

TeC 教科書

Ver. 4.0.0

徳山工業高等専門学校  
情報電子工学科

Copyright © 2017 by  
Dept. of Computer Science and Electronic Engineering,  
Tokuyama College of Technology, JAPAN

本ドキュメントは CC-BY-SA ライセンスによって許諾されています。ライセンスの内容を知りたい方は  
<http://creativecommons.org/licenses/by-sa/4.0/deed.ja> でご確認ください。

# 目次

|              |                  |          |
|--------------|------------------|----------|
| <b>第 1 章</b> | <b>はじめに</b>      | <b>1</b> |
| 1.1          | この科目で学ぶこと        | 1        |
| 1.2          | 勉強の進め方           | 1        |
| 1.3          | 教材用コンピュータ        | 1        |
| <b>第 2 章</b> | <b>情報の表現</b>     | <b>3</b> |
| 2.1          | コンピュータ内部の情報表現    | 3        |
| 2.1.1        | 電気を用いた情報の表現      | 3        |
| 2.1.2        | ビット              | 3        |
| 2.1.3        | より複雑な情報の表現       | 3        |
| 2.2          | 数値の表現            | 4        |
| 2.2.1        | 2 進数             | 4        |
| 2.2.2        | 2 進数と 10 進数の相互変換 | 4        |
| 2.2.3        | 16 進数            | 5        |
| 2.3          | 負数の表現            | 6        |
| 2.3.1        | 符号付き絶対値表現        | 6        |
| 2.3.2        | 補数表現             | 6        |
| 2.3.3        | 1 の補数による負数の表現    | 6        |
| 2.3.4        | 2 の補数による負数の表現    | 7        |
| 2.3.5        | 2 の補数を求める手順      | 8        |
| 2.3.6        | 2 の補数から元の数を求める手順 | 8        |
| 2.4          | 2 進数の計算          | 8        |
| 2.4.1        | 正の数の計算           | 8        |
| 2.4.2        | 負の数を含む計算         | 8        |
| 2.5          | 小数の表現            | 9        |
| 2.5.1        | 2 進数による小数の表現     | 9        |
| 2.5.2        | 10 進数との相互変換      | 9        |
| 2.6          | 文字の表現            | 10       |
| 2.6.1        | 文字コード            | 10       |
| 2.6.2        | ASCII コード        | 10       |
| 2.6.3        | JIS 文字コード        | 10       |
| 2.7          | 補助単位             | 11       |
| 2.8          | コンピュータの基本回路      | 11       |

|              |                   |           |
|--------------|-------------------|-----------|
| 2.8.1        | 論理演算と論理回路         | 11        |
| 2.8.2        | 基本的な論理回路          | 11        |
| 2.8.3        | 演算回路              | 12        |
| 2.8.4        | 記憶回路              | 13        |
| 2.9          | まとめ               | 14        |
| <b>第 3 章</b> | <b>組み立て</b>       | <b>15</b> |
| 3.1          | ハンダ付け             | 15        |
| 3.2          | 部品の取り付け           | 15        |
| 3.2.1        | 抵抗器               | 15        |
| 3.2.2        | 積層セラミック・コンデンサ     | 16        |
| 3.2.3        | IC                | 16        |
| 3.2.4        | 集合抵抗器とラダー抵抗器      | 16        |
| 3.2.5        | フェライトビーズ          | 16        |
| 3.2.6        | ジャンパ              | 16        |
| 3.2.7        | 圧電スピーカ            | 17        |
| 3.2.8        | 電解コンデンサ           | 17        |
| 3.2.9        | LED(ランプ)          | 17        |
| 3.2.10       | スイッチ              | 18        |
| 3.2.11       | JTAG コネクタ         | 18        |
| 3.2.12       | 入出力ポートコネクタ        | 18        |
| 3.2.13       | 電源コネクタ            | 19        |
| 3.2.14       | PS/2 コネクタ         | 19        |
| 3.2.15       | VGA コネクタ          | 19        |
| 3.2.16       | ゴム足の取り付け          | 20        |
| 3.2.17       | プッシュスイッチの頭の取り付け   | 20        |
| 3.3          | 完成                | 20        |
| 3.3.1        | 命令表の貼り付け          | 20        |
| 3.3.2        | 目視確認              | 20        |
| 3.3.3        | 設計データの書き込み        | 20        |
| <b>第 4 章</b> | <b>マイコンの操作</b>    | <b>21</b> |
| 4.1          | 各部の名称             | 21        |
| 4.2          | コンソールパネル          | 21        |
| 4.2.1        | コンソールパネルの構成       | 21        |
| 4.2.2        | コンソールパネルの操作方法     | 23        |
| 4.3          | リセットスイッチ          | 25        |
| 4.4          | 操作音を小さくする         | 25        |
| 4.5          | モード               | 25        |
| <b>第 5 章</b> | <b>機械語プログラミング</b> | <b>27</b> |
| 5.1          | コンピュータの構成         | 27        |
| 5.1.1        | 一般的なコンピュータの構成     | 27        |

|        |                                  |    |
|--------|----------------------------------|----|
| 5.1.2  | TeC の構成                          | 28 |
| 5.1.3  | TeC 内部の記憶装置                      | 28 |
| 5.2    | 機械語プログラミング                       | 29 |
| 5.2.1  | 機械語命令                            | 29 |
| 5.2.2  | ノイマン型コンピュータの特徴                   | 29 |
| 5.2.3  | 機械語プログラミング                       | 29 |
| 5.3    | 特殊な命令                            | 29 |
| 5.3.1  | NO (No Operation) 命令             | 29 |
| 5.3.2  | HALT (Halt) 命令                   | 29 |
| 5.4    | データ転送命令                          | 30 |
| 5.4.1  | LD (Load) 命令                     | 30 |
| 5.4.2  | ST (Store) 命令                    | 31 |
| 5.5    | 算術演算命令                           | 32 |
| 5.5.1  | ADD (Add) 命令                     | 32 |
| 5.5.2  | SUB (Subtract) 命令                | 32 |
| 5.6    | ジャンプ命令                           | 33 |
| 5.6.1  | JMP (Jump) 命令                    | 33 |
| 5.6.2  | JZ (Jump on Zero) 命令             | 34 |
| 5.6.3  | JC (Jump on Carry) 命令            | 35 |
| 5.6.4  | JM (Jump on Minus) 命令            | 35 |
| 5.6.5  | ジャンプ命令の利用                        | 35 |
| 5.6.6  | JNZ (Jump on Not Zero) 命令        | 38 |
| 5.6.7  | JNC (Jump on Not Carry) 命令       | 38 |
| 5.6.8  | JNM (Jump on Not Minus) 命令       | 38 |
| 5.7    | 比較命令                             | 39 |
| 5.7.1  | CMP (Compare) 命令                 | 39 |
| 5.8    | シフト (桁ずらし) 命令                    | 40 |
| 5.8.1  | SHLA (Shift Left Arithmetic) 命令  | 40 |
| 5.8.2  | SHLL (Shift Left Logical) 命令     | 40 |
| 5.8.3  | SHRA (Shift Right Arithmetic) 命令 | 41 |
| 5.8.4  | SHRL (Shift Right Logical) 命令    | 41 |
| 5.9    | 論理演算命令                           | 43 |
| 5.9.1  | AND (Logical AND) 命令             | 43 |
| 5.9.2  | OR (Logical OR) 命令               | 45 |
| 5.9.3  | XOR (Logical XOR) 命令             | 45 |
| 5.10   | アドレッシングモード                       | 46 |
| 5.10.1 | ダイレクト (直接) モード                   | 46 |
| 5.10.2 | インデクスド (指標) モード                  | 46 |
| 5.10.3 | イミディエイト (即値) モード                 | 46 |
| 5.10.4 | アドレッシングモードの使用例                   | 47 |
| 5.11   | 入出力                              | 48 |
| 5.11.1 | I/O マップ                          | 48 |

|              |                         |           |
|--------------|-------------------------|-----------|
| 5.11.2       | IN (Input) 命令           | 48        |
| 5.11.3       | OUT (Output) 命令         | 49        |
| 5.12         | TeC パラレル入出力 (PIO)       | 50        |
| 5.13         | TeC アナログ入力 (AIN)        | 51        |
| 5.14         | TeC シリアル入出力 (SIO)       | 51        |
| 5.14.1       | シリアル入出力                 | 51        |
| 5.14.2       | パソコンとの接続                | 51        |
| 5.14.3       | I/O ポート                 | 52        |
| 5.14.4       | シリアル出力プログラム             | 52        |
| 5.14.5       | シリアル入力プログラム             | 52        |
| 5.14.6       | シリアル入出力データ              | 52        |
| 5.15         | まとめ                     | 56        |
| <b>第 6 章</b> | <b>高度なプログラミング</b>       | <b>57</b> |
| 6.1          | クロス開発                   | 57        |
| 6.1.1        | TeC クロス開発環境             | 57        |
| 6.1.2        | アセンブラ                   | 57        |
| 6.1.3        | ダウンロード                  | 58        |
| 6.2          | スタック                    | 59        |
| 6.2.1        | 仕組み                     | 59        |
| 6.2.2        | PUSH 命令                 | 60        |
| 6.2.3        | POP 命令                  | 60        |
| 6.3          | サブルーチン                  | 61        |
| 6.3.1        | 仕組み                     | 61        |
| 6.3.2        | CALL 命令                 | 61        |
| 6.3.3        | RET (Return) 命令         | 62        |
| 6.4          | 時間の計測                   | 65        |
| 6.4.1        | マシンステート                 | 65        |
| 6.4.2        | 1ms タイマ (マシンステートの応用)    | 65        |
| 6.4.3        | 0.2 秒タイマ (入れ子サブルーチンの利用) | 65        |
| 6.4.4        | 1 秒タイマ (多重ループの利用)       | 66        |
| 6.5          | 数値の入出力                  | 67        |
| 6.5.1        | 2 進数の出力                 | 67        |
| 6.5.2        | 16 進数の出力                | 67        |
| 6.5.3        | 10 進数の出力                | 68        |
| 6.5.4        | 10 進数の入力                | 68        |
| 6.6          | 符号付数の入出力                | 70        |
| 6.6.1        | 符号付 10 進数の出力            | 70        |
| 6.6.2        | 符号付 10 進数の入力            | 70        |
| 6.7          | アドレスデータ                 | 71        |
| 6.7.1        | TeC のアドレスデータ            | 71        |
| 6.7.2        | 文字列出力サブルーチン             | 71        |

|             |                                 |           |
|-------------|---------------------------------|-----------|
| 6.7.3       | ジャンプテーブル                        | 71        |
| 6.8         | 割込み (Interrupt)                 | 73        |
| 6.8.1       | TeC の割込み種類                      | 73        |
| 6.8.2       | TeC の割込み動作                      | 73        |
| 6.8.3       | EI (Enable Interrupt) 命令        | 74        |
| 6.8.4       | DI (Disable Interrupt) 命令       | 74        |
| 6.8.5       | RETI (Return from Interrupt) 命令 | 74        |
| 6.8.6       | PUSHF (Push Flag) 命令            | 74        |
| 6.8.7       | POPF (Pop Flag) 命令              | 75        |
| 6.9         | タイマ割込み                          | 75        |
| 6.9.1       | インターバルタイマ割込み                    | 75        |
| 6.9.2       | プログラミング例                        | 76        |
| 6.10        | コンソール割込み                        | 78        |
| 6.11        | 入出力割込み                          | 79        |
| 6.11.1      | TeC の入出力割込み                     | 79        |
| 6.11.2      | 入出力割込みの使用例                      | 79        |
| 6.12        | マシンステートとスピーカ                    | 81        |
| 6.12.1      | スピーカーの仕組み                       | 81        |
| 6.12.2      | スピーカ駆動プログラム                     | 81        |
| 6.12.3      | 電子オルゴールプログラム                    | 82        |
| 6.13        | まとめ                             | 83        |
| <b>付録 A</b> | <b>TeC クロス開発環境</b>              | <b>85</b> |
| A.1         | 始めに                             | 85        |
| A.2         | アセンブルコマンド                       | 85        |
| A.3         | ダウンロードプログラム                     | 85        |
| A.4         | アセンブラの文法                        | 85        |
| A.4.1       | 行フォーマット                         | 85        |
| A.4.2       | ラベル                             | 86        |
| A.4.3       | 疑似命令                            | 86        |
| A.4.4       | 機械語命令                           | 87        |
| A.5         | アセンブラの文法まとめ                     | 88        |
| A.6         | プログラム例                          | 90        |
| A.7         | アセンブル実行例                        | 90        |
| A.8         | IPL プログラム                       | 91        |
| A.9         | 機械語プログラムファイル形式                  | 92        |
| <b>付録 B</b> | <b>命令表</b>                      | <b>93</b> |
| <b>付録 C</b> | <b>参考資料</b>                     | <b>95</b> |
| C.1         | TeC7 基板回路図                      | 95        |
| C.2         | TeC7 ピン配置表                      | 95        |





## 第 1 章

# はじめに

### 1.1 この科目で学ぶこと

現代、コンピュータと呼ばれているものはスーパーコンピュータと呼ばれる高価で高性能なものから、マイコンと呼ばれ炊飯器やエアコンに組み込まれている小型のものまで、全て同じ原理に基づき動作しています。

この原理は、1946年に米国の数学者フォン・ノイマン (Von Neumann) が提案したと言われています。現代のコンピュータの、ほぼ、全てのものは、ノイマンの提案した同じ原理を使用しており「ノイマン型コンピュータ」と呼ばれます。

「ノイマン型コンピュータ」が出現して既に約 70 年の時間が経過しましたが、未だにこれを上回る「自動機械」の構成方法は発明されていません。だから、パソコンのような一般の人がコンピュータだと考えている装置だけでなく、炊飯器やエアコンの制御装置のようなコンピュータらしくない装置まで、「自動機械」が必要などころには全て「ノイマン型コンピュータ」が使用されているのです。恐らく、あと数十年は、「ノイマン型コンピュータ」の時代が続くでしょう。

この教科書は、教育用コンピュータ (TeC) を用いて、**ノイマン型コンピュータの動作原理**をしっかりと勉強できるようにしています。ここでしっかりと勉強しておけば、将来、どんな新型のコンピュータに出会ったとしても、恐れることはありません。所詮、「ノイマン型コンピュータ」の一種ですから、皆さんはその正体を簡単に見抜くことができるはずです。

「ノイマン型コンピュータ」の原理をきちんと理解しておくことは、皆さんにとって、寿命の長いエンジニアになるための大切なステップとなります。しっかりと、がんばってください。

### 1.2 勉強の進め方

この教科書は、高専や工業高校の 1 年生と 2 年生が、教育用コンピュータ TeC を教材に、ノイマン型コンピュータの基本を学ぶことを想定して作ってあります。1 章から 5 章の途中までを 1 年で学び、残りを 2 年生 (半年～1 年間) で学びます。

1 年生では、(1) コンピュータの内部で情報を表現する方法、(2) TeC の組み立て、(3) TeC の基本的なプログラミングを学びます。2 年生では、TeC を用いた高度なプログラミングを学びます。

### 1.3 教材用コンピュータ

私達の身近にあるパーソナルコンピュータ (パソコン) や携帯電話のようなコンピュータシステムは、高度で複雑すぎて「ノイマン型コンピュータ」の原理を学ぶための教材としては適していません。

原理を学ぶのに適した単純で小さなコンピュータ (マイコン) を、専用に開発しました。このマイコンは TeC (Tokuyama Educational Computer : 徳山高専教育用コンピュータ、図 1.1) と呼ばれます。

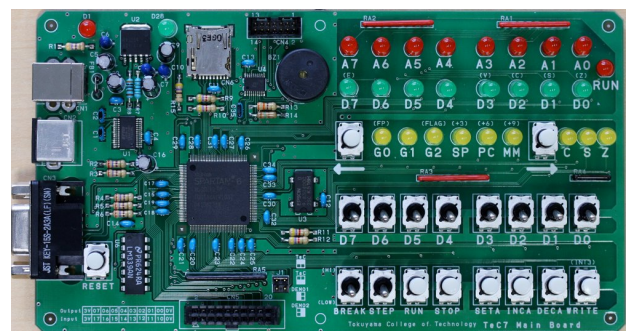


図 1.1 教育用コンピュータ TeC

TeC の特徴は単純で小さなことです。単純で小さなことで、次のようなメリットがあります。

単純 ノイマン型コンピュータの本質的な部分だけを勉強しやすい。現代のパソコン等は、勉強するには難しすぎる。

小型 自宅に持ち帰り宿題ができる。授業時間外でも、納得がいくまで色々と試してみることができる。

TeC には 2003 年から使用している TeC6 と、2011 年から使用を開始した TeC7 の二種類があります。本書では TeC7 を基本に解説しています。TeC6 について知りたい場合は本書の古いバージョン、「TeC 教科書 Ver. 3.2.2」(<https://github.com/tctsigemura/TecTextBook/blob/v3.2.2/tec.pdf>) を見て下さい。

## 第2章

# 情報の表現

この章では、コンピュータ内部でどのように情報が表現されているのか、その情報をどのような回路で扱うことができるのか、簡単に紹介します。

### 2.1 コンピュータ内部の情報表現

人は、情報を音声や文字、絵等で表現することができます。コンピュータも表面的には音声や文字、絵を扱うことができますが、コンピュータの内部は電子回路で構成されているので、最終的には電圧や電流で情報を表現せざるを得ません。

$$\text{情報の表現} = \left( \begin{array}{l} \text{人：音声，文字，絵，...} \\ \text{コンピュータ：電圧，電流} \end{array} \right)$$

#### 2.1.1 電氣を用いた情報の表現

電圧や電流で情報表現する方法には、いろいろなアイデアがあります。しかし、現代のコンピュータは、電圧がある/ない (ON/OFF) か、電流が流れている/流れていない (ON/OFF) のような、二つの状態だけを用いて情報を表現する方法を使っています。二つの状態だけを考えれば良いので回路が作りやすいからです。

図 2.1 を見て下さい。電氣の「ON/OFF」で「おおかみが来たか/来ていないか」のどちらの状態なのかを伝達できる「情報の表示装置」を実現することができます。電氣の「ON/OFF」を用いて情報を表現することができました。

#### 2.1.2 ビット

前の例では、ランプの「ON/OFF」を用いて「二つの状態のどちらなのか」を表しました。このような、「二つのどちらか」を表す情報が「情報の最小単位」になります。情報の最小単位のことを「ビット (bit)」と呼びます。

on/off のどちらか → 情報の最小単位 (ビット)

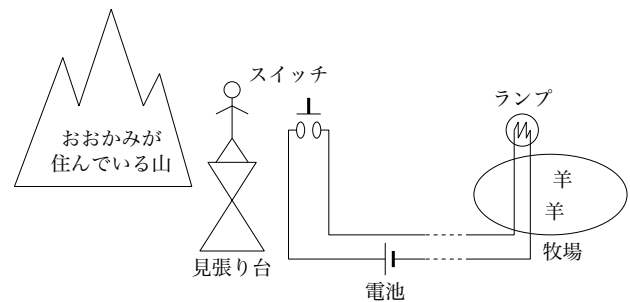


図 2.1 おおかみが来た情報表示装置

表 2.1 おおかみが来た情報

| ビット値   | 意味         |
|--------|------------|
| 0(off) | おおかみは来ていない |
| 1(on)  | おおかみが来た！！  |

通常、ビットの値は「ON/OFF」ではなく、「1/0」で書きます。

$$\left( \begin{array}{ll} \text{ON} & : 1 \\ \text{OFF} & : 0 \end{array} \right)$$

「おおかみが来た情報」は、ビットの値に表 2.1 のように対応付けできます。

#### 2.1.3 より複雑な情報の表現

二つの状態では表現できない、もっと複雑な情報はどうやって表現したら良いでしょうか。

前の例で、やって来たおおかみの頭数により牧場の人に対応を変化させたい場合が考えられます。そのためには、「来たか/来なかったか」だけの情報だけでは十分ではなく、「たくさん来たか」を知らせる必要があります。それには、複数のビットを組み合わせて使用します。

表 2.2 に複数のビットの組み合わせて表現した「拡張おおかみが来た情報」を、図 2.2 に「拡張おおかみが来

表 2.2 拡張おおかみが来た情報

| ビット値 | 意 味               |
|------|-------------------|
| 00   | おおかみはきていない (平気)   |
| 01   | おおかみが1頭来た (戦う)    |
| 10   | おおかみが2頭来た (?)     |
| 11   | おおかみがたくさん来た (逃げる) |

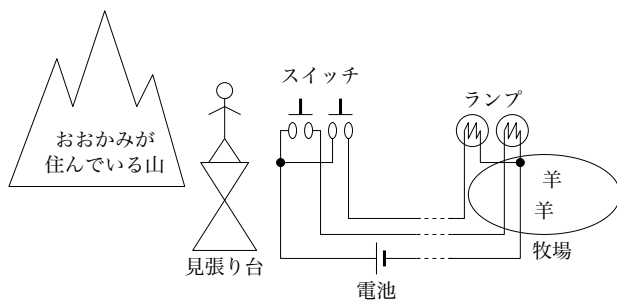


図 2.2 拡張おおかみが来た情報表示装置

表 2.3 ビットの組合せの数

| ビット数 | ビットの組合せ                            | 組合せ数  |
|------|------------------------------------|-------|
| 1    | 0 1                                | 2     |
| 2    | 00 01 10 11                        | 4     |
| 3    | 000 001 010 011<br>100 101 110 111 | 8     |
| ...  | ...                                |       |
| n    |                                    | $2^n$ |

た情報表示装置」を示します。

このように2ビット用いれば4種類の情報を表現することができます。表 2.3 はビット数と表現できる組合せの関係を示したものです。一般に、 $n$ ビットを用いると  $2^n$  種類の情報を表現することができます。システム内で必要なビット数を決めて、その組合せに意味付けをすれば、どんな情報だって表現できます。

ビットだけでは情報の単位として小さすぎるので、4ビットまとめたもの、8ビットまとめたものにも名前があります。「4ビット」を「1 ニブル」, 「8ビット」を「1 バイト」と呼びます。

## 2.2 数値の表現

ビットの組合せをどのように意味付けするかは、前節の例のように「システムが扱う必要のある情報」により、毎回、約束すればよいのです。しかし、どのコンピュー

表 2.4 4 ビットの2進数

| bit 3<br>( $b_3$ ) | bit 2<br>( $b_2$ ) | bit 1<br>( $b_1$ ) | bit 0<br>( $b_0$ ) | 意味 |
|--------------------|--------------------|--------------------|--------------------|----|
| 0                  | 0                  | 0                  | 0                  | 0  |
| 0                  | 0                  | 0                  | 1                  | 1  |
| 0                  | 0                  | 1                  | 0                  | 2  |
| 0                  | 0                  | 1                  | 1                  | 3  |
| 0                  | 1                  | 0                  | 0                  | 4  |
| 0                  | 1                  | 0                  | 1                  | 5  |
| 0                  | 1                  | 1                  | 0                  | 6  |
| 0                  | 1                  | 1                  | 1                  | 7  |
| 1                  | 0                  | 0                  | 0                  | 8  |
| 1                  | 0                  | 0                  | 1                  | 9  |
| 1                  | 0                  | 1                  | 0                  | 10 |
| 1                  | 0                  | 1                  | 1                  | 11 |
| 1                  | 1                  | 0                  | 0                  | 12 |
| 1                  | 1                  | 0                  | 1                  | 13 |
| 1                  | 1                  | 1                  | 0                  | 14 |
| 1                  | 1                  | 1                  | 1                  | 15 |

タでも同じ方法で意味付けされている情報もあります。その一つが数値の表現方法です。

### 2.2.1 2進数

コンピュータの内部では、数値を2進数で表すのが普通です。我々が普段使用している10進数と、2進数の特徴比較を次に示します。

#### 10進数の特徴

- (1) 0～9の10種類の数字を使用する。
- (2) 1桁毎に10倍の重みをもつ。

#### 2進数の特徴

- (1) 0と1の2種類の数字を使用する。
- (2) 1桁毎に2倍の重みをもつ。

コンピュータの内部では、ビットが情報の表現に使用されています。そこで、ビット値の0/1をそのまま2進数の1桁と考えれば数値が表現できます。

例えば、4ビット用いると0～15の数が表 2.4 のように表現できます。一般に  $n$  ビットで  $0 \sim 2^n - 1$  の範囲の数表現することができます。

### 2.2.2 2進数と10進数の相互変換

表 2.4 に示した4ビットの範囲なら、2進数と10進数の対応を暗記することが可能です。しかし、8ビットの場合ならどうでしょう？

組合せは256もあり、とても暗記できそうにありません。対応を計算で求める必要があります。

$$\begin{array}{rcl}
10_{10} & = & 1010_2 \\
\div 2 \downarrow & & \\
5_{10}^{\dots 0} & = & 0101_2^{\dots 0} \\
\div 2 \downarrow & & \\
2_{10}^{\dots 1} & = & 0010_2^{\dots 1} \\
\div 2 \downarrow & & \\
1_{10}^{\dots 0} & = & 0001_2^{\dots 0} \\
\div 2 \downarrow & & \\
0_{10}^{\dots 1} & = & 0000_2^{\dots 1}
\end{array}$$

図 2.3 数値を 2 で割った時の余り

## 1. 2 進から 10 進への変換

2 進数の桁ごとの重みは、桁の番号を  $n$  とすると  $2^n$  になります。

$$\begin{array}{cccc}
b_3 & b_2 & b_1 & b_0 \\
2^3 = 8 & 2^2 = 4 & 2^1 = 2 & 2^0 = 1
\end{array}$$

2 進数の数値は、その桁の重みと桁の値を掛け合わせたものの合計です。例えば 2 進数の  $1010_2$  は、 $2^3$  の桁が 1、 $2^2$  の桁が 0、 $2^1$  の桁が 1、 $2^0$  の桁が 0 ですから、次のように計算できます。

$$\begin{aligned}
1010_2 &= 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 0 \\
&= 8 \times 1 + 4 \times 0 + 2 \times 1 + 1 \times 0 \\
&= 8 + 0 + 2 + 0 \\
&= 10_{10}
\end{aligned}$$

## 2. 10 進から 2 進への変換

次に、10 進数を 2 進数に変換する方法を考えます。ここで着目するのは桁の移動です。

図 2.3 に例を示すように、2 進数を 2 で割ると右に 1 桁移動します。その時の余りは最下位の桁からはみ出した数になります。同じ値の 10 進数を 2 で割っても余りは同じです。

つまり、数値を 2 で割った時の余りは 2 進数を右に 1 桁移動したときはみ出してきた数を表しています。そこで図 2.4 のように、2 で割る操作を繰り返しながらはみ出して来た数を記録すれば、もとの数を 2 進数で表したときの 0/1 の並びが分かります。

## 2.2.3 16 進数

2 進数は、桁数が多くなり書き表すのに不便です。そこで、表 2.5 のように 2 進数 4 桁をまとめて 16 進数 1 桁で書き表します。9 より大きな数字が無いので、16 進数ではアルファベットを数字の代用にします。

$$\begin{array}{rcl}
2) \ 10 & & \\
2) \ 5 \dots 0 & & \text{余りを右から順に並べる} \\
2) \ 2 \dots 1 & & \text{と } 1010_2 \\
2) \ 1 \dots 0 & & \\
\hline
0 \dots 1 & &
\end{array}$$

図 2.4 10 を 2 進数に変換する例

表 2.5 2 進数, 16 進数, 10 進数対応表

| 2 進数              | 16 進数           | 10 進数            |
|-------------------|-----------------|------------------|
| 0000 <sub>2</sub> | 0 <sub>16</sub> | 0 <sub>10</sub>  |
| 0001 <sub>2</sub> | 1 <sub>16</sub> | 1 <sub>10</sub>  |
| 0010 <sub>2</sub> | 2 <sub>16</sub> | 2 <sub>10</sub>  |
| 0011 <sub>2</sub> | 3 <sub>16</sub> | 3 <sub>10</sub>  |
| 0100 <sub>2</sub> | 4 <sub>16</sub> | 4 <sub>10</sub>  |
| 0101 <sub>2</sub> | 5 <sub>16</sub> | 5 <sub>10</sub>  |
| 0110 <sub>2</sub> | 6 <sub>16</sub> | 6 <sub>10</sub>  |
| 0111 <sub>2</sub> | 7 <sub>16</sub> | 7 <sub>10</sub>  |
| 1000 <sub>2</sub> | 8 <sub>16</sub> | 8 <sub>10</sub>  |
| 1001 <sub>2</sub> | 9 <sub>16</sub> | 9 <sub>10</sub>  |
| 1010 <sub>2</sub> | A <sub>16</sub> | 10 <sub>10</sub> |
| 1011 <sub>2</sub> | B <sub>16</sub> | 11 <sub>10</sub> |
| 1100 <sub>2</sub> | C <sub>16</sub> | 12 <sub>10</sub> |
| 1101 <sub>2</sub> | D <sub>16</sub> | 13 <sub>10</sub> |
| 1110 <sub>2</sub> | E <sub>16</sub> | 14 <sub>10</sub> |
| 1111 <sub>2</sub> | F <sub>16</sub> | 15 <sub>10</sub> |

何進数で書いてあるのかをはっきりさせるために、表 2.5 のように数値の右に小さな字で何進数かを書き加えます。ときには、数字の代わりに「2 進数=’b’」,「16 進数=’H’」を加えることもあります。例えば、「01100100<sub>2</sub>」を 01100100<sub>b</sub>、「64<sub>16</sub>」を 64H のように書きます。

## 問題

- 表 2.5 「2 進数, 16 進数, 10 進数対応表」を暗記しなさい。
- 10 進数の 16, 50, 100, 127, 130 を、2 進数 (8 桁), 16 進数 (2 桁) で書き表しなさい。
- 2 進数の 00011100, 00111000, 11100000 を 16 進数 (2 桁), 10 進数で書き表しなさい。
- 16 進数の 1F と AA を 2 進数 (8 桁) で書き表しなさい。また、10 進数で書き表しなさい。

表 2.6 符号付き絶対値表現 (4 ビット)

| 負数  | 2 進数              | 正数  | 2 進数              |
|-----|-------------------|-----|-------------------|
| -7  | 1111 <sub>2</sub> | +7  | 0111 <sub>2</sub> |
| -6  | 1110 <sub>2</sub> | +6  | 0110 <sub>2</sub> |
| -5  | 1101 <sub>2</sub> | +5  | 0101 <sub>2</sub> |
| ... | ...               | ... | ...               |
| -1  | 1001 <sub>2</sub> | +1  | 0001 <sub>2</sub> |
| -0  | 1000 <sub>2</sub> | +0  | 0000 <sub>2</sub> |

 $b = 10$  進数

|             |     |             |
|-------------|-----|-------------|
| $n = 2$ 桁   | 100 |             |
| $b^n = 100$ | -25 | 75 は 25 に対す |
| $x = 25$    | 75  | る 10 の補数    |

 $b = 10$  進数

|                |     |             |
|----------------|-----|-------------|
| $n = 2$ 桁      | 99  |             |
| $b^n - 1 = 99$ | -25 | 74 は 25 に対す |
| $x = 25$       | 74  | る 9 の補数     |

図 2.5 2 桁の 10 進数で補数の例

## 2.3 負数の表現

拡張おおかみが来た情報表示装置では、「表示装置の二つのランプ (2 ビット) の読み方」を約束しました。つぎに、数値の場合は、「 $n$  個のビットを 2 進数として読む」ことを約束しました。

今度は、負の数が必要になりました。そこで、ビットの新しい読み方を約束します。この節で出てくるビットは「符号付き数値」を表しています。以下では、「符号付き数値を表すビット」をどのように読むかを説明します。

### 2.3.1 符号付き絶対値表現

使用できるビットのうち一つを符号を表すために使います。これを「符号ビット」と呼ぶことにします。通常、符号ビットには最上位 (左端) のビットを使用します。

表 2.6 のように 4 ビット使用して、 $-7 \sim +7$  の範囲を表すことができます。この表現方法は分かりやすく都合が良いのですが、実際に使われることはあまりありません。

### 2.3.2 補数表現

$n$  桁の  $b$  進数において  $b^n$  から  $x$  を引いた数  $y$  を  $x$  に対する「 $b$  の補数」と言います。 $n$  桁の  $b$  進数でにおいて  $b^n - 1$  から  $x$  を引いた数  $z$  を  $x$  に対する「 $(b-1)$  の補数」と言います。

$$y = b^n - x \quad (y \text{ は } x \text{ に対する } b \text{ の補数})$$

$$z = b^n - 1 - x \quad (z \text{ は } x \text{ に対する } (b-1) \text{ の補数})$$

10 進数の例を図 2.5 に、2 進数の例を図 2.6 に示します。

### 2.3.3 1 の補数による負数の表現

図 2.6 「4 桁の 2 進数で補数の例」にあるように、 $11...1_2$  からもとの数 ( $x$ ) を引いた数を  $x$  に対する「1 の補数」と呼びます。表 2.7 は  $0 \sim 7$  に対する 1 の補数

|                 |                    |                     |
|-----------------|--------------------|---------------------|
| $b = 2$ 進数      |                    | 0110 <sub>2</sub> は |
| $n = 4$ 桁       | 10000 <sub>2</sub> | 1010 <sub>2</sub> に |
| $b^n = 10000_2$ | -1010 <sub>2</sub> | 対する                 |
| $x = 1010_2$    | 0110 <sub>2</sub>  | 2 の補数               |

|                    |                    |                     |
|--------------------|--------------------|---------------------|
| $b = 2$ 進数         |                    | 0101 <sub>2</sub> は |
| $n = 4$ 桁          | 1111 <sub>2</sub>  | 1010 <sub>2</sub> に |
| $b^n - 1 = 1111_2$ | -1010 <sub>2</sub> | 対する                 |
| $x = 1010_2$       | 0101 <sub>2</sub>  | 1 の補数               |

図 2.6 4 桁の 2 進数で補数の例

表 2.7 4 ビット 2 進数の 1 補数

| もとの数 | 補数へ変換               | 補数                |
|------|---------------------|-------------------|
| 0    | $1111_2 - 0000_2 =$ | 1111 <sub>2</sub> |
| 1    | $1111_2 - 0001_2 =$ | 1110 <sub>2</sub> |
| 2    | $1111_2 - 0010_2 =$ | 1101 <sub>2</sub> |
| 3    | $1111_2 - 0011_2 =$ | 1100 <sub>2</sub> |
| 4    | $1111_2 - 0100_2 =$ | 1011 <sub>2</sub> |
| 5    | $1111_2 - 0101_2 =$ | 1010 <sub>2</sub> |
| 6    | $1111_2 - 0110_2 =$ | 1001 <sub>2</sub> |
| 7    | $1111_2 - 0111_2 =$ | 1000 <sub>2</sub> |

を計算したものです。

「0001<sub>2</sub> に対する 1 の補数を -0001<sub>2</sub> を表現するために使用する。」、「0010<sub>2</sub> に対する 1 の補数を -0010<sub>2</sub> を表現するために使用する。」つまり、「**1 の補数を負の数**を表現するために使用する。」と約束すれば、図 2.7 のように  $-7 \sim +7$  の範囲の数を表現できます。一般に、1 の補数表現を用いた  $n$  ビット符号付き 2 進数が表現できる数値の範囲は次の式で計算できます。

$$-(2^{n-1} - 1) \sim 2^{n-1} - 1$$

図 2.7 で +7 と -7 が結んでるのは、「互いに 1 の補数」であることを表すためです。また、「互いに 1 の補数」の 2 数を見比べるとビットの 0/1 が入れ替わった関



|    |                   |   |   |   |   |   |   |   |   |
|----|-------------------|---|---|---|---|---|---|---|---|
| -7 | 1000 <sub>2</sub> | - | - | - | - | - | - | - | + |
| -6 | 1001 <sub>2</sub> | - | - | - | - | - | - | + |   |
| -5 | 1010 <sub>2</sub> | - | - | - | - | - | + |   |   |
| -4 | 1011 <sub>2</sub> | - | - | - | - | + |   |   |   |
| -3 | 1100 <sub>2</sub> | - | - | - | + |   |   |   |   |
| -2 | 1101 <sub>2</sub> | - | - | + |   |   |   |   |   |
| -1 | 1110 <sub>2</sub> | - | + |   |   |   |   |   |   |
| -0 | 1111 <sub>2</sub> | + |   |   |   |   |   |   |   |
| +0 | 0000 <sub>2</sub> | + |   |   |   |   |   |   |   |
| +1 | 0001 <sub>2</sub> | - | + |   |   |   |   |   |   |
| +2 | 0010 <sub>2</sub> | - | - | + |   |   |   |   |   |
| +3 | 0011 <sub>2</sub> | - | - | - | + |   |   |   |   |
| +4 | 0100 <sub>2</sub> | - | - | - | - | + |   |   |   |
| +5 | 0101 <sub>2</sub> | - | - | - | - | - | + |   |   |
| +6 | 0110 <sub>2</sub> | - | - | - | - | - | - | + |   |
| +7 | 0111 <sub>2</sub> | - | - | - | - | - | - | - | + |

図 2.7 1 の補数を用いた符号付き数値

係になっていることが分かります。1 の補数は引算だけでなく「ビット反転」でも計算できます。

次に 1 の補数からもとの数を求める計算を考えてみましょう。以下のように 1 の補数 ( $z$ ) を求める式を  $x =$  の形に変形しても同じ形になるので、 $x$  と  $z$  は「互いに 1 の補数」です。1 の補数からもとの数を求める計算は、もとの数から 1 の補数を求める計算と同じです。(ビット反転のビット反転で元に戻る。)

$$z = 2^n - 1 - x \quad (z \text{ は } x \text{ に対する 1 の補数})$$

$$x = 2^n - 1 - z \quad (x \text{ は } z \text{ に対する 1 の補数})$$

1 の補数を使用した方法も、実際に使われることはあまりありません。コンピュータの内部で実際に使用されるのは、次に説明する 2 の補数による表現です。

#### 2.3.4 2 の補数による負数の表現

図 2.6 「4 桁の 2 進数で補数の例」にあるように、 $100...0_2$  からもとの数 ( $x$ ) を引いた数を  $x$  に対する「2 の補数」と呼びます。表 2.8 は 0 ～ 8 に対する 2 の補数を計算したものです。5 ビットで表現されている部分もありますが、四角で囲んだ 4 ビットに注目してください。

「0001<sub>2</sub> に対する 2 の補数を -0001<sub>2</sub> を表現するために使用する。」、「0010<sub>2</sub> に対する 2 の補数を -0010<sub>2</sub> を表現するために使用する。」つまり「2 の補数を負の数を表すために使用する。」と約束すれば、図 2.8 のように -8 ～ +7 の範囲の数を表現できます。(ただし 1000<sub>2</sub> は -8 を表現することとします。4 ビットでは +8 を表

表 2.8 4 ビット 2 進数の 2 補数

| もとの数 | 補数へ変換 |                   |   |                   | 補数                    |
|------|-------|-------------------|---|-------------------|-----------------------|
| 0    | 1     | 0000 <sub>2</sub> | - | 0000 <sub>2</sub> | = 1 0000 <sub>2</sub> |
| 1    | 1     | 0000 <sub>2</sub> | - | 0001 <sub>2</sub> | = 1 1111 <sub>2</sub> |
| 2    | 1     | 0000 <sub>2</sub> | - | 0010 <sub>2</sub> | = 1 1110 <sub>2</sub> |
| 3    | 1     | 0000 <sub>2</sub> | - | 0011 <sub>2</sub> | = 1 1101 <sub>2</sub> |
| 4    | 1     | 0000 <sub>2</sub> | - | 0100 <sub>2</sub> | = 1 1100 <sub>2</sub> |
| 5    | 1     | 0000 <sub>2</sub> | - | 0101 <sub>2</sub> | = 1 1011 <sub>2</sub> |
| 6    | 1     | 0000 <sub>2</sub> | - | 0110 <sub>2</sub> | = 1 1010 <sub>2</sub> |
| 7    | 1     | 0000 <sub>2</sub> | - | 0111 <sub>2</sub> | = 1 1001 <sub>2</sub> |
| 8    | 1     | 0000 <sub>2</sub> | - | 1000 <sub>2</sub> | = 1 1000 <sub>2</sub> |

|    |                   |   |   |   |   |   |   |   |   |
|----|-------------------|---|---|---|---|---|---|---|---|
| -8 | 1000 <sub>2</sub> |   |   |   |   |   |   |   |   |
| -7 | 1001 <sub>2</sub> | - | - | - | - | - | - | - | + |
| -6 | 1010 <sub>2</sub> | - | - | - | - | - | - | + |   |
| -5 | 1011 <sub>2</sub> | - | - | - | - | - | + |   |   |
| -4 | 1100 <sub>2</sub> | - | - | - | - | + |   |   |   |
| -3 | 1101 <sub>2</sub> | - | - | - | + |   |   |   |   |
| -2 | 1110 <sub>2</sub> | - | - | + |   |   |   |   |   |
| -1 | 1111 <sub>2</sub> | - | + |   |   |   |   |   |   |
| 0  | 0000 <sub>2</sub> | + |   |   |   |   |   |   |   |
| 1  | 0001 <sub>2</sub> | - | + |   |   |   |   |   |   |
| 2  | 0010 <sub>2</sub> | - | - | + |   |   |   |   |   |
| 3  | 0011 <sub>2</sub> | - | - | - | + |   |   |   |   |
| 4  | 0100 <sub>2</sub> | - | - | - | - | + |   |   |   |
| 5  | 0101 <sub>2</sub> | - | - | - | - | - | + |   |   |
| 6  | 0110 <sub>2</sub> | - | - | - | - | - | - | + |   |
| 7  | 0111 <sub>2</sub> | - | - | - | - | - | - | - | + |

図 2.8 2 の補数を用いた符号付き数値

現することはできません。)

一般に、2 の補数表現を用いた  $n$  ビット符号付き 2 進数が表現できる数値の範囲は次の式で計算できます。

$$-2^{n-1} \sim 2^{n-1} - 1$$

次に 2 の補数からもとの数を求める計算を考えてみましょう。以下のように 2 の補数 ( $z$ ) を求める式を  $x =$  の形に変形しても同じ形になるので、 $x$  と  $z$  は「互いに 2 の補数」です。2 の補数からもとの数を求める計算は、もとの数から 2 の補数を求める計算と同じです。

$$z = 2^n - x \quad (z \text{ は } x \text{ に対する 2 の補数})$$

$$x = 2^n - z \quad (x \text{ は } z \text{ に対する 2 の補数})$$

現代のコンピュータは、ほとんどの機種で 2 の補数表現を採用しています。2 の補数表現を用いると 2 進数の足し算・引き算が、正負のどちらの数でも同じ手順で計

| 10 進数 | 2 進数                |            |
|-------|---------------------|------------|
| 07    | 0111 <sub>2</sub>   | (10 進の 7)  |
| + 05  | + 0101 <sub>2</sub> | (10 進の 5)  |
| 12    | 1100 <sub>2</sub>   | (10 進の 12) |
| 12    | 1100 <sub>2</sub>   | (10 進の 12) |
| - 05  | - 0101 <sub>2</sub> | (10 進の 5)  |
| 07    | 0111 <sub>2</sub>   | (10 進の 7)  |

何進数で計算しても同じ結果になる。

図 2.9 2 進数と 10 進数の計算を比較

**算できます** (詳しくは後述)。「手順が同じ＝演算回路が同じ」ことになりますので、コンピュータを製作する上では非常に都合がよいのです。

### 2.3.5 2 の補数を求める手順

$-x$  を  $n$  ビット 2 の補数表現 ( $y$ ) で表します。2 の補数の定義より、 $y$  は次のように計算できます。

$$\begin{aligned}
 y &= 2^n - x \\
 &= (2^n - 1 - x) + 1 \\
 &= (x \text{ に対する } 1 \text{ の補数}) + 1
 \end{aligned}$$

「 $x$  に対する 1 の補数」はビット反転で簡単に求めることができます。「 $x$  に対する 2 の補数」は「 $x$  をビット反転した後、1 を加える」ことにより簡単に求めることができます。

ビット反転した後、1 を加える。

### 2.3.6 2 の補数から元の数を求める手順

次に、逆変換について考えます。既に 2.3.4 で触れたように 2 の補数と元の数は互いに「2 の補数」ですから、2 の補数を求めるのと同じ計算で、2 の補数から元の数を求めることができます。

ビット反転した後、1 を加える。

## 2.4 2 進数の計算

ここでは、2 進数の和と差の計算方法を学びます。

### 2.4.1 正の数の計算

2 進数の計算も 10 進数と同じ要領です。桁上がりが 10 ではなく、2 で発生することに注意してください。図 2.9 に例を示します。2 進数で計算しても 10 進数で計算しても同じ計算結果になります。

$3 + (-5) = -2$  (4 ビット 2 の補数表現)

3 を 2 進数に変換すると 0011<sub>2</sub>  
 $-5$  を 2 進数に変換すると 1011<sub>2</sub>

「和を計算する」

$$0011_2 + 1011_2 = 1110_2 = -2_{10}$$

図 2.10 結果が負になる足し算の例

$5 + (-3) = 2$  (4 ビット 2 の補数表現)

