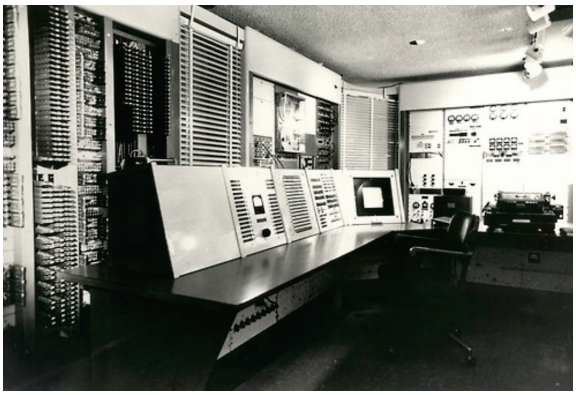
PDP-8の再評価と再構成

和田 英一[*1]

概要プログラミングの楽しみ方のひとつは, アセンブリ言語を使って昔の計算機の機械語プログラムを書くことである. そういう対象になる計算機は少なからず存在するが, 代表としてとりあげて異存がないと思われるのは, 1960 年代後半から 5 万台超を出荷した DEC(Digital Equipment Corporation) 社の PDP-8(Programmed Data Processor) であろう.
PDP-8 の愛好者はいまだに多く, ウェブ上に様々な情報が散見される. ソフトウェアによるシミュレータもあり, また PDP-8 の通りに動くハードウェアを作ったという報告もある.
私もかなり前から PDP-8 のシミュレータを作り, 簡単なプログラムを走らせたりしてきたが, 最近またいろいろなプログラムを書いてみた結果, 改めて PDP-8 のアーキテクチャの功罪を評価できるほどになった.
一方, 単なるシミュレータ上だけでなく, 更に実感のある PDP-8 を手元に置きたいと思うようになり, ネット上に Raspberry Pi を使った PDP-8 の実装経験を見たこともあって, 自分でも Raspberry Pi にいくらかの入出力装置を付けた個人用 PDP-8 を構成しようと思いたった.
これまで Scheme で書いてあった PDP-8 のシミュレータやアセンブラは Raspberry Pi で動く Python に書きなおした.
Raspberry Pi の GPIO(General-Purpose Input/Output) の入出力ピンは 26 本. PDP-8 のパネルのライト (12x2) とスイッチ (12+7) の個数を処理するには足りないので, Port Expander MCP23017 を利用する. MCP23017 を使うと GPIO ピンは 2 本減るが, ピンは MCP23017 の 1 個あたり 16 本増えるので, 2 個使えば十分である.

TX-0[*2]



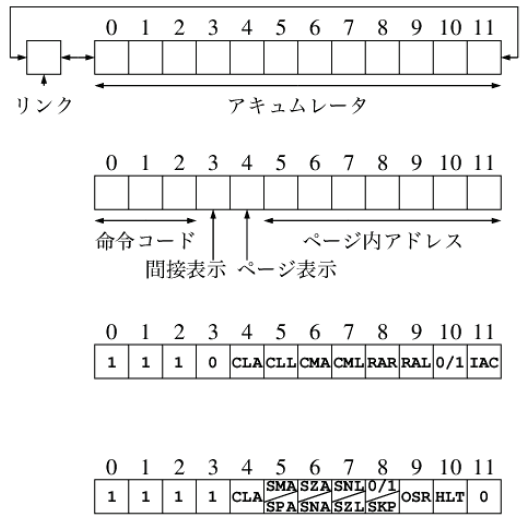
私が初めて MIT を訪れたのは 1958 年秋であった. いろいろなところを見学したが, 強烈に覚えているのは TX-0 の計算機室である.
TX-0 は MIT の Lincoln Laboratory で試作したトランジスタ計算機で, サイクルタイム 6μ 秒. 1 語 18 ビットで 4096 語の記憶装置を持つ. 命令語も 18 ビットだが, 命令部分が 2 ビット, 番地部分が 16 ビットであった. 命令部が 2 ビットなので, 命令は 4 種類しかなく, 00 は `sto x`, 01 は `add x`, 10 は `trn x` でこれはアキュムレータが負なら x へジャンプする. 最後の 11 は `opr x` で, 後続のビット構成でアキュムレータや入出力機器を操作する.
後年この TX-0 を設計仕直し商用化したのが PDP-1 といわれている.
TX-0 がその後どうなったかは知らぬが, 私が 1973 年から 74 年に MIT の電気工学科に滞在した折, きれいな roff 出力のレポートを提出した学生が TX-0 を使ったといったから, 学生が使える計算機になっていたのであろう.

[*1] IIJ 技術研究所
ew@iij.ad.jp
[*2] https://en.wikipedia.org/wiki/TX-0

PDP-8

MIT からスピンアウトして設立されたひとつが Digital Equipment Corporation (DEC) で, PDP-1 の後も後継機を販売していた. その中でミニコンとして成功したのが PDP-8 である [0].
私は PDP-8 を直接使ったことはないが, 私のいた東大計数工学科には DEC の Link-12 があり, これは LINC(Laboratory INstrument Computer) と PDP-8 の合いの子であったから, 私は専ら PDP-8 モードにして使い, ユニークなアーキテクチャの機械を楽しんだ.
Wikipedia を見ると, PDP-8 の初出荷は 1965 年 3 月で, 各種のモデルを合わせ 5 万台以上が売れ, 廉価版の PDP-8/S は 1 万ドル以下だったそうだ.
今でも PDP-8 の愛好者は少くなく, ネット上に様々な情報がある (後述). なにしろ極めてすっきり美しいアーキテクチャなので, シミュレータを作るのは容易で, 私も何度作ったか分らない.

PDP-8 のアーキテクチャ[*3]



簡単に概要を示すと, 1 語は 12 ビット, 記憶装置は 12 ビットの 4096 語. 1 ページが 128 語の 32 ページで構成される.
演算装置の中心は 12 ビットのアキュムレータで, 2 の補数 (4096 の補数) で演算される. アキュムレータの左に 1 ビットのリンクがあり, 加算の結果繰上がりがあると, リンクのビットは反転される. アキュムレータのシフトは回転シフトであり, リンクを経由する. アキュムレータのクリアはリンクに影響しない.
図の 2 段目は記憶装置参照命令で 12 ビット. 下のような構成である.
左端の 3 ビットが命令コード. 右端の 7 ビットがページ内アドレスである. ビット 4 が 0 の時は第 0 ページの番地を示し, 1 の時はその命令のあるページ内でのアドレスを示す.
ビット 3 が 1 の時は, その語の内容を番地とする (間接番地.)
記憶場所の八進法で 0010 から 0017 までを間接番地で参照すると, この番地の内容を 1 増やしてから使われるから要注意である.
記憶場所参照命令は, 命令コードの 0 から 5 が次のようである.

```
0 AND
1 TAD
2 ISZ
3 DCA
4 JMS
5 JMP
```

AND は番地の指す記憶場所の内容とアキュムレータのビット毎の AND をとり, 結果をアキュムレータに置く.
TAD は番地の指す記憶場所の内容をアキュムレータに加える. 繰上がりはリンクを反転する.

