第1章　計算機システムの構造と原理

第1章ではまずものごとのはじめとして、計算機というのはそもそもの何をする装置なのか、という事柄からはじめて、それを大規模に効率よく行うためにはどうしたらいいか、という観点から計算機システムの構造を解説する。「計算機はもちろんよく目にするが、その中はどうなっているのかさっぱり分からない」という人にも、きっと納得していただけるはずである。

1.1　計算機とは何をする装置か?

まず、この節の表題どおり、「計算機というのは何をする装置か?」という質問を投げかけてみたい。あなたならどのような答を考えつかれるか? たとえば次のようなものか?

△ 計算機とは、計算をするための装置である。

それだったら、計算機と電卓は同じものか? そうではないでしょう?

ここで「大規模な」とか「大量の」とか「高速に」とつけ加えようとする方は、残念ながら「いまいち」である。確かに世界最初の電子計算機は数表のようなものを大量に「計算」するために作られたのだけれど、現在の計算機の用途としては「計算」はごくマイナーな用途である。たとえばJRの切符を予約したり、建築図面を描いたり、英文抄録をまがりなりにも和訳したりすることはあまり「計算」らしくない。そこであなたは次のような答を考えつかれるかも知れない。

○ 計算機とは、情報を処理するための装置である。

まあ、確かにそうなのだが。しかし、この定義ではあまりにもばくぜんとしていませんか。「情報」とは何か? 「処理する」とはどうすることか?

そこで、唐突だが2進数が出てくる。つまり、世の中にはさまざまな情報が満ちあふれているが、その最小単位は「はい」「いいえ」、「ある」「ない」、のように2つのうちどちらである、という情報だと考えることができそうである。これをもっとコンパクトに書くため「1」「0」の文字を使うことにしよう。この情報の最小単位のことを「ビット (bit)」と呼ぶ。

世の中の (少なくとも計算機で扱えるような) 情報はすべて、この最小単位であるビットを複数個組み合わせて現すことができる。たとえば任意の自然数を現そうと思えば、その数を 2 進数表記してやればよい。普通の (10 進数の)N 桁表記を

$$D_N D_{N-1} D_{N-2} \ldots D_2 D_1 D_0$$

(ただし D_i は 0〜9 のいずれか) とすると、この表記が現す数は

$$D_0 \times 10^0 + D_1 \times 10^1 + D_2 \times 10^2 + \ldots + D_N \times 10^N$$

となる (言い替えれば、D_0 は 1 の桁、D_1 は 10 の桁、D_2 は 100 の桁、…、D_N は 10^N の桁、ということですね)。これと同様に、2 進表記

$$B_N B_{N-1} B_{N-2} \ldots B_2 B_1 B_0$$

ただし B_i は 0〜1 のいずれか、とするとそれが現す数は

$$B_0 \times 2^0 + B_1 \times 2^1 + B_2 \times 2^2 + \ldots + B_N \times 2^N$$

つまり B_0 は 1 の桁、B_1 は 2 の桁、B_2 は 4 の桁、…、B_N は 2^N の桁となる。たとえば 10 進表記での 17 は、2 進だと `10001` となる。そうしち面倒くさく考えなくても、長さが 5 の 0 と 1 の列であれば、$2^5 = 32$ 通りのうちどれか、という情報を現すことが可能なのはおわかりであろう。

ということで、本書では以下「情報」を「ビット列」で現せることにする。そうすると、計算機にはビット列を与えることになる。そして、計算機は内部でそれを「処理」した後、結果の「情報」をやはりビット列の形で返してくれるようにできるだろう。「処理する」という言葉はあいまいな感じがするので、もっと直接的に「加工する」といい直して、以下では

◎ 計算機とは、ビット列を加工するための装置である。

ということにする。たとえば、`10001` というビット列と `00100` というビット列を与えて「足し算」という加工をしてもらうと `10101` というビット列が得られるわけである。

