

解答冊子

- [解答冊子](#)
 - [1章](#)
 - [1.1.2 CPUとは](#)
 - [1.1.3 メモリとは](#)
 - [1.2.2 ビットとデータ表現](#)
 - [1.2.3 2進数と10進数、16進数](#)
 - [1.2.4 2進数と負の数（二の補数表現）](#)
 - [2章](#)
 - [2.2.1.0](#)
 - [2.2.1.1](#)
 - [2.2.1.2](#)
 - [2.2.2.3](#)
 - [2.2.4.1](#)
 - [2.2.6](#)
 - [2.2.8](#)
 - [2.3.4](#)
 - [3章](#)
 - [3.2](#)
 - [3.4 おまけ問題](#)

1章

1.1.2 CPUとは

問題

- (1) コンピュータの五大要素を全て述べよ。
- (2) CPUは、コンピュータの五大要素でいう 何 と 何 を担っているか。

解答

- (1) 制御装置、演算装置、記憶装置、入力装置、出力装置 （順不同）
- (2) 制御装置 と 演算装置

1.1.3 メモリとは

問題

- (1) 書き込みが自由にできるメモリをアルファベット三文字で何というか。
- (2) (1) に対して、書き込みが自由に**できない**メモリをアルファベット三文字で何というか。
- (3) メモリにおいて、データを格納する一つ一つの場所を指す番号を何というか。
- (4) 上の図について、A が格納されている (2) は、いくつか。

解答

- (1) RAM
- (2) ROM
- (3) 「アドレス」（あるいは「番地」でも可）
- (4) 3 番地

1.2.2 ビットとデータ表現

問題

- (1) 0 と 1 の組み合わせにより表現された数字列を、何というか。
- (2) (1) を構成するそれぞれの 0 や 1 を何と呼ぶか。
- (3) (1) を 8つ区切り にしたとき、それぞれをどういった単位で呼ぶか。
- (4) asciiコードにおいて、 3 は 何列目の何行目 か。
- (5) asciiコードにおいて、 . は 何列目の何行目 か。

解答

- (1) ビット列
- (2) ビット
- (3) バイト
- (4) 3列目 の 3行目
- (5) 2列目 の 14行目

1.2.3 2進数と10進数、16進数

問題

(1) 173 を 8桁の2進数と、16進数に直しなさい

(2) 01001011 を 10進数に直しなさい

解答

(1) 10101101、0xAD

173 は $128 + 32 + 8 + 4 + 1$ である。

よって、累乗に直すと $2^7 + 0 + 2^5 + 0 + 2^3 + 2^2 + 0 + 2^0$ だから、
10101101 となる。

また、16進数で 1010 は A、1101 は D であるから、173 は 0xAD となる。

(2) 75

まず、二進数を 2の累乗 に直したい。

$01001011 = 0 + 2^6 + 0 + 0 + 2^3 + 0 + 2^1 + 2^0$ となるから、

$01001011 = 64 + 8 + 2 + 1$ であり、

75 が答えとなる。

1.2.4 2進数と負の数（二の補数表現）

問題

(1) 二進数8桁で -20 を表現せよ。ヒント：通常の 20 は 00010100 である。

(2) 10000110 は、符号なし二進数でいくつか。また、符号付き二進数でいくつか。

解答

(1) 11101100

20 (00010100) を、ビット反転すると 11101011 になる。

これに +1 すると 11101100 になる。

(2) 符号なし：134、符号付き：-122

符号なし

$10000110 = 2^7 + 0 + 0 + 0 + 0 + 2^2 + 2^1 + 0$ より、

$10000110 = 128 + 4 + 2$ であり、計算して 134 となる。

符号あり

一番左の符号ビットが 1 なので負の数になる。符号なしと同様に変換できない。

正の数に変換する。ビット反転して 01111001 となり、+1 して 01111010 となる。

これを10進数に直すと、

$01111010 = 2^6 + 2^5 + 2^4 + 2^3 + 2^1$

$01111010 = 64 + 32 + 16 + 8 + 2$ より 122 となる。

負の数なので、マイナスをつけて -122 が答え。

おまけ

二の補数表現では、正の数 → 負の数 だけではなく、負の数 → 正の数 の変換も行うことが出来る。
これを利用すると、「2回同じ手順を繰り返すと、元に戻る」ことが分かる。

$20 (00010100) \rightarrow -20 (11101100) \rightarrow 20 (00010100)$ といったように。

この「2回行くと元に戻る」性質を **involution** と言ったりする。

名前を覚えなくても良いが、戻ってくる性質のことは知っているとな面白いかもしれない。

2章

2.2.1.0

問題

GR5 に値 120 を格納する。

解答例

```
MAIN      START
          LD      GR5, =120      ; せっかくなのでリテラルを使ってみる。サンプルのようにラベルを使っても良い
          RET
          END

-----
; ラベルを使う場合
MAIN      START
          LD      GR5, A
          RET
A          DC      120
          END
```