5 を 2 進数に変換すると 0101<sub>2</sub>  
 $-3$  を 2 進数に変換すると 1101<sub>2</sub>

「和を計算する」

$$0101_2 + 1101_2 = \boxed{1}0010_2 = 2_{10}$$

(5 ビット目の 1 は桁あふれで消滅)

図 2.11 結果が正になる足し算の例

### 2.4.2 負の数を含む計算

2 の補数表現を用いると、正の数だけのときと同じ要領で負の数を含む計算ができます。ここでは和の計算を例に説明します。

#### 正の数と負の数の和

正の数 ( $X$ ) と負の数 ( $-A$ ) ( $A$  の 2 の補数 ( $B$ )) の和の計算を考えます。

$X + B$  の計算 ( $X + (-A)$ )

1. 結果が負の場合 ( $|A| > |X|$ )  
 解は  $-(A - X)$  になるはず！

$$\begin{aligned}
 X + B &= X + (2^n - A) \\
 &= 2^n - (A - X)
 \end{aligned} \tag{2.1}$$

(2.1) 式は、正解  $-(A - X)$  の 2 の補数表現になっています。図 2.10 に計算例を示します。

2. 結果が正またはゼロの場合 ( $|X| \geq |A|$ )  
 解は  $(X - A)$  になるはず！

$$\begin{aligned}
 X + B &= X + (2^n - A) \\
 &= 2^n + (X - A) \\
 &= X - A
 \end{aligned} \tag{2.2}$$

(2.2) 式の  $2^n$  は、桁あふれにより結果に残らないので、正解  $(X - A)$  となります。図 2.11 に計算例を示します。



$$(-1) + (-3) = (-4) \text{ (4ビット2の補数表現)}$$

-1 を 2 進数に変換すると  $1111_2$

-3 を 2 進数に変換すると  $1101_2$

「和を計算する」

$$1111_2 + 1101_2 = \boxed{1}1100_2 = -4_{10}$$

(5 ビット目の 1 は桁あふれで消滅)

図 2.12 負の数と負の数の足し算の例

## 負の数と負の数の和

負の数  $(-A_1)$  (2 の補数  $(B_1)$ ) と負の数  $(-A_2)$  (2 の補数  $(B_2)$ ) の和
$$B_1 + B_2 \text{ の計算 } ((-A_1) + (-A_2))$$

解は  $-(A_1 + A_2)$  になるはず！

$$\begin{aligned} B_1 + B_2 &= (2^n - A_1) + (2^n - A_2) \\ &= 2^n + (2^n - (A_1 + A_2)) \end{aligned} \quad (2.3)$$

$$= 2^n - (A_1 + A_2) \quad (2.4)$$

(2.3) 式の最初の  $2^n$  は、桁あふれにより消滅するので (2.4) 式は正解  $-(A_1 + A_2)$  の 2 の補数表現になっています。図 2.12 に計算例を示します。

以上のように、2の補数表現を使用すると負の数を含んだ足し算が簡単に計算できます。ここでは省略しますが、引き算も同様に正負の区別をすることなく計算できます。

## 問題

1. 次の数を「2 の補数表現を用いた 4 ビット符号付き 2 進数」で書き表しなさい。  
 $3_{10}, -3_{10}, 5_{10}, -5_{10}, 6_{10}, -6_{10}$
2. 次の数を「2 の補数表現を用いた 8 ビット符号付き 2 進数」で書き表しなさい。  
 $3_{10}, -3_{10}, 8_{10}, -8_{10},$   
 $30_{10}, -30_{10}, 100_{10}, -100_{10}$
3. 次の「2 の補数表現を用いた 4 ビット符号付き 2 進数」を 10 進数で書き表しなさい。  
 $1100_2, 1011_2, 0100_2, 1101_2$
4. 次の 2 進数は「2 の補数表現を用いた符号付き 2 進数」です。空欄を埋めなさい。

1) 
$$\begin{array}{r} 0011\ 1100_2 \\ +\ 0010\ 1101_2 \\ \hline \end{array} \rightarrow \begin{array}{r} \boxed{\phantom{00}}_{10} \\ \boxed{\phantom{00}}_{10} \\ \hline \boxed{\phantom{00}}_{10} \end{array}$$

2) 
$$\begin{array}{r} 0110\ 0100_2 \\ +\ 1000\ 0001_2 \\ \hline \end{array} \rightarrow \begin{array}{r} \boxed{\phantom{000}}_{10} \\ +\ \boxed{\phantom{000}}_{10} \\ \hline \boxed{\phantom{000}}_{10} \end{array}$$

3) 
$$\begin{array}{r} 1110\ 0100_2 \\ +\ 0100\ 0001_2 \\ \hline \end{array} \rightarrow \begin{array}{r} \boxed{\phantom{000}}_{10} \\ +\ \boxed{\phantom{000}}_{10} \\ +\ \boxed{\phantom{000}}_{10} \\ \hline \end{array}$$

4) 
$$\begin{array}{r} 1110\ 0100_2 \\ +\ 1100\ 0001_2 \\ \hline \end{array} \rightarrow \begin{array}{r} \boxed{\phantom{000}}_{10} \\ \boxed{\phantom{000}}_{10} \\ + \\ \boxed{\phantom{000}}_{10} \end{array}$$

## 2.5 小数の表現

この節では、小数を 2 進数で表現する方法を紹介します。

### 2.5.1 2進数による小数の表現

コンピュータ内部で、2進数を用いて、小数を含む数を表現する方法は何種類あります。ここでは、その中でも最も基本的な固定小数点形式を紹介します。

これまでの 2 進数は、暗黙のうちに小数点が右端にあると考えました。小数点の位置を右端以外だと約束すれば、小数を含む数の表現も可能になります。以下に 4 ビットの 2 桁目に小数点があると考えた場合の例を示します。

$$00.00_2 = 0.0_{10}$$

$$00.01_2 = 0.25_{10}$$

$$00.10_2 = 0.5_{10}$$

$$00.11_2 = 0.75_{10}$$

$$01.00_2 = 1.0_{10}$$

$$01.01_2 = 1.25_{10}$$

$$01.10_2 = 1.5_{10}$$

$$01.11_2 = 1.75_{10}$$

$$10.00_2 = 2.0_{10}$$

...

$$11.11_2 = 3.75_{10}$$

小数点を中心に、左は1桁毎に2倍、右は1桁毎に1/2倍になります。(10進数では、左は1桁毎に10倍、右は1桁毎に1/10倍になりましたね。)

### 2.5.2 10 進数との相互変換

- ## 1. 2進数から10進数への変換

整数の場合と同様に桁の重みを加算すれば 10 進数に変換できます。次に例を示します。

$$10.01_2 = 2 + 1/4 = 2.25_{10}$$

- ## 2. 10進数から2進数への変換

整数の場合は、 $1/2$  倍することで右にシフト (桁移

|                    |         |       |
|--------------------|---------|-------|
| 2進数                | ×2は     | 10進数  |
| 0.101 <sub>2</sub> | 左シフトと同じ | 0.625 |
| ✓                  |         | × 2   |
| 1.010 <sub>2</sub> |         | 1.250 |
|                    |         |       |
| 2進数                | ×2は     | 10進数  |
| 0.010 <sub>2</sub> | 左シフトと同じ | 0.250 |
| ✓                  |         | × 2   |
| 0.100 <sub>2</sub> |         | 0.500 |
|                    |         |       |
| 2進数                | ×2は     | 10進数  |
| 0.100 <sub>2</sub> | 左シフトと同じ | 0.500 |
| ✓                  |         | × 2   |
| 1.000 <sub>2</sub> |         | 1.000 |

10進数で計算したとき、小数点を横切って整数部に出てきた数を小数点の右に順番に並べると 0.101<sub>2</sub> になる。

図 2.13 小数を表す 10 進数から 2 進数への変換例

動) し、小数点を横切った 0/1 を判定しました。(小数点を 1 が横切ると、余りが出ていました。)

図 2.13 に例を示すように、小数点以下の数値は、2 倍することで左にシフトし、小数点を横切った 0/1 を判定すれば 2 進数に変換できます。

### 3. 整数部と小数部の両方がある場合

整数部分と小数部分を分離して、別々に計算します。

### 問題

- 次の 2 進数を 10 進数に変換しなさい。
  - 0.1001<sub>2</sub>
  - 0.0101<sub>2</sub>
  - 11.11<sub>2</sub>
- 次の 10 進数を 2 進数に変換しなさい。
  - 0.0625<sub>10</sub>
  - 0.1875<sub>10</sub>
  - 0.4375<sub>10</sub>
- 次の 10 進数を 2 進数に変換しなさい。(難問)
  - 0.8<sub>10</sub>
  - 0.7<sub>10</sub>

## 2.6 文字の表現

この節では、文字を 2 進数で表現する方法を紹介します。(文字情報を表しているビットの読み方を約束する。)

### MSB と LSB

「2 の補数表現を用いると、最上位ビットが 1 なら負の数と分かります。」のように**最上位ビット**と言う言葉をよく使います。

「最上位ビット」と言うのは長いので、これを英語にした「Most Significant Bit」の頭文字を取り MSB と略することが良くあります。覚えておいて下さい。

同様に「最下位ビット」のことは英語で「Least Significant Bit」なので、略して LSB と呼びます。これも、覚えておきましょう。

### 2.6.1 文字コード

文字の場合、数値のような規則性を期待することはできません。そこで、使用する文字の一覧表を作成し、1 文字毎に対応する 2 進数(コード)を決めます。この一覧表のことを「文字コード表」と呼びます。文字コード表を各自(コンピュータメーカ等)が勝手に定義すると、コンピュータの間でのデータ交換に不便です。そこで、規格として制定してあります。

### 2.6.2 ASCII コード

ASCII(American Standard Code for Information Interchange) コードは、1963 年にアメリカ規格協会(ANSI)が定めた情報交換用の文字コードです。英字、数字、記号等が含まれます。現代のパーソナルコンピュータ等で使用される文字コードは、ASCII コードが基本になっています。図 2.14 に ASCII 文字コード表を示します。

文字コード表で 00H ~ 1FH と 7FH は、機能(改行等)を表す特殊な文字になっています。20H(SP)は空白を表す文字です。

ASCII 文字コードは 7 ビットで表現されます。しかし、コンピュータの内部では 1 バイト(8 ビット)単位の方が扱いやすいので、最上位に 0<sub>2</sub> を補って 8 ビットで扱うことがほとんどです。

### 2.6.3 JIS 文字コード

日本では JIS(Japan Industrial Standard: 日本工業規格)の一部として、8 ビットコード(英数記号+カナ)と、16 ビットコード(英数記号+カナ+ひらがな+カタカナ+漢字...)が定められています。

JIS 8 ビットコードは、ASCII コードに半角カタカナ

|          |   | (上位3ビット) |          |    |   |   |   |   |     |
|----------|---|----------|----------|----|---|---|---|---|-----|
| (下位4ビット) |   | 0        | 1        | 2  | 3 | 4 | 5 | 6 | 7   |
|          | 0 | NUL      | DLE (SP) | 0  | @ | P | ` | p |     |
|          | 1 | SOH      | DC1      | !  | 1 | A | Q | a | q   |
|          | 2 | STX      | DC2      | "  | 2 | B | R | b | r   |
|          | 3 | ETX      | DC3      | #  | 3 | C | S | c | s   |
|          | 4 | EOT      | DC4      | \$ | 4 | D | T | d | t   |
|          | 5 | ENQ      | NAK      | %  | 5 | E | U | e | u   |
|          | 6 | ACK      | SYN      | &  | 6 | F | V | f | v   |
|          | 7 | BEL      | ETB      | '  | 7 | G | W | g | w   |
|          | 8 | BS       | CAN      | (  | 8 | H | X | h | x   |
|          | 9 | HT       | EM       | )  | 9 | I | Y | i | y   |
|          | A | LF       | SUB      | *  | : | J | Z | j | z   |
|          | B | VT       | ESC      | +  | ; | K | [ | k | {   |
|          | C | FF       | FS       | ,  | < | L | \ | l |     |
|          | D | CR       | GS       | -  | = | M | ] | m | }   |
|          | E | SO       | RS       | .  | > | N | ^ | n | ~   |
|          | F | SI       | US       | /  | ? | O | _ | o | DEL |

図 2.14 ASCII 文字コード表

を追加したものです。記号、数字、英字の部分は、ほぼ、同じ並びになっています。

## 2.7 補助単位

長さや重さを書くとき、1,000m を 1km、1,000g を 1kg のように書きました。この k のような記号を補助単位と呼びます。

コンピュータの世界でよく使用される補助単位には、k(キロ=10<sup>3</sup>)、M(メガ=10<sup>6</sup>)、G(ギガ=10<sup>9</sup>)、T(テラ=10<sup>12</sup>) 等があります。(1,000 倍毎に補助単位があります。) パソコンのカタログに「CPU のクロックは 1GHz」のような記述を見かけますね。よく使う補助単位を表 2.9 にまとめます。

記憶容量を表す場合の補助単位は、1,000 の代わりに 2<sup>10</sup> = 1,024 を使用します。1,000 も 1,024 も k で表現すると紛らわしいので、k の代わりに Ki と書き表すこともあります。つまり、1kB = 1KiB = 2<sup>10</sup>B = 1,024B、1MB = 1MiB = 2<sup>20</sup>B = 1,048,576B となります。(「B」はバイトを表す。)

「H.D.D. の容量は 500GB」のように表示されている場合は、記憶容量を表しているので G = 2<sup>30</sup> で表示されています。同じことを「H.D.D. の容量は 500GiB」のように表示することもあります。

表 2.9 補助単位まとめ

| 一般的に             |    |     | 記憶容量            |    |     |
|------------------|----|-----|-----------------|----|-----|
| 値                | 記号 | 読み方 | 値               | 記号 | 読み方 |
| 10 <sup>3</sup>  | k  | キロ  | 2 <sup>10</sup> | Ki | キビ  |
| 10 <sup>6</sup>  | M  | メガ  | 2 <sup>20</sup> | Mi | メビ  |
| 10 <sup>9</sup>  | G  | ギガ  | 2 <sup>30</sup> | Gi | ギビ  |
| 10 <sup>12</sup> | T  | テラ  | 2 <sup>40</sup> | Ti | テビ  |

## 2.8 コンピュータの基本回路

前の節までで、コンピュータ内部での情報の表現方法を勉強しました。コンピュータの内部では、どんな情報も電気の「ON/OFF」で表現しているのです。

この節では、電気の「ON/OFF」を使用して、計算や記憶をする回路を勉強します。

### 2.8.1 論理演算と論理回路

論理 (Yes/No, 真/偽, True/False) を対象とする演算 (計算) のことを**論理演算**と呼びます。論理の「真/偽」をビットの「1/0」に対応させることにより、論理も電気の「ON/OFF」で表現することができます。

ここでは、論理演算と、論理演算をする回路を紹介します。論理演算をする回路のことを**論理回路**と呼びます。ここで紹介する論理回路が、コンピュータを構成する基本回路になります。

### 2.8.2 基本的な論理回路

基本的な論理回路を 4 種類、組合せてできたものを 2 種類、紹介します。

#### 1. 論理積 (AND)

二つのビットを入力し、両方が 1 のときだけ出力が 1 になるような論理演算です。

| 入力 |   | 出力 |
|----|---|----|
| A  | B | X  |
| 0  | 0 | 0  |
| 0  | 1 | 0  |
| 1  | 0 | 0  |
| 1  | 1 | 1  |

ANDの真理値表

$X = A \cdot B$   
ANDの論理式

A  B = X

ANDの回路記号

#### 2. 論理和 (OR)

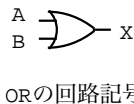
二つのビットを入力し、どちらかが 1 のとき出力が 1 になるような論理演算です。

| 入力 |   | 出力 |
|----|---|----|
| A  | B | X  |
| 0  | 0 | 0  |
| 0  | 1 | 1  |
| 1  | 0 | 1  |
| 1  | 1 | 1  |

ORの真理値表

$$X = A + B$$

ORの論理式



### 3. 否定 (NOT)



一つのビットを入力し、入力と逆の論理を出力する論理演算です。



| 入力 | 出力 |
|----|----|
| A  | X  |
| 0  | 1  |
| 1  | 0  |

NOTの真理値表

$$\mathbf{X = \overline{A}}$$

NOTの論理式

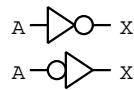

 A  X


 A  X

NOTの回路記号

$$X = \overline{A}$$

NOTの論理式



### 4. 排他的論理和 (XOR)

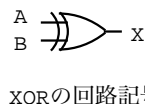
二つのビットを入力し、二つが異なるとき出力が1になるような論理演算です。

| 入力 |   | 出力 |
|----|---|----|
| A  | B | X  |
| 0  | 0 | 0  |
| 0  | 1 | 1  |
| 1  | 0 | 1  |
| 1  | 1 | 0  |

XORの真理値表

$$X = A \oplus B$$

XORの論理式



### 5. NAND

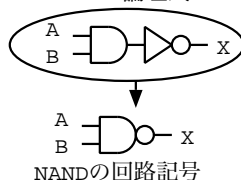
否定 (NOT) と論理積 (AND) を組み合わせた演算です。

| 入力 |   | 出力 |
|----|---|----|
| A  | B | X  |
| 0  | 0 | 1  |
| 0  | 1 | 1  |
| 1  | 0 | 1  |
| 1  | 1 | 0  |

NANDの真理値表

$$X = \overline{A \cdot B}$$

NANDの論理式



### 6. NOR

否定 (NOT) と論理和 (OR) を組み合わせた演算です。

| 入力 |   | 出力 |   |
|----|---|----|---|
| A  | B | C  | S |
| 0  | 0 | 0  | 0 |
| 0  | 1 | 0  | 1 |
| 1  | 0 | 0  | 1 |
| 1  | 1 | 1  | 0 |

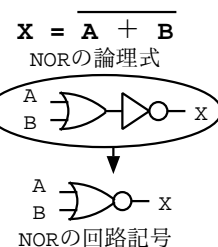
半加算器の真理値表

半加算器の回路図

図 2.15 1 ビット半加算器

| 入力 |   | 出力 |
|----|---|----|
| A  | B | X  |
| 0  | 0 | 1  |
| 0  | 1 | 0  |
| 1  | 0 | 0  |
| 1  | 1 | 0  |

NORの真理値表



### 2.8.3 演算回路

「基本的な論理回路」を組み合わせることにより、足し算／引き算等のより複雑な演算を行う回路を実現できます。

#### 1. 半加算器

例えば、1 ビットの足し算回路は図 2.15 ように実現できます。この例では A、B の 2 ビットを入力し、和 (S) と上の桁への桁上がり (C) を出力する回路を示しています。このような回路のことを「半加算器」と呼びます。

#### 2. 全加算器

半加算器は桁上りを出力しますが、自分自身は下の桁からの桁上りを入力することができません。桁上りの入力を備えた 1 ビット足し算器を「全加算器」と呼び、図 2.16 のように実現することができます。

#### 3. n ビット加算器

図 2.17 のように半加算器と全加算器を組み合わせることで、任意ビットの加算器を実現することができます。

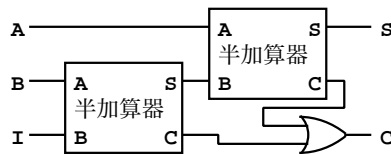
#### 4. n ビット 1 の補数器

1 の補数を作る回路です。1 の補数はビット反転によってできるので、NOT でできます。図 2.18 に

| 入力 |   |   | 出力 |   |
|----|---|---|----|---|
| A  | B | I | C  | S |
| 0  | 0 | 0 | 0  | 0 |
| 0  | 0 | 1 | 0  | 1 |
| 0  | 1 | 0 | 0  | 1 |
| 0  | 1 | 1 | 1  | 0 |
| 1  | 0 | 0 | 0  | 1 |
| 1  | 0 | 1 | 1  | 0 |
| 1  | 1 | 0 | 1  | 0 |
| 1  | 1 | 1 | 1  | 1 |

A:計算の入力  
B:計算の入力  
I:桁上がりの入力

全加算器の真理値表



全加算器の回路図

図 2.16 1 ビット全加算器

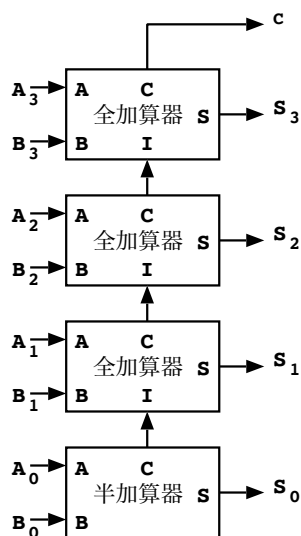


図 2.17 4 ビット加算器

4 ビットの例を示します。簡単ですね。

#### 5. n ビット 2 の補数器

2 の補数を作る回路です。2 の補数は、1 の補数に 1 足すとできるので、1 の補数器の出力に半加算器を組み合わせるとできます。図 2.19 に 4 ビットの例を示します。

演算回路を数種類紹介しました。ここでは紹介しませんでした。引き算回路等も同様に製作可能です。

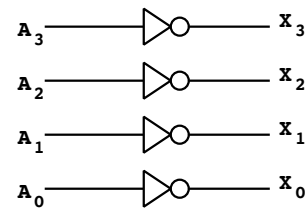


図 2.18 4 ビット 1 の補数器

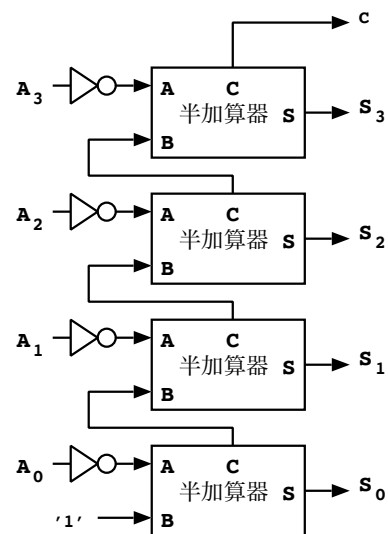


図 2.19 4 ビット 2 の補数器

#### 2.8.4 記憶回路

「基本的な論理回路」を組み合わせることにより、記憶機能を実現することもできます。ここでは、最も簡単な RS-FF(RS フリップフロップ)を紹介します。

図 2.20 に RS フリップフロップの真理値表と回路図を示します。RS-FF は S(Set) と R(Reset) の二つの入力と、状態 ( $Q, \bar{Q}$ ) の出力を持つ回路です。S=0,R=1 を入力することにより、回路はリセットされ出力 Q は 0 になります。S=1,R=0 を入力することにより、回路はセットされ出力 Q は 1 になります。 $\bar{Q}$  には、常に Q の否定が出力されます。

S=0,R=0 を入力すると、回路は直前の値を「記憶」します。S=1,R=1 は入力してはいけない組合せです。RS-FF は、回路をリセット／セットした後で「記憶」状態にすることにより、「値を記憶する回路 (=記憶回路)」として働きます。

### 論理回路素子

回路を製作するには、基本的な論理回路を内蔵した集積回路 (論理 IC) を用いることができます。

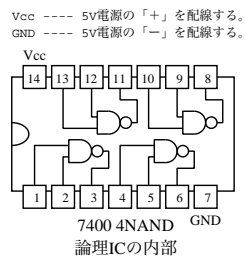
様々な論理 IC が販売されていますが、ここでは、74 シリーズのものを紹介します。下の例に示したのは、NAND ゲートを四つ内蔵した 7400 と呼ばれる IC です。同様な IC で、AND, OR, XOR, NOT 等の回路を内蔵したものがあります。

このような IC 同士を配線して組み合わせることにより、本文で紹介した演算回路や記憶回路を実現することができます。

### 論理 IC

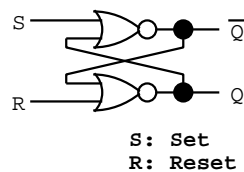


外観



| 入力 |   | 出力 |
|----|---|----|
| S  | R | Q  |
| 0  | 0 | 記憶 |
| 0  | 1 | 0  |
| 1  | 0 | 1  |
| 1  | 1 | 禁止 |

RS-FFの動作



RS-FFの回路

図 2.20 RS フリップフロップ

## 2.9 まとめ

この章では、まず、ノイマン型コンピュータの内部では、情報が電気 (電子) 回路で扱い扱いやすい 2 進数で表現されていること学びました。次に、数値が 2 進数で表現できることや計算方法、文字データの表現法を学びました。最後に、コンピュータの基本回路である論理回路について学びました。

# 第3章

## 組み立て

この章では、TeC7の組み立て方を説明します。何年間も使用するマイコンなので、慎重に組み立てて下さい。

### 3.1 ハンダ付け

マイコン本体は基板に部品をハンダ付けして組み立てます。ハンダ付けの要領を図 3.1 に示します。上手なハンダ付けができると、ハンダが富士山型になります。ハンダ付けの経験がある人にコツを習って、少し練習して下さい。

次のような部品から順にハンダ付けします。

- 1. 背の低い部品
- 2. 基板中心の部品
- 3. 壊れにくい部品

### 3.2 部品の取り付け

組み立ては、部品をハンダ付けして基板に取り付ける作業です。以下の説明では、取り付ける部品が表になっています。表の部品の「記号」と同じ「記号」が、基板の上に白い文字で書いてあるので、その場所に部品をハンダ付けします。

#### 3.2.1 抵抗器

抵抗器は図 3.2 のような外観をした部品です。抵抗値は、部品のカラーコードで見分けます。正しい抵抗値の抵抗器を選んで、図 3.3 のように足を曲げて基板に差

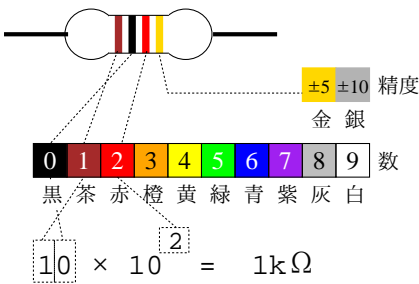


図 3.2 抵抗器

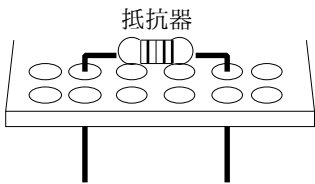


図 3.3 抵抗器の実装

表 3.1 抵抗器

| 記号    | カラーコード | 説明               |
|-------|--------|------------------|
| R1,15 | 黄紫茶金   | 470 Ω 1/4W カーボン  |
| R2-6  | 赤紫茶金   | 270 Ω 1/4W カーボン  |
| R7    | 茶緑黒茶茶  | 1.5k Ω 1/4W 金属皮膜 |
| R8    | 橙黒黒茶茶  | 3.0k Ω 1/4W 金属皮膜 |
| R9-12 | 黄紫赤金   | 4.7k Ω 1/4W カーボン |
| R13   | 橙橙茶金   | 330 Ω 1/4W カーボン  |
| R14   | 茶緑茶金   | 150 Ω 1/4W カーボン  |

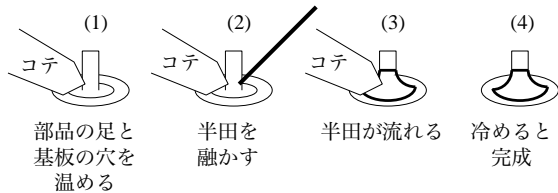


図 3.1 ハンダ付け手順

し込みます。このとき、部品と基板の間に隙間ができないように気を付けて下さい。

抵抗器の一覧を表 3.1 に示します。カラーコードを良く確認して、間違えないように部品を選択して下さい。抵抗器に向きはありませんが、カラーコードの読みやすさを考えて方向を統一する方が良いでしょう。



表 3.2 積層セラミック・コンデンサ

| 記号               | 型番  | 説明    |
|------------------|-----|-------|
| C1,2             | 47  | 47pF  |
| C3,4,12-15,17-34 | 104 | 0.1μF |
| C35              | 101 | 100pF |
| C6,8,10,11       | 475 | 4.7μF |

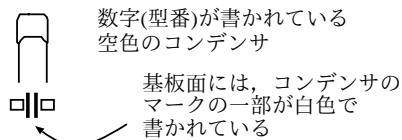


図 3.4 積層セラミック・コンデンサ

表 3.3 IC

| 記号 | 型番    | 説明      |
|----|-------|---------|
| U3 | K516  | 水晶発振 IC |
| U6 | LM339 | 電圧比較 IC |

### 3.2.2 積層セラミック・コンデンサ

積層セラミックコンデンサは4種類、合計31個あります。そのうち、24個は同じ部品です。一覧を表 3.2 に示します。向きはありませんが、完成したとき表面の文字が読みやすい方向に取り付けると良いでしょう。図 3.4 を良く見て、ハンダ付けして下さい。

### 3.2.3 IC

表 3.3 の部品を基板にハンダ付けします。IC には向きがありますので、間違えないように注意して下さい。図 3.5 の説明を良く読んで、方向を間違えないように取り付けて下さい。この時、図 3.6 のように「IC の足を修正して」基板の穴に差し込んで下さい。

部品が傾いたり、浮き上がったりしないようにするには、IC の足2カ所程度を「仮にハンダ付けし、問題が無いか確認して」から他の足をハンダ付けすると良いでしょう。足の多い部品を間違えてハンダ付けしてしまうと、取り外すのが絶望的に大変ですので、くれぐれも間違えないように注意して下さい。

### 3.2.4 集合抵抗器とラダー抵抗器

集合抵抗器とラダー抵抗器は図 3.7 のような部品です。一覧を表 3.4 に示します。この部品にも向きがありますので、注意して下さい。IC と同じ要領で、傾きや浮き上がりが無いよう注意深くハンダ付けしましょう。

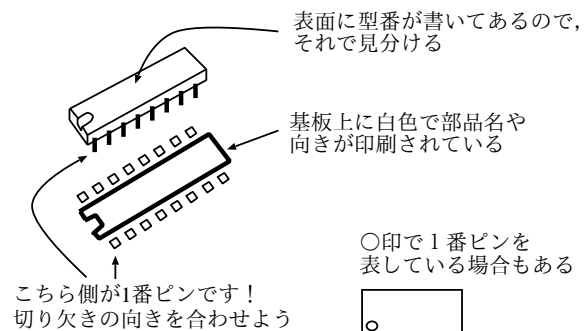


図 3.5 IC 取付の説明図

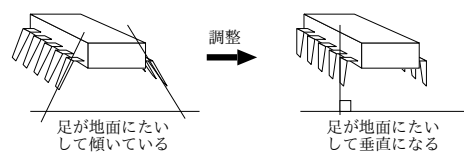


図 3.6 IC 足の調整方法

表 3.4 集合抵抗器

| 記号    | 型番                     | 説明           |
|-------|------------------------|--------------|
| RA1,2 | M9-1-471<br>(L91S 471) | 470 Ω (8 素子) |
| RA3   | M9-1-391<br>(L91S 391) | 390 Ω (8 素子) |
| RA4   | M5-1-391<br>(L51S 391) | 390 Ω (4 素子) |
| RA5   | 8L103                  | ラダー抵抗器       |

表 3.5 フェライトビーズ

| 記号      | 型番 | 説明 |
|---------|----|----|
| FB1,2,3 | なし | なし |

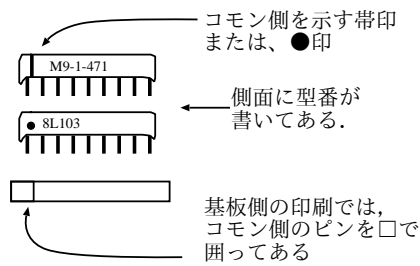
### 3.2.5 フェライトビーズ

フェライトビーズは電源のノイズを弱くする働きのある部品です。図 3.8 の部品をハンダ付けします。この部品は向きはありません。リード線の先を基板に差し込んでハンダ付けします。

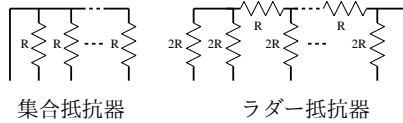
### 3.2.6 ジャンパ

表 3.6 の部品をハンダ付けします(1個だけです)。熱に弱い部品なので手早くハンダ付けしてください。ハンダ付けができたなら、図 3.9 のように上半分の部品を差し込みます。図をよく見て上下を間違えないように注意し





(1)方向の見分け方



これらの部品の内部構造は上図のようになっている。それぞれ、一番左のピンがコモンと呼ばれ、部品のマークがある側である。

(2)内部の構造

図 3.7 集合抵抗器とラダー抵抗器



図 3.8 フェライトビーズ

表 3.6 ジャンパ

| 記号 | 型番 | 説明 |
|----|----|----|
| J1 | なし | なし |

表 3.7 圧電スピーカ

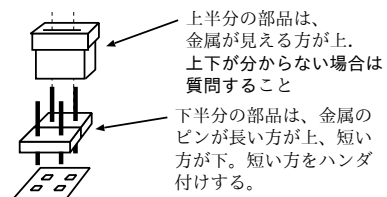
| 記号  | 型番 | 説明     |
|-----|----|--------|
| BZ1 | なし | 圧電スピーカ |

てください。

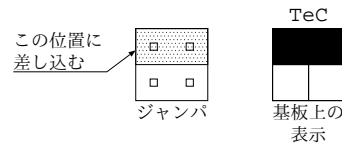
差し込む位置は TeC の位置です。図 3.9 を確認して下さい。

### 3.2.7 圧電スピーカ

圧電スピーカは TeC が音を出すためのスピーカの役割をします。表 3.7、図 3.10 の部品です。向きはありません。基板との隙間ができないように気を付けて下さい。



(1)組み立て方



(2)差し込み位置

図 3.9 ジャンパ

円筒形の部品です。向きはありません。

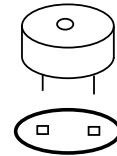


図 3.10 圧電スピーカ

表 3.8 電解コンデンサ

| 記号              | 型番            | 説明         |
|-----------------|---------------|------------|
| C0,C5,C7,C9,C16 | 47 $\mu$ F25V | 47 $\mu$ F |

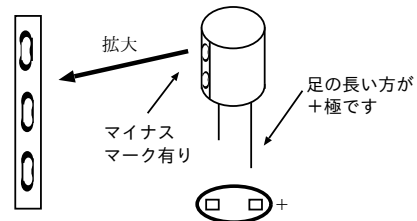


図 3.11 電解コンデンサ

### 3.2.8 電解コンデンサ

表 3.8 の部品をハンダ付けします。部品の形状は図 3.11 のようなものです。この部品も、向きがありますので取り付け方向に注意が必要です。長い足の方が「+」になります。図 3.11 を良く見て、向きを間違えないようにしてください。

### 3.2.9 LED(ランプ)

LED(ランプ) は、マイコンを使用するとき、いつも注視する部品です。傾いていたりすると、とても気にな

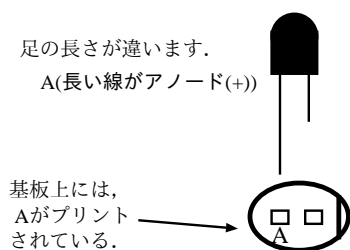


図 3.12 LED の向きの見方

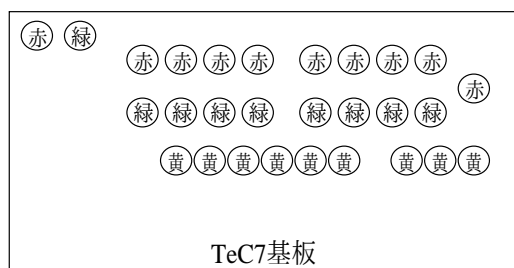


図 3.13 LED の配置

りますので綺麗に取り付けて下さい。アノード(+), カソード(-)の見分け方は図 3.12 の通りです。方向を間違えないように注意しましょう。LEDは、赤、緑、黄の3色があります。3色のLEDは図 3.13 のように配置します。LEDをハンダ付けする要領は次の通りです。

1. 同じ色を一斉に**アノード(+)**だけハンダ付けする。  
(黄→緑→赤の順に中心から取り付ける。)
2. LEDが垂直になっているか確認する。(垂直になっていない場合は、再度温めて修正する。)
3. LEDが奥までささっているか確認する。
4. カソードをハンダ付けする。
5. リード線を切る。

### 3.2.10 スイッチ

スイッチには、**トグルスイッチ**と**プッシュスイッチ**の2種類があります。トグルスイッチは上下に倒すタイプのスイッチ、プッシュスイッチは押しボタンになったスイッチです(図 3.14 参照)。

スイッチは、マイコンを操作するときいつも触る部品です。スイッチが傾いていると操作性が悪くなりますので、まっすぐになるよう慎重にハンダ付けしてください。また、スイッチは温まりにくいので、ハンダ付けするときは少し余計に熱するようにしてください。

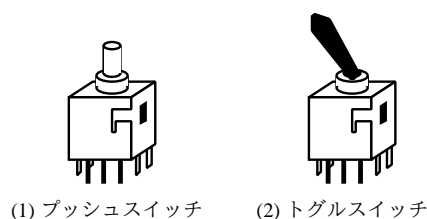


図 3.14 スイッチ

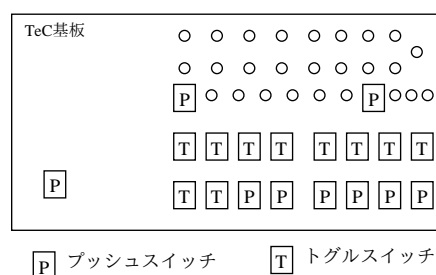


図 3.15 スイッチの配置

表 3.9 JTAG コネクタ

| 記号  | 型番 | 説明             |
|-----|----|----------------|
| CN4 | なし | 小さい 14 ピンのコネクタ |

スイッチの配置は図 3.15 の通りです。間違えないように配置してください。スイッチをハンダ付けする手順は以下の通りです。

1. 足を穴にしっかり差し込み、ハンダ面に7本の足が均等に出ていることを確認する。
2. 足のうち1本をハンダ付けする。
3. スイッチが傾いていないか確認する。  
(傾いていた場合は、温め直して修正する。)
4. 他の足をハンダ付けする。

### 3.2.11 JTAG コネクタ

マイコン内部の回路を書き換えるときに使用するコネクタです。表 3.9 の部品を使用します。この部品にも向きがあります。図 3.16 を良く見て、向きを間違えないように取り付けて下さい。

### 3.2.12 入出力ポートコネクタ

マイコンの外部に追加の回路を接続するとき使用するコネクタです。表 3.10 の部品を使用します。この部品にも向きがあります。図 3.17 を良く見て、向きを間違えないように取り付けて下さい。

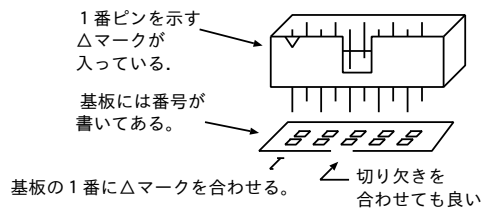


図 3.16 JTAG コネクタ

表 3.10 入出力ポートコネクタ

| 記号  | 型番 | 説明             |
|-----|----|----------------|
| CN5 | なし | 大きい 20 ピンのコネクタ |

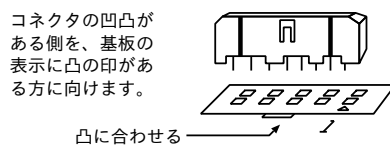


図 3.17 入出力ポートコネクタ

表 3.11 電源コネクタ

| 記号  | 型番 | 説明         |
|-----|----|------------|
| CN1 | なし | USB-B コネクタ |

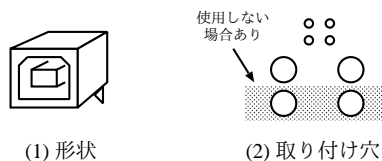


図 3.18 電源コネクタ

## 3.2.13 電源コネクタ

表 3.11 の電源コネクタは、図 3.18 (1) のような形状の部品です。このコネクタからマイコンに電源を供給します。パソコンの USB ポートと接続してパソコンとの通信に使用することもあります。図 3.18 (2) に示す基板上的取り付け穴にハンダ付けします。付属のコネクタ部品によっては手前の二つの穴を使用しないこともあります。このコネクタの取り付けでは、失敗するとひどい火傷をする恐れがあります。以下の手順を良く読んで慎重に作業してください。

1. 穴にしっかり差し込む。
2. 大きな穴とコネクタの端子を十分熱する。

表 3.12 PS/2 コネクタ

| 記号  | 型番 | 説明        |
|-----|----|-----------|
| CN2 | なし | PS/2 コネクタ |



図 3.19 PS/2 コネクタ

表 3.13 VGA コネクタ

| 記号  | 型番 | 説明       |
|-----|----|----------|
| CN3 | なし | VGA コネクタ |

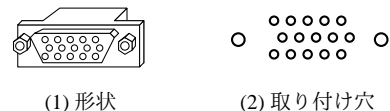


図 3.20 VGA コネクタ

3. 大きな穴が塞がるまで、ハンダをどんどん融かし込む。
4. 十分に冷めるのを待つ。
5. 部品が傾いていないか確認する。
6. 小さな穴に部品の足をハンダ付けする。

## 3.2.14 PS/2 コネクタ

表 3.12 はマイコンにパソコン用のキーボードを接続するコネクタです。外観は図 3.19 (1) の通りです。ハンダ付けの要領は 電源コネクタ と同様です。火傷に気を付けてハンダ付けして下さい。

## 3.2.15 VGA コネクタ

表 3.13 はマイコンにパソコン用のディスプレイを接続するコネクタです。外観は図 3.20 (1) の通りです。取り付け穴の配置を図 3.20 (2) に示します。大きな穴に部品の足を差し込むとロックされる仕組みになっています。大きな穴はハンダ付けしません。

1. ロックされるまで穴にしっかり差し込む。
2. 小さなピンをハンダ付けする。

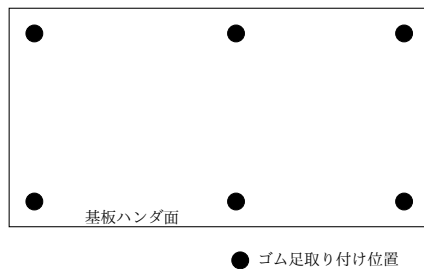


図 3.21 ゴム足の取り付け位置



図 3.22 プッシュスイッチの頭の取り付け方

### 3.2.16 ゴム足の取り付け

基板の裏側に足をつけます。足は、両面テープの付いた黒い円盤状の部品です。足を取り付ける場所は、[図 3.21](#) の通りです。取り付け場所の裏側 (基板の表側) に○印が付いています。

まず、アルコールで周辺をきれいに拭いてから、鉄板とシールを剥がして、両面テープで指定の場所に貼り付けて下さい。

### 3.2.17 プッシュスイッチの頭の取り付け

最後に、プッシュスイッチに頭を取り付けます。[図 3.22](#) を良く見て取り付けて下さい。

## 3.3 完成

### 3.3.1 命令表の貼り付け

ケースのフタの裏に、命令表を貼り付けて下さい。

### 3.3.2 目視確認

ご苦労さまでした。これでマイコンが完成しました。でも、もう一度、部品の種類、向きが間違っていないか目視確認して下さい。

### 3.3.3 設計データの書き込み

間違えがなかったら、教員に頼んでマイコンの設計データを書き込んでもらって下さい。(設計データの書き込みに JTAG コネクタが使用されます。) これで、マイコンとして動作するようになったはずです。

## 第 4 章

# マイコンの操作

この章では、マイコン (TeC) の操作方法を学習します。併せてマイコンの内部にあるレジスタや記憶装置も覚えましょう。

### 4.1 各部の名称

図 4.1 に、マイコン各部の名称 (または役割り) を示します。各部の役割りは次の通りです。

#### 1. コンソールパネル

マイコンにプログラムを入力したり、マイコンの内部を観察したり、プログラムの実行を指示したりするために使用する部分です。データの入力や表示は 2 進数を用います。本マイコンの顔です。

#### 2. リセットスイッチ

マイコンをリセットする押しボタンスイッチです。

#### 3. ジャンパ

マイコンのモードを切替えます。通常は、「TeC」の位置にジャンパーを挿して「TeC モード」で使います。高学年で「TaC モード」を使用します。「DEMO1 モード」、「DEMO2 モード」は、電子オルゴールのデモプログラムが予め入力された状態になるモードです。

#### 4. スピーカー

コンソールパネルを操作したとき、操作を確認するための音を出します。また、マイコンに入力したプログラムで音を出すこともできます。

#### 5. 電源コネクタ

マイコンに電力を供給するコネクタです。付属のアダプタに USB ケーブルを用いて接続します。USB ケーブルでパソコンと接続して通信させることも可能です。

#### 6. キーボードコネクタ

パソコン用の PS/2 キーボードを接続することがで

きます。この機能は、「TaC モード」で使用します。

#### 7. ディスプレイコネクタ

パソコン用の ディスプレイを接続することができます。この機能は、「TaC モード」で使用します。

#### 8. $\mu$ SD スロット

携帯電話などで使用する  $\mu$ SD メモリカードを挿入します。この機能は、「TaC モード」で使用します。

#### 9. JTAG コネクタ

マイコン本体の回路を変更するとき使用します。

#### 10. 入出力ポートコネクタ

マイコンの外部に追加の回路を接続するために使用します。

### 4.2 コンソールパネル

図 4.2 にコンソールパネルの様子を示します。この図を見ながら以下を読んで下さい。

#### 4.2.1 コンソールパネルの構成

コンソールパネルは、以下のランプやスイッチにより構成されています。コンソールパネルが、マイコンボードの約半分の面積を使用していますが、これは、操作性を考慮した結果です。これ以上、小さくするとスイッチの間隔が狭くなり過ぎ、操作性が悪くなります。

