コンピュータのしくみ

放送大学 岡部 洋一

まえがき

近年，ラジオ，テレビ，ビデオ，携帯電話などに代表される家電製品の中核がLSI化され，その内部はほとんどわからなくなってしまった。かつては，ラジオ少年，無線マニアと呼ばれる若者がいたが，現在では彼等も製品を買ってきて使うだけで，中味は知らないようになってしまった。

いわゆるブラックボックス化であるが，はたしてそれでよいのであろうか。科学は，未だ人類が知らないことを，より知っていこうという努力の結果，発展してきた。家電製品などは，自然物ではなく人工物であり，理解はもっと容易である。それすら，理解を放棄してしまうのでは，将来の科学技術の発展はとても覚つかない。

幸いにしてコンピュータは驚くほど簡単な原理で作られている。そのほぼ全容を理解することは，他の機器に比べると極めて容易である。せめて，コンピュータのしくみと動作原理ぐらいは理解して欲しいというのが，この講義の目的である。

ご自身が，科学技術に対し，進取の気性を持っていただきたいという希望と，特に放送大学の学生さんは，他への影響力も大きいことから，周りにそのような気運を醸成していただきたいと希望している。

なお，放送教材は概念を伝えるように構成しているが，印刷教材である本書は，かなり詳細にいたるまで記載している。そのため，初めての人にはやや難しいかも知れない。逆に，その気になればコンピュータを設計できるほどのレベルまで記載したつもりである。初めての人は，そのつもりで，本書のすべてを理解できないからとがっかりしないで，まず概要をつかむようにして欲しい。そして，将来，より興味が沸いてきたら，再び本書に立ち返られることを期待する。

目　　次

放送とのおよその対応表を掲載する。

放送	本テキスト
第 1 回	第 1 章
第 2 回	第 2 章，第 3 章 3.2 節まで
第 3 回	第 3 章残り，第 4 章 4.3 節まで
第 4 回	第 4 章残り
第 5 回	第 5 章 5.2 節まで
第 6 回	第 5 章残り
第 7 回	第 6 章 6.3 節まで
第 8 回	第 6 章残り
第 9 回	第 7 章
第 10 回	第 8 章
第 11 回	第 9 章 9.5 節まで
第 12 回	第 9 章残り
第 13 回	第 10 章 10.3 節まで
第 14 回	第 10 章残り
第 15 回	第 11 章

1 ディジタルとは

《目標＆ポイント》ディジタルとは何か，アナログとの比較を行う。ディジタル回路の象徴的存在はコンピュータであるが，その中心となっているマイクロプロセッサの概要，また近年，アナログ信号をディジタル回路で処理する場合のしくみについても説明する。

《キーワード》ディジタル，アナログ，コンピュータ，マイクロプロセッサ，AD 変換，DA 変換

1.1 ディジタル回路

現代はまさに**電子回路 (electronic circuit)** の時代である。家電製品のような身の回りにあるものから，世界にまたがる通信システムにまで，ありとあらゆるところに電子回路が使われている。

電子回路というと**アナログ回路 (analog circuit)** と**ディジタル回路 (digital circuit)** がある。しかし，その大部分はディジタル回路であり，特に，**マイクロプロセッサ (micro-processor)** と呼ばれる，情報を自由に処理できる回路が入っていることが多い。

最近はほとんどの電子回路が集積回路化され，人間の目で見えるサイズをはるかに下回るようになってきた。そのため，人々の関心が，その動作原理というよりは，その利用の方法に向くようになってきている。つまり，動作の第一原理がわからなくても構わないような心理状態に陥っている。しかし，物事を本当に理解できるためには，可能な限り内部ま

で立ち入る必要があろう。

幸いにして，ディジタル回路の動作原理は極めて簡単である。せめて，この簡単なディジタル回路のしくみぐらいは理解しておきたいものである。本書では，基礎的な論理回路から順に説き起こし，時系列信号を処理できるシークエンス回路，コンピュータといった順に，より高次のディジタル回路のしくみについて説明する。

さて，多くの家電製品などでは，マイクロプロセッサは操作ボタンの働きを理解して，入力を参照しながら機器に適した出力を発生するという，いわば黒子の役割を演じているが，**コンピュータ (computer)** の世界では情報処理そのものが目的であるため，マイクロプロセッサはまさに王様であり，それに各種の周辺装置が繋がっている。本書では，コンピュータを最終のゴールとして，主としてマイクロプロセッサのしくみを理解する。

1.2　ディジタルとアナログ

電子回路の多くは，外部から何らかの情報を取り入れ，それを処理して，信号を外部へ送り出す形となっている。例えば，使用量が刻々変化する給水系の，タンクの水位を一定にすることを考えよう。この制御系は，水位を電気信号へ変換するセンサーを持った入力部分と，その電気信号を処理して，制御に必要な電気信号を作り出す処理部分と，処理された電気信号を変換して，タンクへの流入量を制御する弁を動かす出力部分，の 3 部分からなっている。なかには，電子時計のように，内部で信号を生成し，それを処理して，出力する，つまり，処理部分と出力部分しかない回路も存在するが，例外的なものであろう。

これら外界から取り入れる情報，回路内の電気信号，外界へ送り出す

情報は，大きく分けて，アナログ量とディジタル量に分類される。アナログとかディジタルという言葉は，取り扱う信号の性質に対して，つけられたものである。

温度という量は，例えば，10°C と 11°C の間の 1°C の間に 10.1°C とか 10.11°C とかいくらでも無数に取りうる値を持つ。このように連続的に値を取りうる**連続量 (continuous value)** を**アナログ (analog)** 量という。光の強度も水位もみなアナログ量である。アナログ量をアナログ電気信号に変化したものを入力とし，それを処理してアナログ電気信号を出力として出す回路をアナログ回路と呼ぶ。

これに対し，とびとびの値をとる量は**ディジタル (digital)** 量と呼ばれる。例えば，パチンコ玉の数，モールス信号を送る電鍵の開と閉の状態といった**不連続量 (discontinuous value)** は，ディジタル量である。このようなディジタル量を入力とし，それを処理してディジタルの出力を出す回路をディジタル回路と呼ぶ。

1.3　2 進ディジタル回路

究極の不連続量は，たった二つしか値を取りえないようなケースである。よくディジタルというと 0 または 1 という言葉を聞くと思うが，ほぼすべてのディジタル回路がこのたった二つの値を組み合わせて構成されている。これを **2 進 (binary)** 系という。0，1 に対応する電気信号としては普通，低い電圧レベル (通常 0V) と高い電圧レベル (通常は正の電源電圧) を用いる。

たった二つでは二つの区別しか伝えられないと思うかもしれないが，**図** 1.1 のように，伝達に複数の信号線を使えば，多くの状態を伝えることができる。例えば，2 本の信号線を使えば，2 本の線に 00，01，10，11

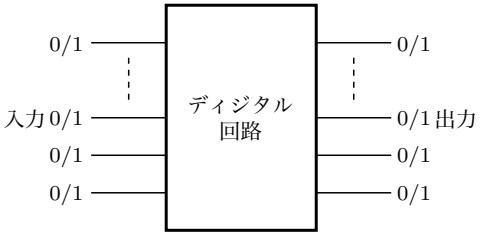


図 1.1　複数の 2 進信号線による入出力処理

の四つの電圧を載せることにより，四つの状態を伝えることができる。

二つの選択肢の一方を選ぶときの情報量を 1 **ビット (bit)** と呼ぶ。したがって，0，1 を送るチャネルが n 個ある場合，$n$bit の情報量を伝える能力があることになる。2 進の情報伝達の線数を**ビット幅 (bit width)** または単に**幅 (width)** というので，**図** 1.1 の入出力は幅 $n$bit ともいう。

自然数のように取りうる値がいくつかあるものは，**2 進表現 (binary representation)** し，その 2 進表現の 0 と 1 の組合せを回路の入力とすればよい。数の 2 進表現とは 0，1，2，3，4，... を 0，1, 10，11，100，... と，たった 2 になるだけで 1 桁繰り上がる数の表現法である。ちょっとわかりづらいかもしれないが，10 で 1 桁繰り上がる **10 進表現 (decimal representation)** と丁寧に比較していくと理解できる。

1960 年代ごろまでは電子回路と言えばアナログ回路を指した。しかし現在は，それが急速にディジタル化しつつある。パソコンのような純粋なディジタル機器はいうまでもないが，計測や制御，製造機械から家庭電化製品にいたるまで，あらゆるものにディジタル技術が応用されている。それは，ディジタル技術が，複雑な機能を容易に実現できる能力を持っているからである。

ディジタル回路はアナログ量の処理には適していないように思われる

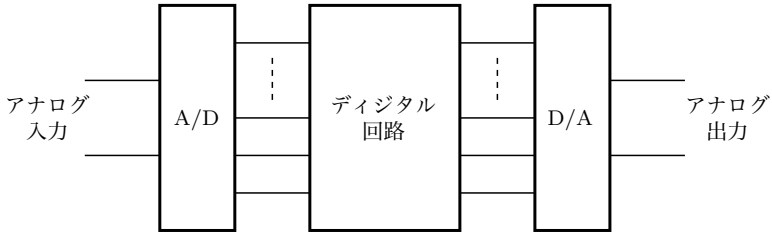


図 1.2　ディジタル回路によるアナログ処理

だろうが，決してそのようなことはない。アナログ量を，ある小さな刻みを単位にして自然数で表現し，それを 2 進表現して複数の線によりディジタル回路の入力にすればよいわけである。またディジタル回路の出力も複数の 2 進の線により取り出し，それを逆に変換して，極めて多値の出力として出せば，あたかもアナログ回路のように動作させることができる。

この場合，入力のアナログ量は一定の刻みに丸められてしまうし，出力も完全なアナログ量とは言い難い。しかしこのようなディジタル化に伴う誤差は現実にはほとんど問題とはならない。タンクの水位を 1mm 以下の単位まで測ることは多くの場合意味がないし，波などがあれば，そもそも測定すらできない。水流のバルブを 1μm の確度で制御してもほとんど意味がない。このようにどんなアナログ量にも必要な精度や確度があるからである。十分精度を上げれば，水の量も，水分子数で表現できるし，光の強さも光子数で表現できるから完全に量子化されてしまう。一見，荒唐無稽のような話に聞こえるかも知れないが，こうした限界に達した技術もないわけではない。なお，ディジタル化は**量子化 (quantization)** とも言われる。

図 1.2 のように，入力側のアナログ量を 2 進ディジタル量へ変換する**ア**

ナログ-ディジタル変換器 (analog-digital convertor) または **AD 変換器 (AD convertor)** と，出力側の 2 進ディジタル量をアナログ的な量へ変換する**ディジタル-アナログ変換器 (digital-analog convertor)** または **DA 変換器 (DA convertor)**[1] を精度良く作成しておけば，ディジタル回路によるアナログ処理は途中の回路による歪みなどが発生しないため，むしろ品質の良い処理ができる。

1.4 ディジタルの特徴

ディジタルの強みは，信号のレベルとレベルの間に隙間があることである。隙間の間隔を十分大きくとれば，信号伝達や記録の際，雑音に強くなる。つまり，誤りを少なくできる。このため，従来アナログ処理が主流であったオーディオなどの分野でも，光ディスクに見られるディジタル録音のように，こういった処理が大幅に取り入れられるようになっている。

さらに，ディジタルは，プログラミングにより，いくらでも複雑な処理を行うことができる。ディジタル回路の代表であるコンピュータの出現により，いくらでも複雑な情報処理ができるようになったことから，かつては機械的部品の組合せなどで処理してきた制御機構なども，ほとんどすべて，電気信号に変換された後に電子回路で処理されるようになってきている。例えば，完全に機械的仕掛けだった自動車エンジンの制御なども，現在はほとんど電子的に処理されるようになってきている。さらに，家電製品に代表される多くの製品の回路には，必ずその中心にコンピュータが配置されるようになってきている。ただし，ディジタル回路でアナログ出力を得ようとすると，先に示したように，精度の高いディ

1) 変換回路ともいう。

ジタル-アナログ変換器が必要である。

一方で，アナログ回路の利用比率はかなり下がってきたとはいえ，半導体の**素子 (device)**[2] の持つ限界速度を十分に生かすことができ，テレビ，携帯電話といった無線通信に使われる高周波の処理には欠かせない。さらに，最近は脳機能との類似性から，再評価されつつある。

演習問題 1

問題 1.1　次の用語を理解したかどうか確認せよ。

1）アナログ，ディジタル

2）ビット

3）ビット幅

4）AD 変換器，DA 変換器

問題 1.2　いろいろな時計の動作原理を考え，ディジタルかアナログかを考察してみよ。

問題 1.3　アナログテレビとディジタルテレビは，なぜ，そのように呼ばれるか，調べてみよう。

問題 1.4　幅 4bit の信号線があるとき，何種類の情報を伝えることができるだろうか。また自然数を 4bit の 2 進表現した場合，最大値はいくつになるだろうか。最大値を 2 進表現と 10 進表現で示せ。

2）素子とはシステムを構成している部品のことで，本書で説明しているディジタル回路やアナログ回路では，回路を構成するトランジスタのこと。

2 スイッチ素子

《目標＆ポイント》コンピュータを構成する回路の基礎となるのが，論理回路である。論理回路は，電気信号により開閉されるスイッチを使って構成される。本章では，いろいろなスイッチ，特に現在，主として使われている半導体スイッチである FET を中心に，その構造や機能について述べる。

《キーワード》スイッチ素子，リレー，電子管，トランジスタ，FET，MOS，n-MOS，p-MOS，集積回路，ムーアの法則

2.1 スイッチ素子

論理回路の入出力は 2 進化されており，0 と 1 のみで構成されている。回路というからには電気信号を処理する回路であり，0 とは文字通り 0V の電位であるが，1 は普通，正の電源電圧 V_h とする。このような 2 状態しかない信号を処理するには，何らかの**スイッチ素子 (switching device)** を用いるのが便利である。というのは，スイッチ素子も ON と OFF の 2 状態しかないからである。

スイッチ素子の代表例は**リレー (relay)** であろう。これは**図** 2.1 に示すように，コイルとその作る磁場により引かれる鉄片により構成されている。コイルに電流を流すことにより，この可動鉄片が引かれ，それにより，電流路が ON になったり OFF になったりする。

接点の取り付け位置により，**図** 2.2(a) に示すように，コイルに電流が流れていないときに OFF で，電流が流れると ON になるリレーも作る

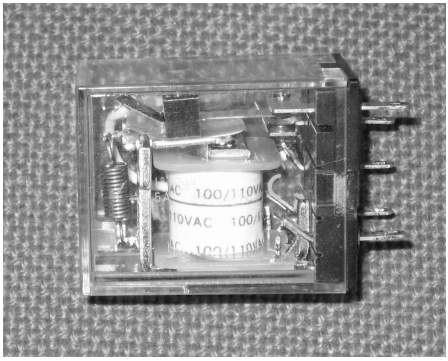


図 2.1　リレー

ことができるし，逆に (b) のように，電流の流れていないときに ON で，電流が流れると OFF になるリレーも作ることができる。さらに，一つのリレーに複数の接点を付けることもできるなど，多機能であることから，比較的容易に複雑な論理回路を作ることができる。動作がわかりやすく，堅牢でもあるが，応答時間が数 ms から数十 ms ほどと遅いという決定的な問題を有する。

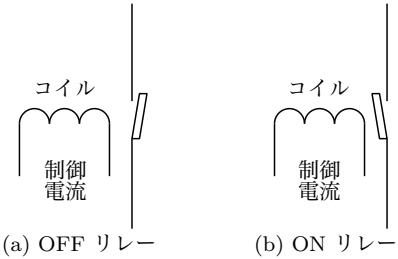


図 2.2　OFF リレーと ON リレー

2.2 電子素子と MOS FET

コンピュータの発展につれ，速度の遅いリレーに代わって，速度の速い電子スイッチである**電子素子 (electronic device)** が利用されるようになった。まず，

図 2.3 に示す**電子管 (electronic tube)** が使われた。[1] これは真空中を流れる電子の流れを電子の嫌いな負電圧を利用して制御するもので，電子の質量が極めて軽いため，応答速度も 1μs 以下と極めて速いものであった。しかし，電子を真空中に出すために加熱が必要であり，そのヒーターや真空を維持する必要から，数千時間ほどの短い寿命であることが大問題であった。また，大きさもここに見られる十数 cm のものから，小さくても数 cm というかなりのサイズを要した。

図 2.3　電子管

1) **真空管 (vacuum tube)** ともいう。

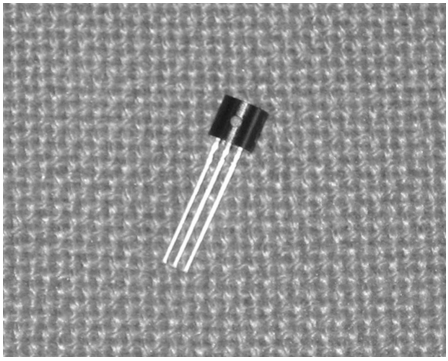
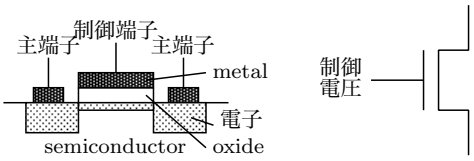


図 2.4　トランジスタ

このため，ヒーターを要しない半導体を用いた**トランジスタ (transistor)** と呼ばれる電子素子が開発された。**図** 2.4 に示したものは 5mm ほどのサイズであるが，ケースを除外した本体は 1mm を切る。半導体はもともと負電荷である電子や正電荷である正孔 [2)] が存在している材料であり，ヒーターを要しない。また，全体が固体であって真空も必要としないことから，半永久と言われるほどの長寿命であり，さらに，応答速度も 1ns 以下にできることから現在のコンピュータの開発の最大の要因となった。

トランジスタは 3 端子素子であり，主端子間を流れる主電流を第 3 の端子にかける電位により制御することができる。かつてはトランジスタと言えばバイポーラトランジスタと呼ばれるものを指したが，現在は **MOS 電界効果トランジスタ (MOS field-effect transistor)**，略して **MOS FET** のことを指す。また，**電界効果トランジスタ (field-effect**

2) 半導体中では，電子のとるエネルギーによって，結晶格子の影響の結果，つまり正電荷のような動作をするものがある。これらを正孔という。ここでは単なる正の電荷と理解して構わない。



(a) 断面図　　(b) 回路記号

図 2.5　n-MOS FET の断面図と回路記号

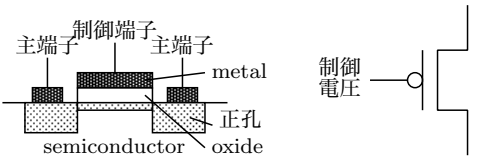
transistor) を，単に **FET** と呼ぼう。

MOS とは現在一番使われている FET の構造であり，**metal-oxide-semiconductor** を略したものである。**図** 2.5(a) に示す構造の断面図のように，主電流の流れる半導体の上に，キャパシタ [3] を構成する絶縁物である酸化シリコンが置かれ，その上に金属の制御端子が置かれていることを，上から順に読んだものである。この制御端子にかける電位により，電流路を流れやすくしたり，流れにくくすることにより制御を行うので，制御端子にはほとんど電流が流れ込まないことから，僅かな電力で主端子間の大きな電力を制御できる。

主電流を担っているのが負電荷である電子である場合，これを **n-MOS FET** という。以後，**n-MOS** と略そう。n は negative charge に由来している。回路記号は MOS 構造に似せた**図** 2.5(b) に示すようなものであるが，回路では図のように縦横を入れ替えて描くことが多い。

制御端子に正の電圧をかけると負電荷である電子は制御端子下に居やすくなり，その結果，大きな電流が流れる。制御端子に負の電圧をかけると負電荷である電子は制御端子下に居づらくなり，その結果，電流は流れにくくなり，さらに大きな負電圧をかけると，電子がまったくなく

3) キャパシタとは，絶縁物を二つの導体で挟んだ構造で，導体間に電位差を与えることにより電荷を溜めることができる。



(a) 断面図　　(b) 回路記号

図 2.6　p-MOS FET の断面図と回路記号

なり，電流は流れなくなる。

これらの場合，主回路の上下の端子はまったく対等であり，制御端子は，二つの端子の電位の平均値を基準にした電位で作用するが，多くの場合，二つの端子の電位の低い方を基準にした電位で制御できると考えて差し支えない。この電流の流れ出す制御端子電位はしきい値電圧と呼ばれ，V_{th} で表される。V_{th} は半導体の製造行程であらかじめ調整することができ，論理回路で使われる FET では $0.2V_h$ 程度に調整される。

リレーともっとも異なるのは，制御が電流によるのではなく電圧によること，OFF 時はほぼ完全なる開放であるが，ON 時は無視できない抵抗が残ることである。しかし，寿命が半永久的と圧倒的に長いこと，極めて高速で動作することに加え，消費電力が極めて小さいこともあり，現在の論理回路はすべて半導体で作られている。

逆に主電流を担っているのが正電荷である正孔である場合，これを **p-MOS FET** という。以後，**p-MOS** と略そう。p は positive charge に由来している。**図** 2.6 に構造の断面図と回路記号を示す。

制御端子に正の電圧をかけると正電荷である正孔は制御端子下に居づらくなり，その結果，電流は流れにくくなり，さらに大きな正電圧をかけると，正孔が完全になくなり，電流は流れなくなる。制御端子に負の電圧をかけると正電荷である正孔は制御端子下に居やすくなり，その結

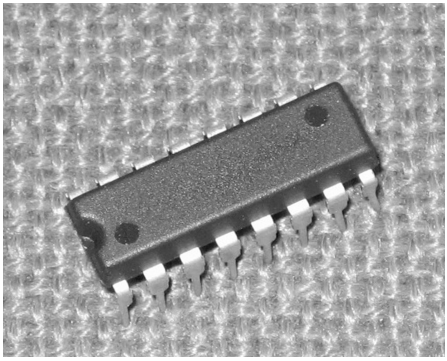


図 2.7　集積回路

果，大きな電流が流れる。構造は n-MOS とまったく同じであり，電子が正孔になっただけであるが，制御端子の電位の作用が反対であるため，制御端子側に否定の意味の○を付けてある。

n-MOS と p-MOS という反対の機能を持つ 2 種類の素子が得られたことで，リレーのような設計の多様性が得られ，さらにいずれ詳細を説明するが，極めて低消費電力の回路も実現できるようになったため，現在，FET はコンピュータの世界でもっとも利用される素子となったのである。

2.3　集積回路

チップと呼ばれる 1cm ほどの半導体の上に，半導体を材料とする**トランジスタ (transistor)** やその他の部品も含め多量に搭載したものを**集積回路 (integrated circuit)** または **IC** という。もともとは，アナログ回路，特に直流増幅器という回路を作成する際，特性の揃ったトランジスタを 2 個用意する必要があり，そのため同じ半導体材料に同時に作成するのがよいだろうということで発明された技術である。

集積回路には，それ以外にも，素子ごとの容器が不要な点，素子ごと

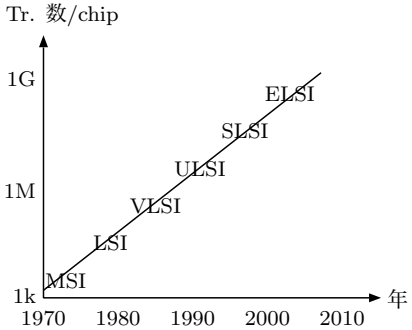


図 2.8　集積回路の歴史

の接続のためのピン，コネクタなどの部品が不要なことから，かなり複雑な回路を極めて小型に作ることができるといった特長がある。同じような理由から個別部品を集めたものよりはるかに安価である。さらに外部接続数が少ない分だけ信頼性が高くなる。トランジスタも小さくでき，配線も細く短くできるので，小電力，高速となるといった多くの特長があるため，急速に発展した。**図** 2.7 は 2cm ほどのケースに入っている本体 5mm 平方ほどのチップの集積回路である。かなり低集積のものであるが，それでも 100 個ほどのトランジスタが入っている。

現在，集積回路はアナログ回路であろうとディジタル回路であろうと，あらゆる電子回路で使われている。特にディジタル回路では，アナログ回路と比較し，入出力の本数が数倍は多く，極めて膨大な数のトランジスタを必要とする。このため，集積回路の必要性はますます高くなり，微細化の技術とともにそれがさらに高集積化に結び付き，最終的に，**マイクロプロセッサ (micro-processor)** や**メモリー (memory)** が一つの半導体のチップの上に作成可能となってきたのである。

集積回路の規模は，現在もディジタル回路を中心に年々大きくな

りつつあり，**図** 2.8 に概略を示したように素子数が 1k(千) 程度以下の **SSI(small scale integration)**，10k 程度以下の **MSI(medium scale integration)**，100k 程度以下の **LSI(large scale integration)**，1M(100 万) 程度以下の **VLSI(very large scale integration)**，10M 程度以下の **ULSI(ultra large scale integration)**，100M 程度以下の **SLSI(super large scale integration)** と発展し，現在は 1G(10 億) 程度の **ELSI(extra large scale integration)** まで開発され，コンピュータのプロセッサが複数，数 cm^2 のシリコンチップに載るようにまでなってきている。

この集積度がほぼ年数の指数関数的に増加していく経験則を**ムーアの法則 (Moore's law)** と呼び，1.6 倍/年あるいは 4 倍/3 年あるいは 10 倍/5 年のペースで増加していく。このように，特にディジタル回路の発展には，集積回路の進展が深く関与しており，これなくして，現在のディジタル全盛はありえなかったと言えよう。

演習問題 2

問題 2.1 次の用語を理解したかどうか確認せよ。

1) MOS FET
2) n-MOS
3) p-MOS
4) IC
5) ムーアの法則

3 基本論理回路

《目標＆ポイント》論理回路とは 0，1 で与えられる (一般には複数の) 入力を処理して，対応する 0 または 1 の出力を出す回路である。その基礎となるのは NOT，AND，OR であるが，これらがスイッチ素子をどう組み合わせて構成されるのかについて説明する。

《キーワード》論理回路，否定，NOT，論理積，AND，論理和，OR，NAND，NOR

3.1 論理否定 NOT

論理回路 (logic circuit) とは，0 または 1 の 2 値をとる入力 (一般には複数) に対し，2 値の出力を出す回路である。通常，2 進表現の 0，1 を回路内のもっとも低い電位である 0[V]，およびもっとも高い電位である V_h に対応させる。高い電位とは言っても 1 から数 V の程度である。出力電位を上げたり下げたりするには，スイッチにより出力を V_h に接続したり，0 に接続したりするのがもっとも簡単である。ここに，前章で述べたようなスイッチ素子を利用するのである。

0 と 1 からなるディジタル量を処理する基本論理回路として，しばしば NOT，AND，OR 回路があげられる。というのは，後に示すように，これら三つの回路があれば，いかなる論理回路もこれらの合成で作成できることがわかっているからである。

まず NOT であるが，**論理否定 (logical negation)** とも言い，偽に対

$$Out = \overline{In} = \mathrm{NOT}(In)$$

In	Out
0	1
1	0

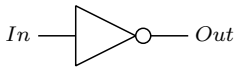


図 3.1　NOT の真理値表と回路記号

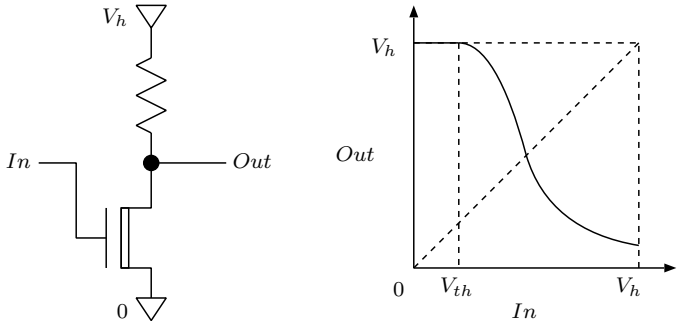


図 3.2　n-MOS NOT 回路とその入出力特性 (n-MOS の主回路を白く表現することにより，$In = 0$ のとき OFF であることを示した)

して真，真に対して偽となる論理である。論理を反転するので，回路的には**インバータ (inverter)** ともいう。2 進表現では，偽を 0 に，真を 1 に対応させる。つまり「'not A' is 1 only when A is not 1.」である。これを**図** 3.1 に示す**真理値表 (truth table)** と呼ばれる表で表現する。式で表すときには，$\overline{In}$ のように全体の上にバーをつけるか，関数形式で $\mathrm{NOT}(In)$ のように表す。また，同図に回路記号も示したが，三角形は増幅器を示し，○は否定を示す。

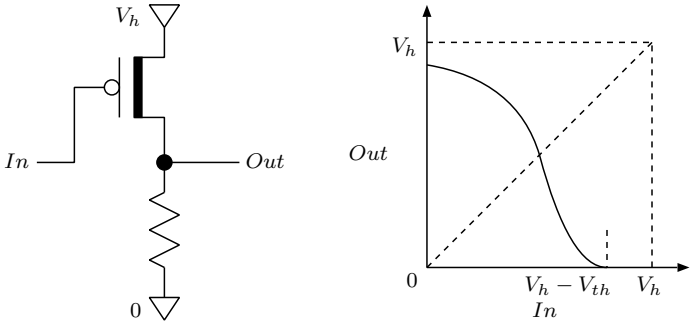


図 3.3　p-MOS NOT 回路 (p-MOS の主回路を黒く表現することにより，$In = 0$ のとき ON であることを示した)

これを n-MOS で実現するには，**図** 3.2 に示すように，接地された n-MOS と抵抗で作成することができる。同右図に見られるように，$In = 0$ のときには，n-MOS が開放，つまり OFF となり，Out は抵抗により V_h に引き上げられ，$Out = V_h$ となる。In を徐々に上げていくと，In が V_{th} のときから，n-MOS は導電性を持つようになってくる。その結果，抵抗との引き合いで決定される出力電位は徐々に下がっていく。$In = V_h$ となっても，FET には抵抗が残るため，Out は 0 近くなることはあっても完全に 0 にはならない。

こうした抵抗と n-MOS で作られた回路を **n-MOS 回路 (n-MOS circuit)** という。したがって，これは n-MOS NOT 回路である。この回路の問題点は，$In = V_h$ のときでも，抵抗と FET を経由して電流が流れ続け，抵抗や FET による電力損失が存在することである。また，集積回路にする際，抵抗が面積をとり，集積度が高くとれないことである。

同様に**図** 3.3 に示すように，V_h に接続された p-MOS と，0 に接続された抵抗で NOT を作成することもできる。$In = V_h$ のときには p-MOS は完全に OFF となるので，Out は抵抗により引き下げられ完全に $Out = 0$

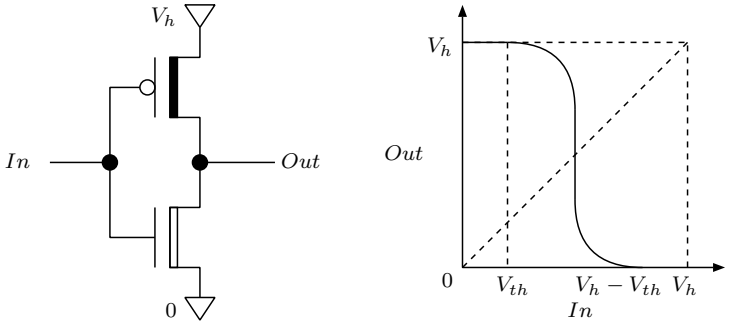


図 3.4　c-MOS NOT 回路 ($In = 0$ のとき ON となる p-MOS 主回路を黒く，OFF となる n-MOS 主回路を白く表現した)

となる。逆に $In = 0$ のときには，p-MOS は ON となるが，完全には抵抗 0 とはならないので，Out は引き上げられるが，完全には V_h とはならない。こうした p-MOS と抵抗で作られた回路を **p-MOS 回路 (p-MOS circuit)** という。この回路も $In = 0$ のとき抵抗による電力損失がある。また，集積度も低い。

抵抗を排除し，MOS FET だけで構成した回路も存在する。その回路は**図** 3.4 に示すように，n-MOS 否定回路の抵抗を p-MOS で置き換えたもの，あるいは p-MOS 否定回路の抵抗を n-MOS で置き換えたものである。性格の反対なスイッチング素子を V_h と 0 の間に縦に入れ，その中間点から出力を取り出したものである。

ここに示した回路では，$In = 0$ のときには，上の p-MOS は ON で抵抗が残るものの，下の n-MOS は完全に OFF となるため，Out は p-MOS により引き上げられ，$Out = V_h$ となる。逆に $In = V_h$ のときには，下の n-MOS は ON で抵抗は残るものの，上の p-MOS は完全に OFF となり，Out は n-MOS により引き下げられ，$Out = 0$ となる。

このように，n-MOS と p-MOS を巧みに組み合わせた FET 回路を **c-**

MOS 回路 (c-MOS circuit) という。c は complimentary の略で，相補的，つまり，互いに相手を補完するという意味である。以下，**c-MOS** と略そう。c-MOS 回路はいつもどちらかの FET が OFF となっているため，電力を使うのは，出力が切り替わるときだけであり，極めて低消費電力である。また，大面積を必要とする抵抗もないため，高い集積度が確保できる。このため，現在の論理回路はほとんど c-MOS 回路になっている。

3.2　論理積 AND と論理和 OR

論理回路で重要なものに，論理否定 NOT に加えて，論理積 AND と論理和 OR がある。詳細はいずれ説明するが，これら 3 種類の基本論理回路があると，どんな入出力関係を持つ論理回路も設計可能になるからである。

まず AND であるが，**論理積 (logical multiplication)** とも言われ，すべての入力が真のときのみ，真となる論理である。つまり「'A and B' is 1 only when $A = 1$ and $B = 1$.」である。例えば 2 入力のときの真理値表は**図 3.5** のようになる。式で表すときにはしばしば積の形で表現され，「$\cdot$」で結合するか，代数のようにそのまま変数を結合する。実際，普通の積でも，一つでも 0 があると，積は 0 となり，すべてが 1 のときのみ，積は 1 となる。

OR とは**論理和 (logical addition)** とも言われ，複数入力のうち一つでも 1 があると，出力が 1 となる回路である。つまり「'A or B' is 1 only when $A = 1$ or $B = 1$.」[1] である。入力 2 個の OR の真理値表は**図 3.6** のようになる。式で表すときにはしばしば和の形で表現され，「$+$」で結