2.2.1.1

問題

アドレス 0x0015 に ST 命令を使用して値 500 を格納する。

解答例

```
MAIN      START
          LD      GR0, =500      ; せっかくなのでリテラルを使ってみる。もちろんラベルを使っても良い
          ST      GR0, #0015     ; アドレスは 0x がついている16進数。10進数で 21 を指定しても良い
          RET
          END

-----
; ラベル、10進数を使う場合
MAIN      START
          LD      GR0, A
          ST      GR0, 21
          RET
A          DC      500
          END
```

2.2.1.2

問題

ラベル `VALUE` を作成し、値 1 を格納する。

次に、その一行下に `DC 3` を行う。

最後に、指標レジスタを上手く使い `VALUE` のアドレスの1つ後を指定し、入っているデータを `GR0` に入れる。

解答例

指標レジスタに 1 を指定することで、`VALUE + 1` アドレスを実現する。

方法は「`GR1`などに 1 を格納し、`LD GR0, VALUE, GR1` とする」であり、

ラベルを追加して `DC` しても良いし、`DC` とリテラルで入れても良いし、`LAD` を使用しても良い。

```
MAIN    START
        LAD    GR1, 1          ; GR1 を指標レジスタとする。1を入れる
        LD     GR0, VALUE, GR1 ; VALUE のアドレスの 1つ後 のアドレスに入っているデータを GR0
        に入れる
        RET
VALUE   DC     1
        DC     3
        END
```

2.2.2.3

問題

1. `ADDA` 命令では 0 となり、`ADDL` 命令ではオーバーフローする足し算
2. 引き算は、符号を変えた足し算として行われる。`100 + (-2)` (`0000000001100100 + 1111111111111110`) など、単に桁が溢れる計算ではオーバーフローは起きない。では、桁あふれとオーバーフローの違いは何か

解答例

1. まず、`ADDA` つまり符号付きの足し算では 0 となるから、`1 + (-1)` や `2 + (-2)` のように、同じ数を引く (`a + (-a)`)。このうち、`0 + (-0)` を除く全てで成り立つ。

```
MAIN    START
        LD     GR0, A
        LD     GR1, B
        ADDL   GR0, GR1
        RET
A        DC     1
B        DC     -1
        END
```

2. 桁あふれとオーバーフローの違いは、符号が関係する。同じ符号で足し算を行った結果、違う符号になってしまった場合にオーバーフローとする。本来、同じ符号を足すから `1 + 3 = 4` や

$-13 + (-100) = -113$ など、どう頑張っても同じ符号の値が得られるはずである。しかし、表現できる値の範囲を超え桁が溢れると、符号が変わってしまう。この「表現できる値の範囲を超え桁が溢れる」をオーバーフローとする。50000 + 50000 などを行い、得られる2進数の最上位ビットを確認してほしい。

2.2.4.1

問題

CPA と CPL では、FR の値が異なる結果になることがある。

では、CPA では FR = 000 となり、CPL では FR = 010 (SFが1) となる計算を考えてみよう。

解答例

二つの数字を a と b とする。

ポイントは、「符号付きだと a の方が大きい」であり、符号なしだと「符号なしだと b の方が大きい」だ。

負の数は、符号なしだと「符号ビットが数値 $2^7 = 128$ になる」から、非常に大きな値になる。

よって、 a : 正の数 と b : 負の数 を比較すると良い！

```
MAIN    START
        LD      GR0, A
        LD      GR1, B
        CPL     GR0, GR1
        RET
A        DC      1
B        DC     -1
        END
```

2.2.6

問題

1 から 100 までの和の合計を求め、sum ラベルに格納する。

つまり、 $1 + 2 + 3 + 4 + \dots + 99 + 100$ の答え求め、sum に保存する

解答例

```
MAIN    START
        LAD     GR0, 0      ; GR0 を 1, 2, 3, ... と変えていく
        LAD     GR1, 0      ; GR1 を計算結果とする
FOR      ADDA    GR0, =1     ; GR0 を 1 増やす
        ADDA    GR1, GR0    ; GR1 = GR1 + GR0
        CPL     GR0, count  ; GR0 と 100 を比較
        JMI     FOR        ; GR0 < 100 なら繰り返す
        ST      GR1, sum    ; sum に計算結果を保存
        RET
count    DC      100
sum      DS      1
        END
```

2.2.8

問題（お遊びコーナー）

永遠に終わらないプログラムを作ってみよう！

方法1：分岐命令で自分自身、あるいはそれ以前のメモリ番地に飛ぶ

方法2：PUSH と RET を使って前に戻る

解答例

方法1

```
MAIN    START
LOOP    JUMP    LOOP    ; 自分自身に飛ぶ。つまり、ここをもう一度実行
        RET
        END
```

固まったように見えるが、実行ログをスクロールすると勝手に一番下まで戻される。動いてはいるようだ。

```
LOOP    NOP
        JUMP    LOOP
```

