■6 群（コンピュータ･基礎理論とハードウェア）- 4 編（コンピュータアーキテクチャ（I）標準的）

7 章 命令パイプライン

（執筆者：五島正裕）［2018 年 12 月 受領］

■概要■

命令パイプラインは，命令レベル並列性を利用する最も基本的な方式の一つである．したがって，その理解のためには，まず命令レベル並列性に関する理解が必要である．命令の逐次的な実行によって定義される命令セットアーキテクチャの意味論に対して，命令の実行の順序を入れ替えると間違った結果を生じるとき，それらの命令の間には先行制約があるという．先行制約の原因には，命令間のデータ依存と制御依存がある．データ依存は，フロー依存，逆依存，出力依存，入力依存に分類される．

命令パイプラインは，1 つの命令の実行を複数のステージに分割し，複数の命令を流れ作業的に実行する．1. 命令フェッチ，2. デコード，3. 実行，4. メモリアクセス，5. ライトバック の 5 ステージへと分割する 5 ステージパイプラインは，教科書的ではあるが，そこそこの実用性を備えている．命令パイプラインのパイプライン動作は，パイプラインダイアグラムによって描写することができる．パイプライン動作を妨げる潜在的要因をパイプラインハザードという．パイプラインハザードは，1. 構造ハザード，2. データハザード，3. 制御ハザード の 3 種に分類することができる．ハザードを解決する一般的な手法としてパイプラインストールがある．パイプラインをストールさせると，パイプラインにはパイプラインバブルが発生し，効率が低下する．構造・データ・制御ハザードのそれぞれに特化したストールサイクルの削減手法がある．命令パイプラインの効率性は，CPI によって測ることができる．

パイプライン実行の重要性が認識されるにつれて，CISC と RISC という分類が提唱された．RISC は，最適化コンパイラによるコード生成を背景とし，パイプライン実行の効率の向上を目標として，ISA の変更を含む最適化を行ったものである．RISC は，成功はしたものの，過適応の側面がある．

【本章の構成】

本章では，命令レベル並列性（7-1 節），命令パイプライン（7-2 節），CISC と RISC（7-3 節）について，それぞれ述べる．

■6 群 - 4 編 - 7 章

7-1 命令レベル並列性

（執筆者：五島正裕）[2018 年 12 月 受領]

7-1-1 命令の逐次実行と並列実行

マシン語命令，あるいは，単にマシン命令は，その黎明期には，一つずつ逐次的に実行されることがほとんど自明のことであっただろう．命令の逐次実行とは，すなわち，先行する命令の実行がすべて完了してから，後続の命令の実行を始めることである．

しかし時代が下り，より高い性能が求められるようになると，この逐次実行が足枷となる．そこで，この逐次実行の制約を部分的に緩和し，複数の命令を並列に実行するプロセッサが現れた．命令の並列実行とは，逐次実行とは反対に，先行する命令の実行がすべて完了する前に，後続の命令の実行を始めることと定義される．

もちろん，あらゆる命令を並列に実行できるわけではない．マシン語プログラムに内在する，命令の並列実行の可能性のことを，命令レベル並列性（Instruction-Level Parallelism：ILP）という．プログラムの持つ ILP を利用して実際に命令を並列に実行することを，ILP を抽出する（Exploit）という．

命令の並列実行が一般的になるにつれ，どの命令とどの命令をどの程度並列に実行すべきかを厳密に議論する必要が生じる．そして，そのためにはまず，その限界，すなわち何を守るべきなのかを厳密に定義する必要がある．ここにおいて，元の逐次実行と同じ結果を生じることを制約条件とすることは，経緯からすれば合理的である．そこで，命令セットアーキテクチャ（Instruction-Set Architecture：ISA）の一部として，逐次実行と同じ結果を生じる必要がある旨が明示的に定義されるようになった．すなわち，ISA における命令の意味論は，逐次実行によって定義されることになったのである．現在では，より厳密に，プログラムから観測可能なプロセッサのあらゆる状態があらゆる時点において逐次実行のそれと変わらないと定義される．

