手を動かして学ぶ！コンピュータアーキテクチャとアセンブリ言語プログラミングの基本

はじめに

こんにちは。皆さんはアセンブリ言語は得意ですか？私は得意ではありません、苦手と言っていいでしょう。何故苦手か考えました、手軽に実行できる環境が無いのが原因の１つではないかと考えました。今やプログラミング環境というのは無料でネットで拾ってきたコンパイラにパスを通してさっと実行するような時代です。人によっては用意されている仮想環境で済ませた人もいるかもしれません。そんな時代に生まれてしまった私たちはいかにして低レベルについて学ぶのか、そうだ！シミュレータだ！
シミュレータは良いとしても、どんなアセンブリ言語でやろうかなという話なのですが、

命令セット	適しているか	備考
Intel	x	仕様が膨大すぎる
ARM	x	ライセンスの関係で仕様書を取り寄せるのが難しい
RISC-V	△	悪くないけど公式の仕様書が英語（RISC-V原典を使えばいける？）
CASL-2	△	シンプルだし実装も難しくなさそうだけど、提示されている仕様がかっちりしていない

色々考えた結果、既存のものの完全なシミュレートは難しそうなので、拡張をするのが良い気がします。という訳でそんな感じのやつを実装したので以下のリンクからどうぞ。

MLFEシミュレータ

MLFEシミュレータのgithubリンク

インストール方法

シェルは`Powershell`を使っております。マニュアルではWindows10で動作チェックしております。

このアプリケーションソフトはPython環境を前提としています。とりあえずは以下の環境で実行するのですが、おそらく`Python 3.7`以降なら実行可能です。

```
> python --version
Python 3.9.0
```

このリポジトリをローカルにクローンするか、直接ダウンロードしてみてください。

```
> git clone https://github.com/
```

ダウンロードしたらそのディレクトリに移動します。
mlfe.pyが本体です。バージョン確認をして上手く動作するかを確認してください。

```
> python mlfe.py --version
```

manualディレクトリには取扱説明書が入っております。

まずHelloWorldをしてみよう

sampleディレクトリに入っている`hello.fe`を例にこのプログラムの実行のしかたを説明したいと思います。

```
> python mlfe.py sample\hello.fe
HelloWorld
```

なんだかわかんないけど実行できましたね、OKです。 `hello.fe` の中身を見てみましょう。

```
PGM     START
        OUT     ='HelloWorld', =10
        RET
        END
```

`PGM` はラベルです。そのアドレス番地に名前を付けているような処理です。

最初と最後に `START` と `END` がありますね。これはプログラムの始めと終わりを表しています。 `START` が書かれた所からプログラムの実行が始まります。

`OUT` は文字の始めのアドレス番地と文字数が入っているアドレス番地を入れることで標準出力できる命令です。しかし不思議な表記ですね、アドレスらしかぬものが記述されています。

これは、 `=` と書くことでデータの確保とその場所を呼び出すという操作を勝手にしてくれるものです。

最後に `RET` ですね。これはプログラムの `return` に相当するもので、本来は `CALL` とセットになっているものですが、何からも呼ばれていないこの状態ではプログラムの終了を表しています。C言語の `main` 関数の最後の `return 0;` のようなものですね。

データがどう展開されているかを見たい

さきほど `=` の話の所でデータの確保という言い方をしましたが、じゃあどんなふうにデータの確保がなされているのか気になりますよね？嘘でも気になるって言っといてください。 データの展開は `--dry-assembly` オプションで見ることが出来ます。

```
> python mlfe.py sample\hello.fe --dry-assembly
    0 START
    1 PUSH    0       GR0
    2 PUSH    0       GR1
    3 PUSH    0       GR2
    4 PUSH    0       GR3
    5 LD      GR0     19
    6 LD      GR1     19
    7 LD      GR3     20
    8 CPL     GR1     33
    9 JZE     14
   10 LD      GR2     23      GR1
   11 WRITE   GR0     GR2
   12 ADDL    GR1     GR3
   13 JUMP    8
   14 POP     GR3
   15 POP     GR2
   16 POP     GR1
   17 POP     GR0
   18 JUMP    21
   19 DATA    0
   20 DATA    1
   21 NOP
   22 RET
   23 DATA    72
   24 DATA    101
   25 DATA    108
   26 DATA    108
   27 DATA    111
   28 DATA    87
   29 DATA    111
   30 DATA    114
   31 DATA    108
   32 DATA    100
   33 DATA    10
   34 END
```

