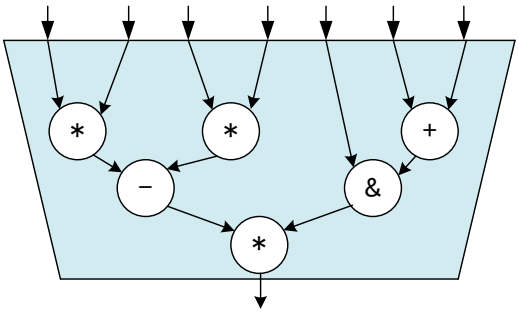

図 5.1: 木構造のプロセッシングユニット

5.2 プロセッシングユニット最適化に関する既存手法

文献 [8] では，木構造のプロセッシングユニットを自動合成する手法を提案している．対象となる木構造のプロセッシングユニットを図 5.1 に示す．

本論文の提案手法は，文献 [8] の改良であり，対象となるプロセッシングユニットの構造が木構造から MISO 構造に拡張された点で優位性がある．生成系において実行速度が最速のプロセッシングユニットを生成し，再構成系でプロセッシングユニットの粒度を小さくしていく点など，アルゴリズムの全体の流れは提案手法と同じである．また，文献 [8] では部分マッチングを考慮している．

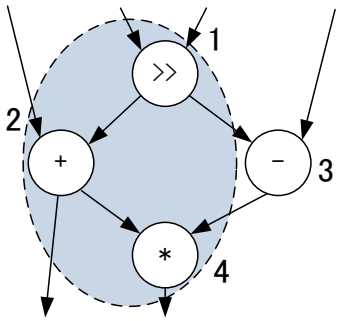

図 5.2: コンベックスでないサブグラフ

5.3 その他の専用演算器合成に関する既存手法

本節では，その他の専用演算器合成に関する既存手法について紹介する．

文献 [1] の手法

文献 [1] では，深さ優先探索を用いた，コンベックスな構造を持つ専用演算器の合成手法を提案している．コンベックスとは，サブグラフ内の 2 つのノードの間に，サブグラフに含まれないノードを経由する経路が存在しないことである．図 5.2 のサブグラフでは，ノード 1 からノード 4 への経路を考えた時，サブグラフに含まれていないノード 3 を経由する，ノード 1→ ノード 3→ ノード 4 の経路が存在するため，コンベックスではない．

文献 [2] の手法

文献 [2] では，MISO 構造の専用演算器を自動合成する手法を提案している．サブグラフ p に対して，ソフトウェア実行時間 $T_{sw}(p)$ とハードウェア実行時間 $T_{hw}(p)$ を求め，以下の式でゲインを求めている．

$$Speedup(p) = \frac{T_{sw}(p)}{T_{hw}(p)} \tag{5.1}$$

$$Gain(p) = Speedup(p) \times Occurrence(p) \tag{5.2}$$

ただし，$Occurrence(p)$ は DFG 中のサブグラフ p の出現回数である．また，サブグラフのマッチングに文献 [7] を用いている．面積制約を満たし，実行時間が最小となる専用演算器の集合を探索する．

