4章 電卓を作ろう

本章では、アセンブリ言語を実際に使って、少し実用的なものを作ってみることにする。
具体的には、式を入力すると計算を行って、計算結果を出力してくれる「電卓」のようなものを作る。

4.1 電卓に必要なもの（要件定義）

早速だが、電卓ってなんだろう。

数字を入力するボタンがあって、次のような機能がある。

- 足し算・引き算・掛け算・割り算（まとめて四則計算と呼ぶ）
- パーセント計算
- 税率計算
- メモリ機能（M+, M-, MR などのボタンを見たことあるだろう）
- など……。

正直、私は四則計算くらいしか使ったことないので、パーセントとか税率とか、メモリは特によく分かってないです。
ということで、とりあえず四則計算が出来るような電卓を作ろう。実際の電卓に入力するみたいに、計算する順番で式を入力していく。
$3 + 5 \times (4 - 2)$ がしたければ、4 - 2 × 5 + 3 の順番で押すように、「計算する順番」で入力する。

ということで、改めて電卓の要件をまとめよう。

- 入力を受け取れる
- 入力内容を理解して、数値と演算子を解釈できる（4 + 3 を「4 と 3 を足し算する」と理解できる）
- 四則計算ができる
- 結果を出力できる

これくらいかな。とりあえずこれら要件を、個別に満たしていこう。
それぞれの要件パーツがしっかり動くよう「単体テスト」出来たら、組み合わせて安全にソフトを作る。

なお、仕様として、以下の記号を代用する。

- 掛け算の記号 $\times$ は `*`（アスタリスク）とする。3×5 は `3 * 5` と書く。
- 割り算の記号 $\div$ は `/`（スラッシュ）とする。$3 \div 5$ は `3 / 5` と書く。

4.2 入力を受け取る

まずは入力を受け取ろう。入力できなければ何も始まらない。
入力を受け取るにはどうしただろうか。分からなければ前章を確認してみよう。

入力を受け取るには、`IN` 命令を使用する。
そして、入力を保持するためのメモリ領域も必要になる。`DS` 命令で確保しよう。
入力する文字数が不確定（「1+1」かもしれないし、「9+8-7*6/5」みたいな長い式かもしれない）なので、最大の 256語 分を確保しておく。

```
INPUT   START
        IN      STR, =256
        RET

STR     DS      256

        END
```

とりあえず、入力したものを受け取る機構はできた。

なお、見やすさのために処理や変数で空行を挟んでいるが、空行は無くても良い。
極論を言ってしまえば、以下のような書き方でも動く。

```
INPUT START
 IN STR,=256
        RET
STR DS   256
  END
```

処理に影響は出ないので、自分のわかりやすいように書こう。

4.3 入力を分解して解釈する

正直このパートが一番難しい。心してかかろう。

まず、実際に打ち込む計算式を考えてみよう。何か法則性があるかもしれない。

> 1 + 1
> $(4-7)\times 4$ $\rightarrow$ 4 - 7 * 5
> $(3\times 5-2)\times 7$ $\rightarrow$ 3 * 5 - 2 * 7

入力を見てみると、「数字 演算子 数字 演算子 数字 ...」と、数字と演算子が交互に現れるようだ。
ということで、入力を解釈する簡単な方法は、「一文字ずつ確認していって、数字なら保持して、演算子なら計算」だろうか。
演算子については、`+` なのか `-` なのか `*` なのか `/` なのか、適切に分けて それぞれに合った計算 に飛ばなければならない。
具体的に言うなら、「3が来た。おっ次は + か、足し算だな。足す数はなんだろう？」といった処理をしたい。
概念的に、処理を言語化してみよう。

```
入力文字で繰り返し（入力を一文字ずつ見るため）
    文字が演算子か？

    演算子なら
        対応する計算に飛ぶ
    演算子じゃないなら
        数字なので保持する
```

