はじめに

- 今回は、命令セットについての講義の最終回です。
  - OPコードに指定する演算操作の種類について、順に説明します。
  - OPコードに指定できる演算は、基本的にハードウェアで行います。つまり、その演算を行うための演算器を備えている、ということです。

- 教科書の対応範囲は、以下の通りです。
  - 2.2.6項（p.58～64）

(1) 算術演算命令

- 2項演算については、加算、減算、乗算、除算/剰余の四則演算が基本です。
- 単項演算については、絶対値や符号反転などの命令があります。
- データの表現方法が異なるので、同じ種類の演算であっても、固定小数点用と浮動小数点数用とで独立した命令が用意され、さらに、データサイズ（ビット数）によっても異なる命令が用意されています。
  - 従って、加算命令といっても、何種類もの加算命令が用意されています。
- このほか、コンピュータによっては平方根や対数などの初等関数、三角関数を求める命令を備えているものもあります。

- 1命令でも、当然、複雑な演算ほど実行するのに時間がかかります。
- 通常、ワードサイズの固定小数点数に対する加算/減算が1サイクルで終了するようにハードウェを設計します。
  - 1サイクルというのは、プロセッサの動作周期です。動作周波数の逆数で、2GHzのプロセッサの1サイクルは$1/(2\times10^9)$=0.5ns（ナノ秒）になります。
  - 固定小数点数の乗算で数サイクルかかります。
  - 固定小数点数の除算の場合は数十サイクルです。

- 演算結果を求めるだけでなく、同時に条件コード（condition code）を設定する場合もあります。
  - 条件コードで以下のような状態を記憶します。
    - 演算結果が負になったか否か（符号ビット）
    - 演算結果が0となったか否か
    - 最上位桁からの桁上げ（carry）
    - オーバフローが発生したか否か

    など
  - 条件コードは条件分岐命令（後述）で使用します。

(2) 比較命令/関係演算命令

- 比較命令は減算命令とほぼ同じ機能の命令なのですが、条件コードを設定するだけで、演算結果は出力しません（厳密に言えば、演算結果を捨てています）。
- コンピュータによっては、比較だけでなく、関係演算までを行う命令を備えているものもあります。
  - 比較するだけでなく、「>」、「≦」、「≠」などの条件が成立するかどうかの判定までを行います。
  - 真（1）か偽（0）かを出力します。

(3) 論理演算命令

- AND、OR、NOT、XOR(exclusive-OR)などの論理演算をビット毎に独立に行います。

  8ビットの固定用数点数に対する演算例:

  – AND演算
    - 10110111 and 00111100 → 00110100
      – 10110111の中央の4ビットを取り出すために、00111100とandをとる。取り出すビット位置(または消し去るビット位置)を指定するデータ(00111100)をマスク(mask)という。
      – 10110111の両端の2ビットずつを、強制的に0にしている、と考えることもできる。

- OR演算
  - 10110111 or 11000011 → 11110111
    - 10110111の両端の2ビットずつを、強制的に1にしている。
- NOT演算
  - 10110111 → 01001000
- XOR演算
  - 10110111 xor 01100100 → 11010011
    - 10110111の青色のビットのみを反転するために、01100100とxorをとる。
  - 10110111 xor 11111111 → 01001000
    - NOTと同じ。従って、XOR命令があればNOT命令は要らない。しかし、XOR命令はソースオペランドが2個必要で命令語長が長くなり、プログラムサイズの増大につながる。
      - » XOR命令だけにするか、それともXOR命令とNOT命令の両方を備えるかは、命令セットの設計者の判断による。

