ABSTRACT

Title of Thesis:	A LOGIC SYSTEM FOR FIBONACCI NUMBERS EQUIVALENT TO 64-BIT BINARY
	Jiani Shen, Master of Science, 2018
Thesis Directed By:	Professor Robert W. Newcomb, Electrical & Computer Engineering

Compared to the most commonly used binary computers, the Fibonacci computer has its own research values. Making study of Fibonacci radix system is of considerable importance to the Fibonacci computer.

Most materials only explain how to use binary coefficients in Fibonacci base to represent positive integers and introduce a little about basic arithmetic on positive integers using complicated but incomplete methods. However, rarely have materials expanded the arithmetic to negative integers with an easier way.

In this thesis, we first transfer the unsigned binary Fibonacci representation with minimal form(UBFR(min)) into the even-subscripted signed ternary Fibonacci representation(STFRe), which includes the negative integers and doubles the range over UBFR(min). Then, we develop some basic operations on both positive and negative integers by applying various properties of the Fibonacci sequence into arithmetic. We can set the arithmetic range equivalent to 64-bit binary as our daily binary computers, or whatever reasonable ranges we want.

A LOGIC SYSTEM FOR FIBONACCI NUMBERS EQUIVALENT TO 64-BIT BINARY

by

Jiani Shen

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
[Master of Science]
[2018]

Advisory Committee:
Professor [Robert W. Newcomb], Chair
Professor A. Yavuz Oruc
Professor Gang Qu

Acknowledgements

Sincere thanks to dear Professor Robert Newcomb. He guides me to do the thesis and reviews the papers for me diligently. Professor Newcomb is a really kind, considerate person. Even when I was in anxiety and depression for a long time, he still did not give me up. He inspires me in many aspects, teaching me to go out of the shadows and get back to life. I'm very grateful to him.

Professor Newcomb devoted himself to scientific research and education throughout his life. Wish him everlasting health.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

UBFR: the unsigned binary Fibonacci representation, with coefficients in the range of {0,1}

UBFR(min): the minimal form of the unsigned binary Fibonacci representation, with coefficients in the range of {0,1}

UBFR(max): the maximal form of the unsigned binary Fibonacci representation, with coefficients in the range of {0,1}

UQFRe{-1,0,1,2}: the unsigned quaternary Fibonacci representation only using even-subscript Fibonacci numbers, with coefficients in the range of {-1,0,1,2}

UQFRo{-1,0,1,2}: the unsigned quaternary Fibonacci representation, only using odd-subscript Fibonacci numbers, with coefficients in the range of {-1,0,1,2}

STFRe{-1,0,1}: the signed ternary Fibonacci representation only using even-subscript Fibonacci numbers, with coefficients in the range of {-1,0,1}

STFRo{-1,0,1}: the signed ternary Fibonacci representation only using odd-subscript Fibonacci numbers, with coefficients in the range of {-1,0,1}

subscript d: decimal system

Chapter 1: Introduction

1.1 Mathematical Foundation

There are many ways to represent numbers, among which binary and decimal are two most familiar uniform base number systems. It is obvious that useful codes are closely related to the number systems upon which they are based. Therefore, in search for other comparable or even better codes, it is worthy of looking for number systems that have algebraic structure as well as mathematical properties most useful to coding theory. The Fibonacci sequence of numbers, which is also named the golden section sequence, is such a special system. Introduced in 1202 by the mathematician Leonardo Fibonacci, theories and applications of Fibonacci numbers in modern mathematics began to develop rapidly starting since 60th years of the 20th century [1]. In the first Section, we will explore the mathematical foundation of Fibonacci sequence, including the simplest but critical properties for the Fibonacci numbers and Fibonacci coding theory.

1.1.1 Simplest Properties for The Fibonacci Numbers

The Fibonacci sequence is the series of numbers that obey some specific rule, which can be written as:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34 \ldots \tag{1.1}$$

It's apparent that the Fibonacci numbers are a recurrence numerical sequence, beginning with 0 and 1, and the next number is calculated by adding up the two numbers before it. Therefore, the Fibonacci sequence of numbers can be defined by the linear recurrence equation [2]:

$$F_i = F_{i-1} + F_{i-2};\ F_0 = 0, F_1 = 1 \tag{1.2}$$

Sometimes we will use the alternate form [3] of this equation $F_i = F_{i+1} - F_{i-1}$ for the convenience of deduction and calculation. If needed, we can expand the Fibonacci sequence by including the negative-subscripted Fibonacci numbers like F_{-1}, F_{-2} ..., which also follows the above recurrence relation.

As to the connection between the Fibonacci sequence and the golden section, here is the general term formula:

$$F_n = 1/\sqrt{5}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] \tag{1.3}$$

The Fibonacci sequence of integers is closely related to irrational numbers [4], which is very intriguing. And $\varphi = (1+\sqrt{5})/2 = 1.6180$ ... is famous as the golden ratio [5].

In this thesis, we will not use the knowledge of the gold section, but only the following simplest mathematical properties of the Fibonacci numbers [2, 6]:

$$F_1 + F_2 + \cdots + F_n = F_{n+2} - 1 \tag{1.4}$$

$$F_1 + F_3 + F_5 + \cdots + F_{2k-1} = F_{2k} \tag{1.5}$$

$$F_2 + F_4 + F_6 + \cdots + F_{2k} = F_{2k+1} - 1 \tag{1.6}$$

Based on the recurrence relation, we can attain (1.4) - (1.6) by mathematical reasoning as follows:

For the equation (1.5),

$$F_1 + F_3 + F_5 + \cdots + F_{2k-1} = \sum_{i=1}^{k} F_{2i-1} = \sum_{i=1}^{k} (F_{2i} - F_{2i-2})$$
$$= \sum_{i=1}^{k} F_{2i} - \sum_{i=1}^{k} F_{2i-2} = \sum_{i=1}^{k} F_{2i} - \sum_{i=1}^{k-1} F_{2i} = F_{2k}$$

For the equation (1.6),

$$F_2 + F_4 + F_6 + \cdots + F_{2k} = \sum_{i=1}^{k} F_{2i} = \sum_{i=1}^{k} (F_{2i+1} - F_{2i-1})$$
$$= \sum_{i=1}^{k} F_{2i+1} - \sum_{i=1}^{k} F_{2i-1} = \sum_{i=1}^{k} F_{2i+1} - \left(\sum_{i=1}^{k-1} F_{2i+1} + F_1 \right) = F_{2k+1} - 1$$

For the equation (1.4), in accordance with (1.5) and (1.6),

$$F_1 + F_2 + \cdots + F_{2k-1} + F_{2k} = F_{2k} + F_{2k+1} - 1 = F_{2k+2} - 1$$

1.1.2 Fibonacci Coding

Recalling the classical binary code which we are all familiar with, the essence of it is that every number can be represented as:

$$\mathrm{N} = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_{i-1}2^{i-1} + \cdots + b_0 2^0 \tag{1.7}$$

where $b_i \in \{0,1\}$ is a binary coefficient, $2^i (i = 0, 1, 2, \ldots, n-1)$ is the weight of the i-th digit.

Besides, we have the following mathematical property of binary numbers, which is of considerable importance to the binary system computer:

$$2^{n-1} + \cdots + 2^i + \cdots + 2^1 + 2^0 = 2^n - 1 \tag{1.8}$$

Using the similar idea, we find that every number is the sum of a collection of different Fibonacci numbers, which will be proved in Chapter 2. Thus, the n-bit Fibonacci code means the following positional representation of natural numbers:

$$\mathrm{N} = a_n F_n + a_{n-1} F_{n-1} + \cdots + a_i F_i + \cdots + a_2 F_2 + a_1 F_1 \tag{1.9}$$

where $a_i \in \{0,1\}$ is also a binary coefficient, $F_i (i = 1, 2, 3, \ldots, n)$ is the weight of the i-th digit.

The n-bit Fibonacci code (1.9) can be abbreviated as (1.10), which is called an unsigned binary Fibonacci representation (UBFR) of a non-negative integer N:

$$\mathrm{N} = a_n a_{n-1} \ldots a_i \ldots a_2 a_1 \tag{1.10}$$

The brief notation (1.10) consists of the n bits, starting from the highest bit a_n to the lowest bit a_1.

It should be noted that there are several unsigned binary Fibonacci representations, which means we can choose different sets of Fibonacci numbers to represent the same number. As we will introduce in Chapter 2, among these representations, the minimal form and the maximal form of UBFR are of particular interest. Besides, there are conversions from binary expressions to multilevel expressions, such as ternary Fibonacci representations, and quaternary Fibonacci representations. Besides, the coefficients are not limited to zero and unity.

1.2 Fibonacci Logic System Introduction

As is known to all, we can have different logic systems to express the same number. Binary and decimal systems are two typical representatives of the conventional uniform base number systems. Besides, mixed base systems are widely applied to measurements.

We use the binary system in almost all computers, since the binary system computer has its unique, incomparable advantages over other common systems such as ternary or decimal systems. However, the binary system still has some shortcomings and inconvenience in applications. Therefore, we are always searching for possible comparable or even better systems, and the Fibonacci system is apparently a good choice because of its fault tolerance [7] resulting from the redundancy presented in the Fibonacci base.

1.2.1 Computer for Binary Base

The conventional computers with binary base have three most important advantages as:

1. It's technically easy to implement. It is easy to use a bi-stable circuit to represent binary digits 0 and 1.

2. It's highly reliable. Since we only use digits 0 and 1 in binary system, it's much safer during transmission and processing, which ensures high reliability of the computer.

3. The operation rules are simple. Compared with decimal numbers, the operation rules of binary numbers are much simpler, which not only simplifies the structure of the arithmetic units, but also improves the speed of operations.

However, binary system computers do have their defects, such as difficulty of memory and read. What's more, the binary system is an incomplete set if we delete some weight 2^i on the i+1 digit. Every number has its unique representation in the base 2 system, which indicates the binary system is not very fault-tolerant.

1.2.2 Computer for Fibonacci Base

Comparing the classical binary code (1.7) with the unsigned binary Fibonacci code (1.9), as well as comparing the property for the binary numbers (1.8) with the properties for the Fibonacci numbers (1.4) - (1.6), we can find both have some similar parts. We have a good opportunity using Fibonacci numbers as a base to create effective algorithms for Fibonacci calculators, since we can adopt the advantages present in the binary system like simple expressions, similar mechanisms.

Another important point is that the Fibonacci system forms a complete set by itself or even when any one of the Fibonacci numbers is absent from the system, the property of which is not present in the binary system. If any one of the Fibonacci numbers is deleted, we still can express every non-negative integer. Therefore, we can pursue the goal of fault tolerance using the simplicity property of the Fibonacci system.

At the end of this section, we want to show an example of getting started in the research of the Fibonacci device.

Based on Zeckendorf's theorem introduced in the next Chapter, scientists Alexey Stakhov, Alexey Borisenko and Svetlana Matsenko displayed a Fibonacci counter for the minimal form [1] as a comparison with the classical binary counter.

