タイミング制約を緩和するクロッキング方式

神保 潮

博士（情報学）

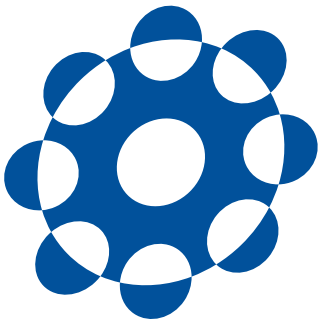
総合研究大学院大学
複合科学研究科
情報学専攻

平成３０（２０１８）年度

学位論文 博士（情報学）

タイミング制約を緩和するクロッキング方式

神保 潮



総合研究大学院大学

SOKENDAI (The Graduate University for Advanced Studies)

2019 年 03 月

本論文は総合研究大学院大学 複合科学研究科 情報学専攻に
博士 (情報学) 授与の要件として提出した博士論文である

概要

半導体プロセスの微細化に伴う素子遅延のばらつきの増加が，回路設計における大きな問題となりつつある．ばらつきが増大していくと，従来のワースト･ケースに基づいた設計手法は悲観的になりすぎる．

この問題に対処するため，Razor などの動作時にタイミング故障を検出し回復する手法が提案されてきた．タイミング故障とは，遅延の動的な変化によって設計者の意図とは異なる動作が引き起こされる過渡故障である．ワースト･ケース設計では，熱暴走がない限りタイミング故障は発生しない．タイミング故障を検出し回復する手法では，タイミング故障の発生を許容することで，ワースト･ケース設計から脱却できる可能性をもつ．我々もまた，二相ラッチと Razor と似た TF 検出とで構成される動的タイム･ボローイングを可能にするクロッキング方式を提案してきた．

しかしながら，タイミング故障を検出し回復する手法の課題として，ダイナミック･プリチャージ･ロジック，特に SRAM への適用が考慮されてこなかったという問題があった．また，動的タイム･ボローイングを可能とする方式は，ごく簡単な回路によって最低限の動作確認がされただけであった．実用化のためには，最終的には，Out-of-Order プロセッサなどの現実的な回路に対して適用した上で，LSI 化し，評価を行う必要がある．また，既存の回路を入力として，提案クロッキング方式が適用された回路を出力する自動変換ツールの開発が不可欠である．

本論文では，その最終段階までには至ってはいないが，そのための重要なステップとして，以下を行った：

1. SRAM を対象としたタイミング故障検出手法
2. 簡単な回路に対する自動変換と方式の評価
3. 二相ラッチ化手法
4. Razor の Rocket への適用

まず，SRAM を対象としたタイミング故障検出手法を提案した．この提案手法により，タイミング故障検出技術がプロセッサ内の主要なコンポーネントであるレジスタ･ファイルや L1C に対しても適用可能となり，これらがボトルネックとなり得る問題を解消した．

また，動的タイム・ボローイングを可能にするクロッキング方式ための自動変換ツールのフレームワークを確立した．リプル・キャリー・アダーを用いたカウンタを対象として，タイミング故障検出と回復のための回路を付加した回路に自動変換した．出力された回路を FPGA に実装して評価し，通常の単相 FF 方式に対して 1.6 倍の周波数で動作することを確認した．

動的タイム・ボローイングを可能にするクロッキング方式の適用手法として，二相ラッチに基づくパイプライン設計手法が必要とされる．二相ラッチに基づくパイプライン設計手法として，単相 FF を用いてデザインされた回路を自動的に二相ラッチを用いた回路に変換するアルゴリズムを提案した．実験により，提案手法が，ゲート数約 3.4 万，配線数約 9.7 万程度の回路に対しても，約 375 秒の実用的な時間で動作することを確認した．

また，動的タイム・ボローイングを可能にするクロッキング方式を Rocket に適用する前段階として，Razor を適用し，それを FPGA 上に実装した．Rocket はある種の命令に対して Out-of-Order 完了を許すプロセッサである．タイミング故障からの回復を可能とするためには，完全な In-Order 完了プロセッサでなければならないため，これは問題である．そのため，Rocket の完全な In-Order 化を行った．

目 次

図 目 次

表 目 次

第1章

序論

半導体プロセスの微細化に伴って，素子遅延のばらつきが大きな問題となりつつある [1]．ここで特に問題とされているのは，チップ間に跨る (Die-to-Die: D2D) システマティックなばらつきではなく，チップ内の (WithIn-Die: WID) ランダムなばらつきである．

ランダムばらつきの問題は，半導体の微細化に伴って生じる本質的な問題であり，原理的に避けえない．現在の半導体製造技術では，トランジスタや配線といった回路素子のサイズは，原子数十個というオーダとなり，原子 1 個分の製造誤差が大きな影響を持つようになる．トランジスタや配線の機械的な寸法の他，トランジスタ中の不純物の濃度などもランダムにばらつく．

図 1.1 に，微細化に伴ってばらつきが増加することの影響を示す．同図は 2 つの

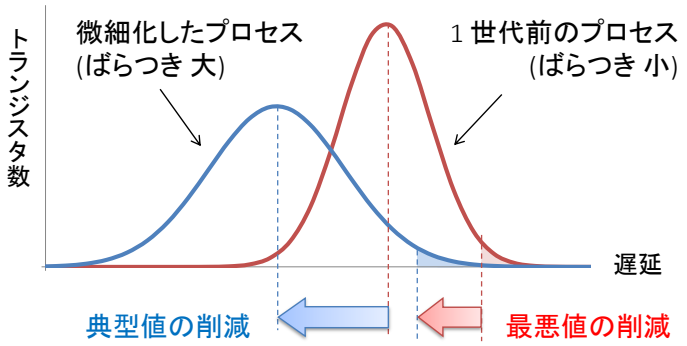


図 1.1: 2 つのテクノロジ・ノードの素子遅延の分布．

異なるプロセス技術により製造される素子遅延の分布を表している．微細化に伴うランダムばらつきの増大により，微細化したプロセスでの素子遅延の分布は，1 世代前のプロセスのそれに比べて，分散が大きくなる．そのため，歩留まりによって決まる最悪値の削減幅は，典型値の削減幅に比べて少なくなってしまう．

さらに微細化が進むと，ばらつきも更に増大し，典型値と最悪値の差はより広がっていく．その結果，従来の最悪値に基づいた設計手法では，いずれ LSI の動作速度は向上しなくなってしまう．

1.1　ばらつき

ロジックの遅延のばらつきは一般に，その原因によって，**PVT**，もしくは，**PVTA ばらつき**に分類される (**P**rocess, **V**oltage, **T**emperature, **A**ging)．本論文では，これらに加えて，ロジックの入力の変化の仕方によるばらつきも考慮する．これを**入力ばらつき** (**I**nput) と呼ぶ．入力ばらつきについては 2.1 節で詳しく述べる．

各ばらつきは，時間的および空間的な成分を有する．

空間的成分　PVTAI ばらつきの空間的成分は次のようにまとめられる：

P：P ばらつきは，半導体製造プロセスにおいて発生する要因であり，その原因に基づいてダイ間（die-to-die : **D2D**）のシステマティックなもの，およびダイ内（within-die : **WID**）のランダムなばらつきに大別される．後者は，ドーパント変動，ライン･エッジ･ラフネスなどのランダムな現象によって引き起こされる．

V：電源電圧は D2D にばらつく．同時に，電圧ドループ（voltage droop）は，モジュールや部分回路など，WID の局所領域の変動を引き起こす．

T：温度は，周囲の温度，冷却効率，ダイの負荷などの動作条件に基づいて，D2D に変化する．同時に，WID でも変動する．WID における高温の局所領域は，**ホット･スポット**と呼ばれる．

A：経年劣化の度合いは，各回路素子の温度および電流に依存する．したがって，A ばらつきは，上記の他のばらつきに応じて D2D および WID 成分をもつ．

I: ロジックの入力となる記憶素子や外部入力の状態に応じて，ロジックの遅延が変動する．すなわち，WID 成分をもつ．

時間的成分　PVTAI ばらつきの時間的成分，すなわち各要因が遅延を変化させる時間的な粒度は，以下である：

P: P ばらつきは製造時に発生し，それ以後時間的には変化しない．

A: 経年劣化の影響は月～年単位で増加する．

T: 回路の各所の温度は，負荷の変動ために秒以下の単位で急速に変動することがある．

V: 回路の各所の電圧は，IR ドロップのためにクロック･サイクル，すなわち，ナノ秒単位で変動し得る．

I: ロジックの入力となる記憶素子や外部入力の変化の仕方がクロック･サイクルごとに変化することで，ロジックの実効的な遅延もクロック･サイクルごとに変動する．このことについて，2.1 節で詳しく述べる．

これらのばらつきは，下に行くにつれて粒度がより細かい．特に，P/ATVI ばらつきは，それぞれ静的/動的ばらつきと呼ばれ，区別される．

細粒度ばらつきに対する対応手法　LSI の最小加工寸法が小さくなるにつれて，空間的および時間的に細粒度な成分が大きな影響を持つ．この問題に対処するために，ワースト･ケースの遅延に基づくものではなく，個々の回路素子の実際の遅延に基づく方法が提案されている．

たとえば，統計的な静的タイミング解析（Statistical Static Timing Analysis: SSTA）[2, 3] は，統計モデルに基づいてパス遅延を推定することで，パス上のすべての要素が最悪の遅延を持つ可能性は非常に低いことを推定に組み込み，WID な P ばらつきの影響を低減する．しかし，この手法は動的なばらつき（VTAI）に関しては対象としていない．

一方で，次の節で説明するタイミング故障検出・回復手法は，動的なばらつきに対応する回路及びアーキテクチャ上の手法である．

1.2　タイミング故障検出・回復

タイミング故障（Timing Fault : **TF**）とは，遅延の動的な変化によって設計者の意図とは異なる動作が引き起こされる過渡故障である．ワースト・ケース設計では，想定した動作条件内のワースト・ケースの遅延を見積もった上で，その遅延に対しても TF が発生しないように設計する．したがって，ワースト・ケース設計に基づいて設計・製造された LSI では，TF は，想定した動作条件を外れた状況以外では発生しない．そのような状況は，温度センサの故障によって熱暴走を起こした場合などには実際に観測することができる．

一方で，TF を検出し TF から回復する機構を備える **TF 検出・回復手法** [4–9] では，想定した動作条件内において TF の発生が許容される．3.2 節で詳しく述べる **Razor** [4–6] はその代表例である．

TF 検出・回復手法は，**DVFS** (Dynamic Voltage and Frequency Scaling) [10] と組み合わせると，見積もりではない，実際の遅延に応じた動作を実現することができる．ワースト・ケース設計では，TF が発生しないようにワースト・ケースの遅延を見積もり，見積もりに応じて **V**（Voltage：電源電圧）と **F**（Frequency：動作周波数）の組が設定される．TF 検出・回復手法では，それより V を下げる，または，F を上げることができる．そのようにすると，いずれ TF が発生し検出されるが，その検出直前の V–F が，見積もりではない，そのチップのその時の動作環境における実際の遅延に応じた V–F である．後は，TF が頻発しないように V–F を調整すればよい．このようにすれば，見積もりではない実際の遅延に応じた動作を実現することができる．

しかし，Razor などの既存の手法の効果は，実際には V–F のマージンを削減することに留まる．TF を起こしうるクリティカルなパスはチップ内に数多く存在する．そのため，そのチップのその時の動作環境における実際のワースト遅延を超えて V–F を設定すると，それらのパスのうちどれか 1 つで TF が発生する確率は 1 に近くなる．すなわち，毎サイクルのように，TF を検出し，回復処理を行うことになり，チップとしてほとんど機能しなくなってしまう．逆に言うと，Razor などの既存の手法の効果は，ワースト・ケース設計における見積もりには含まれていた V–F のマージンを削減することに留まると言える．

これらのことについては，3.1 節でより正確に述べる．

1.3 動的タイム･ボローイングを可能とする方式

TF 検出・回復の考えを推し進め，我々は，**動的タイム･ボローイングを可能にするクロッキング方式**を提案してきた [11–19]．本方式は，TF 検出と二相ラッチという既存の方式を組み合わせたものではあるが，この組み合わせによってはじめて**動的タイム･ボローイング** (Dynamic Time Borrowing: **DTB**) という効果が得られる．

通常用いられる単相 FF における FF は，クロック･エッジの一瞬だけ開いて，信号を通過させると解釈することができる．それとは異なり，二相ラッチにおけるラッチは，サイクルの半分の期間で開いており，この期間に信号はラッチを透過的に通過することができる．しかし，通常の二相ラッチを用いたクロッキング方式では，この期間を活用することはできない．この期間を活用すべくサイクル･タイムを短縮すると，TF が発生するからである．通常の二相ラッチ方式では，信号はラッチが閉じている期間に到着し，開いた瞬間に通過することになる．逆に言うと，二相ラッチに TF 検出を組み合わせることによってはじめて，信号は開いているラッチを安全に通過できるようになる．

そして，信号が開いているラッチを通過することによって，動的タイム･ボローイングという効果が得られるようになる．開いているラッチはバッファと等価であり，それを信号が通過する時，回路は長大な組み合わせ回路のように動作する．この結果，ステージの実効遅延の長短は前後のステージ間で自動的に融通されるようになる．

通常の二相ラッチ方式で可能になるタイム･ボローイングは，**静的**タイム･ボローイングと呼ぶべきものであり，設計時にクリティカル･パス遅延がステージ間で融通される．それに対して，提案のクロッキング方式で可能となる**動的**タイム･ボローイングでは，動作時に実効遅延がステージ間で融通される．すなわち DTB では，あるステージで実効遅延がサイクル･タイム以上に伸びてしまった場合，この超過分を次のステージに持ち越すことができる．この際，次のステージの実効遅延が短ければ，この超過分は相殺される．

このとき回路は，クリティカル･パス遅延よりはるかに短い各ステージの実効遅延の平均値で動作する．更に，チップ内のランダムなばらつきは，大数の法則によって平均化される．最大動作周波数は，TF を検出できる周波数で決まり，通常のクロッキング方式のちょうど 2 倍となる．

1.4　本論文の貢献

動的タイム・ボローイングを可能とする方式は我々が提案してきたものであるが，ごく簡単な回路によって最低限の動作確認がされただけであった [13]．実用化のためには，最終的には，Out-of-Order プロセッサなどの現実的な回路に対して適用した上で，LSI 化し，評価を行う必要がある．また，既存の回路を入力として，提案クロッキング方式が適用された回路を出力する自動変換ツールの開発が不可欠である．

本論文では，その最終段階までには至ってはいないが，そのための重要なステップとして，以下を行った：

1. SRAM を対象としたタイミング故障検出手法
2. 簡単な回路に対する自動変換と方式の評価
3. 二相ラッチ化手法
4. Razor の Rocket への適用

以下，それぞれについて述べる：

1. SRAM を対象としたタイミング故障検出手法　現実の回路においては，組み合わせ回路に加えて SRAM も欠くべからざる要素である．そして SRAM の読み出し回路は，ダイナミック・プリチャージ・ロジックとして実装されることが多い．にもかかわらず，Razor をはじめとする既存の TF 検出・回復手法は，専らスタティック・ロジックのみを対象としており，ダイナミック・プリチャージ・ロジックへの適用については言及すらされていなかった．

そこで本論文では，ダイナミック・プリチャージ・ロジックとして，特に SRAM を対象とした TF 検出手法を提案する．この提案手法により，TF 検出技術がプロセッサ内の主要なコンポーネントであるレジスタ・ファイルや L1C に対しても適用可能となったことで，これらがボトルネックとなり得る問題を解消した．

なお，この手法は，動的タイム・ボローイングを可能とするクロッキング方式に限らず，一般の TF 検出・回復手法にも適用可能である．

2. 簡単な回路に対する自動変換と方式の評価 まず，自動変換ツールのフレームワークを確立した．リプル･キャリー･アダーを用いたカウンタを対象として，TF 検出と回復のための回路を付加した回路に自動変換した．出力された回路を FPGA に実装して評価し，少なくとも簡単な回路に対しては提案クロッキング方式が想定した効果を発揮することを確認した．

なおこの際には，次で述べる二相ラッチ化は手動で行っている．

3. 二相ラッチ化手法 1.3 節で述べたように，提案クロッキング方式は二相ラッチによるクロッキング方式をベースとするものである．一般に二相ラッチによるパイプライン設計は困難であるため，設計手法についても考慮する必要がある．

本論文では，二相ラッチに基づくパイプライン設計手法として，単相 FF を用いてデザインされた回路を自動的に二相ラッチを用いた回路に変換するアルゴリズムを提案した．

単相 FF を用いてデザインされた回路をラッチを用いた回路に変換する問題は，最小カット問題の一種に帰着する．ただしこの際，始点から終点に至るすべての道にカット･エッジをただ 1 つ含むという制約がある．既存の最小カット･アルゴリズムでは，この制約を満たすことができない．本稿では，この制約が，カットが逆方向カット･エッジを含まないことと等価であることを証明し，逆方向カット･エッジのない最小カットを見つけるアルゴリズムを提案する．このアルゴリズムにおいて最もオーダが大きい部分は既存の最大フロー･アルゴリズムであり，提案アルゴリズム全体のオーダはこれより悪化することはない．実験により，ゲート数約 3.4 万，配線数約 9.7 万程度の回路に対しても，約 375 秒の実用的な時間で最適解が求められることが分かった．

4. Razor の Rocket への適用 動的タイム･ボローイングを可能とするクロッキング方式を適用する前段階として，Razor を現実的なプロセッサ Rocket に適用する方法を示した．Rocket は，RISC-V アーキテクチャに完全準拠するスカラ･プロセッサで，CSR (Control and Status Registers) を持ち，Linux をブートすることができる．著者が調査した限り，このような現実的なプロセッサに対して TF 検出を適用した事例はない．論文の Razor 化 Rocket は，FPGA 上に実装され Linux をブートできるものとしては，TF 検出最初のテスト･ベッドとなるであろう．

Rocket はある種の命令に対して Out-of-Order 完了を許すプロセッサである．TF

からの回復のためには In-Order 完了を必要とするため，Rocket の完全な In-Order 化が必要となる．

1.5　本論文の構成

次章以降の本論文の構成は，以下のとおりである：

第 2 章　**クロッキング方式**

本章では，背景知識として，既存のクロッキング方式について実効遅延の観点からまとめる．単相 FF や二相ラッチ方式はワースト・ケース設計によるものであり，これらの手法はワースト・ケースの遅延よりもサイクル・タイムを小さくすることができない．本章で述べるダイアグラムにより，後述の実効遅延に基づく動作を可能にする方式との違いが明らかにされる．

第 3 章　**タイミング故障検出・回復**

本章では，TF 検出手法として Razor について詳述する．Razor はタイミング故障検出のための FF である Razor FF と，タイミング故障からの回復技術からなる．

第 4 章　**SRAM のタイミング故障検出**

本章では，本論文の貢献の 1 つである，SRAM への TF 検出の適用手法を詳述する．本提案に関しては TF 検出の適用手法として独立しているため，第 3 章において Razor について述べた後での方が話のつながりがよいと考え，動的タイム・ボローイングを可能にするクロッキング方式の章よりも先に述べる．

第 5 章　**動的タイム・ボローイングを可能にするクロッキング方式**

本章では，まず我々が既に提案した動的タイム・ボローイングを可能にするクロッキング方式について述べる．動的タイム・ボローイングを可能にするクロッキング方式は，大まかには TF 検出と二相ラッチを組み合わせたものであるが，その特徴を活かすための特別の工夫を要している．そうした回路構成のポイントについて述べる．また，動的タイム・ボローイングにつ

いて詳しく述べ，既存手法では成しえなかった，実効遅延に基づく動作が可能になる理由を示す．

次に，このクロッキング方式の回復機構も含めた評価を行う．対象は RCA カウンタである．

第 6 章 二相ラッチ化手法

本章では，本論文の貢献の 1 つである，二相ラッチ化手法として提案したアルゴリズムを述べる．FF を用いた回路をラッチを用いた回路に変換する問題は，最小カット問題の一種に帰着できる．その特殊な最小カット問題を解くためのアルゴリズムを提案する．

第 7 章 **Razor の Rocket への適用**

本章では，本論文の貢献の 1 つである，Razor の Rocket を対象とした適用について述べる．Rocket のマイクロ・アーキテクチャを説明し，Rocket への Razor の適用において必要なポイントをまとめる．最後に，Rocket の適用による回路オーバーヘッドを FPGA において評価する．

第 8 章 結論

本論文の内容についてまとめ，今後の展望を示す．

第2章

クロッキング方式

本章では，次章で述べる動的タイム・ボローイングを可能にするクロッキング方式をよりよく理解するために，まず既存のクロッキング方式を説明する．2.1 節では，クロッキング方式の理解に便利な**タイミング・ダイアグラム**を導入する．2.2 節で，クロッキング方式の表現について述べる．2.3 節では，クロッキング方式の要諦について述べる．2.4 節で単相 FF，2.5 節で二相ラッチのタイミング制約について説明する．2.6 節では，二相ラッチによって可能になる静的タイム・ボローイングについて述べる．

2.1 タイミング・ダイアグラム

図 2.1 に示すグラフを，本論文ではタイミング・ダイアグラム（あるいは，単にダイアグラム）と呼んでいる．通常のタイミング・チャートが論理値—時間の関係を表すのに対して，タイミング・ダイアグラムは時間—空間の関係を表す．同図中，右方向が時間を，下方向が回路中を信号が伝わって行く方向を表し，時間の経過につれて信号が伝わっていく様子を俯瞰することができる．

実際のロジックには，それぞれ遅延が異なるパスが数多く存在する．ダイアグラムでは，入力の変化によって出力が変化した時，その信号伝達を，入力が変化した点から出力が変化した点までを（右下がりの）直線矢印で結んで表す．

実効遅延　ロジック中の信号の伝達の仕方は，ロジックの入力の変化の仕方によって異なる．一部の信号の遷移はマスクされるため，一般にすべてのパスが出力の変化

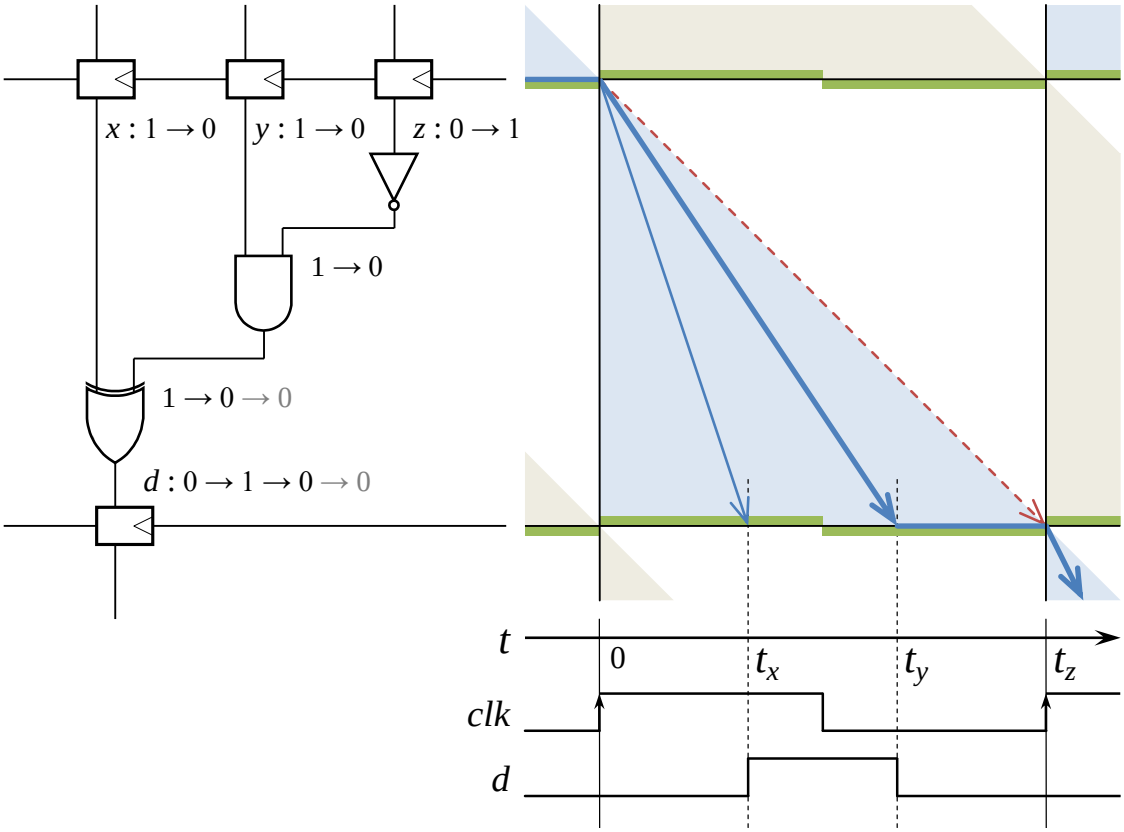


図 2.1: 単相 FF のタイミング・ダイアグラム

に関与する訳ではない．ロジック中のあるパスを通った信号によってロジックの出力が変化したとき，そのパスは活性化されたと言う．

ダイアグラムでは，あるサイクルにおいて最後の出力の変化をもたらした信号の伝達を実線矢印で表す．この実線矢印の遅延（図上で縦方向の距離）を，そのサイクルの**実効遅延**と呼ぶ．

ダイアグラム上で実線矢印が存在可能な範囲は，ロジック内の最小遅延とクリティカル・パス遅延を表す直線に挟まれた三角形の領域となる．ダイアグラムではこの領域を網掛けにより示す．図中の網掛けの二色については後述する．

なおダイアグラムでは，各ステージのクリティカル・パスに対応する直線矢印の角度を 45° としている．こうすることによって，各ステージの遅延は，ダイアグラム上のステージの幅によって表現することができる．

入力ばらつき 実効遅延という言葉を用いるなら，**入力ばらつき**は，ロジックの入力の変化の仕方に応じて生じる実効遅延のばらつきと定義することができる．

ロジックの出力が一度も変化しなかった時，実効遅延は 0 と考えられる．すなわち入力ばらつきによって，ロジックの実効遅延は 0 からクリティカル・パス遅延まで変化することになる．他の要因によってはロジックの（クリティカル・パス）遅延は数割程度しかばらつかないことを考えると，入力ばらつきは非常に大きいと言える．

2.2 クロッキング方式の表現

次に，図 2.1 でのクロッキング方式の表現を説明する．

エッジ・トリガ動作 同図はマスタ–スレーブ構造を持つ FF を念頭に描かれている．同図において，FF の下にある縦実線はラッチが閉じている状態を，縦実線と次の縦実線の間の空白は，ラッチが開いている (transparent) 状態を，それぞれ表している．信号の矢印が実線にぶつかった場合，ラッチが開くまで信号は下流側に伝わらない．エッジ・トリガ動作は，マスタ–スレーブ・ラッチを互い違いに記述することで生じる隙間から信号が「漏れる」様子で直感的に表すことができる．

フェーズ パイプライン動作を行う際には，FF と次の FF に挟まれたロジックがパイプライン・ステージとなり，各クロック・サイクルごとに各ステージが並列に動作を行うことになる．

パイプライン動作においては，一連の処理 —— 典型的には，パイプライン型プロセッサにおける 1 つの命令の処理 —— は，あるサイクルにおいてあるステージで処理された後，次のサイクルにおいて次のステージの処理へと次々引き継がれていく．この一連の処理のことをある**フェーズ**の処理と呼ぶ．

ダイアグラムでは，あるフェーズの処理と次のフェーズの処理を，矢印が存在し得る領域の網掛けの色を分けることで区別している．

2.3 クロッキング方式の要諦

クロッキング方式の要諦は，あるフェーズの信号が前後のフェーズの信号と「混ざる」ことがないように分離した上で，処理を次のサイクルに次のステージへと引

き継いでいくことである．

ダイアグラム上では，以下の 2 つの条件が満たされていればよい：

1. 実線矢印をたどって，次のサイクルに次のステージへと至ることができる．
2. 矢印が存在し得る範囲を表す網掛けの領域が，前後のフェーズの, すなわち，色の異なる網掛けの領域と重ならない．

クロッキング方式のタイミング制約は，この 2 条件から導かれる．

次章からは，ダイアグラムを用いてそれぞれのクロッキング方式について説明する．

2.4　単相 FF 方式

単相 FF 方式が上記の条件を満たして正しく動作するためには，各ステージにおいて，あるクロック・エッジで入力側の FF の出力が変化してから，次のクロック・エッジまでに出力側の FF の入力に信号が到着しなければならない．すなわち，サイクル・タイムを τ とすると，各ステージのロジックのクリティカル・パスの遅延が τ 未満であればよいということになる．このことを，最大遅延制約は 1τ/1ステージと表現することとする．

図 2.1（および，図 2.2 (a)）では，クリティカル・パスの遅延を表す赤い 45° の線がちょうど次のクロック・エッジに到着しており，最大遅延制約の限界を達成した場合を表している．なお，簡単のため，FF やラッチのセットアップ/ホールド時間やスキューなどは省略しているが，これらを議論に組み込むことは容易である．

通常，クリティカル・パスが活性化される確率は高くない．図 2.1 のように，実効遅延とクリティカル・パス遅延の差の分だけ，無駄な待ち時間が生じることになる．

2.5　二相ラッチ方式

図 2.2 (b) に，二相ラッチ方式のダイアグラムを示す．二相ラッチ方式は，単相 FF 方式における FF を構成するマスタ，スレーブの 2 つのラッチのうちの 1 つをロジックの中間へと移したものと理解することができる．移されたラッチによって分割された後のステージを特に**半ステージ**と呼ぶ．

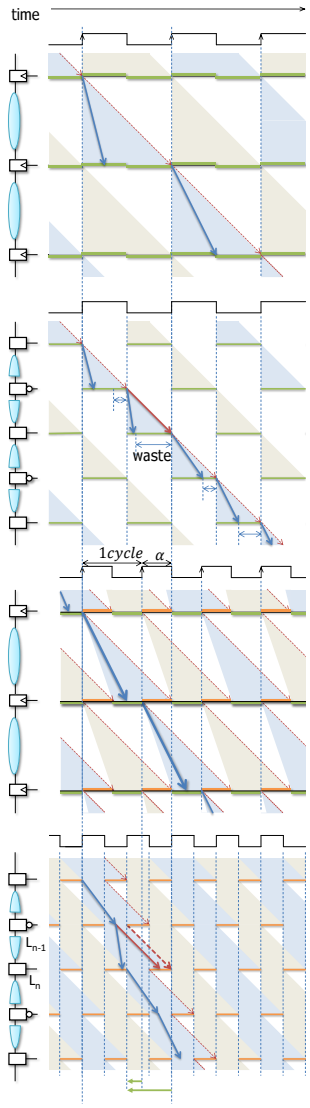


図 2.2: 各クロッキング方式のタイミング･ダイアグラム：
上から，(a) 単相 FF，(b) 二相ラッチ，(c) Razor FF，(d) DTB を可能にする方式

