SimH で
PDP-7 ベアメタル
プログラミング

— シミュレータ上で
アセンブリ言語によるプログラミングを体験！ —

［著］大神祐真

技術書典 14 新刊
2023 年 5 月 20 日　 ver 1.0

■免責

本書は情報の提供のみを目的としています。
本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。
また本書では、™、®、© などのマークは省略しています。

はじめに

本書を手に取っていただき、ありがとうございます！

本書では、「SimH」というシミュレータを用いて、「PDP-7」という 1960 年代のコンピュータをアセンブリ言語で直接制御するベアメタルプログラミングを行います。

本書の構成

本書は以下の 4 章構成です。

第 1 章「SimH のセットアップと初めてのプログラム」

PDP-7 と SimH の紹介と、SimH のセットアップ、そして初めてのプログラムとして「実行を停止するだけのプログラム」を作ってみます。

第 2 章「計算を行ってみる (疑似乱数生成)」

基本的な計算を行う例として線形合同法による疑似乱数生成を行ってみます。

第 3 章「テレタイプで文字の入出力 (HELLO WORLD!/エコーバック)」

テレタイプによる文字の入出力を紹介し、「HELLO WORLD!」を出力するプログラムと、入出力両方を用いてエコーバックを行うプログラムを作ってみます。

第 4 章「ベクタースキャンディスプレイとライトペン (図形描画/ヒット検出)」

ベクタースキャンディスプレイとライトペンの使い方を紹介し、ベクタースキャンディスプレイへの簡単な図形の描画とライトペンによる図形のヒット期間の検出を行います。

想定する作業環境

私が Debian を普段使用しているため、基本的に Debian 系の Linux ディストリビューションで作業を行うことを想定しています。Windows の WSL2 の Ubuntu でも、以降の章で説明する SimH のビルドといくつかのプログラムの実行が行えることは一応確認していますので、Windows でも恐らく問題ないかと思います。ただ、macOS は未検証です。

サンプルコードについて

本書の各章で実装するプログラムを SimH 独自のスクリプト形式で記述したものを以下の GitHub リポジトリで公開しています。(SimH 独自のスクリプト形式については第 1 章のコラムで説明しています。)

- `https://github.com/cupnes/pdp7-baremetal-programming-with-simh-samples`

電子版や本書の更新情報について

本書の電子版は下記のウェブページで公開しています。

- `http://yuma.ohgami.jp/`

本書の内容について訂正や更新があった場合もこちらのページに記載します。何かおかしな点があった場合等は、まずこちらのページをご覧ください。

目次

第 1 章
SimH のセットアップと初めてのプログラム

この章では、PDP-7 と SimH の紹介と、SimH のセットアップ、そして初めてのプログラムとして「実行を停止するだけのプログラム」を作ってみます。

1.1 PDP-7 と SimH について

PDP-7 は、Digital Equipment Corporation(DEC) というメーカーから 1965 年にリリースされた 18 ビットのコンピュータです。大きさが冷蔵庫程もあるのですが、当時「メインフレーム」と呼ばれていたコンピュータよりは小さいということで「ミニコンピュータ」と呼ばれていたそうです。DEC は PDP-7 以前・以後にも「PDP-<番号>」というコンピュータをいくつもリリースしています。中でも PDP-7 は「初めて UNIX が作られたコンピュータ」として歴史的なコンピュータです。

SimH は、PDP シリーズのような歴史的なコンピュータを現代の PC 上で再現するシミュレータです。SimH にはバージョン 3 系以前の「"classic" version[*1]」と、バージョン 4 系の「Open-SimH[*2]」があります。バージョン 4 系で大幅に機能追加が行われており、本書でも使用する Type 340 というディスプレイのシミュレーションはバージョン 4 系で追加されています。そのため、本書ではこの Open-SimH というバージョン 4 系の SimH を使用します。

[*1] http://simh.trailing-edge.com/
[*2] https://opensimh.org/

1.2　SimH のセットアップ

SimH は Linux ディストリビューションのパッケージリポジトリにも入っていたりしますが、大抵バージョン 3 系のものであるため、公式の GitHub リポジトリからソースコードを取得してビルドします。

♣ ソースコードを取得

バージョン 4 系 (Open-SimH) の GitHub リポジトリは以下の URL です。

- `https://github.com/open-simh/simh`

こちらから `git clone` 等でソースコードを取得してください。なお、本書では master ブランチの `09899c18` (2023 年 1 月 22 日のコミット) 時点を使用しています。

▼リスト 1.1: 操作例

```
$ git clone https://github.com/open-simh/simh.git
 ・・・ clone時の出力(省略) ・・・
$ git checkout 09899c18  ←念の為(しなくても問題ないはず)
 ・・・ git checkout時の出力(省略) ・・・
```

念の為、`09899c18` をチェックアウトしていますが、基本的に最新を使用して問題ないはずです。もし大きな仕様変更等があった場合は `09899c18` をチェックアウトするか、あるいは本書の内容を適宜読み替えてください。

♣ ビルドに必要なソフトウェアをインストール

SimH はビルドのルールが Makefile で、プログラムが C 言語で書かれています。そのため、まずはこれらが一通り入った `build-essential` という Debian パッケージをインストールすると良いです。[*3]

加えて、ビルドに必要となる Debian パッケージが `libpcap-dev` ・ `libpcre3-dev` ・ `vde2` ・ `libsdl2-dev` ・ `libsdl2-ttf-dev` ・

[*3] apt が使用できない OS については、それぞれの OS の方法で「make」と「gcc」のインストールを行ってください。

libedit-dev の 6 つですので、これらもインストールしてください。[*4][*5]

▼リスト 1.2: 操作例

```
$ sudo apt update
 ・・・ apt update時の出力(省略) ・・・
$ sudo apt install build-essential
 ・・・ apt install時の出力(省略) ・・・
$ sudo apt install libpcap-dev libpcre3-dev vde2
 ・・・ apt install時の出力(省略) ・・・
$ sudo apt install libsdl2-dev libsdl2-ttf-dev libedit-dev
 ・・・ apt install時の出力(省略) ・・・
```

♣ ビルド

ビルドは、取得したソースコードのディレクトリへ移動して make pdp7 を実行するだけです。

▼リスト 1.3: 操作例

```
$ cd simh     ←ソースコードのディレクトリへ移動
$ make pdp7   ←ビルド
 ・・・ make時の出力(省略) ・・・
$ ls BIN/pdp7
BIN/pdp7      ←生成確認
```

ビルドが完了すると、直下の BIN ディレクトリ内に pdp7 という実行ファイルが生成されています。

♣ インストール

BIN ディレクトリに生成された pdp7 を、今後パス指定無しで実行できるようにしておいてください。Linux であれば、pdp7 に実行権限を付与した上で、 BIN ディレクトリの絶対パスを環境変数 PATH へ追加するか、 PATH に既に列挙されているディレクトリへ配置すれば良いです。

[*4] ビルドに必要なパッケージについては Open-SimH の GitHub リポジトリ内の SIMH-V4-status.md の「Linux - Dependencies」に書かれています。ただ、そこには libsdl2_ttf-dev というパッケージ名が書かれていますが、Debian や Ubuntu にそのような名前のパッケージは無く、おそらく libsdl2-ttf-dev の間違いであると思われるので、本書ではこちらをインストールするようにしています。

[*5] その他の OS の場合は、Open-SimH の GitHub リポジトリ内の SIMH-V4-status.md の「Build Dependencies」以下の OS 別の説明を参照してください。

▼リスト 1.4: 操作例

```
$ chmod +x BIN/pdp7
$ echo 'export PATH=/<配置した場所への絶対パス>/simh/BIN:$PATH' >> ~/.bashrc
```

1.3　実行を停止するだけのプログラムを作ってみる

適宜使い方の紹介をしつつ、最初のプログラムとして「実行を停止するだけのプログラム」を作ってみます。

♣ SimH を起動する

SimH を起動する際は、シェル上で pdp7 コマンドを実行するだけです。

▼リスト 1.5: 操作例

```
$ pdp7

PDP-7 simulator V4.0-0 Current        git commit id: 09899c18
sim>
```

起動すると `sim>` というプロンプトが表示された状態になります。

♣ SimH コマンド: examine と deposit

基本操作として SimH のコマンドを 2 つ紹介します。

1 つ目は `examine` コマンドです。これは、指定されたレジスタ[*6]やメモリアドレスの値を表示するコマンドです。例えば、リスト 1.6 のように実行すると、「次に実行する命令のアドレス」が入っている「PC(プログラムカウンタ)」というレジスタの値を表示します。

▼リスト 1.6: PC の値を表示

```
sim> examine pc  ←大文字/小文字どちらも可
PC:     00000    ←数値は8進数
sim>
```

2 つ目は `deposit` コマンドです。これは、第 1 引数で指定されたレジスタやメモリアドレスへ第 2 引数で指定された値を設定するコマンドです。例えば、リスト 1.7

[*6] プロセッサが使用する小容量で高速な記憶領域です。プロセッサの命令は基本的にレジスタかメモリの内容に対して演算を行います。

のように実行すると、PC レジスタへ 0o100[*7]を設定します。

▼リスト 1.7: PC へ 0o100 を設定

```
sim> deposit pc 100  ←数値を指定する際も8進数
sim>
```

なお、 `examine` コマンドと `deposit` コマンドは、共にコマンドの 1 文字目 (`e` あるいは `d`) が指定されていればそれぞれのコマンドとして認識されます。例えば、PC レジスタの値を表示する際はリスト 1.8 のように実行することもできます。(以降、本書では `examine` コマンドは `e` で、 `deposit` コマンドは `d` で実行します。)

▼リスト 1.8: PC へ設定した値を確認

```
sim> e pc
PC:     00100
sim>
```

加えて、両コマンドでは「Switches」と呼ばれるオプションを指定することで、表示あるいは設定する際のデータ形式をデフォルトの 8 進数から変更できます (表 1.1)。

▼表 1.1: examine・deposit コマンドの Switches 一覧

Switches	データ形式
-a	ASCII(18 ビットワードに 1 文字)
-c	ASCII(18 ビットワードに 3 文字詰め)
-m	アセンブリ
-o / -8	8 進数 (デフォルト)
-d / -10	10 進数
-h / -16	16 進数
-2	2 進数

また、 `examine` コマンドは、メモリアドレスの範囲指定ができます。 `<開始アドレス>-<終了アドレス>` あるいは `<開始アドレス>/<長さ>` と指定することでその範囲の内容を表示します。

Switches や範囲指定の挙動についてはそれらを使用する際に改めて紹介します。それに加えて、実は `examine` と `deposit` には他にも機能があったりするのですが、本書では特に重要ではないため、もし興味があれば後述のコラムで紹介する `help` コマンド等を参照してみてください。

[*7] 「0o」は 8 進数表記を表す接頭辞です。

♣ PDP-7 命令: HLT

アセンブリ言語でのプログラミングのために、本書では PDP-7 の CPU 命令を適宜紹介していきます。

まず紹介するのは `HLT` 命令です (表 1.2)。これは PDP-7 の実行を停止する命令です。シミュレータ的にもこの命令の実行でシミュレーション動作が停止します。

▼表 1.2: PDP-7 命令紹介: HLT

アセンブリ表記	機能
HLT	実行を停止

♣ SimH 上でベアメタルプログラミング

それでは、 `HLT` 命令を使用して「実行を停止するだけのプログラム」を作って実行してみましょう。(「作る」といっても `HLT` 命令 1 つだけですが。)

アドレス 0o100 以降にプログラムを配置することにします。 `deposit` コマンドを用いて、アドレス 0o100 へアセンブリ形式で `HLT` 命令を配置してみましょう (リスト 1.9)。

▼リスト 1.9: アドレス 0o100 へ HLT 命令を配置

```
sim> d -m 100 hlt  ←命令も大文字/小文字どちらも可
sim> e -m 100
100:    HLT        ←メモリアドレス0o100にHLT命令が配置された
sim> e 100
100:    740040     ←"HLT"というのはアセンブリ表記で実際の機械語はコレ
sim>
```

リスト 1.9 では、 `HLT` 命令が実際の機械語としては `740040` (8 進数) であることも分かりました。PDP-7 は命令長が固定長なので、その他の命令も 18 ビットの命令長です。

プロセッサが解釈するのは機械語なので、プロセッサに実行させたい命令は機械語で配置しておく必要があります。ただ、SimH は `deposit` 命令の `-m` という Switch を使うことで、メモリアドレスに配置する命令をアセンブリの表現で指定することができます。(すると、リスト 1.9 で確認した通り、実際には機械語で命令が配置されます。) 本書ではこの方法でメモリ上にプロセッサの命令を一つずつ並べていくことでプログラミングを行います。

アドレスが 1 進む単位は 18 ビット (しかもこれは命令長と同じ)

PDP-7 はアドレスが 1 進む際に進む量も 18 ビットです。現代の一般的なコンピュータはアドレスが 1 進むと 1 バイト (たいてい 8 ビット) 進みますが、これが PDP-7 の場合は 18 ビット (これを PDP-7 では「1 ワード」と呼んだりします) 進むという訳です。そして、PDP-7 は命令長も 18 ビット固定長なので、1 命令進む毎にアドレスも 1 進みます。

♣ SimH コマンド: go と run

それでは、これを実行してみましょう。SimH で実行を開始するコマンドとして `go` と `run` の 2 つがあります。違いは以下の通りです。

go コマンド
: 実行を開始するだけ。特にハードウェアのリセット等は行わない。

run コマンド
: シミュレータ上でハードウェア (レジスタの値等) をリセットした上で実行を開始する。

書式は両コマンド共に同じです。引数に何も指定されていなければ現在の PC レジスタに設定されているアドレスから実行を開始します。もし第 1 引数の指定があれば、そこに指定されたアドレスから実行を開始します。詳しくは後述のコラムで紹介する `help` 等を参照してみてください。

ここでは特にハードウェアのリセット等は不要なので、 `go` コマンドでアドレス 0o100 を指定して実行してみます (リスト 1.10)。なお、既に PC レジスタに 0o100 を設定済みの場合、引数無しでも構いません。

▼リスト 1.10: 実行してみる

```
sim> go 100

HALT instruction, PC: 00101 (CAL 0)  ←実行を停止(Halt)した
sim>
```

これで初めてのプログラム「実行を停止するだけのプログラム」を作って、動作確認することができました。

プロンプトに戻れない/実行を中断したい場合は Ctrl+e

実装にバグがありプロンプト (`sim>`) に戻れなくなってしまった、あるいはそうでなくとも実行を中断したいという場合は、Ctrl+e(Ctrl キーを押しながら e キー) で SimH 上の実行を強制的に中断し、プロンプト (`sim>`) に戻ることができます。

1.4　終了方法について

♣ 現在の状態の保存 (save) と保存した状態の復元 (restore)

SimH を終了する前に、現在のメモリやレジスタの状態を保存する場合は `save` (省略形 `sa`) コマンドを使用します (リスト 1.11)。

▼リスト 1.11: save コマンドの使用例

```
sim> save hoge.sav  ←カレントディレクトリにhoge.savというファイル名で状態を保存
sim> sa fuga.sav    ←「sa」のみでも可
sim> sa ~/piyo.sav  ←パス指定も可
sim>
```

