計算機 SYSTEM Report 1

Sho Iwamoto*

2005 年 11 月 17 日

1 概要と目的

1.1 概要

1. 3 つの int 値に対して，その最大値を返す関数 max3 を作成する。
2. C における strcmp および strcat を作成する。

1.2 目的

Assembly 言語による基礎的な関数を作成して，computer program の基本が assembly 言語にあることを実感し，assembly 段階での各関数の振る舞いを観察する。

同時に，assembly 言語における，loop，条件分岐，および基本的な memory acce ss の扱い方を把握する。

2 最大値関数 max3

2.1 方法

call-max3.c：**呼び出し元** C program

まず，関数 max3 を呼び出すための C program を記述した (**call-max3.c**)。この program はとても単純なものであるため，詳細に関しては省略する。ちなみに，ここで関数 max3 は

```
1    extern int max3(int x, int y, int z);
```

と定義した。

exec_max3：**実行** command

次に，debug などを容易にするために，**call-max3.c** と **max3.s** を gcc で compile，その後に作成された program **max3** を実行する command を作成した。詳細に関しては省略する。

* g450318 理学部物理学科 [id:Misho]

max3.s : assembly file

以下のような assembly を組んだ。

```
1       .text
2             .align 2
3             .globl _max3
4
5       _max3:
6             # Retreat
7             mflr r0
8             stw r0, 8(r1)
9             mfcr r0
10            stw r0, 4(r1)
11
12            # Stack-frame
13            stwu r1, -64(r1)
14
15            # MAIN
16            cmp cr7,r3,r4
17            bge cr7,NEXT      # OK->NEXT
18            mr r3,r4
19      NEXT:
20            cmp cr7,r3,r5
21            bge cr7,RETURN
22            mr r3,r5
23      RETURN:
24            la r1, 64(r1)
25            lwz r0, 4(r1)
26            mtcr r0
27            lwz r0, 8(r1)
28            mtlr r0
29            blr
```

■1,2 行目　この部分はよく解らなかった。特に**.align** の部分がわからなかったので，今後調べたい。

■3 行目　ここで global な関数 max3 を宣言している。

■5〜13 行目および 24〜29 行目　Link register と condition register の値を待避させた後に stack frame を確保する動作，および，その逆の動作（stack frame 破棄，各値を復帰）を行う部分である。**詳細はよく解らなかった**ので，講義資料に書かれたものをそのまま利用している。

■15〜22 行目　この部分が関数の中心である。

基本的には，まず **x**(=**r3**) と **y**(=**r4**) を比較し，大きい方の値を **x** に代入し，その後 **x** と **z**(=**r5**) を比較して大きい方を **x** に代入するという動作をしている。この **x** がそのまま戻り値となる。

細かく見れば，16 行目で **x** と **y** の比較結果を **cr7** に代入して，次に 17 行目で，それが **x** ≧ **y** という結果であれば何もせずに次の処理 **NEXT** へと移る。そうでなければ 18 行目で **x** に **y** を代入して，**x** ≧ **y** となるようにしている。

同じことを **x** と **z** に行えば，求める処理は完了である。

2.2　結果

以下に実行結果の 1 例を記す。

```
8:g450318@ca11501:~/Documents/Comsys/20051104/report1:% ./exec_max3
** ComSys Report 1-A : 2005/11/17(Thu) **
```

```
[450318C Sho Iwamoto]

Please input three-values[INTEGER!].
1st ? 412
2nd ? -533
3rd ? 0
max(412,-533,0) = 412

** END **
```

参考までに，他にいくつかの結果を示しておく。正しく実現されていることが分かる。特に，最大値が **x**,**y**,**z** のいずれであっても正しく動作していることに注目して欲しい。

```
max(-2,5,0) = 5
max(-8,0,3) = 3
```

2.3 考察

Source を見る限りでは，どのような 3 つの int 値に対しても正しい結果を返せそうである。特に，加算も減算も行っていないので，overflow が起こらないという点は大きい。

また，比較においては往々にして，正負の値を扱うときに error が生じやすいが，本関数では，比較に用いている **cmp** が正負について正しく動作する以上，そのような問題は生じないと考えられる。

実際，実行結果を見ても特に問題は生じていないようである。

以下，おまけとして，（一般の人が）気になるであろう点を挙げる。

入力値が非整数の場合

この場合，

```
** ComSys Report 1-A : 2005/11/17(Thu) **
[450318C Sho Iwamoto]

Please input three-values[INTEGER!].
1st ? 6.7
2nd ? 3rd ? max(6,0,0) = 6

** END **
```

