CASL II 実習マニュアル

第零版 revision02

大黒学

CASL II 実習マニュアル・第零版 revision02
著者——大黒学

2013 年　1 月 10 日（木）　第零版発行
2014 年 10 月　2 日（木）　第零版 revision02 発行

目次

第 1 章　CASL II の基礎

1.1　プログラム

1.1.1　文書と言語

文字を並べることによって何かを記述したものは、「文書」と呼ばれます。

文書を作るためには、記述したいことを意味として持つように、文字を並べていく必要があります。そして、そのためには、文字をどのように並べればどのような意味になるかということを定めた規則が必要になります。そのような規則は、「言語」と呼ばれます。

人間に読んでもらうことを第一の目的とする文書を書く場合は、日本語や中国語やアラビア語のような、「自然言語」と呼ばれる言語が使われます。自然言語というのは、人間の社会の中で自然発生的に形成された言語のことです。

言語には、自然言語のほかに、人間が意図的に設計することによって作られた、「人工言語」と呼ばれるものもあります。人間ではなくてコンピュータに読んでもらうことを第一の目的とする文書を書く場合は、通常、自然言語ではなくて人工言語が使われます。

1.1.2　プログラムとプログラミング

コンピュータに何らかの動作を実行させるためには、実行してほしいことがどんな動作なのかということを記述した文書をコンピュータに与える必要があります。そのような文書は、「プログラム」と呼ばれます。

プログラムを作成するためには、プログラムを書くという作業だけではなくて、プログラムの構造を設計したり、プログラムの動作をテストしたり、その不具合を修正したりするというような、さまざまな作業が必要になります。そのような、プログラムを作成するために必要となるさまざまな作業の全体は、「プログラミング」と呼ばれます。

1.1.3　プログラミング言語

プログラムというのも文書の一種ですから、それを書くためには何らかの言語が必要になります。プログラムを書く場合には、プログラムを書くことだけを目的として作られた人工言語を使うのが普通です。そのような、プログラムを書くための専用の言語は、「プログラミング言語」と呼ばれます。

プログラミング言語には、たくさんの種類があります。例を挙げると、Fortran、COBOL、Lisp、Pascal、Basic、C、AWK、Smalltalk、ML、Prolog、Perl、PostScript、Tcl、Java、Ruby、…… というように、枚挙にいとまがないほどです。

1.2　アセンブラ言語

1.2.1　機械語

第 1.1 節で説明したように、プログラミング言語にはたくさんの種類があります。しかし、コンピュータのハードウェアは、ほとんどすべてのプログラミング言語を理解することができません。コンピュータのハードウェアに理解することができるのは、「機械語」(machine language) または「マシン語」と呼ばれるひとつの言語だけです。

ハードウェアの仕様が異なるコンピュータは、異なる機械語を持っています。ですから、「機械語」と呼ばれるプログラミング言語は、一種類だけではありません。コンピュータのハードウェアの仕様ごとに、さまざまな機械語が存在することになります。

1.2.2　言語処理系

コンピュータのハードウェアには理解することのできないプログラミング言語で書かれたプログラムをコンピュータに実行させるためには、そのためのプログラムが必要になります。そのようなプログラムは、「言語処理系」(language processor) または単に「処理系」(processor) と呼ばれます。

処理系は、コンパイラやインタプリタなどに分類されます。

「コンパイラ」(compiler) というのは、プログラムを機械語に翻訳するプログラムのことで、「インタプリタ」(interpreter) というのは、プログラムを逐次的に解釈しながら実行するプログラムのことです。

1.2.3 アセンブラ言語とは何か

機械語でプログラムを書くことは、ハードウェアに大きく依存するプログラムが必要な場合に要求される仕事です。しかし、人間が機械語を使ってプログラムを書いたり、人間が機械語で書かれたプログラムを読んだりすることは、きわめて困難です。そのため、機械語でプログラムを書くことが要求される場合、通常は、機械語を使う代わりに、それよりも少しだけ人間にとってプログラムを書きやすくて読みやすい、「アセンブラ言語」(assembler language) と呼ばれる言語が使われます[1]。

機械語とアセンブラ言語のプログラムには、大きな共通点があります。それは、どちらの言語で書かれたプログラムも、「命令」(instruction) と呼ばれるものが並んでできている、というところです。個々の命令は、コンピュータのハードウェアが実行することのできる基本的な動作を表現しています。機械語の命令とアセンブラ言語の命令とは、原則的には一対一で対応しています。

機械語の命令とアセンブラ言語の命令とのあいだには、相違点もあります。それは、コンピュータの動作を表現する方法の違いです。機械語の命令は、コンピュータの動作をビット列で表現しています。それに対して、アセンブラ言語の命令は、コンピュータの動作を英字や数字や空白やコンマなどを使って表現しています[2]。

たとえば、機械語の一つの命令は、

```
0001 0100 0100 0101
```

というようなビット列ですが、アセンブラ言語の命令は、

```
␣LD␣GR4,GR5
```

というように、英字、数字、空白、コンマなどを使って書きます（␣は、「ここに 1 個の空白がある」ということを示している図形です）。

機械語よりもアセンブラ言語のほうが人間にとってプログラムを書きやすくて読みやすい理由は、このような命令の表現方法の違いにあります。

機械語というのはコンピュータのハードウェアの仕様ごとに異なっていますので、アセンブラ言語もそれと同様に、コンピュータのハードウェアの仕様ごとに異なっています。

1.2.4 アセンブラ

アセンブラ言語で書かれたプログラムを機械語に翻訳することを、「アセンブルする」(assemble) と言います。そして、プログラムをアセンブルする処理系は、「アセンブラ」(assembler) と呼ばれます。

アセンブラがアセンブルの対象とするプログラムは「ソースコード」(source code) と呼ばれます。そして、アセンブラによってソースコードがアセンブルされた結果は「オブジェクトコード」(object code) と呼ばれます。

アセンブラ言語で書かれたプログラムは、アセンブラを使ってアセンブルすることも可能ですが、人間の手作業でアセンブルすることも可能です。人間の手作業によるアセンブルは、「ハンドアセンブル」(hand assenble) と呼ばれます。

1.2.5 CASL II について

「情報処理の促進に関する法律」という日本の法律は、その第 7 条で、

> 経済産業大臣は、情報処理に関する業務を行う者の技術の向上に資するため、情報処理に関して必要な知識及び技能について情報処理技術者試験を行う。

と定めています。この条項にもとづいて、情報処理技術者試験センター[3]という組織によって、「情報処理技術者試験」と呼ばれる国家試験が実施されています。

情報処理技術者試験にはいくつかの試験区分があるのですが、それらの試験区分のひとつに、「基本情報技術者試験」と呼ばれるものがあります。

基本情報技術者試験の出題範囲の中には、プログラミングが含まれています。プログラミングの問題は、C、COBOL、Java、CASL II という 4 種類のプログラミング言語のそれぞれを使って作られています。受験者は、それらの問題の中から、自分の得意な言語を使った問題を選択して解答することができます。

---

[1] アセンブラ言語は、「アセンブリ言語」(assembly language) と呼ばれることもあります。
[2] アセンブラ言語の命令は、「ニーモニック」(mnemonic) と呼ばれることもあります。
[3] URL は `http://www.jitec.jp/` です。

CASL II というのは、COMET II というコンピュータのアセンブラ言語です。ただし、COMET II は、実在するコンピュータではなくて、基本情報技術者試験のために仕様が策定された架空のコンピュータです。

1.2.6 CASL II アセンブラと COMET II シミュレータ

CASL II で書かれたプログラムを COMET II の機械語に翻訳するアセンブラは、「CASL II アセンブラ」(CASL II assembler) と呼ばれます。

CASL II の実習をするためには CASL II アセンブラが必要だ、ということは言うまでもありません。しかし、それだけでは不十分です。その理由は、COMET II というのが実在しない架空のコンピュータだからです。すなわち、CASL II で書かれたプログラムを CASL II アセンブラで COMET II の機械語に翻訳したとしても、それを実行することのできるコンピュータが実在しませんので、そこで手詰まりになってしまうというわけです。

CASL II のプログラムをアセンブルした結果を実行するためには、実在するハードウェアの上で COMET II をシミュレートするプログラムが必要になります。そのようなプログラムは、「COMET II シミュレータ」(COMET II simulator) と呼ばれます。

インターネットの上には、CASL II の実習をするためのソフトとしてさまざまなものが公開されています。それらのほとんどすべては、CASL II アセンブラと COMET II シミュレータとをひとつのプログラムに統合しているか、またはそれらを別々のプログラムにして、ひとつのアーカイブに同梱しているかのどちらかです。したがって、それらのソフトのうちのどれかひとつをダウンロードしてインストールすれば、CASL II の実習をするための環境が整うことになります。

1.2.7 この文章について

この文章（「CASL II 実習マニュアル」）は、プログラミング言語として CASL II を使って、プログラムというものの書き方について説明する、ということを目的とするチュートリアルです。

このチュートリアルは、情報処理技術者試験センターが開発した「CASL II シミュレータ」というソフトを使って CASL II の実習をすることを想定して書かれています。

1.3 COMET II

1.3.1 COMET II の基礎

CASL II というのは、COMET II というコンピュータのアセンブラ言語です。したがって、CASL II を理解するためには、COMET II というコンピュータの仕様を理解しておく必要があります。

COMET II は、ごく普通のノイマン型のコンピュータです。「ノイマン型」(von Neumann architecture) というのは、次の二つの方式を採用しているコンピュータのタイプのことです。

プログラム内蔵方式	英語では stored program method。処理の対象となるデータが主記憶装置に格納されるのと同じように、プログラムも主記憶装置に格納されるという方式。
逐次制御方式	英語では sequential control method。プログラムを構成している命令がひとつひとつ順番に取り出されて実行されるという方式。

1.3.2 主記憶装置

コンピュータは、その内部に「主記憶装置」(primary memory) と呼ばれる装置を持っています。これは、コンピュータがデータを処理するための作業場所だと考えることができます。コンピュータがデータを処理するためには、処理の対象となるデータと、処理の手順となるプログラムを、主記憶装置に置いておく必要があります。

主記憶装置の中には、「語」(word) と呼ばれる記憶場所が一列に並んでいます。

COMET II の主記憶装置は、65536 個の語から構成されています。

COMET II の機械語のプログラムを構成している個々の命令は、主記憶装置の 1 語または 2 語に格納されます。

1.3.3 アドレス

主記憶装置を構成している個々の語は、「アドレス」(address) と呼ばれる 0 またはプラスの整数によって識別されます。日本語では、アドレスは、「番地」という助数詞を付けて、0 番地、1

番地、2 番地、……というように呼ばれます。

COMET II の場合も、その主記憶装置を構成している 65536 のそれぞれの語に対して、0 番地から 65535 番地までのアドレスが、順番に割り振られています。

1.3.4　ビット

0 または 1 のどちらかの状態を保持することのできるもの、あるいは 2 進数の一桁は、「ビット」(bit) と呼ばれます。主記憶装置を構成している個々の語は、何個かのビットを一列に並べたものです。COMET II の場合、その主記憶装置を構成している個々の語は、16 個のビットから構成されています。

COMET II の語を構成している 16 個のビットのそれぞれには、0 番から 15 番までの番号が与えられています。番号は、左端（最上位ビット）が 15 番で、右端（最下位ビット）が 0 番です。

1.3.5　整数の表現

COMET II では、整数は 16 ビットの 2 進数で表現されます。ですから、符号のない整数ならば、0 から $2^{16}-1$ まで、つまり 0 から 65535 までを表現することができます。アドレスは 0 番地から 65535 番地までですので、アドレスも 1 語によって表現することができることになります。

符号のある整数を表現する場合、マイナスの整数は、2 の補数で表現されます。その場合、表現することのできる整数の範囲は、-2^{15} から $2^{15}-1$ まで、つまり −32768 から 32767 までです。

1.3.6　文字の表現

COMET II では、文字は、日本工業規格が策定した JIS X 0201 という文字コードによって表現されます。これは、英字、数字、特殊文字、片仮名を 8 ビットのビット列で表現する文字コードです。

COMET II では、1 個の文字は 1 語に格納されます。ところが、COMET II の語は 16 ビットなのに対して、JIS X 0201 で表現された文字は 8 ビットです。8 ビットの文字は、16 ビットの語の中に、どのように格納されるのでしょうか。

COMET II では、文字を語に格納する場合、文字は下位の 8 ビット（7 番から 0 番まで）に格納されて、上位の 8 ビット（15 番から 8 番まで）はすべて 0 になります。たとえば、`A` という文字は、

```
0100 0001
```

というビット列で表現されますので、語には、

```
0000 0000 0100 0001
```

というように格納されます。

1.3.7　CPU

コンピュータは、その内部に「CPU」(central processing unit) と呼ばれる装置を持っています。これは、主記憶装置に格納されている命令にしたがって、さまざまな処理を実行する装置です。

CPU は、次の 3 種類のものから構成されます。

制御装置　英語では control unit。主記憶装置から命令を取り出して、それがあらわしている動作が実行されるように、コンピュータを構成しているさまざまな装置の動作を制御する装置。

ALU　英語では arithmetic logic unit。算術演算と論理演算を実行する装置。

レジスタ　英語では register。処理の対象となるデータや何らかの状態を保持するための記憶装置。主記憶装置に比べると、はるかに少量のデータしか保持できない。しかし、主記憶装置よりもはるかに高速で読み書きをすることができる。

1.3.8　レジスタ

COMET II の CPU は、次の 4 種類のレジスタを持っています。

汎用レジスタ　英語では general register、略称は GR。演算の対象となるデータを保持するレジスタ。演算の結果も、汎用レジスタに格納される。COMET II は

	8 個の GR を持っていて、それぞれの GR は、GR0、GR1、・・・、GR7 という名前で識別される。長さは、すべて 16 ビット。
フラグレジスタ	英語では flag register、略称は FR。命令を実行した結果に関する状態を保持するレジスタ。COMET II は 1 個の FR を持っていて、その長さは 3 ビット。
スタックポインタ	英語では stack pointer、略称は SP。スタックの最上段のアドレスを保持しているレジスタ。COMET II は 1 個の SP を持っていて、その長さは 16 ビット。
プログラムレジスタ	英語では program register、略称は PR。次に実行する命令が格納されている主記憶装置のアドレスを保持しているレジスタ。命令を実行すると、その命令の長さ（単位は語）が加算されて、次の命令のアドレスに変化する。COMET II は 1 個の PR を持っていて、その長さは 16 ビット。

レジスタについては、必要になったときに、もう少し詳しく説明することにしたいと思います。

1.4 CASL II の基礎の基礎

1.4.1 命令

アセンブラ言語を使ってプログラムを書くというのは、「命令」(instruction) と呼ばれるものを並べていくということです。それらの命令の大多数は、コンピュータのハードウェアが実行することのできる基本的な動作を表現しています。

アセンブラ言語で書かれたプログラムをアセンブルすると、原則的には、そのプログラムを構成しているそれぞれの命令は、機械語の 1 個の命令に変換されます。ただし、アセンブラ言語の命令の中には、機械語の命令に変換されないものや、2 個以上の命令に変換されるものもあります。

1.4.2 行

改行によって区切られている文字列のそれぞれの部分は、「行」(line) と呼ばれます。

CASL II のプログラムは、いくつかの行から構成されます。

CASL II のプログラムを構成する行は、次の二つの種類に分類することができます。

命令行　命令が書かれている行。

注釈行　注釈（第 1.8 節参照）だけが書かれている行。

1.4.3 命令行

CASL II では、命令は、1 行の中に 1 個だけ書くことができます。2 個以上の命令を 1 行に書いたり、1 個の命令を 2 行以上に分けて書いたりする、ということはできません。

命令行は、次の 4 種類のものから構成されます。

- 命令コード (operation code)
- オペランド (operand)
- ラベル (label)
- 注釈 (comment)

これらの構成要素のうちで、命令行がかならず含んでいないといけないのは、命令コードだけです。それ以外の 3 種類は、かならず書かなければならないというわけではありません。

命令コードについては第 1.5 節で、オペランドについては第 1.6 節で、ラベルについては第 1.7 節で、注釈については第 1.8 節で説明することにしたいと思います。

1.5 命令コード

1.5.1 命令コードの基礎

アセンブラ言語の命令には、さまざまな種類があります。それらの命令の種類は、「命令コード」と呼ばれる名前によって識別されます。

CASL II の命令コードは、2 文字から 5 文字までの英字の大文字によって作られています。たとえば、START、OUT、LD、ST というような命令コードがあります。

1.5.2　もっとも単純な命令行

命令行の中には、かならず命令コードが含まれていないといけません。そして、1 個の命令行の中に書くことのできる命令コードは、1 個だけです。1 個の命令行の中に 2 個以上の命令コードを書くことはできません。

命令コードを行の先頭にいきなり書く、ということはできません。命令コードの左側には、少なくとも 1 個の空白を書く必要があります。なぜなら、行が英字で始まっている場合、アセンブラは、その行の先頭にはラベルが書かれていると判断するからです。

NOP という命令があります。これは、「何もしない」という動作を意味している命令です。

NOP という命令を含む命令行を書く場合に最低限必要なものは、1 個の空白と命令コードだけです。したがって、

␣NOP

というのは、もっとも単純な命令行の一例です。

1.5.3　命令の大分類

CASL II の命令は、細かく分類すると 36 種類のものがあって、それぞれの種類ごとに、それを識別する命令コードが定められています。

CASL II の命令は、もう少し大きく、次の 3 種類に分類することも可能です。

アセンブラ命令	アセンブラに対する指示をあらわす命令。機械語の命令には変換されない。
マクロ命令	複数の機械語の命令に変換される命令。
機械語命令	1 個の機械語の命令に変換される命令。

1.6　オペランド

1.6.1　オペランドの基礎

アセンブラ言語では、動作の対象に関する記述のことを「オペランド」(operand) と呼びます。

第 1.5.2 項で説明したように、命令行の中には、かならず 1 個の命令コードが含まれていないといけません。それに対して、オペランドは、命令行の中に必ず書かなければならないものではありません。

たとえば、NOP という命令の場合、オペランドを書く必要はありません。NOP 命令がオペランドを必要としない理由は、それがあらわしている「何もしない」という動作の場合には、その動作の対象を記述する必要がないからです。

CASL II が持っている 36 種類の命令のうちで、オペランドを必要としない NOP のような命令は、きわめて少数です。大多数の命令は、対象を記述する必要のある動作を意味しています。

たとえば、LD という命令は、汎用レジスタから別の汎用レジスタへデータを転送する、または主記憶装置から汎用レジスタへデータを転送する、という動作を意味しています。この命令の場合、NOP 命令の場合とは違って、命令コードを書くだけでは動作を記述したことにはなりません。なぜなら、ただ単に、

␣LD

と書いただけでは、「データを転送する」ということしか分からないからです。「どこから」「どこへ」データを転送するのかということを記述しなければ、命令として完全なものにはならないのです。

1.6.2　オペランドの書き方の例

動作の対象を記述する必要がある場合、その対象は、オペランドとして記述されます。オペランドはどのような構文で書けばいいのかということは、それぞれの命令の種類ごとに決まっています。たとえば、LD 命令のオペランドは、

[どこへ] , [どこから]

という構文で書きます。たとえば、

```
GR4,GR5
```

というオペランドは、「汎用レジスタの GR5 から汎用レジスタの GR4 へ」という意味になります。

命令コードとオペランドとのあいだは、1 個以上の空白で区切る必要があります。たとえば、汎用レジスタの GR5 の内容を汎用レジスタの GR4 へ転送する命令を書く場合は、

```
␣LD␣GR4,GR5
```

というように、命令コードとオペランドとのあいだを空白で区切らないといけません。

1.7 ラベル

1.7.1 ラベルの基礎

第 1.3.3 項で説明したように、主記憶装置を構成している個々の語は、「アドレス」(address) と呼ばれる 0 またはプラスの整数によって識別されます。

機械語のプログラムの中にある、主記憶装置に対する読み書きをする命令は、読み書きの対象となる主記憶装置の語を指定するために、その語のアドレスを含んでいます。

アセンブラのプログラムも、本質的には機械語と同じです。ただし、アセンブラのプログラムの中では、語は、0 またはプラスの整数ではなくて、「ラベル」(label) と呼ばれるものによって指定されます。

ラベルというのは、アドレスに与えられた名前のことです。語に与えられた名前ではない、という点に注意してください。あくまで、35 番地とか 72 番地というような整数に与えられた名前です。

1.7.2 ラベルの作り方

ラベルは、次の規則に従ってさえいれば、プログラムを書く人が自由に作ることができます。

- 使うことのできる文字は、英字の大文字または数字。
- 長さは 8 文字まで。
- 先頭の文字は英字の大文字でなければならない。
- 汎用レジスタの名前（GR0 から GR7 まで）は予約語（あらかじめ意味が予約されている単語）なので、ラベルとしては使えない。

たとえば、A、A1、KAMENOTE などはラベルとして使うことができますが、次のような名前をラベルとして使うことはできません。

AC/DC	使うことのできない文字 (/) を含んでいる。
ISOGINCHAKU	長さが 8 文字を超えている。
2F	先頭の文字が数字。
GR7	汎用レジスタの名前と同じ。

1.7.3 ラベルの定義

アドレスに対して名前としてラベルを与えることを、ラベルを「定義する」(define) と言います。

ラベルを定義したいときは、そのラベルを命令行の先頭に書きます。そして、その右側に命令コードを書きます。ラベルと命令コードとのあいだは、1 個以上の空白で区切る必要があります。

命令行の先頭にラベルを書くと、主記憶装置の中にある、その命令行を翻訳した結果が格納されている部分の先頭のアドレスに対して、そのラベルが名前として与えられます。たとえば、

```
HOGE␣LD␣GR4,GR5
```

という命令行を書いたとすると、その先頭に書かれている HOGE というラベルは、主記憶装置の中にある、この LD 命令に対応する機械語の命令が格納されている語のアドレスに対して名前として与えられます。

COMET II の機械語の命令には、長さが 1 語のものと 2 語のものとがあります。たとえば、汎用レジスタから汎用レジスタへデータを転送する命令の長さは 1 語で、主記憶装置から汎用レジスタへデータを転送する命令の長さは 2 語です。

2 語の機械語の命令に翻訳される命令行でラベルを定義した場合、そのラベルは、その機械語の命令が格納されている 2 語のうちで、小さいほうのアドレスに対して名前として与えられます。

1.8　注釈

1.8.1　注釈の基礎

プログラムを読んで、それを理解するというのは、人間にとって非常に頭脳を酷使する作業だと言えます。ですから、プログラムを理解する上でヒントとなるようなことがそのプログラムの中に書かれていると、とても助かります。プログラムの中に含まれている、プログラムを読む人間に向けて書かれた文字列は、「注釈」(comment) と呼ばれます。

1.8.2　注釈を書くための規則

プログラムの中に注釈を書く場合には、使っているプログラミング言語が定めている規則に従ってそれを書く必要があります。CASL II の場合、注釈は次のような規則に従って書かないといけません。

- 注釈は、アセンブラが許すならば、どんな文字を使って書いてもかまわない。
- 注釈を書くことのできる場所は、命令行の末尾、または注釈行の中である。
- オペランドを持つ命令行の末尾に注釈を書く場合は、そのオペランドの右側に 1 個以上の空白を書いて、その右側に注釈を書く。

  例 ␣LD␣GR4,GR5␣私は注釈です。
- オペランドを持たない命令行の末尾に注釈を書く場合は、命令コードの右側に 1 個以上の空白を書いて、その右側にセミコロン (;) を書いて、その右側に注釈を書く。

  例 ␣NOP␣; 私は注釈です。
- 注釈行というのは、0 個以上の空白、セミコロン (;)、注釈を、この順序で並べることによってできる行のことである。

  例 ; 私は注釈です。

第 2 章　アセンブラ命令

2.1　アセンブラ命令の基礎

2.1.1　アセンブラ命令とは何か

第 1.5.3 項で説明したように、アセンブラに対する指示をあらわす命令は、「アセンブラ命令」と呼ばれます。

アセンブラ命令は、アセンブラに対する指示をあらわしている命令ですので、それに対応している機械語の命令というものは存在しません。

2.1.2　アセンブラ命令の種類

CASL II には、次の 4 種類のアセンブラ命令があります。

START	プログラムの先頭をアセンブラに知らせる。
END	プログラムの終わりをアセンブラに知らせる。
DC	特定のデータを主記憶装置に設定する。
DS	主記憶装置の上に領域を確保する。

START 命令と END 命令についてはこの節で説明しますが、DC 命令については第 2.3 節で、DS 命令については第 2.4 節で説明することにしたいと思います。

2.1.3　START 命令

START は、プログラムの先頭をアセンブラに知らせる命令です。

CASL II のプログラムは、最初の命令がかならず START 命令でないといけません。そして、START 命令は、かならずラベルを定義しないといけません。

START 命令によって定義されたラベルは、「実行開始番地」と呼ばれるアドレスに名前として与えられます。実行開始番地というのは、機械語のプログラムの実行をそれから開始する命令のアドレスのことです。

START 命令のオペランドとしては、0 個または 1 個のラベルを書きます。
オペランドとして 1 個のラベルを書いた場合は、そのラベルを名前として持つアドレスが実行開始番地になります。たとえば、

```
A START B
```

という START 命令は、B というラベルを名前として持つアドレスを実行開始番地にします。ですから、A というラベルは、B と同じアドレスを意味することになります。
オペランドを書かなかった場合は、START 命令の次の命令を機械語に変換したものが格納されている語のアドレスが実行開始番地になります。つまり、機械語のプログラムの先頭を実行開始番地にするということです。たとえば、

```
C START
```

という START 命令は、機械語のプログラムの先頭のアドレスを実行開始番地にします。ですから、C というラベルは、機械語のプログラムの先頭のアドレスを意味することになります。

2.1.4 END 命令

END は、プログラムの終わりをアセンブラに知らせる命令です。
CASL II のプログラムは、最後の命令がかならず END 命令でないといけません。
END 命令にオペランドは必要ではありません。
END 命令では、ラベルを定義することはできません。

2.2 何もしないプログラム

2.2.1 何もしないプログラムの基礎

この節では、何もしない CASL II のプログラムを紹介したいと思います。
CASL II では、何もしないプログラムは、少なくとも 3 行で書くことができます。
第 2.1 節で説明したように、CASL II のプログラムは、その最初の命令が START で、最後の命令が END でないといけません。ですから、何もしないプログラムも、最初の命令は START で、最後の命令は END です。それでは、そのあいだに書かないといけない、もうひとつの命令というのは、いったい何なのでしょうか。
何もしないプログラムの START と END のあいだに書かないといけないのは、プログラムを終了させる命令です。

2.2.2 プログラムを終了させる命令

CASL II のプログラムは、かならず、自分の動作の最後に実行される命令として、プログラムを終了させる命令を含んでいないといけません。
プログラムを終了させたいとき、CASL II では、RET (RETurn from subroutine) という命令を書きます。この命令には、オペランドを書く必要はありません。

2.2.3 何もしないプログラムの例

それでは、何もしない CASL II のプログラムの例を、実際に書いてみましょう。
何もしないプログラムを、空白を節約して書くと、次のようになります。

```
NOTHING START
 RET
 END
```

CASL II のプログラムは、通常、命令コードの先頭の文字が縦方向に一直線に並ぶように、空白を入れて書きます。なぜなら、そうすることによって、プログラムが読みやすくなるからです。
上のプログラムに空白を加えて、命令コードの先頭の文字が縦方向に一直線に並ぶようにしてみましょう。そうすると、次のようになります。

```
NOTHING  START
         RET
         END
```

それでは、テキストエディターを使って、このプログラムを入力して、nothing.cas というファイルに保存してください。

ちなみに、CASL II のプログラムをファイルに保存する場合は、通常、 .cas という拡張子をファイル名に付けます。

2.2.4　何もしないプログラムのオブジェクトコード

それでは、 nothing.cas に保存したプログラムを、情報処理技術者試験センターの CASL II シミュレータを使ってアセンブルしてみてください。

情報処理技術者試験センターの CASL II シミュレータは、 .obj という拡張子を持つファイルにオブジェクトコードを出力します。このオブジェクトコードは、人間にも読みやすいように、テキストデータになっています。

nothing.cas は、次のようなオブジェクトコードに翻訳されます。

```
NAME:NOTHING
STARTadr:0000R
SIZE:0001
0000:8100
END
```

重要なのは、4 行目にある、

```
0000:8100
```

という部分です。これは、アドレスと、そのアドレスの語に格納される機械語の命令を、16 進数で記述したものです。コロン (:) の左側がアドレスで、右側が命令です。つまり、0 番地の命令が、

```
1000 0001 0000 0000
```

だということです。ちなみに、 8100 は、CASL II の RET に対応する COMET II の命令です。

2.2.5　仮のアドレスと実際のアドレス

ところで、先ほど見ていただいたオブジェクトコードでは、 RET 命令に対応する機械語の命令は、0 番地に格納されることになっていました。しかし、このアドレスは、あくまで仮のものです。

情報処理技術者試験センターの CASL II シミュレータは、オブジェクトコードを作成するとき、プログラムは主記憶装置の 0 番地から始まる領域に格納されると仮定して、アドレスを決定します。つまり、オブジェクトコードの中のアドレスは、プログラムの先頭の命令のアドレスを 0 番地としたときの相対的な位置をあらわしているのです。

主記憶装置の上でのプログラムの実際の位置は、オペレーティングシステムがそのプログラムを主記憶装置に格納するときに決定されることになります。

2.2.6　エラー

プログラムの中には、何らかの間違いが含まれていることがあります。そのような、プログラムの中に含まれている間違いは、「エラー」(error) と呼ばれます。

アセンブラは、ソースコードがエラーを含んでいる場合、そのエラーについてのメッセージを出力します。そのような、エラーについてのメッセージは、「エラーメッセージ」(error message) と呼ばれます。

たとえば、 GET という命令は CASL II にはありませんので、次のプログラムの 2 行目にはエラーがあります。

プログラムの例　error.cas

```
ERROR    START
         GET
         END
```

それでは、このプログラムをアセンブルしてみましょう。そうすると、次のようなエラーメッセージが出力されます。

```
OP   2:          GET
```

エラーメッセージの先頭に出力される 2 文字は、次のように、命令行のどこにエラーがあるのかということを示しています。

LB　ラベルのエラー。

OP　命令コードのエラー。

OD　オペランドのエラー。

その 2 文字の右側には、エラーのある行の行番号と、その行そのものが出力されます。

2.3 DC 命令

2.3.1 DC 命令の基礎

CASL II には、特定のデータを主記憶装置に設定するという機能があります。この機能は、DC (Define Constant) という命令を書くことによって利用することができます。

DC 命令は、アセンブラによって、主記憶装置に設定される何らかのデータに翻訳されます。そのデータは、機械語の命令として実行されることを意図したものではありませんので、もしもそれが機械語の命令として実行されると、予期しない動作が実行されることになりかねません。

ですから、DC 命令は、それが実行されることのないように、RET 命令よりも下に書くのが普通です。

DC 命令でラベルを定義すると、そのラベルは、DC 命令によって主記憶装置に設定されたデータのアドレスに対して与えられます（データが 2 語以上の場合は、その先頭のアドレス）。

2.3.2 定数

DC 命令には、オペランドを書く必要があります。

DC 命令のオペランドとして書かれるものは、「定数」(constant) と呼ばれます。定数というのは、特定のデータを記述したもののことです。DC 命令は、オペランドに書かれた定数があらわしているデータを主記憶装置に設定します。

定数には、次の 4 種類のものがあります。

10 進定数	10 進数で整数を記述した定数。
16 進定数	16 進数で整数を記述した定数。
文字定数	文字列を記述した定数。
アドレス定数	ラベルによって、それが与えられているアドレスを記述した定数。

2.3.3 10 進定数

10 進定数は、10 進数で整数を記述した定数です。記述することのできる範囲は、−32768 から 32767 までです。この範囲にない 10 進定数を書いた場合は、それを 2 進数で表記したときの下位 16 ビットだけが語に設定されます。

プログラムの例　decimal.cas

```
DECIMAL  START
         RET
         DC    -1
         END
```

それでは、このプログラムをアセンブルすることによってできたオブジェクトコードを見てみましょう。0 番地と 1 番地は、次のようになっています。

```
0000:8100
0001:FFFF
```

0 番地にある 8100 が RET 命令に相当する機械語の命令で、1 番地にある FFFF が、DC 命令によって設定されたデータです。

第 1.3.5 項で説明したように、COMET II では、符号のある整数は 2 の補数で表現されますので、−1 は、

```
1111 1111 1111 1111
```

という 2 進数で表現されます。これを 16 進数で表記したものが、FFFF です。

2.3.4 16 進定数

16 進定数は、4 桁の 16 進数で整数を記述した定数です。ただし、これが 16 進定数だということをアセンブラに知らせるために、先頭にシャープ (#) を書く必要があります。たとえば、10 進数の 15 をあらわす 16 進定数は、

```
#000F
```

と書きます。

プログラムの例　hexdec.cas

```
HEXDEC   START
         RET
         DC    #000F
         END
```

このプログラムをアセンブルして、オブジェクトコードを見てみましょう。そうすると、1 番地に、

```
0001:000F
```

というように、15 という整数が設定されているはずです。

2.3.5　文字定数

文字定数は、文字列を記述した定数です。ただし、これが文字定数だということと、文字定数がどこで終わっているかということをアセンブラに知らせるために、文字定数の先頭と末尾には、一重引用符 (’) を書く必要があります[1]。たとえば、 ABC という文字列をあらわす文字定数は、

```
’ABC’
```

と書きます。

第 1.3.6 項で説明したように、COMET II では、1 個の文字は 1 語に格納されます。ですから、ABC という文字列は、連続する 3 語の領域に格納されます。

プログラムの例　string.cas

```
STRING   START
         RET
         DC    ’ABC’
         END
```

このプログラムのオブジェクトコードを見ると、次のように、1 番地に A、2 番地に B、3 番地に C の文字コードが設定されているはずです。

```
0001:0041
0002:0042
0003:0043
```

文字定数を使って文字列を主記憶装置に設定する DC 命令でラベルを定義した場合、そのラベルは、文字列の先頭の文字が設定される語のアドレスに対して与えられます。

文字定数の中では、一重引用符という文字は、連続する 2 個の一重引用符によってあらわされます。たとえば、

```
fielder’s choice
```

という文字列をあらわす文字定数は、

```
’fielder’’s choice’
```

と書く必要があります。

2.3.6　アドレス定数

アドレス定数は、ラベルによって、それが与えられているアドレスを記述した定数です。

たとえば、 NAMAKO というラベルが 1 番地に与えられているとするとき、

```
DC NAMAKO
```

という DC 命令を書いたとすると、この DC 命令によって、1 という整数が主記憶装置に設定されることになります。

プログラムの例　address.cas

```
ADDRESS  START
         RET
HUNDRED  DC    100
         DC    HUNDRED
```

