平成 12 年度

学士学位論文

FPGA による自己タイミング型パイプライン演算機構の実現

A FPGA Implementation of Self-Timed Pipelined ALU

1010441　　古家 俊之

指導教員　　岩田 誠 助教授

2000 年 12 月 1 日

高知工科大学 情報システム工学科

要 旨

FPGAによる自己タイミング型パイプライン演算機構の実現

古家 俊之

超高性能、極省電力を達成可能な自己タイミング型パイプライン機構によるデータ駆動型VLSIプロセッサDDMPが開発され、現在、商品化の途についたばかりである。

このような状況で、さらに応用分野を拡大するためには、各応用分野にある程度特化したＤＤＭＰを迅速に設計、開発することが要請されている。

従来はシミュレーションによって設計検証をおこなっていたが、ＶＬＳＩシステムが大規模化する傾向にある昨今、検証時間が膨大になる問題がある。

本研究では、実用的規模の回路を作り込める最近のＦＰＧＡを活用して新規開発ＤＤＭＰの設計内容を短期間で検証する手法の確立を目的としている。よって、本論文では、ＦＰＧＡ上での自己タイミング型パイプライン処理機構の最適設計指針を示し、その結果をもとに最適パイプラインステージ数を導き出し算術、論理演算機構に適用し目標性能を達成できる見通しを得た。

キーワード　ＦＰＧＡ、ＤＤＭＰ、自己タイミング型パイプライン処理機構

Abstract

A FPGA Implementation of Self-Timed Pipelined ALU

Toshiyuki Furuie

A data driven VLSI processor DDMP which achieves by super-high performance and ultra low power consumption by the self-timing pipelined mechanism wasdeveloped, and it is on the middle to the way to apply verious commercial fields.

It is highly reguired to tall down the design and developing time that DDMP which specialized in each applicable field to some extent is designed and developed quickly to customise DDMP chip, becouse it is important to sell products at the high time.

The simulation method was used for design verification conventionally, but there was a problem to which verification time becomes huge these days according to the increasing size of VLSI systems.

A method which utilizes the latest FPGA which makes and puts the circuit of a practical scale, and verifies the contents of a design of target DDMP chips for a short period of time is aimed in this research.

Therefore, in this paper, the optimum design paramenters of the self-timed pipeline processing mechanism on FPGA was clearly shown, and the prospect that the number of optimum pipeline stages was drawn on the basis of the result, it applied to arithmetic and a logic operation mechanism, and a target performance could be attained was acquired.

key words ＦＰＧＡ、ＤＤＭＰ、Self-timed pipeline processing mechanism

目次

第 1 章　序論 **1**

第 2 章　データ駆動プロセッサの構造 **4**

2.1　データ駆動プロセッサの動作 4

2.2　各ブロックの説明 6

2.2.1　M（Flow Merging Module）部 6

2.2.2　MM（Matching Memory）部 6

2.2.3　ＦＰ（Functional Processing Unit）部 6

2.2.4　ＣＰＳ（Cache Program Storage）部 7

2.2.5　D （Flow Diverting Module) 7

第 3 章　論理的な最適設計指針の検証 **8**

3.1　ゲート数とゲート段数 8

3.1.1　ゲート数とゲート段数の性質 8

3.1.2　共有化によるゲート数の減少 9

3.1.3　共有化の不可能な例 9

3.2　パイプラインステージ 10

3.2.1　パイプラインステージ化のスループットの上昇 11

3.2.2　ゲート段数の違いにより生じるスループットの変化 12

3.2.3　パイプラインステージ数とスループットの関係 16

3.2.4　パイプラインステージ数とハードウェア領域の関係 18

3.2.5　パイプラインステージ数の論理的最善数 19

第 4 章　FP(Functional Processing Unit) 部の設計と構造 **20**

4.1　設計指針 20

4.2 FP（Functional Processing Unit）部の構造 20
4.2.1 C(Self-timing transfer control circuit) 21
4.2.2 Operation 部 22
論理演算 23
シフト（Logical Shift） 23
算術演算 23
比較 23
条件分岐 24
世代操作 24
その他 25
4.2.3 DECODE 部 25
4.2.4 SELECT 部 26
4.3 設計 26
4.3.1 設計指針１－共有化 26
4.3.2 設計指針２－パイプライン化 28
4.3.3 再利用への対応 29

第 5 章 結論 **31**

謝辞 **32**

参考文献 **33**

付録 A FP 部の各命令のアルゴリズム **34**
A.1 論理演算 34
A.2 シフト 34
A.3 算術演算 34
A.4 比較 35

A.5 条件分岐 36
A.6 世代操作 37
A.7 その他 37

図目次

2.1 プログラム 4

2.2 Basic structure of data-driven processor 5

2.3 パケットフォーマット 5

3.1 シンボル内の段数の決定 8

3.2 共有化の例 9

3.3 共有化の不可能な例 10

3.4 パイプラインステージ 10

3.5 ステージ数 11

3.6 パイプラインステージ２段 12

3.7 パイプラインステージ数ｎ個 14

3.8 $y = -\frac{T+a}{a^2n+-(a-T)a} + \frac{1}{a}$ のグラフ 17

3.9 $y2 = -\frac{1}{kn+H+1}$ のグラフ 18

3.10 スループット性能の上限 19

4.1 FP 部の構造図 21

4.2 入力パケット (上)、出力パケット (下) 21

4.3 DECODE 部の構造 25

4.4 共有化をおこなった命令 26

4.5 最適パイプライン 29

A.1 ABS の構造図 36

A.2 GE の構造図 37

表目次

4.1 Op.code 表 1 22
4.2 Op.code 表 2 22
4.3 独自アルゴリズム 28
4.4 SUB 出力 28

A.1 Over Flow 35

第１章

序論

IT 革命といわれる近年、インターネットの普及に伴いコンピューターのソフトウェア、ハードウェアをはじめ関連分野の研究、発展は目まぐるしいものとなり、従来使用していたものも数多く新しいものへと変化している。その数多くの変化の１つに非ノイマン型プロセッサの開発が挙げられる。

従来使用されていたノイマン型コンピュータは、１９４５年、フォアノイマンによって提案された。これはコンピュータの主記憶装置（現在のＲＡＭに相当している部分）の中にデータだけではなくて、プログラムも一緒に入れられる方法である。この方法を擁したプログラム内蔵型コンピュータでは内臓されたプログラムに従って、命令を１つずつ順番にＣＰＵ（中央演算処理装置）に、プログラムカウンタという一種の制御装置を用いて呼び出して実行する。しかし、この制御装置は各プロセッサに１つしかないために命令は１度に１つづつしか読み出せないという弱点を持つ。このような方法を使用しているプロセッサををノイマン型プロセッサといい、ノイマン型プロセッサを擁すコンピュータをノイマン型コンピュータという。またこれらは今日までコンピュータの基本となっており現在使われているコンピュータのほとんどがノイマン型である。

一方、非ノイマン型では、プログラムはデータが入力されることによって読み出され、データが複数入力されれば複数のプログラムが読み出されるため、並列にデータ処理が行える、このため高速処理が可能になり画像認識など高度な処理もリアルタイムで実行することができる。。また、データに制御機能をもたせて、演算部にデータが揃った時だけ作動するクロックレス構造により、低消費電力でプロセッサを動作させることができる [2]。以上のことより、非ノイマン型並列処理方式であるデータ駆動プロセッサに焦点をあてる。

ここでデータ駆動プロセッサの簡単な歴史について触れる。非ノイマン型の並列処理プロセッサは、当初超高速コンピュータを実現するために開発された。シャープでは１９８３年、民生機器の高機能化、小型化、および低価格化を目指し、開発に着手した。その後、世界で初めての非ノイマン型データ駆動方式により、映像、画像、音声などの膨大なマルチメディア情報を、超低消費電力で高速・並列処理する、次世代のデータ駆動メディアプロセッサ＜ＤＤＭＰ＞を開発した。[2]

では、そのデータ駆動プロセッサを作成するとき、実際のカスタムＬＳＩ上で作成をおこなうには、時間的にも経済的にも無理であるうえリスクが伴うという問題がある。そこで登場するのがＦＰＧＡである。

FPGA（Field Programmable Gate Array）とは、ハードウェアを設計データによって、動的に（使用時に）作り変えることができるゲート・アレイである。FPGA にはいろいろなタイプがあるが、ここでは SRAM（スタティック・ラム）型を例にとって概念的な説明をすると、外部から FPGA に書かれた情報によって、多数の論理ブロックを一時的に結線し動作させるものである。まず、普通のコンピュータを使って回路をハードウェア記述言語により記述し、コンパイルし、構成データを作る。動作のシミュレーションが OK となれば、構成データは FPGA へダウンロードされる（構成データは普通の SRAM へ書き込むように FPGA へ書き込まれる）。構成データにしたがって、あらかじめ FPGA 内部に用意されている種々の論理ゲート、論理関数、マクロなレジスタや論理ブロックが結線される。実行段階に入ると、FPGA 中に構成されたものは、入力ピンからの信号を受けて論理動作を行ない出力ピンに結果を出力する。カスタムチップや ASIC のように永続的なハードウェアが構成されるのではなく、SRAM に書かれた構成データに応じて一時的に回路ができているだけである。したがって、電源を切れば、回路はリセットされる。