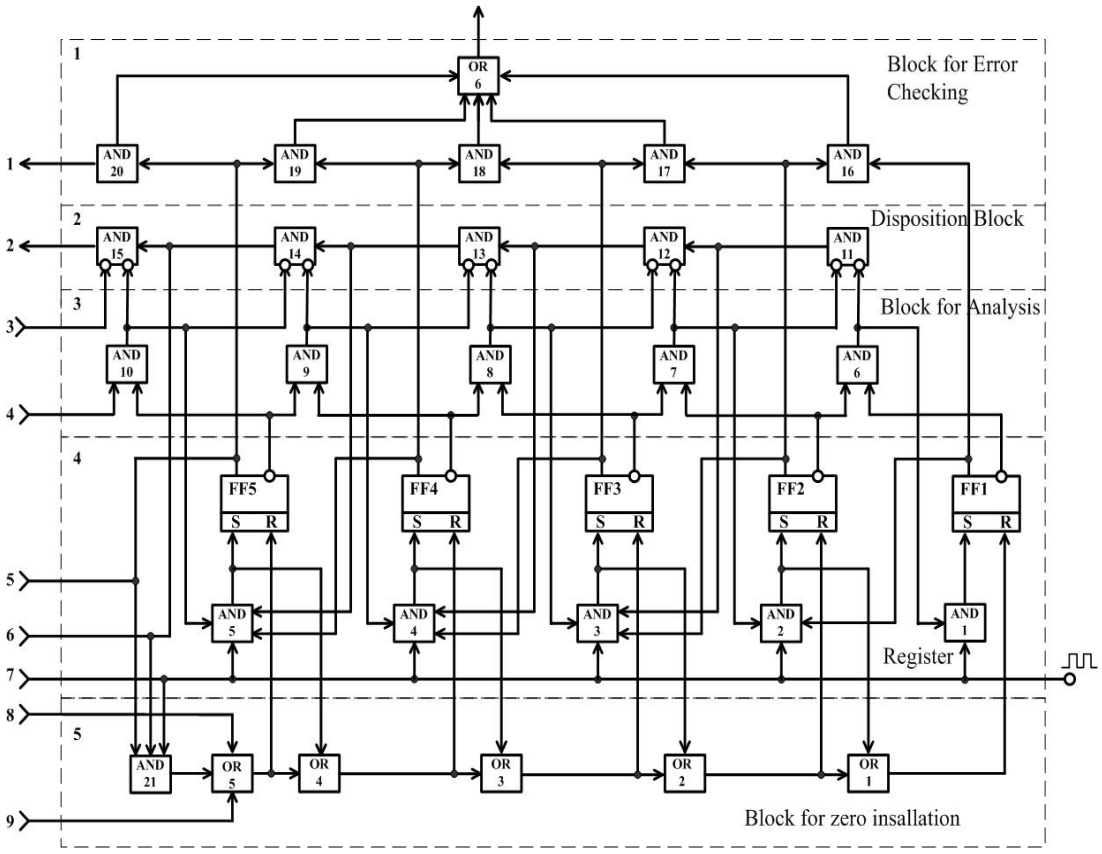

Figure 1.1 The Fibonacci counter for the minimal form [1]

As the figure 1.1 shows, the regular structure of the Fibonacci counter can be of unlimited length, which is the same as for the binary counter. We will not describe the structural and functional details of this counter, since it is not what we are going to discuss in this thesis. But we want to point out that the Fibonacci counter gives some advantages over the conventional binary counters. First, It's noise-immune. This is achieved as the result of the presence of the "forbidden" state in the Fibonacci counter. Second, it has high speed. This is caused by the absence of the preceding carry-overs needed in the binary counters, which saves much time. Third, it is informational reliable, since it consists of a block for error checking.

Therefore, it makes sense to use the Fibonacci counter instead of the classical binary counter, if we require some device with the properties of high noise immunity, sufficiently high speed but with familiar regular structure.

By this example, we want to show that the Fibonacci counter for the minimal form, based on Zeckendorf's theorem, is the first step for the deep research of reliable Fibonacci computers. It inspires us to continue developing the Fibonacci systems, figuring out some basic operations among numbers rather than limited to counting.

1.3 Project Overview

According to the contents introduced above, it is worth studying the Fibonacci logic system deeper. The Fibonacci code has valuable mathematical properties, deserves more research, and these properties make the Fibonacci system better than the binary system in some places. Therefore, it makes sense to sufficiently utilize the previous knowledge of the Fibonacci numbers and go further to develop some easier ways to perform basic operations in the Fibonacci system.

1.3.1 Overview of The Project

In this project, we first need to review and sufficiently make use of the important mathematical properties of the Fibonacci numbers, because these are the bases of our project. We study the knowledge of Zeckendorf's theorem and deduce the minimal & maximal forms from it. There are several unsigned binary Fibonacci Representations, but the UBFR for the minimal form is the most widely used for Fibonacci codes.

Most materials only explain how to use binary coefficients in Fibonacci base to represent non-negative integers, and a small number of them introduce some basic arithmetic on non-negative integers through complicated but incomplete methods. However, rarely have materials expanded the Fibonacci expressions and the arithmetic to negative integers. Even though some papers mentioned a little about the negative expressions and arithmetic, such as using complementation, those methods are impractical because of complication and waste of bits used as sign symbols.

In order to realize the negative representations and operations more easily, we transfer the UBFR(min) into multilevel Fibonacci representations like the ternary Fibonacci representation, or the quaternary Fibonacci representation. Then, we find the even-subscripted signed ternary Fibonacci representation(STFRe) is the optimal one to realize our goal. STFRe can easily solve the issues in expressions of negative integers, double the operation range compared with UBFR(min), and thus make the basic arithmetic much less complicated.

Based on STFRe, we develop the four fundamental operations in the Fibonacci system: addition, subtraction, multiplication and division. This method can easily process both non-negative and negative integers and solves the overflow problem without unnecessary bits.

Finally, since our daily computers are usually 32-bit binary or 64-bit binary, we can set the arithmetic range equivalent to 64 bits for comparison. It should be noted that the arithmetic range is not limited to 64 bits but whatever reasonable ranges we require.

1.3.2 Contributions of The Thesis

1. Implementation of Fibonacci arithmetic based on the STFRe{-1,0,1}.
    1) Develop principles for the four fundamental arithmetic operations, namely addition, subtraction, multiplication and division, on integers based on the STFRe{-1,0,1}.
    2) Solve carry problems occurring in the arithmetic operations.

3) Create flowcharts for the whole procedures of the four arithmetic operations.
4) Write programs to implement the four arithmetic operations on the computer.

2. Handling of overflow problem.
   1) Develop some clever ways to check overflow occurring in the arithmetic operations.
   2) Apply overflow checking methods to the practical use to avoid further mistakes or unnecessary steps.

Chapter 2: Fibonacci Representations

2.1 Unsigned Binary Fibonacci Representation

As we have mentioned in the previous Chapter, every non-negative integer is the sum of some set of Fibonacci numbers. Usually we will exclude the Fibonacci number F_1 and start with F_2 to represent a number, which will be explained in the next Sections. According to [7], although using F_1 is convenient for error correction, starting with F_2 gives a more "efficient system" to process.

It should be noted that different sets of Fibonacci numbers can have the same sum, thus those sets represent the same positive integer. For example, if we want to express the number 3 in the Fibonacci system, we can either represent 6 by $F_2 + F_5 = 6$ or by $F_2 + F_3 + F_4 = 6$. Thus, there are two sums for 6. It is the result of the completeness property of the Fibonacci base.

In the Fibonacci system, every non-negative integer can be expressed with appropriate coefficients. Since the binary Fibonacci representation(BFR) in (1.10), namely N $= a_n a_{n-1} \dots a_i \dots a_2 a_1$, is the basic form in the system, we will get started with it.

2.1.1 Zeckendorf's Theorem

As mentioned above, given an arbitrary non-negative integer, usually there are several possible expressions available in the BFR. For example, if we want to represent the number 3 in the 4-bit Fibonacci system, i.e. $a_4 a_3 a_2 a_1$, we can either express as $3 = F_2 + F_3 = 0110$ or as $3 = F_4 = 1000$. The representation is for the most significant bit on the left. Here, "4-bit" means we use 4 Fibonacci numbers, from F_1 to F_4, as weights to represent a non-negative integer, and we usually set the coefficient $a_1$as 0 for generality, which will be explained later. The representation is written from the most significant digit to the least significant digit, left to right.

The redundancy property discussed above is what we desire for a fault tolerant computer [8]. However, in order to make Fibonacci calculations and operations performed easily, we prefer to express every number in a uniform, unique form. Thus, among all those redundant expressions, we want to use algorithms to choose a fixed, unique expression in the application of the Fibonacci system.

Then here comes the important Zeckendorf's theorem [9]. But before proving it, we need to firstly verify another theorem, labeled as "Theorem 1" in [1]: Given an arbitrary positive integer N, it has exactly one representation in the form:

$$N = F_i + r \tag{2.1}$$

where $F_i (i = 2, 3, 4, \dots)$ is some Fibonacci number, and r is some non-negative integer subjected to:

$$0 \leq r < F_{i-1} \tag{2.2}$$

First we can directly find that (2.1), (2.2) and (1.2) indicate the range of N must be $F_i \leq N < F_{i+1}$. What we need to prove next is that the inequality is unique for each positive number N.

Now let's prove this in detail. Recalling (1.1), we have the Fibonacci sequence as $\{F_0 = 0, F_1 = 1, F_2 = 1, 2, 3, 5, \dots, F_i, F_{i+1}, \dots\}$. Based on the recurrence relation (1.2), it's obvious that starting from the lowest Fibonacci number $F_2 = 1$, the Fibonacci sequence is strictly increasing. Since any positive integer N is no less than 1, we can pick exactly one pair of adjacent Fibonacci numbers F_i and F_{i+1} from the Fibonacci sequence, and that pair should have the following inequality relationship with a given positive integer N:

$$F_i \leq N < F_{i+1} \tag{2.3}$$

In terms of the inequality (2.3) as well as both positive integers of F_i and N, we define the following difference:

$$r = \mathrm{N} - F_i \tag{2.4}$$

where r is a non-negative integer. We find (2.1) is an alternate form of (2.4).

Then subtracting F_i from each number in (2.3), we get:

$$0 \leq \mathrm{N} - F_i = r < F_{i+1} - F_i \tag{2.5}$$

Based on the difference (2.4) and the alternate form of recurrent relation (1.2), we finally have the condition (2.2), and the whole proof is done.

Usually a number system is most useful in operations if it has a unique representation of every integer. Therefore, we need some unique expression in the Fibonacci systems to easily process the calculations. Then here comes Zeckendorf's theorem: Every positive integer can be expressed as the unique some of non-consecutive Fibonacci numbers [1].

To prove Zeckendorf's theorem, we should demonstrate the following two points: 1. Every positive integer can be expressed as some of non-adjacent Fibonacci numbers. 2. For each positive integer, it's Zeckendorf representation is unique.

Proof. Using the idea of the theorem proved above, we come upon a simple algorithm to deduce the Zeckendorf representation. Suppose we want to represent a positive integer N in the n-bit Fibonacci code, the weights of which, from highest to lowest, are listed as follows:

$$\{F_n, F_{n-1}, \dots, F_{i+1}, F_i, \dots, F_2, F_1\} \tag{2.6}$$

It's obvious the positive integer N is subjected to the condition

$$0 < N < F_{n+1} \tag{2.7}$$

because in terms of (2.3) and (2.4), the binary coefficient $a_{n+1} = 1$ and we need the extra weight F_{n+1}.

Now let's compare the positive number $N < F_{n+1}$ with the weight F_n in the highest digit: If $F_n \leq N < F_{n+1}$, then the corresponding binary coefficient $a_n = 1$. Then we set the remainder $\mathrm{r}_1 = N - F_n$. According to the previous theorem, r_1 should satisfied the condition $0 \leq \mathrm{r}_1 < F_{n-1}$. Therefore, the coefficient a_{n-1} adjacent to a_n must be 0. If $\mathrm{N} < F_n$, we directly set $a_n = 0$ and the remainder $\mathrm{r}_1 = N$.

After the first comparison, we need to check whether $\mathrm{r}_1 = 0$. If $\mathrm{r}_1 = 0$, then we just need to set all the other coefficients as 0 and the Zeckendorf representation is finished. If r_1 is still a positive integer, we continue a comparison between r_1 and the next adjacent Fibonacci number candidates $F_{n-2}, \dots, F_{i+1}, F_i, \dots$. In terms of the previous theorem, we can find the exact one pair of adjacent Fibonacci numbers F_i and F_{i+1} , connected with r_1 in the inequality $F_i \leq \mathrm{r}_1 < F_{i+1}$. Therefore, it comes to the corresponding $a_i = 1$. And since the remainder $r_2 = r_1 - F_i$ satisfies the condition $0 \leq \mathrm{r}_2 < F_{i-1}$, it automatically follows $a_{i-1} = 0$.

Based on the simple algorithm mentioned above, we can deduce all the binary coefficients $\{a_n, a_{n-1}, \dots, a_{i+1}, a_i, \dots, a_2, a_1\}$ to represent the positive integer N, and the representation contains the property that no consecutive coefficients a_{i+1} and a_i are both equal to 1. Besides, the representation is unique, because the previous theorem demonstrates that the inequality $F_i \leq N < F_{i+1}$ for each positive number N is unique. Therefore, the proof of Zeckendorf's theorem is done.

