名古屋工業大学 先進セラミックス研究センター　井田 隆 2024 年 3 月 19 日更新

3．コンピュータの基礎（3）–算術演算–
Arithmetic operation

ディジタル・コンピュータを使った計算のしかたを紹介します。普通の加減乗除の計算のことを「算術演算 arithmetic operation」と言います。ディジタル・コンピュータでは数値を二進数 (binary number) を使って表します。二進数の算術演算は，前に説明した「論理演算 logical operation」の組み合わせで実現できます。

3–1　論理回路による足し算　Addition with logical circuits

二進数の 1 桁は 0 か 1 の値しかとりません。二進数での 1 桁のことを 「1 ビット bit」と呼びます。はじめに，「1 ビットの足し算回路」について考えます。十進数 (decimal number) の表記で

$$0+0=0$$
$$0+1=1$$
$$1+0=1$$
$$1+1=2$$

という 4 通りの場合について正しい答えを出力するための回路です。十進数での 2という数は二進数 (binary number) では 10 と表されます。このことをはっきりとさせるために，

$$(2)_{\mathrm{d}}=(10)_{\mathrm{b}}$$

のように表す場合があります。上の例を二進数で表現すると，

$$0+0=00$$
$$0+1=01$$
$$1+0=01$$
$$1+1=10$$

と書けます。

このような計算をどのような電子回路で実現できるでしょうか？おおまかには，２本の入力信号線と２本の出力信号線を備えた回路になるはずです。足し算の２つの入力を A, B の２本の入力信号線に割り当てて，出力のうち「$2^0=1$」の桁の数字を Y_0 という出力信号線に割り当て，「$2^1=2$」の桁の数字（繰り上がり）を Y_1 という出力信号線に割り当てます。全体としては，Fig. 3.1.1 のような形になるはずです。

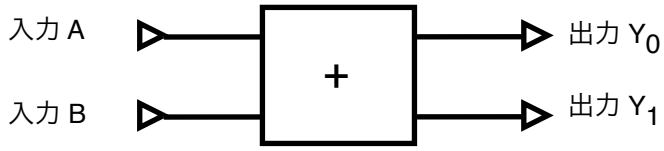

Fig. 3.1.1　1 桁の足し算回路。Y_0 は 2^0 の桁の数，Y_1 は 2^1 の桁の数を出力する。

Fig. 3.1.1 の中で □ で表されている回路の中身を考えましょう。出力 Y_0，Y_1 と入力 A, B の間には Table 3.1.1 のような関係があります。

Table 3.1.1　「1 ビット」の足し算回路の論理値表。
入力 A, B と，2^1 の桁の出力 Y_1，2^0 の桁の出力 Y_0

A	B	Y_1	Y_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 3.1.1 で表されるような関係を論理式を使って書くために，例えば次のように考えます。Table 3.1.1から，「出力 Y_0 が 1 になる」のは，「$A = 0$ かつ $B = 1$」の場合と，「$A = 1$ かつ $B = 0$」の場合との二通(ふたとお)りあります。これを論理式で表せば，

$$Y_0 = \left(\overline{A} \wedge B\right) \vee \left(A \wedge \overline{B}\right)$$

となります。この関係は，第 1 回の論理回路に出てきた「排他的論理和 XOR」と同じです。つまり，XOR 回路を使えば一桁の足し算の $2^0 = 1$ の桁の数値を得る回路になります。

つぎに，出力 Y_1 と入力 A, B の間の関係は，

$$Y_1 = A \wedge B$$

となります。この関係は，「論理積 AND」そのものです。つまり，AND ゲートを使えば 1 桁の足し算の繰り上がりの部分，$2^1 = 2$ の桁の数値を得る回路になります。

これらのことをまとめれば，１ビットの足し算をするための電子回路としては全体として，Fig. 3.1.2 のような構成にすれば良いと考えられます。この「１ビットの足し算回路」は**半加算器** half adder（ハーフ アダー） とも呼ばれます。

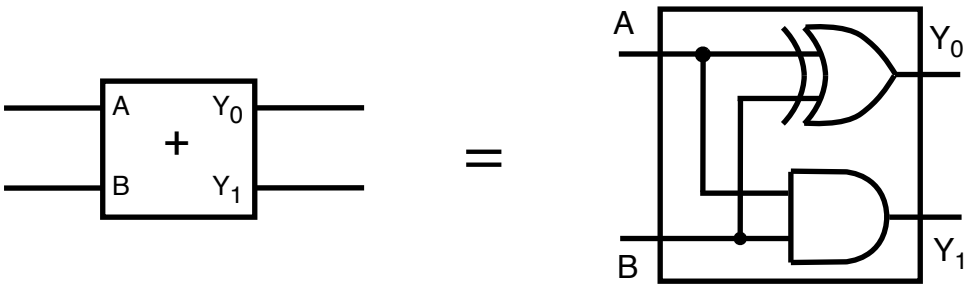

Fig. 3.1.2　「１ビット」の足し算回路 (2)（半加算器）
XOR ゲートと AND ゲートの組み合わせで実現できる。

次に，任意の桁数の足し算回路について考えます。足し合わせる２つの数が両方とも２進数で N 桁だとします。つまり，２つの数 A, B がそれぞれ，

$$A = A_{N-1}A_{N-2}\cdots A_j\cdots A_1A_0$$
$$B = B_{N-1}B_{N-2}\cdots B_j\cdots B_1B_0$$

のように表されるとします。これらを足し合わせた数 Y は，最大で $N+1$ 桁になります。和 Y を

$$Y = Y_NY_{N-1}\cdots Y_j\cdots Y_1Y_0$$

と表すことにします。

足し算の結果の 2^j の桁の数字 Y_j は，A_j と B_j，それと 2^{j-1} の桁からの**繰り上がり**（くりあがり） carry（キャリー） によって決まります。2^j の桁の二つの入力を A, B，2^{j-1} の桁からの繰り上がりを C，2^j の桁の出力を Y_0，2^j 桁から 2^{j+1} 桁へのの繰り上がりを Y_1 と表すことにすれば，各桁について Table 3.1.2 の論理値表で表されるような「３入力１ビット足し算回路」を使えば良いことがわかります。

Table 3.1.2　３入力１ビット足し算回路の論理値表

A	B	C	Y_1	Y_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

「２入力１ビット足し算回路」を Fig. 3.1.2 の記号で表すことにすれば，これを使って「３入力１ビット足し算回路」を Fig. 3.1.3 のように組み立てることができます。このような働きをする回路のことは**全加算器 full adder**（フル アダー）とも呼ばれます。

