第5回
2018/12/18

「数の表現・組合せ回路・演算器」

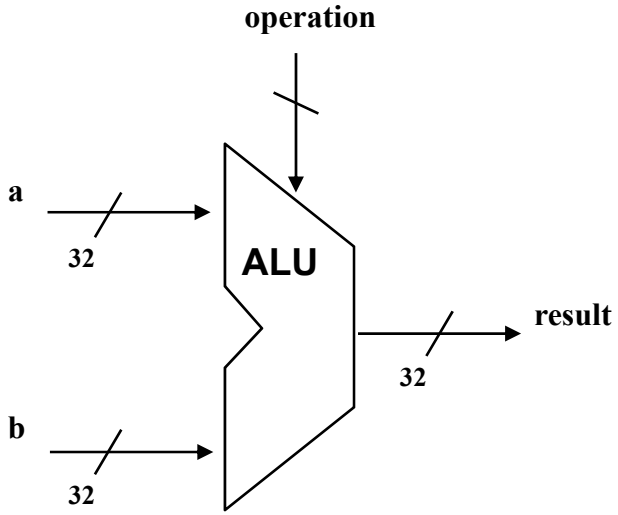
今回から数回に分けてプロセッサの「中身」

機械語から演算へ：ハードウェアアーキテクチャ

- これまで、機械の抽象化を学んできた
  - 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

  これらは、プロセッサと人間のインタフェース
- 次に行うこと:
  - プロセッサは、どのように実現されるのか？

  → デジタル回路技術による、その仕組みは？



計算機内の数字の表現

- ビット列は単にビット列であり、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
  0000 0001 0010 0011 0100 0101 0110...
  ビット列をどのように数値(まずは整数)と解釈するか
  特に負の数が登場すると、規約は自明ではない

- まず整数について取り上げる。実数は今日の最後

負の整数の可能な表現法

n-bitあると、2^n通りの数値を表現可能

- 符号+絶対値
  Signed Magnitude
  符号ビット+abs(x)
  000 = +0
  001 = +1
  010 = +2
  011 = +3
  100 = -0
  101 = -1
  110 = -2
  111 = -3
  ↓
  符号ビット

1の補数
One's Complement
負数はビット反転
000 = +0
001 = +1
010 = +2
011 = +3
100 = -3
101 = -2
110 = -1
111 = -0
↓
符号ビット

2の補数
Two's Complement
負数 x は 2^n - abs(x)
000 = +0
001 = +1
010 = +2
011 = +3
100 = -4
101 = -3
110 = -2
111 = -1
↓
符号ビット

これが主流

- 違い: 表せる正負のバランス、ゼロの表現数、演算や操作の容易性

MIPSにおける数の表現

- 32 bitの符号なし整数, C言語のunsigned int相当

  $0 \sim 2^{32}-1$ (= 4,294,967,295)

- 32 bit の符号付き整数, C言語のint相当: 2の補数表現

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ *maxint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

最上位ビット (MSB=most significant bitと呼ぶ)が、1なら負の数
ちなみに、最下位ビットは(LSB = least sigfinicant bit)と呼ぶ

2の補数に対する基本操作

- 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
  - 符号反転 =「全ビット反転し、1を足す」
  - [Q] 上記が、正から負でも、負から正でも成り立つことを確かめよ
- n ビットの数をnビット以上の数に変換
  - たとえば、MIPSの 16 bit の即値 (immediate)は32 bitに自動的に変換されて解釈される
  - 「増えたbitに0をつける」ではダメ
  - 符号ビット(sign bit)であるMSBを他のビットにコピー

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

  - これを“符号拡張”と呼ぶ

加算と減算

- 2進数の加算 → 基本は小学校の桁上がり

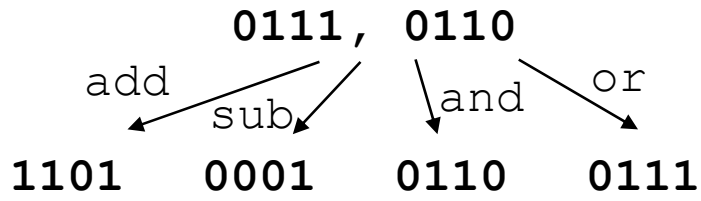
```
  0111
+ 0110
  1101
```

- 2進数の減算→ a-b を a+ (-b)と解釈

```
  0111          0111
- 0110    ⇒  + 1010
                ×10001      答えは1
```

- 正+正、正+負・・・が同じ論理で可能 → 2の補数表現が主流な理由

ビット表現された数値を演算



- 今回とりあげるのは、2つの数値を入力し、1つの数値を出力する演算

- このような演算を、プロセッサはどう実現しているか

→ ブール代数・論理ゲート・論理回路の登場

ブール代数

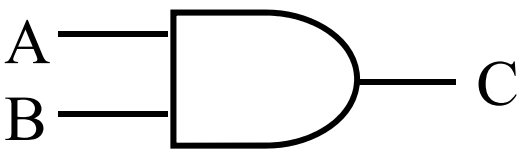
- ブール代数：変数がすべて0か1である代数
- 代表的な演算子
  - OR (+): A+B → Aが1かBが1なら、1。両方0なら0 (加算とは違う)
  - AND (･): A･B → AもBも1なら1。違うなら0
  - NOT: $\overline{A}$ → Aが0なら1, 1なら0
- 真理値表：
  - 入力の全パターン → 出力の対応表

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

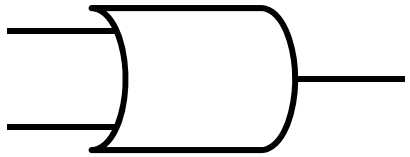
[Q]「積和標準形」と真理値表の関係は?

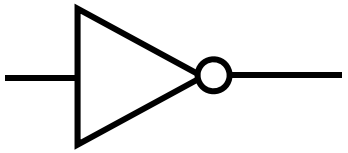
A	B	C	$(A \cdot B \cdot \overline{C})+(\overline{A} \cdot B)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

論理ゲート



AND

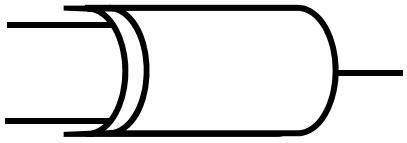

OR


NOT

ほか、代表的なもの


NAND


NOR


XOR

A	B	NAND	NOR	XOR
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

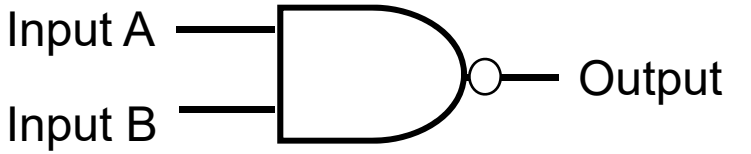
※あらゆる論理式はNANDゲートだけ(もしくはNORだけ)の組み合わせで表現できる

