

YellowSoft

構造化アセンブラ YAH8

基本プログラムサンプル集

(H8/300H、H8S 版)

このサンプル集の内容

YAH8 基本プログラムサンプル集は、構造化アセンブラ YAH8 を初めて使う方を対象に、アセンブラプログラムの基本的ルーチンをサンプルにしたものです。

サンプルプログラムの実行方法

サンプルプログラムは各題材ごとにフォルダに分けて格納されています。プロジェクトは一部作成していますが完全ではありません。そのままでは動きません。ユーザの環境に合わせてプロジェクトを完成させて下さい。

実行方法

- 1 まず、プロジェクトを開きます。各フォルダの中に TEST.YIP というファイルが含まれています。これがプロジェクトです。
- 2 プロジェクトの設定を開きます。
- 3 「ターゲット (必須)」のページを開きます。
- 4 「オブジェクトの形式」を選んで下さい。イエロースコープをもっている方は「イエロースコープでシミュレーション」を選ぶと良いでしょう。ハードに依存しないプログラムなのでシミュレーションで動作させるだけで十分です。シミュレーションを選んだ場合はここですべての設定は終了です。
- 5 「スタートアップ (必須)」のページを開きます
- 6 ここで「スタートアップルーチン」を選択して下さい。最初は CS3048.ASM になっています。
- 7 「セグメント定義 (必須)」のページを開きます。
- 8 ここでユーザのシステムに合わせて自動作成して下さい。
- 9 以上で設定終了です。
- 10 プロジェクトの設定を閉じ、実行をすればプログラムが走ります。イエロースコープを使う場合は「実行ログウインドウ」を開いてから実行をして下さい。

デバッグ用ルーチン

DISP.ASM というファイルがあります。これはデバッグ用表示ルーチンですべてのサンプルプログラムで使用されます。したがってすべてサンプルプログラムでこのファイルをプロジェクトに含めるようにします。

構造化アセンブラ

このサンプルプログラムは構造化アセンブラ命令を使用しています。構造化アセンブラの仕様については以下のドキュメントを参照下さい。

- 1 YellowIDE のフォルダの下にある「マニュアル」フォルダの中の「YAH8 構造化命令仕様.PDF」
- 2 ヘルプファイルの「YAH8 構造化命令」の章

YAH8 アセンブラプログラムのきまり

ここでは YAH8 の使い方に関して簡単に説明します。詳しい仕様に関しては以下のドキュメントを参照下さい。

- 1 YellowIDE のフォルダの下にある「マニュアル」フォルダの中の「YAH8 言語仕様.PDF」
- 2 ヘルプファイルの「H8 アセンブラ疑似命令」の章

1 セグメント名

セグメント名は次の決まりに従って付けて下さい。

目的	セグメント名
プログラムコード、サブルーチン	TEXT
定数データ	DATA_CONST
初期値のあるデータ	DATA
初期値のないデータ	BSS

例	
segment	TEXT
◦	
◦	
segment	DATA_CONST
◦	
◦	
segment	DATA
◦	
◦	
segment	BSS
◦	
◦	

2 宣言

他のファイルで定義されたラベルを参照する場合は extern 宣言します。

逆に他のファイルから参照されるラベルは public 宣言をします。

例	extern abc1 def2
	public sub1

3 `_main` ラベル

プログラムは最初 `_main` というラベルからスタートします。したがって必ずラベル `_main` を定義して下さい。また `_main` は `public` 宣言します。

例	segment	TEXT
	<code>public _main</code>	
	<code>_main:</code>	

4 データの定義

4.1 初期値のあるデータ

初期値のあるデータは `DATA` セグメントまたは `DATA_CONST` セグメントに以下の疑似命令を使って定義します。

型	疑似命令
バイト	<code>DC.B</code>
ワード	<code>DC.W</code>
ロング	<code>DC.L</code>

ワード型、ロング型の直前にバイト型の定義がある場合は `EVEN` 疑似命令を直前に挿入して下さい。文字列はダブルコーテーション(")でくくればアスキーコードの列に分解されます。

例	segment	DATA
	<code>data1:</code>	<code>DC.B H'12, H'34, H'56</code>
	<code>str1:</code>	<code>DC.B "Hello, Word",0</code>
		<code>EVEN</code>
	<code>data2:</code>	<code>DC.W H'1234</code>
	<code>data3:</code>	<code>DC.L H'12345678</code>

4.2 初期値のないデータ

初期値のないデータ、つまり領域だけの確保は `BSS` セグメントに以下の疑似命令を使って定義します。

型	疑似命令
バイト	<code>DS.B</code>
ワード	<code>DS.W</code>
ロング	<code>DS.L</code>

ワード型、ロング型の直前にバイト型の定義がある場合は `EVEN` 疑似命令を直前に挿入して下さい。

例	segment	BSS
	<code>data1: DS.B 12</code>	<code>;12 バイト確保</code>

ファイル構成

¥Application

¥YAH8BASIC

Disp.ASM	デバッグ用表示ルーチン
¥ALITH1	データ列のチェックサムを求める
¥ALITH2	64 ビット加算
¥ALITH3	64 ビット減算
¥ALITH4	符号なし 32 ビット乗算
¥ALITH5	符号なし 32 ビット除算
¥BCD1	BCD N 桁 10 進加算
¥BCD2	BCD N 桁 10 進減算
¥BCD3	BCD データをバイナリに変換
¥BCD4	バイナリをメモリ上の BCD データに変換
¥BIT1	レジスタの '1' が立っているビットの数を数える
¥BIT2	レジスタの値を 2 進文字列に変換
¥BIT3	2 進文字列をバイナリに変換
¥CONV1	ASCII コードをバイナリに変換
¥CONV2	2 バイトの ASCII コードをバイナリに変換
¥CONV3	バイナリを ASCII コードに変換
¥CONV4	1 バイトバイナリを 2 個の ASCII コードに変換
¥DEC1	10 進文字列をバイナリに変換
¥DEC2	バイナリを 10 進文字列に変換
¥HEX1	16 進文字列をバイナリに変換
¥HEX2	バイナリを 16 進文字列に変換
¥MOV1	メモリクリア
¥MOV2	メモリフィル
¥MOV3	メモリコピー
¥MOV4	メモリ比較
¥STR1	文字列のコピー
¥STR2	文字列の連結
¥STR3	文字列の比較
¥STR4	文字列の長さを得る
¥STR5	英小文字を大文字に変換
¥STR6	英大文字を小文字に変換

