

1. FORTH が生まれた世界

1970 年代のミニコン上のプログラミング環境を感じられるプロセスを、今風の自作ワンボードマイコンと母艦 PC の組み合わせの上に実現し、68000 機械語・アセンブリ言語で FORTH のワードを整えてゆきます。

私にとって FORTH の魅力は、言語としての側面よりも、実行環境としての側面にある。

- サブルーチンのアドレスを 2 重ポインタでたどって呼び出してゆく、シンプルな「インタプリタ」。
- 機械語サブルーチンではあるが、部品化して組み合わせるための仕組みである「単語」「辞書」「コンパイル」
- これらの単純な仕組みの組み合わせで「アプリケーション志向」でアプリケーションにくみ上げてゆくプログラミングスタイル。

UNIX でプログラミングに目覚め「小さな部品を組み合わせで問題を解いてゆく」スタイルがしみついている、今でもシェルスクリプトでバッククォート、sed を日常的に使う私にとって、もうひとつの「部品を組み合わせで問題を解いてゆく」仕組みとして魅力的である。UNIX がない時代の「部品箱」アプローチはとても興味深い。

FORTH を発明した Charles Moore は、機械語サブルーチンの束とインタプリタを引っ提げて、顧客の求めるミニコンに開発環境を一から移植し、その上にアプリケーションを組んだ。この作業を数 10 種類のミニコンに対して繰り返す中で、FORTH 言語が形作られていった。

ならば、私も彼のたどった道をたどろう。新しい CPU に対して、アセンブリ言語を使って機械語サブルーチンを一つ一つ作ってゆく。最終的に FORTH 言語が動くようにする。彼が数 10 回たどった道だが、私がさらに 1 回たどってもいいじゃないか。このシンプルな開発環境・実行環境を体で味わいたい。ということだ。

- サブルーチン先頭のアドレスの列をソフトウェア部品として実行する。実行は 2 重ポインタの参照なので実行コードへのジャンプは高速である。
- プログラムはテキストで表現されているが、アドレスの列に変換した状態でメモリ上に展開しているので高速実行できる。
- テキストを解釈する際は空白で区切られた「単語」単位に区切り、単語に対応する部品のアドレスを見つけ出し実行する。

幾つかの機械語部品以外は、すべて数 10 個のアドレスの列(ポインタの配列)なのでコンパクトであり、当時の限られたメモリ容量でもある程度の規模のアプリケーションを組むことが出来た。

数キロバイトのメモリ、ハードディスクと低速の端末、機械語によるプログラミングで腕を磨いた Moore は、自分独自の開発環境を作り上げてゆく、さまざまなマシンの仕事を受け手元のソフトウェア部品をインタプリタで組み合わせ仕事をごなしでゆく中で彼が作り出したものが FORTH という開発環境だった。60 年後半から 70 年代前半にはコンパイラやライブラリは提供されているがコード品質はよくわからない状況で、自分の手元にある機械語サブルーチンの束を組み合わせ顧客要望に応える。新たな CPU の仕事を受けるとアセンブラを作るところから始め、用意した機械語ルーチン群を「移植」してゆく。数 10 種類のコンピュータに対してそれを繰り返す中で FORTH という仕組みが出来上がっていった。

- アドレス解決済のサブルーチン呼び出しの列を順次呼び出してゆく実行方法(内部インタプリタ)
- 入力プログラムを単語に区切って、単語を名前-コードのペアのエントリの片方向リスト(辞書)で検索して見つかったコードのアドレスを実行する(外部インタプリタ)
- 単語の列で構成されたモジュールを、辞書で検索してコードのアドレスをメモリ上に順次並べ、そのリストを新しい辞書エントリとする(コンパイル)

という処理をベースとして、

FORTH として最も初期の記述は(Moore74)"FORTH: A NEW WAY TO PROGRAM A MINI-COMPUTER"にある。RAM 8k バイト、ハードディスクと端末のつながったミニコンピュータ、フロントパネルのスイッチを操作して、数 10 バイトのブートローダを 1 バイト 1 バイト手打ちで入力して、紙テープからコンパイラを読み込ませ実行する環境。おそらく、FORTH 実行環境は紙テープに打ち込んでおき、手打ちブートローダから読み込んで RUN していたのでしょう。

FORTH 実行環境は 2k バイトだとのこと。現在の我々の住まいの 1000 分の 1、ひょっとすると 100 万分の一かもしれない。アプリケーションも数キロバイト、データ領域も数キロバイトの規模感である。現在を生きる我々がこの規模感の世界で何か役に立つ・実用的なものを開発すること自体に意味がないことは、私自身も承知している。だが、Moore はこの環境で天文台向けアプリケーション(おそらく、望遠鏡を天空の一点に向けながら写真を撮像するようなものなのだろう)を開発し、当時の天文台の人気を博したというのだ。小さな部品の組み合わせ、言語ではあるが、既にある部品とアプリケーション用の部品を追加で開発し組み合わせる。2kB のメモリでも動作する「インタプリタ」と「コンパイラ」なら、我々の世代なら VTL や Tiny BASIC を思い浮かべるだろう。だが、FORTH には「ソース読み込み・解釈部」自体も幾つかの部品の組み合わせでできており、ここをカスタマイズすることもできる。例えば、単語の文字並びの定義も変えることができる。あるアプリケーションの機能を単語として定義してそろえておくだけでなく、実際にアプリケーションを実行する際の「入力」の構文も専用化することができる。この辺りは他の言語・実行環境ではなかなかできるものではない。

Moore は「ジョン・マッカーシから Lisp を学んだ」ことがあると言う。1960 年に学士取得後にスタンフォードで 2 年間数学を学んだ時のことらしい。Lisp のインタラクティブ性、(S 式の範囲ではあるが)パーザも自作できる環境についても知っていたと推測する。

IBM704 で Fortran II を使用(1958 年)。このプログラムをアセンブラに圧縮し、衛星軌道を決定(1959 年)。一方、スタンフォード線形加速器センターで電子ビームのステアリングを最適化するためにバロウズ B5500 用の Algol を習得(1962 年)。Charles H Moore and Associates として、タイムシェアリングサービスをサポートするために Fortran-Algol トランスレータを作成しました(1964 年)。また、最初のミニコンピュータでリアルタイムのガスクロマトグラフをプログラミングした(1965 年)。モハスコ社でオーダーエントリネットワークをプログラムするために Cobol を習得した(1968)。[PROGRAMMING A PROBLEM ORIENTED LANGUAGE, Moore70b]

FORTH を発明するまでにこれだけの経験を積んでいた。Fortran/Algol 言語に熟知し、最終的にアセンブリ言語で書くことを躊躇しない(というか、アセンブリ言語で書かざるをえなかった)し、実際にアセンブリ言語で書いて仕事できたのだろう。

FORTH の誕生

[Evolution-FORTH]には Moore が FORTH を誕生させた経緯について説明がある。

1960 年に、スミソニアン天文台で天文関係の計算を行うプログラムを開発していた。彼自身が開発・蓄積したプログラムの「構成管理」のためのインタプリタを自分で開発したというのが、のちの Forth につながっているらしい。

