（1） CPU の高速化

前回の紹介では、「マシンサイクルの短縮」、「メモリーの階層化」についてご説明しました。マシンサイクルの短縮のためには、半導体の集積技術の高度化が非常に有効に効き、演算のビット幅の拡大が高速化のために画期的な飛躍をもたらしています。また、メモリーの階層化ということで、一番高速な記憶手段であるチップ内の記憶装置にキャッシュメモリーという考え方を導入して、外部の記憶装置の写しを持つことによって高速化を実現しています。

今回の紹介では、「並列化」、「アルゴリズムの工夫」の 2 点について、ご説明したいと思います。前回の 2 つの技術は、集積度を上げたり、メモリー容量を増やしたりと、どちらかというと力技で速くしているという感がありますが、今回は少し小技、というと語弊がありますが、知恵を使った高速化です。

（2） 並列化

並列化というと、マルチ CPU で、同じ演算を同時に実行するとか、それぞれ別の演算をさせて、その結果を統合し結果を得るなど、CPU を並列にいくつか持つというイメージがあると思いますが、一つの CPU の中でも、高速化のために並列化技術を用いています。

ここで、命令の実行のステップをもう一度おさらいしてみます。以下のような処理が必要でした。

- ✓ 命令をメモリーから読み出す。
- ✓ 何をする命令であるか解読する。
- ✓ 処理するためのデータをメモリーから読み出す（メモリーデータを処理する命令の場合だけ）
- ✓ 処理するためのレジスターを読み出す（レジスターデータを処理する命令の場合だけ）
- ✓ データを処理する演算などを実行する
- ✓ 演算結果をレジスターに格納する、またはメモリーに格納する
- ✓ 次の命令のメモリーアドレスを準備し、次の命令の読み出しに備える。

これらの処理をもう少し詳細に見てみましょう。これらの処理は同時にできるもの、順に処理をしなければならない関係にあるものがあります。例えば、命令を読んだ後でないと命令の解読はできません。メモリーのデータを読んだ後でないと、演算はできません。などです。これらを整理してみます。

命令を読み出す。この処理はすべての始まりですので、それ以降の処理と同時に実行するのは難しそうです。命令読み出しは専門用語で命令フェッチまたはインストラクションフェッチと呼びますので、この処理を IF（インストラクションフェッチの略）と呼ぶことにします。

次に命令の解読処理ですが、これは命令コードをもとにして、その命令を処理するマイクロプログラムの先頭アドレスを生成するという処理を行うのでした。そのあとにメモリーからデータを読み出す処理をしなければなりませんが、そのアドレスの計算をする必要があります。このアドレス計算処理は、命令の解読処理と同時に実行できそうです。命令解読とアドレス計算の処理を ID(インストラクションデコードの略)と呼ぶことにします。

次にメモリーからデータを読み出します。命令の読み出しが IF だったのですが、データの読み出しなので呼び方を変えます。処理するデータのことをオペランドと呼びます。ですから、メモリーからデータを読み出すことを専門用語ではオペランドフェッチと言いますので、これを OF（オペランドフェッチの略）と呼ぶことにします。

次は、レジスターとメモリーから読み出したデータの演算と、その結果のレジスターへの格納ですが、レジスターを読んで、演算して、レジスターへの格納までが 1 マシンサイクルに収まるということは前回のマシンサイクルの短縮の説明の中で説明しました。ということで、このステップを、演算の実行を意味する専門用語エクセキュート（Execute：実行という意味）を使い、略して EX と呼びます。

命令の実行を上記の略称を用いて表現すると、

IF→ID→OF→EX

の 4 ステップのマシンサイクルが必要そうです。つまり、このままで考えると一つの命令の実行は 4 マシンサイクル必要で、2 命令ではその倍の 8 マシンサイクルとなりそうです。