このように実装したいのだが、少し問題がある。「1 + 1」に、この処理で行おうとすると、

1文字目 : `1` -> 数字なので保持する
2文字目 : `+` -> 演算子なので計算する

となり、足される数が分からない状態で計算を行おうとしてしまう。
そこで、今は「対応する計算 の時に、足される数を受け取ってから計算する」ように記述してみる。

```
; 対応する計算
+ のとき
    足す数を保持する
    足し算する
- のとき
    引く数を保持する
    引き算する
```

のように、冗長だが飛んだ先で数字を手に入れる。

これに合うように、コードを書いてみよう。

```
; 何文字目か を GR1 で保持する
loop    START
        ; 入力した文字でくり返し
        LD      GR0, STR, GR1  ; 一文字受け取って GR0 へ
        LAD     GR1, 1, GR1    ; ADDA GR1, =1 と同じ。メモリを削減できる
        CPL     GR0, =#0080    ; asciiコードの範囲外か
        JPL     FIN            ; 範囲外なら文字じゃない。これ以上後ろに式は続かないのでループ
終わり

        ; 文字が演算子か？
        CPA     GR0, ='+'      ; 文字 '+' と比較
        JZE     PLUS           ; 対応する計算 PLUS に飛ぶ
        CPA     GR0, ='-'
        JZE     MINUS
        CPA     GR0, ='*'
        JZE     MUL
        CPA     GR0, ='/'
        JZE     DIV

        ; 演算子じゃないなら数字なので保持
        SUBA    GR0, =#0030    ; 数字を 数値 に変換
        LD      GR2, GR0       ; GR2 に数値を保存
        JUMP    loop

; 対応する計算
; + のとき
PLUS    LD      GR0, STR, GR1
        LAD     GR1, 1, GR1
        SUBA    GR0, =#0030
        LD      GR3, GR0       ; 足す数を GR3 に保持
        NOP                    ; 実際に足し算を行う。今は NOP で仮実装
        JUMP    loop           ; まだ後ろに式が続いてるかもしれないのでジャンプ
; - のとき
MINUS   LD      GR0, STR, GR1
        LAD     GR1, 1, GR1
        SUBA    GR0, =#0030
        LD      GR3, GR0
        NOP                    ; 実際に引き算を行う。今は NOP で仮実装
        JUMP    loop
; * のとき
MUL     LD      GR0, STR, GR1
        LAD     GR1, 1, GR1
        SUBA    GR0, =#0030
        LD      GR3, GR0
        NOP                    ; 実際に掛け算を行う。今は NOP で仮実装
        JUMP    loop
; / のとき
DIV     LD      GR0, STR, GR1
        LAD     GR1, 1, GR1
        SUBA    GR0, =#0030
        LD      GR3, GR0
        NOP                    ; 実際に割り算を行う。今は NOP で仮実装
        JUMP    loop
```

```
FIN     RET

STR     DC      '3-5+7/4'
        END
```

かなり長くなるが、こんな感じだろうか。

入力した文字列について、指標レジスタでインデックス修飾（久しぶりに出てきた単語）を行い、1文字ずつ GR0 に読み出す。
文字じゃなかったら `loop` から抜ける。
文字が 演算子 だったら、それぞれ「この文字？」と比較して飛ぶ。
どの演算子もヒットしなかったら、数字だと信じて 数値に変換。GR2 に格納する。

対応する計算に飛んだ先では、まずは入力文字を読んで 数値 を GR3 に入れる。
そのあと、GR2 と GR3 で実際に計算を行う。
この後にも式が続いてるかもしれないので、 `loop` に飛んでループ再開。

ここで、「最初は数字なので GR2 に格納しloop」、「次は演算子なので飛ぶ。飛んだ先で次の数字を GR3 に。またloop」と処理が起きるので、つぎの `loop` では 演算子 がヒットする。
書き方を変えるならば、
`入力 = 数字 {演算子 数字}の繰り返し`
というように、始めに数字を読んで、次からは演算子と数字をセットで処理 を繰り返す、を行う。