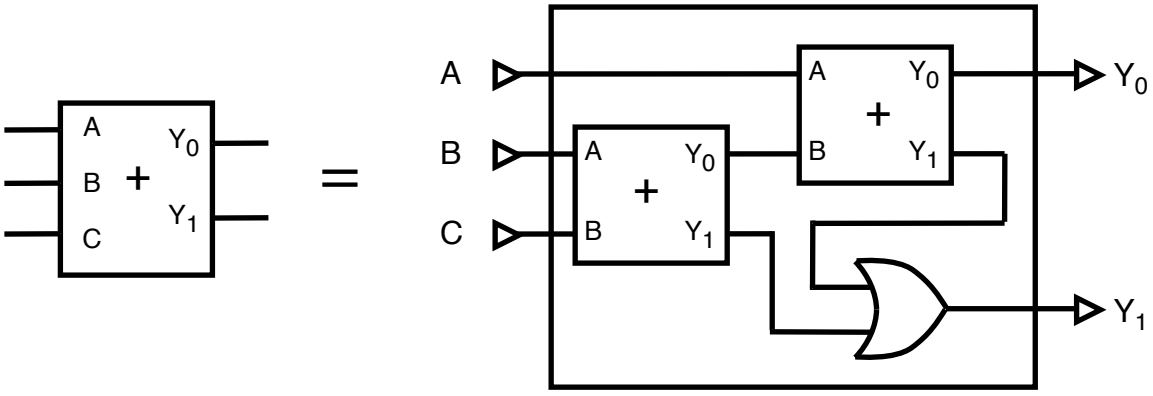

Fig. 3.1.3　「３入力１ビット足し算回路」（全加算器）

Fig. 3.1.3 で表されるような全加算器を必要な数だけ集めれば，任意のビット数の算術和を求める回路を作れることになります（Fig. 3.1.4）。

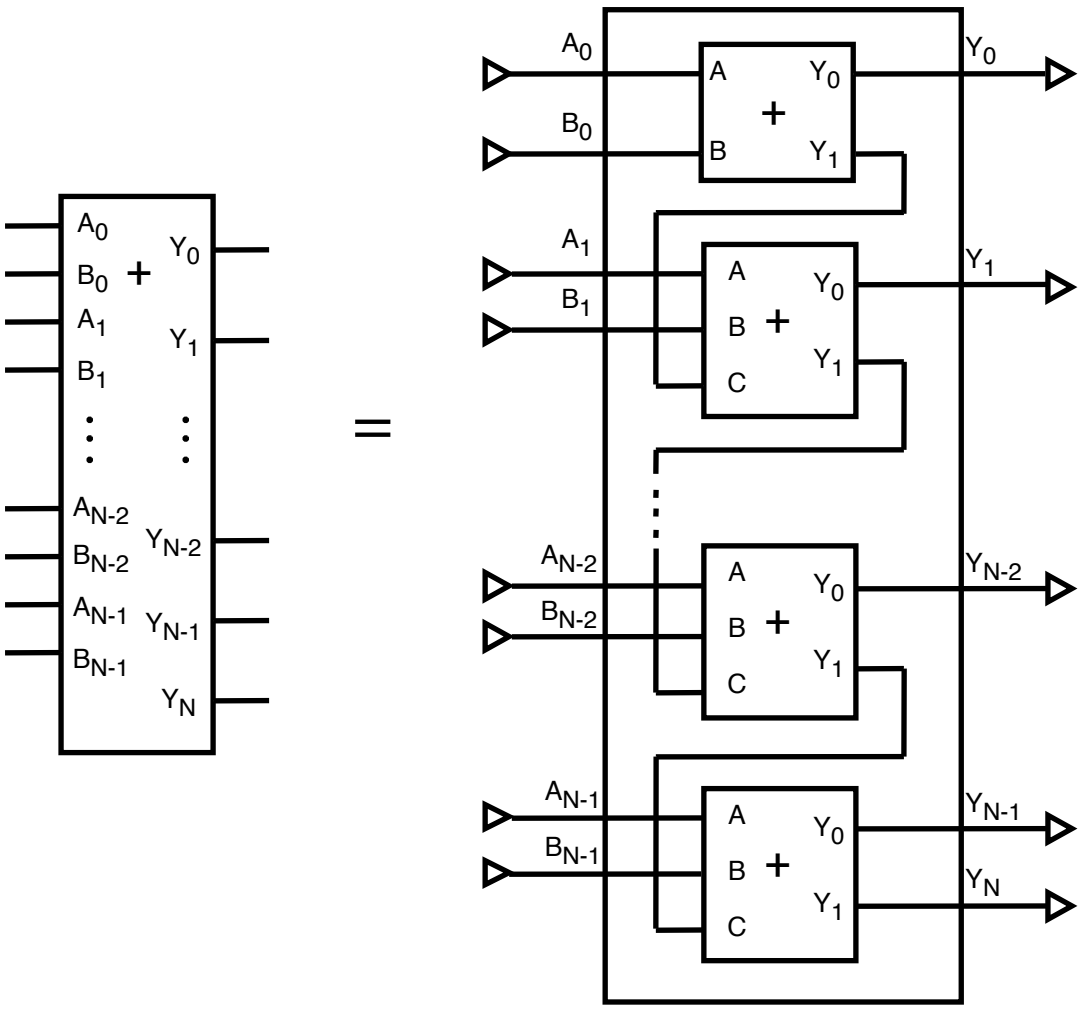

Fig. 3.1.4　「２入力 N ビット足し算回路」（リップルキャリー加算器）

Fig. 3.1.4 に示すような加算器は，**リップルキャリー加算器** ripple carry adder（順次桁上げ加算器）と呼ばれます。このように足し算回路の作られる例もあるようですが，この回路の組み合わせ方では，最上位ビットの処理をする全加算器が，下位の加算器からの繰り上がり（キャリー）信号を受け取るまで，少し長めの時間待たされることになります。実際には，繰り 上がりの部分を並列に処理することで演算を高速化する方法がとられます（補足 3.1.A）。

3－2　論理回路による引き算
Subtraction with logical circuits

引き算の結果は負（マイナス）になることもあります（補足 3.2.A）。一般的なディジタルコンピュータでは，負の整数を表現するために「補数表現」という方法を使います（補足 3.2.B）。たとえば 8 ビットの２進数で整数を表現する際に最上位ビットが「0」のときは $(0000\,0000)_b = (0)_d$，$(0000\,0001)_b = (1)_d$，$(0000\,0010)_b = (2)_d$，…，$(0111\,1111)_b = (127)_d$ のように普通に値を対応させますが，最上位ビットが「1」のとき

は，$(1000\,0000)_b = (-128)_d$，$(1000\,0001)_b = (-127)_d$，…，$(1111\,1110)_b = (-2)_d$，$(1111\,1111)_b = (-1)_d$ のように対応させます。このようにすれば，結果が -128 から 127 の整数の範囲におさまるような足し算は，負の数が含まれていたとしても 3－1 節の「足し算回路」をそのまま使って計算できます。

例えば，「$9-5$」の計算をするときに，「$9+(-5)$」と見なせば「足し算」になります。

$$(9)_d = (0000\,1001)_b$$
$$(-5)_d = (1111\,1011)_b$$

という表現に対して，二進数での加算を行えば

$$\begin{array}{cccccccccc} & & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ +) & & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

のようになります。$(0000\,1001)_b + (1111\,1011)_b = (1\,0000\,0100)_b$ であり，繰り上がり（キャリー）に相当する最上位ビットを無視すれば $(0000\,0100)_b = (4)_d$ となり，「$9-5=4$」に対応する結果が得られます。

補数表現を用いた場合には，有効なビット数によって，その意味する値が変わるということに注意する必要があります。たとえば，「1111 1111」という表現は 8 ビットでは -1 という数を表しますが，16 ビットでは 255 という数を表します。

参考のために，「２入力１ビット引き算回路」について考えます。この回路は

$$0-0=0$$
$$0-1=-1$$
$$1-0=1$$
$$1-1=0$$

という４通りの場合について正しい答えを出力すれば良いのでしょう。十進数での「-1」という数を，２ビットの補数表現で 11 と表すことにします。この２ビットのうちの上位ビットは「繰り下がり」を表すと考えます。この場合，上の例は，

$$0-0=00$$
$$0-1=11$$
$$1-0=01$$
$$1-1=00$$

と書けます。この関係を論理値表にまとめれば，Table 3.2.1 のようになります。

Table 3.2.1　２入力１ビット引き算回路の論理値表。入力 A，B，出力 Y_0 と繰り下がり Y_1

A	B	Y_1	Y_0
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

また，このような出力を実現する回路を Fig. 3.2.2 の記号で表すことにします。

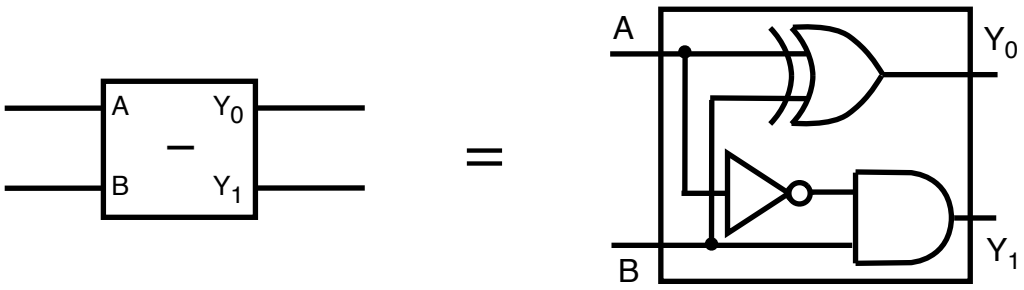

Fig. 3.2.2　「２入力１ビット引き算回路」

さらに，繰り下がりの入力まで考慮して，３つの１ビット入力 A, B, C から $(A - B - C)$ の結果を出力するような３入力１ビット引き算回路を考えることにします（Table 3.2.2）。$0 - 1 - 1 = -2$ ですが，十進数での「-2」という数は，２ビットの補数表現では「10」と表されます。

Table 3.2.2　３入力１ビット引き算回路の論理値表。

2^j の桁の入力 A，B と 2^{j-1} の桁からの繰り下がり C，

2^j の桁からの繰り下がり Y_1 と 2^j の桁の出力 Y_0

A	B	C	Y_1	Y_0
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Fig. 3.2.2 の「２入力１ビット引き算回路」から「３入力１ビット引き算回路」を Fig. 3.2.3 のように組み立てることができます。

