Zeckendorf Integer Arithmetic

Peter Fenwick
Department of Computer Science, The University of Auckland,
Private Bag 92019, Auckland, New Zealand
p.fenwick@auckland.ac.nz

1 Introduction

There are many ways of representing numbers, with a comprehensive account given by Fraenkel[3]. The more familiar representations, and the only ones relevant to the present discussion, represent an integer N as the scalar product

$$N = \mathbf{D} \cdot \mathbf{W}$$

where $\mathbf{D}$ is the *digit vector* (the visible digits of the representation) and $\mathbf{W}$ is a *weight vector*. To conform with normal conventions for displaying number representations these vectors are written in the order

$$\ldots w_i, w_{i-1}, \ldots, w_2, w_1, w_0$$

The weight vector is in turn derived from the *base vector* $\mathbf{B}$ by

$$w_k = \prod_{i=0}^{k-1} b_i$$

In the conventional *uniform base* number systems such as binary or decimal, $b_i = b \quad \forall\ i$ and $w_k = b^k$, where the constant b is the base of the number system. So a number such as $40 = 1 \times 2^5 + 1 \times 2^3$ has the binary (base 2) representation 101000. For measurements in a mixed base system, the base vector has an appropriate mixture of values. As an example for {miles, yards, feet, inches}, the base vector is $\mathbf{B} = \{1760, 3, 12, 1\}$ and the weight vector is $\mathbf{W} = \{63360, 36, 12, 1\}$ to give lengths in inches.

But Zeckendorf has shown [8] that the Fibonacci numbers

$$F_n = \ldots, 34, 21, 13, 8, 5, 3, 2, 1, 1 \qquad \text{(least-significant weight on right)}$$

can be used as the weight vector, in conjunction with a digit vector $\mathbf{D}$ in which $d_i \in \{0, 1\}$, for a representation which resembles a binary number. This gives what is now called the *Zeckendorf representation* of the integers; we will denote the Zeckendorf representation of N as $\mathcal{Z}(N)$. The Zeckendorf representation usually omits the redundant bit corresponding to $F_1 = 1$, so that the least-significant bit corresponds to F_2 (which is also equal to 1). For example as $30 = 21 + 8 + 1$, $\mathcal{Z}(30) = 1010001$. It has the important property that the Zeckendorf representation of a positive integer will never have two or more adjacent 1s; by the definition of its Fibonacci weights, any bit string such as $\ldots 00110 \ldots$ is equivalent to $\ldots 01000 \ldots$.

The Zeckendorf representations are of more than just intellectual interest. For example, Apostolico and Fraenkel[1] and Fraenkel and Klein[4] show that the Zeckendorf representations (although they do not call them by that name) are the basis of a "variable length" representation of the integers. These representations are important in coding theory, where a sequence of integers must be represented as a stream of bits, such that the average length of each integer in the bit stream is minimised, and the representations are self-delimiting. By transmitting the Zeckendorf representation least-significant bit first and following its most significant 1 by another 1, we get the illegal sequence $\ldots 011$ which can act as a terminating "comma".

Here though, we are more interested in showing that it is possible to perform arithmetic on integers in the Zeckendorf representation.

2 Arithmetic with Zeckendorf integers.

There is little prior work in this connection. Graham, Knuth and Patashnik[7] discuss the addition of 1 in the Zeckendorf representation, but do not proceed to actual arithmetic. Freitag and Phillips [6] discuss addition and multiplication, and refer to Filiponi [2] and their own earlier paper [5] for subtraction. Thus no previous work discusses arithmetic as a coherent whole, covering all of the major operations, including multiplication and division.

The emphasis of this paper is frankly pragmatic, developing practical algorithms to perform the arithmetic operations. All have been implemented

and tested on a computer. Most of the algorithms are developed by analogy with conventional arithmetic methods, supplemented as necessary by the requirements and constraints of the Zeckendorf representation. For example, multiplication will be performed by the addition of suitable multiples of the multiplicand, selected according to the bit pattern of the multiplier. Division will use a sequence of trial subtractions, as in normal long division.

2.1 Addition

We start addition by adding each pair of bits as separate numbers, giving an initial sum whose digits are $d_i \in \{0, 1, 2\}$, where each d_i corresponds to its Fibonacci number F_i. We then sweep over the whole representation until there is no further change, applying the following rules to eliminate the 2s (which are illegal digits) and consecutive 1s. The representation must be extended by one place to include as a trailing digit the d_1 term which is usually omitted.

