第2章

もっとも基本的なプロセッサ高速化技法

パイプライン処理の概念と実際

パイプラインとはMPUの命令実行を高速化する手法の一つであり，現在では，ほとんどすべてのMPUで採用されている．RISCのパイプライン処理は，見事なまでにヘネシー＆パターソンが提唱した5ステージのパイプラインにしたがっている（通称ヘネパタ本を参照のこと）．前半では一度に1命令を実行する通常のパイプライン（シングルパイプライン）についての基本概念を説明する．2命令以上を同時実行するものはスーパースカラと呼ぶが，これと対比する場合はユニスカラパイプライン，あるいは単にスカラパイプラインとも呼ばれることもあるようだ．後半ではシングルパイプラインの代表とも言えるR3000，SHシリーズ，ARM，V800シリーズのパイプライン構造を解説する．

パイプライン処理の概念

1 パイプラインとは

● 流れ作業＝パイプライン

コンピュータの性能を向上させる方法については，いろいろ考案されている．**パイプライン**とは，ハードウェアを並列化して性能を向上させるための一般的な手法である．その基本的な考え方は，プログラム内蔵方式を提唱したフォン・ノイマンによってすでに提案されていたという．たとえば，MPUの命令実行に比べて10倍以上も遅いメモリアクセスが存在する状況下で効率的に命令の処理を行うために，命令の実行とメモリアクセスをオーバラップして処理することが考えられた．これが，パイプライン処理の原型である．

パイプラインの基本的な考え方はごく自然なものである．なにもコンピュータの技術に固有なものではない．自動車の製造ラインや電子部品工場などで行われている流れ作業は，パイプラインそのものである．一つの製品が数分後ごとに完成していくようすを思い浮かべてほしい．実際，パイプラインの呼び名は，石油が次々とパイプを通過していく石油化学パイプラインと動作が似ていることに由来している．

各工程が1単位時間かかるN工程からなる処理を考える．単純に考えると，この処理を終了するためにはN時間を要する〔**図1**(**a**)〕．これをN人の人が流れ作業によって各工程を分担し，前の工程から受け継いだ製品に1単位時間で加工を施して，後の工程に引き継ぐようにする〔**図1**(**b**)〕．この場合，もともとの処理ではN時間に一つしか製品が完成しないが，流れ作業では見かけ上，1単位時間に一つの製品が完成することになる．つまり，処理速度はN倍に改善される．これがパイプラインの原理である．

● ステージ，段数，ハザード

ここで，各工程をパイプラインの**ステージ**という．「段」という表現も使われ，N工程から構成されるパイプラインはNステージパイプライン，またはN段パイプラインと呼ばれる．また，あるステージを分担する人が手間取って，そこでの処理を1単位時間以内に終わらせることができないような場合は，パイプライン処理に乱れが生じ，処理性能が低下する．パイプラインステージでの処理を単位時間内に終わらせることを阻害する要因を**ハザード**という．

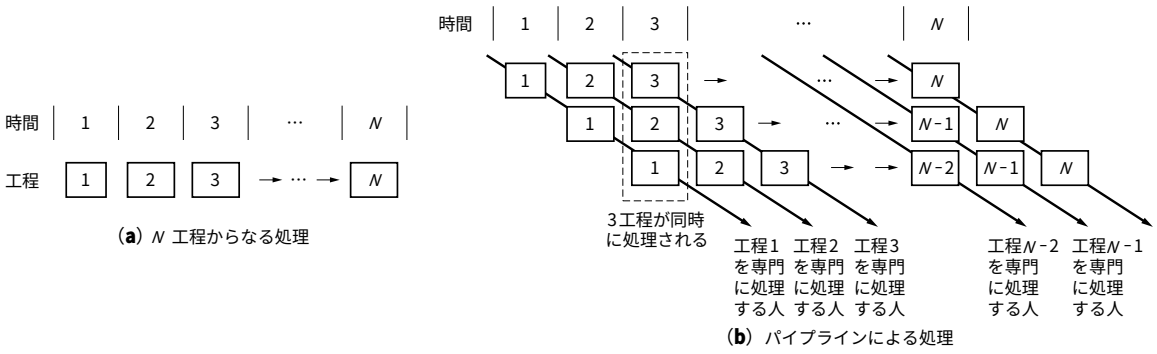


図1　パイプライン処理の概念

パイプライン処理をコンピュータに適用する場合は,各ステージが並列に処理できることが前提である.ハードウェア資源を共有するステージがあると,ハザードが生じ,待ち合わせが必要になる(これを**ストール**という).逆にいうと,ハードウェア資源が競合しないようにパイプラインステージを分割するのがプロセッサ設計者の腕の見せどころである.