Here, we give an example to illustrate the whole procedure discussed above. Suppose we want to represent a positive integer $N = 50$ in the 10-bit Fibonacci code:

I. Since $34 = F_9 \leq N = 50 < F_{10} = 55$, we set $a_{10} = 0$ and $a_9 = 1$.

II. Set the remainder $r = N - F_9 = 50 - 34 = 16$. Automatically follows $a_8 = 0$.

III. Compare the remainder r with the next adjacent Fibonacci numbers starting from F_7, F_6, F_5 ... until we find $13 = F_7 \leq r = 16 < F_8 = 21$. Set $a_7 = 1$.

IV. Set the remainder $r = r - F_7 = 16 - 13 = 3$. Automatically follows $a_6 = 0$.

V. Compare the remainder r with the next adjacent Fibonacci numbers starting from F_5, F_4, F_3 ... until we find $3 = F_4 \leq r = 3 < F_5 = 5$. Set $a_5 = 0$ and $a_4 = 1$.

VI. Set the remainder $r = r - F_4 = 3 - 3 = 0$. Set the remaining binary coefficients 0s, namely $a_3 = a_2 = a_1 = 0$. Terminate the procedure.

VII. The Zeckendorf representation is $N = 50_d = 0101001000$

It is worth mentioning that in the Zeckendorf representation, a_1 is always equal to 0. The reason is: If we have $a_2 = 1$, it automatically follows $a_1 = 0$ as proved above. If we have $a_2 = 0$, it indicates currently the remainder r' is subjected to the inequality $0 \leq r' < F_1 = 1$, meaning that r' is exactly 0 and a_1 should be 0. Therefore, the binary coefficient a_1 is always equal to 0 based on the Zeckendorf representation.

Therefore, the maximum positive integer M that can be expressed by the n-bit Fibonacci code in the Zeckendorf representation is:

$$\mathrm{M} = \begin{cases} F_n + F_{n-2} + \cdots + F_4 + F_2 = F_{n+1} - 1 & if\ n\ is\ even \\ F_n + F_{n-2} + \cdots + F_5 + F_3 = F_{n+1} - 1 & if\ n\ is\ odd \end{cases} \tag{2.8}$$

It's apparent that the positive integer should be less than the weight F_{n+1}, if we want to express it in the n-bit Zeckendorf representation. This in turn proves the constraint in (2.7).

2.1.2 The Maximal & Minimal Forms

We have different forms of the Fibonacci representation to represent a non-negative integer, including binary and other multilevel Fibonacci representations. Among BFRs, we are especially interested in the maximal and the minimal forms. These two forms make calculations and operations in the Fibonacci systems much easier because of their uniqueness.

The Zeckendorf representation is also called the minimal form of the unsigned binary Fibonacci representation, abbreviated as the UBFR(min) and it is unique. As we have discussed above, the UBFR(min) has two special properties:

1. In every UBFR(min), the lowest bit a_1 is always equal to 0. Therefore, any n-bit Fibonacci code (1.10) in the minimal form can be expressed as:

$$N = a_n a_{n-1} \dots a_i \dots a_2 0 \tag{2.9}$$

And we can further abbreviate (2.9) as $\mathrm{N} = a_n a_{n-1} \dots a_i \dots a_2$ for convenience. The truncated n-bit Fibonacci code starts from the lowest bit a_2.

2. In every UBFR(min), no two adjacent bits can be 1 together. This means after each bit $a_{i+1} = 1$, it automatically follows $a_i = 0$. In the Boolean functions, the property is written as [10, 11]:

$$a_{i+1} a_i = 0, \quad i = 2,3, \dots, n-1 \tag{2.10}$$

Based on the process of the algorithm, it's obvious that the UBFR(min) consists of the least number of 1s compared to the other forms of UBFR to represent the same non-negative integer. We can use the simple algorithm mentioned before to obtain the n-bit UBFR(min) for an arbitrary non-negative integer N, which is subjected to $N < F_{n+1}$. It should be noted that the # of bits n is subjected to $n \geq 2$, otherwise it's meaningless, since a_1 is excluded from consideration. Here comes the flow chart of the minimal representation:

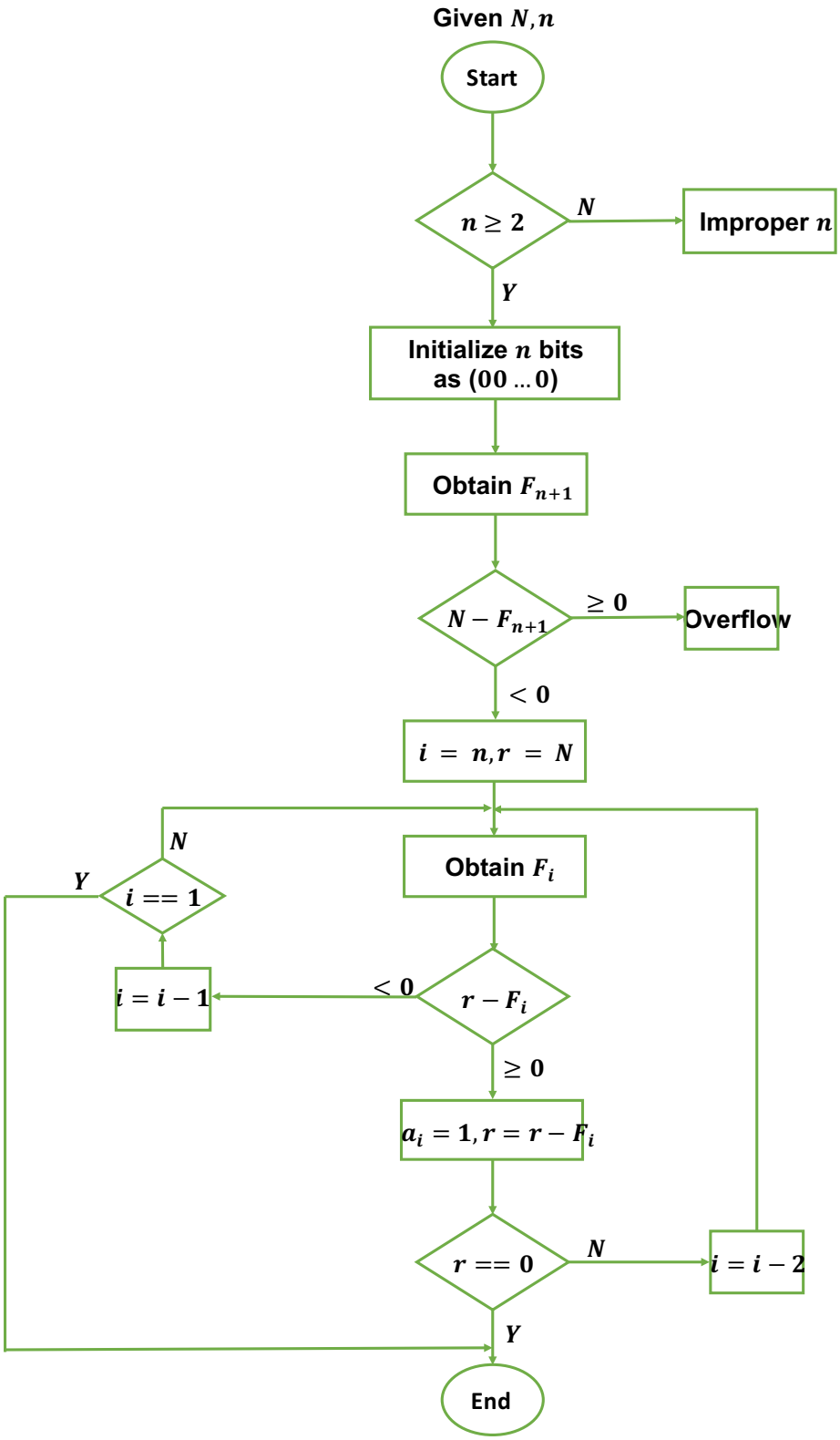

Figure 2.1 Flowchart for algorithm to obtain UBFR(min)

The maximal form of the UBFR, abbreviated is the UBFR(max), as its name implies, is a unique representation consisting of the most number of 1s compared to the other forms of BFR to represent the same non-negative integer. And in the Boolean functions, the UBFR(max) has the following property [12]:

$$a_{i+1} + a_i \geq 1, \quad i = 2,3, \dots, n-1 \tag{2.11}$$

This Boolean expression means every two consecutive bits must contain at least one 1 in the maximal form.

In terms of the recurrence relation (1.2), we can transfer from the minimal to the maximal via: Each time we replace "100" by "011" from right to left (or from left to right). And repeat this step until there is no more "100" in the representation.

For example, in the 8-bit UBFR(min), the number $N = 30$ is expressed as $N = 30 = F_8 + F_6 + F_2 = 10100010$. It should be noted that we exclude a_1 from the expression from now on because it's always 0 in the UBFR(min). Based on the maximal form rule, we have the transferring steps as follows:

$$\begin{aligned} N = 30 &= 10100010 \\ &= 10011010 \\ &= 01111010 \end{aligned}$$

In terms of the maximal form rule, we find there is no two adjacent 0s existing in the UBFR(max) except the integer 0. The integer 0 has the same expression $\underbrace{00 \dots 0}_{n}$ both in the n-bit UBFR(min) and UBFR(max).

In reverse, we can transfer from the maximal to the minimal via: Each time we should replace "011" by "100". But it should be a left-to-right scan (i.e. from the highest to the lowest bit) to avoid accumulation of the left-propagating carries [14]. And we need to repeat this step until no two adjacent 1s occur together. Here, we recover the maximal form of $N = 30$ obtained above to its minimal form:

$$\begin{aligned} N = 30 &= 01111010 \\ &= 10011010 \\ &= 10100010 \end{aligned}$$

One more thing, any UBFR can be transferred to the UBFR(min) using the minimal form rule, and transferred to the UBFR(max) using the maximal form rule.

Usually, we choose the UBFR(min) for calculations and operations of non-negative integers. In addition to its property of uniqueness, we have the following advantages:

1. It can be directly deducted by the simple algorithm shown in figure 2.1 without intermediate steps.
2. It contains the lowest number of 1s. Compared to the other UBFRs, we can avoid as many carries as possible when doing some calculation with the UBFR(min). And it looks simple and clear.
3. It can be easily converted to the signed ternary Fibonacci representation (STFR), which will be discussed later in detail.

2.2 Binary to Multilevel Conversion

We find in both the UBFR(min) and the UBFR(max) so far we have represented the non-negative integers. Even if we include the Fibonacci number with negative subscripts (like F_{-1}, F_{-2}…), it's still difficult to express the non-negative integers. The reason is that the values of the negative-subscripted Fibonacci numbers are with alternating signs. For example, based on the recurrence relation, we have $F_{-1} = -1$, $F_{-2} = 1$, $F_{-3} = -2$, $F_{-4} = 3$…, which makes the negative integers hard to be included only by binary Fibonacci representations.

In the binary coefficient Fibonacci system, if we want to include the negative part, we need to adopt some methods such as using complemented representations or sign-magnitude representation [7]. However, those methods are either wasting more bits or inconvenient to implement.

In order to figure out some easier way, we attempt to convert the binary coefficients to the multilevel coefficients in the Fibonacci logic system. And in the next two sub-sections, we are going to talk about quaternary conversions and ternary conversions in details.

Through these conversions, we can efficiently process the calculations as well as extend the range to the negative part, which are added features of the Fibonacci numbers in addition to redundancy.

2.2.1 Unsigned Quaternary Fibonacci Representation

First, let's recall the general formula of the n-bit UBFR excluding a_1 to express a non-negative integer N:

$$N = \sum_{i=2}^{n} a_i F_i, \quad a_i \in \{0, 1\} \tag{2.12}$$

The maximal integer than can be represented in this formula is

$$M = \sum_{i=2}^{n} F_i = F_{n+2} - 2$$

Therefore, the general n-bit UBFR is in the range of $[0, M]$. The flowchart in figure 2.2 shows a conversion algorithm to obtain a general n-bit UBFR for a given non-negative integer N [7].