そうではなく，ILP を陽に定義する ISA も幾つか出現したが，何れも短命に終わっている．命令の意味論を逐次実行によって定義することは，単に経緯からというだけではなく，技術的にも合理性があると考えてよいだろう．

以下では，命令の意味論を逐次実行によって定義する ISA について述べる．

7-1-2 先行制約とデータ依存 / 制御依存

あるプログラムの実行において，命令の逐次実行によって定まる命令の実行の全順序を，シーケンシャルオーダという．

シーケンシャルオーダにおいて，先行する命令 I_p と後続の命令 I_s に対して，I_p と I_s の実行順序がシーケンシャルオーダ通りでないとプログラムの実行結果も変わってしまうとき，I_p と I_s の間には先行制約があるという．ILP は，先行制約を満たす範囲内に制限される．

先行制約を生じる原因には，以下の 2 種の命令間の依存関係がある．

データ依存 先行する命令と，その結果を使用する後続の命令との間に生じる．

先行する命令が実行されて結果が生産されるまで，その結果を消費する後続の命令を

実行することはできない．

制御依存 先行する（条件）分岐命令と後続の命令の間に生じる．

先行する分岐命令が実行されて，その結果（分岐の成立 / 不成立）が決まるまで，後続の命令を実行することはできない．

通常の ISA においては，シーケンシャルオーダ上の先行する命令 I_p と後続の命令 I_s の間のデータの授受は，これらの命令が同一のロケーションに対して定義（書き込み），参照（読み出し）を行うことによって実現される．したがって，データ依存は，更に，**表 7·1** に示す 4 種に分類される．

表 7·1 データ依存

		I_p	
		読み出し	**書き込み**
I_s	**読み出し**	**入力依存**	**逆依存**
	書き込み	**フロー依存**	**出力依存**

これら 4 種のうち，本当の意味で先行制約を生じるのはフロー依存のみであり，真の依存という．逆依存と出力依存は，ロケーションの再利用によって生じる偽の依存であり，リネーミングによって解消することができる．入力依存は先行制約を生じない．

■6 群 - 4 編 - 7 章

7-2 命令パイプライン

（執筆者：五島正裕）[2018 年 12 月 受領]

7-2-1 命令実行のフェーズ

1 つの命令の実行は，典型的には，以下の 5 つのフェーズに分解することができる：

F Fetch．命令フェッチ：命令を，命令メモリ，あるいは，命令キャッシュから取り出す．

D Decode．デコード：命令をデコードする．また，レジスタファイルから必要なデータを読み出す．

E Execute．実行：読み出されたデータに対して，命令によって指定された演算（Operation）を実行する．

M Memory．メモリアクセス：(必要であれば) データメモリ，あるいは，データキャッシュへのアクセスを行う．

W Write-back．書き戻し：演算の結果（Result）を，レジスタファイルへ書き戻す．

7-2-2 命令パイプライン

命令パイプラインでは，1 クロックサイクルごとに，命令実行の各フェーズを流れ作業的に実行する．この場合，各フェーズのことを特にパイプラインステージと呼ぶ．

1 命令の処理をどのようにステージに分割するかは，半導体テクノロジーや，性能とコストのトレードオフなどによって決まる．前述の 5 つのフェーズのそれぞれを 1 つのステージとしたものは，（フル）5 ステージパイプラインと呼ばれ，教科書的ではあるものの，そこそこの実用性を備えている．現代でも，ローエンドのプロセッサには，ほどよいトレードオフを提供する．

命令パイプライン中を命令が進む様子は，**図 7･1** のようなパイプラインダイアグラムによって表すことができる．図中，下方向がシーケンシャルオーダを示し，右方向が時間（サイクルタイム）の経過を示す．例えば，命令 I_1 は，サイクル C_1，C_2，C_3，C_4，C_5 に，それぞれ，F，D，E，M，W ステージにある．サイクル C_2 には，命令 I_1 が F から D ステージに進むと同時に，命令 I_2 が F ステージに現れる．

