# コンピュータ工学入門　付録

2016 年 2 月 12 日 更新
2018年 6月 6日更新

鏡 慎吾
佐野 健太郎
滝沢 寛之　　共著
岡谷 貴之
小林 広明

コロナ社

# A 順序回路：発展編

本章では，順序回路の設計に関する発展的なトピックを扱う。具体的には，順序回路の設計における状態数の最小化と，非同期式順序回路について述べる。

## A.1　状態数の最小化

5 章で設計した自動販売機の順序回路は，最初に状態遷移図を作成した時点で状態数は最小になっていた。一方，仕様が複雑で必要な内部状態の数が多くなると，実際には必要ない，冗長な状態を作成してしまうことがある。一般に，状態の数が少ないほど最終的な回路はより単純になる。したがって，不要な状態は除去し，状態の数を必要最小限にまで減らすのが望ましい。

完成した状態遷移図を元に，内部状態を削減する方法を考える。例えば，**図 A.1** 左のような状態遷移図（の一部）があったとする。入力 0 を得て $S_1$ へ状態が遷移するところから順序回路は動作を開始するものとする。簡単のため，入力および出力いずれも 1 ビットの 2 進数としている。つまり例えば，$S_1$ において，外部から入力 0 が入ると $S_3$ へ，入力 1 が入ると $S_2$ へ移る。そのとき，それぞれ出力 1 および 0 を出す。

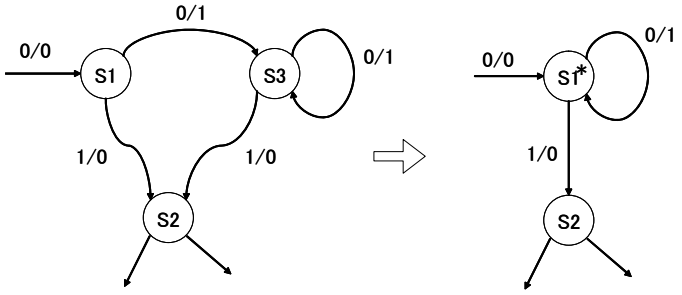



**図 A.1**　状態数の最小化の例

ここで，$S_1$ と $S_3$ の 2 つの状態に注目する。これらはいずれも，入力 1 を受けて $S_3$ に遷移し，その際出力 0 を出し，この点で $S_1$ と $S_3$ の動作は同一である。また，$S_1$，$S_3$ において 0 が入力されたとき，2 つともに 1 を出力する。その際の遷移は，$S_1$ は $S_3$ へ，$S_3$ は自分

自身（$S_3$）へと移るから，移る先も同一である。このように詳細を見てゆくと，$S_1$ と $S_3$ をまとめて一つの内部状態で表せると分かる。図 A.1 右のように，2 つをまとめて新たな内部状態（$S_1^*$ とした）を作れば良いと分かる。つまり，図 A.1 左は右のように書き換えても，順序回路としての動作は同じである。

この例のように，入出力および遷移が共通な状態のいくつかを 1 つにまとめるというのはそれほど難しい作業ではない。より難しいのは，複数の状態のペア（あるいは組み合わせ）を，同時に 1 つにまとめる操作を行ったときのみ状態数を削減できるような場合である。

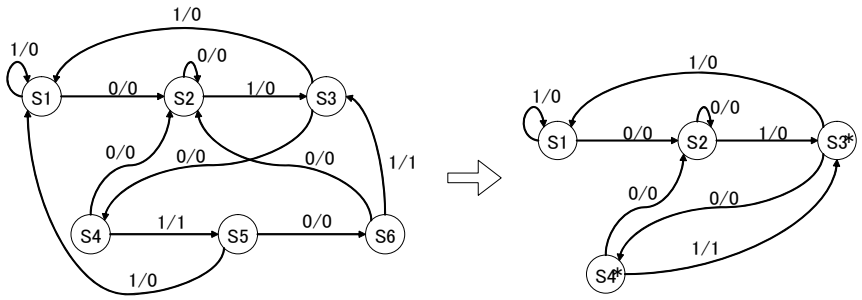



**図 A.2**　状態数の最小化の例

そのような例を**図 A.2** に示す。同図左が元の状態遷移図である。先に結論を述べると，内部状態 $S_3$ と $S_5$ を一つにまとめ，$S_4$ と $S_6$ を一つにまとめることができる。結果が同図右で，$S_3$ と $S_5$ をまとめたものを $S_3^*$，$S_4$ と $S_6$ をまとめたものを $S_4^*$ と表記している。

なぜこのように状態を減らせるのかは，次のように考えると理解できる。まず，$S_3$ と $S_5$ に注目すると，これらは共通して入力 1 で $S_1$ に遷移し，その際の出力は 0 である。しかし，入力 0 のとき $S_3$ と $S_5$ はそれぞれ別の状態，$S_4$ と $S_6$ に遷移してしまうので，結局これらを 1 つにまとめることはできないように思われる。しかしながらもし，さらに $S_4$ と $S_6$ を 1 つにまとめることができたらどうなるかを考える。もしそうできれば，$S_3$ と $S_5$ は，$S_4$ と $S_6$ をまとめた 1 つの内部状態へと入力 0 で遷移することになる。しかもその際の出力は 0 で共通である。$S_3$ と $S_5$ への遷移について考えれば，$S_6$ と $S_4$ からは入力 1 があったときにそれぞれ $S_3$ と $S_5$ に遷移しており，遷移の際の出力は 1 とこちらも共通している。この他に $S_3$ へは $S_2$ からも遷移できるが，これは $S_3$ と $S_5$ をまとめることによる影響を受けない。したがって，$S_4$ と $S_6$ を 1 つにまとめることができれば，$S_3$ と $S_5$ も 1 つにまとめられると結論される。では，$S_4$ と $S_6$ を 1 つにまとめることは可能だろうか？同じように分析すると，$S_3$ と $S_5$ を 1 つにまとめることができれば， $S_4$ と $S_6$ は 1 つにまとめられることが分かる。つまり $S_3$ と $S_5$ のペアと，$S_4$ と $S_6$ のペアを同時に，1 つにまとめれば，それぞれのペアの片

方の状態を削除でき，トータルでは 2 つの状態を減らすことができる。これを実行した結果が図 A.2 右である。

このように，ある複数の内部状態の集合を 1 つにまとめると，他の複数の内部状態の集合を 1 つにまとめられる，というように依存関係が存在する場合がある。状態数の削減は，このような可能性を考慮しつつ行う必要がある。

## A.2　非同期式順序回路

4〜5 章では，主としてクロック信号に同期して動作する順序回路を考えてきた。これに対し，クロック信号を要しない非同期式の順序回路もある。4.1.3 項で述べたように，非同期式の順序回路は同期式の回路よりも設計が難しく，したがって世の中での利用は，ある程度限られる。ここでは，そのような非同期式の順序回路について簡潔に説明する。

非同期式の順序回路は**図 A.3** のような構造を持つ。図のように非同期式の順序回路にも，外部との接点となる入出力と内部状態があり，さらにこれらの間を結ぶ組合せ回路があり，この点では同期式の順序回路と変わらない。構造上の違いは，非同期式の順序回路は，同期式の順序回路が持つようなクロック入力に同期する記憶回路を持たないことである。

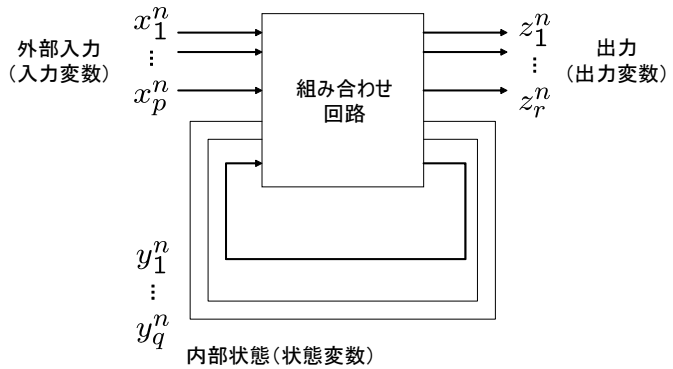



**図 A.3**　非同期式順序回路の概要図

この構造上の違いに関連して，非同期式と同期式とでは内部状態の遷移を起こす契機（きっかけ）が異なる。同期式の順序回路では，内部状態の遷移はクロックパルスを契機とする。一方で非同期式の順序回路では，内部状態の遷移は外部からの入力の変化を契機とする。同期式の場合，組合せ回路から出力された内部状態は，クロック信号に同期して動作する記憶回路を経由した。非同期式の場合，内部状態は直接，組合せ回路に帰還（フィードバック）される。

この違いにより，非同期式順序回路の動作は，同期式順序回路のそれとはかなり異なる。上で述べたように，状態の遷移は外部入力の変化を契機に起こる。外部入力が変化すると，組

合せ回路は外部への出力を変化させると同時に内部状態も変化させる。内部状態は組合せ回路に直接帰還されるため，内部状態の変化はただちに組合せ回路に入力される。その結果は，外部出力と内部状態を（再び）変化させてしまう可能性がある。このように組合せ回路が内部状態を直接自身に帰還するため，一般には以上の動作が瞬時に何度も繰り返されてしまうことがある。

もしそのような動作の結果，永久に内部状態の変化が繰り返し起こってしまうならば（その回路は発振状態にあると言える），一般には使いものにはならない。しかしながら，組合せ回路の部分をうまく設計すれば，外部入力の変化を契機に起こったこの状態変化の繰り返しが，少なくとも有限時間のうちには終わり，再度外部入力が変化するまでの間，内部状態がそれ以上変化しないようにできるかもしれない。これが達成されているとき，回路は安定状態にあると言う。外部入力を色々変化させても，1 つの安定状態から別の安定状態に移るように回路が動作するようにできれば，状態機械としての機能を実現できる。

したがって，そのような条件を満たす組合せ回路を設計することが目標となる。ただし，外部入力に任意の変化を許すと設計は一般に難しくなるので，外部入力の変化を，例えば同時に二つ以上の入力変数（ビット）が変化することはないと仮定し，その条件下で設計を行うなどする。また，安定状態に到達する途中で出力はいろいろと変化するが，その変化中の出力は無視し，安定状態に到達したときの出力のみ，意味を与える。

このように，同期式の順序回路に比べると一般に設計の難しい非同期式の順序回路であるが，動作速度が高速にできることや，クロック信号を要しないため回路設計に自由度が高まることなど利点も多いため，設計の難しさとこれらの利点を天秤にかけて実際の利用が決められる。

# B キャッシュメモリの構成

8.5 節では，キャッシュメモリの目的と基本的な動作を学んだ。本章では，その具体的な実現方式について議論する。

## B.1 データ格納位置の決定

キャッシュメモリの動作を実現するためには，まず，いま要求されたメモリアドレスのデータがキャッシュ内にあるのかどうか，あるならばどこにあるのかを判断しなくてはならない。そのためにはキャッシュ内にはデータが保存されているだけでは駄目で，それが主記憶のどこにあったかの情報も一緒に記憶しておかなくてはならない。この主記憶内での位置を示す情報をタグと呼ぶ。キャッシュメモリが格納する情報の単位を**キャッシュライン** (cache line) または**キャッシュブロック** (cache block) と呼ぶ。1 本のキャッシュラインにはデータとタグ，その他の管理情報が含まれる。管理情報としては例えば 有効 (valid) ビットなどがあり，有効なデータが入っているときのみ 1 を格納しておく。

具体例で考えよう。主記憶のアドレス長が 32 ビット，キャッシュライン 1 本当たりのデータサイズを 64 バイトとする。キャッシュへの格納は，主記憶の先頭から 64 バイトずつ区切った組ごとに行われる。従ってアドレスのうち下位 6 ビット (0 ～ 63 の数字) はキャッシュライン内の位置を表すことになる。残りの上位 26 ビットが，元の主記憶上の位置を知るために記録しておかなくてはならないものである。

いまキャッシュ内へのデータの置き方には何の制限もないとしよう。要求されたメモリアドレスのデータがキャッシュ内にあるかを知るためには，**図 B.1** のように，すべてのキャッシュラインについてこのタグの等値比較を行わなくてはならない。この方式は，後述するセットアソシアティブ方式のうち連想度が最大の場合に相当するため，**フルアソシアティブ** (full associative) 方式と呼ばれる。キャッシュメモリのサイズにもよるが，よほど小さなキャッシュでない限りこれは現実的には製造できない。

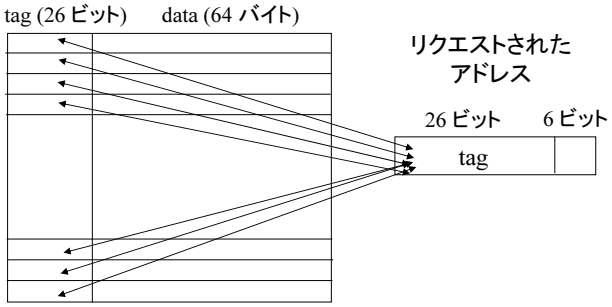



**図 B.1**　フルアソシアティブキャッシュメモリ。すべてのキャッシュラインのタグとの等値比較を行わなければならないため，現実的には製造が難しい。

この問題を回避するため，キャッシュメモリ内でデータを格納できる位置を制限するという方策がとられる。一番シンプルなものは**ダイレクトマップ** (direct mapping) 方式と呼ばれ，キャッシュ内での格納位置がメモリアドレスから一意に定められる。先ほどと同一の条件でキャッシュメモリの容量が 64 キロバイトの場合を**図 B.2** により説明する。保持できるキャッシュライン数は 1024 本となる。キャッシュ内の格納位置は 0 ～ 1023 の 10 ビットの整数で表され，これをインデックスと呼ぶ。主記憶アドレスのうち，キャッシュライン内の位置を表す下位 6 ビットに続く中位 10 ビットをインデックスとし，これにより格納位置を決定する。タグは残りの上位 16 ビットとなる。

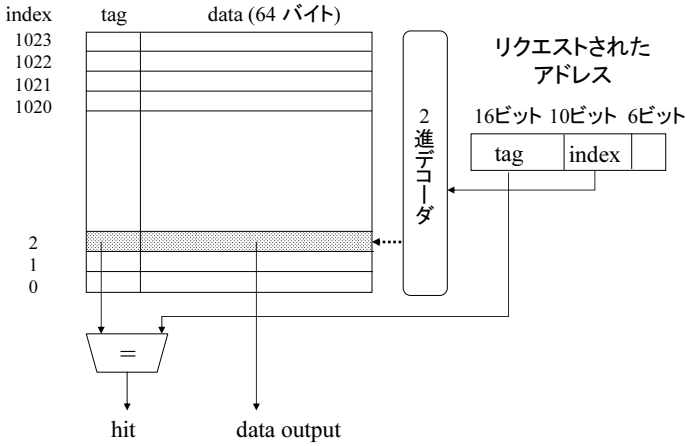



**図 B.2**　ダイレクトマップ方式のキャッシュメモリ。図では省略しているが，インデックスで指定されたキャッシュラインからは，タグだけではなく管理情報も読み出して確認し，ヒットかミスかを決める。

この場合，あるアドレスへのアクセス要求があった際に，インデックスで指定されるキャッシュライン 1 本だけを読み出しそのタグを等値比較すればよいため，現実的な実装が可能である。その代償としてヒット率を上げにくいという問題が生じる。例えば交互にアクセスする 2 つのデータのアドレスがたまたま同じインデックスを持っていたとすると，たとえキャッシュ内の他の位置にはまだまだ空きがあったとしても，一方へのアクセスの度に他方をキャッシュから追い出してしまうことになる。

そのため，実際にはこれらの間をとった方式である**セットアソシアティブ** (set associative) と呼ぶ構成を取ることが多い。これは，ある主記憶アドレスのデータの格納位置の候補を複数用意するもので，候補が $n$ 個の場合を $n$ ウェイセットアソシアティブ ($n$-way set associative) と呼ぶ。整数 $n$ は連想度，ウェイ数などと呼ばれる。前述のダイレクトマップ方式は $n = 1$ の場合に相当する。先ほどまでと同条件で連想度 $n = 4$ の場合を**図 B.3** を見ながら考える。全部で 1024 本あるキャッシュラインが 4 セットに分けられ，各セット内で 0 ～ 255 のインデックスを割り当てる。8 ビットの中位アドレスによってこのインデックスを指定するとセット内の位置が定まるため，キャッシュラインの候補は 4 本に絞られる。たまたま同じインデックスのアドレスへのアクセスが集中しても 4 ヶ所までならデータはキャッシュに残り続けることができ，またキャッシュヒット・ミスの判定も，たかだか 4 つのタグの等値比較で済ますことができる。

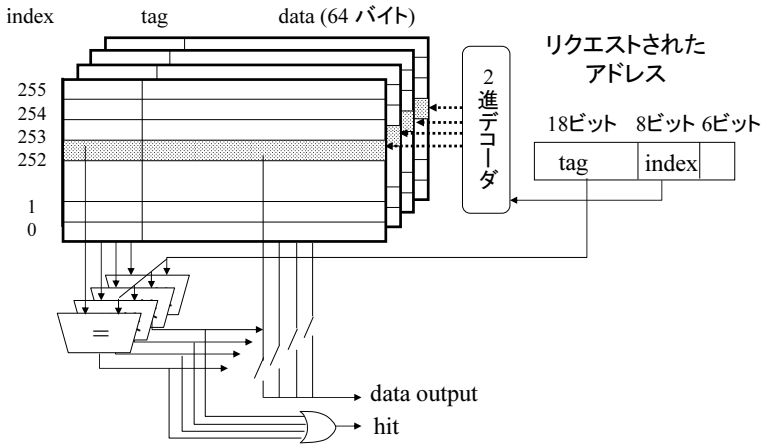



**図 B.3**　4 ウェイセットアソシアティブキャッシュメモリ。キャッシュメモリ全体を 4 つに分割して 4 並列でアクセスしていると考えるとわかりやすい。

## B.2 データ入れ替え方式

キャッシュに空きがないときに，どのデータを追い出すかも問題となる。$n$ ウェイセットアソシアティブの場合，$n$ 本のラインのうちから 1 つを選ばなくてはならない。ダイレクトマップ方式の場合は $n = 1$ であるから何も考える必要はない。連想度 $n$ が 2 以上の場合，理想的には将来参照されないものを選びたいわけだが，未来のことは知りようがない。

時間的局所性の原則に立てば，$n$ 本のうち最も長く参照されていないものが，今後も参照されないであろうと考えることができる。これを選択する方式を **LRU** (least recently used) と呼ぶ。LRU を厳密に実装するためには，ウェイの最終参照時刻順の情報をキャッシュライン内の管理情報として保持する必要がある。連想度が大きい場合は現実的には困難であり，適当な近似を導入した擬似 LRU が採用されることが多い。あるいはさらに割り切って，単純にランダムに入れ替える方式もしばしば用いられる。

## B.3 書き込みアクセス

以上では主に読み出しアクセスについて述べてきた。書き込みについても基本的には同様であるが，いくつか注意しなくてはならない点がある。

まず，キャッシュ内にデータが存在しないアドレスに対して書き込みを行う場合は，該当するキャッシュライン分のデータを主記憶からキャッシュにコピーしてきた上で書き込み処理を行わなくてはならない。

次に，キャッシュに書き込みを行った際に主記憶との整合性をどのように取るかという問題がある。最も確実な方法は，キャッシュへの書き込み時は必ず主記憶へも書き込む方式である。これは**ライトスルー** (write through) 方式と呼ばれる。容易に想像できるように，これでは書き込み速度を上げることはできない。

これに対して，書き込み時にはキャッシュのみを変更し，そのラインがキャッシュから追い出されるときに初めて主記憶に書き戻す方式を**ライトバック** (write back) 方式と呼ぶ。これにより主記憶へのアクセス回数の低減が期待できる。さらに効率を上げるため，書き込みがあったかどうかを示す変更 (modified) ビットをキャッシュラインに管理情報として持たせることが多い。あるラインがキャッシュから追い出される場合，変更ビットが立っていなければそのラインの内容は主記憶と同一であることがわかるため，主記憶へのコピーは省略できる。ただし，マルチプロセッサの場合など，主記憶にアクセスするシステムが他にも存在

する場合は別途対策が必要である。

## 章 末 問 題

【1】 主記憶のアドレス長が 32 ビットであるメモリシステムにおいて，データ容量 256 K バイトの 2 ウェイセットアソシアティブキャッシュを考える。キャッシュライン 1 本当たりのデータサイズを 32 バイトとする。以下の各項に答えよ。ただし 1 K = 1024 とする。

(a) このキャッシュメモリには何本のキャッシュラインが格納できるか答えよ。

(b) メモリアドレスのうち，タグ部とインデックス部がそれぞれ何ビットか答えよ。

(c) このキャッシュメモリを構成するために必要な物理メモリサイズを求めよ。ただし，データとタグ以外の管理情報のサイズは無視してよい。

【2】 データ容量が 32 K バイト，キャッシュライン 1 本当たりのデータサイズが 64 バイトのダイレクトマップキャッシュを持つシステムを考える。キャッシュヒット時間は 2 ns であり，ミスペナルティ時間は 18 ns である。ただし 1 K = 1024 とする。

各要素が 4 バイト整数を保持する $1024 \times 1024$ 行列が主記憶内の配列に行優先順序 (row-major order) で記憶されているとする。ただし行優先順序とは，配列の先頭から順に行列の $(1,1)$ 要素，$(1,2)$ 要素，$\cdots(1,1024)$ 要素を格納した後，$(2,1)$ 要素，$(2,2)$ 要素，$\cdots(2,1024)$ 要素を格納していき，同様に繰り返して最後に $(1024,1)$ 要素，$(1024,2)$ 要素，$\cdots$ $(1024,1024)$ 要素を格納する順序を指す。

この行列の全要素にアクセスするプログラムで，行優先で走査するものと列優先 (column-major) で走査するものの 2 種類を考える。行優先走査は上記の配列格納順序と同じ順序で走査するものであり，列優先走査は，行列の $(1,1)$ 要素，$(2,1)$ 要素，$\cdots$ $(1024,1)$ 要素を読み出した後，$(1,2)$ 要素，$(2,2)$ 要素，$\cdots$ $(1024,2)$ 要素を読み出していく方式である。

プログラムを起動した時点でこの行列の要素はキャッシュには全く保持されていないとし，他のプログラム等の影響はないとする。

(a) 行優先で走査した場合の平均メモリアクセス時間を求めよ。

(b) 列優先で走査した場合の平均メモリアクセス時間を求めよ。

# C 入出力システム

本章では，これまで棚上げにしてきた入出力装置とプロセッサとの間の情報のやり取りについて説明する。

## C.1　メモリマップ I/O 方式と入出力専用命令方式

MIPS を含む多くのプロセッサでは，**メモリマップ I/O** (memory-mapped I/O) と呼ばれる方式が採用されている。その概念を**図 C.1** に示す。プロセッサとメモリとを接続しているデータ信号線，アドレス信号線に，入出力装置が並行して接続されている。

ここで，メモリ空間を複数の領域に分けることを考える。例えば 0 ～ 0xffff ffff までアドレスを持つ 32 ビットプロセッサで，0 ～ 0x9fff ffff までをメモリに，0xa000 0000 ～ 0xa000 ffff までを装置 A に，0xa001 0000 ～ 0xa001 ffff までを装置 B に，0xa002 0000 ～ 0xa002 ffff までを装置 C にそれぞれ割り当てたとする。メモリを含めて各装置は常にアドレス線を見張っており，自分へ割り当てられたアドレスへのアクセスがあった場合にのみ動作する。

例えば 0xa001 0100 を実効アドレスとするロード命令が実行されると，入出力装置 B はメモリと同様の動作手順でそのアドレスに該当する値をデータ線に送り出すが，他の入出力装置やメモリは何もしない。この構成により，プロセッサから見て入出力装置はメモリと全く同様に扱える。入出力の処理をメモリ空間にマップしていることからメモリマップ I/O と呼ばれる。

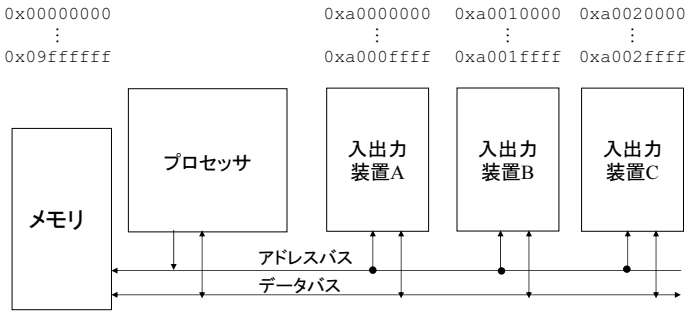



**図 C.1**　メモリマップ I/O の概念

この例でいうデータ信号線やアドレス信号線のように，複数の装置の間で信号をやり取りするために共有されている信号線は，複数の装置からの信号が乗り合うことから**バス** (bus) と呼ばれる。**データバス** (data bus)，**アドレスバス** (address bus) などのように用いる。

一方，例えば x86 アーキテクチャのように，メモリマップ I/O 方式を採用せずに入出力専用の命令を用意しているプロセッサも存在する。概念的には，メモリ用のバスのほかに入出力専用の I/O アドレスバス・ I/O データバスが存在すると考えればよい。それ以外の考え方は全く同様である。

バスを通じてやり取りされる数値が何を意味してどのように扱われるかは，入出力装置ごとに取り決められており，プロセッサはその取り決めに従って値を書き込んだり，値を読み出したりするプログラムを実行すればよい。典型的には，書き込まれる値は出力データや制御指令を符号化 (数値化) したものであり，読み出されるデータは装置の状態や入力データを符号化したものとなる。

## C.2 割　込　み

入出力装置との情報のやり取りは，基本的にはメモリと同様の方式で行うことがわかった。ところで，入出力装置側でデータが準備できるタイミングがわからない場合，あるいは準備できるまで大変長い時間がかかる場合は何が起きるだろうか。

例えば補助記憶装置からデータを読み込みたい場合を考える。補助記憶として広く用いられている**ハードディスク** (hard disk) は，磁性体で覆われた硬い円盤を高速回転させ，その表面上の目的位置に磁気読み書きヘッドを可動アームにより位置決めすることにより，ランダムアクセスを実現している。欲しいデータの位置を指示してから，実際に磁気読み書きヘッドが目標位置まで機械的に動いて磁気信号を読み取り，データが用意できるまでには，プロセッサから見た場合には大変に長い時間がかかる。具体的にはミリ秒のオーダであり，これはプロセッサが 1 命令を実行する時間の実に 100 万倍程度に相当する。

別の例としてキーボードからの入力を待つ場合を考えると，どのくらい待てばよいかわからないし，そもそもいつまで待っても入力は来ないかも知れない。

このような場合に対応するためには，入力処理を一定時間ごとに繰り返し，実際に有効なデータが出てくるまで続けるという方法が考えられる。このような動作を**ポーリング** (polling) と呼ぶ。しかしポーリングを効率的に行うのは難しい。ポーリング間隔が長ければ入力に対する反応が遅くなり，ひどいときには必要なデータを取りこぼしてしまう。間隔が短ければ，プロセッサは処理時間の多くを無駄に費やしてしまい，本来の仕事を満足に行えなくなる。

この状況を避けるため，ほとんどのプロセッサには**割込み** (interrupt) と呼ばれる機構が用意されている。入出力装置側で準備ができた時点で，そのときプロセッサが何の処理を行っているかに関係なく，実行中の命令に「割込む」ことを可能とする機構である。プロセッサは，発生した割込みの種類に応じた処理 (**割込みハンドラ**, interrupt handler) に制御を移し，その完了後にもとの命令に復帰して，何事もなかったように処理を継続する。

例えばハードディスクの場合，必要なデータの位置をディスク装置に指示した後，プロセッサはデータが返って来るのを待たずに別の処理に移る。データの準備ができたら，ディスク装置は割込み信号をプロセッサに送る。この時点で実行していた命令をプロセッサは一時中断し，ディスク装置からの値の読み出しを (メモリマップ I/O 方式の場合はロード命令により，あるいは専用命令方式の場合は入力命令により) 行い，無事データを得ることができる †。

割込みの動作が関数呼び出しに似ていることに気づけば，その実現方法はある程度想像できる。割込みハンドラをメモリ上に用意しておき，割込みがあった場合には，その先頭アドレスにジャンプする。また，割込み開始時に実行中だった命令のアドレスを保存しておき，割込みハンドラの終了時にその位置にリターンする。割込み要因ごとに個別の割込みハンドラを用意する場合，それらへのジャンプ先を記憶しておく表を**割込みベクタテーブル** (interrupt vector table) と呼ぶ。そのようなベクタ方式を用いずに，常に同一の割込みハンドラにジャンプし，ハンドラ内で割込み要因を調査する方式もしばしば用いられる。

関数呼び出しと異なるのは，割り込みハンドラが正しく動作するための準備をあらかじめ行っておくことができない点である。例えば MIPS で関数呼び出しを行う場合，一時レジスタ (t0 ∼ t9) の値は，呼び出された関数の側で上書きされてしまってもよいように，あらかじめスタックに退避してからジャンプする。ところが割込みはいつ発生するのかわからないので，「あらかじめ退避しておく」ことなどできるはずがない。MIPS で割込みハンドラが勝手に使ってよいのは，そのために予約されているレジスタ k0, k1 のみであり，その他のレジスタを使う場合は割込みハンドラ内でメモリに退避する必要がある。プロセッサによっては，レジスタ群を退避したり，専用のスタック領域を用意したり等の準備を自動で行ってくれる場合もある。

† 無事データを得たあとは，そのデータを用いる処理と，割り込まれた方の処理の両方が実行可能な状態になる。それらをその後どのような順番で実行していくかを決めるのはオペレーティングシステムの仕事である (D.3 節)。

割込み機構は大変便利なため，入出力処理だけではなく，プログラム実行上の例外的な処理を扱うためにも用いられる。例えば演算命令を実行した結果オーバーフローが生じた場合に割込みを発生させることで，その始末を行わせることができる。このような割込みは**ソフトウェア割込み** (software interrupt)，あるいはそれがプログラムの内部的な要因により生じることに注目して**内部割込み** (internal interrupt) などと呼ばれる †。オーバーフローのほか，ゼロによる除算，メモリ保護違反などを取り扱う際などにも用いられる。

## C.3　DMA 転　送

入出力装置が大量のデータを扱う場合，データ転送をすべてプロセッサ経由で行うのは効率が悪い。そのため多くのコンピュータでは **DMA** (direct memory access) と呼ばれる転送方式が用意されている。例えば，ハードディスク上の大きなデータをメモリに転送する場合に，いったんプロセッサ内に読み出し，次いでメモリにストアすることを全データに対して繰り返す代わりに，ハードディスクからメモリへデータを直接転送する。これにより，転送中もプロセッサは別の処理をすることが可能となる。