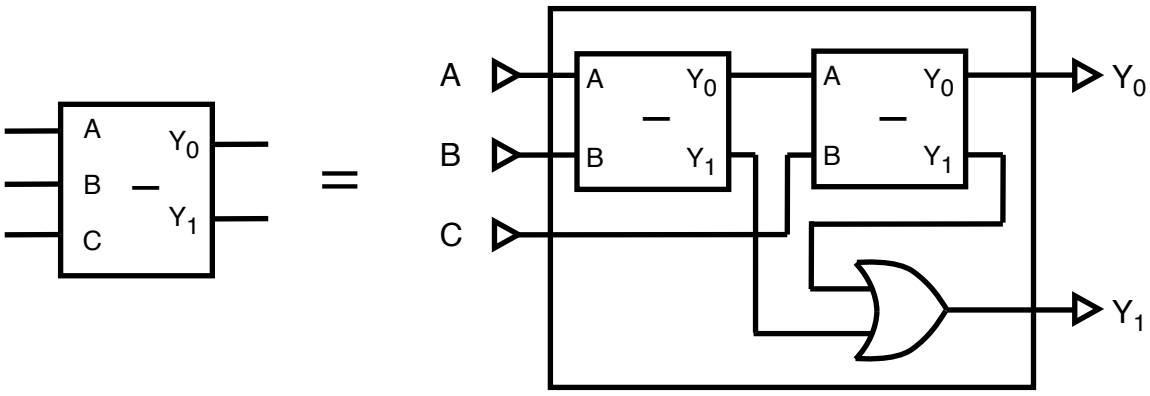

Fig. 3.2.3　「３入力１ビット引き算回路」

この「３入力１ビット足し算回路を」を必要な数だけ集めれば，任意のビット数の算術的な差を求める回路が作ることができるはずです（Fig. 3.2.4）。

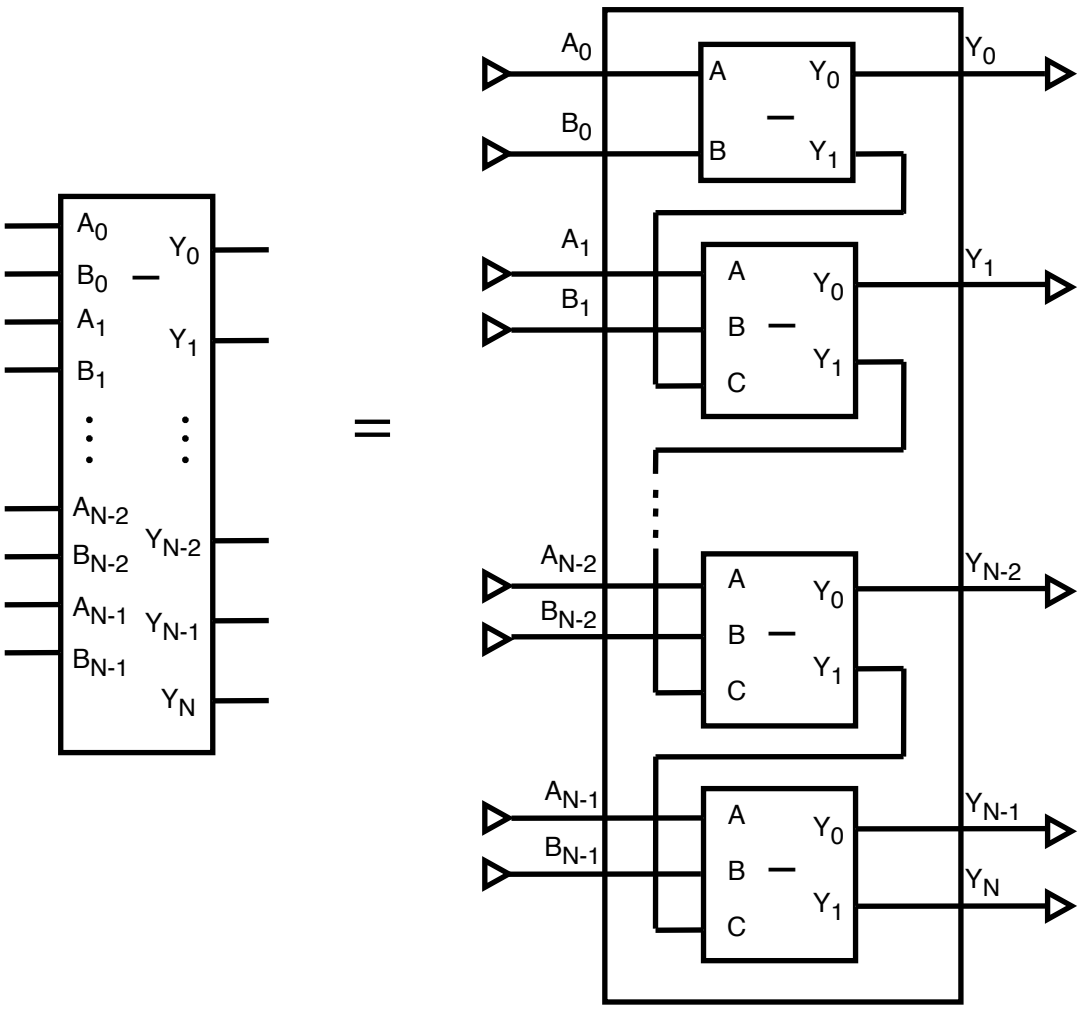

Fig. 3.2.4　「２入力 N ビット引き算回路」

３−３　論理回路によるかけ算，割り算
Multiplication & Division with logical circuits

日本の初等教育（小学校での教育）では，九九（くく）を使ってかけ算を計算する方法を学びます。これは十進法で数を表すからで，数を二進法で表すことにすれば $0\times0=0$, $0\times1=0$，$1\times0=0$，$1\times1=1$ の４つの関係だけ憶えれば良いことになります。後は「桁をずらす」ことと「和をとる」ということができればよいので，ディジタル電子回路を使って，二進法で表される数値のかけ算をすることは，難しいことではありません。

「九九を憶えなければ，かけ算ができない」と思う人もいるかもしれませんが，九九を知らなくてもかけ算はできます。「**ロシア農民のかけ算**」Russian peasant multiplication として知られている方法が有名です。実際には，この方法は古代エジプトでも使われていたらしく「**古代エジプトのかけ算**」ancient Egyptian multiplication とも呼ばれます。

「ロシア農民のかけ算」と呼ばれる方法を使って，たとえば，19×25 という計算をしてみます。はじめに 19 と 25 を横に並べて書いて，次の行には19 を 2 で割った数 9（あまり

は無視する；以下同じ）と 25 に 2 をかけた数（あるいは 25+25）50 を並べて書きます。その次の行には 9 を 2 で割った数 4 と 50 に 2 をかけた数 100 を書きます。どんどん繰り返していって，左側の数が 1 になるまで繰り返します。最後に，左側の数が奇数となっている行の右側の数を足してやればかけ算の答えになる，という方法です。

１９	○	２５
９	○	５０
４		１００
２		２００
１	○	４００

２５＋５０＋４００＝４７５

ここで，19 を 2 で割っていって奇数かどうかを調べる操作は，二進法表記を下位ビットから順に求めていくことと同じです。つまり，十進法での 19 は二進法では $(1\,0011)_b$ となります。また，このことは，$19 = 2^4 + 2^1 + 2^0$ と同じ意味です。ロシア農民のかけ算では，下の式で表されるような方法で計算をしていると考えれば良いでしょう。

$$
\begin{aligned}
19 \times 25 &= (2^4 + 2^1 + 2^0) \times 25 \\
&= 25\times2^4 + 25\times2^1 + 25\times2^0 \\
&= 25\times2\times2\times2\times2 + 25\times2 + 25 \\
&= 400 + 50 + 25
\end{aligned}
$$

二進法のかけ算には，九九は必要ではありません。たとえば 19×25 の計算は，

$(19)_d = (1\,0011)_b$，$(25)_d = (1\,1011)_b$ から，

					1	0	0	1	1
×)					1	1	0	0	1
					1	0	0	1	1
		1	0	0	1	1			
	1	0	0	1	1				
	1	1	1	0	1	1	0	1	1

となり，$(1\,1101\,1011)_b = 2^8 + 2^7 + 2^6 + 2^3 + 2^1 + 2^0 = 256 + 128 + 64 + 16 + 8 + 2 + 1 = 475$ のようになります。

ただし，かけ算（乗算）の処理を早くするために，九九と同じように，複数ビットを同時に処理する方法が実際には用いられます。かけ算をハードウェアで実現するものを乗算器 multiplier と呼び，いろいろなバリエーションがあります。**ウォレス・ツリー** Wallace tree（ウォレスの木）と**ブースの乗算アルゴリズム** Booth's multiplication algorithm を使う方法などが知られています。

二進数での整数同士のわり算（除算）も特別に難しいことではなく，例えば $477 \div 19$ の計算をするとき，$(477)_d = (1\,1101\,1101)_b$ と $(19)_d = (1\,0011)_b$ とから，

										1	1	0	0	1
1	0	0	1	1	)	1	1	1	0	1	1	1	0	1
						1	0	0	1	1				
							1	0	1	0	1			
							1	0	0	1	1			
										1	0	1	0	1
										1	0	0	1	1
													1	0

のように計算すれば，商は $(1\,1001)_b = (25)_d$，余りは $(10)_b = (2)_d$ と求まります。ただし，実際に割り算の計算が必要な場合には，ここまで述べた**整数** integer でなく，多くの場合に後述する**浮動小数点数** floating-point number を使うことが前提となるでしょう。

3－4　数の表現　Expressions of numbers

「数」には整数，有理数，実数，複素数などの種類があり，扱う数の種類に応じてコンピュータ内部での表現のしかたも変わります。

コンピュータで扱える数の代表的なものに，（一定の範囲の）**整数** integer と**浮動小数点数** floating-point number とがあります。普通に利用できるコンピュータの処理系では，扱える整数の絶対値の大きさに制限があります。プロセッサが 8 bit あるいは 16 bit, 32 bit の処理能力しか持っていなかった時期の影響はまだ残っていますが，パーソナルコンピュータでは，2005 年頃（Microsoft Windows XP Professional x64 Edition の頃）に 64 bit プロセッサの搭載が普通になりました。スマートフォンでは，2013 年に発売された Apple iPhone 5S から 64 bitプロセッサが搭載されるようになったと言われます。プロセッサの処理能力の向上につれて，音声や画像，動画を処理する能力も高くなっています（補足 3.4.A）。

8 bit では 0 ～ 255（あるいは −128 ～ 127）の整数，16bit では 0 ～ 65 535，32 bit では 0 ～ 4 294 967 2950 の整数（約 43 億；4.3×10^9）までを扱うことができます。64 bit では 0 ～ 18 446 774 073 709 551 6150（約 1845 京；1.845×10^{19}）までの整数を扱うことができます（補足 3.4.B）。

