# 2024 機械情報夏学期
# ソフトウェア第二

担当：岡田 慧 k-okada-soft2@jsk.t.u-tokyo.ac.jp,

9. 数値計算

## 1　計算機上でのデータの表現

### 1.1　情報量とビット

情報量はある事象が起きた際，それがどれほど起こりにくいかを表す尺度であり，よく起こる事は情報量が少ないが，滅多に起こらない事は情報量が多いと言う．

事象 $E$ が起こる確率を $P(E)$ とするとき，事象 $E$ が起きたときに受け取る情報量は

$$I(E) = log\frac{1}{P(E)} = -logP(E)$$

となる．log の底が 2 のときに単位は bit となる．2 つに一つの場合の情報量は $-log(1/2) = 1bit$ となり，万が一という場合の情報量は $-log(1/10000) = 13.288bit$ という．

一般に 8 bit を 1 byte というが，過去には 5 ビットから 12 ビットまであり，IBM System/360 で 8 ビットバイトを採用したことを契機に，現在は 1byte=8bit が定着した．正確に 1bit=8byte を明示したい場合はオクテット，オクテットバイトという．1 バイトで 0〜255 の数値を表現することができる．

### 1.2　負の数の表現

負の表現は 2 の補数（ある自然数に足したとき桁が一つ増える数のうちもっとも小さいもの）を使って表現する．2 の補数の計算法は 2 つある．一つは与えられた n ビットの値に対してそのビット数より一桁多く，最上位ビットが 1 残りが 0 の数値であり，この数から元の数を引けばよい．計算後は最上位ビットは無視する．もう一つは与えられたビットの値を全て反転させ 1 を加える（1 の補数に 1 を加える方法）である．

例えば，00100100 = 0x24 = 36 の補数は

```
   100000000
-)  00100100
-------------
    11011100
```

```
   11011011   (00100100 のビットを反転させた値)
+)        1
-------------
   11011100
```

として計算できる．補数表現のメリットは加算処理により（桁上がり・算術オーバフローを無視すれば）減算を表現できる点である．すなわち 36-36 の計算は 36+(-36) と解釈し下の様に加算器を用いて計算できる．

```
   00100100
+) 11011100
------------
   00000000
```

```
   11011100
    1011100  絶対値部分を考える
    1011011  1を引く
    0100100  ビットを反転 = 0x24 = 36
```

また，補数表現が与えられた時の解釈は次の様にすればよい．例えば 11011100 が与えられた場合，最上位ビットが 1 であることから負の値だと分かる．したがって，最上位ビットを抜いて，1 を引き 1 の補数とし，ビットを反転し元の数値する．

補数表現を用いた場合，1 バイトで表現できる数値は-128〜127 となる．C 言語では `unsigned char` としたときには 0〜255 となり，`char` としたときには-128〜127 となる．

2 の補数表現と加算に関するプログラムを以下に示す．[1]

```
/* test-minus.c */


#include <stdio.h>

int print_bit (unsigned char s) {
  int i;
  printf("0x%02x : ", s);
  for (i = 0; i < 8; i++) {
    printf("%c", ((0x80)&s)?'1':'0');
    s = s << 1;
  }
  printf("\n");
}
```

```
int main (int argc, char* argv[]) {
  int i;
  unsigned char u1, u2, u3;
  u1 = 36;
  printf("u1 = "); print_bit(u1);
  printf("u1 = %d\n", u1);
  // 2's complement
  u2 = (0xff ^ u1) + 0x01;
  printf("u2 = "); print_bit(u2);
  printf("u2 = %d\n", u2);
  printf("u2 = %d\n", (char)u2);
  u3 = u1 + u2;
  printf("u3 = ");  print_bit(u3);
}
```

## 1.3 小数点の表現

小数点は浮動小数点数として表現する．固定小数点と比較すると誤差は発生しやすいが大きな値や小さい値など表現できる値の幅が大きいため現在では一般的に採用されている．

実数の計算機表現では IEEE 754 形式と呼ばれる方式が一般的になっている．

例えば，単精度実数の表現は以下になる．

$$(-1)^{\text{符号部 }(N)} \times 2^{\text{指数部 }(E)-127} \times (1 + \text{仮数部 }(S))$$

```
31 30     23 22                     0
 N EEEEEEEE SSSSSSSSSSSSSSSSSSSSSSS
 0 10000000 01000000000000000000000
```

符号部 1 ビットを，指数部 8 ビット（符号付き整数），仮数部 23 ビット（符号なし整数）からなり，符号部は 1 であれば負の値を示し，指数部は 127 を追加した値が格納され，仮数部は整数部分が 1 であるような小数部分を表す．

2 の小数点表現は $2^1 2^0 . 2^{-1} 2^{-2} 2^{-3} 2^{-4}$ で表され，$10.1010_2$ は $1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2 + 0.5 + 0.125 = 2.625_{10}$ となる．

例えば 2.5 の浮動小数点表現を考えるには，整数部は $2 = 10_2$，小数部は $0.5 = .1_2$ から $10.1_2$ となり，仮数部の整数部分が 1 となるようにすると $1.01 \times 2^1$ となる．ここから符号部 (1 ビット)0, 仮数

[1] ^ はビット排他的論理和 (XOR) 演算子である．

部 (23 ビット)01000000000000000000000, 指数部 (8 ビット,127 バイアス)=1+127=128=10000000 となり，合わせて

```
0 10000000 01000000000000000000000
```

となる．

浮動小数点数の加算では両方の値の指数部が同じになるように小さい値の仮数部をシフトし，仮数部同士を加算する．最後に仮数部の整数部分が 1 となるように指数部，仮数部を調整する．例えば 2.5 + 15.5 はそれぞれの浮動小数点表現が

```
f = 2.500000    : 0 10000000  (1)01000000000000000000000
f = 15.500000   : 0 10000010  (1)11110000000000000000000
```

となる．指数部を揃えると

```
f = 2.500000    : 0 10000010  (0)01010000000000000000000
f = 15.500000   : 0 10000010  (1)11110000000000000000000
```

となり，仮数部を加算すると

```
f = 18.000000   : 0 10000010 1(0)01000000000000000000000
```

と計算できる．仮数部の整数部分が 1 になるように調整すると

```
f = 18.000000   : 0 10000011  (1)00100000000000000000000
```

なお，倍精度実数では符号部 1 ビットを，指数部 11 ビット（符号付き整数），仮数部 52 ビット（符号なし整数）となる．

浮動小数点数の精度は単精度実数で 10 進数 7 桁，倍精度実数で 10 進数 16 桁程度である．これは 1 の次の大きい最小の正の浮動小数点数を $1+\epsilon_M$ と表現したときの $\epsilon_M$ をマシンイプシロンと呼び，基数 $m$，仮数部が $p$ 桁の場合に $\epsilon_M = m^{-(p-1)}$ として計算する．つまり，単精度の場合は $2^{-23} \simeq 1.192092... \times 10^{-7}$ となり，倍精度の場合は $2^{-52} \simeq 2.220446... \times 10^{-16}$ となる．

```
int print_bit (float f) {
  int i;
  unsigned int *j = ((unsigned int *)&f);
  unsigned int s = *j;
  for (i = 0; i < 32; i++) {
    printf("%c", ((0x80000000)&s)?'1':'0');
    if ( (i == 0) || (i == 8) ) printf(" ");
    s = s << 1;
  }
  printf("\n");
}
```

```
// test-float.c

#include <stdio.h>

int main (int argc, char* argv[]) {
  float f; // 32bit, 4byte

  f = 2.5;
  printf("f = %-12f : ", f);
  print_bit(f);
}
```

