Vol.2019-ARC-235 No.24
Vol.2019-SLDM-187 No.24
Vol.2019-EMB-50 No.24
2019/3/17

タイミング・フォールト検出回復手法の Rocket に対する適用

神保 潮[1,a)]　塩谷 亮太[2,b)]　五島 正裕[3,c)]

概要：Razor はタイミング障害の検出/回復のための代表的な手法である．本稿では，Razor を RISC-V コアの Rocket に適用する．Rocket は一種のアウトオブオーダースカラプロセッサであり，このことが Razor への適用において問題である．その問題点を明確にし，解決方法を示す．また，Razor 適用による性能の変化について評価する．

キーワード：ばらつき，タイミング故障検出，タイム・ボローイング，Razor FF，Rocket

1. はじめに

チップ内のランダムな**ばらつき**の増大 [1] により，従来のワースト・ケースに基づいた設計ではチップの性能の向上が見込めなくなりつつある．微細化により，遅延の典型値は短縮されている一方で，ばらつきの増大によって分散は大きくなっている．そのため，歩留まりを一定とすると，ワースト遅延は，ティピカル遅延ほどには短縮されなくなる．こうした傾向が続けば，微細化が進むにつれてティピカル遅延とワースト遅延の差は広がっていき，将来的には，ワースト遅延が短縮されなくなってしまうことも考えられる．

そのため，ワースト・ケースより実際に近い遅延 (実効遅延) に基づいた動作を実現する手法が提案されている [2]．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 [3–6] がある．

タイミング故障検出

回路遅延の動的な変動によって生じる過渡故障を **タイミング故障** (Timing Fault: **TF**) と呼ぶ．ワースト・ケース設計では，ワースト・ケースにおいてもこの TF が発生しないように，十分なマージンを取った電圧やクロック周波数を設定する．TF が生じるのは，サーマル・センサの故障による熱暴走など，想定外の場合に限られる．

[1] 総合研究大学院大学 複合科学研究科
School of Multidisciplinary Sciences, SOKENDAI
[2] 東京大学大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
[3] 国立情報学研究所 アーキテクチャ科学研究系
Systems Architecture Research Division, NII
[a)] ushio@nii.ac.jp
[b)] shioya@ci.i.u-tokyo.ac.jp
[c)] goshima@nii.ac.jp

TF を検出・回復する手法を用いることで，ワースト・ケース設計で定められる限界を超えて回路を高い周波数，または低電圧で動作させることができる．TF 発生による IPC の低下が十分小さく，周波数の向上（低電圧化）による恩恵が相殺されない範囲で，回路の実効遅延に応じた周波数や電源電圧で動作させることが可能になる．

TF を検出・回復する手法の代表例として Razor がある．しかし，著者が調査した限り，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．

本稿の内容

本稿は，現実的なプロセッサ に Razor を適用する方法を示す．Rocket は，RISC-V アーキテクチャに完全準拠する scalar プロセッサである．CSR (Control and Status Registers) を持ち，Linux をブートすることができる．本稿の 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 検出の最初のテスト･ベッドとなるであろう．

以下，本稿は次のように構成される．2 章では，TF からの回復手法について述べる．3 章では Rocket のマイクロ・アーキテクチャについて述べる．4 章では Razor の適用手順について述べる．

2. Razor FF

本節では Razor [7, 8] について述べる．

2.1 Razor FF のタイミング故障検出

Figure 1は Razor のパイプライン [8] 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 Figure 1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

Vol.2019-ARC-235 No.24
Vol.2019-SLDM-187 No.24
Vol.2019-EMB-50 No.24
2019/3/17

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。

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．本本稿の文脈では、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) Figure 1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．
subsection 2.3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。

subsection 4.6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。

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．

Figure 1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

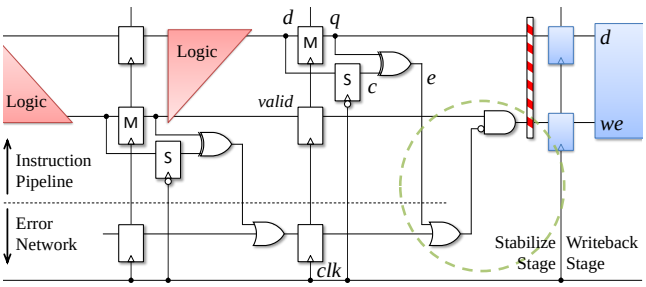

図 1　Razor FF のブロック・ダイアグラム

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．

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．

Figure 2の t-diagram 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 t-diagram 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 t-diagram 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．Figure 2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．

