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

吉田宗史[†] 倉田成己[†] 塩谷亮太[††]
五島正裕[†] 坂井修一[†]

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対する対策の 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．しかし既存の回復技術は，単純なスカラ・プロセッサにしか対応することができなかった．我々は以前，out-of-order スーパスカラ・プロセッサにおける，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うことで，複雑な out-of-order スーパスカラ・プロセッサにも対応できる回復技術を提案している．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．評価の結果，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，回復方式のフォールト耐性を確認した．

A Detection/Recovery Scheme of Timing-Fault-Tolerant Out-of-Order Processor

SHUJI YOSHIDA,[†] NARUKI KURATA,[†] RYOTA SHIOYA,[††]
MASAHIRO GOSHIMA[†] and SHUICHI SAKAI [†]

A decrease of the feature size of LSIs leads to an increase of the effect of random variation. Techniques that detect and recover from timing faults can solve this problem. Existing techniques, however, can only be applied to simplest scalar processors. In a former proposal, we proposed a new technique that can be applied to complex out-of-order superscalar processors. The point is how to handle the faults that occurs inside of the Reorder Buffer (ROB) and the Load/Store Queue (LSQ). In this paper, we examine details of a detection/recovery scheme of our former proposal, such as physical constitution, concrete method of implementation, the recovery penalty. Evaluation shows that a rate of decline of IPC fluctuates by factors except for timing faults and that our recovery scheme is more tolerant to faults.

1. はじめに

半導体プロセスの微細化に伴い，素子のランダムな**ばらつき**の問題が顕在化しつつある[1),2)]．微細化によって，素子性能の平均 (typical) 値は向上するものの，ばらつきの増大によってワースト値の向上は減殺される．従来の LSI 設計は，このワースト値に基づくワースト・ケース設計である．したがって，このまま微細化を進めても従前のような性能向上は期待できなくなる．

ばらつきへの対策としては，様々なレベルの技術が考えられる[3)] が，回路〜アーキテクチャ・レベルの技術として，タイミング・フォールトの検出と回復がある．

タイミング・フォールト検出/ 回復

タイミング・フォールトとは，回路遅延の動的な変動によって生じる過渡故障である．本稿ではタイミング・フォールト以外のフォールトを扱わないので，単にフォールトとあった場合にはタイミング・フォールトのことを指すと解されたい．

ワースト・ケース設計では，ワースト・ケースにおいてもタイミング・フォールトが発生しないように設計を行う．そのため，フォールトが生じるのは，サーモ・センサの故障による熱暴走などの想定外の状況においてのみである．一方で，ワースト・ケースにおいてもフォールトが発生しないような見積もりは，ばらつきが増大したプロセスでは悲観的になり過ぎる．

タイミング・フォールトを検出/ 回復する技術は，特に DVFS —— Dynamic **V**oltage and **F**requency Scaling と組み合わせて用いることで，このような問題に対処することができる．すなわち，ワースト・ケースよりも低い電圧(V)，高い周波数(F)で動作させ，その結果生じるフォールトを検出し，回復すればよい．回復にはペナルティが付随するから，その影響が十分に小さくなるように，フォールトの発生確率が十分に低い V/F の組みを見つける．このようにすれば，個体差や動作環境に応じた実際的の遅延に基づく動作が可能となり，ワースト・ケース設計の悲観的過ぎる見積もりから脱却することができる．

† 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
†† 名古屋大学 大学院 工学研究科
Graduate School of Engineering, Nagoya University

このようなプロセッサを対象とする回復技術としては Razor Ⅱ[4),5)] が提案されているが，Razor Ⅱ では制御系のフォールトに対応できない．これに対し，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[6)]．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．

本稿の提案

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．

本稿の構成は以下の通りである：続く 2 章では，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 章で，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 4 章で，以前までの我々の提案の回復技術について述べる．5 章では，回復方式のペナルティなど，提案の検出/回復方式の詳細を述べる．6 章では，フォールトの発生率と IPC の評価を示す．

2. タイミング・フォールト検出/ 回復技術

本章では，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[4)] 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[7)] 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[8)~10)]．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．2.1 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，2.2 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．

2.1 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 1 (a) に，アーキテクチャ・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[11)] で原則的には構成される☆．

