試験対策テキスト［【ベーステクノロジ編】

無料体験入学者用

TAC

はじめに

応用情報技術者試験(AP)は2009年春より実施が開始される試験区分であり，従来のソフトウェア開発技術者試験(SW)の後継に位置付けられる試験です。対象者像は，

「高度IT人材となるために必要な応用的知識・技能をもち，
高度IT人材としての方向性を確立した者」

とされています。基本情報技術者試験(FE)で求められる基本的な知識に加え，さらに専門的・詳細な内容を含めた応用的知識が問われることになります。

本書は応用情報技術者試験の出題範囲であるテクノロジ系，ストラテジ系，マネジメント系のうち，テクノロジ系の基礎となる情報理論やハードウェア，ソフトウェア等に関する分野の知識を網羅しています。その上で，読者の皆さんが効率よく学習が行えるよう，基礎的な用語や考え方を分かりやすく解説するように心がけました。

本書により，読者のみなさんが応用情報技術者試験に合格されることを願ってやみません。

2008年9月

TAC情報処理技術者講座

目　次

第3章　ハードウェア

学習テーマ　3-1

プロセッサアーキテクチャ

(1)　構成要素と命令実行の流れ

現在のコンピュータの多くは，「命令をメモリ(主記憶装置)から取り出してプロセッサによって逐次実行する」形式の，プログラム記憶方式(ノイマン型)コンピュータである。

プロセッサ(CPU：Central Processing Unit)は，コンピュータにおいて制御と演算を担当する中核部分であり，プロセッサの構成要素はアーキテクチャ(設計)によって異なるが，基本的には下図のようになる。

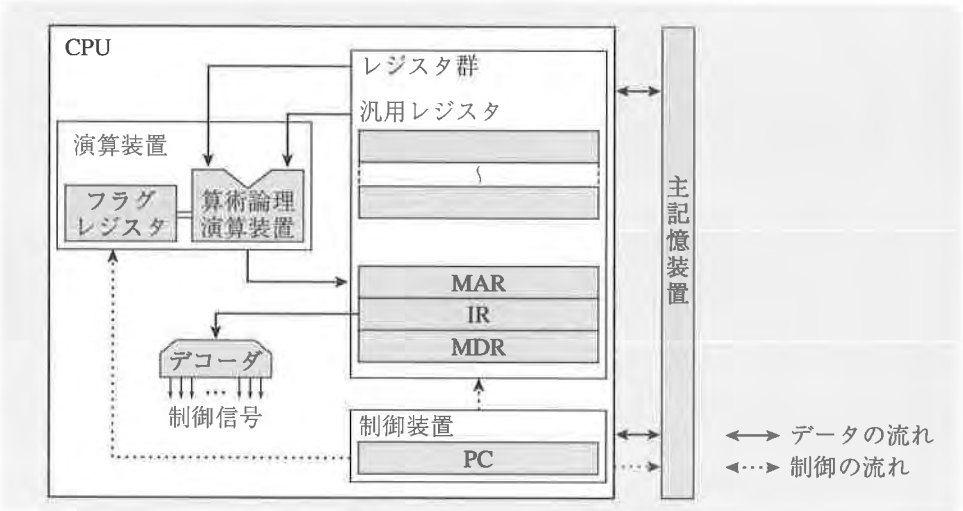


図3.1　プロセッサの構成

図に示したのは，オペランド(演算対象)の多くを汎用レジスタに保持するタイプの，「汎用レジスタアーキテクチャ」である。現在，多くのプロセッサがこの方式をとっている。

過去に用いられたアーキテクチャとしては，演算専用スタックにオペランドを積み上げていくスタックアーキテクチャ，演算専用レジスタ(アキュムレータ)に格納するアキュムレータアーキテクチャなどがある。

各構成要素の概要を次に示す。

表3.1　プロセッサ構成要素の概要

名　称	機　能
ALU (Arithmetic Logic Unit) 算術論理演算装置	算術演算や論理演算などの演算を行う。
PC (Program Counter) プログラムカウンタ ／命令アドレスレジスタ	現在実行中の(もしくは次に実行する)命令の格納場所(アドレス)を格納する。
GR (General purpose Register) 汎用レジスタ	演算対象やアドレス情報といったさまざまな情報を格納する。
FR (Flag Register) フラグレジスタ	演算結果の正負に関する情報を格納する。
IR (Instruction Register) 命令レジスタ	命令そのものを格納する。
ID (Instruction Decorder) デコーダ	命令を解読し，命令に応じた制御信号を生成する。
MAR (Memory Address Register) メモリアドレスレジスタ	アクセス対象となるメモリの位置(番地)情報を格納する。
MDR (Memory Data Register) メモリデータレジスタ	メモリから読み込むデータまたはメモリに書き込むデータの内容を格納する。

図で示した構成のプロセッサによってプログラムを実行するさいの，基本的な手順を以下に示す。

A：命令フェッチ(取出し)

[1]　PCの内容をMARへ転送する。

[2]　MARで指定されたアドレスを選択し，そこに格納されたデータ(命令)をMDRへ読み込む。

[3]　MDRの内容をIRに転送する。

B：命令デコード(解読)

[1]　IRに格納された命令のうち，命令コード部をデコーダによって解読し，実行する演算の種類を判別する。

[2]　その演算を実行するのに必要な演算回路を選択する。

C：オペランドの取出し(オペランドフェッチ)