浮動小数点数 floating-point number とは，十進法では「$9.192\ 631\ 770 \times 10^9$」

「$2.997\ 924\ 58 \times 10^8$」「$6.626\ 070\ 15 \times 10^{-34}$」「$1.602\ 176\ 634 \times 10^{-19}$」

「$1.380\ 649 \times 10^{-23}$」などのように「$s \times b^e$」のように表記される数を意味します。このような「$s \times b^e$」の表現は「**指数表記** exponential notation」と呼ばれます。また「$s \times b^e$」と表現するとき，s は**仮数** significand，b は**底** base または**基数** radix，e は**指数** exponent と呼ばれます。

二進法での浮動小数点数は基数を $b = 2$ とする指数表記によるもので，その表現のしかたについては 1985 年に制定された **IEEE 754**（アイトリプルイー・なな・ごー・よん）規格に定められています（補足 3.4.C）。普通に利用できるほとんどの計算システムで，浮動小数点数はこの規格に従って扱われています。当初は主に 32 bit または 64 bit で一つの浮動小数点数が表され，それぞれ**単精度浮動小数点数** single-precision floating-point number，**倍精度浮動小数点数** double-precision floating-point number と呼ばれました。

倍精度浮動小数点数では，正負の符号に 1 bit，仮数部に 52 bit，指数部に 11 bit が割り当てられます。仮数部の 52 bit は，十進法での有効数字約 16 桁に相当します。絶対値の大きい数としては 1.798×10^{308} 程度までの数を扱うことができます（補足 3.4.D）。

IEEE 754 規格は 2008 年に改訂が行われ，この内容が IEEE 754-2008 規格，それ以前の内容が IEEE 754-1985 規格と呼ばれる場合もあります。IEEE 754-2008 規格では， 128 bit で一つの浮動小数点数を表す**四倍精度浮動小数点数** quadruple-precision floating-point number が定められました。四倍精度浮動小数点数では，正負の符号に 1 bit，仮数部に 112 bit，指数部に 15 bit が割り当てられ，十進法での有効数字はおよそ 34 桁に相当します（補足 3.4.E）。しかし，現時点でこの規格に従う計算システムを利用できるのは，まだ特殊な環境に限られます（補足 3.4.F）。

3－5　この章の最後に

コンピュータで扱える浮動小数点数 f は一般式として

$$f = \frac{I}{2^n}$$

$I \in$ integer （整数）

$n \in$ integer （整数）

と表されます。浮動小数点数として表せるのは，分母が 2の整数乗として表せる分数だけです。また，整数として扱えるビット数には限りがあります。

例えば，$\frac{1}{3}$ や $\frac{1}{10}$ のような有理数を，浮動小数点数では正確に表すことはできません。

高校までに学習する数の概念（自然数，整数，有理数，実数）と，コンピュータで普通に扱える「64 bit 浮動小数点数」「32 bit 整数」「64 bit 整数」との関係をベン図 Venn diagram にすれば， Fig. 3.5.1 のようになります。

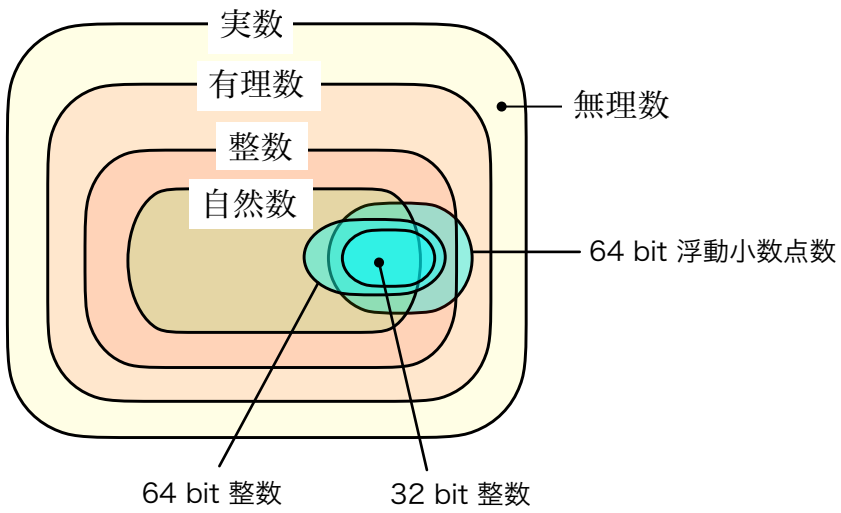

Fig. 3.5.1　64 bit 浮動小数点数と 64 bit 整数，32 bit 整数，64 bit 整数，自然数，整数，有理数，実数との関係を示すベン図

補足

（補足 3.1.A）足し算の繰り上がり（キャリー）の並列演算（↩）

足し算の繰り上がり（桁上げ; carry）の並列演算をする仕組みについては，多くの方法が提案され，そのような仕組みが**並列接頭辞加算器** parallel prefix adder と呼ばれる場合がある。

２つの整数 A, B が

$$A = (A_{n-1}\cdots A_3 A_2 A_1 A_0)_{\mathrm{b}}$$
$$B = (B_{n-1}\cdots B_3 B_2 B_1 B_0)_{\mathrm{b}}$$

と表されるとする。これらの加算を行う時に，i 桁目から $i+1$ 桁目に繰り上がるキャリー c_{i+1} は

$$c_{i+1} = \big(A_i \wedge B_i\big) \vee \Big[\big(A_i \operatorname{xor} B_i\big) \wedge c_j\Big] \tag{3.1.A.1}$$

のように表される。この「i 桁目から繰り上がるキャリー」 c_{i+1} が，下の桁からのキャリー c_i（キャリー・イン）を伝播 (propagate)するものか，この i 段階目ではじめて作られた (generate) ものかを区別することにして，式 (3.1.A.1) の表現を

$$c_{i+1} = G_i \vee \big(P_i \wedge c_i\big) \tag{3.1.A.2}$$
$$G_i = A_i \wedge B_i \tag{3.1.A.3}$$
$$P_i = A_i \operatorname{xor} B_i \tag{3.1.A.4}$$

と書き直す。ここで G_i は i 桁目でのキャリーの**生成** generation を意味し，P_i は**伝播** propagation を意味する。

例えば 4 bit どうしの加算の場合について考える。$c_0 = c_0$ として，上位の桁の繰り上がりは，式 (3.1.A.2) の表現を使えば，

$$c_0 = c_0 \tag{3.1.A.5}$$
$$c_1 = G_0 \vee (c_0 \wedge P_0) \tag{3.1.A.6}$$
$$\begin{aligned} c_2 &= G_1 \vee (c_1 \wedge P_1) = G_1 \vee \Big\{\big[G_0 \vee (c_0 \wedge P_0)\big] \wedge P_1\Big\} \\ &= G_1 \vee \big(G_0 \wedge P_1\big) \vee \big(c_0 \wedge P_0 \wedge P_1\big) \end{aligned} \tag{3.1.A.7}$$
$$\begin{aligned} c_3 &= G_2 \vee (c_2 \wedge P_2) = G_2 \vee \Big\{\big[G_1 \vee (c_1 \wedge P_1)\big] \wedge P_2\Big\} = G_2 \vee \big(G_1 \wedge P_2\big) \vee \big(c_1 \wedge P_1 \wedge P_2\big) \\ &= G_2 \vee \big(G_1 \wedge P_2\big) \vee \Big\{\big[G_0 \vee (c_0 \wedge P_0)\big] \wedge P_1 \wedge P_2\Big\} \\ &= G_2 \vee \big(G_1 \wedge P_2\big) \vee \big(G_0 \wedge P_1 \wedge P_2\big) \vee \big(c_0 \wedge P_0 \wedge P_1 \wedge P_2\big) \end{aligned} \tag{3.1.A.8}$$
$$\begin{aligned} c_4 &= G_3 \vee (c_3 \wedge P_3) = G_3 \vee \Big\{\big[G_2 \vee (c_2 \wedge P_2)\big] \wedge P_3\Big\} = G_3 \vee \big(G_2 \wedge P_3\big) \vee \big(c_2 \wedge P_2 \wedge P_3\big) \\ &= G_3 \vee \big(G_2 \wedge P_3\big) \vee \Big\{\big[G_1 \vee (c_1 \wedge P_1)\big] \wedge P_2 \wedge P_3\Big\} \end{aligned}$$

$$
\begin{aligned}
&= G_3 \vee \left(G_2 \wedge P_3\right) \vee \left(G_1 \wedge P_2 \wedge P_3\right) \vee \left(c_1 \wedge P_1 \wedge P_2 \wedge P_3\right) \\
&= G_3 \vee \left(G_2 \wedge P_3\right) \vee \left(G_1 \wedge P_2 \wedge P_3\right) \vee \left\{ \left[G_0 \vee (c_0 \wedge P_0)\right] \wedge P_1 \wedge P_2 \wedge P_3 \right\} \\
&= G_3 \vee \left(G_2 \wedge P_3\right) \vee \left(G_1 \wedge P_2 \wedge P_3\right) \vee \left(G_0 \wedge P_1 \wedge P_2 \wedge P_3\right) \vee \left(c_0 \wedge P_0 \wedge P_1 \wedge P_2 \wedge P_3\right)
\end{aligned}
\tag{3.1.A.9}
$$

