修 士 論 文

ウェーブパイプラインを用いたマルチスレッド型プロセッサアーキテクチャに関する研究

指導教官　日比野 靖 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

池田 吉朗

1999 年 2 月 15 日

要 旨

現在の一般的な計算機の多くが高速な処理のために何らかの並列性を利用している。利用する並列性の違いによってスーパースカラ、VLIW、マルチプロセッサなどの方式があるが、これらは通常どれもパイプライン処理を伴っている。このパイプライン処理の効率を向上させ、高いスループットを実現するアーキテクチャとしてマルチスレッド型プロセッサアーキテクチャがある。マルチスレッド型プロセッサはパイプラインをストールさせることがなく、リソースの使用効率を最大限に保つ。

このマルチスレッド型プロセッサを高速に動作させる手段として本研究ではウェーブパイプラインの導入を考える。マルチスレッド型プロセッサの持つ特徴がウェーブパイプラインの導入を有利にすることを明らかにし、より高い性能を達成するための設計手法を提案する。

提案する設計手法には、マルチスレッド型プロセッサをウェーブ動作させるためのステージ構成、ステージ内部の回路設計に加え、素子/配線モデル、レイアウト法、さらにはコストを低く抑えるための工夫などが含まれる。

これらの手法は伊藤 [1] らによって設計されたマルチスレッド型プロセッサである MUP に適用され、効率的なウェーブ動作、低コストを実現するウェーブパイプライン版 MUP を設計し、シミュレーションにより評価する。

目 次

序論

パイプライン処理の効率を向上させ、高いスループットを実現するアーキテクチャとしてマルチスレッド型プロセッサアーキテクチャがある。そのマルチスレッド型プロセッサの高速化手法として本研究ではウェーブパイプラインの採用を考える。

ウェーブパイプラインを用いたプロセッサの設計手法は確立されておらず、専用の CAD もない。各設計段階で遭遇する様々な選択肢において、明確な根拠のないままに先へ進むような妥協が各所にあることを予め断っておく。

本論文は 8 章で構成される。

まず第 1 章でマルチスレッド型プロセッサについて説明する。伊藤 [1] らによって設計された MUP の概要についても記述する。第 2 章ではウェーブパイプラインについて説明し、それをマルチスレッド型プロセッサに導入する利点を明らかにする。第 3 章ではウェーブパイプラインを念頭に置いた設計手法を提案し、その準備を行う。第 4 章ではウェーブ動作するパイプラインプロセッサの最適なステージ分割を検討する。第 5 章ではステージ内部の回路をどのように設計すべきかを考察する。第 6 章ではウェーブパイプラインを導入する上でのコストとパフォーマンスの関係を議論する。第 7 章では、第 3 章から第 5 章までの設計手法を適用し、第 6 章の結果を反映したウェーブパイプライン版 MUP の設計を具体化し、動作可能な周波数などをシミュレーションにより評価する。得られた結果に対する考察などを最後の第 8 章でまとめる。

第 1 章

マルチスレッド型プロセッサ

1.1 通常のパイプラインプロセッサ

現在の一般的な計算機の多くがパイプライン処理を行っている。しかし、単一の命令ストリームをパイプライン処理する場合、パイプラインハザードの問題が避けられない。構造ハザード (資源競合) を避けるために複数の機能ユニットを用意したり演算パイプラインを採用し、データハザードや制御ハザード (分岐や割り込み) を避けるために静的または動的スケジューリングを行ってもハザードを完全に回避することはできない。

動的スケジューリングや命令の並列実行のための複雑な制御を持つ高度なスーパースカラプロセッサが多く現れたが、競争力として Time to Market が重視され LSI 設計の短期間化が要求されるという流れの中、複雑な制御を伴うアーキテクチャは設計の複雑化により動作周波数の向上や機能追加などへの対処といった面で設計上または製造上の問題が大きい。CMOS テクノロジの微細化に伴う配線遅延やクロックスキューの相対的増加などを考えると、将来的なプロセス技術の上で非常に複雑なプロセッサを設計するのはより困難になる。

そこで考えられるのが、ハザードのないパイプラインを持ち、複雑な制御なしで複数のスレッドを並列に実行するマルチスレッド型プロセッサである。

1.2 マルチスレッド型プロセッサ

マルチスレッド型プロセッサは、パイプライン段数と同数の独立な命令流 (スレッド) を並列に実行し、図 1.1のように全てのステージを独立なスレッドの命令 (つまりデータ依存、制御依存関係のない命令) で埋め、さらにスレッド数と同数のレジスタセットを用意することで、通常のパイプライン処理で問題となるデータ依存、制御依存、資源競合をなくし、パイプラインハザードを完全に回避する。一つのスレッドだけに着目すればパイ

プライン化されない逐次処理であるため通常のパイプライン処理よりもレイテンシは大きいが、複数のスレッドを並列かつストールなしに走らせることができるのでシステム全体のスループットという意味での処理能力は向上する。ソフトウェアから見れば SMP と同じに見え、マルチプロセッサを仮想的に実現しているという見方ができる。

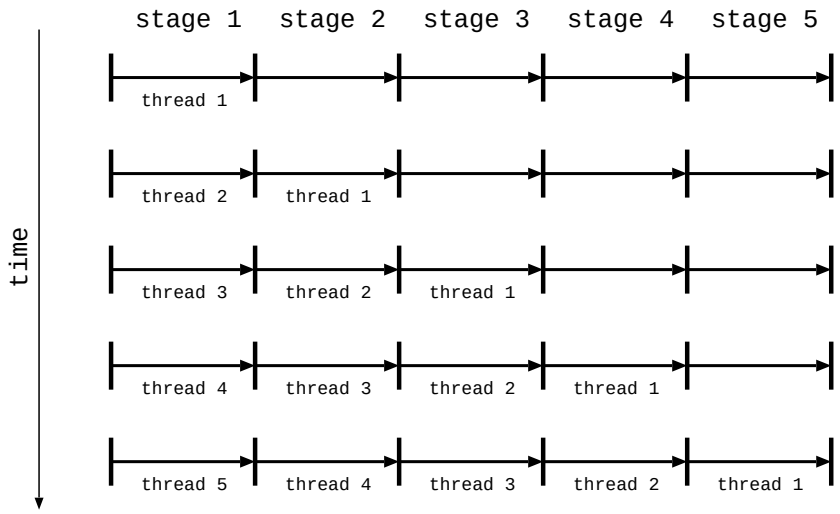

図 1.1: マルチスレッド型プロセッサのパイプライン

マルチスレッド型プロセッサの考え方自体は古くからあるが、あまり普及することはなかった。しかし近年になって、IBM の 2 スレッド版 PowerPC[12] や Tera MTA[10] などの商用ベースのプロセッサが発表され、将来の有望なアーキテクチャの一つとして注目されるようになった。

以下にその特徴を示す。

- ハザードが発生せずパイプラインはストールしない。よって各パイプラインステージのリソース使用効率は非常に高くなる。
- 独立な複数個のスレッドを並列に実行する。実装するスレッド数によるが、マルチプログラミング環境、マルチメディアやネットワーク関係の処理の性質などを考えると並列に実行可能なスレッドは十分に存在すると考えられる。
- スレッドの切り替えをハードウェア的に行うので、割込みによるソフトウェア的なコンテクストスイッチよりもオーバーヘッドが格段に小さい。
- 毎サイクルのメモリ参照を可能にするには、十分な容量を持つパイプラインキャッシュが必要になる。
- メモリレイテンシが許容され (パイプライン化により隠蔽できる)、プロセッサを busy に保つことができる。

- 同時に処理可能なスレッドが十分に存在しないと高いスループットが得られない。最悪の場合、パイプライン化されない単一プロセッサと同程度になってしまう。
- 一つのスレッドだけに着目するとパイプライン化されない逐次処理であるためフォワーディングの必要がなく、パイプラインにフィードバックがない。つまりデータはパイプラインを一方向にしか流れない。
- 一部を除いてレジスタはスレッド数個分用意されるのでスレッド間でアクセス競合が生じない。

複数のスレッドが同時に走るので十分なバンド幅を提供するメモリ構成が重要である。また、並列に実行可能なスレッドの抽出、プロセッサを遊ばせないようなスレッドの入れ換えなど、コンパイラや OS のサポートが性能を大きく左右する。

レイテンシが大きいためターンアラウンドタイムが要求される場合 (データベースなどロックを使うものなど) には適さないが、画像や映像処理などには性能を発揮できる。

1.3 MUP

伊藤 [1] らは MUP(Multi-threaded Ultra-pipeline Processor) と呼ぶマルチスレッド型プロセッサを設計した。MUP はマルチスレッド型プロセッサの特徴を満たすための最低限のハードウェアしか持たない。ハードウェアを単純化することで、先端的なプロセス技術を意欲的に採用し、その上で高周波数動作、高集積度を実現することができると考えている。本研究で考えるウェーブパイプラインを導入するにあたっても、ハードウェア構成が単純な方が都合が良い。

MUP は前述のマルチスレッド型プロセッサの利点に加え、パイプラインを深く (17 段) 短ピッチにすることで高周波数動作を狙っている。命令セット、メモリマップは MIPS R2000 に準拠し、キャッシュは分離型とする。ただし浮動小数点演算、乗除算、シフト演算はサポートしない。スレッドはステージ数に等しい 17 本固定とし、単純に round-robin に切り替える。また、スレッド数に等しい 17 組のレジスタセット (汎用レジスタファイル + 特殊レジスタ郡) を持つ。

MUP は現在、FPGA を用いた実装による検証を行っている [3]。

1.3.1 MUP のキャッシュ

MUP はパイプライン化されたキャッシュを持つことにより毎サイクルのメモリアクセスが可能である。キャッシュにアクセスするスレッドが毎サイクル切り替わるため局所性に乏しく、容量を大きくしなければならない。よってキャッシュメモリのアクセスを高速

化するには、メモリセルアレイを複数に分割しパイプライン化するなどの工夫 [2] が必要である。

ミス時はスレッド ID をバッファ(スレッドバッファ) に格納し、スレッドを無効化して以降のステージではそのスレッドに対しては何もしない。ブロック転送終了後、スレッド ID をバッファから復帰させる。復帰後は最初のステージからの再開となる。ブロック転送は他のスレッドをストールさせないように行い、スループットを低下させないためにはキャッシュ-メモリ間のバンド幅が大きくなくてはならない。MUP はブロック転送待ちの間に他の実行可能なスレッドをかわりに入れるような制御は持たない。

1.3.2 MUP のパイプライン

MUP のステージ構成を表 1.1に示す。

表 1.1: MUP のパイプラインステージ

ステージ名				動作
TS1	TS2	-		スレッド選択
IF1	IF2	IF3	IF4	命令フェッチ
RF1	RF2	-		レジスタフェッチ
EX1	EX2	EX3	-	実行
DF1	DF2	DF3	DF4	データフェッチ
WB1	WB2	-		ライトバック

各ステージで行う処理は次のとおり。

1. **TS1:** スレッド ID とスレッド有効ビットを生成する。
2. **TS2:** PC、CRIN(内部例外原因レジスタ)、SR(ステータスレジスタ) を読み出す。
3. **IF1:** 命令キャッシュのアドレスデコード。
4. **IF2:** 命令キャッシュのメモリセルを読み出す。
5. **IF3:** 命令キャッシュのタグを読み出す。
6. **IF4:** スレッドバッファにアクセスする。命令キャッシュがミスだった場合はスレッド ID を退避しスレッドを無効化する。
7. **RF1:** 命令デコード。
8. **RF2:** 汎用レジスタ、PC、CRIN、SR、EPC(例外 PC)、CREXT(外部例外レジスタ) を読み出す。

9. **EX1:** ALU の第 1 ステージ (桁上げ生成ビット、桁上げ伝搬ビット生成)。
10. **EX2:** ALU の第 2 ステージ (桁上げ先見)。
11. **EX3:** ALU の第 3 ステージ (結果生成)。
12. **DF1:** データキャッシュのアドレスデコード。
13. **DF2:** データキャッシュのメモリセル読み出し/書き込み。
14. **DF3:** データキャッシュのタグを読み出す。
15. **DF4:** PC 計算の第 1 ステージとスレッドバッファアクセス。
16. **WB1:** PC 計算の第 2 ステージ。
17. **WB2:** 汎用レジスタ、特殊レジスタへの書き込み。

高周波数で動作させるために各ステージの論理段数を小さく (3〜5 段) 抑えている。これにより 0.1μm CMOS テクノロジ上で 1GHz での動作が可能である。

1.3.3 MUP をさらに高速に動作させるには

MUP をさらに高速化 (動作周波数を向上) するには、ラッチを追加してパイプラインをさらに細分化するなどして、遅延が最も大きなステージの遅延を小さくする必要がある[1]。しかし、ステージの最大遅延を短縮したり、新たにラッチを追加しようとすると次のような問題に直面する。

問題 1 MUP はパイプラインを細分化することで高周波数動作を実現するため各ステージの最大ゲート段数を最小限に抑えており、これ以上ステージの遅延を短縮するのは困難。また、クリティカルパスがメモリアクセス部分である場合など遅延を小さくできないステージがあることも考えなければならない。

問題 2 遅延の小さなステージを遊ばせないように (最大) 遅延が均衡するようにステージ分割するのは困難。

問題 3 高周波数で動作させるとクロックスキューの影響が大きくなり、クロックの均等な分配が困難になる。

これらの問題を回避する手段の一つとして、特にマルチスレッド型プロセッサにおいてはウェーブパイプラインの導入が効果的と考える。

[1] 通常のパイプラインではステージの最大遅延でサイクルタイムが決まるため。

第 2 章

ウェーブパイプライン

ウェーブパイプライン[1]では各ステージは完全に処理を終える前に次の処理を開始する。そのため、レイテンシよりも小さなサイクルタイムで動作するが、一方でステージ内に複数のデータが同時に存在することになるためタイミング制約が厳しくなる。

ウェーブパイプラインを採用することで、機能ユニット単体など比較的小規模な回路では通常のパイプラインに比べて動作周波数は 2〜7 倍に向上する [7]。

2.1 ウェーブパイプラインの基本原理

まずウェーブパイプラインの基本原理について概念的な図を用いて説明する。S1〜S8 の 8 ステージから成るパイプラインの各ステージの遅延の様子が図 2.1のようになっていると仮定する。斜線部[2]はステージ内部の論理素子が動作する可能性のある部分を表す。よってこの部分が重ならないようにデータを送り込めば正常な動作が保証される。

通常のパイプラインでは、最も遅延の大きなステージの最大遅延でサイクルタイムが決まる。ここでは S3 の最大遅延でサイクルタイムが決まる。この時のパイプラインの動作の様子を図 2.2に示す。全てのステージが同じタイミングで動き、また、どの時刻にもステージ内部には 1 つのデータしか存在しない。

図 2.2は斜線部の間の隙間が大きく、これは遊んでいる状態の素子が多いことを示している[3]。そこでこの隙間を詰めていくと図 2.3の様になり、最大遅延と最小遅延の差が最大となるステージの遅延差にまでサイクルタイムを短縮できる。ここでは S5 の遅延差でサイクルタイムが決まる。これがウェーブパイプラインである。通常のパイプラインよりも

[1] maximum rate pipelining とも呼ばれ、文字通りパイプライン動作その速度を最大限にまで上げる方式である。

[2] 実際にはこのような単純な形状になることはないが、簡単のため三角形で表す。

[3] 典型的な CMOS 回路では素子が idle になっている時間はサイクルタイムの 90%を越える [9]。

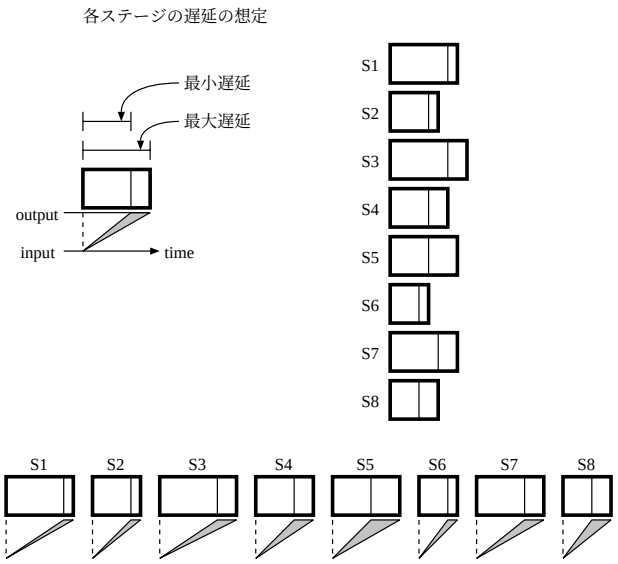

図 2.1: ステージの遅延

短いサイクルタイムで、各ステージが別々のタイミングで動く[4]ことがわかる。また、ステージ内部のデータが前後のデータとオーバーラップしており、複数のデータが同時にステージ内部に存在していることになる。図 2.2に比べて斜線部の間が詰まっており、論理素子レベルでリソースの使用効率が良くなっていることを意味している。

通常のパイプライン方式で図 2.3と同じサイクルタイムを実現するにはステージをさらに分割しラッチを追加することが必要であるが、ウェーブパイプラインはラッチの数を増やさずにサイクルタイムを短縮する。

ウェーブパイプラインの主な特徴を以下に示す。

- 動作周波数がステージ内部の遅延差の最大値に依存する。よって最大遅延に依存する通常のパイプライン方式よりも高周波での動作が期待できる。クリティカルパスの遅延を短縮できな場合でも、最小遅延の方を最大遅延に近付けることで高速化が可能である。これは前述の MUP を高速化する上での問題 1 を解決する。

  また、遅延の絶対値がサイクルタイムに束縛されなくなるので、通常のパイプラインの様に最大遅延が均衡するようにステージ分けしなくてもリソースの使用効率を高く保てる。つまり前述の問題 2 を解決するものである。

- 全てのステージ間ラッチへ入れるクロックが同位相である必要がない。よって全てのラッチへ均等にクロックを分配する困難を回避する手段としてウェーブパイプラ

[4]図 2.3中の d1～d8 は、各ステージが動作するタイミングが最初のステージに比べてどの位ずれているかを表す。ステージの動作タイミングを調整する方法には幾つかある (後述)。

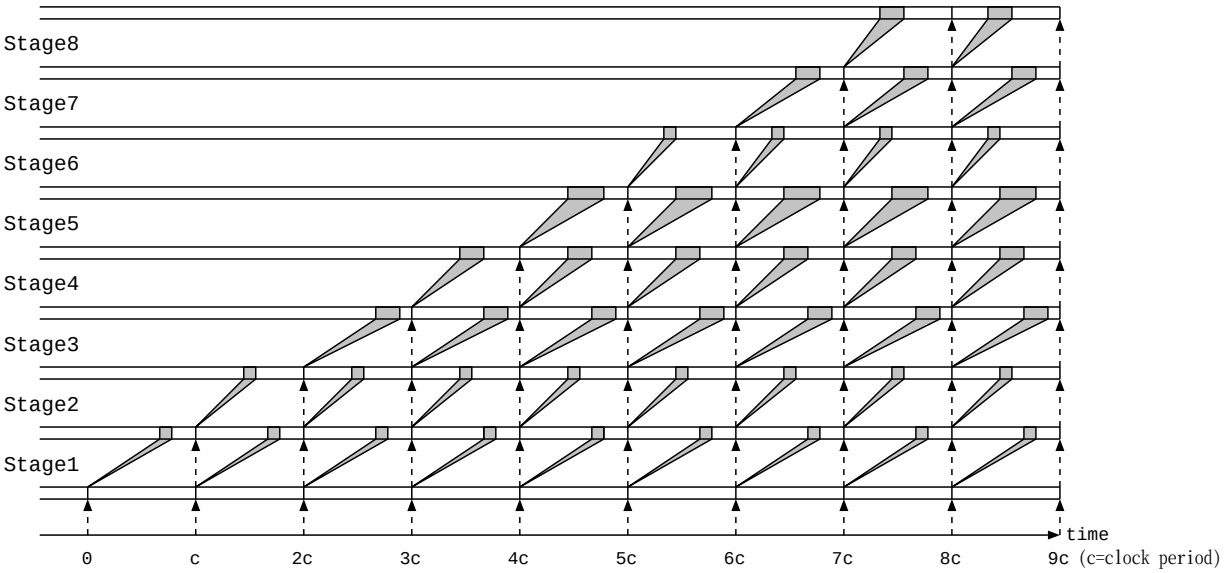

図 2.2: 通常のパイプライン

インが有効になることがある。これは前述の MUP を高速化する上での問題 3 を緩和する。

一方、データはラッチに保存されずに論理素子の遅延を記憶要素の様に扱う (素子遅延のために素子に一時的にデータがたまるのを利用する) ので、タイミング制約が厳しくなり、例えばクロックオーバーヘッドは通常のパイプラインの 2 倍の影響を持つようになる。

