アーキテクチャ　第６回　追加メモ　アセンブラプログラム(1)

条件分岐とフラグレジスタの仕組

授業の資料の中では、ほぼ約束事のように、CPA 命令の次に条件分岐命令（JPL や JMI、JZE、JNZ など）を続けて使うとしているが、実は必ずしもそうとはかぎらない。 命令の一覧表や教科書の説明は、CPA プラス条件分岐という定型以外の使い方があることを示唆している。 そのあたりを追加説明する。 但し、ごちゃごちゃするのが面倒な学生は、ここの説明は忘れて構わない。 授業での定型は、CPA 命令プラス条件分岐ということにしておく。 また、試験の解答などで、そうでない（たとえば CPA を端折った）プログラムを示す場合は、そのことを明記してほしい。

まず、フラグレジスタ FR の動作について、きちんと説明してみよう。 フラグレジスタにある3つのビットはそれぞれ、

SF（値の符号ビット＝値が2の補数で見た時に正か0なら0、負なら1）、
ZF（ゼロビット＝値のビットがオールゼロなら1、そうでなければ0）、
OF（オーバーフロービット＝オーバーフローを起こしていれば1）

となっている。 フラグレジスタの値を決定するのは、授業の説明では CPA 命令ということになっているが、実は算術論理演算ユニット（ALU）の出力の値を取り込んでいて、どの命令の時に取り込むかは命令の一覧表の右端の欄の〇によって表示されている。 CPA 命令だけではなく、ADDA 命令などの算術演算命令やシフト命令、更には何とロード命令でもフラグビットが立つのである。 これはロード命令の時に（メモリから汎用レジスタに転送される途中で）ALU を通過するためだと思われる。

FR のビットをセットするような命令を実行すれば必ず、ビットは上書きされるので、基本的には条件分岐命令（JPL 命令など）の直前の命令を実行した時の ALU の状況（SF, ZF, OF）が記録されて、次の条件分岐に反映される。 だから、条件分岐命令の直前で CPA 命令によって比較して結果を FR に書き込むのが望ましい（間違いない）、と言えるだろう。（もしかしたらその前の命令で欲しい状況の FR がセットされていて、CPA 命令が無駄な時もあるかも知れないが）

FR のビットと条件分岐の関係は、命令の表に書かれている通りで、若干ややこしく書かれている。 これは、書き方がややこしいだけで、原理は簡単なので、原理を理解すると良いかも知れない。

JPL 命令は、正の時だけ分岐し、0の時と負の時は分岐しない。 これは SF と ZF の組合せで判別できる。 符号ビット SF が0つまり値が正かゼロであって、かつゼロビット ZF が0つまり値がゼロでない、という条件になる。 だから、0ではなくて正の時だけ分岐することになる。

JMI 命令は、負の時だけ分岐し、0の時と正の時は分岐しない。 これは SF だけで判別できる。 符号ビット SF が1つまり値が負の時だけ、分岐する。 値が0の時には、符号ビットは（2の補数では）0になる。

JZE 命令（ゼロのときだけ分岐）と JNZ 命令（ゼロ以外のときだけ分岐）は、どちらもゼロビット ZF だけで判別できる。

つまりまとめると、2の補数表現をしたときの符号ビットの振舞いと、JPL 命令に望まれる振舞い（正のときだけジャンプし、0と負の時はジャンプしない）とをつき合わせてみると、どうしても SF が0でかつ ZF が0、となってしまうのである。
なお、オーバーフローについては、JOV 命令は FR のオーバーフロービットを見て、1であればジャンプする。

ここで、CPA 命令についてもう一度考えてみる。 CPA r1, r2 となっている場合、命令表の動作記述によれば、「(r1)と(r2)の算術比較を行い、比較結果によって FR に次の値を設定する」、と書かれ、表には (r1)>(r2)なら 0, 0 といったようになっている。 これは当たり前だが、(r1)−(r2)の引き算の結果が正かゼロ・負か、になっている。 内部回路としては、CPA 命令は算術減算命令 SUBA と同じことをやっているわけなのだが、1つだけ違うのは、引き算の結果の値を汎用レジスタに書き戻さないのである。 SUBA r1, r2 では引き算の結果 r1−r2 の値を r1 に書き込む（同時に FR も更新する）のだが、CPA r1, r2 では FR を更新するだけであって引き算の結果を r1 に書き込むことをしない。 CPA 命令では引き算をした結果の値には興味が無い。 符号（FR のビ