ここでご紹介する並列化というのは、この 4 つのステップの処理を並列化しようとするものです。何を並列化しようというのでしょうか？

例えば、L 命令 1→L 命令 2→A 命令→S 命令のように命令が並んでいたとすると、

1． 最初の L 命令 1 の IF が終了し、

2． ID を実行するときに、次の L 命令 2 の IF を同時に実行します。

3． 次は L 命令 1 の OF と L 命令 2 の ID と A 命令の IF を同時に実行、

4． その次は L 命令 1 の EX と L 命令 2 の OF と A 命令の ID と S 命令の IF を同時に実行。

このような具合に、続いている命令の IF と ID と OF と EX を同時に実行するという並列化を実現します。下図参照。これによって、命令の実行時間は見かけ上はライトブルーに塗られている EX の時間だけとみなすことができるのです。もちろん最初の命令だけは 4 ステップかかっているように見えますが、プログラムが走行中はそれぞれの命令実行時間は EX の時間のみとなっているのです。このように、命令実行の処理ステップを並列化し、各ステップでは一度に一つの命令を処理することを命令パイプライン処理と呼びます。

IF	ID	OF	EX
L命令1			
L命令2	L命令1		
A命令	L命令2	L命令1	
S命令	A命令	L命令2	L命令1
	S命令	A命令	L命令2
		S命令	A命令
			S命令

上の例ではたまたま L 命令→L 命令→A 命令→S 命令という組み合わせだったのですが、命令の組み合わせによっては、このパイプライン処理が有効にならない場合があります。分岐命令がある場合です。パイプラインは命令アドレスが+2 ずつとか+4 ずつ進む前提で次の命令、次の次の命令、次の次の次の命令と先に読むことができて、それぞれの解読も先行してすることができましたが、分岐命令があると、実行されない命令を先に読んで解読をしてしまうことになります。このような場合は分岐先の命令で、一からパイプライン処理を始めることになり、EX ステージの実行時間しかかかっていなかったものが、IF～EX までの 4 ステップ分がかかってしまうことになります。このような意味で、プログラムを組む場合は、ループ構造にして分岐を使ったプログラム（ループ構造にする方法はメモリーの使用量を節約できるのでよく使われます）よりは、実行回数分、縦に同じ処理を並べてしまった方が、速度を速めることができます。最近はメモリーをふんだんに使えますので、このような高速化という方法も覚えておいてください。

（3） アルゴリズムの工夫

乗算の処理プログラムを以前、「コンピュータのしくみ（命令、プログラム）（2011 年 10 月）」の中で、フローチャートで示したことがありました。乗数の回数だけ、被乗数

の加算を繰り返すという処理でした。言うまでもなく、乗数によって処理時間が異なりますが、その回数に応じた時間がかかるという、プログラムとしてはお粗末なものでした。本日はその汚名を返上します。

下に、我々が日常行っている 10 進数で掛け算の筆算を示します。

```
   34
 ×  12
-----
   68
  34
-----
  408
```

コンピュータでは何度も言っていますように、2 進数にて処理をします。では、2 進数で、同じように掛け算をするとどうなるでしょうか？4 ビット同士の掛け算を下に示します。10 進数に直すと、6× 5 の掛け算です。

```
    0110
 ×  0101
--------
    0110
   0000
  0110
 0000
--------
 0011110
```

