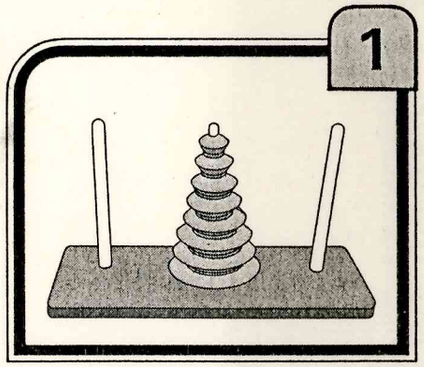
スタック計算機入門 1



デビット・M・バルマン

訳 井田昌之

スタック計算機は，よく知られた「フォン・ノイマン・モデル」とは大変異なった構造をしている．だから，スタック計算機をまったく知らない計算機の専門家が多いのも事実である．われわれは数年にわたってスタック計算機を使用してきた．そこで，たとえば「ゼロアドレス・マシン」という言葉を聞いて，「1語当りゼロビットである」というような解釈をする人がいることを考えると，ときどきスタック計算機というものがもっと普及していたら，と思うことがあった．幸いなことに，COMPUTER 誌のテクニカル・エディタである Jack Grimes も同じことを考えていて，このテーマで特集号を出したいと思っていることがわかった．

インテル社やザイログ社から LSI のスタック・マシンが近く出るという噂[*1]があることから，スタック・マシンの話を扱うのはちょうどよい機会だと思う．またヒューレット・パッカード社が HP 3000 とは異なるスタック・マシンを発表するという噂もある．さらに私は，もしアムダール社がこの分野で興味深い発表を行なうとしても驚きはしないだろう．マイクロデータ社も 32/S 上のシステムを実際に市場化するという政策を発表している．このほかにここ数年のあいだに，いくつかのメーカから高級言語マシンが発表されるかもしれない．

スタック・マシンと高級言語マシンとは同一ではないにもかかわらず，私は，スタック・アーキテクチャが一般的なプログラミング言語の多くにふさわしいものだと信じている．

私は半導体メーカの数社によって，導入されているだろう新しい計算機が，スタック計算機ではなく，Pascal 計算機と呼ばれることを期待する．ご存知のように，Pascal は Algol に似た高級言語で，もとになる Algol が持たなかったいくつかの望ましい特徴を持って，Niklaus Wirth によって開発された言語である[*2]．Pascal は数年前に出てきたものであるが，その普及の理由の一つは，Pascal で書かれた Pascal コンパイラが簡単に手に入ることである．このコンパイラを用いて，新しい計算機にコンパイラをブートストラップする方法については，いくつかの文献が出ている[*3]．コンパイラは擬似スタック・マシンに対するコードを生成する．そのコードをエミュレートあるいはインタプリトするプログラムを書く必要がある．そしてこの擬似スタック・マシンにそっくりな機械を作るというアイデアはすでに何人もの人によって説えられている．

このスタック・マシンのアイデアは1961年に西ドイツの F. L. Bauer という学生によって導入されたものだ，と私は思っている．Bauer と K. Samelson は特殊なスタック・マシンの特許をそのときまでに取っていた．私はこのアイデアに興味をそそられてはいたが，1970年になるまでスタック・マシンで働く機会がなく，そのままになっていた．

1970年に私は，バロース 5500 に基づくあるオンライン・システムの製作に参加した．1974年までにわれわれは2台のバロース5700に 510 台の端末を接続して動かした．そしてそのシステムは毎日毎日休むことなく24時間稼動し，1秒から3秒間隔の検索要求と，6秒から12秒間隔程度のデータベース更新要求を処理していた．2台のバロース 5700 は，それぞれ4マイクロ秒のコアメモリを 192K バイトだけしか持っていないにもかかわらず，同時に6個から15個のプログラムを走らせることができた．このオンライン・システムは4人のプログラマによって作成された．そのときから私は，計算機はスタックに基づくべきであるという私の信念を一層強くした．

この本の第1の文章[*1]は，スタック・アーキテクチャの基本的なアイデアを紹介し，スタックに基づく，あるいは興味深いスタック機構を持つ多くのマシンの概要を述べている．これらの文章が読者に有用であり福音をもたらすことを期待している．

スタック・アーキテクチャの研究を継続したいと思う読者は，Chu の本[*2]の Doran の章か，あるいは Stone の本[*3]の McKeeman の章から始めるとよい．さらにより進んだ扱いに対しては，Organick の本[*4]がおそらくベストであろう．また Pratt の本[*5]は，数種のプログラミング言語で必要となる実行時の環境の問題をいくつか扱っている．（参考文献の完全なリストは後［6，7月号］にまとめられる．）

HP 3000 についての大規模な性能評価プロジェクトの結果に関する Blake の記事[*6]は，たしかにコンピュータの設計者にとってしばらくのあいだは有益なものになるだろう．Blake は私たちにスタック・マシンの最適設計のための多くの情報を与えてくれる．さらに高級言語の実行に実際に用いられる機種について，スタック・アーキテクチャだけでなくいくつかの伝統的設計による計算機についても価値ある洞察を与えてくれている．

John Couch と Terry Hamm による文献[*6]は，この本の中の他の記事よりも進んだものである．この記事は，どんな計算機に対してでもコンパイラを書こうとする人にとって，価値がある．加えてこの二人は，スタック・マシンのより難しい詳細と問題点をあばくために，注意深い研究を行なっている．ブール式の評価についての章では，値が完全にスタック上で計算され，それに基づいた条件分岐が実行されるようならば，記憶域の最適化も可能となることが書かれており，それにも注意すべきである．

Duncan は，アメリカ以外のスタック・マシンの歴史について，より啓蒙的な見方を与えてくれている[*6]．また1960年頃にイギリスで行なわれたすばらしい仕事についても述べている．さらに印象深いのは，オーストラリアの C. L. Hamblin の発明である．Hamblin はおそらく常に主要な計算機設計者（アーキテクト）の一人としてランクされるだろう．Duncan によって引用される Hamblin の意見，すなわち『充分に柔軟な制御命令があるなら，そのシステムでは制御の転送は不要になる』ということは，1957年に出版された論文で公表されていた．このことは，Hamblin が，プログラミングにおいて，GOTO 論争が起こるはるか以前に，機械語命令から GOTO をなくしたいと思っていたことを示している．

Duncan の論文に書かれているサーベイを完全にするために，われわれは同じ頃にアメリカで行なわれている研究の概要を知る必要がある．われわれが知る限りでも Hamblin と同じようなすぐれた計算機設計者がアメリカにもいたのである．

スタック計算機を述べるのに，最近までアメリカではただ一つの会社，つまりバロース社だけを論ずるだけだったが，ごく最近 HP 3000 シリーズによってヒューレット・パッカード社がこの舞台に登場してきた．またマイクロデータ社が 32/S を 1973 年頃に発表しているが，まだ市場性はあまりない．

バロースは，B5000，B5500，B5700にはじまり，B6700，B7700へと継続してスタック計算機を生産してきた長い歴史を持っている．バロースにおけるスタック計算機のアイデアは，おそらく Robert S. Barton によって与えられたものだろう．その中でも B5000の設計はすべて Barton によって考えられたものと思われる．そのマシンの設計には他の者も明らかに参加していたが，その構造のすべては本質的に Barton だけに依るものである．数冊の本を除いて，すべて Barton によってなされたことは，バロースの開発者のあいだでも完全に同意さ

*1（訳注） インテル社の 8086 およびザイログ社の Z8000 が噂の機種といえよう．1977 年後半の BYTE 誌などに「Pascal マシンが出るか？」というような議論が載せられているが，それらの震源はこの記事ではないかと訳者は推測している．8086 については，同様にスタック演算能力をもつ日電の μCOM 1600 などとともに **bit** Vol. 10, No. 7 に「16ビットマイクロプロセッサ新時代」の題で紹介されている．

*2（訳注） Pascal の言語仕様については，**bit** Vol. 10, No. 1～10 に和田英一先生のわかりやすい記事がある．

*3（訳注） 日本での文献としては疋田輝雄氏による「コンパイラのキットを用いた Pascal の移植」日経エレクトロニクス，1976年，No. 149 が比較的にわかりやすい（他の文献をそこから孫引きするとよい）．

*1（訳注） 本文中で「この本」というのはすべて IEEE の COMPUTER 誌5月号のことを指している．「第1の文章」とは，この翻訳の第2回（6月号），第3回目（7月号）の部分に相当する．

*2（訳注） Y. Chu: "High-level language Computer Architecture", Academic Press である．

*3（訳注） H. S. Stone: "Introduction to Computer Architecture" らしいが，訳者はまだ見ていない．

*4（訳注） E. I. Organick: "Computer System Organization", Academic Press. 日本語版は土居範久訳「計算機システムの構造」共立出版，1978 が出ている．

*5（訳注） T. W. Pratt: "Programming Languages: Design and Implementation", Prentice Hall である．

*6（訳注） この翻訳には収められていない．同号の中の他の記事である．

れるものと思われる．

Barton は，B5000 をただのスタック・マシンとしてではなく，Algol マシンとして見ていた．だから機械の構造とその機械語は，できるだけ Algol に近くなるように意識して作った．そして，B5000の上で他の高級言語の実行が効果的にできるようないくつかの機構をつけ加えた．

Barton は数年にわたってバロースを出たり入ったりして，あるときは従業員として，あるときは外部のコンサルタントとして働いている．現在，彼はシステム・リサーチ部門のエンジニアリング・ディレクタである．

Barton は，オーストラリアの Hamblin とは独立で，またほとんど同じ頃に，スタック・マシン・アーキテクチャを開発し，画期的な結論に達した．その結論というのは，『ポーランド記法はディジタル計算機を制御する自然な方法である』ということである．1958年に Barton は，論理についての Kopi の本を読んでいるときに，ポーリッシュ・マシン*1 のアイデアをほとんど即座に考えだした．そのとき，彼は，ポーランド記法への２段階の変換をし，コンパイルされたコードを実行させようと思った．そして，サブルーチンの制御とパラメータに対してだけでなく，ループの制御にもスタックを用いようとした．そのときにはスタックによるループの制御は実行に移されなかったが，遅かれ早かれ使われる必然性をもっていた．

B5000を作ろうというアイデアは，1959年に Barton によって提案された．次いでバロース社は1959年春にその製作プロジェクトを開始した．その機械とオペレーティング・システムとコンパイラは1962年に完成した．そしてB5000は1963年初冬に初出荷された．

バロースの Balgol プロジェクトは大変重要な影響を及ぼした．この Balgol とはB220上に作成された Algol 58言語である．B220はスタック・マシンではなかったが，コンパイラは中間言語にポーランド記法を用いていた．コルーチンがコンパイラ中に大規模に取り入れられており，各コルーチンはコンパイル中の記憶域の必要量によって切り換えることができる．これは中間段階にアセンブリ言語を持たず，大変，高速なコンパイラであった．すなわち１機械語命令を生成するのに，60ステップほどが走行するだけであった．これは現在でもおそらくあまりない速さである．

Joel Erdwinn は Balgol プロジェクトのリーダーであり，その基本設計者かつ製作者であった．また，Jack Merner はこのプロジェクトの中で大変重要な人物であり，David Dahm は初期に重要な役割を果たしていた．

Balgol コンパイラが果たした重要な役割は次のような点である．バロース5000にはアセンブリ言語をいっさいなくすべきだと Barton は考えていたが，このことをバロースの経営者に理解させるのは難しいことであった．しかし前に述べたように，ポーランド記法をその中間言語にもつ Balgol がB220のカードリーダの読取り速度でコンパイルできるという事実は，Barton 自身と彼の高級言語マシンのアイデアに対するある種の確実性を与えていた．

経営者に対して，技術的な原則を守ることは，しばしば技術者と設計者が直面する課題となる．バロース経営陣の折衷案から Barton のアイデアを守った人は，Lloyd Turner であった．Turner はB5000の実現に際して，プロジェクトを完了できるようにプッシュするという役割を果たした人である．Turner は Algol 60 でB5000のコンパイラとオペレーティング・システム（バロースではマスタ・コントロール・プログラム（MCP）と呼ばれている）を作成しようという試みを保護する立場になったが，B5000の構造は典型的なものとは異なっていたので，より多くの保護を行なう必要があった．すべてのシステム・ソフトウェアは Algol で作られた．それだけでなく，アセンブラは存在せず，誰の手にも使用不可能であった．これが，受け入れるのに困難なアイデアであるということは，普通の商業誌で，1975年（Multics の後！*2）に“オペレーティング・システムは完全に高級言語で書けるか”という議論がされていたという事実でも証拠立てることができる．

