として想定している．この場合には，前半期(15回)で本書の第1～4章を，後半期(15回)で第5，6章を，それぞれ講述・学習するとよい．第7章は実際のアーキテクチャ例や最近のトピックスを知るために選択的に講述・学習するか，大学院科目内容に回すとよい．もし，本講義科目に15回(半年)しか割り当てられないならば，第2，5，6章を中心に講述・学習し，第1，3，4，7章は選択的に講述・補習するとよい．

各章末の「演習課題」はオープンクエスチョン（公開質問状）であり，著者自身まだ解答を持っていないものを挙げた．これらの課題にコメントすることが「これからのコンピュータアーキテクチャ学の取り組むべき課題」と考えている．どうか，1題でも良いから考察を加え，そのレポートを著者宛(電子メールアドレス：shibayam@dj.kit.ac.jp）に送ってほしい．

現代のコンピュータメーカの世界は(やむを得ずかもしれないが)「性能第一主義」に陥っている．しかし，現代のコンピュータ技術を支えているのは「芸術」に近いユニークさやオリジナリティであることを思い出してほしい．今では商用マイクロプロセッサの性能競争の最先端に位置するRISCもスーパスカラアーキテクチャも，既製品や既成事実をひっくり返すことから発達してきた．本書の読者諸氏や演習課題のレポート提出者から，「世の中に1台しかないコンピュータアーキテクチャ」の設計者が生まれることを望みたい．

本学の同僚である平田博章氏には，丁寧な査読をして頂き，山のような誤りの指摘やコメントを頂いた．ここに深謝いたします．また，コンピュータサイエンスの専門コア科目として必要最小限の事項を1冊に詰めたために，本書は，教科書としては，我が国の出版界の常識から少々はずれた大部である．それが日の目を見るのは，オーム社出版部の英断である．ここに感謝します．

最後に，「ハードウェアとソフトウェアのいずれが欠けても良いアーキテクチャはできない」ことを実践的に教えてくれる我が家のアーキテクト：妻・真木子と4人の子供達・風野，すみれ，ののみ，蒼宙に，心から「おおきに」．

なお，著者のプロフィールは，公私ともども，

http://www.ark.dj.kit.ac.jp/~shibayam/

にある．どうぞ時間があれば立ち寄って下さい．

1997年早春　京都・松ケ崎にて

柴山　潔

ドオフポイントであるコンピュータアーキテクチャによって，そのコンピュータシステムの機能や性格が決まる．

マシン命令によって実現（提供）している機能レベルが，論理的構造を決めるソフトウェアと物理的構造を決めるハードウェアの役割分担の境界線である．したがって，「コンピュータシステムにおけるハードウェア機能とソフトウェア機能の分担方式を定めること」，すなわち「ハードウェア/ソフトウェアトレードオフを決めること」が**コンピュータアーキテクチャ設計**にあたる．「コンピュータアーキテクチャ設計」を**方式設計**ともいう．

たとえば，「命令セットアーキテクチャ設計」とは，「マシン語としてのマシン命令機能や形式，処理対象とするデータの種類や形式などを具体的に定めること」である．したがって，たとえば，「プロセッサアーキテクチャ設計」とは，「プロセッサ機能である制御方式や演算方式を実現するハードウェア構成方式を定めること」といえる．

コンピュータアーキテクチャ設計においては，このハードウェアとソフトウェアの機能分担方式に起因する種々の対立点がある．たとえば，コンピュータによる問題解決の高速化は，そのコンピュータの特定応用分野向けの専用化と考えられる．すなわち，コンピュータの高速化や専用化を追求すればするほど，そのコンピュータは柔軟性を失い，応用分野が局限されるようになる．これとは逆に，応用分野をできる限り広く考えてコンピュータの汎用化を図ると，今度は専用コンピュータの利点である高速性が損なわれることになる．

〔2〕 **コンピュータアーキテクトの役割**　コンピュータアーキテクチャ設計とは，「設計目標の実現のためにコンピュータにおける種々の対立する機能のバランスをとってやること」ともいえる．

コンピュータアーキテクチャの設計者，すなわち**コンピュータアーキテクト**(computer architect)は，アーキテクチャ設計に要する期間を考慮に入れてハードウェア技術やソフトウェア技術の現状と将来および対象とする応用の処理構造を的確に分析し，どのようなハードウェアおよびソフトウェアの組合わせによって設計目標とするコンピュータの機能を実現するかを決定しなければならない．

〔3〕 **コンピュータアーキテクチャを決定する要因**　コンピュータアーキテクチャは，4.1.2 項で述べるように，ハードウェア技術やソフトウェア技術の進

処理された結果データの格納先を示す．演算命令の場合は，原則として，デスチネーションオペランドは1つでよい．

〔3〕 **オペランドによって指定されるメモリ** 1.2.1項でも述べたように，マシン命令のオペランドで指定される命令やデータの格納元や格納先となるメモリには，図2.3に示すように，次の2種類がある．これらはそれぞれの特性に応じて使い分けられ，ノイマン型コンピュータの代表的な特徴となっている．

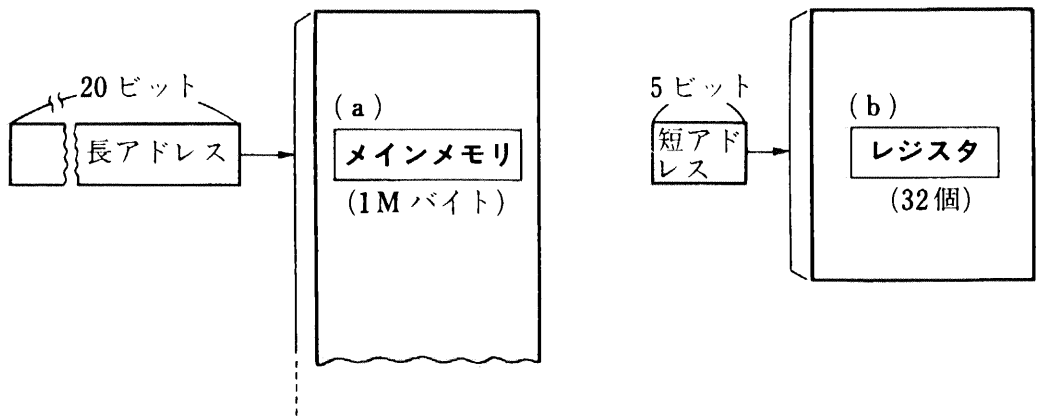

図 2.3 オペランド対象のメモリ

(a) **メインメモリ**：プロセッサ外部に置かれる（文字通りの）主要なメモリである．(b)に比べると，相対的に大容量であるが低速動作である．大容量のメインメモリの格納場所を識別するためには，オペランドとして長いアドレスビットが必要となる．たとえば，1Mバイトのアドレスを一意に直接識別するためにはオペランド長として20ビットが必要である．このため，アドレス修飾（2.2.3項〔2〕参照）が必須となる．