Removal of '2' digits From the fundamental relation that $F_n = F_{n-2} + F_{n-1}$, it is readily shown that $2F_n = F_{n+1} + F_{n-2}$. In digit patterns, we replace $\ldots 00200 \ldots$ by $\ldots 01001 \ldots$, subtracting the 2 and adding the two 1's to the nearby positions. Equivalently, a digit pattern $x2yz$ transforms to $(1 + x)0y(1 + z)$. A least-significant digit pattern of $\ldots 20$ clearly overflows beyond the least significant bit. We handle this by temporarily extending the representation by one place to include the d_1 digit so that the original $\ldots 020$ converts to $\ldots 1001$. (This rule does *not* apply to the d_2 and d_1 terms with weights of 1; this is covered by the special case below.)

Removal of adjacent 1s Again using the fundamental relation $F_n = F_{n-2} + F_{n-1}$, we can replace two adjacent non-zero digits by a more-significant 1. This step should be performed by a left-to-right scan through the representation to avoid a "piling up" of the left-propagating carry with long runs of 1s.

Least-significant 1s The first rule fails if we have a 2 in the least-significant (F_2) digit, because there is nowhere to receive the rightward carry propagation. In this case we restore the F_1 term and replace the least-significant $\ldots 20$ by $\ldots 11$, which has the same numeric value. If the *extended* bit pattern is now $\ldots 111$, the first two 1s may be eliminated

Addition	Fibonacci weights	F_{i+1}	F_i	F_{i-1}	F_{i-2}
Consecutive 1s		x	y	1	1
	becomes	x	$y+1$	0	0
Eliminate a 2	here $x \geq 2$	w	x	y	z
	becomes	$w+1$	$x-2$	y	$z+1$
Add, right bits		F_3	F_2	F_1	
$d_2 \geq 2$	here $x \geq 2$		x	0	
	becomes		$x-1$	1	
$d_2 \geq 2$ (alternate)		w	x	0	
	becomes	$w+1$	$x-2$	0	
$d_1 = 1$			0	1	
	becomes		1	0	

Subtraction	Fibonacci weights	F_{i+2}	F_{i+1}	F_i	F_{i-1}	F_{i-2}
eliminate -1		1	0	0	0	-1
	becomes	0	1	1	0	-1
	and again	0	1	0	1	0

Table 1: Adjustments and corrections in addition and subtraction

by the "adjacent 1s" rule. If the F_3 bit is a 0, bit pattern $\ldots 011$, we can immediately transform to $\ldots 100$ (still extended), and then eliminate the extension bit. It may in turn be replaced by $\ldots 100$, by the rule for consecutive 1s. (This rule could be eliminated entirely by an extension of the representation to include d_0 with a non-standard weight $w_0 = 1$.)

Remove the temporary d_1 term If at any stage, $d_2 = 0$ and $d_1 = 1$ we can set $d_2 = 1$ and set $d_1 = 0$ (the two bits have the same weight), which is equivalent to discarding the d_1 term which was introduced. Setting $d_2 = 1$ may force a removal of adjacent 1s.

Zeckendorf addition has *two* carries, one going one place left to higher significance and one two places right to lower significance. The first is entirely analogous to the carry of conventional binary arithmetic, while the second reflects the special nature of the Zeckendorf representation.

These adjustments and corrections are summarised in Table 1, which also includes the sign-fill from subtraction (Section 2.2). Note that in all cases

augend		1	0	0	0	0	1	0	1	= 38
addend			1	0	0	0	0	1	0	= 23
initial sum		1	1	0	0	0	1	1	1	= 61
consecutive 1s	1	0	0	0	0	1	0	0	1	= 61
result	1	0	0	0	0	1	0	0	1	= 61
Check – 38 + 23 = 61										

augend				1	0	0	0	1	0	= 15
addend			1	0	0	0	0	1	0	= 23
initial sum			1	1	0	0	0	2	0	= 38
carries			1	1	0	0	1	0	0	1 = 38
consecutive 1s		1	0	0	0	0	1	0	0	1 = 38
remove F_1 bit		1	0	0	0	0	1	0	1	= 38
result		1	0	0	0	0	1	0	1	= 38
Check – 15 + 23 = 38										

Figure 1: Two addition examples, $(38 + 23)$ and $(15 + 23)$

which show a 1 being inserted, the real action is to *add* the 1 to the previous value of that digit; eliminating one 2 may very well change another 1 to a 2, which must in turn be corrected. The removal of a 2 is likewise performed as a *subtraction*, rather than a simple deletion.