言葉の定義

1 バイナリ

バイナリとバイナリデータは同じ意味で使用します。コンピュータが処理する数値はすべてバイナリです。たとえば値を加算したり、乗算したりする操作はすべてバイナリで行われます。バイナリはコンピュータが内部でもっている実際の値です。実際の値をどう表現するかはいろいろな表現方法があります。以下にそれらを示します。

2 アスキーコード

1文字を1バイトの数値に置き換えたものです。たとえば文字'1'はH'31です。文字としての1と、数値としての1では意味が異なります。画面に文字'1'を表示させたい場合はバイナリ(コンピュータがもっている実際の値)で1を送出してもだめです。アスキーコードはH'31ですからH'31に変換してから送じます。

3 文字列

文字を並べたものを文字列といいます。コンピュータから見ると文字列はアスキーコードに変換された単なる数値の並びです。

文字列	Hello
コンピュータから見た場合	H'48 , H'65 , H'6C , H'6C , 6F

4 10進文字列

文字列の一種で10進文字('0'~'9')しか使わない文字列です。

5 16進文字列

文字列の一種で16進文字('0'~'9'、'A'~'F'または'a'~'f')しか使わない文字列です。

5 2進文字列

文字列の一種で2進文字('0'が'1')しか使わない文字列です。

デバッグ関数

ファイル名 Disp.asm

0 で終わる文字列の表示

ファイル名 Disp.ASM

サブルーチン名

DispMsg

引数

ER0 文字列の先頭アドレス

戻り値

なし

動作

アドレス ER0 から始まる文字列を表示します。文字列の最後は 0 で終わっていなければなりません。

ヒント

C 言語の fputs 関数を呼び出して実現しています。具体的な使い方に関しては、各サンプルプログラムを見て下さい。

31 ?CALL _fputs(ER0, ER1=stdout)

?CALL は構造化命令です。C 言語の関数を呼び出すときこの命令を使います。C 言語の関数を呼び出すときは関数名の先頭に下線()を付けます。

引数はレジスタを指定します。long 型、ポインタ型引数の場合 ERn レジスタを引数に指定します。それ以外の場合は Rn レジスタまたは En レジスタを使用します。fputs の引数は両方ともポインタ型なので上記のように ER0、ER1 レジスタを使って引数を渡しています。

ER1=stdout のように定数で初期化することもできます。

C 言語の関数を呼び出すとどのレジスタが破壊されるかわからないので、スタックポインタ以外のすべてのレジスタを PUSH/POP しています。

すべてのレジスタの値を表示する

ファイル名 Disp.ASM

サブルーチン名

DispReg

引数

なし

戻り値

なし

動作

ER0 から ER7 までのレジスタを表示します。具体的な使い方に関しては、各サンプルプログラムを見て下さい。

ヒント

64 MOV.L ER7,ER3

65 ADDS.L #4,ER3 ;スタックには戻り番地が積まれているのでその分を戻す

スタックポインタ ER7 の値ですが、このサブルーチン(DispReg)が呼ばれる段階で戻り番地(4 バイト)がスタックに積まれるため、ER7 の値は 4 だけ少なくなっています。そのまま表示したのでは 4 だけ少ない値が表示されてしまいます。そのため 65 行目で 4 を加算しています。

メモリ領域を 32 バイト分表示する

ファイル名 Disp.ASM

サブルーチン名

DispMem

引数

ER0 表示するメモリ領域の先頭アドレス

戻り値

なし

動作

アドレス ER0 から始まる 32 バイト分のメモリ内容を表示します。具体的な使い方に関しては、各サンプルプログラムを見て下さい。

ヒント

C 言語の printf 関数を使って、16 バイトずつ、2 回に分けて表示しています。

メモリ操作

フォルダ MOV1 ~ MOV4

メモリ領域のクリア

フォルダ MOV1

サブルーチン名

MemClear

引数

ER0 メモリ領域の先頭アドレス

ER1 メモリ領域のサイズ

戻り値

なし

動作

アドレス ER0 から始まる ER1 バイトのメモリ領域を 0 にクリアします。

ヒント

6 extern DispReg

7 extern DispMem

他のファイルで宣言されたサブルーチンを呼び出す場合は extern 宣言する必要があります。

14 segmentTEXT

サブルーチンは TEXT というセグメント名で始めます。

15 public MemClear

このサブルーチンが他のファイルから呼び出される可能性がある場合は public 宣言します。

23 ?SWHILE(ER0 < ER1)

先頭が '?' で始まる命令は構造化命令です。この例の場合、ER0 が ER1 より小さい場合は ?ENDW までの命令をくりかえすという意味です。

36 segmentTEXT

37 public _main

38 _main:

YAH8 で必ず _main というラベルからプログラムを開始して下さい。

53 segmentDATA

初期値のあるデータは DATA というセグメントに記述します。

メモリ領域を定数で埋める

フォルダ MOV2

サブルーチン名

MemFill

引数

ER0 メモリ領域の先頭アドレス
ER1 メモリ領域のサイズ
R2L 値

戻り値

なし

動作

アドレス ER0 から始まる ER1 バイトのメモリ領域を R2L の値で埋めます。

ヒント

前項の MemClear と比べて、R2L への 0 代入がなくなっただけです。R2L は呼び出し側で設定します。

24 INC.L #1,ER0

H8 の場合、インクリメント・デクリメントは上記のようにオペランドの第一項に増分を指定します。指定できる増分は#1 か#2 です。ただしバイト型レジスタの場合は増分の指定はありません。

