はじめに

- 今回は、命令セットアーキテクチャについて、特に、「マシン命令とはどういうものか？」について講義します。

- 教科書の対応範囲は、以下の通りです。
  - 2.2.2項（p.38～43）

プログラム例

- 例として、右のようなC言語のプログラムを考えます。
  - 配列a[]の要素の和を求める簡単な関数です。
  - これをコンピュータハードウェアでどうやって実行するかを見ていきます。

```
int sum(int a[], int n)
{
    int i, s;

    s = 0;
    for( i=0; i<n; i++ )
        s += a[i];
    return s;
}
```

- これを「プログラム」などと言っていますが、コンピュータから見れば、これはただの文字列でしかありません。
- 実行するためには、マシン命令を用いてプログラムを表現しなければなりません。

- そこで、プログラミング言語で記述されたプログラムを、マシン命令のプログラムに変換（翻訳）するのが、コンパイラというソフトウェアです。

- まず、このプログラムが、IA-32（x86）の命令でどのように表現されるかを見てみましょう。
  - IA-32 は、Intel社が定義した32ビットプロセッサ用の命令セットです。
    - Intel社のCoreシリーズプロセッサの命令セットです。AMD社製のRyzenシリーズのプロセッサでも採用されています（命令セット互換）。
    - おそらく、皆さんが使っているPCのプロセッサはほとんどIntel社製またはAMD社製でしょう。

- 使用したコンパイラは、先のプログラムを右のように置き換えてから翻訳しました。
  - プログラムの意味は同じです。
  - 四角で囲んだ部分をマシン命令に翻訳したのが、次ページ。

```
int sum(int a[], int n)
{
    int i, s;

    s = 0;
    i = 0;
    if( n > 0 )
        for(; i<n; i++ )
            s = s + a[i];
    return s;
}
```

```
命令アドレス    命令語          アセンブリ表記
-----------------------------------------------
8048580:    03 04 93    add   (%ebx,%edx,4),%eax
8048583:    42          inc   %edx
8048584:    39 ca       cmp   %ecx,%edx
8048586:    7c f8       jl    8048580
```

- 上記のプログラムは、IA-32（x86）の命令セット用に、gccでコンパイルしたものです。

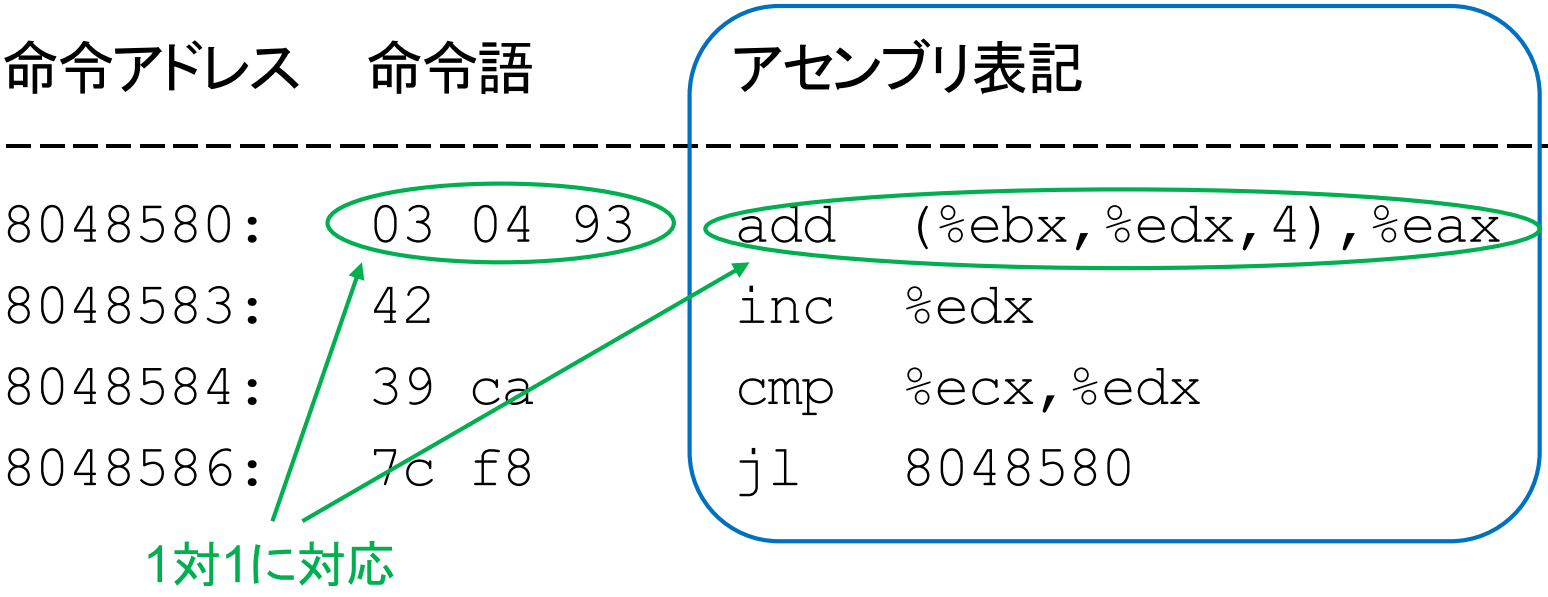
- では、このプログラムの読み方について、順に説明していきます。

```
命令アドレス    命令語          アセンブリ表記
-----------------------------------------------
8048580:    03 04 93        add   (%ebx,%edx,4),%eax
8048583:    42              inc   %edx
8048584:    39 ca           cmp   %ecx,%edx
8048586:    7c f8           jl    8048580
```