[1] 一重引用符は、「アポストロフィー」と呼ばれることもあります。

```
        END
```

このプログラムには、DC 命令が二つあります。1 個目の DC 命令は、100 という整数を 1 番地に設定します。そして、そのアドレスに対して、HUNDRED というラベルを与えます。そして、2 個目の DC 命令は、HUNDRED というラベルが与えられているアドレスを 2 番地に設定します。

このプログラムをアセンブルすることによってできたオブジェクトコードを見ると、1 番地と 2 番地は次のようになっています。

```
0001:0064
0002:0001R
```

ここで注意しないといけないことは、第 2.2.5 項で説明したように、オブジェクトコードの中で使われているアドレスは、あくまで仮のものだということです。たとえば、HUNDRED というラベルは、オブジェクトコードでは 1 番地というアドレスに与えられていますが、このプログラムが実際に主記憶装置に格納されたとき、そのアドレスが必ず 1 番地になるとは限りません。

オブジェクトコードをよく見ると、その 2 番地の内容は、

```
0002:0001R
```

というように、その末尾に R と書かれています。この R は、「自分の左側のアドレスは、プログラムの先頭を 0 番地と仮定したときの相対的 (relative) なアドレスである」ということを意味しています。

2.3.7 定数の列

これまで紹介したプログラムでは、DC 命令のオペランドとして、1 個の定数を書いていましたが、DC 命令のオペランドとして書くことができる定数は、1 個だけではありません。

DC 命令のオペランドは、

定数 , 定数 , ···

というように、2 個以上の定数をコンマ (,) で区切って並べて書くことも可能です。そうすると、それらの定数によってあらわされるデータが、オペランドの中に並んでいる順番と同じ順番で、主記憶装置に設定されることになります。ですから、

```
    DC    -1,#000F,'ABC',HUNDRED
```

という 1 個の DC 命令は、

```
    DC    -1
    DC    #000F
    DC    'ABC'
    DC    HUNDRED
```

という 4 個の DC 命令と同じ意味になります。

それでは、次のプログラムをアセンブルして、どのようなオブジェクトコードができるかを確認してみてください。

プログラムの例 sequence.cas

```
SEQUENCE START
         RET
HUNDRED  DC    100
         DC    -1,#000F,'ABC',HUNDRED
         END
```

2.4 DS 命令

2.4.1 DS 命令の基礎

CASL II には、主記憶装置の上に任意の大きさの領域を確保するという機能があります。この機能は、DS (Define Storage) という命令を書くことによって利用することができます。

機械語のプログラムの中で、DS 命令によって確保される領域がどのような位置に置かれるかというのは、その DS 命令が書かれている位置によって決定されます。ですから、DS 命令は、DC 命令の場合と同じように、確保された領域の中にあるデータが機械語の命令として実行されることのないように、RET 命令よりも下に書くのが普通です。

2.4.2 語数

DS命令には、オペランドを書く必要があります。

DS命令のオペランドとして書かれるものは、「語数」と呼ばれる、0またはプラスの整数をあらわしている10進数です。

DS命令を含むプログラムをアセンブルすると、その中のDS命令は、主記憶装置の上の領域に変換されます。領域の大きさは、オペランドとして書かれた語数によって指定されます。語数は、主記憶装置の上に確保される領域の大きさを、語を単位として指定していると解釈されます。たとえば、

```
DS 6
```

というDS命令は、6語の大きさの領域を確保します。

プログラムの例 ds.cas

```
DS      START
        RET
        DS    6
        END
```

このプログラムをアセンブルして、オブジェクトコードを見てみましょう。

```
NAME:DS
STARTadr:0000R
SIZE:0007
0000:8100
END
```

このように、DS命令は、機械語にも特定のデータにも変換されていません。しかし、6語の大きさの領域が確保されているということは、プログラムの大きさから判断することができます。オブジェクトコードの3行目に、

```
SIZE:0007
```

と書かれているのは、このプログラムの大きさが7語だということを示しています。その7語のうちの1語はRET命令に相当する機械語の命令で、残りの6語は、DS命令によって確保された領域です。

2.4.3 DS命令のラベル

DS命令でラベルを定義すると、そのラベルは、DS命令によって主記憶装置の上に確保された領域の先頭のアドレスに対して与えられます。

プログラムの例 dslabel.cas

```
DSLABEL START
        RET
AREA    DS    6
        DC    AREA
        END
```

このプログラムをアセンブルして、オブジェクトコードを見てみましょう。そうすると、次のように、RET命令に相当する機械語の命令が0番地に設定されていて、1番地というアドレスが7番地に設定されているはずです。

```
0000:8100
0007:0001R
```

つまり、DS命令によって定義されたAREAというラベルは、1番地というアドレスに与えられているわけです。この1番地というアドレスは、DS命令によって確保された領域の先頭のアドレスです。

2.4.4 DS命令によって確保された領域の内容

DS命令は、主記憶装置の上に領域を確保するわけですが、その領域に対して、いかなる変更も加えません。ですから、DS命令によって確保された領域に特定のデータ（たとえば0）が格納されているということを前提としてプログラムを書いた場合、そのプログラムは、期待したとおりに動くとは限りません。

第3章 ロードとストアとロードアドレス

3.1 ロードとストアとロードアドレスの基礎

3.1.1 この章について

この章では、ロード、ストア、ロードアドレスという3種類の動作について説明したいと思います。

そこで、まず最初にこの節では、ロード、ストア、ロードアドレスというのは、いったいどのような動作なのか、ということについて説明します。

3.1.2 ロード

ロードというのは、どこかにあるデータを汎用レジスタへ転送するという動作のことです。

どこからデータを転送するのか、つまりロードの転送元は、主記憶装置の語か、または汎用レジスタです。つまり、ロードというのは、次の2種類の動作の総称だと考えることができます。

- 主記憶装置の語の内容を汎用レジスタへ転送する。
- 汎用レジスタの内容を別の汎用レジスタへ転送する。

ロードは、転送元が主記憶装置の語の場合も、汎用レジスタの場合も、LD (LoaD) という命令を使うことによって実行することができます。この命令は、データの転送元と転送先、つまりどこからどこへデータを転送するのかということをオペランドとして記述する必要があります。

3.1.3 ストア

ストアというのは、汎用レジスタの内容を主記憶装置の語に転送するという動作のことです。

ロードとストアは、どちらも、ある場所にあるデータを別の場所へ転送する、という共通点を持っています。それらのあいだの相違点は、転送先です。ロードの転送先は汎用レジスタで、ストアの転送先は主記憶装置の語です。

ストアは、ST (STore) という命令を使うことによって実行することができます。この命令も、LD 命令と同じように、データの転送元と転送先、つまりどこからどこへデータを転送するのかということをオペランドとして記述する必要があります。

3.1.4 ロードアドレス

ロードアドレスというのは、特定のデータを汎用レジスタに設定するという動作のことです。

ロードアドレスは、LAD (Load ADdress) という命令を使うことによって実行することができます。この命令は、どの汎用レジスタにデータを設定するのかということと、そこに設定するデータそのものをオペランドとして記述する必要があります。

3.2 ダンプ

3.2.1 ダンプの基礎

レジスタの内容や主記憶装置の内容などを表示することを、それらを「ダンプする」(dump) と言います。

CASL II の実習をするとき、レジスタの内容や主記憶装置の内容をダンプすることができれば、プログラムが動作した結果がどうなったかということを確認することができますので、とても便利です。ところが、COMET II には、レジスタの内容や主記憶装置の内容をダンプするという動作をする命令は存在しません。

情報処理技術者試験センターが開発した CASL II シミュレータは、COMET II というコンピュータをシミュレートしているわけですが、それがシミュレートしているコンピュータと、COMET II とは、まったく同じものではありません。前者には、後者にはない命令が追加されています。前者に追加されている命令の中には、レジスタの内容をダンプするものと、主記憶装置の内容をダンプするものが含まれています。ですから、それらの命令を使うことによって、レジスタの内容や主記憶装置の内容を確認することができます。

この節では、DREG という命令と DMEM という命令を紹介します。これらの二つの命令は、CASL II には存在しないもので、それに対応する機械語の命令も、COMET II には存在しないもので

す。それらは、情報処理技術者試験センターが開発した CASL II シミュレータだけに存在する命令です。

3.2.2 DREG 命令

DREG (Dump REGister) という命令は、8 文字のメッセージを出力したのち、すべてのレジスタの内容をダンプします。

DREG 命令を使うためには、その準備として、出力する 8 文字のメッセージを作って、そのメッセージの先頭の文字が格納されている語のアドレスを示すラベルを定義しておく必要があります。

DREG 命令のオペランドは、メッセージの先頭を示すラベルです。

プログラムの例　dreg.cas

```
DREG     START
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムをアセンブルして実行すると、次のように、MESSAGE1 というメッセージとともに、すべてのレジスタの内容が表示されます。

```
* MESSAGE1 *    PR:0000 [ DREG ]  FR:0(OF:0 SF:0 ZF:0) SP:FFOO
GR0:0000 GR1:0000 GR2:0000 GR3:0000 GR4:0000 GR5:0000 GR6:0000 GR7:0000
```

3.2.3 DMEM 命令

DMEM (Dump MEMory) という命令は、8 文字のメッセージを出力したのち、主記憶装置の内容を、指定された範囲でダンプします。

DMEM 命令を使うためには、その準備として、DREG 命令の場合と同じように、出力する 8 文字のメッセージを作って、そのメッセージの先頭の文字が格納されている語のアドレスを示すラベルを定義しておく必要があります。そしてさらに、ダンプする主記憶装置の開始位置と終了位置を示すラベルも、定義しておく必要があります。

DMEM 命令のオペランドは、メッセージの先頭を示すラベル、開始位置を示すラベル、終了位置を示すラベルを、コンマで区切って並べたものです。

プログラムの例　dmem.cas

```
DMEM     START
         DMEM  MSG,DUMPS,DUMPE
         RET
MSG      DC    'MESSAGE1'
DUMPS    DC    'Hello, world!'
DUMPE    DC    -1
         END
```

このプログラムをアセンブルして実行すると、次のように、MESSAGE1 というメッセージとともに、主記憶装置の内容が DUMPS から DUMPE までの範囲で表示されます。

```
* MESSAGE1 *    Start:000D End:001A
 0008: ---- ---- ---- ---- ---- 0048 0065 006C       Hel
 0010: 006C 006F 002C 0020 0077 006F 0072 006C  lo, worl
 0018: 0064 0021 FFFF ---- ---- ---- ---- ----  d!.
```

3.3 LD 命令

3.3.1 LD 命令についての復習

第 3.1.2 項で説明したように、ロードは、LD (LoaD) という命令を使うことによって実行することができます。

3.3.2 LD 命令のオペランド

LD 命令のオペランドは、

どこへ , どこから

という構文で書きます。「どこへ」のところに書くのは、転送先の汎用レジスタの名前で、「どこから」のところに書くのは、転送元の語のアドレスに与えられたラベル、または転送元の汎用レジスタの名前です。

3.3.3 主記憶装置の語から汎用レジスタへのロード

主記憶装置の語から汎用レジスタへデータをロードしたいときは、オペランドとして、転送先の汎用レジスタの名前と、転送元の語のアドレスに与えられたラベルを書きます。たとえば、HOGE というラベルで示されるアドレスの語の内容を汎用レジスタの GR5 へ転送したいならば、

```
GR5,HOGE
```

というオペランドを書けばいいわけです。

プログラムの例 ldgs.cas

```
LDGS    START
        LD    GR5,HOGE
        DREG  MSG
        RET
HOGE    DC    #0F0F
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、HOGE 番地の語に格納されている #0F0F というデータを汎用レジスタの GR5 へロードして、そののち DREG 命令でレジスタの内容を表示します。ですから、GR5 の内容は、

```
GR5:0F0F
```

と表示されるはずです。

3.3.4 汎用レジスタから汎用レジスタへのロード

汎用レジスタから汎用レジスタへデータをロードしたいときは、オペランドとして、転送先の汎用レジスタの名前と転送元の汎用レジスタの名前を書きます。たとえば、汎用レジスタの GR5 の内容を汎用レジスタの GR4 へ転送したいならば、

```
GR4,GR5
```

というオペランドを書けばいいわけです。

プログラムの例 ldgg.cas

```
LDGG    START
        LD    GR5,HOGE
        LD    GR4,GR5
        DREG  MSG
        RET
HOGE    DC    #0A0A
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、HOGE 番地の語に格納されている #0A0A というデータを汎用レジスタの GR5 へ転送して、次に、 GR5 の内容を GR4 へ転送して、次に、 DREG 命令でレジスタの内容を表示します。ですから、 GR4 と GR5 の内容は、

```
GR4:0A0A GR5:0A0A
```

と表示されるはずです。

3.4 ST 命令

3.4.1 ST 命令についての復習

第 3.1.3 項で説明したように、ストアは、 ST (STore) という命令を使うことによって実行することができます。

3.4.2 ST 命令のオペランド

ST 命令のオペランドは、 LD 命令とは逆に、

どこから	,	どこへ

という構文で書きます。「どこから」のところに書くのは、転送元の汎用レジスタの名前で、「どこへ」のところに書くのは、転送先の語のアドレスに与えられたラベルです。たとえば、汎用レジスタの GR5 の内容を、 GEMO というラベルで示されるアドレスの語へ転送したいならば、

```
GR5,GEMO
```

というオペランドを書けばいいわけです。

プログラムの例　st.cas

```
ST      START
        LD    GR5,HOGE
        ST    GR5,GEMO
        DMEM  MSG,HOGE,GEMO
        RET
HOGE    DC    #0A0A
GEMO    DS    1
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、 HOGE 番地の語に格納されている #0A0A というデータを汎用レジスタの GR5 へロードして、次に、 GR5 の内容を GEMO 番地の語へストアして、次に、 HOGE 番地と GEMO 番地の語の内容を DMEM 命令で表示します。 HOGE 番地には最初から #0A0A が格納されていて、ロードとストアによって GEMO 番地にも #0A0A が格納されることになりますから、

```
0A0A 0A0A
```

と表示されるはずです。

3.5　LAD 命令

3.5.1　LAD 命令についての復習

第 3.1.4 項で説明したように、ロードアドレスは、 LAD (Load ADdress) という命令を使うことによって実行することができます。

3.5.2　LAD 命令のオペランド

LAD 命令のオペランドは、

どこに	,	何を

という構文で書きます。「どこに」のところに書くのは、データを設定する汎用レジスタの名前で、「何を」のところに書くのは、その汎用レジスタに設定するデータをあらわしている定数です。

LAD 命令のオペランドで、「何を」のところに書くことのできる定数は、第 2.3 節で説明した、 DC 命令のオペランドに書くことのできる 4 種類の定数のうちの、次の 3 種類です。

- 10 進定数
- 16 進定数
- アドレス定数

文字定数は、 LAD 命令では使うことができません。

プログラムの例　lad.cas

```
LAD     START
        LAD   GR3,-1
        LAD   GR4,#ABAB
        LAD   GR5,HOGE
        DREG  MSG
        RET
HOGE    DS    1
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、−1 というデータ (#FFFF) を汎用レジスタの GR3 に設定して、次に、 #ABAB というデータを汎用レジスタの GR4 に設定して、次に、 HOGE 番地というデータ（9 番地）を汎用レジスタの GR5 に設定して、次に、 DREG 命令でレジスタの内容を表示します。ですから、

```
GR3:FFFF GR4:ABAB GR5:0009
```

と表示されるはずです。

3.6 アドレス修飾

3.6.1 アドレス修飾の基礎

機械語の命令の多くは、主記憶装置の語を操作の対象とします。命令が操作の対象とする主記憶装置の語のアドレスは、その命令の「実効アドレス」(effective address) と呼ばれます。

実効アドレスは、「アドレス修飾」(address modification) と呼ばれる計算の結果として得られます。

アドレス修飾というのは、二つの整数の加算です。整数のひとつは、命令自体が持っているアドレスで、もうひとつは、「指標レジスタ」(index register) と呼ばれるレジスタの内容です。たとえば、命令自体が持っているアドレスが 3 番地で、指標レジスタの内容が 5 だとすると、実効アドレスは 8 番地ということになります。

3.6.2 COMET II の指標レジスタ

COMET II では、汎用レジスタが指標レジスタとして使われます。ただし、指標レジスタとして使うことができるのは、GR1 から GR7 までの汎用レジスタです。つまり、GR0 だけは、指標レジスタとして使うことができません。

3.6.3 CASL II でのアドレス修飾

主記憶装置の語を操作の対象とする CASL II の命令は、指標レジスタが何も指定されていない場合、オペランドに書かれたラベルによって示されるアドレスがそのまま実効アドレスとなります。たとえば、

```
    ST      GR3,HOGE
```

という ST 命令では、HOGE というラベルが示しているアドレスがそのまま実効アドレスです。

CASL II で、アドレス修飾をしたい場合、つまりラベルが示しているアドレスと指標レジスタの内容とを加算した結果を実効アドレスにしたい場合は、そのラベルの右側に、コンマと、指標レジスタとして使う汎用レジスタの名前を書きます。たとえば、

```
    ST    GR3,HOGE,GR5
```

という ST 命令では、HOGE というラベルが示しているアドレスに GR5 の内容を加算した結果が実効アドレスになります。

プログラムの例 modify.cas

```
MODIFY   START
         DMEM  MSG,HOGE,GEMO
         LAD   GR3,#DADA
         LAD   GR5,4
         ST    GR3,HOGE,GR5
         DMEM  MSG,HOGE,GEMO
         RET
HOGE     DC    'ABCDEFG'
GEMO     DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず最初に、HOGE 番地から始まる 8 語の領域を表示します。その領域には、最初は、

```
    0041 0042 0043 0044 0045 0046 0047 FFFF
```

というように、A から G までの文字と −1 が格納されていますが、その後、ST 命令が、#DADA というデータをその領域にストアします。その ST 命令の実効アドレスは、HOGE というラベルが示しているアドレスに、GR5 の内容を加算した結果です。GR5 には 4 が格納されていますので、

```
    0041 0042 0043 0044 DADA 0046 0047 FFFF
```

というように、それまで E という文字が格納されていた語に、#DADA がストアされることになります。

3.6.4 LAD 命令のアドレス修飾

LAD 命令でも、オペランドで指標レジスタを指定することができます。

LAD 命令のオペランドを、

汎用レジスタ , 定数 , 指標レジスタ

というように書くと、定数があらわしている整数と指標レジスタの内容とが加算されて、その結果として得られた整数が汎用レジスタに設定されることになります。

プログラムの例 ladmod.cas

```
LADMOD   START
         LAD   GR5,4
         LAD   GR3,2,GR5
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムの二つ目の LAD 命令は、定数があらわしている 2 という整数と、GR5 に格納されている 4 という整数とを加算した結果、つまり 6 という整数を GR3 に設定します。

3.6.5　インクリメントとデクリメント

LAD 命令のオペランドに書く指標レジスタは、データを設定する対象となる汎用レジスタと同じものでも構いません。たとえば、

```
    LAD   GR3,2,GR3
```

というように、2 と GR3 の内容とを加算した結果を GR3 に設定する、ということが可能です。この場合、GR3 の内容は、それ以前よりも 2 だけ大きくなります。

レジスタに格納されている整数を 1 だけ大きくすることを、レジスタを「インクリメントする」(increment) すると言い、レジスタに格納されている整数を 1 だけ小さくすることを、レジスタを「デクリメントする」(decrement) すると言います。

インクリメントとデクリメントは、LAD 命令を使うことによって実行することができます。たとえば、

```
    LAD   GR3,1,GR3
```

という LAD 命令は GR3 をインクリメントして、

```
    LAD   GR3,-1,GR3
```

という LAD 命令は GR3 をデクリメントします。

プログラムの例 incdec.cas

```
INCDEC   START
         LAD   GR3,4
         DREG  MSG
         LAD   GR3,1,GR3
         DREG  MSG
         LAD   GR3,-1,GR3
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず、GR3 に 4 を設定して、次に GR3 をインクリメントして、次に GR3 をデクリメントします。ですから、GR3 の内容は、4、5、4 と変化します。

第 4 章　加算と減算

4.1　加算と減算の基礎

4.1.1　この章について

この章では、加算をする命令と減算をする命令について説明したいと思います。

そこで、まず最初にこの節では、加算と減算の命令について理解するための予備知識について説明します。

4.1.2 演算命令

CASL II の機械語命令のうちの 13 種類は、何らかの演算を実行する命令です。それらの命令は、総称して「演算命令」と呼ばれます。この章で説明する加算と減算の命令も、演算命令に分類されます。

4.1.3 算術演算命令と論理演算命令

13 種類の演算命令のうちの 10 種類は、2 種類で一組になっています。つまり、五つの組が存在するということです。

それらの五つの組のそれぞれは、「算術演算命令」と呼ばれる 5 種類の命令のうちの一つと、「論理演算命令」と呼ばれる 5 種類の命令のうちの一つから構成されています。

算術演算命令と論理演算命令とのあいだにある相違点は、整数の範囲です。どちらの命令も、処理の対象を整数とみなすのですが、算術演算命令と論理演算命令とでは、処理の対象とする整数の範囲が違います。

算術演算命令が処理の対象とする整数の範囲は、プラスとマイナスの両方にまたがっています。それに対して、論理演算命令が処理の対象とする整数の範囲は、ゼロまたはプラスです。

もう少し厳密に言うと、算術演算命令が処理の対象とする整数は、-2^{15} から $2^{15}-1$ まで、つまり -32768 から 32767 までで、論理演算命令が処理の対象とする整数は、0 から $2^{16}-1$ まで、つまり 0 から 65535 までです。算術演算命令が処理の対象とする整数は「符号付き整数」と呼ばれ、論理演算命令が処理の対象とする整数は「符号なし整数」と呼ばれます。

算術演算命令は、最上位ビット（つまり左端の 15 番のビット）を整数の符号とみなします。0 ならばプラス、1 ならばマイナスです。たとえば、

```
1111 1111 1111 1111
```

というビット列は、左端の最上位ビットが 1 ですので、算術演算命令はそれをマイナスの整数とみなして -1 と解釈します。

それに対して、論理演算命令は、最上位ビットを単に 2^{15} の桁とみなします。たとえば、

```
1111 1111 1111 1111
```

というビット列を、論理演算命令は 65535 というプラスの整数と解釈します。

4.1.4 演算オペランド形式

演算命令は 13 種類あるわけですが、それらのうちで、「シフト演算命令」と呼ばれる 4 種類を除いた 9 種類は、オペランドの形式がまったく同じです。このチュートリアルでは、それらの 9 種類の命令で使われるオペランドの形式のことを、「演算オペランド形式」と呼ぶことにしたいと思います。

演算オペランド形式には、2 通りの書き方があります。

第一の書き方は、

汎用レジスタ名 , ラベル , 指標レジスタ名

という形です。この形のオペランドを書いた場合は、汎用レジスタ名で指定された汎用レジスタの内容と、実効アドレスで指定された主記憶装置の語の内容が、演算の対象となります。そして、演算の結果は、汎用レジスタ名で指定された汎用レジスタに設定されます。たとえば、

```
GR3,HOGE,GR5
```

というオペランドを持つ、シフト演算命令以外の演算命令は、汎用レジスタの GR3 の内容と、HOGE 番地と GR5 番地を加算することによって得られた実効アドレスの語の内容に対して演算を実行して、その結果を GR3 に設定します。

ラベルの右側のコンマと指標レジスタ名は省略してもかまいませんので、

汎用レジスタ名 , ラベル

という形の書き方も可能です。この形のオペランドを書いた場合は、ラベルが与えられているアドレスが、そのまま実効アドレスになります。たとえば、

```
GR3,HOGE
```

というオペランドを持つ、シフト演算命令以外の演算命令は、汎用レジスタの GR3 の内容と、HOGE 番地の語の内容に対して演算を実行して、その結果を GR3 に設定します。

第二の書き方は、

汎用レジスタ名$_1$, 汎用レジスタ名$_2$

という形の書き方です。この形のオペランドを書いた場合は、汎用レジスタ名$_1$で指定された汎用レジスタの内容と、汎用レジスタ名$_2$で指定された汎用レジスタの内容が、演算の対象となります。そして、演算の結果は、汎用レジスタ名$_1$で指定された汎用レジスタに設定されます。たとえば、

```
GR3,GR4
```

というオペランドを持つ、シフト演算命令以外の演算命令は、汎用レジスタの GR3 の内容と GR4 の内容に対して演算を実行して、その結果を GR3 に設定します。

4.1.5　加算と減算の命令

CASL II には、加算と減算の命令として、次の 4 種類のものがあります。

ADDA	ADD Arithmetic	算術加算命令
ADDL	ADD Logical	論理加算命令
SUBA	SUBtract Arithmetic	算術減算命令
SUBL	SUBtract Logical	論理減算命令

名前から分かるとおり、 ADDA と SUBA は算術演算命令、 ADDL と SUBL は論理演算命令です。

4.2　フラグレジスタ

4.2.1　フラグレジスタについての復習

この節では、フラグレジスタについて説明します。

フラグレジスタについては、第 1.3.8 項で簡単に説明しましたので、まず、そこで説明したことを復習しておくことにしましょう。

COMET II は、「フラグレジスタ」と呼ばれる 1 個のレジスタ（英語では flag register、略称は FR）を持っています。

フラグレジスタは、命令を実行した結果に関する状態を保持するレジスタで、長さは 3 ビットです。

4.2.2　フラグレジスタの構成

フラグレジスタを構成している 3 個のビットのそれぞれは、「フラグ」(flag) と呼ばれます。

「フラグ」という言葉は、「旗」という意味です。このネーミングは、それが持っている 0 または 1 という二つの状態を、「旗が寝ている状態」と「旗が立っている状態」になぞらえたものです。

フラグレジスタを構成している 3 本のフラグは、左から右へ（上位から下位へ）順番に、次のような名前と意味を持っています。

オーバーフローフラグ	英語では overflow flag、略称は OF。演算の結果が、表現が可能な範囲の外に出ることを「オーバーフロー」(overflow) と言う。演算の結果が、算術演算命令の場合は −32768 から 32767 までの範囲、論理演算命令の場合は 0 から 65535 までの範囲から外に出ると、オーバーフローが発生する。このフラグには、オーバーフローが発生したならば 1、発生しなかったならば 0 が設定される。
サインフラグ	英語では sign flag、略称は SF。「サイン」(sign) というのは符号のこと。このフラグには、演算の結果の最上位ビット（つまり左端の 15 番のビット）が 1 ならば 1、0 ならば 0 が設定される。算術演算命令の場合だけではなく、論理演算命令の場合も、演算の結果の最上位ビットが 1 ならば 1 が設定される。
ゼロフラグ	英語では zero flag、略称は ZF。このフラグには、演算の結果が 0 ならば 1、それ以外ならば 0 が設定される。

4.2.3 フラグレジスタを変化させる命令

CASL II のすべての機械語命令が、自分の動作の結果に関する状態をフラグレジスタに設定するわけではありません。フラグレジスタを変化させる可能性があるのは、演算命令と LD 命令の 14 種類だけです。それら以外の命令は、フラグレジスタの状態を変化させず、それ以前の状態を維持します。たとえば、ST 命令を実行したのちのフラグレジスタの状態は、常に、その命令を実行する以前の状態と同じです。

それでは、LD 命令を使って、フラグレジスタの変化を確かめてみましょう。

プログラムの例 ldsf.cas

```
LDSF    START
        LD    GR5,HOGE
        DREG  MSG
        RET
HOGE    DC    #8000
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、HOGE 番地の語に格納されている #8000 という整数を汎用レジスタの GR5 へロードして、そののち DREG 命令でレジスタの内容を表示します。ロードした整数の最上位ビットは 1 ですから、フラグレジスタのサインフラグには、

```
FR:2(OF:0 SF:1 ZF:0)
```

というように、1 が設定されているはずです。

プログラムの例 ldzf.cas

```
LDZF    START
        LD    GR5,HOGE
        DREG  MSG
        RET
HOGE    DC    0
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、HOGE 番地の語に格納されている 0 という整数を汎用レジスタの GR5 へロードして、そののち DREG 命令でレジスタの内容を表示します。ロードしたのは 0 ですから、フラグレジスタのゼロフラグには、

```
FR:1(OF:0 SF:0 ZF:1)
```

というように、1 が設定されているはずです。

なお、LD 命令は、オーバーフローを発生させることのない命令ですので、オーバーフローフラグには常に 0 を設定します。

4.3 ADDA 命令

4.3.1 ADDA 命令の基礎

「算術加算命令」と呼ばれる ADDA (ADD Arithmetic) は、汎用レジスタの内容と主記憶装置の語の内容、または二つの汎用レジスタの内容を、符号付き整数とみなして加算して、その結果を汎用レジスタに設定する命令です。

二つのビット列を符号付き整数とみなして加算することを、「算術加算」と呼ぶことにします。

4.3.2 汎用レジスタと主記憶装置の語との算術加算

ADDA 命令のオペランドとして、汎用レジスタ名とラベルを書くと、名前で指定された汎用レジスタの内容と、実効アドレスで指定された主記憶装置の語の内容とを算術加算して、その結果を、名前で指定された汎用レジスタに設定する、という意味になります。たとえば、

```
ADDA  GR5,HOGE
```

という命令は、汎用レジスタの GR5 の内容と、HOGE 番地の語の内容とを算術加算して、その結果を GR5 に設定する、という意味になります。

プログラムの例 addags.cas

```
ADDAGS  START
        LAD   GR5,-1
```

```
        ADDA   GR5,HOGE
        DREG   MSG
        RET
HOGE    DC     3
MSG     DC     'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR5 に −1 という整数を設定して、次に GR5 の内容と HOGE 番地の語の内容とを算術加算して、そののち DREG 命令でレジスタの内容を表示します。HOGE 番地には 3 が格納されていますので、演算の結果は 2 になります。したがって、GR5 の内容は、

```
GR5:0002
```

と表示されるはずです。

4.3.3　汎用レジスタと汎用レジスタとの算術加算

ADDA 命令のオペランドとして、二つの汎用レジスタの名前を書くと、名前で指定された二つの汎用レジスタの内容を算術加算して、その結果を、左に書いた名前で指定された汎用レジスタに設定する、という意味になります。たとえば、

```
ADDA  GR3,GR4
```

という命令は、汎用レジスタの GR3 の内容と GR4 の内容とを算術加算して、その結果を GR3 に設定する、という意味になります。

プログラムの例　addagg.cas

```
ADDAGG  START
        LAD    GR3,10
        LAD    GR4,-2
        ADDA   GR3,GR4
        DREG   MSG
        RET
MSG     DC     'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR3 に 10、GR4 に −2 という整数を設定して、次に GR3 の内容と GR4 の内容とを算術加算して、そののち DREG 命令でレジスタの内容を表示します。GR3 の内容は、

```
GR3:0008
```

と表示されるはずです。

4.3.4　ADDA 命令によるフラグレジスタの変化

ADDA 命令はフラグレジスタをどのように変化させるかということを確認してみましょう。

プログラムの例　addaof.cas

```
ADDAOF  START
        LAD    GR3,32767
        LAD    GR4,1
        ADDA   GR3,GR4
        DREG   MSG
        RET
MSG     DC     'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR3 に 32767、GR4 に 1 という整数を設定して、次に GR3 の内容と GR4 の内容とを算術加算して、そののち DREG 命令でレジスタの内容を表示します。32767 に 1 を算術加算するとオーバーフローが発生しますので、フラグレジスタのオーバーフローフラグには、

```
FR:6(OF:1 SF:1 ZF:0)
```

というように、1 が設定されているはずです。また、演算の結果の最上位ビットが 1 になりますので、サインフラグにも 1 が設定されます。

プログラムの例　addasf.cas

```
ADDASF   START
         LAD   GR3,2
         LAD   GR4,-3
         ADDA  GR3,GR4
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR3 に 2、GR4 に −3 という整数を設定して、次に GR3 の内容と GR4 の内容とを算術加算して、そののち DREG 命令でレジスタの内容を表示します。演算の結果は −1 で、その最上位ビットは 1 ですので、フラグレジスタのサインフラグには、

```
FR:2(OF:0 SF:1 ZF:0)
```

というように、1 が設定されているはずです。

プログラムの例 addazf.cas

```
ADDAZF   START
         LAD   GR3,3
         LAD   GR4,-3
         ADDA  GR3,GR4
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR3 に 3、GR4 に −3 という整数を設定して、次に GR3 の内容と GR4 の内容とを算術加算して、そののち DREG 命令でレジスタの内容を表示します。演算の結果は 0 ですので、フラグレジスタのゼロフラグには、

```
FR:1(OF:0 SF:0 ZF:1)
```

というように、1 が設定されているはずです。

4.4 ADDL 命令

4.4.1 ADDL 命令の基礎

「論理加算命令」と呼ばれる ADDL (ADD Logical) は、汎用レジスタの内容と主記憶装置の語の内容、または二つの汎用レジスタの内容を、符号なし整数とみなして加算して、その結果を汎用レジスタに設定する命令です。

二つのビット列を符号なし整数とみなして加算することを、「論理加算」と呼ぶことにします。

4.4.2 汎用レジスタと主記憶装置の語との論理加算

ADDL 命令のオペランドとして、汎用レジスタ名とラベルを書くと、名前で指定された汎用レジスタの内容と、実効アドレスで指定された主記憶装置の語の内容とを論理加算して、その結果を、名前で指定された汎用レジスタに設定する、という意味になります。たとえば、

```
ADDL  GR5,HOGE
```

という命令は、汎用レジスタの GR5 の内容と、HOGE 番地の語の内容とを論理加算して、その結果を GR5 に設定する、という意味になります。

プログラムの例 addlgs.cas

```
ADDLGS   START
         LAD   GR5,4
         ADDL  GR5,HOGE
         DREG  MSG
         RET
HOGE     DC    3
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR5 に 4 という整数を設定して、次に、GR5 の内容と HOGE 番地の語の内容とを論理加算して、そののち DREG 命令でレジスタの内容を表示します。HOGE 番地には 3 が格納されていますので、加算の結果は 7 になります。したがって、GR5 の内容は、

GR5:0007

と表示されるはずです。

4.4.3　汎用レジスタと汎用レジスタとの論理加算

ADDL 命令のオペランドとして、二つの汎用レジスタの名前を書くと、名前で指定された二つの汎用レジスタの内容を論理加算して、その結果を、左に書いた名前で指定された汎用レジスタに設定する、という意味になります。たとえば、

```
ADDL  GR3,GR4
```