- 通常のパイプラインに比べて、目標の動作周波数の達成に必要なラッチ数が少ない。これもクロックの均等な分配の問題、つまり前述の問題 3 を緩和する。
- 論理ゲート単位のレベルでリソースの使用効率が向上する。パイプライン処理 (機能ユニットのレベルでリソースの使用効率を上げる) よりもミクロのレベルでリソースの使用効率を上げるものとみることができる。

しかし、設計にあたっては以下のような問題がある。

- 一般的な設計法でないために専用の CAD がない。遅延差の最大値を短縮する方法や、既存の CAD をうまく利用する方法を考えなければならない。
- 慎重に遅延モデルを考える必要がある。精密過ぎず、かつある程度現実的なシミュレーションを行うために、素子や配線の遅延を適当にモデル化しなければならない。
- ウェーブパイプライン動作のテストや検証法なども確立されていない。

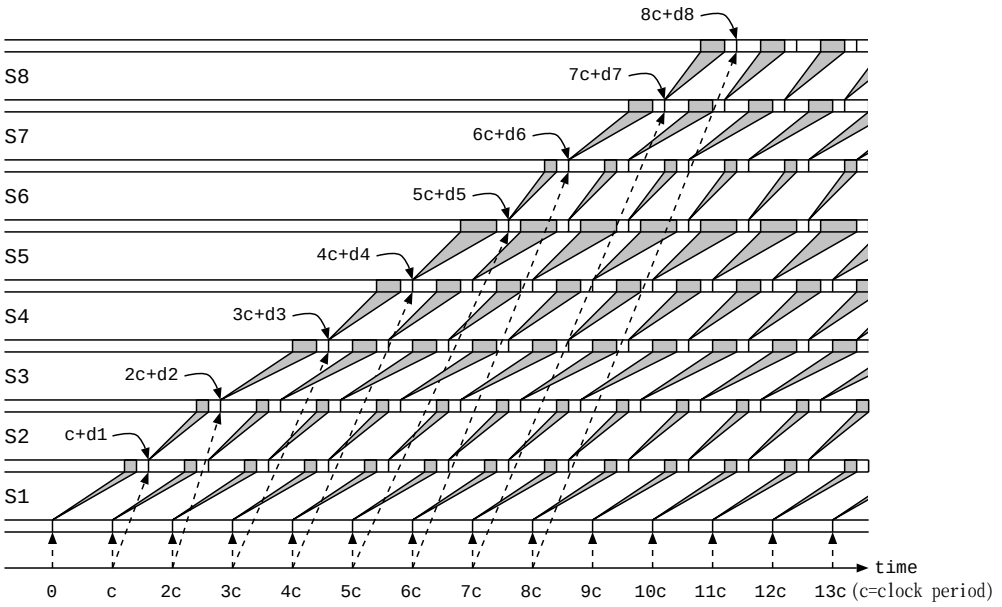

図 2.3: ウェーブパイプライン

2.2 マルチスレッド型プロセッサとウェーブパイプライン

マルチスレッド型プロセッサがウェーブパイプラインに適している点としてそのパイプライン構造が挙げられる。

前述のようにウェーブパイプラインでは各ステージがばらばらのタイミングで動作できるが、一般的なシングルスレッドのプロセッサの場合、図 2.4(a) の様に EX ステージにフォワーディングのためのフィードバックパスがあり、前後のラッチの動作タイミングを合わせなければならない。よってクロックに合わせてステージをウェーブ動作させるために図の様に遅延素子を加えるなどの工夫が必要である。しかしこれは大きなコスト増になる。一方マルチスレッド型プロセッサにはそのようなパスがないので、(b) のように単純にウェーブパイプライン化することができる。

また、特に MUP のような単純な構造 (キャッシュミスの扱い方、スレッド切り替えの単純さ) のプロセッサはウェーブパイプライン (CAD がない、タイミング制約が厳しい) の導入にとって非常に都合が良いものである。

次章以降、ウェーブパイプラインのための設計手法を検討し、それらを MUP の適用していく。

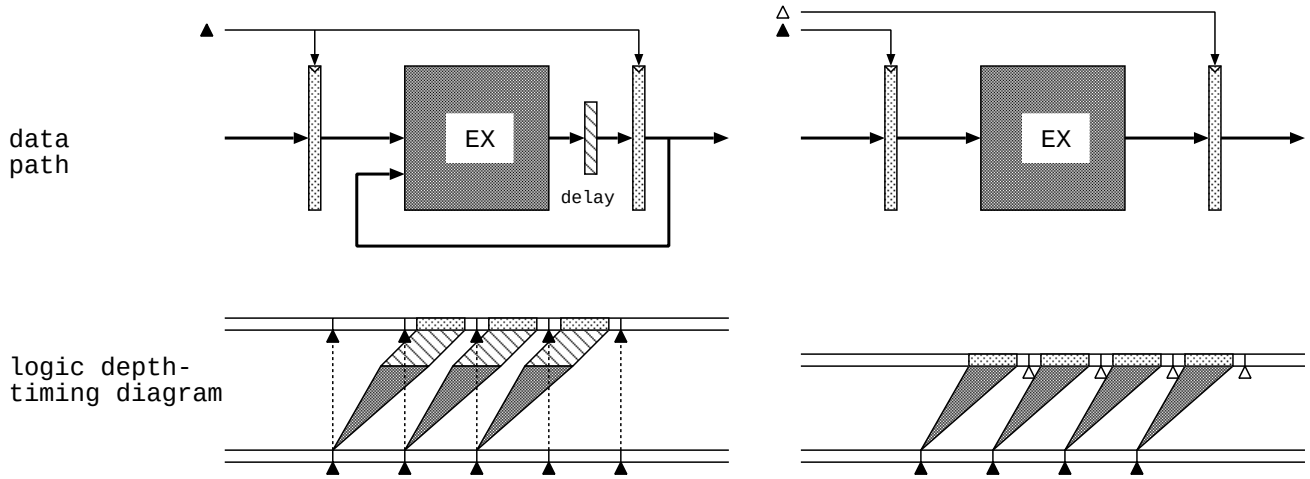

図 2.4: フィードバックするパスがある場合とない場合

第 3 章

ウェーブパイプラインのための設計

[1] では、1GHz で動作することを目標に設計している。その結果が 17 というスレッド数であり、3〜5 段というステージ当りのゲート段数であった。ウェーブパイプラインの場合、スレッド数[1]やサイクルタイムが、各ステージの遅延をどこまで均衡させられるかということに相互に依存し合っているため、目標としてサイクルタイムやスレッド数をあらかじめ決めにくい。そこで、最初にステージ構成を決めてしまい、その後の各段階で高速に動作するための設計を行っていく過程でサイクルタイムやスレッド数やウェーブ数を決める。

よって最適なウェーブ数やスレッド数についてはここでは考えない。

3.1 設計の流れ

以下に設計の大まかな流れを示す。各設計手順については次節以降、順を追って説明する。

準備 論理素子や配線を適切にモデル化し、必要なパラメータを与える。また、モデルに基づいて遅延を評価する方法を決める。

1. ステージ構成 (ラッチの配置) を決める。
2. スレッド数 (> ステージ数) を仮に決め、それをもとにレジスタセット、キャッシュの設計を行う。
3. 最も遅延差が大きくなりそうなステージの遅延を均衡させる。
4. サイクルタイムを決定する。ここでスレッド数 (= 総ウェーブ数) が決まる。しかしこれが 2 で決めた仮の値と大きく違っていた場合は 2 からやり直す。

[1] 通常のパイプラインではスレッド数 = ステージ数だが、ウェーブパイプラインではステージ内部に同時に存在するデータ数 (ウェーブ数) が 1 を越えるので、スレッド数 = 総ウェーブ数 ≧ ステージ数となる。

5. 決定したサイクルタイムでの動作を目標に、他のステージの遅延を均衡させる。

6. 各ステージの遅延状況をもとに、ステージの動作タイミングを調整する。

1 は第 4 章、3,5 は第 5 章で主題とする。

3.2 素子/配線モデルとパラメータ

まず準備として素子や配線をモデル化し、必要なパラメータを与える。

本研究では設計ツールとして PARTHENON[13] を使うため、そのネットリスト記述言語である nld が使用する標準的なセルライブラリと同様に、素子遅延は (固定成分 + 付加容量に比例する成分) で表す。全ての論理素子について、遅延式の他、面積、ドライブ能力、全ての外部端子容量を与える。レジスタについても遅延式の他、面積、セットアップ時間、ホールド時間、ドライブ能力、全ての外部端子容量を与える。

配線遅延は簡易レイアウトの結果から得た配線長に比例した値を、その配線をドライブする素子の出力端子の付加容量に加味する[2]。よって素子遅延はインスタンス毎に異なり、その接続状況やレイアウト結果によって変化する。

各パラメータは 0.25μ CMOS プロセスを想定し、HSpice シミュレーションにより得る。

コストを評価するためには欠陥率などの値も必要になるが、これについては第 6 章に記す。

3.3 遅延の測定方法

遅延の測定は上のモデルとパラメータに従って行う。

ステージ内部の遅延差の最大値を得るには、単純にデータの入力から出力までの最大時間と最小時間を計測するだけではいけない。ステージ内部のどこで遅延の差が最大になるかわからないからである。

遅延差が最も大きくなるのは必ずしも出力端子であるとは限らず、図 3.1のように回路の中ほどで遅延差が最大になることもあり得る。

遅延を均衡させたり、サイクルタイムを決めるには、ステージ内部の全ての場所における最小・最大遅延がわからなければならない。そのため全てのゲートに対して入力端子からの遅延を見積もり、それをもとに遅延の均衡、サイクルタイムの決定を行う。

[2]詳細は Appendix B 参照。

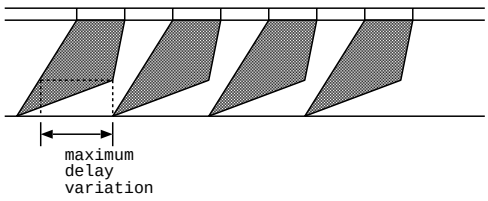

図 3.1: 遅延差が最大になる部分

3.3.1 遅延の見積もり

ステージの入力からあるゲートまでの最小・最大遅延を見積もりたいとする。ステージの入力からそのゲートまでの全てのパスについて、そのパスが通過するゲート遅延 (配線遅延分を含む) の和を計算する。パスに含まれる全てのゲート遅延を min{ 立ち上がり時間, 立ち下がり時間 } として計算したものをそのパスの最小遅延、パスの全てのゲートの遅延を max{ 立ち上がり時間, 立ち下がり時間 } として計算したものをそのパスの最大遅延とする。このようにして全てのゲートにおける最小・最大遅延を見積もる。

例えば図 3.2のような回路の場合、入力端子 A から出力までのパスの最大遅延は $D_A = \max\{D3, d3\}$、同様に B,C からのパスの最大遅延は $D_B = D_C = \max\{D1, d1\} + \max\{D2, d2\} + \max\{D3, d3\}$、D からのパスの最大遅延は $D_D = \max\{D2, d2\} + \max\{D3, d3\}$ となり、g3 における最大遅延は $\max\{D_A, D_B, D_C, D_D\}$ と見積もる。最小遅延も同じ要領で見積もる。

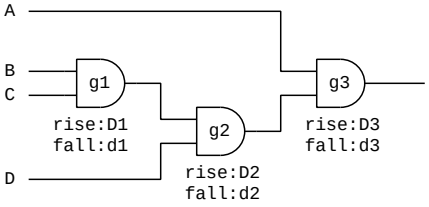

図 3.2: 遅延の見積もりの例

論理の関係やデータ依存によって実際には活性化しないパス (false path) があったり、パス上の全てのゲートが min{ 立ち上がり時間, 立ち下がり時間 } または全てのゲートが max{ 立ち上がり時間, 立ち下がり時間 } で動作することはない[3]のでこの見積もりは正確な値ではないが、見積もられた最大遅延値は十分に大きく、最小遅延値は十分に小さい。よって遅延差の見積もりは十分に大きな値になり、これをもとに決めたサイクルタイムは

[3]NAND の後に NAND がある場合などを考えれば明らか。立上り時間と立下り時間がほぼ等しくなるようにトランジスタサイズを調整した素子を使えば、一つのパスについて遅延値が一意に決まり、問題が簡単になる。

正常な動作に十分な時間である。

この方法の利点の一つとして、遅延差が最大になるのが常にステージの出力端子であることが挙げられる。そのため見積もり上、図 3.1のような状況になることがない。

(**証明**)
ある n 入力のゲート gを考える。その入力 i1,i2,$\cdots$,in に値が到着する時刻がそれぞれ T_{e_1}〜T_{l_1},$\cdots$,T_{e_n}〜T_{l_n}のようにばらついていたとする。gの出力が遷移する時刻は最も早くて $\min_i\{T_{e_i}\}+(g$の遅延)、最も遅くて $\max_i\{T_{l_i}\}+(g$の遅延) である。よって gの出力時刻のばらつきは、その差 $\max_i\{T_{l_i}\}-\max_i\{T_{e_i}\}$ である。一方、gの前段のゲートの出力時刻のばらつきは最大のもので $\max_i\{T_{l_i}-T_{e_i}\}$ である。$\max_i\{T_{l_i}\}-\max_i\{T_{e_i}\}\geq \max_i\{T_{l_i}-T_{e_i}\}$ であるから、gの出力時刻のばらつきは必ず gの前段のゲートの出力時刻のばらつきのどれよりも大きくなる。よってステージの出力に近い程ばらつきは大きくなる。

これが真であるのはデータ依存を考慮していないからである。

入力値の変化が出力に反映されるまでの遅延時間が入力端子毎に異なるような素子の場合、タイミングによっては入力側と出力側で遅延差が違ってくることがある。図 3.3で、i1,i2 の変化が出力に反映されるまでの時間をそれぞれ 1,2 とする。i1 にデータが到着する時刻が 10〜12、i2 にデータ到着する時刻が 8〜10 のようにばらついていたとすると、この場合入力側での遅延差は $12-8=4$ である。一方、出力からデータが出る時刻は i1 経由で 11〜13、i2 経由で 10〜12 となるので、出力側での遅延差は $13-10=3$ である。このように入力側の方が遅延差が大きくなるケースがあるので、遅延差は素子の入力側で評価しなければならない。

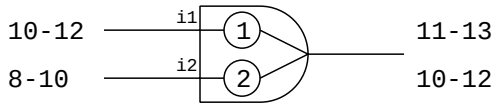

図 3.3: ゲートの入力側と出力側で遅延差が異なる例

第 4 章

ステージ構成

本章では、マルチスレッド型プロセッサをウェーブ動作されるためのステージ構成を考え、それを MUP の適用しウェーブパイプライン版 MUP のステージ構成を決める。

プロセッサ全体を理想的にウェーブ動作させることができればラッチなしでパイプライン動作し、ウェーブパイプラインの恩恵を最大限に享受できるが、通常はラッチが不可欠な場所が存在するためにそれは不可能である。ラッチを置く意味は次の 2 つある。

- 機能ユニットへのアクセスタイミングを調整する。
- 遅延差を一旦 0 に戻すことによって、遅延差の広がりを抑える。

状態機械である以上、記憶要素に関しては、その使用が競合したり操作の順序が前後したりしないように使用にあたってタイミングを調整する必要がある。よって記憶要素の入力側にはラッチを置くのが望ましい。また、効率的に遅延差を短縮できそうにない場所には場ラッチを置く必要が出てくることもあるだろう。必要な場所にラッチを置き、その場所以外はウェーブ動作する、つまりラッチ無しで動作するとすれば、これでステージ分けができる。ただしこの時点ではスレッド数 (= 総ウェーブ数) はまだ未知である。

4.1 レジスタの扱いとラッチの配置

複数のスレッドが共有しているレジスタへのアクセスはスレッド間で競合しないように厳密にとタイミングを調整する必要があるため、この種のレジスタの前にはラッチが必要である。ステージ内部にこのようなレジスタがある場合、そのステージを (レジスタの遅延 + 回路が安定するまでの時間) よりも小さなサイクルタイムで動かすことはできない。図 4.1はステージ内部に複数のスレッドで共有されるレジスタがあるためにサイクルタイムが遅延差で決まらない例である。図中の白い部分[1]がレジスタを表している。この

[1] 記憶要素の遅延のばらつきは一般の組合せ論理よりも小さいと考えられるので細い形状にしてある。

部分は組合せ論理部分のように前後のデータとオーバーラップさせることができず、さらにデータ間にはレジスタを安定させるためのレジスタオーバーヘッド (hold time + setup time) 分の時間間隔をおく必要がある。(a) のようにステージの遅延の中でレジスタが占める割合が小さな場合は遅延差の最大値でサイクルタイムが決まるが、(b) のようにレジスタの遅延が占める割合が大きな場合はレジスタの遅延とオーバーヘッドの和の方が大きくなり、こちらがサイクルタイムを決める。そのため (b) 上のような動作はできず、(b) 下のように動作する。

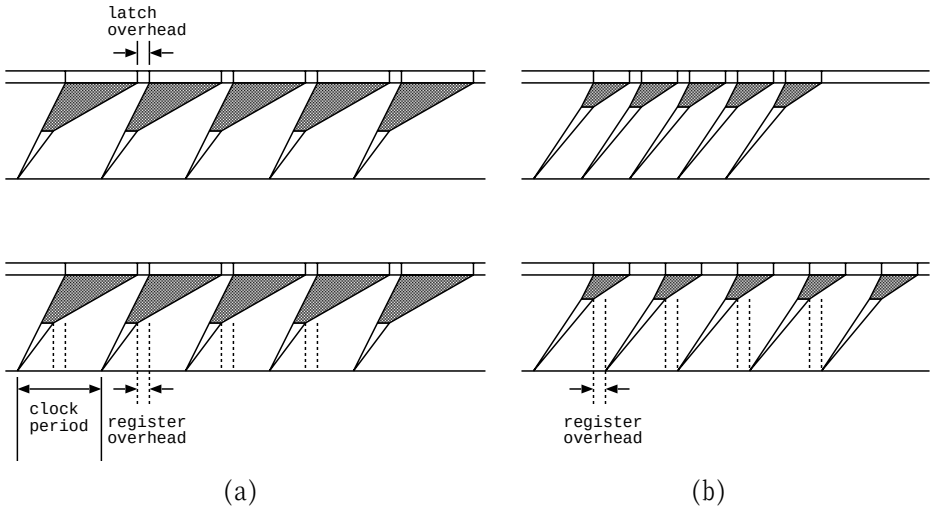

図 4.1: 記憶要素の扱い

マルチスレッド方式では、汎用レジスタファイルなどはスレッド毎に用意される。スレッド毎に用意されるレジスタに関してはタイミング制約は緩やかになる。これらのレジスタは連続したサイクルでアクセスされることがない。よってあるレジスタにアクセスした後、そのレジスタの set up 時間だけ待てば次のデータを受け入れることができ、図 4.1(b) 上のような動作が可能である。