新しい計算機に対するオペレーティング・システムやコンパイラが，再びアセンブリ言語で作成されることはありそうにないので，高級言語によって最初になされたものの結果を見ることは非常に有益である．B5000に対する最初の Algol 60コンパイラは Algol 60で書かれていた．そしてそれは手で機械語に翻訳された．それが満足に動くようになると，すぐ，それは自分自身の Algol プログラムをコンパイルするために用いられた．この中で，最近のコンパイラに用いられているブートストラップ手法に慣れた人びとを仰天させるような違いは，ただ１つだけである．それは，コンパイルされたものは，手で翻訳されたものより小さくそして高速であった！ ということである．この事実は，B5000のスタック・アーキテクチャによってもたらされたものである．

*1（訳注） ポーランド記法に基づくマシン．

*2（訳注） Multics は PL/I で書かれている．このため“！”がつけられているのであろう．

B5000の全体の最大コアサイズは192Kバイトであり，今日の典型的な，スタックによらない大型計算機上のオペレーティング・システムの常駐部*1よりも小さい．このことはB5000のスタック・アーキテクチャがもたらすものであり，MCP の特徴によるものではない．

これを示すためにB5000 MCP の能力について考察する必要がある．カードリーダ，ラインプリンタなどはすべて MCP の機能の一部として自動的にディスクへスプールされる．仮想メモリは，いわゆる存在ビット割込みによるディスクからの自動セグメント取込みによって実現されている．B5000は最初からマルチプロセッサとマルチプログラミングの両者をサポートするように計画してあった．最初のマシンでさえ，２つの強連結プロセッサ（MP）と，同時に混在する15以上の独立したジョブをサポートする（MVT*2）ように計画してあった．B5000は通常 192K バイトの主記憶で出荷されたが，いくつかのものはそれより小さな大きさで出荷された．オペレーティング・システムと Algol コンパイラは，48Kバイトに満たない領域で動作する．B5000 の Algol は階層的なデータ構造と高水準の割込み処理を除いて，本質的には PL/I の機能をすべて持っていることを考えれば，スタック・アーキテクチャの価値はますます明らかになってくる．

おそらく Barton がB5000の設計で心にとめていた主な原則は，**コードの圧縮**であっただろう．このことについては，この本の他の記事に載っている．このデザインに次いで重要な基本原則は，サブルーチンの基本的な性質に関するものであった．手続きの概念に基づくサブルーチンは，帰り番地の確保よりもささいなものとして組み込まれた．Jack Merner の手続き構造の分野における貢献はB5000の設計に重要なものであった．また，命令コードの長さは一定でなければならないという基本原則も考えられていた．このためB5000には相対分岐だけが含められた．すべてのコード・セグメントやデータをアクセスするために，ハードウェア・ディスクリプタを使用するという考え方はスタック・アーキテクチャにとって本質的ではないが，多くのスタック・マシンで用いられている．ハードウェア・ディスクリプタの商用機における使用はB5000が最初であった．Barton はアトラスとページングについての文献を読んだのち，ディスクリプタの使用を決心した．

ヒューレット・パッカード社の HP 3000 は1971～72年に設計された．その設計者のうちの数人は，以前にバロースで働いており，B5500とB5700の影響をはっきりとみることができる．HP 3000は，おそらく将来のスタック・マシンが用いるであろう構造をはっきりと表わした特徴を持っている．この本の Blake の記事で議論されているように，HP 3000 はほとんど最適な効率を持つように，命令セットの詳細な設計が行なわれている．

マイクロデータ社の 32/S は Bill Roberts と Dennis Allison によって1972年に設計された．32/S はたしかにスタック・マシンであるが，それ以上に高級言語マシンというほうがふさわしい．Allison は Algol 60 に類似した高級言語 MPL*3 を最初に設計し，これを商用的には PL/I の派生語といっていた．その後 MPL を動かすために，マイクロデータ3200上にマイクロプログラム制御式計算機を設計した．それは，B1700と同様に，現在生産されている真の高級言語マシンの良い例であろう．Allison は明らかにその改良点をいくつか考えているが，32/S はたしかにスタック・アーキテクチャの歴史の一隅をなすものといえる．

注意してほしいのは，スタック・マシンでは，本質的には，汎用レジスタは存在しないことである．この汎用レジスタの代役はスタック・ハードウェアによって自動的に行なわれている．プログラマや技術者たちが，計算機について考える上でのこの本の影響を一言でいうならば，私は次のように言いたい．「次にマイクロプロセッサについてのサーベイをそれらの機能一覧をつけて出版するなら，読者はレジスタ数や命令セット中の命令数という欄をみて，『結構いい数だけれど，いったいこれにどんなおもしろさがあるんだろう？』と言うだろう」*4．

〔全３回〕

*1（訳注）たとえば，IBM の OS/VS 2 は250～350K バイト．

*2（訳注）Multiple Variable Task の略．

*3（訳注）Micodata Programming Language の略．

*4（訳注）後に次第に明らかになるが，原著者は「マイクロプロセッサの動向は，ここ数年の不活発な計算機構造についての議論を活発化する役割を担うことができる．それらは多少ともスタック機能をもっていて，これは正しい方向である」と述べており，汎用レジスタ志向でない見方を期待しているようである．

また，汎用レジスタの役割の代行について補足するならば，ここでは CPU 中にトップオブスタック・レジスタ（TOS レジスタ）を置く話について述べているものと思われる．TOS レジスタはプログラマには見えない．そして主記憶上のスタックの延長として働くようになっている．TOS レジスタとそれを管理するハードウェアの存在は，「スタック・マシンの場合，その処理速度はスタックがおかれているメモリの速さでおさえられてしまうのではないか？」という一義的な疑問に対する解答ともいえる．

訳者雑感

マイクロプロセッサの記述力とスタック

井田昌之

●マイクロプロセッサはスタック・マシンではない

しかし，その方向に向かっているのではないかという勝手な推測をしている．この筆者の推測が誤っているかどうかは個人的な問題であるが，ここに紹介した「スタック計算機入門」(全3回)はスタック・マシンと呼ばれる計算機の構造や実例について比較的よくまとめられている．だから，これからの計算機の構造などについて思いをめぐらす材料の一つになると思う．

原著者は「マイクロプロセッサの製作者たちは，おそらくここ数年の計算機構造についての不活発さを正すチャンスを持っている」という趣旨を述べている．その点最近の16ビットチップはその方向に沿っていると，とれないことはない．

原著者の主張の中で興味深いもう一つの点は，「システム記述用言語として高級言語を使おうというのは，次第に経済性の面からも現実性がでてきた．そしてその高級言語として，ブロック構造を持ち，再帰的手続きを許す型の言語が使われるのなら，必然的にスタックが必要となる」という点である．

事実スタックは，入れ子になった環境の保存に便利であり，割込み処理やリエントラント（再入可能）手続きなどに対してもきわめて自然に対処できる．純スタックでは一番上のものしか見えないので，局所環境を高速にアクセスできるようにスタック内部をさすポインタレジスタをおく．そしてそれをベースレジスタとして用いるというのも自然な拡張である．さらに静的な入れ子環境（宣言の入れ子）と動的な入れ子環境（実行順序の入れ子）を分離して考えられるような手段が提供されれば，原著者のいうスタック・マシンが誕生する．

こうした構造をハードウェア的に備えたマイクロプロセッサは，いままでなかった．しかしたとえば 8086 には筆者の知る範囲でも，若干のエイドがついている(「16ビットマイクロプロセッサ新時代」**bit** Vol. 10, No. 7 参照)．また，もちろん多少のオーバーヘッドとなるが，ソフトウェア的にこうしたスタックを形成することは他のマイクロプロセッサでも可能である．

さらにスタックに関連したことをいくつかつけ加えれば，μCOM 1600 では『スタックの上の2つを加えて(かつそれらを除去し)その値を新しくスタックの上におく』というスタック加算命令のオペランドの組合せで記述できる．これは PDP-11 などのスタック能力があるとされるミニコンと同一の機構である．また「スタックに局所変数をおき，いわゆる変数域を独立に割り当てることはしない」というスタック・マシンでの使用感覚の一部は，8080 においても味わうことができる．

事実，筆者らの製作した Lisp 専用機 ALPS/I のシステム・プログラムの記述で使用した RAM 領域は，グローバルなスイッチ群とバルクメモリとの入出力バッファだけである．(関数の引数は入出力バッファ上のままで渡し，値はレジスタ返しをする，なども行なわれている．)それら RAM 上の作業域の定義はアセンブラで70行程度しかない．

また，こうしたスタック主体の手続き構成では，**再帰性と再入性**を与えることがかなり容易であるという事実が意外に知られていない．これは残念なことだと思う．

たとえば，FORTRAN で「再帰的に」プログラムを書くのは大変なことである．後藤英一先生の書かれたすばらしい解説(「連載 LISP 入門」**bit**, 1974)も，その前半に記されている「再帰的な流れをもった FORTRAN コーディング」の説明がなかったら，もっと中身がわかりやすかった，という声が筆者の周囲で聞かれたこともある．もっとも，そこが興味深い点ではあったが．

逆に，FORTRAN 流の考えでマイコンのプログラムを作ったらどうなるか？ prog 変数のやたらと多い prog 多用の Lisp プログラムという感じになる．特に小さなサブルーチンやテストプログラムを作ったときにはっきりそれがわかる．せっかくスタックがあるのに，作業用の変数域をとったり，再帰的な概念をループ化して考えて組んだりした覚えが筆者にもある．

まったく，単純スタックでもそれを使いこなすには相当の時間がかかるものである．また80系のマイコンのプログラムで XTHL や XCHG をうまく作ったものが少ないのも残念なことである．

そんなわけで，次に単純なスタック構造の上に再帰的手続きをどうやって組み立てるかについて述べてみたい．再帰的呼出しとその引数がスタックの上に順に積まれ，かつ参照されていく点は，スタック・マシンの原点ではないかと，勝手ににやにやしている．

●再帰的定義とマイクロプロセッサのスタック

——アッカーマン関数は 8080 なら書く気がするが，IBM のアセンブラでは書く気がしない

アッカーマン関数を例にとりあげてみよう．

アッカーマン関数の定義をマッカーシーの条件式を使って書くと次のようになる．

$$
\begin{aligned}
&ackermann\,[m;\,n]\\
&=[m=0\rightarrow n+1;\\
&\quad n=0\rightarrow ackermann\,[m-1;\,1];\\
&\quad T\rightarrow ackermann\,[m-1;\,ackermann\,[m;\,n-1]\,]\,]
\end{aligned}
$$

$$
\left(\begin{array}{l}
ackermann = \textbf{if } m=0 \textbf{ then } n+1\\
\qquad \textbf{else if } n=0 \textbf{ then } ackermann\,[m-1;\,1]\\
\qquad \textbf{else } ackermann\,[m-1;\\
\qquad\qquad ackermann\,[m;\,n-1]]
\end{array}\right)
$$

筆者の用いているコーディング手法* に基づいてこれを 8080 のアセンブラに落とすと次のようになる．

```
ACKERMAN  POP H;  帰り番地のポップ
          POP D;  nのポップ
          XTHL ;  mのポップと帰り番地の再格納
          HIFZ INCRN;  m=0 なら INCRN へ
          DIFZ DECRM;  n=0 なら DECRM へ
          DCX H;  m=1
          PUSH H;  m=1 を確保
          INX H;  mの値を復旧                「ット
          PUSH H;  次の再帰的呼出しのためのmのセ
          DCX D;  n=1 の計算                 「ット
          PUSH D;  次の再帰的呼出しのためのnのセ
          CALL ACKERMAN
          PUSH D;  値は D-E レジスタにあるので，
                   それを次の再帰的呼出しのために
                   nとしてセット
          CALL ACKERMAN;  mはすでにセットされ
                          ている
          RET   ;  CALL 文とあわせて JMP ACKE-
                   RMAN でもよい
INCRN     INX D;  n+1                        「帰る
          RET   ;  D-E レジスタに値 n+1 をもって
DECRM     DCX H;  m-1                        「ット
          PUSH H;  次の再帰的呼出しのためのmのセ
          LXI  D,1;  1を D-E レジスタにロード
          PUSH D;  次の再帰的呼出しのためのnのセ
                   ット
          CALL ACKERMAN
          RET;  CALL 文とあわせて JMP ACKERM-
                AN でもよい
```

ここで HIFZ および DIFZ は各レジスタペアがともにゼロならばオペランドの番地へ飛ぶマクロである．HIFZ は，たとえば次のような定義である．

```
HIFZ    MACRO  HZ
        MOV    A,H
        ORA    L
        JZ     HZ
        ENDM
```

このコーディングでは2つの引数 m, n を先に push して *ackermann* へとびこむ，つまり引数の上に帰り番

* 「Lisp マシン製作奮戦記(2)」**bit**, Vol. 10, No. 15, 1978.