メモリ(主記憶)のアドレスを16進表記で表しています。

命令語を16進表記で表しています。

- 1行に1命令を記述します。
- 主記憶の8048580番地から、命令語030493を格納する、と解釈してください。
- 命令語が3バイトの長さなので、次の命令語は8048580+3=8048583番地から格納します。
- IA-32では、命令によって命令語の長さが異なることに注意してください。



- プロセッサは、メモリから命令語を読み出して実行します。
  - 1つの命令語を実行したら、基本的にはメモリ上で並んでいる順に、次の命令語を読み出して実行します。
- しかし、ハードウェアに合わせて命令語を16進表記(2進表記ならなおさら)していたのでは、人(プログラマ)にとって扱いにくくて不便です。そこで、少しはわかりやすくなるようにと考え出されたのが、アセンブリ表記(アセンブリ言語)です。
  - もちろん、アセンブリ言語でプログラムを書いたら(プログラミング言語がなかった時代はそうしていました)、それをマシン語のプログラムに変換しなければなりません。その変換ソフトがアセンブラです。

```
命令アドレス    命令語          アセンブリ表記
-----------------------------------------------
8048580:    03 04 93    add   (%ebx,%edx,4),%eax
8048583:    42          inc   %edx
8048584:    39 ca       cmp   %ecx,%edx
8048586:    7c f8       jl    8048580
```

$01000010_{(2)} = 42_{(16)}$

OPコード
（上位5ビット）

オペランド
（下位3ビット）

演算（操作）の対象データの格納場所（またはデータ値そのもの）を指定する。この例では、edxと名付けられたレジスタを指定している。

演算（操作）の種類を表す。操作コード（operation code）、オペコード、オプコードともいう。この例では、increment（1を加算する）を表す。アセンブリ表記では、”INC”のように略号を用いるが、これをニモニック（mnemonic）コードという。

