計算機 SYSTEM Report 2

Sho Iwamoto*

2005 年 12 月 8 日

1 概要と目的

1.1 概要

1. 以前に作成した関数 **mystrlen** を用いて，C における strcat を作成する。
2. 多倍長整数の加算を行う関数 **ex_add** を作成する。
3. 関数 max2[*1]を用いて，4 つの int 値に対してその最大値を返す関数 **max4** を作成する。

1.2 目的

Assembly 言語において作成した関数の中で更に別の関数を呼び出す，という動作を実装し，stack を用いた assembling を行う。

更に，多倍長計算を行う関数 **ex_add** を作成する。

2 文字列処理関数 mystrcat（再）

2.1 方法

下準備

まず，関数 **mystrcat** を呼び出すための C program を準備した（**call-mystrcat.c**）。前回使った **call-mystr.c** をほとんどそのまま[*2]流用している。

次に，実行 command を準備した。但し，今回は前回とは違い，GNU make を用いるようにした。詳しい利用方法については，Appendix A を参照して欲しい。

mystrcat.s : assembly file

以下のような assembly を組んだ。

---

* g450318 理学部物理学科 [id:Misho]

*1 2 つの int 値を比較して最大値を返す関数。以前に作成してある。

*2 ただし，前回 scanf を用いて処理していた部分は全て fgets で処理するよう改良している。

```
1       .text
2             .align 2
3             .globl _mystrcat
4
5       _mystrcat:
6             # Retreat & Stack-frame
7             mflr r0
8             stw r0, 8(r1)
9             mfcr r0
10            stw r0, 4(r1)
11            stwu r1, -64(r1)
12            # parameter area
13            stw r3, 88(r1)
14            stw r4, 92(r1)
15
16            # Main
17            bl _mystrlen
18            mr r7,r3
19            lwz r3, 88(r1)
20            lwz r4, 92(r1)
21            sub r4,r4,r7
22      CAT_CONNECT:
23            lbzx r5,r4,r7
24            stbx r5,r3,r7
25            cmpi cr7, r5, 0
26            beq cr7,RETURN
27            addi r7,r7,1
28            b CAT_CONNECT
29
30      RETURN:
31            # Return
32            la r1, 64(r1)
33            lwz r0, 4(r1)
34            mtcr r0
35            lwz r0, 8(r1)
36            mtlr r0
37            blr
```

Code 自体はとても単純であり，特にここで説明すべきこともないと思うので，簡単に流れを述べるにとどめておく。

○1〜11 **行目**　いわゆる変数の宣言など，処理の前準備段階である。

○13〜14 **行目**　関数 **mystrlen** の呼び出しを行うため，予め全ての引数を memory に退避させた。

○16〜20 **行目**　その後，**mystrlen** を呼び出し， 退避させた引数を復帰させた。

○21〜28 **行目**　ここの処理は，前回のものと全く同じである。詳しくは前回の report[*3]を参照のこと。

○32〜37 **行目**　Stack frame 破棄の後，この関数の最初で退避させた registers を復帰させて，関数を閉じている。

2.2 結果

以下に実行結果の 1 例を記す。

```
** ComSys Report 2-A : 2005/11/18(Fri) **
[450318C Sho Iwamoto]

MaxLength
? 10
```

[*3] http://www.misho-web.com/univ/files/comsys051117.pdf にも掲載してある。

```
NOTE: String must be 'one-line string'(no CR/LF)
Str1 (length =< 10)
 -> Mijikai b
Str2 (length =< 10)
 -> un dana.

Raw data : str1='Mijikai b',str2='un dana.'
MYstrcat -> Mijikai bun dana.

** END **
```