ここで注意しておきたいのは、それぞれのビット列が何を意味しているか、という「解釈」のしかたはあくまで人間にまかされているということである。上の例はそれぞれのビット列を「数を 2 進数表現にしたもの」と思えば 17 と 12 を加えた値 29 を求めていると解釈できるが、ビット列の各桁が友人 A、B、C、D、E のそれぞれについて「最近一緒に飲みにいったかどうか」および「一緒にスキーにいったかどうか」を意味するものだと思えば「最近一緒に飲みにいったかまたはスキーにいった人」を求めているとも解釈できるかも知れない。

そして、計算機は与えられたビット列を決まったやり方で加工して結果のビット列を求めるという点では高い能力を持つが、そのビット列をあなたが

解決したいと思う問題にあわせて解釈した時実際に正しい解になっているかどうかはそれとはまた別のことがらであり[1]、むしろそちらの方が現実の問題を解く上ではむずかしい課題である。

とりあえずその、むずかしい方はおいておいて、「ビット列を加工する」にはどんなしかけがあればよいかについて次節以降で考えてみよう。

1.2 計算機のしくみ

1.2.1 ビットの表現

計算機を作るには、まずビット値をどうやって表現するかを決めなければならない。筆者は買ったことがないが、「パチンコ玉コンピュータ」というおもちゃがある。あれは、「玉のある/なし」を使ってビット値を表現しているわけである。とはいえ、パチンコ玉では場所を取るし、操作するのにも時間がかかってしまう。ここはやはり、「電気」を用いて表現するのがよさそうである。

たとえば、ビットを入力するのにはスイッチを用い、出力するのにはランプを用いるものとしよう。図 1.1 のように配線すれば、入力 A を 1 にすれば出力 B も 1 になるし、入力が 0 なら出力も 0 である。

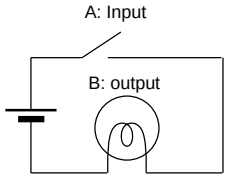

図 1.1: ランプとスイッチによる入力と出力

これでは出力が入力そのまんまだからあまりおもしろくない。そこで、入力を A と B の 2 つとして、「A と B が両方 1 のとき出力 C が 1」というのと「A と B の少なくとも一方が 1 のとき出力 C が 1」というの、つまりブール代数でいう and 演算と or 演算にしてみたのが図 1.2 である。何のことはない、理科の実験でやる「直列と並列」ですね。

では、この方法で回路を組み合わせればビットのどんな加工でもできるか、というと残念ながらそうはいかない。早い話が「入力 A が 1 のとき出力 B が 0」つまり not 演算からして作ることができない[2]。

[1] たとえば「親しい友人を求める」という問題を「飲みにいった」「スキーにいった」という情報だけに基づいて解こうとしてよいかどうか、ということ。

[2] スイッチについている表示を書き換えて「0」と「1」を反対にしてしまうという手もあるが、ちょっとずるい、というか not 演算を作ったとはあまりいえない。

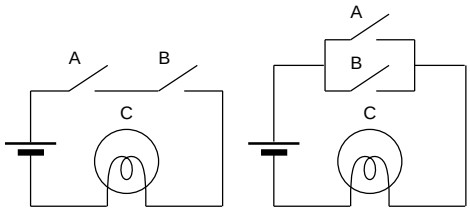

図 1.2: ランプとスイッチによる and と or

1.2.2　演算素子とゲート

そこで次に、リレーを使うことにする。リレーというのは手のかわりに電磁石で on/off するスイッチのことで、ここでは図 1.3 のようにふだんはバネの力でスイッチが閉じていて、入力回路に電流が流れると電磁石の力でスイッチが切れるようなものを使うことにする。これで確かに not 回路をつくることができた。

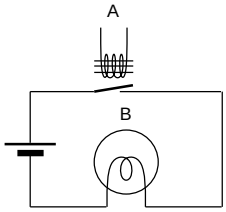

図 1.3: リレー回路

そのほかの種類の演算もリレーで行わせることができる。たとえば図 1.4 のように 2 つの入力回路 A、B の両方に電流が流れたときはじめてスイッチが切れるようにバネを調節しておけば、 not (A and B) の演算 (nand 演算と呼ばれる) を行わせることができる。