[1]　IRに格納された命令のうち，アドレス部(アドレス修飾部なども含む)を用いてオペランドの有効アドレスを算出し，MARに転送する。

[2]　MARで指定されたアドレスに従って，メモリからオペランドをMDRに読み込む。

D：命令実行及び結果格納

[1]　B－[2]で選択された演算回路を用いて，各種レジスタ及びMDRの内容を用いた演算を行うよう，演算装置に制御信号を送出する。

[2]　演算結果をGRやメモリに格納する。

[3]　次ステップの準備として，PCの値を「次に実行すべき命令が格納されたアドレス」に変更する。

[4]　A－[1]に戻る。

このように，一般的には命令の取出しと実行が繰り返されることによってプログラムが実行される。

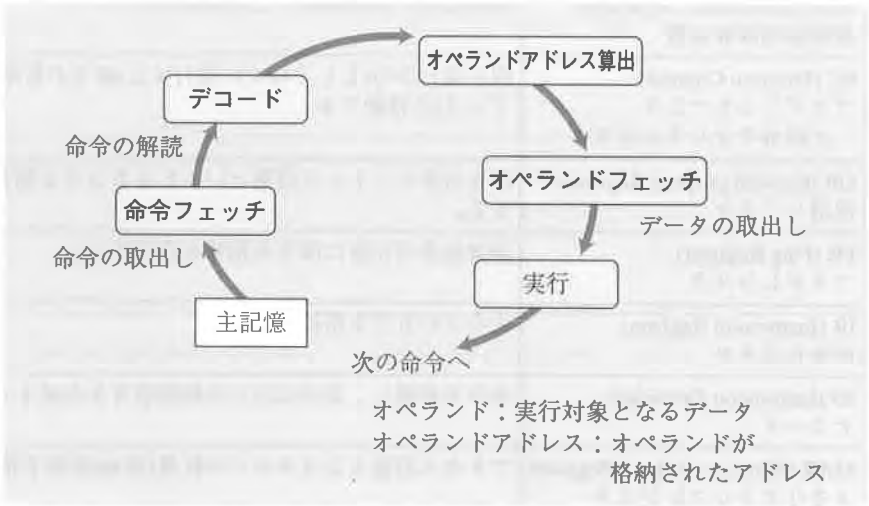


図3.2　命令の実行順序

参考　PC更新のタイミング

動作原理として必須ではないが，命令フェッチ時にPCの値を自動的に加算(インクリメント)しておけば，D－[3]では「分岐などによる特別なPCの指定」以外は行う必要がなくなる。一般的な逐次実行型のコンピュータでは，この動作が組み込まれていることが多い。

覚えよう

命令実行の手順

命令フェッチ → デコード → オペランドフェッチ → 実行

(2)　命令の構成

命令コード部とオペランド部

プロセッサとメモリの間でデータのやりとりを行うさいには，16，32，64ビットなどのある決まった長さのデータを一つの単位とする。このデータ単位をワード(語)とよぶ。通常，ワードの長さは汎用レジスタのビット幅と等しい。

一般に一つの命令は1～数ワードで構成されており，各命令の長さ(これを命令語長とよぶ)を固定するか可変にするかは，プロセッサのアーキテクチャによって異なる。

各命令語の内容は，大きく

命令コード部(命令部，操作部)
…その命令の内容を表す

オペランド部(アドレス部，番地部)
…レジスタやメモリアドレスなど，演算対象の情報を表す

の二つに分けられる。このオペランド部でオペランドをいくつ指定するかは，命令の種類や採用するアーキテクチャによって異なる。

参考　nアドレス方式

オペランド部にn個のメモリアドレスが明示的に指定されているものを“nアドレス方式”とよんで分類することもある。各方式の違いは以下のようになる。

表3.2　nアドレス方式

名　称	主に使用されるアーキテクチャ	命令の例	動　作
0アドレス方式	スタック	ADD	スタックから2つPOPし，加算した結果をPUSH
1アドレス方式	アキュムレータ	ADD A ST　B	A番地の内容をアキュムレータに加算 アキュムレータの内容をB番地に格納
2アドレス方式	汎用レジスタ	ADD A, B	A番地の内容とB番地の内容を加算し，A番地に上書き
3アドレス方式	汎用レジスタ	ADD A, B, C	A番地の内容とB番地の内容を加算し，C番地に格納

また，“ADD　A, B”のようにメモリ上のアドレスを同時に二つ指定することはできないが，「複数用意された汎用レジスタのどれを使用するか」を指定して

LD　GR1, A　　(A番地の内容を汎用レジスタ1に転送)
ADD GR1, B　　(B番地の内容を汎用レジスタ1に加算)

のような記述は可能である，というコンピュータもある。このような命令方式については，汎用レジスター一つを0.5と数えて“1.5アドレス方式”とよぶこともある。

命令コードの種類

命令コード部には，各命令と1対1に対応したビットパターンが命令コードとして格納される。各命令は，その性質によって以下のように分類できる。

表3.3　命令の分類

名　称	内　容
データ転送命令	主記憶－レジスタ間，レジスター レジスタ間などでデータ転送を行う。
算術演算命令	加減乗除(四則演算)などの演算を行う。
論理演算命令	AND(論理積)，OR(論理和)，シフトなどの演算を行う。
制御命令	分岐やサブルーチンコールなど，実行順序の変更を行う。
入出力命令	入出力チャネルの起動／監視など，入出力に関する処理を行う。
特殊命令	割込みの禁止／解除，記憶保護など，上記に当てはまらない処理を行う。

アドレス修飾（アドレッシング）

オペランド部は，さらにいくつかの部分に分割できる。

以下に，「メモリアドレスを一つ，汎用レジスタを一つ指定する」タイプの，典型的な命令の構成例を示す。

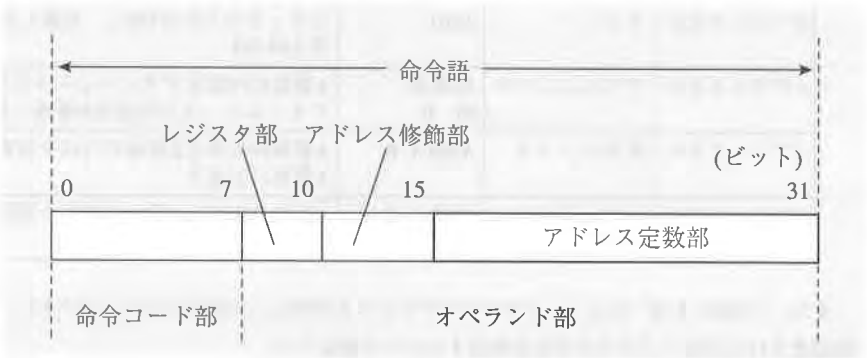


図3.3　命令の構成例

オペランドがメモリ上のどのアドレスに格納されているかという情報，すなわち有効アドレスは，

・アドレス修飾部で示される修飾方式に従って，
・アドレス定数部に演算を施す

ことで得られる。アドレス修飾部には，たとえば直接アドレス指定であれば“00001”といったように，それぞれの修飾方法に対応づけられたビット情報が格納される。

アドレス修飾の方式はまず大きく絶対方式と相対方式に分かれ，さらに具体的な有効アドレス計算法の違いによっていくつかに細分される。以下に，各アドレス修飾方式の概要を示す。