算術演算の回路を含め、プロセッサは多数の論理ゲートからなる

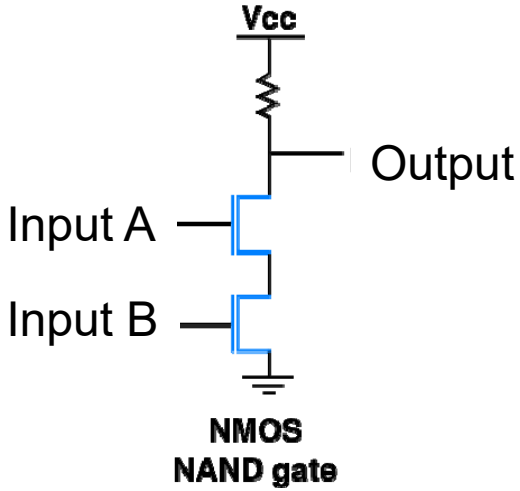
余談：ゲートとトランジスタ

- 「プロセッサは多数のゲートから成る」
- 「プロセッサは多数のトランジスタから成る」

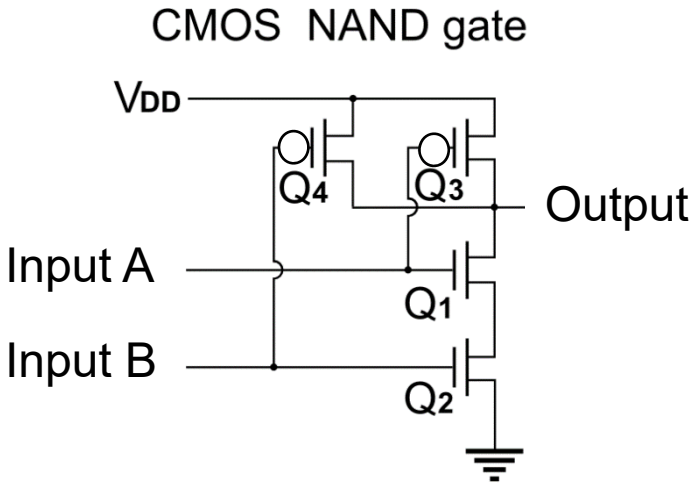
両者の関係は？ → ゲートは複数のトランジスタから成る



NANDゲートの中身？

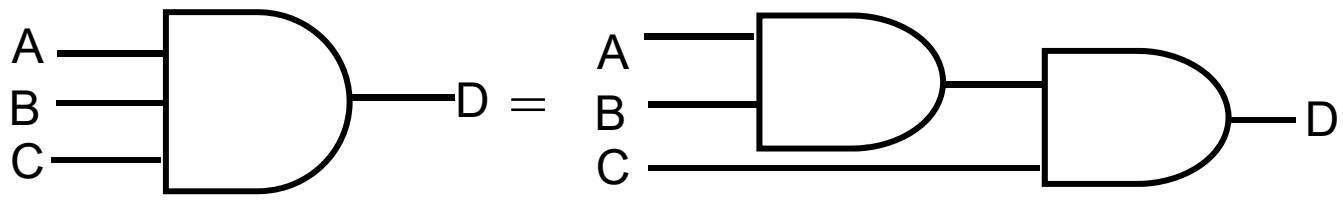


NMOS型NANDゲート
(トランジスタ2つ)

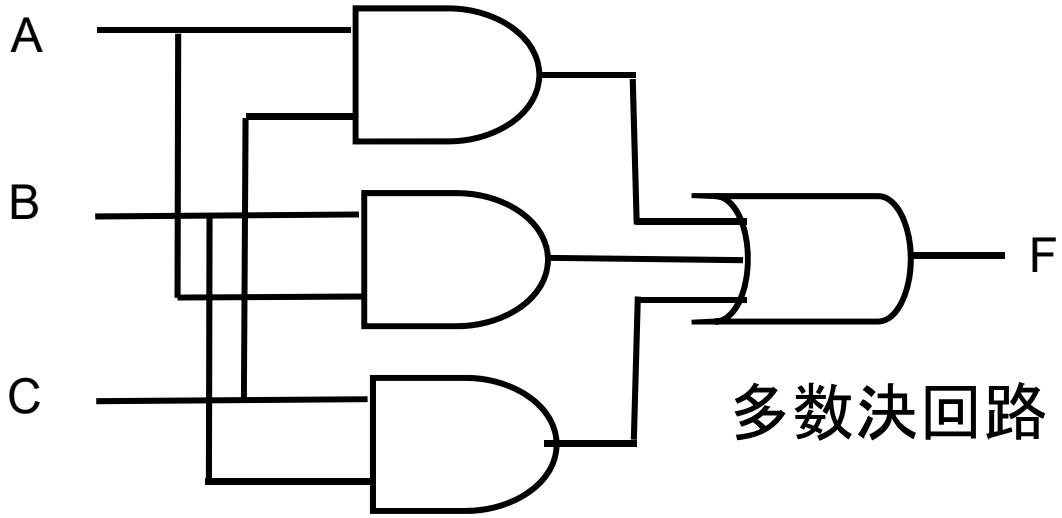


CMOS型NANDゲート
(トランジスタ4つ)
→ 省電力などのため主流

論理回路:ゲートの組み合わせ



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	F
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

組み合わせ回路と逐次回路

論理回路(logic circuit)は、大きく分けて2種類

組み合わせ回路 (combinational circuit)：　記憶・状態を持たない論理回路であり、数学の関数のよう(入力が決まれば出力は一意)に動作する

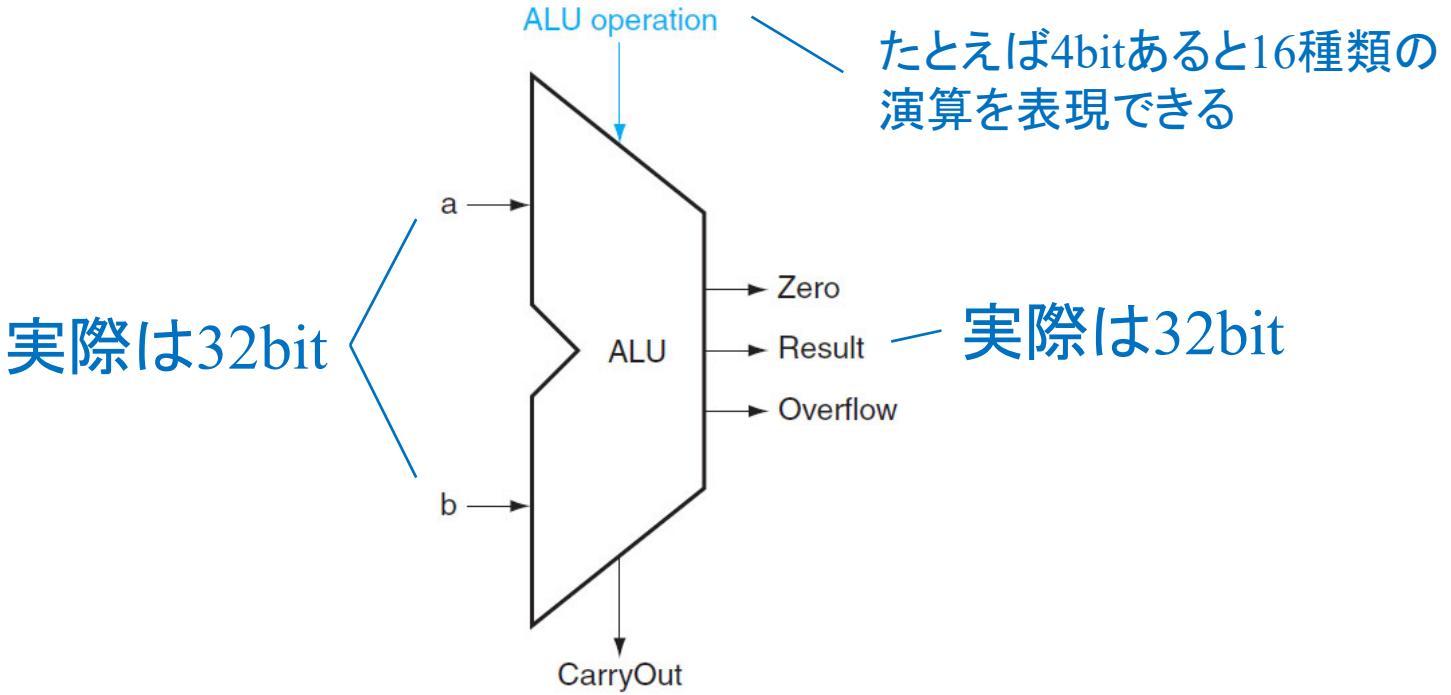
- 今回の授業の例はほぼすべて組み合わせ回路

問題：これだけでは「レジスタ」が作れない

→ 逐次回路 (sequential circuit)の必要性 → 次回！

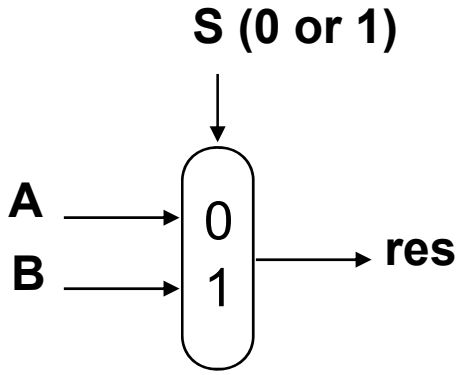
演算装置: ALU (Arithmetic Logic Unit)

- 今日の山場：2つの32bit整数を入力にとり、1つの32bit整数を出力する組み合わせ回路を設計する。
- 出力は、入力のadd, sub, and, or, slt (set less than)などの結果
- 演算種類を、”operation”という別入力で決められる
- このような回路ブロックを、ALUと呼ぶ