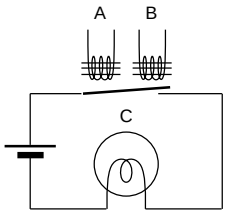

図 1.4: リレーによる nand 回路

組み合わせ回路が複雑になってくると、いちいちリレーの形を書くのは煩

雑である。そこで、もっと抽象化した記号を用いて、電池の両極につながる線も省略して on/off される線だけを書くことにする。具体的には図 1.3、図 1.4 を図 1.5 のように書く。このような抽象化された演算素子のことを「ゲート」と呼ぶ。図 1.5 では上のものは「not ゲート」、下のものは「nand ゲート」である。

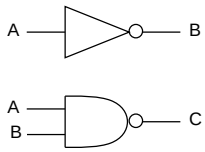

図 1.5: 抽象化された回路記号

ところで、リレーのもう 1 つの重要な性質は、エネルギーを増幅できることである。一般に 1 つのスイッチが流すことのできる電流は限られているので、スイッチ 1 個にたくさんの電球をつなぐことはできない。また、スイッチには固有の抵抗があるので、100 個の入力の and をとろうと思っても 100 個のスイッチを全部直列につないだら十分な電流が流れないかもしれない。しかし、リレーがあれば少ない電流で大きな電流を on/off できるので、これらの問題を解決することができ、十分高度なビット加工を行う回路を組み立てることができる。実際、電子計算機ができる以前に、リレーを演算素子とした計算機 (リレー計算機) が作られていたこともある。

1.2.3 VLSI

リレー計算機はスイッチの鉄片がかちゃかちゃ動くのに時間がかかるし、接触不良が多く信頼性が低いという問題もある。そこでこれらの問題を解決して、高速かつ信頼性の高い情報処理を可能にしたのが電子計算機である。同じ電子計算機でもゲートに使われる演算素子が真空管 (第 1 世代)、トランジスタ (第 2 世代)、IC(第 3 世代)、VLSI (第 4 世代) のように変遷してきている。ここでは古い話をしている時間はないので、VLSI の原理をごくおおざっぱに説明しておこう。

まず、VLSI を作るにはシリコン (圭素、Si) の単結晶 (ウェハー) を用意する。これ自体は電気を通さないのだが、この上にホウ素やリンの分子をごく微量加えてやる (拡散させる) と、その部分は電気を通すようになる。だから、微細な模様をデザインしてその模様にそって拡散を行えば、好きな形の電気配線がウェハー上に作れるわけである。

ここで、リンを加えた部分は (リンがシリコンより多くの電子を原子の外周に持つため) 電子が余分に生じて電気を流すという性質を持ち (n 型領域)、逆にホウ素では (シリコンより外周の電子が少ないため) 電子の不足 (正孔) が

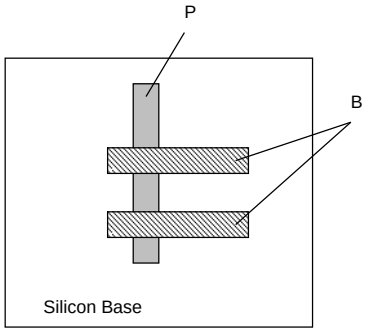

図 1.6: VLSI の構造

生じて電気を流すという性質を持つ (p 型領域)。ここで、断面図が図 1.7 のように、n 型領域が途切れていて間に電極 (これもパターン焼きつけで作れる) を取り付けた形になっている部分を作る。すると、電極が「+」の時にはそのプラス電荷に引き付けられて来た電子がギャップの所に集まって電気が通り、電極が「-」の時には逆に電子がマイナス電荷と反発して逃げてしまうので電気が通らないことになる。

これでつまり、電極の+/-に応じてスイッチが on/off できることになる。これを「電界効果トランジスタ」ないし「FET」と呼ぶ。これはリレーと違って動く部分がないし、微細な領域で低電圧で動作させられるため高速である。なお、n 型の変わりに p 型を使えばこれと反対の性質を持つ (つまり電極が「-」の時に電気が通る) トランジスタも作れ、その両者をペアにして組み合わせて回路を構成する CMOS VLSI が現在の主流になっている。