[*3] https://sites.fas.harvard.edu/~cscie287/fall2017/slides/PDP-8%20Introduction%20to%20Programming%20pp.%202-4%20to%202-27.pdf

ISZ は番地の指す記憶場所の内容を 1 増やし, その結果が 0 になれば, 次の命令をスキップする.
DCA はアキュムレータの内容を番地の指す記憶場所に入れ, アキュムレータをクリアする.
JMS はこの命令の次の番地を, 番地の指す記憶場所に入れ, その次の命令に進む. サブルーチンジャンプ.
JMP は番地の指す記憶場所の命令に進む.
命令コード 6 は入出力命令で, テレタイプのキーボードから読んだり, プリンタに印字したり, 紙テープリーダから読み込んだりする. この命令の説明は省略する.
命令コード 7 はマイクロプログラム命令で, 図に示すようにビット 3 の 0, 1 に従って, グループ 1 とグループ 2 とがある.
グループ 1 はビットの 4,5,6,7 が 1 だと, アキュムレータをクリア (CLA), リンクをクリア (CLL), アキュムレータの各ビットを反転 (CMA), リンクを反転 (CML) する.
ビット 8, 9 はアキュムレータとリンクを繋げた左 (RAL) または右 (RAR) 回転で, ビット 10 の 0/1 に従って 1 ビットか 2 ビット (RTL, RTR) になる. ビット 11 が 1 だと, アキュムレータを 1 増やす (IAC).
マイクロプログラムでは実行順が重要である. 最初がクリア, 次が反転, 次が IAC, 最後が回転となる.
グループ 2 は, ビット 4 が 1 の時, アキュムレータのクリア (CLA), ビット 5,8 が 1,0 の時, アキュムレータの負 (ビット 0 が 1) でスキップ (SMA), 1,1 の時, アキュムレータの正でスキップ (SPA). ビット 6,8 が 1,0 の時, アキュムレータの=0 でスキップ (SZA), 1,1 の時, アキュムレータの $\neq$0 でスキップ (SNA). ビット 7,8 が 1,0 の時, リンクの $\neq$0 でスキップ (SNL), 1,1 の時, リンクの=0 でスキップ (SNL) である.
ビット 9 が 1 の時, コンソールの 12 ビットのスイッチをアキュムレータとビットごと OR する. ビット 10 が 1 の時, 計算機を停止する (HLT). 実行順はまずスキップ, クリア, スイッチ読込みと停止である.
プログラムでは, CMA IAC のように書くだけである. こう書くとアキュムレータがビット反転され, 1 足されるので, アキュムレータの値を負にする. PDP-8 には減算命令はなく, A−B は B をアキュムレータに置き, CMA IAC をやってから A を足す.
ついでだが, PDP-8 のビット演算は AND しかない. A XOR B は A + B から A AND B の 2 倍を引く. A OR B は CMA(CMA(A) AND CMA(B)) とする.
ところで PDP-8 ではプログラムは原則八進法で書く. 格納開始番地は星印 (*) を使い

```
*200
```

のように書く. 記号アドレスの定義は

```
start,
```

のようにコンマ (,) を使う. 命令のアドレス部には記号アドレスや, この命令の番地を示すピリオド (.) や, プラス, マイナス (+, -) に数を続ける.
斜線 (/) は行末までのコメントの開始を示す. プログラムの最後には ($) を書く.
例えばアキュムレータにある正を数を十進法の 10 で割り, 商を q, 剰余を r に置くサブルーチン div10 を八進の 100 番地からに入れるには (以後プログラムは小文字で記す.)

```
*100
div10,0 /サブルーチンの入口. 帰り番地が入る
  dca r /被除数を r に入れる
  dca q /商の場所 q をクリアする
  tad r /部分剰余をアキュムレータに置く
  tad m10 /10 を引く
  spa /剰余が正ならスキップ
  jmp .+3 /負になったので後始末へ
  isz q /剰余を 1 増やす (↗)
  jmp .-4 /4 命令前へ戻る
  tad p10 /引き過ぎた剰余を戻す
  dca r /剰余を r に入れる
  jmp i div10 /主プログラムへ戻る
m10,-12 /定数 -10
p10,12 /定数 10
r,0 /剰余の場所
q,0 /商の場所
$
```

のように書く.

PDP-8 の算術ルーチン

PDP-8 は RISC の走りみたいなものなので, 乗除算からサブルーチンの世話になる. それらを纏めたライブラリがネットにある. 例えば乗算はこんな具合である.
このプログラムはなかなか超絶技巧である. 呼び出すとき乗数をアキュムレータに置き, `jms mult` で飛ぶ. 被乗数はこのサブルーチンジャンプ命令の次に置く. サブルーチンではまずリンクを零にする. 乗数が正ならスキップ; 負なら `cma` でアキュムレータの 1 と 0 を交換し, `iac` で 1 を加え正にする. 同時にリンクを反転する. 正の乗数を `mp1` に置く. アキュムレータが零になったので, 積の場所 `mp5` を零にする.

```
000 mult, 0                                  (↗)
001  cll
002  spa /test for negative multiplier
003  cma cml iac
004  dca mp1
005  dca mp5
006  tad i mult
007  sna /test for zero mutiplier
008  jmp mpsn+2 /jmp if multiplicand=0
009  spa /test for negative multiplicand
010  cma cml iac
011  dca mp2
012  tad thir
013  dca mp3
014 mp4, tad mp1 /multiply loop proper
015  rar
016  dca mp1
017  tad mp5
018  szl /test if multiplicand should be added
019  tad mp2
020  cll rar
021  dca mp5 (↗)
022  isz mp3 /test for end of loop
023  jmp mp4
024  tad mp1
025  rar
026 mpsn, szl
027  jmp comp
028  dca mp1
029  tad mp5
030 mpz, isz mult /exit to calling program
031  jmp i mult
032 comp, cma cll iac /complement prduct
033  dca mp1
034  tad mp5
035  cma
036  szl
037  iac
038  jmp mpz
039 thir, 7764 /eleven in decimal
040 mp1, 0
041 mp5, 0
042 mp2, 0
043 mp3, 0
```

6 行目で被乗数を読み込む. それが零なら 8 行目から飛ぶ. それが負なら 10 行目で正にし, リンクを反転する. つまりリンク =1 は積の負を示す (以下でこのビットを `s` という. 正の被乗数を `mp2` に置く.
カウンタ `mp3` に 7764_8 つまり -12_{10} を置き, 14 行目からループが始まる.
14 行目 (カウンタ$=-12$, `s` はリンク) 乗数を取り右回転シフト. `s` はビット 0 に, リンクには乗数の最下位が入る. これを `mp1` に置く. つまり乗数は 1 桁減り, 代りに `s` が入った.
17 行目 積 (`mp5`) を取り, リンクが 1 なら被乗数を足す. そして右回転シフト. 積の最下位がリンクに入り, 1 ビット右にずれた積が戻る.
再び 14 行目 (カウンタ$=-11$) 乗数を右回転シフト. `s` はビット 1 に, 積の最下位がビット 0 に. 乗数の下から 2 桁目がリンクに.
こうしてカウンタ$=-1$ で積の右回転シフトが済んだ時, `s` はビット 11 に, 積の最下位がビット 10 にある. ここでループを脱出し, 25 行目で積を右回転する. `s` がリンクに入り, 積の下半が `mp1` に収まる. 27 行目でリンクが 1 なら積の符号反転のため, `comp` へ進む. その必要がないなら, 積の下を `mp1` に置き, 上をアキュムレータに置き, 帰り番地を 1 増やして戻る.
32 行目から倍長語の符号反転で, 下の語でリンクを零にし, ビットを反転し, 1 を足す. この時, 語が零なら繰