という命令は、汎用レジスタの GR3 の内容と GR4 の内容とを論理加算して、その結果を GR3 に設定する、という意味になります。

プログラムの例　addlgg.cas

```
ADDLGG   START
         LAD   GR3,32767
         LAD   GR4,1
         ADDL  GR3,GR4
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR3 に 32767、GR4 に 1 を設定して、次に、GR3 の内容と GR4 の内容とを論理加算して、そののち DREG 命令でレジスタの内容を表示します。32767 に 1 を論理加算すると、結果は 32768（16 進数で書くと #8000 ）になりますので、GR3 の内容は、

GR3:8000

と表示されるはずです。

4.4.4　ADDL 命令によるフラグレジスタの変化

ADDL 命令を実行したとき、フラグレジスタの内容はどのように変化するのでしょうか。

まず、ADDL 命令を使った上の二つのプログラムのそれぞれを実行したときに、フラグレジスタの内容がどのように表示されるか、ということを確認しておきましょう。

addlgs.cas の論理加算は、オーバーフローを発生させず、最上位ビットは 0 で、結果は 0 ではありませんので、フラグレジスタには、

FR:0(OF:0 SF:0 ZF:0)

と設定されるはずです。

addlgg.cas を実行すると、フラグレジスタは、

FR:2(OF:0 SF:1 ZF:0)

と表示されるはずです。

算術加算の場合だと、32767 に 1 を加算するとオーバーフローが発生しますので、オーバーフローフラグに 1 が設定されることになります。それに対して、論理加算の場合は、32767 に 1 を加算してもオーバーフローは発生しませんので、オーバーフローフラグには 0 が設定されます。

サインフラグに 1 が設定されているのを見て、不審に思った人がいるかもしれません。「サイン」というのは符号という意味ですから、不審に思うのも無理はありません。実は、COMET II は、論理演算命令を実行した場合も、その結果を符号付き整数とみなして、その符号をサインフラグに設定するのです。言い換えれば、算術演算命令の場合も論理演算命令の場合も、結果の最上位ビットが 1 ならばサインフラグには 1 が設定されるということです。したがって、論理演算命令の結果が #8000 から #FFFF まで（10 進数で書くと 32768 から 65535 まで）の範囲内だった場合は、最上位ビットが 1 ですので、サインフラグには 1 が設定されることになります。

次に、論理加算でオーバーフローが発生するプログラムを書いて、実行してみましょう。

プログラムの例　addlof.cas

```
ADDLOF   START
         LAD   GR3,65535
         LAD   GR4,1
         ADDL  GR3,GR4
         DREG  MSG
         RET
```

```
MSG       DC    'MESSAGE1'
          END
```

このプログラムは、まず汎用レジスタの GR3 に 65535、GR4 に 1 という整数を設定して、次に GR3 の内容と GR4 の内容とを論理加算して、そののち DREG 命令でレジスタの内容を表示します。65535 に 1 を論理加算するとオーバーフローが発生しますので、フラグレジスタのオーバーフローフラグには、

```
FR:5(OF:1 SF:0 ZF:1)
```

というように、1 が設定されているはずです。また、演算の結果が 0 になりますので、ゼロフラグにも 1 が設定されます。

4.5 SUBA 命令

4.5.1 SUBA 命令の基礎

「算術減算命令」と呼ばれる SUBA (SUB Arithmetic) は、汎用レジスタの内容から、主記憶装置の語の内容、または別の汎用レジスタの内容を、それらを符号付き整数とみなして減算して、その結果を汎用レジスタに設定する命令です。

ひとつのビット列から別のビット列を、それらを符号付き整数とみなして減算することを、「算術減算」と呼ぶことにします。

4.5.2 汎用レジスタからの主記憶装置の語の算術減算

SUBA 命令のオペランドとして、汎用レジスタ名とラベルを書くと、名前で指定された汎用レジスタの内容から、実効アドレスで指定された主記憶装置の語の内容を算術減算して、その結果を、名前で指定された汎用レジスタに設定する、という意味になります。たとえば、

```
SUBA  GR5,HOGE
```

という命令は、汎用レジスタの GR5 の内容から、HOGE 番地の語の内容とを算術減算して、その結果を GR5 に設定する、という意味になります。

プログラムの例 subags.cas

```
SUBAGS    START
          LAD   GR5,7
          SUBA  GR5,HOGE
          DREG  MSG
          RET
HOGE      DC    3
MSG       DC    'MESSAGE1'
          END
```

このプログラムは、まず汎用レジスタの GR5 に 7 という整数を設定して、次に GR5 の内容から HOGE 番地の語の内容を算術減算して、そののち DREG 命令でレジスタの内容を表示します。HOGE 番地には 3 が格納されていますので、演算の結果は 4 になります。したがって、GR5 の内容は、

```
GR5:0004
```

と表示されるはずです。

4.5.3 汎用レジスタからの汎用レジスタの算術減算

SUBA 命令のオペランドとして、二つの汎用レジスタの名前を書くと、左に書いた名前で指定された汎用レジスタの内容から、右に書いた名前で指定された汎用レジスタの内容を算術減算して、その結果を、左に書いた名前で指定された汎用レジスタに設定する、という意味になります。たとえば、

```
SUBA  GR3,GR4
```

という命令は、汎用レジスタの GR3 の内容から GR4 の内容を算術減算して、その結果を GR3 に設定する、という意味になります。

プログラムの例 subagg.cas

```
SUBAGG    START
```

```
        LAD   GR3,10
        LAD   GR4,2
        SUBA  GR3,GR4
        DREG  MSG
        RET
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR3 に 10、GR4 に 2 という整数を設定して、次に GR3 の内容から GR4 の内容を算術減算して、そののち DREG 命令でレジスタの内容を表示します。GR3 の内容は、

```
GR3:0008
```

と表示されるはずです。

4.5.4　SUBA 命令によるフラグレジスタの変化

SUBA 命令はフラグレジスタをどのように変化させるかということを確認してみましょう。

プログラムの例　subaof.cas

```
SUBAOF  START
        LAD   GR3,-32768
        LAD   GR4,1
        SUBA  GR3,GR4
        DREG  MSG
        RET
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR3 に −32768、GR4 に 1 という整数を設定して、次に GR3 の内容から GR4 の内容を算術減算して、そののち DREG 命令でレジスタの内容を表示します。−32768 から 1 を算術減算するとオーバーフローが発生しますので、フラグレジスタのオーバーフローフラグには、

```
FR:4(OF:1 SF:0 ZF:0)
```

というように、1 が設定されているはずです。また、演算の結果は #7FFF（10 進数で書くと 32767）で、最上位ビットが 0 ですので、サインフラグには 0 が設定されます。

プログラムの例　subasf.cas

```
SUBASF  START
        LAD   GR3,2
        LAD   GR4,3
        SUBA  GR3,GR4
        DREG  MSG
        RET
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR3 に 2、GR4 に 3 という整数を設定して、次に GR3 の内容から GR4 の内容を算術減算して、そののち DREG 命令でレジスタの内容を表示します。演算の結果は −1 で、その最上位ビットは 1 ですので、フラグレジスタのサインフラグには、

```
FR:2(OF:0 SF:1 ZF:0)
```

というように、1 が設定されているはずです。

プログラムの例　subazf.cas

```
SUBAZF  START
        LAD   GR3,3
        LAD   GR4,3
        SUBA  GR3,GR4
        DREG  MSG
        RET
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR3 と GR4 のそれぞれに 3 という整数を設定して、次に GR3 の内容から GR4 の内容を算術減算して、そののち DREG 命令でレジスタの内容を表示します。演算の結果は 0 ですので、フラグレジスタのゼロフラグには、

```
FR:1(OF:0 SF:0 ZF:1)
```

というように、1 が設定されているはずです。

4.6 SUBL 命令

4.6.1 SUBL 命令の基礎

「論理減算命令」と呼ばれる SUBL (SUB Logical) は、汎用レジスタの内容から、主記憶装置の語の内容、または別の汎用レジスタの内容を、それらを符号なし整数とみなして減算して、その結果を汎用レジスタに設定する命令です。

ひとつのビット列から別のビット列を、それらを符号なし整数とみなして減算することを、「論理減算」と呼ぶことにします。

4.6.2 汎用レジスタからの主記憶装置の語の論理減算

SUBL 命令のオペランドとして、汎用レジスタ名とラベルを書くと、名前で指定された汎用レジスタの内容から、実効アドレスで指定された主記憶装置の語の内容を論理減算して、その結果を、名前で指定された汎用レジスタに設定する、という意味になります。たとえば、

```
SUBL  GR5,HOGE
```

という命令は、汎用レジスタの GR5 の内容から、 HOGE 番地の語の内容とを論理減算して、その結果を GR5 に設定する、という意味になります。

プログラムの例 sublgs.cas

```
SUBLGS   START
         LAD   GR5,7
         SUBL  GR5,HOGE
         DREG  MSG
         RET
HOGE     DC    3
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR5 に 7 という整数を設定して、次に、 GR5 の内容から HOGE 番地の語の内容を論理減算して、そののち DREG 命令でレジスタの内容を表示します。HOGE 番地には 3 が格納されていますので、減算の結果は 4 になります。したがって、 GR5 の内容は、

```
GR5:0004
```

と表示されるはずです。

4.6.3 汎用レジスタからの汎用レジスタの論理減算

SUBL 命令のオペランドとして、二つの汎用レジスタの名前を書くと、左に書いた名前で指定された汎用レジスタの内容から、右に書いた名前で指定された汎用レジスタの内容を論理減算して、その結果を、左に書いた名前で指定された汎用レジスタに設定する、という意味になります。たとえば、

```
SUBL  GR3,GR4
```

という命令は、汎用レジスタの GR3 の内容から GR4 の内容を論理減算して、その結果を GR3 に設定する、という意味になります。

プログラムの例 sublgg.cas

```
SUBLGG   START
         LAD   GR3,10
         LAD   GR4,2
         SUBL  GR3,GR4
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR3 に 10、 GR4 に 2 を設定して、次に、 GR3 の内容から GR4 の内容を論理減算して、そののち DREG 命令でレジスタの内容を表示します。 GR3 の内

容は、

```
GR3:0008
```

と表示されるはずです。

4.6.4　SUBL 命令によるフラグレジスタの変化

SUBL 命令を実行したとき、フラグレジスタの内容はどのように変化するのでしょうか。サインフラグとゼロフラグに関しては、これまでに紹介した演算命令と同じですので、注意する必要があるのはオーバーフローフラグだけです。

それでは、論理減算でオーバーフローが発生するプログラムを書いて、実行してみましょう。

プログラムの例　sublof.cas

```
SUBLOF    START
          LAD    GR3,0
          LAD    GR4,1
          SUBL   GR3,GR4
          DREG   MSG
          RET
MSG       DC     'MESSAGE1'
          END
```

このプログラムは、まず汎用レジスタの GR3 に 0、GR4 に 1 という整数を設定して、次に GR3 の内容から GR4 の内容を論理減算して、そののち DREG 命令でレジスタの内容を表示します。0 から 1 を論理減算するとオーバーフローが発生しますので、フラグレジスタのオーバーフローフラグには、

```
FR:6(OF:1 SF:1 ZF:0)
```

というように、1 が設定されているはずです。また、演算の結果が #FFFF になりますので、サインフラグにも 1 が設定されます。

第 5 章　比較演算

5.1　比較演算の基礎

5.1.1　比較演算とは何か

この章では、「比較演算」と呼ばれる演算を実行する命令について説明したいと思います。

比較演算というのは、二つのビット列を比較して、それらのあいだの大小関係を調べる、という演算のことです。つまり、二つのビット列のうちのどちらが大きい数値を表現しているか、ということを調べる演算です。

CASL II が持っている 13 種類の演算命令のうちの 2 種類は、比較演算を実行する命令です。それらの 2 種類の命令は、「比較演算命令」と呼ばれます。

比較演算命令は、汎用レジスタの内容と主記憶装置の語の内容、または二つの汎用レジスタの内容に対して比較演算を実行して、フラグレジスタのサインフラグとゼロフラグにその結果を設定します。オーバーフローフラグには常に 0 を設定します。汎用レジスタと主記憶装置の語には、いかなる変化も与えません。

5.1.2　比較演算命令

CASL II には、比較演算命令として、次の 2 種類のものがあります。

CPA	ComPare Arithmetic	算術比較命令
CPL	ComPare Logical	論理比較命令

名前から分かるとおり、CPA は算術演算命令、CPL は論理演算命令です。

5.1.3　比較演算命令のオペランド

比較演算命令のオペランドの形式は、第 4.1.4 項で説明した、演算オペランド形式です。

5.1.4 比較演算命令によるフラグレジスタの設定

比較演算命令は、フラグレジスタのオーバーフローフラグに、常に 0 を設定します。そして、サインフラグとゼロフラグには、比較の結果を次のように設定します。

	SF	ZF
オペランドの左側 > オペランドの右側	0	0
オペランドの左側 = オペランドの右側	0	1
オペランドの左側 < オペランドの右側	1	0

この表は、「比較演算というのは、汎用レジスタを変化させず、オーバーフローを発生させない減算のことである」と考えると覚えやすいと思います。比較演算命令は、オペランドの左側に書かれたものから右側に書かれたものを、オーバーフローを気にしないで減算したときに、その結果がプラスになるのかゼロになるのかマイナスになるのか、ということをサインフラグとゼロフラグに設定するのです。

5.2 CPA 命令

5.2.1 CPA 命令の基礎

「算術比較命令」と呼ばれる CPA (ComPare Arithmetic) は、汎用レジスタの内容と主記憶装置の語の内容、または二つの汎用レジスタの内容を、符号付き整数とみなして比較して、その結果をサインフラグとゼロフラグに設定する命令です。

二つのビット列を符号付き整数とみなして比較することを、「算術比較」と呼ぶことにします。

5.2.2 算術比較によるフラグレジスタの設定

それでは、算術比較の結果はどのようにフラグレジスタに設定されるのか、プログラムを実行して確かめてみましょう。

プログラムの例 cpa.cas

```
CPA     START
        LAD   GR1,5
        CPA   GR1,A
        DREG  MSG
        CPA   GR1,B
        DREG  MSG
        CPA   GR1,C
        DREG  MSG
        RET
A       DC    4
B       DC    5
C       DC    6
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず、左が 5 で右が 4 というオペランドで CPA 命令を実行します。左から右を減算した結果はプラスですので、

```
FR:0(OF:0 SF:0 ZF:0)
```

というように、サインフラグには 0、ゼロフラグにも 0 が設定されます。

次に、左が 5 で右も 5 というオペランドで CPA 命令を実行します。左から右を減算した結果はゼロですので、

```
FR:1(OF:0 SF:0 ZF:1)
```

というように、サインフラグには 0、ゼロフラグには 1 が設定されます。

次に、左が 5 で右が 6 というオペランドで CPA 命令を実行します。左から右を減算した結果はマイナスですので、

```
FR:2(OF:0 SF:1 ZF:0)
```

というように、サインフラグには 1、ゼロフラグには 0 が設定されます。

5.2.3　マイナスの整数を対象とする算術比較

先ほどのプログラムは、プラスの整数とプラスの整数とを算術比較するものでしたが、算術比較は、マイナスの整数についても正しく比較することができますので、その点を実際のプログラムで確かめてみましょう。

プログラムの例　cpa2.cas

```
CPA2     START
         LAD   GR1,-5
         CPA   GR1,A
         DREG  MSG
         CPA   GR1,B
         DREG  MSG
         LAD   GR1,-32768
         CPA   GR1,C
         DREG  MSG
         RET
A        DC    -6
B        DC    5
C        DC    32767
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず、左が −5 で右が −6 というオペランドで CPA 命令を実行します。左から右を減算した結果はプラスですので、

```
FR:0(OF:0 SF:0 ZF:0)
```

というように、サインフラグには 0、ゼロフラグにも 0 が設定されます。

次に、左が −5 で右が 5 というオペランドで CPA 命令を実行します。左から右を減算した結果はマイナスですので、

```
FR:2(OF:0 SF:1 ZF:0)
```

というように、サインフラグには 1、ゼロフラグには 0 が設定されます。

次に、左が −32768 で右が 32767 というオペランドで CPA 命令を実行します。左から右を減算した結果はマイナスですので、

```
FR:2(OF:0 SF:1 ZF:0)
```

というように、サインフラグには 1、ゼロフラグには 0 が設定されます。−32768 から 32767 を減算すると、SUBA 命令の場合はオーバーフローが発生しますが、CPA 命令の場合にはオーバーフローは発生せず、比較の結果も正しく設定されます。

5.3　CPL 命令

5.3.1　CPL 命令の基礎

「論理比較命令」と呼ばれる CPL (ComPare Logical) は、汎用レジスタの内容と主記憶装置の語の内容、または二つの汎用レジスタの内容を、符号なし整数とみなして比較して、その結果をサインフラグとゼロフラグに設定する命令です。

二つのビット列を符号なし整数とみなして比較することを、「論理比較」と呼ぶことにします。

5.3.2　論理比較によるフラグレジスタの設定

それでは、論理比較の結果はフラグレジスタにどのように設定されるのか、プログラムを実行して確かめてみましょう。

プログラムの例　cpl.cas

```
CPL      START
         LAD   GR1,5
         CPL   GR1,A
         DREG  MSG
         CPL   GR1,B
         DREG  MSG
         CPL   GR1,C
         DREG  MSG
         RET
A        DC    4
B        DC    5
```

```
C       DC    6
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、左が 5 で右が 4、左が 5 で右も 5、左が 5 で右が 6、というそれぞれのオペランドで CPL 命令を実行します。このプログラムの中で CPL 命令がサインフラグとゼロフラグに設定するものは、次の表のとおりです。

	SF	ZF
左が 5 で右が 4	0	0
左が 5 で右も 5	0	1
左が 5 で右が 6	1	0

このプログラムの中で比較の対象となっているすべての整数は、最上位ビットが 0 です。このように、最上位ビットが 0 の整数どうしを比較する場合、CPA 命令と CPL 命令は、同じ動作をすることになります。

5.3.3 最上位ビットが 1 の整数を対象とする論理比較

最上位ビットが 1 の整数を、CPA 命令がマイナスの整数とみなして比較するのに対して、CPL 命令は、それをプラスの整数とみなして比較します。

それでは、実際のプログラムで確かめてみましょう。

プログラムの例 cpl2.cas

```
CPL2    START
        LAD   GR1,#FFFB
        CPL   GR1,A
        DREG  MSG
        CPL   GR1,B
        DREG  MSG
        RET
A       DC    #FFFA
B       DC    #0005
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず、左が #FFFB で右が #FFFA というオペランドで CPL 命令を実行します。左から右を減算した結果はプラスですので、

```
FR:0(OF:0 SF:0 ZF:0)
```

というように、サインフラグには 0、ゼロフラグにも 0 が設定されます。ちなみに、#FFFB は符号付き整数とみなすと −5、#FFFA は符号付き整数とみなすと −6 です。このように、最上位ビットが 1 の整数どうしを比較する場合も、最上位ビットが 0 の整数どうしを比較する場合と同様に、CPA の結果と CPL の結果は同じになります。

次に、左が #FFFB で右が #0005 というオペランドで CPL 命令を実行します。左から右を減算した結果はプラスですので、

```
FR:0(OF:0 SF:0 ZF:0)
```

というように、サインフラグには 0、ゼロフラグにも 0 が設定されます。#FFFB は符号付き整数とみなすと −5 ですから、CPL ではなくて CPA だとすると、減算の結果はマイナスになります。このように、比較の対象となる二つの整数の最上位ビットが、一方は 1 で他方は 0 の場合、CPL と CPA は異なる動作をします。

第 6 章 ビット演算

6.1 ビット演算の基礎

6.1.1 ビット演算とは何か

この章では、「ビット演算」と呼ばれる演算を実行する命令について説明したいと思います。

「ビット演算」という言葉は、ビットを対象として実行される演算のことを指して使われる場合と、ビット列を対象として実行される演算のことを指して使われる場合とがあります。ビット

列を対象とするビット演算というのは、ビット列を構成しているそれぞれのビットに対してビット演算を実行するという演算のことです。

CASL II が持っている 13 種類の演算命令のうちの 3 種類は、ビット演算を実行する命令です。このチュートリアルでは、それらの 3 種類の命令を、「ビット演算命令」と呼ぶことにします。

情報処理技術者試験センターが公開している CASL II の仕様書では、ビット演算命令は、「論理演算命令」と呼ばれています。しかし、このチュートリアルでは、「演算の対象を符号なし整数とみなす演算命令」という意味で「論理演算命令」という言葉を使っていますので、それと区別するために、ビット演算を実行する命令のことは「ビット演算命令」と呼ぶことにしました。

6.1.2　ビット演算命令

CASL II には、ビット演算命令として、次の 3 種類のものがあります。

`AND`	AND	論理積命令
`OR`	OR	論理和命令
`XOR`	eXclusive OR	排他的論理和命令

ビット演算命令は、汎用レジスタの内容や主記憶装置の語の内容を、整数とみなすのではなく、単なるビット列とみなして演算を実行します。ですから、他の演算命令とは違って、算術演算命令と論理演算命令との組にはなっていません。

6.1.3　ビット演算命令のオペランド

ビット演算命令のオペランドの形式は、第 4.1.4 項で説明した、演算オペランド形式です。

6.1.4　ビット演算命令によるフラグレジスタの設定

ビット演算命令も、他の演算命令と同じように、演算の結果に応じてフラグレジスタを変化させます。フラグレジスタのそれぞれのフラグには、次のようなものが設定されます。

オーバーフローフラグ	常に 0 を設定する。
サインフラグ	演算の結果として得られたビット列の最上位ビットが 1 ならば 1、0 ならば 0 を設定する。
ゼロフラグ	演算の結果として得られたビット列のすべてのビットが 0 ならば 1、そうでなければ 0 を設定する。

6.2　AND 命令

6.2.1　AND 命令の基礎

「論理積命令」と呼ばれる `AND` は、汎用レジスタの内容と主記憶装置の語の内容、または二つの汎用レジスタの内容に対して、「論理積」と呼ばれるビット演算を実行して、その結果を汎用レジスタに設定する命令です。

6.2.2　ビットを対象とする論理積

ビットを対象とする論理積というのは、与えられた二つのビットの両方が 1 のときだけ結果が 1 になって、それ以外のときは結果が 0 になる、というビット演算のことです。

A と B という二つのビットが与えられたとき、A と B の論理積は、次の表のようになります。

A	B	論理積
0	0	0
0	1	0
1	0	0
1	1	1

6.2.3　ビット列を対象とする論理積

ビット列を対象とする論理積というのは、与えられた二つのビット列を構成しているそれぞれのビットに対して論理積を実行して、その結果を並べることによってできるビット列を求める、という演算のことです。たとえば、

0000 0000 1111 1111

というビット列と、

0000 1111 0000 1111

というビット列との論理積を求めると、

0000 0000 0000 1111

という結果が得られます。

プログラムの例 and.cas

```
AND     START
        LAD   GR1,#00FF
        AND   GR1,HOGE
        DREG  MSG
        RET
HOGE    DC    #0F0F
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR1 に #00FF というビット列を設定して、次に GR1 の内容と HOGE 番地の語の内容との論理積を求めて、そののち DREG 命令でレジスタの内容を表示します。 HOGE 番地には #0F0F というビット列が格納されていますので、論理積の結果は #000F になります。したがって、 GR1 の内容は、

GR1:000F

と表示されるはずです。

6.2.4 マスク

論理積命令は、ビット列の中の一部分だけを残して、それ以外の部分をすべて 0 にしたい、というときに便利な命令です。

ビット列の中の一部分だけを残して、それ以外の部分をすべて 0 にする、という処理は、「マスク処理」(mask processing) と呼ばれます。

マスク処理をしたいときは、まず、ビット列の中の残したい部分が 1 になっていて、0 にしたい部分が 0 になっているビット列を作ります。そのような、残したい部分と 0 にしたい部分とを 1 と 0 で塗り分けたビット列は、「マスク」(mask) と呼ばれます。

マスク処理は、処理の対象となるビット列とマスクとの論理積を求めることによって実行することができます。たとえば、

1010 0110 1010 0110

というビット列の下位 8 ビットだけを残して、上位 8 ビットをすべて 0 にした、

0000 0000 1010 0110

というビット列を求めたいならば、

0000 0000 1111 1111

というマスクを作って、このマスクと、下位 8 ビットだけを残したいビット列との論理積を求めればいいわけです。

プログラムの例 mask.cas

```
MASK    START
        LAD   GR1,#A6A6
        AND   GR1,HOGE
        DREG  MSG
        RET
HOGE    DC    #00FF
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、マスク処理を実行することによって、 #A6A6 というビット列の下位 8 ビットだけを残して、上位 8 ビットをすべて 0 にした、 #00A6 というビット列を求めています。

6.3　OR 命令

6.3.1　OR 命令の基礎

「論理和命令」と呼ばれる OR は、汎用レジスタの内容と主記憶装置の語の内容、または二つの汎用レジスタの内容に対して、「論理和」と呼ばれるビット演算を実行して、その結果を汎用レジスタに設定する命令です。

6.3.2　ビットを対象とする論理和

ビットを対象とする論理和というのは、与えられた二つのビットの両方またはどちらか一方が 1 のときは結果が 1 になって、両方とも 0 のときだけ結果が 0 になる、というビット演算のことです。

A と B という二つのビットが与えられたとき、A と B の論理和は、次の表のようになります。

A	B	論理和
0	0	0
0	1	1
1	0	1
1	1	1

6.3.3　ビット列を対象とする論理和

ビット列を対象とする論理和というのは、与えられた二つのビット列を構成しているそれぞれのビットに対して論理和を実行して、その結果を並べることによってできるビット列を求める、という演算のことです。たとえば、

```
0000 0000 1111 1111
```

というビット列と、

```
0000 1111 0000 1111
```

というビット列との論理和を求めると、

```
0000 1111 1111 1111
```

という結果が得られます。

プログラムの例　or.cas

```
OR      START
        LAD   GR1,#00FF
        OR    GR1,HOGE
        DREG  MSG
        RET
HOGE    DC    #0F0F
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR1 に #00FF というビット列を設定して、次に GR1 の内容と HOGE 番地の語の内容との論理和を求めて、そののち DREG 命令でレジスタの内容を表示します。 HOGE 番地には #0F0F というビット列が格納されていますので、論理和の結果は #0FFF になります。したがって、 GR1 の内容は、

```
GR1:0FFF
```

と表示されるはずです。

6.3.4　ビットを立てる処理

ビットを強制的に 1 にすることを、ビットを「立てる」と言うことがあります。また、ビットが 1 であることを、ビットが「立っている」と表現することがあります。

論理和命令は、ビット列の中にあるいくつかの特定のビットを立てて、それら以外のビットはそのままの状態にしておきたい、というときに便利な命令です。

いくつかの特定のビットを立てたいときは、まず、立てたい位置のビットだけが 1 で、それ以外のビットが 0 になっているビット列を作ります。そして、そのビット列と、ビットを立てたいビット列との論理和を求めます。たとえば、

```
1010 0110 1010 0110
```

というビット列の下位 8 ビットを立てた、

```
1010 0110 1111 1111
```

というビット列を求めたいならば、

```
0000 0000 1111 1111
```

というビット列を作って、このビット列と、下位 8 ビットを立てたいビット列との論理和を求めればいいわけです。

プログラムの例 turnon.cas

```
TURNON   START
         LAD    GR1,#A6A6
         OR     GR1,HOGE
         DREG   MSG
         RET
HOGE     DC     #00FF
MSG      DC     'MESSAGE1'
         END
```

このプログラムは、#A6A6 というビット列の下位 8 ビットを立てることによって、#A6FF というビット列を求めています。

6.4 XOR 命令

6.4.1 XOR 命令の基礎

「排他的論理和命令」と呼ばれる XOR (eXclusive OR) は、汎用レジスタの内容と主記憶装置の語の内容、または二つの汎用レジスタの内容に対して、「排他的論理和」と呼ばれるビット演算を実行して、その結果を汎用レジスタに設定する命令です。

6.4.2 ビットを対象とする排他的論理和

ビットを対象とする排他的論理和というのは、与えられた二つのビットの一方が 1 で他方が 0 のときは結果が 1 になって、両方とも 0 のときと両方とも 1 のときとは結果が 0 になる、というビット演算のことです。

A と B という二つのビットが与えられたとき、A と B の排他的論理和は、次の表のようになります。

A	B	排他的論理和
0	0	0
0	1	1
1	0	1
1	1	0

6.4.3 ビット列を対象とする排他的論理和

ビット列を対象とする排他的論理和というのは、与えられた二つのビット列を構成しているそれぞれのビットに対して排他的論理和を実行して、その結果を並べることによってできるビット列を求める、という演算のことです。たとえば、

```
0000 0000 1111 1111
```

というビット列と、

```
0000 1111 0000 1111
```

というビット列との排他的論理和を求めると、

```
0000 1111 1111 0000
```

という結果が得られます。

プログラムの例 xor.cas

```
XOR      START
         LAD    GR1,#00FF
```

```
        XOR   GR1,HOGE
        DREG  MSG
        RET
HOGE    DC    #0F0F
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR1 に #00FF というビット列を設定して、次に GR1 の内容と HOGE 番地の語の内容との排他的論理和を求めて、そののち DREG 命令でレジスタの内容を表示します。HOGE 番地には #0F0F というビット列が格納されていますので、排他的論理和の結果は #0FF0 になります。したがって、GR1 の内容は、

GR1:0FF0

と表示されるはずです。

6.4.4　ビットの反転

ビットが 0 ならばそれを 1 にして、1 ならばそれを 0 にすることを、ビットを「反転させる」と言います。ビットを反転させる演算は、「否定演算」と呼ばれます。

排他的論理和命令は、ビット列の中にあるいくつかの特定のビットを反転させて、それら以外のビットはそのままの状態にしておきたい、というときに便利な命令です。

いくつかの特定のビットを反転させたいときは、まず、反転させたい位置のビットだけが 1 で、それ以外のビットが 0 になっているビット列を作ります。そして、そのビット列と、ビットを反転させたいビット列との排他的論理和を求めます。たとえば、

1010 0110 1010 0110

というビット列の下位 8 ビットを反転させた、

1010 0110 0101 1001

というビット列を求めたいならば、

0000 0000 1111 1111

というビット列を作って、このビット列と、下位 8 ビットを反転させたいビット列との排他的論理和を求めればいいわけです。

プログラムの例　not.cas

```
NOT     START
        LAD   GR1,#A6A6
        XOR   GR1,HOGE
        DREG  MSG
        RET
HOGE    DC    #00FF
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、#A6A6 というビット列の下位 8 ビットを反転させることによって、#A659 というビット列を求めています。

6.4.5　符号の反転

符号がプラスならばそれをマイナスにして、マイナスならばそれをプラスにすることを、符号を「反転させる」と言います。

第 1.3.5 項で説明したように、COMET II では、マイナスの整数は 2 の補数によって表現されます。ですから、整数の符号は、次の処理を実行することによって反転させることができます。

(1) すべてのビットを反転させる。
(2) 1 を加算する。

たとえば、6 の符号を反転させた結果を求めたいならば、まず、6 をあらわしている、

0000 0000 0000 0110

というビット列のすべてのビットを反転させて、その結果として得られた、

1111 1111 1111 1001

というビット列に 1 を加算します。そうすることによって得られた、

```
1111 1111 1111 1010
```

というビット列が、−6 をあらわしている 2 の補数です。

マイナスの整数をプラスに反転させる場合も、処理は同じです。−6 の符号を反転させたいならば、まず、−6 をあらわしている、

```
1111 1111 1111 1010
```

というビット列のすべてのビットを反転させて、その結果として得られた、

```
0000 0000 0000 0101
```

というビット列に 1 を加算します。そうすることによって、

```
0000 0000 0000 0110
```

というように、6 をあらわしているビット列を求めることができます。

プログラムの例 invert.cas

```
INVERT   START
         LAD   GR1,6
         XOR   GR1,ALLONE
         LAD   GR1,1,GR1
         DREG  MSG
         XOR   GR1,ALLONE
         LAD   GR1,1,GR1
         DREG  MSG
         RET
ALLONE   DC    #FFFF
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR1 に 6 という整数を設定して、次に XOR 命令と LAD を使ってその符号を反転させて、そののち DREG 命令でレジスタの内容を表示します。 GR1 の内容は、

```
GR1:FFFA
```

と表示されるはずです。これは、−6 をあらわしています。

このプログラムは、そののちさらに、−6 の符号を反転させて、ふたたび DREG 命令でレジスタの内容を表示します。 GR1 の内容は、

```
GR1:0006
```

と表示されるはずです。

第 7 章　シフト演算

7.1　シフト演算の基礎

7.1.1　シフト演算とは何か

この章では、「シフト演算」と呼ばれる演算を実行する命令について説明したいと思います。

シフト演算というのは、ビット列の 0 と 1 のパターンを左右に移動させるという演算のことです。ビット列に対してシフト演算を実行することを、ビット列を「シフトする」と言います。

ビット列を左にシフトした場合は右側に、右にシフトした場合は左側に、空のビットができます。それらの空のビットには、原則として 0 が補填されます。ただし、 SRA 命令だけは例外で、1 が補填される場合もあります。

ビット列が 2 進数で整数を表現しているとみなした場合、そのビット列を左へ 1 ビットだけシフトするというのは、それがあらわしている整数を 2 倍するということを意味しています。たとえば、10 を意味している、

```
0000 0000 0000 1010
```

というビット列を左へ 1 ビットだけシフトした、

```
0000 0000 0001 0100
```

というビット列は、$10 \times 2 = 20$ を意味しています。

同じように、ビット列を右へ 1 ビットだけシフトするというのは、それがあらわしている整数を 2 分の 1 倍するということを意味しています。たとえば、10 を意味している、

```
0000 0000 0000 1010
```

というビット列を右へ 1 ビットだけシフトした、

```
0000 0000 0000 0101
```

というビット列は、$10 \div 2 = 5$ を意味しています。

さらに、ビット列を左へ n ビットだけシフトするというのは、それがあらわしている整数を 2^n 倍するということで、ビット列を右へ n ビットだけシフトするというのは、それがあらわしている整数を 2^n 分の 1 倍するということです。

CASL II が持っている 13 種類の演算命令のうちの 4 種類は、シフト演算を実行する命令です。それらの 4 種類の命令は、「シフト演算命令」と呼ばれます。

7.1.2　シフト演算命令

CASL II には、シフト演算命令として、次の 4 種類のものがあります。

SLL	Shift Left Logical	論理左シフト命令
SRL	Shift Right Logical	論理右シフト命令
SLA	Shift Left Arithmetic	算術左シフト命令
SRA	Shift Right Arithmetic	算術右シフト命令

名前から分かるとおり、SLL と SRL は論理演算命令、SLA と SRA は算術演算命令です。

7.1.3　シフト演算命令のオペランド

CASL II が持っている 13 種類の演算命令のうちで、これまでに紹介した 9 種類は、第 4.1.4 項で説明した、演算オペランド形式という形式を使ってオペランドを書きます。

シフト演算命令に分類される 4 種類の命令は、演算命令の一種ですが、オペランドの形式は演算オペランド形式ではなくて、シフト演算命令に固有の独自形式です。

シフト演算命令のオペランドは、

汎用レジスタ名 , 定数

と書きます。そうすると、汎用レジスタ名で指定された汎用レジスタの内容が、定数で指定されたビット数だけシフトされます。たとえば、

```
GR3,5
```

というオペランドを持つシフト演算命令は、汎用レジスタの GR3 の内容を 5 ビットだけシフトします。

定数と指標レジスタの内容とを加算した結果で、シフトするビット数を指定することも可能です。その場合、オペランドは、

汎用レジスタ名 , 定数 , 指標レジスタ名

と書きます。そうすると、定数と、指標レジスタ名で指定された指標レジスタの内容とを加算した結果が、シフトするビット数になります。たとえば、GR1 の内容が 3 だとするとき、