パイプライン処理は,まず大型計算機で採用された.その後,半導体の集積技術が進み,MPUでも大量のトランジスタが利用可能になると,MPUにも採用されるようになる.パイプライン処理の採用を大々的に表明したMPUは,NECのV60が最初ではないかと筆者は記憶している.それ以前のIntelの8086でもオペランドフェッチと実行をパイプライン化していたが,Intelがパイプラインを明言したのは80386以後(最近のIntelの発言ではPentium以降)となっている.一方,68000系のMPUも古くからバスサイクル同期のパイプライン処理をしていたようである.しかし,こちらもパイプラインを明言したのは68060が初めてだったと思う.68060はすでにスーパースカラ構造になっていたので,シングルパイプライン時代の68000系のパイプライン構造は不明である.

2　パイプラインの理論

● パイプラインステージ

CISC初期においてもパイプライン構造を採用しているものがあった.しかし,それらのMPUにおいてパイプライン処理は有効に機能していたとはいえない.各MPUメーカーがパイプラインを強調しなかったのは,それが性能に寄与していなかったからではないかと考えられる.しかし,RISCの登場によってパイプライン処理はにわかに脚光を浴びる.RISCのパイプラインは,CISCとは異なり,全命令でパイプラインのステージ数は固定であることが多い.筆者だけの感覚かもしれないが,命令フェッチ,命令デコード,実行という処理の流れも,その区切りが明確になっているように感じる.

RISCの存在意義は,パイプライン処理をいかに効率的に実現できるかにかかっているといっても過言ではない.このため,RISCでは命令やオペランドをキャッシュからフェッチすることを前提としている.通常のメモリはアクセス時間が遅いので,メモリアクセスステージの処理時間が他のステージに比べて長くなり,効率的なパイプライン処理を行うことはできない.ステージの処理時間を均一化するため,キャッシュの導入は必然だったといえる.キャッシュの導入により,メモリアクセスステージが1または2クロックという固定クロック数で処理できるようになった.

RISCのパイプラインは,コンピュータアーキテクチャの有名な教科書で学ぶことができる.それが,HennessyとPattersonによる『コンピュータアーキテクチャ』(通称ヘネパタ本)である.この教科書では,仮想的なMPUとしてDLX(デラックスと発音する)というMPUを定義し,そのパイプラインとして次の5ステージの処理が提案されている.もっともDLXはMIPSのR2000/R3000と非常に近い(同じ?)構造をしており,以下はR3000のパイプラインそのものということもできる.ただし,ヘネパタ本ではメモリがキャッシュであることをとくに強調してはいない.

● RISCのパイプライン処理

RISCのパイプライン処理を**図2**に示す.パイプラインがスムーズに動作する場合は,全ステージ数と同じ数の命令が(理論的には)同時実行できる.

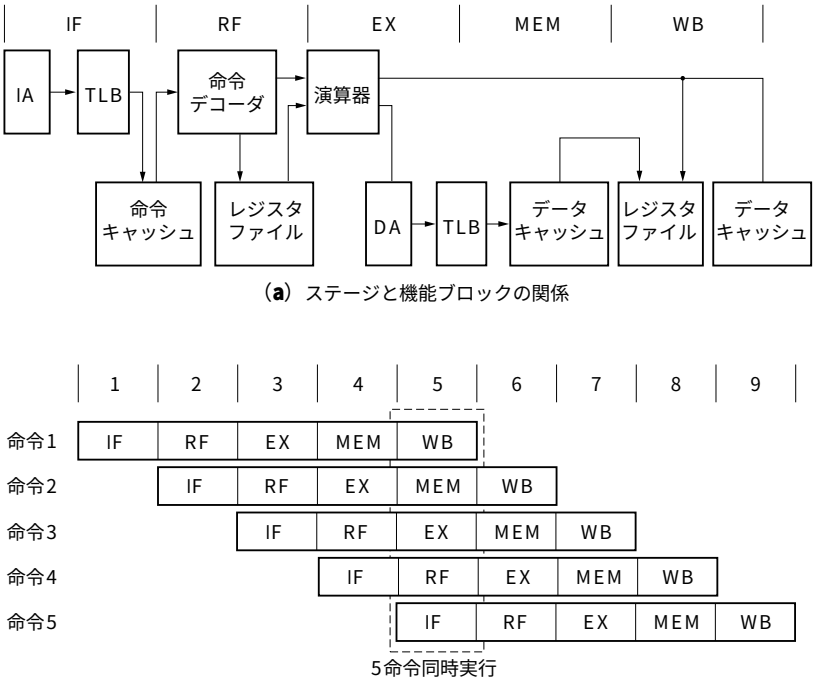


図2　RISCのパイプライン処理

(1) 命令フェッチ(IF)

命令キャッシュから命令を取り出す.