(b) **レジスタ**：プロセッサ内部にあり，(a)に比べると，相対的に小容量であるが高速動作する．少数のレジスタ番号を識別できればよいので，オペランド長は短くて済む．たとえば，32個のレジスタを識別するためには5ビット長のオペランドで済む．アドレス空間はメインメモリに比べると格段に小さく，また高速アクセスが必要とされるので，レジスタだけへのアクセスは，原則として，アドレス修飾を行わずに，オペランドとして直接にレジスタ番号を指定することによって行われる．

〔4〕 **オペランド数による命令形式の分類** マシン命令はそれが持つオペランドの個数によって分類することができる．命令種類の中でも特に典型的な2項演算命令では，1命令の実行に2つのソースオペランドと1つのデスチネー

みが発生すると実行中のプロセスが一時中断されて，**割込み処理**が実行される．

割込み処理機能は**割込みハンドラ**（interrupt handler）ともいい，その処理の大半がOSの機能として実現される．割込みの発生による割込み処理の具体的な処理手順はおおよそ次のようになる．

(1) 割込みの受付

(2) 割込み要因の識別

(3) プロセスが共用するハードウェア資源の内容の退避

(4) 割込み要因ごとの割込み処理

(5) 退避していたハードウェア資源の内容の回復

(6) 割込み処理の完了

割込み処理中は原則として割込み禁止とし，他の割込みが発生しても現在の割込み処理が終了するまで，新たな割込みの受付けは待たされる．この原則が覆る場合や複数の割込みが同様のタイミングで生じた場合の処理方式については，〔2〕で詳述する．

また，割込み処理手順は，実際には，制御アーキテクチャとしてハードウェア機構（マイクロプログラムを含む）/OS/ユーザプログラム（ソフトウェア）の機能分担によって実現され，トレードオフがある．たとえば，(1)はソフトウェアでは実現不可能なのでハードウェア機構が，(2)は高速処理を要求されるのでハードウェア機構が，(3)(5)(6)はハードウェア機構とユーザプログラムとの橋渡しであるのでOSが，(4)は割込み要因ごとに処理内容が異なるのでOSまたはユーザプログラムが，それぞれ分担する方法が考えられる．

〔2〕 **多重レベル割込み**　2.3.1項〔2〕で述べたように，割込みにはその要因によって多種多様なものがある．また，これらは独立した事象であるものが大半であり，同時に発生する可能性もある．プロセッサを1つしか持たないノイマン型コンピュータにおいては，複数の割込みについても逐次的に処理せざるをえない．そのための機構が**多重レベル割込み**（優先度付き割込み）機構である．多重レベル割込み機構では，割込み要因ごとに割込み処理の優先度を付与し，その高低（緊急性）によって複数の割込みを順序付けて処理する．すなわち，多重レベル割込み機構は

- ある割込み処理時には，それよりも優先度の低い割込みは禁止される
- 一方，ある割込み処理時にそれよりも優先度の高い割込みが発生した場合

の両方の措置とも行う必要がある．(2)の「再開が可能であること」を**再開可能**(restartable)という．割り込まれて命令無効の措置をされた命令は再開可能でなければならないのが普通である．割り込まれた命令に対する(1)(2)の措置の実行可能性によって割込みを次のように大別することができる．

(a) **正確な割込み**(precise interrupt)：回復が正しく行われ，かつ，割り込まれた命令が再開可能であることを保証できる割込みである．「正確な割込み」とは

- 割り込まれた（割込みが発生した，割込みを発生させた）命令に先行するすべての命令の実行は完了する
- 割り込まれた（割込みが発生した，割込みを発生させた）命令とそれに接続するすべての命令は未実行であり，もう一度最初から再開可能である

の両要件とも満たしている割込みといえる．

(b) **不正確な割込み**(imprecise interrupt)：何らかの理由で，マシン状態の回復が不十分であったり再開可能であることを保証できない割込みである．割込みが不正確になる理由としては，① 高度なパイプライン処理（5.2節参照）など複雑な順序制御を行っていて，一部のハードウェア資源の回復が不可能である；② 命令実行の高速化のために，意図的に（マシン命令仕様として），一部のハードウェア資源の退避を省略して，それを回復しない；などがある．

マシン命令フェッチ時やオペランドデータフェッチ時のページフォールト（6.2.4項〔2〕参照）は正確な割込みを要件とする要因の典型例である．ページフォールトは実行（演算）しようとした命令やデータがメインメモリにない場合に発生するもので，その命令機能そのものが直接的な割込み要因となったわけではない．したがって，ページフォールトが発生した（割り込まれた）命令は再開可能でなければならない．そのほかにも，「IEEE* 規格で正確な割込みであることを要求しているIEEE標準の浮動小数点数演算における命令実行例外」などもある．

命令実行の高速化のためにハードウェア資源の時間的/空間的多重化を行っ

---

* IEEE（Institute of Electrical and Electronics Engineers）：電気および電子技術に関する研究を目的とする国際学会．さまざまな規格を提案している．ここでいう「IEEE規格」は浮動小数点数表現の標準化のためにIEEEが提案した規格．

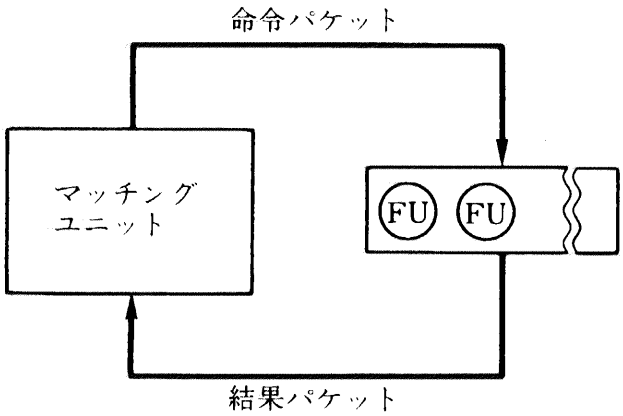

図 4.12 データフローマシンの動作原理

れる.

(2) 実行可能な命令は OP コードやオペランドなどをひとまとめにした転送単位の**命令パケット**（packet）として FU 群に送出される.

(3) FU で実行された命令パケットは演算結果としてそれを使う命令をひとまとめにした**結果パケット**となって再びマッチングユニットにフィードバックされる.

〔4〕 **種々のデータフローマシン** データフローマシンは，その命令コード(データフローグラフ)がリエントラント*1かどうかで，次の2種類に大別できる.

(a) **静的データフローマシン**：命令コード（データフローグラフ）のスケジューリングはコンパイル時に行われるのでリエントラントではない．ハードウェア機構は単純であるが，再帰呼出し*2の実現のためには命令コードのコピーが必要となり，それがオーバヘッドとなる.