具体的に言うなら、

3 を GR2 に入れる。
+5 が見えた。5を足そう。3 + 5 で 8 だな。
-7 が見えた。7を引こう。8 - 7 で 1 だな。
+4 が見えた。2を足そう。1 + 4 で 5 だな。
あ、文字が終わったらしい。じゃあ計算結果は 5 だ！

といった流れになる。

4.4 四則計算

四則計算では、足し算、引き算、掛け算、割り算を定義する。
といっても、足し算引き算については、既に `ADDA` や `SUBA` 命令などがあるので良いだろう。
掛け算と割り算について実装を考えよう。

4.4.1 掛け算

そもそも、掛け算ってなんだっただろうか。小学2年生に戻った気持ちで掛け算の定義を思い返してみてほしい。

掛け算は、「同じものを何回も足す」だった。「〇を△回足す」ことを、「〇 × △」で書けるようにした。
つまり、ちょっとお堅く書くなら、次のような定義になる。

$$n \times m := \underbrace{n + n + n + \cdots + n}_{m\text{個}}$$

ということで、「かけられる数を、繰り返しで 掛ける数 回だけ足す」ことで、掛け算が実装できそうだ。
この実装には、`forループ` の考え方を使えば出来そうだろう。カウンタを用意して、ループ回数を決める。

```
; 掛けられる数を GR2, 掛ける数を GR3 に入れているとする。
MAIN    START
        LAD     GR2, 3
        LAD     GR3, 5          ; 3 * 5 となる
        LAD     GR4, 0          ; GR4 の初期化
        LAD     GR5, 0          ; GR5 の初期化

MUL     CPA     GR3, =0         ; 掛ける数が 0 じゃないかの確認
        JZE     M_END           ; 0 だったら計算しないで M_END に
; カウンタを GR4 とする。繰り返し上限は GR3
M_MAIN  LAD     GR4, 1, GR4     ; GR4 に 1 を足す
        ADDA    GR5, GR2        ; 一時的に GR5 に GR2 を足す。掛け算の結果が GR5 に格納
        CPA     GR4, GR3        ; カウンタ と 繰り返す回数 を比較
        JMI     M_MAIN

M_END   LD      GR2, GR5        ; 掛け算の結果を GR2 に移す
        RET

        END
```

4.4.2 割り算

割り算も同様に、簡単な定義と実装方法を考えたい。

定義から行きたいのだが、これは一筋縄に行かない。小学校では、割り算は以下のどっちかで習う。

- 全体の数 n を、m個ずつ に分けて、何個の塊が作れるか。
- 全体の数 n を、m分割しようとしたとき、一つの塊が何個の要素から作られるか。

それぞれ、
「11個のみかんを5個ずつに分けたとき、何セットと余りができるか」
「11個のみかんを5人に配るとき、一人あたり何個配れて余りがいくつか」
という例が挙げられる。
では、この二つの定義のうち、どっちが実装に向いているだろうか。考えてみよう。

11個のみかんを5個ずつに分けたとき
これは、全体11個から「5個を取り出す」操作を繰り返すことになる。
まず、全体11個から5個取り出せば、5個の塊が 1 つ出来る。11 - 5
次に、残った6個から5個取り出せば、5個の塊がまた作れる。合計 2 つ出来る。6 - 5
残った1個から5個取り出すことはできないので、答えは 2セット と 余りが1個 になる。

11個のみかんを5人に配るとき
これは、全体11個から「5人が1つずつ取っていく」操作を繰り返すことになる、
まず、1週目で5人が順番に、全体11個から1つずつ取っていく。11 - 1 - 1 - 1 - 1 - 1
次に、2週目で5人が順番に、残った6個から1つずつ取っていく。6 - 1 - 1 - 1 - 1 - 1
残った1個は、全員に分配できないので、答えは 一人あたり2個 と 余り1個 になる。