ある命令を横方向に見れば，その命令がいつどのステージに進んだかが分かる．逆に，あるサイクルを縦方向に見れば，そのサイクルには各ステージにどの命令がいるかが分かる．例えば，サイクル C_5 には，F，D，E，M，W ステージのそれぞれには，I_5，I_4，I_3，I_2，I_1 があり，パイプライン全体としては 5 つの命令が同時に処理されいてることが分かる．次のサイクル C_6 では，I_1 がパイプラインを抜け，I_6 が新たに投入されるので，F，D，E，M，W ステージのそれぞれには，I_6，I_5，I_4，I_3，I_2 があり，やはりパイプライン全体としては 5 つの命令が同時に処理される．以降も，命令が新たに投入される限り，パイプライン全体としては 5 つの命令が同時に処理されることになる．このような並列性のことを特に時間並列性

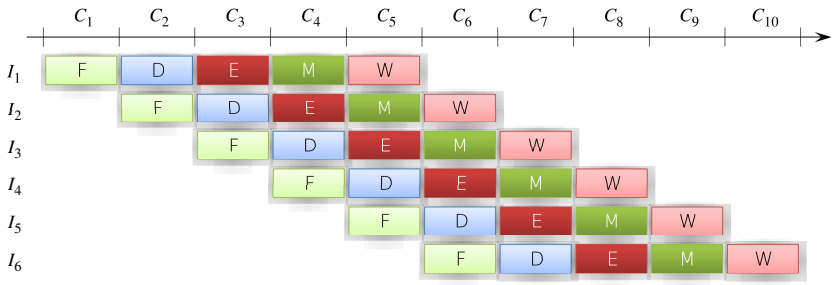


図 7·1　パイプラインダイアグラム

と呼ぶことがある．

7-2-3 パイプラインハザード

図 7·1 に示したような，毎サイクルすべての命令が次のステージに進める状態は理想的なもので，実際にはいくつかの原因により，幾つかの命令が次のステージに進めないことがある．進めないのは，それを無視しして次のステージに進むと逐次実行とは異なる，すなわち，間違った結果を生じるからである．このとき，その潜在的な要因をパイプラインハザードと呼ぶ．

パイプラインハザードは，以下の 3 種に分類される．

構造ハザード　ハードウェア資源の不足による．

データハザード　命令間のデータ依存，特にフロー依存による．

制御ハザード　命令間の制御依存による．

なお，前節の議論では，データハザードと制御ハザードを命令を単位として扱ったが，命令パイプラインにおいては，命令の実行がステージへと細分されるため，ステージ単位で考える必要がある．

以下の項で，それぞれについて述べる．

(1) 構造ハザード

図 7·2 に，構造ハザードのあるパイプライン実行の様子を示す．同図では，命令 I_1 はロード命令（*load*）であり，サイクル C_4 でメモリにアクセスする．

このとき，命令のフェッチ（F ステージ）とメモリアクセス（M ステージ）が同一のメモリに対して行われるとすると，I_1 の M と I_4 の F は同時には実行できないことになる．仮に M の方が優先的に実行されるとすると，I_4 は，I_4 ではなく，I_1 によってロードされた値を命令かのようにして実行してしまうことになる．