と書ける。式 (3.1.A.7)–(3.1.A.9) に含まれる「複数の論理変数の論理積」$\left(A_1 \wedge A_2 \cdots \wedge A_n\right)$ と，「複数の論理変数の論理和」$\left(A_1 \vee A_2 \cdots \vee A_n\right)$ の論理演算は，「多入力の論理積回路」と「多入力の論理和回路」を使って，それぞれ同時に処理することができる。

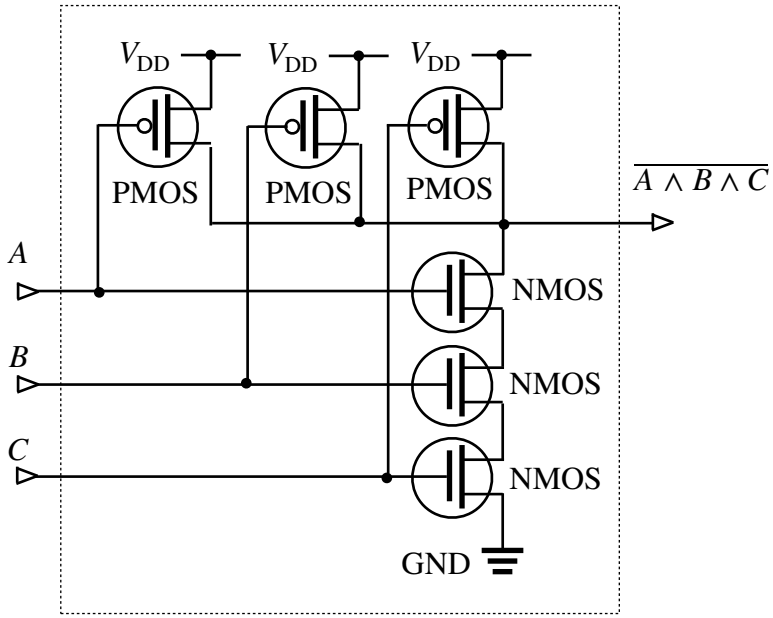

Fig. 3.1.A.1　３入力の NAND 回路

キャリー先読み加算器 carry look-ahead adder （桁上先見加算器）は，このような表現を電子回路で実現するように設計される。３入力の否定論理積 NAND 演算 $\overline{A \wedge B \wedge C}$ は，３つの PMOS を並列に，３つの NMOS を直列に接続すれば作れる（Fig. 3.1.A.1），それに否定 NOT 演算回路を接続すれば $A \wedge B \wedge C$の 演算結果を同時に出力する３入力の論理積演算回路が作れることになる。同じように４入力以上の多入力の論理積回路を作ることができて，このことは論理和回路についても同様である。

このことは，式 (3.1.A.6)–(3.1.A.9) のような論理式で表される各桁ごとのキャリーイン（下の桁からの繰り上がり）を確定させ るために必要となる論理演算処理による信号の遅延は，AND 演算 1 回分と OR 演算 1 回分だけということ を意味する。式 (3.1.A.5)–(3.1.A.9) によってキャリー $c = (c_4 c_3 c_2 c_1 c_0)_\mathrm{b}$ が確定していれば，$A = (A_3 A_2 A_1 A_0)_\mathrm{b}$ と $B = (B_3 B_2 B_1 B_0)_\mathrm{b}$ の加算の結果 $Y = (Y_4 Y_3 Y_2 Y_1 Y_0)_\mathrm{b}$ の各桁は，

$$Y_0 = c_0 \operatorname{xor} P_0 \tag{3.1.A.10}$$

$$Y_1 = c_1 \operatorname{xor} P_1 \tag{3.1.A.11}$$

$$Y_2 = c_2 \operatorname{xor} P_2 \tag{3.1.A.12}$$

$$Y_3 = c_3 \operatorname{xor} P_3 \tag{3.1.A.13}$$

$$Y_4 = c_4 \tag{3.1.A.14}$$

によって得られる。式 (3.1.A.10)–(3.1.A.13) で表される一連の排他的論理和 XOR 演算は，算術的な加算とは違って，繰り上がりが発生しても無視すれば良いだけなので，これも同時に並列に処理できる。

以上のことを回路図に置き換えると，4ビットのキャリー先読み加算器は，図 3.1.A.2 のような回路で表されるであろう。

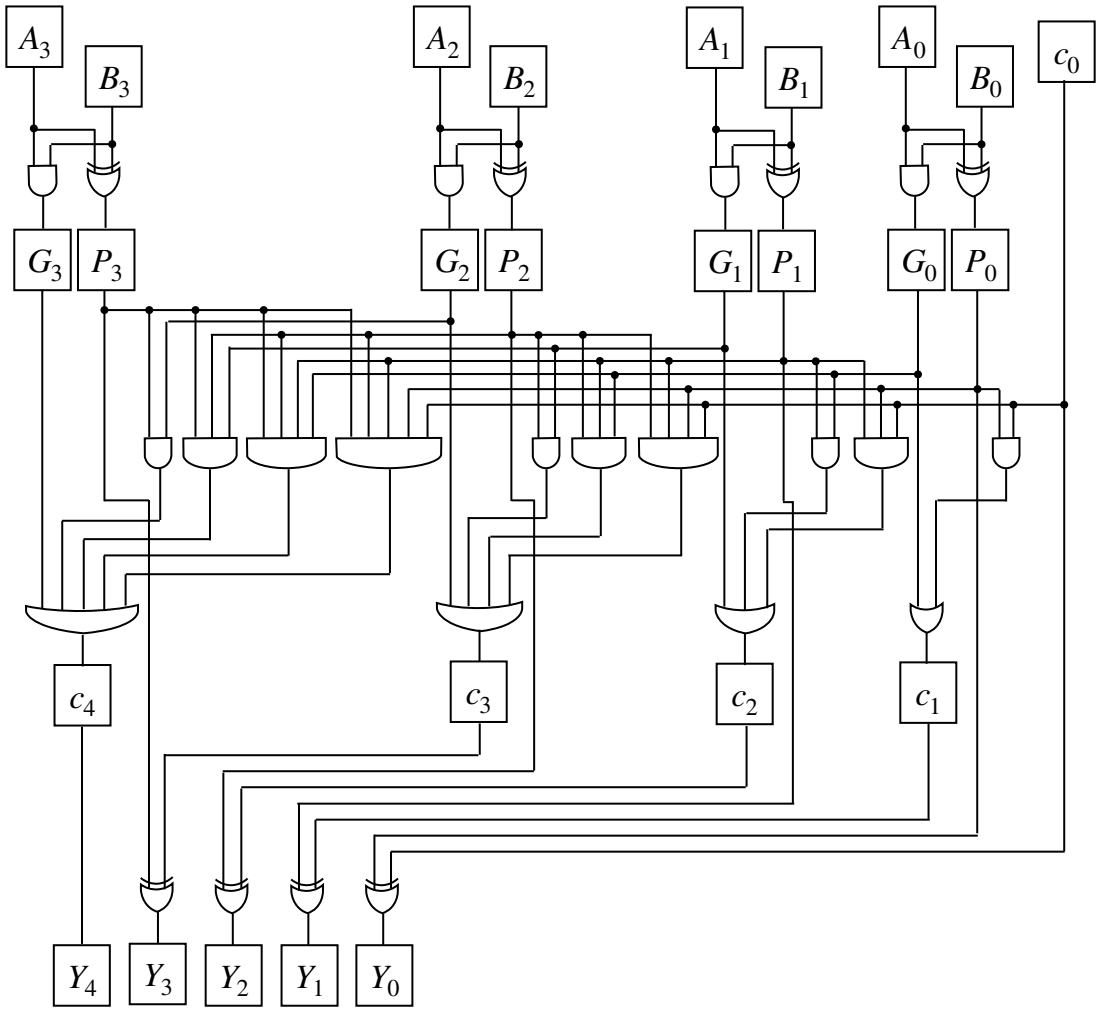

Fig. 3.1.A.2　4 ビットのキャリー先読み加算器の考え方

また，ド・モルガンの法則から

$$(A \wedge B) \vee (C \wedge D) = \overline{\overline{A \wedge B} \vee \overline{C \wedge D}} = \overline{\overline{A \wedge B} \wedge \overline{C \wedge D}} \tag{3.1.A.15}$$

という関係が成立するので，AND 演算と OR 演算との組み合わせは，NAND 演算と NAND 演算との組み合わせに置き換えることができる。

このことから，Fig. 3.1.A.2 に示した回路図を，Fig. 3.1.A.3 のように描きなおせば，使用するトランジスタの数と信号の遅延を少し減らすこと ができる。

NOT 回路が 2 個のトランジスタと 1 段分の信号遅延，N 入力の NAND 回路が $2N+2$ 個のトランジスタと 1 段分の信号遅延，N 入力の AND 回路が $2N+4$ 個のトランジスタと 2 段分の信号遅延，N 入力の OR 回路が $2N+4$ 個のトランジスタと 2 段分の信号遅延，XOR 回路が 6 個のトランジスタと 2 段分の信号遅延を持つとすれば，Fig. 3.1.A.2 の回路では

$(6+6)\times 4+6+8+10+12+6+8+10+6+8+6+12+10+8+6+6\times 4=188$ 個のトランジスタと 8 段分の信号遅延を持つことになるが，Fig. 3.1.A.3 の回路では $(4+6)\times 4+2\times 3+4+6+8+10+4+6+8+4+6+4+10+8+6+4+6\times 4=158$ 個のトランジス タと，6 段分の信号遅延で済むことになる。