このように FPGA 中に構成された回路は、ハードウェアかソフトウェアかは一概に言えない。もしコンピュータのメモリ中の情報をソフトウェアであると定義すれば、これはソフトウェアだし、回路が普通のハードウェアと同じように演算を実行するという意味ではハードウェアとも言える。最近は、内部の論理ブロックはますますマクロ化し、また実行時に部

分的な再構成も可能なものが市場に現れている。[1]

また、従来のハードウェアは高価なので、どうしても応用問題毎に（専用的に）作るわけにはいかない。しかしFPGAを使えば専用マシンを安価で作り込めるのである。

これまでに、自己タイミング型パイプライン機構によるデータ駆動型VLSIプロセッサが開発され、現在、商用化の途についたばかりである。このような状況で、さらに応用分野を拡大するためには、各応用分野にある程度特化したDDMP（Data－Driven Media Processor）を迅速に設計・開発することが重要になってくる。従来は、シミュレーションによって、設計内容を検証したがVLSIシステムが大規模化する傾向にある現状では、時間がかかりすぎる。したがって本研究では、前述したように実用的規模の回路を作り込める最近のFPGAを活用して新規開発ＤＤＭＰの設計内容を短期間で検証する手法の確立を目的としている。

本論文ではＦＰＧＡ上での自己タイミング型パイプライン処理機構の最適設計指針を示し、算術論理演算機構に適用した。

以下２章ではデータ駆動プロセッサの構造について記述し、３章ではハードウェア、スループットを中心に論理的に最善策を検証する。また４章ではＦＰ（Functional Processing Unit）についての構造について述べ、３章での論理的最善策に従って設計をおこなう。

第 2 章

データ駆動プロセッサの構造

2.1　データ駆動プロセッサの動作

データ駆動プロセッサでは、図２．１で示すようなデータ駆動プログラムの演算を行なうことが可能である。これらはあるノードからその後続ノードへの有向枝（アーク）のリストから構成されており、パケットデータのヘッダ部分に連結される。

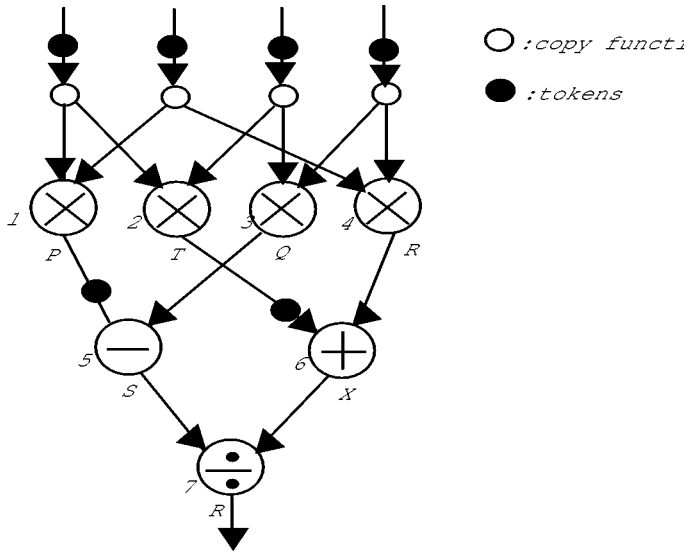

図 2.1　プログラム

図２．２は基本的なデータ駆動プロセッサの構造を示している。まず外部よりデータパケット (図２．３参照) が入力され、その後適切な合流モジュール (M : Flow Merging Module) を通過して、パケットは待ち合わせ記憶部 (MM : Matching Memory) に至る。ここでは、世代照合を行ない対になるパケットが到着するまで保持される。もし、ここで対になるパケットが到着すれば２つのパケットは Dest node(行き先ノード),Generation id(世

代番号),Operation code(オペレーションコード), とデータ同士を結合し１つのパケットを生成し関数的処理ユニット（FP : Functional Processing Unit）に送られる。ここではOperation Code で指定された演算の結果とそれに伴うデータを適切に書き換える。その後ＣＰＳに送られる。

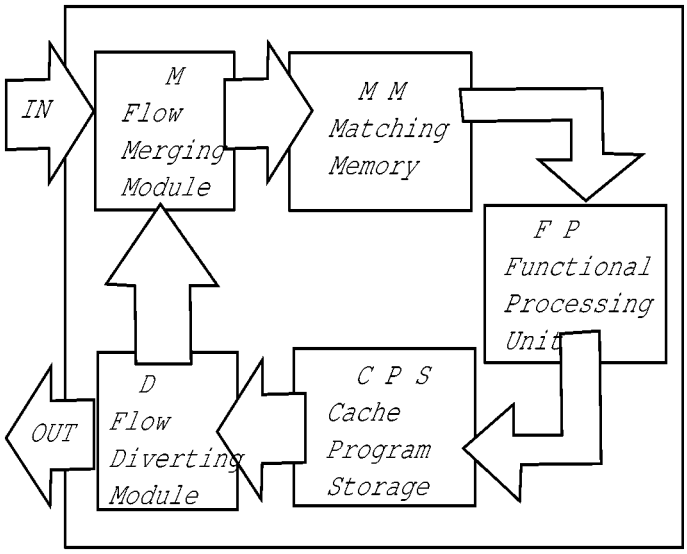

図 2.2　Basic structure of data-driven processor

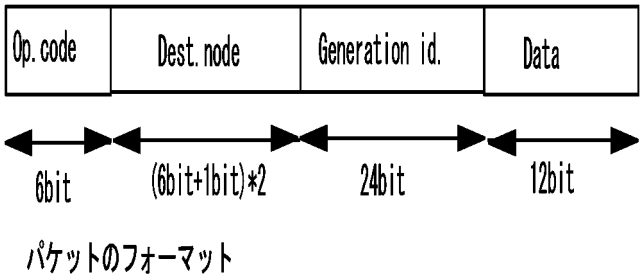

図 2.3　パケットフォーマット

ＣＰＳでは図２．１のようなプログラムからOperation Code、Dest node、定数フラグなどがCPS メモリ内に保持されており、送られてきたパケットの次の行き先を書き換える。このときのパケットの形式は入力パケットと同様の形式である。最後に分流モジュール（D : Flow Diverting Module ）に送られ行き先に適した場所へと送られる。

2.2 各ブロックの説明

では、次に各ブロックについて構造を簡単に説明する。

2.2.1 Ｍ（Flow Merging Module）部

合流要素Mでは２つの独立な自己同期パイプラインから単一な自己同期パイプラインを通るパケットの流れへ、２つのパケット流を結合する。合流要素Mに到達するパケットの流れは、最初に来たものから処理される。もしほぼ同時に２つのパケットが到着したら、２つの衝突発進信号が、調停操作回路での飽和反応型ＳＲフリップフロップの１連の動作によって調停される。このブロックでは、２つのゲート信号のうち１つを、ラッチパルスとＳｅｎｄ－Ｏｕｔ信号を作動させるようにするので、最初のパケットは、調停信号によって２つのデータパスを多重通信式にしてインプットパスからアウトプットパスに移される。その時、ラッチパルスは調停操作回路にフィードバックされて、他のパケットはアウトプットパスに伝えられる。

2.2.2 ＭＭ（Matching Memory）部

ここでは、前述の合流要素Mより送られてきたパケットの Generation.id(世代番号) を判断基準として２つのパケットを１つにする。Generation.id(世代番号) が同一のパケット同士を１つにし、あらかじめプログラムを読み込み記憶しているメモリにアドレス、データ、読み込み信号を送り、Dest.Node（行き先ノード）Op.code（オペレーションコード）を読み取り、それらを付け加えて次のブロックへと送られる。

2.2.3 ＦＰ（Functional Processing Unit）部

設計や構造については、４章で詳細を示すが簡単に説明をおこなう。ＦＰ部ではプログラム上でいう演算処理を行なう。本研究では演算処理の基本命令は、AND、OR、EXOR、NAND、NOR、ENOR、ADD、SUB、ABS、DIF、SYNC、NOP、SGN、AGN、BIT、

SWBIT、EQ、SWEQ、GE、SWGE、GT、SWGT、LSH、MAX、MIN の全２５種を実現した。(詳細は４章、付録Ａを参照)。パケットの流れとしては、ＦＰ部への入力パケットより Op.code（オペレーションコード）で指定された演算の結果を出力しＣＰＳへと送る。

2.2.4 ＣＰＳ （Cache Program Storage）部

ＦＰ部より流れてきたパケットより Dest.node(行き先ノード) を読み取り、ＭＭ（Matching Memory）と同様にメモリにアドレス、読み取り信号を送り古い Dest.node は新しい Dest.node に置きかえられる。したがって、出力パケットはデータ駆動プロセッサへの入力パケットと同一形式で生成される。

2.2.5 Ｄ （Flow Diverting Module)

分流要素 D は、２つの自己同期パイプラインにパイプラインを分ける。２つのアウトプットパイプライン間の切り替えは、インプットパケットに含まれている１つ以上の特定のフラグによりコントロールされる。もしすべてのフラグがアクティブなら、分流要素Ｄは２つのパイプラインにパケットの流れを複写する。