1) 英語で 'A or B' というと，A と B のいずれか片方が 1 の場合のみで，両方とも 1 の場合を除外するかの語感もあるが，論理学ではこれも含めるので注意。

$$Out = In_1 \cdot In_2 = \mathrm{AND}(In_1, In_2)$$

In_1	In_2	Out
0	0	0
0	1	0
1	0	0
1	1	1

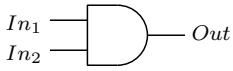


図 3.5　AND の真理値表と回路記号

$$Out = In_1 + In_2 = \mathrm{OR}(In_1, In_2)$$

In_1	In_2	Out
0	0	0
0	1	1
1	0	1
1	1	1

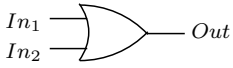


図 3.6　OR の真理値表と回路記号

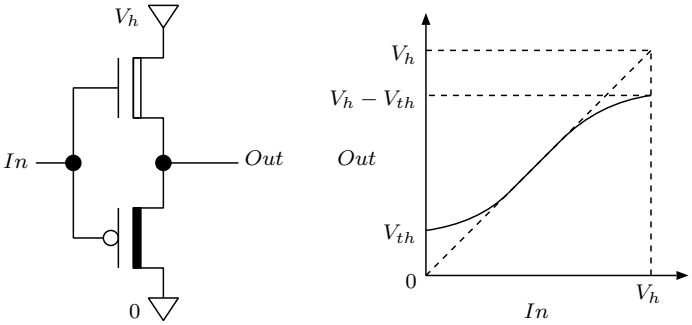


図 3.7　緩衝増幅器

合する。実際，普通の和でも，すべてが 0 のときのみ，和は 0 となり，一つでも 1 があると，和は 0 とはならない。普通の和と異なるのは，一つでも 1 があれば，いくつ 1 があっても結果を 1 にしてしまうことである。

NOT と AND と OR は重要な基本論理回路であるが，残念なことに，FET のような電子素子を使うと，AND と OR は簡単には実現できない。その理由を述べよう。

先に述べた n-MOS NOT，p-MOS NOT，c-MOS NOT の三つの NOT 回路の上下の素子，つまり抵抗と FET，あるいは二つの FET を入れ替えると，どのような動作をするのであろうか。**図** 3.7 に示したものは，c-MOS NOT の上下を入れ替えたものである。$In = 0$ であると，下の p-MOS が ON，上の n-MOS が OFF となるから $Out = 0$，また $In = V_h$ であると，下の p-MOS が OFF，上の n-MOS が ON となるから，$Out = V_h$ になると思われがちであるが，そう簡単ではない。というのは，スイッチを制御する電位差は，FET 両端の電位の低いほうを基準にするからである。

$In = 0$ であると，n-MOS は OFF，p-MOS は ON なので，Out は下が

る。しかし，Out は完全には 0 とはならないのである。p-MOS が ON となるには，制御電極の電位が，p-MOS の上下端子の高いほうの電位，つまり Out の電位より V_{th} だけ低くなくてはならない。したがって，$In = 0$ の場合，Out は $V_{th}(\approx 0.2V_h)$ 程度までしか下がらないのである。[2)] 同様に $In = V_h$ のときにも，Out が $V_h - V_{th} \approx 0.8V_h$ ぐらいになると，低い Out 電位を基準にした n-MOS の制御電位が V_{th} 程度になり，それ以上，ON にはできなくなる。

このため入力電位を 0 から V_h まで変えても，Out の電位は V_{th} から $V_h - V_{th}$ までしか変化しない。入力の変動幅よりも出力の変動幅のほうが小さい，つまり利得 1 以下の増幅器にしかならないので，論理回路としては使われない。しかし，これらの回路は，負荷に何を繋いでも，あまり影響を受けないため，**緩衝増幅器 (buffer amplifier)** と呼ばれる。

このように，n-MOS を上に，p-MOS を下にした回路は，電圧的には増幅作用がないため，論理回路には使いづらい。NOT のように n-MOS を下に，p-MOS を上にした回路は入力信号は完全に反転する。したがって FET を用いると，NOT 的な回路は作りやすいが，反転のない AND とか OR のような回路は，作りづらいことがわかるであろう。反転のない AND や OR 回路は，以下に述べる NAND や NOR 回路などを組み合わせて実現する。これらは NOT(AND) および NOT(OR) であり，ちょっとした工夫で簡単に AND や OR にできるからである。

3.3 NAND 回路と NOR 回路

FET を使った場合には，NAND 回路や NOR 回路しかできない。まず NAND から見てみよう。**図 3.8** に示すように，NAND 回路とは

2) 第 2.2 節参照

$$Out = \mathrm{NAND}(In_1, In_2)$$

In_1	In_2	Out
0	0	1
0	1	1
1	0	1
1	1	0

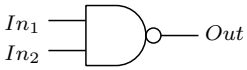


図 3.8　NAND の真理値表と回路記号

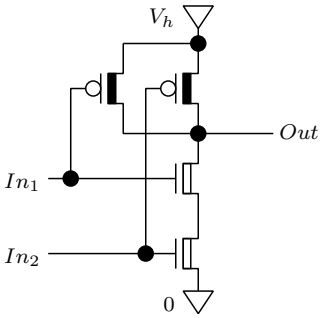


図 3.9　c-MOS NAND 回路

NOT(AND) つまり，出力が AND 回路の出力を反転したものになる論理回路である。回路記号の小丸も，AND の結果を否定することを示している。

この回路は NOT の回路から容易に想像できる。まず NAND の FET による 2 入力回路を**図** 3.9 に示す。これらの回路では両入力が 1 のときのみ，下の回路が全体として ON になり，上の回路が全体として OFF に

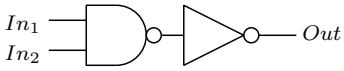


図 3.10　AND は NOT(NAND)

$$Out = \mathrm{NOR}(In_1, In_2)$$

In_1	In_2	Out
0	0	1
0	1	0
1	0	0
1	1	0

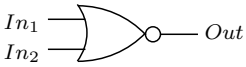


図 3.11　NOR の真理値表と回路記号

なって，出力は 0 となる。それ以外のときには，下の回路が全体として OFF，上の回路が全体として ON になって，出力は 1 となる。

FET を使って AND を実現するには，**図** 3.10 に示すように，AND = NOT(NAND) の形で構成する。

NOR 回路とは**図** 3.11 に示すように，NOT(OR) つまり，出力が OR 回路の出力を反転したものになる論理回路である。

2 入力の NOR 回路は NAND 回路と同様に，**図** 3.12 のようにして構成できる。

これで，種々の論理回路を自由に設計するための基本回路である NOT，NAND，NOR が揃ったことになる。なお，論理式は，NOT をバーなる単項演算子で，また AND，OR を・，+なる 2 項演算子を使って書くこ

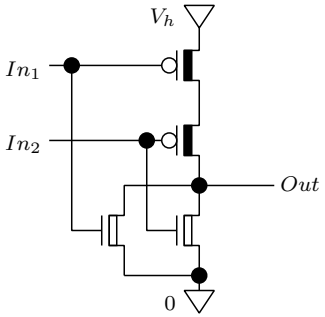


図 3.12　c-MOS NOR 回路

とが多いが，本書のように NAND，NOR が多くなってくると，バーが多くなって見づらくなる。そこで，本書では NOT( )，AND( )，OR( )，NAND( )，NOR( ) といった関数形も併用する。

演習問題 3

問題 3.1　次の用語を理解したかどうか確認せよ。

1）n-MOS NOT 回路
2）c-MOS NOT 回路
3）真理値表
4）緩衝増幅器
5）c-MOS NAND 回路

問題 3.2　c-MOS の 3 入力 NAND 回路を，2 入力回路を拡張して構成してみよ。

問題 3.3　2 入力 OR を NOR を使って構成してみよ。必要に応じ NOT

を使用してよい。

問題 3.4　c-MOS の 3 入力 NOR 回路を，2 入力 NOR の延長として構成してみよ。

4 組合せ論理回路

《目標＆ポイント》FET の比較的簡単な組合せで構成することができる論理否定 NOT，論理積 AND，論理和 OR といった基本論理回路 (実際は NOT，NAND，NOR) を組み合わせると，もっと複雑な回路を作ることができる。こうした基本論理回路を組み合わせて作られる組合せ論理回路について説明する。

《キーワード》組合せ論理回路，EOR，半加算器，全加算器，マルチプレクサ，デマルチプレクサ，デコーダ，エンコーダ，AND-OR 回路，NAND-NAND 回路，NOR-NOR 回路

4.1 多入力 NAND，NOR

前章で，2 入力の NAND や NOR について述べた。4 入力ぐらいまでの NAND や NOR は，同様な回路で作成するが，非常に多くの入力を持つ多入力の AND，OR，NAND，NOR 回路は，回路の動作速度が遅くなることから 1 段で構成することは少ない。これらは簡単な多段化回路を用いて実現できる。これらを組み合わせると多入力 NAND や NOR を作ることができる。こうした基本論理回路の組合せにより構成された回路を**組合せ論理回路 (combinational logic circuit)** という。また，単に**論理回路 (logic circuit)** と略すことも多い。

多入力 NAND や NOR を作る例として，2 入力 NAND や NOR しか持っていなかったとし，それらを組み合わせて，3 入力以上の NAND や NOR を作ることができることを示そう。まず，二つの 2 入力 AND を

$$Out = In_1 \oplus In_2$$

In_1	In_2	Out
0	0	0
0	1	1
1	0	1
1	1	0

図 4.1　EOR の真理値表

用意し，その二つの出力を 2 入力 NAND に入れると，NAND(AND) と 2 段構造となる。前段は二つの AND であるが，全体の動作を調べてみると結局 4 入力 NAND になっている。前段の AND を NOT(NAND) に書き換えると，NAND(AND) → NAND(NOT(NAND)) となり，4 入力 NAND が 2 入力 NAND と NOT だけと，計 5 個の基本論理回路で構成できる。なお，さらに NAND(NOT) → OR → NOT(NOR) であるので，全体は NOT(NOR(NAND)) となり二つの NAND の出力の NOR の NOT と，やや簡素化された計 4 個の基本論理回路で構成できる。

4.2　EOR 回路と加算器

2 入力 1 出力の論理回路で，OR とほぼ同じであるが，2 入力とも 1 のときには 0 となる回路を**排他的論理和 (exclusive OR)** または **EOR** という。あるいは，二つの入力が異なるときだけ 1 を出力する回路と言ってもよく，**図** 4.1 に示す真理値表で表すことができる。

これは，片方を制御信号とみなし，もう片方の入力から出力へ出ていく信号を制御する回路と見ることもできる。その場合，制御信号が 0 であると，主回路の信号はそのまま伝わっていくが，制御信号が 1 である

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

図 4.2　半加算器の真理値表

と，主回路の信号は否定されて出力される。

やや天下りであるが，EOR は

$$A \oplus B = \mathrm{OR}(\mathrm{AND}(A, \overline{B}), \mathrm{AND}(\overline{A}, B)) \tag{4.1}$$

の式で実現できる。あるいは

$$A \oplus B = \mathrm{NAND}(\mathrm{NAND}(A, \overline{B}), \mathrm{NAND}(\overline{A}, B)) \tag{4.2}$$

と書くこともできる。これらの式を論理的に導くこともできるが，その詳細は次章を参照して欲しい。

2 入力の**算術和 (arithmetic addition)** を得る回路を**半加算器 (half adder)** という。真理値表で表すと，**図** 4.2 のようになる。A と B の算術和の 0bit 目を S，1bit 目を C と表した。真理値表からわかるように $S = A \oplus B$，$C = A \cdot B$ である。

複数桁の加算を行おうとすると，**キャリー (carry)** と呼ばれる次の桁への繰り上げを考慮しなければならない。これを C_o[1] とする。また，各桁では，下の桁からのキャリー C_i[2] が入ってくるので，$A+B$ ではなく，$A+B+C_i$ の計算を行わなければならない。$A+B+C_i = (A+B)+C_i$

1) この桁から出ていく (out) キャリーなので suffix o を付ける。
2) この桁に入ってくる (in) キャリーなので suffix i を付ける。

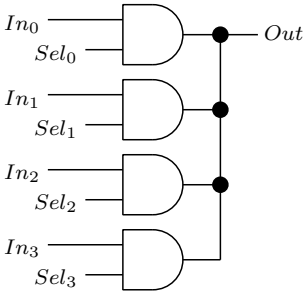


図 4.3　マルチプレクサ

であるので，S を得るには，A，B からなる半加算器の出力の後ろに C_i との半加算器をつければよい。キャリー C_o は，$A+B$ にキャリーがあれば，当然 1 となるし，さらに $A+B$ の 1 桁目と C_i の和に繰り上げがあれば当然 1 となるので，これらの OR をとればよいことになる。これを**全加算器 (full adder)** という。

4.3　マルチプレクサとデマルチプレクサ

2 入力 AND 回路の入力の一つを制御入力，もう一つを主入力とみなすと，制御入力が 1 のときには主入力がそのまま出力に現れるが，制御入力が 0 のときには主入力は出力に伝わらず，出力は 0 となる。この意味で AND は信号伝達のゲートのように動作する。

例えば，入力として n 個の通信線があり，出力は 1 本のとき，入力の 1 本を選択して出力に出すには，**図 4.3** に示すように，この n 個の線をそれぞれ 2 入力 AND の片方の入力 In_i に入れ，残る入力に選択信号 Sel_i を入れると，選択信号で選ばれた線だけが出力に接続されることになる。こうした選択回路は**マルチプレクサ (multiplexer)** と呼ばれている。

逆に 1 本の通信線が入ってきて，その上に乗ってきたデータを，タイ

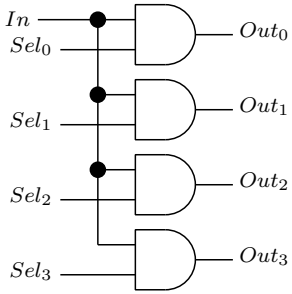


図 4.4　デマルチプレクサ

ミングによって，n 本の線のいずれかに分配しようとすると，**図** 4.4 に示すように，この通信線を n 本に分岐し，それぞれ n 個の 2 入力 AND に入れてやればよい。出力を分配したい AND の残る 1 本の入力，つまり制御入力を 1 にしてやると，その出力だけが，入力の通信線の信号を伝えることになる。この回路は**デマルチプレクサ (demultiplexer)** と呼ばれる。

4. 4　AND-OR 回路

任意の入出力関係を実現する組合せ回路は，知識と経験がないと設計できないと思われたかも知れないが，多少回路規模が大きくてもよいのならば，かなり組織的に設計することができる。本節と次節で，その組織的構成法について紹介する。

論理回路は一般に複数の入力と複数の出力を持ちうる。また，その動作は入力に発生するありとあらゆる可能なビットパターンに対する，出力パターンを示した**真理値表 (truth table)** により完全に記述できる。

どんな真理値表からスタートしてもよいのだが，**図** 4.5 に示す全加算器を例にして，C_o を出力する論理回路を考えてみよう。まず，(000) が

	A	B	C_i	C_o	S
M_0	0	0	0	0	0
M_1	0	0	1	0	1
M_2	0	1	0	0	1
M_3	0	1	1	1	0
M_4	1	0	0	0	1
M_5	1	0	1	1	0
M_6	1	1	0	1	0
M_7	1	1	1	1	1

図 4.5　真理値表の例 (全加算器)

入ってきたときにのみ 1 を出力する論理回路を考えよう。(000) は，厳密には $(0,0,0)$ のことであるが，以下このように略記する。この論理回路の出力を M_0 とすると，

$$M_0 = \mathrm{AND}(\overline{A}, \overline{B}, \overline{C_i}) \tag{4.3}$$

のように，三つの入力の NOT の AND をとったものである。$\overline{A}$ などは A などの否定を表している。(A, B, C_i) が (000) のとき，$(\overline{A}, \overline{B}, \overline{C_i})$ は (111) となる。一方，3 入力 AND は入力が (111) ときのみ 1 を出力するから，(A, B, C_i)=(000) のときのみ M_0=1 となる。同様な考察で，**図 4.5** の各

行に対応した M_0 から M_7 は次のように表される。

$$\begin{aligned} M_0 &= \mathrm{AND}(\overline{A}, \overline{B}, \overline{C_i}) \\ M_1 &= \mathrm{AND}(\overline{A}, \overline{B}, C_i) \\ M_2 &= \mathrm{AND}(\overline{A}, B, \overline{C_i}) \\ M_3 &= \mathrm{AND}(\overline{A}, B, C_i) \\ M_4 &= \mathrm{AND}(A, \overline{B}, \overline{C_i}) \\ M_5 &= \mathrm{AND}(A, \overline{B}, C_i) \\ M_6 &= \mathrm{AND}(A, B, \overline{C_i}) \\ M_7 &= \mathrm{AND}(A, B, C_i) \end{aligned} \tag{4.4}$$

さて，C_o は $M_3,\ M_5,\ M_6,\ M_7$ のいずれかが 1 のときのみ 1 であるから，これら四つのパターンの OR で与えられる。つまり，

$$C_o = \mathrm{OR}(M_3, M_5, M_6, M_7) \tag{4.5}$$

まったく同様に

$$S = \mathrm{OR}(M_1, M_2, M_4, M_7) \tag{4.6}$$

となる。

これら C_o，S を表す式へ，式 (4.4) の $M_0, M_1, \cdots$ を代入すれば，複数の入力の NOT，AND，OR を組み合わせればよいことが理解できよう。

これまでに述べた作業より，ありとあらゆる論理回路は，NOT と AND と OR を使うことにより実現できることが理解できよう。また，NOT，AND，OR の配置は，真理値表の 0，1 の配置に合わせて組織的に行えばよいことも理解できよう。こうした任意の論理回路の実現法を **AND-OR 回路 (AND-OR circuit)** という。

4.5 NAND-NAND 回路

電子回路の場合，こうした論理は AND，OR の代わりに NAND や NOR を使って実現する必要がある。まず，**ド・モルガンの法則 (de Morgan law)** について説明しよう。これは，次の二つの法則をまとめたものである。

$$\begin{aligned} \text{OR} &= \text{NAND(NOT)} \\ \text{AND} &= \text{NOR(NOT)} \end{aligned} \tag{4.7}$$

これらの法則は，OR は，入力のいずれかが 1 であると，1 となる。つまり，「OR はすべての入力が 0 のときだけ例外的に 0 を出力する」こと，一方「AND はすべての入力が 1 のときだけ例外的に 1 を出力する」という例外処理に着目すると簡単に理解できる。

まず前者であるが，すでに述べたように左辺の OR はすべての入力が 0 のときだけ 0 となる論理である。一方，右辺の AND はすべてが 1 のときだけ 1 となるから，すべての入力を反転して AND に与える，つまり AND(NOT) とすることで，すべての入力が 0 のときだけを例外処理とする。この出力を反転させれば，すべての入力が 0 のときだけ 0 となり，両辺は一致する。後者については，各自で解いて欲しい。

例えば前節に示した C_o は，四つの要素 M_i の OR からなっているが，その OR をド・モルガンの法則を使って，NAND(NOT) に差し替える。次に各 M_i の否定が必要となるが，各 M_i は AND により構成されているので，NAND となる。

$$
\begin{aligned}
\overline{M_0} &= \mathrm{NAND}(\overline{A}, \overline{B}, \overline{C_i}) \\
\overline{M_1} &= \mathrm{NAND}(\overline{A}, \overline{B}, C_i) \\
\overline{M_2} &= \mathrm{NAND}(\overline{A}, B, \overline{C_i}) \\
\overline{M_3} &= \mathrm{NAND}(\overline{A}, B, C_i) \\
\overline{M_4} &= \mathrm{NAND}(A, \overline{B}, \overline{C_i}) \\
\overline{M_5} &= \mathrm{NAND}(A, \overline{B}, C_i) \\
\overline{M_6} &= \mathrm{NAND}(A, B, \overline{C_i}) \\
\overline{M_7} &= \mathrm{NAND}(A, B, C_i)
\end{aligned} \tag{4.8}
$$

また OR → NAND(NOT) と置き換えると，

$$
\begin{aligned}
C_o &= \mathrm{NAND}(\overline{M_3}, \overline{M_5}, \overline{M_6}, \overline{M_7}) \\
S &= \mathrm{NAND}(\overline{M_1}, \overline{M_2}, \overline{M_4}, \overline{M_7})
\end{aligned} \tag{4.9}
$$

と変形できるのである。これらの式から $\overline{M_0}, \overline{M_1}, \cdots$ を消去すれば，いかなる出力も NOT，NAND (AND に対応)，NAND (OR に対応) を組み合わせればよいことが理解できよう。また，これら基本ゲートの配置は，真理値表の 0，1 の配置に合わせて組織的に行えばよいことも理解できよう。

つまり，NOT と NAND だけがあれば，いかなる論理も構成できてしまうのである。NOT は 1 入力 NAND とみなすこともできるので，「どんな論理も NAND だけで構成できる」と表現することが多い。これを **NAND-NAND 回路 (NAND-NAND circuit)** という。

これらを回路図にすると**図 4.6** のようになる。ただし，いくつの入力を持つ NAND でも手に入るという前提で書いた。なお，この図の一番上

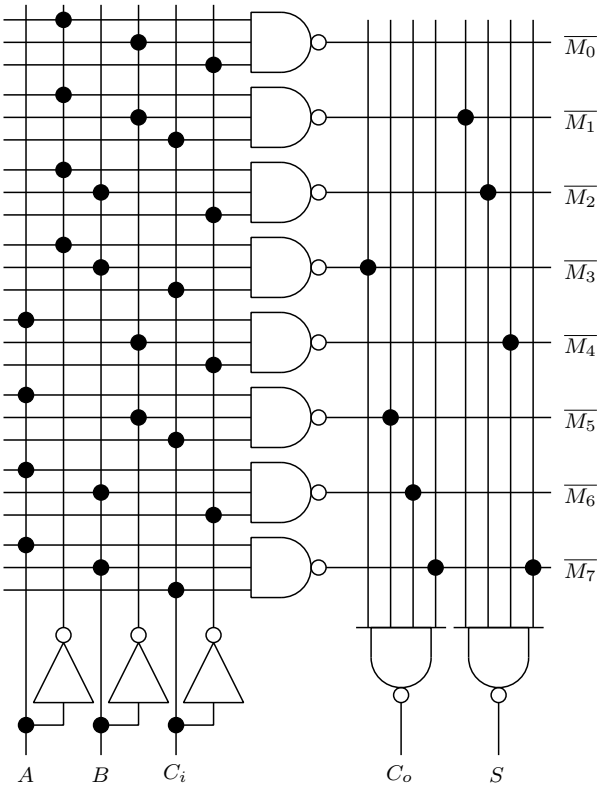


図 4.6　NAND-NAND 回路

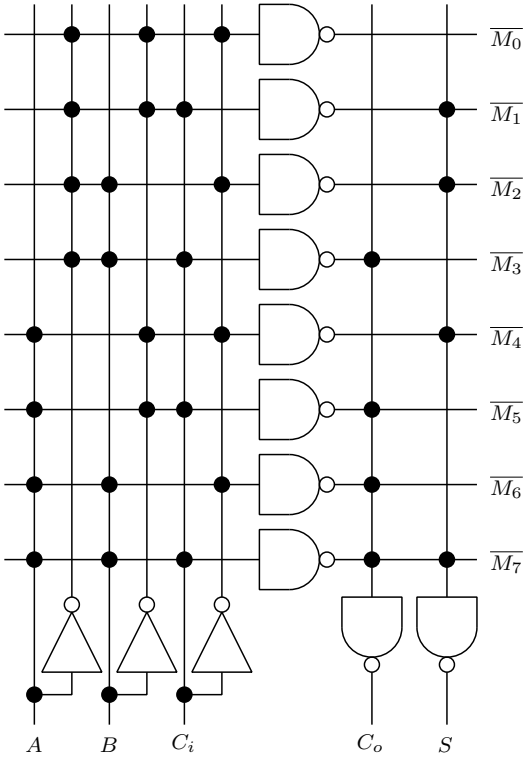


図 4.7　簡略表記による NAND-NAND 回路 (NAND の入力は 1 本にまとめて簡略記載されているが，実際にこのように配線するとショートしてしまうので注意)

の NAND ゲートは，最終出力にいっさい関係しておらず，不要である。しかし，現在の集積回路の内部では見やすい構造とすることを旨としているので，不要な回路でも真理値表との対応関係から，残しておくことが多い。

これを，簡略化して**図** 4.7 のように表すことがある。多入力 NAND の入力を 1 本の線にまとめて表すこの表記法は，実際の回路と対応がとれ

ず，誤解を招きやすいので，安易な導入は危険であるが，見やすいのでしばしば利用されている。なお，実際にいくつかの論理の出力を 1 本の線に接続すると，いわゆる短絡状態を起こす可能性があり，場合によっては論理回路の壊滅的破壊を招きかねないので，絶対に行ってはならない。この図を，**図** 4.5 の真理値表と比較してみると，極めて強い対応がとれていることが理解できよう。左半平面の NAND 面では，真理値表の入力側の 1 に対して，A, B, C_i との接続が対応し，真理値表の 0 に対して，$\overline{A}, \overline{B}, \overline{C_i}$ との接続が対応している。また右半平面の NAND 面では，真理値表の出力側の 1 が対応している。

この左半分の NAND 回路は，n 桁の 2 進表現された数を解析して，その数の内容に対応した 2^n 本の線のいずれかを選ぶ回路である。2 進表現コードを展開することから**デコーダ (decoder)** と呼ばれる。ここに示した NAND 回路は，選ばれた線だけが 0 となる回路である。デコーダとは厳密には 1 本の線にだけ 1 を出す回路であるが，その場合にはすべての線に NOT を付加すればよい。

逆に 2^n 本の線のいずれかに 1 を入れると，その線の番号に対応する数の 2 進表現を出力する回路を，**エンコーダ (encoder)** という。NAND-NAND 回路の右側の設計方針にしたがって，3bit のエンコーダの回路を構成すると，**図** 4.8 のようになる。

なお，デコーダと分配器，エンコーダと選択器の組合せもよく用いられる。

4.6 NOR-NOR 回路

任意の論理回路は NAND-NAND 回路で実現できたが，NOR-NOR 回路でも実現できる。ただし，現在は主として NAND(NAND) が使われて

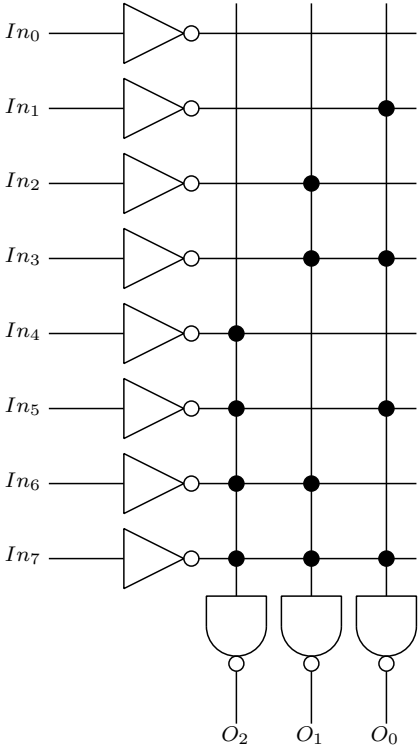


図 4.8　簡略表記によるエンコーダ回路

いるため，本節は読み飛ばしてもらっても差し支えない。

ド・モルガンの法則を使うと，AND-OR 回路の AND は NOR(NOT) に，OR は NOT(NOR) に書き換えることができる。こうすると，式 4.4 の M_0 は AND で構成されるので，次のように書かれる。

$$M_0 = \mathrm{NOR}(\mathrm{NOT}(\overline{A}), \mathrm{NOT}(\overline{B}), \mathrm{NOT}(\overline{C_i})) = \mathrm{NOR}(A, B, C_i)$$

以下同様に次の各式が得られる。

$$\begin{aligned} M_0 &= \mathrm{NOR}(A, B, C_i) \\ M_1 &= \mathrm{NOR}(A, B, \overline{C_i}) \\ M_2 &= \mathrm{NOR}(A, \overline{B}, C_i) \\ M_3 &= \mathrm{NOR}(A, \overline{B}, \overline{C_i}) \\ M_4 &= \mathrm{NOR}(\overline{A}, B, C) \\ M_5 &= \mathrm{NOR}(\overline{A}, B, \overline{C_i}) \\ M_6 &= \mathrm{NOR}(\overline{A}, \overline{B}, C_i) \\ M_7 &= \mathrm{NOR}(\overline{A}, \overline{B}, \overline{C_i}) \end{aligned} \tag{4.10}$$

また C_0 や S は OR で構成されているため NOT(NOR) で実現できる。

$$\begin{aligned} C_o &= \mathrm{NOT}(\mathrm{NOR}(M_3, M_5, M_6, M_7)) \\ S &= \mathrm{NOT}(\mathrm{NOR}(M_1, M_2, M_4, M_7)) \end{aligned} \tag{4.11}$$

が得られる。このように，すべての式を NOT を含む NOR だけの式にすることができる。これを **NOR-NOR 回路 (NOR-NOR circuit)** という。

以上で論理回路についての説明は終了する。論理回路とは，入力側の 0，1 の空間パターンを出力側の別の空間に変換する回路であると言い替えることができる。コンピュータなどのより高度な情報処理機器は，時間的にも空間的にも変化するパターンを処理することができる。次章では，時間方向に変化するパターンの処理方法について説明しよう。

演習問題 4

問題 4.1　次の各問に答えよ。

1）NAND(NOT) が OR と等しいことを，それぞれの真理値表を書くことにより確認せよ。なお，NAND も OR も 2 入力としてよい。

2）NOR(NOT) が AND と等しいことを，それぞれの真理値表を書くことにより確認せよ。なお，NOR も OR も 2 入力としてよい。

問題 4.2　4 入力 NOR は二つの 2 入力 OR の出力を NOR に入れることにより実現できる。

1）OR が NOT(NOR) であることを利用して，NOT と NOR だけで実現せよ。さらに，NOR(NOT) が NOT(NAND) と等しいことを利用して，簡素化せよ。

2）上に示した二つの式を A と B に 0，1 を順次入れていくことで，確かめよ。

3）二つの式を (論理) 回路図にしてみよ。

問題 4.3　半加算器を (論理) 回路図一つにまとめてみよ。

問題 4.4　全加算器を式でなく (論理) 回路図に書いてみよう。

問題 4.5　NOT を 1 入力 NAND とみなせることについて，c-MOS 回路の立場から考察してみよ。同様に，真理値表の立場からも考察してみよ。

問題 4.6　EOR 回路を，NAND-NAND 回路で構成してみよう。不要な部分を消去すると，前に示したものと同じ回路になることを示せ。

5 シークエンス回路

《目標＆ポイント》前章で述べた論理回路は，入力を入れるとすぐに出力が決定する。しかし，コンピュータに代表される各種の情報処理装置では，出力は今入ってきたばかりの入力も参考にして作られるだろうし，過去に入ってきた入力も参考にして作られる。こうしたシークエンス回路と呼ばれるディジタル回路について学ぼう。

《キーワード》シークエンス回路，順序回路，同期式回路，内部状態，状態遷移，遅延回路，パストランジスタ，クロック，レジスタ，プリチャージ論理回路

5.1 シークエンス回路の標準形

過去に入ってきた入力も参考にして出力を決定するコンピュータに代表される多くの情報処理装置では，回路内にいくつかの時間を遅らせる**遅延 (delay)** 要素が入っているはずである。こうしたディジタル回路は特に**シークエンス回路 (sequential logic circuit)** と呼ばれる。シークエンスとは時系列のことであり，シークエンス回路とは，時系列信号，つまりシークエンスを処理する回路という意味である。シークエンス回路は**順序回路 (sequential logic circuit)** とも呼ばれるが，順序を順番と勘違いされるなど，わかりづらいので，本書ではシークエンス回路の用語で統一する。

回路内には，一般的にはいろいろな遅延要素があってもよいが，ここでは，単位遅延時間 τ の整数倍の遅延要素しか考えないこととする。一

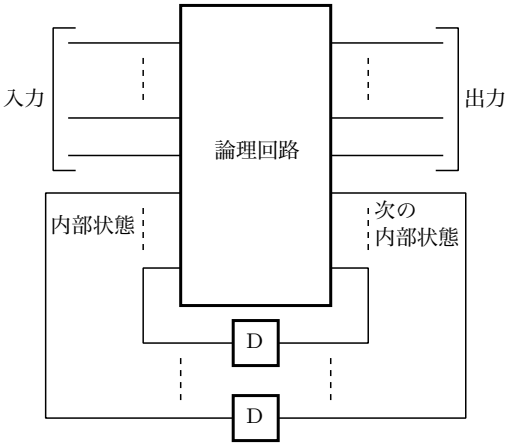


図 5.1　シークエンス回路の標準形

種の時間方向のディジタル化である。信号の高さも，複数の並列信号を使うことにより，2 進化されているとする。シークエンス回路から遅延要素をすべて外へ取り出すと，残りは信号を直ちに処理する部分だけ，つまり，前章で述べたただの論理回路となってしまう。整数倍の遅延は 1 単位遅延の出力をいったん論理回路へ戻し，そのまま，また別の 1 単位遅延の入力とし，それを再び論理回路へ戻すことを繰り返すことにより実現できるから，他の遅延回路はすべて複数の 1 単位遅延であるとしてよい。さらに，遅延を与えるのにすべて同じ周期的パルスを利用して，すべて同期して遅延が与えられる回路を**同期式回路 (synchronous circuit)** と呼ぶ。

つまり**図** 5.1 のようにまとめることができる。遅延回路の出力を，**内部状態 (internal state)** と呼ぼう。すると，シークエンス回路は次のように理解することができる。回路には内部状態があり，現在の内部状態は，1 回前の内部状態と現在の入力で決定される。こうすることで，現

在の内部状態は過去のすべての時系列入力で決定される。現在の出力は，現在の内部状態と入力で決定される。切符の自動販売機に使われるようなディジタル回路も，巨大な電子コンピュータもすべてこうしたシークエンス回路である。

5.2 状態遷移図と状態遷移表

切符の自動販売機，つまり券売機を例に，シークエンス回路を作ってみよう。いくらでも複雑な券売機が設計可能であるが，ここでは簡単な例として，10 円玉 3 枚で切符を 1 枚出す券売機を考える。キャンセルはないものとする。機械からはシークエンス回路への入力として，コイン受けに 10 円玉があるかどうかを検出するセンサ (この出力を In とする) がある。また出力としては，コイン受けからコインを取り込む $Take$，切符を発券する $Ticket$ がある。簡単のために，$Take$ を出力すると，In は直ちに 0 になるものとする。