## 1.4 バイトオーダ

多バイトのデータをメモリ上に配置する場合の方式. 例えば 2 進数で 00010010001101001100111011110 は 16 進数で 0x1234ABCD と表すことができるが，メモリ上にデータの上位バイトから 12 34 AB CD と並べる方式をビッグエンディアン (MSB:Most Significant Bit), データの下位バイトから CD

AB 34 12 と並べるリトルエンディアン (LSB:Least Significant Bit) と呼ぶ．IBM, Sun, Motorola は MSB を採用し，Intel は LSB を採用している．また，最近の組み込み系の CPU(ARM, PowerPC, SH4) 等はエンディアンを切り替えられるバイエンディアンという仕組みをもつものもある．

```
int print_byte (unsigned int s) {
  int i;
  unsigned char *p;
  p = (unsigned char*)&s;
  for (i=0;i<sizeof(unsigned int);i++) {
    printf("%02x", *p);
    p++;
  }
  printf("\n");
}
```

```
// test-byteorder.c

#include <stdio.h>

int main (int argc, char* argv[]) {
  unsigned int u1; // 32bit, 4byte

  u1 = 0x1234abcd;
  printf("u1 = %x : ", u1);
  print_byte(u1);
}
```

# 2 浮動小数点数の演算における数値計算誤差

## 2.1 数値計算誤差

1. オーバーフロー／アンダーフロー (overflow, underflow)
   演算結果が指数部で表現できる最大値や最小値を越えると生じる。

2. 桁落ち (cancellation of significant digit)
   ほぼ等しい数同士で引き算をする際に、有効数字が減少する誤差．上位の桁がゼロになると正規化によってそれを詰め以下の桁に 0 を強制的に挿入する．その結果有効数字が減少することで生じる．

3. 情報落ち (loss of trailing digits)
   非常に大きな値と非常に小さな値の加算や減算を行う際に，小さい数の値が反映されない誤差．指数は大きな値の指数に揃えられるため，小さな値の仮数部は大きくシフトされ，その結果仮数部の表現範囲を越え情報が欠落する．

4. 丸め誤差 (roundoff error)
   浮動小数点は 2 進数の有限小数で表されるが，2 進数で表せない無限小数は途中から値を切り捨てるため，正確には表現しきれないために生じる誤差．

## 2.2 浮動小数点数表現

コンピュータ内部で浮動小数点数を扱う際には，符号部，指数部，仮数部に分けて保管する．この時に，仮数部の最上位桁が 0 にならないように指数部を調整し正規化される．例えば通常の小数点数表記で 33.3333 に対応する正規化された浮動小数点数表記は $0.333333 \times 10^2$ になる．

桁落ちの例として，$33.3333 - 33.3332$ を計算してみる．この時有効数字は 6 桁となっている．

$$
\begin{aligned}
& 33.3333 - 33.3332 \\
= \; & 0.333333 \times 10^2 - 0.333332 \times 10^2 \\
= \; & 0.000001 \times 10^2
\end{aligned}
$$

最後に正規化を行うと

$$0.000001 \times 10^2 \rightarrow 0.100000 \times 10^{-3}$$

とする．このとき仮数部の小数第２桁以降は仮数部の最上位桁が０にならないように調整する為に追加された値であり有効桁は１桁となっている．

情報落ちの例として，$33.3333 - 0.0000333333$ を考えてみる．これの浮動小数点表記は

$$\begin{aligned} & 33.3333 - 0.0000333333 \\ = \; & 0.333333 \times 10^2 - 0.333333 \times 10^{-4} \\ = \; & 0.333333333333 \times 10^2 \end{aligned}$$

最後に正規化を行うと

$$0.333333333333 \times 10^2 \rightarrow 0.333333 \times 10^2$$

となり，0.0000333333 の値が無視されてしまう．実際には，浮動小数点数の加算減算は 1) 指数部分を同じにするように仮数部分を調整する，2) 仮数部分の加算減算を行う，3) 結果を正規化する，という手続きになるため，1) の段階で情報が落ちている．

## 2.3 交換則，結合則

浮動小数点数演算においては交換則 ($commutative : a+b=b+a, a \times b = b \times a$) は成立するが，結合則 ($associative : (a+b)+c = a+(b+c), a \times (b \times c) = (a \times b) \times c$) は成立しない．

$a = 111113, b = -111111, c = 7.51111$ を例に考えてみる．それの浮動小数点数表記は

$$a = 0.111113 \times 10^6, b = -0.111111 \times 10^6, c = 0.751111 \times 10^1$$

となる．有効桁 6 桁で $(a+b)+c$ と $a+(b+c)$ を計算すると以下の様になる．

$$\begin{aligned} (a+b)+c &= (0.111113 \times 10^6 + (-0.111111) \times 10^6) + 0.751111 \times 10^1 \\ &= 0.000002 \times 10^6 + 0.751111 \times 10^1 \\ &= 0.200000 \times 10^1 + 0.751111 \times 10^1 \\ &= 0.951111 \times 10^1 \\ &= 9.51111 \\ a+(b+c) &= 0.111113 \times 10^6 + ((-0.111111) \times 10^6 + 0.751111 \times 10^1) \\ &= 0.111113 \times 10^6 + ((-0.111111) \times 10^6 + 0.000008 \times 10^6) \\ &= 0.111113 \times 10^6 + (-0.111103) \times 10^6 \\ &= 0.000010 \times 10^6 \\ &= 0.100000 \times 10^2 \\ &= 10.0000 \end{aligned}$$

また，$u = 20000, v = -6, w = 6.00003$ を例に乗算の交換則を考えてみる．それぞれの浮動小数点数表記は

$$u = 0.200000 \times 10^5, v = -0.600000 \times 10^1, w = 0.600003 \times 10^1$$

となる.

$$\begin{aligned}
u * (v + w) &= 0.200000 \times 10^5 * (-0.600000 \times 10^1 + 0.600003 \times 10^1) \\
&= 0.200000 \times 10^5 * (0.000003 \times 10^1) \\
&= 0.0000006 \times 10^6 \\
&= 0.600000 \times 10^0 \\
&= 0.600000 \\
(u * v) + (u * w) &= (0.200000 \times 10^5 * (-0.600000) \times 10^1) + (0.200000 \times 10^5 * 0.600003 \times 10^1) \\
&= -0.120000 \times 10^6 + 0.1200006 \times 10^6 \\
&= -0.120000 \times 10^6 + 0.120001 \times 10^6 \\
&= 0.000001 \times 10^6 \\
&= 0.100000 \times 10^1 \\
&= 1
\end{aligned}$$