メモリ領域のコピー

フォルダ MOV3

サブルーチン名

MemCopy

引数

ER0 コピー元のメモリ領域の先頭アドレス
ER1 コピー先のメモリ領域の先頭アドレス
ER2 コピーサイズ

戻り値

なし

動作

アドレス ER0 から始まる ER2 バイトのメモリ内容をアドレス ER1 から始まるメモリ領域にコピーします。なお、コピー元のメモリ領域とコピー先のメモリ領域が重なってはいけません。

ヒント

実際問題として大きなメモリ領域のコピーはワード単位、あるいはロング単位でコピーを行った方が効率的です。このサンプルではバイト単位で行っています。

ただし、ワード・ロング単位でコピーを行う場合はコピー元、コピー先のアドレスが偶数でなければなりません。

ワード単位のコピーの例

```
?SWHILE(ER0 < ER2)
    MOV.W @ER0+,R3      ;R3 を仲介としてコピー
    MOV.W R3,@ER1
    INC.L #2,ER1
?ENDW
```

@ER0+は、ER0 が自動的に 2 だけインクリメントされます。しかし ER1 は第二オペランドのため@ER1+とは記述できません。したがって INC.L 命令で 2 だけ増やします。

算術演算

フォルダ ARITH1 ~ ARITH5

データ列のチェックサムを求める

フォルダ ARITH1

サブルーチン名

Checksum

引数

ER0	データ列の先頭アドレス
ER1	データ列のサイズ

戻り値

R2L	チェックサム
-----	--------

動作

アドレス ER0 から始まるサイズが ER1 のデータ列においてチェックサムを求めます。

ヒント

チェックサムとは、すべてのデータの和です。データ列を通信によって転送する場合に応用されます。転送元、転送先でチェックサムを計算し、データ列と一緒にチェックサムも送ります。チェックサムを比較することによってある程度通信の確かさを確認できます。

和の計算においてはオーバーフロー（桁あふれ）を無視します。下位バイトだけをチェックサムとします。

64 ビット減算

フォルダ ARITH3

サブルーチン名

Sub64

引数

ER0	被減算数の下位 32 ビット
ER1	被減算数の上位 32 ビット
ER2	減算数の下位 32 ビット
ER3	減算数の上位 32 ビット

戻り値

ER0	減算結果の下位 32 ビット
ER1	減算結果の上位 32 ビット

動作

ER1:ER0 - ER3:ER2 を計算して結果を ER1:ER0 に格納します。

ヒント

64 ビット加算とほぼ同じです。ADDX が SUBX に変化しただけです。加算の場合は桁あふれをオーバーフローと呼びましたが、減算の場合はボローと呼びます。どちらもキャリーフラグがセットされることには変わりはありません。

32 ビット乗算

フォルダ ARITH4

サブルーチン名

UMul32

引数

ER0 被乗数
ER1 乗数

戻り値

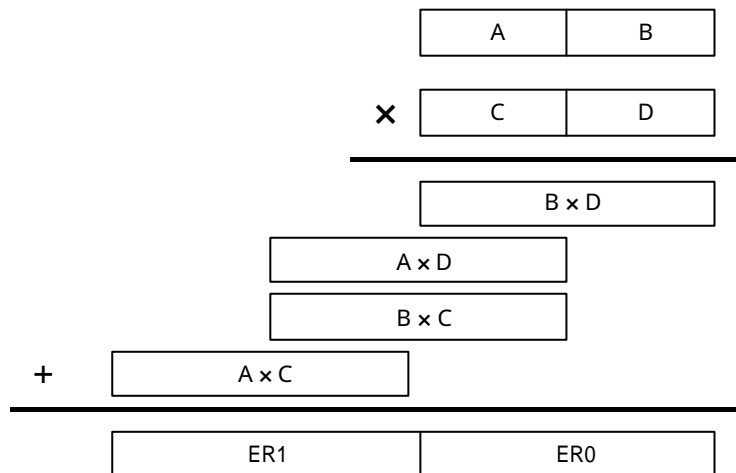
ER0 乗算結果の下位 32 ビット
ER1 乗算結果の上位 32 ビット

動作

ER0 × ER1 を計算して結果を ER1:ER0 に格納します。

ヒント

32 ビット × 32 ビットの乗算命令はありません。したがって 16 ビット × 16 ビットの乗算命令を 4 回行い、それらを足して求めます。原理は簡単で以下ようになります。



乗算自体は難しくありません。問題は 3 つの項の加算が出てくることです。そのため桁あふれが 2 回発生することがあります。したがってキャリーを別の場所に保存しておく必要があります。それが 44 行目です。

44 ADDX.B R4H,R4L ; R4L <- R4L + 0 + キャリー

R4H は常に 0 で R4L にはキャリーフラグだけが加算されます。

32 ビット除算

フォルダ ARITH5

サブルーチン名

UDiv32

引数

ER0 被除数
ER1 除数

戻り値

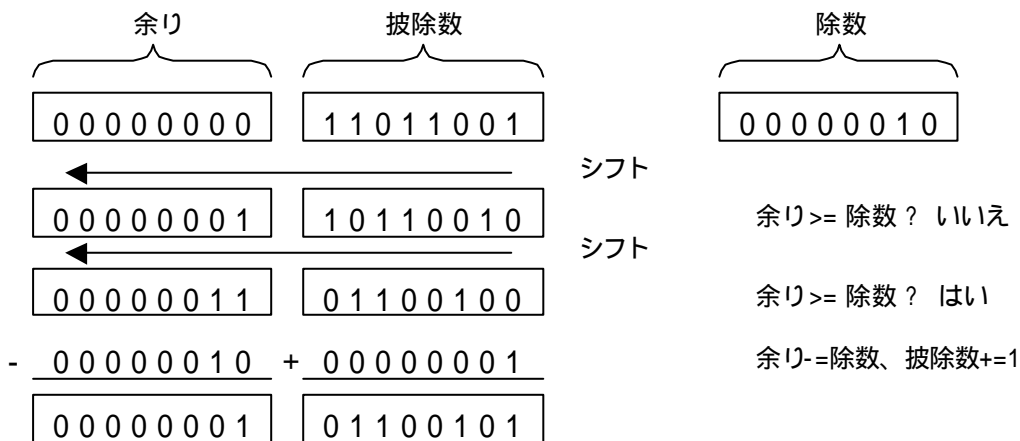
ER0 除算結果の商
ER2 除算結果の剰余

動作

ER0 ÷ ER1 を計算して商を ER0 に、剰余を ER2 に格納します。

ヒント

32 ビット ÷ 32 ビットの命令はありません。しかし乗算のように 16 ビットに分解して行うことはできません。1 ビットの減算を 32 回繰り返す方法をとります。方法は以下のとおりです。(8 ビット ÷ 8 ビットの例)



