

YACASL2

コマンドラインインターフェイスで動作するオープンソースの CASL II 処理システム

j8takagi

Copyright © 2010-2023 j8takagi

目次

1	YACASL2の概要	1
2	YACASL2の使用例	2
2.1	実行結果の出力だけを表示	2
2.2	アセンブル結果の確認	2
2.3	実行時のレジスタとメモリを表示	4
2.3.1	特定のレジスタを表示	6
2.3.2	プログラム終了時の値を表示	6
2.3.3	プログラムのステップ数を表示	7
2.4	アセンブルと実行を別に行う	7
2.5	1語の解析	7
2.6	CASL II ライブラリの使用	7
2.6.1	数値を出力する	8
3	casl2の呼び出し	9
	オプション	9
4	comet2の呼び出し	11
	オプション	11
5	dumpwordの呼び出し	13
	注意	13
	オプション	13

1 YACASL2の概要

YACASL2 は、UNIX や Linux のコマンドラインインターフェイスで動作するオープンソースの CASL II 処理システムです。CASL II は情報処理試験で用いられるアセンブラ言語で、次の資料により仕様が公開されています。

情報処理技術者試験 情報処理安全確保支援士試験 試験で使用する情報技術に関する用語・プログラム言語など Ver 4.3 (https://www.ipa.go.jp/shiken/syllabus/ps6vr7000000i9g3-att/shiken_yougo_ver4_3.pdf) [PDF ファイル]
別紙 2 アセンブラ言語の仕様 (5 ページから 12 ページまで)

YACASL2 は、CASL II アセンブラ言語で記述されたファイルを実行し、仮想マシン COMET II 上で実行します。アセンブルと実行は、連続で行うこともできますし、アセンブル結果をファイルに保存してあとから実行することもできます。YACASL2 の動作は CASL II の仕様に準拠しているため、情報処理試験の問題をはじめ各種参考書やサイトに記載された CASL II のプログラムをそのままアセンブルして実行できます。また、本パッケージ中に CASL II のサンプルプログラムが多数収録されています。

YACASL2 は、「ふつうの処理系」として動作します。YACASL2 の操作は、端末上のコマンドラインインターフェイス (CLI) で操作します。YACASL2 は、次のような動作内容をすべてテキストで出力します。

- ラベルとアドレスの対応
- アセンブル結果
- 実行時のレジスタの内容
- 実行時のメモリの内容

YACASL2 では、機械コードモニターを使い、動作中の CPU やメモリーを調べたりデバッグしたりすることもできます。

また、出力された動作内容は、GNU/Linux のさまざまなコマンド、たとえば、`cat`、`less`、`grep`、`wc`などを使って解析できます。

2 YACASL2 の使用例

YACASL2 は、テキストファイルに記述された CASL プログラムを処理します。以下の例で用いられる CASL プログラムのファイルは、テキストエディタなどで作成するか、インストールしたディレクトリの中にある `as` ディレクトリからコピーしてください。

2.1 実行結果の出力だけを表示

インストール時にコマンド実行の確認に使った `hello.casl` は、次のような内容です。CASL II のマクロ命令 `OUT` は、文字列を出力します。

```
$ cat hello.casl
MAIN      START
          OUT      OBUF,LEN
          RET
OBUF      DC      'Hello, World!'
LEN       DC      13
          END
```

次のコマンドを実行すると、CASL II のアセンブルと仮想マシン COMET II 上での実行が連続で行われ、文字列が出力されます。

```
$ casl2 hello.casl
Hello, World!
```

`addl.casl` は、3 と 1 の和を求めます。

```
$ cat addl.casl
;;; ADDL r,adr
MAIN      START
          LD       GR1,A
          ADDL    GR1,B
          RET
A         DC      3
B         DC      1
          END
```

このプログラムには出力命令がないため、オプションなしで実行した場合には結果が出力されません。

```
$ casl2 addl.casl
$
```

実行内容を確認するには、後述のように CPU 内にあるレジスタやメモリの内容を表示するか、結果を出力するための処理を追加する必要があります。

2.2 アセンブル結果の確認

`casl2` の処理途中で行われるアセンブルの結果を表示するには、オプション `-a` を指定します。また、ラベルとアドレスの対応表を表示するには、オプション `-l` を指定します。