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

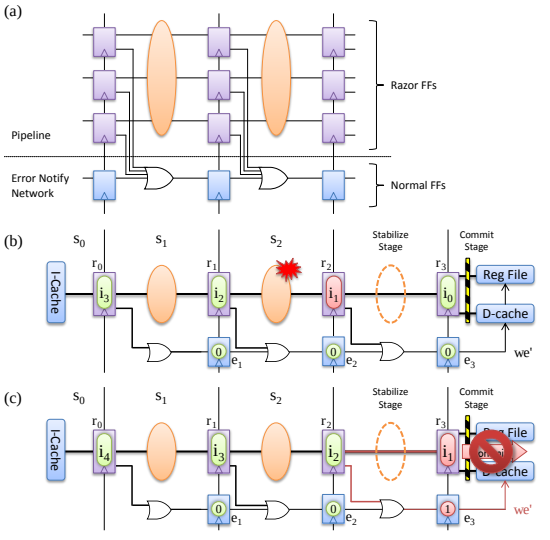

図 1　アーキテクチャ・レベルの回復技術のパイプライン

☆ 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．

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．

2.2 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1 (b)，(c) を例に Razor II のパイプラインの動作を説明する：

図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. Out-of-Order プロセッサと Razor II の限界

前章で述べた 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，3.1 節でコミットについて詳しく述べ，3.2 節で Razor II の問題点について述べる．

3.1 Out-of-Order プロセッサのコミット

3.1.1 コミットに関わるモジュール

図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

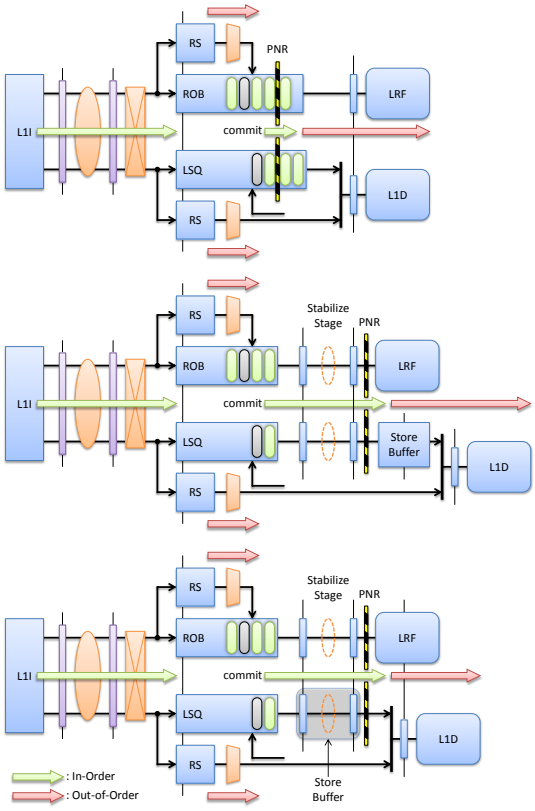

図2 前提モデル（上）と提案モデル（中，下）

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 が依存元の命令の結果を供給する．☆

3.1.2 ROB/LSQ

図 3 に ROB と LSQ の典型的な構成例を示す[12]．

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．

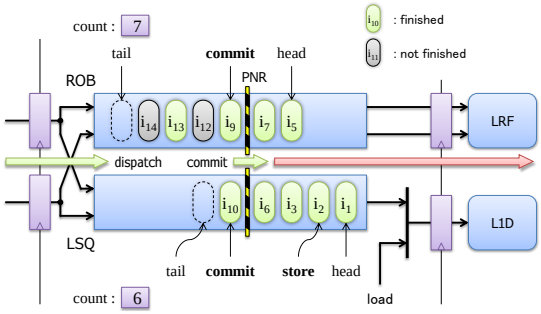

図 3　リオーダ・バッファ (ROB) と ロード / ストア・キュー (LSQ)

☆ 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 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 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 4) 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．

4. 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，前章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[6]．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．本章では，以前までの提案の概要をまとめる．

4.1 In-Order Passing through PNR

図 2（中）は提案手法のプロセッサ・モデルである．3.1.2 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 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．[6]

4.2 Imprecise Cancellation

2.2 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，3.1.2 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．

4.3 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 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．

5. 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．本章ではまず，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．

5.1 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．

一般に初期化は，掛けることよりも，正しく解除することに困難を極める．ある PL では初期化が解除されているが，別の PL では解除されていないなどということが起こっては，正しく解除されたことにならない．そのため，初期化信号の割り当てと初期化の解除，どちらも網羅的に行える機構が肝要となる．