このセーブデータファイルに何か決まった拡張子があるのかは分かっていません。本書では.sav という拡張子を付けることにしておきます。

そして、`save` コマンドで保存した状態は `restore` (省略形 `rest` あるいは `get` でも可) コマンド[*8]で復元できます (リスト 1.12)。

▼リスト 1.12: restore コマンドの使用例

```
sim> restore hoge.sav  ←カレントディレクトリのhoge.savに保存された状態を復元
sim> rest fuga.sav     ←「rest」のみでも可
sim> get ~/piyo.sav    ←「get」でも可。パス指定も可
sim>
```

[*8] 「セーブした状態のロード」と考えると"load"と打ってしまいそうになりますが、`save` コマンドで保存した状態を復元するコマンドは `restore` ですのでご注意ください。加えて、本書では使用しませんが `load` というコマンドはそれはそれで存在します。同様にファイルを引数に取るコマンドであり、うっかり `save` コマンドで保存したファイルを指定して実行してもエラーになったりはしないため、「何故か復元できない？」という事になりがちなので、重ねてご注意ください・・。

♣ SimH を終了する (exit)

SimH を終了する際は `exit` コマンドを使用します。このコマンドは `quit` ・ `bye` でも可で、それぞれに `exi` ・ `q` ・ `by` という省略形もあります。例えば、`bye` コマンドの省略形の `by` コマンドを使用するとリスト 1.13 の通りです。

▼リスト 1.13: bye コマンドの使用例

```
sim> by
Goodbye
$    ←元のシェルのプロンプトへ戻ってくる
```

次章以降はまた別のプログラムを作成しますので、続けて作業する場合も一旦終了し、SimH を起動し直してください。

【コラム】CPU 使用率が跳ね上がってしまう場合は

SimH を実行する環境にもよるのかもしれませんが、デフォルトでは SimH は無制限に CPU を使用するため、次章以降でループ処理を実装する等で実行時間が長くなると CPU 使用率が跳ね上がって高止まりしがちです。

SimH には CPU 消費を制限するスロットルの設定ができます。例えば SimH 上で 1 秒間に実行する命令数を 100 万 (1M) に制限する場合、リスト 1.14 の様に設定を行います。

▼リスト 1.14: 秒間の命令数を 100 万 (1M) に制限

```
sim> set throttle 1M
sim>
```

`set throttle` は、100 万単位 (`xM`) の他にも指定できる形式があります (表 1.3)。なお、詳しくはこの章の最後で紹介する `help` コマンドをご覧ください。(`help set throttle` というコマンドで参照できます。)

▼表 1.3: set throttle に指定できる形式一覧

形式	意味
`xM`	秒間の実行命令数を (x * 100 万) に制限
`xK`	秒間の実行命令数を (x * 1000) に制限
`x%`	CPU 使用率 x% に制限
`x/t`	x 回の命令実行後に t ミリ秒のスリープを入れる

ただし、これも SimH を実行する環境によるのかもしれませんが、私の場合、`x%` 指定ではその通りに SimH のプロセス (pdp7 プロセス) が制限されませんでした。その上、なぜか Ctrl+e で実行を停止することもできなくなってしまうので、pdp7 プロセス自体を Linux の `kill` コマンドで強制終了させる必要がありました。

また、`x/t` に関しても、これは確かに CPU 消費を抑えられるのですが、Ctr+e を押下してから停止するまでの時間もなぜか長くなってしまうのでこれも使いにくい感じです。

そのため、`xM` か `xK` で自身の環境に良い設定値を見つけて設定するのが良いと思います。なお、私の環境 (CPU: Core i7 1.8GHz、OS: Debian 11) では、`1M` の設定で pdp7 プロセスの CPU 使用率を 6% 程まで抑えることができ、次章以降で作成するプログラムの実行にも問題ない様子でした。

【コラム】simh スクリプトについて

SimH では、シェルスクリプトの様に SimH のコマンド並べたテキストファイルを読み込ませて実行させることができます。例えば、この章で行った「実行を終了するだけのプログラム」の場合、リスト 1.15 の様に書くと、`hlt` 命令のメモリへの配置と実行、そして SimH 自体の終了までを行うスクリプトとなります。

▼ リスト 1.15: 1 章の内容をスクリプトで記述

```
# [メイン処理]
d -m 100 hlt

# [実行]
go 100

# [終了]
by
```

`#` で始まる行はコメントですので、無くても構いません。見るとわかる通りかと思いますが、単に SimH のコマンドを並べているだけです。このテキストファイルを例えば 01-halt.simh というファイル名で保存したとすると、リスト 1.16 のように実行できます。

▼リスト 1.16: スクリプトの実行例

```
$ pdp7 01-halt.simh

PDP-7 simulator V4.0-0 Current        git commit id: 09899c18

HALT instruction, PC: 00101 (CAL 0)  ← hlt命令のメモリへの配置と実行
Goodbye                              ← SimH自体も終了
$
```

以降、本書ではこの様な SimH のコマンドが並べられたスクリプトを「simh スクリプト」と呼ぶことにします。拡張子は何であっても (あるいは無くても) スクリプトの実行に支障はありません。本書では.simh とします。[*9]

そして、本書のサンプルのリポジトリには、それぞれの章で行う内容を simh スクリプトの形で置いています。リスト 1.15 の内容もサンプルリポジトリに 01-halt.simh というファイル名で置いています。

リスト 1.15 の様に、サンプルリポジトリ内のそれぞれの simh スクリプトには、実行を開始する `go` コマンドや、SimH 自体を終了する `bye` コマンドまで書かれています。「単に `deposit` コマンドによる命令のロードだけを行いたい」といった場合には、適宜 `go` コマンドや `bye` コマンドをコメントアウトしてください。

なお、一部のサンプルはそもそもプログラム自体が自動的に停止 (`HLT`) しないものもあります。そのようなサンプルについては Ctrl+e で停止させてください。するとスクリプトに書かれている後続の `bye` コマンドが実行され SimH 自体が終了します。それぞれのサンプルの挙動について詳しくはそれぞれの章をご覧ください。(基本的にはそれぞれの章で行っている内容をスクリプト化しただけのものです。)

1.5 SimH コマンド: help

SimH はヘルプを内蔵しており、 `help` コマンドで閲覧できます。この章の最後に、 `help` コマンドの使い方を簡単に紹介します。

[*9] Open-SimH のリポジトリ内にもこのような拡張子で simh スクリプトが置かれていたりします。

まず、引数なしで実行すると `help` コマンド自体の説明が表示されます (リスト 1.17)。

▼リスト 1.17: 引数なしで実行してみる

```
sim> help
Help is available for devices

   HELP dev
   HELP dev SET
   HELP dev SHOW
   HELP dev REGISTERS

Help is available for the following commands:

    !              ASSERT         ASSIGN         ATTACH         BOOT
    BREAK          CALL           CAT            CD             CONTINUE
    COPY           CP             CURL           DEASSIGN       DEBUG
    DELETE         DEPOSIT        DETACH         DIR            DISKINFO
    DO             DUMP           ECHO           ECHOF          ELSE
    EVALUATE       EXAMINE        EXIT           EXPECT         GO
    GOTO           HELP           IDEPOSIT       IEXAMINE       IF
    IGNORE         LOAD           LS             MKDIR          MOVE
    MV             NEXT           NOBREAK        NODEBUG        NOEXPECT
    NORUNLIMIT     NOSEND         ON             PROCEED        PWD
    RENAME         RESET          RESTORE        RETURN         RM
    RMDIR          RUN            RUNLIMIT       SAVE           SCREENSHOT
    SEND           SET            SHIFT          SHOW           SLEEP
    STEP           TAR            TESTLIB        TYPE
sim>
```

英語ではありますが、できることが簡単に説明されています。まず、`help <デバイス名>` というコマンドで、SimH がシミュレートしているデバイスのヘルプを閲覧できます。例えば CPU に関するヘルプを閲覧する際は `help cpu` を実行します (リスト 1.18)。

▼リスト 1.18: CPU のヘルプを見る

```
sim> help cpu
CPU help

CPU device SET commands:  ← setコマンドで有効/無効を設定できるパラメーター一覧

set CPU NOEAE
set CPU EAE
 ・・・省略・・・
```

```
CPU device SHOW commands:  ← showコマンドで現在の状態を表示できるパラメーター一覧

show CPU IDLE
show CPU HISTORY{=arg}

The CPU device implements these registers:  ← CPUのレジスター一覧

  Name            Size  Purpose
  ------------------------------
      PC            15  program counter
      AC            18  accumulator
      L              1  link
      MQ            18  multiplier-quotient
・・・省略・・・

sim>
```

ここで注目すべきは CPU のレジスタ一覧です。これを見ると `PC` が 15 ビットであることや、 `AC` が 18 ビットであることが分かります。

また、デバイスの各種設定を有効化/無効化する `set` コマンドは後ほど 3 章以降で使用します。SimH がシミュレートしているデバイス一覧は `show devices` というコマンドで得られます。このコマンドも 3 章で使用していますので、詳しくはそちらをご覧ください。

次に、SimH のコマンドのヘルプを見てみましょう。引数なしで実行した際に「Help is available for the following commands:」の箇所に一覧されていた内容が SimH で使用できるコマンド一覧です。試しに `examine` コマンドのヘルプを見てみます (リスト 1.19)。

▼リスト 1.19: examine コマンドのヘルプを見る

```
sim> help examine
PDP-7 help.  Type <CR> to exit, HELP for navigation help
Examining and Changing State
    There are four commands to examine and change state:

            EXAMINE (abbreviated E) examines state
            DEPOSIT (abbreviated D) changes state
・・・省略・・・
    The "object list" consists of one or more of the following, separated by
    commas:

            register                the specified register
・・・省略・・・
            address                 the specified location
```

```
            address1-address2       all locations starting at address1 up to
                                    and including address2
            address/length          all location starting at address up to
                                    but not including address+length
 ・・・省略・・・

    Additional information available:

    Switches    Examples

PDP-7 Commands Examining and Changing State Subtopic? ▌  ←ここでユーザ入力待ち
になる
```

`examine` コマンド自体が高機能なコマンドである事に加えて、`deposit` コマンドについても一緒に説明しているのでとても長いヘルプです。リスト 1.19 では色々と省略していますが、`examine` と `deposit` が `e` や `d` でも良い事や、引数としてレジスタやアドレスを取る事などが書かれています。

ここで、`sim>` のプロンプトには戻ってこず、「PDP-7 Commands Examining and Changing State Subtopic?」という質問でユーザ入力待ちになっています。これは SimH のヘルプ機能の特徴的なところで、表示したヘルプにサブトピックがある場合、「サブトピックを表示するか？」という質問と共にユーザ入力待ちになります。

今回の場合、「Additional information available:」に並んでいる「Switches」と「Examples」が表示できるサブトピックです。ここでは「Switches」を見てみましょう (リスト 1.20)。

▼ リスト 1.20: サブトピック Switches のヘルプを見る

```
PDP-7 Commands Examining and Changing State Subtopic? switches

Switches
    Switches can be used to control the format of display information:

            -a                  display as ASCII
            -c                  display as character string
            -m                  display as instruction mnemonics
            -o or -8            display as octal
            -d or -10           display as decimal
            -h or -16           display as hexadecimal
            -2                  display as binary

    The simulators typically accept symbolic input (see documentation with e
ach
    simulator).
```

```
PDP-7 Commands Examining and Changing State Subtopic?
```

`examine` や `deposit` で指定可能な Switches の一覧が表示されました。

再びサブトピック指定のユーザ入力待ちになっています。ヘルプ機能から抜ける場合は何も指定せずに Enter を押下します (リスト 1.21)。

▼リスト 1.21: サブトピック閲覧から抜ける

```
PDP-7 Commands Examining and Changing State Subtopic?   ←何も指定せずEnter

Commands

    Additional information available:  ←さらに他のサブトピックが提案される

    Resetting_Devices
    Examining_and_Changing_State
・・・省略・・・

PDP-7 Commands Subtopic?  ←何も指定せずEnter

PDP-7

    Additional information available:

    Commands

PDP-7 Subtopic?  ←もう一度何も指定せずEnter
sim>  ←元のプロンプトに戻ってきた
```

以上が SimH のヘルプ機能の簡単な使い方です。ひとまずは「サブトピックを表示するモードに入った場合、そこから戻るには Enter を連打すれば良い」と覚えておけば十分です。

なお、SimH について詳しくはドキュメントも参照してみてください。本書で対象にしている Open-SimH の場合、リポジトリの doc ディレクトリにドキュメントが置かれています。(なんと Word の doc 形式ではありますが。)simh_doc.doc 辺りが SimH の使い方に関するドキュメントです。

第 2 章
計算を行ってみる (疑似乱数生成)

この章では、基本的な計算を行う例として線形合同法による疑似乱数生成を行ってみます。

なお、この章の内容を実装した simh スクリプトはサンプルリポジトリ内の 02-lcgs.simh です。

2.1 線形合同法について

線形合同法は疑似乱数を求める手法の一つです。式としては式 2.1 の通りで、この漸化式で得られる X_{n+1} が擬似乱数となります。

式 2.1: 線形合同法の漸化式

$$X_{n+1} = (A \times X_n + B) \bmod M$$

この漸化式は無限に擬似乱数を生成する訳ではなく周期性があります。M の剰余を取っているため、その周期は最大でも M です。周期を最大 (M) にできる定数 A・B・M の条件は以下の通りであることが知られています[*1]。

1. B と M が互いに素である。
2. A-1 が、M の持つ全ての素因数で割りきれる。
3. M が 4 の倍数である場合は、A-1 も 4 の倍数である。

[*1] ref. 線形合同法 - Wikipedia https://ja.wikipedia.org/wiki/%E7%B7%9A%E5%BD%A2%E5%90%88%E5%90%8C%E6%B3%95

ここでは、A = 5、B = 3、M = 262144 とすることにします。262144 は 8 進数で 1000000 です。0o1000000 と剰余をとる事は、下位 18 ビットだけを残してそれより上位のビットを捨てる事と同じです。PDP-7 はメモリアクセス幅が 18 ビットであり計算の命令で使用する AC レジスタ等も 18 ビットの大きさであるため、特に何もしなくとも 18 ビットより上位のビットは捨てられます。そのため PDP-7 上においては、M = 262144 とすることで $\mathrm{mod} M$ を省略できます。

以上を踏まえると、PDP-7 上で計算する場合においては、A = 5、B = 3、M = 262144 の場合、漸化式は式 2.2 となります。

式 2.2: PDP-7 上にて、A = 5、B = 3、M = 262144 の場合の漸化式

$$X_{n+1} = 5 \times X_n + 3$$

PDP-7 は左からビットを数える (が本書では右から数えることにします)

実は PDP-7 ではビット番号は最も左のビット (最上位のビット) から順にビット 0、ビット 1、ビット 2、と数えます。左側が最下位ビット (LSB) で右側が最上位ビット (MSB) という訳です。

ですが、これは現代のよく知られた CPU などの慣習からすると (加えて私としても) 直感的に分かり難く誤解を招く恐れもあるので、本書では最も右側のビットを最下位ビットとして、右から順にビット 0、ビット 1、ビット 2、と数えることにします。

2.2　漸化式を組み立てる

それでは、前節で確認した漸化式を PDP-7 上で実現してみます。

♣ X_n と X_{n+1} には AC レジスタを使う

X_n と X_{n+1} には AC レジスタを使うことにします。AC(ACcumulator) レジスタは 18 ビットのレジスタで、PDP-7 の大抵の演算命令でこのレジスタを使用します。このレジスタにこの漸化式の入力値である X_n が入っていると計算がしやすいです。