重要な部品：マルチプレクサ (Multiplexor)

- 制御入力Sに基づいて、入力を一つ選択する

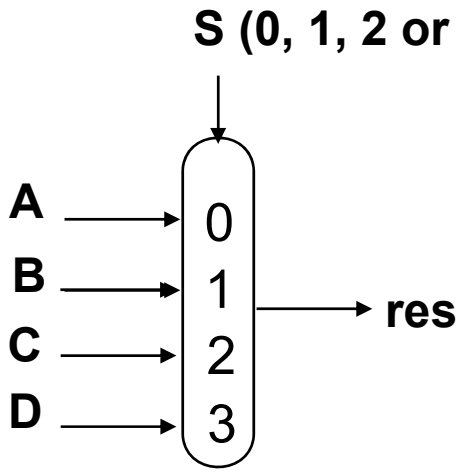


この図は、2入力のマルチプレクサ
入力は、実際はA, B, Sの3つ
Sが0ならAを、1ならBを出力

[Q] 真理値表を書いてみよう
[Q] どのようにゲートから実現？

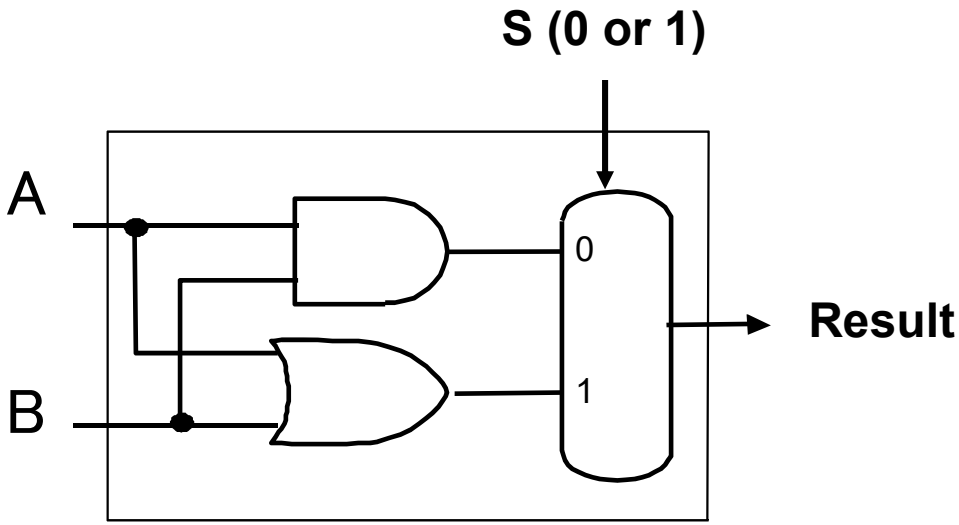
- 一般的には、多入力マルチプレクサ：2^n入力とnビットの制御入力S

4入力2bitの例



ALUの出発点

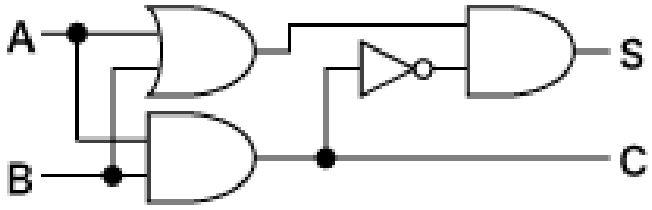
- 1bit のANDやORは、ゲート1個でできる
- 「ANDとOR両方」に使える回路を作るには？
  - 複数方法は考えられるが、マルチプレクサを使った例：



この1bit ALU (単純版)を32個使うと、addi命令とori命令に対応可能な、32bit ALUができる

1bit加算

- 加算は、and/orと異なり、桁上がり(carry)を考える必要
- 1bit + 1bitの加算結果は、00, 01, 10のいずれか → 2bit必要



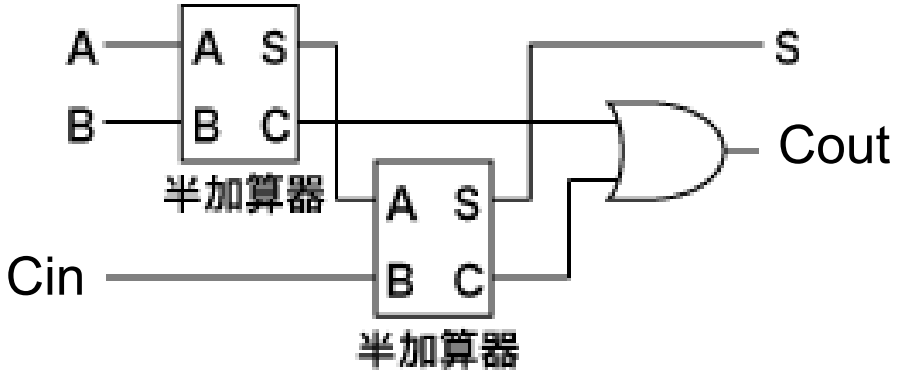
(Sum)

(Carry out)

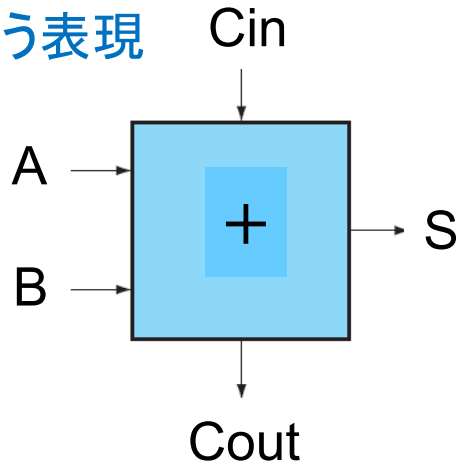
1bitの 半加算器 (Half adder)

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

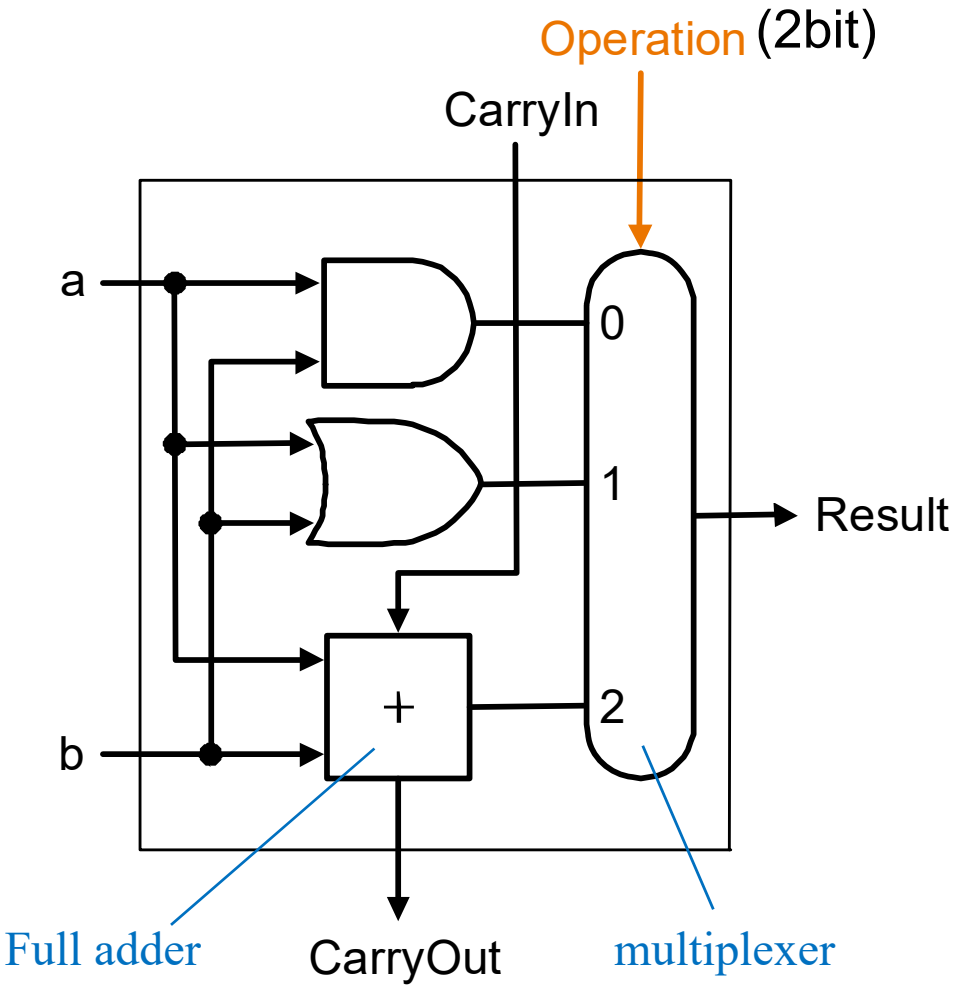
- 複数bitの加算を可能にするには、下の桁からやってくる”carry-in”も考える必要
  → 全加算器 (Full adder)