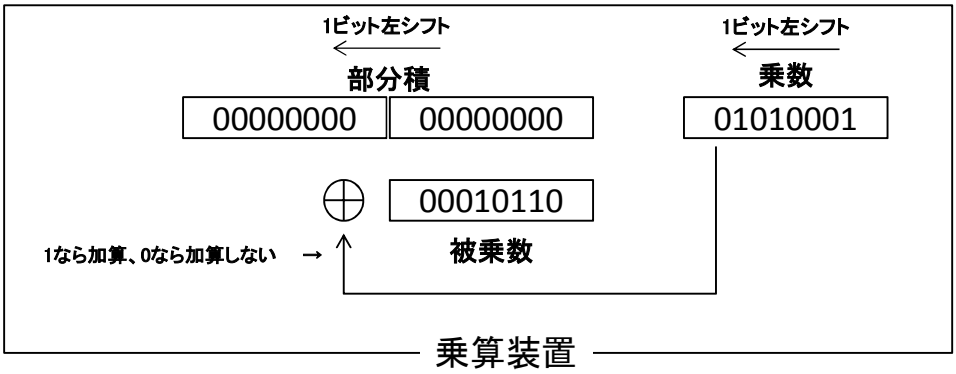
どうでしょうか？

2 進数というのは、数字が 0 か 1 だけですので、1 ならば足す、0 ならば足さない。という具合に、被乗数をその乗数の桁に対応して、部分積を加算するかしないかだけを考えれば済みますので、単純です。下のような乗算装置（8 ビット×8 ビット）というのを考えてみました。

1. 乗数を右上のレジスターに、被乗数を下のレジスターにセットします。
2. 部分積を入れるレジスター（レジスター2 個）をゼロにしておきます。
3. 乗数レジスターの 1 番上のビットが 1 なら部分積レジスターと被乗数レジスターを加算、0 なら何もしません。
4. 部分積レジスターと乗数レジスターを左に 1 ビットシフトします。以降 3 と 4 を乗数のビット数回繰り返します。

2 進数を左に 1 ビットシフトするという処理は、元の 2 進数を 2 倍することになります。ということは、ｎビット左にシフトすると 2 のｎ乗倍と同じことです。今乗数も 1 番上のビットから順に 1 ビットずつ見ていきますから、最終的に最初に加算してできた部分積

は2の乗数のビット数乗の重みをもった部分積となるわけで、上で行った筆算と同じ結果となります。このような乗算装置を備えていれば、乗数のビット数分のステップ数で乗算が実現でき、賢く乗算が実現できたと思いませんか？



もう少し高速化する別の方法を紹介しましょう。

我々人間（日本人としておきます）は、小学校で掛け算の99（くく）を教わりますね。これを使いましょう！ですが、コンピュータですので、2進数ですから、いえいえ、16進数ですから、FF（「ふふ」とでも呼びましょうか）「掛け算のふふ」をコンピュータに覚えさせたらどうでしょうか？つまり、0～Fの16進数同士の積を暗記しておくのです。

下に示す筆算のように16進数一桁ずつの部分積を出して、それをそれぞれの重みを考慮して加算すれば掛け算ができます。かなり高速化できますね。

```
   1B
 × 2A
-----
   6E ←―― B×A
    A ←―― 1×A
   16 ←―― B×2
    2 ←―― 1×2
-----
  46E
```

ところで、「掛け算のふふ」などと言っていますが、これはどうすればいいのでしょう？

ROM（ろむ）というのを聞いたことがありますか？Read Only Memoryの略で読み出し専用メモリーというものです。よくCD-ROMというのは聞いたことがありますよね。これは中身のデータはすでに書かれていて読むことしかできないCDです。コンピュータの中にも半導体素子としてROMが使われます。この「掛け算のふふ」の場合は、8ビットのアドレス（つまり容量256バイト）のROMを用意します。そして、4ビットの2進数（16進数

一桁分）二つ分をアドレスとし、メモリーの内容をその積としておけば、16 進数一桁同士の掛け算がこの ROM をアクセスするだけで、求められます。

これだけでも、結構速くなったと思いますが、さきほど、「日本人としておきます」などと意味深なことを言いましたが、日本の子供は掛け算の 99 ですが、インドの子供たちは 99×99 を暗記しているのだそうです。これはすごいですね。これはなんと呼ぶのかわかりませんが、これも応用して、コンピュータでは「掛け算のふふふふ」を覚えさせましょう！下に示す通り、答えは一発です！この場合はアドレスが 16 ビットの ROM（64K バイト）を使えば実現できます。

```
   1B
 × 2A
-----
  46E
```

以上のように、乗算については、乗算装置のような仕組みや、「掛け算のふふ（ふふ）」などのような仕組みを導入することで、高速化が実現されます。

アルゴリズムの工夫については、これ以外にもまだご紹介したいものがありますが、今回はここまでにして、つづきは次回にご紹介することとさせていただきます。