(b) **動的データフローマシン**：共用される命令コードは呼出しごとにタグ（**色付きトークン**（colored token）という）などで識別されるので，そのコードはリエントラントである．ループ(繰返し)，再帰呼出しなどの種々の制御構造を実現できる．一方で，ハードウェア規模は大きくなるという短所がある.

また，データフローモデルをいずれの処理機能レベルに適用するかによって，

---

*1 リエントラント（reentrant：再入可能）：2つ以上のプロセスによって共有されているルーチンが一時中断後も同じ状態で再開可能であること.

*2 再帰呼出し（recursive call）：手続きやサブルーチン中で自分自身を呼び出す制御構造.

(c)　**特殊アーキテクチャ支援**：① コンパイルの対象プロセッサアーキテクチャがパイプライン処理や命令レベル並列処理をとっている；② 対象コンピュータがベクトルコンピュータや並列コンピュータなどである；という場合には，コンパイラ支援機能をそれぞれの特殊なアーキテクチャに依存して特化できる．これらについては，それぞれのコンパイラ支援アーキテクチャについて説明する項で詳述する．

本節の以下の項では，コンパイルの対象が汎用コンピュータであるか，またはコンパイラ支援機能そのものに汎用性がある場合について詳述する．

〔4〕　**インタプリタ支援機能**　前の〔3〕の(b)で述べたプログラミング支援機能の代表例はデバッグ支援である．プログラミング過程におけるデバックを支援するためには，コンパイラ機能よりもむしろプログラミング言語の**インタプリタ**機能が必須となる．このインタプリタ機能を支援するアーキテクチャとして，たとえば，次のようなものがある．

- **仮想ハードウェア**：プロセッサやメモリなどの物理的ハードウェア資源を仮想化して論理的に用意する．特に，レジスタをメインメモリ内に持つ**ソフトウェアレジスタ**がしばしば利用される．
- **ハードウェアプロファイラ**（profiler）：デバッグ時に必要となる実行環境の情報を収集したり保持するハードウェア機構である．分岐予測機能を支援する分岐履歴テーブル（5.4.4 項〔4〕参照）などはハードウェアプロファイラの一種である．

これらはいずれもインタプリタの高速化を目指しており，これを押し進めると高級言語マシンアーキテクチャ（4.2.6 項参照）に至る．

4.3.2　コード最適化におけるコンパイラとハードウェアとのトレードオフ

〔1〕　**コード最適化**　コンパイル過程は，① 字句解析；② 構文解析；③ 意味解析；④ コード生成；⑤ コード最適化；に分かれ，普通この順で処理が進められ，最終的に対象コンピュータのマシン命令列（オブジェクトコード）が得られる．

①～③のプログラム解析は対象プログラミング言語に依存するが対象コンピュータアーキテクチャには依存しない処理であるのに対し，④と⑤は対象プログラミング言語には依存しないが対象コンピュータアーキテクチャに依存する

ンメモリ）の動作速度より数倍（たとえば，100 MHz クロックのプロセッサの1クロックサイクル時間が10 ns で，大容量 DRAM のアドレス指定も含めたメモリアクセス時間は数十～百 ns 程度）速い．したがって，プロセッサの動作中にメモリアクセスがあると，プロセッサの動作は中断される．ノイマン型コンピュータアーキテクチャの設計では，このプロセッサ動作の中断時間（メモリアクセス待ち時間）に対して，① 除去；② 隠蔽（見かけ上除去）；③ 短縮；のいずれかの方法によって対処することが必要となる．

具体的には，次の例のような種々のアーキテクチャ技術が開発されている．

- **レジスタ**：レジスタはプロセッサと同程度の速度で動作する超高速一時メモリである．処理途中のデータをメインメモリではなくこのレジスタに置くと，（データに対する）メモリアクセスそのものをなくすことができるので，①にあたる．
- **ロード/ストアパイプライン**：メインメモリへのアクセスを演算用とは別のパイプラインで行う方法で，②にあたる．（5.2.3項で詳述）
- **キャッシュメモリ**：③の対処方法であり，命令やデータの一部をメインメモリよりも高速動作するキャッシュという一時メモリに置き，メモリアクセス時間を短縮する．（6.3 節で詳述）

現代のノイマン型コンピュータアーキテクチャとしては，これらの技術は必須となっている．

〔2〕 **プロセッサの高機能化**　プロセッサの高機能化とは

(a)　高速化（高性能化）

(b)　広い適用性の獲得

の2つである．(a) は主としてハードウェアによる機能分担によって，(b) は主としてソフトウェアあるいはファームウェアによる機能分担によって，それぞれ実現できる．したがって，(a) と (b) はトレードオフとなる．また，(b) をファームウェアによって実現する場合は，性能と専用化範囲の拡大との両立を狙っており，ファームウェアによってこのトレードオフの解決を図っている．

さらに，プロセッサをハードウェア機構によって高速化する際の要点は，① 制御機構の高速化；② 演算機構（ALU）の高速化；とに細分できる．

〔3〕 **実行順序制御の高速化**　〔2〕の① は「実行順序制御の高速化」とみなせる．これはマシン命令当りのクロックサイクル数である CPI (3.3.2 項参照)

要する時間 T_P〔秒〕は，実行される ステージ 総数が $(I+D-1)$ 個であることから

$$T_P=(I+D-1)\times P \tag{5.1}$$

で表される．ただし，1ステージは1ピッチで処理されるものとする．

すなわち，最初の$(D-1)$個のステージ(**スタートアップ時間**(start-up time)という)の経過後，次の $(I-D+1)$ 個のステージの間はすべてのステージが常に並列に実行でき（定常期），最後に $(D-1)$ 個のステージ（**クリーンアップ時間**（clean-up time）という）経過後全命令の処理（実行）が終了する．

パイプライン処理を行わない場合の I 個の命令を実行するのに要する時間 T_N〔秒〕は

$$T_N=I\times D\times P \tag{5.2}$$

である．ここで，深さに比べて実行命令総数が格段に大きい $(I\gg D)$ の場合には，式（5.1）で D が無視できて

$$T_P\approx I\times P \tag{5.3}$$

となり，式（5.2）と比較すると

$$T_P\approx\frac{T_N}{D} \tag{5.4}$$

が成立し，パイプライン処理によって約 D 倍の処理性能が得られることになる．

たとえば，$D=4, I=100, P=20$〔ns〕とすると，$T_P=2.06$〔マイクロ(micro-，μ：$\times10^{-6}$) 秒〕となる．

前の〔5〕で述べたパイプラインの深さと命令実行サイクル時間とがトレードオフとなることは，式（5.1）における D と T_P との関係からも説明できる（D が増えると T_P も大きくなり性能が低下する）．

さらに，式（5.1）は〔6〕で述べた深さ D とピッチ P とがトレードオフとなることも示している．すなわち，パイプライン性能 T_P を一定とする条件で，パイプラインを深くするとピッチは短くなり(ステージ当りの処理量が減る)，逆に，パイプラインを浅くするとピッチは大きくなる（ステージ当りの処理量が増える）．ただし，パイプライン処理は，式（5.1）において D を無視できるくらい多数の命令を投入しなければ，有効に機能しないので，一般的には，パイプラインピッチのほうが性能向上には効く．この点を利用しているのがスーパ

パイプラインである.