単にラッチの位置を動かしただけなので，二相ラッチ方式の最大遅延制約は，基本的には，$0.5\tau/1$ 半ステージ となり，単相 FF の $1\tau/1$ステージ と変わらない．

2.6　静的タイム・ボローイング

ただし二相ラッチ方式では，この制約を部分的に緩和できることがある．単相 FF 方式では，エッジ・トリガ動作により，信号が次のステージへと伝播するタイミングがクロック・エッジに限定される．一方，二相ラッチ方式では，ラッチが開いている期間を活用することによって，遅延をステージ間で融通できる場合がある．

このことは一般に，タイム・ボローイングと呼ばれる．本稿では，動的タイム・ボローイング (Dynamic Time Borrowing: **DTB**) と区別するため，二相ラッチのそれを**静的タイム・ボローイング** (Static Time Borrowing: **STB**) と呼ぶことにする．

図 2.3 に，静的タイム・ボローイングの様子を示す．同図のように半ステージ間の遅延がバランスされていない場合に，STB は効果がある．単相 FF 方式では，サイクル・タイムは最も長いステージのクリティカル・パス遅延によって決まるため，短いステージでは無駄な時間が生じる．一方，二相ラッチ方式では，同図のように，クリティカル・パス遅延を表す直線が一本に結べれば，前述したクロッキング方式の 2 条件が満たされる．同図中，最も長い半ステージには 1τ が割り当てられている．すなわち，二相ラッチの最大遅延制約は，1 つの 0.5 ステージに限れば，$1\tau/0.5$ステージと，単相 FF 方式の 2 倍となる．ただし全体では，遅延の累積で $0.5\tau/0.5$ステージと，単相 FF 方式のそれと変わらない．

逆に，半ステージ間で遅延がバランスしている場合には，STB の恩恵は生じない．この場合，図 2.2 (b) に示すように，信号は必ず次のラッチが閉じている期間に到着しなければならず，開いている期間は使われない．開いている期間を活用すべくそれ以上にサイクル・タイムを短縮した場合には，クリティカル・パスが連続で活性化するといずれサンプリング期間に間に合わず，TF となってしまう．

回路設計においては，まずステージ間で遅延をバランスさせることが肝要であり，STB を積極的に活用することは勧められてはいない．

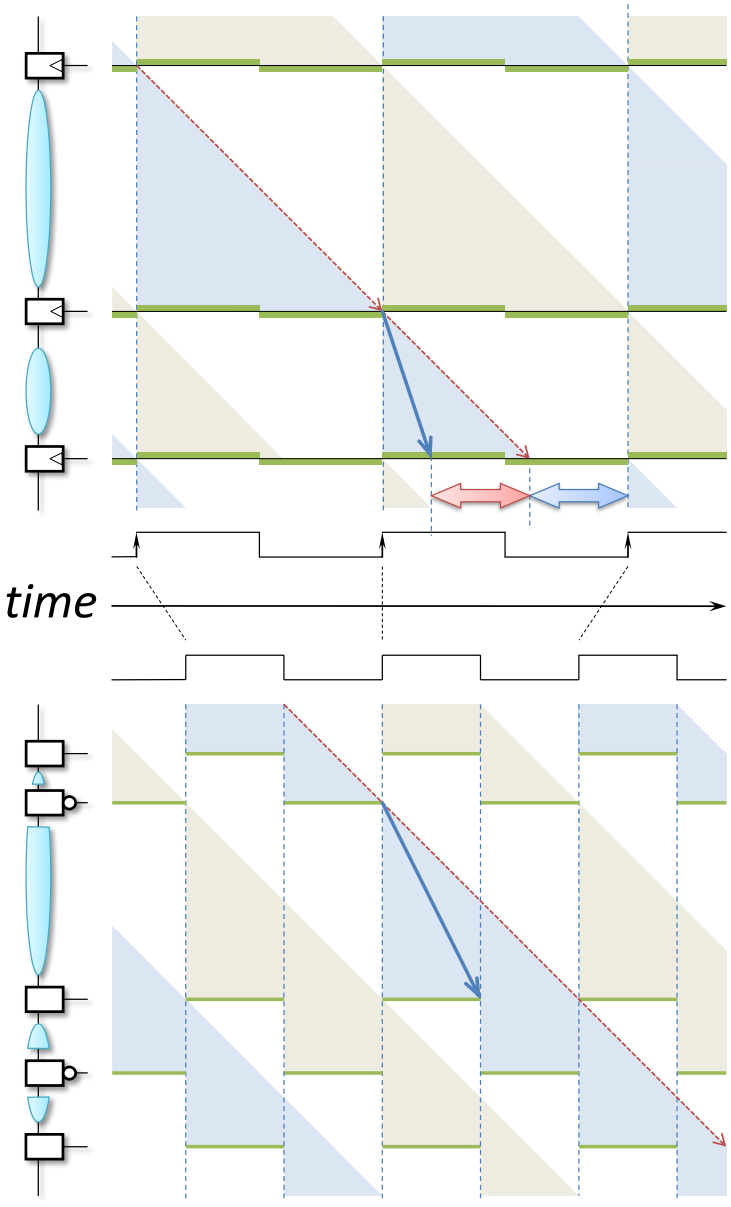


図 2.3: 静的タイム・ボローイング (STB)

第3章

タイミング故障検出・回復

本章では，タイミング故障検出・回復手法として，特に Razor について説明する．まず，3.1 節では，DVFS をタイミング故障検出・回復手法と組み合わせることで，どのようにばらつきの問題に対処するのかを説明する．次に，3.2 節は，Razor FF がタイミング故障を検出するためにどのように使用されるかを述べ，3.3 節が Razor FF のタイミング制約を要約する．

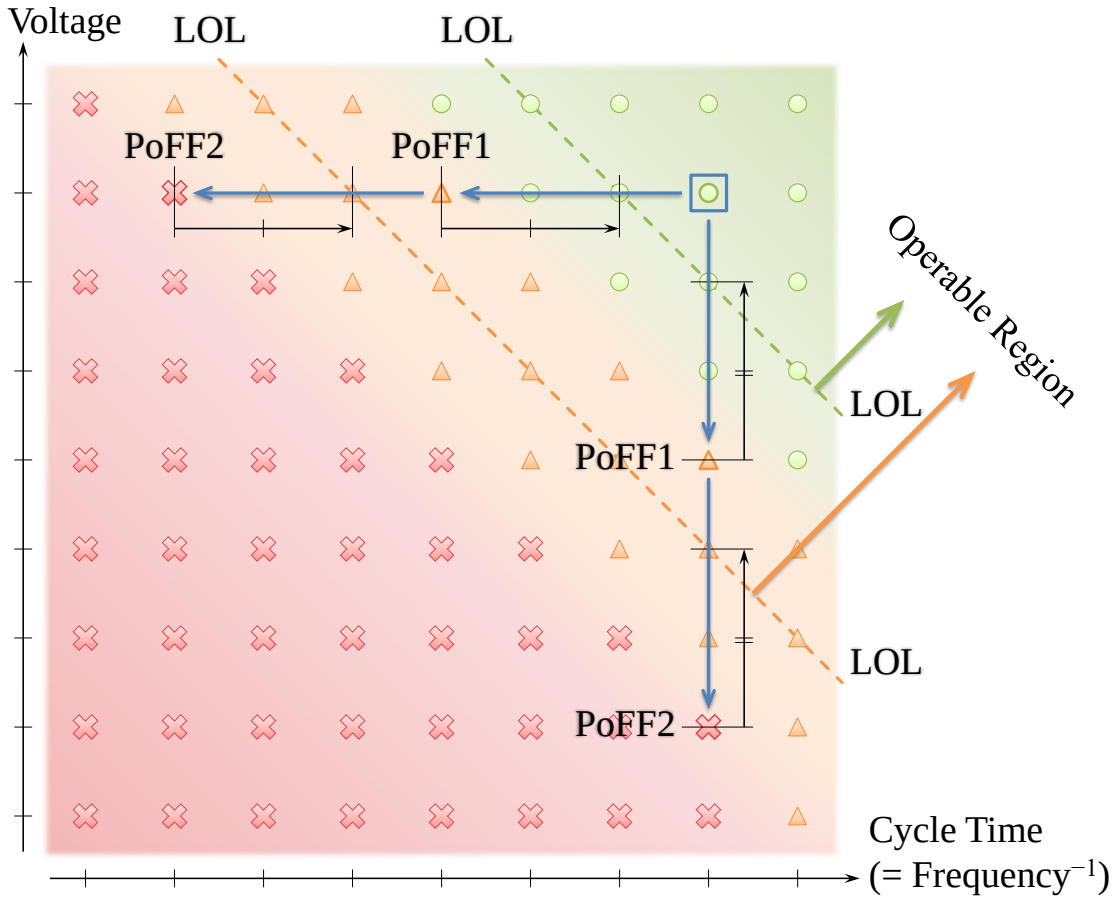
3.1 タイミング故障検出・回復と DVFS

タイミング故障検出技術は細粒度ばらつきのワースト・ケース設計を避けることができる．本節では，DVFS と組み合わせたタイミング故障検出がどのようにワースト・ケース設計を超えるかを述べる [4–6, 20]．タイミング故障を検出する技術については，3.2 節で述べる．

図 3.1 は shmoo プロットを示し，同じ回路の多数の個体のテスト結果をプロットする．グラフにおいて，垂直軸は電圧であり，水平軸はサイクル時間，すなわちクロック周波数の逆数である．同じ電圧およびサイクル・タイムであっても，個々の回路の遅延は，プロセスばらつきの存在のために変化し得る．個々の回路でタイミング故障が発生するかどうかは異なる．図 3.1 のマークの意味は次のとおりである：

○ PASS: タイミング故障がいずれのダイにおいても全く発生しない．

△ Timing Fault: タイミング故障が一度以上，一つ以上のダイ上で発生し，それらの



○ : PASS
△ : Timing Fault
× : Detection Miss

	Timing Fault Detection	Worst-Case Design
△ PoFF1	Fault	(Fault &) Failure
× PoFF2	Failure	

図 3.1: タイミング故障検出と DVFS による電圧/サイクル・タイムの改善

フォールトは全て検出される．

× Detection Miss: タイミング故障が一度以上，一つ以上のダイ上で発生し，一つ以上のダイでフォールトの検出に失敗する．

△ と × で示される点で，タイミング故障を引き起こす個体の割合は 0 ではない．ここでは，レートは以下で説明する *ORTF* とは異なる．レートはプロセスばらつきの影響を受けるが，ORTF はチップの動作の動的変動の影響を受ける．

タイミング故障検出なし回路の動作可能領域 図 3.1 では，回路が正方形のマーク(□)で示される電圧とサイクル・タイムのポイントで動作する場合，タイミング故障なしで動作する．この点から，電圧またはサイクル・タイムが DVFS により下げられると，最初の△ にてタイミング故障が発生する．この点が **PoFF1** である．PoFF1 はタイミング故障検出がない回路の最初の**故障**のポイントと呼ばれる．なぜなら，その故障は単一のタイミング故障によって引き起こされるからである．

回路の動作可能領域は，図の**上側**の点線で示されているように，PoFF1 からの余裕分だけ離れている下側動作限界（**lower operation limit: LOL**）の右上に示されており，これはワースト・ケース設計によって与えられる動的なばらつきへのマージン分だけ，PoFF1 とは離れている．タイミング故障検出**なし**回路の実際の動作点は通常 LOL から選択される．

タイミング故障検出あり回路の動作可能領域 対照的に，タイミング故障検出**あり**回路は PoFF1 を超えて動作可能領域を拡張することができる．PoFF1 から，電圧またはサイクル・タイムが低下すると，回路は最初の ×，**PoFF2** でタイミング故障を検出し損ねる．タイミング故障検出**あり**回路はこの時点で初めて故障する．

したがって，区別するために，最初の**故障 (fault)** と**故障 (failure)** のポイントをそれぞれ **PoFF1** と **PoFF2** と呼ぶ．タイミング故障検出**なし**回路の場合，最初の**故障 (fault)** のポイントは最初の**故障 (failure)** のポイントと等価である．

タイミング故障検出**あり**回路の動作可能領域は，LOL の右上であり，図の**下側**の点線によって示される．これは PoFF1 ではなく，PoFF2 から動的なばらつきに対するマージンの分だけ離れている．このマージンは，前述のタイミング故障検出**なし**回路のワースト・ケース設計と同じである．したがって，タイミング故障検出**あり**回路は，PoFF1 と PoFF2 との間の差によって動作可能領域を拡張する．

タイミング故障検出あり回路の動作点 タイミング故障検出なし回路とは異なり，タイミング故障検出の回路の動作点は LOL から必ずしも選択されない．

電圧またはサイクル・タイムが PoFF1 を超えて低下すると，ORTF は徐々に増加し，タイミング故障から回復するオーバーヘッドも徐々に増加する．したがって，十分に低い ORTF を保証するように動作点を選択すべきである．

その結果，ダイの動作点は，その特定の動作条件下でのダイの個々の部分回路の実際の遅延に基づいて決定される．たとえば，プロセスばらつきのためにダイの

クリティカルパスの遅延が幸いにも短くなったり，クリティカルパスが長時間アクティブ化されなかったりすると，ダイは LOL の近くで動作する．

特に，マージンが PoFF1 と PoFF2 の間の幅より大きい場合，動作点は LOL に設定される．この場合，PoFF1 は動作可能領域に含まれておらず，回路が LOL で動作していてもタイミング故障は決して発生しない．

マルチビットのタイミング故障　本節では，この手法は，(S)RAM を使用するとき，以下に説明するように，しばしばマルチビット・タイミング故障を引き起こす．

比較的小さな遅延を有する RAM のエントリ（行）に対する読み出し動作が継続する場合，デバイスは電圧またはサイクル・タイムを低下させる．次に，より長い遅延を有する別のエントリが読み取られる場合，電圧またはサイクル時間がそのエントリ内の複数のセルにとって低すぎるため，マルチビット・タイミング故障が生じる．最悪の場合，エントリのすべてのビットがタイミング故障を引き起こす可能性がある．

このマルチビット・タイミング故障は，検出に使用されるメソッドに影響する．第 4 章で述べる SRAM への提案手法はマルチビット・タイミング故障の発生に対応している．

3.2　Razor FF

本節では Razor [21, 22] について述べる．

3.2.1　Razor FF のタイミング故障検出

図 3.2 は Razor のパイプライン [22] のブロック・ダイアグラムを示す．1 つの Razor FF は，**メイン FF** と**シャドウ・ラッチ**によって構成される．それぞれを図 3.2 では **M** と **S** と表している．Razor FF では，シャドウ・ラッチには，メイン FF へのクロック clk より Δ だけ位相の遅れたクロック clk_d が供給される．その結果，メイン FF とシャドウ・ラッチで 2 回，入力 d のサンプリングを行うことになる．それらの値が異なっていれば，TF が検出され，エラー e がアサートされる．本稿では，シャドウ・ラッチとしてネガティブ・エッジトリガ FF を用いることでこれが実現されている．この Razor FF の TF 検出は事後的である、すなわち、タイミング故障が

検出された時点までに間違った値が次の段階で既に使用されていることに留意されたい。

3.2.2 Razor II におけるタイミング故障からの回復

タイミング故障からの回復は、**アーキテクチャ・ステート(AS)** の保護が基本である．本論文の文脈では、タイミング故障から保護するのは、整数および浮動小数点レジスタファイルだけでなく、**コントロール・ステータス・レジスタ (Control Status Registers: CSR)**，および L1D も AS に含まれる。

Razor を適用したプロセッサのパイプラインにおいて，AS は次のように保護される：

1. 図 3.2 に示すように、より長いパス遅延を持つパイプライン・ラッチは、TF を検出するために Razor FF に置き換えられる．

   エラーネットワークが追加されて、各 Razor FF の e 出力をパイプライン・ステージに沿って収集し、誤った結果による AS の更新を無効にする．

   3.2.3 節で詳述されている理由により、空の**スタビライズ・ステージ**が挿入される。

7.3.6 節で述べるように，プロセッサは、TF の影響がパイプラインから取り除かれた後、保護された AS から再実行することができる。

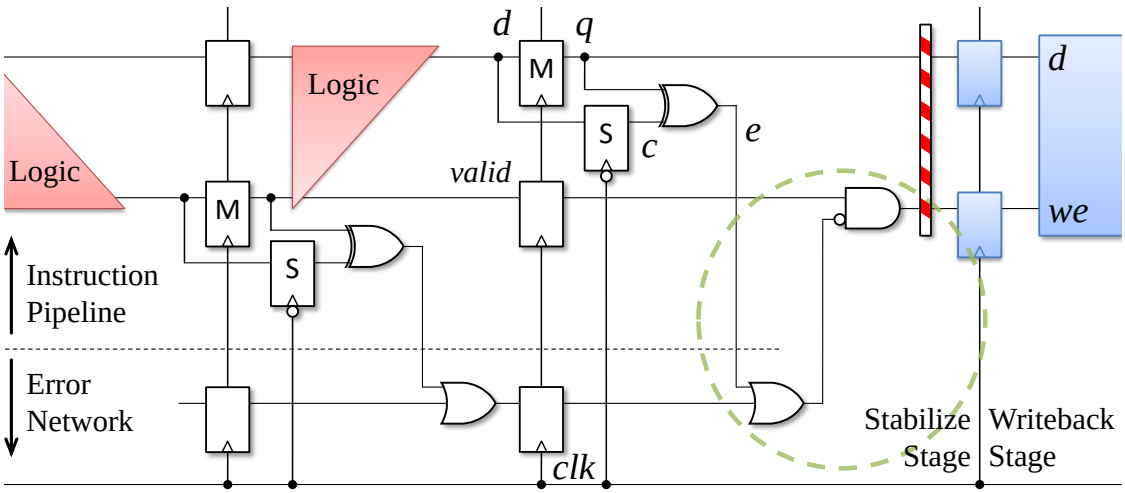


図 3.2: Razor FF のブロック・ダイアグラム

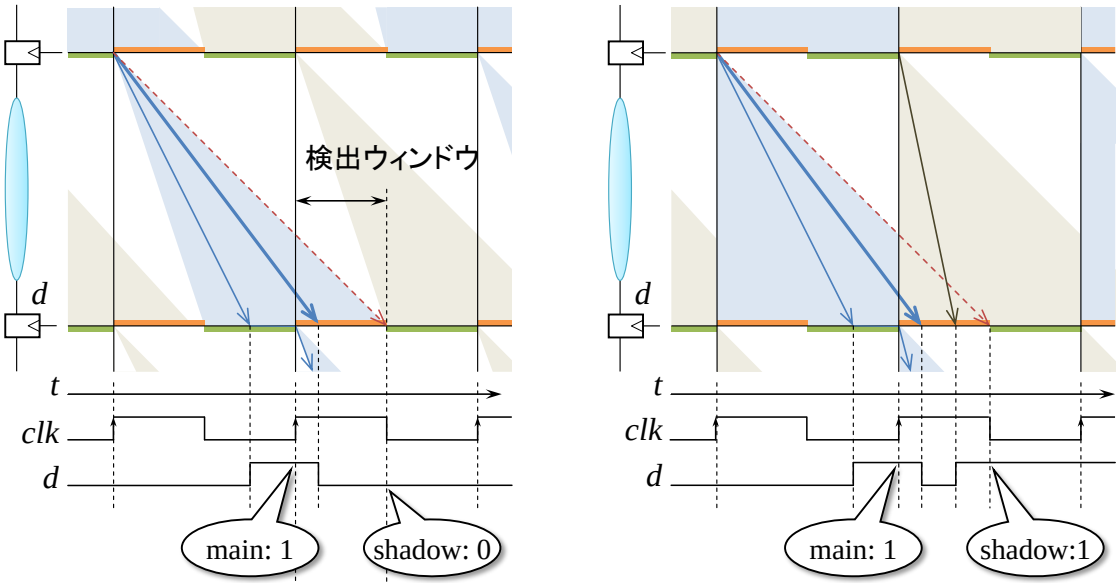


図 3.3: Razor のショート・パス問題

3.2.3　スタビライズ・ステージ

Razor FF の事後的なエラーに対応するために，空のスタビライズ・ステージが，ライトバック・ステージの前に必要とされる．

図 3.2 に示すように、ライトバック・ステージの開始 FF は Razor FF で置き換えることはできない．仮にそうした場合であっても、置き換えられた Razor FF が TF を検出するまでに，AS が誤って更新されてしまう．この意味では、これらの FF は、図の赤と白のストライプを持つバリアバーで示される**ポイント・オブ・ノー・リターン**である．有効な書き込み要求がこれらの FF に一度セットされると、AS は不可逆的に更新される．

これらの FF は Razor FF に置き換えることができないため、ライトバック・ステージの直前のスタビライズ・ステージでは、TF を引き起こす可能性のあるロジックを持つことができない．スタビライズ・ステージの唯一の役割は、図に破線の円で示すように、TF でこれらの FF にセットされようとする誤りを含む書き込み要求を無効にすることである．

3.2.4 Razor のショート・パス問題

クロック・スキューに起因するホールド・タイム違反など，ショート・パスが原因で遅延制約が満たされない問題を**ショート・パス問題**と呼ぶ．Razor には，Razor 特有のショート・パス問題がある．

図 3.3 のダイアグラムを用いて，Razor のショート・パス問題を説明する．あるサイクルにおいて，TF が発生しているとする．メイン FF のサンプリング時には 1 であるが，真の値は 0 である．シャドウ・ラッチが正しい値をサンプリングするためには，左のダイアグラムのように，ロジックのショート・パスを通った信号がシャドウ・ラッチのサンプリング・タイミングよりも後に到達しなければならない．こうなっていれば，メイン FF とシャドウ・ラッチの値が異なるため，正しく TF を検出することができる．一方で，仮に右のダイアグラムに示されているように，ショート・パスが存在している場合，あるフェーズにおいてショート・パスを通った信号が，前のフェーズの信号と「混ざる」．その結果，シャドウ・ラッチが真の値とは異なる値をサンプリングしてしまう．その結果，検出漏れ (false negative) が生じており，これは致命的である．当然，逆の誤検出 (false positive) も存在する．

このため Razor は，Razor 特有の最小遅延制約をもつ．図 3.3 では，シャドウ・ラッチのサンプリングを 0.5τ 遅らせているため，最小遅延制約は $0.5\tau/1$ステージ となる．前節と同様に，サイクル・タイムに対する検出ウィンドウの割合を α とすると，最小遅延制約は $\alpha\tau/1$ステージ となり，単相 FF 方式より $\alpha\tau$ だけ厳しくなる．ショート・パスに遅延素子を挿入するなどして，ロジックの最小遅延を $\alpha\tau$ 以上にする必要がある．

3.3 Razor FF のタイミング制約

図 2.2 (c) に，Razor FF のダイアグラムを示す．同図では，$\Delta = 0.5\tau$，すなわち，半周期遅れたクロックをシャドウ・ラッチに供給している．ダイアグラムでは，FF の下の濃さの異なる縦実線（橙色）が，TF 検出ウィンドウを表している．

クリティカル・パスの遅延に対応する 45°の破線が検出ウィンドウの下端までに到着するなら，TF が発生したとしても検出し，回復することができる．そのため，45°の破線矢印はジグザグとなる．TF 検出を行わない単相 FF や二相ラッチでは，45°の破線は一直線になっている（同図 (a)，(b)）．

TF 検出を行う方式では，このジグザグの分だけ，クリティカル・パス遅延を超えてサイクル・タイムを短縮することができる．サイクル・タイムに対する検出ウィンドウの割合を α とすると（図では $\alpha = 0.5$），最大遅延制約は $(1+\alpha)\tau/1$ステージとなり，単相 FF 方式より $\alpha\tau$ だけ改善される．

3.4　Razor FF の限界

Razor は，遅延が τ より長いパスが，チップのどこか 1 か所ででも活性化されると，TF となって回復のペナルティを被ることになる．

そのため実際には，TF の発生確率が十分に小さくなるようにする必要がある．すなわち，個々の個体の動作状況に合わせた実際のクリティカル・パス遅延にほぼ一致するようにサイクル・タイムを制御する．この場合，α を大きくする意味はないので，例えば 0.1 程度に設定する．

このことは，設計時に見積もったのではない，実際のクリティカル・パス遅延を基にサイクル・タイムを設定することを意味する．結局，実際の Razor の効果は，設計時に課せられるタイミング・マージンを削減するに留まる．

これに対して，第 5 章で詳述する DTB を可能にするクロッキング方式は，TF の発生確率自体を下げる効果を持つ．

第4章

SRAMのタイミング故障検出

4.1 本章の内容

第 3 章で述べたように，TF の発生自体は許容し，TF の検出・回復を行う手法として Razor FF が提案されている．しかし，Razor FF を含む従来の TF 検出技術は，その適用の対象としてスタティック・ロジックが暗黙のうちに想定されており，特にダイナミック・プリチャージ・ロジック (dynamic precharged logic) に対してそのまま適用することは考慮されていない．SRAM の読出しは通常ダイナミック・プリチャージ・ロジックとして実装されるため，Razor FF の対象外である．言うまでもなく，SRAM は LSI において欠くべからざる要素であり，SRAM の適用を考慮しないことで従来手法は網羅性を欠いている．

そこで本章では，Razor FF を SRAM に適用することについて考察し，新たな適用手法を提案する．本章で明らかにするように，SRAM の読出しに対してスタティック・ロジックと同様の適用を行うだけでは，V/F の改善を実現できない．本章における提案は SRAM のビットラインの状態に応じてプリチャージの制御を行うことで，より広い V/F での動作を実現する．本提案はメモリ・セルに変更を加えず，センス・アンプ周辺への変更のみで実現でき，回路面積をほとんど増加させない．

本章の構成は以下のようになる：4.2 節において Razor FF の SRAM 読み出し回路へのナイーブな適用の問題を論じる．4.3 節においてこの問題に対する提案を述べる．4.4 節ではこの提案の評価を述べる．

4.2　SRAM の読み出しのタイミング故障検出の問題

Razor では，その適用の対象としてスタティック・ロジックが暗黙のうちに想定されており，特に**ダイナミック・プリチャージ・ロジック** (dynamic precharged logic) に対してそのまま適用することはできない．このことは，特に SRAM で問題となる．SRAM の読出しは通常，ダイナミック・プリチャージ・ロジックとして実装されるため，Razor をそのまま適用することができない．SRAM は，今日の LSI において欠くべからざる要素であり，SRAM に適用できないことは Razor の重大な欠点であると言える．

本節では，何故 Razor のナイーブな適用が SRAM に対して困難であるかについて述べる．初めに，4.2.1 節と 4.2.2 節で一般的な SRAM の読み出し回路と，そのサイクルタイム制約について述べる．次に，4.2.3 節では，SRAM の読み出し回路に対する Razor のナイーブな適用の限界について述べる．

4.2.1　SRAM 読み出し回路

対象の SRAM　本章の目的はプロセッサコアに使用されているレジスタ・ファイルや L1D などの SRAM についてである．

ばらつきの増加に応じて，IBM や Intel などのプロセッサベンダは近年，そのプロセッサコア内の SRAM に，差動センスアンプ従来のダブルエンド・ビットラインの代わりに，ドミノセンスアンプのシングルエンド・ビットラインを採用している [23–25]．

したがって，本論文はシングルエンドビットラインの SRAM に注目する．他方については，シャドウ・センスアンプ [26] や，入江らの手法 [27] が提案されている．

読み出し回路の構成と動作

一般的に，SRAM の読出し動作は，ダイナミック・プリチャージ・ロジックによって実装される．

図 4.1 (上) に SRAM の構成を示す．SRAM 読み出し回路は，その入力がアドレス・ビットであり，その出力がそのアドレス・ビットによって指定されるエントリのデータである．列上の全てのメモリ・セルは一つのビットラインに接続され，そのうちの一つがワードライン・アサーションによって選択される．

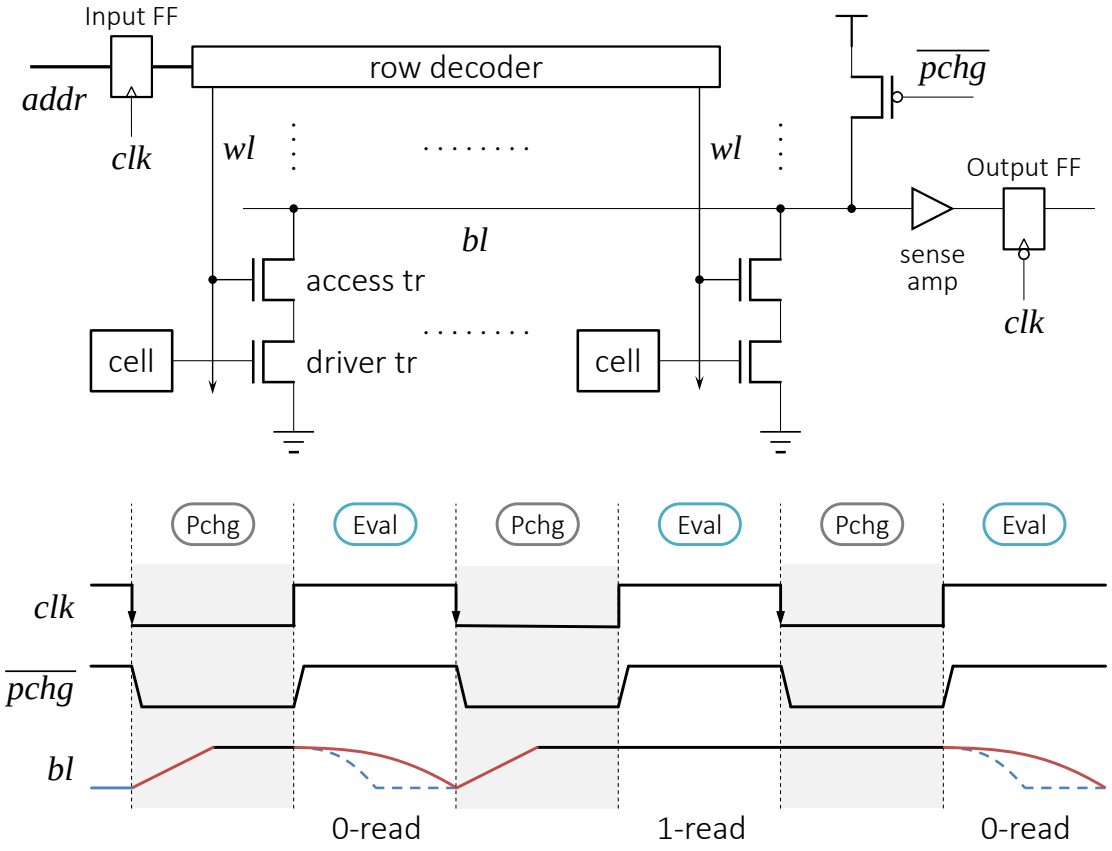


図 4.1: SRAM の読み出し回路と動作.