#### アドレスランプ

最上段の 8 個の赤ランプはアドレスランプです。アドレスランプは、主記憶を操作するとき、操作対象となる主記憶の番地 (アドレス) を表示します。ランプは点灯している状態が 2 進数の '1'、消灯している状態が 2 進数の '0' を表現しています。

#### データランプ

上から 2 段目の 8 個の緑ランプは、選択されたレジスタまたは主記憶の内容を表示します。

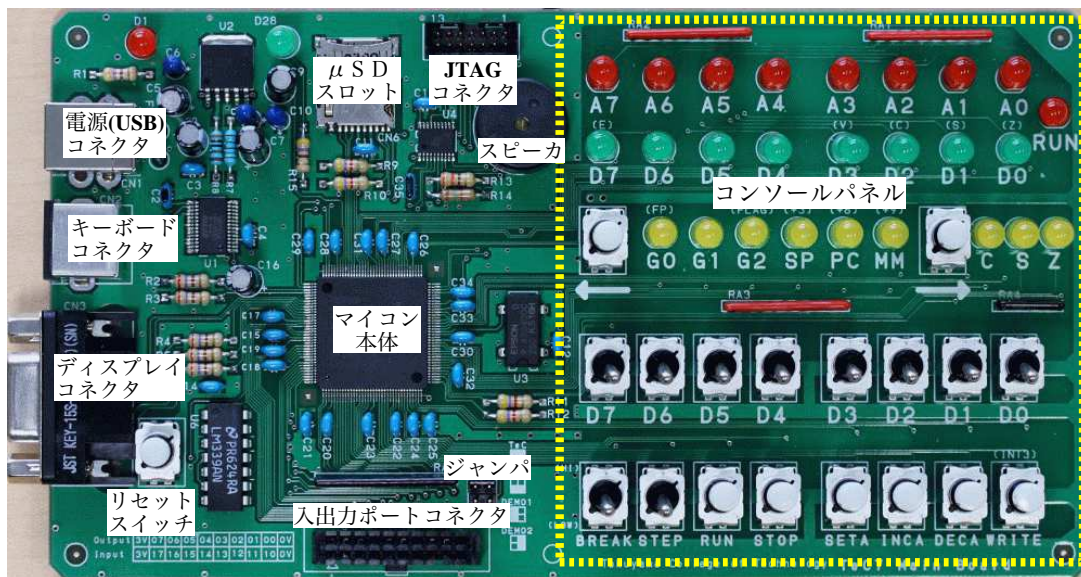


図 4.1 各部の名称

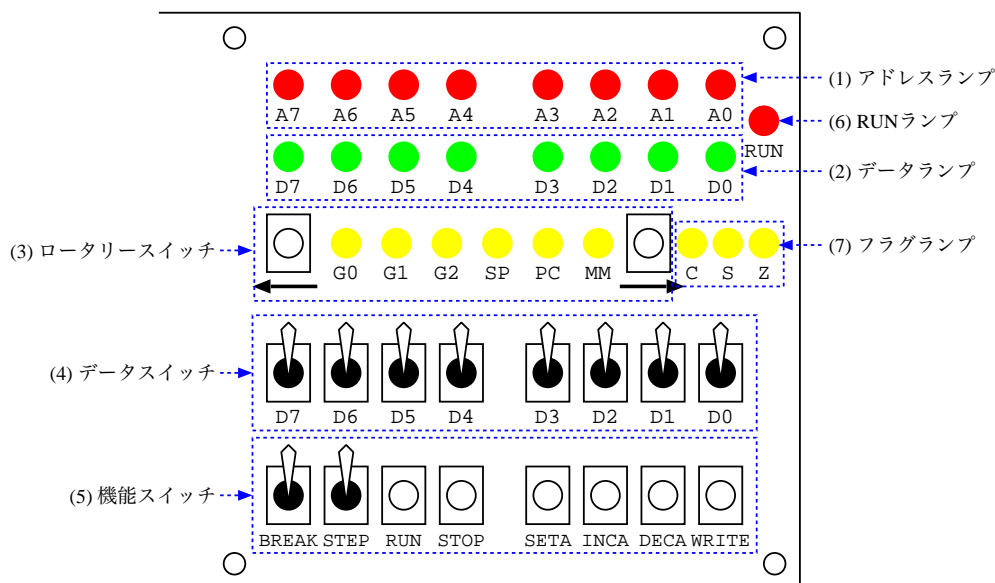


図 4.2 コンソールパネル

### ロータリースイッチ

上から3段目の6個の黄色ランプと左右の押しボタンスイッチは、ロータリースイッチの代用です。通常のロータリースイッチは背が高く、ビデオカセットケースにマイコンを収めるための障害になるのでこのようにしました。

ランプのうち一つが点灯し、レジスタ (G0, G1, G2, SP, PC) または主記憶 (MM:Main Memory) のどれが

選択されているか表示します。左右の押しボタンスイッチ (←, →) によって、点灯するランプを変更することができます。ロータリースイッチで選択されたものの内容が、データランプに表示されます。

### データスイッチ

8個のトグルスイッチはデータやアドレスの値を入力するためのものです。トグルスイッチを上倒して2進数の'1'を、下に倒して2進数の'0'を入力します。



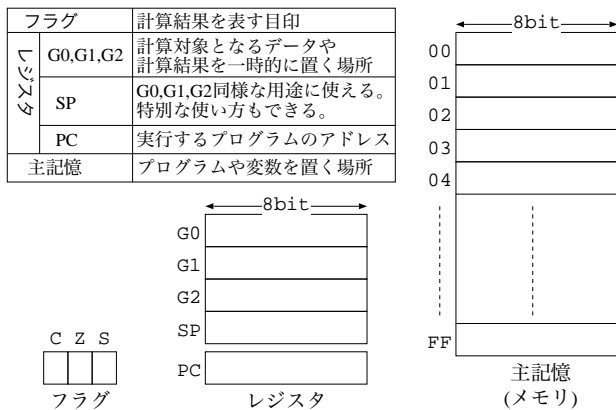


図 4.3 TeC の記憶装置

### 機能スイッチ

一番下の列の 8 個のスイッチには、様々な機能が割り当てられています。左から順に、ブレークポイントモード (BREAK)、ステップ実行モード (STEP)、プログラム実行する (RUN)、プログラム停止する (STOP)、アドレスをセットする (SETA)、アドレスを進める (INCA)、アドレスを戻す (DECA)、書き込む (WRITE) の機能を持ちます。押しボタンスイッチは、長押しでリピート機能が働きます。

### 実行ランプ

右上の RUN ランプは、CPU がプログラムを実行中であることを表します。CPU が正しくない命令を実行した場合は点滅して異常を知らせます。

### フラグランブ

実行ランプ下の三つの黄色ランプ C (Carry), S (Sign), Z (Zero) は、フラグの値を表示します。

#### 4.2.2 コンソールパネルの操作方法

コンソールパネルからプログラムを実行させたり実行を途中で止めたりすることができます。また、図 4.3 に示す TeC 内部の情報にアクセスすることができます。TeC の内部には 3 ビットのフラグ、8 ビットのレジスタが 5 個、8 ビット × 256 のメモリが内蔵されています。以下では、コンソールパネルの操作方法を目的別に説明します。

#### フラグの表示

フラグの状態は C, S, Z の三つのランプに常時表示されています。フラグの値を表示するために何も操作をする必要はありません。フラグの値をコンソールパネルから変更する方法は用意されていません。

### レジスタの表示と書き換え

五つのレジスタ (G0, G1, G2, SP, PC) を表示する方法と、書き換える方法を説明します。レジスタの値は緑色のデータランプに表示されます。8 個のランプで 8 ビットの情報を表現します。どのレジスタをデータランプに表示するかはロータリースイッチで選択します。

レジスタの値を書き換える手順は、次の通りです。

1. ロータリースイッチで目的のレジスタを選択する。
2. 書き込むデータをデータスイッチにセットする。
3. **WRITE スイッチ**を押す。

### 主記憶 (メモリ) のアドレス指定

図 4.3 を見ると分かるように、メモリには  $00_{16} \sim FF_{16}$  (2 進数で表現すると  $0000\ 0000_2 \sim 1111\ 1111_2$ ) の番地 (アドレス) が付けられています。メモリへ対する表示や書き換え等の操作は 1 アドレス (8 ビット) 毎に行います。メモリ・アドレスは、**アドレスランプ**にセットすることで指定します。アドレスランプにアドレスをセットする方法は次の通りです。

1. ロータリースイッチを MM に合わせる。
2. アドレスをデータスイッチにセットする。
3. **SETA (Set Address) スイッチ**を押す。  
(アドレスがアドレスランプに表示される。)
4. アドレスを次の番地に進めたいときは、  
**INCA (Increment Address) スイッチ**を押す。
5. アドレスを前の番地に戻したいときは、  
**DECA (Decrement Address) スイッチ**を押す。

### 主記憶 (メモリ) の表示

アドレスランプにアドレスをセットするだけです。

1. ロータリースイッチを MM に合わせる。
2. 目的のアドレスをアドレスランプで指定する。
3. データランプにメモリの内容が表示される。

### 主記憶 (メモリ) の書き換え

アドレスを指定して書き換えます。

1. 上の手順で目的のメモリ番地の内容を表示する。
2. データスイッチに書き込みたい値をセットする。
3. **WRITE スイッチ**を押す。
4. メモリにデータが書き込まれる。

なお、**WRITE スイッチ**を押したとき、アドレスが自動的に次の番地に変化します。これは、連続したデータの書き込みに便利だからです。

### プログラムの実行

プログラムはメモリにデータと同様に書き込みます。プログラムの開始番地を PC にセットすることで TeC がプログラムの場所を認識します。以下にプログラムを実行する手順を、例 4-1 に実行例を示します。

1. プログラムをメモリに書き込む。
2. プログラムの開始番地を PC に書き込む。
3. **BREAK, STEP スイッチ**を下に倒す。
4. **RUN スイッチ**を押す。(RUN ランプ点灯)
5. プログラムの実行終了。(RUN ランプ消灯)

### プログラムのステップ実行

プログラム中の命令を 1 命令ずつ実行することを**ステップ実行**と言います。作ったプログラムが予定通りに動かないとき原因を調べるために使用します。(このような作業を**デバッグ**と言います。)以下にステップ実行の手順を、例 4-2 にステップ実行の例を示します。

1. プログラムをメモリに書き込む。
2. プログラムの開始番地を PC レジスタに書き込む。
3. **STEP スイッチ**が上に倒れていることを確認する。(倒れていなかった場合は倒す。)
4. **RUN スイッチ**を押す。
5. PC にセットしてあった番地の 1 命令が実行され、プログラムが停止する。
6. レジスタやメモリの状態が予想通りか確認する。
7. 確認が終わったら RUN スイッチを押して、次の命令を実行させる。
8. プログラムの終わりまで以上の操作を繰り返す。

### ブレークポイントを使用した実行

デバッグをするときに目的の命令まで一気に進んでから、ステップ実行をしたい場合があります。そのとき、一気に目的の命令まで進むのにブレークポイントを使用できます。ブレークポイントとは、プログラムの実行を停止する場所(アドレス)のことです。次のような手順になります。

1. プログラムをメモリに書き込む。
2. プログラムの開始番地を PC レジスタに書き込む。

### 例 4-1 最も簡単なプログラムの実行

下は、何もしない命令 (NO 命令) を三つ実行した後、停止命令 (HALT 命令) を実行して停止するプログラムです。このプログラムを打ち込んで実行します。

最も簡単なプログラムのリスト

| 番地               | 命令               | 命令の種類 |
|------------------|------------------|-------|
| 00 <sub>16</sub> | 00 <sub>16</sub> | NO    |
| 01 <sub>16</sub> | 00 <sub>16</sub> | NO    |
| 02 <sub>16</sub> | 00 <sub>16</sub> | NO    |
| 03 <sub>16</sub> | FF <sub>16</sub> | HALT  |

### プログラム打ち込み手順

1. ロータリースイッチを MM に合わせる。
2. アドレスランプに 00<sub>16</sub> をセットする。
3. データスイッチに 00<sub>16</sub> をセットする。
4. WRITE スイッチを 3 回押しメモリ 00<sub>16</sub> 番地～02<sub>16</sub> 番地に 00<sub>16</sub> を書き込む。
5. データスイッチに FF<sub>16</sub> をセットする。
6. WRITE スイッチを押しメモリ 03<sub>16</sub> 番地に FF<sub>16</sub> を書き込む。

### プログラム実行手順

1. ロータリースイッチを PC に合わせる。
2. データスイッチに 00<sub>16</sub> をセットする。
3. WRITE スイッチを押し PC を 00<sub>16</sub> にする。
4. **BREAK, STEP スイッチ**を下に倒す。
5. **RUN スイッチ**を押して実行を開始する。
6. 一瞬でプログラム実行が終了する。
7. PC は、HALT 命令実行後なので、その次の番地 04<sub>16</sub> を指している。
8. 何もしない命令と停止命令しか実行していないので、PC 以外のレジスタ、フラグ、メモリに変化はない。

3. **BREAK スイッチ**が上に **STEP スイッチ**が下に倒れていることを確認する。(倒れていなかった場合は倒す。)
4. プログラムの実行を停止させる命令のアドレスをデータスイッチにセットする。
5. **RUN スイッチ**を押す。
6. データスイッチで指定したアドレスの命令を**実行**後、停止する。



**例 4-2 プログラムのステップ実行**

前の例と同じプログラムをステップ実行モードで実行してみます。

**プログラムの実行手順**

1. 前の例と同様にプログラムを打ち込む。
2. STEP スイッチを上倒して (ステップ実行モードにして), 前の例と同様にプログラムの実行を開始する。
3. ステップ実行モードなので  $00_{16}$  番地の命令だけを実行し停止する。このとき, PC は次命令の番地  $01_{16}$  を指している。
4. 再度 RUN スイッチを押すと,  $01_{16}$  番地の命令を実行し PC が  $02_{16}$  を指して停止する。
5. 以後, RUN スイッチを押す度に, 1 命令実行しては停止する。

**例 4-3 ブレークポイントモードでの実行**

前の例と同じプログラムをブレークポイントモードで実行してみます。

**プログラムの実行手順**

1. 前の例と同様にプログラムを打ち込む。
2. 実行を停止したい命令のアドレス (今回は,  $01_{16}$ ) をデータスイッチにセットする。
3. STEP スイッチを **下**に, BREAK スイッチを上倒して (ブレークポイントモードにして), 前の例と同様にプログラムの実行を開始する。
4. ブレークポイントモードなので  $01_{16}$  番地の命令を実行したら停止する。このとき, PC は次命令の番地  $02_{16}$  を指している。
5. 再度 RUN スイッチを押すと  $02_{16}$  番地からプログラムの実行を再開する。
6.  $03_{16}$  番地には HALT 命令が格納されているので実行が停止する。

## 4.3 リセットスイッチ

基板の左下にある RESET と書かれた押しボタンスイッチのことです。TeC をリセットしたいときに使用します。リセットスイッチを押すと, フラグ, レジスタ, アドレスランプ, TeC に内蔵された入出力装置の状態が ('0' に) クリアされます。メモリの内容と, ロータリー

スイッチの状態はクリアされません。

## 4.4 操作音を小さくする

押しボタンスイッチを押すと, ボタンが押されたことが確認できるように, 「ピッ」と短い電子音が鳴るようになっています。しかし, 静かな部屋で使用する場合, この音が邪魔になります。

このようなときは, **STOP スイッチ**を押しながら **RESET スイッチ**を押して下さい。電子音が, かすかに聞こえる「プッ」という音に変わります。

## 4.5 モード

これまで操作方法を説明してきたマイコンボード (**TeC7** と呼びます。) は, TeC として使用できるだけでなく, 2 進数の入力練習用に楽譜を打込める電子オルゴールとして使用したり, もっと高度な学習をするためにパソコンのように使用したりすることができます。以下のようにジャンパの設定を変更してから電源を入れることでモードを切替えることができます。

**TeC モード** ジャンパを **TeC** の位置 (上側) に挿入してから電源を入れます。TeC7 を TeC として使用できる通常のモードです。

**DEMO1 モード** ジャンパを **DEMO1** の位置 (左側) に挿入してから電源を入れます。このモードは TeC モードの変形バージョンです。TeC の主記憶の  $80H$  番地から  $BFH$  番地の間に予め電子オルゴールプログラムを書き込んだ状態になります。音楽データを  $00H$  番地から打ち込んで **RESET**, **RUN** の順にボタンを押すと演奏が始まります。2 進数データを TeC に入力する練習用に使用します。

**DEMO2 モード** ジャンパを **DEMO2** の位置 (右側) に挿入してから電源を入れます。このモードも TeC モードの変形バージョンです。TeC の主記憶の  $00H$  番地から  $BFH$  番地の間に予め電子オルゴールプログラムと音楽データを書き込んだ状態になります。**RESET**, **RUN** の順にボタンを押すと予め書き込まれた音楽データの演奏が始まります。

**TaC モード** ジャンパを **TaC** の位置 (下側) に挿入してから電源を入れます。このモードでは, TeC7 が **16 ビット版 TeC** として動作します。**16 ビット版 TeC** のことを **TaC** と呼びます。TaC





|    |                                                                                   |                                                                                   |   |                                                                                    |                                                                                     |
|----|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|---|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|    |  |  |   |  |  |
| ド  | 0100 0001<br>1110 1000                                                            | 1000 0011<br>1110 1000                                                            | ソ | 0110 0010<br>1001 1100                                                             | 1100 0100<br>1001 1100                                                              |
| レ  | 0100 1001<br>1101 0001                                                            | 1001 0011<br>1101 0001                                                            | ラ | 0110 1110<br>1000 1011                                                             | 1101 1100<br>1000 1011                                                              |
| ミ  | 0101 0010<br>1011 1010                                                            | 1010 0101<br>1011 1010                                                            | シ | 0111 1011<br>0111 1100                                                             | 1111 0111<br>0111 1100                                                              |
| ファ | 0101 0111<br>1010 1111                                                            | 1010 1111<br>1010 1111                                                            | ド | 0111 1111<br>0111 0101                                                             | 1111 1111<br>0111 0101                                                              |

図 4.4 電子オルゴールデータ

は TacOS (<https://github.com/tctsigemura/TacOS>) を実行できるパソコンのようなコンピュータです。

## 演習

電子オルゴールのデータを TeC に打ち込みなさい。  
手順は次の通りです。

1. ジャンパを DEMO1 の位置に差し込んで電源を投入し TeC を **DEMO1 モード** にする。
2. 電子オルゴールデータは図 4.4 を参照し自分の好きな曲を入力する。例えば、4 分音符のド、レ、ミを連続して入力する場合は、次の表のように主記憶にデータを打込む。

| アドレス      | データ       |    |
|-----------|-----------|----|
| 0000 0000 | 1000 0011 | ド  |
| 0000 0001 | 1110 1000 |    |
| 0000 0010 | 1001 0011 | レ  |
| 0000 0011 | 1101 0001 |    |
| 0000 0100 | 1010 0101 | ミ  |
| 0000 0101 | 1011 1010 |    |
| 0000 0110 | 0000 0000 | 終了 |
| ...       | ...       |    |
| 1111 1111 | 0000 0000 |    |

3. **RESET**, **RUN** の順にボタンを押し演奏を試す。
4. 演習が終わったらジャンパーを TeC の位置に戻す。

## 第5章

# 機械語プログラミング

この章では、TeC のプログラミングを学習します。まず、TeC の内部構成を勉強し、ノイマン型コンピュータの例として、TeC が適切であることを確認します。次に、TeC の命令と、それを使用したプログラムの作成を勉強します。この章の内容をマスターすれば、ノイマン型コンピュータがどのようなもので、何ができて何ができないのか分かってきます。

### 5.1 コンピュータの構成

プログラミングの対象となるコンピュータの構成を確認します。ここでは、一般のコンピュータの構成と、TeC の構成の両方を説明します。TeC の構成は一般のコンピュータの構成を簡単化したもので、原理的には同じものだと分かります。

#### 5.1.1 一般的なコンピュータの構成

一般的にパソコン等の構成は、図 5.1 のようになっています。各部分の役割は次の通りです。

##### 1. CPU (Central Processing Unit: 中央処理装置)

命令を読み込んで、命令に従い計算をするコンピュータの心臓部です。計算機能、制御機能を持っています。CPU が他の部分に積極的に働きかけることにより、コンピュータが働きます。

##### 2. 主記憶装置 (メモリ)

プログラムやデータを記憶する記憶装置です。CPU が高速に読み書きすることができます。

##### 3. 入出力インタフェース

入出力装置をコンピュータに接続するための回路です。入出力装置の種類毎に専用のインターフェース回路が必要になります。

##### 4. 入出力装置

時計やコンピュータの外部とデータのやりとりをする装置のことです。タイマ、キーボード、マウス、

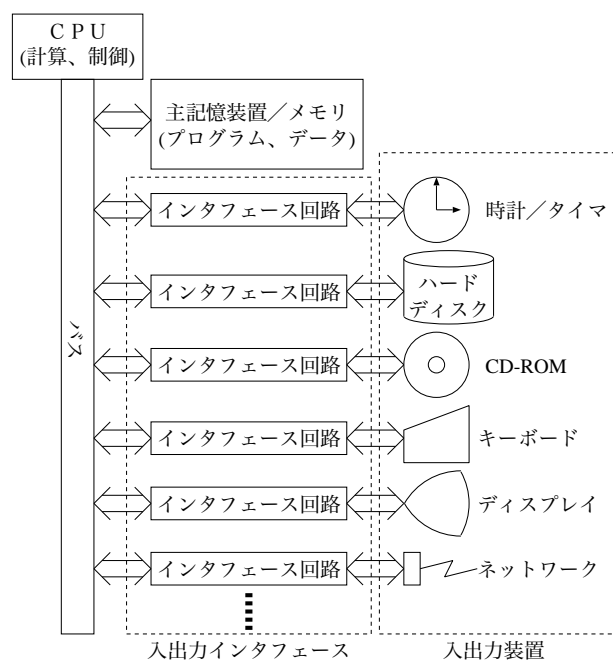


図 5.1 一般的なコンピュータの構成

ディスプレイ、プリンタや通信装置等がこれにあたります。ハードディスクドライブや CD-ROM ドライブ等も入出力装置の仲間ですが、データを記憶する装置であるので少し性格が異なります。そこで、これらは補助記憶装置と呼ばれることがあります。

##### 5. バス

CPU と主記憶装置や入出力インタフェースを接続する配線のことです。アドレスやデータ、制御信号等がバスで伝達されます。CPU はバスを制御することにより、バスに接続された装置とデータのやりとりをします。

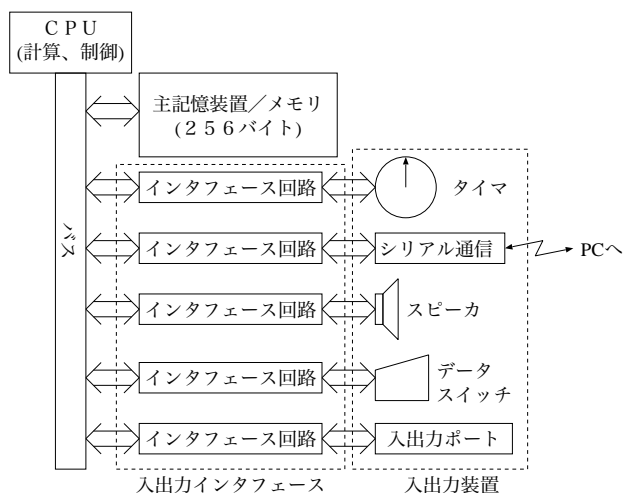


図 5.2 TeC の構成

### 5.1.2 TeC の構成

TeC の構成は図 5.2 のようになっています。接続されている入出力装置が少ない、メモリの容量が小さい等の違いはありますが、基本的には一般的なコンピュータの構成と同じです。

これらの大部分は、「図 4.1 各部の名称」で「マイコン本体」と表示された正方形の LSI の内部に集積されています。各部分の役割りは次の通りです。

#### 1. CPU

命令を読み込んで命令に従い計算をする TeC の心臓部です。計算機能、制御機能を持っています。TeC の CPU は約 30 種類の命令を理解できます。

#### 2. 主記憶装置 (メモリ)

TeC のメモリは、8 ビット構成、256 アドレスのものです。一つのアドレスに 8 ビットの情報を記憶することができます。0~255 (00<sub>16</sub>~FF<sub>16</sub>) の範囲でアドレスを指定するためには、アドレス情報も 8 ビット必要ですので、8 ビットアドレスと呼ぶこともあります。

#### 3. 入出力インタフェース

インターフェース回路を通してバスと入出力装置が接続されます。

#### 4. 入出力装置

入出力装置として、シリアル通信回路、スピーカ、コンソールパネルのデータスイッチ、タイマ、入出力ポートが搭載されています。

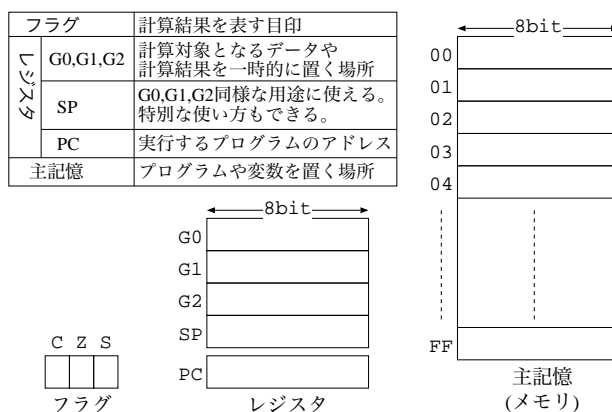


図 5.3 TeC の記憶装置

### 5. バス

CPU と主記憶装置や入出力インタフェースの間で次の情報を伝達します。

- 8 ビットのアドレス
- 8 ビットのデータ
- いくつかの制御情報

### 5.1.3 TeC 内部の記憶装置

コンピュータはプログラムを実行するために内部に 2 進数で情報を記憶します。TeC が内部に持っている記憶装置を図 5.3 にまとめます。プログラムは主記憶 (メモリ) に記憶されます。CPU は主記憶 (メモリ) からプログラム中の命令を読み込んで、フラグやレジスタを使用しながら計算をします。

#### 1. フラグ

図 5.2 の「CPU」内部にある 3 ビットの記憶装置です。直前に行った計算結果の特徴 (ゼロだった等) を記録し条件判断に利用します。

#### 2. レジスタ

図 5.2 の「CPU」内部にある 8 ビットの記憶装置です。各レジスタの役割は図中の表に示した通りです。

#### 3. 主記憶 (メモリ)

図 5.2 の「主記憶装置」のことです。256 アドレス (256 バイト) の容量があります。ここにプログラムやデータを記憶します。

## 5.2 機械語プログラミング

機械語命令の羅列が機械語プログラムです。機械語プログラムを作る作業が機械語プログラミングです。

### 5.2.1 機械語命令

コンピュータの CPU は、メモリから命令を取り出し、取り出した命令を解釈し、決められた計算等を行います。CPU が解釈できる状態の命令を、「CPU = 機械」が解釈できる命令なので「**機械語命令**」と呼びます。機械語命令は 2 進数でメモリに書き込みます。

2 進数を用いた機械語の表現方法だけでは人間にとって分かり難いので、命令を意味する英語を簡略化した短い綴で表します。これを「**ニーモニック**」と呼びます。

| 機械語命令                  | ニーモニック | 意味           |
|------------------------|--------|--------------|
| 0000 0000 <sub>2</sub> | NO     | No Operation |
| 1111 1111 <sub>2</sub> | HALT   | Halt         |

### 5.2.2 ノイマン型コンピュータの特徴

TeC もノイマン型コンピュータの一種です。ノイマン型コンピュータの特徴として次の 3 点があげられますが、これは、TeC にも当てはまります。

1. プログラム内蔵（ストアードプログラム）方式  
データだけでなく、プログラムもメモリに記憶する方式です。TeC でもプログラムはメモリに書き込んで実行します。
2. 逐次実行  
プログラムの命令をメモリのアドレス順に一つ一つ順番に実行することを言います。
3. 2 進法  
プログラムやデータの表現に 2 進数を使います。

### 5.2.3 機械語プログラミング

ノイマン型コンピュータ上での機械語プログラミングとは、逐次実行により実行される順に機械語命令の羅列を作る作業のことです。作ったプログラムは、2 進数にしてメモリに書き込んで実行します。

TeC にどのような機械語命令があるかを、次の節から種類別に説明します。

#### 命令表

TeC でどんな機械語命令が使用できるかは、本体のケースに張り付けた命令表（または、「[図 B.1](#) TeC7 命令表」）により確認することができます。これまでにでてきた NO, HALT 命令について、命令表を確認して下さい。ニーモニックと命令の名前、命令のフォーマットが分かりますね。

その他にも、まだ習っていない情報がたくさん掲載されています。一通り勉強が終わったら、この命令表だけで TeC を自由にプログラミングできるようになります。

## 5.3 特殊な命令

TeC の機械語命令の中で特別なものから説明します。

### 5.3.1 NO (No Operation) 命令

**意味：** 何もしない命令です。

**ニーモニック：** NO

**命令フォーマット：** NO 命令は 1 バイト長の命令です。命令フォーマットの見方も説明します。

| 第 1 バイト           |                                 |
|-------------------|---------------------------------|
| OP                | GR XR                           |
| 0000 <sub>2</sub> | 00 <sub>2</sub> 00 <sub>2</sub> |

命令フォーマットは、その機械語命令が 2 進数でどのように表現されるかを表します。NO 命令の場合、機械語が 1 バイトであること、その 1 バイトの内容が、OP フィールド 4 ビットが 0000<sub>2</sub>、GR フィールド 2 ビットが 00<sub>2</sub>、XR フィールド 2 ビットが 00<sub>2</sub> であることが命令フォーマットの図から分かります。NO 命令以外の命令も、フィールドのビット数は同じです。

### 5.3.2 HALT (Halt) 命令

**意味：** プログラム実行を停止します。

**ニーモニック：** HALT

**命令フォーマット：** 1 バイト長の命令です。

| 第 1 バイト           |                                 |
|-------------------|---------------------------------|
| OP                | GR XR                           |
| 1111 <sub>2</sub> | 11 <sub>2</sub> 11 <sub>2</sub> |

## 5.4 データ転送命令

TeC の CPU とメモリの間でデータ転送をする機械語命令を説明します。

### 5.4.1 LD (Load) 命令

Load は、「荷物等を積み込む」と言う意味の英語です。LD は、Load の綴を縮めたものです。

**意味：** 主記憶（メモリ）からレジスタへデータを転送します。（メモリからレジスタへ値をコピーします。メモリの値は変化しません。）

**フラグ：** 変化しません。

**ニーモニック：** LD GR,A[,XR]

GR は、転送先レジスタを表します。A はデータの置いてあるメモリのアドレス（番地）を表します。[,XR] は、省略可能です。将来、きちんと説明するまで、[,XR] は常に省略（この部分を書かない）してください。

**命令フォーマット：** LD 命令は 2 バイトの長さを持ちます。命令の各ビットは次の通りです。

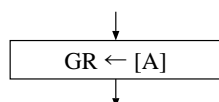
| 第1バイト             |       | 第2バイト     |
|-------------------|-------|-----------|
| OP                | GR XR |           |
| 0001 <sub>2</sub> | GR XR | aaaa aaaa |

- OP フィールド（4 ビット）は LD 命令を示す 0001<sub>2</sub> にします。
- GR フィールド（2 ビット）はデータを格納するレジスタを表します。GR フィールドの値は次の表の意味を持ちます。

| GR              | 意味 |
|-----------------|----|
| 00 <sub>2</sub> | G0 |
| 01 <sub>2</sub> | G1 |
| 10 <sub>2</sub> | G2 |
| 11 <sub>2</sub> | SP |

- XR フィールド（2 ビット）は、将来、詳しく説明するまでは常に 00<sub>2</sub> にします。
- 第2バイト（aaaa aaaa）は、A（番地）を 2 進数で格納します。

**フローチャート：** フローチャートでは、LD 命令を次のように描きます。[ と ] を書き忘れないように、注意して下さい。



### RUN ランプが点滅したら

CPU が機械語命令として解釈できない命令を実行しようとしたことを表します。PC が解釈できなかった命令の次の番地を指した状態で CPU が停止します。PC の値からおかしな命令を見付けて訂正してください。

### RUN ランプに注意

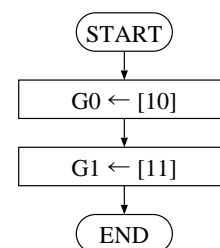
RUN ランプはプログラム実行中に点灯しています。プログラムが暴走し終了しない時は、RUN ランプが点灯したままになります。プログラム実行中でも、コンソールパネルを操作してメモリやレジスタの値を表示できるので、プログラムが暴走していることに気づかないことがあります。RUN ランプを常に気にするようにして下さい。

**使用例：** LD 命令を使用したプログラムの例を示します。なお、ニーモニック欄の数値は 10 進数ですので、注意してください。

| 番地               | 機械語                               | ラベル | ニーモニック   |
|------------------|-----------------------------------|-----|----------|
| 00 <sub>16</sub> | 10 <sub>16</sub> 0A <sub>16</sub> |     | LD G0,10 |
| 02 <sub>16</sub> | 14 <sub>16</sub> 0B <sub>16</sub> |     | LD G1,11 |
| 04 <sub>16</sub> | FF <sub>16</sub>                  |     | HALT     |

例の中で機械語は 16 進数で書いてありますが、2 進数に変換すると 0 番地の命令は 0001 0000<sub>2</sub>、0000 1010<sub>2</sub> です。上の命令フォーマットと対応を確認してください。

このプログラムをフローチャートで表現すると、次のようになります。





## 5.4.2 ST (Store) 命令

Store は、「倉庫に保管する」という意味の英語です。ST は、Store の綴を縮めたものです。

**意味：** レジスタからメモリへデータを転送します。  
(レジスタの値は変化しません。)

**フラグ：** 変化しません。

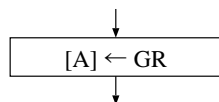
**ニーモニック：** ST GR,A[,XR]

GR は、転送元レジスタを表します。A はデータを保管するメモリのアドレス (番地) を表します。[,XR] は、省略可能です。将来、きちんと説明するまで、[,XR] は常に省略してください。

**命令フォーマット：** ST 命令は 2 バイトの長さを持ちます。命令のフォーマットは次の通りです。各フィールドの意味は、LD 命令と同様です。

| 第1バイト             |       | 第2バイト     |
|-------------------|-------|-----------|
| OP                | GR XR |           |
| 0010 <sub>2</sub> | GR XR | aaaa aaaa |

**フローチャート：** フローチャートでは、ST 命令を次のように描きます。[と] を書き忘れないように、注意して下さい。



**使用例：** 10 番地のデータを 11 番地にコピーして停止するプログラムの例を示します。番地と機械語の欄はいつも 16 進数で書くので、この例から小さく 16 と書くのを止めます。なお、ニーモニック欄の数値は 10 進数です。

| 番地 | 機械語   | ラベル | ニーモニック   |
|----|-------|-----|----------|
| 00 | 10 0A |     | LD GO,10 |
| 02 | 20 0B |     | ST GO,11 |
| 04 | FF    |     | HALT     |

## 問題

- 11<sub>16</sub> 番地のデータを 12<sub>16</sub> 番地に、10<sub>16</sub> 番地のデータを 11<sub>16</sub> 番地にコピーして停止するプログラムを書きなさい。
- 10<sub>16</sub> 番地のデータと 11<sub>16</sub> 番地のデータを交換して停止するプログラムを書きなさい。
- これらのプログラムを、実際に TeC で実行して正しく動くことを確認しなさい。

## プログラムの作成手順

ST 命令の使用例のように、表を作成しながらプログラムを書きます。手順は次の通りです。

1. 表の枠を書く。
2. ニーモニックでプログラムを書く。プログラムに間違いが無いか、この段階で良く考える。
3. 機械語命令の長さを考えながら番地欄を記入する。(これは、機械的な作業。ミスをしないよう慎重に。)
4. 機械語欄を記入する。(これも、機械的な作業。ミスをしないように。)
5. 全部記入したら、TeC に打ち込んで実行してみる。
6. うまく動かなかったら、全てのステップを再度確認し間違いを探す。
7. うまく動いたら完成!!! (うまく動いたかどうかの確認も慎重に!)

表に記入したら終わりではありません。TeC に打ち込んで実際に動くことを確認してください。動くはずなのに動かないことが多い (まず、動かないと考えた方が良いでしょう) です。

実際に動かし正しく動作することを確認できるまでは、プログラムは完成していません。

## うまく動かない場合

プログラムが正しく動かない人は、1 命令ずつ実行しながら、メモリやレジスタの値が予想通りに変化するか調べて下さい。1 命令ずつ実行する方法は、「プログラムのステップ実行」(p.24) に説明してあります。

面倒くさいと思わないで、地道に問題点を捜しましょう。問題点を見つけるコツが分かってきたら、演習がすごく楽になります。慣れるまで少し辛抱して下さい。

## 5.5 算術演算命令

名前の通り算術計算（普通の計算）をするための命令です。TeC は足算と引算だけができます。

### 5.5.1 ADD (Add) 命令

ADD 命令は、名前の通り、加算（足算）をします。

**意味：** レジスタの値とメモリデータの和を計算し、結果を元のレジスタに格納します。（メモリの値は変化しません。）

**フラグ：** 計算の結果により変化します。

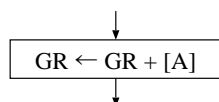
**ニーモニック：** ADD GR,A[,XR]

GR は、足し算の対象になるレジスタを表します。このレジスタの値とメモリの値が足し合わされ、結果がこのレジスタに格納されます。A と [,XR] の意味は LD 命令と同様です。

**命令フォーマット：** ADD 命令は 2 バイトの長さを持ちます。命令の各フィールドの意味は、LD 命令と同様です。

| 第1バイト             |       | 第2バイト     |
|-------------------|-------|-----------|
| OP                | GR XR |           |
| 0011 <sub>2</sub> | GR XR | aaaa aaaa |

**フローチャート：** フローチャートでは、ADD 命令を次のように描きます。[ と ] を書き忘れないように、注意して下さい。



**使用例：** 7 番地のデータと 8 番地のデータの和を計算し、9 番地に格納するプログラムの例を示します。

| 番地 | 機械語   | ラベル | ニーモニック   |
|----|-------|-----|----------|
| 00 | 10 07 |     | LD G0,7  |
| 02 | 30 08 |     | ADD G0,8 |
| 04 | 20 09 |     | ST G0,9  |
| 06 | FF    |     | HALT     |

このプログラムを実行するときは、予め 7 番地、8 番地に足し合わせるデータを格納しておく必要があります。例えば、7 番地に 1、8 番地に 2 を格納した状態でこのプログラムを実行すると、計算結果の 3 が 9 番地に格納されます。

### 5.5.2 SUB (Subtract) 命令

Subtract は、「引き算をする」という意味の英語です。SUB は Subtract の綴を縮めたものです。

**意味：** レジスタの値とメモリデータの差を計算し、結果を元のレジスタに格納します。（メモリの値は変化しません。）

**フラグ：** 計算結果により変化します。

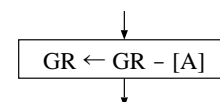
**ニーモニック：** SUB GR,A[,XR]

GR は、引き算の対象になるレジスタを表します。このレジスタの値からメモリの値が引かれ、結果がこのレジスタに格納されます。A と [,XR] の意味は LD 命令と同様です。

**命令フォーマット：** SUB 命令は 2 バイトの長さを持ちます。命令の各フィールドの意味は、LD 命令と同様です。

| 第1バイト             |       | 第2バイト     |
|-------------------|-------|-----------|
| OP                | GR XR |           |
| 0100 <sub>2</sub> | GR XR | aaaa aaaa |

**フローチャート：** SUB 命令は次のように描きます。[ と ] を書き忘れないように、注意して下さい。



**使用例：** 7 番地のデータから 8 番地のデータを引いた結果を 9 番地に格納するプログラムの例を示します。

| 番地 | 機械語   | ラベル | ニーモニック   |
|----|-------|-----|----------|
| 00 | 10 07 |     | LD G0,7  |
| 02 | 40 08 |     | SUB G0,8 |
| 04 | 20 09 |     | ST G0,9  |
| 06 | FF    |     | HALT     |

このプログラムを実行するときは、予め 7 番地に引かれる数、8 番地に引く数を格納しておく必要があります。

## 問題

10<sub>16</sub> 番地のデータと 11<sub>16</sub> 番地のデータの和を 12<sub>16</sub> 番地に、差を 13<sub>16</sub> 番地に格納するプログラムを作成し、TeC で実行して正しく実行できたことを確かめなさい。



## 5.6 ジャンプ命令

ノイマン型コンピュータの特徴は逐次実行です。命令を番地の順に一つ一つ実行します。プログラムの実行が進んで行く流れを「プログラムの流れ」と言います。プログラムの流れは PC (Program Counter) によって管理され、通常は PC が順次増加します。

しかし、プログラムの同じ部分を繰り返したり、条件によりプログラムの動きを変更する目的で、流れを別の場所に飛ばすこと (Jump) も必要です。

ジャンプ命令はこのような目的で PC の値を変更し、プログラムの流れを別の番地へ飛ばすものです。プログラムの流れは飛んで行った先にあるプログラムを順番に実行する流れになり、もとに戻って来ることはありません。

### 5.6.1 JMP (Jump) 命令

この命令を実行するとプログラムの流れが、必ず指定の番地にジャンプします。

**意味：** プログラムの流れをジャンプさせます。

**フラグ：** 変化しません。

**ニーモニック：** JMP A[,XR]

A はジャンプ先のアドレス (番地) を表します。  
[,XR] は、省略可能です。将来、きちんと説明するまで、[,XR] は常に省略 (この部分を書かない) してください。

**命令フォーマット：** JMP 命令は 2 バイトの長さを持ちます。GR フィールドは必ず 00<sub>2</sub> にします。第 2 バイトがジャンプ先のアドレスを示します。

| 第1バイト             |                    | 第2バイト     |
|-------------------|--------------------|-----------|
| OP                | GR XR              |           |
| 1010 <sub>2</sub> | 00 <sub>2</sub> XR |           |
|                   |                    | aaaa aaaa |

**フローチャート：** JMP 命令はフローチャートの線に対応します。次の命令で詳しく説明します。

**使用例：** 00<sub>16</sub> 番地の命令をいつまでも繰り返す (止まらない) プログラムの例を示します。

| 番地 | 機械語   | ラベル | ニーモニック   |
|----|-------|-----|----------|
| 00 | 30 04 |     | ADD G0,4 |
| 02 | A0 00 |     | JMP 0    |

### ラベル

JMP 命令の使用例では、ジャンプする先のアドレスをニーモニックの中に直接書きました。もしも、プログラムに変更があり、ジャンプする先のアドレスが変化したらどうでしょうか。プログラムに直接アドレスが書いてあると、それを書き直す必要が生じます。

番地が変化してもニーモニックで書いたプログラムを変更しなくて良いように、アドレスの代わりにその場所に付けた記号を使うと便利です。

場所に付けた記号のことをラベルと言います。ラベルを用いてプログラムを書き換えると次のようになります。

| 番地 | 機械語   | ラベル  | ニーモニック   |
|----|-------|------|----------|
| 00 | 30 04 | LOOP | ADD G0,4 |
| 02 | A0 00 |      | JMP LOOP |

### データの定義 (DC 命令)

ラベルを用いることにより、ニーモニックで書いたプログラムの取扱いが非常に便利になりました。しかし、データの部分はニーモニックに記述できていません。これではニーモニックだけでプログラムの全体像が理解できません。

そこで、データを記述するための “DC (Define Constant)” 命令を追加します。DC 命令は、機械語の代わりにオペランドで指定した値のデータを生成します。また、DC 命令の行にラベルを付けることによりデータもラベルで参照できるようになります。次は、JMP 命令の使用例を DC 命令を用いて書き直したものです。4 番地に値 “1” のデータがあることが、うまく記述できました。

| 番地 | 機械語   | ラベル  | ニーモニック     |
|----|-------|------|------------|
| 00 | 30 04 | LOOP | ADD G0,ONE |
| 02 | A0 00 |      | JMP LOOP   |
| 04 | 01    | ONE  | DC 1       |

**領域の定義 (DS 命令)**

DS 命令は DC 命令に良く似た命令です。DC 命令は機械語命令の代わりにデータを生成しました。DS 命令は結果を格納するための領域を生成します。指定された数値は領域の大きさになります。