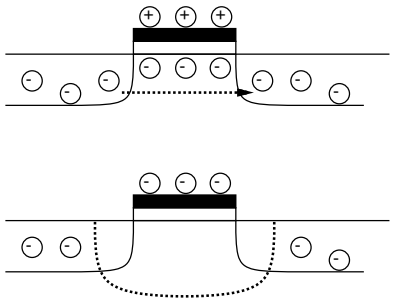

図 1.7: VLSI 上の電界効果トランジスタ

ではどうやってウェハーの上に微細な模様に従った拡散を行わせられるのだろうか? それはまず、模様を普通の大きさで作って、それを写真に取ってフィルムを作る。次に、ウェハーの上に光に感応する樹脂を塗って、フィルムを通した光をレンズで縮小して投射する。そうすると、光があたった部分

だけ樹脂が変質する。あとで洗浄液につけると変質した部分だけが流れ落ちるので、結果として模様通りの幕がウェハーに残る。あとはホウ素やリンをガス化した炉にウェハーを入れると、幕がなくなった部分に模様にそって分子が拡散する。実際にはまずホウ素、次にリンという風にこれを何段階か繰り返して非常に多数のトランジスタを持つ回路をウェハー上に焼き込むのである。これを VLSI という。

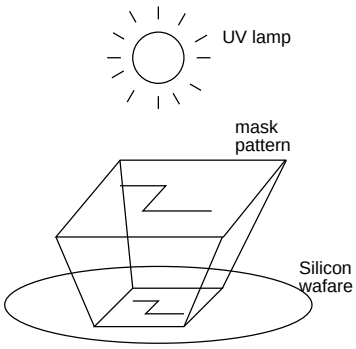

図 1.8: VLSI の製作原理

1.3 計算機と論理回路

1.3.1 フリップフロップとラッチ

さて、ではもう少し具体的に計算機に使われる回路を見てみよう。図 1.9 にフリップフロップと呼ばれる回路を示す。この図で一番左の状態をまず見て頂きたい。A、B はともに 1、上のゲートのもう 1 方の入力は 1 だから上のゲートの出力 C は 0、従って下のゲートの片方の入力が 0 だから下のゲートの出力 D は 1、そこで上のゲートの入力が 1 で、つじつまが合っている。(もしつじつまが合わないと、回路は不安定であり発振 — 状態が振動すること — したりする。)

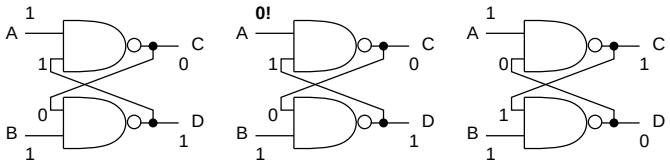

図 1.9: RS フリップフロップ

ここで、図中央のように入力 A をちょっとの間 0 にしたとする。すると上のゲートの出力は 0 → 1 に変化し、その結果下のゲートの出力は 1 → 0 に変化することになる。すると、入力 A が再び 1 に戻った後も、この回路はさっきとは逆に C が 1、D が 0 という状態を保持する。つまり、フリップフロップは与えられた情報 (最後に A と B のどちらが 0 だったか) を保持することができるわけである。