地が積まれた状態で処理がはじまる．たとえば，次のような呼出し手順によって起動される．

```
        LXI   H,2 ; m=2
        PUSH  H
        LXI   H,1 ; n=1
        PUSH  H
        CALL  ACKERMAN ; ackermann [2 ; 1]
```

また関数値は D-E レジスタに入れられて呼び出した手続きに帰るものとしている．

筆者の経験ではこうした作り方が速度・見やすさの点でも一番良いようである．なお，ACKERMAN と 8 文字で書いているのは利用しているアセンブラの制約の通りに記しているためである．

この手続きの問題点があるとするなら *ackermann* の場合はよいが，他の場合（たとえば引数が 4 つ）には引数の取り出しに際して pop，pop，xthl では破綻をきたすのではないかということである．確かにそうである．しかし，引数が 4 つ以上の関数というのはあまりないのである．Lisp の関数でも筆者の組み込んだ約 100 のものはすべて 3 個以下の引数しかない．

call 命令と push，pop 命令などの利用によって再帰的呼出しが自然に行なわれる点に注意してほしい．また局所変数が 1 つも使われていない点にも注意してほしい．

筆者が慣れているもう一つの言語に IBM のアセンブラがあるが，アッカーマンをそれで書く気はまったくしない．

●アッカーマン関数はより高速になる

——マイコン向けアッカーマン関数の段階的最適化

次のような考え方で高速化できる．

①不要な再帰的呼出しをやめる．

ackermann [*m* ; *n*−1] 以外はループにできる．

②ジャンプ命令を減らす．

③レジスタの内容の他用

その記述を示そう．

```
ACKERMAN  POP   H
          POP   D
          XTHL
          HIFZ  INCRN
LOOP      DIFZ  DECRM
          DCX   H
          PUSH  H
          INX   H
          PUSH  H
          DCX   D
          PUSH  D
          CALL  ACKERMAN
          POP   H
          HIFNZ LOOP ;  H-L レジスタがゼロでないなら LOOP へ．
                        次の再帰的呼出しの m=0→n+1 の部分に相当
INCRN     INX   D
          RET
DECRM     DCX   H ;     m−1
          INX   D ;     ここへ来る場合，必ず D-E ゼロである．したがって IN
                        により定数 1 を作っている
          HIFNZ LOOP ;  次の再帰的呼出しに相当
                        る．m および n は H-L
                        D-E にそれぞれセット
                        れている
          INX   D ;     m=0→n+1 で帰る
          RET
```

このコーディングにより，アッカーマン関数の最高版（？）が定義されている．

ackermann [1; 1] の実行に 2 M 版の 8080 とすれば 152.5 μ 秒を要することになる．COMPUTER 誌の中の Blake の記事によれば，HP 3000 のアセンブラで 24 μ 秒，コンパイラ版の BASIC で 71μ 秒である．

フレーム・スタッキングや命令の先読みが行なわれるという 8086 なら，どの程度の速度とコードの圧縮が可能になるか楽しみである．

●もう 1 つの話

——スタック・アーキテクチャはかなり浸透している

スタック・マシンという言葉は出てこなくても，その構造は身のまわりにかなり浸透している．たとえば，

① Algol 型言語の処理系およびコンパイル・オブジェクトの環境．

Algol はもちろんのこと，PL/I や Pascal を語るにもスタックは不可欠であり，スタック・マシンを仮想しその命令体系を中間言語とする作成方法もいくつかの処理系においてみられている．またそうした型でコンパイラの説明をしている書物もみられる．スタックとヒープがその主な構成要素である．

② さらに，一般の高級言語の概念構造と中間言語に広く用いられている．

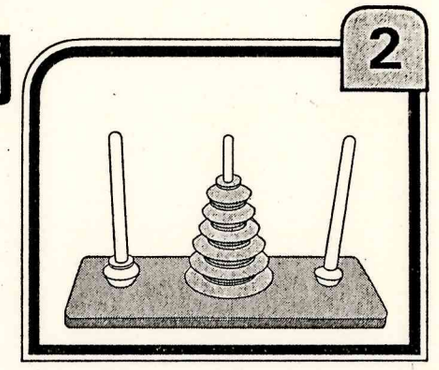
たとえば，コンパイラを勉強するには目を通す必要がある Lecture Notes 21 の「Compiler Construction」にはスタック・アーキテクチャを想定して成功した実例として，BCPL や Pascal などとならんで，筆者のいる大学で最も普及した言語であるガバー社の FORTRAN G コンパイラがならんでいるのである！

FORTRAN G コンパイラは pop と呼ばれるスタック・マシンを想定して作られている．pop の命令セットは約 100 あり，2 つのスタックをもつとされる．

以上，ALPS/II のデザインを書いたり消したりとか，小さなコンパイラの作成など，現在筆者が手がけていることの中でもやもやとしていることを書いたが，これをもって「スタック計算機入門」の訳者雑感とすることにする．なぜこの翻訳をする気になったかについて感じていただければ幸いである．

（いだ まさゆき　青山学院大学）

スタック計算機入門



デビット・M・バルマン

訳 井田昌之

1. 序

●基本的なアイディア

スタック，すなわち「後入れ先出し(last-in first-out)」記憶が，最近なじみ深くなってきた．

その理由は，多くのマイクロプロセッサがいろいろの形でスタックをそなえているからである．あるマイクロプロセッサでは，スタックにサブルーチンの帰り番地を入れられるようになっているし，また，一時的なデータを入れられるようなマイクロプロセッサもある．

スタックという言葉は，「一番上」に情報をおくという意味である．すでにスタックにあるものを「押し下げる（push down)」というアイディアからきている．スタックの中の情報を取り出すには，一番上のものを「はじき出す（pop up)」という操作で行なう．

この「純粋なプッシュダウン」と，より一般化されたアイディアによるスタックとは，区別して考えるほうがよいだろう．純粋なプッシュダウンでは，push と pop という 2 つの操作だけが許されている．つまり push は，スタックの一番上に新しいデータを置く．そしてそのデータは，さらに新しいデータによって push されるまで，あるいは pop 操作によってスタックの一番上から取り除かれるまで，その場所に置かれる．これら 2 つの操作は次のように表わすことができる．

```
PURE_PUSH  X    STACK_TOP:=M[X]
PURE_POP   X    M[X]:=STACK_TOP
```

この右側の表現は，左側のスタック操作の意味を表わしている．この定義では，スタック・トップの場所とスタック中に含まれる要素の数を常に持ち，具体的な push，pop 操作を行なう内部機構が存在することを前提としている*．この内部の操作は，使用者には見えないし，ここにも示してない．M[X] というのは全メモリをあるベクトル M として扱うことを意味している．すなわち，M[0] はメモリの最初の番地（アドレス）を表わし，M[X] は X で決まるメモリのある番地を表わす．

さて，データを使うときには，「後入れ先出し」の基本原則に従う．だから，一番上以外のデータ，つまりスタックの中のデータを参照する手段はない．どうしても中のデータを取り出したいときの唯一の手段は，pop 操作を連続していって，求めるデータの上にあるデータをすべて取り除くしかない．

「純粋なプッシュダウン」は強力なアイディアであるが，実際的で効率よく使おうという目的には制限がありすぎる．ここで導入しようとする一般化とは，スタックの中のデータを参照する能力についてである．通常，スタックはその計算機の主記憶中にある．また CPU 中にスタック・トップの番地を指すスタックポインタ（SP）がある．したがって，スタック内のデータを参照するには，スタックポインタを利用することになる．

この場合のスタックと主記憶に対する 2 つの基本操作は次のようになる．

```
PUSH  X    SP:=SP+1
           M[SP]:=M[X]
POP   X    M[X]:=M[SP]
           SP:=SP−1
```

直観的に，たとえば，M[SP−8] とか M[SP−N] といった番地を参照することによって，スタックの内部を

* （訳注）スタックのために使われる領域の管理と，その中の場所を位置ぎめする機構のことである．

のぞくことができる．この表記方法では，スタックは「上」へ，つまりスタックポインタの値を増加させる方向へ延びていくことに注意してほしい．多くのマイクロプロセッサでは，設計者たちは，スタックを「下」へ延びるように扱っている．しかし両者に特に大きな違いはない．後節で，スタック内のある番地を覚えるというアイディアを解説する．このことにより，それを利用して，スタック内のデータを呼び出せるようになる．

なお，この解説の用語や表記方法は，バロース社[1]，ヒューレット・パッカード社[2]のマニュアルに依っている*1．

●1命令当りの番地の数

ここでは，ほとんどの命令がスタック・トップにあるオペランド（被演算数）に対して働くよう定義されている場合に限り，その計算機を「スタック・マシン」と考えることにする．

たとえば乗算命令は，トップの数とその次の数を掛け合わせ，この2つのオペランドを除去したあとのスタックの上にその結果をおく．一般的にいって，2つの命令だけが番地をオペランドに持つことができる．すなわち push と pop である．push 命令では，オペランドを取り出すべき番地を指定し，pop 命令では pop されたオペランドを格納すべき番地を指定する．後節で，番地を指定すると便利な他の二，三の命令について解説する．

初期の計算機の中には，1命令当り3つのオペランドアドレス*2を有するものがあった．またある計算機は，1命令当り2つのオペランドアドレスを持っていた．しかし今日の多くの計算機では，1命令当り1つのオペランドアドレスというのが普通である[3]．そして「汎用」レジスタを，1つまたは複数個のオペランドとするものが多い．レジスタの番地というのは，対象となるレジスタの番号のことである．

番地部を小さくしようとする傾向の一つの理由は，平均命令長を減少させるためである．命令のそれぞれに番地を持たない（ゼロアドレス）スタック・マシンは，こうしたコードの圧縮傾向中の究極的な段階で現われることになる．後でこの問題について，さらに細かく調べることにする．

2. サブルーチン制御と連係

●サブルーチンの重要性

ソフトウェアの最も重要な概念の一つに，サブルーチンがある．モジュラ・プログラミングや構造化プログラミングの基本的なアイディアは，大きなプログラムをサブルーチンと呼ばれる多くの小さな，理解しやすいモジュールに分割することにある．しかし，こうしたプログラミング・スタイルは計算時間と使用時間を無駄に使用する，という議論がしばしばきかれる．この議論の根底には，

「サブルーチンの呼出しと復帰の機構は，多くの計算機の構造（アーキテクチャ）において，あとからの思いつきでつけ加えられたと思われる」

という認識があると思う．

「サブルーチンの呼出しおよび復帰の最適な機構としては，スタックを用いるべきだ」

というのは驚くべきことではない．良形のプログラムでは，サブルーチンは「後入れ先出し」で完全に入れ子になるだろう．

●帰り番地

現在の計算機はすべて，帰り番地をどこかに確保し，制御をそのサブルーチンに渡すような「サブルーチン呼出し命令」といったものを基本的に備えている．「どこか」というのは，通常，固定されたレジスタか，サブルーチン呼出し命令中に指定された場所である．しかしながら，この場合各サブルーチンの再度の呼出しを可能にするには，プログラマが，帰り番地を別に確保していくようなソフトウェアを書いておかねばならない*3．