「カードトレイ 2 枚を埋め尽くす」規模に成長した彼のプログラムの再コンパイルの手間と時間を最小にするために、プログラムを制御するカードを読み取る簡単なインタプリタを開発した。これにより、彼は再コンパイルすることなく、複数の衛星のために異なる方程式を構成することができた。

このインタプリタには、現代の Forth に受け継がれているいくつかのコマンドとコンセプトがあり、主にスペースで区切られた「単語」を読むコマンドと、数字を外部形式から内部形式に変換するコマンド、それに IF ... ELSE 構成がある。

彼は、自由形式の入力が、特定の列にフォーマットするという、より一般的な Fortran のやり方、これは列間違いにより果てなく再実行を繰り返す結果になるやり方よりも効率的で(より小さく、より速いコード)、信頼できるものであることを発見しました。

FORTH の発展

スタンフォード時代に、スタンフォード線形加速器センターでプログラムを書き、2 マイル電子加速器のビームステアリングを最適化したという。

そのプログラムを制御するために、彼は自分のインタプリタを拡張し、パラメータ渡しのためのプッシュダウンスタック、明示的に値を取得・保存できる変数、算術・比較演算子、手続きを定義・解釈する機能などを管理できるようにしたものを使用した。

1965 年からはフリーのプログラム書きとして様々な仕事をこなした。都度、インタプリタを活用していた。

60 年代終盤にはミニコンピュータが登場し、それに伴ってテレタイプ端末が登場したが、ムーアはこの端末用に演算子を追加して文字の入出力を管理するようになった。

1968 年、モハスコインダストリーズ社に入社し、そこで、IBM1130 ミニコンピュータ+2250 グラフィックディスプレイ用のコンピュータグラフィックスプログラムを開発した。

2250 用のコードを生成するために、プログラムにクロスアセンブラを追加し、さらにプリミティブエディタとソース管理ツールも追加しました。このシステムは、IBM のソフトウェアが静的な 2D 画像しか描けなかった時代に、アニメーションの 3D 画像を描くことができた。また、遊びで、初期のビデオゲーム「Spacewar」を書いたり、「Algol Chess」プログラムを、初めて「FORTH」と呼ばれる新しい言語に変換したりもした。彼は、FORTH が非常にシンプルになったことに感動した。

このころ、彼が育ててきたインタプリタ(もはや、実行環境といってもいいだろう)は Forth という名前を持つことになった。

ムーア氏は、2250 をプログラミングするための Forth ベースの 1130 環境が、1130 のソフトウェアが開発された Fortran 環境よりも優れていることに気づき、1130 コンパイラに拡張した。その結果、ループコマンド、ソースを 1024 バイトのブロックに分けて管理する概念とその管理ツールなど、現在 Forth で認識されているコンパイラの機能のほとんどが追加された。

最も重要なのは、辞書が追加されたことだ。手続きは名前を持つようになり、インタプリタは名前のリンクリストを検索して一致するものを探しました。これはスタンフォード大学のコンパイラから学んだことで、1980年代までForthで広く使われていた。

1130システム上で、辞書・コンパイラが確立してゆく。また、リターンスタックも入った。

最後に、ルーチンをネストするための簡単なメカニズムを提供するために、「リターンスタック」と呼ばれる第2のスタックが追加された。リターンアドレス用に予約されたスタックを持つことの利点は、呼び出しの前後に「バランス」を取る必要がなく、もう1つのスタックをパラメータ受け渡しに自由に使えることであった。

モハスコ社でのプロジェクトと中断、この時代に「タスクを定義し管理する仕組み」「現在使われている仕組みに似たディスクブロックバッファ」が導入されている。

Forthに関する最初の論文はモハスコ社で書かれた [Moore,1970a]。1970年、モハスコ社は、新しい Univac 1108 で受注システムのための専用線ネットワークを処理するという野心的なプロジェクトを Moore 氏に任せた。彼は、Forth を 1108 に移植し、トランザクション処理を行う COBOL モジュールとのインタフェースを確保した。1108 の Forth は、アセンブラでコード化されていた。入出力のメッセージをバッファリングし、各行を処理するタスク間で CPU を共有する。また、入力を解釈し、適切な COBOL モジュールを実行した。このバージョンの Forth には、タスクを定義し管理する仕組みや、現在使われている仕組みに似たディスクブロックバッファを効率的に管理する仕組みが追加されていた。

このころ、[PROGRAMMING A PROBLEM ORIENTED LANGUAGE, Moore, 70b]を書いた。

しかし、不況のあおりを受けて、モハスコ社は 1108 プロジェクトの完成を待たずに中止を決定した。Moore は直ちにその旨を伝え、怒りの詩を書き、出版されることのなかった Forth に関する本 [Moore, 1970b]を書きました。それは、Forth ソフトウェアの開発方法を説明し、シンプルさと革新性を奨励するものでした。

私がこの本を書こうと思ったきっかけである。1970年、万国博覧会が万博と呼ばれていたころ、アメリカ東海岸のできごとである。C言語もない(C言語誕生は1972年である)、コンピュータ付属のコンパイラ・ライブラリが玉石混交だった(ドラゴンブック第1版の発売が1986年であった。その前10年程度は研究テーマとなるくらいで、市井のプログラム書きがコンパイラ理論など知っているはずがない、そんな時代につくられたツールの品質たるやどんなものなのだったろうか)ころ、「人が作ったソフトウェアは信用できない、自分の手で自分のアプリケーションに必要な機能を構築するのだ」という意思に基づいて生まれたソフトウェア構築環境、それがForthだった。

2. Forth の哲学

Moore78の最初に書かれている基本原則の系「Do It Yourself!」には「標準的なライブラリを使うべきという通例に従わずに、自分でサブルーチンを書きなさい」とある。Evolution-Forthには説明があり、

自分でサブルーチンを書く前に、書き方を知らなければならない。これは、現実的には、以前に書いたことがあるということであり、そのため、始めるのが難しいのです。しかし、一度試してみてください。同じサブルーチンを何台ものコンピュータと言語で何十回も書いているうちに、かなり上手になるはずだ

ということなのだ。アセンブリ言語

2 章 実行環境の整備

70 年当時のミニコン環境を整えます。機械語/アセンブラでの開発がメインで、私個人として経験のない 68000 を選びました。

今風の開発環境として、SBC(Single Board Computer)とエミュレータソフトウェアを用意しました。SBC は、68008 CPU, 128k SRAM, PIC18F CPU からなる基板です。エミュレータは [Musashi](#) をベースにしました。68000 ファミリーを FPU/MMU まで備えたなかなかのエミュレータだと思います。

70 年代の実行環境・開発環境

Moore がたどった道をたどるために、70 年当時のミニコン環境に近づけます。CPU とメモリとスイッチパネル、数 MByte のハードディスクと、タイプライタ端末、紙テープリーダー・パンチャです。

ミニコン自体は 19 インチラック 1 本に CPU/メモリとフロントパネルが刺さっています。磁気テープ装置やハードディスクも含めてラック数本でコンピュータ 1 台を構成していたようです。オペレータの一人は、**タイプライタ端末**の前に座ってコンピュータを操作するふりをしています。

図 2-1. [PDP-11 ミニコンピュータ全景](#)(PDP-11 の前でポーズを取る Dennis Ritchie と Ken Thompson)