内部状態としては，機械が 1 円も受け取っていない 0 円状態，10 円を受け取った 10 円状態，20 円受け取った 20 円状態の 3 状態がある。次の 10 円を受け取ると，発券して 0 円状態に戻ればよいから，30 円状態は不要である。これらの内部状態も 2 進表現にする必要があるので，それぞれ 00，01，10 としよう。

0 円状態で $In = 0$ のときには，ずっと 0 円状態にいる。$In = 1$ となると，10 円が入ったので，$Take = 1$ として 10 円を取り込み，同時に自身は 10 円状態となる。同様に 10 円状態でも，$In = 0$ のときにはずっと 10 円状態に居続ける。$In = 1$ で $Take = 1$ を出力して，自身は 20 円状態となる。20 円状態でも，$In = 0$ のときにはずっと 20 円状態である。$In = 1$ で $Take = 1$ を出力するが，同時に $Ticket = 1$ を出力して，自身

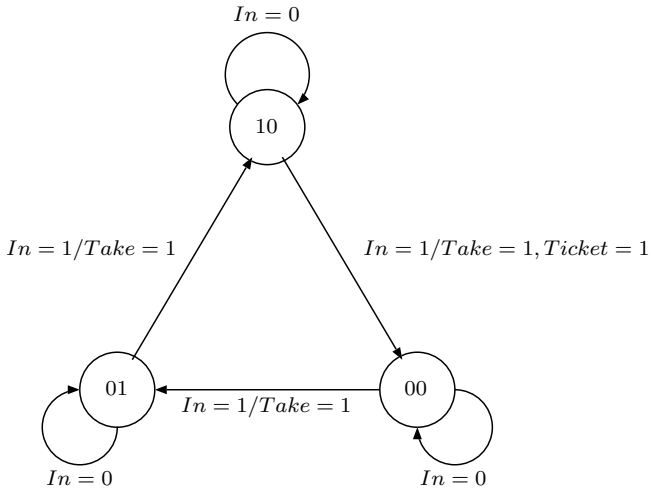


図 5.2　券売機の状態遷移図

は 0 円状態に復帰する。

この流れを，**図** 5.2 に示す**状態遷移図 (transition diagram)** と呼ばれるグラフで示す。状態遷移図では遷移を決める入力とその際の出力を，「入力/出力」の形で記載する。また出力が 0 の場合には，ここにあるように記載を省くことが多い。意味は明らかであろう。

さて，**図** 5.1 の論理回路はどのように設計したらよいのであろうか。それは**図** 5.2 の状態遷移図から容易に求めることができる。例えば，同図の (00) 状態から (01) 状態への矢印グラフを見ると，$In = 1/Take = 1$ と記載されている。したがって，現在の内部状態が (00) で入力の In が 1 の場合，次の状態が (01) で出力は $Take = 1, Ticket = 0$ となる論理回路を作ればよいことになる。

図 5.3 のように，各矢印ごとに，遷移前の状態とそのときの入力を左に，遷移先の状態とすべての出力を右に書いたものを**状態遷移表 (transition**

In	S_1	S_2	S_1'	S_2'	$Take$	$Ticket$
0	0	0	0	0	0	0
1	0	0	0	1	1	0
0	0	1	0	1	0	0
1	0	1	1	0	1	0
0	1	0	1	0	0	0
1	1	0	0	0	1	1

図 5.3　券売機の状態遷移表

table) と呼ぶ。この表は，**図** 5.1 の上の四角の論理回路の入出力関係を表しているから，この表の形の組合せ論理回路を作成し，各 S_i' を，単位遅延を介して S_i に戻せばよい。

5.3　遅延回路

単位遅延はどのように作成したらよいのであろうか。その基本は，入力に入った信号を少し遅らせて出力に出す回路である。これにはキャパシタ [1)] の電圧維持機能を利用すればよい。あるタイミングの入力の値をキャパシタに蓄え，それを次のタイミングで取り出せばよいのである。

具体的な遅延回路を**図** 5.4 に示す。図に見られるように，この回路には二つのスイッチがついている。各スイッチの開閉を制御する信号を ϕ_1，ϕ_2 で示した。各スイッチは，これらの信号が高い電圧のときに ON となり，低いときに OFF となる。まず，左のスイッチを ON にし，入力の電位でキャパシタを充電する。続いて左のスイッチを OFF にするが，キャ

1）キャパシタとは，2 枚の導体の間に薄い絶縁体を挟んだ構造を持ったもので，正電荷と負電荷の引き合う力を利用して，電荷を蓄えることができる。この機能により，両端の電位差を維持する機能を有する。

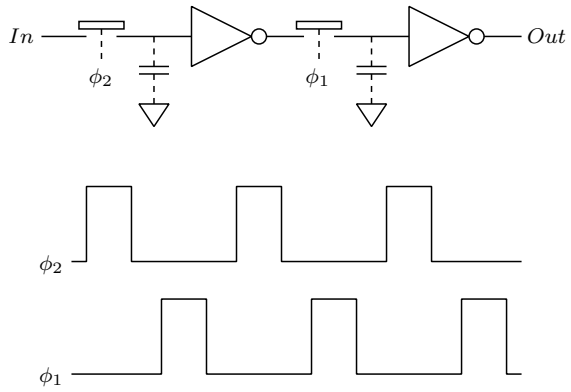


図 5.4　遅延回路 (ϕ_1，ϕ_2 は二つのスイッチを開閉するための信号で，高いときに ON，低いときに OFF となる)

パシタの電圧は，しばらくの間，入力電位を維持する。続いて右のスイッチを ON にすると，信号は出力側に伝達される。そこで，右のスイッチを OFF にする。その信号のレベルは，しばらくの間，右のキャパシタによって維持される。また，この出力は論理回路を経由して直ちに左側の入力に反映される。

インバータ (inverter)(NOT のこと) が二つ入っているが，これは，後段のキャパシタや論理回路を駆動する際，前段のキャパシタの電位が下がらないようにするためのものである。例えば，左のインバータがないと，右のスイッチを ON にした際，電圧は二つのキャパシタで分配されてしまう。仮に両キャパシタ容量が同じであると，電位は半分になってしまう。

一見，スイッチは一つで済みそうであるが，一つのスイッチだと，それを ON にした際，出力が直ちに入力に反映してくるため，入力と出力の分離ができなくなる。例えば，その値が前の値と異なる場合などには，

いずれの入力を伝播しているのか，いい加減になってしまう。特に外部の回路が遅延回路の出力を否定したものを入力に返してくるような場合には，0 と 1 が入れ替わり回路内をぐるぐる回り続ける異常現象が起きてしまう。これは，**ラットレーシング (rat racing)** と呼ばれる。

したがって，2 スイッチの場合にも，二つのスイッチを同時に ON としてはいけない。ちょうどパナマ運河で，上流側の扉と下流側の扉を交互に開けないと，水が無制限にどんどん流れていってしまうのと同じようなものである。

キャパシタは実際には，あらわには付けない。もともと，配線は，接地との間にいくばくかの静電容量を持っているからである。僅かな配線容量では，あっという間に電圧が減衰してしまいそうであるが，遅延回路のスイッチを ON-OFF するコンピュータのクロック (周期性のある ON-OFF 信号) は，この減衰時間に対し十分高速である。

ここで注意して欲しいのは，各部の信号レベルは，ϕ_1 が V_h から 0 になっても，次の ϕ_2 が V_h になるまではそのまま維持されることである。このため，「ϕ_1 の期間」というと，厳密に ϕ_1 が V_h の期間だけを指すこともあれば，次に ϕ_2 が V_h になるまでの長めの期間を指すこともある。同様に，ϕ_2 の期間についても同様である。

スイッチには c-MOS の**パストランジスタ (pass transistor)** を用いる。これは，左右の線を接続したり，切断したりするまさにスイッチの機能を持つ回路である。FET はゲートの電位で ON になったり OFF になったりするので，FET 一つを左右の線の間に入れてみよう。

FET として n-MOS を使ったものを**図 5.5** に示す。スイッチとはどちらかの電位を残りの端子に伝えるものなので，とりあえず，左の線の電位が決まっていて，それを右に伝えることを考えよう。まず左の電位が 0 のときを考えよう。n-MOS のゲート電位を V_{th} 以下にすれば FET は

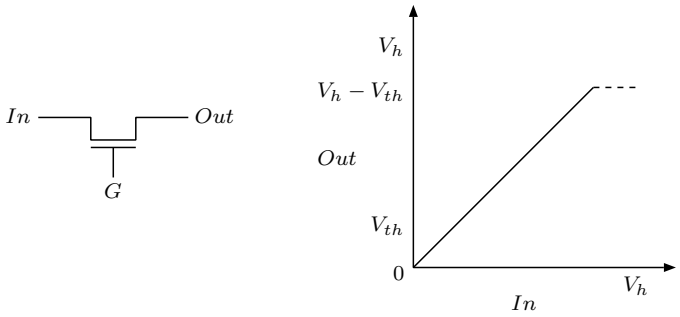


図 5.5　n-MOS パストランジスタ (右図は $G = V_h$ のときの入出力特性で入力が V_h 以下になると出力を V_h 以上に持ち上げる能力が低下する。$G = 0$ のときには完全に OFF)

OFF になり，右の電位はフリーになる。一方，ゲート電位を V_{th} 以上にすれば FET は ON になり，右の電位は左の電位と等しくなりスイッチとして正常に動作している。

次は左の電位が V_h のときであるが，ゲート電位が V_{th} 以下であれば FET は OFF になり，右の電位は同じくフリーになる。問題はゲート電圧が V_h のときである。これは緩衝増幅器のところで述べたのと同じ現象であるが，右の電位が十分低いときには，FET は ON となり，右の電位は V_h 側に引き上げられていく。しかし，右の電位がゲート電位より V_{th} 低いところを越えると，FET の ON 機能は十分働かなくなってしまう。例えばゲート電位が V_h であっても，出力は $V_h - V_{th}(\approx 0.8V_h)$ で落ち着いてしまうのである。まとめると，n-MOS パストランジスタは，ゲートが V_h のとき ON，ゲートが 0 のとき OFF のスイッチとして動作するが，ゲート電位も左も V_h のときのみ，不十分な ON スイッチとなってしまう。

同様に，p-MOS パストランジスタは，ゲートが 0 のとき ON，ゲート

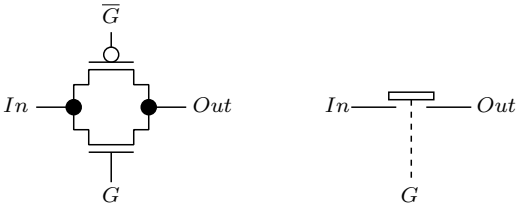


図 5.6　c-MOS パストランジスタと回路記号

が V_h のとき OFF のスイッチとして動作するが，左の電位もゲート電位も 0 のときのみ，不十分な ON スイッチとなってしまう。

そこでもし，0 から V_h までのいかなる電位でもきちんと動作するパストランジスタゲートを得ようとすると，**図** 5.6 のような形とせざるをえない。しかし，スイッチとして n-MOS パストランジスタだけを使っても，最悪，入力が V_h のときでも，約 $0.8V_h$ の出力は確保できる。これだけの電位が確保できれば，次に何らかの論理ゲートを入れれば誤動作は起きないので，スイッチを n-MOS パストランジスタだけで構成することはしばしば行われている。また，ゲート信号が否定論理で与えられているときには，p-MOS パストランジスタだけを使うなど，臨機応変な使い方がなされている。したがって，本書でも，この図の右の回路記号を使っているからといって，その具体的なスイッチの構成を問わないものとする。

本節で示したこうした遅延回路は，歴史的に **D-フリップフロップ (delayed-flip-flop)**，略して **D-FF** とも呼ばれる。この遅延回路に与えるパルス ϕ_1，ϕ_2 を，どの遅延回路でも同じパルス発生器から与えれば，**同期式回路 (synchronous circuit)** となる。また，これら二つのパルス源を**クロック (clock)** と呼ぶ。

図 5.3 に示した状態遷移表を組合せ論理回路にしたものと，**図** 5.4 に

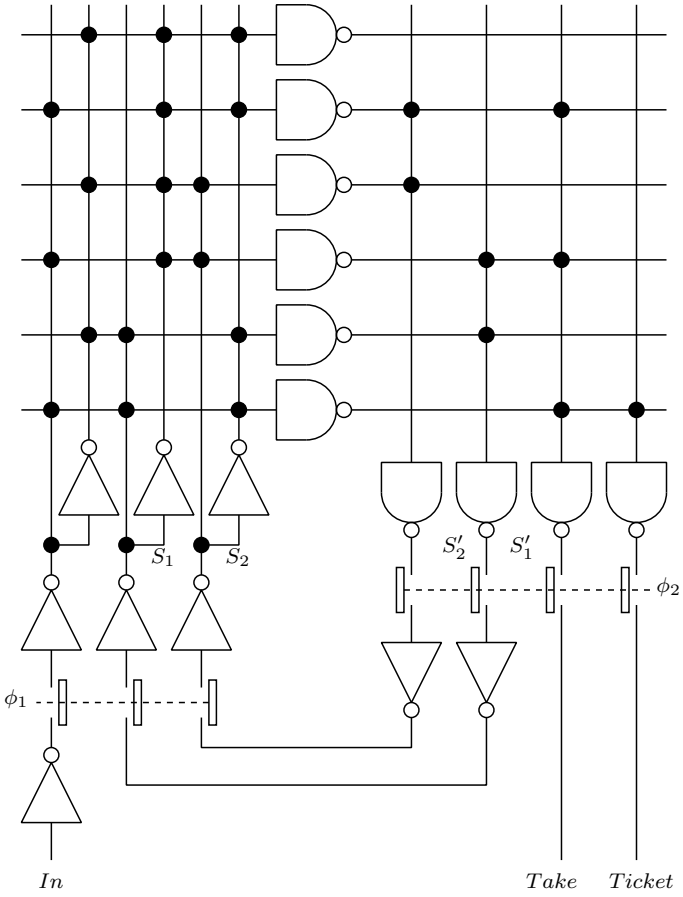


図 5.7　券売機の全体の回路

示した D-FF を融合すればシークエンス回路となる。ただし，遅延回路内の最初の NOT は組合せ論理回路の最後の NAND で置き換えられるので，全体の回路は**図** 5.7 のようになる。

この回路はたまたま，In に 1 が 3 回入ってくると，3 回目に $Ticket$ に 1 が出力される。こうした定まった個数のビットが入るごとに 1 を出力する回路は，**カウンタ (counter)** と呼ばれる。つまり，この回路で $Ticket$ のみを出力とする回路は，3bit カウンタである。

5.4 レジスタ

パストランジスタ (pass transistor) を利用した回路には，D-FF 以外にもいくつか面白いものがある。その一つは，何らかの記憶機能を持つ**メモリー (memory)** 関係の回路である。メモリーも過去を記憶するので，当然シークエンス回路の仲間であるから，当然のことながら，同じような要素を持つのである。

先の章で示すように，コンピュータは，プロセッサとメモリーと周辺回路で構成される。しかし，プロセッサ内部にも若干のメモリーが存在する。このため，プロセッサ外のメモリーを**外部メモリー (external memory)**，プロセッサ内のメモリーを**内部メモリー (internal memory)** と呼ぶ。外部メモリーは大容量のものが多く，応答速度よりは集積度を重視して設計される。これに対し，内部メモリーは集積度よりは応答速度を重視する。

内部メモリーは**レジスタ (register)**[2] とも呼ばれ，任意のときにデータを入れ，任意のときにそれを取り出すことができる。基本的には D-FF

2) レジスタは，もともと金銭登録機のことである。お金の出入りを計算し，それを記憶しておく機械である。プロセッサ内でも算術計算をする際，一時的な記憶が必要であり，その類似性からレジスタというようになったのである。

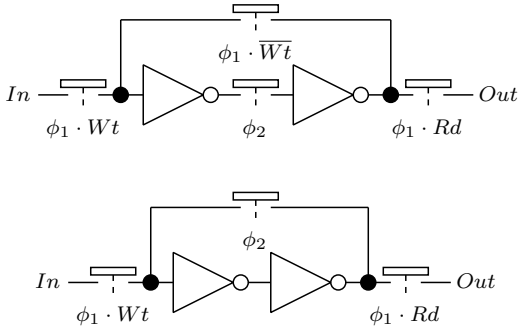


図 5.8　レジスタ回路 (上図は D-FF 型，下図は簡易型。Wt は書き込み，Rd は読み出し信号)

の出力を入力に直接接続しループを構成することで，永久記憶を実現することができるが，**図** 5.8 の上図に示すように，書き込みをしたいときには ϕ_1 のタイミングでループを開けて，書き込み信号のほうを導入する。出力はループを構成したままでも読み出すことができる。

図 5.8 の下図には上図の簡易型を示すが，ループが全体として NOT を構成しないため，ラットレーシングが発生しないことが保証されているので，パナマ運河方式はとっていない。しかし，最初のインバータの前の電位が長時間経過すると，正確な 1 や 0 の値から動いていってしまう危険がある。このため，ϕ_2 のタイミングで毎周期ごとにループを構成し，きちんとした電位に戻す**リフレッシュ (refresh)** という作業を行っている。

普通のレジスタでは，1 個のデータしか記憶できないが，複数のデータを記憶する方法として，複数のレジスタを直列に配置し，新しいデータを入れると古いデータはより奥へ押し込まれていく形式の**シフトレジスタ (shift register)** というものがある。データは押し込まれていくだ

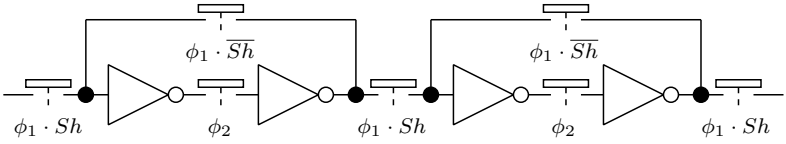


図 5.9　シフトレジスタ回路 (Sh はシフト信号)

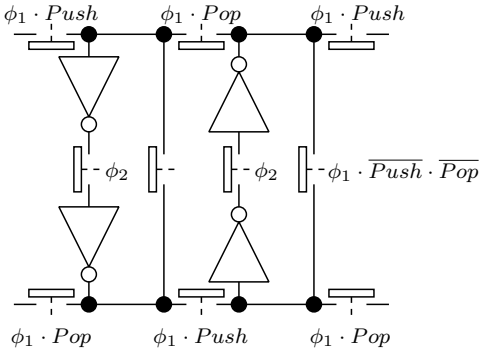


図 5.10　スタックレジスタ回路

けなので，データの内容は一番奥，つまり最後の出口でしか見られない。このため，最初に入れたデータが最初に現れてくるので，こうした機能は **FIFO(first-in-first-out)** と呼ばれる。

シフトレジスタの回路を**図** 5.9 に示す。この場合には，各レジスタ間のデータがすべて異なりうることから，ちゃんと運河方式を採用した D-FF の連続したようなものでなければならない。ただし，D-FF ではクロックのくるたびに，データは右へ移動していくが，この回路では一番左に書き込みを行ったときにだけ，データ移動が起こる。新しいデータが入ってこない場合には，それぞれループを構成して，リフレッシュを繰り返すことになる。

同じように複数のデータを記憶するが, データを入れる側でしかデータを

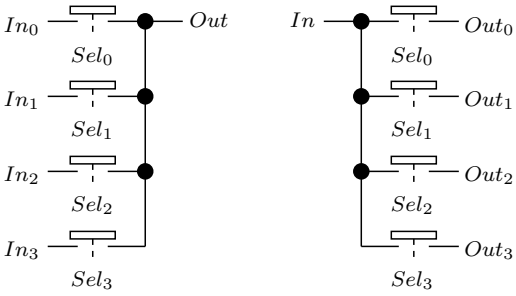


図 5.11　マルチプレクサとデマルチプレクサ (Sel_n は回線選択信号)

取り出さない形式のレジスタがあり，**スタックレジスタ (stack register)** と呼ぶ。この場合にはデータは左右に動かせる必要がある。最後に入れたデータが最初に出てくることから，この機能は **LIFO(last-in-first-out)** と呼ばれる。回路を**図** 5.10 に示す。新しいデータを入れることを**プッシュ (push)**，最後に入れたデータを取り出すことを**ポップ (pop)** という。この回路では D-FF 的な基本回路の間を繋ぐゲートが 2 種類あり，プッシュの際は右シフト信号 $Push$ が ON し，ポップの際は左シフト信号 Pop が ON する。シフトレジスタの場合には，一番右のデータはいつでも読めるが，スタックレジスタの場合には，ポップしたデータを再び読むことはできない。

5.5　セレクタ回路

パストランジスタは入力側の論理状態を出力側に転送することができる。この応用として，もっとも簡単でかつしばしば使われる回路は，**図** 5.11 に示す**マルチプレクサ (multiplexer)** および**デマルチプレクサ (demultiplexer)** である。前節で AND を使って構成した回路と比較すると，トランジスタ数が少なく，簡単な構成であることがわかる。このよ

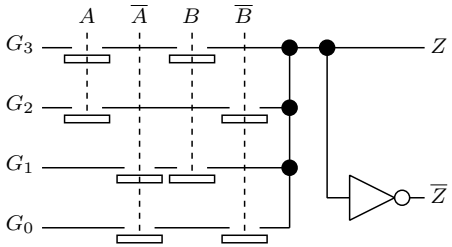


図 5.12　任意関数発生回路

うに，パストランジスタの開閉能力を利用して信号を選択する回路を**セレクタ回路 (selector circuit)** と呼ぶ。

セレクタ回路を用いると，一般に複雑な論理を簡単に構成することができる。一例として**図** 5.12 に示す任意関数発生回路を見て欲しい。A, B の論理値により，縦 4 本のうちいずれか 1 本が導通し，残る 3 本はすべて開放となる。このため，出力 Z はこの導通の線に接続された G_i $(i = 0, \cdots, 3)$ のいずれかの電位と等しくなる。つまり，出力は次のように記載できることになる。

$$Z = G_3 \cdot (A \cdot B) + G_2 \cdot (A \cdot \overline{B}) + G_1 \cdot (\overline{A} \cdot B) + G_0 \cdot (\overline{A} \cdot \overline{B}) \qquad (5.1)$$

この関数を改めて見てみると，A，B の AND-OR 論理になっており，かつ G_i の各値を 0 または 1 のいずれかに選ぶと，A，B の任意の論理関数が実現できることが理解できよう。

改めて基本ゲートである c-MOS の NOT，NAND，NOR を見てみると，これらはすべて 0 と V_h をセレクタで選択して出力へ出す回路とも言える。なお，セレクタ回路には若干の注意が必要である。まず，パストランジスタが直列にあまり長くなると，遅延が無視できなくなってくる。n 個のパストランジスタが直列になると，およそ，NOT 回路の遅延

時間の n^2 倍の遅延が発生する。このため，たかだか 4 パストランジスタで止めるのがよい。それ以上になる場合には，いったんインバータなどで論理を確定し，多段にするのがよい。

また，スイッチとして n-MOS パストランジスタのような c-MOS パストランジスタ以外を使った場合には，出力は完全に 0 または V_h にならない可能性があるので，出力側は必ずインバータなどを使って，信号を強固なものにするのがよい。また同様な理由から，セレクタ回路の出力をそのまま，次段のパストランジスタのゲートに接続することは危険である。

5.6　プリチャージ論理回路

シークエンス回路は，準備期間と実行期間の二つの期間を交互に繰り返しながら論理を進めていく。この準備期間，ϕ_2 のクロック期間中に，出力を常に V_h に持っていき，ϕ_1 の実行期間に出力をそのまま V_h のままにしておくか，0 に引き下げるかを決定するという動作原理で動く論理回路がしばしば用いられる。準備期間に出力を V_h に持っていくことを**プリチャージ (pre-charge)** という。

図 5.13 にプリチャージ方式の NAND や NOR 回路を示すが，通常の c-MOS 論理回路の p-MOS 側を FET 一つにしただけのものである。しかし，p-MOS 側の回路が著しく簡単になるため，回路面積がおよそ半分になり，集積度に対する寄与は大きいので，実際の回路では多用される。

若干の注意が必要であろう。まず，プリチャージは上に置かれた p-MOS によって行われるから，そのゲートには $\overline{\phi_2}$ を与えなければならない。また，下の n-MOS に与えられる信号は，ϕ_1 期間は意味を持つが，ϕ_2 期間は 0 でなければならないことである。

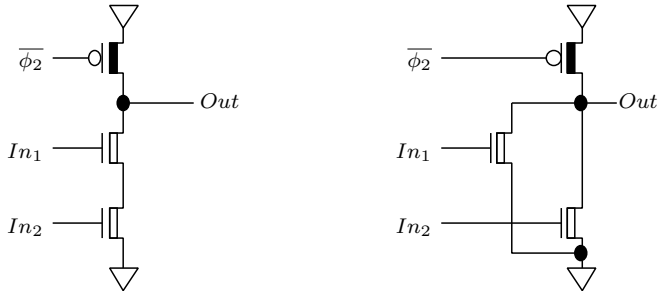


図 5.13　プリチャージによる NAND と NOR 回路

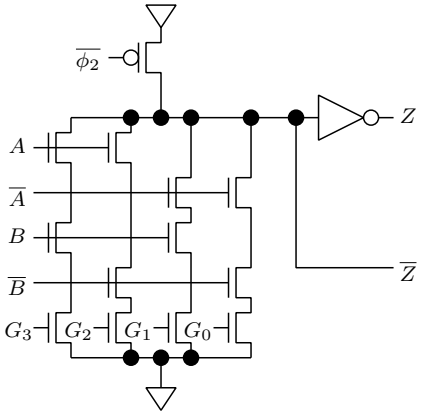


図 5.14　2 入力の任意関数を合成できるプリチャージ回路 (ϕ_2 以外の入力はすべて ϕ_1 のタイミングで与えられる)

一例として，前節で示した (A, B) の任意の論理関数を合成する回路を示そう。任意の関数は式 (5.1) に示したが，さらに次式のように表すことができる。

$$Z = G_3AB + G_2A\overline{B} + G_1\overline{A}B + G_0\overline{A}\overline{B} \tag{5.2}$$

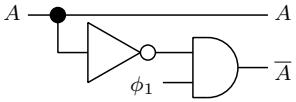


図 5.15　プリチャージ回路で ϕ_1 のタイミング以外 0 となる否定を作る回路

つまり，3 入力 AND の OR である。この程度複雑な論理になると，通常の組合せ論理回路では相当な数の FET を必要とするが，プリチャージ回路では**図** 5.14 のように，極めて簡単となる。なお，プリチャージ回路の直接の出力は $\overline{Z}$ であるが，論理が確定していないとき，つまり ϕ_2 が立ち上がってから ϕ_1 が立ち上がるまでは出力は 1 である。この出力の後に NOT を付けた出力 Z は，論理が確定していないときは 0 を出力する回路となる。この図に示した回路は，第 9 章で説明する算術論理回路 (ALU) の重要な要素となる。

なお，A が ϕ_2 で 0 だったとし，それを利用して $\overline{A}$ を作ろうとするとき，単純に NOT を通すだけではいけない場合がある。例えば，ϕ_1 のタイミング以外では 0 になって欲しい場合，単純に A の NOT をとると，ϕ_2 で 1 になるからである。このような場合には**図** 5.15 のように，ϕ_1 と AND をとるとよい。

参考のために，これにより実現できる任意関数の一覧を**図** 5.16 に示しておく。

演習問題 5

問題 5.1　10 円玉 2 枚で 1 枚のチケットを発券する券売機の状態遷移表

G_3	G_2	G_1	G_0	関数
0	0	0	0	0
0	0	0	1	$\overline{A} \cdot \overline{B}$
0	0	1	0	$\overline{A} \cdot B$
0	0	1	1	$\overline{A}$
0	1	0	0	$A \cdot \overline{B}$
0	1	0	1	$\overline{B}$
0	1	1	0	$A \oplus B$
0	1	1	1	$\overline{A} + \overline{B}$
1	0	0	0	$A \cdot B$
1	0	0	1	$\overline{A \oplus B}$
1	0	1	0	B
1	0	1	1	$\overline{A} + B$
1	1	0	0	A
1	1	0	1	$A + \overline{B}$
1	1	1	0	$A + B$
1	1	1	1	1

図 5.16　任意関数の一覧

を書け。

問題 5.2　$Take = 1$ を出力すると In は自動的に 0 になるとしたが，クロックが速いと，この間の時間遅れが無視できなくなる。つまり $Take = 1$ を出力しても，In はすぐには 0 にはならない。こうしたときには $In = 0$ になるまで待つ必要が出てくる。上記 30 円券売機に対し，そのときの状態遷移図および状態遷移表を書け。

ヒント　それぞれの内部状態に移行する前に，待つための内部状態を用意する必要がある。

問題 5.3　**図** 5.8 の上図の場合，書き込みと読み出しのデータが異なっていても，これら二つの動作を同時に実行できることを確かめよ。

問題 5.4　NOR(AND$(A, B), C)$ を出力する通常の c-MOS 回路とプリチャージ回路を示し，回路規模を比較してみよ。

前半のまとめ

本書の構成としては，まだ，半分に達していないが，この章までが一つの到達点である。これからコンピュータという複雑な装置の説明に入るが，実はコンピュータと言えど，組合せ論理回路とシークエンス回路，さらに回路間の接続の開閉を行うスイッチであるパストランジスタだけで構成できるのである。その意味で，この章までの理解が重要であることがわかろう。

ここまでの学習に基づいて，コンピュータについては，これらの回路がどのように利用されているかを理解するだけですむのである。本章までの理解，特に，組合せ論理回路の構成法，シークエンス回路の構成法が不確かな人は，ここで，一旦，きちんとした復習をして欲しい。

6 データの内部表現とその処理

《目標＆ポイント》徐々にコンピュータの話に移行していくが，本章ではコンピュータで扱うデータが，コンピュータ内部でどのように表現され，どのように処理されるかの概念について学ぶ。

《キーワード》コード，エンコード，デコード，16 進表現，ビット，整数，符号なし整数，符号あり整数，補数，2 の補数，加算，減算，乗算，除算，オーバフロー，突き放し法

6.1 データの 2 進表現

コンピュータは 0 と 1 からなるデータしか扱えない。しかし，現実のデータは 10 進数だったり，小数だったり，文字だったり，はたまた音や静止画や動画だったりする。これらのコンピュータ内での 2 進表現を**コード (code)** と言い，2 進表現に変換することを**エンコード (encode)** という。逆に 2 進表現から現実的なデータにすることを**デコード (decode)** という。とは言っても，コンピュータの出力機器はディジタルのデータを直接受け取ることのできるものが多いので，デコーディングの作業は音声データぐらいしか行われない。

本章では，ほとんどの議論を 2 進表現されたコードで表現するが，長いものになると，0 と 1 の羅列になり必ずしも読みやすいものではない。このため，2 進表現を 4 個ずつまとめた **16 進表現 (hexadecimal**

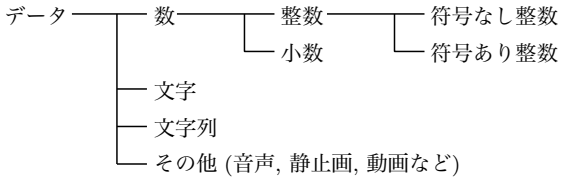


図 6.1　コンピュータで扱うデータ

representation) もしばしば利用される。2 進表現の 0000[1] から 1111 を 0 から 15 と表現しようというものである。ただ 10 以上は 2 桁で記載するのは，見づらいので，アルファベットを使う。10 は `A`，11 は `B`，12 は `C`，13 は `D`，14 は `E`，15 は `F` と表記する[2]。こうすると 0000 0000 から 1111 1111 は，`00` から `FF` と表現されることになる。13 などと書かれたときに，**10 進表現 (decimal representation)** の 13 なのか 16 進表現の `13`(10 進表現では 16+3=19) かの区別が付かなくなるので，16 進表現では，頭に `0x`(または `x`) を付けて，`0x13` などのように表現することにしよう。`0xFF` は 255，`0x100` は 256 となる。

2 進表現は通常圧倒的に桁が多いので 10 進や 16 進とは区別しやすいが，混乱を招きやすいときには必要に応じその旨記載する。

コンピュータで扱うデータは**図** 6.1 に示すように，数，文字，文字の集合である文字列，画像，音声，映像といったさまざまな対象を処理する。基本的には入出力装置があれば，何でも扱える。しかし，そもそもの基本となったのは数と文字である。本章はこの数と文字の内部における表現と，その扱い方について学ぶ。

第 1 章でも簡単に述べたが，**ビット (bit)** という概念がある。もともとは通信の際の情報量を表す単位であるが，0 または 1 で 2 状態のいずれ

1) 2 進表現では，なるべくデータ幅に対応して 0 を入れるようにした。
2) 16 進表現には小文字 `a` から `f` も対等に使われる。

かを通信できるので 1bit，0 または 1 を 2 組用意すれば 00，01，10，11 の四つの状態を通信できるが，これで 2bit，以下 0 または 1 を n 組用意すれば 2^n 個の状態を通信でき，それが $n$bit ということになる。通信の世界ではもう少し厳密に定義されていて，各状態の生起確率にも依存するが，ディジタルの世界では，簡単に 0 または 1 を送るための線の本数と理解してよいだろう。

数を例にしていうと，bit とは 2 進表現された場合の桁数のことを指す。1bit あると，0 または 1 の二つの数を表現することができる。2bit あると，00，01，10，11 の四つの数を表現できることになる。さらに一般に $n$bit あると，0...00，0...01，...，1...11 と 2^n 個の数を表現することができる。また，必要に応じ，負数を含む正負整数を表現することもできるが，それについては以後の節を読んで欲しい。

文字を処理する場合も同様で，コンピュータの内部では 0，1 の集合で処理されるが，集合のサイズが n 個あると 2^n 個の文字を区別することができる。この集合のサイズもビットというので，$n$bit あると 2^n 個の文字を区別して処理することができると言ってよい。例えば，英数字の文字数を表すのに必要なビット幅 (英語大文字小文字で 26 の 2 倍，数字で 10，これに若干の記号を入れて 64 を少し越える) ということで，7bit あれば十分であるが，切りのよい数 (コンピュータの世界では，2，4，8，16，...)[3] ということで 8bit を使って英数字を表現していると言ってもよいだろう。英大文字の「A」は `0x41` である。