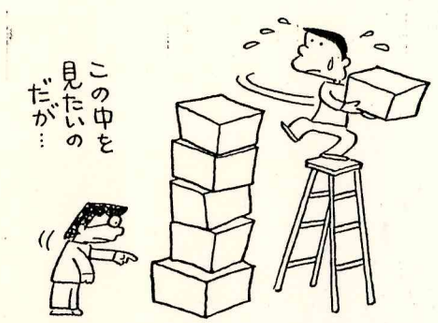
さらに悪いことに，多くの計算機の場合，ソフトウェアでスタックをつくるには，複数の命令が必要であり，また，そのサブルーチンに固有な（ローカルな）場所に帰り番地を格納するプログラムが作られがちである．その結果，あるプログラム中の各サブルーチンはその帰り番地のための場所を浪費することになる．入っているがまだ出ていないサブルーチンのそれぞれの番地をまとまって記憶するようなことはしない場合が多い．さらに再入的（リエントラント）サブルーチンに対する準備もない．

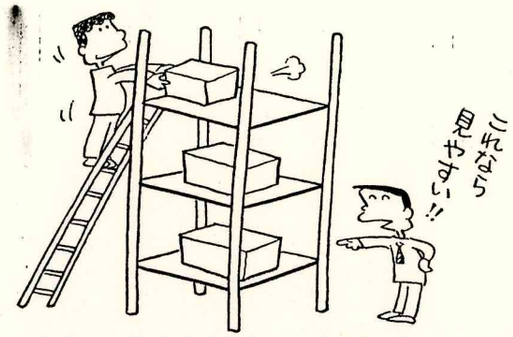
つまり，あるサブルーチンが現在の実行が完了する前に他のプロセスによって呼び出されるならば，呼出し命令によって格納された場所から帰り番地を取り出し，その次の呼出しが完了するまでそれを保持する手段がなければならない．さらに，このサブルーチンへの他の呼出しが処理されるときには，先のものとは別な確保用の場所が必要となる．またこれらは，サブルーチンからの復帰では逆の順序でスタックから出されねばならない．このことは，もしマルチプログラミングと割込み処理があるなら，サブルーチンの帰り番地は，後入れ先出しスタックに格納されねばならないことを意味している．

サブルーチンの出入りが重要なことは，マイクロプロセッサの多くの設計者はわかっていた．このため，マイクロプロセッサの特徴をいくつか真似することによって，より大きな計算機が改良できるという特筆すべき事態がいまや到来している*．

●引数の扱い

サブルーチンの出入りを合理的に行なわせることにより，それ以上の利益が得られる．多少なりとも大きなサブルーチン構造では，呼出し側と呼び出された側のルーチンのあいだで値や番地の受け渡しが必要となる．そうした変数を参照する最もよい方法は，それらをそのサブルーチンを呼んでいる直前のスタックに push する方法である．そうすればそれらは，そのサブルーチンの実行のあいだだけ記憶域を占めることになる．しかし，スタックの手法以外では，すべて各サブルーチンに対して恒久的に記憶域を割りつけるか，あるいは同時に呼び出されるおそれのないサブルーチンを正確に注意深く決定し，それらのルーチンのあいだで記憶場所を共有することが必要になる．だから，前者は記憶域を浪費し，後者は失敗の原因になる．





●局所変数

引数と違って局所変数は，呼出し側と呼ばれた側のルーチンのあいだで渡されないということを除いて，引数に対するものとまったく同じ議論が局所変数に対しても可能である．それらの最も良い置き場所はスタックの上である．事実，引数として渡されない局所変数のほうが多くなる傾向があるなら，その理由づけはさらにはっきりとする．スタック上にこれらすべてをおくことの利点が理解できれば，スタックによらない場合に生じる記憶域の非効率な使用法や，ソフトウェアによる明示的な場所の共有による危険を回避することができる．

●再入（reentrancy）と再帰（recursion）

帰り番地，引数，局所変数についていままで述べたことは，スタック手法のもう一つの長所については触れていなかった．もしスタックがこれらすべてのものに利用されるなら，再入プログラムおよび再帰プログラムをつくることが容易になる．

事実，スタックを用いた場合，再入性のないプログラムを書くことは難しい．つまり，そのルーチンを実行させれば自動的に帰り番地，引数，局所データのコピーを別に持つことになり，それらはそのルーチンの他の実行によって変更を受けることはない．この問題は再入的あるいは再帰的ルーチンのいずれかに対して解決しなければならない．再帰的なルーチンとは，直接的にあるいは最終的にそのルーチンを呼ぶ他のルーチンを呼ぶようなものである．そうした再帰呼出しの場合，以前の実行が完了していない状態で，実行を再開する．もしそれぞれの実行に対して独立したコピーを保持していないのならば，そのルーチンの一時的な値はすべて再帰呼出し中でこわれてしまうだろう．

---

*1 （訳注） 日本語訳に際しては，土居範久訳「計算機システムの構造 —— バロース大型計算機シリーズ」（共立出版．1978）を参考にし，バロース関連の訳語については，それに準じた．

*2 （訳注） オペランドアドレス：命令の対象となるオペランドに指定された番地．

*3 （訳注） 自分自身を呼び出すような再帰的な手続きや，他のプロセスによって共有される再入的な手続きの場合などである．これらの場合，帰り番地をスタックする必要がある．このことを汎用レジスタ・マシンでは，ソフト的にやらねばならないので，再帰性・再入性をプログラムに与えることはそのコンパイラを重くし，かつ実行時の効率を悪くするきらいがある．

* （訳注） 訳者の知るかぎりでも，たとえば，Datapoint 社の 6600 は，8080 の構造を包含するように作られている．

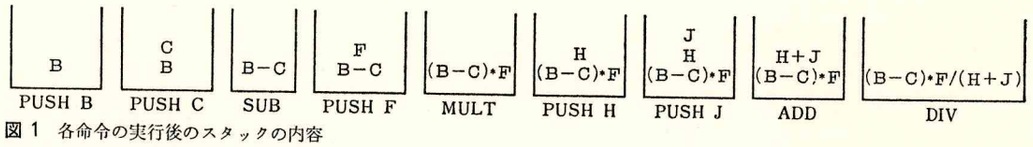


図 1　各命令の実行後のスタックの内容

再入的ルーチンとは，実行が完了する前に他の実行が再び進められるようなものである．これはたとえば，割込みの結果生じてくる．

いずれの場合にしても，中間結果や帰り番地はすべて確保されていなければならない．たとえどんなに多くそのルーチンが同時に使われていたとしても．

●プログラムとタスクの区別

プログラムとタスクというよく用いられる用語の正確な意味について，ほとんど合意をみないのは不幸なことである．

たとえば，「プログラム」，「ジョブ」，「タスク」，「プロセス」を考えてみよう．これらの言葉についての理解の違いが各メーカのマニュアルでも見られるのは好ましいことではない．異なった概念を統一的に考えるひとつの方法は，具体的なそれぞれのタスクを，時間的に変化しないアルゴリズム（決して修正されないプログラムコード）と，時間的に変化する実行記録（特定の実行に関連したデータ）との結合と見ることである．つまり，

タスク≡プログラムコード＋スタック

である．

要するにスタックは，一つのタスクを実行したときの自然な記録となっている．サブルーチンの帰り番地は，ハードウェア，ソフトウェアのいずれによったとしても，スタック中で明確に入れ子になる．引数とその入口と出口に沿ってまったく自然に入れ子になる．マイクロコンピュータでは ROM（読出し専用メモリ）の使用がふえている．この場合，ミニコンでのアセンブリ言語を使ったプログラミングのように，データをコードと混在させるのは望ましくないのは明白である．

「スタック以外にこれらのデータや引数を置くのに良い場所があるだろうか？」

プログラム中に共通して用いられる変数はすべて，そのプログラムの実行以前にスタック上に場所を割りつけるだけでよい．つまり大きくみれば，共通変数もそのプログラムには局所的なデータにすぎない．もしこのようになれば，常に記憶域にはただ1つのプログラムコードがあれば済む．独立したスタックを持つことによって一時にいくつもの「コピー」が走行できるのである．

3.　式の評価

算術式や論理式の評価には，スタック・マシンのほうが優れているということは以前から知られていた．スタック・マシンでは，呼出しが最小限になるようにレジスタや主記憶を動的にかつ自動的に割りつけることができる．算術式中の演算実行順序は，演算子の優先順位とカッコによって決定される．次の式を考えてみよう．

(B−C)＊F/(H+J)

スタック・マシンでこの式を評価するために，図1に示すコードが用いられる．

この図では，3つの新しい演算子が導入されている*．その演算子はスタック上の2つをオペランドとする算術演算子である．その定義は次のようになる．

```
SUB   M[SP−1]:=M[SP−1]−M[SP]
      SP:=SP−1
MULT  M[SP−1]:=M[SP−1]＊M[SP]
      SP:=SP−1
ADD   M[SP−1]:=M[SP−1]+M[SP]
      SP:=SP−1
```

これらの命令は，スタックの上にある2つに対して対応する操作をほどこし，その結果をそのときのトップの次に置く．そしてスタックポインタを1減らす．単項命令である補数演算（complement）は次のように定義される．

```
COMP    M[SP]:=COMP  M[SP]
```

このような単項命令は，スタックポインタを変えることなく，そのスタックのトップの要素に対してただ単に演算をほどこす．ここで上の式を次の形で書き直そう．

B C−F＊H J+ /

このカッコのない形式は**後置記法**，あるいはこの概念を考え出したポーランド人論理学者 Lukasiewicz にちなんで，**逆ポーランド記法**として知られている．ここで注意してほしいのは，逆ポーランド記法は，必要となるスタック・マシンでの命令の順序に正確に対応するということである．現在のすべてのコンパイラは，（明に暗に）最初に算術式や論理式をポーランド記法に変換し，次に機械語に変換する．スタック・マシンのコンパイラは，この第二の変換を行なう必要がない．

算術式のもう一つの表現方法は木(tree)である．図2はこの算術式を木で表現している．

各演算子は木のノードとして表わされ，それはそのすぐ下のノードの値に対して演算を行なう．葉（leaf）のノードは式の本来のオペランドを表わしている．この例の中の乗算は，用いられる2つの値が確定するまで実行できない．つまりB−Cを乗算の前に実行しなければならない．逆ポーランド記法は枝渡り[4]（正確には end-order traversal）によって木をたどることにより，その木から生成することもできる．枝渡りの径路は図中の破線で示されている．逆ポーランド記法を生成するには通過した葉のノードを順に書きくだせばよい．現在のコンパイラは，コンパイル中にプログラムを木表現ではっきり扱う方向へ向かっている．生成された機械語を最適化するのにはこれは特に有効である．

4.　汎用レジスタ・マシンとの比較

●高級言語

ハードウェア・コストは，この数年急速に低下している．そしてこの傾向はおとろえるきざしがないようだ．しかし逆に，ソフトウェア・コストのほうは指数的に増加しており，その上がり方はおそらく次第に急激になるだろう．

これら2つの事実は逃がれることはできず，次のような一つの結論に向かうだろう．

「極端に小さな問題を除いて，アセンブリ言語の使用はすべて消え去り，高級言語の使用に置き換えられるに違いない」

このことを認識していない管理者による大型ソフトウェア・プロジェクトは，数年のうちに，高級言語によるものと競合できなくなるだろう．しかし，スタック・マシン以外の上での高級言語は，一般に効率的でない．Wichmann[5]は，B 5000 と 20 以上の計算機に対して ALGOL コンパイラを比較し，次のように述べている．

『結論を述べるならば，この特殊な機械の生成するコードは，競合する他の一般的な構造の機械に比べて約半分の大きさであり，きわめてコンパクトなものである』

Wichmann は IBM 360 や 370，ユニバック 1108，CDC 6600 などを始めとする，今日よく用いられる多くの計算機とB5000を比較している．1969年に行なった筆者の研究では，IBM 370 は典型的なアルゴリズムを表現するのに，B5000の 50〜100% 以上のビット数が機械語の構成に必要である．スタック構造と IBM 360 との興味深い比較は，Niklaus Wirth の初期の著作[6]にもある．