上がりが出てリンクを 1 にする．次に上の語のビットを反転し, リンクが 1 なら 1 を足して終わる．中々よく出来ている．
ただ, 4000×4000 は 60000000 にはならず, 4000 が正だと思って 20000000 になってしまう．
一方, 除算は倍精度の被除数を単精度の除数で割るもので, 剰余の符号は被除数のそれに合わせる．呼出しは, サブルーチンジャンプ命令の次に被除数の下部分と除数を置き, 被除数の上をアキュムレータに置く．
このプログラムの中心は 33 行目から 45 行目のループである．被除数の 2 語を左へシフトしながら除数を引ければ引くを繰り返す．41 行目から入った時, 28 行目でスキップしているからリンクは 0 だ．そこで下の被除数を左回転するから, 最上位のビットがリンクへ行く．33 行目から今度は上の被除数を左回転し, 下からのビットを繋げる．その結果をとりあえず元の場所に置く．
除数を引く (というか − 除数を足す)．引ければキャリーがリンクに入る．リンクが 1 なら引けたので差をしまう．41 行目で下を取り出し, 左回転するから, 商が右から入り, 上のビットがリンクへ行く．

```
000 divide, 0                                  (↗)
001  cll
002  spa /dividend<0?
003  cma cml /yes complement and set c(l)
004  dca hdivnd /high order dividend
005  snl
006  cma
007  dca sdvnd /set dividend sign switch
008  tad i divide /fetch low order dividened
009  szl
010  cma cll iac /yes: complement
011  dca ldivnd /low order dividend
012  szl /carry?
013  isz hdivnd /yes
014  isz divide
015  tad i divide /fetch divisor
016  cll
017  sma
018  cma cml iac /negate it
019  dca divsor /save divisor
020  snl /was it <0?
021  cma /yes: ac=-1
022  tad sdvnd
023  dca snswer /answer sign switch
024  cll
025  tad divsor /compare divisor
026  tad hdivnd /with dividend
027  isz divide
028  szl cla /over flow?
029  jmp i divide /yes: divisor<dividend
030  tad m13 /13 shifts(↗)
031  dca divcnt
032  jmp dv2
033 dv3, tad hdivnd
034  ral
035  dca hdivnd /dividend left shift
036  tad hdivnd
037  tad divsor /compare divisor:dividend
038  szl
039  dca hdivnd /remainder after subtract
040  cla
041 dv2, tad ldivnd /quotient bits
042  ral /enter here
043  dca ldivnd
044  isz divcnt /done 12?
045  jmp dv3 /no: continue
046  tad hdivnd /remainder
047  isz sdvnd /dividend<0?
048  cma iac /yes
049  dca hdivnd
050  tad ldivnd /quotient
051  isz snswer /answer<0?
052  cma iac /yes: negate
053  cll
054  jmp i divide /exit
055 hdivnd, 0
056 ldivnd, 0
057 divsor, 0
058 sdvnd, 0
059 snswer, 0
060 divcnt, 0
061 m13, -15
```

除算で面倒なのは被除数が大き過ぎて商がオーバーフローすることのチェックである．大雑把にいえば上の被除数の絶対値が除数の絶対値より大きければ駄目だがそう簡単ではない．それを念頭に置き, プログラムを見なおそう．
リンクを 0 にし, 上の被除数 < 0 なら 1 の補数をとり, リンクを 1 にする．被除数の正負は上で決るので上から始めるが, 符号反転に下からキャリーが来るのは一旦我慢する．正の上の被除数を hdivnd にしまう．7 行目

では被除数の正/負に従って, −1/0 を sdvnd に置く. 剰余の符号を被除数に合わせるために記憶する. 8 行目で下の被除数を取り, 必要に応じて反転し, それでキャリーが出れば上の被除数に 1 を足す. 15 行目で除数を得, 負にして 19 行目で divsor にしまう.
18 行目で除数が正ならリンクを 1 にし, そうだと 21 行目に来て, アキュムレータを −1 にする. それに sdvnd を足すから, 被除数と除数のいずれか正の時, −1 が snswer に入る. つまり商の正負を記憶する. (ともに正の時は −2.)
24 行目からは被除数と除数の大小比較で, 被除数が大きければ, リンクを 1 にして帰る.
その後除算のループを実行し, 修了後, 先程のフラグに従って符号を定めて戻る.
このプログラムでは, isz を巧みに使っているのが分る.

書き直した乗除算ルーチン

私は新しいアーキテクチャや新しいプログラミング言語に出会ったとき, それを体験するのに乗除算と UNIX の cal のようなカレンダーのプログラムを書いてみることにしている. PDP-8 の評価にもそれをやってみた. 次にそれをざっと説明する. 前述の乗算プログラムと違い, 2 の補数のまま計算するのが好きなので, 今回もそうした. また PC-1 の w 命令のように, 初めに単精度の部分積 c があって, 単精度の乗数 b と被乗数 a に対し, 倍精度の a*b+c を得るものである.
要するに, 部分積 c に乗数 b を下の桁から 1 ビットずつ見ながら, 1 なら被乗数を足し, 0 なら足さぬで, 部分積を右シフトするものである. 乗数の最左のビットは −1 なので, その桁が 1 なら被乗数を引く.
乗算の進むに従い, 乗数は短くなり, 部分積は長くなるので, 部分積の下の方は乗数の場所を使う. 倍精度の部分積の右シフトにはリンクビットが使える. 面倒なのは右シフトのとき, 部分積の符号を正しく入れることである.

	4	5	6	7	0	1	2	3
	−4	−3	−2	−1				
4	L0	L1	L2	L3	4	5	6	7
−4					L4	L5	L6	L7
5	L1	L2	L3	L4	5	6	7	L0
−3					L5	L6	L7	0
6	L2	L3	L4	L5	6	7	L0	L1
−2					L6	L7	0	1
7	L3	L4	L5	L6	7	L0	L1	L2
−1					L7	0	1	2
0	4	5	6	7	0	1	2	3
	L4	L5	L6	L7				
1	5	6	7	L0	1	2	3	4
	L5	L6	L7	0				
2	6	7	L0	L1	2	3	4	5
	L6	L7	0	1				
3	7	L0	L1	L2	3	4	5	6
	L7	0	1	2				

	L4	L5	L6	L7	0	L1	L2	L3
	4	5	6	7	0	1	2	3
	−4	−3	−2	−1				
4	0	L7	L6	L5	4	3	2	1
−4					L4	L3	L2	L1
5	1	0	L7	L6	5	4	3	2
−3					L5	L4	L3	L2
6	2	1	0	L7	6	5	4	3
−2					L6	L5	L4	L3
7	3	2	1	0	7	6	5	4
−1					L7	L6	L5	L4
0	L4	L3	L2	L1	0	L7	L6	L5
	4	3	2	1				
1	L5	L4	L3	L2	1	0	L7	L6
	5	4	3	2				
2	L6	L5	L4	L3	2	1	0	L7
	6	5	4	3				
3	L7	L6	L5	L4	3	2	1	0
	7	6	5	4				

左の表はアキュムレータが 3 ビットで, その左にリンクビットがあったとして, 3 ビットの 2 数と足したとき (左の表) と引いたとき (右の表) の結果 (各ますの上段) と, なって欲しい値 (下段) である.
減算の方から見ると, 上の欄外に 4,5,6,7,0,1,2,3 とあるのはビットそのものである. その下の-4,-3,-2,-1 は 2 の補数のときの解釈. 左の欄外にも同様の表記がある.
右下の 0 は左欄外の 3 から上欄外の 3 を引いた結果.
その左隣の 1 は 3 から 2 を引いた結果で, ますの下半分が空白なのは, このまま結果を示す.
この約束で第 1 象限をみると, 負の数から正の数を引いた結果になっている. 最上段の中央の右の 4 は, 4 のビット列から 0 のビット列を引き, 4 のビット列になったが (上段), 算術演算の解釈としては, -4 から 0 を引き, -4(L4) になって欲しいことを示す.
解釈の仕方が分って加算の表を見ると, 第 2 象限と第 4 象限, つまり負同士, 正同士の加算はビット列の結果のままでよいが, 正と負の加算では, ビット列に L がある (リンクが 1) のときは, リンクを反転したのが演算結果であることが分る.
すなわち, 2 の補数の 2 つの数を足したとき, 両方が同符号なら, リンクビットはそのままでよいが, 異符号なら, リンクを反転するようプログラムを書かなければならない. a+b は

```
cla cll    /アキュムレータとリンクをリセット
tad a      /a を足す
spa cla    /a が正ならスキップ アキュムレータを零に
cml        /リンクを反転
```

```
tad b      /b を足す
spa        /b が正ならスキップ
cms        /リンクを反転
tad a      /a を足す
rar        /右回転シフト
```

のようにする.

