プログラム解説

伊藤祐輔

平成 20 年 11 月 25 日

1　プログラム解説

プログラムリストはレポートの後に示す。

1.1　前処理

アセンブラコードを変換する前にいくつか処理をしておく必要がある。ここでは、前処理として、文字コードを解釈し、小文字へ変換する。そして、いくつかあるスペースやタブなどを取り除いて、命令とラベル、オペランドの区切りを一つのスペース、オペランド中の区切りを “,” のみに変換する。コメントもここで取り除く。

1.2　要素に分ける

前処理によって整理されたコードを 1 行ずつ取り出す。この 1 行ごと処理をする。1 行からスペースを区切り文字として要素に分ける。要素は必ず 6 つ以下になるはずであるのでそれ以上ならばエラーとする。要素数が分かればどこがラベルでどこが命令なのかがすぐに分かる。

要素数に対する仕分けは以下の通りである。

1 RET 命令やマクロ等

2 命令またはマクロ+オペランド、またはラベル+RET

3 ラベル+命令またはマクロ+オペランド

4 から 6 ラベル+命令+GR0, +123 等

(4 から 6 はオペランドの区切り文字はカンマであるが、スペースが含まれていると要素に分かれてしまうため、それも考慮する。)

以上のようにすると命令やオペランドを仕分けできる。

命令かラベルかを調べるには命令リストにその文字列が存在するか調べれば良い。従って、命令リストを用意し、それと比較してラベルであるかを確認する。比較するとき、命令であれば、どの命令かを番号で覚えておく。こうすることで、アセンブル時にもう一度比較する必要が無くなる。

また、同時にその行で使用するメモリ量も計算する。これはラベルテーブル作成時やアセンブル時に必要になる。

1.3　ラベルテーブル作成

ラベルテーブルを作成するにはそれぞれの行で使用するメモリ量が必要になる。これはすでに前の段階で求めてあるのでそのまま使用すれば簡単に作成できる。

1.4 命令を翻訳

すでに要素を解析するときに行と命令リストが対応しているので、そのまま命令リストに従って機械語に翻訳する。オペランドはここで翻訳する。

オペランドにラベルが出現したらラベルテーブルを参照すれば良い。

1.5 結果表示

翻訳された機械語を表示する。エラーはエラー発生時に配列オブジェクトに順に格納されていくため、これを表示する。

2 クラス設計

このプログラムはアセンブラ本体の `CSAssembler` クラスと `main` 文で構成されている。

2.1 main 文

これがプログラムの開始である。まず、引数から入力ファイルのパスを取り出して、ファイルの内容をすべて読み込む。ファイルが正常に読み込めた場合、アセンブラクラスをインスタンス化して、アセンブルを実行する。

アセンブルが終了したら、結果を取り出し、表示する。エラーがあればそれも表示する。

2.2 CSAssembler クラス

これがアセンブルを行うメインのクラスである。(ヘッダファイル CSAssembler.h 参照) このクラスが提供するメソッドはアセンブルを実行する `assemble` のみである。

ソースファイルと結果は次のプロパティで提供される。

source アセンブラコードを入力する。

`NSString*`型。

startAddress 開始番地を取得。

labelTable ラベルテーブルを取得。

`NSDictionary*`型

error エラーのリストを取得。

エラー構造体 (`AsmError*`) の一覧が配列 (`NSArray*`) で格納されている。構造体は `NSValue*`でラップされている。

asmLines アセンブラ結果を取得。

アセンブル結果の構造体 (`AsmLine*`) の一覧が配列 (`NSArray*`) で格納されている。構造体は `NSValue*`でラップされている。

2.3 命令リストヘッダファイル

クラスとは別に命令リストのヘッダファイルを用意した。(CSInstructionList.h)

