

基礎ソフト実験 UNIX・アセンブラ実習 第3回

実習内容

プログラムにバグがあるとき、レジスタやメモリの変化を観察したり、68000 エミュレータの機能であるブレークポイントを使って、バグを解決するための手がかりを得ることができます。今日の実習では、自分でアセンブラプログラムを入力し、そのプログラムのテスト実行とバグの発見、デバックを行います。

1. プログラムの入力

プログラムの入力には前回までの実習で使った `emacs` を使います。今回、実行するプログラムのファイル名は `fact.s` です。そこで、Terminal のウインドウ上で、「`emacs fact.s &`」 と入力しましょう。

```
$ emacs fact.s & <Enter キー>
```

`emacs` のウインドウが開くのを待ちます。

今回は、実際に下記のプログラムファイル `fact.s` を `emacs` を使って入力しましょう。このプログラムは、バグの発見とデバックを練習するために故意に間違いを入れてあります。ただし、この間違いは文法的な間違いではありません（気づいてもとりあえずそのまま入力してください）。

(注意) Pascal や C のような高級言語だと、たったの 1 行で済むような処理でも、アセンブラプログラムでは、何行にもなりプログラムも読みづらくなる。そのため、見やすい（読みやすくて分かりやすい）プログラムを心掛ける事が重要となる（アセンブラに限ったことではないが）。例えば、コメント文の活用、インデント（行頭の位置）の調節、シンボル名の付け方、Tab を使った単語の位置整列などを利用すると良い。

```
/* sample program fact.s 階乗 d1 := d0 ! (d0 >= 0) */

.section .text
start:
    moveq    #1,%d1
    cmp      #0,%d0
    beq      end_of_program
    move.l   %d0,%d2
loop:
    mulu     %d2,%d1
    subq     #1,%d2
    cmp.w    #0,%d2
    bge      loop
end_of_program:
    stop     #0x2700          /* 終了 */
```

入力が終わったらファイルを保存しましょう。 `emacs` 上で、

Ctrl-x Ctrl-s（上書き保存） または **Ctrl-x Ctrl-w**（別名で保存）

ファイルの中身を `less` などのコマンドで確認しましょう。

```
$ less fact.s <Enter キー>
```

課題 1. プログラム `fact` の以下の命令の意味を説明せよ。(3)については `sub` との違いを明確に示せ。

- (1) `mulu %d2, %d1`
- (2) `cmp.w #0, %d2`
 `bge loop`
- (3) `subq #1, %d2`

(注意) `bge` 命令, `subq` 命令と `sub` 命令の違いについては, 命令表を自分で調べてみることを。自分で積極的にプログラムを実行しながら試したり, 調べることも実習の一部です。

2. アセンブラ

前回の演習と同様に、`m68k-as` コマンドを使って、アセンブラソースプログラムファイル `fact.s` をアセンブルしましょう。ターミナル上で次のコマンドを実行してみましょう。

```
$ m68k-as -t 400 fact.s <Enter キー>
```

もし、文法的な間違いがある場合は、以下のようなエラーメッセージが出ます（あくまで例です）。その場合は、エラーメッセージを手がかりに `emacs` で `fact.s` ファイルを修正しましょう。

```
fact.s: Assembler messages:
fact.s:9: Error: parse error -- statement `cmp #0.%d0' ignored
fact.s:12: Error: Unknown operator -- statement `loop ' ignored
```

エラーメッセージがなかったら（なくなったら）、実行ファイルが出来ているか `ls` コマンドで確認しましょう。

```
% ls fact.* <Enter キー>
fact.LIS      fact.abs      fact.map      fact.s
```