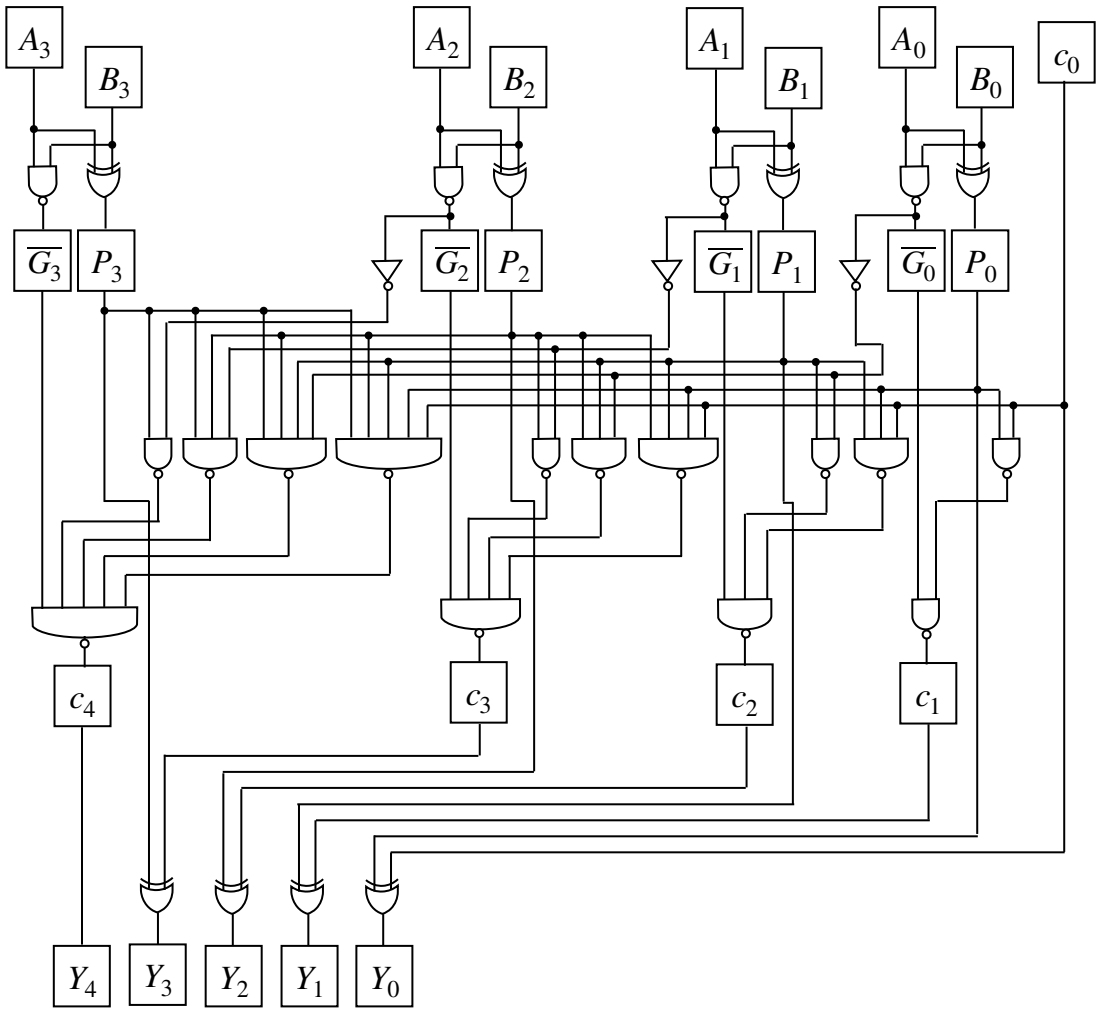

Fig. 3.1.A.3　４ビットのキャリー先読み加算器の実装

なお，半加算器で 12 個のトランジスタと 2 段分の信号遅延，全加算器で $12\times 2+6=30$ 個のトランジスタと 5 段分の信号遅延があるとすれば，4ビットのリップルキャリー加算器では $30\times 3+12=102$ 個のトラ ンジスタと 17 段分の信号遅延が生じることになる。

以下，4ビットのキャリー先読み加算器，Fig. 3.1.A.2 の回路に沿って，具体的な数値 $A=9, B=13$ をあてはめて，動作のしかたを確かめてみよう。

$$A=(9)_{\mathrm{d}}=(1001)_{\mathrm{b}}=(A_3A_2A_1A_0)_{\mathrm{b}}$$

$$B=(13)_{\mathrm{d}}=(1101)_{\mathrm{b}}=(B_3B_2B_1B_0)_{\mathrm{b}}$$

とする。第一段階では，各桁について半加算器による加算をおこなう。その結果は

$$(G_3P_3, G_2P_2, G_1P_1, G_0P_0)=(10,01,00,10)$$

となる。ここでは $1=2^0$ の桁より下位の桁からの繰り上がりは考えなくて良いので，$c_0=0$ とする。

このとき，キャリー先読み加算器を用いれば，式 (3.1.A.5)–(3.1.A.9) の関係から，各桁ごとのキャリーの値は，

$$
\begin{aligned}
c_0 &= c_0 = 0 \\
c_1 &= G_0 \vee (c_0 \wedge P_0) = 1 \vee (1 \wedge 0) = 1 \vee (1 \wedge 0) = 1 \\
c_2 &= G_1 \vee \left(G_0 \wedge P_1\right) \vee \left(c_0 \wedge P_0 \wedge P_1\right) = 0 \vee (1 \wedge 0) \vee (0 \wedge 0 \wedge 0) = 0 \\
c_3 &= G_2 \vee \left(G_1 \wedge P_2\right) \vee \left(G_0 \wedge P_1 \wedge P_2\right) \vee \left(c_0 \wedge P_0 \wedge P_1 \wedge P_2\right) \\
&\quad = 0 \vee (0 \wedge 1) \vee (1 \wedge 0 \wedge 1) \vee (0 \wedge 0 \wedge 0 \wedge 1) = 0 \\
c_4 &= G_3 \vee \left(G_2 \wedge P_3\right) \vee \left(G_1 \wedge P_2 \wedge P_3\right) \vee \left(G_0 \wedge P_1 \wedge P_2 \wedge P_3\right) \vee \left(c_0 \wedge P_0 \wedge P_1 \wedge P_2 \wedge P_3\right) \\
&\quad = 1 \vee \cdots = 1
\end{aligned}
$$

となる。これらのキャリー $\{c_4, c_3, c_2, c_1, c_0\} = \{1,0,0,1,0\}$ の値と伝播 $\{P_3, P_2, P_1, P_0\} = \{0,1,0,0\}$ の値，式 (3.1.A.10)–(3.1.A.14) の関係から，

$$
\begin{aligned}
Y_0 &= c_0 \operatorname{xor} P_0 = 0 \operatorname{xor} 0 = 0 \\
Y_1 &= c_1 \operatorname{xor} P_1 = 1 \operatorname{xor} 0 = 1 \\
Y_2 &= c_2 \operatorname{xor} P_2 = 0 \operatorname{xor} 1 = 1 \\
Y_3 &= c_3 \operatorname{xor} P_3 = 0 \operatorname{xor} 0 = 0 \\
Y_4 &= c_4 = 1
\end{aligned}
$$

が得られる。 $Y = (Y_4 Y_3 Y_2 Y_1 Y_0)_{\mathrm{b}} = (10110)_{\mathrm{b}} = (16)_{\mathrm{d}} + (4)_{\mathrm{d}} + (2)_{\mathrm{d}} = (22)_{\mathrm{d}}$ となることから，$9 + 13 = 22$ の計算を正しくできていることが確かめられる。

キャリー先読み加算器では，ビット数が増えると，**生成** G_i や**伝播**（でんぱ） P_i の出力が，多くの論理演算素子の入力信号として分配されることになる。一つの素子の出力信号に対して，それを入力信号とする他の素子の数を**ファンアウト** fan-out と呼ぶ。ファンアウトの数には限りがあるので，キャリー先読み加算器で扱 えるビット数も限られたものになる。

ファンアウトを抑えるためのキャリー並列計算(並列接頭辞加算)の工夫のしかたは既に詳しく調べられている。その中で，**コーギー・ストーン加算器** Kogge-Stone adder と呼ばれる仕組みが比較的良く用いられるようである。 (↩)

（補足 3.2.A）負の数（↩）

歴史的には，負の数とその加減算の規則（正の数を引くことと，負の数を足すことが同じことであり，負の 数を引くことと正の数を足すことが同じことだということなど）については，古代中国の漢王朝の時代 (202 BC – AD 220)，概ね 1 世紀頃までにまとめられた「**九章算術**」という数学書に記載されている内容が，確認されている中で最も古いという説が有力である。 (↩)

（補足 3.2.B）「２の補数」と「１の補数」（↩）

負の整数を表す補数表現として標準的に用いられる「２の補数」と呼ばれる表現について取り上げているが，それとは別に「１の補数」と呼ばれる表現もある。日本語で「２の 補数」と呼ばれることは，英語で

は「two's complememt」と表記されるのに対して「1の補数」は「ones' complement」と表記される。「two's」と「ones'」とで，アポストロフィ記号「'」の位置が違うことに注意せよ。このことから，これらの用語が，英語圏では異なる文脈で理解されていることがわかる（補足 3.2.B.1）。