**フラグ**

いくつかのプログラムを作成して動かして見ました。そのとき、TeC の C, S, Z ランプが点灯したり消灯したりしたのが分かったでしょうか？

これらのランプは、対応したフラグ (Flag: 旗) の値を表示しています。どんなときランプが点灯し (フラグが 1 になり)、どんなときランプが消灯した (フラグが 0 になった) のか説明します。

**C (Carry) フラグ** Carry は「桁を繰り上げる」という意味です。C フラグは、計算中に 8bit の最上位桁からの「桁上がり」や、8bit の最上位桁での「桁借り」が発生したことを表します。つまり、計算結果が 255 を超えてしまったことや、0 より小さくなったことを表します。C フラグは、計算値が符号無しと考えたときのオーバーフローを表しています。

**S (Sign) フラグ** Sign は、「符号」を意味します。計算の結果を符号付き 2 進数として解釈した場合、負の値になることを表します。つまり、計算結果の 8bit の最上位ビットが 1 のとき 1 になります。

**Z (Zero) フラグ** Zero は、名前の通り計算の結果がゼロになったことを表します。つまり、計算結果の 8bit の全てのビットが 0 のとき 1 になります。

これら三つのフラグは、命令表で「フラグ変化」の欄に「○」印が付いている演算命令を実行する度に計算結果によって変化します。つまり、直前の演算の結果を反映しています。これまでに出てきた命令では、ADD, SUB がフラグを変化させる命令です。

**5.6.2 JZ (Jump on Zero) 命令**

Z フラグが 1 のとき (計算結果が 0 だったとき) だけジャンプします。

**意味:** Z フラグが 1 ならジャンプします。

**フラグ:** 変化しません。

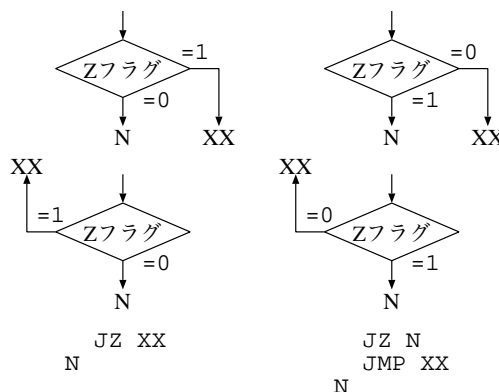
**ニーモニック:** JZ A[,XR]

JMP 命令と同様です。

**命令フォーマット:** JZ 命令は 2 バイトの長さを持ちます。GR フィールドは必ず 01<sub>2</sub> にします。第 2 バイトがジャンプ先のアドレスを示します。

| 第1バイト             |                    | 第2バイト     |
|-------------------|--------------------|-----------|
| OP                | GR XR              |           |
| 1010 <sub>2</sub> | 01 <sub>2</sub> XR | aaaa aaaa |

**フローチャート:** JZ 命令のフローチャートは、次のように様々な描き方が考えられます。例には、JMP 命令と組合せて菱形一つに対応させたものもありますが、こだわる必要はありません。場合によって、柔軟にアレンジして下さい。



**使用例:** 結果がゼロだったら停止するプログラムの例を示します。定数の“1”を使用するために DC 命令でメモリの 07<sub>16</sub> 番地にデータ 01<sub>16</sub> を置きました。定数を使用するためにはメモリ上に定数データを置く必要があります。

| 番地 | 機械語   | ラベル  | ニーモニック      |
|----|-------|------|-------------|
| 00 | 30 07 | LOOP | ADD GO, ONE |
| 02 | A4 06 |      | JZ STOP     |
| 04 | A0 00 |      | JMP LOOP    |
| 06 | FF    | STOP | HALT        |
| 07 | 01    | ONE  | DC 1        |

**C フラグの詳しい説明**

C フラグは、計算中に、8bit の最上位桁からの「桁上がり (Carry)」が発生したことや、8bit の最上位桁での「桁借り (Borrow)」が発生したことを表します。例えば、次の計算では最上位桁からの「桁上がり」は発生しませんので、C フラグは“0”になります。

$$\begin{array}{r}
 0000\ 0001_2 \quad (1) \\
 + \quad 0000\ 0011_2 \quad (3) \\
 \hline
 \boxed{0} \quad 0000\ 0100_2 \quad (4) \\
 \text{C}
 \end{array}$$

次の計算では最上位桁からの「桁上がり」が発生し、C フラグが“1”になります。C フラグは、計算値が符号無しと考えたときのオーバーフローを表しています。

$$\begin{array}{r}
 \text{符号無し} \quad \text{符号付} \\
 1111\ 1111_2 \quad (255) \quad (-1) \\
 + \quad 0000\ 0001_2 \quad (1) \quad (+1) \\
 \hline
 \boxed{1} \quad 0000\ 0000_2 \quad (\text{オーバーフロー}) \quad (0) \\
 \text{C}
 \end{array}$$

また、引き算でも C フラグが 1 になることがあります。例えば、次のように小さな数から大きな数を引いた場合です。機械的に引き算をすると、最上位桁で桁借りが発生します。この時も、C フラグが 1 になります。

$$\begin{array}{r}
 \text{符号無し} \quad \text{符号付} \\
 0110\ 0100_2 \quad (100) \quad (+100) \\
 - \quad 0110\ 1110_2 \quad (110) \quad (+110) \\
 \hline
 \boxed{1} \quad 1111\ 0110_2 \quad (\text{オーバーフロー}) \quad (-10) \\
 \text{C}
 \end{array}$$

このように引き算の場合は、C フラグは、計算が符号無しと考えたときの、負へのオーバーフローを表しています。

**5.6.3 JC (Jump on Carry) 命令**

C フラグが 1 のときだけジャンプします。

**意味：** C フラグが 1 ならジャンプします。

**フラグ：** 変化しません。

**ニーモニック：** JC A[,XR]

JMP 命令と同様です。

**命令フォーマット：** JC 命令は 2 バイトの長さを持ちます。GR フィールドは必ず 10<sub>2</sub> にします。第 2 バイトがジャンプ先のアドレスを示します。

| 第 1 バイト           |                    | 第 2 バイト   |
|-------------------|--------------------|-----------|
| OP                | GR XR              |           |
| 1010 <sub>2</sub> | 10 <sub>2</sub> XR | aaaa aaaa |

**フローチャート：** JC 命令のフローチャートは、JZ 命令と同様な考えで描きます。JZ 命令を参考にして下さい。

**5.6.4 JM (Jump on Minus) 命令**

S フラグが 1 のとき (計算結果が負だったとき) だけジャンプします。名前が、JS 命令ではなく JM 命令になっているので注意して下さい。

**意味：** S フラグが 1 ならジャンプします。

**フラグ：** 変化しません。

**ニーモニック：** JM A[,XR]

JMP 命令と同様です。

**命令フォーマット：** JM 命令は 2 バイトの長さを持ちます。GR フィールドは必ず 11<sub>2</sub> にします。第 2 バイトがジャンプ先のアドレスを示します。

| 第 1 バイト           |                    | 第 2 バイト   |
|-------------------|--------------------|-----------|
| OP                | GR XR              |           |
| 1010 <sub>2</sub> | 11 <sub>2</sub> XR | aaaa aaaa |

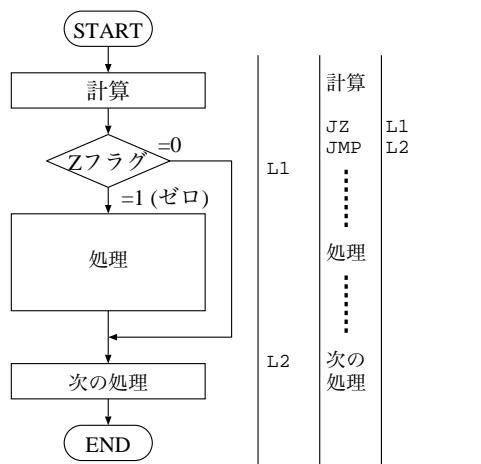
**フローチャート：** JM 命令のフローチャートも、JZ 命令と同様な考えで描きます。JZ 命令を参考にして下さい。

**5.6.5 ジャンプ命令の利用**

以下にジャンプ命令を使用して、「条件判断を行う場合」と「繰り返し処理を行う場合」を説明します。更にその後の例題を理解してから「問題」に挑戦して下さい。

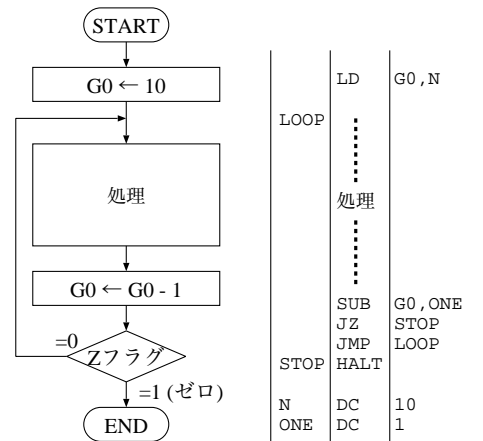
## 条件判断 1

ジャンプ命令を用いて、条件判断の機能があるプログラムを作ることができます。次のフローチャートとプログラムは、計算結果がゼロだった場合だけ処理をするものです。このように、ある条件の場合だけ処理をするプログラムを作ることができます。



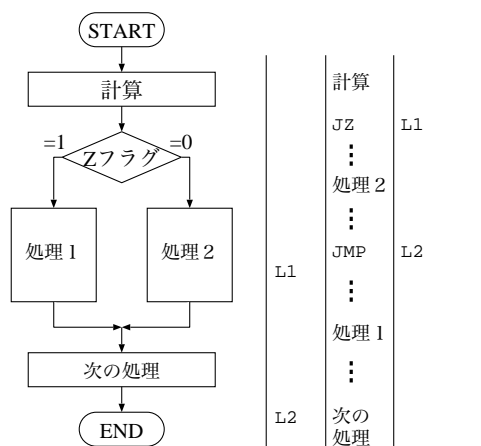
## 繰り返し処理 1

次のように、ジャンプ命令を用いて同じ操作を繰り返すプログラムを作ることができます。次のフローチャートとプログラムは、点線部分を 10 回繰り返すためのものです。



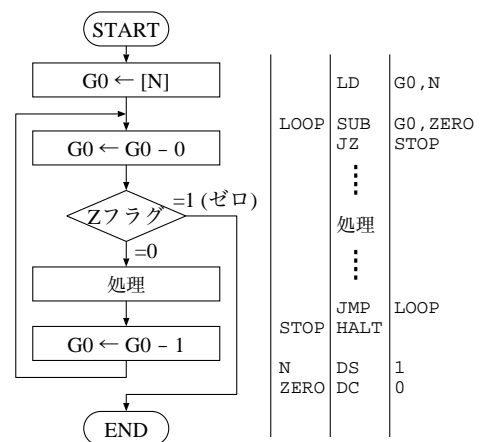
## 条件判断 2

条件によって、二つの処理のどちらかを選んで実行するプログラムを作ることができます。次のフローチャートとプログラムは、計算結果がゼロだった場合は「処理 1」を、ゼロ以外だった場合は「処理 2」を実行します。



## 繰り返し処理 2

計算結果によって繰り返し回数が決まる場合等、繰り返し回数がゼロの場合も考慮しなければならないことがあります。そのような場合は、条件判断を前に移動すると、うまく処理できます。次のフローチャートとプログラムは、点線部分を N 回繰り返すためのものです。条件判断の前の引算は、G0 の値でフラグを強制的に変化させるためのものです。



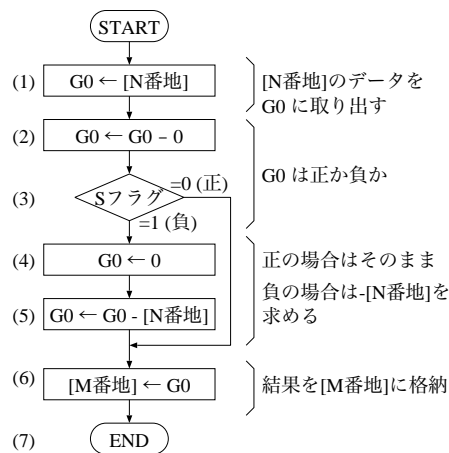
**例題 5-1** 絶対値を求める。

**問題：** N 番地のデータの絶対値を計算し，M 番地に格納するプログラムを作りなさい。

**考え方：** 値が正なのか負なのかは，値からゼロを (SUB 命令で) 引き，そのときのフラグの変化で調べます。

負の値の絶対値は，ゼロからその値を引くことで求めることができます。

**解答：** 次のようなプログラムを作ります。



注意：[N番地]は，N番地に格納されているデータのこと

| 番地 | 機械語   | ラベル   | ニーモニック |         |
|----|-------|-------|--------|---------|
| 00 | 10 10 | START | LD     | G0,N    |
| 02 | 40 0F |       | SUB    | G0,ZERO |
| 04 | AC 08 |       | JM     | L1      |
| 06 | A0 0C |       | JMP    | L2      |
| 08 | 10 0F | L1    | LD     | G0,ZERO |
| 0A | 40 10 |       | SUB    | G0,N    |
| 0C | 20 11 | L2    | ST     | G0,M    |
| 0E | FF    |       | HALT   |         |
| 0F | 00    | ZERO  | DC     | 0       |
| 10 | FF    | N     | DC     | -1      |
| 11 | 00    | M     | DS     | 1       |

**解説：** プログラムの内容を説明します。

- (1) G0 レジスタに値をロードします。
- (2) G0 レジスタから 0 を引きます。計算結果によりフラグが変化します。
- (3) JM 命令は S フラグの値によりジャンプします。
- (4), (5) この部分は，値が負だった場合のみ実行されます。0 - [N 番地] を計算し，値の絶対値を G0 に求めます。
- (6) G0 の値を M 番地に格納します。

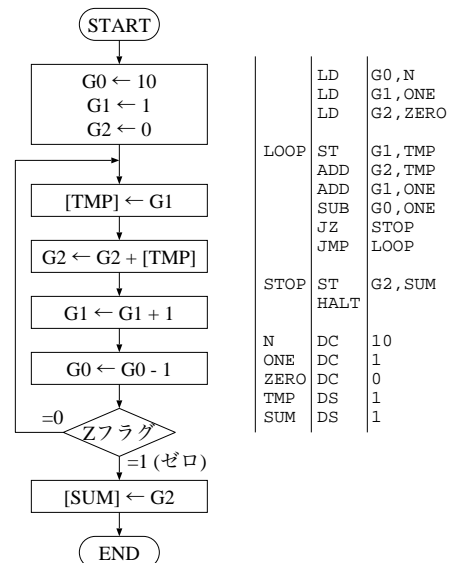
**例題 5-2** 1 + 2 + 3 + ... + 10 を計算する。

**問題：** 1 + 2 + 3 + ... + 10 を計算し結果を SUM 番地に求めるプログラムを作りなさい。

**考え方：** TeC では G0, G1, G2, SP の 4 つのレジスタが自由に使用できます。それぞれのレジスタに足す値を記憶する，合計を記憶する等の役割りを分担させ，繰り返し (10 回繰り返し) をうまく利用すると計算できます。

TeC にはレジスタの値同士を足し算する命令がありません。一方のレジスタ値をメモリに格納してから，足し算する必要があります。

**解答：** G0 を繰り返し回数のカウンタ，G1 をレジスタに足す数の記憶 (1, 2, 3, ..., 10)，G2 を合計の記憶に使用することにしました。SP は使用しませんでした。フローチャートとプログラムは次のようになります。

**問題**

1. N 番地の値が 0 なら M 番地に 0 を，そうでなければ M 番地に 1 を格納するプログラムを作りなさい。
2. N 番地と M 番地の値の大きい方を L 番地に格納するプログラムを作りなさい。
3. N 番地と M 番地の値のかけ算を計算し，L 番地に結果を格納するプログラムを作りなさい。

**ヒント：** かけ算は足し算の繰り返しでできます。

## 5.6.6 JNZ (Jump on Not Zero) 命令

Z フラグが 0 のとき (計算結果がゼロ以外だったとき) だけジャンプします。

**意味:** Z フラグが 0 ならジャンプします。

**フラグ:** 変化しません。

**ニーモニック:** JNZ A[,XR]

**命令フォーマット:** JNZ 命令は 2 バイトの長さを持ちます。GR フィールドは必ず 01<sub>2</sub> にします。第 2 バイトがジャンプ先のアドレスを示します。

| 第1バイト             |                    | 第2バイト     |
|-------------------|--------------------|-----------|
| OP                | GR XR              |           |
| 1011 <sub>2</sub> | 01 <sub>2</sub> XR | aaaa aaaa |

**フローチャート:** JNZ 命令のフローチャートも、JZ 命令と同様な考えで描きます。JZ 命令を参考にしてください。

**使用例:** JZ 命令の使用例で紹介した「結果がゼロだったら停止するプログラムの例」を JNZ 命令を使用して書き直しました。JNZ 命令を使用したほうが随分シンプルになります。

| 番地 | 機械語   | ラベル  | ニーモニック     |
|----|-------|------|------------|
| 00 | 30 05 | LOOP | ADD GO,ONE |
| 02 | B4 00 |      | JNZ LOOP   |
| 04 | FF    |      | HALT       |
| 05 | 01    | ONE  | DC 1       |

## 5.6.7 JNC (Jump on Not Carry) 命令

C フラグが 0 のときだけジャンプします。

**意味:** C フラグが 0 ならジャンプします。

**フラグ:** 変化しません。

**ニーモニック:** JNC A[,XR]

**命令フォーマット:** JNC 命令は 2 バイトの長さを持ちます。GR フィールドは必ず 10<sub>2</sub> にします。第 2 バイトがジャンプ先のアドレスを示します。

| 第1バイト             |                    | 第2バイト     |
|-------------------|--------------------|-----------|
| OP                | GR XR              |           |
| 1011 <sub>2</sub> | 10 <sub>2</sub> XR | aaaa aaaa |

**フローチャート:** JNC 命令のフローチャートも、JZ 命令と同様な考えで描きます。JZ 命令を参考にしてください。

## 5.6.8 JNM (Jump on Not Minus) 命令

S フラグが 0 のとき (計算結果がゼロか正だったとき) だけジャンプします。名前が、JNS 命令ではなく JNM 命令になっているので注意して下さい。

**意味:** S フラグが 0 ならジャンプします。

**フラグ:** 変化しません。

**ニーモニック:** JNM A[,XR]

**命令フォーマット:** JNM 命令は 2 バイトの長さを持ちます。GR フィールドは必ず 11<sub>2</sub> にします。第 2 バイトがジャンプ先のアドレスを示します。

| 第1バイト             |                    | 第2バイト     |
|-------------------|--------------------|-----------|
| OP                | GR XR              |           |
| 1011 <sub>2</sub> | 11 <sub>2</sub> XR | aaaa aaaa |

**フローチャート:** JNM 命令のフローチャートも、JZ 命令と同様な考えで描きます。JZ 命令を参考にしてください。

## 問題

- 「例題 5-1 絶対値を求める」プログラムを JNM 命令を用いて書き直しなさい。
  - 「例題 5-2 1+2+3+...+10 を計算する」プログラムを JNZ 命令を用いて書き直しなさい。
  - N 番地と M 番地の値のかけ算を計算し、L 番地に結果を格納するプログラムを JNZ 命令を用いて作り直しなさい。
  - N 番地と M 番地の値の和を計算し L 番地に結果を格納するプログラムを作りなさい。但し、オーバーフローが発生した場合は 8bit で表現できる最大の値 255 を L 番地に格納することとします。(数値は符号なし数とします。)
- 参考:** このような計算を飽和演算 (ほうわえんざん) と言います。
- 前の問題では JC 命令か JNC 命令を使用したはずですが、JC 命令を使用していた場合は JNC 命令、JNC 命令を使用していた場合は JC 命令を用いて作り直しなさい。

## 5.7 比較命令

### 5.7.1 CMP (Compare) 命令

Compare は、「比較をする」という意味の英語です。CMP は Compare の綴を縮めたものです。比較は引き算により行います。レジスタの値からメモリの値を引いて、その結果によりフラグを変化させます。引き算結果そのものは、どこにも格納しないで捨てます。

**意味：** レジスタの値とメモリデータを比較します。引き算の結果により、フラグだけが変化します。

**フラグ：** 計算結果により変化します。

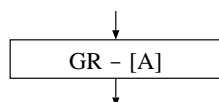
**ニーモニック：** CMP GR,A[,XR]

GR で指定されたレジスタの値と、メモリの値が比較されます。A と [,XR] の意味は LD 命令と同様です。

**命令フォーマット：** CMP 命令は 2 バイトの長さを持ちます。各フィールドの意味は、LD 命令と同様です。

| 第1バイト             |       | 第2バイト     |
|-------------------|-------|-----------|
| OP                | GR XR |           |
| 0101 <sub>2</sub> | GR XR | aaaa aaaa |

**フローチャート：** CMP 命令は次のように描きます。



**使用例：** A, B 二つのデータで大きい方を選んで、C に格納するプログラムの例です。

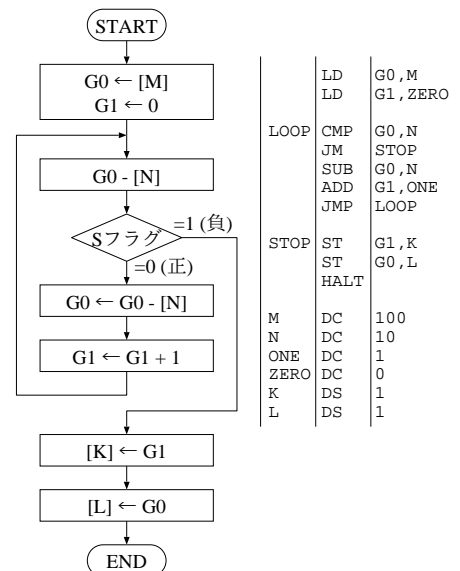
| 番地 | 機械語   | ラベル   | ニーモニック   |
|----|-------|-------|----------|
| 00 | 10 0D | START | LD GO,A  |
| 02 | 50 0E |       | CMP GO,B |
| 04 | AC 08 |       | JM LB    |
| 06 | A0 0A |       | JMP LC   |
| 08 | 10 0E | LB    | LD GO,B  |
| 0A | 20 0F | LC    | ST GO,C  |
| 0C | FF    |       | HALT     |
| 0D | 64    | A     | DC 100   |
| 0E | C8    | B     | DC 200   |
| 0F | 00    | C     | DS 1     |

**例題 5-3** 割算を計算する。

**問題：** M 番地の値を N 番地の値で割り、商を K 番地、余りを L 番地に求めるプログラムを作りなさい。

**考え方：** TeC には割算命令がありません。割算は引き算の繰り返しで計算できます。割られる数から、割る数を引くことを繰り返します。引くことができなくなったら終了します。このとき、引いた回数が商、引いた結果が余りになります。

**解答：** フローチャートとプログラムは次のようになります。



**解説：** CMP 命令と JM 命令で G0 が N 番地のデータより大きい確認してから、引き算を実行します。

### 問題

- 「例題 5-1 絶対値を求める」プログラムを CMP 命令を用いて書き直しなさい。
- 「例題 5-2 1+2+3+...+10 を計算する」プログラムを G0 レジスタを使用しないように書き直しなさい。(CMP 命令を用いると簡単にできる。)

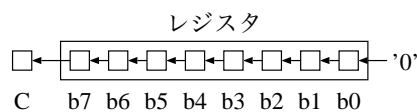


## 5.8 シフト（桁ずらし）命令

データの2進数を左右に桁移動する命令をシフト命令と言います。TeC は以下に説明する4種類のシフト命令を持っています。

### 5.8.1 SHLA (Shift Left Arithmetic) 命令

左算術（算術＝Arithmetic）シフトと言います。レジスタのデータを左方向に1ビットずらしします。右側から常に“0”が入力されます。左にはみ出したビットは、Cフラグの値になります。



**意味：** レジスタの値を左に1ビットシフトします。

**フラグ：** Cフラグは上の説明のように、S、Zフラグはシフト結果の値により変化します。

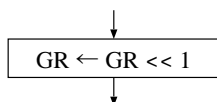
**ニーモニック：** SHLA GR

GRが、シフトされるレジスタを表します。GRの値とレジスタの対応は、LD命令等と同じです。

**命令フォーマット：** SHLA命令は1バイトの命令です。XRフィールドは必ず00<sub>2</sub>にします。

| 第1バイト             |                    |
|-------------------|--------------------|
| OP                | GR XR              |
| 1001 <sub>2</sub> | GR 00 <sub>2</sub> |

**フローチャート：** SHLA命令は次のように描くことにします。なお、この書き方は、Java言語やC言語のシフト演算子を真似たものです。



**使用例：** SHLA命令は次のSHLL命令と全く同じ動作をしますので、SHLA命令とSHLL命令で兼用の使用例をSHLL命令の後に示します。

### 5.8.2 SHLL (Shift Left Logical) 命令

左論理（論理＝Logical）シフトと言います。この命令はSHLAと全く同じ動作をします。（左シフトでは、算術と論理の差はない）

**意味：** レジスタの値を左に1ビットシフトします。

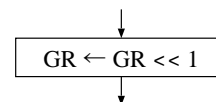
**フラグ：** SHLA命令と同様です。

**ニーモニック：** SHLL GR

**命令フォーマット：** SHLL命令は1バイトの命令です。XRフィールドは必ず01<sub>2</sub>にします。

| 第1バイト             |                    |
|-------------------|--------------------|
| OP                | GR XR              |
| 1001 <sub>2</sub> | GR 01 <sub>2</sub> |

**フローチャート：** SHLL命令は次のように描くことにします。SHLA命令と全く同じ動作をするので、フローチャートの描き方も、SHLA命令と全く同じです。

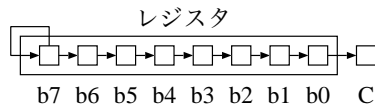


**使用例：** A番地のデータを2ビット左にシフトしB番地に格納する例です。左に2ビットシフトすることは×4を計算することと同じです。例ではSHLA命令を用いていますが、SHLL命令を用いても全く同じ結果になります。

| 番地 | 機械語   | ラベル | ニーモニック |       |
|----|-------|-----|--------|-------|
| 00 | 10 07 |     | LD     | G0, A |
| 02 | 90    |     | SHLA   | G0    |
| 03 | 90    |     | SHLA   | G0    |
| 04 | 20 08 |     | ST     | G0, B |
| 06 | FF    |     | HALT   |       |
| 07 | 01    | A   | DC     | 1     |
| 08 | 00    | B   | DS     | 1     |

## 5.8.3 SHRA (Shift Right Arithmetic) 命令

右算術（算術＝Arithmetic）シフトと言います。レジスタのデータを右方向に1ビットずらしします。左側からシフト前のデータの最上位ビット（符号ビット）と同じ値が入力されます。右にはみ出したビットは、Cフラグの値になります。



**意味：** レジスタの値を右に1ビット算術シフトします。

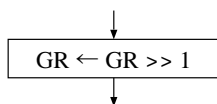
**フラグ：** SHLA 命令と同様です。

**ニーモニック：** SHRA GR

**命令フォーマット：** SHRA 命令は1バイトの命令です。XR フィールドは必ず 10<sub>2</sub> にします。

| 第1バイト             |                    |
|-------------------|--------------------|
| OP                | GR XR              |
| 1001 <sub>2</sub> | GR 10 <sub>2</sub> |

**フローチャート：** SHRA 命令は次のように描くことにします。

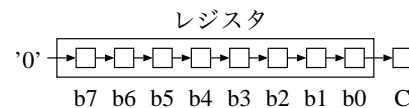


**使用例：** A, B 番地のデータを1ビット右にシフトし、C, D 番地に格納します。右1ビットシフトと、÷2の結果は同じです。算術シフトの場合は、負の数でも正しく 1/2 の値になります。

| 番地 | 機械語   | ラベル | ニーモニック   |
|----|-------|-----|----------|
| 00 | 10 0B |     | LD G0, A |
| 02 | 92    |     | SHRA G0  |
| 03 | 20 0D |     | ST G0, C |
| 05 | 14 0C |     | LD G1, B |
| 07 | 96    |     | SHRA G1  |
| 08 | 24 0E |     | ST G1, D |
| 0A | FF    |     | HALT     |
| 0B | 08    | A   | DC 8     |
| 0C | F8    | B   | DC -8    |
| 0D | 00    | C   | DS 1     |
| 0E | 00    | D   | DS 1     |

## 5.8.4 SHRL (Shift Right Logical) 命令

右論理（論理＝Logical）シフトと言います。レジスタのデータを右方向に1ビットずらしします。左側から“0”が入力されます。右にはみ出したビットは、Cフラグの値になります。



**意味：** レジスタの値を右に1ビット論理シフトします。

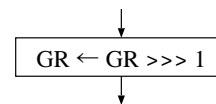
**フラグ：** SHLA 命令と同様です。

**ニーモニック：** SHRL GR

**命令フォーマット：** SHRL 命令は1バイトの命令です。XR フィールドは必ず 11<sub>2</sub> にします。

| 第1バイト             |                    |
|-------------------|--------------------|
| OP                | GR XR              |
| 1001 <sub>2</sub> | GR 11 <sub>2</sub> |

**フローチャート：** SHRL 命令は次のように描くことにします。



**使用例：** A, B 番地のデータを1ビット右にシフトし、C, D 番地に格納します。右1ビットシフトと、÷2は同じ結果になります。論理シフトの場合は、128以上の大きな数でも正しく 1/2 の値になります。

| 番地 | 機械語   | ラベル | ニーモニック   |
|----|-------|-----|----------|
| 00 | 10 0B |     | LD G0, A |
| 02 | 93    |     | SHRL G0  |
| 03 | 20 0D |     | ST G0, C |
| 05 | 14 0C |     | LD G1, B |
| 07 | 97    |     | SHRL G1  |
| 08 | 24 0E |     | ST G1, D |
| 0A | FF    |     | HALT     |
| 0B | 7F    | A   | DC 127   |
| 0C | 80    | B   | DC 128   |
| 0D | 00    | C   | DS 1     |
| 0E | 00    | D   | DS 1     |

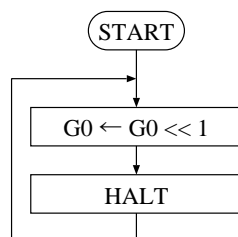
**例題 5-4** シフトの動作確認

**問題：** 4種類のシフト命令について，G0レジスタの値をシフトしながら動作を確認しなさい．

**プログラム：** 次のプログラムを作成します．

| 番地 | 機械語   | ラベル | ニーモニック  |
|----|-------|-----|---------|
| 00 | 90    | L0  | SHLA G0 |
| 01 | FF    |     | HALT    |
| 02 | A0 00 |     | JMP L0  |

**フローチャート：** フローチャートで描く場合は，次のように HALT を表現することにします．



**実行方法：** 次の手順で実行します．

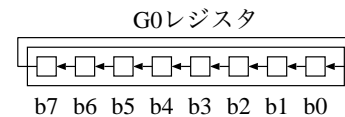
1. PC を 00<sub>16</sub> にする．
2. G0 レジスタに適当なデータを書き込む．
3. ロータリースイッチを G0 に合わせたまま，RUN ボタンを押す．
4. G0 の値がシフトされたことが，データランプで確認できる．
5. 再度，RUN ボタンを押す．
6. G0 の値が更にシフトされる．

一度 RUN ボタンを押すと HALT 命令で停止します．そのとき，PC は JMP 命令を指していますので，再度 RUN ボタンを押すと，もう一度，最初からプログラムが実行されます．

SHLA 命令の動作が良く分かったら別のシフト命令でも試してみましょう．

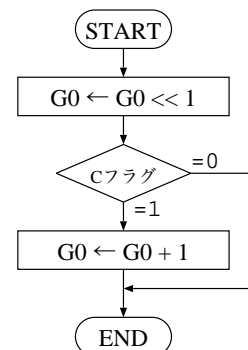
**例題 5-5** ビットの回転

**問題：** 次の図のように，G0 レジスタの値を 1 ビット左回転するプログラムを作りなさい．



**考え方：** TeC には，ビットを回転する命令がありません．次の手順で目的を達成します．

1. シフト命令で G0 を 1 ビットシフトする．
2. はみ出したビットが C フラグに保存されているので，C フラグを調べる．
3. C フラグが 1 だった場合は，G0 の最下位ビットを 1 にする．



**プログラム：** プログラムにすると次のようになります．**例題 5-4** と同様に最後に JMP 命令を追加しました．

| 番地 | 機械語   | ラベル | ニーモニック      |
|----|-------|-----|-------------|
| 00 | 91    | L0  | SHLL G0     |
| 01 | A8 05 |     | JC L1       |
| 03 | A0 07 |     | JMP L2      |
| 05 | 30 0A | L1  | ADD G0, ONE |
| 07 | FF    | L2  | HALT        |
| 08 | A0 00 |     | JMP L0      |
| 0A | 01    | ONE | DC 1        |

**実行方法：** **例題 5-4** と同様です．実行する度に，データランプの表示が回転します．

**例題 5-6** シフトを用いた高速乗算

**問題：** シフトを用いて、A の 10 倍の値を B に求めるプログラムを作りなさい。

**考え方：** TeC には、かけ算命令がありません。そのため、繰り返しでかけ算をする方法を、以前、紹介しました。ここでは、繰り返しによる方法より高速なかけ算を紹介します。  
×10 は、次の式のように変形できます。

$$B = A \times 10 = A \times 8 + A \times 2$$

×2 は、1 ビット左にシフトすることと同じです。×8 は、3 ビット左にシフトすることと同じです。シフトした結果同士を足し合わせることで、10 倍を計算することができます。

**プログラム：** プログラムにすると次のようになります。TMP は、一時的に A の 2 倍の値を記憶するために使用します。

| 番地 | 機械語   | ラベル | ニーモニック      |
|----|-------|-----|-------------|
| 00 | 10 0C |     | LD GO, A    |
| 02 | 90    |     | SHLA GO     |
| 03 | 20 0E |     | ST GO, TMP  |
| 05 | 90    |     | SHLA GO     |
| 06 | 90    |     | SHLA GO     |
| 07 | 30 0E |     | ADD GO, TMP |
| 09 | 20 0D |     | ST GO, B    |
| 0B | FF    |     | HALT        |
| 0C | 03    | A   | DC 3        |
| 0D | 00    | B   | DS 1        |
| 0E | 00    | TMP | DS 1        |

**問題**

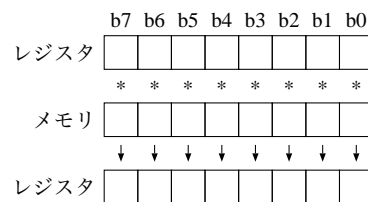
1. A の値の 7 倍を B に求めるプログラムを作りなさい。
2. A の値が符号無し 2 進数の場合、A の値の 1/4 倍を B に求めるプログラムを作りなさい。
3. A の値が符号無し 2 進数の場合、A の値の 1.5 倍を B に求めるプログラムを作りなさい。

**5.9 論理演算命令**

「2.8.1 論理演算と論理回路」で学んだ論理演算を行う機械語命令です。

**5.9.1 AND (Logical AND) 命令**

AND 命令は、レジスタとメモリデータのビット毎の論理積 (AND) を計算します。ビット毎の論理積とは、次の図のように、レジスタとメモリの対応するビット同士の論理積の計算のことです。計算結果はレジスタに格納されます。



**意味：** ビット毎の論理積を計算します。結果は元のレジスタに格納します。(メモリの値は変化しません。)

**フラグ：** C フラグは常に 0 になります。S、Z フラグは計算の結果により変化します。

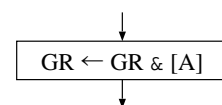
**ニーモニック：** AND GR, A[, XR]

GR は、計算の対象になるレジスタを表します。このレジスタの値とメモリの値の論理積が計算されます。結果はこのレジスタに格納されます。A と [, XR] の意味は LD 命令と同様です。

**命令フォーマット：** AND 命令は 2 バイトの長さを持ちます。各フィールドの意味は、LD 命令と同様です。

| 第1バイト             |       | 第2バイト     |
|-------------------|-------|-----------|
| OP                | GR XR |           |
| 0110 <sub>2</sub> | GR XR | aaaa aaaa |

**フローチャート：** AND 命令は次のように描くことにします。これも、Java 言語や C 言語の演算子を真似たものです。



**使用例：** A 番地のデータと B 番地のデータの論理積を計算し、C 番地に格納するプログラムの例を示します。

| 番地 | 機械語   | ラベル | ニーモニック   |
|----|-------|-----|----------|
| 00 | 10 07 |     | LD G0,A  |
| 02 | 60 08 |     | AND G0,B |
| 04 | 20 09 |     | ST G0,C  |
| 06 | FF    |     | HALT     |
| 07 | 63    | A   | DC 63H   |
| 08 | 0F    | B   | DC 0FH   |
| 09 | 00    | C   | DS 1     |

|         |    |    |    |    |    |    |    |    |
|---------|----|----|----|----|----|----|----|----|
|         | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| A = 63H | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  |
|         | *  | *  | *  | *  | *  | *  | *  | *  |
| B = 0FH | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  |
|         | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  |
| C = 03H | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  |

### AND 命令の応用

AND 命令は、データの特定のビットを 0 にクリアしたり、データの特定のビットの 1/0 を確かめたり、データの一部のビットだけを取り出したりするために使用できます。

1. 特定の 1 ビットがゼロかそうでないか判定できます。次は G0 の最下位ビット (LSB) がゼロならジャンプする例です。01<sub>16</sub> との AND の結果が 00<sub>16</sub> になることから、LSB が 0 だったことが分かります。

| ラベル | ニーモニック     |
|-----|------------|
|     | ...        |
|     | AND G0,ONE |
|     | JZ L1      |
|     | ...        |
| L1  | ...        |
|     | ...        |
| ONE | DC 01H     |

2. バイト中の連続した数ビットを取り出すことができます。次は G0 の、b3, b2 の 2 ビットだけを右詰めにして取り出す例です。下の図のように、まず AND 命令で目的の 2 ビット以外を 0 にします。次に右に 2 回シフトして右詰めにします。

| ラベル | ニーモニック     |
|-----|------------|
|     | ...        |
|     | AND G0,MSK |
|     | SHRL GO    |
|     | SHRL GO    |
|     | ...        |
| MSK | DC 0CH     |

### 16 進数の表記

ニーモニック中で数値を 16 進数で書き表したいことがあります。そのときは、前のプログラム中の DC 命令のように、数値の後ろに “H” を付けます。16 進数が “A”～“F” で始まると数値なのかラベルなのか分かりません。このような場合は数値であることが分かるように先頭に “0” を付加します。例えば F8H は 0F8H と書きます。

|         |    |    |    |    |    |    |    |    |
|---------|----|----|----|----|----|----|----|----|
|         | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|         | X  | X  | X  | X  | X  | X  | X  | X  |
|         | *  | *  | *  | *  | *  | *  | *  | *  |
|         | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  |
|         | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  |
| ANDの結果  | 0  | 0  | 0  | 0  | X  | X  | 0  | 0  |
|         | ↘  | ↘  | ↘  | ↘  | ↘  | ↘  | ↘  | ↘  |
| SHRLの結果 | 0  | 0  | 0  | 0  | 0  | X  | X  | 0  |
|         | ↘  | ↘  | ↘  | ↘  | ↘  | ↘  | ↘  | ↘  |
| SHRLの結果 | 0  | 0  | 0  | 0  | 0  | 0  | X  | X  |

3. 2<sup>n</sup> の倍数になるように切り捨てることができます。次は G0 の値を 2<sup>3</sup> = 8 の倍数になるように切り捨てる例です。下 3 ビットをゼロにすると 8 の倍数になります。

| ラベル | ニーモニック     |
|-----|------------|
|     | ...        |
|     | AND G0,MSK |
|     | ...        |
| MSK | DC 0F8H    |

4. 2<sup>n</sup> で割った余りを求めることができます。次は G0 の値を 2<sup>3</sup> = 8 で割った余りを求める例です。下 3 ビットだけを取り出すと 8 で割った余りになります。

| ラベル | ニーモニック     |
|-----|------------|
|     | ...        |
|     | AND G0,MSK |
|     | ...        |
| MSK | DC 07H     |

このように、2 の累乗 (2,4,8,16,...) で割った商や余りは、シフトや論理演算で簡単に計算できます。また、2 の累乗の倍数に切捨てたり、切り上げたりする計算も、論理演算を使うと簡単にできます。

## 5.9.2 OR (Logical OR) 命令

OR 命令は、レジスタとメモリデータの対応するビット同士の論理和 (OR) を計算します。計算結果はレジスタに格納されます。

**意味：** ビット毎の論理和を計算します。結果は元のレジスタに格納します。(メモリの値は変化しません。)

**フラグ：** C フラグは常に 0 になります。S, Z フラグは計算の結果により変化します。

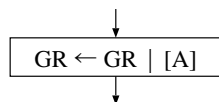
**ニーモニック：** OR GR,A[,XR]

GR は、計算の対象になるレジスタを表します。このレジスタの値とメモリの値の論理和が計算されます。結果はこのレジスタに格納されます。A と [,XR] の意味は LD 命令と同様です。

**命令フォーマット：** OR 命令は 2 バイトの長さを持ちます。各フィールドの意味は、LD 命令と同様です。

| 第1バイト             |       | 第2バイト     |
|-------------------|-------|-----------|
| OP                | GR XR |           |
| 0111 <sub>2</sub> | GR XR | aaaa aaaa |

**フローチャート：** OR 命令は次のように描くことにします。これも、Java 言語や C 言語の演算子を真似したものです。



## OR 命令の応用

OR 命令は、データの特定のビットを 1 にするために使用できます。次の例は、G0 の上位 4 ビットを全て 1 にします。

| ラベル  | ニーモニック     |
|------|------------|
|      | ...        |
|      | LD G0,DATA |
|      | OR G0,MSK  |
|      | ...        |
| DATA | DC OAAH    |
| MSK  | DC OF0H    |

|       |    |    |    |    |    |    |    |    |
|-------|----|----|----|----|----|----|----|----|
|       | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|       | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
|       | +  | +  | +  | +  | +  | +  | +  | +  |
|       | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  |
|       | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  |
| ORの結果 | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  |

## 5.9.3 XOR (Logical XOR) 命令

XOR 命令は、レジスタとメモリデータの対応するビット同士の排他的論理和 (XOR) を計算します。計算結果はレジスタに格納されます。

**意味：** ビット毎の排他的論理和を計算します。結果は元のレジスタに格納します。(メモリの値は変化しません。)

**フラグ：** C フラグは常に 0 になります。S, Z フラグは計算の結果により変化します。

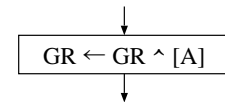
**ニーモニック：** XOR GR,A[,XR]

GR は、計算の対象になるレジスタを表します。このレジスタの値とメモリの値の排他的論理和が計算されます。結果はこのレジスタに格納されます。A と [,XR] は LD 命令と同様です。

**命令フォーマット：** XOR 命令は 2 バイトの長さを持ちます。各フィールドの意味は LD 命令と同様です。

| 第1バイト             |       | 第2バイト     |
|-------------------|-------|-----------|
| OP                | GR XR |           |
| 1000 <sub>2</sub> | GR XR | aaaa aaaa |

**フローチャート：** XOR 命令は次のように描きます。



## XOR 命令の応用

XOR 命令を使用してデータの特定ビットの 0/1 を反転できます。TeC には NOT 命令がないので XOR 命令で代用します。次は G0 の上位 4 ビットの 0/1 を反転します。

| ラベル  | ニーモニック     |
|------|------------|
|      | ...        |
|      | LD G0,DATA |
|      | XOR G0,MSK |
|      | ...        |
| DATA | DC OAAH    |
| MSK  | DC OF0H    |

|        |    |    |    |    |    |    |    |    |
|--------|----|----|----|----|----|----|----|----|
|        | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|        | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
|        | ⊕  | ⊕  | ⊕  | ⊕  | ⊕  | ⊕  | ⊕  | ⊕  |
|        | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  |
|        | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  | ↓  |
| XORの結果 | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  |

### 5.10 アドレッシングモード

LD, ST, ADD, SUB, CMP, AND, OR, XOR, JMP, JZ, JC, JM, JNZ, JNC, JNM 命令は、どれも次のような同じフォーマットでした。これまで、これらの命令の  $XR$  部分は  $00_2$  にしてきました。

| 第1バイト |       | 第2バイト     |
|-------|-------|-----------|
| OP    | GR XR |           |
| OP    | GR XR | aaaa aaaa |

$XR$  に  $00_2$  以外を指定することにより、メモリアドレス表現方法を変更することができます。このアドレス表現方法のことをアドレッシングモードと呼びます。TeCで使用できるアドレッシングモードは、 $XR$  の値により次の4種類があります。

| XR     | 意味           |            |
|--------|--------------|------------|
| $00_2$ | ダイレクトモード     | (直接モード)    |
| $01_2$ | G1 インデクストモード | (G1 指標モード) |
| $10_2$ | G2 インデクストモード | (G2 指標モード) |
| $11_2$ | イミディエイトモード   | (即値モード)    |

(同様な表が「図 B.1 TeC7 命令表」にも掲載されているので、そちらも確認してください。)