## 2.4 Python プログラムによる確認

```
$ python
Python 2.7.15rc1 (default, Nov 12 2018, 14:31:15)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from decimal import *
>>> getcontext().prec = 6

>>> a, b, c = Decimal(111113), Decimal(-111111), Decimal('7.51111')
>>> (a + b) + c
Decimal('9.51111')
>>> a + ( b + c )
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.00003')
>>> u * ( v + w )
Decimal('0.60000')
>>> u * v + u * w
Decimal('1')
```

## 2.5 桁落ちの例

二次方程式 $ax^2 + bx + c = 0$ の解の公式は

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

であった．

ここで，$a = 1.0, b = -100000.0, c = 1.0$ という係数でこの解の公式を適用することを考えると $|b| >> |ac|$ から $|b| \approx \sqrt{b^2 - 4ac}$ となる．したがって，ここでほぼ等しい値同士で引き算する際に有効数字が減少する桁落ちが発生し，誤差が極端に悪くなる[2]．

そこで，$x_1, x_2$ のいずれかが桁落ちが生じない点に着目し，桁落ちを生じないようなアルゴリズムを考える．ここでは

$$x_1 x_2 = \frac{c}{a}$$

の関係を利用して以下の様に答えを求める．

$$x_1' = \frac{-b - sign(b)\sqrt{b^2 - 4ac}}{2a}$$

$$x_2' = \frac{c}{a x_1'}$$

上記のプログラムの例が以下になる．

```
#include <math.h>
#include <stdio.h>

#define sign(x) (x/(fabs (x)))

int main () {
  double a = 1.0, b = -100000.0, c = 1.0;

  double x1 = (- b - sqrt(b*b - 4*a*c))/(2*a);
  double x2 = (- b + sqrt(b*b - 4*a*c))/(2*a);

  printf("a  = %27.20e\n", a);
  printf("b  = %27.20e\n", b);
  printf("c  = %27.20e\n", c);
  printf("x1 = %27.20e\n", x1);
  printf("x2 = %27.20e\n", x2);
  printf("e1   %27.20e\n", a * x1 * x1 + b * x1 + c);
  printf("e2   %27.20e\n", a * x2 * x2 + b * x2 + c);


  double x1_ = (- b - sign(b)*sqrt(b*b - 4*a*c))/2*a;
  double x2_ = c / (a * x1_);
  printf("x1'= %27.20e\n", x1_);
  printf("x2'= %27.20e\n", x2_);

  printf("e1'  %27.20e\n", a * x1_ * x1_ + b * x1_ + c);
  printf("e2'  %27.20e\n", a * x2_ * x2_ + b * x2_ + c);
}
```

これを実行すると

```
a  =  1.00000000000000000000e+00
b  = -1.00000000000000000000e+05
c  =  1.00000000000000000000e+00
```

[2] 10 進数から 2 進数の丸め誤差，さらに $\sqrt{b^2 - 4ac}$ の計算の丸め誤差も生じている．

```
x1 =  1.00000033853575587273e-05
x2 =  9.99999999899999966146e+04
e1   -3.38435755864452403330e-07
e2    0.00000000000000000000e+00
x1'=  9.99999999899999966146e+04
x2'=  1.00000000010000011393e-05
e1'   0.00000000000000000000e+00
e2'   0.00000000000000000000e+00
```

という結果が得られる．b を-1000, -1000000, -1000000000 と変えて結果を見てみるとよい．

## 2.6 浮動小数点の問題例

1. 以下のプログラムにおいて while ループが終了する理由を説明せよ．

2. また，same, diff のどちらが表示されるか．その理由はなぜか．

```
#include <stdio.h>

void sub(float f)
{
  union {
    float f;
    unsigned int i;
  } x;
  unsigned char *p; unsigned int *l;
  int i;            unsigned int g;
  p = (unsigned char *)&f;
  l = (unsigned int *)&f;
  x.f = f;
  g=x.i;
  printf(" %-12g : % ", f, *l);
  for ( i = 0; i < sizeof(float); i++) {
    printf("%02x", *p++);
  }
  printf(" : ");
  for ( i = 0; i < 32; i++ ) {
    printf("%c", ((0x80000000)&g)?'1':'0');
    if ( (i == 0) || (i == 8) ) printf(" ");
    g = g<<1;
  }
  printf("\n");
}
```

```
int main () {
  float e = 1, w, a=0.6, b=a+a+a;
  int i = 0;
  do {
    i++; e /= 2; w = 1+e;
    printf("%2d:", i); sub(e);
    printf("  w"); sub(w);
  } while ( w > 1 );

  if ( b == 1.8f )
    printf("same\n");
  else
    printf("diff\n");

  sub (a);
  sub (0.6f);
  sub (b);
  sub (1.8f);

  return 0;
}
```

1). e の値を 2 で割りつづけることで e が 1 と比べて非常に小さくなる。その結果 1 と e の和をとった際に情報落ち誤差が生じ $w(=1+e)>1$ が成り立たなくなり，while ループは終了する．

プログラムの実行中の様子を以下に示す．$w=1+e$ が 1 になっていることが分かる．

```
22: e 2.38419e-07  : 34800000 00008034 : 0 01101001 00000000000000000000000
```

```
    w 1            : 3f800002 0200803f : 0 01111111 00000000000000000000010
23: e 1.19209e-07  : 34000000 00000034 : 0 01101000 00000000000000000000000
    w 1            : 3f800001 0100803f : 0 01111111 00000000000000000000001
24: e 5.96046e-08  : 33800000 00008033 : 0 01100111 00000000000000000000000
    w 1            : 3f800000 0000803f : 0 01111111 00000000000000000000000
diff
```

2. diff が表示されるコンピュータで小数を表す際には 2 進数の有限小数を用いて表現し，2 進数の有限小数で表せない場合は最も近い有限小数で近似するが，その際に丸め誤差が生じる．0.6 は 2 進数では無限小数となるため丸め誤差が生じている．結果として b は正確には 1.8 にならないため diff が表示される．

気になる者は

```
printf("%.28f\n", a);
printf("%.28f\n", 0.6f);
printf("%.28f\n", b);
printf("%.28f\n", 1.8f);
```

のようにして確認すると良い．

## 2.7 多倍長精度浮動小数点数計算

ここまで扱ってきた浮動小数点数は仮数部の桁数が固定されているものであった．より多い仮数部の桁数を指定できる浮動小数点数を多倍長精度浮動小数点数と呼ぶ．浮動小数点数は仮数部の桁数は CPU のアーキテクチャにより決まっているため，多倍長精度浮動小数点数を使う場合はソフトウェアの命令を組み合わせて実現するため，計算時間はより多くかかる．

多倍長精度浮動小数点数計算を行う環境は未だ発展途上であり外部のライブラリを利用する必要がある．ここでが GNU MP(gmp:The GNU Multiple Precision Arithmetic Library) を利用する方法を紹介する[3]が，この他にも CRlibm(Correctly Rounded mathematical library), RealLib などが存在する．

このように，数値計算の途中で発生する誤差を減らすための解決策としては

1. アルゴリズムを改良する
2. マシンイプシロンを小さくする

の 2 つがある．後者は前者に比べて簡単に実現でき，また最近の数式処理ソフトやプログラミング言語では簡単に利用できる環境がサポートされつつある．

```
2  //  gcc test-gmp.c -lgmp; ./a.out
3  #include <stdio.h>
4  #include <math.h>
5  #include <gmp.h>
6
7  #define sign(x) (x/(fabs (x)))
8
9  int main () {
```