また、計算結果 (X_{n+1}) も AC レジスタに設定することで、計算後、AC レジスタはそのままに再度計算を行うことで X_{n+2} 以降も求めやすくなります。

♣ 定数 5 と AC レジスタを乗算 (MUL 命令)

前述の通り、AC レジスタには X_n(特に最初の場合は X_0) が入っているという前提で、まずは $5 \times X_n$ を計算してみます。

`MUL` 命令で乗算を行えます (表 2.1)。

▼表 2.1: PDP-7 命令紹介: MUL

アセンブリ表記	機能
MUL	この命令自身の次のアドレスの値と AC レジスタの値を 符号無し整数として乗算。結果の上位 18 ビットを AC レジスタ、 下位 18 ビットを MQ レジスタへ設定

MQ(Multiplier Quotient) レジスタも 18 ビットのレジスタです。このレジスタは正に `MUL` 命令のためのレジスタで、乗算結果の 36 ビット値の下位 18 ビットをこのレジスタへ格納します。(上位 18 ビットは AC レジスタへ)

それでは、`MUL` 命令を使用して定数 5 と AC レジスタの乗算を実装してみます。SimH 上でリスト 2.1 の様に入力してください。

▼リスト 2.1: 5 と AC レジスタの乗算を実装

```
sim> d -m 201 mul
sim> d 202 5
sim> d -m 203 hlt  ←一旦ここまでで停止
sim>
```

この様に、`MUL` 命令を置いた次のアドレスに数値を置いておくことで、`MUL` 命令実行時にその数値を読み取り、AC レジスタの値との乗算を行います。(なお、アドレス 0o201 以降に配置しているのは次節で関数化する際の都合です。)

♣ 一旦乗算だけで動作確認

それでは、これを実行してみましょう (リスト 2.2)。

▼リスト 2.2: 5 と AC レジスタの乗算を実行

```
sim> d ac 2        ←例としてX_0 = 2としておく
sim> go 201

HALT instruction, PC: 00204 (CAL 0)
sim> e -d ac
AC:     000000     ←上位18ビット: 0
sim> e -d mq
MQ:     000010     ←下位18ビット: 10(10進数)
sim>
```

5 × 2 = 10 を計算できました。なお、1 章でも紹介しましたが、`examine` / `deposit` コマンドは、`-d` (あるいは `-10`) を指定することで、数値を 10 進数で表示/設定できます。

♣ MQ レジスタの値を AC レジスタへ設定 (LACQ 命令)

そして、次の演算のために MQ レジスタに設定されている値を AC レジスタへ設定しておくことにします。それを行ってくれる命令が `LACQ` です (表 2.2)。

▼表 2.2: PDP-7 命令紹介: LACQ

アセンブリ表記	機能
LACQ	MQ レジスタの値を AC レジスタへ上書きする MQ レジスタの値は変わらない

`MUL` 命令の後に `HLT` 命令を置いていましたが、これを `LACQ` 命令へ変更しておきます (リスト 2.3)。

▼リスト 2.3: MUL 命令の後を LACQ 命令へ変更

```
sim> d -m 203 lacq
sim>
```

♣ 定数 3 を加算 (ADD 命令)

続いて、乗算結果が格納されている AC レジスタへ定数 3 を加算します。加算には `ADD` 命令を使用します (表 2.3)。

▼表 2.3: PDP-7 命令紹介: ADD

アセンブリ表記	機能
ADD <アドレス>	指定されたアドレスの値を AC へ加算 負数は 1 の補数で指定

`ADD` 命令は引数 (これを「オペランド」と呼んだりします) として「アドレス」を取ります。そのため、加算する値を予めメモリ上のどこかに配置しておく必要があります。ここでは、アドレス 0o260 に定数 3 を置いておくことにします。定数の配置と `ADD` 命令の追加はリスト 2.4 の通りです。

▼リスト 2.4: ADD 命令を追加

```
sim> d 260 3            ←アドレス0o260へ3を配置
sim> d -m 204 add 260   ←アドレス0o260の値をACへ加算
sim> d -m 205 hlt       ←ここまでで停止
sim>
```

ここまでで漸化式 $X_{n+1} = 5 \times X_n + 3$ の実装完了です。改めて実装した内容を表示するとリスト 2.5 の通りです。

▼リスト 2.5: 実装内容を確認

```
sim> e -m 201-205
201:    MUL
202:    CAL 5   ←ここは単なる数値(定数5)なので逆アセンブル結果は気にせず
203:    LACQ
204:    ADD 260
205:    HLT
sim> e 202      ←アドレス0o202は数値として確認
202:    000005  ←ちゃんと定数5が入っている
sim> e 260      ←アドレス0o260の定数を確認
260:    000003  ←定数3
sim>
```

♣ 実装した漸化式の動作確認

それでは、実行してみましょう (リスト 2.6)。

▼リスト 2.6: 漸化式の動作確認

```
sim> d ac 2     ← ACへX_0として2を設定
sim> go 201     ← X_1 = 5 * 2 + 3

HALT instruction, PC: 00206 (CAL 0)
sim> e -d ac
AC:     000013  ← X_1 = 13
sim> go 201     ← X_2 = 5 * 13 + 3

HALT instruction, PC: 00206 (CAL 0)
sim> e -d ac
AC:     000068  ← X_2 = 68
sim> go 201     ← X_3 = 5 * 68 + 3

HALT instruction, PC: 00206 (CAL 0)
sim> e -d ac
AC:     000343  ← X_3 = 343
sim>
```

ちゃんと計算できていて良さそうです。

簡単な漸化式であるため、計算すれば予測はできてしまいますが、これでも周期 262144、すなわち 262144 回漸化式を繰り返さないと同じ値は出てこないものになっています。

2.3　関数化する

この節では、前節で実装した漸化式を関数 (サブルーチン) 化し、別の処理から呼び出せるようにします。

♣ 関数呼び出し (JMS)

関数呼び出しには `JMS` という命令を使います (表 2.4)。

▼表 2.4: PDP-7 命令紹介: JMS

アセンブリ表記	機能
JMS <アドレス>	指定されたアドレスへ戻り先アドレスを設定 指定されたアドレス +1 へジャンプ

挙動は動作させてみた方がわかりやすいので、少し実験してみましょう (リスト 2.7)。

▼リスト 2.7: JMS 命令の実験

```
sim> d -m 100 jms 300
sim> d pc 100    ← PCへJMSを配置したアドレスを設定
sim> s           ← 1ステップだけ実行

Step expired, PC: 00301 (CAL 0)  ← JMSに指定したアドレス+1へジャンプした
sim> e 300
300:    000101  ← JMSに指定したアドレスへJMSを実行したアドレス+1が設定された
sim>
```

`s` というコマンドは `step` というコマンドの省略形で、コマンド名の通りステップ実行するコマンドです。試しに実行してみる際に、実行したい部分の最後にいちいち `HLT` 命令を置くのが面倒であったりする場合等は、この様に予め PC レジスタへ実行したいアドレスを設定した上でステップ実行する、という方法もあります。

リスト 2.7 を見ると、 `JMS` 命令の挙動として、 `JMS` 命令に指定されたアドレス 0o300 へ「 `JMS` 命令を実行したアドレス +1」である 0o101 が戻り先として設定さ

れ[*2]、次に実行するアドレスとして「`JMS` 命令に指定されたアドレス +1」である 0o301 が PC レジスタに設定されていることが分かります[*3]。

以上が `JMS` 命令の挙動です。本書では、`JMS` 命令に指定するアドレスをその関数のアドレスと呼ぶことにします。先述の通り、実際に実行されるのは「`JMS` 命令に指定したアドレス +1」からである点にご注意ください。

♣ 関数からのリターン (JMP I)

前項で `JMS` 命令を使えば戻り先アドレスを保存しつつ関数へジャンプできることが分かりました。関数から呼び出し元へ戻る際は `JMP` という命令に「間接アドレス指定 (`I`)」という指定を追加した命令を使用します。

まず、通常の `JMP` 命令の挙動を紹介します (表 2.5)。

▼表 2.5: PDP-7 命令紹介: JMP

アセンブリ表記	機能
JMP <アドレス>	指定されたアドレスへジャンプする

単純に「指定されたアドレスへジャンプする」だけです。具体的には「指定されたアドレスを PC レジスタへ設定する」だけです。[*4] `JMS` 命令の様に「戻り先アドレスをどこかに保存する」等といったことは行いません。

そして、「間接アドレス指定」ですが、これは PDP-7 に限らず現代の CPU でも持っているアドレス指定方式で、「指定されたアドレスを使うのではなく、指定されたアドレスに書かれているアドレスを使う」というものです。PDP-7 の (少なくとも SimH 上の) アセンブリ言語では、例えば `JMP I <アドレス>` の様に、間接アドレス指定に対応した命令に `I` を付けると間接アドレス指定を行えます。間接アドレス指定の `JMP` 命令 (`JMP I` 命令) の機能をまとめると表 2.6 の通りです。

▼表 2.6: PDP-7 命令紹介: JMP I

アセンブリ表記	機能
JMP I <アドレス>	指定されたアドレスに書かれているアドレスへジャンプする

[*2] 現代の一般的な CPU であれば戻り先アドレスはスタックに積む所ですが、PDP-7 にはスタックという概念は (私の調べる限り) 無いようです。

[*3] そのため、このまま実行を続けると、0o301 に書かれている命令から順に実行していくことになります。

[*4] ちなみに、相対ジャンプ命令は無い様です。

`JMP I` を使うと、関数の中から戻り先へジャンプすることができます。挙動を確認するために少し実験してみましょう (リスト 2.8)。

▼ リスト 2.8: JMP I 命令の実験

```
sim> d -m 301 jmp i 300  ← 0o300に書かれているアドレスへジャンプ(return)
sim> e -m 301
301:    JMP I 300        ←アドレス0o300に、「returnするだけの関数」がある状態
sim> d -m 100 jms 300    ← 0o100へ、JMS 300(アドレス0o300の関数呼出)を配置
sim> d pc 100            ← PCへアドレス0o100を設定
sim> s                   ← PC=0o100の状態で、1ステップだけ実行

Step expired, PC: 00301 (JMP I 300)  ← JMSに指定したアドレス+1へジャンプした
sim> e 300
300:    000101  ← JMSに指定したアドレス(0o300)へ関数のreturn先として「このJMS命令のアドレス+1(0o101)」が設定された
sim> s                   ← PC=0o301の状態で、1ステップだけ実行

Step expired, PC: 00101 (CAL 0)  ← JMP I 300が実行され、実行したJMS命令の次のアドレス(0o101)へ戻ってきた
sim>
```

これで、関数の中から return することもできるようになりました。

♣ [補足] 間接アドレスで 0o1x を指定した場合の特殊な挙動に注意

間接アドレス指定で指定するアドレスが 0o1x(0o10 以上かつ 0o17 以下の値) の場合、参照先の値を予めインクリメントするという特殊な挙動があります。

どういうことなのか、実際の挙動を紹介します。先程、`JMP I 300` という命令を試しましたが、これを `JMP I 10` に変えるとどうなるのか実験してみるとリスト 2.9 の通りです。

▼ リスト 2.9: 関節アドレス指定のアドレスが 0o10 の場合の挙動

```
sim> d -m 11 jmp i 10  ← 0o10に書かれているアドレスへジャンプ(return)
sim> e -m 11
11:    JMP I 10        ←アドレス0o10に、「returnするだけの関数」がある状態
sim> d -m 100 jms 10   ←アドレス0o100へ、JMS 10(アドレス0o10の関数呼出)を配置
sim> d pc 100          ← PCへアドレス0o100を設定
sim> s                 ← PC=0o100の状態で、1ステップだけ実行

Step expired, PC: 00011 (JMP I 10)   ← JMSに指定したアドレス+1へジャンプした
sim> e 10
10:    000101  ← JMSに指定したアドレス(0o10)へ関数のreturn先として「このJMS命令のアドレス+1(0o101)」が設定された
sim> s                 ← PC=0o11の状態で、1ステップだけ実行
```

```
Step expired, PC: 00102 (CAL 0)  ←アドレス0o10に格納されていた値(0o101)+1へジ›
›ャンプしてきた
sim> e 10
10:     000102          ←インクリメントされている
sim>
```

`JMP I 10` は、まとめると以下の挙動となっています。

1. アドレス 0o10 の先の値 (0o101) をインクリメント (0o102 になる)
2. アドレス 0o10 に格納されているアドレス (0o102) へジャンプ

関節アドレス指定で 0o300 を指定した時のように、通常は上記の 1. の挙動はありません。関節アドレス指定で指定するアドレスが 0o1x の場合だけ上記のような挙動となります。[*5]

この様に 0o10 から 0o17 の範囲では、関節アドレス指定で特殊な挙動をするため、基本的に処理は 0o20 以降に実装するほうがトラブルが少なそうです。きりの良さもあり、本書では基本的に処理は 0o100 以降に実装するようにしています。

♣ 漸化式を関数化

それでは、前節で作成した漸化式の処理を関数化して呼び出してみましょう。

まずは、アドレス 0o201 から 0o205 に実装していた漸化式の一番最後 (0o205) に置いていた `HLT` 命令を `JMP I` 命令へ変更します (リスト 2.10)。

▼リスト 2.10: 0o205 の HLT を JMP I へ変更

```
sim> e -m 201-205  ←現状の確認
201:    MUL
202:    CAL 5
203:    LACQ
204:    ADD 260
205:    HLT
sim> d -m 205 JMP I 200
```

[*5] このような挙動であることは SimH のソースコードで確認しました。本書で対象としている Open-SimH の#09899c18 コミット時点の場合、`JMP I` 命令の実装は PDP18B/pdp18b_cpu.c の 963 行目から 971 行目の辺りにあります。そこに `if (Ia (MA, &MA, 1))` という if 文があり、この時、変数 `MA` には関節アドレス指定のアドレスが格納されています。`Ia` 関数は、PDP-7 に対する実装が、同ファイルの 1779 行目から 1799 行目にあります。この `Ia` 関数の中に `if ((ma & B_DAMASK & ~07) == 010)` という if ブロックがあり、ここで「ma(関節アドレス指定のアドレス) が 0o1x か否か」の判定を行っています。そしてその判定が真である場合、`add 1 before use` というコメントが付けられている処理が実行されることで、関節アドレス指定のアドレス先の値がインクリメントされます。この様な実装は他の命令の関節アドレス指定でも同様であるようで、少なくとも `LAC` 命令では同様に実装されていました。

```
sim> e -m 201-205
201:    MUL
202:    CAL 5
203:    LACQ
204:    ADD 260
205:    JMP I 200  ← JMP I命令へ変更された
sim>
```

これで、0o200 を指定して `JMS` 命令で関数呼び出しすれば、0o200 に戻り先アドレスが設定され、0o201 から実行開始した後に、0o205 の `JMP I 200` で 0o200 に設定されている戻り先アドレスへ return するようになります。(この様にするために前節ではアドレス 0o201 以降に漸化式を実装していました。)

♣ 動作確認

動作確認のため、この関数を呼び出す処理を実装しておきます (リスト 2.11)。

▼ リスト 2.11: 漸化式の関数を呼び出す実装を追加

```
sim> d -m 100 jms 200  ←アドレス0o200の関数を呼び出し
sim> d -m 101 hlt      ←戻ってきたら実行停止
sim>
```