次にあるのが, 私の乗算サブルーチンで, 3 行目まででカウンタを十進で −11 に設定し, ループに入る. muls が 1 桁乗算の下請けで, 13 行目でアキュムレータに 1 を置き, 14 行目の and で乗数の最下位のビットを取り出す. 零でなければ被乗数 r0 を置く. その後が上述のリンクの扱いである. muls の下請けが 2 箇所から呼ばれているのは, 11 回の加算のあと, 1 回の減算をするからである. 7 行目でループを抜けたあと, 9 行目までで被乗数を負にし, もう一度下請けにいき, サブルーチンから戻る.

積の最大値は 3777*3777+3777=17774000, 最小値は −4000*3777−4000=−2000000 である.

```
000 mul,0                                  (↗)
001 cla cll
002 tad m13
003 dca r3 /set counter
004 loop, jms muls
005 isz r3 /test counter
006 jmp loop
007 tad r0 /negate muliplicand
008 cma iac
009 dca r0
010 jms muls
011 jmp i mul /return
012 muls,0 /mul subroutine
013 cla cll iac
014 and r1  /test bit
015 sza cla
016 tad r0 /test sign of multiplicand
017 spa(↗)
018 cml
019 dca r4
020 tad r2 /add partial product
021 spa
022 cml /set link
023 tad r4 /add multiplicand
024 rar /shirt right partial product
025 dca r2
026 tad r1 /shift right multiplier
027 rar
028 dca r1
029 jmp i muls
030 r0,0 /multiplicand
031 r1,0 /multiplier
032 r2,0 /partial product
033 r3,0 /counter
034 r4,0 /workspace
035 m13,-13 /-11 in decimal
```

除算も 2 の補数のまま計算する. 2 の補数の除算では, 剰余の符号は通常のアーキテクチャやプログラミングと違い, 除数の符号に合わせるのが合理的という私の持論に従ったサブルーチンにしてある.

倍精度の被除数 n と, 零でない単精度の除数 d に対して, 単精度の商 q と単精度の剰余 r を得る. 除算だから当然オーバーフローがあり得る. 単精度が 12 ビットの場合, 正の除数は $1 \leq d \leq 2^{n-1}-1$, 負の除数は $-2^{n-1} \leq -1$ の範囲にあり, 除数が正のときの剰余は $0 \leq r < d$, 負のときの剰余は $d \leq r < 0$ である.

除算では先ず除数を負にしたものを用意する. 引き放し除算では, 部分剰余と除数が同符号なら除数を引き, 異符号なら足す. 減算命令のない PDP-8 では, 減算に備えて負の除数が必要になるので, それを作るのである.

6 行目から 12 行目で倍精度の被除数を左 1 ビットシフトし, 19 行までで除数を引くか足すする. この演算の結果, 被除数の符号が変らなければ, この除算はオーバーフローする. オーバーフローのテストは 12 行目辺りに入れる必要があるが, 今はさぼっている.

23 行目でカウンタを設定したあと, r0 と r1 を左シフトしながら, 部分剰余の符号を見ながら除数の加減算を続ける. 引くときには r0 の右端から商の 1 のビットを挿入する.

テストデータで実行した結果は次のようだ. 私の Happy Hacking Calculator[*4] の octal mode で検算すると,

[*4] http://www.iijlab.net: ew/hhcmanualen.pdf

	被除数	除数	商	剰余
a	1777 3777	3777 3777	3777	3776
b	1777 7777	4000 4000	7777	
c	6000 4000	3777 4000	0000	
d	6000 0000	4000 3777	4000	

a は 1777 3777 ent 3777 / 3777 pop 3776. b は 1777 7777 ent 4000 chs −4000 / −4000 chs 4000 pop −1 10000 + 7777. c は 6000 4000 ent 100000000 − −17774000 3777 / −4000 10000 + 4000 pop 0. d は 3777 4000 ent 100000000 − −20000000 4000 ent 10000 − −4000 / 3777 pop −4000 10000 + 4000.

となって一応合っていることが分る．ビットパターンで出力するサブルーチンを使って表示しているから変に見えるが安心してよい．

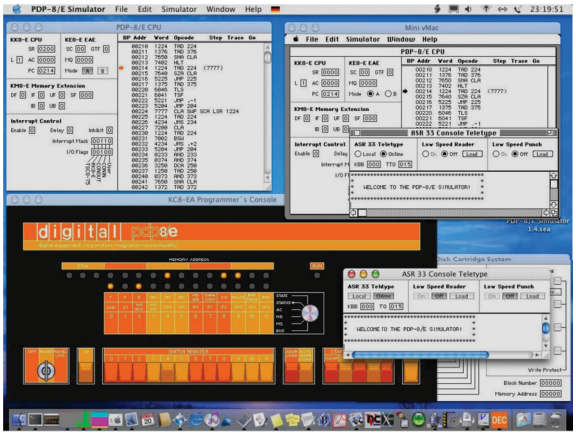
```
000 div,0
001  cla cll
002  tad r2
003  cma iac
004  dca r3  /neg divisor
005  cll
006  tad r0
007  tad r0
008  dca r0  /r0 left shift
009  ral
010  tad r1
011  tad r1  /r1 left shift
012  dca r1  /need overflow check
013  rar     /link to acc
014  tad r2
015  spa cla /skip if same sign
016  jmp .+3
017  tad r3  /same sign sub
018  jmp .+2
019  tad r2  /diff sign add
020  tad r1
021  dca r1
022  tad m13 /init counter
023  dca r4
024 loop, cll cla
025  tad r0 /left shift r0
026  tad r0
027  dca r0
028  ral
029  tad r1  /left shift r1
030  tad r1
031  dca r1
032  rar(↗)
```

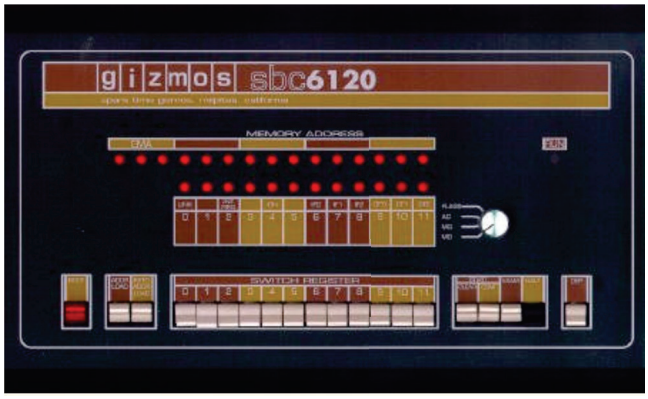
(↗)