（単に「`ls <Enter キー>`」と実行すると関係ないファイルも表示されるため、「`fact.`」で始まるファイルだけを表示させた）

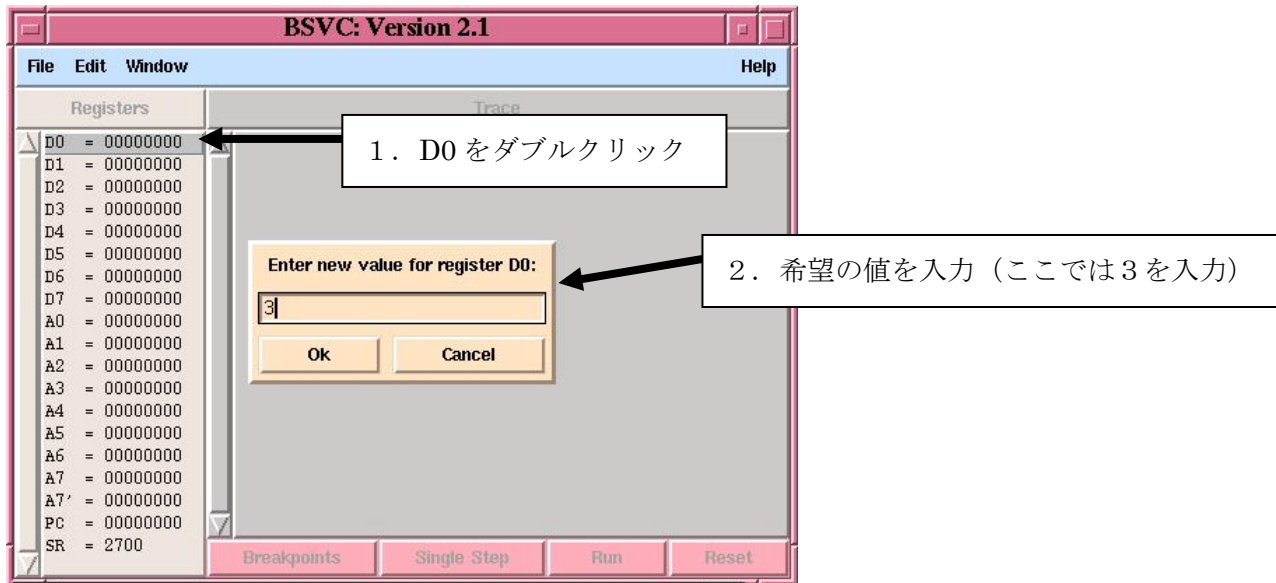
3. エミュレータ

次に、68000 エミュレータ(`m68k-emu`)を実行しましょう。

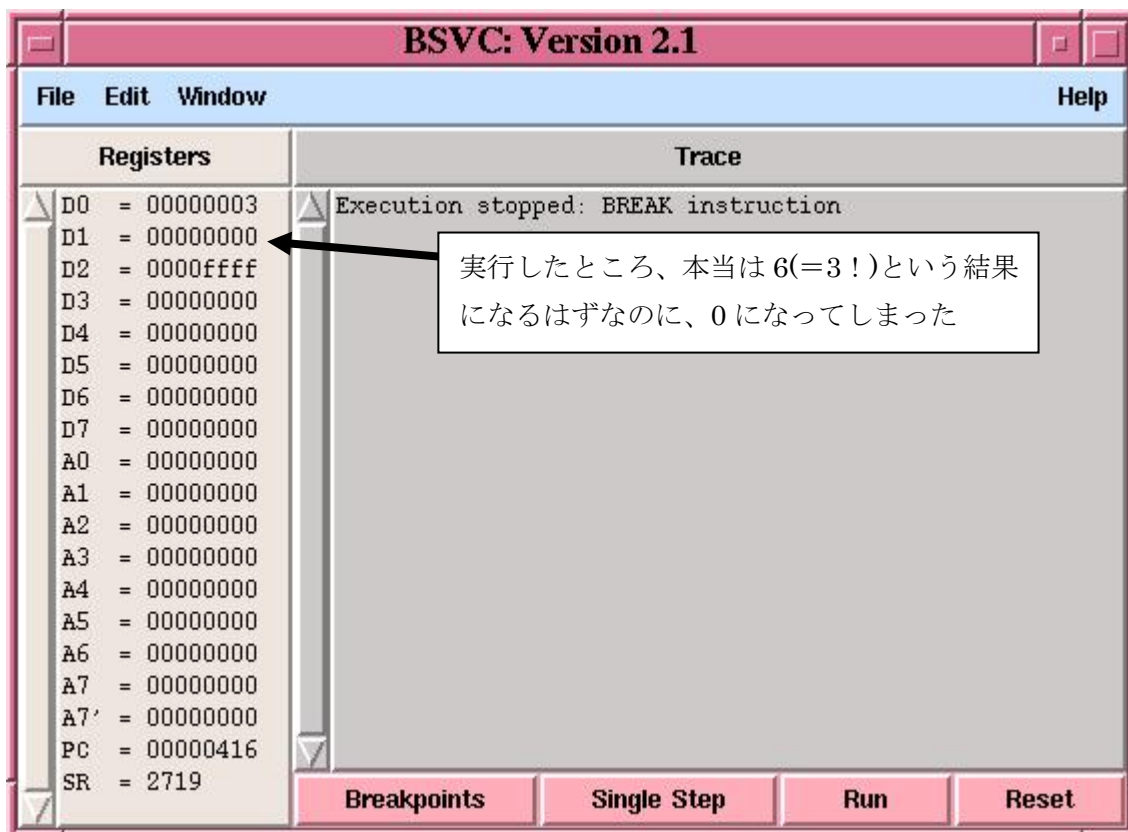
```
% m68k-emu & <Enter キー>
```

File メニューから、Load Program を実行し実行したいプログラムを選びます。ここでは、`fact.abs` を選択してください。Window メニューの、Memory Viewer, Program Listing を実行すると、アセンブラプログラムの実行の様子がモニターできます。