表3.4　アドレス修飾方式の概要

名　称		求められる有効アドレス※
絶対	直接アドレス指定方式	アドレス定数部の値
	間接アドレス指定方式	(アドレス定数部の値)番地に格納された値
相対	自己相対アドレス指定方式	アドレス定数部の値＋PCの値
	ベースアドレス指定方式 (ディスプレースメント)	アドレス定数部の値＋ベースレジスタの値
	インデックスアドレス 指定方式	アドレス定数部の値＋インデックスレジスタの値

※各方式を単独で用いた場合。詳しくは「参考：アドレス修飾の組合せ」を参照

たとえば，下図左の条件において，各アドレス修飾方式を用いて得られる有効アドレスは，それぞれ下図右のようになる。

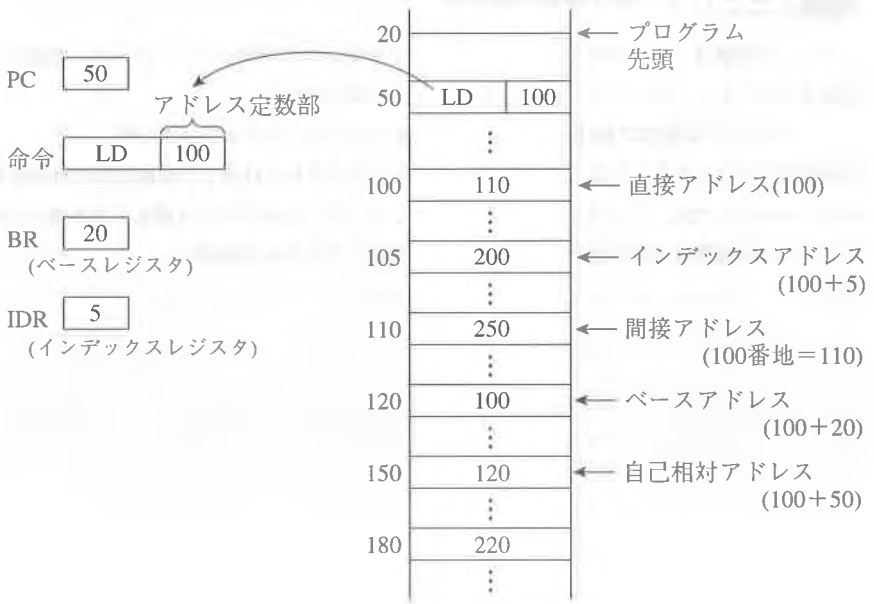


図3.4　有効アドレス計算の例

一般に相対方式の方が，限られたビット数でより広範囲のアドレス空間を表現することが可能である。また，自己相対アドレス指定方式やベースアドレス指定方式を利用することで，プログラムの再配置が可能となる。

もう一つの相対方式であるインデックスアドレス指定方式は，配列処理などの「ある領域を連続的にアクセスする」処理に適している。

覚えよう

再配置可能のための技術

→ ベースアドレス指定方式，自己相対アドレス指定方式

連続領域処理のための技術

→ インデックスアドレス指定方式

参考　**即値アドレス指定方式**

有効アドレスによってメモリアクセスを行うのではなく，アドレス定数部の値をそのまま演算対象として用いる方式の命令もある。これを即値アドレス指定方式とよぶ。

参考　**アドレス修飾の組合せ**

アドレス修飾は，一つの命令で一つしか指定できないわけではない。たとえば，間接アドレス指定方式とインデックスアドレス指定方式を組み合わせて指定し，

(アドレス定数部の値)番地に格納された値＋インデックスレジスタの値

を有効アドレスとするようなことも可能である(もちろんそれに対応した修飾部の設計が必要)。実際のコンピュータでは，ベースアドレス指定方式とインデックスアドレス指定方式を組み合わせ，プログラムの再配置及び配列操作利便性の両面を実現しているものが多い。

(3)　割込み

これまでの説明では触れなかったが，実際には命令実行と次の命令実行の間に，別のプログラムを実行するなど，何らかの要因によって処理を中断せざるを得ない場合もある。このために割込み制御が存在する。このような制御を行うために，割込みレジスタとよばれる専用のレジスタが用意されており，各種割込みが発生した場合，その内容に応じた値(割込みコード)がセットされる。

制御装置は次の命令フェッチに先んじてこの割込みレジスタの内容を調べ，割込み発生が確認されれば，その割込みの種類に応じてPSW(Program Status Word)を設定し直す。PSWとは，プログラムカウンタや動作モードなど，実行に必要な情報をセットにしたものである。