上記のように 0 で初期化された余りと被除数を並べて左方向にシフトします。これが以下の命令です。

```
19            SHLL.L        ER0
20            ROTXL.L      ER2
```

SHLL によって左側に押し出されたビットはキャリーフラグに入ります。次の ROTXL 命令によってキャリーフラグが ER2 の最下位ビットに入ります。

シフトしたら、余りと除数を比較します。もし余りが除数以上なら、余りから除数を引き算します。そして被除数をインクリメントします。同じことをビット数分繰り返せば、被除数に商が求まります。

コード変換

フォルダ CONV1 ~ CONV4

アスキーコードをバイナリに変換

フォルダ CONV1

サブルーチン名

Ascii2Bin

引数

R0L アスキーコード

戻り値

R0H バイナリに変換された値

動作

R0L に格納されたアスキーコードをバイナリ表現に変換して R0H に格納します。

ヒント

コンピュータの世界では、半角文字は 0~H'FF のコードで管理されます。たとえば数字、英大文字は次のようなコードが割り付いています。

数字	0	1	2	3	4	5	6	7	8	9
コード	H'30	H'31	H'32	H'33	H'34	H'35	H'36	H'37	H'38	H'39

英大文字	A	B	C	D	E	F	G	H	I	J	K	L	M
コード	H'41	H'42	H'43	H'44	H'45	H'46	H'47	H'48	H'49	H'4A	H'4B	H'4C	H'4D
英大文字	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
コード	H'4E	H'4F	H'50	H'51	H'52	H'53	H'54	H'55	H'56	H'57	H'58	H'59	H'60

このようなコードをアスキーコードと言います。アスキーコードは覚える必要はなく、アセンブラ上では文字をシングルコーテーション(')でくくれば、アスキーコードになります。たとえば文字 A のアスキーコードは'A'です。'A'と書くことと、H'41 と書くことはまったく同じことです。

さて、16 進数で使われる文字 0~9、a~f、A~F を、文字ではなく数値つまり H'0~H'9、H'A~H'F に変換します。つまり

'0' ~ '9' = H'30 ~ H'39 H'0 ~ H'9
'A' ~ 'F' = H'41 ~ H'46 H'A ~ H'F
'a' ~ 'f' = H'61 ~ H'66 H'A ~ H'F <- 文字ではなく数値なので大文字小文字の区別はない

このような処理アスキーバイナリ変換と言います。キーボードから数値を入力するとき、必ずこの処理が必要になります。上記を見れば分かるとおり'0'~'9'の場合は 0x30 を引けばバイナリになります。'A'~'F'の場合は 0x41 を引いた後に H'A(=10)を加算すればバイナリになります。

2 バイトのアスキーコードをバイナリに変換

フォルダ CONV2

サブルーチン名

WAscii2Bin

引数

R0 アスキーコード

戻り値

R1L バイナリに変換された値

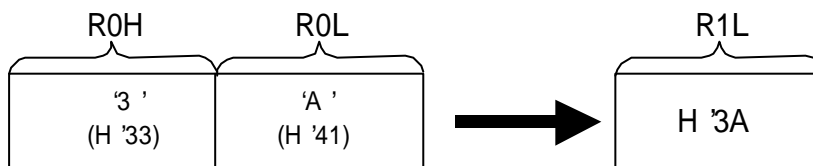
動作

R0 に格納された 2 バイトのアスキーコードをバイナリ表現に変換して R1L に格納します。

ヒント

前項で扱ったアスキーバイナリ変換を 2 回続けて、1 バイトのバイナリデータを得ます。

たとえば、キーボードからの入力がある '3', 'A' だとすればアスキーコード H'33、H'41 を受信します。この 2 バイトのアスキーコードを変換して H'3A にし、レジスタ R1L に格納します。



バイナリをアスキーコードに変換

フォルダ CONV3

サブルーチン名

Bin2Ascii

引数

R0L バイナリデータ

戻り値

R0H アスキーコードに変換された値

動作

R0L に格納されたバイナリデータをアスキーコードに変換して R0H に格納します。

ヒント

今度は逆に、バイナリデータをアスキーコードに変換します。表示装置に数値を表示する場合などに使用します。このサブルーチンは以下の変換をします。

バイナリ	アスキーコード
H'0 ~ H'9	H'30 ~ H'39
H'A ~ H'F	H'41 ~ H'46

H'0 ~ H'9 の場合はバイナリに H'30('0')を加算すればアスキーコードになります。

H'A ~ H'F の場合はバイナリに H'41('A')を加算して H'A(10)を減算すればアスキーコードになります。

1 バイトのバイナリを 2 個の ASCII コードに変換

フォルダ CONV4

サブルーチン名

WBin2Ascii

引数

R0L 1 バイトバイナリデータ

戻り値

R1 2 バイトのアスキーコード

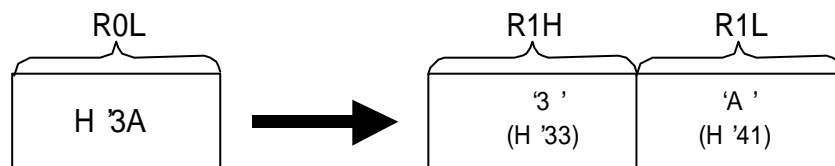
動作

R0L に格納された 1 バイトのバイナリデータを 2 個のアスキーコードに変換して R1 に格納します。

ヒント

前項で扱ったバイナリアスキー変換を 2 回続けて、2 バイトのアスキーコードを得ます。

たとえば、表示装置に数値 H'3A を表示したいとします。その場合、表示装置に '3' 'A' を送る必要があります。これはアスキーコードで H'33, H'41 を送れば良いことになります。