複数の専用演算器の合成を考慮し，全体の最適化を図っている点で優れているが，専用演算器と DFG のサブグラフとの部分マッチングを考慮していない．

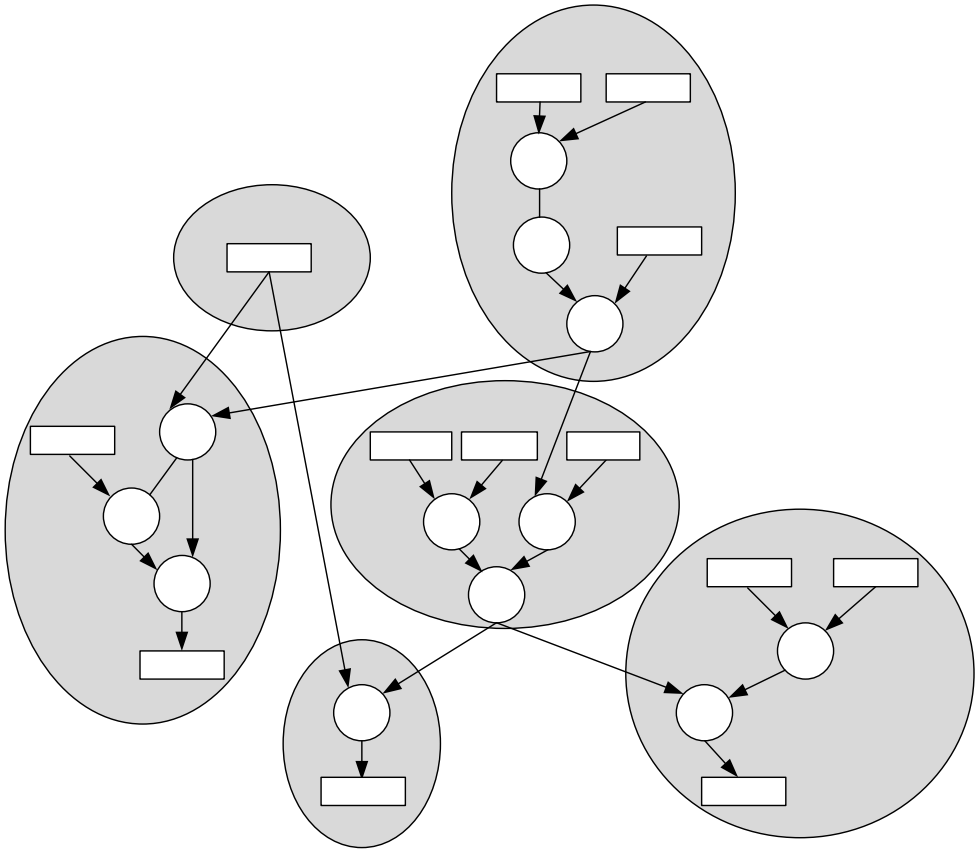

図 5.3: MISO 構造に分割された DFG

文献 [4] の手法

文献 [4] では，MIMO (Multiple Input,Multiple Output) 構造の入力アプリケーションに対し，最大の MISO 構造のサブグラフで分割し，その後そのサブグラフをマージすることで MIMO 構造の専用演算器を生成している．図 5.3 のようにアプリケーションを MISO 構造のサブグラフに分割し，それぞれのサブグラフを専用演算器で実行するか汎用演算器で実行するか選択する．また，サブグラフ同士をマージすることで，MIMO 構造の専用演算器も合成することができる．

MIMO 構造の専用演算器にも対応しているが，部分マッチングを考慮していない．また，専用演算器の粒度が自然と大きくなってしまい，他のアプリケーションへの柔軟性に欠けるという欠点が見られる．

文献 [14] の手法

文献 [14] では，コンベックスな専用演算器の合成手法を提案している．階層アルゴリズムを用いることで，文献 [1] よりも高速な探索を図っている．それ以外については，文献 [1] と目立った差異は見られない．

文献 [15] の手法

文献 [15] では，MIMO 構造のカスタム命令を全て列挙する手法を提案している．convex なサブグラフを列挙し，それらをマージすることで，MIMO 構造に対応させている．しかし，生成した専用演算器と CDFG のサブグラフのマッチングは行っておらず，あくまで生成可能な専用演算器を列挙するだけにとどまっている．

文献 [16] の手法

文献 [16] では，DFG 中のサブグラフの出現回数を考慮した専用演算器合成手法を提案している．深さ優先探索を用いて，木構造の専用演算器を合成している．複数の専用演算器の最適化を図っているが，木構造にしか対応しておらず，部分マッチングも考慮していない．