〔8〕 **パイプライン処理の理論的 CPI** また,パイプライン処理を行うコンピュータの命令当りのクロックサイクル数 CPI_P は,T_P を求めた場合と同様に実行されるステージ総数が $(I+D-1)$ 個であることから,1ステージ(ピッチ)が1クロックで実行できるならば

$$CPI_P=\frac{I+D-1}{I} \tag{5.5}$$

となる.ここで,D を無視できるくらい投入命令数 I が大きくなる $(I \gg D)$ と

$$CPI_P \approx 1 \tag{5.6}$$

となる.パイプライン処理による性能向上とは,理論的には,「CPI を“1”に近づけること」といえる.

前の〔7〕での例と同様に,たとえば,$D=4$, $I=100$ とすると,$CPI_P=1.03$ となる.

しかし,実際には,次の 5.2.2 項で詳述するパイプラインハザードなどによって理論的な性能向上の達成は阻害される.

〔9〕 **パイプライン処理のハードウェア機構** パイプライン処理機能にもハードウェア/ソフトウェアトレードオフがあるが,プロセッサの高速化を目標としているので,十分なハードウェア機構による支援が必要である.

パイプライン処理のためには,図 5.10 に示すハードウェア構成例のように,各ステージ間にそれぞれが授受する情報の一時格納用レジスタとして,バッファやラッチ(5.1.3 項〔5〕の脚注参照)というハードウェア機構が必須となる.

〔10〕 **パイプライン処理と割込み** 複数のマシン命令がステージ単位で時間的/空間的に多重化されているので,パイプライン処理における割込み(2.3 節で詳述)の制御は,パイプライン処理を行わない場合よりも,格段に難しくなる.特に,(命令)パイプライン処理における割込みの制御は,次のような理由で,複雑で難しい.

- 割込み発生時にパイプラインに投入されている(実行されている)マシン命令は複数であり,マシン状態数が格段に多く,複雑に入り組んでいる.
- 割込み発生時にパイプラインに投入されている複数個のマシン命令の処理ステージは相異なり,それぞれの命令実行サイクルが途中の種々のステージで中断されるので,各命令機能の割込み処理後の措置方法(2.3.2 項〔4〕

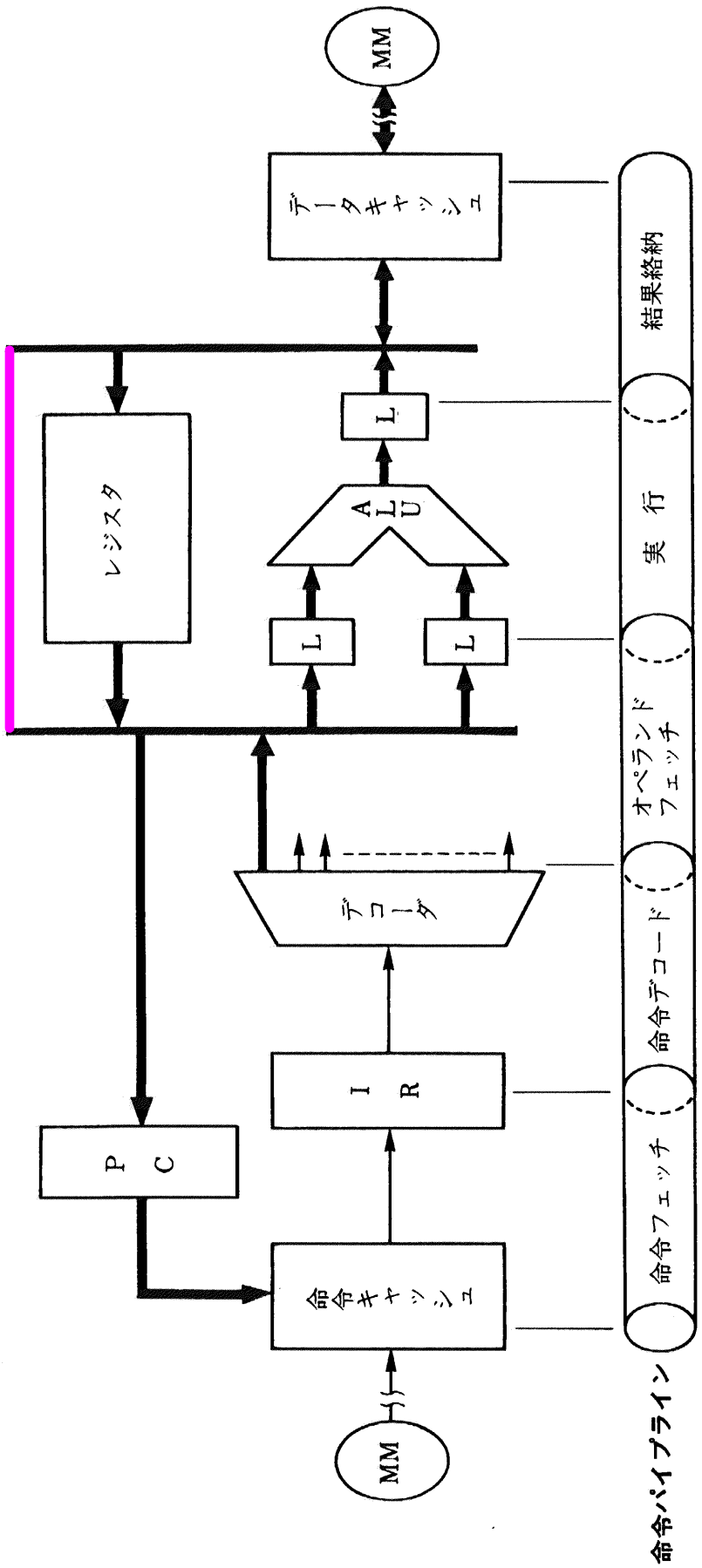

図 5.10 パイプライン処理の命令実行サイクルとハードウェア構成例

ードが生じる．これを**ライトアフタリードハザード**（write after read hazard：**WAR ハザード**）という．WAR ハザードは，先行命令がまだ読み出していないデータを後続命令が上書きして消そうとすることによって生じる．

