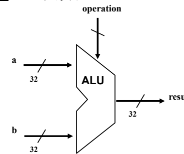
第5回
2012/5/22

「いよいよハードウェアの世界に:
数の表現と組合せ回路と演算器の設計」

機械語から演算へ:ハードウェアアーキテクチャ

- 今まで学んできたこと:
  - 機械の抽象化:
    - 様々な命令セットアーキテクチャ (ISA)
    - アセンブリ言語と機械語
  - 性能 (速度、サイクル、周波数、CPIなど)
- 次に行うこと:
  - これらのアーキテクチャをどのようにデジタル回路技術によるハードウェアとして実装するのか?



演算回路へ:計算機内の数字の表現

- ビット列は単にビット列である(特に内在する意味はない)
  — 「あるビット列をどのように数字として解釈するか」という規約がビット列と数字の間の関係を定める
- 二進数 (基数2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  十進数では: 0...2^n-1の正の整数
- 実際は、もっと複雑——プログラム言語でもプリミティブ型として定義:
  ビット列は有限(超えるとオーバーフローエラーがおきる)
  分数や実数の表現
  負の数の表現
  e.g., MIPS にはsubi 命令がない; addi は負の数を加算できる)
- 様々な数の表現法
  - 整数:どのように負の数を表現?
    i.e., どのようなビット列のパターンが、どの整数に対応?
  - 実数: どのような精度・指数? (後述)...
  - 演算回路の設計も表現に依存

負の整数の可能な表現法

符号と大きさ Signed Magnitude 符号ビット+abs(x) abs(x)	1の補数 One's Complement 負数はビット反転	2の補数 Two's Complement 負数 x は 2^n - abs(x)
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- 課題: バランス、ゼロの数、演算や操作の容易性
- どれがなぜ一番好ましいのか?

MIPS

- 32 bit の符号付き整数 - 2の補数表現:

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
0000 0000 0000 0000 0000 0000 0000 0010two = + 2ten
...
0111 1111 1111 1111 1111 1111 1111 1110two = + 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 2,147,483,647ten   maxint
1000 0000 0000 0000 0000 0000 0000 0000two = – 2,147,483,648ten   minint
1000 0000 0000 0000 0000 0000 0000 0001two = – 2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0010two = – 2,147,483,646ten
...
1111 1111 1111 1111 1111 1111 1111 1101two = – 3ten
1111 1111 1111 1111 1111 1111 1111 1110two = – 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = – 1ten
```


2の補数による演算操作

- 2の補数の値の符号の反転
  - ビットの反転と、符号の反転は異なることに注意
  - 整数値 xに対し、そのn ビットの2の補数表現を x', ビット反転をinv(x')とすると、
    - $x + (-x) = 0$、一方 $x' + inv(x') = 2^n-1$
  - (-x)' = inv(x') + 1 である。Q: これを証明せよ
- n ビットの数をnビット以上の数に変換するには:
  - MIPSの 16 bit の即値 (immediate)は32ビットに自動的に変換
  - 符号ビット(sign bit)である最上位ビット (MSB=most significant bit)を他のビットにコピー

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - "符号拡張" (lbu vs. lb)

加算と減算

- 小学生の算数 (1を桁あげ(carry)または桁下げ(borrow))

```
   0111          0111          0110
+ 0110         - 0110         - 0101
```

- 2の補数の演算は容易
  - 減算は2の補数の加算となる

```
  0111          0111
- 0110        + 1010
  0001          0001
```

- オーバーフロー (有限な値の範囲を演算結果が超えてしまうこと):
  - e.g., 二つのnビットの数を加算して、結果がn+1ビットになってしまうとき