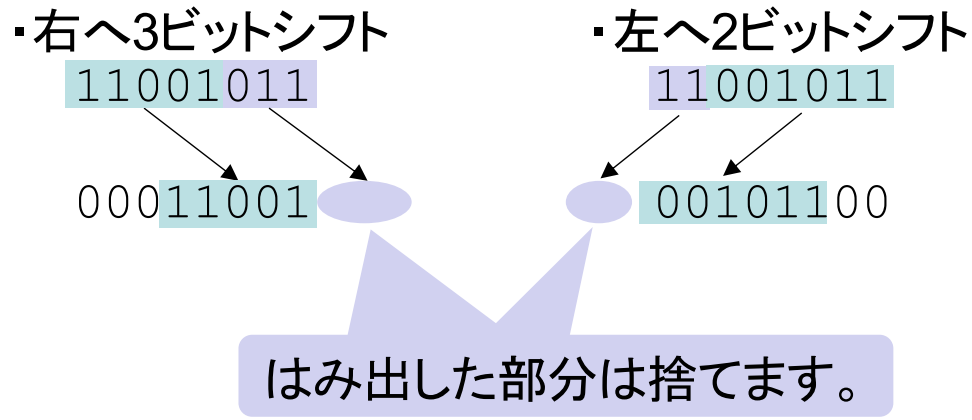
(4) 型変換命令

- データの表現方法に関する変換を行う命令
  - 固定小数点数→浮動小数点数　の変換
  - 浮動小数点数→固定小数点数　の変換

【注意】 教科書には10進数に関する記述がありますが、現在はほとんど無視してよいでしょう。昔は10進数の1桁を4または8ビットの2進数で表現して、これを並べるBCD表現（教科書のp.92-93）というのがありましたが、現在は使用しません。

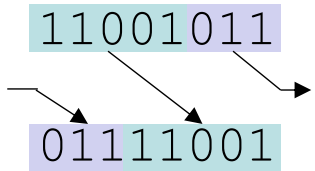
(5) シフト命令

- シフト（shift）演算
  - ビット列を右または左方向へずらす操作
    - 例） 8ビットの固定小数点数の場合

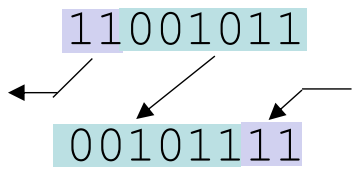


- 循環シフト（rotate）演算
  - ビット列をシフトしますが、はみ出した部分を反対の端から挿入します。
    - 例） 8ビットの固定小数点数の場合

・右へ3ビットローテート



・左へ2ビットローテート



シフト命令演算の算術的意味

- “左へ1ビットシフトする” ＝ “2倍する”
  - 例） 00000101 (=5) → 0001010 (=10)
- “右へ1ビットシフトする” ＝ “2で割る”
  - 例） 00000101 (=5) → 00000010 (=2)
    余りの1は失われる

よって、

- “左へkビットシフトする” ＝ “2^k倍する”
- “右へkビットシフトする” ＝ “2^kで割る”
  - 通常、シフト命令は1サイクルで実行可能なので、“2^k倍する”/“2^kで割る”のなら、乗算/除算命令を使用するよりもシフト命令を用いる方がかなり高速

- 前ページの内容は正の数に対してシフト演算を行なった場合の話です。

- 負の数の場合でもうまくいくでしょうか？
  - 正か負かに関わらず（つまり、プログラムで正か負かを調べることなく）、“2^k倍する”/“2^kで割る”目的で使用できると便利です。

- 負の数の左シフト
  - 正の数aに対して2の補数で負の数を表す場合の$-a$は、nビットの固定小数点数で2^n-aと表される。
  - これを通常の2進数として左へ1ビットシフトすると、

$$(2^n-a)\times 2 = 2^{n+1}-2a = 2^n+(2^n-2a)$$

これは左端からはみ出た分なので、失われる。

  - 結局、シフト結果は(2^n-2a)となり、これは$-2a$を表す。
  - したがって、負の数の場合でも、1ビット左へシフトすることは、2倍することを意味する。
  - よって、“左へkビットシフトする” ことは、その対象が正か負かに関わらず、“2^k倍する” ことと同じである。

- 負の数の右シフト
  - $-a$（a>0）の表現である2^n-aを通常の2進数として右へ1ビットシフトすると、
    $$(2^n-a)\div 2 = 2^{n-1}-a/2$$
  - これでは正の数になってしまって、$-a/2$にはなってくれません。
  - そこで、シフト結果に2^{n-1}を加えて（つまり、左端から1を挿入して）$2^{n-1}-a/2+2^{n-1}=2^n-a/2$ とすると、$-a/2$の表現になります。
  - したがって、負の数の場合は左から1を挿入することによって、1ビット左へシフトすることは、2で割ることを意味する。
  - よって、“左へkビットシフトする” ことで、負の数の場合でも、“2^kで割る”という演算を行うようにできる。