図 4.1 (下) に示すように，SRAM の読み出し動作はプリチャージと評価が交互に行われることで実現されている．本論文のタイミング・チャートでは，プリチャージ期間を影によって表現する．各期間における動作は次である：

プリチャージ期間:

プリチャージ pMOS トランジスタがオンになり，ビットライン*bl* が 1 にプリチャージされる．

評価期間:

アドレス・デコーダで選択されたワードラインがアサートされて，メモリ・セルの値に応じて，ドライバ・トランジスタがオンまたはオフになる．

0-read:

ドライバ・トランジスタがオンであれば，bl がディスチャージされて 0 になる．本論文では，このような読出しを **0-read** と呼ぶ．

1-read:

ドライバ・トランジスタがオフであれば，bl はディスチャージされずに 1 に保たれる．本論文では，このような読み出しを **1-read** と呼ぶ．

4.2.2　SRAM 読み出し回路のタイミング制約

SRAM では各期間の遅延に基づいてサイクルタイム制約が決定される．

評価遅延　1-read の遅延は，0 と考えることができる．なぜならば，bl は評価期間の最初から正しい値である 1 であるからである．しかしながら，実際は 1-read が行われるにせよ，読み出し回路は，ディスチャージのワースト・ケース遅延の時間後に bl がディスチャージされていないことを確認するまでは，その読み出しが 1-read であるかどうかを決定することはできない．したがって，総じては，**評価の遅延はディスチャージの遅延によって決定される．**

プリチャージ遅延　0-read 時のディスチャージの遅延は，メモリ・セルの大きさや RAM のエントリ数（ビットラインに接続されたメモリ・セルの数）から制約されるため，困難である．一方で，プリチャージの遅延は，プリチャージ pMOS トランジスタのサイズが比較的自由にできることから，ある程度の遅延削減が容易である．

以降，評価のワースト・ケースにおける遅延に対して，プリチャージのワースト・ケースにおける遅延が半分の場合を例として説明する．プリチャージの遅延が評価の遅延に対してどの程度であるかの影響については，4.3 節における提案手法の説明の際にまとめて考察する．

サイクルタイム制約　図 4.2 の上部は，4 種の SRAM 読み出し回路についての回路のダイアグラム（左）とタイミング・チャート（右）を示す．このタイミング・チャートは図 3.1 における最初の故障のポイントの電源電圧における動作に対応しており，これによりサイクルタイムの下限を示すことができる．

図 4.2 (a) は従来回路を示す．この図では，$\overline{pchg}$ は clk によって生成され，その前半・後半はそれぞれ評価・プリチャージ期間に割り当てられている．結果とし

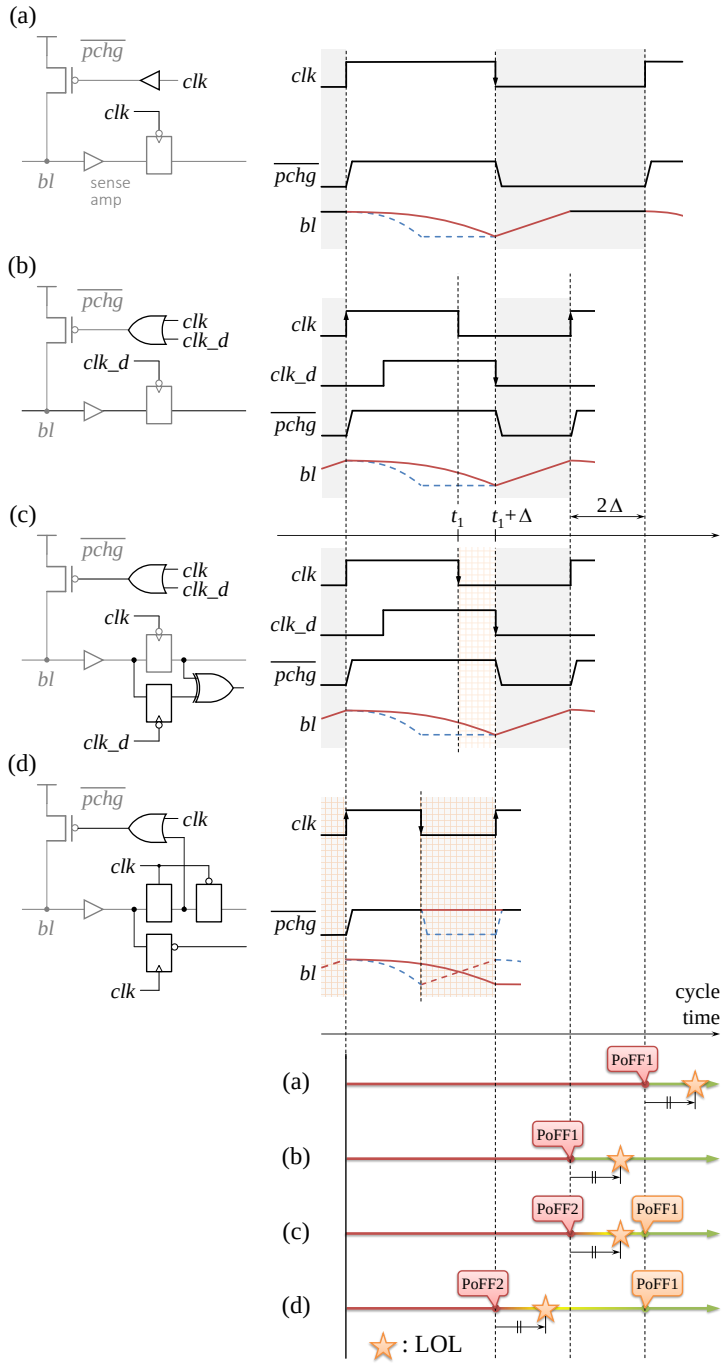


図 4.2: SRAM 読み出し回路の構成と動作の比較:
(a) 従来，(b) 遅延されたクロックに対して最適化，(c) Razor FF のナイーブな適用，
(d) 提案

て，プリチャージ期間はプリチャージが実際に終了した後にアイドル時間が生じている．

図 4.2 (b) は clk を Δ だけ遅らせたクロックである clk_d を用いて削減されたサイクルタイムをもつ回路を示している．この図では，出力 FF は clk ではなく clk_d に同期する．このタイミング･チャートでは，bl のワースト･ケースにおける遷移は clk ではなく clk_d のサンプリング・エッジに間に合わなければならない．

このような手法でサイクル・タイムを短縮する場合でも，サイクル・タイムは評価とプリチャージのワースト･ケース遅延の和が最小となり，それ以上は短縮されない．これは，SRAM には，FF のサンプリング時までに bl の遷移が間に合わなければならないという制約の他に，サイクル・タイムを制約する次の要因が存在するからである：

Precharge-after-evaluation 制約:

> プリチャージは評価が完了した後に開始しなければならない．そうでなければ，0-read において，プリチャージによって，bl は正しい値である 0 にディスチャージされない．

Evaluation-after-precharge 制約:

> 評価は，プリチャージが完了した後に開始しなければならない．そうでなければ，1-read において，ダイナミック･プリチャージ･ロジックの評価は 1 から 0 に一方向にしか遷移しないため，bl は 0 から 1 に遷移することができない．

これらの制約を満たすために，サイクル・タイムは評価とプリチャージのワースト･ケース遅延の和よりも長くなければならない．図 4.2 (b) の回路は Δ を調整することによってこの下限を実現する．

PoFF のサイクルタイム制約　図 4.2 の下部はこれらの回路の動作可能な領域を示す．これは適当な電源電圧に対する図 3.1 のポイントに対応する．

この図では，lower operating limit (LOL) を☆マークで示す．3.1 節で述べたように，TF 検出なしの回路については，LOL は PoFF1，つまり最初のフォールトから，マージン分だけ離れている．同図に示されているように，(b) の回路の PoFF1 と LOL は (a) の回路のそれに対して 2Δ だけ離れて平行に削減される．

4.2.3 SRAM 読み出し回路への Razor のナイーブな適用の問題

この節では，SRAM 読み出し回路への Razor のナイーブな適用について述べ，そのサイクル・タイム短縮効果における問題について明らかにする．

構成と動作 図 4.2 (c) は SRAM 読み出し回路への Razor FF のナイーブな適用について示す．この図に示されるように，図 4.2 (b) の出力 FF が，同図 (c) においては Razor FF に置き換えられている．

このタイミングチャートの点線に示すように，ティピカル・ケースにおいては，メイン FF のサンプリングエッジである時刻 t_1 の前に bl は最終的に正しい値に遷移を完了する．このケースでは，正しい値である 0 がメイン FF によってサンプルされ，次のステージに伝達される．

同図の実線に示すように，ワースト・ケースにおいては，時刻 t_1 に bl が間に合わず，メイン FF は間違った値である 1 を次のステージに伝える．しかし，bl は検出期間，つまり時刻 t_1 から $t_1 + \Delta$ において正しい値である 0 に変化するため，3.2 節に示したようにこのタイミング故障は正しく検出される．

サイクル・タイム制約 3.2 節で述べたように，Razor FF をスタティック・ロジックに適用したとき，Δ を増加させることによって，サイクル・タイムを削減することができる．

一方で，以下で述べる理由により，Razor FF を SRAM にナイーブに適用したときは，Δ を増加させたとしても，サイクル・タイムを評価とプリチャージのワースト・ケースにおける遅延の和よりも削減することができない．

この回路は図 4.2 (b) で述べたのと同様に precharge-after-evaluation 制約と evaluation-after-precharge 制約を満たす必要がある．(b) では，メイン FF は clk_d のサンプリングエッジにおいて正しい値をサンプルしなければならず，(c) では，シャドウ FF がそうでなければならない．

このように，図 4.2 の下部で示すように，(b) の PoFF1 は (c) の PoFF2 と等しく，それらの動作領域も等しい．

ナイーブな適用の限られたメリット したがって，次の理由から，この (c) に示されるナイーブな適用は，(b) に対してメリットがない．

まず，上記に述べたように，動作領域が等価である．一方で，実際の動作ポイントについては互いにわずかに異なる．3.1 節に述べたように，タイミング故障検出を備えた回路の動作ポイントは，タイミング故障の発生率に応じて PoFF1 以下のポイントに落ち着く．このケースでは，マージンの幅が PoFF1 と PoFF2 間の幅よりも短いかどうかに応じて，実際の動作ポイントは次のように異なる：

- もしマージンがより小さければ，図 4.2 の下部に示すように, 動作ポイントは LOL と PoFF1 との間のポイントに落ち着く．したがって，(c) のサイクル·タイムは期待に反して (b) 以上になってしまう．
- この図と異なり，このマージンがより大きければ, (b) と (c) の LOL は (c) の PoFF1 の右側にシフトする．結果として，(c) の動作ポイントもシフトされた LOL にフィックスされる．

まとめると，(c) に示したような Razor FF の SRAM へのナイーブな適用によるサイクル·タイムは，(b) で示したような遅延されたクロックを使用して回路を最適化した場合以上にしかならない．(c) においても，同じだけ遅延されたクロックが必要とされることに注意する．

(c) の (b) に対する少ない利点は，パイプライン・ステージの下流のタイミングデザインが容易であることである．図 4.2 (b) の回路は遅延されたクロック clk_d の立下りエッジにおいて結果を出力する一方で，(c) の回路は (a) と同様に，遅延されていないクロック clk の立下りエッジにおいて結果を出力する．これは次パイプライン・ステージのタイミング制約を緩和するが，SRAM 読み出しステージのタイミング故障のリスクがある．

4.3　提案：SRAM のためのタイミング故障検出

本節では，SRAM に対する新たな Razor FF の適用手法を提案する．提案手法は，メインのサンプリング時点でのビットラインの状態に応じてプリチャージを制御するものである．提案によって，4.2.3 節で述べた問題が解消され，サイクル·タイムを評価とプリチャージのワースト·ケース遅延の和よりも短縮することができる．

初めに 4.3.1 節でアプローチについて述べ，4.3.2 節で提案手法の回路構成とその動作について述べ，TF が正しく検出されることを確認する．4.3.3 節で提案回路の

詳細なデザインについて述べる．4.3.4 節では提案手法適用後のサイクル・タイム制約について述べる．最後に 4.3.5 節ではオーバーヘッドについて議論する．

4.3.1 プリチャージと検出期間のオーバーラップ

本論文で提案する方式のアプローチは，評価とプリチャージの期間のオーバーラップによるものである．このオーバーラップされた期間を**検出/プリチャージ期間**と呼ぶ．

このオーバーラップを可能にするために，メイン FF のサンプリングエッジにおける bl の値に応じた**コンディショナル・プリチャージ**を導入する．もし bl の値が 0 である場合は，検出/プリチャージ期間中は通常と同様に bl のプリチャージを行う．一方で，bl の値が 1 である場合は，検出/プリチャージ期間中は bl のプリチャージを行わず，タイミング故障の検出を行う．

4.2.3 節で述べたように，ナイーヴな適用では，検出期間とプリチャージ期間との重ね合わせは，ダイナミック・プリチャージ・ロジックとしての動作を保証するための制約を破ってしまう．一方で，提案手法では 0-read が遅れるときは，メイン FF のサンプリング時点ではビットラインが 1 であるから， プリチャージは行われない．したがって，検出期間の間にビットラインが 1 から 0 に遷移することができ，シャドウ FF に正しい値がサンプリングされ，TF 検出の正しさが保証される．

4.3.2 基本構成と動作

コンディショナル・プリチャージはビットラインごとに一つの追加ゲートで実現できる．

基本構成　図 4.3 は，提案回路の詳細なブロックダイアグラムを示す．この回路と図 4.2 (c) に述べたナイーブな適用との違いは次である：

1. エラー信号 e のためのロジックは XOR から単純化される．
2. ビットラインに対してキーパが付与される．
3. 主要な違いとして，2 入力のゲートがコンディショナル・プリチャージのために $\overline{pchg}$ を生成するために付加される．このゲートの 2 つの入力は，clk と $\overline{pe}$

(precharge enable) である．$\overline{pe}$ を得るために，メインとシャドウの FF は，マスター/スレイブ・ラッチの連なりとして描かれている．機能的には，$\overline{pe}$ は q と等価であり，つまりメイン FF のサンプル後の値である．$\overline{pe}$ が q よりも好

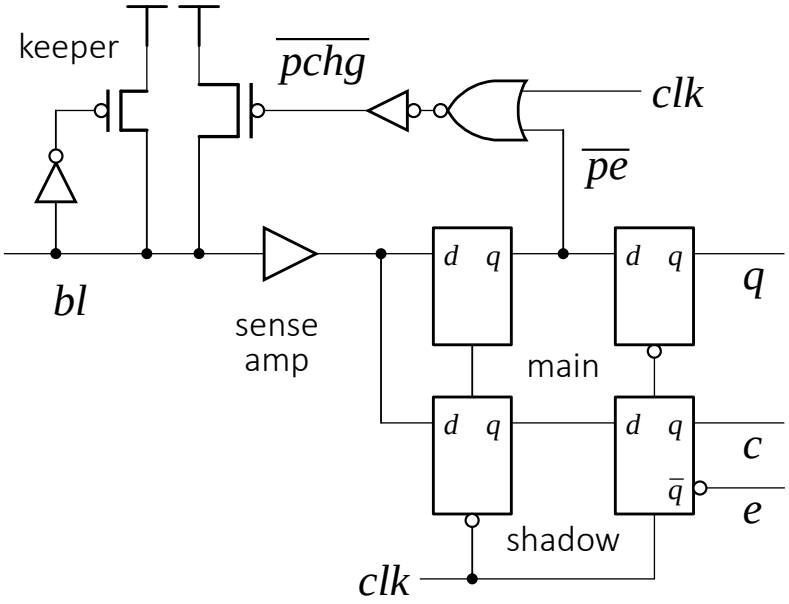
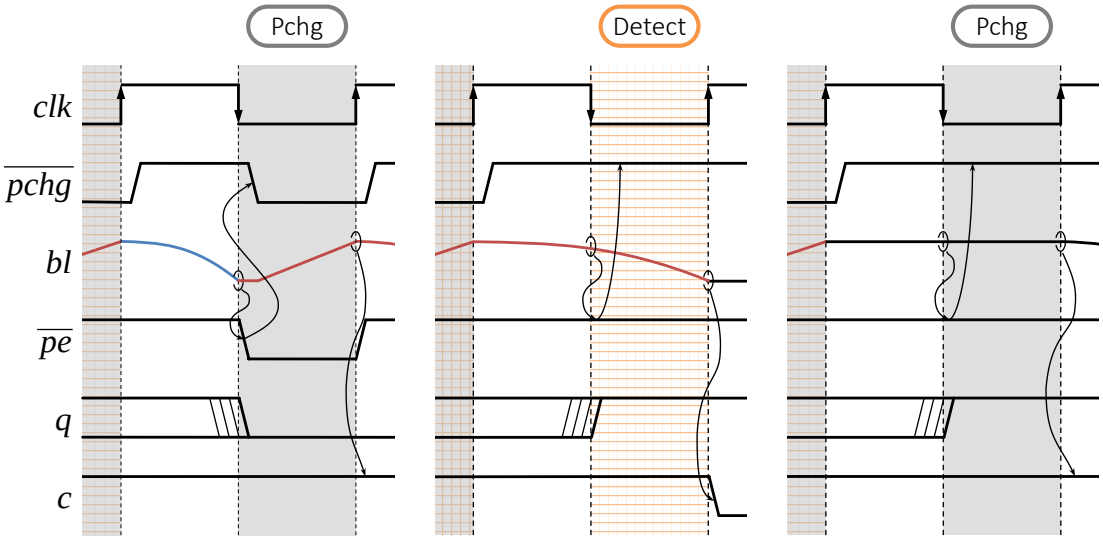


図 4.3: 提案回路のブロックダイアグラム



(1) 0-read w/o delay　(2) 0-read w/ delay　(3) 1-read

図 4.4: タイミング・チャート

ましい理由は後述する．

遅延されたクロックの排除 図 4.3 で示すように，プリチャージの遅延はサイクルタイムの半分以下であるので，メインとシャドウの FF に逆相の FF を利用でき，遅延されたクロックは必要としない．この場合，提案回路のオーバーヘッドはビットラインごとの 2 入力のゲートのみである．

提案 SRAM の動作 検出/プリチャージ期間がオーバーラップしていても，提案回路は正しく動作する．

4.2.1 節に述べたように，1-read の遅延は 0 と考えることができる．したがって，考慮すべきすべてのケースは，遅れのない 0-read と，遅れた 0-read と，1-read に分類できる．それぞれを図 4.4 (1 to 3) に示す．

(1) 遅れのない 0-read: bl は遅延なくディスチャージされ，メイン FF のサンプリング・エッジの前に正しい値である 0 に落ち着く．次に，サンプルされた値 0 は $\overline{pe}$ に反映され，プリチャージ pMOS トランジスタはオンにされる．

結果として，bl は従来の回路と同様にプリチャージされる．

(2) 遅れた 0-read: bl はメイン FF のサンプリング・エッジにおいてまだ十分にディスチャージがなされず，この遅れによって，メイン FF は間違った値である 1 をサンプルする．次に，サンプルされた値 1 は $\overline{pe}$ に反映され，プリチャージ pMOS トランジスタはオフのままに保たれる．

結果として，bl のディスチャージは検出/プリチャージ期間において継続され，シャドウ FF は検出/プリチャージ期間の終わりにおいて正しい値である 0 をサンプルする．この値 0 は，メイン FF によってサンプルされた 1 とは異なっているので，エラー信号 e がアサートされる．

(3) 1-read: bl はディスチャージされず，1 のまま保たれる．したがって，メイン FF は 1-read の正しい値である 1 をサンプルする．次に，サンプルされた値である 1 が $\overline{pe}$ に反映され，プリチャージ pMOS トランジスタはオフのままに保たれる．ここまでの振る舞いは (2)，つまり遅れた 0-read の場合と同じである．

一方で，この場合は，bl がプリチャージなしであっても 1 のまま保たれ，シャドウ FF は検出/プリチャージ期間の終わりにおいて正しい値である 1 をサン

プルする．この場合は，値 1 はメイン FF によってサンプルされた値である 1 と等しいので，e はアサートされない．

(2) 遅れた 0-read におけるプリチャージの抑制　(2) のケース，つまり遅れのある 0-read において，bl はタイミング故障を検出するためにプリチャージされず，次のクロック・サイクルにおける評価は（もしそれが 1-read ならば）正しく行われることはできない．言い換えると，次のクロック・サイクルの evaluation-after-precharge 制約は満たされていない．これがコンディショナル・プリチャージによって引き起こると考えられる唯一の問題であるように一見考えられる．

しかし，これは無意味な制約となっている．なぜならば，このクロック・サイクルにおいてタイミング故障が検出されているためである．タイミング故障検出・回復機構をもつ回路は一般的に，任意のクロック・サイクルにおいてタイミング故障が検出された場合，それ以後のクロック・サイクルにおける動作は同じ順序で再実行される必要がある．したがって，次のクロック・サイクルの読み出し動作は，このクロック・サイクルの読み出し動作が再実行された後に行われるのだから，直後に正しく実行される必要はない．

基本的にタイミング故障検出・回復機構をもつプロセッサは，タイミング故障を起こした命令とそれより上流の命令は全てキャンセルされ，再実行される [5,28,29]．

4.3.3　デザインの詳細

プリチャージ・イネーブル・ロジック　FF 全体とそのマスターラッチのサンプルタイミングは同じである．したがって，メイン FF の 2 つの出力，すなわち q および $\overline{pe}$ は，サンプリングエッジ後の最初の半クロック・サイクルについて同じ値を有する．後半のクロック・サイクルでは，q は同じ値を維持するが，$\overline{pe}$ は bl に透過的である．

前半のクロック・サイクルでのみ $\overline{pchg}$ を制御する必要があるため，$\overline{pchg}$ の制御に関して，q と $\overline{pe}$ は機能的に等価である．しかし，論理設計の観点からは，$\overline{pe}$ が q よりも好ましい．

図 4.5 は，$\overline{pchg}$ が (1)q によって，(2)$\overline{pe}$ によってそれぞれ制御されている場合に，遅れのない 0-read の後に遅れのある 0-read があるタイミング･チャートを示す．これらのタイミング･チャートは次のように説明される：

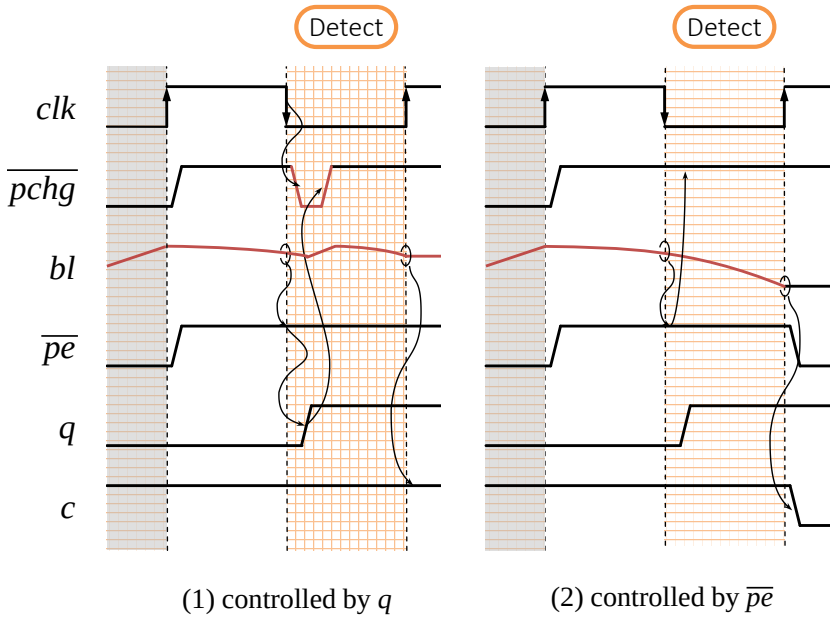


図 4.5: ハザード

(1）q による 前のクロック・サイクルにおいて 0-read が正常に実行されたため，q は立下りエッジにおいて 0 である．その後，メイン FF は立ち下がりエッジで遅れた値である 1 をサンプリングする．従って，q は，立ち下がりエッジの直後に 0 から 1 に変化する．

結果として，メイン FF のクロック・トゥ・データ遅延の大きさ分だけ，$\overline{pchg}$ にハザードが生成される．

すると，プリチャージ pMOS トランジスタが瞬間的にオンし，bl のディスチャージが妨げられる．シャドウ FF は誤った値である 1 をサンプリングして検出ミスを招く可能性がある．

(2)$\overline{pe}$ による $\overline{pe}$ は立ち下がりエッジの半クロック・サイクル前に 1 に変化し，立ち下がりエッジでは確かに 1 であるため，ハザードは生じない．これは，マスターラッチがこの半クロック・サイクルでプリチャージされた bl に対して透過的であるためである．

q によって引き起こされるハザードは，4.4 節で述べる SPICE シミュレーションで再現される．

エラー・ロジック　遅延のない 0-read（4.3 節 (1)）において，c はシャドウ FF のサンプリング・エッジで，タイミング故障によってではなく，プリチャージによって 1 になる．したがって，この値の組み合わせをエラーとみなしてはならない．

表 4.1 は，エラー・ロジックの真理値表を示す．その結果，エラー・ロジックは，q の組み合わせロジックを必要とせずに，$e = \bar{c}$ に簡略化される．要約すると，シャドウ FF が遅れた 0-read をサンプリングする場合に限り，e がアサートされる．

キーパ　1-read（図 4.4 (3)）では，ビットラインが 1 のままであるため，プリチャージは不要である．しかしながら，連続した 1-read があるとき，漏れ電流のためにビットラインが 0 に遷移する可能性があることに留意すべきである。

このシチュエーションに対処するために SRAM 読み出し回路以外の通常のダイナミック･プリチャージ･ロジックと同様のビットラインキーパを組み込む．

一般的に，キーパ pMOS トランジスタは，それが小さな漏れ電流を補正するためだけに必要とされるために，非常に小さくデザインすることができる．したがって，このディスチャージの速度に与える影響は些細である．4.5 節で述べる評価結果は，このキーパの負の影響も含んでいる．

4.3.4　サイクルタイム制約

提案手法を適用した場合の SRAM のサイクル・タイム制約について図 4.2 (d) に示す．図 4.2 (d) のように，プリチャージの遅延は評価の遅延の半分であり，逆相の FF がメインとシャドウの FF に用いられる．そして後半のクロック・サイクルが検出/プリチャージ期間に割り当てられる．

表 4.1: エラー・ロジックの真理値表

Case	q	c	e
—	0	0	
(1) 0-read w/o delay	0	1	0
(3) 1-read	1	1	0
(2) 0-read w/ delay	1	0	1

bl のワースト・ケースにおける遷移は，シャドウ FF のサンプリング・エッジに間に合わなければならないので，サイクル・タイムはディスチャージのワースト・ケース遅延にまで削減される．この図に示すように，提案のサイクル・タイムは，従来 (a) の 1/2，遅延されたクロックに最適化された (b) と Razor FF のナイーブな適用である (c) の 2/3 である．

これにより，図の下側に示すように、操作可能領域が拡張されます。 本提案の PoFF1 は（a)、(c）の回路と同等であるが、PoFF2 は他の回路と比較して最も左にある。

4.3.5 オーバーヘッド

3.2 節と 4.3.3 節に述べたように，提案では，ベースとなる回路に対して, 出力側 FF に対して 1 つのシャドウ FF と，各ビットラインごとに 1 つのキーパが面積，消費電力オーバーヘッドを生じさせる．しかし，本論文で想定しているレジスタ・ファイルや L1C に用いられる RAM の全体の面積に比べると，付加されるシャドウ FF やキーパの回路面積は極めて小さい．追加される FF の面積の計算を行ってこれを評価したところ，メモリセルのゲート幅よりも何倍か大きいスタンダードセルライブラリにおける FF を想定して場合でさえ，128 エントリの RAM に対してオーバーヘッドは 4.0% 以下である．一方で，4.3.4 節で述べたように本手法によるサイクルタイムの削減幅は従来回路に対して 1/2 である．したがって，オーバーヘッドを考慮しても周波数や消費電力削減の効果が見込まれる．

4.4 SRAM のタイミング故障検出 の評価

この節では，SPICE による提案の評価について説明する．評価においては，プロセスばらつきの効果を含めるためにモンテカルロ・シミュレーションを行った．まず，4.4.1 節はシミュレーションの環境と基本条件を述べる．4.4.2 節は，提案された回路の動作をどのように検証したかを述べる．4.4.3 節は，プリチャージ信号にどのようにハザードを再現したかを述べる．最後に，4.4.4 節は提案の動作可能領域を推定する，

表 4.2: 評価環境

Circuit and Layout Editor	Virtuoso Version IC6.1.5_ISR15
Layout Parametric Extraction	Calibre xACT3D Version 2012.3.31_26
Simulation	HSPICE Version H-2013.03
Technology Library	FreePDK45 [30]

4.4.1　評価環境と基本的な条件

表 4.2 は本評価で利用したソフトウェアとテクノロジ・ライブラリを示す.