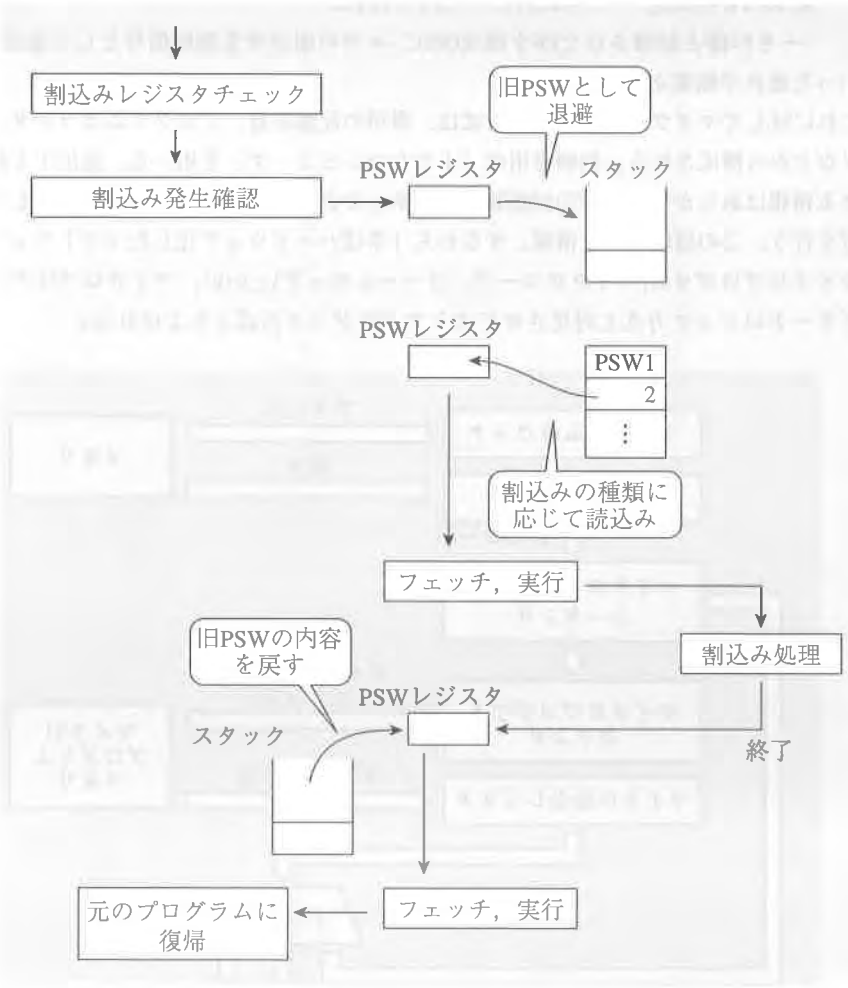


図3.5　割込みレジスタとPSW

(4)　プロセッサの分類

プロセッサを制御機構の観点で分類すると，マイクロプログラム方式とワイヤードロジック方式に大別できる。また，「1命令をどこまで高機能にするか」をはじめとする設計思想の観点で分類すると，RISCとCISCに大別できる。

マイクロプログラム方式とワイヤードロジック方式

ワイヤードロジック(布線論理制御)方式は，各命令コードに対してどのような制御信号を出すかという制御を，すべてハードウェアで実現する方式である。具体的にはフリップフロップ(FF)などの論理回路があらかじめ決まったパターンで結線されており，

命令内容の決定 → それに対応したFFをONに

→ そのFFと結線されたFFを順次ONに → その組合せを制御信号として送出

といった流れで制御が行われる。

これに対してマイクロプログラム方式は，専用の記憶装置，プログラムカウンタ，命令レジスタなどから構成される，制御専用の「小さなコンピュータ」を用いる。送出する制御信号に関する情報はあらかじめ専用の記憶装置に格納しておき，命令コードの内容に応じて取出し・実行を行う。この格納された情報，すなわち「半ばハードウェア化したソフトウェア」のことをマイクロプログラム(マイクロコード，ファームウェア)といい，マイクロプログラム方式はワイヤードロジック方式と対比させてストアドロジック方式ともよばれる。

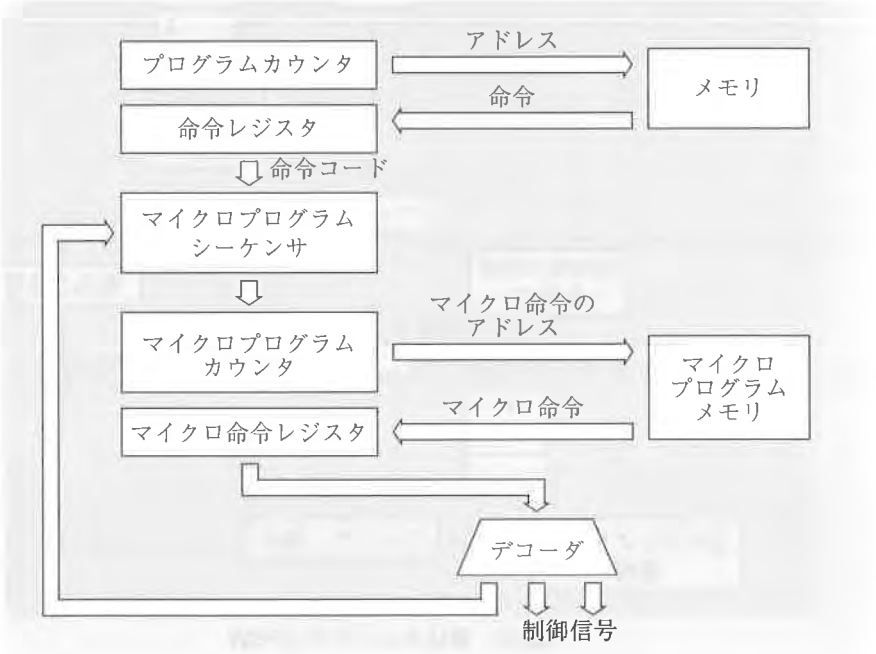


図3.6　マイクロプログラム方式

参考　**水平型マイクロコードと垂直型マイクロコード**

マイクロコードは，その構造によって水平型と垂直型に大別できる。

水平型：各装置に対する制御信号をすべて並べ，一つのマイクロコードとする。

垂直型：並列制御が可能な部分ごとにいくつかに分け，それぞれを一つのマイクロコードとする。

一つの命令が，複数のマイクロコードで実現されることになる。

一般に水平型の方が速度面で有利であるが，マイクロコード用のメモリ領域が大きくなる，メモリ効率が悪くなるといった面もある。

両方式を1命令当たりの命令実行速度で比較すると，マイクロコードの取出し・実行という動作を伴うマイクロプログラム方式よりも，すべてハードウェアで制御されるワイヤードロジック方式の方が速い。

しかし，使用する命令の種類(命令セット)が複雑になると，すべてをワイヤードロジック方式で実現するのは難しくなる。また，ワイヤードロジック方式では命令セットの変更や，他コンピュータのエミュレーションは困難であるが，マイクロプログラム方式であれば，マイクロコードの内容を変更することで対応できる。