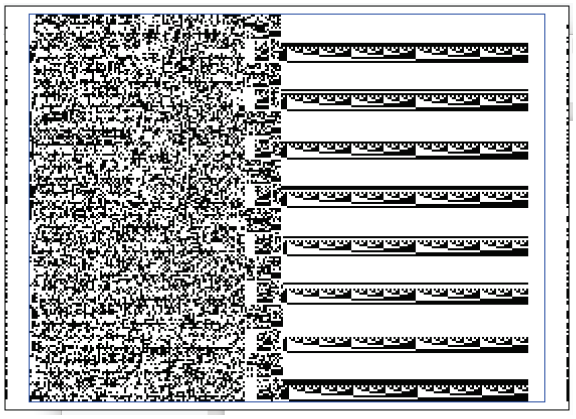
```
033  tad r2
034  spa cla  /skip same sign
035  jmp .+6
036  iac
037  tad r0   /same set quotient bit
038  dca r0
039  tad r3   /sub
040  jmp .+2
041  tad r2   /diff sign add
042  tad r1
043  dca r1
044  isz r4
045  jmp loop
046  cll cla cml rar /4000 ac
047  and r1
048  tad r2   /compare signs
049  spa cla
050  jmp .+3  /jump diff sign
051  iac      /prepare 1
052  jmp .+4
053  tad r1   /diff sign
054  tad r2
055  dca r1
056  tad r0   /r0*2+0/1
057  tad r0
058  dca r0
059  jmp i div
060 r0,0 /lower dividend, quotient
061 r1,0 /higher dividend, remainder
062 r2,0 /divisor
063 r3,0 /negative divisor
064 r4,0 /counter
065 m13,-13
```

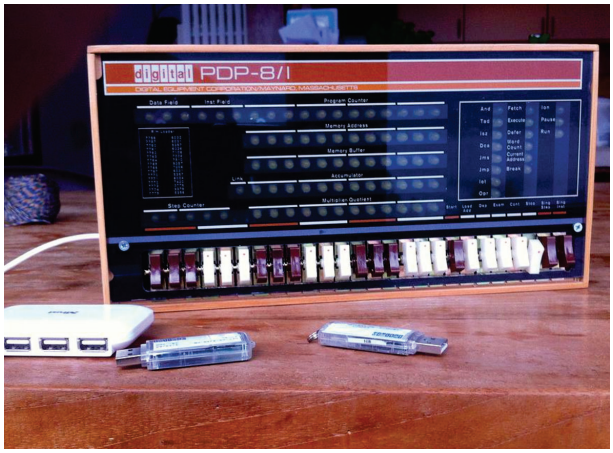
cal のプログラムも完成しているが, ここでは割愛する．PDP-8 の少ない命令とレジスタを使ってアルゴリズムを考案するのは, それ自体が楽しいが, 現時点ではノスタルジーでしかないのも十分理解している．

PDP-8 好きはあちこちに





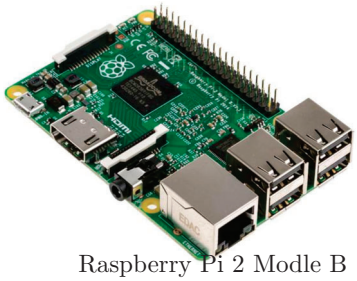




左上は Mac 用の PDP/8E シミュレータである．テレタイプの音まで聞こえるという．
右上はシングルチップの PDP-8．左下は私が昔作った PDP-8 のシミュレータで，PDP-8 の 1 語 12 ビット，4096 語の記憶装置を示している．収まっているのは，van der Poel 先生の Lisp シミュレータで，右半分のヒープ領域が見える [1]．右下は Raspberry Pi で駆動する PDP-8I で，実機の 2/3 のサイズらしい [2]．

自分の PDP-8 を作る

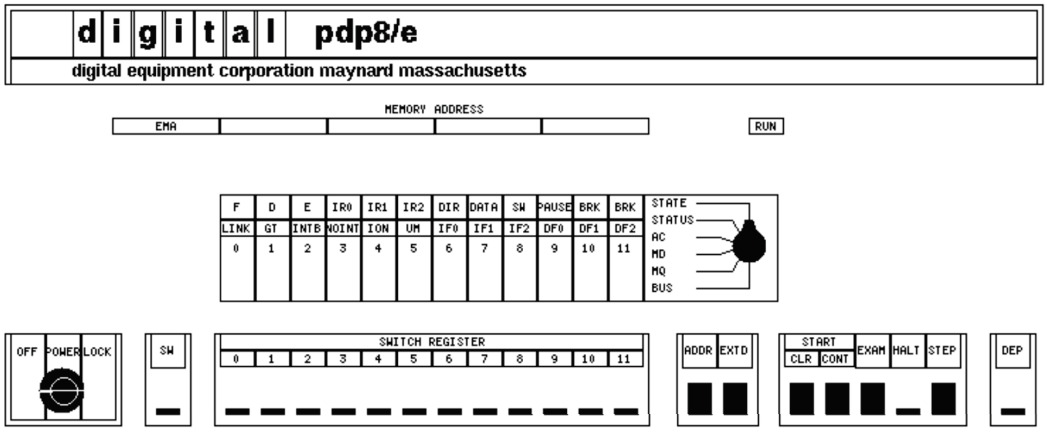
PDP-8 のシミュレータを私は C, Scheme, Python で書いたりした．だから PDP-8 のプログラムはすぐに走らせてみることが出来るが，やはりライトが点滅する様子も見たい．ネットを探すと作ってみたという記事が見付かる．すこし考えてみると，Raspberry Pi で Python のシミュレータを動かし，Raspberry Pi の GPIO で 12 個 ×2 のライトを点け，12 個 + アルファのスイッチを読み込めば，ある程度の実感が得られそうである．GPIO のピンの数は 26 本だが，MCP23017 を使えば 1 個で 16 本に増やせるから，23017 を 4 個用意すれば出来そうである．PDP-8 の入出力は 33 型テレタイプであったが，そういうものは既にないので，パソコンなどで代用する．



Raspberry Pi 2 Modle B

MCP23017

PDP-8E の面構えで参考にしたのは次の図である．



これを見て起した正面パネルは次の図のようだ (次ページ.) 元の顔を見比べてほしい.

自家用 PDP-8 の操作パネルに搭載するライトとスィッチは, 元の PDP-8 にあるもの全部をつける必要はないので, とりあえず次の図にあるものにした.

LED によるライトは 12 ビットで 2 段. 上段がプログラムカウンタで下段がアキュムレータである. 記憶装置を読み出す時は, 上段が番地で下段がその内容を示す. リンクが表示できないのは不便かもしれないが, 本家の PDP-8 では下段に切替えスィッチがあって, STATUS 位置にするとビット 0 がリンクを表示する. その時, 他のビットは別の状態を示すが, その辺は無視するのでリンク表示も割愛されてしまった.

スィッチは, スィッチレジスタに対応する 12 個のものの他, その右手にある 7 個を用意する. ADDR はスィッチレジスタの設定を CPMA(Central Processor Memory Address) レジスタへ転送するものである. その右に隙間があるが, ここには拡張記憶装置用の EXTD というスィッチがあるが, 今回はそれもないのでそのスィッチも省略した.

CLR はレジスタ類をクリアして走行状態にするもの, CONT はクリアせずに走行を再開するものである.

走行状態の時は RUN のライトが点灯する.

EXAM は CPMA の差す記憶場所の内容を下段のライトに示し, CPMA の内容を 1 増やす.

HALT は走行している計算機を停止する. RUN ライトは消える.

STEP はこれを押すと, プログラムを 1 ステップずつ実行する. DEP はデポジットキーで, 上段のライトの示す CPMA の場所に, スィッチレジスタの内容を転送する. EXAM と同様で, CPMA の値は毎回 1 ずつ増える. これらのスイッチには作業用として番号もつけてある.