などのようになるが，これは c 側の scanf 関数による読み込み時の問題である。

では，関数 max3 に直接 float 値などを与えたらどうなるのであろうか。これについては（c からの）上手い呼び出し方法が思いつかなかったので，**実験を行っていない**[*1]。おそらく，例えば int と同じ長さのもので

[*1] どうやら，C も同時に勉強しなければならないらしい……。

あれば，それを int 型と解釈したときの値（char なら ascii code）として処理されるであろう。しかし，長さが違うもの（double など）ではどういう振る舞いを示すのか，全く想像ができない。今後実験してみたい。

2.4 今後の課題

- Assembly code の 1,2 行目の意味
- 同じく 5〜13 行目 & 24〜29 行目の動作の意味[*2]
- 関数 max3 に直接 float 値などを与えた場合の動作

3 文字列処理関数 mystrcmp，mystrcat

3.1 方法

call-mystr.c：**呼び出し元** C program

先ほどと同様に，関数 mystrcmp，mystrcat を呼び出すための C program を記述した (**call-mystr.c**)。

話が脇道にそれてしまうが，余談としてこの program について軽く触れておこう。

この program を実行するとまず，扱おうとする 2 つの文字列の長さを尋ねられる。この値を元にして，文字列の領域を確保している。[*3][*4]

その後，2 つの文字列の入力が求められる[*5]。'one line' という制限はこの c program 上のものである。Assembly 言語で書いた関数は，改行も同様に解釈できるはずである**（実験していない）**。実は，この課題の中で最も苦労したのが，この部分であった。c を始めて数週間目の私としては，かなり高度な program を組んだつもりであるが，もっと上手い実装があるかもしれないので，**改良の余地はあるだろう**[*6]。

なお，実はこの部分に文字列の長さを取得している処理があるのだが，折角なのでこれも mystrlen として assembly code の中で定義している。

入力が終わると，順に mystrlen，mystrcmp，mystrcat の実行結果が表示される。mystrcmp の部分では，c の strcmp 関数の出力も並行して表示してある。それ以外の関数については，実行結果を見れば明らかであろう。

なお，それぞれの関数は以下のように定義してある。

```
1    extern int mystrlen(char*);
2    extern int mystrcmp(char*,char*);
3    extern char* mystrcat(char*,char*);
```

exec_mystr：**実行** command

先ほどと同様の実行 file を作成した。なお，作成される program 名は **mystr** である。

mystr.s：assembly file

次頁のような assembly を組んだ[*7]。

[*2] 各 mnemonic の意味についてはわかっているのだが……。

[*3] (領域長)=(文字列長)*2 + 1 である。これは文字列末の null 文字，および strcat での連結を考慮したものである。

[*4] ここで，可変長配列の宣言を用いているが，これは c99 で拡張されたものであり，ANSI では動かない。

[*5] ここで文献 [1, p.319 他] を参考にした。

[*6] 詳しい人がいたら comment して欲しい……。

[*7] 綺麗に 1page に収まっているのは偶然である。

```
.text
      .align 2
      .globl _mystrlen
      .globl _mystrcmp
      .globl _mystrcat
RETURN:
      la r1, 64(r1)
      lwz r0, 4(r1)
      mtcr r0
      lwz r0, 8(r1)
      mtlr r0
      blr


_mystrlen:
      # Retreat & Stack-frame
      mflr r0
      stw r0, 8(r1)
      mfcr r0
      stw r0, 4(r1)
      stwu r1, -64(r1)
      # MAIN
      mr r4,r3
      li r3, 0
LEN_LOOP:
      lbzx r5,r4,r3
      cmpi cr7, r5, 0
      beq cr7,RETURN
      addi r3,r3,1
      b LEN_LOOP

_mystrcmp:
      # Retreat & Stack-frame
      mflr r0
      stw r0, 8(r1)
      mfcr r0
      stw r0, 4(r1)
      stwu r1, -64(r1)

      #MAIN
      mr r5,r3
      mr r6,r4
CMP_LOOP:
      lbz r3,0(r5)
      lbz r4,0(r6)
      sub r3,r3,r4
      cmpi cr7,r3,0
      bne cr7,RETURN
      cmpi cr7,r4,0
      beq cr7,RETURN
      addi r5,r5,1
      addi r6,r6,1
      b CMP_LOOP

_mystrcat:
      # Retreat & Stack-frame
      mflr r0
      stw r0, 8(r1)
      mfcr r0
      stw r0, 4(r1)
      stwu r1, -64(r1)

      #MAIN
      li r7,0
CAT_LOOP:
      lbzx r5,r3,r7
      addi r7,r7,1
      cmpi cr7, r5, 0
      bne cr7, CAT_LOOP
      subi r7,r7,1
      sub r4,r4,r7
CAT_CONNECT:
      lbzx r5,r4,r7
      stbx r5,r3,r7
      cmpi cr7, r5, 0
      beq cr7,RETURN
      addi r7,r7,1
      b CAT_CONNECT
```