もちろん DMA 転送中にプロセッサがバスを使用しなくてはならなくなった場合は，プロセッサを待たせるか，DMA 転送を一時中断するか等の措置が必要となる。キャッシュメモリが有効に機能していれば，そのような状況が起こる確率を十分に低くすることができる。

## C.4　入出力とオペレーティングシステム

パーソナルコンピュータも含め，多くのコンピュータでは複数のプログラムが同時に並行して動作する。D 章で学ぶ通り，オペレーティングシステム (OS) によりそのような動作が実現される。

その場合，入出力操作を各プログラムが自由に行うことは望ましくない。例えばハードディスクの同一領域に複数のプログラムがそれぞれのデータを勝手に書き込むと，結果は期待されたものとは異なってしまう。操作によってはコンピュータシステムの安定動作自体を損なってしまうことすら生じ得る。

† ほぼ同じ意味で**例外** (exception) という語を用いる場合もある。D.3 節を参照されたい。

そのため多くの OS では，プログラムが入出力操作を直接行うことを禁止している。これを実現するため，多くのプロセッサには，あらゆる操作が許される**特権モード** (supervisor mode, **カーネルモード**, kernel mode) と，操作が一部制限される**ユーザモード** (user mode) が用意されている。通常のプログラムはユーザモードで動作し，入出力を行う命令の実行は許されない。入出力を行いたい場合は特権モードで動作する OS に処理を依頼する。割込みハンドラも OS の一部であり，入出力装置からの割込みはすべて OS が処理する。OS は，複数のプログラムからの依頼や複数の装置からの割込みを適切に処理し，システム全体としての整合性を保つ。

ユーザプログラムからの OS 機能呼び出しは**システムコール** (system call) と呼ばれ，高級言語で書かれたプログラム上では関数呼び出しと同じように見える。しかし，単なる関数呼び出しでこれを実現することはできない。ユーザモードでの命令実行によって勝手に特権モードに移行できるようでは，モードを分けた意味がないからである。ここでソフトウェア割込みが活用される。ユーザプログラムがソフトウェア割込みを明示的に発生させる命令 † を実行すると，プロセッサは特権モードに移行し割込みハンドラが (すなわち OS が) 実行される。

以上のように，入出力を行うために必要なソフトウェアは，ユーザプログラム側からの呼び出される部分も装置側から呼び出される部分もすべて OS の一部として動作する。そのようなソフトウェアを**デバイスドライバ** (device driver) と呼ぶ。通常，デバイスドライバは入出力装置の製造者が提供し，機器導入の際にインストールされるものであるが，メジャーな装置については OS に最初から付属していることも多い。

オペレーティングシステム側から見た割込みの詳細については D.3 節にて学ぶ。

## C.5 入出力システムの実際

図 C.1 で示したような，入出力装置がすべて同一のバスに接続されているモデルはあくまで概念的なものである。実際には，入出力装置の特性に応じてさまざまな種類のバスが使い分けられる。具体的には，転送データ量の多いものや少ないもの，高速転送の必要なものやそうでもないもの，コンピュータの動作中に自由に取り外しできる必要があるものとそうでないもの等に応じた使い分けがなされる。異なる種類のバスの間は，両方のバスと情報のやり取りの可能な**バスブリッジ** (bus bridge)，**バスアダプタ** (bus adapter) などと呼ばれる回路により結合される。

† 例えば MIPS には syscall 命令が用意されている。

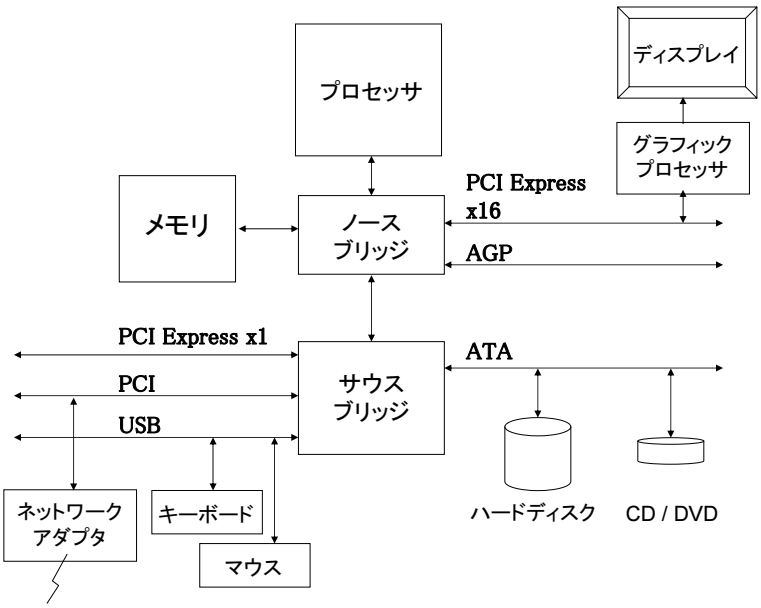



**図 C.2**　典型的なパーソナルコンピュータの入出力構成

典型的なパーソナルコンピュータの入出力の構成を**図 C.2** に示す。さまざまな種類のバスが，ノースブリッジ (north bridge)，サウスブリッジ (south bridge) と呼ばれるバスブリッジ回路を介して接続されている。プロセッサと直結されたノースブリッジには, 転送データ量の多いバスが接続される。例えばメモリバスや，ディスプレイへのグラフィック描画装置 (グラフィックスボード) を接続するための AGP バス，PCI Express バスなどが代表例である。最近はノースブリッジの機能がプロセッサに内蔵されることも多い。サウスブリッジには，ハードディスク・光学ディスク等の補助記憶装置を接続する ATA バス，コンピュータの動作中に取り外し可能な装置を接続する USB，その他さまざまな装置を接続可能な PCI バスなど，ノースブリッジほど速度が必要とされないものが接続される。それぞれのバスにはそれぞれの通信手順 (通信プロトコル) が定められており，プロセッサと入出力装置は，バスブリッジを介してその通信手順に従いながら相互通信を行っていることになる。

この考え方をさらに拡張していくことで，コンピュータどうしの通信が実現できる。例えばイーサネット (Ethernet) と呼ばれるコンピュータ間通信用のバスとコンピュータ内の入出力バスが，**ネットワークアダプタ** (network adapter, network interface) と呼ばれる一種のバスブリッジによって結合され，定められた手順で通信を行う。そこからさらに話を発展させ，多数のコンピュータと，特に，必ずしも同一主体の管理下になく，さらには世界のどこに存在するかすらもわからない不特定多数のコンピュータとの通信を実現するのが 10 章のネットワーク技術であるといえる。

## 章 末 問 題

【1】 MIPS プロセッサを用いたシステムにおいて，ある入力装置からのデータを読み出すことを考える。アドレス 0xa002 0000 の 1 ワードが装置の状態に割り当てられており，その最下位ビットが 1 のときデータの準備ができたことを示すとする。アドレス 0xa002 0004 の 1 ワードが入力データに割り当てられている。データの準備ができるまで待ち続けた後，読み出した値をレジスタ v0 に得るプログラムをアセンブリ言語で書け。ただしレジスタ a0 に値 0xa002 0000 がセットされているとし，レジスタ t0 の値は破壊してよい。

【2】 最大 4 MBytes/s のレートで 4 バイトずつのデータをプロセッサに送ってくる入力装置があるとする。ポーリングによってこのデータを取りこぼしなく受け取りたい。ポーリング 1 回に 100 クロックサイクルが必要であるとし，プロセッサのクロック周波数は 1 GHz とする。実行時間のうちポーリングに費やされる割合を求めよ。ただし $1\ \mathrm{M} = 1000^6$ と考えてよい。

# D オペレーティングシステム

6 〜 7 章では，機械語命令によるプログラムがどのように実行されるかを学んだが，現在の多くのコンピュータではそのようなプログラムが複数，同時並行で動作できるようになっている。各プログラムは物理的に用意されている主記憶装置より大きなメモリ空間を扱うことができ (8 章)，またプログラムどうしで競合せずに入出力装置を扱うことができる (C 章)。本章では，そのような機構を実現するために応用プログラムとハードウェアを仲介するプログラムの集合体であるオペレーティングシステムの基礎を学ぶ。

## D.1 オペレーティングシステムの役割

現在のコンピュータは，利用目的ごとに異なるプログラムを実行することによって，様々な用途に使われている。例えばワードプロセッサや表計算ソフトのように，コンピュータの利用目的に合わせて実行されるプログラムのことを**応用ソフトウェア** (**アプリケーションソフトウェア**, application software, アプリケーションプログラム, application program) と呼ぶ。一方，応用ソフトウェアとハードウェアとの間を仲介することによって，ハードウェアの機能を効率的に使うことを目的としたソフトウェアが用意されており，それらは**システムソフトウェア** (system software) と呼ばれている。

**オペレーティングシステム** (operating system, **OS**, **基本ソフトウェア**) とはシステムソフトウェアの一種である。広義には，電源投入時から停止時までコンピュータのハードウェアを制御しつつ，利用者に対して使い勝手のよい操作方法を提供し，さらには応用ソフトウェアが効率よく動作する環境を構築・維持する多様なプログラムの集合体である。狭義には，応用ソフトウェアが効率よく動作する環境を構築・維持する**制御プログラム** (control program) と呼ばれる部分のみを指して OS と呼ぶ (**図 D.1**)。現在，一般的なパーソナルコンピュータで広く使われている広義の OS として，マイクロソフト社の Windows，アップル社の MacOS，およびフリーソフトウェアである Linux 等が挙げられる。以下の説明では，特に明記しない限り広義の意味で OS という用語を用いる。

OS の主な役割は，ハードウェアの有効活用と使いやすさの向上に大別できる。

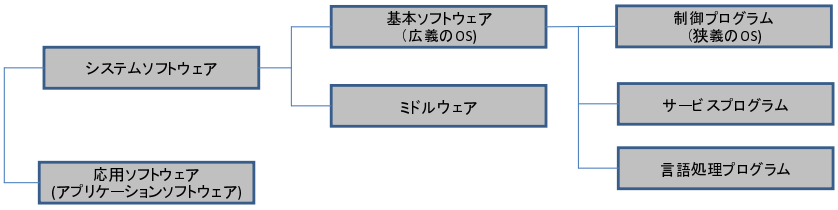



**図 D.1**　ソフトウェアの分類。この章ではシステムソフトウェアの一種である (広義の) オペレーティングシステムについて学ぶ。

### D.1.1　ハードウェアの有効活用

コンピュータの性能を評価する尺度として

- **スループット** (throughput): 単位時間あたりの処理能力のこと。
- **ターンアラウンドタイム** (turnaround time): コンピュータに処理開始の指示をしてから終了までに要する時間のこと。
- **レスポンスタイム** (response time): データ入力から出力開始までに要する時間のこと。**応答時間**とも呼ばれる。
- RASIS (Reliability, Availability, Serviceability, Integrity, Security): コンピュータシステム全体としての信頼性を表す概念であり，「信頼性」「可用性」「保守性」「保全性」「機密性」の頭文字から構成される造語。

などが挙げられる。これらの尺度に基づいて，コンピュータの性能を高めることが OS の最も重要な役割である。高度で複雑なハードウェアの機能を有効活用することによって高い処理性能を達成するため，OS は，メモリ・プロセッサ・入出力装置などのハードウェア資源 (**リソース**, resource) をそれぞれの応用ソフトウェアに割り当てたり，それらの活用状況を監視したりする機能を持っている。例えば，多くの OS は複数の応用ソフトウェアの実行を一度に管理し，プロセッサの使用権を定期的に他の応用ソフトウェアに切り替えることによって，見かけ上同時に実行しているように見せる機能を持っている。これによってプロセッサが何も処理をしていない時間 (**遊休時間**, **アイドル時間**, idle time) を極力削減し，プロセッサの処理能力の有効活用を実現している。

また，コンピュータの補助記憶装置（ハードディスクや DVD など）には，非常に多くのデータを格納することができる。それらを整理して管理するために，**ファイルシステム** (file system) が提供されている。データを**ファイル** (file) という単位で格納し，**ディレクトリ** (directory, **フォルダ**, folder) に分類して管理できるようになっている（**図 D.2**）。それらのデータへのアクセス権限の管理も，OS の重要な役割の一つである。

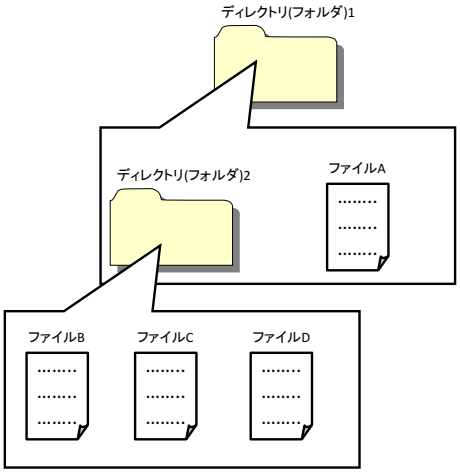



**図 D.2**　ファイルシステム。データはファイル単位で保存されている。ファイルはディレクトリによって階層的に管理されている。

#### D.1.2　使いやすさの向上

現在の多くの OS は，コンピュータの操作画面を直感的にわかりやすい絵（アイコン）などを用いて表現した**グラフィカルユーザインタフェース** (graphical user interface, **GUI**) を提供している。そのようにコンピュータの操作方法を抽象化した結果，利用者はハードウェアの詳細を全く意識することなく，コンピュータを利用できる。このように利用者の使い勝手を向上させることも OS の重要な役割である。

また，現在のコンピュータには膨大な種類のハードウェアが存在し，それぞれが異なる制御方法を必要とする。一方，応用ソフトウェアはなるべく多くの種類のハードウェア上で動作することが望ましい。このため，OS はハードウェアを抽象化し，応用ソフトウェアは OS の仲介によってハードウェアを利用する。このようにソフトウェアを階層的に構成することによって，容易にハードウェアを置き換えたり拡張したりすることができるようになっている。

さらに，OS はコンピュータの状態を必要に応じて記録し，コンピュータの管理業務を支援している。管理者は記録された管理情報 (**ログ**, log) を頼りにして，コンピュータで起こっている事象を把握し，その後の管理業務に反映させることができる。

以下，本章では D.2 節において OS の主要な構成要素について述べたのちに，OS で用いられている特に重要な概念であるリソース管理と仮想記憶についてそれぞれ D.3 節および D.4 節で述べる。

## D.2　オペレーティングシステムの構成要素

広義の OS の構成要素は，制御プログラム，サービスプログラム，および言語処理プログラムに大別される (**図 D.3**)。

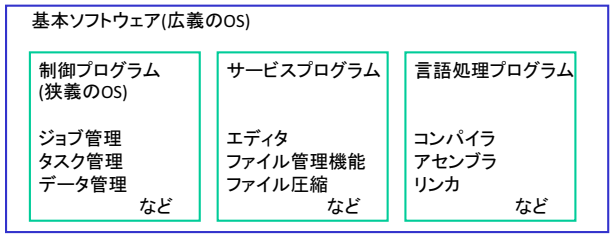



**図 D.3**　オペレーティングシステムの構成要素。広義の OS とは，ハードウェアの機能を有効活用しつつ，利用者に使い勝手の良い環境を提供するための多様なプログラムの集合体である。

### D.2.1　制御プログラム

狭義には，制御プログラムのことを OS と呼ぶ。制御プログラムは多くの機能を持っているが，その中でも特に重要な機能としてジョブ管理，タスク管理，およびデータ管理が挙げられる。**ジョブ** (job) とは，人間から見た仕事の単位である。それに対して，**タスク** (task) とはコンピュータから見た仕事の単位である [†1]。また，データの入出力やファイルの管理がデータ管理 (ファイル管理) 機能である。

（**1**）　**ジョブ管理**　　利用者はジョブ単位でコンピュータに仕事を依頼する [†2]（しばしば「ジョブを投入する」と表現される)。そのジョブの管理において中心的役割を担っているのが，**ジョブスケジューラ** (job scheduler) と呼ばれるプログラムである。ジョブスケジューラは投入されているジョブの中から優先順位の高いジョブを取り出し，その実行に必要なリソースを割り当てて，その実行状況を監視，制御する。その結果，スループットやターンアラウンドタイム，レスポンスタイムといった観点からハードウェアの持つ処理能力の有効利用を実現する。

また，プリンタなどの低速な入出力装置を利用するジョブが複数存在している場合，入出力したいデータを一時的にハードディスク上に保存し，それぞれのジョブは入出力動作の完了を待たずに次の処理へと移る。このような目的で入出力データを一時的に保存するための記憶領域を**スプール** (SPOOL, simultaneous peripheral operation on-line) と呼び，スプールを使った入出力をスプーリング (spooling) と呼ぶ。このスプーリングによってジョブは低速な入出力装置が空くのを待つ必要がなくなり，複数のジョブの実行を効率化することができる。

---

[†1] タスクの厳密な定義は，OS によって異なる。ここではコンピュータから見たときに仕事の単位となるものを総称してタスクと呼び，後述するプロセスやスレッドを含むものとする。

[†2] 一般的に，複数のプログラム（**ジョブステップ**, job step）を実行することによってジョブの実行が完了する。各ジョブステップを実行するために利用者がコンピュータに指示を出さなければならないとしたら，利用者は現在のジョブステップが終了するまで待っていなければならないし，ジョブステップが完了してから次のジョブステップが実行されるまでの間はプロセッサが何も処理をしない時間になってしまう。

（**2**） **タスク管理**　利用者からコンピュータに投入されたジョブは，コンピュータの立場から見ると複数のタスクから構成されているのが一般的である。複数のタスクが円滑に実行されるように，適切な順番やタイミングでリソースを割当て，実行されるタスクを切り替えることをタスクスケジューリングと呼ぶ。**タスクスケジューラ** (task scheduler) は，実行可能状態のタスクの中から最も優先順位の高いタスクを選択し，プロセッサを一定期間使用する権利を与えて実行状態に遷移させる。そのようなタスクスケジューリングを短い間隔で繰り返すことにより，見かけ上複数のタスクを同時実行し，プロセッサの処理能力を休みなく効率的に使うことによってコンピュータのスループットを向上させる。そのような機能をマルチプログラミング (マルチタスク, マルチプロセス) と呼ぶ。

（**3**） **データ管理**　OS の重要な機能の一つとして，記憶装置に格納されているデータを管理する機能が挙げられる。例えば，コンピュータ利用者はハードディスクの物理的な構成を全く考慮することなく，ファイル単位でハードディスク上にデータを格納することができる。格納されたファイルはディレクトリによって階層的に分類され，管理されている。また，OS によっては複数人で共用してコンピュータを利用する場合を考慮して各ファイルの所有者が明確に区別されており，そのファイルの閲覧や変更などの**アクセス権限** (permission, access control) も設定されている。この機能は，一般利用者と管理者の権限を区別し，OS の重要なファイルを保護するという意味でも重要な役割を果たしている。

また，もしも記憶装置上に格納されているデータの入出力のために，その記憶装置のハードウェアを直接制御するようなプログラムを記述する必要があるとすれば，そのソフトウェア開発には多大な労力を要する。労力のみならず，記憶装置の種類や型番の変更に伴ってプログラムのデータ入出力部分を修正する必要があり，そのように記述された応用ソフトウェアは特定の記憶装置でしか動作しない（汎用性や移植性を損なう）という点でも深刻な問題である。このため，記憶装置上のデータに対する入出力を抽象化し，具体的なハードウェア制御に関しては OS が担当するようになっている。応用ソフトウェア開発者は，OS が提供しているデータへのアクセス方法を利用することによって，記憶装置の変更の影響を受けない汎用的なプログラムを作成することができる。

なお，上記の例のようにOSが応用ソフトウェアに対して機能を提供する場合には，**システムコール** (system call, **スーパーバイザコール**, supervisor call) と呼ばれる関数群として提供されている。すなわち，応用ソフトウェアのプログラム中からシステムコールを呼び出すことによって，OSが提供している機能を利用することができる。また，システムコールは抽象度の低い機能しか提供しないため，それらを容易に利用するための抽象度の高い関数群やさまざまな補助関数群を揃えたものが**ライブラリ関数** (library function) として提供される。システムコールおよびライブラリ関数の仕様は，アプリケーションプログラムとOSとの間のインタフェースを定める規約と捉えることができる。そのような規約は **API** (application programming interface) と呼ばれる。

### D.2.2　サービスプログラム

サービスプログラムはユーティリティプログラムとも呼ばれ，コンピュータの利用を支援するためにOSに標準で付属するプログラムの総称である。そのOSを使ううえで大半の利用者が必要とし，頻繁に実行するようなプログラムが含まれている。サービスプログラムとして付属するプログラムの種類はOSによって大きく異なるが，多くのOSに付属する代表的なサービスプログラムとしては以下のものが挙げられる。

- エディタ: ファイルを編集するために使われるプログラム
- ファイル管理: ファイルのコピーや削除などを行うプログラム群
- ファイル圧縮: ファイルの容量を削減するためのデータ圧縮とその伸張を行うプログラム

### D.2.3　言語処理プログラム

9章で詳しく説明した，コンパイラ，アセンブラ，リンカ等の言語処理プログラムも，広義のOSの一部として挙げられる。

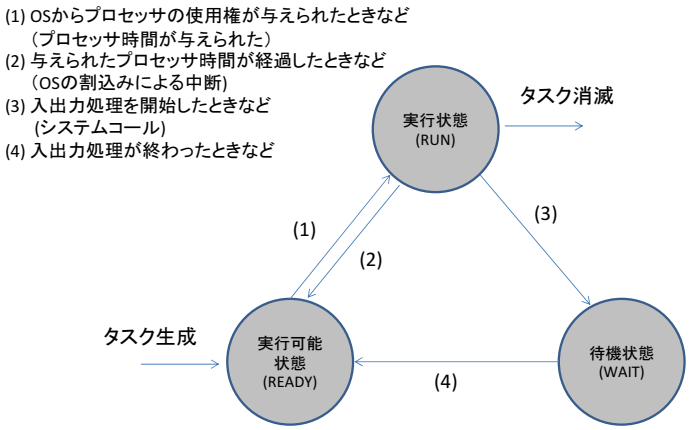



**図 D.4**　タスクの状態遷移

## D.3　マルチプログラミングと割込み

**マルチプログラミング** (multi programming, **マルチタスク**, multi task, **マルチプロセス**, multi process) とは，1 台のコンピュータで複数のタスクをあたかも同時に実行しているように見せる機能である。例えば，あるタスクがユーザからの入力待ちのためにプロセッサを使わない時，プロセッサを使用する権利を他のタスクに切替えることでプロセッサの使用効率を高めることができ，結果としてコンピュータのスループットの向上を期待できる。また，別の例として，ウェブブラウザとワードプロセッサを同時に実行することにより，ウェブ画面を見ながら文書作成をすることが可能になり，ユーザの利便性や作業効率の向上が期待できる。

マルチプログラミングの機能を有する OS では，複数のプログラムが主記憶装置上に読み込まれ，いつでも実行可能な**実行可能状態** (**ready 状態**) になっている。それらのタスクの中から一つのタスクが選択され，ある一定期間だけプロセッサを使用する権利が与えられる（しばしば「**プロセッサ時間** (processor time, **CPU 時間**, CPU time) が与えられる」と表現される)。与えられたプロセッサ時間を使って実際に実行されているタスクのことを，**実行状態** (**run 状態**) のタスクと呼ぶ。また，入出力待ちなどの要因により，一時的に処理を先に進めることができない状態にあるタスクのことを，**待機状態** (**wait 状態**) のタスクと呼ぶ。タスクは，その実行が完了するまで上記の 3 つの状態の間を遷移する（**図 D.4**)。

タスクスケジューラは，実行可能状態のタスクの中から最も実行の優先順位の高いタスクを選び，一定期間だけ実行状態に遷移させる。このようにタスクを短い時間間隔で切替えて実行する時分割実行により，コンピュータのスループットを向上させることができる。タスクを選択するタスクスケジューリング方式としては，実行可能状態のタスクを単に順番に一定時間ずつ実行する**ラウンドロビン** (round robin) 方式から，複数のプロセッサをなるべく平等に使用するスケジューリング，締切までの時間の短いタスクを優先的に実行するスケジューリングなど，そのコンピュータで重要視される性能指標に合せて様々な方式が使用される。

実行中のタスクをなんらかの理由で中断し，他のタスクを実行することが必要になったとき，OS は C.2 節で述べた **割込み**を用いる。例えばマルチプログラミングでは，タイマ装置から一定周期で発生するタイマ割込みによって実行中のタスクを一時中断し，OS 内のタスクスケジューラに制御を移すことで，他のタスクに実行を切替える。中断されたタスクは，タスクスケジューラから再度選択された際に，中断された時の状態から実行を再開する。また，タスクが入出力を必要とするとき，タスクは OS に対してシステムコールを行い，入出力を行うことを伝える。C.2 節で述べた通りシステムコールは割込みを引き起こし，これによって処理が OS に移り，入出力が開始される。一般的に入出力には時間がかかるため，OS は別のタスクに実行を切替える。入出力が完了すると，機器から入出力割込みが発生することで OS に制御が移り，スケジューラによって中断されたタスクが再開される。

割込みは，割込まれたタスクとの関係によって**内部割込み** (internal interrupt) と**外部割込み** (external interrupt) に分類される。内部割込みは割込まれるタスクが要因となって生じるのに対して，外部割込みはそれ以外の外部要因によって生じる。割込みの種類を**図 D.5** に示す †。

通常，コンピュータのプログラムは実行ファイルとして補助記憶装置に保管されている。このプログラムが実行されるとき，その実行ファイルは主記憶装置に読み込まれ，メモリ領域やプロセッサ時間などのリソースが割当てられて実行可能な状態になる。このように，OS はリソースを確保して**プロセス** (process) と呼ばれる実行単位を生成して管理する。ファイルやメモリ領域へのアクセス権限なども，プロセス単位で管理されている。このため，例えばあるプロセスが不正なメモリ操作をしたとしても，他のプロセスに割当てられているメモリ領域のデータを壊すことはないようになっている（メモリ保護)。

† ここでは原理原則的な意味に基づいて用語を分類した。実際の各システムにおける用語の定義はさまざまである。極端な例として，MIPS ではこれらすべてを**例外** (exception) と呼ぶ。他には，内部・外部に関わらず例外的な事象により生じるものだけを例外と呼ぶ場合や，内部割込みをすべて例外と呼ぶ場合などがある。システムコール割込みだけを指して**ソフトウェア割込み** (software interrupt) と呼ぶ流儀や，逆にこの図における例外のみを指してプログラム割込みと呼ぶ流儀もある。

**内部割込み（ソフトウェア割込み）**：

**例外** ゼロ除算，オーバーフロー，メモリ保護違反など，プログラムの実行中に例外的事象が発生したときに生じる割込み

**システムコール割込み（スーパーバイザコール割込み）** 入出力を開始するときなど，プログラムが OS のサービスを呼び出すために自発的に発生させる割込み

**外部割込み（ハードウェア割込み）**：

**マシンチェック割込み** ハードウェアの異常，誤動作が発見された時に生じる割込み

**入出力割込み** 入出力処理が完了したり，誤動作したときに生じる割込み

**タイマ割込み** タスクスケジューラに定期的に制御を移すことなどを目的として，一定時間が経過するたびに生じる割込み

**コンソール割込み** ユーザがコンソールから介入したときに生じる割込み

**図 D.5** 割込みの種類。プログラム自身が発生させる内部割込みと，外的要因から発生する外部割込みに大別される。

プロセスよりも細かい単位として**スレッド** (thread) がある。プロセスがプログラムの実行に必要なメモリ領域やプロセッサ時間などのすべてのリソースを割当てられた実行単位であるのに対して，スレッドはプロセッサ時間だけを割当てられた実行単位である。プロセスは複数のスレッドを生成することができ，それらは同一のメモリ空間を共有する。複数のスレッドにそれぞれプロセッサ時間を割当てることにより，近年のコンピュータはそれらのスレッドを複数のハードウェアで同時に実行 (並列実行) することが可能である。プロセスの実行に必要な処理を複数のスレッドに分割し，複数のハードウェアで手分けして並列実行することにより，すべての処理を逐次に実行するよりも高速に実行できる。このような複数のスレッドを使った並列処理のことを，**マルチスレッド処理** (multi-thread processing, multi threading) と呼ぶ。

## D.4 仮 想 記 憶

コンピュータに搭載されている主記憶装置の容量は有限である。しかし，現在の OS で管理されたコンピュータでは，実際に搭載されている主記憶装置の容量を超えるプログラムを作成したり実行したりすることができる。このような機能を実現しているのが，**仮想記憶** (virtual memory) である。

仮想記憶方式の基本的な考え方は，実際の主記憶装置 (実記憶装置) に加えて補助記憶装置も記憶領域として使って仮想的に大きなメモリ空間を構成し，OS が適切にデータを管理することによって，必要なデータが常に実記憶装置にあるかのようにプログラムに見せるということである。以下，実記憶装置の記憶領域を実記憶空間 (実空間) と呼び，そのアドレスを**実アドレス** (real address, **物理アドレス**, physical address) とする。一方，仮想記憶のアドレスを**仮想アドレス** (virtual address, **論理アドレス**, logical address) と呼ぶ。プログラムはデータにアクセスする際に仮想アドレスを使うが，実記憶装置は実アドレスを使っている。このため，実記憶装置にアクセスするためには，仮想アドレスを実アドレスに変換する必要がある。このアドレス変換を行う機構を，**動的アドレス変換機構** (Dynamic Address Translator, **DAT**) と呼ぶ。仮想アドレスと実アドレスとの対応は，動的アドレス変換機構が管理している。プログラムが仮想アドレスを使ってデータにアクセスする場合，実際にはその仮想アドレスに対応する実アドレスのデータがアクセスされる。仮想記憶は実記憶よりも大きいため，仮想アドレスの一部だけは実アドレスと対応づけられているが，実記憶装置の容量を超える領域の仮想アドレスは補助記憶装置上の場所に対応づけられている (**図 D.6**)。