これを満たすには，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．後に詳しく述べるが，初期化信号がサイクルに同期してこのネットワーク上を伝わり，PL を上流から順に初期化していく．提案の初期化方式の振る舞いは，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．データ・パス上のレジスタの値を初期化したところで，それがあるサイクルのデータとして扱われるのみで有用性はない．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．

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．

5.2 バック・エッジの問題と対策

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の回復方式では，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 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．同図では初期化された PL を水色で，フォールトの影響が及んだ PL を赤色で示す．図の番号はサイクル数を表している：

図 4 (1) は，rst_{pc} にセットされた初期化信号により，PL r_0 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_0 にセットされる．PL r_1〜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．

図 4 (2)：このサイクルの終わりには，PL r_1 が初期化され，初期化信号が rst_1 にセットされる．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．

図 4 (3)：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 PL r_2 のフォールトの影響が，バック・エッジにより PL 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 PL 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．**図 5** に，その対策を施した場合の動作を示す．各図は図 4 と同じサイクルを示している：

図 5 (1)：バック・エッジを制御するカウンタをバック・エッジの戻り先のステージ（ここでは s_1）の手前に設置する．同相である PL rst_0 に初期化信号がセットされるタイミングで，ctl_0 に信号をセットする．

図 5 (2)：ctl_0 にセットされた信号により，バック・エッジを無効にする．ctl_0 に信号がセットされている間は，毎サイクル，カウンタの値を 1 増やす．

図 5 (3)：バック・エッジが無効であるため，PL r_2 のフォールトの影響が PL r_1 に及ぶことはない．これにより，初期化が正しく行われることが保障される．また，このサイクルにおいて，カウンタの値がバック・エッジ長に等しくなる．これにより，ctl_0 の信号をリセットする．この次のサイクルにおいて，バックエッジを有効化し，カウンタの値を 0 に戻す．

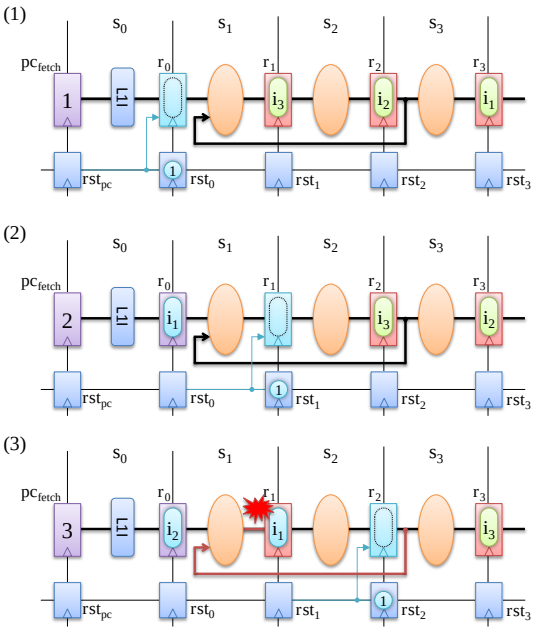

図 4 バック・エッジによる問題

5.3 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．

5.3.1 回復における制約

4.2 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．

5.3.2 回復の流れ

図 6 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 6 中，pc_{fetch} は fetchPC を，

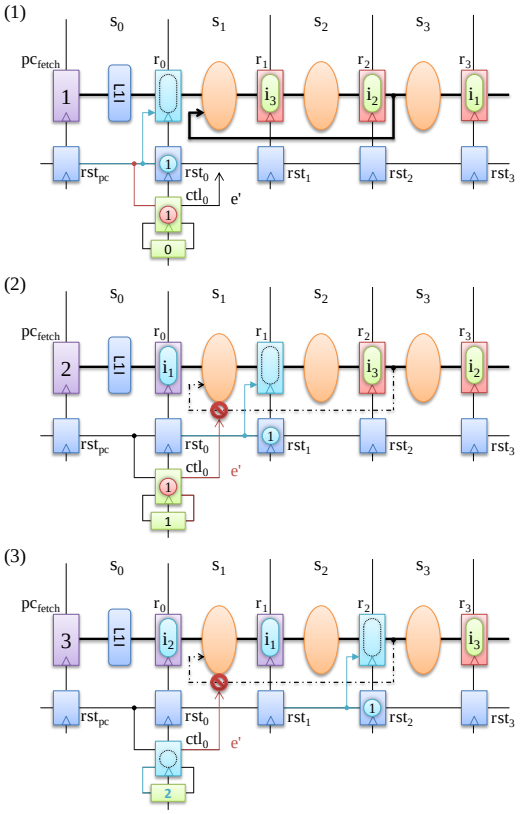

図 5　カウンタによるバック・エッジへの対策

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 6 (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 6 (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，r_1 と同相のレジスタ e_1 はセットされていない．