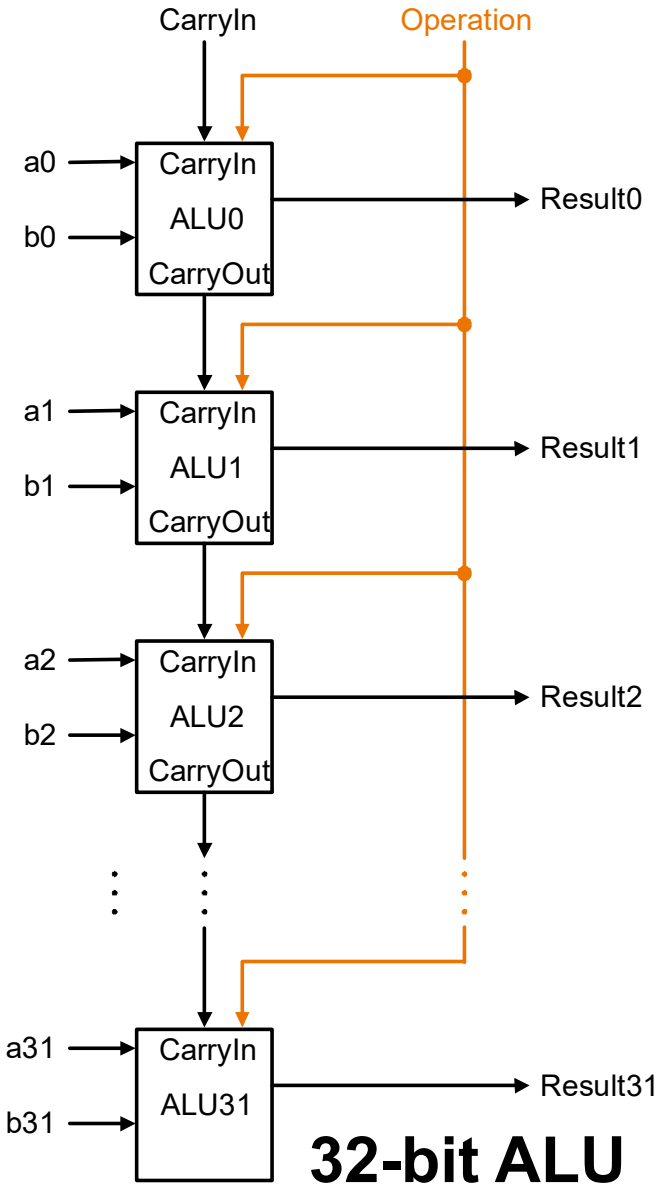
以降、こう表現



and/or/addに対応するALUの構成



1-bit ALU



32-bit ALU

減算（a – b）の実現は?

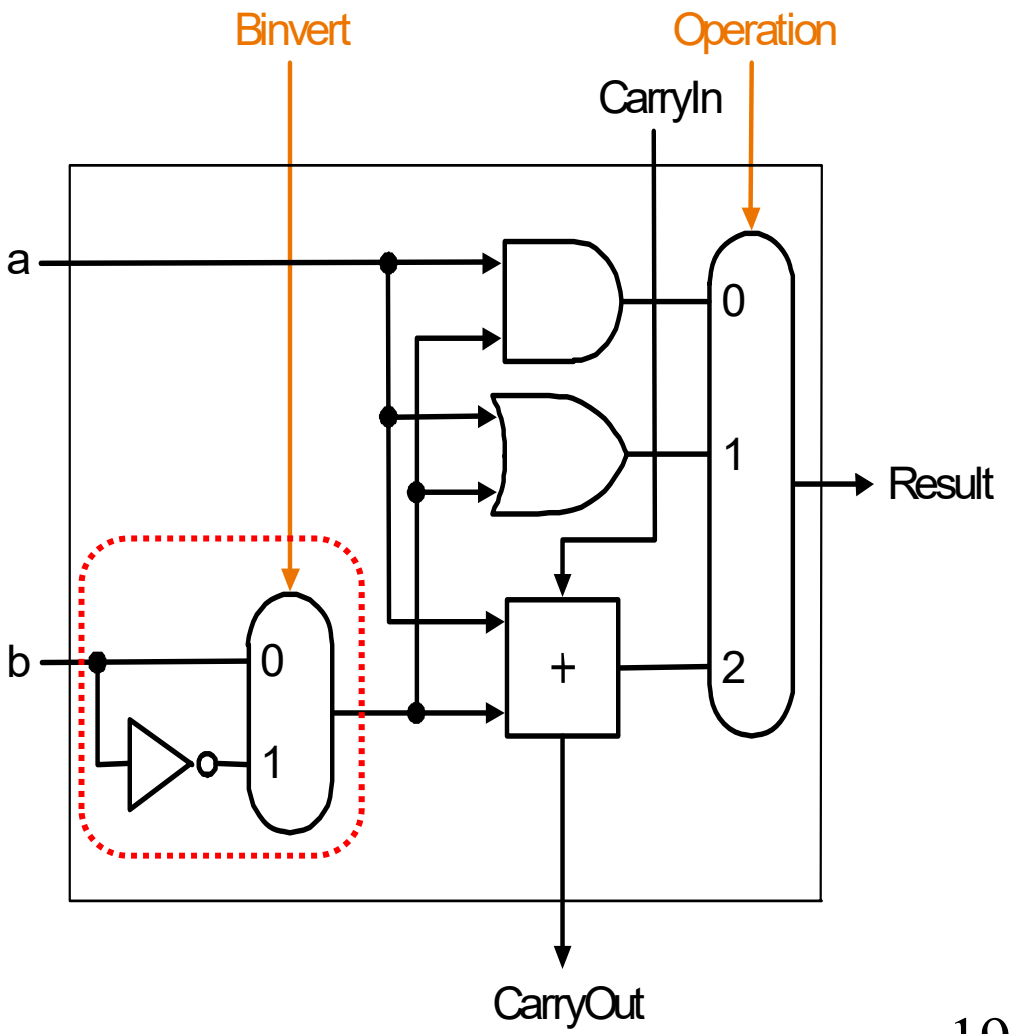
- 2の補数を用いる: bを符号反転して、加算すれば良い
- 符号反転の方法は？