まぁ，main routine 自体は前回作ったものと全く同じなので，実行結果について特筆すべきことはないだろう。

2.3 考察

■Stack の扱い 今回の課題の point は，stack の扱いであった。というわけで，今回はそこに注目して説明する。

mystrcat 自身が確保した stack frame は 64byte であり，**mystrcat** はその部分には全く手を付けていない。

8 行目と 10 行目で，link register(LR) と condition register(CR) を，親関数の linkage area に保存している。Linkage area は 4byte の領域 6 個分であり，LR と CR はそれぞれその 3 番目と 2 番目に置くように定められているので，このような動作となっている。

そして，13〜14 行目で，引数を親関数の parameter area に保存している。

以上のことを表 1 にまとめておいた。

表 1 mystrcat における stack の利用

0 (-64)		
8 (-56)		
⋮	⋮	⋮
40 (-16)		
48 (-8)		
56 (0)		Condition Register
64 (8)	Link Register	
80 (16)		
88 (24)	第 1 引数 (r3)	第 2 引数 (r4)
96 (32)		

2.4 今後の課題

特にない。と言うか，前回述べたとおりである。

3 多倍長の加算 ex_add

下準備

関数 **ex_add** を呼び出すための C program を準備した（**call-exadd.c**）。いつものことだが，この program の作成に最も時間を費やしてしまった。

というわけで，またもや余談としてこの program について軽く触れておこう。

この program は，入力に 10 進数と 16 進数を許してある。というか，10 進数を許す処理を組むのに時間がかかった。しかし，やはり「大きい数の計算をしたい！」という欲望は，10 進数の数に対して生じるものであり，16 進数しか許容しないような program をそこに用意してもらったとしても，その欲望を満たすには不十分なのだ。

10 進数を 16 進数に直す部分は，かなり spaghetti になってしまった。というか，spaghetti な上に，冗長で非効率的なのだからどうしようもない。一応いくつか改善案は思いついたが，それを実装すると spaghetti から綾取りのヒモへと更に進化しそうだったし，なによりこの課題は C の課題ではないので，このような適当な program のままで submit することになってしまった[*4]。**もっと綺麗で高速な処理を思いついた人がいたら，教えて欲しいものだ……。**

なお，getValue については，上述の拡張をするために完全に自作したが，printValue については予め用意されていた code を参考にした。というかほぼあれと同じである[*5]。

また，想定以上の桁数を入力された場合には，全 bit を 1 で埋める（FFFFFFF……FFF）ようにしてある[*6]。

mystrcat.s : assembly file

以下のような assembly を組んだ。

```
1      .text
2            .align 2
3            .globl _ex_add
4      _ex_add:
5            # Linkage / stack frame
6            mflr r0
7            stw r0, 8(r1)
8            mfcr r0
9            stw r0, 4(r1)
10           stwu r1, -64(r1)
11
12           li r9,0
13     main:
14           subi  r3, r3,1
15           lbzx  r7, r3,r4
16           lbzx  r8, r3,r5
17           add   r7, r7,r8
18           add   r7, r7,r9
19           stbx  r7, r3,r6
20           srawi r9, r7,8
21           cmpi cr7, r3,0
22           bgt  cr7, main
23
24           # Return
25           la r1, 64(r1)
26           lwz r0, 4(r1)
27           mtcr r0
28           lwz r0, 8(r1)
29           mtlr r0
30           blr
```

[*4] あー恥ずかしい。
[*5] あれ以外の処理を思いつきそうにない。
[*6] atoi の実装と同じである。

実は，この code では，addc も adde も使用していない。単純な加算と bit shift だけで処理をしている。これは，計算を byte 単位で行ったためである。

これまでの授業の流れから察すると，やはりここはどうしても word 単位での加算処理をすべきであるとは思った。しかし，引数が unsigned char 型で指定されている中で，それをわざわざ 4 つずつくくって word で処理して，更に最上位の部分の加算を別 routine で実行する，というような処理はしたくなかったのだ。