次のコマンドでは `hello.casl` の、ラベルとアドレスの対応表と、アセンブル結果と、実行結果が表示されます。`OUT` はアセンブラ命令で複数の機械語命令で構成されているため、命令行 1 行に対して、複数行のコードが生成されます。

```
$ casl2 -a -l hello.casl

Assemble hello.casl (0)

Label::::
```

```

MAIN.LEN ---> #0020
MAIN ---> #0000
MAIN.OBUF ---> #0013

Assemble hello.casl (1)
hello.casl: 1:MAIN START
hello.casl: 2: OUT OBUF,LEN
#0000 #7001
#0001 #0000
#0002 #7002
#0003 #0000
#0004 #1210
#0005 #0013
#0006 #1220
#0007 #0020
#0008 #F000
#0009 #0002
#000A #1210
#000B #0021
#0021 #000A
#000C #1220
#000D #0022
#0022 #0001
#000E #F000
#000F #0002
#0010 #7120
#0011 #7110
hello.casl: 3: RET
#0012 #8100
hello.casl: 4:OBUF DC 'Hello, World!'
#0013 #0048
#0014 #0065
#0015 #006C
#0016 #006C
#0017 #006F
#0018 #002C
#0019 #0020
#001A #0057
#001B #006F
#001C #0072
#001D #006C
#001E #0064
#001F #0021
hello.casl: 5:LEN DC 13
#0020 #000D
hello.casl: 6: END
Hello, World!

```

add1.caslの、ラベルとアドレスの対応表と、アセンブル結果は、次のようになります。

```
$ casl2 -a -l add1.casl
```

```
Assemble add1.casl (0)
```

```

Label:::
MAIN ---> #0000
MAIN.A ---> #0007
MAIN.B ---> #0008
MAIN.C ---> #0009

Assemble addl.casl (1)
addl.casl: 1:;;; ADDL r,adr
addl.casl: 2:MAIN START
addl.casl: 3: LD GR1,A
          #0000 #1010
          #0001 #0007
addl.casl: 4: ADDL GR1,B
          #0002 #2210
          #0003 #0008
addl.casl: 5: ST GR1,C
          #0004 #1110
          #0005 #0009
addl.casl: 6: RET
          #0006 #8100
addl.casl: 7:A DC 3
          #0007 #0003
addl.casl: 8:B DC 2
          #0008 #0002
addl.casl: 9:C DS 1
          #0009 #0000
addl.casl: 10: END

```

なお、オプション-Aを指定すると、アセンブル結果が表示される時点で処理が終了します。仮想マシン COMET II でのプログラム実行はされません。

2.3 実行時のレジスタとメモリを表示

YACASL2では実行中のCPUのレジスタとメモリの内容をそれぞれ、-tと-dを指定することで表示できます。

また、-Mで、仮想マシン COMET II のメモリ容量を語 (16 ビット) 単位で指定できます。小さいプログラムを実行するときは、メモリ容量を小さくすれば結果が見やすくなります。