文献 [17] の手法

文献 [17] では，与えられたカスタム命令セットを実行する専用演算器の合成手法を提案している．複数のカスタム命令を実行できる専用演算器を生成することで，面積の削減を図っている．ただし，複数のカスタム命令を 1 つの専用演算器で共有すると，単一のカスタム命令を実行する専用演算器と比較して実行時間が長くなってしまう．また，アプリケーション中で専用演算器の使用される回数が増えるわけではない．

5.4 本章のまとめ

本章では，専用演算器合成に関する既存手法について紹介し，解説した．

5.2 節**プロセッシングユニット最適化に関する既存手法**では，既存のプロセッシングユニット最適化手法 [8] を紹介した．文献 [8] の手法は，部分マッチングを考慮しているものの，木構造のプロセッシングユニットにしか対応していなかった．

5.3 節**その他の専用演算器合成に関する既存手法**では，その他の専用演算器合成に関する既存手法 [1, 2, 4, 14, 15, 16, 17] を紹介した．いずれの手法も専用演算器と CDFG のサブグラフの部分マッチングを考慮していなかった．

第6章

結論

本論文では，部分マッチングを考慮し MISO 構造に対応した，プロセッシングユニット最適化手法を提案した．また，既存手法との比較を行い，提案手法の優位性を示した．

第 2 章 **SIMD 型プロセッサコア向け HW/SW 協調合成システム：SPADES** では，まず SPADES の対象とする SIMD 型プロセッサコアのアーキテクチャモデルについて説明した．そして，SPADES の構成要素である「アプリケーション解析」,「並列化コンパイラ」,「ハードウェア/ソフトウェア分割」,「ハードウェア生成」および「ソフトウェア生成」についてそれぞれ解説した．

第 3 章**部分マッチングを考慮し MISO 構造に対応したプロセッシングユニット最適化手法**では，まずプロセッシングユニットの構成について述べた．また，プロセッシングユニット最適化の問題定義を行い，プロセッシングユニットが対象とする命令について定義した．その後，部分マッチングを考慮し MISO 構造に対応したプロセッシングユニット最適化手法を提案した．提案手法は生成系と再構成系に分けられ，生成系の構成要素であるクラスタリング，バインディング，プロセッシングユニット内パイプライン化およびスケジューリングと，再構成系の構成要素であるリクラスタリング，リバインディング，プロセッシングユニット内パイプライン化およびスケジューリングについてそれぞれ解説を行った．

第 4 章**計算機実験**では，提案手法を計算機上に実装し，シミュレーションを行った．その結果，プロセッシングユニット付加前と比べてアプリケーション実行時間が最大 29.4%削減された．また，MISO 構造で部分マッチングを考慮しない場合 (最大 15.8%)，木構造で部分マッチングを考慮した場合 (最大 10%) と比較し，それぞれ 86%，194%の改善が見られた．これにより，単一のプロセッシングユニットを付加した場合において，提案手法によるプロセッシングユニットが最大の実行時間削減を実現していることを示すことができた．

第 5 章**関連研究**では，文献 [8] で提案されている，プロセッシングユニット最適化に関する既存手法について説明を行った．その後，専用演算器合成手法に関するその他の研究 [1, 2, 4, 14, 15, 16, 17] について紹介した．

今後の課題として，以下の 3 つが挙げられる．

1. **プロセッシングユニット候補の選択手法の改良**

   提案手法ではクラスタリングにおいて，プロセッシングユニットの候補となるパターンの中から，できるだけ演算ノードの梱包数が多いプロセッシングユニットを採用し，梱包数が同じ場合はスロットの占有率が低いものを優先する．しかし，これらの条件を満たすプロセッシングユニットが，必ずしもアプリケーション実行時間の削減により大きく貢献するとは限らない．