もちろん，これは 8bit で文字の形まで表現しているわけではない。文章処理のような場合，文字をそのままの形で蓄積するより，対応コード

3) コンピュータの世界では，このように 2 の冪乗 (べきじょう) を使うことが多いが，これに美意識を感じて引きずられ過ぎる人も少なからず居り，過剰な設計をする場合も散見される。

で蓄積するほうがはるかにコンパクトであるし，間違いも少ないからである。

コンピュータはこうした数や文字を表すのに必要な情報を一挙に並列に入力したり，出力したりして処理を行う[4]。こうした数や文字などを転送する場合の並列のビット数を**ビット幅 (bit width)**，あるいは単に**幅 (width)** ともいう。平行な配線の線数と思ってよいだろう。数や文字のようなデータを送受する線の幅は**データ幅 (data width)** ともいう。これ以外にも，メモリーのアドレスを指定する線の場合の**アドレス幅 (address width)** もある。

数を処理するには，大きな数まで取り扱えるよう，幅は広いほどよさそうであるが，広ければ広いなりに，集積回路の面積を多く使う。このため，適切な幅がある。文字を扱うには，8bit や 16bit の幅が便利である。このため，多くのコンピュータのデータ幅は 8，16，32bit というものが多い。

なお，単に `0x41` というと，文字ならば「A」であるし，整数ならば 10 進表現で 65(2 進数 0100 0001) を指す。いずれなのであろうか。実は，これらデータを扱っているプログラムが心得ているのである。日本語の「ー」が音引きなのか 1 なのかが，前後の文脈がないと理解できないのと同じようなことである。

6.2　整数の内部表現

数，特に整数を扱うには 4，8，16，32bit といったデータ幅を使うのが普通である。創成期のパソコンは 10 進表現 1 桁の計算ができるように，0 から 9 の 2 進表現で 0000 から 1001 に対応して，計算対象のデータの

4) 最初のころのコンピュータは，入出力を直列にして 1 本の線を使って受け渡しをしたものもある。

10 進	2 進	16 進	10 進	2 進	16 進
0	0000	0x0	8	1000	0x8
1	0001	0x1	9	1001	0x9
2	0010	0x2	10	1010	0xA
...	...	...	...	...	...
...	...	...	14	1110	0xE
7	0111	0x7	15	1111	0xF

図 6.2　4bit 符号なし整数の 10 進，2 進，16 進表現の対応表

ビット幅を 4bit とした。その後，英数字を表すのに必要なビット幅に対応して，8bit のものが作られるようになった。このため 8bit を 1byte[5] ともいう。しばしば，b が bit，B が byte の略として使われることがある。

さらに，大きな数の演算が一気にできるようにと，ビット幅 16bit や 32bit のパソコンが作られるようになってきており，かつての大型コンピュータをしのぐものが個人宅にも置かれるようになってきている。ちなみに，Unix では単精度 16bit，倍精度 32bit のものが多い。本書ではデータ幅は 16bit のものを基本とするが，本章に限り，数式計算の全容をつかめるよう，データ幅を 4bit と狭くして例示を行う。

以下，説明にあたって，2 進表現の最上位のビットのことを **MSB(most significant bit)**，最下位のビットのことを **LSB(least significant bit)** と呼ぶことにする。10 進表現の 0，1，2，3，...，15 を 4bit の 2 進表現すると，0000，0001，0010，0011，...，1111 となる。この場合，0 から 15 までの 16 個の数を表現することができる。ビット列をこのように正整数に対応させたものを**符号なし整数 (unsigned integer)** という。念のた

5）1byte の定義は機種により微妙に異なるが，現在はほぼ 8bit に定着しつつある。

めに，符号なし整数の 10 進，2 進，16 進表現の対応表を**図** 6.2 に示す。

当り前であるが，符号なし整数は正数しか存在しない。負数も表現しようとすると工夫が必要である。実は，コンピュータの数表示や四則演算は，昔存在した 10 進数の機械式計算機からの歴史を色濃く引き継いでいる。機械式計算機では各桁ごとに 10 個の歯を持つ歯車が用意されており，その歯車の移動角に対応して 0 から 9 までの数字が表示されるようになっている。これにある数を加える際には，各桁ごとに加える数だけ歯車の歯を動かすことにより，数字を増やす。この数字が 9 を越えると次は 0 になるのであるが，その際，次の桁の歯車の歯をさらに一つ動かす仕掛けが組込まれている。これにより，繰上げ (キャリー) が達成される。引き算を行う際には，減数だけ，各桁の歯を逆に動かして数字を減ずる。どこかの桁で数字が 0 から 9 に推移する際には，一つ上の桁の歯車を歯一つ逆転させる。これにより，繰下げ (ボロウ) が達成される。

機械式計算機で，200 から 1 を引いてみよう。1 の桁では 0 が 9 になる。この結果，10 の桁に対し繰下げが起こり，10 の桁の 0 は 9 になる。さらに 100 の桁に対し繰下げが起こり，100 の桁は 2 から 1 になる。結果は 199 となる。機械式計算機には負数を表現する能力はない。例えば −1 は 0 から 1 を引くことで実現できそうであるが，機械式計算機で 0 から 1 を引くと面白いことが起こる。1 の桁は 0 から 9 なるが，10 の桁に繰下げが発生する。このため，10 の桁も 0 から 9 になり，100 の桁に繰下げが発生する。このように繰下げがどんどん上位桁へ伝播していき，9 がずらっと並ぶことになる。例えば，4 桁しか表現できない機械式計算機であると，9999 となる。5 桁目に繰下げがあるときには，警報がチンと鳴るようになっている。チンとなることで，正数の 9999 と区別できるのである。つまり，機械式計算機では −1 を表現するには，チンと鳴った先の 9999 を使うのである。同様に −2 はチンと鳴った先の 9998 となる。

10 進	2 進	16 進	10 進	2 進	16 進
−8	1000	0x8	0	0000	0x0
−7	1001	0x9	1	0001	0x1
...	...	...	2	0010	0x2
−2	1110	0xE	...	...	...
−1	1111	0xF	7	0111	0x7

図 6.3　4bit 符号あり整数の補数表現 (10 進表現以外は補数)

2 進数の場合も，これと同じ原理で負数を表現する。ビット幅 4 の場合，下 3 桁は機械式計算機の原理で得られた数表現を入れる。MSB は機械式計算機でチンと鳴った先，つまり負数の場合には 1 を，正数の場合には 0 を入れる。−1 は 000 から 001 を引いて得られる 111 に MSB1 を追加し，1111 で表現するのである。1111 は実は 1 0000(= 2^n，10 進数では 16) から 0001 を引いて得られる結果と等しい。したがって 10 進数 −2 は，1 0000 から 0010 を引いて得られる 1110 となる。以下同様にして順に計算し，−7 は 1 0000 から 0111 を引いて，1001 となる。下 3 桁をいくら組み合わせても最大 7 までしかないので，1 から 7 までの正数，−1 から −7 までの負数，それに 0 を合わせて 15 個の数の表現が完成することになる。実はビット幅 4 では 16 個の数を表現できるはずである。残る 2 進表現は 1000 であるが，これはおそらく 8 または −8 に対応させるのがよさそうであるが，MSB が 1 であるため，−8 に対応させる。こうして得られた**符号あり整数 (signed integer)** のコード表を**図** 6.3 に示す。

こうした規則に従って得られた負数と対応する正数は，互いに相手の**補数 (complement)**[6] と呼ばれる。これらの正負の対を見ると，すべて

6) ここに述べた補数は 2 の補数 (2^n の補数のこと) と呼ばれる。これに対し，対の和が 1 の連続となる補数は 1 の補数 (11...1 の補数のこと) と呼ばれる。同様に 10 進表

$$
\begin{array}{rrr}
 & 1011 & (11) \\
+ & 0011 & (3) \\
\hline
 & 1110 & (14)
\end{array}
$$

図 6.4　2 進表現による符号なし整数の加算 (括弧内は 10 進表現)

の 0 と 1 を反転し，それに 1 を加えた形となっている。元の数とその 0 と 1 を反転した数の和は 1111 とすべての bit が 1 となる。これに 1 を加えれば 1 0000 であるので，このことから直ちに理解できよう。ただし，0 と −8 は例外的に補数が存在しない。このような補数を採用すると，次節に示すように，正数や負数が入り乱れたときの加減算の計算が簡単になるのである。

6.3　2 進表現の加減算

まず，**2 進表現 (binary representation)** された符号なし整数の**加減算 (addition and subtraction)** の仕方について学ぼう。例えば 11+3 の計算は**図** 6.4 のようになされる。ここで，いくつかの桁で繰り上げが発生すること，ビットによっては 1+1=10 にさらに繰り上げを加えて，10+1=11 になることなどに注意して欲しい。

この計算で若干問題があるのは，極めて大きな数同士を加えると，元は 4bit 以内の数であっても，和が 16 を越えてしまい，5bit を使わないと表現できなくなってしまうことがあることである。結果を断固 4bit 以内で表示しようとすると，誤った数字になってしまう。こうしたビット

現の場合，和が 10...0 となるものは 10 の補数と呼ばれ，和が 99...9 となるものは 9 の補数と呼ばれる。

$$\begin{array}{rrr} & 1101 & (-3) \\ + & 0101 & (5) \\ \hline & 0010 & (2) \end{array}$$

図 6.5　2 進表現による符号あり整数，負数+正数の加算 (ビット幅を越える部分は無視する)

幅内で結果が正しく表示できないことを**オーバフロー (overflow)** と呼ぶ。この 5bit 目を，通常のコンピュータでは，加算全体のキャリー出力 C_{out} として，計算結果とは別のところへ出力する。符号なし整数の加算 (実は減算も同じ) では，キャリー出力 $C_{out} = 0$ がならば，計算は正しいことになる。

次に符号あり整数の加算を考えよう。オーバフローのことを無視すれば，正数と正数の和は符号なし整数の加算とまったく同じである。

負数のある場合の加算を考えよう。前節で述べたように，負数は補数表現する。つまり元の負数に 1 0000$(= 2^n)$ を加えた数を用いることにする。例えば -1 を表すには，1111 とする。こうすると，正数と負数の加算の結果は 16 だけ大きな値となるが，**図** 6.5 のように，ビット幅の範囲では正しい結果となる。

結果がビット幅で正しく表示できないのは，正数と正数の和が 7 を越える場合，または負数と負数の和が -8 を下回る場合である。正数と負数の和は，ビット幅内のみ着目すれば必ず正しい答を与える。オーバフローという言葉は，本来ビット幅を越える，つまりこの場合には 5bit 目に 1 が出てくることを表すが，符号あり整数の場合には，ビット幅の範囲で正しい答を表現できない場合に**オーバフロー (overflow)** という用語を用いる。つまりオーバフローは，二つの正数の和でかつ加算結果の

MSB が 1 か，二つの負数の和でかつ加算結果の MSB が 0 の場合を検出すればよい。

減算 (subtraction) により**差 (difference)** を求めるプロセスは容易である。減数のほうを符号反転して加算すればよい。符号反転とは補数を得ることであるので，すべての 0 と 1 をビット反転し，1 を加えればよい。符号あり整数の場合，−8 の補数は存在しないので，注意が必要である。ちなみに，−8(1000) をビット反転すると 7(0111) であるが，これに 1 を加えると，8 となって範囲外となる。この際，オーバフローが発生する。

どのような場合に減算の計算結果が 4bit 幅の範囲で正しく表示されるのか，4bit 幅の範囲で正しく表示されない場合には，5bit 目に繰り上げされたキャリーで，真の値が判断できるはずであるが，どのように判断すればよいのかは，符号なし整数の減算の場合と符号あり整数の減算の場合で異なるが，これは各人で考えてもらいたい。

符号なし整数では最大値 15，符号あり整数では −8 から 7 までの 16 個の数しか表現できない。これでは，あまりに小さな数しか計算の対象にできないことになる。データ幅を大きくしないで，もっと大きな数を扱うには，元の数を 4bit ごとに区切って扱えばよい。例えば 4bit 数を 2 組用意すれば，256 個の数を扱うことができる。10 進表現の数でも 1 桁で表現しろと言われれば，0 から 9 までの 10 個の数しか表現できないが，2 桁使えば，0 から 99 まで表現できるようになるのと同じ理由である。

それでは，こうした大きな数の加減算はできるのであろうか。**図** 6.6 には非加数，加数とも，8bit の符号あり整数の加算を示す。まず，下の 4bit の加算は符号なし整数の加算を行っており，上の 4bit の加算は符号あり整数の加算を行っている。さらに下 4bit の加算はオーバフローを起こしており，さらに上の桁へのキャリーが発生している。したがって，上

$$\begin{array}{rrrr} & 0011 & 1101 & (61) \\ + & 1110 & 0110 & (-26) \\ \hline & 0010 & 0011 & (35) \end{array}$$

図 6.6　2 進表現による大きな数の加算

の 4bit の加算の際には，この下からのキャリーを加えて行っている。これだけの手順で，8bit の加算が可能なのである。

二つの大きな整数の加減算の手順を改めて述べると

1） まず，二つの整数を 4bit の整数倍の bit 数に揃える。この際，ともに大きい bit 数の整数のほうに合わせる。符号あり整数で負数の場合には，最上位まで 1 を埋める。

2） それぞれの 4bit ごとに下位から順に加算を行う。

3） 下の 4bit の加算全体でキャリーが発生した場合には，上の 4bit 加算へ繰り上げる。

4） 最上位の 4bit 加算は符号あり整数か符号なし整数かを意識して行う。それ以外の 4bit 加算はすべて符号なし整数として行う。

このようにして，いくら大きな加減算でも実行することができるのである。

6.4　乗算

乗算 (multiplication) も，基本的には 10 進表現の乗算とまったく同様であり，**被乗数 (multiplicand)** を桁をずらして並べ，加算を行うことで，処理する。**図** 6.7 に，符号なし整数の例を示すが，加算器が同時に二つの量の加算しかできないことから，毎行ごとに加算を行うことが，通常の乗算と異なっている。2 進表現におけるメリットとして，**乗数 (multiplier)**

$$12 \times 11 = 132$$

	1100	(12)
×	1011	(11)
	0000	(0)
+)	1100	(12 × 1)
	1100	(12)
+) 1	100	(12 × 2)
10	0100	(36)
+) 00	00	(0 × 4)
10	0100	(36)
+) 110	0	(12 × 8)
1000	0100	(132)

図 6.7　符号なし整数の乗算

の各ビットには 0 と 1 しかないため，0 ならば 0000(実際には，加算をスキップする) を，1 ならば被乗数そのものを置くだけで，各桁での乗算は不要である。

乗算の結果である**積 (product)** の桁数を考察してみよう。10 進表現の積の場合には，4 桁の整数同士の積は最大 8 桁になる。2 進表現の積の場合でも同様に，4bit 同士の積は最大 8bit になる。標準のビット幅をしばしば**ワード (word)** と呼ぶ[7]。したがって，1word と 1word の積の結果は，最大，2word になるので，通常，2word 分の領域を確保する。

7) ワードとは，厳密にはメモリーのデータ幅のことである。

C_{out}	累計 z と乗数 $\underline{y}$	S_{out}	説明 (被乗数 $x = 1100$)
	0000 $\underline{1011}$		z に 0000 を置数する
	0000 $0\underline{101}$	$\underline{1}$	下位ビットを右シフト
	0000 $0\underline{101}$	$\underline{1}$	上位ビットを右シフト
+)	1100		$S_{out} = 1$ なので被乗数 x を加える
	1100 $0\underline{101}$		累計
	1100 $00\underline{10}$	$\underline{1}$	下位ビットを右シフト
	0110 $00\underline{10}$	$\underline{1}$	上位ビットを右シフト
+)	1100		$S_{out} = 1$ なので被乗数 x を加える
1	0010 $00\underline{10}$		累計
1	0010 $000\underline{1}$	$\underline{0}$	下位ビットを右シフト
	1001 $000\underline{1}$	$\underline{0}$	上位ビットを右シフト
			$S_{out} = 0$ なので何もしない
	1001 0000	$\underline{1}$	下位ビットを右シフト
	0100 1000	$\underline{1}$	上位ビットを右シフト
+)	1100		$S_{out} = 1$ なので被乗数 x を加える
1	0000 1000		累計
1	0000 0100		下位ビットを右シフト
	1000 0100		上位ビットを右シフト
			累計が積 $132 (= 128 + 4)$ となる

図 6.8　乗算の手順 $(12 \times 11 = 132)$

乗算をコンピュータ内で行う際は，**図** 6.8 のように，結果のほうを，**シフタ (shifter)** と呼ばれる回路で，右シフトさせながら加算していく。また，乗数のほうも右シフトさせ，シフト溢れ S_{out} が 0 か 1 かによって，0 のときには何もせず，1 のときには被乗数を加算する。このため，結果となる累計 z の右の空き部分に，乗数 y をつなぎ，一緒にシフトしている。これは，ALU レジスタを無駄に使わないことと，z のシフトと y のシフトを同時に一つの命令でできることなどである。見やすくするため，y 部分には下線を付した。

シフト作業はまず，下位ワード (厳密には上位ワードの LSB も含む) を右シフトし，溢れたデータを S_{out} にセットする。次に上位ワード (厳密には C_{out} を含む) を右シフトする。このように，必要に応じ，溢れビットを S_{out} として記憶できるような機能を持ったシフタを用意する必要がある。

加算は 1word 用の加算器で無理なく行うことができる。加算は S_{out}=1 のときのみ実行され，S_{out}=0 のときには実行されない。また，加算の結果，キャリーが生じたときには，C_{out} に保存しておく。

同じ原理で，符号あり整数の乗算も可能である。ただし，正数と正数の積はこれでよいが，いずれかに負数が入っていると，計算は複雑になり，次のような工夫が必要となる。説明に当たり，負数 $-x$ の補数表現 $16-x$ を $\bar{x}$ と記す。$-y$ についても同様である。また，積が負数 $-xy$ になる場合には 8bit の補数である $256-xy$ を求めることになる。

正数 $x\times$ **正数** y**:**　特別な配慮は必要ない。

負数 $(-x)\times$ **正数** y**:**　$256-xy=(16-x)y+16(16-y)$ なので，積の 8bit 補数表現は，$\bar{x}\,y$ に $16\,\bar{y}$ を加える必要がある。

正数 $x\times$ **負数** $(-y)$**:**　$256-xy=x(16-y)+16(16-x)$ なので，積の 8bit 補数表現は，$x\,\bar{y}$ に $16\,\bar{x}$ を加える必要がある。

負数 $(-x)\times$ **負数** $(-y)$**:** $xy = (16-x)(16-y) + 16(x+y) - 256$ となるが，256 は 8bit の範囲を越えるので無視することにすれば，$\overline{x}\,\overline{y}$ に $16x$ と $16y$ を加える必要がある。

今，もともとの 4bit の被乗数および乗数を X，Y で表そう。もし，x が正数であると $X = x$ であるし，負数であると $X = 16 - x$ である。Y についても同様とする。すると，上記の四つの結果は次の 1 行で表現することができる。

$$XY + \mathrm{MSB}(X)\overline{Y} \times 16 + \mathrm{MSB}(Y)\overline{X} \times 16 \tag{6.1}$$

ここで，$\mathrm{MSB}(X)$ などは X などが正数ならば 0，負数ならば 1 を与える。コンピュータは，条件により計算手順を変更するのは時間を浪費するため，嫌われる。このため，この式のように，条件に依存しない計算手順が好まれる。

このように，乗算の基本的手順は変わらないが，積の結果である 2word の上位ワードのほうに，補正を加える必要がある。この手法による $(\pm 7)\times(\pm 6)$ の計算を，**図 6.9** に示す。なお，同図右下の $(-7)\times(-6)$ の計算で，最終の結果の 9bit 目に 1 があるが，これは 256 であり，8bit の範囲を越えるため，無視することができる。その結果は期待通り 42 となる。

いずれも積の結果は 2word としたが，電卓のように扱える最大桁数が決まっていて，引き続く計算で 1word しか利用できない場合もある。このような場合には，2word の結果を出してから，上位ワードに 0 と 1 が混載していたら，オーバフローエラーとみなす。上位ワードが 0 のみの場合は，1word で表現できる正数であり，1 のみの場合は，1word で表現できる負数であり，それ以外の場合は，オーバフローとなるからである。

$7 \times 6 = 42$

	0111	(7)
×	0110	(6)
	0000	(0)
0	111	(14)
01	11	(28)
000	0	(0)
0010	1010	(42)
0000		(0)
0000		(0)
0010	1010	(42)

$(-7) \times 6 = -42$

	1001	$(16-7)$
×	0110	(6)
	0000	(0)
1	001	$(32-14)$
10	01	$(64-28)$
000	0	(0)
0011	0110	$(96-42)$
1010		$(\overline{6} \times 16)$
0000		(0)
1101	0110	(-42)

$7 \times (-6) = -42$

	0111	(7)
×	1010	$(16-6)$
	0000	(0)
0	111	(14)
00	00	(0)
011	1	(56)
0100	0110	(70)
0000		(0)
1001		$(\overline{7} \times 16)$
1101	0110	(-42)

$(-7) \times (-6) = 42$

		1001	$(16-7)$
×		1010	$(16-6)$
		0000	(0)
1		001	(18)
00		00	(0)
100		1	(72)
0101		1010	(90)
0110			$(\overline{-6} \times 16)$
0111			$(\overline{-7} \times 16)$
1	0010	1010	$(256+42)$

図 6.9　符号あり整数同士の乗算

6.5 除算

除算 (division) は乗算の逆演算なので，**被除数 (dividend)**2word，**除数 (divisor)**1word，**商 (quotient)**1word としてよい。

除算については二つの方法がある。まず，**足し戻し法 (restoring method)** と呼ばれる方法であるが，これは 10 進表現の通常の除算のような計算である。一般の足し戻し法では，各桁の計算の際，除数が何回引けるかを推定する必要がある。人間の場合には暗算でおよその目途をつけるのであるが，機械ではそうは行かないので，除数を何回か引いていき，引き過ぎたら除数を足して元へ戻すことを行う。**図** 6.10 左に，足し戻し法を符号なし整数の 2 進表現に適用させた場合の例を示すが，結果の各桁の値は 0 または 1 しかないので，引けるか引けないかの判断だけである。それでも毎回引いてみて，引けなければ戻すというやや面倒な作業が必要となる。

これに対し，**突き放し法 (non-restoring method)** と呼ばれる方法がある。この方法は**図** 6.10 右に示すが，各桁で除数を引いていき，結果が負になっても元に戻さないのである。ただし，そこで引き算は中止する。次の桁の計算では，前の桁で引き過ぎてしまったのだから，今度は除数を足していき，結果が正になったところで止める。以下，符号反転するまで，加算または減算を繰り返すのである。

この突き放し法は，2 進表現された数の場合には特に有利に働く。除数を何回か引いたり，加えたりすると言ったが，これが 1 回，しかも必ず 1 回の加減算となるからである。このように，減算が失敗したときにも構わず次のビットの計算に移動できる分，計算速度は速くなる。

なお，被除数が極めて大きく，除数が極めて小さいと，商が 1word を越えてしまうことがある。これをチェックするには，除算を始める前に，

$135 \div 12 = 11 \cdots 3$

		1011	(11)
1100	)1000	0111	(135)
−)	110	0	$[-12 \times 8]$
	0010	01	(39)
−)	11	00	$[-12 \times 4]$
	111	011	(−9)
戻し)	11	00	$[+12 \times 4]$
	010	011	(39)
−)	1	100	$[-12 \times 2]$
	00	1111	(15)
−)		1100	$[-12 \times 1]$
	0	0011	(3)

$135 \div 12 = 11 \cdots 3$

		1011	(11)
1100	)1000	0111	(135)
−)	110	0	$[-12 \times 8]$
	0010	01	(39)
−)	11	00	$[-12 \times 4]$
	111	011	(−9)
+)	1	100	$[+12 \times 2]$
	00	1111	(15)
−)		1100	$[-12 \times 1]$
	0	0011	(3)

図 6.10　符号なし整数の除算 (従来の足し戻し法と突き放し法)

被除数の上位の 1word と除数の 1word を比較すればよい。除数のほうが小さいときには，商は上位に値を持ち，2word になるので，エラーとすればよい。

符号あり整数の場合にも，まったく同様の議論が成立する。突き放し法による除算の一例を**図** 6.11 に示す。若干ルールがわからないかも知れないが，図中，除数が正の場合には，下線付き数字が 0 であると次の行で除数を減算，1 であると加算としている。除数が負の場合には，逆になる。

$44 \div 6 = 7 \cdots 2$

```
          0111        (7)
       ----------
0110 )0010 1100       (44)
  −)  011  0     [−6 × 8]
      ----------
      111  11         (−4)
  +)   01  10     [+6 × 4]
      ----------
       01  010        (20)
  −)    0  110     [−6 × 2]
      ----------
        0  1000        (8)
  −)       0110     [−6 × 1]
      ----------
           0010        (2)
```

商は $\underline{1000}$ のビット反転

$(-44) \div 6 = -7 \cdots -2$

```
          1001       (−7)
       ----------
0110 )1101 0100      (−44)
  +)  011  0     [+6 × 8]
      ----------
      000  01         (4)
  −)   01  10     [−6 × 4]
      ----------
       10  110      (−20)
  +)    0  110     [+6 × 2]
      ----------
        1  1000       (−8)
  +)       0110     [+6 × 1]
      ----------
           1110       (−2)
```

商は $\underline{0111}$ のビット反転+1

$44 \div (-6) = -7 \cdots 2$

```
          1001       (−7)
       ----------
1010 )0010 1100       (44)
  +)  101  0     [−6 × 8]
      ----------
      111  11         (−4)
  −)   10  10     [+6 × 4]
      ----------
       01  010        (20)
  +)    1  010     [−6 × 2]
      ----------
        0  1000        (8)
  +)       1010     [−6 × 1]
      ----------
           0010        (2)
```

商は $\underline{1000} + 1$

$(-44) \div (-6) = 7 \cdots -2$

```
          0111        (7)
       ----------
1010 )1101 0100      (−44)
  −)  101  0     [+6 × 8]
      ----------
      000  01         (4)
  +)   10  10     [−6 × 4]
      ----------
       10  110      (−20)
  −)    1  010     [+6 × 2]
      ----------
        1  1000       (−8)
  −)       1010     [+6 × 1]
      ----------
           1110       (−2)
```

商は $\underline{0111}$

図 6.11　突き放し法による符号あり整数の除算

$$40 \div 6 = 6 \cdots 4$$

		0110	(6)	
0110	)0010	1000	(40)	
−)	011	0	$[-6 \times 8]$	
	111	10	(-8)	
+)	01	10	$[+6 \times 4]$	
	01	000	(16)	
−)	0	110	$[-6 \times 2]$	
	0	0100	(4)	
−)		0110	$[-6 \times 1]$	
		1110	(-2)	通常はここで終了
+)		0110	[6]	被除数が正なので | 除数 | を加える
		0100	(4)	修正された余り

図 6.12　突き放し法による余りの処理

結果である商は，除数が正の場合には，被除数以外の下線付き数字を上から順に読んで，0 と 1 を反転したものが除算結果となっている。除数が負の場合には，そのままを並べたものとなっている。ただし，除算結果が負の場合には 1 を加える。

突き放し法はこのように便利であるが，若干注意が必要である。例えば，$40 \div 6$ の計算をしてみよう。**図** 6.12 に示すように，通常の 6 の倍数の加減算の終了後，**余り (remainder)** が -2 となっている。正の余りとなるはずであるが，符号が逆である。このため，最後に引いた数を戻す

必要がある。なお，商はこのままでよい。

本書では，余りの正しい符号は被除数の符号と同じであるとしている。したがって，除算の最後で，被除数が正数で，余りが負数となってしまったときには，除数の絶対値を加え，被除数が負数で，余りが正数となってしまったときには，除数の絶対値を減ずるという補正をする必要がある。

コンピュータ上で除算を行う場合には，乗算と同様，算術演算をなるべく同じところで行いたいので，累計と結果をシフトさせながら，計算を行う。**図** 6.13 に，**図** 6.12 に対応するコンピュータ内での除算の手順を示す。**図** 6.8 に示した乗算の手順と同様に，累計 z の右の空き部分に，結果となる y をつめて記載している。見やすくするため，y 部分には下線を付した。

また，上位ワード (厳密には下位ワードの MSB を含む) の左シフトの際，溢れを S_{out} として記憶する必要がある。下位ワード (厳密には上位ワードの $\overline{\text{MSB}}$ を含む) の際の溢れは無視する。このように，除算でも，必要に応じ，溢れビットを S_{out} として記憶できるような機能を持ったシフタを用意する必要がある。

なお，除数が正の場合には説明にあるように「下位ワードを左シフト ($\overline{\text{MSB}}$ を埋める)」とあるが，除数が負の場合には「下位ワードを左シフト (MSB を埋める)」となることに注意して欲しい。

6.6 小数の内部表現

前節では整数の表し方について述べたが，本節では小数はどのように表すかについて述べよう。小数は，まず 仮数 $\times 2^{\text{指数}}$ という表現をする。**仮数 (mantissa)** は通常，1 以上 10 未満の符号付き小数とする [8)] 。ま

8) 仮数を，0.1 以上 1 未満の符号付き小数とするシステムもある。この場合，1.0 は $+0.1 \times 2^{+1}$ と表現される。

S_{out}	累計 z と結果 $\underline{y}$		説明 (除数 $x = 0110$)
	0010	1000	累計 z に被除数を設定
$\underline{0}$	0101	1000	上位ワードを左シフト
−)	0110		S_{out} =0 なので | 除数 x | = 6 を減ずる
	1111	1000	累計
	1111	000$\underline{0}$	下位ワードを左シフト ($\overline{\text{MSB}}$ を埋める)
$\underline{1}$	1110	000$\underline{0}$	上位ワードを左シフト
+)	0110		$S_{out} = 1$ なので $\|x\| = 6$ を加える
	0100	000$\underline{0}$	累計
	0100	00$\underline{01}$	下位ワードを左シフト ($\overline{\text{MSB}}$ を埋める)
$\underline{0}$	1000	00$\underline{01}$	上位ワードを左シフト
−)	0110		$S_{out} = 0$ なので $\|x\| = 6$ を減ずる
	0010	00$\underline{01}$	累計
	0010	0$\underline{011}$	下位ワードを左シフト ($\overline{\text{MSB}}$ を埋める)
$\underline{0}$	0100	0$\underline{011}$	上位ワードを左シフト
−)	0110		$S_{out} = 0$ なので $\|x\| = 6$ を減ずる
	1110	0$\underline{011}$	累計
	1110	$\underline{0110}$	下位ワードを左シフト ($\overline{\text{MSB}}$ を埋める)
+)	0110		余りと被除数の符号が異なるため，6 を戻す。y が負の場合は 1 を加える
	0100	$\underline{0110}$	余りと商

図 6.13　除算 $40 \div 6 = 6 \cdots 4$ の手順 (説明中の MSB は，上位ワードの MSB である。)

た**指数 (exponent)** は，符号付き整数とする。

これで，非常に大きい数からごく小さい数まで，かなりの広い範囲の数を扱うことができる。ただし，指数と仮数の両者を収容しなければならないので，相当のビット幅が必要となり，データ幅の小さい CPU では実現しないことが多い。例えば Unix[9] の単精度と呼ばれる浮動小数点表示でも 32bit 幅が要求され，より高精度の計算に使われる倍精度には 64bit を割り振っている。なお，単精度の場合，指数には 8bit，仮数に 24bit を割り当てることが多い。

小数の加減乗除のやり方については，少し頭をひねれば想像がつくと思うので，各自，考えてみて欲しいが，概要を述べれば，加減算については，指数を合わせるように仮数を調整してから，仮数同士の計算を行う。また，乗除算については，結果の仮数部は乗除数および被乗除数の仮数同士の乗除算の結果を用い，結果の指数部は乗除数および被乗除数の指数同士の加減算の結果を用いる。結果の仮数部の符号については，同符号の際は正，異符号の際は負とする。

6.7 文字の内部表現

まず文字のコードと言っても，文字の形をそのままデータとして扱うわけではない。例えば文字列を記憶しておく際，各文字ごとに番号を付け，その番号だけを記憶しておけば記憶容量はうんと減る。この番号を**文字コード (character code)** という。出力の際は，その番号ごとに形を対応させればよい。一般に，文字の形を記憶するには，大きな記憶容量が必要であるので，この対応表はたった一つだけ記憶しておき，ワープロ

9) Unix はパソコンの Windows に対応する OS の一つで，より大きなコンピュータ用に開発された。動作が安定なため，現在でもサーバなどの OS として使われている。Linux などの祖先。

ソフトなどのような文字列を切ったり貼ったりしているときには，番号のような軽いコードで処理する。これら文字列をディスプレーやプリンタに出力するときだけ，形の記憶領域を見に行けばよいのである。

文字は表したい総字数によって，コード化するのに必要なビット幅が異なる。例えば英数字をすべてとなると，数字 10 文字，英字大小 52 文字，それとピリオドや%などの記号を加えると 64 個を少し越える。64 個を 2 進表現すると 6bit であるので，やや余裕を持たせ 7bit を使うことにしている。ちょっと面倒なのは数字である。数字のコードを，表す数そのものの 2 進表現にしておけば簡単であったのだが，ほとんどのコード表で，数字は 2 進表現とは異なるコードになっている。

図 6.14 に代表的な文字コードである**アスキーコード (ASCII code)** を示す。第 0 列と 1 列は主として通信用の制御コードが割り当てられているが，現在はあまり使われないので，ほとんどを省略した。LF は改行，CR は復帰，SP は空白，DEL は消去を意味する。0 から 9 までの文字コードが `0x30` から `0x39` と，数そのものの 2 進表現でないことを確認して欲しい。さらに，文字の国際化や種々の余裕を持たせるため文字コードのビット幅は 8bit であると言ってもよいだろう。

日本語の文字にはさらにひらがなとカタカナ，さらに数千字の漢字がある。そこで，まず幅を 8bit として，カタカナ (半角) を収容した時代がある。漢字も含めようとすると，当然，8bit の幅でも不足である。そこで，16bit の幅を使い，ひらがな，カタカナ (全角)，漢字のすべてを収容している。面倒なことに，日本語の文字を収容するには，歴史に依存したいくつかの手法がある。Windows や Mac 系のシステムで使われている**シフト JIS コード (shift JIS code)** と，Unix 系のシステムで使われている **EUC コード (EUC code)** がある。これ以外にも，日英の切り替わるときに特別なコードを入れ，その間は 16bit コードであると認識