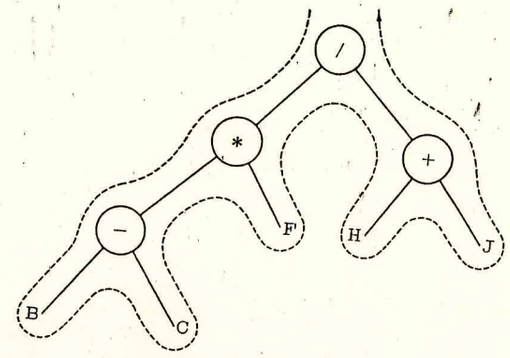


図 2　B C−F＊H J+/ の木表現

●コードの圧縮

レジスタ・マシンの場合，サブルーチンの引数や一般的な変数の役割をするレジスタが不足すると，現在の値を格納し，後にそれらを再参照するような命令を作る必要がある．そうしたデータに対する余分なメモリアクセスはスタック・マシンでも必要である．しかし，レジスタ・マシンはこれに加えてレジスタや記憶域を参照する余分なビット量をもった命令形式が必要であり，さらにそれらを取り出す（fetch する）時間がかかる．

レジスタ・マシンで必要となるデータは，明示的クラスと暗示的クラスの2つに分けられる．明示的クラスとは，高級言語の使用者によって明示的に記述される変数のことである．暗示的クラスとは，中間結果のためにコンパイラによってとられるものである．スタック・マシンは暗示的クラスのデータを必要としない．そして明示的アドレスに要するビット数もより少なくてすむ．暗示的アドレスが不要なことは，スタック・マシンの特徴であるコードの圧縮の主な理由の1つになっている．

コンパイラはいわゆる「匿名の」変数をいくつも用いる．つまり，そのプログラムの作成者が実際に参照できないが，指示した計算をコンパイルする過程で生じる変数がある．これらは一時的に記憶されそして一度だけ用いられる．また再び参照されることはない．スタックはそれらに自動的に領域を割りつけ，そしてそれらの使用

---

* （訳注）DIV という命令も加えられている．その定義は次のようになる．

```
DIV   M[SP−1]:=M[SP−1]/M[SP]
      SP:=SP−1
```

はスタックの pop によって行なわれる．このとき，同時にその記憶域の割付けが解除されたことになる．

記憶用コストが低くなったとしても，このコード圧縮は，なおも計算速度にかかわることに変りはないだろう．レジスタ・マシンは50～100% 多いビット数を格納し，それらを取り出す余分なメモリサイクルが必要となる．このようにしてレジスタ・マシンは，その表面的な速度以下に低速化することになる．

一つのアドレスに対してスタック・マシンがより少ないビット数ですむ理由の一つに，自動局所環境の概念がある．多くの変数の中で，そのプログラムを通して呼び出しできねばならない，いわゆる「広域的(グローバル)」な変数は，通常少ししかない．他のものはほとんど各サブルーチンの中の一時的な変数である．それゆえ，それらは「局所的（ローカル)」と呼ばれる．

これに対処する方法としてデータを「ページング」するという方法をとる計算機が多かった．残念なことにこれは，プログラマの注意深い配慮と，大きな（そしてそれゆえに低速であり高価な）コンパイラとある種のページング・ハードウェアとのあいだの緻密な協力が必要である．

一方，スタック・マシンはこの広域・局所の分化に理想的なほど合っている．**現在の局所環境の先頭を示すCPU レジスタをおく．** すなわち，局所変数と引数はこの環境ポインタ（EP）を通して呼び出される．この EP は，あるレジスタ・マシンのベースレジスタと似ているが，ハードウェアによって自動的にセットされる点で異なっている．サブルーチンの入口では，サブルーチン引数そして旧 EP，次に帰り番地が順にスタックに push される．現在入っているサブルーチンに対する新 EP は，これらの語（スタック・マーカーワード）をさすことになる．そしてスタック上にゼロを push するか単にスタックポインタの値を増すことによって，局所変数に対する領域を割りつける．

* （訳注） 広域的環境ポインタ：広域的な変数はスタックの一番底にとられている．これらを参照するためのポインタとしてスタック領域の一番底をさすポインタをおいておけば，そのポインタからの隔りで広域変数を直接参照することができる．このポインタを広域的環境ポインタと呼んでいる．

** （訳注） この2つの計算機の場合，スタックトップをさすスタックポインタと，現在対象となっている局所環境の底をさす環境ポインタと，広域的変数参照のための広域的環境ポインタの3つのハードウェアレジスタによってスタックが管理されている．後に示す他のスタック計算機では，さらに多くのレジスタを用意しているものもある．

```
begin
  real A, B, J, K;
  procedure P(X, Y);
  begin
        ⋮
     A←A+B;
        ⋮
  end P;
  procedure R1 (…);
  begin
    real A, B;
        ⋮
     P(J, K);
        ⋮
  end R1;
  procedure R2 (…);
  begin
    integer A, B;
        ⋮
     P(K, J);
        ⋮
  end R2;
        ⋮
end.
```

図 3 ブロック構造と環境

このようにして，スタック・マシンでの番地はほとんど，環境ポインタからの隔りだけか，広域的環境ポインタ*からの隔りで表現できる．すべてのサブルーチンではそれぞれ128以上の変数が必要なことがほとんどないならば，7ビットの番地で十分となることが多い．このことはスタック・マシンに一層の効果をもたらす．環境ポインタの内容はハードウェアで調整されるので，「ベース」レジスタを参照するのにレジスタ・マシンで必要な4ビットあまりのものは不要である．B5700 と HP 3000 はこうした自動調整された EP だけによっている**.

●ブロック構造の環境

いままでのところでは，サブルーチンの出入りや変化する環境に対するアクセスの正確なメカニズムはあいまいなままであった．新しいハードウェア構成法について述べる前に，入れ子になったブロックと入れ子になった手続きの概念から始めることにしよう．なお，Pratt の本[7]は大変すぐれているので参照してほしい．

図3は，ある ALGOL プログラムである．その主プログラム中には手続き $R1$，$R2$，Pが宣言されている．Pが $R1$ または $R2$から呼び出されるとき，Pの実行が完了したときに制御を正しいほうへ，つまり $R1$か $R2$のどちらかへ返せるように帰り番地をスタックしなければならない．

しかしながら，P中で参照されている変数A，Bは，P中では宣言されていないことに注意する必要がある．ではそれぞれは何を意味するのだろうか？ Pが $R1$ から呼ばれたのならば，$R1$ の **real** A および B が，$R2$ から呼ばれたのならば，$R2$ の **integer** A および B が，それにあたるだろうか．あるいは外側のブロック中のAとBを指すのだろうか．

ALGOL では，変数がある手続き中で参照されてはいるが，そこで宣言されていない場合，その手続きが呼ばれた場所からではなく，その手続きの宣言の外側（コンパイル時の外側）に入れ子されているとして探され，解決される．このようにして第2行で宣言されたAおよびBが対応するものとなる*1.

もしこうした決定がされなかった場合，$R1$から呼び出された場合には浮動小数点用の命令が，$R2$から呼び出された場合には固定小数点用の命令がPのために必要となる（多くの計算機においては*2）．そして少なくとも実行時になんらかの形で，変数の型のチェックが必要となる．

*1 （訳注） こうした**静的結合** (static bind) は一般的なものである．これに対して，たとえば Lisp では呼出し手続き中の A と B が実行時に呼出し順序に従って結合 (bind) され，AとBが何になるかは実行時にならなければ決定されない．こうした結合は**動的結合** (dynamic bind) と呼ばれる．Aho らの "Principles of Compiler Design", Addison-Wesley, 1977, などにも説明がある．

*2 （訳注） タグ付き計算機の場合には，対象となるデータの型がデータ語につけられており，データ型によらず同一の加算命令でよい．

そこでわれわれは，環境には二種類あることに気がつく．つまり帰り番地に関連した**動的環境**と，広域的環境に関連した**静的環境**である．現存するスタック・マシンの設計者は，静的環境に対するアドレッシングをまったく異なった方法で処理している．実行時に，局所変数と広域変数は速く呼出しができなければならないが，その中間の変数のアドレッシングについての合意は存在しないようだ．つまり，三つまたはそれ以上の環境レベルがあるとき，それに対してハードウェアで対処すべきかソフトウェアで処理すべきなのだろうか？ 現在実行している手続きの外側だが，最も外側のブロック中ではないものに対するアドレッシングは，**アップレベル・アドレッシング**と呼ばれる．またサブルーチンの出入りの仕方については，異なった方法が数多く存在している．

これらの違いについては次節（次号）で議論しよう*3.

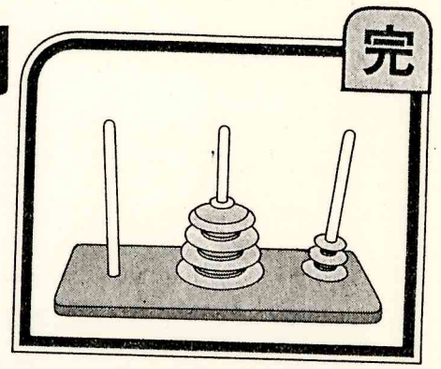
*3 （訳注） 「環境」には2種類あることに重要な意味がある．そしてスタックの内部のデータを参照するための機構をどうするかということも重要なポイントである．

参考文献

1) バロース：バロース B5500, 参照マニュアル，デトロイト，1968.
2) ヒューレット・パッカード：HP 3000, システム参照マニュアル，Cupertino, 1973.
3) C. Gordon Bell and Allen Newell : Computer Structures : Readings and Examples, McGraw-Hill, ニューヨーク，1971.
4) D. E. Knuth : The Art of Computer Programming, Vol. 1, Addison-Wesley, 1968.
（邦訳は，広瀬健・米田信夫・筧捷彦訳「基本算法」1, 2, サイエンス社）
5) B. A. Wichmann : ALGOL 60 Compilation and Assesment, Academic Press, 1973, p. 207.
6) N. Wirth : "Stack vs. Multiregister Computers", ACM SIGPLAN Notices, March 1968, pp. 13–19.
7) T. W. Pratt : Programming Languages : Design and Implementation, Prentice-Hall, 1975.

（[全3回] 訳 いだ まさゆき 青山学院大学）

スタック計算機入門



デビッド・M・バルマン

訳 井田昌之

スタック計算機のいろいろ

1) バロース B5500

B 5500 (そして B 5000 と B 5700)[8] と HP 3000 とは，よく似た構造をしている．中でも，この二つの計算機の「アップレベル・アドレッシング」[*1] の方法は同じやりかたである．すなわち，一番内側で実行されているブロックまたは手続きの外ではあるが完全に広域的ではない変数は，**静的連鎖(スタティック・チェイン)**を介してソフト的に呼び出される．これを考えた設計者は，ほとんどのプログラムは，2つ以上の入れ子レベルになることはなく，したがって中間の入れ子レベルの変数に対する機構として高価なハードウェアを持つことは賢明でないと感じていた．この論点は FORTRAN のような原始的な言語では確かに妥当のようだ．ただし，B 6700 では他の手法をとっている．

B 5500 のスタックの構造を図1に示す．他の図の場合にもいえるが，構成法を理解するのに不要な細かな点は省いてある．

B 5500 の場合，広域変数と広域手続き名はスタック自身の中にはなく，プログラム参照表 (PRT) の中にある．PRT の先頭番地は CPU 中の R レジスタ (rR) で指し示される．つまり rR は広域環境のポインタの役割を果たしている．PRT を，主記憶の離れた場所にたまたま置かれたスタックの一部だとみなせば，構造が単純になり，理解がしやすくなろう．

B 5500 にはスタックを指す2つの CPU レジスタがある．一つは S レジスタ(rS)であり，最高位の語の番地(アドレス)を指している．もう一つは F レジスタ (rF) であり，局所環境ポインタとして使われる．通常，rF は最上位の戻り制御語 (RCW)[*2] を指しているが，一時的に最上位の マークスタック 制御語 (MSCW) を指すこともある．

さて，プログラムを実行し始めるときには，rF はスタックの底を指している．このスタックにプログラム実行中の式の評価に必要なさまざまな一時的な変数も保持される．一方，手続きやサブルーチンを呼び出したときは，次のような複雑な手順をとる．

(1) マークスタック制御語 (MSCW) をスタックへ push する．この MSCW はそのときの rR と rF の値といくつかのフリップフロップの値によって構成される．次に rF がこの MSCW を指すように変えられる．

(2) 呼び出されているサブルーチンの引数をそのスタックに push する．

(3) 次に，戻り制御語 (RCW) を作る．RCW はそのときの rF と帰り番地といくつかの CPU ステータスからなる．現在 rF はステップ(1)で作られた MSCW を指しているが，次に rF はこの新しい RCW を指すように変えられる．