合わせるとライトが 12+12+2(リンクと run)=26. スィッチが 12+7=19. 計 45 個が必要になる. Raspberry Pi の GPIO のピンは元々 26 本あるが, MPC23017 に 2 本使うので 24 本. MPC23017 を 2 個採用すると, 1 個当たり 16 本あるから 32 本. この 56 本のうち 45 本を使うことになる.

部品

トグルスイッチ ミヤマ

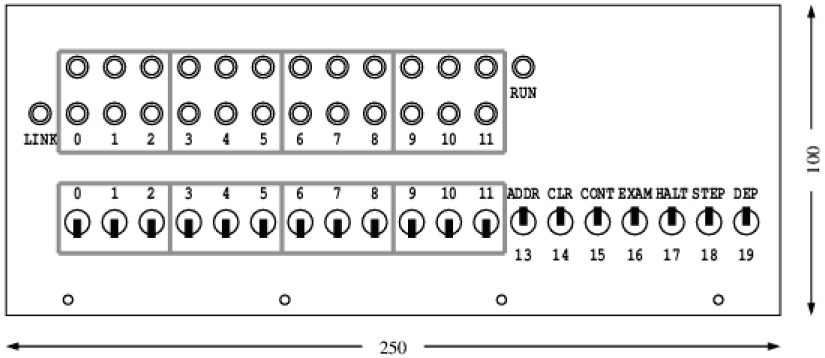
MS500 BB ON-(ON) 7 個 200*7=1400 1400*1.08=1512 円

MS500 KB ON-OFF 12 個 170*12=2040 2040*1.08=2203 円

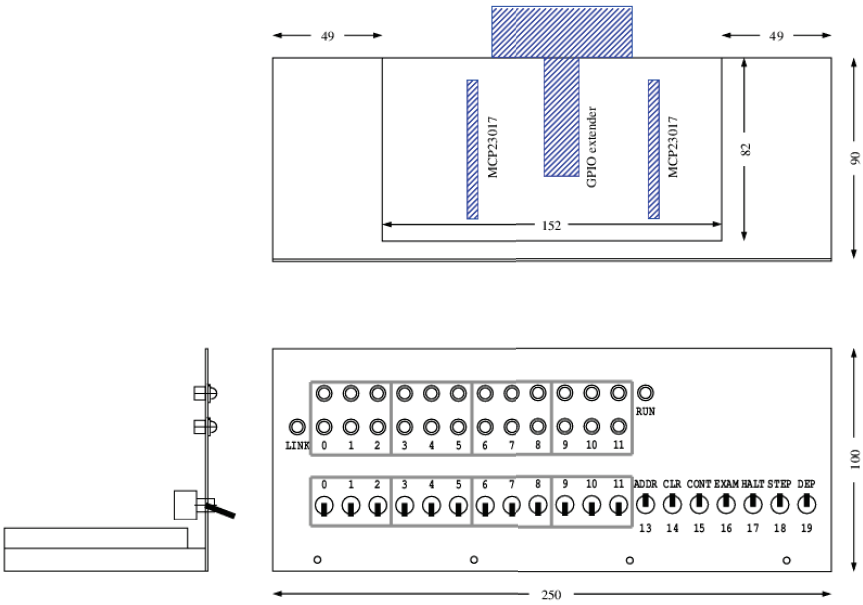
抵抗内蔵 LED I-06249 3 個 240*3=720 円

LED ブラケット P-11465 3 個 150*3=450 円

1512+2203+1170=4885 円
スイッチサイエンス MCP23017 Raspberry Pi 用 GPIO エクスパンダ 972*2+150=2094 円
Raspberry Pi B+ 用 T 型 I/O 延長基盤 837+150=987 円

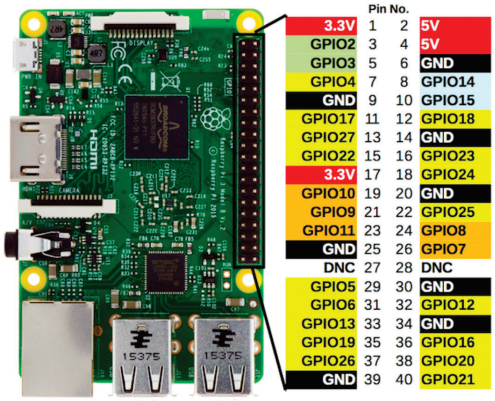


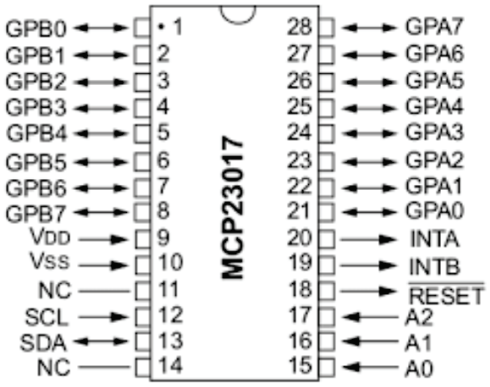
最初の設計の全体図はこのようだ．アルミのパネルは木製の台に取り付ける．その木製の台にはブレッドボードを 2,3 枚取り付け，その上で配線する．



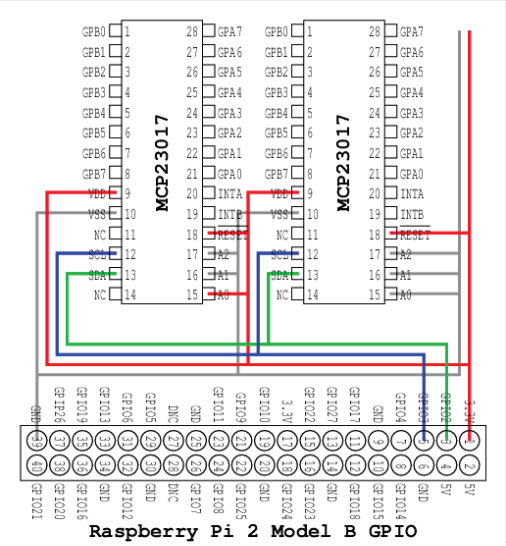
上の図は最初の案で，中央のブレッドボードに T 型延長基盤を載せ，左右のブレッドボードに GPIO エキスパンダを載せる．GPIO とエキスパンダ間の配線は次のようだ．GPIO のピンの配置と MCP23017 のピンの配

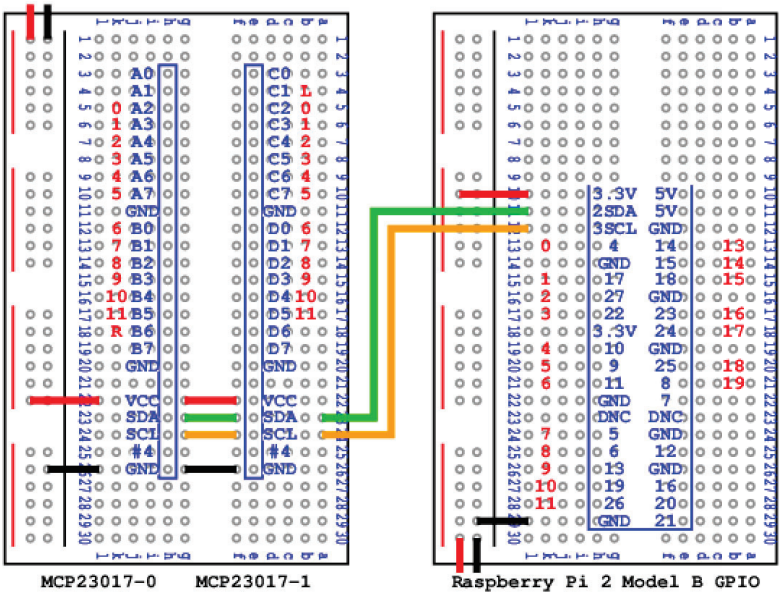
置を次に示す.