	0	1	2	3	4	5	6	7
0			SP	0	@	P	‘	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6				6	F	V	f	v
7			’	7	G	W	g	w
8			(	8	H	X	h	x
9			)	9	I	Y	i	y
A	LF		*	:	J	Z	j	z
B			+	;	K	[	k	{
C			,	<	L		l	|
D	CR		-	=	M	]	m	}
E			.	>	N	^	n	~
F			/	?	O	_	o	DEL

図 6.14　アスキーコード (表の上欄には上位 3bit，横欄には下位 4bit を示す)

させる **JIS コード (JIS code)** と呼ばれるものもある。

さらに，国際化が進み，世界中の代表的なすべての文字に対するコードを作ろうという **UTF コード (UTF code)** と呼ばれる 24bit コードもある。これらの詳細については，本書の趣旨からずれるため，他書を参考にして欲しい。

文字を 1 次元的に並べた集合を**文字列 (character string)** あるいは**テ**

キスト (text) と呼ぶ。文字数は短いものもあれば，長いものもある。いつも同じ長さに収容しようとすると，記憶が無駄になるので，多くの場合，次に述べる二つの方法のいずれかを利用して，メモリーを有効に利用する。

第一の方法は，文字列の最初に文字列の長さを記載しておく方法である。しかし，16bit 幅の場合，その幅内に書ける最大数は 65,536 であるので，これ以上の文字列は定義できない。これ以上の文字列の場合には 65,536 個の文字列に分解して収容することになる。

第二の方法は，文字列の最後に特別な記号を入れておく方法である。ASCII コードを見てみると，表の最初のほうは空けてある。例えば 0 番を文字列の終端記号として利用する。これであると，メモリーの許す限りいくらでも長い文字列が収容できる。ただ，前から順番にチェックしていかないと文字列の終了点がわからないという欠点もあり，いずれが便利なのか一概には言えない。

演習問題 6

問題 6.1　**図** 6.2 に示した符号なし整数の 10 進，2 進，16 進表現の対応表を完成させよ。

問題 6.2　**図** 6.3 に示した 2 進，16 進，10 進 (正負) の正負の数の対応表を完成させよ。

問題 6.3　11+7=18 の計算でオーバフローが起こることを示せ。

問題 6.4　二つの正整数に対応して 2 次元座標系を組み，横軸，縦軸とも 0 から 15 とする。この座標系で縦軸と横軸の加算を行った際，オー

バフローの起こる領域，起こらない領域を区別してみよ。この結果，全計算の約半分がオーバフローを起こすことが理解できよう。

問題 6.5　符号あり整数として −8(1000) に順に 1 を加えていき，4bit の範囲のみに着目すると，**図** 6.3 になることを確認せよ。

問題 6.6　二つの符号あり整数に対応して 2 次元座標系を組み，横軸，縦軸とも −8 から 7 とする。この座標系で縦軸と横軸の加算を行った際，オーバフローの起こる領域，起こらない領域を区別してみよ。この結果，全計算のおよそ 1/4 がオーバフローを起こすことが理解できよう。またオーバフローを起こすことを検出するには，上記の判断法が適切であることも理解できよう。

問題 6.7　**図** 6.9 に示した正負 4 種類の積に対し，**図** 6.8 のようなコンピュータ内の乗算の手順を示せ。

問題 6.8　二つの符号なし整数に対応して 2 次元座標系を組み，横軸，縦軸とも 0 から 15 とする。この座標系で縦軸と横軸の乗算を手計算で行ってみて，結果が 1word を越える領域，越えない領域を区別してみよ。

問題 6.9　二つの符号あり整数に対応して 2 次元座標系を組み，横軸，縦軸とも −8 から 7 とする。この座標系で縦軸と横軸の乗算を手計算で行ってみて，結果が 1word を越える領域，越えない領域を区別してみよ。

7 コンピュータ

《目標&ポイント》ここまでの準備により，いよいよコンピュータのしくみについての議論が可能となった。本章では，コンピュータのしくみの概要，その構成要素について述べる。

《キーワード》コンピュータ，中央処理装置 (CPU)，バス，データ処理部，制御部，命令，制御線，フラグ，算術論理回路 (ALU)，レジスタ，シフタ，フラグレジスタ，メモリー，RAM，ROM，周辺装置

7.1 コンピュータの概要

現在のコンピュータは各種の計算が可能であるが，それ以外にもワープロとして文書作成ができたり，静止画や動画を扱ったり，それこそ何でもできる。

しかし，もともと，**コンピュータ (computer)** は，時間のかかる計算を，人手を使わず，かつ正確に短時間で行うために開発された計算専用機械であった。それが，現在のように何でも扱える機械として進化してきたのである。電卓もコンピュータの一種である。どちらかというと，電卓という場合には，人が計算の順序を指示するのに対し，コンピュータという場合には，計算の指示は，事前にメモリーに書き込まれて，それを次々にこなしていくというイメージが強いが，電卓とコンピュータのきちんとした仕切りはない。

コンピュータはおよそ**図 7.1** に示すような構成となっている。まず

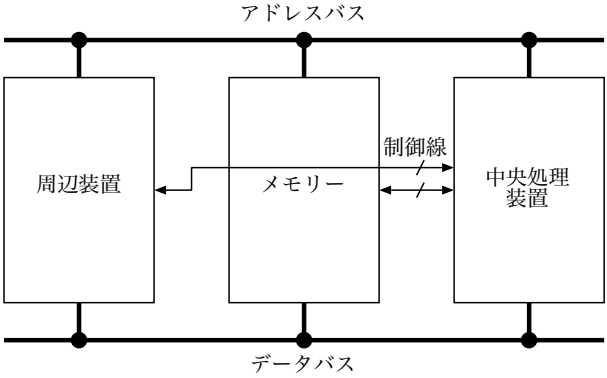


図 7.1　コンピュータの構成の概要 (以下，太線はバス，細線に付けられた斜線は複数線を示す)

動作を指示する命令やデータを書き込んだ**メモリー (memory)**，これら命令を実行する**中央処理装置 (central processing unit)** あるいは **CPU**，キーボード，ディスプレー，ハードディスクなどの各種の**周辺装置 (peripheral unit)**[1] などから構成されている。

これらがデータを送受するためには，**バス (bus)**[2] と呼ばれるビット幅の線からなる平行線を利用する。情報を送りたい装置はバスに対して付けられたスイッチを ON にして，情報をそこに載せる。情報を受けたい装置は，同様にバスに対して付けられたスイッチを ON にして，情報をそこから受け取る。つまり，情報の送受に関わる装置だけがバスに接続して，情報の授受を行うのである。まずは，装置間でデータを授受するための**データバス (data bus)** が必要である。このバスのビット幅は前章でも述べたように，創成期は 4bit であったが，その後，8bit，16bit，

1) かつては本当に大きな装置だったので，装置と訳したが，現在は集積回路チップである。英文では unit なので違和感はない。
2) バスとは乗合バスに由来し，いくつかの装置が共用で使う通信線のことである。

32bit と増加しつつある。本書では 16bit 幅を前提にして議論を行おう。つまり，**図** 7.1 の下方にある太線は 16 本の平行線である。

計算の元になるデータや，計算の指示を与える命令は，メモリーに置かれる。このメモリーの何番地にアクセスするかを知らせるためのバスも必要である。これを先のデータバスに対し**アドレスバス (address bus)** という。昔のプログラムはデータ幅 8bit の 256 アドレスぐらいの量に書くことができた。このため，アドレスを指定するために必要なビット幅は，256 アドレスに相当する 8bit で済んだ。しかし，プログラムの規模が大きくなると，その 2 乗であるビット幅 16bit でかつ約 64k アドレス (16bit) が必要となるようになってきた。これより大きなアドレス空間を必要とする場合には，32bit 幅 (約 4G アドレス) を使ったり，容量の大きなハードディスク上に記憶し，それを順番に 16bit でアドレス表現のできるメモリー (つまり約 64k アドレス) に移動してきたりする。本書ではデータ幅もアドレス幅も 16bit として説明を続ける。つまり，**図** 7.1 の上方にある太線も 16 本の平行線である。

周辺装置は，仮想的にメモリーのアドレス空間の一部に割り当てる場合が多い。例えば，CPU がメモリーだと思って `0xFFF0` から `0xFFFF` 番地のいずれかからデータをもらうと，キーボードの文字が読み込まれたりするといった形式である。このため，メモリーと周辺装置は対等の位置に記載している。

現在のコンピュータは，文字や画像を取り扱ったり，種々の判断をするなど，一見，計算機械のようには見えないかも知れないが，その構成要素は最初に作られたコンピュータとほぼ同じである。そこでまず，計算のみを目的としたやや古いタイプのコンピュータの構成について学ぼう。古いとは言っても，現在のコンピュータもほぼ同じような構成で作られている。しかし，本書では，よりコンピュータの全貌がつかめるよ

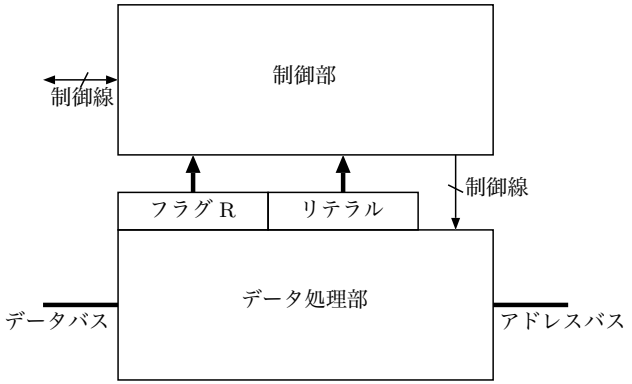


図 7.2　CPU の構成の概要 (フラグ R はフラグレジスタの意味)

う，意識的に，データ幅もアドレス幅も現在としては狭い幅である 16bit であるとして説明を試みる。

7.2　中央処理装置 CPU

コンピュータの中心は何と言っても CPU である。すでに前章に示したように，どんな情報機械でも必ずシークエンス回路で実現できる。しかし，こうした計算に特化した機械を純粋なシークエンス回路で作ろうとすると，そのサイズは巨大になることが知られている。そのため，ある程度，機能分化した機械により実現するのがよい。このため CPU は概ね**図** 7.2 のような構成をしている。

ここで**データ処理部 (data processing unit)** とは，それこそ加減算を行う回路である。[3)] いうならば，算盤 (そろばん)，あるいは電卓と言ったほうがよい対応かも知れない。電卓はそれ自身計算機械であるが，機

3) 英語では装置 (チップ) 内の機能部分も unit である。本書では装置内の機能部分は部と訳した。

能としては限りなくデータ処理部に対応している。データ処理部は計算を担当していることからもわかるように，レジスタと呼ばれる記憶部分を除外すれば，ほとんどの部分が組合せ論理回路であるので，その理解には第 4 章の知識が全面的に役に立とう。

この電卓のボタンを押す頭脳のようなものが**制御部 (control unit)** である。制御部はメモリーから次々と**命令 (instruction)** を読み込み，それを順に理解してデータ処理部を制御していく。そういう意味で，メモリー上の命令群は指令書のようなものである。制御部はシークエンス回路そのものであるため，その理解には第 5 章の知識が全面的に役に立とう。

また**フラグ (flag)** とは，データ処理部での実行の成否を制御部に知らせるための情報であり，電卓のエラー表示のようなものである。例えば，加算の結果がデータ幅を越えたときなどに，ここに 1 がセットされる。フラグは制御部に対するデータ処理部からの入力であり，制御部はそれらを見て，以後の計算を続行すべきかなどを決定する。また，制御部からデータ処理部への出力は制御線を介して行われる。これ以外に，制御部とデータ処理部間に，アドレス幅程度のデータのやりとりができるように，**リテラル (literal)** と呼ばれる入出力ポートが用意されている。

7.3　データ処理部

前述のように，本書では，加減算器やシフタや一時メモリーは持っているが，乗除算器は持たないコンピュータしか扱わないことにする。かつて，機械式の計算機があったが，それは加算と減算しかできなかったが，それと同じ機能を持っていると考えてよい。なお，乗除算は桁をずらしながら加減算を行うことで達成できる [4]。

4) 数値計算を専門とするスーパーコンピュータや 3D を多用するゲーム機のような，乗除算を多く行うコンピュータは，専用の乗除算器を有している。

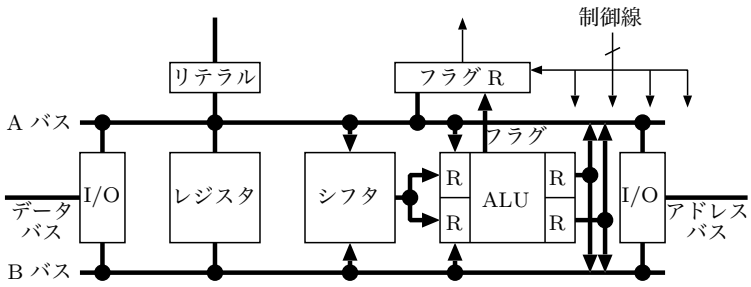


図 7.3　データ処理部 (R はレジスタ，I/O は外部への入出力)

データ処理部 (data processing unit) はいろいろなデータの処理を行うが，処理内容によって異なる回路に機能分化しているため，こうしたデータを，機能回路から別の機能回路に動かす方法が必要である。それには**図** 7.3 に示すように，**バス (bus)** と呼ばれるデータ幅のビット数だけ平行に引かれた線を利用する。どの機能回路もバスとの間にスイッチで繋がっており，ある機能回路がスイッチを接続してバスにデータを載せ，そのデータを必要とする別の機能部分が，バスにスイッチ接続してデータをもらう。こうして，データをやりとりしたい二つの機能回路が共通のバスを利用するのである。

機能部分の中で中核となるのは，数値計算を行う**算術論理回路 (arithmetic logic unit)** あるいは **ALU** である [5)]。これは加減算のような数値計算をするもので，多くの場合，2 入力を受けて，1 出力を直ちに出す。このためよく Y 字の形で記載される。

加減算だけでなく，数値の大小比較をしたり，ビットごとの AND や OR などの論理演算をしたりする。また正数からその補数を作り出すような計算もする。この場合には，1 入力 1 出力となるが，こうした場合に

5) ALU も英語では unit である。

は，片方の入力を利用しないだけである。当然，計算結果は若干の遅延はあるものの直ちに出力されるので，論理回路の一種である。また，いろいろな計算に対処できるように，複数の論理回路を含むことになるが，できたら可変的な論理回路として設計しておくほうがよいであろう。

ALU は制御部に対し，計算の際のオーバフローなどの異常や数値比較などの大小結果などを知らせるためのフラグと呼ばれる特別な出力を持つ。またフラグの内容は制御部の指示に基づいて，**フラグレジスタ (flag register)** と呼ばれる一時メモリーに記憶しておくことができ，その内容は制御部に利用される。

乗除算や論理演算などで必要なビットシフトを行うために，**シフタ (shifter)** が ALU の手前に置かれる。これは，バスのデータを取り込んで，所定のビット数だけずらしたデータを出力する。

計算途中の値を保持する一時メモリーである**レジスタ (register)** も，いくつか必要である。このレジスタだけが，任意のクロック数だけ遅延を与えることのできる，いわばデータ処理部で唯一のシークエンス回路である。

図中**リテラル (literal)** とあるのは，制御部とデータ処理部の間の情報の授受のためのパスであり，A バスそのものの分岐である。主として，命令にアドレスが入っているとき，その処理をデータ処理部に依頼するための授受を行う。

制御線については次節で説明する。

7.4　制御部

制御部 (control unit) は，作業命令書に基づいてデータ処理部のあちこちのスイッチの ON，OFF を行い，データに順次，いろいろの処理を

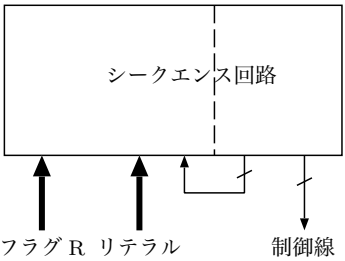


図 7.4　制御部

していく。実体は，**図** 7.4 に示すように，前述のシークエンス回路そのものである。

主な仕事は以下のようであり，これによって命令を順次こなしていくのである。詳細については第 10 章で説明する。

- 次に命令の入っているアドレスを計算する。多くの場合，現在のアドレスに 1 を加える。
- そのアドレスをアドレスバスに載せ，メモリーからの返事を待って，その内容をデータバスから拾う。
- 命令を解釈し，リテラルにも処理に必要なデータを送る。また同時に，データ処理部内の各所のゲートの開閉などの制御信号を作り出す。

制御部の入力の主なものは，データ処理部が出すフラグである。実際には，フラグの内容はフラグレジスタに一時的に記憶され，それを利用する。また，外部のメモリーや周辺装置の状態を示す信号も入力として使われる。

出力の主なものは，**制御線 (control lines)** によりデータ処理部へ送られ，データ処理部の動作を制御する。具体的には，次のような制御を行う。

- 各機能回路間のデータのやり取りの制御。各機能回路とバスの間のスイッチを制御して，データをバスに載せたり，データをバスから得たりする。
- ALU の機能の制御。ALU は加算，減算，符号反転，比較など多数の機能を持っているが，その機能を選択する。
- シフタにシフト量を与える。
- フラグをフラグレジスタに移動する。

また，外部のメモリーや周辺装置への制御信号も出力される。

7.5　メモリー

命令も含めデータはすべてメモリー内に格納されている。現在のコンピュータは必要なソフトウェアが爆発的に増大し，メモリーだけでは収容できなくなっている。このため，周辺装置にハードディスクなどを置き，データは基本的にはそこに格納し，現在まさに利用しようという直前に，メモリーに移動してきて利用するようになってきている。このため，コンピュータの起動直後に動作するハードディスクから重要なデータを移動するような機械語の初期化プログラムは，メモリー上の書き換え不能な場所に置いておく必要がある。その他の大部分の領域は書き換え可能であるほうが便利である。

実際，メモリーには物理的[6]には読み出し専用メモリーと書き換え可能メモリーの 2 種類があり，CPU から見ると，論理的には連続的に繋がっているようになっている。先に示した周辺装置も物理的にはまったく異なる存在ではあるが，論理的[7]には RAM や ROM と同じように，

6）コンピュータ屋さんの好きな言葉であるが，実際の装置がどうなっているかを示す場合に物理的という表現をする。

7）同じく，実際の装置には依存せず，見掛け上どのように見えるかを示す場合に論理

図 7.5　メモリー空間の割り当ての例 (0x0000 と 0xFFFF 以外は比較的自由に選べる)

メモリーの一部に見えるように細工がされている。

このため，論理的なメモリー空間は，およそ**図** 7.5 のように構成されている。CPU は最初にメモリーの 0xFFFF 番地を読みに行き，そこに記された初期アドレスの命令を実行に行く。

書き換え可能メモリー (random access memory) あるいは **RAM** は CPU のレジスタのようなものである。ただし，CPU のレジスタは後述するように，1bit 当たり 7 個の FET で構成されているが，外付けのメモリーの RAM は 1bit 当たり 2 個ほどの FET で構成されている。ただし，これほど FET 数が削減できるのは，メモリーが極めて巨大であり，複雑な処理を行う周辺回路の FET 数が無視できることによっている。また，CPU 内のメモリーに比べ，読み出しの速度が遅い。

的という表現をする。

書き換え不能な**読み出し専用メモリー (read only memory)** あるいは **ROM** はアドレスを与えるとあらかじめ書き込まれた固定のデータが出力される。その内容は，コンピュータ自身では書き換えることができない。実体は，先に述べた NAND-NAND 回路と同じものである。NAND-NAND 回路も，入力を与えると出力が決定してしまうことから，対応は明らかであろう。

メモリーはアドレスを指定しても，CPU 内のメモリーのようにすぐには返事のデータが返されるとは限らない。一般に，応答遅れがやや大きい。このため，データバスにデータが確実に乗った，あるいはデータを確実に受け取ったことを CPU に伝える制御線を持っている。さらに，RAM のようにデータを受ける場合とデータを送る場合があるときには，CPU からどちらであるかを指示する制御線も必要である。

7.6　周辺装置

周辺装置とはキーボード，マウス，ディスプレー，ハードディスク，CD ドライブ，ネットワークといった CPU とメモリーを除くほとんどすべてのものを指す。キーボードやマウスのように入力専用のものもあれば，ディスプレーのような出力専用のものもあり，ハードディスクやネットワークのように入出力共用のものもある。

いずれも，論理的にはメモリーと同じように見せることができる。このため，周辺装置には，任意のアドレスを名乗れるような半固定の設定スイッチが付いている。さらに，RAM と同じような制御線が必要である。

なお，ハードディスクやフロッピーディスクや CD ドライブなどでは，その装置の選択が終了しても，さらに，その内部のアドレスを選択する必要がある。これは装置選択後にデータバスを利用して内部のアドレス選

択をし，その後に同じデータバスを利用してデータを送受するなど，何回かの操作によりアクセスする。

演習問題 7

問題 7.1　次の用語を理解したかどうか確認せよ。

1）中央処理装置，周辺装置，メモリー
2）バス，データバス，アドレスバス
3）制御部，データ処理部
4）制御線，フラグ
5）ALU，シフタ，I/O
6）フラグレジスタ，リテラル
7）RAM，ROM

問題 7.2　装置側から見ると，バスへデータを出力するスイッチとバスからデータを入力するスイッチがあり，これらのいずれかを ON にするか，いずれも OFF にするかの三つの状態が必要となる。バスに三つの装置 A，B，C がついているとして，A から B に信号を送る場合に各装置がどのような状態になければいけないかを考察してみよ。

8 プログラム

《目標&ポイント》コンピュータのしくみを理解する前に，コンピュータを自由に動かすためのプログラムについて，特にコンピュータとの関わりを中心に学ぼう。

《キーワード》プログラム，プログラミング，機械語，アセンブラ言語，高水準プログラム言語，コンパイラ，インタプリタ，演算命令，移動命令，ジャンプ命令，逐次プログラム，分岐，ジャンプ，無条件ジャンプ，条件ジャンプ，応用ソフトウェア

8.1 プログラム

現在のコンピュータは主として，ワープロ，表計算などに利用されている。これらは，**応用ソフトウェア (application software)** を起動することにより，実行がなされている。もう少し，コンピュータに堪能な人は，C とか Java とかいった**高水準プログラム言語 (high level program language)** と呼ばれる文章を**プログラミング (programming)** することにより，自分や他人に必要とされる**プログラム (program)** を作成し，それをコンパイルして応用ソフトウェアを作成し，それを実行しているかも知れない。コンパイルという用語を知らない人は，すぐ後に説明するので，それを読んで欲しい。ワープロ，表計算などといった応用ソフトウェアも，実はソフトウェアメーカーが高水準プログラム言語を用いてプログラミングし，コンパイルしたものを動作させているのである。

こうした応用ソフトウェアと呼ばれるものは，すべて，**機械語 (machine language)** と呼ばれるコンピュータの動作を直接制御する**命令 (instruction)** により構成されている。一見意味のない 0 と 1 のビット列に見えるので **2 進プログラム (binary program)** とも呼ばれる。C とか Java を高水準プログラム言語と言ったが，ほとんど機械語そのものに対応した**アセンブラ言語 (assembler language)** と呼ばれるものがあり，それとの対比で高水準と呼ぶのである。高水準プログラム言語で書かれたプログラムもその内容を解釈され，機械語に変換される。こうした翻訳を行う言語を**コンパイラ (compiler)** という。言語によっては翻訳をしないで，実行時に行単位で機械語に変換され，実行されるものもある。こうした通訳を行う言語を**インタプリタ (interpreter)** と呼ぶ。いずれにせよ，実行時には機械語になっているのである。

8.2 命令の種類

高水準プログラム言語では，1 行の中にいろいろな作業を含ませることが可能であるが，機械語の命令のレベルになると，各命令は単純なことしかできない。そもそも，コンピュータは計算機械から出発し，その構造も前章に示したように，データ処理部と呼ばれる計算に特化した部分と，それを動かす制御部からなっていることからわかるように，大部分の命令は**演算命令 (arithmetic instruction)** と呼ばれる計算のためのものである。

四則演算の手続きから想像できるものに加え，次のようなものが考えられる。なお，下記文章で，**レジスタ (register)** とは CPU 内部の一時メモリーのことである。

- 一つのレジスタの内容の NOT

- 一つのレジスタの内容に 1 を加える。および 1 を減ずる
- 一つのレジスタの内容の符号反転 (補数をとる)
- 一つのレジスタの内容の任意の数のビットシフト
- 二つのレジスタの内容のビットごとの AND，OR，EOR など
- 二つのレジスタの内容の加算，減算。また LSB にキャリーの入った加減算
- シフトを伴う加減算

こうしたレジスタの内容は，元をたどると，周辺装置から持ってくることが多い。また，各計算過程における中間結果はレジスタ上に置かれるが，最終的には周辺装置に移動され，初めてユーザの目に触れることになる。

より複雑な計算で，中間結果が極めて多くなると，それらデータを外部に置かれたメモリーに置き，それを必要に応じ，レジスタに読み込んで計算を行う。あるいは，表計算のようなものでは，周辺装置の一つであるハードディスクから表全体をメモリーへ移動し，そこから必要な部分だけをレジスタへ移動して計算し，計算結果を元のメモリー上の表へ戻すことが行われる。いずれにせよ，周辺装置を含め，外部メモリーとレジスタ間のデータのやり取りが必要である。そこで，外部メモリーや周辺装置とのデータの移動を目的として，**移動命令 (move instruction)** と呼ばれる次のような命令が必要である。

- 外部メモリー (周辺ユニットも含む) の指定したアドレスから指定したレジスタへのデータの読み込み
- レジスタから外部メモリー (周辺ユニットも含む) の指定したアドレスへのデータの書き出し

CPU によっては，これら二つのカテゴリーの命令の混ざったものを実行する命令を備えたものもある。例えば，レジスタの内容と外部メモリー

上のデータを直接加算する命令などである。本書では，外部メモリーとのやり取りは，レジスタとの移動に限り，各種算術演算はレジスタ間でしか行わないこととした。

これだけの命令で基本的な動作はできそうであるが，多くのプログラムでは**ジャンプ命令 (jump instruction)**[1] というものを多用する。機械語で書かれた命令は，通常メモリー上に置かれる。それも原理的には，メモリーの 1 アドレスに 1 命令が書かれ，さらに，これらの命令は低アドレス側から高アドレス側に順に実行されると考えてよい。このように，アドレスの順に実行されるプログラムのことを**逐次プログラム (sequential program)** という。

しかし，メモリー上には命令とデータが混載されていることが多い。したがって，命令の書かれた領域がいくつかに分かれているときなどには，命令領域の最後に，次の命令の書かれている領域にジャンプできるようなジャンプ命令を記載することが必要となる。また，条件によって，実行することを変えたいときにもジャンプ命令が必要である。さらには，いくつかのデータの総和を計算するようなとき，メモリーからデータをレジスタに移動することと，その内容を次から次に合計値に加算していくといった同じ動作の繰り返しが必要となる。こうしたときにも繰り返しを行うためのジャンプ命令が必要である。この場合，放っておくと，永久に同じ命令領域を繰り返すことになるので，何らかのきっかけで，ジャンプを停止し，繰り返しを終了することが必要となる。こうした条件によりジャンプ先が異なるようなジャンプ命令は**分岐 (branch)** 命令とも呼ばれる。

本書では，ジャンプ先が常に決まっているジャンプを**無条件ジャンプ**

1) ジャンプ命令には HLT など，必ずしもジャンプするとは限らないものも含まれているため，制御命令と呼ばれることもある。

(unconditional jump)，条件によりジャンプ先が異なるジャンプを**条件ジャンプ (conditional jump)** と呼ぼう。条件ジャンプとは，条件が真の場合は次のアドレスを実行するが，偽の場合には指定されたアドレスに**ジャンプ (jump)** する。このような条件ジャンプの条件は，**図** 7.3 に示した ALU からの**フラグ (flag)** を利用する。

例えば 1 から i までの連続した整数の合計を計算するようなときは，次のように，条件ジャンプを利用する。

1） あるレジスタ A を累計用とし，それに 0 を入れる。
2） 別のあるレジスタ B に最大数 i を入れる。
3） レジスタ A と B の和を計算し，結果を A に入れる。
4） B の内容を 1 減らす。B が 0 でなければ 1 行前へジャンプする。
5） A は総和となる。

この下から 2 行目が条件ジャンプであり，非ゼロフラグ (結果が 0 でないときに立つフラグ) を検知してジャンプする。フラグが 0 のときには，次の行へ移動する。

8.3　高水準プログラム言語における分岐/ジャンプ

本節は高水準プログラム言語をある程度知っている人を対象に，高水準プログラム言語における分岐/ジャンプ文が，機械語の分岐/ジャンプ命令とどのように関わっているかを示すために記載した。したがって，興味のない人は読み飛ばしてもらって構わない。

高水準プログラム言語にも，if 文，while 文，do 文などと呼ばれるいくつかの分岐/ジャンプを伴うプログラム用の文がある。

図 8.1 に見られるように，これらはいずれも条件判定を行う文と，その条件により実行される文がある。

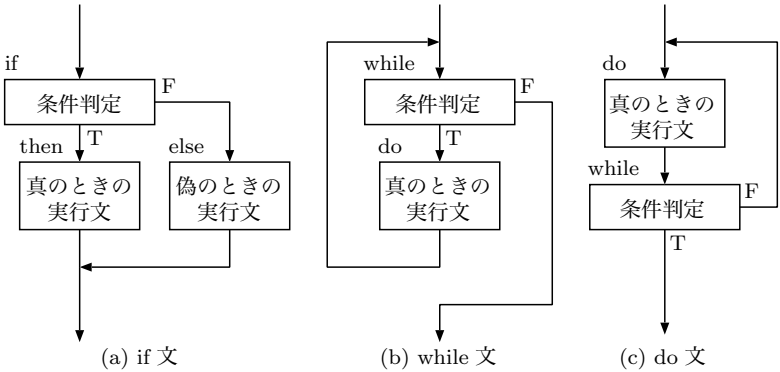


図 8.1　高水準プログラム言語に見られる分岐/ジャンプ文の例

(a) の if 文は if に続く条件式に記載された条件を調べ，それが真ならば then 以下に書かれた実行文 (一般には複数文) を，偽ならば else 以下に書かれた実行文 (一般には複数文) を実行する。
(b) の while 文は while に続く条件式に記載された条件を調べ，それが真のうちは do 以下に書かれた実行文を実行する。
(c) の do 文は，まず do に続く実行文を 1 回実行し，続く until に続く条件式を調べ，それが真となるまでは do 以下に書かれた実行文を実行する。

この他，else 文のない if 文，主として回数を指定して同じことを実行する for 文，条件が真のうちは do 以下を実行する do 文 (until が while になっている文) などもあるが，いずれもここに示した三つの典型例と本質的には変わらない。

こうした条件判定により，分岐したり反復したりする文でも，**図** 8.2 に示すように，メモリー上には 1 次元的に展開されている。これらの命令群が**図** 8.1 と同じ動作をすることはすぐに確認できるであろう。このメモリー上に展開された機械語のプログラムを見ると，ほとんど逐次プ

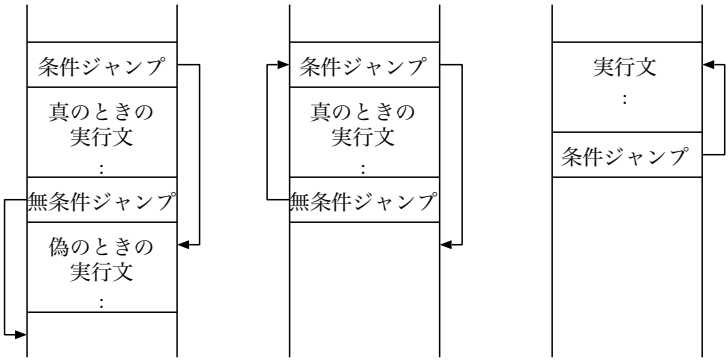


図 8.2　メモリー上に展開された分岐/ジャンプ文

ログラムとなっているが，例外的に**条件ジャンプ (conditional jump)** と**無条件ジャンプ (unconditional jump)** があることがわかろう。

8.4　機械語の命令セットと命令の実行

さて，コンピュータに必要な機械語の命令には，具体的にどんなものが必要であろうか。本章で今まで述べてきたように，大きく分けて三つのカテゴリーが必要である。さらに各カテゴリーに属する代表的な命令を，**図** 8.3 に記述した。ただし，必ずしもこのすべてが必要ではないし，これ以外の命令があっても大きな問題があるわけでもない。あくまでも，命令セットの一例であると理解して欲しいが，本書ではこれらの命令をセットとして持つコンピュータを前提とする。

実際の機械語命令は，**命令コード (instruction code)** と呼ばれる 2 進表現されたものであり，基本的には外部メモリーに置かれ，CPU はこれを順に読み取りながら，作業を実行していくことになる。命令コードもコードの一種であり，厳密には第 6 章のコードの説明に含めるべきで

命令	説明
移動命令	メモリーや周辺装置とのデータのやり取り
LD i, n	メモリーの n 番地から Reg. i 番へロード
ST i, n	Reg. i からメモリーの n 番地へストア
演算命令	算術や論理演算を行う
MOV i, k	Reg. i の内容を Reg. k に入れる
NEG i, k	Reg. i の補数を Reg. k に入れる
ADD1 i, k	Reg. i に 1 を加えた結果を Reg. k に入れる
SUB1 i, k	Reg. i から 1 を引いた結果を Reg. k に入れる
ADD i, j, k	Reg. i と Reg. j の加算結果を Reg. k に入れる
SUB i, j, k	Reg. i から Reg. j を減算した結果を Reg. k に入れる
SHL i, k, n	Reg. i を n ビット左シフトした結果を Reg. k に入れる
SHR i, k, n	Reg. i を n ビット右シフトした結果を Reg. k に入れる
NOT i, k	Reg. i の各ビットの NOT を Reg. k に入れる
AND i, j, k	Reg. i と Reg. j のビットごとの AND をとり， 結果を Reg. k に入れる
OR i, j, k	Reg. i と Reg. j のビットごとの OR をとり， 結果を Reg. k に入れる
EOR i, j, k	Reg. i と Reg. j のビットごとの EOR をとり， 結果を Reg. k に入れる
ジャンプ命令	無条件ジャンプ，条件ジャンプなど
JP n	n 番地へ飛ぶ (次の実行命令をメモリーの n 番地とする)
JPZ n	零フラグが立っていればメモリーの n 番地へジャンプ， そうでないときには次の番地へ移動
JPN n	負フラグが立っていれば n 番地へジャンプ
JPC n	MSB より上位へキャリーがあれば n 番地へジャンプ
JPO n	オーバフロー (符号あり整数) ならば n 番地へジャンプ
HLT	動作を停止する (自分自身のアドレスへジャンプする)