- 命令語のどの部分（ビットフィールド; bit field）が何を表すかを定義したものを命令形式（instruction format）と言います。

- 命令語の主要なフィールド（命令語の構成要素）は以下の通りです。（図2.6、図2.7）
  - OPコード（operation code）:
    - 演算/操作の種類。基本的には2項演算か単項演算。
  - オペランド（operand）:
    - 演算/操作対象のデータの格納場所。演算に使用する値（定数）を指定することもある。
    - ソースオペランド（source operand）:
      - 演算に使用するデータ（つまり、演算数や被演算数）の格納場所を表すオペランド。
    - デスティネーションオペランド（destination operand）:
      - 演算結果の格納場所を表すオペランド。

- では、プログラム例について、1命令ずつその機能を見ていきます。
  - プロセッサの種類によって異なりますが、マシン命令とはどういうものか、だいたいのイメージをつかんでください。
    - このイメージがつかめなければ、後の話にはほとんどついていけません。

- その前に、データの格納場所について説明しておきます。以下の2種類があります。
  - メモリ（主記憶）：　アドレス（番地）で記憶場所を指定します。1つの番地に1バイト（8ビット）のデータを格納します。
    - 例えば、4バイトのデータを100番地に格納する場合（命令語では100番地と指定）は、実際には100～103番地に格納します。
  - レジスタ：　プロセッサ内にある少数の記憶装置。1個のレジスタのビット数は、ワード長に固定です。番号（名前）でどのレジスタを使用するかを指定します。
    - IA-32では、eax（0番）、ebx（3番）、ecx（1番）、edx（2番）、など、主に8個のレジスタを用います。
  - 最終的には、以上のいずれかをオペランドで指定します。

```
命令アドレス    命令語        アセンブリ表記
-----------------------------------------------
8048580:    03 04 93     add   (%ebx,%edx,4),%eax
8048583:    42           inc   %edx
8048584:    39 ca        cmp   %ecx,%edx
8048586:    7c f8        jl    8048580
```

- OPコードはadd（加算）。
- 第1ソースオペランドはレジスタeaxで、デスティネーションオペランドも兼ねる。
- 第2ソースオペランドはメモリ。
  - そのアドレスはebx+edx×4で求める。
  - レジスタedxに変数`i`の役割をさせ、レジスタebxに配列`a[]`の先頭アドレスを格納しておけば、`a[]`の要素はint型（つまり4バイト）なので、上記のアドレスは`a[i]`の格納場所を計算していることになる。（配列要素は順に並んでメモリ上に配置される。）
- レジスタeaxに変数`s`の役割をさせることにすれば、この命令は、ソースコードの `s = s + a[i]` の計算を行う。

```
命令アドレス    命令語          アセンブリ表記
-----------------------------------------------
8048580:    03 04 93    add   (%ebx,%edx,4),%eax
8048583:    42          inc   %edx
8048584:    39 ca       cmp   %ecx,%edx
8048586:    7c f8       jl    8048580
```

- OPコードはincrement（1だけ増加）。
- ソースオペランドはレジスタedxで、デスティネーションオペランドも兼ねる。
- レジスタedxの内容に1を加えて、その結果をedxに書き込むが、edxに変数iの役割をさせているので、この命令はソースコードの i++ の計算を行う。

```
命令アドレス    命令語          アセンブリ表記
-----------------------------------------------
8048580:    03 04 93    add   (%ebx,%edx,4),%eax
8048583:    42          inc   %edx
8048584:    39 ca       cmp   %ecx,%edx
8048586:    7c f8       jl    8048580
```