（c） **出力依存**　「先行命令 I_X が書き込む資源を後続命令 I_Y が書込みに使用する」依存である．当該命令と出力依存となる先行命令がデータを書き込む前に，当該命令が同じ資源にデータを書き込もうとするとハザードが生じる．これを**ライトアフタライトハザード**（write after write hazard：**WAW ハザード**）という．WAW ハザードは，同一資源を書込み対象とする先行命令と後続命令の書込み順序が逆転し誤りとなる（新しいデータに古いデータが上書きされる）ことによって生じる．

5.2.3 パイプライン処理の最適化

〔1〕 **最適化の目的と手法**　パイプライン処理の最適化とは，適切なハードウェア/ソフトウェアトレードオフによってパイプライン処理の高速化を図ることである．

特に，パイプライン処理においては，前の 5.2.2 項で述べた種々のハザードによって，式 (5.4) に示した理論的な性能向上が阻害される．このハザードの発生を防止すれば，パイプライン処理がコンピュータ性能の向上に効果的に貢献する．

パイプライン処理の最適化手法は次の 2 つに大別できる．

(a) **ハードウェア機構の多重化**：パイプライン機構を空間的あるいは時間的に強化（多重化）してパイプライン処理を高速化する．

(b) **パイプラインハザードの解消**：ハザードの種類ごとにさまざまな解消方法があり，それらにはハードウェア/ソフトウェアトレードオフがある．具体的な方法については，〔7〕で詳述する．

〔2〕 **ハードウェア機構の多重化による構造ハザードの解消**　ハードウェア資源へのアクセス競合による構造ハザードを避けるために，パイプラインをハードウェアによって多重化（太く）する手法である．たとえば，次のような方法がある．

① メインメモリ（キャッシュ）やレジスタのアクセスポートを多重化して，複数オペランドデータを同時に読み出せるようにする．

② **並列演算**：ALUを多種類・複数個設けて，これらを並列に演算させる．スーパスカラアーキテクチャ（〔3〕参照）を実現する部分機構ともなる．(5.1.1項〔4〕参照)

③ **ロード/ストアパイプライン** (load/store pipeline)：メインメモリへのアクセス（ロードは読出し，ストアは書込み）処理を他の演算命令の実行用パイプラインとは別の独立したパイプラインで行う．(5.4.5項〔1〕で詳述)

④ 命令フェッチステージと命令実行（オペランドフェッチを含む）ステージとを別々のパイプラインとする．

⑤ **ライトバッファ** (write buffer)：キャッシュ（6.3節参照）への書込みアクセスによってメインメモリへの書込みアクセスも生じる場合に，その待ち時間を吸収するために設けるバッファ（一時格納装置）である．(5.4.5項〔3〕参照)

〔3〕 **スーパスカラとスーパパイプライン**　ハードウェア機構のうちパイプラインそのものを多重化することによって高速化を図っているアーキテクチャとしては，特に，次の2つが代表的である．

- **スーパスカラ** (superscalar)：複数の命令フェッチ/デコード機構やALUを独立したパイプラインとして並列動作させることによって，パイプラインの空間的な多重度を上げる高速化手法である．(5.3.2項参照)
- **スーパパイプライン** (superpipeline)：パイプラインを深くすることによって時間的な多重度を上げる高速化手法である．(〔4〕参照)

これらはいずれもかなりのハードウェア機構の支援が必要となる．

〔4〕 **スーパパイプラインアーキテクチャの特徴**　**スーパパイプライン**は，図5.9に示したように，単純なパイプライン処理における1ステージを複数(たとえば，“2”）に分割すること（これは「半ピッチにすること」にあたる）によって時間的多重度を上げるパイプライン高速化手法である．5.2.1項〔7〕で示したように，ピッチが短くなるとパイプライン性能は向上する．

単純な命令パイプライン処理と比べた場合の，スーパパイプライン処理のアーキテクチャ上の特徴としては

- 普通のパイプライン処理よりもさらに細かく単純な操作がステージの処理機能となるので適応性が増す

となる．m を**スーパスカラ度**という．

5.2.3 項〔5〕で比較したとおり，パイプラインピッチを n 分の 1 にしたスーパパイプライン処理性能は元のパイプライン処理性能の n 倍であるから，$n=m$ の場合のスーパパイプライン処理とスーパスカラ処理は，（投入命令数がパイプラインの深さよりも格段に大きい場合）ほぼ同程度の性能向上が期待できることになる．たとえば，あるスカラ処理のピッチを半分にしたスーパパイプライン処理とそのスカラ処理のパイプライン 2 本によるスーパスカラ処理とは十分多数の投入命令の実行において，いずれも元のスカラ処理の約 2 倍の性能向上を見込めることになる．

たとえば，5.2.1 項〔7〕と同じ設定で，$m=2$，$D=4$，$I=100$，$P_{SS}=20$〔ns〕とすると，$T_{SS}=1.03$〔ns〕となり，同じ条件のスカラ処理性能 $T_P=2.06$〔ns〕の 2 倍の性能が期待できる．

〔3〕 **スーパスカラ処理の理論的 CPI**　次に，スーパスカラ処理の CPI_{SS} について，理論的に考察してみよう．

5.2.1 項〔8〕と同様に，パイプラインの深さを D，1本のパイプラインに投入する命令数を I とすると，図 5.23 から明らかなように

$$CPI_{SS}=\frac{CPI_P}{m}=\frac{I+D-1}{I\times m} \tag{5.15}$$

となる（ただし，$m\geq 2$）．そして，$I\gg D$ の場合は

$$CPI_{SS}\approx\frac{1}{m}<1 \tag{5.16}$$

となる．

スーパスカラは CPI を“1”以下にすることによって処理性能の高速化を図る手法といえる．

たとえば，〔2〕と同じ設定条件のスーパスカラ処理では，$CPI_{SS}=0.515$ となる．（参考：$CPI_P=1.03$，$CPI_{SP}=1.07$）

〔4〕 **スーパスカラ処理のハードウェア機構**　スーパスカラ処理は多重命令パイプライン処理であるが，それを実現するハードウェアとしては，単に，パイプラインを複数装備するだけではなく，そのほかにも次のようなハードウェア機構が必須となる．