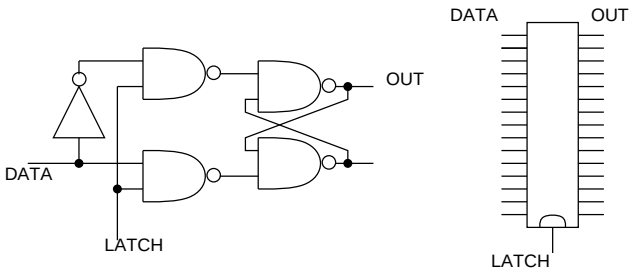

図 1.10: RS フリップフロップ

さて、実際に計算機の回路として使われる場合には図 1.10 左のように前段ゲートを追加して使うのが普通である。この場合、LATCH 線を普段は 0 にしておく。すると前段のゲート出力は 1 なので OUT の状態は変化しない。入力を記憶したい場合には LATCH 線を 1 にする。すると DATA 線の状態がフリップフロップに記憶され、OUT は DATA と同じになる。LATCH を 0 に戻すとその状態は固定され、あとは DATA が変化しても記憶された状態が保持される。このような回路を「入力をそのまま固定する」ことから「ラッチ」と呼ぶ。通常はこのような回路をデータ幅 (32 ビット CPU なら 32 個) ぶん並べて 1 本の LATCH 線で制御する。(これを図 1.10 右のように描く。)

1.3.2　記憶装置 (メモリ)

計算機がビット列を加工していく上では、大量のデータを扱う必要がある。そこで、もっぱら大量のビット列を格納しておくことだけを行う VLSI チップが多く作られ売られている — これをメモリチップという。

メモリチップの中で実際にビットを格納する部分 (メモリセル) には、前述のフリップフロップを使えばよい。この方式は電源を切らない限り特に何もしなくても記憶内容が静的に保持されるので SRAM(static random access memory) と呼ばれる。SRAM は 1 セルあたり 4 つのゲートがいるので、記憶容量の点からは不利である。もう 1 つの主要な方式は DRAM(daynamic RAM) であり、これは小さなコンデンサ (電荷を蓄えるような構造) を使って記憶を保持する。ただし、コンデンサに蓄えた電荷は時間がたつと自然放電してなくなってしまうので、定期的に蓄えた値を読み出しては書き込む必要

がある。これをリフレッシュと呼び、DRAM のチップ上にはそのための回路が組み込まれている。

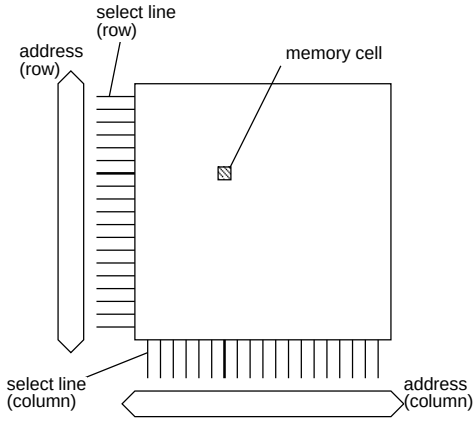

図 1.11: メモリチップの構成

DRAM でも SRAM でもメモリチップの内部は概念的には図 1.11 のようになっていて、メモリセルが格子状に配列される。このなかで特定のセルをアクセスするためには、行と列の選択線をそれぞれ 1 本だけ「1」にすることでその交点にあるセルが選ばれてアクセスされる。

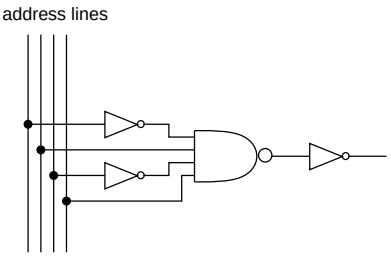

図 1.12: アドレスデコーダ

アドレス線から選択線のどれか 1 つを選ぶ部分をアドレスデコーダと呼ぶ。図 1.12 に、アドレス線が「0101」の時だけ出力が 1 になる回路を示した。(N 入力の nand ゲートは VSLI 上では問題無く作れる。)

実際の DRAM では 1 チップの容量が 16Mbit 程度 (1M $=$ 1024 $\times$ 1024) なので、アドレス線は 24 本 (縦横 12 本ずつ) ということになる。このようなチップを 8 個とか 32 個、小さな板の上に取り付けたものを SIMM(single inline memory module) と呼び、これを単位として本体に取り付けることが多い。これは、計算機のメモリは 1 バイト (8 ビット) とか 4 バイト単位でアクセスされることが普通なので、その数だけメモリチップを並べておいて同