(2) 命令デコード(RF)

フェッチした命令をデコードする．同時にレジスタオペランドをフェッチする.

(3) 命令実行(EX)

デコード結果とフェッチしたレジスタの値を基に命令を実行する．ロード/ストア命令の場合は実効アドレスの計算を行う．分岐命令の場合は分岐先アドレスを計算する.

(4) オペランドフェッチ(MEM)

EXステージで計算したアドレスに対応するメモリの値をデータキャッシュからリードする.

(5) ライトバック(WB)

EXステージで計算した結果，またはMEMステージでフェッチしたオペランドをレジスタに格納する．ストア命令の場合はデータキャッシュにライトする.

上のパイプラインではアドレス変換のステージがないが，これはIFまたはMEMステージに先立って行われる．この詳しい説明は後半で解説するR3000のパイプラインの実際の項に譲る.

RISCのパイプラインの特徴は，アドレス計算をする専用のステージがなく，EXステージで代用している点である．このため，アドレス計算用の演算器と命令実行用の演算器(実際は加算器)をそれぞれ別個に用意する必要はない．これはRISCの「ロード/ストアアーキテクチャ」という特徴に由来する．つまり，一つの命令では2回加算を行うことがない，1命令で1回だけ演算器を使用するという制限の下で，レジスタとレジスタ間の加算，または，アドレス計算(ロード/ストア命令)は別の命令に分かれて定義されている.

● データハザードとフォワーディング

パイプラインの処理が乱れるハザードは，RISCのパイプラインでも発生する.それを詳しく見ていこう．まずはレジスタの依存関係に起因するハザードである．レジスタ間のリード/ライトの前後関係で，次の4種類が考えられる.

(1) RAW(Read After Write)ハザード

これは，レジスタライトの完了前に後続命令によって同一のレジスタをリードしようとした場合に生じる(**図3**).

(2) WAR(Write After Read)ハザード

これは，レジスタから値をリードする前に後続命令によって同一のレジスタにライトをしようとした場合に生じる.

(3) WAW(Write After Write)ハザード

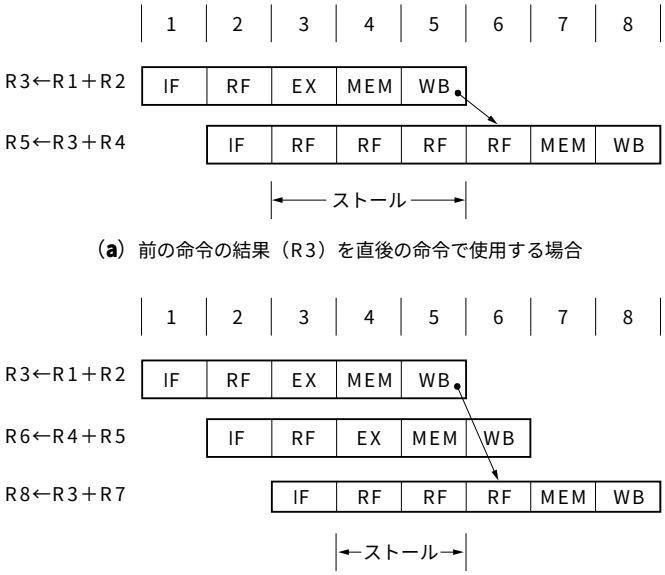


図3　RAW（Read After Write）**ハザード**

これは，同一レジスタへのライト順序が狂う場合に生じる．

(4) RAR（Read After Read）ハザード

一応挙げたが，レジスタへの変更がともなわないので，このようなハザードは存在しない．

以上はデータに起因するハザードなので，総称して**データハザード**と呼ばれる．しかし，**(2)**および**(3)**のハザードは命令の実行順序が狂わない限り発生しない．通常のパイプラインでは発生しないが，スーパースカラ構造では発生することがある．これは後半で説明する．

当面の課題は**(1)**のRAWハザードである．これは，フォワーディング，バイパス，または，ショートサーキットと呼ばれる手法で解決可能である．つまり，EX，MEM，WBステージからRFステージへのバイパス回路を設けることで解決できる（**図4**）．RISCでは，パイプライン処理を乱さないために，フォワーディングはなかば常識である．

しかし，パイプラインのステージ数が多い場合，具体的には，レジスタをフェッチするステージ（RF）とレジスタへの書き込みステージ（WB）の間の段数が多いと，各ステージからRFステージへのバイパス経路がその段数分必要なので，実行ステージ（EX）へ与えるデータのセレクタが巨大になってしまう．これはもちろん動作周波数にも影響を与える．どの程度フォワーディングを行うかは悩むところである．

