修士論文

タイミング・フォールト耐性を持つ回路/アーキテクチャ技術

Timing-Fault-Tolerant Circuit and Architecture Techniques

平成25年02月06日提出

指導教員
坂井 修一 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-116446 吉田 宗史

概要

半導体プロセスの微細化に伴って増大するランダムばらつきにより，従来のワースト・ケースに基づいた設計は悲観的になりすぎている．この問題の対策の1つに，タイミング・フォールトを動的に検出/回復する技術がある．このような技術により，実効的な遅延に基づいた動作を実現でき，プロセッサのサイクル・タイムを短縮できる．プロセッサを対象とするものは，一般に，タイミング・フォールトを検出する回路レベルの技術と，検出後に回復を行うアーキテクチャ・レベルの技術の，2つからなる．

しかし，既存の検出/回復技術では，従来用いられるクロッキング方式の遅延制約により実際にはサイクル・タイムを短縮できず，かつ，フォールトの発生箇所の考慮が不十分であることから，単純なスカラ・プロセッサにしか適用できない．そこで本稿では，二相ラッチ方式と呼ばれるクロッキング方式とこの技術を組み合わせることで，実効的な遅延の平均に基づく動作を可能とし，サイクル・タイムを半減出来る検出技術を提案する．さらに，制御系モジュールの内部で発生するフォールトへの対策を行うことで，複雑な out-of-order プロセッサにも適用できる回復技術を提案する．

目 次

第 1 章　はじめに　4

第 2 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　8

2.1 入力ばらつきと既存のクロッキング方式 8
2.1.1 タイミング・ダイアグラムと入力ばらつき 9
2.1.2 単相 FF 方式 11
2.1.3 二相ラッチ方式 12
2.1.4 静的タイム・ボローイング 14
2.1.5 Razor 16
2.1.6 Razor のショート・パス問題 18
2.2 提案手法 19
2.2.1 回路構成 19
2.2.2 動的タイム・ボローイング 20
2.2.3 Razor と提案手法の比較 23
2.2.4 既存のクロッキング方式との比較 25
2.3 評価 26
2.3.1 動作周波数 29
2.3.2 回路面積のオーバーヘッド 29
2.4 関連研究 30
2.4.1 ReCycle 30
2.4.2 TIMBER 31
2.4.3 Bubble Razor 33
2.5 本章のまとめ 35

第 3 章　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　37

3.1 タイミング・フォールト検出/ 回復技術 37
3.1.1 アーキテクチャ・レベルの回復技術 37
3.1.2 Razor Ⅱ の回復技術 39
3.2 Out-of-Order プロセッサと Razor Ⅱ の限界 40
3.2.1 Out-of-Order プロセッサのコミット 40
3.2.2 Razor Ⅱ の回復技術の問題点 43

3.3 Timing-Fault-Tolerant OoO Processor 44
3.3.1 In-Order Passing through PNR 44
3.3.2 Imprecise Cancellation 44
3.3.3 Initialization of Pipeline above PNR 45
3.4 提案手法の検出/回復方式 45
3.4.1 初期化方式の構成 45
3.4.2 バック・エッジへの対策 46
3.4.3 回復技術 47
3.4.4 回復のペナルティ 49
3.5 評価 50
3.5.1 評価モデルと評価環境 50
3.5.2 評価結果 51
3.6 本章のまとめ 52

第 4 章 おわりに **54**

参考文献 **55**

第1章　はじめに

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，チップ間に跨るシステマティックなばらつきではなく，チップ内のランダムなばらつきである．これは，トランジスタや配線のサイズが原子のサイズに近づくために生ずる本質的な問題であり，原理的に避けえない．

ばらつきが増大していくと，従来の**ワースト値**に基づいた設計手法は悲観的になりすぎる．微細化が進むにつれて，ばらつきの増大により，平均値とワースト値の差は広がっていく．その結果，LSI の設計上の動作速度が向上しなくなる恐れもある．

そのため，ワースト・ケースより実際に近い遅延に基づいた動作を実現する手法が提案されている．設計段階において遅延のばらつきを統計的に扱う **SSTA**（Statistic Static Timing Analysis : 統計的静的タイミング解析）[2] もその一例である．SSTA によれば，ワースト・ケースほど悲観的ではない遅延見積もりを行うことができる．

動的タイミング・フォールト検出・回復　SSTA のように，設計時に用いられる静的な手法に対し，動作時にタイミング・フォールトを検出し回復する動的な手法がある．

タイミング・フォールト（Timing Fault : **フォールト**）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り，その場合でもフォールトが発生しないように設計する．したがって，そのように設計・製造された LSI では，原則フォールトは発生しない．実際にフォールトが起こるのは，想定した動作条件を外れた場合，例えば，温度センサの故障による熱暴走を起こした場合などに限られる．

一方で，フォールトの発生自体は許容し，フォールトの検出・回復を行う手法が考えられる．2.1.5 節で詳しく述べる **Razor** [3, 4, 5] は，その代表例である．このような手法と **DVFS** (Dynamic Voltage and Frequency Scaling) [6] を組み合わせると，以下のように，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．

図 1.1 にその様子を示す．図 1.1 中×印はワースト・ケースで定められた DVFS の **V**（Voltage：電源電圧）と **F**（Frequency：動作周波数）の組を表している．

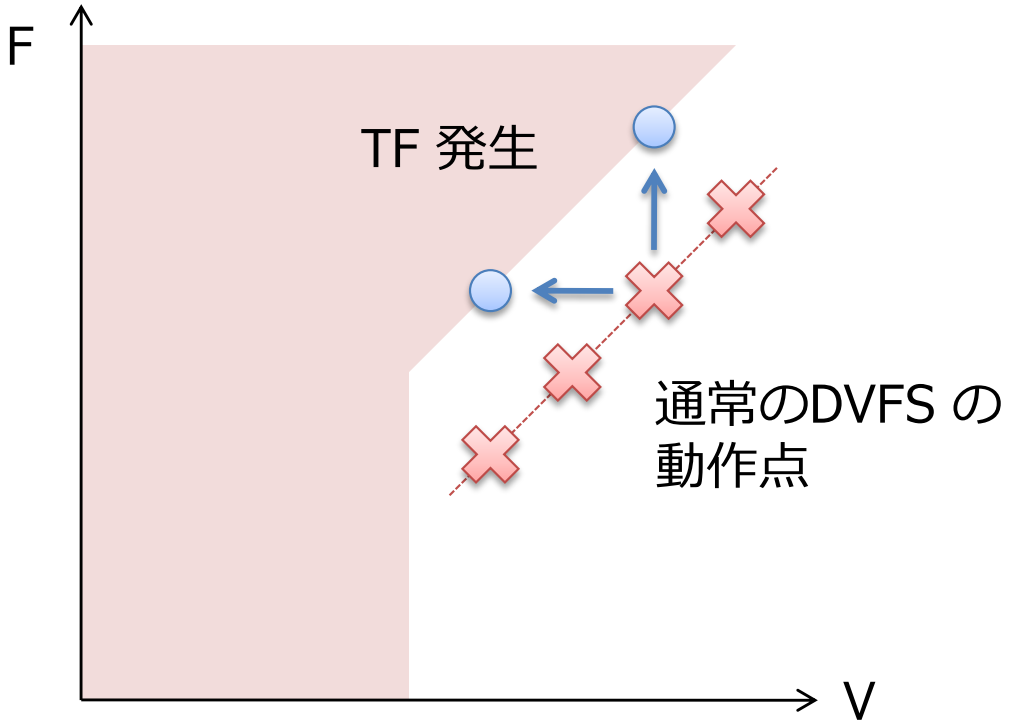

図 1.1: タイミング・フォールト検出・回復と DVFS の組み合わせ

ワースト・ケース設計では，このように，フォールトが発生しないよう十分なマージンを取って V–F が設定される．フォールト検出・回復を行う手法では，ここより 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，いずれフォールトが発生し検出される．図 1.1 中〇印で表される検出直前の 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，フォールト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，ワースト・ケース設計で必要であったマージンを劇的に削減することができる．

実効遅延の平均 (typical) に基づく動作　動的にフォールトを検出・回復する手法は，ロジックの実効的な遅延がサイクル・タイムより長いとフォールトとして検出する．そこでこれらの手法では，更に，クリティカル・パスの遅延ではなく，それよりはるかに短い**実効遅延**に基づく動作が可能になるのではないかと期待される．実効遅延の正確な定義は，2.1.1 節で述べる．

実際にロジックの実効遅延の分布を調べると，その平均はクリティカル・パスの遅延の半分程度以下であり，分布は遅延小に大きく偏っている [7]．特に，クリティカル・パスが活性化する確率は 1/100 程度である一方，実効遅延が 0 となる確率は 1/2 程度にもなる．なぜなら，ロジックの出力が変化する確率は 1/2 程度であり，出力が変化しなかった場合には実効遅延は実質 0 となるから

である．

したがって，クリティカル・パスのワーストではなく，実効遅延の平均(typical)に基づく動作が可能であるなら，動作周波数2倍も夢ではない．

既存手法の限界　しかし実際には，Razorなどの既存の手法では，実効遅延に基づいた動作を実現すること，つまり，クリティカル・パスで規定されるサイクル・タイムより短縮することは実際上できない．2.2.3節において簡単な実験を元に詳しく述べるが，チップ内にフォールトを起こしうるパスは無数に存在するため，サイクル・タイムを短縮すると，パスのうちどれか一つで必ずフォールトが発生しうる．フォールトが発生するたび，検出し，回復処理を行わなければならず，チップとしてそもそも機能しないのである．

すなわち，既存のフォールト検出・回復手法でできることは，実質，DVFSのマージンを削減することであると言える．

動的タイム・ボローイング　このことからまた，以下のことが示唆される：フォールト検出・回復によってサイクル・タイムをクリティカル・パスの遅延より短縮するためには，実効遅延の平均(typical)が小さいだけでは不十分であり，実効遅延の分散が十分に小さい必要がある．

しかし現実には，実効遅延の分散を小さくする手法など恐らくない．そこで本稿で提案するのは，実効遅延の分散を緩和することによって，実効遅延の平均に近いサイクル・タイムでの動作を可能とするクロッキング方式である．これは，端的に言えば，フォールト検出と二相ラッチを組み合わせたもので，このことにより**動的タイム・ボローイング**が可能になる．2.2.2節で詳しく述べるが，従来からある二相ラッチ方式で可能になるタイム・ボローイングは，言わば**静的タイム・ボローイング**と呼べるもので，設計時にステージ間で遅延を融通するものである．本提案で可能となる動的タイム・ボローイングとは，実行時に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というものである．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．このことにより実効遅延の分散を吸収し，実効遅延の平均に近いサイクル・タイムでの動作が可能となるのである．

フォールト検出限界の改善　動的タイム・ボローイングを行っても，サイクル・タイムは実効遅延の分布に従って無制限に短縮できるわけではない．フォールトを検出する手法には，これ以上削減するとフォールトが正しく検出できなくなる**検出限界**が存在する．Razorの検出限界は，クリティカル・パス遅延の2/3倍程度である．提案手法では更に，フォールトの検出方法を工夫することにより，この検出限界をクリティカル・パス遅延の1/2倍へと削減することができる．

提案手法は，主にこれら 2 点により，サイクル・タイム 1/2，すなわち，動作周波数 2 倍を達成することができる．

回復技術の提案　また，我々はこれまでに，out-of-order スーパスカラ・プロセッサを対象としたアーキテクチャ・レベルの回復手法について提案してきた [8, 9]．我々の手法ではリオーダ・バッファ (ROB) やロード / ストア・キュー (LSQ) の内部で発生するフォールトへの対処を行い，Razor Ⅱ では対応できない制御系のフォールトにも対処することができる．

本稿では，この検出/回復方式について，物理的な構成や具体的な実装方法，回復のペナルティなどの詳細について検討し評価を行う．また，提案の回復方式は，構成の仕方によって回復のペナルティが大きく変化する．シミュレーションによって，この回復のペナルティが IPC に与える影響を評価する．

以下，2 章では，回路レベルの検出技術である，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について説明する．3 章ではアーキテクチャ・レベルの回復技術である以前の提案を簡単に説明し，この検出/回復方式について詳細に述べる．

第2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

以下，2.1 節ではまず，我々が**タイミング・ダイアグラム**と呼ぶ図を提案する．その上で，実効遅延の正確な定義を与え，さらに様々な既存のクロッキング方式のタイミング制約について述べる．続く 2.2 節で，提案手法の構成・動作を示す．提案手法は，タイミング・ダイアグラム上で理論的に導き出されたものである．そこで 2.3 節では，比較的簡単なものではあるが，現実の回路に提案手法を適用した結果，実際に 2 倍の動作周波数を実現できることを示す．2.4 節では，提案手法の関連研究について述べる．

2.1　入力ばらつきと既存のクロッキング方式

どのようなクロックを分配するか，フリップ・フロップ (FF) とラッチのどちらを用いるかといった，同期式順序回路の同期動作を規定する方式を**クロッキング方式**という． 本章では，主に既存のクロッキング方式について述べる．

クロッキング方式が正しく動作するための**タイミング制約**は，クロッキング方式によって大きく異なる．本稿では特に，ロジックの遅延の最大値を与える**最大遅延制約**が，クロッキング方式によってどのように変わるかに興味がある．最大遅延制約は，一次近似的にはクロックのサイクル・タイムによって与えられるが，FF/ラッチのセットアップ/ホールド・タイム，クロック–データ遅延，および，クロック・スキューなどを考慮する必要がある [10]．しかし，説明が煩雑になり過ぎるため，本稿では，サイクル・タイムのみに着目し，その他の要因については省略することにする．必要であれば，これらを議論に追加することは容易である．

クロッキング方式を理解する上では，我々が**タイミング・ダイアグラム (t-diagram)** と呼ぶ図を用いると都合がよい．また，特にフォールト検出を行うクロッキング方式では，ロジックの**実効遅延**と呼ぶ概念が重要になる．以下，2.1.1 節で t-diagram と実効遅延について紹介した後，2.1.2 節以降で，既存のクロッキング方式とその最大遅延制約について述べる．

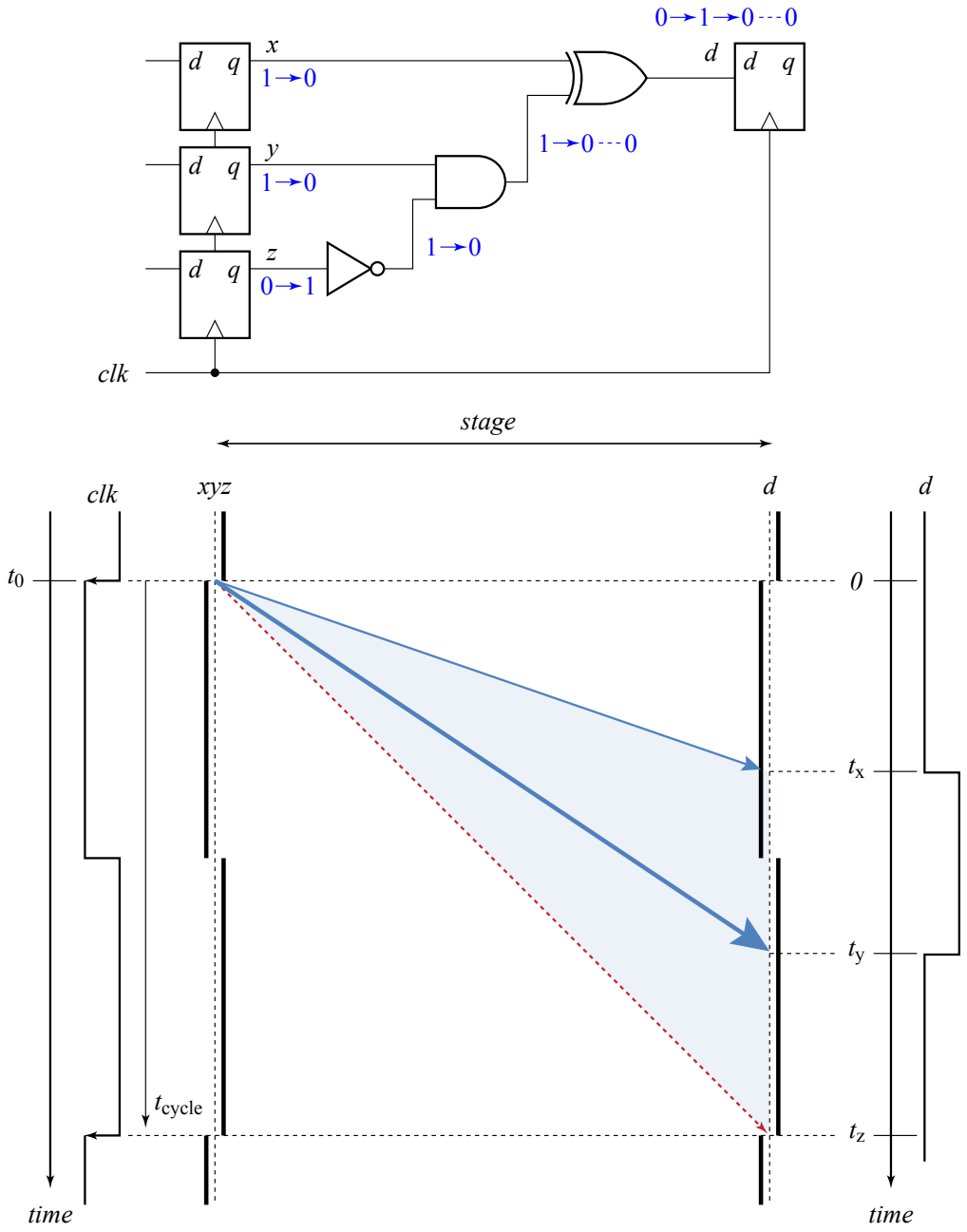

図 2.1: タイミング・ダイアグラム (t-diagram) と実効遅延

2.1.1 タイミング・ダイアグラムと入力ばらつき

図 2.1（上）の回路において，信号が伝わる様子を同図（下）に示す．同図（下）の図を，我々は，**タイミング・ダイアグラム (t-diagram)** と呼んでいる．通常のタイム・チャートが論理値-時間の 2 次元を持つに対して，t-diagram は時間-空間の 2 次元を持つ．通常のタイム・チャートでは，右方向が時間を，上

下方向が論理値を表す．タイム・チャートは，論理値の時間的変化を表現するが，1 本の波形で表すことができるのは回路の特定の 1 点の振る舞いに限られる．複数の点にまたがる動きを把握するためには，複数の波形を並べなければならない．それに対して t-diagram は，下方向が時間を，右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．

図 2.1（上）に示す回路で，時刻 $t = 0$ に 3 つの FF の出力 (x, y, z) が $(1, 1, 0)$ から $(0, 0, 1)$ に遷移したとする．x，y，z から d に至るパスの遅延をそれぞれ t_x，t_y，t_z とすると，ロジックの出力 d は，時刻 $t = t_x$，t_y において $0 \to 1 \to 0$ と遷移する．z から d に至るパス上を伝わる信号は，y から d に至るパスの信号によってマスクされるため，時刻 $t = t_z$ には出力は変化しないことに注意されたい．同図の右端にある波形が，d における通常のタイム・チャート（を右に 90° 回転したもの）である．

パスの活性化とタイミング・ダイアグラム　同図のように t-diagram では，ロジックの入力において入力が変化した時刻から，同ロジックの出力において出力が変化した時刻までを直線矢印で結ぶことによって，信号の伝わる様子を表す．

図 2.1 に示した例では，前述したように，z から d に至るパスを通る信号は途中でマスクされるため，時刻 $t = t_z$ においては出力 d は変化しない．パスを通った信号によって実際にロジックの出力が変化したとき，その信号によってそのパスが**活性化**したと言う．

t-diagram では，パスを活性化した信号の伝達を実線矢印で表す．活性化しなかった場合には，途中でマスクされた段階で信号は物理的には消失しているが，仮想的に点線矢印で表すことにする．