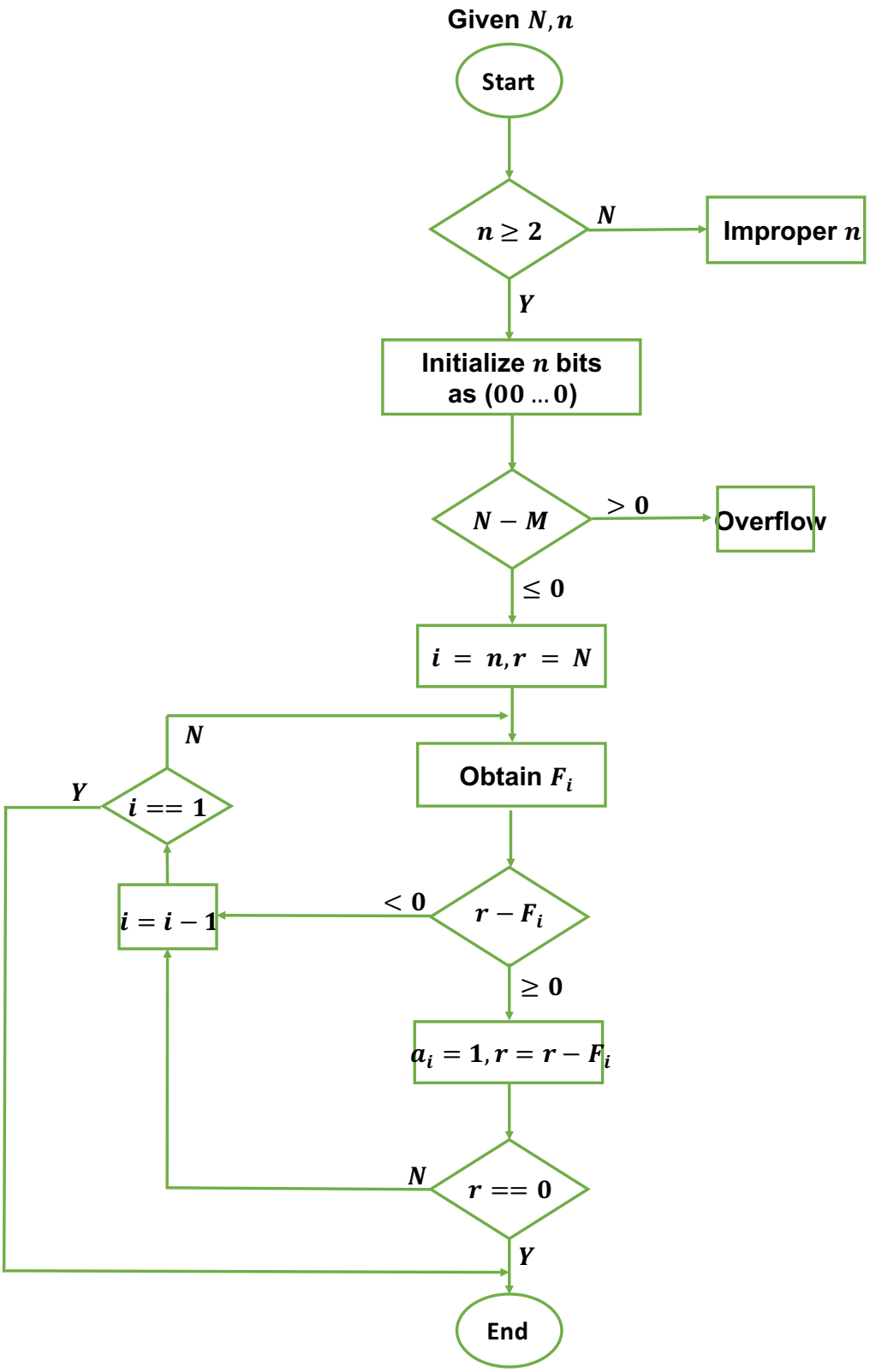

Figure 2.2 Flowchart for algorithm to obtain a general UBFR

In order to compare figure 2.1 and figure 2.2, we still exclude a_1 from consideration and set $a_1 = 0$ in figure 2.2. Indeed, a_1 can be included in figure 2.2. The differences between figure 2.1 and figure 2.2 are:

Figure 2.1 is the minimal form of UBFR, which contains the least number of 1s and complies with the rule (2.10). Figure 2.2 is just a general form of UBFR without special rules.

Given n-bit capacity, in the UBFR(min), the maximal value is $M = F_{n+1} - 1$; in a general UBFR, the maximal value is $M' = F_{n+2} - 2$. It's obvious that $M < M'$ under the premise that $n \geq 2$, since that we utilize the available Fibonacci numbers as many as possible in a general UBFR.

After we obtain an n-bit UBFR for a given non-negative integer N in terms of the flowchart in the figure 2.2, we can use it to generate an unsigned quaternary Fibonacci representation using only even-subscripted Fibonacci numbers (the definition is that we only use even-subscripted Fibonacci numbers, the corresponding coefficients of which can have four different candidates in the range {-1,0,1,2}, to represent a non-negative integer), i.e. UQFRe [13]:

$$q_{2i}^{e} = a_{2i-1} + a_{2i} - a_{2i+1} \tag{2.13}$$

where $q_{2i}^{e} \in \{-1,0,1,2\}, i = 1, 2, \dots, k$, and the superscript e just emphasizes that we only use even-subscripted Fibonacci numbers. $a_j \in \{0,1\}, j = 1,2, \dots n$ is the binary coefficient in the corresponding n-bit UBFR. And we have $k = \lceil n/2 \rceil$.

Therefore, we can use the UBFR and its corresponding UQFRe to represent the same integer N:

$$\mathrm{N} = \sum_{i=2}^{n} a_i F_i = \sum_{j=1}^{k} q_{2j}^{e} F_{2j}\,, \quad k = \left\lceil \frac{n}{2} \right\rceil \tag{2.14}$$

It should be noted that in the UQFRe converted from an n-bit UBFR, there are still about n coefficients but with all the odd-subscripted coefficients always equal to 0. And the even-subscripted coefficients q_{2i}^{e} can take four possible values among {-1,0,1,2}, which is shown in the table 2.1 [3]:

$$UBFR - to - UQFR_e\ Coefficient\ Conversion$$

Condition	a_{2i-1}	a_{2i}	a_{2i+1}	$q_{2i}^{e} = a_{2i-1} + a_{2i} - a_{2i+1}$
0	0	0	0	0
1	0	0	1	-1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	2
7	1	1	1	1

Table 2.1 Conversion from UBFR to UQFRe

As [15] mentions, if we subtract 1 from all subscripts in (2.13), we can easily use the same n-bit UBFR to get the unsigned quaternary Fibonacci representation using only odd-subscripted Fibonacci numbers, i.e. UQFRo. The relation is very analogous to (2.13):

$$q_{2i-1}^{o} = a_{2i-2} + a_{2i-1} - a_{2i} \tag{2.15}$$

where $q_{2i-1}^{o} \in \{-1,0,1,2\}, \mathrm{i} = 1, 2, \dots, k'$ and the superscript o just emphasizes that we only use odd-subscripted Fibonacci numbers. $a_j \in \{0,1\}, j = 1,2, \dots n$ is the binary coefficient in the corresponding n-bit UBFR. And we have $k' = [m/2] + 1$. It's obvious that in UQFRo, the odd-subscripted coefficients q_{2i-1}^{o} has the same value range as the even-subscripted ones.

Compared with the n-bit UBFR, we only need to take about n/2-bit of digits into consideration when using the corresponding UQFRe or UQFRo, since the other half must be 0. However, it seems that both UQFRe and UQFRo still can not express the negative integers. What's worse, the coefficients of them have four possible values to choose from, which makes expressions as well as calculations even more complicated.

In fact, we use the UQFR to bring up the STFR (the definition is that we use non-negative-subscripted Fibonacci numbers, the corresponding coefficients of which can have three different candidates, to represent both negative and non-negative integers), which is the most important topic in our thesis.

2.2.2 Signed Ternary Fibonacci Representation

In terms of [15], there are several unsigned ternary Fibonacci representations existing. Among them, we have two special representations that only use even-subscripted Fibonacci numbers, called the UTFRe{-1,0,1} and the UTFRe{0,1,2}.

In fact, both UTFRe{-1,0,1} and UTFRe{0,1,2} are two special examples of the UQFRe. Recalling the table 2.1, it shows how to convert the binary coefficients in a general UBFR to the quaternary coefficients in the corresponding UQFRe. If we use the unique UBFR(min) instead of an arbitrary one, we can eliminate the possible quaternary coefficient 2. The reason is that in every minimal form, no two adjacent bits are 1 together, which indicates that the cases 3, 6 and 7 will not occur based on the UBFR(min). Therefore, the UTFRe{-1,0,1} is derived from the UBFR(min). Instead if we use the unique UBFR(max), we can eliminate the possible quaternary coefficient -1. Since in every maximal form, every two adjacent bits should at least have one 1, which indicate that only cases 2, 3, 5, 6 and 7 are possible in the UBFR(max). Thus, we can use the UBFR(max) to deduce the UTFRe{0,1,2}.

Although the UTFRe{-1, 0, 1} and the UTFRe{0,1,2} are all unique ternary representations, we are especially interested in the UTFRe{-1,0,1}. Through a simple form of complementation, i.e. -1 <-> 1 and 0 <-> 0, we can easily convert a positive number N to its corresponding negative –N, or vice versa. Here are three examples:

1. We are given a positive integer $N_1 = 5 = F_6 - F_4 = 1 - 1\ 0$. It should be noted that in the UTFRe{-1,0,1}, we usually omit the odd-subscripted bits, and start with the lowest ternary bit t_2. Then, if we make the complementation on every bit of N_1, we can easily obtain the negative one as $-N_1 = -110 = -F_6 + F_4 = -5$.

2. Given a negative integer $N_2 = -12 = -F_6 - F_4 - F_2 = -1 - 1 - 1$, making the same complementation, we directly get the positive one as $-N_2 = 111 = F_6 + F_4 + F_2 = 12$.

3. If given the integer 0, it's apparent that we still get 0 after the complementation.

Therefore, the UTFRe{-1,0,1} automatically leads to the signed Fibonacci representation using only even-subscripted Fibonacci numbers, namely STFRe{-1,0,1}. And the relation of the UBFR(min)-to-STFRe{-1,0,1} conversion is as follows:

$$t_{2i}^{e} = a_{2i-1} + a_{2i} - a_{2i+1} \tag{2.16}$$

where $t_{2i}^{e} \in \{-1,0,1\}, \mathrm{i} = 1, 2, \ldots, k$, and the superscript e just emphasizes that we only use even-subscripted Fibonacci numbers. $a_j \in \{0,1\}, j = 1,2, \ldots n$ is the binary coefficient in the corresponding n-bit UBFR(min). And we have $\mathrm{k} = \lceil n/2 \rceil$. We find (2.16) is almost the same as (2.14). That's why we say the UTFRe{-1,0,1} is a special example of the UQFRe above. Besides, we notice that in the STFRe{-1,0,1}, we can use the sign of the most significant nonzero coefficient to determine the sign of the integer, which is based on (1.6):

$$F_{2k} + F_{2k-2} + \cdots + F_4 + F_2 = F_{2k+1} - 1 < F_{2k+2}$$

After making the simple complementation, the negative integers are contained in the STFRe{-1,0,1} and a doubling of range over the UBFR(min) is realized. In the n-bit UBFR(min), the maximal integer is $\mathrm{M} = F_{n+1} - 1$ and thus the range is $[0, M]$. And now after the UBFR(min)-to-STFRe{-1,0,1} coefficients conversion, the range is doubled as $[-M, M]$.

What's more, we can make the conversions of UBFR(min)-to-UTFRo{-1,0,1} and UBFR(max)-to-UTFRo{0,1,2} simply by subtracting 1 from all subscripts in (2.16), the manner of which is almost the same as (2.15). Through making the complementation, namely -1 <-> 1 and 0 <-> 0, the negative integers can be included and hence the UTFRo{-1,0,1} is automatically transferred to the STFRo{-1,0,1}.

Derived from the n-bit UBFR(min), we have the STFRe{-1,0,1}

$$\mathrm{N} = \sum_{i=1}^{k} t_{2i}^{e} F_{2i}\,, \quad k = \left\lceil \frac{n}{2} \right\rceil \tag{2.17}$$

and the STFRo{-1,0,1}

$$\mathrm{N} = \sum_{i=1}^{k'} t_{2i-1}^{o} F_{2i-1}\,, \quad k' = \left\lceil \frac{n}{2} \right\rceil + 1 \tag{2.18}$$

Since UBFR(min) is unique, STFRe{-1,0,1} and STFRo{-1,0,1} are also unique based on the conversion rules.