● ロード遅延と遅延ロード

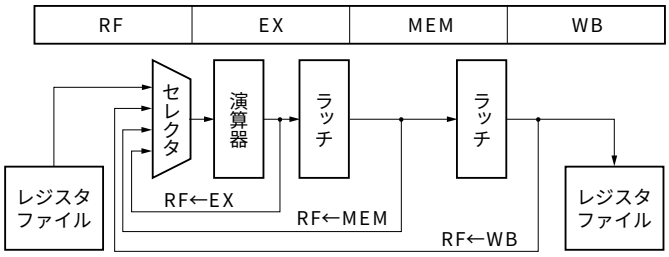
ロードした値を直後の命令で使用する場合を考える．この場合，MEMステージで値が初めて確定する．このとき後続命令はEXステージにあるのでフォワーディングは不可能である〔**図5**（**a**）〕．なにも対処しないと変更前のレジスタの値をフェッチしてしまう．この待ち時間を**ロード遅延**という．

このため，プログラムの意図どおりに命令を処理するには，パイプラインの**インタロック**が必要となる．インタロックとはハザードの有無をテストし，ハザードがある場合はハザード原因が解決するまでパイプラインを停止する機構である．

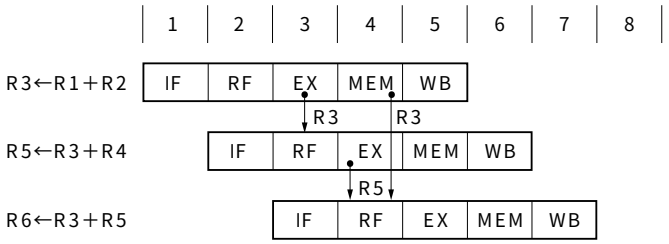
また，停止しているサイクルを**パイプラインストール**（パイプラインバブル）と呼ぶ．**図2**で示す5ステージ構成のパイプラインなら1クロックストールさせればよい〔**図5**（**b**）〕．

パイプラインのストールは，処理性能の低下を意味する．それを回避する手法の一つは，プログラムの意図を損わない範囲で命令の順序を入れ替えることである．いまの場合，1クロック分（1命令分）待ち合わせればいいので，ロードした値を参照する命令と後続の無関係な命令を入れ替えればよい．

入れ替えるべき適当な命令がない場合は，NOP命令を挿入することになる〔**図5**（**c**）〕．この手法はデータハザードの回避にも有効である．このような命令入れ替えや命令挿入を，**命令スケジュール**と呼ぶ．

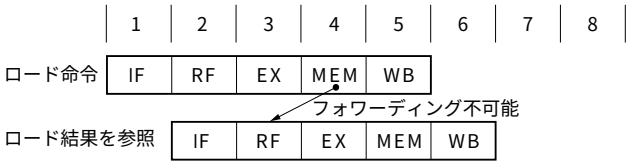


（**a**）バイパス回路



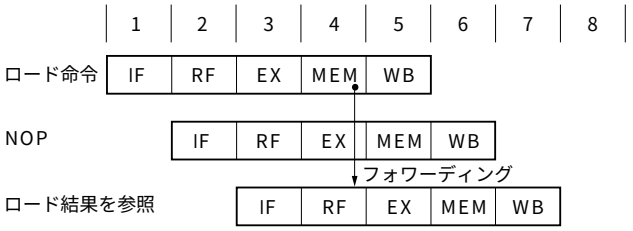
（**b**）フォワーディング（ストールしない）

図4　バイパス回路とフォワーディング



（**a**）遅延ロード

（**b**）インタロック



（**c**）NOP命令を挿入

図5　遅延ロード

RISCのアセンブラは命令スケジュールを当然のように行っている(禁止の設定も可能)．つまり，アセンブラが「勝手に」最適化するので，プログラマが書いたとおりの順序でコード生成が行われるとは限らないのである．この事実を知ったとき，筆者は少々衝撃を受けたが，いまでは慣れてしまった．

RISCは制御構造の単純化を目標としているから，インタロックは歓迎すべきものではない．ロード遅延をそのまま許し，アセンブラによる命令スケジュールによってのみストールを回避しようという考えが

図6　遅延分岐

ある．これが**遅延ロード**である．MIPSのR2000/R3000は遅延ロードを許すアーキテクチャを採用している．

ただし，R4000からはインタロックするアーキテクチャに変更された．これは，現実問題として，命令の並び替えができる場合が少なく，多くの場合はNOP命令が挿入されてしまうからであろう．NOP命令の挿入により，全体としての命令処理は1クロック余分にかかるが，これはストールで1クロックインタロックしても同じである．それならNOP命令がない分，命令コードのサイズを小さくできるという利点がある．