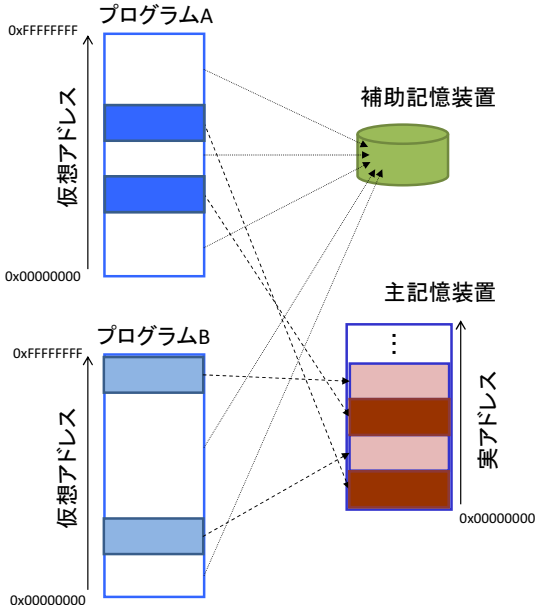



**図 D.6**　仮想アドレスと実アドレス。仮想空間の一部のみが主記憶装置上に対応付けられている。対応付けは DAT によって動的に管理されている。

前述のように，プログラムは仮想アドレスを使って仮想領域にあるデータにアクセスする。仮想アドレスは補助記憶装置上の場所を指している可能性があり，アクセスされたデータは物理的には補助記憶装置上に保管されているかもしれない。この場合，OS はプログラムの実行を一時中断し，そのプログラムがアクセスしようとしているデータを実記憶装置に移動する。このとき，補助記憶装置上の場所と対応づけられていた仮想アドレスが実アドレスと対応するように，動的アドレス変換機構がアドレスの対応付けを変更する。その結果，プログラムが再開後に同じ仮想アドレスにアクセスするとき，その仮想アドレスは物理アドレスに変換され，実記憶装置上に移動されたデータにアクセスすることになる。

仮想記憶方式には，大きく分けてセグメント方式とページング方式がある。

**セグメント** (segment) 方式は，仮想記憶領域をプログラムコードやデータ，スタックなどの意味がある単位 (セグメント) に分割して管理する。仮想アドレスは，セグメント固有の番号とそのセグメント内での相対アドレスで表現される。各セグメントが記憶されている場所はセグメントテーブルと呼ばれる表で管理されている。プログラムがアクセスしようとしたセグメントが実記憶装置上にない場合には，補助記憶装置からそのセグメントを移動 (ロールイン) し，逆に不要なセグメントを補助記憶装置に移動 (ロールアウト) する。

**ページング** (paging) 方式では，仮想記憶領域がその内容とは無関係に一定の大きさ (例えば 4KB) に分割される。分割された領域の一つ一つを**ページ** (page) と呼ぶ。仮想アドレスは，ページ固有の番号とページ内の相対アドレスで表現される。仮想アドレスからページの物理アドレスを求めるのは，動的アドレス変換機構の役目である。プログラム実行中にアクセスされた仮想アドレスのページが実記憶装置上にないことがわかったとき，動的アドレス変換機構は**ページフォルト** (page fault) という割込みを発生させてプログラムを一時中断し，必要なページを実記憶装置に移動 (ページイン) する。このとき，不要なページを補助記憶装置に移動 (ページアウト) する。不要なページを選択する方法としては，**FIFO** (first-in first-out) 方式や **LRU**(least recently used) 方式などがある。前者は最も早くページインしたページをページアウトする方式であり，後者は最も長い間使われなかったページをページアウトする方式である。

## 章　末　問　題

【１】 以下の用語について説明せよ。

(1) 仮想記憶

(2) マルチプログラミング

(3) プロセスとスレッド

(4) 割込み

(5) ファイルシステム

# E プロセッサの実現

6 〜 7 章では，コンピュータの構成を，命令セットアーキテクチャのレベルで概観した。すなわち，そこで出てきた ALU や汎用レジスタ等の構成要素がどのような回路で出来ているか，また，それらがどのような回路で制御されているかなどについてはブラックボックスとして扱い，動作のみに注目して学んだ。これに対して本章では，これらの構成要素の具体的な構造やその制御のしかたを見ていくことで，マイクロアーキテクチャのレベルでコンピュータがどのように動作するかを学ぶ。

本章のもう一つの目的は，本書の前半で学んだ論理代数，組合せ回路，順序回路といった基礎項目が，プロセッサの設計にどのように生かされるかを明らかにすることにある。読者によっては，6 〜 7 章の内容がそれより前の内容から突然飛躍したように感じる場合もあったかも知れない。本章でその飛躍の間を埋めることで，これらの基礎的な項目と実際のプロセッサの設計が地続きであることを示す。

MIPS は比較的簡潔な命令セットをもつプロセッサであるが，それでもその全体の設計を詳細に見ていくのは本書の範囲を大きく超えている。そのため，MIPS の命令セットのうちごく一部のみを持つ簡易プロセッサを考えることにする。特に，組合せ回路や順序回路としての設計を手計算で行える範囲に留めることに主眼をおいている。

## E.1 命令セット

MIPS の命令セットを以下のように簡略化する †。

- 算術論理演算はレジスタ-レジスタ間の addu, subu, and, or, nor のみで，即値は取らない
- 分岐命令は beq のみ
- メモリはプログラムの格納にのみ用いる。つまりロード・ストア命令は存在しない。データはすべてレジスタの中だけで考える
- 命令はメモリから必ず 1 クロックサイクルで読み出せる

† 命令の追加については E.7 節および章末問題にて議論する。

命令の動作は MIPS と同様とし，汎用レジスタ群も MIPS と同じものを備えるとする。割込みは考慮しない。

## E.2 全 体 構 成

プロセッサ全体の構成を**図 E.1** に示す。命令を処理する回路を IF，ID, EX の 3 つに分解して考える。各部の詳細は順次説明していくとして，ここでは概要のみを説明する。

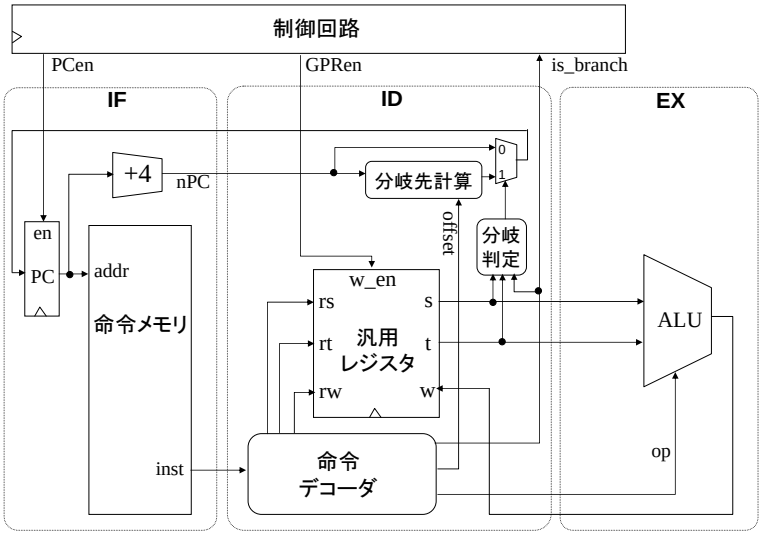



**図 E.1**　簡易プロセッサの全体構成。クロック信号の配線は省略している。本章の他の図も同様に表す。

IF 部は命令フェッチ (Instruction Fetch) を行う。すなわちプログラムカウンタ (PC) で示されたアドレスの命令を命令メモリから読み出す。

ID 部は命令デコード (Instruction Decode) と汎用レジスタ読み出しを行う。分岐命令の場合は，分岐判定・分岐先計算もここで行う。

EX 部は演算の実行 (EXecution) を行う部分であり，ALU による算術論理演算を行い，結果を汎用レジスタに書き戻す。

回路の遅延を考慮し，信号が IF，ID，EX 各部を流れるにはそれぞれ最大で 1 クロックサイクルを要するとする。演算命令は，IF - ID - EX の 3 クロックサイクルで実行され，分岐命令は，IF - ID の 2 クロックサイクルで実行される。

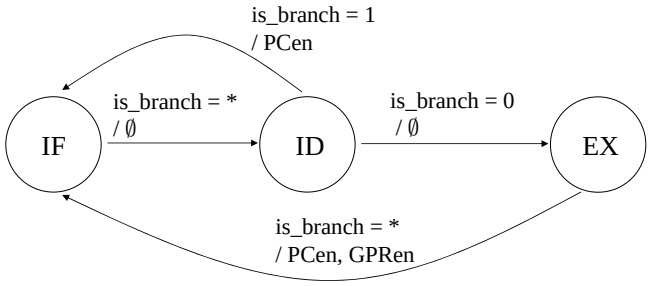



**図 E.2** 簡易プロセッサの状態遷移図。入力信号の * は，0 の場合と 1 の場合の遷移が同一であることを意味している。出力信号は 1 になるもののみが記載されており，∅ はすべて 0 であることを表す。

このような実行制御を 3 状態の有限状態機械で表すと，**図 E.2** のような状態遷移図となる。IF，ID，EX 各部の動作に対応させて IF 状態，ID 状態，EX 状態を設ける。入力 is_branch は，命令が分岐命令であれば 1，そうでなければ 0 となる信号であり，ID 部の中の命令デコーダにより生成される。このような状態遷移を実現するような順序回路が，この簡易プロセッサの制御回路となる。制御回路への入力 is_branch が制御される側の回路によって生成されるのが奇妙に見えるかも知れないが，何ら問題はない。出力信号 PCen, GPRen は各種レジスタへの書き込みタイミングを指定するものであり，詳細は E.4 節で説明する。

## E.3 構 成 要 素

以下では，まずいくつかの基本的な回路部品について整理した後で，主要な構成要素である命令メモリ，汎用レジスタ，ALU について述べる。

### E.3.1 基本的な回路部品

（1） **書き込みイネーブルつきレジスタ** 5 章で学んだ通り，**D フリップフロップ**を $n$ 個並べると $n$ ビット**レジスタ**として用いることができる。本章では立ち上がりエッジトリガ型の D フリップフロップを用いる。したがって，$n$ ビットの入力信号が，クロック信号が立ち上がるごとに記憶される。しかし，プロセッサのように複数のレジスタが存在してさまざまな役割を果たすシステムでは，クロックが立ち上がる度に毎回記憶内容が書き換わってしまうのではなく，明示的に指定したときにだけ記憶内容が書き換わるような機能があると便利である。

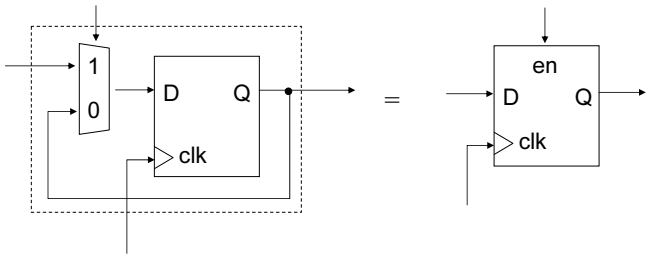



**図 E.3**　書き込みイネーブルつき D フリップフロップの構成 (左) とその回路シンボル (右)

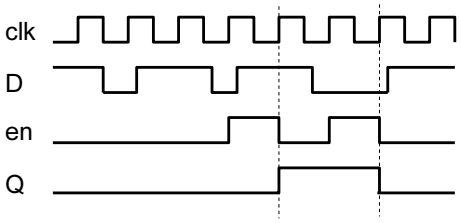



**図 E.4**　書き込みイネーブルつき D フリップフロップの動作タイミングチャート。図中にクロックの立ち上がりは 8 回あるが，そのうち en 信号が 1 になっている 2 回のみ，実際に Q の更新が行われる。

そのような機能を持たせるためには，例えば**図 E.3** のような方法がある。D フリップフロップからの出力と外部入力のどちらかをマルチプレクサで選択したものを改めて入力としている。マルチプレクサの制御入力が 0 のときは，現在の出力を再び記憶するため記憶内容は変化しない。制御入力として 1 が与えられた直後のクロック立ち上がりのタイミングでのみ，外部入力の値で記憶内容が更新される。動作のタイミングチャートを**図 E.4** に示す。

マルチプレクサの制御入力は，フリップフロップへの書き込みの有効・無効を指定する役割を果たすため，書き込みイネーブル (write enable) 信号，あるいは単に**イネーブル信号** (enable signal) と呼ぶ。書き込みイネーブル機能を持つ D フリップフロップは，しばしば図 E.3 の右側のようなシンボルで記載される。en がイネーブル信号を表す。

このような D フリップフロップを $n$ 個並べ，すべての en 端子を共通の書き込みイネーブル信号に接続することで，書き込みイネーブルつき $n$ ビットレジスタを構成することができる。本章の図の中で，クロック入力と en 入力を持つ四角形として描かれているものは，すべてそのようなレジスタを表す。

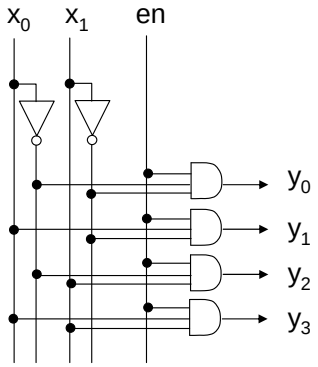



**図 E.5**　出力イネーブルつきデコーダの構成

（**2**） **出力イネーブルつきデコーダ**　3 章で導入した **2 進デコーダ**においても，常に 2 進数のデコード結果を出力するのではなく，明示的に指定されたとき以外は全出力を 0 に固定しておきたいときがある。

例えば，デコーダの各出力が複数のレジスタそれぞれの書き込みイネーブル信号に接続されている構成を考える。この構成により，1 つのレジスタだけを選択して書き込むことが可能となるが，状況によってはどのレジスタにも書き込みたくないこともある。このような場合は，デコーダ回路に出力イネーブル信号 en を設け，en が 0 のときは全出力が 0 になるようにすると便利である。デコーダにそのような機能を加えるには，出力段のすべての AND ゲートの入力に en 信号を接続するだけでよい。出力が 4 本の場合の構成を**図 E.5** に示す。

### E.3.2　命 令 メ モ リ

メモリの内部的な構造については 8 章で述べた。ここではそれを命令メモリとして使う場合に，外部からどう見えるかについてのみ言及する。

プログラムの実行時は，命令メモリは読み出し専用と考えてよいため，書き込み動作は考慮しない。一般には，読み出したいアドレスを入力信号として与えるとともに，読み出しを指示する制御信号を与えることで，そのアドレスに格納された命令が出力される。ただし本章では，命令メモリからは常に 1 クロックサイクルで命令が読み出せると仮定しているため，制御信号については省略し，図 E.1 に示したように「アドレス信号を与えれば，1 クロックサイクル内に命令が出てくる」回路であると扱ってよい。このように省略して考える限りは一種の組合せ回路だとみなせることになる †。

常に 1 クロックサイクルで読み出せるという仮定は，キャッシュメモリが常にヒットすると仮定していることに相当する。

† もちろん内部構造としては記憶を含むため，あくまでも順序回路である。書き込み機能を無視し，複雑な制御動作を省略して考えることで，組合せ回路のようにみなせるというだけであることに注意する。

### E.3.3 汎用レジスタ

MIPS には 32 ビットの汎用レジスタが 32 本用意されている。このようにレジスタが多数集まった構造は**レジスタファイル** (register file) と呼ばれる。

MIPS の場合，2 項演算が基本なので 2 本のレジスタを同時に読み出すことができるようになっている。書き込みは 1 本のレジスタのみを指定する。具体的には，図 E.1 に示すように入力 rs, rt, rw, w および w_en と，出力 s, t を持つ回路ブロックとなる。読み出しレジスタ番号 rs，rt を入力するとそれに対応して各レジスタの値 s，t が出力される。一方，書き込みレジスタ番号 rw を入力するとともに書き込みイネーブル信号 w_en を 1 にすると，クロック信号立ち上がりのタイミングで対応するレジスタに入力データ w が書き込まれる。

レジスタファイルはその名のとおり D フリップフロップによるレジスタを多数並べることで作るのが基本である。実際上は，ある程度以上大規模なレジスタファイルは SRAM 技術を用いて作ることが多いが，ここでは基本通り作ることにしよう。

レジスタファイルの構造は**図 E.6** のようになる。この図では 0 番レジスタも区別せずに示しているが，実際の MIPS の 0 番レジスタは常に値 0 を出力する。従ってこれは実際にはレジスタではなく，単に定数 0 を出力し，入力を無視する回路で置き換えればよい。

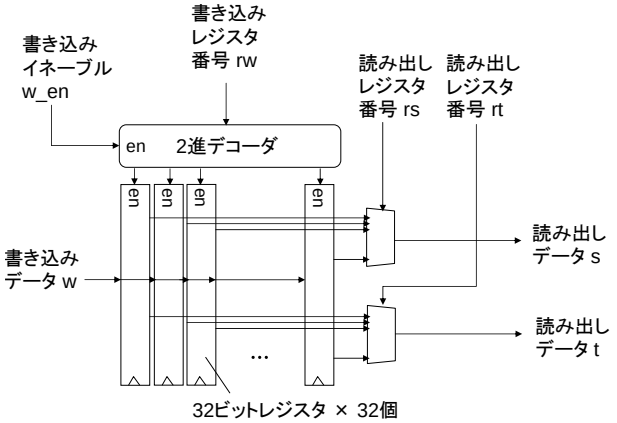



**図 E.6** 汎用レジスタの構造 (レジスタファイル)

指定されたレジスタ 2 本の出力を行うため，32 対 1 の 32 ビット幅マルチプレクサを 2 つ用意し，両方の入力に 32 本のレジスタの出力それぞれを接続する。各マルチプレクサの制御入力として rs，rt を与えればよい。

入力に関しては，まず入力データ w をすべてのレジスタに接続する。この状態で，書き込みたいレジスタのみのイネーブル信号を 1 にすればよい。各レジスタのイネーブル信号を作るため 2 進デコーダを用いる。5 ビットの書き込みレジスタ番号 rw がデコードされ，32 本のデコーダ出力のうち 1 本のみが 1 になる。さらにデコーダ自身の出力イネーブル信号に w_en を接続する。これにより，w_en がのときのみ書き込みが行われるようになる。

### E.3.4 ALU

32 ビットの入力を 2 つ受け取り，各種の演算結果を 32 ビット値として出力する **ALU** を用意する。演算の種類は入力 op によって指定する。

この章で設計する簡易プロセッサでは，加算，減算，ビットごとの and，or，nor の合計 5 種類の演算を行う。これらを区別するため，op は少なくとも 3 ビット必要である。2 つの 32 ビット値と op が定まれば出力が一意に定まるため，ALU は 67 (= 32 + 32 + 3) ビット入力，32 ビット出力の組合せ論理回路となる。

op の値と演算の種類は，任意に対応付けて構わない。そうであれば，できるだけ命令デコードが簡単になるように，命令エンコード表と見比べながら op を定めるのがよい †。少し先回りして表 表 E.1 を見ると，命令の下位 3 ビットをそのまま op として使うのが一番簡単そうだとわかる。

以上の仕様に基づく ALU を実現する最も安直な構成を**図 E.7** に示す。32 ビットの加算器，減算器，ビットごと and，or，nor 回路のそれぞれを用意し，その出力をマルチプレクサに入力して選択信号 op に基づいて選べばよい。

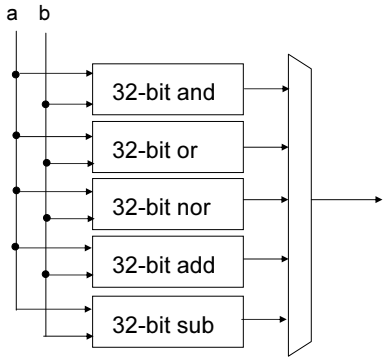



**図 E.7**　ALU の構成 (その 1)

ビットごと and，or，nor は，入力のうちペアになるビットを and，or，nor の各ゲートにつないで出力を取り出すだけの回路である。ここで，nor が or の出力を反転したものであったことに気をつけると，回路を一部共有化できることがわかる。

† より厳密に考えると，命令デコードを簡単をした分 ALU の回路が複雑になってしまうということもあり得るため，全体のバランスを考慮する必要がある。

加算器の構成方法については 3 章で既に述べた。その際，加算器にわずかな補助回路を加えることで，減算器を構成できることも指摘した (3 章の章末問題【 **6** 】)。このことを利用すると，加算器と減算器で回路を共有できる。要点は減算を $a-b=a+(-b)=a+\overline{b}+1$ のように置き換えられることであった。ここで $\overline{b}$ は $b$ の各ビットを反転したものを表す。まず入力 $b$ の側に，加算か減算かに応じてビットの反転・非反転を切り替えることのできる回路を取りつける。残る 1 の加算のためには，最下位ビットの全加算器へのキャリー入力 $c_0$ に 1 を入力すればよかったので，加算か減算かに応じてこの入力を 0 または 1 に切り替える。

これらの工夫の結果，ALU は**図 E.8** のような回路で実現できる。信号 $s_0$ により入力 $b$ のビット反転と，最下位ビットへのキャリー入力を行う。演算回路は加算器，ビットごと and, or，nor の 4 種類となったので，それらの出力の選択は 2 ビットの選択信号で行える。これを $s_2s_1$ としよう。ALU 回路を完成させるには，これらの信号 $s_2$，$s_1$，$s_0$ を op から生成する組合せ回路があればよい。

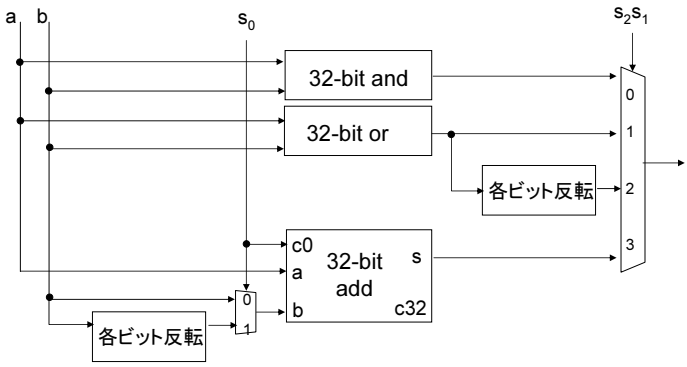



**図 E.8**　ALU の構成 (その 2)。OR と NOR 回路と，加算器・減算器のそれぞれで回路を一部共有している。

---

**例題 E.1**　$\mathrm{op}_2$，$\mathrm{op}_1$，$\mathrm{op}_0$ の 3 ビットの入力から，信号 $s_2$，$s_1$，$s_0$ をそれぞれ生成する 3 つの組合せ回路を設計したい。これらのそれぞれをカルノー図で表し，最も簡単な積和型論理回路で実現せよ。ただし，$s_0$，$s_1$，$s_0$ のそれぞれを独立に簡単化してよい。

---

**【解答】**　まず $s_0$ について考える。op が命令の下位 3 ビットだったことを思い出しながら **表 E.1** を参照すると，$\mathrm{op}_2\ \mathrm{op}_1\ \mathrm{op}_0$ が 001 のとき (addu) は $s_0$ を 0 とし，011 のとき (subu) は $s_0$ を 1 とする。その他の場合はドントケアでよい。

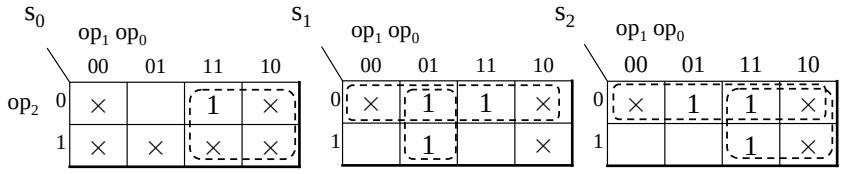



**図 E.9**　$s_0$，$s_1$ および $s_2$ のカルノー図

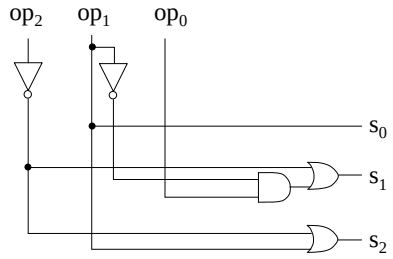



**図 E.10**　op 信号から $s_0$，$s_1$ および $s_2$ の生成回路

次に $s_2s_1$ については，$\mathrm{op}_2$ $\mathrm{op}_1$ $\mathrm{op}_0$ が

- 100 (and) のとき：$s_2s_1 = 00$
- 101 (or) のとき: $s_2s_1 = 01$
- 111 (nor) のとき: $s_2s_1 = 10$
- 001 (addu) または 011 (subu) のとき: $s_2s_1 = 11$
- それら以外のとき: ドントケア

となる。これらをカルノー図で表すと**図 E.9** のようになる。

カルノー図に基づいて簡単化することで

$$s_0 = \mathrm{op}_1$$

$$s_1 = \overline{\mathrm{op}_2} + \overline{\mathrm{op}_1} \cdot \mathrm{op}_0$$

$$s_2 = \overline{\mathrm{op}_2} + \mathrm{op}_1$$

を得る。回路図は**図 E.10** のようになる。

◇

## E.4　各部の動作と構成

命令メモリ，レジスタファイル，ALU といった主要な構成要素が用意できたので，それらの間の情報の流れを見ていくことにする。本章の簡易プロセッサは，既に述べたとおり IF，ID，EX の 3 つの部分で構成される。これら各部の動作を順に見ていく。

信号の流れを明確に述べるために，いくつかの記法の約束をしておこう。

配線上を流れたりレジスタに格納されたりする信号は，英数字からなる信号名で表される。多ビット信号の場合，そのうち一部のビットを signal[0]，signal[1] や，signal[15:10] などのように表す。signal[0] は 多ビット信号 signal の LSB であり，signal[1] は LSB の次のビットである。signal[14:9] は，signal のうち最下位から数えて 10 ビットめから 15 ビットめまでの 5 ビットの信号を表す (ビット番号を 0 から数えていることに注意する)。

信号 $y$ が信号 $x$ に接続されていることを，$y = x$ と表す。これと似ているが，レジスタ $r$ に信号 $x$ が書き込まれることを $r \leftarrow x$ と表す。前者は組合せ回路的な関係を表しており，クロック信号のタイミングとは関係ない。一方で後者は，クロック信号の立ち上がりのタイミングでのみ書き込みが行われることに注意する。本章で出てくるレジスタはすべて書き込みイネーブルつきなので，すべて

$$\mathrm{REG} \leftarrow \mathrm{value} \quad \textbf{if}\ \mathrm{REGen}$$

の形で現れる。イネーブル信号 REGen が 1 のときのクロック立ち上がりでのみ書き込みが行われることを表す。

memory(addr) でアドレス addr のメモリの内容を表す。同様に GPR(num) で num 番汎用レジスタの内容を表す。ALU(op, a1, a2) で，ALU に a1，a2 を入力し演算 op を行った際の結果を表す。

### E.4.1　IF (命令フェッチ) 部

IF 部では，プログラムカウンタ PC の値がアドレスとして命令メモリへ入力される。メモリから読み出された命令 inst は ID 部に引き渡される。並行して，現在の PC から 1 命令分，つまり 4 バイト進んだところのアドレスが計算され，信号 nPC として ID 部に引き渡される。分岐命令が成立する場合以外は PC をこの値に更新することになるが，この時点ではまだ命令がデコードされていないため分岐がどうなるかはわからない。よって PC はまだ更新しない。

$$\mathrm{inst} = \mathrm{memory}(\mathrm{PC})$$

$$\mathrm{nPC} = \mathrm{PC} + 4$$

### E.4.2　ID (命令デコード) 部

**(1)　命令エンコード表**　表 E.1 に，今回実現する各命令が MIPS でどのように 2 進符号化 (エンコード) されているかを示す。本章の設計でもこのエンコーディングをそのまま採用する。

**表 E.1**　命令エンコード表

| | 31–26 | 25–21 | 20–16 | 15–11 | 10–6 | 5–0 |
|---|---|---|---|---|---|---|
| addu | 000000 | rs | rt | rd | 00000 | 100001 |
| subu | 000000 | rs | rt | rd | 00000 | 100011 |
| and | 000000 | rs | rt | rd | 00000 | 100100 |
| or | 000000 | rs | rt | rd | 00000 | 100101 |
| nor | 000000 | rs | rt | rd | 00000 | 100111 |

| | 31–26 | 25–21 | 20–16 | 15–0 |
|---|---|---|---|---|
| beq | 000100 | rs | rt | offset |

（**2**） **命令デコーダ**　命令レジスタ IR に格納された機械語命令は，**命令デコーダ**により解釈され，それによって次の動作が決定される。今回の設計においては，IR の値から

- rs, rt, rw: レジスタ番号
- offset: 定数オフセット
- op: ALU が実行する演算の種類
- is_branch: 命令の分類 (演算命令か，分岐命令か)

のそれぞれを生成するのが命令デコーダの仕事である。

読み出しレジスタ番号 rs と rt は，R 型命令も I 型命令も命令中の固定フィールド rt，rs からそれぞれ切り出してくるだけでよい。書き込みレジスタ番号 rw は R 型命令のみで用いるが，この場合も固定フィールド rd から切り出せばよい†。結局，命令の種類判定が終わっているかどうかとは関係なく，単に各位置から 5 ビットずつを切り出してくるだけでよい。

I 型命令で用いる定数 (即値) フィールドに関しても同様にその位置は固定であるため，単純に切り出せばよい。今回は即値演算命令が存在しないため，beq 命令の分岐先の指定にのみ用いる。演算種類 op についても同様に固定位置から切り出せる。

したがって，各信号は以下のように生成する。

$$\mathrm{rs} = \mathrm{inst}[25{:}21]$$

$$\mathrm{rt} = \mathrm{inst}[20{:}16]$$

$$\mathrm{rw} = \mathrm{inst}[15{:}11]$$

$$\mathrm{offset} = \mathrm{inst}[15{:}0]$$

$$\mathrm{op} = \mathrm{inst}[2{:}0]$$

† なお，ロード・ストア命令を実現する場合は rd ではなく rt フィールドを読み出さなくてはならない。章末問題で扱う。

このように，命令長がすべて 32 ビットで固定であること，また命令フォーマットが規則的であることは，命令デコードを容易にしている。

残るは命令の分類を表す信号である。一般のプロセッサでは比較的複雑な組合せ論理回路となるところだが，今回はレジスタ間演算と分岐命令を区別する信号 is_branch を生成するだけでよい。表 E.1 を参照し，これら以外の命令が現れることはない (すなわちドントケア) と仮定すると

$$\text{is\_branch} = \text{inst}[28]$$

とするだけで得ることができる。

以上の通り設計した命令デコーダを**図 E.11** にまとめる。今回は単に配線の取り回しだけで済んでしまったが，一般には組合せ論理回路になる。

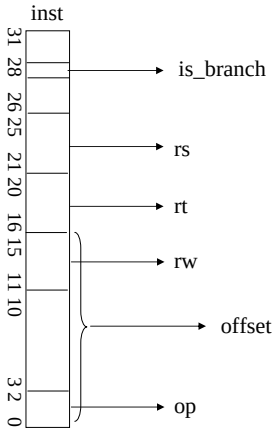



**図 E.11**　命令デコーダの構成

**(3) レジスタ読み出し**　命令デコードと並行して rs と rt を用いて

$$s = \text{GPR}(\text{rs})$$

$$t = \text{GPR}(\text{rt})$$

のように汎用レジスタの値を読み出す。これらの値は分岐判定に用いられるとともに，ALU の入力に用いるため EX 部に引き渡される。

（4） **分岐判定・分岐先計算**　ID 部では，分岐判定と分岐先の計算も行う。分岐命令は beq のみであるため，その成立判定には「分岐命令であり」かつ「s と t が等しい」ことを確認すればよい。分岐成立信号 btaken は以下のように生成され，成立時に 1，不成立時に 0 を取る。

$$\mathrm{btaken} = \mathrm{is\_branch} \cdot (\mathrm{s} == \mathrm{t})$$

ここで s == t は s と t が等しいときに 1，それ以外のときに 0 となる式であり，これと is_branch の論理積を計算している。等値比較器を用いて **図 E.12** 左の回路で計算できる。