   また，図 6.1 に示した 2 つのプロセッシングユニット候補から 1 つを選択してプロセッシングユニットを生成する場合，現在の手法では (a) と (b) の間に差異を見出すことができない．同じ構造を持つサブグラフの出現回数や，各 DFG の実行回数を考慮し，アプリケーション実行時間の削減により大きく貢献するプロセッシングユニットを生成するべきである．

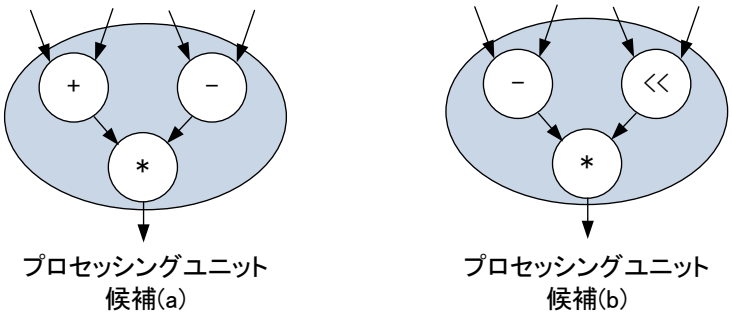

図 6.1: プロセッシングユニット候補の例

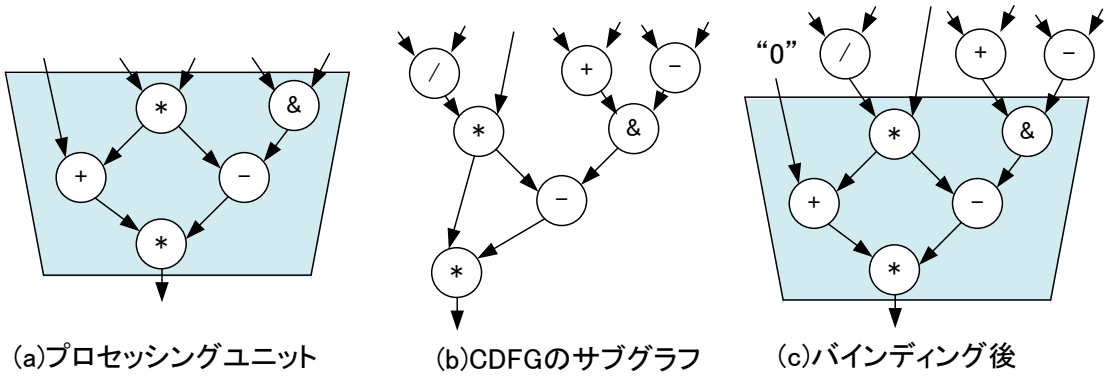

図 6.2: 部分マッチングが可能な場合

2. **出力エッジを複数持つ演算ノードを含むプロセッシングユニットのバインディング**

   提案手法では，プロセッシングユニットの中に出力エッジを複数持つ演算ノードを含む場合，そのノードがバインディング条件を満たさなければ，バインディング全体が失敗したとみなし，部分マッチングも行わないことにしている．ただし，実際には複数の出力エッジを持つノードがバインディング条件を満たさなかった場合にも，部分マッチングが可能な場合がある (図 6.2)．どのような条件でバインディングが可能なのか判別することができれば，部分マッチングが可能となりアプリケーション実行時間の削減につながると考えられる．

3. **複数のプロセッシングユニットの生成**

   本論文では，生成するプロセッシングユニットの個数を 1 個に限定した上で実験を行った．今後の課題として，生成するプロセッシングユニットの個数を限定せず，実行時間制約を満たし面積が最小となるプロセッシングユニットのセットを探索する必要がある．文献 [2] や文献 [16] で，面積制約を満たしアプリケーションの実行時間を最小にするカスタム命令セットの探索手法を提案しているが，専用演算器の部分マッチングを考慮した場合更に複雑な問題になることが予想される．

謝辞

本論文全般にわたり，御指導ならびに御助言を授かった戸川望教授，柳澤政生教授，大附辰夫教授に深く感謝いたします.