それではこれを実行してみましょう (リスト 2.12)。

▼ リスト 2.12: 一連の動作確認

```
sim> d ac 2     ← X_0として2を設定
sim> go 100     ←関数呼び出しを実施

HALT instruction, PC: 00102 (CAL 0)
sim> e -d ac    ←結果確認
AC:     000013  ← X_1 = 13
sim> go 100     ←もう一度実施

HALT instruction, PC: 00102 (CAL 0)
sim> e -d ac
AC:     000068  ← X_2 = 68
sim> go 100     ←さらにもう一度実施

HALT instruction, PC: 00102 (CAL 0)
sim> e -d ac
AC:     000343  ← X_3 = 343
sim>
```

関数化した場合も結果は変わらずに実行できることを確認できました。

第 3 章

テレタイプで文字の入出力 (HELLO WORLD!/エコーバック)

この章では、テレタイプによる文字の入出力を紹介し、「HELLO WORLD!」を出力するプログラムと、入力出力両方を用いてエコーバックを行うプログラムを作ってみます。

なお、この章の内容を実装した simh スクリプトはサンプルリポジトリ内の 03-1-hello.simh(「HELLO WORLD!」を出力するプログラム) と 03-2-echoback.simh(エコーバックを行うプログラム) です。

3.1 テレタイプとは

テレタイプ (Teletype) は、キーボードとプリンタが一体となった入出力デバイスです。[*1]通信により遠隔とやり取りできるタイプライターという事で「テレタイプ」や「テレタイプライター」と呼ばれていました。キーボードとプリンタを備えていて通信もできることから、当時はメインフレームや PDP-7 等のミニコンピュータの入出力用の端末としても使われていたそうです。[*2]なお、SimH のコードによると、PDP-7

[*1] 現在の「TTY」という言葉はテレタイプ (TeleTYpe) から来ています。

[*2] ref. テレタイプ端末 - Wikipedia https://ja.wikipedia.org/wiki/%E3%83%86%E3%83%AC%E3%82%BF%E3%82%A4%E3%83%97%E7%AB%AF%E6%9C%AB

には「KSR-33」というテレタイプが接続されていたようです。[*3]

SimH 上でテレタイプは、SimH を起動しているターミナル画面上への文字出力とキー入力となります。

3.2　指定された 1 文字を出力する関数を作る

♣ テレタイプで 1 文字出力する (TLS 命令)

テレタイプで 1 文字出力する命令として `TLS` 命令があります (表 3.1)。

▼表 3.1: PDP-7 命令紹介: TLS

アセンブリ表記	機能
TLS	AC レジスタの下位 8 ビットに設定された文字 (ASCII) をテレタイプの出力用バッファへ設定

`TLS` 命令は出力用バッファに設定するだけです。バッファに設定できた時点で命令自体は完了してしまうのでご注意ください。例えば、`TLS` 命令の直後に `HLT` 命令を置いた場合、バッファに設定された文字が出力される前に `HLT` 命令で実行を停止してしまいます。実験してみるとリスト 3.1 の通りです。

▼リスト 3.1: TLS 命令の実験

```
sim> d ac 101           ← 0o101はASCIIの'A'
sim> d -m 100 tls
sim> d -m 101 hlt       ← TLSの直後にHLTを配置
sim> go 100

HALT instruction, PC: 00102 (CAL 0)  ←文字の出力無く停止してしまう
sim> d -m 101 jmp 101  ← TLSの直後を無限ループに変更
sim> go 100
A                       ←文字が出力された
```

アドレス 0o101 に配置した `jmp 101` はこの `JMP` 命令自身のアドレスへジャンプし続ける無限ループです。2 度目の `go` コマンド実行後はこの無限ループにより、待っていても戻ってこないので、Ctrl+e で抜けてください。

[*3] ref. https://github.com/open-simh/simh/blob/09899c18/PDP18B/pdp18b_defs.h#L82

examine/deposit の-a/-c はレジスタには使えない模様

第 1 章で紹介した通り、 examine / deposit コマンドには ASCII 形式で表示/設定する -a という switch があります。 TLS 命令の実験 (リスト 3.1) で AC レジスタへ ASCII の'A' を設定する際、なぜそれを使わなかったのかというと、どうやらレジスタへの設定には対応していないようだったためです。

試してみると分かりますが、 -a / -c はレジスタに対して使用すると、 examine は何も効いていないのかデフォルトの 8 進数で表示され、 deposit は Invalid argument となります。確認できていませんが、その他にも同様にレジスタには使えない switch があるかもしれません。

なお、 deposit コマンドの -a の switch でメモリへ ASCII 文字を配置すると、その際になぜかビット 7 に 1 が設定される挙動がありますが、本書で扱う範囲では特に害は無いため気にしなくて構いません。(これについては後ほどまた説明します。)

♣ 1 文字ずつ出力を待ち合わせる (TSF 命令・TCF 命令)

TSF 命令と TCF 命令を使うと 1 文字ずつ出力を待ち合わせる事ができます (表 3.2)。

▼表 3.2: PDP-7 命令紹介: TSF・TCF

アセンブリ表記	機能
TSF	文字出力完了フラグがセットされている場合、次の命令をスキップする
TCF	文字出力完了フラグをクリアする

実は内部的に文字出力の完了を示すフラグがあり、 TLS 命令でバッファに設定された文字の出力が完了するとこのフラグがセットされます。 TSF 命令実行時、このフラグがセットされている場合は次の命令を飛ばして次の次の命令から実行します。

TCF 命令は文字出力完了フラグをクリアする命令です。文字出力完了フラグはハードウェアのリセットを除いて、 TCF 命令以外ではクリアされません。

TSF 命令と TCF 命令を使用すると、例えばリスト 3.2 の様に 1 文字ずつ待ち合わせながら文字出力を行わせる事ができます。

▼リスト 3.2: 1 文字ずつ出力を待ち合わせる

```
sim> d -m 301 tcf       ←フラグをクリア
sim> d -m 302 tls       ←文字をバッファへ設定
```

```
sim> d -m 303 tsf       ←フラグがセットされていれば次の命令をスキップ
sim> d -m 304 jmp 303   ← TSF命令へ戻る
sim> d -m 305 hlt       ←停止
sim>
```

予め TCF 命令で文字出力完了フラグをクリアしておき、TLS 命令実行後、フラグがセットされる (文字出力が完了する) まで TSF 命令を繰り返す、という流れです。

試しに実行してみましょう (リスト 3.3)。

▼リスト 3.3: 1 文字ずつ出力を待ち合わせる実装を試す

```
sim> d ac 101
sim> go 301
A
HALT instruction, PC: 00306 (CAL 0)  ←文字出力を待ち合わせてから停止できた
sim>
```

♣ 関数化する

この節の最後に、前項で実装した処理を関数化します。最後の HLT 命令を JMP I に変更すれば、「AC レジスタに設定された文字を出力する」関数となります (リスト 3.4)。

▼リスト 3.4: 1 文字出力する関数化する

```
sim> d -m 305 jmp i 300  ← JMP I命令へ変更
sim> e -m 301-305
301:    TCF
302:    TLS
303:    TSF
304:    JMP 303
305:    JMP I 300  ←出力完了したらreturnする関数になった
sim>
```

これで、アドレス 0o300 で呼び出せる関数となりました。

♣ 動作確認

では、動作確認してみましょう。関数呼び出しして戻ってきたら停止するだけの処理を用意し、それを go コマンドで実行します (リスト 3.5)。

▼リスト 3.5: 1 文字出力する関数の動作確認

```
sim> d -m 100 jms 300  ←アドレス0o300の関数呼び出し
sim> d -m 101 hlt      ←関数から戻ったら停止
```

```
sim> go 100              ←実行開始
A
HALT instruction, PC: 00102 (CAL 0)
sim>
```

これで、「指定された 1 文字を出力する関数」を実装することができました。

3.3　指定された文字列を出力する関数を作る

続いて、前節の関数を用いて今度は文字列を出力する関数を作成してみます。ここでは「AC レジスタにアドレスで指定されている文字列を出力する」関数をアドレス 0o400 に作成することにします。[*4]

♣ AC レジスタの値をメモリへ書き込み (DAC 命令)

まず AC レジスタに設定されている値 (文字列のアドレス) を今後のためにメモリへ書き込んでおきます。これが言わば関数のローカル変数ということになります。AC レジスタの値をメモリへ書き込む命令は `DAC` 命令です (表 3.3)。

▼表 3.3: PDP-7 命令紹介: DAC

アセンブリ表記	機能
DAC <アドレス>	AC レジスタの値を指定されたアドレスへ書く

先述の通り AC レジスタは 18 ビットのレジスタであり、`DAC` 命令のメモリアクセスの単位も 18 ビットです。

それでは、この命令を使用して AC レジスタに設定されている文字列のアドレスをメモリへ書き込むように実装します。ここでは、アドレス 0o460 に書き込むようにしてみます (リスト 3.6)。

▼リスト 3.6: AC レジスタの値をアドレス 0o460 へ書き込み

```
sim> d -m 401 dac 460
sim>
```

♣ アドレス先の文字を取得 (LAC I 命令)

次に、文字を出力する関数へ渡すために、文字列から 1 文字取得します。使用する

[*4] そのため、処理自体は 0o401 以降に行います。

命令は LAC I 命令 (間接アドレス指定の LAC 命令) です。まず、純粋な LAC 命令を紹介します (表 3.4)。

▼表 3.4: PDP-7 命令紹介: LAC

アセンブリ表記	機能
LAC <アドレス>	指定されたアドレスの値を AC レジスタへ設定

DAC 命令とは逆に、指定されたアドレスの値を読み出して AC レジスタへ設定してくれる命令が LAC 命令です。メモリアクセス単位は DAC 命令と同様に 18 ビットです。

これに間接アドレス指定 I を付けた LAC I 命令については表 3.5 の通りです。

▼表 3.5: PDP-7 命令紹介: LAC I

アセンブリ表記	機能
LAC I <アドレス>	指定されたアドレスに書かれているアドレスの値を AC レジスタへ設定

これをリスト 3.7 の様に使用します。

▼リスト 3.7: 0o460 に書かれているアドレスから 1 文字取得

```
sim> d -m 402 lac i 460
sim>
```

この様に使用することで、アドレス 0o460 に書かれているアドレスから読み出して AC レジスタへ設定します。アドレス 0o460 には文字列のアドレス、即ち 1 文字目のアドレスが書かれているため、このアドレスから読み出すことで 1 文字目の値 (ASCII) を読み出すことができ、それが AC レジスタへ設定されます。なお、後述しますが、この様に読み出しを行うため、用意する文字列のデータ形式としては、18 ビット単位で ASCII の値を並べることにします。

なお、後ほどループのためのジャンプ命令を設置し、この LAC I 命令を繰り返し実行することにします。そのため、この LAC I 命令は 1 文字目だけでなくそれ以降もこの命令で文字の取得を行います。

♣ 取得した文字が NULL 文字か否かで分岐 (SZA 命令)

そして、終端処理として、取得した文字が文字列末尾の NULL 文字 (0) の場合はそこで文字の取得をやめる必要があります。AC レジスタの値が 0 か否かによって分岐する命令が SZA 命令です (表 3.6)。

▼表 3.6: PDP-7 命令紹介: SZA

アセンブリ表記	機能
SZA	AC レジスタが 0 の場合、次の命令をスキップ

なお、値は 2 の補数で扱うため、0 値は純粋に 0 のみです。[*5]
この命令も先程の `LAC I` 命令に続いて配置しておきます (リスト 3.8)。

▼リスト 3.8: 取得した文字が NULL 文字か否かで分岐

```
sim> d -m 403 sza
sim>
```

♣ 取得した文字が NULL 文字の場合、return

もし取得した文字が NULL 文字の場合、文字列の出力は完了しているのでこの関数からは return することにします。`SZA` 命令は「AC レジスタが 0 の場合、次の命令をスキップ」というものなので、以下の流れで命令を並べることにします。

1. `SZA` 命令
2. 4. へジャンプする `JMP` 命令 ← NULL 文字の場合、この命令がスキップ
3. この関数から return する `JMP I` 命令
4. この関数の後続処理

これを実装するとリスト 3.9 の通りです。

▼リスト 3.9: NULL 文字の場合、return

```
sim> d -m 404 jmp 406     ← NULL文字でないなら後続処理へジャンプ
sim> d -m 405 jmp i 400   ← NULL文字ならreturn
sim>
```

♣ AC レジスタの文字を出力する

それでは後続処理の部分を実装します。ここまでで、AC レジスタに取得した文字が NULL でない事が分かっているので、この文字を前節で作成した文字出力関数を使用して出力します (リスト 3.10)。

▼リスト 3.10: AC レジスタの文字を出力

```
sim> d -m 406 jms 300  ←文字出力の関数呼び出し
sim>
```

[*5] 1 の補数の場合は全てのビットが 1 の値も 0 値となります。

♣ 文字のアドレスを 1 つ進める

1 文字出力したので、アドレス 0o460 に設定されている文字のアドレスを 1 つ進めましょう。流れとしては、以下の 3 ステップです。

1. アドレス 0o460 の値 (文字のアドレス) を AC レジスタへ設定 (`LAC` 命令)
2. AC レジスタへ 1 を加算 (`ADD` 命令)
3. AC レジスタの値をアドレス 0o460 へ設定 (`DAC` 命令)

なお、 `ADD` 命令で AC レジスタへ加算する値はアドレスで指定する必要があるため、定数 1 を予め適当なアドレス (ここでは 0o461 にします) に書いておきます。

以上を実装するとリスト 3.11 の通りです。

▼リスト 3.11: 文字のアドレスを 1 つ進める

```
sim> d 461 1            ←アドレス0o461へ定数1を設定
sim> d -m 407 lac 460   ←アドレス0o460の値をACレジスタへ設定
sim> d -m 410 add 461   ← ACレジスタへアドレス0o461の値(定数1)を加算
sim> d -m 411 dac 460   ← ACレジスタの値をアドレス0o460へ設定
sim>
```

一応の補足ですが、アドレス 0o407 の `LAC` 命令の次の `ADD` 命令のアドレスが 0o410 なのは 8 進数であるためです。[*6]

♣ 文字の取得へ戻る (ループ処理)

この関数の最後に、文字取得を行っていた所まで戻る `JMP` 命令を配置して、ループ処理を完成させます (リスト 3.12)。

▼リスト 3.12: 文字の取得へ戻る (ループ処理)

```
sim> d -m 412 jmp 402
sim>
```

これで「AC レジスタにアドレスで指定されている文字列を出力する」関数が完成しました。実装した命令列を改めて表示するとリスト 3.13 の通りです。

▼リスト 3.13: 指定された文字列を出力する関数

```
sim> e -m 401-412
401:    DAC 460
402:    LAC I 460
403:    SZA
404:    JMP 406
```

[*6] 8 進数は私もあまり見慣れないので、一瞬「アドレスが飛んでる？」のかと思ってしまったり、あるいは 0o408 という 8 進数では存在しない値を書いてしまったりしますので・・、一応の補足です。

```
405:    JMP I 400
406:    JMS 300
407:    LAC 460
410:    ADD 461
411:    DAC 460
412:    JMP 402
sim>
```