フロントパネル: コンピュータを操作するためのスイッチが並んでいます。CPU を止めておくスイッチ、CPU 停止中に RAM 上に 1 バイト 1 バイトデータを書き込んでゆくためのスイッチがならんでいます。

図 2-2. [ミニコン PDP-11 のフロントパネル](#)(実際には、レプリカ PiDP-11 のパネルです)



当時のミニコンピュータは、電源投入直後にメモリ上にはなににもなく、フロントパネルのスイッチを操作して、0番地から1バイトずつIPLを書き込み、それを実行して実行したいプログラムを紙テープから読み込むというのが使い始めの手続きとなっていました。

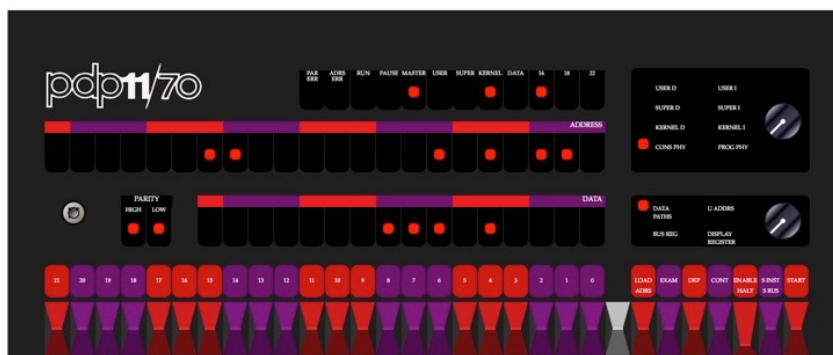
具体的にどんな感じで操作するのかは、以下のYouTubeムービーがよい感じを出しています。

[Programming a PDP-11 Through the Front Panel by BitDanz Blog](#)

ムービーを今すぐ見られない方々のために、いかにざっと説明を書いておきます。

- 機械語(8進数データ)をプリントアウトしておく。
- 先頭アドレス(01000)を下側のスイッチをON/OFFして2進数で設定する。
下側のスイッチ(スイッチレジスタ)は数値(アドレス、データ)を入力するためのもので、3個一組で色分けされており、8進数3ビットまよりのダンプデータの入力が分かりやすくなっています。
- LOAD ADRSスイッチを引き、スイッチの値をアドレスレジスタに設定する。
- スイッチレジスタのスイッチを操作し、データを設定する。
- DEPスイッチを引き、先ほど設定したデータの値をメモリに書き込む。
- (アドレスレジスタの値が自動的にインクリメントされるので)再度スイッチレジスタに次のデータを設定し、DEPスイッチを引くと、2番目のデータが格納される。

こうして、1ワード1ワードメモリに打ち込んでゆくのです。



左下のスイッチレジスタ(18個)で8進数を入力します。



右下のコントロールスイッチを使い、アドレス設定、データ書き込みを行います。



各スイッチの説明は以下の通りです。

スイッチ名	説明
LOAD ADRS	スイッチレジスタの内容をバスアドレスレジスタに書き込む。書き込んだ結果がアドレスレジスタ(18bit LED)に表示される。この値を使い、EXAM, DEP, START スイッチを押しそれぞれの動作を行う。
EXAM (Examine)	バスアドレスがさすメモリの内容をデータレジスタ(16bit LED)に表示する。再度このスイッチを引くと、次のアドレスの内容を表示する。つまり、バスアドレスが自動的にインクリメントされる。
DEP (Deposit)	スイッチレジスタの内容をメモリに書き込む。書き込む位置はバスアドレスである。
CONT (Continue)	先ほど停止した続きで実行再開する。
ENABLE/ HALT	ENABLE: CPU にプログラム実行させる。 HALT: CPU を停止する。スイッチを引くとシングルステップ実行する。
START	システムリセット後、プログラムを実行する。

これで数キロバイト打ち込むのは耐えられないので、「紙テープからプログラムをロードするプログラム」「ハードディスク先頭からプログラムをメモリにロードするプログラム」(IPL)だけを打ち込み、Fortran コンパイラやアセンブラをロードして実行させていたと思われます。

また、ブートローダを ROM で持たせることもできたようで、80 年代の PDP-11 のマニュアルを見ると、テンキー(実際には 8 個だけだが)でアドレス・データが入力できるコンソールと、それもなく実行開始・停止だけがでるコンソールも出てきています。後者はブートローダを ROM で持つことが前提になっていることがわかります。

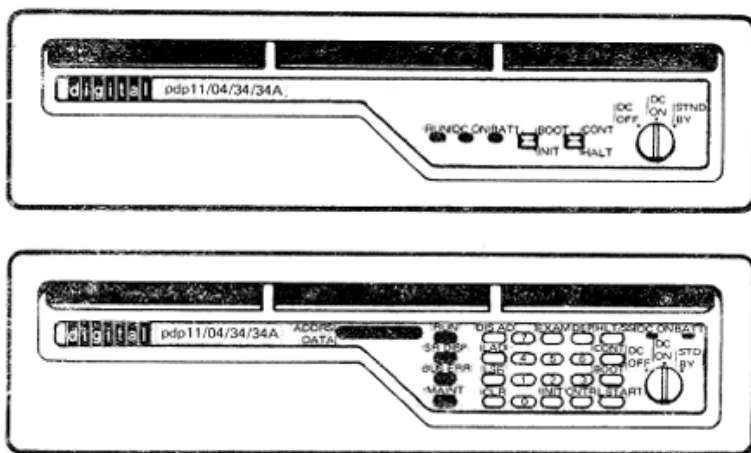


図 2-5. 80 年代の PDP-11 のフロントパネル(上が通常型パネル、下がプログラマー用パネル)

今回の実行環境

これだけ盛り上げておいて恐縮ですが、今回の実行環境ではスイッチパチパチのパネルは用意していません。大昔に自作コンピュータを操作したときの経験を思い出すと、数十バイトの IPL ロードでさえ「やってられねえ」感しかないからです。申し訳ありません。

SBC 版では、起動直後にブートロードモードに入り、シリアルターミナル(Teraterm 等)のテキストアップロード機能を用いてダンプデータファイルをアップロードします。

SBC 内部の PIC のファームウェアがダンプデータファイルをメモリ上に展開し、スタートコマンドを入力するとプログラムを実行します。

エミュレータ版は、Linux コマンドラインプロンプトでダンプデータファイルを引数として与えると、エミュレータ起動時に読み取りメモリ上に展開し、プログラムの実行を開始します。

アセンブル・リンクはいずれにおいても Linux プロンプトから、Linux コマンドのアセンブラ・リンクを起動します。70 年代の環境では、

- ブートローダを手で打ち込む。
- 紙テープのアセンブラプログラムを読み込み実行する。
- アセンブラソースコードを紙テープから読み込み、メモリ上に展開する。
- ターゲットプログラムを実行する。