[3] sudo apt install libgmp-dev としてインストールできる

```
10    unsigned long int prec;
11    mpf_t a, b, c;
12    prec = 64;
13    mpf_init2(a, prec);
14    mpf_init2(b, prec);
15    mpf_init2(c, prec);
16
17    mpf_set_d(a,1);
18    mpf_set_d(b,-100000);
19    mpf_set_d(c,1);
20
21    mpf_t bb, mb, ac, b2ac, a2, x1, x2;
22    mpf_init2(bb, prec);
23    mpf_init2(mb, prec);
24    mpf_init2(ac, prec);
25    mpf_init2(b2ac, prec);
26    mpf_init2(a2, prec);
27    mpf_init2(x1, prec);
28    mpf_init2(x2, prec);
29    mpf_mul(bb, b, b);
30    mpf_mul(ac, a, c);
31    mpf_mul_ui(ac, ac, 4);
32    mpf_mul_ui(a2,  a, 2);
33    mpf_sub(b2ac, bb, ac);
34    mpf_sqrt(b2ac, b2ac);
35    mpf_neg(mb, b);
36    mpf_sub(x1, mb, b2ac);
37    mpf_add(x2, mb, b2ac);
38    mpf_div(x1, x1, a2);
39    mpf_div(x2, x2, a2);
40
41    printf("a :");
42    mpf_out_str(stdout, 10, 10, a);
43    printf("\n");
44    printf("b :");
45    mpf_out_str(stdout, 10, 10, b);
46    printf("\n");
47    printf("c :");
48    mpf_out_str(stdout, 10, 10, c);
49    printf("\n");
50    printf("x1:");
51    mpf_out_str(stdout, 10, prec, x1);
52    printf("\n");
53    printf("x2:");
54    mpf_out_str(stdout, 10, prec, x2);
55    printf("\n");
56
57    mpf_t r, axx, bx;
58    mpf_init2(r, prec);
59    mpf_init2(axx, prec);
60    mpf_init2(bx, prec);
61    mpf_mul(axx, a, x1);
62    mpf_mul(axx, axx, x1);
63    mpf_mul(bx, b, x1);
64    mpf_add(r, axx, bx);
65    mpf_add(r, r, c);
66    printf("e1:");
67    mpf_out_str(stdout, 10, prec, r);
```

```
68    printf("\n");
69
70    mpf_mul(axx, a, x2);
71    mpf_mul(axx, axx, x2);
72    mpf_mul(bx, b, x2);
73    mpf_add(r, axx, bx);
74    mpf_add(r, r, c);
75    printf("e2:");
76    mpf_out_str(stdout, 10, prec, r);
77    printf("\n");
78  }
```

```
 2  #include <iostream>
 3  #include <gmpxx.h>
 4
 5  int main (int argc, char *argv[]) {
 6    mpf_set_default_prec(1024);
 7    mpf_class a, b, c;
 8
 9    a = 1;
10    b = -100000;
11    c = 1;
12
13    mpf_class x1, x2;
14    x1 = ( - b - sqrt ( b * b - 4 * a * c )) / ( 2 * a );
15    x2 = ( - b + sqrt ( b * b - 4 * a * c )) / ( 2 * a );
16
17    std::cerr << "a  = " << a << std::endl;
18    std::cerr << "b  = " << b << std::endl;
19    std::cerr << "c  = " << c << std::endl;
20    std::cerr << "x1 = " << x1 << std::endl;
21    std::cerr << "x2 = " << x2 << std::endl;
22
23    mpf_class e1, e2;
24    e1 = a * x1 * x1 + b * x1 + c;
25    e2 = a * x2 * x2 + b * x2 + c;
26    std::cerr << "e1 = " << e1 << std::endl;
27    std::cerr << "e2 = " << e2 << std::endl;
28  }
```

## 宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし，また講義中にでてきたキーワードについて知らなかったもの, 興味のあるものを調べ「学籍番号.txt」としててアップロードすること．テキストファイルはワードファイルなどだと確認出来ないことがあるため，emacs/vi 等のテキストエディタを使って書こう．プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい．
画像で提出する場合は，各自のマシンの Mac アドレスが分かるようにすること．例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい．
ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと．また授業中に質問した者はその旨を記すこと．質問は成績評価時の加点対象となる．

キーワード：丸め誤差／打ち切り誤差／情報落ち／桁落ち，bignum

- 問題 l :

  1. 1.2 節 test-minus.c を参考に学籍番号の下 2 桁-上 2 桁の計算を，下 2 桁+上 2 桁の補数として計算してみよ．余裕があれば下 4 桁-上 4 桁を補数を利用して計算が可能なようにビット幅を増やしてみよ．
  2. 1.3 節 test-float.c で 2.625 のビット表現を出力してみよ．この時，以下の計算では same/diff のどちらが表示されるか調べてみよ。また、その理由を述べよ。

     ```
     float a = 2.625;
     float b = a + a + a;
     if ( b == 7.875f )
       printf("same\n");
     else
       printf("diff\n");
     ```

     さらに、以下の計算が期待通りの結果にならないのは何故か？

     ```
     float g = 0;
     for(int i = 0; i < 1000000; i++) {
       g = g + f;
     }
     ```

  3. 1.4 節 test-byteorder.c で short, int, long の場合のバイトオーダを調べてみよ．

- 問題２：二次方程式の求解プログラムを C 言語，並びに Python で書いてみよ．有効桁の違いで解の精度が変わることを確認せよ．

- 任意問題:実際の場面では数値計算ライブラリを利用する．一番有名なのは Lapack や GSL(GNU Scientific Library) であろうが，最近では C/C++言語では eigen[4] が，python では scipy,

[4] http://eigen.tuxfamily.org/

numpy[5] 等がよく利用されることが多い．これらのライブラリをつかった数値計算を行ってみよ．例えば連立方程式の解，あるいは，数値積分の例を以下に示す．

- 任意問題：上記でおこなった計算を行うためのアルゴリズムを調べ，自ら実装し結果を比較してみよ．例えば連立方程式の解を求めるプログラムでピボット選択をする場合としない場合で答えがどう変わるか調べてみよ．
- 任意問題：その他の数値計算例も自ら実装してみるとよい．例えば固有値，固有ベクトルの計算等

[5] http://www.scipy.org,http://www.scipy.org/NumPy_for_Matlab_Users

```
# http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html
from numpy import *
import scipy.linalg
a = array([[1.0  , 1.0/2, 1.0/3],
           [1.0/2, 1.0/3, 1.0/4],
           [1.0/3, 1.0/4, 1.0/5]])
b = array([1.0,2.0,3.0])
# solve ax = b
print("a=",a)
print("b=",b)
x = linalg.solve(a,b)
print("x=",x)
print("check=",dot(a,x))

```

以下のプログラムは `sudo apt-get install libgsl-dev` としてライブラリをインストールした後 `g++ -o integrate integrate.cpp `pkg-config gsl --cflags --libs`` としてコンパイルすること.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void * params) {
  double f = sin(x);
  return f;
}

int main (void)
{
  gsl_integration_workspace * w
    = gsl_integration_workspace_alloc (1000);

  double result, error;

  gsl_function F;
  F.function = &f;