```
GR3,5,GR1
```

というオペランドを持つシフト演算命令は、汎用レジスタの GR3 の内容を 8 ビットだけシフトします。

7.1.4　シフト演算命令によるフラグレジスタの設定

シフト演算命令も、他の演算命令と同じように、演算の結果に応じてフラグレジスタを変化させます。フラグレジスタのそれぞれのフラグには、次のようなものが設定されます。

オーバーフローフラグ	シフトによって最後に送り出されたビットの値。
サインフラグ	演算の結果として得られたビット列の最上位ビットが 1 ならば 1、0 ならば 0 を設定する。

ゼロフラグ　　演算の結果として得られたビット列のすべてのビットが 0 ならば 1、そうでなければ 0 を設定する。

7.2 SLL 命令

7.2.1 SLL 命令の基礎

「論理左シフト命令」と呼ばれる SLL (Shift Left Logical) は、汎用レジスタの内容を、指定されたビット数だけ左へシフトする命令です。

この命令は、符号なし整数を 2^n 倍したいときに使うことができます。

SLL 命令を使ってビット列をシフトすることを、「論理左シフト」と呼ぶことにします。

7.2.2 ビット列の論理左シフト

それでは、SLL 命令を使って、ビット列を論理左シフトするプログラムを書いてみましょう。

プログラムの例　sll.cas

```
SLL     START
        LAD   GR1,#0801
        SLL   GR1,4
        DREG  MSG
        RET
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR1 に #0801 というビット列を設定して、次に、そのビット列を 4 ビットだけ論理左シフトして、そののち DREG 命令でレジスタの内容を表示します。GR1 の内容は、

```
GR1:8010
```

と表示されるはずです。

4 ビットの論理左シフトというのは、符号なし整数を $2^4 = 16$ 倍するということです。#0801 があらわしている整数は 10 進数では 2049 で、#8010 があらわしている整数は 10 進数では 32784 ですから、確かに 16 倍されているということが分かります。

7.3 SRL 命令

7.3.1 SRL 命令の基礎

「論理右シフト命令」と呼ばれる SRL (Shift Right Logical) は、汎用レジスタの内容を、指定されたビット数だけ右へシフトする命令です。

この命令は、符号なし整数を 2^n 分の 1 倍したいときに使うことができます。

SRL 命令を使ってビット列をシフトすることを、「論理右シフト」と呼ぶことにします。

7.3.2 ビット列の論理右シフト

それでは、SRL 命令を使って、ビット列を論理右シフトするプログラムを書いてみましょう。

プログラムの例　srl.cas

```
SRL     START
        LAD   GR1,#8000
        SRL   GR1,4
        DREG  MSG
        RET
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタの GR1 に #8000 というビット列を設定して、次に、そのビット列を 4 ビットだけ論理右シフトして、そののち DREG 命令でレジスタの内容を表示します。GR1 の内容は、

```
GR1:0800
```

と表示されるはずです。

4 ビットの論理右シフトというのは、符号なし整数を 2^4 分の 1 倍（16 分の 1 倍）するということです。#8000 があらわしている整数は 10 進数では 32768 で、#0800 があらわしている整数は 10 進数では 2048 ですから、確かに 16 分の 1 倍されているということが分かります。

7.4　SLA 命令

7.4.1　SLA 命令の基礎

「算術左シフト命令」と呼ばれる SLA (Shift Left Arithmetic) は、汎用レジスタの下位 15 ビットの内容を、指定されたビット数だけ左へシフトする命令です。

SLA 命令は、符号をあらわしている最上位ビットを変化させません。ですから、この命令は、符号付き整数を 2^n 倍したいときに使うことができます。

SLA 命令を使ってビット列をシフトすることを、「算術左シフト」と呼ぶことにします。

7.4.2　プラスの整数の算術左シフト

それでは、SLA 命令を使って、プラスの整数を算術左シフトするプログラムを書いてみましょう。

プログラムの例　sla.cas

```
SLA      START
         LAD   GR1,#000A
         SLA   GR1,4
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR1 に #000A という整数を設定して、次に、その整数を 4 ビットだけ算術左シフトして、そののち DREG 命令でレジスタの内容を表示します。GR1 の内容は、

```
GR1:00A0
```

と表示されるはずです。

4 ビットの算術左シフトというのは、符号付き整数を $2^4 = 16$ 倍するということです。#000A というのは 10 進数では 10 で、#00A0 というのは 10 進数では 160 ですから、確かに 16 倍されているということが分かります。

算術左シフトでは、符号をあらわす最上位ビットは変化しません。プログラムを書いて確認しておくことにしましょう。

プログラムの例　sla2.cas

```
SLA2     START
         LAD   GR1,#4001
         SLA   GR1,1
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR1 に #4001 という整数を設定して、次に、その整数を 1 ビットだけ算術左シフトして、そののち DREG 命令でレジスタの内容を表示します。#4001 というのは、2 進数で書くと、

```
0100 0000 0000 0001
```

ということになります。SLA 命令がシフトするのは下位 15 ビットですので、14 番ビットの 1 は、最上位ビットを変化させることなく、ビット列の外へ送り出されます。したがって、GR1 の内容は、

```
GR1:0002
```

と表示されるはずです。

もしも、このプログラムの中の SLA 命令が SLL 命令だったとするならば、SLL 命令は最上位ビットを特別扱いしませんので、#4001 というビット列を 1 ビットだけシフトした結果は、#8002 になるはずです。SLA 命令の場合は、そうならずに、14 番ビットの 1 はビット列の外へ送り出されることになります。

シフト命令は、シフトによって最後に送り出されたビットの値をオーバーフローフラグに設定します。上のプログラムの SLA 命令は、最後に 1 を送り出しますので、フラグレジスタは、

```
FR:4(OF:1 SF:0 ZF:0)
```

というように、オーバーフローフラグが立っています。

7.4.3 マイナスの整数の算術左シフト

次に、SLA 命令を使って、マイナスの整数を算術左シフトするプログラムを書いてみましょう。

プログラムの例 sla3.cas

```
SLA3     START
         LAD   GR1,#FFF6
         SLA   GR1,4
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR1 に #FFF6 という整数を設定して、次に、その整数を 4 ビットだけ算術左シフトして、そののち DREG 命令でレジスタの内容を表示します。GR1 の内容は、

```
GR1:FF60
```

と表示されるはずです。

マイナスの整数の場合も、4 ビットの算術左シフトというのは、それを $2^4 = 16$ 倍するということです。#FFF6 というのは 10 進数では -10 で、#FF60 というのは 10 進数では -160 ですから、確かに 16 倍されているということが分かります。

このプログラムの場合、SLA 命令が最後に送り出すビットは 1 です。したがって、フラグレジスタは、

```
FR:6(OF:1 SF:1 ZF:0)
```

というように、オーバーフローフラグが立っています。

算術左シフトでは符号をあらわす最上位ビットが変化しないというのは、マイナスの整数の場合も同じです。プログラムを書いて確認しておくことにしましょう。

プログラムの例 sla4.cas

```
SLA4     START
         LAD   GR1,#8001
         SLA   GR1,1
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR1 に #8001 という整数を設定して、次に、その整数を 1 ビットだけ算術左シフトして、そののち DREG 命令でレジスタの内容を表示します。#8001 というのは、2 進数で書くと、

```
1000 0000 0000 0001
```

ということになります。SLA 命令がシフトするのは下位 15 ビットですので、14 番ビットの 0 は、最上位ビットを変化させることなく送り出されます。したがって、GR1 の内容は、

```
GR1:8002
```

と表示されるはずです。

7.5 SRA 命令

7.5.1 SRA 命令の基礎

「算術右シフト命令」と呼ばれる SRA (Shift Right Arithmetic) は、汎用レジスタの下位 15 ビットの内容を、指定されたビット数だけ右へシフトする命令です。

SRA 命令は、符号をあらわしている最上位ビットを変化させません。また、この命令は、シフトによってできた空のビットに、最上位ビットの値を補填します。たとえば、

1000 0000 0000 0000

というビットを SRA 命令で 4 ビットだけシフトすると、

1111 1000 0000 0000

というビット列が得られます。

このような性質があることから、SRA 命令は、符号付き整数を 2^n 分の 1 倍したいときに使うことができます。

SRA 命令を使ってビット列をシフトすることを、「算術右シフト」と呼ぶことにします。

7.5.2　プラスの整数の算術右シフト

それでは、SRA 命令を使って、プラスの整数を算術右シフトするプログラムを書いてみましょう。

プログラムの例　sra.cas

```
SRA      START
         LAD   GR1,#00A0
         SRA   GR1,4
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR1 に #00A0 という整数を設定して、次に、その整数を 4 ビットだけ算術右シフトして、そののち DREG 命令でレジスタの内容を表示します。GR1 の内容は、

GR1:000A

と表示されるはずです。

4 ビットの算術右シフトというのは、符号付き整数を 2^4 分の 1 倍（16 分の 1 倍）するということです。#00A0 というのは 10 進数では 160 で、#000A というのは 10 進数では 10 ですから、確かに 16 分の 1 倍されているということが分かります。

7.5.3　マイナスの整数の算術右シフト

次に、SRA 命令を使って、マイナスの整数を算術右シフトするプログラムを書いてみましょう。

プログラムの例　sra2.cas

```
SRA2     START
         LAD   GR1,#FF60
         SRA   GR1,4
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず汎用レジスタの GR1 に #FF60 という整数を設定して、次に、その整数を 4 ビットだけ算術右シフトして、そののち DREG 命令でレジスタの内容を表示します。GR1 の内容は、

GR1:FFF6

と表示されるはずです。

マイナスの整数の場合も、4 ビットの算術右シフトというのは、それを 2^4 分の 1 倍（16 分の 1 倍）するということです。#FF60 というのは 10 進数では -160 で、#FFF6 というのは 10 進数では -10 ですから、確かに 16 分の 1 倍されているということが分かります。

もしも、このプログラムの中の SRA 命令が SRL 命令だったとするならば、どうなるでしょうか。SRL 命令は、最上位ビットも含めて、空のビットに 0 を補填しますので、#FF60 というビット列を 4 ビットだけシフトした結果は、#0FF6 になります。SRA 命令の場合は、空のビットに最上位ビットの値を補填しますので、シフトした結果は #FFF6 になります。

第 8 章　分岐

8.1 分岐の基礎

8.1.1 分岐とは何か

CPU の中にある制御装置は、主記憶装置から命令を取り出して、それにしたがってコンピュータを構成しているさまざまな装置の動作を制御します。主記憶装置が命令を取り出す語のアドレスは、プログラムレジスタ (PR) に格納されています。

機械語の命令の多くは、自分自身の長さ（語数）をプログラムレジスタに加算します。ですから、主記憶装置の中に並んでいるそれぞれの命令は、それらが並んでいる順番のとおりに実行されていくことになります。

しかし、命令の中には、何らかの条件が成り立っている場合、自分自身の長さをプログラムレジスタに加算するのではなく、オペランドに記述された特定のアドレスをプログラムレジスタに設定するものもあります。そのような命令が実行された場合、場合によっては、次の位置にある命令ではなくて、どこか離れた位置にある命令が、次に実行されることになります。

何らかの条件が成り立っている場合に、オペランドに記述された特定のアドレスをプログラムレジスタに設定する命令は、「分岐命令」と呼ばれます。

条件が成り立っていたために、分岐命令によって、次に実行されるのが次の命令ではなくて、オペランドで指定された場所にある命令になることを、プログラムの実行が「分岐する」と言います。そして、分岐した場合に次に実行される命令のアドレスは、「分岐先」と呼ばれます。

8.1.2 分岐命令

CASL II には、分岐命令として、次の 6 種類のものがあります。

JPL	Jump on PLus	正分岐命令
JMI	Jump on MInus	負分岐命令
JNZ	Jump on Non Zero	否零分岐命令
JZE	Jump on ZEro	零分岐命令
JOV	Jump on OVerflow	オーバーフロー分岐命令
JUMP	unconditional JUMP	無条件分岐命令

これらの分岐命令のうちで、JUMP 命令は、「無条件分岐命令」という名前のとおり、条件とは無関係に、プログラムの実行を必ず分岐させます。それ以外の分岐命令は、フラグレジスタを構成しているそれぞれのフラグの値にもとづいて、分岐するかどうかを決定します。

JUMP 命令以外の分岐命令のそれぞれは、次の表に示される条件にもとづいて分岐します。

	OF	SF	ZF
JPL		0	0
JMI		1	
JNZ			0
JZE			1
JOV	1		

この表の中の 0 は、フラグが 0 であることが分岐の条件だという意味で、1 は、フラグが 1 であることが分岐の条件だという意味です。そして、空欄は、0 であっても 1 であっても、分岐するかどうかの判断には影響を与えない、という意味です。

たとえば、JPL 命令は、オーバーフローフラグの値とは無関係に、サインフラグが 0 でかつゼロフラグも 0 ならば分岐します。同じように、JMI 命令は、オーバーフローフラグとゼロフラグの値とは無関係に、サインフラグが 1 ならば分岐します。

8.1.3 分岐命令のオペランド

分岐命令のオペランドとしては、通常、ラベルを書きます。そうすると、条件が成り立っていた場合、そのラベルが与えられているアドレスが、分岐先としてプログラムレジスタに設定されることになります。たとえば、

```
    JNZ   HOGE
```

という命令は、もしもゼロフラグの値が 0 ならば、HOGE 番地をプログラムレジスタに設定する、という意味になります。

分岐命令の場合にも、アドレス修飾は可能です。分岐命令で、アドレス修飾の結果を分岐先にしたい場合は、

ラベル , 指標レジスタ名

という形のオペランドを書きます。そうすると、条件が成り立っていた場合には、ラベルが与えられているアドレスと、指標レジスタ名で指定された指標レジスタの内容とを加算することによって得られた実効アドレスが、分岐先としてプログラムレジスタに設定されることになります。たとえば、

```
    JZE   HOGE,GR5
```

という命令は、もしもゼロフラグの値が 1 ならば、HOGE 番地と GR5 の内容とを加算することによって得られた実効アドレスをプログラムレジスタに設定する、という意味になります。

8.2 ループ

8.2.1 ループの基礎

第 8.1.1 項で説明したように、機械語の命令の多くは、自分自身の長さ（語数）をプログラムレジスタに加算します。ですから、主記憶装置の中に並んでいるそれぞれの命令は、それらが並んでいる順番のとおりに実行されていくことになります。

しかし、分岐命令を使うことによって、プログラムレジスタに対して任意のアドレスを設定することが可能になります。もしも、現在のプログラムレジスタの内容よりも小さな数値をプログラムレジスタに設定したとすると、プログラムの実行は、同じところを何回もぐるぐると回り続けることになります。そのような、何回も実行されるプログラムの部分は、「ループ」(loop) または「繰り返し」(iteration) と呼ばれます。

8.2.2 無限ループ

実行が終了する可能性のないループは、「無限ループ」(infinite loop) と呼ばれます。

それでは、無限ループを含むプログラムを実際に書いてみましょう。

プログラムの例　infinite.cas

```
INFINITE START
LOOP     DREG  MSG
         JUMP  LOOP
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、まず DREG 命令でレジスタの内容を表示して、次に、LOOP 番地をプログラムレジスタに設定します。LOOP 番地に格納されているのは、先ほど実行した DREG 命令ですから、ふたたびレジスタの内容を表示して、LOOP 番地をプログラムレジスタに設定します。したがって、このプログラムを実行すると、レジスタの内容を表示するという動作が無限に実行され続けることになります。

このプログラムのような、無限ループを実行し続けるプログラムは、コンロロールキーを押しながら C のキーを押すことによって終了させることができます。

8.2.3 ループからの脱出

プログラムは、しばしばループを必要とします。しかし、必要とされるループは、多くの場合、無限ループではなくて、何らかの条件にもとづいて、さらに続行するか、それとも終了するか、ということを判断しながら実行が繰り返されるループ、すなわち、いつかはそこから脱出することのできるループです。そのようなループを作るためには、JUMP 命令以外の分岐命令を使う必要があります。

プログラムの例　downto.cas

```
DOWNTO   START
         LAD   GR1,10
LOOP     SUBA  GR1,ONE
         DREG  MSG
```

```
        JPL   LOOP
        RET
ONE     DC    1
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタのGR1に10という整数を設定して、次にGR1の内容から1を減算して、次にDREG命令でレジスタの内容を表示します。そして、JPL命令で、条件が成り立っているならばLOOP番地をプログラムレジスタに設定します。

JPL命令は、サインフラグが0でかつゼロフラグも0という条件で分岐する命令です。このプログラムでは、GR1の内容から1を減算した結果がプラスのあいだは、

FR:0(OF:0 SF:0 ZF:0)

というように条件が成り立っていますので、LOOP番地へ分岐することになります。しかし、GR1の内容は、10、9、8、7、……と1ずつ減っていきます。減算の結果が0になると、

FR:1(OF:0 SF:0 ZF:1)

というようにゼロフラグが立ちますので、その時点でループの実行は終了することになります。

8.2.4 比較演算命令を使ったループ

先ほど紹介したプログラムは、汎用レジスタの内容を、

10、9、8、7、6、5、4、3、2、1、0

というように1ずつ減らしていって、0になったところで終了するものでしたが、それとは逆に、

0、1、2、3、4、5、6、7、8、9、10

というように1ずつ増やしていって、10になったところで終了するプログラムは、どのように書けばいいのでしょうか。

問題は、9に1を加算して10になっても、符号が変化するわけではありませんので、フラグレジスタの内容はそれ以前の状態から変化しないということです。したがって、加算の命令の直後に分岐命令を書いたとしても、10になったところでループから脱出するということはできません。

このようなときに必要になるのが、比較演算命令です。

比較演算命令を使えば、ビット列とビット列とを比較して、その結果によってフラグレジスタを変化させることができます。汎用レジスタの内容と10とを比較すれば、汎用レジスタの内容が10よりも小さいか等しいか大きいかということがフラグレジスタに設定されます。

プログラムの例 upto.cas

```
UPTO    START
        LAD   GR1,0
LOOP    ADDA  GR1,ONE
        CPA   GR1,TEN
        DREG  MSG
        JMI   LOOP
        RET
ONE     DC    1
TEN     DC    10
MSG     DC    'MESSAGE1'
        END
```

このプログラムは、まず汎用レジスタのGR1に0という整数を設定して、次にGR1の内容に1を加算して、次にGR1の内容と10とを比較して、次にDREG命令でレジスタの内容を表示します。そして、JMI命令で、条件が成り立っているならばLOOP番地をプログラムレジスタに設定します。

JMI命令は、サインフラグが1という条件で分岐する命令です。このプログラムでは、GR1の内容が10よりも小さいあいだは、

FR:2(OF:0 SF:1 ZF:0)

というように条件が成り立っていますので、LOOP番地へ分岐することになります。しかし、GR1の内容は、0、1、2、3、……と1ずつ増えていきます。GR1の内容と10とが等しくなると、

FR:1(OF:0 SF:0 ZF:1)

というようにサインフラグが 0 になりますので、その時点でループの実行は終了することになります。

8.3　データ列処理

8.3.1　データ列処理の基礎

いくつかのデータが並んでできている列のことを、「データ列」と呼ぶことにしましょう。

データ列を構成しているそれぞれのデータを処理するためには、個々のデータの処理を繰り返すループを作る必要があります。

主記憶装置の連続するそれぞれの語の中に、データ列を構成しているそれぞれのデータが格納されているとするとき、そのデータ列を構成しているそれぞれのデータを処理するためには、ループの内部が実行されるたびに、処理の対象となる語のアドレスを変化させていく必要があります。これは、アドレス修飾を使うことによって実現することができます。

アドレス修飾というのは、第 3.6 節で説明したように、ラベルによって示されるアドレスと、指標レジスタの内容とを加算することです。アドレス修飾によって得られたアドレスは、「実効アドレス」と呼ばれます。指標レジスタが指定されていない場合は、ラベルによって示されるアドレスがそのまま実効アドレスになります。

8.3.2　データ列のロード

データ列を処理するプログラムの手始めとして、主記憶装置の連続する語から、データ列を構成している個々のデータを汎用レジスタにロードするプログラムを書いてみましょう。

プログラムの例　ldseq.cas

```
LDSEQ    START
         LAD   GR2,0
LOOP     LD    GR1,DATA,GR2
         DREG  MSG
         LAD   GR2,1,GR2
         CPA   GR2,LENGTH
         JMI   LOOP
         RET
LENGTH   DC    7
DATA     DC    6,3,2,7,8,9,4
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、あらかじめ GR2 に 0 を設定したのちにループに入ります。そして、ループの中で、DATA 番地に GR2 の内容を加算することによって得られた実効アドレスの語からデータを GR1 にロードして、DREG 命令でレジスタの内容を表示して、GR2 をインクリメントします。そして、GR2 がデータ列の長さ（LENGTH 番地に格納されている 7 という整数）よりも小さいならばループの先頭へ分岐して、そうでなければループから抜けます。

ループは、GR2 の内容を、0、1、2、3、4、5、6、7 と変化させて、それが 7 になったときに終了します。ですから、DATA 番地から始まる 7 個の語の内容が、順番に GR1 にロードされることになります。

8.3.3　データ列の合計

次に、データ列を構成しているそれぞれのデータの合計を求めるプログラムを書いてみましょう。

プログラムの例　sum.cas

```
SUM      START
         LAD   GR1,0
         LAD   GR2,0
LOOP     ADDA  GR1,DATA,GR2
         LAD   GR2,1,GR2
         CPA   GR2,LENGTH
         JMI   LOOP
         DREG  MSG
         RET
LENGTH   DC    7
DATA     DC    6,3,2,7,8,9,4
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、あらかじめ GR1 と GR2 のそれぞれに 0 を設定したのちにループに入ります。そして、ループの中で、DATA 番地に GR2 の内容を加算することによって得られた実効アドレスの語に格納されている整数を GR1 に加算して、そののち GR2 をインクリメントします。そして、GR2 がデータ列の長さよりも小さいならばループの先頭へ分岐して、そうでなければループから抜けます。

このプログラムは、ループから抜けたのち、DREG 命令でレジスタの内容を表示します。GR1 は、

```
GR1:0027
```

と表示されるはずです。16 進数の 27 は 10 進数で書くと 39 で、これは、6 と 3 と 2 と 7 と 8 と 9 と 4 を合計した結果です。

8.3.4 データ列の写像

次に、データ列を構成している個々のデータに対して何らかの処理を実行して、その結果から構成されるデータ列を作るという処理をするプログラム、つまりデータ列の写像を求めるプログラムを書いてみましょう。

プログラムの例 twice.cas

```
TWICE    START
         LAD   GR2,0
LOOP     LD    GR1,DATA,GR2
         SLA   GR1,1
         ST    GR1,RESULT,GR2
         LAD   GR2,1,GR2
         CPA   GR2,LENGTH
         JMI   LOOP
         DMEM  MSG,DATA,DUMPE
         RET
LENGTH   DC    7
DATA     DC    6,3,2,7,8,9,4
RESULT   DS    7
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、あらかじめ GR2 に 0 を設定したのちにループに入ります。そして、ループの中で、DATA 番地に GR2 の内容を加算することによって得られた実効アドレスの語に格納されている整数を GR1 にロードして、それを 1 ビットだけ算術左シフトして、その結果を、RESULT 番地に GR2 の内容を加算することによって得られた実効アドレスの語にストアして、そののち GR2 をインクリメントします。そして、GR2 がデータ列の長さよりも小さいならばループの先頭へ分岐して、そうでなければループから抜けます。

このプログラムは、ループから抜けたのち、DATA 番地から DUMPE 番地までの語の内容を DMEM 命令で表示します。

DATA 番地から始まる七つの語の内容は、

```
0006 0003 0002 0007 0008 0009 0004
```

と表示されるはずです。これは、プログラムの中に DC 命令で記述したとおりの内容です。そして、RESULT 番地から始まる七つの語の内容は、

```
000C 0006 0004 000E 0010 0012 0008
```

と表示されるはずです。これは、DATA 番地から始まる七つの語の内容のそれぞれを 2 倍した結果です。

8.4 選択

8.4.1 選択の基礎

第 8.2 節で、分岐命令を使うことによってループを作る方法について説明したわけですが、分岐命令の用途は、ループを作ることだけではありません。分岐命令には、動作を選択するという、もうひとつの用途があります。

与えられた目的をプログラムが果たすためには、しばしば、条件に応じていくつかの動作のうちからひとつを選択して実行する、ということが必要になります。条件による動作の選択は、分

岐命令を使うことによって記述することができます。

ループを作るときに使われるのは、プログラムの上のほうへ分岐する分岐命令だったわけですが、動作の選択に使われるのは、プログラムの下のほうへ分岐する分岐命令です。つまり、条件が成り立っている場合にいくつかの命令をスキップすることによって、動作を選択するわけです。

8.4.2　実行するかしないかの選択

1 個の分岐命令を使うことによって、条件が成り立っているときは動作を実行するけれども、成り立っていないときはその動作をスキップするようにプログラムを書くことによって、動作を実行するかしないかという選択を記述することができます。ただし、その場合に使われる分岐命令は、その条件が成り立っているときに分岐する命令ではなくて、その条件が成り立っていないときに分岐する命令です。たとえば、演算の結果が 0 だったときに動作を実行したいときは、演算の結果が 0 ではないときに分岐する JNZ 命令を使って、

```
            演算命令
            JNZ   SKIP
            結果が 0 だったときに実行したい動作
SKIP        演算の結果にかかわらず実行したい動作
```

というように書きます。

プログラムの例　zero.cas

```
ZERO    START
        LAD   GR1,0
        LAD   GR2,0
LOOP    LD    GR3,DATA,GR2
        JNZ   SKIP
        LAD   GR1,1,GR1
SKIP    LAD   GR2,1,GR2
        CPA   GR2,LENGTH
        JMI   LOOP
        DREG  MSG
        RET
LENGTH  DC    12
DATA    DC    0,3,0,0,-2,0,1,0,0,-5,0,4
MSG     DC    'MESSAGE1'
        END
```

これは、12 個のデータのうちに 0 が何個あるかということを GR1 で数えるプログラムです。実行すると、 GR1 の内容が、

```
GR1:0007
```

と表示されるはずです。これは、0 が 7 個あったという意味です。

このプログラムは、個々のデータを GR3 にロードして、それが 0 のときだけ GR1 をインクリメントします。0 のときだけインクリメントするという選択を実行するためには、0 ならばインクリメントをスキップする必要がありますので、分岐命令としては、 JNZ 命令が使われています。

8.4.3　二つの動作の選択

条件が成り立っているときは A という動作を実行して、成り立っていないときは B という動作を実行するというような、二つの動作のうちのどちらかを選択するという動作を記述するためには、二つのスキップが必要です。たとえば、演算の結果が 0 だったかそうでなかったかという条件で、二つの動作のうちのどちらかを選択したいならば、

```
            演算命令
            JNZ   SKIP1
            結果が 0 だったときに実行したい動作
            JUMP  SKIP2
SKIP1       結果が 0 ではなかったときに実行したい動作
SKIP2       演算の結果にかかわらず実行したい動作
```

というように書きます。

プログラムの例　evenodd.cas

```
EVENODD START
        LAD   GR2,0
LOOP    LD    GR1,DATA,GR2
        AND   GR1,MASK
        JNZ   SKIP1
```

```
          LAD   GR3,#8888
          JUMP  SKIP2
SKIP1     LAD   GR3,#1111
SKIP2     ST    GR3,RESULT,GR2
          LAD   GR2,1,GR2
          CPA   GR2,LENGTH
          JMI   LOOP
          DMEM  MSG,DATA,DUMPE
          RET
MASK      DC    #0001
LENGTH    DC    7
DATA      DC    2,11,3,12,6,7,14
RESULT    DS    7
DUMPE     DC    -1
MSG       DC    'MESSAGE1'
          END
```

これは、7 個の整数から構成されるデータ列を、#8888 または #9999 から構成されるデータ列に写像するプログラムです。偶数は #8888 に写像され、奇数は #1111 に写像されます。

整数が偶数なのか奇数なのかということは、最下位ビットが 0 か 1 かということを調べることによって判定することができます。調べたい整数と #0001 というマスクとの論理積を求めると、その結果は、偶数ならば #0000 、奇数ならば #0001 になります。

このプログラムは、ループから抜けたのち、 DATA 番地から DUMPE 番地までの語の内容を DMEM 命令で表示します。

DATA 番地から始まる七つの語の内容は、

```
0002 000B 0003 000C 0006 0007 000E
```

と表示されるはずです。これは、プログラムの中に DC 命令で記述したとおりの内容です。そして、 RESULT 番地から始まる七つの語の内容は、

```
8888 1111 1111 8888 8888 1111 8888
```

と表示されるはずです。これは、 DATA 番地から始まる七つの語の内容のそれぞれを、偶数ならば #8888 に、奇数ならば #1111 に写像した結果です。

第 9 章　サブルーチン

9.1　サブルーチンの基礎

9.1.1　サブルーチンとは何か

プログラムというのは、コンピュータに実行してほしい動作を記述した文書のことです。コンピュータに実行してほしい動作というのは、多くの場合、とても複雑なものです。

プログラムは、それが記述している動作の複雑さに比例して複雑なものになります。プログラムは、複雑になればなるほど、人間にとって理解しにくいものになります。

複雑なプログラムを書く場合には、それを人間にとって理解しやすいものにする工夫が必要です。人間にとって理解しやすいプログラムを書くための工夫のひとつは、適切な大きさの部品にそれを分解することです。

プログラムは、部品となるプログラムを組み合わせることによって構築することができます。プログラムの部品となるプログラムは、「サブルーチン」(subroutine) と呼ばれます。

サブルーチンは、自分の動作の一部分として、別のサブルーチンの動作を実行することができます。サブルーチンを実行することを、サブルーチンを「呼び出す」(call) と言います。プログラムは、いくつかのサブルーチンを、あるものが別のものを呼び出すという関係で結び合わせることによって構築されます。

ちなみに、ひとつのプログラムを構成しているサブルーチンのうちで、プログラム全体の要となっているひとつのサブルーチン、すなわち、それ以外のすべてのサブルーチンが、それから直接または間接的に呼び出されることになるひとつのサブルーチンは、「メインルーチン」(main routine) と呼ばれます。

9.1.2　サブルーチンに関連する命令

コンピュータは、通常、サブルーチンを組み合わせることによってプログラムを構築することを容易にするための命令を持っています。この章では、CASL II が持っているそのような命令を紹介したいと思います。

この章で紹介するのは、次の 4 種類の命令です。

CALL	CALL	コール命令
RET	RETurn from subroutine	リターン命令
PUSH	PUSH	プッシュ命令
POP	POP	ポップ命令

9.2　CALL 命令と RET 命令

9.2.1　CALL 命令

サブルーチンを組み合わせることによってプログラムを構築するためには、あるものが別のものを呼び出すという関係で、サブルーチンとサブルーチンとを結び合わせる必要があります。

第 9.1.1 項で説明したように、「呼び出す」(call) というのは、サブルーチンの動作を実行することです。

サブルーチンを呼び出したいとき、すなわち、サブルーチンの動作を実行したいときは、「コール命令」と呼ばれる CALL という命令を使います。この命令は、サブルーチンの先頭のアドレスをプログラムレジスタ (PR) に設定します。

CALL 命令は、 JUMP 命令とよく似ています。どちらも、アドレスをプログラムレジスタに設定する命令です。しかし、これらの二つの命令は、まったく同じ動作をするわけではありません。

CALL 命令の目的は、分岐ではなく、サブルーチンを呼び出すことです。その目的を果たすためには、呼び出したサブルーチンの動作が終了したのちに、 CALL 命令の次の命令を実行することができるように、その命令のアドレスを保管しておく必要があります。ですから、 CALL 命令は、自分の次にある命令のアドレスを保管して、そののちサブルーチンの先頭のアドレスをプログラムレジスタに設定します。

CALL 命令は次の命令のアドレスを、いったいどこに保管するのか、ということについては、次の節で説明することにしたいと思います。

9.2.2　CALL 命令のオペランド

CALL 命令のオペランドとしては、通常、ラベルを書きます。そうすると、そのラベルが与えられているアドレスが、サブルーチンの先頭のアドレスとしてプログラムレジスタに設定されることになります。たとえば、

```
CALL  HOGE
```

という命令は、自分の次にある命令のアドレスを保管したのち、 HOGE 番地をプログラムレジスタに設定する、という意味になります。

CALL 命令の場合にも、アドレス修飾は可能です。 CALL 命令で、アドレス修飾の結果をプログラムレジスタに設定したい場合は、

ラベル , 指標レジスタ名

という形のオペランドを書きます。そうすると、ラベルが与えられているアドレスと、指標レジスタ名で指定された指標レジスタの内容とを加算することによって得られた実効アドレスが、サブルーチンの先頭のアドレスとしてプログラムレジスタに設定されることになります。たとえば、

```
CALL  HOGE,GR5
```

という命令は、自分の次にある命令のアドレスを保管したのち、 HOGE 番地と GR5 の内容とを加算することによって得られた実効アドレスをプログラムレジスタに設定する、という意味になります。

9.2.3 RET 命令

サブルーチンは、自分の動作が終了したとき、自分を呼び出したサブルーチンの続きが実行されるように、CALL 命令によって保管されているアドレスをプログラムレジスタに設定する必要があります。

CALL 命令によって保管されているアドレスをプログラムレジスタ (PR) に設定するために使われるのが、「リターン命令」と呼ばれる RET という命令です。サブルーチンを書くときは、かならず、その動作が終了したときに、RET 命令が実行されるようにしないといけません。

メインルーチンも、その動作が終了したときに、RET 命令が実行されるように書く必要があります。その理由は、メインルーチンも、OS の中の CALL 命令がそれを呼び出すことによって実行されるからです。

RET 命令は、オペランドを必要としない命令です。

9.2.4 二つのサブルーチンから構成されるプログラム

それでは、CALL 命令と RET 命令を使って、二つのサブルーチンから構成されるプログラムを書いてみましょう。

プログラムの例 callret.cas

```
CALLRET  START
         DREG  MSGM
         CALL  SUB
         DREG  MSGM
         RET
MSGM     DC    'MAINROUT'
;
SUB      DREG  MSGS
         RET
MSGS     DC    'SUBROUTI'
         END
```

このプログラムでは、サブルーチンとサブルーチンとにあいだに注釈行を挿入してありますが、この注釈行は、絶対に必要なもの、というわけではありません。しかし、このように注釈行を挿入すると、どこからどこまでがひとつのサブルーチンなのかということが一目で分かりますので、プログラムが読みやすくなります。

このプログラムを実行すると、3 回、レジスターの内容が表示されます。MAINROUT というメッセージを伴っているものは、CALLRET 番地から始まるサブルーチン（これがメインルーチンです）の中の DREG 命令が表示したもので、SUBROUTI というメッセージを伴っているものは、SUB 番地から始まるサブルーチンの中の DREG 命令が表示したものです。

SUB 番地から始まるサブルーチンの中の DREG 命令は、レジスターの内容に加えて、

```
 * StackAreaDump *    Start:EFFF End:EFFF
  EFF8: ---- ---- ---- ---- ---- ---- ---- 0004          .
```