図 4.6 は,SPICE シミュレーションに使用される回路を示している．これは 128 エントリの 12 ポート・レジスタ・ファイルに基づいている．ローカルビットライン(lbl)あたり 8 個のセルがあり，NAND は 2 個のローカルビットラインを $lbl2$ にマージしてグローバルビットライン(gbl) を駆動する．gbl あたり 8 個の NAND があり，INV は gbl を $gbl2$ に増幅し，lbl と gbl の配線容量はそれぞれ 4.6fF と 50fF である． このシミュレーションで使用されるトランジスタのタイプは，FreePDK45 の VTG と VTL である．VTG は一般的であり，VTL は低い閾値電圧を有する．

本シミュレーションの基本条件は次のようである: 温度は 55°C，電源電圧は 1.1V，サイクル・タイムは 800ps である．

モンテカルロ・シミュレーションの 1 セットあたりの試行回数は $10,000$ 回とした．MOS トランジスタの閾値電圧は，FreePDK45 のベース MOS トランジスタモデルの閾値電圧にランダム値を加えることにより，試行ごとにランダムに変化させた．ランダム値は，σ/μ を 10% に設定した正規分布を使用して生成された．これは ITRS よりも厳しい条件である (7%)．このランダム値は，WID の変動に対応する．本評価では，ベース MOS トランジスタのグレードを変えることによって D2D 変動を表現した．FreePDK45 のベース MOS トランジスタのティピカルとワースト・ケースダイ用の 2 つのグレード，すなわちノミナルおよび SS-コーナー MOS モデルを使用した．

4.4.2　検証

手法　次の動作をシミュレートした：

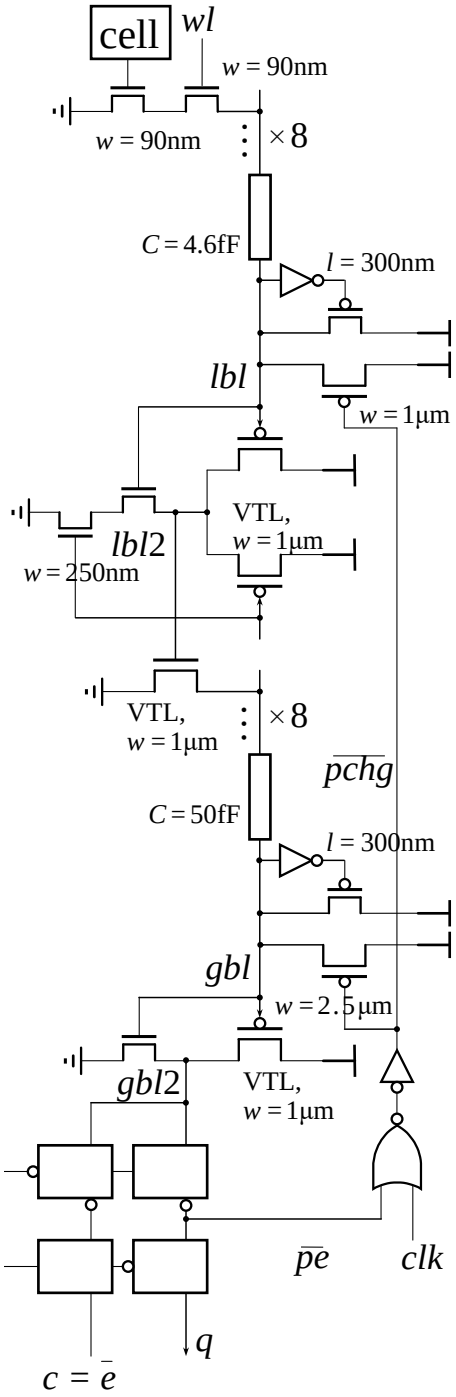


図 4.6: シミュレーション用回路: 特記のない限り，デバイスタイプは VTG，ゲート幅（w）は 90nm，ゲート長（l）は 50nm

1. 2 つのセル ($cell0$ and $cell1$) に異なる値を書く．
2. $cell0$，$cell0$，$cell1$，$cell1$，$cell1$ の順で 2 つのセルを読み取ることで，2 クロック・サイクルですべてのアクセスパターンをカバーする．
3. サイクル・タイムを 300ps に設定し，操作 1 と 2 を繰り返す．
4. 電源電圧を 0.6V に設定し，操作 1 と操作 2 を繰り返す．

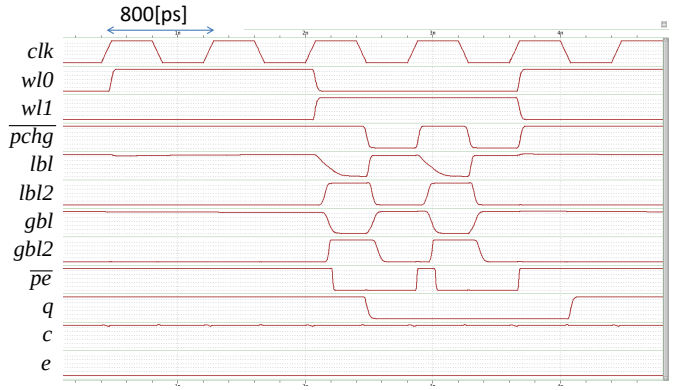
結果　図 4.7 は回路内の信号の波形を示す．

図 4.7a は，電源電圧が 1.1V, サイクル・タイム 800ps の条件下での動作波形を示している．最初の 2 サイクルでは $cell0$ が読みだされるが，これは 1-read であるため bl はディスチャージされない．このとき $\overline{pchg}$ がアサートされず，プリチャージが抑制されている．続く 2 サイクルでは $cell1$ が読みだされ，0-read であるため bl がディスチャージされる．このときは $\overline{pchg}$ がアサートされ，それぞれプリチャージが行われている．続くサイクルでは再び $cell0$ が読みだされ，1-read であるため bl はディスチャージされない．

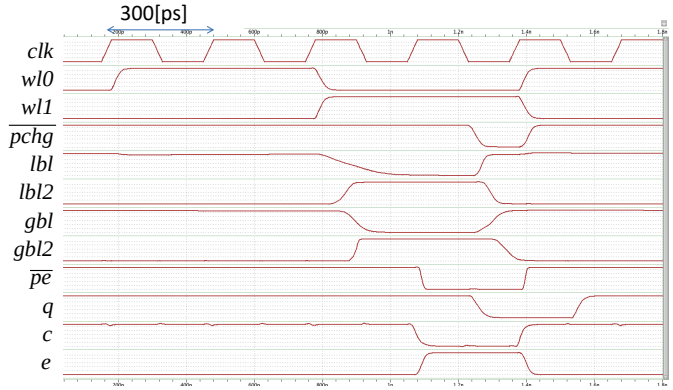
次に，図 4.7b は，電源電圧が 1.1V, サイクル・タイム 300ps の条件下での動作波形を示している．最初の 2 サイクルでは 1-read が行われるが，先ほどと変わらず適切に動作する．続くサイクルでは 0-read が行われるが，bl のディスチャージがメインのサンプリング・タイミングまでに十分になされず，q が 1 となって，タイミング故障が発生している．このときは $\overline{pchg}$ が抑制され，bl のディスチャージが継続する．そしてシャドウのサンプリング時までにディスチャージが十分になされて，c が 0 となって，$error$ が 1 となる．したがってタイミング故障発生時に適切にそれを検出することができている．以降のサイクルの振舞いについては，4.3.2 節で既に述べたように，実際はタイミング故障からの回復処理を行うことになるため，この評価では特に意味を持たない．

次に，図 4.7c は，電源電圧が 0.6V, サイクル・タイム 800ps の条件下での動作波形を示している．動作に関しては先ほどと変わらないため，省略する．このケースにおいてもタイミング故障を検出することができている．

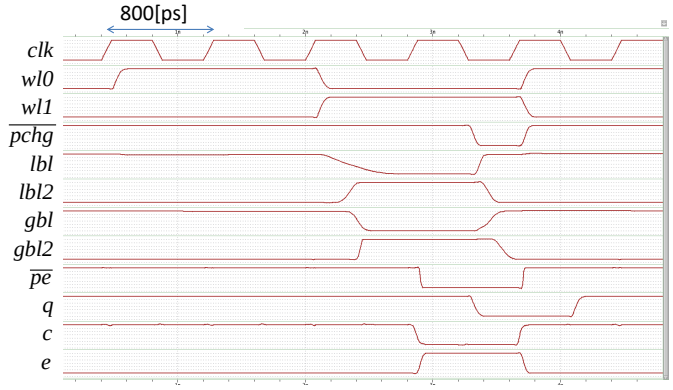
これらのシミュレーションに基づいて，提案された回路が正しく動作することを確認した．



(a) サイクル・タイム 800ps, 電源電圧 1.1V（タイミング故障が発生しない）



(b) サイクル・タイム 300ps, 電源電圧 1.1V（タイミング故障が発生する）



(c) サイクル・タイム 800ps, 電源電圧 0.6V（タイミング故障が発生する）

図 4.7: 提案回路内の信号の波形

4.4.3　ハザードの再現

手法　4.3.3 節で説明したように，$\overline{pchg}$ が q によって制御されている場合，$\overline{pchg}$ 上にハザードが生じる可能性がある．本評価ではモンテカルロ・シミュレーションのセットを用いてハザードを再現した．試行のために，サイクル・タイム 450ps，電圧 0.9V という条件の下に $\overline{pchg}$ が q によって制御された回路を実行した．シミュレーションにはティピカル MOS トランジスタモデルを使用した．

結果　図 4.8 a は，回路がハザードを起こした場合の信号の波形を示している．これと比較して，$\overline{pchg}$ が $\overline{pe}$ で制御されている回路の動作を図 4.8 b に示す．両方の回路は，閾値電圧の同様のパターンを有する．

図 4.8 a では，0-read は遅れ，q は clk の立ち下がり後に 0 から 1 に変化した．しかし，遷移には長い時間がかかり，clk の立下りの十分後に完了した．結果として，$\overline{pchg}$ にハザードが発生し，lbl と gbl がプリチャージされた．$\overline{pchg}$ が 1 に戻った後，0-read の遷移が再開された．しかし，遷移はシャドウ FF でサンプリングする前に完了できず，c と e はそれぞれ 1 と 0 にとどまっていた．その結果，回路はタイミング故障を検出しなかった．

図 4.8 b では，回路はタイミング故障を検出することができた．この結果は，論理設計の視点から $\overline{pe}$ が q よりも好ましいことを示している．

4.4.4　動作可能領域の評価

本節では，プロセスばらつきの存在下で提案回路による動作可能領域の拡大を評価する．

評価手法　0-read のモンテカルロ・シミュレーションを 2 回行った．D2D のばらつきの影響を表現するために，ティピカルな MOS トランジスタとワーストな MOS トランジスタモデルを使用した．0.025V ステップで電圧を 0.525V から 1.2V まで増加させ，25ps ステップで 100ps から 1,000ps までのサイクル・タイムを増加させることにより，上記の一連のシミュレーションを実行した．

結果　図 4.9 は，シミュレーションによって生成されたシュムプロットを示している．a は典型的ケースのダイを示し，b は最悪ケースのダイをそれぞれ示している．

(a) q によって制御

(b) $\overline{pe}$ によって制御

図 4.8: ハザード: a は q によって制御され，b は $\overline{pe}$ によって制御される．

図中の色の強さは，タイミング故障が発生する個体の割合を示す．3.1 節で述べたように，このレートは ORTF とは異なる．破線と実線は，PoFF1 と PoFF2 を示している．ただし，0 の色の強さは，PoFF1 を強調するために手動で割り当てている．

この図にグラデーションが現れる領域では，タイミング故障が起きる個体の割合は 0 から 1 に段階的に変化する．電圧が低下すると，領域の水平幅が拡大すること

が分かる．これは，低い電源電圧では，MOS トランジスタの閾値電圧の遅延変動に対する影響が大きいためである．

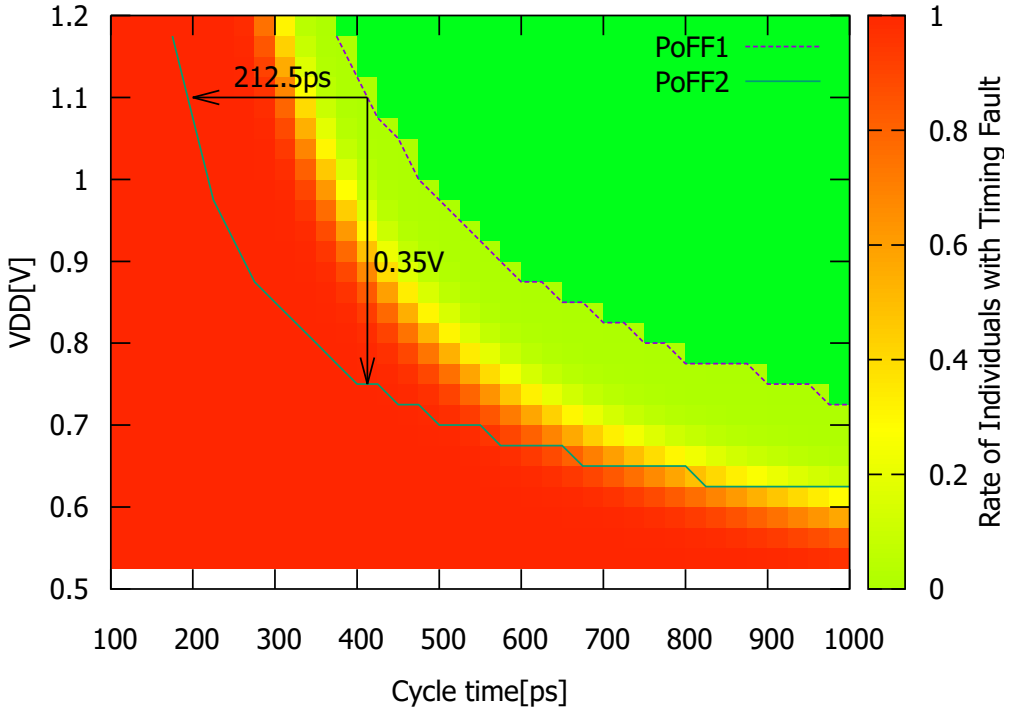
3.1 節で説明したように，提案回路は PoFF1 と PoFF2 の差で動作可能領域を拡張する． 4.3.4 節で説明したように，提案の最小サイクル時間は従来の 1/2 に短縮されている．従って，同じ電圧において，PoFF2 のサイクル時間は PoFF1 のサイクル時間の半分である．これは図 4.9 で確認されている．a では，サイクル・タイムは，1.1V の電圧において 412.5ps から 200ps に減少し，電源電圧は 412.5psi のサイクル・タイムにおいて 1.1V から 0.75V に削減される．

b のプロットは，a が右上に移動したように見える．これは，b の典型的なケースのトランジスタのパラメータは，すべて a の最悪の場合のダイのパラメータと同等に悪いためである．

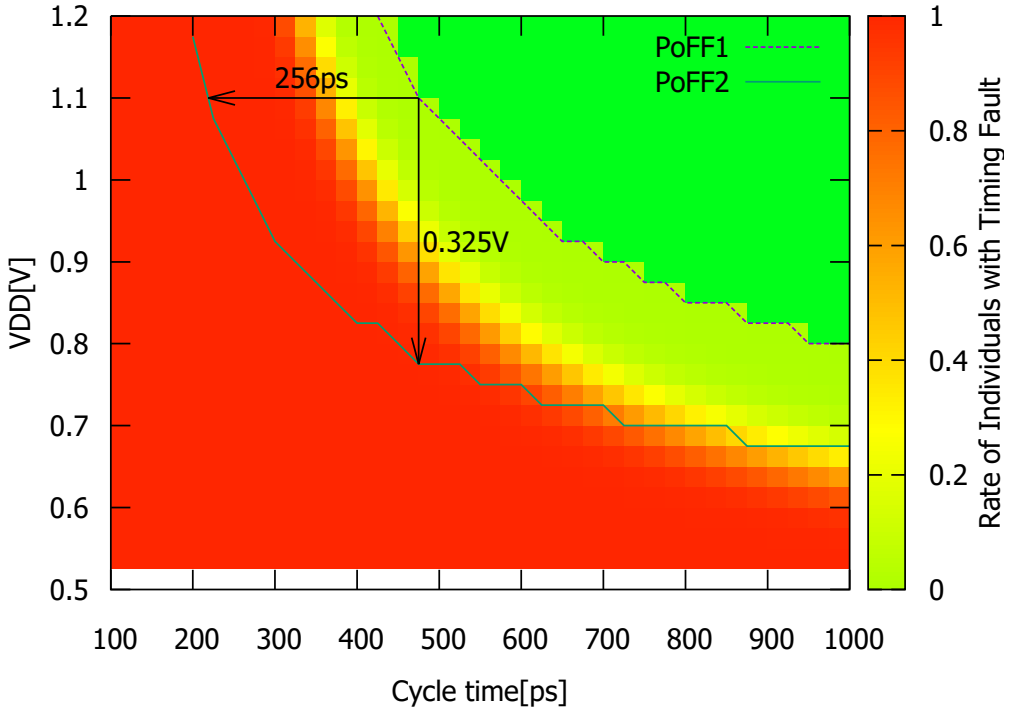
4.5　本章のまとめ

LSI のばらつきの対策として，TF 検出・回復がある．しかし，TF 検出回路である Razor FF を SRAM のようなダイナミック･ロジックに適用する手法は考えられておらず，Razor の適用はスタティック･ロジックで構成された回路に限られていた．SRAM ではプリチャージによって正しい値が消えてしまうために，Razor FF のシャドウに正しい値がサンプリングされず，TF 検出が完全には行われないという問題があった．

本章では，SRAM への Razor FF の適用手法を提案した．提案手法では SRAM のビットラインの読出しの結果，ディスチャージが行われていない場合にプリチャージを抑制する．SPICE シミュレーションによる動作検証によって，TF 検出が正しく行われることを確認した．



(a) 典型的ケースのダイ



(b) 最悪ケースのダイ

図 4.9: 提案 SRAM 回路のシュムプロット: a 典型的-，b 最悪-ケースのダイにそれぞれ対応.

第5章

動的タイム・ボローイングを可能とするクロッキング方式

5.1 動的タイム・ボローイングを可能にするクロッキング方式の構成

我々は入力ばらつきにおける平均遅延に基づいた動作を可能にする手法として，動的タイム・ボローイング（**DTB**）を可能にするクロッキング方式を提案してきた[13, 18].

5.1.1 回路構成と動作

図 5.1 に，DTB を可能にする方式の回路構成を模式的に示す．本方式は，基本的には，二相ラッチと TF 検出との組み合わせである．すなわち，同図上に示すような二相ラッチの回路のラッチ部分を，Razor の TF 検出回路に置き換えたものと考えてよい．なお，3.2 節で述べたように，本稿では TF 検出にダブル・サンプリングを用いた場合の説明を行うが，実用的な設計では遷移検出を想定する．

3.2.4 節で述べた Razor 特有のショート・パス問題を回避するため，ショート・パスに遅延を挿入する必要があるが，以下の工夫を行う：同図上の二相ラッチの回路では，ロジックのショート・パスとクリティカル・パスとが，図中〇印で示すゲートで合流した後，ラッチに接続されている．この場合，合流するゲート〇を二重化し，それぞれをメインとシャドウに接続する．その上で，シャドウに至るショート・パスにのみ遅

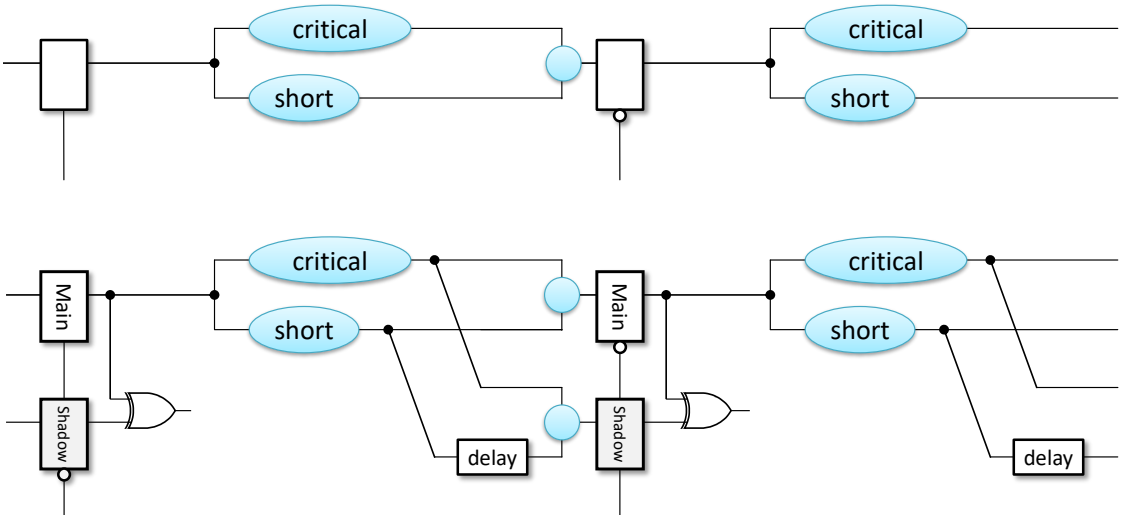


図 5.1: 二相ラッチ（上）と動的タイム・ボローイングを可能にする方式（下）の回路

延を挿入する．これにより，以下の 2 つを両立することができる：

- Razor 特有のショート・パス問題は，ショート・パスによりシャドウが正しい値をサンプリングできない問題であるから，シャドウに至るショート・パスに遅延を挿入すれば解消される．逆に，
- メインに至るパスに遅延を挿入しないことによって，ショート・パスが活性化した場合の実効遅延が伸びることが避けられる．5.1.2 節で詳述するように，これにより DTB の効果が最大化される．

実際の回路は，同図のようにショート・パスとクリティカル・パスがきれいに二分されている訳ではない．実際の遅延の挿入方法は [18] に詳しい．

5.1.2 動的タイム・ボローイング

2.5 節で述べたように，二相ラッチ方式においてはラッチの開いている期間を利用することは原則不可能であった．開いている期間を利用すべく，クリティカル・パス遅延よりサイクル・タイムを短くすると，クリティカル・パスが連続で活性化した場合に TF が発生するためである．DTB を可能にする方式では，TF 検出・回復を組み合わせることにより，ラッチの開いている期間を積極的に利用することが可能となる．

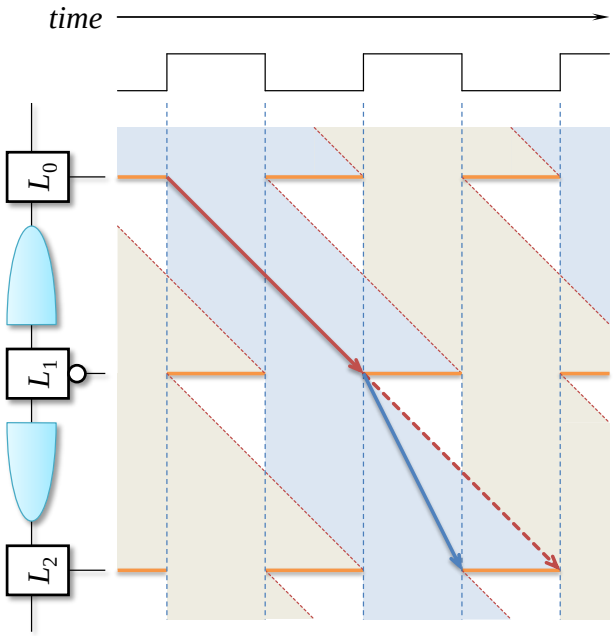


図 5.2: 動的タイム・ボローイング (DTB)

そしてこの結果，動作時に各ステージ間での**実効遅延の融通**が可能になる．図 5.2 に，DTB を可能にする方式のダイアグラムを示す．同図では，最初の半ステージでクリティカル・パスが活性化しているが，直後の半ステージで実効遅延が 0.5τ のパスが活性化したため，ぎりぎり TF を起こすことなく動作した場合を表している．逆に，直後の半ステージで再びクリティカル・パスが活性化した場合には，TF として検出されることになる．

遅延の「借金」 このように DTB を可能にする方式では，ラッチの開いている期間を利用することによって，遅延の累積を解消することができる．ダイアグラム上における，直線矢印がつながってステージ間を伝播する様子は DTB の効果を表している．

このように，遅延の累積を解消するためには実効遅延が短いことが望ましい．5.1.1 節で述べたように，ショート・パス問題のための遅延の挿入はメインに至るパスには行わないが，それは実効遅延をできる限り短縮するためである．

遅延の「貯金」　同図では，網掛けの領域が上下にオーバーラップしているが，これは図 5.1 に示す二重化されたパスの上で起こっている．すなわち，前のフェーズのシャドウに至るクリティカル・パスと，次のフェーズのメインに至るショート・パスにおける信号の伝達が同時に起こり得るため，ダイアグラム上でオーバーラップして見えるのである．したがって，別のフェーズが「混ざる」ことはない．

ショート・パスが連続で活性化した場合には，(同図ではオーバーラップの裏で）信号はラッチの閉じている期間に到着する．そこで，ラッチが開くまで待たされることになる．

したがって DTB を可能にする方式では，遅延の「借金」を持ち越して解消することができるが，遅延の「貯金」を持ち越すことは残念ながらできない．

タイミング制約　DTB を可能にする方式の最大遅延制約は，Razor と同様，TF 検出の検出限界によって決まる．図 5.2 のように，クリティカル・パスの遅延に対応する 45°の破線が検出ウィンドウの下端までに到着するなら，TF を検出することができる．

ただし DTB を可能にする方式では，前述したオーバーラップによって，サイクル・タイムを更に短縮することが可能となる．最大遅延制約は $1\tau/0.5$ステージとなり，単相 FF 方式や二相ラッチ方式に比べ，最大 2 倍の動作周波数の向上を見込むことができる．

大数の法則 と 入力ばらつき　開いている期間においては，ラッチはバッファとして機能する．すなわち，開いている期間を信号が通過する限りにおいては，各半ステージのロジックは，長大な 1 つの組み合わせ回路として動作することになる．このため，大数の法則により，入力ばらつきの平均値に基づく動作が可能となるのである．

5.1.3　クロッキング方式ごとの最小サイクル・タイムの比較

本章の最後に，各クロッキング方式における 1ステージのクリティカル・パス遅延 c と，シャドウ FF/シャドウ・ラッチへのショート・パス遅延 s に対する最小・最大サイクル・タイムについてまとめる．各クロッキング方式の最小/最大遅延制約を満た

すように最小/最大サイクル・タイム τ は，表 5.1 のようにまとめられる．Razor は，DTB を可能にする方式と同じく，$\alpha = 0.5$ とした．

TF 検出を行う方式では，最大のサイクル・タイムがシャドウに至るショート・パスの遅延に応じて決まる．提案においては $1/2 \times c$ から s までのサイクル・タイムを取り得るため，c を所与とすると，s は $1/2 \times c$ 以上である必要がある．

5.2 適用手法の概要

本章では，34-bit のリプル・キャリー・アダーを用いたカウンタへの提案方式の適用に関して詳述する．提案方式の適用は，単相 FF 方式で構成された回路を対象として，二相ラッチ化と TF 検出のための回路変換と，TF からの回復のための機構の付与によって行う．

なお，以降はパスの遅延はパス上の論理ゲートの個数によって計算する．また，二相ラッチ化における自由度を増やすため，FPGA のキャリー・チェーンは使用しない．また，遅延素子には LUT を用いる．

5.2.1 二相ラッチ化とタイミング故障検出機構の付与

図 5.3 に，7-bit のリプル・キャリー・アダーを用いたカウンタに対して，回路変換を行う例を示す．

まず，第 6 章で提案するアルゴリズムを用いて，二相ラッチ方式への変換を行う．クリティカル・パスがラッチを境に二分されるようにラッチ挿入が行われる．

次に，ラッチを Razor latch へ置き換える．5.1.2 節で述べたように，DTB を可能にするクロッキング方式では半ステージのクリティカル・パス遅延によって最小サイ

表 5.1: クロッキング方式の最小/最大サイクル・タイム

方式	最小	最大
単相 FF	c	N/A
二相ラッチ	c	N/A
Razor	$2/3 \times c$	$2 \times s$
DTB を可能にする方式	$1/2 \times c$	s

クル・タイムが決まり，クリティカル・パス遅延の 1/2 を超える遅延をもつパスが検出対象である．この回路の半ステージのクリティカル・パス遅延は 4 であるから，2 つ以上の LUT を通るパスの終端ラッチを Razor latch へ置き換える．

次に Razor latch に至るショート・パス遅延がショート・パス問題を起こさないように，Razor latch に至るショート・パスの一部の回路素子を複製し，遅延素子を挿入する．図中の紫色の素子はこうして挿入された素子である．ここで，5.1.3 節で述べたように，シャドウ・ラッチへのショート・パスはクリティカル・パスの 1/2 以上でなければならないため，本例ではショート・パスが 2 つ以上の LUT を通過するように遅延素子を挿入する．ただし，34-bit のリプル・キャリー・アダーカウンタに対しては，配線遅延などのばらつきを考慮し，ショート・パスの遅延がクリティカル・パスの 2/3 以上となるように多めに遅延素子を挿入する．

最後に，Razor latch が出力するエラー信号を半ステージごとに OR ゲートによって集約し，コミット・ステージへ伝搬する．伝搬された回路全体のエラー信号は，次章で述べる回復のための制御回路に入力される．

5.2.2　回復機構の付加

再実行　TF が発生しコミットされなかったフェーズの再実行においては，再び TF を起こさないようにする必要がある．そのために，再実行時に周波数を下げる方式と，各パイプライン・ステージでの 1 つのフェーズの実行を複数サイクル継続して行う方式が存在する [31]．本論文では後者の方式を採用する．

この再実行方式では，サイクル・タイム τ がクリティカル・パス遅延 d の $1/n$ 倍以上であるようなとき，元の n 倍のサイクル数を再実行されるフェーズの実行にかける．こうすることで，TF を発生させることなく実行を完了できる．このために，後続のフェーズの実行開始を $(n-1)\times$ ステージ数だけ遅らせる．再実行されたフェーズの結果は $n\times$ ステージ数の間確定しないため，その間はコミットを行わない．

例えば，5.1.2 節で述べたように，提案方式のサイクル・タイム τ の最小はクリティカル・パス遅延 d の 1/2 であるため，$n=2$ の場合までを考慮すればよい．したがって，再実行されるフェーズの後続は 1 ステージ数分だけ後に実行される．また，コミット・ステージへの伝搬は 2 ステージ数分だけ停止した後に行う．