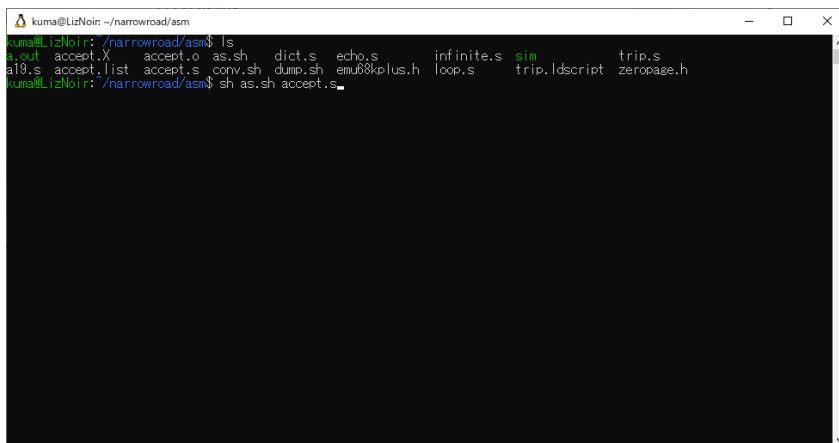
と思われます。かなり面倒なことが想像できます。そういう観点からは今回の「Linux コマンドでアセンブルする」はかなりのズルかも知れません。

一方、実行開始後は打つ手はありません。デバuggなしでのデバuggとなります。ブレークポイントとシングルステップまで許容してもいいかもしれません。

具体的な手順

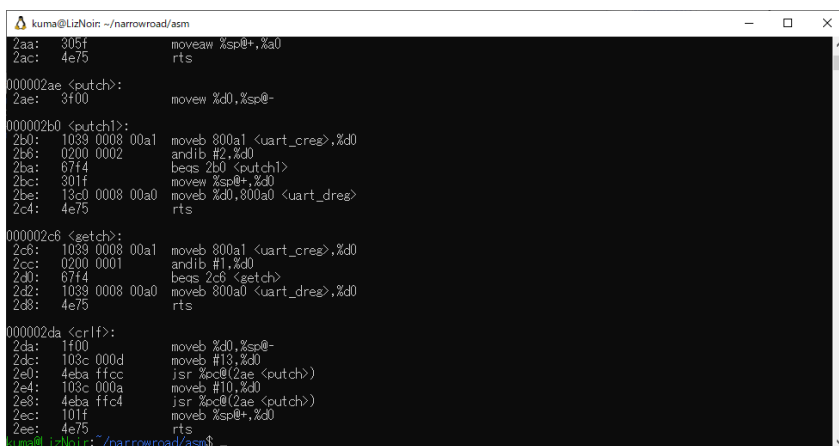
1. ソースコードをアセンブルする。

`sh as.sh [source-file]`を実行します。以下ではソースコード `accept.s` をアセンブルしています。



```
kuma@LizNoir: ~/narrowroad/asm
kuma@LizNoir: ~/narrowroad/asm$ ls
a.out  accept.X  accept.o  as.sh  dict.s  echo.s  infinite.s  sim  trip.s
a19.s  accept.list  accept.s  conv.sh  dump.sh  emu68kplus.h  loop.s  trip.ldscript  zeropage.h
kuma@LizNoir: ~/narrowroad/asm$ sh as.sh accept.s
```

アセンブルが完了すると画面にリストファイルが出力されます。



```
kuma@LizNoir: ~/narrowroad/asm
2aa: 305f      moveaw %sp@+,%a0
2ac: 4e75      rts
000002ae <putch>:
2ae: 3f00      movew %d0,%sp@-
000002b0 <putch1>:
2b0: 1039 0008 00a1  moveb 800a1 <uart_creg>,%d0
2b6: 0200 0002  andib #2,%d0
2ba: 67f4      beq 2b0 <putch1>
2bc: 301f      movew %sp@+,%d0
2be: 13c0 0008 00a0  moveb %d0,800a0 <uart_dreg>
2c4: 4e75      rts
000002c8 <getch>:
2c8: 1039 0008 00a1  moveb 800a1 <uart_creg>,%d0
2cc: 0200 0001  andib #1,%d0
2d0: 67f4      beq 2c8 <getch>
2d2: 1039 0008 00a0  moveb 800a0 <uart_dreg>,%d0
2d8: 4e75      rts
000002da <crlf>:
2da: 1f00      moveb %d0,%sp@-
2dc: 103c 000d  moveb #10,%d0
2e0: 4eba ffc0  jsr %pc@(2ae <putch>)
2e4: 103c 000a  moveb #10,%d0
2e8: 4eba ffc4  jsr %pc@(2ae <putch>)
2ec: 101f      movew %sp@+,%d0
2ee: 4e75      rts
kuma@LizNoir: ~/narrowroad/asm$
```

2. ダンプファイルに変換する。

`sh dump.sh > [dump-file]`を実行します。以下ではダンプファイル `accept.X` を生成しています。

```

kuma@LizNoir: ~/narrowroad/asm
2a0: 305f      moveaw %sp@+,%a0
2a2: 4e75      rts

000002ae <putch>:
2ae: 3f00      moveb %d0,%sp@-

000002b0 <putch1>:
2b0: 1039 0008 00a1  moveb 800a1 <uart_creg>,%d0
2b2: 0200 0002      andib #2,%d0
2ba: 67f4      beqsb 2b0 <putch1>
2bc: 301f      moveaw %sp@+,%d0
2be: 13c0 0008 00a0  moveb %d0,800a0 <uart_dreg>
2c4: 4e75      rts

000002c6 <getch>:
2c6: 1039 0008 00a1  moveb 800a1 <uart_creg>,%d0
2cc: 0200 0001      andib #1,%d0
2d0: 67f4      beqsb 2c6 <getch>
2d2: 1039 0008 00a0  moveb 800a0 <uart_dreg>,%d0
2d8: 4e75      rts

000002da <crlf>:
2da: 1f00      moveb %d0,%sp@-
2dc: 103c 000d      moveb #13,%d0
2e0: 4eba ffc0      jsr %pc@(2ae <putch>)
2e4: 103c 000a      moveb #10,%d0
2e8: 4eba ffc4      jsr %pc@(2ae <putch>)
2ec: 101f      moveb %sp@+,%d0
2ee: 4e75      rts
kuma@LizNoir:~/narrowroad/asm$ sh dump.sh > accept.X_

```

アセンブラとリンカ

68000 用として、binutils を 68000 用にビルドして使っています。

スクリプト `as.sh` では、アセンブラ `m68k-elf-as` を実行後、リンカ `m68k-elf-ld` で未解決シンボルの解決を行っています。

スクリプト `dump.sh` では `m68k-elf-objdump` コマンドを使ってダンプファイルを生成後、`sed` により PIC ファームウェアがロード可能な形式に変換しています。