83 segmentBSS

84 disp: DS.B 8+1

初期化しないデータは上記のように BSS セグメントに定義します。DS.B は領域だけ確保する疑似命令です。上の例では変換したデータを保存する 9 バイトの領域を確保しています。

2 進化 10 進表現(BCD)

フォルダ BCD1 ~ BCD4

BCD N桁10進加算

フォルダ BCD1

サブルーチン名

AddBcd

引数

ER0 被加算数の先頭アドレス
ER1 加算数の先頭アドレス
ER2 桁数 ÷ 2 0 であってはいけない

戻り値

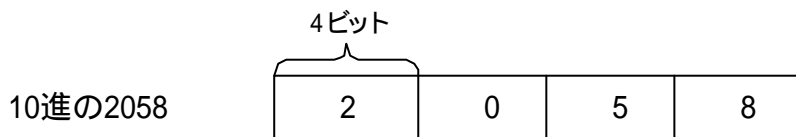
なし

動作

アドレス ER0 から格納された BCD データにアドレス ER1 から格納された BCD データを加算します。加算する桁数の半分を ER2 に渡します。桁数は偶数でなければなりません。

ヒント

BCD(2進10進)データとは、以下のように1ニブル(4ビット)に数値 H'0 ~ H'9 を入れて10進数表現で表すものです。



4ビットは本来 H'0 ~ H'F までの15の数値を格納できるのに対して、H'0 ~ H'9 の10個の数値しか使わないので、BCDは贅沢な表現方法と言えます。しかし、この表現に対応した命令は少なくBCDで演算するのは非常に効率が悪いです。そのためBCDはほとんど使われずにデータを10進で表示するための中間的保存手段として使われるぐらいです。演算自体はバイナリに変換したほうが効率的です。

BCDに対応した数少ない命令にDAAがあります。この使い方は以下のとおりです。

- 1 BCDデータをADDまたはADDXで加算します。
- 2 ADD、およびADDXはバイナリの命令なので加算結果は違ってきます。
- 3 そこでADD、ADDX命令の後にDAAを実行して、値を補正します。

プログラムで言えば、30、31行目にあたります。

```
30          ADDX.B      R3H,R3L          ;バイナリで加算
31          DAA         R3L             ;10進補正
```

BCD N桁10進減算

フォルダ BCD2

サブルーチン名

SubBcd

引数

ER0 被減算数の先頭アドレス
ER1 減算数の先頭アドレス
ER2 桁数 ÷ 2 0 であってはいけない

戻り値

なし

動作

アドレス ER0 から格納された BCD データからアドレス ER1 から格納された BCD データを減算します。減算する桁数の半分を ER2 に渡します。桁数は偶数でなければなりません。

ヒント

加算の場合とほぼ同じです。補正のための命令は DAS になります。

29	SHLL.W	E3	; キャリー復帰 E3[7] -> C
30	SUBX.B	R3H,R3L	; バイナリで減算
31	DAS	R3L	; 10進補正
32	ROTXR.W	E3	; キャリーの保存 C -> E3[7]

これはループ文中の処理です。SUBX により発生するボロー（キャリー）をループの次の実行まで保存しておくために E3 の最下位ビットにキャリーフラグの値をコピーします。（32 行目）そして 29 行目でキャリーフラグの値を元に戻しています。

キャリーフラグはループに入る前にクリアしておくことも忘れてはいけません。

23	SUB.W	E3,E3	; E3 をキャリー保存に使う
----	-------	-------	-----------------

これは E3 をクリアする命令です。キャリーの保存に使う E3 レジスタをクリアすることも忘れてはいけません。

BCD データをバイナリに変換

フォルダ BCD3

サブルーチン名

Bcd2Bin

引数

ER0 BCD データの先頭アドレス
ER1 桁数 ÷ 2

戻り値

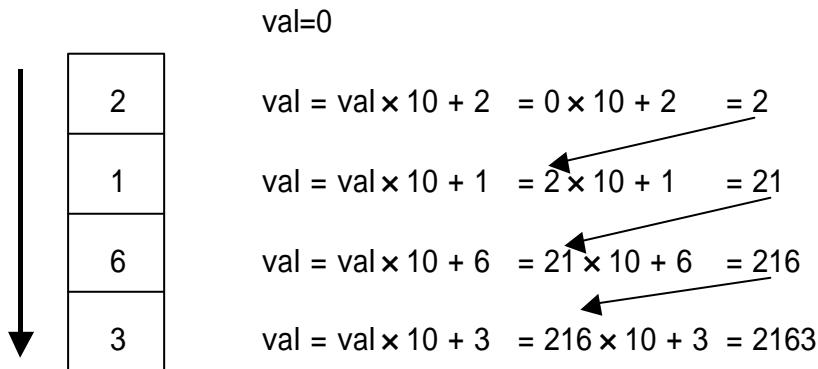
ER2 変換されたバイナリデータ

動作

アドレス ER0 から格納された BCD データをバイナリに変換して結果を ER2 に格納します。変換する桁数の半分を ER1 に渡します。桁数は偶数でなければなりません。

ヒント

BCD データをバイナリに変換します。BCD データは 4 ビットごとに 10 進表現で格納されていますから、4 ビットずつデータを取り出し以前の値を 10 倍した後、加算すれば求める値が得られます。その様子を図示すると以下ようになります。



36	SHLL.L ER2	;2 倍
37	MOV.L ER2,ER5	
38	SHLL.L ER2	;4 倍
39	SHLL.L ER2	;8 倍
40	ADD.L ER5,ER2	;8 倍+2 倍=10 倍
41	ADD.L ER4,ER2	;上位を足しこむ