We find both t_{2i}^{e} and t_{2i-1}^{o} are in the set {-1,0,1}, which makes the ternary addition and subtraction very easy to implement. If we are required to calculate the sum of the two integers N_1 and N_2 in the n-bit Fibonacci system, we can express N_1 in the STFRe{-1,0,1} and express N_2 in the STFRo{-1,0,1}. Then, we can directly obtain their sum expressed in the STFR using full-subscripted Fibonacci numbers, called STFRf{-1,0,1}, by interleaving the bits of N_1 and N_2:

$$N_1 + N_2 = \sum_{i=1}^{m} t_i F_i\,, \quad m = n + 1 \tag{2.19}$$

where

$$t_i = \begin{cases} t_i^{e}, & if\ i\ is\ even \\ t_i^{o}, & if\ i\ is\ odd \end{cases} \tag{2.20}$$

Through the same method, we can easily get the difference between N_1 and N_2.

The advantages of the above method are:

1. We can immediately get the sum or the difference by interleaving, without an actual calculation.
2. There is no need to worry about carries.

3. The STFR contains the negative integers in calculations, which is an improvement compared to the UBFR.

4. When both STFRe{-1,0,1} and STFRo{-1,0,1} are available, it is simple for actual hardware implementations such as the register loading [3].

However, there are still some problems with this method:

1. In the example, we only consider about the addition or the subtraction between two integers. If there is a third added or minuend, we can't get the answer only through interleaving. Although we can do full- to even-/odd- subscript conversions introduced in [3] and continue using the interleaving method, it's still very complicated and has many preliminary preparation transformations to implement.

2. In addition to addition and subtraction, we also need to figure out how to solve multiplication and division, which are more complicated than the previous two. It's apparent that the method of interleaving is not enough.

3. If we have three STFRs to process, namely processing the STFRe{-1,0,1}, the STFRo{-1,0,1} and the STFRf{-1,0,1}, it is messy and more error prone for complicated operations, which

Based on the disadvantages mentioned above, it is a better idea to do calculations using a uniform form. Therefore, we first use the simple algorithm in figure 2.1 to obtain the UBFR(min) for N_1 and N_2, and then convert both into the STFRe{-1,0,1} or STFRo{-1,0,1}.

2.3 Advantages of The STFRe{-1,0,1}

In this Chapter, we demonstrate how to express a non-negative integer in the UBFR(min), which is the first step to introduce decimal numbers into the Fibonacci system. Then, since the UBFR(min) can only represent the non-negative integers, we try conversions between different Fibonacci representations. Finally, we choose to express integers and implement their operations with the STFRe{-1,0,1}.

And below are the advantages of the STFRe{-1,0,1}:

1. It doubles the range of numbers over the UBFR(min) by including the negative part.

2. It allows the expressions of both the non-negative integers as well as their negative ones to be realized easily, without using the Fibonacci numbers with negative subscripts or wasting more bits for the sign. We only need to invert all ternary coefficients in the number N's STFRe{-1,0,1} to obtain –N.

3. We can directly know the sign of an integer by checking the most significant non-zero bit of its STFRe{-1,0,1}.

4. Compared to the other Fibonacci representations mentioned before, it is most useful for hardware implementations using three-state devices [3], which indicates a further step for realizing the Fibonacci computer.

5. Compared to the STFRo{-1,0,1}, the STFRe{-1,0,1} is more convenient when processing carries in calculations, which will be explained in the next Chapter. And since the binary computer is usually 64 bits, using the STFRe{-1,0,1} is more intuitive for the comparison between the binary device and the Fibonacci one.

Finally, we display some examples in table 2.2 to illustrate the decimal-to-UBFR(min)-STFRe{-1,0,1} conversion.

$$Decimal - to - UBFR(min) - to - STFRe\{-1, 0, 1\}\ Conversion$$

Decimal System	$UBFR(min)$ *of the absolute value*								$STFRe\{-1, 0, 1\}$			
	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	t_8^e	t_6^e	t_4^e	t_2^e
-30	*1*	*0*	*1*	*0*	*0*	*0*	*1*	*0*	*-1*	*-1*	*0*	*-1*
-20	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*0*	*-1*	*0*	*0*	*1*
-10	*0*	*0*	*1*	*0*	*0*	*1*	*0*	*0*	*0*	*-1*	*-1*	*1*
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*1*
2	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*1*	*-1*
3	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*1*	*0*
4	*0*	*0*	*0*	*0*	*1*	*0*	*1*	*0*	*0*	*0*	*1*	*1*
5	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*1*	*-1*	*0*
6	*0*	*0*	*0*	*1*	*0*	*0*	*1*	*0*	*0*	*1*	*-1*	*1*
7	*0*	*0*	*0*	*1*	*0*	*1*	*0*	*0*	*0*	*1*	*0*	*-1*
8	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0*
9	*0*	*0*	*1*	*0*	*0*	*0*	*1*	*0*	*0*	*1*	*0*	*1*

Table 2.2 Decimal-to-UBFR(min)-to-STFRe{-1,0,1} conversion

Chapter 3: Fibonacci Arithmetic Based on The STFRe

3.1 Principle of The Four Fundamental Operations

In order to better understand the mechanism of how a base number system represent numbers, we can express an integer as the scaler product [14]

$$N = \mathbf{D} \cdot \mathbf{W} \tag{3.1}$$

where **D** is a row vector of digits in the representation, and **W** is a column vector of weights corresponding to the digits. Usually, the weight vector is written in the order with the decreasing subscripts as:

$$\mathbf{W} = [\dots, w_i, w_{i-1}, \dots, w_2, w_1, w_0] \tag{3.2}$$

We know that the weight vector **W** is derived from its base vector **B**:

$$\mathbf{B} = [\dots, b_i, b_{i-1}, \dots, b_2, b_1, b_0\ ] \tag{3.3}$$

where b_0 is usually equal to 1. And the relationship between **W** and **B** is:

$$w_k = \prod_{i=0}^{k} b_i \tag{3.4}$$

Here are some examples to demonstrate the expression in (3.1):

1. The binary system and the decimal system are two most typical representatives of the uniform base systems. The integer 36 can be obtained as $36 = [1\ 0\ 0\ 1\ 0\ 0] \cdot [2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0]^T$ in the binary system with the uniform base 2, and $36 = [3\ 6] \cdot [10^1\ 10^0]^T$ in the decimal system with the uniform base 10.

2. The mixed base systems are usually applied to measurements. As an example for the imperial units of weight measurement {tons, hundredweights, stones, pounds, ounces}, the base vector is $\mathbf{B} = [20, 8, 14, 16, 1]$. And according to (3.4), it derives the weight vector as $\mathbf{W} = [35840, 1792, 224, 16, 1]^T$ to unify the units in ounces.

As to the Fibonacci logic system, we can use the same idea to represent integers. In the unsigned binary Fibonacci system, i.e. UBFR, the Fibonacci numbers can be used in the weight vector **W** as $\mathbf{W} = [\dots, F_5, F_4, F_3, F_2, F_1]^T = [\dots, 5, 3, 2, 1, 1]^T$ (the least significant weight on the rightmost) and the digit vector **D** is composed by the binary element $a_i \in \{0, 1\}$. To represent a given non-negative integer N using an n-bit UBFR, the scalar product is written as:

$$\mathrm{N} = [a_n, a_{n-1}, \dots, a_2, a_1] \cdot [F_n, F_{n-1}, \dots, F_2, F_1]^T \tag{3.5}$$

It's obvious that (3.5) is exactly the same as (1.9). Since we mainly focus on the minimal form of the UBFR, i.e. UBFR(min), which is also called the Zeckendorf representation, the scalar product (3.5) is subjected to the following two conditions:

1. We set $a_1 = 0$, meaning that we always omit the weight F_1 in the UBFR(min).
2. We have the relationship $a_{i+1}a_i = 0, i = 1, 2, 3, \dots, n-1$, namely no two or more adjacent bits being 1s.

For example, the number 32 is represented as $32 = F_8 + F_6 + F_4 = 21 + 8 + 3$ in the UBFR(min), thus the corresponding scalar product is written as:

$$32 = [1\ 0\ 1\ 0\ 1\ 0\ 0\ 0] \cdot [21\ 13\ 8\ 5\ 3\ 2\ 1\ 1]^T$$

As to the scalar product of the integer N in the STFRe{-1,0,1}, which is converted from the UBFR(min), (3.5) is changed as:

$$\mathrm{N} = [t_{2k}^e, t_{2k-2}^e, \dots, t_4^e, t_2^e] \cdot [F_{2k}, F_{2k-2}, \dots, F_4, F_2]^T \tag{3.6}$$

where the even-subscripted ternary coefficient t_{2i}^e is obtain from the corresponding UBFR(min) by the relation (2.16). Therefore, to represent the same number 32 using the STFRe{-1,0,1}, the scalar product is written as:

$$32 = [1\ 1\ 1\ 0] \cdot [21\ 8\ 3\ 1]^T$$

The scalar product (3.6) is very useful in this thesis, since it not only helps us better understand the properties of the STFRe{-1,0,1}, but also makes the Fibonacci calculations using STFRe{-1,0,1} easier to do, which will be discussed in this Chapter later.

There is little previous work discussing basic arithmetic operations on integers in the Fibonacci system, especially using the STFRe{-1,0,1}, which contains many good advantages. Therefore in this thesis, our main purpose is to develop practical and simple algorithms to perform some basic arithmetic operations in the Fibonacci system, like addition, subtraction, multiplication and division, on both positive and negative integers using the STFRe{-1,0,1}.

3.1.1 Addition & Subtraction

Most of the new algorithms discussed in this Chapter can be developed either by resembling the conventional arithmetic methods used in binary and decimal systems, or by improving the arithmetic ideas applied to the UBFR(min) in [14].

Among the four major arithmetic operations, addition and subtraction are the simplest two. Therefore, let's start the discussion about addition first. Analogous to the binary system, in the STFRe{-1,0,1}, the sum of two given integers is calculated by adding each pair of ternary coefficients on the same digit as separated numbers. Thus, the first step gives an initial sum with coefficients on each digit being $d_{2i}^e \in \{\pm 2, \pm 1, 0\}$, where each coefficient corresponds to the positive even-subscripted Fibonacci numbers F_{2i}.

It's obvious that the possible coefficient ± 2 are out-of-code in the STFRe{-1,0,1} and we need to use some methods to eliminate them.

Then here comes the very important relation [2]:

$$3F_i = F_{i-2} + F_{i+2} \tag{3.7}$$

After shifting, we can directly get the alternate relation:

$$2F_i = F_{i-2} - F_i + F_{i+2} \tag{3.8}$$

Obviously (3.7) and (3.8) are used to eliminate 3 and 2. And after each item is multiplied by -1 in (3.7) and (3.8), namely $-3F_i = -F_{i-2} - F_{i+2}$ and $-2F_i = -F_{i-2} + F_i - F_{i+2}$, the alternate forms are used to eliminate -3 and -2. Therefore, theses relations are the key to calculations in STFRe{-1,0,1}. Besides, although the illegal coefficients ± 3 will not appear in the initial addition because the given augend and addend are expressed correctly using the STFRe{-1,0,1}, but they may occur after carries to the left or the right.

Based on the fundamental recurrent equation (1.2), below is the process to deduct the relations (3.7) and (3.8):

$$\begin{cases} F_i = F_{i+2} - F_{i+1} & ① \\ F_i = F_{i-2} + F_{i-1} & ② \end{cases}$$

$$① + ② \Rightarrow 2F_i = F_{i+2} - F_{i+1} + F_{i-2} + F_{i-1}$$

$$= F_{i-2} - F_i + F_{i+2} \quad [from\ (3.8)]$$

$$2F_i = F_{i-2} - F_i + F_{i+2} \Leftrightarrow 3F_i = F_{i-2} + F_{i+2}$$

After the initial summation, we need to scan the whole representation from the least significant digit to the most significant one, namely from the right to the left, or from the opposite direction, using the relations (3.7) and (3.8) to eliminate the illegal ± 2 and ± 3. Then we should repeat the same step until every even-subscripted ternary coefficient is a legal one, i.e. in the set {-1,0,1}.

Before the exact calculation, we need to do some preliminary preparations:

1. Given the augend N_1 and the addend N_2, we record their signs and obtain their absolute values $|N_1|$ and $|N_2|$.

2. Using the algorithm in figure 2.1, we get the unique minimal form of the UBFR, i.e. the Zeckendorf representation, for $|N_1|$ and $|N_2|$ respectively. Besides, in this preliminary step, we can also find the possible overflow number and terminate the progress early

3. Based on the relation (2.16), we implement the UBFR(min)-to-STFRe{-1,0,1} conversion.

4. We check the recorded signs of N_1 and N_2, and make a complementation to the corresponding STFRe{-1,0,1} if finding any sign negative.