ット）にだけ興味がある。 そこが SUBA との違いになる。

では、if 文での條件指定との関連を考えてみよう。 もし if (a>b) であれば、a と b を CPA して、その結果で JPL(Jump Plus) すればよい。 では、もし if (a>=b) であったらどうするか？ 少し自分で考えてみてほしい。

2通り考えられる。

1つ目は、条件を逆にして考える方法である。 元の if (a>=b) { AA } else { BB } という構造を、 if (a<b) { BB } else { AA } と書き換える。 AA と BB の位置を換えたので、同じことになるはずである。 更に if (b>a) {BB} else {AA} と書き換えれば、単純に、b を a と比較して、その結果で JPL すればよい。

2つ目は、条件はそのままで、a>b の場合と a=b の場合を順番に処理する方法である。 a と b とを CPA すると SF と ZF がセットされるので、JPL 命令と JZE（Jump on Zero）命令を続けて実行する。 命令表を見ると分岐命令は FR の値をセットしない（〇がついていない）ので、JPL 命令の後に続けて JZE 命令を連続して実行しても FR の値は元の侭である。だから JZE も正しい（望ましい）FR の値を参照する。 具体的には、もし a=b であれば、CPA 命令が FR を SF=0, ZF=1 にセットし、JPL 命令では分岐しないで次の命令に進み、次の JZE 命令で条件成立し分岐する。

```
CPA    a, b     本当は a, b には汎用レジスタ等のオペランドが書かれる
JPL    Label1   正の時 Label1 へ飛ぶ、0か負なら次の命令へ進む
JZE    Label1   0の時 Label1 へ飛ぶ、負なら次の命令へ進む、正のときは前の JPL 命令で Label1 へ飛んでいる
                負の場合の処理が続く
```

アセンブラの動作・仕組

まず、いくつかの用語の意味を整理しておこう。

アセンブリ言語またはアセンブラ言語　ニーモニックコードでプログラムを書くときの書き方（プログラミング言語）

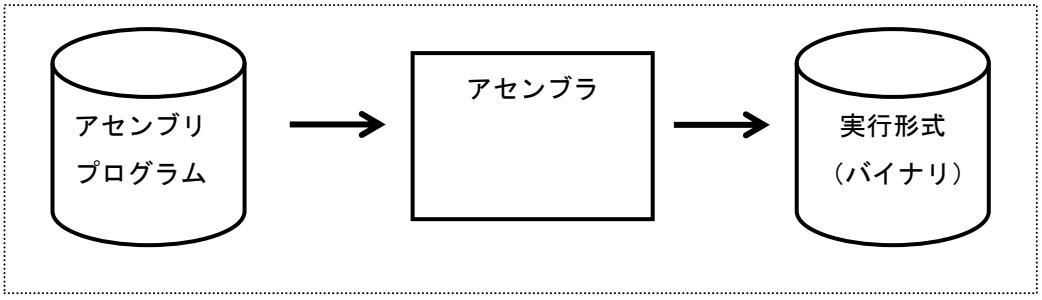
アセンブリプログラムまたはアセンブラプログラム　アセンブラ言語で書かれたプログラム

アセンブラ　アセンブラプログラムを入力として、機械命令（0/1 の列）を出力するプログラム。コンパイラと同様の役割

たとえば、

```
LD    GR3, 100
```

はアセンブリ言語で書かれた文で、アセンブリプログラムの 1 行になる。機械命令は 16 進で 1030 0100 のようなもの。



アセンブラは、原則としてアセンブラプログラムの 1 行 1 行を1つの命令であると解釈して、機械命令を生成する。 つまり、1 行に 1 命令が対応する。 これは Java や C 言語などの高級言語との大きな違いである。 Java などでは 1 行という単位は意味を持たず、たとえば if (x>y) { z=x-y; } else { z=y-x; } のように改行を入れずに書いても文としては正しい。 他方、アセンブラ言語では行が意味を持っている。