上記は ER2 を 10 倍する命令です。10 倍を 8 倍と 2 倍の和として求めています。つまり

$10 \times ER2 = 2 \times ER2 + 8 \times ER2$ です。ロング型の場合、除算の命令を使うより効率的です。

バイナリを BCD データに変換

フォルダ BCD4

サブルーチン名

Bin2Bcd

引数

ER0 バイナリデータ
ER1 変換された BCD データを格納する領域の先頭アドレス

戻り値

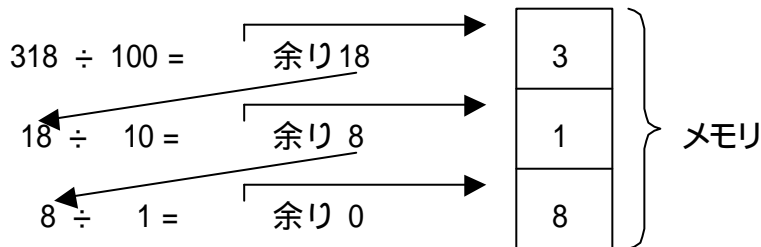
なし

動作

ER0 にあるバイナリデータを BCD データに変換して、結果をアドレス ER1 から始まる領域に格納します。

ヒント

バイナリデータを BCD に変換します。原理は次のとおりです。たとえばデータが 10 進で 318 だとします。100 の桁を求めるには 100 で割った商が 100 の桁になります。そのときの余り 18 を 10 で割ればその商が 10 の桁になります。



問題は割り算ですが、割り算は効率が悪いのでこのサンプルでは、引き算を繰り返し、何回引き算ができるかをカウントしています。たとえば $318 \div 100$ の場合、318 から 100 は 3 回引くことができます。4 回引くとマイナスになってしまいます。マイナスにならない引き算できる回数が商と等しくなります。以下の部分が商を求めるプログラムです。

```
29           ;引き算ループ  
30           ?SWHILE (ER0 >= ER4)  
31                 SUB.L  ER4,ER0  
32                 INC.B  R2H  
33           ?ENDW
```

0 で終了する文字列の処理

フォルダ名 STR1 ~ STR6

文字列のコピー

フォルダ STR1

サブルーチン名

StrCpy

引数

ER0 転送元文字列の先頭アドレス
ER1 文字列をコピーする転送先の先頭アドレス

戻り値

なし

動作

アドレス ER0 にある文字列をアドレス ER1 から始まるメモリ領域にコピーします。

ヒント

コンピュータ上で文字列は、アスキーコードのバイト列として扱われます。

例 Hello 'H' , 'e' , 'l' , 'l' , 'o'
 H'48 , H'65 , H'6C , H'6C , H'6F

しかし単なるバイト列としただけでは不十分です。なぜなら文字列の長さが分からないからです。上記の例では文字'o'の後に何かのデータがあった場合、それが文字列の続きかどうか判断できないからです。解決方法は2つあります。

1 文字列の最初に文字数を格納する

5	'H'	'e'	'l'	'l'	'o'
---	-----	-----	-----	-----	-----

2 文字列の最後に文字列の終わりを意味する 0 を格納する

'H'	'e'	'l'	'l'	'o'	0
-----	-----	-----	-----	-----	---

1の方法の長所は文字数がすぐ分かることです。短所は文字数に限界があることです。かりに文字数の格納領域が1バイトだとすれば255文字までしか扱えません。2の方法は文字数に限界がありません。しかし、0を忘れた場合に文字列処理プログラムが無制限に処理を続けてしまう恐れがあります。このサンプルではC言語でも採用されている2の方法をとることにします。

25 ?UNTIL(R2L != 0)

0で終わる文字列処理は上記のように読み込んだデータが0かどうかを終了条件になります。

文字列の連結

フォルダ STR2

サブルーチン名

StrCat

引数

ER0 連結される文字列の先頭アドレス
ER1 連結する文字列の先頭アドレス

戻り値

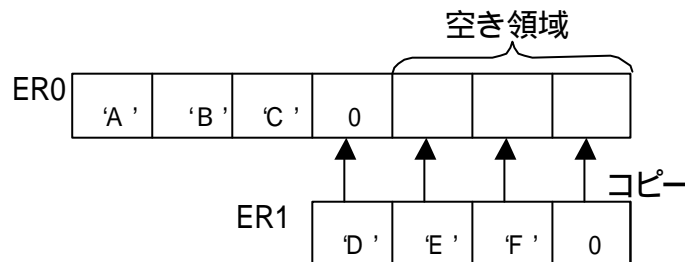
なし

動作

アドレス ER0 の文字列の最後に、アドレス ER1 の文字列を連結します。

ヒント

最初に、文字列 ER0 の最後の 0 を見つけます。そこから文字列 ER1 をコピーします。ER0 の文字列の後ろには連結される文字列の大きさ分の空き領域が必要であることを注意して下さい。



```
56                    segmentDATA  
57            str1:    DC.B    "Hello, ", 0  
58                    DS.B    5  
59            str2:    DC.B    "World", 0
```

文字列はダブルコーテーション(")でくくれば、アセンブラがアスキーコードの列に自動で変換します。最後の 0 は忘れずに付けて下さい。

58 行目は、文字列 str2 を連結するための空き領域の確保です。

文字列の比較

フォルダ STR3

サブルーチン名

StrCmp

引数

ER0	比較する文字列 1 の先頭アドレス
ER1	比較する文字列 2 の先頭アドレス

戻り値

R2L 文字列が一致した場合 R2L=0、一致しなかった場合は、最初に一致しなかった文字のアスキーコードの差が R2L に入る。

動作

アドレス ER0 の文字列 1 とアドレス ER1 の文字列 2 を比較します。比較の結果完全に一致した場合は R2L=0 が返されます。一致しなかった場合は、最初に一致しなかった文字のアスキーコードの差、(文字列 1-文字列 2)が R2L に格納されます。

ヒント

23 ?SWHILE (MOV.B @ER0+,R2L ,, MOV.B @ER1+,R2H ,, R2L==R2H)