Razor はクリティカル・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，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．

Vol.2019-ARC-235 No.24
Vol.2019-SLDM-187 No.24
Vol.2019-EMB-50 No.24
2019/3/17

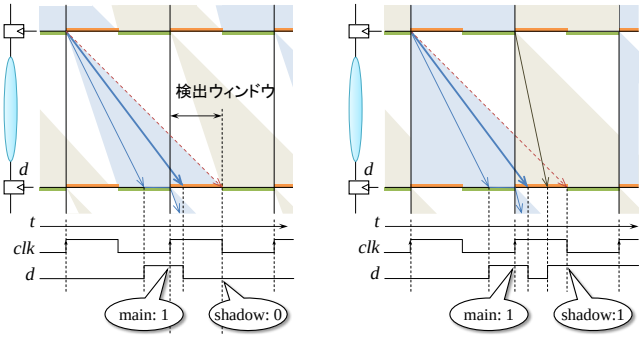

図 2　Razor のショート・パス問題

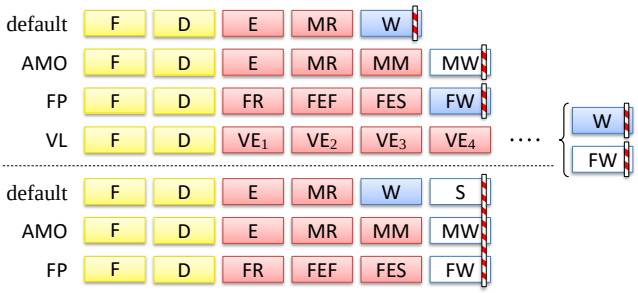

図 3　Rocket の pipeline：変更前（上）と変更後（下）

3. Rocket のマイクロ・アーキテクチャ

4 章において 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 [9, 10] 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（SS 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．

3.1 pipeline 構成

図 3 に，Rocket の pipeline 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 pipeline 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 pipeline 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．

VL　整数 MUL/DIV，FP FDIV/FSQRT 命令に対しては，可変長の pipeline 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 pipeline 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，AS 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，4.4 章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．

3.2 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 pipeline 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

Vol.2019-ARC-235 No.24
Vol.2019-SLDM-187 No.24
Vol.2019-EMB-50 No.24
2019/3/17

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．

3.3 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，Table 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，pipeline 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 table である．LL 命令がキューに移されるとき，また，キューから pipeline 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，pipeline 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インターロックする．（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インターロックし，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 *replay* 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．

replay　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 replay する．

表 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	Scoreboards
VL			
load w/ L1D miss	Interlock & Replay on L1D miss		

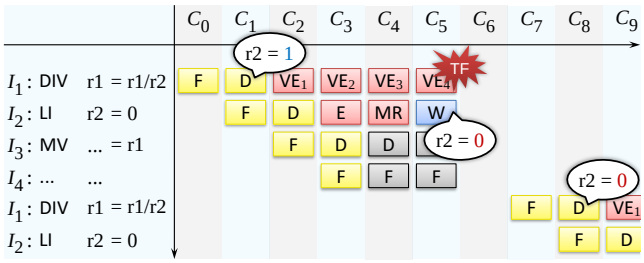

図 4　AS の Out-of-Order 更新による問題

3.4 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 4 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．
- 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，pipeline 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，4 章 で述べるように，AS の out-of-order 更新を

Vol.2019-ARC-235 No.24
Vol.2019-SLDM-187 No.24
Vol.2019-EMB-50 No.24
2019/3/17

表 2　更新前と更新後のハザード検出の真理値表

Register Numbers Match ?