というものを表示します。これについては、次の節で説明することにしたいと思います。

9.3 スタック

9.3.1 スタックの基礎

データ列で、データの追加と取り出しが一方の端だけに限定されているものは、「スタック」(stack) と呼ばれます。

スタックにデータを追加することを、スタックにデータを「プッシュする」(push) と言います。そして、スタックからデータを取り出すことを、スタックからデータを「ポップする」(pop) と言います。

スタックというのは、行き止まりのある洞窟のようなものだと考えるといいでしょう。行き止まりのある洞窟に物体を詰め込んでいって、そののちそこから物体を取り出そうとすると、物体は、詰め込んだ順番とは逆の順番で取り出されることになります。また、スタックというのは、物体を上下に積み重ねたようなものだと考えることもできます。物体を上下に積み重ねていって、そののちそこから物体を降ろそうとすると、積み重ねた順番とは逆の順番で降ろされることになります。

たとえば、A、B、C、D、E という順番で、スタックにデータをプッシュしたとしましょう。そうすると、スタックの中は、

E	D	C	B	A

という状態になります。この状態のスタックからデータをポップしていくと、E、D、C、B、A という順番でデータがポップされることになります。

9.3.2　スタック領域

COMET II は、主記憶装置の一部分をスタックとして使います。スタックとして使われる主記憶装置の一部分は、「スタック領域」と呼ばれます。

第 1.3.8 項で紹介した「スタックポインタ」というレジスタ（英語では stack pointer、略称は SP）は、最後にプッシュされたデータが格納されている語のアドレスを保持しています。

COMET II においては、データをプッシュするというのは、

(1) スタックポインタをデクリメントする。
(2) スタックポインタに格納されているアドレスが示している語にデータを設定する。

という動作を実行することで、データをポップするというのは、

(1) スタックポインタに格納されているアドレスが示している語からデータを取得する。
(2) スタックポインタをインクリメントする。

という動作を実行することです。

スタックポインタに格納されているアドレスが示している主記憶装置の語は、スタックの「最上段」と呼ばれます（このネーミングは、物体を上下に積み重ねたものというスタックのイメージにもとづくものです）。

9.3.3　スタックとサブルーチン

前の節で、CALL 命令は、自分の次にある命令のアドレスを保管したのちにサブルーチンの先頭のアドレスをプログラムレジスタに設定すると説明しましたが、自分の次にある命令のアドレスをどこに保管するのかということについては説明しませんでした。

CALL 命令がアドレスを保管する場所は、スタックです。したがって、CALL というのは、自分の次にある命令のアドレスをスタックにプッシュしたのちにサブルーチンの先頭のアドレスをプログラムレジスタに設定する命令で、RET というのは、スタックからポップしたアドレスをプログラムレジスタに設定する命令ということになります。

サブルーチンの実行が終わったのちに戻るべき命令のアドレスを保管するためにスタックを使うというのは、理にかなったことです。なぜなら、サブルーチンの実行が終了したのちにそれを呼び出したサブルーチンに戻っていく順序は、サブルーチンがサブルーチンを呼び出した順序とは逆になるからです。

たとえば、サブルーチン A がサブルーチン B を呼び出して、サブルーチン B がサブルーチン C を呼び出して、サブルーチン C がサブルーチン D を呼び出して、サブルーチン D がサブルーチン E を呼び出したとしましょう。サブルーチンがサブルーチンを呼び出すという関係を矢印で示すと、

$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$$

というようになります。そして、スタックには、A の中のアドレス、B の中のアドレス、C の中のアドレス、D の中のアドレスが、この順番でプッシュされています。サブルーチンの動作が終了したときは、それを呼び出したサブルーチンに戻る必要があります。サブルーチンからサブルーチンに戻るという関係を矢印で示すと、

$$A \leftarrow B \leftarrow C \leftarrow D \leftarrow E$$

というようになります。ですから、スタックからアドレスをポップすれば、必然的に、それが戻るべきサブルーチンの中のアドレスということになります。

9.3.4　スタック領域のダンプ

DREG 命令は、OS ではないプログラムによって何らかのデータがスタックにプッシュされている場合、レジスタの内容に加えて、スタック領域の内容も表示します。

前の節で紹介したプログラムを実行すると、DREG 命令が、3 回、実行されます。2 回目に実行される DREG 命令は、SUB 番地から始まるサブルーチンの中にあるものです。このサブルーチンは、メインルーチンから呼び出されることによって実行されますので、その中の DREG 命令が実行されるとき、スタックには、戻るべきアドレスがプッシュされています。ですから、この DREG 命令は、レジスタの内容に加えて、

```
* StackAreaDump *    Start:EFFF End:EFFF
 EFF8: ---- ---- ---- ---- ---- ---- ---- 0004          .
```

というように、スタック領域の内容も表示します。表示されているのは #EFF8 番地から #EFFF 番地までで、#EFFF 番地には、CALL 命令によってプッシュされた #0004 というアドレスが格納されています。

9.4 PUSH 命令と POP 命令

9.4.1 PUSH 命令と POP 命令の基礎

前の節で説明したように、CALL というのは、サブルーチンの実行が終了したのちに戻るべき命令のアドレスをスタックにプッシュしたのちにサブルーチンの先頭のアドレスをプログラムレジスタに設定する命令で、RET というのは、スタックからポップしたアドレスをプログラムレジスタに設定する命令です。

ところで、COMET II のスタックは、CALL 命令と RET 命令だけのために存在するものなのでしょうか。答えはノーです。それは、さまざまな目的のために自由に使うことのできる汎用的な道具です。COMET II は、スタックを自由に使うことができるように、スタックにデータをプッシュする PUSH 命令と、スタックからデータをポップする POP 命令を持っています。

9.4.2 PUSH 命令

スタックにデータをプッシュしたいときは、「プッシュ命令」と呼ばれる PUSH という命令を使います。この命令は、実効アドレスをスタックにプッシュします。

特定のデータをスタックにプッシュしたいときは、PUSH 命令のオペランドとして、プッシュしたいデータをあらわす定数を書きます。そうすると、その定数によってあらわされるデータがスタックにプッシュされることになります。たとえば、

```
PUSH  #2FA3
```

という命令は、#2FA3 というデータをスタックにプッシュする、という意味になります。

PUSH 命令の場合にも、アドレス修飾は可能です。PUSH 命令で、アドレス修飾の結果によって示される語の内容をスタックにプッシュしたい場合は、

定数 , 指標レジスタ名

という形のオペランドを書きます。そうすると、定数によってあらわされるデータと、指標レジスタ名で指定された指標レジスタの内容とを加算することによって得られた実効アドレスがスタックにプッシュされることになります。たとえば、

```
PUSH  6,GR5
```

という命令は、6 という整数と GR5 の内容とを加算することによって得られた実効アドレスをスタックにプッシュする、という意味になります。

指標レジスタの内容をそのままスタックにプッシュしたい場合は、

0, 指標レジスタ名

という形のオペランドを書きます。そうすると、指標レジスタ名で指定された指標レジスタの内容がそのままスタックにプッシュされることになります。たとえば、

```
PUSH  0,GR5
```

という命令は、GR5 の内容をスタックにプッシュする、という意味になります。

9.4.3 POP 命令

スタックからデータをポップしたいときは、「ポップ命令」と呼ばれる POP という命令を使います。この命令は、スタックからデータをポップして、そのデータを汎用レジスタに設定します。

POP 命令のオペランドには、汎用レジスタ名を書きます。そうすると、汎用レジスタ名で指定された汎用レジスタに、スタックからポップしたデータが設定されます。たとえば、

```
    POP   GR1
```

という命令は、スタックからデータをポップして、そのデータを GR1 に設定する、という意味になります。

プログラムの例　pushpop.cas

```
PUSHPOP  START
         DREG  MSG
         PUSH  #7777
         DREG  MSG
         LAD   GR5,#8888
         PUSH  0,GR5
         DREG  MSG
         POP   GR1
         DREG  MSG
         POP   GR1
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、DREG 命令でレジスターとスタックの内容を表示しながら、#7777 と #8888 という二つのデータをスタックにプッシュして、そののちそれらのデータをポップします。#7777 のプッシュは、PUSH 命令のオペランドに定数を書くという方法を使っていて、#8888 のプッシュは、PUSH 命令のオペランドに 0 と指標レジスタ名を書くという方法を使っています。

9.4.4　汎用レジスタの退避と復元

PUSH 命令と POP 命令の最大の目的は、汎用レジスタの退避と復元です。

汎用レジスタを使うサブルーチンを呼び出した場合は、その前と後とで、汎用レジスタの内容に変化が生じている可能性があります。ですから、サブルーチンを呼び出す前と後とで内容が変化しては困る汎用レジスタがある場合は、呼び出す前にその汎用レジスタの内容をどこかに退避させて、呼び出したのちにそれを復元する必要があります。汎用レジスタの内容を退避させるための場所としては、通常、スタックが使われます。

プログラムの例　refuge.cas

```
REFUGE   START
         LAD   GR1,#1111
         LAD   GR2,#2222
         DREG  MSGM
         PUSH  0,GR2
         CALL  SUB
         POP   GR2
         DREG  MSGM
         RET
MSGM     DC    'MESSAGE1'
;
SUB      LAD   GR1,#FFFF
         LAD   GR2,#AAAA
         RET
         END
```

このプログラムのメインルーチンは、GR1 に #1111 を設定して、GR2 に #2222 を設定したのち、SUB 番地から始まるサブルーチンを呼び出します。そのサブルーチンは、GR1 に #FFFF を設定して、GR2 に #AAAA を設定します。GR1 に関しては、最初に設定した #1111 は失われてしまいますが、GR2 に関しては、サブルーチンを呼び出す前にその内容をスタックに退避させて、サブルーチンが終了したのちにそれを復元していますので、#2222 が温存されます。

第 10 章　マクロ命令とその他の機械語命令

10.1 マクロ命令の基礎

10.1.1 この章について

この章では、これまでの章には登場しなかった CASL II の命令と構文について説明していきたいと思います。

この章で説明するのは、4 種類のマクロ命令、2 種類の機械語命令、そして「リテラル」と呼ばれる構文です。

10.1.2 マクロ命令についての復習

マクロ命令については、すでに第 1.5.3 項で簡単に説明しましたので、まずは、そこで説明したことを復習しておくことにしましょう。

CASL II の命令は、アセンブラ命令、マクロ命令、機械語命令という 3 種類のグループに分類することができます。

マクロ命令というのは、複数の機械語の命令に変換される命令のことです。

10.1.3 マクロ命令の種類

マクロ命令には、次の 4 種類のものがあります。

`IN`	INput	入力命令
`OUT`	OUTput	出力命令
`RPUSH`	Register PUSH	レジスタ一括退避命令
`RPOP`	Register POP	レジスタ一括復元命令

10.2 IN 命令

10.2.1 IN 命令の基礎

COMET II には、入出力装置を接続することができます。COMET II に接続された入出力装置は、文字列の読み込みと書き込みができます。1 回の読み込みや 1 回の書き込みで読み込まれたり書き込まれたりする文字列の単位は、「レコード」(record) と呼ばれます。

`IN` は、入力装置から 1 レコードを読み込んで、それを主記憶装置に格納する命令です。

`IN` 命令を実行しても、汎用レジスタの内容は、それ以前の状態から変化しません。しかし、フラグレジスタの内容は、変化するかもしれません。

10.2.2 入力領域と入力文字長領域

`IN` 命令を使って入力装置から文字列を読み込むためには、あらかじめ、「入力領域」と呼ばれる領域と、「入力文字長領域」と呼ばれる領域を、主記憶装置の中に作っておく必要があります。

入力領域というのは、`IN` 命令によって読み込まれた文字列が格納される領域です。必要な長さ（語数）は 256 語です。読み込まれた文字列を構成する個々の文字は、入力領域の先頭から順番に、1 語に 1 文字で格納されます。レコードを区切るための文字列（改行など）は、入力領域には格納されません。

入力された文字列の長さが 256 文字よりも短い場合、余った領域は、`IN` 命令が実行される前の状態を維持します。逆に、入力された文字列の長さが 256 文字よりも長い場合、257 文字目以降の文字は切り捨てられます。

入力文字長領域というのは、`IN` 命令によって読み込まれた文字列の長さが格納される領域です。必要な長さ（語数）は 1 語です。

文字を読み込む位置がファイルの終わり (end of file, EOF) に到達しているときに `IN` 命令が実行された場合、入力文字長領域には −1 が格納されます。

10.2.3 IN 命令のオペランド

`IN` 命令のオペランドは、

$\boxed{\text{ラベル}_1}$, $\boxed{\text{ラベル}_2}$

と書きます。ラベル $_1$ は入力領域の先頭のアドレスで、ラベル $_2$ は入力文字長領域のアドレスです。たとえば、

```
    IN     INAREA,INLENGTH
```

という IN 命令は、入力装置から文字列を読み込んで、 INAREA 番地から始まる領域にその文字列を格納して、 INLENGTH 番地にその文字列の長さを格納します。

プログラムの例　in.cas

```
IN        START
          IN     INAREA,INLENGTH
          DMEM   MSG,INAREA,DUMPE
          RET
INAREA    DS     256
INLENGTH  DS     1
MSG       DC     'MESSAGE1'
DUMPE     DC     -1
          END
```

このプログラムは、 IN 命令を使って入力装置から文字列を読み込んで、 DMEM 命令で入力領域と入力文字長領域を表示します。このプログラムを実行すると、

```
IN ?
```

というプロンプトが表示されますので、文字列を入力して、最後にエンターキーを押してください。たとえば、

```
I love CASL II.
```

という文字列を入力したとすると、入力領域の先頭の部分は、

```
0008: 0049 0020 006C 006F 0076 0065 0020 0043  I love C
0010: 0041 0053 004C 0020 0049 0049 002E 0000  ASL II..
```

と表示されて、入力文字長領域（ MESSAGE1 という文字列が格納されている領域の直前）は、 000F と表示されます。入力文字長領域に格納された #000F （10 進数で書くと 15）は、入力された文字列の長さが 15 文字だということを意味しています。

10.3　OUT 命令

10.3.1　OUT 命令の基礎

OUT は、主記憶装置に格納されている文字列を、出力装置に 1 レコードとして書き込む命令です。

OUT 命令を実行しても、汎用レジスタの内容は、それ以前の状態から変化しません。しかし、フラグレジスタの内容は、変化するかもしれません。

10.3.2　出力領域と出力文字長領域

OUT 命令を使って出力装置に文字列を書き込むためには、あらかじめ、「出力領域」と呼ばれる領域と、「出力文字長領域」と呼ばれる領域を主記憶装置の中に作って、書き込みに必要なデータをそこに格納しておく必要があります。

出力領域というのは、 OUT 命令を使って出力装置に書き込みたい文字列を格納しておく領域です。出力領域には、書き込みたい文字列を構成する個々の文字を、出力領域の先頭から順番に、1 語に 1 文字で格納しておきます。

第 1.3.6 項で説明したように、COMET II では、文字を語に格納する場合、文字を下位の 8 ビットに格納して、上位の 8 ビットはすべて 0 にします。たとえば、 A という文字は、

```
0000 0000 0100 0001
```

というビット列によってあらわされます。しかし、 OUT 命令を使って文字列を出力装置に書き込む場合、上位の 8 ビットは無視されますので、そのすべてを 0 にする必要はありません。ですから、

```
1101 1001 0100 0001
```

というビット列を語に格納した場合も、 A という文字が書き込まれることになります。

出力文字長領域というのは、 OUT 命令を使って出力装置に書き込みたい文字列の長さを格納しておく領域です。

10.3.3 OUT 命令のオペランド

OUT 命令のオペランドは、

ラベル$_1$, ラベル$_2$

と書きます。ラベル$_1$は出力領域の先頭のアドレスで、ラベル$_2$は出力文字長領域のアドレスです。たとえば、

```
    OUT     OUTAREA,OUTLENGT
```

という OUT 命令は、OUTAREA 番地から始まる、OUTLENGT 番地に格納されている長さの領域に格納されている文字列を出力装置に書き込みます。

プログラムの例 out.cas

```
OUT       START
          OUT    OUTAREA,OUTLENGT
          RET
OUTAREA   DC     'I love CASL II.'
OUTLENGT  DC     15
          END
```

このプログラムは、OUT 命令を使って出力装置に文字列を書き込みます。このプログラムを実行すると、

```
I love CASL II.
```

という、長さが 15 文字の文字列が出力装置に書き込まれます。

10.4 RPUSH 命令と RPOP 命令

10.4.1 RPUSH 命令と RPOP 命令の基礎

第 9.4 節で説明したように、PUSH 命令を使うことによってスタックにデータをプッシュすることができて、POP 命令を使うことによってスタックからデータをポップすることができます。そして、この二つの命令の最大の目的は、汎用レジスタの退避と復元です。サブルーチンを呼び出す前に汎用レジスタの内容を退避させておいて、そのサブルーチンの実行が終了したのちに、退避させたデータを復元したいときには、通常、この二つの命令が使われます。

RPUSH と RPOP は、GR0 を除くすべての汎用レジスタについて退避と復元を実行したい、というときに使うことのできるマクロ命令です。

RPUSH と RPOP は、どちらも、オペランドを必要としない命令です。

10.4.2 RPUSH 命令

RPUSH は、GR1、GR2、GR3、・・・、GR7 という順番で、汎用レジスタの内容をスタックにプッシュするマクロ命令です。したがって、RPUSH 命令を機械語の命令に変換した結果は、

```
    PUSH  0,GR1
    PUSH  0,GR2
    PUSH  0,GR3
    PUSH  0,GR4
    PUSH  0,GR5
    PUSH  0,GR6
    PUSH  0,GR7
```

という機械語命令の列を機械語に変換した結果と同じだと考えることができます。

10.4.3 RPOP 命令

RPOP は、GR7、GR6、GR5、・・・、GR1 という順番で、スタックからポップしたデータを汎用レジスタに設定するマクロ命令です。したがって、RPOP 命令を機械語の命令に変換した結果は、

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

という機械語命令の列を機械語に変換した結果と同じだと考えることができます。

プログラムの例　rpushpop.cas

```
RPUSHPOP START
         LAD   GR1,#1111
         LAD   GR2,#2222
         LAD   GR3,#3333
         LAD   GR4,#4444
         LAD   GR5,#5555
         LAD   GR6,#6666
         LAD   GR7,#7777
         DREG  MSGM
         RPUSH
         CALL  SUB
         RPOP
         DREG  MSGM
         RET
MSGM     DC    'MESSAGE1'
;
SUB      LAD   GR1,#FFFF
         LAD   GR2,#EEEE
         LAD   GR3,#DDDD
         LAD   GR4,#CCCC
         LAD   GR5,#BBBB
         LAD   GR6,#AAAA
         LAD   GR7,#9999
         DREG  MSGS
         RET
MSGS     DC    'MESSAGE2'
         END
```

このプログラムは、GR1 から GR7 までのすべての汎用レジスタにデータを設定して、次に RPUSH 命令を実行して、そののち SUB 番地から始まるサブルーチンを呼び出しています。ところが、そのサブルーチンは、GR1 から GR7 までのすべての汎用レジスタに、それまでとは異なるデータを設定していきます。しかし、汎用レジスタの以前の内容はスタックに退避させてありますので、RPOP 命令を実行することによって、汎用レジスタをもとの内容に復元することができます。

10.5　その他の機械語命令

10.5.1　この節について

第 1.5.3 項で説明したように、CASL II には、細かく分類すると 36 種類の命令があります。

36 種類の命令のうちで、これまでに登場した命令は、34 種類です。したがって、いまだに登場していない命令が、まだ 2 種類あるわけです。それは、SVC と NOP という二つの機械語命令です。

この節では、SVC と NOP という、最後に残った二つの機械語命令について説明したいと思います。

10.5.2　SVC 命令

SVC (SuperVisor Call) は、「スーパバイザコール命令」と呼ばれます。これは、OS の内部にある機能を実行する命令です。

SVC 命令は、オペランドで指定されたアドレスにある、OS の内部の機能を実行します。たとえば、

```
    SVC   38
```

という SVC 命令は、38 番地にある OS の内部の機能を実行します。

SVC 命令を実行すると、汎用レジスタとフラグレジスタの内容が変化する可能性があります。

IN 命令と OUT 命令は、マクロ命令ですから複数の命令に変換されるわけですが、IN や OUT を変換した結果の中には、SVC 命令も含まれています。

10.5.3　NOP 命令

NOP (No OPeration) は、「ノーオペレーション命令」と呼ばれます。これは、何もしない命令です。

NOP 命令は、プログラムの中のひとつの場所に複数のラベルを与えたいときなどに使うことができます。たとえば、

```
HOGE    NOP
GAMO    LAD   GR0,31
```

と書くことによって、GR0 に 31 を設定するという命令のある場所に、HOGE と GAMO という二つのラベルを与えることができます。ただし、NOP 命令自体にも長さがありますので、厳密に言えば、HOGE 番地と GAMO 番地は同じアドレスではありません。

10.6 リテラル

10.6.1 リテラルの基礎

CASL II では、「リテラル」と呼ばれるものをオペランドの中に書くことができます。
リテラルは、

=[定数]

という構文を持つ記述です。この中の「定数」のところに書くことのできるものは、10 進定数、16 進定数、文字定数のいずれかです。たとえば、

```
=538   =#F60A   ='A'
```

というようなリテラルを書くことができます。
リテラルは、その中の定数があらわしているデータを主記憶装置の語に設定する DC 命令を生成する、という意味を持つ記述です。たとえば、

```
    LD    GR1,=3
```

という LD 命令を書いたとしましょう。この中の =3 という記述は、リテラルです。この命令を含むプログラムをアセンブルすると、自動的に、

```
    DC    3
```

という DC 命令が生成されて、3 という整数が主記憶装置の語に設定されます。

10.6.2 リテラルの値

リテラルの意味は、「その中の定数があらわしているデータを主記憶装置の語に設定する DC 命令を生成する」ということですが、リテラルは、意味だけではなくて、「値」と呼ばれるものも持ちます。
リテラルの値は、そのリテラルによって生成された DC 命令がデータを設定した主記憶装置の語のアドレスです。
オペランドの中にリテラルを書くと、そのリテラルの値が、処理の対象となる語のアドレスになります。たとえば、

```
    LD    GR1,=3
```

という LD 命令は、自動的に生成された DC 命令によって主記憶装置に格納されているデータを GR1 に設定します。ですから、3 という整数が GR1 に設定されることになります。

プログラムの例 literal.cas

```
LITERAL  START
         LD    GR0,=3
         LD    GR1,=#CDEF
         LD    GR2,='A'
         DREG  MSG
         RET
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、GR0、GR1、GR2 のそれぞれに、リテラルを使ってデータを設定して、そののち DREG 命令でレジスタの内容を表示します。実行すると、GR0 と GR1 と GR は、

```
GR0:0003 GR1:CDEF GR2:0041
```

と表示されるはずです。

第 11 章　乗算と除算

11.1　加算の繰り返しによる乗算

11.1.1　この章について

COMET II は、加算と減算の命令は持っていますが、乗算と除算の命令は持っていません。しかし、乗算や除算も、さまざなま命令を組み合わせることによって実行することができます。

この章では、乗算や除算を実行させるためには、命令をどのように組み合わせればよいか、ということについて説明したいと思います。

11.1.2　乗算の基礎

乗算というのは、加算の繰り返しです。たとえば、3×7 という乗算の結果は、

$3+3+3+3+3+3+3$

というように 7 個の 3 を加算するか、

$7+7+7$

というように 3 個の 7 を加算することによって求めることができます。

11.1.3　加算の繰り返しによる乗算のプログラム

それでは、加算を繰り返すことによって乗算をするプログラムを書いてみましょう。

プログラムの例　iteradd.cas

```
ITERADD  START
         LAD    GR0,0
         LD     GR1,B
         JMI    EXIT
LOOP     JZE    EXIT
         ADDA   GR0,A
         SUBA   GR1,=1
         JUMP   LOOP
EXIT     ST     GR0,PRODUCT
         DMEM   MSG,PRODUCT,DUMPE
         RET
A        DC     3
B        DC     7
PRODUCT  DS     1
DUMPE    DC     -1
MSG      DC     'MESSAGE1'
         END
```

このプログラムは、A 番地の整数について、B 番地の整数回だけ加算を繰り返して、その結果を PRODUCT 番地に格納して、DMEM 命令で PRODUCT 番地を表示します。A 番地には 3、B 番地には 7 が格納されていますので、PRODUCT 番地には 21（16 進数では #0015 ）が格納されます。

11.2　減算の繰り返しによる除算

11.2.1　除算の基礎

除算というのは、減算の繰り返しです。それ以上は減算ができなくなるまで減算を繰り返したとき、減算を繰り返した回数は何回だったかということが、減算の商になります。そして、それ以上は減算ができなくなったときに残っている数値が、減算の余りになります。

たとえば、$34 \div 7$ という除算は、

1 回目　$34-7 \rightarrow 27$

2 回目　$27-7 \rightarrow 20$

3 回目　$20-7 \rightarrow 13$

4 回目　$13-7 \rightarrow 6$

というように 34 から 7 を減算していくと、4 回で、それ以上は減算ができなくなりますので、商は 4 ということになります。そして、4 回目の減算の結果が 6 ですから、余りは 6 ということになります。

11.2.2 減算の繰り返しによる除算のプログラム

それでは、減算を繰り返すことによって除算をするプログラムを書いてみましょう。

プログラムの例 itersub.cas

```
ITERSUB  START
         LAD   GR0,0
         LD    GR1,A
LOOP     CPA   GR1,B
         JMI   EXIT
         SUBA  GR1,B
         ADDA  GR0,=1
         JUMP  LOOP
EXIT     ST    GR0,QUOTIENT
         ST    GR1,MOD
         DMEM  MSG,QUOTIENT,DUMPE
         RET
A        DC    34
B        DC    7
QUOTIENT DS    1
MOD      DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、A 番地の整数から B 番地の整数を減算するという動作を繰り返して、減算することのできた回数を QUOTIENT 番地に格納して、減算することができなくなったのちに残った整数を MOD 番地に格納して、DMEM 命令で QUOTIENT 番地と MOD 番地を表示します。A 番地には 34、B 番地には 7 が格納されていますので、QUOTIENT 番地には 4 が格納されて、MOD 番地には 6 が格納されます。

11.3 筆算による乗算

11.3.1 筆算による乗算の基礎

第 11.1 節で、加算を繰り返すことによって乗算をする方法について説明しましたが、乗算の方法は、それだけではありません。たとえば、筆算というのも、乗算をする方法のひとつです。筆算による乗算は、単純な加算の繰り返しよりも効率よく結果を求めることができます。

人間は、筆算をするときに 10 進数を使いますが、コンピュータに筆算をさせるときは、2 進数が使われます。たとえば、2 進数の筆算で、

$$11010 \times 1001 \quad (26 \times 9)$$

という乗算をすると、次のようになります。

				1	1	0	1	0	(26)
×					1	0	0	1	(9)
				1	1	0	1	0	
			0	0	0	0	0		
		0	0	0	0	0			
	1	1	0	1	0				
	1	1	1	0	1	0	1	0	(234)

11.3.2 筆算による乗算のプログラム

それでは、筆算による乗算のプログラムを書いてみましょう。

プログラムの例 mulpape.cas

```
MULPAPE  START
         LAD   GR0,0
         LD    GR1,A
         LD    GR2,B
LOOP     JZE   EXIT
         LD    GR3,GR2
         AND   GR3,=#0001
         JZE   SKIP
         ADDL  GR0,GR1
```

```
SKIP     SLL   GR1,1
         SRL   GR2,1
         JUMP  LOOP
EXIT     ST    GR0,PRODUCT
         DMEM  MSG,PRODUCT,DUMPE
         RET
A        DC    26
B        DC    9
PRODUCT  DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、A 番地の整数と B 番地の整数とを筆算で乗算して、その結果を PRODUCT 番地に格納して、DMEM 命令で PRODUCT 番地を表示します。A 番地には 26、B 番地には 9 が格納されていますので、PRODUCT 番地には 234（16 進数では #00EA ）が格納されます。

このプログラムは、あらかじめ、GR0 に 0 を設定して、GR1 と GR2 にそれぞれの整数をロードしたのち、GR1 は左へ、GR2 は右へ、1 ビットずつシフトしながら、GR2 の最下位ビットが 1 ならば GR1 を GR0 に加算する、ということを繰り返します。このループは、GR2 が 0 になったときに終了します。

11.4　筆算による除算

11.4.1　筆算による除算の基礎

第 11.2 節で、減算を繰り返すことによって除算をする方法について説明しましたが、除算の方法は、それだけではありません。たとえば、筆算というのも、除算をする方法のひとつです。筆算による除算は、単純な減算の繰り返しよりも効率よく結果を求めることができます。

人間は、筆算をするときに 10 進数を使いますが、コンピュータに筆算をさせるときは、2 進数が使われます。たとえば、2 進数の筆算で、

$$110011 \div 1001 \quad (51 \div 9)$$

という除算をすると、次のようになります。

```
          1 0 1
      ------------
1 0 0 1 ) 1 1 0 0 1 1
          1 0 0 1
        -----------
            1 1 1
              0
          ---------
          1 1 1 1
          1 0 0 1
        -----------
            1 1 0
```

11.4.2　筆算による除算のプログラム

それでは、筆算による除算のプログラムを書いてみましょう。

プログラムの例　divpape.cas

```
DIVPAPE  START
         LD    GR1,A
         LD    GR2,B
         LAD   GR3,#0001
;
LOOP1    CPA   GR1,GR2
         JMI   EXIT
         SLL   GR2,1
         SLL   GR3,1
         JUMP  LOOP1
;
EXIT     LAD   GR4,0
;
LOOP2    CPA   GR1,GR2
         JMI   SKIP
         ADDA  GR4,GR3
         SUBA  GR1,GR2
```

```
SKIP     SRL   GR2,1
         SRL   GR3,1
         CPA   GR1,B
         JPL   LOOP2
         JZE   LOOP2
;
         ST    GR4,QUOTIENT
         ST    GR1,MOD
         DMEM  MSG,QUOTIENT,DUMPE
         RET
A        DC    51
B        DC    9
QUOTIENT DS    1
MOD      DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、A 番地の整数を B 番地の整数で除算して、その商を QUOTIENT 番地に格納して、その余りを MOD 番地に格納して、DMEM 命令で QUOTIENT 番地と MOD 番地を表示します。A 番地には 51、B 番地には 9 が格納されていますので、QUOTIENT 番地には 5 が格納されて、MOD 番地には 6 が格納されます。

このプログラムには二つのループがあります。LOOP1 から始まるループと、LOOP2 から始まるループです。

LOOP1 から始まるループは、割られる数の下の適切な位置に割る数を書くためのものです。割る数 (GR2) が割られる数 (GR1) よりも大きくなるまで、割る数 (GR2) を左へ 1 ビットづつシフトしていきます。そして、それと同時に、GR3 に設定された #0001 というデータも、左へ 1 ビットづつシフトしていきます。このデータは、商の適切な位置に 1 を書くために使われます。

LOOP2 から始まるループは、割る数 (GR2) と、商の適切な位置に 1 を書くためのデータ (GR3) とを、右へ 1 ビットずつシフトしながら、減算が可能なときだけ、商の適切な位置に 1 を書いて（GR4 に GR3 を加算します）、割られる数 (GR1) から割る数 (GR2) を減算します。このループは、割られる数 (GR1) よりも割る数（GR2 ではなくて、B 番地に格納されている本来の割る数）のほうが大きくなったときに終了します。

第 12 章　文字列処理

12.1　文字列処理の基礎

12.1.1　文字列処理の基本的な考え方

この章では、文字列を処理するプログラムについて説明したいと思います。

COMET II では、文字列は、1 語に 1 文字で、連続する主記憶装置の領域に格納されます。ですから、文字単位で文字列を処理する方法は、第 8.3 節で説明した、データ列を処理する方法と同じです。

12.1.2　文字列の逆転

まず最初に、とても簡単な文字列処理の例として、文字列を逆転させるプログラムを書いてみましょう。

プログラムの例　reverse.cas

```
REVERSE  START
         IN    INAREA,INLENGTH
         CALL  REVSUB
         OUT   OUTAREA,INLENGTH
         RET
;
REVSUB   LAD   GR2,0
         LD    GR3,INLENGTH
         LAD   GR3,-1,GR3
LOOP     LD    GR1,INAREA,GR2
         ST    GR1,OUTAREA,GR3
         LAD   GR2,1,GR2
         LAD   GR3,-1,GR3
         CPA   GR2,INLENGTH
```

```
          JMI   LOOP
          RET
;
INAREA    DS    256
INLENGTH  DS    1
OUTAREA   DS    256
          END
```

このプログラムは、文字列を読み込んで、その文字列を逆転させた文字列（文字を逆の順序に並べ替えた文字列）を出力します。

メインルーチンは、IN 命令を使って文字列を読み込んで、その文字列を INAREA から始まる領域に、その文字列の長さを INLENGTH に格納します。そして、REVSUB から始まるサブルーチンを呼び出して、そののち、OUTAREA から始まる領域に格納された文字列を、OUT 命令を使って出力します。

REVSUB から始まるサブルーチンは、GR2 を文字列の先頭の位置 (0) からインクリメントすると同時に、GR3 を文字列の末尾の位置（長さ − 1）からデクリメントしながら、INAREA から始まる領域に格納されている文字列の GR2 番目の文字を GR1 にロードして、OUTAREA から始まる領域の GR3 番目にそれをストアします。そして、GR2 の内容と INLENGTH の内容とが等しくなったとき、ループから脱出して終了します。

実行例

```
IN ? I am a cat.
.tac a ma I
```

12.1.3　文字の置き換え

次に、文字列の中に含まれているすべての空白をドットに置き換えるプログラムを書いてみましょう。

プログラムの例　sp2dot.cas

```
SP2DOT    START
          IN    INAREA,INLENGTH
          CALL  S2DSUB
          OUT   OUTAREA,INLENGTH
          RET
;
S2DSUB    LAD   GR2,0
LOOP      LD    GR1,INAREA,GR2
          CPL   GR1,=' '
          JPL   SKIP
          JMI   SKIP
          LD    GR1,='.'
SKIP      ST    GR1,OUTAREA,GR2
          LAD   GR2,1,GR2
          CPA   GR2,INLENGTH
          JMI   LOOP
          RET
;
INAREA    DS    256
INLENGTH  DS    1
OUTAREA   DS    256
          END
```

このプログラムは、文字列を読み込んで、その文字列に含まれているすべての空白をドットに置き換えた文字列を出力します。

S2DSUB から始まるサブルーチンは、GR2 を 0 からインクリメントしながら、INAREA から始まる領域に格納されている文字列の GR2 番目の文字（先頭は 0 番目）を GR1 にロードして、それが空白ならば GR1 にドットをロードして、GR1 の内容を OUTAREA から始まる領域の GR2 番目にストアします。そして、GR2 の内容と INLENGTH の内容とが等しくなったとき、ループから脱出して終了します。

実行例