次に分岐先アドレスの計算を考える。MIPS 命令セットでは，条件分岐命令の offset フィールドには「分岐がなかった場合に実行すべき命令のアドレス，つまり PC + 4 から数えて何命令先に分岐すればよいか」を入れることと規定されている。よって分岐先アドレス btarget は，少々ややこしいが

$$\mathrm{btarget} = \mathrm{nPC} + 4 \times \mathrm{offset}$$

として生成できる (図 E.12 右)。PC + 4 は nPC という信号名で IF 部から引き渡されていたことを思い出そう。1 命令が 4 バイトなので，offset は 4 倍してから加算する必要がある点にも注意したい。4 倍の計算は 2 ビット左シフトするだけで実現できる。なお，分岐先はプログラムの前方であることも後方であることもあるので offset は負の数も取り得る。

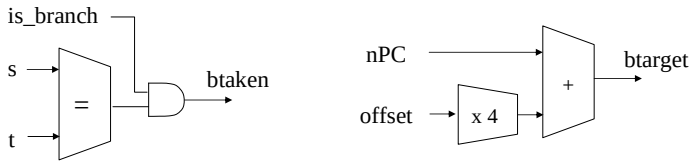



**図 E.12**　分岐判定 (左)，分岐先計算 (右) 回路の構成

これでようやく次の PC を決定するための用意が整った。btarget と btaken および nPC の値を用いて，以下のように PC に書き込む。

$$\mathrm{PC} \leftarrow \begin{cases} \mathrm{btarget} & (\mathrm{btaken} == 1) \\ \mathrm{nPC} & (\mathrm{otherwise}) \end{cases} \quad \textbf{if } \mathrm{PCen}$$

### E.4.3　EX (実 行) 部

EX 部では，ID 部から op，s，t を受け取って ALU による演算を施し，結果 w を rw 番汎用レジスタに書き込む。既に ALU は設計済みのため，単に

$$\mathrm{GPR(rw)} \leftarrow \mathrm{ALU(op, s, t)} \quad \textbf{if } \mathrm{GPRen}$$

とすればよい。

**表 E.2** 制御回路の状態遷移表と出力表

| $Q_1$ | $Q_0$ | is_branch | $Q'_1$ | $Q'_0$ | PCen | GPRen |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | × | × | × | × |
| 1 | 1 | 1 | × | × | × | × |

## E.5 制 御 回 路

IF，ID，EX の各部の設計が完了したので，残るはそれらを順次動作させていくための制御回路の設計である。状態遷移図は図 E.2 に既に示した。IF，ID，EX の各部を 1 状態とする合計 3 状態の有限状態機械なので，$Q_1Q_0$ の 2 ビットで状態を表そう。入力は is_branch の 1 ビットのみである。

出力信号は PCen，GPRen の 2 ビットを設ける。PCen は PC への書き込みイネーブル信号である。GPRen は汎用レジスタファイルへの書き込みイネーブル信号である。

---

**例題 E.2** 状態遷移図 図 E.2 に基づき，制御回路の状態遷移表と出力表を真理値表の形で表せ。また，状態遷移関数，出力関数をカルノー図で表し，それぞれを最も簡単な積和型の論理式で表すことで，この制御回路を設計せよ。

---

**【解答】** 状態遷移表・出力表は**表 E.2** のとおりになる。ただし，次時刻の $Q_1$，$Q_0$ をそれぞれ $Q'_1$，$Q'_0$ と書いた。これらをカルノー図で表すと**図 E.13** のようになり，以下の通り簡単化できる。

$$
\begin{aligned}
Q'_0 &= \overline{Q_1} \cdot \overline{Q_0} \\
Q'_1 &= Q_0 \cdot \overline{\text{is\_branch}} \\
\text{PCen} &= Q_1 + Q_0 \cdot \text{is\_branch} \\
\text{GPRen} &= Q_1
\end{aligned}
$$

これらより，制御回路は**図 E.14** のとおり設計できる。

◇

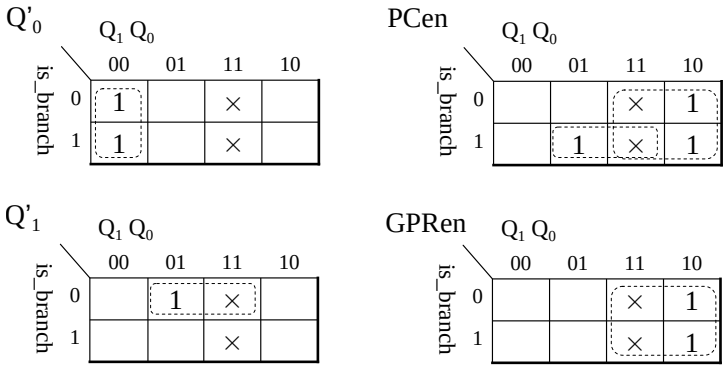



**図 E.13**　制御回路の状態遷移関数と出力関数のカルノー図

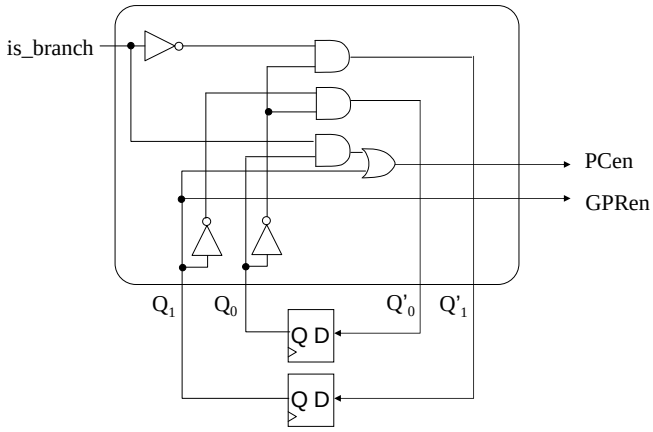



**図 E.14**　制御回路

## E.6　動　作　例

以上で簡易プロセッサの設計は完了である。最後に具体的なプログラムの動作を見てみよう。汎用レジスタ a0, a1 に整数 $m$, $n$ $(m \leqq n)$ を与えて以下のプログラムを実行すると，$m$ から $n$ までの整数の総和 $\sum_{k=m}^{n} k$ が計算され，汎用レジスタ v0 に返される †。

† 極度に省略された命令セットなので，ややトリッキーなプログラミングがされている。1，2 行めでは，値 1 を作るために nor と減算を駆使している。即値演算命令が使えればこのようなことをする必要はない。7 行めの beq は，定数を分岐条件とすることで無条件ジャンプの代わりに使われている。

```
    nor  $t0, $zero, $zero     # t0 ← -1
    subu $t0, $zero, $t0       # t0 ← 0 - (-1) = 1
    or   $v0, $zero, $zero     # v0 ← 0
L1: addu $v0, $v0, $a0         # v0 ← v0 + a0
    beq  $a0, $a1, L2          # a0 = a1 なら L2 へ分岐
    addu $a0, $a0, $t0         # a0 ← a0 + t0 (= a0 + 1)
    beq  $zero, $zero, L1      # L1 へジャンプ
L2: or   $zero, $zero, $zero   # 何もしない（終了）
```

このプログラムをメモリアドレス 0x8000 から始まる領域に配置し，レジスタ a0, a1 にそれぞれ 1 と 2 を与えて実行した場合のタイミングチャートを**図 E.15** に示す。クロックサイクル 22 において，計算結果 $\sum_{k=1}^{2} k = 3$ がレジスタ v0 に書き込まれていることがわかる。

## E.7　プロセッサ構成法の一般論

これまで見てきたように，プロセッサ全体を流れる信号の系統は，メモリ，レジスタファイル，分岐ユニット，ALU といった各構成要素間をデータが受け渡されていく信号の道筋と，それらの各構成要素の動作タイミングを指示する制御信号の道筋に大別できる。前者は**データパス** (data path) と呼ばれ，後者は**制御パス** (control path) と呼ばれる。

プロセッサを含め，あらゆる演算処理システムのデータパスを一般化して考えると，**図 E.16** のように，複数のレジスタと，それらの間を接続する組合せ論理回路からなると考えることができる [†1]。レジスタはクロック信号の立ち上がりのタイミングで組合せ回路の出力を記憶するので，クロックサイクル時間は，レジスタ間の回路遅延 [†2] の最大値よりも長くなくてはならない。すなわち，システム全体の中でもっとも回路遅延の大きな信号伝達経路によって，そのシステムを駆動できるクロック周波数が制限される。この最も遅延の大きな経路のことを**クリティカルパス** (critical path) と呼び，その遅延をクリティカルパス遅延 (critical path delay) と呼ぶ。プロセッサの命令実行の流れを複数のクロックサイクルに分割する際は，クリティカルパス遅延ができるだけ小さくなるようにうまく設計することが重要である。

---

[†1] この図 E.16 を，すべてのレジスタが一箇所にまとまるように描き直すことで，4 章の図 4.2 と同様の形になることに注意する。

[†2] ただし複数クロックサイクルをかけて信号を伝搬させる回路については，回路遅延をサイクル数で割ったものを考える。

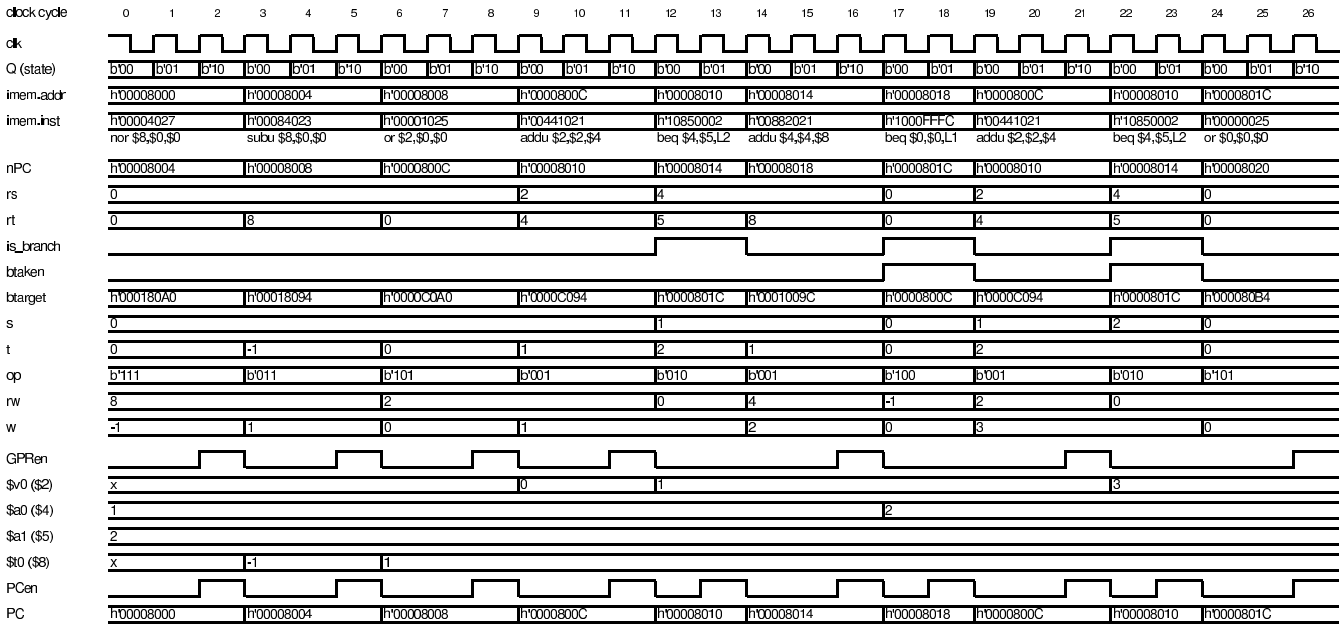



**図 E.15** 設計した簡易プロセッサの動作例。ただし回路遅延は無視できるほど小さいとしている。先頭に h' が付いている値は 16 進数であり，b' がついている値は 2 進数である。値が不定のときは x と表示している。クロックサイクル 0 では，命令メモリアドレス (imem.addr = 0x8000) に対応して命令 (imem.out = 0x00004027) が命令メモリから得られるとともに，アドレスに 4 を加えた信号 nPC が生成されている。命令の各ビットから読み出しレジスタ番号 rs, rt が生成される。btarget も生成されるがここでは使用されないため無意味な値となっている。レジスタから読み出された値 s と t，書き込みレジスタ番号 rw と演算種別 op も生成され，ALU により演算結果 w が直ちに得られる。クロックサイクル 2 で GPRen が 1 であることにより，次のクロックの立ち上がりで演算結果 w はレジスタ番号 rw = 8 (t0) に書き込まれる。同様に PCen が 1 であることにより，nPC の内容が PC に書き込まれ，これが次に実行される命令アドレス (imem.addr) となる。クロックサイクル 12 からは，アドレス 0x8010 の命令 0x10850002 が読み出され同様に実行される。分岐命令であるため is_branch が 1 となるが，分岐条件は不成立のため btaken は 0 であり，btarget は用いられずに nPC の内容がそのまま PC に書き込まれる。一方クロックサイクル 17 から始まる分岐命令では，btaken が 1 になり，nPC ではなく btarget が PC に書き込まれ分岐が生じる。分岐命令の実行中は rw, op, w などには無意味な値が現れているが，使用されない。

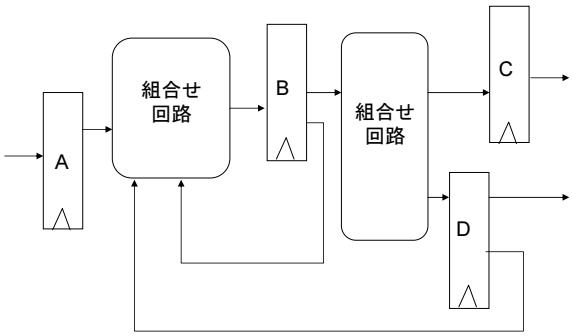



**図 E.16**　データパスの一般構成

本章で設計した簡易プロセッサを，6 〜 7 章で紹介したその他の命令も実行できるように拡張するためにはどのような考え方をすればよいかを簡単に触れておく。今回設計した演算命令と同形式の命令の追加 (例えば xor 命令や slt 命令など) は容易であろう。ジャンプ命令も，分岐命令よりむしろ簡単に追加できる。本章の設計でサポートしなかった即値演算命令の実現のためには，命令の即値フィールドを ID 部から EX 部に引き渡せるような信号経路をデータパスに追加する必要がある。これらの追加には制御タイミングの変更は必要ない。jal 命令はレジスタ ra への書き込みを行うため，IF 状態に戻る際に GPRen を 1 とする必要がある。

ロード・ストア命令の追加を追加するには，IF, ID, EX のほかに，メモリアクセスを行う MEM (MEMory access) を状態として追加する。レジスタファイルが SRAM で実装される場合は，書き込み時間を確保するため WB (Write Back) 状態をさらに追加することが多い。ロード・ストア命令の追加はは章末問題で扱う。

# 章 末 問 題

【1】 本章で述べた簡易プロセッサで，汎用レジスタの内容がすべて 0 の状態から，以下の命令列を実行した。各命令の実行が完了した時点で，以下の各信号が取る値をそれぞれ示せ (ただし beq 命令については (d) を除く)。ただし命令実行が完了する時点とは，次の命令アドレスを確定するために PCen 信号が 1 となった状態でクロックが立ち上がる直前を指す。信号名は図 E.1 を参照せよ。

(a) 信号 is_branch

(b) 信号 rs

(c) 信号 t

(d) ALU の出力信号

```
L1:    nor  $8, $zero, $zero
       subu $9, $zero, $8
       addu $9, $9, $9
       beq  $9, $zero, L1
```

【2】 本章で設計した有限状態機械 (図 10.2) では，状態 EX からは入力信号 is_branch の値に関わらず常に状態 IF に遷移するとしている。実際には，is_branch が 1 のときは状態 EX に遷移することがあり得ないため，is_branch が 1 のときの EX から遷移先およびその際の出力はドントケアと考えて設計することもできる。この場合，例題 10.2 の設計がどのように影響されるか考えよ。

【3】 あるプロセッサに含まれるレジスタ・レジスタ間，およびレジスタ・入出力間の信号伝達経路のうち，遅延が最大のものと最小のものの遅延はそれぞれ 8 ns，2 ns であった。このプロセッサが正常に動作できるクロック周波数の上限を求めよ。

【4】 本章で設計したプロセッサに lw 命令，sw 命令を追加するため，**図 E.17** のようなデータパスを考える。簡単のため，データメモリは命令メモリとは別に用意し，1 入力 1 出力のレジスタファイルと同様にアクセスできるものとする。itype 信号は，命令デコード結果が演算命令，分岐命令，ロード命令，ストア命令であったときにそれぞれ 00，01，10，11 を取る 2 ビットの信号であるとし，is_branch はそれが 01 のとき，is_aluop はそれが 00 のときに限り 1 となる信号である。

(a) **図 E.18** のような状態遷移を実現するための制御回路を設計したい。状態遷移関数，出力関数を表す真理値表を作成せよ。ただし，IF, ID, EX, MEM の各状態をそれぞれ 00, 01, 10, 11 と 2 進符号化するものとし，状態遷移図に記載のない有向エッジはドントケアとする。(余力があればカルノー図を用いて簡単化せよ)

(b) lw，sw の機械語命令は **表 E.3** の通りである。命令デコーダを設計せよ。すなわち，命令信号 inst から itype，rw，op を 生成する組合せ回路を示せ。未定義命令は現れない (ドントケア) としてよい。

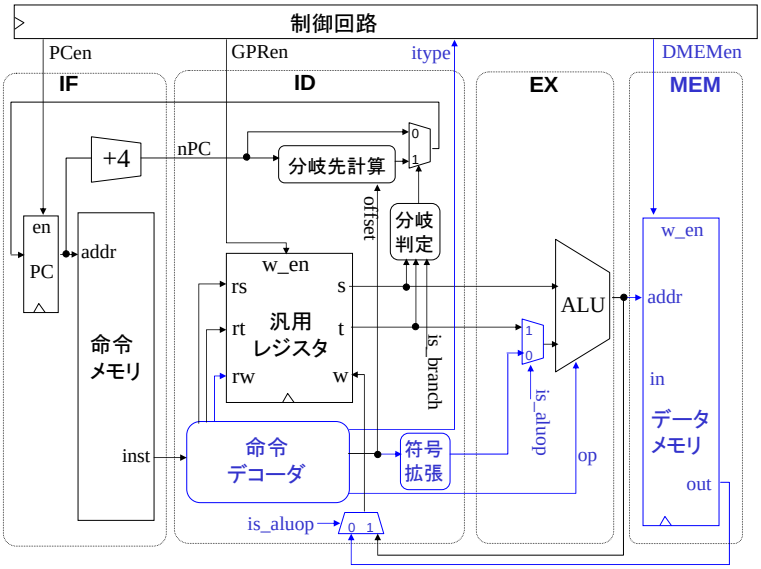



**図 E.17**　データパスへのロード・ストア命令の追加

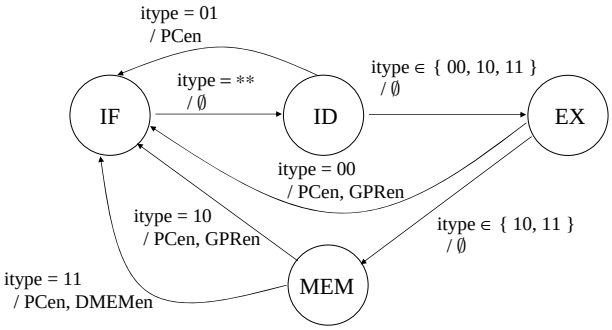



**図 E.18**　ロード・ストア命令を追加した場合の状態遷移図

**表 E.3**　lw と sw の命令エンコーディング

| | 31 … 26 | 25 … 21 | 20 … 16 | 15 … 0 |
|---|---|---|---|---|
| lw | 100011 | rs | rt | offset |
| sw | 101011 | rs | rt | offset |

# F コンピュータの高性能化

6 〜 7 章では，MIPS 命令セットを題材としてコンピュータ内でのプログラム実行の様子について学んだ。さらに E 章では MIPS を思い切って簡略化した命令セットを考えて，それを実現する回路構成，すなわちマイクロアーキテクチャの一例を示した。一方，世の中では MIPS 以外にも多数の命令セットアーキテクチャが用いられている。また，同じ命令セットアーキテクチャに対しても多数のマイクロアーキテクチャが混在している。

本章では，MIPS 以外のさまざまなアーキテクチャに視野を広げて，これまで話し切れなかった各種のプロセッサ構成技術を，コンピュータの高性能化という観点で概観する。

## F.1 コンピュータの性能

コンピュータの高性能化について議論するためには，そもそもコンピュータの性能とは何なのかを明らかにしておく必要がある。最も代表的な指標は，ターゲットして想定している処理の実行時間である。実行時間は以下のような各要因に分解して考えることができる。

$$\text{実行時間} = \text{クロックサイクル時間} \times \text{実行命令数} \times \text{CPI} \tag{F.1}$$

ここで **CPI** (clock cycles per instruction) は，1 命令を実行するために何クロックサイクルを要するかを表す値である。この式から，以下のような基本的な原則を見出せる。

- プログラムの実行時間を短くするためには，クロックサイクル時間を短くし，命令数を少なくし，CPI を小さくすればよい。

E.7 節で見たように，クロックサイクル時間はクリティカルパスの遅延によって定まる。クリティカルパスの遅延を小さくするには，デバイス技術が重要なのはもちろんであるが，回路構成も大きく影響してくる。そして回路構成の変化は一般に CPI の変化や命令数の変化を伴う。例えば，クリティカルパスを短くするために命令の実行を複数クロックサイクルに分割したならば，その分 CPI は増加する。ある命令をより粒度の小さな複数の命令に分割したならば，同じ処理を実現するために必要な命令数は増加する。詳しくは本章で順次議論していくが，これらの各要因を，その時代時代のデバイス技術に合わせてバランスよく考慮することが高性能化の肝となる。

ここまでの議論では，暗黙のうちに「CPI は命令の種類によらず等しい」としてきた。しかし一般には CPI は命令の種類によって異なることに注意が必要である。したがって，想定する処理全体の中で，各命令がどの程度の割合で実行されるかを考える必要がある。ある処理の中で命令 $i$ が実行される回数を 実行命令数$_i$，命令 $i$ の CPI を CPI$_i$ と書くと，式 (F.1) は

$$\text{実行時間} = \text{クロックサイクル時間} \times \sum_i (\text{実行命令数}_i \times \text{CPI}_i) \tag{F.2}$$

のように一般化される。

この式は，ある命令の CPI を改善したときに，それが実行時間全体に及ぼす影響は，その命令が実行される頻度に依存することを示している。ある意味当たり前ともいえるこの関係は，以下の重要な原則につながる。

- 実行時間全体を改善するためには，頻繁に実行される命令を優先して改善すべきである。

他の指標として，単位時間 (秒) 当たりに実行できる命令数を用いることがある。1 **MIPS** (Mega instructions per second) は，1 秒間に 100 万命令を実行できることを意味する。しかし，同じ計算をするために必要な命令数は命令セットによって違うため，異なるアーキテクチャの性能比較に用いることにはほとんど意味が無い。

命令セットによる影響をいくぶん受けにくい指標として，単位時間当たりの浮動小数点数演算の実行回数 **FLOPS** (floating-point operations per second) を用いることもある。1 MFLOPS (Mega FLOPS) は，1 秒間に 100 万回の浮動小数点数演算を実行できることを意味する。同様に，GFLOPS (Giga FLOPS)，TFLOPS (Tera FLOPS)，PFLOPS (Peta FLOPS) などのように任意の接頭辞をつけて用いることができる。

## F.2 コンピュータのコスト

性能に関する議論は，常にそれを実現するコストを考慮した上で行わなければ意味がない。

プロセッサやメモリが半導体チップ上に作られるのが当然である現在，製造コストを決める最大の要因は回路面積である。コストが回路面積に少なくとも比例するのは想像に難くない。さらに，回路面積が大きくなるとその中に製造欠陥が含まれる可能性が高まるため，実際には面積の増大とともにコストは急速に増大する。

半導体集積回路は基本的にはトランジスタの集まりなので，回路面積は，それを構成するトランジスタ数と，1 個のトランジスタの面積をかけ合わせたもので決まる。1 個のトランジスタの面積は半導体技術の進歩とともに急速に小さくなっていくため，同一面積に実装できる回路規模は年々急速に大きくなっていく。この傾向を予測した有名な経験則として「1 個の半導体チップ上に乗るトランジスタの数は 1 〜 2 年で 2 倍になる†」という**ムーアの法則** (Moore's law) が知られている。Intel 社の創業者によるこの予測は，半導体業界の数値目標としての意味を持つようになり，実際にこの傾向での技術発展が続けられている。

さらにうれしいことに，トランジスタが小さくなると，トランジスタ 1 個当たりの信号遅延も小さくなる。結果として，半導体集積回路は年々大規模化し，高速化していくこととなった。1971 年に開発された最初のマイクロプロセッサ 4004 は 2300 個のトランジスタを集積し，クロック周波数 750 kHz で動作した。それから 40 年が経過した現在，1 チップには 10 億個ものトランジスタが集積され，数 GHz で動作する。

製造のためのコストだけではなく，設計・開発に要するコストもコンピュータのアーキテクチャを左右する重要な要因である。設計が複雑化すると，人件費に代表される固定費が増大し，そのコストを回収できなくなってしまう。さらに，1 〜 2 年で回路規模が倍になるような日進月歩の世界では，開発期間が長くかかりすぎると，市場に出る頃には既に競争力を持たない製品になってしまうリスクも大きい。

このような多様で，かつ時間とともに変化する要因が複合的に影響することにより，最適なコスト性能比を実現するアーキテクチャは時代とともに大きく変遷していくこととなる。

## F.3　さまざまな命令形式

### F.3.1　演算構成とオペランド指定方式

MIPS のように数十本ものレジスタをプロセッサに当たり前のように内蔵できるようになったのは，コンピュータの歴史の中では比較的最近のことである。それ以前は，演算に使用できるレジスタはごく少数で，またそれぞれの役割がある程度定まっていることが多かった。演算はレジスタ間だけではなく，メモリを直接指定して行われるのが通常であった。

† 1965 年の時点で，向こう 10 年は 1 年で 2 倍と予測した (Electronics, 38(8), 1965)。その後 1975 年に，2 年で 2 倍と修正している (IEEE International Electron Devices Meeting, 1975)。

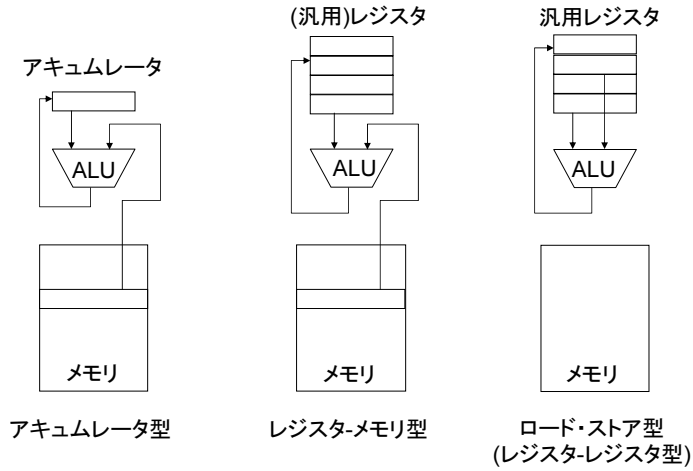



**図 F.1**　演算操作の変遷

最初期の基本的構成としは，**図 F.1** の左に示すような，演算用レジスタが 1 つだけの構成が挙げられる。この場合のレジスタはしばしば**アキュムレータ** (accumulator, 累算器) と呼ばれ，一連の計算がこのレジスタを中心に行われる。複数のレジスタを実装できるようになると，選択されたレジスタ間，または選択されたレジスタ-メモリ間での演算を行うようになった。さらに多くのレジスタが実装できるようになり，レジスタ間のみで演算を行うロード・ストア型アーキテクチャが主流となった。

このような演算まわりの構成により，演算命令のオペランドの指定方式も変わってくる。アキュムレータ方式の場合は，オペランドとしてはメモリアドレスを指定するだけでよい。アキュムレータは，暗黙のうちに入力オペランドの一方になるとともに出力オペランドにもなる。このような命令形式は 1 オペランド型と呼ばれる。複数レジスタが用意されている場合はさらにレジスタ名も指定する必要があり，2 オペランド型と呼ばれる。この場合，オペランドのうち一方は入力と出力を兼ねることになる。これらの命令形式では各レジスタは完全には汎用でないことが多く，例えばメモリアドレスの指定に用いられるインデックスレジスタ，分岐条件を指定するフラグレジスタ等の専用 (または半専用) レジスタがしばしば用意される。

四則演算を始めとして，数学で用いられる基本的演算の多くが 2 項演算であることを考えると，MIPS で採用されているような，入力オペランドを 2 つ，出力オペランドを 1 つ持つ 3 オペランド型の命令形式は自然なものであるといえる。一方，オペランドの指定が冗長になる場合も多く，命令サイズが無駄に大きくなる傾向がある。そのため，ロード・ストア型アーキテクチャでも 2 オペランド型の命令が採用される場合もある。いくつかの命令セットでの代表的な命令の形態を表 **表 F.1** にまとめる。