図 8.3　命令セットの例 (Reg. はレジスタの略)

あったかも知れないが，説明の都合上本章で行う。

命令コードのデータ幅は，本書ではメモリーの幅が 16bit であることから，基本的には 16bit の整数倍とする。それも可能な限り 16bit とし，やむをえない場合のみメモリー 2 語分，つまり 32bit としたい。実は，データ処理部を制御するにはもっと多くのビット数の制御線が必要である。しかし，レジスタの指定等，オプションが異なるものをすべて異なる命令と見ても，その総数は $2^{16} = 65536$ 種類もないのである。したがって適切なるコーディング，つまりビット割り当てを行えば，上記の条件を満たすことは可能なのである。

これらの命令のうちで，移動命令やジャンプ命令は，オプションとしてメモリー上の 16bit のアドレスを指定する必要があるので，どうやっても命令幅 16bit では収容できず，32bit を使うことにする。その他の命令は 16bit に収容することを試みよう。オプションであるレジスタの選択やシフト量の指定は，それぞれ 4bit ずつ必要であるので，オプションの多いものは，命令そのものに使えるビット数は少ない。一方，オプションの少ないものは，命令そのものに使えるビット数は多い。そこで次のような戦略でコーディングの割り当てを考えることとする。

オプション (レジスタ指定など) の多い命令から順に並べ，3 オプションあるものは命令の区別に 4bit を用い，2 オプションあるものは命令の区別に 8bit を用い，1 オプションあるものは命令の区別に 12bit を用い，0 オプションあるものは命令の区別に 16bit を用いればよい。アドレス指定をするものは，前述のように，32bit 幅を必要とするが，アドレス指定の 16bit を除く部分の 16bit については，通常の同じような手法を適用する。こうして得られた命令コード割り当ての一例を，**図 8.4** に示す。なお，将来の拡張性を考慮し，ところどころ余裕を持たせて割り当てを行った。

命令	命令コード			
3 オプション命令				
ADD i, j, k	0000	(i)	(j)	(k)
SUB i, j, k	0001	(i)	(j)	(k)
AND i, j, k	0100	(i)	(j)	(k)
OR i, j, k	0101	(i)	(j)	(k)
EOR i, j, k	0110	(i)	(j)	(k)
SHL i, k, n	1000	(i)	(k)	(n)
SHR i, k, n	1001	(i)	(k)	(n)
2 オプション命令				
MOV i, k	1100	0000	(i)	(k)
NEG i, k	1100	0001	(i)	(k)
ADD1 i, k	1100	0010	(i)	(k)
SUB1 i, k	1100	0011	(i)	(k)
NOT i, k	1100	0100	(i)	(k)
2 オプション命令	2 語命令			
LD i, n	1110	0000	0000	(i)
	(n)			
ST i, n	1110	0000	0001	(i)
	(n)			
1 オプション命令	2 語命令			
JP n	1111	0000	0000	0000
	(n)			
JPZ n	1111	0000	0000	0001
	(n)			
JPN n	1111	0000	0000	0010
	(n)			
JPC n	1111	0000	0000	0011
	(n)			
JPO n	1111	0000	0000	0100
	(n)			
0 オプション命令				
HLT	1111	1111	1111	1111

図 8.4　命令コードの例 (括弧内は対応レジスタや数を 4bit で表したもの)

命令を実行する際には，まずメモリーから命令をもらってくる**フェッチ (fetch)**[2] と呼ぶ作業が必要である。これは次のようにして行う。

1) メモリー上で次の命令が入っているアドレスを記憶している**プログラムカウンタ (program counter)** または **PC** と呼ばれるレジスタ (本書では普通のレジスタの一つを利用する) の内容を，CPU 内部のデータバスを経由してアドレスバスに流す。同時に *ReadMemory* 信号 (メモリーに読み込みをしたいことを伝える) を 1 にする。

2) *MemoryEnable* 信号 (メモリーがデータをデータバスに載せたことを CPU に伝える) が 1 になるのを待ち，1 になったら次のステップへ移動する。

3) データバス上のデータを**命令レジスタ (instruction register)** または **IR** と呼ばれるレジスタ (本書では普通のレジスタの一つを利用する) へ入れる。

4) ほとんどの場合，次の命令はメモリー上の次のアドレスに入っているので，あらかじめ，PC の値を 1 増やしておく。

5) ここでいったん終了し，メモリーの実行ステップへ移動する。

このフェッチの作業によりメモリー上の命令を持ってきて，それに引き続き命令レジスタ上の命令を実行することになる。

例えば，条件ジャンプは次のようにして実行される。条件ジャンプ命令では，どのフラグを見てジャンプすべきか，ジャンプする場合には何番地へジャンプするかの指示がある。ここでは，ジャンプ先の情報はメモリー上，次のアドレスに記載されているものとする。

1) フラグを見て条件不成立ならば，PC を 1 増やして，フェッチの作業に戻る。

---

2) fetch とは取り込みのこと。

2） 条件成立ならば，フェッチの作業と同じような手法で，メモリー上の次のアドレスの命令を PC に取り込む。

3） フェッチの作業に戻る。

他の命令もおおよそこれらの例のようにして実行される。

8.5 蓄積プログラム方式

これらの命令の実行手順を見ると，シークエンス回路の議論を思い起こすかも知れない。実際，制御ユニットの中心部分はまさにシークエンス回路なのである。

今まで，データと命令は同じメモリー上に置かれていると，気楽に記述してきたが，黎明期のコンピュータでは，**データ (data)** はメモリー上に置かれたが，コンピュータに行わせる作業手順はすべてシークエンス回路として，回路の構造に組み込まれていた。これらの**命令 (instruction)** を，書き換え可能なメモリー上に置こうというのは一つの大発明であった。さらにこの際，命令もデータも同じメモリー上に混載されるようになった。これが**蓄積プログラム方式 (stored program concept)** という概念である [3)] 。

蓄積プログラム方式では，命令もデータも同じ外部メモリー上の連続した領域に記憶されるが，一番最初に実行される命令だけはメモリーの特定の位置に置かれている必要がある。その他の命令やデータはメモリー上の比較的自由な場所に置くことが可能である。次の命令がメモリー上，離れたところにある場合には，ジャンプ命令を使う。また，メモリー上の任意の場所にあるデータを操作 (ロード，ストア) する命令が用意されている。こうした機能を利用して，命令とデータの混載が可能となった

3） ニューマン (ノイマン) 型コンピュータとも呼ばれる。

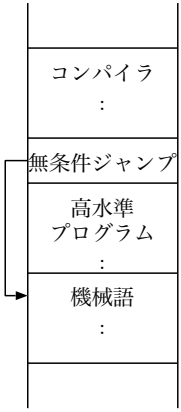


図 8.5　蓄積プログラム方式により，高水準プログラムを実行する概念図 (コンパイラの出力データだった機械語にジャンプし，それらを命令として実行する。)

のである。

この機能は命令群のフレキシビリティを高めただけでなく，高級言語の出現という驚くべき効果も産み，利用者にやさしいプログラム環境へと繋がり，現在のコンピュータの繁栄を生み出したのである。

高級言語とは，人間にとって読みやすいテキストで書かれた命令書である。このテキストはコンパイラにとっては，テキストデータである。コンパイラは，**図** 8.5 に示すように，テキストデータを翻訳して，機械語である 2 進データとして出力するのである。そして，その先頭アドレスにジャンプすることにより，応用プログラムを実行することができるようになったのである。まさに，あるときはデータとして扱い，あるときには命令として扱うという器用な手法が確立したのである。

演習問題 8

問題 8.1 次の用語を理解したかどうか確認せよ。

1) 機械語，アセンブラ言語，高水準プログラム言語
2) コンパイラ，インタプリタ
3) 演算命令，移動命令，ジャンプ命令
4) 逐次プログラム
5) プログラムカウンタ，命令レジスタ
6) 蓄積プログラム方式

問題 8.2 次の各プログラムを実行するには，上記のどの分岐/ジャンプ文を使うのがよいだろうか。(この問題は高水準プログラムを知っている人のみが対象である。)

1) メモリー上に 1 文字 1 アドレスを使って書かれた文字列がある。この文字列は `0x0000` を終了記号としている。これを順に読んでいき，メモリー上の別の領域にコピーしていきたい。ただし，終了記号までをコピーする。
2) 数値データを 1 個読み込み，それが負ならば符号反転したものを，正ならばそのままの値を出力したい。
3) 数値データを 1 行ずつ読んでいき，その合計を計算したい。数値が正の間は実行するが，0 または負になったところで計算を終了する。

9 データ処理部

《目標＆ポイント》CPU を構成する二つのユニットのうち，データ処理部に入っている種々の回路群の詳細を順に説明していく。データ処理部は CPU 全体の性能を決定するため，もっとも多くの工夫がされているところであるが，具体的イメージを作るため，回路群にはそれぞれ一例を示して解説する。

《キーワード》データ処理部，バス，クロック，制御線，レジスタ回路，シフタ，算術論理回路 (ALU)，キャリー，キャリー伝播時間，制御コード

9.1 データ処理部とタイミング

CPU を構成する二つのユニットのうち，電卓のようなデータの処理を受け持っている**データ処理部 (data processing unit)** には，レジスタ，シフタ，算術論理回路 (ALU) の三つの機能回路が搭載されている。レジスタは一種のシークエンス回路であるが，パストランジスタと NAND 二つで構成される簡単なものである。シフタは一種の論理回路とも言えるが，パストランジスタだけで構成することが可能である。ALU は入口や出口にレジスタを持つこともあるが，重要な部分は組合せ論理回路である。本書の場合には，第 5.6 節で述べたプリチャージ論理回路とパストランジスタを中心とした回路で実現する。なお，データ幅は 16bit とするが，回路例などを示すときには，適宜，小さな幅のものを示すこととする。

データ処理部の詳細を説明する前に，データの移動や処理がどのクロッ

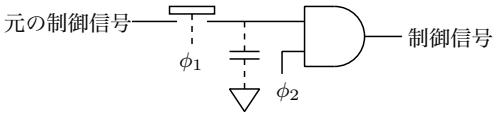


図 9.1　ϕ_1 の制御信号を ϕ_2 で送出する回路

クタイミングで行われるかを示しておこう。まず大原則として，すべてのデータの移動や処理は ϕ_1 のタイミングで行う。データ処理部内におけるデータの移動以外にも，制御データや，CPU と外部のメモリーや周辺装置とのデータも，原則として ϕ_1 で移動するものとする。

ϕ_2 のタイミングは入出力のループが起きないよう，パナマ運河方式をとるための，いわば交通整理のためやむをえず導入する。しかし，ループを作らないところならば，ϕ_2 のタイミングで処理をしてもよいはずである。

データ処理部を注意深く見てみると，ALU には前後にレジスタが付いており，ALU の処理を ϕ_2 のタイミングで行っても，何ら問題がないことがわかる。したがって，最初の ϕ_1 で入力レジスタにデータを入れ，続く ϕ_2 で算術演算や論理演算を行って出力レジスタにデータを格納し，次の ϕ_1 で出力レジスタからデータを出せば，1 クロック周期の節約ができることになる。

こうした ϕ_1 のタイミングを待たずに処理することにより，高速化を果たしている。さらに，どの命令も 1 クロック周期で実行できることになるため，後に述べるパイプライン処理がやりやすくなるメリットもある。

このような半サイクル後に命令を実行するには，最初の ϕ_1 のタイミングで ϕ_2 における制御信号も用意しておき，**図** 9.1 に示す遅延回路の半分の回路を利用して，半サイクルずらして制御線に載せればよい。

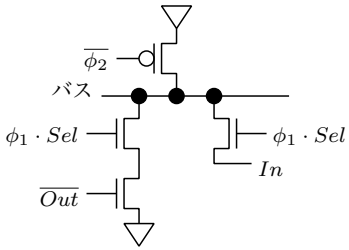


図 9.2　機能ブロックとバスとの結合回路

9.2　バス

データ処理部内の各機能ブロック間のデータの伝達に用いられているデータバスとのアクセスの仕方について述べよう。

これには三つの関係の仕方が存在する。データを送る，受ける，それと無関係の三状態である。これをプリチャージ回路で実現するには，**図** 9.2 のような回路にする必要がある。機能ブロックへの入力は簡単であり，Sel で選択するだけでよい。出力をバスに出すには ϕ_1 のタイミングで $\overline{Out}$ によって，電位を引き下げればよい。この出力をバスに接続するかどうかは，Sel(Select) で決定される。このように出力として 0/1 および無関係の三状態を選べる回路を**トライステートバッファ (tri-state buffer)** と呼ぶ。

シフタなどの回路は，バスからデータをもらって，同じバスの異なる線にデータを返す。このため，出力データをいったん蓄えて，次のタイミングで出力するような配慮が必要である。このためには，1 クロック遅延を与える遅延回路，もしくは何らかのレジスタが必要となる。

ALU には同時に最大 2 入力が必要である。このため，バスの本数が少ないと，ALU 側にいくつかのレジスタを置くなどの配慮が必要となる。

バスを多くするのは，速度も速くなり，一見楽のようであるが，集積回路のチップの面積は限られており，バスをどんどん増やすのは常に得策とは限らない。要するに，何を重視し，何を犠牲にするかの問題となる。こうした，回路の大まかな構造を**アーキテクチャ (architecture)** という。本書ではバス A とバス B の 2 バスを用いよう。なお，本書の ALU は演算のタイミング (ϕ_2) とデータ移動のタイミング (ϕ_1) が異なるため，2 バスシステムでも，ALU は入力側にも出力側にもレジスタを必要とする。

9.3 レジスタ

レジスタ (register) とはデータを一時的に蓄えておくメモリーのことである。各バスとはスイッチを経由して入出力の授受ができるようになっているが，バスの線間の情報のやりとりはしないので，線ごとにデータのビット幅だけのメモリーを配置すればよい。また，当然メモリーの内容は書き換え可能でなければならない。

図 9.3 に 1bit 分のレジスタ回路の構成を示す。基本的には前に示した **図** 5.8 の簡易型と同じであるが，2 バスに接続可能なことだけが異なる。この回路をデータ幅の個数だけ用意したものをレジスタ 1 個とする。簡易型のレジスタを採用したのは，回路規模が小さくなることに加え，読み込んだデータを同じバスへ戻すことはないこと，A バスから B バスへ直ちに移動するような作業も滅多にないことによる。

この内容を書き換えるには，ϕ_1 のクロックでいずれかのバスの値に設定する。このとき，$\phi_2 = 0$ のため，ループは開いているので，書き換えは容易である。また，レジスタの内容をバスに出力したいときには，同じく ϕ_1 のタイミングで出力側のスイッチをバスに接続する。つまり，ϕ_1 のタイミングはデータの移動など主たる動作に充てられている。ϕ_2 はリ

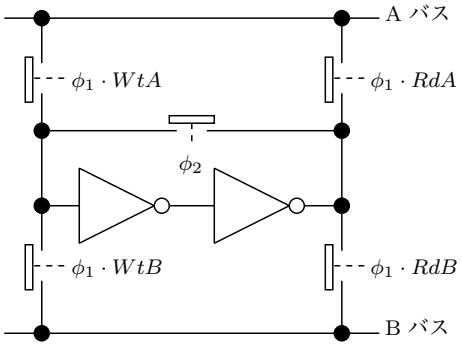


図 9.3　レジスタ回路 (Wt，Rd は，各バスからの書込み，読み出し)

フレッシュに用いられる。

レジスタは多いほど便利がよいが，その分チップ面積をとるので，数個に抑える。この例の場合には 16 個のレジスタを考えよう。データ幅を 16bit とすると，上記の回路と同じものが $16 \times 16 = 256$ あることになる。

なお，ALU の入力レジスタは，データをバスから ϕ_1 で受領して，ϕ_2 で ALU 本体に渡さねばならない。この場合には，**図** 9.3 の右側の $\phi_1 \cdot RdA$ などを外したものを利用する。ϕ_2 で動くゲートは ALU 出力レジスタ側に用意することにする。ALU は，この出力の NOT も必要とするが，それは二つの NOT の間から出力を取り出すことで達成する。同様に ALU の出力レジスタは，データを ALU より ϕ_2 で受領して，ϕ_1 でバスに渡さねばならない。この場合には，同図左側の $\phi_1 \cdot WtA$ などを一つにして ALU 本体出力に接続し，$\phi_2 \cdot Wt$ に置き換え，さらに中央の ϕ_2 を ϕ_1 に変更することで達成する。

9.4 制御線

制御線については，どの図でもあまり詳細を示していない。それはあまりにも本数が多いからである。しかし，いろいろな方法で，その本数を減らすことも可能である。

まず，レジスタを見てみよう。レジスタは 1bit メモリー 16 個が組になって，動作する。書き込みも読み出しも 16 個を一緒に動作させてよい。したがって，書き込み制御線も読み出し制御線もまとめて 1 本でよい。レジスタはこのような 16bit のものが全部で 16 個あるので，書き込み制御線も読み出し制御線もそれぞれ計 16 本でよい。このように，他の機能ブロックでも，16bit 分のゲートを同時にまとめて動作させてよいものは，すべてまとめることにする。

さて，レジスタを一度に複数選択することは少ない。多くても，入力用に 1 個，出力用に 1 個といった具合であろう。このような場合，どのレジスタを選ぶかだけを指定すればよい。16 個のいずれかを選ぶには，0 から 15 を 2 進表現して 0000 から 1111 を指定すればよい。つまり制御部からデータ処理部に対し，4bit の情報を指定するだけでよい。入力用と出力用を同時に指定しても，8bit で十分である。もちろん，レジスタのすぐ傍で，このレジスタ番号をデコードして，16 本の線のいずれかを選択できるようにする必要があるが，制御線としてはかなりの本数節減が果たせることになる。同様に，シフタのシフト量も 0 から 15 であるので，2 進化すれば 4bit で可能である。

その他，後述する ALU では K，P，R の動作決定のためそれぞれに 4 本ずつの計 12，入力レジスタには読み込み先に対応してそれぞれ 4(バス 2，シフタ出力 1，シフト量 1) あるが，2 進化できるので 3 となり 2 台で計 6。フラグ関連の制御信号などを加えるとおよそ二十数本となる。二

つの出力レジスタはそれぞれ 2 の計 4 であるが，これらは汎用レジスタなどとのデータ移動命令側で設定することにする。

合計して 50 本程度となるが，パイプライン処理をすれば，二つのタイミングで ALU 情報と移動情報を分けて送れるので，二十数本でよいことになる。それにしてもかなりの本数となる。詳細は，本章第 9.7 節で述べる。 将来，制御部の設計の際に，この多くの制御線が必要であるという事実が重い意味を持ってくる。

9.5　シフタ

シフタは乗除算や論理演算などでしばしば使われる。シフトには，1bit だけシフトするものもあるが，本書では何ビットでもシフトできる**バレルシフタ (barrel shifter)** と呼ばれるものを扱う。シフトして空いたところへ何を埋めるかによって，いくつかの種類がある。空いたところへ 0 を埋める場合もあれば，回転シフトと言って，溢れたビットを，逆から埋めていく場合もある。こうしたことを考慮し，本書に紹介するシフタは，基本的には A バスの内容をシフトするが，下から B バスの内容を埋めていくものとする。つまり，A バスと B バスを繋いでシフトするのがよい。もう少し正確にいうと，A バスと B バスの内容を並べて 2word とし，その任意の位置から 1word を切り出して，出力する機能を有するようにする。B バスに 0 を入れておけば，シフトの結果空いたところには 0 がつまり，B バスに A バスと同じ内容を入れておけば，回転シフトとなる。

こうしたことを考慮し，**図 9.4** のような**シフタ (shifter)** が完成する。なお，16bit バスでは図が大き過ぎるので，4bit バスを仮定した。実際，ShL_0，ShL_1，ShL_2，ShL_3(ShL は shift left の意味) を順に 1 に

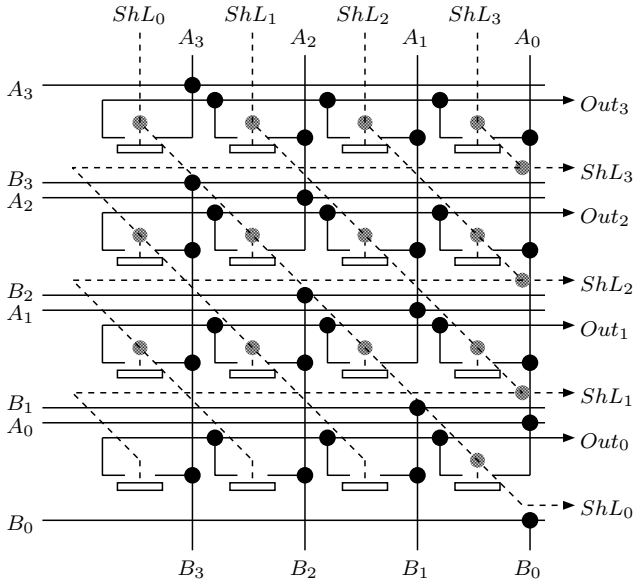


図 9.4　シフタ (見やすいように制御用関連の配線は破線で示した)

すると，$(Out_3, Out_2, Out_1, Out_0)$ が (A_3, A_2, A_1, A_0)，(A_2, A_1, A_0, B_3)，(A_1, A_0, B_3, B_2)，(A_0, B_3, B_2, B_1) と変化していく。

なお，乗除算では 1bit のシフトしかないが，シフト溢れを利用することがある。これは実は，シフト前の MSB または LSB であるので，簡単な作業で取り出すことができ，S_{out} という形で記憶しておく。

9.6　算術論理回路 ALU

算術論理回路 (arithmetic logic unit) または **ALU** はもっとも設計の難しい回路である。まず，キャリーのある加算ができなければならない。補数も作られなければならない。ビットごとの AND や OR といったビット演算もできるとありがたい。数値の比較もできるとよい。とい

うことで，単なる加算器の集合よりも，高機能の演算ができる回路を考えよう。

しかし，いきなり高機能の回路と言われても考えづらいと思われるので，まずキャリーに着目して，加算器の改良から取り掛かろう。加算器の問題は何桁にも及ぶキャリーの伝播時間が馬鹿にならないことである。同期式回路の場合，すべての論理処理が終了するまで，次のクロックパルスを送れないから，実はこのキャリー伝播時間がコンピュータ全体の速度を決めてしまうのである。

比較的簡単な回路で，かつ，比較的速い動作をするキャリーの計算法として**マンチェスタキャリーチェーン (Manchester carry chain)** という方式が提案されている。これは，**図 4.5** に示した全加算器の真理値表を見てみると，ほとんどの欄で C_o と C_i が等しいという事実を利用した方式である。この結果，多くの場合，各桁ではキャリー C_o を改めて計算することなく，C_i をそのまま後段に伝播すればよいことになる。

式できちんと確認しておこう。まず，下からのキャリーのない場合には次式が成立する。

$$S = A \oplus B \tag{9.1}$$

$$C_o = A \cdot B \tag{9.2}$$

しかし一般には下の桁からのキャリー C_i があるので，次式が成立する。

$$P = A \oplus B \tag{9.3}$$

$$S = P \oplus C_i \tag{9.4}$$

$$\begin{aligned} C_o &= A \cdot B + (A + B) \cdot C_i \\ &= \overline{P} \cdot A \cdot B + P \cdot C_i \end{aligned} \tag{9.5}$$

上の 2 式は，A と B の算術和 1 桁目をとりあえず P とし，それにさらに

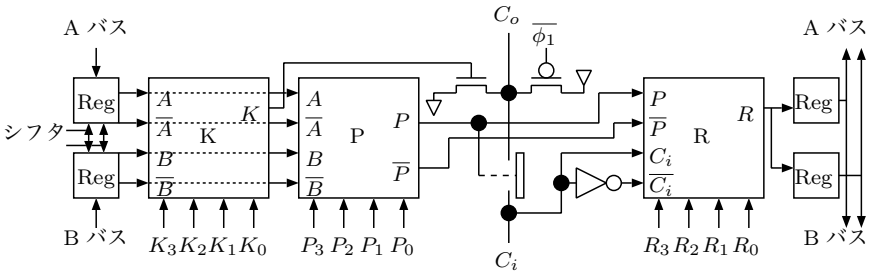


図 9.5　ALU(シフタから入る入力にはシフト結果とシフト量の選択が可。K と P の出力には $\overline{\phi_1}$ との AND が入る。)

C_i を加えたものを S とし直すというので直感的である。キャリーの式は，下からのキャリー C_i が 1 のときは，A または B のいずれか，または双方が 1 ならば上へのキャリーが 1 となり，また C_i が 0 でも，A および B が 1 ならば 1 となることを示している。さらに，$A+B$ を $P=A\oplus B$ に置き換え，前半に $\overline{P}$ を付けても，全体の結果が変わらないことを示している (各自で確認せよ)。さらに，前の項は $\overline{P}(A+B)$ としても構わない。

$P=1$ であるとキャリーはそのまま上に伝えてよいことになるので，P を作り出す機能は**伝達子 (propagator)**P と呼ばれる。伝達がない場合には，$A\cdot B$ に基づき新たなキャリーを発生することになるが，この機能を**生成子 (generator)**G と呼ぶ。さらに，この否定を**消滅子 (killer)**K と呼ぶ。消滅子はプリチャージ論理の際, 使われる。S は P と C_i の EOR であるが，この論理を行う部分を R と記載しよう。

図 9.5 に 1bit 分の ALU の概念図を示すが，ここで K，P，R を固定の論理を扱うブロックとはしないで，**図** 5.12 に示した任意の論理関数を実現できる回路に置き換えると，極めてたくさんの演算を行うことができるようになるのである。

C_o は ϕ_1 でのプリチャージ論理により決定される。ϕ_2 になって初めて，

K によりその電位を下げるかどうかが決まる。これに対応して，K と P の機能ブロックの出力には $\overline{\phi_1}$ との AND ゲートを置くことにより，このプリチャージ論理との整合性をとっている。

これらの機能ブロックに送る信号 K，P，R をいろいろ変化させたときに実現できる機能の例を，**図** 9.6 に示す。先に第 8 章プログラムの **図** 8.3 で示した機械語の命令セットの演算命令と比較し，やや多めの機能が示されている。例えば，C_{in} を ALU 全体に対するキャリーインとして，$A+B+C_{in}$，$A-B-C_{in}$ などの加減算は，データ幅以上の加減算の際，必要な命令である。

ALU は全体として，その動作結果に応じた**フラグ (flag)** を出す。これらフラグは，ALU 全体の C_{out}(キャリーアウト)，シフトの際の S_{out}(シフト溢れ)，オーバフロー，減算の結果の正負などであるが，**図** 9.7 に示す**フラグレジスタ (flag register)** と呼ばれる 16bit の一時メモリーへ蓄積され，A バスを経由して，他のレジスタや ALU などで利用することができる。これらのうち，4，5，6bit 目は演算で扱う数を符号あり整数とみなした場合の結果を示す。第 6 章で述べたように，符号なし整数のオーバフローは C_{out} で判定できるため，ここで示すオーバフローフラグとは，符号あり整数のオーバフローを検出するものである。

フラグレジスタの MSB には，**フラグビット (flag bit)** と呼ばれる特別の役割が与えられている。この特定のビットは，上記 0 から 7bit 目までに記載されたフラグのうち，いずれかが選ばれ，そのコピーが記録される。どのビットが選ばれるかは，制御部から与えられる当該演算の指令の中に記載される。このフラグビットは，条件分岐の際利用されたり，これから述べる ALU の条件付演算命令に利用される。**図** 9.6 中，ALU 全体へのキャリーイン C_{in} の欄に書かれた FB とは，このフラグビットを意味する。フラグレジスタの 0bit 目には，1 回前のフラグビットも蓄

	K	P	R	C_{in}
Zero	$0000:0$	$0000:0$	$0000:0$	0
A	$0000:0$	$1100:A$	$1100:P$	0
B	$0000:0$	$1010:B$	$1100:P$	0
$\overline{A}$	$0000:0$	$0011:\overline{A}$	$1100:P$	0
$\overline{B}$	$0000:0$	$0101:\overline{B}$	$1100:P$	0
$-A$	$1100:A$	$0011:\overline{A}$	$0110:P\oplus C_i$	1
$-B$	$1010:B$	$0101:\overline{B}$	$0110:P\oplus C_i$	1
$A+1$	$0011:\overline{A}$	$1100:A$	$0110:P\oplus C_i$	1
$B+1$	$0101:\overline{B}$	$1010:B$	$0110:P\oplus C_i$	1
$A-1$	$1100:A$	$0011:\overline{A}$	$1001:\overline{P\oplus C_i}$	1
$B-1$	$1010:B$	$0101:\overline{B}$	$1001:\overline{P\oplus C_i}$	1
$A\cap B$	$0000:0$	$1000:A\cap B$	$1100:P$	0
$A\cup B$	$0000:0$	$1110:A\cup B$	$1100:P$	0
$A\oplus B$	$0000:0$	$0110:A\oplus B$	$1100:P$	0
$A+B$	$0001:\overline{A}\cap\overline{B}$	$0110:A\oplus B$	$0110:P\oplus C_i$	0
$A+B+C_{in}$	$0001:\overline{A}\cap\overline{B}$	$0110:A\oplus B$	$0110:P\oplus C_i$	FB
$A-B$	$0010:\overline{A}\cap B$	$1001:\overline{A\oplus B}$	$0110:P\oplus C_i$	1
$A-B-C_{in}$	$0010:\overline{A}\cap B$	$1001:\overline{A\oplus B}$	$0110:P\oplus C_i$	FB
$B-A$	$0100:A\cap\overline{B}$	$1001:\overline{A\oplus B}$	$0110:P\oplus C_i$	1
$B-A-C_{in}$	$0100:A\cap\overline{B}$	$1001:\overline{A\oplus B}$	$0110:P\oplus C_i$	FB

図 9.6　ALU の演算命令の例 (算術記号と区別するため AND は ∩，OR は ∪ と記した)

bit	フラグ
15	フラグビット
9	S_{out}
8	MSB への C_{in}
7	LSB
6	MSB(符号あり整数として負，LT)
5	符号あり整数として非正 (LE)
4	符号あり整数としてオーバフロー
3	符号なし整数として正 (GT)
2	C_{out}(符号なし整数のオーバフロー)
1	零 (結果が 0)
0	前回のフラグビット

図 9.7　フラグレジスタの内容 (10 から 14bit は未定義)

積されるため，これをうまく利用すれば，かなり前の ALU の状態を利用することも可能である。

次に条件付演算命令について述べよう。乗除算は，複数のステップによって達成されるが，ステップによっては，MSB などの値によって，実行する計算が異なる。これを，通常のプログラムの条件分岐命令によって実行すると，かなりの時間がかかる。そのため，前命令でフラグレジスタのフラグビットに MSB などの値を記憶し，次命令でその値により実行する機能を変えるという手法を使うことにより，高速化が果たせる。もちろん，すべての条件分岐をこのようにするのは，限りなく機能が増えるので得策ではないが，乗除算のように比較的よく使われ，かつ次命令が定まったものに適用するのは，極めて有効である。こういう形で拡

	K	P	R	機能	C_{in}	FB
乗算ステップ	0000	1100	1100	A	0	0
	0001	0110	0110	$A+B$	0	1
除算ステップ	0010	1001	1001	$A-B$	0	0
	0001	0110	0110	$A+B$	0	1

図 9.8　ALU の条件付演算命令の例 (フラグビット FB の値によって動作を変える。乗算，除算の途中の計算は，フラグの一つである MSB によって機能を変える必要のあるものがある)

張された ALU の機能の一例を**図** 9.8 に示す。

ここまでの説明でシフタと ALU はかなり深い関係にあることを予想できたかも知れない。実際，シフタの主なる活躍場所は乗除算である。シフタも入力側または出力側のいずれかにレジスタが必要であるが，ここではシフタの出力側に置き，かつ関連の深い ALU の入力レジスタと兼用している。シフタの出力が必要な場合も，それを直接シフタからは出さないで，ALU を素通りさせて利用するようになっている。この際，ALU の動作は ϕ_2 になされるから，時間ロスは発生しない。

9.7　制御コード

今まで，命令と制御という言葉をあまり区別なく用いてきたが，制御部から制御線を経由してデータ処理部に渡されるものを**制御コード (control code)** と呼ぼう。一方，**命令 (instruction)** または**命令コード (instruction code)** とは，機械語のプログラムの形で書かれるものであり，制御部はこれを解釈して，制御コードの形にしてデータ処理部に与えるものとする。

さてここで，制御部から与えられる制御コードの形を示しておこう。

bit 位置	内容
22 – 19	K
18 – 15	P
14 – 11	R
10 – 9	条件付演算の選択コード
8 – 6	フラグビットの選択コード
5 – 4	C_{in}の選択コード
3	フラグレジスタ設定
2	ALU の A レジスタへ出力
1	ALU の B レジスタへ出力
0	A バスの内容をϕ_2でリテラルへ移動

図 9.9　演算命令に関する制御コード (ϕ_1 で与えられ ϕ_2 で実行)

制御コードはいずれも 23bit 幅で，ϕ_1 にも ϕ_2 にも与えられる。命令コードのビット幅の 16bit が，制御コードのビット幅の 23bit に対し，やや短くて済むのは，一種の符号化がなされているからである。命令コードは制御部で解釈，つまりデコードされ，この 23bit 幅に変換される。

まず ϕ_1 のタイミングで与えられるのは，ALU に関する演算命令に絡む制御コードであり，ϕ_2 で実行される。そのコードは**図** 9.9 のような意味を持っている。

ϕ_2 のタイミングで与えられるのは，データ処理部内のデータ移動命令に関する制御コードであり，ϕ_1 のタイミングで移動が実行される。そのコードは 2 種類あり，MSB が 0 か 1 かでコードの割り当てが変わる。

MSB が 0 の場合は，リテラルの絡まない通常の移動であり，概要を**図** 9.10 に示す。レジスタ，I/O ポート，ALU レジスタ間の移動を行う。移動には二つのバスを独立に利用できるので，A バス，B バスのそれぞれ

bit 位置	内容
22	0
21 – 17	B バスの通信元
16 – 11	B バスの通信先
10 – 6	A バスの通信元
5 – 0	A バスの通信先

図 9.10　リテラルの関係しない移動命令に関する制御コード (ϕ_2 で与えられ ϕ_1 で実行)

bit 位置	内容
22	1
21	1: データ処理部より制御部へ。0: 制御部から
20 – 5	制御部とのリテラルバスに直結
4	1: レジスタ。0: I/O，ALU レジスタ，フラグ R など
3 – 0	上記レジスタの詳細