(2) データハザード

図 7·3 に，データハザードのあるパイプライン実行の様子を示す．同図では，I_1 の計算した結果 (r4) を I_2 が使用している．

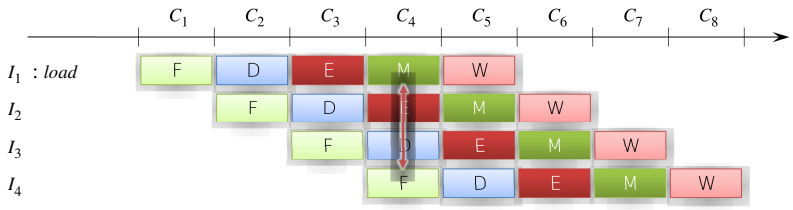


図 7･2　構造ハザード

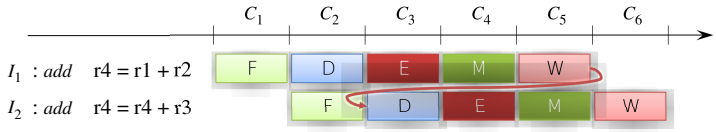


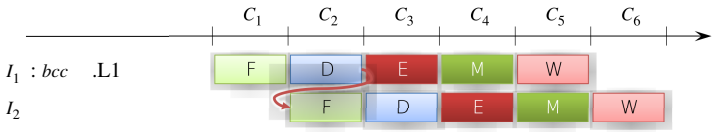
図 7･3　データハザード

しかし，サイクル C_3 において，I_2 がレジスタファイルから r4 の値を読もうとするとき (D)，I_1 はまだ r4 の値を計算中である．結果をレジスタファイルに書き込むのは，C_5 まで待たなければならない．サイクル C_3 においてレジスタファイルから読み出されるのは r4 の古い値であり，I_2 は間違った値を使って計算を行うことになる．

命令パイプラインにおいては，入力依存は言うまでもなく，逆依存と出力依存もハザードを生じない．命令パイプラインでは，D ステージと W ステージを通過する順序がシーケンシャルオーダの逆順にはならないからである．

(3) 制御ハザード

図 7･4 に，制御ハザードのあるパイプライン実行の様子を示す．同図では，I_1 は条件分岐命令（*bcc*）である．条件分岐命令の結果が成立か不成立かが決まるのは，命令セットアーキテクチャや実装に依存するが，早くて D ステージの終わり（同図では C_2 の終わり），遅いと E ステージの終わり（C_3 の終わり）になる．



[b]

図 7･4　制御ハザード

例えサイクル C_2 の終わりに分岐方向が決定したとしても，そのときには既に I_2 はフェッチされてしまっている．I_1 の正しい分岐先が I_2 ではなかった場合，I_2 は間違って実行されることになる．

7-2-4 ハザードの解決

正しい実行のためには，ハザードは何らかの方法で解決しなければならない．

(1) パイプラインストール

ハザードを解決する最も一般的な方法は，パイプラインを（部分的に）停止するパイプラインストールである．パイプラインストールは，パイプラインインターロックともいう．ハザードが生じる根本原因は，(全体ではないとはいえ) 命令を並列に実行することである．したがって，極端な話，先行する命令がパイプラインを抜けるまで後続の命令を投入しなければ，必ず逐次実行と同じ結果を得ることができる．もちろん，無駄に停止する必要なないので，状況に応じて最小限にすればよい．

図 7･3 に示したデータハザードの例を，ストールによって解決する様子を**図 7･5** に示す．前述のように，サイクル C_3 の D ステージにおいて，I_2 はレジスタファイルから必要な値を読み出すことができない．そこで，次のサイクル C_4 においては，I_2 は D ステージにとどまる．そして，次の E ステージには，自身の代わりにパイプラインバブルを注入する．注入されたバブルはそれ以降，何もしない nop 命令であるかのように，パイプラインのステージを下流へと進んでいくことになる．

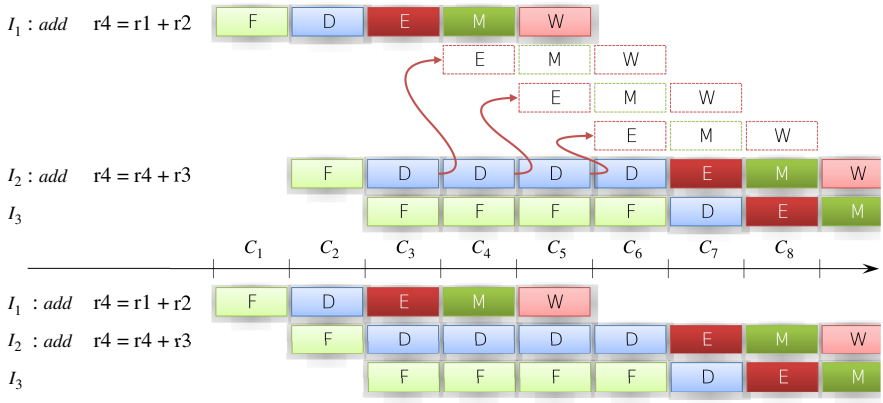


図 7･5　パイプラインストール

この I_2 の代わりにバブルを注入すると意識することは，ハードウェア実装上重要である．さもないと，古い値を読んだ I_2 が下流へと進むかのように動作することになり，パイプラインの状態は（少なくともいったんは）間違った値で更新されることになる．