Preceding Instruction LL ?		Original 0	Original 1	Modified 0	Modified 1
0			1		1
1			1	1	1

無効化する以外にない．

4. 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．section 3 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．pipeline・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) AS の out-of-order 更新の無効化
(2) AS の特定
(3) 投機状態と非投機状態の分離
(4) pipeline の変更
(5) エラー通知ネットワークの追加
(6) pipeline 再初期化の追加

以下，それぞれの項目について述べる．

4.1 AS の Out-of-Order 更新の無効化

3.3 章で述べたように，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 AS 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 4 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，AS は out-of-order に更新されることはない．

表 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．

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 pipeline は，AS の更新に関して考慮する必要がなくなる．

4.2 AS の特定

整数/FP RF に加えて，AS 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 [11]．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．

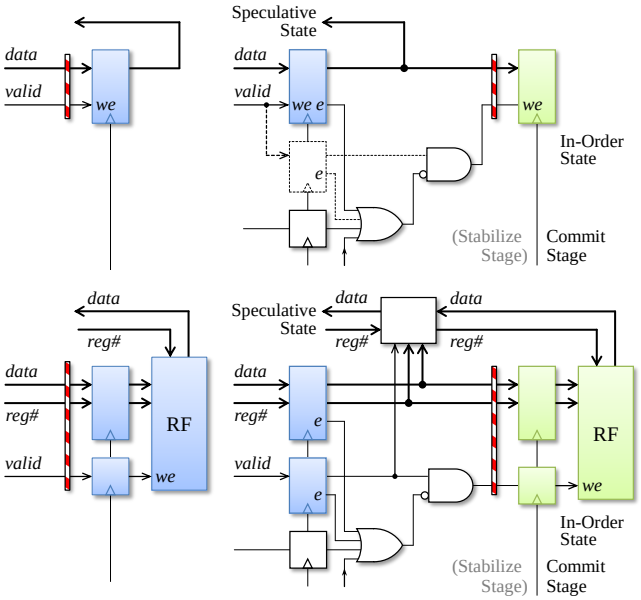

図 5　変更前（左）と変更後（右）の AS・レジスタ（上）とレジスタ・ファイル（下）

4.3 投機状態と非投機状態の分離

2 章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，AS 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．

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．

Figure 5（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．

Vol.2019-ARC-235 No.24
Vol.2019-SLDM-187 No.24
Vol.2019-EMB-50 No.24
2019/3/17

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．

4.4 pipeline 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：

- 3.1 章で述べたように，AS 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．
- Figure 3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（W 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 pipeline 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，AS の in-order 更新が保証される：

- 4.1 章で述べたように，VL pipeline は，考慮する必要がなくなる．
- Figure 3に示されるように，残り，すなわち，default，AMO，FP pipeline は，同じ長さの固定長 pipeline となる．

4.5 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 pipeline 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 W・ステージに到着すればよい [12]．

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 W・ステージまでの最短経路に基づいて構築される．

4.6 pipeline 再初期化

TF の影響を pipeline から取り除くためには，pipeline・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，pipeline の**再初期化** (re-initialization) が必要であると我々は主張してきた [12]．Rocket のようなスカラ・プロセッサに対しては，pipeline・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，pipeline 再初期化を選択した．

プロセッサ全体の**リセット木**を，AS 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 pipeline に沿ったステージごとの再初期化も提案してきた [12]．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．

5. 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 3** は開発とテスト環境をまとめたものである．

5.1 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，subsection 4.4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，[7] 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．

表 3 開発とテストの環境

Platform	lowRISC SoC project [13]
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

Vol.2019-ARC-235 No.24
Vol.2019-SLDM-187 No.24
Vol.2019-EMB-50 No.24
2019/3/17

5.2 結果

表 4 は，プロセッサコアとL1D 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．

6. おわりに

本章では，Razor を Rocket に適用した．元々の Rocket は，AS 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．

我々は，より効果的な周波数向上や低電圧動作を可能にする手法として，動的タイムボローイング（Dynamic Time Borrowing: **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いる [14–16]．現在，提案手法の回路への実装による評価を行っている．今までは，カウンタのような小規模な回路への適用のみが行われていた [15, 16]．今後はこの手法への適用のポイントの多くも Razor への適用で踏まえたポイントが活用される．

また，我々は現在，NORCS [17] 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，このプロセッサに動的タイム・ボローイングを可能にするクロッキング方式を適用し，より詳細な評価を行う予定である．

謝辞 本研究の一部は，文部科学省科学研究費補助金 No. 16H02797 による．

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] 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).

表 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	↑	↑

[4] 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).

[5] 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).

[6] 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).

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

[8] 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).

[9] 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).

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

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

[12] 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).

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

[14] 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).

[15] 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.

[16] 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).

[17] 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).