第 3 章

論理的な最適設計指針の検証

3.1　ゲート数とゲート段数

ここでは、主にＦＰ部内部の基本命令の設計に関わるゲート数とゲート段数について検証する。また、共有化の説明をおこない、基本命令の１つであるＧＥのアルゴリズムの決定について最善策を模索する。

3.1.1　ゲート数とゲート段数の性質

ゲート数が増加すればするほど、ハード量は増加する。このときゲート段数が増加しなければ（既存のゲートとの並列処理ならば）スループット（時間効率）に変化は無い。

ゲート段数とは並列になっているゲートの段数であり、ゲート段数が増加するとスループットは低下する。また、１つのシンボル（モジュール）内で２つの並列な処理が行なわれる場合（図３．１参照。）、Ａのゲート段数とＢのゲート段数を比べたときにゲート段数が多い方に依存されるので、この場合はＢに依存されこのシンボルのゲート段数は４と最後のＡＮＤゲートを加えた値となるので５となる。

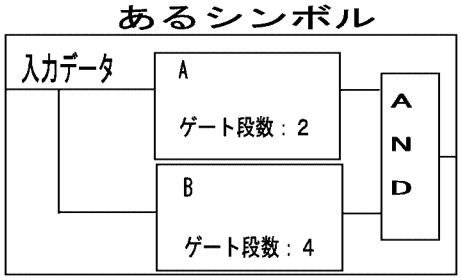

図 3.1　シンボル内の段数の決定

3.1.2　共有化によるゲート数の減少

並列処理の共有化を行ない無駄なハード領域（ゲート数）を減少させる。ABS を例に挙げて説明する。図３．２で示すように共有化前の ABS 命令では ADD 後残りの演算をおこなうが、ADD での結果を ABS の入力とすれば今まで使用していた ADD のハード領域の無駄が省ける。これによって、ゲート数の減少をはかる。ただし、このことによって図中の ABS は ADD の出力結果がないと出力することができなくなる。よって、ABS の出力結果を出すのに必要なゲート段数は、アルゴリズムに違いが無ければ共有化する前と後とでの変化は見られないということである。

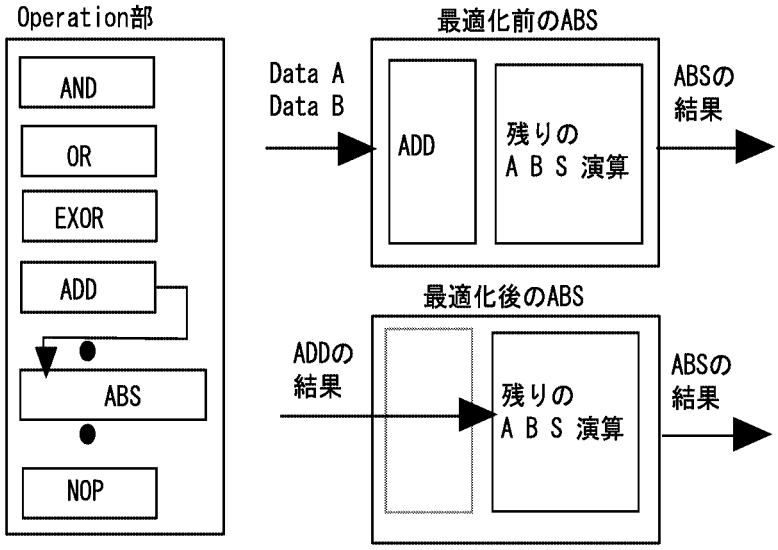

図 3.2　共有化の例

3.1.3　共有化の不可能な例

共有化ができないときの例を挙げる。図３．３に示すようにある演算を行いその演算結果同士を ADD（加算）する命令があるとする。このとき、ある命令内で必要な値は演算後の Data A、B の値（図中の Data A’、B’）であり、DATA A,B の ADD の値は不必要である。であるから、このような状況のときには共有化は行なえないのである。以上のことより一般的に共有化が行なえるのは、入力データの演算結果が使用できる部分であるといえる。

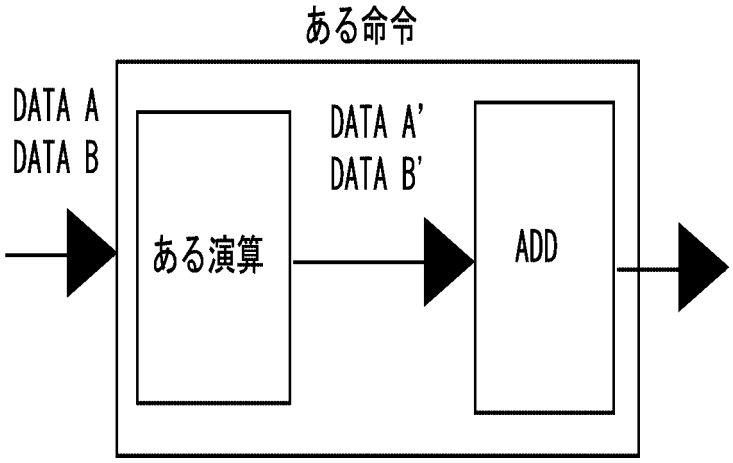

図 3.3　共有化の不可能な例

3.2　パイプラインステージ

一般的にパイプラインステージ数を増加させれば、スループットとハードウェア領域が増加する。（詳細は３．３．１パイプラインステージ化のスループット上昇を参照）では、スループットが最も良く更にハードウェア領域が少なくてもすむ最善のパイプラインステージ数は幾つなのか、具体的数字として論理的に検証する。

パイプラインステージの構造について簡単に説明する。パイプラインステージとは、ＦＰ内部でいえばＣエレメントと Data Latch 同士の区間のことである。(図３．４参照)

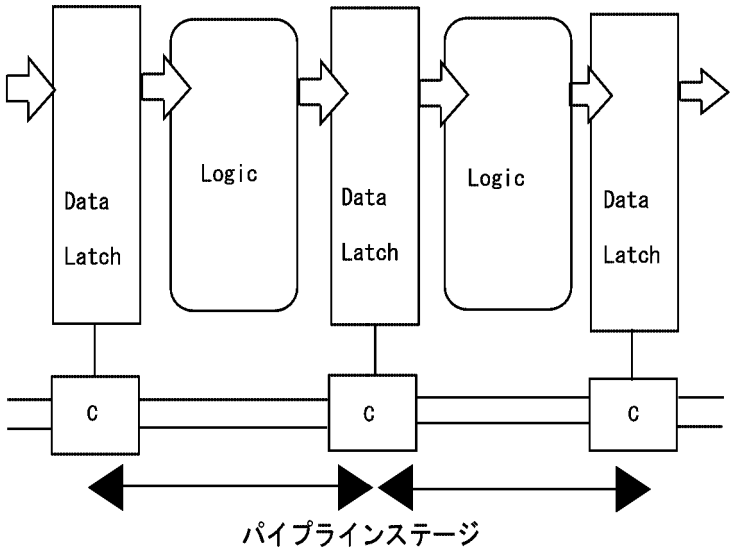

図 3.4　パイプラインステージ

パイプラインのステージ間のデータ転送は自己同期転送制御装置（C エレメント）の連鎖でコントロールされる（図３．４中 C)。一般的に C エレメントは Send-in 信号と Ack-in 信号の両方がアクティブな時に Data Latch にクロック信号を生成する。すなわち、Send-in 信号は先行ステージの処理されているパケットが Data Latch のインプットに入れられる用意ができていることを示し、Ack-in 信号は次ステージは空であることを示している。

3.2.1　パイプラインステージ化のスループットの上昇

パイプラインステージの論理的最善数を検証する前に、スループットの上昇の説明を記す。パイプラインステージ化することでスループットは上昇すると記してきたが、それは何故なのかを説明し、証明をおこなう。尚、分割の際すべてのパイプラインステージ内の通過ゲート段数は同じであり演算時間も等しいとする。

単純に考えると Data Latch が１つ増えているわけだからゲート段数が増加してスループットも低下すると思う。しかしパイプラインステージが２つの時、前項で説明した C エレメントのコントロールによって、２つ目のステージにパケットデータが到着し、出発したとき、１つ目のステージへ新しいパケットデータが送られる。このことによってゲート段数は増加するが、スループットは上昇する。