以下では LD 命令と ST 命令を例に、4 種類のアドレッシングモードを説明します。ここでは説明しませんが、ADD, SUB, CMP, AND, OR, XOR 命令でも、LD 命令や ST 命令と同様にアドレッシングモードが使用できます。

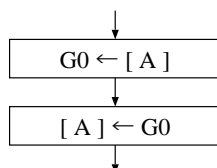
#### 5.10.1 ダイレクト (直接) モード

これまで使用してきたのは、このモードです。命令の第2バイトがメモリアドレスを直接に表します。

ニーモニックでは次のように書きました。ここで、A, B は、データを置いたメモリのアドレスです。

```
LD  G0,A
ST  G0,B
```

フローチャートでは、次のように描きました。



#### 5.10.2 インデクスト (指標) モード

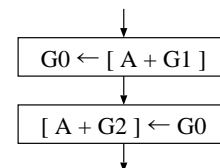
G1 または G2 レジスタの値と、命令の第2バイトの値の合計がメモリアドレスを表します。 $XR$  の値により、G1, G2 どちらのレジスタを使用するか決まります。G0 レジスタは使用できないので注意してください。

このモードは、配列データをアクセスするとき使用できます。また、ジャンプ命令でも使用できます。ジャンプ命令でインデクストモードを使用する方法は、「6.7.3 ジャンプテーブル」で説明します。

ニーモニックでは次のように書きます。このプログラムは、 $A + G1$  番地のデータを G0 に読込み、 $A + G2$  番地に格納しています。

```
LD  G0,A,G1
ST  G0,A,G2
```

フローチャートでは、次のように描きます。



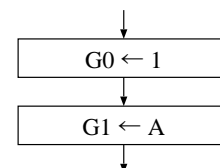
#### 5.10.3 イミディエイト (即値) モード

第2バイトがデータそのものになるモードです。ST 命令やジャンプ命令では使用できません。(これらの命令で  $XR$  を 11 にすると、CPU が命令を実行しようとしたとき、命令コードのエラーになり RUN ランプが点滅します。)

ニーモニックでは次のように書きます。

```
LD  G0,#1
LD  G1,#A
```

最初の LD 命令は、G0 を 1 にします。次の LD 命令は、G1 を A の番地 (内容ではない) にします。フローチャートでは、次のように描きます。





## 5.10.4 アドレッシングモードの使用例

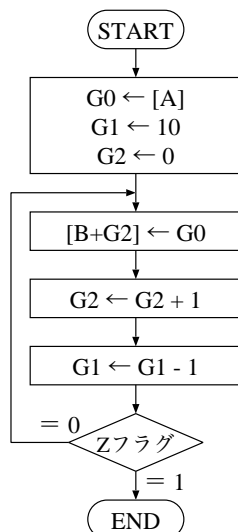
A 番地のデータで、B 番地からの 10 バイトの配列を初期化するプログラムの例を示します。

G1, G2 レジスタの初期化と G1, G2 レジスタの値を ± 1 する部分にイミディエイトモードを使用しました。

また、10 回の繰り返して 10 バイトの領域を初期化できるように、ST 命令の格納アドレスが G2 の値で 1 番地ずつずれていきます。ずらすためにインデックスモードを使用しました。

| 番地 | 機械語            | ラベル  | ニーモニック     |
|----|----------------|------|------------|
| 00 | 10 11          |      | LD G0,A    |
| 02 | 17 0A          |      | LD G1,#10  |
| 04 | 1B 00          |      | LD G2,#0   |
| 06 | 22 12          | LOOP | ST G0,B,G2 |
| 08 | 3B 01          |      | ADD G2,#1  |
| 0A | 47 01          |      | SUB G1,#1  |
| 0C | A4 10          |      | JZ STOP    |
| 0E | A0 06          |      | JMP LOOP   |
| 10 | FF             | STOP | HALT       |
| 11 | AA             | A    | DC OAAH    |
| 12 | 00 00 00 00 00 | B    | DS 10      |
| 17 | 00 00 00 00 00 |      |            |

次に、このプログラムのフローチャートを示します。イミディエイトモードとインデックスモードを、フローチャート上でどのように表現しているか、よく確認してください。



## 問題

1. これまでに出てきたプログラムをイミディエイトモードを使用して書き換えなさい。
2. A 番地からの 5 バイトのデータの合計を B 番地に求めるプログラムを作りなさい。
3. A 番地からの 5 バイトのデータを B 番地からの 5 バイトにコピーするプログラムを作りなさい。
4. A 番地からの 5 バイトのデータのビットを反転するプログラムを作りなさい。
5. A 番地と B 番地からの 5 バイトの論理和を C 番地からの 5 バイトに求めるプログラムを作りなさい。
6. A 番地からの 5 バイトのデータのなかで、奇数の個数を B 番地に求めるプログラムを作りなさい。  
(ヒント：奇数の LSB は必ず 1 になっている。)
7. A 番地からの 5 バイトのデータの最大の値を B 番地に求めるプログラムを作りなさい。
8. A 番地からの 5 バイトのデータの合計を B 番地に求めるプログラムをインデックスモードを使用しないで作りなさい。  
(ヒント：プログラムが自身を書き換えながら動作する。)

表 5.1 I/O マップ

| I/O マップ |           |             |
|---------|-----------|-------------|
| 番地      | Read      | Write       |
| 0       | データスイッチ   | ブザー         |
| 1       | データスイッチ   | スピーカ        |
| 2       | SIO 受信データ | SIO 送信データ   |
| 3       | SIO ステータス | SIO コントロール  |
| 4       | タイマ現在値    | タイマ周期       |
| 5       | タイマステータス  | タイマコントロール   |
| 6       | 空き        | INT3 コントロール |
| 7       | 入力ポート     | 出力ポート       |
| 8       | ADC CH0   | 空き          |
| 9       | ADC CH1   | 空き          |
| A       | ADC CH2   | 空き          |
| B       | ADC CH3   | 空き          |
| C       | 空き        | 空き          |
| D       | 空き        | 空き          |
| E       | 空き        | 空き          |
| F       | 空き        | 空き          |

## 5.11 入出力

「図 5.2 TeC の構成」に示した主記憶は、LD、ST 命令等を使用することでアクセスできることが分かりました。ここで説明する入出力 (Input Output=略して I/O) 命令は、入出力インターフェース回路をアクセスするための命令です。読み込みを行う IN 命令、書き込みを行う OUT 命令の 2 種類があります。

### 5.11.1 I/O マップ

いくつか存在するインターフェース回路の中で、どれをアクセスするかは I/O アドレス (メモリアドレスとは別のもの) により指定します。TeC の I/O アドレスは  $0_{16} \sim F_{16}$  までの 16 番地です。I/O アドレスの一覧を「I/O マップ」と言います。TeC の「I/O マップ」を表 5.1 に示します。

「表 5.1 I/O マップ」から次のことが分かります。(詳しくは、後の方で説明します。)

1. 0 番地の Read (読み込み) により、データスイッチの値を読み込むことができる。
2. 0 番地の Write (書き込み) により、ブザーにアクセスできる。
3. 1 番地の Read (読み込み) でも、データスイッチの値を読み込むことができる。

4. 1 番地の Write (書き込み) により、スピーカーにアクセスできる。
5. 2, 3 番地は、SIO (シリアル入出力) を用いてパソコンと通信するために使用される。
6. 4, 5 番地は、タイマを使用して時間を計るために使用される。
7. 6 番地は、コンソールパネルから発生する INT3 割込を制御する。
8. 7 番地は、入出力ポートの値を読み書きするために使用される。
9. 8, 9, A, B 番地は ADC (A/D コンバータ: 電圧を計測して数値に変換する回路) の値を読み取るために使用される。
10. C, D, E, F 番地は使用されない。

I/O マップは「図 B.1 TeC7 命令表」にも掲載されているので確認してください。

### 5.11.2 IN (Input) 命令

入出力インターフェース回路からデータを入力します。

**意味:** 4 ビットの I/O アドレスで指定した入出力インターフェース回路からデータをレジスタに入力します。

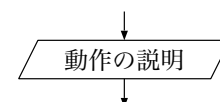
**ニーモニック:** IN GR,P

GR は値を入力するレジスタを表します。P は入出力インターフェース回路の I/O アドレスです。

**命令フォーマット:** IN 命令は 2 バイトの長さを持ちます。XR フィールドは必ず  $00_2$  にします。第 2 バイトの上位 4 ビットも必ず  $0000_2$  にします。

| 第1バイト    |           | 第2バイト         |
|----------|-----------|---------------|
| OP       | GR XR     |               |
| $1100_2$ | GR $00_2$ | $0000_2$ pppp |

**フローチャート:** IN 命令は次のように描くことにします。「動作の説明」は各自が工夫して下さい。



**使用例:** 次のプログラムはデータスイッチの状態を I/O アドレス 0 番地から読み取り、G0 レジスタに格納するものです。ロータリースイッチを G0 に合わせた状態で、プログラムを実行してください。

データスイッチを変化させると即座にデータランプの表示が変化します。

プログラムは無限ループになっているので、実行を開始したら STOP ボタンを押すまで止まりません。

| ラベル   | ニーモニック               |
|-------|----------------------|
| START | IN G0,0<br>JMP START |

### IN 命令の応用

次のプログラムは、データスイッチから数値を入力し、合計を G0 に求めるものです。ロータリースwitchを G0 に合わせて実行すると、2 進数入力、2 進数表示の簡単な電卓として使用できます。

操作手順は次の通りです。

1. プログラムを入力する。
2. PC にプログラムの実行開始番地をセットする。
3. ロータリースwitchを G0 に合わせる。
4. データスイッチにデータをセットする。
5. RUN ボタンを押す。
6. データの件数分、4, 5 を繰り返す。
7. データランプに合計が表示されている。

| ラベル   | ニーモニック                                                   |
|-------|----------------------------------------------------------|
| START | LD G0,#0                                                 |
| LOOP  | IN G1,00H<br>ST G1,TMP<br>ADD G0,TMP<br>HALT<br>JMP LOOP |

### 問題

1. データスイッチから 0 で終了する値を次々入力し、合計を X 番地に求めるプログラムを作りなさい。
2. A 番地からの 5 バイトを、実行開始時にデータスイッチにセットしてあった値で埋めつくすプログラムを作りなさい。

### 5.11.3 OUT (Output) 命令

入出力インターフェース回路へデータを出力します。

**意味：** レジスタのデータを 4 ビットの I/O アドレスで指定した入出力インターフェース回路へ出力します。

**ニーモニック：** OUT GR,P

GR は出力する値を持っているレジスタを表します。

P は入出力インターフェース回路のアドレスです。

**命令フォーマット：** OUT 命令は 2 バイトの長さを持ちます。XR フィールドは必ず 11<sub>2</sub> にします。第 2 バイトの上位 4 ビットも必ず 0000<sub>2</sub> にします。

| 第 1 バイト           |                    | 第 2 バイト                |
|-------------------|--------------------|------------------------|
| OP                | GR XR              |                        |
| 1100 <sub>2</sub> | GR 11 <sub>2</sub> | 0000 <sub>2</sub> pppp |

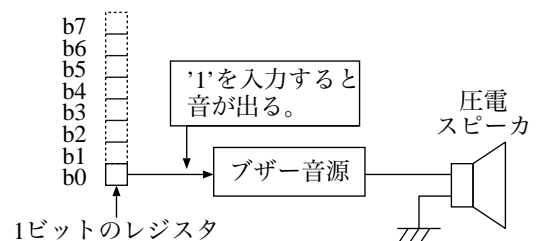
**使用例：** 次のプログラムはデータスイッチの値を読み取り、ブザー用のポートに出力するものです。

ブザーは図のような構造になっており、I/O アドレス 0 番地に書き込んだ値で鳴ったり止まったりします。プログラム実行中にデータスイッチを操作して、最下位ビットを 1 にするとブザーが鳴ります。逆に最下位ビットを 0 にするとブザーの音が止まります。

プログラムは無限ループになっているので、実行を開始したら STOP ボタンを押すまで止まりません。ブザーが鳴っている状態でプログラムを停止しても、ブザーが鳴りっぱなしになります。その場合は、RESET スイッチを押して下さい。

| ラベル   | ニーモニック                               |
|-------|--------------------------------------|
| START | IN G0,00H<br>OUT G0,00H<br>JMP START |

I/O アドレス 0 番地



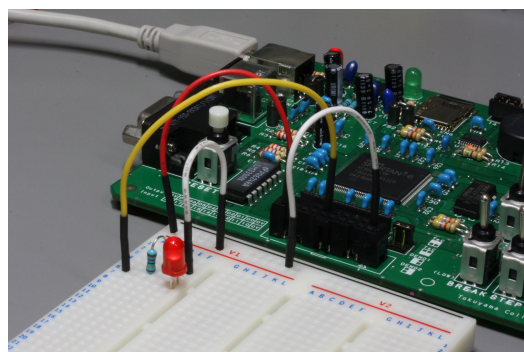
## 問題

1. データスイッチのビット 7(D7) をオンにしている間、ブザーを鳴らすプログラムを作りなさい。  
(但し、論理演算命令を用いること.)
2. データスイッチのビット 7(D7) をオンにしている間、ブザーを鳴らすプログラムを作りなさい。  
(但し、シフト命令を用いること.)
3. データスイッチのビット 0(D0) をオンにしている間、ブザーを鳴らすプログラムを作りなさい。  
(但し、データスイッチの 7(D7) がオンになったら、ブザーを止めて終了するようにすること.)

## 5.12 TeC パラレル入出力 (PIO)

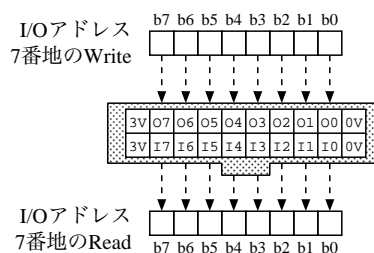
TeC には「図 4.1 各部の名称」の「入出力ポートコネクタ」を通し 8 ビットのデータを並列に入力・出力するパラレル入出力 (Parallel Input Output : 略して **PIO**) 機能があります。

次の写真は PIO を使用し、ブレッドボード上の LED をプログラムで点灯・消灯できるようにした例です。

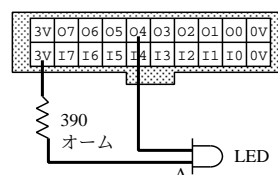


下に入出力ポートコネクタのピン配置図を示します。I/O アドレスの 7 番地に OUT 命令で書込んだ 2 進数データがコネクタの  $O_7 \sim O_0$  に出力されます。(対応するビットの “1” が 3.3V, “0” が 0V になります。) I/O アドレスの 7 番地を IN 命令で読むとコネクタの  $I_7 \sim I_0$  に入力された電圧に対応したデータを入力できます。

入出力ポートコネクタの “0V” ピンは TeC の GND に接続されています。“3V” ピンは TeC の 3.3V 電源に接続されています。外部に接続した回路の電源として使用することができます。



上の写真では、ブレッドボードを用いて次図のように配線してあります。ポートに “0” を出力すると LED が光ります。



### 5.13 TeC アナログ入力 (AIN)

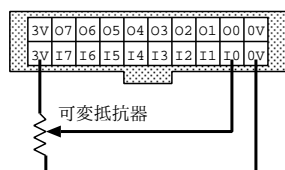
TeC には、基板上の入出力ポートコネクタ (図 4.1 参照) を通してアナログ電圧値を入力できる機能 (Analog INput : 略して **AIN**) があります。

TeC の AIN には、同時に 4 つのアナログ電圧が入力できます。アナログ電圧は入出力ポートコネクタの  $I_3$ ,  $I_2$ ,  $I_1$ ,  $I_0$  ピンから入力します。入力電圧は、0V~3.3V を 256 等分した数値 (0V=0, 3.3V=255) に変換されます。数値に変換された結果は、I/O アドレス 8H~BH で読み取ることができます。次の表に、入力ピンと I/O アドレスの対応をまとめます。

| I/O 番地          | 名称      | 入力ピン  |
|-----------------|---------|-------|
| 8 <sub>16</sub> | ADC CH0 | $I_0$ |
| 9 <sub>16</sub> | ADC CH1 | $I_1$ |
| A <sub>16</sub> | ADC CH2 | $I_2$ |
| B <sub>16</sub> | ADC CH3 | $I_3$ |

入出力ポートコネクタの  $I_3$ ,  $I_2$ ,  $I_1$ ,  $I_0$  ピンは、パラレル入力にもアナログ入力にも使用されます。パラレル入力ポートから見ると、入力電圧が 1.6V 付近を境界に “0” と “1” が切り替わります。

次にアナログ電圧を入力する配線例を示します。可変抵抗器を変化させると入力値が変化します。入力値をプログラムが読み取るためには、I/O アドレスの 8H 番地 (ADC CH0) を IN 命令で読みます。



#### 定数の定義 (EQU 命令)

IN 命令の使用例ではデータスイッチの値を読み込むための I/O アドレス “0” がプログラム中に直接書いてありました。データスイッチのことだと分かりやすい名前を用いるとプログラムが読みやすくなります。EQU 命令は、名前 (ラベル) を定義するための命令です。EQU 命令のオペランドの値をラベルに割り付けます。他の命令と異なり、機械語やデータを生成しません。

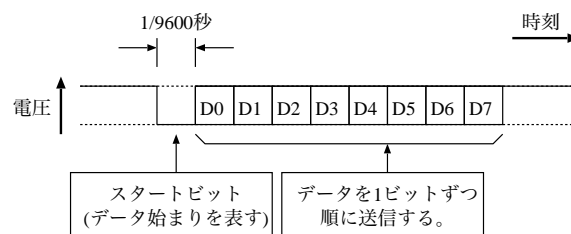
| 番地 | 機械語   | ラベル   | ニーモニック    |
|----|-------|-------|-----------|
| 00 |       | DSW   | EQU 00H   |
| 00 | C0 00 | START | IN GO,DSW |
| 02 | A0 00 |       | JMP START |

### 5.14 TeC シリアル入出力 (SIO)

TeC には、パソコンと接続できるシリアル入出力 (Serial Input Output : 略して **SIO**) インターフェースが備えてあります。前出の IN, OUT 命令を使用して、SIO インターフェースにアクセスします。

#### 5.14.1 シリアル入出力

名前の通り (Serial = 直列) データを 1 ビット毎、転送する方式です。次の図のように、時間による電圧の変化としてデータを表現します。また、1 ビットの時間として 1/9600 秒を用います。1/9600 秒で 1 ビットを送る通信速度を 9600bau (ボー) と言います。

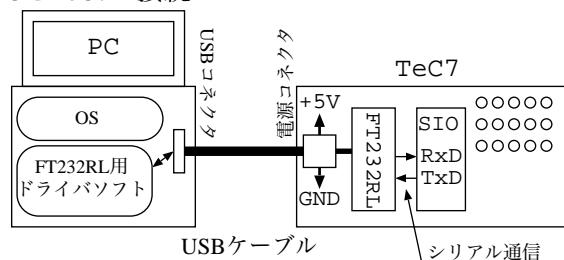


#### 5.14.2 パソコンとの接続

パソコンと接続するときは、TeC の電源コネクタとパソコンの USB コネクタを接続します。TeC には「USB シリアル変換 IC」として FTDI 社の FT232RL が実装されているので、パソコンから電源が供給されると同時にシリアル通信も可能になります。<sup>\*1</sup>

次の図にシリアル通信をするときのパソコンと TeC の関係を示します。FT232RL と TeC の SIO インタフェース回路の間がシリアル通信部分になります。TeC のプログラムは SIO インタフェース回路を操作するだけなので、その先に FT232RL や USB ケーブルがあることは分かりません。

#### PC と TeC7 の接続



<sup>\*1</sup> パソコンの OS によっては FT232RL 用のドライバをインストールする必要があります。その場合は、FTDI 社の Web (<http://www.ftdichip.com/>) から VCP ドライバを入手してパソコンにインストールしてください。

## 5.14.3 I/O ポート

SIO の I/O ポートは、2 番地、3 番地に配置されています（「表 5.1 I/O マップ」参照）。内容は次の通りです。

## 受信データ

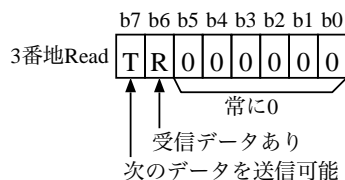
2 番地を IN 命令で読むと受信データを読み出すことができます。

## 送信データ

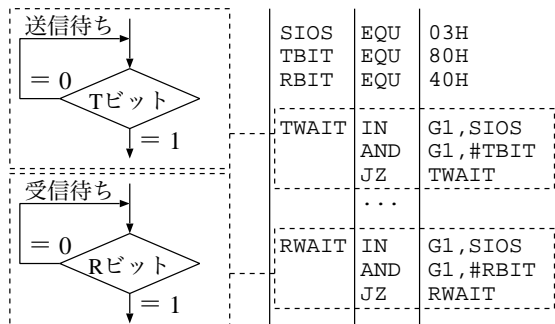
2 番地に OUT 命令で送信データを書き出すことができます。

## ステータス

3 番地を IN 命令で読むと次図のようなデータを読み込むことができます。各ビットの意味も図の通りです。

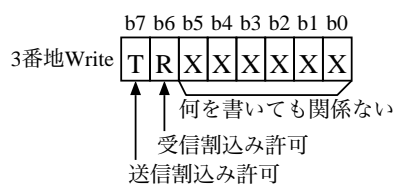


プログラムは次の手順でステータスを調べ、データの入出力が可能か判断します。この判断は頻繁に使用するので、フローチャートを次のように簡単化して描くことにします。



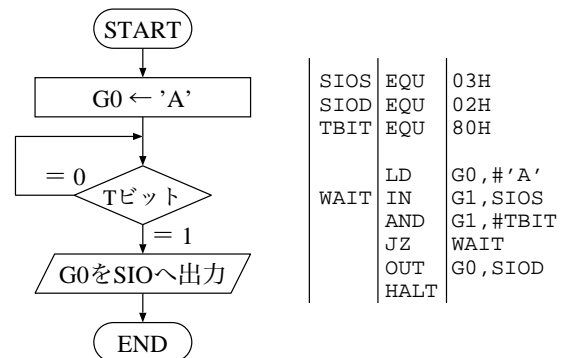
## コントロール

3 番地へ OUT 命令で次図のようなデータを書き込みます。詳細は、「6.11 入出力割込み」で説明します。



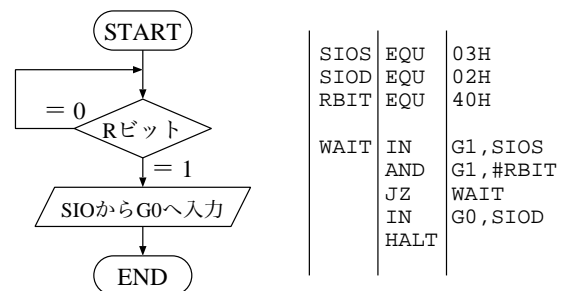
## 5.14.4 シリアル出力プログラム

シリアル出力を行うプログラムの一般的なフローチャートを次の図に示します。I/O アドレス 3 番地の「T ビット」が「1」になり、送信可能になるのを待って出力データを 2 番地に書き込みます。T ビットが 1 か 0 か調べるために、AND 命令を使用しています。これで、2 番地に書き込んだデータがパソコンに送信されます。



## 5.14.5 シリアル入力プログラム

シリアル入力を行うプログラムの、一般的なフローチャートを次の図に示します。受信データが届き、I/O アドレス 3 番地の「R ビット」が「1」になるのを待って、2 番地からデータを読み込みます。受信したデータは G0 に格納されます。



## 5.14.6 シリアル入出力データ

TeC がパソコンと通信するときは、「2.6 文字の表現」で勉強した ASCII コードを使用する約束とします。パソコンのキーボードを叩くと対応する文字の文字コードが TeC に送信され、TeC が文字コードを送信すると対応する文字がパソコンの画面に表示されます。パソコンに、このように動作するターミナルエミュレータを動かしておく必要があります。



### ターミナルエミュレータ

その昔、ホストコンピュータのディスプレイとキーボードの役目をした文字端末（下の写真）をシミュレーションするソフトウェアのことです。文字端末は現代のパソコンのような外観ですが、ホストコンピュータから受信した文字を表示するディスプレイと、ホストコンピュータに送信する文字をタイプするキーボードの機能しかありません。UNIX または macOS の**ターミナル**や Windows の **Tera Term** 等がターミナルエミュレータの一種です。

ここでは TeC がホストコンピュータの役割になります。キーボードやディスプレイを持たない TeC は、シリアル入出力を通して文字端末のキーボードとディスプレイを使用します。ターミナルエミュレータを動作させたパソコンは TeC の文字端末として働きます。ターミナルエミュレータは TeC から受信したデータを ASCII 文字コード（「2.6 文字の表現」参照）とみなし、対応する文字を画面に表示します。また、キーボードが押されると、押された文字の ASCII 文字コードを TeC へ送信します。

**Tera Term** はパソコンと別のコンピュータが通信することを主に作っていますが、UNIX や macOS の**ターミナル**はターミナルエミュレータが動作するのと同じパソコンをホストコンピュータと見做します。そこで、UNIX または macOS ではシリアル通信をターミナルエミュレータに中継する **screen プログラム**や **tip プログラム**と組合せて使用します。

TeC の SIO を使用するときには、TeC とパソコンをケーブルで接続するだけでなく、パソコンにターミナルエミュレータ等を起動し、パソコンを文字端末として使用できるように準備する必要があります。



写真: <http://commons.wikimedia.org/wiki/File:Televideo925Terminal.jpg> (パブリックドメイン)

### ニーモニック中の文字データ表記

文字を扱うプログラムを書くとき、文字コードが必要になります。そのときは、ニーモニック中に数値の代わりに、「'A'」のように書くことができます。文字 A の文字コード (41H) という意味です。（文字コードには ASCII コードを用います。）

ニーモニック中の 'A' は、41H または 65 と書いたのと全く同じ意味になります。

DC 命令では、連続した文字の定義に**"文字列"**を使用できます。下のプログラム中にあるように、**"TeC"**の表記で 3 バイトのデータが生成されます。

| 番地 | 機械語   | ラベル   | ニーモニック      |
|----|-------|-------|-------------|
| 00 | 13 41 | START | LD GO, #'A' |
| 02 | FF    |       | HALT        |
| 03 | 42 43 |       | DC 'B', 'C' |
| 05 | 45 65 |       | DC "TeC"    |
| 07 | 43    |       |             |

### ターミナルエミュレータ画面の改行

ターミナルエミュレータの表示を改行するには、復帰コード (CR) と改行コード (LF) の 2 文字を TeC からパソコンへ送るのが一般的です。

復帰コード (CR: 文字コードは 0DH) は文字カーソルを左端に移動するための文字、改行コード (LF: 文字コードは 0AH) は文字カーソルを一行下に移動するための文字です。

しかし、実際に筆者のパソコンで試してみると、改行コードのみの送信で復帰と改行の両方ができてしまいました。（復帰コードを送っても害はありません。）

ターミナルエミュレータにより、必要な復帰コードと改行コードの組合せが変化するようです。自分が実際に使用するシステムに合わせて、臨機応変に対応してください。

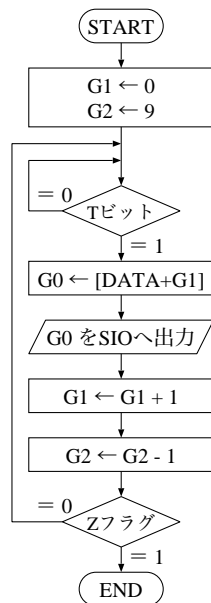


**例題 5-7** 文字列出力 1

**問題：** SIO ヘローマ字で自分の名前を出力するプログラムを作りなさい。

**考え方：** 筆者の場合、SHIGEMURA の 9 文字を出力するプログラムを作成します。データとして準備した 9 文字を、インデクスドモードのアドレッシングを使用し順に出力します。

**フローチャート：** 次のようになります。



**プログラム：**

| 番地 | 機械語   | ラベル   | ニーモニック         |
|----|-------|-------|----------------|
| 02 |       | SIOD  | EQU 02H        |
| 03 |       | SIOS  | EQU 03H        |
| 00 | 17 00 | START | LD G1,#0       |
| 02 | 1B 09 |       | LD G2,#9       |
| 04 | C0 03 | LOOP  | IN G0,SIOS     |
| 06 | 63 80 |       | AND G0,#80H    |
| 08 | A4 04 |       | JZ LOOP        |
| 0A | 11 17 |       | LD G0,DATA,G1  |
| 0C | C3 02 |       | OUT G0,SIOD    |
| 0E | 37 01 |       | ADD G1,#1      |
| 10 | 4B 01 |       | SUB G2,#1      |
| 12 | A4 16 |       | JZ END         |
| 14 | A0 04 |       | JMP LOOP       |
| 16 | FF    | END   | HALT           |
| 17 |       |       |                |
| 17 | 53 48 | DATA  | DC "SHIGEMURA" |
| 19 | 49 47 |       |                |
| 1B | 45 4D |       |                |
| 1D | 55 52 |       |                |
| 1F | 41    |       |                |

**例題 5-8** 文字列出力 2

**問題：** SIO ヘローマ字で “Tokuyama Kousen”, “Shigemura” のような 2 行を出力するプログラムを作りなさい。

**考え方：** 繰り返しにより、1 文字ずつ SIO へ出力します。1 行目データと 2 行目データの間に、復帰コードと改行コードを置きます。これらのコードも通常の文字と同じように扱って構いませんので、プログラムは 2 行分を一気に出力するもので構いません。

**プログラム：** プログラムにすると次のようになります。データの最後の 00H は、データの終わりを示す目印です。文字コードが 00H の文字は通常使用しませんので、これを文字ではなく終わりの印に使用しました。

この方法だと、出力するデータの長さが変化してもプログラムを変更する必要がありません。

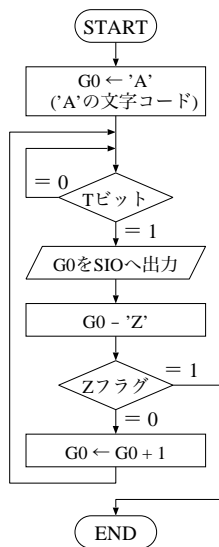
| 番地 | 機械語   | ラベル  | ニーモニック         |
|----|-------|------|----------------|
| 02 |       | SIOD | EQU 02H        |
| 03 |       | SIOS | EQU 03H        |
| 00 | 17 00 |      | LD G1,#0       |
| 02 | 19 15 | L0   | LD G2,DATA,G1  |
| 04 | 5B 00 |      | CMP G2,#0      |
| 06 | A4 14 |      | JZ END         |
| 08 | C0 03 | L1   | IN G0,SIOS     |
| 0A | 63 80 |      | AND G0,#80H    |
| 0C | A4 08 |      | JZ L1          |
| 0E | CB 02 |      | OUT G2,SIOD    |
| 10 | 37 01 |      | ADD G1,#1      |
| 12 | A0 02 |      | JMP L0         |
| 14 | FF    | END  | HALT           |
| 15 |       |      |                |
| 15 | 54 6F | DATA | DC "Tokuyama   |
| 17 | 6B 75 |      | Kousen"        |
| 19 | 79 61 |      |                |
| 1B | 6D 61 |      |                |
| 1D | 20 4B |      |                |
| 1F | 6F 75 |      |                |
| 21 | 73 65 |      |                |
| 23 | 6E    |      |                |
| 24 | 0D 0A |      | DC 0DH,0AH     |
| 26 | 53 68 |      | DC "Shigemura" |
| 28 | 69 67 |      |                |
| 2A | 65 6D |      |                |
| 2C | 75 72 |      |                |
| 2E | 61    |      |                |
| 2F | 0D 0A |      | DC 0DH,0AH     |
| 31 | 00    |      | DC 00H         |

**例題 5-9** ‘A’～‘Z’の文字を表示

**問題：** SIO へ ‘A’～‘Z’の文字を連続して出力するプログラムを作りたい。

**考え方：** ‘A’～‘Z’の文字コードは連続しています (「図 2.14 ASCII 文字コード表」参照)。最初に ‘A’の文字コードをレジスタにロードし SIO へ出力します。次に、レジスタに 1 加えて文字コードを増やします。増やした結果は ‘B’の文字コードですので、それを出力します。後は同様に ‘Z’の文字コードまで出力します。

**フローチャート：** 次の通りです。



**プログラム：** G0 レジスタを文字コードの格納に、G1 レジスタを「T ビット」のチェックに使います。出力可能になったら G0 の文字コードを SIO へ出力します。出力した後で出力した文字を調べ、‘Z’ だったら終了します。

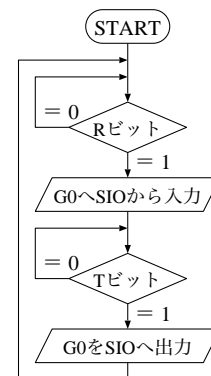
| 番地 | 機械語   | ラベル   | ニーモニック       |
|----|-------|-------|--------------|
| 02 |       | SIOD  | EQU 02H      |
| 03 |       | SIOS  | EQU 03H      |
| 00 | 13 41 |       | LD G0, #'A'  |
| 02 | C4 03 | WAIT  | IN G1, SIOS  |
| 04 | 67 80 |       | AND G1, #80H |
| 06 | A4 02 |       | JZ WAIT      |
| 08 | C3 02 |       | OUT G0, SIOD |
| 0A | 53 5A |       | CMP G0, #'Z' |
| 0C | A4 12 |       | JZ OWARI     |
| 0E | 33 01 |       | ADD G0, #1   |
| 10 | A0 02 |       | JMP WAIT     |
| 12 | FF    | OWARI | HALT         |

**例題 5-10** echo プログラム

**問題：** SIO から入力した文字をそのまま SIO へ送り返す (こだま) プログラムを作りたい。

**考え方：** STOP ボタンが押されるまで終了しないプログラムを作ります。

**フローチャート：** 次のようなアルゴリズムでできるはずです。



**プログラム：** G0 レジスタを文字コードの格納用に、G1 レジスタを「R ビット」、「T ビット」のチェック用に使います。IN 命令で G0 へ読み込んだ文字コードを、OUT 命令で書き戻すことにより文字を送り返します。

| 番地 | 機械語   | ラベル   | ニーモニック       |
|----|-------|-------|--------------|
| 02 |       | SIOD  | EQU 02H      |
| 03 |       | SIOS  | EQU 03H      |
| 00 |       |       |              |
| 00 | C4 03 | START | IN G1, SIOS  |
| 02 | 67 40 |       | AND G1, #40H |
| 04 | A4 00 |       | JZ START     |
| 06 | C0 02 |       | IN G0, SIOD  |
| 08 | C4 03 | WAIT  | IN G1, SIOS  |
| 0A | 67 80 |       | AND G1, #80H |
| 0C | A4 08 |       | JZ WAIT      |
| 0E | C3 02 |       | OUT G0, SIOD |
| 10 | A0 00 |       | JMP START    |

**問題**

1. SIO から 1 文字入力する度に ‘\*’ を送り返すプログラムを作りたい。(パスワード入力を真似する.)
2. SIO から 1 文字入力する度に、入力した文字がアルファベット小文字の場合大文字に変換し、そうでなければそのまま SIO へ出力するプログラムを作りたい。

### 5.15 まとめ

この章では教育用コンピュータ TeC の機械語プログラミングを学びました。手作業で機械語を作成すること（ハンドアセンブル）を前提にしました。これらを通してノイマン型コンピュータの特徴である以下の三つを体感できたはずです。

1. データだけでなくプログラムも主記憶に置くこと（ストアードプログラム方式）
2. プログラムの命令は PC で指定されたアドレス順に一つずつ実行されること（逐次実行方式）
3. データもプログラムも 2 進数で表現すること（2 進法）

そして何より、1 ビットでも間違えると TeC が正しく動作しないことを痛感したと思います。これは TeC に限らず現代のノイマン型コンピュータ全てに共通です。また、C 言語や Java 言語などの高級言語を用いた場合であっても、1 文字の間違いも許容されないことは共通です。ノイマン型コンピュータを使用する以上、ハードウェアもソフトウェアも 1 ビットも間違ってはならないことをよく覚えておいて下さい。

ハンドアセンブルによりプログラムを作成するのは効率が悪いので、アセンブラというプログラムで機械語を自動的に作ることもできます。TeC 用のアセンブラは <https://github.com/tctsigemura/Tasm> から入手可能です。

## 第 6 章

# 高度なプログラミング

### 6.1 クロス開発

第 5 章では、ハンドアセンブルにより機械語のプログラムを作成しました。プログラムをニーモニックから機械語に手作業で変換する際、ミスが多発しました。このようにミスが多発するので、実は、ハンドアセンブルによるプログラムの開発は、ほとんど行われません。

この授業の中では、原理をよく理解してもらうために、あえて面倒なハンドアセンブルをたくさんしてもらいました。そろそろ、機械語の原理はよく分かってきたので、楽にプログラミングする方法を教えます。

プログラムを作成することを「プログラムを開発する」と言います。楽に「開発する」ために使用するコンピュータやプログラムのことを、「開発環境」と言います。「開発環境」には、「セルフ開発環境」と「クロス開発環境」があります。

「セルフ開発環境」は、パソコンで Java 言語や C 言語でプログラムを作成し実行するような、プログラムを開発するコンピュータと実行するコンピュータが同じ場合に用いる「開発環境」です。

「クロス開発環境」は、TeC のような小さなコンピュータのプログラム開発によく用いられます。他のコンピュータ上でプログラムを作成し、出来上がったプログラムを目的のコンピュータに転送して実行するような「開発環境」です。「クロス開発」環境を用いたプログラムの開発は、「クロス開発」と呼ばれます。

#### 6.1.1 TeC クロス開発環境

「TeC クロス開発環境」は、**アセンブラとダウンロードプログラム**からなります。「TeC クロス開発環境」の入手とパソコンへのインストール方法は、<https://github.com/tctsigemura/Tasm> を参照してください。以下では、UNIX または macOS に「TeC クロス開

発環境」がインストールされていることを前提に説明します。また、「TeC クロス開発環境」の詳細は、「付録 A TeC クロス開発環境」を参照してください。

#### 6.1.2 アセンブラ

ニーモニックから機械語に変換する作業を、これまでは、手作業（ハンドアセンブル）により行ってきました。このような単純で機械的な作業はコンピュータが得意なものです。この作業を行うプログラムを「アセンブラ」と言います。アセンブラは、ニーモニックで記述したプログラムを機械語に変換します。

#### TeC 用アセンブラの使用方法

TeC 用のアセンブラは、UNIX や macOS、または Windows 上で動作する「クロスアセンブラ」です。クロスアセンブラを使用したプログラムの作成手順は次の通りです。

1. テキストエディタを用いてプログラムを入力します。プログラムを格納するファイルの拡張子は“.t7” でなければなりません。例えば、以下のニーモニックを“xxx.t7”ファイルに書込みます。

```

1 ; 入力例
2     ld g0,a
3     st g0,b
4     halt
5 a   dc 10
6 b   ds 1

```

1 行目が、注釈の例です。‘;’ から行末までが注釈になります。ラベルのない行（2, 3, 4 行）は、一つ以上の空白を書いた後、命令を書きます。ラベルのある行（5, 6）は、行の 1 文字目からラベルを書きます。なお、例では命令のインデントが揃っていますが、揃える必要はありません。

2. アセンブラを実行します。アセンブラのコマンド名は、`tasm7` です。実行例は、次の通りです。

```
$ tasm7 xxx.t7
アセンブル成功
結果は [xxx.lst] と [xxx.bin] に格納しました。
```

```
[xxx.lst]
ADR CODE      Label  Instruction

00           ; 入力例
00 10 05             LD      GO,A
02 20 06             ST      GO,B
04 FF              HALT
05 0A      A        DC      10
06 00      B        DS      1
```

入力にミスがなければ、アセンブルリストと機械語プログラムが生成されます。

入力にミスがあった場合は、例えば次のようなエラーメッセージが表示され、エラーの場所と原因が分かるようになっています。

```
$ tasm7 yyy.t7
*** エラー [未知のニーモニック] ***
エラー発生場所：ファイル [yyy.t7] の 3 行で
エラー行の内容：[ SL GO,B]
エラートークン：[SL]
```

## TeC 用アセンブラの出力

### 1. アセンブルリスト

画面と拡張子 “.lst” のファイルに、アセンブルリストが出力されます。アセンブルリストは、入力したニーモニックと、それから作られた機械語の対応を分かり易く表示したものです。ハンドアセンブルで行っていた機械語作成の結果と同じになっていることを確認してください。

### 2. 機械語プログラムファイル

拡張子 “.bin” のファイルに機械語が格納されます。機械語プログラムファイルの内容は次図の通りです。

|                 |
|-----------------|
| ロードアドレス (1 バイト) |
| プログラム長 (1 バイト)  |
| ...             |
| 機械語 (1 バイト以上)   |
| ...             |

ファイルの 1 バイト目は、機械語プログラムの TeC 主記憶上の配置アドレスです。機械語は、このバイトで表される番地を先頭にして主記憶に書き込まれます。ロードアドレスは、アセンブラの `ORG` 命令で指示することができます。ORG 命令を使用しない場合のロードアドレスは、0 番地になります。ファイルの 2 バイト目は、機械語プログラムの長さです。3 バイト目以降の機械語の長さをバイト単位で表します。

### 6.1.3 ダウンロード

アセンブラを実行することにより、拡張子 “.bin” のファイルに機械語が格納されました。次に、この機械語を TeC の主記憶にダウンロードします。

転送にはパソコンと TeC をケーブルで接続し通信できる状態にした上で、パソコンで `tsend7` コマンド、TeC で IPL (Initial Program Loader : 付録 A.8 参照) プログラムを実行します。`tsend7` コマンドと IPL プログラムが通信してパソコンから TeC にプログラムが転送されます。

ケーブルの接続方法は「5.14 TeC シリアル入出力」を参照してください。ダウンロード手順は次の通りです。

1. パソコンで `tsend7` プログラムを起動します。下の実行例のように機械語が格納されたファイル (xxx.bin) を指定します。TeC を受信状態にするように指示が表示されます。
2. TeC で IPL プログラムを起動します。IPL はメモリの E0H 番地から FFH 番地の ROM 領域に予め格納されています (「図 B.1 TeC7 命令表」のメモリマップ参照)。PC に E0H をセットし RUN ボタンを押すと IPL が起動します。
3. パソコン側で Enter を入力します。

```
$ tsend7 xxx.bin
TeC7 を受信状態にして Enter キーを押して下さい。
[00] [07] [10] [05] [20] [06] [ff] [0a] [00]
$
```

以上の操作で、コンソールパネルから機械語を打ち込むこと無く、機械語が TeC の主記憶の 00H 番地から始まる領域に書き込まれました。主記憶に書き込まれたプログラムを実行する方法は、コンソールパネルから機械語を打ち込んだ場合と同様です。

### ニーモニック入力上の注意

- TeC のアセンブラは、文字データ以外では、アルファベット大文字と小文字を区別しません。小文字で記述しても、アセンブリリストには大文字に変換されて表示されます。
- ‘A’～‘F’ の文字で始まる 16 進数は、前に ‘0’ を付加する必要があります。アルファベットで始まる 16 進数は、ラベルと区別ができないためです。(例 F1H → 0F1H)
- 空白の表現には、スペースとタブのどちらも使用できます。インデントを揃えるにはタブを使用すると便利です。また、空白の文字数はいくつでも構いません。読みやすいように、上手に空白を使用してください。
- ‘;’ をニーモニックの中に入書くと、次の文字から行末までが注釈（コメント）になります。上手に注釈を入れて、読みやすいプログラムにしてください。
- 文字や文字列の表記も、これまで使用してきた‘文字’や“文字列”の記述が使用できます。実際に使用する記号は、‘’（パソコンの日本語キーボードでは、シフトを押しながら「7」）と、“”（シフトを押しながら「2」）です。本書の記述の中では、左と右のクォートに異なる記号を用いている場合もありますが、実際には同じ記号です。

### 問題

1. 「例題 5-5 ビットの回転」をクロス開発環境を用いてやり直さない。
2. 「例題 5-3 割算を計算する」をクロス開発環境を用いてやり直さない。
3. 「例題 5-8 文字列出力 2」をクロス開発環境を用いてやり直さない。

### 注意

同じシリアル通信機能を「クロス開発環境」と「文字列出力プログラム」の両方で使用します。プログラムのダウンロード時には、パソコンのターミナルプログラムを終了する必要があります。