● 制御ハザード

パイプライン処理を乱すハザードにはデータハザードのほかに**制御ハザード**がある．これは分岐によるハザードである．ブランチハザードともいう．RISCでは，条件分岐は汎用レジスタの値で分岐条件を決定する．MPUによっては，CISCと同じく条件フラグを採用しているものもある．この場合の制御ハザードはフラグハザードともいう．

さて，条件分岐の場合，分岐条件が確定するまで分岐先の命令フェッチができない〔**図6**(**a**)〕．これによるストールは命令スケジュール(条件確定を早くする)で回避できる場合もある．

条件フラグを使用する場合でも命令スケジュール可能だが，そのアルゴリズムは非常に難しい．RISCが条件フラグを採用しない理由の一つは，コンパイラでの命令スケジュールを簡単にするためである．なお，条件分岐で分岐条件が成立して分岐することをTAKEN，分岐条件が成立せず分岐しないことをNOT TAKEN(あるいはNO TAKEN)という．

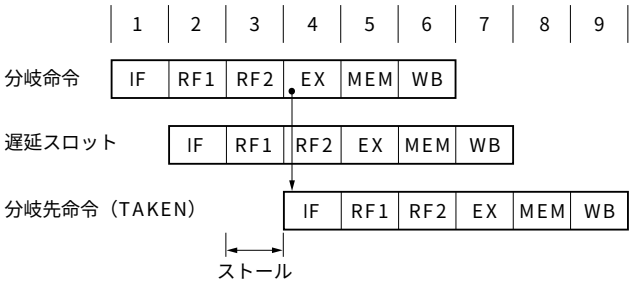


図7　命令デコードが2ステージの場合の制御ハザード

● 遅延分岐

パイプライン処理を乱さないため，ストール期間中も(通常は無効化してしまう)分岐命令の後続命令(これを遅延スロットという)を実行させるという考えがある．**図2**に示すパイプラインでは，EXステージでTAKEN/NOT TAKENが決定される．したがって分岐先の命令フェッチは，1クロックのストール後に実行可能である．

TAKENする場合，通常なら分岐命令の後続命令は実行を禁止しなければならない．しかし，その遅延スロットの命令を実行してから，分岐先の命令をフェッチする構造にすれば，パイプラインはストールする必要はない〔**図6**(**b**)〕．TAKENしない場合は，もともとストールしない．これを**遅延分岐**と呼ぶ．

このような遅延スロットを設ければ，命令スケジュールを行うことができる．分岐命令の前方にある命令を遅延スロットにもってくることで，分岐命令によるストールはなくなる．ただし，遅延スロットに入れる適当な命令がない場合は，NOP命令を入れることになる．

R2000/R3000のパイプラインはこのようになっているが，現実問題としては，分岐命令の分岐先アドレスもEXステージで計算される(したがって，分岐条件を判断するための専用の演算器が別個に必要である)ため，それとほぼ同時に分岐先を命令フェッチするのはタイミング的に厳しい．動作周波数を向上させるためには，遅延分岐を採用しつつも，もう1クロック遅れさせるのが望ましい〔**図6**(**c**)〕．このあたりをうまく回避するのが回路設計技術であるということもできるが…．一般的には，分岐予測を行うことでストールを解消することが可能である．

さて，制御ハザードではTAKENの決定が遅いほどストール期間が長くなる．これはステージ数の多いパイプラインで顕著になる．たとえば，可変長命令を採用するx86のようなMPUにおいては命令デコードに時間がかかる．一般的には，パイプラインで少なくとも2ステージ分が必要である．

たとえば，

IF　RF1　RF2　EX　MEM　WB

の6ステージからなるパイプラインを考える．TAKENの決定はEXステージなので，これまでの説明より1クロック遅いことになる．このとき分岐命令でのストールは2クロックである(**図7**)．1クロックを遅延スロットで埋め合わせるとしても，さらに1クロックだけ処理に余計な時間がかかる．あとで述べるスーパーパイプラインでは，EXステージより前のステージ数がさらに増加し，分岐命令のストールによる性能低下は深刻なものとなる．

● 分岐予測

分岐命令の処理を高速化するために，**分岐予測**という機構が採用される．これは，分岐先アドレスをパイプラインのより早いステージで生成し，分岐先の命令フェッチを早期に行う手法である．具体的には，分岐ターゲットバッファ(BTB：Branch Target Buffer)，または分岐予測テーブル(BPT：Branch Prediction Table，BHT：Branch History Table)と呼ばれるキャッシュを用意し，分岐命令のアドレス，分岐履歴情報，予測される分岐先アドレスを格納しておく．