おそらく想定とは若干違う code になっているかもしれないが，そういうわけなので了解願いたい。

■1〜10，24〜30 行目 「いつもの処理」である。今回は内部で stack を全く使用していないため，stack は 64 個確保になっている。

■12 行目 r9 は，繰り上がる数（とはいえ byte 単位の演算だが）である。carry bit の役割を果たしていると考えればいい。

■14 行目 r3 は，桁数として与えられた引数である。

r4，r5 が，それぞれの数の最上位の address を示しているので，最下位の address は

$$\mathbf{r4} + (\mathbf{r3} - 1), \qquad \mathbf{r5} + (\mathbf{r3} - 1)$$

となる。そのため，いきなり r3 から 1 を減じている。

counter を 1 ずつ減じればよいところが，byte 単位での処理の最大の利点であろう。

■15〜18 行目 処理すべき位の数字をそれぞれ r7，r8 に byte 単位読みだしして，互いを加算し，更に carry（r9）を加算している。

■19 行目 ここで store byte している点に注意して欲しい。この処理により，和である r7 のうち下位 8bit 分が store されている。

■20 行目 19 行目で store されなかった部分を繰り上げ処理に回すため，r7 を 8 つ right shift して，それをそのまま carry へ放り込んでいる。

ここで，immidiate な shift を行わせたかったので srawi 命令を利用した。Algebraic ではあるが，しかし byte 単位の演算なので最上位は明らかに 0 であり，その点を気にする必要は無い。

■21〜22 行目 最上位の桁の処理（r3 が 0 の場合の処理）が終われば，処理は全て終了である。

3.1 結果

一般的な場合のいくつかの実行結果を載せておく。

```
** ComSys Report 2-B : 2005/12/07(Thu) **
[450318C Sho Iwamoto]

Bytes
? 4
```

```
Please input two-values.
[if started with '0x', calculated as unsigned hexadecimal.]

Num-1
 -> 0x88888888
Num-2
 -> 0x789abcde

  a = 88888888
  b = 789ABCDE
a+b = 01234566

** END **
```

```
** ComSys Report 2-B : 2005/12/07(Thu) **
[450318C Sho Iwamoto]

Bytes
? 2

Please input two-values.
[if started with '0x', calculated as unsigned hexadecimal.]

Num-1
 -> 400
Num-2
 -> 20

  a = 0190
  b = 0014
a+b = 01A4

** END **
```

```
** ComSys Report 2-B : 2005/12/07(Thu) **
[450318C Sho Iwamoto]

Bytes
? 6
```

```
Please input two-values.
[if started with '0x', calculated as unsigned hexadecimal.]

Num-1
 -> 5151321549512
Num-2
 -> 21589654654444

  a = 04AF62B0EAC8
  b = 13A2BBB24DEC
a+b = 18521E6338B4

** END **
```

やはり 10 進数のでかい数の足し算がスパッと決まると気分が良い。

3.2 考察

演算単位の問題

これを word 単位でやるとどのようになるのか，という点が最大であろう。

引数が byte である以上，code そのものはややこしくなるが，加算処理が $\frac{1}{4}$ 回程度になるため，処理速度はかなり早くなるだろう。

引数を word にすると，word 単位の演算にしても code は綺麗なままだが，しかし汎用性に欠ける。

ということは，やはり，byte を引数とした，内部が word 単位の演算であるような code が一番すばらしいと言うことになるのだろうか。

拡張性について

実は，減算処理も作ろうかと思った。おそらく，r8 の値を 2 の補数にしてやればすぐにできる。次のような感じだろうか。

```
1         li r9,1
2         li r10,255
3   main:
4         subi  r3, r3,1
5         lbzx  r7, r3,r4
6         lbzx  r8, r3,r5
7         sub   r8,r10,r8
8         add   r7, r7,r8
9         add   r7, r7,r9
10        stbx  r7, r3,r6
11        srawi r9, r7,8
12        cmpi cr7, r3,0
13        bgt  cr7, main
```

と思って組んだら，ものの 5 分であっさり出来てしまった。というわけで，おまけとして付け加えてある。ただし，十分に debug は行っていないので，変なことをしたら変なことになるかも知れない。。。。