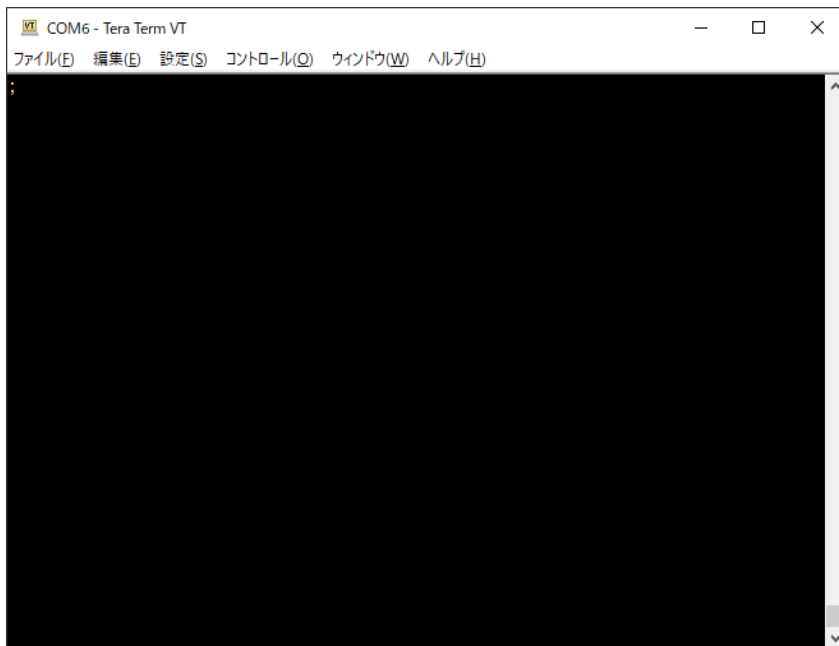
```

kuma@LizNoir: /narrowroad/asm$ cat as.sh
#!/bin/sh
b=`basename -s .s "$1"`
m68k-elf-as -o ${b}.o "$1" &&
m68k-elf-ld -T trip.ldscript ${b}.o &&
( m68k-elf-objdump -D a.out | tee ${b}.list )
kuma@LizNoir: /narrowroad/asm$ cat dump.sh
#!/bin/sh
m68k-elf-objdump -D a.out |sed -n '
/^[^*]*[0-9A-Fa-f][0-9A-Fa-f]*:/[
s/^\([^\ ]*\)[^*]*$/\1 \2/
s/^\([^\ ]*\)[^*]*$/\1 \2/
s/^\([^\ ]*\)[^*]*$/\1 \2/
s/^\([^\ ]*\)[^*]*$/\1 \2/
p

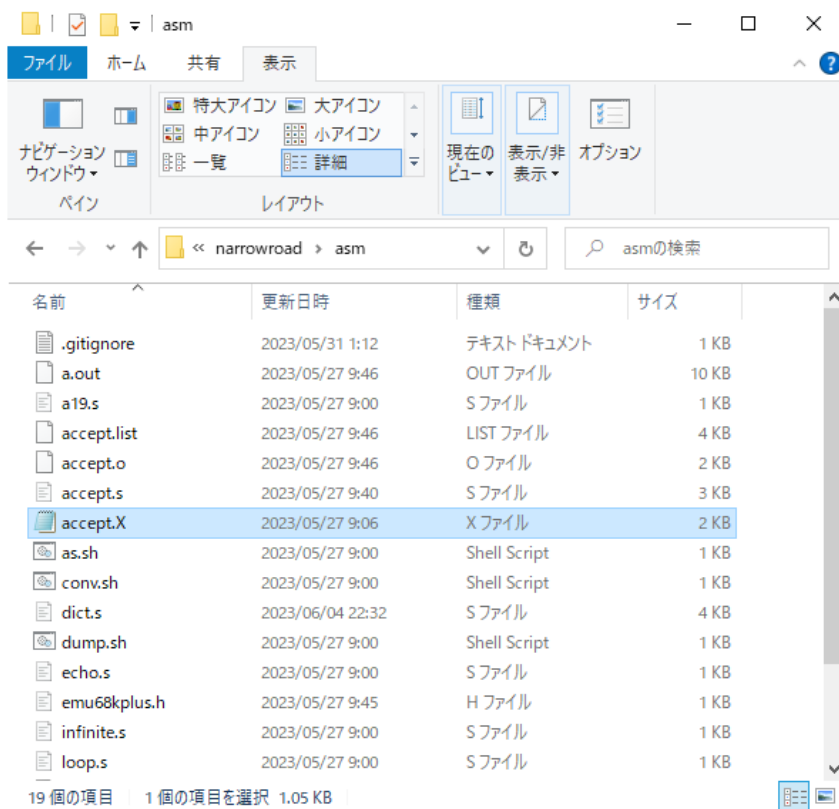
```

3. 実行(SBC 版): ダンプファイルをドラッグアンドドロップする

SBC(emu68k_plus)を起動する。Teraterm に ; だけのプロンプトが表示される。



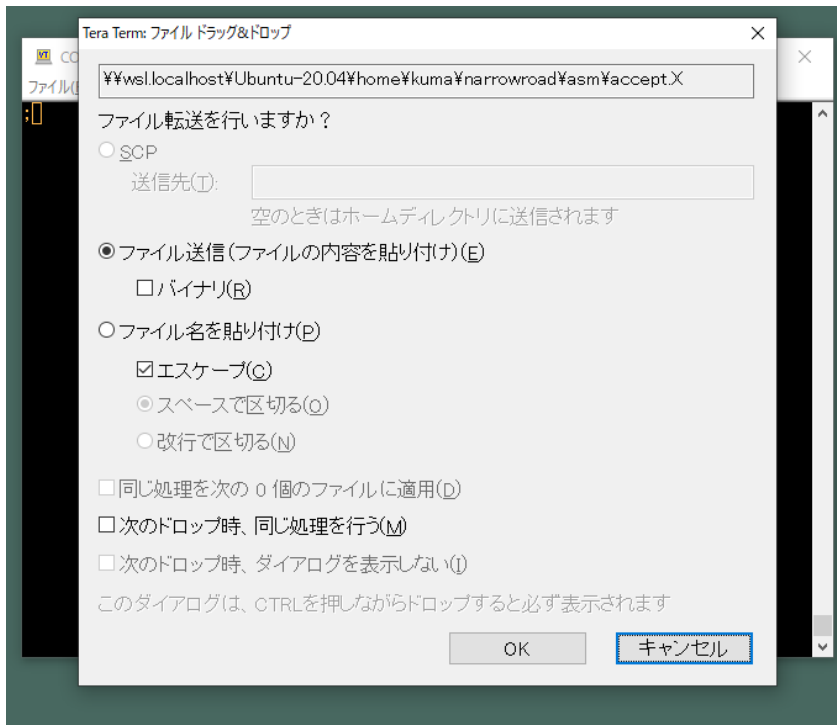
この状態で、Windows のエクスプローラからダンプファイル **accept.X** を掴んで、Teraterm にドラッグアンドドロップする。



WSL のファイルを Windows エクスプローラで開くには、`\\wsl$\\Ubuntu-20.04\\home\\kuma\\...\\accept.X` のように、`\\wsl$\\Ubuntu-20.04\\` を先頭に付けてエクスプローラを開けるとよいようである。

すると、以下のダイアログを出してくるので、OK を押す。アップロードが始まる。

8k BASIC でも数秒で終わる。さすがに 115200bps は速い。



アップロードが終わっても Teraterm 画面には何も変化がない。アップロード中は読み込みと RAM への展開に全力を注ぎ、エコーバックは一切行わないのだ。エコーバックを行わないことにより、PIC ファームウェアは 115200bps で突っ込まれても十分処理できる。