**表 F.1**　各種の命令セットアーキテクチャでの演算命令の形態。ただしレジスタ名は各アーキテクチャのものではなく統一的な表記でまとめている。オペコード OP も仮想的なもので，対応する 2 項演算子を ⊙ と表記した。任天堂のゲーム機ファミリーコンピュータに採用されたことで有名な 8 ビットプロセッサ 6502 は典型的なアキュムレータ型であり，オペランド A (アキュムレータ) は明示的に書かれない。同時期の Z80 は一部の演算の出力にアキュムレータ以外のレジスタを取れるため，オペランド A を明記している。16〜32 ビットの変遷期の代表的なアーキテクチャである x86 と MC680x0 は，複数のレジスタおよびメモリを組み合わせる 2 オペランド型を採用した。MIPS，PowerPC，ARM などロード・ストア型アーキテクチャの多くは 3 オペランド方式を採用するが，SuperH のようにロード・ストア型でありながら 2 オペランド方式を採用するものもある。

| アーキテクチャ | 命令 | 動作 |
|---|---|---|
| 6502 (MOS Technology) | `OP 10, Ri` | `A ← A ⊙ mem[Ri + 10]` |
| Z80 (ZiLOG) | `OP A, Rs`<br>`OP A, (Ri+10)` | `A ← A ⊙ Rs`<br>`A ← A ⊙ mem[Ri + 10]` |
| x86 | `OP Rd, Rs`<br>`OP Rd, [Ri+10]` | `Rd ← Rd ⊙ Rs`<br>`Rd ← Rd ⊙ mem[Ri + 10]` |
| MC680x0 (Motorola) | `OP Rs, Rd`<br>`OP (10,Ri), Rd` | `Rd ← Rd ⊙ Rs`<br>`Rd ← Rd ⊙ mem[Ri + 10]` |
| MIPS / PowerPC / ARM | `OP Rd, Rs1, Rs2` | `Rd ← Rs1 ⊙ Rs2` |
| SuperH | `OP Rs, Rd` | `Rd ← Rd ⊙ Rs` |

**表 F.2**　MC68020 のアドレッシングモードの一部 (これですべてではない)。演算命令 `OP` に対応する演算子を ⊙ で表している。

| モード名 | 命令表記 | 動作 |
|---|---|---|
| 即値 | `OP #10, Rd` | `Rd ← Rd ⊙ 10` |
| レジスタ直接 | `OP Rs, Rd` | `Rd ← Rd ⊙ Rs` |
| レジスタ間接 | `OP (Ri), Rd` | `Rd ← Rd ⊙ mem[Ri]` |
| 〃 (ポストインクリメント) | `OP (Ri)+, Rd` | `Rd ← Rd ⊙ mem[Ri]; Ri ← Ri + 4` |
| 〃 (プレデクリメント) | `OP -(Ri), Rd` | `Ri ← Ri - 4; Rd ← Rd ⊙ mem[Ri]` |
| 〃 (ディスプレースメント) | `OP (10,Ri), Rd` | `Rd ← Rd ⊙ mem[10 + Ri]` |
| 〃 (インデックス) | `OP (10,Ri,Rj*4), Rd` | `Rd ← Rd ⊙ mem[10 + Ri + Rj*4]` |
| メモリ間接 (プレインデックス) | `OP ([10,Ri,Rj*4], 20), Rd` | `Rd ← Rd ⊙ mem[mem[10 + Ri + Rj*4] + 20]` |
| 〃 (ポストインデックス) | `OP ([10,Ri],Rj*4,20), Rd` | `Rd ← Rd ⊙ mem[mem[10 + Ri] + Rj*4 + 20]` |

### F.3.2　命令の大規模化・複雑化

集積度の向上とともに，レジスタ数を増やすだけでなく，大規模な，あるいは複雑な演算回路を用意することが可能となった。それとともに，単一の命令でより多くの処理をできるように命令を大規模化・複雑化するする傾向が生まれた。

端的な例は取り扱うデータのサイズである。1 命令が扱うデータのサイズは，4 ビットや 8 ビットが主流だった時代から，16 ビット，32 ビットと増えて行き，現在は 64 ビットも用いられている。大きな数の計算に便利なだけではなく，大きなメモリアドレスを扱うことができるようになる点でも重要である。

動作面でも複雑な命令が導入されるようになった。MIPS にはメモリアクセスの際のアドレッシングモードは 1 つしかないことを学んだが，より一般にはさまざまなモードが考えられ，実際に多くのプロセッサが多様なアドレッシングモードを採用した。例えば Motorola 社 MC68020 は多くのパーソナルコンピュータやワークステーション，機器組込み用に用いられ，その命令体系の美しさに定評のあったアーキテクチャであるが，**表 F.2** に示すように多数のアドレッシングモードを備えていた。6 章の例題 6.8 で見たように，MIPS の場合，整数配列の要素にアクセスするために，スタックポインタからの相対位置として配列の先頭アドレスを求め，配列のインデックスに語長である 4 をかけたものをそれに加えたものを計算してから，ロード命令を実行する必要があった。一方，表 F.2 のインデックス付きレジスタ間接モードを使えば，この処理は 1 命令で行える。

もっと極端に複雑な例として，メモリのあるアドレス範囲からの特定の値の探索，あるアドレス範囲のメモリの内容の他の範囲へのコピー，多項式計算などを行う命令を備えるプロセッサも現れた。

これらの取り組みは，式 (F.1) において実行命令数を減らすことを狙ったものと解釈できる。その背景には，プロセッサとメモリの速度差がある。急速に高速化し続けるプロセッサに対して，メモリのアクセス速度は緩やかにしか向上しなかった。その結果，メモリから命令を読み出す部分が性能のボトルネックになるようになった。そのような状況では，個々の命令を複雑化することで，同じ処理を実現するために必要な命令読み出し数を削減しようというのは当然の戦略である。プログラムを格納するメモリ領域の節約にもつながるため，メモリが高価な時代にはその点でも合理的であった。命令フォーマットとしては，複雑な命令は長いビット長で表し，単純な命令は短いビット長で符号化する可変長命令の採用が一般的である。

このように複雑な命令セットを備えるコンピュータは, 後述する **RISC** (reduced instruction set computer, 縮小命令セットコンピュータ) への対義語として **CISC** (complex instruction set computer, 複合命令セットコンピュータ) と呼ばれるようになった。

## F.4　布線論理制御とマイクロプログラム制御

命令の実行制御，すなわち各命令の動作を実現するよう適切な順序でプロセッサ内部の状態を遷移させ，対応した制御信号を生成することは，順序回路によって実現できる。このことは E.5 節で簡単な例を通じて学んだ。このようにプロセッサの制御部を順序回路として実装する方式は**布線論理制御** (wired-logic control) と呼ばれる。

前節で述べたような命令複雑化の傾向により，制御回路の設計も複雑化することとなった。回路設計のほとんどを人手で行っていた時代には，これは開発コストの大幅な増大を招いた。

回路の複雑化が問題なのであれば回路化しなければよい。代わりに，状態遷移表そのものを ROM としてプロセッサ内部に持ち，そこから制御信号を順次読み出していくことでも，プロセッサの制御は実現できる。状態遷移表の 1 行を読み出すと，制御信号と次に遷移すべき状態が書かれている。その制御信号を出力するとともに，次の状態の行を読み出す。これを繰り返すことによる制御方式を**マイクロプログラム制御** (microprogrammed control) と呼ぶ。プロセッサの 1 命令の実行が，状態遷移表の複数の行の間を遷移していくことによって実現されることになり，その 1 行 1 行をマイクロ命令，その全体をマイクロプログラムあるいはマイクロコードと呼ぶ。

マイクロプログラム制御の導入により，開発コストは大幅に削減された。一度制御部を設計してしまえば，新しい命令の導入や既存の命令の動作変更，あるいは開発途中で発覚した設計ミスの修正も，マイクロプログラムの変更のみで対応できる。命令の大規模化・複雑化とともに，制御方式の主流はマイクロプログラム制御となり，その傾向は RISC が台頭するまで続くこととなった。

## F.5 CISC と RISC

### F.5.1 RISC の登場

メモリが遅いことを前提とした命令複雑化の流れは，キャッシュメモリ技術の確立によって大きな転回点を迎えた。命令読み出しにかかる平均時間が大幅に短縮されるようになったため，実行命令数を減らすことの重要性は相対的に低くなり，むしろ命令の実行にかかる時間の方がボトルネックになるようになった。

また，複雑な命令動作を実現するために不可欠なマイクロプログラム制御が速度面で足かせになるようになった。マイクロプログラム制御の大前提は，メモリから命令を読み出すのにかかる時間よりも，マイクロプログラムを書き込んだ ROM の読み出しにかかる時間の方が大幅に短いことである。マイクロプログラム用の小さな ROM は，主記憶と比べると確かに格段に速かった。しかし，キャッシュメモリの登場によりこの前提が崩れてしまった。

キャッシュメモリの導入により，例えば 1 クロックサイクル当たり平均 1 命令が読み出せるようになったとしよう。この命令読み出し速度を無駄なく活用するためには，CPI = 1 を実現しなくてはならない。複雑化した命令セットでこれを実現するのはもちろん困難である。実際，CPI = 1 に近い値を実現するためには後述するパイプライン処理が不可欠であり，パイプライン処理を効率よく行うためには，できるだけ単純で，規則的に動作できる命令セットが要求される。同時に，単純な命令であれば布線論理による制御が可能であり，ボトルネックとなってしまったマイクロプログラム制御を排除できる。

このような思想で設計されたコンピュータは RISC と呼ばれる。命令を単純化するため，式 (F.1) における実行命令数は増大するが，その分 CPI が低減され，また布線論理制御によりクロックサイクル時間も短縮されると期待できる。これらの効果により総合的に実行時間の短縮を狙うのが RISC といえる。6 〜 7 章で学んだ MIPS は，RISC 型の典型というべきアーキテクチャである。RISC に分類されるアーキテクチャにもさまざまなものがあるが，多くに共通するのは，デコードしやすい固定長で規則的な命令フォーマット，比較的多くの汎用レジスタ群，ロード・ストア型アーキテクチャといった特徴である。

このような転回が生じた背景には，キャッシュメモリの他にもいくつかの要因がある。1 つは高水準言語の普及とコンパイラ技術の成熟である。アセンブリ言語によるプログラミングが主流であった頃は，CISC 型のリッチな命令はプログラマにとって便利なものであった。ところが高水準言語によるプログラミングの場合，コンパイラにとっては単純な命令の組み合わせで実行コードを生成するのは造作もないことであり，むしろ複雑な動作をする命令を使いこなすことの方が困難である。結果として，CISC 型の大規模な命令セットのうち，頻繁に使用されるのはごく一部の単純なものばかりであるという状況が生じた。そのような状況のもと，RISC は式 (F.2) から帰結される「頻繁に実行される命令を優先して改善すべき」という原則に適っていた。

別の要因として，コンピュータの設計にコンピュータが利用できるようになった点が挙げられる。マイクロプログラム制御の利点として，その修正容易性・再利用性によって制御回路の設計・開発コストを下げられる点があった。しかし**コンピュータ援用設計** (CAD, computer aided design) の普及により，制御回路の設計は自動化が進み，修正や再利用の必要性は低くなった。

### F.5.2 パイプライン処理

一般に 1 つの命令の実行は複数のステップに分解される。例えば MIPS の命令であれば，6 〜 7 章で見てきたように，メモリからの命令読み出し，命令デコードとレジスタ読み出し，ALU 実行または実効アドレス計算，メモリアクセス，レジスタ書き込みなどの各ステップの組合せで実現された。E 章では，これを大幅に簡略化したモデルを考えて， 1 ステップが 1 クロックサイクルを費やすような制御を実現した。ただし，命令メモリアクセスも 1 クロックサイクルで行える (キャッシュメモリが常にヒットする) と仮定している。このような動作をさせる場合の各命令の CPI は，その動作を構成するステップ数となる。例えば全命令が $n$ 個のステップで動作するならば，そのプロセッサの平均 CPI は $n$ となる。

この場合のプロセッサの各部の動作をよく考えてみると，非常に無駄が多いことがわかる。命令フェッチの間は，命令デコーダも ALU もレジスタファイルもただ遊んでいるだけである。同じく命令デコードの間も，メモリや ALU は使われていない。

このような無駄をなくすには次のように動作させればよい。メモリからフェッチされた命令は，続いて命令デコーダに送られるが，それと同時に次の命令のフェッチも行ってしまう。最初の命令が ALU に送られている間に，2 番目の命令をデコードし，同時に 3 番目の命令をフェッチする。動作の流れを**図 F.2** に示す。このように，複数のステップを時間的にオーバラップさせながら，流れ作業のように動作させていく方式を**パイプライン処理** (pipeline processing) と呼ぶ。オーバラップして実行される各ステップはステージ (stage) と呼ばれる。

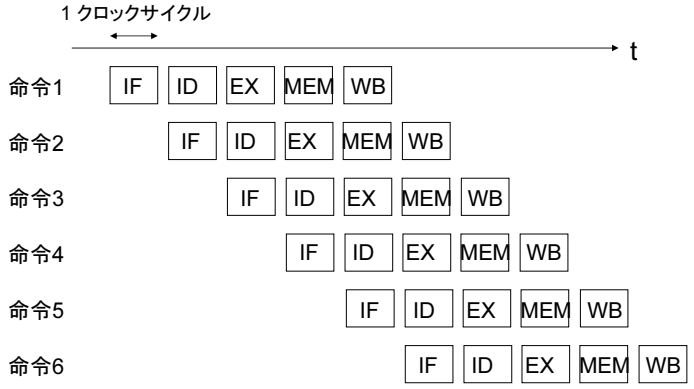



**図 F.2** 5 段の命令パイプライン

このようにプロセッサの動作をパイプライン化することで，式 (F.1) における CPI を低減することができる。理想的には CPI を 1 とすることができる。一般に，命令実行の一連の処理を $n$ 個のステージに分割し，それを理想的にパイプライン動作させることができれば CPI は $1/n$ 倍になり，従って $n$ 倍の高速化が可能である †。$n$ 個のステージが並列動作することによる並列処理の効果であると考えることもできる。

もっとも，理想である CPI = 1 を実際に実現するのはそう簡単ではない。まず，メモリからの命令読み出しが平均 1 クロックサイクルに近い速度で行えなければならない。データ読み書きのためのメモリアクセスについても同様である。よって高速なキャッシュメモリの導入は不可欠である。

CPI = 1 を実現するためには，パイプラインが乱れなく規則的に動作する必要がある。パイプラインの動作が乱されてしまうような状況をハザード (hazard) と呼び，いくつかに分類される。ハザードが生じると，それが解消されるまでパイプラインの動作を一部停止 (**ストール**, stall) しなくてはならない。ストールが発生すると CPI は悪化し，性能を稼げなくなる。

パイプライン処理では複数のステージが時間的にオーバラップして実行されるため，同じハードウェアが複数のステージから同時に利用されないようにしなくてはならない。これに違反する場合を**構造ハザード** (structural hazard) と呼ぶ。例えば，**図 F.3** の例では lw 命令によるメモリからのロード操作と，命令 4 のメモリからのフェッチが同時に発生している。1 次キャッシュが命令用とデータ用に分かれていれば，このハザードは (キャッシュがヒットする限りは) 避けることができる。実際，現代の多くのプロセッサでそのような構成を取っている。

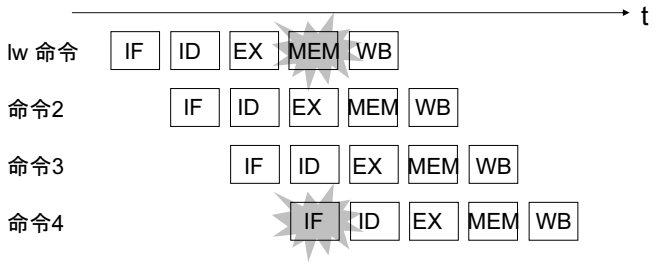



**図 F.3**　構造ハザードの例

† この場合，あくまで複数の命令列からなる処理全体の**スループット** (throughput) が $n$ 倍になっているだけである点に注意する。単一の命令の実行開始から終了までの時間，すなわち**レイテンシ** (latency) は，パイプライン化前と変わらない。

命令間にデータ依存性がある場合も，パイプラインの制御は難しい。例えば**図 F.4** では，or 命令の計算結果はレジスタ t1 に保存され，後続する and 命令が利用している。しかし，and 命令がレジスタから値を読み出す時点では，or 命令はまだレジスタに計算結果を書き込んでいない。このようなハザードを**データハザード** (data hazard) と呼ぶ。データハザードを避けるために，レジスタを介さずに命令間で直接値を受け渡すフォワーディング (バイパス) 経路がしばしば設けられる。

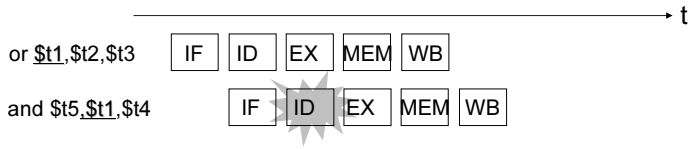



**図 F.4** データハザードの例

ある意味でパイプライン処理と最も相性が悪いのが分岐命令である。分岐をするためには，分岐条件を判定し，分岐先の命令アドレスを計算しなくてはならず，それまでは次の命令のフェッチを開始できない。**図 F.5** の例では，beq 命令の分岐判定が完了する前に and 命令をフェッチしてしまっているが，分岐判定の結果によっては and 命令は実行されないかも知れない。これを**制御ハザード** (control hazard) と呼ぶ。制御ハザードの影響を平均的に低減するため，多くのプロセッサでは，**分岐予測** (branch prediction) を行って後続命令をパイプラインに読み込み続けるようにする。予測が外れた場合は読み込み済みの命令をキャンセルして正しい分岐先に戻らなくてはならない。そのため，予測の精度を高めることと，いざ外れた場合のペナルティを小さくすることが重要である。

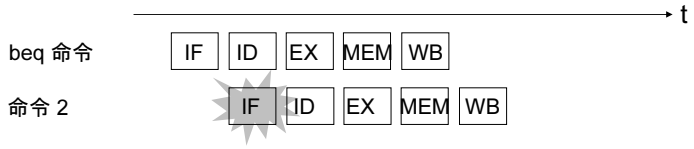



**図 F.5** 制御ハザードの例

### F.5.3 パイプラインの実現例

具体例として，E 章で設計した簡易化 MIPS プロセッサをパイプライン化したものを**図 F.6** に示す。図 E.1 に示した元の構成に対し， 以下のような変更により実現されている。

- IF から ID，ID から EX ステージへ受け渡す信号をいったん記憶しておくための中間レジスタを設ける。これにより，例えば IF ステージが次の命令を読み出し始めても，ID ステージは影響を受けずに前の命令をデコードすることができる。すなわちステージ間の独立動作が可能となる。
- 制御回路は取り払う。PCen により制御されていた PC は毎クロックサイクル更新される。GPRen により制御されていた汎用レジスタは，算術論理演算命令が EX ステージに届いたとき (ex_active が 1 のとき) にのみ書き込まれる。これにより，IF，ID，EX の各ステージがオーバラップしながら並列動作する。ただし分岐が成立する場合は，それが判明するのは ID ステージであるため，分岐動作は 1 命令分遅れて行われる (章末問題 3 の「遅延分岐」を参照)。
- 図 F.6 では省略しているが，汎用レジスタブロック内にデータハザードを解消するためのフォワーディング経路を設ける。具体的には，レジスタから読み出された値 s から

$$\mathrm{s\_fw} = \begin{cases} \mathrm{d} & (\mathrm{ex\_active} \cdot (\mathrm{rs} == \mathrm{rd}) \cdot (\mathrm{rd} \mathrel{!=} 0) == 1) \\ \mathrm{s} & (\mathrm{otherwise}) \end{cases}$$

により s_fw を生成し，s の代わりに用いる。すなわち，ex_active が 1 で，かつ rs と rd が等しく 0 以外の値を取るときに，s の代わりに値 d を用いる。レジスタ値 t についても同様である。これらにより，1 つ前の命令の演算結果が汎用レジスタに書き込まれる前に後続命令が直接利用できる。

E.6 節のプログラム例の動作を **図 F.7** に示す。ただし，遅延分岐に対応するため一部の命令順序を入れ替えてある。図 E.15 と見比べると，全体の実行時間が大幅に短縮されていることがわかる。パイプライン化前は CPI は 2 または 3 だったのに対して，パイプライン化により CPI = 1 となっている。

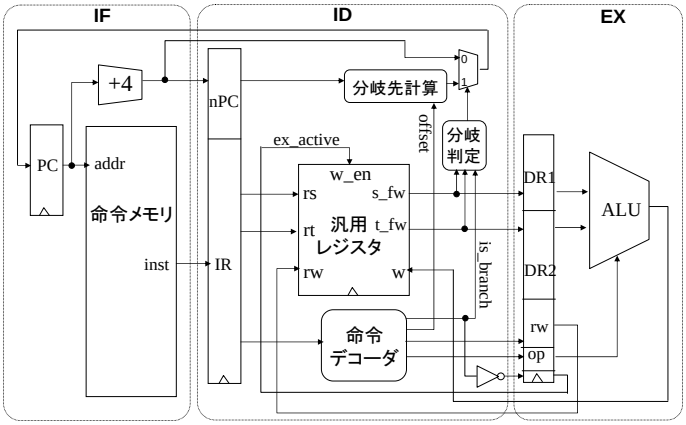



**図 F.6**　E 章の簡易プロセッサのパイプライン化

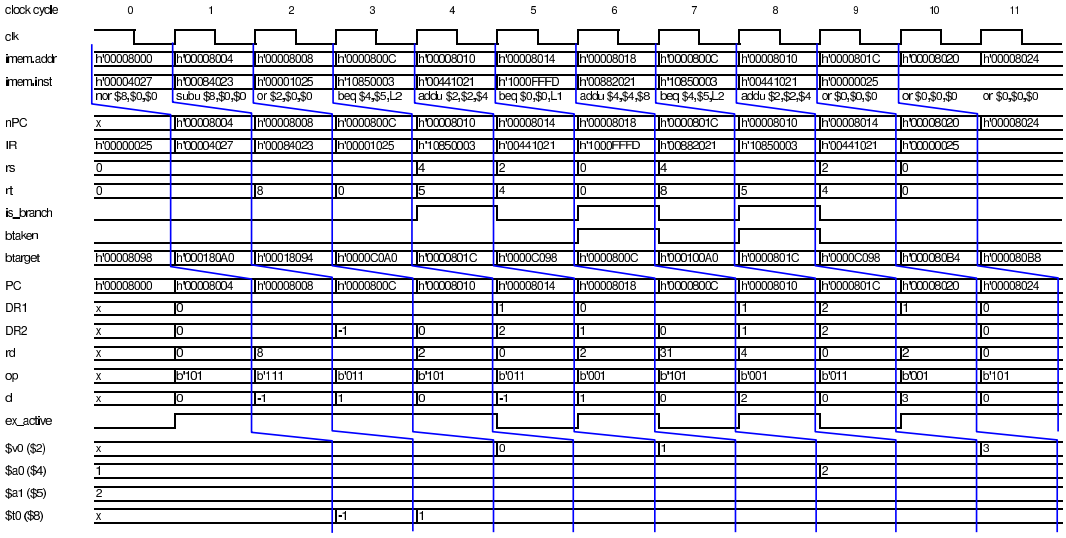



**図 F.7** パイプライン化された簡易プロセッサの動作例。各命令の実行完了には図中の青い線に沿って 3 クロックサイクルを要しているが，各ステージがオーバラップして動作することで，スループットとしては 1 命令あたり 1 クロックサイクルでの実行ができている。

### F.5.4 CISC vs RISC

1980 年代に RISC 型プロセッサは盛んに開発され，高い性能を発揮することが明らかになった。ではそのまま CISC 型プロセッサは駆逐されたのかというと，決してそうではない。当時既にパーソナルコンピュータ分野で大きなシェアを獲得していた Intel 社の x86 は CISC に分類されるアーキテクチャであり，本書執筆時点まで市場をほぼ独占し続けている。

原理的に有利なはずの RISC がこの分野で CISC に勝てなかった最大の理由は，ソフトウェア互換性の壁だと考えられている。命令セットが変わると，それまで用いられていたソフトウェアは (少なくともコンパイルし直さなければ) 動作させることができない。Intel 社が命令セットの互換性を維持し続けたことが，他の命令セットアーキテクチャに対する参入障壁となった。市場を独占することで，x86 プロセッサは量産効果による低コスト化の恩恵も受けることができ，競争力を高める結果となった。

近年の x86 プロセッサは，命令セットこそ CISC のままであるが，マイクロアーキテクチャはむしろ RISC に近い構造となっている。特に Pentium Pro と呼ばれる製品以降は，メモリから読み出した x86 命令をプロセッサ内部で複数の RISC 型命令 ($\mu$OP と呼ばれる) に分解し，RISC 型パイプラインで実行するようになっている。結局，命令セットアーキテクチャが RISC であるか CISC であるかは，あまり本質的ではなかったといってもよい。重要なのは RISC の思想に基づいたマイクロアーキテクチャであった。

一方，ソフトウェアの互換性がそれほど重要でない組込み機器や，ゲーム機，携帯端末などの市場では，MIPS も含めた RISC 型プロセッサが主流となっている。とはいえ RISC 型に分類されるプロセッサも最近は比較的多数の命令を備えるものが増えており，RISC と CISC の境界はやはり曖昧になっている。

## F.6　さらなる高速化

### F.6.1　クロックサイクル時間の短縮

プロセッサのパイプライン化を考える際，パイプラインの段数を増やせば，1 ステージ分の回路遅延は短くなるため，クロックサイクル時間を短縮することができる。例えば Intel 社の Pentium 4 は 20 段以上のパイプラインにより命令を実行し，最高 4 GHz 弱のクロック周波数で動作した。このようにパイプライン段数を増やす技術はしばしば**スーパーパイプライン** (super pipeline) と呼ばれる。

パイプラインが深くなると，いざハザードが生じてストールさせた際のペナルティが大きい。性能の低下を防ぐためには高度な分岐予測が必要となる。別のアプローチとして，分岐命令の使用を減らせるような命令セット上の工夫がなされる場合もある。ARM などの一部のアーキテクチャは，命令フォーマット内に実行条件あるいは**プレディケート** (predicate) と呼ばれるフィールドを用意しており，以下の例のように，指定された条件が成立したときだけその演算を実行させることができる。

```
CMP R1, R2          # R1 と R2 の比較結果を専用レジスタに保存
ADDEQ R3, R4, R5    # 比較結果が「等しい」ならば，R3 ← R4 + R5
```

条件分岐と異なって命令実行の流れは変化しないため，制御ハザードの発生を減らす効果がある。また，命令数の削減にもつながる。

### F.6.2　命令レベル並列性

CPI をさらに削減するための鍵は並列性の利用である。1 クロックサイクルで複数の命令をフェッチし同時に実行できれば，CPI を 1 未満にすることができる。

もちろん任意の命令を常に同時に実行してよいわけではない。多くの命令の間には，命令 A の実行が終わらないと命令 B を開始できないなどといった関係があり，そのような依存関係によって潜在的に並列実行可能な命令数は制限される。このような命令間の並列実行可能性を**命令レベル並列性** (instruction-level parallelism) と呼ぶ。

従来の命令セットとの互換性を保ったまま命令レベル並列性を活用するためには，複数の命令間の依存関係を動的に解析し，並列に実行してよい命令だけを取り出して実行する機構が必要となる。これを実現するプロセッサは**スーパースカラ** (superscalar) と呼ばれる †。この際，実行する命令順を変えずに実行することを**インオーダ実行** (in-order execution) と呼び，命令順も入れ替えながら可能な限り多くの命令を同時実行することを**アウトオブオーダ実行** (out-of-order execution) と呼ぶ。アウトオブオーダ実行を行う方が高い並列性を抽出できるが，その分複雑な機構が必要となる。

命令セットの互換性を保つ必要がない場合は，並列実行できる命令の抽出をコンパイラに任せるという戦略も取り得る。この場合，複数の「命令」を単純に並べたものが新しい命令となり，それをそのまま複数の演算ユニットで並列に実行する。非常に長い命令フォーマットを持つことになるため，この方式は **VLIW** (very long instruction word) と呼ばれる。スーパースカラと比べてハードウェアが簡素化されるのがメリットであるが，並列性が十分に抽出できない場合はプログラム用のメモリが無駄遣いされてしまうという問題もある。

### F.6.3 データ並列性

別の並列性として，**データ並列性** (data parallelism) が挙げられる。例えば画像処理や音声処理，あるいはある種の科学技術計算などでは，同一の演算を多数のデータに適用することが頻繁に行われる。単一の命令 (あるいは命令列) で複数のデータを処理する機構を導入することで，このような処理を高速化できる。このような並列処理様式を一般に **SIMD** (single instruction stream, multiple data stream) と呼ぶ。これに対して，より一般の並列処理様式，すなわち複数のデータを独立した命令 (あるいは命令列) により処理する様式を **MIMD** (multiple instruction stream, multiple data stream) と呼ぶ。

SIMD 型の並列処理を実現する方法は，大別して空間並列型と時間並列型に分類できる。空間並列型では，複数の同一構造の演算器を並べ，それらのすべてを同一の命令で制御するとともに，それぞれに並列にデータを供給することで並列処理を行う。浮動小数点数ベクトル $[a_1, a_2, \cdots]$ と $[b_1, b_2, \cdots]$ の加算を行う例を**図 F.8** に示す。スーパーコンピュータや，特定アプリケーション専用プロセッサの分野では古くから用いられて来たが，近年はパーソナルコンピュータ用のプロセッサに付加命令として導入されるようになった。このような付加命令は SIMD 命令と呼ばれることが多い。

† 後述するベクトルプロセッサと対比させた命名と考えられる。スーパースケーラと発音される場合も多い。

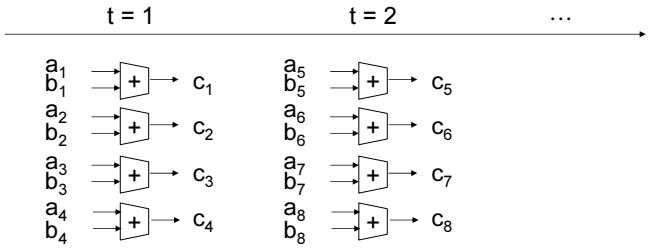



**図 F.8**　空間並列型の SIMD 処理。4 並列の演算器を備える例を示している。

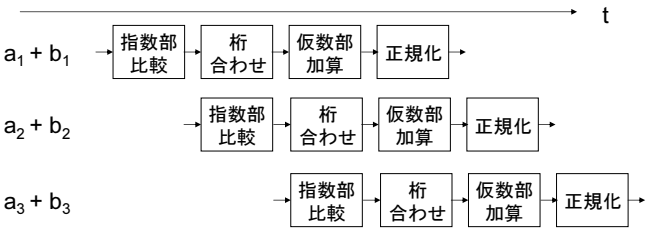



**図 F.9**　時間並列型の SIMD 処理

例えば x86 への付加命令セットである MMX は，64 ビットの浮動小数点レジスタに格納されたビット列を，8 個の 8 ビット整数，あるいは 4 個の 16 ビット整数とみなして並列にデータ処理を行う命令群である。同じく SSE 命令セットは，専用レジスタに格納されたデータに対して浮動小数点演算を並列実行することができる。