表3.5　ワイヤードロジック方式とマイクロプログラム方式の比較

	ワイヤードロジック方式	マイクロプログラム方式
命令実行速度	速い	遅い
高機能命令の実現	困難	可能
仕様変更／エミュレーション	困難	可能

RISCとCISC

初期のコンピュータはすべてワイヤードロジック方式で実現されており，命令セットも単純なものだったが，マイクロプログラム方式が利用されるようになると，ユーザ要求に従って複雑・高機能な命令が多数追加されるようになっていった。

この結果，プロセッサに「高機能な命令を実現できるが，使用頻度の高い単純な命令に関しては逆に実行効率が落ちてしまう」という傾向が生じるようになった。この反省から生まれたのが，RISC(Reduced Instruction Set Computer) とよばれるコンピュータの設計思想である。これに対して，単純なものから複雑・高機能なものまで，多数の命令をセットとして用意した“従来設計”のコンピュータは，CISC(Complex Instruction Set Computer)とよばれる。

RISCは，CISCと比較して次のような特徴をもつ。

・使用頻度の高い最小限の命令だけで命令セットを構成する

この結果，命令の平均語長は短くなり，固定長にするのが容易となる。多くのRISC設計のプロセッサでは，メモリアクセスを伴う命令はロード命令及びストア命令だけに限定し，その分汎用レジスタを多く用意している。

また，複雑な処理は「命令の複数組合せ」で行うことになるので，プログラムのステップ数は多くなる。これは，高速化においてコンパイラの最適化が果たす役割が大きくなることを意味している。

・制御は(基本的に)ワイヤードロジック方式で行う
・ハードウェア的に制御を行うので，1命令の実行速度は速くなる

表3.6　RISC／CISCの比較

	RISC	CISC
命令の種類	少ない	多い
1命令の実行速度	速い	遅い
固定長命令の実現	容易	困難
プログラムのステップ数	多い	少ない
制御	ワイヤードロジック	マイクロプログラム

なお，RISC／CISCの区別は絶対的なものではない。昨今では，もともとCISC設計であったプロセッサもアーキテクチャにRISCの要素をとり入れ，改良するなど両者の境界は薄れてきている。

学習テーマ　3-2

プロセッサ性能と高速化技法

(1)　性能評価指標

プロセッサの性能評価指標は，何を基準にするかによっていくつかに分けられる。

クロック周波数

プロセッサの命令実行は，一定間隔で発生する信号に同期して行われる。この同期信号の発生頻度をクロック周波数とよび，一般に1秒当たりの発生数，すなわちHz(ヘルツ)を単位として表される。現在のパーソナルコンピュータでは，数100M～数GHzのプロセッサが主流である。

また，クロック周波数の逆数(1クロック当たりの時間)のことを，クロックサイクル(クロック周期)とよぶ。

各命令が実行にそれぞれ何クロックサイクルを費やすか(これをCPI：Cycles Per Instructionとよぶ)はプロセッサによって決まっているので，これを用いて1命令当たりの実行時間を求めることができる。たとえば，クロック周波数が800MHzで，CPIの平均値が5であるプロセッサがあると，このプロセッサの平均命令実行時間は

(1クロックサイクルの長さ)×(CPIの平均値)
＝(クロック周波数の逆数)×(CPIの平均値)
＝$(1／(800\times10^{6}))\times5$
＝$1.25\times10^{-9}\times5$ [秒]
＝6.25 [ナノ秒]

となる。

他の条件が同じであれば，理論的には命令実行時間がクロック周波数に反比例することになるので，同一アーキテクチャのプロセッサ間において，命令実行性能を比較するための指標としてクロック周波数を用いるのは有効である。

しかし，アーキテクチャの異なるプロセッサの性能を比較する場合，単純にクロック周波数だけで判断することは適切ではない。

MIPSとFLOPS

MIPS(Million Instructions Per Second)は，プロセッサが1秒間に実行できる命令数の平均値を，100万(＝10^{6})を単位として表したものである。

たとえば，前述のようにクロック周波数が800MHzで，CPIの平均値が5のプロセッサがあった場合，MIPS値は

(1秒間の命令実行数)／10^{6}
＝((クロック周波数)／(CPIの平均値))／10^{6}
＝$(800\times10^{6}／5)／10^{6}$
＝160

となる。なお，MIPS値は「単位をマイクロ秒としたときの平均命令実行時間」の逆数に等しい(この例ならば6.25ナノ秒＝0.00625マイクロ秒なので，1／0.00625＝160)。

MIPS値はあくまでも「1命令の実行」に関する指標であるから，クロック周波数と同様，アーキテクチャの異なるプロセッサの性能を比較するために用いるのは適切ではない。

また，命令当たりではなく，演算の処理回数に着目した性能評価指標の代表的なものに，FLOPS(FLoating point number Operations Per Second)がある。これは1秒間に実行可能な浮動小数点数演算の回数を指標としたもので，主に科学技術計算の分野において用いられる性能評価指標である。

MIPS値のように最初から単位調整は行われておらず，必要に応じて

1MFLOPS：1秒間に100万回の浮動小数点数演算を実行する
1GFLOPS：1秒間に10億回　　〃
1TFLOPS：1秒間に1兆回　　〃

といったように，M，G，Tなどの補助単位を付して用いることが多い。

FLOPS値も，異なるアーキテクチャをもったプロセッサ間の性能比較に用いるのは適切とはいえない。1回の浮動小数点数演算でどのような精度の処理を行うかは，アーキテクチャによって異なってくるからである。

アーキテクチャの異なるプロセッサについて性能を相互比較したい場合は，MIPS値やFLOPS値を直接比較するのではなく，ベンチマークを用いるのが望ましい。

覚えよう

MIPS：1秒当たりの“命令”実行数を，100万単位で表現
1MIPS＝1命令／マイクロ秒
FLOPS：1秒当たりの“浮動小数点数演算”実行数
科学技術計算分野で用いられる

異なるアーキテクチャのプロセッサは，MIPS／FLOPSでは単純に比較できない。

命令ミックスとMIPS値

実際のプロセッサでは，1命令の実行に必要なクロックサイクル数(CPI)は，命令ごとに異なっている。したがって，CPIの平均値を求めるためには，各命令の所要クロックサイクル数，及びプログラム中での出現頻度を考慮した「期待値」を算出する必要がある。このときに用いる，命令の分類及び各命令の出現頻度のセットのことを，命令ミックスとよぶ。