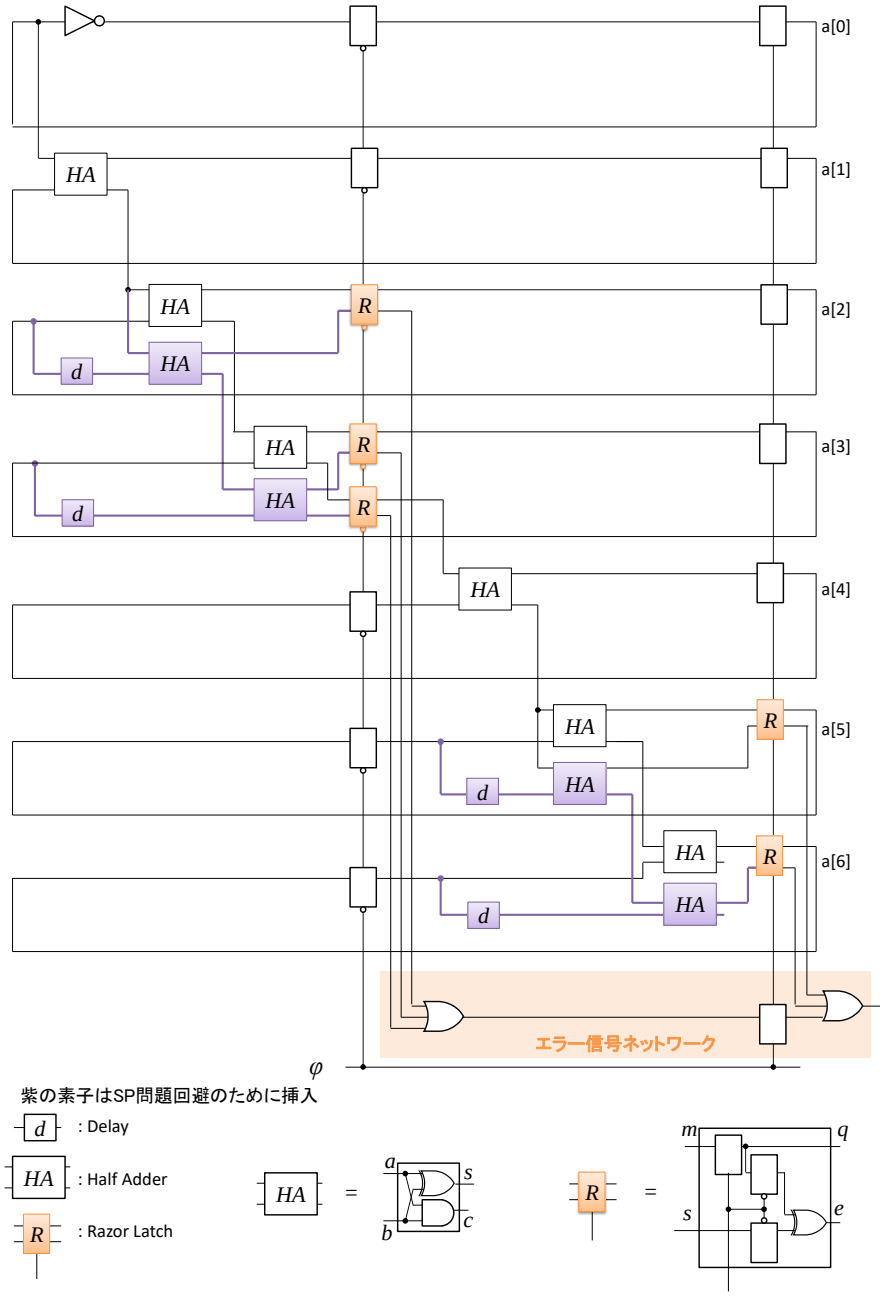


図 5.3: 7-bit のリプル・キャリー・アダーカウンタへの適用

提案方式を適用したカウンタの回復動作　図 5.4 に回復機構を組み込んだカウンタのブロック図を示す．

カウンタをプロセッサにおける program counter に見立て，コミット済みの program counter を記憶する FF を加えている．また，TF 検出の結果を得た後にコミットが

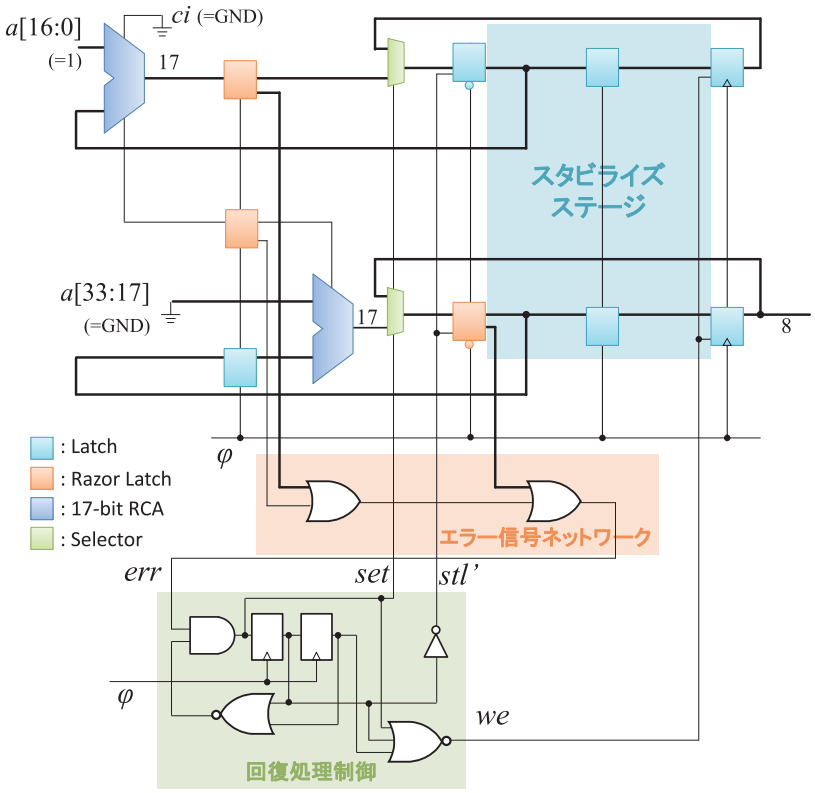


図 5.4: 回復機構を含めたカウンタの回路構成

なされる必要があるため，スタビライズ・ステージを設けている．

Razor FF から出力されたエラー信号は，エラー信号ネットワークに集約される．ネットワークの最終的な結果である err は回復処理の制御回路に入力される．回復処理の制御回路は，err を入力として，5.2.2 節で述べた再実行処理を制御する以下の信号を生成する：

set 再実行開始時のカウンタの値をセットするためのセレクト信号

stl 再実行中のカウンタの更新を止めるための信号

we アーキテクチャ・ステート更新のイネーブル

本回路は 1 ステージの回路であるから，再実行されるカウンタ計算の後続の計算は 1 サイクル待たされた後に実行される．また，コミット・ステージへの伝搬は 2 サイクルだけ停止した後に行う．したがって，回復のたびに必要なペナルティは 3 サイクルである．

図 5.5 に TF の検出の後に再実行が行われる様子を示す．同図の右側は制御回路が出力する信号群の波形を示したものである．まず，赤矢印で示されるパスの活性化によって TF が発生すると，そのサイクルの後半にエラー信号 err がアサートされる．同タイミングで，we がディスエーブルされ，アーキテクチャ・ステートの更新が止まる．また，同タイミングで set がアサートされ，カウンタにアーキテクチャ・ステートからの正しい値がロードされる．その次のサイクルから，stl がステージ数だけのサイクルの間アサートされることでカウンタの更新を止める．こうして，周波数が高くとも演算に十分な時間が確保され，再実行時は TF を起こすことなく演算が行われる．we のディスエーブルは $2\times$ ステージ数のサイクル数の後に解除される．その後，アーキテクチャ・ステートが再実行された演算の結果によって更新される．

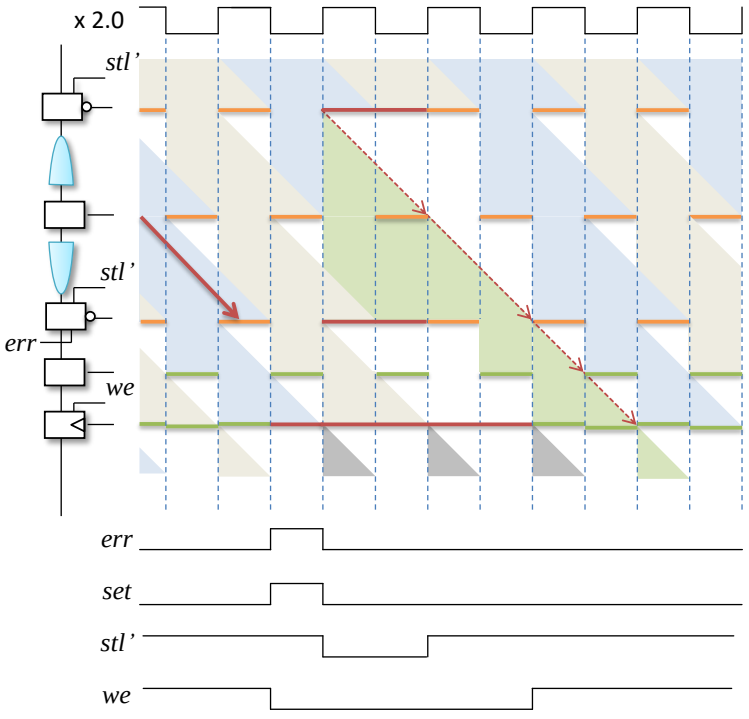


図 5.5: DTB を適用した回路における回復と再実行の様子

5.3　カウンタへの適用

本章では，各クロッキング方式を適用したカウンタを FPGA に実装し，その動作可能な周波数を測定する．対象のカウンタはリプル・キャリー・アダーを用いたもので，クロック・サイクルごとに 1 ずつカウントアップする．5MHz ずつ周波数を上げていき，クロック・サイクルごとに 1 ずつカウントアップが行われなくなる（直前の）周波数を測定する．

5.3.1　評価方法

対象の回路構成　比較するクロッキング方式は，単相 FF，二相ラッチ，Razor，提案方式の 4 つである．

提案方式の回路構成は，図 5.4 に示した．

なお，Razor においては，5.1.3 節で述べたように，単相 FF 方式の 1.5 倍の周波数で動作することを想定し，クリティカル・パス遅延の 2/3 を超える遅延をもつパスを検出対象とし，シャドウ・ラッチに至るショート・パスはクリティカル・パス遅延の 1/2 を超えるように遅延挿入を行う．

評価環境　使用したボードは Digilent Nexys 4 DDR で，搭載 FPGA は Xilinx Artix-7 XC7A100T-1CSG324C である．

論理合成，配置配線には Vivado Design Suite 2016.3 を用いた．逆相ラッチと Razor FF の挿入は，5.2.1 節で述べたツールによって行った．

評価方法　34bit のカウンタの上位 8bit を FPGA ボードに備え付けられた 8-bit LED に出力する．配置配線に不要な制約を与えないため，カウンタの上位 8bit はそれぞれ 2 つ FF を介した後に LED に出力しており，カウンタの値は 2 サイクルの遅れで常時 LED に表示される．各クロッキング方式が正しく動作したかどうかは，この 8-bit LED の表示が 1 ずつカウントアップするどうかを目視によって確認する．最高 300MHz では，上位 8bit のうちの最下位ビットは，$300\text{MHz} \div 2^{(34-8)} \simeq 4.5\text{Hz}$ で点滅するため，目視で確認できる．下位 $(34-8=)$ 26bit については LED などによって直接には観測していないが，パス遅延がより短いため，TF が起こる確率は無視してよい．

リプル・キャリー・アダーを用いたカウンタのクリティカル・パスは，リプル・キャリー・アダーのキャリー・チェーンである．動作周波数を上げていった時に最初に起こる TF は，最上位桁へのキャリーの伝搬が間に合わなくなるというものである．すなわち，8-bit LED の値が，本来 $011\ldots11 \rightarrow 100\ldots00$ と変化するところ，TF 発生時には，$011\ldots11 \rightarrow 000\ldots00$ と変化することになる．

TF 検出・回復 3.2.2 節で述べたように，TF 検出・回復を行う方式では回復は数サイクルで行われるため，上述した $011\ldots11 \rightarrow 000\ldots00$ の変化は目視では観測できない．TF の発生は，エラー信号がアサートされた後（しばらくの間）点灯するエラー LED によって確認する．その結果，TF が発生する周波数の領域では，目視で 8-bit LED は 1 ずつカウントアップし続けているように見えるが，このエラー LED が点滅することになる．

動作周波数が TF の検出限界を超えると，メイン FF に加えて，シャドウ・ラッチにも間に合わなくなるため、エラー LED も点灯しなくなる．

したがってこの評価では，TF 検出を行わない手法 (単相 FF, 二相ラッチ) については，TF が発生せずに動作する周波数を ; TF 検出を行う手法 (Razor, 提案方式) については，TF が検出・回復可能な周波数を，それぞれ測定することになる．

一般には，3.2.2 節で述べたように，周波数向上による性能向上と TF からの回復のペナルティによる性能低下のトレードオフが存在するが，この評価では，回復のペナルティは考慮していない．後述するように，このようなカウンタでは TF がほとんど起こらない．したがって，周波数を上げても，回復のペナルティによる性能低下が，周波数向上による性能向上を上回ることはない．

5.3.2 カウンタにおける TF 発生率

本章では TF 検出を備えた方式に関して，カウンタにおける TF 発生率（Occurrence Rate of Timing Faults: ORTF）について記す．

カウンタの段数を $n = 34$ とおく．一段の桁上げのゲート遅延を一律に d_c とすると，桁 i の変化するときのパスの遅延 d_i は $d_c i$ で与えられる．特に CP の遅延 c は $d_c n$ と計算できる．また，桁 i が変化する確率は $(1/2)^{i-1}$ で与えられる．

Razor においては，サイクル・タイム t に対して，$d_i > t$，すなわち $d_c i > t$ であるような桁 i の値が変化する際に TF が発生する．適用対象のカウンタでは $i+1$ 桁

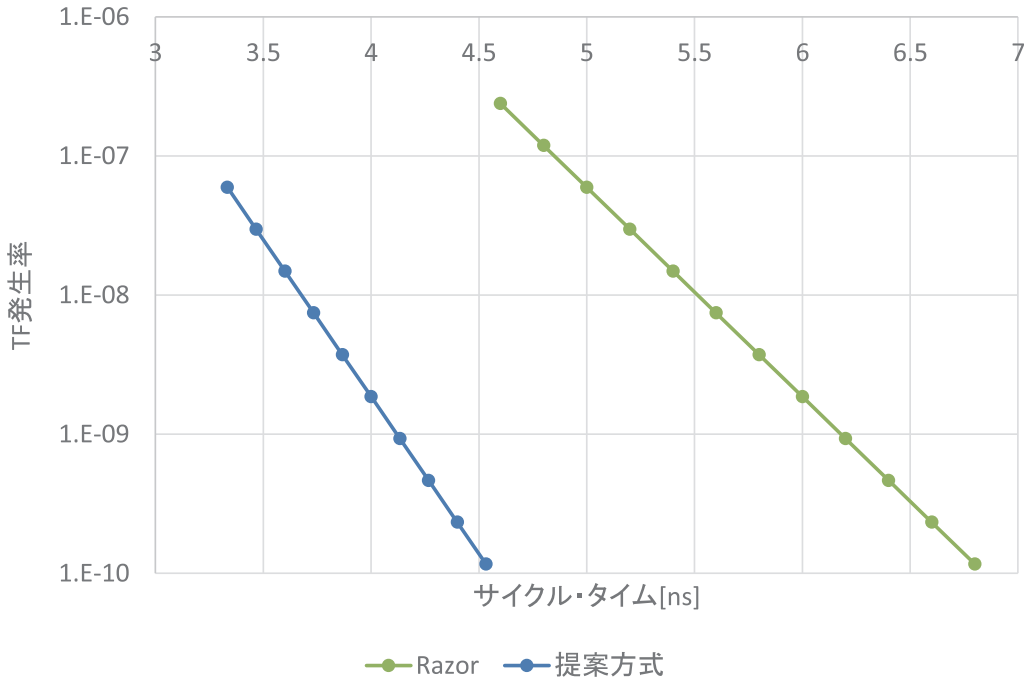


図 5.6: サイクル・タイムに対する TF 発生率

が反転するときは i 桁も反転することに注目すると，$d_c i > t$ である 1 番桁が小さい i の変化の確率が ORTF となることが分かる．

提案方式においては，$d_c i > 3/2 \times t$ であるような桁 i の変化が TF を起こす．Razor と同様にそうした i のうち最も桁が小さい i のパスの活性化確率が ORTF となる．

このサイクル・タイムと ORTF の関係を図 5.6 に示す．ここでは c は 0.2ns とした．ORTF の軸は対数表示である．Razor と提案方式はそれぞれの方式において，TF が起きる最大のサイクル・タイムから，TF が検出できる最小のサイクル・タイムまでをプロットしている．

提案方式の ORTF は，同じサイクル・タイムのときの Razor に対して小さく，例えば $t = 4.5ns$ 付近の点では 1/1000 ほど小さい．また，検出限界付近における ORTF を比べても提案方式の方が低く，提案方式は再実行のペナルティを比較的低く抑えたままサイクル・タイムを削減していくことができることが示されている．

しかし，いずれの手法でも TF の発生率は小さく，周波数を上げることによる性能向上を妨げるほどではない．そのため，本論文では動作可能な周波数のみを評価対象とする．

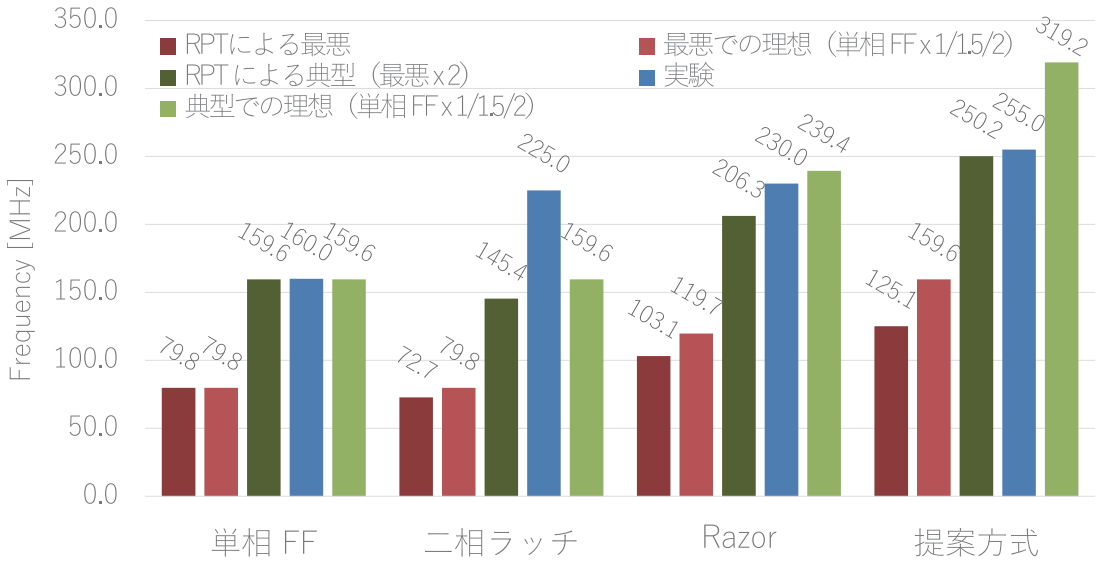


図 5.7: 各クロッキング方式の理論値と実験結果

5.3.3 実験結果

図 5.7 はそれぞれの手法について実験結果を示す．また，以下の理論値を実験結果と共に示す：

RPT による最悪（濃赤） Vivado のタイミング・レポート (RPT) から得られるクリティカル・パスのワースト・ケース遅延を基に，第 2 章と 3.2 節と 5.1 節で示した式から計算した，すなわち，クリティカル・パス遅延 d に対して最小サイクル・タイムは，単相 FF: d，二相ラッチ: d, Razor: $2/3 \times d$, 提案方式: $1/2 \times d$ となるとして計算した．実際の製品の定格周波数となる．

最悪での理想（赤） 第 5 章で述べた式でパスの遅延 d を一定とした場合，つまり単相 FF 方式の最悪 (79.8MHz) を，それぞれ，1，1，1.5，2 倍にしたもの（最小サイクル・タイムの 1，1，2/3，1/2 の逆数）．

RPT による最悪（濃赤）は，最悪での理想（赤）よりはいずれも若干小さくなっている．これは，単相 FF 以外の方式では，逆相ラッチやシャドウ FF，遅延素子などのオーバーヘッドによりクリティカル・パス遅延 d の方が若干伸びるためである．

RPT による典型（濃緑）　ティピカル･ケースを想定し，RPT の結果を 2 倍にしたもの．単相 FF の実験結果とよく符合するため，実験環境（温度・電圧）ではワースト･ケースのおよそ 2 倍となっていると考えられる．

実験結果（青）　8-bit LED によって，カウンタが正しく動作していることを観測できた最大の周波数．提案方式は単相 FF 方式に対して 1.6 倍の周波数で動作することを確認できる．

典型での理想（緑）　単相 FF 方式の典型 (159.6MHz) を，それぞれ，1，1，1.5，2 倍にしたもの（最小サイクル・タイムの 1，1，2/3，1/2 の逆数）．

実験結果（青）が，RPT による典型（濃緑）と理想（緑）との間の値を示すことは，実験環境（温度・電圧）において回路が想定通りの動作していることを示す．

実験結果（青）は，単相 FF，Razor，提案方式では，典型の場合とほぼ相違がない．一方，二相ラッチについては，典型の場合よりも実験結果が高い周波数を示した．

二相ラッチにおける DTB　これは，以下のように，カウンタでは二相ラッチでは実際には利用できない DTB が起きるからである．

カウンタの値が，$011\ldots10 \rightarrow_{-1} 011\ldots11 \rightarrow_{0} 100\ldots00 \rightarrow_{+1} 100\ldots01$ と変化するとき，$\rightarrow_0$ の遷移において最上位ビットへの桁上げが伝搬され，クリティカル・パスが活性化する．その直前・直後のサイクルにおいては，変化するのは，最下位のビットのみであり，実効遅延は非常に小さい．

したがって，$\rightarrow_0$ のサイクルにおいては，遅延の累積のない状態から始まり，ここで 0.5cycle 分借金をしたとしても，次のサイクルには解消されるから，TF を起こすことなく動作することができる．

この場合の最小サイクル･タイムは $2/3 \times d$ となり，実験結果とよく符合する．

しかし，カウンタではない一般的な回路においては，クリティカル･パスが連続で活性化されることを考慮しなければならない．したがって，TF 検出を備えない二相ラッチ方式では，実験結果（青）のような高い周波数で動作させることはできない．

第6章

二相ラッチ化手法

FF を用いた回路をラッチを用いた回路に変換する問題は，最小カット問題の一種に帰着する．ただしこの際，始点から終点に至るすべての道にカット・エッジをただ 1 つ含むという制約がある．既存の最小カット・アルゴリズムでは，この制約を満たすことができない．本稿では，この制約が，カットが逆方向カット・エッジを含まないことと等価であることを証明し，逆方向カット・エッジのない最小カットを見つけるアルゴリズムを提案する．このアルゴリズムにおいて最もオーダが大きい部分は既存の最大フロー・アルゴリズムであり，提案アルゴリズム全体のオーダはこれより悪化することはない．実験により，ゲート数約 3.4 万，配線数約 9.7 万程度の回路に対しても，約 375 秒の実用的な時間で最適解が求められることが分かった．

6.1 本章の内容

同期回路における同期動作を実現する方式には，フリップ・フロップ (**FF**) を用いたものの他に，二相の**ラッチ**を用いたものがある．前者に比べて後者は，タイミング制約が緩いという利点がある [32–35] が，設計はより煩雑である．そこで，前者を入力として，後者を自動的に生成することが考えられる．

逆相ラッチ挿入問題　そのためには，図 6.1（上）から（中）に示すように，

1. FF をラッチに変更する と同時に，
2. FF と次の FF とに挟まれたロジックの中央に**逆相の**ラッチを挿入する

という変換を行えばよい．ただし逆相ラッチを挿入する際には，以下の制約がある：ロジック内のあらゆるパスに対して，逆相ラッチはただ 1 つ挿入されなければならない．

また，逆相ラッチの挿入位置に関しては，以下の 2 つの評価基準がある：

1. ラッチの挿入個数は少ないほどよい．
2. クリティカル・パスを短縮するため，挿入位置はロジック内の各パスの遅延を等分することが望ましい．

グラフ・カット　これらの評価基準に対して最適な挿入位置を求めるにあたって，回路は，図 6.1（下）に示すようなグラフに写像することができる：

- ロジックの，FF やゲートなどのインスタンスを頂点，ネットをエッジとする．

  インスタンスとネットには，信号の流れる向きがあるから，グラフ（エッジ）は有向となる．

- エッジのコストは，ステージ内の各パスの中央ほど低く，両端ほど高く設定する．

また，以下のようなダミーの頂点とエッジを追加する：

- 入/出力側の FF の前/後に，始点 s/終点 t を追加する．s/t と FF を結ぶエッジのコストは ∞ とし，それらがアルゴリズムに影響を与えることのないようにする．

- ネットの分岐にダミーの頂点を挿入する（図中 d）．

するとグラフは，ラッチが挿入されたエッジにおいて，s 側と t 側に二分される．このような二分割をグラフの**カット**という．二つの分割にまたがるエッジ（この問題ではラッチが挿入されるエッジ）を**カット・エッジ**という．カット・エッジのコストの総和をカットの**サイズ**という．すると問題は，サイズ最小のカットを求める**最小カット**の問題（の一種）に帰着される．同図の例では，破線で示すカットが，サイズ $1+1=2$ で最小である．

なお，図中下側のカット・エッジのように，ネットの分岐に対して挿入したダミー d に対しては，その入力側を選ぶことで，複数の出力先をまとめて 1 個のラッチを挿入することを表現することができる．

単一カット・エッジ制約　逆相ラッチ挿入問題では，前述したように，ラッチはロジック内のあらゆるパスにただ 1 つ挿入されなければならない．この制約は，グラフの言葉では以下のようになる：

単一カット・エッジ制約 始点 s から終点 t へ至るあらゆる道上にカット・エッジが 1 つ存在する

なお，逆相ラッチ挿入問題では，この制約を満たすカットは必ず存在する；すな

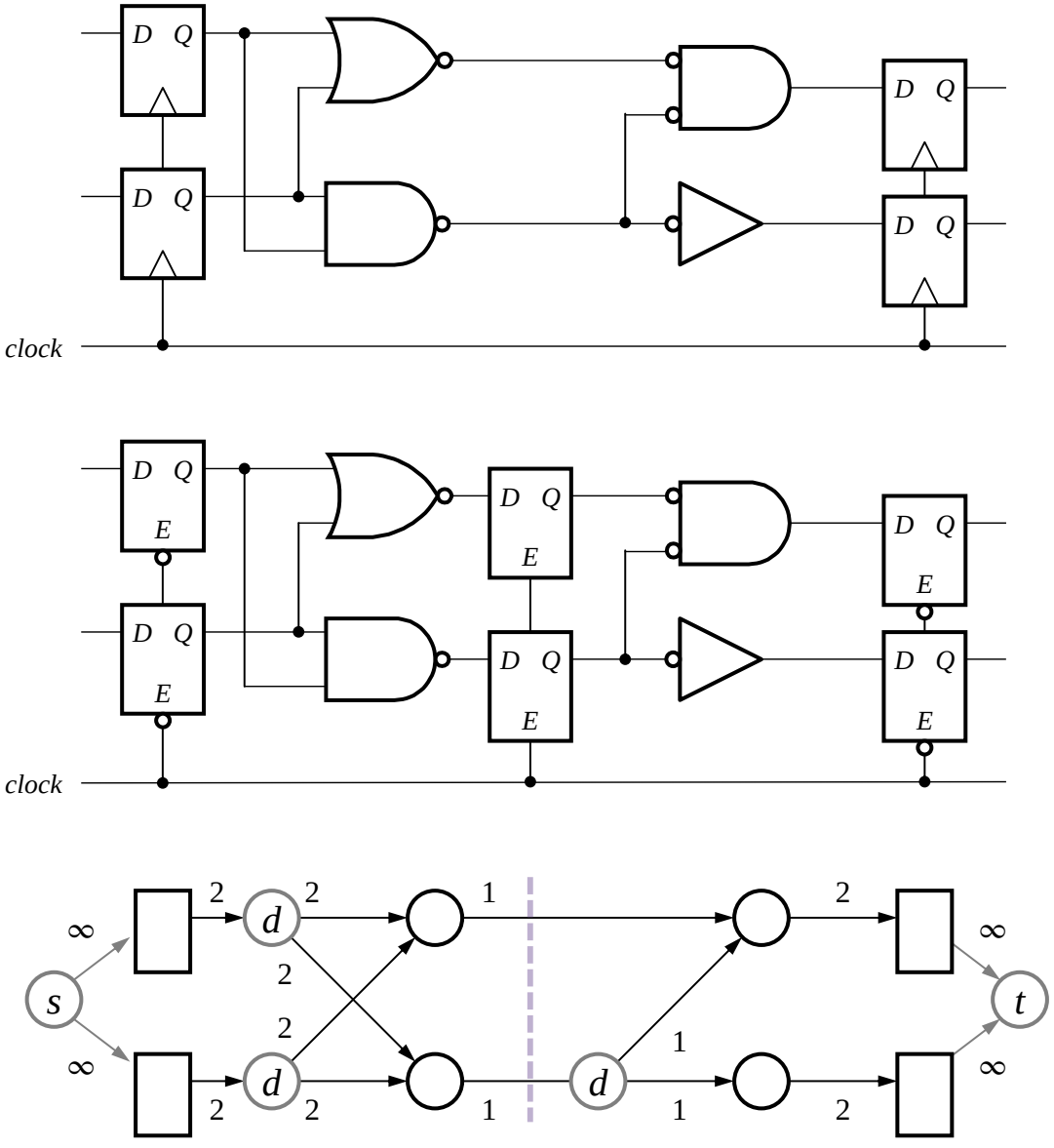


図 6.1: FF（上），二相ラッチを用いた回路（中）と，そのグラフ（下）

わち，入力側ラッチの直後，あるいは，出力側ラッチの直前で分割したものである．特に前者の場合，逆相ラッチを挿入された回路は元の FF の回路と等価となる．

既存の最小カット・アルゴリズム　既存の最小カット・アルゴリズムは，道上のカットの数を意識せず，この問題にそのまま用いることはできない．6.4.6 節で示すように，実用的な回路に対しても単一カット・エッジ制約を満たさないカットが選ばれることがある．

そこで本稿では，この単一カット・エッジ制約を満たすカットのうちでサイズ最小となるものを見つけるアルゴリズムを提案する．本稿の構成は以下のとおりである：6.2 節で既存のアルゴリズムについてまとめた後，6.3 節で提案のアルゴリズムについて詳しく述べる．6.4 節では，逆相ラッチ挿入問題に対してアルゴリズムを実行した結果について述べる．