また、このようなレジスタの使用に関してはスレッド間でタイミングを合わせる必要がないので、必ずしもレジスタをクロックに同期させる必要はなく、レジスタを駆動するトリガをデータと共に送ることが考えられる。図 4.2はレジスタの同期/非同期によって生じる違いを、サイクルタイムが遅延差によって決まる場合とレジスタの遅延によって決まる場合に分けて示している。同期的にすると、ラッチオーバーヘッドの分だけレイテンシが増加するが、最大遅延と最小遅延の両方に等しく加わるので遅延差に影響することはなく、ラッチの効果で遅延差が短縮するのでサイクルタイムは小さくできる。非同期的にすると、必要なラッチ数が減りステージのどこにレジスタを置いてもよくなるので設計に柔軟性が持てるが、トリガとデータのレーシングに気を付けなければならなくなる。ここで

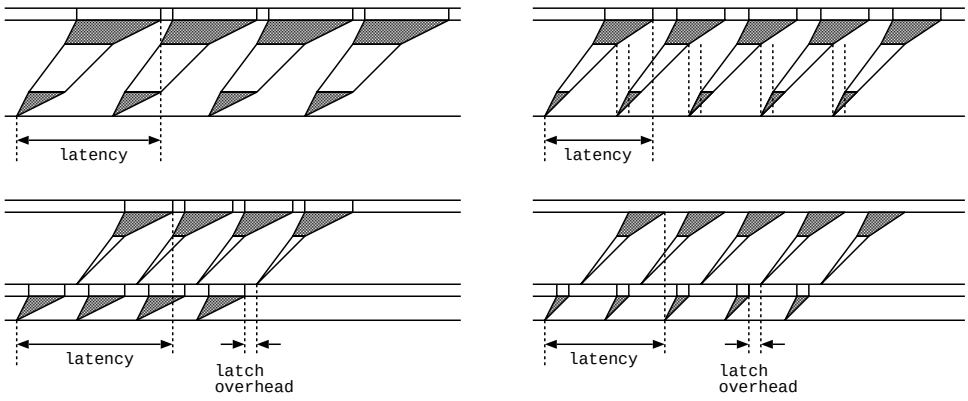

図 4.2: レジスタの同期/非同期

は同期式のレジスタを採用し、レジスタの前には常にラッチを置く。結局、全てのレジスタの前にはラッチを置くことになる。よって使用する記憶要素と、それを用いる順番が決まればラッチが必要な場所が決まる (暫定的なステージ分け)。

この種のラッチが多いと効果的なウェーブ動作の支障になるので、記憶要素へのアクセス回数は少ない方が良い。そのため、依存関係のない記憶要素は同時にアクセスするようにし、レジスタ類の使用はできるだけまとめて行う。

暫定的なステージ分けができたら、明らかに遅延差が他のステージに比べて大きくなりそうなステージがないか調べ、もしあれば、そのステージの途中に遅延差を 0 にする目的でラッチを入れる。この時点でラッチの配置、つまりステージ分けが終了する。

レジスタと同様、キャッシュも記憶要素だが、これに関してはここでは詳細な議論はしない。キャッシュのメモリアクセス部分がウェーブ動作できればよいが、ここではウェーブ動作しないものと仮定する。ウェーブ動作するキャッシュについては [11] などがある。[8] の vector register file の構成法も参考になるだろう。

4.2 ステージの動作タイミングの調整

各ステージの動作タイミングを調整しないとステージの遅延差で決まるサイクルタイムによる動作はできない。

動作タイミング調整の例を図 4.3に示す。(a) は調整しない場合、(b) はステージの遅延をクロックに合わせて調整した場合、(c) はステージの遅延に合わせてクロックを調整した場合[2]、(d) は 2 相クロックによって調整した場合である。クロックオーバーヘッドなど

[2]意図的にクロックパスに加えるスキューは、不可避なクロックスキューと区別するために constructive

を考えず、単純にステージの遅延だけを考慮すると、(a) は 3.25ns、それ以外は遅延差の最大値に等しい 3ns のサイクルタイムで動作する。(b) はステージ内部の全てのデータパスに遅延素子を入れることになるので回路量の増加が大きいが、使用する記憶要素などの関係上クロックを調整できない場合には有効である。(c) はクロックパスに遅延を挿入することでタイミングを調整する。例えば図 2.3中の d1〜d8 に相当する遅延を各ステージへのクロックパスに加える。(c) のようにクロックを分配する場合、データが一方向にしか流れないマルチスレッド型プロセッサのパイプラインは非常に都合が良い。また (c) を見ると 1 番目のラッチと 3 番目のラッチのずれが 3ns でサイクルタイムに等しい。このような場合には (d) が有効である。また、使用する記憶要素の関係で同時に動かさなければならないラッチが多い場合にも有効である。

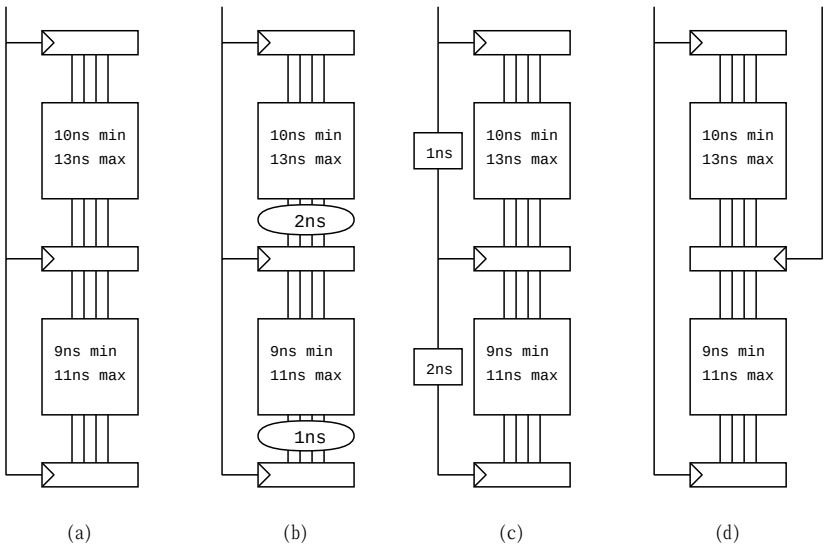

図 4.3: ステージの動作タイミングの調整

ラッチへのクロックが様々に異なる場合は、クロックパスに遅延を入れるのが最も簡単だろう。ラッチが動作するタイミングが数通りしかない場合には複相クロックを使う方が良い。

4.3 MUP への適用

以上の事項を考慮してウェーブパイプライン版 MUP のパイプラインを表 4.1のようなステージ構成にした。使用する機能ユニットをはじめ、キャッシュミスの扱い、割り込み方式などはオリジナルの MUP と同一とする。

skew または intentional skew と呼ばれる。

キャッシュに関しては以下を想定している。これは [4] による。

[**キャッシュの想定**]　アドレスデコードからデータ排出/書き込みまでをキャッシュが請け負うものとする。9 ステージにパイプライン化されており、通常動作 (非ウェーブ動作) するものとする。ヒット/ミスを含む全ての出力は 9 サイクル目に同時に排出されるものとする。オリジナルの MUP とは異なり、バスの占有率を下げるためにライトバック、ライトアロケートを採用する。

表 4.1: ウェーブ動作版 MUP のパイプラインステージ

ステージ名	動作
TS	スレッド選択
IF1〜IF9	命令フェッチ
ID	命令デコード
RF	レジスタフェッチ
EX	実行
DF1〜DF9	データフェッチ
WB	ライトバック

レジスタの前とキャッシュの前後にはラッチがあり、これら記憶要素への入力は全てラッチから直接 (組合せ論理を介さずに) 入るようにする。各ステージの動作を、使用する記憶要素を挙げて説明する。

1. **TS**

   オリジナルの MUP ではレジスタファイルとスレッドバッファは別々のステージでアクセスしていたが、4.1 節での考察により、同ステージ内でアクセスするようにした。

   - **I-TVF, D-TVF (read)**　命令/データキャッシュ用スレッド有効フラグレジスタ (オリジナルの MUP のスレッドバッファに相当)。round-robin に生成されたスレッド ID の有効/無効を確認する。スレッド毎に用意される。
   - **PC (read)**　プログラムカウンタ。スレッド毎に用意される。
   - **CREXT (read)**　外部例外レジスタ。外部例外は全てのスレッドに対して有効なので、CREXT は一つしか存在せず全てのスレッド間で共有される。
   - **SR (read)**　ステータスレジスタ。動作モード (ユーザ/カーネル) や例外の発生状況/マスクなどの情報を持つ。スレッド毎に用意される。

2. **IF1〜IF9**

   IF1〜IF9 は命令キャッシュのパイプラインにあてられる。キャッシュはレイアウト上、物理的に孤立しているので前後にラッチを置く。プロセッサ側としては、IF1 で命令キャッシュにアドレス、スレッド ID、スレッド有効フラグを渡し、IF9 でデータとヒット / ミス情報を受け取る。

3. **ID**

   命令デコードを行う。このステージでは記憶要素は使用しない[3]。

4. **RF**

   ここで使用されるレジスタは全てスレッド数個分用意されているものばかりなので、このステージは図 4.2(b) 上のような動作が可能である。このような動作ができるのは、ウェーブパイプラインによる恩恵というよりむしろマルチスレッド方式による恩恵である。

   - **GR (read)**　汎用レジスタファイル。スレッド数個分用意されるので面積が大きく、レイアウト上孤立させる必要がある。そのため GR を含む RF ステージはキャッシュ同様、出力側にもラッチを要する[4]。
   - **I-TVF (write)**　命令キャッシュ用スレッド有効フラグレジスタ。命令キャッシュのヒット / ミスに従ってスレッドの有効/無効の情報を書き込む。スレッド毎に用意される。
   - **CRIN (read)**　内部例外原因レジスタ。スレッド毎に用意される。
   - **EPC (read)**　例外 PC。スレッド毎に用意される。

5. **EX**

   命令実行 (ALU) と分岐先アドレスの計算をする。記憶要素は使用しない。

6. **DF1〜DF9**

   DF1〜DF9 はデータキャッシュのパイプラインにあてられる。プロセッサ側としては、DF1 でデータキャッシュにアドレス、スレッド ID、スレッド有効フラグ、書き込みデータ、LD/ST、W/H/B[5]を渡し、DF9 でデータとヒット / ミス情報を受け取る。

[3] 後に第 7 章でこのステージにも記憶要素が入れることになるが、ステージ構成には影響しない。
[4] 4.1 節に従ってラッチを配置した場合 RF と EX の間にラッチは入らない。
[5] ワードアクセス、ハーフワードアクセス、バイトアクセスを指定。

7. **WB**

全てのスレッドで共有される CREXT を使うこのステージはウェーブ動作できない。もしくは wave degree (ステージ内に同時に存在する平均ウェーブ数) = 1。

- **GR (write)**　汎用レジスタファイル。
- **D-TVF (write)**　データキャッシュ用スレッド有効フラグレジスタ。
- **EPC (write)**　例外 PC。
- **CRIN (write)**　内部例外原因レジスタ。
- **CREXT (write)**　外部例外レジスタ。全てのスレッドで共有される。

以上、全 23 ステージ構成で、ウェーブ動作するのは TS,ID,EX の 3 ステージである。よってスレッド数は 20 + (TS ステージの wave degree) +(ID ステージの wave degree) + (EX ステージの wave degree) となる。

4.3.1　サイクルタイム

ウェーブ動作しない IF1〜IF9、DF1〜DF9、WB が動作可能な最短周期はこれらの最大遅延で決まる。これを T_1 とする。

$$T_1 = \max\{\text{ キャッシュのサイクルタイム}, WB\text{の最大遅延 }\} \tag{4.1}$$

TS ステージはウェーブ動作するが、CREXT(全てのスレッドが共有) を持つため、動作可能な最短周期 T_2 は 4.1 節より、

$$\begin{aligned} T_2 \;=\; & \max\{CREXT\text{のアクセス時間} + CREXT\text{のレジスタオーバーヘッド}, & (4.2)\\ & TS\text{ステージの遅延差の最大値 }\} & (4.3)\end{aligned}$$

となる。ID ステージはウェーブ動作し、内部に記憶要素を持たないので、遅延差の最大値で動作可能な最短周期が決まる。これを T_3 とする。

$$T_3 = ID\text{ステージの遅延差の最大値} \tag{4.4}$$

RF ステージはレジスタアクセスしかない上、全てオーバーラップさせられるので、このステージの動作可能な最短周期 T_4 は

$$T_4 = \max\{\text{ レジスタの遅延差, レジスタのセットアップ時間 }\} \tag{4.5}$$

となる。ID ステージと同様に EX ステージもウェーブ動作し、内部に記憶要素を持たないので、遅延差の最大値で動作可能な最短周期が決まる。これを T_5とすると、

$$T_5 = EX\text{ステージの遅延差の最大値} \tag{4.6}$$

となる。T_1 〜 T_5の中で最大のものがサイクルタイム T_c を決める。

$$\begin{aligned} T_c &= \max\{T_1, T_2, T_3, T_4, T_5\} && (4.7)\\ &+ \text{ラッチオーバーヘッド} && (4.8)\\ &+ \text{クロックオーバーヘッド} \times 2 && (4.9) \end{aligned}$$

4.3.2 ステージの動作タイミング

CREXT を使う TS と WB には同じクロックを入れるが、WB の入力側のラッチはデータキャッシュ出力のラッチと同じになるので、結局 TS、DF1〜DF9、WB に同じクロックを入れることになる。同様に、ID の入力側のラッチは命令キャッシュ出力のラッチと同じになるので、IF1〜IF9、ID に同じクロックを入れる。

RF ステージは他のステージとタイミングを合わせる必要がないので別のクロックを入れる。

残りは EX で、これにはまた別のクロックを入れる。

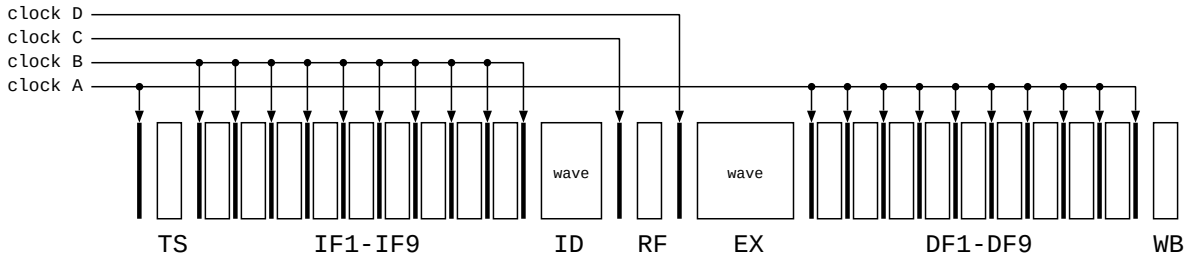

図 4.4: 各ステージの動作タイミング

以上、図 4.4のように合計 4 系統のクロックが必要である。23 ステージ構成で、必要なクロックが 4 系統ということを考えると 4 相分のクロックを用意した方が良いだろう。

このようにステージの動作タイミングを決めた結果、TS,ID,RF,EX の各ステージの遅延の和がサイクルタイムの倍数でなければならないなどの制約が生じる。これには図 4.3(b) の方法で対処できる。

第 5 章

ステージ内部の回路設計

前述のように、ウェーブパイプライン動作の周波数を上げるには遅延差を短縮することである。よって遅延を均衡させること (delay balancing) が動作周波数向上の鍵となる。そのためには遅延差が小さくなるようにステージ内部の回路を設計することが必要である。

5.1 遅延にばらつきが生じる原因

遅延のばらつき (delay variation) の原因には、ファンイン/アウト数、配線長、端子の負荷容量などの違いの他、電源電圧、トランジスタサイズ、温度変化、ノード電位とゲート入力の状態の違い (データ依存)、製造プロセスのばらつき、電源電圧や信号のノイズなどが挙げられる。

どの原因をターゲットにするかで delay balancing の方法が異なる。

5.2 遅延を均衡させる方法

遅延のばらつきを抑える方法としては以下が考えられる。

- 遅延の小さなパスに遅延素子を挿入する。遅延素子の遅延に比べて非常に小さな遅延差は解消できない。
- トランジスタのサイズを調整して素子の遅延を調整する。ある程度遅延の均衡した比較的小規模な回路に適している。
- 遅延が均衡するような論理に組み直す。CAD のサポートがなくては困難。
- 使用するセルの種類を制限する。多入力ゲートや複合ゲートなどの使用は最大遅延の短縮には効果的だが、遅延差の短縮には効果がなく、むしろ遅延を均衡させるプロセスを複雑にしてしまう。

本研究では、ウェーブパイプライン用の CAD がないこと、対象がプロセッサ全体と大規模であることから、遅延素子の挿入に限定する。遅延素子としてはバッファを使用するが、ドライバ用途で使用されるバッファと区別するために以降、遅延素子の用途で使用するバッファを遅延バッファと呼ぶ。

前節で挙げた遅延のばらつきの原因のうち、遅延バッファを挿入することで解決されるのは、ファンイン/アウト数、配線長、端子の負荷容量の違いである。バッファを挿入するだけでもかなりの遅延均衡が可能であると考える。

対象とする回路は、遅延バッファを挿入する前から論理レベルである程度遅延が均衡していること、遅延を均衡させるのに必要な遅延バッファが少なくて済むことが望ましい。例として加算器を考えると、

- 最大遅延に合わせて遅延バッファを入れることを考えると、最大遅延が小さい方が遅延バッファが少なくて済む。よって RCA (Ripple Carry Adder) は不適当。
- レイアウトなどまで考慮して効果的に遅延バッファを入れるには規則的な構造の方が良い。

といったことが言える。そこで、RCA よりも遅延が小さく、CLA (Carry Lookahead Adder) よりも規則的な構造を持つ BLC (Binary Lookahead Carry) 加算器[1]を MUP の加算器として考え、次節ではこれを例に遅延バッファの挿入を試みる。

5.3 遅延バッファの挿入

5.3.1 ステージの遅延を均衡させるには

目標となる遅延差が決まっていれば、必要以上に遅延バッファを入れてしまうのを防ぐことができる。よって、まず最も遅延差が大きくなりそうな (遅延が均衡しそうにない) ステージに対して delay balancing(遅延バッファの挿入) を施す。その結果得られた遅延差にラッチやクロックのオーバーヘッドを加えた値でサイクルタイムが決まるので、他のステージに対してはこの遅延差を目標に delay balancing を施す。特に最大遅延がこのステージの遅延差よりも小さなステージでは、delay balancing の必要がなくなる[2]。

delay balancing の対象となるのは記憶要素以外の部分、つまり組合せ論理部分と、論理素子を経由しないでステージを通り過ぎるだけのパス (以降、スルーするパスと呼ぶ) である。

[1] Brent and Kung parallel adder などとも呼ばれる。

[2] 3.1 節に示したように設計手順を決めた理由である。

図 5.1のように、まずスルーするパスに遅延バッファを入れる。スルーするパスの遅延が論理回路部分の最小遅延に等しくなったところで、論理回路内部とスルーするパスの両方に並行して遅延バッファを入れていく。

遅延バッファの数が多くなると、コストの上昇だけでなく、消費電力の上昇、配線遅延の増加にもつながる。よってスルーするパスが少ない程必要な遅延バッファが減りそれらが緩和される。このことについては第 6 章で議論する。