図 6 (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 6 (3) ：このサイクルの終わりには，i_1 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．

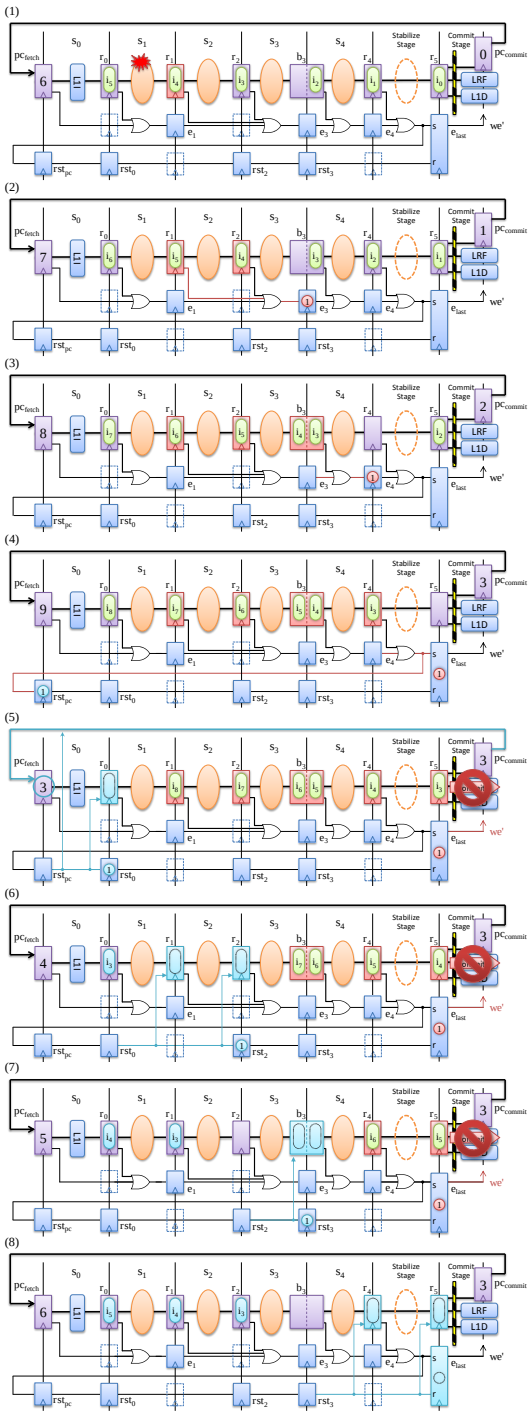

図 6　提案手法の回復技術

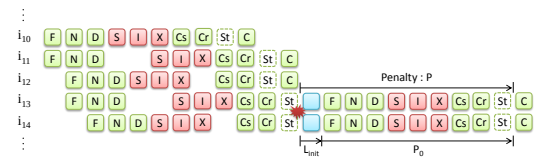

図 7 回復のペナルティ

図 6 (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 r_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 6 (5)：このサイクルの終わりには，フォールトの影響は PL r_5 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．ここで，フォールトの影響を受けた可能性のある命令の PC で書き換えるのではなく，その時点で PNR を超えていない命令の PC で書き換えを行っていることに注意されたい．

図 6 (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 6 (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 6 (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．更に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．

5.4 回復のペナルティ

図 7 は，一般的な out-of-order スーパスカラ・プロセッサのパイプライン・チャートである．この図を基に回復のペナルティについて論じる．

命令 i_{13}，i_{14} がスタビライズ・ステージを通過中に，エラー信号が PNR に到達し，フォールトの発生が検出されたとする．検出後は回復処理を行うことで，前の命令 i_{12} までの AS が保護され，初期化により，初期状態のパイプラインで再実行が行われる．つまり，前の命令 i_{10}〜i_{12} との依存関係が解消された状態で，命令 i_{13}，i_{14} の再実行が行われることとなり，初期化後，途中でストールすることなくスタビライズ・ステージまで到達する．そのため，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の初期化のレイテンシを L_{init} とすると，$P = P_0 + L_{init}$ で表される．