The addition may be compared with conventional binary addition. Binary addition (or decimal, or in any other polynomial number system) has a single carry which propagates to more-significant digits. If we start the add from the least-significant end we need only a single pass and the carry management is readily included in the standard simple algorithm. The two carries of Zeckendorf addition make the operation much more complicated and seem to necessitate multiple passes to absorb carries.

To illustrate, Figure 1 shows the addition of $38+23 = 10000101+1000010$ and $15+23 = 100010+1000010$. Both display the decimal value to the right of each line to emphasise that the correction and redistribution of bits does not affect the value. One example shows the temporary extension of the Zeckendorf representation to include the F_1 term. Each line presents the representation and value after the operation given at the start of the line. The various rules of Table 1 may be applied in any order, possibly changing the intermediate values but not the final result.

subtrahend	1	0	1	0	0	0	0	1	= 48
minuend	1	0	0	0	0	1	0	0	= 37
subtract digit-by-digit			1	0	0	-1	0	1	= 11
rewrite 1000				1	1	-1	0	1	= 11
rewrite 0110, cancelling -1				1	0	0	1	1	= 11
rewrite adjacent 1s				1	0	1	0	0	= 11
Result				1	0	1	0	0	= 11

Figure 2: Example of subtraction – $(48 - 37)$

2.2 Subtraction

For subtraction say $X - Y \to Z$, where X is the minuend, Y the subtrahend and Z the difference, we start with a digit-wise subtraction $x_i - y_i \to z_i$, giving $z_i \in \{-1, 0, 1\}$. The two values 0 and 1 pose no problem, as they are valid digits in $\mathcal{Z}(Z)$.

The case $z_i = -1$ is rather more difficult. From where $z_i = -1$ we scan to its left looking for the next most-significant 1 bit. Then rewrite this bit by the Fibonacci rule $100\ldots \to 011\ldots$, and then repeat rewriting the rightmost 1-bit of the pair of 1s

$$1000\ldots \to 0110\ldots \to 001011\ldots \to 00101011\ldots$$

until one of the two rightmost 1 bits coincides in position with the -1 of the result and cancels it, leaving a 0 result. (There may of course be *no* more significant 1. This corresponds to a negative result; we introduce a suitable large F_n and proceed from there, producing an "F_n complement" as discussed later.) The scan for digits $z_i = -1$ should be performed from most-significant to least-significant digits. This action is included in Table 1 earlier.

The preceding rule eliminates all of the digits whose value is -1, but often introduces other digits greater than 1, or pairs of adjacent 1s. All of these situations must be handled by the rules already introduced for addition. Subtraction is therefore an extension of addition.

Figure 2 shows an example, subtracting 37 from 48. As with addition, the various rewriting rules may be applied in any order; changing that order will change the finer details of the subtraction.

Note that we cannot easily propagate a borrow left from the place where $z_i = -1$. The rewriting rule steps *two* positions at each step and without knowing the distance to the next-significant 1 we know neither the alignment of the 01 bits which are introduced nor which of the two final 11 bits will be finally cancelled. (Conventional binary subtraction rewrites, for example, 10000 as $01100 + 100 \rightarrow 01110 + 10 \rightarrow 01111 + 1$; the final +1 is cancelled against the 1 of the subtrahend. Each stage proceeds by only one place and there is no ambiguity in reversing the process for the conventional right-to-left borrow propagation.)

2.3 Complementing

Subtraction quickly leads to negative numbers and their representations. Computer designers now prefer the 2s complement representation in which a number and its complement, added as unsigned quantities, total 2^n.

By analogy we can represent a negative value by its F_n complement. Also by analogy we say that a value is negative if its representation has its most-significant bit a 1. But this immediately introduces a major problem. An F_n complement representation has F_{n-2} values with a leading 1 and F_{n-1} values with a leading 0; there are about 1.6 times as many positive values as negative and about 38% of all positive values have no complement! (Complementing seemed almost incomprehensible until its asymmetrical range was realised. By analogy with binary numbers it was expected that there would be similar numbers of positive and negative values but there was no simple way of differentiating signed integers.)

If a positive number N requires n bits, then $N < F_{n+1}$. The signed number $-N$ requires at least $n + 2$ bits and must use at least the F_{n+3} complement. (Numbers of this precision will require at least the original n bits, place space for the sign. The "sign" of a negative number is, by definition, a '1' bit, but this 1 *must* be followed by a 0 for a valid Zeckendorf representation.)