一方の時間並列型では，並列化すべき演算を複数のステージに分割し，そこに処理対象のデータを順に送り込んでいくことでパイプライン的に実行する。先ほどの浮動小数点配列ベクトルの加算の例で考えよう。各要素の浮動小数点加算は，指数の比較，桁合わせ，仮数部の加算，正規化・丸めといったステップに分解できるが，これらを**図 F.9** のようにパイプラインステージとして実装し，そこにベクトルの要素を順に流し込んでいくことで演算を行う。このような方式を実現したプロセッサまたはコンピュータは，今の例のようにベクトル演算に高い効果を発揮するため**ベクトルプロセッサ** (vector processor) またはベクトルコンピュータ (vector computer) などと呼ばれる †。比較的少ない演算器を休みなく回すことで高速処理を実現できる優れた方式であり，スーパーコンピュータといえばベクトル型という時代が長く続いた。半導体集積化技術が進展して演算器を多数用意することが容易になった近年では，必ずしも唯一解とはいえなくなっており，むしろ MIMD 型のスーパーコンピュータが多く開発されるようになった。

† 一般に「SIMD」と「ベクトル」という単語自体は，空間並列型，時間並列型のいずれの場合にも使用される。よって単に SIMD 処理とかベクトル処理などと呼ばれた場合に，どちらの実現方式が用いられているかは注意して判断する必要がある。ただし大まかな傾向としては，「SIMD 命令」という用語は空間並列型の付加命令に用いるのがほとんどであり，「ベクトルコンピュータ」「ベクトルプロセッサ」という用語は時間並列型の処理機構を採用したものを指すことがほとんどである。

グラフィックス描画もデータ並列性の高い処理である。ディスプレイへ画像を表示する入出力装置であるグラフィックスボードには大量のデータ処理を行う演算ハードウェアが搭載され，その能力が劇的に進歩したことにより **GPU** (graphics processing unit) と呼ばれるようになった。GPU では，表示する要素ごとの座標変換や色計算といった一連の処理がパイプライン化されて時間並列型で行われるとともに，各ステージは空間的にも並列化されており，極めて高度なデータ並列処理プロセッサとなっている。また量産効果により，コスト性能比の高い並列処理が可能となっている。グラフィックス以外の分野へ応用しようとする動きも盛んであり，**GPGPU** (general-purpose GPU; 汎用 GPU) などと呼ばれる。

### F.6.4 スレッドレベル並列性

プロセッサの高速化は，以上で述べたようなマイクロアーキテクチャの進化と回路の微細化によるクロックサイクル時間の短縮および並列性の利用によって進められてきた。しかし，その路線にも限界が見えてきた。

まずクロックサイクル時間の短縮が物理的限界に近いところまで来ている。仮にクロック周波数 10 GHz で動作させたとすると，1 クロックサイクルでは信号は真空中の光速度でも 3 cm しか伝搬できない。

加えて，消費電力の問題が顕在化してきた。トランジスタが正常動作できる温度には上限があるため，消費電力が大きすぎるとプロセッサの冷却が問題となる。また，携帯機器などに用いる場合はバッテリの持続時間に直接影響する。理論的には，トランジスタのサイズと電圧を同じ割合で減少させ不純物濃度を増加させることで，単位面積あたりの消費電力を変えずに同じ回路を高集積化・高速化できるという**スケーリング則** (scaling law) が知られている。しかし実際には電源電圧の低下には物理的な限界があり，サイズと同じ割合では減少できていない。また，パイプライン化のような回路変更による高速化はスケーリング則には従わない。結果として単位面積あたりの消費電力は増大を続けてきた。さらに，微細化・低電圧化にともなって，回路図上では本来流れないはずのところを流れる**リーク電流** (leakage current) による消費電力が大きな割合を占めるようになってきた。これらを考慮して全体の消費電力を抑える設計を行わなくてはならない。

並列性の利用についても，通常のアプリケーションでは抽出できる命令レベル並列性には限界があり，平均すると 3 程度が上限とされる。データ並列性はアプリケーションに大きく左右され，その利用が常に有効とは限らない。

そのため近年では，**スレッドレベル並列性** (thread-level parallelism) の活用に関心が移っている。近年のコンピュータの利用形態を考えると，1 台のコンピュータで単一のプログラムを走らせるだけという状況は少なく，常に複数のプログラムが並行 † して動作しているのが通常である。また単一のプログラム内でも，処理の流れ，すなわちスレッドを複数に分けて並行して走らせる**マルチスレッド処理**が普及してきた。異なるスレッドからであれば，互いに依存しない命令を容易に取り出すことができる。

スーパースカラ型プロセッサにおいて，複数のスレッドから命令を取り出して演算器を余すことなく動作させる方式を**同時マルチスレッディング** (simultaneous multithreading, SMT) と呼ぶ。Intel 社のプロセッサにおけるハイパースレッディングと呼ばれる実装がよく知られている。

複数のスレッドを並列処理する方法として，より直接的なのはプロセッサを複数用いる**マルチプロセッサ** (multi processor) 方式である。スーパーコンピュータや，あるいはサーバ用途のコンピュータでは古くから用いられてきたが，近年は膨大な数のトランジスタが 1 チップ内で利用可能になったことから，1 チップ内に複数のプロセッサを実装する**マルチコア** (multi core) 方式が広く採用されるようになり，パーソナルコンピュータ分野でも一般化した。

この方向の研究開発は現在精力的に進められており，性質の異なる複数のプロセッサを集積する**ヘテロジニアスマルチコア** (heterogeneous multi core)，数十個を超えるプロセッサを集積する**メニーコア** (many core) などと呼ばれる技術も盛んに研究されている。これらの技術では，単純なマルチスレッド処理だけではなく，時間並列的な動作により大量のデータ並列処理を行うなど先進的な並列化が試みられており，今後の動向が注目されている。

## 章 末 問 題

**【1】** あるプログラムの高速化を目的として，プロセッサを改良することにした。そのプログラムでは全実行時間の 60 % をある特定の演算処理が占めていたので，この演算処理を高速化することとした。プログラム全体を 2 倍速く実行するためには，その演算処理を何倍に高速化する必要があるか。2.5 倍，3 倍ならどうか。

**【2】** あるプロセッサの命令群 A, B, C, D の CPI がそれぞれ 2, 4, 4, 3 であり，あるプログラムにおける各命令群の実行頻度が 40 %，30 %，20 %，10% であったとする。このプロセッサを改良して各命令群の CPI を 2, 3, 3, 5 になるようにした。平均 CPI はどのように変わるか。

† 並列 (parallel) と並行 (concurrent) を使い分けていることに注意したい。複数の処理が，複数のハードウェアで実際に同時に実行されている場合に「並列」という用語を用いる。「並行」という用語は複数の処理の流れが共存している場合に用いられ，概念として「並列」を含むが，常に並列であるとは限らない。

【3】 分岐予測技術が確立していなかった頃に設計された MIPS 命令セットは，パイプラインの制御ハザードを解消するため，**遅延分岐** (delayed branch) と呼ばれる方式を採用している。すなわち，分岐命令やジャンプ命令直後の 1 命令は常に実行され，実際の分岐やジャンプは 1 命令遅れて行われる。分岐・ジャンプ命令の直後 (遅延スロットと呼ばれる) に適切な命令を配置するのはコンパイラあるいはアセンブラの役目である。同様に，初期の MIPS 命令セットでは，メモリステージからのデータハザードを解消するために 1 スロットの遅延ロードも採用されていた。すなわち，あるレジスタへのロード命令実行後，実際にそのレジスタの値を使用できるのは翌々命令以降であった。

第 9 章 章末問題 2 のプログラムを，遅延分岐，遅延ロードが行われる場合に正しく動作するように書き換えよ。

# G コンピュータネットワーク: 発展編

本章では，10 章で述べ切れなかったコンピュータネットワークの動作の詳細について説明する。

## G.1 Ethernet による通信

10 章にて述べた通り，Ethernet ではデータをフレームとして送信する。**図 G.1** の示すように，フレームはヘッダ，送信データ，トレイラにより構成される。ヘッダの先頭 12 バイトは宛先および送信元の **MAC アドレス**を格納するフィールドである。

続く 2 バイトは，データ長またはデータの上位層プロトコルの種類 (タイプ) を表すフィールドである。データ部分は最大で 1500 バイトであるため，1500 以下の値の場合にはデータ長を，さもなくばタイプを表すことになる。データフィールドの後のフレームの最後のトレイラには，4 バイトの **FCS** (frame check sequence) フィールドが設けられている。FCS はフレームの誤りを検出するためのもので，宛先アドレス，送信元アドレス，長さ・タイプ，データから計算した **CRC**（cyclic redundancy check）値を格納する。受信ノードでは同様に CRC を計算し，FCS フィールドの値と一致しない場合は誤りが発生したとしてそのフレームを破棄することとなる。

Ethernet におけるフレームによるデータの送受信をまとめると，**図 G.2** のようになる。この例では，ノード A がノード C にフレームを送信する。ノード A では，上層のインターネット層より送られてきた分割済みのデータ（1500 バイト以下）に対し，ヘッダとトレイラを付加してフレームを生成する。次に，他のノードがフレーム送出をしていないことを確認のうえ，ノード A はフレームをケーブルに送出する。他のノード B,C は，ケーブルに流れるデータを監視している。ノード B では，フレームのヘッダを読み，送信先 MAC アドレスが自分アドレスと一致しないことを検出する。このような自分宛でないフレームは取り込まれない。一方，送信先 MAC アドレスが自分のアドレスと一致するノード C では，自分宛のフレームとしてデータとトレイラ部分を取り込む。最後に FCS によりエラーがないことを確認し，読み込んだデータ部分を上層のインターネット層へ送る。

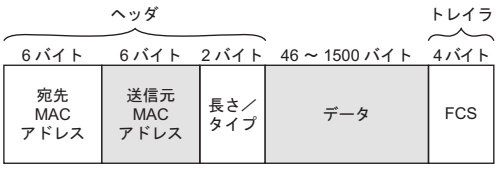



**図 G.1** Ethernet フレームのフォーマット

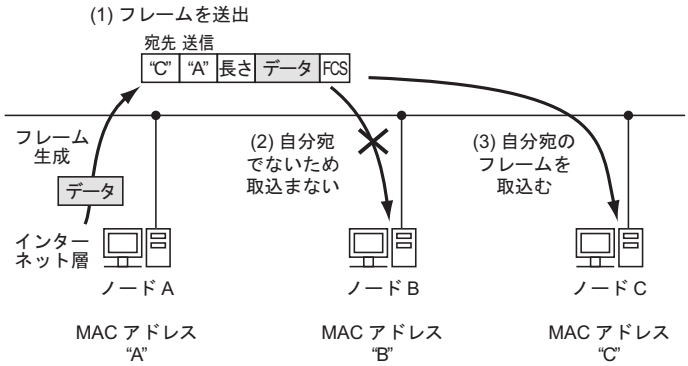



**図 G.2** Ethernet によるフレームの送受信

## G.2 TCP の通信手順

TCP コネクションの確立は，**図 G.3** に示すように以下の手順により行われる。

1) まず，送信ノード A が受信ノード B に対する A → B の接続要求パケット (Syn=1) を送信する。
2) ノード B は，これに対し A → B の接続を承諾する (Ack=1) とともに，B → A の接続要求 (Syn=1) を行うパケットを送り返す。
3) ノード A は B → A の接続を承諾するパケット (Ack=1) を返し，A・B 間のコネクションが確立される。
4) データ送信の終了後，コネクションを切断するために，ノード A はノード B に A → B の切断要求 (Fin=1) を送信する。
5) ノード B は，これに対し，A → B の切断要求を承諾する応答 (Ack=1) を返す。
6) B → A の切断要求 (Fin=1) をノード A に送信する。
7) 最後に，ノード A は B → A の切断要求を承諾する応答 (Ack=1) を返す。

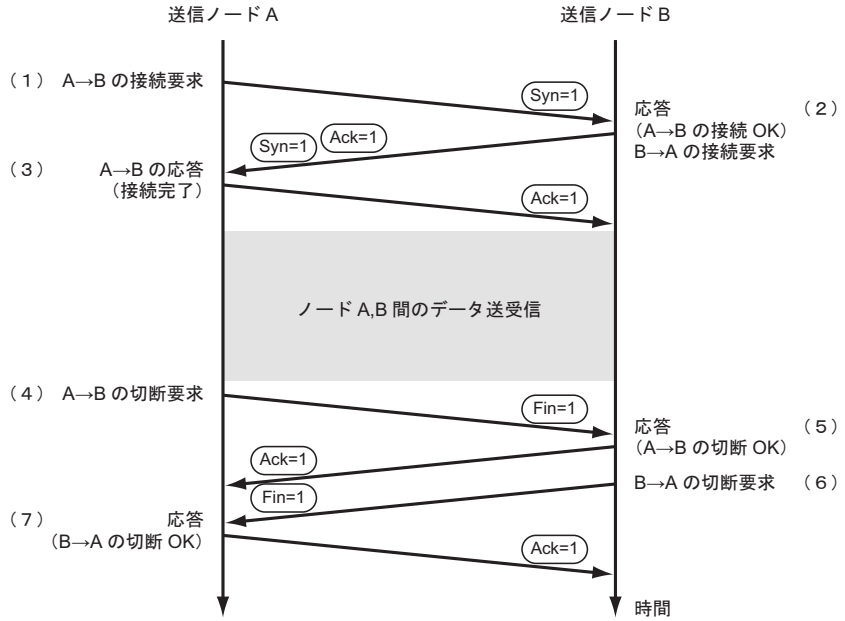



**図 G.3**　コネクション（接続）の確立と切断

データ送信でも，パケットが届いたことを確かめる確認応答の仕組みが用いられている。**図 G.4**(1) は 3 つのパケット送信とその確認応答の例である。送信ノード A は，受信ノード B にパケット 1 を送信する（①）。ノード B はパケットを受信後，その確認応答パケットをノード A に返信する（②）。ノード A はパケット 1 が無事届いたことを知り，次のパケットの送信を行う。確認応答がある一定時間内に帰ってこないタイムアウトの場合には，送出パケットが失われた可能性が高いため，ノード A は同じパケットの再送信を行う。このようにして，パケットが相手に届いたことを確認しながらデータを送信することにより，信頼性の高いデータ送信が実現されている。

しかしながら，パケット毎に確認応答を待ってから次のパケットを送信していたのでは，時間がかかりすぎてしまう。そこで，実際には，ウィンドウ制御を用いて，確認応答を待たずに複数のパケットを送信し通信性能を向上させている。図 G.4(2) は，ウィンドウサイズが 4 の例である。この例では，送信ノードは，4 つのパケットまでを確認応答を待たずに送信する。その途中に確認応答が返ってきており，次のパケットの送信が可能となる。このようにして，確認応答を待つ間にも効率良く別のパケット送信を行うことができるが，再送の可能性があるためウィンドウサイズの分だけパケットをバッファメモリに保持したり，それぞれのパケット毎にタイムアウトを判定するためのタイマーを実装する必要がある。この他，受信側のバッファがなくなった場合等，送信側に一時的に送信停止を要求するフロー制御なども行われる。

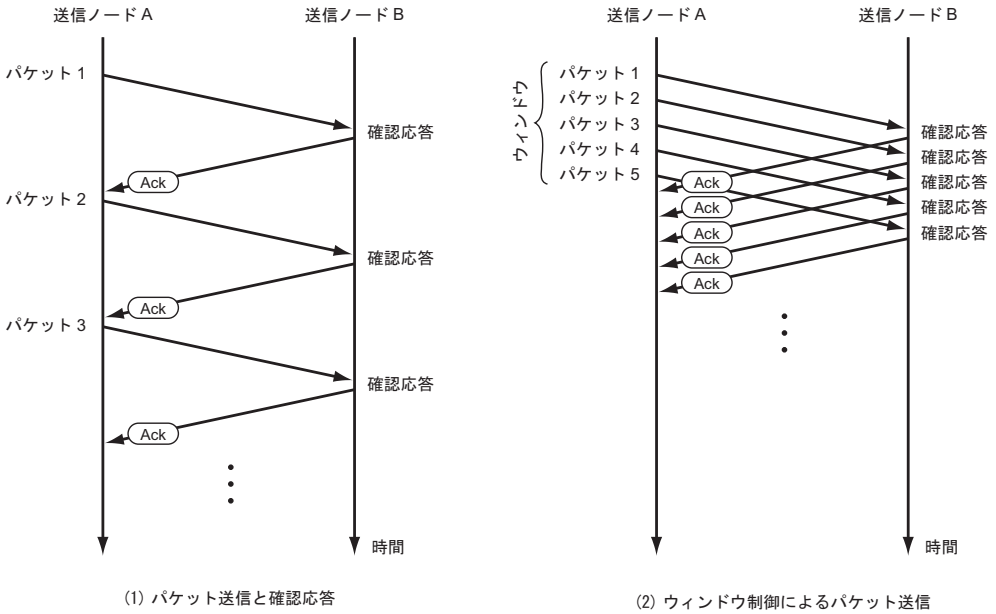



**図 G.4**　確認応答を伴った TCP のパケット送信

## G.3　アプリケーション層プロトコルの例: POP3

TCP/IP のアプリケーション層プロトコルの例として，POP3 によるメール受信の手順を説明する。**図 G.5** に示すように，POP3 によるメールサーバとの一連のやり取り（セッション）は以下のように行われる。

1) まず，クライアントのメールソフトとサーバの POP3 プログラム間で，TCP のコネクションを確立する。
2) 次に，POP3 サーバが応答メッセージを送信し，セッションが開始される。
3) クライアントは，受信するメールのユーザ名を USER コマンドによりサーバへ送信する。
4) コマンドが正しく受理されると，サーバは応答メッセージ”+OK”を返す。
5) 続いてクライアントは，パスワードを PASS コマンドにより送信する。
6) パスワードが正しければサーバが応答する。

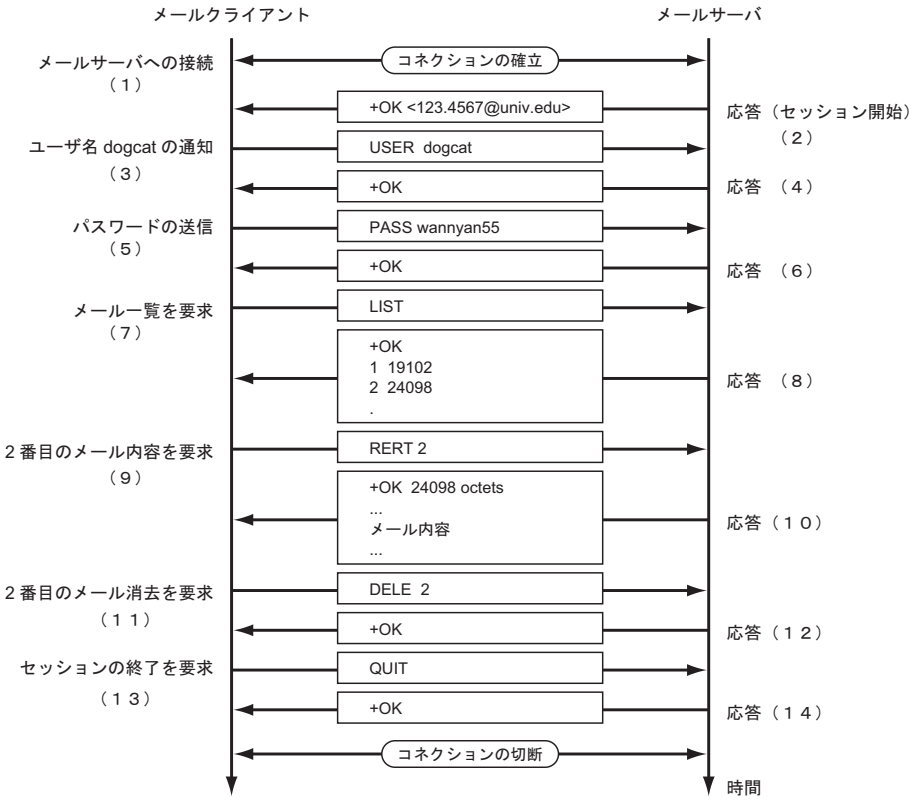



**図 G.5**　POP3 の例

パスワード認証の後では，クライアントはそのユーザの受信メールボックスの内容を様々なコマンドにより問い合わせることができるようになる。例では，まず，LIST コマンドにより受信ボックス内のメール一覧を要求している (⑦)。これに対し，サーバは 2 通のメールがあることを応答している (⑧)。次に，RETR(retrieve) コマンドにより，2 番目のメールの内容送信を要求している (⑨)。これに対し，サーバはそのメール内容をクライアントに送信し，最後に”.”を送る (⑩)。続いてクライアントは 2 番目のメールを DELE コマンドにより削除する (⑪,⑫)。最後に，QUIT コマンドによりセッションを終了する (⑬,⑭)。セッション終了後，TCP コネクションが切断される。メールソフトウェアというのは，サーバよりメールを受信する際に以上のやり取りを適切に行うソフトウェアなのである。

# H 計算機の歴史: 資料

本章には，紙面の都合から 11 章に掲載できなかった写真・図版等を列挙する。これらのうち一部は，以下の各ライセンスのもとで公開された作品およびパブリックドメインの作品の二次利用である。

| | |
|---|---|
| CC-BY 2.0 | http://creativecommons.org/licenses/by/2.0/ |
| CC-BY 2.5 | http://creativecommons.org/licenses/by/2.5/ |
| CC-BY-SA 2.5 | http://creativecommons.org/licenses/by-sa/2.5/ |
| CC-BY-SA 3.0 | http://creativecommons.org/licenses/by-sa/3.0/ |

**図 H.1** ジョン・ネイピア [by Samuel Freeman，public domain]。http://commons.wikimedia.org/wiki/File:John_Napier.JPG

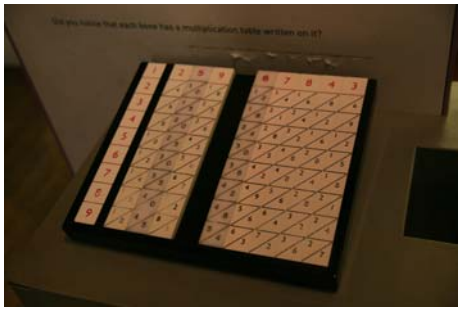


**図 H.2** ネイピアの骨 [ロンドン科学博物館，佐野健太郎撮影]。

**図 H.3** ヴィルヘルム・シッカート [著作者不明，public domain]。http://commons.wikimedia.org/wiki/File:Wilhelm_Schickard.jpg

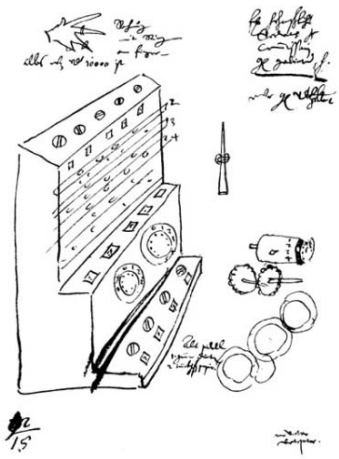


**図 H.4** シッカートの計算機のスケッチ [by Wilhelm Schickard, public domain]。http://commons.wikimedia.org/wiki/File:Rechenmaschine_wilhelm_schickard.png

**図 H.5** シッカートの計算機の複製 [ドイツ博物館，佐野健太郎撮影]。

**図 H.6** ブレーズ・パスカル [著作者不明, public domain]。http://commons.wikimedia.org/wiki/File:Blaise_pascal.jpg

**図 H.7** パスカリーヌ [by David Monniaux, CC-BY-SA 3.0]。http://commons.wikimedia.org/wiki/File:Arts_et_Metiers_Pascaline_dsc03869.jpg

**図 H.8** ゴットフリート・ライプニッツ [by Christoph Bernhard Francke, public domain]。http://commons.wikimedia.org/wiki/File:Gottfried_Wilhelm_von_Leibniz.jpg

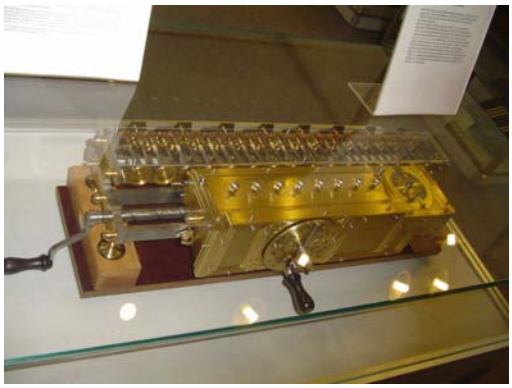


**図 H.9** ライプニッツの計算機 (Stepped Reckoner) の複製 [by Kolossos (@de.wikipedia.org), CC-BY-SA 3.0]。`http://commons.wikimedia.org/wiki/File:Leibnitzrechenmaschine.jpg`

**図 H.10** タイガー手回し計算器 [東北大学 山本悟教授所蔵，佐野健太郎撮影]。

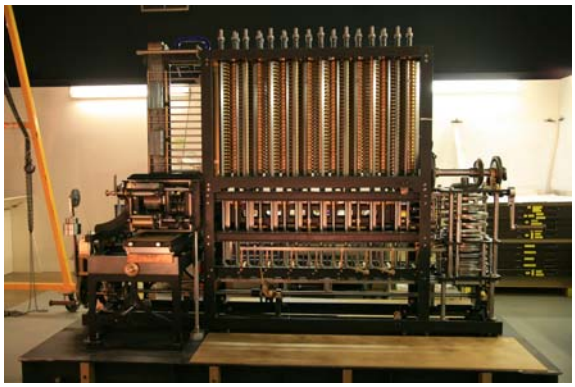


**図 H.11** バベッジの第 2 階差機関の複製 [ロンドン科学博物館，佐野健太郎撮影]。

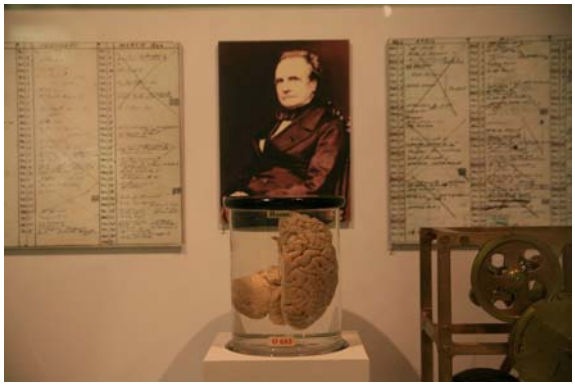


**図 H.12** チャールズ・バベッジの右脳 [ロンドン科学博物館，佐野健太郎撮影]。

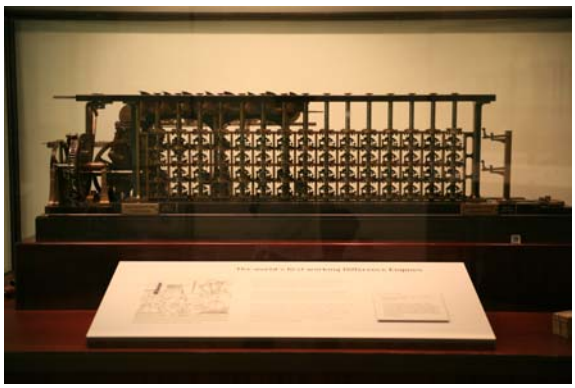


**図 H.13** シュウツ親子の階差機関 3 号機 [ロンドン科学博物館，佐野健太郎撮影]。

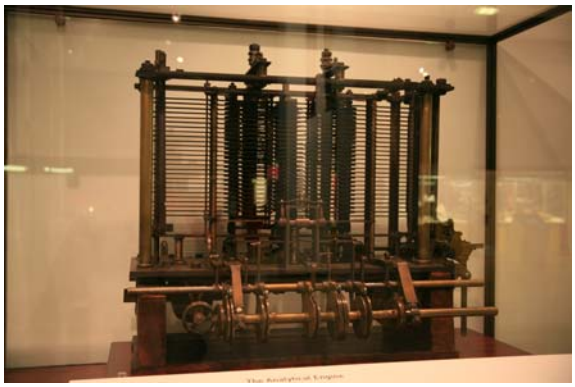


**図 H.14** バベッジの解析機関の一部 [ロンドン科学博物館，佐野健太郎撮影]。

**図 H.15** 解析機関の演算部である「ミル」[ロンドン科学博物館，佐野健太郎撮影]。

**図 H.16** エイダ・ラブレス [by Margaret Sarah Carpenter, public domain]。http://commons.wikimedia.org/wiki/File:Ada_Lovelace.jpg?uselang=ja

**図 H.17** コンラート・ツーゼ [by Wolfgang Hunscher, Dortmund, CC-BY-SA 3.0]。http://commons.wikimedia.org/wiki/File:Konrad_Zuse_(1992).jpg?uselang=ja

**図 H.18** Z1 の複製 [by Stahlkocher (@de.wikipedia.org), CC-BY-SA 3.0]。`http://commons.wikimedia.org/wiki/File:Zuse_Z1.jpg?uselang=ja`

**図 H.19** ABC マシン [by Manop (@th.wikipedia.org), CC-BY-SA 2.5]。`http://commons.wikimedia.org/wiki/File:Atanasoff-Berry_Computer.jpg`

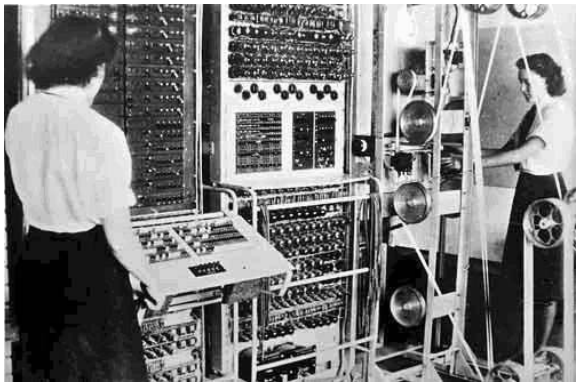


**図 H.20** コロッサス [The National Archives, United Kingdom, public domain]。`http://commons.wikimedia.org/wiki/File:Colossus.jpg`