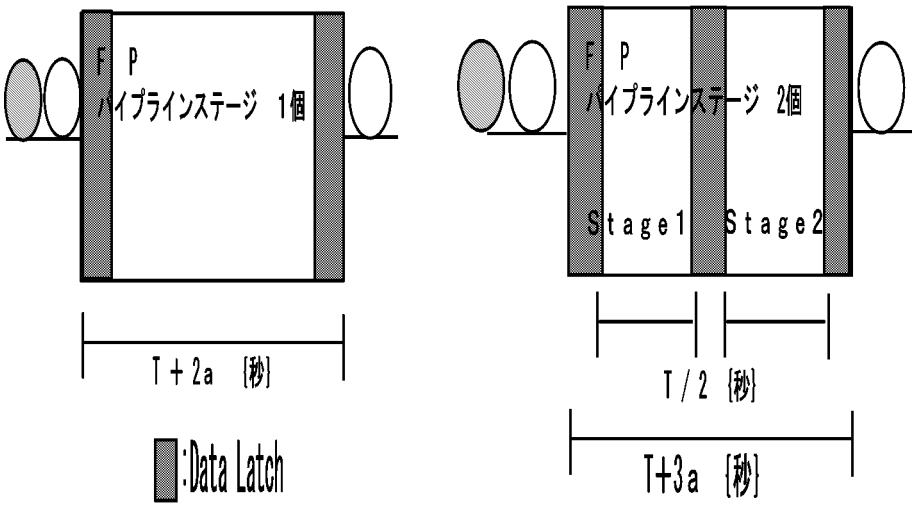

図 3.5　ステージ数

では、次に本当にスループットは上昇しているのか論理的に検証してみる。図３．５で表すように、パイプラインステージが１個のときは１つのデータの演算にかかる時間はＴ(秒)であり、1 つの Data Latch でのかかる時間は a(秒) である。結果、かかる時間はＴ＋ 2a (秒) であり１秒間に処理できる回数は１/ (Ｔ＋ 2a) （回/秒）となる。次に、パイプラインステージが２個のときは Stage1 で（Ｔ/２）（秒）かかるので、結果かかる時間はＴ＋ 3a（秒）であり１秒間に処理できる回数は２/ (Ｔ＋ 3a) （回/秒）となる。上記２つの１秒間に処理できる回数を比べると、１個のときが１/(T+2a) であり２個のときが２/(T+3a) である。1 個のときは

$$\frac{1}{T+2a} = \frac{2}{2(T+2a)} = \frac{2}{2T+4a}$$

となるので、

$$\frac{2}{2T+4a} < \frac{2}{T+3a}$$

となり、結果パイプラインステージ２個のほうが時間効率は良くなる。

一般式や更に詳しい内容は後述とする。

3.2.2　ゲート段数の違いにより生じるスループットの変化

前項では、パイプラインステージ内部の演算ゲート段数は同じとして検証をおこなった。では、パイプラインステージ内部のゲート段数が異なるとスループットはどのように変化するのかを検証する。

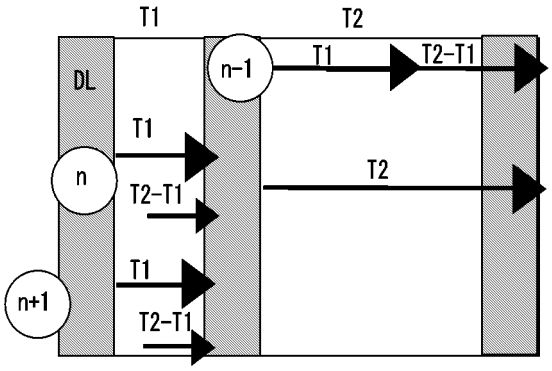

図 3.6　パイプラインステージ２段

図３．６で表すようなステージ数２つのときを例に考える。１つ目のパイプラインステージ内の演算時間を T1、２つ目のパイプラインステージ内の演算時間を T2 としＤＬ（Data Latch）内でかかる時間を a とする。またこのとき Ｔ１ ＜ Ｔ２ とする。このときのスループットを計算すると、ｎ番目のデータが１つ目のパイプラインステージの処理が終わったとき、ｎ－１番目のデータは２つ目のパイプラインステージの処理中であるため、ｎ番目のデータは（Ｔ２－Ｔ１）だけ待ち合わせが発生する。次にｎ番目のデータが２つ目のパイプラインステージの処理の途中のときｎ＋１番目のデータは同様に（Ｔ２－Ｔ１）の待ち合わせが発生する。よってｎ番目のデータを処理するのにかかる時間は、

$$T1 + T2 + (T2 - T1) + 3a$$

となり、スループットは、

$$\frac{2}{T1 + T2 + (T2 - T1) + 3a}$$

となる。では、パイプラインステージ内の演算時間が同じのときと、比べてみる。前項より、パイプラインステージ内の演算時間が同じときのスループットは、

$$\frac{2}{T + 3a}$$

であり、T=T1+T2 であるから

$$\frac{2}{T + (T2 - T1) + 3a} < \frac{2}{T + 3a}$$

となり、パイプラインステージ内の演算時間が同じときよりもスループットは低下している。では、パイプラインステージ数が１つのとき（パイプライン化をおこなって無いとき）と比べるとどうであろうか。パイプラインステージ数が１つのときのスループットは、

$$\frac{1}{T + 2a} = \frac{2}{2T + 4a}$$

であるから、分子は同じであるので分母どうしを比較する。

$$(2T + 4a) - (T1 + T2 + (T2 - T1 + 3a))$$

T=T1+T2 であるから、

$$2T1 + 2T2 + 4a - T1 - T2 - T2 + T1 - a3$$

$$= 2T1 + \alpha > 0$$

となり、パイプライン化しないよりはスループットは上昇している。

以上のことよりパイプラインステージ数２つのときに限っては、パイプライン化しないよりはスループットは上昇するが、パイプラインステージ内の演算時間を等しくしたときよりは、スループットは低下していることがわかった。

では、パイプラインステージが多くなってきたときは、どのような変化がおこるのかを検証する。

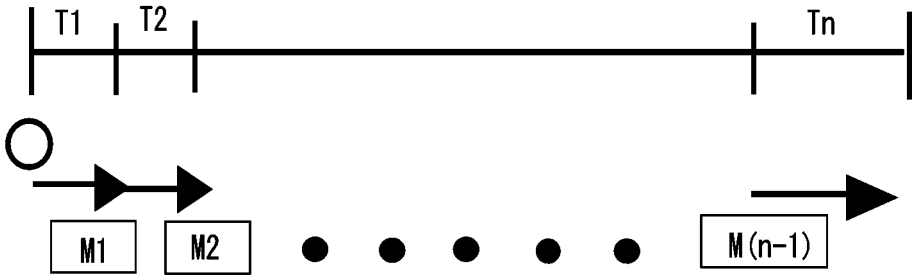

図 3.7　パイプラインステージ数ｎ個

パイプラインステージ数がｎ個のとき、各パイプラインステージ内演算時間を、Ｔ１、Ｔ２、、、Ｔｎとし、パイプラインステージの順序は変えることが可能なのでＴ１ ＜ Ｔ２ ＜ 、、、＜ Ｔｎとする。また各 Data Latch での待ち合わせ時間をＭｎとする。（図３．７参照)。では、このときの１回の処理時間を検証する。まず待ち合わせ１（図中、Ｍ１）を計算すると、

$$(T_2 - T_1) + (T_3 - T_2) + \cdots + (T_n - T_{n-1})$$

$$= T_n - T_1$$

同様にＭ２を計算する。

$$M2 = (T_3 - T_2) + (T_4 - T_3) + \cdots + (T_n - T_{n-1})$$

$$= T_n - T_2$$

更に、Ｍｎ-1 を計算する。

$$M_{n-1} = (T_n - T_{n-1})$$

Mの総和は、

$$(T_n - T_1) + (T_n - T_2) + ... + (T_n - T_n - 1)$$

となるので、１回の処理時間は、

$$T + (n+1)a + (T_n - T_1) + (T_n - T_2) + ... + (T_n - T_n - 1)$$

である。このとき $T = T_1 + T_2 + ... + T_n$ であり、Mは（ｎ－１）個存在するので、

$$nT_n + (n+1)a \tag{3.1}$$

となる。これは、１回の処理時間はパイプライン内で最も時間がかかるステージの演算時間に依存されることをあらわしている。

　では、次に同様にパイプラインステージ内の演算時間が等しいものと比較する。パイプラインステージ内の演算時間が等しいものの１回処理するのにかかる時間は、

$$T + (n-1)a$$

で表せるので、

$$\begin{aligned}&(nT_n + (n+1)a) - (T + (n-1)a)\\ &= nT_n - T > 0\end{aligned} \tag{3.2}$$

となり１回処理するのにかかる時間は演算時間が等しい方が短くなり、スループットも高いといえる。

　次にパイプライン化していないものは、

$$\frac{1}{T + 2a} = \frac{n}{nT + 2na}$$

なので、比較すると、

$$\begin{aligned}&(nT + 2nT) - (nT_n + na + a)\\ &= n(T - T_n) + (n-1)a\end{aligned} \tag{3.3}$$

となり、$T - T_n > 0$ より数式（３．３）は正であるといえる。これはパイプライン化していない方が、１回の処理時間は長くスループットも低いことをあらわす。

以上のことより、パイプラインステージ内の各ステージの演算時間が異なると、演算時間が等しいときよりスループットは低下するものの、パイプライン化しないときよりは、スループットが上昇することがわかった。