ライトは抵抗内蔵の LED を MCP23017 のピンと GND の間に接続し, スイッチは GPIO の端子と 3.3v の間に pull down 接続したので, 抵抗類の配線が不要になった.

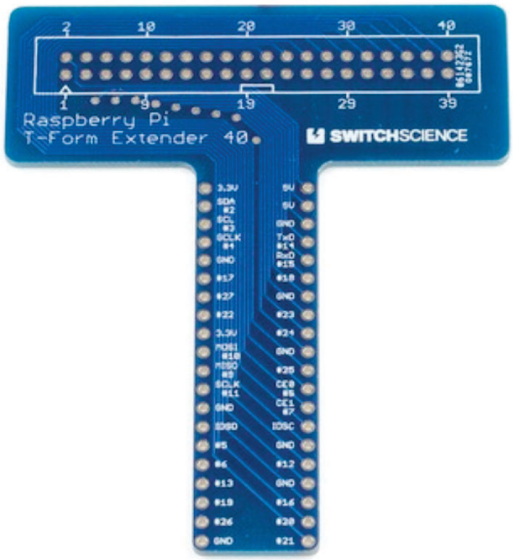




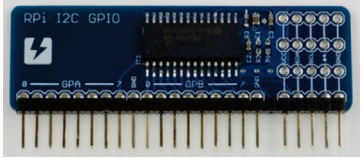
最初の案では 3 枚であったブレッドボードは, その後の検討の結果, 2 枚で足りることが分り, ブレッドボードからはみ出していた T 型延長基盤もブレッドボード内に収まることになった.

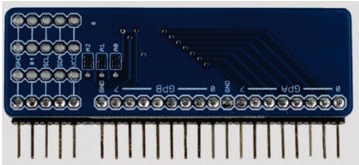
左の図がそのブレッドボードの配線である. 実際にはこのように横に並ぶのではなく, 右のボードが左のボードの上につながる. (最後のページの写真を参照)

これらのブレッドボードに差さる延長基盤とエクスパンダは次の図のようなものである.



Raspberry Pi B+ 用 T 型 I/O 延長基板

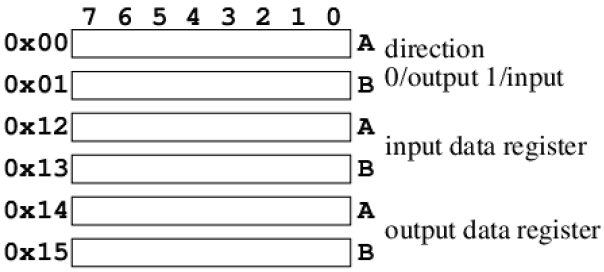




MCP23017 Raspberry Pi 用
GPIO エクスパンダ

I^{2}C

Inter-Integrated Circuit
GPIO と MCP23017 との間の通信には, I^{2}C というプロトコルを使う. 実はこれを使うので, PDP-8 のシミュレーションは予想外に遅くなることが分った.
I^{2}C では SDA (serial data) と SCL (serial clock) の 2 本を使う. これは GPIO の 2 番と 3 番のピンを使うことになっている. 通信のビットレートは 100KHz と 400KHz. もっと高速なのもあるが, このシミュレータでは 100KHz のまま使っている.
2 個あるエクスパンダには 0x20 と 0x21 のアドレスを付ける. それぞれのエクスパンダで A0〜A7, B0〜B7 の 16 個の I/O が制御出来る. それぞれのエクスパンダには 0x00, 0x01, 0x12, 0x13, 0x14, 0x15 のアドレスを持つレジスタがあり, GPIO はそれらのレジスタと情報をやりとりする.



`bus.write_byte_data(dev, outreg, data)` で dev の outreg へ data を書き,
`data=bus.read_byte_data(dev, inreg)` で dev の inreg から data を読む.
今回の入力は GPIO から直接なので, この機能は使わない.

LED の制御

Rapberry Pi で動く PDP-8 のシミュレータは, 前述のように Python で記述してある. 命令のフェッチ, 実行のループから状態を LED で表示するルーチンを呼ぶ. それがこの LED 制御のプログラムである. ブレッド

ボードの図にあるように, 左端にあるアキュムレータの 0 のビットが, エキスパンダのレジスタの 0 で右側にあるからまずビットの左右反転をする. ここは設計ミスというべきである.
下のプログラムの先頭の bitrev が 12 ビット反転で, Python だとこんな風になる. その下の display がシミュレータから呼ばれるプログラムで, グローバル変数, lacc(リンクとアキュムレータ, 13 ビット), pc(プログラムカウンタ, 12 ビット), run(1 ビット) を受け取り, マスクしたりシフトしたりして, レジスタ用のパターンを作っている.

```
def bitrev(n): #12bit reverse
 return int('{:012b}'.format(n)[::-1],2)
def display():
    global lacc,pc,run
    rev=bitrev(pc)  #display pc and run
    areg=(rev&0o77)<<2
    breg=(rev&0o7700)>>6
    bus.write_byte_data(0x20,0x14,areg)
    bus.write_byte_data(0x20,0x15,breg|(run<<6))
    rev=bitrev(lacc&0xfff) #display acc and link
    areg=(rev&0o77)<<2
    breg=(rev&0o7700)>>6
    bus.write_byte_data(0x21,0x14,areg|(lacc>>12)<<1)
    bus.write_byte_data(0x21,0x15,breg)

rev(PC) BA9876543210
 breg 7654321076543210 areg
       R
rev(AC) BA9876543210
 breg 7654321076543210 areg
                    L
```

テストプログラム

シミュレータが出来たので, 走らせるプログラムを用意しなければならない. とりあえず, Python でインタプリタを書く必要があるが, PDP-8 のアーキテクチャが簡単なので, それは難なく完成した.
PDP-8 で走るプログラムには, 前述の Lisp があるが, それ以外にも欲しい. 今回用意したのは, 素因数分解のプログラムで, 23 ビット (倍精度で正の数) を素因数分解するものである.
それには倍精度被除数の除算の剰余を得るサブルーチンと, 倍精度の正数の平方根をとるサブルーチンが必要である. 下に示すプログラムで, `(rem n d)` と `(sqrt n)` は除算を使わない剰余と開平を Scheme で試みたものである. これで上手くいくことが分ったので, PDP-8 のアセンブリ言語でプログラムした.

```
(define (rem n d) ;n を d で割った剰余を計算する
 (let ((s 0))
  (set! d (do ((e d (* e 2))) ((>= e n) e) ;d を d>=n になるまで 2 倍し
   (display (list e)) (set! s (+ s 1))))   ; その回数を s とする
  (display (list d))
  (do ((s s (- s 1))) ((or (< s 0) (= n 0)) n) ;s が負になるか n が 0 になるまで
   (display (list n d))
   (if (> n 0) (set! n (- n d))   ;n が正なら n ← n−s とし
               (set! n (+ n d)))  ;n が負なら n ← n+s とする
   (set! d (/ d 2)))))            ;d を 2 で割る
(rem 100 8)(8)(16)(32)(64)(128)(100 128)(-28 64)(36 32)(4 16)(-12 8)=> -4
```

```
(rem 100 4)(4)(8)(16)(32)(64)(128)(100 128)(-28 64)(36 32)(4 16)(-12 8)
(-4 4)=> 0     ; 実行例
(rem 96 4)(4)(8)(16)(32)(64)(128)(96 128)(-32 64)(32 32)=> 0
(define (sqrt n)  ;n の平方根を計算する
 (let ((q 0))      ; 平方根の初期値
  (do ((s 22 (- s 2))) ((< s 0))  ;s を 22 から 2 ずつ減らす
   (let ((a (fix:lsh (+ (* q 4) 1) s))) ;4q+1 を s 桁左シフトし a とする
    (if (>= n a) (begin (set! q (+ q q 1)) (set! n (- n a)))
                 (set! q (+ q q))) ;n>=a なら q ← 2q+1, n ← n−a とし
    (display (list a n q))))       ;n<a なら q ← 2q とする
  q))
(sqrt 99) (1024 99 0)(256 99 0)(64 35 1)(80 35 2)(36 35 4)(17 18 9)=> 9
(sqrt 100) (1024 100 0)(256 100 0)(64 36 1)(80 36 2)(36 0 5)(21 0 10)=> 10

(define (primetest n)   ; 奇数 n の素数性を調べる
  (let ((a (floor->exact (sqrt n)))  ;n の平方根を a とする
        (dd '(1 2 2 4 2 4 2 4 6 2 6 4 2 4 6 6 2 6 4 2 6 4 6 8 4 2 4
              2 4 8 6 4 6 2 4 6 2 6 6 4 2 4 6 2 6 4 2 4 2 10 2 10))
        (ind 0) (d 2)) ;dd は 2,3,5,7 で割れない疑似素数列の差, d は試す除数
  (let loop ()
    (set! d (+ d (list-ref dd ind)))  ; 次の除数を作る
    (if (> d a) (display (list 'prime n)) ;d が a を超えたら素数
     (if (= (modulo n d) 0) (display (list 'composite n)); 割り切れたら合成数
      (begin (set! ind (+ ind 1)) ;dd の列を循環して除数を作る用意
       (if (= ind 52) (set! ind 4))
       (loop)))))))  ;loop へ戻る
```

