CASL2 のメモ

CASL II のメモ.

目次

        - ADDA(ADD Arithmetic)
        - SUBL(Subtract Logical)
        - SUBA(Subtract Arithmetic)
        - AND
        - OR
        - XOR
        - SLL(Shift Left Logical)，SRL(Shift Right Logical)
        - SLA(Shift Left Arithmetic)，SRA(Shift Right Arithmetic)
    - 比較演算命令
        - CPL(Compare Logical)
        - CPA(Compare Arithmetic)
    - 条件分岐命令
        - JPL(Jump on Plus)
        - JMI(Jump on Minus)
        - JZE(Jump on Zero)
        - JNZ(Jump on Non Zero)
        - JOV(Jump on Overflow)
        - JUMP(unconditional Jump)
    - PUSH と POP
        - PUSH
        - POP

プログラムの実行

Windows 10 環境

1. IPA 配布のシミュレータ（微妙）

作業ディレクトリに`CASL2.bat`と`RUN.bat`を予めコピーしておく.

オプションについては`C:\javaCASL2_2.0\Manual.pdf`を参照.

```
$ CASL2 プログラム名.cas //アセンブルをして「プログラム名.obj」を生成
$ RUN プログラム名.obj //実行
```

2. VS Code 上での実行（授業後半だと機能不足になる）

CASL2/COMET2 拡張機能をインストールする.

3. CCASL II（未検証だが，動作するらしい）

CCaslII(Windows 版)をダウンロード.

Linux 環境

> WSL1 では動作しないらしい.
> WSL を使いたい場合は，WSL2 を使うこと.

CCaslII(Linux 版)から `ccasl2-4.10.tar.gz` をダウンロードし，それを解凍する.

```
$ wget http://hyamag1979.wp.xdomain.jp/wp-content/uploads/ccasl2-4.10.tar.gz
$ tar zxvf ccasl2-4.10.tar.gz
$ sudo mv (ファイル名) /usr/local/bin/
```

これは 32 bit アプリケーションなので，それに必要なものをインストールする.

```
$ sudo apt install lib32z1
$ sudo apt install lib32stdc++6
```

アセンブル

`caslasm` を用いることでアセンブルできる.

```
$ caslasm (ファイル名).cas
// リスティングファイルを作成して，アセンブル
$ caslasm (ファイル名).cas -l
```

リンク

`casllink` を用いることでリンクできる.

```
$ casllink (ファイル名).obj
// シンボルファイルを作成して，リンク
$ casllink (ファイル名).obj -sym
```

実行

`caslsim` を用いることで実行できる.

```
$ caslsim (ファイル名).exe
```

デバッグ

`casldeb`を用いることでデバッグできる.

```
$ casldeb (ファイル名).exe
// シンボルファイル付きデバッグ
$ casldeb (シンボルファイル名).sym (オブジェクトファイル名).exe
```

最低限の使い方

`-?`：コマンドの一覧を表示する.

`-g`：プログラムを実行する（ブレークポイントがあるとその直前で停止する）.

`-r`：現時点での汎用レジスタやフラグレジスタなどの値を表示する.

`-r GR~`：レジスタの値を変更する.

`-t`：1 行実行する.

`-u (開始番地) (終了番地)`：開始番地から終了番地までの逆アセンブル.

`-bp (番地)`：指定した番地にブレークポイントを作る.

`-bl`：ブレークポイントを表示する.

`-q`：デバッグ終了.

プログラムの書き方

- セミコロン「;」から行末までコメントとして認識される（C 言語の/**/みたいなものはない）
- ラベルは行の 1 文字目からスペースを入れずに書く
- ラベルは 1~8 文字で 1 文字目は英大文字，2 文字目以降は英大文字 or 数字（小文字はダメ）
- START 命令で実行される行を指定でき，この行のラベルがプログラム名になる
- END 命令でプログラムの記述を終了できる
- RET 命令でプログラムの動作を終了されることができる（必須）
- #をつけて数字を書くと，16 進数として認識される（例. #1234）
- 文字列はシングルクォーテーション「'」で囲む（ダブルクォーテーションはダメ）

```
ラベル     START              ; 注釈
[ラベル]  プログラムの実行部分
[ラベル]  RET
[ラベル]  記憶領域の割り当て
[ラベル]  定数値の定義
          END
```

フラグレジスタ

OF(オーバーフローフラグ)

オーバーフローが発生すると 1，しない場合は 0 をセットする．

SF(サインフラグ)