  //Function: int gsl_integration_qags (const gsl_function * f,
  //                                    double a, double b, double epsabs,
  //                                    double epsrel, size_t limit,
  //                                    gsl_integration_workspace * workspace,
  //                                    double * result, double * abserr)
  gsl_integration_qags (&F, 0, M_PI/2, 0, 1e-7, 1000,
    w, &result, &error);

  printf ("result          = % .18f\n", result);
  printf ("estimated error = % .18f\n", error);

  gsl_integration_workspace_free (w);

  return 0;
}

```

# A　コンピュータアーキテクチャ[6]

コンピュータハードウェアの基本設計をコンピュータアーキテクチャと呼び，その設計をする人はコンピュータアーキテクトと呼ばれる．コンピュータアーキテクチャの内容は CPU の命令セットやレジスタなどの基本的なものから，その実行方式，バス等のシステムのインターフェース，メモリコントローラや，マルチプロセッサ構成など多岐に渡る．また，ハードウェアはソフトウェアと異なり一度設計，製造したら更新することが難しいという制約があるが，それを十分考慮したうえ，ハードウェアの要素を選択，結合し，より機能的，高性能，高信頼ででかつ低コストのコンピュータを構成するための技術といえる．

また，応用 (application:usage and context) により設計の目標は大きく異なる．応用には天気予報，ゲノム解析などの科学技術では高速計算，大規模メモリ，小数点計算が重要になり，Cray T3E や IBM BlueGene 等の専用計算機が設計されてきている．また，データーベースや Web アプリなどの商用では，データ移動やメモリや I/O のバンド幅が重要になり，AMD Opteron, Intel Xeon などがあり，ゲームやオフィス利用などのデスクトップ応用ではメモリ幅，グラフィクス，ネットワーク，価格等が重要になり，Intel, Core i7, AMD Athlon などがある．また，最近ではゲーム機，電話等の携帯デバイス応用が増えている．ここでは省電力，小型，無線等が重要になり，Core2 Mobile, Atom, AMD など開発が盛んである[7]．

## A.1　コンピュータ第 1 世代から第 4 世代まで

コンピュータは使われる論理素子の違いにより第 1 世代から第 4 世代に分類されている．

第 1 世代は真空管を用いたもので ENIAC(Electronic Numerical Integrator and Computer, ペンシルバニア大学，1946 年，17467 本の真空管，70000 個の抵抗器，10000 個のキャパシタ，30t, 21 平方メートル，弾道軌道計算），Manchester Mark I(マンチェスター大学，1949 年，4200 本の真空管) EDSAC(Electronic Delay Storage Automatic Calculator, ケンブリッジ大学，1949 年, 3000 本の真空管，0 - 99 の整数の 2 乗の表，素数のリスト) EDVAC(Electronic Discrete Variable Automatic Computer, ペンシルバニア大学, 1951 年, 6000 本の真空管と 12000 個のダイオード, 7,850kg, 45 平方メートル，弾道軌道計算）などがある．

1958 年からはトランジスタ論理素子を用いた第 2 世代コンピュータが主流になり応用が広がる．IBM 7070 (’58), DEC PDP-1(’60) などが代表的な計算機である．60 年代に例えば IBM はビジネス，科学計算，実時間等の異なるマーケット別に 4 つの計算機シリーズを販売し，それぞれ別の命令セット，IO システム，記憶装置，アセンブラ，コンパイラ，ライブラリ，が存在し，ソフトウェア工学やコンピュータアーキテクチャといった概念は薄かった．

1964 年に発表した IBM System/360[8]から IC が利用される第 3 世代コンピュータとなる．またこのコンピュータでは始めて，コンピュータアーキテクチャと実装を区別し，コンピュータファミリという考えを確立した．360 では小型から大型までの 6 つモデルからなるコンピュータシリーズを開発し，同時に 40 機種の共通周辺機器を発表した．全てのモデルで同じ命令セットが動くようにし，必要に応じてソフトウェアを変更することなく，上位モデルにアップグレードする事が可能になった．また，過去の計算機の命令セットをエミューレーションするオプションを提供した．さ

[6] コンピュータの構成と設計　ハードウエアとソフトウエアのインタフェース：デイビッド・A. パターソン , ジョン・L. ヘネシーが代表的な教科書である．

[7] PC Watch に不定期連載されている後藤弘茂の Weekly 海外ニュースでは最新のコンピュータアーキテクチャ事情が紹介されている http://pc.watch.impress.co.jp/docs/column/kaigai/

[8] このプロジェクトにかかった費用は 50 億ドル (現在の価格で 280 億ドル 2.5 兆円) であり，同時期のアポロ計画は 250 億ドルである．

らに商用の計算機では始めてオペレーティングシステムを搭載した．65 年には DEC PDP-8 という計算機を発表している．

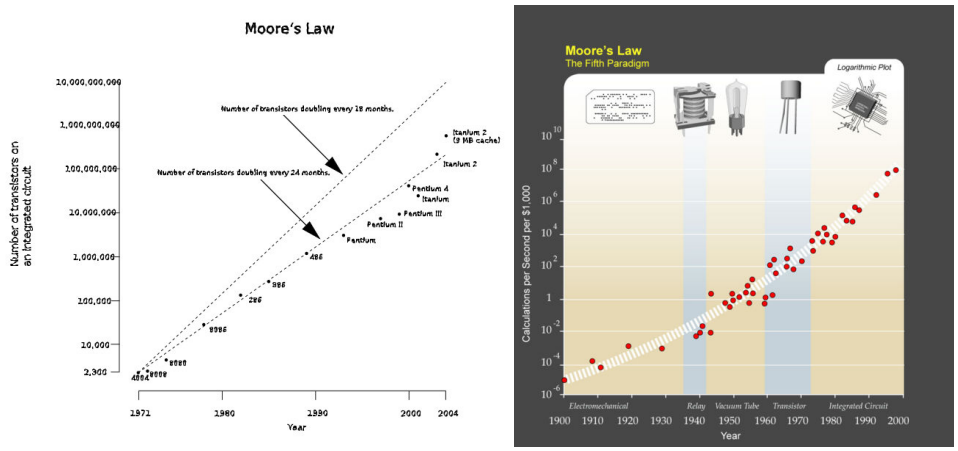
1970 年から大規模集積回路 (LSI) を用いた第 4 世代コンピュータが登場する．Intel の初めての CPU 4004 は 71 年，74 年には 8bit の Intel 8080 が，75 年には Zilog Z80 が販売される．また，1976 年には Apple I($666), NEC-TK-80(88,000 円）が登場し，現代のパーソナルコンピュータの基礎が築かれていく．さらに 76 年には世界初のスーパーコンピュータとして CRAY-I が発売される．

## A.2 コンピュータアーキテクチャを考える上でのいくつかの法則

### A.2.1 ムーアの法則

Intel の創業者のゴードン・ムーア (Gordon E. Moore, Ph.D, Intel 創業者の一人) が提唱したとされる経験則．トランジスタの集積密度は 18-24ヶ月で倍になる，というものである．

これまでに何回も Moore の法則の限界が指摘されてきているが，これまでは Moore の法則にしたがい成長してきている．また，これに基づきメモリサイズは 2 年，プロセッサスピードは 1.5 年，ディスク容量は 1 年で 2 倍になってきている．




Intel チップの速度の向上，ムーアの法則：1900 年代からの現代まで (Wikipeida より)

この 30 年間での CPU の進化は以下の 2 つのプロセッサを比べれば一目瞭然である．

**Intel 4004 (1971)**

- 応用：計算機
- 10uM PMOS
- 2300 トランジスタ
- $13mm^2$
- 108kHz
- 12Volts
- 4bit
- Single-cyclde datapath

**Intel Pentium 4 (2003)**

- 0.09uM CMOS (1/100X)
- 55,000,000 トランジスタ (20,000X)
- $101mm^2$ (10X)
- 3.4GHz (10,000X)
- 1.2Volts (1/10)
- 32/64 bit data (16X)
- 22-stage パイプライン
- 4 命令/サイクル（スーパースカラ）
- 2 レベルのキャッシュメモリ
- SIMD 命令，ハイパースレッド

### A.2.2 アムダールの法則

システムの一部を改良した時に全体として期待できる性能向上の程度知るための法則であり，ジーン・アムダール (Gene Amdahl, Ph.D. IBM の 704, 709, 7030, 360 の設計者，アムダール社の創業者）により提唱された．システム全体からみて P の割合の部分に対して S 倍の性能向上が合った場合，全体の性能向上は

$$\frac{1}{(1-P)+\frac{P}{S}}$$

で計算できる．

例えば，システム全体の 12%を 100 倍に高速化した場合の全体の性能向上は 1.13 倍にしかならない．

この法則はそもそも計算機の並列化に関する法則だが，以下の応用が可能である．例えば大きなプロジェクトがあり，並列で作業できる仕事が 5 割，10 人で作業する場合は $\frac{1}{(1-0.5)+\frac{0.5}{10}} = 1.82$ となる．もし，これに倍の人数を掛けて 20 人のスタッフを動員しても $\frac{1}{(1-0.5)+\frac{0.5}{10}} = 1.90$ の性能向上しか期待できないことが分かる．

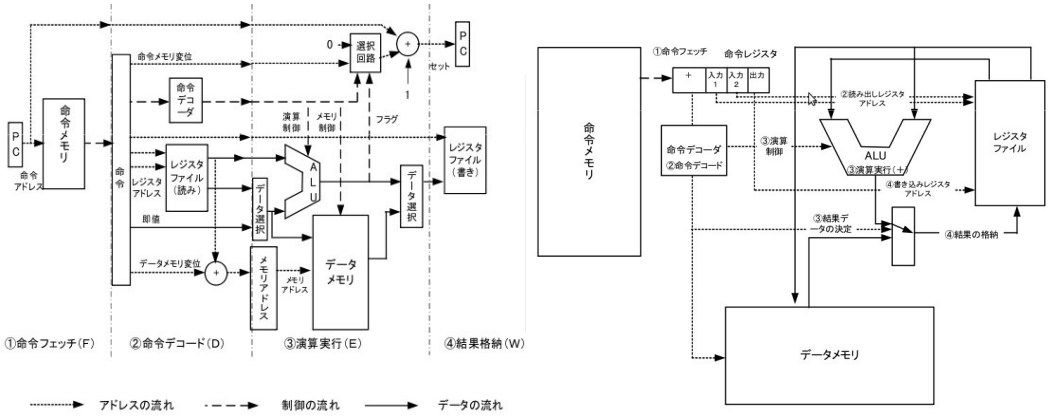
### A.2.3 スループットとレイテンシ

CPU の性能を考えるときに重要な概念がスループットとレイテンシがある．スループットとは単位時間あたりの処理能力であり，レイテンシとは結果を要求してからその結果を得るまでにかかる遅延時間のことである．

## A.3 CPU の構造と動作

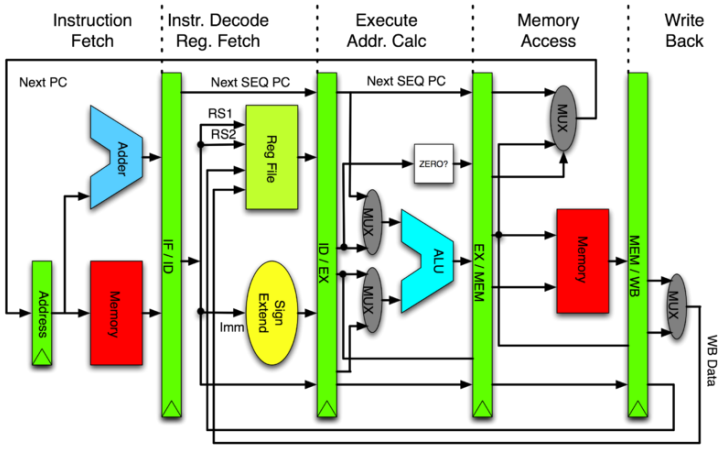