3.4 「HELLO WORLD!」を出力してみる

♣ 出力する文字列をメモリへ配置

それでは、前節の関数を使用した文字列出力の例として「HELLO WORLD!」という文字列を出力してみましょう。

まず、メモリ上に文字列を定義します。ここでは、アドレス 0 以降に並べていくことにします (リスト 3.14)。

▼ リスト 3.14: 出力する文字列をメモリ上に配置

```
sim> d -a 00 H
sim> d -a 01 E
sim> d -a 02 L
sim> d -a 03 L
sim> d -a 04 O
sim> d    05 40  ←スペース
sim> d -a 06 W
sim> d -a 07 O
sim> d -a 10 R
sim> d -a 11 L
sim> d -a 12 D
sim> d -a 13 !
sim> d    14 0   ← NULL文字
sim>
```

`deposit` コマンドの `-a` で ASCII の文字データを 1 文字ずつ並べていきます。なお、`-a` では、配置するアドレス後に半角スペース区切りで与えた最初の 1 文字が「配置する文字」として扱われてしまう[*7]ので、スペースや NULL 文字は `-a` を使わず、8 進数指定で配置しています。

[*7] 例えば、`d -a 5 ' '` の様にコマンドを実行した場合、アドレス 5 にはチルダ (') の ASCII コードが配置されてしまいます。

deposit -a で設定される謎のビット

リスト 3.14 でメモリへ配置した文字について、例えばアドレス 0 に配置した'H' を `e 0` コマンドで見てみると `000310` (8 進数) という値が設定されていることが分かります。ASCII の'H' は 8 進数で 0o110 なので、ビット 7 に謎の 1 が設定されています。この様に、 `deposit` コマンドの `-a` の switch で ASCII 文字をメモリへ設定すると何故かビット 7 に 1 が設定されます。

ただ、試してみると分かりますが、このビットが設定されていても `TLS` 命令での文字出力に害はないため、本書では特に気にしないことにします。

♣ 動作確認

では、動作確認してみます。例によって、関数呼び出しして戻ってきたら停止するだけの処理を用意し、AC レジスタに文字列のアドレス (0) を設定した上で、実行します (リスト 3.15)。

▼リスト 3.15: 文字列出力関数の動作確認

```
sim> d -m 100 jms 400  ←アドレス0o400の関数呼び出し
sim> d -m 101 hlt      ←関数から戻ったら停止
sim> d ac 0            ←文字列のアドレスをACレジスタへ設定
sim> go 100            ←実行開始
HELLO WORLD!
HALT instruction, PC: 00102 (CAL 0)
sim>
```

PDP-7 で「HELLO WORLD!」ができました！

3.5　入力された 1 文字を取得する関数を作る

♣ テレタイプで 1 文字取得する (KRB 命令)

続いて、テレタイプでキーボード入力の取得を行ってみます。使用する命令は `KRB` 命令です (表 3.7)。

`KRB` 命令を使用すると、キーボードで入力された文字が ASCII コードで格納されているバッファから AC レジスタへ 1 文字取得されます。

実験してみましょう。ここでは、無限ループで `KRB` 命令を繰り返し実行している最中に何らかのキー入力を行い、Ctrl+e で抜けた後、 `deposit` コマンドで AC レジスタを見てみます (リスト 3.16)。

▼表 3.7: PDP-7 命令紹介: KRB

アセンブリ表記	機能
KRB	テレタイプの入力用バッファから 1 文字読み出して AC レジスタの下位 8 ビットへ設定

▼リスト 3.16: KRB 命令の実験

```
sim> d -m 100 krb
sim> d -m 101 jmp 100  ← KRB命令を無限に繰り返す
sim> e ac
AC:     000000         ← ACレジスタの値を事前に確認
sim> go 100            ←実行開始
A                      ← aキーを押下
Simulation stopped, PC: 00100 (KRB)  ← Ctrl+eで抜ける
sim> e ac
AC:     000301         ←'A'が取得された
sim>
```

入力した文字が AC レジスタに取得される様子を確認できました。なお、小文字の入力であっても大文字で表示/取得されるのは PDP-7 のテレタイプが大文字のみであるためであるようです。

また、取得された値に関しても、`deposit -a` で文字を配置した場合と同様に、ビット 7 に謎の 1 が設定されていますが、特に動作に影響はないので本書では気にしないことにします。

♣ 入力フラグを待ってから取得するようにする

入力バッファにも出力同様に入力バッファへの格納が完了したことを意味するフラグがあります。このフラグの状態によって分岐する命令が `KSF` 命令です (表 3.8)。なお、このフラグをクリアする専用の命令は特にありません。入力フラグは `KRB` 命令の実行でクリアされます。

▼表 3.8: PDP-7 命令紹介: KSF

アセンブリ表記	機能
KSF	文字入力完了フラグがセットされている場合、次の命令をスキップする

それでは、この命令を使用して、入力完了フラグを待ってから文字の取得を行うように実装してみます (リスト 3.17)。なお、今回はアドレス 0o501 以降に実装することにします。(例によってこれを後ほど関数化します。)

▼リスト 3.17: 入力フラグを待ってから文字を取得

```
sim> d -m 501 ksf       ←入力フラグがセットされていたら次の命令をスキップ
sim> d -m 502 jmp 501   ←再度KSF命令へ(入力を待つ)
sim> d -m 503 krb       ←入力バッファから文字をACレジスタへ取得
sim>
```

何らかのキー入力があるまでアドレス 0o501 と 0o502 の間をループします。キー入力があり、入力完了フラグがセットされるとアドレス 0o503 以降が実行されるという流れです。

♣ 関数化する

最後に return のための `JMP I` 命令を追加すると、アドレス 0o500 で呼び出し可能な関数となります (リスト 3.18)。

▼リスト 3.18: 1 文字取得する関数化する

```
sim> d -m 504 jmp i 500  ← return
sim> e -m 501-504        ←関数の命令列を表示
501:    KSF
502:    JMP 501
503:    KRB
504:    JMP I 500
sim>
```

♣ 動作確認

一応動作確認しておきましょう。例によって関数を呼び出して停止するだけの処理を 0o100 に配置して実行してみます (リスト 3.19)。

▼リスト 3.19: 1 文字取得する関数の動作確認

```
sim> d -m 100 jms 500  ←アドレス0o500の関数呼び出し
sim> d -m 101 hlt      ←関数から戻ったら停止
sim> e ac
AC:     000000         ←実行前はACレジスタは0
sim> go 100            ←実行開始
B                      ←何らかのキー入力
HALT instruction, PC: 00102 (JMP 100)
sim> e ac
AC:     000302         ←'B'が取得された
sim>
```

3.6　エコーバックプログラムを作って動作確認

♣ テレタイプ入力に全二重設定を行う

「エコーバック」とは、「入力した文字をそのまま出力する」ことです。文字の入出力を扱うデバイスの動作確認のプログラムとしてよく用いられます。[*8]

ただ、前節までの動作確認の際、特に何もプログラムせずとも、入力した文字が画面に出力されてしまっていました。これは SimH 上のテレタイプのデバイス設定によるものなのですが、これではエコーバックの動作確認がしづらい[*9]ため、デバイスの設定を変更します。

SimH がシミュレートしているデバイスは `show devices` (省略形 `sh dev`) というコマンドで一覧できます (リスト 3.20)。

▼リスト 3.20: SimH がシミュレートしているデバイス一覧

```
sim> show devices
PDP-7 simulator configuration

CPU     idle disabled
CLK     60Hz, devno=00
PTR     devno=01
PTP     devno=02
TTI     devno=03  ← TeleType Input
TTO     devno=04  ← TeleType Output
LPT     devno=65-66
DRM     disabled
RB      disabled
DT      devno=75-76, 8 units
G2OUT   devno=05
G2IN    devno=43-44
DPY     disabled
sim>
```

テレタイプのデバイスは入力と出力の 2 つあります。TTI が入力側、TTO が出力側です。

設定を変更するのは TTI です。TTI の設定で `FDX` (Full DupleX、全二重) を有効化すると、入力した文字を画面へ出力しなくなります。元に戻す場合は `HDX` (Half DupleX、半二重) を有効化すると、これまで通り入力した文字を画面へ出力するようになります。

*8 マイコンのシリアル (UART) 通信の動作確認等でよく用いられるかと思います。

*9 「取得した文字をそのまま出力する」プログラムを動かすと、入力した文字に対して「勝手に行われる出力」と「自作のプログラムの出力」で二重に文字が出力されてしまいます。

設定の変更は `set` コマンドで行います。設定変更と動作確認の様子はリスト 3.21 の通りです。

▼リスト 3.21: TTI の FDX/HDX の動作確認

```
sim> d -m 100 jmp 100  ←確認のための無限ループ
sim> set tti fdx       ←全二重を設定
sim> go 100            ←無限ループを実行
                       ←キー入力しても出力されなくなった
Simulation stopped, PC: 00100 (JMP 100)  ← Ctrl+eで抜ける
sim> set tti hdx       ←半二重へ戻す
sim> go 100            ←無限ループを実行
HOGEPIYOFUGA           ←入力した文字が出力される状態に戻った
Simulation stopped, PC: 00100 (JMP 100)  ← Ctrl+eで抜ける
sim> set tti fdx       ←今後のために全二重設定にしておく
sim>
```

♣ 1 文字取得と出力を繰り返してエコーバック

ここまでで、「入力された 1 文字を取得する関数 (アドレス 0o500)」と「指定された 1 文字を出力する関数 (アドレス 0o300)」を作成しました。共に AC レジスタを使用するので、単にこれを繰り返し実行するだけでエコーバックが行えます。実装してみましょう (リスト 3.22)。

▼リスト 3.22: 文字の取得と出力を繰り返す

```
sim> d -m 100 jms 500  ←キー入力を待ち、ACへ1文字取得
sim> d -m 101 jms 300  ← ACの文字を出力
sim> d -m 102 jmp 100  ←最初へ戻る
sim>
```

♣ 動作確認 (ただし・・)

それでは、動作確認してみましょう (リスト 3.23)。

▼リスト 3.23: 動作確認・・ただし・・

```
sim> go 100        ←実行開始
HOGEPIYOABCDEF     ←入力した文字が出力されている
Simulation stopped, PC: 00501 (KSF)  ← Ctrl+eで一旦抜ける
sim> go 100        ←もう一度実行開始
ABCEHOGE    ← Enter押下で同じ行の行頭へ行ってしまう
```

このプログラムは終了しないので、終了させる際は Ctrl+e で抜けてください。

最初の実行では、キーボードで全て小文字で「hogepiyoabcdef」と入力しました。大文字になるのは仕様ですので、期待通りに動作しています。

2 度目の実行が問題で、「hogehoge」と入力した後、Enter を押下し、「abc」と入力してみました。その際、Enter によってカーソルは行頭に移動しましたが、次の行へ移動しなかったため、同じ行のまま出力されてしまいました。

♣ Enter 押下で何を取得しているか確認

これは、ASCII の制御文字で言うと、「CR(Carriage Return、行頭復帰) は出力されたが、LF(Line Feed、改行) が出力されなかった」という状態です。おそらく Enter の押下で取得した制御文字が CR で、それを出力しただけなので Enter 押下時の挙動が CR(行頭復帰) のみになってしまったのかと思われます。

一応確認してみましょう。先程作成したプログラム (リスト 3.22) で、アドレス 0o102 の `JMP` 命令を無限ループするものへ変更し、Enter 押下時に AC レジスタに何が格納されているのかを見てみます (リスト 3.24)。

▼リスト 3.24: Enter 押下時の AC レジスタを確認

```
sim> d -m 102 jmp 102  ←無限ループへ変更
sim> go 100            ←実行開始
                       ← Enterを押下
Simulation stopped, PC: 00102 (JMP 102)  ← Ctrl+eで抜ける
sim> e ac
AC:     000215         ← ACにはCRが格納されていた
sim>
```

想定通り、Enter 押下時に AC レジスタには CR(0o15) が設定されていました。そのため、対処としては出力した文字が CR である場合は、追加で LF も出力するようにすれば良さそうです。

もし Enter の挙動が異なった場合は適宜読み替えてください

この辺りの挙動は SimH のテレタイプ実装と OS のシェルの挙動の問題であるため、環境によってはここで紹介している挙動とは異なるかもしれません。そのような場合は後述の説明を適宜読み替えてください。

例えば、もし「次の行へ進む (LF) が行頭復帰 (CR) されない」という挙動であった場合は、次項以降では「取得した文字が LF の場合、追加で CR も出力する」といったように読み替えてください。

♣ AC の値に応じて分岐させる

まずは、1 文字出力する関数から戻ってきた後、AC レジスタの値が CR か否かで分岐するようにします。使用する命令は `SAD` 命令です (表 3.9)。

▼表 3.9: PDP-7 命令紹介: SAD

アセンブリ表記	機能
SAD <アドレス>	AC レジスタの値がアドレス先の値と等しくない場合、次の命令をスキップする

前項の確認で、Enter が押下されていた場合、1 文字出力する関数から戻ってきた後 (アドレス 0o102 時点) の AC レジスタは 0o215 である事が分かっています。そのため、アドレス 0o102 で `SAD` 命令を使用して、AC レジスタを 0o215 と比較することにします。なお、`SAD` 命令にはアドレスを渡す必要があるので、定数 0o215 を適当なアドレスとして 0o160 にでも配置しておくことにします。

そして、AC レジスタと 0o215 を比較した結果、等しければ LF を出力する処理へジャンプし、等しくなければ最初 (文字入力) へ戻るようにします。

以上を踏まえて `SAD` 命令を用いた分岐処理を実装するとリスト 3.25 の通りです。

▼リスト 3.25: SAD 命令を用いた分岐処理

```
sim> d 160 215          ←アドレス0o160へCR(+0o200)を配置
sim> d -m 102 sad 160   ← AC != CR(アドレス0o160)なら 、次の命令をスキップ
sim> d -m 103 jmp 105   ← (AC == CRの場合)LF出力処理(アドレス0o105)へジャンプ
sim> d -m 104 jmp 100   ← (AC != CRの場合)最初へ戻る
sim>
```

♣ LF 出力処理を実装

AC レジスタが ASCII の CR と等しい場合に実行する LF の出力処理を実装します。といっても、やることは AC レジスタへ ASCII 制御文字 LF(0o12) を設定して文字出力関数 (アドレス 0o300) を呼び出すだけです。

AC レジスタへ定数 0o12 を設定するにあたって、適当なアドレスにこの値を配置して `LAC` 命令でロードさせてもよいのですが、ここでは新しい命令を使ってみることにします。使用する命令は `LAW` 命令です (表 3.10)。

▼表 3.10: PDP-7 命令紹介: LAW

アセンブリ表記	機能
LAW <値 (12 ビット)>	指定された値を AC レジスタの下位 12 ビットへ設定する 上位 6 ビットには 0o76 が設定される

`LAW` 命令は指定した値を AC レジスタへ設定してくれる命令なのですが、挙動が少々特殊なので、実験で説明します (リスト 3.26)。

▼リスト 3.26: LAW 命令の実験

```
sim> d -m 0 law 12  ←実験のLAW命令(0o12をロードさせてみる)
sim> d -m 1 hlt     ← LAW命令が終わったら停止
sim> go 0           ←実行開始

HALT instruction, PC: 00002 (CAL 0)
sim> e ac
AC:     760012      ←上位6ビットに謎の0o76がある
sim> e -m 0         ←アドレス0に置いた命令を確認(アセンブリ表記)
0:      LAW 12
sim> e 0            ←アドレス0に置いた命令を確認(機械語表記)
0:      760012      ←機械語をそのままACレジスタに設定する挙動である模様
sim>
```