D ステージにとどまった I_2 命令は（実装もよるが，典型的には）レジスタファイルを読み出し続ける．また，図中の曲線矢印によって示されるように，とどまっている間は E ステージにバブルを注入し続ける．

そして，サイクル C_6 には，I_1 が書き込んだ正しい値を読み出すことになる．そこでストールは解除され，次のサイクル C_7 からは，バブルではなく I_2 が，E ステージより下流へと進んでいくことになる．

この間，I_2 の後続にある命令 I_3 も先に進めないことに注意されたい．(実装によるが，典型的には)I_3 は F ステージにおいてフェッチされ続けることになる．

まとめると，パイプラインストールは，以下の 2 つの処理からなる．

1. 進むと不都合な命令があるステージとその上流のステージを停止する．

2. より下流のステージは停止せず，バブルを注入する．

前述したように，あるサイクルを縦に見れば，そのサイクルにどのステージにどの命令がいるかが分かる．例えば，サイクル C_5 では (D と E の間の隙間は無視して)F，D，E，M，W の各ステージにはそれぞれ，I_3，I_2，バブル，バブル，I_1 がいることが分かる．すなわち，このサイクルには，バブルのせいで，3 命令しか同時に処理されていない．

なお，図 7･3（上）のようにバブルをすべて描くと紙面をとるので，慣れてくれば同図下のように描いてもパイプラインの動作を把握できるようになる．

このように，パイプラインストールは，万能ではあるものの，ストールしたサイクル数 = 注入したバブルの数だけ，パイプラインの効率が低下する．そこで，以下では，構造，データ，制御の各ハザードに特化した，ストールサイクルの削減手法について説明する．

(2) 構造ハザード — ハーバードアーキテクチャ

構造ハザードは，ハードウェア資源の不足によるものなので，ハードウェアの追加によって比較的容易に解決することができる．

例えば，図 7･2 の例では，命令フェッチ (F) とデータアクセス (M) を同時にできるようにすればよい．古典的には，命令メモリとデータメモリを分離するアーキテクチャはハーバードアーキテクチャと呼ばれた．逆に，命令・データ統合（Unified）メモリを持つアーキテクチャは，プリンストンアーキテクチャと呼ばれた．最近のプロセッサでは，命令キャッシュとデータキャッシュを分離（Separate）することが普通であるが，これをハーバードアーキテクチャと呼ぶこともある．

(3) データハザード — バイパス

図 7･3 の例では，I_2 が E ステージで計算を行うサイクル C_4 の直前のサイクル C_3 において，I_1 は計算を済ませており，必要なデータはパイプラインのどこかには既に存在していることになる．これが I_2 の計算で利用できないのは，このデータがレジスタファイルに書き込まれてから読み出されるという経路をたどるためである．

したがって，レジスタファイルをバイパスする経路を設ければ，図 7･5 に示したようなストールをすることなしに，I_2 は I_1 の結果を得ることができる．

元の，レジスタファイルを通過する経路は，W ステージの終わりから D ステージの始まりへとデータを送ることになる．それに対して，このレジスタファイルのバイパスは，E ステージの終わりから，同じ E ステージの始まりへとデータを送ることになる．バイパスを使ったデータの転送は，データフォワーディングと呼ばれる．

E ステージで結果が得られる通常の命令の場合には，バイパスによってストールを解消することができる．一方で，M ステージで結果が得られるロード命令の場合には，バイパスだけでストールを解消することはできない．

(4) 制御ハザード — 遅延分岐

制御ハザードも，基本的には，データハザードと同様である．ただし，命令間の依存関係を媒介する値が異なる．**表 7･2** に，データ・制御ハザードにおいて命令間の依存関係を媒介する値をまとめる．制御ハザードにおいては，命令間の依存関係を媒介するのは，レジスタ値ではなく，次の命令の PC（Next PC）の値である．更に，媒介する値が生産される点と消