命令（LD とか ST とか ADDA とかの OP 部分）は、アセンブラプログラムではニーモニックコードで書き、それをアセンブラが対応する機械命令の 0/1 パターンに変換する。

オペランドの欄は、GR0～GR7 と書いてあれば汎用レジスタであると判断する（予約語である。 この文字列は一般のラベルとしては許されない）。 また数字だけ書かれているもの、もしくは１６進数を示す#記号の後に１６進の数字が書かれているもの

(#3A5D など)はアドレスの値であると判断する。 それ以外の英大文字から始まる英数字の列(大文字のみ、かつ8文字まで)は「ラベル」と判断する。 ラベルは Java での変数名のようなもので、メモリ上の領域(変数の場所)を指す名前になる。 ラベルはアセンブラによってアドレスに置き換えられる。 (この仕組は後でもう少し説明する)

アセンブラ言語の 1 行の書き方は、命令表のあるプリントの6ページの 2.1 節(上半分)に書かれている。
アセンブラプログラムのそれぞれの行の内容は、同じページの 2.2 節にあるとおり、機械命令(命令として書いているもの、LD とか ST とか)か、あとはアセンブラを制御するような「アセンブラ命令」と呼ばれる行か、マクロ命令(我々は使わない)か、のどれかである。
アセンブラ命令とは、アセンブラの行であるが実行の対象でない命令(行)であって、アセンブラに対して指示を与える。具体的には、1つのアセンブラプログラムの先頭と最後を示す START 文と END 文、および命令以外のメモリ領域(変数や定数)を確保するための DS 文と DC 文がある。
DS (Define Storage) 文は、変数のためにメモリ領域を確保する。 引数に、確保したい語数を書くことができるので、たとえば5語分の領域を確保したければ、 DS 5 と書けばよい。 通常先頭にラベルをつけ、そのラベルは領域の先頭のアドレスに対応する。 つまり、 L1 DS 5 と書くと、ラベル L1 はこの5語からなる領域の先頭のアドレスに対応し、領域は L1, L1+1, L1+2, L1+3, L1+4 の5つの語(=16 ビットの領域)から成る。
DC (Define Constant) 文は、定数のためにメモリ領域を確保する。 正確には 1 語の領域を確保し、指定した初期値を設定する。たとえば、 L2 DC 9 と書くと、領域 1 語分が確保され、その初期値を 9 とし、ラベル L2 が付けられる。 この領域はプログラム実行中に書き込みが可能であり、変数として使うことができるので、その意味で用途は定数に限定してはいない。

ラベルの使い方について、考えてみる。 ラベルの考え方は、アドレスを書く代わりにラベルを書くことである。 アドレスの値を直接に書く(プログラマが指定する)代わりに、アセンブラが自動的に決めてくれる。 たとえば

```
        LD    GR3, HENSUU
        …
        …
HENSUU  DS    1        ← 命令を上から順に配置して行った結果、ここの場所が 351 番地になったとする
```

のように、ラベル HENSUU の付いたメモリ上の場所(1 語の領域)が、アドレス 351 番地になっていたとすると、ラベル HENSUU に対応するアドレスは 351 番地であると言う。 この 351 番地という数は、アセンブラがプログラムをメモリ上に上から順に配置して行った結果、HENSUU の場所がこのアドレスになったということであり、プログラマが特に指定したわけではない。 アセンブラが決めたのである。 もし、プログラムに加筆して命令が1つ増えたら、HENSUU の場所は1つ次に繰り下がって 352 番地になるが、再アセンブルすればアセンブラは HENSUU が 352 番地であると認識する。
そのようにして決まったラベル HENSUU とアドレス 351 番地の対応を用いて、アセンブラは LD 命令のオペランド部分に、具体的なアドレス 351 を埋める。つまり

```
        LD    GR3, 351
```