**図 H.21** コロッサスに用いられた真空管 [ロンドン科学博物館，佐野健太郎撮影]。

**図 H.22** ホワード・エイケン [撮影者不明，public domain]。`http://commons.wikimedia.org/wiki/File:Aiken.jpeg`

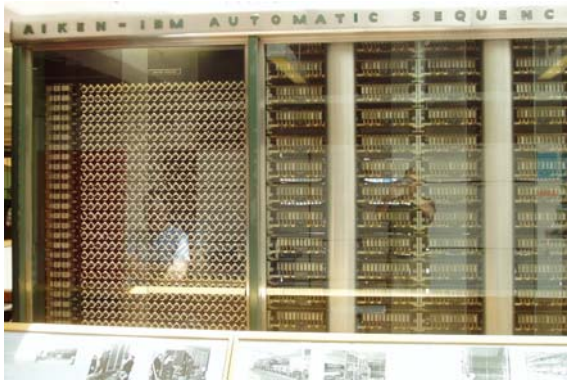



**図 H.23** ハーバード・マーク I [by Daderot (@en.wikipedia.org)，CC-BY-SA 3.0]。`http://commons.wikimedia.org/wiki/File:Harvard_Mark_I_Computer_-_Left_Segment.jpg`

**図 H.24** グレース・ホッパー [by James S. Davis, public domain]。http://commons.wikimedia.org/wiki/File:Grace_Hopper.jpg

**図 H.25** ENIAC [U. S. Army Photo, public domain]。http://ftp.arl.mil/ftp/historic-computers/

**図 H.26** EDVAC [US Army Photo, public domain]。http://ftp.arl.mil/ftp/historic-computers/

**図 H.27** ジョン・フォン・ノイマン [Los Alamos National Laboratory, Unless otherwise indicated, this information has been authored by an employee or employees of the Los Alamos National Security, LLC (LANS), operator of the Los Alamos National Laboratory under Contract No. DE-AC52-06NA25396 with the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this information. The public may copy and use this information without charge, provided that this Notice and any statement of authorship are reproduced on all copies. Neither the Government nor LANS makes any warranty, express or implied, or assumes any liability or responsibility for the use of this information]。`http://commons.wikimedia.org/wiki/File:JohnvonNeumann-LosAlamos.gif`

**図 H.28** UNIVAC I (中央の人物はエッカート) [U.S. Census Bureau, public domain]。`http://commons.wikimedia.org/wiki/File:UNIVAC_1_demo.jpg`

**図 H.29** ウィリアムズ・キルバーン管 [by ArnoldReinhold (@en.wikipedia.org), CC-BY-SA 3.0]。`http://commons.wikimedia.org/wiki/File:Williams_tube.agr.jpg`

**図 H.30** モーリス・ウィルクスと製作中の EDSAC I [Computer Laboratory, University of Cambridge, CC-BY 2.0]。`http://www.cl.cam.ac.uk/relics/archive_photos.html`

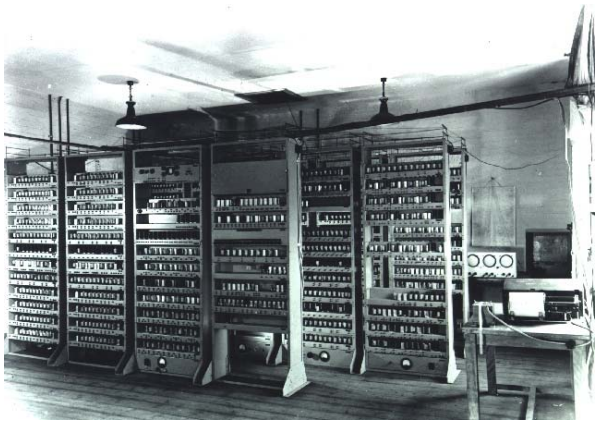


**図 H.31** EDSAC I [Computer Laboratory, University of Cambridge, CC-BY 2.0]。`http://www.cl.cam.ac.uk/relics/archive_photos.html`

**図 H.32** 水銀遅延線 [ロンドン科学博物館，佐野健太郎撮影]。

**図 H.33** EDSAC2 [Computer Laboratory, University of Cambridge, CC-BY 2.0]。http://www.cl.cam.ac.uk/relics/archive_photos.html

**図 H.34** DEC PDP-1 [by Matthew Hutchinson, CC-BY 2.0]。http://commons.wikimedia.org/wiki/File:PDP-1.jpg


**図 H.35** IBM System/360 [Bundesarchiv, B 145 Bild-F038812-0014 / Schaack, Lothar, CC-BY-SA 3.0]。http://www.bild.bundesarchiv.de/archives/barchpic/search/_1420437377/

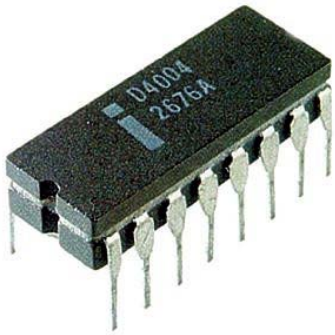



**図 H.36** Intel 4004 プロセッサ [by LucaDetomi (@it.wikipedia.org), CC-BY-SA 3.0]。`http://commons.wikimedia.org/wiki/File:Intel_4004.jpg`

**図 H.37** Altair 8800 [by Michael Holley, CC-BY-SA 3.0]。`http://commons.wikimedia.org/wiki/File:Altair_8800_Computer.jpg`

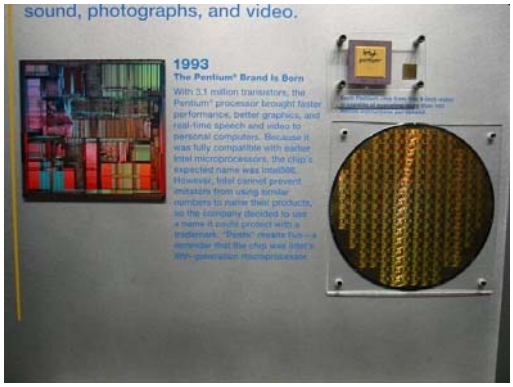



**図 H.38** Intel Pentium プロセッサ [インテル博物館，佐野健太郎撮影]。

**図 H.39** Cray-1 [ドイツ博物館，佐野健太郎撮影]。

# 章末問題解答

## 1 章

【1】 (a) 21 (b) 20.6875

【2】 (a) 10011101000 (b) 1111010.1010110011001100 (c) 11110101.1011001100110011

【3】 いずれの数も MSB が 1 であるため，補数変換により正の数 $X$ にし，10 進数に変換する。$-X$ が答。

-102

(a) ~~$=38$~~ (b) 略。(Windows 付属の「電卓」，Mac 付属の「計算機」などには 10 進数・2 進数の変換機能があるため，答え合わせに活用するとよい)

【4】 負号 $-$ を除いた正の数 $X$ の 8 ビット 2 進数表現に対し，補数変換を行う。

(a) 10110001 (b) 略

【5】 (a) 01011101 (b) 11011110

【6】 (a) 00000110 (b) 略

【7】 (a) $1.01000\times2^4$ となるため, 01000001101000 (b) $1.00011\times2^0$ となるため, 00111111100011

【8】 (a) $1.01010\times2^4$ となるため, 01000001101010 (b) $1.01011\times2^4$ となるため, 01000001101011

【9】 いずれの場合にも，正規化した仮数の表現には小数点以下 6 桁以上が必要であったが，丸めにより 5 桁に抑えたため誤差が生じた。

## 2 章

【1】 真理値表は以下のようになる。

| $A$ | $B$ | $C$ | $A(BC+\overline{C})$ | $A(B+\overline{C})$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

【2】 $BC+\overline{C}=BC+\overline{C}$ を証明する。以下のように式変形すればよい。

$$
\begin{aligned}
BC+\overline{C} &= BC+(B+\overline{B})\overline{C} \\
&= BC+B\overline{C}+\overline{B}\overline{C} \\
&= BC+B\overline{C}+B\overline{C}+\overline{B}\overline{C} \\
&= B(C+\overline{C})+(B+\overline{B})\overline{C} \\
&= B+\overline{C}
\end{aligned}
$$

【3】 左辺の $A(BC+\overline{C})$ の双対は $A+(B+C)\overline{C}$ となり右辺の $A(B+\overline{C})$ の双対は $A+B\overline{C}$ と

なる。真理値表は次のようになり，これらが同じ論理関数であることが確認できる。

| $A$ | $B$ | $C$ | $A+(B+C)\overline{C}$ | $A+B\overline{C}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**【4】** **【1】**の真理値表から

$$A\overline{B}\overline{C}+AB\overline{C}+ABC$$

**【5】** 真理値表は次のようになる。

| $A$ | $B$ | $C$ | $f(A,B,C)$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

また，主加法標準形は次のようになる。

$$\overline{A}\overline{B}\overline{C}+\overline{A}BC+A\overline{B}C+AB\overline{C}$$

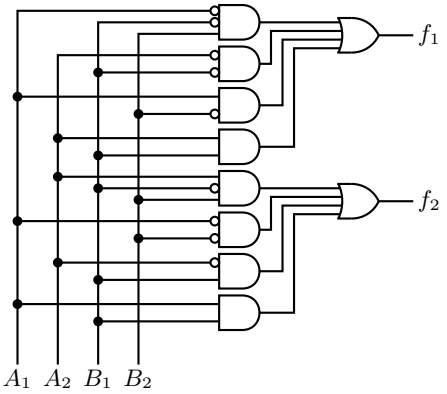
**【6】** $A+B\overline{C}$ を否定論理積のみで表すと

$$\begin{aligned}A+B\overline{C}&=A+(B\uparrow\overline{C})\uparrow(B\uparrow\overline{C})\\&=(A\uparrow A)\uparrow(((B\uparrow\overline{C})\uparrow(B\uparrow\overline{C}))\uparrow((B\uparrow\overline{C})\uparrow(B\uparrow\overline{C})))\end{aligned}$$

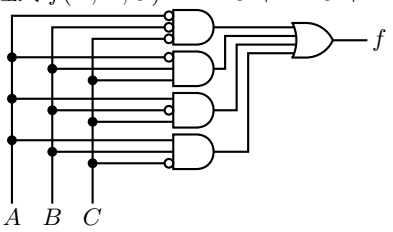
となる。

## 3章

**【1】** $f_1$，$f_2$ ともに最も簡単な 4 通りの論理式のうち最初のものを回路化すると，以下のようになる。

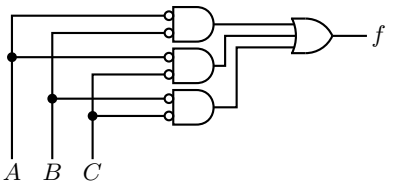



【2】 論理式 $f(A,B,C) = \overline{A}\overline{B}\overline{C} + \overline{A}BC + A\overline{B}C + AB\overline{C}$ を回路化して以下を得る。



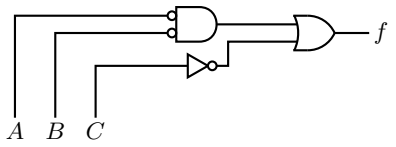

【3】 以下のカルノー図より論理式 $f(A,B,C) = \overline{A}\overline{B} + \overline{A}\overline{C} + \overline{B}\overline{C}$ が得られ，以下の回路図を得る。

| $A$ \ $BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | | 1 |
| 1 | 1 | | | |




【4】 カルノー図は以下のようになり，$f(A,B,C) = \overline{A}\overline{B} + \overline{C}$ を得る。

| $C$ \ $AB$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | × | 1 |
| 1 | 1 | | × | |

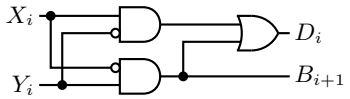



【5】 $i$ 桁目の入力を $X_i$，$Y_i$，出力を $D_i$，繰下がりを $B_i$ と書くと，半減算器，全減算器の真理値表は，それぞれ以下の通りとなる。

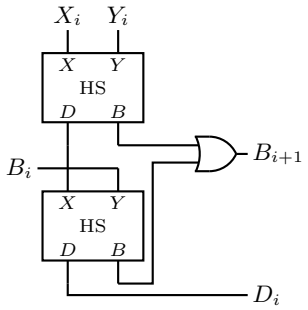
| $X_i$ | $Y_i$ | $D_i$ | $B_{i+1}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

| $X_i$ | $Y_i$ | $B_i$ | $D_i$ | $B_{i+1}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

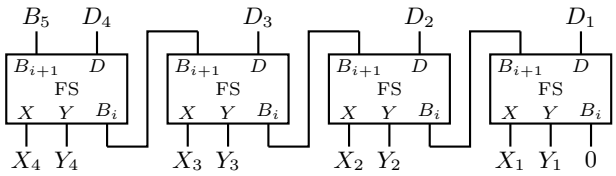
半減算器の回路図は，真理値表より以下のとおりとなる。



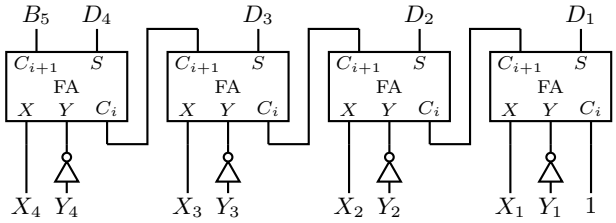

全減算器の回路図は，上記の真理値表から直接作成してもよいが (省略)，半減算器 HS を 2 つ用いて以下のように構成することもできる。




この全減算器 FS を 4 つ用いて，4 ビット減算器は以下のように構成できる。



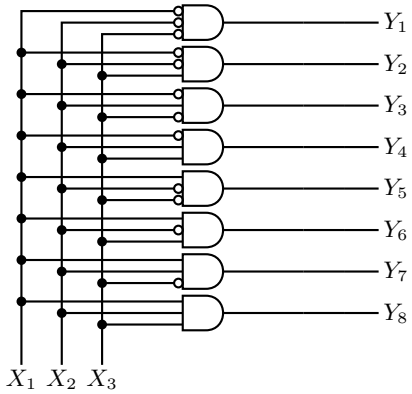

【6】 $n = 4$ として，以下のように構成できる。引く数 $Y_i$ に対する 2 の補数を，各ビットの反転 (否定) と，最下位 FA のキャリー入力を $+1$ にすることで表現する。




【7】 3 ビットデコーダの真理値表は以下の通りである。なお以降では，見やすさのため，真理値表のうち値が 0 の欄は記入を省略する場合がある。

| $X_1$ | $X_2$ | $X_3$ | $Y_1$ | $Y_2$ | $Y_3$ | $Y_4$ | $Y_5$ | $Y_6$ | $Y_7$ | $Y_8$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | | | | | | | |
| 0 | 0 | 1 | | 1 | | | | | | |
| 0 | 1 | 0 | | | 1 | | | | | |
| 0 | 1 | 1 | | | | 1 | | | | |
| 1 | 0 | 0 | | | | | 1 | | | |
| 1 | 0 | 1 | | | | | | 1 | | |
| 1 | 1 | 0 | | | | | | | 1 | |
| 1 | 1 | 1 | | | | | | | | 1 |

これより以下の回路図が得られる。



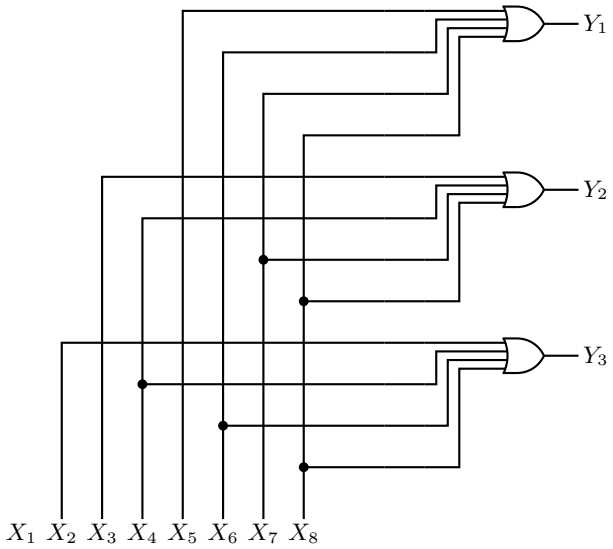

3 ビットエンコーダの真理値表は以下の通りである。ただし記載のない入力行に対する出力はドントケアとする。

| $X_1$ | $X_2$ | $X_3$ | $X_4$ | $X_5$ | $X_6$ | $X_7$ | $X_8$ | $Y_1$ | $Y_2$ | $Y_3$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | 0 | 0 | 0 |
| | 1 | | | | | | | 0 | 0 | 1 |
| | | 1 | | | | | | 0 | 1 | 0 |
| | | | 1 | | | | | 0 | 1 | 1 |
| | | | | 1 | | | | 1 | 0 | 0 |
| | | | | | 1 | | | 1 | 0 | 1 |
| | | | | | | 1 | | 1 | 1 | 0 |
| | | | | | | | 1 | 1 | 1 | 1 |

これより論理式は

$$Y_1 = X_5 + X_6 + X_7 + X_8$$
$$Y_2 = X_3 + X_4 + X_7 + X_8$$
$$Y_3 = X_2 + X_4 + X_6 + X_8$$

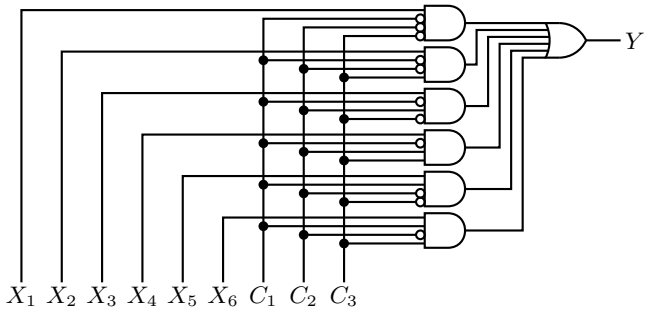
と表され，回路図は以下の通りとなる。




**【8】** 6 入力 $X_1, \cdots, X_6$ を選択するためには, 3 ビットの制御入力が必要となる。これを $C_1, C_2, C_3$ として，出力がドントケアとなる行を省略すると，真理値表は以下のようになる。

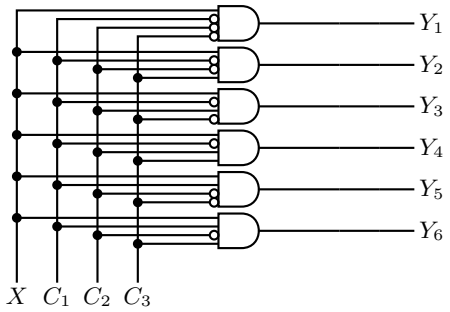
| $C_1$ | $C_2$ | $C_3$ | $Y$ |
|---|---|---|---|
| 0 | 0 | 0 | $X_1$ |
| 0 | 0 | 1 | $X_2$ |
| 0 | 1 | 0 | $X_3$ |
| 0 | 1 | 1 | $X_4$ |
| 1 | 0 | 0 | $X_5$ |
| 1 | 0 | 1 | $X_6$ |

回路図は以下のとおり。



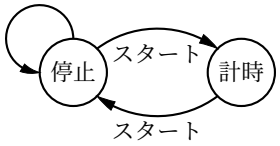

同様に，6 出力デマルチプレクサの真理値表と回路図は以下のとおりである。

| $C_1$ | $C_2$ | $C_3$ | $Y_1$ | $Y_2$ | $Y_3$ | $Y_4$ | $Y_5$ | $Y_6$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | $X$ | | | | | |
| 0 | 0 | 1 | | $X$ | | | | |
| 0 | 1 | 0 | | | $X$ | | | |
| 0 | 1 | 1 | | | | $X$ | | |
| 1 | 0 | 0 | | | | | $X$ | |
| 1 | 0 | 1 | | | | | | $X$ |

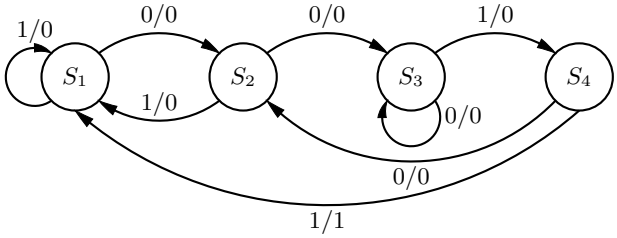



## 4 章

**【1】** リセット/表示を 00:00:00 にする




**【2】**



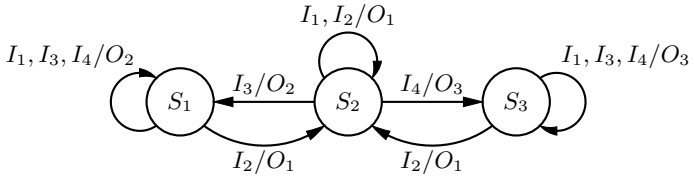

**【3】** 状態，入力，出力の集合を以下のように定める。ここで，フロアセンサによるフロア感知とボタン押下は同時には発生しないものとする。

状態: $S_1$ (上昇中), $S_2$ (停止中), $S_3$ (下降中)

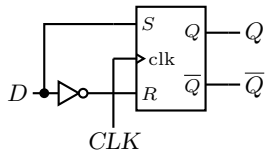
入力: $I_1$ (入力なし), $I_2$ (フロア感知), $I_3$ (上昇ボタン押下), $I_4$ (下降ボタン押下)

出力: $O_1$ (出力なし), $O_2$ (モータ上昇指令), $O_3$ (モータ下降指令)

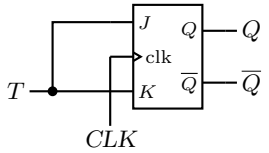
状態遷移図は以下のとおりとなる。ただし，上昇・下降ボタンによる操作は停止中にのみ可能であるとした。



【4】




【5】




## 5章

【1】 入力 $x$ の系列 “0000” に対して出力 $z$ が 1 となる順序回路の状態遷移表・出力表は，問図 4.1 より，以下のとおりとなる。

| | $x$ | |
|---|---|---|
| | 0 | 1 |
| $S_1$ | $S_2$ | $S_1$ |
| $S_2$ | $S_3$ | $S_1$ |
| $S_3$ | $S_4$ | $S_1$ |
| $S_4$ | $S_4$ | $S_1$ |

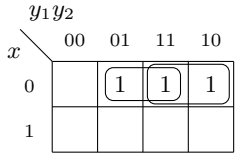
| | $x$ | |
|---|---|---|
| | 0 | 1 |
| $S_1$ | 0 | 0 |
| $S_2$ | 0 | 0 |
| $S_3$ | 0 | 0 |
| $S_4$ | 1 | 0 |

状態 $S_1, S_2, S_3, S_4$ に符号 00, 01, 10, 11 を割り当てると，状態遷移・出力を表す真理値表は以下のようになる。ただし，現時刻の状態変数を $y_1, y_2$，次時刻の状態変数を $Y_1, Y_2$ とする。

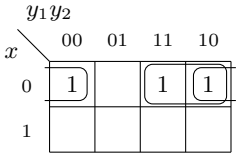
| $x$ | $y_1$ | $y_2$ | $Y_1$ | $Y_2$ | $z$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

ただちに得られる $z = \overline{x}y_1y_2$ は明らかに最も簡単な論理式である。$Y_1$ と $Y_2$ についてはカルノー図をかくと，それぞれ

$Y_1$




$Y_2$

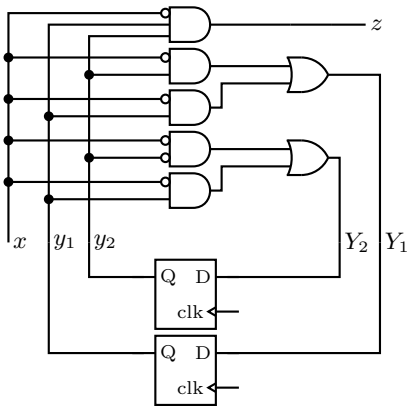



となり，以下の最も簡単な論理式を得る。

$$Y_1 = \overline{x}y_1 + \overline{x}y_2$$

$$Y_2 = \overline{x}y_1 + \overline{x}\overline{y_2}$$

よって回路図は以下のとおりとなる。




なお，$Y_1$ と $Y_2$ には共通の積項 $\overline{x}y_1$ が含まれている。これに該当する回路 (上から 3 個目と 5 個目の AND ゲート) を共有すれば，さらに簡単な回路が得られる。

同様に，入力系列 “0011” を受け取って 1 を出力する順序回路は，4 章の問題【 **2** 】解答の状態遷移図から，以下の状態遷移表，出力表，それらに対応する真理値表が得られる。

| | $x$ | |
|---|---|---|
| | 0 | 1 |
| $S_1$ | $S_2$ | $S_1$ |
| $S_2$ | $S_3$ | $S_1$ |
| $S_3$ | $S_3$ | $S_4$ |
| $S_4$ | $S_2$ | $S_1$ |

| | $x$ | |
|---|---|---|
| | 0 | 1 |
| $S_1$ | 0 | 0 |
| $S_2$ | 0 | 0 |
| $S_3$ | 0 | 0 |
| $S_4$ | 0 | 1 |

| $x$ | $y_1$ | $y_2$ | $Y_1$ | $Y_2$ | $z$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |

$Y_1$ と $Y_2$ のカルノー図は以下のとおりであり，

$Y_1$

| $x$ \ $y_1y_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | | 1 |
| 1 | | | | 1 |

$Y_2$

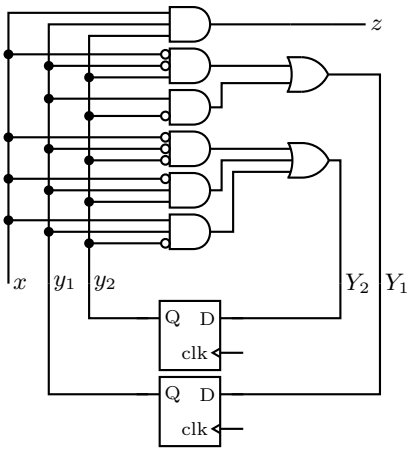
| $x$ \ $y_1y_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | | 1 | |
| 1 | | | | 1 |

以下の最も簡単な論理式と回路図が得られる。

$$Y_1 = \overline{x}\overline{y_1}y_2 + y_1\overline{y_2}$$

$$Y_2 = \overline{x}\overline{y_1}\overline{y_2} + \overline{x}y_1y_2 + xy_1\overline{y_2}$$

$$z = xy_1y_2$$



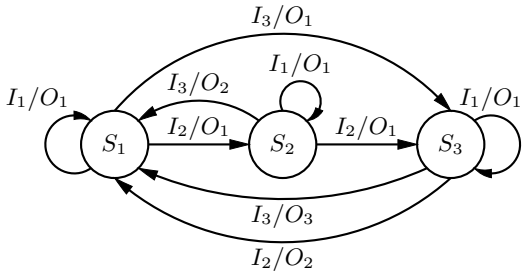

【2】 投入済み金額 0 円，50 円，100 円を区別する必要があるため，最小の状態数は 3 である。状態，入力，出力の集合を以下のように定める。

状態: $S_1$ (0 円投入済み), $S_2$ (50 円投入済み), $S_3$ (100 円投入済み)

入力: $I_1$ (入力なし), $I_2$ (50 円投入), $I_3$ (100 円投入)

出力: $O_1$ (出力なし), $O_2$ (品物を出す), $O_3$ (品物と釣り銭 50 円を出す)

状態遷移図は以下のとおり。




【3】 状態遷移表，出力表は以下のとおりとなる。

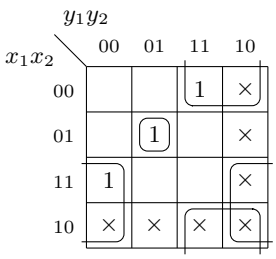
| | $I_1$ | $I_2$ | $I_3$ |
|---|---|---|---|
| $S_1$ | $S_1$ | $S_2$ | $S_3$ |
| $S_2$ | $S_2$ | $S_3$ | $S_1$ |
| $S_3$ | $S_3$ | $S_1$ | $S_1$ |

| | $I_1$ | $I_2$ | $I_3$ |
|---|---|---|---|
| $S_1$ | $O_1$ | $O_1$ | $O_1$ |
| $S_2$ | $O_1$ | $O_1$ | $O_2$ |
| $S_3$ | $O_1$ | $O_2$ | $O_3$ |

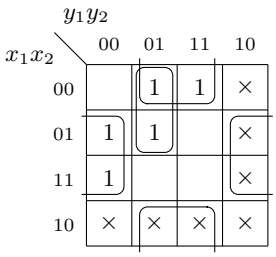
状態 $S_1, S_2, S_3$，入力 $I_1, I_2, I_3$，出力 $O_1, O_2, O_3$ にそれぞれ 00, 01, 11 を割り当てる。入力変数を $x_1, x_2$，出力変数を $z_1, z_2$，現時刻の状態変数を $y_1, y_2$，次時刻の状態変数を $Y_1, Y_2$ と書くと，状態遷移表，出力表を表す真理値表は，以下のようになる。

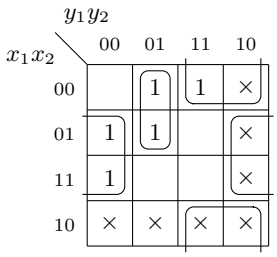
| $x_1$ | $x_2$ | $y_1$ | $y_2$ | $Y_1$ | $Y_2$ | $z_1$ | $z_2$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | × | × | × | × |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | × | × | × | × |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | × | × | × | × |
| 1 | 0 | 0 | 1 | × | × | × | × |
| 1 | 0 | 1 | 0 | × | × | × | × |
| 1 | 0 | 1 | 1 | × | × | × | × |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | × | × | × | × |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |

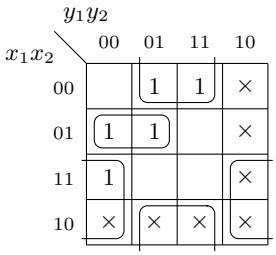
$Y_1$ のカルノー図は以下のとおり。

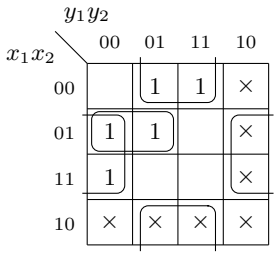


$Y_2$ は，以下の 4 通りがいずれも最も簡単な論理式を与える。









$z_1$ と $z_2$ のカルノー図は以下のようになる。

$z_1$

| $x_1x_2$ \ $y_1y_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | × |
| 01 | | | | × |
| 11 | | | 1 | × |
| 10 | × | × | × | × |