命令フェッチ時(IFステージ)にBTBを参照し，ヒット(登録してある分岐命令のアドレスと命令フェッチアドレスが一致)すれば，分岐履歴情報にしたがって，分岐先アドレスを出力し，命令フェッチを行いながら，TAKEN/NOT TAKENの判定を待つ．予測が成功すればフェッチした命令をそのままデコードすればよい．

予測が失敗すれば，実際にEXステージで計算されるアドレスから命令フェッチをやり直し，BTBの分岐履歴情報を更新する(**図8**)．BTBにヒットし予測が成功する場合はストールがなくなる．BTBにヒットしない場合は，分岐予測を行わない場合と同じタイミ

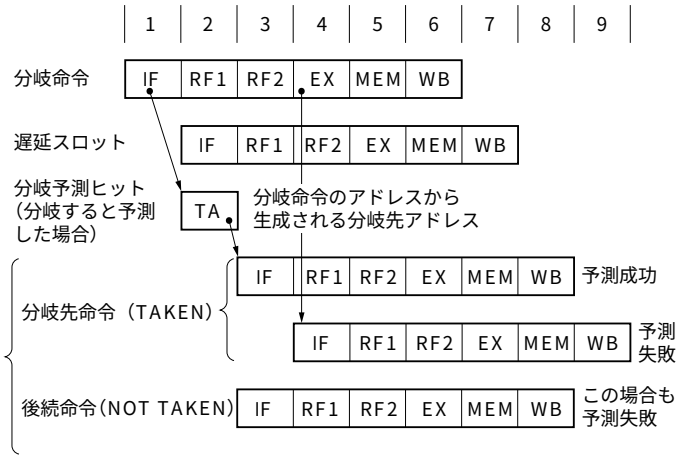


図8　分岐予測

ングで分岐命令が処理される．

しかし，BTBにヒットするのに予測が失敗する場合は，何もしない場合に比べて，パイプラインの回復処理にかえって時間がかかってしまうことがある．これが，分岐予測失敗時のペナルティである．したがって，分岐予測を採用しても予測が失敗ばかりすると，かえって性能が低下するのでヒット率を向上させるための工夫が必要である．

図8のパイプラインのモデルではBTBにヒットすると予測した分岐先アドレスから命令フェッチを行うが，MPUによっては（予測する）分岐先の数命令をBTBに格納しておき，そこから命令をフェッチする方法を採用する．こうすることにより，パイプラインは予測していない方向の命令も同時にフェッチできるので，分岐予測が失敗した場合のペナルティを最小化できる．

また，分岐予測の成功する確率が高いと思われる場合は，TAKEN/NOT TAKENが決定するまで，予測した分岐先から命令をどんどん先取り（プリフェッチ）する手法もある．パイプラインのステージ数が大きく，TAKEN/NOT TAKENの決定がパイプラインの遅い（後段の）ステージで行われる場合，予測が成功すれば効果的である．逆に予測が失敗したときのペナルティは大きくなる．分岐予測の成功率によほどの自信があるか，失敗時の回復処理がかなり高速化されてないと採れない方式であるが，最近のMPUではけっこうポピュラーである．

● 分岐予測の方法

予測の方法は分岐履歴情報による場合が多い．これは分岐する確率を示す1～2ビットのフラグであり，BTBに登録されている分岐命令ごとに存在する．分岐履歴情報が1ビットの場合は1であるとき「分岐する」，0であるとき「分岐しない」と予測する．これは，その分岐命令が過去1回で分岐したか否かを示している．つまり，以前分岐した分岐命令は今回も分岐すると予測するわけである．

分岐履歴情報が2ビットの場合はもう少し慎重である．ビット列への意味のもたせ方はいろいろ考えられるが，たとえば，11，10で「分岐する」，01，00で「分岐しない」と予測する．これは，その分岐命令が，過去2回において何回「連続して」分岐したかを示す．分岐する傾向が大きい方向に予測するわけだ．

なお，分岐する（と予測する）分岐命令のみをBTBに登録する方法もある．この場合は分岐履歴情報は不要で，BTBにヒットすれば「分岐する」，ヒットしなければ「分岐しない」と予測する．この場合，分岐予測が成功する確率は，分岐履歴情報が1ビットの場合とほぼ同等であるが，BTBの回路規模は約半分になる．