演算結果が正のとき 0，負のとき 1 をセットする．

ZF(ゼロフラグ)

演算結果が零のとき 1，そうでないなら 0 をセットする．

リテラル

DC 命令を省略できる記法（処理数が変わるわけではない）．

(1) 使わない例

```
PRG     START
        LD      GR1,DATA
        RET
DATA    DC      10
        END
```

(2) 使う例

```
PRG     START
        LD      GR1,=10
        RET
        END
```

基本的な命令

DC(Define Constant)

定数値を主記憶装置（メモリ）に設定する．

```
ラベル　　DC　　定数値
```

DS(Define Storage)

指定した語数分の領域を確保する．

```
ラベル　　DS　　語数
```

LAD(Load Address)

値を汎用レジスタに設定する．

```
LAD　　GRx(xは0~7),数字
```

アドレスを汎用レジスタに設定する．

```
LAD　　GRx,アドレス
```

LD(Load)

主記憶装置に格納されている値を汎用レジスタに設定する．

```
LD　　GRx,アドレス
```

右側の汎用レジスタに設定されている値を，左側に設定する．

```
LD　　GRx,GRy
```

第 2 項目の値と第 3 項目（**指標レジスタ**）の値を加算し，その結果を**実行アドレス**とする．実行アドレスに設定されている値を読み込み、第 1 項目の汎用レジスタに設定する。このような番地指定の方法を**インデックスアドレッシング**（**指標アドレス指定**）と呼ぶ。

```
LD　　GRx,数字,GRy ; GRx <- (数字＋GRy)に設定されている値
```

```
LD      GRx,adr,GRy ; GRx <- adr+(GRyに設定されている値)のアドレス
; 例
; LD    GR1,DATA,GR2
```

リテラルを用いることで数値を入れることができる.

```
LD      GRx,=数値
```

ST(Store)

主記憶装置に汎用レジスタの値を設定する.

```
ST      GRx,アドレス
```

アドレス＋数値に汎用レジスタの値を設定する.

```
ST      GRx,数字,アドレス ; GRxに設定されている値 -> (数字＋アドレス)
```

IN

キーボード入力された文字と文字数を受け取る．第 1 項目で文字列，第 2 項目で文字数を取得する。

```
IN DATA,LEN

; 具体的なDATAとLENの設定（領域の確保）
; LEN   DS      1
; DATA  DS      256
```

OUT

文字列を画面に出力する．第 1 項目に文字列，第 2 項目に文字数を指定する。

```
OUT DATA,LEN

; ハローワールドでの例
; MAIN  START
;       OUT     DATA,LEN
;       RET
; DATA  DC      'Hello, World'
; LEN   DC      12
;       END
```

CALL

プログラムを呼び出す命令．一旦，戻り番地をスタックに格納させることで，呼び出し元に戻れる．

```
CALL    OUTDEC ; OUTDECは副プログラムの入口名
```

演算

論理(Logical)：符号を考慮しない（つまり，全て正として扱う）．

算術(Arithmetic)：符号を考慮する．

ADDL(ADD Logical)

符号なし数値の足し算をする．表現できる範囲を超えると OF が 1 にセットされる。

```
ADDL    GR1,GR2 ; GR1 <- GR1+GR2の値
ADDL    GR3,=50 ; リテラル使わないと50番地のよくわからないところを参照してしまう．
```

ADDA(ADD Arithmetic)

符号付き数値の足し算をする．最上位ビットが 1 なら負として扱う。表現できる範囲を超えると OF が 1 にセットされる。

```
ADDA    GR1,GR2         ; GR1 <- GR1+GR2
ADDA    GR3,=-50        ; リテラル必要!!
```

SUBL(Subtract Logical)

符号なし数値の引き算をする．表現できる範囲を超えると OF が 1 にセットされる。

```
SUBL    GR1,GR2 ; GR1 <- GR1-GR2
SUBL    GR3,=50 ; リテラル使わないと50番地のよくわからないところを参照してしまう．
```

SUBA(Subtract Arithmetic)

符号付き数値の引き算をする．最上位ビットが 1 なら負として扱う。表現できる範囲を超えると OF が 1 にセットされる。

```
SUBA    GR1,GR2         ; GR1 <- GR1-GR2
SUBA    GR3,=-50        ; リテラル必要!!
```

AND

論理積を取る.

```
AND     GR1,GR2         ; GR1 <- GR1 AND GR2
AND     GR3,=#00FF      ; リテラル必要!!
```