```
   1111
+  0011
  10010
```

注: 2の補数でのオーバーフローは、単純な桁あげではない。

オーバーフローの検出

- 正の数と負の数の加算ではオーバーフローは起きない。
- 符号が同じ数同士の減算ではオーバーフローは起きない
- オーバーフローは、結果の値が符号に影響を与える時に起きる:
  - 二つの正の数を加えて、負の数を得るとき、
  - 逆に、二つの負の数を加えて、正の数を得るとき、
  - または、正の数から負の数を引いて、負の数を得るとき、
  - 逆に、負の数から正の数を引いて、正の数を得るとき

- ここで、A + BとA – Bを考えるとき
  - B = 0 のとき、オーバーフローは起きるか?
  - A = 0 のとき、オーバーフローは起きるか?
    - 0000 - 1000 (minint)
      => 0000 + inv(1000) + 1
      => 0000 + 0111 + 1 => 1000 (minint)

オーバーフローの影響

- 例外(ソフトウェアの割り込み)が起きる
  - ある特定の例外用のアドレスに強制的にジャンプ
  - 実行再開のために、割り込まれた時のアドレスが保存される(つまり、例外が起きた演算のアドレス)

- 処理の詳細は、ソフトウェアシステムや、プログラム言語に依存する
  - 例: 飛行機の制御の場合に対し、宿題の計算

- 必ずしも例外を起こしたいわけではない
  —MIPS 命令では、unsigned命令が例外を起こさない:
  `addu, addiu, subu`

  注: `addiu` は符号拡張を行う!
  注: `sltu, sltiu` が、符号なしの比較のために存在

復習: ブール代数と論理回路

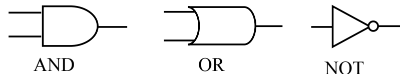
- 問題: A, B, and Cという 3つ入力がある論理関数(Logical Function)がある:

  出力 D は、一つの入力が真であれば真 (3入力 or)
  出力 E は、三つの入力が全てが真であれば真 (3入力 and)
  出力 F は、二つの入力が真であれば真 (3入力多数決回路)

- Q1: これらの論理関数を真理値表(Truth Table)を示せ.

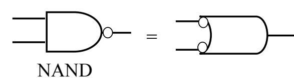
- Q2: これらの論理関数をブール代数(Boolean Algebra)上のブール式(Boolean Expression)で示せ

- Q3: これらの関数を実装する論理回路(Logical Circuits)を2入力のAND, OR, NOTゲートで構成せよ

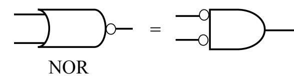


AND　　OR　　NOT

復習:いくつかのゲートの同値関係と表現法

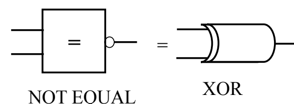


NAND



NOR

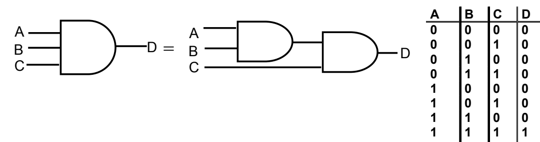
xor はexclusive or、つまり、両方の入力が一致すると 0、一致しないと 1



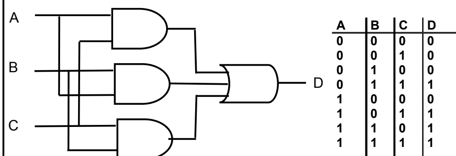
NOT EQUAL　　XOR

0	0	0
0	1	1
1	0	1
1	1	0

復習:三入力のゲート



A	B	C	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



A	B	C	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