表 7･2　データ・制御ハザードを媒介する値

ハザード	媒介する値	生 産	消 費
データ	レジスタ値	E/M ステージの終わり	E ステージの始まり
制御	next PC	決まっていない	F ステージの始まり

費される点が異なる．データハザードの場合には，レジスタ値が生成される点と消費される点は E ステージの終わりと始まりに決まっている．一方，制御ハザードの場合，next PC の消費は F ステージの最初に決まっているものの，生成，すなわち，条件分岐命令の成立 / 不成立をどのステージで決定できるかは，命令セットアーキテクチャ（ISA）や実装によって異なる．

次章で述べる RISC 世代の ISA では，パイプラインのなるべく上流において決定できるよう，ISA のレベルから最適化が図られてきた．その結果，D ステージの終わりまでに分岐の成立 / 不成立から next PC の計算までを完了できる（あるいは，それと等しい効果を持つ）マシンが多い．それでもなお，図 7･4 に示したように，ハザードを完全に解決するには至らず，1 サイクルのストールが必要になる．

そこで，この世代の RISC マシンは遅延分岐によって ISA レベルでストールの解消を図った．遅延分岐では，条件分岐命令の次の命令の位置を遅延スロットとして，遅延スロットに置かれた命令は分岐の成立 / 不成立に関わらず実行されると ISA で定義される．すなわち，条件分岐命令の実行の効果の発生は，その直後（遅延スロット）からではなく，更にその次のスロットからへと遅延されるわけである．

条件分岐の成立 / 不成立に関わらず実行してよい命令を見つけることができない場合には，nop 命令で遅延スロットを埋める必要がある．最適化コンパイラは，遅延スロットに置くことのできる 1 命令を高い確率で探し出すことができた．

しかし，遅延スロットは，次章で述べるスーパスカラプロセッサにおいては，負の遺産となった．

この遅延の概念は，前述したロード命令後のデータ・ハザードによるストールにも適用することができる．本節で述べたすべての技術を組み合わせると，ストールのないパイプライン・マシンを構築することができる．その例である MIPS プロセッサの MIPS とは，当初は，Microprocessor without Interlocking Pipeline Stages の略であった．

7-2-5 CPI と IPC

ストールサイクルの多寡による命令パイプラインの効率性は，CPI（Cycles Per Instruction）によって表すことができる．CPI は，(クロックサイクルで測った) スループットの逆数である．CPI は，1 命令当たりにかかった平均サイクル数である．図 7･1 に示したような理想的な状態では，毎サイクル 1 命令の実行が完了している．逆に，1 命令当たりで見ると，(平均)1 サイクルごとに完了しているので，CPI は 1 となる．ハザードがある状況においては，ストールによって CPI は 1 より低下することになる．すなわち，命令パイプラインにおいては，CPI 1.0 を目指すことになる．

CPI の逆数は，IPC（Instructions Per Cycle）と呼ばれ，(クロックサイクルで測った) スループットそのものである．次章で述べる ILP プロセッサでは，1 サイクルに複数の命令を実行

可能であるため，CPI は 1.0 を切り得る．そこで，より直感的に理解可能な IPC を，性能指標として用いることが普通になった．例えば，CPI 0.33 より IPC 3.0 の方が，何が起こっているのか想像しやすいであろう．

■参考文献

1) Randal Bryant(著), “ David O’Hallaron(著), 五島正裕 (監訳) 他：コンピュータ・システム プログラマの視点から, ” 丸善出版, ISBN-10: 4621302019, 2019.