- 以上より、シフト命令としては、算術シフトと論理シフトの2種類を設けます。

	左シフト	右シフト
算術シフト	右端から0を挿入	左端から符号ビットの値を挿入
論理シフト	右端から0を挿入	左端から0を挿入

正の数の時には0を、負の数の時には1を、それぞれ挿入する、ということは、符号ビットの値をコピーして挿入するのと同じことです。

同じ操作ですから、左シフトについては、算術シストと論理シフトに命令の種類を分ける必要はないかもしれません。

- ところで、皆さんは、負の数の割り算はできますか？
  - いや、大学生ですからできるはずなのですが、これから話題にすることについては、ひょっとしたら今までに考えたことなどないかもしれません。

- では、問題です。(−5) ÷ 2 の商と余りを答えてください。

(−5)÷2 の商と余りは？

- 1つの答えは、商が−2、余りが−1、です。

- でも、答えはもう1つあります。商が−3、余りが1、です。
- 算術右シフトしたとき、どうなるのかを確認しておきましょう。

  1ビット、右へシフト

  – 11111011 (=−5) → 11111101 (=−3)
  – つまり、余りが正になるように商を求めています。

(6) ビット操作命令

- 特殊レジスタ（各種の制御ビットなどを集めた特殊な用途のレジスタ）に対して、指定したビットをセット（1にする）/リセット（0にする）/反転する、などの操作を行う。
- 一般のデータに対しては、論理演算命令を用いて、指定したビットのセット/リセット/反転などが可能。

(7) データ転送命令

- 以下の装置間でデータを転送（コピー）する。
  - レジスタから別のレジスタにデータをコピーする。
  - レジスタからメモリにデータをコピーする。
    - この場合のデータ転送命令を特に「ストア（store）命令」という。
  - メモリからレジスタにデータをコピーする。
    - この場合のデータ転送命令を特に「ロード（load）命令」という。
  - メモリから、メモリの別のアドレスにデータをコピーする。
    - 実際には、メモリからメモリへのコピーは一度にはできない（メモリは、一度には、読むか書くかのいずれかのみ）。メモリから読み出してプロセッサ内のあるレジスタにコピーした後、そのレジスタの値をメモリの別の番地に書き込む。メモリアクセスが2回必要であり、上記のロード/ストア命令に比べて倍程度の実行時間を要する。

レジスタ間でのデータ転送命令

- レジスタ間でデータ転送を行う場合は、わざわざ転送（move）命令を設けなくても、演算命令で代用できる。
  - （例）レジスタ5の値を、レジスタ8にコピーする。
    - 転送（move）命令:　　mov　r5, r8　　（r5 → r8）
    - 加算（add）命令:　　add　r5, 0, r8　　（r5 + 0 → r8）
    - 論理和（or）命令:　　or　r5, 0, r8　　（r5 or 0 → r8）
  - 特殊な場合として、NOP命令も定義できる。
    - NOP（No OPeration）命令:　何もしない命令
    - （例）　mov　r0, r0　　（r0 → r0）
      - 加算命令や論理和命令でこの転送命令を実現しても良い。

（注）　以上のようにして、命令の種類を減らすことにより物理的なハードウェア量（ゲート数）やハードウェア設計の複雑度を減らす工夫をする。

符号拡張

- 異なるサイズの装置間で固定小数点数を転送する時には、符号について配慮が必要です。
  - 以下、32ビットのプロセッサにおいて、32ビットのレジスタとメモリのある1バイトの領域とでデータを転送する場合を例に考えます。
    - ただし、これはデータの転送だけでなく、演算器内でも、データのサイズを変更するときには必要となる配慮です。
- ストア命令
  - レジスタ内の4バイトのデータのうち、最下位バイトの値をメモリに書き込みます。上位3バイトのデータは失われます(レジスタには残ったままですが)ので、注意が必要です。