(4) 呼び出されているサブルーチン中で宣言された局所変数の記憶域を，この RCW の上に割りつける．

(5) サブルーチンの実行が開始される．

トップ・オブ・スタックポインタ，つまり rS は常にスタック・トップを指すようになっている．上記の過程のときでも rS はスタック・トップを常に指している．

もし，他の1つまたは複数のサブルーチンが実行されているあいだにあるサブルーチンが呼び出されるならば，そのたびごとに同一の過程がとられる．その結果，「入ってはいるが，まだ出ていないさまざまな環境をさかのぼることができるようなポインタの連鎖」が残される．現在の環境からこの連鎖をさかのぼることにより，中間的な環境を参照できるようになる．

2つの最も頻繁に呼び出される環境は即座に使用できるようになっている．すなわち，広域的な変数は rR を参照して呼び出せる．引数や局所変数は局所環境に含まれる．引数は rF からの負のオフセットによって参照される．局所変数は rF からの正のオフセットで参照される．

このようにしてハードウェアによって自動的にロードされ，セーブされる2つのベースレジスタ (rR と rF) を利用して，即座に3種の最もよく用いられる値を呼び出すことができる．明らかにサブルーチンの出口では，この逆の過程をとればよい．つまり，以前の CPU の状態に戻すには各制御語から rF を再ロードするわけである．

すべてのスタック操作は主記憶上のスタックに対して行なわれるが，CPU に高速な一時記憶があるなら計算をもっと速くできる．これが図1中のAレジスタ(rA)とBレジスタ (rB) の目的である．**この2つのレジスタはスタックの CPU 内部への延長部分として働き**，すべての操作は実際にはこのレジスタに対して働く．つまりこの2つのレジスタは CPU によって自動的に割りつけられたアキュムレータあるいは汎用レジスタと考えてもよい．これらのレジスタはコンパイラなどからもかくされている．**スタックへの push は実際には rA または rB に置かれるのである**．どちらも空でないならば，rB が主記憶上の部分へ移され，空がつくられる．rA または rB が空なときにスタック操作 (たとえば乗算) を実行しようとするなら，その両者が必要なオペランド (被演算数) を持つように調整される．スタックの調整が常に行なわれるわけではない．

乗算が実行されたときに，たとえば rA が空で積が rB にある場合，空であるレジスタを自動的に主記憶上のスタック域から満たしはしない．なぜなら，次の操作がスタックに新しい値を push する可能性も等しく存在するからだ．

スタック・マシンはレジスタ・マシンに比べ，割込み処理に向いていると思う．B 5500 は割込みを単に，

「予期しない手続き呼出し」，

として扱う．B 5500 が割込みを認識したとき，割込み戻り制御語 (IRCW) がスタックに push され，割込みが処理される．

このように汎用レジスタ・マシンでは大変時間のかかる仕事である CPU の状態のセーブを，B 5500 の場合は，大変効果的に行なう．IRCW は通常の RCW とほんのわずか異なっているだけなので，**割込みを「予期し**

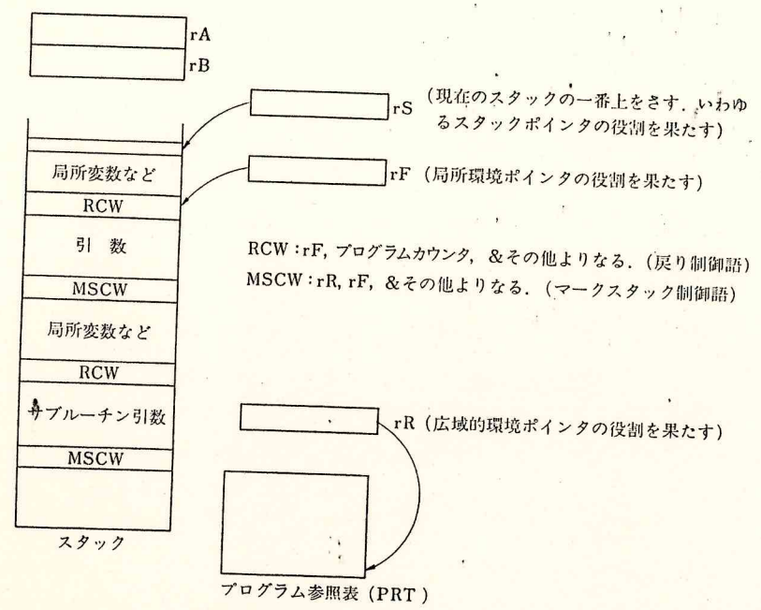


図1 B 5500 のスタック構造

*1 (訳注) 前回の「スタック計算機入門(2)」で定義してある．現在実行している最も内側の環境からは外(上)ではあるが，一番外側のレベル (グローバル) ではないところで定義されているものに対するアドレッシング．

*2 (訳注) 「戻り制御語」．その他のバロースに関する邦訳は，土居範久訳「計算機システムの構造」共立出版，1978を参考にして決めさせていただいた．

ない手続き呼出し」として扱うことは，機械の構造によく適合する．割込みからの復帰は本質的にはプログラムカウンタ，rF やいくつかのフリップフロップなど，IRCW に格納されたすべてのものを復元することである．機械の状態はこれによって割り込まれた時点にすべて戻される．

2) ヒューレット・パッカード HP3000

HP 3000 は図2のようなスタック構造をもつ16ビットのミニコンである．HP 3000 の構造を述べるには B 5500 とどう違うかを述べるのがよい．

広域変数を独立した領域に置く代わりに，HP 3000 はそれらをスタックの底におく．このことによって概念的にもすっきりとする．

B 5500 と HP 3000 のレジスタのあいだの直接の対応はない．したがって，この2つの計算機の各レジスタを同一に考えることはできない．対応があるものは図から明らかである．

スタックの底（ベース）は DB レジスタ(rDB)によって指されている．スタック領域の上限は Z レジスタ(rZ)に入れられている．これによってスタックあふれエラーはハードウェアで発見できる．SM レジスタ（rSM）はメモリ中のスタックの最上位の語を常に指している．プログラムの実行がはじまるとき rSM は広域変数領域のすぐ上を指している．最初のQレジスタ（rQ）の位置もここを指している．rQ は，サブルーチンに入ると次の手順によって影響を受ける．

（1） そのサブルーチンへの引数をすべてスタックに入れる．

（2） 帰り番地，インデクスレジスタ，デルタQ（その他のスタックマーカーに戻る変位）そして CPU の状態からなる4語のスタックマーカーが形成される．このスタックマーカーはスタックに push され，rQ がこの新しいスタックマーカーのトップを指すように変更される．

（3） サブルーチンに対するすべての局所変数をこのスタックマーカーの上に割りつける．

（4） サブルーチンの実行を開始する．

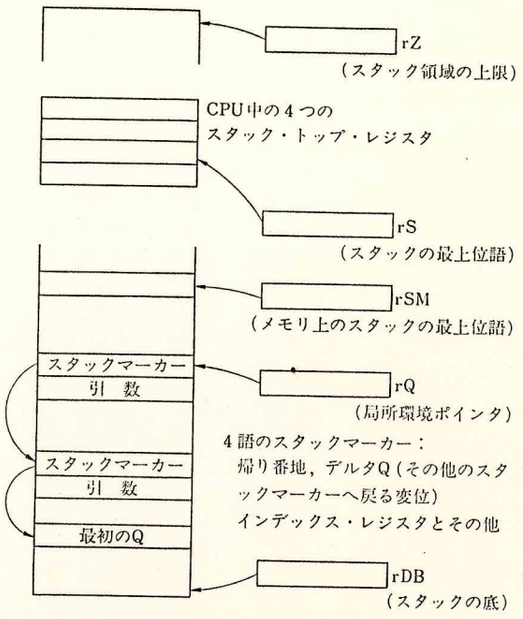


図 2　HP 3000 のスタック構造

スタックポインタ rSM はもちろん常にスタック・トップを指すように調整されている．B 5500 はサブルーチンの入口に MSCW と RCW をもっているが，MSCW はこのルーチンに入る前の状態に戻れるように，引数を他のものから独立させる役割をもっていた．しかし，HP 3000 では一つのスタックマーカーしかない点に注意してほしい．

では HP 3000 はこのことをどうやって行なっているのだろうか？ サブルーチンの出口のための命令は，そのオペランドにスタックマーカー以下の削除されるべき語の数を持っているのである*1．

B 5500 でのように主記憶上の3つの最もよく用いられる領域は，2つの CPU レジスタ中に保たれたベースを通して即座に呼出しできる．広域的領域は rDB からの正のオフセットで呼出しできる．局所的な環境は rQ を通して参照できる．スタックマーカーの下のサブルーチン引数は rQ からの負のオフセットで呼び出せる．そしてそのサブルーチンの局所変数は rQ に対する正のオフセットを通して到達できる．すべての中間的な段階にある変数は一番上のスタックマーカーからポインタの連鎖をさかのぼることによって呼び出せる．

HP 3000 はスタックの上部を主記憶ではなく，4つの CPU レジスタ中に保持するようになっている．4つの CPU レジスタにより実行時間をかなり高速化することができることが，この本*2 の Blake の記事で述べられている．これらのレジスタはプログラマには通常見えず，ハードウェアによって自動的に調整される．

計算中の状態（ステータス）は基本的にはスタック中におかれるので，HP 3000 は割込みを認識したとき，四語の状態語をスタックにセーブするだけで高速に割込み処理を行なうことができる．

*1（訳注） Pascal の最も単純な処理系である Pascal-P で考えられている仮想スタック・マシンでも同様になっている．P コードの RET(復帰)命令にはオペランドに削除されるべき語数（スタックマーカーの語数と引数の語数）を持たせている．特に HP 3000 に対してだけではないが，Pascal-P を考えあわせながら眺めていくと大変おもしろい．

*2（訳注） 原書 COMPUTER Vol. 10, No. 5.


```
BEGIN
REAL A,B;
REAL PROCEDURE GLOP(X);
  BEGIN
      .
      .
      .
  END GLOP;
BEGIN
    REAL B,C;
    .
    .
    .
    BEGIN
     REAL C,D;
     .
     .
     .
     A:=A*B+GLOP(D);
     .
     .
     .
   END;
   .
   .
   .
END;
.
.
.
END;
```

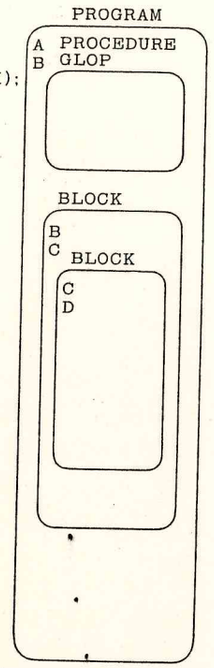


図 3　入れ子になった ALGOL プログラムの例

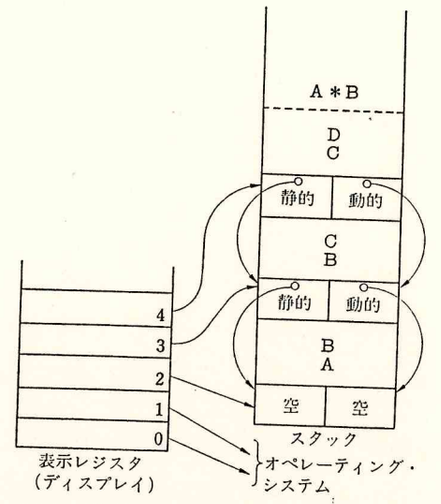


図 4　GLOP の実行直前

3) バロース B 6700 と 7700

バロース B 6700（そして B 6500 と B 6800）[*] は，大型の B 5500 と考えることができる．B 5500 は 32K 語（1語 48 ビット）と2つの CPU までを持つことができるが，B 6700 は1メガ語（1語 51 ビット——48データビットと3タグビット）の記憶装置と3つの CPU までを持つことができる．

ここでの興味の対象となる点は，B 6700 のスタック構造がどうなっているかということである．B 5500 や HP 3000 では，中間レベルでのスタック中の要素，現在実行しているサブルーチンの外側ではあるが最も外側にないものは，ソフトウェアによる静的な連鎖によってだけ呼び出すことができた．このことは，それぞれのマシンにはアドレスレジスタが2つだけ存在したことからの結果である．つまり，一つは広域環境を指し，もう一つは局所環境を指していた．