■6 群 - 4 編 - 7 章

7-3 CISC と RISC

（執筆者：五島正裕）[2018 年 12 月 受領]

CISC/RISC は, Complex/Reduced Instruction-Set Computer, 複合/縮小命令セットコンピュータの意である．RISC の発音は Risk と同じで，CISC はシスクと読む．

RISC の概念は，IBM の John Cocke のアイデアに影響を受けて始まり，California 大学 Berkley 校の David Patterson と，Stanford 大学の John Hennessy により大きく発展した．Patterson は，この新しい種類のマシンを RISC という名前を付け，同時に，既存のマシンには CISC という名前を付けて区別することにした[1)]．CISC はレトロニムの代表例となっている．

7-3-1 RISC の台頭

RISC の台頭の理由は以下の 3 つにまとめられる．

背景 最適化コンパイラによるコード生成

目的 命令パイプラインの実行効率の向上

手段 ISA の変更を含む最適化

(1) 背景：最適化コンパイラによるコード生成

CISC は，最も初期のコンピュータが正常進化したものである．当時は，ISA の進化，すなわち拡張が，コンピュータの進化だと考えられていた．新しい命令を追加していった結果, RISC が現れる 1980 年代初頭には，メインフレームやミニコンピュータの ISA は巨大なものとなっていた．

しかし，ISA の拡張がコンピュータの進化であるというは，人間がアセンブリ言語を用いてプログラミングを行うという前提においてであった．人間が高級言語を用いてプログラミングを行い，最適化コンパイラがアセンブリコードを生成するようになると，この事情は一変する．コンパイラにとって，ISA に追加されてきた高機能な命令を生成することは大変難しかった．これらの高機能な命令は，無用の長物と化したのである．

(2) 目的：命令パイプラインの実行効率の向上

一方で，半導体技術の進歩により，1980 年代初頭にはシングルチップの 32-bit マイクロプロセッサが視野に入ってきた．そこで，機能を最小限に削ってシングルチップに収めてみると，当時の，タンスや大型冷蔵庫サイズのメインフレームやミニコンピュータを凌駕する性能を示したのである[2)]．

これは，命令パイプラインの効率の違いによる．RISC が良いというよりは，当時の CISC の命令パイプラインの効率が低すぎたのである．後述する ISA の特徴のため，CISC の命令パイプラインは乱れまくっていた．

(3) 手段：ISA の変更を含む最適化

そこで RISC は，最適化コンパイラによるコード生成を背景として，命令パイプラインの実行効率の向上を目的として，発展していった．**表 7･3** に，CISC と RISC の違いをまとめ

表 7·3　CISC と RISC の違い．C，P 欄は，RISC のその特徴が，(C) 最適化コンパイラによるコード生成を背景としたもの か，(P) パイプライン実行の効率の向上を目的としたもの かを表す．また，S 欄は，(S) スーパスカラ実装への影響 を表す．

CISC	RISC	C	P	S
多数の高機能な命令 **可変長文字列に対してコード変換を行う命令など．**	**少数の基本的な命令** **単純で基本的な命令のみに絞った結果．数え方にもよるが，数十〜百命令程度．**	○	○	○
多入力のアドレシングモード **ベースレジスタ値+（スケーリング付き）インデクスレジスタ値+即値オフセットなど，二項加算より複雑な演算を行う．**	**アドレシングモードの概念がほぼない** **レジスタ値+即値オフセットが普通で，ロード命令に限りレジスタ値+レジスタ値が可能なものもあるが，何れにせよ二項加算に限られる．**	○		○
複雑なアドレシングモード **プッシュ/ポップ，ポストインクリメント/デクリメントなど，1 命令中で同じレジスタの参照と更新が行われる．**	**アドレシングモードの概念がほぼない** **同上**	○	◎	◎
特殊なレジスタがある **スタックポインタ，ベース/インデクスレジスタなど，特定の命令やアドレシングモードで特に用いられるレジスタがある．**	**特殊なレジスタはない** **（乗算の結果を格納するレジスタなどの例外を除き）特定の命令やアドレシングモードで使用されるレジスタはない．スタックポインタなどは，汎用レジスタのうちの 1 本を使用すると規約で定める．**	○		○
可変長命令 **多数の命令をエンコードするため，使用頻度の低い命令は多バイトを要する．先頭から順に見ていかないとレジスタ番号などが分からないこともある．**	**固定長命令と，少数の命令フォーマット** **通常 32b の固定長命令と，3〜6 種程度の命令フォーマット．固定長命令は next PC の計算を容易にする．少数の命令フォーマットは，デコード，特にレジスタ番号の抽出を容易にする．**		○	○
メモリオペランドに対して演算が可能 **ソース，デスティネーションの何れかにメモリオペランドをとれる．パイプライン設計が極めて困難．**	**ロード/ストアアーキテクチャ** **専らメモリにアクセスするロード命令/ストア命令と，その他の演算命令は別．ただし，それらの間で中間値を保存するためにも汎用レジスタが多めに必要となる．**		◎	◎
遅延分岐の概念はない **逆に言えば，パイプライン動作は ISA のレベルにおいて隠蔽されている．**	**遅延分岐** **パイプライン動作が ISA を通して上位に影響を及ぼしている．コンパイラが遅延スロットに置く命令を探すことを想定．**	○	◎	×
少数の汎用レジスタ **メモリオペランドに対する演算を用いれば，弊害は少ない．**	**多数の汎用レジスタ** **典型的には 32 本．(レジスタ数に余裕があるので) 必ず 0 が読めるゼロレジスタがある．**			
2 オペランド形式が標準的 **R1 += R2 の形式．R1 の元の値が必要な場合には，あらかじめレジスタ間でコピーを行う必要がある．メモリオペランドに対する演算を用いれば，弊害は少ない．**	**3 オペランド形式が標準的** **R3 = R1 + R2 の形式．32 本のレジスタを指定する 5b のオペランドフィールドを 3 つ，32b の命令中にエンコードすることは難しくない．**			