入力ばらつきとタイミング・ダイアグラム　t-diagram では，ロジックのクリティカル・パスの遅延に対応する矢印を赤で描くこととし，その角度を 45° と決める．この約束により，あるロジックのクリティカル・パスの遅延は，t-diagram 上のロジックに対応する領域の横幅によって表現することができる．このことは特に，2.1.4 節でタイム・ボローイングの説明をする上で重要となる．

実際のロジックでは，ばらつきのため，遅延は連続的に変化する．そのため，矢印の存在範囲は，ロジックの最小遅延の矢印とクリティカル・パスの遅延の赤矢印に上下を挟まれた領域となる．

t-diagram では，網掛けを施してこの領域を示す．

実効遅延　あるロジックにおいて最後に活性化されたパスの遅延を，このロジックの**実効遅延**と呼ぶことにする．図 2.1 の場合，時刻 $t = t_z$ においてクリ

ティカル・パスを通った信号が到着するはずだが，マスクされたため，ロジックの出力 d は変化しない．この場合，実効遅延は t_y となる．

時刻 $t = t_y$ において出力 d が変化した時には実効遅延が t_y であることは分からない．時刻 $t = t_z$ において d が変化しなかったことを見て初めて t_y であったことが分かる．このように，実効遅延は事後的に分かることに注意されたい．

t-diagram では実効遅延に対応する矢印を太実線で表す．クリティカル・パスが活性化した場合には，45° の赤矢印を更に太くして表す．

入力ばらつきと実効遅延

ロジックへの入力の変化の仕方によって出力の変化の仕方も様々であり，どのパスが最後に活性化されるかは毎サイクル異なる．つまり実効遅延は，入力の変化の仕方によって大きくばらつく．このことを**入力ばらつき**と呼ぶ．

入力ばらつきは，他のばらつきに比べて非常に大きい [7]．出力が直前のサイクルから変化しなかった場合には，実効遅延は実質 0 となる．すなわち，入力ばらつきは，0 からクリティカル・パス遅延まで変化するのである．他のばらつきによる遅延の変化が高々数十 % 程度であることを考えると，この変化の度合いは非常に大きいと言える．また，ロジックの出力の変化率は 1/2 程度であることが知られている．すなわち，1/2 程度の高い確率で実効遅延は 0 となることにも注意する必要がある．

以下では，既存のクロッキング方式として，単相 FF，二相ラッチ，および，Razor の 3 方式について紹介し，主にその最大遅延制約を示す．

2.1.2 単相 FF 方式

図 2.2 左に，単相クロックとエッジ・トリガ型 FF を組み合わせた単相 FF 方式の t-diagram を表す．

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 の下にある実線はラッチが閉じている状態を，実線と実線の間の空白は，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．

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．

一連の処理（例えば，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．t-diagram では，あるフェーズの処理と次のフェーズの処理を，矢印が存在し得る三角形の領域の網掛けの色を分けることで区別している．

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．t-diagram 上では，以下の 2 つの条件が満たされていればよい：

1. クリティカル・パスの遅延を表す 45° の赤線をたどって，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，前後のフェーズのそれと重ならない．

クロッキング方式のタイミング制約は，この条件から導かれる．

最大遅延制約　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・エッジで上流側の FF の出力が変化してから，次のクロック・エッジまでに下流側の FF の入力に信号が必ず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，ロジックのクリティカル・パスの遅延が各ステージあたり τ 未満であればよい．このことを，最大遅延制約は $1\tau/1$ ステージ と表現することにする．図 2.2 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，実際に最大遅延制約の限界を達成した場合を表している．

なお実際には，FF のクロック–データ遅延やセットアップ・タイム，クロック・スキューのため，ちょうど次のクロック・エッジに到着するようにはできない．また，クロック・スキューや FF のホールド・タイムのため，遅延の最**小**値に関する制約である**最小遅延制約**も生じる．ただし前述したように，本稿ではこれらについては考慮しない．

2.1.3　二相ラッチ方式

図 2.2 右が，二相のクロックとラッチを組み合わせた二相ラッチ方式の t-diagram である．

二相ラッチは，マスター–スレーブ構造を持つ FF を構成する 2 つのラッチのうちの 1 つを，ロジックの中ほどに移動したものと理解することができる．単相 FF 方式の 1 ステージに相当するロジックを，このラッチが二分する形になる．

二相ラッチと言っても，実際には，二相のクロックを用いる必要はない．正相と逆相の 2 種類ラッチを用いることで，クロックは（デューティ比 50% の）単相とすることができる．

図 2.2 右に示すように，t-diagram では，ラッチが開いた瞬間同士をクリティカル・パスの遅延を表す 45° の赤線で結んでおり，赤線の上の三角形の網掛けの中を信号は通過する．信号は，ラッチが開いた瞬間から出発し，次のラッチが閉じている期間に到着し，このラッチが開くまで待つことになる．

このように，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，ラッチが開いている期間は原則使うことができない．この理由については，後で詳しく述べる．

ダブル・パイプライニングとの比較　ここで注意すべきことは，単相 FF で組まれた回路をベースに二相ラッチ化したとしても，回路の処理能力，スループットは変わらないということである．

このことは，同じ単相 FF の回路をベースに，ラッチを移すのではなく，FF を挿入した場合を考えると分かりやすいであろう．各ロジックの中ほどに FF を挿入することは，パイプライン・ピッチを倍にすることで，ダブル・パイプライニングとも呼ばれることもある．この場合，ステージ数が 2 倍となって各ロジックの遅延が半分となった分，クロック周期を 2 倍にして，スループットを 2 倍にすることができる[1]．

一方，同じ単相 FF の回路を二相ラッチ化した場合には，クロック周期は原則変わらず，スループットも変わらない．これは，図 2.2 の左右で，どちらも 1 サイクルあたり 1 フェーズしか処理されていないことによって確認できる．

同様に，パイプライン動作における並列実行の単位であるステージは，ベースとなった単相 FF のステージと変わらない[2]．すなわち，二相ラッチ方式においては，正相のラッチから次の正相のラッチまでが 1 ステージ であり，正相のラッチから次の逆相のラッチまでは 0.5 ステージ と表現することができる．

最大遅延制約　二相ラッチ方式の最大遅延制約は $0.5\tau/0.5$ ステージ であり，単相 FF 方式の $1\tau/1$ ステージ と，率としては変わらない．前述したように単相 FF を二相ラッチ化しても回路のスループットは変わらないのだから，結論としては自明であろう．以下では，この制約を守らないとどのような不具合が起こるかを具体的に説明しよう．

これ以上にサイクル・タイムを短縮すると，クリティカル・パスが連続して活性化した場合に不具合が発生する．**図 2.3**（右）は，$(0.5\tau \times 1.33)/0.5$ ステージ となるまでサイクル・タイムを短縮した場合を表している．同図中に爆発のマークで示した点において，以下のように，前述した条件を満たしていない：

[1] もちろん，ハザードの影響が増大するため，回路の実効的な処理能力が必ず 2 倍になるという訳ではない．ステージ数は，アーキテクチャなど上位設計との協調によって決められるべきことで，回路レベルで自由に増減できるものではない．

[2] ダブル・パイプライニングの場合とは対照的に，単相 FF か二相ラッチかは上位設計に影響を及ぼさないので，上位設計とは独立に回路レベルで自由に選択できる．

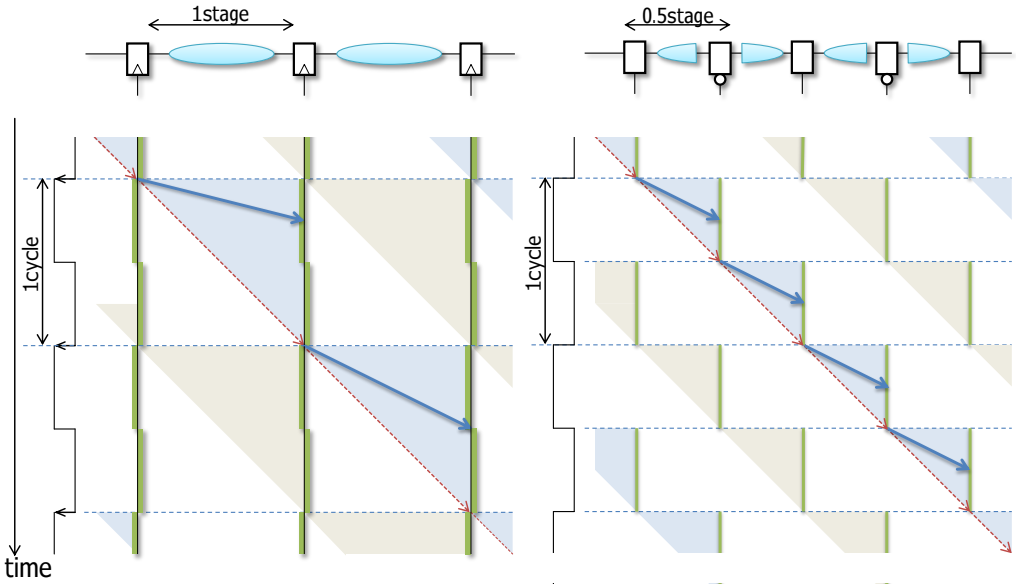

図 2.2: 単相 FF（左）と二相ラッチ（右）の t-diagram

1. 45° の赤い直線が次のフェーズのラッチの閉じた期間にかかっており，次のサイクルに次のステージに処理が引き継がれない．

2. 次のフェーズの三角形の領域との重なりが生じている．

実際に連続してクリティカル・パスが活性化すると，信号が間に合わない上，次のフェーズの信号と「混じっ」てしまう．

このようなことを防ぐためには，最大遅延制約は $0.5\tau/0.5$ ステージ でなければならない．その結果，前述したように，信号は必ず次のラッチが閉じている期間に到着することになり，矢印の存在領域は図 2.3（左）の網掛けした三角形の内部に限られる．各ステージでクリティカル・パスが活性化しなかったとしても，ラッチが開くまで信号の伝播は待たなければならない．

2.1.4 静的タイム・ボローイング

二相ラッチ方式は，単相 FF 方式に比べ設計がやや煩雑になるが，サイクル・タイムをステージ間で融通することにより各ステージごとの遅延制約を緩和することができる．この効果はタイム・ボローイングとして知られている [10]. 本稿では，提案手法における動的タイム・ボローイングと区別して，この効果を**静的タイム・ボローイング**と呼ぶことにする．

図 2.4 に，ステージ間の遅延がバランスしていない場合の単相 FF 方式（左）と二相ラッチ方式（右）の t-diagram を示す．前述したように，各ステージのクリティカル・パスの遅延を表す赤線が 45° であることに注意されたい．同図

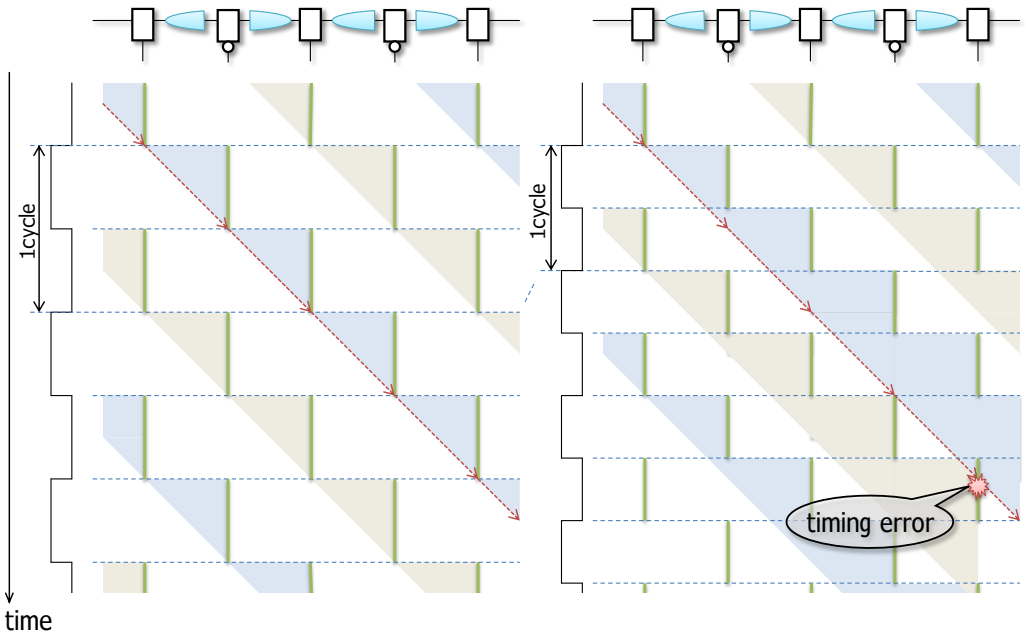

図 2.3: 二相ラッチ方式のサイクル・タイムを短縮した場合の t-diagram（右）

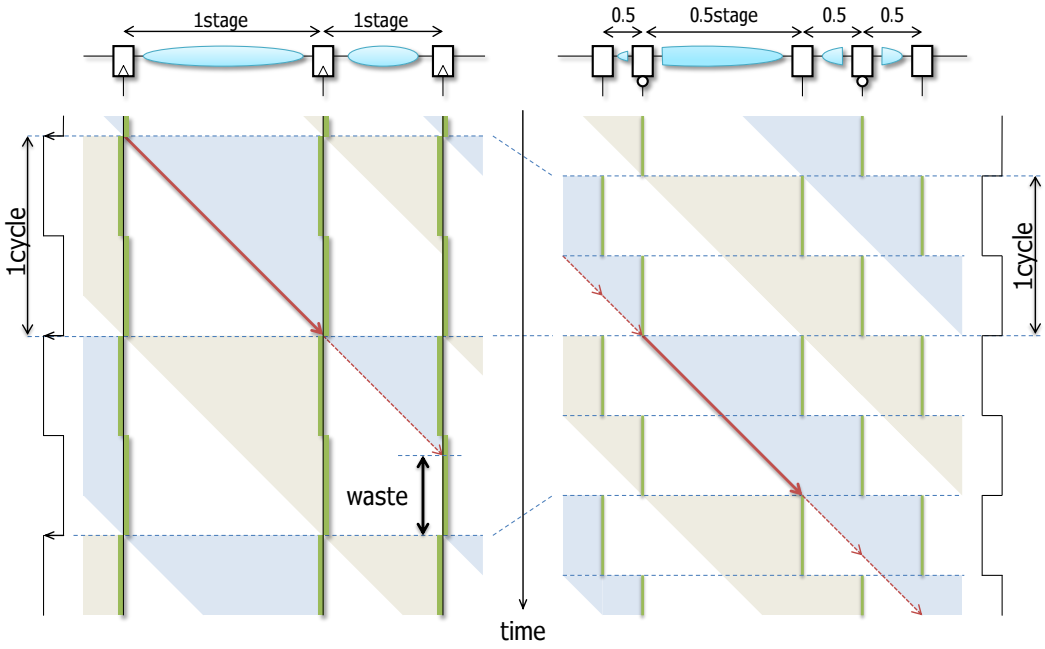

図 2.4: 静的タイム・ボローイング

のように，各ステージのクリティカル・パスの遅延の違いは，t-diagram 上ではステージの横幅の違いによって表される．

単相 FF 方式では，すべてのステージにおいて最大遅延制約 $1\tau/1$ ステージ

を満たさなければならない．そのため，クリティカル・パスの遅延のワースト，すなわち，各ステージのクリティカル・パスの遅延の中で最も大きいものによってサイクル・タイムが定まる．その結果，クリティカル・パスの遅延の小さいステージでは，クリティカル・パスが活性化した場合であっても，次のクロック・エッジを待つことになる．この待ち時間を，図中では waste と示した．単相 FF 方式では，この待ち時間を有効に活用する手段はない．

一方，二相ラッチ方式では，ラッチの開いている期間を利用することで，この制約を緩和することができる．図 2.4（右）に示されているように，正しく動作する条件はラッチが開いている期間にクリティカル・パスの遅延に対応する 45° の赤い直線が通ることである．同図（右）において，前述したクロッキング方式が正しく動作するための 2 つの条件が満たされていることを確認されたい．

同図（右）は，ステージ間の遅延のインバランスが最も大きい場合のもので，最も遅延の大きい 0.5 ステージに 1τ を費やしている．この 0.5 ステージにおいて，上流側のラッチが開くと同時に出発した 45° の赤い直線が，次のラッチが閉じる直前に到着している．したがって，これ以上この 0.5 ステージに時間を割り当てることはできない．一方で，その前後の遅延の小さい 0.5 ステージにはごく短い時間しか割り当てられていない．すなわち，遅延の長い 0.5 ステージは短い 0.5 ステージからサイクル・タイムの一部を借りている (borrow) という訳である．ただし，借りられた時間は決して返されることがない．

このタイム・ボローイングの結果，ステージ間の遅延がバランスしていない場合に，単相 FF 方式に比べてサイクル・タイムを短縮することができる．二相ラッチ方式の最大遅延制約は，特定の 0.5 ステージにおいては $1\tau/0.5$ ステージとなり，単相 FF 方式の $1\tau/1$ ステージに対して 2 倍となる．ただし，全てのステージにおける平均の遅延制約は，前述したように，$0.5\tau/0.5$ ステージである．

なお，設計においては，まずステージ間の遅延をバランスさせることが肝要であり，タイム・ボローイングの効果を積極的に利用することは推奨されない．この性質は，クロック・スキューに対する耐性に効果があり [11]，実際にはスキュー耐性のために採用されることが多いようである．

2.1.5 Razor

図 2.6（上）に，Razor FF の回路構成を示す [3]．Razor FF は，通常の FF（Main FF）と，Shadow Latch によって構成される．Shadow Latch には，Main FF へのそれより位相の遅れたクロックが供給されており，Main FF と Shadow Latch で 2 回，信号のサンプリングを行う．それらの値を比較して，異なっていればフォールトとして検出する．なお，フォールト検出後は，パイプライン・フラッ

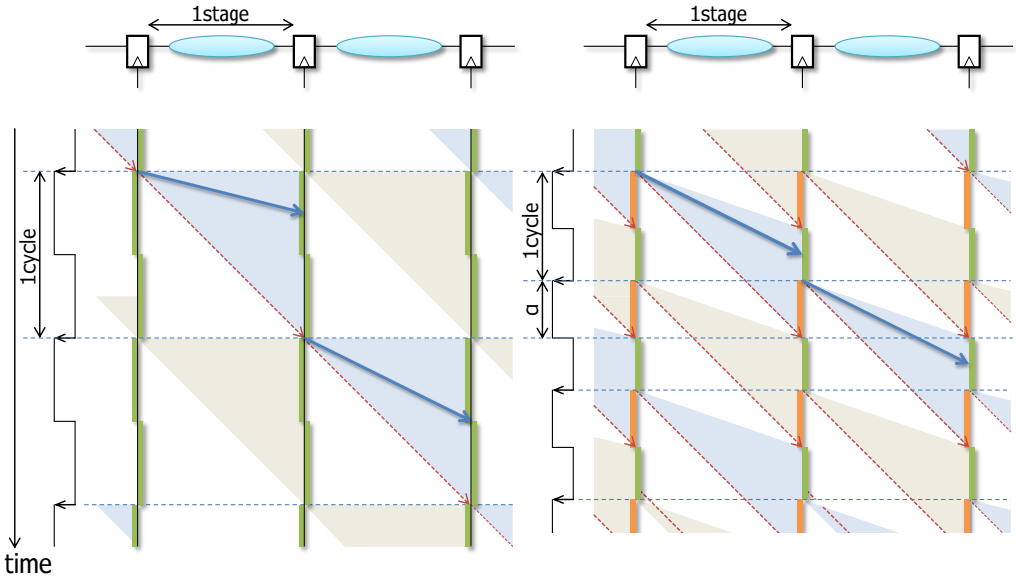

図 2.5: 単相 FF（左）と Razor（右）の t-diagram

シュなど，アーキテクチャ・レベルの手法によってフォールトからの回復が行われる [3, 9].