?SWHILE の条件式 (R2L==R2H)の前には上記のようにニーモニックを複数記述することができます。ニーモニックの命令を実行してから、条件判定が行われます。またニーモニックはループの中に入っていますので何回も実行されます。ニーモニックおよび条件式はカンマ(,)を 2 つ並べて区切って下さい。

文字列の長さを得る

フォルダ STR4

サブルーチン名

StrLen

引数

ER0 長さを求める文字列の先頭アドレス

戻り値

R1 文字列の長さ、最後の 0 は長さに含まれない。

動作

アドレス ER0 の文字列の長さを R1 に格納します。文字列の最後の 0 は長さには含まれません。

ヒント

文字列の中の英小文字を英大文字に変換する

フォルダ STR5

サブルーチン名

StrUpper

引数

ER0 変換する文字列の先頭アドレス

戻り値

なし

動作

アドレス ER0 で示される文字列中の英小文字('a' ~ 'f')を英大文字('A' ~ 'F')に変換します。

ヒント

アスキーコードでは大文字に H'20 を加算すれば小文字になります。小文字から H'20 を減算すれば大文字になります。

```
22                   ?SIF (R1L >= 'a' && R1L <= 'z')
23                   ADD.B #'A'-'a',R1L
24                   MOV.B R1L,@ER0
25                   ?ENDI
```

23 行目の 'A' - 'a' は H'41-H'61 = -H'20 です。レジスタと数値の減算命令はないので、数値をマイナスにして加算命令を行う必要があります。

文字列の中の英大文字を英小文字に変換する

フォルダ STR6

サブルーチン名

StrLower

引数

ER0 変換する文字列の先頭アドレス

戻り値

なし

動作

アドレス ER0 で示される文字列中の英大文字('A' ~ 'F')を英小文字('a' ~ 'f')に変換します。

ヒント

アスキーコードでは大文字に H'20 を加算すれば小文字になります。小文字から H'20 を減算すれば大文字になります。

10 進文字列処理

フォルダ DEC1 ~ DEC2

10 進文字列をバイナリに変換する

フォルダ DEC1

サブルーチン名

Dec2Bin

引数

ER0 10 進文字列の先頭アドレス

戻り値

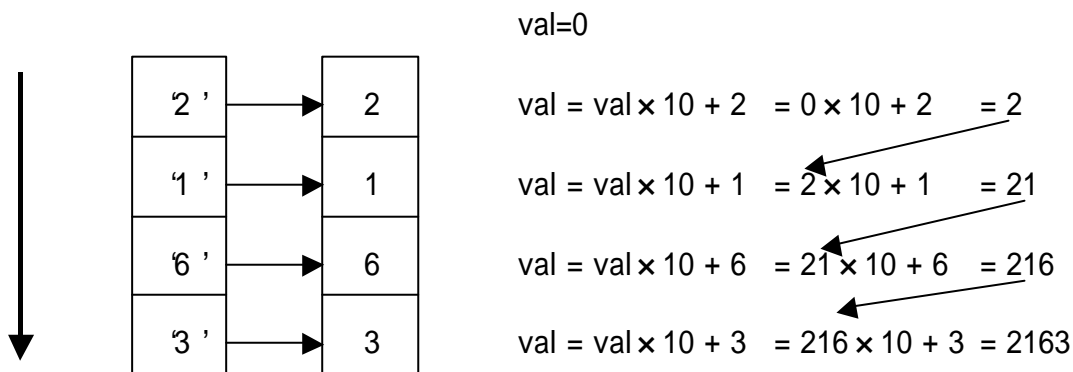
ER1 変換されたバイナリデータ

動作

アドレス ER0 で示される 10 進文字列をバイナリに変換して ER1 に格納します。

ヒント

文字列を先頭から走査し、アスキーコードからバイナリに変換します。次に、一つ前の値を 10 倍した後、加算すれば求める値が得られます。その様子を図示すると以下ようになります。



26	SHLL.L ER1	;2 倍
27	MOV.L ER1,ER3	
28	SHLL.L ER1	;4 倍
29	SHLL.L ER1	;8 倍
30	ADD.L ER3,ER1	;8 倍+2 倍=10 倍

上記は ER1 を 10 倍する命令です。10 倍を 8 倍と 2 倍の和として求めています。つまり

$$10 \times \text{ER1} = 2 \times \text{ER1} + 8 \times \text{ER1} \quad \text{です。}$$

ロング型の場合、除算の命令を使うより効率的です。

バイナリを 10 進文字列に変換する

フォルダ DEC2

サブルーチン名

Bin2Dec

引数

ER0 変換するバイナリ
ER1 変換された文字列を格納する領域の先頭アドレス

戻り値

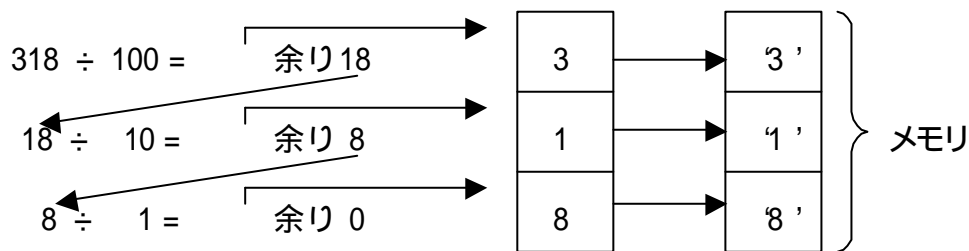
なし

動作

ER0 のバイナリの値を、10 進文字列に変換して先頭アドレスが ER1 で示される領域に格納します。文字列の格納領域は、桁数と最後の 0 の分の領域が必要です。

ヒント

バイナリデータを 10 進文字列に変換します。原理は次のとおりです。たとえばバイナリデータが 10 進で 318 だとします。100 の桁を求めるには 100 で割った商が 100 の桁になります。そのときの余り 18 を 10 で割ればその商が 10 の桁になります。このようにして各桁の数値を求め、'0'(H'30)を足しこんでアスキーコードに変換します。