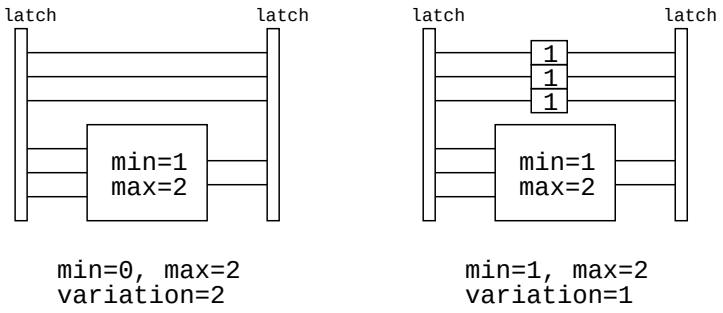

図 5.1: スルーするパスと組合せ論理部分

大きな遅延が必要なパスには大きなバッファを入れればいい訳ではない。ウェーブパイプラインは、素子遅延のために一時的にデータが素子に停滞するのを利用している。1 個の素子に複数のデータを停滞させることはできないので、大きなバッファを入れるとウェーブ数に制限が生じる。

遅延バッファの挿入は、回路内部のあらゆる部分の遅延を均衡させるように行われる要がある。よってステージ内部の全素子の 1 個 1 個を対象に、delay balancing を施していく。

以上のように、ウェーブパイプラインのためには回路内部の全ての素子における遅延情報を基に設計を進める必要がある。全ての素子の遅延情報を動的な (データ依存を考慮した) 解析によって得ようとするのは極めて能率が悪い。よって delay balancing に必要な遅延情報は 3.3 節で示した静的な (素子や配線の接続状況のみを考慮した) 解析によって得る。動作確認を含む動的解析は delay balancing を施した後の回路に対して行う。

> delay balancing で使用する遅延情報を静的解析から得ることに不安を感じるかも知れない。下の表は、遅延のデータ依存性が大きい回路として 4bit の小規模な加算器を例に、3.3 節で示した見積もり方法と Verilog による[3]遅延測定をした結果を示す。

[3] ランダムな入力を与えたシミュレーションを 2000 回実行した。

測定方法	見積もり (max/min)	Verilog(max/min)
遅延挿入前	13.54ns / 3.51ns	13.54ns / 3.51ns
遅延挿入後	14.58ns / 12.88ns	14.58ns / 12.89ns

既に示したように 3.3 節の方法で見積もった値は悲観的なものであるが、表では互いに近い値になっており、この見積もりが悲観的ながらもある程度の正確性を持っていることを示している。

本研究では静的解析のツールとして PARTHENON[13]、動的解析のツールとして Verilog を使用する。PARTHENON は動作記述からネットリストを自動生成できる設計ツールで、現在、MUP は PARTHENON の動作記述言語である sfl で記述されている。

動作記述から遅延バッファによる遅延均衡、簡易レイアウト、遅延測定、動作確認といった作業を含めて以下にその手順のフローチャートを示す。

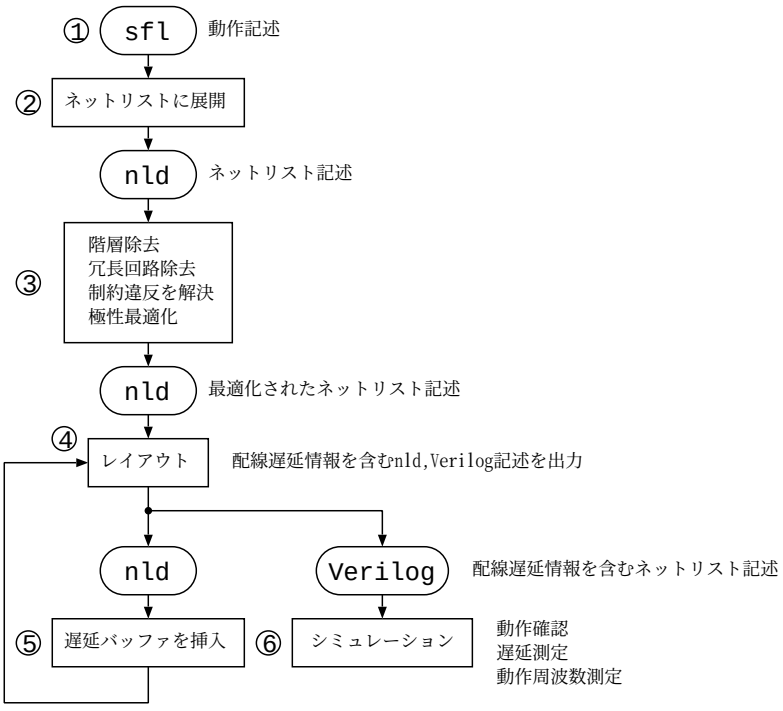

図 5.2: 設計のフローチャート

準備 3.2 節のモデルに基づいて PARTHENON、Verilog それぞれのセルライブラリを作成する。

① 動作記述言語 sfl による動作記述を行う。ステージ毎に独立にネットリストが得られるように記述する。

② ステージ毎にネットリスト (nld) に展開する。PARTHENON に含まれるツールを用いる。

③ 階層を取り払い冗長回路を除去する。制約違反 (ファンアウトの過多など) を解決 (ドライバを入れるなど) した後、極性最適化を施す。これらは PARTHENON に含まれるツールを用いる。

④ レイアウトを行う。1 回目のレイアウト (初期レイアウト) と 2 回目以降のレイアウトとではレイアウト対象が異なる。

> **初期レイアウト**　レイアウト後にバッファが挿入されることを考慮し、挿入される遅延バッファ数を見積もって[4]バッファ用のスペース (バッファスロット) を確保する。初期レイアウトでは各セルの間にバッファスロットを入れて等間隔に、かつ総配線長が短くなるようにセルを配置する。
>
> **バッファのレイアウト**　2 回目以降は新たに挿入されたバッファのみをレイアウトの対象とする。バッファスロットに入っているバッファ[5]と空きバッファスロットに対して、ペア交換法により、新たに挿入されたバッファを総配線長が短くなるような位置に配置する。

レイアウト結果から得られた配線長をもとに計算した配線遅延は素子遅延に反映され、素子遅延に加えて配線遅延の情報まで含んだ nld を生成する。さらにその nld と 1 対 1 に対応する Verilog 記述も生成する[6]。

⑤ 遅延を均衡させるために遅延素子を挿入する。挿入方法については次節で説明する。挿入された遅延バッファを含めた回路量から面積を求める (④で用いる)。

⑥ 生成された Verilog 記述を用いたタイミングシミュレーションにより、動作確認、達成可能な動作周波数の測定、データ依存を考慮した遅延測定を行う。

5.3.2　遅延バッファを挿入するアルゴリズム

遅延バッファを挿入するアルゴリズムは以下を満たすようなものであることが望ましい。

- 余分な遅延バッファを入れないこと。

  > 遅延バッファを挿入するアルゴリズムとしては、目標とする遅延差を与え、それを満たすように遅延バッファを入れていく方法と、許される遅延バッファ数の範囲で遅延差を短縮するように遅延バッファを入れていく方法が考えられるが、5.3.1 節のはじめに書いた理由からここでは前者をとる。

- 全ての素子 1 個 1 個について[7]delay balancing を施すこと。

[4] 経験的には対象となる回路の回路量の 3〜4 倍のバッファが入る。

[5] よって①の最適化時に制約違反を解決するために挿入されたバッファは対象外になる。

[6] nld と Verilog のインタフェース、レイアウト方法の詳細は Appendix B, C 参照。

[7] 3.5.1 節より。

- 遅延バッファの挿入の効果や遅延差が短縮していく傾向がわかるように、また、目標を達成したり効果が現れなくなったと判断し次第、終了できるように、挿入した遅延バッファの数に対してステージの遅延差の最大値が単調減少となること。

これらをできる限り満たすアルゴリズムとして次のようなものを考えた。これは前節の④と⑤のイタレーション部分に相当する。5.3のような回路を例にステージの遅延を均衡させる手順を説明する。

step 0 1 回目のレイアウトが済んでいて、素子遅延には配線遅延が含まれていることを前提とする。

step 1 最も遅延差の大きな出力端子を見つける。そこの遅延差が目標の遅延差を下回っていたら、目標が達成されたとして step 5 の処理を経て終了とする。

例えば図 5.3で最も遅延差の大きな出力端子が G であったとする。

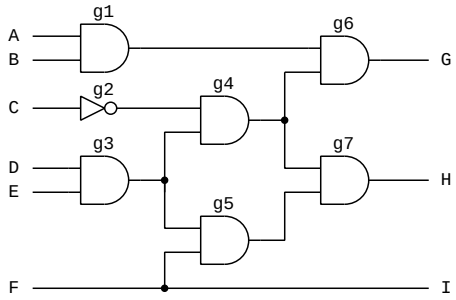

図 5.3: 対象となる回路

step 2 その出力端子につながる全てのパスの集合を抽出する。

出力端子 G につながる全てのパスの集合は図 5.4の様になる。

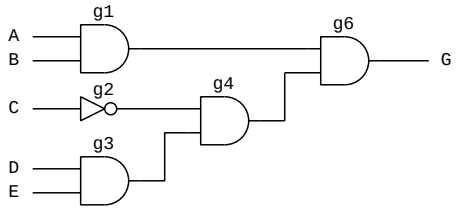

図 5.4: G につながるパス

step 3 step 2 で得たパスの中で最も遅延の小さなパスを選ぶ。このパスの中にバッファを入れることで step 1 で得た出力端子における遅延差が短縮できる。このパスに沿って出力側から順に[8]素子を探索し、各素子の入力側で[9]遅延差を求め (第 3 章参照)、遅延バッファを挿入すべきかどうかを判断する。バッファ挿入の条件を満たす場所が見つからないまま全ての素子を探索し終えたら、これ以上遅延バッファを入れても遅延差を短縮することができないとして step 5 へ。バッファスロットを使い切ってしまい、バッファを挿入できない場合も同様に step 5 へ。

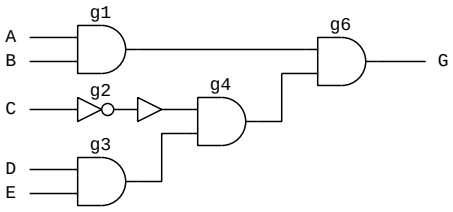

図 5.5: 遅延バッファを挿入

図 5.4の 5 つのパスの中で最も遅延の小さなパスが C — g2 — g4 — g6 — G だったとすると、このパスがバッファ挿入の対象となる。このパスに含まれる 3 つの素子を出力側から g6, g4, g2 の順に探索し、バッファを入れるべき場所かどうかを判定する。g2 と g4 の間にバッファを入れるべきと判断された場合、図 5.5のように遅延バッファが入るが、その結果、今まで g4 をドライブしていた g2 がバッファをドライブするようになるため g2 の遅延が変化する。すると g2 を包含する他のパスの遅延も変化する。よって配線遅延以外にも、g2 の遅延情報を更新する必要がある。

step 4 遅延バッファの挿入によって影響を受けた素子の遅延情報を更新し、新たに挿入されたバッファをレイアウトして step 1 へ。つまり遅延バッファを 1 個挿入する度に step 1 から再開する。

こうすることで常にその時点で最も遅延のばらつきの大きな出力端子の遅延 (= ステージの遅延差の最大値[10]) を均衡させる方向で遅延バッファが挿入される。これにより、挿入した遅延バッファ数に対して遅延差は単調減少になる。

[8] 出力に近い素子ほど他のパスと共有される確率が小さいため、バッファ挿入の副作用 (挿入されたバッファを包含する他のパスの遅延の増加が原因で遅延差が増えてしまうなど) が小さく、バッファ挿入位置としてより適していると考えられる。実際、入力側から順に素子を探索した場合と比較して良い結果が得られた。

[9] 3.3.1 節より。

[10] 見積もり上、遅延差が最も大きくなるのは出力端子であることを 3.3.1 節で示した。

step 5 この時点では全ての出力端子における遅延差は均衡しているが、遅延の絶対値は均衡していない。各出力端子の遅延状況をみて、遅延の小さな端子の遅延を遅延の大きな端子に合わせるように出力端子に遅延バッファを挿入する。

step 6 目標の遅延差を達成したが、空きバッファスロットが多い場合、余分な面積が使われていることになり、コスト、配線遅延を無駄に増加させている。よってこの結果からより正確なバッファ数の見積もりを行い、図 5.2の設計手順の④初期レイアウトからやり直す。

また、バッファ挿入の条件を満たす場所がなかったり、バッファスロットを使い切ってしまって、さらに、達成された遅延差が目標から大きく離れていた場合も、より正確なバッファ数の見積もりを行い、図 5.2の設計手順の④ 初期レイアウトからやり直す。

限界まで多数のバッファスロットを確保したがそれを使い切り、かつ、目標の遅延差を達成できなかった場合は、ウェーブパイプラインの導入は (コストに見合ったパフォーマンスを得る手段としては) 不適切だったと判断する。

■

以上をウェーブ動作する全てのステージについて行う。ただし、前述のように最初に delay balancing の対象となるステージだけは達成すべき遅延差の目標が異なる。MUP の場合はキャッシュのサイクルタイムを目標にすることになる。他のステージはこのステージの遅延差を目標に delay balancing を行う。

上の step 3 及び step 5 で、遅延バッファを挿入するかどうかを決定する判定基準が幾つか考えられる。ここでは二通り考え、図 5.6を使って説明する。ともに、2 入力以上の素子のみを対象とする。従ってバッファやインバータはバッファ挿入の対象外とする。(a) は遅延バッファ挿入前を表す。

[**戦略 A**]　最大遅延が大きくなると必要な遅延バッファ数が増えるため、最大遅延は大きくならない方が良い。最大遅延が大きくならないように遅延バッファを挿入することで、バッファ数 (面積や配線遅延にも影響してくる) を抑えることができる。図 5.6(b) でバッファをもう一つ入れると (c) のように最大遅延が増加するので、この基準ではこれ以上挿入しない。

[**戦略 B**]　最大遅延が大きくなることを許して少しでも遅延差を小さくしようとするもの。必要な遅延バッファは増えるが遅延差は小さくできる。この基準では (c) のように最大遅延の増加は無視してとにかく遅延の差を縮めようとする。

次に、step 3 で出力側から順に素子をたどる理由を説明する。ステージのある出力端子 o1 の遅延差は目標の値にまで短縮できるが、別の出力端子 o2 の遅延差は目標の値にまで短縮できない場合を考える。入力側から順にたどった場合、o1 の遅延差は目標の値より

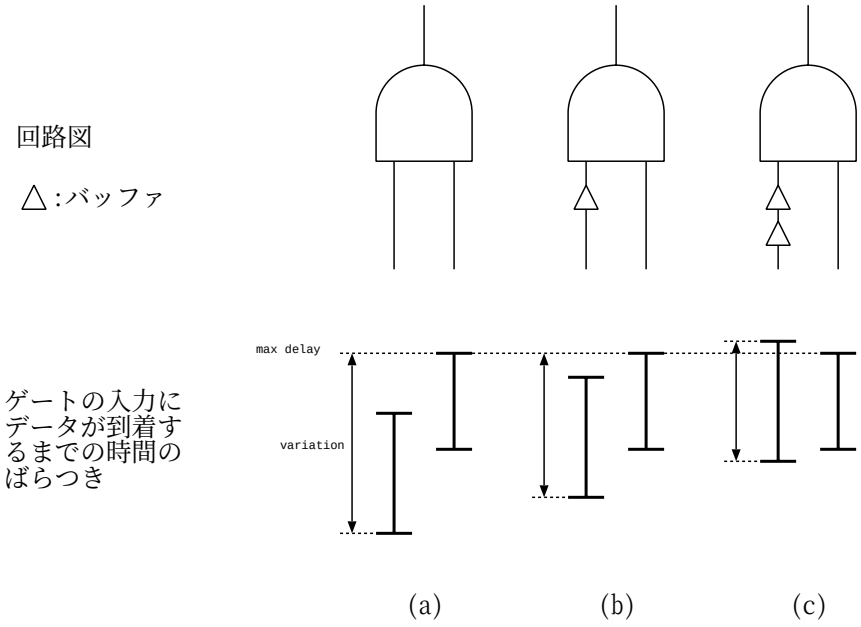

図 5.6: 遅延バッファを入れる戦略

も小さくなる。しかし o2 の遅延差は目標の値よりも小さくできず、結局これがこの回路の動作可能なサイクルタイムを決めてしまう。つまり o1 の遅延差を必要以上に短縮してしまい、そのために必要以上のバッファが使用されたことになる。このことは、回路の動作周波数が未知で、可能な限り遅延差の短縮を行うようにした場合に特に問題になる。これを避けるには出力側から順に素子をたどらなければならない。

最後に、分岐点の後に直接バッファが入るような場所ではバッファ数を減らせる場合がある。図 5.7左のような状況では buffer chain を用いることで右のようにバッファ数を 6 個から 3 個に減らすことができる。

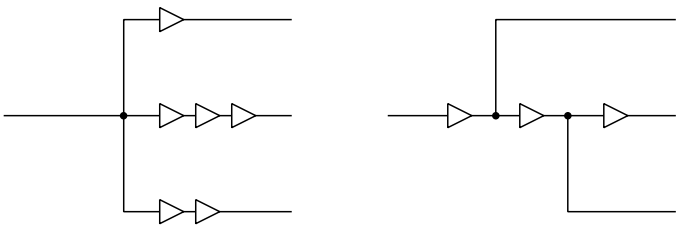

図 5.7: buffer chain

ただし接続状況が変化すれば素子がドライブする付加容量が変化するので、その素子の遅延に変化があることは考慮しなければならない。

5.3.3 加算器への適用 (配線遅延を考慮しない場合)

プロセッサ全体の delay balancing をする前段階として、ウェーブパイプライン版 MUP に搭載する加算器に上述の delay balancing の方法を適用してみる。

ここでは配線遅延を (よってレイアウトも) 考慮しないで delay balancing を試みる。配線遅延が支配的でない大きなプロセステクノロジ上を想定した場合の評価や、遅延バッファ挿入のアルゴリズムのみの純粋な評価には適している。

PARTHENON 附属のサンプルのセルライブラリを用いている。

使用する加算器は 32bit の BLC 加算器[11]で、遅延バッファ挿入前の各出力端子の遅延状況は図 5.8のようになっている。横軸は出力端子名 (0〜31)、縦軸はそれぞれの出力端子における最大遅延と最小遅延を 3.3.1 節の方法で見積もった値である。

以下、前節で示した戦略 A を採用した場合と戦略 B を採用した場合の両方の結果を示す。どちらも遅延差の目標を 0 (バッファを挿入すべき場所がなくなるまで続ける) としてアルゴリズムを動かした。

最大遅延が大きくならないように遅延バッファを挿入

前節で示した戦略 A を採用したものである。図 5.9左は、挿入したバッファ数 (横軸) と、それに応じて遅延差の最大値 (縦軸) が短縮する様子を表したものである。挿入した遅延バッファ数に対して遅延差が単調減少になっていることがわかる。この例ではバッファを 540 個挿入した時点で終了した (バッファを挿入すべき場所がなくなった)。換算ゲート数 (グラフ中に gates で示した数値) にして、もともとの回路のゲート数にほぼ等しい数のバッファが入ることになる。

図 5.10左は、遅延バッファ挿入後の各出力端子の遅延状況を図 5.8と同じ形で表したものである。ただし出力端子の遅延の絶対値を均衡させる前の状態である。図 5.8に比べて遅延差が短縮しており、最大遅延は増加していないことがわかる。

最大遅延が大きくなるのを許して遅延バッファを挿入

前節で示した戦略 B を採用したものである。図 5.9右は、挿入したバッファ数 (横軸) と、それに応じて遅延差の最大値 (縦軸) が短縮する様子を表したものである。挿入した遅延バッファ数に対して遅延差が正確には単調減少になっていない。