に書き直す。 これを行うには、最初にプログラムを最初から最後まで全部読んで、すべてのラベルについて対応するアドレス値を確定し(フェーズ1と呼ぶ)、次に、もう一度プログラムを最初から最後まで全部読んで、ラベルが出てくる命令を見つけてそのラベルをアドレス値に置き換える(フェーズ2と呼ぶ)。 このように2回全体を読み直すやり方を、2フェーズのアセンブラと呼ぶ。 2フェーズの処理は、アセンブラだけでなく、高級言語のコンパイラでも発生する。 また、工夫して1回だけで済ませる手法も開発されているので、興味ある人は調べてみると良かろう。

オペランドとしてメモリアドレスを直接指定するのではなく、ラベルを使うメリットは、次のようなものである。

第1に、プログラミングの途中で変数の置き場所が未だ決まっていない時に、変数名（ラベル）で書いておいて、あとから変数を置く場所を DS なり DC なりで確保してラベルをつけることができる。
第2に、ラベルで名前付けすれば、プログラムが分かりやすくなる。#32A8 番地へストアというよりは、変数 KEKKA にストアという方が意味が分かりやすい。
第3に、他の高級言語（C 言語など）のプログラムとつないで使う時に、高級言語側で変数として名前を与えたものを参照したり、逆にアセンブラプログラム内で名前付けした変数を高級言語側で参照したりできる。現在のアセンブラプログラムの使い方が、高級言語プログラムのごく一部をアセンブラ化して高速化したりハードを直接制御したりする場合が多いので、お互いに参照できるのが望ましい。
まだ、その他にもメリットがあるかも知れないので、考えてみてほしい。
では、デメリットはあるだろうか？ どうしてもアドレスで指定したい場合もあるかも知れない。 たとえばハードウェアの制御をする時に、アドレスがあらかじめ決まっているというケースはあるだろう。 これはアドレスの値を直接指定せざるを得ない。 しかしそれでも、プログラムの読みやすさ（あとから他の人が見て、何をやっているか一目瞭然に分かる）のためには、ラベルによって名前付けした方がよい。 COMET-II ではできないが、他のアセンブラでは「ラベルに値（アドレス値）を設定する」というアセンブラ命令が使えるケースもある。

ジャンプ命令（条件分岐を含む）のとび先として指定するラベルの意味は、今まで見た変数に付けるラベル（＝変数名）の考え方と同じである。 つまり、ラベルはメモリ上の（領域の）アドレスに置き換わるものである。 ジャンプ命令のとび先は、「何番地へ飛べ」という、そのとび先のアドレスであるから、ラベル＝アドレスである。 ラベルを付けるのは、変数の場合は DS 命令や DC 命令による領域確保と同時に行っていたが、ジャンプのとび先の場合は、とび先の命令の行のラベル欄に書く。 これによって、この命令の置かれるアドレスがラベルに対応づけられ（フェーズ1）、ジャンプ命令に書かれたラベルがこのアドレスに置き換えられる（フェーズ2）。

LAD 命令を使うか使わないか

定数を使いたい時、2つの方法がある。 1つは LAD 命令で作り出す方法、もう1つはメモリ中にあらかじめ定数を用意しておく（上記の DC 命令を使う）方法である。 下記に例を示す。

```
      LAD   GR3, 5                          LD   GR3, CONST5
                                            …
                                    CONST5  DC   5
```

どちらがよいか、どちらが得か、自分でも考えてみてほしい。

実行時間　どちらもメモリアクセス1回で、大きな違いはないと思う

メモリ消費量　命令の長さは LAD も LD も2語である。 LD の方が定数をメモリに取った分だけ余分にメモリを食う

プログラムを読んだ時のわかりやすさ　大きな違いは無いと思う

書変えやすさ　たとえば5を7にしたいという場合である。 ソースレベルでの書変え（ソースを書変えて再アセンブル）は大差ないだろう。 バイナリレベルで書変えるときは、DC で書いてあるとそのアドレスの場所の定数5を書き直すので、分かりやすいだろう。 LAD の場合 LAD 命令の場所を見付け、そのオペランド（2語目）の値を書き直すことになる。 バイナリレベルでの書変えは非常に特殊なケースで、通常は想定しなくてよいと思う。

あまり差はないというのが、山内の感触である。 この授業ではどちらで書いても良いが、LAD と LD の使い方を間違えないこと。
LD　GR3, 5　はまずい。