addl.caslでは、次のようにCPUのレジスタとメモリの内容を表示できます。

```
$ casl2 -t -d -M16 addl.casl | less
```

```
Assemble addl.casl (0)
```

```
Assemble addl.casl (1)
```

```
Executing machine codes
```

```
#0000: Register:::
```

```
#0000: GR0: 0 = #0000 = 0000000000000000
```

```
#0000: GR1: 0 = #0000 = 0000000000000000
```

```
#0000: GR2: 0 = #0000 = 0000000000000000
```

```
#0000: GR3: 0 = #0000 = 0000000000000000
```

```
#0000: GR4: 0 = #0000 = 0000000000000000
```

```

#0000: GR5:      0 = #0000 = 0000000000000000
#0000: GR6:      0 = #0000 = 0000000000000000
#0000: GR7:      0 = #0000 = 0000000000000000
#0000: SP:       16 = #0010 = 0000000000010000
#0000: PR:       0 = #0000 = 0000000000000000
#0000: FR (OF SF ZF): 000
#0000: Memory:::
#0000: adr : 0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E
-----
#0000: 0000: 1010 0007 2210 0008 1110 0009 8100 0003 0002 0000 0000 0000 0000 0000 0000

#0002: Register:::
#0002: GR0:      0 = #0000 = 0000000000000000
#0002: GR1:      3 = #0003 = 0000000000000011
#0002: GR2:      0 = #0000 = 0000000000000000
#0002: GR3:      0 = #0000 = 0000000000000000
#0002: GR4:      0 = #0000 = 0000000000000000
#0002: GR5:      0 = #0000 = 0000000000000000
#0002: GR6:      0 = #0000 = 0000000000000000
#0002: GR7:      0 = #0000 = 0000000000000000
#0002: SP:       16 = #0010 = 0000000000010000
#0002: PR:       2 = #0002 = 0000000000000010
#0002: FR (OF SF ZF): 000
#0002: Memory:::
#0002: adr : 0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E
-----
#0002: 0000: 1010 0007 2210 0008 1110 0009 8100 0003 0002 0000 0000 0000 0000 0000 0000

#0004: Register:::
#0004: GR0:      0 = #0000 = 0000000000000000
#0004: GR1:      5 = #0005 = 0000000000000101
#0004: GR2:      0 = #0000 = 0000000000000000
#0004: GR3:      0 = #0000 = 0000000000000000
#0004: GR4:      0 = #0000 = 0000000000000000
#0004: GR5:      0 = #0000 = 0000000000000000
#0004: GR6:      0 = #0000 = 0000000000000000
#0004: GR7:      0 = #0000 = 0000000000000000
#0004: SP:       16 = #0010 = 0000000000010000
#0004: PR:       4 = #0004 = 0000000000000100
#0004: FR (OF SF ZF): 000
#0004: Memory:::
#0004: adr : 0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E
-----
#0004: 0000: 1010 0007 2210 0008 1110 0009 8100 0003 0002 0000 0000 0000 0000 0000 0000

#0006: Register:::
#0006: GR0:      0 = #0000 = 0000000000000000
#0006: GR1:      5 = #0005 = 0000000000000101
#0006: GR2:      0 = #0000 = 0000000000000000

```

```

#0006: GR3:      0 = #0000 = 0000000000000000
#0006: GR4:      0 = #0000 = 0000000000000000
#0006: GR5:      0 = #0000 = 0000000000000000
#0006: GR6:      0 = #0000 = 0000000000000000
#0006: GR7:      0 = #0000 = 0000000000000000
#0006: SP:       16 = #0010 = 0000000000010000
#0006: PR:        6 = #0006 = 0000000000000110
#0006: FR (OF SF ZF): 000
#0006: Memory:::
#0006: adr : 0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E
-----
#0006: 0000: 1010 0007 2210 0008 1110 0009 8100 0003 0002 0005 0000 0000 0000 0000 0000

```

2.3.1 特定のレジスタを表示

addl.caslのレジスタやメモリの中で、実行中に値が変化しているのはGR1だけです。こうした場合は、grepを使って表示される内容を絞り込むことで動作を検証しやすくなります。

```

$ casl2 -t addl.casl | grep 'GR1:'
#0000: GR1:      0 = #0000 = 0000000000000000
#0002: GR1:      3 = #0003 = 0000000000000011
#0004: GR1:      5 = #0005 = 0000000000000101
#0006: GR1:      5 = #0005 = 0000000000000101

```

この内容を、先に出力したアセンブル結果と比較してください。次の表のように、PRとGR1、命令行が対応していることがわかります。

PR	GR1	命令行
#0000	#0000	(なし)
#0002	#0003	LD GR1,A
#0004	#0004	ADDL GR1,B
#0006	#0006	ST GR1,C

2.3.2 プログラム終了時の値を表示

grepとtailを組み合わせれば、プログラム終了時の値を表示できます。

addl.caslでプログラム終了時のGR1の値を確認するには、次のようにします。

```

$ casl2 -t addl.casl | grep 'GR1:' | tail -1
#0006: GR1:      5 = #0005 = 0000000000000101