3.2.3　パイプラインステージ数とスループットの関係

ここではパイプラインステージ数とスループットの関係を検証する。まず、演算にかかる時間をＴ（秒）、Data Latch でかかる時間をａ（秒）、パイプラインステージ数をｎ（個）（ただしｎは正の整数）とすると、Data Latch 内はフリップフロップなので

$$T > a > 0 \tag{3.4}$$

が成り立つ。また１回処理するのにかかる時間は以下のように成る。

$$T + (n+1)a \qquad (秒)$$

更に１秒間に処理できる回数は以下のように成る。

$$\frac{n}{T + (n+1)a} \qquad (回/秒) \tag{3.5}$$

ここで、パイプラインステージの数を限りなく増加させると以下のように成る。

$$\lim_{n\to\infty} \frac{n}{T + (n+1)a}$$

$$= \lim_{n\to\infty} \frac{\frac{n}{n}}{\frac{T}{n} + a + \frac{a}{n}}$$

$$= \frac{1}{a}$$

となり、1/a に収束する。これはパイプラインステージ数を限りなく増加させても１秒間で処理できる回数は１/a（回/秒）以上にはならないことを示す。では、以上のことより検証をおこなう。まず数式３．５の１秒間に処理できる回数をｙとおく。

$$y = \frac{n}{T + (n+1)a}$$

$$y = \frac{n}{an + T + a} \tag{3.6}$$

$n \div (an + T + a) = \frac{1}{a}... - \frac{T}{a} - 1$ となるので

$$\begin{aligned} y &= \frac{-\frac{T}{a} - 1}{an + T + a} + 1 \\ y &= -\frac{T + a}{a^2 n + a^2 + aT} + \frac{1}{a} \\ y &= -\frac{T + a}{a^2 n + -(a - T)a} + \frac{1}{a} \end{aligned} \tag{3.7}$$

となる。更に n ＝1 のとき

$$\begin{aligned} y &= -\frac{T + a}{a^2 - a^2 - aT} + \frac{1}{a} \\ &= \frac{T + a}{aT} + \frac{1}{a} \quad > 0 \end{aligned}$$

となる。また一般的な１秒間に処理できる回数は以下のようにも書ける。

$$\frac{T + a}{(n - 1)a + aT} + \frac{1}{a}$$

よってこのグラフは直角双曲線 $y = -\frac{T+a}{a^2 n}$ を x 軸方向に $1 - \frac{T}{a}$, y 軸方向に $\frac{1}{a}$ だけ平行移動したものであり、漸近線は２直線 $x = 1 - \frac{T}{a}$、$y = \frac{1}{a}$ である。図３．８参照

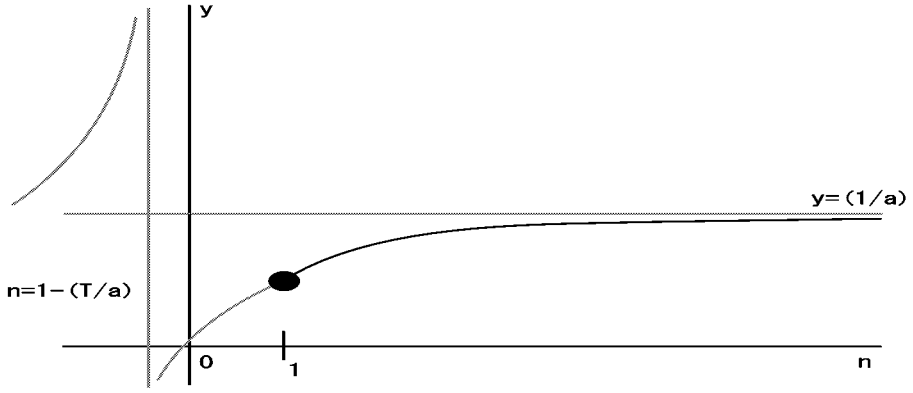

図 3.8　$y = -\frac{T+a}{a^2 n + -(a-T)a} + \frac{1}{a}$ のグラフ

3.2.4　パイプラインステージ数とハードウェア領域の関係

では、次にパイプラインステージ数とハードウェア領域の関係を検証する。

まず演算に必要なハードウェア量（ゲート数）をH（個）、パイプラインステージ数をｎ（個）Data Latch のハードウェア量（ゲート数）をｂとすると、

$$H > b > 0 \tag{3.8}$$

が成り立つ。また、１回の処理の総ハードウェア量は以下のようになる。

$$H + (n+1)b \qquad (個)$$

これは、総ハードウェア量はパイプラインステージ数が増加すればするほど、比例して増加することを示す。ここで、ハードウェア量は少ないほうが好ましいので逆数をとる。ハードウェア量を、ｙ２とおくと

$$y2 = \frac{1}{bn + (b+H)} \tag{3.9}$$

となる。このとき $n = \frac{H+1}{b}$ のとき分母は０となるので、この式のグラフは、直角双曲線 $y2 = \frac{1}{bn}$ をｘ軸方向に $\frac{H+1}{b}$ だけ平行移動したものであり、漸近線は２直線 $n = \frac{H+1}{b}$、$y2 = 0$ である。図３．９参照。

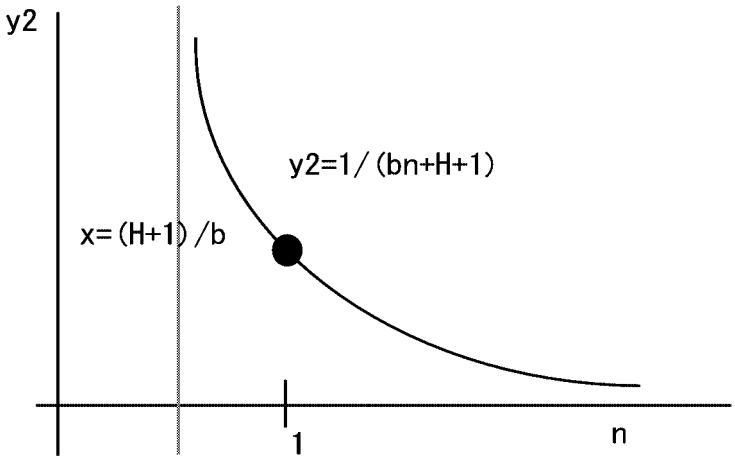

図 3.9　$y2 = -\frac{1}{bn+H+1}$ のグラフ

3.2.5　パイプラインステージ数の論理的最善数

前述したとおり、スループット性能はパイプラインステージ数の増加に伴い上昇する。しかし、そこにはCエレメントの性能という上限が存在する。

自己タイミング型演算機構はCエレメントでコントロールされていることは前述してあるが、そのコントロールしているもの自体のスループット性能を越える性能が、演算機構にあったとしても、Cエレメントの性能以上の性能をだすことはできない。

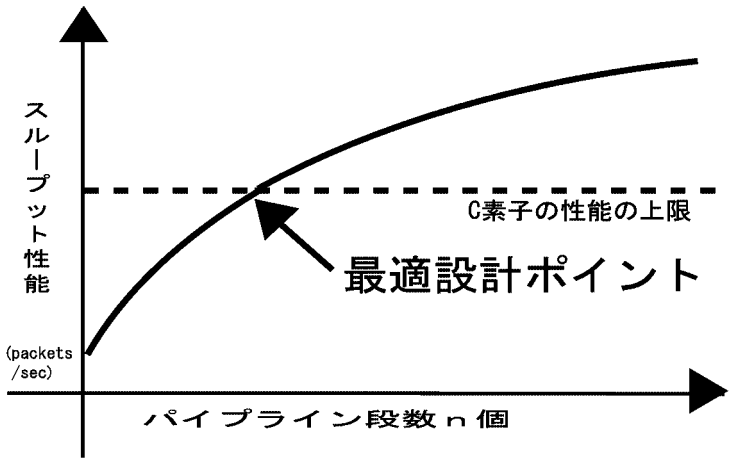

図 3.10　スループット性能の上限

要するに、Cエレメントのスループット性能（図中点線）と数式３．７の曲線（図中実線）のスループット性能が低い方が演算機構全体のスループットの性能となる（図３．１０参照)。よって２つの線の交点が最適設計ポイントであり、それ以下の最大のｎがパイプラインステージ数の論理的最善数といえる。

第 4 章

FP(Functional Processing Unit) 部の設計と構造

4.1 設計指針

ここでは、前章での最適設計指針にしたがって、データ駆動プロセッサの主要部分の１つでありゲート段数、ゲート数が最も多く存在し、演算部のアルゴリズムに改善の余地があるであろうＦＰ部の設計をおこなう。このときの主な設計指針はハードウェア領域の減少のための共有化を行なうことと、スループット上昇のためのパイプライン化において最善パイプラインステージ数を導出し、適用をおこなうことである。

4.2 FP（Functional Processing Unit）部の構造

まず始めに、FP 部の構造を記す。（図４．１参照 ）。尚、図４．１は構造を便宜的に記したものであるため、実際の構造とはパイプラインステージ数が異なる。

入力データは、データを記憶するモジュールである Data Latch を通る。そして、Op.code(オペレーションコード) は DECODE 部へ、その他のデータ (Dest.node,Generation id., DataA,DataB) は Operation 部へとそれぞれ流れる。その後、DECODE 部、Operation 部での 結果の値を SELECT 部へ送り Data Latch を通り FP 部の最終的な出力となる。