以下の画面はダンプコマンドを叩いたあとの状態です。ダンプコマンドは 1 文字 **!** (びっくりマーク) です。

```
COM6 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(C) ウィンドウ(W) ヘルプ(H)

;
0000 0000 F000 0000 0200 5441 4575 1545 5445
0010 7361 6D70 6C65 2073 7472 696E 6701 0203
0020 0000 5544 4D55 4545 1455 5455 5155 4501
0030 4551 4555 4555 4545 4451 04C5 4555 55C5
0040 5551 5444 4154 4045 4545 5544 0545 5545
0050 C555 5D51 4551 1545 4545 5545 5557 0551
0060 4141 4545 5C54 4545 4414 5504 55D5 5455
0070 5145 4465 6045 5515 4540 5455 D445 4144
0080 5555 7504 4456 5455 5544 5555 44D5 5554
0090 545D 5555 D444 5D55 5556 4541 4445 5554
00A0 45FF D544 5555 5544 5555 545D 5545 5554
00B0 5545 4155 4445 45C4 5044 0545 4545 4540
00C0 5554 5D45 5545 5455 5454 4504 5555 4754
00D0 5545 44D4 44D5 4544 5544 D055 5444 C444
00E0 5404 4555 144D 1445 4445 5445 3545 4445
00F0 5054 5555 5554 5545 5544 5755 4544 5554
0100 0455 5554 1455 45C4 5555 5555 5455 4545
0110 4544 4D74 4554 5655 5545 5144 4455 5455
0120 4545 5544 41C4 5444 1575 7545 C055 4555
0130 5554 5554 5545 4554 5445 4541 4505 5D50
0140 144D 5445 5515 0455 5445 4445 5551 554C
0150 5154 4344 5555 5545 4151 4445 5154 1555
0160 5444 4145 4575 5405 0554 5445 4515 4545
0170 4454 455D 5455 4504 5444 144D 4404 5455
```

プログラム実行開始は...(ピリオド3個連打)です。start ss = 0, bp = 0000と表示され68008のリセットが解除されます。上記の表示は見た目だけでブレークポイントが使えるわけではありません。

```
COM6 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(C) ウィンドウ(W) ヘルプ(H)

0180 4565 4545 5155 4556 4554 5514 4454 5044
0190 5445 5754 55E5 5455 5544 D445 5654 5446
01A0 6551 5544 44D5 4555 5444 5441 4555 5554
01B0 4455 5455 5555 4444 0445 5544 4544 D555
01C0 5555 7445 5455 5145 5454 5544 5544 5455
01D0 4441 5555 4545 D515 C545 7575 4447 5755
01E0 C745 4545 5445 5545 5444 5554 7545 4405
01F0 5555 4554 5545 4545 4554 5555 5454 5554
0200 207C 0000 0F00 7240 4EBA 0010 4EBA 00CC
0210 4EBA 007C 4EBA 00C4 60E6 3F09 3F02 3248
0220 3401 323C 0000 B441 6F00 005A 4EBA 0098
0230 0C00 000D 6700 004E 0C00 000A 6700 0046
0240 0C00 0008 6700 001A 0C00 007F 6700 0012
0250 3F00 4EBA 005A 301F 1380 1800 5241 60C6
0260 0C41 0000 6F00 FFC0 5341 103C 0008 4EBA
0270 003E 103C 0020 4EBA 0036 103C 0008 4EBA
0280 002E 60A2 3001 341F 325F 4E75 60FE 3F08
0290 3F01 3F00 3200 0641 FFFF 6D00 000A 1018
02A0 4EBA 000C 60F0 301F 321F 305F 4E75 3F00
02B0 1039 0008 00A1 0200 0002 67F4 301F 13C0
02C0 0008 00A0 4E75 1039 0008 00A1 0200 0001
02D0 67F4 1039 0008 00A0 4E75 1F00 103C 000D
02E0 4EBA FFCC 103C 000A 4EBA FFC4 101F 4E75
start ss = 0, bp = 0000
```

今回の例で用いたaccept.Xは行入力ルーチンです。通常文字を受信・エコーバックするとともにバッファに貯めます。^Hにより直前の文字を抹消することができます。行編集コマンドはこれだけです。リターンを押すと入力完了です。このテストプログラムでは入力して得た文字列をそのままダンプしています。よって以下のように同じ表示が2行続くことになります。


```
COM6 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(C) ウィンドウ(W) ヘルプ(H)

01A0 6551 5544 44D5 4555 5444 5441 4555 5554
01B0 4455 5455 5555 4444 0445 5544 4544 D555
01C0 5555 7445 5455 5145 5454 5544 5544 5455
01D0 4441 5555 4545 D515 C545 7575 4447 5755
01E0 C745 4545 5445 5545 5444 5554 7545 4405
01F0 5555 4554 5545 4545 4554 5555 5454 5554
0200 207C 0000 0F00 7240 4EBA 0010 4EBA 00CC
0210 4EBA 007C 4EBA 00C4 60E6 3F09 3F02 3248
0220 3401 323C 0000 B441 6F00 005A 4EBA 0098
0230 0C00 000D 6700 004E 0C00 000A 6700 0046
0240 0C00 0008 6700 001A 0C00 007F 6700 0012
0250 3F00 4EBA 005A 301F 1380 1800 5241 60C6
0260 0C41 0000 6F00 FFC0 5341 103C 0008 4EBA
0270 003E 103C 0020 4EBA 0036 103C 0008 4EBA
0280 002E 60A2 3001 341F 325F 4E75 60FE 3F08
0290 3F01 3F00 3200 0641 FFFF 6D00 000A 1018
02A0 4EBA 000C 60F0 301F 321F 305F 4E75 3F00
02B0 1039 0008 00A1 0200 0002 67F4 301F 13C0
02C0 0008 00A0 4E75 1039 0008 00A1 0200 0001
02D0 67F4 1039 0008 00A0 4E75 1F00 103C 000D
02E0 4EBA FFCC 103C 000A 4EBA FFC4 101F 4E75
start ss = 0, bp = 0000
line input
line input
```

3 章 旅立ち

PROGRAMMING A PROBLEM ORIENTED LANGUAGE, Moore, 70b は、Moore 師匠が 1970 年に書いた文書である。自前の環境を 10 年使い続けて、開発環境とアプリケーション構築に関する彼の考えを説明している。この文書は FORTH という開発環境そのものの説明だが、この 9 章は、新しいマシンを前にしてこの環境をどのようにして構築してゆくか、のあらすじをざっと説明している。

移植ではない、構築なのだ。別のアーキテクチャの CPU をまたいで生きてゆかねばならないのだから、共通のディスクアクセス機構、共通の OS、共通のライブラリを前提として、元のソースコードにちょいちょいと手直しだけで「移植」することは期待できない。C コンパイラなんてもちろん存在しない。下手するとアセンブラさえ使えない環境、機械語をばちばち打ち込んで実行するだけの環境に、自分の手元にある環境一式を「構築」してゆく。これまで頼りにしていたものすべてから切り離され、生きてゆかねばならない、そんな気持ちができるかも。「月は地獄だ」「Dr. STONE」「サバイバル」そんな作品を思い出しながら、彼のたどった道を自分の足で歩いてみたい。そんな気がするのだ。

以下では、Moore, 70b の 9 章の文言を引用しながら、私が歩いた道筋を説明してゆく。よろしければ、ご自分で好きな CPU を選んで実行環境をつくっていただいても構わない。