やっていることは正直同じなのだが、1つずつ引くのか、一気に5引くのか、細かい違いがある。
「-1 を5回も繰り返す、これが何回出来るか」をしていては、計算の回数がとんでもなくなってしまう。
そこで、前者の方式を採用しよう。

式にするなら以下の通り。

$$n \div m = \text{商} \cdots \text{余り} \iff n \left(\underbrace{- m - m - \cdots - m}_{\text{何回出来るか = 商}} - \text{余り} \right)$$

ということで、「割られる数から、割る数を 引けるだけ繰り返しで引く」ことで、割り算が実装できそうだ。
無条件で繰り返し続けて、「残り < 割る数」になったら、これ以上引けないので繰り返しから抜ける。
繰り返した回数を 割り算の結果 として保持する。

```
; 割られる数を GR2, 割る数を GR3 に入れているとする。
MAIN    START
        LAD     GR2, 11
        LAD     GR3, 2          ; 11 / 2 となる
        LAD     GR4, 0          ; GR4 の初期化

DIV     CPA     GR3, =0         ; 割る数が 0 じゃないかの確認
        JZE     E_DIV0          ; 0 だったら E_DIV0 に飛ぶ
; カウンタを GR4 とする。
D_MAIN  CPA     GR2, GR3        ; 割られる数･残り と 割る数を比較
        JMI     D_END           ; 残り < 割る数 なら D_END に飛ぶ
        LAD     GR4, 1, GR4     ; GR4 に 1 を足す
        SUBA    GR2, GR3        ; GR2 - 割る数。一回割った残りが GR2 に再格納される
        JUMP    DIV             ; DIV に戻って繰り返し

; 割り算の結果 今は商がGR4、余りがGR2に入っている
D_END   LD      GR3, GR2        ; 余りを GR3 に移す
        LD      GR2, GR4        ; 割り算の結果を GR2 に移す
        RET

; 0除算のエラー文を出力
E_DIV0  OUT     ='Error: divided by 0', =19
        RET

        END
```

割り算、掛け算では、後の計算につなげるために 結果を GR2 に移す処理を追記している。

4.5 出力

計算結果を出力する。ここは入力の逆なので簡単、と言いたいところだけど、少しめんどくさい。

前章の問題で、「数値を数字に変換して出力する」を行ったが、あれは不十分だったりする。
というのも、1桁の数字についてはあれで良いのだが、2桁以上になるとおかしくなる。
具体的に考えてみよう。前章での実装は、「0x30を数値に足して、数値に対応する数字のasciiコードを作る」だった。
数値が 6 の時は、0x36 となり 文字 `6` が作れる。良いだろう。
数値が 0 の時も、0x30 となり 文字 `0` が作れる。これも良い。
しかし、数値が 10 の時、10は16進数で 0x0A なので 0x3A になる。0x3A は 文字だと `:` になってしまう。
数値が 35 の時は、35は16進数で 0x23 なので 0x53 になる。0x53 は 文字だと `S` になってしまう。

ということで、「数値を一桁ずつ解釈して、都度 数字 に変換する」必要がある。

1章でもやったが、私たちの使う10進数は、10や100、1000など 10^n を基準に桁が上がる。
ということで、各位の数を手に入れるには、「10で割って、その余りを見る」を繰り返す。
この方法では、一の位から十、百、…と順番に出てくる。そのため、出力するときには逆順でなければならない。
逆順で取り出すには、格好のツールが存在した。先に入れたものを最後に取り出す。スタックを活用すれば実装しやすそうだ。

やることは 計算結果÷10をし続けて余りを積む なので、割り算の処理を参考にループしよう。
計算結果を÷10して余りを積む、商をさらに÷10して積む、…を繰り返し、商が0になるまで行う。