図 9.11　リテラルとの移動命令に関する制御コード (ϕ_2 で与えられ ϕ_1 で実行)

の通信元と通信先を 5bit で指定することになる。B バス経由で ALU の B 入力レジスタを通信先とした場合には，シフト量を指定して，シフト結果を送ることも，シフト量を送ることもできるよう，6bit としている。

MSB が 1 の場合は，リテラルの絡む移動であり，概要を**図** 9.11 に示す。A バスを経由して，レジスタ，I/O ポート，ALU レジスタ，フラグレジスタのいずれかとリテラル間の移動を行う。行き先として ALU の A 入力レジスタを指定する場合には，1bit 左シフトしたデータを送ることもできる。

演習問題 9

問題 9.1　次の用語を理解したかどうか確認せよ。

1）バスとトライステートバッファ
2）レジスタ
3）バレルシフタ
4）マンチェスタキャリーチェーン
5）フラグレジスタ，フラグビット

問題 9.2　バスが 1 系統しかない場合，ALU にはいくつのレジスタが必要か。また，レジスタをどこに置くのがよいであろうか。バスが 2 系統の場合，3 系統の場合についても考察せよ。

問題 9.3　シフタのデータを ALU を素通りにして利用したい。K，P，R をどのようにしたらよいか。

問題 9.4　各ビットごとに EQ=NOT(EOR) を計算したい。K，P，R をどのようにしたらよいか。

10 制御部

《目標＆ポイント》CPU を構成する二つのユニットのうち，データ処理部を操作する人間のような役割を演じる制御部について説明する。制御部の構成は，すでに学んだシークエンス回路であるが，その構成やその行う作業について理解を深める。

《キーワード》制御部，シークエンス回路，フラグ，電卓，蓄積プログラム方式，フェッチ，デコード，実行，マイクロプログラム

10.1 固定的作業を行う制御部

データ処理部を制御するのが，**制御部 (control unit)** である。コンピュータの機能部分で，データ処理部分は，主として取り扱うデータのサイズが変わるという歴史を歩んできたが，制御部はその考え方が変わってきたため，若干理解が難しいかも知れない。しかし，根本的にはシークエンス回路であり，それに補助的機能ブロックを付け加えることにより，中心部を簡素にしたり，高速化したりする工夫があったと考えると，理解がしやすいであろう。

もっとも基本となるのは，同じ作業を繰り返し実行する制御部である。例えば，時計のように，クロックごとにカウンタを増やしていくような計算機械がこれに該当する。この作業は，例えば ALU に 1 を加える作業だけを繰り返すことで実行できるが，もちろん，ALU がオーバフローしても止まらず，ALU の有効ビット部分がいったん 0 に戻るだけで無限

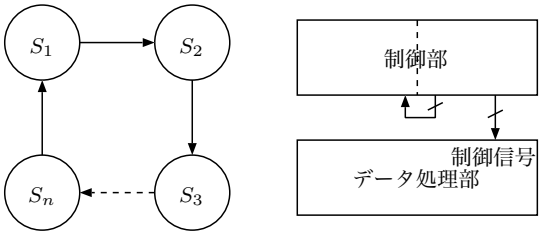


図 10.1　固定作業を行う制御部

に周期は続く。

こうした固定作業を行う制御部は，**図** 10.1 左に見られるように，分岐のない，言いかえれば，分岐条件を与える入力がなく出力しか持たない状態遷移図で表現される。上に示した時計のような例では，状態が 1 個で，かつ遷移のつど，ADD1 のような制御コードが出力される。また**図** 10.1 右に示されるシークエンス回路で実現できる。

特別な場合として，周期性のない状態遷移図も存在する。これは最後の状態から最初の状態への矢印のないものであるが，クロックが来たときにどこにも遷移先がないのは困るので，最後の状態に来ると，以後は自分自身に遷移する矢印にしたがって，無限ループに陥ることになる。例えば限られた数のレジスタ上のデータの合計を求め，それをレジスタ上に書き出して終了するようなプログラムがこれに対応する。

10.2　フラグに依存する制御部

データ処理部，なかでも ALU はその作業結果に応じて**フラグ (flag)** を設定する。**図** 10.2 左に示すように，制御部がこのフラグを見て動作を変えるようになると，いわゆるプログラムのジャンプ命令を実行できるようになる。ALU のフラグには，$A - B$ の結果が 0 か負か非正か，また

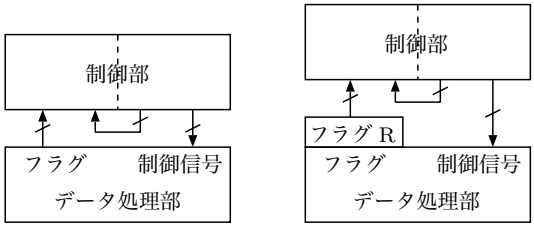


図 10.2　フラグに依存する制御部

ビット列として大であるかを示すものがある。前者は算術演算の条件判定に使えるし，後者は論理演算の条件判定に使える。その結果に応じて，条件ジャンプ命令を実行することが可能となり，プログラムの適用範囲は極めて広くなる。

フラグの結果を命令 1 回限りで利用する場合には，**図** 10.2 左の構造で十分であるが，それをもっと先の命令まで利用することもある。例えば，命令によっては情報が多く，普通より長いものもありうる。こうしたとき，命令を全部読み込むまで，フラグの内容を維持したいことがある。もちろん，フラグの内容をいったん汎用のレジスタに記憶し，それを読み出すことも可能であるが，こうした可能性が数多く発生する場合には，**図** 10.2 右のように，フラグ専用の**フラグレジスタ (flag register)** を用意するのが得策である。

10.3　電卓

電卓 (electronic desktop calculator) とは正式には電子卓上計算機の略であったが，現在は電卓という言葉が定着しているので，以下こう呼ぼう。電卓は蓄積プログラム方式ではないが，ほとんどパソコンと同じような機能を持っているので，勉強のために，具体的構成を述べよう。

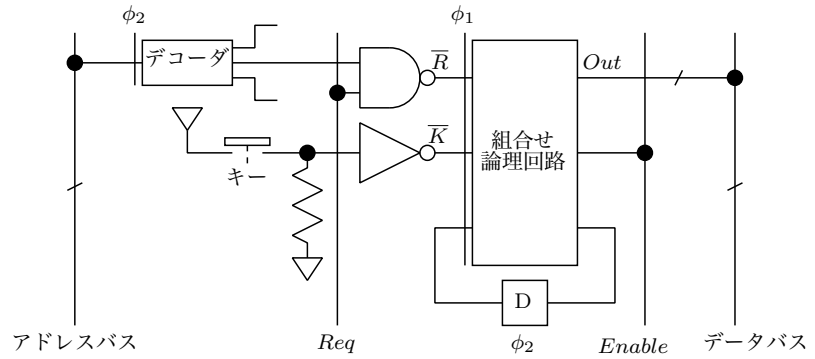


図 10.3　キーボード回路 (組合せ論理回路の前にある ϕ_1 などの線はスイッチ群を意味する)

なお，ここでは，教育的効果を考慮し，コンピュータとの連続性を意識した電卓を示しているが，本当の電卓は，ここで述べるものよりずっと簡素化されている。しかし，動作としての本質はあまり変わらない。

電卓の入出力と言えば，十数個のキーからなるキーボードと，10 桁程度の数字を表示するディスプレーだけである。キーボードは全体として一つの周辺装置を構成するが，ここでは説明の都合上，周辺装置のアドレス一つ一つにキーが結び付けられているものとしよう。これらキーボードに割り付けるアドレスは，数字キーに対しては `0x0000` から `0x0009`，演算キーについては余裕をもって `0x000A` から `0x001F` としておこう。

例えば，キーボードの数字キー「7」が押されているかどうかを知るには，CPU より `0x0007` 番地のアドレスを指定し Req(Request) 信号を与える。キーボードが押されていれば，データバスに `0x0007` を載せて Enable 信号を返す。押されていない場合には `0xFFFF` を返すものとする。

キーボタンについては次のような扱いをする。ボタンを押した場合，押す時間によらず，同じ結果が得られるように配慮する必要がある。ま

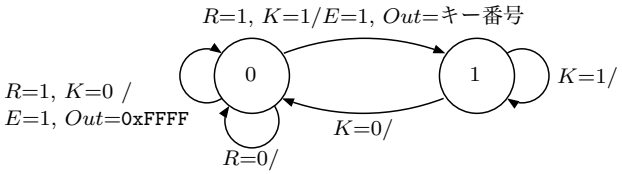


図 10.4　キーボード回路のシークエンス回路用の状態遷移図

た，ボタンのような機械的接点には，いったん接触した後，振動のために何回か離れたりついたりする**チャタリング (chattering)** という現象が発生する。この影響を除去する必要もある。このため，ボタンの後に**図** 10.3 のような回路を用意し，いったん接触した後には，V_h が配線に充電され，その後，抵抗により徐々に電位が下がるようにする。この結果，接点が開になっても，この電位がおよそ $V_h/2$ を切るまでは，K は 1 のままであり，閉になっていると理解される。

この回路に接続されているシークエンス回路は，**図** 10.4 に示すような状態遷移図を持ち，次のように動作する。

まず，内部状態は基本的には 0 で待機している。Req が来ても $K = 0$ の場合には，Out =0xFFFF として $Enable$ で返事をする。$K = 1$ で Req が来ると，Out = キー番号として $Enable$ により返事を返す。この後すぐに Req が来ても重複返事を返さないよう，内部状態を 1 にして K が再び 0 に戻るまで，待機する。K が再び 0 に戻れば，内部状態 0 の待機状態に復帰する。

出力側の数字ディスプレーはもっと単純である。まず，ディスプレーは出力用周辺装置として 1 アドレスが与えられているものとする。数字 1 文字を表示するには 7 エレメントぐらいが必要であるので，10 個の数字としても全部で 70 個ぐらいである。さらに，小数点やエラー表示などを入れても 7bit のデータ幅の情報で，エレメントの指定は十分可能で

ある。また，指定されたエレメントを明るくするか暗くするかの指定に 1bit を割り当て，計 8bit のデータを CPU から送り，その下位 7bit をデコードして必要なエレメントに必要な情報を送ればよい。

残るは CPU であるが，蓄積プログラム方式ではないので，**図** 10.2 に示したものでよい。ただし，レジスタは X と Y と Z の三つとする。また，演算は逆ポーランド算法と呼ばれる順で行う。これは，置数，置数，演算のようにキーを押す。例えば 12+31 は，1，2，ENTER，3，1，+ と押す。また，負数を入れるときには，数字列を入れてから負符号キー (–) を押すこととする。

CPU 内のシークエンス回路の概要は次のようである。ただし，数値入力の初期状態であるかを示す *Init* なる内部変数を用意し，スタート時に $Init = 1$ とする。

1） キーボードチェック。具体的にはアドレスを順番に増加していき，$Req = 1$ にしてキーのチェックに行く。キーが押されていれば *Enable* とともに有為なデータが戻ってくる。押されていなければ，`0xFFFF` が戻ってくるので，これを繰り返す。

2） 数字キーでかつ $Init = 1$ であれば，数字キーからのデータを直接 X レジスタに入れ，$Init = 0$ とする。$Init = 0$ であれば，X レジスタの内容を 10 倍し，このデータを加える。

3） ENTER キーであれば，$Init = 1$ とし，Y レジスタ→ Z レジスタ，X レジスタ→ Y レジスタなる移動を行う。

4） 四則演算キーであれば，$Init = 1$ とし，Y レジスタに対して X レジスタの内容で四則演算を行う。例えば Y÷X の順の計算を行う。結果を X レジスタに入れ，Z レジスタ→ Y レジスタなる移動を行う。

5） 負符号キー (–) であれば，$Init = 1$ とし，X レジスタの内容を補数にする。

6） AC(All Clear) であれば，$Init = 1$ とし，すべてのレジスタの内容を 0 にする。
7） C であれば，$Init = 1$ とし，X レジスタの内容を 0 にする。
8） X レジスタの内容をディスプレーに表示する。
9） 第 1 項へ無条件ジャンプする。

これで，例えば $(2+3) \times (4+5)$ のような計算も可能である。まず，4+5 を計算し，それをレジスタにプッシュしてから，2+3 を計算し，ポップしたものを掛ければよいので，4，ENTER，5，+，2，ENTER，3，+，× と押せばよい。上記の作業を実行する制御部の持つべき状態遷移図の作成は，諸君の演習として残しておこう。

10.4 蓄積プログラム方式

シークエンス回路の動作は，プログラムの形式に書くこともできる。また，プログラムで記述された作業を，シークエンス回路にやってもらうようにすることも可能である。ここでは状態遷移図と機械語プログラムとの変換を考えよう。ただし，2 進表現された機械語ではわかりづらいので，適宜，読みやすく記載した。

例として，状態遷移図**図** 5.2 および状態遷移表**図** 5.3 に示した券売機に対応する機械語プログラムを，**図** 10.5 に示す。なお，In，$Take$，$Ticket$ は周辺装置のアドレスとし，レジスタ R0 には 0，R1 には 1 が代入されているとする。また Addr0，Addr1，Addr2 はメモリー上のアドレスとする。

これから次のような手順が想像できる。

- 複数の矢印が出ている丸に対応し，条件チェックに続く条件ジャンプが必要

```
Addr0:  LD(2, In);      // In の内容を R2 へ移動
        SUB(2, 0);      // R2 から 0 を引く
        JPZ(Addr0);     // 零ならば Addr0 へジャンプ
        ST(1, Take);    // そうでなければ Take へ 1 を出力
Addr1:  LD(2, In);      // In の内容を R2 へ移動
        SUB(2, 0);      // R2 から 0 を引く
        JPZ(Addr1);     // 零ならば Addr1 へジャンプ
        ST(1, Take);    // そうでなければ Take へ 1 を出力
Addr2:  LD(2, In);      // In の内容を R2 へ移動
        SUB(2, 0);      // R2 から 0 を引く
        JPZ(Addr2);     // 零ならば Addr2 へジャンプ
        ST(1, Take);    // そうでなければ Take へ 1 を出力
        ST(1, Ticket);  // Ticket へ 1 を出力
        JP(Addr0);      // 無条件に Addr0 へジャンプ
```

図 10.5　券売機の状態遷移図に対応するプログラム

- ジャンプ先はプログラムを書いてアドレスが決まってから入れる
- 出力は順次出す
- その他，逐次処理に合わない流れがあれば無条件ジャンプが必要

なお，この手順で作成された機械語プログラムを 1 行 1 クロックで動かすと，状態遷移図のものよりは遅くなるので注意して欲しい。例えば，最後のほうにある *Take* と *Ticket* は 2 回のクロックを使っているが，元は 1 回であった。

同様にして逆の変換は次のようになる。

- 条件ジャンプに対応して丸を置く
- 条件ジャンプの前に記載されている条件チェックを，各矢印に

記載されている入力欄に書く

- ジャンプ先を見て，矢印の行き先を決定する
- 順次出される出力を各矢印の出力欄に書く
- 無条件ジャンプは状態遷移ではないので，これまでの遷移図の中に埋没するはずである
- 各矢印を見て，1 クロックで処理できない作業が複数入っている場合には，間に適宜丸を挿入し，作業を分割する

状態遷移図とプログラムを比較してみると，おそらくプログラムのほうが読みやすいであろう。一般に，コンピュータに大きな作業をさせたいときには，プログラムのほうが便利である。一方，シークエンス回路は解釈の時間が不要であり，速度が速い。さらに，プログラムでは，これを解釈して実行していく何らかの頭脳が必要である。その頭脳はシークエンス回路で作成せざるをえない。したがって，現実的な答えとしては，比較的サイズの大きな作業はプログラムで書き下し，最小限の頭脳だけをシークエンス回路で作成するという形になろう。

前節までの構造では，実行すべき制御コードは，制御部内のシークエンス回路の形に記載する必要がある。これを，命令コードの集合，つまり機械語プログラムという形で，外部のメモリー上に置いておき，それを解析しながら制御コードを作り出していけるようになると，コンピュータはほとんど何でもできるようになる。これが**蓄積プログラム方式 (stored program concept)** である。

このしくみが導入されると，ほぼ現在のコンピュータのできることはすべてできるようになる。**図** 10.6 に蓄積プログラム方式によるプログラム実行の手順を示す。この話の中で，汎用レジスタの一つを**プログラムカウンタ (program counter)** または **PC** と呼ぼう。ここには次に実行されるアドレスが記憶される。また，もう一つの汎用レジスタを**命令レ**

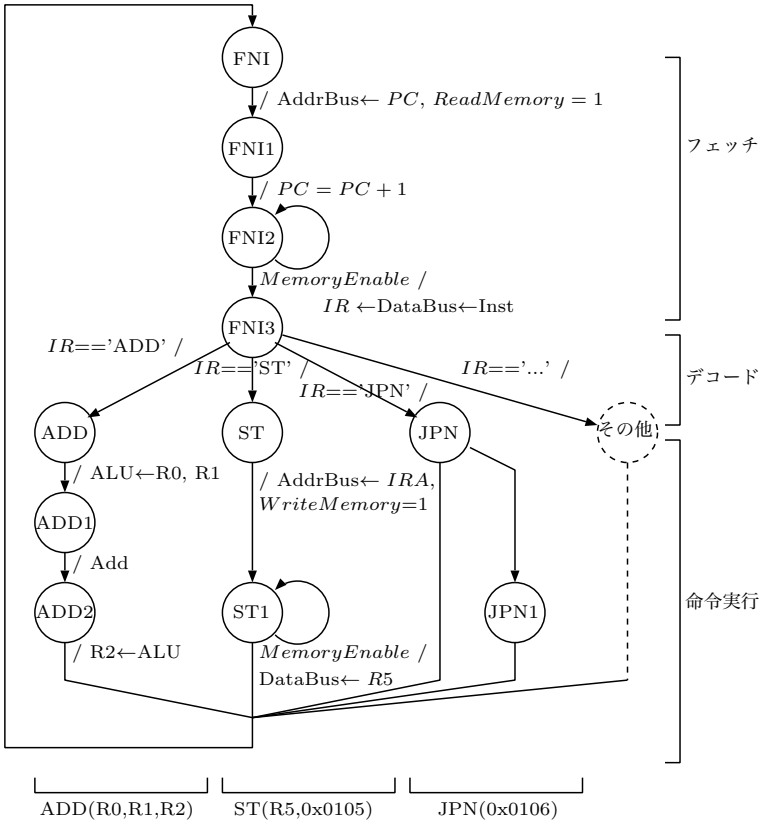


図 10.6　制御部の状態遷移図 (‘=’ は代入，‘==’ は左右が等しいことを意味する。)

ジスタ (instruction register) または **IR** と呼ぼう。ここには現在実行中の命令が記憶される。2 語命令の場合には，2 語目のアドレス情報を入れるレジスタも必要であるので，これを IRA と呼ぼう。

蓄積プログラム方式はフェッチ，デコード，実行という三つの作業からなる。まず，外部メモリーから次の命令を持ってくる**フェッチ (fetch)** という作業が行われる。この際，次の命令の入っているアドレスは PC に入っているから，その値をアドレスバスに載せる。そして，通常，さらに次の命令は一つ先のアドレスに入っているので，PC を一つ増やしておく。次はメモリーがデータ (命令) を返事してくるのを待つ。データが来れば，それを IR に入れる。具体的な作業手順は次のようになる。

1） ϕ_1: PC の内容を，バス A，I/O を経由してアドレスバスに載せ，外部制御線を使ってメモリーに読み出し要請 $ReadMemory$ を送付する。同時に，PC の内容を ALU の入力レジスタにも設定する。

2） ϕ_2: PC を 1 増やす。

3） ϕ_1: $MemoryEnable$ により，メモリーが準備できたことを確認したら，その内容を I/O 経由で IR へ移動する。また，ALU の出力レジスタの内容を PC へ設定する。

2 語命令の場合には，同様な手順で，次のメモリーの内容に入っているアドレスを，もう一つのレジスタである IRA に入れる。

次は**デコード (decode)** であるが，これはメモリーから来た命令を解釈して，データ処理部の理解できる形に直す作業である。例えば，レジスタ 5 の内容を `0x0105` 番地のアドレスに入れよという命令 `ST(5, 0x0105)` に対応する機械語は，**図** 8.4 に従うと `0xE015 0105` であるが，これを，次に述べる実行プロセスに対応した個々のゲートの開閉の指示に変更しなければならない。ϕ_1 期間の指示も ϕ_2 期間の指示も，この作業で一緒に作られる。この作業は，通常，1 クロック周期でなされる。

最後は**実行 (execute)** であるが，移動命令や演算命令やジャンプ命令をまさに行う。まず移動命令であるが，例えば `ST(5, 0x0105)` は次のように実行される。

1）ϕ_1: IRA から `0x0105` をバス A，I/O を経由して，アドレスバスに設定する。同時に外部制御線を使ってメモリーに書き込み要請 $WriteMemory$ を送付する。

2）ϕ_1: $MemoryEnable$ が戻ってきたら，レジスタ 5 のデータをバス A，I/O を経由して，外部データバスに載せ，書き込みを終了する。

演算命令は，例えば `ADD(0, 1, 2)` は次のように実行される。

1）ϕ_1: レジスタ 0 とレジスタ 1 の内容を ALU の入力レジスタに移動する。

2）ϕ_2: 加算を行い，結果を ALU の出力レジスタに入れる。

3）ϕ_1: 出力レジスタの結果を，レジスタ 2 に移動する。

ジャンプ命令，例えば `JPN(0x0106)` は次のように実行される。

1）ϕ_1: フラグレジスタ LT をチェックし，0 ならば制御部の次の状態を FNI とする。1 ならば制御部の次の状態を JPN1 とする。

2）ϕ_1: IRA にある `0x0106` を，バス A を経由して PC にセットする。

なお，フェッチ，デコード，実行の各作業にかかる時間は，ALU の演算を ϕ_2 のタイミングで行うため，外部メモリーの遅延がなければ，いずれも 2 クロック周期以内である。この各作業が同じクロック周期内でなされるということが，先に述べるパイプライン処理という高速処理に役立つのである。

コンピュータを蓄積プログラム方式に対応した構造にするには，**図** 10.7 に示すように，外部メモリーを置くことはあたりまえであるが，その内容を制御部へ送るしくみが必要となる。ここでは，データバスに接続されているリテラルポートを利用している。

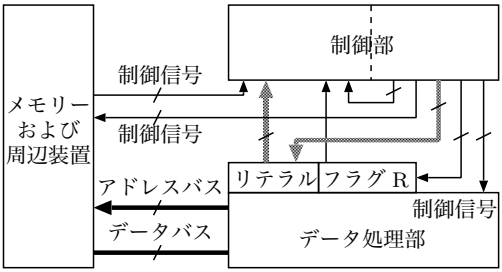


図 10.7　蓄積プログラム方式に対応した制御部

ここまでで，コンピュータのしくみについての本質的説明はほぼ終了である。本章のここから先は，やや細かいことであり，気楽に読んで欲しい。

10.5　マイクロプログラム

制御部は**シークエンス回路 (sequential logic circuit)** であるが，まだまだそのサイズは巨大である。そこでさらなる改良を試みよう。前節に示した方法で，**図** 10.6 に示した状態遷移図をプログラム化してみよう。その結果は**図** 10.8 のようになる。なお，out は制御線への出力命令，jp はこのプログラム内でのジャンプとする。

このように，制御部のシークエンス回路の動作をプログラムの形式で書いたものを**マイクロプログラム (micro-program)**，略して**μP(micro-program)** という。以下，μP とはマイクロプログラムの省略形である。この μP をプログラムとして実行すると，その出力は元のシークエンス回路の出力とまったく同じになるはずである。そのためには，制御部の中にコンピュータのようなものを作る必要がある。なお，μP は外部メモリーとは別の制御部内の**μP メモリー (micro-program memory)** に記

```
FNI:      out(MV,PC,AddrBus);   // PC 上のアドレスをアドレスバスへ
          out(MV,1,MemRead);    // メモリーに「読み出し」を伝える
(FNI1):   out(ADD1,PC);         // PC カウンタを 1 増やす
(FNI2):   out(MV,DataBus,IR);   // データバスのデータを IR へ
(FNI3):   jp(1,IR);             // IR に入っている ADD などの
                                          対応アドレスへジャンプ
ADD:      out(MV,RX,INA);       // ALU 入力レジスタに設定
          out(MV,RY,INB);       // ALU 入力レジスタに設定
(ADD1):   out(ADD);             // 加算の命令コードを出力
(ADD2):   out(MV,OUT,RZ);       // ALU 出力レジスタを RZ に移動
          jp(1,FNI);            // FNI へ戻る
ST:       out(MV,IRA,AddrBus)   // IRA の内容をアドレスバスへ
          out(MV,1,MemWrite);   // メモリーに「書き込み」を伝える
(ST1):    out(MV,RX,DataBus);   // RX の内容をデータバスへ
          jp(1,FNI);            // FNI へ戻る
JPN:      out(MV,Flag,FlagR);   // FlagR へフラグを入れる
          jp(FlagR,JPN1);       // FlagR=1 のとき JPN1 へジャンプ
          jp(1,FNI);            // FNI へ戻る
JPN1:     out(MV,IRA,PC);       // IRA の内容を PC にセット
          jp(1,FNI);            // FNI へ戻る
```

図 10.8　マイクロプログラム化されたシークエンス回路の動作 (out は制御線への出力，jp は μP 上でのジャンプ命令。MV は移動命令)

憶しておくこととする。

このような，コンピュータの中のコンピュータといった二重構造にすると何が得なのだろうか。まず，シークエンス回路の場合には，AND-OR 回路の AND 側は，フラグビット数と命令ビット数と内部状態ビット数の合計のビット幅という非常に大きなビット幅を必要とするため，すべてのビットパターンに対応させることは不可能であり，通常のメモリーに置き換えることはできないことに注意して欲しい。しかし，これをプログラム化すると，その中で起こりうるパターン数だけの**μP アドレス (micro-program address)** を持つメモリーを用意するだけでよい。

メモリー化できるということは，μP メモリーを書き込み可能な ROM(PROM) 化するとか，別のメモリーとすることにより，複雑な詳細設計を最後まで遅らせることができるという利点がある。また，その内容を変えることにより，容易にいろいろな命令セットに対応できる中央処理装置を作成することができる。さらに，設計者にとって，状態遷移図を描くよりもプログラムを書くほうがやさしいので，設計が楽になるという特長も無視できない。

μP 制御回路 (micro-program control circuit) は，さすがに純粋なシークエンス回路で構成する。その状態遷移図を**図** 10.9 に示す。**図** 10.8 に示した μP からわかるように，μP 命令には out と jp しかないので，状態遷移図は極めて簡単である。out の場合には，その行に書かれた制御線に送るコードを直接出力する。jp の場合には，jp 命令コード後半に書かれたアドレスを，μP のアドレスカウンタである**μPC(micro-program counter)** に入れる。out 命令は μP アドレスを持たず，jp 命令は制御コードを持たない。

こうした処理のできるための CPU の構造を**図** 10.10 に示す。その場合，図中破線で囲った制御回路と書かれた部分は，μP メモリーに記憶され

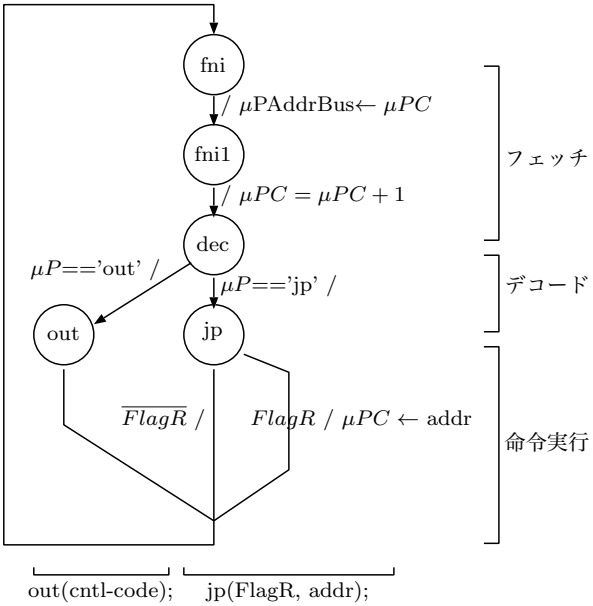


図 10.9　マイクロプログラム制御回路の状態遷移図 (‘=’ は代入，‘==’ は左右が等しいことを意味する。)

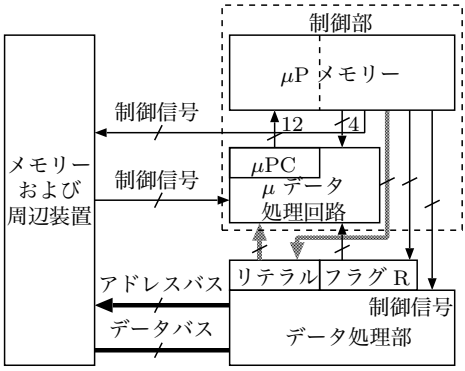


図 10.10　制御部のマイクロプログラム化

た μP を順次解析して実行していく必要があるので，ちょうどコンピュータの CPU と同じような作業をすることになる。つまり，破線で囲った部分はコンピュータそのものと同じような構造であり，コンピュータと同様な**μP データ処理回路 (micro-program data processing circuit)** と**μP 制御回路 (micro-program control circuit)** を持つことになる。

μP 制御回路は，プログラム化された μP の命令を格納している μP メモリーそのものである。この例では命令は 2 種類，つまり 1bit で表現できるが，もっと複雑な間接アドレス指定や関数呼び出しなどのジャンプ命令を加えても，4bit もあれば十分である。また，μP アドレスの幅は 12bit もあれば十分である。一方，制御コードのビット幅は 20 から 50bit 程度必要である。本書の例では 46bit を出力し，その後の回路で，半分を ϕ_1，残る半分を ϕ_2 のタイミングで，データ処理部への制御線として出力するようにしている。

μP データ処理回路はジャンプの際，μPC の計算を行う。しかし，ほとんどの場合，処理すべき計算は μPC を 1 増やすことだけである。ときに，ジャンプ命令として，「5 番地先へジャンプ」のような相対的番地で指定できるものも含む場合がある。こうした場合には，任意の数との加算器が必要である。こうした相対的な番地指定を，通常の**直接アドレス指定 (direct addressing)** に対し**間接アドレス指定 (indirect addressing)** という。

こうした回路によって，マイクロプログラム方式の制御回路は，外部メモリー上のプログラムを解釈していき，あたかも通常の制御回路のような制御コードを作成していくのである。

10.6　高速化への工夫

コンピュータの開発につれて，種々の命令はあればあるほど便利であるということから，その数がどんどん多くなる傾向にあった。当然，それだけ制御線の数も増え，制御部のサイズも大きくなっていった。ところが，各命令の使用状況を調べたところ，かなりの命令がほとんど使われないことがわかってきた。このため，使われない命令を整理し，また乗除算のような面積をとる部分を避け，加減算などの単純な処理の組合せに戻すことにより，逆に速度を上げようというコンセプトが出てきた。

1980 年代初頭より，このコンセプトに基づき設計されたコンピュータが出現し，それらを **RISC(reduced instruction set computer)** という。これに対し，従来の複雑な命令を持つものを **CISC(complex instruction set computer)** という。かつては，高集積が難しく，かつ内部クロックの周期内で外部メモリーにアクセスできたので，レジスタを少なくし，その代わりに外部メモリーを使用してきた。このため，算術演算も直接メモリーのデータに対して行われることが多く，算術演算命令ですら，外部メモリーアドレスを与えるといったことで，横に長い複雑な命令セットが好まれて使われてきた。

これに対し現在は，高集積が楽になり，またクロックが速くなり，なるべくメモリーアクセスをしない設計となってきた。つまりは，多くのレジスタを用意し，算術演算などはすべて高速なレジスタ間で行うという設計になってきたのである。こうなると，メモリーとレジスタ間のロードおよびストア以外に，メモリーアドレスを必要とする命令がなくなり，一気に命令の再整理が進んだのである。

命令はあまり内容を解読しなくても済むよう，固定長となり，また，後に述べるパイプライン化を最大限に利用して，演算はなるべく 1 クロッ

ク内に納まるように設計された。

一方，プログラム中にある関数やサブルーチン呼び出しに便利なスタックは持たず，これらはコンパイラやアセンブラにより解決するようにした。つまり，高速化のために，便利さを犠牲にしたのである。便利さと言っても，末端のユーザから見ると高速なだけ便利であり，あくまでも高級言語から機械語を作り出すまでの翻訳プログラムへのユーザからは見えない負担になっているだけである。

ところで，実際のマイコンの CPU は一時 RISC 化が進んだが，多量に売られているものは，RISC の概念を一部取り入れつつも，かなり CISC 的である。それは，技術的に良いものであっても，過去の資産を生かすためには伝統を継承するほうが効率的であるからである。技術の良いものが必ずしも普及製品とならないのは，他の家電製品などにも多くの例がある。ちなみに，RISC は統一的に設計されているためわかりやすく，本章の制御部の説明は RISC を前提とした。

RISC とあいまって進んだ技術が**パイプライン (pipeline)** 処理と呼ばれるものである。文字通り，パイプラインの中の石油のように，隙間なく無駄なく処理をしていこうというものである。

制御部の仕事は概ね

1） 命令のフェッチと PC の増加
2） 命令とオプションのデコード
3） 命令の実行

といった手順を次々に実行していくことにある。しかし，多くの場合，これら三つの仕事は独立していて，同じ回路を使うことは少ない。そこで，最初の命令実行の第 1 段階が終わって第 2 段階を実行するとき，次の命令実行の第 1 段階を開始してしまおうという考えが成立する。輪唱のようなものである。うまく行けば 3 倍の速度が得られる。メモリーア

クセスのあるフェッチや移動命令では，メモリーの準備を待つ必要があり，これらの命令の周期は必ずしも一致しないが，一般には，並列実行するほうが速くなる。

また，次の命令が前の命令の結果を利用するような場合には，無駄な作業をしてしまうことになるが，それでも，パイプライン処理をすれば，必ず速くなる。

もちろん，そのためには CPU の構造も変えなければならない。例えば，命令レジスタは複数必要になる。算術演算命令とアドレス計算が，かち合わないようにしなければならないなどの工夫も，必要である。しかし，CPU の構造を若干変えるだけで，かなりの速度向上が得られることは大きな魅力である。