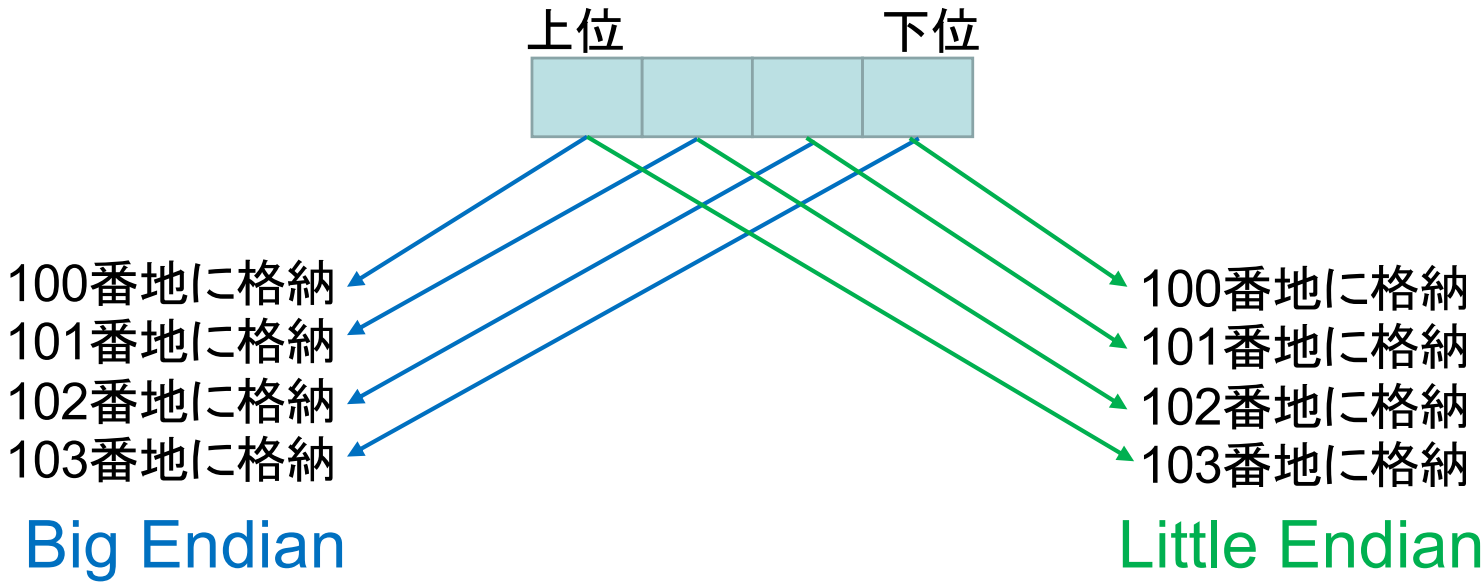
余談ですが、試験で乗除算アルゴリズムに関する問題を出すと、符号拡張で間違える人がかなりいます。基本的なことですので、きっちりと押さえておいてください。

- ロード命令（符号なし）
  - メモリから読み出した1バイトのデータをレジスタの最下位バイトにコピーします。
  - 上位3バイトについては全てのビットを0にします。
    - $0C_{(16)}$ (=12) → $0000000C_{(16)}$ (=12)
    - $FF_{(16)}$ (=255) → $000000FF_{(16)}$ (=255)
- ロード命令（符号付き）
  - メモリから読み出した1バイトのデータをレジスタの最下位バイトにコピーします。
  - 上位3バイトについては、全てのビットを、読み出した1バイトのデータの符号ビットの値で埋めます。
    - $0C_{(16)}$ (=12) → $0000000C_{(16)}$ (=12)
    - $FF_{(16)}$ (=−1) → $FFFFFFFF_{(16)}$ (=−1)

符号拡張

バイトオーダ（byte order）

- 例えば、4バイトのデータをメモリの100～103番地に格納するものとします。メモリにどのように格納するかで、2つの方法が考えられます。



- メモリ番地の小さい方から大きい方へと順に、データの上位バイトから書き込んでいくのがビッグエンディアン（Big Endian）、対して下位バイトから書き込んでいくのがリトルエンディアン（Little Endian）。
  - 名称の由来は「ガリバー旅行記」の中で、卵を尖った小さい方から割るか大きくて丸い方から割るかで、民族間で論争していることによりますが、しょーもないのでこれ以上は説明する気になりません。
- 結局、ビッグエンディアンとリトルエンディアンのどちらでもよいのですが、統一されていないことが実際上の厄介な問題をもたらします。
  - データだけであっても、コンピュータの種類が異なればバイナリデータをそのまま使用できない、ということになります。