このプログラムは、レジスタ `d0` の値の階乗を計算しレジスタ `d1` に格納します。レジスタの値は画面の左にあるレジスタをダブルクリック（2度続けてマウスのボタンをたたく）し、値を入力することによって値が変更できます（下図）。3 の階乗を計算させて見ましょう。 `d0` に 3 を設定します。



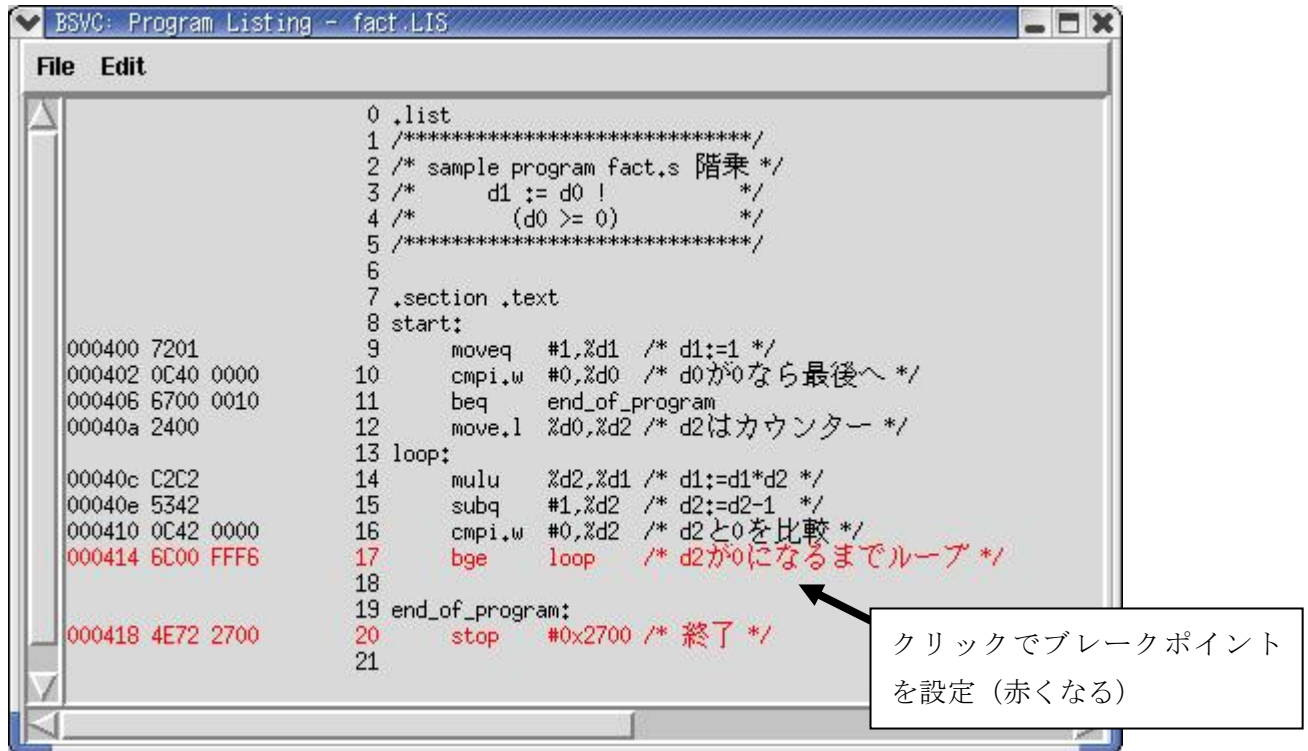
Run ボタンを押して実行してみましょう。計算結果が格納されるレジスタ D1 は 0 になってしまいました。プログラムが期待通りには動いていないことが分かりました。プログラム中に、まだバグが残っているようです。データレジスタ D1 が 0 になってしまう原因を、これから探していきましょう。



4. ブレークポイント

プログラムを実行する際に、実行途中でメモリやレジスタの値を確認したい場合には、ブレークポイントを利用すると便利です。ブレークポイントを設定すると Run ボタンを押したときにブレークポイントを設定した箇所で、プログラムは一時的に停止します。

ブレークポイントを設定してみましょう。エミュレータのメインウィンドウにある **Breakpoints** ボタンを押して番地を設定することもできますが、**Program Listing** のウィンドウでブレークポイントを設定したい命令をクリックする方が簡単です。ブレークポイントが設定されると、その命令箇所は赤い文字で表示されます。今回は、ループの分岐の際のレジスタを調べるために、ループの分岐命令(`bge loop`)をクリックしましょう。`bge loop` の行が赤くなりました。