次に FP 部における入出力パケットについて説明する（図４．２参照)。入力パケットは

4.2 FP（Functional Processing Unit）部の構造

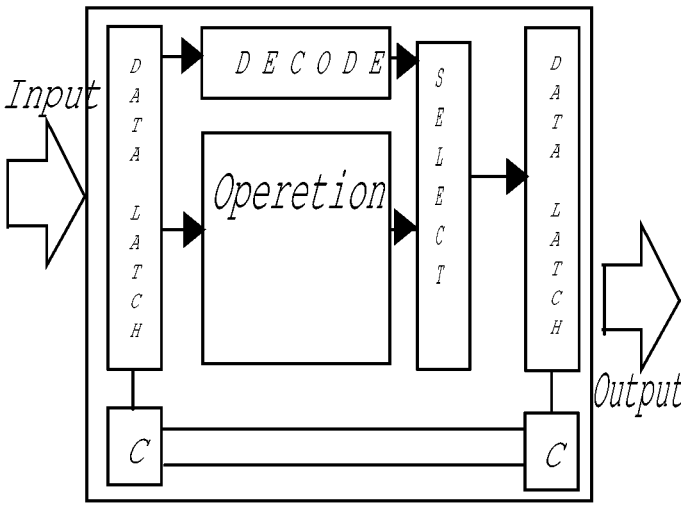

図 4.1 FP 部の構造図

Op.code6bit、Dest.node(6bit+1bit)×２（+1bit は左入力、右入力のフラグ。×２は Truth と Faulse)、Generation id.24bit、Data A Data B(データＡ、Ｂ。最上位 bit を符号 bit、数字は２の補数表現とする。）12bit ×２ の全６８ bit となる。出力パケットは Op.code 6bit、Dest.mode 6bit+1bit、Generation id. 24bit、 Result 12bit の４９ bit となる。

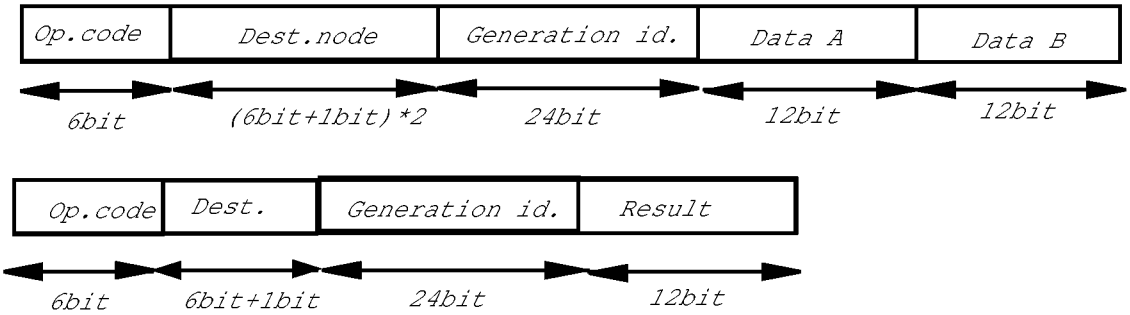

図 4.2 入力パケット (上)、出力パケット (下)

4.2.1 C(Self-timing transfer control circuit)

Data Lutch 同士を結ぶ自己同期転送制御装置。C は、前の Data Latch よりパケットが送られ現 Data Latch の Input に入れられる用意が出来ていること、次の Data Latch は空であることの２つを信号によってやりとりし、現 Data Latch にパケットを送れという信号を出す。これによって、時間短縮を計る。

4.2.2 Operation 部

Operation 部では、Dest_node,Generation id.,Data A,Data B, を演算する。ここで注意することは、Op.code での指定に関わらず全ての演算を行なうということである。よってここでの論理展開は重要であり、ゲート数に直接関係をもたらす部分である。

命令は、論理演算、シフト (Logical shift)、算術演算、比較、条件比較、世代操作、その他の７項目から成り、全２６種類作成した。入力パケット、出力パケットを 見てもらえば解るが、命令は全て入力は２入力で、出力は１出力である。１入力の命 令については、各命令で詳細を記す。Op.code と命令の対応の全てを表にした（表４．１、４．２参照)。また、上位３ bit と下位３ bit で分けて判り易くしてある。

Op.code		命令
001	000	EXOR
	001	OR
	010	NAND
	011	NOR
	100	MUL
	101	SYNC
	110	NOP
010	000	ADD
	001	ABS
	100	SUB
	101	DIF
011	000	AGN
	001	SGN

表 4.1 Op.code 表 1

Op.code		命令
100	000	AND
	001	BIT
	010	SWBIT
101	000	ENOR
	001	EQ
	010	SWEQ
110	000	GT
	001	SWGT
	100	GE
	101	MAX
	110	MIN
	111	SWGE
111	000	LSH

表 4.2 Op.code 表 2

4.2 FP（Functional Processing Unit）部の構造

各命令について記す。尚、アルゴリズムについては付録を参照。

論理演算

命令　AND,OR,EXOR,NAND,NOR,ENOR の６種類から成る。

入出力数　２入力。１出力。

変化データ　Data A, Data B.

最大通過ゲート段数　１段

シフト（Logical Shift）

命令　LSH(A のデータを B のデータ分シフトする。) の１種類から成る。

入出力数　２入力。１出力。

変化データ　Data A, Data B.

最大通過ゲート段数　２段

算術演算

命令　ADD(加算),SUB(減算),DIF(A-B の絶対値),ABS(A+B の絶対値),MAX(A,B の数の大きい方を出力),MIN(A,B の数の小さい方を出力),MUL(掛け算) の７種類から成る。

入出力数　２入力。１出力。

変化データ　Data A, Data B.

最大通過ゲート段数　ABS １４段

比較

命令　EQ(A=B のとき、出力１。A≠B のとき出力０。), GE(A ≧ B のとき出力１。A ＜ B のとき出力０。), GT(A ＞ B のとき出力１。A ≦ B のとき出力０), BIT(A,B の

AND で一致してなければ出力１、一致していれば出力０) の４種類から成る 。

入出力数　２入力。１出力。

変化データ　Data A, Data B.

最大通過ゲート段数　GE、GT １７段

条件分岐

命令　SWEQ(A=B のとき True より Data A を出力。A≠B のとき False より Data A を出力。), SWGE(A ≧ B のとき True より Data A を出力。A ＜ B のとき False より Data A を出力), SWGT((A ＞ B のとき True より出力１。A ≦ B のとき False より出力０), SWBIT(A,B の AND で一致してなければ True より出力１、一致していれば False より出力 ０) の４種類から成る。

入出力数　２入力。１出力。

変化データ　Data A, Data B, Dest_node(行き先ノード)=True or False。

最大通過ゲート段数　SWGE、SWGT １７段

世代操作

命令　SGN(Generation id.(世代番号) の下位１２ bit に B のデータを格納する。), AGN(Generation id.(世代番号) の下位１２ bit を A のデータに書く。) の２種類から成 る。

入出力数　２入力。 １出力。

変化データ　Data A, DataB, Generation id(世代番号)。

最大通過ゲート段数　０段

その他

命令　SYNC(出力は右入力に依らず、左入力), NOP(入力データをそのまま出力。) の２種類から成る。

入出力数-SYNC-　２入力。１出力。

入出力数-NOP-　１入力。１出力。

変化データ-SYNC-　Data B。

変化データ-NOP-　変化無し。

最大通過ゲート段数　０段

4.2.3　DECODE 部

基本的命令２６種のオペレーションコードを判別し指定された命令を割り出す部分。入力は Op.code(オペレーションコード)6bit。出力は指定されたものを出力“１”とし、その他を出力“０”とする。Op.code(オペレーションコード) は、上位 3bit、下位 3bit ずつ判別し、そのまま出力する形をとった。これは、再利用への対応とともにＳＥＬＥＣＴ部での使い易さを考えたものである。（図４．３参照）

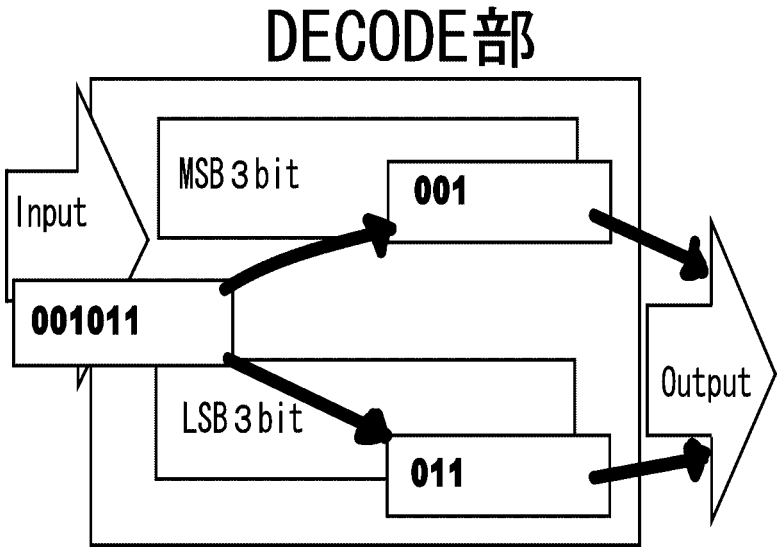

図 4.3　DECODE 部の構造

4.2.4 SELECT 部

DECODE 部よりの出力と Operation 部よりの演算結果を照合し、指定された Op.code の演算結果を出力する。アルゴリズムとしては、２つの出力のＯＲ（論理和）をとる。