6.2　既存のアルゴリズム

最小カットを求めるには，最大フロー最小カット定理に基づいて，最大フローを求めた結果として最小カットを求めることが一般的である．6.2.1 節と 6.2.2 節では，そのようなアルゴリズムの例として，最も基本的なフォード・ファルカーソンのアルゴリズムを概説する．

しかし，このような最大フローに基づく方法では，6.1 節で述べた単一カット・エッジ制約を満たすことはできない．6.2.3 節で紹介する無向グラフに変換する方法もまた，単一カット・エッジ制約を満たさない．6.2.4 節で紹介する探索による方法は，単一カット・エッジ制約を満たすことを念頭に設計されたものであるが，実行時間に問題がある．

6.2.1　フォード・ファルカーソンのアルゴリズム

前述のように，最小カットは，フロー・ネットワークにおける最大フローを求めた結果から求めることが一般的である．なお，フロー・ネットワークにおいては，エッジに与える重みはフローの**容量**と呼ぶが，前章におけるコストと同じと考えてよい．

そのための最大フロー・アルゴリズムとしては，**フォード・ファルカーソンのアルゴリズム** (Ford-Fulkerson algorithm) [36] が最も基本的である．

残余ネットワーク フォード・ファルカーソンのアルゴリズムでは，元のフロー・ネットワークに対して**残余ネットワーク** (residual network) というネットワークを生成する．元のフロー・ネットワークにおいて容量 $c(u,v)$ のエッジ $u \to v$ にフロー $f(u,v)$ を流したとき，残余ネットワークにおける u，v 間には順方向と逆方向の 2 つの有向エッジを張る：

順方向 まだあと $c_f(u,v) = c(u,v) - f(u,v)$ だけ流せるという意味で，容量 $c_f(u,v)$ の順方向エッジ

逆方向 逆に，$c_b(u,v) = f(u,v)$ だけ減らすことができるという意味で，容量 $c_b(u,v)$ の逆方向エッジ

なお，$c_f(u,v) + c_b(u,v) = c(u,v) - f(u,v) + f(u,v) = c(u,v)$ である．

増加道 残余ネットワークにおいて s から t へと至る道を**増加道** (augmenting path) という．s から t へのフローは，増加道の容量の最小値（すなわち，増加道を形成するすべてのエッジの容量のうちの最小値）だけ，増加させることができる．

増加道に上述した逆方向エッジが含まれる場合には，そのエッジのフローを逆に減少させることになる．このおかげで，増加道をグリーディに見つけて行っても最大フローが求まるというのがフォード・ファルカーソンのアルゴリズムの要諦である．証明は，[37] などを参照されたい．

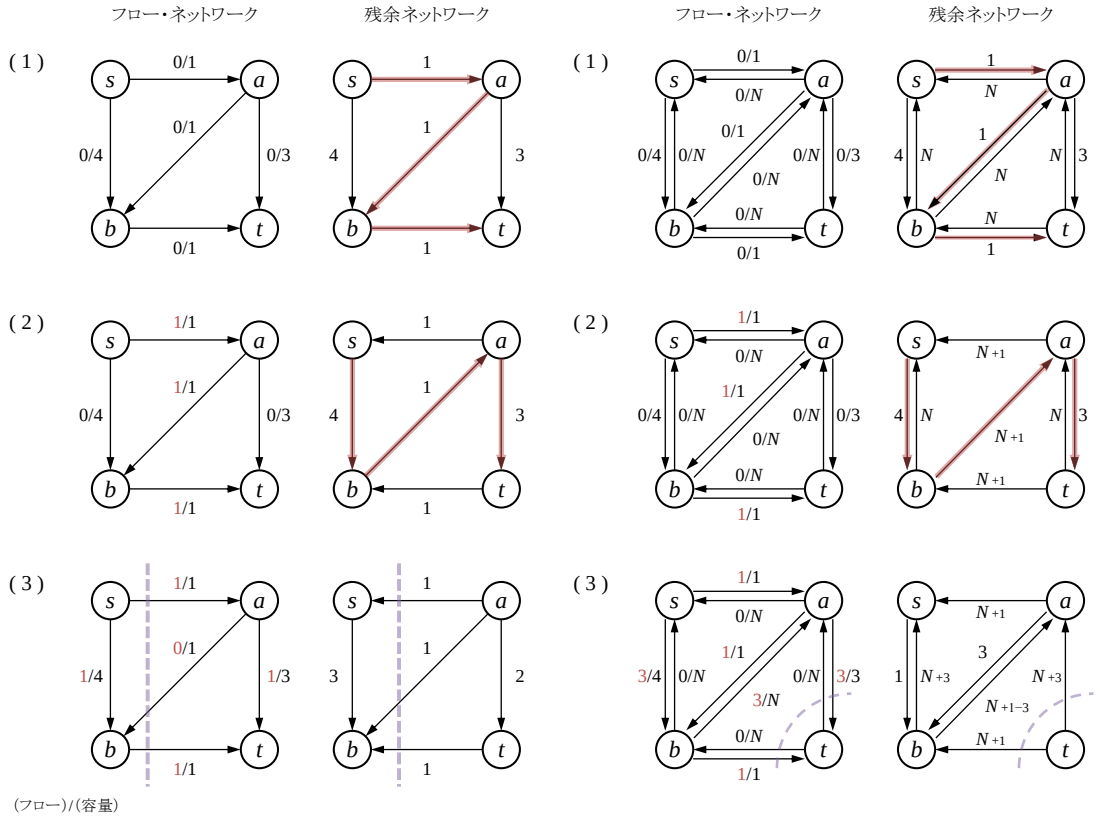
アルゴリズムの動作例 図 6.2 (左) の例を用いてフォード・ファルカーソンのアルゴリズムの動作を説明する．同図中，1.〜 3. の各段階において，左がフロー・ネットワーク，右がそのフロー・ネットワークに対応する残余ネットワークを示す．一般に，フロー・ネットワークにおいては，エッジに「(フロー)/(容量)」と付す．また，残余ネットワークにおいては，容量 0 となったエッジは除去する[1]．

アルゴリズムは，以下のように進む：

1. 初期状態では，フローは 0 とする．

   したがって右の残余ネットワークは，左のフロー・ネットワークの容量をそのまま写したものとなる．

[1] 主に図の見やすさのため．要は容量 0 のエッジを含む増加道を見つけないようにすればよいので，プログラムでは容量 0 のエッジとして残しておいた方が実装が容易であろう．



(a) オリジナル　　(b) 提案

図 6.2: オリジナル（左）と提案（右）のフォード･ファルカーソンのアルゴリズムの動作例

この残余ネットワークにおいて，s から t へ至る増加道を探す．辞書順だと，$s \to a \to b \to t$ が見つかる（太矢印）．

2. この増加道に最大のフローを流す．この増加道を形成するエッジ $s \to a$，$a \to b$，$b \to t$ の容量はすべて 1 であるから，それぞれに容量一杯のフロー 1 を流すことになる．すると，左のフロー･ネットワークが得られる．

   更にこのフロー･ネットワークから右の残余ネットワークを得る．エッジ $s \to a$，$a \to b$，$b \to t$ のそれぞれに容量一杯のフロー 1 を流したため，残余はそれぞれ 0 となる．これらのエッジに対しては逆に，フローを 1 だけ減らすことができるという意味で，容量 1 の逆向きのエッジを張る．

   この残余ネットワークにおいては，増加道 $s \to b \to a \to t$ が見つかる（同じく太矢印）．

3. この増加道に最大容量である 1 のフローを流すと，左のフロー・ネットワークが得られる．エッジ $a \to b$ には，先ほど 1 のフローを流したが；今回，同じく 1 のフローを今度は逆向きに流したので，フローはキャンセルされて 0 に戻る．

   更にこれから右の残余ネットワークが得られる．この残余ネットワークにおいては，s から t へ至る増加道はもはや見つからないので，アルゴリズムは終了する．

最大フロー　増加道 $s \to a \to b \to t$ と $s \to b \to a \to t$（図 6.2 中の太矢印 2 本）によって 1 ずつ増加されたので，最大フローは合計 2 となる．

物理的なフローは，これらの 2 つの増加道の重ね合わせである；すなわち，道 $s \to a \to t$ と $s \to b \to t$ に 1 ずつ流されている．キャンセルされるので，エッジ $a \to b$ にはフローは流されない（0 である）．

6.2.2 最大フロー最小カット定理

フォード・ファルカーソンのアルゴリズムにおいては，最小カットは，アルゴリズム終了時の残余ネットワークにおいて，s から到達可能な頂点とそれ以外の頂点への分割として与えられる．図 6.2 の場合，終了時の残余ネットワーク，すなわち，3. 右において，s から到達可能であるのは b のみであるので，最小カットは $\{s, b\}$ と $\{a, t\}$ である．

2 つの部分をまたがるカット・エッジのうち，$s \to a$ と $b \to t$ は，s 側から t 側へ向かう順方向エッジであり；$a \to b$ は逆に，t 側から s 側へ向かう逆方向エッジである．最大フロー最小カット定理の文脈においては，上述した最大フロー2 に対する最小カットは，順方向カット・エッジの容量の総和，すなわち，$c(s, a) + c(b, t) = 1 + 1 = 2$ で与えられ；逆方向カット・エッジの容量，すなわち，$c(a, b) = 1$ は含めない．順方向カット・エッジのフローはそれぞれの容量一杯であり，同時に，逆方向カット・エッジのフローはすべて 0 であるとき，最大フローは実現される（最大フロー最小カット定理）．

この最小カットでは，道 $s \to a \to b \to t$ を構成する 3 つのエッジ $s \to a$，$a \to b$，$b \to t$ はすべてカット・エッジとなっており，単一カット・エッジ制約は満たされていない．

6.2.3　無向グラフにおける最小カット

前節までで述べた，のような最大フロー・アルゴリズムを用いる以外には，無向グラフ化して最小カットを求めることが考えられる．

無向グラフに対する全域的な最小カットも，永持・茨木のアルゴリズム [38,39] などを用いて，効率よく求めることができる．

しかし，無向グラフにおける最小カットも，単一カット・エッジ制約を満たすとは限らない．実際，図 6.2 の例では，フォード・ファルカーソンのアルゴリズム場合と同じ最小カット（図 6.2（左）の 3.）が得られる[2]．

6.2.4　探索による方法

単一カット・エッジ制約を満たすため，我々は，探索による方法を試してきた [18]．

探索木のノードは，カット・エッジ候補の集合とする．すなわち，探索木の葉において，この集合の要素がカット・エッジとなる．探索ノードの展開は，この集合にエッジを 1 つ加えることになる．展開のたびに，加えられたエッジからグラフを上流，下流へと経路探索し，経路上のエッジを候補から除外する．この除外によって，単一カット・エッジ制約は保証される．この探索木上で最良優先探索を行えば，単一カット・エッジ制約を満たすカットのうちサイズ最小のものが得られる．

しかしこの方法は，計算量の大きさと，探索空間の広さのため，実用的な時間内に結果を得られていない：

計算量の大きさ　エッジの候補からの除外は，エッジ数 E に対して $O(E)$ の時間がかかる．そのため，通常の探索問題に比べ，展開に時間がかかる．

探索空間の広さ　例えば，始点側 FF から終点側 FF までの最短経路がエッジ 10 段あり，1 段ごとにカット・エッジの候補が 100 あるとすると，解空間は 100^{10} にもなる．探索順序を工夫して 1/100 のノードの探索で解が求まったとしても，探索ノード数は依然 100^9 もある．

実際，6.4 節で評価するエッジ数 10 万程度の回路に対して探索を行ったところ，2 日経過しても終了しなかった．なお，提案手法では，約 375 秒で終了する（6.4.4 節）．

[2] ただし最小カットのサイズは，$c(s,b)+c(a,t)=1+1=2$ ではなく，$c(a,b)=1$ を含む 3 となる．

なお，カット・エッジを加えると残りの解空間が変化するため，A^* アルゴリズムのためのヒューリスティック関数は見つけられていない．

6.3 提案アルゴリズム

本稿で求めるべきカットには，始点から終点へ至るすべての道にカット・エッジがただ 1 つ現れるという単一カット・エッジ制約がある．実はこの制約は，逆方向カット・エッジがないという逆方向カット・エッジなし制約と等価である．そこで本稿では，逆方向カット・エッジを含まない最小カットを求めるアルゴリズムを考える．まず，6.3.1 節で，単一カット・エッジ制約と逆方向カット・エッジなし制約が等価であることを証明する．その後，6.3.2 節と 6.3.3 節で，アルゴリズムの手順と動作例を示す．アルゴリズムの正しさの証明は，改めて 6.3.4 節で行う．最後に 6.3.5 節で，提案アルゴリズムの計算量は採用した最大フロー・アルゴリズムより増加しないことを述べる．

6.3.1 逆方向カット・エッジなし制約

定理 1. フロー・ネットワークにおいて，以下は等価である：

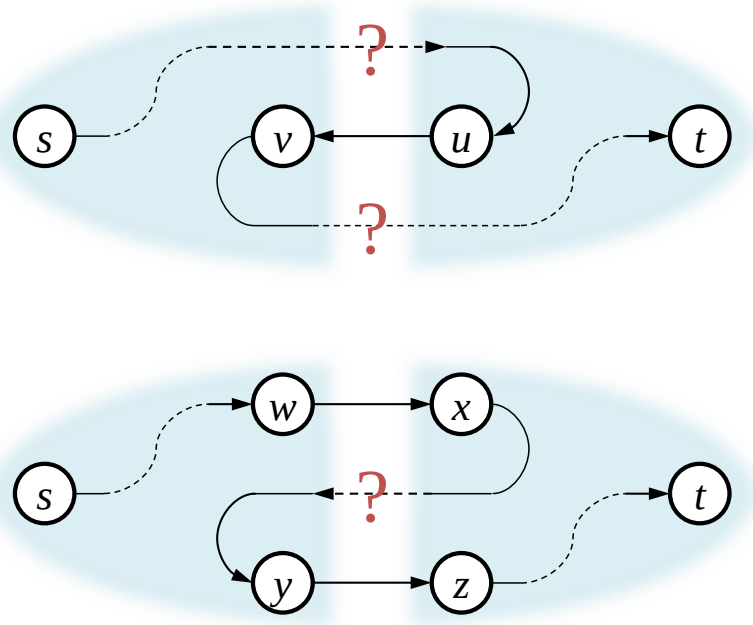


図 6.3: 逆方向カット・エッジを含むカット

単一カット・エッジ制約 始点 s から終点 t へ至るすべての道にカット・エッジがただ 1 つ現れる．

逆方向カット・エッジなし制約 逆方向カット・エッジがない，すなわち，カット・エッジはすべて順方向である．

証明． 単一カット・エッジ制約 $\Rightarrow$ 逆方向カット・エッジなし制約を，背理法で証明する．s から t へ至る道にカット・エッジがただ 1 つ現れ，かつ，それが逆方向であると仮定する．その逆方向カット・エッジを $u \to v$ とする（図 6.3 上）．すると，s から u，v から t へ至る道は（順方向の）カット・エッジを含むことになり，ただ 1 つという仮定と矛盾する．

逆も，同じく背理法で証明する．カット・エッジはすべて順方向エッジであり，かつ，s から t へ至る道のうち，カット・エッジが 2 つ以上現れるものがあると仮定する．2 つの順方向カット・エッジを $w \to x$ と $y \to z$ とする（図 6.3 下）．すると，道 $x \to \cdots \to y$ は，逆方向カット・エッジを含むことになり，カット・エッジがすべて順方向であるという仮定と矛盾する． □

したがって，6.1 節の問題などのために，始点から終点へ至るすべての道にカット・エッジがただ 1 つ含まれるようなカットを求めるためには，逆方向カット・エッジのないカットを求めればよい．

6.3.2　提案アルゴリズムの手順

提案の逆方向カット・エッジを含まない最小カット・アルゴリズムは，以下のとおりである：

1. **前処理**　元のフロー・ネットワークの全てのエッジに対して，容量 N の逆平行エッジを追加する．
2. **最大フロー**　このフロー・ネットワークに対して，任意の最大フロー・アルゴリズムによって最小カットを求める．

N は，十分に大きい値，具体的には，元のフロー・ネットワークにおけるカット・サイズの最大値より大きい値であればよい．そしてこの最大値は，元のフロー・ネットワークにおけるすべてのエッジの容量の総和で抑えられる．

証明は 6.3.4 節で行うが，直感的には，このアルゴリズムは以下のようにして逆方向カット・エッジを避ける：あるエッジを逆方向カット・エッジとして選ぼうとすると（元のエッジではなく）追加された逆平行エッジの容量 N がカット・サイズに加算され，最小カットとして選ばれない十分に大きな値になる．

6.3.3 動作例

本節では主に，前節における 2.，すなわち，最大フローを求める部分の動作を説明する．

動作例 図 6.2（右）に，提案アルゴリズムの動作を示す．元のフロー・ネットワークは同図（左）のものと同一である．同図（右）では，すべてのエッジに対して容量 N の逆平行エッジが追加されている．元のフロー・ネットワークのすべてのエッジの容量の総和は 10 であるので，$N = 11$ とした．

最大フロー・アルゴリズムとしては，同図（左）と同じフォード・ファルカーソンのアルゴリズムを用いている．したがって，同図（左）/（右）では，同一の最大フロー・アルゴリズムが異なる初期フロー・ネットワークに対してどのように振る舞うかを見ることになる．アルゴリズムは以下のように進むが，実際，2. までは，同図（左）と変わらない：

1. 増加道 $s \to a \to b \to t$ が見つかる（太矢印）．
2. この増加道に最大容量である 1 のフローを流すと，フロー/残余ネットワークが得られる．この残余ネットワークにおいては，増加道 $s \to b \to a \to t$ が見つかる（太矢印）．
3. ただし，この増加道のフローは（左）とは異なる．この増加道のフローは，(左) では $b \to a$ の 1 によって制限されていた．(右) では，容量 $N + 1$ の逆平行エッジ $b \to a$ が追加されたため，$b \to a$ ではなく，$a \to t$ の 3 によって制限されることになる．

   この結果，$a \to t$ は飽和し，残余ネットワークにおいて増加道はもはや見つからず，アルゴリズムは終了する．

最大フロー最小カット　最大フローは $1+3=4$ となり，元のフロー・ネットワークにおける最大フロー 2 より大きい．これは，元のフロー・ネットワークに存在しない容量 N の逆平行エッジ $b \to a$ に 2 のフローを流すことによって達成されていることに注意する必要がある．

6.2.1 節で述べたとおり，フォード・ファルカーソンのアルゴリズムでは，最小カットはアルゴリズム終了時の残余ネットワークにおいて s から到達可能な頂点とそれ以外の頂点への分割として与えられる．同図の場合，s から b，そして a へ到達可能であるので，最小カットは $\{s, a, b\}$ と $\{t\}$ となる．カット・エッジは，$a \to t$ と $b \to t$ であり，逆方向エッジは含まれない．また，s から t へ至るあらゆる道上でカット・エッジは 1 つである．

最小カットは，最大フローと同じ 4 である．これは，次節で証明するように，元のフロー・ネットワークにおいて逆方向カット・エッジを含まないカットのうちで最小のものである．

6.3.4　提案アルゴリズムの正しさと停止性

定理 2.　提案アルゴリズムによって得られるカットは，元のフロー・ネットワークのカットの中で，逆方向カット・エッジなし制約（単一カット・エッジ制約でも等価）を満たすもの があれば，それらのうちでサイズ最小のものである．

証明.　図 6.4 の上の列は，それぞれ，あるフロー・ネットワークのすべてのカットを，

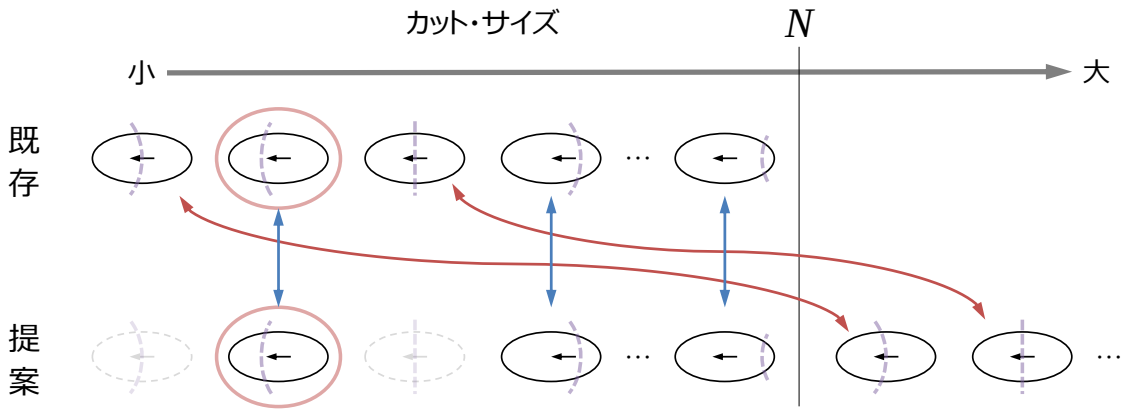


図 6.4: 既存（上）/提案（下）によって計算されたサイズによるカットの昇順列

それらのカット・サイズに従って昇順に並べたものである．図中のアイコンは，楕円は元のフロー・ネットワークを，矢印はエッジを，破線はカットを，それぞれ模式的に表している．特に，左向き矢印が破線に重なっている場合，この矢印は逆方向カット・エッジとなっている．

これらのカットに対して，仮に，提案手法の前処理を施した，すなわち，すべてのエッジに容量 N の逆方向エッジを追加した場合のカット・サイズを求めよう．下の列は，上の列のカットを，この新たに求めたカット・サイズの昇順に並べ直したものである．同じカットを異なるサイズに従って並べ直しただけであるから，上/下の列のカット間には，1 対 1 の関係がある．上下を結ぶ矢印は，この 1 対 1 の関係を示している．

これらのカットの中には，元のフロー・ネットワークにおいてカット・エッジに逆方向エッジを含むものと含まないものがある．その結果，下では以下のような順序の変化がある：

逆方向カット・エッジを含むカット 前処理を施した場合には，追加された逆平行エッジの容量 N が加算される．

N は，カット・サイズの最大値より大きい値としたことに注意されたい．したがって，N が加算されたカットは，下の列では最後尾に移動することになる．

逆方向カット・エッジを含まないカット 前処理を施した場合でも，追加された逆平行エッジの N の容量が加算されないため，カット・サイズは変化しない．その結果，下の列でもその位置に残されている．

前処理後に任意の最大フロー・アルゴリズムを適用する．

まず，その最大フロー・アルゴリズムの停止性が保証されているなら，提案アルゴリズム全体でも停止することは明らかである．

そして，その最大フロー・アルゴリズムが最小カットを見つけられるならば，図中，下の列において一番左，○を付けたカットが選ばれる．

ここで，逆方向カット・エッジを含まないカットのみに注目しよう．前処理を施した場合でもカット・サイズは変化しないため，それらの間の順序は上下の列の間で変化しないことが分かる（上下を結ぶ直線の矢印）．

したがって，提案アルゴリズムで得られた最小カットは，上の列においては，左から二番目，同じく○を付けて示されたカットと全く同一のものである．これは，元

のフロー・ネットワークのカットの中で逆方向カット・エッジを含まないもののうち，サイズ最小のものである．□

6.3.5　逆平行エッジ追加による計算量の変化

最大フロー・アルゴリズムは数多く存在する [40] が，6.4 節の実験では，より実用的な**エドモンズ・カープのアルゴリズム** (Edmonds-Karp algorithm) [41] を用いた．エドモンズ・カープのアルゴリズムの実行時間は，頂点数を V，エッジ数を E とすると，$O(VE^2)$ である [37].

エドモンズ・カープのアルゴリズムを用いた場合，提案では逆方向エッジを追加する前処理によってエッジ数 E は 2 倍になるので，実行時間は $2^2 = 4$ 倍になるように思われるが，実際にはそうではない．[42] の評価では，1.5 倍程度にとどまっている．それは，以下の理由による．

エドモンズ・カープのアルゴリズムをはじめ，フォード・ファルカーソンのアルゴリズムをベースとする最大フロー・アルゴリズムでは，残余ネットワークの逆平行エッジも平行エッジと区別なく扱われる．$O(VE^2)$ という実行時間は，この逆平行エッジを考慮に入れたものである．

提案における逆平行エッジは，通常なら増加操作によって随時追加されるものを，初期状態から追加するに過ぎない．したがって，残余ネットワークを用いる最大フロー・アルゴリズムを用いた場合，提案手法の実行時間はオーダ上は悪化しない．

特にプログラムにおいては，このような逆平行エッジは，必要に応じて追加/削除するのではなく，予めすべてのエッジに逆平行エッジを追加したうえで，その容量を 0 に初期化することによって実現することになろう．この場合，提案手法における前処理とは，逆平行エッジを追加するのではなく，この初期値を 0 から N に変更するに過ぎない．6.4 節のプログラムでも，そのように実装されている [43].

6.3.4 節で述べたように，提案手法ではよりサイズの大きいカットを探すことになる．そのためには，より多くの増加道を探索する必要がある．実際の実行時間の差は，このために生じると考えてよい．すなわち，ベースの最大フロー・アルゴリズムと提案手法の実行時間の差は，オーダ上のものではなく，トポロジ的には同一のフロー・ネットワークにおける容量の違いによるものである．

6.4 実験

6.1 節で紹介した逆相ラッチの挿入位置を求める問題を例として，提案アルゴリズムを適用した．

6.4.1 プログラム開発･実行環境

表 6.1 に，開発･実行環境をまとめる．

C#と，C# のグラフ用のライブラリ QuickGraph [44] を用いてプログラムを記述した．

6.3.5 節で述べたように，最大フロー･アルゴリズムとしては，エドモンズ･カープのアルゴリズムを用いた．エドモンズ･カープのアルゴリズムは，QuickGraph にも AlgorithmExtensions.MaximumFlowEdmondsKarp として含まれているが，インターフェイスが他の部分と合わなかったため，今回は自前で記述したものを用いた．記述にあたっては，[43] を参考にした．

6.4.2 実験対象

6.1 節で述べたように，回路を基にしたフロー･ネットワークに対して上記のプログラムを実行した．

回路としては，リプル･キャリー･アダーを用いた 32-bitカウンタと，RISC-V ISA [45] に準拠する 64-bit スカラ･プロセッサ Rocket [46] を用いた．

Vivado Design Suite 2016.3 を用いて Xilinx Artix-7 FPGA をターゲットにダウンロード可能なネットリストを入力とし，FF のデータ出力から FF のデータ入力に至

表 6.1: 開発･実行環境

CPU	Intel Core i7-4770, 3.40GHz
RAM	DDR3, PC3-12800, 8GB × 2
OS	Windows 10 Pro, Ver. 1703
C#開発環境	Visual Studio 2015
ビルド	Release（最適化あり，デバッグ出力なし）
FPGA 開発環境	Vivado Design Suite 2016.3
FPGA	Xilinx Artix-7

る連結な部分を 1 つのステージとして切り出し [14, 17]，ステージごとに提案アルゴリズムを適用した．

カウンタは，6.4.6 節で結果を詳しく見るためのもので，FPGA の持つハードウェア・キャリー・チェインを用いずに，ユーザ・ロジックでリプル・キャリー・アダーを構成した．

6.4.3　エッジの容量

逆相ラッチの挿入位置を求める問題に対して，エッジの容量は以下のように定めた [14, 17]．

評価基準　6.1 節で述べたように，逆相ラッチの挿入位置に関しては，以下の 2 つの評価基準がある：

1. ラッチの挿入個数は少ないほどよい．
2. クリティカル・パスを短縮するため，挿入位置はロジック内の各パスの遅延を等分することが望ましい．

より正確には，2. は以下のように修正される：

2. パスを正確に等分することが重要性は，そのパスの長さに依存する．

すなわち，クリティカル・パスでは，パスを等分することが重要である一方；非クリティカル・パスでは，そのパスの短さに応じて，1. 個数 の優先度が高くなる．

エッジの容量　そこで，エッジの容量は，パスを等分するラッチを 1 個として，等分しないラッチを C 個分と考えることにする．例えば，あるエッジの容量が 10 である時には，そのエッジを選択することで全体で 11 個のラッチが削減できるなら，そのエッジのずれは許容されることになる．

今回は，エッジの容量 C は，そのエッジを含むパスのうち最長のものに対して，以下のヒューリスティックな評価関数を用いた：

$$C(d,p) = B(p)^{10d}$$
$$B(p) = (N-n)p^M + n$$

d クリティカル・パス長で正規化した，最も中央に近い挿入位置からのずれ．

p クリティカル・パス長で正規化した，そのエッジを含むパスのうち最長のもの長さ．

N クリティカル・パスにおいて，クリティカル・パス比 10% のずれがラッチ何個分に相当するか．今回は 10.

n 長さ 0 の仮想的なパスにおいて，クリティカル・パス比 10% のずれがラッチ何個分に相当するか．今回は 2.

M 非クリティカル・パスにおいて，N の大きさの効果を緩和する度合いを表すパラメタ．今回は 1.5.