時にアクセスする方が速くて簡単だからである。

1.3.3　演算回路

データを記憶しておく回路ができたので、次に演算を施す回路を考えよう。一般に、n ビットの入力から任意の演算を行って 1 ビットの出力を行う場合、その演算規則を「積和標準形」つまり入力のうちいくつか (およびその not) に and 演算を行った「項」を複数 or する形で現すことができる。

例えば 2 進数 2 桁の足し算を考えよう。入力 (2 桁 ×2) と出力 (2 桁および桁上がり 1 桁) の関係を現すと表 1.1 のようになる。ここで 3 つの出力をそれ

表 1.1: 2 桁どうしの足し算

A_1	A_0	B_1	B_0	C	O_1	O_0
0	0	0	0	0	0	0
0	1	0	0	0	0	1
1	0	0	0	0	1	0
1	1	0	0	0	1	1
0	0	0	1	0	0	1
0	1	0	1	0	1	0
1	0	0	1	0	1	1
1	1	0	1	1	0	0
0	0	1	0	0	1	0
0	1	1	0	0	1	1
1	0	1	0	1	0	0
1	1	1	0	1	0	1
0	0	1	1	0	1	1
0	1	1	1	1	0	0
1	0	1	1	1	0	1
1	1	1	1	1	1	0

ぞれ積和標準形で現すと

$$
\begin{aligned}
O_0 &= A_0\overline{B_0} + \overline{A_0}B_0 \\
O_1 &= \overline{A_0}A_1\overline{B_1} + \overline{A_0A_1}B_1 + A_1\overline{B_0B_1} + \overline{A_1B_0}B_1 + A_0\overline{A_1}B_0\overline{B_1} + A_0A_1B_0B_1 \\
C &= A_1B_1 + A_0A_1B_0 + A_0B_0B_1
\end{aligned}
$$

となる。なお、AB は A と B の and、$A+B$ は A と B の or、$\overline{A}$ は A の not を表している。

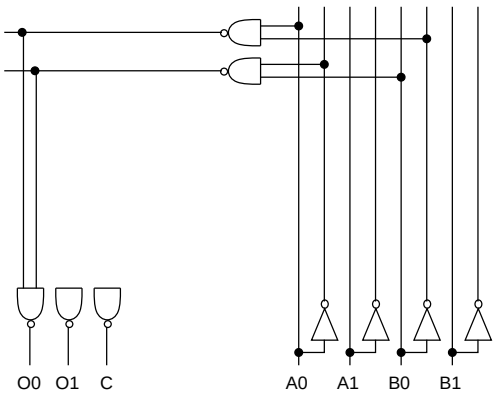

図 1.13: ゲートアレイによる論理回路

これを論理回路にするには例えば図 1.13 のような規則的なやり方が可能である。ここではまず、各入力からそのままと not したものを作る。次に、各論理式の項 (and でつながったもの) について 1 個ずつ nand ゲートを用意し、その入力を対応する入力線 (またはその not) につなぐ。(そのためには N 入力の nand ゲートが必要だが、これを VLSI 上の素子として作るのは特に問題なくできる。) そして、各出力に対しても nand ゲートを用意し、各項に対応するゲートの出力をここに接続する。

このようにして任意の入力と出力の関係を実現する論理回路を組み立てることができる。「足し算」「引き算」など複数の演算を切り替えたければ演算の種類を指定する制御線も入力として扱い、それを含めた論理式から設計すればよい。

ただし、大きな桁数の加算や乗算を一段で行わせようとすると回路が巨大になってしまい、また実際の計算機で使われる演算器は高速性が要求されることから、本当の演算器 (ALU — Arithmetic Logical Unit) はこのような規則的な設計とは別の方法で構成した論理回路を用いる。

1.3.4 クロックとシーケンサ

ここまでのところで一応、データを格納したり演算したりする部品はできた。しかし、これらの部品を組み合わせて思ったように動かす方法についてまだ説明していない。

皆様は PC などについて「クロックが 66MHz の…」などという言葉を聞かれたことがあるだろう。クロックは名前の通り「時計」であってシステム全体を動かすタイミングを制御する。具体的には図 1.14 のように決まった周期で 0/1 を反復する出力を出すような回路である。(この反復回数が 1 秒間に