とすると分かりやすいかも

方法2

```
MAIN    START
LOOP    PUSH    LOOP    ; 自分自身のアドレスをスタックに積む
        RET          ; スタックの値を PC に設定する。つまり LOOP に戻る
        END
```


2.3.4

問題

さて、ここまで説明した内容で「おまじない」がすべて説明された。
ということで、次のプログラムについて、上から下まで全て解釈してみよう。

MAIN	START	PRG1
	LD	GR2, =30
PRG1	LD	GR0, A
	LAD	GR1, 50
	SUBL	GR1, GR0
	ST	GR1, B
	RET	
A	DC	100
B	DS	1
	END	
PRG2	LD	GR0, C
	RET	
C	DC	130

解答例

まず、プログラムの実行開始位置を PRG1 に設定する

GR2 にリテラルを使って 30 を格納する（実行されない）

PRG1 の宣言。GR0 に A ラベルの中身（100）を格納する。この処理をする命令のアドレスに PRG1 の名前が紐づく。

GR1 に 50 を格納する。

論理減算で GR1 から GR0 を引く。論理なので -GR0 は 65436 として解釈される。結果は $50 + 65436 = 65486$

B ラベルのついたメモリ番地に GR1 の中身（65486）を格納する

スタックに何もないので、RET で終了する

A ラベルの付いた領域に 100 を入れる

B ラベルから 1語ぶん 領域を確保する

プログラムの終了。以降の3行は解釈されず、機械語に変換されない。実行もされない。

PRG2 の宣言。GR0 に C ラベルの中身（130）を格納する。

RET 命令。

C ラベルの付いた領域に 130 を入れる。

ちなみに、これを機械語に直すと以下ようになる（16進数）

1020 000C 1000 000A 1210 0032 2710 1110 000B 8100 0064 FFFF 001E

3章

3.2

問題

GR0 の初期値を 3 にして、5回繰り返してみよう。

解答

「繰り返すには、limit の値はどうなっていると良いか」を考える。

- 1行目を出力したときは、GR0 は $3 + 1 = 4$ になる。 よって、繰り返すには $4 < limit$ である。
- 2行目を出力したときは、GR0 は $4 + 1 = 5$ になる。 同様に、繰り返すには $5 < limit$ である。
- 3行目を出力したときは、GR0 は $5 + 1 = 6$ になる。 $6 < limit$ なら繰り返す。
- 4行目を出力したときは、GR0 は $6 + 1 = 7$ になる。 $7 < limit$ なら繰り返す。
- 5行目を出力したときは、GR0 は $7 + 1 = 8$ になる。 $8 < limit$ なら繰り返す。

よって、5回目のときに $8 < limit$ を満たさないなら、これ以上繰り返されない。

limit = 8 が適切と分かる。

```
MAIN      START
          LAD      GR0, init      ; GR0 をカウンタとして使う。初期値 3 を代入

FOR        OUT      string, len   ; 繰り返したい部分の先頭を FOR とラベル付け  <----┐
          ADDA     GR0, one       ; カウンタを 1 増やす                        │
          CPL      GR0, limit     ; カウンタ と 繰り返し回数 を比較          │
          JMI      FOR           ; カウンタ < 繰り返し回数 なら FOR に戻る  ----┘

          RET

string DC    'Hello, World!'
len    DC    13
limit  DC    8          ; 繰り返し回数。8に変更。
init   DC    3          ; カウンタの初期化に使う値。3に変更。
one     DC    1
          END
```

3.4 おまけ問題

問題

今のプログラムは、何行ぶん出力したのかが分かりにくい。

そこで、1: Hello, World! のように、先頭に何個目の Hello, World! なのか分かるようにしたい。

解答例

```
MAIN      START
          IN      limit, =1      ; 繰り返し回数を標準入力

          LD      GR0, limit     ; 移植部分
          SUBA    GR0, =#0030
          ST      GR0, limit

          LAD     GR0, 0         ; GR0 をカウンタとして使う。初期値0 を代入

FOR        ADDA   GR0, =1        ; カウンタを 1 増やす

; ここに、数値→数字の変換を記述
          LD      GR1, GR0       ; カウンタを直接弄らなくていいように、GR1 にコピー
          ADDA    GR1, =#0030    ; 数値 + 0x30 すれば 数字になる
; row に、変換した数字を格納
          ST      GR1, row
; 文字列を出力
          OUT     row, len       ; 先頭は row 。stringだと ': Hello, World!'から出力が始まる

          CPL     GR0, limit     ; カウンタ と 繰り返し回数 を比較
          JMI     FOR           ; カウンタ < 繰り返し回数 なら FOR に戻る

          RET

row        DS      1
string     DC      ': Hello, World!'
len        DC      16           ; row が 1文字、': ' が 2文字、全体で 1 + 2 + 13 = 16文字
limit      DS      1           ; 繰り返し回数
          END
```