> グラフ中で遅延差が増加している部分は、図 5.6(c) の様に、バッファを挿入されたパスの遅延が大きくなった結果、そのパスにつながる別の出力端子 (この時点で delay balancing の対象となっている出力端子とは別の出力端子) における遅延差が大きくなってしまったことを表す。しかし直ちにその出力端子に対して delay balancing が

[11] Appendix A 参照

行われるので、この遅延差の増加は一時的なものである。よって遅延差が増加する部分はグラフでは突起状になる。

この例ではバッファを 805 個挿入した時点で終了した (バッファを挿入すべき場所がなくなった)。換算ゲート数 (グラフ中に gates で示した数値) にして、もともとの回路のゲート数よりもかなり多数のバッファが入ることになる。

図 5.10右は、遅延バッファ挿入後の各出力端子の遅延状況を図 5.8と同じ形で表したものである。図 5.10左に比べてさらに遅延差が短縮しているが、最大遅延が増加していることがわかる。この後で出力端子の遅延の絶対値を均衡させなければならない (5.3.2 節の step 5) が、最大遅延が大きくなった分、そこで必要になる遅延バッファの数が大きくなる。

5.3.4 加算器への適用 (配線遅延を考慮した場合)

前節では最大遅延の増加を許す戦略 B の方が良い結果を得られた。しかし配線遅延を考慮した場合、最大遅延が増加すると必要なバッファ数だけでなく配線遅延も増加するため、コスト的には大きな損失となる。よって戦略 A の方が現実的と言える。ここでは戦略 A のみを適用する。

図 5.11は前節と同じ加算器を用いて配線遅延の評価やレイアウトを含めて delay balancing を実行したものである。初期レイアウトでは前節の結果を参考にバッファスロットを 800 個確保した。

図 5.11(a) は比較対象として、全てのバッファを挿入し終えるまでバッファの配置改善をせず、全てのバッファを挿入した後に、挿入された全てのバッファを対象に配置改善を行ったものである。バッファをちょうど 600 個挿入した時点でバッファを挿入できる場所がなくなって終了した (バッファ挿入による遅延均衡の限界)。バッファスロット数を変えたり配線抵抗/配線容量を求めるパラメータ[12]を変えた場合[13]でも、限界までバッファを挿入したときの遅延差は 170ps 程度であることから、ここで使用したライブラリとアルゴリズムでは面積・配線遅延にかかわらず遅延差は 170ps あたりが限界と考えられる。

図 5.11(b) は第 5 章で提案したレイアウトも含めた手法を適用したものである。つまり、バッファを 1 個入れる度にその配置を最適なものに改善したものである。バッファを 770 個挿入した時点でバッファを入れるべき場所がなくなった。十分なバッファスロットを確保して限界までバッファを挿入することができれば前述の理由により 170ps 程度まで遅延を均衡させることができると予想される。

[12] Appendix B 参照。

[13] バッファスロット数を変えれば面積が増え、配線遅延が増加する。配線抵抗/配線容量を求めるパラメータを変えれば配線遅延が変化する。つまり、どちらも配線遅延の影響を変化させたことに相当する。

両者の結果を、バッファ挿入前のデータと合わせて表 5.1に示した。図 5.11と表 5.1のデータはともに実配線長で遅延を評価したものである。また、バッファ挿入前のデータは全て初期レイアウト後のものである。

表 5.1: delay balancing の効果

測定対象		遅延差 [ps]	総配線長 [grid]
バッファ挿入前	バッファスロット無し	722(最大遅延 797)	66403
	バッファスロット有り	910	130133
図 5.11(a)	バッファ再配置前	176	274962
(バッファ600 個)	バッファ再配置後	417	204563
図 5.11(b)	600 個挿入時点	307	171983
(バッファ770 個)	終了時点	219	202267

$(1\ \mathrm{grid} = 0.25\mu m)$

戦略 A の中で、バッファを挿入した場合の遅延状況を見積もらなければならないが、レイアウトの結果バッファがどこに配置されるか知ることはできないので、バッファに付加される配線遅延は仮想配線長[14]から見積もり、これをもとにバッファを挿入すべきかの判定を行う。そのためこの判定に誤りがあることがあり、グラフ中に突起部分が生じる。

表 5.1より、バッファ再配置前の (a) は (b) よりも遅延差が小さくできている。(b) では常に配線長が最短になる場所にバッファが配置されていくため、配線長が非常に短くなる位置に配置できた場合、バッファの遅延値が十分な値を持たなくなってしまうことがこの原因と考えられる。しかし、バッファ再配置前の (a) はバッファのレイアウトを全く考慮していないため総配線長が非常い大きい。総配線長が大きいと配線領域を圧迫し、配線が困難になり、場合によっては配線が不可能になる。これを避けるためにバッファを再配置することによって総配線長の短縮を試みた (バッファ再配置後の (a)) が、その結果、遅延差、総配線長の両面において (b) よりも大きく劣ってしまった。(b) は (a) よりもバッファ数が 170 個多いにもかかわらずバッファ再配置後の (a) よりも配線長が小さくなっている。これにより、(b)、つまり第 5 章で提案した手法が遅延差、総配線長の両面で良い結果を与えるものであることがわかる。

図 5.11(b) ではバッファ挿入によって遅延差が $910/219 = 4.16$ より 1/4 以下に短縮しているが、ウェーブパイプラインを導入しない場合との比較評価を行う場合、比較対象とすべきなのは、図 5.11のグラフの左端の値ではなく、バッファスロットを確保しないで初期レイアウトを行った回路の最大遅延値である。これは表 5.1の最上段に示されている。

[14] Appendix C 参照

797/219 = 3.64 より、バッファスロットの確保からバッファの挿入、バッファの配置改善までを含めた一連の諸操作によって動作周波数が 3.64 倍になったことになる。よってこの回路のケースでは、ウェーブパイプラインの導入によって 3.64 倍の周波数で動作可能になったことになる。

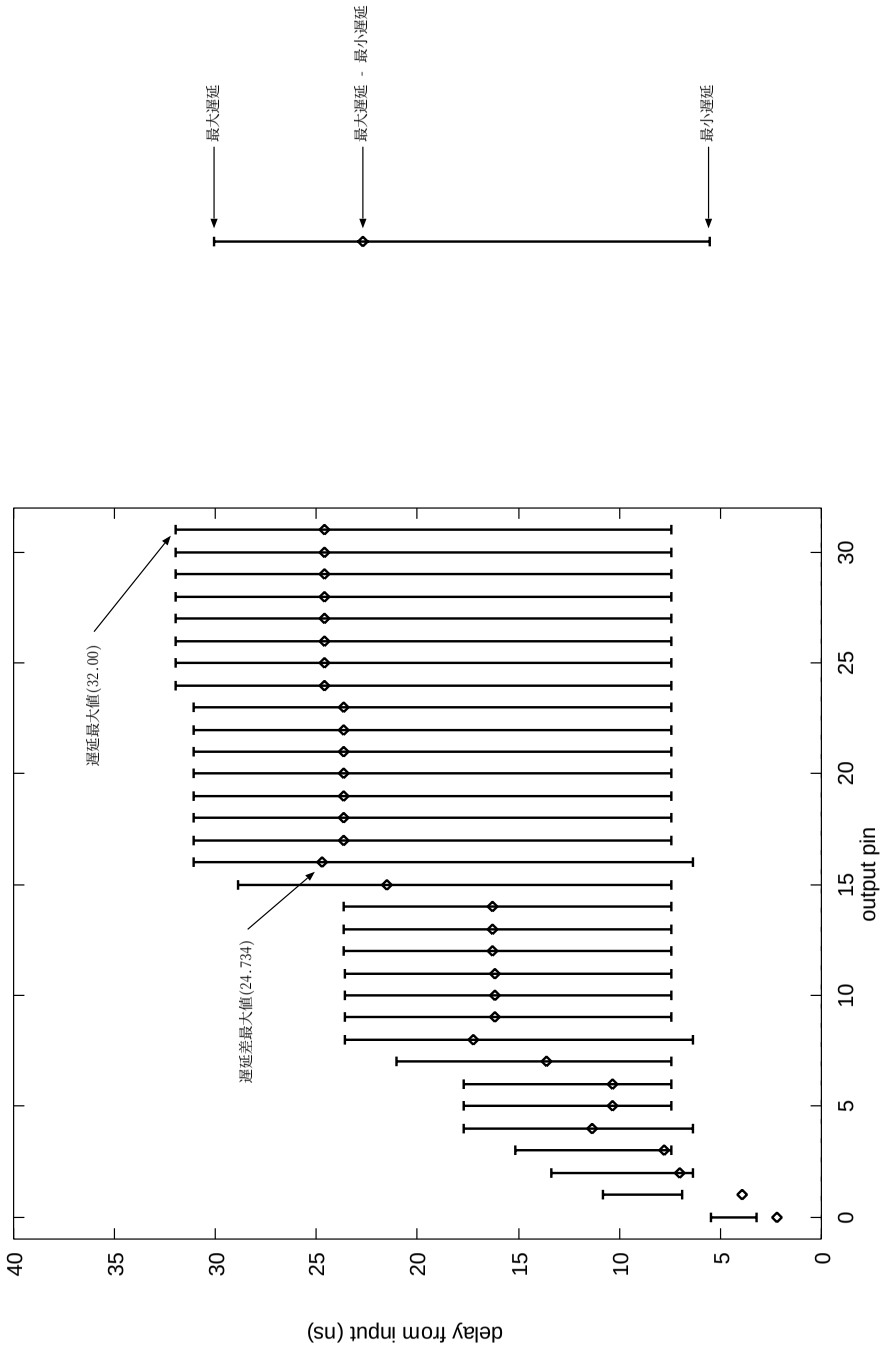

図 5.8: 遅延バッファ挿入前の出力端子の遅延状況

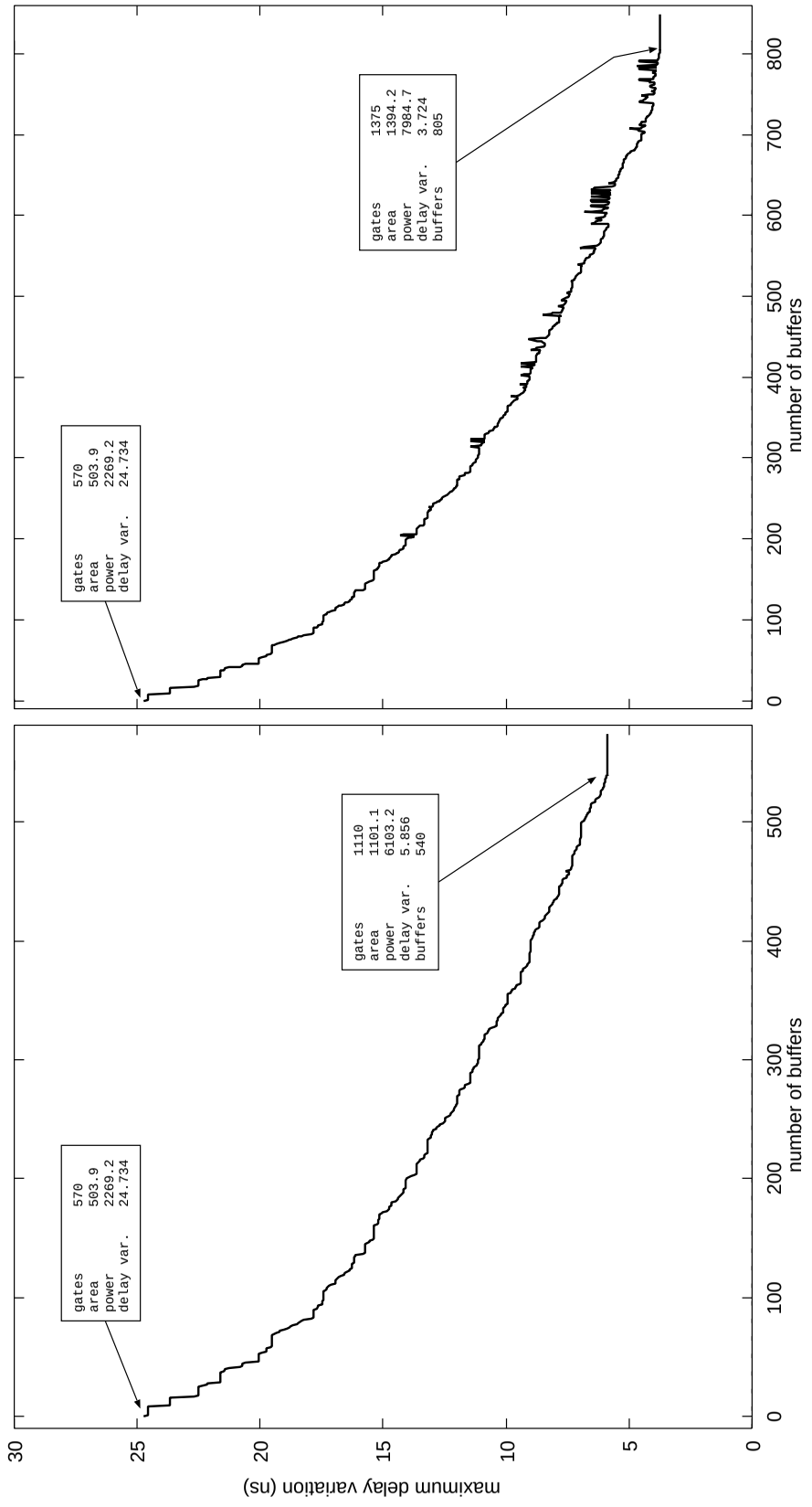

図 5.9: 遅延差が短縮する様子

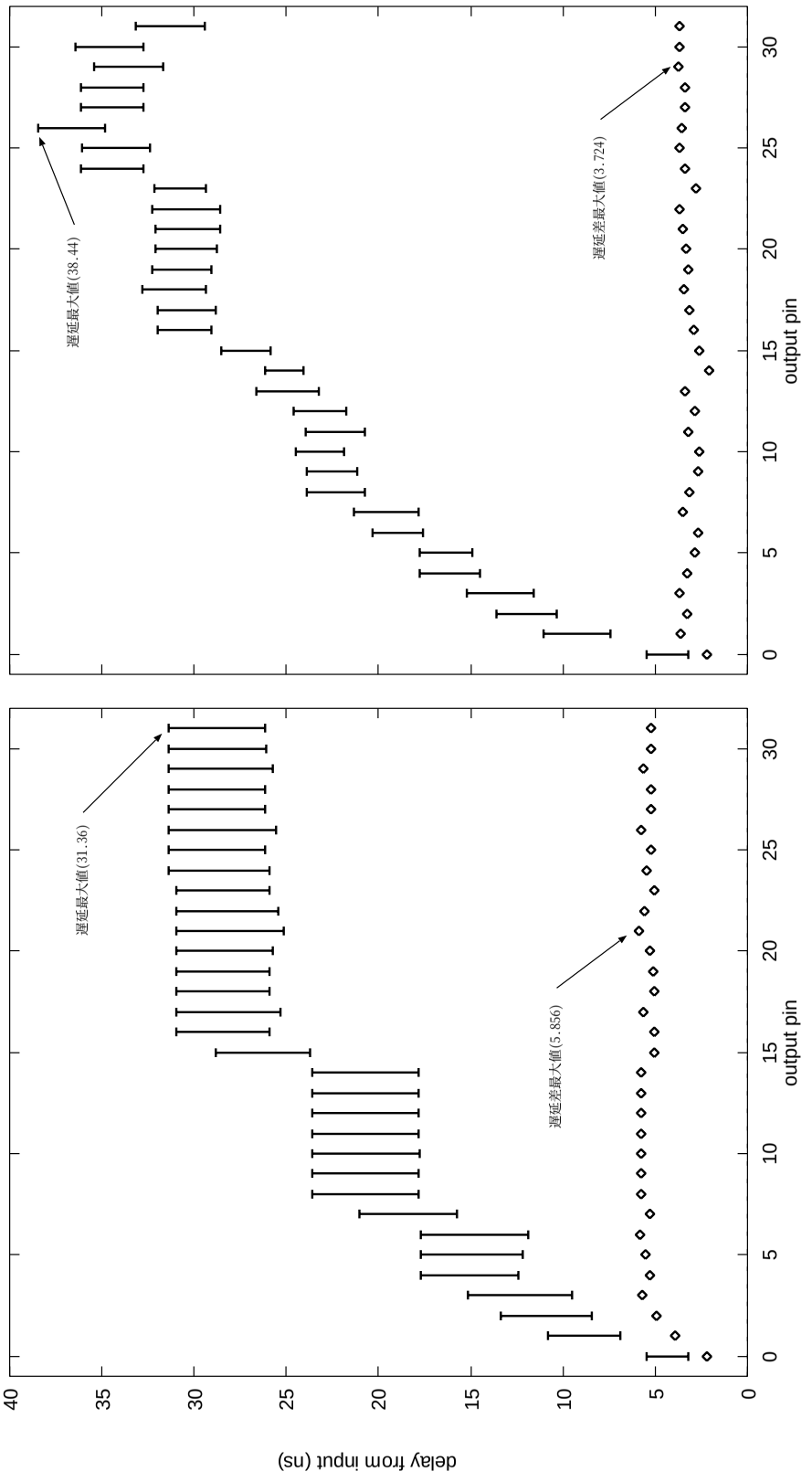

図 5.10: バッファ挿入後の出力端子の遅延状況

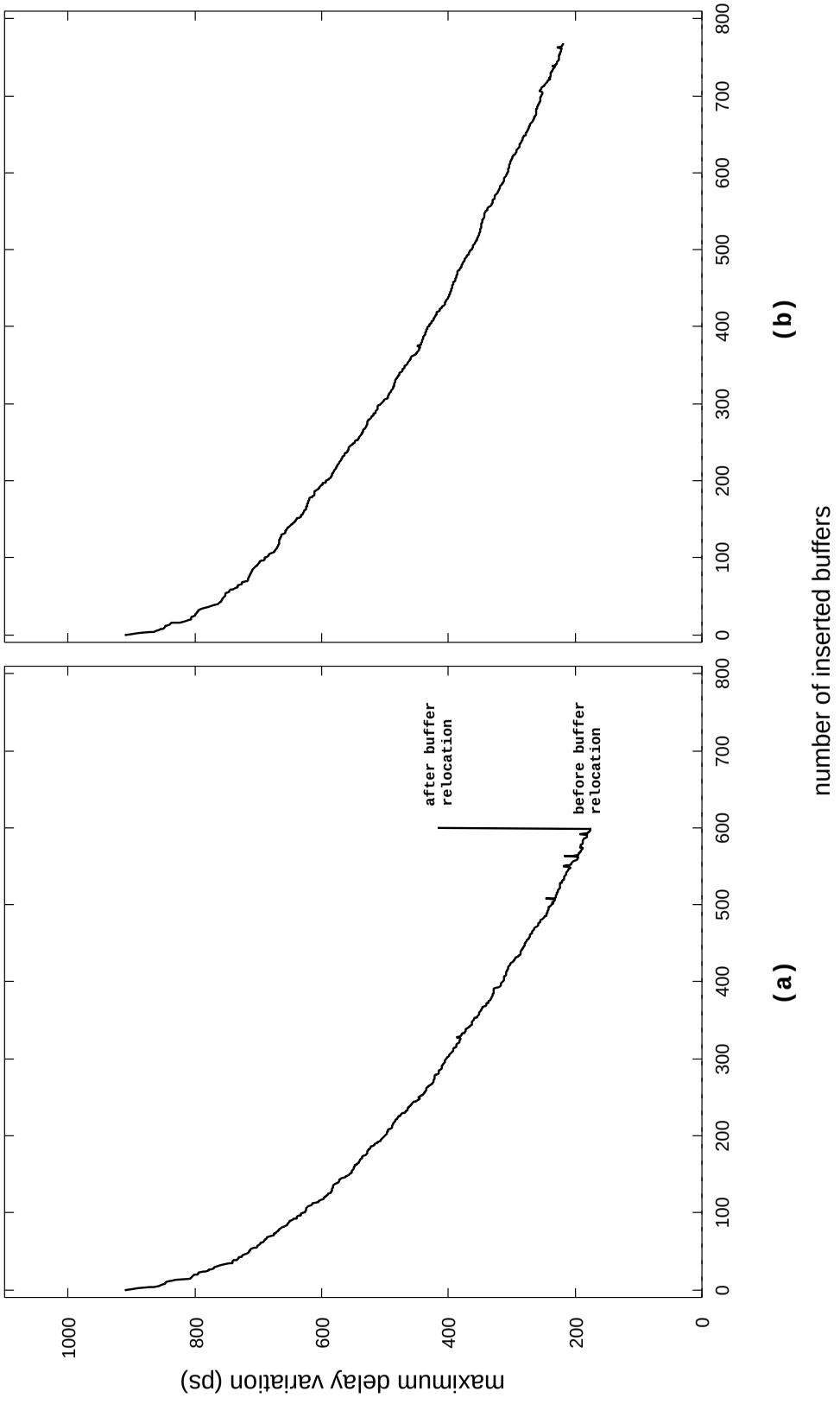

図 5.11: 遅延差が短縮する様子

第 6 章

コストとパフォーマンス

ある組み合せ回路に対して、それを通常動作 (非ウェーブ動作) させるかウェーブ動作させるかという判断は、ウェーブパイプライン化によって得られるパフォーマンス増が遅延バッファ挿入によるコスト増に見合うものであるかどうかにかかる。ウェーブ動作をさせる上でコストやパフォーマンスを左右する要素を明らかにすればウェーブパイプラインの導入を有利にするための設計を考えることができる。