図 2.5 は単相 FF 方式と Razor の t-diagram を比較したものである．同図では Main FF と逆相の，すなわち，半周期遅れたクロックを Shadow Latch に供給している．t-diagram 上における FF の下の橙色の実線は，フォールトの検出ウィンドウを表している．検出ウィンドウの，上端で Main FF が，下端で Shadow Latch が信号のサンプリングを行い，その値を比較する．したがって，検出ウィンドウに実効遅延に対応する太矢印が到着していれば，フォールトとなる可能性がある．ただし，偶数回変化して元に戻った場合には，フォールトとならない．

Razor のようにフォールト検出を行うクロッキング方式では，t-diagram 上で正しく動作する 2 つの条件の 1 つ目は，以下のように修正される．2 つ目は変わらない：

1. クリティカル・パスの遅延を表す赤線でなくても，次のサイクルに次のステージへと至る矢印が描ける．

2. 矢印が存在し得る三角形の領域が，前後のフェーズのそれと重ならない．

最大遅延制約　単相 FF 方式では，クリティカル・パスの遅延に対応する 45° の赤線が次のクロック・エッジに間に合う必要があるため，最大遅延制約は $1\tau/1$ ステージ となる．

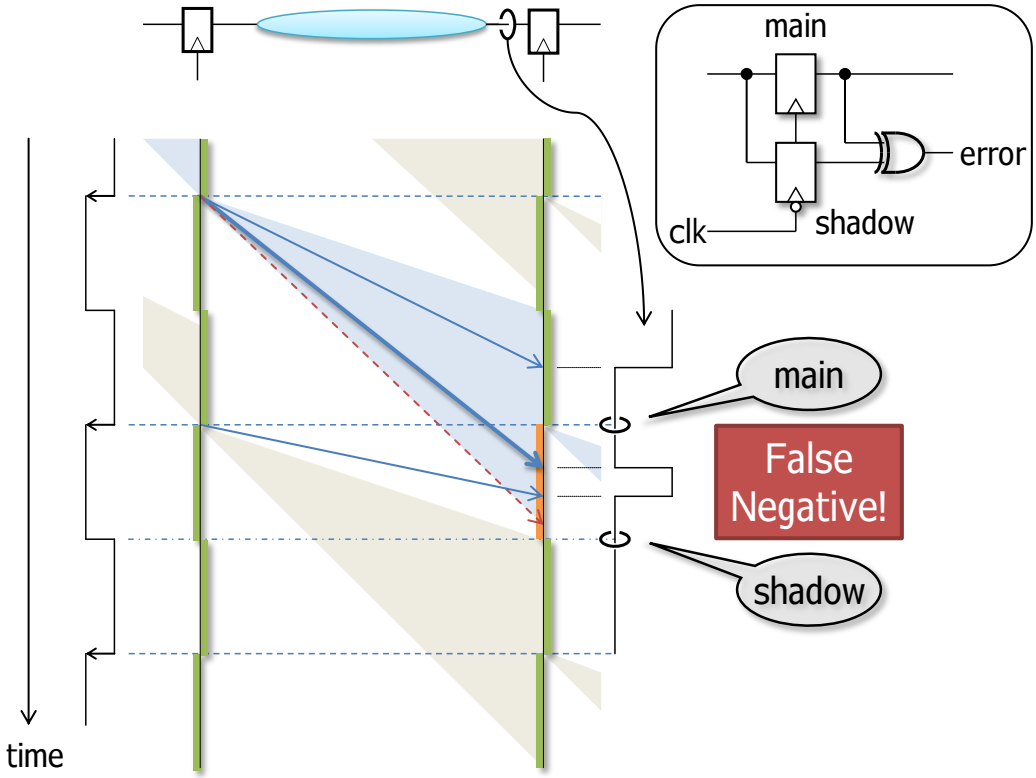

図 2.6: Razor の回路構成とショート・パス問題

一方，Razor では，クリティカル・パスの遅延に対応する 45° の赤線が検出ウィンドウの下端までに到着すれば，フォールトとして処理することができる．したがって，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．

実効遅延に対応する太矢印が検出ウィンドウの上端より先に到着していれば，フォールトとならない．すなわち，フォールトとならない最大遅延制約は $1\tau/1$ ステージ となる．

2.1.6 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，特有のショート・パス問題がある．

図 2.6 を用いて，Razor のショート・パス問題を説明する．Main FF と Shadow Latch の値を比較することでフォールトを検出する．Shadow Latch が正しい値を

サンプリングするためには，ロジックのショート・パスを通った信号が Shadow Latch 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．さもないと，図に示されているように，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，Shadow Latch が本来とは異なる値をサンプリングする可能性がある．その結果，誤検出 (false positive) となれば問題ないが，検出漏れ (false negative) となると致命的である．

このため Razor は，Razor 特有の最小遅延制約を生じる．図 2.6 では，Shadow Latch 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．

このように Razor には，最大と最小遅延制約の間に，サイクル・タイムに対する検出ウィンドウの割合 α を介して，直接的なトレードオフが存在する．

2.2 提案手法

本章では，二相ラッチ方式とフォールト検出を組み合わせたクロッキング方式を提案する．これにより，**動的タイム・ボローイング**が可能になる．以下，2.2.1 節で提案手法の回路構成を述べ，2.2.2 節以降で，既存のクロッキング方式との比較を行い，提案手法の特徴や優位性を示す．

2.2.1 回路構成

図 2.7 は提案手法の回路構成である．図 2.7 上は二相ラッチの回路の概略図である．ロジックのショート・パスとクリティカル・パスとが，あるゲート（図中○印）で合流した後，ラッチに接続されている．

図 2.7 下は提案手法の回路の概略図である．フォールト検出のために，各ラッチに逆相で動作する Shadow Latch とサンプリングされた値を比較する XOR ゲートを追加する．Razor で用いられる Razor FF の Main FF をラッチに置き換えた構造となる．ここでは便宜上，Razor Latch と呼ぶことにする．

そして，2.1.6 節で述べた Razor のショート・パス問題が起きないよう，ロジックに遅延を挿入する．遅延を挿入するのは Shadow Latch に至るショート・パスにだけ入れればよい．フォールト検出時は Shadow Latch が開き，Main Latch が閉じている状態であり，Main Latch は前サイクルの値を保持しているため，次サイクルのショート・パスの活性化の影響を受けないからである．

このため，ショート・パスとクリティカル・パスの合流するゲートを二重化し，Shadow Latch に至るショート・パスにのみ遅延を挿入する．Main Latch に

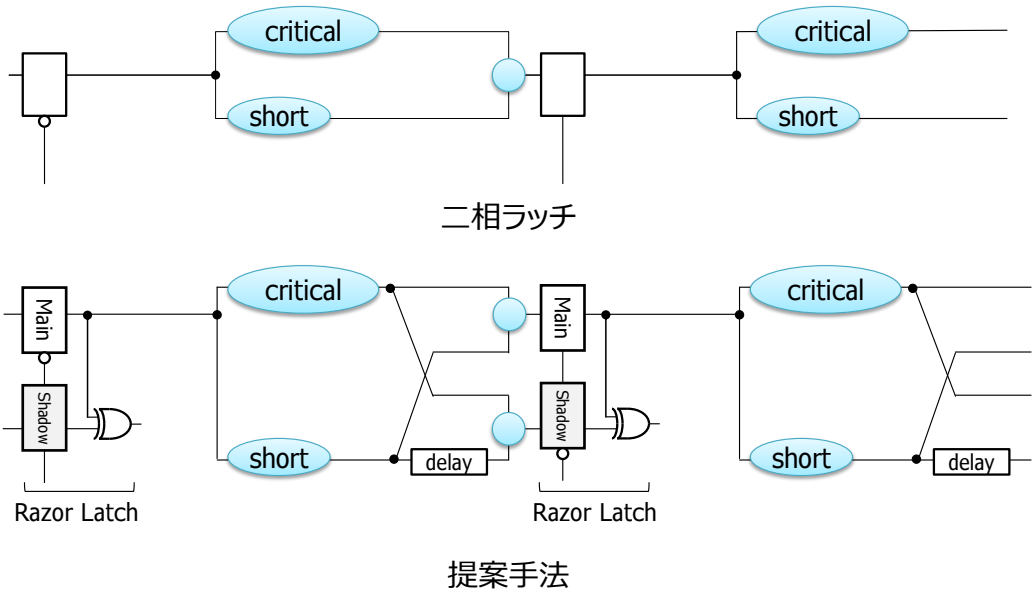

図 2.7: 提案手法の回路構成

至るショート・パスには遅延が挿入されていないので，ロジックの遅延分布がクリティカル・パスの遅延の方に偏る心配もない．

2.2.2 動的タイム・ボローイング

図 2.8 は，二相ラッチ方式と提案手法の t-diagram を比較したものである．2.1.3 節で述べた制約上，二相ラッチ方式では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，ラッチが開いている期間は原則使うことができない．各ステージでクリティカル・パスが活性化しなかったとしても，ラッチが開くまで信号の伝播は待たなければならない．

提案手法では二相ラッチ方式によって本来的には利用可能であったこのラッチの開いている期間を，フォールト検出を設けることにより利用する．これにより，動作時に各ステージで実効遅延を融通することが可能となる．

実効遅延を融通するとはどのようなことかについて，**図 2.9** において説明する．図 2.9 は提案手法の t-diagram を拡大したものである．説明のため，各パイプライン・ラッチに L_0，L_1，L_2 と名前を付ける．

以降，ラッチ L_{i-1} と L_i の間のステージにおける実効遅延の値 $E(i)$ を遅延の**支出**，ラッチが閉じている時間 $I(i)$ を遅延の**収入**と定義する．単相 FF 方式や Razor の場合，1 ステージ毎に $I(i) = \tau$，二相ラッチ方式や提案手法の場合，0.5 ステージ毎に $I(i) = 1/2\tau$ の遅延の収入がある．そして，ステージにおける遅延の**収支**を $D(i) = I(i) - E(i)$ で表し，$D(i) \geq 0$ の場合を遅延の**黒字**，$D(i) < 0$ の場合を遅延の**赤字**と定義する．

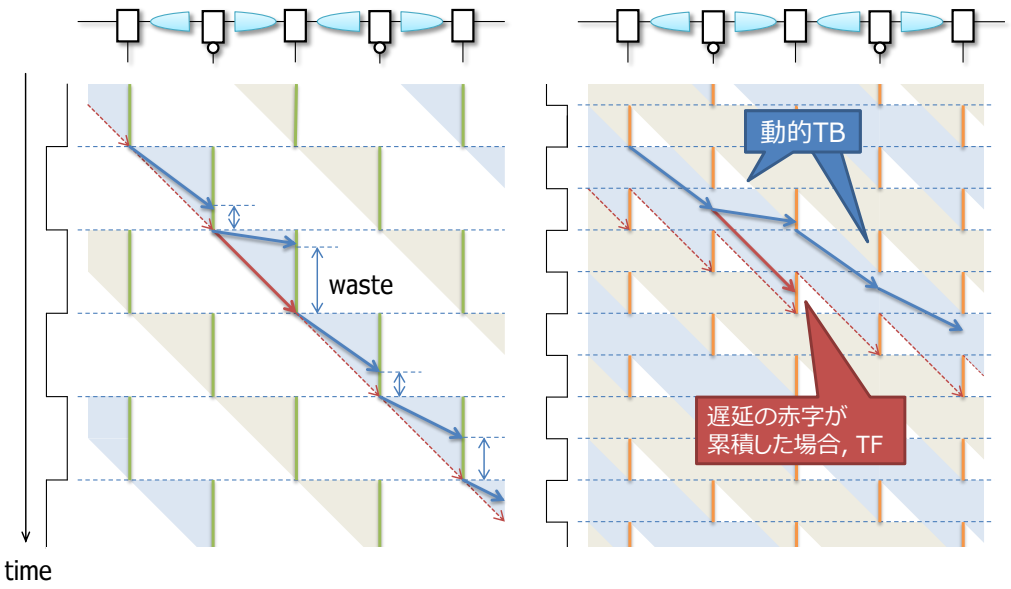

図 2.8: 二相ラッチ方式（左）と提案手法（右）の t-diagram

今，L_0 と L_1 の間のステージで L_0 の開いた瞬間から信号が伝播し，実効遅延 $E(1) = 3/4\tau$ で L_1 の値が変化したとする．この時，このステージにおける遅延の収支 $D(1)$ は，$D(1) = 1/2\tau - 3/4\tau = -1/4\tau$ となり，遅延の赤字が生まれる．仮に，実効遅延 $E(1) = 1/4\tau$ で L_1 の値が変化したとすると，$D(1) = 1/2\tau - 1/4\tau = 1/4\tau$ となり，遅延の黒字が生まれる．同図緑色の点線は，損益分岐（$D(i) = 0$ となる境界）を表している．

$D(1) = -1/4\tau$ の状態で，L_1 と L_2 の間のステージでクリティカル・パス（遅延 τ）が活性化した場合，このステージにおける遅延の収支 $D(2)$ は，$D(2) = 1/2\tau - \tau = -1/2\tau$ となる．この時，L_0 から L_2 までのステージにおける遅延の収支の**累積**は $\sum D(i) = D(1) + D(2) = -3/4\tau$ となる．提案手法では，このように遅延の赤字が累積し，$-\tau \leq \sum D(i) < -1/2\tau$ となった場合をフォールトとして検出する．

次に，L_1 と L_2 の間のステージで遅延の短いショート・パス（遅延 $1/8\tau$）が活性化した場合を考える．この時，$D(2) = 1/2\tau - 1/8\tau = 3/8\tau$ で，遅延の黒字が生まれ，遅延の収支の累積も $\sum D(i) = -1/4\tau + 3/8\tau = 1/8\tau$ の黒字となる．

このように提案手法ではラッチの開いている区間を利用することで，遅延の赤字の累積を解消することが可能である．入力ばらつきに着目した，実効遅延を融通させるこの時間の貸し借りのことを**動的タイム・ボローイング** と呼ぶ．t-diagram 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動的タイム・ボローイングの効果を表しているといえる．

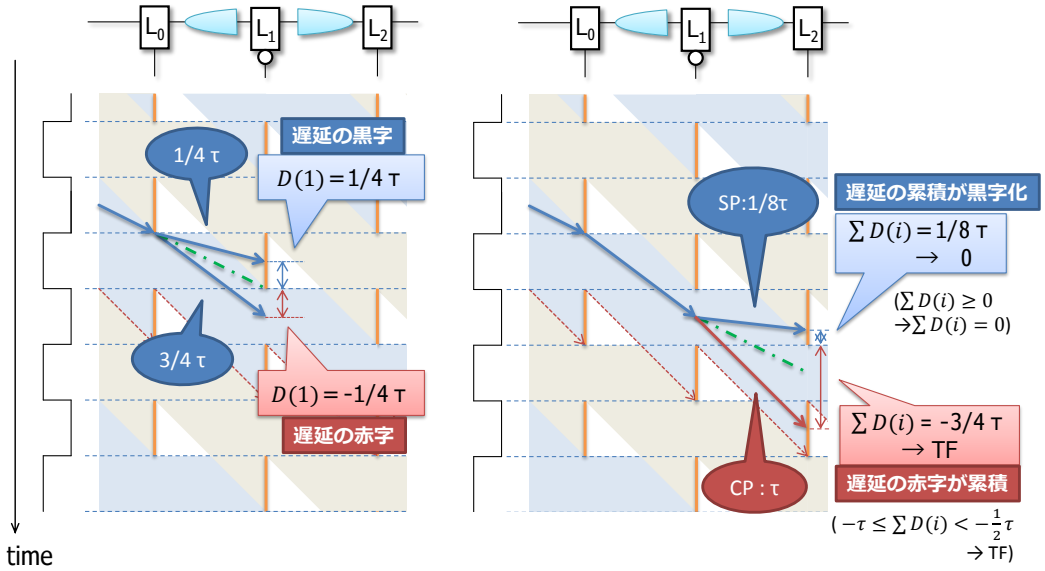

図 2.9: 動的タイム・ボローイング

なお，遅延の収支の累積が黒字になった場合は，ラッチ L_n が閉じている状態で値が変化したことになるので，次のステージに信号が伝播するタイミングはラッチの開く瞬間となる．そのため，$\sum D(i) \geq 0$ の場合，次ステージにおいて累積した黒字の分を捨て，$\sum D(i) = 0$ として見る．

最大遅延制約　再度，図 2.8 に着目する．提案手法では，遅延の赤字が累積し，$\sum D(i) = -\tau$ となった場合をフォールト検出限界となるようサイクル・タイムを定める．各ステージにおいて生じうる，遅延の赤字の最大値は $D(i) = -1/2\tau$ である．そのため，$\sum D(i) = -1/2\tau$ の状態からクリティカル・パスが活性化し，$\sum D(i) = -\tau$ になる場合をワースト遅延の境界と定める（図 2.8 右，赤点線）．すなわち，ラッチ L_{n-1} の閉じる上端からラッチ L_n の検出ウィンドウの下端までがワースト遅延の境界となる．

このようにすると，クリティカル・パスの遅延によって定められるワースト遅延の境界が t-diagram 上において階段状となり，サイクル・タイムを詰めることが可能となる．これにより，ラッチの開いている区間を利用できるだけでなく，サイクル・タイムを 0.5 ステージ分のロジックのクリティカル・パスの遅延によって決定できる．

このことから，提案手法の最大遅延制約は，$1\tau/0.5$ ステージ と表すことができ，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．

なお，提案手法の t-diagram においてフェーズが「混ざって」いるように見

えるが，前節で述べたショート・パスとクリティカル・パスの合流するゲートを二重化し，各フェーズの信号の経路を分けてあるため，実際にはフェーズの信号が「混ざる」ことはない．

2.2.3 Razor と提案手法の比較

動的タイム・ボローイングの効果は Razor と比較することで，さらに明確なものになる．**図 2.10** は Razor と提案手法の t-diagram である．

Razor は FF を用いたフォールト検出回路であるためタイム・ボローイングができない．前述したように，Razor は 1 ステージ毎に $I(i) = \tau$ の遅延の収入がある．しかし，仮に実効遅延の値が $E(i) \leq \tau$ でステージにおける遅延の収支が $D(i) \geq 0$ で黒字であっても，次のクロック・エッジまで待たなければならず，次のステージに遅延の黒字を持ち越すことはできない．そのため，各ステージにおいて独立に遅延の収支 $D(i)$ を見なければならない．

実効遅延の値が $E(i) > \tau$ とサイクル・タイムを超え，遅延の収支が $D(i) < 0$ で赤字となった時点で，必ずフォールトとして検出する．フォールトが検出されるごとに回復処理が行われるため，その回復オーバーヘッドは無視できないものとなる．

一方，提案手法では，ロジック上の全遅延の存在領域をラッチの開いている区間にも広げることで，複数ステージ間に渡る多段のパスが形成される．これにより，各ステージ独立で遅延の収支 $D(i)$ を見るのではなく，全ステージにおける遅延の収支の累積 $\sum D(i)$ を見ることが可能となる．

遅延の赤字の累積が $\sum D(i) < -1/2\tau$ となった場合にフォールトを検出する．動的タイム・ボローイングにより，あるステージでクリティカル・パスのような遅延の大きいパスが活性化したとしても，その後のステージで遅延の小さいパスが活性化することで，遅延の赤字の累積を減らし，フォールトの発生を抑えることができるのである．