ここにはCASLの命令やマクロ、その機械語やメモリ量、オペランド数が記載されている。ここへ命令を追加すればアセンブラの拡張が可能である。

3 実行結果

まず、次のようなアセンブルコードを用意した。

```
start
lea gr1,0
lea gr3,0,gr1
call .rdmpd
lea gr1,1
lea gr2,0,gr1
call .rdmpd
st gr3,wrk
loop add gr1,wrk
call .rdmpd
st gr2,wrk
lea gr2,0,gr1
cpa gr1,c10000
jmp loop
exit
wrk ds 1
c10000 dc 10000
end
```

(以下、アセンブルソースは四角い枠、出力結果は丸い枠で示す。)

これをアセンブルすると次のように表示される。正しくアセンブルされている。

```
1: 0000 12100000 - lea gr1,0
2: 0002 12310000 - lea gr3,0,gr1
3: 0004 80f0f02e - call .rdmpd
4: 0006 12100001 - lea gr1,1
5: 0008 12210000 - lea gr2,0,gr1
6: 000a 80f0f02e - call .rdmpd
7: 000c 1130002a - st gr3,wrk
8: 000e 2010002a - loop add gr1,wrk
9: 0010 80f0f02e - call .rdmpd
10: 0012 1120002a - st gr2,wrk
11: 0014 12210000 - lea gr2,0,gr1
12: 0016 4010002b - cpa gr1,c10000
13: 0018 64f0001c - jmp loop
14: 001a 64f0f0b0 - exit
15: 001c 0000 - wrk ds 1
16: 001d 2710 - c10000 dc 10000
Start at 0000(0)
-----Label Table-----
wrk = 001c(28)
loop = 000e(14)
.rdmpd = f020(61472)
c10000 = 001d(29)
```

ここで、開始アドレスを

```
start loop
...
```

と指定すると、

```
...
Start at 000e(14)
...
```

と出力され、正しく開始アドレスが設定されていることが分かる。

次に、

```
...
wrk ds 10
...
```

として、ds 命令の領域を大きくしてみる。結果は

```
...
15: 001c 0000 - wrk ds 10
16: 0026 2710 - c10000 dc 10000
Start at 000e(14)
-----Label Table-----
wrk = 001c(28)
loop = 000e(14)
.rdmpd = f020(61472)
c10000 = 0026(38)
```

と表示され。正しく領域の指定が出来ている。

また、出力結果は変化しないが、区切り文字を

```
 start
 lea [tab]   [tab]gr1,0
 lea[tab]gr3,0,   gr1[tab]   ; comment
 call      .rdmpd ; comment1
 ...
```

のように、タブやスペースを挿入しても正しく変換できた。コメントも無視されている。

また、dc 命令で、値を 16 進数や負に変更しても正しく変換できていることが確認できた。

```
...
16: 0026 fffd - c10000 dc -3
...
```

```
...
16: 0026 f9f9 - c10000 dc #f9f9
...
```

4 エラー処理

エラー処理が正しくできているか確認するために、次のようにわざと間違いを含んだアセンブラコードを入力する。

```
1: start loop
2: lea gr1,,0
3: lea gr3,0,gr1
4: call .rdmpdp
5: lea gr1,1
6: lea gr2,-1,gr1
7: call .rdmpd
8: st gr3,wrk
9: loop add gr1,wrk
10: call .rdmpd
11: st gr2,wrk
12: loop lea gr2,0,gr1
13: cpa gr1,c10000
14: jmp loop
15: exit
16: wrk ds -11
17: c10000 dc s
18: end
```

出力されたエラーは以下の通りである。正しくすべてのエラーが出力されている。

```
-----Error List-----
12: Label already exist
2: Invalid operand
4: Label not found
16: Invalid constant
17: Label not found
```

5 考察

プログラムはオブジェクト指向を用いたため、全体の構成が見通しの良いものとなった。

また、このクラスは汎用的に作ってあるため、将来エミュレータに搭載することが可能である。

機能は要求どおりすべて搭載したが、まだエラー処理があまいところがあるので、これは改良の余地がある。

6 感想・その他

要求定義書は製作していないが、プログラムの流れを設計したので、予定通りのものが出来てよかった。

上にも書いたが、これを流用してエミュレータの機能を拡張したいと思う。