(1) bをビット反転させる

(2) 1を加算する

(1)の実現：各1-bitの加算器に入力する前にビット反転

(2)の実現：最下位bitのCarry-inを1にする

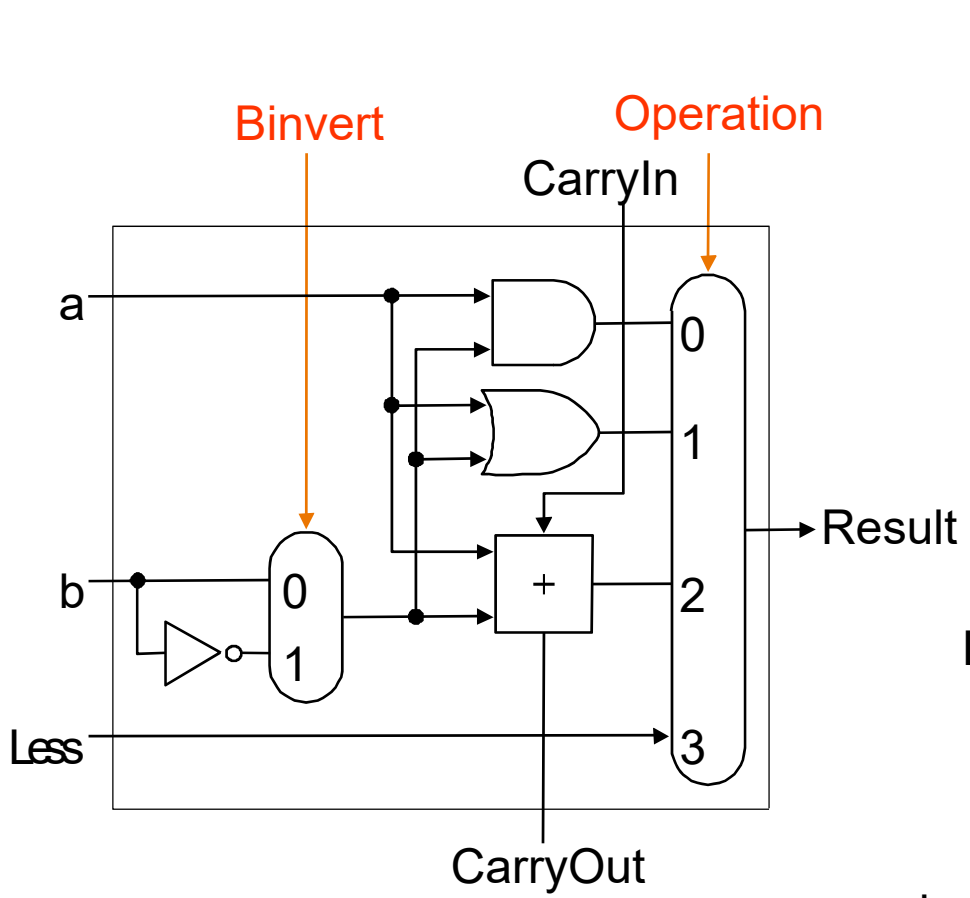


ALU の他の MIPS命令の実装

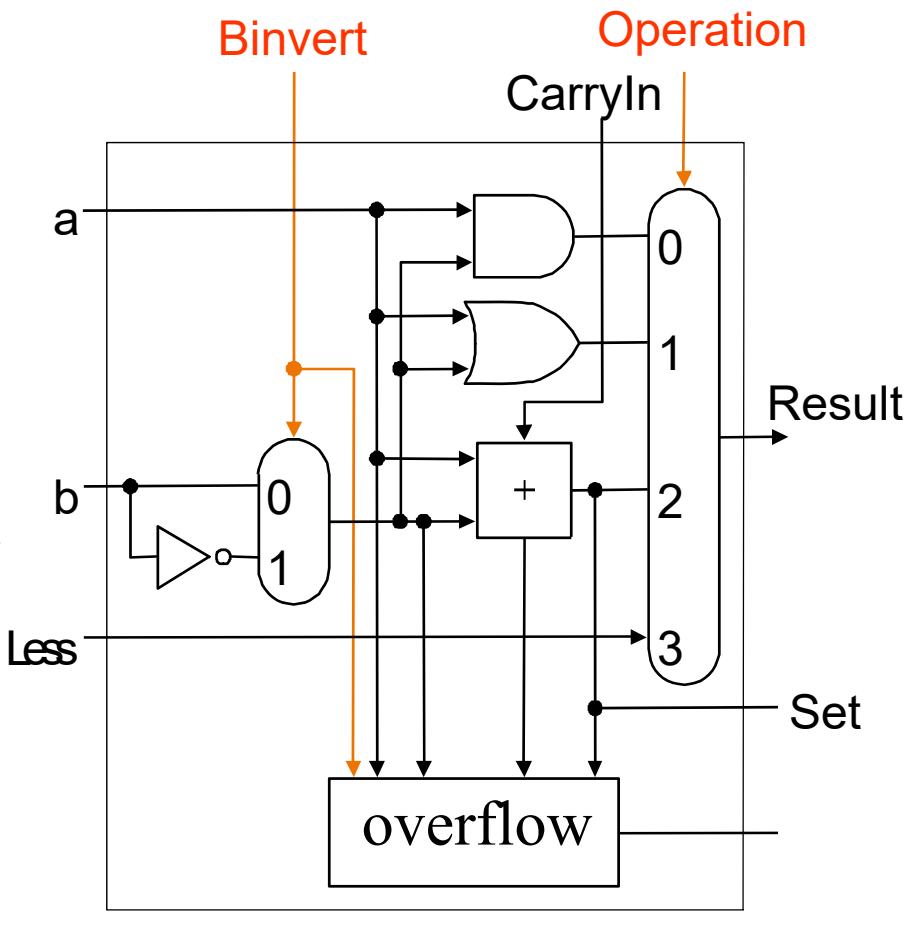
- set-on-less-than 命令 (slt)のサポート
  - 入力1<入力2なら、出力は1, そうでなければ0
  - 算術命令の一種
  - 減算を用いれば良い: (a-b) < 0 は a < bと等価

- 等号演算も必要 (beq命令)
  - 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

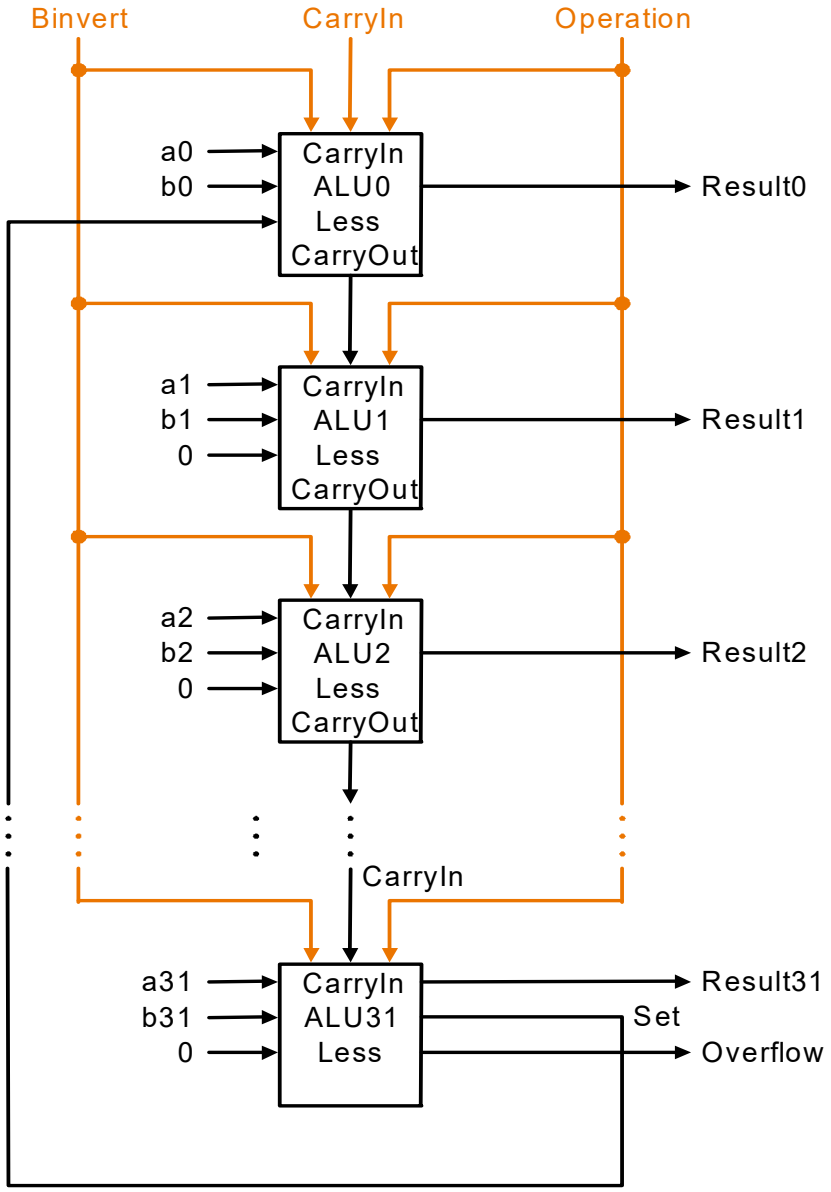

Binvert
Operation
CarryIn
a
b
Less
0
1
+
0
1
2
3
Result
CarryOut

a


Binvert
Operation
CarryIn
a
b
Less
0
1
+
0
1
2
3
Result
Set
overflow

b

Sltの実装(続き)