「1の補数 ones' complement」という表現は，例えば 8 bit の場合に，「その数を足した時にすべてのビットが 1 で表される数 $(1111\,1111)_b$ になる」という意味があるので，複数形の「ones」が用いられる。また，これはすべてのビットを反転させることと同じことで，例えば十進法の $(12)_d$ は二進法で $(0000\,1100)_b$ と表現されるのに対して，「1の補数」では十進法の $(-12)_d$ を $(1111\,0011)_b$ と表現する。

一方で「2の補数 two's complement」という表現では，「2 を基数 radix（底 base）とした場合に…」という文脈が使われる。8 bit の場合，これは「2^8」という表現の中での「2」の意味（「2^8」という表現の中での 8 は指数 exponent の意味）であり，「『その数』を足した時に $2^8 = (1\,0000\,0000)_b$ と表される数になる」という意味合いがある。例えば十進法の $(12)_d$ は二進法で $(0000\,1100)_b$ と表現されるのに対して，「2の補数」表現では，十進法の $(-12)_d$ を $(1111\,0100)_b$ と表現する。

「2の補数」を用いる場合，正の数値と同じ絶対値の負の数値のビット列表現を求めるのであれば，すべてのビットを反転させた後に「1 を加える」操作をすれば良いと言われる。たとえば $(12)_d = (0000\,1100)_b$ のビットを反転させると「1の補数」表現 $(1111\,0011)_b$ となるが，「2の補数」表現では，$(1111\,0011)_b$ は $(-13)_d$ を表す。「2の補数」では $(-12)_d$ は $(1111\,0100)_b$ と表現されるので，ビットを反転させたあとに 1 を加えれば良い関係にあることが確認できる。逆に，「負の数」を同じ絶対値の「正の数」に置き換えたければ，「1 を引く」操作をしてからビット反転をすれば良い。

「1の補数」ではゼロを表現するために（8 bit の場合）$(0)_d = (0000\,0000)_b$ と $(0)_d = (1111\,1111)_b$ という二通りの表現が存在して少し紛らわしいという問題がある。また，負の数の加算を普通の加算回路で処理したときに，最上位ビットからの繰り上がり（キャリー）が 1 であれば加算の結果にさらに「1 を加える」ことが必要になる。

「1の補数」での二進数表現を $(\cdots)_{b1}$，「2の補数」での二進数表現を $(\cdots)_{b2}$ と表すことにする。例えば，

$$(12)_d = (0000\,1100)_{b1} = (0000\,1100)_{b2}$$

に対して，8ビットでの「1の補数表現」と「2の補数表現」として

$$(-3)_d = (1111\,1100)_{b1} = (1111\,1101)_{b2}$$
$$(-15)_d = (1111\,0000)_{b1} = (1111\,0001)_{b2}$$

を加える計算を試みる。

「2の補数」では，最上位ビットの繰り上がり(キャリー)まで含めて二進数での加算をすれば

$$(12)_d+(-3)_d = (0000\,1100)_{b2}+(1111\,1101)_{b2} = (1\,0000\,1001)_b$$
$$(12)_d+(-15)_d = (0000\,1100)_{b2}+(1111\,0001)_{b2} = (0\,1111\,1101)_b$$

となり，キャリーを無視すれば

$$(0000\,1001)_b = (9)_d$$
$$(1111\,1101)_b = (-3)_d$$

となって，$(12)_d+(-3)_d = (9)_d$ と $(12)_d+(-15)_d = (-3)_d$ が計算できていることがわかります。

「1の補数」の場合，同じように最上位ビットの繰り上がり(キャリー)まで含めて二進数での加算をすれば

$(12)_{d}+(-3)_{d} = (0000\,1100)_{b1}+(1111\,1100)_{b1} = (1\,0000\,1000)_{b}$

$(12)_{d}+(-15)_{d} = (0000\,1100)_{b1}+(1111\,0000)_{b1} = (0\,1111\,1100)_{b}$

となり，

$(9)_{d} = (0000\,1001)_{b1}$

$(-3)_{d} = (1111\,1100)_{b1}$

と比較すれば，「加算の結果，最上位ビットでのキャリーがあったときには，そのキャリーを捨てて最下位ビットに1を加算する」という操作をすることにすれば，「1 の補数」を使うのであっても，負数(ふすう)を含めた整数の加算(かさん)・減算(げんざん)で，特別に深刻なパフォーマンスの低下が生じるわけではない。 (↩)

（補足 3.2.B.1）Ones' complement と two's complement（↩）

中国語では two's complement は补码と翻訳され，ones' complement は人的补语と翻訳される。「补」は日本語では「お化粧」のような意味で，「码」は日本語では「数や番号を表す記号」，「语」は「言語」の意味合いだが，「补码」という語は日本語の「補数」と同じ意味で使われ，特に「足すとゼロになる数」という意味合いで使われるようである。 (↩)

（補足 3.4.A）音声と画像，動画に使われるビット数と転送速度（↩）

音声，音楽

CD-audio 16 bit, 44.1 kbps，2ch;74分

DVD-audio, Hi-Res audio 24 bit, 192 kbps;4.7GB

静止画

Facebook プロフィール画像 160 × 160 ~ 2.6 万画素，1 Byte/色 ⇒ 約 8 kB

Instagram 1080 × 1080 ~ 100万画素，1 Byte/色 ⇒ 約 3 MB

iPhone 7s+ (5.5 inch) 1080 × 1920 ~ 200 万画素，1 Byte/色 ⇒ 約 6 MB

iPhone 13 (6.1 inch) 2532 × 1170 ~ 300 万画素，1 Byte/色 ⇒ 約 9 MB

iPad Pro 9.7 inch 2048×1536 ~ 300万画素，1 Byte/色 ⇒ 約 9 MB

写真，印刷

4800×1200dpi = ピクセルサイズ横 0.005 mm，縦 0.02 mm，

4色 1bit/色，L判 (127 mm × 89 mm)，ピクセル数 約 1億，⇒ 約 5 MB

動画，アニメーション

デジタルビデオ 640 × 480 1Byte/色 フレームレート 15 fps ⇒ 約 110 Mbps

HDTV(ハイビジョン) 1920 × 1080 フレームレート 30 fps ⇒ 約 1.5 Gbps

4K テレビ 3840 × 2160 フレームレート 30 fps ⇒ 約 6 Gbps

8K テレビ 7680 × 4320 10,12 bit/色 フレームレート 24–120 fps

(↩)

（補足 3.4.B）64 bit の整数（↩）

Python 言語で利用できる数値計算ライブラリである NumPy の現時点のバージョンでは，データ型 (dtype) として uint64 (unsigned integer 64 bit) とすれば符号のつかない 64 bit 整数を明示的に扱うことができて， int64 (integer 64 bit) とすれ ば「２の補数」表現の符号付き 64 bit 整数を扱える。 (↩)

（補足 3.4.C）IEEE（アイトリプルイー） (↩)

IEEE は米国に本拠地を置く Institute of Electrical and Electronics Engineers で，日本語に翻訳すれば「電気電子工学研究所」に近いのだが，日本でも IEEE，アイトリプルイーと呼ばれるのが普通である。現時点で世界最大の学術研究団体と言われている。 (↩)

（補足 3.4.D）倍精度 (64 bit) 浮動小数点数（↩）

IEEE 754 規格の**倍精度浮動小数点数**では，指数部の 11 ビットが $\{e_{10}\, e_9 \cdots e_0\} \neq \{0\, 0 \cdots 0\}$ かつ $\{e_{10}\, e_9 \cdots e_0\} \neq \{1\, 1 \cdots 1\}$ の場合に，指数部の数値 e が

$$e = 2^{10} e_{10} + 2^9 e_9 + \cdots + 2 e_1 + e_0 - 1023$$

と表されるとし，仮数部の 52 ビット $\{f_{51}, f_{50}, \cdots, f_0\}$ は，二進法の小数表記で $(1\,.\, f_{51}\, f_{50} \cdots f_0)_\mathrm{b}$ を表すとする。符号 s の場合に，全体として

$$(-1)^s \times \left(1 + \frac{f_{51}}{2} + \frac{f_{50}}{2^2} + \cdots + \frac{f_0}{2^{52}}\right) \times 2^{\left(2^{10} e_{10} + 2^9 e_9 + \cdots + 2 e_1 + e_0 - 1023\right)}$$

を表すことになる。このように表される数は**正規数**または**正規化数** normal number と呼ばれる。 指数部の最大値は，$\{e_{10}\, e_9 \cdots e_0\} \neq \{1\, 1 \cdots 1\}$ であることから $(e_{10}\, e_9 \cdots e_1\, e_0)_\mathrm{b} = (1\, 1 \cdots 1\, 0)_\mathrm{b} = (2\,046)_\mathrm{d}$ であり，正規化数の絶対値の最大は

$$\left(1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{52}}\right) \times 2^{2046-1023} = \frac{1 - \dfrac{1}{2^{53}}}{1 - \dfrac{1}{2}} \approx 2^{1024} \approx 1.798 \times 10^{308}$$

となります。また，指数部の最小値は，$\{e_{10}\, e_9 \cdots e_0\} \neq \{0\, 0 \cdots 0\}$ であることを前提とすれば $(e_{10}\, e_9 \cdots e_1\, e_0)_\mathrm{b} = (0\, 0 \cdots 0\, 1)_\mathrm{b} = (1)_\mathrm{d}$ であり，正規化数の最小の絶対値は

$$\left(1 + \frac{0}{2} + \frac{0}{2^2} + \cdots + \frac{0}{2^{52}}\right) \times 2^{1-1023} = 2^{-1022} \approx 2.225 \times 10^{-308}$$