4.3 設計

前述の設計指針にしたがってＦＰ部の設計をおこなう。前述の設計指針をまとめると以下のようになる。

設計指針１　ハードウェア領域の減少のため共有化、アルゴリズムの工夫をおこなう。

設計指針２　スループット上昇のため、最適パイプライン化をおこなう。

4.3.1 設計指針１－共有化

最適設計指針に従って実際に共有化をおこなった基本命令を記す（図４．４参照)。ADD（SUB）出力を使用して、ABS（DIF）を求める。AND 出力を使用して BIT を求め、BIT 出力を使用して SWBIT を求める。ENOR 出力を使用して EQ 出力を求め、EQ 出力を使用して SWEQ を求める。GT 出力を使用して SWGT を求める。GE 出力を使用して、MAX、MIN、SWGE 出力を求める。

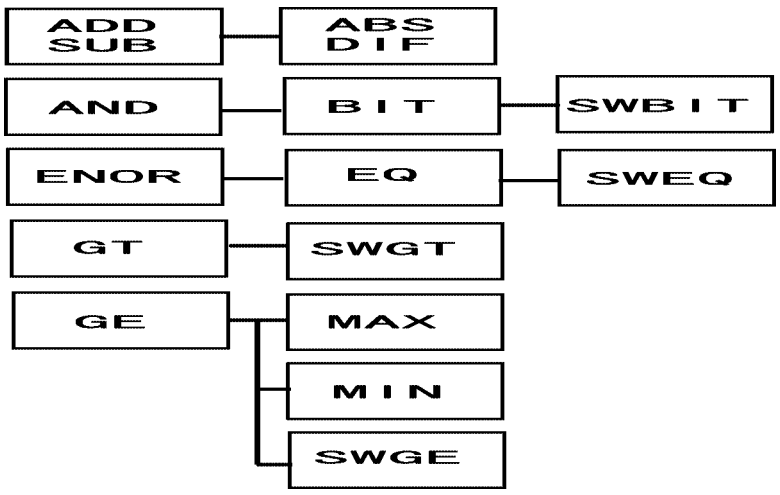

図 4.4　共有化をおこなった命令

基本命令ＧＥのアルゴリズムの決定

では、ここで共有化を行なわなかったものについて触れる。前述のとおり、共有化をおこなうことによってハードウェア領域は減少する。しかしアルゴリズムによっては、ゲート段数が増加しスループットが低下することがある。そのとき、どちらを優先すべきかが問題となる。ここでは、本研究で設計、構築をおこなうＦＰ内部の基本命令ＧＥ（A≧Bのとき出力１。A＜Bのとき出力０）を例にどちらを優先すれば良いのかを検証する。

ＳＵＢ出力を使用する共有化とＧＥ独自のアルゴリズム（アルゴリズムの詳細は付録Ａを参照。）を使用する場合について検討する。まず表３．１、３．２でのＴＯＴＡＬ部分のゲート段数の説明をしておく。ＴＯＴＡＬ部分のゲート段数は単純な加算ではなく、桁上げ（もしくは、桁下げ）信号を出すのに必要なゲート段数を加算していき最終的な bit での出力のゲート段数を加えて総ゲート段数としている。では検討をおこなう。ＳＵＢ出力を使用した場合共有化により、ＳＵＢの結果を出力するのにハード量（ゲート数）は使用しないが、ゲート段数が１９段かかるためスループットは独自アルゴリズムを使用する場合と比べると低下する（表参照。）。その上、ＳＵＢ出力のままではＧＥの結果にはならないので更に１，２段は要する。では、独自アルゴリズムを使うとどうであろうか。同様に表をみれば判る通りゲート数は６０個増加するもののゲート段数は１４段で済ますことができる。後々のパイプライン化を考えると他の基本命令と比較してゲート段数が多い基本命令のゲート段数は少しでも減らしておきたい（詳細は次の項目で触れる。）ので、ここでは独自アルゴリズムによって基本命令ＧＥを構築するものとする。

尚、本研究においては、このような結論となったが他の基本命令のゲート段数や、スループットとハードウェア量の優先順位によってはこの限りではない。

	ゲート数	段数
0～3 bit	20	6
4～7 bit	20	6[1]
8～11 bit	20	5[1]
TOTAL	60	14

[1] 桁下げ信号は 4 段

表 4.3 独自アルゴリズム

	ゲート数	段数
0～3 bit	86	9[1]
4～7 bit	86	9[1]
8～11 bit	88	11
TOTAL	無	19

[1] 桁上げ信号は 4 段

表 4.4 SUB 出力

4.3.2 設計指針２－パイプライン化

前述の最適設計指針に従って、パイプライン化をおこなう。

まず C エレメントのスループット性能は

$$\frac{1}{10} \qquad (\text{回}/ns)$$

となる。また、前述の数式３．５における T＝３１、a＝２となり、

$$\frac{n}{31+2n+2} = \frac{n}{33+2n} \qquad (\text{回}/ns)$$

となる。以上のことより最適設計ポイントは

$$\begin{aligned} \frac{n}{33+2n} &= \frac{1}{10} \\ 33+2n &= 10n \\ n &= \frac{33}{8} \\ &= 4.1... \end{aligned}$$

となり、n の最大値は４となる。この結果に従ってパイプライン化を行なう。その際各パイプラインステージ内の演算回路ゲート段数を、できる限り等しくする必要がある。前述したが、パイプラインステージ内のゲート段数の最も多い部分にパイプラインステージ全体の処理時間は依存される。よって、ゲート段数が最も多いものを基本に考える。すると最も

ゲート段数が多いものは ADD（SUB）、ABS（DIF）の７段、８段である。しかも、ABS（DIF）は ADD（SUB）の出力結果が必要であるため、後述するが４ BIT ずつでは４段に収まらない。また ADD（SUB）、ABS(DIF) は桁上げ先見加算機を基本に設計しているため、次 bit の出力結果を得るには桁上げ信号が必要である。更に４段のパイプライン化において、最後の４つめの段には SELECT 部が必要であり全演算結果を要する。以上のことにより、ADD(SUB)、ABS(DIF) を６ bit 演算にする必要がある。その結果、各パイプラインステージ内の段数は、７段、８段、８段、８段となる。（図４．５参照）

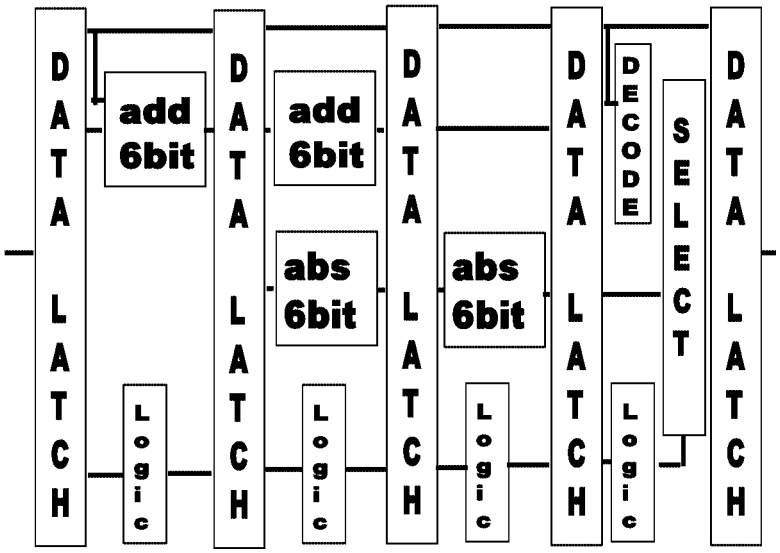

図 4.5 最適パイプライン

4.3.3 再利用への対応

ここでは、再利用への対応として拡張されやすいよう構築した部分の説明をおこなう。

4bit 演算

Data が１２ Bit であるため１２ Bit 用のものを作成していたが、再利用、拡張を考慮し４ Bit 用のものを作成しそれを３つ並べて１２ Bit 用とした。しかし、命令 LSH については全場合のシフトを想定し場合分けのアルゴリズムを使っているためこの限りではない。また前述のとおり ADD(SUB)、ABS(DIF) も６ bit 演算とした。

OP.code

共有化との関係で Op.code を決定してあるので、再利用の際、命令を増やすときに共有化する命令の部分に追加しやすくしてある。

第 5 章

結論

本研究では、２章ではデータ駆動プロセッサの構造について記述した。また３章ではハードウェア、スループットを中心に論理的に最善策を検証した。また４章ではＦＰについての構造について述べ、３章での論理的最善策に従って設計をおこい目標性能を達成できる見通しを得た。

今後の課題として挙げられるのは、論理的最善策の決定におけるパラメータを増加させ一般化にすることである。本研究の結果では、例外の無い非同期回路の最善パイプラインステージ数しか検索は出来ない。また並列処理のパイプライン化は並べ方によっては、全体の演算時間が大きく変動する。最善のパイプラインステージ数は一般化の式より導きだすことができるが、どの部分を並列処理にして各パイプラインステージ内のゲート段数は幾つが最善なのかは導くことはできない。よって、更なる研究で、並列処理のパイプライン化における各パイプラインステージ内の最善ゲート段数の検証を必要とする。

将来展望として、今後の課題で挙げた事柄が解決できれば、最善のパイプラインステージ数が導きだすことができ、パイプラインステージ内の最善ゲート段数をも導くことができる。そうなれば、これから更に巨大になるであろうＬＳＩの設計、構築においてパイプライン化が円滑、かつ正確におこなうことができるようになる。これによって、大幅な時間短縮と有益なハードウェア領域を取得することができる。

謝辞

本研究において、懇切なる御指導、御鞭撻を賜わった高知工科大学の岩田 誠助教授に心より感謝の意を表します。

本研究を進めるにあたり、適切な御助言、御指導を賜った大森 洋一助手に深く感謝の意を表します。

日頃から御支援、適切な御指導を頂いた細美 俊彦氏に感謝の意を表します。

日頃からご支援頂いた情報システム工学科 岩田研究室の方々、別役 宣奉氏、前田 庸亮氏、森川 大智氏、橋本 正和氏、仮屋 文彦氏に感謝の意を表します。

参考文献

[1] http://darwin.kz.tsukuba.ac.jp/bit.FPGA.html

[2] http://www.sharp.co.jp/sc/gaiyou/news/970331.htm

[3] H.Terada,S.Miyata,M.Iwata,"DDMP's: self-timed super-pipelined data-driven multimedia processors", Proc.of IEEE,87(2),pp.282-296(1999).