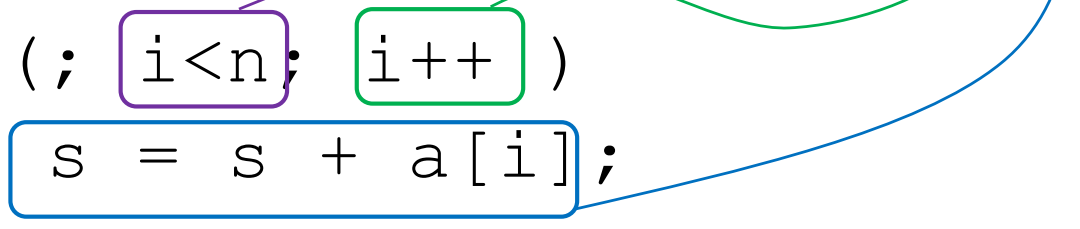
- OPコードはcompare（比較）。
- ソースオペランドはレジスタedxとecx。
- レジスタedxの内容からecxの内容を引き、その結果が正、負、0、のどれになったかを条件コード（condition code）と呼ぶビットに記憶する。
  - 普通の減算命令ではないので、引き算を行なった結果の値はedxに書き込まずに捨てる。
- レジスタedx、ecxがそれぞれ変数i、nに対応しているので、ループ終了条件の `i<n` の判定のための計算を行っている。
  - 単にiとnを比較しただけで、条件 `i<n` が真か偽かまでは判定していないことに注意！

命令アドレス	命令語	アセンブリ表記
8048580:	03 04 93	add (%ebx,%edx,4),%eax
8048583:	42	inc %edx
8048584:	39 ca	cmp %ecx,%edx
8048586:	**7c f8**	**jl 8048580**

- OPコードはbranch on less-than。条件分岐で、小さかった場合に分岐する。
- ソースオペランドは命令アドレスの8048580 。
- 前の比較命令の結果が負であった場合は、次に実行する命令は8048580番地の命令とする。
  - これを「8048580番地に分岐する」という。
- 前の比較命令の結果が負でなかった場合は、分岐しない。つまり、次に実行する命令は8048588番地の命令である。ソースコードのforループを抜けることに対応する。

- ソースコードとの対応をまとめると

命令アドレス	命令語	アセンブリ表記
8048580:	03 04 93	add (%ebx,%edx,4),%eax
8048583:	42	inc %edx
8048584:	39 ca	cmp %ecx,%edx
8048586:	7c f8	jl 8048580



- では、次に、同じC言語プログラムが、IBM POWER(PowerPC) の命令でどのように表現されるかを見てみましょう。

  - IBM社が同社のプロセッサ用に定義した32ビットプロセッサ用の命令セットです。
    - 現在は、IBM社の大規模サーバで使用されています。
    - かつては、Apple社のMacコンピュータや、Sony社のプレステ3や任天堂Wiiなどのゲーム機にPowerPCが使われていました。

- 使用したコンパイラ(gcc)は、先のプログラムを右のように置き換えてから翻訳しました。
  - もちろん、プログラムの意味は同じです。
  - 命令セットに合わせて、翻訳の戦略も変わります。

```
int sum(int a[], int n)
{
    int i, s;

    s = 0;
    i = 0;
    if( n <= 0 )
        return s;
    for(; n!=0; i++, n-- )
        s = s + *(a+i);
    return s;
}
```

a+iは、メモリ上でaからint型データのi個分離れた場所を表します。従って、その内容*(a+i)はa[i]と同じです。

- ループ部分をコンパイルした結果です。
  - もう細かくは説明しません。
  - しかし、一見しただけで、x86のコードとは全く異なっていますね。

命令アドレス	命令語	アセンブリ表記		意味
10000490:	5569103a	rlwinm	r9,r11,2,0,29	r9=r11<<2
10000494:	396b0001	addi	r11,r11,1	r11=r11+1
10000498:	7c09502e	lwzx	r0,r9,r10	r0=mem[r9+r10]
1000049c:	7c630214	add	r3,r3,r0	r3=r3+r0
100004a0:	4200fff0	bdnz+	10000490	if --ctr!=0, goto 10000490

i

a

s

n

```
for(; n!=0; i++ )
    s = s + *(a+i);
```

- 「命令セットの違いがプロセッサの種類の違いである」ということがイメージできましたか？
  - 命令セットが違うということは、そこで定義された命令が（機能的に似ていたとしても）違うということ。そして、命令が違うということは、アセンブリ言語も違うということ。
  - したがって、アセンブリ言語でプログラムを書くのが当たり前だった昔は、種類が異なるコンピュータでプログラムを実行するためには、プログラムを書き直さなければなりませんでした。
- 現代のように、（C言語などの）プログラミング言語を用いてプログラムを書く場合は、プロセッサの違いなど意識する必要はありませんが。
  - とは言うものの、特に優秀なプログラマは、速いプログラムを書くために、プログラミング言語を使いながらもプロセッサ（ハードウェア）やコンパイラの最適化技術を意識したプログラムの書き方をしています。これは余談ですが。。。