多数決回路→積和標準形は？

復習(続き)

Q: 組み合わせ回路(Combinatorial Circuit)とは何か?

ブール論理 (Boolean Logic)との関係は?

Q: 積和標準形(conjunctive canonical form)とは何か?

Q: 組み合わせ回路における真理値表とは?

Q: すべての組合せ回路は積和標準形にて表現可能か?

Q: もし可能ならば、どのような手続きで構成するか?

(以上、わからない人はWebで調べよ)

積和標準形と真理値表

- 命題変数またはその否定，すなわちX，$\neg X$の形をリテラルという．
- リテラルから∧のみで作られる式を基本積という．基本積から∨のみで作られる式を**積和標準形**であるという．
  例：$(\neg X \wedge Y \wedge \neg Z) \vee (X \wedge Z) \vee (\neg Y \wedge \neg Z)$
- 任意の式は次の手続きで，同値な積和標準形に変形される．
  ① $\neg(A \vee B)$を$\neg A \wedge \neg B$で置きかえる (ド・モルガンの法則)
  ② $\neg(A \wedge B)$を$\neg A \vee \neg B$で置きかえる (ド・モルガンの法則)
  ③ ¬¬AをAで置きかえる
  ④ $A \wedge (B \vee C)$を$(A \wedge B) \vee (A \wedge C)$で置きかえる (分配則)
  ⑤ $(A \vee B) \wedge C$を$(A \wedge C) \vee (B \wedge C)$で置きかえる (分配則)

- **任意のn入力の論理回路は2^n通りの0-1の組み合わせを網羅した真理値表で表わされる。**

- **任意の論理回路に等価な積和標準形の論理式は、以下によって構成できる**
  - **入力を命題変数とし、出力が1となる行で入力が0となる変数Xをリテラル"¬X"**、1となる変数Xを"X"と表記して、それら全てのリテラルの基本積をとる
  - 上記で生成した全ての基本積の和をとった式を構築する

演算装置: ALU (Arithmetic Logic Unit)

- **`andi` と `ori` 命令を実装するALUを設計しよう**
  - **1 bit のALUを作り、32個並べればよい**

命令 (operation)

a

b

結果(result)

Op	a	b	res
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

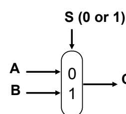
- **可能な実装 (積和標準形):**

$$\overline{Op}\cdot a\cdot b + Op\cdot\overline{a}\cdot b + Op\cdot a\cdot\overline{b} + Op\cdot a\cdot b$$

(実は多数決回路)

マルチプレクサ (Multiplexor)

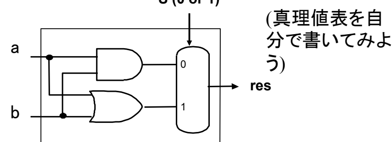
- **制御入力に基づいて、入力を一つ選択する**



注: これは、2入力のマルチプレクサと呼ばれるが実際はA, B, Sの3つ入力がある
多入力マルチプレクサは、2^n入力に対し、nビットの制御入力Sが存在する

- **MUXを用いてALUを作成してみよう:**

S (0 or 1)



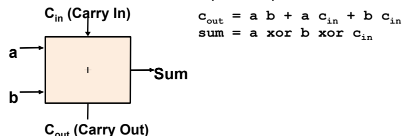
(真理値表を自分で書いてみよう)