（a） **命令発行機構**　命令デコードステージと命令実行ステージとの間に

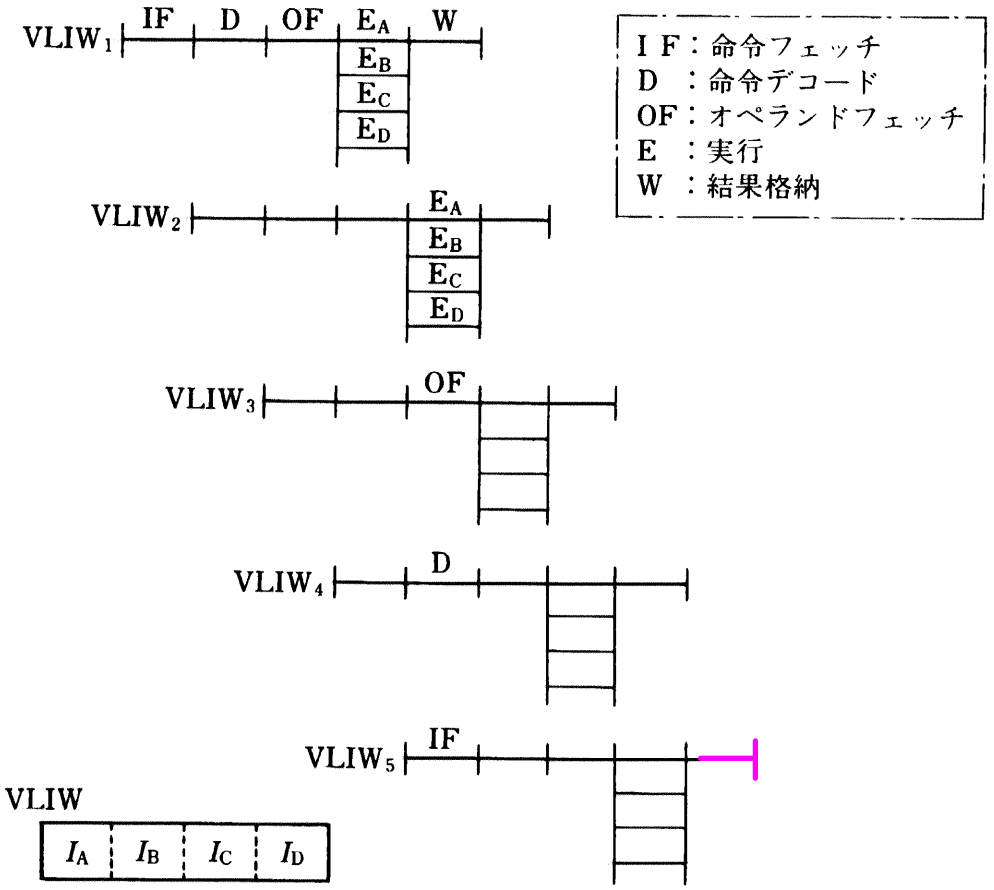

図 5.31 VLIW 処理の基本原理

ン処理の高速化を図る命令レベル並列処理の一種である.

マイクロプログラム制御コンピュータにおける代表的なマイクロ命令形式は「水平型マイクロ命令形式」といい,このVLIWと同じように各種のFUやハードウェア機構と固定的に対応する命令フィールドを連結した形式である.VLIWは水平型マイクロ命令形式（マイクロアーキテクチャ）をマシン命令形式（マシン命令アーキテクチャ）に適用したものと考えてよい.

〔2〕 **VLIWアーキテクチャの特徴** VLIWと5.3.2項で述べたスーパスカラはいずれも代表的な命令レベル並列処理アーキテクチャである.スーパスカラアーキテクチャとの比較の観点から,VLIWアーキテクチャの特徴をまとめてみよう.（表5.1参照）

(a) 単一命令ワード（ただし,VLIW）をフェッチ・デコードする.

(b) パイプラインの一部ステージ（主として,命令実行ステージ）だけを多重化（並列化）する.

(c) 既存スカラのオブジェクトコード（コンパイラが生成したコード）をそ

位置付けると，図6.2に示したように，(プロセッサ) ⇔ キャッシュ ⇔ メインメモリ ⇔ ファイル装置 ⇔ (ユーザ) というようになる．すなわち

- **キャッシュ**：プロセッサが直接情報を授受するメモリ階層であり，アクセス速度をメモリ性能として重視する
- **ファイル装置**：ユーザ（人間）が直接（意識的に）情報を授受するメモリ階層であり，容量をメモリ性能として重視する
- **メインメモリ**：キャッシュとファイル装置の中間にあり，キャッシュより大きい容量とファイル装置よりも高速のアクセス速度との両立を要求される

となり，これがノイマン型コンピュータのメモリアーキテクチャの原理となっている．

現代のコンピュータシステムにおけるレジスタ，キャッシュ，メインメモリ，ファイル装置の代表的な性能（容量とアクセス速度）例については表6.1に掲げてある．

表 6.1　メモリ階層の実例とその代表的性能例

メモリ階層	容量〔バイト〕	アクセス速度〔ns〕
レジスタ	64~1K	1~5
キャッシュ	64K~1M	3~10
メインメモリ	16M~16G	50~400
ファイル装置(ハードディスク)	1G~64G	5000~20000 **×1000**

〔3〕 **種々のメモリ階層例**　現代のノイマン型コンピュータのメモリ階層の実例を次に列挙する．この例では，(1) → (11) の順で容量性能が良くなり，逆に，(11) → (1) の順でアクセス速度性能が良くなる．

(1) **レジスタ**：処理（演算）の直前/直後のデータを一時的に格納するハードウェア資源であり，プロセッサ内部のALU/FUに最近接した最高速のメモリ階層である．現代のノイマン型コンピュータを特徴付けている．(2.2.2項〔3〕参照)

(2) **キャッシュ**（**1次キャッシュ**，本項〔1〕および6.3節で詳述）

(3) **中間キャッシュ**（**2次キャッシュ**，**ワークメモリ**，6.3.7項〔4〕参照）：キャッシュとメインメモリとの中間に置かれ，両者のアクセス時間差を吸収する．メインフレームコンピュータやスーパコンピュータなどの大型コ

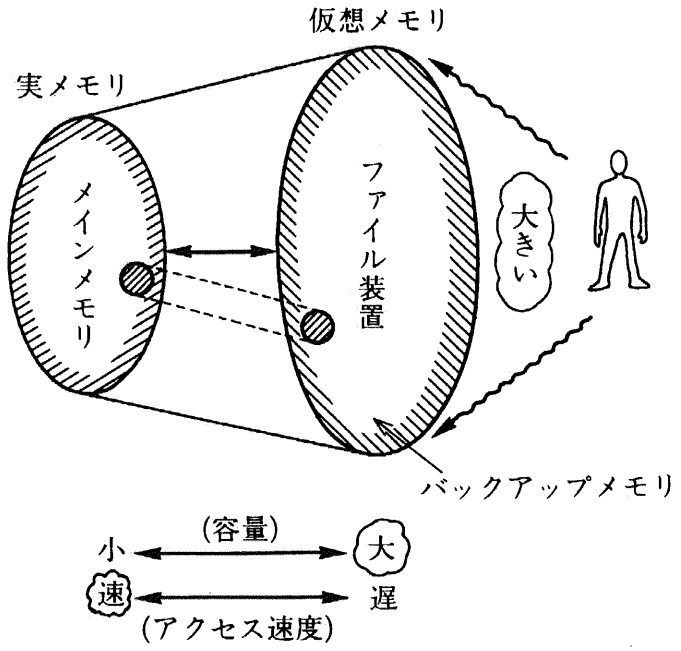

図 6.4　仮想メモリ

メインメモリ内の対象データのアドレスに関して

●マシン命令（プログラム）によって指定する**論理アドレス（仮想アドレス）**

●実際のメインメモリ素子に付けられている**物理アドレス（実アドレス）**

をおのおの独立させる．そして，ユーザ（プログラマ）やコンパイラには，メインメモリ空間として論理アドレス空間(ファイル装置容量の空間規模がある)を仮想的に見せる．この論理アドレス空間を構成するメモリ（実体はファイル装置）を**仮想メモリ**という．これに対して，プロセッサなどのハードウェア機構が直接アクセスする物理アドレス空間である実際のメインメモリを**実メモリ**(real memory) という．