What's more, there are four points needed to be noticed:

1. Although it's possible that more than one STFRe{-1,0,1}s can represent the same integer N, the STFRe{-1,0,1} converted from the UBFR(min) of N is unique. For example, the integer 22 can be either expressed as $22 = F_8 + F_2$ or as $22 = F_{10} - F_8 - F_6 - F_4 - F_2$ in the STFRe{-1,0,1}. But it only could be $22 = F_8 + F_2$ after the UBRF(min)-to-STFRe{-1,0,1} conversion. This property sometimes makes arithmetic operations convenient.

2. In terms of what we mentioned above, we can sweep over the whole representation of the initial addition either from the least significant digit to the most significant one, or from the opposite direction. However, it's better to adopt the former scan direction. The reason is that sometimes we can omit carries on the most significant digit by the right-to-left direction, which is convenient for calculations, especially for multiplication (discussed later). For example, if we are required to do the addition and have the initial sum as [1 -1 0 0] + [1 -1 0 0] = [2 -2 0 0], where the four coefficients correspond to the weights F_8, F_6, F_4 and F_2, then we need to correct the initial expression because of the existence of non-ternary coefficients. If scanning from left to right, we have the corrected form as [1 -1 -1 0 0]. And if scanning from right to left, the corrected form is as [1 1 -1 0]. It's apparent that, in this example, the former correction needs one more digit and the latter one is still using four digits.

3. No other non-ternary coefficients will occur in the process of carries except ± 2 and ± 3. The reason is that during each round of scanning, we only check and eliminate the rightmost illegal coefficient and this can effectively avoid a "pilling up" of the carries exceeding ± 2 and ± 3. Based on the knowledge discussed above, the maximal integer represented by the 8-bit UBFR(min) is $33_d = [10101010]_{UBFR(min)}$, it corresponds to $33_d = [1111]_{STFRe\{-1,0,1\}}$ in the STFRe{-1,0,1} after the UBFR(min)-to-STFRe{-1,0,1} conversion. Then, if we want to know the addition of [1 1 1 1] + [1 1 1 1], the processing steps are: [1 1 1 1] + [1 1 1 1] =[2 2 2 2] ⇒ [2 2 3 -1] ⇒ [2 3 0 0] ⇒ [3 0 1 0] ⇒ [1 0 1 1 0] = 66_d. Apparently, in this representative example, no illegal coefficients beyond ± 2 and ± 3 occur.

4. Sometimes we need to eliminate the illegal coefficient ± 2 on the least significant digit, namely $t_2^e = 2$ or $t_2^e = -2$. Then here comes a carry toward right on t_0^e, which corresponds to the weight F_0. Usually we don't care about the coefficient t_0^e, since the corresponding F_0 is equal to 0. And we don't show the digit t_0^e in the STFRe{-1,0,1}. Besides, the fourth point indicates why we use the STFRe{-1,0,1} instead of the STFRo{-1,0,1}. Using the STFRo{-1,0,1}, if we have the illegal coefficient ± 2 on the rightmost digit, namely $t_1^o = 2$ or $t_1^o = -2$, then we will have a carry on t_{-1}^o, which corresponds to the weight F_{-1}. Unlike t_0^e, we can not omit t_{-1}^o, because F_{-1} is non-zero. This may result in more negative-subscripted Fibonacci numbers included in further consecutive additions, inconvenient to be implemented on a computer.

To better illustrate the whole process of addition, figure 3.1 shows two typical examples using the STFRe{-1,0,1}.

$$Given\ augend\ N_1 = 15, addend\ N_2 = -6$$

$\lvert N_1 \rvert (UBFR(min))$	0 1 0 0 0 1 0 0	= 15
$\lvert N_2 \rvert (UBFR(min))$	0 0 0 1 0 0 1 0	= 6
$\lvert N_1 \rvert (STFRe\{-1,0,1\})$	1 -1 1 -1	= 15
$\lvert N_2 \rvert (STFRe\{-1,0,1\})$	0 1 -1 1	= 6
$N_1 (STFRe\{-1,0,1\})$	1 -1 1 -1	= 15
$N_2 (STFRe\{-1,0,1\})$	0 -1 1 -1	= -6
$initial\ sum$	1 -2 2 -2	= 9
$carries\ \&\ remove\ t_0^e$	1 -2 1 1	= 9
$carries$	0 1 0 1	= 9
$result(STFRe\{-1,0,1\})$	0 1 0 1	= 9

$$Given\ augend\ N_1 = -12, addend\ N_2 = -10$$

$\lvert N_1 \rvert (UBFR(min))$	0 0 1 0 1 0 1 0	= 12
$\lvert N_2 \rvert (UBFR(min))$	0 0 1 0 0 1 0 0	= 10
$\lvert N_1 \rvert (STFRe\{-1,0,1\})$	0 1 1 1	= 12
$\lvert N_2 \rvert (STFRe\{-1,0,1\})$	0 1 1 -1	= 10
$N_1 (STFRe\{-1,0,1\})$	0 -1 -1 -1	= -12
$N_2 (STFRe\{-1,0,1\})$	0 -1 -1 1	= -10
$initial\ sum$	0 -2 -2 0	= -22
$carries$	0 -3 1 -1	= -22
$carries$	-1 0 0 -1	= -22
$result(STFRe\{-1,0,1\})$	-1 0 0 -1	= -22

Figure 3.1 Two examples of addition using the STFRe{-1,0,1}

These two examples show the addition of 15 + (-6) = [1 -1 1 -1] + [0 -1 1 -1] = [0 1 0 1] = 9 and (-12) + (-10) = [0 -1 -1 -1] + [0 -1 -1 1] = [-1 0 0 -1] = -22 in detail. The former result is positive and the latter is negative. We find that the sign of the integer

can be determined by the most significant non-zero digit. In the examples, the operation's meaning is shown to the left of each line, and the Fibonacci representation resulted from the corresponding operation is displayed in the middle of the line. In order to demonstrate that the conversions and the corrections will not change the value of each integer, we also show in figure 3.1 the decimal value to the right of the line for check.

Then here comes the discussion about subtraction. In the STFRe{-1,0,1}, the difference of two given integers is calculated by a digit-wise subtraction as separate numbers. The first step gives an initial difference with coefficients $d_{2i}^{e} \in \{\pm 2, \pm 1, 0\}$ on each digit, where d_{2i}^{e} corresponds to the weight F_{2i}. It's obvious that the illegal coefficients to be corrected are still ± 2 and ± 3 (occurring in carries), which are the same as the ones in addition. Therefore, addition and subtraction are the same in the STFRe{-1,0,1} except that we change a plus sign to a minus sign.

This is a great advantage compared with the operations using the UBFR(min). When solving arithmetic operations in the UBFR(min), only 0 and 1 are legal bits and we need to use different complicated methods to treat addition and subtraction respectively. In Zeckendorf addition, we need to care about removal of the illegal coefficient 2, removal of the adjacent 1s and replacement of the non-zero coefficient on the least significant digit. In Zeckendorf subtraction, we need to remove the invalid digit -1, which is more difficult to eliminate than 2 in Zeckendorf addition. What's worse, both two operations can hardly process the negative integers. However, we can apply one simple algorithm to both addition and subtraction using the STFRe{-1,0,1}. The following figure 3.2 is an example to illustrate the whole process of subtraction using the STFRe{-1,0,1}.

$$\boldsymbol{Given\ subtrahend\ N_1 = 27, minuend\ N_2 = 33}$$

$\lvert N_1 \rvert (UBFR(min))$	1 0 0 1 0 0 1 0	= 27
$\lvert N_2 \rvert (UBFR(min))$	1 0 1 0 1 0 1 0	= 33
$\lvert N_1 \rvert (STFRe\{-1,0,1\})$	1 1 -1 1	= 27
$\lvert N_2 \rvert (STFRe\{-1,0,1\})$	1 1 1 1	= 33
$N_1 (STFRe\{-1,0,1\})$	1 1 -1 1	= 27
$N_2 (STFRe\{-1,0,1\})$	1 1 1 1	= 33
$initial\ difference$	0 0 -2 0	= -6
$carries$	0 -1 1 -1	= -6
$result(STFRe\{-1,0,1\})$	0 -1 1 -1	= -6

Figure 3.2 An example of subtraction using the STFRe{-1,0,1}

It's obvious that subtraction is actually the same as addition in the signed even-subscripted ternary Fibonacci system. And the simple method can process the negative integers easily just by making complementation on each digit. Thus it's more convenient using the STFRe{-1,0,1} than the UBFR(min).

Finally, we have to admit that compared with the binary addition/subtraction (or other conventional polynomial number system like decimal, ternary), the Fibonacci addition/subtraction seem much more complicated. The reason is that in the conventional number system, a single carry only propagates to more significant digits. We can manage carries only in a single pass, if we start the addition/subtraction from the least significant digit to the most significant one. But the propagations in the Fibonacci system have both left and right directions and we need several passes to eventually get a correct representation.

3.1.2 Multiplication

Let's first talk about multiplication using the STFRe{-1,0,1}. By resembling the idea of conventional arithmetic methods, multiplication using the STFR{-1,0,1} will be developed by the sum of suitable multiples of the multiplier, which are selected in terms of the STFRe{-1,0,1} of the multiplicand.

At the beginning of this Chapter, we have introduced the expression of an integer N as a scalar product $N = \mathbf{D} \cdot \mathbf{W}$, where **D** is a digit vector and **W** is a weight vector corresponding to **D**. Different base number systems have different **W**s and restrictions on the digits in **D**s. And we mentioned that the scalar product of the integer N in the STFRe{-1,0,1} can be converted from the UBFR(min), namely

$$N = [t_{2k}^e, t_{2k-2}^e, \dots, t_4^e, t_2^e] \cdot [F_{2k}, F_{2k-2}, \dots, F_4, F_2]^T$$

this is in the form (3.6), which is very useful in the implementation of Fibonacci arithmetic operations using the STFRe{-1,0,1}, especially multiplication. Therefore, building on the representation in (3.6), we now explain in detail the whole process of multiplication using the STFRe{-1,0,1} by referring to the idea of conventional multiplication.

Given the multiplicand X, and the multiplier Y, both of which are integers, we wish to calculate their product Z, i.e. $Z = X \times Y$. According to (3.1), we first represent X as the scalar product, namely $X = X \cdot W$. Then based on (3.6) and (2.17), we have $X \cdot W = [t_{2k}^e, t_{2k-2}^e, \dots, t_4^e, t_2^e] \cdot [F_{2k}, F_{2k-2}, \dots, F_4, F_2]^T = \sum_{i=1}^{k} t_{2i}^e F_{2i}$ in the STFRe{-1,0,1}. It's obvious that the scalar product of X is just another form of X's STFRe{-1,0,1}. Now the product Z can be expressed as $Z = X \cdot W \times Y = (\sum_{i=1}^{k} t_{2i}^e F_{2i}) \times Y = \sum_{i=1}^{k} t_{2i}^e \cdot (F_{2i} \cdot Y)$.

The preceding step demonstrates that the product Z is the addition of multiples of the multiplier Y, i.e. $F_{2i} \cdot Y$, multiplied by the multiplicand X's ternary coefficients t_{2i}^e. Therefore, the most important step in Fibonacci multiplication is to obtain the multiples $F_{2i} \cdot Y$. Unlike multiplication in the conventional binary system, where we can make multiples just by "left shifting", the scaling in Fibonacci multiplication should be done by analogy with the recurrence rule for generating the Fibonacci numbers in (1.2), $F_i = F_{i-1} + F_{i-2};\ F_0 = 0, F_1 = 1$.

Let's use the symbol M_i to represent the Fibonacci multiples $F_i \cdot \mathrm{Y}$. Then based on the recurrence relation (1.2), we can generate the M_i as follows:

$$M_1 = F_1 \cdot Y_{STFRe\{-1,0,1\}} = Y_{STFRe\{-1,0,1\}}$$
$$M_2 = F_2 \cdot Y_{STFRe\{-1,0,1\}} = Y_{STFRe\{-1,0,1\}}$$
$$M_3 = F_3 \cdot Y_{STFRe\{-1,0,1\}} = (F_1 + F_2) \cdot Y_{STFRe\{-1,0,1\}} = M_1 + M_2$$
$$\cdots$$
$$M_i = M_{i-2} + M_{i-1}$$
$$\cdots$$
$$M_k = M_{k-2} + M_{k-1}$$

where k is the subscript of the most significant digit in $X_{STFRe\{-1,0,1\}}$. What's more, there are three points that should be noticed:

1. Since we perform Fibonacci multiplication using the STFRe{-1,0,1}, we can directly omit the Fibonacci multiples with odd subscripts when doing the subsequent addition.