```
IN ? dog  cat   mouse    rabbit     monkey
dog..cat...mouse....rabbit.....monkey
```

12.1.4 小文字から大文字への変換

英字には、小文字と大文字があります。そこで、文字列に含まれているすべての小文字を大文字に変換するプログラムを書いてみることにしましょう。

第 1.3.6 項で説明したように、COMET II では、文字は、日本工業規格が策定した JIS X 0201 という文字コードによって表現されます。

JIS X 0201 では、A、B、C、D、……という英字の大文字は、それと同じ順番で、65、66、67、68、……という連続した整数によってあらわされます。そして、a、b、c、d、……という英字の小文字は、それと同じ順番で、97、98、99、100、……という連続した整数によってあらわされます。ですから、大文字をあらわす整数と、それに対応する小文字をあらわす整数とのあいだには、常に 32 の差がある、ということになります。

プログラムの例 low2up.cas

```
LOW2UP   START
         IN    INAREA,INLENGTH
         CALL  L2USUB
         OUT   OUTAREA,INLENGTH
         RET
;
L2USUB   LAD   GR2,0
LOOP     LD    GR1,INAREA,GR2
         CALL  ISLOWER
         CPL   GR3,=0
         JZE   SKIP
         SUBL  GR1,=32
SKIP     ST    GR1,OUTAREA,GR2
         LAD   GR2,1,GR2
         CPA   GR2,INLENGTH
         JMI   LOOP
         RET
;
ISLOWER  LAD   GR3,0
         CPL   GR1,='a'
         JMI   EXIT
         CPL   GR1,='z'
         JPL   EXIT
         LAD   GR3,1
EXIT     RET
;
INAREA   DS    256
INLENGTH DS    1
OUTAREA  DS    256
         END
```

このプログラムは、文字列を読み込んで、その文字列に含まれているすべての英字の小文字を大文字に変換した文字列を出力します。

ISLOWER から始まるサブルーチンは、GR1 の内容が英字の小文字かどうかを判定して、GR3 に、英字の小文字ならば 1、そうでないならば 0 を設定します。

実行例

```
IN ? Arithmetic Logic Unit
ARITHMETIC LOGIC UNIT
```

12.1.5 大文字のみの取り出し

これまでに紹介した文字列処理の二つのプログラムは、どちらも、処理の対象となる文字列と処理した結果の文字列とが同じ長さになるものばかりでしたが、文字列処理は、文字列の長さが変化しないものばかりではありません。

文字列の長さが変化する文字列処理の例として、文字列から英字の大文字だけを取り出して並べることによって、大文字だけの文字列を作るプログラムを書いてみましょう。

プログラムの例 acronym.cas

```
ACRONYM  START
         IN    INAREA,INLENGTH
         CALL  ACROSUB
         OUT   OUTAREA,OUTLENGT
         RET
;
```

```
ACROSUB  LAD   GR2,0
         LAD   GR3,0
LOOP     LD    GR1,INAREA,GR2
         CALL  ISUPPER
         CPL   GR4,=0
         JZE   SKIP
         ST    GR1,OUTAREA,GR3
         LAD   GR3,1,GR3
SKIP     LAD   GR2,1,GR2
         CPA   GR2,INLENGTH
         JMI   LOOP
         ST    GR3,OUTLENGT
         RET
;
ISUPPER  LAD   GR4,0
         CPL   GR1,='A'
         JMI   EXIT
         CPL   GR1,='Z'
         JPL   EXIT
         LAD   GR4,1
EXIT     RET
;
INAREA   DS    256
INLENGTH DS    1
OUTAREA  DS    256
OUTLENGT DS    1
         END
```

このプログラムは、文字列を読み込んで、その文字列から英字の大文字だけを取り出して、それらの大文字から構成される文字列を出力します。

実行例

```
IN ? Arithmetic Logic Unit
ALU
```

12.2　文字列検索

12.2.1　文字列検索の基礎

文字列の一部分となっている文字列は、「部分文字列」(substring) と呼ばれます。

文字列の中に特定の部分文字列が含まれているかどうかを調べて、含まれている場合はその位置を求めるという処理は、「文字列検索」(string searching) と呼ばれます。

12.2.2　先頭の部分文字列

文字列検索のプログラムを書くためのウォーミングアップとして、文字列の先頭に特定の部分文字列が存在しているかどうかということを調べるプログラムを書いてみましょう。

プログラムの例　head.cas

```
HEAD     START
         IN    INAREA1,INLEN1
         IN    INAREA2,INLEN2
         CALL  HEADSUB
         DMEM  MSG,RESULT,DUMPE
         RET
;
HEADSUB  LD    GR1,INLEN1
         CPL   GR1,INLEN2
         JMI   NOTFOUND
         LAD   GR1,0
LOOP     LD    GR2,INAREA1,GR1
         CPL   GR2,INAREA2,GR1
         JPL   NOTFOUND
         JMI   NOTFOUND
         LAD   GR1,1,GR1
         CPA   GR1,INLEN2
         JMI   LOOP
         LAD   GR0,0
         JUMP  FOUND
NOTFOUND LAD   GR0,-1
FOUND    ST    GR0,RESULT
```

```
         RET
;
INAREA1  DS    256
INLEN1   DS    1
INAREA2  DS    256
INLEN2   DS    1
RESULT   DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、2 個の文字列を読み込んで、1 個目の文字列の先頭に 2 個目の文字列が含まれているならば 0 を、そうでないならば −1 を RESULT 番地にストアして、DMEM 命令で RESULT 番地を表示します。

1 個目の文字列の先頭に 2 個目の文字列が含まれているかどうかを実際に調べるという処理をするのは、HEADSUB から始まるサブルーチンです。

このサブルーチンは、まず、1 個目の文字列の長さと 2 個目の文字列の長さとを比べます。2 個目のほうが長いならば、1 個目の中に 2 個目が含まれている可能性はありませんので、−1 を RESULT 番地にストアします。

1 個目と 2 個目とが同じ長さか、または 1 個目のほうが 2 個目よりも長いならば、先頭から順番に文字を比較するループに入ります。そして、もしも一致しない文字が見つかったならば、ループから抜けて −1 を RESULT 番地にストアします。文字が一致したまま 2 個目の最後まで文字の比較が終わった場合は、ループから抜けて 0 を RESULT 番地にストアします。

実行例

```
IN ? assembler
IN ? assem
* MESSAGE1 *    Start:022E End:022F
 0228: ---- ---- ---- ---- ---- ---- 0000 FFFF        ..

IN ? assembler
IN ? assym
* MESSAGE1 *    Start:022E End:022F
 0228: ---- ---- ---- ---- ---- ---- FFFF FFFF        ..

IN ? assembler
IN ? assemblers
* MESSAGE1 *    Start:022E End:022F
 0228: ---- ---- ---- ---- ---- ---- FFFF FFFF        ..
```

12.2.3 文字の検索

文字列検索のプログラムを書くためのウォーミングアップとして、もう一つ、プログラムを書いてみましょう。今度は、文字列の中で 1 個の文字を検索するプログラムです。

プログラムの例 charsch.cas

```
CHARSCH  START
         IN    INAREA1,INLEN1
         IN    INAREA2,INLEN2
         CALL  CHARSCHS
         DMEM  MSG,RESULT,DUMPE
         RET
;
CHARSCHS LD    GR1,INLEN2
         CPA   GR1,=1
         JMI   NOTFOUND
         LAD   GR1,0
LOOP     LD    GR2,INAREA1,GR1
         CPL   GR2,INAREA2
         JZE   FOUND
         LAD   GR1,1,GR1
         CPA   GR1,INLEN1
         JMI   LOOP
NOTFOUND LAD   GR1,-1
FOUND    ST    GR1,RESULT
         RET
;
INAREA1  DS    256
INLEN1   DS    1
```

```
INAREA2  DS    256
INLEN2   DS    1
RESULT   DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、2 個の文字列を読み込んで、1 個目の文字列の中に 2 個目の文字列の先頭の文字が含まれているならばその位置（先頭の文字を 0 番目と数えます）を、そうでないならば −1 を RESULT 番地にストアして、 DMEM 命令で RESULT 番地を表示します。

1 個目の文字列の中に 2 個目の文字列の先頭の文字が含まれているかどうかを実際に調べるという処理をするのは、 CHARSCHS から始まるサブルーチンです。

このサブルーチンは、まず、2 個目の文字列の長さを調べて、それが 0 ならば、−1 を RESULT 番地にストアします。

2 個目の長さが 1 以上ならば、先頭から順番に文字を比較するループに入ります。そして、もしも一致する文字が見つかったならば、ループから抜けて、見つかった文字の位置を RESULT 番地にストアします。文字が一致しないまま 1 個目の最後まで文字の比較が終わった場合は、ループから抜けて −1 を RESULT 番地にストアします。

実行例

```
IN ? assembler
IN ? mouse
* MESSAGE1 *    Start:0228 End:0229
 0228: 0004 FFFF ---- ---- ---- ---- ---- ----  ..

IN ? assembler
IN ? cat
* MESSAGE1 *    Start:0228 End:0229
 0228: FFFF FFFF ---- ---- ---- ---- ---- ----  ..
```

12.2.4　文字列検索のプログラム

それでは、文字列検索のプログラムを書いてみましょう。

文字列検索のプログラムは、ウォーミングアップとして書いた二つのプログラムを組み合わせることによって書くことができます。

プログラムの例　search.cas

```
SEARCH   START
         IN    INAREA1,INLEN1
         IN    INAREA2,INLEN2
         CALL  SEARCHS
         DMEM  MSG,RESULT,DUMPE
         RET
;
SEARCHS  LD    GR1,INLEN2
         CPA   GR1,=1
         JMI   NOTFD1
         LD    GR1,INLEN1
         SUBL  GR1,INLEN2
         JMI   NOTFD1
         LAD   GR1,1,GR1
         ST    GR1,INLEN3
         LAD   GR1,0
LOOP1    LD    GR2,INAREA1,GR1
         CPL   GR2,INAREA2
         JPL   SKIP
         JMI   SKIP
         CALL  HEADSUB
         CPL   GR3,=0
         JZE   FOUND1
SKIP     LAD   GR1,1,GR1
         CPA   GR1,INLEN3
         JMI   LOOP1
NOTFD1   LAD   GR1,-1
FOUND1   ST    GR1,RESULT
         RET
;
HEADSUB  LD    GR4,INLEN2
         CPL   GR4,=2
```

```
          JMI   FOUND2
          ADDL  GR4,GR1
          ADDL  GR4,INLEN2
          CPL   GR4,INLEN1
          JPL   NOTFD2
          LAD   GR4,1,GR1
          LAD   GR5,1
LOOP2     LD    GR6,INAREA1,GR4
          CPL   GR6,INAREA2,GR5
          JPL   NOTFD2
          JMI   NOTFD2
          LAD   GR4,1,GR4
          LAD   GR5,1,GR5
          CPA   GR5,INLEN2
          JMI   LOOP2
FOUND2    LAD   GR3,0
          RET
NOTFD2    LAD   GR3,-1
          RET
;
INAREA1   DS    256
INLEN1    DS    1
INAREA2   DS    256
INLEN2    DS    1
INLEN3    DS    1
RESULT    DS    1
DUMPE     DC    -1
MSG       DC    'MESSAGE1'
          END
```

このプログラムは、2 個の文字列を読み込んで、1 個目の文字列の中に 2 個目の文字列が含まれているならばその位置（先頭の文字を 0 番目と数えます）を、そうでないならば −1 を RESULT 番地にストアして、 DMEM 命令で RESULT 番地を表示します。

1 個目の文字列の中に 2 個目の文字列が含まれているかどうかを実際に調べるという処理をするのは、 SEARCHS から始まるサブルーチンです。

このサブルーチンは、まず、2 個目の文字列の長さを調べて、それが 0 ならば、−1 を RESULT 番地にストアします。さらに、1 個目の文字列の長さと 2 個目の文字列の長さを比較して、2 個目のほうが長いならば、−1 を RESULT 番地にストアします。

2 個目の長さが 1 以上で、1 個目が 2 個目よりも短くなければ、先頭から順番に、1 個目の中の文字と 2 個目の先頭の文字とを比較するループに入ります。そして、もしも一致する文字が見つかったならば、 HEADSUB から始まるサブルーチンを呼び出します。このサブルーチンは、一致した文字から始まる 1 個目の文字列の部分文字列の先頭に 2 個目の文字列が含まれているかどうかを調べて、含まれているならば 0 を、そうでないならば −1 を GR3 に設定しますので、サブルーチンから戻ってきたのちに GR3 の内容を調べて、それが 0 ならば、見つかった部分文字列の位置を RESULT にストアして、0 ではないならば検索を続行します。

HEADSUB から始まるサブルーチンは、まず、1 個目の文字列の中で見つかった文字の右側に、2 個目の文字列と一致する十分な長さが残っているかどうかを調べます。もしも十分な長さが残っていないならば、−1 を RESULT 番地にストアします。

十分な長さが残っているならば、一致した文字の右側と、2 個目の文字列とを比較するループに入ります。そして、もしも一致しない文字が見つかったならば、ループから抜けて −1 を GR3 に設定します。文字が一致したまま 2 個目の最後まで文字の比較が終わった場合は、ループから抜けて 0 を GR3 番地に設定します。

12.3 整数から文字列への変換

12.3.1 整数から 2 進数の文字列への変換

この節では、ビット列によって表現されている整数を文字列に変換するという処理について説明したいと思います。

まず最初に、整数を 2 進数の文字列に変換するプログラムを書いてみたいと思います。

COMET II の内部では、整数は 2 進数で表現されていますので、それをそのまま 2 進数の文字列に変換するというのは、とても簡単です。

プログラムの例 int2bin.cas

```
INT2BIN  START
         LD    GR0,INTEGER
         LAD   GR1,15
LOOP     SRL   GR0,1
         JOV   SKIP1
         LD    GR2,='0'
         JUMP  SKIP2
SKIP1    LD    GR2,='1'
SKIP2    ST    GR2,BINARY,GR1
         LAD   GR1,-1,GR1
         CPA   GR1,=-1
         JPL   LOOP
         OUT   BINARY,LENGTH
         RET
;
INTEGER  DC    #39CE
BINARY   DS    16
LENGTH   DC    16
         END
```

このプログラムは、INTEGER 番地に格納されている、#39CE という整数を 2 進数の文字列に変換して、その結果を出力します。実行すると、

```
0011100111001110
```

と出力されるはずです。

変換の結果は、BINARY から始まる領域に格納されます。GR1 は、文字をストアする位置を示すインデックスレジスターとして使われています。0 をストアするか 1 をストアするかということは、GR0 にロードされた整数を右へ 1 ビットだけ論理シフトすることによって判断しています。シフト命令は、最後に送り出されたビットをオーバーフローフラグに設定しますので、その直後に JOV 命令を実行すると、1 が送り出された場合は分岐して、0 が送り出された場合は分岐しません。

12.3.2　整数から 16 進数の文字列への変換

次に、整数を 16 進数の文字列に変換するプログラムを書いてみましょう。

プログラムの例　int2hex.cas

```
INT2HEX  START
         LD    GR0,INTEGER
         LAD   GR1,3
LOOP     LD    GR2,GR0
         AND   GR2,=#000F
         CPL   GR2,=10
         JMI   SKIP1
         ADDL  GR2,=55
         JUMP  SKIP2
SKIP1    ADDL  GR2,=48
SKIP2    ST    GR2,HEXDEC,GR1
         SRL   GR0,4
         LAD   GR1,-1,GR1
         CPA   GR1,=-1
         JPL   LOOP
         OUT   HEXDEC,LENGTH
         RET
;
INTEGER  DC    #39CE
HEXDEC   DS    4
LENGTH   DC    4
         END
```

このプログラムは、INTEGER 番地に格納されている、#39CE という整数を 16 進数の文字列に変換して、その結果を出力します。実行すると、

```
39CE
```

と出力されるはずです。

整数を 16 進数の文字列に変換するための考え方は、2 進数の文字列へ変換する場合と、基本的には同じです。16 進数というのは、2 進数を 4 桁ずつ区切って、それぞれの 4 桁を 1 文字で表記したもののことです。したがって、2 進数の文字列への変換では 1 ビットずつ右シフトしてい

たところを、4 ビットずつ右シフトして、右端の 4 桁を文字に変換していけばいいということになります。

2 進数の文字列への変換では、シフト命令によって最後に送り出されたビットがオーバーフローフラグに設定されることを利用して、文字が 0 なのか 1 なのかということを判断したわけですが、16 進数の文字列への変換では、この方法は使えません。そこで、このプログラムでは、

```
0000 0000 0000 1111
```

というマスクとの論理積を求めることによって、右端の 4 ビットを取り出しています。

JIS X 0201 では、0、1、2、3、……という文字は、それと同じ順番で、48、49、50、51、……という連続した整数によってあらわされます。ですから、0 から 9 までの整数は、それに 48 を加算することによって、文字に変換することができます。

16 進数では、桁の数値が 10 から 15 までの場合、それは、A、B、C、D、E、F という英字によって表記されます。JIS X 0201 では、A、B、C、D、……という英字の大文字は、それと同じ順番で、65、66、67、68、……という連続した整数によってあらわされますので、10 から 15 までの整数は、それに 55 を加算することによって、文字に変換することができます。

12.3.3 整数から 10 進数の文字列への変換

次に、整数を 10 進数の文字列に変換するプログラムを書いてみましょう。

プログラムの例 int2dec.cas

```
INT2DEC  START
         LD    GR0,INTEGER
         LAD   GR1,0
LOOP1    CALL  DIVIDE
         ADDL  GR2,=48
         ST    GR2,DECIMAL,GR1
         LAD   GR1,1,GR1
         CPA   GR1,=4
         JMI   LOOP1
         ADDL  GR0,=48
         ST    GR0,DECIMAL,GR1
         OUT   DECIMAL,LENGTH
         RET
;
DIVIDE   LAD   GR2,0
LOOP2    CPA   GR0,WEIGHT,GR1
         JMI   EXIT
         LAD   GR2,1,GR2
         SUBL  GR0,WEIGHT,GR1
         JUMP  LOOP2
EXIT     RET
;
INTEGER  DC    #39CE
WEIGHT   DC    10000,1000,100,10
DECIMAL  DS    5
LENGTH   DC    5
         END
```

このプログラムは、INTEGER 番地に格納されている、#39CE という整数を 10 進数の文字列に変換して、その結果を出力します。実行すると、

```
14798
```

と出力されるはずです。

このプログラムの考え方は、上で紹介した、整数を 2 進数や 16 進数の文字列に変換するプログラムの考え方とは、かなり異なっています。

整数をあらわす 10 進数のそれぞれの桁は、右から左へ順番に、10^0、10^1、10^2、……という重みを持っています。そこで、WEIGHT から始まる 4 語の領域のそれぞれに、10^4、10^3、10^2、10^1 という、10 進数の桁の重みを設定しておきます。

LOOP1 から始まるループは、10^4 の桁から順番に、桁の文字を求めて、それを DECIMAL から始まる領域にストアする、ということを繰り返します。そして、ループから脱出したのち、10^0 の桁の文字をストアします。

DIVIDE から始まるサブルーチンは、個々の桁があらわしている整数を求めます。GR0 から桁の重みを減算していって、減算の回数を GR2 に求めます。

12.4　文字列から整数への変換

12.4.1　2 進数の文字列から整数への変換

この節では、整数を表現している文字列を解釈して、その整数を表現するビット列を求めるという処理について説明したいと思います。

まず最初に、2 進数の文字列を整数に変換するプログラムを書いてみたいと思います。

プログラムの例　bin2int.cas

```
BIN2INT  START
         IN     INAREA,INLENGTH
         CALL   B2ISUB
         DMEM   MSG,INTEGER,DUMPE
         RET
;
B2ISUB   LAD    GR0,0
         LAD    GR1,0
LOOP     SLL    GR0,1
         LD     GR2,INAREA,GR1
         CPL    GR2,='0'
         JZE    SKIP
         OR     GR0,=#0001
SKIP     LAD    GR1,1,GR1
         CPL    GR1,INLENGTH
         JMI    LOOP
         ST     GR0,INTEGER
         RET
;
INAREA   DS     256
INLENGTH DS     1
INTEGER  DS     1
DUMPE    DC     -1
MSG      DC     'MESSAGE1'
         END
```

このプログラムは、文字列を読み込んで、その文字列を 2 進数として解釈したときにそれが表現している整数を求めて、その整数が格納されている主記憶装置の語を DMEM 命令で表示します。

B2ISUB から始まるサブルーチンは、 INAREA から始まる領域に格納されている 2 進数を解釈して、それが表現している整数を INTEGER に格納します。

LOOP から始まるループは、 GR0 を左へ 1 ビットずつ論理シフトしながら、2 進数を構成しているそれぞれの文字について、それが 0 ではないならば、 GR0 と #0001 との論理和を GR0 に設定する、ということを繰り返します（したがって、 0 以外の文字は、すべて 1 と解釈されます）。

実行例

```
IN ? 101111010110
* MESSAGE1 *    Start:0122 End:0123
 0120: ---- ---- 0BD6 FFFF ---- ---- ---- ----    ..
```

12.4.2　16 進数の文字列から整数への変換

次に、16 進数の文字列を整数に変換するプログラムを書いてみましょう。

プログラムの例　hex2int.cas

```
HEX2INT  START
         IN     INAREA,INLENGTH
         CALL   H2ISUB
         DMEM   MSG,INTEGER,DUMPE
         RET
;
H2ISUB   LAD    GR0,0
         LAD    GR1,0
LOOP     SLL    GR0,4
         LD     GR2,INAREA,GR1
         CALL   HEXVAL
         OR     GR0,GR2
         LAD    GR1,1,GR1
         CPL    GR1,INLENGTH
         JMI    LOOP
         ST     GR0,INTEGER
         RET
```

```
;
HEXVAL   CPL   GR2,='A'
         JMI   SKIP1
         SUBL  GR2,=55
         JUMP  SKIP2
SKIP1    SUBL  GR2,=48
SKIP2    RET
;
INAREA   DS    256
INLENGTH DS    1
INTEGER  DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、文字列を読み込んで、その文字列を 16 進数として解釈したときにそれが表現している整数を求めて、その整数が格納されている主記憶装置の語を DMEM 命令で表示します。

H2ISUB から始まるサブルーチンは、 INAREA から始まる領域に格納されている 16 進数を解釈して、それが表現している整数を INTEGER に格納します。

LOOP から始まるループは、 GR0 を左へ 4 ビットずつ論理シフトしながら、16 進数を構成しているそれぞれの文字について、それが表現している整数と GR0 との論理和を GR0 に設定する、ということを繰り返します

HEXVAL から始まるサブルーチンは、 GR2 に格納されている文字を、16 進数の数字と解釈して、それが表現している整数を GR2 に設定します。

実行例

```
IN ? BD6
* MESSAGE1 *    Start:012A End:012B
 0128: ---- ---- 0BD6 FFFF ---- ---- ---- ----    ..
```

12.4.3 10 進数の文字列から整数への変換

次に、10 進数の文字列を整数に変換するプログラムを書いてみましょう。

プログラムの例 dec2int.cas

```
DEC2INT  START
         IN    INAREA,INLENGTH
         CALL  D2ISUB
         DMEM  MSG,INTEGER,DUMPE
         RET
;
D2ISUB   LAD   GR0,0
         LAD   GR1,0
LOOP1    CALL  TENTIMES
         LD    GR2,INAREA,GR1
         SUBL  GR2,=48
         ADDL  GR0,GR2
         LAD   GR1,1,GR1
         CPL   GR1,INLENGTH
         JMI   LOOP1
         ST    GR0,INTEGER
         RET
;
TENTIMES LAD   GR3,0
         LD    GR4,GR0
LOOP2    ADDL  GR0,GR4
         LAD   GR3,1,GR3
         CPL   GR3,=9
         JMI   LOOP2
         RET
;
INAREA   DS    256
INLENGTH DS    1
INTEGER  DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、文字列を読み込んで、その文字列を 10 進数として解釈したときにそれが表現している整数を求めて、その整数が格納されている主記憶装置の語を DMEM 命令で表示します。

D2ISUB から始まるサブルーチンは、 INAREA から始まる領域に格納されている 10 進数を解釈して、それが表現している整数を INTEGER に格納します。

LOOP から始まるループは、 GR0 を 10 倍しながら、10 進数を構成しているそれぞれの文字について、それが表現している整数と GR0 とを加算した結果を GR0 に設定する、ということを繰り返します

TENTIMES から始まるサブルーチンは、 GR0 に格納されている整数を 10 倍した結果を GR0 に設定します。

実行例

```
IN ? 3030
* MESSAGE1 *    Start:012A End:012B
 0128: ---- ---- 0BD6 FFFF ---- ---- ---- ----    ..
```

第 13 章　再帰

13.1　再帰の基礎

13.1.1　再帰とは何か

この章では、「再帰」(recursion) と呼ばれるものを使ったプログラムの書き方について説明したいと思います。

再帰というのは、全体と同じものが一部分として含まれているという性質のことです。再帰という性質を持っているものは、「再帰的な」(recursive) と形容されます。

ここに、1 台のカメラと 1 台のモニターがあるとします。まず、それらを接続して、カメラで撮影した映像がモニターに映し出されるようにします。そして次に、カメラをモニターの画面に向けます。すると、モニターの画面には、そのモニター自身が映し出されることになります。そして、映し出されたモニターの画面の中には、さらにモニター自身が映し出されています。このときにモニターの画面に映し出されるのは、再帰という性質を持っている映像、つまり再帰的な映像です。

また、先祖と子孫の関係も再帰的です。なぜなら、先祖と子孫との中間にいる人々も、やはり先祖と子孫の関係で結ばれているからです。

13.1.2　基底

再帰という性質を持っているものは、全体と同じものが一部分として含まれているわけですが、その構造は、内部に向かってどこまでも続いている場合もあれば、どこかで終わっている場合もあります。

再帰的な構造がどこかで終わっている場合、その中心には、その内部に再帰的な構造を持っていない何かがあります。そのような、再帰的な構造の中心にあって、その内部に再帰的な構造を持っていないものは、その再帰的な構造の「基底」(basis) と呼ばれます。

先祖と子孫の関係では、親子関係というのが、その再帰的な構造の基底になります。

13.1.3　再帰的なサブルーチン

サブルーチンは、再帰的に書くことが可能です。サブルーチンを再帰的に書くというのは、自分の中で自分自身を呼び出すサブルーチンを書くということです。再帰的な構造を持っている概念を取り扱うサブルーチンは、再帰的に書くほうが、再帰的ではない方法で書くよりもすっきりした記述になります。

サブルーチンを再帰的に書く場合は、基底に到達しているかどうかを判断して、もしも基底に到達していた場合は、自分自身を呼び出さないで終了する、ということが必要になります。基底に到達している場合も自分自身を呼び出すように書いたとすると、そのサブルーチンは、無限に自分自身を呼び出し続けることになってしまいます。

13.1.4 再帰とスタック

サブルーチンを再帰的に書くというのは、自分自身を呼び出すサブルーチンを書くということです。サブルーチンが自分自身を呼び出した場合、呼び出されたサブルーチンは、必然的に、自分が使っているものと同じ汎用レジスタを使うことになります。したがって、自分自身を呼び出したのちに、それ以前の汎用レジスタの内容が必要になる場合には、第 9.4.4 項で説明したように、自分自身を呼び出す前に、自分が使っている汎用レジスタの内容をスタックにプッシュしておいて、自分自身を呼び出したのちにそれをポップする必要があります。

それでは、再帰的なサブルーチンを実際に書いてみましょう。

プログラムの例 recurse.cas

```
RECURSE  START
         LAD   GR1,4
         CALL  RECSUB
         RET
;
RECSUB   DREG  MSG
         CPL   GR1,=0
         JZE   BASIS
         PUSH  0,GR1
         LAD   GR1,-1,GR1
         CALL  RECSUB
         POP   GR1
BASIS    DREG  MSG
         RET
;
MSG      DC    'MESSAGE1'
         END
```

このプログラムのメインルーチンは、GR1 に 4 という整数を設定したのち、RECSUB から始まるサブルーチンを呼び出します。

RECSUB から始まるサブルーチンは、RECSUB から始まるサブルーチン（つまり自分自身）を自分の中で呼び出していますので、再帰的なサブルーチンです。

RECSUB は、GR1 の内容をスタックにプッシュしたのち、GR1 をデクリメントして、自分自身を呼び出します。そして、呼び出した自分自身の動作が終了したのち、スタックからポップしたデータで GR1 を復元します。ただし、GR1 の内容が 0 の場合は、何もしないで終了します。したがって、GR1 の内容が 0 の場合というのが、この再帰の基底になります。

RECSUB は、呼び出された直後と終了する直前に、DREG 命令でレジスタとスタックの内容を表示します。それを見ると、GR1 の内容は、次のように変化しています。

```
0004 0003 0002 0001 0000 0000 0001 0002 0003 0004
```

前半の 0004 から 0000 までは、呼び出された直後の DREG 命令が表示したもので、後半の 0000 から 0004 までは、終了する直前の DREG 命令が表示したものです。このように、GR1 の内容は、再帰が進むにつれて減っていって、再帰から戻るにつれて増えていきます。

13.2 再帰による乗算

13.2.1 再帰による乗算の基礎

第 11 章で、乗算をするプログラムについて説明しましたが、乗算をするプログラムは、再帰を使って書くことも可能です。

乗算という概念は、再帰的な構造を持っています。なぜなら、乗算の定義は、

$$\begin{cases} & a \times 0 = 0 \\ b \geq 1 \text{ ならば} & a \times b = a + (a \times (b-1)) \end{cases}$$

というように、再帰的に書くこともできるからです。

13.2.2 再帰による乗算のプログラム

それでは、再帰による乗算のプログラムを書いてみましょう。

プログラムの例 mulrecu.js

```
MULRECU  START
         LD    GR1,A
```

```
         LD    GR2,B
         CALL  MULSUB
         ST    GR0,PRODUCT
         DMEM  MSG,PRODUCT,DUMPE
         RET
;
MULSUB   CPL   GR2,=0
         JZE   BASIS
         PUSH  0,GR2
         LAD   GR2,-1,GR2
         CALL  MULSUB
         POP   GR2
         ADDL  GR0,GR1
         RET
BASIS    LAD   GR0,0
         RET
;
A        DC    26
B        DC    9
PRODUCT  DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、A 番地の整数と B 番地の整数とを乗算して、その結果を PRODUCT 番地に格納して、DMEM 命令で PRODUCT 番地を表示します。A 番地には 26、B 番地には 9 が格納されていますので、PRODUCT 番地には 234（16 進数では #00EA ）が格納されます。

このプログラムのメインルーチンは、乗算の対象となる整数を GR1 と GR2 のそれぞれにロードしたのち、MULSUB から始まるサブルーチンを呼び出します。

MULSUB から始まるサブルーチンは、再帰を使って、GR1 と GR2 を乗算します。計算の結果は GR0 に格納されます。

13.3　階乗

13.3.1　階乗の基礎

n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果は、n の「階乗」(factorial) と呼ばれて、$n!$ と書きあらわされます。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

階乗という概念は、再帰的な構造を持っています。なぜなら、階乗は、

$$\begin{cases} 0! = 1 \\ n \geq 1 \text{ ならば } \quad n! = n \times (n-1)! \end{cases}$$

というように再帰的に定義することができるからです。

13.3.2　再帰による階乗のプログラム

それでは、再帰を使って階乗を求めるプログラムを書いてみましょう。

プログラムの例　fact.cas

```
FACT     START
         LD    GR1,N
         CALL  FACTSUB
         ST    GR0,FACTORI
         DMEM  MSG,FACTORI,DUMPE
         RET
;
FACTSUB  CPL   GR1,=0
         JZE   BASIS
         PUSH  0,GR1
```

```
          LAD   GR1,-1,GR1
          CALL  FACTSUB
          POP   GR1
          CALL  MULTIPLY
          RET
BASIS     LAD   GR0,1
          RET
;
MULTIPLY  LAD   GR3,0
          LD    GR2,GR0
LOOP      CPL   GR2,=0
          JZE   EXIT
          ADDL  GR3,GR1
          LAD   GR2,-1,GR2
          JUMP  LOOP
EXIT      LD    GR0,GR3
          RET
;
N         DC    5
FACTORI   DS    1
DUMPE     DC    -1
MSG       DC    'MESSAGE1'
          END
```

このプログラムは、N 番地の整数の階乗を求めて、その結果を FACTORI 番地に格納して、DMEM 命令で FACTORI 番地を表示します。N 番地には 5 が格納されていますので、FACTORI 番地には 120（16 進数では #0078 ）が格納されます。

このプログラムのメインルーチンは、階乗を計算する対象となる整数を GR1 にロードしたのち、FACTSUB から始まるサブルーチンを呼び出します。

FACTSUB から始まるサブルーチンは、再帰を使って、GR1 の階乗を計算します。計算の結果は GR0 に格納されます。

MULTIPLY から始まるサブルーチンは、GR0 と GR1 を乗算します。計算の結果は GR0 に格納されます。

13.4 フィボナッチ数列

13.4.1 フィボナッチ数列の基礎

第 0 項が 0、第 1 項が 1 で、第 2 項以降はその直前の 2 項を足し算した結果である、という数列は、「フィボナッチ数列」(Fibonacci sequence) と呼ばれます。フィボナッチ数列の第 0 項から第 14 項までを表にすると、次のようになります。

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
第 n 項	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377

フィボナッチ数列というのは再帰的な構造を持っている概念ですので、その第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 0 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

というように再帰的に定義することができます。

13.4.2 再帰によるフィボナッチ数列のプログラム

それでは、再帰を使ってフィボナッチ数列の項を求めるプログラムを書いてみましょう。

プログラムの例 fibona.cas

```
FIBONA    START
          LD    GR1,N
          CALL  FIBOSUB
          ST    GR2,NTHTERM
          DMEM  MSG,NTHTERM,DUMPE
          RET
;
FIBOSUB   CPL   GR1,=0
          JZE   BASIS0
```

```
          CPL   GR1,=1
          JZE   BASIS1
          PUSH  0,GR1
          LAD   GR1,-1,GR1
          CALL  FIBOSUB
          PUSH  0,GR2
          LAD   GR1,-1,GR1
          CALL  FIBOSUB
          LD    GR3,GR2
          POP   GR2
          POP   GR1
          ADDL  GR2,GR3
          RET
BASIS0    LAD   GR2,0
          RET
BASIS1    LAD   GR2,1
          RET
;
N         DC    14
NTHTERM   DS    1
DUMPE     DC    -1
MSG       DC    'MESSAGE1'
          END
```

このプログラムは、N 番地の整数を項数とするフィボナッチ数列の項を求めて、NTHTERM 番地にその結果を格納して、DMEM 命令で NTHTERM 番地を表示します。N 番地には 14 が格納されていますので、NTHTERM 番地には 377（16 進数では #0179）が格納されます。

このプログラムのメインルーチンは、求めるべきフィボナッチ数列の項数を GR1 にロードしたのち、FIBOSUB から始まるサブルーチンを呼び出します。

FIBOSUB から始まるサブルーチンは、再帰を使って、GR1 を項数とするフィボナッチ数列の項を求めます。計算の結果は GR2 に格納されます。

13.5　最大公約数

13.5.1　最大公約数の基礎

n と m がプラスの整数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(greatest common measure, GCM) と呼びます。たとえば、54 と 36 の最大公約数は 18 です。

n と m の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる次のような再帰的な手順を実行することによって求めることができます。