OR

論理和を取る.

```
OR      GR1,GR2         ; GR1 <- GR1 OR GR2
OR      GR3,=#0020      ; リテラル必要!!
```

XOR

排他的論理和を取る.

```
XOR     GR1,GR2         ; GR1 <- GR1 XOR GR2
XOR     GR3,=#FFFF      ; リテラル必要!!
```

SLL(Shift Left Logical)，SRL(Shift Right Logical)

符号なし数値をビットシフトさせる.

```
SLL     GR1,2 ; GR1を2ビット左シフトする（リテラル要らない!!）
SRL     GR2,2 ; GR2を2ビット右シフトする（リテラル要らない!!）
```

```
SLL     GR1,4,GR2 ; GR1 <- GR2を4ビット左シフトさせたもの
SRL     GR1,4,GR2 ; GR1 <- GR2を4ビット右シフトさせたもの
```

SLA(Shift Left Arithmetic)，SRA(Shift Right Arithmetic)

符号付き数値をビットシフトさせる．ただし，符号ビットはそのまま。

```
SLA     GR1,2 ; GR1を2ビット左シフトする（リテラル要らない!!）
SRA     GR2,2 ; GR2を2ビット右シフトする（リテラル要らない!!）
```

比較演算命令

論理(Logical)：符号を考慮しない（つまり，全て正として扱う）．

算術(Arithmetic)：符号を考慮する．

CPL(Compare Logical)

2 つの符号なし数値 A，B を比較する（A－B を演算してる）．

OF：常に 0.

SF：A>=B なら 0，A<B なら 1 をセットする．

ZF：A=B なら 1，そうでないなら 0 をセットする．

```
CPL     GR1,GR2 ; 演算結果は残らない
CPL     GR3,=10 ; リテラルが必要
```

CPA(Compare Arithmetic)

2 つの符号付き数値 A，B を比較する（A－B を演算してる）．

OF：常に 0.

SF：A>=B なら 0，A<B なら 1 をセットする．

ZF：A=B なら 1，そうでないなら 0 をセットする．

```
CPA     GR1,GR2 ; 演算結果は残らない
CPA     GR3,=0  ; リテラルが必要
```

条件分岐命令

JPL(Jump on Plus)

演算結果が正のとき（**ZF=0**かつ**SF=0**）にラベル先へ分岐する．満たしていなければ，そのまま次の行を実行する。

```
JPL     JMP     ; 条件満たしたとき，ラベルJMPへ分岐
JPL     1,GR1   ; 条件満たしたとき，GR1の中身+1番地の行へ移動（普通使わない）
```

JMI(Jump on Minus)

演算結果が負のとき（**SF=1**）にラベル先へ移動する．満たしていなければ，そのまま次の行を実行する。

```
JMI     JMP     ; 条件満たしたとき，ラベルJMPへ移動
JMI     1,GR1   ; 条件満たしたとき，GR1の中身+1番地の行へ移動（普通使わない）
```

JZE(Jump on Zero)

演算結果が零のとき（**ZF=1**）にラベル先へ移動する．満たしていなければ，そのまま次の行を実行する。

```
JZE     JMP     ; 条件満たしたとき，ラベルJMPへ移動
JZE     1,GR1   ; 条件満たしたとき，GR1の中身+1番地の行へ移動（普通使わない）
```

JNZ(Jump on Non Zero)

演算結果が零でないとき（**ZF=0**）にラベル先へ移動する．満たしていなければ，そのまま次の行を実行する。

```
JNZ     JMP     ; 条件満たしたとき，ラベルJMPへ移動
JNZ     1,GR1   ; 条件満たしたとき，GR1の中身+1番地の行へ移動（普通使わない）
```

JOV(Jump on Overflow)

OF=1のときにラベル先へ移動する．満たしていなければ，そのまま次の行を実行する。

```
JOV     JMP     ; 条件満たしたとき，ラベルJMPへ移動
JOV     1,GR1   ; 条件満たしたとき，GR1の中身+1番地の行へ移動（普通使わない）
```

JUMP(unconditional Jump)

問答無用で分岐する．

```
JUMP    JMP     ; ラベルJMPへ移動
JUMP    1,GR1   ; GR1の中身+1番地の行へ移動
```

PUSH と POP

PUSH

```
PUSH    0,GR1    ; GR1の中身をプッシュ
```

POP

```
POP     GR2      ; GR2にポップ
```