Complementing is most easily handled by subtraction from zero; there seems to be no simple complementing rule. In comparison with binary arithmetic it is complicated considerably by the bi-directional carries and by the "sign-fill" pattern of 101010..., whose alignment with respect to the significant bits is not easily decided.

Some complements are shown in Table 2. We see that a negative number is characterised by a leading 1010... bit pattern, rather than the 1111...

N	$\mathcal{Z}(N)$	$F(8)$ comp	$F(9)$ comp	$F(10)$ comp	$F(11)$ comp
1	1	101010	1010101	10101010	101010101
2	10	101001	1010100	10101001	101010100
3	100	101000	1010010	10101000	101010010
4	101	100101	1010001	10100101	101010001
5	1000	100100	1010000	10100100	101010000
6	1001	100010	1001010	10100010	101001010
7	1010	100001	1001001	10100001	101001001
8	10000	100000	1001000	10100000	101001000
9	10001	–	1000101	10010101	101000101
10	10010	–	1000100	10010100	101000100
11	10100	–	1000010	10010010	101000010

Table 2: Illustration of Fibonacci $F(n)$ complements

usually associated with binary numbers. The $1010\ldots$ pattern has two alignments with respect to the bits of the value being complemented; these two alignments and interactions with the "numerically significant" bits lead to two different bit patterns in the complement. In the example, $\mathcal{Z}(7) = 1010$ and the two patterns are $\ldots 0001$ for n even and $\ldots 01001$ for n odd.

2.4 Multiplication

In the introduction we discussed the representation of an integer N as the scalar product

$$N = \mathbf{D} \cdot \mathbf{W}$$

where $\mathbf{D}$ is the *digit vector* (the visible digits of the representation) and $\mathbf{W}$ is a *weight vector*. We now develop multiplication by analogy with conventional multiplication, building on this representation. Only positive values will be considered for both multiplication and, later, division.

To calculate the product $Z \leftarrow X \times Y$, we first write X (the multiplier) as $\mathbf{X} \cdot \mathbf{W}$, giving

$$Z \leftarrow \mathbf{X} \cdot \mathbf{W} \times Y$$

whence

$$Z \leftarrow \sum x_i \cdot w_i \cdot Y \ = \sum x_i \cdot (w_i \cdot Y)$$

The product is the sum of appropriately weighted multiples of the multiplicand Y, each multiple in turn multiplied by the multiplier digit x_i. In a

multiplicand					1	0	0	1	0	1	= 17
multiplier						1	0	1	0	0	= 11
Make Fibonacci Multiples of multiplicand											
F_3 multiple			1	0	0	0	0	0	0	0	= 34
F_4 multiple			1	0	1	0	0	1	0	1	= 51
F_5 multiple		1	0	1	0	1	0	0	0	1	= 85
F_6 multiple	1	0	1	0	1	0	0	0	0	0	= 136
Accumulate appropriate multiples											
add F_4 multiple			1	0	1	0	0	1	0	1	= 51
add F_6 multiple	1	0	1	0	1	0	0	0	0	0	= 136
product =	1	0	0	1	0	0	1	0	0	0	1	= 187

Check that $17 \times 11 = 187$

Table 3: Example of Zeckendorf multiplication (17×11)

uniform base number system the scaling is easily done by "left shifting", or appending 0s to the right of **Y**, as in standard long multiplication.

With Fibonacci arithmetic, the scaling must mirror the generation of the Fibonacci numbers themselves; we can no longer use simple shifts or inclusion of 0s. The weight vector **W** is now the Fibonacci numbers; given a multiplicand Y, we generate its *Fibonacci multiples* M_n as –

$$M_1 = M_2 = \mathcal{Z}(Y),\ M_3 = M_1 + M_2,\ \ldots,\ M_k = M_{k-1} + M_{k-2},\ \ldots$$

and then add these weighted by the bits of $\mathcal{Z}(X)$. (All arithmetic is of course done using the Zeckendorf addition of Section 2.1.)

An example of multiplication given in Table 3. It shows first the two factors, then the multiples of the multiplicand, and finally the steps of the multiplication proper.