リスト 3.26 の結果から、 LAW 命令は単にその機械語命令自体をそのまま AC レジスタへ設定している事がわかります。この命令を使用すると上位 6 ビットに無駄な 0o76 が設定されてしまう事になりますが、文字出力の際、 TLS 命令は AC レジスタの下位 8 ビットしか見ないので問題ありません。

それでは、 LAW 命令を使用して、アドレス 0o105 以降に LF 出力処理を実装してみます (リスト 3.27)。

▼リスト 3.27: LF 出力処理を実装

```
sim> d -m 105 law 12   ← ACへLFを設定
sim> d -m 106 jms 300  ← ACの文字を出力
sim>
```

これで LF を出力する処理ができました。

最後に、LF を出力した後 (アドレス 0o107) にも最初へ戻る JMP 命令を追加すれば、エコーバックを行う処理の完成です (リスト 3.28)。

▼リスト 3.28: エコーバックの処理完成

```
sim> d -m 107 jmp 100  ← LF出力後、最初へ戻る
sim> e -m 100-107      ←実装した内容を確認
100:    JMS 500
101:    JMS 300
102:    SAD 160
103:    JMP 105
104:    JMP 100
105:    LAW 12
106:    JMS 300
107:    JMP 100
sim>
```

♣ 改めて動作確認

では、改めて動作確認してみましょう (リスト 3.29)。

▼ リスト 3.29: 改めてエコーバックの動作確認

```
sim> go 100  ←実行開始
HOGEHOGE
ABC          ← Enter押下で次の行へ移動した！
```

今回も「hogehoge」と入力した後、Enter を押下し、「abc」と入力してみた所、今度はちゃんと Enter キー押下で次の行へ移動しました！

これでエコーバックプログラムは完成です。なお、このプログラムには `HLT` 命令を入れていないので、例によって終了する際は Ctrl+e で抜けてください。

第 4 章

ベクタースキャンディスプレイとライトペン (図形描画/ヒット検出)

この章では、ベクタースキャンディスプレイとライトペンの使い方を紹介し、ベクタースキャンディスプレイへの簡単な図形の描画とライトペンによる図形のヒット期間の検出を行います。

なお、この章の内容を実装した simh スクリプトはサンプルリポジトリ内の 04-square-hit-msg.simh です。

4.1 ベクタースキャンディスプレイとライトペンとは

ベクタースキャンディスプレイを紹介するにあたって、まず現代の一般的なディスプレイは、「ラスタスキャン」と呼ばれる方式で、描画する画像を画素あるいはピクセル等と呼ばれる点に分解し、左上から右下へ走査 (スキャン) を行ってディスプレイ上に点を並べることで画像を描画します。対して、ベクタースキャンディスプレイは、任意の座標に線や点を描画する方式のディスプレイです。そのため、ディスプレイのコントローラへは専用の命令等で座標データを与えることになります。PDP-7 では「Type 340」というベクタースキャンディスプレイが使用されていたようです。YouTube で「PDP-7 Type 340」といったキーワードで検索すると、Type 340 を PDP-7 で動かしている様子が動画で公開されていたりしますので、ぜひ見てみてください。[*1]なお、解像度は 1024x1024 で、座標系の原点は左下です。描画機能として

[*1] 例えば、こちらの動画などです: 「DEC PDP-7 w/ Type 340 display running Munching

持っているのは点と直線を描く機能のみで、曲線を描く機能はありません。また、色も黒背景に白 (光による点あるいは線) のみで、輝度のパラメータ設定ができます。

次に、ライトペンは当時のポインティングデバイスです。光センサを内蔵しておりディスプレイに押し当ててディスプレイから発する光を受光するというものです。ベクタースキャンで描画している箇所は描画による光を発しているためそれを受光することで「描画した図形にヒットしているか否か」を検出します。

4.2　SimH 同梱のテストスクリプトで描画方法を紹介

Open-SimH の GitHub リポジトリにはベクタースキャンディスプレイへ描画を行うテスト用の simh スクリプトが同梱されています。この節ではこのテストスクリプトを用いて Type 340 というベクタースキャンディスプレイへの描画方法を紹介します。

♣ テストスクリプトを動かしてみる

まずはテストスクリプトを動かして、SimH 上でシミュレートしているベクタースキャンディスプレイへの描画を見てみましょう。

ベクタースキャンディスプレイのテストスクリプトは、Open-SimH の GitHub リポジトリ内の PDP18B/tests/test340.simh にあります。Open-SimH の GitHub リポジトリを clone あるいはダウンロードした場所へ移動して、この simh スクリプトを実行してみましょう (リスト 4.1)。

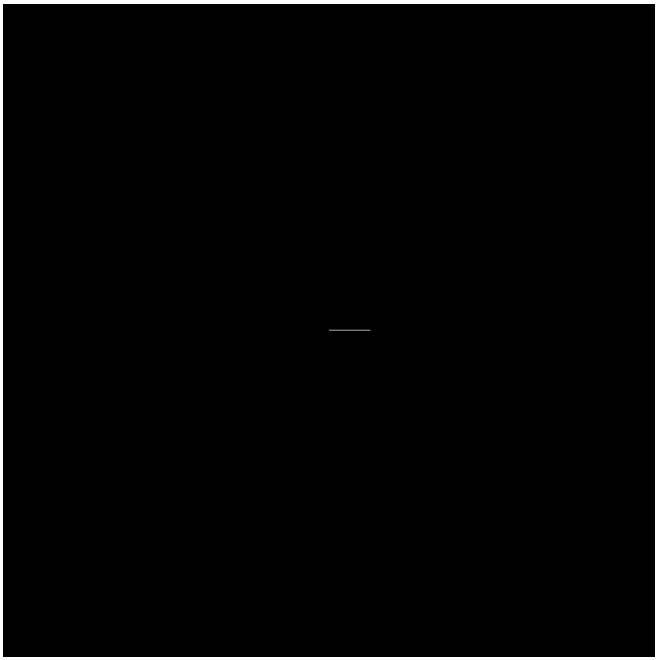
▼リスト 4.1: ベクタースキャンディスプレイのテストスクリプトを動かす

```
$ cd <Open-SimHをclone/ダウンロードしたパス>
$ pdp7 PDP18B/tests/test340.simh  ←スクリプトを起動

PDP-7 simulator V4.0-0 Current        git commit id: 09899c18
<Open-SimHをclone/ダウンロードしたパス>/PDP18B/tests/test340.simh-3> set debu›
›g stdout
%SIM-INFO: Debug output to "STDOUT"
Debug output to "STDOUT" at Sun Apr  9 14:44:13 2023
PDP-7 simulator V4.0-0 Current        git commit id: 09899c18
sim> go                             ←実行開始
DBG(2)> DPY IOT: 700606, 761000
DBG(3)> DPY IOT: 700601, 761000
DBG(2003)> same as above (1000 times)
```

Squares and Spirograph」https://youtu.be/V4oRHv-Svwc

```
DBG(2006)> DPY IOT: 700606, 761000
 ・・・省略・・・

Simulation stopped, PC: 00102 (IDSI)  ← Ctrl+eで抜ける
sim> by                               ← SimHを終了
Goodbye
%SIM-INFO: Debug output disabled
$
```

スクリプトを起動すると真っ黒なウィンドウが表示されます。このスクリプトはプログラムをメモリ上に配置した後、PC に開始アドレスを設定するだけなので、`go` コマンドを手動で実行して開始させます。すると、中央より少し右辺りに短い線が描画されます (図 4.1)。



▲図 4.1: test340.simh でウィンドウに描画された様子

test340.simh の内容はリスト 4.2 の通りです。次項からこの内容を解説していきます。

▼リスト 4.2: test340.simh

```
set g2out disabled

set debug stdout
set dpy enabled
set dpy debug

# Small test program.
dep -m 100 law 1000
dep -m 101 idla
dep -m 102 idsi
dep -m 103 jmp 102
dep -m 104 jmp 100

# Small display list.
#Go to point mode, set scale, set intensity.
dep 1000 020117
#Stay in point mode, set x=1000.
dep 1001 021000
#Go to vector mode, set y=1000.
dep 1002 301000
#Escape, intensify, delta x=100.
dep 1003 600100
#Stop.
dep 1004 003000

dep pc 100
# Ready to go or single step.
```

♣ スクリプト解説: デバイス等の設定

まず、冒頭で `set` コマンドを用いてデバイス等の設定を行っています (リスト 4.3)。

▼リスト 4.3: デバイス等の設定

```
set g2out disabled

set debug stdout
set dpy enabled
set dpy debug
```

`set g2out disabled` では、「GRAPHICS-2」というハードウェアのディスプレイのシミュレーションを無効化しています。GRAPHICS-2 というものもベクタースキャンで描画が行えるハードウェアだったようで、Open-SimH のコード内のコメ

ントによると UNIX V7 では TTY に使用していたとのことです。[*2]それ故なのか、少なくとも Open-SimH の#09899c18 コミット時点ではテキスト表示のみに対応しているとのことです。この GRAPHICS-2 のシミュレーションが内部的にベクタースキャンディスプレイのシミュレーションとコンフリクトするため、予め無効にしています。[*3]

`set debug stdout` を行うと、SimH のデバッグ情報の出力が有効化されると共にその出力先が標準出力へ設定されます。デバッグ情報の出力の無効化や標準出力以外の出力先への設定等、詳しくは `help set debug` をご覧ください。

`set dpy enabled` で、ベクタースキャンディスプレイのシミュレーションが有効化され、ベクタースキャンディスプレイの出力のためのウィンドウが開きます。この時点ではウィンドウ内は真っ黒です。なお、 `dpy` というのが SimH 内でのベクタースキャンディスプレイのデバイス名です。

最後に `set dpy debug` でベクタースキャンディスプレイのデバッグ出力を有効化しています。

デバッグ出力が不要であれば、 `set g2out disabled` と `set dpy enabled` だけで構いません。[*4]

♣ スクリプト解説: ディスプレイの設定と開始のプログラム

続いて、スクリプトのリスト 4.4 の部分についてです。

▼ リスト 4.4: ディスプレイの設定と開始のプログラム

```
# Small test program.
dep -m 100 law 1000
dep -m 101 idla
dep -m 102 idsi
dep -m 103 jmp 102
dep -m 104 jmp 100
```

`dep` は `deposit` コマンドの省略形です。最低限 `d` の 1 文字があれば認識するので、 `dep` でも OK です。ここでは計 5 回の `deposit` コマンドでアドレス 0o100 から 0o104 に 5 つの命令を配置しています。

まず `law 1000` について、 `LAW` 命令は第 3 章で紹介した命令で、指定された値を AC レジスタの下位 12 ビットへ設定します。(そして上位 6 ビットには 0o76 が設

*2 PDP18B/pdp18b_g2tty.c の 96 行目以降のコメント箇所です。

*3 試しに `set g2out disabled` を行わずに `set dpy enabled` を行うと、「DPY device number conflict at 05」というメッセージと共に失敗します。

*4 サンプルの simh スクリプトではこの 2 つのみを行っています。

定されます。) ここでは、AC レジスタの下位 12 ビットに 0o1000 を設定しています。

続いて、 `IDLA` 命令は新たに登場した命令で表 4.1 の通りです。

▼表 4.1: PDP-7 命令紹介: IDLA

アセンブリ表記	機能
IDLA	AC レジスタの下位 12 ビットを描画処理の開始アドレスとして設定し、描画処理を開始させる

Type 340 には専用命令があります。 `IDLA` は、その専用命令を並べた領域の開始アドレスを Type 340 へ設定し、Type 340 の処理を開始させる命令です。先程、AC レジスタの下位 12 ビットには 0o1000 を設定しましたので、このアドレスを Type 340 へ開始アドレスとして設定し、そこから Type 340 は描画処理を開始することになります。アドレス 0o1000 以降に専用命令を並べるための `deposit` コマンドはこれより少し後の行にありますので後ほど紹介します。

`IDSI` も今回新たに登場した命令です (表 4.2)。

▼表 4.2: PDP-7 命令紹介: IDSI

アセンブリ表記	機能
IDSI	Type 340 の STOP 割り込みが設定されていたら次の命令をスキップする

先程の `IDLA` 命令により Type 340 側で処理が始まっています。これ以降の処理はその停止を待ち受ける処理です。後ほど紹介しますが、Type 340 の専用命令を用いた処理は最終的に「停止 」のパラメータを設定して終わりとなります。その際に STOP 割り込みも設定します。 `IDSI` 命令は STOP 割り込みが設定されているか否かで分岐する命令です。なお、少なくとも SimH の実装において、Type 340 の割り込みは「何らかの割り込みが設定されている間、Type 340 の命令実行を止める」という挙動です。[*5]

続くアドレス 0o103 に、アドレス 0o102(`IDSI` 命令) へのジャンプがあることから、Type 340 の処理が終了するまで `IDSI` 命令を繰り返す、という訳です。

[*5] display/type340.c の 158 行目に実装がある `ty340_cycle` 関数が「SimH の周期処理の中から呼び出されて Type 340 命令を一つ実行する」関数です。この関数を繰り返し呼び出すことで Type 340 の命令列が一つずつ実行されます。なお、この関数は `u->status` が 0 でない場合は何もしないようになっています。 `u->status` 変数は STOP 割り込み等の何らかの割り込みがあった場合に設定される変数です。そのため、Type 340 の何からの割り込みが設定されている場合、Type 340 の命令実行が止まります。

そして、Type 340 の処理が終了し、アドレス 0o103 の命令がスキップされた場合、アドレス 0o104 の `JMP` 命令により最初に戻ります。Type 340 は処理を終了すると描画した内容も消えてしまうので、このように繰り返すことで描画内容を表示させ続けています。

♣ スクリプト解説: Type 340 命令 (パラメータモード)

続いて「Small display list.」のコメント行以降を解説します。まずはリスト 4.5 の部分です。

▼リスト 4.5: パラメータモードで初期設定

```
# Small display list.
#Go to point mode, set scale, set intensity.
dep 1000 020117
```

Type 340 命令は `deposit` コマンドの `-m` が対応していないため、命令の機械語を直接書いています。この機械語を理解する必要があるため、Type 340 命令のフォーマットを説明します。

Type 340 命令のフォーマットはモード別に存在します。 `IDLA` 命令によって開始してから最初に実行する命令は「パラメータモード」というモードで実行する事が決まっています。そのため、アドレス 0o1000 に配置している `020117` (8 進数) という機械語もパラメータモードで解釈されることになりますので、まずはこのモードのフォーマットを説明します (表 4.3)。[*6]

ビット 15-13(2-4) へ設定するモード番号について、基本的に Type 340 の命令は次

[*6] なお、パラメータモードの命令のパースは display/type340.c の 873 行目から 904 行目辺りで行っています。もし興味があれば見てみてください。

[*7] SimH のコードでは、PDP-7 における「左端が 0」のビット番号付けでビットフィールドの操作が行えるようにいくつかのマクロ関数を用意して使用しています。display/type340.c の 81 行目から 90 行目に定義されている `GETFIELD()` や `TESTBIT()` といったマクロ関数がそうです。Type 340 命令のパースを行っている箇所ではこれらのマクロ関数を使用して「左端が 0」のビット番号で記述しているため、コードを読む際の補助として命令フォーマット紹介の表に「左端が 0」のビット番号も併記しています。