66×10^6 回だと 66MHz。) クロックの回路はいわゆる発振器で、どちらかというとアナログ回路に属する。

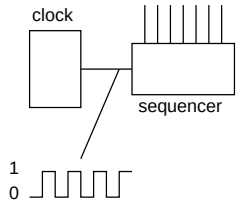

図 1.14: クロックとシーケンサ

次に、クロックの出力をシーケンサに与える。シーケンサはクロックのきざみに従って、一定の規則に従って出力を 0/1 に切り替えるものである。(中の回路までやっていると大変だから略した。)

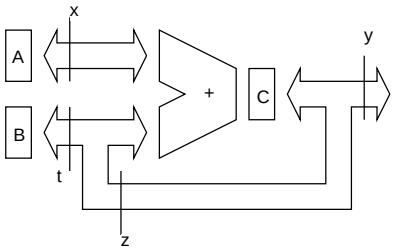

図 1.15: 簡単な計算システムのダイアグラム

さて、これでどうやって部品が組み合わせられるのだろう? 例えば 1.15 のような回路を考えてみる。ここで A、B、C はラッチで真ん中のは加算回路である。そして、たて線のところはデータ線の接続を on/off するゲートが設けてある (データ線の幅は 4 ビットでも 32 ビットでもなんでもよい)。ここで A、B に入力装置からデータをセットできたとする。ここで B の値に A の値の 2 倍を加えた値を最終的に出力するには

- まず x と t を on にしてラッチ C に $A + B$ を計算する。
- t と y を off、z と x を on にして C の値と A の値を計算したものを再度 C に設定する。
- 最後に z を off、y を on にして C の値を出力する。

という順序で制御を行えばよい。それはシーケンサが x〜t の 4 本の線をクロックのタイミングに従って on/off することで行えるわけである。

1.4 計算機システムの構成

1.4.1 プログラムの概念と CPU の動作

さて、前節最後に示した様な方法では、せっかく組んだ回路にある特定の計算ないし処理しか行わせることができない。これではもったいない。ではどうしたらいいだろうか?

答えは、各制御線の on/off をシーケンサ回路として作ってしまう代わりに、**メモリからビット列を読み出してきてその 0/1 に従って制御線を on/off すること**である。このビット列を**命令語**と呼ぶ。ある命令語が実行し終ったら、メモリの次の場所にある命令語を持って来る。これを繰り返すことで、メモリに書いてある命令語の列、すなわちプログラムを順番に実行していくような回路が作れることになる。これが計算機の CPU(中央処理装置) のやっていることである。

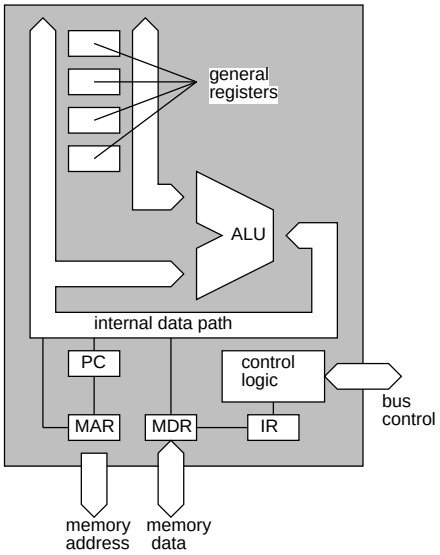

図 1.16: 計算機 CPU のブロックダイアグラム

ごく簡単化された CPU のブロックダイアグラムを図 1.16 に示す。ここで、メモリアドレスレジスタ (MAR) にメモリアクセスを行いたい番地を入れて CPU から外部に送り出すと、メモリアクセス (読み出しまたは書き込み) が行える。読み出しの場合はメモリから読まれたビット列はメモリーデータレジスタ (MDR) に格納される。書き込みの場合には MDR の内容がメモリに転送される。