3 Division

Division is, as might be expected, the reversal of multiplication. The procedure is precisely that of conventional "long division", but adapted to use the Fibonacci multiples of multiplication rather than the scaled multiples of conventional arithmetic. Starting with the dividend, we try to subtract successively decreasing Fibonacci multiples of the divisor, entering quotient bits

dividend		1	0	0	1	0	0	0	1	0	1	0	1	= 300
divisor								1	0	0	1	0	1	= 17
Make Fibonacci Multiples of divisor														
F_2 multiple								1	0	0	1	0	1	= 17
F_3 multiple						1	0	0	0	0	0	0	0	= 34
F_4 multiple						1	0	1	0	0	1	0	1	= 51
F_5 multiple					1	0	1	0	1	0	0	0	1	= 85
F_6 multiple				1	0	1	0	1	0	0	0	0	0	= 136
F_7 multiple			1	0	1	0	1	0	0	0	0	0	1	= 221
F_8 multiple		1	0	1	0	1	0	0	0	0	0	0	1	= 357
F_9 multiple	1	0	1	0	1	0	0	0	0	0	0	1	0	= 578
Trial subtractions														
F_9 overdraw														
F_8 overdraw														
F_7 residue =					1	0	1	0	0	0	1	0	0	= 79
F_5 overdraw														
F_4 residue =							1	0	0	1	0	1	0	= 28
F_2 residue =									1	0	1	0	0	= 11
quotient =								1	0	0	1	0	1	= 17
remainder =									1	0	1	0	0	= 11

Check - $300 \div 17 = 17$ (+rem = 11)

Table 4: Example of Zeckendorf division ($300 \div 17$)

as appropriate, a 0 for an unsuccessful subtraction and a 1 for a successful subtraction. Again, we restrict ourselves to positive inputs.

Table 4 shows an example of Zeckendorf division. Note here that the dividend, assumed unsigned, must be extended by at least 2 bits to accommodate the negative values which arise during division.

We then enter the cycle of trial subtractions. At each stage, if the residue[1] is negative ("unsuccessful" subtraction), we enter a 0 bit in the quotient and restore the previous residue. If the residue is positive ("successful" subtraction), we enter a quotient bit of 1 and use the new residue for the next trial

[1]There is no generally accepted term for the working value from which divisor multiples are subtracted during division. While some authors use "partial remainder" or "partial dividend", the preference here is for "residue".

subtraction. As an optimisation with a successful subtraction, we can avoid the next multiple completely, going immediately from say M_i to M_{i-2}. The subtraction with the M_{i-1} multiple cannot succeed because that would give two consecutive 1s in the quotient.

4 Conclusions

Although we have demonstrated the main arithmetic operations on Zeckendorf integers, this arithmetic is unlikely to remain more than a curiosity. It is much more complex than normal binary arithmetic based on powers of 2 and the Zeckendorf representations are themselves more bulky than the corresponding binary representations.

Another problem lies in the representation of fractions. Powers of 2 extend naturally to negative powers and fractional values. Extending the Fibonacci numbers F_n for $n < 0$ repeats the values for positive n, but with alternating sign; they provide no way of representing fractions.

References

[1] A. Apostolico, A.S. Fraenkel, "Robust transmission of unbounded strings using Fibonacci representations", *IEEE Trans. on Inf. Th.*, Vol IT–33 (1987), pp 238–245.

[2] P. Filiponi, "The Representation of Certain Integers as a Sum of Distinct Fibonacci Numbers", Tech Rep. 2B0985. Fondazione Ugo Bordoni, Rome (1985).

[3] A.S. Fraenkel, "Systems of Numeration", *Amer. Math. Monthly*, Vol 92 (1985) pp 105–114.

[4] A.S. Fraenkel and S.T. Klein, "Robust universal complete codes for transmission and compression", *Discrete Applied Mathematics* Vol 64 (1996) pp 31–55.

[5] H.T. Freitag and G.M. Phillips. "On the Zeckendorf form of F_{kn}/F_n", *The Fibonacci Quarterly*, Vol 34, No 5, (1996) pp 444–446.

[6] H.T. Freitag and G.M. Phillips. "Zeckendorf arithmetic". *Applications of Fibonacci Numbers* Vol 7, pp129–132 G E Bergum, A N Philippou and A F Horadam (Eds.), Kluwer, Dordrecht 1998.

[7] R.L. Graham,, D.E. Knuth,, and O. Patashnik, *Concrete Mathematics*. Addison-Wesley, Reading, MA 1991, pp. 295-297.

[8] E. Zeckendorf, "Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas", *Bull. Soc. Roy. Sci. Liège*, Vol 41 (1972), pp 179–182

AMS Classification : 11B39, 03H15