```

sum_10.caslは、1から10までの整数の和を求め、GR2に格納してからメモリにストア（書き込み）します。

```

$ cat sum_10.casl
;;; sum_10.casl
;;; 1から10までの整数をすべて加算した値をメモリーに格納する
MAIN    START
        XOR     GR2,GR2      ; GR2を初期化
        LD     GR1,FST      ; GR1に初項をメモリーから転送
LOOP    ADDL   GR2,GR1      ; ループ先頭。GR2 <- GR2 + GR1
        ADDL   GR1,STEP     ; GR1 <- GR1 + 公差
        CPL   GR1,LST      ; GR1が末項より大きい場合は終了
        JPL   FIN          ; ↓
        JUMP  LOOP         ; ループ終端
FIN     ST     GR2,RST      ; GR2の結果をメモリーに転送
        RET

```

```

FST    DC    1          ; 初項
LST    DC    10       ; 末項
STEP   DC    1        ; 公差
RST    DS    1        ; 結果
      END

```

sum_10.caslでプログラム終了時の GR2 の値を確認するには、次のようにします。

```

$ casl2 -t sum_10.casl | grep 'GR2:' | tail -1
#000E: GR2:      55 = #0037 = 0000000000110111 = '7'

```

2.3.3 プログラムのステップ数を表示

grepとwcを組み合わせれば、プログラムのステップ数を表示できます。

```

$ casl2 -t hello.casl | grep 'GR1:' | wc -l
11
$ casl2 -t add1.casl | grep 'GR1:' | wc -l
3

```

sum_10.caslはプログラム内にループがあるため、ステップ数が大きくなります。

```

$ casl2 -t sum_10.casl | grep 'GR2:' | wc -l
53

```

2.4 アセンブルと実行を別に行う

casl2に-Oファイル名を指定すると、オブジェクトファイルを作成できます。

```

$ casl2 -Ohello.o hello.casl

```

作成されたオブジェクトファイルの内容は、odを使って確認できます。テキストファイルではないため、catなどでは確認できません。

```

$ od -t x2 hello.o
0000000 7001 0000 7002 0000 1210 0013 1220 0020
0000020 f000 0002 1210 0021 1220 0022 f000 0002
0000040 7120 7110 8100 0048 0065 006c 006c 006f
0000060 002c 0020 0057 006f 0072 006c 0064 0021
0000100 000d 000a 0001
0000106

```

オブジェクトファイルの実行には、comet2を使います。

```

$ comet2 hello.o
Hello, World!

```

2.5 1語の解析

CASL IIでは、1語（1 word、16ビット）を単位としてデータが処理されます。dumpwordは、指定した1語を10進数、16進数、2進数で表示します。

```

$ dumpword 72
72:      72 = #0048 = 0000000001001000 = 'H'

```

2.6 CASL II ライブラリの使用

YACASL2のas/casl2libディレクトリには、CASL IIで記述されたライブラリファイルが格納されています。

このフォルダには、たとえば次のようなプログラムが含まれています。

```

OUTL      out1.casl. GR1 に格納された値を、0 以上 65535 以下の整数として出力します。

```

- OUTA outa.casl。GR1に格納された値を、-32767以上32767以下の整数として出力します。
- MULL mull.casl。GR1とGR2に格納された値を0以上65535以下の整数と見なし、積をGR3に格納します。
- DIVL divl.casl。GR1とGR2に格納された値を0以上65535以下の整数と見なし、商をGR3、剰余をGR0に格納します。

2.6.1 数値を出力する

3と1の和を求めるaddl.caslで演算結果を出力するには、まずaddl.caslを編集します。CASL IIのCALL命令でOUTLを副プログラムとして呼び出します。

```
$ cat addl_outl.casl
MAIN    START
        LD      GR1,A
        ADDL    GR1,B
        CALL    OUTL
        RET
A       DC      3
B       DC      1
        END
```