B 6700 で付加されたハードウェア機構の一つに，各 CPU 中の表示レジスタ（ディスプレイ・レジスタ）がある．この表示レジスタは 32 個のアドレスレジスタであり，それらはそれぞれ「生きた」入れ子環境を指している．表示レジスタはサブルーチンの出入りに際して，スタックと同様な扱いをうける．DISPLAY[0]と DISPLAY[1]の2つのレジスタには，オペレーティング・システムに対する環境情報が入れられる．DISPLAY[2]～DISPLAY[31]はプログラム全体の生きた環境を反映するようにサブルーチンの出入口で処理される．B 5500 で述べた静的連鎖を構成するリンクは B 6700 においても作られるが，**B 6700 ではこれらのリンクは表示レジスタ中にそのコピーがとられている．**

このようにしてすべての要素はアドレス対（i, j）によって参照される．ここで i はこのプログラムにおける静的な入れ子レベルで，j はそのブロック中での変数の番号である．

図 3, 4, 5 は，Organick の B 6700 に対するより完全な分析から取り入れたものである．これらの図では基本的な B 6700 の構造をみるため，いくつかの細かな点は省いてある．

図3は ALGOL プログラム例であり，入れ子を表わす図をつけてある．広域的（グローバル）には2つの実変数と手続き GLOP が宣言されている．そしていくつかの入れ子になったブロックがあり，それらは各レベルごとに宣言された変数を持っている．レベルは等しい

*（訳注） E. I. Organick : "Computer System Organization", Academic Press, 1973 と思われる．

が，他のブロックと並列に宣言されたブロックもある．そうした並列ブロックの中では，宣言された変数は動的割付けされ，保護される．ブロックと手続きは大変似ている．なぜなら，ある手続きの本体は一つのブロックとして常に扱われるからだ．しかしブロックは引数を持たず，復帰のための情報を必要としない．なぜなら，それらのコードは順次実行されるからだ．

図4に，実手続き GLOP 実行直前の表示レジスタとスタックの状態を示す．GLOP は関数である．いま，A への代入文の計算の途中で呼び出されたとしよう．このとき，スタックにはその代入文を含むブロックについての情報が上に残されており，呼び出された手続きに対するものはまだ入っていない．したがって，左側の静的連鎖は右側の動的連鎖と同じである*1.

一番内側のブロックの局所変数 C と D の上に現在評価されている式 (A:=A*B+GLOP(D)) における A*B の値が一時的に置かれている．しかし最も重要なことは，3つの環境のいずれもが表示レジスタの1つを指すことによって呼出しができる点である．もちろん重複した名前があるので，コンパイラはそれらを区別し，適切な参照法を生成するものとする．

図5は，GLOP 実行中のスタックと表示レジスタの状態を示してある．GLOP のコードに対しては2つの環境だけが見えている．つまり GLOP はそれ自身の局所変数 (DISPLAY[3] を通して参照される) と一番外側のブロック (DISPLAY[2] を通して参照される) で定義

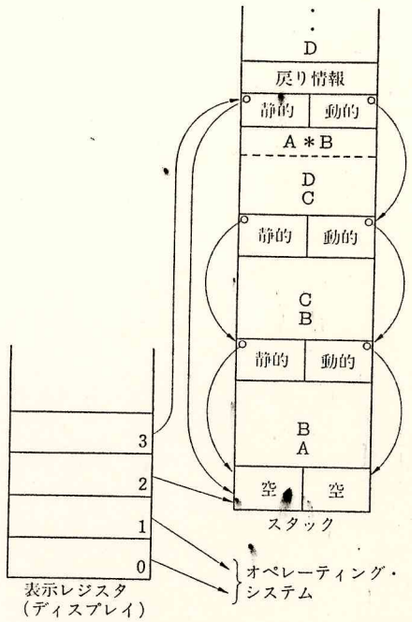


図5 GLOP の実行中

された項目を参照することができる．スタック中の他の2つの環境は GLOP には見えない．そしてその静的および動的連鎖はもはや認識できない．

GLOP の実行中には，スタックにさまざまな一時記憶がとられ，またいくつかの手続きを呼び出すかもしれないが，その実行が終了したとき A*B の上のスタック・トップにその値を残し，動的連鎖を介して呼び出した所に戻る．表示レジスタも GLOP 実行直前の状態に戻される．このときのスタックにおけるただ1つの違いは，GLOP から返された値だけである．以前の状態に表示レジスタを戻すのに必要な情報は GLOP 実行中の スタックにある静的連鎖に入っている．

ALGOL, PL/I, PASCAL のようなブロック構造言語で書かれたプログラムは，その実行に静的および動的連鎖を必要とする．ただ一つの問題は，その両者に対してハードウェアの支援があるか否かということである．

4) データジェネラル ECLIPSE

データジェネラル社は以前の NOVA シリーズにはなかったいくつかのスタック機能を ECLIPSE[10] に追加した．スタック以外の ECLIPSE の特徴はここでは述べない．また他のデータジェネラル社の計算機にもいくつかのスタック機能がつけられている*2.

ECLIPSE のスタックの特徴は，計算機の最終的な構

*1 (訳注) 動的連鎖とは，実行の前後関係を示すものであり，手続き中では，いわゆる帰り番地が相当する．また，この場合 BEGIN ブロックの入れ子なので，動的連鎖 (実行順序) と静的連鎖(変数の入れ子レベル)は同じことになる．ところが図5になると，一番内側の BEGIN ブロックから手続き GLOP が呼び出されている．このときの実行順序はしたがって，その BEGIN ブロックからの呼出しなので，その動的連鎖は A への代入文を含む BEGIN ブロックを指すことになる．しかし，呼び出された手続き GLOP の変数の入れ子関係はその BEGIN ブロックの内側ではなく一番外側のプログラムの内側でしかない．したがって，静的連鎖は図5ではスタックの底を指すことになる．

*2 (訳注) 「他のデータジェネラル社の計算機」が NOVA シリーズを意味するなら疑問がある．もっともスタック機能の意味にもよるが，2つのインデクスレジスタ(AC2, AC3)しかないし，また，よくスタックの代用して一般に利用される自動増減機能 (Auto Increment-Decrement：参照するたびに +1 または −1 する機能) は 20～27 番地の間接参照ならば Auto Increment, 30～37 番地の間接参照ならば Auto Decrement と分けられており，直接的にスタックを代行することは訳者の経験ではできなかったからである．あるいは他のシリーズがあったのだろうか？

造にソフトウェアからの要求がどう影響するかの一例となる．しかしながら，それについてメーカはほとんど述べていない．NOVA のシステム・プログラムに用いられている手続き呼出し手順に詳しい者は，ECLIPSE に導入されたスタック操作にすっかり満足する．**スタックポインタ，フレームポインタ，スタックリミット，そしてスタックフォルトアドレス**が4つの固定された記憶域に割りつけられる．これら4つの値はファームウェアによって保守される．スタックポインタは現在のスタック・トップを示すために用いられる．フレームポインタは局所変数を呼出しできるように，スタック中の最も内側の局所環境の先頭を指している．ファームウェアはスタックへの push のたびにスタックリミットと比べ，スタックあふれをチェックする．そしてあふれた場合には，スタックフォルトアドレスに保持された場所に分岐する．

ECLIPSE では，スタックを利用するいくつかの命令がファームウェアで構成されている．たとえば，「復帰ブロック」を構成し，それを主記憶中のスタックに push する命令がある．復帰ブロックは帰り番地と4つのレジスタからなる．復帰ブロックをスタックから pop し，レジスタやプログラムカウンタを再格納する命令も存在する．復帰ブロックと局所変数に対する領域をフレームといい，フレームポインタによって指される．復帰ブロックを push し，語数をスタックポインタに加算することによって局所領域を割りつける SAVE という命令も存在する．

5) DEC

PDP-11 にはスタック能力があるとしばしばいわれている．たしかにシステム・プログラマに対するいくつかの補助手段があるが，スタック機能は初歩的なものである．通常述べられる特徴というのは，オートインクリメントのインデクシングである．つまり次のようなものが書ける．

ADD (Rt)+, @Rt

この命令は，インデクスレジスタ Rt を用いてメモリをアドレスし，スタック的に働くように Rt の内容を自動的に増加する．次の手順にまとめられる．

temp :=M[Rt]
Rt :=Rt+1
M[Rt]=M[Rt]+temp

この複雑な操作の結果として主記憶中に下へのびるスタックを形成し，その上でスタック加算が実行される*1.

まず，スタックポインタとして利用されるレジスタ Rt で指された場所の内容が一時的に確保される．次にスタックポインタが1増加され，結果としてスタックの一番上のものが除去される．次に temp に確保された旧スタック・トップが新スタック・トップに加えられ，その和が再び新スタック・トップに置かれる．

『このことはスタック機能をまったく持たないことよりよいが，いくつかのメモリ参照を命令中で直接に行なうことになる．』

サブルーチンの入口におけるスタックのマーキングやサブルーチンの出口におけるスタックの復旧に対する手段がないので，PDP-11 をスタック・マシンとみなすのは原則として理屈に合わない．こうした意味での PDP-11 に関する興味深い解説は Hamlet の論文[11] を見るとよい．

PDP-10 にもいくつかの有用なスタック命令があるが，スタック・マシンと考えることはできない．

6) テキサス・インスツルメンツ TI 990 と TI 9900

このシリーズの TMS 9900 や他のものは，スタックに関して大変興味深い構造を持っている．TMS 9900 は CPU に3つのレジスタしかない．ステータスレジスタ，プログラムカウンタ(両者の解説はここでは行なわない)とワークスペースポインタ (WSP) である．
WSP は 16 個の 16 ビット汎用レジスタとして利用される主記憶中の一連の領域を指し示す*2.

「文脈の切換え」のための命令がある．それは WSP に対して主記憶中の新しい領域を指させ，旧値を新領域中のある場所におく．以前の文脈に戻るためには現在のワークスペースから WSP をリロードする．これらの命令は規律だった方法で用いなくともよい．しかしそれらは環境 (局所変数など) のスタッキングを適度なソフトウェアの努力だけで実現するためのすべての能力を備えている．

7) バロース B 1700 SDL 計算機

B 1700 は，いろいろの会議でも紹介されている[12),13)]．この機械ではプログラムのそれぞれの実行時にその構造

*1 (訳注) 同様の機構はいくつかの他の機械でも見られる．メーカによって特にとなえられてはいないが，訳者の確認した範囲でも，たとえば16ビットマイクロプロセッサ μCOM-1600 (「16ビットマイクロプロセッサ新時代」**bit** Vol. 10, No. 7, 1978 などを参照) にも存在する．

*2 (訳注) このことは「TI 9900 には 16 個の 16 ビット汎用レジスタがあるが，それらは主記憶上に置かれている」とだけ説明されている場合が多い．

(アーキテクチャ) が決定される.

ハードウェアはマイクロプログラム化されている. その上にそのマイクロプログラムを保持する記憶装置は, 実行時に動的に書き換えられる点に大きな特徴がある. 多くのマイクロプログラム化された計算機と同様に, B 1700のマイクロプログラミングは, ハードウェアそのものよりもソフトウェアにふさわしい機械語を提供するように利用されている.

B 1700 は実行時に命令を変えることができるので, いくつかの異なったプログラム言語を同時に使えるように命令セットやアーキテクチャを妥協させる必要がない*. その代わりに現在走っているプログラムの型によって異なったマイクロプログラムを制御記憶上にロードする.

B 1700 のマイクロプログラムによって作り出されるアーキテクチャの1つにスタック・アーキテクチャを持つものがある. ソフトウェア開発言語機械 (SDL マシン) がそれである. 不幸なことに, SDL マシンの詳細な, そして完全な唯一の説明書[14]は出版されていない. McCrea のこのすばらしい文献は SDL マシンのスタック・アーキテクチャと命令セットについて明解な説明を与えている.