減算の結果がマイナス

MSB (31ビット目)が1

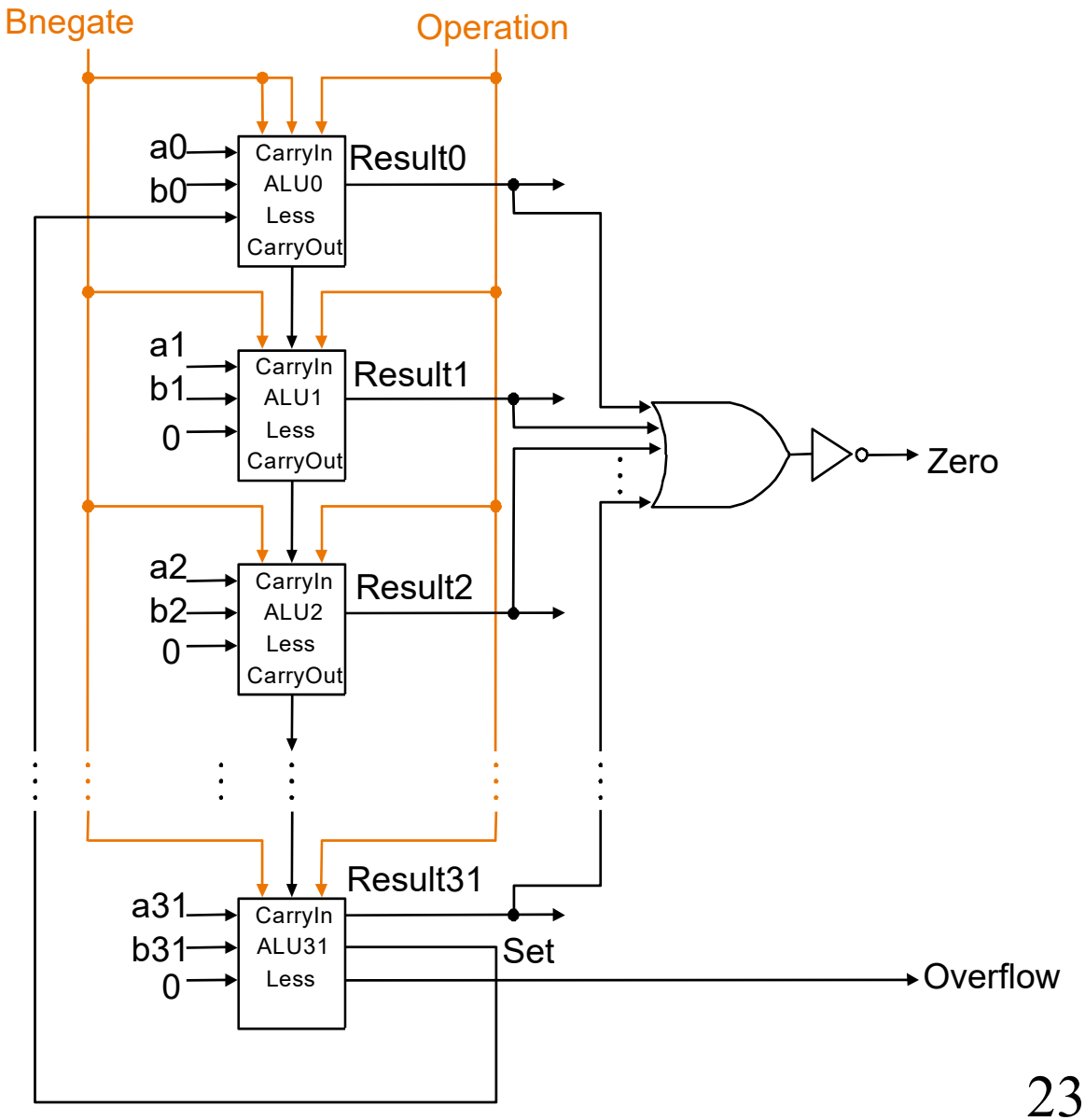
その結果をResult0に反映
他の桁のResultは0に

等号演算

- 制御信号に注意:

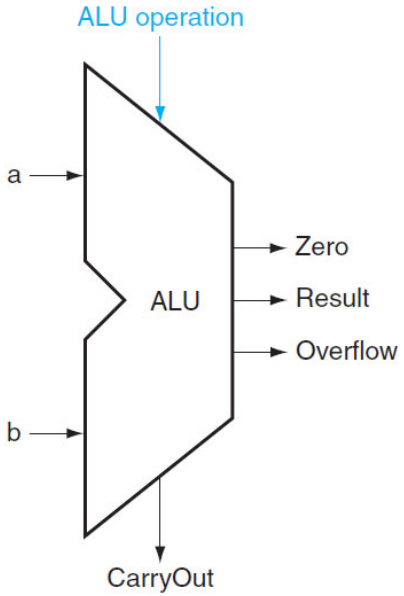
```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

- *注: Zeroは結果が演算の結果が0のとき1となる*



ここまでのまとめと、これからの議論

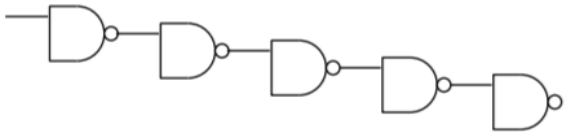
- **MIPSの命令セットを実装するALUの一部を構築した**
  - **複数演算に対応 → マルチプレクサを用いて、必要な出力を選択**
  - **1-bit ALU を複製することによって、32-bitのALUを構築できる**
  - **2の補数を用いることによって、減算やその他の比較演算を容易に実装**



ここからの議論:

「速度/性能」を考えると、改良が必要である(特に加算)

- 必要な理由:ゲートの動作にはlatency(遅延時間)がある(1ns以下だが)
- 全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
- 直列接続のうち、最も長いものを “クリティカルパス (critical path)”と呼ぶ



Ripple Carry Adder

- これまで学んだ加算器:
  - 1bit Full-adderをつなぎあわせる
  - (小学校で学んだ)桁上がり方法で計算

  → Ripple carry adder (ripple: 波状)と呼ばれる
- わかりやすいが、遅いという欠点
  - 32bit加算器のクリティカルパス長は1bitの約32倍 → 遅延が約32倍

- 加算は重要な演算なので、遅くては困る
- 特に、MIPSプロセッサでは命令ごとに毎回 PC = PC+4 が必要！
  - PC: program counter

Carry Lookahead Adder (CLA)

- 「i-bit目の桁上がりを、Ripple carryより高速に計算できないか？」
- i-bit目の加算器のCarry-inをc_i, Carry-outをc_{i+1}とする
- 一般式: $c_{i+1} = (a_i \cdot b_i) + ((a_i + b_i) \cdot c_i)$
- $g_i = a_i \cdot b_i$, $p_i = a_i + b_i$を使って書き直すと$c_{i+1} = g_i + (p_i \cdot c_i)$

漸化式の展開:

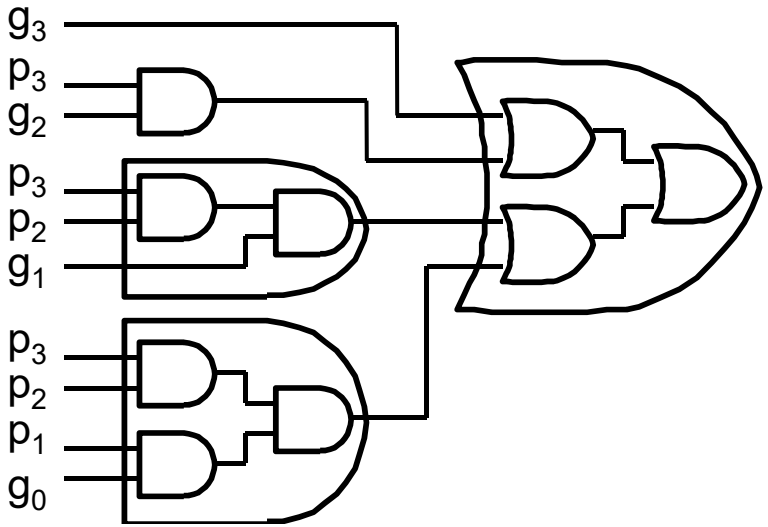
$$c_1 = g_0 + p_0.c_0$$

$$c_2 = g_1 + p_1.g_0 + p_1.p_0.c_0$$

$$c_3 = g_2 + p_2.g_1 + p_2.p_1.g_0 + p_2.p_1.p_0.c_0$$

$$c_4 = g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0 + p_3.p_2.p_1.p_0.c_0$$

c_4を計算する回路



Carry Lookahead Adder (CLA) (続き)

Ripple Carryと何が変わった？

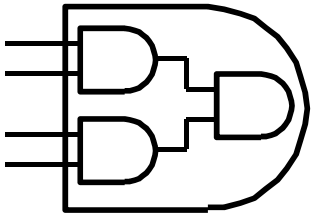
n入力ANDゲートとn入力ORゲートがあれば、c4を求めるクリティカルパス長は3ゲート分 (に見える)

1. g_i, p_iを独立に求めるゲート
2. ANDゲートたち ($p_3.p_2.p_1.p_0.c_0$など)
3. 最後にまとめるORゲート

では、任意のビット長の加算を高速にできるか？話は簡単ではない

- n入力ゲートの遅延は、2入力ゲートより遅い。O(log n)で近似
- ビット長が多くなると回路が複雑

→ 4bit程度で留めるのが通常

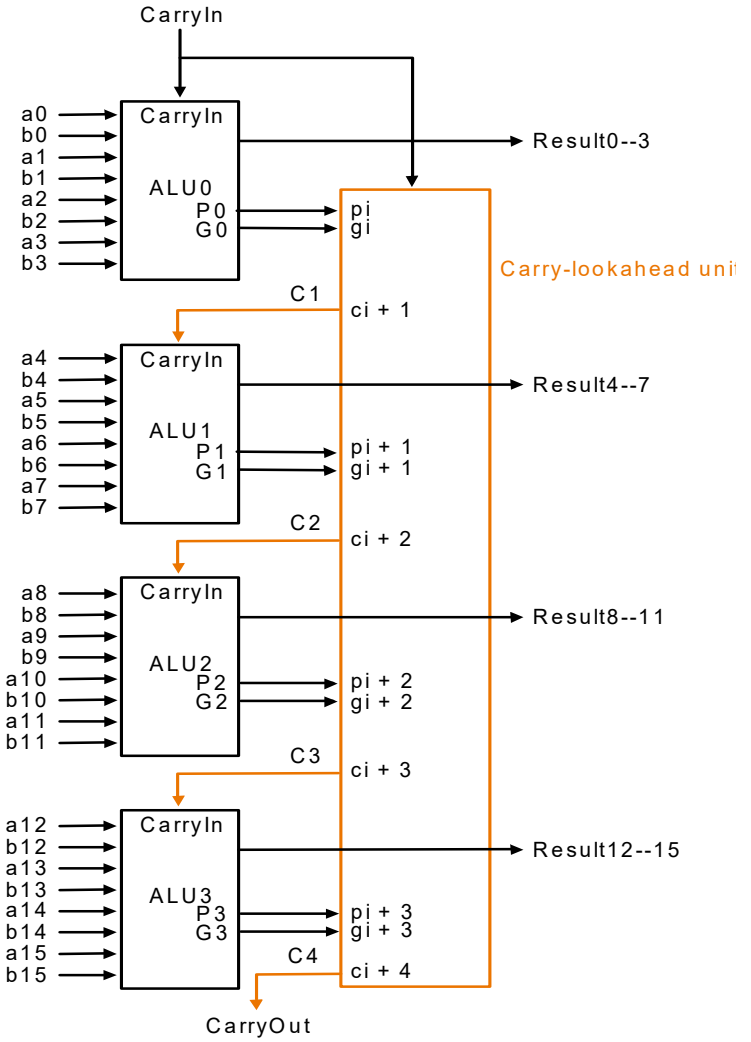


- 32bit/64bit 加算をCLAの考えを使って行うには？

4bit CLAを用いてより大きな加算器を作成

- アイデア1: 4-bit CLA 加算器を ripple carryで結ぶ？
- アイデア2: 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?



乗算について

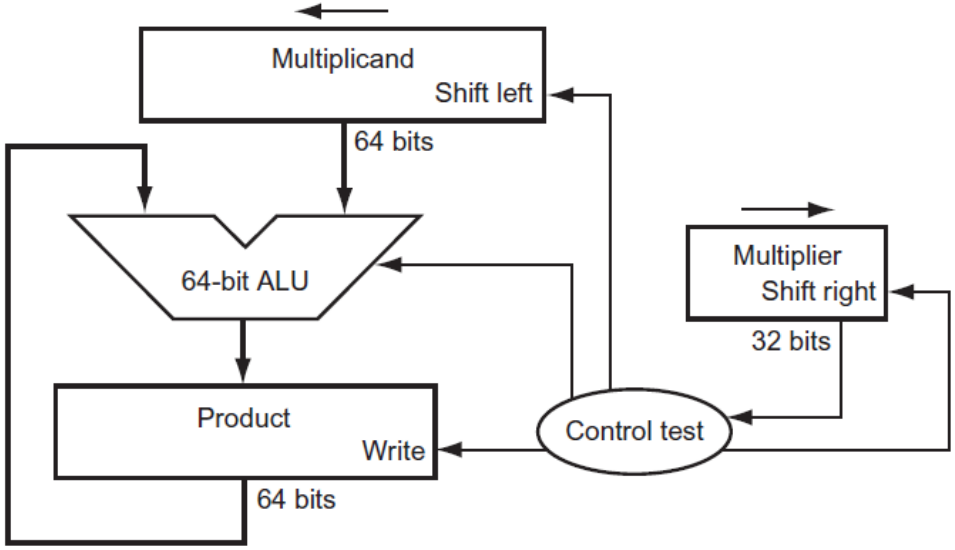
- 加減算より複雑 (多数のゲートを必要)
- 基本は小学校で習った計算方法
  - シフトと加算で計算される

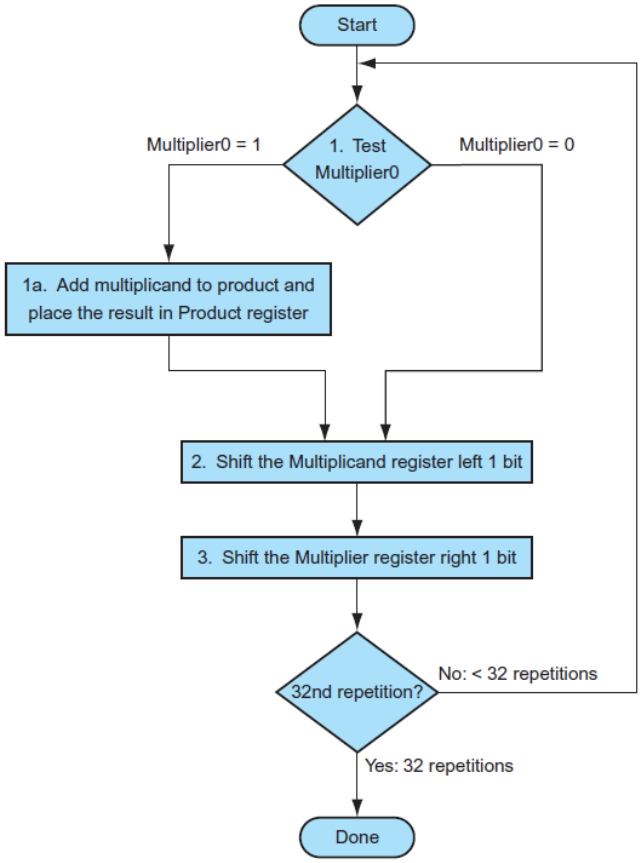
```
    1010   (被乗数)(Multiplicand)
  x 1011   (乗数)(Multiplier)
    1010
   1010
  0000
 1010
1101110
```

一般的に、m-bit x n-bitの結果は最大(m+n)bit。MIPSの場合は

- mul命令: 下位32bitだけ残す