変更したらcasl2を、複数のファイルを指定して実行します。

```
$ casl2 addl_outl.casl ~/yacasl2/as/casl2lib/outl.casl
4
```

3 casl2の呼び出し

casl2は、引数として指定された CASL ファイルをアセンブルし、仮想マシン COMET II 上で実行します。CASL ファイルは、アセンブラ言語 CASL II で記述されたテキストファイルです。引数が指定されない場合は、エラーメッセージを表示して終了します。

```
$ casl2 hello.casl
```

複数の CASL ファイルを指定することで、副プログラムを呼び出すこともできます。

```
$ casl2 add1_out1.casl ~/yacasl2/as/casl2lib/out1.casl
```

オプション

casl2は、次のオプションを指定できます。

- s
--source CASL ファイルの内容を表示します。
- l
--label ラベルの一覧を次の形式で表示します。表示後、ほかの作業を続行します。
 <プログラム名>.<ラベル名> ---> <アドレスの 16 進数表現>
- L
--labelonly
 -lと同じ形式でラベルの一覧を表示します。表示後、ほかの作業は続行せず、終了します。
- a
--assembledetail
 アセンブル詳細結果を表示し、ほかの作業を続行します。
- A
--assembledetailonly
 アセンブル詳細結果を表示して終了します。
- o<OBJECTFILE>
--assembleout<OBJECTFILE>
 アセンブル結果をオブジェクトファイル<OBJECTFILE>に出力し、ほかの作業を続行します。出力されたオブジェクトファイルは、comet2で実行できます。オブジェクトファイルを指定しない場合、出力先は a.o です。オブジェクトファイルは 1 つだけ指定できます。
- O [<OBJECTFILE>]
--assembleoutonly [<OBJECTFILE>]
 アセンブル結果をオブジェクトファイル<OBJECTFILE>に出力し、終了します。出力されたオブジェクトファイルは、comet2で実行できます。オブジェクトファイルを指定しない場合、出力先は a.o です。オブジェクトファイルは 1 つだけ指定できます。
- t
--trace
--tracearithmetic
 プログラム実行中のレジスタの値を次の形式で表示します。
 <PR 値の 16 進数表現>: <レジスタ>: <値の 10 進数表現> =
 <値の 16 進数表現> = <値の 2 進数表現>['=' 文字']
 - <PR 値の 16 進数表現>と<レジスタ>、<値の 16 進数表現>は、4 けたの 16 進数で表されます。<PR 値の 16 進数表現>と<値の 16 進数表現>には、先頭に#が付きます。範囲は#0000から#FFFFです

- <値の 10 進数表現>は符号の付いた 10 進数です。範囲は-32768 から 32767 です。
- <値の 2 進数表現>は、16 けたの 2 進数で表されます。範囲は、0000000000000000 から 1111111111111111 です
- [= '文字']は、レジストリの値が「文字の組」の範囲に含まれる場合に表示されます。

表示されるレジスタには、次の種類があります。

GRO GR1 GR2 GR3 GR4 GR5 GR6 GR7

汎用レジスタ

SP スタックポインタ

PR プログラムレジスタ

FR フラグレジスタ

例えば、次のように表示されます。

```
#0002: GR1:      3 = #0003 = 0000000000000011
```

-T

--tracelogical

-tと同じように、プログラム実行中のレジスタの値を表示します。ただし-tと異なり、<値の 10 進数表現>は符号の付かない 10 進数です。値の範囲は 0 から 65535 です。

-d

--dump メモリの内容をすべて表示します。

-M <MEMORYSIZE>

--memorysize <MEMORYSIZE>

アセンブルおよび実行時のメモリサイズ<MEMORYSIZE>を 0 から 65535 の範囲で指定します。指定しない場合、512 です。

-C <CLOCKS>

--clocks <CLOCKS>

実行時のクロック周波数<CLOCKS>を 0 より大きい整数で指定します。指定しない場合、クロック周波数は 5000000 です。

-v

--version

cas12のバージョンを表示して終了します。

-h

--help cas12の使用方法を表示して終了します。

4 comet2の呼び出し

comet2は、引数として指定されたオブジェクトファイルを仮想マシン COMET II 上で実行します。オブジェクトファイルは、cas1に-oまたは-oを指定して出力します。

```
$ comet2 hello.o
```