■max3.s を流用した部分　1〜14 行目，および各関数の冒頭部については，先ほどの説明と全く同じであるので省略する。なお，**RETURN** の部分は各関数全く同じ動作であるので，1 つにまとめてある。Retreat & Stack-frame の部分は，**まとめる方法が解らなかった**。Link register が絡む部分なので，この部分の subroutine 化は難しそうである。

■22〜30 行目　mystrlen 関数の部分である。基本的には，文字列を 1 文字ずつ読み込んでそれが null ならその時の値（何文字目か）を返す，という動作である。

○23,24 行目　戻り値を **r3** に設定したかったので，**str** の address **ad** を **r4** に退避させて，何文字目を読んでいるのかを示す変数 **n** を **r3** に設けた。

○26〜28 行目　**ad** の **n** 個先の文字を，**r5** に読み込んでいる。つまり，この段階では **n**+1 文字目を読んでいるのである。

そして，これが null 文字であったら **RETURN** に飛ばして終了となる。このとき，**n**+1 文字目が null であるので，即ち **n**=**r3** の値は確かに文字数になっていることに注意しておく。

○29〜30 行目　文字列が終わらないのなら，**n** を increment して，再び同じことをやらせるのである。

■40〜53 行目　mystrcmp 関数の部分である。これは，最初の文字を見比べて，異なっていれば「違う！」と返し，同じであればその次の文字を見比べて……，という感じである。そして，最終的に文字列の末端の null の位置まで同じなら「同じ！」と返すのである。

mystrlen では，文字列の seek 方法として「開始位置の○文字先」という方法を用いたが，今回は開始位置の値自体を increment していることに注意して欲しい。

○41,42 行目　**r3** と **r4** を作業領域に使いたかった（戻り値が **r3** だし）ので，**str1**，**str2** の address **ad1**，**ad2** をそれぞれ **r5** と **r6** に移してやる。

○44〜46 行目　先ほども述べたが，**r5**，**r6** 共に，「読み込む文字の位置」を直接指定していることに注意しておく。まず，**r3** と **r4** に，それぞれ文字を読み込む。そして，その文字の ascii code の差を取って **r3** に入れる。ascii code は alphabet に関しては辞書順[*8]であるので，これが戻り値となるのである。

○47〜48 行目　ただし，これをそのまま返すわけにはいかない。1 文字目が違っていればそれでいいが，1 文字目が同じならその次の文字を比べなければならない。この部分は，「違っていればそれでいい」の部分である。差 **r3** を，即値 0 と比較して，違っていれば **RETURN** で終了となる。

○49〜50 行目　現在読んでいる文字の比較結果が「同じ」であっても，処理を終了して良い場合がある。2 つの文字列が完全に一致している場合である。これは，最後の null 文字までが一致しているということを意味する。つまり，2 つの文字が同じ[*9]で，更にそれが null の場合にその文字列が「同じ」であると言えるのだ。**r4** は，**str2** から読み込んだ文字の ascii code がそのまま残っている。これが null，即ち 0 なら，**RETURN** で終了となる。戻り値は，先ほどの差の計算結果である 0 となる。

○51〜53 行目　読み込む位置を 1 つ increment して，再び繰り返させる部分である。

■63〜78 行目　mystrcat 関数の部分である。**str1** の末尾にある null の位置以降を，**str2** で置き換えてやれば良いだけのことである。戻り値は，**str1** の開始 address をそのまま，である。

なお，**str1** に十分に領域が確保されてあるかどうかは確認しない[*10]。

[*8] Original の strcmp では，辞書順を”0-9A-Za-z”としている。それと同じ実装である。

[*9] 当然，それ以前の文字も全て同じである。

[*10] Original の strcat 関数も確認していない。その確認は，C の programmer に求められている。実際，**call-mystr.c** ではその

○64 **行目**　**str1** の開始 address **ad1**(=**r3**) の，**n** つ後の文字について考える，という意味の **n** を，**r7** に設ける。mystrlen と同じ seek 方法である。
○66〜69 **行目**　**ad1** の，**n** つ後の文字を，**r7** に読み出す。それが即値 0 でなければ（つまりそこが終端でなければ）**n** を increment して，同じことをやらせる。文字の終端であれば，次の処理に移る。