この機械は, 複数個のハードウェア (ファームウェア?) スタックを持つ唯一のものであるという点で特に興味深い. このスタック構造は, Randell と Russell によって述べられたコンパイラ・プロジェクト[15]に基づいている. 大変似たアーキテクチャの完全な説明が McKeeman によって与えられている[16]. ここでは, McCrea の未公開の文献に若干手を加え, 簡略化し, SDL マシンの説明を行なおう. その中で用いられる「ディスクリプタ(記述子)」という言葉は, ある項目に対するポインタとさらにいくつかの情報の集まりを指す.

SDL マシンは, 5つのスタックと1つの表示レジスタ群をもっている. それぞれの目的は次のようなものである.

名前スタック: ブロックや手続きに入ったとき, 局所的に使われていた変数に対するディスクリプタがここに push される.

値スタック: 名前スタックにディスクリプタが push されたとき, それらによって示されるデータがこのスタックに push される.

評価スタック: 式の中で用いられるデータのディスクリプタはこのスタックに push される.

制御スタック: 他のスタックのいくつかの動的な歴史が管理される.

プログラム・ポインタスタック: 手続きとブロックの出入りについての動的な歴史がこのスタックに保持される.

表示レジスタ: この配列は B 6700 に関して述べられたものと本質的に同一のものである.

通常のスタック・マシンでは, 1つのスタックに入れるものをこのようにいくつかのスタックに分離し, それぞれを異なった目的で使用することは B 1700 の上でうまくいっている. しかしながら, B 1700 が任意ビット長の項目を呼び出し, 処理できる能力を持つことにどの程度依存しているかは明らかではない. それぞれのスタックは必要な語長だけで構成されている.

B 1700 の可変フィールド機能によって可能となった, 命令のほとんど最適と思われるコード化を除いては, SDL マシンでの手法は B 6700 のそれに類似している.

8) マイクロデータ 32/S

このあまり知られていない計算機[17]はマイクロデータ 3200 をマイクロプログラム化することによって作られており, より深い研究の価値がある. この機械はマイクロデータ・プログラミング言語 (MPL) にふさわしいような高級言語マシンとして設計されている. B 5000 や HP 3000 のように中間レベルの環境は静的連鎖をたどるように設計されている.

9) マイクロコンピュータ

モトローラ 6800, インテル 8080, その他のマイクロコンピュータは, ある種のスタック能力を持っている. 多くの場合それはスタックとして用いられる主記憶中の領域を指す MPU 中のスタックポインタであるか, あるいは MPU 自身の中の小さなスタックである. このスタックに対して通常用意された操作は次の3つに分類できる.

① 最小限の機能としてサブルーチン呼出しによって帰り番地がスタックに push される.

② MPU のレジスタすべてと帰り番地とをスタックに push する命令が存在する.

③ 中間結果をセーブできるよう, それぞれのレジスタを push したり pop したりできる.

これらの能力はマイクロプロセッサをスタック・マシンと呼べるほどのものではないが, 正しい方向への第一ステップということができる. マイクロプロセッサの製作者たちは, おそらくここ数年の計算機構造についての不活発さを正す機会を探ろうとしていると期待したい.

* (訳注) たとえば FORTRAN の実行に便利な機械語命令のセットと, COBOL の実行に便利な機械語命令のセットは異なっている. また, 実行に有利となる計算機の構造も異なっている. このため, 汎用機の構造と命令セットは妥協の産物となるか, あるいはすべてをとりこんだような大きなものとなる, ときびしい意見を述べているようだ.

●インデクスレジスタ：存在すべきか否か？*

スタックについて議論するとき, インデクスレジスタの話を避けるのはむずかしい. 多くのスタック・マシンはインデクスレジスタを持たないが, HP3000 は1つだけ持っている. スタック・マシンにおいてインデクスレジスタがいかに価値があるかを述べるのは難しい. HP 3000についての研究によれば, インデクスレジスタは大変効果的であることを示しているようだ. それらに対する意見もいろいろとゆれている.

スタック・マシンに対するインデクスレジスタの価値を疑う理由のいくつかは, 次のようなものである. ブロック転送や走査の命令, (そしておそらくは他のベクトル命令) の存在があればこうしたレジスタの必要性の多くを取り除く. 彼らの主張における速度に関する議論は, もし十分な数の CPU レジスタ (たとえば 16 とか 32) がスタック・トップのために用意されているならば, おそらくは消滅するだろう. このようにしてインデクスレジスタ (としての語) も自動的に割りつけられる. たとえば, バロース B 7700 はスタック・トップの 32 語を CPU 中のレジスタに保持しており, またいくつかのベクトル命令を持っている.

●スタック・マシンの将来は？

Doran は, スタックに関してコンピュータ・アーキテクチャの現状を次のようにうまく解説している[18].

『一般にハードウェアは, ソフトウェアの要求に合致するように変わっていく. スタックはソフトウェアにおいて広く用いられているが, 多くの計算機はハードウェア・スタックを持っていない. この矛盾の理由は, 多くの小型および中型計算機の製作者たちが60年代初期からのある期間, 構造的な改革を省略し節約してきたからである. ソフトウェアからの要求も大きくなったので, 古いアーキテクチャの高速化が進められ, 大きな変更もされなかったのであろう. 当時の設計者は, 現在の生産ラインがスタックの重要性に気づくことになるとは十分把握していなかったようだ. ここで議論された計算機の多くはそうした期間のあとに作られたものであり, そしてそれらはおそらくスタック・マシンあるいはスタックのより一層の利用という傾向を表わしている.

こうした方向を支えるソフトウェアの傾向としてシステム・プログラミングに対する PL/S のような高級言語の使用が挙げられる. システム・デザイナたちはハードウェアの設計というすべてのプログラマたちに最も影響を与える機会を持っている. そして彼らが再帰的な, そしてブロック構造の言語を用いるならば, 彼らはスタックを必要とすることになるだろう.』

スタック計算機を長いあいだ使ってきた者にとって, 機械語の互換性の要求を除いてなぜスタック・マシンが一向に作られないかという理由を理解するのは難しい. 経済的な現実性からも業界において高級言語ですべて記述されるようになれば, 機械語についてのそうした残された論点は近い将来消えるだろう. もちろん B 5000 の設計者や製作者たちはこのことを 1960 年に言っていた.

5. 関連する概念

●タグ付きメモリ

スタック・アーキテクチャとこのタグ付きメモリの関係は, それらがともに進んだ, しかしあまり用いられていない概念という点だけであろう. タグ付きメモリは, 主記憶にたとえば3ビットないし6ビットの小さなタグをつけるというアイディアである. 最も完全な議論は Iliffe によると思われる[19]. しかし Doran の B 6700 と ICL 2900 シリーズとの比較[20]にはより新しい取扱いがなされている.

このアイディアの特徴は, 異なった種類のオペランド (固定小数点, 浮動小数点, 10進など) に対して, あるい

* (訳注) 原文は「to be or not to be ?」
スタック・マシンは語単位の処理を基本としているため, ベクトルその他の配列型のデータを保持し, 高速に参照することは得意ではない. 多くの場合, その配列に対するディスクリプタをスタックし, 実際のデータ領域はその外部にとられることになる. 配列中の各要素の参照は, したがってディスクリプタからそのベクトルの先頭番地をとりだし, 与えられた添字のチェックをし, 対応する番地を算出し, その語を読み出すことを行なうことによりなされる. 二次元配列の場合は, ディスクリプタの参照は二段に行なわれることになり, さらに時間がかかる. インデクスレジスタがあれば, 添字からオフセットアドレスを計算し, 直ちに求めるアドレスをうることができる. しかし, 単純なインデクスレジスタの使用は誤った添字の使用などにより, 簡単にプログラムをこわしてしまう結果となる. これらのことの上にこの節の議論がなりたっている.

は異なった長さのオペランドに対して，独立した命令を持つより，むしろそのときその命令が扱っているオペランドのタイプを知らせるような小さなタグをデータ語に付加するという点にある．これは命令コードを表現するのに必要なビット数を明らかに減らし，コードの圧縮をより進める．タグ付きアーキテクチャの計算機に対するコンパイラも，また小さくそしてより速くなる．

●可変長オペランドとオペランド依存型命令

可変長オペランドについての議論は，常にその速度や効率についての疑問に向かっていくようだ．可変長オペランドを計算機が処理するには速度の犠牲がともなうと思われているが，これは外見だけで実際のものではない．Barton はいくつかの興味深いアイディアを持った小論文[21]を書いているが，その中で次のように述べている．

『機械を直接扱うプログラマは，機械と用いるデータ構造をどう適合させるかについての選択権を持っている．逆にそのことから，表現範囲の制限の設定，主記憶の無駄使い，無駄なフィールドパック・アンパック操作などを行ない，プログラムに必要な記憶装置や増加する実行時間を無駄にしている．独立した大きさの情報が扱えれば命令は短くなり，単純化できる．』

Jack Lipovski による価値ある文献[22]は，マイクロプロセッサにおいてスタック・アーキテクチャと可変長オペランドをどのように結合するかを示している．

謝　辞

これを書くにあたって，Doran と McCrea の文献，そして Organick の本によるところが大きく，深謝の意を表する．また Barton との議論は計算機の構造原理を明確化するのに役立った．

＊（訳注）　たとえば，B 6700 の加算命令は，浮動小数点データに対しても固定小数点データに対して働く．B 6700 のデータ語は，48ビットのデータ語と3ビットのフラグを持つ．このフラグを見て対応する型合せが自動的に行なわれる．また，フラグとして制御語が示されている語をオペランドにしようとすると，イリーガルフォルトが生じる．

参考文献

参考文献（(7)までは第2回に掲載されているので省略）

8) R. S. Barton: "A New Approach to the Functional Design of a Digital Computer", Proceedings of WJCC, 1961, pp. 393-396.
9) E. I. Organick: "Computer System Organization", Academic, Press, 1973, (邦訳は土居範久訳「計算機システムの構造」共立出版，1978).
10) Data General: Programmer's Reference Manual—— Eclipse Line Computers, 1975, (邦訳は，日本ミニコン「ECLIPSE 解説書」015-500024-02).
11) R. Hamlet: "The PDP-11 as B 5500 in Teaching Systems Progamming", ACM SIGPLAN Notices, 1976, pp. 47-52.
12) W. T. Wilner: "B 1700 Memory Utilization", AFIPS Conference Proceedings, 1972 FJCC, pp. 579-586.
13) W. T. Wilner: "Design of the B 1700", ibid, pp. 489-497.
14) D. R. McCrea: "The SDL Virtual Machine for the Burroughs B 1700 Computer", 未公開文献.
15) B. Randell and L. J. Russell: "ALGOL 60 Implementation", Academic Press, 1964.
16) W. M. McKeeman: "Stack Computers", (in) Introduction to Computer Architecture, H. S. Stone, ed., Science Research Associates, 1975.
17) Microdata: "Computer Reference Manual —— Micro 32/S", 1974.
18) R. W. Doran: "Architecture of Stack Machines", (in) High-level Language Computer Architecture, Yaohan Chu, ed., Academic Press 1975.
19) J. K. Iliffe: "Basic Machine Principles", American Elsevier, 1968.
20) R. W. Doran: "The ICL 2900 Computer Architecture", Computer Architecture News, ACM SIGARCH, pp. 24-47, 1976, p. 106.
21) R. S. Barton: "Ideas for Computer Systems Oganization: A Personal Survey", (in) Software Engineering Vol. 1, T. Tou, ed., Academic Press, 1970 p. 14.
22) G. J. Lipovski: "On a Stack Organization for Microcomputers", (in) Microarchitecture of Computer Systems, R. Hartenstein and R. Zaks, eds., American Elsevier 1975.

〔完〕

（訳　いだ まさゆき　青山学院大学）

第12回 情報科学若手の会シンポジウム開催

開催期日　昭和44年7月12日(木)～14日(土)
会　　場　鞍ケ池ロッヂ(豊田市矢並町法沢)
定　　員　40名(22歳～28歳位までの方)
参加資格　情報科学およびその関連分野で研究・実務に従事している若手研究者・技術者
参 加 費　8000円(遠距離の方には，交通費の一部補助を予定)
申込み方法　A 4 判用紙に氏名，所属，学生，連絡先，電話番号，現在関心ある分野と討論したいテーマを記入の上送付のこと
締切期日　昭和45年6月末日(必着)
申込み・問合せ先　〒464 名古屋市種区不老町
名古屋大学 工学部 情報工学教室
福村研究室内　若手の会事務局
Tel　052-781-5111 内線5804 (岡崎)