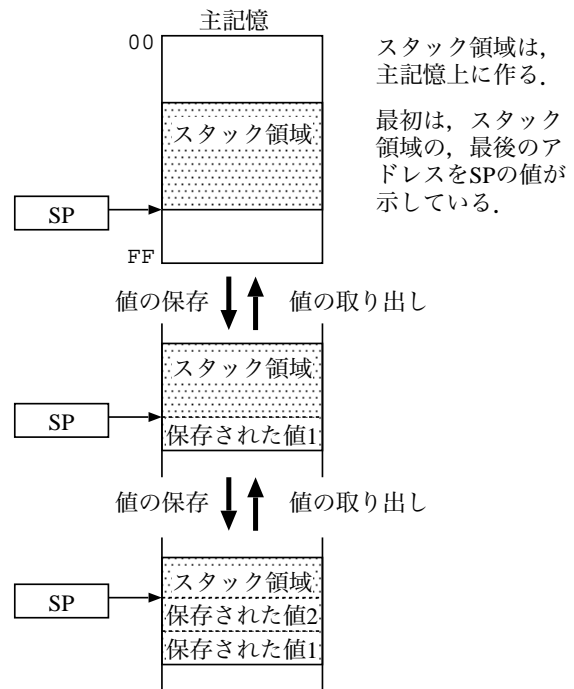


図 6.1 スタックの仕組み

## 6.2 スタック

スタックは、一時的にデータを保管するために使用できます。例えば、レジスタの個数が足りなくなったときレジスタの値をスタックに一時退避しておき、後でレジスタに復元することができます。

同じことを、ST, LD 命令と「名前を付けた領域」を用いて行うことも可能ですが、スタックを利用すると「名前を付けた領域」を使用する必要がありません。スタックもメモリ上の領域ですが、名前を付けたりすることなしに、プログラムのあちこちから同じ領域を繰り返し使用することができます。

### 6.2.1 仕組み

スタックは、CPU のスタックポインタ (SP レジスタ) により管理されます。図 6.1 のように、最初は SP がスタック領域の最後の番地を記憶しています。スタックに値を保存するときは、SP の値から 1 を引き新しいアドレスを決め、そのアドレスに値を保存します。値を保存する度に SP の値は減少していきます。

スタックから値を取り出すときは、SP の示すアドレスの値を取り出し、次に SP の値に 1 を加えます。値を取り出す度に SP の値は増加していきます。

このように、スタックを用いると、保存したのとは逆の順番で値を取り出すようになります。最後に入れたデータを最初に取り出すので、スタックは LIFO (Last In First Out) 方式のデータ構造です。

### 6.2.2 PUSH 命令

Push は、「押す」という意味の英語です。データをスタックに「押し込む」のに使用します。

**意味：** レジスタの値をスタックに押し込む。

**ニーモニック：** PUSH GR

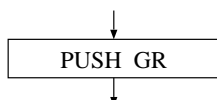
GR で指定されたレジスタがスタックに押し込まれます。GR が SP を表す場合は、SP の値がスタックに押し込まれます。

**命令フォーマット：** PUSH 命令は、1 バイト長の命令です。XR フィールドは必ず 00<sub>2</sub> にします。

| 第1バイト             |                    |
|-------------------|--------------------|
| OP                | GR XR              |
| 1101 <sub>2</sub> | GR 00 <sub>2</sub> |

**動作の詳細：** PUSH 命令は、まず、スタックポインタ (SP) の値を 1 減らします。次に、SP の値をアドレスとみなし、主記憶の SP 番地に指定されたレジスタの値を書き込みます。

**フローチャート：** 次のように描くことにします。



### 6.2.3 POP 命令

Pop は、「飛び出る」という意味の英語です。データをスタックから取り出すのに使用します。

**意味：** スタックのデータをレジスタに取り出す。

**ニーモニック：** POP GR

GR で指定されたレジスタにスタックからデータが取り出されます。GR が SP を表す場合は、スタックからデータが取り出され、SP に格納されます。

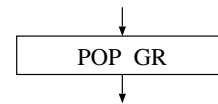
**命令フォーマット：** POP 命令は、1 バイト長の命令です。XR フィールドは必ず 10<sub>2</sub> にします。

| 第1バイト             |                    |
|-------------------|--------------------|
| OP                | GR XR              |
| 1101 <sub>2</sub> | GR 10 <sub>2</sub> |

**動作の詳細：** POP 命令は、まず、主記憶の SP 番地からデータを取り出し、指定されたレジスタに格納し

ます。次に、SP の値を 1 増やします。

**フローチャート：** 次のように描くことにします。



### PUSH/POP 命令の応用例 (1)

PUSH 命令と POP 命令を組み合わせると、レジスタの値を一時的に保管することができます。まず、SP の値を初期化します。DCH 番地に初期化する理由は、「図 B.1 TeC7 命令表」のメモリマップを参照して考えてください。スタックは LIFO なので、**PUSH (保存) した順序と逆の順序で POP (復元) する必要があります。**

| ラベル | ニーモニック           |
|-----|------------------|
|     | LD SP, #ODCH     |
|     | ...              |
|     | G0, G1 を使用する処理 1 |
|     | ...              |
|     | PUSH G0 ; G0 を保存 |
|     | PUSH G1 ; G1 を保存 |
|     | ...              |
|     | G0, G1 を使用する処理 2 |
|     | ...              |
|     | POP G1 ; G1 を復元  |
|     | POP G0 ; G0 を復元  |
|     | ...              |
|     | 処理 1 の続き         |
|     | ...              |

### PUSH/POP 命令の応用例 (2)

PUSH 命令と POP 命令を組み合わせると、レジスタの値を別のレジスタにコピーすることができます。TeC にはレジスタ間のデータ転送命令がないので、このテクニックは役に立ちます。次の例では G0 の値を G1 にコピーします。

| ラベル | ニーモニック           |
|-----|------------------|
|     | LD SP, #ODCH     |
|     | ...              |
|     | PUSH G0 ; G0 を保存 |
|     | POP G1 ; G1 に復元  |
|     | ...              |

### 問題

スタックを利用して G0 と G1 の値を交換する方法を考えなさい。



## 6.3 サブルーチン

プログラムの中に同じ処理が何度も出てくることがあります。また、一つのまとまった機能を実現している部分があり、他の部分と独立させた方がプログラムの見通しが良くなる場合があります。例えば、「5.14 TeC シリアル入出力 (SIO)」の例題プログラムの多くで、1 文字入力や 1 文字出力を行うプログラムの部分がありました。このような、一つのまとまった機能を実現している部分がそうです。

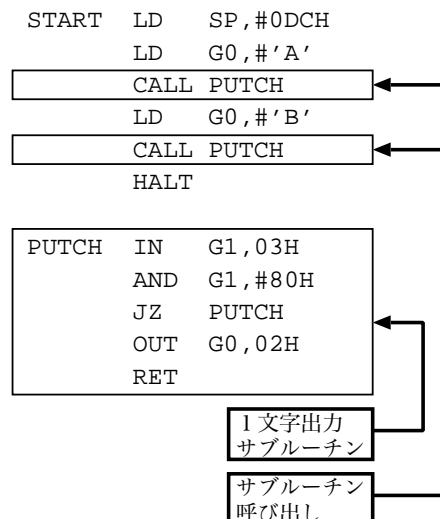
このような部分はサブルーチン（副プログラム）としてまとめることにより、繰り返し呼び出して使用したり、他の部分から独立させて見やすいプログラムにすることができます。ここでは、サブルーチンを作る方法を学びます。

### 6.3.1 仕組み

サブルーチンは、他のプログラムから呼び出されて実行されます。サブルーチンの実行が終わったら、呼び出したプログラムに戻り続きを実行します。

サブルーチンを呼び出すための CALL 機械語命令と、サブルーチンの最後から呼び出し元に戻るための RET 機械語命令を新たに導入します。

次のプログラムは、1 文字出力機能をサブルーチンとして独立させ、メインルーチン（主プログラム）から 2 回呼び出します。プログラムは先頭から実行が開始され CALL 命令で PUTCH サブルーチンへジャンプします。PUTCH サブルーチンを最後まで実行すると RET 命令でメインルーチンに戻り、CALL 命令の次の命令から実行を再開します。



### 6.3.2 CALL 命令

Call は、「呼び出す」という意味の英語です。サブルーチンを呼び出すために使用します。CALL 命令は JMP 命令に似ていますが、サブルーチンから戻る準備をしてからジャンプしなければなりません。CALL 命令は、自身の次の命令のアドレスをスタックに保存 (PUSH) してから、サブルーチンにジャンプします。

**意味：** サブルーチンを呼び出す。

**フラグ：** 変化しません。

**ニーモニック：** CALL A[,XR]

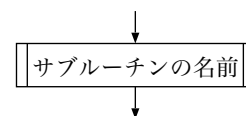
A 番地のサブルーチンを呼び出します。[,XR] は、インデックスドモードでジャンプアドレスを計算する場合に使用します。

**命令フォーマット：** CALL 命令は 2 バイトの長さを持ちます。各フィールドの意味は JMP 命令と同様です。GR フィールドは必ず 00<sub>2</sub> にします。

| 第1バイト             |                    | 第2バイト     |
|-------------------|--------------------|-----------|
| OP                | GR XR              |           |
| 1011 <sub>2</sub> | 00 <sub>2</sub> XR | aaaa aaaa |

**動作の詳細：** CALL 命令は、まず、PC の値 (CALL 命令実行時には既に次の命令のアドレスになっている) を、スタックに保存 (PUSH) します。次に、サブルーチンにジャンプします。

**フローチャート：** 次のように描きます。箱の中には、呼び出すサブルーチンの名前を書きます。



**使用上の注意：** CALL 命令はスタックを使用するので、SP を初期化してから使用しなければなりません。

### 6.3.3 RET (Return) 命令

Return は、「戻る」という意味の英語です。RET は Return の綴を縮めたものです。サブルーチンから戻るために使用します。

**意味：** サブルーチンから戻る。

**フラグ：** 変化しません。

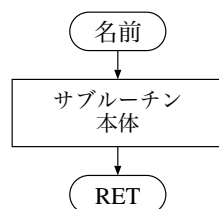
**ニーモニック：** RET

**命令フォーマット：** RET 命令は、1 バイト長の命令です。

| 第1バイト             |                                 |
|-------------------|---------------------------------|
| OP                | GR XR                           |
| 1110 <sub>2</sub> | 11 <sub>2</sub> 00 <sub>2</sub> |

**動作の詳細：** スタックから値を一つ取り出し PC にセットします。スタックには CALL 命令が保存した PC の値が保存されているので、プログラムの実行は、サブルーチンを呼び出した CALL 命令の次の命令に移ります。

**フローチャート：** 次のフローチャートの、最後の角を取った四角が RET 命令に対応します。フローチャートは、サブルーチン全体を示しています。



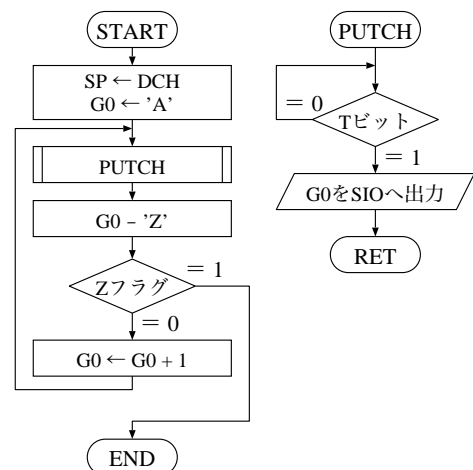
**使用上の注意：** RET 命令は POP 命令と同様な方法でスタックから値を取り出します。サブルーチンの中で PUSH 命令や POP 命令を使用した場合は、スタックの状態がサブルーチンの実行開始時と同じになるようにしてから、RET 命令を実行しなければなりません。サブルーチンの中で、PUSH 命令の実行回数と POP 命令の実行回数が同じになるようにして下さい。

#### 例題 6-1 ‘A’～‘Z’の文字を表示 (改良版 1)

**問題：** SIO へ ‘A’～‘Z’の文字を連続して出力するプログラムを、前ページの 1 文字出力サブルーチン (PUTCH) を使用して作りなさい。

**考え方：** 基本的な考え方は「例題 5-9 ‘A’～‘Z’の文字を表示」と同じです。1 文字出力サブルーチンは G0 レジスタにセットされた値を SIO へ出力します。メインルーチンは、G0 レジスタに ‘A’～‘Z’の文字の文字コードをセットして、サブルーチンを呼び出します。

**フローチャート：** サブルーチンの呼び出しとサブルーチンは次のように描きます。



**プログラム：** CALL, RET 命令がスタックを使用するので、まず、SP の初期化をします。G0 レジスタを文字コードの格納に、G1 レジスタを「T ビット」のチェックに用います。

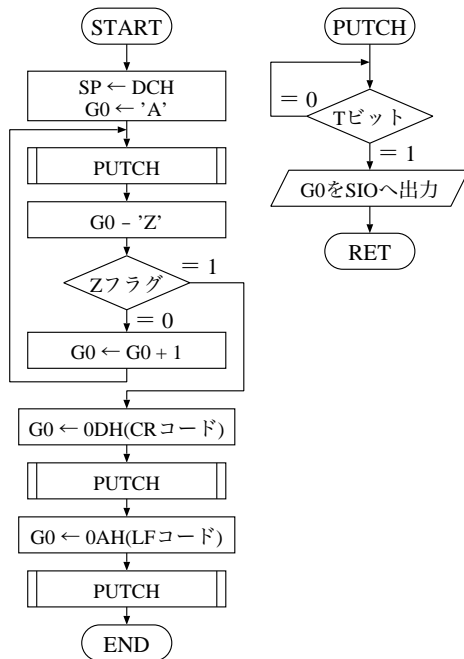
| 番地 | 機械語   | ラベル   | ニーモニック       |
|----|-------|-------|--------------|
| 02 |       | SIOD  | EQU 02H      |
| 03 |       | SIOS  | EQU 03H      |
| 00 | 1F DC | START | LD SP, #ODCH |
| 02 | 13 41 |       | LD G0, #'A'  |
| 04 | B0 0F | LOOP  | CALL PUTCH   |
| 06 | 53 5A |       | CMP G0, #'Z' |
| 08 | A4 0E |       | JZ OWARI     |
| 0A | 33 01 |       | ADD G0, #1   |
| 0C | A0 04 |       | JMP LOOP     |
| 0E | FF    | OWARI | HALT         |
| 0F | C4 03 | PUTCH | IN G1, SIOS  |
| 11 | 67 80 |       | AND G1, #80H |
| 13 | A4 0F |       | JZ PUTCH     |
| 15 | C3 02 |       | OUT G0, SIOD |
| 17 | EC    |       | RET          |

**例題 6-2** ‘A’～‘Z’の文字を表示（改良版 2）

**問題：** SIO へ ‘A’～‘Z’の文字を連続して出力した後、SIO へ改行コードを出力するプログラムを作りなさい。

**考え方：** サブルーチン（PUTCH）をメインルーチンの複数の箇所から呼び出します。

**フローチャート：** 改行するために、CR、LF を出力します。



**プログラム：** PUTCH ルーチンは、「例題 6-1 ‘A’～‘Z’の文字を表示（改良版 1）」と同様なので省略しました。

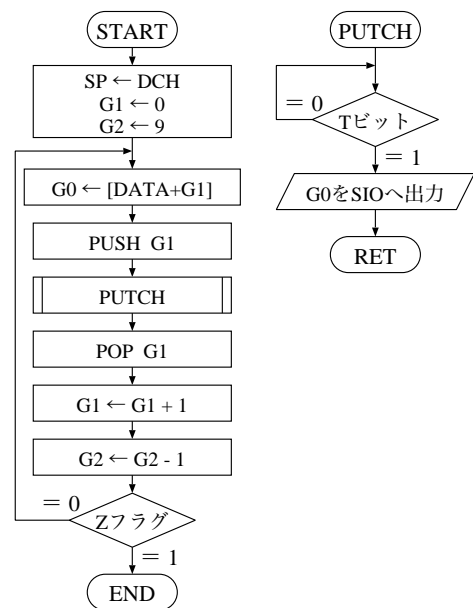
| 番地 | 機械語   | ラベル   | ニーモニック       |
|----|-------|-------|--------------|
| 02 |       | SIOD  | EQU 02H      |
| 03 |       | SIOS  | EQU 03H      |
| 00 | 1F DC | START | LD SP, #0DCH |
| 02 | 13 41 |       | LD G0, #'A'  |
| 04 | B0 17 | LOOP  | CALL PUTCH   |
| 06 | 53 5A |       | CMP G0, #'Z' |
| 08 | A4 0E |       | JZ OWARI     |
| 0A | 33 01 |       | ADD G0, #1   |
| 0C | A0 04 |       | JMP LOOP     |
| 0E | 13 0D | OWARI | LD G0, #13   |
| 10 | B0 17 |       | CALL PUTCH   |
| 12 | 13 0A |       | LD G0, #10   |
| 14 | B0 17 |       | CALL PUTCH   |
| 16 | FF    |       | HALT         |
| 17 | C4 03 | PUTCH | ...          |

**例題 6-3** 文字列出力（改良版 1）

**問題：** SIO へローマ字で自分の名前を出力するプログラムを作りなさい。

**考え方：** 「例題 5-7 文字列出力 1」をサブルーチンを使用して作りなおします。

**フローチャート：** G1 レジスタが PUTCH サブルーチンで破壊される（Tビットのチェックで使います。）ので、メインルーチン側で G1 レジスタをスタックに保存しています。



**プログラム：** PUTCH ルーチンは省略します。

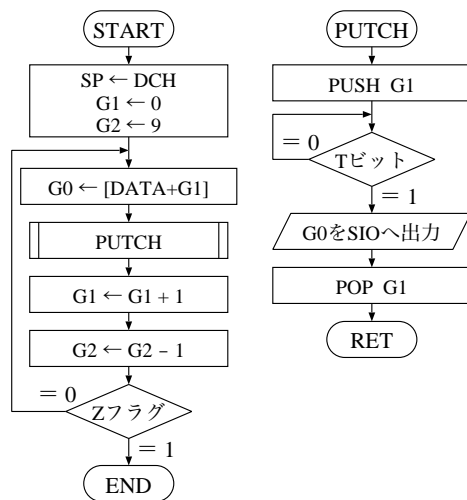
| 番地 | 機械語   | ラベル   | ニーモニック          |
|----|-------|-------|-----------------|
| 02 |       | SIOD  | EQU 02H         |
| 03 |       | SIOS  | EQU 03H         |
| 00 | 1F DC | START | LD SP, #0DCH    |
| 02 | 17 00 |       | LD G1, #0       |
| 04 | 1B 09 |       | LD G2, #9       |
| 06 | 11 1E | LOOP  | LD G0, DATA, G1 |
| 08 | D4    |       | PUSH G1         |
| 09 | B0 15 |       | CALL PUTCH      |
| 0B | D6    |       | POP G1          |
| 0C | 37 01 |       | ADD G1, #1      |
| 0E | 4B 01 |       | SUB G2, #1      |
| 10 | A4 14 |       | JZ END          |
| 12 | A0 06 |       | JMP LOOP        |
| 14 | FF    | END   | HALT            |
| 15 |       |       |                 |
| 15 | C4 03 | PUTCH | ...             |
| 1E |       |       |                 |
| 1E | 53 48 | DATA  | DC "SHIGEMURA"  |
| 20 | ...   |       |                 |

**例題 6-4** 文字列出力 (改良版 2)

**問題：** SIO ヘローマ字で自分の名前を出力するプログラムを作りなさい。

**考え方：** 「例題 5-7 文字列出力 1」をサブルーチンを使用して作りなおします。

**フローチャート：** PUTCH サブルーチンが G1 レジスタを破壊しないようにしました。「例題 6-3 文字列出力 (改良版 1)」より完成度の高い PUTCH サブルーチンです。



**プログラム：** PUTCH ルーチンも掲載します。

| 番地 | 機械語      | ラベル   | ニーモニック          |
|----|----------|-------|-----------------|
| 02 |          | SIOD  | EQU 02H         |
| 03 |          | SIOS  | EQU 03H         |
| 00 | 1F DC    | START | LD SP, #0DCH    |
| 02 | 17 00    |       | LD G1, #0       |
| 04 | 1B 09    |       | LD G2, #9       |
| 06 | 11 1E    | LOOP  | LD G0, DATA, G1 |
| 08 | B0 13    |       | CALL PUTCH      |
| 0A | 37 01    |       | ADD G1, #1      |
| 0C | 4B 01    |       | SUB G2, #1      |
| 0E | A4 12    |       | JZ END          |
| 10 | A0 06    |       | JMP LOOP        |
| 12 | FF       | END   | HALT            |
| 13 | D4       | PUTCH | PUSH G1         |
| 14 | C4 03    | WAIT  | IN G1, SIOS     |
| 16 | 67 80    |       | AND G1, #80H    |
| 18 | A4 14    |       | JZ WAIT         |
| 1A | C3 02    |       | OUT G0, SIOD    |
| 1C | D6       |       | POP G1          |
| 1D | EC       |       | RET             |
| 1E | 53 48 49 | DATA  | DC "SHIGEMURA"  |
| 21 | 47 45 4D |       |                 |
| 24 | 55 52 41 |       |                 |

**問題**

1. SIO から 1 文字を入力するサブルーチン (GETCH) を作成しなさい。GETCH は、入力した文字の文字コードを G0 に格納して戻るものとします。
2. 前出の 1 文字出力サブルーチン (PUTCH) と GETCH を利用し、「例題 5-10 echo プログラム」を作りなおしなさい。
3. 「例題 5-3 割算を計算する」を参考に、割算サブルーチン DIV を作成しなさい。DIV は、G0 に割られる数、G2 に割る数をセットして呼び出され、G1 に商、G0 に余りを格納して戻ります。
4. 「例題 5-6 シフトを用いた高速乗算」を参考に、G0 の数値の 10 倍を計算し G0 に求めるサブルーチン MUL10 を作成しなさい。

| ラベル   | ニーモニック    | ステート |
|-------|-----------|------|
| MS1   | PUSH GO   | 6    |
|       | LD GO,#n  | 5    |
| MATU  | SUB GO,#1 | 5    |
|       | JZ KAERU  | 4/5  |
|       | JMP MATU  | 5    |
| KAERU | POP GO    | 6    |
|       | RET       | 6    |

図 6.2 1ms タイマーサブルーチン

## 6.4 時間の計測

コンピュータで、正確な時間を計る必要がある場合があります。時間を計測するための専用の回路を使用できる場合もありますが、TeC のような小規模マイコンは専用の回路を持っていないことがあります。そこで、プログラムだけで時間を計測する工夫が必要になります。

### 6.4.1 マシンステート

プログラムだけで一定の時間が経過するのを待つには、各命令の実行に必要な時間を調べて、ちょうど目的の時間が必要な命令の組合せを実行させます。

命令表にステート数と書かれた欄があります。この数値が命令の実行に要する時間を表します。例えば、NO 命令はステート数 3 の命令ですので、**3 マシンステート**の時間が必要になります。

TeC の場合、**1 マシンステート**は、回路の動く基準になるクロック信号の 1 周期分になります。TeC のクロック信号は、2.4576MHz です。1 マシンステートは、 $1/(2.4576 \times 10^6) = 0.4069 \times 10^{-6} = 0.4069\mu s$  です。例えば、NO 命令はステート数が 3 なので、実行に  $0.4069\mu s \times 3 = 1.2207\mu s$  の時間がかかります。他の命令も、同様に実行時間が決まります。

### 6.4.2 1ms タイマ (マシンステートの応用)

マシンステートの合計を調節して、実行にちょうど 1ms の時間がかかるサブルーチンを作ります。クロックの周波数から、1 秒が  $2.4576 \times 10^6$  マシンステートになりますので、1ms は、 $(2.4576 \times 10^6) \times (1 \times 10^{-3}) = 2,457.6$  マシンステートになります。

実行に、2,458 マシンステートを要するサブルーチンを作ります。ステート数が多いので繰り返しを使用しないと、うまくいきそうにありません。繰り返しを作るには、図 6.2 のようなプログラムが必要です。

| ラベル   | ニーモニック         |
|-------|----------------|
| START | ; メインルーチン      |
|       | LD SP,#0DCH    |
|       | CALL MS200     |
|       | CALL MS200     |
|       | CALL MS200     |
|       | CALL MS200     |
|       | CALL MS200     |
|       | HALT           |
|       | ; 200ms サブルーチン |
|       | MS200 PUSH GO  |
| L1    | LD GO,#200     |
|       | CALL MS1       |
|       | SUB GO,#1      |
|       | JZ L2          |
| L2    | JMP L1         |
|       | POP GO         |
|       | RET            |
| MS1   | ; 1ms サブルーチン   |
|       | PUSH GO        |
|       | LD GO,#174     |
|       | MATU SUB GO,#1 |
|       | JZ KAERU       |
|       | JMP MATU       |
|       | KAERU POP GO   |
|       | RET            |

図 6.3 0.2 秒タイマーサブルーチン

このプログラムの実行に要するステート数は、繰り返し回数を  $n$  とすると、 $6 + 5 + (5 + 4 + 5) \times (n - 1) + 5 + 5 + 6 + 6$  となります。ステート数が 2,458 になるような  $n$  を計算すると、 $n$  は 174 になります (3 ステート不足)。図 6.2 のプログラムで  $n$  を 174 にしたものが、実行に約 1ms かかるサブルーチンになります。

### 6.4.3 0.2 秒タイマ (入れ子サブルーチンの利用)

1ms タイマサブルーチンを利用すると、もっと長い時間のタイマサブルーチンを簡単に作ることができます。0.2 秒タイマルーチン (図 6.3 の MS200 サブルーチン) は、1ms タイマサブルーチンを 200 回呼び出すように作ります。「サブルーチンの中から別のサブルーチンを呼び出すこと」を「サブルーチンの入れ子」と言います。ここでは「サブルーチンの入れ子」を利用しています。

図 6.3 のプログラム全体では、MS200 サブルーチンを 5 回呼び出して、実行に約 1 秒かかるプログラムの例になっています。

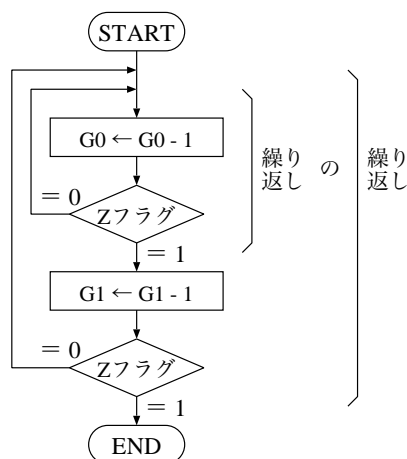


図 6.4 2重ループ<sup>o</sup>

| 番地 | 機械語   | ラベル   | ニーモニック |          |       |
|----|-------|-------|--------|----------|-------|
| 00 | 13 6C | START | LD     | G0, #108 | ; 5   |
| 02 | 17 CC | L1    | LD     | G1, #204 | ; 5   |
| 04 | 1B 07 | L2    | LD     | G2, #7   | ; 5   |
| 06 | 4B 01 | L3    | SUB    | G2, #1   | ; 5   |
| 08 | A4 0C |       | JZ     | L4       | ; 4/5 |
| 0A | A0 06 |       | JMP    | L3       | ; 5   |
| 0C | 47 01 | L4    | SUB    | G1, #1   | ; 5   |
| 0E | A4 12 |       | JZ     | L5       | ; 4/5 |
| 10 | A0 04 |       | JMP    | L2       | ; 5   |
| 12 | 43 01 | L5    | SUB    | G0, #1   | ; 5   |
| 14 | A4 18 |       | JZ     | L6       | ; 4/5 |
| 16 | A0 02 |       | JMP    | L1       | ; 5   |
| 18 | 00    | L6    | NO     |          | ; 3   |
| 19 | 00    |       | NO     |          | ; 3   |
| 1A | FF    |       | HALT   |          | ; 3   |

図 6.5 3重ループを用いた1秒タイマー

#### 6.4.4 1 秒タイマ（多重ループの利用）

約  $1ms$  のタイマを 200 回呼び出すサブルーチンを 5 回呼び出す方法では、誤差がたまって、正確な 1 秒タイマではなくなっている可能性があります。正確な 1 秒タイマを、1 から作りなおします。

1 秒タイマは、2,457,600 マシンステートのプログラムになります。1ms タイマでは、繰り返し 1 回が 14 ステートでした。同じステート数だとすると、 $2,447,600/14 = 174,828$  回 ループを繰り返す必要があります。カウンタに使用したレジスタは 8 ビットですから、256 回のカウントが限界です。そこで、図 6.4 に示すフローチャートのように繰り返しを更に繰り返す (2 重ループ) プログラムを作成する必要があります。

ところが、256 回を 256 回繰り返しても、まだ、繰り返し回数の 174,828 回に足りません。そこで、実際には「「繰り返し」の繰り返し」の繰り返し（3 重ループ）を作ります。図 6.5 にプログラムを示します。繰り返し回数は、プログラムを書いてプログラム各部のステート数が定まった後で、試行錯誤で決めました。

## 問題

1. 0.2 秒タイマサブルーチンを使用した，1 秒タイマプログラムを入力して実行しなさい。
2. 0.2 秒タイマサブルーチンを利用して，ブザーを 1 秒間隔で 0.2 秒間鳴らすプログラムを作りなさい。  
このプログラムは STOP を押すまで繰り返し実行するものとします。
3. 多重ループを利用した 1 秒タイマプログラムを入力して実行しなさい。
4. 2 重ループの練習として，SIO に 'A' 10 文字の行を，5 行出力するプログラムを作りなさい。
5. 入れ子サブルーチンの練習として，SIO に 'A' 10 文字の行を，5 行出力するプログラムを作りなさい。  
(1 行出力サブルーチンを 5 回呼び出す．1 行出力サブルーチンは，1 文字出力サブルーチンを 10 回呼び出す.)
6. 最初の 3 分間は 1 秒毎にブザーを鳴らし，3 分経過したらブザーを鳴らしっぱなしにするプログラム (3 分タイマ) を作りなさい。

| 番地 | 機械語   | ラベル   | ニーモニック      |
|----|-------|-------|-------------|
| 02 |       | SIOD  | EQU 2       |
| 03 |       | SIOS  | EQU 3       |
| 00 | 1F DC | START | LD SP,#0DCH |
| 02 | 14 29 |       | LD G1,DATA  |
| 04 | 1B 08 |       | LD G2,#8    |
| 06 | 13 31 | L1    | LD G0,#'1'  |
| 08 | 95    |       | SHLL G1     |
| 09 | A8 0D |       | JC L2       |
| 0B | 13 30 |       | LD G0,#'0'  |
| 0D | B0 1E | L2    | CALL PUTCH  |
| 0F | 4B 01 |       | SUB G2,#1   |
| 11 | A4 15 |       | JZ L3       |
| 13 | A0 06 |       | JMP L1      |
| 15 | 13 0D | L3    | LD G0,#0DH  |
| 17 | B0 1E |       | CALL PUTCH  |
| 19 | 13 0A |       | LD G0,#0AH  |
| 1B | B0 1E |       | CALL PUTCH  |
| 1D | FF    |       | HALT        |
| 1E | D4    | PUTCH | PUSH G1     |
| 1F | C4 03 | PL1   | IN G1,SIOS  |
| 21 | 67 80 |       | AND G1,#80H |
| 23 | A4 1F |       | JZ PL1      |
| 25 | C3 02 |       | OUT G0,SIOD |
| 27 | D6    |       | POP G1      |
| 28 | EC    |       | RET         |
| 29 |       |       |             |
| 29 | CA    | DATA  | DC OCAH     |

図 6.6 2進数を表示するプログラム

## 6.5 数値の入出力

これまでに SIO を使用した文字の入出力を勉強しました。ここでは数値の入出力を扱います。

### 6.5.1 2進数の出力

データを2進数として表現したものを人間に読める形式で SIO に出力します。ビットの0は文字の'0'に、ビットの1は文字の'1'に変換します。図 6.6 に1バイトのデータを表示するプログラムを示します。

このプログラムは、DATA 番地の内容を上位ビットから順に'0'、'1'の文字に変換し、SIOへ出力します。8ビット分の出力が終わると、CR (0DH)、LF (0AH)を出力して画面を改行します。文字の出力には前出の PUTCH サブルーチンを使用します。パソコンの画面に、DATA 番地のデータ CAH を2進数で表した、“11001010”が表示されます。

| 番地 | 機械語         | ラベル   | ニーモニック        |
|----|-------------|-------|---------------|
| 02 |             | SIOD  | EQU 2         |
| 03 |             | SIOS  | EQU 3         |
| 00 | 1F DC       | START | LD SP,#0DCH   |
| 02 |             |       | ; 上位桁の出力      |
| 02 | 14 38       |       | LD G1,DATA    |
| 04 | 97          |       | SHRL G1       |
| 05 | 97          |       | SHRL G1       |
| 06 | 97          |       | SHRL G1       |
| 07 | 97          |       | SHRL G1       |
| 08 | 11 28       |       | LD G0,HSTR,G1 |
| 0A | B0 1D       |       | CALL PUTCH    |
| 0C |             |       | ; 下位桁の出力      |
| 0C | 14 38       |       | LD G1,DATA    |
| 0E | 67 0F       |       | AND G1,#0FH   |
| 10 | 11 28       |       | LD G0,HSTR,G1 |
| 12 | B0 1D       |       | CALL PUTCH    |
| 14 |             |       | ; 改行の出力       |
| 14 | 13 0D       |       | LD G0,#0DH    |
| 16 | B0 1D       |       | CALL PUTCH    |
| 18 | 13 0A       |       | LD G0,#0AH    |
| 1A | B0 1D       |       | CALL PUTCH    |
| 1C | FF          |       | HALT          |
| 1D |             |       | ; 1文字出力       |
| 1D | D4          | PUTCH | PUSH G1       |
| 1E | C4 03       | WAIT  | IN G1,SIOS    |
| 20 | 67 80       |       | AND G1,#80H   |
| 22 | A4 1E       |       | JZ WAIT       |
| 24 | C3 02       |       | OUT G0,SIOD   |
| 26 | D6          |       | POP G1        |
| 27 | EC          |       | RET           |
| 28 |             |       | ; 16進数字       |
| 28 | 30 31 32 33 | HSTR  | DC "01234567" |
| 2C | 34 35 36 37 |       |               |
| 30 | 38 39 41 42 |       | DC "89ABCDEF" |
| 34 | 43 44 45 46 |       |               |
| 38 |             |       | ; 出力するデータ     |
| 38 | CA          | DATA  | DC OCAH       |

図 6.7 16進数を表示するプログラム

### 6.5.2 16進数の出力

DATA 番地のデータを16進数で SIO に出力するプログラムを図 6.7 に示します。

まず、データの上位4ビットを取り出し、その値から'0'~'F'の1文字を選択し SIOへ出力します。次に、下位4ビットを用いて同様の出力をします。最後に、CR (0DH)、LF (0AH)を出力し画面を改行します。パソコンの画面に“CA”が表示されます。



| ラベル    | ニーモニック      | コメント        |
|--------|-------------|-------------|
| PUTDEC | LD SP,#0DCH | ; SP 初期化    |
|        | LD GO,DATA  | ; 割られる数     |
|        | LD G2,#10   | ; 割る数       |
|        | CALL DIV    | ; 割算        |
|        | ST GO,D1    | ; 余り (1 桁)  |
|        | PUSH G1     | ; 商を        |
|        | POP GO      | ; 割られる数に    |
|        | CALL DIV    | ; 割算        |
|        | ST GO,D2    | ; 余り (10 桁) |
|        | ST G1,D3    | ; 商 (100 桁) |

図 6.8 各桁に分する部分

### 6.5.3 10 進数の出力

DATA 番地のデータを 10 進数で SIO に出力するプログラムを考えましょう。扱うデータは 8 ビットの符号無し 2 進数 (0 ~ 255) とします。10 進数で表示するには、10 進数の一桁一桁に分解した後、一桁毎に対応する数字に変換して出力します。

#### 各桁への分解

次のような手順で数値を一桁一桁に分解します。下にデータが 123 だった場合の計算例を示しますので、計算例と比較しながら手順を確認して下さい。

1. データを 10 で割って商と余りを求めます。この時、余りが最下位桁 (1 桁) の値です。
2. 商を更に 10 で割って新しい商と余りを求めます。新しい余りが下から 2 桁目 (10 桁) の値です。
3. 新しい商は下から 3 桁目 (100 桁) の値になります。

#### 計算例

```

10) 123
   10) 12...3   (余り 3 が 1 桁の値)
      1...2     (余り 2 が 10 桁の値)
           (商 1 が 100 桁の値)
```

以上の手順をプログラムにするためには、割算機能が必要です。ここでは、「例題 5-3 割算を計算する」をサブルーチン化して DIV という名前で準備済みと仮定します。図 6.8 は割算サブルーチン DIV を使用した桁への分解プログラム (部分) です。各桁の値は、D1, D2, D3 の 3 バイトに格納されます。後で、これらの値を上から順に出力します。

| ラベル | ニーモニック      | コメント        |
|-----|-------------|-------------|
|     | LD GO,D3    | ; 100 位桁の数値 |
|     | ADD GO,#'0' | ; 文字コードへ変換  |
|     | CALL PUTCH  | ; SIO へ出力   |

図 6.9 一桁目を数字に変換し出力する部分

#### 各桁の出力

桁への分解ができれば、分解した各桁を上から順に数字に変換して SIO へ出力します。数字の文字コードが '0' ~ '9' まで連続していることを利用すると、数値から数字への変換は簡単にできます。

$$\text{数字} = \text{数値} + \text{'0'の文字コード}$$

変換結果を PUTCH サブルーチンを利用して SIO へ出力します。図 6.9 は 100 位桁の値を数字に変換し出力するプログラム (部分) です。10 位桁、1 位桁も同様に出力すれば 3 桁の数値を表示できます。

### 6.5.4 10 進数の入力

10 進数の出力ができるようになったので、今度は入力を考えましょう。10 進数の出力と逆の手順で行います。まず、一桁毎に入力し数字を数値に変換します。次に、各桁の値を合成して一つの数値にします。

#### 各桁の入力

入力は符号無し 8bit 2 進数で表現できる 0 ~ 255 の数値とします。今回はプログラムの作りやすさを優先して、3 桁の数字しか入力されない前提で考えます。100 未満の数値を入力するときは、“012” のように上の 1 桁に '0' を入力します。10 未満の数値を入力するときは、“001” のように上の 2 桁に '0' を入力します。

1 文字入力し入力した文字の文字コードから '0' の文字コードを引きます。これで、数字に対応した数値が求められます。入力された各桁の値は、メモリ上の領域に保存しておきます。

$$\text{数値} = \text{数字} - \text{'0'の文字コード}$$

ここまでの手順をプログラムにすると図 6.10 のようになります。プログラム中で 1 文字入力 (GETCH) 毎に 1 文字出力 (PUTCH) しているのは、入力した文字がパソコンの画面に表示されるようにするためです。



| ラベル    | ニーモニック      | コメント        |
|--------|-------------|-------------|
| GETDEC | LD SP,#0DCH | ; SP 初期化    |
|        | CALL GETCH  | ; 1 文字入力    |
|        | CALL PUTCH  | ; エコーバック    |
|        | SUB GO,#'0' | ; 文字コードから値に |
|        | ST GO,D3    | ; 100 位桁    |
|        | CALL GETCH  | ; 1 文字入力    |
|        | CALL PUTCH  | ; エコーバック    |
|        | SUB GO,#'0' | ; 文字コードから値に |
|        | ST GO,D2    | ; 10 位桁     |
|        | CALL GETCH  | ; 1 文字入力    |
|        | CALL PUTCH  | ; エコーバック    |
|        | SUB GO,#'0' | ; 文字コードから値に |
|        | ST GO,D1    | ; 1 位桁      |

図 6.10 3 桁入力し数値に変換してメモリに保存する部分

| ラベル | ニーモニック     | コメント     |
|-----|------------|----------|
|     | LD GO,D3   | ; 100 位桁 |
|     | CALL MUL10 | ; × 10   |
|     | ADD GO,D2  | ; 10 位桁  |
|     | CALL MUL10 | ; × 10   |
|     | ADD GO,D1  | ; 1 位桁   |
|     | ST GO,DATA | ; 結果を保存  |

図 6.11 各桁の値を合成する部分

### 各桁の値の合成

下の式で目的の値を求めることができます。2 行目のように計算すれば 10 倍を計算するサブルーチンを 2 回呼び出せば良いことになります。

$$\begin{aligned} \text{値} &= 100 \text{ 位桁} \times 100 + 10 \text{ 位桁} \times 10 + 1 \text{ 位桁} \\ &= ((100 \text{ 位桁} \times 10) + 10 \text{ 位桁}) \times 10 + 1 \text{ 位桁} \end{aligned}$$

この計算をプログラムにすると図 6.11 のようになります。ここで MUL10 は、GO 値の 10 倍を計算するサブルーチンです。予め、「例題 5-6 シフトを用いた高速乗算」を参考に、MUL10 を作成しておく必要があります。

### 掛算の回数

一般に、掛算や割算は時間のかかる計算です。TeC のような掛算命令や割算命令を持たない小さなコンピュータにとっては特に時間がかかります。掛算や割算の回数は少なくするべきです。「各桁の値の合成」で掛算の回数が少なくなるように式を変形したのは、そのような意味からです。

### 問題

1. SIO から入力した文字の文字コードを 16 進数で出力するプログラムを作成しなさい。(1 文字入力サブルーチン (GETCH)、16 進数出力サブルーチン (PUTHEX) を作成し、これらを用いてプログラムを完成しなさい。)
2. 本文中のプログラムを参考に、DATA 番地のデータを 10 進数で出力するプログラムを作成しなさい。
3. 10 進数の出力プログラムを、出力する数値の桁数に合わせて先行する余分な '0' が出力されないよう改良しなさい。
4. 10 進数の出力プログラムを、D1, D2, D3 領域の代わりにスタックを使用するように改造しなさい。
5. 本文中のプログラムを参考に、10 進数を入力し DATA 番地に格納するプログラムを作成しなさい。
6. 10 進数を入力するプログラムを、数字以外が入力されるまで入力を受け付けるように改造しなさい。(3 桁固定でなく、10[エンター] や 1[エンター] で入力できるプログラムにする。)
7. 10 進数を入力し、10 加えた値を 10 進数で出力するプログラムを作成しなさい。(10 進数入力サブルーチン (GETDEC)、10 進数出力サブルーチン (PUTDEC) を作成し、これらを用いてプログラムを完成しなさい。)
8. 二つの 10 進数を入力し、和を 10 進数で出力するプログラムを作成しなさい。
9. ゼロが入力されるまで 10 進数を入力し、合計を 10 進数で出力するプログラムを作成しなさい。

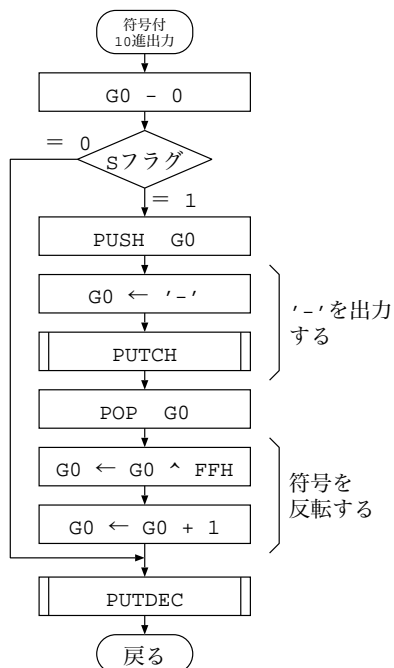


図 6.12 符号付き 10 進出力

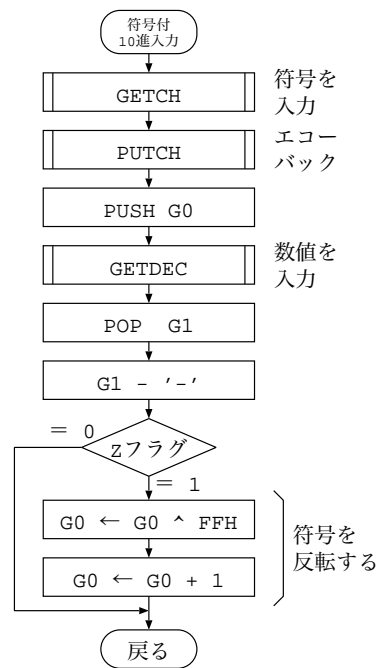


図 6.13 符号付き 10 進入力

## 6.6 符号付数の入出力

「2.3 負数の表現」で学んだように、負の数は 2 の補数表現を用いて表現します。TeC は 8bit 符号付 2 進数を扱うので -128 ~ +127 の数値を扱うことができます。

データが符号付であるか符号無しであるかは、そのデータをプログラムがどのように扱うかで決まります。特にデータの外部表現を左右する入出力ルーチンは、処理内容がはっきり区別されます。

### 6.6.1 符号付 10 進数の出力

図 6.12 に G0 に格納された符号付 2 進数の値を SIO へ出力するサブルーチンのフローチャートを示します。なお、G0 に格納した 1 文字を出力する PUTCH サブルーチン、G0 に格納した値を 10 進数として出力する PUTDEC サブルーチンが別に必要です。

まず、G0 の値と 0 を比較し、G0 の値が正か負かを判断します。負の場合は、'-' を出力した後、G0 の値の符号を反転します。符号の反転は、「2.3.6 2 の補数から元の数を求める手順」で学習したように、「**ビット反転した後、1 を加える**」ことで実行できます。ビットの反転には XOR 命令を使用します。最後に、PUTDEC サブルーチンを用いて G0 の値を 10 進数で出力します。