- ここまでは、IA-32（やPowerPC）を例に、マシン命令とはどういうものかを見てきましたが、これから先は、より一般的な見地で命令セットアーキテクチャを考えていくことにします。

オペランド数による分類

- 2項演算では、普通、2つのソースオペランドと1つのデスティネーションオペランドが必要です。
- しかし、オペランド数が多いと、命令語長も長くなりがちです。命令語長は短いに越したことはありません。
  - オペランドにメモリを指定する場合はそのアドレスをオペランドフォールドに書き込むことになりますが、一般にメモリは容量が大きいので、アドレスのビット数も長くなります。したがって、オペランドの数は命令語の長さに大きな影響を与えます。
    - メモリ容量が2^kバイトであれば、アドレスはkビット必要です。

- オペランドにレジスタを指定する場合は、レジスタの本数が多くはないので、オペランドフィールドのビット数も少なくて済みます。したがって、オペランドの数は、メモリオペランドの場合に比べると影響は小さくなります。
  - 一般に、レジスタの本数は32程度までなので、その番号を指定するのには5（$=\log_2 32$）ビットもあれば十分です。
  - しかし、それでもやはり、限られたビット数で命令語を表現するためには、無視できない問題です。

- そこで、マシン命令のオペランド数について考えていくことにします。
  - 現在は、オペランドでメモリかレジスタを指定しますが、昔のコンピュータにはレジスタはありませんでした。以降の説明では、基本的にメモリオペランドを前提としますが、レジスタが設けられるようになった経緯も含めて説明します。

3アドレス方式

- 3つのメモリオペランドを指定（図2.9）
  - 2つのソースオペランド*src1*、*src2*
  - 1つのデスティネーションオペランド*dest*

- アセンブリ表記例
  - *Opc　src1, src2, dest*
  - *Opc　dest, src1, src2*

- 演算：　*dest* ← *src1* ※ *src2*
  （※は2項演算子）

- 命令語長が長くなるのが欠点

2アドレス方式

- 2つのメモリオペランドを指定（図2.10）
  - 1つのソース兼デスティネーションオペランド*dest*
  - 1つのソースオペランド*src*

- アセンブリ表記例
  - *Opc　src, dest*
  - *Opc　dest, src*

- 演算：　*dest* ← *dest* ※ *src*
  （※ は2項演算子）

- 命令語長が短くできる。しかし、、、

- *a*=*b*+*c* の計算に対して
  - 3アドレス方式なら「*a*=*b*+*c*」の1命令
  - 2アドレス方式なら「*a*=*b*; *a*=*a*+*c*」の2命令

  命令語長は短くなっても命令数が増えてしまう（実行時間も増える？）。

- *a*=*b*+*c*+*d*+*e*+*f* の計算に対して
  - 3アドレス方式なら「*a*=*b*+*c*; *a*=*a*+*d*; *a*=*a*+*e*; *a*=*a*+*f*」の4命令
  - 2アドレス方式なら「*a*=*b*; *a*=*a*+*c*; *a*=*a*+*d*; *a*=*a*+*e*; *a*=*a*+*f*」の5命令