引数で指定できるオブジェクトファイルは1つだけです。引数が指定されない場合は、エラーメッセージを表示して終了します。複数の引数を指定した場合、2番目以降の引数は無視されます。

オプション

comet2は、次のオプションを指定できます。

-t

--trace

--tracearithmetic

プログラム実行中のレジスタの値を次の形式で表示します。<値の10進数表現>は符号の付いた10進数です。範囲は-32768から32767です。

<PR 値の16進数表現>: <レジスタ>: <値の10進数表現> = <値の16進数表現> = <値の2進数表現> [= '文字']

- <PR 値の16進数表現>と<値の16進数表現>は、先頭に#が付いた4けたの16進数で表されます。範囲は、#0000から#FFFFです
- <値の2進数表現>は、16けたの2進数で表されます。範囲は、0000000000000000から1111111111111111です
- [= '文字']は、レジストリの値が「文字の組」の範囲に含まれる場合に表示されます。

例えば、次のように表示されます。

```
#0002: GR1:      3 = #0003 = 0000000000000011
```

表示されるレジスタには、次の種類があります。

```
GRO GR1 GR2 GR3 GR4 GR5 GR6 GR7
```

汎用レジスタ

SP スタックポインタ

PR プログラムレジスタ

FR フラグレジスタ

-T

--tracelogical

-tと同じように、プログラム実行中のレジスタの値を表示します。ただし、-tと異なり、<値の10進数表現>は符号の付かない10進数です。値の範囲は0から65535です。

-d

--dump メモリの内容をすべて表示します。

-M <MEMORYSIZE>

--memorysize <MEMORYSIZE>

実行時のメモリサイズ<MEMORYSIZE>を0から65535の範囲で指定します。指定しない場合、512です。

-C <CLOCKS>

--clocks <CLOCKS>

実行時のクロック周波数<CLOCKS>を0より大きい整数で指定します。指定しない場合、5000000です。

```
-v  
--version    comet2のバージョンを表示して終了します。  
  
-h  
--help      comet2の使用方法を表示して終了します。
```

5 dumpwordの呼び出し

dumpwordは引数として指定された数値を、整数、#0000 から#FFFF までの範囲の 16 進数、2 進数で表示します。文字の組に該当する場合は、「=」のうしろに文字が表示されます。引数は、10 進数または先頭に「#」の付いた 16 進数で指定します。表示される整数は、オプションにより符号付きか符号なしかを指定します。オプションなしの場合は符号付きです。整数の範囲は、符号付きの場合は-32768 以上 32767 以下、符号なしの場合は 0 以上 65535 以下です。

```
$ dumpword 10
10:      10 = #000A = 0000000000001010 = '\n'
```

引数が指定されない場合は、使い方を表示して終了します。複数の引数を指定した場合、1 つ目の引数だけが表示され、2 つ目以降の引数は無視されます。

注意

マイナスの数や 16 進数はシェルの仕様により、そのままでは指定できません。

マイナスの数を指定するときは、次のように--を付けます。

```
$ dumpword -- -72
-72:     -72 = #FFB8 = 1111111110111000
```

先頭に「#」を付けて 16 進数を指定するときは、次のように「'」で囲みます。

```
$ dumpword '#0048'
#0048:   72 = #0048 = 0000000001001000 = 'H'
```

オプション

dumpwordは、次のオプションを指定できます。

```
-a
--arithmetic
    出力される整数の範囲を-32,768 以上 32,767 以下にします。オプションなしの場合と
    同じです。

-l
--logical
    出力される整数の範囲を 0 以上 65,535 以下にします。

-v
--version
    dumpwordのバージョンを表示して終了します。

-h
--help
    dumpwordの使用方法を表示して終了します。
```