SIMD(single instruction multiple data) とか**ベクトル処理 (vector processing)** と言われる工夫もある。数値計算に特化したコンピュータでは，複数の数値に同じ手順の計算を行うことが少なくない。つまり，数の組 (ベクトルということが多い) に対して，順に同じ手順を行うことになる。こうした場合，ALU を複数用意し，同時に同じ命令を実行させることにより，計算速度を上げる工夫である。

現在のコンピュータはこうしたいくつかの工夫のもとに，どんどん高速化されていっていることを理解して欲しい。

演習問題 10

問題 10.1　次の用語を理解したかどうか確認せよ。

1）固定作業を行う制御部

2） フラグに依存する制御部

3） 蓄積プログラム方式に対応した制御部

4） マイクロプログラム

5） RISC，パイプライン，ベクトル処理

問題 10.2　p. 143 の時計の例に対し，制御部の具体的な回路を描いてみよ。

問題 10.3　**図** 10.5 から逆の変換をして，券売機の状態遷移表が得られることを確認してみよ。

問題 10.4　**図** 10.9 に示した状態遷移図を持つ μP 制御回路のシークエンス回路を完成してみよ。

11 コンピュータの将来

《目標&ポイント》技術の進展に合わせ，電子デバイスの性能はどのように変化したのであろうか，またその結果，コンピュータはこれからどのように発展していくのであろうか，将来を展望する。

《キーワード》動作速度，電力損失，スケール則，ムーアの法則，遅延時間，消費電力，汎用コンピュータ，専用コンピュータ，スーパーコンピュータ

11.1 c-MOS ゲートの動作速度と電力損失

前述のように，コンピュータの発展は集積回路の発展と強く結び付いている。その発展の元には，**スケール則 (scaling rule)** という原理が存在する。これは一口で言えば，トランジスタのサイズを小さくすればするほど，その特性が高機能になるという概念であり，このために，半導体メーカーは競って微細化技術を開発し，これが**ムーアの法則 (Moore's law)** と呼ばれる指数関数的経験則に沿った発展を促したのである。

本節はスケール則のやや詳しい説明を行う。c-MOS インバータは消費電力 0 と言ったが，実際には電力消費もあるし，動作速度も有限である。遅延について言えば，デバイスそのものも動作遅れがあるし，さらに大きな原因として，次段のゲートや配線の持つ静電容量の充電時間遅れがある。静電容量があるから出力電圧を上げる際には充電時間がかかるし，出力電圧を下げる際には放電時間がかかる。また，充放電の際，電力損失も発生する。

デバイスそのものが持つこの時定数は**内因性遅延 (intrinsic delay)** と呼ばれる。これは，n-MOS FET や p-MOS FET で，ゲート電位を変えて OFF から ON にする際，ゲート下に電荷を呼び込んでチャネルを形成する時間である。概ね，ゲート容量 C_g とチャネル抵抗 R_c との積 $\tau_0 = C_g R_c$ で与えられる。この時間は，結局は電荷がゲートを通過する**走行時間 (transit time)** と一致する。内因性遅延はデバイスの動作に起因するため，集積回路のいろいろな遅延を議論するための比較標準のような使われ方をするために，しばしば現れるが，実はもっと大きな遅延があるために，普通は無視できることが多い。

その他の遅延は総じて**外因性遅延 (extrinsic delay)** と呼ばれる。次段のゲート容量の充放電時間はまさにゲート容量へチャネル抵抗を経由して電荷を出し入れする時間で計算できる。もし，次段ゲート容量が前段のゲート容量と同じであると，その時間は $\tau_0 = C_g R_c$ で与えられる。もし，次段に複数の FET が接続されていると，$C = fC_g$ が成立するため $\tau = fC_g R_c = f\tau_0$ となる。ここで f は一つの FET が後段のいくつの FET を駆動するかを示す**ファンアウト (fan out)** と呼ばれる概念である。

現在の集積回路，あるいはボード上に作られた回路でもっとも大きな遅延は，配線の持つ容量の充放電時間である。というのは，回路規模が大きくなるにつれ，FET 間の接続が膨大になってきたからであり，また，関係の深い FET 同士を必ずしもすぐ近くに配置するのが困難になってきたからでもある。実は現在，前に述べた内因性遅延やファンアウトによる遅延はほとんど無視でき，配線遅延が圧倒的になっている。

配線容量は出力線から接地側，つまり 0V 側に対して静電容量を持つ。出力電圧を上げるときには，電源である V_h より p-MOS を経由して，この静電容量に充電する必要がある。FET のチャネル抵抗 (時間とともに

変化するので平均的な抵抗) を R_c とすると，充電にはおよそ $\tau = CR_c$ ぐらいの時間が必要となる。また，出力電圧を下げるときには FET のチャネル抵抗によって放電を行うが，それにも同じ程度の時間を必要とする。先程のファンアウトの係数を利用して，ここでも $C = fC_g$ と記載すると，$\tau = fC_gR_c = f\tau_0$ となる。

いずれの時定数も R_c に，それぞれ該当する静電容量を掛けることにより得られるので，種々の静電容量の総和をゲート容量を基準にして fC_g と表せば，

$$C = fC_g \tag{11.1}$$

なので，そのゲート全体の遅延時間は

$$\tau = f\tau_0 \tag{11.2}$$

となる。

容量の充放電の際，各 FET では僅かであるが，エネルギー損失が発生する。この損失は，充放電電流がチャネル抵抗を流れる際に発生するが，もともと容量に溜まっているエネルギー $CV_h^2/2$ を放出することになる。クロック周波数が f_c のとき，クロックのたびごとに出力論理が反転するものとすると，その平均電力は

$$P_0 = f_cCV_h^2/2 \tag{11.3}$$

で与えられることになる。再び $C = fC_g$ とし，このゲートの出力の反転確率を p としよう。すると，このゲートでの電力消費は

$$P = pfP_0 \tag{11.4}$$

となる。ただし，$P_0 = f_cC_gV_h^2/2$ である。

ゲートの動作時間と消費電力が得られたので，次に，これらがFETのサイズにどう依存するかを調べよう。簡単にサイズと書いたが，FETには縦も横も高さもある。通常スケール則という場合には，縦横高さすべてを同じ比$1/k$に縮小することを意味する。さらに，単位長当たりの電場が一定に維持されるよう，かける電圧も同じ比$1/k$に縮小することにする。

FETの内因性遅延τ_0は，電荷のチャネルを通過する時間，つまり走行時間で与えられる。したがって次式が成立する。

$$\tau_0 = C_g R_c = \frac{L}{v} = \frac{L^2}{\mu V_{ds}} \tag{11.5}$$

速度は電場の強度に比例するがその比例係数をμとした。またV_{ds}は主電極間の電位差である。LもV_{ds}も$1/k$になると，走行時間も$1/k$に比例して短くなり，結局$\tau_0 \propto 1/k$が成立する。つまりはゲートごとの動作時間も$\tau \propto 1/k$となり，高速で動作するようになるのである。これは大変に朗報である。加工技術を上げて微細化すればするだけ，回路は高速で動作できるようになるのである。つまり，クロック周波数f_cもk倍に上げられることになる。

続いて消費電力を検討しよう。消費電力の式の中にC_gが現れるが，このサイズ依存性を検討しておく必要がある。一般に容量は面積に比例し，電極間の距離に反比例するから，結局$C_g \propto 1/k$となる。これにV_h^2が掛かること，さらにf_cがk倍で速くなることから，$P \propto 1/k^2$となる。つまり単体のFETの消費電力はサイズとともに急速に減少する。チップ単位面積当たりの消費電力は，集積度がk^2で上がるため，FETのサイズに依存せず，一定となる。これも朗報である。消費電力は発熱に対応するが，これがサイズに依存せずほぼ一定であるということは，集積化の障害要因にならないことを示している。

以上が 2000 年ごろまでのトレンドであり，FET のサイズを減らすことにより高集積，高速になるだけで，発熱は変わらないというスケール則が成立したため，可能な限り，サイズを減らす努力をしてきたのである。

2000 年を越えるころから，このスケール則に影が差してきた。それは，FET の出力電圧が小さくなり過ぎると，温度による揺らぎの影響が出てくることである。このため，V_h は約 1V 程度から下げることが困難となってきたのである。つまり，サイズだけ $1/k$ になるが，電圧は一定というスケール則を考えねばならなくなってきたのである。

これはチャネルにかかる電場を強くし，結果として充放電時間は $1/k^2$ に比例して短くなり，高速化は果たせるものの，FET1 個当たりの消費電力は k に比例し，単位面積当たりの発熱は k^3 で増大することとなったのである。もちろん，いきなり k^3 に比例し始めたわけではないが，それでも発熱はかなり深刻な問題となってきたのである。

この発熱の増大に対処するには，放熱をよくするか，集積度を抑えるしかなくなる。かつては IC をそのまま置いていたのが，大きな放熱板を必要とするようになり，現在では水冷も本気で考えられている。集積度を抑えることも少しずつ始まっている。あるいは，部分ごとに小まめに電源を落とすなどの，細かい省エネ設計なども採用されつつある。

11.2　汎用コンピュータと専用コンピュータ

本書で述べてきたコンピュータは，特に使用目的を限定しない何にでも使える汎用コンピュータと呼ばれるものであった。しかし，使用目的がある程度確定している場合には，その目的に沿った設計をするほうが高い処理速度が得られてよい。

例えば，**スーパーコンピュータ (supercomputer)** と呼ばれるものは，

科学技術などの巨大な数値計算に適したコンピュータである。データ幅を大きくし，かつ多くの算術計算を同時並行できるように SIMD であるベクトルプロセッサと呼ばれるしくみを搭載したものが多い。さらに，近年は 1 チップに複数の CPU を載せたものをさらに多数密に結合したマルチ CPU のものも多くなりつつある。

ゲーム用マシンなども特別の CPU 設計がなされる。特に映像に頼ったゲームなどの場合には，物体の反射光などの処理のための計算量が多くなる。このため，浮動小数点計算専用とか，光線の計算に特化した ALU などを搭載した CPU が使われる。このように，ある程度目的がはっきりしたコンピュータには，必要とされる機能を強化した CPU を設計するのが普通である。

11.3　将来のコンピュータ

2000 年ごろまでは，将来のコンピュータというと，ひたすら微細化して高集積化し，高速と高機能を同時に果たしながら開発を進めてきた。そういう意味でコンピュータのロードマップを描くことは比較的容易であった。おそらく，より大きなメモリー空間を持ち，より高いクロック周波数で動作し，より大きなデータ幅を持ち，といったそれまでの傾向を延長した特性により将来予測すればよかった。

しかし，21 世紀に入るころから，こうした予測は急速に難しくなってきた。それは，高集積に限界が見え出してきたからである。このため，どの機能を重点的に開発するのかといった選択の幅が急に増えてきたと言える。一言で言えば，専用マシンの設計に移行しつつあるのである。つまり，どのような応用分野が成長するのかによって，どのようなコンピュータが必要なのかが決まってくるため予測が急速に難しくなってき

たと言えよう。

しかし，コンピュータの根本的な構成はそれほど変わらない。おそらく，今後も CPU を中心にメモリーと周辺装置が置かれるという根本原理は変わらないであろう。ただ，周辺装置の種類は大いに変わりうる。

同様に CPU 自身もデータ処理部と制御部で構成されることも，大きくは変わらないであろう。変わるとしたら，データ処理部の ALU の構成ぐらいかも知れない。

11.4　おわりに

現在，子供の理科離れが甚だしい。また，この講義のような要素を組み上げていく，いわゆるボトムアップタイプの講義の人気も下がっていると聞いている。ものごとを大づかみで理解するのが好まれ，そうしたトップダウンの講義が歓迎されるようである。しかし，本当にそれだけでよいのであろうか。今，家庭に限らず，工場や電力会社，通信会社でも，中身を知らないでも済むような制御装置が行きわたっている。しかし，複雑な故障が発生した場合，制御装置の前でジタバタしても何の解決にもならない場合が多い。制御装置の末端に何が繋がっており，それがどうなったかが想像できると復旧も速いのである。

災害のような深刻な問題が発生したときにも，本質に遡って，物事に対処できる能力を持つことが必要なのである。そうした立場から，本書ではあえてボトムアップの原理原則から理解できるような構成を試みた。

一般にボトムアップの議論は，ある程度一生懸命にならないと理解し切れないものである。しかし，いったん，ある程度のところまでわかってしまうと，全体が突然見えてくるようになる。算数のようなものなのである。この講義を漫然として聞いてしまった人も多いかも知れないが，

また，突然疑問を感じて，復習してみたくなるかも知れない。そういうときには，押し付けられてではなく，自分の意志で学ぼうと思っているので，理解は速いと思う。ぜひ，そんなチャンスを利用して，ちょっとでも物事の本質を理解する力を養ってもらいたいと希望する。

演習問題 11

問題 11.1　電圧一定のときのスケール則を確認せよ。

参考文献

本書の執筆にあたり，次の書籍が大変参考になった。しかし残念なことに，この書は現在は絶版となっている。

- Carver Mead and Lynn Conway, “Introduction to VLSI Systems”, Addison-Wesley Publishing Company, 1980

また，最近は，この書に記載したようなコンピュータのハードウェアの詳細を記述した書は得難くなっている。したがって，まずは，本書をしっかり理解するのがよいと思う。

その他，論理回路やシークエンス回路のようなコンピュータの構成要素については，「論理回路」とか「ディジタル回路」といったような名称の多くの書があるので，それらを参考にされたい。

また，個々の用語に関する情報は，Web に数多く記載されているので，Google などの検索システムで探すことを勧める。

演習問題解答

1 章

1.1 本文を見よ。以後，用語の理解については解答は示さない。

1.2 通常，昔の文字盤表示の時計がアナログで，数字表示の時計がディジタルと言われている。しかし，文字盤表示の時計も，振子とかテンプを使って歯車を一歯ずつ動かしているので，実は振子などの周期を計数しており，ディジタルなのである。もちろん，現在の多くの時計は，水晶振動子の発振している波を数えているので，これもディジタルである。

厳密な意味でのアナログ時計は，水漏式や蝋燭の燃えるのを利用した極めて古典的なものしかないかもしれない。

1.3 古いテレビは，映像の各点の明るさがアナログ的であった。さらに，各点と言ったが，垂直方向は走査線により確かに不連続であるが，水平方向は連続であるので，これもアナログ的である。

しかし，地上波ディジタルで代表されるディジタルテレビでは，水平，垂直とも不連続なため，まさに点の集合であり，さらに各点の明るさも不連続な階調を持つことから，その意味でもディジタルである。

1.4 4bit の信号線のそれぞれを 0 または 1 にできることから，すべての組合せは $2^4 = 16$ となり，16 種類の情報を伝えることができる。

自然数を 2 進表示した場合，4bit では 0000，0001，0010，...，1111 の自然数が表現できる。最小数はいうまでもなく 0 であるが，最大数は 1111，つまり $1 + 2 + 2^2 + 2^3 = 15$ となる。

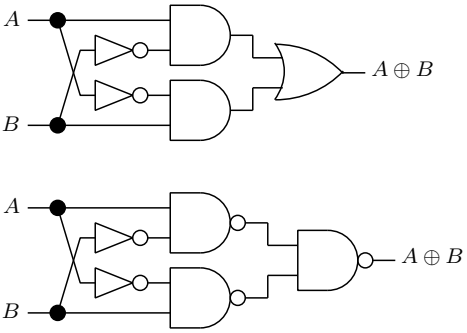


図.1　EOR の回路

3 章

3.2　2 入力 NAND の下を 3 個の n-MOS の直列回路とし，上を 3 個の p-MOS の並列回路とし，新たに増えた n-MOS および p-MOS のゲートを In_3 とする。

3.3　2 入力 NOR の出力のあとに NOT をつける。

3.4　2 入力 NOR の下を 3 個の n-MOS の並列回路とし，上を 3 個の p-MOS の直列回路とすればよい。新たに増えた n-MOS および p-MOS のゲートを In_3 とする。

4 章

4.1　1）真理値表での確認は各自に任せる。NAND はすべての入力が 1 のときにのみ，出力は 0 となる。したがって NAND(NOT) はすべての入力が 0 のときにのみ 0 となり，OR と同じ動作となる。こ

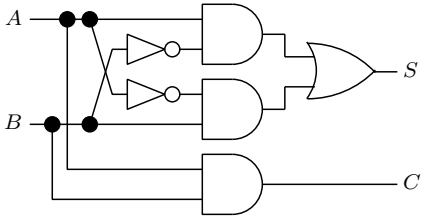


図.2　半加算器の回路 (S の回路は図 .1 のいずれでもよい)

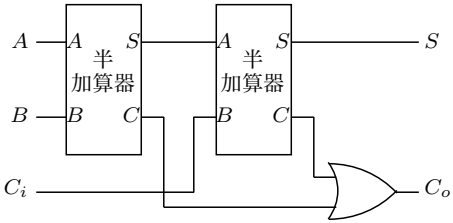


図.3　全加算器

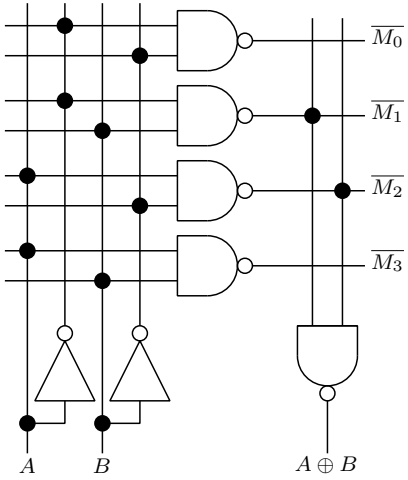


図.4　EOR の NAND-NAND 回路 ($\overline{M_0}$ と $\overline{M_3}$ を作る回路は不要であるので，これらを消去すると，図 .1 の下図と一致する)

れは入力数によらない一般的性質である。

2）真理値表での確認は各自に任せる。NOR はすべての入力が 0 のときにのみ，出力が 1 となる。したがって NOR(NOT) はすべての入力が 1 のときにのみ 1 となり，AND と同じ動作となる。これは入力数によらない一般的性質である。

4.2　1）$\mathrm{OR}(X,Y,Z,W)$=$\mathrm{OR}(\mathrm{OR}(X,Y),\ \mathrm{OR}(Z,W))$ の右辺に OR=NOT(NOR) を代入すると，

$$\begin{aligned}&\mathrm{OR}(X,Y,Z,W)\\&\quad=\mathrm{NOT}(\mathrm{NOR}(\mathrm{NOT}(\mathrm{NOR}(X,Y)),\ \mathrm{NOT}(\mathrm{NOR}(Z,W))))\\&\quad=\mathrm{NAND}(\mathrm{NOR}(X,Y),\ \mathrm{NOR}(Z,W))\end{aligned}$$

が誘導できる。

2）**図** 4.1 と同じ結果が出れば OK。

3）**図** .1。

4.3　**図** .2。

4.4　**図** .3。半加算器を組合せで記載したが，AND，OR，NOT にするには，half-adder の部分に**図** .2 をはめこむ。

4.5　NAND は入力数個の p-MOS による並列回路と，同数の n-MOS による直列回路で構成されるが，これが 1 入力であると，それぞれ 1 素子だけの並列，直列回路になり，結局，直列も並列もない 1 素子だけの NOT 回路そのものになる。

NAND の真理値表は入力がすべて 1 のときにのみ，出力は 0 となり，それ以外の入力の組合せでは 1 となる。これが 1 入力となると，入力が 1 のときにのみ，出力は 0 となり，それ以外，つまり入力が 0 のときには 1 となる。これは NOT の真理値表である。

4.6　**図** .4。

In	S	S'	$Take$	$Ticket$
0	0	0	0	0
1	0	1	1	0
0	1	1	0	0
1	1	0	1	1

図.5　20 円券売機の状態遷移表

5 章

5.1　内部状態は二つで十分であるので，これらを 0 と 1 とすれば，**図** .5 のような状態遷移表が得られる。

5.2　状態遷移図は**図** .6。状態遷移表は略。

5.3　まず**図** 5.8 の上図の場合，ϕ_1 のタイミングで，レジスタの内容は Wt が 1 ならば外部入力により，Wt が 0 ならばレジスタ自身の内容により書き換えられることを確認する。続いて同じく ϕ_1 のタイミングで，Wt も Rd も 1 のとき，回路は左右が独立に動作し，左半分は外部入力で書き換えられるが，出力は右半分の論理状態でのみ決定されることを確認する。

5.4　プリチャージ回路のみ**図** .7 に示すが，全サイズは通常の c-MOS 回路の半分以下である。

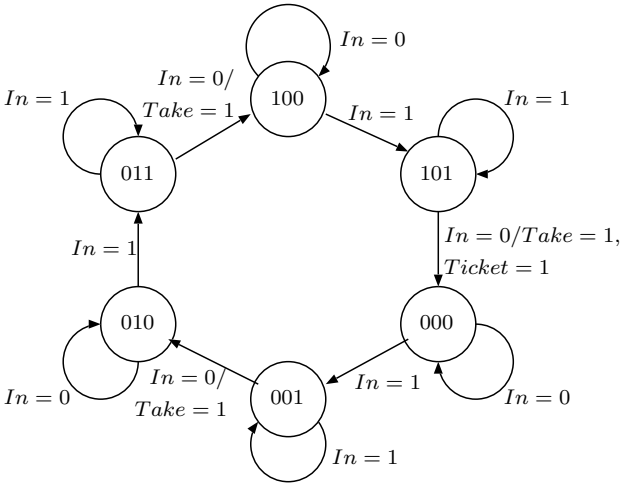


図.6　In がすぐには変化しない場合の 30 円券売機の状態遷移図

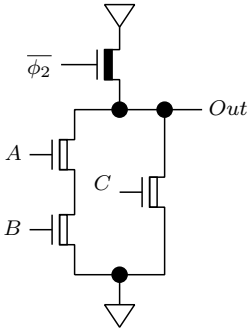


図.7　$\mathbf{NOR}(\mathbf{AND}(A, B), C)$ を出力するプリチャージ回路

10 進	2 進	16 進	10 進	2 進	16 進
0	0000	0x0	8	1000	0x8
1	0001	0x1	9	1001	0x9
2	0010	0x2	10	1010	0xA
3	0011	0x3	11	1011	0xB
4	0100	0x4	12	1100	0xC
5	0101	0x5	13	1101	0xD
6	0110	0x6	14	1110	0xE
7	0111	0x7	15	1111	0xF

図.8　4bit 符号なし整数の 10 進，2 進，16 進対応表

6 章

6.1　**図** .8 参照。

6.2　**図** .9 参照。

6.3

$$\begin{array}{rrr} & 1011 & (11) \\ + & 0111 & (7) \\ \hline & 10010 & (18) \end{array}$$

となり，5bit 目に 1 が立っている。10 進の計算で 16 以上になっているので，それからもオーバフローがわかる。

6.4　**図** .10 で，全領域の 1/2 強でオーバフローが起きない。

6.5　**図** 6.3 になればよい。

6.6　**図** .11 で，全領域の 3/4 強でオーバフローが起きない。

6.7　結果が正しいことを確認できればよい。

6.8　**図** .12 で，全領域の 76/256=0.3 でオーバフローが起きない。ビッ

10 進	2 進	16 進	10 進	2 進	16 進
−8	1000	0x8	0	0000	0x0
−7	1001	0x9	1	0001	0x1
−6	1010	0xA	2	0010	0x2
−5	1011	0xB	3	0011	0x3
−4	1100	0xC	4	0100	0x4
−3	1101	0xD	5	0101	0x5
−2	1110	0xE	6	0110	0x6
−1	1111	0xF	7	0111	0x7

図.9　4bit 符号あり整数の補数表現 (10 進表現以外は補数)

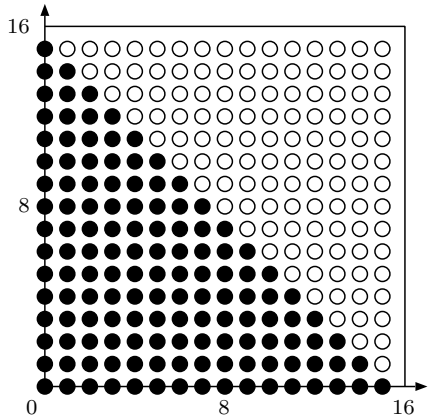


図.10　4bit 符号なし整数の加算でオーバフローを起こさない領域を黒丸，起こす領域を白丸で示す。

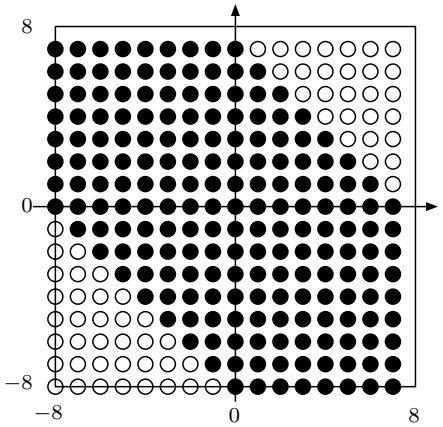


図.11　4bit 符号あり整数の加算でオーバフローを起こさない領域を黒丸，起こす領域を白丸で示す。

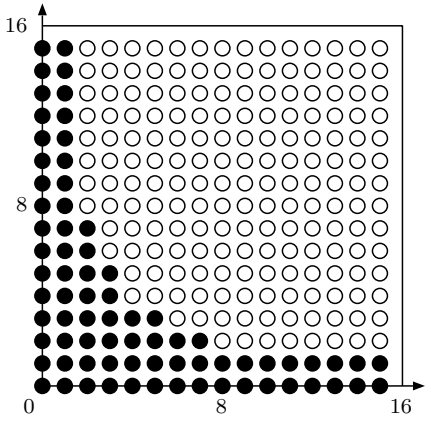


図.12　4bit 符号なし整数の乗算でオーバフローを起こさない領域を黒丸，起こす領域を白丸で示す。

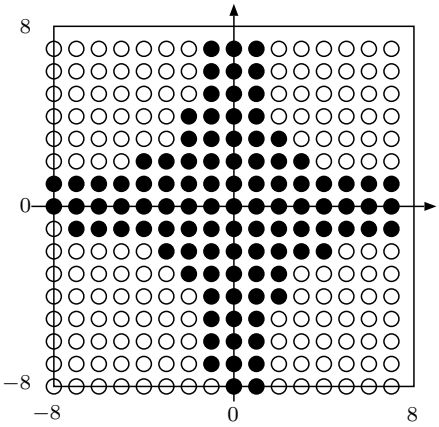


図.13　4bit 符号あり整数の乗算でオーバフローを起こさない領域を黒丸，起こす領域を白丸で示す。

ト幅 n のとき，この領域の面積比はおよそ $(2+n\log 2)/n^2$ となるが，この式に n=4 を代入すると，やはりほぼ 0.3 となる。n が非常に大きいと，この比は限りなく 0 になることに注意。

6.9　**図** .13 で, 全領域の 101/256=0.39 でオーバフローが起きない。ビット幅 n のとき，この領域の面積比はおよそ
$2(1+(n-1)\log 2)/n^2$ となるが，この式に n=4 を入れると，やはりほぼ 0.38 となる。n が非常に大きいと，この比は限りなく 0 になることに注意。

7 章

7.2　A は出力スイッチのみ ON，B は入力スイッチのみ ON，そして C は両スイッチとも OFF であればよい。

8 章

8.2 プログラムの書き方については知らない人が多いかと思うが，以下のプログラムを読んでみると，何をしようとしているかわかるであろう。if 文はすぐわかるであろう。do 文はどんな条件でも，必ず 1 回はループ内を実行する場合に使用する。一方，while 文は条件が不成立の場合は，1 回もループ内を実行しない場合に使用する。

1) テキストのコピー (do 文)

```
i = 0x1000;        \\ 格納元の先頭アドレスを設定
j = 0x2000;        \\ 蓄積先の先頭アドレスを設定
do {               \\ ループの開始
    load(x, i);    \\ 格納元の先頭アドレスの内容を
                   \\   読んで，x レジスタへ入れる
    store(x, j);   \\ x レジスタの内容を蓄積先の
                   \\   アドレスへ書き出す
    i = i + 1;     \\ 格納元のアドレスを一つ進める
    j = j + 1;     \\ 蓄積先のアドレスを一つ進める
} until (x == 0x0000)
                   \\ x レジスタが 0x0000 なら終了
```

2) 絶対値の計算 (if 文)

```
if (x < 0) {       \\ if 文: データが負ならば
    x = -x;        \\ then 文: 負ならば符号反転する
} else {           \\ else 文: それ以外はそのまま
    x = x;         \\   (この行はなくてもよい)
```

```
}
```

3）数の合計計算 (while 文)

```
i = 0x1000;        \\ 格納先アドレスの設定
y = 0;             \\ 合計用 y レジスタをクリア
load(x, i);        \\ メモリーからデータを格納する
while (x > 0) {    \\ x が正の間は以下の作業を行う
    y = y + x;     \\ y に x を加える
    i = i + 1;     \\ 格納先アドレスを一つ進める
    load(x, i);    \\ メモリーからデータを格納する
}
```

9 章

9.2　ALU は入出力を合わせてバスと 3 箇所で接続を必要とする。バス 1 系統では，そのうち一つとしか通信ができないので，最低，2 個のレジスタを必要とする。例えば，2 個の入力にそれぞれレジスタを用意すれば，まず二つのレジスタに入力データを入れ，出力を一気に計算して目的の汎用レジスタへ送り込めばよい。

同様に，バス 2 系統では，1 個のレジスタが必要となる。例えば出力レジスタのみあれば，2 系統のバスから同時に入力を与え，計算結果をいったん出力レジスタに蓄え，次のタイミングで送出すればよい。

バス 3 系統では，同時に 2 系統のバスに入力を与え，同じタイミングで残る 1 系統のバスに出力を送出すればよい。

In	S	S'	Out
	fni	dec	μPAddrBus$\leftarrow \mu PC$, $\mu PC = \mu PC + 1$
μP==’out’	dec	out	IR$\leftarrow$options
μP==’jp’	dec	jp	μPC Reg$\leftarrow$jumpAddr
	out	fni	Out
$FlagR$	jp	fni	$\mu PC \leftarrow$addr
$\overline{FlagR}$	jp	fni	

図.14　μP 制御回路

9.3　A レジスタ (または B レジスタ) の内容をそのまま出力へ送ればよいから，**図** 9.6 の A(または B) と同じ設定の $K = 0000$，$P = 1100$(または 1010)，$R = 1100$ とすればよい。

9.4　ド・モルガンの法則を使うと $\mathrm{NOT}(A \oplus B) = \mathrm{NOT}(A \cdot \overline{B} + \overline{A} \cdot B) = \mathrm{NOT}(A \cdot \overline{B}) \cdot \mathrm{NOT}(\overline{A} \cdot B) = (\overline{A} + B) \cdot (A + \overline{B}) = \overline{A} \cdot A + \overline{A} \cdot \overline{B} + A \cdot B + B \cdot \overline{B} = \overline{A} \cdot \overline{B} + A \cdot B$ なので，$K = 0000$，$P = 1001$，$R = 1100$ とすればよい。

10 章

10.2　本文にもあるように，内部状態は 1 個である。それを 0 とすると，状態遷移表は

S	S'	$Add1$
0	0	1

と 1 行である。実は内部状態を 1 とするほうが回路は簡単になる。

S	S'	$Add1$
1	1	1

この場合，余計な回路をすべて削ぎ落すと，D-フリップフロップの出力と入力をループにして，その接続線から $Add1$ を実行する制御線を分岐すればよい。

10.3　作業の結果が一致することを確認できればよい。

10.4　状態遷移表があれば，それからシークエンス回路を得るのは容易である。内部状態は全部で 6 個あり，その 6 個から出る 8 本の矢印ごとに 1 行を対応させた状態遷移表を作成すればよい。これができれば，正解である。

しかし，ϕ_1 と ϕ_2 の両サイクルをうまく使うことにすると，**図** .14 に示すように，より小さな状態遷移表で済む。その場合，フェッチ，デコード，実行の四つのサイクルの 4 個の開始点の丸のみを内部状態とすれば，十分であることがわかる。これらを順次 fni，dec，out，jp としよう。表の出力は ϕ_2 に出力されるが，コンマ以後は ϕ_1 のタイミングで出力される。

この表では入力，出力，内部状態がすべて文章で表記されているが，例えば 4 個の内部状態を 00，01，10，11 に対応させるなど，2 進化は容易である。なお，'=' は代入，'==' は左右が等しいことを意味する。

11 章

11.1　$\tau \propto \tau_0 \propto (1/k^2)/(1) = 1/k^2$ より $f_c \propto k^2$。$C \propto C_g \propto 1/k$ より $P \propto f_c C_g V_h^2/2 \propto (k^2)(1/k)(1^2) = k$。

このように，高速にはなるが，FET 当たりの消費電力はどんどん大きくなる。

f_c は無理して上げる必要はないので，$f_c \propto k$ ぐらいに抑えることにすると，$P \propto 1$ とできるが，これでも，集積度が上がっていくので，面積当たりの消費電力は深刻な問題となる。

f_c を上げないことにすれば，FET 当たりの消費電力は $1/k$ に比例して下がり，面積当たりの消費電力は k に比例して増加する程度に抑えられる。現在は，この辺りでしのいでいるのである。

著者紹介

●岡部　洋一●（おかべ・よういち）

1943 年	東京に生まれる
1967 年	東京大学工学部卒業
1972 年	東京大学大学院工学系研究科 (工学博士)
1972 年より	東京大学講師，助教授，教授
2006 年より	放送大学教授，副学長，理事
2011 年	放送大学長
専攻	電子工学・情報工学
主な著書	超伝導エレクトロニクス (1985 年，共著，オーム社)
	電気磁気学基礎論 (1988 年，共著，電気学会)
	絵でわかる半導体と IC (1994 年，編著，日本実業出版社)
	素人の書いた複式簿記 (2001 年，単著，オーム社)
	電磁気学の意味と考え方 (2008 年，単著，講談社)

放送大学教材　1570102–1–1411（テレビ）

改訂版 コンピュータのしくみ

発　行　　2014 年 3 月 20 日　第 1 刷
著　者　　岡部洋一
発行所　　一般財団法人　放送大学教育振興会
〒 105-0001　東京都港区虎ノ門 1–14–1　郵政福祉琴平ビル
電話　03–3502–2750

Printed in Japan ISBN978–4–595–31500–8 C1355