  命令数の差は1命令のみ。よって加算する項数が多くなれば、1命令増えたことのペナルティは全体から見ると小さくなり、また、命令語長短縮の効果が大きくなる。

1アドレス方式

- 1つのメモリオペランドを指定（図2.11）
  - 1つのソースオペランド（*src*）のみ
  - 1つのソース兼デスティネーションオペランドとして、アキュムレータ（accumulator）*acc*を使用する。
    - アキュムレータはプロセッサ内の演算器の近くに設ける。
    - 1個しか存在せず、演算では必ず使用するので、わざわざ命令語の中でオペランドとして指定する必要はない。
- アセンブリ表記例
  - *Opc　src*
- 演算：　*acc* ← *acc* ※ *src*　　（※は2項演算子）
- 2アドレス方式に対して、メモリへのアクセスがアキュムレータへのアクセスに置き換えられることにより、命令の処理時間が短くなる？

1・1/2アドレス方式

- 1つのメモリオペランドと1つのレジスタオペランドを指定（図2.12）
  - 1つのソースメモリオペランド（*src*）
  - 1つのソース兼デスティネーションレジスタオペランド*reg*
    - アキュムレータに味を占めて、その数を増やす。これをレジスタ（register）と呼ぶ。
    - 複数存在するので、そのどれを使用するのかをオペランドで指定しなければならなくなった。しかし、レジスタの本数は少ないので、オペランドフィールドのビット数も少なくて済む。
      - 1/2というのは、本当に長さが半分になったという意味ではなく、0より大きいが1より小さい、という程度の意味。

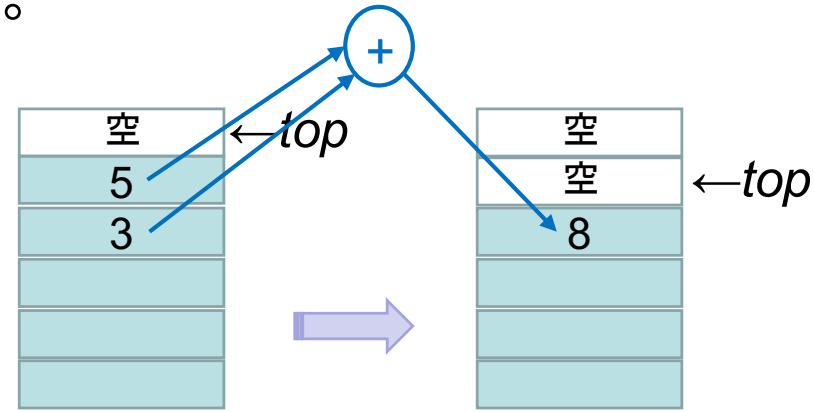
- アセンブリ表記例
  - *Opc　src, reg*
  - *Opc　reg, src*