長さは，ユニット遅延でも実遅延でもよい．今回は，ユニット遅延を用いた．

この関数は，以下のように，前述の評価基準を満たす．

まず $C(d,p)$ は，$B(p)$ を底として，ずれ d に対して指数関数的に増加する．

底 $B(p)$ は，そのエッジを含むパス（のうち最長のもの）の長さによって，$(n,N]$ の範囲で変化する．エッジがクリティカル・パス上にある $(p=1)$ 場合，$B(1)=(N-n)\times 1^M+n=N$；一方，長さ 0 の仮想的なパス上にある $(p=0)$ 場合，$B(0)=(N-n)\times 0^M+n=n$ となる．

そして C は，最も中央に近い位置に挿入された場合 $(d=0)$，$C(0,p)=B(p)^0=1$ と，p に関わらず 1 となる．逆に中央からずれた場合には，パスの長さによって以下のように変化する：

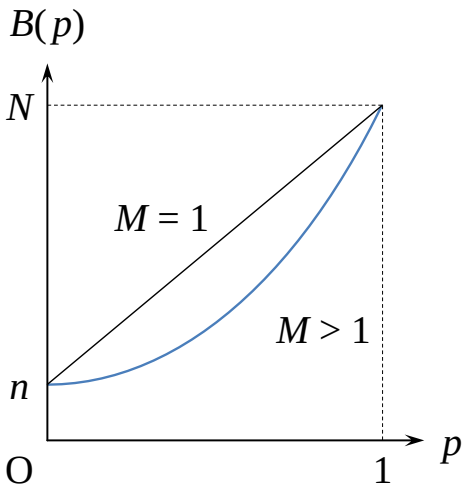


図 6.5: $B(p)$ のグラフ

- クリティカル・パス上 ($p = 1$，$B(1) = N$) では，10% ずれた ($d = 0.1$) 場合，$C(0.1, 1) = N^{10\times0.1} = N$；最もずれた ($d = 0.5$) 場合，$C(0.5, 1) = N^5$ となる．ずれ d に対して指数関数的に増加する結果，クリティカル・パスにおいては，中央からのずれが許容されにくくなる．

- 仮想的な長さ 0 のパス上 ($p = 0$，$B(0) = n$) では，同様に，10% ずれた場合，$C(0.1, 0) = n$，最もずれた場合，$C(0.5, 0) = n^5$ となる．ずれ d に対して，同じく指数関数的に増加はするものの，$n < N$ であるだけ増分は抑えられ，ずれが許容されやすくなる．

M は，非クリティカル・パスにおいて，N の大きさの効果を緩和する．図 6.5 に，$B(p)$ のグラフを示す．M を 1 より大きくすると，p が 1（クリティカル・パス）よりわずかに短くなるだけで $B(p)$ が大きく減少することになる．結果，クリティカル・パスよりわずかに短いパス上において，中央からのずれが許容されやすくなり，逆により大きい N の採用を可能にする．

なお，1. 個数 と 2. 遅延 の，いずれを優先すべきかは対象によって異なるため，ユーザがパラメタを適切に調整する必要がある．上記の評価関数の場合，パラメタ N，n，M のうち，一次的には N によって調整することができる．N を大きくすれば，2. 遅延 がより優先される．

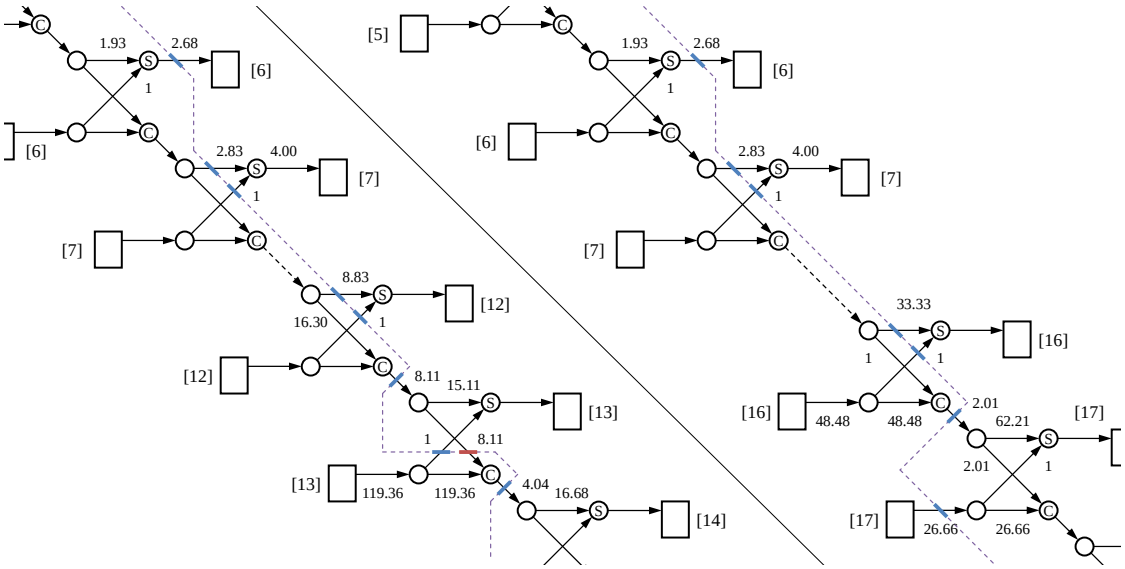


図 6.6: ベース（左）と提案手法（右）によって得られたカウンタのフロー・ネットワーク

6.4.4 実験結果

表 6.2 に結果を示す．同表中，ベースは，前処理を施さずにエドモンズ･カープのアルゴリズムを適用した結果である．Rocket については，最大のステージの結果のみを示す．

まず，当然のことではあるが，提案手法では逆方向カット･エッジ数が 0 となっている．

実行時間は，Rocket の場合でも，ベースの 1.5 倍程度にとどまっている．表 6.2 に示した Rocket のステージは，全ステージの中でも突出して巨大なもの（除算器の一部）で，その他のステージはすべてその 1/10 程度以下に過ぎない．このような巨大なステージに対しても，約 375 秒という実用的な時間で最適解が求められることが分かる．

6.4.5 実行時間に関する考察

前述したように，実行時間は，Rocket の場合でもベースの 1.5 倍程度にとどまっている．6.3.5 節で述べたように，エドモンズ･カープのアルゴリズムの実行時間 $O(VE^2)$ は逆方向エッジを考慮に入れたもので，逆方向エッジを追加する前処理によって実行時間は $2^2 = 4$ 倍にはならない．

特にプログラムにおいては，この逆方向エッジは，必要に応じて追加/削除する

表 6.2: 実験結果

	カウンタ		Rocket	
	ベース	提　案	ベース	提　案
頂点数	190		34,347	
エッジ数	281		96,795	
順方向カット･エッジ	45	43	6,318	8,301
逆方向カット･エッジ	6	0	3,287	0
カット･サイズ	146.7	262.8	9,642.0	12,503.7
実行時間 [sec]	0.0143	0.0147	240.0	375.3

のではなく，予めすべてのエッジに逆方向エッジを追加したうえで，その容量を 0 に初期化することによって実現することになろう．この場合，提案手法における前処理とは，逆方向エッジを追加するのではなく，この初期値を 0 から N に変更するに過ぎない．実際，今回のプログラムでは，そのように実装されている [43].

6.3.4 節で述べたように，提案手法ではよりサイズの大きいカットを探すことになる．そのためには，より多くの増加道を探索する必要がある．実際の実行時間の差は，このために生じると考えてよい．すなわち，ベースの最大フロー・アルゴリズムと提案手法の実行時間の差は，オーダ上のものではなく，トポロジ的には同一のフロー・ネットワークにおける容量の違いによるものである．

6.4.6　カウンタの詳細な結果

図 6.6 に，カウンタに対してベースのエドモンズ・カープのアルゴリズムと提案手法を適用して得られた回路のフロー・ネットワークを示す．同図中，四角は FF を，Ⓢ と Ⓒ は，部分和とキャリーを求める部分回路を，◯ は複数出力のためのダミー（6.1 節参照）を，それぞれ表す頂点である．カウンタであるため，左右，対となる FF は，物理的には同一の FF に対応する．そして，破線は求められたカットを表す．

双方において，第 0〜6 ビットまでは，Ⓢ の出力側に，カット・エッジがあるのに対して；第 7 ビット以降では，Ⓢ の（出力側ではなく）入力側にカット・エッジがある．これは，下位からのキャリーの遅延が徐々に長くなり，第 7 ビット以降では，入力側に 2 つのラッチを置いた方が容量の和が小さくなるためと考えてよい．

ベースでは，第 13（図中）と 14〜18 ビット（図外），計 6 本の逆方向エッジが現れている．また，クリティカル・パスであるキャリー・チェイン上に複数のラッチが挿入される．

一方，提案では，逆方向エッジは現れず，キャリー・チェイン上には，第 16 ビットの次にラッチがただ 1 つ挿入される．

6.5 本章のまとめ

FF を用いた回路からラッチを用いた回路に変換する際などには，単一カット・エッジ制約，すなわち，始点から終点に至るすべての道にカット・エッジを 1 つ含むという制約を満たすカットを見つける必要がある．

本稿では，まず，単一カット・エッジ制約が，逆方向カット・エッジなし制約，すなわち，カットが逆方向カット・エッジを含まないことと等価であることを証明した．その上で，逆方向カット・エッジを含まない最小カットを求めるアルゴリズムを提案し，その正しさを証明した．

このアルゴリズムにおいて最もオーダが大きい部分は既存の最大フロー・アルゴリズムであり，提案アルゴリズムのオーダはこれより悪化することはない．

現実的な回路から生成されたグラフに対して提案手法を適用したところ，巨大な回路に対しても約 375 秒という実用的な時間で最適解が求められることが分かった．

第7章

RazorのRocketへの適用

7.1 本章の内容

本章では，Razorを Rocket [46] に適用し，それを FPGA 上に実装する方法について述べる．Rocket は RISC-V ISA に準拠するスカラ･プロセッサである [45]．Rocket は，**CSR** (Control and Status Registers) を持ち，Linux をブートすることができる．著者が調査した限り，このような現実的なプロセッサに対して TF 検出を適用した事例はない．本論文の Razor 化 Rocket は，FPGA 上に実装され Linux をブートできるものとしては，TF 検出最初のテスト･ベッドとなるであろう．

しかし，Razorを Rocket に適用することは容易くはなかった．それは，Rocket は言わば「out-of-order スカラ･プロセッサ」だからである．Rocket のパイプラインは，一部の命令に対してアーキテクチャ･ステートの out-of-order な更新を許す．それらの命令とは，具体的には，キャッシュ･ミスを伴うロード命令，整数乗除算命令，そして，浮動小数点除算・開平命令など，長いレイテンシを持つものである．これらの長いレイテンシを持つ命令に引き続く命令は，依存しなければ，先にアーキテクチャ･ステートを更新することができる．

この実装は安全である．なぜなら，これらの長レイテンシ命令は，例外を引き起こさないとされているからである．これらの命令は，いずれアーキテクチャ･ステートを更新し，その時点でアーキテクチャ･ステートは in-order な状態となる．

しかしながらこの想定は，Razor を適用する場合には当てはまらない．これらの長レイテンシ命令は，例外を起こさないかもしれないが，TF を起こすからである．したがって，Razor を適用するにあたって，通常必要となる変更に加えて，このアー

キテクチャ・ステートの out-of-order 更新を無効化する必要があった．

本章の構成は以下のとおりである：まず 7.2 節で，Rocket のマイクロアーキテクチャについてまとめる．7.3 節 では，アーキテクチャ・ステートの out-of-order 更新に注目しながら，Razor の Rocket への適用方法について説明する．7.4 節では，評価結果についてまとめる．

7.2　Rocket のマイクロアーキテクチャ

7.3 節において Razor を Rocket に適用する方法について説明するため，本節では Rocket のマイクロアーキテクチャについてまとめる．

Rocket は，RISC-V ISA の RV64G variant [45, 46] を実装する．すなわち Rocket コアは，整数 ALU・乗除算器に加えて FPU を持つ．

Rocket は（スーパスカラではなく）スカラ・プロセッサであるが，out-of-order 実行機構を備え，1 次データ・キャッシュ (**L1D**) ミスを伴うロード命令や可変長命令の長いレイテンシの隠蔽を図る．

7.2.1　パイプライン構成

図 7.1 に，Rocket のパイプライン構成を示す．基本的には Rocket は，いわゆる (full) 5-stage pipeline を持つスカラ・プロセッサである．本章では，これら 5 つのステージを，F: 命令フェッチ，D: デコードとレジスタ・ファイル (**RF**) からの読み出し，E: 実行，MR: メモリ読み出し，W: ライトバックと呼ぶ．

ただし以下のタイプの命令に対しては，専用のパイプラインが設けられている：

AMO いわゆる fetch-and-add のようなアトミック・メモリ・オペレーション (Atomic Memory Operation: **AMO**) に対しては，read-modify-write に 1 対 1 に対応する 3 ステージが専用に設けられている．同図中，これらのステージは以下のとおりである；E: アドレス計算，MR: L1D からの読み出し，MM: modify 操作の実行，MW: LID への書き込み．

FP 主要な浮動小数点命令に対しては，固定長のパイプラインが用意されている．ステージは以下のとおり；FR: FP RF の読み出し, FEF・FES: 実行，FW: FP RF への書き込み．

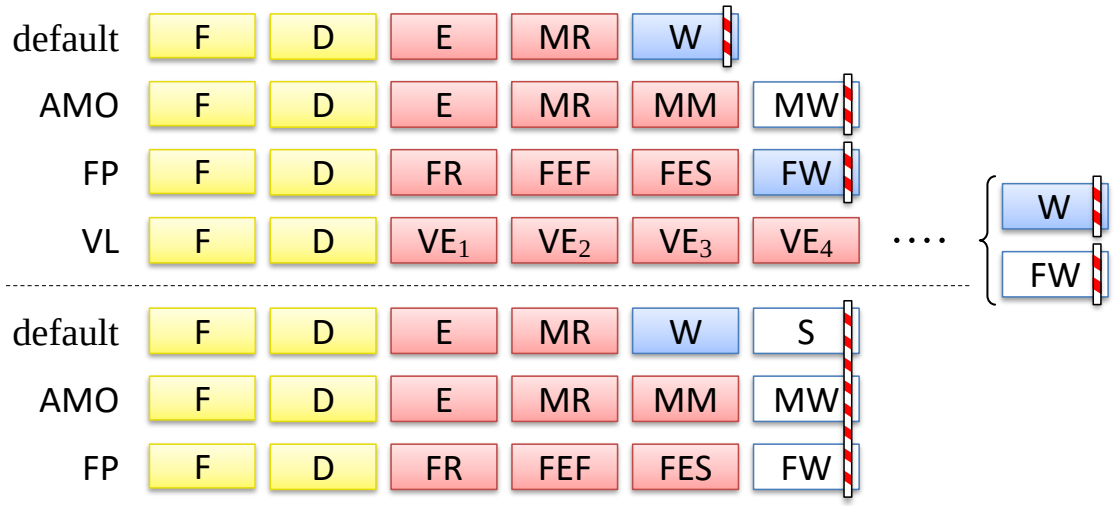


図 7.1: Rocket のパイプライン：変更前（上）と変更後（下）

VL 整数 MUL/DIV，FP FDIV/FSQRT 命令に対しては，可変長のパイプラインが用意されている．ステージは以下のとおり；VE_1，VE_2，...: 実行，W/FW: 対応する RF への書き込み．

同図中，同じラベルを付されたステージは物理的に同一である．特に，L1D に対するすべての読み/書きは，MR/MW において行われる．したがって，（AMO ではない普通の）ロード命令/ストア命令は，default/AMO パイプラインで，それぞれ実行されることになる．

また同図中，赤白ストライプのバリアで示されているように，Rocket の実装では，アーキテクチャ・ステートの更新を，W/MW/FW ステージの間ではなく，終わりのエッジにおいて行う．このことは，7.3.4 節で述べるスタビライズ・ステージの挿入において重要な意味を持つ．

7.2.2 Out-of-Order 実行

Rocket の L1D は，ノンブロッキング・キャッシュであり，複数の L1D ミスに対するメモリ・アクセスをオーバーラップ実行することができる．Rocket の out-of-order 実行機構は，L1D ミスを起こしたロード命令や VL 命令の長いレイテンシの隠ぺいを図る．これらの命令を **LL** 命令と呼ぶことにする．LL 命令に後続の命令は，依存しない場合には，実行を継続し，その結果を RF に書き込むことさえできる．

Rocket はスカラ・プロセッサであるから，構造ハザードを回避するため，基本的には LL 命令も他の命令と同様に命令パイプラインの中を進むことになる．したがって，後続の命令が先に進むためには，先行する LL 命令が道を譲らなければならない．

そのため，LL 命令を実行する各 ユニットは，追い越しのための**キュー**を持つ．これらのキューは，out-of-order スーパスカラ・プロセッサにおける命令キューとは異なり，以下のように働く：

- LL 命令は，ライトバック・ステージの終わりのエッジにおいて，一旦，対応するキューに移される．そのため，後続の命令は，この LL 命令より先に，ライトバック・ステージに進むことができる．
- この LL 命令は，結果を得た後にこのキューからライトバック・ステージへと戻され，**サイクル・スチーリング**によって，結果を RF に書き込む．

7.2.3　ハザードの解決

上述した out-of-order 実行機構のため，Rocket がどのようにハザードを検出し，解決するかは，表 7.1 に示すように，命令のタイプによって異なる．

検出

通常のスカラ・プロセッサでは，パイプライン内の命令のソースとデスティネーションのレジスタ番号を比較することによってハザードの検出行われる．それに対して Rocket では，LL 命令がキューへと移されるため，この方法だけでは不十分である．そのため，各キューに**スコアボード**が用意されている．

Rocket におけるスコアボードは，対応する RF エントリに書き込みを行う LL 命令があるかどうかを示す 1 ビットのフラグのテーブルである．LL 命令がキューに移され

表 7.1: Rocket におけるハザードの検出と解決

Preceding Instruction	Solution	Detection for Interlock	
		in Pipeline	in Queues
default	Interlock	Comparators for Interlock	
VL			Scoreboards
load w/ L1D miss	Interlock & Replay on L1D miss		

るとき，また，キューからパイプラインに戻されるときに，そのデスティネーションに対応するフラグがセット/リセットされる．すなわち，フラグがセットされている RF エントリは，対応する LL 命令がまだ結果を書き込んでいないため，not ready である．

したがって後続の命令は，D ステージにおいて，以下の 2 つの方法によってハザードを検出する：

インターロック用比較器 後続の命令は，そのソースと，パイプライン内にある先行する命令のデスティネーションとを比較する．

スコアボード 後続の命令は，そのソースに対応するスコアボードのフラグをチェックする．(それに加えて，後続の命令は，そのデスティネーションに対応するフラグもチェックし，同一ユニットに対する出力依存を解決する．)

解決

インターロックに加えて，Rocket は L1D ミスに対しては**リプレイ**も用いる：

インターロック ほとんどの命令に対しては，Rocket は通常のインターロックを用いる．上述したように，ハザードは，後続の命令が D ステージにいるときに検出される．したがってインターロック機構は，F と D ステージのみを停止することになる．

リプレイ ロード命令が L1D ミスを起こしたときには，D ステージより先に進んでしまった後続の命令をリプレイする．

7.2.4 Out-of-Order 実行とタイミング故障検出

Out-of-order 実行は性能向上をもたらすが，Rocket の実装は，そのままでは TF 検出に用いることはできない．

図 7.2 に，問題となる振る舞いを示す：

C_1 LL 命令 I_1 は，レジスタから r2 $= 1$ を読む．

C_2 Rocket は，I_1 を含む先行する命令に依存していないため，I_2 が先に進むことを許す．

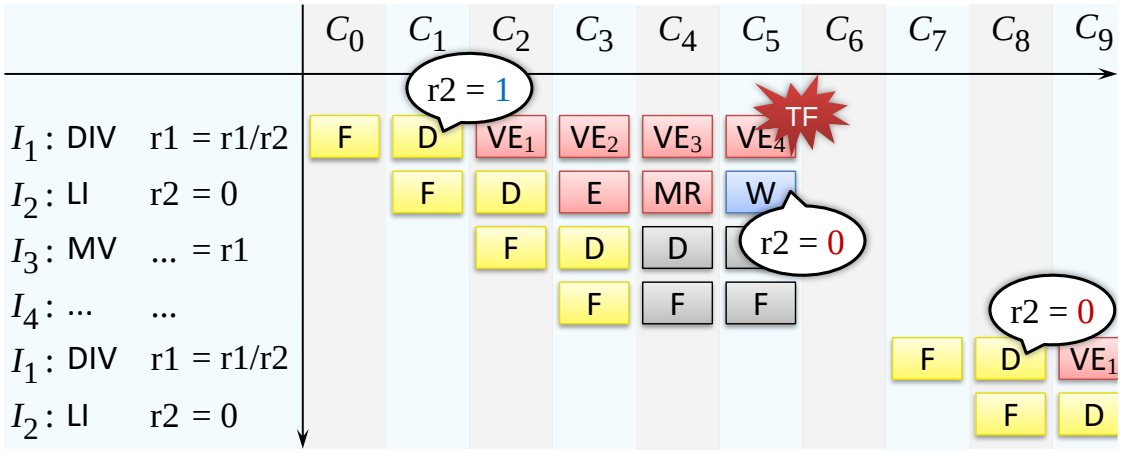


図 7.2: アーキテクチャ・ステートの Out-of-Order 更新による問題

C_3 一方で I_3 は，r1 を通して I_1 に依存しているため，実行ステージに進むことができない．

C_4 そこで，インターロック機構は，F と Dステージを停止し，I_3 と I_4 はそこに留まる（図中，灰色のステージ）．

C_5 I_2 は，out-of-order に，すなわち，I_1 が RF を更新する前に，r2 $=0$ に更新する．

同じサイクルに，TF が検出される．この TF は，I_1 を実行するステージ内で発生した可能性があり，I_1 は間違っている（かもしれない）結果によって RF を更新することはできない．

C_6 したがって，パイプラインはフラッシュされ，…

C_7 I_1 のフェッチからやり直す．

C_8 すると I_1 は，I_2 によって更新された r2 $=0$ を読むことになる．

Rocket の実装では，この out-of-order な更新は問題とならない．なぜなら，I_1 は例外を起こすことなく RF を更新すると仮定されているからである．しかしながら，この仮定は TF 検出を行うプロセッサには当てはまらない．なぜなら，I_1 は，TF によって RF を更新できない可能性があるからである．

TF 検出を Rocket に適用するためには，out-of-order スーパスカラ・プロセッサのような命令のリオーダリングを実現するか，7.3 節 で述べるように，アーキテクチャ・ステートの out-of-order 更新を無効化する以外にない．

7.3 Razor の Rocket への適用

本節では，Razor を Rocket に適用する方法について述べる．7.2 節 の議論に基づいて，Rocket のマイクロアーキテクチャを変更した．パイプライン・レジスタを Razor FF に置き換えることに加えて，以下の項目を実施した：

1. アーキテクチャ・ステートの out-of-order 更新の無効化
2. アーキテクチャ・ステートの特定
3. 投機状態と非投機状態の分離
4. パイプラインの変更
5. エラー通知ネットワークの追加
6. パイプライン再初期化の追加

以下，それぞれの項目について述べる．

7.3.1 アーキテクチャ・ステートの Out-of-Order 更新の無効化

7.2.3 節で述べたように，Rocket は，依存しない場合には，先行する LL 命令より前に後続の命令がアーキテクチャ・ステートを更新することを許す．したがって，あたかも依存しているかのように扱えば，この状況を避けることができる．例えば，図 7.2 において，もし I_2 が I_1 に依存してい（るかのように扱われ）れば，I_2 は D ステージで停止し，アーキテクチャ・ステートは out-of-order に更新されることはない．

表 7.2 に示すように，ロジックがハザードを検出する条件は，以下のように変更される：

変更前 レジスタ番号が一致すれば．

表 7.2: 更新前と更新後のハザード検出の真理値表

Preceding Instruction LL ?	Register Numbers Match ? 0	1	0	1
		Original		Modified
0		1		1
1		1	**1**	1

変更後 レジスタ番号が一致するか，レジスタ番号の一致/不一致にかかわらず，先行する命令が LL であれば．

この変更の結果，VL パイプラインは，アーキテクチャ・ステートの更新に関して考慮する必要がなくなる．

7.3.2　アーキテクチャ・ステートの特定

整数/FP RF に加えて，アーキテクチャ・ステートには，**CSR** (Control and Status Registers) と **next PC** が含まれる．

Next PC は，TF 時にプログラムを再開する命令の PC である．

RISC-V は，CSR として例外 PC (epc) を定義している [47]．Rocket は，epc を例外発生時にのみ更新する．この epc を我々が必要とする next PC に変更することは不可能ではないが，安全のため，毎 サイクル更新される「本当の」next PC を追加することにした．

7.3.3　投機状態と非投機状態の分離

3.2 節で述べたスタビライズ・ステージを挿入するにあたっては，アーキテクチャ・ステートの更新を停止することに加えて，最新の値が読めることを考慮しなければならない．例えば，CSR に書く/読む 2 つの命令 CSRW epc, rs1; と CSRR rd, epc; が連続して実行される場合，先行する命令によって書かれた値を，後続の命令は読む必要がある．もしレジスタへの書き込みを単に 1 ステージ遅らせただけでは，更新が 1 サイクル遅れ，最新の値を読むことができなくなる．

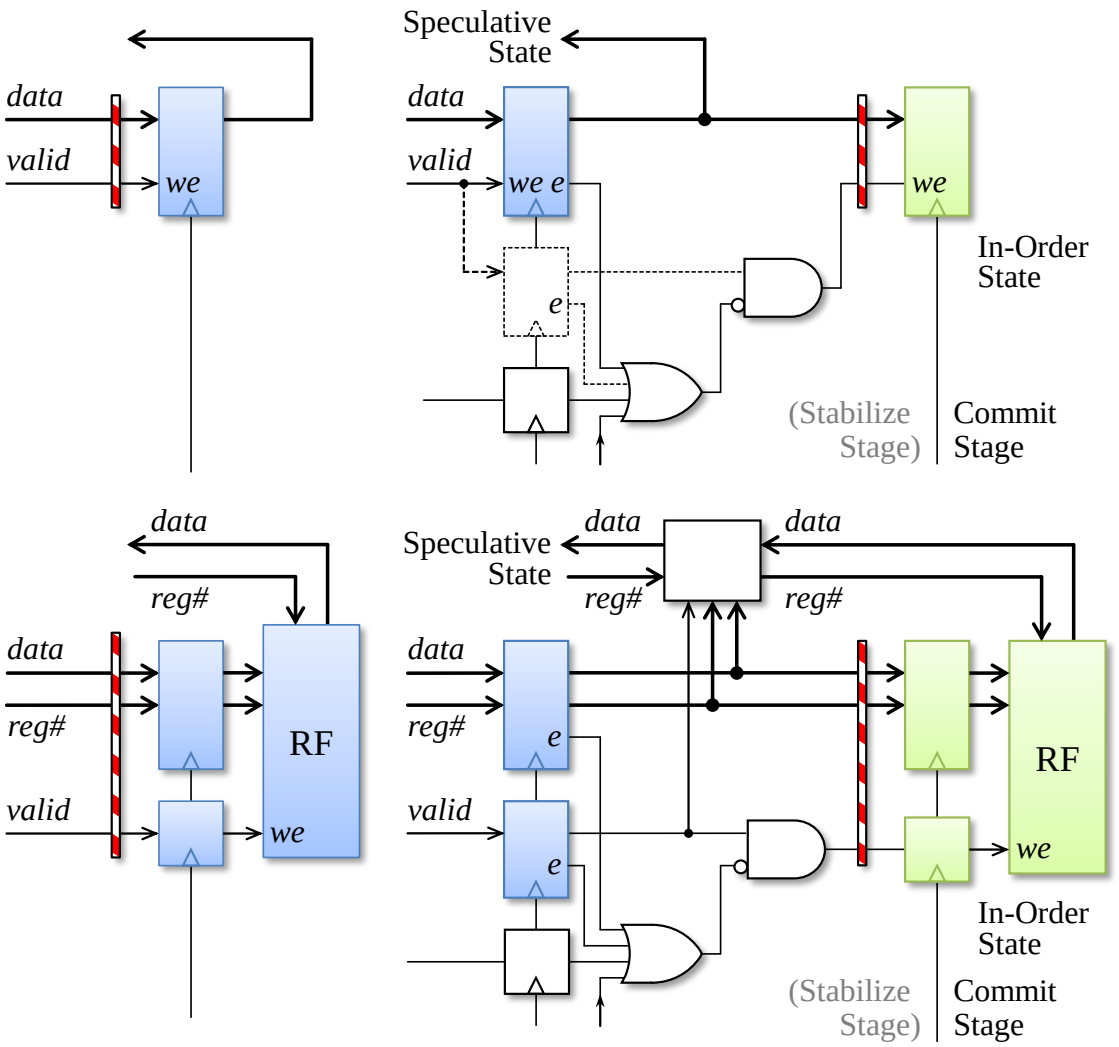


図 7.3: 変更前（左）と変更後（右）のアーキテクチャ・ステート・レジスタ（上）とレジスタ・ファイル（下）

この観点からは，スタビライズ・ステージを挿入すると考えるより，通常の out-of-order スーパスカラ・プロセッサと同様に，投機状態と非投機状態を分離すると考える方が都合がよい．

図 7.3（上）に，この考え方を示す．右側では，**非投機状態 (in-order state)** を保持するレジスタが追加されている．元のレジスタは**投機状態 (speculative state)** を保持することになる．元の投機レジスタは，間違っているかもしれない結果によっても更新され，最新の値を提供する．投機レジスタの値は，TF が検出されなければ，毎 サイクル，非投機レジスタへとコミットされる．

非投機レジスタは，TF からの回復に際してのみ読み出される．また，回復のためのパス以外は，元のレジスタの周辺の回路は変更する必要がない．

同図（下）に，同じ考え方をレジスタ･ファイルに適用した場合を示す．レジスタ･ファイル全体を複製することは高コストなので，右側では，レジスタ･ファイルは 1 ステージ下流に移動され，投機的だが最新の値を提供するためのバイパス回路が付加される．

この考え方は，元々コミット･ステージを持つ「本当の」out-of-order スーパスカラ･プロセッサに TF 検出を応用するとき，より一層重要となる．

7.3.4　パイプラインの変更

Rocket は，投機･非投機状態に関連して，以下のような，あまり一般的ではない特徴を持つ：

- 7.2.1 節で述べたように，アーキテクチャ･ステートは，W/MW/FW ステージの間にではなく，終わりのエッジにおいて更新される．
- 図 7.1 においてそれぞれ色付き/空白の矩形で示されるように，W ステージには（ライトバックのためのではなく）例外処理などのためのロジックがある一方，MW と FW ステージはほとんど空である．

したがって，同図（下）に示されるようにパイプラインを変更した．同図中，S はスタビライズ･ステージで，C/MC/FC はそれぞれ W/MW/FW に対するコミット･ステージである．

この結果，以下のようにして，アーキテクチャ･ステートの in-order 更新が保証される：

- 7.3.1 節で述べたように，VL パイプラインは，考慮する必要がなくなる．
- 図 7.1 に示されるように，残り，すなわち，default，AMO，FP パイプラインは，同じ長さの固定長パイプラインとなる．

7.3.5 エラー通知ネットワーク

我々が以前提案したように，エラー信号は，TF を起こした命令とともにパイプラインを進む必要はなく，TF によって誤っている可能性のある結果と，同時かより先にライトバック・ステージに到着すればよい [28].