ALUの異なる実装に関して

- **「最適」な実装を決定するのは難しい←以前は、ゲート数が少ないのが良い指針とされていた。**
  - **一つのゲートの(出力)ファンアウトの制限**
  - **あまり多入力のゲートは好ましくない (why?)**
  - **ゲートの段数は少ないほうが良い (why?)**
  - **我々にとっては、設計が理解しやすい方が良い**
- **加算用の 1-bit のALU を考えてみよう(全加算器):**

C_{in} (Carry In)

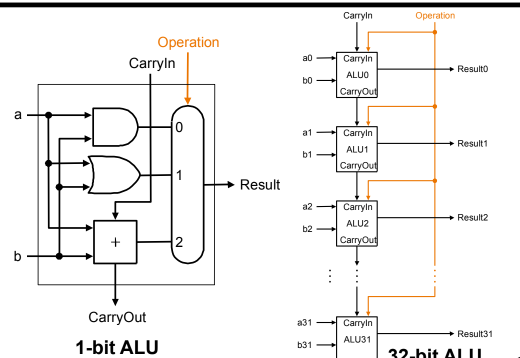
```
cout = a b + a cin + b cin
sum = a xor b xor cin
```



C_{out} (Carry Out)

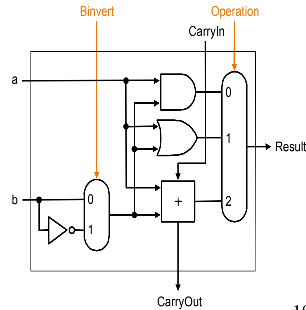
- **add, andと or の機能を有する1-bit ALU はどのように実装できる?**
- **それをどのように拡張すると32-bit ALUになる?**

単純な32 bit ALUの構成



減算 (a – b) の実装は?

- 2の補数を用いる: bを符号反転して、加算すれば良い
- Q: どのように符号反転すればば良い?
- 答: 1の補数をとり(ビット反転)、1を加算する
- 各1-bitの加算器で1の補数をとり、減算時Carry inを1にすれば良い。

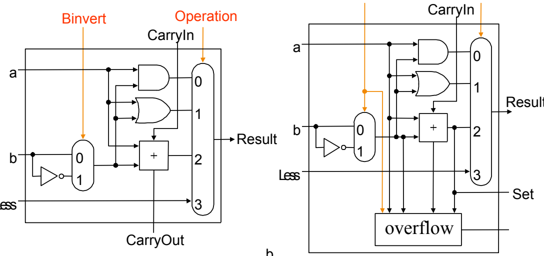


ALU の他の MIPS命令の実装

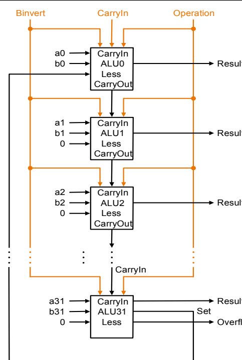
- set-on-less-than 命令 (slt)のサポート
  - 復習: slt は算術命令であった
  - もし rs < rt ならば1で、そうでなければ0
  - 減算を用いれば良い: (a-b) < 0 は a < bと等価
- 等号演算も必要 (beq $t5, $t6, $t7)
  - ここでも減算を用いれば良い: (a-b) = 0 は a = bと等価

Sltの実装

- 実際に回路が先ほどのアイデアを実装しているか?



Sltの実装(続き)



減算の結果がマイナス

MSB (31ビット目)が1

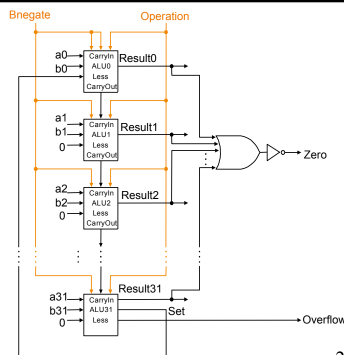
その結果をResult0に反映、他の桁のResultは0に

等号演算

- 制御信号に注意:

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

•*注: Zeroは結果が演算の結果が0のとき1となる*



結論

- MIPSの命令セットを実装するALUの一部を構築した
  - 鍵となるアイデア: マルチプレクサを用いて、必要な出力を選択
  - 2の補数を用いることによって、減算やその他の比較演算を容易に実装
  - 1-bit ALU を複製することによって、32-bitのALUを構築できる
- ハードウェアに関する重要な点
  - 全てのゲートが動作している
  - ゲートの速度はゲートの入力数に影響される←他入力ゲートは遅くなる
  - 回路の速度は直列に接続されたゲートの数に影響される
    (“クリティカルパス (critical path)”上のゲート数が速度を決定。1ゲートあたり数nsの遅延)
- 我々の主題は動作原理の理解にあるので、このような単純なアーキテクチャでも良かった。しかしながら、このままでは遅い
  - アーキテクチャの変更により、性能を大幅に向上可能である
    (ソフトウェアでより良いアルゴリズムを用いるのに似ている)