夏のプロシンの時点では, ハードウェアはすべて完成していたが, シミュレータのスイッチ読込みプログラム部分が未完であった. また時間のあるときに, その辺のプログラムを追加し, この名機のブートの気分を味わいたいと考えている.

参考文献

[0] INTRODUCTION TO PROGRAMMING PDP-8 Family Computers

[1] 和田英一, 中西流「目でみる GC」再現, 2006 年夏のプログラミング・シンポジウム報告集.

[2] https://www.raspberrypi.org/blog/pidp-8i-remaking-the-pdp-8i/

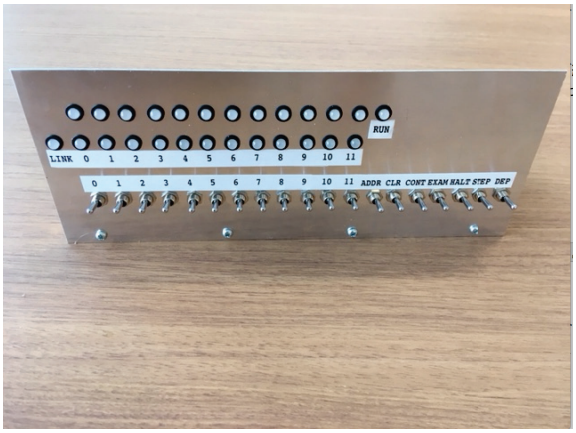
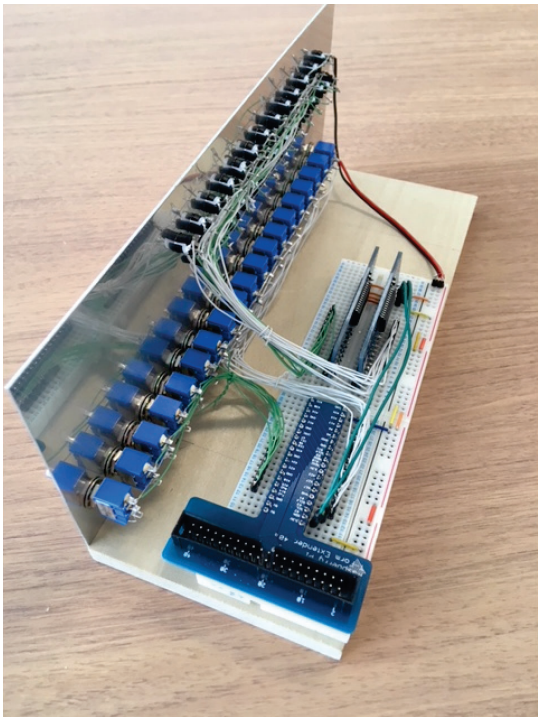
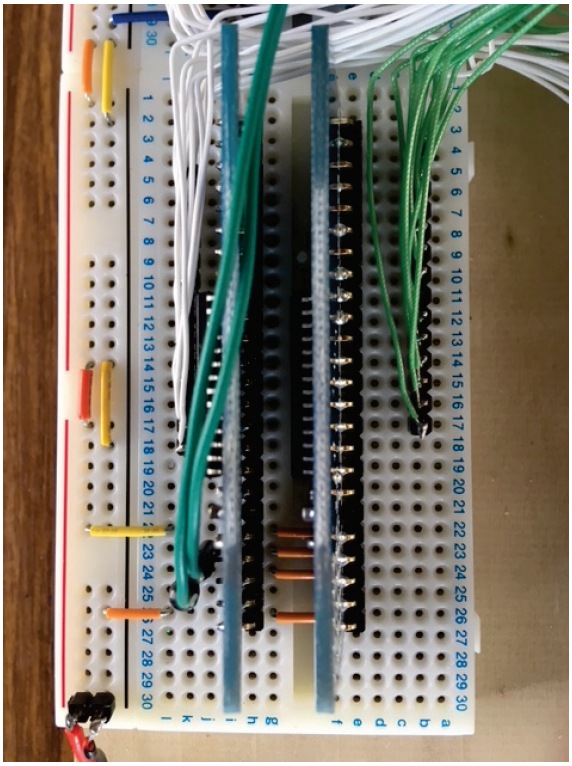
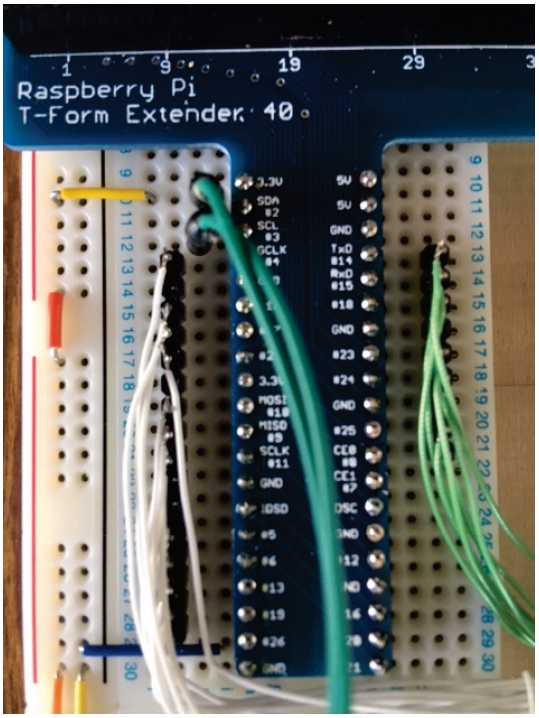
討論

Q. 竹内 シミュレータの速度はどうか.

A. ソフトウェアのシミュレータは実機よりも遥かに速い. しかし I^2C のシリアル通信は相当遅い. 速くする方法はあるが, 速度には関心がないので変更してはいない.

Q. 無名氏 TX-0 は何台あったか.

A. 試作機なので 1 台だと思う.

写真

右 上から見た様子
下 正面パネル
下の下 GPIO エクスパンダ
下の右 T 型 I/O 延長基盤の付近