何か予想よりもいっぱい出てきましたね、ちょっとのことでもアセンブリだとアホみたいに長いのが怖いですね。

さてなぜこんなに長いのかは二つ理由があります。ひとつは`OUT`命令のおかげです。

`OUT`命令はマクロ命令に属する命令で、いくつかの機械語命令という基本的な命令が集まってできていてちょっとの記述でいろんなことをしてくれるショートカットみたいなものです。

もう一つは、`OUT`命令の後ろに書いた二つの`=`のおかげです。`=`を書くとデータの確保を行ってくれると言いましたが、先ほどの出力結果の23行から32行の値を見てみてください。`DATA`と書かれた後ろに10進数のASCIIコードで`HelloWorld`と見えませんか？そんな感じで`END`の前にずらっと確保してくれます。33行の`10`は`=10`と書いたときのものですね。

「オプション長いな」とか「16進数とか2進数で見たいんだけど」って方もご安心ください。詳しくはマニュアルを読んでほしいのですが、以下のように実行すると使いやすいと思います。

```
> python mlfe.py sample\hello.fe -d -b
       0 START
       1 PUSH    0       GR0
      10 PUSH    0       GR1
      11 PUSH    0       GR2
     100 PUSH    0       GR3
     101 LD      GR0     10011
     110 LD      GR1     10011
     111 LD      GR3     10100
    1000 CPL     GR1     100001
            ~省略~
   11111 DATA    1101100
  100000 DATA    1100100
  100001 DATA    1010
  100010 END
```

`-d`オプションが`--dry-assembly`の略記です。後ろの`-b`（2進数）を`-x`（16進数）や`-d`（10進数）と指定することで表示の進数を変えることができます。お好みでどうぞ。

データがどんなふうに実行されているかを追ってみたい

さきほどの展開されたものをみるとJUMPとかRETとかどこかへ跳びそうなのがちらほら見られます。どんなふうに実行されているか気になりますよね？そんなときはこんな`--trace-line`オプションを使います。

```
> python mlfe.py sample\hello.fe --trace-line
    0 START
    1 PUSH    0       GR0
    2 PUSH    0       GR1
    3 PUSH    0       GR2
    4 PUSH    0       GR3
    5 LD      GR0     19
    6 LD      GR1     19
    7 LD      GR3     20
    8 CPL     GR1     33
    9 JZE     14
   10 LD      GR2     23      GR1
   11 WRITE   GR0     GR2
H   12 ADDL    GR1     GR3
   13 JUMP    8
    8 CPL     GR1     33
    9 JZE     14
   10 LD      GR2     23      GR1
   11 WRITE   GR0     GR2
e   12 ADDL    GR1     GR3
   13 JUMP    8
    8 CPL     GR1     33
    9 JZE     14
   10 LD      GR2     23      GR1
   11 WRITE   GR0     GR2
l   12 ADDL    GR1     GR3
   13 JUMP    8
    8 CPL     GR1     33
    9 JZE     14
   10 LD      GR2     23      GR1
   11 WRITE   GR0     GR2
l   12 ADDL    GR1     GR3
   13 JUMP    8
```

```
     8 CPL     GR1     33
     9 JZE     14
    10 LD      GR2     23      GR1
    11 WRITE   GR0     GR2
o   12 ADDL     GR1     GR3
    13 JUMP    8
     8 CPL     GR1     33
     9 JZE     14
    10 LD      GR2     23      GR1
    11 WRITE   GR0     GR2
W   12 ADDL     GR1     GR3
    13 JUMP    8
     8 CPL     GR1     33
     9 JZE     14
    10 LD      GR2     23      GR1
    11 WRITE   GR0     GR2
o   12 ADDL     GR1     GR3
    13 JUMP    8
     8 CPL     GR1     33
     9 JZE     14
    10 LD      GR2     23      GR1
    11 WRITE   GR0     GR2
r   12 ADDL     GR1     GR3
    13 JUMP    8
     8 CPL     GR1     33
     9 JZE     14
    10 LD      GR2     23      GR1
    11 WRITE   GR0     GR2
l   12 ADDL     GR1     GR3
    13 JUMP    8
     8 CPL     GR1     33
     9 JZE     14
    10 LD      GR2     23      GR1
    11 WRITE   GR0     GR2
d   12 ADDL     GR1     GR3
    13 JUMP    8
     8 CPL     GR1     33
     9 JZE     14
    14 POP     GR3
    15 POP     GR2
    16 POP     GR1
    17 POP     GR0
    18 JUMP    21
    21 NOP
    22 RET
```