ここでは、遅延の均衡化をバッファの挿入によって行う場合のコストとパフォーマンスについて考える。

6.1 パフォーマンスの評価

6.1.1 パラメータと計算法

パフォーマンスは達成可能な動作周波数で評価する。通常動作の場合とウェーブ動作の場合の両方の動作周波数を得るために必要なパラメータとして以下を用意する。

- T_{max}　バッファ挿入前の最大遅延
- ΔT　バッファ挿入後の遅延差の最大値
- $\Delta_{latch}, \Delta_{clock}$　ラッチ、クロックオーバーヘッド
- x　遅延バッファ挿入対象となる部分の面積増加率
- $\beta(x)$　遅延差短縮率[1]を x の関数として表したもの

$$\text{通常のパイプライン動作時の周期} \quad = \quad T_{\max} + \Delta_{latch} + \Delta_{clock} \tag{6.1}$$

[1]遅延バッファ挿入前の最大遅延に対する、遅延バッファ挿入後の遅延差の比の逆数。遅延バッファを挿入したことによって、そのステージが動作可能な周波数が何倍になったかを表す。

$$\text{ウェーブパイプライン動作時の周期} = \Delta T + \Delta_{latch} + 2\Delta_{clock} \tag{6.2}$$

$$\text{よって、} \tag{6.3}$$

$$\text{動作周波数向上比} = \frac{T_{max} + \Delta_{latch} + \Delta_{clock}}{\Delta T + \Delta_{latch} + 2\Delta_{clock}} \tag{6.4}$$

$$= \frac{T_{\max} + \Delta_{latch} + \Delta_{clock}}{\frac{1}{\beta(x)} T_{max} + \Delta_{latch} + 2\Delta_{clock}} \tag{6.5}$$

この動作周波数向上比をパフォーマンスの向上比とする。

6.1.2 加算器への適用

前章で使用したものと同じ加算器に対してパフォーマンス評価を適用する準備として図 6.1に、挿入したバッファ数 (横軸) と動作周波数向上比 (縦軸) の関係を示す。最大遅延が大きくならないようにバッファを挿入した場合が図 6.1左、最大遅延が大きくなるのを許してバッファを挿入した場合が図 6.1右である。これらはそれぞれ、図 5.9、図 5.10の縦軸を、バッファ挿入前の最大遅延に対する比の逆数に変換したものに相当する。すなわち縦軸は、バッファ挿入によって、対象となっている加算器が動作可能な周波数が何倍になったかを表す。

バッファ挿入の効果は、バッファを入れ尽くす限界点に近い程大きいことがわかる。これは特定の加算器を対象にしたものであり、この結果に一般性はないが、限界点に近い程効果が大きいという性質は一般性を持つと考えられる。

前述のように、バッファ挿入の対象となるのは組合せ論理とステージをスルーするパスである。そこで図 6.1右に、スルーするパスへ挿入される分のバッファを加えたものを図 6.2に示す。

この図では MUP の設計を参考に、ステージをスルーするパス全ての遅延を 1ns 増加させるためには 60 個のバッファを要すると仮定している。

さらに、(6.5) 式からパフォーマンスの向上比を求める準備として、図 6.2の横軸をバッファ挿入対象となる部分の面積増加率 (x)、縦軸を動作周波数向上率に変換したものを図 6.3に示す。これは前述の式中の$\beta(x)$ に相当する。

(6.5) 式に従ってパフォーマンス向上比を計算した結果はコストと合わせて最後に示す。

6.2 コストの評価

6.2.1 パラメータと計算法

ダイコストで評価する。以下の諸式は [5] による。コストを得るために必要なパラメータとして以下を用意する。

- ρ　欠陥率
- A_{die}　バッファ挿入前のダイ面積
- a　バッファ挿入対象となる部分がダイ全体に占める面積比
- x　バッファ挿入対象となる部分の面積増加率

$$\text{ダイコスト} = \frac{\text{ウェーハコスト}}{\text{ウェーハ当りダイ数}}\text{より、} \tag{6.6}$$

$$\text{ダイコスト増加率} = \frac{1}{\text{ウェーハ当りダイ数} \times \text{ダイ歩留まり増加率}} \tag{6.7}$$

$$\text{ウェーハ当りのダイ数} = \frac{\text{ウェーハ面積}}{A_{die}}\text{より、} \tag{6.8}$$

$$\text{ウェーハ当りのダイ数増加率} = \frac{1}{\text{ダイ面積増加率}} \tag{6.9}$$

$$\text{ダイ歩留まり} = Ne^{-dA_{die}}\text{より、} \tag{6.10}$$

$$\text{ダイ面積増加率を}\alpha\text{とすると、} \tag{6.11}$$

$$\text{ダイ歩留まり増加率} = e^{-\rho A_{die}(\alpha-1)} \tag{6.12}$$

$$\text{ダイコスト増加率} = \alpha e^{\rho A_{die}(\alpha-1)} \tag{6.13}$$

$$\alpha = ax + (1-a)\text{だから、} \tag{6.14}$$

$$\text{ダイコスト増加率} = (ax+1-a)e^{\rho A_{die}a(x-1)} \tag{6.15}$$

このダイコスト増加率をコストの増加率とする。

6.2.2　加算器への適用

図 6.4と図 6.5に上式に従ったコストのグラフを示す。図 6.4は$\rho = 1.0$、$A_{die} = 1$ として、バッファ挿入対象となる部分がダイ全体に占める割合を変えてプロットしたものである。図 6.5はバッファ挿入対象となる部分がダイ全体に占める割合を 20%、$A_{die} = 1$ として、欠陥率を変えてプロットしたものである。

6.3　ウェーブパイプラインの導入が適切と判断する条件

図 6.1より、挿入するバッファが多い程その効果が大きい。組合せ論理部分をいじらずにただスルーするパスにバッファを入れるだけでも若干の高速化は可能だが、それではとてもコストに見合うだけのパフォーマンスは得られないことは明らかである。

ダイコストは、製造プロセスや、バッファ挿入の対象となる部分が占める面積比に依存する。歩留まりがチップ面積や欠陥率に依存するからである。製造技術が発達して歩留ま

りが向上すればコストはより低くなり、コストに見合うパフォーマンス向上の実現は易しくなるだろう。

一方、バッファ挿入の対象となる部分が占める面積比は回路設計のレベルで縮小できる。前述のように、バッファ挿入の対象となるのは組合せ論理部分とスルーするパスである。よってスルーするパスが少なくなるような工夫をすることでコストを抑えることができる。

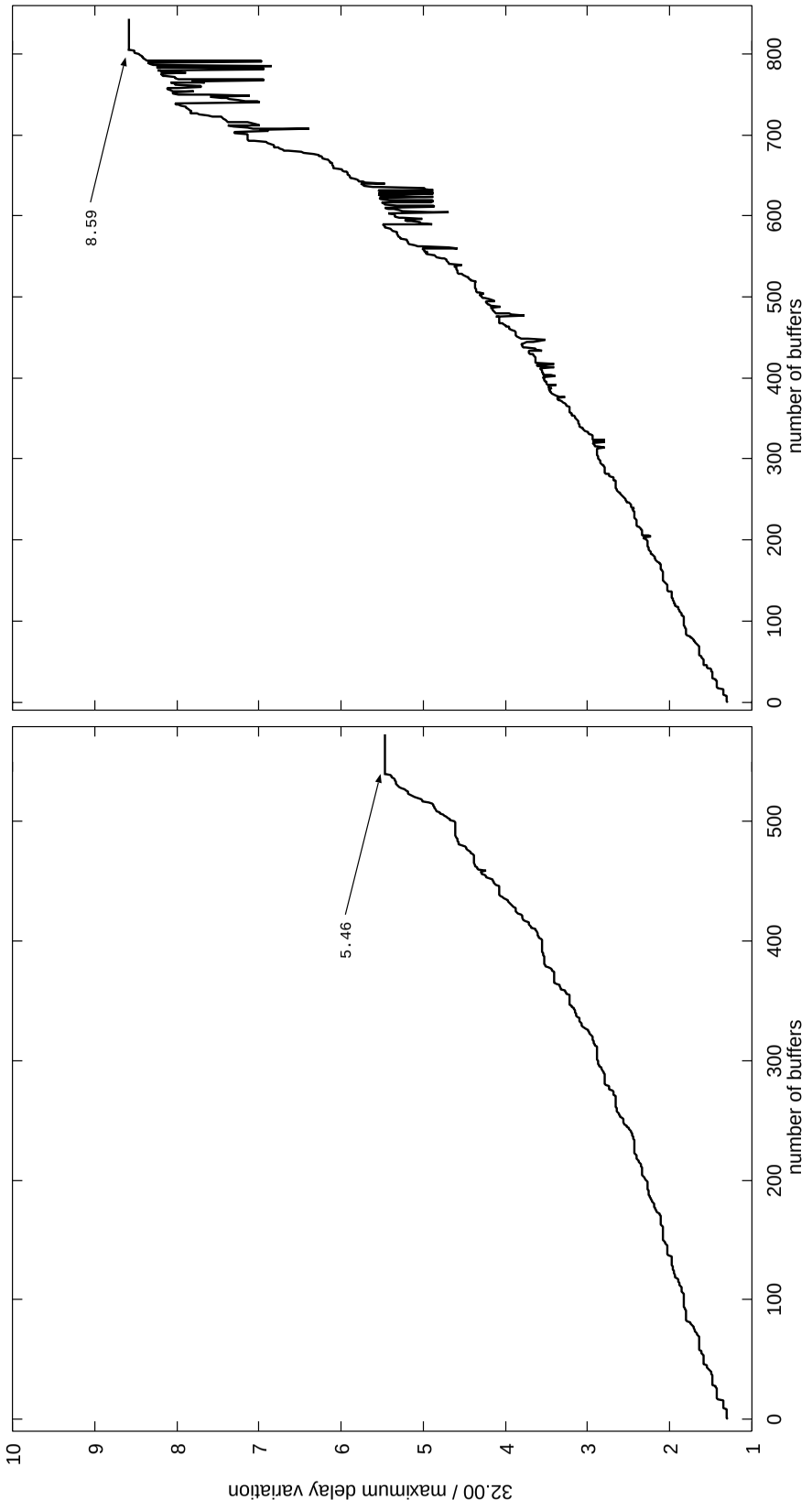

図 6.1: バッファ数と動作周波数向上

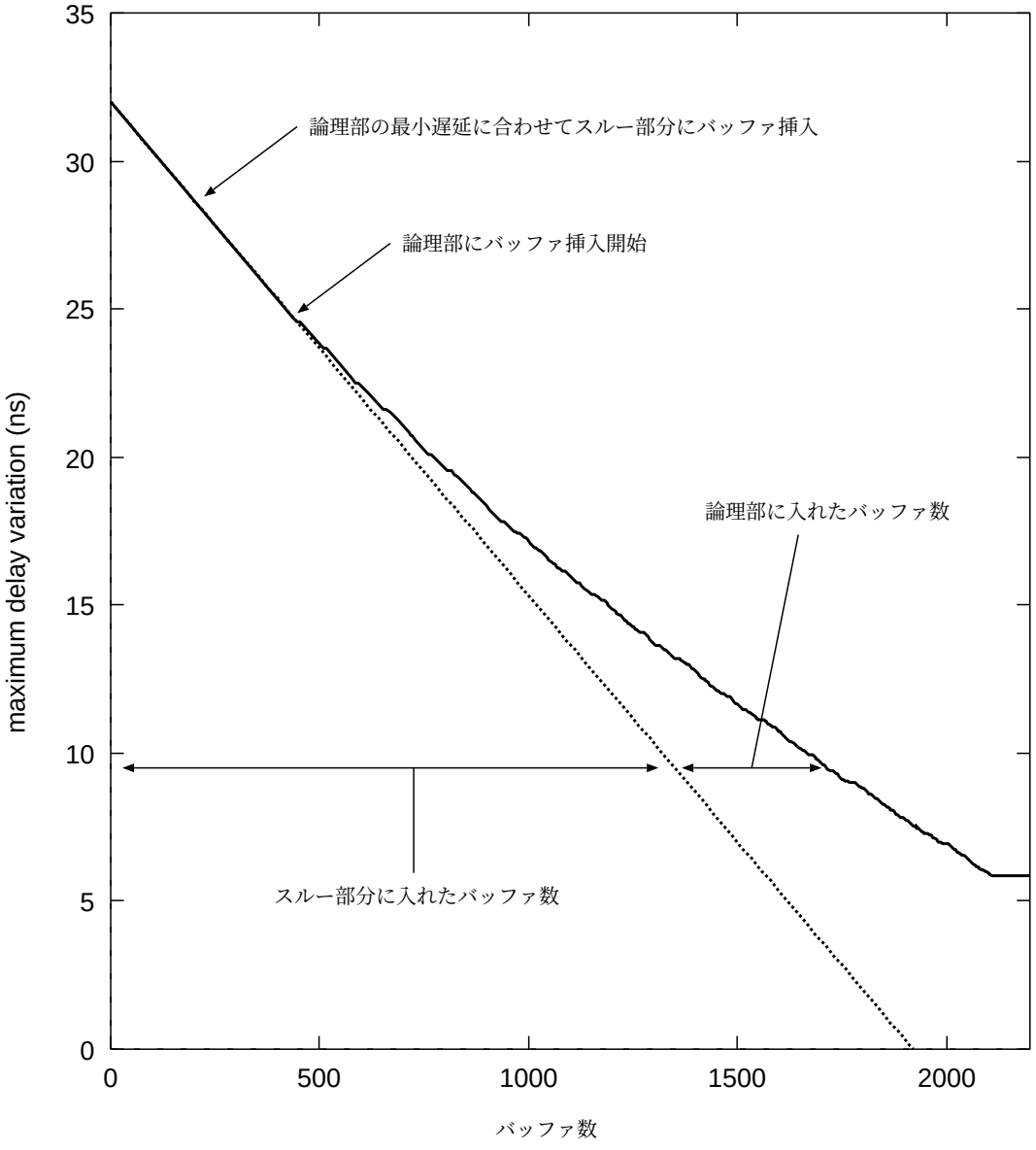

図 6.2: バッファ数と遅延差 (スルーするパスを考慮)