[*8] このビットのパース処理は display/type340.c で行っていて、ビット 12(5) が 1 の時、Type 340 用の構造体 (`struct type340`) の `lp_ena` というメンバー変数にビット 11(6) を設定します。そして、この `lp_ena` 変数が使われているのは実質 1 箇所だけで、それは、 `lp_ena` が 1 の時、ライトペンの描画箇所とのヒット検出時に `ty340_lp_int` という関数を呼び出す、という処理です (同ファイルの 259 行目から 261 行目)。ただ、この `ty340_lp_int` 関数は中身が空あるいは `printf()` するだけ (make 時の設定によりどちらかが選ばれる) なので、パラメータモードのビット 12,11(5,6) は特に意味はないという訳です。

▼表 4.3: Type 340 命令フォーマット紹介: パラメータモード

ビット番号 (右端が 0)	ビット番号 (左端が 0)[*7]	機能
17-16	0-1	使用しない
15-13	2-4	次の命令のモード番号を設定
12,11	5,6	ビット 12(5) が 1 の時、ビット 11(6) をディスプレイ側のライトペン有効化ビットへ設定 (ただし現状の Open-SimH では特に意味はない[*8])
10	7	1 の時、描画を停止する
9	8	ビット 10(7) と共に 1 の時、STOP 割り込みを設定
8-7	9-10	使用しない
6,5-4	11,12-13	ビット 6(11) が 1 の時、1 をビット 5-4(12-13) の値だけ左シフトした値をベクトルの大きさの単位量として設定
3,2-0	14,15-17	ビット 3(14) が 1 の時、ビット 2-0(15-17) の値を輝度として設定

の命令をどのモードで解釈するかを毎回設定します。[*9]モード番号とモード名の対応は表 4.4 の通りです。[*10]色々とモードがありますが、図形を描いたりする上ではこれらの中の一部のモードで十分です。(本書でも全てを紹介するわけではありません。)

▼表 4.4: モード番号とモード名の対応

モード番号 (カッコ内は 2 進数)	モード名
0(000)	パラメータモード
1(001)	ポイントモード
2(010)	スレーブモード
3(011)	キャラクタモード
4(100)	ベクタモード
5(101)	ベクタコンティニューモード
6(110)	インクリメントモード
7(111)	サブルーチンモード

また、ビット 6,5-4(11,12-13) で設定する「ベクトルの大きさの単位量」について、後ほど紹介しますが、PDP-7 では始点とそこからの X,Y 方向のベクトル (向きを持った大きさ) を指定することで線を描きます。ここで設定する単位量がベクトルの

[*9] 一部のモードではそうでないものもありますが。

[*10] コードとしては、display/type340.c の 98 行目に enum で定義されています。

X 成分・Y 成分に掛け算されます。

テストスクリプトに戻ると、アドレス 0o1000 へ `020117` (8 進数) という命令を配置していました。これをパラメータモードで解釈すると表 4.5 の通りです。

▼表 4.5: 0o020117 をパラメータモードで解釈すると

ビット番号 (右端が 0)	0o020117 の該当ビット	意味
17-16	00	使用しない
15-13	001	次の命令はポイントモードで解釈される
12,11	0,0	ビット 12 が 0 なので何もしない
10	0	0 なので何もしない
9	0	0 なので何もしない
8-7	00	使用しない
6,5-4	1,00	ビット 6 が 1 なので、1 を 0 回左シフトした値、即ち 1 をベクトルの大きさの単位量として設定
3,2-0	1,111	ビット 3 が 1 なので、ビット 2-0 の値を用いて輝度として 7(最大) を設定

まとめると、`020117` の命令では、ベクトルの大きさの単位量として 1 を、輝度として最大の 7 を設定し、次の命令のモードはポイントモードであると設定しています。

♣ スクリプト解説: Type 340 命令 (ポイントモード)

続いてはスクリプトのリスト 4.6 の部分です。

▼リスト 4.6: ポイントモードで座標設定

```
#Stay in point mode, set x=1000.
dep 1001 021000
#Go to vector mode, set y=1000.
dep 1002 301000
```

0o1000 の命令で次の命令を解釈するモードとしてポイントモードを指定していたため、0o1001 の命令はポイントモードで解釈されます。なお、先に説明してしまうと、「#Stay in point mode」というコメントからもわかる通り、0o1001 の命令では次のモードとしてポイントモードを指定したまま、0o1002 の命令もポイントモードとして解釈されます。

ポイントモードのフォーマットは表 4.6 の通りです。[*11]

[*11] ポイントモードのパースは display/type340.c の 906 行目から 926 行目で行っています。

▼表 4.6: Type 340 命令フォーマット紹介: ポイントモード

ビット番号 (右端が 0)	ビット番号 (左端が 0)	機能
17	0	使用しない
16	1	0 の時 X 座標、1 の時 Y 座標として ビット 9-0(8-17) の値を設定
15-13	2-4	次の命令のモード番号を設定
12,11	5,6	ビット 12(5) が 1 の時、ビット 11(6) をディスプレイ側の ライトペン有効化ビットへ設定 (ただし現状の Open-SimH では特に意味はない)
10	7	この座標自体を点で描画する (=1) か否 (=0) か
9-0	8-17	座標値

ポイントモードは 1 度の命令で X か Y いずれかの座標を設定します。そのため、X と Y で 1 度ずつ計 2 回の命令で座標の指定が完了します。

ビット 10(7) への 1 の設定は座標確定時に行うこと

なお、ビット 10(7) に 1 を設定する際は、座標が確定する 2 回目の命令で行う必要があります。というのも、SimH の実装上、このビットが設定されていたらその時点で内部で保持している X 座標・Y 座標の場所にドットを描いてしまうためです。[*12]

ちなみに、現物のディスプレイではビット 10(7) により光の点が周囲のにじみと共に描画されたのではないかと思うのですが、現状のシミュレータでは本当に 1 ドットだけ点が描画されるだけです。

以上がポイントモードの紹介です。ビットの対応を表で示すのは省略しますが、アドレス 0o1001 と 0o1002 に配置していた `021000` と `301000` という命令はそれぞれ、X 座標に 0o1000 を、Y 座標にも 0o1000 を設定しています。いずれもビット 10(7) は 0 なので、この座標自体に点の描画はされません。

そして 1 つ目では次のモード番号として 1(ポイントモード) を設定することでポイントモードを継続し、2 つ目で 4(ベクタモード) を設定していることから、次の命令はベクタモードで解釈されることになります。

[*12] コードとしては display/type340.c の 922 行目から 925 行目です。

♣ スクリプト解説: Type 340 命令 (ベクタモード)

続いてスクリプトのリスト 4.7 の部分です。

▼リスト 4.7: ベクタモードで直線を描画

```
#Escape, intensify, delta x=100.
dep 1003 600100
```

一つ前の命令で指定していた通り、この命令はベクタモードで解釈されます。ベクタモードのフォーマットは表 4.7 の通りです。[*13]

▼表 4.7: Type 340 命令フォーマット紹介: ベクタモード

ビット番号 (右端が 0)	ビット番号 (左端が 0)	機能
17	0	0 の時、次の命令も引き続きベクタモードで解釈する 1 の時、パラメータモードに戻る
16	1	0 の時、線の描画は行われず座標移動のみ 1 の時、線の描画と座標移動が行われる
15	2	0 の時、現在の Y 座標に (dy * 単位量) を足す (上移動) 1 の時、現在の Y 座標から (dy * 単位量) を引く (下移動)
14-8	3-9	dy
7	10	0 の時、現在の X 座標に (dx * 単位量) を足す (右移動) 1 の時、現在の X 座標から (dx * 単位量) を引く (左移動)
6-0	11-17	dx

ベクタモードは、現在の座標から X 軸・Y 軸それぞれの向きを持った大きさ (即ちベクトル) で示される地点まで座標を移動します。その際、ビット 16(1) が 1 であればその間に直線を描画します。

ビット 17(0) が 0 の場合、次の命令もベクタモードで解釈します。その際、前回のベクタモードの命令で移動した先が、次のベクタモードでの開始座標となります。

ビット 15(2) とビット 7(10) の機能の説明にある「単位量」は、パラメータモードのビット 6,5-4(11,12-13) で設定された単位量の事です。

画面範囲を超えた場合もパラメータモードへ戻るので注意

画面の範囲である X・Y 共に 0 以上 1023 以下の座標を超えて描画しようとした場合も、パラメータモードに戻ります。ベクタモードの連続で描画してい

[*13] ベクタモードのパースは display/type340.c の 951 行目から 960 行目で行っています。

> る中で不意に画面範囲から出てしまった場合、その次の命令が本来ベクタモードで解釈されるはずがパラメータモードで解釈され、意図せぬ挙動となってしまいます。画面範囲から出ないようにご注意ください。

以上を踏まえてアドレス 0o1003 に配置している `600100` を解釈すると、Y 座標の移動は無しで、X 座標には (0o100(10 進数で 64) * 単位量 1) を足す、その際に直線を描画する、という訳で現在の座標から X 軸の正方向 (即ち右方向) へ 64 ドット分の直線を描画、という意味になります。

加えて、ビット 17(0) が 1 なので、次の命令はパラメータモードで解釈されることになります。

♣ スクリプト解説: 再度パラメータモード (描画停止)

スクリプトの Type 340 命令部分の最後はリスト 4.8 です。

▼リスト 4.8: 描画を停止

```
#Stop.
dep 1004 003000
```

これはパラメータモードで解釈されるので、フォーマットは先述した表 4.3 の通りです。

パラメータモードのフォーマットに従って `003000` を解釈すると、ビット 10(7) とビット 9(8) が共に 1 なので、描画を停止し STOP 割り込みを設定、という意味になります。

以上が、テストスクリプト内の Type 340 命令部分の解説でした。

♣ スクリプト解説: さいごに

テストスクリプトの最後はリスト 4.9 の部分です。

▼リスト 4.9: PC の設定

```
dep pc 100
# Ready to go or single step.
```

PC にディスプレイの設定と開始のプログラムを置いていたアドレス 0o100 を設定しています。この後、 `go` コマンド等で実行を開始することで、アドレス 0o1000 以降に配置した Type 340 命令が繰り返し実行されます。

以上が、SimH 同梱のテストスクリプトを用いた Type 340 の描画方法の紹介でした。

4.3　ベクタースキャンディスプレイで図形描画

それでは、前節で紹介した内容を踏まえて簡単な図形を描画してみましょう。ここでは、前節で紹介したテストスクリプトを元に正方形を描画する simh スクリプトを作成してみます。

♣ ベース部分を作成

説明の都合上、前節で紹介したテストスクリプトをリスト 4.10 の様に書き直します。

▼リスト 4.10: 説明の都合上、テストスクリプトを書き直す

```
# [メイン処理]
# 0o1000をACレジスタの下位12ビットへ設定
d -m 100 law 1000
# ACレジスタの下位12ビットを描画処理の開始アドレスとして設定し、描画処理を開始
d -m 101 idla
# STOP割り込みが設定されていたら次の命令をスキップ
d -m 102 idsi
# - 設定されていない場合: 描画開始直後の処理まで戻る
d -m 103 jmp 102
# - 設定されている場合: 最初へ戻る
d -m 104 jmp 100

# [Type 340処理]
# パラメータモード: 単位量:1、輝度:7(最大)
d 1000 020117
# ポイントモード: X座標:0o1000
d 1001 021000
# ポイントモード: Y座標:0o1000
d 1002 301000
# ベクタモード: 線の描画:有り、X座標:+0o100、Y座標:+0
d 1003 600100
# パラメータモード: 描画停止、STOP割り込み設定
d 1004 003000

# [実行]
# GRAPHICS-2シミュレーションの無効化
set g2out disabled
# ベクタースキャンディスプレイシミュレーションの有効化
set dpy enabled
# メイン処理を実行
go 100

# [終了]
```

```
by
```

自動で実行を開始したり、Ctrl+e で抜けるとそのまま SimH も終了するようにしています。それ以外はテストスクリプトと同等です。

以降ではこのスクリプトへ変更・追加を行っていきます。

♣ ベクタモードを継続するようにする

現在は初期座標から右方向に長さ 0o100 の直線を引くだけです。同じ長さで上方向、左方向、下方向にも直線を引くことで正方形を描画するようにしてみましょう。

まずは、アドレス 0o1003 の命令で右方向に直線を描画する際、ベクタモードを抜けないようにビット 17 を 0 にしておきます。`deposit` コマンドでアドレス 0o1003 へ `600100` を設定している行を、ビット 17 を 0 にした `200100` を設定するように変更してください (リスト 4.11)。

▼リスト 4.11: ベクタモードを継続するようにする

```
 ・・・省略・・・
# ベクタモード: 線の描画:有り、X座標:+0o100、Y座標:+0
d 1003 200100  ←変更
# パラメータモード: 描画停止、STOP割り込み設定
 ・・・省略・・・
```

♣ 上、左、下方向の直線を描画する

続いて、アドレス 0o1004 以降に上方向、左方向、下方向へ直線を描画するベクタモードの命令を追加します。

例として上方向に長さ 0o100 の直線を描画する命令 `240000` について、ベクタモードの各ビットとの対応を示すと表 4.8 の通りです。

▼表 4.8: 上方向へ長さ 0o100 の直線を描画

ビット番号 (右端が 0)	設定値 (0o240000)	意味
17	0	次の命令も引き続きベクタモードで解釈する
16	1	線の描画と座標移動を行う
15	0	現在の Y 座標に (dy * 単位量) を足す
14-8	0o100	dy = 0o100
7	0	現在の X 座標に (dx * 単位量) を足す
6-0	0	dx = 0

同様に、左方向へ長さ 0o100 の直線を描画する命令は `200300` 、下方向へ描画する命令は `740000` となります。なお、下方向へ描画する命令でベクタモードの命令は最後なので、この命令はビット 17 に 1 を設定しています。

これらの命令をアドレス 0o1004 以降へ配置するようにします (リスト 4.12)。

▼リスト 4.12: 上、左、下方向の直線を描画するようにする

```
・・・省略・・・
d 1003 200100
# ベクタモード: 線の描画:有り、X座標:+0、Y座標:+0o100  ←追加(ここから)
d 1004 240000
# ベクタモード: 線の描画:有り、X座標:-0o100、Y座標:+0
d 1005 200300
# ベクタモード: 線の描画:有り、X座標:+0、Y座標:-0o100
d 1006 740000                                    ←追加(ここまで)
# パラメータモード: 描画停止、STOP割り込み設定
・・・省略・・・
```

♣ 描画を停止する

最後にアドレス 0o1007 へ描画停止の命令 `003000` を配置するようにしたら完成です (リスト 4.13)。

▼リスト 4.13: 描画を停止するようにする

```
・・・省略・・・
# パラメータモード: 描画停止、STOP割り込み設定
d 1007 003000  ←変更

・・・省略・・・
```