Razor の限界と提案手法の効果　2.1.5 節において，Razor の最大遅延制約はサイクル・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 $\alpha\tau$ だけ改善されると述べた．ところが実際には，DVFS におけるマージンを削減する効果しかない．ここでは簡単な実験を例に，その理由を説明する．

サイコロを 2 個振り，出た目の積により実効遅延が決定するとしよう．この時のクリティカル・パスの遅延は，36 $(= 6 \times 6)$ である．サイコロの目の和ではなく積としたのは，入力ばらつきが遅延小に偏ること [7] を少しでも再現するためである．この場合，クリティカル・パスの活性化率は $1/36$ と [7] で示された値 $1/100$ よりやや大きく，目の積の期待値は 12.25 とクリティカル・パスの半分の 18 $(= 36 \div 2)$ よりやや小さい．

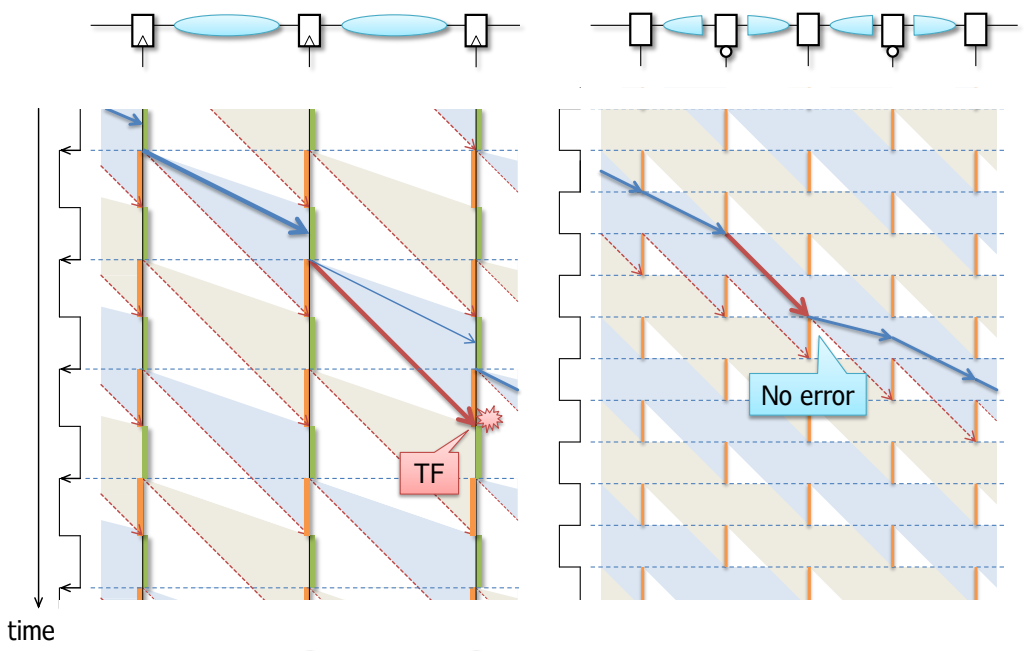

図 2.10: Razor（左）と提案手法（右）の t-diagram

Razor では，サイクル・タイムをτとすると,「目の積がτを超えた時」フォールトとなる．フォールトを発生することなく連続何サイクル動作を継続できるかを測った．その結果を**図 2.11** の Razor 1,10, 100 に示す．1, 10, 100 は，チップ内でフォールトを起こし得ると目されたパスの数である．Razor では，フォールトを起こし得るパスには，通常の FF に替えてフォールト検出を行う Razor FF を挿入する．したがって，1, 10, 100 は，通常の FF の代わりに挿入された Razor FF の数と考えてよい．

同図から分かるように，フォールトを起こし得るパス数が 1 の場合(Razor 1)には，サイクル・タイム$\tau = 30$程度まで削減しても，平均して連続 50 サイクル弱はフォールトを起こさずに動作することができ，実効遅延に基づいた動作が実現されたと言えないことはない．しかし，パスの数が 10 (Razor 10)，100 (Razor 100)の場合，$\tau = 36$からわずかでも削減すると，たちまち連続動作できるサイクル数はほぼ 0 となってしまう．これは，パスが 100 もあれば，1 つくらいは 6 のゾロ目を振るからである．この結果は，Razor では，クリティカル・パスで規定されるサイクル・タイムより短縮することは実際上できないことを示している．

提案手法の場合は,「出た目の積とサイクル・タイム（τ） をそれぞれ半分にし，その差（遅延の赤字）の累積が$1/2\tau$を超えた時」フォールトとなる．仮に，クリティカル・パスの積の目が出たとしても，次のステージで積の値が小さければ，大幅に遅延の借金を解消でき，フォールトが発生しない．

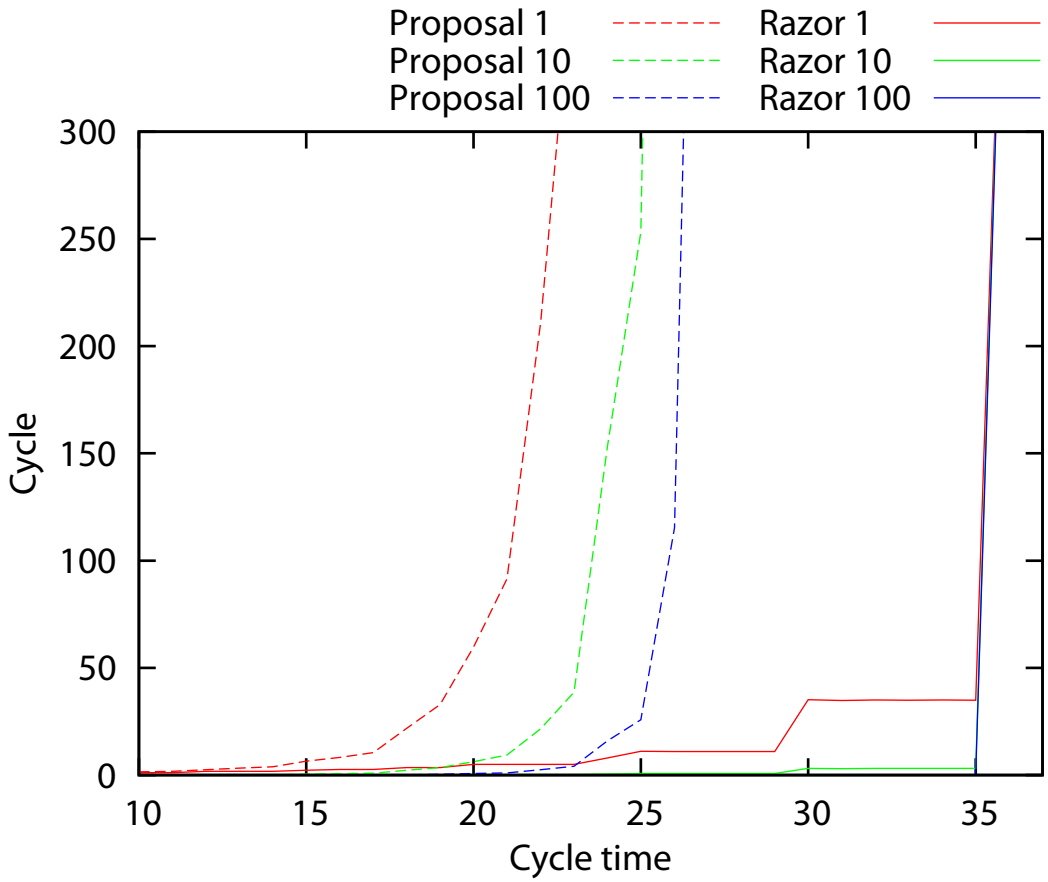

図 2.11: Razor の限界と提案手法の効果

提案手法に適用した結果が，図 2.11 の Proposal 1,10, 100 である．このように，パスの数が 100 であっても $\tau = 26$ 程度までサイクル・タイムを削減することができ，遥かに Razor を上回る．実際の実効遅延の分布は，サイコロ 2 個の目の積よりも遅延小に偏っているので，更なる削減が期待される．

2.2.4 既存のクロッキング方式との比較

表 2.1 は，単相 FF 方式，二相ラッチ方式，Razor，そして提案手法のタイミング制約をまとめたものである．各ステージのクリティカル・パスの遅延は均等であるとし，単相 FF 方式の動作周波数を f とする．

単相 FF 方式はステージ間で回路遅延を融通できない方式であり，フォールトも検出できないため，一番遅延の大きいステージのクリティカル・パスの遅延に合わせて動作周波数が決定する．言わば,「クリティカル・パス遅延のワースト」で動作周波数が決まる方式といえる．

二相ラッチ方式は静的タイム・ボローイングが可能になる方式であり，設計

表 2.1: 既存のクロッキング方式との比較

	ワースト・ケース設計	動的 TF 検出・回復	
	最大遅延制約	最大遅延制約	TF 検出制約
単相 FF	通常 ×ステージ間で時間を融通できない ×動作時に赤字が出たら暴走	Razor ×ステージ間で時間を融通できない ○動作時に赤字が出たら破綻・再建	
	ワースト遅延 各ステージ: 1τ	ワースト遅延 各ステージ: (1+α) τ	実効遅延 各ステージ: 1τ
二相ラッチ	静的タイム・ボローイング ○設計時にステージ間で時間を融通 ×動作時に赤字が出たら暴走	動的タイム・ボローイング ◎動作時にステージ間で時間を融通 ◎赤字が累積したら破綻・再建	
	ワースト遅延 累積の平均: 1τ	ワースト遅延 各ステージ: 2τ	実効遅延 累積の平均: 1τ

時にステージ間で回路遅延を融通できる．2.1.3 項で述べたような「クリティカル・パス遅延の累積」で動作周波数を決定できる．ところが各ステージのクリティカル・パスの遅延がバランスしている場合，ラッチの開いている区間を生かすことができないので，動作周波数は単相 FF 方式と変わらず f である．

Razor FF は，フォールト検出により，一番遅延の大きいステージのクリティカル・パスの活性化を許すが，フォールト検出限界を超えないように設計する方式である．そのため実効遅延での動作が可能となり，Razor は「実効遅延のワースト」で動作周波数が決定する．動作周波数は検出ウィンドウの割合を α とすると，$(1+\alpha)f$ となる．

提案手法は，そのフォールト検出を二相ラッチ形式に適用したものである．ラッチの開いている区間を利用できるため,「実効遅延の累積」で回路を動かすことが可能になる．また単相 FF 方式の 1 ステージに相当するロジックが二分され，フォールト検出によりワースト遅延の境界が階段状となり，サイクル・タイムを詰めることが可能となる．これにより，単相 FF 方式の半ロジック分の動作周期，すなわち $2f$ の動作周波数を実現できる．

2.3 評価

2.2 章で述べた回路構成で正しく動作するか，実際に動作周波数が向上す

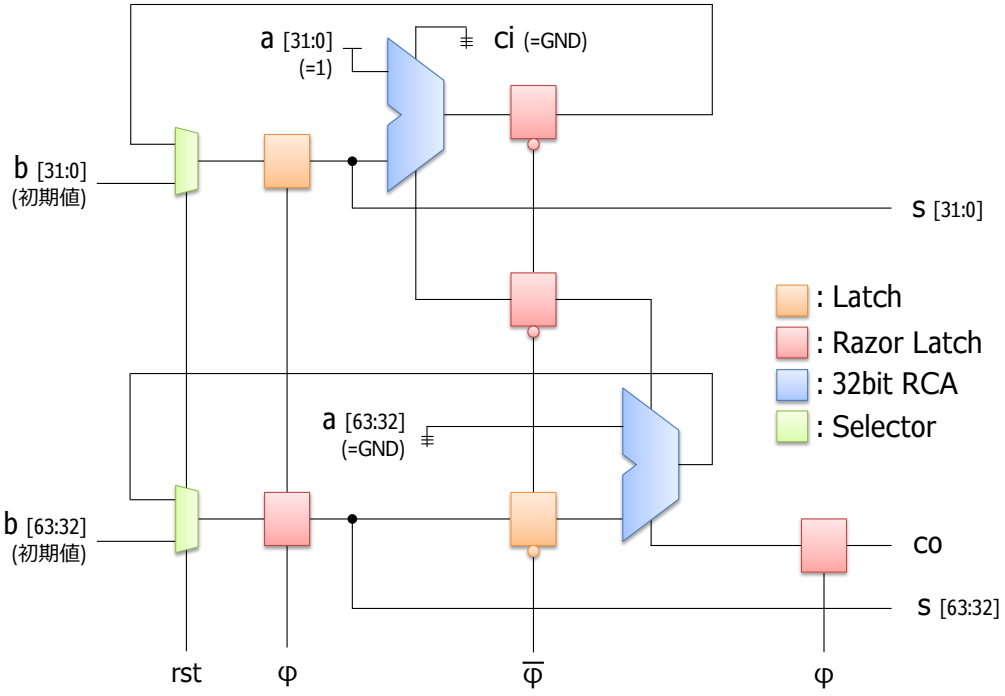

図 2.12: 提案手法を適用した 64bit アップ・カウンタ

るかの理論の検証の為， 提案手法を適用した 64bit のリプルキャリー・アダー（Ripple Carry Adder : RCA）を用いたアップ・カウンタを FPGA に実装し，動作の確認と各種クロッキング方式との比較を行った．FPGA は Xilinx 社の Virtex-6 XC6VLX760-2ff1760 を用いている．**図 2.12** にアップ・カウンタのブロック図を示す．ここでのセレクタは初期化の役割を果たしている．

64bit の RCA を下位 32bit と上位 32bit の 2 つに分ける．キャリー・ネットワークにラッチを挟むことにより，ロジックを均等に二分し，二相ラッチ方式を適用する．

そして，フォールトが起こりうるステージである，RCA の出力をサンプリングするラッチ を Razor Latch に置き換える．部分和 s の出力をサンプリングした後のステージは，出力をそのまま次のラッチに流す遅延の小さいステージであるので，Razor Latch に変更する必要はない．

図 2.13 は 6bit の RCA を例として Razor のショート・パス問題を回避するための遅延素子を挿入した図である．図中の白抜きの長方形はキャリーを出力する多数決回路を，網掛けした長方形は遅延素子を表す．

RCA では，ci から co に至るキャリー・ネットワークがロジックのクリティカル・パスとなり，入力 a, b からのパスが概ねショート・パスである．すなわち，RCA のショート・パスとクリティカル・パスは部分和 s を出力する XOR ゲートで合流しているといえる．これを二重化し，Razor Latch の Main Latch

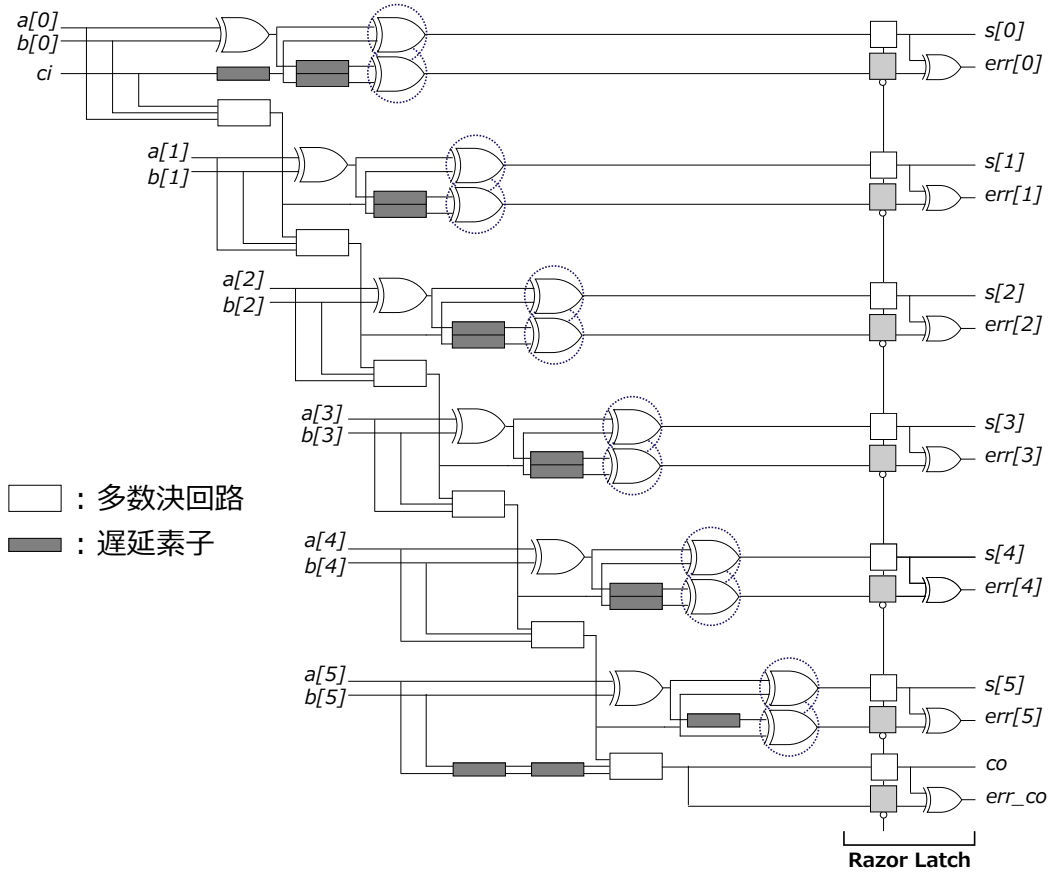

図 2.13: RCA への適用

と Shadow Latch に至るパスを分ける．

Virtex-6 では，LUT（Look Up Table:真理値表）を 1 つのモジュールとしてインスタンシエートすることができる [12]．XOR ゲートと多数決回路はゲートレベルでの段数に違いがあるが，LUT のパラメータを書き換えることにより，同じ遅延を持つ同一の LUT で表現することが可能である．これにより，パス上の LUT の数でパスの遅延を見積もることができる．遅延素子はバッファ回路を LUT で表現したものである．

クリティカル・パス上の LUT の段数は 6 段である．そのため，Shadow Latch に至るショート・パス上の LUT の段数が 3 段以上になるように遅延を挿入する．通常出力側に遅延を挿入すればよいが，上位ビットに行くにつれ，クリティカル・パス自体の遅延が伸びてしまう可能性がある．その際は入力側にも挿入箇所を振り分けることで対処する．

同様の方法で図 2.12 の 2 つの 32bit RCA に遅延を挿入し，提案手法の適用を行った．以下にその評価結果を示す．

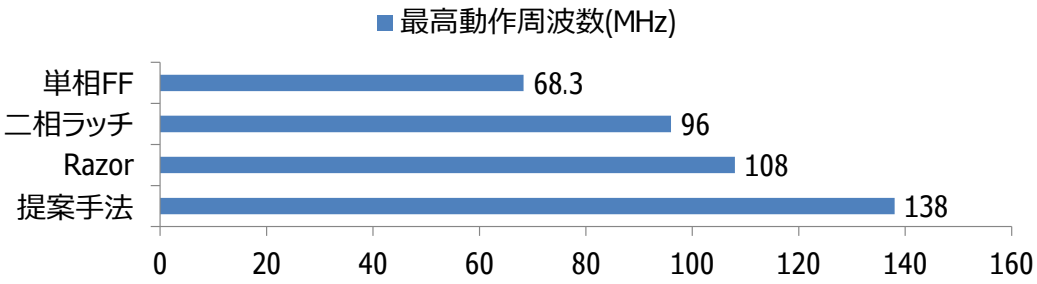

図 2.14: 最高動作周波数の比較

2.3.1 動作周波数

64bit の RCA を用いたアップ・カウンタに各クロッキング方式を適用し，ある特定のビットが変化しなくなる最高動作周波数を測定した．FPGA 基盤上の LED にアップ・カウンタの部分和の出力とフォールト検出のエラー信号を表示させ，PLL 設定スイッチで周波数を調整して評価を行った．フォールト検出を備えた Razor と提案手法においては，フォールトの検出限界を最高動作周波数として測定している．**図 2.14** はその結果である．提案手法の最高動作周波数は 138MHz となり，単相 FF 方式の 2 倍，二相ラッチ方式の 1.4 倍，Razor の 1.3 倍の動作周波数を計測できた．