  - 加算と乗算に関して、このような改良された構成を見てみよう

問題: ripple carry 加算器は遅い

- 32-bit ALU は 1-bit ALUほど速いか?→キャリーのripple(波状の伝播)
- 加算、特にキャリーを伝播させるには、とのような実装があるだろうか?
  - 二つの極端な実装法: ripple carry と 積和標準系による算出

今までの加算器で、rippleはどのように伝播されていくか? どのように除去できるか→ 積和演算によって、あらかじめ算出しておけば良い?

$c_1 = b_0c_0+a_0c_0+a_0b_0$
$c_2 = b_1c_1+a_1c_1+a_1b_1 = b_1(b_0c_0+a_0c_0+ a_0b_0)+a_1(b_0c_0+a_0c_0+a_0b_0)+a_0b_0$
$= b_0b_1c_0+a_0b_1c_0+ a_0b_0b_1+a_1b_0c_0+a_0a_1c_0+a_0a_1b_0+a_0b_0$
$c_3 = b_2c_2 + a_2c_2 + a_2b_2 =$
$c_4 = b_3c_3 + a_3c_3 + a_3b_3 =$

これはとても現実的ではない(why)? => 積和標準系Aの積の項数を | A | とする
O(| c_n|)は?

Carry-lookahead 加算器

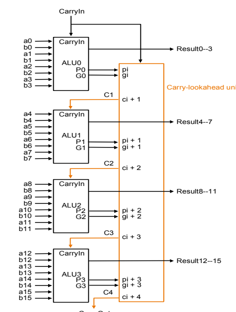
- 二つの極端なアプローチの中間的アプローチ
- 設計時の考察:
  - Ripple carryではcarry-inを伝播させていて、遅かった。一方積和では、上位桁のcarry-inの算出が困難である。
  - 考察1: 入力a_i, b_iに対してキャリーが生成される条件は?
    - Generator: $g_i = a_i b_i$
  - 考察2: 入力a_i, b_iに対してキャリーが伝播される条件は?
    - Propagator: $p_i = a_i + b_i$
- これでrippleを除去できただろうか?

$c_1 = g_0 + p_0c_0$
$c_2 = g_1 + p_1c_1 = g_1 + p_1(g_0 + p_0c_0) = g_1 + p_1g_0 + p_1p_0c_0$
$c_3 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$
$c_4 = g_3 + p_3c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

これは使える! (why)? => O(|| c_n||)、ただし|| c_n|| はp, gで表した標準系の項数

この原則を用いて、より大きな加算器を作成すると

- このままでは、16 bit や32bitの加算器は作れない。(上位桁のキャリー計算が複雑すぎる)
- アイデア: 4-bit CLA 加算器をripple carryで結んだら
- より良いアイデア: もう一度carry look-aheadを行う!! → CLAの階層化
  Q: このようにすると、キャリー算出のオーダーはどうなる?
- Q:これでうまくいくことを証明せよ。
  - ヒント
  - $C_1 = G_0 + P_0CarryIn$
    ...



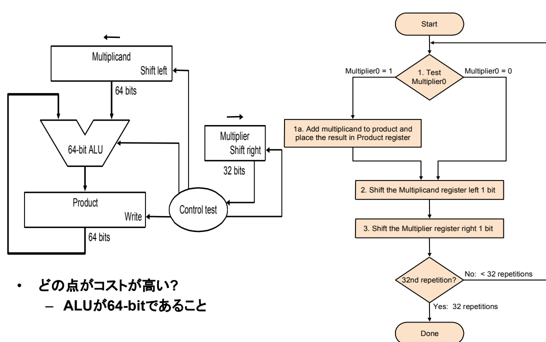
乗算回路

- 加減算よりずっと複雑
  - 基本的には、シフトと加算で計算される
- 加減算より、より時間がかかり、シリコン面積も食う
- 小学校で習ったアルゴルリズムから開始して、より複雑なアルゴリズム