2. We can use the subscript k' of the most significant non-zero digit in $X_{STFRe\{-1,0,1\}}$, instead of the subscript k of the most significant digit in $X_{STFRe\{-1,0,1\}}$, to determine when we can stop the generation of M_i. Namely we can stop generating the Fibonacci multiples M_i after we achieve $M_{k'}$ rather M_k mentioned before. Since we have $k' < k$, this improvement will save many unnecessary additions.

3. In order to conveniently test this algorithm on computers, we first calculate the Fibonacci multiplication of $|X|$ and $|Y|$, and then determine the sign. This will make additions and prevention of overflows (discussed in the next section) much easier.

Therefore, after we calculate the Fibonacci multiples from M_1 to $M_{k'}$, we add the even-subscripted ones weighted by the corresponding ternary coefficients of $X_{STFRe\{-1,0,1\}}$. What's more, we find all the arithmetic steps mentioned above are done by using Fibonacci addition discussed in the preceding subsection.

To better illustrate the whole process of multiplication in the STFRe{-1,0,1}, we provide an example in figure 3.3.

$$\textbf{Given multiplicand } N_1 = 9, \textbf{multiplier } N_2 = -9$$

$\lvert N_1 \rvert (UBFR(min))$	1 0 0 0 1 0	= 9
$\lvert N_2 \rvert (UBFR(min))$	1 0 0 0 1 0	= 9
$\lvert N_1 \rvert (STFRe\{\text{-1,0,1}\})$	1 0 1	= 9
$\lvert N_2 \rvert (STFRe\{\text{-1,0,1}\})$	1 0 1	= 9
$N_1 (STFRe\{\text{-1,0,1}\})$	1 0 1	= 9
$N_2 (STFRe\{\text{-1,0,1}\})$	-1 0 -1	= -9
Make Fibonacci Multiples of The Multiplier N_2		
$F_1 \cdot N_2$ (omit)	-1 0 -1	= -9
$F_2 \cdot N_2$	-1 0 -1	= -9
$F_3 \cdot N_2$ (omit)	-2 0 -2	= -18
carries & remove t_0^e	-1 0 1 0	= -18
$F_4 \cdot N_2$	-1 -1 1 -1	= -27
$F_5 \cdot N_2$ (omit)	-2 -1 2 -1	= -45
carries	-1 0 1 1 -1	= -45
$F_6 \cdot N_2$	-1 -1 0 2 -2	= -72
carries & remove t_0^e	-1 -1 0 1 1	= -72
Accumulate Multiples Weighted by t_{2i}^e in $N_1(STFRe\{\text{-1,0,1}\})$		
add $1 \cdot F_2 \cdot N_2$	0 0 -1 0 -1	= -9
add $1 \cdot F_6 \cdot N_2$	+ -1 -1 -1 1 0	= -81
result$(STFRe\{-1,0,1\})$	-1 -1 -1 1 0	= -81

Figure 3.3 An example of multiplication using the STFRe{-1,0,1}

Based on the example in figure 3.3, we find the procedure of Fibonacci multiplication can be divided into two main parts: the first part is to make possibly needed Fibonacci multiples of the multiplier N_2; the second part is to accumulate appropriate multiples weighted by the corresponding even-subscripted ternary coefficients t_{2i}^e in the multiplicand N_1(STFRe{-1,0,1}). It should be noted that we can exchange the roles of these two factors to get the same answer.

Actually, the Fibonacci multiplication using STFRe{-1,0,1} is composed by multiple Fibonacci additions, which make it much more inconvenient than multiples in the conventional polynomial number systems like binary, decimal. However, compared to Zeckendorf multiplication, it is much easier to process (because Zeckendorf addition is more inconvenient than addition using STFRe{-1,0,1}) and can manage the negative integers conveniently.

3.1.3 Division

Here comes the discussion about Fibonacci division using the STFRe{-1,0,1}. Since we are talking about integer division, given the dividend D_1 and the divisor D_2, there are mainly three conditions:

1. $|D_1| < |D_2|$. In this condition, we can directly obtain the quotient $Q = 0$ and the absolute value of the remainder $|R| = |D_1|$. And the sign of R is determined by the signs of both D_1 and D_2.
2. $D_2 = 0$. Since the divisor can't be 0, we will receive a "wrong divisor" warning and the procedure is terminated.
3. $|D_1| \geq |D_2|$ and $D_2 \neq 0$. It's a normal case and we mainly focus on it in this subsection.

The case 1 and the case 3 can be combined as $D_2 \neq 0$. What's more, unlike Fibonacci multiplication, there is no worry about overflow in the quotient Q, because we always have $|Q| \leq |D_1|$ in the integer division.

As a reverse of Fibonacci multiplication, which is done by continuous Fibonacci additions, the procedure of Fibonacci Division mainly consists of a sequence of Fibonacci subtractions. We perform Fibonacci division by analogy with the conventional long division, except that we use the Fibonacci multiples mentioned in Fibonacci multiplication rather than the normal multiples.

We keep making Fibonacci multiples of the absolute value of the divisor $|D_2|$ just as what we do in Fibonacci multiplication

$$M_1 = F_1 \cdot |D_2|_{STFRe\{-1,0,1\}} = |D_2|_{STFRe\{-1,0,1\}}$$
$$M_2 = F_2 \cdot |D_2|_{STFRe\{-1,0,1\}} = |D_2|_{STFRe\{-1,0,1\}}$$
$$M_3 = F_3 \cdot |D_2|_{STFRe\{-1,0,1\}} = M_1 + M_2$$
$$\dots$$
$$M_i = M_{i-2} + M_{i-1}$$
$$\dots$$
$$M_k = M_{k-2} + M_{k-1}$$

until we find the largest Fibonacci multiple M_k not exceeding the dividend D_1. Then we need to use the UBFR(min) as an intermediate step, creating an auxiliary vector B with size of k to store the bits of the quotient. Since the difference between $|D_1|$ and M_k is proved as non-negative, we set the bit B_k of B equal to 1. And because of the property (2.10), namely $a_{i+1}a_i = 0, i = 2, 3, \dots, n-1$, of the UBFR(min), we can directly omit the multiple M_{k-1} and set the corresponding bit B_{k-1} as 0. Then we

start the procedure of trail subtractions with the residue $r = |D_1| - M_k$ and the presently largest multiple M_{k-2}.

We subtract M_{k-2} from r. If the difference is negative, we set B_{k-2} as 0 and move to the next decreasing multiple M_{k-3} for comparison. If the difference is zero, we set B_{k-2} as 1 and all the rest bits, namely $B_{i<k-2}$, as 0. If the difference is positive, we set B_{k-2} as 1 & B_{k-3} as 0, $r = r - M_{k-2}$, and move to the following decreasing multiple M_{k-4}. We repeat the above steps until every bit is filled in B.

After getting the UBFR(min) of the quotient, we convert it to the corresponding STFRe{-1,0,1}. What's more, we need to save the value of the last non-negative residue r, because it's the absolute value of the remainder, namely $r = |R|$. And R's sign is determined by the signs of both D_1 and D_2.

During these steps, we have two points deserving of be mention:

1. We use $|D_1|$ and $|D_2|$ during the whole procedure of division, and determine the sign at the end of the calculation. The reason is that directly using D_1 and D_2 will result in a wrong quotient. For example, given $D_1 < 0$, $D_2 > 0$ and $|D_1| > |D_2|$, if we don't use the absolute values, the quotient would be 0 since every multiple is larger than the divisor.

2. We can't directly calculate the quotient in the STFRe{-1,0,1}. Since we have the ternary coefficient -1 in the STFRe{-1,0,1} and it may appear in the representation of a positive integer, we can't directly determine whether a digit is -1 in the STFRe{-1,0,1} of the quotient by trial subtractions.

Finally, to better elaborate the whole procedure of Fibonacci division using the STFRe{-1,0,1}, we display an example in figure 3.4.

$$Given\ dividend\ D_1 = -30, divisor\ D_2 = 7; supposed\ quotient\ Q, remainder\ R$$

$\lvert D_1 \rvert (UBFR(min))$	1 0 1 0 0 0 1 0	= 30
$\lvert D_2 \rvert (UBFR(min))$	0 0 0 1 0 1 0 0	= 7
$\lvert D_1 \rvert (STFRe\{-1,0,1\})$	1 1 0 1	= 30
$\lvert D_2 \rvert (STFRe\{-1,0,1\})$	0 1 0 -1	= 7
$Make\ Fibonacci\ Multiples\ of\ The\ Divisor\ D_2$		
$F_1 \cdot \lvert D_2 \rvert$	0 1 0 -1	= 7
$F_2 \cdot \lvert D_2 \rvert$	0 1 0 -1	= 7
$F_3 \cdot \lvert D_2 \rvert$	0 2 0 -2	= 14
$carries\ \&\ remove\ t_0^e$	1 -1 0 1	= 14
$F_4 \cdot \lvert D_2 \rvert$	1 0 0 0	= 21
$F_5 \cdot \lvert D_2 \rvert$	2 -1 0 1	= 35
$carries$	1 -1 0 0 1	= 35
$F_5 \cdot \lvert D_2 \rvert > \lvert D_1 \rvert, stop\ making\ multiples$		
$Successive\ Subtractions$		
$F_4\ residue$	0 1 0 1	= 9
$F_2\ residue$	0 0 0 2	= 2
$carries\ \&\ remove\ t_0^e$	0 0 1 -1	= 2
$\lvert Q \rvert (UBFR(min))$	1 0 1 0	= 4
$\lvert Q \rvert (STFRe\{-1,0,1\})$	0 0 1 1	= 4
$\lvert R \rvert (STFRe\{-1,0,1\})$	0 0 1 -1	= 2
$Q(STFRe\{-1,0,1\})$	-1 0 -1 0	= -4
$R(STFRe\{-1,0,1\})$	0 0 -1 1	= -2

Figure 3.4 An example of division using the STFRe{-1,0,1}

The example in figure 3.4 shows the procedure of Fibonacci division consists of the two main parts: the first part is to make possibly needed Fibonacci multiples of the divisor D_2; the second part is to perform a sequence of trail subtractions. We find

Fibonacci division is easier to implement than Fibonacci multiplication, but we need to care about the signs of the quotient and the remainder.

3.2 Problems Handling

When we perform the arithmetic operations in the Fibonacci system, there are two most important problems to be dealt with:

1. How to manage the representations and the operations of the negative integers. Since we only use the Fibonacci numbers with non-negative subscripts as weights, whose values are always non-negative, it makes the problem of handling the negative integers difficult to solve.
2. How to manage the overflow problems. We find almost no materials talking about solutions to overflows during the procedure of the Fibonacci arithmetic operations.

In the n-bit unsigned binary Fibonacci system using the minimal form, the maximal integer is $F_n + F_{n-2} + F_{n-4} + \cdots = F_{n+1} - 1$, represented as [1 0 1 0 ...] Therefore, the value range expressed by the n-bit UBFR(min) is $[0, F_{n+1} - 1]$. And if there is a carry 1 propagated to the (n+1)th bit during an arithmetic operation, then the result must be overflow. It seems that the overflow problem is easy to handle in the UBFR(min). However, since we can't express the negative integers using the UBFR(min), we need to convert it to the STFRe{-1,0,1}, which is emphasized in the preceding Chapter. But checking whether an intermediate result or a final answer is overflow based on the STFRe{-1,0,1} is trickier than the UBFR(min).

In this section, we mainly discuss the advantages of using the STFRe{-1,0,1} to include the negative parts as well as clever ways to check overflows in the STFR{-1,0,1}.

3.2.1 Complementation

Including the negative integers in a Fibonacci system is important for further practical implementation in Fibonacci computers.

In this subsection, we are going to introduce a supplementary representation called "M complemented representation" [7] for the binary Fibonacci Representation, namely BFR, as an example, which allows the negative integers to be included in the BFR. And we are going to compare it with the complementation used in the STFRe{-1,0,1}.