となる。正規化数の十進数での有効数字の桁数は，最低で

$$\frac{52}{\log_2 10} \approx 15.65$$

最高で

$$\frac{53}{\log_2 10} \approx 15.95$$

となる。このことから，倍精度浮動小数点数を用いる場合に，十進数で正確な数値を指定する必要があ れば（例えば円周率など），16 桁で表現すべきと言われる。

IEEE 754 規格の倍精度浮動小数点数では，指数部の 11 ビットが，$\{e_{10}\,e_9\,\cdots\,e_0\} = \{0\,0\,\cdots\,0\}$ の場合には，指数 部を $e = -1022$ として，仮数部の 52 ビット $\{f_{51}, f_{50}, \cdots, f_0\}$ は，二進法の小数表記で $(0\,.\,f_{51}\,f_{50}\,\cdots\,f_0)_\mathrm{b}$ を表すと する。符号 s の場合に

$$(-1)^s \times \left(\frac{f_{51}}{2} + \frac{f_{50}}{2^2} + \cdots + \frac{f_0}{2^{52}}\right) \times 2^{-1022}$$

を表すことになる。このように表される数は，日本では**非正規数**または**非正規化数** subnormal number と呼ばれる（補足 3.4.D.1）。非正規数のゼロ以外の最小の絶対値は

$$\left(\frac{1}{2^{52}}\right) \times 2^{-1022} = 2^{-1074} \approx 5 \times 10^{-324}$$

となる。このとき非正規数の十進数での有効数字の桁数は，最低となり

$$\frac{1}{\log_2 10} \approx 0.3$$

となる。

IEEE 754 規格の倍精度浮動小数点数では，指数部の 11 ビットが $\{e_{10}\,e_9\,\cdots\,e_0\} = \{1\,1\,\cdots\,1\}$ の場合には，仮数部 の 52 ビットが $\{f_{51}, f_{50}, \cdots, f_0\} = \{0,0,\cdots,0\}$ のときには符号部 s の値に応じて正負の**無限大** infinity, $\pm\infty$を表し， $\{f_{51}, f_{50}, \cdots, f_0\} \neq \{0,0,\cdots,0\}$ の場合には**非数** NAN (not a number) を表すとされる。 (↩)

（補足 3.4.D.1）IEEE 754 での subnormal number（↩）

IEEE 754 での subnormal number は，normal number から不連続に変化するわけではなく，この領域では絶対値が小さくなるほど，少しずつ有効数字が少なくなるだけなので，「非正規数」と呼ぶより「準正規数」などと呼ぶ方が本来はふさわしいと思われる。中国では「次正規数」の表記が用いられる。 (↩)

（補足 3.4.E）四倍精度 (128 bit) 浮動小数点数（↩）

IEEE 754-2008 規格の**四倍精度浮動小数点数**では，指数部の 15 ビットが，$\{e_{14}\,e_{13}\,\cdots\,e_0\} \neq \{0\,0\,\cdots\,0\}$ かつ $\{e_{14}\,e_{13}\,\cdots\,e_0\} \neq \{1\,1\,\cdots\,1\}$ の場合に，指数部の数値 e が

$$e = 2^{14}e_{14} + 2^{13}e_{13} + \cdots + 2e_1 + e_0 - 16\,383$$

と表されるとし，仮数部の 112 ビット $\{f_{111}, f_{110}, \cdots, f_0\}$ は，二進法の小数表記で $(1\,.\,f_{111}\,f_{110}\,\cdots\,f_0)_\mathrm{b}$ を表すとする。符号 s の場合に，全体として

$$(-1)^s \times \left(1 + \frac{f_{111}}{2} + \frac{f_{110}}{2^2} + \cdots + \frac{f_0}{2^{112}}\right) \times 2^{\left(2^{14}e_{14}+2^{13}e_{13}+\cdots+2e_1+e_0-16\,383\right)}$$

を表すことになる。このように表される数は**正規数**または**正規化数** normal number と呼ばれる。 指数部の最大値は，$\{e_{14}\,e_{13}\,\cdots\,e_0\} \neq \{1\,1\,\cdots\,1\}$ であることから $(e_{14}\,e_{13}\,\cdots\,e_1\,e_0)_\mathrm{b} = (1\,1\,\cdots\,1\,0)_\mathrm{b} = (32\,766)_\mathrm{d}$ であり，正規数の絶対値の最大は

$$\left(1+\frac{1}{2}+\frac{1}{2^2}+\cdots+\frac{1}{2^{112}}\right)\times 2^{32\,766-16\,383}=\frac{1-\dfrac{1}{2^{113}}}{1-\dfrac{1}{2}}\times 2^{16\,383}\approx 2^{16\,384}$$

となる。倍精度浮動小数点数ではこの $2^{16\,384}$ という数値を扱うことは不可能だが，倍精度浮動小数点数しか使えない環境でも，$2^{16\,384}$ の近似的な十進法表現を求めることはできる。

$$\begin{cases} x\times 10^n = 2^{16\,384} \\ 1\le x<10 \end{cases} \Rightarrow \begin{cases} \log_{10} x + n = 16\,384\log_{10}2 \\ 0\le \log_{10} x<1 \end{cases}$$

と式の変形をして，$16\,384\log_{10}2\approx 4932.08$ であることから，$n=4932$, $\log_{10}x\approx 0.08$ となり，

$$2^{16\,384}\approx 10^{0.08}\times 10^{4932}\approx 1.2\times 10^{4932}$$

と見積もられる。

また，指数部の最小値は，$\{e_{14}\,e_{13}\,\cdots\,e_0\}\neq\{0\,0\,\cdots\,0\}$ であることを前提とすれば $(e_{14}\,e_{13}\,\cdots\,e_1\,e_0)_\mathrm{b}$ $=(0\,0\,\cdots\,0\,1)_\mathrm{b}=(1)_\mathrm{d}$ であり，四倍精度浮動小数点数の正規数の最小の絶対値は

$$\left(1+\frac{0}{2}+\frac{0}{2^2}+\cdots+\frac{0}{2^{112}}\right)\times 2^{1-16\,383}=2^{-16\,382}=10^{-4932+0.53}\approx 3.4\times 10^{-4932}$$

と見積もられる。四倍精度浮動小数点数の正規数の十進数での有効数字の桁数は，最低で

$$\frac{112}{\log_2 10}\approx 33.72$$

最高で

$$\frac{113}{\log_2 10}\approx 34.02$$

となる。このことから，四倍精度浮動小数点数を用いる場合に，十進数で正確な数値を指定する必要があれば（例えば円周率など），35 桁で表現すべきと言うことになる（補足 3.4.E.1）。 (↩)

（補足 3.4.E.1）浮動小数点数での算術演算（↩）

加算 addition は可換 $(A+B=B+A)$ だが，減算 subtraction は非可換 $(A-B\neq B-A)$ である。「引かれる方の数」は被減数（ミニュエンド） minuend，「引く方の数」は減数（サブトラヘンド） subtrahend と呼ばれる。 浮動小数点数では，減算 とは，減数の符号のみを反転させて加算をおこなうことと同じことである。

浮動小数点数（指数表現）での加算 addition では，本来可換のはずの加算に，非対称な構造が現れる。足し合わせ る二つの数のうち，絶対値の大きい方の数を被加数（アージェンド） augend，絶対値の小さい方の数を加数（アドゥンド） addend と呼ぶ。加算とは「被加数に加数を足す（加える）」あるいは「被加数を加数の分増やす」操作と考える。浮動小数点数どうしでの加算では，和（加算の結果）の符号 (±) は，加数の符号にはよらず被加数の符号と一致する。浮動小数点数での加算では，加数の桁を被加数とそろえるために，被加数の指数 exponent と一致するまで加数 addend の指数 exponent を増やして，加数 addend の仮数 significand をその分下位側にシフトする。このシフト操作 のとき，加数の仮数部の上位桁は 0 で埋められ，加数の仮数 significand が持っていた情報のうち，シフト後 の仮数部の最下位桁より低い桁の部分が含んでいた情報は失われることになる。一方で被加数 augend の仮数 significand の含んでいた情報はこの時点では維持されたままである。

被加数と加数の符号が等しい場合には，被仮数と加数の仮数 significand の加算を行い，最上位桁で繰り上がり carry が生じた場合には，キャリーも含めて和の仮数 significand を 1 ビット下位側にシフトし，指数を 1 増やせば正規数 normal number になる。このとき，被加数と加数の含んでいた情報のうち，仮数部の 最下位ビットの含んでいた情報のみが失われる。

被加数と加数の符号が異なる場合には，加数 addend の仮数 significand の「２の補数」表現を求めてから加算を行えば良い。ただし，被加数と加数の符号が異なる場合の加算には，近い数を引くと有効数字の減る「**桁落ち** loss of significance」と呼ばれる現象が現れる場合がある。 (↩)

（補足 3.4.F） 128 bit の浮動小数点数（四倍精度浮動小数点数）の利用（↩）

2022 年 1 月の時点で， Python 言語で利用できる数値計算ライブラリである NumPy で，データ型 (dtype)として float128 型という名称が存在しているが，128 bit 浮動小数点数の演算がサポートされているわけではない。 (↩)