仮想メモリ機構は，論理アドレス空間（仮想アドレス空間）と物理アドレス空間（実アドレス空間）の対応関係（**アドレスマッピング**(address mapping) という）を管理/制御する機能を担う．

〔3〕 **仮想メモリの効果**　仮想メモリはハードウェア/ソフトウェアトレードオフに対して次のような効果を与える．

(a) **実アドレス空間サイズの制約を事実上なくしている．**

●コンパイラは，メインメモリの物理的な実装容量（実アドレス空間サイズ）による制約を受けることなく，マシン命令中に論理的なオペランドアドレス（仮想アドレス）を置くことができる．

御する.

ページセグメンテーションの仮想アドレス V は仮想セグメント番号 v_s と仮想ページ番号 v_p とページ内相対アドレス（オフセット）o から成る. また, アドレス変換テーブルとして, 単一のセグメントテーブルと複数のページテーブルを用いる. セグメントテーブルにはページテーブル番号が格納されている.

ページセグメンテーションにおけるアドレス変換の手順は次のようになる. (図 6.7 参照)

(1) セグメントテーブルを仮想セグメント番号 v_s で検索してページテーブル番号 p を得る.

(2) (1) で得られたページテーブルを仮想ページ番号 v_p で検索し, 実ページ番号 r_p を得る.

(3) 実ページ番号（実ページの開始アドレス）とオフセットによって実アドレス R を求める.

ページセグメンテーションの長所は次のとおりであり, ページングとセグメンテーションの各長所の両立を図っている.

① セグメントという論理的なまとまりのある単位で最初のマッピングがとられるのでその単位で論理的意味が活用できる（〔3〕の ① ②）.

② セグメントを決定したあとは通常のページングによるので, 実メモリの管理が簡単で, またその利用効率も優れている（〔2〕の①②）.

一方, 欠点は次のとおりである.

③ セグメントサイズに比例してページテーブルが大きくなる.

④ アドレス変換時にセグメントテーブルとページテーブルの 2 つのテーブルを引く必要があり, 変換時間が長くなる.

ページングとセグメンテーションの両者の特長を活用できるという点で, 現代のノイマン型コンピュータの大半がこのページセグメンテーションを採用している. ページセグメンテーションの短所 ③ ④ の対策については, 6.2.5 項で述べる.

6.2.4 ブロック置換

〔1〕 **ブロック置換機構** 6.2.2 項〔3〕で述べたように, アクセス対象の命令やデータが実メモリ（メインメモリ）にない場合にはバックアップメモリで

ック置換アルゴリズムあるいは**ページ（セグメント）置換アルゴリズム**という．以下の説明では，ページ置換アルゴリズムを代表例にとる．

ページ置換においては参照局所性を活用する．すなわち，ページ置換アルゴリズムは「参照局所性が高いページはそのまま実メモリ中に残し，それが最も低いページを追い出す」とするのが適切である．ページ置換アルゴリズムは参照局所性の高低を判断する（定性的あるいは定量的）指標を定めている．

現代のノイマン型コンピュータで利用されている主要ページ置換アルゴリズムをあげてみよう．

(a) **LRU** (least recently used)：最終アクセス時刻が最も古いページを追い出す．時間的参照局所性を活用しているが，アクセスの履歴を保存・管理する機能などのために置換機構は複雑になる．

(b) **ランダム** (random)：無作為に追い出すページを決める．実現が簡単であるが，参照局所性はまったく考慮されない．

(c) **FIFO** (first in first out)：一番最初にメインメモリに入ったページを最初に追い出す．これも「時間的参照局所性を利用している」といえるが，LRUほど（「参照を正確にチェックしているわけではない」という意味で）参照局所性を十分には反映できない．置換機構の実装に要するコストは(a)と(b)との中間である．

(d) **ワーキングセット** (working set)（6.2.5項〔5〕で詳述）

ページ置換アルゴリズムの評価指標としては，ある一定時間内でのページフォールトの発生率（**ページフォールト率**という）がある．

ページフォールト率は，ページ置換アルゴリズムよりもむしろ，対象とするプログラムやデータの参照局所性によって左右されるので，いずれの問題に対してもページフォールト率が低くなるようなページ置換アルゴリズムを見つけるのは困難である．したがって，仮想メモリ機構におけるハードウェア/ソフトウェアトレードオフの設計の立場からは，ページ置換アルゴリズムの実装（次の6.2.5項〔7〕参照）に要するコストなどが重視される．

6.2.5 仮想メモリの要素技術

〔1〕 **仮想メモリ機構の高速化** 仮想メモリはユーザやコンパイラ（厳密に

詰め直す；③循環バッファ（キュー）の出口にある実ページ番号のページをスワップアウトする；という手順，FIFOでは，①循環バッファ（キュー）の出口にある実ページ番号のページをスワップアウトする；②スワップアウトされた実ページ番号を循環バッファから抜き出し，それを循環バッファ（キュー）の入口から詰め直す；という手順，にそれぞれよる．循環バッファの操作は複雑である．特にLRUでは，循環バッファの途中からの抽出と詰直し操作が参照のたびに必要となり，また，実ページ数が多くなるとその操作に時間がかかる．FIFOでは，ブロック置換アルゴリズムがFIFO構造をもとにしており，スワップアウトする実ページ番号を循環させてキューに詰め直すだけで①～③は一度に行うことができる．したがって，FIFOでの循環バッファ操作の負荷は実ページ数と無関係で同一である．

〔8〕 **バックアップメモリの更新** メインメモリはファイル装置の一部のコピーであるので，スワップアウトされるブロックがメインメモリ上では読出しアクセスだけで書込みアクセスされていなければ，メインメモリをファイル装置にスワップアウトする操作は不要である(スワップインだけでよい)．しかし，メインメモリが書き換えられた場合，それをファイル装置に反映させる（文字通りの置換を行う）必要がある．

メインメモリ⇔ファイル装置間のスループットはプロセッサ（キャッシュ）⇔メインメモリ間のそれに比べると低く，一方，一度に転送する命令/データ量すなわちブロックサイズは大きい．したがって，仮想メモリでは，バックアップメモリの更新は，メインメモリへの書込み時ではなく，ブロック置換（スワップアウト）時に行うのが普通である．これは，キャッシュメモリにおけるライトバック方式（6.3.5項〔2〕参照）にあたる．