- n と m が等しいならば、n が、n と m の最大公約数である。
- n が m よりも大きいならば、n から m を減算した結果を d とする。そして、m と d の最大公約数を求めれば、その結果が n と m の最大公約数である。
- m が n よりも大きいならば、m から n を減算した結果を d とする。そして、n と d の最大公約数を求めれば、その結果が n と m の最大公約数である。

13.5.2　再帰による最大公約数のプログラム

それでは、再帰を使って最大公約数を求めるプログラムを書いてみましょう。

プログラムの例　gcm.cas

```
GCM       START
          LD    GR1,N
          LD    GR2,M
          CALL  GCMSUB
          ST    GR3,GCMRES
          DMEM  MSG,GCMRES,DUMPE
          RET
;
GCMSUB    CPL   GR1,GR2
          JZE   BASIS
          JMI   SKIP1
          SUBL  GR1,GR2
          JUMP  SKIP2
SKIP1     SUBL  GR2,GR1
```

```
SKIP2    CALL  GCMSUB
         RET
BASIS    LD    GR3,GR1
         RET
;
N        DC    54
M        DC    36
GCMRES   DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このプログラムは、N 番地の整数と M 番地の整数の最大公約数を求めて、GCMRES 番地にその結果を格納して、DMEM 命令で GCMRES 番地を表示します。N 番地には 54、M 番地には 36 が格納されていますので、GCMRES 番地には 18（16 進数では #0012）が格納されます。

このプログラムのメインルーチンは、最大公約数を求めるべき整数を GR1 と GR2 にロードしたのち、GCMSUB から始まるサブルーチンを呼び出します。

GCMSUB から始まるサブルーチンは、再帰を使って、GR1 と GR2 の最大公約数を求めます。計算の結果は GR3 に格納されます。このサブルーチンの場合、自分自身の呼び出しから戻ってきたあとは汎用レジスタを使っていませんので、スタックを使って汎用レジスタの内容を退避させておく必要はありません。

13.6 再帰による整数から 10 進数への変換

13.6.1 再帰による整数から 10 進数への変換の基礎

第 12.3 節で、整数を文字列に変換するプログラムについて説明しましたが、そのようなプログラムは、再帰を使って書くことも可能です。そこで、この節では、再帰を使って整数を 10 進数に変換するプログラムについて説明したいと思います。

n が整数だとすると、n をあらわす 10 進数は、

n を 10 で除算した商をあらわす 10 進数	n を 10 で除算した余りをあらわす数字

というように、再帰的な構造を持っています。この再帰の基底は n が 0 の場合で、その場合、n をあらわす 10 進数は空文字列（長さが 0 の文字列）になります。ただし、n が最初から 0 の場合は、それを、空文字列ではなく 0 であらわす必要があります。

13.6.2 再帰による整数から 10 進数への変換のプログラム

それでは、再帰を使って整数を 10 進数に変換するプログラムを書いてみましょう。

プログラムの例 inttod.cas

```
INTTOD   START
         LD    GR1,INTEGER
         LAD   GR2,4
         CALL  I2DSUB
         OUT   DECIMAL,LENGTH
         RET
;
I2DSUB   CPL   GR1,=0
         JZE   BASIS
         CALL  DIVTEN
         ADDL  GR3,=48
         ST    GR3,DECIMAL,GR2
         LAD   GR2,-1,GR2
         CALL  I2DSUB
BASIS    RET
;
DIVTEN   LAD   GR4,0
LOOP     CPA   GR1,=10
         JMI   EXIT
         LAD   GR4,1,GR4
         SUBL  GR1,=10
         JUMP  LOOP
EXIT     LD    GR3,GR1
         LD    GR1,GR4
         RET
```

```
;
INTEGER  DC    #39CE
DECIMAL  DC    '00000'
LENGTH   DC    5
         END
```

このプログラムは、INTEGER 番地に格納されている、#39CE という整数を 10 進数の文字列に変換して、その結果を出力します。実行すると、

```
14798
```

と出力されるはずです。

このプログラムのメインルーチンは、10 進数に変換する整数を GR1 にロードして、GR2 に 4 を設定したのち、I2DSUB から始まるサブルーチンを呼び出します。そうすると、10 進数に変換した結果が DECIMAL から始まる領域に格納されますので、それを OUT 命令で出力します。GR2 は、DECIMAL から始まる領域の何番目の語に数字をストアするか、ということを示します。

I2DSUB から始まるサブルーチンは、再帰を使って、GR1 を 10 進数に変換します。

DIVTEN から始まるサブルーチンは、GR1 を 10 で除算します。商は GR1 に格納されて、余りは GR3 に格納されます。

13.7　再帰による 10 進数から整数への変換

13.7.1　再帰による 10 進数から整数への変換の基礎

整数を文字列に変換するプログラムだけではなくて、文字列を整数に変換するプログラム、つまり、整数を表現している文字列を解釈して、その整数を表現するビット列を求めるプログラムも、再帰を使って書くことが可能です。そこで、この節では、再帰を使って 10 進数を整数に変換するプログラムについて説明したいと思います。

10 進数がどのような整数をあらわしているかということは、

右端の桁を除いた 10 進数の意味 $\times$ 10 + 右端の桁の意味

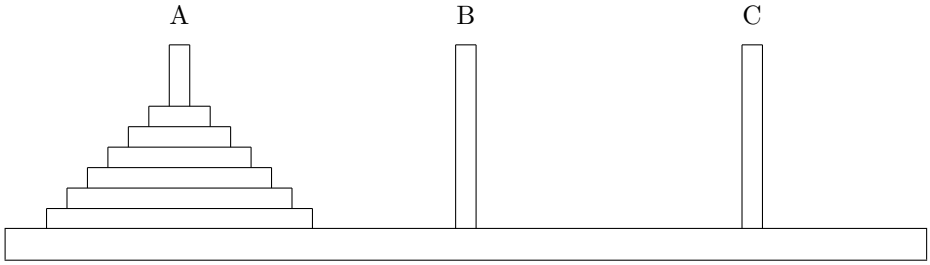
というように、再帰的に解釈することができます。この再帰の基底は 10 進数が空文字列の場合で、その場合、その 10 進数は 0 を意味していると解釈されます。

13.7.2　再帰による 10 進数から整数への変換のプログラム

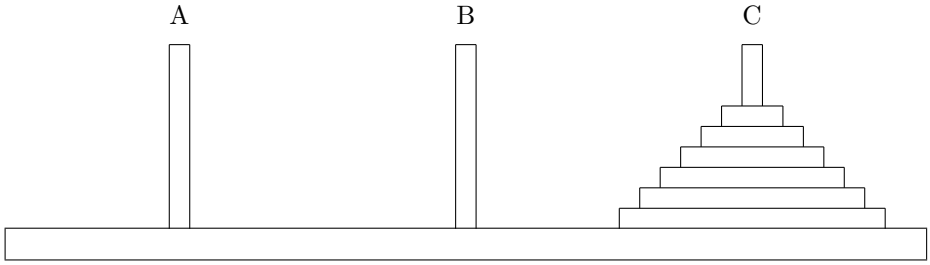
それでは、再帰を使って 10 進数を整数に変換するプログラムを書いてみましょう。

プログラムの例　dtoint.cas

```
DTOINT   START
         IN    INAREA,INLENGTH
         LD    GR1,INLENGTH
         LAD   GR1,-1,GR1
         CALL  D2ISUB
         ST    GR2,INTEGER
         DMEM  MSG,INTEGER,DUMPE
         RET
;
D2ISUB   CPA   GR1,=-1
         JZE   BASIS
         PUSH  0,GR1
         LAD   GR1,-1,GR1
         CALL  D2ISUB
         POP   GR1
         CALL  TENTIMES
         LD    GR3,INAREA,GR1
         SUBL  GR3,=48
         ADDL  GR2,GR3
         RET
BASIS    LAD   GR2,0
         RET
;
TENTIMES LAD   GR4,0
         LD    GR5,GR2
LOOP     ADDL  GR2,GR5
         LAD   GR4,1,GR4
         CPL   GR4,=9
```



初期状態



最終状態

図 13.1: ハノイの塔

```
          JMI    LOOP
          RET
;
INAREA    DS     256
INLENGTH  DS     1
INTEGER   DS     1
DUMPE     DC     -1
MSG       DC     'MESSAGE1'
          END
```

このプログラムは、文字列を読み込んで、その文字列を 10 進数として解釈したときにそれが表現している整数を求めて、その整数が格納されている主記憶装置の語を DMEM 命令で表示します。

このプログラムのメインルーチンは、読み込んだ 10 進数の長さを GR1 にロードして、それをデクリメントしたのち、D2ISUB から始まるサブルーチンを呼び出します。そうすると、読み込んだ 10 進数を整数に変換した結果が GR2 に格納されますので、それを INTEGER 番地の語にストアして、その語を DMEM 命令で表示します。GR1 は、解釈の対象となる 10 進数の右端の桁が、INAREA から始まる領域の何番目の語に格納されているか、ということを示します。

D2ISUB から始まるサブルーチンは、再帰を使って、INAREA から始まる領域に格納されている 10 進数を解釈して、それが表現している整数を GR2 に格納します。

TENTIMES から始まるサブルーチンは、GR2 を 10 倍します。

実行例

```
IN ? 3030
* MESSAGE1 *    Start:0132 End:0133
 0130: ---- ---- 0BD6 FFFF ---- ---- ---- ----    ..
```

13.8 ハノイの塔

13.8.1 ハノイの塔の基礎

これまでこの章で紹介してきた問題は、再帰を使うことによってそのプログラムを書くことができるわけですが、再帰を使わないでプログラムを書くことも、それほど難しいことではありません。しかし、問題の中には、再帰を使えばプログラムを簡単に書くことができるけれども、再帰を使わないでプログラムを書くことはきわめて難しい、というものもあります。たとえば、「ハノイの塔」(Tower of Hanoi) と呼ばれるパズルを解くという問題は、そのような問題の一例です。

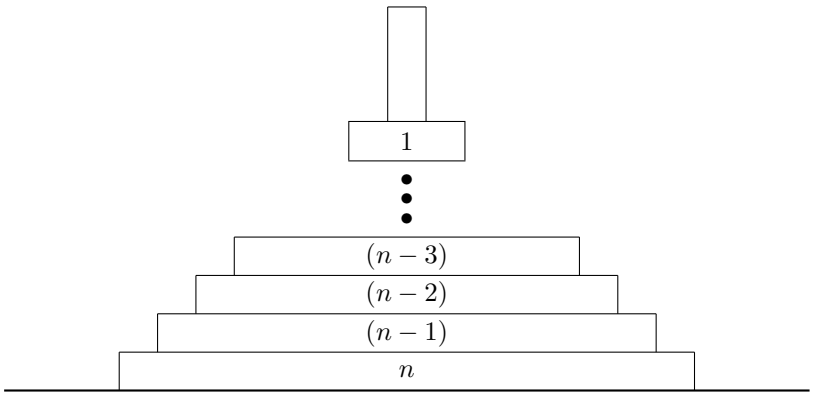


図 13.2: 円盤の番号

ハノイの塔では、3 本の棒が垂直に立っている台と、直径が少しずつ違う何枚かの円盤が道具として使われます。それらの円盤は、中央に穴が開いていて、台の上の棒にはめ込むことができるようになっています。

ハノイの塔は、円盤を動かしていくことによって初期状態から最終状態へ移行させるための手順を求めてください、というパズルです。初期状態と最終状態というのは、図 13.1 のような状態のことです。つまり、初期状態では、すべての円盤が棒 A にはまっていて、しかも下にある円盤ほど直径が大きいという順番になっています。そして最終状態では、すべての円盤が棒 C にはまっていて、そして初期状態と同じように、下にある円盤ほど直径が大きいという順番になっていないといけません。

円盤を動かすときには、次の二つの規則にしたがう必要があります。

- 1 回の操作で実行できるのは、どれかの棒にはめ込まれている円盤のうちでもっとも上にある 1 枚を棒から抜き取って、それを別の棒にはめ込む、ということだけである。
- すでに棒にはめ込まれている円盤よりも直径の大きな円盤をその棒にはめ込むことはできない。

円盤の枚数が n 枚だとするとき、図 13.2 のように、直径の大きな円盤から順番に、n、$(n-1)$、$(n-2)$、$(n-3)$、……、1、という番号がそれぞれの円盤に与えられているとします。そして、n 番目から 1 番目までの円盤を、上へ行くほど小さくなるように重ねたもののことを、「n 円錐」と呼ぶことにします。そうすると、n 円錐から n 番目の円盤を取り除いた部分は、「$(n-1)$ 円錐」と呼ばれることになります。

ハノイの塔の 3 本の棒のそれぞれは、「出発点」、「待避所」、「目的地」という 3 種類の役割を持っていると考えることができます。出発点というのは、n 円錐がそこから出発していく棒のことで、目的地というのは、移動が終わったときに n 円錐がはめ込まれている棒のことです。そして待避所というのは、n 番目の円盤を移動させるために $(n-1)$ 円錐を待避させておくための棒のことです。

ただし、どの棒がどの役割なのかという関係は、固定されていません。パズルの全体としては、棒 A が出発点で、棒 B が待避所で、棒 C が目的地ですが、パズルを解いていく過程で、棒と役割の関係は変化します。

ハノイの塔は、次のような再帰的な手順を実行することによって解くことができます。

ステップ 1　$(n-1)$ 円錐を出発点から目的地経由で待避所へ移動させる。

ステップ 2　n 番目の円盤を出発点から目的地へ移動させる。

ステップ 3　$(n-1)$ 円錐を待避所から出発点経由で目的地へ移動させる。

図 13.3 は、この手順を図で示したものです。

ハノイの塔を解く再帰的な手順の基底は、n が 0 の場合です。円盤の枚数が 0 のハノイの塔は、初期状態と終了状態がまったく同じです。したがって、n が 0 の場合は、何もしないというのが、それを解く手順になります。

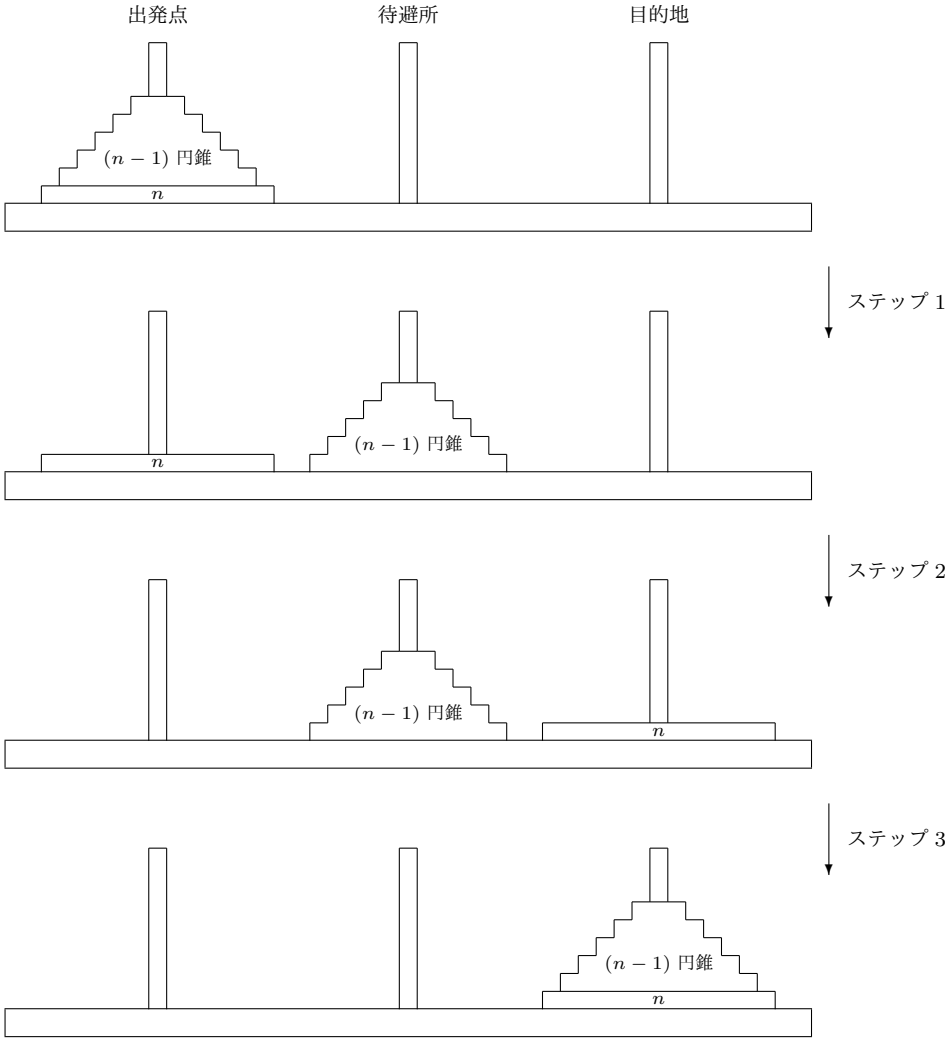


図 13.3: ハノイの塔の解法

13.8.2 再帰によるハノイの塔のプログラム

それでは、再帰を使ってハノイの塔を解くプログラムを書いてみましょう。

プログラムの例 hanoi.cas

```
HANOI    START
         LD    GR1,N
         LD    GR2,='A'
         LD    GR3,='B'
         LD    GR4,='C'
         CALL  HANOISUB
         RET
;
HANOISUB CPL   GR1,=0
         JZE   BASIS
         PUSH  0,GR1
         PUSH  0,GR2
         PUSH  0,GR3
         PUSH  0,GR4
         LAD   GR1,-1,GR1
         LD    GR5,GR3
         LD    GR3,GR4
         LD    GR4,GR5
         CALL  HANOISUB
         POP   GR4
         POP   GR3
         POP   GR2
```

```
          CALL  MOVE
          PUSH  0,GR2
          PUSH  0,GR3
          PUSH  0,GR4
          LD    GR5,GR2
          LD    GR2,GR3
          LD    GR3,GR5
          CALL  HANOISUB
          POP   GR4
          POP   GR3
          POP   GR2
          POP   GR1
BASIS     RET
;
MOVE      ST    GR2,SRC
          ST    GR4,DEST
          OUT   SRC,LENGTH
          RET
;
N         DC    3
SRC       DS    1
          DC    ' -> '
DEST      DS    1
LENGTH    DC    6
          END
```

このプログラムは、N 番地の整数を円盤の枚数とするハノイの塔について、それを初期状態から最終状態へ移行させるための手順を出力します。

このプログラムのメインルーチンは、円盤の枚数を GR1 にロードして、GR2、GR3、GR4 のそれぞれに A、B、C という棒の名前をロードしたのち、HANOISUB から始まるサブルーチンを呼び出します。GR2 は出発点、GR3 は待避所、GR4 は目的地を意味しています。

HANOISUB から始まるサブルーチンは、再帰を使って、GR1 の内容を円盤の枚数とするハノイの塔を解くための手順を出力します。

MOVE から始まるサブルーチンは、円盤をどの棒からどの棒へ移動させるかということを、

出発点 -> 目的地

という形式で出力します。

実行例

```
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

付録 A　練習問題

問題 1

DC 命令が次のように書かれているとします。

```
A         DC    #2345
B         DC    #ABCD
MSG       DC    'MESSAGE1'
```

このとき、A 番地のデータと B 番地のデータとを入れ替えて、A 番地と B 番地の内容を DMEM 命令で表示するプログラムを書いてください。

実行例

```
* MESSAGE1 *    Start:000D End:000E
 0008: ---- ---- ---- ---- ---- ABCD 2345 ----       ..
```

問題 2

DC 命令と DS 命令が次のように書かれているとします。

```
A       DC    7
B       DS    2
DUMPE   DC    -1
MSG     DC    'MESSAGE1'
```

このとき、A 番地のデータをインクリメントした結果を B 番地にストアして、A 番地のデータをデクリメントした結果を B 番地の次の語にストアして、A 番地から DUMPE 番地までの内容を DMEM 命令で表示するプログラムを書いてください。

実行例

```
* MESSAGE1 *    Start:0011 End:0014
 0010: ---- 0007 0008 0006 FFFF ---- ---- ----   ....
```

問題 3

DC 命令と DS 命令が次のように書かれているとします。

```
A       DC    3
B       DC    8
ADD     DS    1
SUB     DS    1
DUMPE   DC    -1
MSG     DC    'MESSAGE1'
```

このとき、A 番地のデータと B 番地のデータとを算術加算した結果を ADD 番地にストアして、A 番地のデータから B 番地のデータを算術減算した結果を SUB 番地にストアして、A 番地から DUMPE 番地までの内容を DMEM 命令で表示するプログラムを書いてください。

実行例

```
* MESSAGE1 *    Start:0011 End:0015
 0010: ---- 0003 0008 000B FFFB FFFF ---- ----   .....
```

問題 4

DC 命令と DS 命令が次のように書かれているとします。

```
A       DC    #F0F0
B       DC    #FF00
AND     DS    1
OR      DS    1
XOR     DS    1
DUMPE   DC    -1
MSG     DC    'MESSAGE1'
```

このとき、A 番地のデータと B 番地のデータとの論理積を AND 番地にストアして、A 番地のデータと B 番地のデータとの論理和を OR 番地にストアして、A 番地のデータと B 番地のデータとの排他的論理和を XOR 番地にストアして、A 番地から DUMPE 番地までの内容を DMEM 命令で表示するプログラムを書いてください。

実行例

```
* MESSAGE1 *    Start:0017 End:001C
 0010: ---- ---- ---- ---- ---- ---- ---- F0F0         .
 0018: FF00 F000 FFF0 0FF0 FFFF ---- ---- ----  .....
```

問題 5

DC 命令と DS 命令が次のように書かれているとします。

```
A       DC    #67BC
B       DS    1
MSG     DC    'MESSAGE1'
```

このとき、A 番地のデータの下位 8 ビットと上位 8 ビットとを置き換えた結果を B 番地にストアして、A 番地と B 番地の内容を DMEM 命令で表示するプログラムを書いてください。

実行例

```
* MESSAGE1 *    Start:0010 End:0011
 0010: 67BC BC67 ---- ---- ---- ---- ---- ----  ..
```

問題 6

DREG 命令でレジスターの内容を表示しながら、GR0 の内容を、0 から出発して 100 よりも小さい範囲で 7 ずつ増やしていくプログラムを書いてください。GR0 の内容は、

```
0 7 14 21 28 35 42 49 56 63 70 77 84 91 98
```

と変化することになります。

問題 7

DC 命令と DS 命令が次のように書かれているとします。

```
LENGTH   DC    8
A        DC    38,32,54,27,48,30,63,34
RESULT   DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
```

このとき、A 番地から始まる 8 語の領域に格納されている整数のうちでもっとも大きいものを求めて、それを RESULT 番地にストアして、A 番地から DUMPE 番地までの内容を DMEM 命令で表示するプログラムを書いてください。

実行例

```
* MESSAGE1 *    Start:0020 End:0021
 0020: 003F FFFF ---- ---- ---- ---- ---- ----  ?.
```

ヒント　このプログラムは、挑戦者がチャンピオンに次々と挑戦していく、という考え方で書くことができます。まず、列の先頭にある整数を暫定的にチャンピオンの座に据えます。そして、二番目以降の整数を、順番にチャンピオンに挑戦させます。チャンピオンよりも大きな整数が出現したときは、チャンピオンはその座を挑戦者に譲ります。そうすると、すべての挑戦が終了した時点でチャンピオンの座に就いている整数が、列の中でもっとも大きいということになります。

問題 8

DC 命令と DS 命令が次のように書かれているとします。

```
LENGTH   DC    14
A        DC    43,27,88,65,31,46,72,83,36,53,24,37,64,22
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このとき、A 番地から始まる 14 語の領域に格納されている整数の列を昇順にソートして、A 番地から DUMPE 番地までの内容を DMEM 命令で表示するプログラムを書いてください。

実行例

```
* MESSAGE1 *    Start:002D End:003B
 0028: ---- ---- ---- ---- ---- 0016 0018 001B        ...
 0030: 001F 0024 0025 002B 002E 0035 0040 0041  .$%+.5@A
 0038: 0048 0053 0058 FFFF ---- ---- ---- ----  HSX.
```

ヒント　いくつかのデータが、先頭から末尾へ行くにしたがってしだいに大きくなるという順番で並んでいるとき、そのデータの列は「昇順」(ascending order) に並んでいると言われます。それとは逆に、先頭から末尾へ行くにしたがってしだいに小さくなるという順番で並んでいるとき、そのデータの列は「降順」(descending order) に並んでいると言われます。

データの列が与えられたとき、その列を構成しているそれぞれのデータが昇順または降順に並ぶように、その列を並べ替えることを、その列を「ソートする」(sort) と言います。

ソートのための手順には、さまざまなものがあります。ここでは、それらのうちで比較的単純な、「バブルソート」(bubble sort) と呼ばれる手順[1]を紹介しましょう。

バブルソートは、隣接するデータを比較して、それらの大小関係が逆になっている場合はそれらの位置を入れ替える、ということを繰り返す、というソートの手順です。

[1] バブルソートは、「単純交換法」と呼ばれることもあります。

この手順が「バブルソート」と呼ばれる理由は、この手順によってデータが移動していく様子が、水の中で泡（バブル）が浮かび上がっていく様子に似ているからです。

例として、次のような、5 個の整数から構成される列を、バブルソートを使って昇順にソートしてみましょう。

```
5  3  6  4  2
```

まず、隣接するデータを比較して、左のほうが右よりも大きいならばそれらの位置を入れ替える、ということを左から右に向かって実行していきます。

```
5  3  6  4  2
↑  ↑           入れ替え

3  5  6  4  2
   ↑  ↑        そのまま

3  5  6  4  2
      ↑  ↑     入れ替え

3  5  4  6  2
         ↑  ↑  入れ替え

3  5  4  2  6
```

そうすると、5 個の整数のうちで最も大きい 6 という整数が右端に移動しますので、次は、右端の整数を除いた 4 個の整数について、同じことを繰り返します。そうすると、

```
3  4  2  5  6
```

というように、2 番目に大きい 5 が右端から 2 番目の位置に移動します。このように、比較する範囲を狭くしながら同じことを繰り返していくと、最後には、

```
2  3  4  5  6
```

というように、5 個の整数が昇順に並ぶことになります。

問題 9

DC 命令と DS 命令が次のように書かれているとします。

```
N        DC    8820
FACTORS  DS    16
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

このとき、N 番地に格納されている整数を素因数分解して、得られた素因数を、FACTORS から始まる領域に格納して、N 番地から DUMPE 番地までの内容を DMEM 命令で表示するプログラムを書いてください。

実行例

```
* MESSAGE1 *    Start:002F End:0040
 0028: ---- ---- ---- ---- ---- ---- ---- 2274         .
 0030: 0002 0002 0003 0003 0005 0007 0007 0000  ........
 0038: 0000 0000 0000 0000 0000 0000 0000 0000  ........
 0040: FFFF ---- ---- ---- ---- ---- ---- ----  .
```

ヒント

n と m が整数だとするとき、n を m で除算したときの余りが 0 ならば、m は、n の「約数」(divisor) であると言われます。たとえば、6 は 18 の約数です。

2 以上の整数のうちで、1 と自分自身を除くいかなる約数も持たないものは、「素数」(prime number) と呼ばれます。たとえば、2、3、5、7、11、13、17 などは素数です。それとは逆に、プラスの整数のうちで、1 と自分自身のほかにも約数を持つものは、「合成数」(composite number) と呼ばれます。たとえば、4、6、8、9、10、12、14、15、16 などは合成数です。

n が 2 以上の整数だとするとき、n の約数であるような素数は、n の「素因数」(prime factor) であると言われます。たとえば、7 は 42 の素因数です。

2 以上の任意の整数は、素因数の積に分解することができます。与えられた整数について、それを素因数の積に分解することを、「素因数分解」(integer factorization) と言います。たとえば、8820 という整数は、

$$2 \times 2 \times 3 \times 3 \times 5 \times 7 \times 7$$

という素因数の積に素因数分解することができます。

n が 2 以上の整数だとするとき、次のような処理を実行することによって、n を素因数分解することができます。

(1) 2 を m とする。
(2) n が 1 ならば終了する。
(3) n を m で除算して、商と余りを求める。
(4) 余りが 0 ならば、m を素因数に追加して、商を n として、(2) に戻る。
(5) 余りが 0 ではないならば、m に 1 を加算した結果を m として、(3) に戻る。

問題 10

2 から 100 までの範囲にあるすべての素数を求めて、それらの素数と、それらの素数の個数を DMEM 命令で表示するプログラムを書いてください。

実行例

```
* MESSAGE1 *    Start:00B7 End:00EA
 00B0: ---- ---- ---- ---- ---- ---- 0002         .
 00B8: 0003 0005 0007 000B 000D 0011 0013 0017  ........
 00C0: 001D 001F 0025 0029 002B 002F 0035 003B  ..%)+/5;
 00C8: 003D 0043 0047 0049 004F 0053 0059 0061  =CGIOSYa
 00D0: 0000 0000 0000 0000 0000 0000 0000 0000  ........
*00E8: 0000 0019 FFFF ---- ---- ---- ---- ----  ...
```

ヒント

n が 2 以上の整数だとするとき、2 から n までの範囲にあるすべての素数を求めるための手順としては、「エラトステネスのふるい」(sieve of Eratosthenes) と呼ばれるものがよく知られています。

2 から n までの範囲にあるすべての素数を求めるエラトステネスのふるいは、次のような手順です。

(1) 2 から n までのすべての整数を並べた列を作る。
(2) 2 を i とする。
(3) 次の (4) と (5) を、i^2 が n よりも大きくなるまで繰り返す。
(4) i が列の中にあるならば、その倍数のうちで列の中にあるものをすべて取り除く。ただし、i 自身は取り除かない。
(5) i に 1 を加算した整数を i とする。

エラトステネスのふるいが終了すると、素数だけが列の中に残ることになります。

問題 11

文字列を読み込んで、その文字列に含まれているすべての英字の大文字を X に変換して、すべての英字の小文字を x に変換して、それ以外の文字はそのままにすることによってできる文字列を出力するプログラムを書いてください。

実行例

```
IN ? Let's "Hello, World!"
Xxx'x "Xxxxx, Xxxxx!"
```

問題 12

文字列を読み込んで、その文字列に含まれているすべての空白を削除することによってできる文字列を出力するプログラムを書いてください。

実行例

```
IN ? dog  cat   mouse    rabbit     monkey
dogcatmouserabbitmonkey
```

問題 13

文字列を読み込んで、その文字列に含まれている連続する空白を 1 個の空白に変換することによってできる文字列を出力するプログラムを書いてください。

実行例

```
IN ? dog  cat   mouse    rabbit     monkey
dog cat mouse rabbit monkey
```

ヒント　このプログラムは、汎用レジスタのうちの 1 個を、文字をコピーする状態にあるかどうかを記憶するために使うことによって、簡単に書くことができます。そのために使う汎用レジスタを、「連続空白フラグ」と呼ぶことにしましょう。連続空白フラグが 0 のときは文字をコピーして、1 のときは文字をコピーしません。

連続空白フラグの初期値は 0 です。空白ではない文字が出現したときも、それに 0 を設定します。そして、それが 0 のときは、普通の文字も、空白もコピーします。ただし、空白をコピーしたときは、連続空白フラグに 1 を設定します。そうすると、空白が連続して出現する場合、2 個目以降の空白はコピーされません。

問題 14

文字列を読み込んで、その文字列をキャピタライズした結果を出力するプログラムを書いてください。

「キャピタライズ」(capitalize) というのは、空白で区切られたいくつかの単語から構成される文字列が与えられたとき、それを構成しているすべての単語について、その先頭の文字が小文字ならばそれを大文字に変換する、という処理をすることです。

実行例

```
IN ? good code is its own best documentation
Good Code Is Its Own Best Documentation
```

問題 15

文字列を読み込んで、その文字列を、16 進数で表記された文字コードを空白で区切って並べることによってできる文字列に変換して、その結果を出力するプログラムを書いてください。

実行例

```
IN ? Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21
```

問題 16

括弧列を読み込んで、それが整形式括弧列ならば `yes`、そうでないならば `no` を出力するプログラムを書いてください。

この問題では、「括弧列」というのは、左丸括弧と右丸括弧という 2 種類の文字を並べることによってできる文字列のことだとします。たとえば、(、))、()(、)(((、……などは括弧列です。なお、空文字列も括弧列だとします。

また、この問題では、「整形式括弧列」というのは、次のように再帰的に定義される括弧列のことだとします。

- 空文字列は整形式括弧列である。
- 整形式括弧列を丸括弧で囲んだものは整形式括弧列である。
- 整形式括弧列と整形式括弧列とを連結したものは整形式括弧列である。

たとえば、空文字列は整形式括弧列ですから、空文字列を丸括弧で囲んだ () という括弧列は整形式括弧列です。そして、() という整形式括弧列を丸括弧で囲んだ (()) という括弧列も整形式括弧列です。同様に、((())) 、(((()))) 、((((())))) なども整形式括弧列です。

また、() は整形式括弧列ですから、それを二つ連結した ()() という括弧列も整形式括弧列です。同様に、(()) と () とを連結した (())() や、()() と () とを連結した ()()() なども整形式括弧列です。

さらに、()() を丸括弧で囲んだ (()())、(())() を丸括弧で囲んだ ((())())、()()() を丸括弧で囲んだ (()()()) なども整形式括弧列です。

実行例

```
IN ? (((()))())
yes

IN ? (((()))))())
no

IN ? (((()))()
no
```

ヒント

括弧列が整形式括弧列かどうかということは、それの深さを調べることによって判定することができます。

括弧列の「深さ」というのは、括弧列の中の位置が、丸括弧に何重に囲まれているかということです。まったく囲まれていなければ 0 で、n 重に囲まれていれば n です。

括弧列を左から右へ走査しながら括弧列の深さを調べていった場合、その括弧列が整形式括弧列ならば、その過程で深さがマイナスになるということはあり得ません。なぜなら、深さがマイナスになるというのは、左丸括弧に対応しない右丸括弧が存在するということだからです。したがって、もしも、走査の途中で深さがマイナスになったならば、その括弧列は整形式括弧列ではないと判定することができます。

また、括弧列が整形式括弧列ならば、丸括弧の深さは、丸括弧列の右端まで走査が終わったときにかならず 0 に戻ります。なぜなら、0 に戻らないというのは、左丸括弧に対応する右丸括弧が存在しないということだからです。したがって、もしも、走査が終わったときに深さが 0 に戻らなかったならば、その場合も、その括弧列は整形式括弧列ではないと判定することができます。

問題 17

中置記法の式を読み込んで、同じ構造を持つ逆ポーランド記法の式にそれを変換した結果を出力するプログラムを書いてください。

この問題では、「中置記法の式」というのは、次のように再帰的に定義される文字列のことだとします。

- 1 個の英字の小文字は変数名である。
- +、-、*、/、% は演算子である。
- 変数名は中置記法の式である。
- 左丸括弧、式、演算子、式、右丸括弧を、この順番で並べたものは中置記法の式である。