単相 FF 方式や Razor に対しては，理論値に近い性能向上が見られたが，二相ラッチ方式の動作周波数が予想よりも高めに出た．その理由の一つに，今回適用したアップ・カウンタという回路の特殊性が考えられる．

アップ・カウンタではキャリーが伝播するステージの前後では必ず最下位ビットが 0 から 1 に変化するだけで，小さい遅延でステージを通過することができる．そのためワースト遅延以上のパスが仮に活性化しても，ラッチの開いている区間に信号が通りさえすれば，次のラッチのサンプリングには間に合う．言わば，実効遅延の貸し借りがフォールト検出という保障がない状況で行われていた結果であると推測される．

2.3.2 回路面積のオーバーヘッド

二相ラッチの回路の総 LUT 数は 292 であるのに対し，提案手法の回路の総 LUT 数は 2062 にも及ぶ．これは RCA における遅延素子が原因である．今回の回路は提案手法が実際に動作するかを確認するための回路であるため，遅延が $O(n)$ であるが，ロジックを二分しやすい RCA を用いて動作確認・評価を行った．実際には遅延が $O(logn)$ のキャリールックアヘッド・アダー（Carry Look-ahead Adder : CLA）が用いられるため，遅延素子の挿入における回路面

積の増加は抑えられると考えられる．**図 2.15** は 8bit の CLA に提案手法を適用した際の図である．

一般的に CLA はキャリールックアヘッド・ジェネレータのトゥリー接続によって実現される．ロジックを二分する点は最上位のキャリールックアヘッド・ジェネレータから折り返す部分である．そこで 2bit のキャリールックアヘッド・ジェネレータを，各桁の g (generate)，p (propagate) をまとめる LUT と，g, p から c (carry) を出力する LUT の 2 つに分割してトゥリー接続を開いた構造にする．

トゥリー接続から折り返す部分にラッチを挿入し，クリティカル・パスを通る信号をサンプリングするラッチを Razor Latch に変更する．ラッチの挿入位置によって前半部と後半部が均等に分割されていないのは，ロジックのクリティカル・パスを全て 1 つのラッチにまとめ，余分な Razor Latch 化を抑えるためである．

そして，ショート・パス問題に対処するため，Razor Latch に至るショート・パスに遅延を挿入する．図 2.15 では，ショート・パスの遅延をクリティカル・パスの遅延に合わせるように挿入している．前半の部分においては，Razor Latch に至るパスが全てクリティカル・パスと同じ遅延を持つため，遅延を挿入する必要はない．このようにして，遅延の挿入による回路面積の増加は抑えられる．

2.4 関連研究

本章では提案手法の関連研究として，**ReCycle** [13]，**TIMBER** [14]，**Bubble Razor** [15] を紹介する．

2.4.1 ReCycle

ReCycle は，cycle time stealing [16] と呼ばれる技術をプロセッサ・パイプラインに適用したものである．**図 2.16** はステージ間の遅延がバランスしていない場合の単相 FF 方式（左）と ReCycle（右）の t-diagram である．

2.1.3 節で述べたように，通常の単相 FF 方式では遅延の大きいステージのクリティカル・パスの遅延によってサイクル・タイムが定まり，遅延の小さいステージでは，サイクル・タイムに無駄が生じてしまう．二相ラッチ方式では，ラッチを置く位置を変更することで，ステージ間で時間を融通していた．

ReCycle は，遅延の大きいステージのロジックのパス遅延を予め解析する．これにより，そのステージの値をサンプリングする単相 FF のサンプリング・タイミングを，最短パスの遅延分まで遅らせることが可能となり，遅延の小さいステージのサイクル・タイムを分配し，遅延の大きいステージに 1 サイクル以上をかけることで，サイクル・タイムを短縮する手法ということができる．

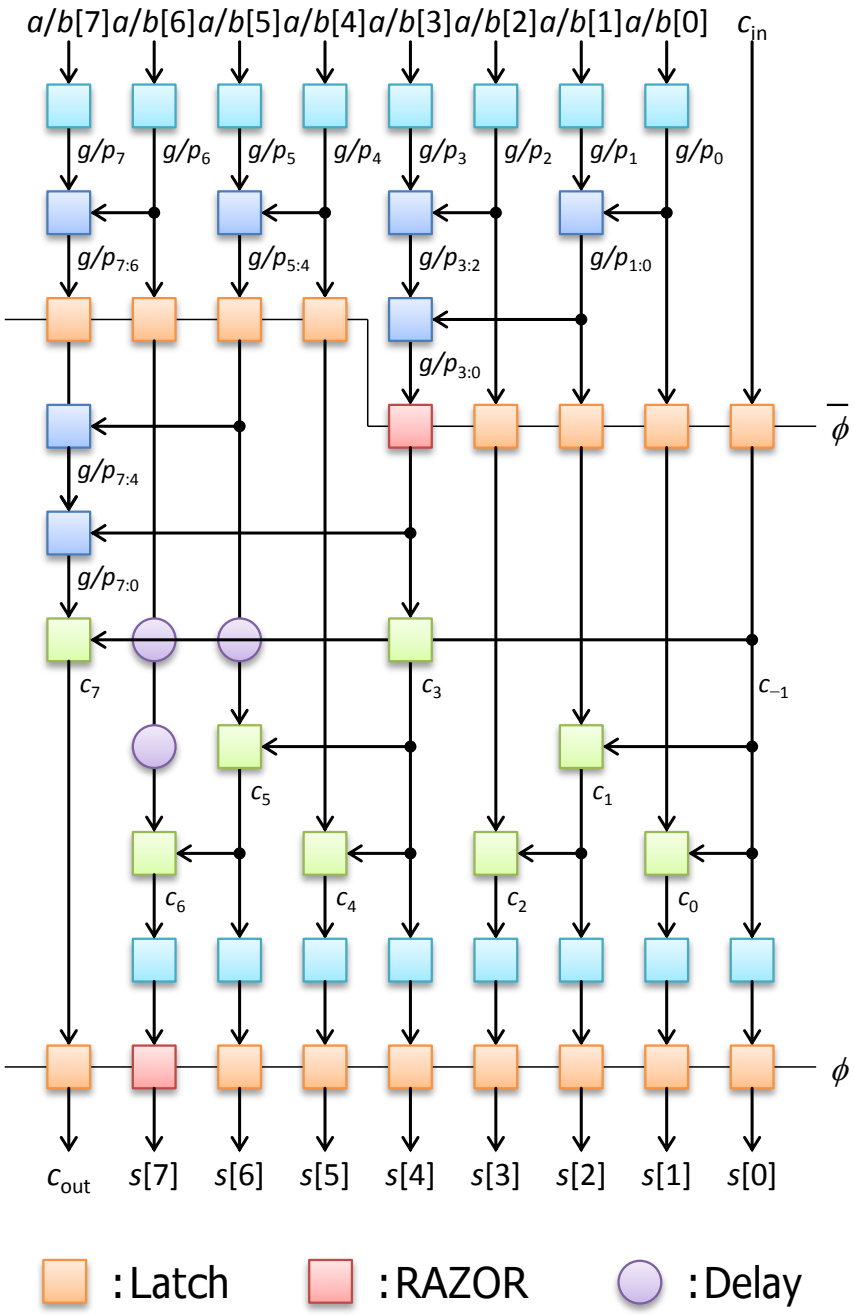

図 2.15: CLA への適用

2.4.2 TIMBER

[14] にはまず，チップ内におけるクリティカル・パスの分布が記されており，クリティカル・パス遅延のステージが連続する確率は極めて小さい，すなわち，クリティカル・パス遅延をもつステージの次のステージは遅延の小さい

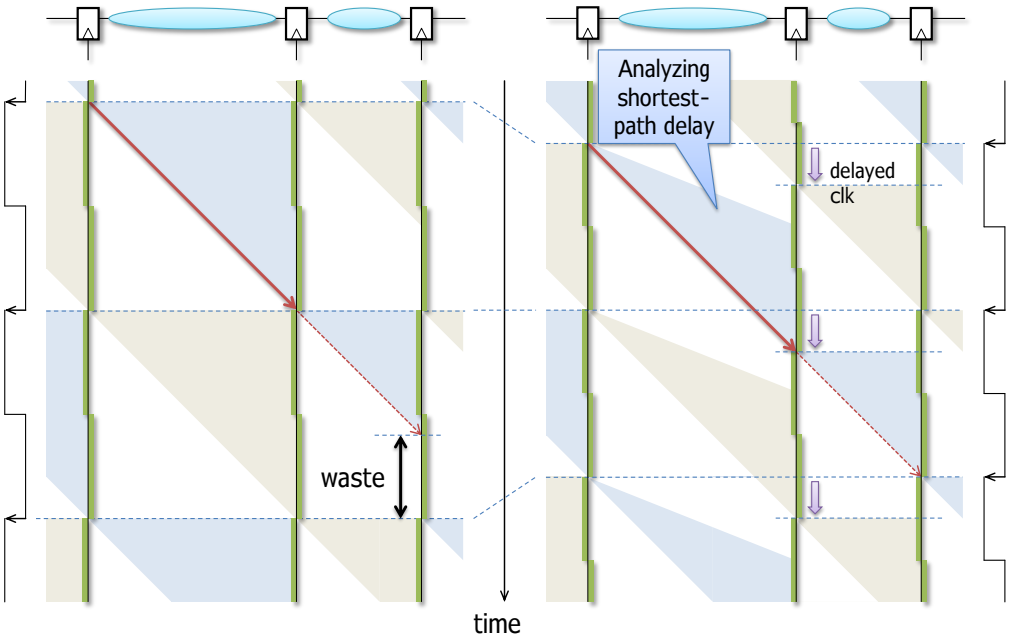

図 2.16: 単相 FF（左）と ReCycle（右）の t-diagram

ステージであることが多いということが示されている．このことを利用して，TIMBER は単相 FF 方式の検出ウィンドウにラッチの開いている区間を作ることによってフォールト検出の発生回数を抑えることを提案している．

図 2.17 に TIMBER の t-diagram を示す．TIMBER はサンプリング・タイミングが異なるマスター・ラッチを複製し，セレクタによりスレーブ・ラッチへの出力を切り替える．これにより，単相 FF 方式ながらラッチの開いている区間ができ，仮にクリティカル・パスが活性化しても次のステージでシグナルを早く伝搬させることで，次段のサンプリングに間に合わせることが可能である．

仮に 2 ステージ連続してクリティカル・パスが活性化した場合は，マスター・ラッチの値を比較してフォールトを検出する．検出時には，アーキテクチャ・レベルによる回復ではなく，セレクタの制御信号によりラッチの開いている区間を広げる回路レベルの処置が施される．

TIMBER の最大遅延制約は，ラッチの開いている区間の割合を β とすると，$(1+\beta)\tau/1$ ステージ となる．

図 2.18 は TIMBER と提案手法を比較した t-diagram である．TIMBER は単相 FF 方式の位相を遅らせた分，検出ウィンドウの割合が多くなり，提案手法や Razor に比べ，2.1.6 節で述べたショート・パス問題が厳しくなる．動作中においても最小遅延制約が変動するため，ショート・パスが満たすべき遅延制約はより厳しいものとなってしまう．

そもそも，TIMBER は半サイクルの間に，ラッチの開いている区間と検出する区間の二つを設けているため，ラッチの開いている区間の割合 β は，Razor

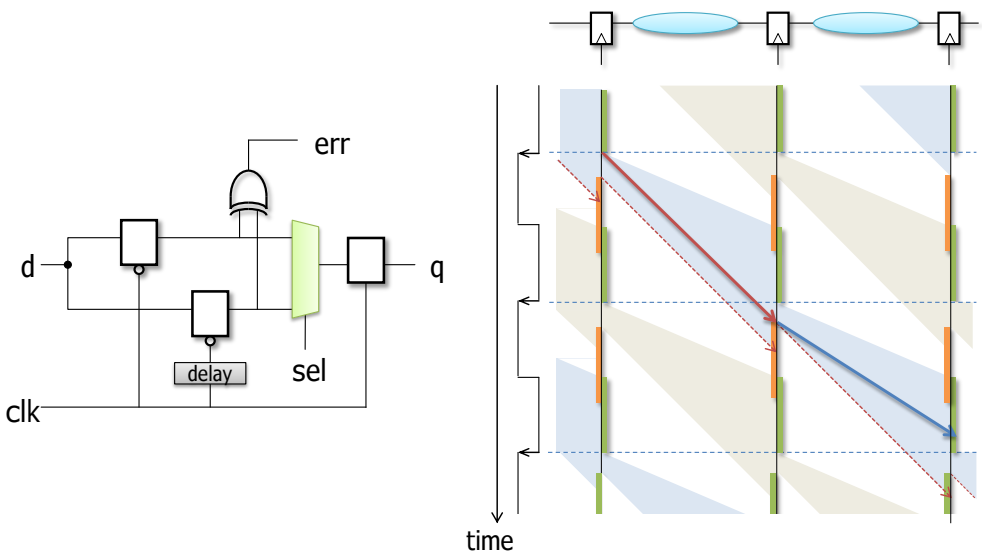

図 2.17: TIMBER の回路構成と t-diagram

などの検出ウィンドウの割合 α よりも小さい．そのため，Razor よりも理論上のサイクル・タイムが短縮されない．

さらに，フォールト検出時のクロックを遅らせる処置が遅延素子をクロックに挿入することで行われており，現実的ではない．遅延素子における遅延が固定値となるため，DVFS などにより周波数を変動させると，期待通りのクロックが生成されないことが予想される．

2.4.3 Bubble Razor

Bubble Razor は二相ラッチ方式にフォールト検出を組み込むという，我々の提案手法と類似したアプローチをとっているが，フォールトの検出ウィンドウのとり方など，動作は大きく異なる．**図 2.19** は Bubble Razor の動作を示した t-diagram である．

我々の提案する手法とは異なり，Bubble Razor ではラッチの開いている区間において値が変化した場合をフォールトと定め，検出ウィンドウを設けている．（図 2.19a）

検出ウィンドウにかかるパスが活性化した場合（図 2.19b-1），クロック・ゲーティングにより，次段のラッチの開いている区間を閉じる．（図 2.19b-2）この閉じている区間を [15] では，Bubble と呼んでいる．これにより，フォールトの発生した次のステージで仮にクリティカル・パス遅延で信号が伝播したとしても，信号をサンプリングすることが可能になり， silent error を回避できる．

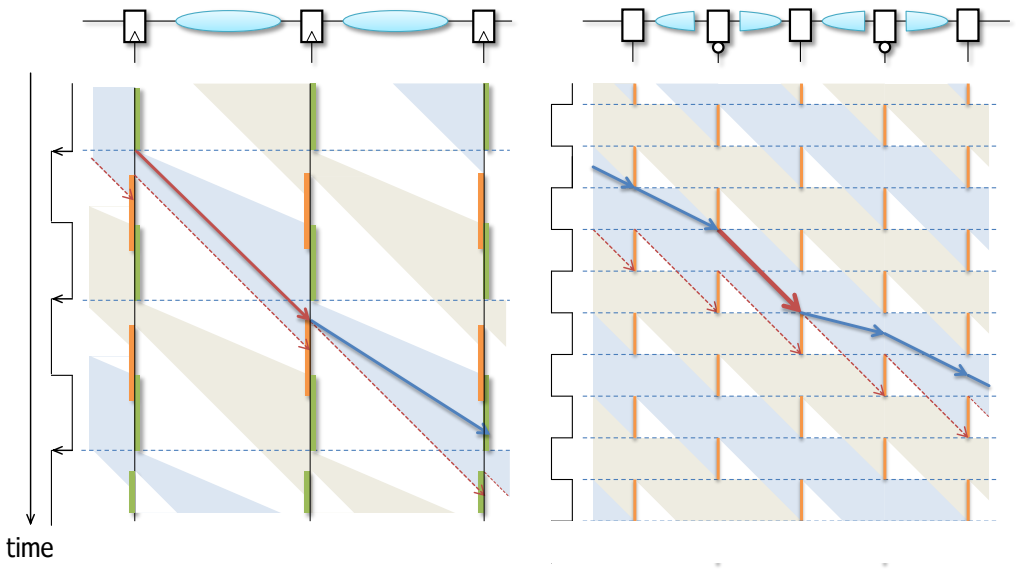

図 2.18: TIMBER（左）と提案手法（右）の t-diagram

しかし，次段のラッチを閉じただけでは，次のフェーズの信号と混ざる問題や，前のフェーズの信号が伝播してしまう問題が生じる．(図 2.19b-3) そのため，Bubble Razor では Bubble が生じたラッチの前後のラッチへと 0.5cycle 遅れて Bubble を伝搬させ続けることにより対処する．(図 2.19b-4)

図 2.19b-5 のように，前のフェーズでもフォールトが発生した場合， Bubble が前後のラッチから同時に到達することがある．この場合は Bubble の伝搬を止め，必要以上にラッチを閉じないように設計されている．これにより，ループ回路やフォワーディングパスへの対応を行っている．

Bubble Razor の最大遅延制約は検出ウィンドウの割合を α とすると，$(0.5+\alpha)\tau/0.5$ ステージ，フォールトとならない最大遅延制約は $0.5\tau/0.5$ ステージとなる．

図 2.20 は Bubble Razor と提案手法を比較した t-diagram である．Bubble Razor と提案手法の最高動作周波数の理想値は共に同じであるが，Bubble Razor は二相ラッチ方式を採用しているにも関わらず，検出ウィンドウをラッチの開いている区間に設けたことにより，通常動作時において，タイム・ボローイングが出来なくなっている．これにより，2.2.3 節で述べた Razor と提案手法の比較同様，Bubble Razor はクリティカル・パスに近い遅延のパスが活性化した場合，必ずフォールトが検出され Bubble が発生する．結局のところ，Bubble Razor はクリティカル・パスの遅延以上にサイクル・タイムを短縮できず，提案手法と比べて，動作周波数は向上できないものと予想される．