問題は割り算ですが、割り算は効率が悪いのでこのサンプルでは、引き算を繰り返し、何回引き算ができるかをカウントしています。たとえば 318 ÷ 100 の場合、318 から 100 は 3 回引くことができます。4 回引くとマイナスになってしまいます。マイナスにならない引き算できる回数が商と等しくなります。以下に商を求めるプログラムの部分を示します。

```
29            ?SWHILE (ER0 >= ER4)  
30                    SUB.L  ER4,ER0  
31                    INC.B  R2H  
32            ?ENDW
```

16 進文字列処理

フォルダ HEX1 ~ HEX2

16 進文字列をバイナリに変換する

フォルダ HEX1

サブルーチン名

Hex2Bin

引数

ER0 16 進文字列の先頭アドレス

戻り値

ER1 変換されたバイナリ

動作

アドレス ER0 で示される 16 進文字列をバイナリに変換して ER1 に格納します。

ヒント

文字列を先頭から走査し、アスキーコードからバイナリに変換します。次に、一つ前の値を 16 倍した後、加算すれば求める値が得られます。原理は 10 進文字列の時と同じです。10 進文字列 バイナリ変換も参照して下さい。違いは 10 倍するところが 16 倍になります。

バイナリを 16 進文字列に変換する

フォルダ HEX2

サブルーチン名

BinHex

引数

ER0 変換するバイナリ
ER1 変換された文字列を格納する領域の先頭アドレス

戻り値

なし

動作

ER0 のバイナリの値を、16 進文字列に変換して先頭アドレスが ER1 で示される領域に格納します。文字列の格納領域は、桁数と最後の 0 の分の領域が必要です。

ヒント

フォルダ CONV4 で説明した、「1 バイトのバイナリを 2 個のアスキーコードに変換」するサブルーチンを利用します。このサブルーチンを上位バイトから 4 回呼び出して、ER0 の 4 バイトのデータを 16 進文字列にします。

```
121                    segmentDATA  
122            cr:            DC.B    "¥n",0
```

122 行目の ¥n は改行の印です。アセンブラはこの印を改行のアスキーコードである H'D に変換します。この文字列は C 言語の関数 fputs に送られますが、fputs 関数では、改行を出力する前に復帰コード H'A を出力します。したがって画面に出力されるコードは H'A、H'D の 2 バイトです。もし、C 言語の関数を利用しないでアセンブラで直接出力するなら、次のように記述します。

```
                          DC.B    "¥r¥n",0
```

¥r は復帰(H'A)の記号です。

ビット操作

フォルダ BIT1 ~ BIT3

レジスタの'1'が立っているビットの数を数える

フォルダ BIT1

サブルーチン名

BitHiCount

引数

ER0 ビットを数えるレジスタ

戻り値

R1L '1'が立っているビットの数

動作

ER0 の'1'が立っているビットの数を求め、R1L レジスタに格納します。

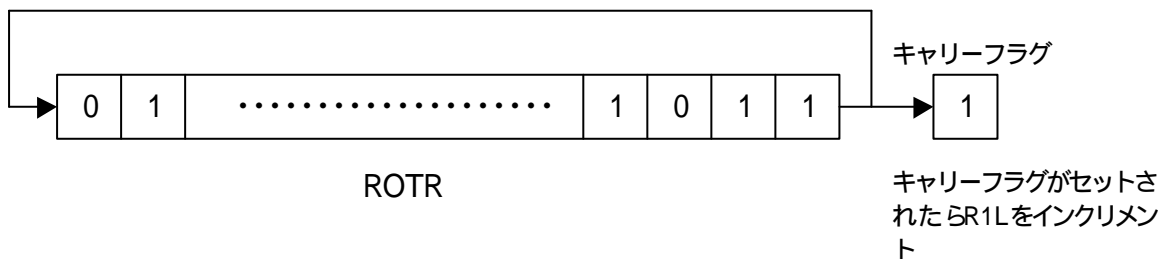
ヒント

レジスタの'1'が立っているビットの数を数えます。たとえば

ER0 = H'04230012 = B'0000 0100 0010 0011 0000 0000 0001 0010

の場合、'1'が立っているビットの数は6 になります。

方法は、ER0 を右にローテートします。そうすると最下位ビットがキャリーフラグに入ります。もしキャリーフラグがセットされていれば R1L をインクリメントします。これを 32 回繰り返せば'1'が立っているビットの数を R1L に求めることができます。



```
23           ROTR.L ER0
24           ?SIF (?FLAG_C)
25                   INC.B R1L
26           ?ENDI
```

上記の図をプログラムにしたのが、23 行目から 26 行目です。?SIF (?FLAG_C)はキャリーフラグがセットされている場合に実行する構造化命令です。

このサブルーチンは ER0 を PUSH/POP していませんが、32 回ローテートして結局元に戻るため PUSH/POP はしていません。

レジスタの値を 2 進文字列に変換する

フォルダ BIT2

サブルーチン名

Bin2Str

引数

ER0	バイナリ変換するデータ
ER1	変換された文字列を格納する領域の先頭アドレス

戻り値

なし

動作

ER0 のバイナリデータを 2 進文字列に変換して、先頭アドレスが ER1 で示される領域に格納します。文字列の格納領域は、桁数と最後の 0 の分の領域が必要です。

ヒント

ER0 を左にローテートして、最上位ビットをキャリーフラグに転送します。キャリーがセットされていればメモリ領域に '1'(H'31) を格納します。キャリーがクリアされていれば '0'(H'30) を格納します。最後に文字列の最後を表す 0 を格納します。

2 進文字列をバイナリに変換する

フォルダ BIT3

サブルーチン名

Str2Bin

引数

ER0 2 進文字列の先頭アドレス

戻り値

ER1 変換されたバイナリデータ

動作

アドレス ER0 で示される 2 進文字列をバイナリに変換して ER1 に格納します。

ヒント

最初に ER1 をクリアします。文字列を先頭から走査し、文字が'1'(H'31)であれば ER1 に 1 を加算します。次も同じことを繰り返しますが、一回ごとに ER1 を左にシフトします。4 ビットの場合の例を以下に示します。