る．同表の C，P 欄は，RISC のその特徴が，(C) 最適化コンパイラによるコード生成を前提としたもの か，(P) パイプライン実行の効率の向上を目的としたもの かを表す．また，S 欄は，(S) スーパスカラ実装への影響 を表す．

RISC の最適化の特徴は，パイプライン動作という実装の都合に合わせて，ISA というインタフェースを直接的に変更した点にある．

7-3-2 RISC の功罪

(1) 単純化による進化

RISC が技術史的に注目されるのは，それが，技術の発展過程においては単純から複雑へと向かうという一般的な経験則に対する反例のように見えたためであろう．RISC の成功以降，単純化こそが進化であるとの言説も散見された．

しかしこれは，プログラマとマシンの間に最適化コンパイラというレイヤが挿入されるという，技術史上異例のことが起こったときのことだという点に注意する必要がある．レイヤが挿入された結果，最適化コンパイラとマシンのインタフェースである ISA の抽象度が下がったと考えれば，反例とまでは言えない．

実際，次章で述べるスーパスカラの時代においては，プロセッサは進化するほどに複雑さを増している．極めて複雑な ISA を持つ x86 プロセッサが RISC ISA のプロセッサの性能を凌駕するに至って，単純化による進化を謳うものはいなくなった．

(2) スーパスカラ実装に対する影響

RISC における最適化の特徴は，パイプライン動作という実装の都合に合わせて，ISA というインタフェースを直接的に変更した点にある．

表 7･3 の S 欄に示したように，RISC ISA の性質の多くは，スーパスカラ実装に対しても有利に働いている．しかし，特に，パイプライン動作に合わせて ISA を直接的に変更した分岐遅延は，対処困難というほどではないものの，実装を複雑にしている．すなわち，RISC はパイプライン実行というその時代の実装技術に対する過適応の側面があると言える．

もちろん，1980 年代においてスーパスカラ実装が一般的になると見通すことはできなかったであろう．しかし，インタフェースに関する一般的な原則はここでも確認することができる．すなわち，少なくとも長持ちさせるという観点からは，インタフェースは，何をするかのみを示すべきで，どのようにすべきかを示すべきではない．

■参考文献

1) D.A. Patterson and C.H. Sequin：“ RISC I: A Reduced Instruction Set VLSI Computer, ” Proc. Int‘l Symp. on Computer Architecture, pp. 443–457, 1981.
2) 五島正裕：“ 20 世紀の名著名論, ” 情報処理, vol. 46, no. 3, p. 317, 2005.（文献 1) の評論）.