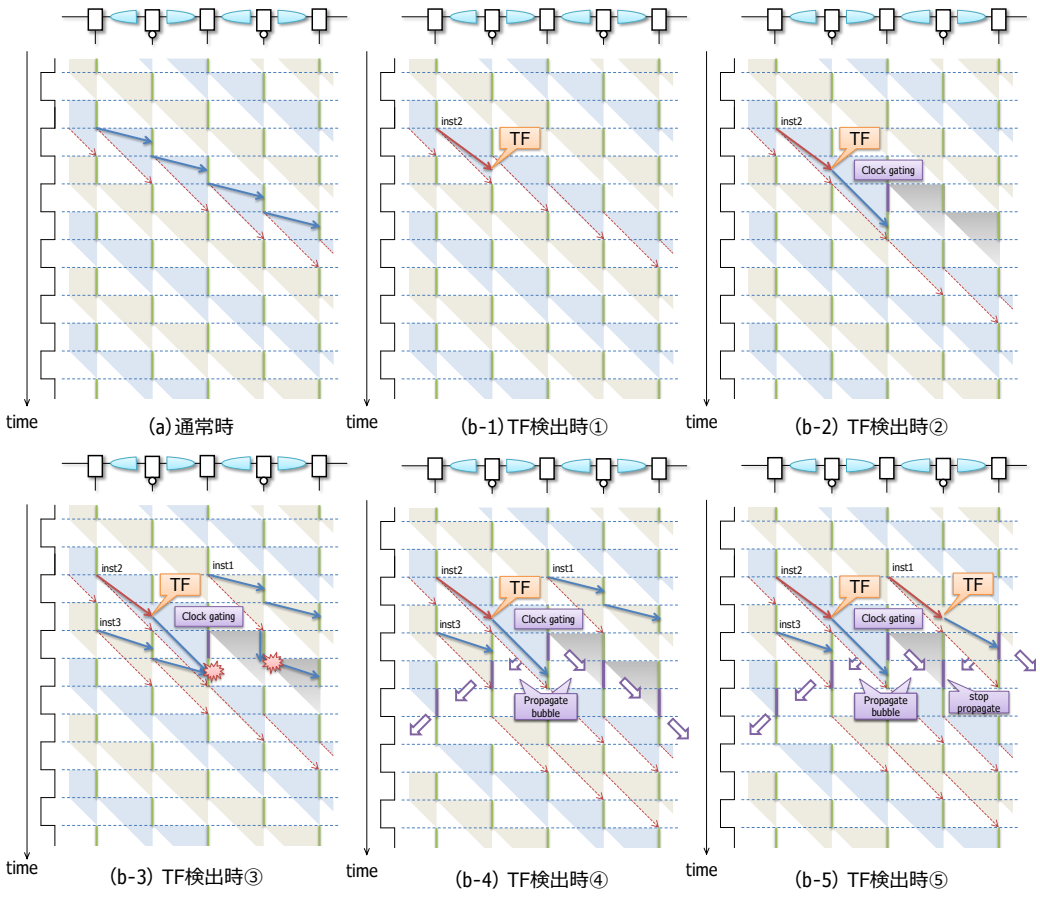

図 2.19: Bubble Razor の t-diagram

2.5 本章のまとめ

本章では，二相ラッチと Razor を組み合わせた動的タイム・ボローイングを可能にするクロッキング方式を提案し，簡単な回路に適用することで評価を行った．ステージ間でワースト遅延ではなく実効遅延を融通することができ，さらにフォールト検出により，半ロジックのクリティカル・パス遅延で動作周波数を決定できるため，通常の 2 倍の動作周波数で動作させることが可能となる．

現在，より一般的な CLA に提案手法を適用しており，その知見の元，ロジックの全パスの遅延に基づいて，回路に提案手法を自動で適用するツールを作成中である．また，SRAM に提案手法を適用した場合の問題点や解決策においても考察を行っている．

最終的には次章で述べる Out-of-Order プロセッサにおけるフォールトからの回復手法 [9] や，NORCS [17] などに代表される，我々の研究室で提案している手法を適用したプロセッサを制作予定である．

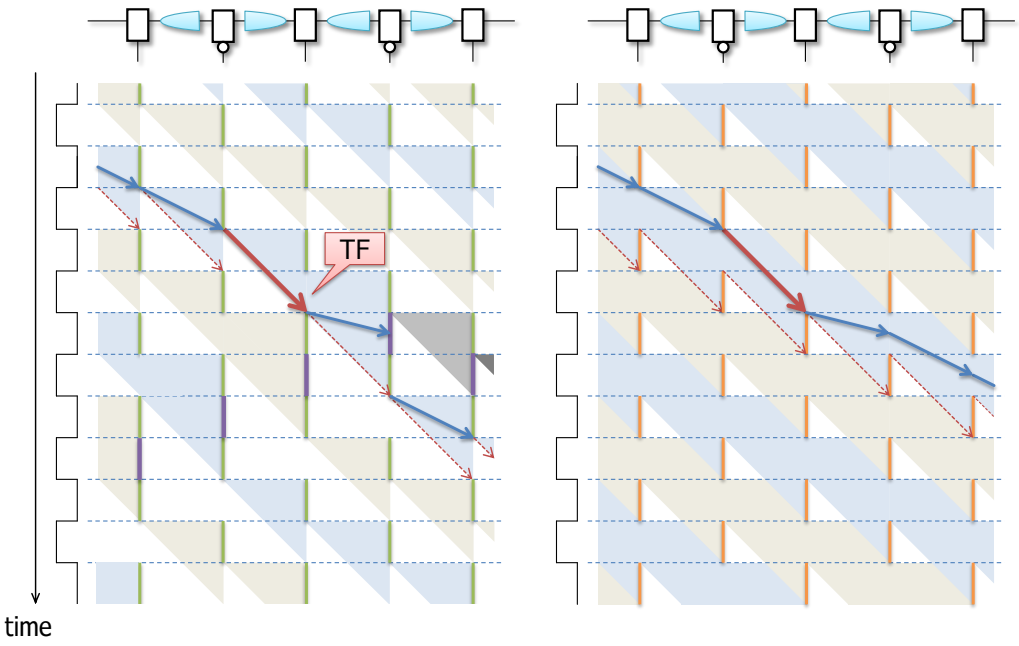

図 2.20: Bubble Razor（左）と提案手法（右）の t-diagram

第3章　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

続く 3.1 節では，まず Razor Ⅱ を含むタイミング・フォールト検出/ 回復技術についてまとめる．その後 3.2 節で，out-of-order スーパスカラ・プロセッサの実際について説明し，Razor Ⅱ の考え方では不十分であることを示す．続く 3.3 節で，以前までの我々の提案の回復技術について述べる．3.4 節では，この検出/回復方式について，物理的な構成や具体的な実装方法，回復のペナルティなどの詳細について検討する．3.5 節では，シミュレーションによって，この回復のペナルティが IPC に与える影響を評価する．

3.1　タイミング・フォールト検出/ 回復技術

本節では，プロセッサを対象としたフォールト検出/ 回復技術の一般的なことがらから始めて，Razor Ⅱ [18] の回復技術に特有のことがらについてまとめる．

前述したように，フォールト検出/ 回復技術は，一部の例外 [15] を除いて，フォールトを検出する回路レベルの技術と，検出後に回復を行うアーキテクチャ・レベルの技術の 2 つからなる [14, 13]．本章では特にアーキテクチャ・レベルの技術について掘り下げる．3.1.1 節でアーキテクチャ・レベルの回復技術の概要について述べ，3.1.2 節において，Razor Ⅱ の回復技術に特有の点についてまとめる．

3.1.1　アーキテクチャ・レベルの回復技術

フォールト検出/ 回復技術を適用するうえで，プロセッサは，他の一般のハードウェアより対応が容易である．それは，プロセッサにはアーキテクチャ・ステートが定義されているからである．

アーキテクチャ・ステート　アーキテクチャ・ステート（**Architecture State : AS**）（以下では AS とする）は通常，命令セット・アーキテクチャにおいて定

義され，PC（を含む PSW）と（論理）レジスタ・ファイル，および，その他の制御レジスタからなる．AS は，主に OS とのインタフェースの一部をなし，たとえば，コンテクスト・スイッチ時にセーブ/レストアされる対象となる．

一方，マイクロアーキテクチャにおいては，PC や（論理）レジスタ・ファイルに加えて，主記憶も AS に含めると都合がよい．そのようにすると，命令の**コミット**を，命令（の実行結果）による AS の不可逆的な更新と定義できるからである．以下，本稿では，この拡張された定義を採用する．

アーキテクチャ・レベル回復技術の概略　プロセッサにおけるフォールト検出/回復は，AS を利用して，次のようにすればよい：

(1) コミットの停止　：コミットを停止することによって，AS をフォールトから保護する．

(2) フォールトの影響の除去　：何らかの包括的な方法によってパイプラインからフォールトの影響を取り除く．その後，保護された AS から実行を再開する．

アーキテクチャ・レベル回復技術のパイプライン　**図 3.1 (a)** に，アーキテク

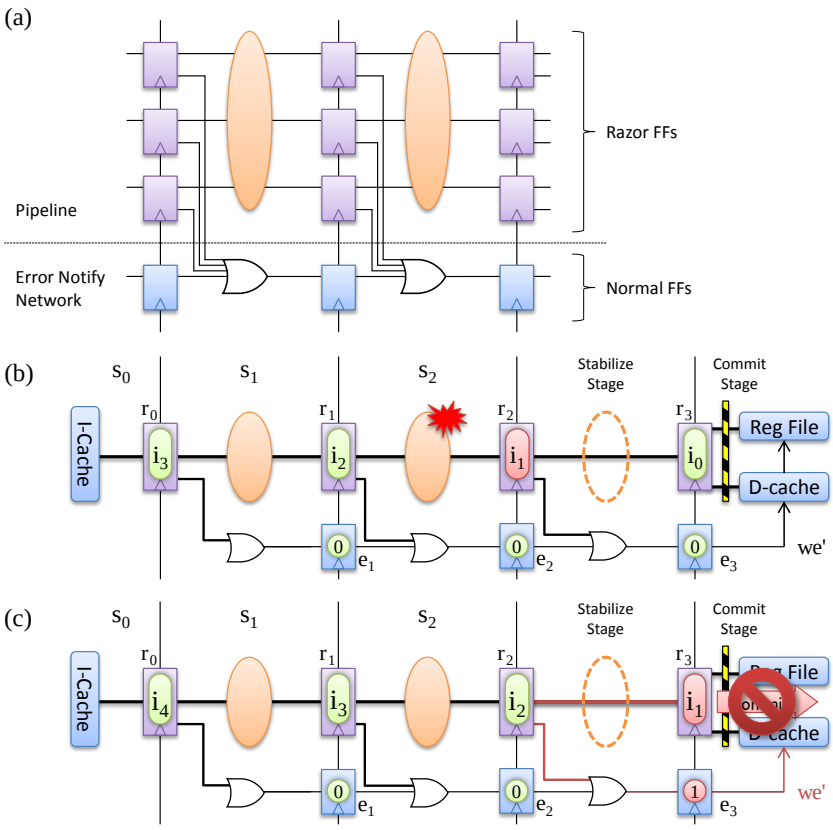

図 3.1: アーキテクチャ・レベルの回復技術のパイプライン

チャ・レベルの回復技術のパイプラインの概略を示す.

パイプライン・ラッチ（Pipeline Latch : **PL** ）（以下ではPL とする）は，Razor FF[19] で原則的には構成される[1].

エラー通知ネットワークは，各ステージにおいて発生したエラー信号を，パイプラインに沿って下流へと通知する．各ステージでは，Razor FF から出力されるエラー信号の OR をとり，次のステージへと出力する．出力されたこのエラー信号は，次のステージにおける OR の入力に含められる．

エラー通知ネットワークによってエラー信号がパイプラインの最下流にあるコミット・モジュールへと伝えられ，コミット・モジュールはコミットを停止する．その後，フォールトの影響の除去を行い，保護された AS から実行を再開する．

3.1.2 Razor Ⅱの回復技術

本節からは，Razor Ⅱの回復技術に特有な点についてまとめる．図 3.1 **(b)**, **(c)** に端的に表れているように，Razor Ⅱの回復技術は，スカラ・プロセッサ（パイプライン・マシン）を意識したものと言える．その特徴は以下の 2 点にまとめられる：

1. 各命令と，その命令が起こしたフォールトのエラー信号がパイプラインを「並んで」下っていく
2. 例外や投機ミスによるパイプライン・フラッシュによってフォールトの影響を取り除く

また，コミット直前のステージでフォールトが発生した場合にもコミットを停止するためには,「何もしない」ステージが必要となる．Razor Ⅱでは，このステージ s_3 を**スタビライズ・ステージ**と呼んでいる．スタビライズ・ステージにおいては命令は「何もしない」ので，このステージにおけるフォールトの発生確率は十分低いとみなすことができる.

以下，図 3.1 (b)，(c) を例に Razor Ⅱのパイプラインの動作を説明する：

図 3.1 (b) は，命令 i_1 がステージ s_2 を通過中にフォールトが発生した（図中，爆発のアイコン）場合を示す．同図は，そのサイクルの終わりを表している．サイクルの終わりに PL r_2 に格納された命令 i_1 はフォールトの影響を受けており，コミットしてはならない．この時点では，r_2 と同相のレジスタ e_2 はセットされていないことに注意されたい.

図 3.1 (c) は，その次のサイクルの終わりを表している．このサイクルの終わりには，i_1 はスタビライズ・ステージを通り，PL r_3 に格納される．エラー

[1]フォールト発生の可能性が低いものに関しては，通常の FF に置き換え可能である.

信号はこのサイクルにおいて OR されて，i_1 が格納されたレジスタ r_3 と同相のレジスタ e_3 がセットされる．

更にその次のサイクルには，e_3 によってライト・イネーブル we' が制御され，i_1 のレジスタ・ファイル/データ・キャッシュへの書き込みが抑制される，すなわち，i_1 のコミットを停止することができる．

このように，エラー信号はフォールトを起こした命令と「並んで」パイプラインを下っていくことになる．したがって，パイプラインを下っていく各命令に 1-bit のエラー・フラグを付加したものと考えてよい．

パイプライン・フラッシュ　Razor Ⅱ では，ゼロ除算などの例外からの回復と同様に，パイプライン・フラッシュによって回復するとしている．後に詳しく述べるが，このパイプライン・フラッシュが Razor Ⅱ の適用範囲をスカラ・プロセッサに限定する主な要因となる．

3.2　Out-of-Order プロセッサと Razor Ⅱ の限界

前節で述べた Razor Ⅱ は，検出技術には大きな問題はないが，回復技術に不十分な点がある．Razor Ⅱ の回復技術は，ごく単純なスカラ・プロセッサでは正しく動作するが，実用的なプロセッサには適用できない．

本章では，リオーダ・バッファ（ROB） とロード/ ストア・キュー（LSQ）を用いる方式の out-of-order スーパスカラ・プロセッサのコミットについて説明し，Razor Ⅱ の技術では限界があることを示す．以下，3.2.1 項でコミットについて詳しく述べ，3.2.2 項で Razor Ⅱ の問題点について述べる．

3.2.1　Out-of-Order プロセッサのコミット

3.2.1.1　コミットに関わるモジュール

図 3.2（上）に，ROB/LSQ を用いる out-of-order プロセッサ・モデルの概観を示す．コミットに関わるモジュールは 2 系統あり，同図中では上下に描かれている：

上　**ROB**：リオーダ・バッファ　　→
　　LRF：論理レジスタ・ファイル
下　**LSQ**：ロード/ ストア・キュー　→
　　L1D：1 次データ・キャッシュ

Out-of-order スーパスカラ・プロセッサでは，命令の実行は out-of-order に行われるが，その結果は in-order に実行した場合と同一でなければならない．ROB/LSQ の役割とは，端的に言えば，in-order な結果を保証しつつ out-of-order 実行を実現することである．具体的には，以下の 2 つにまとめられる：

コミット　コミット，すなわち，AS の不可逆的な更新は，in-order に行われなければならない．そのため ROB/LSQ は，基本的には FIFO で構成され，各エントリは in-order に割り当てられる．LRF/L1D の更新は，実行が終了した命令から順にではなく，各系統ごとに in-order に行われる．

フォワーディング　命令がコミットされ，実行結果が実際に LRF/L1D から読めるようになるまでの間，後続の命令には ROB/LSQ が依存元の命令の

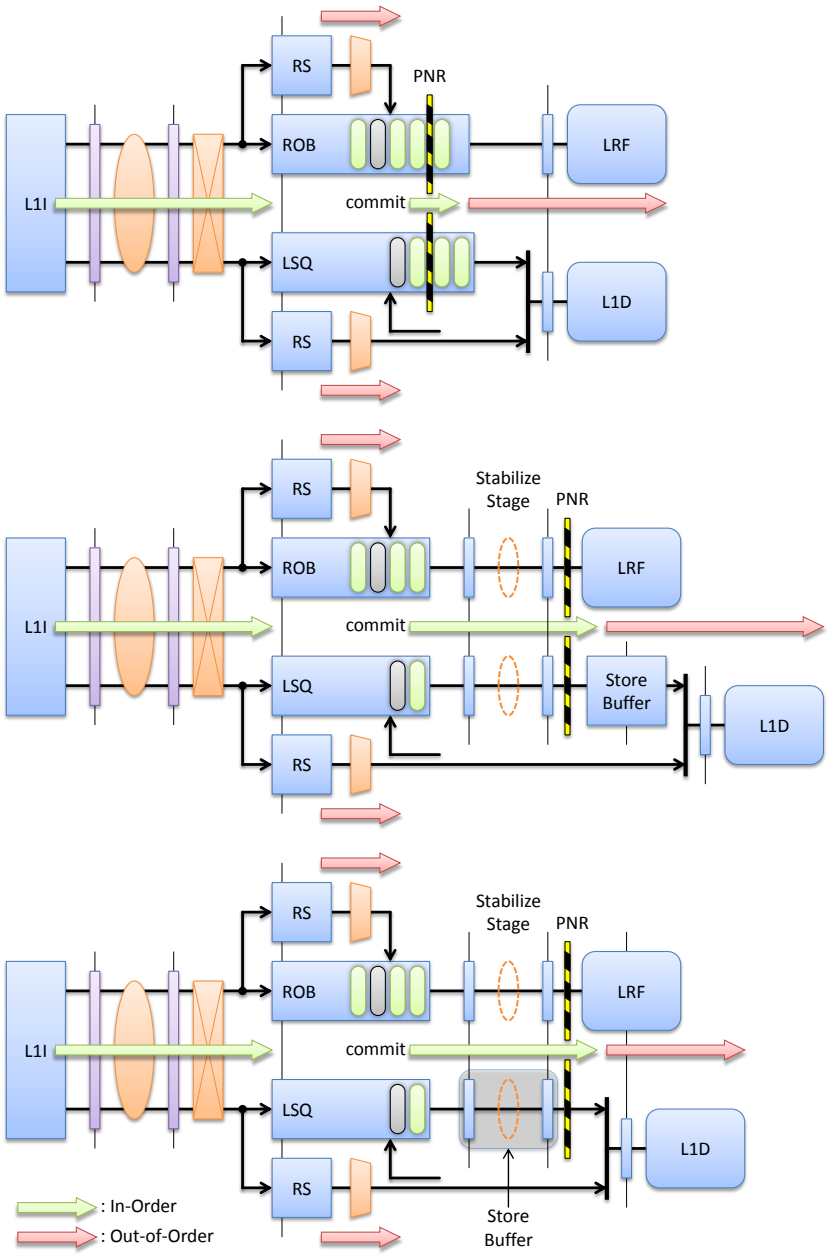

図 3.2: 前提モデル（上）と提案モデル（中，下）

結果を供給する．[2]

3.2.1.2 ROB/LSQ

図 3.3 に ROB と LSQ の典型的な構成例を示す [20].