```
   0010  (被乗数)(Multiplicand)
 x 1011  (乗数)(Multiplier)
```

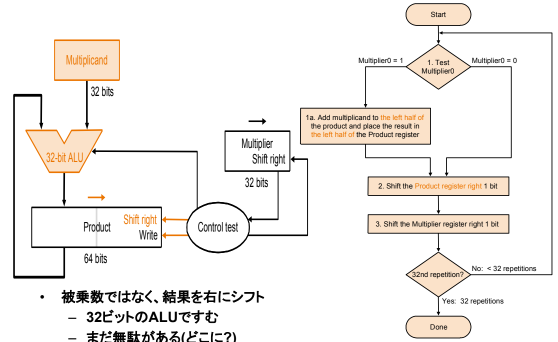
- 負の数: 変換し、乗算する
  - より良いアルゴリズムはあるが、この授業ではカバーしない

乗算回路: 実装(その1)



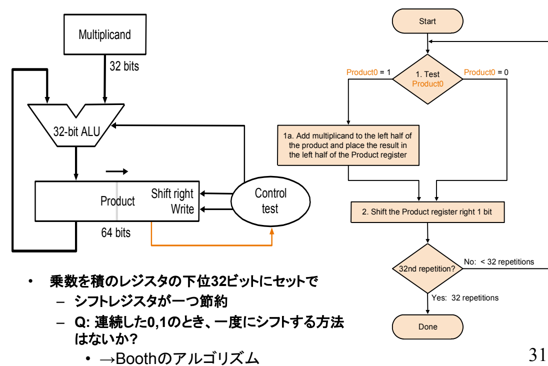
- どの点がコストが高い?
  - ALUが64-bitであること

乗算回路: 実装(その2)



- 被乗数ではなく、結果を右にシフト
  - 32ビットのALUですむ
  - まだ無駄がある(どこに?)

乗算回路: 実装(その3)



- 乗数を積のレジスタの下位32ビットにセットで
  - シフトレジスタが一つ節約
  - Q: 連続した0,1のとき、一度にシフトする方法はないか?
    - →Boothのアルゴリズム

浮動小数点 (概観)

- 以下のような実数を表現したい
  - 小数点以下の桁があるもの, e.g., 3.1416
  - 大変小さい数字, e.g., .000000001
  - 大変大きい数字, e.g., 3.15576×10^9
- 表現法:
  - 符号(sign), 指数(exponent), 仮数(significand):
    - $(-1)^{sign}$ x significand x $2^{exponent}$
  - 仮数部により多くのビットを割り当てると、精度が向上する
  - 指数部により多くのビットを割り当てると、表現可能な値の範囲が広がる
- IEEE 754 浮動小数点規格:
  - 単精度 (single precision): 8 bit exponent, 23 bit significand
  - 倍精度 (double precision): 11 bit exponent, 52 bit significand

IEEE 754 浮動小数点標準規格

- 仮数の一番上の桁の "1" bit は省略される (normalizeされるため)
  - これによって、精度が1ビット儲かる
- 指数部は「下駄履き」される (2の補数表現ではない)→なぜ?
  - all 0 が最小で all 1 が最大
  - 単精度では 127 の下駄履き、倍精度は1023
  - まとめると: $(-1)^{sign} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}$
- 例:
  - decimal: $-.75 = -3/4 = -3/2^2$
  - binary: $-.11 = -1.1 \times 2^{-1}$
  - 浮動小数点: 指数部 = 126 = 01111110
  - IEEE 単精度: 10111111010000000000000000000000

浮動小数点の複雑さ、困難さ

- 浮動小数点の演算は、桁あわせなどが生じ、複雑になる
- overflow に加えて、"underflow"も生じる(演算の結果が表現可能な最小数以下になる。)
- やはり精度が大きな問題
  - IEEE 754 は余分なguard と roundビットを持つ
  - 4種類の丸め方のモードがある
  - 正の0による割り算は "無限大"になる
  - 0割る0は "not a number"となる
  - 他にも、様々な算術演算状の問題がある
- 標準を実装するのは難しい→しかし、8087以降、ほとんどのプロセッサは採用

まとめ

- 計算機の演算は有限の精度によって縛られている(通常の数学との違い)
- ビットパターンはそれ自身は意味がないが、二つの規格がある
  - 2の補数表現(整数)
  - IEEE 754 浮動小数点
- 計算機の命令とその実行がビットパターンに数字としての意味を与える
- 性能や精度は実際のマシンでは大変大きな問題であり、アルゴリズム上、あるいは実装状の様々な問題がある。

- 次はいよいよCPU自身の実装である