分岐予測を行わない場合で，分岐命令を高速化する方法として，分岐先と分岐元の命令を同時にプリフェッチする手法もある．これに関係する特許は，昔は山のようにあった．この方法は回路規模が大きくなるため，あまり現実的でない．といいつつも，Intel系のMPU（とくにIA-64）ではそのような説明をよく目にする．ただし，具体的な実装方法は不明である．米国の特許をよく調べればわかるかもしれない．

● 構造ハザード

構造ハザードとは，パイプラインの二つ以上のステージが一つしかないハードウェア資源を取り合うた

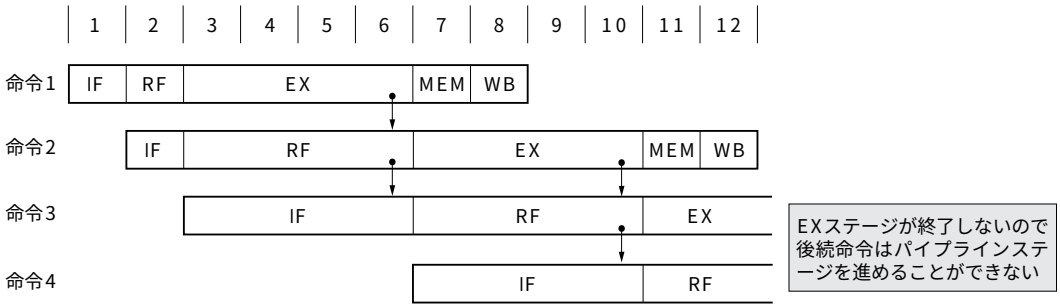


図9　実行ステージが長いパイプライン

めに生じるハザードである．たとえば，5ステージで構成されるパイプラインでは，1時刻に五つすべてのステージが実行される可能性がある．もし，各ステージで同一の演算器などを使用する場合は競合するので，優先されるステージ以外は待ち合わせをする必要がある．

RISCの場合，ほとんどのハードウェア資源は競合しないように設計されているのであまり問題はない．しかし，例外もある．それはキャッシュ(あるいはメモリ)である．**図2(b)**をもう一度見ていただきたい．時刻4において命令1のMEMステージと命令4のIFステージが重なっている．もし，命令1がロード/ストア命令であり，命令とデータキャッシュの区別がなく単一のキャッシュしかない場合は，IFステージもMEMステージもキャッシュアクセスなので，資源の競合が生じる．キャッシュが存在しない場合もメモリアクセスの競合が生じる．この場合は，先にある命令1のMEMステージを優先させ，命令4のIFステージをインタロックして待ち合わせることになる．これは，できるだけパイプラインをインタロックさせないというRISCの考え方に反する．

そこで，多くのMPUでは命令とデータを二つのキャッシュに分割して同時にアクセスできるようにしている．これならアクセスの競合によるインタロックは発生しない．このように命令とデータの供給経路を独立させる方式をハーバードアーキテクチャという．

なお，命令とデータに関しては，TLBが一つしかない場合，アドレス変換時にも資源の競合が生じる．それを避けるため，命令用とデータ用のTLBを独立に用意するアーキテクチャもある．多くの場合，命令はアクセスするアドレス範囲が小さい(あるいは連続している)ため，命令用のTLBをマイクロTLBとして，仮想アドレスと物理アドレスのペアを本当のTLBからキャッシュして持っているのが普通である．

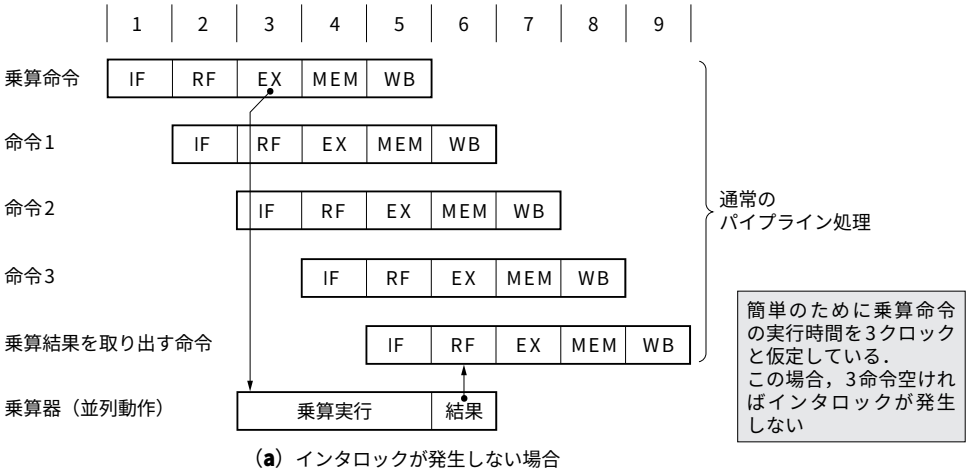
● ステージの処理時間が不均一なパイプライン

さて，パイプラインのステージ間の実行時間が均一でない場合を考える．RISCは命令を1クロックで実行するのが基本であるが，乗除算や浮動小数点演算など1クロックで実行するのが難しい場合もある．