たとえば、次の文字列は、この問題での中置記法の式です。

```
a
(a+b)
((a+b)*c)
((a+b)/(c-d))
```

また、この問題では、「逆ポーランド記法の式」というのは、次のように再帰的に定義される文字列のことだとします。

- 1 個の英字の小文字は変数名である。
- +、-、*、/、% は演算子である。
- 変数名は逆ポーランド記法の式である。
- 式、式、演算子を、この順番で並べたものは逆ポーランド記法の式である。

たとえば、次の文字列は、この問題での逆ポーランド記法の式です。

```
a
ab+
ab+c*
ab+cd-/
```

実行例

```
IN ? ((a+b)/(c-d))
```

```
ab+cd-/
```

ヒント　中置記法の式は、次のような処理を実行することによって、それと同じ構造を持つ逆ポーランド記法の式に変換することができます。

(1) 主記憶装置の領域に、あらかじめ、これから逆ポーランド記法の式になる文字列として、空文字列を格納しておく。

(2) 中置記法の式を左から右へ走査しながら、出現したものに応じて、次の処理を実行する。

- 左丸括弧が出現した場合は、何もしない。
- 演算子が出現した場合は、それをスタックにプッシュする。
- 変数名が出現した場合は、これから逆ポーランド記法の式になる文字列の末尾にそれを連結する。
- 右丸括弧が出現した場合は、スタックから演算子をポップして、これから逆ポーランド記法の式になる文字列の末尾にそれを連結する。

入力される文字列は、必ずしも正しい中置記法の式ばかりとは限りません。ですから、この問題のプログラムは、正しくない式が入力された場合のことも考えて書く必要があります。

たとえば、`(a+b))`という文字列が入力されたとしましょう。これは、左中括弧に対応していない右中括弧がありますので、正しい中置記法の式ではありません。ですから、二つ目の右丸括弧が出現したとき、スタックに演算子は存在しません。

そのような問題に対処できるように、式を処理する前にあらかじめ、演算子ではない何らかのデータ（たとえば`#FFFF`）を安全弁としてスタックにプッシュしておくといいでしょう。もしも、式を処理している途中でポップしたデータが安全弁だったならば、入力された式は正しくなかったことになりますので、そこで処理を終了すればいいのです。ちなみに、処理が正常に終了した場合は、最後に安全弁がスタックに残りますので、処理が終わったのちにそれをポップする必要があります。

また、`(a+b-`という文字列が入力されたとしましょう。これも、正しい中置記法の式ではありません。この文字列を処理した場合、末尾まで走査が終わった時点で、`+`という演算子がスタックに残ったままになります。この場合は、スタックに残った演算子をポップするという処理が必要になります。

問題 18

2 個の 10 進数を読み込んで、それらを加算した結果を 10 進数で出力するプログラムを書いてください。

実行例

```
IN ? 78
IN ? 65
143

IN ? 64032845721095387214388421035 6
IN ? 2308123154337029
64032845721095618026703854738 5
```

ヒント　1 個の 10 進数を 1 語の整数に変換してもいいのですが、その方法だと、0 から 65535 までの整数しか扱えません。10 進数を構成しているそれぞれの桁を 1 個の整数に変換して、整数の列の形で 1 個の 10 進数を表現すれば、上の実行例の二つ目のように、巨大な 10 進数も扱うことができます。ただし、`IN`命令で読み込むことのできる文字列の長さが 256 文字までですので、扱える 10 進数の桁数は 256 桁までです。

読み込んだ 10 進数を整数の列に変換するとき、ついでに、桁の順番を逆にしておきましょう。つまり、1 の桁、10 の桁、100 の桁、……という順番で桁を並べるのです。なぜなら、そのほうが加算の処理が簡単になるからです。また、2 個の 10 進数のうちの、桁数が少ないほうは、値が 0 の桁を上位に追加することによって、両方の桁数が同じになるようにしておきましょう。

加算の処理は、筆算と同じ方法を使うことができます。桁と桁とを加算した結果が 10 以上の場合は、桁上がりの処理をする必要があります。つまり、10 を減算して、一

つ上の桁に 1 を加算するわけです。

最上位の桁を加算した結果が 10 以上の場合は、さらにひとつ上の桁を作って、その値を 1 にする必要があります。

問題 19

大文字または小文字によるローマ数字を読み込んで、それがあらわしている整数を 4 桁の 10 進数で出力するプログラムを書いてください。

「ローマ数字」(Roman numerals) というのは、古代ローマにおいて使用されていた記数法、またはその記数法を使って整数を表現している文字列のことです。

ローマ数字では、I、V、X、L、C、D、M という 7 種類の文字が使われます（i、v、x、l、c、d、m という小文字を使うこともできます）。これらの文字のそれぞれは、次のように、特定の整数をあらわしています。

I = 1　　V = 5
X = 10　　L = 50
C = 100　　D = 500
M = 1000

I、X、C、M の 4 種類の文字に関しては、同じ文字を最大 3 個まで連続して並べることができます。同じ文字を並べたものの全体は、次の表のように、それぞれの文字があらわしている整数を合計した結果をあらわします。

II = 2	XX = 20	CC = 200	MM = 2000
III = 3	XXX = 30	CCC = 300	MMM = 3000

それに対して、V、L、D の 3 種類の文字に関しては、同じ文字を複数個並べるということはできません。

異なる種類の文字を並べた場合も、それらの文字の全体は、それらの文字があらわしている整数を合計した結果をあらわします。ただし、異なる種類の文字を並べる場合は、原則として、大きな整数をあらわしている文字ほど左に書かないといけません。

VI = 6	XI = 11	XXII = 22	XXXVIII = 38
LX = 60	CX = 110	CCXX = 220	CCCLXXX = 380
DC = 600	MC = 1100	MMCC = 2200	MMMDCCC = 3800

大きな整数をあらわしている文字ほど左に書くという規則には、例外があります。それは、「減算則」と呼ばれる規則を使う場合です。

減算則は、「小さな整数をあらわす文字の右側に、それよりも大きな整数をあらわす文字を書いた場合、その 2 文字は、大きな整数から小さな整数を減算した結果をあらわす」という規則です。4、9、40、90、400、900 のそれぞれは、この規則を使うことによって、次のような 2 文字のローマ数字によって表現することができます。

IV = 4	XL = 40	CD = 400
IX = 9	XC = 90	CM = 900

ローマ数字は、以上の規則によって、1 から 3999 までの整数を表現することができます。

XIV = 14	XLIX = 49	MCMLVIII = 1958
MMMCDXLIV = 3444	MMMCDXCVIII = 3498	MMMCMXCIX = 3999

実行例

```
IN ? mmmcdxcviii
3498
```

問題 20

DC 命令と DS 命令が次のように書かれているとします。

```
A         DC    3
```

```
B       DC    5
P       DS    1
DUMPE   DC    -1
MSG     DC    'MESSAGE1'
```

A 番地の整数を a、B 番地の整数を b とするとき、a の b 乗 (a^b) を P 番地にストアして、A 番地から DUMPE 番地までの内容を DMEM 命令で表示するプログラムを、再帰を使って書いてください。

実行例

```
* MESSAGE1 *    Start:002C End:002F
 0028: ---- ---- ---- ---- 0003 0005 00F3 FFFF      ....
```

ヒント

a の b 乗を求めるという計算は、「べき乗」(power) と呼ばれます。
a の b 乗は、b 個の a を乗算することによって求めることができます。つまり、

$$\underbrace{a \times a \times a \times \cdots \times a}_{b\text{ 個}}$$

という計算をすればいいわけです。ただし、a^0 は 1 だと定義します。
たとえば、3^5 は、

$$3 \times 3 \times 3 \times 3 \times 3$$

という計算をすればいいわけですから、243 ということになります。
べき乗という概念は、再帰的な構造を持っています。なぜなら、べき乗は、

$$\begin{cases} & a^0 = 1 \\ b \geq 1 \text{ ならば} & a^b = a \times a^{b-1} \end{cases}$$

というように再帰的に定義することができるからです。

付録 B　練習問題の解答例

問題 1 の解答例

```
SWAP    START
        LD    GR0,A
        LD    GR1,B
        ST    GR0,B
        ST    GR1,A
        DMEM  MSG,A,B
        RET
A       DC    #2345
B       DC    #ABCD
MSG     DC    'MESSAGE1'
        END
```

問題 2 の解答例

```
INCDEC  START
        LD    GR1,A
        LAD   GR2,1,GR1
        ST    GR2,B
        LAD   GR2,-1,GR1
        LAD   GR3,1
        ST    GR2,B,GR3
        DMEM  MSG,A,DUMPE
        RET
A       DC    7
B       DS    2
DUMPE   DC    -1
MSG     DC    'MESSAGE1'
        END
```

問題 3 の解答例

```
ADDSUB  START
        LD    GR1,A
        ADDA  GR1,B
```

```
         ST    GR1,ADD
         LD    GR1,A
         SUBA  GR1,B
         ST    GR1,SUB
         DMEM  MSG,A,DUMPE
         RET
A        DC    3
B        DC    8
ADD      DS    1
SUB      DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

問題 4 の解答例

```
ANDOR    START
         LD    GR1,A
         AND   GR1,B
         ST    GR1,AND
         LD    GR1,A
         OR    GR1,B
         ST    GR1,OR
         LD    GR1,A
         XOR   GR1,B
         ST    GR1,XOR
         DMEM  MSG,A,DUMPE
         RET
A        DC    #F0F0
B        DC    #FF00
AND      DS    1
OR       DS    1
XOR      DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

問題 5 の解答例

```
REPLACE  START
         LD    GR1,A
         LD    GR2,A
         SLL   GR1,8
         SRL   GR2,8
         OR    GR1,GR2
         ST    GR1,B
         DMEM  MSG,A,B
         RET
A        DC    #67BC
B        DS    1
MSG      DC    'MESSAGE1'
         END
```

問題 6 の解答例

```
SEVEN    START
         LAD   GR0,0
LOOP     DREG  MSG
         ADDL  GR0,STEP
         CPL   GR0,LIMIT
         JMI   LOOP
         RET
STEP     DC    7
LIMIT    DC    100
MSG      DC    'MESSAGE1'
         END
```

問題 7 の解答例

```
MAX      START
         LD    GR1,A
         LAD   GR2,1
```

```
LOOP     CPL   GR1,A,GR2
         JPL   SKIP
         LD    GR1,A,GR2
SKIP     LAD   GR2,1,GR2
         CPA   GR2,LENGTH
         JMI   LOOP
         ST    GR1,RESULT
         DMEM  MSG,RESULT,DUMPE
         RET
LENGTH   DC    8
A        DC    38,32,54,27,48,30,63,34
RESULT   DS    1
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

問題 8 の解答例

```
SORT     START
         CALL  SORTSUB
         DMEM  MSG,A,DUMPE
         RET
;
; SORTSUB
; A から始まる領域に格納されている整数の列を昇順にソートする。
SORTSUB  LD    GR3,LENGTH
         LAD   GR3,-1,GR3
LOOP1    LAD   GR1,0
LOOP2    LD    GR0,A,GR1
         LAD   GR2,1,GR1
         CPL   GR0,A,GR2
         JMI   SKIP
         CALL  SWAP
SKIP     LAD   GR1,1,GR1
         CPL   GR1,GR3
         JMI   LOOP2
         LAD   GR3,-1,GR3
         CPL   GR3,=0
         JPL   LOOP1
         RET
;
; SWAP
; 領域 A の GR1 番目と GR2 番目の内容を入れ替える。
SWAP     LD    GR4,A,GR1
         LD    GR5,A,GR2
         ST    GR4,A,GR2
         ST    GR5,A,GR1
         RET
;
LENGTH   DC    14
A        DC    43,27,88,65,31,46,72,83,36,53,24,37,64,22
DUMPE    DC    -1
MSG      DC    'MESSAGE1'
         END
```

問題 9 の解答例

```
PRIME    START
         CALL  PRISUB
         DMEM  MSG,N,DUMPE
         RET
;
; N 番地の整数を素因数分解する。
; 得られた素因数は、FACTORS から始まる領域にストアされる。
PRISUB   LD    GR0,N
         LAD   GR1,2
         LAD   GR2,0
LOOP1    CPL   GR0,=1
         JZE   EXIT1
LOOP2    CALL  DIV
         CPL   GR4,=0
         JZE   SKIP
         LAD   GR1,1,GR1
```

```
          JUMP  LOOP2
SKIP      ST    GR1,FACTORS,GR2
          LAD   GR2,1,GR2
          LD    GR0,GR3
          JUMP  LOOP1
EXIT1     RET
;
; DIV
; GR0 を GR1 で除算する。
; 商は GR3 に、余りは GR4 に格納される。
DIV       LD    GR4,GR0
          LAD   GR3,0
LOOP3     CPL   GR4,GR1
          JMI   EXIT2
          SUBL  GR4,GR1
          LAD   GR3,1,GR3
          JUMP  LOOP3
EXIT2     RET
;
N         DC    8820
FACTORS   DS    16
DUMPE     DC    -1
MSG       DC    'MESSAGE1'
          END
```

問題 10 の解答例

```
ERATOS    START
          CALL  INIT
          CALL  ERASUB
          CALL  STORE
          DMEM  MSG,PRIMES,DUMPE
          RET
;
; INIT
; エラトステネスのふるいを初期化する。
; SIEVE+2 番地から SIEVE+N 番地までのすべての語に
; 1 をストアする。
INIT      LAD   GR1,2
          LAD   GR0,1
LOOP1     ST    GR0,SIEVE,GR1
          LAD   GR1,1,GR1
          CPL   GR1,N
          JMI   LOOP1
          JZE   LOOP1
          RET
;
; ERASUB
; エラトステネスのふるいを実行する。
; SIEVE+合成数番地の語に 0 をストアする。
ERASUB    LAD   GR3,0
          LAD   GR1,1
LOOP2     LAD   GR1,1,GR1
          CPL   GR1,ROOTN
          JPL   EXIT2
          LD    GR0,SIEVE,GR1
          CPL   GR0,=1
          JZE   EXIT1
          JUMP  LOOP2
EXIT1     LD    GR2,GR1
LOOP3     ADDL  GR2,GR1
          CPL   GR2,N
          JPL   LOOP2
          ST    GR3,SIEVE,GR2
          JUMP  LOOP3
EXIT2     RET
;
; STORE
; PRIMES から始まる領域に素数を格納して、
; その個数を M 番地に格納する。
; SIEVE+i 番地の内容が 1 であるような i を
; PRIMES から始まる領域にストアする。
STORE     LAD   GR1,2
```

```
          LAD   GR2,0
LOOP4     LD    GR0,SIEVE,GR1
          CPL   GR0,=1
          JMI   SKIP
          ST    GR1,PRIMES,GR2
          LAD   GR2,1,GR2
SKIP      LAD   GR1,1,GR1
          CPL   GR1,N
          JMI   LOOP4
          JZE   LOOP4
          ST    GR2,M
          RET
;
N         DC    100
ROOTN     DC    10
SIEVE     DS    101
PRIMES    DS    50
M         DS    1
DUMPE     DC    -1
MSG       DC    'MESSAGE1'
          END
```

問題 11 の解答例

```
UPLOW     START
          IN    INAREA,INLENGTH
          CALL  ULSUB
          OUT   OUTAREA,INLENGTH
          RET
;
; ULSUB
; INAREA から始まる領域に格納されている文字列に含まれている
; すべての英字の大文字を X に変換して、
; すべての英字の小文字を x に変換して、
; それ以外の文字はそのままにすることによってできる文字列を
; OUTAREA から始まる領域に格納する。
ULSUB     LAD   GR2,0
LOOP      LD    GR1,INAREA,GR2
          CALL  ISUPPER
          CPL   GR3,=0
          JZE   SKIP1
          LD    GR1,='X'
SKIP1     CALL  ISLOWER
          CPL   GR3,=0
          JZE   SKIP2
          LD    GR1,='x'
SKIP2     ST    GR1,OUTAREA,GR2
          LAD   GR2,1,GR2
          CPA   GR2,INLENGTH
          JMI   LOOP
          RET
;
; ISUPPER
; GR1 の内容が英大文字かどうかを調べる。
; 英大文字の場合: GR3=1
; そうでない場合: GR3=0
ISUPPER   LAD   GR3,0
          CPL   GR1,='A'
          JMI   EXITU
          CPL   GR1,='Z'
          JPL   EXITU
          LAD   GR3,1
EXITU     RET
;
; ISLOWER
; GR1 の内容が英小文字かどうかを調べる。
; 英小文字の場合: GR3=1
; そうでない場合: GR3=0
ISLOWER   LAD   GR3,0
          CPL   GR1,='a'
          JMI   EXITL
          CPL   GR1,='z'
          JPL   EXITL
```

```
         LAD   GR3,1
EXITL    RET
;
INAREA   DS    256
INLENGTH DS    1
OUTAREA  DS    256
         END
```

問題 12 の解答例

```
DELSPC   START
         IN    INAREA,INLENGTH
         CALL  DSSUB
         OUT   OUTAREA,OUTLENGT
         RET
;
; DSSUB
; INAREA から始まる領域に格納されている文字列に含まれている
; すべての空白を削除することによってできる文字列を
; OUTAREA から始まる領域に格納する。
DSSUB    LAD   GR2,0
         LAD   GR3,0
LOOP     LD    GR1,INAREA,GR2
         CPL   GR1,=' '
         JZE   SKIP
         ST    GR1,OUTAREA,GR3
         LAD   GR3,1,GR3
SKIP     LAD   GR2,1,GR2
         CPA   GR2,INLENGTH
         JMI   LOOP
         ST    GR3,OUTLENGT
         RET
;
INAREA   DS    256
INLENGTH DS    1
OUTAREA  DS    256
OUTLENGT DS    1
         END
```

問題 13 の解答例

```
REGSPC   START
         IN    INAREA,INLENGTH
         CALL  RSSUB
         OUT   OUTAREA,OUTLENGT
         RET
;
; RSSUB
; INAREA から始まる領域に格納されている文字列に含まれている
; 連続する空白を 1 個だけに変換ことによってできる文字列を
; OUTAREA から始まる領域に格納する。
; GR4 は、連続する空白を処理中かどうかを示す。
; 処理中ならば 1、処理中でなければ 0。
RSSUB    LAD   GR2,0
         LAD   GR3,0
         LAD   GR4,0
LOOP     LD    GR1,INAREA,GR2
         CPL   GR1,=' '
         JZE   SKIP1
         LAD   GR4,0
         JUMP  SKIP2
SKIP1    CPL   GR4,=1
         JZE   SKIP3
         LAD   GR4,1
SKIP2    ST    GR1,OUTAREA,GR3
         LAD   GR3,1,GR3
SKIP3    LAD   GR2,1,GR2
         CPA   GR2,INLENGTH
         JMI   LOOP
         ST    GR3,OUTLENGT
         RET
;
```

```
INAREA   DS    256
INLENGTH DS    1
OUTAREA  DS    256
OUTLENGT DS    1
         END
```

問題 14 の解答例

```
CAPITAL  START
         IN    INAREA,INLENGTH
         CALL  CAPSUB
         OUT   OUTAREA,INLENGTH
         RET
;
; CAPSUB
; INAREA から始まる領域に格納されている文字列を
; キャピタライズすることによってできる文字列を
; OUTAREA から始まる領域に格納する。
; GR2 は、次の空白以外の文字が単語の先頭かどうかを示す。
; 単語の先頭ならば 1、そうでなければ 0。
CAPSUB   LAD   GR1,0
         LAD   GR2,1
LOOP     LD    GR0,INAREA,GR1
         CPL   GR2,=1
         JZE   SKIP1
         CPL   GR0,=' '
         JPL   SKIP2
         JMI   SKIP2
         LAD   GR2,1
         JUMP  SKIP2
SKIP1    CPL   GR0,=' '
         JZE   SKIP2
         LAD   GR2,0
         CALL  TOUPPER
SKIP2    ST    GR0,OUTAREA,GR1
         LAD   GR1,1,GR1
         CPA   GR1,INLENGTH
         JMI   LOOP
         RET
;
; TOUPPER
; GR0 の内容が英小文字ならば、それを英大文字に変換する。
TOUPPER  CPL   GR0,='a'
         JMI   EXIT
         CPL   GR0,='z'
         JPL   EXIT
         SUBL  GR0,=32
EXIT     RET
;
INAREA   DS    256
INLENGTH DS    1
OUTAREA  DS    256
         END
```

問題 15 の解答例

```
STOHEX   START
         IN    INAREA,INLENGTH
         CALL  S2HSUB
         OUT   OUTAREA,OUTLENGT
         RET
;
; S2HSUB
; INAREA から始まる領域に格納されている文字列を、
; 16 進数で表記された文字コードを空白で区切って
; 並べることによってできる文字列に変換して、
; それを OUTAREA から始まる領域に格納する。
S2HSUB   LAD   GR1,0
         LAD   GR2,0
LOOP     LD    GR0,INAREA,GR1
         CALL  C2HEX
         LAD   GR1,1,GR1
```

```
         CPA   GR1,INLENGTH
         JMI   LOOP
         ST    GR2,OUTLENGT
         RET
;
; C2HEX
; GR0 の下位 8 ビットの内容を 16 進数で表記したものと空白を、
; OUTAREA+GR2 から始まる領域に格納する。GR2 は 3 だけ加算される。
C2HEX    LD    GR3,GR0
         AND   GR3,=#00F0
         SRL   GR3,4
         CALL  IN2HEX
         LD    GR3,GR0
         AND   GR3,=#000F
         CALL  IN2HEX
         LD    GR3,=' '
         ST    GR3,OUTAREA,GR2
         LAD   GR2,1,GR2
         RET
;
; IN2HEX
; GR3 の内容を 1 桁の 16 進数に変換した結果を OUTAREA+GR2 番地に
; 格納して、GR2 をインクリメントする。
IN2HEX   CPL   GR3,=9
         JPL   SKIP1
         ADDL  GR3,=48
         JUMP  SKIP2
SKIP1    ADDL  GR3,=55
SKIP2    ST    GR3,OUTAREA,GR2
         LAD   GR2,1,GR2
         RET
;
INAREA   DS    256
INLENGTH DS    1
OUTAREA  DS    256
OUTLENGT DS    1
         END
```

問題 16 の解答例

```
PAREN    START
         IN    INAREA,INLENGTH
         CALL  PARSUB
         CALL  YESNO
         RET
;
; PARSUB
; INAREA に格納されている括弧列が
; 整形式括弧列かどうかを調べる。
; 整形式括弧列ならば GR0 に 1 を設定し、
; そうでなければ GR0 に 0 を設定する。
PARSUB   LAD   GR0,1
         LAD   GR2,0
         LAD   GR3,0
LOOP     LD    GR1,INAREA,GR2
         CPL   GR1,='('
         JPL   SKIP1
         JMI   SKIP1
         ADDL  GR3,=1
         JUMP  SKIP2
SKIP1    CPL   GR1,=')'
         JPL   SKIP2
         JMI   SKIP2
         SUBL  GR3,=1
         JMI   EXITNO
SKIP2    LAD   GR2,1,GR2
         CPA   GR2,INLENGTH
         JMI   LOOP
         CPL   GR3,=0
         JPL   EXITNO
         RET
EXITNO   LAD   GR0,0
         RET
```

```
;
; YESNO
; yes または no を出力する。
; GR0 が 0 以外ならば yes、0 ならば no。
YESNO    CPL    GR0,=0
         JZE    SKIP
         OUT    YES,YESLEN
         RET
SKIP     OUT    NO,NOLEN
         RET
;
INAREA   DS     256
INLENGTH DS     1
YES      DC     'yes'
YESLEN   DC     3
NO       DC     'no'
NOLEN    DC     2
         END
```

問題 17 の解答例

```
INTORPN  START
         IN     INAREA,INLENGTH
         CALL   I2RSUB
         OUT    OUTAREA,OUTLENGT
         RET
;
; I2RSUB
; INAREA に格納されている中置記法の式を逆ポーランド記法の式に
; 変換して、その結果を OUTAREA に格納し、その結果の長さを
; OUTLENGT に格納する。
I2RSUB   LAD    GR1,#FFFF
         PUSH   0,GR1   スタックに安全弁をプッシュ。
         LAD    GR2,0
         LAD    GR3,0
LOOP1    LD     GR1,INAREA,GR2
         CPL    GR1,=')'
         JZE    RPAREN
         CALL   ISVAR
         CPL    GR7,=1
         JZE    VAR
         CALL   ISOP
         CPL    GR7,=1
         JZE    STACK
         JUMP   SKIP
VAR      ST     GR1,OUTAREA,GR3
         LAD    GR3,1,GR3
         JUMP   SKIP
STACK    PUSH   0,GR1
         JUMP   SKIP
RPAREN   POP    GR1
         CPL    GR1,=#FFFF
         JZE    EXIT   安全弁をポップした場合は処理終了。
         ST     GR1,OUTAREA,GR3
         LAD    GR3,1,GR3
SKIP     LAD    GR2,1,GR2
         CPA    GR2,INLENGTH
         JMI    LOOP1
LOOP2    POP    GR1
         CPL    GR1,=#FFFF
         JMI    LOOP2  安全弁に到達するまでポップ。
EXIT     ST     GR3,OUTLENGT
         RET
;
; ISVAR
; GR0 の内容が変数名（英小文字）かどうかを調べる。
; 変数名の場合:   GR7:1
; そうでない場合: GR7:0
ISVAR    LAD    GR7,0
         CPL    GR1,='a'
         JMI    EXITVA
         CPL    GR1,='z'
         JPL    EXITVA
```

```
         LAD   GR7,1
EXITVA   RET
;
; ISOP
; GR0 の内容が演算子かどうかを調べる。
; 演算子の場合:   GR7:1
; そうでない場合: GR7:0
ISOP     LAD   GR7,1
         CPL   GR1,='+'
         JZE   EXITOP
         CPL   GR1,='-'
         JZE   EXITOP
         CPL   GR1,='*'
         JZE   EXITOP
         CPL   GR1,='/'
         JZE   EXITOP
         CPL   GR1,='%'
         JZE   EXITOP
         LAD   GR7,0
EXITOP   RET
;
INAREA   DS    256
INLENGTH DS    1
OUTAREA  DS    256
OUTLENGT DS    1
         END
```

問題 18 の解答例

```
ADDDEC   START
         IN    INAREA1,NUMLEN1
         IN    INAREA2,NUMLEN2
         CALL  ADSUB
         OUT   OUTAREA,NUMLEN3
         RET
;
ADSUB    CALL  SETMAX
         CALL  STR2NUM1
         CALL  STR2NUM2
         CALL  ADD
         CALL  NUM2STR
         RET
;
; SETMAX
; NUMLEN1 と NUMLEN2 とを比較して、
; 大きいほうを NUMMAX に格納する。
SETMAX   LD    GR0,NUMLEN1
         CPL   GR0,NUMLEN2
         JPL   SKIP1
         LD    GR0,NUMLEN2
SKIP1    ST    GR0,NUMMAX
         RET
;
; STR2NUM1, STR2NUM2
; 10 進数を、それぞれの桁があらわしている
; 整数の列（ただし順序は逆）に変換する。
STR2NUM1 LAD   GR1,0
         LD    GR2,NUMLEN1
         LAD   GR2,-1,GR2
LOOP1    CPA   GR1,NUMLEN1
         JMI   SKIP2
         LD    GR0,=0
         JUMP  SKIP3
SKIP2    LD    GR0,INAREA1,GR2
         SUBL  GR0,=48
SKIP3    ST    GR0,NUM1,GR1
         LAD   GR1,1,GR1
         LAD   GR2,-1,GR2
         CPA   GR1,NUMMAX
         JMI   LOOP1
         RET
;
STR2NUM2 LAD   GR1,0
```

```
          LD    GR2,NUMLEN2
          LAD   GR2,-1,GR2
LOOP2     CPA   GR1,NUMLEN2
          JMI   SKIP4
          LD    GR0,=0
          JUMP  SKIP5
SKIP4     LD    GR0,INAREA2,GR2
          SUBL  GR0,=48
SKIP5     ST    GR0,NUM2,GR1
          LAD   GR1,1,GR1
          LAD   GR2,-1,GR2
          CPA   GR1,NUMMAX
          JMI   LOOP2
          RET
;
; NUM2STR
; 10 進数のそれぞれの桁があらわしている整数の列
; （ただし順序は逆）を 10 進数に変換する。
NUM2STR   LAD   GR1,0
          LD    GR2,NUMLEN3
          LAD   GR2,-1,GR2
LOOP3     LD    GR0,NUM3,GR2
          ADDL  GR0,=48
          ST    GR0,OUTAREA,GR1
          LAD   GR1,1,GR1
          LAD   GR2,-1,GR2
          CPA   GR1,NUMLEN3
          JMI   LOOP3
          RET
;
; ADD
; 整数の列に変換された二つの 10 進数を加算する。
; GR2 は、桁上がりする数値（0 または 1）。
ADD       LAD   GR1,0
          LAD   GR2,0
LOOP4     LD    GR0,NUM1,GR1
          ADDL  GR0,NUM2,GR1
          ADDL  GR0,GR2
          CPL   GR0,=10
          JMI   SKIP6
          LAD   GR2,1
          SUBL  GR0,=10
          JUMP  SKIP7
SKIP6     LAD   GR2,0
SKIP7     ST    GR0,NUM3,GR1
          LAD   GR1,1,GR1
          CPA   GR1,NUMMAX
          JMI   LOOP4
          CPL   GR2,=1
          JMI   SKIP8
          ST    GR2,NUM3,GR1
          LAD   GR1,1,GR1
SKIP8     ST    GR1,NUMLEN3
          RET
;
INAREA1   DS    256
INAREA2   DS    256
NUM1      DS    256
NUM2      DS    256
NUM3      DS    256
OUTAREA   DS    256
NUMLEN1   DS    1
NUMLEN2   DS    1
NUMMAX    DS    1
NUMLEN3   DS    1
          END
```

問題 19 の解答例

```
ROMAN     START
          IN    INAREA,INLENGTH
          CALL  LOW2UP
          CALL  ROMSUB
```

```
          CALL  INT2DEC
          OUT   DECIMAL,DECLEN
          RET
;
; ROMSUB
; INAREA2 から始まる領域に格納されているローマ数字を整数に
; 変換した結果を GR0 に格納する。
; GR7 は、減算則が適用される可能性のある、
; ひとつ前の文字の整数を示す。
; GR7 が 0 の場合、減算則が適用される可能性はない。
; 現在の文字の整数がひとつ前の文字の整数よりも大きいならば、
; 減算則を適用する。
ROMSUB    LAD   GR0,0
          LAD   GR7,0
          LAD   GR1,0
LOOP1     LD    GR2,INAREA2,GR1
          CALL  DIG2VAL
          ADDL  GR0,GR3
          CPL   GR7,=0
          JZE   SKIP
          CPL   GR7,GR3
          JPL   SKIP
          JZE   SKIP
          SUBL  GR0,GR7
          SUBL  GR0,GR7
          LAD   GR3,0
SKIP      LD    GR7,GR3
          LAD   GR1,1,GR1
          CPA   GR1,INLENGTH
          JMI   LOOP1
          RET
;
; DIG2VAL
; GR2 の内容をローマ数字の文字とみなしたときに、
; その文字があらわしている整数を GR3 に格納する。
DIG2VAL   LAD   GR4,0
LOOP2     LD    GR5,ROMDIGIT,GR4
          CPL   GR2,GR5
          JZE   EXIT1
          LAD   GR4,1,GR4
          CPL   GR4,ROMLEN
          JMI   LOOP2
EXIT1     LD    GR3,ROMVALUE,GR4
          RET
;
; LOW2UP
; INAREA から始まる領域に格納されている文字列の中に
; 含まれているすべての英小文字を英大文字に変換した文字列を、
; INAREA2 から始まる領域に格納する。
LOW2UP    LAD   GR1,0
LOOP3     LD    GR0,INAREA,GR1
          CALL  TOUPPER
          ST    GR0,INAREA2,GR1
          LAD   GR1,1,GR1
          CPA   GR1,INLENGTH
          JMI   LOOP3
          RET
;
; TOUPPER
; GR0 の内容が英小文字ならば、それを英大文字に変換する。
TOUPPER   CPL   GR0,='a'
          JMI   EXIT2
          CPL   GR0,='z'
          JPL   EXIT2
          SUBL  GR0,=32
EXIT2     RET
;
; INT2DEC
; GR0 の内容を 4 桁の 10 進数に変換して、
; DECIMAL から始まる領域にその結果を格納する。
INT2DEC   LAD   GR1,0
LOOP4     CALL  DIVIDE
          ADDL  GR2,=48
```

```
          ST    GR2,DECIMAL,GR1
          LAD   GR1,1,GR1
          CPA   GR1,=3
          JMI   LOOP4
          ADDL  GR0,=48
          ST    GR0,DECIMAL,GR1
          RET
;
; GR0 を WEIGHT+GR1 で除算して、商を GR2 に、余りを GR0 に格納する。
DIVIDE    LAD   GR2,0
LOOP5     CPA   GR0,WEIGHT,GR1
          JMI   EXIT3
          LAD   GR2,1,GR2
          SUBL  GR0,WEIGHT,GR1
          JUMP  LOOP5
EXIT3     RET
;
INAREA    DS    256
INLENGTH DS     1
INAREA2   DS    256
ROMDIGIT DC     'IVXLCDM'
ROMVALUE DC     1,5,10,50,100,500,1000
ROMLEN    DC    7
DECIMAL   DS    4
DECLEN    DC    4
WEIGHT    DC    1000,100,10
          END
```

問題 20 の解答例

```
POWER     START
          LD    GR1,A
          LD    GR2,B
          CALL  POWSUB
          ST    GR0,P
          DMEM  MSG,A,DUMPE
          RET
;
POWSUB    CPL   GR2,=0
          JZE   BASIS
          PUSH  0,GR2
          LAD   GR2,-1,GR2
          CALL  POWSUB
          POP   GR2
          CALL  MULTIPLY
          RET
BASIS     LAD   GR0,1
          RET
;
; MULTIPLY
; GR0 と GR1 とを乗算して、
; 結果を GR0 に格納する。
MULTIPLY LAD    GR4,0
          LD    GR3,GR0
LOOP      CPL   GR3,=0
          JZE   EXIT
          ADDL  GR4,GR1
          LAD   GR3,-1,GR3
          JUMP  LOOP
EXIT      LD    GR0,GR4
          RET
;
A         DC    3
B         DC    5
P         DS    1
DUMPE     DC    -1
MSG       DC    'MESSAGE1'
          END
```

参考文献

[JITEC,2012] 「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 2.2」、情報処理技術者試験センター、2012。

[渋谷,2001] 渋谷正行、大内誠、『CASL II プログラミング』、アイテック、2001。

[福嶋,2012] 福嶋宏訓、『基本情報技術者試験 CASL II 完全合格教本・第 3 版』、新星出版社、2012、ISBN 978-4-405-04644-3。

索引