なお、最終的に「[Type 340 処理]」の箇所はリスト 4.14 の様になっています。

▼リスト 4.14: 最終的な Type 340 処理

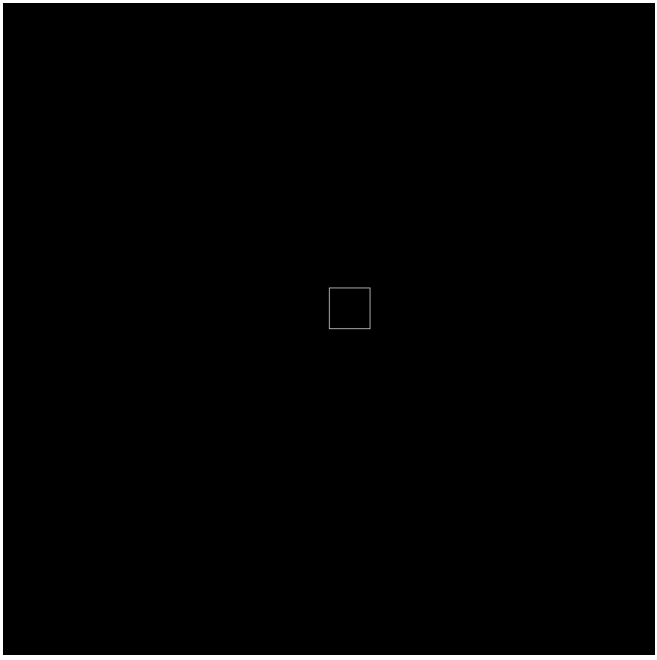
```
# [Type 340処理]
# パラメータモード: 単位量:1、輝度:7(最大)
d 1000 020117
# ポイントモード: X座標:0o1000
d 1001 021000
# ポイントモード: Y座標:0o1000
d 1002 301000
# ベクタモード: 線の描画:有り、X座標:+0o100、Y座標:+0
d 1003 200100
# ベクタモード: 線の描画:有り、X座標:+0、Y座標:+0o100
d 1004 240000
# ベクタモード: 線の描画:有り、X座標:-0o100、Y座標:+0
```

```
d 1005 200300
# ベクタモード: 線の描画:有り、X座標:+0、Y座標:-0o100
d 1006 740000
# パラメータモード: 描画停止、STOP割り込み設定
d 1007 003000
```

♣ 動作確認する

それでは動作確認してみましょう。シェル上で pdp7 コマンドの引数に作成したスクリプトを指定して実行してください。

実行すると、ベクタースキャンディスプレイをシミュレートしているウィンドウに図 4.2 の様に正方形が描画されます。



▲図 4.2: 正方形が描画された様子

複数の図形や複雑な図形を描画する際は

この節では一筆書きで正方形を一つ描画してみました。複数の図形を描画す

る、あるいは複雑な図形を描画するといった場合は一筆書きを頑張らずとも、ベクタモードのビット 16 による座標移動を活用したり、一度ベクタモードを抜けてパラメータモードに戻った後にそこで描画停止せずに再度ポイントモードへ遷移させ座標を設定する、といった方法が使えます。

【コラム】SVG に保存された線の情報から Type 340 命令列を生成するスクリプト

Type 340 命令列の自動生成の例として、SVG 形式で保存された線の情報から Type 340 の命令列を生成するシェルスクリプトを作ってみました。自分用に作った程度のものですが、下記の GitHub リポジトリで公開していますので、もし興味があれば見て/試してみてください。

- `https://github.com/cupnes/sh_pdp7_svg2simh`

Type 340 命令列を生成するだけでなく、それを描画する PDP-7 命令まで生成し、最終的に simh スクリプトができあがるようになっています。ただ、入力として与える SVG 画像の作り方に制約があったり、シェルスクリプトであるため生成に時間がかかったりします。詳しくは上記リポジトリの README.md をご覧ください。

なお、表紙の黒い円の中に描かれているタイトル文字列はこのスクリプトで描画したものです。当時の形を模して円形にくり抜いた灰色のパネルをかぶせていますが、SimH のベクタースキャンディスプレイのウィンドウのスクリーンショットとしては図 4.3 の通りです。なお、この場合の simh スクリプト生成にかかった時間は私の環境[*14]の場合 40 秒程でした。

▲図 4.3: 表紙の内容を描画した状態

4.4　ライトペンによる描画部分のヒット期間検出

♣ ライトペンでできること (少なくとも SimH 上は)

PDP-7 でのライトペンにどのような事ができたのか、実機の挙動は分かりませんが、SimH においては「描画箇所をヒットしたかどうか」で処理を分岐させることができます。

ベクタースキャンディスプレイをシミュレートしているウィンドウにマウスカーソルが表示されていて、これが今のライトペンの座標を示しています。ライトペンはペ

[*14] CPU: Core i7 1.8GHz、OS: Debian 11

ン先端の受光部分を画面に押し当てることで使用していた (と思われ)、ウィンドウ内でマウス左ボタンを押下するとマウスカーソルが十字のアイコンに変わるのは「押し当てている」状況を示しているものと思われます。この「押し当てている」状態で描画した点や線に触れた (ヒットした) かどうかで処理を分岐させることができます。

この節ではヒットしている期間の検出の例として、前節までで作成したスクリプトに追加実装を行い、ヒットしている間「HIT」という文字列を出力するようにしてみます。

♣ 描画部分へのヒットの有無で分岐 (IDSP 命令)

描画した点や線へのヒットの有無で処理を分岐させる命令が `IDSP` 命令です (表 4.9)。

▼表 4.9: PDP-7 命令紹介: IDSP

アセンブリ表記	機能
IDSP	ライトペンを「押し当てている」状態で、 描画した点や線に触れていたら 次の命令をスキップする

♣ 描画部分にヒットしたら文字列を出力させてみる

それでは、 `IDSP` 命令を使用して、正方形の線にヒットしたら文字列「HIT」と出力させるようにしてみます。

まずアドレス 0 に文字列「HIT」の ASCII データと、1 文字出力する関数と文字列を出力する関数を追加します (リスト 4.15)。なお、文字列は何度も出力されるので文字列末尾にスペースを置いています。1 文字/文字列出力の関数は第 3 章で実装したものです。

▼リスト 4.15: 「HIT」の ASCII データと 1 文字/文字列出力の関数を追加

```
# [定数]    ←追加(ここから)
# "HIT "
d -a 00 H
d -a 01 I
d -a 02 T
d    03 40
d    04 0  ←追加(ここまで)

# [メイン処理]
 ・・・省略・・・
```

```
# [Type 340処理]
・・・省略・・・
# パラメータモード: 描画停止、STOP割り込み設定
d 1007 003000

# [300: 指定された1文字を出力する関数]  ←追加(ここから)
・・・第3章のものと同じ・・・

# [400: 指定された文字列を出力する関数]
・・・第3章のものと同じ・・・              ←追加(ここまで)

# [実行]
・・・省略・・・
```

続いて「[メイン処理]」に `IDSP` 命令を用いたライトペンのヒット時の分岐と、ヒットした際に文字列出力関数を呼び出す処理を追加します (リスト 4.16)。

▼リスト 4.16: ヒット時の分岐と文字列出力を追加

```
・・・省略・・・

# [メイン処理]
# 0o1000をACレジスタの下位12ビットへ設定
d -m 100 law 1000
# ACレジスタの下位12ビットを描画処理の開始アドレスとして設定し、描画処理を開始
d -m 101 idla
# ライトペンが描画箇所に触れていたら次の命令をスキップ  ←追加(ここから)
d -m 102 idsp
# - 触れていない場合: 触れている場合の処理を飛ばす
d -m 103 jmp 105
# - 触れている場合: 触れている場合の処理へジャンプ
d -m 104 jmp 110                                ←追加(ここまで)
# STOP割り込みが設定されていたら次の命令をスキップ
d -m 105 idsi                                   ←変更
# - 設定されていない場合: 描画開始直後の処理まで戻る
d -m 106 jmp 102                                ←変更
# - 設定されている場合: 最初へ戻る
d -m 107 jmp 100                                ←変更
# 描画箇所に触れている場合の処理                  ←追加(ここから)
# - ACレジスタへ0(アドレス0o160の値)を設定
d -m 110 lac 160
# - ACレジスタで指定されたアドレスの文字列を出力
d -m 111 jms 400
# - 描画開始直後の処理まで戻る
d -m 112 jmp 102
# メイン処理内定数
# - 定数0
```

```
d 160 0                                          ←追加(ここまで)

・・・省略・・・
```

IDLA 命令で Type 340 の処理を開始した直後 (アドレス 0o102) に IDSP 命令を追加しています。アドレス 0o102 以降は、Type 340 側で描画停止するまでの間繰り返し実行される部分です。そしてこの IDSP 命令の際に描画箇所 (正方形の線) にヒットしていたら「描画箇所に触れている場合の処理」(アドレス 0o110) までジャンプします。

「描画箇所に触れている場合の処理」では、AC レジスタに文字列「HIT (スペース)」のアドレス (0) を設定し文字列出力関数 (アドレス 0o400) を呼び出します。関数から戻ったら、描画開始直後 (アドレス 0o102) までジャンプします。

♣ ヒットで文字列は出力されるが・・・

それでは動作確認してみましょう。作成したスクリプトを pdp7 コマンドの引数に指定して実行してください (リスト 4.17)。

▼リスト 4.17: 一度ヒットすると無限に出力されてしまう

```
$ pdp7 <作成したsimhスクリプト>

PDP-7 simulator V4.0-0 Current        git commit id: 09899c18
HIT HIT HIT HIT HIT HIT HIT  ・・・以降無限に出力・・・
```

終了する際は Ctrl+e で実行を停止させてください。

ベクタースキャンディスプレイをシミュレートしているウィンドウに描画されている正方形の線をクリックしたり、ドラッグしてカーソルが十字の状態で線に触れたりすると「HIT」という文字列が出力されます。ただ、出力され始めると線から離れた後も無限に出力され続けてしまいます。

♣ ヒット割り込みをクリアする (IDRS 命令)

無限に「HIT」が出力され続けてしまうのは、IDSP 命令でヒット判定になり続けているためです。Type 340 にはライトペンのヒットで設定される割り込みがあり、この割り込みは一度設定されると自動的にはクリアされません。この割り込みをクリアするには IDRS 命令を使用します (表 4.10)。

*15 実は実装としては割り込みをクリアする以外にも sim_activate_abs という関数を IDLA 命令の時と同じ引数で呼び出しています。この関数については、デバイス別の「イベントキュー」というものを操作している事は分かっているのですが、それにどういう意味があるのかよく分かってい

▼表 4.10: PDP-7 命令紹介: IDRS

アセンブリ表記	機能
IDRS	ヒット割り込みをクリアする[*15]

それでは、`IDRS` 命令を使用してヒット割り込みをクリアするようにしてみましょう。「描画箇所に触れている場合の処理」内の「描画開始直後の処理まで戻る」の直前に追加します (リスト 4.18)。

▼リスト 4.18: 触れている場合の処理の最後にヒット割り込みクリアを追加

```
 ・・・省略・・・
# 描画箇所に触れている場合の処理
# - ACレジスタへ0(アドレス0o160の値)を設定
d -m 110 lac 160
# - ACレジスタで指定されたアドレスの文字列を出力
d -m 111 jms 400
# - ヒット割り込みをクリア    ←追加
d -m 112 idrs               ←追加
# - 描画開始直後の処理まで戻る
d -m 113 jmp 102            ←変更
# メイン処理内定数
 ・・・省略・・・
```

IDRS 命令無しだとヒット時に線が消える

`IDRS` 命令を追加する前のプログラムでは、ヒット時の挙動として「HIT」が出力され続ける他に、実は「描画した線が消える」という事も起きていました。

これは、4.2 節で `IDSI` 命令を紹介する際に言及した「何らかの割り込みが設定されている間、Type 340 の命令実行を止める」という挙動によるものです。Type 340 は同じ表示を続けるには繰り返し描画し続ける必要がありますが、ヒット割り込みが設定されたことで Type 340 の命令実行が行われなくなり、描画したものが消えてしまった、という訳です。

♣ 動作確認

それでは改めて動作確認してみます (リスト 4.19)。

ません。なお、`IDRS` 命令の実装は PDP18B/pdp18b_dpy.c の `dpy05` 関数の `pulse` 変数が 4 の場合の処理です。また、`IDLA` 命令の実装も同ファイルの `dpy06` 関数の `pulse` 変数が 4 の場合の処理です。

▼リスト 4.19: 動作確認 (線を一度クリック)

```
$ pdp7 <作成したsimhスクリプト>

PDP-7 simulator V4.0-0 Current        git commit id: 09899c18
HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT ›
›HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT ›
›HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT ›
›HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT HIT ›
›HIT HIT  ←ヒットしなくなると止まる
```

今度はヒットしなくなると「HIT」の出力が止まるようになり、これで「ヒットしている期間」を検出できるようになりました。

ただ、リスト 4.19 は線を一度クリックしただけの結果で、これだけ「触れている場合の処理」が走ってしまいます。使い方次第ではありますが、何かに使う際はチャタリング除去のようなことを行うと良いかもしれません。

おわりに

ここまで読んでいただきありがとうございます！ 本書では、シミュレータ「SimH」を用いて PDP-7 のアセンブリ言語によるプログラミングを体験していただきました。PDP シリーズ等の 1960 年代のコンピュータは、もはや目にすることさえ難しく、ましてや「プログラミングをしてみる」機会は無いに等しいです。そのようなことから、SimH の様な現代の PC で動作するシミュレータはありがたい存在です。本書を通して「昔のコンピュータはこんな感じだったのか」と感じて、かつ「楽しんで」いただけたら嬉しいです。

なお、第 1 章でも紹介した通り、PDP-7 は UNIX が初めて作られたコンピュータです。ケン・トンプソンと共に UNIX を作ったデニス・リッチーが、UNIX の進化の歴史についてまとめるために 1979 年に発表した論文[*16]で、「ファイルシステム」や「プロセス」といった仕組みが PDP-7 で動く最初の UNIX の時点で存在していたと書いています。

ファイルシステムやプロセスといった仕組みは、もはや OS の基本的な設計として根付いています。ハードウェアが進化し性能が上がることで、ソフトウェアとしては「できること」が増えていきます。PDP-7 から半世紀以上経過し、ソフトウェア的にできることの幅がとても広くなった現代でも、これらの基本的な設計については塗り替えられることなく存在しているというのは、それだけその設計が強固であることを示していると思います。そして、私としては、できることが少なかったからこそ、シンプルで強固な設計が生まれたのではないか、と考えています。

OS の基本的な設計自体を変える必要が無かった事はこれまでの歴史が物語っています。ただ、変えてみたいならそこを変えても良いのが趣味の自作 OS です。現在、独自の「バイナリ生物学」という考え方に基づく「DaisyOS」という OS を作っています。今は PoC としてゲームボーイ上で動作するものを作った所ですが、これを PDP-7 で動くように設計・実装することで、バイナリ生物学や DaisyOS の考え方・設計をよりシンプルで強固なものにしたいと考えています。

そんな訳で、なぜ今 PDP-7 の本を出したのかというと、このような野望 (?) があるから、という話でした。

[*16] The Evolution of the Unix Time-sharing System `https://www.bell-labs.com/usr/dmr/www/hist.html`

SimH で PDP-7 ベアメタルプログラミング

シミュレータ上で
アセンブリ言語によるプログラミングを体験！

2023 年 5 月 20 日　ver 1.0 （技術書典 14）

著　者　大神祐真
発行者　大神祐真
連絡先　yuma@ohgami.jp
　　http://yuma.ohgami.jp
　　@yohgami (https://twitter.com/yohgami)
印刷所　日光企画

(powered by Re:VIEW Starter)