### A.3.1 CPU の構造

CPU は ALU(Arithmetic Logic Unit：論理演算装置) と呼ばれる演算装置，データを一時記憶するレジスタ，メモリなどの記憶装置へのインターフェース，CPU 全体を制御する制御装置で構成される．



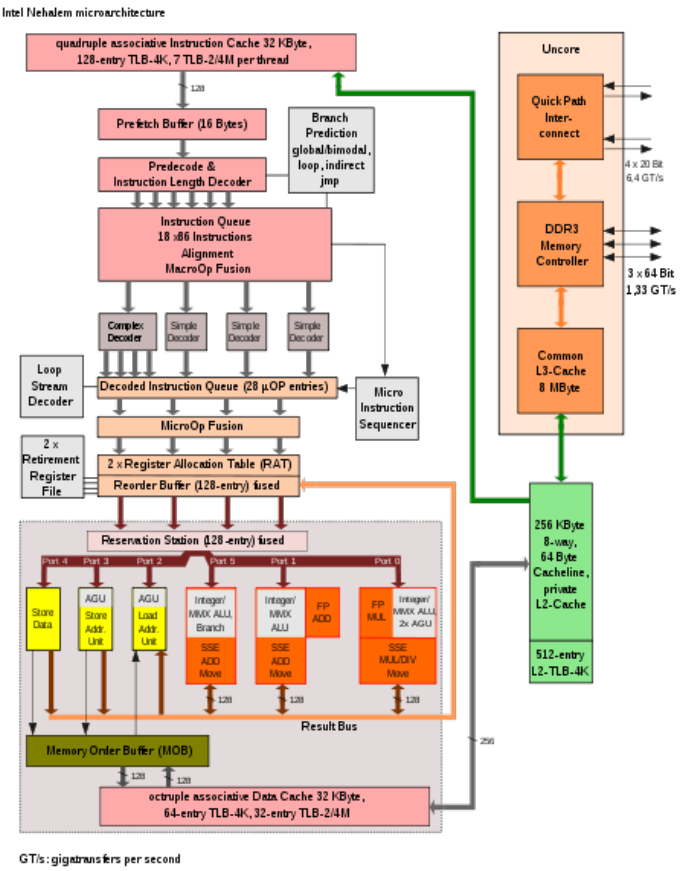

構造に着目したプロセッサ内の処理の流れ，動作に着目したプロセッサ内の処理の流れ[10]

命令メモリから命令をフェッチし，これを解釈（デコード）したあと，ALU を用いて計算し，結果をレジスタに格納している．また，命令によってはレジスタの情報からメモリに書き込んだり，メモリの情報をレジスタに書き込むなどの処理を行う．この概念図では命令メモリとデータメモリが分かれているが現代の実際の CPU アーキテクチャでは両方共メインメモリに格納されている．以下に実際のアーキテクチャの CPU の構造を示す．左が 1988 年にリリースされた MIPS R3000 アーキテクチャであり，右は 2008 年から利用されている Intel Nehalem アーキテクチャである．2-12 コアであり，実行ユニットとして (ALUx3, Loadx2, Storex2) を持っている．




R4000 プロセッサのアーキテクチャ[11]，

[10]図はコンピュータアーキテクチャ 坂井修一著より
[11]http://ja.wikipedia.org/wiki/R2000




Nehalem プロセッサのアーキテクチャ[12]

### A.3.2 CPU の動作

CPU の動作の基本はフェッチ，デコード，実行，ライトバックの 4 つのステージからなる．

フェッチとは実行すべき命令をメモリから取り出す作業である．メモリ上の実行すべき命令の位置はプログラムカウンタと呼ばれるレジスタで指定される．命令がフェッチされるとプログラムカウンタの値はフェッチしたデータ分だけ増加する．

デコードは，フェッチした命令をオペコード（命令コード）とオペランド（データ）に分割する．例えば加算命令ではオペランドに加算すべき数値や，数値があるメモリ上の場所やレジスタが指示されている．

最後に実行ステップが行われ命令で指定された操作を実行する．例えば加算命令では ALU を用いて計算を行う．また，ループ (while,for) や条件 (if,cond) で必要になるジャンプ命令ではプログラムカウンタを操作する．最後に結果をメモリやレジスタに書き込む．これをライトバックと呼ぶ．

あるいは，実行とライトバックをまとめてフェッチ，デコード，実行の 3 ステージとして扱うこともある．

命令を実行した後はフェッチのステージに戻り，次の命令をプログラムカウンタにしたがい読み出す．各ステージは CPU の 1 クロック毎に実行される．

[12] http://en.wikipedia.org/wiki/Intel_Nehalem_(microarchitecture)

### A.3.3 CISC と RISC

CPU のアーキテクチャには大きく CISC(Complex Instruction Set Computer) と RISC(Reduced Instruction Set Computer) がある．CISC は命令数が多く，命令形式も可変長で，個々の命令動作（アドレッシング）は複雑で，レジスタ数は少ないという特徴があり，Intel x86 や, Motorola M68000 などが素の代表である．一方で RISC は命令数が少なく，命令形式が固定長で，個々の命令動作は単純で，レジスタ数が多いという特徴があり，Sun Spark, MIPS, PowerPC, ARM, SuperH などが代表である．

初期の計算機から 80 年代まで CISC が一般的であった．これは命令やアドレッシングの種類を豊富に揃えておけばユーザの要求を満たすと考えていた．しかしながら実際の計算で使われているのはほとんど単純な命令で，複雑な命令を CPU がサポートしてもコンパイラが出力するのが難しいという問題があった．RISC では命令の種類を減らし，演算はレジスターレジスタ間に制限し，メモリの読み書きはレジスタとメモリの転送に限っている．これにより，パイプラインを効率よく回して，回路を単純化して，演算速度の向上を図るものである．Intel でも CISC 命令を RISC に解釈しなおして実行している．

## A.4 命令パイプライン

フェッチ，デコード，実行，ライトバックの各ステージを 1 クロックづつ行う場合，1 命令に 4 クロックかかることになる．そこで，命令パイプライン[14] という考えが生まれた[15]．

命令 A, B, C, D があった場合，命令パイプライン方式では，

1. 最初の 1 クロック目で命令 A がフェッチされる．
2. 2 クロック目で命令 B がフェッチされ，命令 A がフェッチされる．
3. 3 クロック目で命令 C がフェッチされ，命令 B がデコードされる．命令 A が実行される．
4. 4 クロック目で命令 D がフェッチされ，命令 C がデコードされる．命令 B が実行され，命令 A がライトバックされる．

と実行される．これは 4 ステージをもつパイプライン方式と実質的に 1 命令 1 クロックで実行できている．命令パイプラインでは 1 つの命令の完了までの時間は短縮しないが，命令のスループットを改善していることになる．上の例ではスループットは 4 倍となっている．

### A.4.1 スーパーパイプライン

スーパーパイプラインとはパイプラインのステージを細分化しスループットの向上を目指したものである．ステージを細分化することで各ステージの処理内容を軽減することで各ステージに必要な回路規模が減り，動作周波数を上げることが可能になる．例えばクロック速度 N(sec) で 4 ステージのパイプラインとクロック速度 N/2(sec) で 8 ステージのパイプラインを比較すると，一つ

[14] パイプライン処理とは，ある処理要素の出力を別の処理用素の入力になるよう直列に配置することで全体の処理の高速化を図るものである．2 つのプロセスのパイプライン処理を行うパイプもその一つである．

[15] 1961 年 IBM 7030 で最初に導入された．このコンピュータでは他に，命令プリフェッチ，メモリインタリーブ，マルチタスク，メモリ保護機能，汎用割り込み機構，8 ビットバイトといったコンセプトが導入されている．価格は 778 万ドル．1.2MIPS(100 万命令毎秒) の性能であった．ちなみに Pentium 4 は 9,726 MIPS, Cell 21,800MIPS, Core2 Quad Core 56,200MIPS である．

の命令の実行にかかる時間は両方共に 4N(sec) であるが，命令スループットは 4(sec) あたり 4 命令から，7 命令に向上していることが分かる．

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | F | D | E | W | | | |
| 2 | | F | D | E | W | | |
| 3 | | | F | D | E | W | |
| 4 | | | | F | D | E | W |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | F1 | F2 | D1 | D2 | E1 | E2 | W1 | W2 | | | | | | |
| 2 | | F1 | F2 | D1 | D2 | E1 | E2 | W1 | W2 | | | | | |
| 3 | | | F1 | F2 | D1 | D2 | E1 | E2 | W1 | W2 | | | | |
| 4 | | | | F1 | F2 | D1 | D2 | E1 | E2 | W1 | W2 | | | |
| 5 | | | | | F1 | F2 | D1 | D2 | E1 | E2 | W1 | W2 | | |
| 6 | | | | | | F1 | F2 | D1 | D2 | E1 | E2 | W1 | W2 | |
| 7 | | | | | | | F1 | F2 | D1 | D2 | E1 | E2 | W1 | W2 |

### A.4.2 スーパースカラ

スーパースカラとは命令パイプラインの各ステージにおいて，複数の命令を実行するものであり，プロセッサ内の冗長な実行ユニットを使って並列に処理する．命令レベルの並列性を CPU 内に実装したものである．結果として命令のスループットの向上が期待できる．下記の例では 4(sec) あたり 8 命令を実行している．

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | F | D | E | W | | | |
| 1 | F | D | E | W | | | |
| 2 | | F | D | E | W | | |
| 2 | | F | D | E | W | | |
| 3 | | | F | D | E | W | |
| 3 | | | F | D | E | W | |
| 4 | | | | F | D | E | W |
| 4 | | | | F | D | E | W |

## A.5 パイプラインハザード

命令パイプラインが有効になるためには，各命令間の依存関係があってはいけない．

分岐ハザード

分岐命令によってフェッチした命令が無駄になること．分岐命令を実行する場合，次に実行される命令は分岐が成立した場合と不成立の場合で 2 通りある．分岐命令が成立した場合（あるいは成立し無い場合），そうでなかった場合としてフェッチした命令はキャンセルし，再度分岐先の命令をフェッチし実行する必要がある．このような分岐ハザードにより命令のキャンセルが発生し，無駄になった部分をストールあるいはインターロックと呼ぶ．最近のパイプライン数が多いアーキテクチャではストールが無視できず重要な技術となっている．

データハザード

命令が直前の命令の実行結果を利用する場合のハザード例えば，