たとえば，クロック周波数が800MHzのプロセッサにおいて，次のような命令ミックスで表されるプログラムを実行するものとする。

表3.7　命令ミックスの例

命令の種類	所要クロックサイクル数	出現確率
浮動小数点数演算	6	0.2
メモリアクセス	3	0.4
分岐その他	2	0.4

このときCPIの平均値は，

Σ (各命令の所要クロックサイクル数×出現確率)

＝6×0.2 ＋ 3×0.4 ＋ 2×0.4

＝1.2＋1.2＋0.8

＝3.2

となり，平均命令実行時間及びMIPS値はそれぞれ

平均命令実行時間

＝(1／(800×10^6)) × 3.2 [秒]

＝4 [ナノ秒]

MIPS値

＝(800×10^6／3.2) ／ 10^6

＝250

となる。

もちろん，どのようなプログラムを想定するかによって設定する命令ミックスは異なり，それによって求められるMIPS値も変化することになる。命令ミックスとして広く利用されているもののなかには，科学技術計算をシミュレートしたギブソンミックス，事務処理計算をシミュレートしたコマーシャルミックスなどがある。

(2)　プロセッサアーキテクチャによる高速化

命令パイプラインの概念

命令パイプラインは，命令をいくつかのステージ(段階)に分け，各ステージをオーバラップさせながら並列に処理することで処理速度を向上させる技術である。このとき，1ステージを処理するために必要な時間をピッチ，同時に実行できる命令数をパイプラインの深さという。

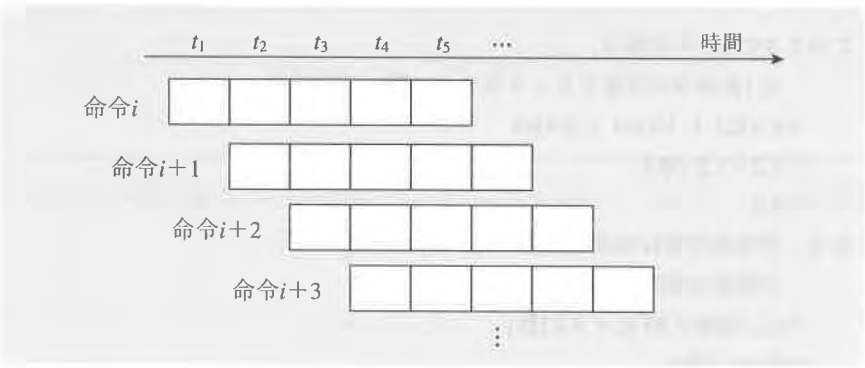


図3.7　命令パイプラインの概念

たとえば，上図のように1命令が5ステージに分割されて実行される場合，パイプラインを用いない場合はn個の命令実行に(5×n)サイクルを要するが，パイプラインを使用すれば，(n+4)サイクルで終了できる。これはみかけ上の命令実行速度が約5倍になることを意味する。

ただしこれは「各命令のサイクル数がすべて同じ」という仮定に基づいたものであり，このばらつきが大きいと，パイプラインの効果は薄れる。この点においてRISCとCISCを比較した場合，サイクル数を揃えやすいRISCの方が，パイプラインの恩恵を受けやすい。

覚えよう

パイプライン：理想的な条件であれば，みかけ上の実行速度を
“ステージ分割数”倍にできる
RISCの方が恩恵を受けやすい

なお，パイプライン制御を実現するためには，ある命令の実行中に，次の命令のフェッチを開始しておく，すなわち命令の「先取り」を可能にしておく必要がある。このような制御機構を先行制御(先回り制御)とよぶ。パイプラインは先行制御の一形態である，ともいえる。

ハザード

パイプライン制御では，命令の実行を並列化できない状況が発生することがある。このような状況をハザードまたは競合とよぶ。ハザードはその内容に応じて次のように分類できる。

表3.8　ハザードの分類

名　称	内　容
制御ハザード	分岐命令やシステムコールによって，先取りしていた命令が無効になることによるハザード。
データハザード	先行命令の結果を使用している場合，その結果待ちによるハザード。
構造ハザード	異なる命令が同じハードウェア資源に同時アクセスし，競合が生じることによるハザード。

ハザードが発生した場合にはそれを検知し，後続命令を停止して実行できる条件が整うまで待つことになる。これをパイプラインストールまたはインタロック(inter lock)とよぶ。

また，ハザードによる無駄を減らすための技術として，

「データ間の依存関係が矛盾しない範囲で，命令の順序を
メモリ上の順序とは異なるものに入れ替えてパイプラインに投入していく」

という手法がある。これをout-of-order実行とよぶ。

制御ハザードについては，コンパイラ及びハードウェアの工夫によって分岐条件の成立／不成立をできるだけ正しく予測し，それに従ってフェッチ先を決定するという分岐予測を用いることで，その発生数を低く抑えることができる。

データハザードについては，

コンパイル時：依存関係のある命令間にNOP(何もしない)命令を挟み込む

発生時：レジスタを介さないバイパス回路を用いて，演算装置の出力結果を演算装置に直接入力する(フォワーディング)

などの対策が考えられる。なお，構造ハザードに関しては，根本的にはハードウェア資源を複数用意するしかない。ただし，その分コストも増加することになるので，コストと性能(どこまでストールを許容するか)の兼ね合いで判断することになる。

パイプライン技術の発展

パイプライン技術をさらに発展させた高速化技術として，以下のようなものがある。

(a)　スーパーパイプライン

一般のパイプラインは命令の実行過程を数ステージに分割するが，スーパーパイプラインはこの段数をさらに深くし，10～数10という多数のステージに分割する。パイプラインの深さを大きく，ピッチをより短くすることによって実現すると考えてもよい。このような設計を採用することで命令の多重度が上がり，さらなる高速化が期待できる。また，ステージ当たりの処理が単純になるので，プロセッサのクロック自体も高速化できる。一方で，あまり段数を深くし過ぎるとパイプラインストールが発生しやすく，その影響も大きくなるというデメリットもある。