[4] 工学博士 田丸啓吉,"論理回路の基礎", 工学図書株式会社,1988.

[5] 井川 満、井関兼四郎、伊東清三、大島利雄、加藤順二、塩田徹治、渡辺信三, "数学Ⅲ", 数研出版株式会社,1966.

[6] 西本敏彦,"微分積分学講義", 培風館,1966.

付録 A

FP 部の各命令のアルゴリズム

A.1　論理演算

各ビットの論理演算をとる。

A.2　シフト

LSH は、B のデータ分 Logical Shift するが、B データの最上位 bit が 0（正数）なら左シフト、1（負数）なら右シフトを行なう。データのビット数より１１ bit のシフトまでなので全ての場合の回路を作成し、DataB よりシフト数を判別し、信号として出力し場合分け回路との論理積（AND）をとり出力としてある。

A.3　算術演算

ADD 参考文献 [4] より 4 bit 桁上げ先見加算回路をベースにした。ここでは、4 bit を 1 ブロックとし 3 ブロック使用し 12bit に対応させてある。更にブロックごとに上位ブロックへの桁上げ信号を発生させているため入力データの bit 数が増加しても拡張しやすく一般的であり、どのレベルのブロックにも対応できる。Over Flow 対策としては、最上位 bit に追加 bit(１３ bit 目) を複製し１３ bit 目を符号 bit として扱った 表A．１ 参照。また、SUB も Op.code の信号より入力データを変化させ、ADD と同じ回路を使用した。これによって、ハードウェア量の減少をおこなった。

MAX,MIN 後に記す GE(A $\geqq$ B のとき出力１。A $<$ B のとき出力０。) を使用する。

13bit	12bit	符号
0	0	正
0	1	正[1]
1	0	負[1]
1	1	負

[1] Over Flow

表 A.1 Over Flow

MAX を例に説明すると、GE よりの出力と Data A の AND をとり、GE よりの出力の否定と Data B の AND をとる。２つの出力の OR をとり、最終出力とする。MIN は Data A と Data B を逆転させる

DIF,ABS ADD,SUB の出力を使用し作成。ADD よりの出力は Op.code により ADD 出力と SUB 出力とに変化する。よってその出力を絶対値表現に変換させる。尚、出力データの最上位 bit は常に０とする。ABS を例に説明すると、ADD を通過した結果を正数の時はそのままで、負数の時は演算回路に通し、２つのデータの各 bit(最上位 bit は除く) の OR をとり最終結果とする。図 3.3 参照。正数のときは絶対値はそのままで良いが、負数のときは２の補数表現に変化させる必要がある。

A.4 比較

EQ 各 bit に ENOR を通し、その結果を AND で判定する。

BIT 各 bit に AND を通し、結果を否定し出力。

GE,GT 方法は２つ考えられる。１つは SUB を使用する方法である。もう１つは最上位 bit より段階的に判定していく方法である。最大通過ゲート数を考え後者で作成する。GE を例に説明する。第１に最上位 bit から調べる。最上位 bit は符号 bit として扱っていることに注意しながら、最上位 bit での $A > B$、$A < B$、$A = B$ の３つの場合を調べる。($A > B$

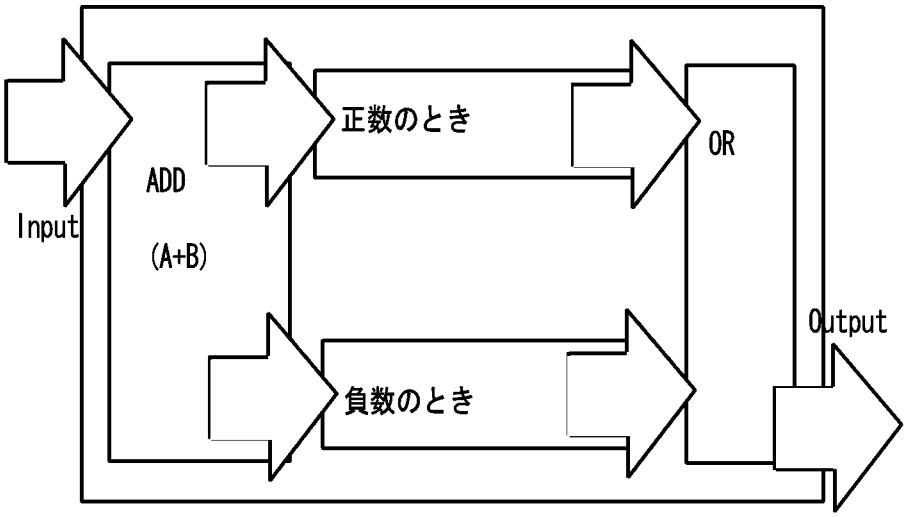

図 A.1　ABS の構造図

の出力は $A > B$ の時１でその他は出力０。$A < B$ の出力は $A < B$ のとき出力０で、その他は出力１。A=B の出力は A=B の時出力１でその他は０。）その後 $A > B$、$A < B$ の結果の OR をとる。$A = B$ のときは、１つ下の bit(この場合は１１ bit 目) に信号を出す（これをイコール信号と仮に呼ぶ)。次に１１ bit を調べる。ここでは１１ bit 目だけの $A < B$、A=B を調べ $A < B$ の結果と上位 bit(この場合は１２ bit 目) からのイコール信号との NAND をとり、$A = B$ の結果と上位 bit からのイコール信号との AND をとる。そして更に下位 bit でも同様に繰り返し最下位 bit まで続ける。最下位 bit では、$A < B$ だけを調べ上位 bit からのイコール信号との NAND をとる。そして最終的にそれぞれの bit の出力の AND をとる。最終的に AND（論理積）をとるので各 bit での出力（$A < B$ の時のみ出力０がでるように作成）が１のときは他での分別が可能であるが、０ならばその出力に依存される。図３．４参照。

A.5　条件分岐

SWEQ EQ を使用する。EQ よりの出力が１のとき Dest.node の True を Dest.node とし、EQ よりの出力が０のとき Dest.node の False を Dest.node とする。

SWGE GE を使用する。GE よりの出力が１のとき Dest.node の True を Dest.node と

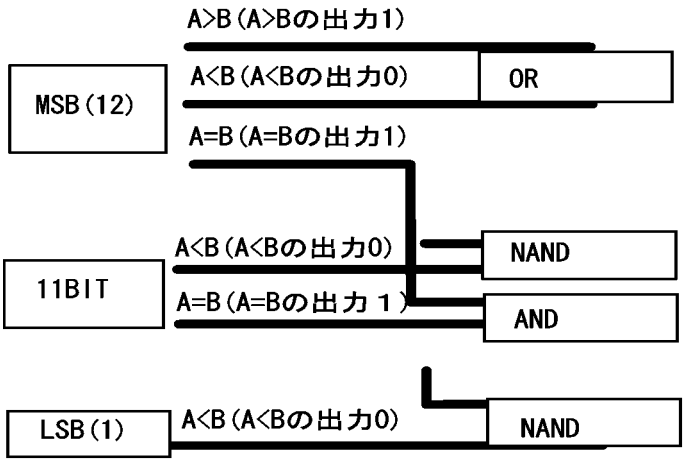

図 A.2　GE の構造図

し、GE よりの出力が０のとき Dest.node の False を Dest.node とする。

SWGT GT を使用する。GT よりの出力が１のとき Dest.node の True を Dest.node とし、GT よりの出力が０のとき Dest.node の False を Dest.node とする。

SWBIT BIT を使用する。BIT よりの出力が１のとき Dest.node の True を Dest.node とし、BIT よりの出力が０のとき Dest.node の False を Dest.node とする。

A.6　世代操作

AGN Generation id. の下位１２ bit を A のデータに書く。

SGN Generation id. の下位１２ bit に B のデータを格納する。

A.7　その他

SYNC 左入力をそのまま出力。

NOP A データをそのまま出力。