- 因みに、
  - IBM系のコンピュータではビッグエンディアンを使用しています。
  - Intel系のコンピュータではリトルエンディアンを使用しています。
  - ネットワーク上でのデータ転送には、ビッグエンディアンを使用することで、取り決めがなされています。

(8) プログラム制御命令

- 原則として、マシン命令はプログラム上で並んでいる順に実行します。
  - したがって、次に実行する命令のアドレスA_nは、現在実行中の命令のアドレスA_cとその命令の命令語長L_cとから、以下のように求まります。

$$A_n = A_c + L_c$$

  - このA_n/A_cをプログラムカウンタ（Program Counter; PC）に記憶して、上記の式に基づいて更新（PC = PC + L_c）することにより、命令実行の流れ（制御の流れ）を管理・制御します。
- これに対して、次に実行する命令を、オペランドで指定したアドレスの命令に変更するのが、分岐（branch、jump）命令です。
  - 分岐命令の場合は、以下のようにPCを更新します。

PC = オペランドで指定したアドレス（実効アドレス）

無条件分岐 vs 条件分岐

- 分岐命令には、以下の2つのオプションがあります。
  - 無条件分岐（unconditional branch）命令： オペランドで指定したアドレスに分岐します。したがって、動作は以下の通り。

    PC = オペランドで指定したアドレス

  - 条件分岐（conditional branch）命令： 条件が成立した場合は、オペランドで指定したアドレスに分岐します。しかし、成立しなかった場合は、プログラム上で並んでいる次の命令に進みます。したがって、動作は以下の通り。

    if (条件が成立)

    then PC = オペランドで指定したアドレス

    else PC = PC + L_c

- プログラミング言語（例えばC言語）の機能との対応関係
  - **goto**文： 無条件分岐命令を用いて実現。
  - **if**文： 条件分岐命令を用いて実現。
  - ループ（**for**文、**while**文など）： ループ用のマシン命令はありません。
    - 例えば、左下のプログラムは、無理やりプログラミング言語で記述すると、右下のプログラムのように実行します。

```
for( i=0; i<n; i++ ) {

    ループ本体

}
```

```
   i=0;
loop:
   if( i<n ) {

       ループ本体

       i++;
       goto loop;
   }
```