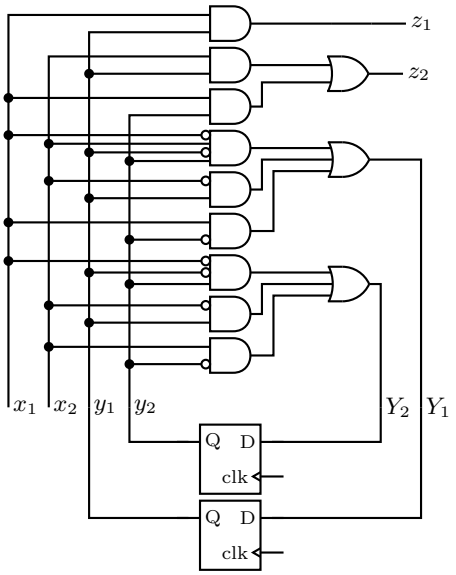
$z_2$

| $x_1x_2$ \ $y_1y_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | × |
| 01 | | | 1 | × |
| 11 | | 1 | 1 | × |
| 10 | × | × | × | × |

これらより，以下の最も簡単な論理式が得られる。

$$
\begin{aligned}
Y_1 &= \overline{x_1}x_2\overline{y_1}y_2 + \overline{x_2}y_1 + x_1\overline{y_2} \\
Y_2 &= \overline{x_1}\,\overline{y_1}y_2 + \overline{x_2}y_1 + x_2\overline{y_2} \\
&= \overline{x_1}x_2\overline{y_1} + x_1\overline{y_2} + \overline{x_2}y_2 \\
&= \overline{x_1}\,\overline{y_1}y_2 + x_2\overline{y_2} + \overline{x_2}y_2 \\
&= \overline{x_1}x_2\overline{y_1} + x_2\overline{y_2} + \overline{x_2}y_2 \\
z_1 &= x_1y_1 \\
z_2 &= x_2y_1 + x_1y_2
\end{aligned}
$$

$Y_2$ の 4 式のうち最初のものを採用すると，回路図は以下の通りとなる。




**【4】** 状態遷移表，出力表は以下のとおり。

| | $I_1$ | $I_2$ | $I_3$ | $I_4$ |
|---|---|---|---|---|
| $S_1$ | $S_1$ | $S_2$ | $S_1$ | $S_1$ |
| $S_2$ | $S_2$ | $S_2$ | $S_1$ | $S_3$ |
| $S_3$ | $S_3$ | $S_2$ | $S_3$ | $S_3$ |

| | $I_1$ | $I_2$ | $I_3$ | $I_4$ |
|---|---|---|---|---|
| $S_1$ | $O_2$ | $O_1$ | $O_2$ | $O_2$ |
| $S_2$ | $O_1$ | $O_1$ | $O_2$ | $O_3$ |
| $S_3$ | $O_3$ | $O_1$ | $O_3$ | $O_3$ |

状態 $S_1, S_2, S_3$ に符号 00, 01, 11 を，入力 $I_1, I_2, I_3, I_4$ に符号 00, 01, 10, 11 を，出力 $O_1, O_2, O_3$ に符号 00, 01, 10 をそれぞれ割り当てる。前問と同様に真理値表をかく。

| $x_1$ | $x_2$ | $y_1$ | $y_2$ | $Y_1$ | $Y_2$ | $z_1$ | $z_2$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | × | × | × | × |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | × | × | × | × |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | × | × | × | × |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | × | × | × | × |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

$Y_1, Y_2, z_1, z_2$ をカルノー図で表すと以下のとおりとなり，

$Y_1$

| $x_1x_2$ \ $y_1y_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | 1 | × |
| 01 | | | | × |
| 11 | | 1 | 1 | × |
| 10 | | | 1 | × |

$Y_2$

| $x_1x_2$ \ $y_1y_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | 1 | 1 | × |
| 01 | 1 | 1 | 1 | × |
| 11 | | 1 | 1 | × |
| 10 | | | 1 | × |

$z_1$

| $x_1x_2$ \ $y_1y_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | 1 | × |
| 01 | | | | × |
| 11 | | 1 | 1 | × |
| 10 | | | 1 | × |

$z_2$

| $x_1x_2$ \ $y_1y_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | | | × |
| 01 | | | | × |
| 11 | 1 | | | × |
| 10 | 1 | 1 | | × |

これらより以下の最も簡単な論理式を得る。

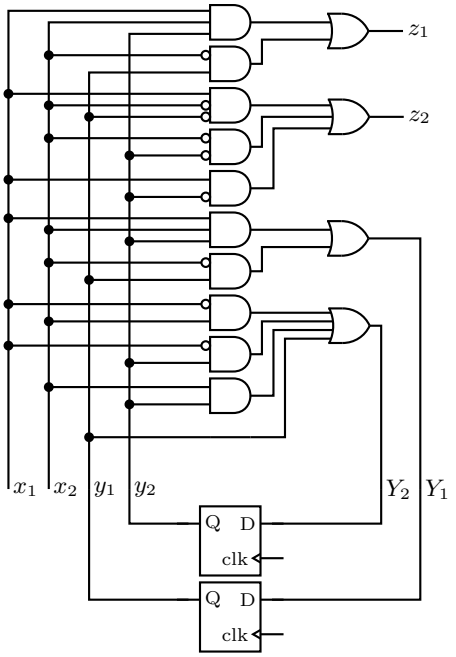
$$Y_1 = x_1x_2y_2 + \overline{x_2}y_1$$

$$Y_2 = \overline{x_1}x_2 + \overline{x_1}y_2 + x_2y_2 + y_1$$

$$z_1 = x_1x_2y_2 + \overline{x_2}y_1$$

$$z_2 = x_1\overline{x_2}\overline{y_1} + \overline{x_2}\overline{y_2} + x_1\overline{y_2}$$

回路図は以下の通りとなる。




なお，$Y_1$ と $z_1$ が全く同一の真理値を持つことに注目して，これらの回路を共有化すれば，より簡単な回路が得られる。

## 6 章

【1】 (a) $0x1234 + 20 = 0x1234 + 0x14 = 0x1248$

(b) $0x1234 - 8 = 0x1234 - 0x8 = 0x122c$

【2】 (a) $0x12345678 + 5 \times 4 = 0x12345678 + 20 = 0x12345678 + 0x14 = 0x1234568c$

(b) $0x12345678 + 128 \times 4 = 0x12345678 + 512 = 0x12345678 + 0x200 = 0x12345878$

【3】 (a)　レジスタ s0 を再利用しながら，かっこの順に計算していけばよい。

```
addu $s0, $s3, $s4
subu $s0, $s2, $s0
addu $s0, $s1, $s0
```

(b)　単純にかっこの順に計算しようとすると，$a - 15$ の結果と $b + c$ の結果の保存のために他のレジスタの破壊が必要となってしまう。式を展開して $x = a + 15 - b - c$ を計算すれば，他のレジスタを破壊せずにすむ。

```
addu $s0, $s1, 15
subu $s0, $s0, $s2
subu $s0, $s0, $s3
```

【4】 レジスタ s1 の値を $k$ と書くと，レジスタ s0 の値は上位 $k$ ビットと下位 $(32 - k)$ ビットが入れ替わることになる。これは $k$ ビットの左循環シフトにほかならない。

【5】 省略

## 7 章

【1】 例えば以下のようにして実現できる。

```
           move  $s0, $zero          # i = 0
           li    $t0, 10
FOR_BEGIN: slt   $t1, $s0, $t0       # i < 10
           beq   $t1, $zero, FOR_END

           ...                       # 中かっこの内容

           addu  $s0, $s0, 1         # i = i + 1
           j     FOR_BEGIN
FOR_END:   ...
```

【2】 (a) s0 は計算対象で，1 ビットずつシフトして行き，最後は 0 になる。t0 は最終結果を保持するレジスタであり，13 を 2 進数で表すと 1101 であることから，結果は ビット 1 の個数 3 になる。t1 は計算途中で使用し，ループごとに s0 の最下位ビットを取り出すのに使われる。最上位側にある 1 のビットを取り出した直後にプログラムは終了するため，常に最後は 1 になる。

(b) ラベル L1 への分岐は，レジスタ s0 が 0 になるまで繰り返される。最初にロードされて以降 s0 が更新されるのは 5 行めの srl 命令だけであり，これが実行される回数は ラベル L1 の命令の実行回数と同じである。2 進数 1101 が 0 になるまでに必要な 1 ビット右シフトの回数は 4 回であり，したがってラベル L1 の命令も 4 回実行される。

【3】 (a) s1 は処理すべき配列要素の残り数で，終了時は 0 になる。v0 はその時点までの最大値を保持するレジスタで，終了時は全体の最大値 22 になる。t0 は読み出した配列要素を格納するレジスタで，終了時は最後の要素 5 になる。 t1 は，それまでの最大値が読み出した配列要素より小さければ 1 になるレジスタで (これが 0 のときは L2 に分岐するため，最大値の更新がスキップされる)，最後のループでは 0 になる。

(b) ※ 1 は配列から要素を読み出す命令であり，配列長が 5 なので 5 回実行される。※ 2 は最大値を更新するときのみ実行される命令であり，配列から 10，20，22 が読み出されたときにのみ実行されるため，3 回実行される。

## 8 章

【1】 行アドレスデコーダには 0x12 = 18 が入力され，列アドレスデコーダには 0x34 = 52 が入力される。各デコーダによって対応する出力信号が駆動されるため，18 行，52 列のセルがアクセスされる。

【2】 $1 + 5 \times 10^{-2} \times 20 = 2$ [ns]

【3】 $1 + p \times 10^{-2} \times 20 = 1.5$ を $p$ について解いて，$p = 2.5$ [%]

## 9 章

【1】 (1) 高水準言語のプログラムを一括して機械語プログラムに変換するコンパイラとは異なり，高水準言語で記述されたプログラムを，逐次解釈し，機械語に変換しながら実行する方

式のこと。

(2) コンピュータの動作を具体的に指示する機械語でのプログラミングの生産性が低いため，コンピュータの動作を抽象的に指示できる，人間にとって扱いやすいプログラミングの言語が考えられてきた。そのような抽象度が高いプログラミング言語を高水準言語と呼ぶ。

(3) 正規表現とは，特定のパターンに適合する文字列の集合を表現するための記述方法のこと。コンパイラでは，字句解析器の動作を定義するために使われる。

(4) 高水準言語の構文規則を定義するためのメタ言語のひとつ。文法中で出てくる概念や変数を示す構文要素は，バッカス記法等により再帰的に定義される。コンパイラの構文解析処理では字句解析処理で得られた字句の並びを その再帰的に定義された構文規則に照らし合わせて文法違反がないことを確認している。

(5) 高水準言語プログラムの構成要素を木構造データとして表現したもの。この木の葉は終端記号となっており，それ以外の節は非終端記号になっている。さらに，プログラムの挙動とは関係ない情報を削除した構文木を抽象構文木と呼ぶ。

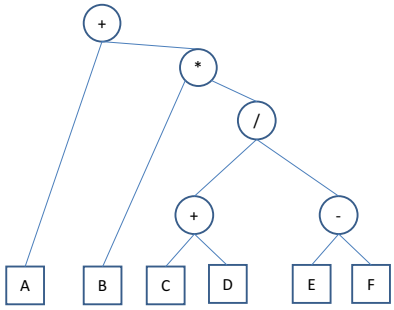
(6) 最適化とは，何らかの意味でプログラムをよりよいものにする処理のことである。例えば，同じプログラムがより高速に実行されるように無駄な命令実行を省いたりする。この最適化は，コンパイラによる言語処理の様々な段階で行われる。

**【2】** UNIX / Linux 環境での実行例を示す。プログラム 9.1 を test.lex というファイル名で保存し，それを使って「a+10」という文字列の字句解析をするには，以下のようなコマンド列を実行すればよい。実行結果もあわせて示す（% はプロンプト）。

```
% ls
test.lex
% flex test.lex
% ls
lex.yy.c  test.lex
% gcc lex.yy.c
% ls
a.out  lex.yy.c  test.lex
% echo "a+10" | ./a.out
An identifier: a
An operator: +
An integer: 10 (10)
%
```

flex コマンドによって，字句解析に必要となる様々な処理が自動的に付加された C プログラム (lex.yy.c) が生成される。それを gcc コマンドでコンパイルすることよって字句解析器の実行ファイル (a.out) が生成されている。上記の例では，echo コマンドを使って「a+10」という文字列を字句解析器へ渡している。その結果の出力から，それぞれの字句が識別子 (identifier)，演算子 (operator)，および整数 (integer) として正しく区別されていることがわかる。

【3】




## 10 章

【1】 通信要求に応じて送受信ノード間にリンクを確立しデータを転送する方式を回線交換方式という。これに対し，パケット交換方式ではデータを複数のパケットに分割し，それぞれのパケットに届け先ノード名を付けてネットワークに送り出す。送出されたパケットは，パケット中継機を経由して届け先ノードに届く。パケット交換方式は，回線交換方式と異なりリンクを特定の通信のために長時間占有することはないため，同時に複数のノード間のデータ転送を並行して行うことが可能であり，比較的小さなデータの通信や，通信相手が頻繁に変わるような用途に向いている。

【2】 複数のノードを接続するためのリンクの幾何学的形状のことをトポロジという。代表的なトポロジにはバス，リング，トーラスがある。バスは 1 本の接続線に全てのノードが接続されたトポロジであり，2 つのノードが通信を行っている間は他のノードが通信できないという欠点があるが，ブロードキャストが容易に行えるという長所がある。リングは複数のノードを環状に接続するトポロジであり，ノード数が増えても隣接するノード間のリンク数は変わらずにネットワークを拡張できるが，同時に最遠ノードへは多くの中間ノードを経ないとパケットが届かなくなる。トーラスはリングを例えば縦横の 2 次元に拡張した接続である。

【3】 通信プロトコルを独立した階層に分離することにより，ある階層の技術が変わったとしても階層間のやりとりを保てば全体として機能を維持することができるため。

【4】 TCP/IP はネットワークインターフェース層，インターネット層，トランスポート層，アプリケーション層から成る。ネットワークインターフェース層はノード間を物理的に接続するためのケーブルの種類や伝送符号・送信速度といった，物理的なデータ伝送技術や方式を定めている。Ethernet がその代表例である。インターネット層はインターネットにおけるパケット交換を実現するためのプロトコルを定めており，IP や ARP がその代表例である。トランスポート層は，TCP および UDP の 2 つのプロトコルにより，その上位層に位置する様々なアプリケーションが容易にデータ伝送を利用可能となるようにソケットに基づく API を提供している。アプリケーション層は個々のアプリケーションごとに必要な情報を送受信するためのプロトコルを定めており，HTTP や SMTP がその代表例である。

【5】 MAC アドレスはネットワーク機器ごとに与えられた世界でただ 1 つの固有の番号であり，送信先のノードを示す番号として使用される。フレーム送受信の手順は付録 G を参照のこと。

【6】 MAC アドレスはネットワーク機器ごとに与えられた世界でただ 1 つの固有の番号であり，ネットワークインターフェース層における物理的なノード間データ伝送に使用される。一方，IP アドレスは，変更が可能な論理的な番号であり，複数のサブネットワークが接続されて出

来たインターネット上でノード間のデータ送信を行う際に用いられる。

【7】 ネットワークアドレスとは，サブネットワークに付けられた IP アドレスであり，例えば上位の 24 ビットの部分を残し他は 0 となっている。そのサブネットワークに属するノードはネットワークアドレスにおける下位ビットのみが異なる IP アドレスを持つ。IP アドレスのうち，ネットワークアドレス部分を全て 1 とした 2 進数をサブネットマスクと呼び，ネットワークアドレス部を取り出すのに使用する。

【8】 TCP は，データを複数のセグメントに分割しながら IP パケットを送受信することによりソケット間のデータ通信を実現するためのプロトコルであり，パケットが相手に確実に届いたかどうかを確認しながら信頼性の高いデータ送信を実現するものである。一方，UDP は TCP から確認応答やパケット再送，フロー制御などを取り去った簡易なプロトコルであり，途中のパケットが多少失われても致命的でなく，むしろデータ送信速度の方が重要な用途に用いられる。

【9】 データを送信する際に，同一のノードにおいて動作する複数のアプリケーションのそれぞれを区別するために用いる仮想的な受信受付窓口のことをポートといい，その番号をポート番号と呼ぶ。ソケットは TCP/IP によりソフトウェアがデータを送受信するための仮想的なインターフェースであり，ノードの IP アドレスとポート番号の組で区別する。

【10】 SMTP, POP3, HTTP, FTP など。

【11】 インターネット上でドメイン名により送信先を指定する場合，ドメイン名からそのノードの IP アドレスへの変換を実現するために DNS が必要である。名前解決の動作は 10.3.2 項を参照のこと。

【12】 あらゆる IP アドレスに対しあらゆるサーバプログラムのポート番号を順に問い合わせていくことにより，サーバの存在とそこで動作しているサービスプログラムを検出することをポートスキャンという。この結果セキュリティホールのあるサーバが見つかると，不正な侵入を許すことにもつながる。

【13】 本来の意味では，他のプログラムを書き換えてその一部となり (感染)，それが実行された際に自己をさらに他のプログラムに感染させることにより自己増殖するプログラムのこと。近年では，他のプログラムの一部となるかどうかに関わらず，また自己増殖するかどうかに関わらず，コンピュータに被害をもたらす悪意のあるプログラムを総称してコンピュータウィルスと呼ぶ傾向がある。

## 付録 B

【1】 キャッシュラインが 32 バイトなので，32 ビットのメモリアドレスのうち，下位 5 ビットはキャッシュライン内の位置を表す。残る上位 27 ビットがタグとインデックスに割り当てられる。

(a) 256 K /32 = 8 K = 8192 [本]。

(b) 2 ウェイなので，8192 本のキャッシュラインを 2 分割し，4096 本が 2 セットあると考える。4096 本のうち 1 本を指定するのがインデックスなので，$2^{12} = 4096$ よりインデックスは 12 ビットである。残る 15 ビットがタグになる。

(c) キャッシュライン 1 本は 32 バイトのデータと 15 ビットのタグからなり，合計 $32 \times 8 + 15 = 271$ ビットである。これが 8192 本あるので，$8192 \times 271 = 1171456$ [ビット]

$= 277504$ [バイト] $= 271$ [K バイト] となる。

【2】 行列の 1 要素が 4 バイトであるため，1 キャッシュラインには 16 要素を保持できる。キャッシュライン数は $32\text{K}/64 = 512$ [本] である。$32\text{K}/4 = 8 \times 1024$ から，このキャッシュメモリは行列の 8 行分 (または 8 列分) を保持する容量を持っている。

(a) 簡単のため，まずは配列の先頭アドレスが 64 の倍数であることを仮定する。この仮定により，$(1,1)$ 要素のアドレスはキャッシュラインの先頭に位置することになる。行優先で走査する場合，最初の $(1,1)$ 要素にアクセスした際にキャッシュミスが発生してヒット時間 + ミスペナルティ時間が費やされるが，続く 15 要素は，空間的局所性のおかげで既にキャッシュラインに取り込まれているため，それぞれヒット時間のみを要する。ここまでの平均メモリアクセス時間は $\{(2+18)+15\times 2\}/16 = 3.125$ [ns] である。以降これが繰り返されるので，3.125 ns が最終的な平均メモリアクセス時間になる。配列の先頭アドレスが 64 の倍数でない場合は，最初のアクセスで後続 15 要素がちょうどキャッシュラインに取り込まれるとは限らないが，最初の 16 要素のアクセスのいずれかでこの状況に至る。この分の誤差は $1024 \times 1024$ 要素全体の平均アクセス時間にはほぼ影響しないと考えてよい。

(b) 列優先で走査する場合，最初の $(1,1)$ 要素へのアクセスでキャッシュミスが発生し，周囲の 16 要素がキャッシュラインに取り込まれる。次の $(2,1)$ 要素は $4 \times 1024 = 4096$ バイト先のアドレスに存在しており，4096 バイトはキャッシュライン 64 本に相当する。よって $(2,1)$ 要素が取り込まれたキャッシュラインのインデックスは，$(1,1)$ 要素のインデックスに 64 を加えたものとなる (インデックスが 512 以上になった場合は，512 で除した余りになる)。以降，キャッシュラインは 64 本おきにしか使用されない。結果，$(8,1)$ 要素のアドレスに対応するインデックスが $(1,1)$ 要素のものと同一になり，これにアクセスしたときに，キャッシュされていた $(1,1)$ 要素周辺の値は使われずに上書きされる。結局キャッシュは 1 度もヒットすることができず，平均メモリアクセス時間は $2 + 18 = 20$ [ns] である。

## 付録 C

【1】 例えば以下のようなプログラムで実現できる。

```
L1:     lw $t0, 0($a0)        # t0 ← mem[0xa0020000]; 状態ワード読み出し
        and $t0, $t0, 1       # t0 ← t0 & 1; 最下位ビット取り出し
        beq $t0, $zero, L1    # 0 と等しいうちは上記処理を繰り返す
        lw $v0, 4($a0)        # v0 ← mem[0xa0020004]
```

このような処理はビジーウェイト (busy wait) などと呼ばれる．条件成立を待つ間はプロセッサ時間が浪費されるため，特殊な状況 (例えば，条件成立までの時間が十分短いとわかっている場合など) 以外では避けるべきである．

【2】 取りこぼしを防ぐには 1 秒に $1000^2$ 回以上のポーリングが必要である。1 秒間のクロックサイクル数 $1000^3$ のうち $100 \times 1000^2$ サイクルがポーリングに費やされるので，その割合は 10 % である。

## 付録 D

【1】 実際の主記憶装置 (実記憶装置) に加えて補助記憶装置も記憶領域として使って大きな仮想記憶 (仮想空間) を構成し，OS が適切にデータを管理することによって，必要なデータが常に実記憶装置にあるかのようにプログラムに見せる記述のこと。

【2】 1 台のコンピュータで複数のタスクをあたかも同時に実行しているように見せる機能である。マルチタスクやマルチプロセスとも呼ばれる。

【3】 プログラムを実行する際，OS はメモリ領域やプロセッサ時間などのリソースを確保してプロセスと呼ばれる実行単位を生成して管理する。プロセスがプログラムの実行に必要なメモリ領域やプロセッサ時間などのすべてのリソースを割当てられた実行単位であるのに対して，スレッドはプロセッサ時間だけを割当てられた実行単位である。プロセスは複数のスレッドを持つことができ，そのような処理をマルチスレッド処理と呼ぶ。

【4】 実行中のタスクをなんらかの理由で中断し，他のタスクを実行することが必要になったときのために，そのような中断と再開を実現するための OS の機能のこと。割込みは，割込まれたタスクとの関係によって内部割込みと外部割込みに分類される。内部割込みは割込まれるタスクが要因となって生じるのに対して，外部割込みはそれ以外の外部要因によって生じる。

【5】 ファイルシステムとは，コンピュータの補助記憶装置に保存されている様々なデータを整理して管理するために提供されている仕組みである。データをファイルという単位で格納し，ディレクトリ (フォルダ) に分類して管理できるようになっている。それぞれのデータへのアクセス権限も OS によって管理されている。

## 付録 E

【1】 各命令について，(a) 分岐命令なら 1，それ以外なら 0，(b) 1 つ目の入力レジスタ番号，(c) 2 つ目の入力レジスタの値，(d) ALU の演算結果をそれぞれ答えればよい。

| | | | | |
|---|---|---|---|---|
| nor 命令 | (a) 0 | (b) 0 | (c) 0 | (d) 0xffffffff (または $-1$) |
| subu 命令 | (a) 0 | (b) 0 | (c) 0xffffffff (または $-1$) | (d) 1 |
| addu 命令 | (a) 0 | (b) 9 | (c) 1 | (d) 2 |
| beq 命令 | (a) 1 | (b) 9 | (c) 0 | |

【2】 状態遷移表の $\{Q_1, Q_0, \text{is\_branch}\} = \{1, 0, 1\}$ の行の遷移先および出力がすべてドントケア (×) となり，対応して，5 個のカルノー図すべての最も右下のセルがドントケアになる。しかしそれを考慮しても最終的な簡単化の結果は変わらない。

【3】 遅延 8 ns の経路がクリティカルパスであり，クロック周波数の上限はこれにより定まる。$1/(8 \times 10^{-9}) = 125 \times 10^6$ より 125 MHz となる。

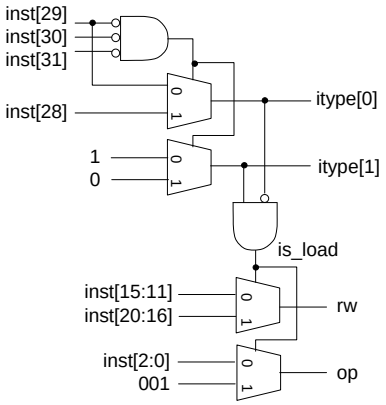
【4】 状態遷移表・出力表をまとめて書くと以下の通り。ただし次時刻の状態を $Q'$ と書く。

| $Q$ | itype | $Q'$ | PCen | GPRen | DMEMen |
|---|---|---|---|---|---|
| 00 | 00 | 01 | 0 | 0 | 0 |
| 00 | 01 | 01 | 0 | 0 | 0 |
| 00 | 10 | 01 | 0 | 0 | 0 |
| 00 | 11 | 01 | 0 | 0 | 0 |
| 01 | 00 | 10 | 0 | 0 | 0 |
| 01 | 01 | 00 | 1 | 0 | 0 |
| 01 | 10 | 10 | 0 | 0 | 0 |
| 01 | 11 | 10 | 0 | 0 | 0 |
| 10 | 00 | 00 | 1 | 1 | 0 |
| 10 | 01 | ×× | × | × | × |
| 10 | 10 | 11 | 0 | 0 | 0 |
| 10 | 11 | 11 | 0 | 0 | 0 |
| 11 | 00 | ×× | × | × | × |
| 11 | 01 | ×× | × | × | × |
| 11 | 10 | 00 | 1 | 1 | 0 |
| 11 | 11 | 00 | 1 | 0 | 1 |

カルノー図などを用いて簡単化すると，以下の論理式を得る。

$$
\begin{aligned}
Q'_1 &= \overline{Q_1} \cdot Q_0 \cdot \overline{\mathrm{itype}[0]} + \overline{Q_1} \cdot Q_0 \cdot \mathrm{itype}[1] + Q_1 \cdot \overline{Q_0} \cdot \mathrm{itype}[1] \\
&= Q_0 \cdot \overline{\mathrm{itype}[1]} \cdot \overline{\mathrm{itype}[0]} + \overline{Q_1} \cdot Q_0 \cdot \mathrm{itype}[1] + Q_1 \cdot \overline{Q_0} \cdot \mathrm{itype}[1] \\
&\quad \text{(いずれも最も簡単)} \\
Q'_0 &= \overline{Q_1} \cdot \overline{Q_0} + \overline{Q_0} \cdot \mathrm{itype}[1] \\
\mathrm{PCen} &= Q_0 \cdot \overline{\mathrm{itype}[1]} \cdot \mathrm{itype}[0] + Q_1 \cdot \overline{\mathrm{itype}[1]} + Q_1 \cdot Q_0 \\
\mathrm{GPRen} &= Q_1 \cdot Q_0 \cdot \overline{\mathrm{itype}[0]} + Q_1 \cdot \overline{\mathrm{itype}[1]} \\
\mathrm{DMEMen} &= Q_1 \cdot Q_0 \cdot \mathrm{itype}[0]
\end{aligned}
$$

**【5】** (itype[1], itype[0]) は，inst[31:29] が 000 のとき (0, inst[28]) であり，それ以外のとき (1, inst[29]) である。rw は，ロード命令のときは inst[20:16] であり，それ以外のときは inst[15:11] である。op は，ロード命令のときは加算を指示する 001 であり，それ以外のときは inst[2:0] である。マルチプレクサを用いてこれらを回路化すると以下のようになる。



## 付録 F

【1】 元の全実行時間を $T$ として，当該演算処理を $k$ 倍高速にしたとすると，全実行時間は

$$0.6T/k + 0.4T$$

になる。プログラム全体を $N$ 倍速くしたいときは，上式が $T/N$ と等しくなる必要があるので $1/N = 0.6/k + 0.4$，すなわち $k = 0.6/(1/N - 0.4)$ が要求される。

$N = 2$ のとき $k = 6$ であり，当該命令を 6 倍高速化する必要がある。$N = 2.5$ のときは $k = \infty$ (無限の速さが必要)，$N = 3$ のときは $k \simeq -9$ (時間をさかのぼることが必要) であり，当該命令のみの高速化では実現できない。

【2】 改良前の平均 CPI は $0.4 \times 2 + 0.3 \times 4 + 0.2 \times 4 + 0.1 \times 3 = 3.1$ であり，改良後の平均 CPI は $0.4 \times 2 + 0.3 \times 3 + 0.2 \times 3 + 0.1 \times 5 = 2.8$ に改善する。

【3】 最も安直な解として以下のプログラム例が挙げられる。

```
        .set noreorder
        move $t0, $zero
        lw   $s0, 0($s1)
        or   $zero, $zero, $zero    # delay slot; no operation
L1:     and  $t1, $s0, 1
        addu $t0, $t0, $t1
        srl  $s0, $s0, 1
        bne  $s0, $zero, L1
        or   $zero, $zero, $zero    # delay slot; no operation
        sw   $t0, 0($s1)
        .set reorder
```

すなわち，遅延スロットに何もしない命令を挿入することで正しく動作することを保証している。何もしない命令を実行する無駄を避けたい場合は，例えば

```
        .set noreorder
        lw   $s0, 0($s1)
        move $t0, $zero             # delay slot
L1:     and  $t1, $s0, 1
        srl  $s0, $s0, 1
        bne  $s0, $zero, L1
        addu $t0, $t0, $t1          # delay slot
        sw   $t0, 0($s1)
        .set reorder
```

のようにすることができる。lw 命令の直前にあった or 命令や，bne 命令の 2 個前にあった addu 命令は，それぞれ lw や bne の後ろに移してもプログラムの意味が変わらないため，このような交換が可能である。

なお，冒頭の `.set noreorder` は，アセンブラが命令順序を入れ替えることを禁止する疑似命令 (アセンブラへの指示文) であり，最後の `.set reorder` はそれを解除する疑似命令である。本書の他のプログラムのように `.set noreorder` がない場合，アセンブラが (元のプログラムは遅延スロットの存在を考慮していないとみなして) 適切な命令順序への入れ替

えと，必要であれば何もしない命令の挿入を行う。