まず練習だ

さて、あなたは今、コンピュータと向き合っている。どうするんだ？ まず練習です。コンピュータを起動したら、無限ループを実行するように割り込み位置を初期化しなさい。いいかい？ それからループを修正してメモリをクリアするようにします。いいですか？ おそらく多くのことを学んだことでしょう。(9.1 章)

BIOS はおろか、ブートローダもなにもない。1 バイト目からフロントパネルのスイッチをばちばちして入力して RUN スwitchを押すしかない、そんなコンピュータを前にして、最初にやることは、コンピュータに慣れること。

Moore 師匠当時のコンピュータはフロントパネルにアドレスバス・データバス状態を示す LED があり、その点滅やシングルステップで実行を見ることができたのでした。

実は、私も 1981 年に自分で「スイッチばちばち」のコンピュータをくみ上げて使い始めていました。が、これが結構つらいのです。結局 PC8801 を買ってからはそちらにひよってしまっていたのでした。なので、今回もスイッチばちばちから始めるというのはやめておきます。代わりにシングルステップを使うことにします。

EMU68kplus には割り込み機能はありません(今のところは)。よって、単純な無限ループを組みます。

```
.org      0
loop:
```

```
bra      loop
/* end */
```

アップロード用ファイルは、

```
=0 6000 fffe
```

辞書(ディクショナリ)

さて、これからが本番です。まだ使うことはできないにしても、辞書の構築を開始します。ここでエントリの形式を選びます。可変長エントリは必須ですが、それでも、ワードサイズとレイアウトはあなたが決めることができます。

既にあるライブラリを頼まず、一から手組みで「オレのシステム」をくみ上げてゆくこの取り組みで、データ・ルーチンに名前を付け呼び出せるようにする仕組みの基本が「辞書(ディクショナリ)」です。ロジカルには、「名前とバイト列のペアをエントリとする」「定義した順にエントリをリストにつなぎ、検索は最新のエントリから最初のエントリにさかのぼってゆく」がその定義で、登録・検索も単純なルーチンで実現できます。

エントリには、定義されるワード、実行されるコード、次のエントリへのリンク、パラメータの4つのフィールドがあります。それぞれについて説明する。

ワードの形式はワード入力ルーチンとともに決定されなければならない。ワードのサイズは固定で、NEXTで定義されたサイズより小さくてもよいが、ハードウェアのワードサイズの倍数でなければならない。しかし、より洗練されたアプリケーションは、出力メッセージを構築するために辞書のワードを使用する。その場合、ワードを切り詰めないことが重要であり、その場合、ワードフィールドは可変長でなければならない。このフィールドのサイズをマークするために、文字カウントではなく、空白文字を使用する必要がある。可変エントリ内で可変ワードフィールドを扱うには、ワードは一方に(後方に)、パラメータは他方向に(前方に)伸びる必要があります。固定または可変ワードサイズのどちらを選ぶかは、基本原則の適用が必要です。

コードフィールドには、テーブルや他の省略形へのインデックスではなく、ルーチンのアドレスを入れるべきです。プログラムの効率性は、3.9で説明するように、エントリが特定された後、コードにたどり着くまでの時間に強く依存します。しかし、プログラムのサイズが小さいと、このアドレスはハードウェアアドレスフィールドより少ないスペースに収めることができます。

リンクフィールドも同様に、ハードウェアで指定されたものより小さくてもよい。これは、現在のエントリからの距離ではなく、次のエントリの絶対位置を含むべきである。

パラメータフィールドは、通常4種類の情報を含む。

- 定数または変数で、サイズは可変です。数値の性質は、実行されるコードによって決定されます。

- 配列 - 数値が格納されるスペースです。配列のサイズはパラメータであるか、または実行されるコードに含まれているかもしれません。
- 定義：仮想のコンピュータ命令を表す辞書の項目の配列。3.9 を参照してください。
- 機械語命令：プログラムによってコンパイルされたコードで、このエントリが実行される際にこのコードが実行される。このようなデータは、おそらくワード境界でアラインされなければならないが、他はその必要がない。(3.6.1. エントリの形式)

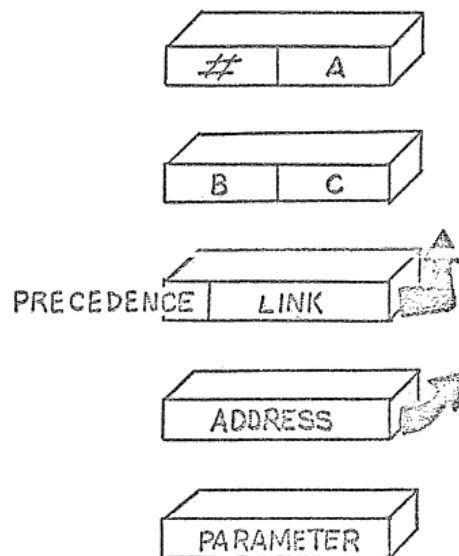


図 4. 辞書エントリの形式

例えば、単語"ABC"のエントリは以下のようになります。

```

1324: 03          ; number of name
1325: 41          ; 1st char of name
1326: 42          ; 2nd char of name
1327: 43          ; 3rd char of name
1328: 0340        ; PRECEDENCE and LINK
132A: 0551        ; machine code of the word, "ABC"
132B:

```

最初の 1 バイトは名前の文字列の長さです。Forth では、文字列はヌル文字終端ではなくて、銭湯 1 バイトに長さを置き、2 バイト目から名前を表す文字を順に並べます。

次の要素は PRECEDENCE and LINK です。この要素はワードアライメントされて置かれます。この例では 2 バイトアライメントです。

PRECEDENCE は、このワードの属性を表すビットです。通常、2 ビットを占めます。LINK は次のワードの先頭を指すポインタです。この例では、1 ワード(16bit)の上位 2 ビットを PRECEDENCE が占め、下位 14 ビットをアドレスを収容します。

LINKは、ワードの単方向リストを順にたどる際に用います。絶対アドレスを用いてもよいし、リンク位置からの相対オフセットでもかまいません。絶対アドレスを用いると、辞書サイズの上限が16kBになりますが、相対オフセットを用いると、1エントリ16kB上限になります。あとで分かってくるのですが、1エントリ16kBも使うことは普通はないのと(特殊な巨大配列を使う場合ぐらい)、単方向リストをたどる処理はプログラムの実行時間に影響を与えない(「コンパイル」時間に効いてくる)ので、相対オフセットで1エントリ手繰るたびにポインタ加算が余分に増えても問題ないでしょう。

コンパイラ(といっても入力文字列からワードを一つ一つ切り出し、辞書を引いてその名前を持つエントリを検索し、得たエントリのアドレスをメモリ上に並べてゆく)ができるまでは、「手で辞書をコンパイル」します。具体的には、「名前」「属性」「名前の列」を順に並べ、辞書エントリを描くアセンブラプログラムを書きます。

通常のやり方は、マクロアセンブラのマクロ機能を用いて辞書エントリを生成し、本体は「名前の列」のエントリシンボルを書き出して、アセンブラに掛けてオブジェクトを生成し、辞書とします。