ここで，increment を先に行ったのは，**r7** が 0 だった場合の処理で新たに Label を設けてそこに branch させる，という処理が面倒だったからである。その方法では，70 行目の **subi** は不要になる。処理速度は若干速くなり，program の大きさは若干大きくなる[*11]と**考えられる**。
○70-71 **行目**　ここで，**str1** の seek 位置 **n** は，上述の事情により null の 1 つ先になっている。そこで，それを 1 つ前の null の位置[*12]に戻してやる。

更に，**str2** の 0 文字目（＝最初の文字）を，**n** 文字目だと見なしてやりたい。そこで，**str2** の開始位置 **ad2**(=**r4**) から **n** を減じてやる。こうすることによって，**str1** の **n** 文字目に **str2'** の **n** 文字目（＝ **str2** の 0 文字目）を書き込む，という処理に出来る。
○73-76 **行目**　というわけで，あとは **str2'** の **n** 文字目を **str1** の **n** 文字目に付け加えてやればいい。もしそれが null であれば，null を **str1** に付け加えた後に終了となる。そうでなければ，**n** を increment してもう一度，である。

3.2　結果

以下に実行結果の 1 例を記す。

```
15:g450318@ca11501:~/Documents/Comsys/20051104/report1:% ./exec_mystr
** ComSys Report 1-B : 2005/11/17(Thu) **
[450318C Sho Iwamoto]

MaxLength
 ? 20

NOTE: String must be 'one-line string'(no CR/LF)
Str1 (length =< 20)
 -> vipper tera tanos
Str2 (length =< 20)
 -> isuwwwwwwwwwww

(0)strlen
Raw data : str1='vipper tera tanos',str2='isuwwwwwwwwwww'
MYstrlen -> 17,16

(1)strcmp
```

確認をしっかり行っている。
[*11] 昔の programmer は処理速度に気を配り，大昔の programmer は program size に気を配る……のかな。
[*12] ここから **str2** を書き込む。

```
MYstrcmp -> vipper tera tanos > isuwwwwwwwwwwww (13)
[strcmp] -> vipper tera tanos > isuwwwwwwwwwwww (13)

(2)strcat
MYstrcat -> vipper tera tanosisuwwwwwwwwwwww

** END **
```

まぁ，期待通りの結果である。mystrcmp 部分について，他にいくつかの結果を書いておく。うまいこと処理できたものである。

```
MYstrcmp -> abcde = abcde (0)
[strcmp] -> abcde = abcde (0)
MYstrcmp -> ado < adobe (-98)
[strcmp] -> ado < adobe (-98)
MYstrcmp -> vipper > vepper (4)
[strcmp] -> vipper > vepper (4)
MYstrcmp -> hellp > hell (112)
[strcmp] -> hellp > hell (112)
```

3.3 考察

処理そのものは上手くできているようである。以下，気になる点をいくつか挙げておく。

■65〜70 行目の処理について　方法の所でも述べたが，元の source では，

```
1     CAT_LOOP:
2           lbzx r5,r3,r7
3           addi r7,r7,1
4           cmpi cr7, r5, 0
5           bne cr7, CAT_LOOP
6           subi r7,r7,1
```

としていたところを，

```
1     CAT_LOOP:
2           lbzx r5,r3,r7
3           cmpi cr7, r5, 0
4           beq cr7, CAT_NEXT
5           addi r7,r7,1
6           b CAT_LOOP
7     CAT_NEXT:
```

と書き換えることが出来る。長さとしては 1 行長くなるが，**subi** の速度を考えれば，処理時間は若干短く

なると思われる。

■文字列領域が不十分な場合の mystrcat の振る舞い　実際に，**call-mystr.c** での文字列領域の確保幅を小さくして処理させてみたが，strcat，mystrcat 共に，文字列領域が十分ある場合と同じ振る舞いをした。これは，strcat も文字列領域が十分であるかどうかを判定していないということであり，c programmer の責任において strcat，mystrcat 共に使用しなければならないのである。

3.4　今後の課題

- Retreat 部分の subroutine 化の方法
- 65〜70 行目の処理について，処理速度と program size のさらなる考察

また，各関数の拡張の idea として，以下のようなことが考えられる。

- 大文字小文字を区別しない strcmp
- 文字列領域の十分さを確認する strcat

更に，余談だが，**call-mystr.c** について以下の拡張が考えられる。

- 改行を含む文字列を上手に読み込む
- strcat の前に領域が十分あるかどうかを確認する subroutine の作成

参考文献

[1] 林晴比古 著,「新訂 新 C 言語入門 シニア編」(SOFTBANK publishing，2004 年)