この考え方にしたがって，ネットワークは，マイクロアーキテクチャ・レベルのステージとは関係なく構築した．より具体的には，回路レベルにおいて，FF 数で数えたライトバック・ステージまでの最短経路に基づいて構築される．

7.3.6 パイプライン再初期化

TF の影響をパイプラインから取り除くためには，パイプライン・フラッシュでは不十分で，特に out-of-order スーパスカラ・プロセッサの場合には，パイプラインの**再初期化** (re-initialization) が必要であると我々は主張してきた [28]．Rocket のようなスカラ・プロセッサに対しては，パイプライン・フラッシュで十分である可能性もあるが，安全のため，そして，将来のため，パイプライン再初期化を選択した．

プロセッサ全体の**リセット木**を，アーキテクチャ・ステートとそれ以外用の 2 つの部分気に分割し，TF に際にしては後者のみを活性化する．

リセット木の負荷をバランスさせるため，我々はパイプラインに沿ったステージごとの再初期化も提案してきた [28]．しかし今回は，クリティカルではなかったため，Rocket が元々備える 1 サイクルでのリセットを再利用することとした．

7.4 評価

Rocket の適用による回路オーバーヘッドを FPGA において評価した．表 7.3 は開発とテスト環境をまとめたものである．

表 7.3: 開発とテストの環境

Platform	lowRISC SoC project [48]
Synthesizer	Synplify Premier with DP J-2014.09-SP1
FPGA Design Tool	Xilinx Vivado Design Suite, Ver. 2017.4
FPGA Board	Nexys 4 DDR Artix-7
FPGA	Xilinx Artix-7 XC7A100T-1CSG324C

7.4.1 モデル

次の三つのモデルについて評価を行った：

Base 最初に，単精度・倍精度小数点 fused-multiply-add ユニット（SFMA/DFMA）が長いことによるオリジナルな Rocket の不均衡なステージについて修正し，ステージを均衡させた．具体的には，元の DFMA には，32.9 ns であり，これは FPU 以外の整数コアのクリティカル・パス（13.5 ns 以下）よりもはるかに長かった．Chisel コードを変更して，SFMA を DFMA と同じ 3 サイクルのレイテンシとし，SFMA/DFMA において FF を移動してから，Synplify の retime オプションを併用することで，FF 挿入位置を移動した．その結果，SFMA/DFMA のクリティカル・パス遅延は 13.5 ns に削減できる．

Stabilized 次に，AS の out-of-order 更新を無効にし，7.3.4 節で説明されているように，スタビライズ・ステージをデザインに追加した．RISC-V ISA テストによりベースとこのモデルを検証した．

Razored 最後に，Razor を適用した．Base モデルから 10% のクロック・サイクルの改善を目標とする；この場合，システムのクリティカル・パスの 90% よりも長い遅延を持つパスが TF を引き起こす可能性がある．これらのパスの終端を Razor FF に変更した．これらのパスの終端が RAM モジュール内にある FF である場合，TF を検出するために，RAM モジュールの入力ポートに接続されている信号を Razor FF に接続した．この Razor FF のメインFF の出力は使用されないが，エラーは使用される．また，[21] に述べたように，この終端に至るショート・パスに遅延要素を挿入することでショート・パス問題を解消した．遅延要素としては，LUT1 を使用した．

7.4.2 結果

表 7.4 は，プロセッサコアと Ł1D を含むモデルの RocketTile モジュール用の FPGA のリソース使用率を示す．

Stabilized モデルでは，ステージを追加しても，リソース使用率が 6% 以下の増加であった．

Razored モデルでは，Razor FF の数は 110 であり，遅延要素の LUT1 の数は 3,645 であった．したがって，増加したリソースのほとんどは遅延要素である．このモデルの FF の増加はごくわずかである．

7.5 本章のまとめ

本章では，Razor を Rocket に適用した．元々の Rocket は，アーキテクチャ・ステートの out-of-order な更新を許していたため，そのまま Razor を適用することはできなかった．本章では，out-of-order な更新を無効化する方法を示した．

表 7.4: リソース使用量

Models	Slice LUTs	Slice Regs	Muxes	Block RAMs	DSP Units
Base	25,137	13,093	893	14	25
Stabilized	25,430	13,608	1,127	↑	↑
Razored	29,129	13,776	1,127	↑	↑

第8章

結論

微細化によるばらつき増大の問題への対策として，回路・アーキテクチャレベルでは，Razor を代表とした手法が提案されていた．Razor は，実質的にクリティカル・パスの活性化率を考慮すると，回路マージンを削る効果はあるものの，その潜在的な検出限界に比べると動作点を大きくすることができていなかった．これに対して，我々は，二相ラッチと Razor を組み合わせた新しいクロッキング方式である動的タイム・ボローイングを可能にするクロッキング方式を提案してきた．動的タイム・ボローイングを可能にするクロッキング方式はステージ間でワースト・ケース遅延ではなく実効遅延を融通することができると考えられる．

動的タイム・ボローイングを可能とする方式は我々が提案してきたものであるが，ごく簡単な回路によって最低限の動作確認がされただけであった．実用化のためには，最終的には，Out-of-Order プロセッサなどの現実的な回路に対して適用した上で，LSI 化し，評価を行う必要がある．また，既存の回路を入力として，提案クロッキング方式が適用された回路を出力する自動変換ツールの開発が不可欠であった．

8.1 本論文のまとめ

本論文では，その最終段階までには至ってはいないが，そのための重要なステップとして，以下を行った：

1. SRAM を対象としたタイミング故障検出手法
2. 簡単な回路に対する自動変換と方式の評価

3. 二相ラッチ化手法

4. Razor の Rocket への適用

以下，それぞれについて述べる：

1. SRAM を対象としたタイミング故障検出手法　現実の回路においては，組み合わせ回路に加えて SRAM も欠くべからざる要素である．そして SRAM の読み出し回路は，ダイナミック･プリチャージ･ロジックとして実装されることが多い．にもかかわらず，Razor をはじめとする既存の TF 検出・回復手法は，専らスタティック･ロジックのみを対象としており，ダイナミック･プリチャージ･ロジックへの適用については言及すらされていなかった．

本論文では，ダイナミック･プリチャージ･ロジックとして，特に SRAM を対象とした TF 検出手法を提案した．提案手法は，メインのサンプリング時点でのビットラインの状態に応じてプリチャージを制御するものである．提案によって，サイクル･タイムを評価とプリチャージのワースト･ケース遅延の和よりも短縮することができる．提案による付加回路の面積は，レジスタ･ファイルや L1C に用いられる RAM の全体の面積に比べ小さい．この提案手法により，TF 検出技術がプロセッサ内の主要なコンポーネントであるレジスタ･ファイルや L1C に対しても適用可能となったことで，これらがボトルネックとなり得る問題を解消した．

なお，この手法は，動的タイム･ボローイングを可能とするクロッキング方式に限らず，一般の TF 検出・回復手法にも適用可能である．

2. 簡単な回路に対する自動変換と方式の評価　自動変換ツールのフレームワークを確立した．リプル･キャリー･アダーを用いたカウンタを対象として，TF 検出と回復のための回路を付加した回路に自動変換した．出力された回路を FPGA に実装して評価し，少なくとも簡単な回路に対しては提案クロッキング方式が想定した効果を発揮することを確認した．

なおこの際には，次で述べる二相ラッチ化は手動で行っている．

3. 二相ラッチ化手法　二相ラッチに基づくパイプライン設計手法として，単相 FF を用いてデザインされた回路を自動的に二相ラッチを用いた回路に変換するアルゴリズムを提案した．

単相 FF を用いてデザインされた回路をラッチを用いた回路に変換する問題は，最小カット問題の一種に帰着する．ただしこの際，始点から終点に至るすべての道にカット・エッジをただ 1 つ含むという制約がある．既存の最小カット・アルゴリズムでは，この制約を満たすことができない．本論文では，この制約が，カットが逆方向カット・エッジを含まないことと等価であることを証明し，逆方向カット・エッジのない最小カットを見つけるアルゴリズムを提案した．このアルゴリズムにおいて最もオーダが大きい部分は既存の最大フロー・アルゴリズムであり，提案アルゴリズム全体のオーダはこれより悪化することはないことを示した．実験により，ゲート数約 3.4 万，配線数約 9.7 万程度の回路に対しても，約 375 秒の実用的な時間で最適解が求められることが分かった．

4. Razor の Rocket への適用　動的タイム・ボローイングを可能とするクロッキング方式を適用する前段階として，Razor を現実的なプロセッサ Rocket に適用する方法を示した．Rocket は，RISC-V アーキテクチャに完全準拠するスカラ・プロセッサで，CSR (Control and Status Registers) を持ち，Linux をブートすることができる．著者が調査した限り，このような現実的なプロセッサに対して TF 検出を適用した事例はない．本論文の Razor 化 Rocket は，FPGA 上に実装され Linux をブートできるものとしては，TF 検出最初のテスト・ベッドとなるであろう．

Rocket はある種の命令に対して Out-of-Order 完了を許すプロセッサである．TF からの回復のためには In-Order 完了を必要とするため，Rocket の完全な In-Order 化が必要となった．

8.2 今後の課題

動的タイム・ボローイングを可能にするクロッキング方式の実用化のためには，最終的には，Out-of-Order プロセッサなどの現実的な回路に対して適用した上で，LSI 化し，評価を行う必要がある．そのためには，次のような課題がある：

Rocket に適用して評価　本論文で提案した，二相ラッチ化手法と，Rocket への Razor への適用を利用し，動的タイム・ボローイングを可能にするクロッキング方式の適用を行う．FPGA を対象とした評価を行い，実際にどの程度クロック周波数の高速化され，実時間が短縮されるかを評価する．

Out-of-order スーパスカラ・プロセッサに適用して評価　動的タイム・ボローイングを可能にするクロッキング方式の適用を out-of-order スーパスカラ・プロセッサに対して行う．現在，NORCS [50] など様々な技術を取り入れた高効率な out-of-order スーパスカラ・プロセッサである雷上動の開発が行われており [49]，これを対象とする予定である．

Out-of-order スーパスカラ・プロセッサに関する適用のポイントは，[28,51] で考察されているが，この適用によって実際に適用する際の課題が明らかにされる．また，スカラ・プロセッサである Rocket との実効遅延の変動の違いや，タイミング故障からの回復におけるペナルティの差異がどのように性能に影響するかを明らかにする．

雷上動における評価では，FPGA のみならず，最終的には LSI 化も視野に入れている．LSI 化における評価では，本論文での貢献である SRAM への TF 検出もまた適用することができる．Razor における最小遅延制約を満たすための遅延は，FPGA においては遅延素子でなされてたが，LSI チップを作成する場合はゲート幅を小さく，あるいはゲート長を長くするなどのアプローチがとれるため，回路面積の影響が FPGA に比べて小さく済むと考えられる．この評価においては，FPGA の LUT のような回路内の素子が均等な遅延を持ち配線遅延が大きく占める回路とは異なるため，その影響の差異を明らかにする．

参考文献

[1] 平本俊郎，竹内　潔，西田彰男: 1. MOS トランジスタのスケーリングに伴う特性ばらつき (小特集, CMOS デバイスの微細化に伴う特性ばらつきの増大とその対策)，電子情報通信学会誌，Vol. 92, No. 6, pp. 416–426（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110007227367/⟩ (2009).

[2] Srivastava, A., Sylvester, D. and Blaauw, D.: *Statistical Analysis and Optimization for VLSI: Timing and Power*, Springer Science & Business Media (2006).

[3] Mukhopadhyay, S., Mahmoodi, H. and Roy, K.: Modeling of failure probability and statistical design of SRAM array for yield enhancement in nanoscaled CMOS, *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, Vol. 24, No. 12, pp. 1859–1880 (online), DOI: 10.1109/TCAD.2005.852295 (2005).

[4] Ernst, D., Kim, N. S., Das, S., Pant, S., Rao, R., Pham, T., Ziesler, C., Blaauw, D., Austin, T., Flautner, K. and Mudge, T.: Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation, *Int'l Symp. on Microarchitecture*, pp. 7–18 (online), DOI: 10.1109/MICRO.2003.1253179 (2003).

[5] Das, S., Tokunaga, C., Pant, S., Ma, W.-H., Kalaiselvan, S., Lai, K., Bull, D. M. and Blaauw, D. T.: RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance, *IEEE J. Solid-State Circuits*, Vol. 44, No. 1, pp. 32–48 (online), DOI: 10.1109/JSSC.2008.2007145 (2009).

[6] Bull, D., Das, S., Shivshankar, K., Dasika, G., Flautner, K. and Blaauw, D.: A power-efficient 32b ARM ISA processor using timing-error detection and correction for transient-error tolerance and adaptation to PVT variation, *Int'l Solid-State Circuits Conf., Digest of Technical Papers*, pp. 284 –285 (online), DOI: 10.1109/ISSCC.2010.5433919 (2010).

[7] Bowman, K. A., Tschanz, J. W., Kim, N. S., Lee, J. C., Wilkerson, C. B., Lu, S. L., Karnik, T. and De, V. K.: Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance, *IEEE J. Solid-State Circuits*, Vol. 44, No. 1, pp. 49–63 (online), DOI: 10.1109/JSSC.2008.2007148 (2009).

[8] Choudhury, M., Chandra, V., Mohanram, K. and Aitken, R.: TIMBER: Time borrowing and error relaying for online timing error resilience, *Design, Automation and Test in Europe*, pp. 1554 –1559 (2010).

[9] Fojtik, M., Fick, D., Kim, Y., Pinckney, N. R., Harris, D. M., Blaauw, D. and Sylvester, D.: Bubble Razor: An architecture-independent approach to timing-error detection and correction, *Int'l Solid-State Circuits Conf.*, pp. 488–490 (online), DOI: 10.1109/ISSCC.2012.6177103 (2012).

[10] Mallik, A., Cosgrove, J., Dick, R. P., Memik, G. and Dinda, P.: PICSEL: Measuring User-Perceived Performance to Control Dynamic Frequency Scaling, *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 70–79 (2008).

[11] 喜多貴信，樽井　翔，塩谷亮太，五島正裕，坂井修一: タイミング制約を緩和するクロッキング方式の予備評価，電子情報通信学会技術研究報告. CPSY, コンピュータシステム， Vol. 109, No. 168, pp. 61–66（オンライン），入手先 ⟨https://ci.nii.ac.jp/naid/110007358826/⟩ (2009).

[12] 喜多貴信，塩谷亮太，五島正裕，坂井修一: タイミング制約を緩和するクロッキング方式の提案，情報処理学会全国大会講演論文集，Vol. 72, pp. 239–240 (2010).

[13] 吉田宗史，広畑壮一郎，倉田成己，塩谷亮太，五島正裕，坂井修一: 動的タイム・ボローイングを可能にするクロッキング方式，情報処理学会論文誌：コンピューティングシステム， Vol. 6, No. 1, pp. 1–16 (2013).

[14] 吉田宗史，広畑壮一郎，倉田成己，五島正裕，坂井修一: 動的タイム・ボローイングを可能にするクロッキング方式の適用手法の評価，情報処理学会研究報告， Vol. 2013-ARC-206, No. 6, pp. 1 – 13（オンライン），入手先 ⟨http://id.nii.ac.jp/1001/00094541/⟩ (2013).

[15] 広畑壮一郎，吉田宗史，倉田成己，五島正裕，坂井修一: 動的タイム・ボローイングを可能にするクロッキング方式の適用手法，情報処理学会研究報告，Vol. 2012-ARC-201, No. 20, pp. 1–8 (2012).

[16] 広畑壮一郎，神原太郎，吉田宗史，倉田成己，五島正裕，坂井修一: 動的タイム・ボローイングを可能にするクロッキング方式の適用，先進的計算基盤システムシンポジウム SACSIS，Vol. 2012, pp. 12–13 (2012). ポスター.

[17] 広畑壮一郎，神原太郎，吉田宗史，倉田成己，五島正裕，坂井修一: 動的タイム・ボローイングを可能にするクロッキング方式の適用手法の実装，情報処理学会研究報告， Vol. 2013, No. 11, pp. 1–9（オンライン），入手先 ⟨https://ci.nii.ac.jp/naid/110009552444/⟩ (2013).

[18] 津坂章仁，谷川祐一，広畑壮一郎，五島正裕，坂井修一: 動的タイム・ボローイングを可能にするクロッキング方式の二相ラッチ生成アルゴリズム，情報処理学会研究報告， Vol. 2014-ARC-211, No. 9, pp. 1–10（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110009808089/⟩ (2014).

[19] 津坂章仁，谷川祐一，広畑壮一郎，五島正裕，入江英嗣，坂井修一: 動的タイム・ボローイングのための二相化アルゴリズムの改良と評価，情報処理学会研究報告，Vol. 2016-ARC-219, No. 23, pp. 1–6（オンライン），入手先 ⟨http://id.nii.ac.jp/1001/00083270/⟩ (2016).

[20] Das, S., Roberts, D., Lee, S., Pant, S., Blaauw, D., Austin, T., Flautner, K. and Mudge, T.: A self-tuning DVS processor using delay-error detection and correction, *IEEE J. Solid-State Circuits*, Vol. 41, No. 4, pp. 792–804 (online), DOI: 10.1109/JSSC.2006.870912 (2006).

[21] Ernst, D. et al.: Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation, *MICRO*, pp. 7–18 (2003).

[22] Bull, D. et al.: A power-efficient 32b ARM ISA processor using timing-error detection and correction for transient-error tolerance and adaptation to PVT variation, *ISSCC*, pp. 284 –285 (2010).

[23] Nakadai, H., Ito, M. and Uetake, T.: Embedded SRAM Technology for High-End Processors, *Fujitsu*, Vol. 61, No. 6, pp. 543–548 (online), available from ⟨http://ci.nii.ac.jp/naid/40017388754/⟩ (2010). Japanese.

[24] Kumar, R. and Hinton, G.: A family of 45nm IA processors, *Int'l Solid-State Circuits Conf., Digest of Technical Papers*, pp. 58–59 (2009). ID: 1.

[25] Warnock, J., Chan, Y. H., Harrer, H., Rude, D., Puri, R., Carey, S., Salem, G., Mayer, G., Chan, Y. H., Mayo, M., Jatkowski, A., Strevig, G., Sigal, L., Datta, A., Gattiker, A., Bansal, A., Malone, D., Strach, T., Wen, H., Mak, P. K., Shum, C. L., Plass, D. and Webb, C.: 5.5GHz system Z microprocessor and multi-chip module, *Int'l Solid-State Circuits Conf., Digest of Technical Papers*, pp. 46–47 (online), DOI: 10.1109/ISSCC.2013.6487630 (2013).

[26] Karl, E., Sylvester, D. and Blaauw, D.: Timing Error Correction Techniques for Voltage-Scalable On-Chip Memories, *IEEE Int'l Symp. on Circuits and Systems (ISCAS)*, pp. 3563–3566 (online), DOI: 10.1109/ISCAS.2005.1465399 (2005).

[27] 入江英嗣，五島正裕，坂井修一: メモリ装置およびメモリ読み出しエラー検出方法 (2008).

[28] 五島正裕，倉田成己，塩谷亮太，坂井修一: タイミング・フォールト耐性を持つ Out-of-Order プロセッサ，情報処理学会論文誌：コンピュー ティングシス テム， Vol. 6, No. 1, pp. 17–30（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110009527308/⟩ (2013).

[29] 吉田宗史，倉田成己，塩谷亮太，五島正裕，坂井修一: タイミング・フォールト耐性を持つ Out-of-Order プロセッサの検出/回復方式，先進的計算基盤シス テムシン ポジ ウム SACSIS，pp. 10–19（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/170000076897/⟩ (2013).

[30] University, N. C. S.: NCSU EDA Wiki, North Carolina State University (online), available from ⟨https://www.eda.ncsu.edu/wiki/NCSU_EDA_Wiki⟩ (accessed 2019/01/09).

[31] Bowman, K. A., Tschanz, J. W., Lu, S. L. L., Aseron, P. A., Khellah, M. M., Raychowdhury, A., Geuskens, B. M., Tokunaga, C., Wilkerson, C. B., Karnik, T. and De, V. K.: A 45nm Resilient Microprocessor Core for Dynamic Variation Tolerance, *IEEE J. Solid-State Circuits*, Vol. 46, No. 1, pp. 194–208 (online), DOI: 10.1109/JSSC.2010.2089657 (2011).

[32] Harris, D.: *Skew-Tolerant Circuit Design*, Morgan Kaufmann Publishers (2001).

[33] Jimbo, U., Yamada, J., Shioya, R. and Goshima, M.: Applying Razor Flip-Flops to SRAM Read Circuits, *IEICE Trans. Electron.*, Vol. E100-C, No. 3, pp. 245–258 (online), DOI: 10.1587/transele.E100.C.245 (2017).

[34] 神保　潮，山田淳二，五島正裕: 動的タイム・ボローイングを可能にするクロッキング方式の適用 (2017). *cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming* (*xSIG 2017*) に採択.

[35] 神保　潮，山田淳二，五島正裕: 動的タイム・ボローイングを可能にするクロッキング方式の適用，情報処理学会論文誌：コンピューティングシステム， Vol. 10, No. 2, pp. 1 – 12（オンライン），入手先 ⟨http://id.nii.ac.jp/1001/00183237/⟩ (2017).

[36] Ford, L. R. and Fulkerson, D. R.: Maximal flow through a network, *Canadian Journal of Mathematics*, Vol. 8, pp. 399 – 404 (online), DOI: 10.4153/CJM-1956-045-5 (1956).

[37] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C.: *Introduction to Algorithms*, The MIT Press (2009).

[38] Nagamochi, H. and Ibaraki, T.: Computing edge-connectivity in multigraphs and capcacitated graphs, *SIAM Journal on Discrete Mathematics*, Vol. 5, pp. 54 – 66 (1992).

[39] 京都大学永持研究室: 研究成果 > アルゴリズムのデモ > 最小カット問題，京都大学（オンライン），入手先 ⟨http://www-or.amp.i.kyoto-u.ac.jp/demo/MINCUT.html⟩（参照 2018/11/05）.

[40] Goldberg, A. V. and Tarjan, R. E.: A new approach to the maximum-flow problem, *Journal of the ACM*, Vol. 35, No. 4, pp. 921 – 940 (online), DOI: 10.1145/48014.61051 (1988).

[41] Edmonds, J. and Karp, R. M.: Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of the ACM*, Vol. 19, No. 2, pp. 248 – 264 (1972).

[42] 神保　潮，五島正裕: 逆方向カット・エッジのない最小カットを求めるアルゴリズム，情報処理学会論文誌：コンピューティングシステム， Vol. 11, No. 1, pp. 1–11 (2018).

[43] Sharaiha, E.: Edmonds Karp in C#, GitHub (online), available from ⟨http://gist.github.com/Eyas/7520781⟩ (accessed 2018/11/05).

[44] de Halleux, J.: QuickGraph, Graph Data Structures and Algorithms for .NET, CodePlex Archive (online), available from ⟨http://quickgraph.codeplex.com/⟩ (accessed 2018/11/05).

[45] RISC-V Foundation: RISC-V Foundation | Instruction Set Architecture (ISA), RISC-V Foundation (online), available from ⟨http://riscv.org/⟩ (accessed 2018/11/05).

[46] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H. and Waterman, A.: The Rocket Chip Generator, Technical Report UCB/EECS-2016-17, EECS Dept., UCB (2016).

[47] RISC-V Foundation: *The RISC-V Instruction Set Manual*.

[48] Bradbury, A. et al.: Tagged memory and minion cores in the lowRISC SoC, lowRISC (online), available from ⟨https://www.lowrisc.org/⟩ (accessed 2018/11/05).

[49] 藤田晃史，倉田成己，塩谷亮太，五島正裕，坂井修一: 面積効率を指向するプロセッサ「雷上動」の設計と実装，電子情報通信学会技術研究報告，Vol. 113, No. 498, pp. 229–234（オンライン），入手先 ⟨https://ci.nii.ac.jp/naid/110009861628/⟩ (2014).

[50] Shioya, R., Horio, K., Goshima, M. and Sakai, S.: Register Cache System not for Latency Reduction Purpose, *Int'l Symp. on Microarchitecture*, pp. 301–312 (online), DOI: 10.1109/MICRO.2010.43 (2010).

[51] 有馬　慧，倉田成己，塩谷亮太，五島正裕，坂井修一: タイミング・フォールト耐性を持つ Out-of-Order プロセッサ，先進的計算基盤シンポジウム SACSIS，pp. 270–279 (2012).

著者発表論文

著者が主著のもの

雑誌論文

[1] Ushio Jimbo, Junji Yamada, Ryota Shioya, and Masahiro Goshima: Applying Razor Flip-Flops to SRAM Read Circuits, IEICE Trans. Electron., Vol. E100-C, No. 3, pp. 245–258, Mar. 2017. （研究論文）.

[2] 神保 潮, 山田 淳二, 五島 正裕: 動的タイム・ボローイングを可能にするクロッキング方式の適用, 情報処理学会論文誌 : コンピューティングシステム, Vol. 10, No. 2, pp. 1–12, Sep. 2017. （研究論文）.

[3] 神保 潮, 五島 正裕: 逆方向カット・エッジのない最小カットを求めるアルゴリズム, 情報処理学会論文誌 : コンピューティングシステム, Vol. 11, No. 1, pp. 1–11, Mar. 2018. （研究論文）.

国際会議発表

[4] Ushio Jimbo, Ryota Shioya, and Masahiro Goshima: Clocking Scheme That Realizes Ballistic Signal Flow, *The ACM Student Research Competition* (*SRC*), 2 pages, Oct. 2018. （ポスター）.

[5] Ushio Jimbo, Ryota Shioya, and Masahiro Goshima: Application of Timing Fault Detection to Rocket Core on FPGA, *Int'l Workshop on Computing Systems and Architectures* (*CSA*), 4 pages, Nov. 2018.（ショートペーパー投稿･査読 (6 pages)/ポスター採択）. (to appear).

査読付き国内会議

[6] 神保 潮, 山田 淳二, 五島 正裕: 動的タイム・ボローイングを可能にするクロッキング方式の適用. *cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming* (*xSIG*), May 2017. （同内容のポスター発表が Poster Award を受賞）.

口頭発表

[7] 神保 潮, 山田淳二, 五島正裕, 坂井修一: ダイナミック・ロジックへのタイミング・フォールト検出手法の適用, 情報処理学会研究報告, Vol. 2014-ARC-210, No. 18, pp. 1 – 8, May 2014.

[8] 神保 潮, 五島 正裕: 動的タイム・ボローイングを可能にするクロッキング方式のスカラ・プロセッサへの適用, 情報処理学会研究報告, Vol. 2017-ARC-226, No. 18, pp. 1 – 8, May 2017.

[9] 神保 潮, 五島 正裕: 逆方向カット・エッジのない最小カットを求めるアルゴリズムの改良, 情報処理学会研究報告, Vol. 2018-ARC-230, No. 35, pp. 1 – 6, Mar. 2018. （情報処理学会 システム・アーキテクチャ研究会, 若手奨励賞を受賞）

[10] 神保 潮, 塩谷 亮太, 五島 正裕: 動的タイム・ボローイングを可能にするクロッキング方式のプロセッサへの適用, 情報処理学会研究報告, Vol. 2018-ARC-232, No. 23, pp. 1 – 8, Jul. 2018.

受賞

[11] 神保 潮: 動的タイム・ボローイングを可能にするクロッキング方式の適用, *The 1st. cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming* (*xSIG*), Poster Award, May 2017.

[12] 神保 潮: 逆方向カット・エッジのない最小カットを求めるアルゴリズムの改良, 第 222 回 情報処理学会 システム・アーキテクチャ研究会, 若手奨励賞, Jul. 2018.

著者が主著でないもの

雑誌論文

[13] Junji Yamada, Ushio Jimbo, Ryota Shioya, Masahiro Goshima, and Shuichi Sakai: Skewed Multistaged Multibanked Register File for Area and Energy Efficiency, IEICE Trans. Inf. & Syst., Vol. E100-D, No. 4, pp. 822–837, Apr. 2017.

[14] Junji Yamada, Ushio Jimbo, Ryota Shioya, Masahiro Goshima, and Shuichi Sakai: Bank-Aware Instruction Scheduler for Multibanked Register File, Journal of Information Processing, Vol. 26, pp. 696–705, Sep. 2018.

査読付き国内会議

[15] Junji Yamada, Ushio Jimbo, Ryota Shioya, Masahiro Goshima, Shuichi Sakai: Bank-Aware Instruction Scheduler for Multibanked Register File, *cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming* (*xSIG*), May 2017.

[16] 崔 ミン誠, 福田 隆, 神保 潮, 五島 正裕, 坂井 修一: 帰納的なシミュレーション・ポイント選出手法の改良, *cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming* (*xSIG*), May 2017.

口頭発表

[17] 崔 ミン誠, 福田 隆, 神保 潮, 五島 正裕, 坂井 修一: 集合的な帰納的シミュレーション・ポイント選出手法の提案, 情報処理学会研究報告, Vol. 2016-ARC-220, No. 13, pp. 1–6, May 2016.

[18] 酒井 一憲, 津坂 章仁, 神保 潮, 五島 正裕, 坂井 修一: 回路素子の静的解析を用いた二相化アルゴリズムの改良第 77 回 情報処理学会 全国大会 講演論文集, Vol. 2015, No. 1, pp. 77–78, Mar. 2015.

謝辞

本研究を進めるにあたり，指導教員である五島正裕教授には，卒業論文，修士課程から，博士課程に至るまで長きにわたり，御指導，御鞭撻を頂きました．ここに深く感謝の意を表します．

坂井修一教授には，卒業論文，修士課程において指導していただき，その後も気にかけて下さりました．

米田友洋教授，合田憲人教授，入江英嗣准教授，鯉渕道紘准教授には，審査において大変有益なご助言を頂きました．

塩谷亮太准教授には，研究に関する数多くのご協力，ご助言をいただきました．

当時坂井研究室の同僚であった山田淳二氏，倉田成己氏，吉田宗史氏，広畑壮一郎氏，津坂章仁氏，酒井一憲氏には，本研究に関する議論等を通じて，多くのご協力をいただきました．

秘書の八木原晴水氏，長谷部環氏，勝紀子氏，樫村純子氏には，研究室での生活や事務手続きに関する数多くのサポートをしていただきました．

その他，研究室に在籍中多くの皆様に，研究生活を通じて様々なご協力，ご支援を頂きました．

ここに深甚なる謝意を表します．

本論文の研究の一部は，文部科学省科学研究費補助金 No. 2380012，および，16H02797，JST CREST「ディペンダブル VLSI システムの基盤技術」の支援により行われたものです．

また，東京大学大規模集積システム設計教育研究センターを通じ，シノプシス株式会社，日本ケイデンス株式会社，メンター株式会社の協力で行われたものです．