### 6.6.2 符号付 10 進数の入力

次のような形式の 10 進数を入力します。

正の数：空白 数字 数字 数字

負の数：'-' 数字 数字 数字

図 6.13 に符号付数値を入力するサブルーチンのフローチャートを示します。なお、SIO から 1 文字を入力し G0 に格納するサブルーチン GETCH、SIO から 10 進数を入力し G0 に格納するサブルーチン GETDEC が、予め準備してあるものとします。

まず、符号を入力しスタックに退避します。続いて 10 進数を G0 に入力します。退避しておいた符号を G1 に取り出し、文字 '-' と比較します。同じなら、G0 の符号を反転します。

#### 問題

1. 符号付の 10 進数出力サブルーチン (PUTSDEC) を作成しなさい。
2. 符号付の 10 進数入力サブルーチン (GETSDEC) を作成しなさい。
3. ゼロが入力されるまで符号付の 10 進数を入力し、合計を符号付 10 進数で出力するプログラムを作成しなさい。

| 番地 | 機械語         | ラベル  | ニーモニック           |
|----|-------------|------|------------------|
| 1E | 53 48 49 47 | DATA | DC "SHIGEMURA",0 |
| 22 | 45 4D 55 52 |      |                  |
| 26 | 41 00       |      |                  |

図 6.14 文字列データの表現例

## 6.7 アドレスデータ

番地をデータとして扱えると便利ことがあります。番地(=アドレス)のデータなので、ここではアドレスデータと呼びます。C 言語を勉強したことのある人は、ポインタのことと言えばピンとくると思います。アドレスデータを使用すると便利な例としては、サブルーチンにデータの置き場所を教える場合が考えられます。

### 6.7.1 TeC のアドレスデータ

TeC では、インデクスドモードのアドレッシングを用いて、レジスタで指定された特定アドレスのデータをアクセスすることが可能です。例えば、12H 番地にあるデータを G0 レジスタにロードするプログラムは、次のようになります。

```
LD    G1,#12H
LD    G0,0,G1
```

G1 インデクスドモードでは、データのアドレスを「第 2 バイト + G1」で求めます。このプログラムの場合、第 2 バイトが 0 ですから、G1 の値がそのままデータのアドレスになります。つまり、G1 に格納されたアドレスデータを用いて、そのアドレスをアクセスします。

### 6.7.2 文字列出力サブルーチン

アドレスデータの使用例として、文字列出力サブルーチンを作ってみましょう。サブルーチンには出力すべき文字列を渡す必要があります。ここでは文字列を「図 6.14 文字列データの表現例」に示すような文字コード 0 で終わるバイト列として表現することになります。(この文字列表現方式は、C 言語と同じです。)

サブルーチンにデータを渡す方法として、これまではレジスタを使用してきました。例えば 1 文字出力サブルーチン (PUTCH) は、出力したい文字の文字コードを G0 に格納して呼び出されるようになっていました。文字列の場合はデータの量が多いのでレジスタにデータ全体を格納してサブルーチンに渡すことができません。そこで、文字列データの置いてあるアドレスを渡すこと

にします。

図 6.15 のプログラムは、文字列出力サブルーチン (PUTSTR) が出力する文字列のアドレスを受け取るようにした例です。G1 に文字列のアドレスをセットして PUTSTR を呼び出します。なお、プログラム中「#STR1」のような記述は STR1 ラベルの値を意味します。どのような機械語へ変換されているか確認してください。

### 6.7.3 ジャンプテーブル

これまで、データのアドレスばかり考えてきましたが、プログラムのアドレスも同様に扱うことができます。インデクスドモードのアドレッシングは、ジャンプ命令や CALL 命令でも使用できます。

このことを利用すると、表に登録してあるいくつかのサブルーチンから一つを選んで実行するプログラムをつくることができます。サブルーチンを登録してある表のことを、ここでは、ジャンプテーブルと呼びます。

#### ジャンプ命令を表にする場合

ジャンプ命令や CALL 命令のインデクスドアドレッシングはジャンプ先アドレスをインデクスレジスタを用いて計算します。例えば、次の JMP 命令は A + G2 番地にジャンプします。

```
JMP    A,G2
```

次のプログラムは、G2 の値を使用して TBL 以下のどれか一つの JMP 命令を選択し、PRG0, PRG1, ... のいずれかにジャンプする例です。つまり、Java 言語や C 言語の switch 文に似た動きをします。JMP 命令は 2 バイトの命令なので、SHLL を用いて G2 の値を 2 倍にしています。

```
LD      G2,N
SHLL    G2
JMP     TBL,G2
TBL     JMP    PRG0
        JMP    PRG1
        ...
```

CALL 命令を用いてサブルーチンを呼び出すこともできます。CALL 命令の次の命令にサブルーチンから戻ってき来るので、注意して下さい。

| 番地 | 機械語   | ラベル    | ニーモニック           |
|----|-------|--------|------------------|
| 02 |       | SIOD   | EQU 02H          |
| 03 |       | SIOS   | EQU 03H          |
| 00 |       |        |                  |
| 00 | 1F DC | START  | LD SP,#ODCH      |
| 02 | 17 2F |        | LD G1,#STR1      |
| 04 | B0 13 |        | CALL PUTSTR      |
| 06 | 17 41 |        | LD G1,#CRLF      |
| 08 | B0 13 |        | CALL PUTSTR      |
| 0A | 17 39 |        | LD G1,#STR2      |
| 0C | B0 13 |        | CALL PUTSTR      |
| 0E | 17 41 |        | LD G1,#CRLF      |
| 10 | B0 13 |        | CALL PUTSTR      |
| 12 | FF    |        | HALT             |
| 13 |       |        |                  |
| 13 |       |        | ； 文字列出力ルーチン      |
| 13 | D0    | PUTSTR | PUSH G0          |
| 14 | D4    |        | PUSH G1          |
| 15 | 11 00 | LOOP   | LD G0,0,G1       |
| 17 | 53 00 |        | CMP G0,#0        |
| 19 | A4 21 |        | JZ RETURN        |
| 1B | B0 24 |        | CALL PUTCH       |
| 1D | 37 01 |        | ADD G1,#1        |
| 1F | A0 15 |        | JMP LOOP         |
| 21 | D6    | RETURN | POP G1           |
| 22 | D2    |        | POP G0           |
| 23 | EC    |        | RET              |
| 24 |       |        |                  |
| 24 |       |        | ； 1 文字出力ルーチン     |
| 24 | D4    | PUTCH  | PUSH G1          |
| 25 | C4 03 | WAIT   | IN G1,SIOS       |
| 27 | 67 80 |        | AND G1,#80H      |
| 29 | A4 25 |        | JZ WAIT          |
| 2B | C3 02 |        | OUT G0,SIOD      |
| 2D | D6    |        | POP G1           |
| 2E | EC    |        | RET              |
| 2F |       |        |                  |
| 2F | 53 48 | STR1   | DC "SHIGEMURA",0 |
| 31 | 49 47 |        |                  |
| 33 | 45 4D |        |                  |
| 35 | 55 52 |        |                  |
| 37 | 41 00 |        |                  |
| 39 | 54 45 | STR2   | DC "TETSUJI",0   |
| 3B | 54 53 |        |                  |
| 3D | 55 4A |        |                  |
| 3F | 49 00 |        |                  |
| 41 | 0D 0A | CRLF   | DC 0DH,0AH,0     |
| 43 | 00    |        |                  |

```

LD      G2,N
SHLL    G2
CALL    TBL,G2
サブルーチンから戻る場所

```

```

...
TBL     JMP     PRG0
JMP     PRG1
...

```

#### アドレスを表にする場合

ジャンプ命令を表にすると表の要素が2バイトになります。アドレスだけを表にすると表の要素が1バイトで済み、メモリの節約になります。

次のプログラムは、アドレスを表にした例です。G1の値を使用し、表からPRG0、PRG1、...のどれかのアドレスを選びG2にロードします。次に、JMP命令でG2の内容が表す番地にジャンプします。

```

LD      G1,N
LD      G2,TBL,G1
JMP     0,G2
TBL     DC     PRG0
DC      PRG1
...

```

次のプログラムは、CALL命令を用いてサブルーチン呼び出す例です。

```

LD      G1,N
LD      G2,TBL,G1
CALL    0,G2
サブルーチンから戻る場所

```

```

...
TBL     DC     PRG0
DC      PRG1
...

```

#### 問題

1. G1レジスタで指定された番地から始まるG0レジスタで指定されたバイト数の領域を、00Hで埋めつくすサブルーチンを作りなさい。
2. C言語のswitch文が、TeCの機械語でどのように表現できるか考察しなさい。

図 6.15 文字列出力サブルーチン

**実効アドレス**

本文では、演算、転送やジャンプの対象となるメモリのアドレスのことを、「データのアドレス」、「ジャンプ先アドレス」と書いてありますが、正しい用語では「実効アドレス (Effective Address)」と呼びます。また、Effective Address を EA と略して書くこともあります。命令表で命令の動作説明の欄に EA とか [EA] の記述が出てきますが、これは実効アドレスそのものや、実効アドレスに置いてあるデータのことを指しています。

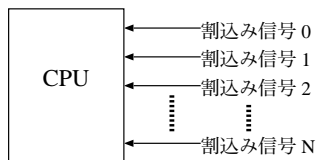


図 6.16 割り込み信号

**6.8 割り込み (Interrupt)**

ノイマン型のコンピュータはプログラムの命令を逐次実行しますので、仕事を順番に処理するのが基本です。しかし緊急の仕事が発生したとき等は、処理中の仕事を後回しにして、緊急の仕事を先に処理しなければならない場合があります。割り込みは、そのような緊急の事象をコンピュータに伝えて、優先される処理を割り込ませる仕組みです。英語では割り込みのことを、Interrupt と言います。

割り込みは CPU の外部で発生した事象をハードウェアにより CPU に伝えることにより発生します。割り込みにはいくつかの種類があり、図 6.16 に示すように、CPU には割り込みの種類毎に入力が用意されます。

割り込みが発生すると、CPU は現在実行中のプログラムを中断し、予め登録された割り込み処理プログラム（「割り込み処理ルーチン」または「割り込みハンドラ」と呼びます）の実行を開始します。割り込み処理ルーチンは割り込みの種類毎に登録することができます。

**6.8.1 TeC の割り込み種類**

TeC の割り込みの種類は 4 種類です。表 6.1 に一覧表を示します。4 種類の割り込みには、「INT0」、「INT1」、「INT2」、「INT3」という名前が付いています。ベクタ

表 6.1 TeC の割り込みの種類

| 名前   | 意味            | ベクタ |
|------|---------------|-----|
| INT0 | インターバルタイマ割り込み | DCH |
| INT1 | SIO 受信割り込み    | DDH |
| INT2 | SIO 送信割り込み    | DEH |
| INT3 | コンソール割込       | DFH |

は割り込み処理ルーチンを登録するメモリの番地です。（主記憶の DCH～DFH 番地の範囲はベクタと呼ばれる特別な領域です。）例えば「INT0」の割り込み処理ルーチンが 12H 番地から始まる場合は、メモリの DCH 番地に、12H を書き込みます。

**インターバルタイマ割り込み**

“INT0” はインターバルタイマに接続されています。インターバルタイマは指定された時間間隔で繰り返し割り込みを発生します。指定できる間隔は 1/75 秒～256/75 秒の範囲です。

**SIO 受信割り込み**

“INT1” は SIO に接続されており SIO が文字を受信する度に割り込みが発生します。

**SIO 送信割り込み**

“INT2” も SIO に接続されており SIO が次の文字を送信可能になる度に割り込みが発生します。

**コンソール割り込み**

“INT3” はコンソールパネルに接続されています。プログラム実行中にユーザがコンソールパネルの WRITE スイッチを押すと割り込みが発生します。

**6.8.2 TeC の割り込み動作**

TeC の CPU はプログラム実行中、前の機械語命令を実行し終え、次の機械語命令を実行開始する前に、割り込みが発生していないかチェックします。割り込みが発生していた場合は次のような動作をします。

1. PC をスタックに保存 (PUSH) する。
2. 割り込み許可フラグをリセットする。（割り込み禁止にする）
3. 割り込みの種類 (INT0～INT3) に対応するベクタから、割り込み処理ルーチンのアドレスを読む。
4. 割り込み処理ルーチンのアドレスを PC にセットする。

### 6.8.3 EI (Enable Interrupt) 命令

Enable は「可能にする」という意味の英語です。EI 命令は割り込み許可フラグをセットし割り込みを「可能」にします。

CPU がリセットされたとき（初期状態）は割り込み禁止状態です。プログラムは、SP の初期化や割り込みベクタの設定等が完了し、割り込みを受け付ける準備ができたとき EI 命令を用いて割り込みを許可します。

割り込みが許可状態かそうでないかは、「割り込み許可フラグ」の状態によって決まります。EI 命令は「割り込み許可フラグ」をセットします。

**意味：** 割り込みを許可する。（割り込み許可フラグをセットする。）

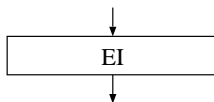
**フラグ：** C, S, Z フラグは変化しません。割り込み許可フラグがセットされます。

**ニーモニック：** EI

**命令フォーマット：** EI 命令は 1 バイト長の命令です。

| 第1バイト             |                 |                 |
|-------------------|-----------------|-----------------|
| OP                | GR XR           |                 |
| 1110 <sub>2</sub> | 00 <sub>2</sub> | 00 <sub>2</sub> |

**フローチャート：** 次のように描きます。



### 6.8.4 DI (Disable Interrupt) 命令

Disable は、「できなくする」という意味の英語です。DI 命令は割り込み許可フラグをリセットし割り込みを「できなく」します。プログラムが EI 命令を用いて割り込みを許可したあと、割り込まれると都合が悪くなったとき使用します。

割り込みが発生し割り込み処理ルーチンの実行が始まった時は、自動的に割り込み禁止状態になります。それ以外で割り込み禁止にしたいとき使用する命令です。

**意味：** 割り込みを禁止する。（割り込み許可フラグをリセットする。）

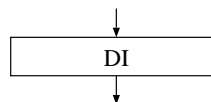
**フラグ：** C, S, Z フラグは変化しません。割り込み許可フラグがリセットされます。

**ニーモニック：** DI

**命令フォーマット：** DI 命令は 1 バイト長の命令です。

| 第1バイト             |                 |                 |
|-------------------|-----------------|-----------------|
| OP                | GR XR           |                 |
| 1110 <sub>2</sub> | 00 <sub>2</sub> | 11 <sub>2</sub> |

**フローチャート：** 次のように描きます。



### 6.8.5 RETI (Return from Interrupt) 命令

割り込み処理ルーチンから通常のプログラムへ戻るための、特別な RET 命令です。RETI 命令を実行して割り込み処理ルーチンが終了すると、CPU は割り込み許可状態になり割り込みが発生したとき実行中だったプログラムに戻ります。

**意味：** 割り込み処理ルーチンから戻る。

**フラグ：** C, S, Z フラグは変化しません。割り込み許可フラグがセットされます。

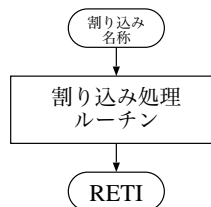
**ニーモニック：** RETI

**命令フォーマット：** RETI 命令は 1 バイト長の命令です。

| 第1バイト             |                 |                 |
|-------------------|-----------------|-----------------|
| OP                | GR XR           |                 |
| 1110 <sub>2</sub> | 11 <sub>2</sub> | 11 <sub>2</sub> |

**動作の詳細：** スタックから値を一つ取り出し PC にセットし、割り込み許可フラグをセットします。

**フローチャート：** 次のフローチャートの最後の角を取った四角が RETI 命令に対応します。フローチャートは割り込み処理ルーチン全体を示しています。



### 6.8.6 PUSHF (Push Flag) 命令

フラグをスタックに保存する命令です。主に割り込み処理ルーチンで使用します。割り込みは、いつ発生するか分かりません。割り込み処理ルーチンが終了したあと以前のプログラムに戻って処理を続行するには、CPU の状態を完全に元通りに復元する必要があります。CPU の状態とはレジスタ (PC, SP を含む) とフラグの値のこと



です。レジスタの値は、PUSH 命令でスタックに退避することができます。フラグの値は、PUSHF 命令でスタックに退避します。

**意味：** フラグをスタックに退避する。

**フラグ：** フラグは変化しません。

**ニーモニック：** PUSHF

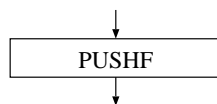
**命令フォーマット：** PUSHF 命令は 1 バイト長の命令です。

| 第 1 バイト           |                                 |
|-------------------|---------------------------------|
| OP                | GR XR                           |
| 1101 <sub>2</sub> | 11 <sub>2</sub> 01 <sub>2</sub> |

**動作の詳細：** スタックに退避されるデータは次のような 1 バイトです。E が割込み許可フラグです。

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | 0 | 0 | 0 | 0 | C | S | Z |
|---|---|---|---|---|---|---|---|

**フローチャート：** 次のように描きます。



### 6.8.7 POPF (Pop Flag) 命令

フラグをスタックから復元する命令です。主に割込み処理ルーチンで使用します。割込み処理ルーチンは POPF 命令を使用してフラグの値を復元してから、RETI 命令で以前のプログラムに戻ります。

**意味：** フラグをスタックから復元します。

**フラグ：** フラグはスタックから取り出した値により変化します。

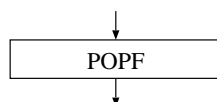
**ニーモニック：** POPF

**命令フォーマット：** POPF 命令は 1 バイト長の命令です。

| 第 1 バイト           |                                 |
|-------------------|---------------------------------|
| OP                | GR XR                           |
| 1101 <sub>2</sub> | 11 <sub>2</sub> 11 <sub>2</sub> |

**動作の詳細：** スタックから取り出すデータは PUSHF 命令と同じ形式です。

**フローチャート：** 次のように描きます。



## 6.9 タイマ割込み

一般に、コンピュータには割込みを発生するタイマが内蔵されています。時間を計ったり複数の仕事を切替えるきっかけとして、タイマから発生する割込みを使用するからです。

パソコンでは複数のプログラムが同時に実行されているように見えますが、これは 1/10 ～ 1/100 秒程度の間隔で実行するプログラムを切替えることにより、複数のプログラムを同時に実行しているように見せているのです。このとき、プログラムを切替えるきっかけになるのはタイマからの割込みです。

### 6.9.1 インターバルタイマ割込み

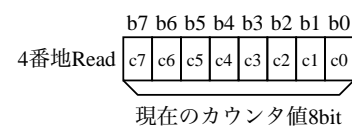
TeC にはプログラムから周期を設定できるインターバルタイマが一つ内蔵されています。インターバルタイマは指定した時間間隔で繰返し割込みを発生します。インターバルタイマには 8 ビットのタイマ・カウンタと 8 ビットの周期レジスタが内蔵されています。

#### I/O ポート

インターバルタイマの I/O ポートは、4 番地、5 番地に配置されています（「表 5.1 I/O マップ」参照）。

#### 1. タイマ・カウンタ

4 番地を IN 命令で読むとタイマ・カウンタの現在値を読み出すことができます。タイマ動作中、この値は 1/75 秒毎にカウントアップされます。そして値が周期レジスタの値と一致したら 0 に戻ります。その後、カウントアップが再開されます。

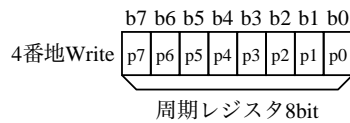


#### 2. 周期レジスタ

周期レジスタ値によって割込みの発生周期が決まります。4 番地に OUT 命令で書き込んだ値が周期レジスタに書き込まれます。書き込む値を  $n$  とすると周期は次の式で計算できます。

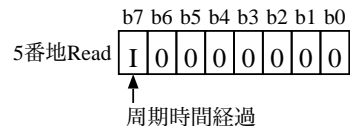
$$\text{周期} = (n + 1) / 75 \text{ 秒}$$

周期レジスタの初期値は 1 秒に相当する 74 です。「周期」を変更するとタイマはカウントを停止します。下記のコントロールを使用してスタートし直す必要があります。



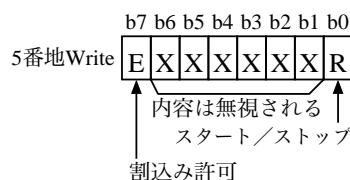
### 3. ステータス

5番地を IN 命令で読むとタイマがスタートしてから一周期の時間が経過したかどうかを割込みを用いることなく知ることができます。一周期時間が経過すると  $b_7$  の値が“1”に変化します。 $b_7$  は「ステータス」を一度読みだすと“0”にクリアされます。その後一周期時間が経過する度に  $b_7$  の値が“1”に変化します。



### 4. コントロール

タイマ割込みの（許可／不許可）、タイマカウンタのカウンタ（スタート／ストップ）の情報を5番地に OUT 命令で書き込みます。 $b_7$  が割込みの許可ビット、 $b_0$  がタイマカウンタのスタートビットです。タイマカウンタをスタートすると、タイマカウンタの値が“00H”にリセットされてからカウントが始まります。



#### 6.9.2 プログラミング例

図 6.17 にインターバルタイマを用いたプログラムの例を示します。このプログラムは1秒間隔でブザーの ON/OFF を繰り返します。

##### メインプログラム

1. スタックポインタの値を初期化します。割込みが発生すると CPU が自動的に PC をスタックに保存するので、**割込みが発生する前にスタックポインタの値を初期化する必要があります。**
2. INT0 の割込みベクタに割込みルーチンの番地を書き込みます。DCH 番地が INT0 の割込みベクタですので、この番地に割込み処理ルーチンの先頭番地を書き込みます。

| ラベル    | ニーモニック     |             | コメント        |
|--------|------------|-------------|-------------|
| BUZ    | EQU        | 00H         | ; ブザーポート    |
| TMRVCT | EQU        | 0DCH        | ; 割込みベクタ    |
| TMRCNT | EQU        | 4           | ; カウンタ      |
| TMRCTR | EQU        | 5           | ; コントロール    |
| START  | ; メインプログラム |             |             |
|        | LD         | SP, #0DCH   | ; SP 初期化    |
|        | LD         | GO, #TMRINT | ; 割込みルーチンを  |
|        | ST         | GO, TMRVCT  | ; ベクタに登録    |
|        | LD         | GO, #74     | ; タイマ初期化    |
|        | OUT        | GO, TMRCNT  | ; 周期=1 秒,   |
|        | LD         | GO, #81H    | ; 割込み許可,    |
|        | OUT        | GO, TMRCTR  | ; カウント開始    |
|        | EI         |             | ; CPU の割込許可 |
| WAIT   | LD         | GO, FLG     | ; フラグを      |
|        | CMP        | GO, #0      | ; チェック      |
|        | JZ         | WAIT        | ; 0の間は待つ    |
|        | LD         | GO, VAL     | ; ブザーポートの   |
|        | XOR        | GO, #01H    | ; 値を反転して    |
|        | ST         | GO, VAL     | ; 記録すると     |
|        | OUT        | GO, BUZ     | ; 同時に出力     |
|        | LD         | GO, #0      | ; フラグをクリア   |
|        | ST         | GO, FLG     | ; 次の割込みを    |
|        | JMP        | WAIT        | ; 待つ        |
|        | ; 共有変数     |             |             |
| FLG    | DC         | 0           | ; フラグ       |
| VAL    | DC         | 0           | ; ブザーポートの   |
|        |            |             | ; 現在値ひかえ    |
|        | ; 割込みルーチン  |             |             |
| TMRINT | PUSHF      |             | ; コンテキストを   |
|        | PUSH       | GO          | ; 保存する      |
|        | LD         | GO, #1      | ; フラグに      |
|        | ST         | GO, FLG     | ; 1 を書き込む   |
|        | POP        | GO          | ; コンテキストを   |
|        | POPF       |             | ; 復元し       |
|        | RETI       |             | ; 割込み終了     |

図 6.17 タイマ割込みプログラムの例

3. インターバルタイマの初期化を行います。1秒ごとの割込みが必要なので74を周期レジスタに書き込みます。続けてコントロールに81Hを書き込むことで割込み可能な状態でタイマがカウントを開始します。
4. EI 命令を実行して CPU を割込み許可状態にします。これで、割込みを受け付ける準備ができました。ループを実行して割込みの発生を待ちます。



TeC は CPU が停止しているときは割込みを受け付けませんので、プログラムを終了しないで割込みの発生を待つ必要があります。

5. 割込みが発生すると割込みルーチンにより共有変数 FLG の値が “1” に書き換えられます。FLG の値が “1” になったらブザーの ON/OFF を変更した後、FLG の値を “0” に戻します。

#### 割込みルーチン

1. メインルーチンがインターバルタイマを起動したので、1 秒に一度、インターバルタイマ割込み (INT0) が発生します。
2. 割込みが発生すると CPU は PC の値をスタックに保存した後、割込みベクタに書いてある番地にジャンプします。メインプログラムが INT0 の割込みベクタに割込みルーチンの先頭番地を書き込んであるので、割込みルーチンへジャンプします。
3. 割込みルーチンは、メインプログラムを実行中の CPU の状態 (コンテキスト) を保存します。割込みルーチンが破壊するのは G0 レジスタだけですが、念のためフラグも保存しています。
4. 割込み処理の内容は FLG 変数に “1” を書き込むことです。
5. 処理が終わったら CPU の状態 (コンテキスト) を元に戻してメインプログラムに RETI 命令で戻ります。

#### 問題

1. 「タイマ割込みプログラムの例」の動作を確認しなさい。
2. “LD G0, #TMRINT” は、どんな機械語に変換されたか確認しなさい。

#### 例題 6-5 3 分間タイマ

**問題：** タイマ割込みを使用した 3 分間タイマを作りなさい。タイマは、3 分経過したらブザーを鳴らしっぱなしにするものとします。

**考え方：** 1 秒間隔でタイマ割込みが発生するようにし割込みの回数で時間を計ります。180 回目の割込みが発生したらブザーを鳴らしたままプログラムを終了します。ブザーはリセットボタンを押して止めます。

**プログラム：** 割込みルーチンが 180 をカウントしたらフラグ (FLAG) セットします。メインルーチンは、フラグがセットされたらブザーを ON にして、プログラムを終了します。

| ラベル    | ニーモニック         | コメント     |
|--------|----------------|----------|
| TMRVCT | EQU ODCH       |          |
| TMRCNT | EQU 4          | ; カウンタ   |
| TMRCTR | EQU 5          | ; コントロール |
|        | ; メインプログラム     |          |
| START  | LD SP, #ODCH   | ; SP 初期化 |
|        | LD G0, #0      | ; フラグと   |
|        | ST G0, FLAG    | ; カウンタ   |
|        | ST G0, CNT     | ; リセット   |
|        | LD G0, #TMRINT |          |
|        | ST G0, TMRVCT  | ; ベクタ設定  |
|        | LD G0, #74     | ; タイマ初期化 |
|        | OUT G0, TMRCNT | ; 周期=1 秒 |
|        | LD G0, #81H    | ; 割込み許可  |
|        | OUT G0, TMRCTR | ; カウント開始 |
|        | EI             | ; 割込み許可  |
| L1     | LD G0, FLAG    | ; フラグが 0 |
|        | CMP G0, #0     | ; の間     |
|        | JZ L1          | ; 繰り返す   |
|        | LD G0, #01H    |          |
|        | OUT G0, 00H    | ; ブザー ON |
|        | HALT           | ; 終了     |
|        | ; 割込み処理ルーチン    |          |
| TMRINT | PUSHF          | ; コンテキス  |
|        | PUSH G0        | ; スト保存   |
|        | LD G0, CNT     | ; 回数のカウ  |
|        | ADD G0, #1     | ; ント     |
|        | ST G0, CNT     |          |
|        | CMP G0, #180   | ; 3 分経過  |
|        | JZ TINT1       |          |
|        | JMP TINT2      |          |
| TINT1  | ST G0, FLAG    | ; フラグ ON |
| TINT2  | POP G0         | ; コンテキ   |
|        | POPF           | ; スト復元   |
|        | RETI           |          |
|        | ; 作業領域         |          |
| FLAG   | DS 1           | ; フラグ    |
| CNT    | DS 1           | ; カウンタ   |

## 6.10 コンソール割込み

コンソールパネルの操作で INT3 割込みを発生することができます。コンソール割込みを許可する手順は次の通りです。

1. スタックポインタを初期化する。
2. INT3 割込みベクタにコンソール割込み処理ルーチンの番地を書き込む。
3. コンソール割込み (INT3) 許可ビットを ON にする。割込み許可ビットは I/O 6 番地の  $b_0$  です (「表 5.1 I/O マップ」参照)。
4. EI 命令で CPU の割込みを許可する。

以上の準備をした上でプログラム実行中にコンソールパネルの WRITE ボタンを押すと割込が発生します。

### コンソール割込みの応用例

図 6.18 コンソール割込みを応用したプログラム例を示します。プログラムはコンソール割込みとタイマ割込みの両方を使用しています。コンソールパネルの WRITE ボタンが押されるたびに 0.3 秒間ブザーを鳴らします。以下の手順で機能が実現されています。

#### 1. 初期化

メインプログラムはスタックポインタとタイマ等のハードウェアを初期化し、CPU を割込み許可状態にします。初期化が完了したら割込み待ちの無限ループに入ります。

#### 2. コンソール割込み

ユーザが WRITE ボタンを押して割込が発生すると、コンソール割込みルーチンが起動されます。コンソール割込みルーチン中でブザーを ON にした上でタイマをスタートします。

#### 3. タイマ割込み

タイマがスタートして 0.3 秒後にタイマ割込が発生します。タイマ割込みルーチンではブザーを OFF にしてタイマを停止します。

| ラベル    | ニーモニック         |            | コメント          |
|--------|----------------|------------|---------------|
| BUZ    | EQU            | 00H        | ; ブザーポート      |
| TMRVCT | EQU            | 0DCH       | ; TMR 割込みベクタ  |
| CONVCT | EQU            | 0DFH       | ; CON 割込みベクタ  |
| TMRCNT | EQU            | 4          | ; TMR (周期)    |
| TMRCTR | EQU            | 5          | ; TMR コントロール  |
| CONCTR | EQU            | 6          | ; CON コントロール  |
| START  | ; メインプログラム     |            |               |
|        | LD             | SP,#0DCH   | ; SP 初期化      |
|        | LD             | GO,#CONINT | ; コンソール割込み    |
|        | ST             | GO,CONVCT  | ; ルーチン登録      |
|        | LD             | GO,#01H    | ; コンソール割込み    |
|        | OUT            | GO,CONCTR  | ; 許可          |
|        | LD             | GO,#TMRINT | ; タイマ割込み      |
|        | ST             | GO,TMRVCT  | ; ルーチン登録      |
|        | LD             | GO,#25     | ; 周期=0.3SEC   |
|        | OUT            | GO,TMRCNT  | ; タイマ周期セット    |
|        | EI             |            |               |
| WAIT   | JMP            | WAIT       | ; 無限ループ       |
|        | ; コンソール割込みルーチン |            |               |
| CONINT | PUSHF          |            | ; コンテキストを     |
|        | PUSH           | GO         | ; 保存する        |
|        | LD             | GO,#01H    |               |
|        | OUT            | GO,BUZ     | ; BEEP ON     |
|        | LD             | GO,#81H    |               |
|        | OUT            | GO,TMRCTR  | ; TIMER START |
|        | POP            | GO         |               |
|        | POPF           |            | ; コンテキストを     |
|        | RETI           |            | ; 復旧する        |
|        | ; タイマ割込みルーチン   |            |               |
| TMRINT | PUSHF          |            | ; コンテキストを     |
|        | PUSH           | GO         | ; 保存する        |
|        | LD             | GO,#00H    |               |
|        | OUT            | GO,BUZ     | ; BEEP OFF    |
|        | LD             | GO,#00H    |               |
|        | OUT            | GO,TMRCTR  | ; TIMER STOP  |
|        | POP            | GO         |               |
|        | POPF           |            | ; コンテキストを     |
|        | RETI           |            | ; 復旧する        |
|        |                |            | ; 戻る          |

図 6.18 コンソール割込みプログラムの例

## 6.11 入出力割込み

CPU の処理速度と入出力装置の動作速度には大きな差があるので、入出力装置を動かすとき CPU は長い待ち状態になります。入出力装置の**動作完了**を待つ CPU がループを回り続けるだけでは時間が勿体ないので、その間に他の仕事ができる仕組みが望まれます。

そこで、入出力装置は**動作が完了**したことを CPU に割込みで知らせることができるよう設計してあります。キーボード、マウス、ハードディスク、ネットワーク等のほとんどの入出力装置が、動作完了の割込みを発生する機構を持っています。このような割込みを「**入出力割込み**」と呼びます。

### 6.11.1 TeC の入出力割込み

TeC は SIO から入出力割込みを発生することができます。SIO からの割込みは、文字を受信したとき発生する「受信割込み」と、次の文字を送信可能になったとき発生する「送信割込み」の 2 種類があります。これらの割込みを使用するかしないかは、割込み許可ビットにより別々に制御できるようになっています。

#### SIO 受信割込み

TeC の SIO は文字を受信したとき、INT1 割込みを発生することができます。割込みの発生を許可する手順は次の通りです。

1. スタックポインタを初期化する。
2. INT1 割込みベクタに SIO 受信割込み処理ルーチンの番地を書き込む。
3. SIO の受信割込み許可ビットを ON にする。割込み許可ビットは、I/O 3 番地の b6 です。（「[5.14.3 I/O ポート](#)」の「コントロール」参照）
4. EI 命令で CPU の割込みを許可する。

文字を受信すると INT1 割込みが発生します。受信した文字を I/O マップ 3 番地の SIO 受信データレジスタ（「[表 5.1 I/O マップ](#)」参照）から読み出すことで INT1 割込みが解除されます。

#### SIO 送信割込み

TeC の SIO は文字を送信することが可能とき INT2 割込みを発生することができます。割込みの発生を許可する手順は次の通りです。

1. スタックポインタを初期化する。

2. INT2 割込みベクタに、SIO 送信割込み処理ルーチンの番地を書き込む。
3. SIO の送信割込み許可ビットをに ON する。割込み許可ビットは I/O 3 番地の b7 です。（「[5.14.3 I/O ポート](#)」の「コントロール」参照）
4. EI 命令で CPU の割込みを許可する。

次の文字を送信することが可能になると INT2 割込みが発生します。次に送信する文字を I/O マップ 3 番地の SIO 送信データレジスタ（「[表 5.1 I/O マップ](#)」参照）に書込みます。これで割込みが解除されます。これ以上、送信する文字が無い場合は、割込み許可ビットを OFF にしてください。

### 6.11.2 入出力割込みの使用例

SIO を割込み駆動で使用する例を示します。

#### SIO 送信割込みの使用

**図 6.19** にローマ字で自分の名前を出力するプログラムの例を示します。「**例題 6-4 文字列出力（改良版 2）**」等と同様なことを割込みを使用して行います。プログラムの動きは次の通りです。

1. メインルーチンで割込みを許可すると、すぐに最初の割込みが発生する。
2. 割込み処理ルーチンの中で最初の文字を SIO データレジスタに書き込む。
3. その後は 1 文字の送信が終了する度に割込みが発生するので、割込み処理ルーチンの中で SIO データレジスタに次の文字を書き込む。
4. 全ての文字を送信したら SIO の送信割込み許可ビットを OFF にし、それ以上の割込みが発生しないようにする。

この例では、メインルーチンは出力が終了するまでループを回りながら待つだけです。本来は、出力は割込みに任せて、他の処理を行うことができます。

#### SIO 受信割込みの使用

**図 6.20** に SIO から制御できるプログラムの例を示します。このプログラムは、ブザーを鳴らす間隔を SIO から受信した文字（アルファベット小文字）によって切替えることができます。リストでは省略していますが、「[6.4.3 0.2 秒タイマ](#)」で作成した MS200 サブルーチンを利用しています。実際に実行する場合は MS200 サブルーチンも打ち込んで下さい。

| ラベル     | ニーモニック            | コメント       |
|---------|-------------------|------------|
| INT2V   | EQU ODEH          |            |
| SIOD    | EQU 02H           |            |
| SIOC    | EQU 03H           |            |
|         | ; メインプログラム        |            |
| START   | LD SP,#0DCH       | ; SP 初期化   |
|         | LD GO,#SIOXINT    | ; 割込みルー    |
|         | ST GO,INT2V       | ; チンを登録    |
|         | LD GO,#80H        | ; SIO 送信割  |
|         | OUT GO,SIOC       | ; 込み許可 ON  |
|         | LD GO,#DATA       | ; データの位    |
|         | ST GO,POINTER     | ; 置をセット    |
|         | LD GO,#1          | ; 出力中 Flag |
|         | ST GO,FLAG        | ; セット      |
|         | EI                | ; 割込み許可    |
| LOOP    | LD GO,FLAG        | ; 出力中フラ    |
|         | CMP GO,#0         | ; グがクリア    |
|         | JZ END            | ; で終了      |
|         | JMP LOOP          | ; 割込み待ち    |
| END     | HALT              |            |
| DATA    | DC "SHIGEMURA"    |            |
|         | DC 0              |            |
|         | ; SIO 送信割込み処理ルーチン |            |
| SIOXINT | PUSHF             |            |
|         | PUSH GO           |            |
|         | PUSH G1           |            |
|         | LD G1,POINTER     | ; 一文字      |
|         | LD GO,0,G1        | ; 取り出す     |
|         | CMP GO,#0         | ; 文字列の     |
|         | JZ OWARI          | ; 終わりか?    |
|         | OUT GO,SIOD       | ; 送信する     |
|         | ADD G1,#1         | ; 次の文字     |
|         | ST G1,POINTER     | ; に進める     |
|         | JMP KAERI         |            |
| OWARI   | LD GO,#00H        | ; SIO 送信割  |
|         | OUT GO,SIOC       | ; 込み許可 OFF |
|         | ST GO,FLAG        | ; Flag クリア |
| KAERI   | POP G1            |            |
|         | POP GO            |            |
|         | POPF              |            |
|         | RETI              |            |
|         | ; 割込み処理ルーチンのデータ   |            |
| POINTER | DS 1              | ; 文字列位置    |
| FLAG    | DS 1              | ; 動作中 FLAG |

図 6.19 SIO 送信割込みの使用例

| ラベル     | ニーモニック            | コメント         |
|---------|-------------------|--------------|
| INT1V   | EQU ODDH          |              |
| BUZZ    | EQU 00H           |              |
| SIOD    | EQU 02H           |              |
| SIOC    | EQU 03H           |              |
|         | ; メインプログラム        |              |
| START   | LD SP,#0DCH       | ; SP 初期化     |
|         | LD GO,#SIORINT    | ; 割込みルーチン    |
|         | ST GO,INT1V       | ; を登録        |
|         | LD GO,#40H        | ; SIO 受信割込み  |
|         | OUT GO,SIOC       | ; 許可ビット ON   |
|         | LD GO,#'a'        | ; 最後に入力した    |
|         | ST GO,IKEY        | ; 文字='a'     |
|         | EI                | ; 割込み許可      |
| LOOP    | LD GO,IKEY        |              |
|         | ST GO,KEY         |              |
|         | CMP GO,#'z'       | ; 'z' が入力され  |
|         | JZ END            | ; たら終了       |
|         | LD GO,#'a'        |              |
| L1      | CALL MS200        | ; 200MS 待ち   |
|         | CMP GO,KEY        |              |
|         | JZ L2             |              |
|         | ADD GO,#1         |              |
|         | JMP L1            | ; 割込み待ち      |
| L2      | LD GO,#1          |              |
|         | OUT GO,BUZZ       | ; ブザー ON     |
|         | LD GO,#'a'        |              |
| L3      | CALL MS200        |              |
|         | CMP GO,KEY        |              |
|         | JZ L4             |              |
|         | ADD GO,#1         |              |
|         | JMP L3            |              |
| L4      | LD GO,#0          |              |
|         | OUT GO,BUZZ       | ; ブザー OFF    |
|         | JMP LOOP          |              |
| END     | HALT              |              |
| KEY     | DS 1              |              |
| IKEY    | DS 1              |              |
|         | ; SIO 受信割込み処理ルーチン |              |
| SIORINT | PUSHF             |              |
|         | PUSH GO           |              |
|         | IN GO,SIOD        | ; 入力した文字を    |
|         | ST GO,IKEY        | ; IKEY に格納する |
|         | POP GO            |              |
|         | POPF              |              |
|         | RETI              |              |

図 6.20 SIO 受信割込みの使用例

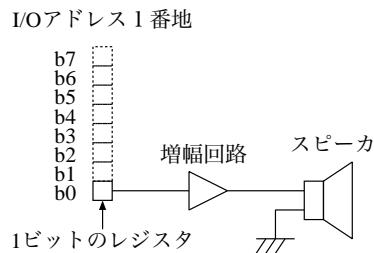


図 6.21 スピーカの回路

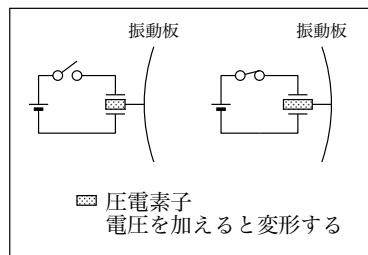


図 6.22 スピーカの回路

## 6.12 マシンステートとスピーカ

TeC には音を鳴らすためのスピーカが準備されています。スピーカには OUT 命令でアクセスします。ここではマシンステートを意識したプログラミングを行いスピーカを鳴らします。

### 6.12.1 スピーカーの仕組み

図 6.21 に示すように I/O アドレス 1 番地（「表 5.1 I/O マップ」参照）の LSB (b0) には 1 ビットのレジスタが存在します。このレジスタの 1/0 によりスピーカに加わる電圧が変化するようになっています。なお、図では省略しましたが「5.11.3 OUT (Output) 命令」で紹介した「ブザー音源」も同じスピーカに接続されています。

図 6.22 にスピーカの構造を示します。加えた電圧により振動板が前後に移動します。レジスタの値を 1, 0, 1, 0, 1, 0, 1, 0 と変化させることにより振動板が前後に振動して音がでます。1kHz の音を出したい時はプログラムで 1, 0 書き込みを 1 秒間に 1000 回 (1000 サイクル) 行う必要があります。

### 6.12.2 スピーカ駆動プログラム

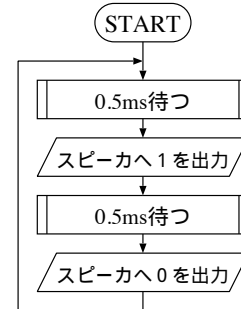
プログラムで 1 秒間に 1000 回のペースで 1, 0 を繰り返すためには、時間を計って 1/2000 秒に 1 度 I/O ポートの書き換えが行われるようにする必要があります。マ

### 例題 6-6 1kHz ブザー

**問題：** スピーカーから 1kHz の音を出すプログラムを作りなさい。

**考え方：** 「6.4.2 1ms タイマ」では、1ms のタイマを作りました。今回の問題では、半分の 0.5ms 毎にスピーカ出力の 0/1 を書き換えるようにします。

**フローチャート：** 次のようになります。



**プログラム：** 若干のステート数の増減は誤差と考慮して無視します。プログラムは、次のようになります。

| 番地 | 機械語   | ラベル   | ニーモニック         |
|----|-------|-------|----------------|
| 00 |       |       | ; 1kHz ブザー     |
| 01 |       | SPK   | EQU 01H        |
| 00 | 1F DC | START | LD SP, #0DCH   |
| 02 | B0 10 | LOOP  | CALL MS5       |
| 04 | 17 01 |       | LD G1, #01H    |
| 06 | C7 01 |       | OUT G1, SPK    |
| 08 | B0 10 |       | CALL MS5       |
| 0A | 17 00 |       | LD G1, #00H    |
| 0C | C7 01 |       | OUT G1, SPK    |
| 0E | A0 02 |       | JMP LOOP       |
| 10 |       |       | ; 0.5ms サブルーチン |
| 10 | D0    | MS5   | PUSH GO        |
| 11 | 13 57 |       | LD GO, #87     |
| 13 | 43 01 | MATU  | SUB GO, #1     |
| 15 | A4 19 |       | JZ KAERU       |
| 17 | A0 13 |       | JMP MATU       |
| 19 | D2    | KAERU | POP GO         |
| 1A | EC    |       | RET            |

シンステートを使用して時間を計り、一定の時間毎にスピーカポートの 0/1 を書き換えるプログラムを作ります。「例題 6-6 1kHz ブザー」のようなプログラムになります。書き換えの間隔によって音の高さが決まります。

### 問題

200Hz ブザーを作りなさい。

### 6.12.3 電子オルゴールプログラム

スピーカ出力プログラムの応用として電子オルゴールプログラムを紹介します。電子オルゴールプログラムでは曲の楽譜をデータとして準備します。データは、音の高さを決めるためのループの繰り返し回数と、音の長さを決めるためのループの繰り返し回数を組み合わせたものです。

#### フローチャート

図 6.23 にフローチャートを示します。プログラムは 3 重のループになっています。最も内側の二つのループが、スピーカに 1 を出力している時間と、0 を出力している時間を計っています。2 番目のループの繰り返しにより音の継続時間を制御します。最も外側のループは曲のデータを順番に取り出しています。