また，始終適切な御指導および御助言を頂きました本学 大智輝氏，奈良竜太氏に深く感謝いたします.

本論文を完成させるにあたりまして様々な御意見と御助言を頂きました，本学 渡辺隆行氏，新川将大氏に心より感謝致します.

最後に，本論文に関する研究活動全般にわたり支援していただいた戸川研究室，柳澤研究室および大附研究室の皆様に感謝いたします.

参考文献

[1] K. Atasu, L. Pozzi, P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," *DAC 2003*, pp. 256–261, Jun. 2003.

[2] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," *FPGA '04*, pp. 183–189, Feb. 2004.

[3] Freescale, http://www.freescale.co.jp/.

[4] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis, "Automatic selection of application-specific instruction-set extensions," in *Proc CODES+ISSS' 06*, pp. 160–165, Oct. 2006.

[5] H. Kawazu, J. Uchida, Y. Miyaoka, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "Suboperation parallelism optimization in SIMD processor core synthesis," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E88–A, no. 4, pp. 876–884, 2005.

[6] 栗原 輝，宮岡 祐一郎，戸川 望，柳澤 政生，大附 辰夫，"SIMD 型プロセッサコアの自動合成のためのパイプライン演算ユニット生成手法," 情報処理学会論文誌, vol.47, no.6, pp.1594–1607, 2006.

[7] Nauty Package, http://cs.anu.edu.au/people/bdm/nauty.

[8] 繁田 裕之，小原 俊逸，戸川 望，柳澤 政生，大附 辰夫，"SIMD 型プロセッサコア設計におけるプロセッシングユニット最適化手法," 電子情報通信学会技術研究報告. ICD, 集積回路，106(551) pp.1–6, 2007.

[9] Texas Instruments, http://www.ti.com/.

[10] N. Togawa, M. Yanagisawa, and T. Ohtsuki, "A hardware/software cosynthesis system for digital signal processor cores," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E82–A, no. 11, pp.2639–2647, 2001.

[11] N. Togawa, Y. Kataoka, Y. Miyaoka, M. Yanagisawa, and T. Ohtsuki, "Area and delay estimation in hardware/software cosynthesis for digital signal processor cores," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E84-A, no. 11, pp. 2639–2647, 2001.

[12] N. Togawa, K. Tachikake, Y. Miyaoka, M. Yanagisawa, and T. Ohtsuki, “A hardware/software partitioning algorithm for processor cores with packed SIMD-type instructions,” *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E86–A, no. 12, pp. 3218–3224, 2003.

[13] 山崎大輔，小原俊逸，戸川望，大附辰夫，“HW/SW 協調合成におけるアプリケーションプロセッサの面積/遅延見積もり手法，” 信学技報，vol. 106, no.113, VLD2006-14, pp.1–6, 2006.

[14] S. Yang, C. Lin, C. Hung, J. Wu, and Y. Wang, “Application-specific instruction generation for SOC processors,” *ISCAS 2007*, pp. 3752–3755, May. 2007.

[15] P. Yu and T. Mitra, “Scalable custom instructions identification for instruction-set extensible processors,” *CASES’ 04*, pp. 69–78, Sep. 2004.

[16] D. C. Zaretsky, G. Mittal, R. P. Dick, and P. Banerjee, “Dynamic template generation for resource sharing in control and data flow graphs,” *VLSID’ 06*, pp. 465–468, Jan. 2006.

[17] M. Zuluaga and N. Topham, “Resource sharing in custom instruction set extensions,” *SASP 2008*, pp. 7–13, Aug. 2008.

本論文に関する発表業績

研究会

1. 橋本識弘, 戸川望，柳澤政生，大附辰夫,“部分マッチングを考慮し MISO 構造に対応した専用演算器合成手法,” 信学技報, VLD2009–83, pp. 89–94, 2010.