この例では，ROB/LSQ はリング・バッファとして構成されている．**head/tail** ポインタは，通常のリング・バッファと同様，有効なエントリの先頭と末尾のエントリを指す．その他に，有効なエントリ数を数えるカウンタ **count** がある．

commit ポインタ　ROB/LSQ に特徴的なのは，**commit** ポインタである．commit ポインタは，次にコミットの対象となるエントリを指す．毎サイクル，commit ポインタの指すエントリから下流に向かって命令を探し，終了していない命令が見つかったら，そのエントリへと commit ポインタを進める．コミット幅以内に終了していない命令がない場合には，コミット幅だけ進められる．commit ポインタの更新により head〜commit の範囲に入った命令は，通常，次のサイクルに読み出され，LRF に送られる．

同図中，**PNR** (Point-of-No-Return) は，commit ポインタの指す命令の直前に位置している．すなわち，commit ポインタが更新されて PNR を超えた命令は，二度と超える前の状態には戻すことができず，必ず LRF を更新しなければならない．

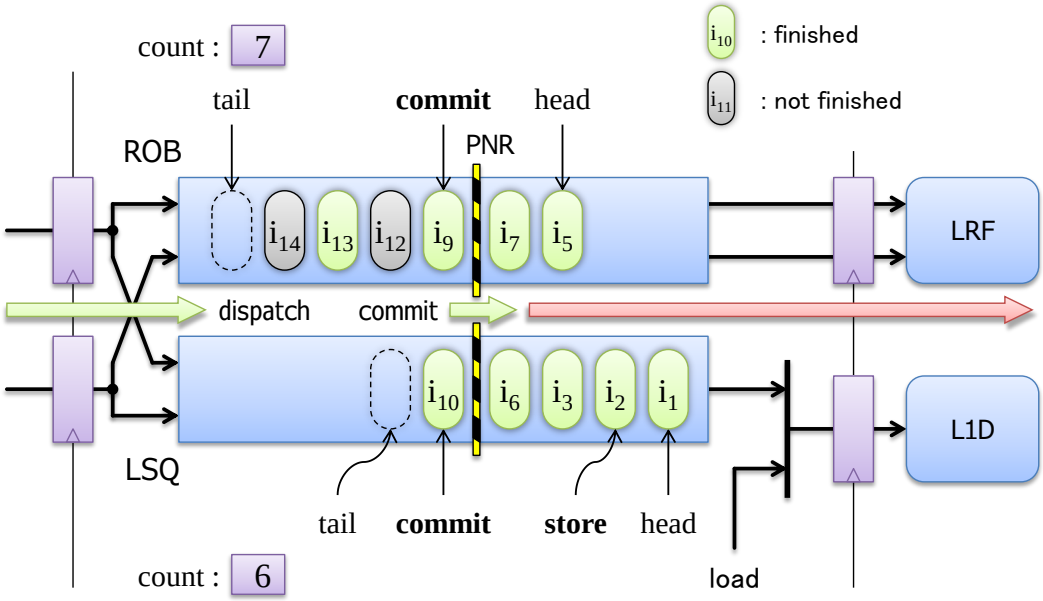

図 3.3: ROB と LSQ

[2]ROB に関しては，フューチャ・ファイルなど，別のモジュールがこの役割を担うことが多い．

LSQ のストア・バッファ　LSQ の場合，head〜commit にあるストア命令はただちに L1D への書き込みを開始できるわけではない．L1D のポートは，通常，ロード命令が優先して使用するため，ストア命令による L1D への書き込みは**サイクル・スチール**によって行われる．すなわち，コミットされたストア命令の読み出しは，L1D のポートの空きを待たなければならない．

図 3.3 の構成では，LSQ の head〜commit の部分が，いわゆる**ストア・バッファ**として，以下の 2 つの役割を果たすことになる：

バッファリング　L1D のポートの不足により LSQ からの読み出しが停止している間も，commit ポインタの更新を続けることができる．

フォワーディング　前述した ROB/LSQ のフォワーディング機能と原則変わりない．

LSQ には head〜commit のエントリのうち，どこまで L1D へ送り出し，次にどこから送り出すかを管理する store ポインタが存在する．

3.2.2　Razor II の回復技術の問題点

ROB → LRF 系と LSQ から L1D 系，それぞれの系統を見れば，命令のデキューは in-order に行われる．しかし全体を一系統とすると，LSQ のストア・バッファにより，ROB/LSQ からデキューされる命令は out-of-order である．このことより，out-of-order プロセッサにおいて，in-order な AS を補償しているのは ROB/LSQ のポインタのみであると言える．そのため，ROB/LSQ のポインタにフォールトが発生した場合，正しい AS が分からなくなり，致命的である．

Out-of-order プロセッサに Razor Ⅱ の回復技術を適用しようとすると様々な問題が生じるが，最も致命的な問題は，この ROB/LSQ のポインタに代表される制御系のフォールトに対処できない点である．その理由は，以下の 2 点により説明できる．

まず，ROB/LSQ にはスタビライズ・ステージを配置することができない．commit ポインタでフォールトが生じた場合，その瞬間に AS は正しくなくなる．その後，コミットを停止，すなわち，commit ポインタの更新を停止しても手遅れである．

次に，パイプライン・フラッシュによる回復では，ポインタのフォールトの影響を除去できない．ROB/LSQ におけるフラッシュのロジックは，tail ＝ commit と tail ポインタを commit ポインタの内容で上書きすると同時に，count を tail − commit の分だけ減らす．そのため，commit ポインタがフォールトによって誤った値となっていた場合，フォールトの影響はむしろ tail ポインタへと拡大するだけで，フォールトの影響を取り除くことはできない．

結局，Razor II の回復技術は，少なくともそのままでは，文献 [18] で例示されているようなごく単純なプロセッサにしか適用できないのである．

3.3 Timing-Fault-Tolerant OoO Processor

前章では，Razor Ⅱ の回復技術は実用的なプロセッサに適用できないことを述べた．我々は以前，前節までの問題を解決し，複雑な out-of-order スーパスカラ・プロセッサにも適用できる技術を提案している [8, 9]．Razor Ⅱ の回復技術との違いを際立たせるこの提案の特徴は，3 つの “I” によって表される．本節では，以前までの提案の概要をまとめる．

3.3.1 In-Order Passing through PNR

図 3.2（中）は提案手法のプロセッサ・モデルである．3.2.1.2 節で述べた LSQ のストア・バッファを別体化し，ストア・バッファのエンキューの直前を PNR とすることで，すべての命令が ROB/LSQ の外部にある PNR を in-order で通過することを保証する．スタビライズ・ステージにより，ROB/LSQ のポインタにフォールトが発生した場合にも，LRF/L1D への書き込みを停止することが可能となる．

また，別体化されたストア・バッファにはバッファリングの機能のみを担い，フォワーディングは LSQ 自体が行う．バッファリングの機能は十分に単純であるため，ストア・バッファは fault-free とすることができる．

ストア・バッファが fault-free であれば，図 3.2（下）のような，シフト・レジスタをベースとする構成を取れ，ストア・バッファ内の最初のステージをスタビライズ・ステージとすることができる．この場合の最小エントリ数は 2 となる．後の評価でもこのモデルを用いる．

なお，ストア・バッファを別体化しても，IPC の低下は平均で 0.7% と十分に低いことが示されている．[8, 9]

3.3.2 Imprecise Cancellation

3.1.2 節で述べたように，Razor Ⅱ はフォールトを起こした命令を特定する．ところが，3.2.1.2 項で述べたバッファのポインタの例などを考えれば，out-of-order スーパスカラ・プロセッサではフォールトの影響を受けた命令を正確に特定することは原理的に不可能である．

実際には，正確に特定する必要などそもそもない．AS を保護するための条件は，フォールトの影響を受けた可能性のある命令がエラー信号より先に PNR に到達しないことのみである．

実際 out-of-order スーパスカラ・プロセッサの場合，フォールトの影響を受けた命令よりも，エラー信号の方が先に PNR に到達することになる．これは，エラー通知ネットワークが単純なパイプラインで構成されるのに対し，命令は命令ウィンドウでバッファリングされるからである．

また，コミットの停止と命令のキャンセルは，Razor Ⅱ の回復技術のようにフォールトを受けた命令の到着を待って行う必要はない．フォールトの発生が判明したら，PNR を超えていない命令は，フォールトの影響を受けたか受けないかに関わらず，速やかにキャンセルしてしまってよい．

3.3.3 Initialization of Pipeline above PNR

フラッシュではなく，PNR より上流を初期化することによってフォールトの影響を取り除く．初期状態のパイプラインが，LRF/L1D に保存された in-order な AS から実行を再開することになる．

3.2.2 節で述べた ROB/LSQ のフラッシュ・ロジックと異なり，初期化のロジックは，すべてのポインタ，カウンタの値を 0 にすることである．当然ながら，フォールトの影響は完全に除去できる．

Fan-out の低減のため，パイプラインの上流から順に初期化を行うネットワークを構築する．この構成については後に詳しく述べる．

3.4 提案手法の検出/回復方式

前章で述べた回復手法は，Imprecise Cancellation と Initialization of Pipeline above PNR に関して，十分に議論されていなかった．本節ではまず，これまでの提案の検出/回復方式の問題点を説明し，その解決方法について説明した後，実際の動作例やペナルティについて述べる．

3.4.1 初期化方式の構成

以前までの提案では，フォールトの除去のために，PL の上流から順に初期化を掛ける方法を用いるとしている．本節では，この初期化方式の構成について述べる．

提案手法の初期化方式を実現するためには，エラー通知ネットワークと同様に，**初期化ネットワーク** をパイプラインと並べて設けると都合がよい．このネットワーク上を初期化信号が伝わっていく．

提案手法の初期化の振る舞いは，1 ステージごとの同期リセットと言える．初期化信号がサイクルに同期しているため，端子は非同期リセット端子を用いて構わない．

ここでの初期化の対象は，データ・パス上のレジスタではなく，制御系のレジスタであることに注意されたい．制御系のレジスタの初期化は，ポインタ・カウンタの値であれば 0 にすることであり，Valid ビットであればそれを落とすことである．

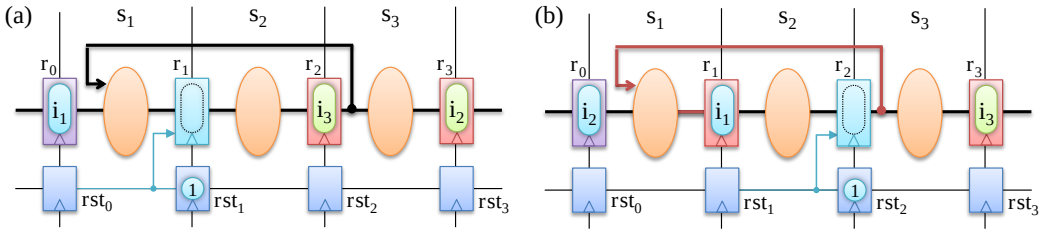

図 3.4: バック・エッジによる問題

実行の再開は，全ての PL が初期化されるのを待ってから行う必要はない．最上流の PL が初期化されたら直ちに命令の再フェッチを行ってよく，その分回復のペナルティを短縮することが可能である．このために満たすべき制約は後に詳しく述べる．

3.4.2 バック・エッジへの対策

パイプラインの上流から順に初期化を行い，全ての PL の初期化を待たずに命令の実行を再開する提案手法の回復技術では，バック・エッジによって初期化がうまくかからない場合がある．**図 3.4** において，ステージ s_3 からステージ s_1 にバック・エッジが存在するモデルを仮定し，このことを示す．

図 3.4(a) は，rst_0 にセットされた初期化信号により，PL r_1 が初期化された状態を表している．同図はこのサイクルの終わりを表している．このサイクルの終わりには，初期化信号が rst_1 にセットされ，PL r_0 には再フェッチされた命令 i_1 が格納されている．再フェッチされた命令は水色で示す．また，PL r_2，r_3 は初期化が行われておらず，フォールトの影響が残っている．フォールトの影響が及んだ PL を赤色で示す．

図 3.4(b) は，この次のサイクルの終わりを表している．このサイクルの終わりには，rst_1 にセットされた初期化信号により，PL r_2 が初期化される．ところが，r_2 が初期化される前に，前のサイクルにおける r_2 のフォールトの影響が，バック・エッジにより r_1 に及んでしまう．再フェッチされた命令 i_1 にフォールトの影響が及び，正しく初期化が行えない．

この問題は，再フェッチされた命令がバック・エッジの出先のレジスタ r_2 に格納されるまで，バック・エッジをアサートしないことで回避できる．具体的な対策としては，カウンタを設け，バック・エッジ長に相当するステージ数だけ，バックエッジを無効にすると言った方法が考えられる．

3.4.3 回復技術

本節では，初期化方式により正しく回復するための制約を挙げた後，以前までの提案と合わせた検出/回復の動作を説明する．

3.4.3.1 回復における制約

3.3.2 節で，AS を保護するための条件を述べた．初期化における，正しい AS からの実行を再開するための条件も同様に，再フェッチされた命令よりも先に初期化信号が PNR に到達することである．この条件は，命令が再フェッチされる前に，パイプラインの最上流の PL の初期化を行うよう設計することで満たされる．

また，Imprecise Cancellation によりフォールトを起こした命令を特定しないため，パイプラインよりも少ない段数でエラー通知ネットワークや初期化ネットワークを構築できる．スキップした PL の数と同じだけ，エラー信号や初期化信号が先に PNR に到達できる．

以上のような点から，フォールトの影響が PNR を超える前に，コミットを停止することで AS を保護し，再フェッチされた命令が PNR に到着する前に，初期化によりフォールトの影響を全て取り除き，AS の更新を再開することが可能となる．

3.4.3.2 回復の流れ

図 3.5 は以前までの提案と今回提案した初期化方式の構成を合わせた回復技術を示したものである．PL r_3 がバッファb_3 に置換された Out-of-Order プロセッサ・モデルを仮定する．図 3.5 中，pc_{fetch} は fetchPC を，pc_{commit} は PNR を通過した命令の nextPC を示す．

エラー通知ネットワークと初期化ネットワークをパイプラインと並べて設ける．スキップしたレジスタは点線で示す．

以下，図 3.5 (1)-(8) を例に，提案手法のパイプラインの動作を説明する．図の番号はサイクル数を表している：

図 3.5(1)　は，ステージ s_1 においてフォールトが発生した（図中，爆発のアイコン）場合を示す．同図は，そのサイクルの終わりを表している．サイクルの終わりに PL r_1 に格納された命令 i_4 はフォールトの影響を受けており，コミットしてはならない．この時点では，Razor Ⅱ 同様，b_1 と同相のレジスタ e_1 はセットされていない．

図 3.5(2)：　このサイクルの終わりには，命令 i_0 が LRF/L1D に書き込まれ，pc_{commit} が 1 に更新される．バッファb_3 に格納された命令 i_3 は実行が完了しておらず，次のサイクルではデキューすることができない．命令 i_4 は PL r_2 に格

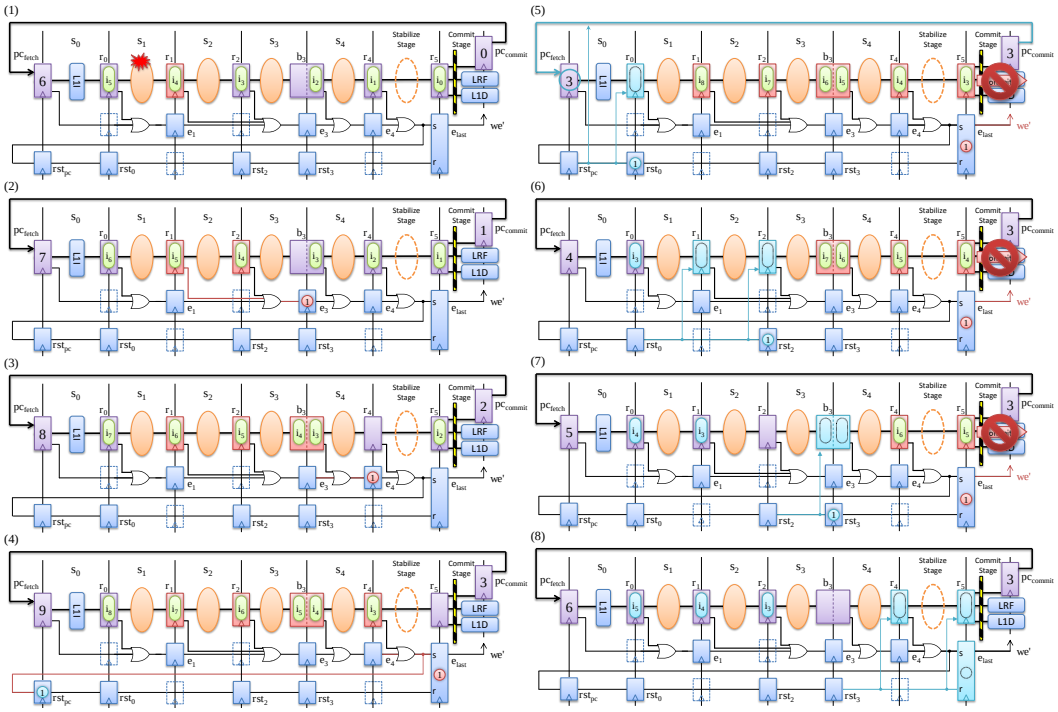

図 3.5: 提案手法の回復技術

納され，フォールトの影響が r_2 まで及ぶ．エラー信号はこのサイクルにおいて OR されて，レジスタ r_3 と同相のレジスタ e_3 にセットされる．

図 3.5(3): このサイクルの終わりには，i_i が LRF/L1D に書き込まれ，pc_{commit} が 2 に更新される．バッファ b_3 に格納された命令 i_3 の実行が完了する．命令 i_4 はバッファ b_3 に格納され，フォールトの影響が b_3 まで及ぶ．エラー信号は，レジスタ r_4 と同相のレジスタ e_4 にセットされる．

図 3.5(4)： このサイクルの終わりには，命令 i_2 が LRF/L1D に書き込まれ，pc_{commit} が 3 に更新される．バッファ b_3 から命令 i_3 がデキューされ，PL r_4 に格納される．この時，命令 i_4 は未だバッファ b_3 に格納されたままである．フォールトの影響は b_4 まで及ぶ．その一方で，エラー信号は PNR 直前のレジスタ e_{last} にセットされる．

また，これと同時に，初期化用ネットワークの最上流のレジスタ rst_{pc} がセットされる．これらは次のサイクルにおける回復処理のトリガとなる．

図 3.5(5)： このサイクルの終わりには，フォールトの影響は PL r_4 に及ぶ．前のサイクルで e_{last} にセットされたエラー信号によって，ライト・イネーブル we' が制御され，LRF/L1D と pc_{commit} への書き込みが抑制される．e_{last} のエラー信号は，全ての PL の初期化が終わるまでセットされたままである．

また，rst_{pc} によって，PL r_0 が初期化され，pc_{fetch} の値が，pc_{commit} の値に書き換えられる．

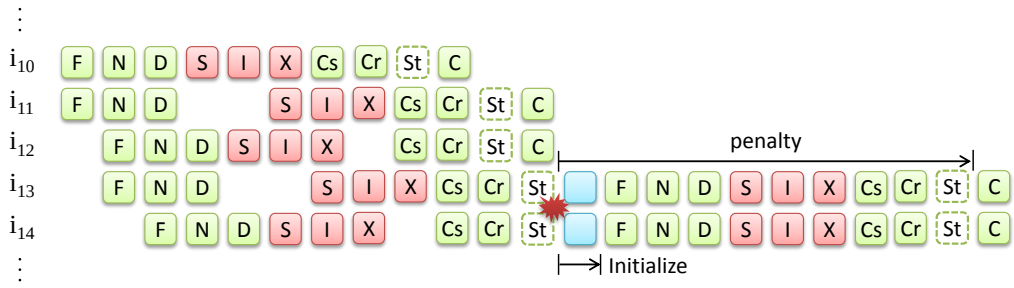

図 3.6: 回復のペナルティ

図 3.5(6)： このサイクルの終わりには，rst_2 がセットされ，PL r_1, r_2 が初期化される．また，再フェッチされた命令 i_3 が PL r_0 に格納される．

図 3.5(7)： このサイクルの終わりには，rst_2 がセットされ，バッファ b_3 が初期化される．再フェッチされた命令 i_3 は，PL r_1 に格納される．

図 3.5(8)： このサイクルの終わりには，rst_5 がセットされ，PL r_4, r_5 が初期化される．これにより，PNR 上流の PL の初期化が全て終了したこととなる．これと同時に，e_{last} の初期化も行う．再フェッチされた命令 i_3 は，PL r_2 に格納される．

この次のサイクルには，e_{last} によってライト・イネーブル we' が制御され，i_1 の LRF/L1D への書き込みが開始され，コミットが再開される．

このように，エラー信号や初期化の信号は，必ずフォールトの影響を受けた命令や再フェッチされた命令よりも先に PNR 付近のレジスタに到達することができ，フォールトから正しく回復することが可能となる．