ここでは、簡単なテキスト処理プログラムにより、辞書エントリを生成し、アセンブラに掛けるコードを生成させるようにします。名前の前方参照を解決する必要があるので、そこはアセンブラの機能を利用するというわけです。

エントリのデータフィールドには、他のエントリのアドレスが並びますが、アドレスでない要素も並べられるようにします。以下の要素です。

- **定数をスタックに置く要素(Literal):** 機械語のイミディエイト命令のように、次の要素を数値とみなしスタックに置きます。ソースコードで定数を書くと、辞書エントリとしてコンパイルするときにこの要素が必要になるのです。
- **ブランチ(スキップ)命令:** 直後の要素をアドレスとみなし、そこまでジャンプすることを指示する要素です。IF-ELSE-THEN, WHILE 等の制御構造を実現するとき、辞書エントリ内で要素をスキップしたり戻ったりすることが必要になるのですが、それを実現します。無条件ブランチと、条件付きブランチの2種類を用意します。条件付きブランチは、「スタックトップの値がゼロならジャンプする」の一つだけとします。大なり、小なりは、ブランチ命令の前に演算を並べてスタックトップにゼロ/非ゼロの結果を返すことで実現します。

ここまでくると、**call, bra, load immediate**の3種類の命令だけからなる特殊なCPUが実行する機械語の世界である。そのCPUは、スタックポインタ2つ、アキュムレータ1個を持ち、データスタックとリターンスタックを管理し、スタック上のデータとアキュムレータの間で演算するという風になる。文献によっては、Forth Virtual Machine と呼ぶものもある。

SAVE, LOAD, DUMP

ここでワードの実行の話になるのかと思えば、Moore 師匠の話はいくつかのサービスルーチンをアセンブラで書く方向に向かいます。

辞書エントリの実行はどうなるねん。確かにサービスルーチンは必要だし、CPUに慣れるというなら、よく知ったサブルーチンを新しいCPUの機械語で書くというのもわかる。ということで、辞書の実行は後のお楽しみということで取っておきます。

最初のエントリはSAVEで、これはプログラムをディスクに保存します。コントロールループがないので、手動でジャンプしなければなりません、少なくとも、多くの作業をやり直すことは最小限に抑えられます。2番目の項目はLOADで、ディスクからプログラム

を再ロードします。ハードウェアのロードボタンがあるかもしれませんが、それと互換性を持ってプログラムを保存できるのであれば、それはそれで結構です。そうでなければ、ロードカードにパンチして初期ロードを提供するのがよいでしょう。しかし、コアから再スタートできるのは常に便利なことです。

SAVE, LOAD は、ハードディスクとメモリの間でデータの読み書きを行うサブルーチンです。

いやあんた、EMU68kplus には外部ストレージがないやんか。SAVE も LOAD もないやろう。ということであまり困りました。

ここで想定している操作は、

- 以前作ったプログラムを RAM 上に LOAD する。
- スイッチぱちぱちで追加入力、ルーチン呼び出し、シングルステップやレジスタウォッチを使いデバッグする。
- 作業が一区切りしたら RAM イメージをディスクに SAVE する。

だと思います。ここでは、

- 以前保存しておいたバイナリイメージを母艦 PC から RAM 上にアップロードする。
- ルーチン呼び出し、シングルステップを使いデバッグ。
- 但しプログラム修正はアセンブラソースコードを修正して再アセンブルする。
- 修正プログラムのデバッグは、再アセンブル結果を RAM 上にアップロードする。

とします。EMU68kplus 上でプログラム変更しないので、モニタの機能だけで作業を進めることができます。

外部ストレージがまだないことと、バイナリを手で修正してデバッグというのも辛いので、ひよってしまいました。すみません。

3 番目のエントリは DUMP で、これはコアをプリンタにダンプします。スイッチで見るとずっと速いので、それほど速くなくてもよいでしょう。このルーチンはおそらく自明なものではありませんが、12 命令以上かかることはないはずで、ほんの少し延期してもいいかもしれません。

せめてこれぐらいはちゃんとやろうと想い、DUMP ルーチンを作りました。

```
/*
 * dodump
 * hex dump a region of RAM storage
 * %a0: begin address
 * %d0: count
 */
dodump:
    move.w    %a1, -(%a7)    /* push %a1 */
    move.w    %d1, -(%a7)    /* push %d1 */
    move.w    %d0, %d1       /* %d1: loop counter */
    move.w    %a0, %a1       /* %a1: address pointer */
    move.w    %a0, %d0
```



```

and.w  #0xffff,%d0      /* address should be even */
move.w  %d0,%a1
and.w  #0xffff0,%d0      /* %d0: actual start address */
/* type initial address */
move.w  %d0,(dbg_port+2)
jsr     (puthex4)
move.b  #':',%d0
jsr     (putch)
jsr     (bl)              /* type a blank */
/* check skip words */
dodump1:
/* prefix five spaces */
move.b  %d0,(dbg_port)
move.w  %a1,%d0
and.w  #0xf,%d0          /* %d1 = %a1 & 0xf, skip count */
dodump2:
beq.b   dodump3          /* if zero, end of five spaces */
/* five spaces */
jsr     (bl)
jsr     (bl)
jsr     (bl)
jsr     (bl)
jsr     (bl)
sub.w   #2,%d0
bge.b   dodump2
dodump3:
/* check loop counter */
and.w   %d1,%d1          /* check loop counter */
beq.b   dodump4
/* word dump loop */
move.w  %a1,%d0
and.w   #0xf,%d0
bne.b   dodump5          /* skip typing address */
/* type address */
move.w  %a1,%d0
jsr     (puthex4)
move.b  #':',%d0
jsr     (putch)
jsr     (bl)
dodump5:
/* put word */
move.w  (%a1),%d0
jsr     (puthex4)
jsr     (bl)
add.w   #2,%a1

```

```

/* check eol */
move.w  %a1,%d0
and.w   #0xf,%d0
bne.b   dodump6      /* to tail check */
/* put crlf */
jsr     (crlf)
dodump6:
sub.w   #2,%d1
bge.b   dodump3
dodump4:
/* all dump over, closing process */
move.w  %a1,%d0
and.b   #0xf,%d0
beq.w   dodumpx
/* do crlf if address %15 != 0 */
jsr     (crlf)
dodumpx:
/* pop registers */
move.w  (%a7)+,%d1      /* pop %d1 */
move.w  (%a7)+,%a1      /* pop %a1 */
rts

```

このどこが12命令やねん。うーん。たぶん、もともと Moore 師匠が想定していたのは、アドレスもなく単に16進変換して出力するだけなんでしょうねえ。ここで私は頑張りました。結果は以下の通りです。だいたいよく見る見慣れた形式にしています。

```
COM4:115200baud - kuma@chinachu:~ VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)
;start ss = 0, bp = 0000
123480102: 00 W M1
|80102 00 W OUT: 00
80103: 80 W M1
|80103 80 W OUT: 80
0080: 80100: 3A W M1
|80100 3A W OUT: 3a
0080: 307C 0080 303C 0024 4EBA 0064 60FE 3F00
0090: 1039 0008 00A1 0200 0002 67F4 301F 13C0
00A0: 0008 00A0 4E75 1039 0008 00A1 0200 0001
00B0: 67F4 1039
```