#### 音符データ

最も内側のループは 20 ステートあります。このループの繰り返し回数で時間を計りスピーカへの出力を音波の半サイクル毎に書き換えます。スピーカへの出力と音波の関係を図 6.24 に示します。

例えば「ラ」の音は  $440\text{Hz}$  ですので半サイクルの時間は  $2,457,600/440/2 = 2,792$  ステート になります。半サイクルの時間をループで測るのでループの繰り返し回数は、 $2,792/20 = 139 = 8B_{16}$  回 となります。この値が音の高さを決めます。

音の長さは、音の出力を何サイクル行うかにより決めます。例えば「ラ」の音は  $440\text{Hz}$  なので、 $220 = DC_{16}$  サイクルで 0.5 秒 になります。このプログラムでは、0.5 秒を 4 分音符一つの時間としました。

「ラ」以外の音も同様に対応するデータ値を決めました。表 6.2 に音データの一覧を示します。

#### プログラム

図 6.25 に完成したプログラムとデータを示します。プログラム中の ORG 命令は、プログラムのアドレスを指定する命令です。この場合、データを  $30\text{H}$  番地に生成するために使用しました。

このプログラムは、休符が表現できない、同じ高さの音が連続すると一つの音になってしまう等の問題があります。改良を考えて下さい。

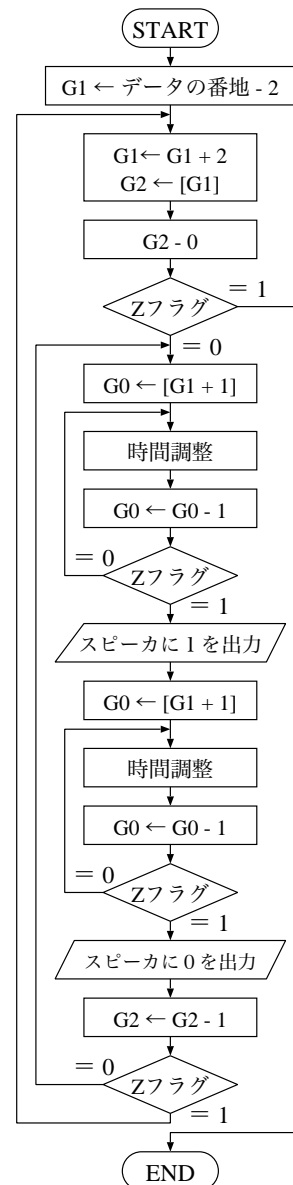


図 6.23 電子オルゴールのアルゴリズム

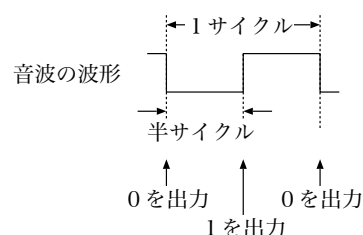


図 6.24 0/1 の出力と音波の関係

| 番地 | 機械語   | ラベル   | ニーモニック         |       |
|----|-------|-------|----------------|-------|
| 01 |       | SPK   | EQU 01H        |       |
| 00 |       |       | ; 電子オルゴール      |       |
| 00 | 17 2E |       | LD G1,#TABLE-2 |       |
| 02 | 37 02 | L9    | ADD G1,#2      |       |
| 04 | 19 00 |       | LD G2,0,G1     |       |
| 06 | 5B 00 |       | CMP G2,#0      |       |
| 08 | A4 2C |       | JZ L16         |       |
| 0A | 11 01 | L10   | LD G0,1,G1     | ; 7   |
| 0C | 00    | L11   | NO             | ; 3   |
| 0D | 00    |       | NO             | ; 3   |
| 0E | 43 01 |       | SUB G0,#1      | ; 5   |
| 10 | A4 14 |       | JZ L12         | ; 4/5 |
| 12 | A0 0C |       | JMP L11        | ; 5   |
| 14 | 13 01 | L12   | LD G0,#01H     | ; 5   |
| 16 | C3 01 |       | OUT G0,SPK     | ; 8   |
| 18 | 11 01 | L13   | LD G0,1,G1     | ; 7   |
| 1A | 00    | L14   | NO             | ; 3   |
| 1B | 00    |       | NO             | ; 3   |
| 1C | 43 01 |       | SUB G0,#1      | ; 5   |
| 1E | A4 22 |       | JZ L15         | ; 4/5 |
| 20 | A0 1A |       | JMP L14        | ; 5   |
| 22 | 13 00 | L15   | LD G0,#00H     | ; 5   |
| 24 | C3 01 |       | OUT G0,SPK     | ; 8   |
| 26 | 4B 01 |       | SUB G2,#1      | ; 5   |
| 28 | A4 02 |       | JZ L9          | ; 4/5 |
| 2A | A0 0A |       | JMP L10        | ; 5   |
| 2C | FF    | L16   | HALT           |       |
| 2D |       |       | ; ドレミの歌        |       |
| 30 |       | TABLE | ORG 30H        |       |
| 30 | C5 E8 |       | DC 0C5H,0E8H   | ; ド   |
| 32 | 49 D1 |       | DC 049H,0D1H   | ; レ   |
| 34 | F7 BA |       | DC 0F7H,0BAH   | ; ミ   |
| 36 | 41 E8 |       | DC 041H,0E8H   | ; ド   |
| 38 | A5 BA |       | DC 0A5H,0BAH   | ; ミ   |
| 3A | 83 E8 |       | DC 083H,0E8H   | ; ド   |
| 3C | A5 BA |       | DC 0A5H,0BAH   | ; ミ   |
| 3E | A5 BA |       | DC 0A5H,0BAH   | ; ミ   |
| 40 | DC D1 |       | DC 0DCH,0D1H   | ; レ   |
| 42 | 52 BA |       | DC 052H,0BAH   | ; ミ   |
| 44 | 57 AF |       | DC 057H,0AFH   | ; ファ  |
| 46 | 57 AF |       | DC 057H,0AFH   | ; ファ  |
| 48 | 52 BA |       | DC 052H,0BAH   | ; ミ   |
| 4A | 49 D1 |       | DC 049H,0D1H   | ; レ   |
| 4C | AF AF |       | DC 0AFH,0AFH   | ; ファ  |
| 4E | AF AF |       | DC 0AFH,0AFH   | ; ファ  |
| 50 | AF AF |       | DC 0AFH,0AFH   | ; ファ  |
| 52 | AF AF |       | DC 0AFH,0AFH   | ; ファ  |
| 54 | F7 BA |       | DC 0F7H,0BAH   | ; ミ   |
| 56 | 57 AF |       | DC 057H,0AFH   | ; ファ  |
| 58 | C4 9C |       | DC 0C4H,09CH   | ; ソ   |
| 5A | 62 9C |       | DC 062H,09CH   | ; ソ   |
| 5C | 52 BA |       | DC 052H,0BAH   | ; ミ   |
| 5E | C4 9C |       | DC 0C4H,09CH   | ; ソ   |
| 60 | A5 BA |       | DC 0A5H,0BAH   | ; ミ   |
| 62 | C4 9C |       | DC 0C4H,09CH   | ; ソ   |
| 64 | C4 9C |       | DC 0C4H,09CH   | ; ソ   |
| 66 | 00    |       | DC 000H        | ; 終了  |

表 6.2 音データ一覧

| 音   | 周波数 (Hz) | 長さ | 高さ |
|-----|----------|----|----|
| ド   | 262      | 83 | E8 |
| ド#  | 277      | 8B | DD |
| レ   | 294      | 93 | D1 |
| レ#  | 311      | 9C | C5 |
| ミ   | 330      | A5 | BA |
| ファ  | 349      | AF | AF |
| ファ# | 370      | B9 | A6 |
| ソ   | 392      | C4 | 9C |
| ソ#  | 415      | D0 | 93 |
| ラ   | 440      | DC | 8B |
| ラ#  | 466      | E9 | 83 |
| シ   | 494      | F7 | 7C |
| ド   | 523      | FF | 75 |

## 6.13 まとめ

この章では、教育用コンピュータ TeC の少し高度な機械語プログラミングを学びました。クロス開発環境を用いてアセンブラで機械語を作成することを前提にしました。

前の章で扱ったのは、ノイマン型コンピュータに最低限必要な機能でした。それに対し、この章では、スタック、サブルーチン、割込み等、ノイマン型コンピュータを実用的に使用するために必要な機能を扱いました。

また、マシンステートやジャンプテーブルの利用、数値の入出力等、プログラミングを行う上でよく使用されるテクニックも紹介しました。

図 6.25 電子オルゴールプログラム





## 付録 A

# TeC クロス開発環境

### A.1 始めに

TeC クロス開発環境は、教育用コンピュータ TeC (Tokuyama kousen Educational Computer) の機械語プログラムを開発するためのパソコン上で動作するプログラム群です。クロス開発環境は、クロスアセンブラ `tasm7`、ダウンロードプログラム `tsend7` からなります。これらを用いて TeC のプログラムをパソコン上でクロス開発する様子を図 A.1 に示します。

これらの入手とパソコンへのインストール方法は、<https://github.com/tctsigemura/Tasm> を参照して下さい。以下では、UNIX または macOS にインストールされた「TeC クロス開発環境」を使用する前提で明しています。

### A.2 アセンブルコマンド

クロスアセンブラは次の形式のコマンド行から起動され、ソースプログラムを入力し、アセンブルリストと機械語を出力します。アセンブルリストは拡張子 `.lst` のファイルに、機械語は拡張子 `.bin` のファイルに格納されます。

```
tasm7 [-l nn] [-w nn] source_file
```

`source_file` は A.4 章で説明する文法で記述されたソースファイルの名前です。ソースファイル名は、拡張子 `“.t7”` でなければなりません。アセンブルリストの整形に関するコマンド行オプションが使用できます。意味は表 A.1 の通りです。

### A.3 ダウンロードプログラム

ダウンロードプログラムは `tasm7` が出力した機械語プログラムをシリアル通信を用いて TeC に転送します。

表 A.1 コマンド行オプション

| オプション              | 意味                                |
|--------------------|-----------------------------------|
| <code>-l nn</code> | リストの 1 ページを <code>nn</code> 行にする。 |
| <code>-w nn</code> | リストの 1 ページを <code>nn</code> 桁にする。 |

次の形式のコマンド行から起動します。

```
tsend7 bin_file [com_port]
```

`bin_file` は `tasm7` が出力した機械語プログラムファイルの名前です。`[com_port]` は、TeC と接続されたシリアルポートを表すデバイスファイルの名前です。`tsend7` を起動すると画面に「TeC を受信状態にする」ように指示が表示されるので、TeC の E0H 番地に格納されている IPL プログラム（「A.8 IPL プログラム」参照）を起動します。

### A.4 アセンブラの文法

以下では `tasm7` の入力となるソースプログラムの文法について説明します。

#### A.4.1 行フォーマット

ソースプログラムは「行」の連なりからなるテキストです。ソースプログラムはソースファイルと呼ばれるテキストファイルに格納されます。

`tasm7` のソースプログラムでは、文字定数と文字列中を除き、英字大文字と小文字は区別されません。空白はトークンの前後にいくつでも挿入できます。但し、行頭の空白だけは「ラベルが存在しない」ことを表す特別なものになります。空白にはスペース（文字コード 20H）とタブ（文字コード 09H）の 2 種類があります。以下に「行」の書式を示します。

```
[ラベル] [命令とオペランド] [:コメント]
```

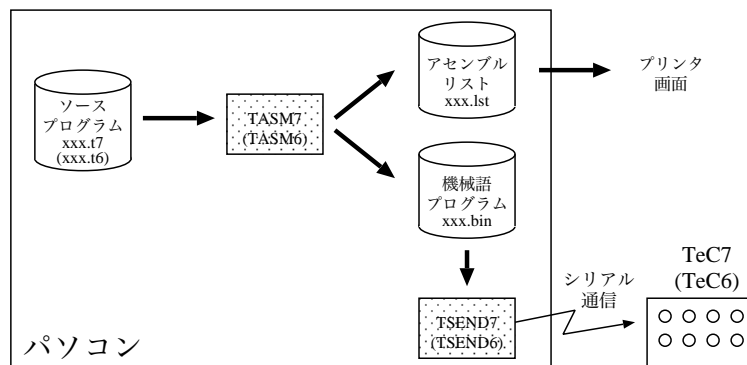


図 A.1 処理の流れ

| ラベル   | 命令  | オペランド      |
|-------|-----|------------|
| START | EQU | 03H        |
| SIZE  | EQU | 100        |
| END   | EQU | START+SIZE |

図 A.2 EQU 命令の使用例

| ラベル   | 命令  | オペランド |
|-------|-----|-------|
| START | ORG | 03H   |

図 A.3 ORG 命令の使用例

| ラベル  | 命令 | オペランド |
|------|----|-------|
| WORK | DS | 10    |

図 A.4 DS 命令の使用例

#### A.4.2 ラベル

行が英字で始まる場合、その行にラベルがあるとみなされます。ラベルは英字で始まり、その後に任意長の英数字が続く文字列です。ラベルが無い行は空白文字で始めます。ラベルの長さに制限はありませんが、アセンブルリストの見栄えから 7 文字以内を推奨します。

行にラベルを書くとラベルが定義されます。EQU 命令を除き、その行のアドレスがラベルの値になります。

#### A.4.3 疑似命令

以下の 4 種類の疑似命令が使用できます。

##### EQU 命令

任意の値を持つラベルを定義する疑似命令です。この命令にはラベルが必ず必要です。命令行のラベルがオペランド式の値で定義されます。式ではラベルの前方参照はできません。EQU 命令の書式を以下に、使用例を図 A.2 に示します。

ラベル EQU 式

##### ORG 命令

機械語を生成するアドレスを変更する疑似命令です。機械語（データも含む）が生成される全ての行より前で使用された場合は、単に次の命令のアドレスを変更しま

す。機械語の生成が開始された後に使用された場合は、指定アドレスに達するまでゼロで初期化された領域を生成します。

現在のアドレスより若いアドレス（逆戻りするアドレス）を指定することはできません。ORG 命令の式ではラベルの前方参照は禁止です。ORG 命令の書式を以下に、使用例を図 A.3 に示します。

[ラベル] ORG 式

##### DS 命令

領域を定義するために使用する疑似命令です。領域の値はゼロで初期化されます。DS 命令の式では、ラベルの前方参照はできません。DS 命令の書式を以下に、使用例を図 A.4 に示します。

[ラベル] DS 式

##### DC 命令

メモリ上に任意の定数を出力するために使用する疑似命令です。オペランドに任意個の式または文字列を書

| ラベル  | 命令 | オペランド   |
|------|----|---------|
| DATA | DC | 1,2,3,4 |
|      | DC | 1+1     |
| CHA  | DC | 'a'     |
| STR  | DC | "abc"   |

図 A.5 DC 命令の使用例

表 A.2 機械語命令 1 に属す命令

| 命令    | 意味           |
|-------|--------------|
| NO    | 何もしない        |
| PUSHF | フラグをスタックに退避  |
| POPF  | フラグをスタックから復旧 |
| EI    | 割り込み許可       |
| DI    | 割り込み禁止       |
| RET   | サブルーチンから復帰   |
| RETI  | 割り込みから復帰     |
| HALT  | プログラム終了      |

| ラベル   | 命令   | オペランド |
|-------|------|-------|
| START | NO   |       |
|       | NO   |       |
| END   | HALT |       |

図 A.6 機械語命令 1 の使用例

くことができます。DC 命令の書式を以下に、使用例を図 A.5 に示します。

$$[\text{ラベル}] \text{ DC } \left\{ \begin{array}{l} \text{式} \\ \text{文字列} \end{array} \right\} [ \left\{ \begin{array}{l} \text{式} \\ \text{文字列} \end{array} \right\} \dots ]$$

#### A.4.4 機械語命令

機械語命令は 6 つのグループに分類できます。以下では 6 つのグループの命令について、書き方を説明します。

##### 機械語命令 1

オペランドの無い 1 バイトの機械語命令です。表 A.2 の命令がこのグループに属します。このグループの命令の書式を以下に、使用例を図 A.6 に示します。

[ラベル] 命令

表 A.3 機械語命令 2 に属す命令

| 命令   | 意味            |
|------|---------------|
| SHLA | 左算術シフト        |
| SHLL | 左論理シフト        |
| SHRA | 右算術シフト        |
| SHRL | 右論理シフト        |
| PUSH | スタックにレジスタを退避  |
| POP  | スタックからレジスタを復旧 |

| ラベル | 命令   | オペランド |
|-----|------|-------|
| L1  | SHRA | SP    |
|     | PUSH | G0    |
|     | POP  | G1    |

図 A.7 機械語命令 2 の使用例

表 A.4 機械語命令 3 に属す命令

| 命令  | 意味           |
|-----|--------------|
| IN  | 入出力ポートから読み込む |
| OUT | 入出力ポートに書き込む  |

#### 機械語命令 2

レジスタ指定付きの 1 バイトの機械語命令です。レジスタとして G0, G1, G2, SP が使用できます。表 A.3 の命令がこのグループに属します。このグループの命令の書式を以下に、使用例を図 A.7 に示します。

[ラベル] 命令 レジスタ

#### 機械語命令 3

オペランドにレジスタと式を書く 2 バイトの機械語命令です。レジスタとして G0, G1, G2, SP が使用できます。表 A.4 の命令がこのグループに属します。このグループの命令の書式を以下に、使用例を図 A.8 に示します。

[ラベル] 命令 レジスタ, 式

#### 機械語命令 4

レジスタとメモリをオペランドに指定できる命令で、全てのアドレッシング・モードが使用できるグループです。レジスタとして G0, G1, G2, SP が、インデクス

| ラベル | 命令  | オペランド  |
|-----|-----|--------|
| SIO | EQU | 03H    |
|     | IN  | G0,SIO |

図 A.8 機械語命令 3 の使用例

表 A.5 機械語命令 4 に属す命令

| 命令  | 意味                |
|-----|-------------------|
| LD  | レジスタにメモリの値をロード    |
| ADD | レジスタとメモリの値の和      |
| SUB | レジスタとメモリの値の差      |
| CMP | レジスタとメモリの値を比較     |
| AND | レジスタとメモリの値の論理積    |
| OR  | レジスタとメモリの値の論理和    |
| XOR | レジスタとメモリの値の排他的論理和 |

| ラベル | 命令  | オペランド      |
|-----|-----|------------|
| L1  | LD  | G0,DATA    |
|     | LD  | G1,#0      |
|     | SUB | G0,DATA,G1 |
|     | ADD | G1,#1      |

図 A.9 機械語命令 4 の使用例

レジスタとして G1, G2 が使用できます。表 A.5 の命令がこのグループに属します。このグループの命令の書式を以下に、使用例を図 A.9 に示します。

オペランドの書式はアドレッシング・モードにより変化します。1 行目がダイレクトモード、2 行目がインデクスモード、3 行目がイミディエイトモードです。

[ラベル] 命令  $\left\{ \begin{array}{l} \text{レジスタ, 式} \\ \text{レジスタ, 式, インデクスレジスタ} \\ \text{レジスタ, \#式} \end{array} \right\}$

#### 機械語命令 5

機械語命令 4 と、ほぼ、同様ですが、イミディエイトモードが使用できない機械語命令のグループです。レジスタとして G0, G1, G2, SP が、インデクスレジスタとして G1, G2 が使用できます。表 A.6 のように ST 命令だけがこのグループに属します。このグループの命令の書式を以下に、使用例を図 A.10 に示します。

[ラベル] 命令  $\left\{ \begin{array}{l} \text{レジスタ, 式} \\ \text{レジスタ, 式, インデクスレジスタ} \end{array} \right\}$

表 A.6 機械語命令 5 に属す命令

| 命令 | 意味                 |
|----|--------------------|
| ST | レジスタの値をメモリオペランドに格納 |

| ラベル | 命令 | オペランド      |
|-----|----|------------|
|     | ST | G0,WORK    |
|     | ST | G0,WORK,G1 |

図 A.10 機械語命令 5 の使用例

表 A.7 機械語命令 6 に属す命令

| 命令   | 意味               |
|------|------------------|
| JMP  | 必ずジャンプ           |
| JZ   | Z フラグが'1' ならジャンプ |
| JC   | C フラグに'1' ならジャンプ |
| JM   | S フラグに'1' ならジャンプ |
| JNZ  | Z フラグが'0' ならジャンプ |
| JNC  | C フラグに'0' ならジャンプ |
| JNM  | S フラグに'0' ならジャンプ |
| CALL | サブルーチン呼び出し       |

| ラベル | 命令   | オペランド |
|-----|------|-------|
|     | JMP  | LOOP  |
|     | CALL | SUB1  |

図 A.11 機械語命令 6 の使用例

#### 機械語命令 6

メモリオペランドだけの機械語命令グループです。表 A.7 に示すジャンプ・コール命令がこのグループに属します。インデクスレジスタとして G1, G2 が使用できます。このグループの命令の書式を以下に、使用例を図 A.11 に示す。

[ラベル] 命令  $\left\{ \begin{array}{l} \text{式} \\ \text{式, インデクスレジスタ} \end{array} \right\}$

### A.5 アセンブラの文法まとめ

次のページにアセンブラの文法を BNF 風にまとめます。

|         |     |                                                                                                              |
|---------|-----|--------------------------------------------------------------------------------------------------------------|
| <プログラム> | ::= | { <行> } EOF                                                                                                  |
| <行>     | ::= | <命令行>   <空行>   <コメント行>                                                                                       |
| <命令行>   | ::= | <ラベル欄> [<命令欄>] [<コメント>] <改行>                                                                                 |
| <空行>    | ::= | <改行>                                                                                                         |
| <コメント行> | ::= | <コメント> <改行>                                                                                                  |
| <ラベル欄>  | ::= | <ラベル>   <空白>                                                                                                 |
| <命令欄>   | ::= | <命令記述1>   <命令記述2>   <命令記述3>   <命令記述4><br>  <命令記述5>   <命令記述6>   <命令記述7>   <命令記述8>                             |
| <命令記述1> | ::= | <命令1>                                                                                                        |
| <命令記述2> | ::= | <命令2> <レジスタ>                                                                                                 |
| <命令記述3> | ::= | <命令3> <レジスタ>, <式>                                                                                            |
| <命令記述4> | ::= | <命令4> <レジスタ>, <アドレス1>                                                                                        |
| <命令記述5> | ::= | <命令5> <レジスタ>, <アドレス2>                                                                                        |
| <命令記述6> | ::= | <命令6> <アドレス2>                                                                                                |
| <命令記述7> | ::= | <命令7> <式>                                                                                                    |
| <命令記述8> | ::= | <命令8> <値列>                                                                                                   |
| <アドレス1> | ::= | # <式>   <式> [, <インデクス>]                                                                                      |
| <アドレス2> | ::= | <式> [, <インデクス>]                                                                                              |
| <値列>    | ::= | <拡張式> {, <拡張式> }                                                                                             |
| <拡張式>   | ::= | <式>   <文字列>                                                                                                  |
| <式>     | ::= | <項>   <項> + <式>   <項> - <式>                                                                                  |
| <項>     | ::= | <因子1>   <因子1> * <項>   <因子1> / <項>                                                                            |
| <因子1>   | ::= | <因子>   +<因子>   -<因子>                                                                                         |
| <因子>    | ::= | <数値>   <ラベル>   ( <式> )                                                                                       |
| <コメント>  | ::= | ; { <文字1> }                                                                                                  |
| <数値>    | ::= | <10進数>   <16進数>   <文字定数>                                                                                     |
| <10進数>  | ::= | <数字> { <数字> }                                                                                                |
| <16進数>  | ::= | <数字> { <16進数字> } H                                                                                           |
| <文字定数>  | ::= | ' <文字2> '                                                                                                    |
| <文字列>   | ::= | " { <文字3> } "                                                                                                |
| <ラベル>   | ::= | <英字> { <英数字> }                                                                                               |
| <英数字>   | ::= | <英字>   <数字>                                                                                                  |
| <文字1>   | ::= | <改行>以外の文字                                                                                                    |
| <文字2>   | ::= | <改行>, ' 以外の文字                                                                                                |
| <文字3>   | ::= | <改行>," 以外の文字                                                                                                 |
| <16進数字> | ::= | 0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F                                                |
| <英字>    | ::= | A   B   C   D   E   F   G   H   I   J   K   L   M   N   O<br>  P   Q   R   S   T   U   V   W   X   Y   Z   _ |
| <数字>    | ::= | 0   1   2   3   4   5   6   7   8   9                                                                        |
| <命令1>   | ::= | NO   PUSHF   POPF   EI   DI   RET   RETI   HALT                                                              |
| <命令2>   | ::= | SHLA   SHLL   SHRA   SHRL   PUSH   POP                                                                       |
| <命令3>   | ::= | IN   OUT                                                                                                     |
| <命令4>   | ::= | LD   ADD   SUB   CMP   AND   OR   XOR                                                                        |
| <命令5>   | ::= | ST                                                                                                           |
| <命令6>   | ::= | JMP   JZ   JC   JM   JNZ   JNC   JNM   CALL                                                                  |
| <命令7>   | ::= | EQU   DS   ORG                                                                                               |
| <命令8>   | ::= | DC                                                                                                           |
| <インデクス> | ::= | G1   G2                                                                                                      |
| <レジスタ>  | ::= | G0   G1   G2   SP                                                                                            |
| <改行>    | ::= | '\n'                                                                                                         |
| <空白>    | ::= | ' '   '\t'                                                                                                   |

## A.6 プログラム例

以下にローマ字で名前を SIO に出力するプログラムの例を示します。

| ADR | CODE        | Label | Instruction | Comment    | Page(1) |
|-----|-------------|-------|-------------|------------|---------|
| 02  |             | SIOD  | EQU         | 02H        |         |
| 03  |             | SIOS  | EQU         | 03H        |         |
| 00  |             |       |             |            |         |
| 00  | 17 00       | START | LD          | G1,#0      |         |
| 02  | 19 15       | L0    | LD          | G2,DATA,G1 |         |
| 04  | 5B 00       |       | CMP         | G2,#0      |         |
| 06  | A4 14       |       | JZ          | END        |         |
| 08  | C0 03       | L1    | IN          | G0,SIOS    |         |
| 0A  | 63 80       |       | AND         | G0,#80H    |         |
| 0C  | A4 08       |       | JZ          | L1         |         |
| 0E  | CB 02       |       | OUT         | G2,SIOD    |         |
| 10  | 37 01       |       | ADD         | G1,#1      |         |
| 12  | A0 02       |       | JMP         | L0         |         |
| 14  | FF          | END   | HALT        |            |         |
| 15  |             |       |             |            |         |
| 15  | 73 69 67 65 | DATA  | DC          | "sigemura" |         |
| 19  | 6D 75 72 61 |       |             |            |         |
| 1D  | 0D 0A       |       | DC          | 0DH,0AH    |         |
| 1F  | 00          |       | DC          | 0          | ; 終わりの印 |

## A.7 アセンブル実行例

TeC7 を使いローマ字で名前を SIO に出力するプログラムをアセンブル・実行した例を示します。

```
$ tasm7 sigemura.t7 <---- アセンブル
```

**アセンブル成功**

結果は [sigemura.lst] と [sigemura.bin] に格納しました。

... 省略...

```
$ tsend7 sigemura.bin <---- ダウンロード
```

TeC7 を受信状態にして Enter キーを押して下さい。

```
[00] [20] [17] [00] [19] [15] [5b] [00] [a4] [14] [c0] [03] [63] [80] [a4] [08] [cb] [02] [37]
```

```
[01] [a0] [02] [ff] [73] [69] [67] [65] [6d] [75] [72] [61] [0d] [0a] [00]
```

```
$ screen /dev/tty.usbserial-xxxxxxx <---- 通信ソフト起動
```

```
sigemura <---- 実行結果
```

```
^A^\ <---- 通信ソフトの終了
```

```
[EOT]
```

## A.8 IPL プログラム

TeC の ROM に格納されている IPL プログラムのアセンブルリストを参考のため以下に示します。TeC のメモリ空間は E0H~FFH 番地が ROM になっており、以下の IPL プログラムが予め格納されています。IPL プログラムは、パソコン上で tasm7 を使用してクロス開発された機械語プログラムをシリアル入出力から受信してメモリに格納します。

| ADR | CODE  | Label | Instruction | Comment          | Page(1) |
|-----|-------|-------|-------------|------------------|---------|
| 02  |       | SIOD  | EQU 02H     |                  |         |
| 03  |       | SIOS  | EQU 03H     |                  |         |
| E0  |       |       | ORG 0E0H    |                  |         |
| E0  | 1F DC | IPL   | LD SP,#0DCH | ; 割込みベクタの直前がスタック |         |
| E2  | B0 F6 |       | CALL READ   | ; ロードアドレスを入力     |         |
| E4  | D0    |       | PUSH G0     |                  |         |
| E5  | D6    |       | POP G1      |                  |         |
| E6  | B0 F6 |       | CALL READ   | ; 長さを入力          |         |
| E8  | D0    |       | PUSH G0     |                  |         |
| E9  | DA    |       | POP G2      |                  |         |
| EA  | A4 FF | LOOP  | JZ STOP     |                  |         |
| EC  | B0 F6 |       | CALL READ   |                  |         |
| EE  | 21 00 |       | ST G0,0,G1  |                  |         |
| F0  | 37 01 |       | ADD G1,#1   |                  |         |
| F2  | 4B 01 |       | SUB G2,#1   |                  |         |
| F4  | A0 EA |       | JMP LOOP    |                  |         |
| F6  |       |       |             |                  |         |
| F6  | C0 03 | READ  | IN G0,SIOS  |                  |         |
| F8  | 63 40 |       | AND G0,#40H |                  |         |
| FA  | A4 F6 |       | JZ READ     |                  |         |
| FC  | C0 02 |       | IN G0,SIOD  |                  |         |
| FE  | EC    |       | RET         |                  |         |
| FF  |       |       |             |                  |         |
| FF  | FF    | STOP  | HALT        | ; PC がゼロになって止まる  |         |

## A.9 機械語プログラムファイル形式

tasm7 が出力する機械語プログラムの形式は次の通りです.

|                 |
|-----------------|
| ロードアドレス (1 バイト) |
| プログラム長 (1 バイト)  |
| ...             |
| 機械語 (1 バイト以上)   |
| ...             |



## 付録 B

# 命令表

図 B.1 に TeC7 の機械語命令表を掲載します。命令表の内容はケース蓋の裏に張り付けてあるものと同じ内容です。

## TEC7命令表

Ver. 3.0 (2012.4.6)

※ステート数：イミディエイト／ダイレクト／インデクスド  
(ジャンプ命令では、条件不成立／ダイレクト／インデクスド)

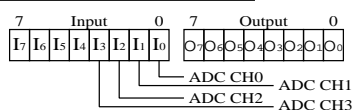
| ニーモニック | 命令                     | 第1バイト |       | 第2バイト     | フラグ変化 | ステート数※ | 動作                 |
|--------|------------------------|-------|-------|-----------|-------|--------|--------------------|
|        |                        | OP    | GRXR  |           |       |        |                    |
| NO     | No Opration            | 0000  | 00 00 | ————      | ×     | 3      | 何もしない              |
| LD     | Load                   | 0001  | GR XR | aaaa aaaa | ×     | 5/7/7  | GR ← [EA]          |
| ST     | Store                  | 0010  | GR XR | aaaa aaaa | ×     | -/7/7  | [EA] ← GR          |
| ADD    | Add                    | 0011  | GR XR | aaaa aaaa | ○     | 5/7/7  | GR ← GR + [EA]     |
| SUB    | Subtract               | 0100  | GR XR | aaaa aaaa | ○     | 5/7/7  | GR ← GR - [EA]     |
| CMP    | Compare                | 0101  | GR XR | aaaa aaaa | ○     | 5/7/7  | GR - [EA]          |
| AND    | Logical And            | 0110  | GR XR | aaaa aaaa | ○     | 5/7/7  | GR ← GR & [EA]     |
| OR     | Logical Or             | 0111  | GR XR | aaaa aaaa | ○     | 5/7/7  | GR ← GR   [EA]     |
| XOR    | Logical Xor            | 1000  | GR XR | aaaa aaaa | ○     | 5/7/7  | GR ← GR ^ [EA]     |
| SHLA   | Shift Left Arithmetic  | 1001  | GR 00 | ————      | ○     | 4      | GR ← GR << 1       |
| SHLL   | Shift Left Logical     | 1001  | GR 01 | ————      | ○     | 4      | GR ← GR << 1       |
| SHRA   | Shift Right Arithmetic | 1001  | GR 10 | ————      | ○     | 4      | GR ← GR >> 1       |
| SHRL   | Shift Right Logical    | 1001  | GR 11 | ————      | ○     | 4      | GR ← GR >> 1       |
| JMP    | Jump                   | 1010  | 00 XR | aaaa aaaa | ×     | -/5/6  | PC ← EA            |
| JZ     | Jump on Zero           | 1010  | 01 XR | aaaa aaaa | ×     | 4/5/6  | if Zero PC ← EA    |
| JC     | Jump on Carry          | 1010  | 10 XR | aaaa aaaa | ×     | 4/5/6  | if Carry PC ← EA   |
| JM     | Jump on Minus          | 1010  | 11 XR | aaaa aaaa | ×     | 4/5/6  | if Sign PC ← EA    |
| CALL   | Call subroutine        | 1011  | 00 XR | aaaa aaaa | ×     | -/6/7  | [--SP]←-PC, PC←-EA |
| JNZ    | Jump on Not Zero       | 1011  | 01 XR | aaaa aaaa | ×     | 4/5/6  | if !Zero PC ← EA   |
| JNC    | Jump on Not Carry      | 1011  | 10 XR | aaaa aaaa | ×     | 4/5/6  | if !Carry PC ← EA  |
| JNM    | Jump on Not Minus      | 1011  | 11 XR | aaaa aaaa | ×     | 4/5/6  | if !Sign PC ← EA   |
| IN     | Input                  | 1100  | GR 00 | 0000 pppp | ×     | 8      | GR ← IO[P]         |
| OUT    | Output                 | 1100  | GR 11 | 0000 pppp | ×     | 8      | IO[P] ← GR         |
| PUSH   | Push Register          | 1101  | GR 00 | ————      | ×     | 6      | [--SP] ← GR        |
| PUSHF  | Push Flag              | 1101  | 11 01 | ————      | ×     | 6      | [--SP] ← FLAG      |
| POP    | Pop Register           | 1101  | GR 10 | ————      | ×     | 6      | GR ← [SP++]        |
| POPF   | Pop Flag               | 1101  | 11 11 | ————      | ○     | 6      | FLAG ← [SP++]      |
| EI     | Enable Interrupt       | 1110  | 00 00 | ————      | ×     | 4      | 割り込み許可             |
| DI     | Disable Interrupt      | 1110  | 00 11 | ————      | ×     | 4      | 割り込み禁止             |
| RET    | Return from subroutine | 1110  | 11 00 | ————      | ×     | 6      | PC ← [SP++]        |
| RETI   | Return from Interrupt  | 1110  | 11 11 | ————      | ×     | 6      | PC ← [SP++], STI   |
| HALT   | Halt                   | 1111  | 11 11 | ————      | ×     | 4      | 停止                 |

| GR | 意味 |
|----|----|
| 00 | G0 |
| 01 | G1 |
| 10 | G2 |
| 11 | SP |

| XR | 意味          |
|----|-------------|
| 00 | ダイレクトモード    |
| 01 | G1インデクスドモード |
| 10 | G2インデクスドモード |
| 11 | イミディエイトモード  |

| メモリマップ        |                 |
|---------------|-----------------|
| Addr          | 内容              |
| 00<br> <br>DB | RAM             |
| DC            | Tmr 割り込みベクタ     |
| DD            | SIO 受信割り込みベクタ   |
| DE            | SIO 送信割り込みベクタ   |
| DF            | Console 割り込みベクタ |
| E0<br> <br>FF | ROM(IPL)        |

| I/Oマップ    |                                                    |
|-----------|----------------------------------------------------|
| Addr      | Read/Write                                         |
| 0         | Data-Sw/b0:Beep                                    |
| 1         | Data-Sw/b0:Speaker                                 |
| 2         | SIO-Data/SIO-Data                                  |
| 3         | b7:Tx Ready / b7:Tx STI<br>b6:Rx Ready / b6:Rx STI |
| 4         | TMR現在値/TMR周期                                       |
| 5         | b7:TMR Poll / b7:TMR STI<br>b0:TMR Enb             |
| 6         | 空き/b0:Console STI                                  |
| 7         | Input/Output                                       |
| 8         | ADC CH0/空き                                         |
| 9         | ADC CH1/空き                                         |
| A         | ADC CH2/空き                                         |
| B         | ADC CH3/空き                                         |
| ....<br>F | 空き/空き                                              |



PIO : Parallel Input Output  
SIO : Serial Input Output  
TMR : Timer  
TMR周期 : 75=1s  
STI : Set Interrupt

図 B.1 TeC7 命令表

## 付録 C

# 参考資料

TeC をもっと活用してもらうために、拡張ボード、TeC の回路図、FPGA のピン配置表等を掲載します。次のページから以下の資料を掲載します。

### C.1 TeC7 基板回路図

図 C.1 に TeC7 本体のプリント基板回路図を示します。

### C.2 TeC7 ピン配置表

図 C.2 に TeC7 に搭載されている Xilinx Spartan-6 FPGA (XC6LX9-2TQG144C) のピンがどのように利用されているかを示します。

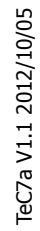


图 C.1 TeC7a 基板回路图

TeC7a Spartan-6 ピン配置

VI.1 2012/10/05

| ピン番号 | VHDL名称     | VHDL属性        | 注釈          | ピン番号 | VHDL名称      | VHDL属性 | 注釈            | ピン番号 | VHDL名称     | VHDL属性        | 注釈          |
|------|------------|---------------|-------------|------|-------------|--------|---------------|------|------------|---------------|-------------|
| 1    | ADC_REF(7) | OUT           | ADC ラダー MSB | 73   |             |        | NC            | 109  |            |               | TCK         |
| 2    | ADC_REF(6) | OUT           | ADC ラダー     | 74   |             |        | NC            | 110  |            |               | TDI         |
| 3    |            |               | GND         | 75   |             |        | NC            | 111  | SIO_RXD    | IN            | SIO 受信データ   |
| 4    |            |               | VCCO        | 76   |             |        | VCCO          | 112  | PS2_DAT    | INOUT(PULLUP) | キーボードデータ    |
| 5    | ADC_REF(5) | OUT           | ADC ラダー     | 77   |             |        | GND           | 113  |            |               | GND         |
| 6    | ADC_REF(4) | OUT           | ADC ラダー     | 78   | DATA_LED(4) | OUT    | D4 ランプ        | 114  | PS2_CLK    | INOUT         | キーボードクロック   |
| 7    | ADC_REF(3) | OUT           | ADC ラダー     | 79   | DATA_LED(5) | OUT    | D5 ランプ        | 115  | VGA_VSYNC  | OUT           | ディスプレイ垂直同期  |
| 8    | ADC_REF(2) | OUT           | ADC ラダー     | 80   | DATA_LED(6) | OUT    | D6 ランプ        | 116  | VGA_HSYNC  | OUT           | ディスプレイ水平同期  |
| 9    | ADC_REF(1) | OUT           | ADC ラダー     | 81   | DATA_LED(7) | OUT    | D7 ランプ        | 117  | VGA_BLUE   | OUT           | ディスプレイ青     |
| 10   | ADC_REF(0) | OUT           | ADC ラダー LSB | 82   | SPK_OUT     | OUT    | スピーカ          | 118  | VGA_GREEN  | OUT           | ディスプレイ緑     |
| 11   | JP_IN(0)   | INOUT(PULLUP) | ジャンパー上段     | 83   | ADDR_LED(0) | OUT    | A0 ランプ        | 119  | VGA_RED    | OUT           | ディスプレイ赤     |
| 12   | JP_IN(1)   | INOUT(PULLUP) | ジャンパー下段     | 84   | ADDR_LED(1) | OUT    | A1 ランプ        | 120  | RESET_SW   | OUT           | RESET スイッチ  |
| 13   |            |               | GND         | 85   | ADDR_LED(2) | OUT    | A2 ランプ        | 121  | EXT_OUT(7) | OUT           | 拡張ポート4番ピン   |
| 14   | BREAK_SW   | IN(PULLUP)    | BREAK スイッチ  | 86   |             |        | VCCO          | 122  |            |               | VCCO        |
| 15   | STEP_SW    | IN(PULLUP)    | STEP スイッチ   | 87   | ADDR_LED(3) | OUT    | A3 ランプ        | 123  | EXT_OUT(6) | OUT           | 拡張ポート6番ピン   |
| 16   | RUN_SW     | IN(PULLUP)    | RUN スイッチ    | 88   | ADDR_LED(4) | OUT    | A4 ランプ        | 124  | EXT_OUT(5) | OUT           | 拡張ポート8番ピン   |
| 17   | STOP_SW    | IN(PULLUP)    | STOP スイッチ   | 89   |             |        | VCCINT        | 125  |            |               | VCCO        |
| 18   |            |               | VCCO        | 90   |             |        | VCCAUX        | 126  | EXT_OUT(4) | OUT           | 拡張ポート10番ピン  |
| 19   |            |               | VCCINT      | 91   |             |        | GND           | 127  | EXT_OUT(3) | OUT           | 拡張ポート12番ピン  |
| 20   |            |               | VCCAUX      | 92   | ADDR_LED(5) | OUT    | A5 ランプ        | 128  |            |               | VCCINT      |
| 21   | SETA_SW    | IN(PULLUP)    | SETA スイッチ   | 93   | ADDR_LED(6) | OUT    | A6 ランプ        | 129  |            |               | VCCAUX      |
| 22   | INCA_SW    | IN(PULLUP)    | INCA スイッチ   | 94   | ADDR_LED(7) | OUT    | A7 ランプ        | 130  |            |               | GND         |
| 23   | DECA_SW    | IN(PULLUP)    | DECA スイッチ   | 95   |             |        | // SD 挿入検出    | 131  | EXT_OUT(2) | OUT           | 拡張ポート14番ピン  |
| 24   | WRITE_SW   | IN(PULLUP)    | WRITE スイッチ  | 96   |             |        | GND           | 132  | EXT_OUT(1) | OUT           | 拡張ポート16番ピン  |
| 25   |            |               | GND         | 97   |             |        | // SD DAT1    | 133  | EXT_OUT(0) | OUT           | 拡張ポート18番ピン  |
| 26   | DATA_SW(7) | IN(PULLUP)    | D7 スイッチ     | 98   | SPI_DIN     | IN     | SPI データ入力     | 134  | EXT_IN(7)  | IN(PULLUP)    | 拡張ポート3番ピン   |
| 27   | DATA_SW(6) | IN(PULLUP)    | D6 スイッチ     | 99   | SPI_SCLK    | OUT    | SPI クロック出力    | 135  |            |               | VCCO        |
| 28   |            |               | VCCINT      | 100  | SPI_DOUT    | OUT    | SPI データ出力     | 136  |            |               | GND         |
| 29   | DATA_SW(5) | IN(PULLUP)    | D5 スイッチ     | 101  | SPI_CS      | OUT    | SPI セレクト出力    | 137  | EXT_IN(6)  | IN(PULLUP)    | 拡張ポート5番ピン   |
| 30   | DATA_SW(4) | IN(PULLUP)    | D4 スイッチ     | 102  |             |        | // SD DAT2    | 138  | EXT_IN(5)  | IN(PULLUP)    | 拡張ポート7番ピン   |
| 31   |            |               | VCCO        | 103  |             |        | VCCO          | 139  | EXT_IN(4)  | IN(PULLUP)    | 拡張ポート9番ピン   |
| 32   | DATA_SW(3) | IN(PULLUP)    | D3 スイッチ     | 104  | ACQ_LED     | OUT    | // SD アクセスランプ | 140  | EXT_IN(3)  | IN(PULLUP)    | コンパレータ OUT1 |
| 33   | DATA_SW(2) | IN(PULLUP)    | D2 スイッチ     | 105  | SIO_TXD     | OUT    | SIO 送信データ     | 141  | EXT_IN(2)  | IN(PULLUP)    | コンパレータ OUT2 |
| 34   | DATA_SW(1) | IN(PULLUP)    | D1 スイッチ     | 106  |             |        | TDO           | 142  | EXT_IN(1)  | IN(PULLUP)    | コンパレータ OUT3 |
| 35   | DATA_SW(0) | IN(PULLUP)    | D0 スイッチ     | 107  |             |        | TMS           | 143  | EXT_IN(0)  | IN(PULLUP)    | コンパレータ OUT4 |
| 36   |            |               | VCCAUX      | 108  |             |        | GND           | 144  |            |               | HSGND       |

専用ピン、電源ピン、未配線ピン

図 C.2 TeC7a ピン配置



**TeC 教科書 Ver. 4.0.0**

発行年月 2017年 3月 Ver.4.0.0  
発 行 独立行政法人国立高等専門学校機構  
徳山工業高等専門学校  
情報電子工学科 重村哲至  
〒745-8585 山口県周南市学園台  
sigemura@tokuyama.ac.jp