また、スタックから何回取り出せばいいか分かるように、桁数も保持しておこう。
スタックが無いのに取り出そうとすると、できないのでエラーになってしまう。

そして、桁数ぶん取り出して1桁ずつメモリに入れる。数値から文字に変換するのを忘れずに。
全ての桁が入ったら、文字列として出力する。

数値を一桁ずつスタックに積む処理の名前は、「数値を文字列に」の意味で `number to string` から `NUMTOS` とした。
`DIV10` で、計算結果 ÷ 10 を行う。
`STACK` で、余りをスタックに積み、桁数を保持する。
`SETANS` で、スタックから取り出して連続したメモリ `ans` に格納する。
`OUTPUT` は名前の通り出力を行う。

ということで次のページに実装例のプログラムを示す。
変なところで改行されると見づらいから改ページさせていただいた。

```
; 値が GR2 に格納されているとする
MAIN    START
        LAD     GR2, 173
        LAD     GR3, 0          ; GR3 の初期化
        LAD     GR4, 0          ; GR4 の初期化

NUMTOS  CPL     GR2, =0         ; 値が 0 じゃなければ、DIV10 に飛ぶ
        JNZ     DIV10
        PUSH    0               ; 値が 0 なら、0 を積んで 桁数を1にして SETANS へ
        LAD     GR4, 1
        ST      GR4, anslen
        JUMP    SETANS
DIV10   CPL     GR2, =0         ; 値が 0 になるまでループ
        JZE     SETANS
        ; GR2 / 10 をして、商を GR3 に
        CPA     GR2, =10
        JMI     STACK
        SUBA    GR2, =10
        LAD     GR3, 1, GR3
        JUMP    DIV10
        ; GR2 / 10 が終わったら、余りは GR2 に入っている
STACK   PUSH    0, GR2          ; インデックス修飾で GR2 の値を PUSH
        LD      GR2, GR3        ; 商を GR2 に移動
        LAD     GR3, 0          ; GR3 を初期化
        LAD     GR4, 1, GR4     ; 桁数を GR4 に保持
        ST      GR4, anslen     ; 桁数を anslen にも格納
        JUMP    DIV10

; スタックから取り出して数字列を作る。今何桁目 を GR3 に入れるとする
SETANS  CPL     GR4, GR3        ; 全桁終わるまで繰り返し
        JZE     OUTPUT
        POP     GR2             ; スタックから取り出し
        ADDA    GR2, =#0030     ; 数値 を 数字 に変換
        ST      GR2, ans, GR3   ; インデックス修飾で「何桁目」を指定する
        LAD     GR3, 1, GR3     ; 今読んだ桁数を増やす
        JUMP    SETANS

; 答えの出力
OUTPUT  OUT     ans, anslen

        RET
anslen  DC      1       ; 答えの桁数
ans     DS      5       ; 値は 0 ~ 65535 の最大5桁
        END
```

4.6 組み合わせる

ここまで作ってきた、入力、解釈、計算、出力を組み合わせて、一つのプログラムにする。
多分もう説明するまでもない。どこに飛ぶべきか、どの記述が必要かをよく観察して作ろう。
特に、レジスタの初期化やラベル・ジャンプ先に気を付ける必要がある。