```
ADD    A B C  ;;  A <- B + C
ADD    D A C  ;;  D <- A + C
```

というコードがあった場合，1 行目の命令でレジスタ B とレジスタ C の値を加算しレジスタ A に書き込み，2 行目でレジスタ A と C の値を加算しレジスタ D に書き込む．この時，2 行目の命令は 1 行目がライトバックされるまで，実行を待つ必要がある．

構造ハザード

同一の時刻のおなじハードウェア資源を利用する競合に起因するハザードである．例えばおなじレジスタやメモリにアクセスするなどがある．

### A.5.1 分岐予測方式

分岐の有無を予測しパイプラインハザードを回避する方法．簡単な分岐予測は 1961 年の IBM 7030 から利用されている．予想が当たればフェッチが有効になりパイプラインはストールしないが，予想が外れればフェッチした命令を破棄して正しい分岐先の命令を再フェッチする必要がある．分岐予測方式には，アドレスが減る方向への分岐をループの最後にある分岐とみなして分岐する可能性が高いと判断する静的分岐予測や，1 ビットまたは 2 ビットのテーブルに過去の分岐結果を保存し予測する動的分岐予測などがある．

### A.5.2 アウトオブオーダ実行

資源の競合を割けるために，CPU 内でプログラムの実行順序を変更して競合が起きないようにすることで，スーパスカラ方式の並列度を挙げる技術である．

例えば以下のコードを 2 個の ALU を持ったスーパスカラ（同時に 2 つの命令が実行できる）計算機で実行する場合を考えると，3 と 4, 4 と 5，5 と 6 は資源の競合があり同時に実行できないことが分かる．