ごちゃごちゃ度が跳ね上がりましたね、落ち着いて見てみます。

左側に何文字ごとかに`HelloWorld`が出力されているのがわかりますね。その上を見てみると`WRITE`が必ずあります。`WRITE`命令は現在のレジスタの内容をポートを指定して出力する命令です。この実装では、レジスタの内容を数字か文字か指定して出力しているという感じです。

`OUT`命令はざっくりいうとメモリを読みだしてレジスタに格納して、レジスタの内容を文字として出力することを文字数分繰り返すという命令の集まりです。マクロ命令の目的はショートカット、いちいち文字列出力の度にループ構造と`WRITE`を書いてたら面倒すぎるので簡単に書けるようにあるわけですね。

最初と最後の怒涛の`PUSH` `POP`はレジスタの内容がマクロ内で書き換えられないようにスタックに退避しておく為の命令です。

ちなみに、データの出力と追跡どちらも行いたいときは`-a`オプションを使ってください。もちろん`-b`とかのフォーマット指定も有効です。

アセンブリの性能を測りたい

何をもってプログラムの性能というのかは難しい所ですが、指標の一つに時間があるでしょう。アセンブリにおいては実行した行の量という事になるのでしょうか、しかしHelloWorld程度はおそらく一瞬で出来てしまうと思います。なので一行の実行時間を変えてみましょう。

```
> python mlfe.py sample\hello.fe --clock-speed 0.05
```

一行にかける時間を0.05秒にしてみました。だいたい動作速度20HzのCPUでやったらどうなるかのテストになりますね。そんなもんがあるかは知りませんが。

```
> Measure-Command{python mlfe.py sample\hello.fe --clock-speed 0.05}

Days              : 0
            ~~省略~~
TotalSeconds      : 5.0297143
TotalMilliseconds : 5029.7143
```

`Powershell`の処理の実行時間を求める機能を使いました。大体全部で5秒かかっていることがわかりましたね。これであなたの書いたアセンブリがどんなスペックがあるかチェックしてみよう！

他のサンプルプログラムは何？

表にしてみました。のちのち作ります。試しに動かしてみてください。

プログラム	概要	実行例
fizzbuzz.fe	FizzBuzzをするプログラムです。	> python mlfe.py fzbz.fe
rdp.fe	簡単な四則演算をしてくれるプログラムです。	> echo "3 * (2 + (10 / 2)) - 8" | python mlfe.py rdp.fe
rdp2.fe	四則演算を計算するアセンブリを出力するコンパイラです。	> echo "3 * (2 + (10 / 2)) - 8" | python mlfe.py rdp2.fe > out.fe
mine_sweeper.fe	マインスイーパーをプレイできるプログラムです。	> python mlfe.py mine_sweeper

まとめ

- MLFEはPythonで動作するアセンブリシミュレータ
- Python環境さえあればGithubからダウンロードして動かすことが出来る
- 実行はターミナルから行う。
- ラベルとはアドレスに名前を付けること。
- `START`と`END`は必須である。
- `OUT`命令は文字を出力するマクロ命令
- `=`はデータ定義と仕様を同時に行う機能
- 実行せずメモリの展開を行うには`--dry-assembly/-d`オプション
- 実行中の命令を追跡するのは`--trace-line/-t`オプション
- オプション出力した中で数値のフォーマットを変えるには`-b/-x/-d`オプション
- 一行に掛ける実行時間の設定は`--clock-speed/-c`オプション
- サンプルプログラムは`Helloworld`以外にFizzBuzzとハノイの塔の解法と四則演算をするプログラムがある