上手く動かせたら、`4+2*2-6`（結果は6）や`3/2+8*5-7`（結果は38）などが計算できるか試してみよう。

```
MAIN    START

INPUT   IN      STR, =256

; 何文字目か を GR1 で保持する
; 入力した文字でくり返し
loop    LD      GR0, STR, GR1  ; 一文字受け取って GR0 へ
        LAD     GR1, 1, GR1    ; ADDA GR1, =1 と同じ。メモリを削減できる
        CPL     GR0, =#0080    ; asciiコードの範囲外か
        JPL     NUMTOS         ; 範囲外なら文字じゃないのでループ終わり

        ; 文字が演算子か？
        CPA     GR0, ='+'      ; 文字 '+' と比較
        JZE     PLUS           ; 対応する計算 PLUS に飛ぶ
        CPA     GR0, ='-'
        JZE     MINUS
        CPA     GR0, ='*'
        JZE     MUL
        CPA     GR0, ='/'
        JZE     DIV

        ; 演算子じゃないなら数字なので保持
        SUBA    GR0, =#0030    ; 数字を 数値 に変換
        LD      GR2, GR0       ; GR2 に数値を保存
        JUMP    loop

; 対応する計算
; + のとき
PLUS    LD      GR0, STR, GR1
        LAD     GR1, 1, GR1
        SUBA    GR0, =#0030
        LD      GR3, GR0       ; 足す数を GR3 に保持
        ADDA    GR2, GR3       ; 実際に足し算を行う
        JUMP    loop           ; まだ後ろに式が続いてるかもしれないのでジャンプ
; - のとき
MINUS   LD      GR0, STR, GR1
        LAD     GR1, 1, GR1
        SUBA    GR0, =#0030
        LD      GR3, GR0
        SUBA    GR2, GR3       ; 実際に引き算を行う
        JUMP    loop
; * のとき
MUL     LD      GR0, STR, GR1
        LAD     GR1, 1, GR1
```

```
        SUBA    GR0, =#0030
        LD      GR3, GR0
        CPA     GR3, =0         ; 掛ける数が 0 じゃないかの確認
        JZE     M_END           ; 0 だったら計算しないで M_END に
        LAD     GR4, 0          ; GR4 の初期化
        LAD     GR5, 0          ; GR5 の初期化
        ; 実際に掛け算を行う
M_MAIN  LAD     GR4, 1, GR4     ; GR4 に 1 を足す
        ADDA    GR5, GR2        ; 一時的に GR5 に GR2 を足す。掛け算の結果が GR5 に格納
        CPA     GR4, GR3        ; カウンタ と 繰り返す回数 を比較
        JMI     M_MAIN          ; MUL に戻ると、文字を取り出すところから動いてしまう
M_END   LD      GR2, GR5        ; 掛け算の結果を GR2 に移す
        JUMP    loop
; / のとき
DIV     LD      GR0, STR, GR1
        LAD     GR1, 1, GR1
        SUBA    GR0, =#0030
        LD      GR3, GR0
        CPA     GR3, =0         ; 割る数が 0 じゃないかの確認
        JZE     E_DIV0          ; 0 だったら E_DIV0 に飛ぶ
        LAD     GR4, 0          ; GR4 の初期化
        ; 実際に割り算を行う
D_MAIN  CPA     GR2, GR3        ; 割られる数・残り と 割る数を比較
        JMI     D_END           ; 残り < 割る数 なら D_END に飛ぶ
        LAD     GR4, 1, GR4     ; GR4 に 1 を足す
        SUBA    GR2, GR3        ; GR2 - 割る数。一回割った残りが GR2 に再格納される
        JUMP    D_MAIN          ; DIV に戻ると、文字を取り出すところから動いてしまう
D_END   LD      GR3, GR2        ; 余りを GR3 に移す
        LD      GR2, GR4        ; 割り算の結果を GR2 に移す
        JUMP    loop

; 計算結果の出力
NUMTOS  LAD     GR3, 0          ; このあと使う GR3, GR4 レジスタを初期化
        LAD     GR4, 0
        CPL     GR2, =0         ; 値が 0 じゃなければ、DIV10 に飛ぶ
        JNZ     DIV10
        PUSH    0               ; 値が 0 なら、0 を積んで 桁数を1にして SETANS へ
        LAD     GR4, 1
        ST      GR4, anslen
        JUMP    SETANS
DIV10   CPL     GR2, =0         ; 値が 0 になるまでループ
        JZE     SETANS
        ; GR2 / 10 をして、商を GR3 に
        CPA     GR2, =10
        JMI     STACK
        SUBA    GR2, =10
        LAD     GR3, 1, GR3
        JUMP    DIV10
        ; GR2 / 10 が終わったら、余りは GR2 に入っている
STACK   PUSH    0, GR2          ; インデックス修飾で GR2 の値を PUSH
        LD      GR2, GR3        ; 商を GR2 に移動
        LAD     GR3, 0          ; GR3 を初期化
        LAD     GR4, 1, GR4     ; 桁数を GR4 に保持
        ST      GR4, anslen     ; 桁数を anslen にも格納
```

```
        JUMP    DIV10

; スタックから取り出して数字列を作る。今何桁目 を GR3 に入れるとする
SETANS  CPL     GR4, GR3        ; 全桁終わるまで繰り返し
        JZE     OUTPUT
        POP     GR2             ; スタックから取り出し
        ADDA    GR2, =#0030     ; 数値 を 数字 に変換
        ST      GR2, ans, GR3   ; インデックス修飾で「何桁目」を指定する
        LAD     GR3, 1, GR3     ; 今読んだ桁数を増やす
        JUMP    SETANS

; 答えの出力
OUTPUT  OUT     ans, anslen
        RET

; 0除算のエラー文を出力
E_DIV0  OUT     ='Error: divided by 0', =19
        RET

STR     DS      256
anslen  DC      1               ; 答えの桁数
ans     DS      5               ; 値は 0 ~ 65535 の最大5桁

        END
```