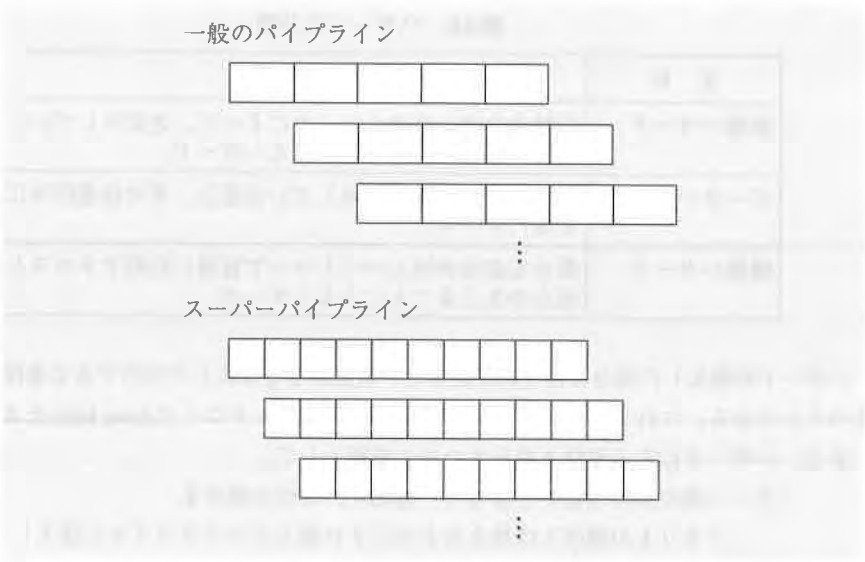


図3.8　スーパーパイプライン

(b)　スーパースカラ

スーパーパイプラインが「段数を深くする」のに対して，スーパースカラは「パイプラインそのものを複数用意する」手法である。スーパースカラを実現するためには，各ステージを実行するハードウェアを複数用意する必要があり，同じステージを並列に制御することで，CPIを1より小さくすることが可能となる。

パイプラインを何本用意するかを，スーパースカラ度とよぶこともある。下図はスーパースカラ度が2の例である。

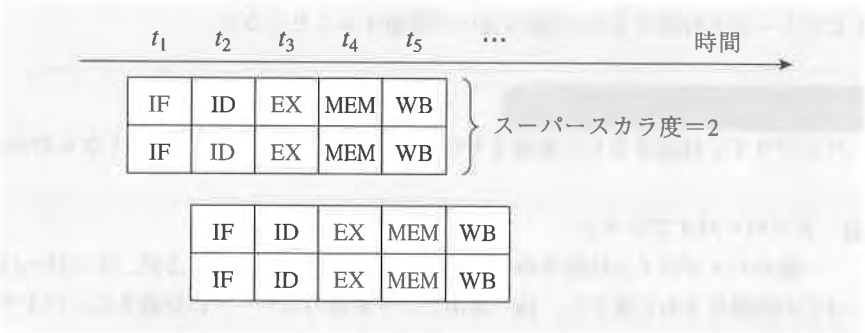


図3.9　スーパースカラ

覚えよう

スーパーパイプライン：ステージ数を増やし，複数命令を並列に処理
スーパースカラ：パイプラインを複数用意し，並列処理することによってCPIを1以下にすることが可能

その他の高速化技法

パイプライン関連以外の高速化技法として，以下のようなものがある。

(a)　VLIW

VLIW(Very Long Instruction Word)方式ではソースプログラムのコンパイル時に，通常ならばそれぞれ別個の命令語として扱われるような処理を一つのフィールドとして扱い，複数フィールドをまとめて「一つの長大な命令」を作成する。まとめられる命令には，依存関係によるハザードを引き起こさないことが求められ，並列度を抽出できない(依存関係のない命令が見つからない)場合はNOP(No OPeratioin；何もしない)命令で不足分を埋める必要がある。

図3.10　VLIW方式における命令語の構成例

プロセッサ側では各フィールドに対応した演算器をそれぞれ用意しておき，各フィールドを並列に実行することによって並列処理を実現する。

VLIW方式はスーパースカラと比較すると，比較的単純なハードウェア構成・制御で高速化が可能となるが，逆にソフトウェア(コンパイラ)側に相当の工夫が求められる。そのため当初はなかなか実用的なプロセッサが普及しなかったが，昨今では各ベンダでの開発も進み，パーソナルコンピュータ用プロセッサでもVLIW方式を採用したものが登場してきている。

(b)　ベクトルプロセッサ

ベクトルとは(x1，x2，…，xn)のような，1組の配列データを意味する。ベクトルプロセッサでは各種データを一つ一つ扱うのではなく，ベクトル形式でまとめて扱うことで高速化を図るものである。主に科学技術計算の分野で用いられる。

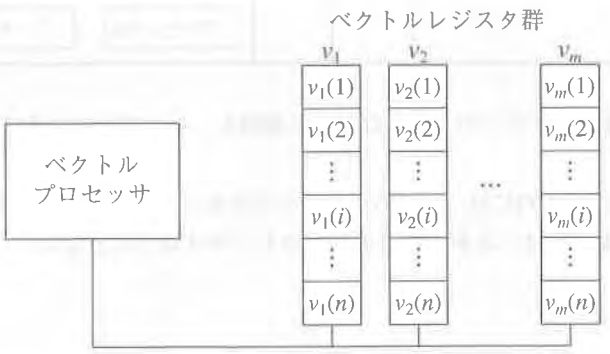


図3.11　ベクトルプロセッサ

(3)　マルチプロセッサ

マルチプロセッサの概念

前記(2)では命令ステージの分割，演算器の並列化など，一つのプロセッサを用いていかに高速化を図るかという技術を述べたが，それとは別に

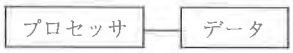
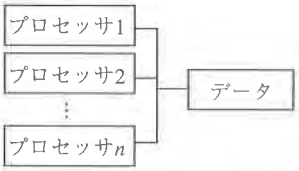
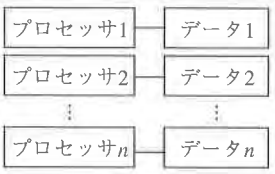
「プロセッサそのものを複数用意し，並列に制御を行う」