CPU は 1 実行サイクルごとにメモリから命令を読み出し、それを命令レジスタ (IR) にセットする。命令を読み出す番地はプログラムカウンタ (PC) に

よって指定される。PC には 1 命令読み出すごとに内部の値を命令の大きさぶん増やす回路が組み込まれているので、これによって次々と連続した命令列を実行することができる。

CPU 内部の制御線すべてを命令のビットと対応させると 1 命令がひどく大きくなってしまうので、普通の CPU では命令はもっと「詰めあわせた」形をしている。これを適当に分解して各制御線を on/off するのが制御ロジックである。

CPU 内部には、データとしてのビット列を入れておくためのレジスタ群もある。これを汎用レジスタと呼ぶ。命令の種類としては、汎用レジスタとメモリとの間でデータをやりとりするもの、汎用レジスタの値を ALU に送って演算し結果を再び汎用レジスタに格納するもの、次に実行する命令の番地を設定するもの、などがある。最後のものは命令の実行が連続した列から「飛び出して」別の位置に移るのでジャンプ命令などと呼ばれる。ジャンプ命令を実行するには単に PC に次の命令の番地を設定すればよい。これらはすべて、CPU 内部のデータパスを通してデータを転送することで行える。

以上をまとめると、CPU の動作は次のようになる。

- PC の内容を MDR に転送して命令の読み出しを開始し、PC の値は次の番地に進める。
- 命令が読めて来たら IR に入れてその内容を分解する。
- 制御ロジックによって命令の実行を開始する。

これを無限に繰り返すのが CPU の動作である。そして反復動作自体はクロックとシーケンサによって制御されるわけである。

1.4.2　計算機システムの構造

計算機システムは CPU だけからできているわけでは当然ない。メモリが必要なことは既に述べたけれど、それ以外に入出力装置などもある。これらは通常、図 1.17 に示すようにバスによって接続されている。このバスは CPU チップ内部のバスよりはずっと低速だが汎用的で、様々なボードを接続することができる。

WS クラスまでの多くのシステムでは、ビデオコントローラ (画面)、シリアルアダプタ (キーボード、マウス)、ディスクコントローラなどもメモリと同じバスに接続されている。ただし CPU とメモリの間の転送速度は重要なので、メモリだけ別途 CPU と接続されているシステムもある。

図 1.17 にあるように、キーボードや画面やディスクなどは計算機の本体からケーブルが出ていて、その先に実際の入出力機器がつながっているわけだが、計算機内部ではこのケーブルは各入出力コントローラに接続され、コントローラがバスを介して CPU とのやりとりを受け持つようになっているわけである。

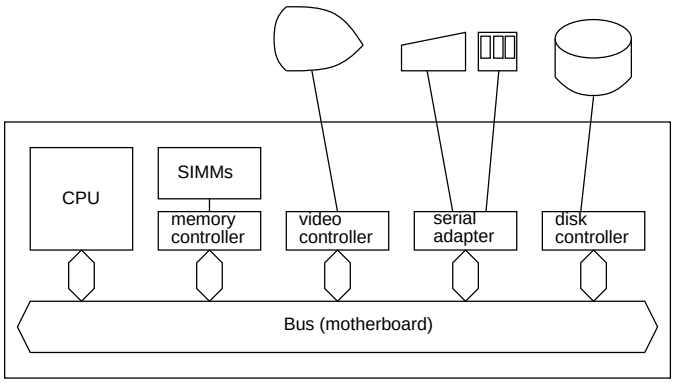

図 1.17: 計算機システムの構造

1.5 まとめと演習問題

この章では計算機システムの目的、原理、構造について、ハードウェア面を中心に一通り説明した。これ以降の章ではすべてソフトウェアの話が中心となるが、その前に「計算機はただの箱みたいに見えるけれどこういうものなのだ」という感触を持って頂ければ幸いである。

演習 1-1. これから演習室で Unix マシンを 1 台開けて中を見ていただくので、その中にどんな部品があるかスケッチし、どれが今回学んだもののうちどれであるかについてできる範囲で整理してみよ。そして、「見分けられた部品の名前」と「それについてどう思ったか」をできるだけ多くの部品についてまとめよ。