4.7 総評と5章への引継ぎ

ここまで作ってきた電卓はどうだっただろう。プログラムは長いし、ループも多くて難しかったかもしれない。
少しでも楽しいと思っていただけたら幸せなのだが、、、難しいね。
正直、「こんなもの作って意味があるのか？」と思ってしまったかもしれない。そんな方は考えてほしい。

今回は CASLⅡ と呼ばれる命令セットを使い、四則計算のできるプログラムを作った。
仮に、このアセンブル結果（機械語）で、意図したように動作するコンピュータがあったらどうだろう。
このプログラムをアセンブルして、できた機械語をCPUのRAMに書き込む。
'0' ~ '9'、'+'、'-'、'*'、'/' が送れる テンキー のようなボタンと、文字が描画できるディスプレイを用意する。それらを CPU につないで電気を流したらどうだろう。
自分の手で機械に命令を書いて、簡易な電卓を創ったと言えないだろうか。
機械を動かす命令を「組み込んで」、自分だけの電卓というハードウェアを自作できる。

パソコンの中のアプリで、よくわからない英語のような記号のような変な言語を書いて、よくわからないけど動く。
それはそうなんだけど、そうじゃない。コンピュータの内部事情と機械語を思って、組み込みエンジニアという職業と向き合ってみてほしい。この「アセンブリ言語」がどういうものだったか、「組み込み系」がどういうものか。
「こういうことがしたい」を叶える機械を、自分で動かし方を制御するところから作る。
少しでも楽しんでいただけたらいいな。

さて、以上で本章は終わりだが、ここまで作成してきた電卓は正直言って不便なことが多い。
人間が計算式を見て、式を適切な順番で打ち込まないと答えがおかしくなってしまう。$3 + 4 \times 5$ を計算したいときに、式をそのままの順で 3 + 4 * 5 と入力してしまうと、 3 + 4 が先に計算されて 7 * 5 で 答えが 35 になってしまう。
また、2桁以上の数字に対応していない。13 + 5 を行おうとすると、「足される数 1 だな。GR2 に 1 入れよう」、「足される数 3 だな。GR2 に 3 入れよう」という順で処理されて 3 + 5 が行われてしまう。
また、これは計算速度の問題ではあるが、掛け算・割り算が信じられないくらい遅い。現状の方法で 3 * 100 や 500 / 2 でも計算しようものなら、数百回の足し引きを必要とする。何分かかるか考えたくもないだろう。
次章では、これらの問題を解決して「式をそのまま打ってもしっかり解釈して、適切に計算する」計算機を作る。