4 最大値関数 max4

4.1 方法

max4.s : assembly file

以下のような assembly を組んだ。[*7]

```
1       .text
2             .align 2
3             .globl _max4
4       _max4:
5             # Linkage / stack frame
6             mflr r0
7             stw r0, 8(r1)
8             mfcr r0
9             stw r0, 4(r1)
10            stwu r1, -64(r1)
11
12            # Main
13            stw r5, 96(r1)
14            stw r6, 100(r1)
15            bl _max2
16            lwz r4, 96(r1)
17            bl _max2
18            lwz r4, 100(r1)
19            bl _max2
20
21            # Return
22            la r1, 64(r1)
23            lwz r0, 4(r1)
24            mtcr r0
25            lwz r0, 8(r1)
26            mtlr r0
27            blr
```

なんとも短い code である。冒頭と末尾はこれまでと同様なので割愛する。

■13〜14 **行目**　まず，第 1 引数 (r3) と第 2 引数 (r4) を **max2** で比較させる。
そのため，引数の退避を行った。**r5** と **r6** がそれぞれ第 3・第 4 引数であり，これを退避させている。
但し，第 1 引数と第 2 引数は（**max2** の処理後は）破棄して構わないので，退避を行わなかった。

■15 **行目**　**max2** の処理を行わせる。**r3** に，第 1 引数と第 2 引数の **max2** の結果が返される。

■16〜17 **行目**　先ほど退避させた第 3 引数を，**r4** に読み込んで，**max2** に渡している。

■18〜19 **行目**　2 度目の **max2** で返された方と，第 4 引数（**r4** に読み込んだ）を比較した。
19 行目の処理で返される返り値 **r3** が，4 つの引数の内の最大値である。

4.2 結果

別に記さなくてもいいかとは思うのだが，一応以下に実行結果の 1 例を記しておく。

```
** ComSys Report 2-C : 2005/11/18(Fri) **
[450318C Sho Iwamoto]
```

[*7] 下準備については，特に記載すべきことが見あたらなかったのでここには記さないでおく。

```
Please input four-values[INTEGER!].
Num-1 ? -6
Num-2 ? -5
Num-3 ? -1
Num-4 ? -2
max(-6,-5,-1,-2) = -1

** END **
```

4.3 考察

今回の stack 利用をまとめると，表 2 のようになる。

表 2　max4 における stack の利用

0 (-64)		
⋮	⋮	⋮
48 (-8)		
56 (0)		Condition Register
64 (8)	Link Register	
80 (16)		
88 (24)	第 1 引数	第 2 引数
96 (32)	第 3 引数 (r5)	第 4 引数 (r6)
104 (40)		

これは，**mystrcat** とほぼ同じ使い方であるが，第 3 引数と第 4 引数しか保存していない点が特徴的である。第 1・第 2 引数を入れるべき場所は，reserved として空けておいた。

今回は，第 3・第 4 引数を引数と見なして parameter area に退避させたが，単純に変数と見る見方もできるので local variables area か saved registers area に退避させてもよかった。一体どちらがいいのだろう？

4.4 今後の課題

- 関数の拡張として，任意数個の int 値を比較する関数 intn が作れそうである。
- 考察の最後に述べた，register の退避位置の問題を解決したい。

付録 A

Make の使い方

1 実行 file の作成

全体を全て make するには，単純に

```
make
```

すればよい（make all でも処理できる）。
また，それぞれの code を全て make するために，以下の command を用意した。

```
make mystrcat
make exadd
make max4
```

2 作成の後に実行

実行 file を生成してそれを即座に実行するために，以下の command を用意してある。

```
make exec_mystrcat
make exec_exadd
make exec_max4
```

3 おまけ

多倍長の減算を行う関数も利用可能である：

```
make exsub
make exec_exsub
```

それから，一応

```
make clean
```

も用意しておいた。