3.4.4 回復のペナルティ

図 3.6 に示すように，提案手法の回復のペナルティ P は，パイプライン段数を P_0，フォールト検出から再フェッチを行うまでにかかるレイテンシを L_{init} とすると，$P = P_0 + L_{init}$ で表される．全ての PL の初期化を待ってから再フェッチをかけるモデルでは，$L_{init} = P_0$ となるが，3.4.3.2 項で述べた最上流の PL の初期化後，直ちに再フェッチを行うモデルにおいては，$L_{init} = 1$ である．そのため，ペナルティは僅かに増加するのみであり，全ての PL の初期化を待つ方式に比べ，回復のペナルティが 1/2 程度短縮される．この回復方式のペナルティの違いによる IPC への影響は次章で評価する．

ステージ数とペナルティの関係　また，どのステージでフォールトが起きたかによって，回復のペナルティは変動しない．これは，エラー通知ネットワークの末端のレジスタ e_{last} の値が変化する確率が，チップ全体の TF 発生率と一致するからである．以下，簡単な式を元にこのことを示す．

チップ全体の TF 発生率を α，ステージ数を n とする．各ステージにおいて一様に TF が発生すると仮定すると，各ステージごとの TF 発生率は α/n となる．この時，e_{last} にエラー信号がセットされる確率 α_{last} は，どのステージでも TF が起きなかった場合以外を考慮するため，$\alpha_{last} = 1 - (1 - \alpha/n)^n$ で表される．一般に $\alpha/n \ll 1$ であるから，$\alpha_l \simeq p$ となる．よって，α_{last} がチップ全体の TF 発生率と一致する．

このことより，ステージ数によって，α_l は変化せず，さらに，$\alpha_{last} \simeq p$ であることから，e_{last} の値の変化のみを見れば，チップ全体の TF の発生を包括的に見ることがわかる．そのため，どのステージで TF が起きたかによって回復のペナルティが変動することはないことが示される．

回復ペナルティによる IPC への影響　次章の評価では，フォールトの発生率を変化させ，回復のペナルティによってどれほど IPC が低下するかを評価する．ここでは，簡単な式を元に IPC の低下率の理論値を概算する．

フォールトが発生しなかった場合の IPC，実行サイクル数，実行命令数をそれぞれ i_b，c_b，N，フォールトの発生率を α，フォールトによる回復のペナルティを P とする．この時，フォールト発生時の IPC i_f は，以下の**式 3.1** で表される：

$$i_f = \frac{N}{c_f} = \frac{c_b i_b}{c_b + c_b \alpha P} = \frac{1}{1 + \alpha P} i_b \tag{3.1}$$

これにより，フォールト発生に伴う IPC の低下率は $(1 + \alpha P)^{-1}$ で表される．

この式によって求められる低下率の理論値は，あくまでも次章の評価において指標として用いるものであり，実際には様々な要素によって IPC は変化する．これについては次章で詳しく述べる．

3.5　評価

本章では，フォールトの回復のペナルティによる IPC への影響について，シミュレーションにより評価した．

3.5.1　評価モデルと評価環境

プロセッサ・シミュレータ**鬼斬弐** [21] によって，SPEC CPU2006 [22] を用いて評価を行った．**表 3.1** に，ストア・バッファ以外のプロセッサの構成をまとめる．LSQ は，図 3.2（下）のストア・バッファがシフト・レジスタで構成されたモデルを用いる．シフト・レジスタのエントリ数は最小サイズの 2 に固定している．

3.4.4 節で述べたように，e_{last} の変化率をチップ全体のフォールトの発生率とする．

フォールトの検出から再フェッチまでのレイテンシ L_{init} が異なる 2 種の初期化モデルを評価する：

BASE フォールトの発生確率を 0 とした場合

WAIT 全ての PL の初期化を待ってから再フェッチを掛けるモデル．3.4.4 節で述べたように，L_{init} はパイプライン段数 P_0 と同値である．ここでは $L_{init} = P_0 = 20$ とする．

IMMEDIATE 提案の回復方式．最上流の PL の初期化後，直ちに再フェッチを行うモデル．$L_{init} = 1$.

3.5.2 評価結果

図 3.7 に，フォールト発生率毎の BASE に対する IMMEDIATE の相対 IPC を示す．発生率は 0.001% ,0.01% ,0.1% ,1% の 4 つのパラメータを取る．図 3.7 上の赤太線は，式 3.1 で求められる低下率の理論値である．

理論値よりも相対 IPC があまり低下していない milc, libquantum では，L2 キャッシュ・ミスの回数が多い．ロード命令がキャッシュ・ミスを起こし，リクエストを行っている間にフォールトが検出されると，パイプラインのロード命令は除去されるが，リクエストは送信されたままである．そのため，命令の再実行時には早く値を読み込めるようになる．フォールトの回復ペナルティが L2 キャッシュ・ミスのペナルティによって隠蔽されるものと推定される．

表 3.1: プロセッサの構成

パラメタ	値
ISA	Alpha 21164A
fetch width	4 inst.
commit width	6 inst.
exec units	int : 2, fp : 2, mem : 2
inst window	int : 32, fp : 16, mem : 16
load/store queue	48/48 entries
load/store ports	1-read + 1-read/write, cycle stealing
branch pred	8KB g-share
miss penalty	10 cycles
BTB	2K-entry, 4-way
L1D	32KB, 4-way, 64B/line, 3 cycles
L2C	4MB, 8-way, 64B/line, 15 cycles
main memory	200 cycles

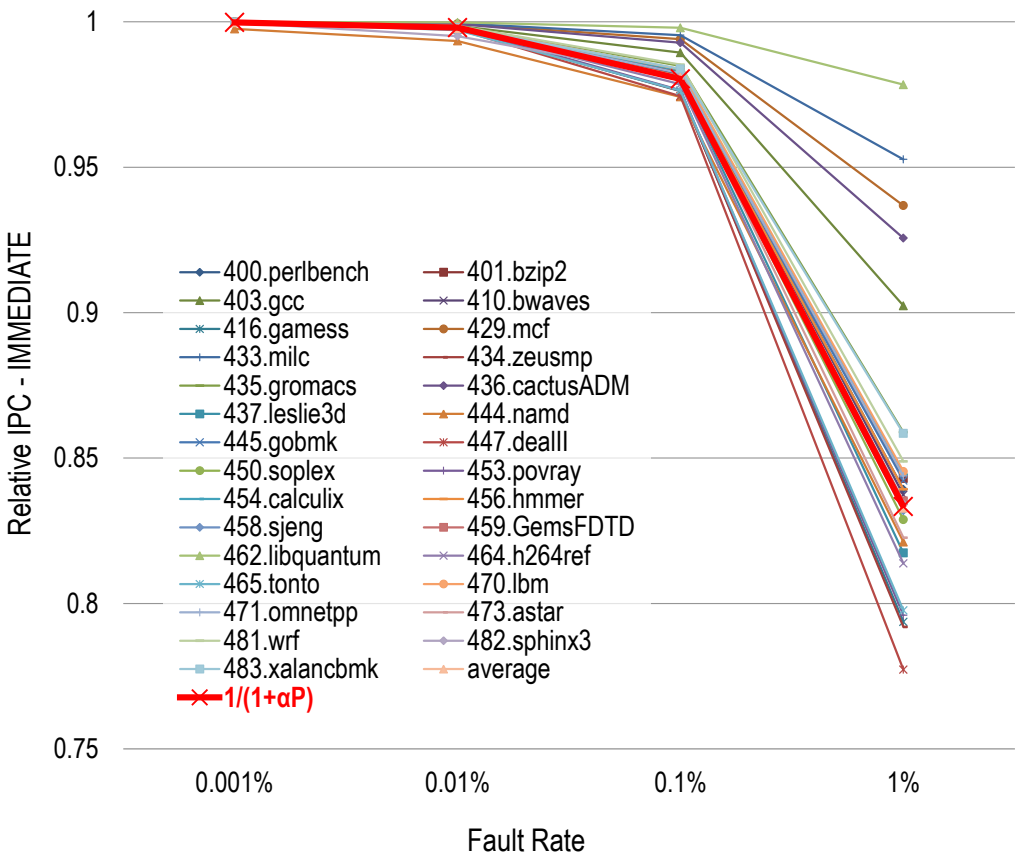

図 3.7: フォールト発生率と BASE に対する相対 IPC の関係

回復形式によるフォールト耐性　**図 3.8** は，フォールト発生率を 0.1% ～ 1.0% まで，0.1% 刻みで変化させた時の BASE に対する IMMEDIATE と WAIT の相対 IPC である．図中赤線は，全ベンチマークの平均の相対 IPC を表している．

この平均の相対 IPC の低下率が 10% となるフォールト発生率は，WAIT の場合，$\alpha = 0.3\%$ 程度だが，IMMEDIATE の場合は，$\alpha = 0.6\%$ 程度と 2 倍まで許容できる．これは，3.4.4 節で述べた，IMMEDIATE の回復ペナルティが WAIT の 1/2 程度となるためである．このことより，全ての PL の初期化を待つ方式よりも，提案の初期化方式の方がフォールト耐性があることが示された．

3.6　本章のまとめ

我々は，これまでに複雑な out-of-order スーパスカラ・プロセッサであってもフォールトから正しく回復できる方式を提案してきたが，検出/回復方式が考慮されていなかった．これに対し，本章ではこの検出/回復方式について，物理的な構成や具体的な実装方法，回復のペナルティなどの詳細について検討し

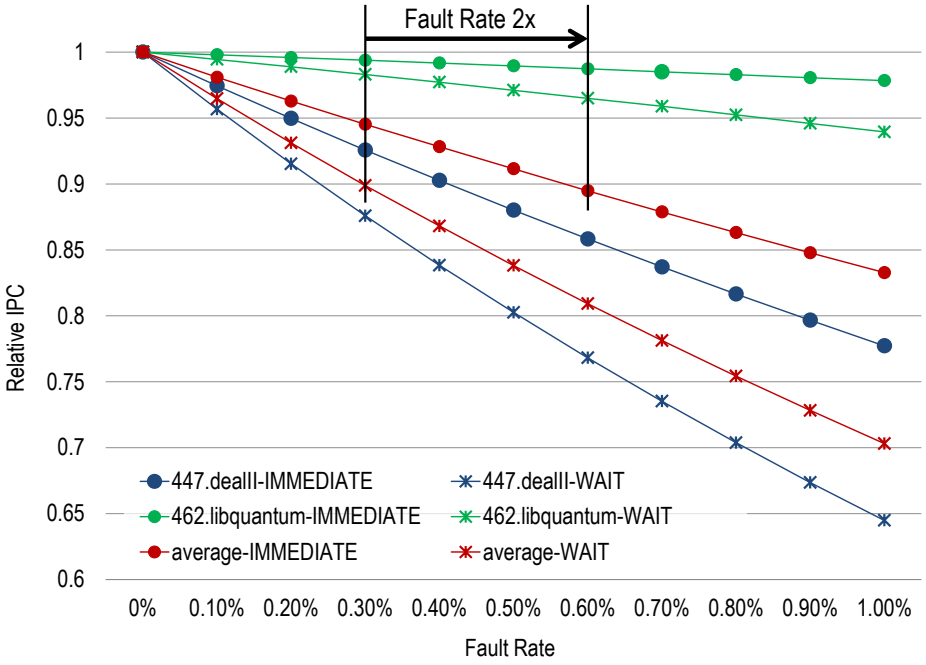

図 3.8: 回復形式によるフォールト耐性の違い

評価を行った．

また，提案の回復方式は，構成の仕方によって回復のペナルティが大きく変化する．シミュレーションによって，この回復のペナルティが IPC に与える影響を評価した．提案の方式では，最上流の PL が初期化された直後に命令を再フェッチすることで，回復のペナルティを短縮できる．シミュレーションによって，フォールト回復のペナルティによる相対 IPC の影響を評価し，IPC の低下率がフォールト以外の要因によって変動することや，全ての PL の初期化を待つ方式よりも，提案の初期化方式の方がフォールト耐性があることを示した．

我々は現在，NORCS[23] 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・プロセッサの開発を行っており，今後はこのプロセッサに提案手法を適用し，より詳細な評価を行う予定である．

第4章　おわりに

タイミング・フォールト検出/回復技術においては，回路レベルの検出技術とアーキテクチャ・レベルの回復技術の 2 つについて，新たな提案を行った．得られた主要な成果は以下の通りである：

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　提案するクロッキング方式は，二相ラッチ方式と Razor を組み合わせることで実現される方式である．従来の二相ラッチ方式で用いられていなかった，ラッチの開いている区間を利用することで，全てのステージの遅延を平均化することが可能となり，実効遅延の平均に基づいた動作を実現できる．

また，簡単な実験を元に，既存の手法である Razor と比べ，実際にサイクル・タイムを短縮できることを示した．評価では簡単な回路に本手法を適用し，FPGA 上に実装することで，最高動作周波数が 2 倍にまで向上することを確認した．

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
我々は，これまでに ROB や LSQ の内部に生じる，制御系のフォールトについて対処することで，複雑な out-of-order スーパスカラ・プロセッサであってもフォールトから正しく回復できる方式を提案してきたが，検出/回復方式が考慮されていなかった．これに対し，本稿ではこの検出/回復方式について，物理的な構成や具体的な実装方法，回復のペナルティなどの詳細について検討し評価を行った．

また，提案の回復方式は，構成の仕方によって回復のペナルティが大きく変化する．シミュレーションによって，この回復のペナルティが IPC に与える影響を評価した．提案の方式では，最上流の PL が初期化された直後に命令を再フェッチすることで，回復のペナルティを短縮できる．シミュレーションによって，フォールト回復のペナルティによる相対 IPC の影響を評価し，IPC の低下率がフォールト以外の要因によって変動することや，全ての PL の初期化を待つ方式よりも，提案の初期化方式の方がフォールト耐性があることを示した．

我々は現在，NORCS[23] 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・プロセッサの開発を行っており，今後はこのプロセッサにこれら 2 つの提案技術を適用し，より詳細な評価を行う予定である．

参考文献

[1] 平本俊郎, 竹内潔, 西田彰男: MOS トランジスタのスケーリングに伴う特性ばらつき, 電子情報通信学会誌, Vol. 92, No. 6 (2009).

[2] Ashish, S., Dennis, S. and David, B.: Statistical Analysis and Optimization for VLSI: Timing and Power, *ISBN: 978-0-387-25738-9* (2005).

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

[4] Blaauw, D., Kalaiselvan, S., Lai, K., Ma, W.-H., Pant, S., Tokunaga, C., Das, S. and Bull, D.: Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance, *Int'l Symp. on Solid-State Circuits Conference (ISSCC)* (2008).

[5] 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, *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 284 –285 (2010).

[6] 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).

[7] 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, 先進的計算基盤シンポジウム SACSIS, pp. 347–354 (2010).

[8] 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).

[9] 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 (2013).

[10] Bakoglu, H.: *Circuits, interconnections, and packaging for VLSI*, VLSI systems series, Addison-Wesley Pub. Co. (1990).

[11] Harris, D.: Skew-tolerant Circuit Design, *Morgan Kaufmann Publishers*, pp. 12–14 (2001).

[12] Xilinx Inc.: Virtex-6 ライブラリ ガイド（HDL 用）, pp. 190–193 (2009).

[13] Tiwari, A., Sarangi, S. R. and Torrellas, J.: ReCycle: Pipeline Adaptation to Tolerate Process Variation, *ISCA*, pp. 323–334 (2007).

[14] Choudhury, M., Chandra, V., Mohanram, K. and Aitken, R.: TIMBER: Time Borrowing and Error Relaying for Online Timing Error Resilience, *Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1554–1559 (2010).

[15] 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, *ISSCC*, pp. 488–490 (2012).

[16] Bernstein, K., Carrig, K. M., Durham, C. M., Hansen, P. R., Hogenmiller, D., Nowak, E. J. and Rohrer, N. J.: *High speed CMOS design styles*, Kluwer Academic Publishers, Norwell, MA, USA (1998).

[17] Shioya, R., Horio, K., Goshima, M. and Sakai, S.: Register Cache System not for Latency Reduction Purpose, *Int'l Symp. on Microarchitecture (MICRO-43)*, pp. 301–312 (2010).

[18] Blaauw, D., Kalaiselvan, S., Lai, K., Ma, W.-H., Pant, S., Tokunaga, C., Das, S. and Bull, D.: Razor II: In-Situ Error Detection and Correction for PVT and SER Tolerance, *Int'l Solid-State Circuits Conference (ISSCC)* (2008).

[19] Ernst, D., Kim, N., Das, S., Pant, S., Pham, T., Rao, R., Ziesler, C., Blaauw, D., Austin, T. and Mudge, T.: Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation, *Int'l Symp. on Microarchitecture* (2003).

[20] Shen, J.: *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill Science/Engineering/Math (2004).

[21] 塩谷亮太, 五島正裕, 坂井修一: プロセッサ・シミュレータ「鬼斬弐」の設計と実装, 先進的計算基盤システムシンポジウム SACSIS, pp. 120–121 (2009). （ポスター）.

[22] The Standard Performance Evaluation Corporation: SPEC CPU2006 suite, http://www.spec.org/cpu2006/.

[23] Shioya, R., Horio, K., Goshima, M. and Sakai, S.: Register Cache System not for Latency Reduction Purpose, *Int'l Symp. on Microarchitecture (MICRO-43)*, pp. 301–312 (2010).

研究業績

雑誌論文

1. 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 (2013).

国内会議（査読付き）

1. 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」,『先進的計算基盤システムシンポジウム SACSIS2012』, Vol. 2012, pp. 382-389 (2012).

2. 吉田 宗史，広畑 壮一郎, 倉田 成己, 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 SACSIS2013』,（投稿中）

口頭発表（査読なし）

1. 吉田 宗史, 有馬 慧, 岡田 崇志, 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 第 73 回全国大会』, pp. 1-91-1-92 (2011).

2. 吉田 宗史, 有馬 慧, 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の予備実験」,『電子情報通信学会技術報告 CPSY2011-7（SWoPP2011）』, pp. 13-18 (2011).

3. 広畑 壮一郎, 吉田 宗史, 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 CLA への適用」,『情報処理学会 第 74 回全国大会』, pp. 1-63-1-64 (2012).

4. 堀口 達也, 吉田 宗史, 倉田 成己, 五島 正裕, 坂井 修一,「耐故障 FPGA アーキテクチャ」,『情報処理学会研究報告 2012-ARC-201（SWoPP2012）』, No. 5, pp. 1-8 (2012).

5. 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 2012-ARC-201（SWoPP2012）』, No. 20, pp. 1-8 (2012).

ポスター論文（査読なし）

1. 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」,『先進的計算基盤システムシンポジウム SACSIS2012』,

受賞

1. 吉田 宗史, 「情報処理学会 第 73 回全国大会, 情報処理学会学生奨励賞」(2011).

謝辞

非常に多くの方々から多大なご指導、ご協力、励ましを頂き、本論文を完成させることができました。

坂井修一教授にはご多忙な中，相談会等で有益かつ鋭いご指摘をいただきました.

五島正裕准教授には，研究内容をテーマの決定から本論文に至るまでご助言をいただき，またあらゆる面で相談にのっていただき指導してくださいました.

倉田成己氏，塩谷亮太氏には，提案手法のシミュレータへの実装，本論文の添削に至るまで様々な面で教わり，サポートしていただきました.

八木原晴水さん，長谷部環さんには，研究室における設備の導入や各種事務手続き・おいしいお菓子など，研究室で過ごすための様々なご支援を頂きました.

その他，CREST メンバーの先生方，研究室メンバーの同期，後輩，冷蔵庫，ソファーベット，菊水湯のおばちゃん，梅干をサービスしてくれる食堂もり川の店員の皆様，いわばヒモである私を住居や飲食，金銭などの面で支えてくれた彼女にも大変助けられました．この場を借りて、感謝の意を表したいと思います.