Reset ボタンを押した後で **Run** ボタンを押して実行しましょう。ブレークポイントの位置で実行がとまりました。この時点でレジスタは

D0: 3 D1: 3 D2: 2

でした。続けて **Run** ボタンを 1 回ずつ押すと

D0: 3 D1: 6 D2: 1

D0: 3 D1: 6 D2: 0

D0: 3 D1: 0 D2: ffff

と変化しました。どうやら **D2** が 0 のときにもループしてしまい **D1** が 0 になってしまったようです。つまり、分岐の条件が間違っていたのです。

条件分岐の代表的なものをあげてみます (`cmp x, y` を実行したとき)。

<code>bge</code>	<code>bgt</code>	<code>beq</code>	<code>ble</code>	<code>blt</code>	<code>bne</code>
<code>y>=x</code>	<code>y>x</code>	<code>y=x</code>	<code>y<=x</code>	<code>y<x</code>	<code>y≠x</code>

`bge loop` を正しいものに置き換えて動かしてみましょう。

emacs でファイルを編集し、(終了していなければファイルを読み直しましょう)

\$ **emacs fact.s &** <Enter キー>

再度、アセンブラを実行し、

```
$ m68k-as -t 400 fact.s <Enter キー>
```

68000 エミュレータを実行しましょう。（終了していなければファイルを読み直しましょう）

```
$ m68k-emu & <Enter キー>
```

d0 が 3 のときに d1 は 6 になりましたか？

4!,2!,1!,0!についても試してみましょう。Reset ボタンを押し、d0 レジスタに値を入れた後で、Run ボタンを押してください。4 では d1 は 18 と表示されたと思います。これは 16 進数で表示されているため $16 \times 1 + 8 = 24 = 1 \times 2 \times 3 \times 4$ で正解です。

5. CCR (SR) と条件付分岐命令

今回のプログラムのサブルーチン中には、下のような条件付分岐命令が含まれています (1)。しかしながら、この条件分岐命令 `cmp` は必要ではなく、(2) のように省いても動作します。なぜでしょうか？

(1)

```
subq    #1,%d2
cmp.w   #0,%d2
bge     loop
```

(2)

```
subq    #1,%d2
bge     loop
```

この理由を理解するためには、まず `cmp` 命令の意味を理解する必要があります。`cmp` 命令は第 2 回の課題にも出てきていますが、ここで詳しく見てみましょう。実は、この比較する命令は、実際にはデスティネーションオペランドからソースオペランドを減算しているのです（命令表 p.310 を必ず見ること）。そして、その演算結果は CCR（コンディションコードレジスタ）に反映されます。この CCR とは、SR（ステータスレジスタ）の下位 8 ビット（実際に使うのは 5 ビット）のことで、エミュレータメインウィンドウ左のレジスタ一覧にも表示されています。

つまり、大小比較のため減算をし、その結果が負ならば CCR の N (Negative) のビットを 1 にし、結果が 0 ならば CCR の Z (Zero) のビットを 1 に設定しているのです。もちろん、結果が正の場合は、CCR の N と Z のビットは 0 のままとなります。こうすることで、CCR を見れば、直前に行われた比較（減算）結果がどうであったかが分かる仕組みになっている訳です。**cmp 命令が sub 命令と違う点は、減算した結果がデスティネーションオペランドには格納されないという点だけであり、sub 命令も減算の結果、cmp 命令と同様に CCR に影響を及ぼします。**

次に、条件付分岐命令 `bcc` (cc には不等号条件が設定される)（命令表 p.372 を必ず見ること）の意味について見てみましょう。この命令は、CCR に応じて必要なロケーションへ制御を移行させるものです。そのため、本例の場合、レジスタ d2 から 1 を減算した結果が 0 でなければ、CCR の Z, N のビットが 0 となり、`bge` 命令はその CCR を参照した結果、`loop` に制御を移行させます。そのため、(2) のプログラムのように、`sub` 命令と `bge` 命令の間の `cmp` 命令を省くことができるのです。

この一連の流れを確認するために、分岐命令の箇所（１）を（２）のように変更し、条件分岐命令付近での CCR（実際には SR の下位 5 ビット）の変化を確認しましょう。

課題 2. 分岐命令（繰り返し）を使って次の計算を行うプログラムを作成せよ（プログラムがきちんと動作することをスタッフに確認してもらうこと）。出席表には、その計算結果のみを記載せよ。

(1) ${}_{10}P_5$

${}_aP_b$ のとき、a はレジスタ d1, b はレジスタ d2 で与えられ、計算結果はレジスタ d0 に格納する仕様とせよ。

(2) $\sum_{k=1}^5 k^k$

$\sum_{k=1}^n k^k$ のとき、n はレジスタ d1 で与えられ、計算結果はレジスタ d0 に格納する仕様とせよ。

今日の実習はここまでです。