〔9〕 **リロケーション** 実行されるプログラム（命令やデータ）の実メモリ上での割付けアドレスは，OSがメインメモリ(実メモリ)の利用効率を考えて仮想メモリ機構の支援によって決定する．このとき，そのプログラムのメインメモリにおける実アドレスが確定する．このアドレス決定/変更を**リロケーション**（relocation：再配置）という．

リロケーションは仮想メモリ機構の実現におけるハードウェア/ソフトウェアトレードオフと密接な関係がある．すなわち，リロケーションを，①ハード

〔3〕 **ダイレクトマッピング** 図6.15に示すように，メインメモリの第M番目のブロックを格納することのできるキャッシュの第C番目のブロックは，たとえば

$$C=(M \bmod B_C) \tag{6.10}$$

などによってあらかじめ決められ固定されている．

ダイレクトマッピングの長所は，マッピング方法が固定されているので

① マッピング機構のハードウェア化や実現が簡単である

② マッピングテーブルの対応セルだけ検索すればよい（連想ではない）ので，キャッシュアクセスにオーバヘッドがない

などである．

一方，ダイレクトマッピングの短所は

③ ブロック置換の自由度が少ないので，同一キャッシュブロックに割り付けられる複数のメインメモリブロックへのアクセスが競合したり連続する場合にヒット率が激減する

などである．

〔4〕 **フルアソシアティブマッピング** フルアソシアティブマッピングの長所は，マッピング方法がまったく自由であるので

① メインメモリの管理方法やブロック置換アルゴリズムの決定における自由度が大きい

などである．

一方，フルアソシアティブマッピングの短所は，複雑な機構のマッピングテーブル（連想メモリ）が必要となることによって

② キャッシュ機構をハードウェアで実装するためのコストが高い

③ 連想検索時間だけキャッシュへのアクセス時間がダイレクトマッピングよりも余計にかかる

などである．

〔3〕のダイレクトマッピングとこのフルアソシアティブマッピングの各特徴は1.1.3項〔4〕で述べた「固定 ⇔ 可変」のトレードオフの実例である．

〔5〕 **セットアソシアティブマッピング** まず，ダイレクトマッピングによってセット（複数個のキャッシュブロック）のマッピング関係を求め，次に，フルアソシアティブマッピングによってそのセット内の対応ブロックを決める

しかし，仮想メモリとキャッシュの最大の違いは，次に示すように，各方式が対象とするメモリ階層とそれに起因する導入目的である．

(a) **仮想メモリ**：メインメモリ ⇔ ファイル装置（補助メモリ）の階層関係を対象とし，メモリ空間の拡大を主目的とする．ユーザ（最終的には，コンパイラなどのマシン命令生成者やプロセッサで実行されるマシン命令そのもの）のためのメモリアーキテクチャの空間的性能の改善手法である．

(b) **キャッシュ**：キャッシュ ⇔ メインメモリの階層関係を対象とし，メインメモリアクセスの高速化を主目的とする．プロセッサのためのメモリアーキテクチャの時間的性能の改善手法である．

したがって，方式の選択においては，これらの導入目的が重要な役割を果たす場合がある．

〔2〕 **仮想メモリとキャッシュの要素技術の相違点** 〔1〕は，「対象メモリ階層の違いによって，ハードウェア機構とソフトウェア機能（特にOS機能）との機能分担方式すなわちハードウェア/ソフトウェアトレードオフがかなり異なる」ことを示している．

たとえば，次のような要素技術における具体的な相違点が指摘でき，これらに注意して各方式におけるハードウェア/ソフトウェアトレードオフを設計する必要がある．

(1) **メモリ容量**：キャッシュとして実装可能な容量はハードウェア規模や高速性（プロセッサに追従できる速度）の要求などによって厳しい制限（オンチップなど）を受けるが，仮想メモリを構成するファイル装置に対するその制限は緩い．すなわち，仮想メモリ空間サイズに対する自由度は大きく，仮想メモリのアドレス変換機構（たとえば，マッピングテーブルサイズやアドレス変換方式）の構成には拡張性や可変性が要求される．

(2) **ミスヒット**：仮想メモリのブロックサイズはキャッシュよりも大きく，また，OSが参照局所性を利用したメインメモリ割付け（プロセス割付け）を行うことができるので，仮想メモリのミスヒット（仮想メモリでは「フォールト」という）率はほとんど無視できる．したがって，仮想メモリの性能評価においてはミスペナルティ時間をそれほど重視する必要がない．

仮想メモリにおけるフォールトは割込み要因となりOSが処理するのに対して，キャッシュのミスヒットはハードウェアによって直ちにミスペナルテ

可能部分（並列性）を抽出する機能である．ベクトル化コンパイラ（7.2.2項〔8〕参照）と同様にループを対象とするものが多い．並列化の障害となるデータ依存や制御依存の解析が主となる（命令間の依存については，5.2.2項〔5〕で詳述）．

(2) **データ分割**：プロセッサ間通信のオーバヘッドを避けるように，データの最適な配置を考える．

(3) **タスク分割**：1個の要素プロセッサで行う単位並列処理機能を「タスク(task)」といい，これを並列コンピュータアーキテクチャに合わせて最適化（最適粒度）とする．

(4) **タスクスケジューリング**：(2)と(3)の結果を利用して，並列タスクの実行順序をスケジューリングする．OSによる動的スケジューリングと併用するのが普通である．

(5) **プログラムリストラクチャリング**(program restructuring；プログラム再構成)：(1)の結果を利用して，ループを並列化したり，並列化できないループを種々の方法（5.4.2項〔8〕で述べたループアンローリングなど）で再構成する．ベクトル化手法が流用できる場合も多い．

(6) **並列化コード生成**：(1)～(5)の結果をもとにして各要素プロセッサのマシン命令コードを生成する．この際には，冗長コードの除去などの最適化が行われる．

(2)(3)(4)はコンパイラによる負荷分散手法であり，**静的負荷分散**という．

〔21〕 **並列OS**　並列OSは並列プログラミング言語処理系とともに並列コンピュータシステムのシステムプログラム機能を担当するので，並列プログラミング機能および並列処理ハードウェア機構の両者のオーバヘッドのない整合を図ることが必須となる．

逐次コンピュータのOS（逐次OS）や分散処理システムのOS（分散OS）との相違点は

- **逐次OSとの相違点**：超多数のハードウェア資源を対象とする
- **分散OSとの相違点**：密結合通信（〔1〕参照）を扱う（分散処理と並列処理との相違については表7.3参照）

である．

したがって，並列OSの要件としては，次のようなものがあげられる．

や　行

ら　行