- 演算：　*reg* ← *reg* ※ *src*　　（※ は2項演算子）

0アドレス方式

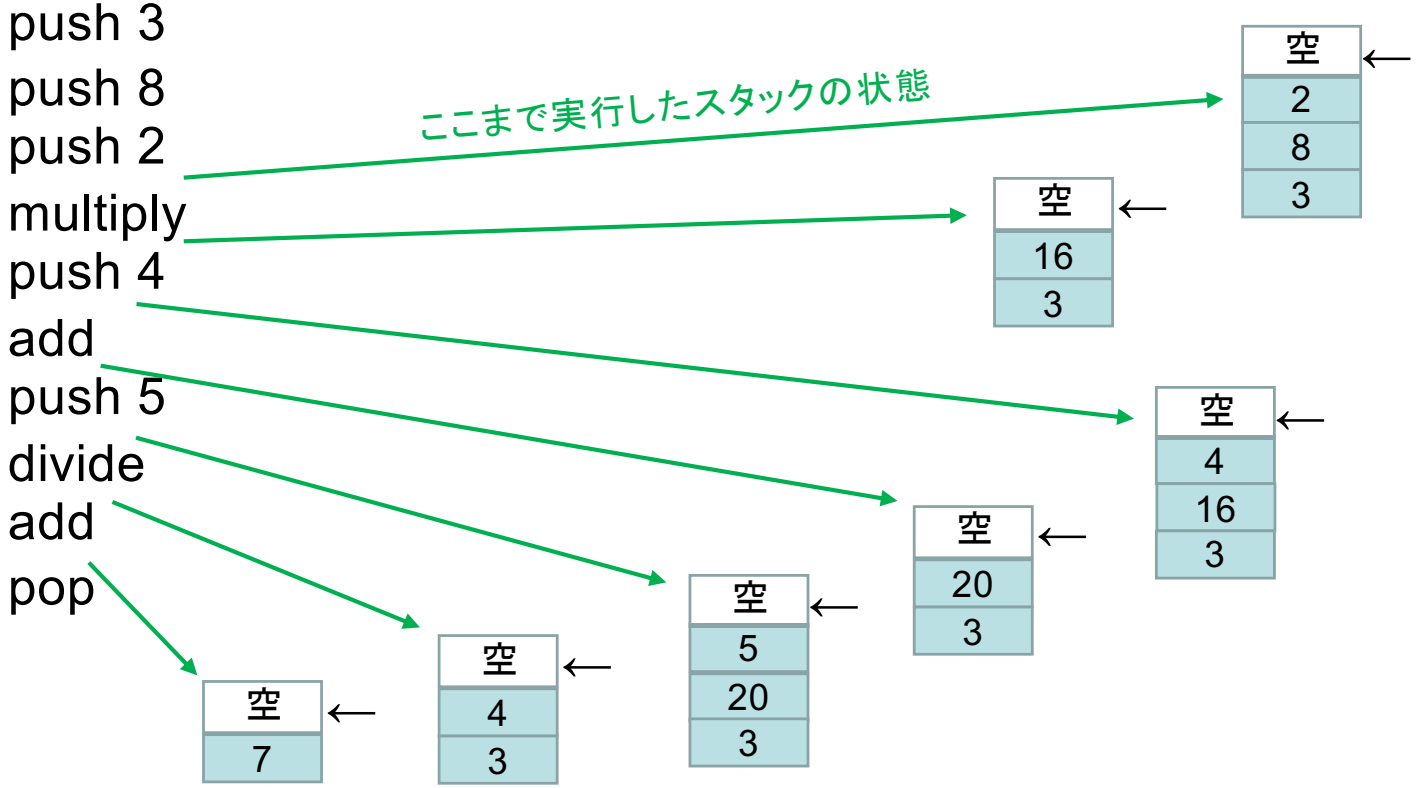
- 特殊なコンピュータで用いられる方式です。
- スタックを用いることで、オペランドを明示的に指定するのを避けることができます。

- 演算：　stack[*top*−2] ← stack[*top*−1] ※ stack[*top*−2]; *top*--
  （※ は2項演算子）
  - スタックを2回ポップして2個のデータを取り出し、それらを演算して演算結果をスタックにプッシュします。
- この他に以下の操作が必要です
  （プッシュ操作は1アドレス方式）
  - プッシュ操作：
    stack[*top*++] ← *operand*
  - ポップ操作：
    operand ← stack[--*top*]



- 例えば、式 3+(8×2+4) ÷5 を計算します。
  - この式を逆ポーランド記法に変換すると３８２ ×４＋５ ÷ ＋になります。
    - 演算子を演算数の後に置く記法です。この記法では、括弧を使う必要はありません。
  - この順番に命令にすると、以下のプログラムが得られます。

push 3
push 8
push 2
multiply
push 4
add
push 5
divide
add
pop



可変長命令と固定長命令

- 同じ演算（操作）を行う命令（つまり同じOPコードの命令）でも、オペランドの指定方法やオペランド数によって命令語の長さが異なる命令を可変長命令といいます。一方、OPコードに対して命令語長が一意に決まる命令を固定長命令と言います。
  - ただし、オペランドの指定方法や数を含めてOPコードにエンコードするものもあります。

- 命令セットの中の全ての命令語の長さが等しいときは「命令が固定長である」という言い方をします。そうでない場合（異なる長さの命令語が含まれている場合）は「命令が可変長である」という言い方をします。