条件分岐の機能

- 条件分岐を実現するには、以下の3つの機能が必要です。

  (a) 値の比較

  (b) 条件が成立するか否かの判定

  - つまり、条件値が真となるか偽となるか。これにとって飛ぶ（Taken）か飛ばない（Not-Taken）かを決めます。

  (c) 分岐先アドレスの計算

  - アドレッシングモードに従って、オペランドから分岐先アドレスを求めます。

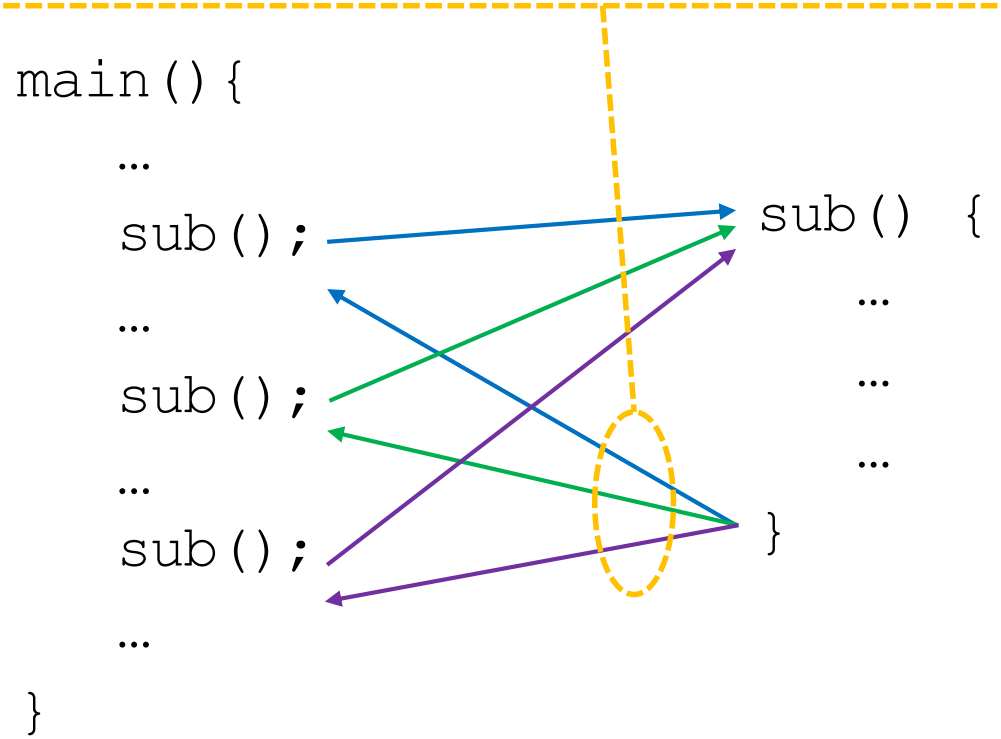
分岐するとき、英語では「The control is taken.」という言い方をするのに因みます。

- 命令セットによって、条件分岐命令の機能は異なります。
  - 1つの条件分岐命令が(a)～(c)の全ての機能を受け持つ命令セットもあります。
  - (a)は比較命令で、(b)(c)は条件分岐命令で受け持つ命令セットもあります。
    - 2命令に分割して条件分岐の機能を果たすので、この2命令間で比較結果の受け渡しを行わなければならなくなります。そのために設けられるのが、条件コードです。
  - (a)(b)を関係演算命令で、(c)を条件分岐命令で受け持つ命令セットも考えられますが、実在はしません。

サブルーチン分岐

- サブルーチン（sub routine）とは？
  - プログラム中で意味や内容がまとまっている作業をひとつの手続きとしてまとめたもの。
  - プログラム中の複数の箇所で同じ処理を行う部分をサブルーチンとすることにより、プログラムサイズを小さくしたり、デバッグ作業の効率化を図ることができる。
  - 呼び出す側の「主」となるもの（メインルーチン）に対して「サブルーチン」と呼ぶ。他に「手続き」「プロシージャ（procedure）」「関数」などと呼ぶ場合もある。
    - プログラミング言語では、結果の値を返すかどうかで「関数」と「手続き」を区別する場合がある。一方、マシン命令のレベルでは、結果の値を返すか否かは、サブルーチンを呼び出す分岐命令には無関係の話であるので、ここでは「関数」も「手続き」も区別しない。

- サブルーチンはプログラム中の複数の箇所から呼ばれることがある。
  - よって、サブルーチンから戻るときには、呼び出した次の命令に戻らなければならない。
  - つまり、サブルーチンからの戻り先は複数個存在し、そのどこに戻るかは、実行時に動的に決まる。



- コール（call, jump-and-link）命令
  - 通常の（条件）分岐命令に、リターンアドレスを保存するための機能を追加した命令。
- リターン（return）命令
  - コール命令で保存したリターンアドレスに分岐する（条件）分岐命令。
  - 専用のリターン命令を備えても良いが、例えば、アドレッシングモードのレジスタ（モード）で分岐先が指定できる分岐命令を備えていれば、リターンアドレスを格納したレジスタをオペランドで指定することによって“リターン”の機能が実現できる。

おわりに

- 以上、マシン命令の機能について見てきました。

- アセンブリ言語でプログラムを作成するプログラマや、コンパイラの設計者は、使用するプロセッサのマシン命令について知らなければなりません。この場合、命令セットはまさに、そのコンピュータハードウェアで何ができるか、を表すものと言えます。
- 一方、命令セットの設計者は、命令をハードウェアでどのように処理する（できる）かを考えながら、また、1つの命令ができるだけ多くの用途に利用可能なように考えて設計する必要があります。どのような命令セットを設計するかで、そのコンピュータの性能、ハードウェア量（コスト）や、さらにはその寿命（これから先どこまで利用され続けることができるか）までもが決まるのです。