- mult命令: (32+32)bitの特殊レジスタを使う

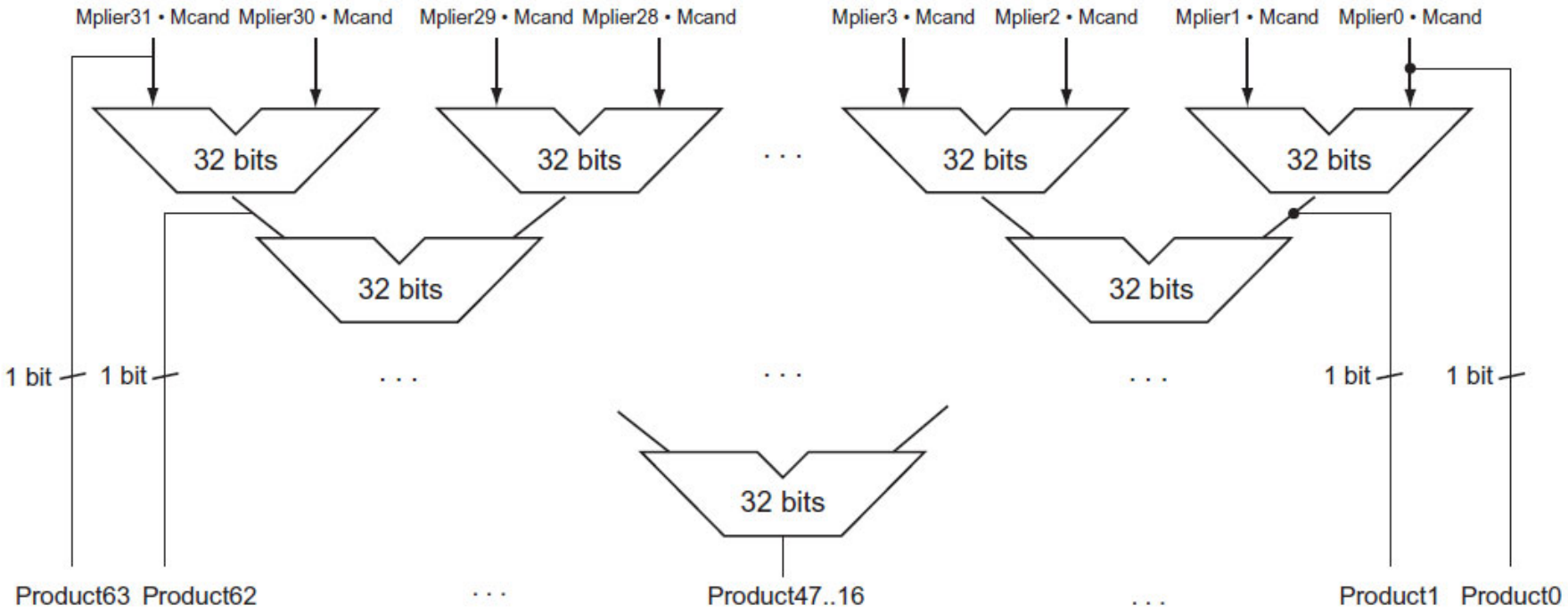
乗算回路: 古典的な実装





※この例は、これまで学んできた
組み合わせ回路ではない。順序回路
最大の欠点は遅さ:n-bitの演算にO(n)以上の遅延

より高速な乗算回路



- この方法は組み合わせ回路
- n-bitの乗算が、(加算の遅延)×O(log n)で可能
- 前ページより大量にゲート/トランジスタを用いるが大丈夫か？

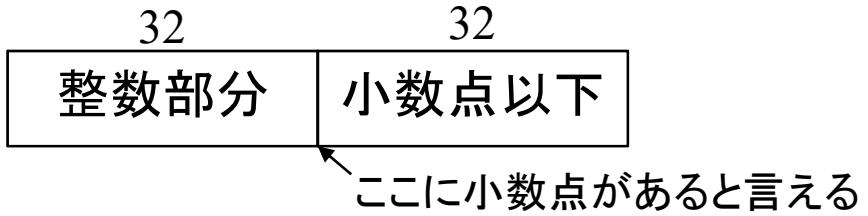
→ Mooreの法則によるトランジスタ数増加は、主に高速化のために使われてきた

実数の計算機上の表現

- 実数を表現したい
  - 3.14159, 1.0/3.0
  - 1.2×10^{-15} (陽子半径(m)), 6.02×10^{23} (mol数), …
- 32bitや64bitなどの、限られたビット数でどうやって表現？
- まず思いつく方法：固定小数点フォーマット

  例：64bitで表せる整数 $\times 2^{-32}$

  このとき、上位32bitが整数部分、下位32bitが小数部分



  この方法では：
  - 表せる範囲：約-2^{31} ～ 2^{31} (10進9桁程度)
  - 分解能：2^{32} (小数点以下9位程度)

  1.2×10^{-15} (陽子半径(m)), 6.02×10^{23}は表せない！

  → 主流の表現法は、浮動小数点フォーマット

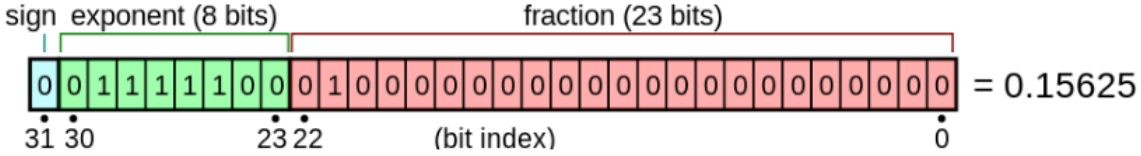
浮動小数点の考え方

- 実数の指数表現 6.02×10^{23} の考え方を使う

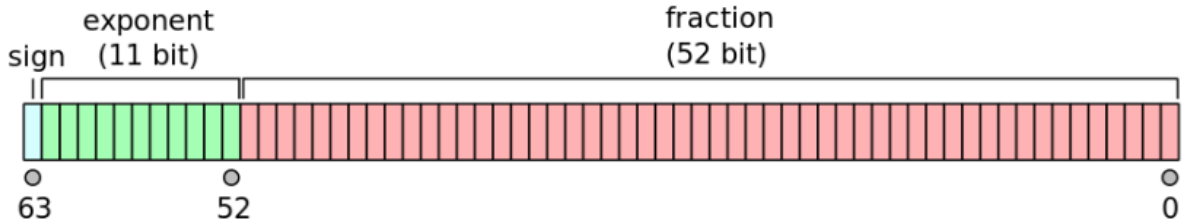
  仮数(fraction/significand)　指数(exponent)

  2進で $1.11110001\ldots \times 2^{88}$

- 符号(sign), 指数(exponent), 仮数(fraction) の3つで実数を表現
- IEEE 754規格で定められるものが主流
  - 単精度 (single precision), 32bit/FP32, C言語のfloat → 有効数字約7桁



  - 倍精度 (double precision), 64bit/FP64, C言語のdouble → 有効数字約16桁



  - ほか、2017年ごろから半精度/FP16にも注目 (s1 + e5 + f10)

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
  - 1≦仮数<2 なので、自明な1を省く → 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)
  - all 0 が最小で all 1 が最大
  - 単精度では bias=127 の下駄履き、倍精度ではbias=1023

まとめると:　表される実数は、$(-1)^{sign} \times (1+fraction) \times 2^{(exponent - bias)}$

- 例: 10進で-0.75
  - 2進表現: -0.11 = -1.1×2^{-1}
  - 指数部 -1+127 = 126 = 01111110

  - IEEE 単精度: 10111111010000000000000000000000

浮動小数点の複雑さ

- 浮動小数点の演算は、桁あわせなどが生じ、複雑になる
  - 加算 (MIPSのadd.s, add.f命令): 指数が大きいほうに合わせて仮数のシフト等が必要
- 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こと)
- やはり精度が大きな問題
  - 下記のC言語を実行してみる

```
double a = 1.0/3.0;
printf("a = %.20f¥n");      // 小数点以下20桁まで表示
```

→ a = 0.33333333333333331483

[Q]doubleの代わりにfloatではどうか？ 6.02 x 10^{23} を表示するとどうなるか？

- 浮動小数点演算を回路で表現するのは難しい
  - 1980年ごろまではソフトで実装
  - (難しいが)近年のプロセッサのほとんどが、ハードウェア(回路)で実装
  - 浮動小数点演算を1秒で何回行えるかが、計算機のメジャーな指標に(Flops)

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
- 演算はゲートの組み合わせである論理回路、特に組み合わせ論理回路で実装される
  - ある演算を実現する回路は1通りではない。クリティカルパスやゲート数の異なるいくつもの方法

- 今回の内容では、まだ「次々に命令を読み取って実行」するプロセッサが実現できない。次回の順序回路で可能に