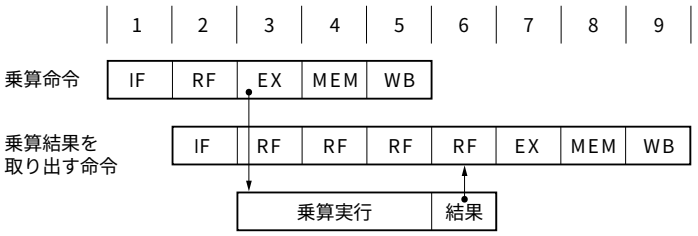
いま，実行ステージ(EX)の処理が4単位時間かかるものとする(**図9**)．この場合，EXステージが終了するまで同時実行中の他のステージも待ち合わせをするので，パイプラインのスループットは実行ステージの処理時間に依存する．ほかのステージの処理時間は実行ステージの処理時間に隠れてしまう．実行ステージの処理時間が長いだけならまだよい．ほかのステージもまちまちな処理時間を有する場合はもっと悲惨である．不均一であればあるほど，パイプラインの処理時間は各パイプラインステージの処理時間の総和に近づいていく(パイプラインの意味がなくなる)．このため，実行ステージ以外のステージの処理時間を均一にすることが肝要である．

パイプラインにおいて実行(処理)時間がかかるのは，特定命令の「実行ステージ」のほかに，メモリの速度に依存する「命令フェッチステージ」や「オペランドフェッチステージ」がある．RISCは，キャッシュを採用することで命令フェッチやオペランドフェッチの処理時間を1クロックに押し込めようとしている．

典型的なRISCであるMIPSアーキテクチャにおいては，全命令の実行クロックを1クロックとするために，実行時間がかかる乗除算は，通常のパイプラインとは独立して並列実行する．そして，乗除算の結果は汎用レジスタではなく，専用のレジスタに格納される．つまり，乗除算命令では汎用レジスタ間のデータハザードは発生しない．このため，乗除算命令の処理は通常のパイプライン動作に影響を与えない．乗除算が完了した後で，専用レジスタから演算結果を取り出せば(専用レジスタから汎用レジスタへの転送命令が用意



（**a**）インタロックが発生しない場合



（**b**）インタロックが発生する場合

図10　MIPSの乗除算命令

されている）インタロックは発生しない．

初期のMIPSプロセッサであるR3000では，乗算と除算の実行時間がそれぞれ12クロックと35クロックである．乗算命令に関していえば，実行を開始してから12クロック後に結果を取り出せばインタロックは発生しない〔**図10**(**a**)〕．プログラム的には乗算命令と結果を取り出す命令の間が12命令分空いていればよい．

一方，12クロック未満で結果を取り出そうとすると，アーキテクチャ的には不本意ながらインタロックしてしまう〔**図10**(**b**)〕．現実的には乗除算命令と結果を取り出す命令の間はせいぜい3命令程度しか空けることができないので，乗除算命令があるとほとんどの場合インタロックしてしまうのだが，コンパイラの頑張りによってはインタロックしない可能性を残している．

3　パイプラインを効率良く動かす各種の方法

● 効率的なパイプライン処理が可能になった理由

歴史的に見れば，キャッシュメモリ（高速なローカルメモリ）がまだ高価で外付けのキャッシュすら現実的でなかった時代，プロセッサの処理はメモリからの命令フェッチにいちばん時間がかかっていた．当然の流れとして，プロセッサの性能を上げるためには，フェッチする命令数を減らすこと，1命令で行う処理を増やすことが考えられた．結果として，上述したように実行ステージが長くなる傾向になるのだが，多くの場合はいちばん時間のかかる命令フェッチと，あまり時間のかからない命令のデコードおよび実行をオーバラップ（パイプライン処理）させて実行効率を上げることが可能になる．これが，その時代の最適解であった．そして，これこそがCISCの考え方である．

その後RISCという選択肢が現れてきた背景には，キャッシュが一般的になり，命令フェッチがもはやプログラムの実行に支配的でなくなったことがある．命令のデコードや実行時間が命令フェッチ時間の影に隠れなくなり，実行する命令数よりも1命令の実行時間のほうが性能に対し支配的になった．RISCでは，基本的に1クロック実行なので，CISCに比べて命令実行時間が1/3から1/5になる．1命令が単純な分，同じ処理に要するコード量は増加するが，RISCになる