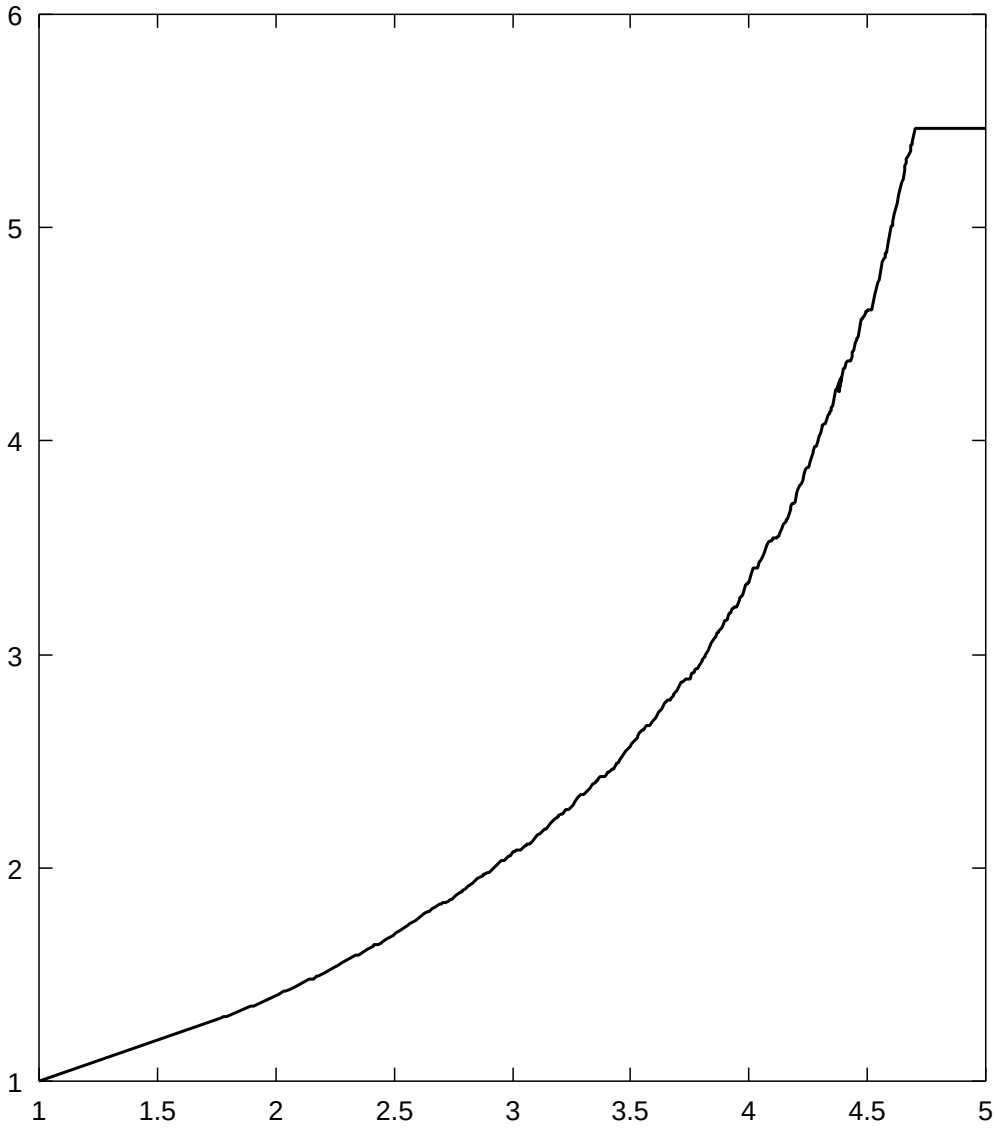

図 6.3: $\beta(x)$

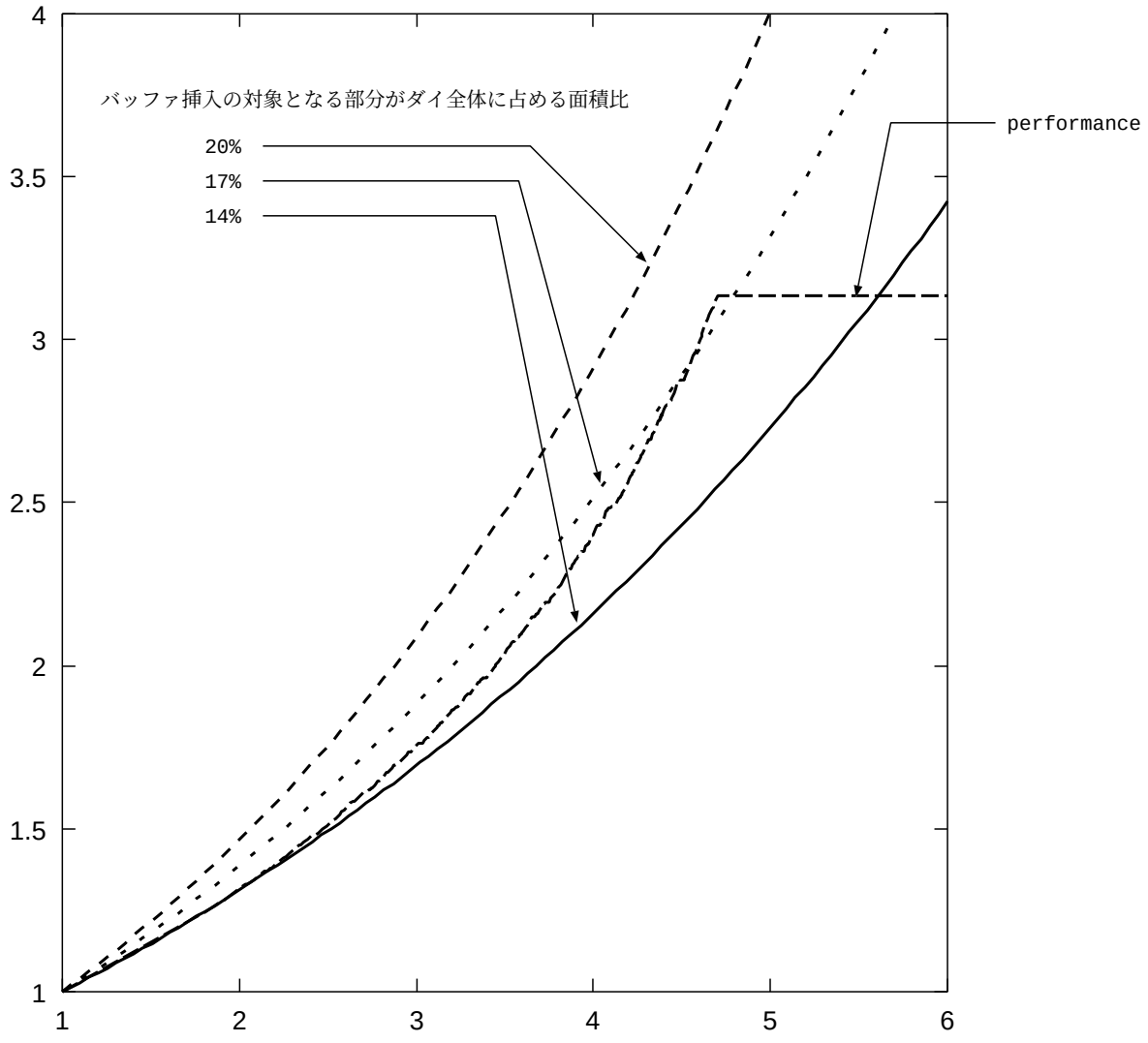

図 6.4: 面積とコスト・パフォーマンス

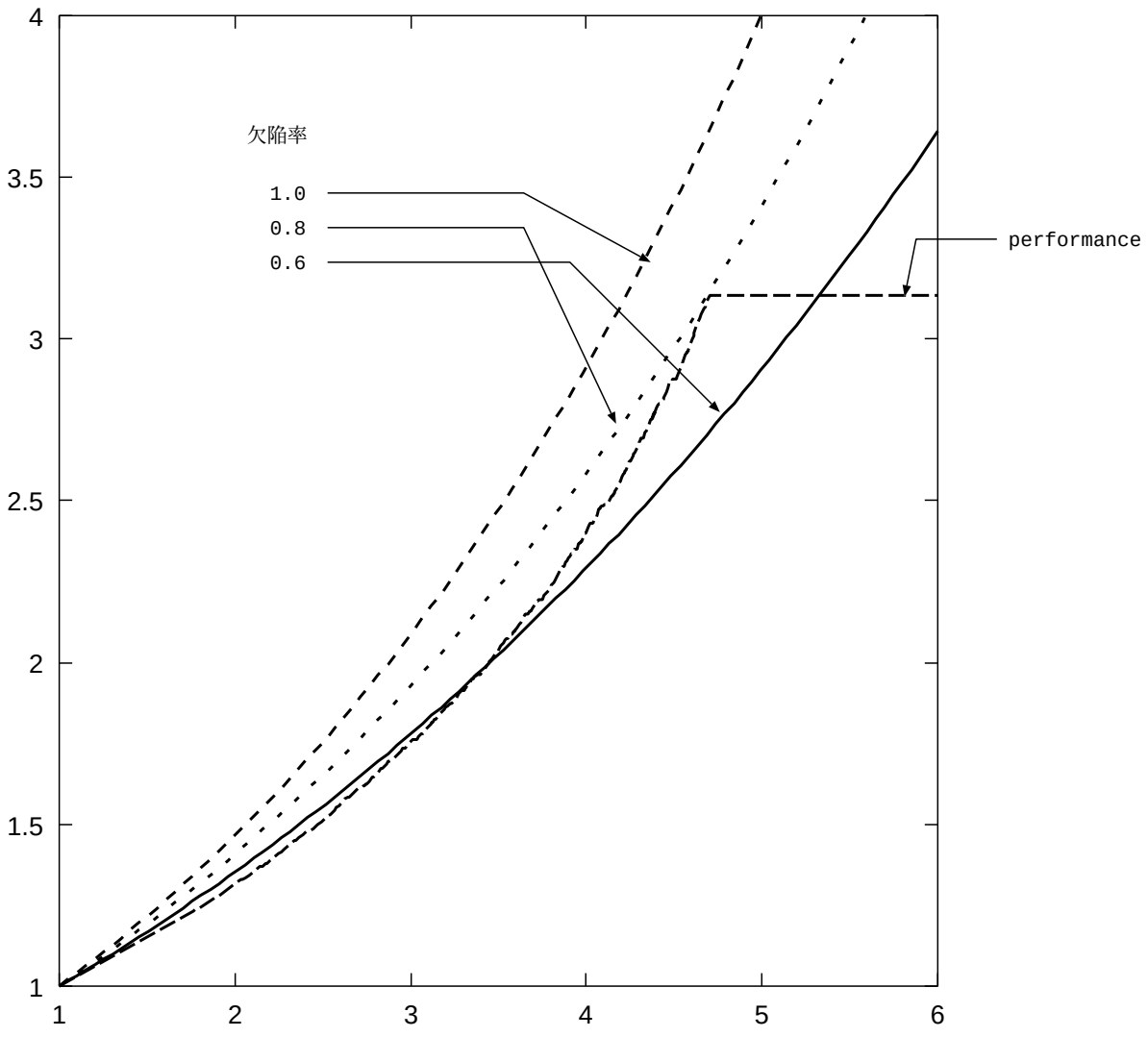

図 6.5: 欠陥率とコスト・パフォーマンス

第 7 章

MUP の設計

ウェーブパイプライン版 MUP の設計として、第 4 章で決めたステージ構成を採用し、各ステージ内部の回路設計に第 6 章の結果を反映させ、第 3 章で決めた方法で遅延を評価し第 5 章で決めた遅延均衡法を適用する。

7.1 コストを抑えるための工夫

第 6 章より、回路構成を設計する段階で遅延バッファを減らしてコストを抑えるには、ウェーブ動作するステージをスルーするパスを減らすことである。それには次の方法が考えられる。

- デコーダを分散させる。

  通常 ID ステージで一括デコードされるような制御情報を、その制御情報を使用する直前にデコードする。または、エンコードした形で情報を流し、使用する直前でデコードする。

  - EX ステージで使う制御信号のうち、ALU 制御と分岐先制御の一部は ID ステージからエンコードされた形で EX ステージまで送る。これによって ID をスルーするパスが 5 本程度減らせる。
  - WB ステージで使う制御信号のうち、汎用レジスタの書き込みポートを指定する信号はデコードせずに DF9 ステージまで送り、DF9 ステージ (WB の直前) でデコードする。これによって ID、EX の各ステージ (共にウェーブ動作する) をスルーするパスを 27 本減らせる。

- 一時レジスタの使用。

  ウェーブ動作するステージを通過するだけの情報は、そのステージの直前で一時レジスタに保存する。その後、必要になる直前にその情報をレジスタから読むことで、ウェーブ動作するステージをバイパスさせる。一時レジスタはスレッド数

個分用意する。スレッド数個分用意されたレジスタに関してはスレッド間でタイミングを調整する必要がないためウェーブ動作の支障にはならない。この方法はそれを利用したものである。

- これを外部例外情報のデータパスに適用することで、ID をスルーするパスを 32 本減らせる。(Appendix D の 図 8.7中の CRE_t 参照)
- これを例外 PC 用のデータパスに適用することで、ID と EX それぞれをスルーするパスを 32 本ずつ減らせる。(同じく EPC_t 参照)
- これを汎用レジスタの書き込みポート指定信号のデータパスに適用することで、ID と EX それぞれをスルーするパスを 5 本ずつ減らせる。(同じく rd_t 参照)
- WB ステージの制御信号に適用することで、EX ステージをスルーするパスを 4 本減らせる。(同じく WB_t 参照)

以上により、ID ステージをスルーするパスを 69 本、EX ステージをスルーするパスを 73 本減らすことができる。遅延を均衡させた結果、ID ステージの最小遅延がバッファ n 個分、EX ステージの最小遅延がバッファ m 個分になったとすると、これによって $69n+73m$ 個のバッファが節約されたことになる。

一時レジスタを使う場合はその分回路量が増加するが、デコーダを分散させた場合、そのデコーダの一部がウェーブ動作するステージに加わっても、いずれそこには遅延バッファが入ることになるので、デコーダが入ることによる回路量の増加はない。

上のそれぞれの方法を適用した MUP の構成については Appendix D に示した。

この時点でステージ構成からステージ内部の回路までが定まった。後はウェーブ動作するステージの遅延均衡を行い、その結果にもとづいて今後サイクルタイム、スレッド数を決めることになる。

7.2 ステージの遅延均衡

7.2.1 達成可能な動作周波数

前述のように、遅延均衡の最初のターゲットになるステージは、最も遅延が均衡しそうにないステージである。MUP の場合これは EX ステージである。

このステージの delay balancing の目標となる遅延差は、(キャッシュのサイクルタイム – クロックオーバーヘッド – ラッチオーバーヘッド) である。

具体的には以下の様に設計を行う。

1. プロセスルールは 0.25μ を想定する。トランジスタの物理特性を HSpice シミュレーションで求め、それをもとにセルライブラリを作成する。

2. 第 3 章〜第 5 章の手法を適用し、遅延バッファの挿入によってステージの遅延を均衡させる。

3. Verilog シミュレーションにより、バッファ挿入によって達成された動作周波数を測定する。

7.2.2 コスト

レジスタの多くがスレッド数個分に多重化されている上、一時レジスタ (これもスレッド数個分用意される) を使うためレジスタの回路量が非常に多くなる。ウェーブ動作しないキャッシュや、記憶要素であるレジスタは delay balancing の対象にならない。よって、遅延バッファが挿入される部分が占める面積は小さくなる。遅延バッファが挿入される部分が占める面積が非常に小さいと、限界までバッファを入れても[1]コストにはあまり影響しないことになる。このことは図 6.4からもうかがえる。

[1]遅延バッファの挿入の効果は限界点に近い程大きい。(6.1 節より)

第 8 章

まとめ

本論文では 7 章に渡って、マルチスレッド型プロセッサとウェーブパイプラインについてその特徴から設計法、評価方法までを議論してきた。

第 1 章では、マルチスレッド型プロセッサの特徴と、MUP の概要を説明した。第 2 章では、ウェーブパイプラインの特徴を説明し、それがマルチスレッド型プロセッサと相性が良いこと、MUP を高速化する手段として有効であることを示した。第 3 章では、ウェーブパイプラインを用いたプロセッサを設計するためのおおまかな手順を示し、その準備として素子/配線モデル、遅延を見積もる方法を提案した。第 4 章では、ウェーブパイプラインのためのステージ構成や各ステージを動作させるタイミングについて考え、レジスタの入力側にはラッチを置いた方が良いという結論を得た。それを MUP に適用してウェーブパイプライン版 MUP のステージ構成を決定した。第 5 章では、レイアウトや配線遅延まで考えてステージの遅延を均衡させる手順を示した。この中で、遅延バッファを挿入するアルゴリズムを二通り提案し、それぞれを MUP の搭載する 32bit 加算器に適用した。第 6 章では、回路をウェーブ動作させることによるパフォーマンス向上比と、バッファを挿入することによるコスト増を評価する方法を示し、それを前章で使用した加算器に適用した。その結果として、回路設計の段階でコスト・パフォーマンス比を上げるにはステージを通過するだけのパスを減らすことが重要であることを明らかにした。第 7 章では、第 4 章で決めたステージ構成を採用し、各ステージ内部の回路設計に第 6 章の結果を反映させたウェーブパイプライン版 MUP を設計した。

素子/配線モデルから遅延均衡の方法、バッファの挿入法など、各段階で様々な手法が考られ、ウェーブパイプライン動作版プロセッサの設計法がまとまっていない。これらの各段階毎にもっと詳細な調査が必要である。

ウェーブパイプラインを導入することによって性能が向上できることは理論的には明らかである。ウェーブパイプライン用の CAD、または既存の CAD をウェーブパイプライン設計に使う方法などが揃えば、プロセッサ設計の大きなトレンドの一つになることは十分に考えられる。

謝辞

本研究を進めるにあたり、終始熱心かつ寛容な御指導を賜わりました、日比野靖教授に心から感謝致します。

また、適切な御助言をして頂きました丹康夫助教授、宮崎純助手、堀口進教授、篠田陽一助教授に深く感謝致します。

さらに、パイプラインキャッシュに関する御助言や HSpice シミュレーションによるセルライブラリの作成に御協力を頂いた鵜飼和歳君、MUP に関してその基本構成から例外処理に至るまで多くの御助言を頂いた杉本朗子さんに感謝致します。

その他、貴重な御意見、御討論を頂きました日比野・中島研究室の皆様をはじめ、多くの方々の御助言に対し厚く御礼申し上げます。

参考文献

[1] 伊藤英治, “関数型プログラムの実行に適したマルチスレッド型プロセッサアーキテクチャに関する研究”, 北陸先端科学技術大学院大学修士論文, 1997

[2] 相原孝一, “マルチスレッド型プロセッサ向きのキャッシュ機構のパイプライン化に関する研究”, 北陸先端科学技術大学院大学修士論文, 1997

[3] 杉本朗子, “マルチスレッド型ウルトラパイプライン・プロセッサの FPGA を用いた実装による検証”, 北陸先端科学技術大学院大学修士論文, 1999

[4] 鵜飼和歳, “パイプライン化によるキャッシュの高周波動作の可能性に関する研究”, 北陸先端科学技術大学院大学修士論文, 1999

[5] M. J. Flynn, “Computer Architecture”,

[6] Fabian Klass, Michael J. Flynn, “Comparative Studies Of Pipelined CIrcuits”, 1993

[7] Kevin J. Nowka, Michael J. Flynn, “High-Performance CMOS System Design Using Wave Pipelining”, 1996

[8] Hidechika Kishigami, Kevin J. Nowka, Michael J. Flynn, “Delay Balancing Of Wave Pipelined Multiplier Counter Trees Using Pass Transistor Muliplexers”, 1996

[9] C. Thomas Gray, Wentai Liu, Ralph K. Cavin, “Wave Pipelining: Theory And CMOS Implementation”

[10] Tera Computer, http://www.tera.com/

[11] Hitachi ULSI Engineering Corp., “A 300MHz 4-Mb Wave-Pipeline CMOS SRAM Using a Multi-Phase PLL”, 1995 IEEE International Solid-State Circuits Conference, pp308-309

[12] IBM Corp., “A Commercial Multithreaded RISC Processor”, 1998 IEEE International Solid-State Circuits Conference, pp234-235

[13] 小栗 清, 名古屋 彰, 雪下 充輝, “はじめての PARTHENON”, CQ 出版社, 1994

[14] Gerry Kane, 監訳 前川守, “mips RISC アーキテクチャ — R2000/R3000 —”, 共立出版株式会社, 1994

[15] Wayne P. Burleson, “Wave-Pipelining: Tutorial and Research Survey”, IEEE Transaction VLSI Systems, 1998

Appendix A

MUP に使用した加算器の回路図

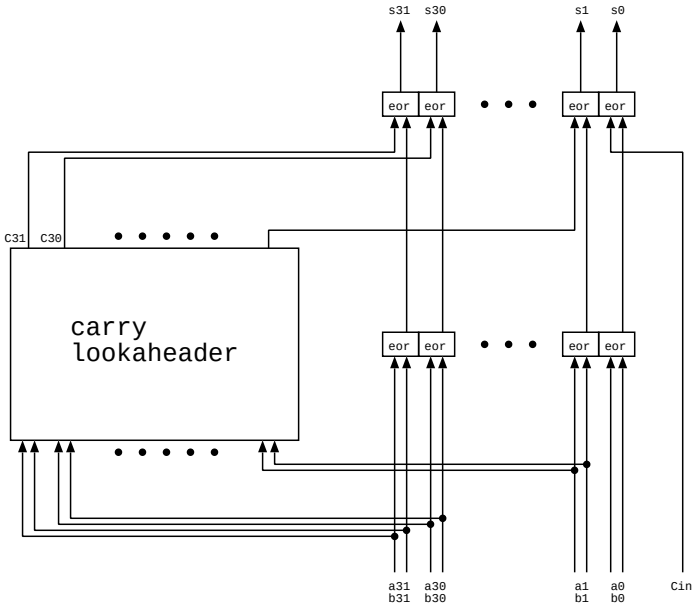

図 8.1: ウェーブパイプライン版 MUP の加算器

桁上げ先見部分は図 8.2のようになっている。図中の BLC(桁上げ先見回路の基本ブロック), PG(桁上げ伝搬ビット、桁上げ生成ビットの生成) と記したブロックの回路図は図 8.3に示した。

ウェーブパイプラインに最も適した加算器という訳ではないが、桁上げ先見を行うので RCA よりも速く、CLA よりも規則的な構造を持つため MUP の加算器として採用した。ウェーブ動作する乗算器の設計に関する [8] でも CSA を用いた乗算器の最終段の CPA にこれを用いている。