という思想の高速化技法もある。これを総称してマルチプロセッサとよぶ。

Flynnの分類

並列計算機システムの分類に用いられる概念の一つに，スタンフォード大学のM.J.Flynnが提唱したものがある。これは，“命令の流れ”と“データの流れ”がそれぞれ単一か複数かによって，プロセッサを次の4種類に分類するものである。

表3.9　Flynnの分類

名　称	命令の流れ	データの流れ	構成イメージ
SISD (Single Instruction stream / Single Data stream)	単一	単一	
SIMD (Single Instruction stream / Multiple Data stream)	単一	複数	
MISD (Multiple Instruction stream / Single Data stream)	複数	単一	
MIMD (Multiple Instruction stream / Multiple Data stream)	複数	複数	

通常の単純なシングルプロセッサはSISDに該当し，いわゆるマルチプロセッサはMIMDに該当する。

SIMDに該当するものには，前述のベクトルプロセッサなどがある。また，MISDは分類上の概念として登場してはいるが，該当するプロセッサはほとんどない。

参考　アレイプロセッサ

SIMDに該当するプロセッサの一つに，アレイプロセッサが挙げられる。アレイプロセッサは，多数の演算用プロセッサを配列状に結合し，それらを一つの制御プロセッサで制御することで高速化を実現したものである。

密結合と疎結合

マルチプロセッサは，各プロセッサが資源をどこまで共有しているかによって，密結合と疎結合に分類できる。

(a)　密結合マルチプロセッサ

密結合では，全プロセッサが一つの主記憶装置(メモリ)を共有し，単一のOSの制御の下で処理を行う。各プロセッサが異なるプロセスを実行できるが，メモリが共有されているため，プロセス間の排他制御，同期制御といった仕組みが重要となる。

(b)　疎結合マルチプロセッサ

疎結合では，各プロセッサがそれぞれメモリとOSを個別にもち，主にディスク装置を共有する。複数のOSを同時に実行できるが，ディスク装置のような資源の排他制御などが重要になる。また，メモリを共有しないため，プロセッサ間で情報を共有する場合は，メッセージ交換などの手段によって行われる。

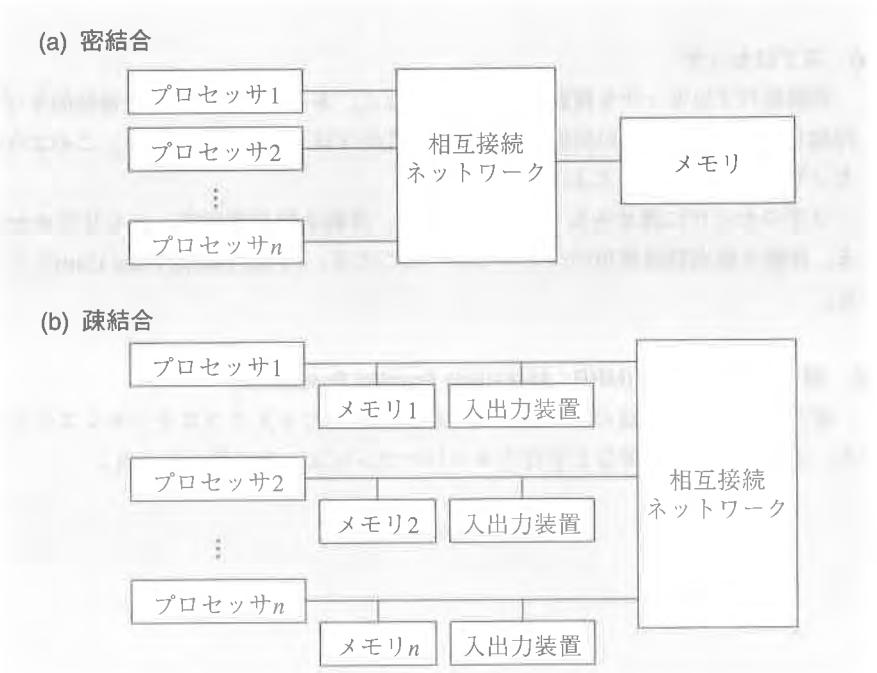


図3.12　密結合と疎結合

覚えよう

密結合：メモリとOSを共有
疎結合：それぞれメモリとOSをもち，メッセージ交換で協調

このほか，タンデム結合マルチプロセッサとよばれる，分散処理的な概念もある。

参考　SMPとASMP

マルチプロセッサシステムには，対称，非対称といった区別の仕方もある。一般的には対称型マルチプロセッサ方式(SMP：Symmetric Multiple Processor)とは「すべてのCPUが同等の立場」，すなわち，プロセッサごとに役割が決められていない方式である。この場合，CPUにかかる負荷は均等に各プロセッサに配分されると考えてよい。これに対してプロセッサごとにOS用，アプリケーション用といった形で役割を決める方式を非対称型マルチプロセッサ方式(ASMP：ASymmetric Multiple Processor)という。

なお，このほかにも密結合マルチプロセッサのことを対称型マルチプロセッサという場合もあるが，一般的には「均等な負荷の配分」と考えればよい。

その他の複数プロセッサ技術

そのほか，複数プロセッサでシステムを構成する技術として，以下のようなものがある。

(a)　コプロセッサ

同機能のプロセッサを複数並べるのではなく，ある処理に特化した補助的なプロセッサを付加し，主プロセッサの負担を軽減する形式のプロセッサ構成がある。このような補助プロセッサをコプロセッサとよぶ。

コプロセッサに課せられる役割としては，浮動小数点数演算，メモリアクセスなどがある。浮動小数点数演算用のコプロセッサのことを，FPU(Floating Point Unit)とよぶこともある。

(b)　超並列プロセッサ(MPP：Massively Parallel Processor)

数百～数万という数のプロセッサを相互接続したマルチプロセッサシステムのことである。大規模な数値解析などを行うスーパーコンピュータに用いられる。