```
1: MOV    A 1    ;;  A <- 1
2: MOV    B 2    ;;  B <- 2
3: MOV    C 3    ;;  C <- 3
4: ADD    X B C  ;;  X <- B + C
5: ADD    Y A B  ;;  Y <- A + B
6: ADD    Z Y A  ;;  Z <- Y + A
```

そこでハードウェア内で命令の実行順序を

```
1: MOV    A 1    ;;  A <- 1
2: MOV    B 2    ;;  B <- 2
5: ADD    Y A B  ;;  Y <- A + B
3: MOV    C 3    ;;  C <- 3
4: ADD    X B C  ;;  X <- B + C
6: ADD    Z Y A  ;;  Z <- Y + A
```

の様に変えることで並列度を向上させようとするものである．また，実行中に利用するレジスタを選択し直しし高速化を実現するレジスタリネーミングという技術もある．

## A.6 SIMD によるベクトル計算

SIMD(Single Instruction Multiple Data) とは 1 回の命令で複数のデータに対する処理を行うもので，例えば 32bit 計算機において 128bit のレジスタを持ち，32bit のデータ 4 個分を一回の処理で行う．このような計算をベクトル計算とよび科学計算目的のスーパーコンピュータではよく採用された手法であったが，90 年代後半からパーソナルコンピュータや，ゲーム機などでも採用されている．SIMD はデータレベルの並列性の一種といえる．

Intel では 1997 年に MMX Pentium を発表したのが最初であり，現在は SSE(Streaming SIMD Extensions) と呼ばれる拡張命令セットが全てのプロセッサでサポートされており，SSE, SSE2, SSE3, SSE4 と発展してきている．

さらに 2011 年から採用されている SnadyBridge マイクロアーキテクチャでは MMX/SSD 後継の SIMD 命令セットとして Intel Advanced Vector Extensions(Intel AVE) がサポートされている．256 ビットのレジスタを用い，1 命令で 8 つの単精度浮動小数点数演算や 4 つの倍精度浮動小数点数演算の計算を実行出来る．

また，近年は 512 ビット長演算を対象とした Intel AVX-512 も実装され始めている

```
// test-mmx.c
// gcc -O0 -S -fverbose-asm -o test-mmx-O0.s test-mmx.c
// gcc -O3 -S -fverbose-asm -o test-mmx-O4.s test-mmx.c
#include <stdio.h>

int main (int argc, char *argv[]) {
  short a[4] = {1,2,3,4};
  short b[4] = {-4,-3,-2,-1};
  int   c[2];
  int   i;

  for(int i = 0; i < 2; i ++) {
    c[i] = a[i*2]*b[i*2] + a[i*2+1]*b[i*2+1];
  }
  for(i=0;i<2;i++) printf(" %3d",c[i]); printf("\n");
#if 0
  asm ( assembler template
      : output operands                  /* optional */
      : input operands                   /* optional */
      : list of clobbered registers      /* optional */
      );
#endif
  asm volatile ("movq %0,%%mm0": :"m"(a[0]));
  asm volatile ("movq %0,%%mm1": :"m"(b[0]));
  /* PMADDWD--Packed Multiply and Add */
  /* SRC : | X3  | X2  | X1  | X0  |  */
  /* DEST: | Y3  | Y2  | Y1  | Y0  |  */
  /* DEST: |X3*Y3+X2*X2|X1*Y1+X0*Y0|  */
  asm volatile ("pmaddwd %%mm1,%%mm0"::); /* mm0 の mm1 最大値 */
  asm volatile ("movq %%mm0,%0":"=m"(c[0]));
  asm volatile ("emms");

  for(i=0;i<4;i++) printf(" %3d",a[i]); printf("\n");
  for(i=0;i<4;i++) printf(" %3d",b[i]); printf("\n");
  for(i=0;i<2;i++) printf(" %3d",c[i]); printf("\n");
  return 0;
}
```

日本語の情報として http://sky.geocities.jp/krypton_pl/x86/x86ext.htm, に MMS/SSE 命令がまとまっている．また，http://www.wakayama-u.ac.jp/~chen/gccxmm.htm に MMX/SSE

プログラミングのよい資料がある[18].

なお，上記の様に命令，ソース，ディスティネーションという順番で並んでいる．これは AT&T 記法と呼ばれる．なお Intel は命令，ディスティネーション, ソースという順番を採用する．上記の例は gcc のアセンブラ (gas:gnu as) が AT&T 記法を採用している点に注意.

[18] [mmx site:intel.co.jp] などで検索すると一次情報が得られる．ftp://download.intel.co.jp/jp/developer/jpdoc/iaopt_j.pdf 等に目を通すと良い.