先に述べた回復方式で，全ての PL の初期化を待ってから再フェッチを行うと，L_{init} は最大で P_0 となる．初期化ネットワーク上のレジスタをいくつかスキップしたり，ネットワークをツリーで構築するなどの工夫により，L_{init} の削減は可能であるが，5.3.2 項で述べたように，最上流の PL の初期化後，直ちに再フェッチを行うことで，$L_{init} = 1$ とすることができる．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最大 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，$\alpha_{last} \simeq \alpha$ 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_{last} は変化せず，さらに，$\alpha_{last} \simeq \alpha$ 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できる．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．

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 1** で表される：

$$i_f = \frac{N}{c_f} = \frac{c_b i_b}{c_b + c_b \alpha P} = \frac{1}{1+\alpha P} i_b \quad (1)$$

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．

6. 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．

6.1 評価モデルと評価環境

プロセッサ・シミュレータ**鬼斬弐**[13] を用いて評価を行う．ベンチマークは，SPEC CPU2006[14] を用いる．**表 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 のストア・バッファは，図 2（下）のように，シフト・レジスタで構成する．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．

5.4 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．5.4 節で述べたように，L_{init} がパイプライン段数 P_0 と同値であるとし，$L_{init} = P_0 = 20$ 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．$L_{init} = 1$．

表 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

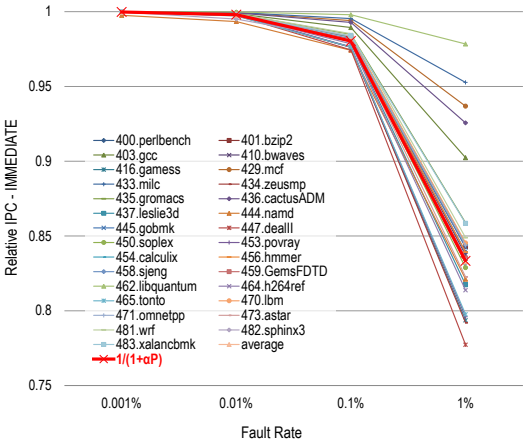

図 8　BASE に対する相対 IPC とフォールト発生率の関係

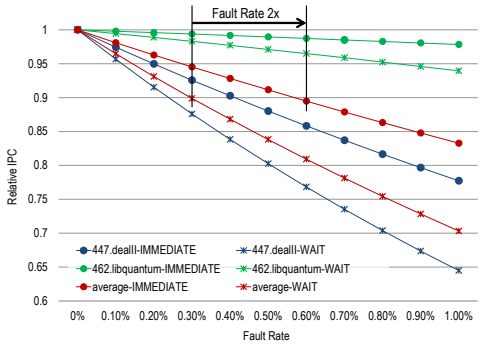

図 9　回復形式によるフォールト耐性の違い

6.2 評 価 結 果

図 8 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 8 上の赤太線は，式 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．

回復形式によるフォールト耐性

図 9 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，5.4 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．

7. おわりに

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[15] 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．また，我々の提案するフォールト検出を備えた回路方式[16] と組み合わせることも検討している．

謝辞 本研究の一部は，JST CREST「ディペンダブル VLSI システムの基盤技術」，および，文部科学省科学研究費補助金 No. 23300013 による．

参考文献

1) 岡田健一: 集積回路における性能ばらつき解析に関する研究, 博士論文, 京都大学 (2003).
2) 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).
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. Computer-Aided Design of Integrated Circuits and Systems*, Vol. 24, No. 12 (2005).
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 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) 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（ACS）, Vol. 6, No. 1, pp. 17–30 (2013).
7) 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).
8) 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).
9) Tiwari, A., Sarangi, S. R. and Torrellas, J.: ReCycle: Pipeline Adaptation to Tolerate Process Variation, *ISCA*, pp. 323–334 (2007).
10) Bowman, K. A., Tschanz, J. W., Lu, S.-L., Aseron, P. A., Khellah, M. M., Raychowdhury, A., Geuskens, B. M., Tokunaga, C., Wilkerson, C., Karnik, T. and De, V. K.: A 45 nm Resilient Microprocessor Core for Dynamic Variation Tolerance, *J. Solid-State Circuits*, Vol. 46, No. 1, pp. 194–208 (2011).
11) 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).
12) Shen, J.: *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill Science/Engineering/Math (2004).
13) 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）.
14) The Standard Performance Evaluation Corporation: SPEC CPU2006 suite, http://www.spec.org/cpu2006/.
15) 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).
16) 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（ACS）, Vol. 6, No. 1, pp. 1–16 (2013).