MUP の他の部分の回路図は割愛する。

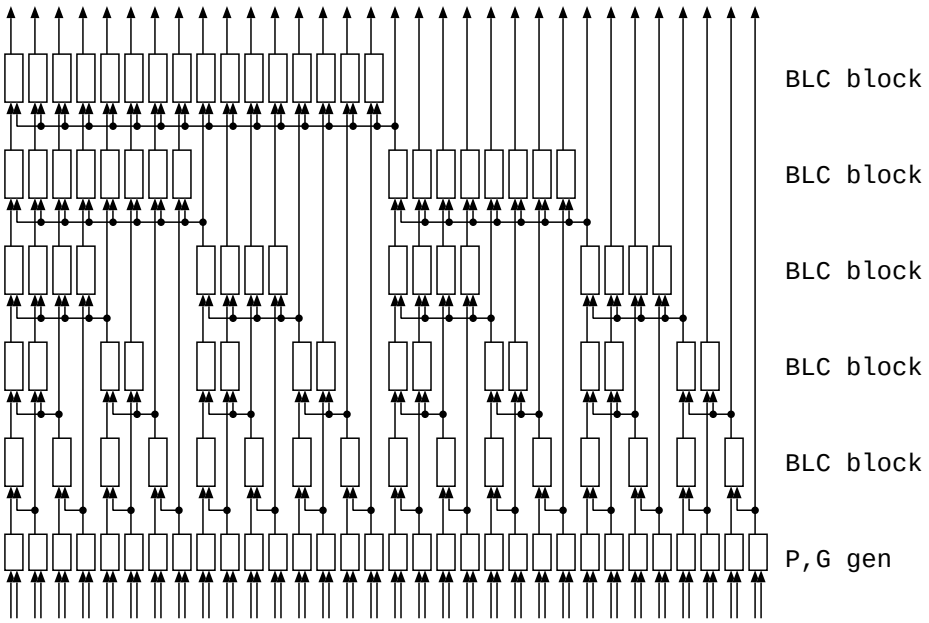

図 8.2: 加算器のブロック図 (桁上げ先見部分)

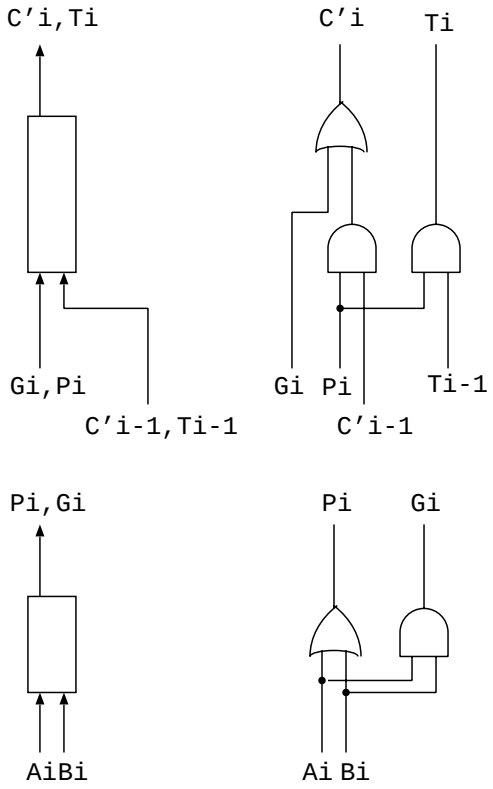

図 8.3: ブロック図とそれに対応する回路

Appendix B

遅延モデル

素子遅延と配線遅延の扱い

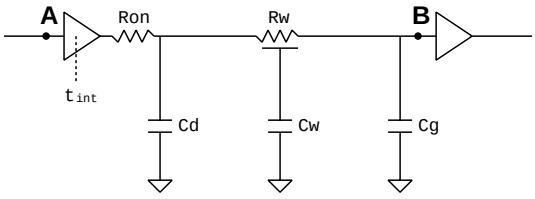

図 8.4: 遅延値の扱い

図 8.4を例に説明する。T_{int}は素子遅延の固定成分、R_{on}はトランジスタの ON 抵抗、C_dは拡散容量、C_gは入力端子容量を表す。

また、C_w、R_wはそれぞれ配線容量、配線抵抗を表し、これらは配線長に比例する値として表現する。その比例定数の導出は以下のとおり。$0.25\mu m$ プロセスを想定し、必要なパラメータは [4] から引用した。

グリッド　1 grid = $0.25[\mu\mathrm{m}]$
配線幅　$0.25[\mu\mathrm{m}]$
酸化膜容量　67.6 (1 層アルミ配線) と 153 (2 層配線) の平均[1]より $110[\mu\mathrm{F/m}^2]$
シート抵抗値　1 層、2 層ともに $0.36[\Omega]$

$$
\begin{aligned}
R_w[k\Omega] &= \text{シート抵抗値 } [\Omega]/\text{配線幅 } [\mu m] \times \text{配線長 } [\mu m] \\
&= 0.36[\Omega]/0.25[\mu m] \times (0.25[\mu m] \times \text{配線長 } [grid]) \\
&= 0.00036[k\Omega] \times \text{配線長 } [grid] \\
C_w[fF] &= \text{酸化膜容量 } [\mu F/m^2] \times \text{配線幅 } [\mu m] \times \text{配線長 } [\mu m] \\
&= 110 \times 10^{-3}[fF/\mu m^2] \times 0.25[\mu m] \times (0.25[\mu m] \times \text{配線長 } [grid]) \\
&= 0.006875 \times [fF] \times \text{配線長 } [grid]
\end{aligned}
$$

[1] x 軸方向と y 軸方向それぞれに均等に配線が分布していると仮定している。

以上により、grid で与えられた配線長から配線抵抗、配線容量を算出する。

配線遅延は素子遅延に含めて考え、図中の A から B までの遅延を $t_{int}+(R_{on}+R_w)\times(C_d+C_g+C_w)$ で近似する。

動的解析ができるようにレイアウト結果を反映した Verilog ソースを生成する必要があるが、nld と Verilog では素子遅延の扱い方に互換性がなく[2]、nld に対応する Verilog 記述を生成する際には注意が必要である。ネットリストの接続状況やレイアウト結果をもとにこれを全てのインスタンスについて計算し、この値を遅延の typ 値として持つように Verilog ソースを生成する。

[2]nld は (固定成分 + 比例定数 × 端子容量) で素子遅延を表すが、Verilog は min/tye/max で表す。

Appendix C

レイアウト方法

レイアウトに関しては以下を想定する。

- スタンダードセル方式 (高さ一定) を想定し、セルの建蔽率を 30%、配線チャネルを 70%とする。これをもとに面積を見積もり、矩形の中にセルを並べる。
- 入出力ピンはそれぞれブロックの上と下に並べる。
- セルは座標の小さな位置から順に並べられ、その後、ペア交換法により総配線長を小さくするための再配置を行う。
- 実配線長はマンハッタン距離で見積もり、仮想配線長は経験的な平均配線長より、ブロックの一辺の長さ (縦幅と横幅の平均) の 1/3 で見積もる。

論理設計が終わってからバッファが大量に入ること、最大遅延ではなく遅延差の短縮が目標であることなどの理由により、以下のような手順を踏む。

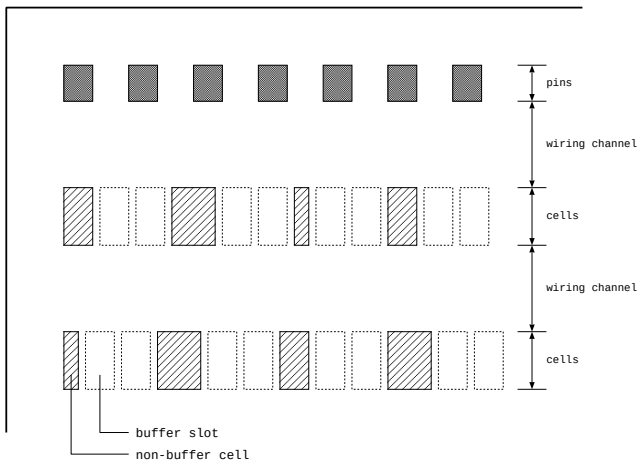

図 8.5: 初期レイアウト

1. **初期レイアウト**　delay balancing によって目標の遅延差を達成するまでに挿入されるバッファ数をあらかじめ知ることはできない。よって初期レイアウトでは、バッファがどの位入りそうかを予測し、バッファ分の面積[3]を確保した面積の中に、図 8.5のように各セルの間にバッファ用のスペース (バッファスロット) を入れて等間隔にセルを配置する。以降、この実セルの配置は固定とする。実セルの配置を変えると遅延差が大幅に増えてしまう恐れがあるからである。
2. (**バッファの挿入**)
3. **バッファのレイアウト**　新たに挿入されたバッファの配置のみを変える。これによって新たに遅延にばらつきが生じるが、さらにバッファを挿入することで解決する。既に配置改善済みのバッファの配置は固定とする。

   上記 2 と 3 を繰り返す (本文参照)。

[3]バッファを入れる場所がなくならないように大き目に与える。対象となる回路の回路量の 3〜4 倍は必要である。

Appendix D

ウェーブパイプライン版 MUP の仕様

データパスの設計には本文中に記述したように以下の点に注意している。

1. レジスタへのアクセスはできるだけまとめて行う。
2. 記憶要素の前にはラッチを置く。(キャッシュと汎用レジスタファイルに関しては前後にラッチを置く。)
3. 記憶要素への入力はラッチから直接 (組合せ論理を介さずに) 入るようにする。
4. ウェーブ動作するステージを何もせずに通過するだけのパスはできるだけ少なくする。

1〜3 はステージ構成やレジスタの配置に反映されている。

4 を満たすための工夫がニ点ある (7.1 節参照)。まず一点目は、制御線はできるだけ使用する直前でデコードしていること。二点目は一時レジスタを使うこと。これによりウェーブ動作するステージを通り過ぎるだけのパスが減る。一時レジスタはスレッド数分用意する。

レジスタ類への入力はラッチから直接入るようにするためデコードされた形でラッチしておく。よってレジスタ類の入出力は図 8.6のようになる。

スレッドが無効 (valid = 0) になる条件は、キャッシュミス時、例外発生時とする。よってキャッシュの他、例外を発生する機能ユニットでも次ステージへ渡す valid ビットが更新される。

最後にウェーブ動作版 MUP の全体の概略図を図 8.7に示す。この図ではスレッド数を 32 と想定している。

各ステージの動作を以下に示す。

- **TS** スレッド選択 — wave pipelined
  - PC,SR を読み出す。
  - CREXT を読み出す。

    CREXT は全てのスレッドで共有するレジスタである。外部例外の検出は DF9 ステージで行われ、検出されれば WB ステージで CREXT が書き換えられる。割込みのための処理は CREXT が書き換えられた 1 サイクル後の (CREXT を書き換えたスレッド自身の)TS ステージから始まる。
  - I-TVF,D-TVF を読み出す。

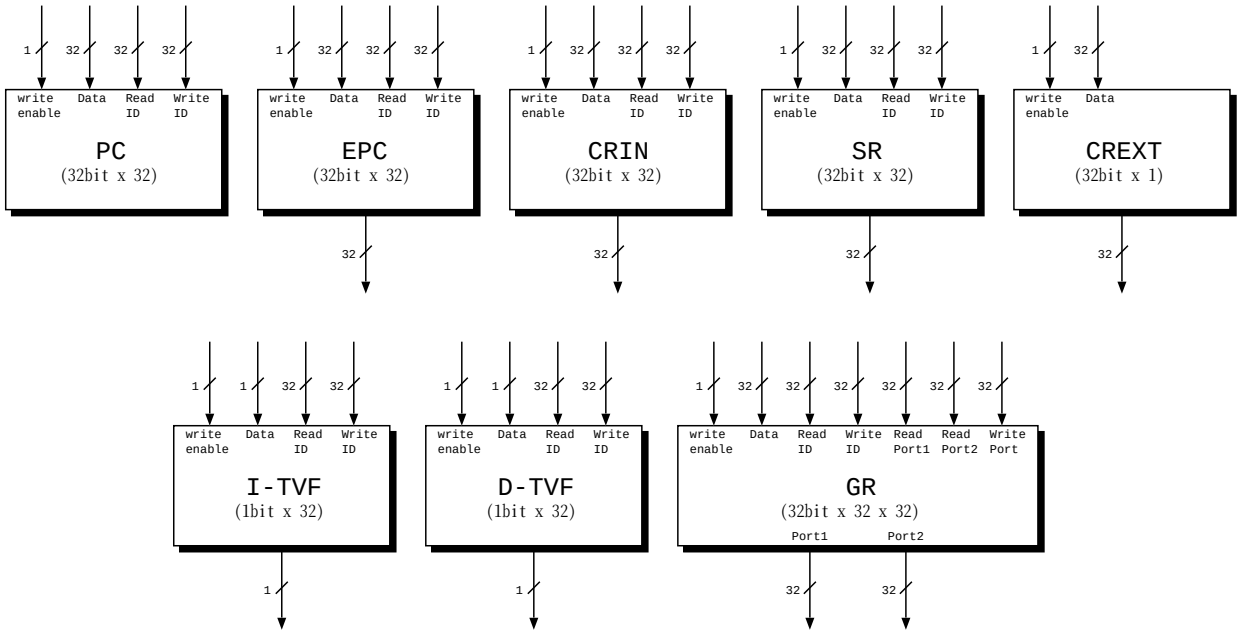

図 8.6: ウェーブ動作版 MUP に使う主なユニットの入出力

MUP では、キャッシュミスによるブロック転送待ちのスレッドは無効とされる。I-TVF は命令キャッシュミスによるブロック転送待ちを示すフラグをスレッド毎に持っている。同様に D-TVF はデータキャッシュミスによるブロック転送待ちを示すフラグをスレッド毎に持っている。I-TVF と D-TVF からカレントスレッドのフラグを読み、I/D キャッシュが共にブロック転送中でなければスレッドは有効とする。

– アドレスエラーチェックをする。

キャッシュのアドレスデコードステージではアドレスエラーチェックをしない。よってキャッシュにアドレスを渡す前にアドレス違反がないかチェックをする。エラーがあった場合はアドレスエラー例外が発生したとして SR を更新しカレントスレッドを無効化する。

- **IF1〜IF9** 命令フェッチ (命令キャッシュ)
  - IF1 で命令キャッシュにアドレス、スレッド ID、スレッド有効フラグを渡す。
  - IF9 で命令キャッシュからデータ、ヒット / ミス情報、ブロック転送終了信号を受け取る。
- **ID** 命令デコード — wave pipelined
  - デコードの分散のため、次の RF ステージで使用する情報のみをデコードする。
  - ウェーブ動作する EX ステージを通過するだけの情報である rd(汎用レジスタファイルの書き込みポート) を一時レジスタに格納する。
  - 命令キャッシュのヒット / ミス結果をみて、ミスだった場合はスレッドを無効化する。

- **RF** レジスタフェッチ — overlapped
  - 汎用レジスタファイル、EPC、CRIN を読み出す。

    これらのレジスタに加えて次項の I-TVF も含めて、このステージで使用されるレジスタは全てスレッド数個分用意されている。また、先の ID ステージでレジスタのアクセスに必要な情報はデコード済であり、組合せ論理を持たない。。このステージがオーバーラップ可能なのはウェーブパイプラインによるものではなく、スレッド数個分用意されたレジスタセットによるものであるが、スルーするパスはウェーブ動作させなくてはならず、遅延バッファの挿入が必要である。
  - 命令キャッシュのヒット/ミス結果を I-TVF に書き込む。
  - EX ステージでは使われない情報である外部例外情報、RFE ビット、ライトバック制御信号を一時レジスタに格納する。
- **EX** 実行 — wave pipelined
  - ID ステージでデコードしなかった情報を (使用する直前で) デコードする。
  - ALU 命令の実行、分岐先アドレスの計算を行う。
  - アドレスエラーチェックをする。
- **DF1〜DF9** データフェッチ (データキャッシュ)
  - DF1 でデータキャッシュにアドレス、スレッド ID、スレッド有効フラグ、ストアデータ、LD/ST、W/H/B を渡す。プロセッサ側では DF8 まで何もしないが、浮動小数点演算、シフト演算があれば有効に利用できるだろう。
  - DF8 で一時レジスタに格納した情報をデータパスに戻す。
  - DF9 でライトバック用の MUX やデコードをし、データキャッシュからデータ、ヒット/ミス情報、ブロック転送終了信号を受け取る。
- **WB** ライトバック
  - スレッドが有効かつ例外が発生していなければ EPC 以外の全てのレジスタに書き込む。スレッドが有効かつ例外が発生していれば EPC にのみ書き込む。スレッドが無効なら何もしない。
  - 外部例外が検出されたら CREXT を書き換える。

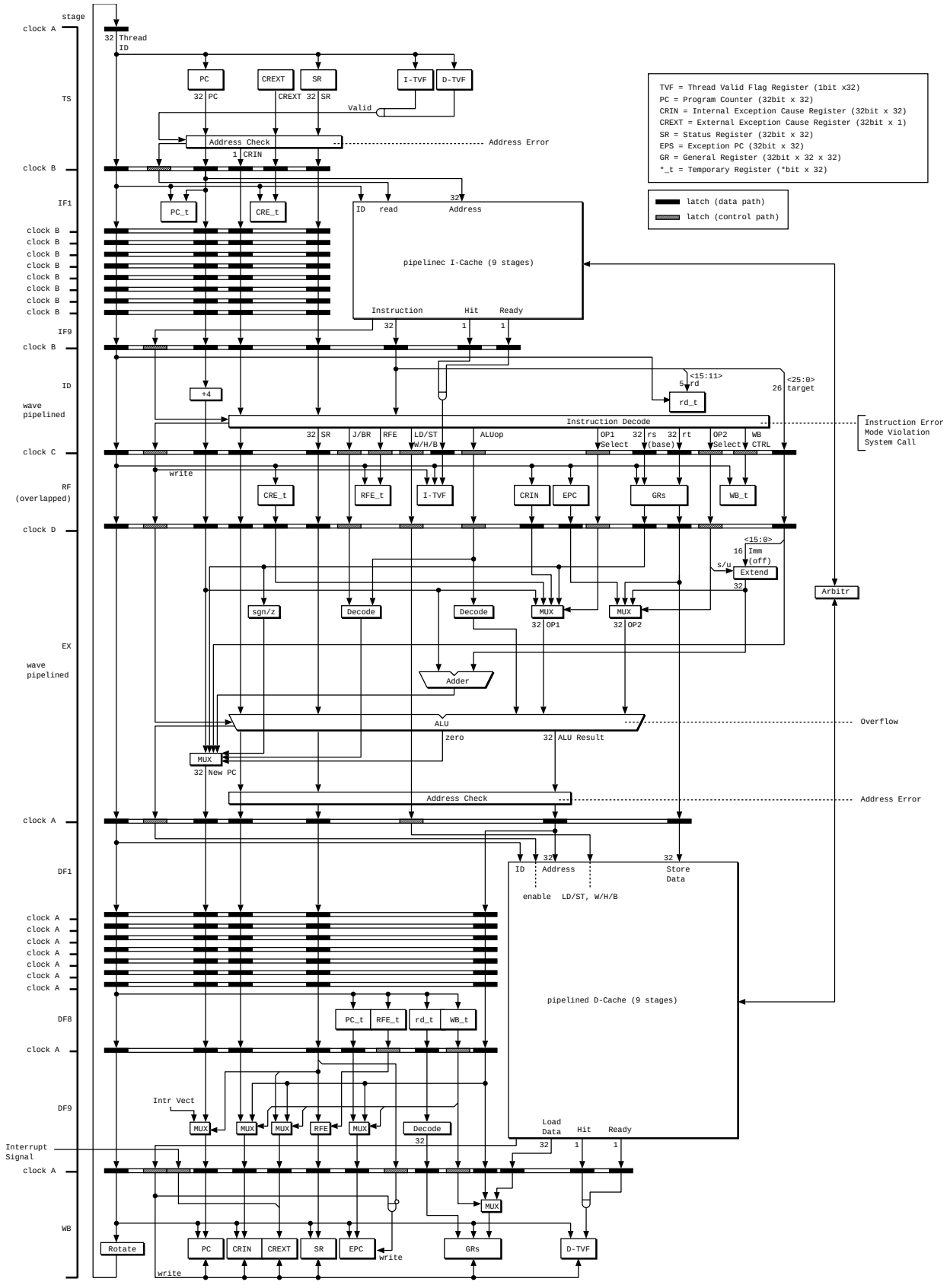

図 8.7: ウェーブ動作版 MUP の概略図