In the "M complemented representation", "M" means the maximal number represented by the general form of the (n-1)-bit UBFR excluding a_1, namely

$$\mathrm{M} = \sum_{i=2}^{n} F_i = F_{n+2} - 2 \rightarrow \underbrace{111\ldots1}_{n-1}$$

starting from the a_2 on the rightmost. The reason for excluding a_1 is to further use this M complemented representation on the UBFR(min).

Based on the figure 2.2 showing the flowchart to obtain a general UBFR, given a non-negative integer N ≤ M, we find the following relationship between the representation of N and M-N:

$$\mathrm{N} \rightarrow (a_n a_{n-1} \dots a_2) \tag{3.9a}$$

$$\mathrm{M} - \mathrm{N} \rightarrow (a_n{}' a_{n-1}{}' \dots a_2{}') \tag{3.9b}$$

where

$$a_i \oplus a_i' = 1, \quad i = 2, 3, \dots, n \tag{3.9c}$$

namely

$$(a_n a_{n-1} \dots a_2) \oplus (a_n{}' a_{n-1}{}' \dots a_2{}') = (111 \dots 1) \rightarrow M \tag{3.9d}$$

Therefore, N-M is the M's complement of N. The sign $\oplus$ is referred from [7], meaning Boolean operator xor.

Based on this "M complemented" idea, we can expand the range into the negative integers. Therefore, given an arbitrary integer N, with $|N| \leq M$, we can obtain its M complemented (n-1)-bit BFR excluding a_1 as follows [7]:

1. Use the algorithm in figure 2.2 to get the general (n-1)-bit UBFR excluding a_1 for $|N|$ as $(a_n a_{n-1} \dots a_2)$.

2. Add one more bit as a sign bit a_s to the leftmost, i.e. $(a_s a_n a_{n-1} \dots a_2)$ and initially set a_s equal to 0, i.e. $(0 a_n a_{n-1} \dots a_2)$.

3. If N is non-negative, $(0 a_n a_{n-1} \dots a_2)$ is its M complemented (n-1)-bit BFR.

4. If N is negative, make 1's complementation on every bit on $(0 a_n a_{n-1} \dots a_2)$ and the corresponding $(1 a_n{}' a_{n-1}{}' \dots a_2{}')$ is N's M complemented (n-1)-bit BFR.

Based on [7], we can further improve the above methods as M complemented (n-1)-bit BFR(min), because the minimal form of BFR can reduce possible carry-operations and therefore speed up the procedure of Fibonacci arithmetic algorithms. Then we need to change the step 1 and the step 4 in the original method:

1'. Use the algorithm in figure 2.1 to get the general (n-1)-bit UBFR(min) for $|N|$ as $(a_n a_{n-1} \dots a_2)$.

4'. If N is negative, we first convert $(a_n a_{n-1} \dots a_2)$ to the maximal form by conversion from "100" to "011", then add 0 to the leftmost, and finally do 1' complementation.

In M complemented (n-1)-bit BFR(min), the integer's sign is determined by the sign bit a_s on the most significant digit (leftmost). If a_s = 1, N is negative; otherwise, N is non-negative. And we use the least number of 1s for faster implementation. Here, we set n = 8 and display two examples as below:

1. If N = 24, then its M complemented 7-bit BFR(min) is just its UBFR(min) plus one more sign bit a_s = 0, as (01000100).

2. If N = -24, then the procedure is as: $(1000100)_{UBFR(min)} \rightarrow (0101111)_{UBFR(max)} \rightarrow (a_s 0101111), a_s = 0 \rightarrow (a_s 1010000), a_s = 1$. Therefore, we have N = -24 $\rightarrow (11010000)$.

As a comparison, M complemented (n-1)-bit BFR(min) has several disadvantages:

1. It needs one extra bit to indicate the integer's sign.

2. The representation has poor readability. We can't directly get what it expresses, especially for the negative integers.

3. The method is complicated, since it involves the conversion from the minimal to the maximal form as well as 1's complementation.

However, the STFRe{-1,0,1} used in the thesis does not have these disadvantages mentioned above:

1. It does not waste extra coefficients functioning as the sign indicator. We can determine the integer's sign directly by checking the most significant digit of the representation.

2. It has very good readability and every digit in the representation seems meaningful. We get the value just by adding the Fibonacci weights multiplied by the corresponding ternary coefficients. Sometimes, we can immediately know the value by applying the Fibonacci properties to the representation. For example, we know $[1111]_{STFRe\{-1,0,1\}} \rightarrow 33_d$ without actual calculation, just using the property (1.6).

3. It makes the complementation between the positive and the negative integers very convenient to realize, without complicated methods. Just by multiplying every digit -1, we convert N to –N. For example, [1010] in the STFRe{-1,0,1} represents 24 and [-10-10] represents -24.

4. Depending on the above three advantages, we perform Fibonacci arithmetic operations, containing both positive and negative, effectively by using the STFRe{-1,0,1}.

In conclusion, to solve the problem about negative expressions and negative arithmetic operations, the STFRe{-1,0,1} has many more advantages than the BFR, even if the BFR uses some supplementary methods such as M complemented complementation to include the negative integers.

3.2.2 Overflow

Solving the problem of overflow is kind of tricky for the STFRe{-1,0,1}. In terms of (2.8), the maximal integer M that can be represented by the n-bit UBFR(min) is $M = F_{n+1} - 1$. Therefore, the range of the n-bit UBFR(min) is $[0, M]$. After the conversion from the n-bit UBFR(min) to the k-bit STFRe{-1,0,1}, we double the range as $[-M, M]$. Here, the relation between n and k is $k = \left\lceil \frac{n}{2} \right\rceil$, where $\left\lceil \frac{n}{2} \right\rceil$ means the least integer greater than or equal to $\frac{n}{2}$. Based on the relation $t_{2i}^{e} = a_{2i-1} + a_{2i} - a_{2i+1}$ in (2.13), if n is even, the most significant digit is with subscript $k = i_{max} = \frac{n}{2}$; if n is odd, the most significant digit is with subscript $k = i_{max} = \frac{n+1}{2}$. Thus, the converted STFRe{-1,0,1} has size of $k = \left\lceil \frac{n}{2} \right\rceil$.

However, the size of $k = \left\lceil \frac{n}{2} \right\rceil$ is only enough for the integer within range. When the Fibonacci arithmetic operations are implemented on a computer, to check whether an intermediate result overflows and terminate the procedure earlier, we first need to

store the result's STFRe{-1,0,1} and then examine. Therefore, the STFRe{-1,0,1} with size $k = \left\lceil \frac{n}{2} \right\rceil$ is not enough to store the integer that is out of range.

Here is an example: Given $n = 8$, we have $k = 4$ in the converted STFRe{-1,0,1} and the corresponding range that can be expressed by the STFRe{-1,0,1} is $[-33, 33]$. Therefore, $N_1 = N_2 = 33_d = [1111]_{STFRe\{-1,0,1\}}$ are all within range. But their addition is

$$\begin{aligned} N_1 + N_2 &= [1111]_{STFRe\{-1,0,1\}} + [1111]_{STFRe\{-1,0,1\}} \\ &= [10110]_{STFRe\{-1,0,1\}} = 66_d \end{aligned}$$

which overflows and needs an extra digit to store.

In another example, still given $n = 8$, we have $N_1' = N_2' = 13_d = [1\ -1\ 00]_{STFRe\{-1,0,1\}}$ within range. And their addition is

$$\begin{aligned} N_1' + N_2' &= [1\ -1\ 00]_{STFRe\{-1,0,1\}} + [1\ -1\ 00]_{STFRe\{-1,0,1\}} \\ &= [1-1-1\ 00]_{STFRe\{-1,0,1\}} = 26_d \end{aligned}$$

which is not overflowing but still needs one more digit to store.

It should be noticed that the condition that some integer in the STFRe{-1,0,1} is within range but needs one more extra digit only exists during the arithmetic operations. It could not happen in the UBFR(min)-to-STFRe{-1,0,1} conversion based on the relation (2.13). Let's take the preceding example $[1-1-1\ 00]_{STFRe\{-1,0,1\}} = 26_d$, which needs one extra digit but does not overflow. In the unique UBFR(min)-to-STFRe{-1,0,1} conversion, we have $11\ -1\ 0_{STFRe\{-1,0,1\}} = 26_d$. This also demonstrates that we may have several ways to represent the same integer but only get one in the UBFR(min)-to-STFRe{-1,0,1} conversion.

Then here comes a question: Given the k-bit STFRe{-1,0,1} converted from the n-bit UBFR(min), where $k = \left\lceil \frac{n}{2} \right\rceil$, how many extra digits do we need in the STFRe{-1,0,1} to represent all the overflowing or not overflowing integer resulting from the valid arithmetic operations? Since the maximal integer is

$$M = F_n + F_{n-2} + \cdots = F_{n+1} - 1$$

the maximal overflowing integer, resulting from two integers within range, is

$$M + M = 2F_{n+1} - 2$$

If adding one extra digit to the STFRe{-1,0,1}, namely size of $k' = k + 1 = \left\lceil \frac{n}{2} \right\rceil + 1$, the maximal integer is

$$M' = F_{2k+2} + F_{2k} + F_{2k-2} + \cdots + F_2 = F_{2k+3} - 1$$

1. If n is even, $M'_e = F_{n+3} - 1$. Then

$$M'_e - (M + M) = (F_{n+3} - 1) - (2F_{n+1} + 2) = F_{n+2} - F_{n+1} + 1 = F_n + 1$$

2. If n is odd, $M'_o = F_{n+4} - 1$. Then

$$M'_o - (M + M) = (F_{n+4} - 1) - (2F_{n+1} + 2) = F_{n+3} + F_{n+2} - 2F_{n+1} + 1 = F_{n+2} + F_n + 1$$

Since $n_{min} = 2$, the differences $M' - (M + M)$ in both cases are all positive, which indicates that, in the sense of value, we only need one extra digit in the STFRe{-1,0,1} to express all the overflowing or not overflowing integers generated in the arithmetic operations.

What's more, we also wonder whether there would be a carry on the (k+2)th digit, indicating that we need two extra digits rather than on in the STFRe{-1,0,1}. Actually, the answer is no, and the reason will be discussed as below.

Given the n-bit UBFR(min), we are going to discuss in detail about the method to check whether an integer, expressed in the $(\lceil \frac{n}{2} \rceil + 1)$-bit STFRe{-1,0,1}, is out of range. To better illustrate, we divide the overflow problem into two conditions:

1) If n is even:

In this case, the STFRe{-1,0,1} is of size $k_e = \frac{n}{2} + 1$ and we have the maximal non-overflowing integer $M = F_{n+1} - 1$. After the UBFR(min)-to-STFRe{-1,0,1} conversion, the integer's STFRe{-1,0,1} should be in $\frac{n}{2}$ digits. And there may be a carry 1 or -1 in the $(\frac{n}{2} + 1)$th digit during the arithmetic operations but the expressed integer is still within range. Therefore,

I. If $t^e_{2\cdot(\frac{n}{2}+1)} = 0$, it must be within range, since the $|M| = |-M| = F_{2\cdot\frac{n}{2}} + F_{2\cdot(\frac{n}{2}-1)} + \cdots + F_{2\cdot 1} = F_{n+1} - 1$, expressed as $0\underbrace{11\ldots1}_{\frac{n}{2}\,1s}$.

II. If $t^e_{2\cdot(\frac{n}{2}+1)} = 1$, implying the integer must be positive, in order to satisfy the condition that the valid positive integer N is $N < M$, we should have $t^e_{2\cdot\frac{n}{2}} = -1$ and the most significant non-zero digit from $t^e_{2\cdot(\frac{n}{2}-1)}$ to t^e_2 should be -1. Thus, the valid STFRe{-1,0,1} with $t^e_{2\cdot(\frac{n}{2}+1)} = 1$ is as 1-1$\underbrace{0\ldots0}_{all\ 0s}$-1…

III. If $t^e_{2\cdot(\frac{n}{2}+1)} = -1$, implying the integer must be negative, according to the opposite property of 1 and -1, we should have $t^e_{2\cdot\frac{n}{2}} = -1$ and the most significant non-zero digit should be 1. Thus, the valid STFRe{-1,0,1} with $t^e_{2\cdot(\frac{n}{2}+1)} = -1$ is as -11$\underbrace{0\ldots0}_{all\ 0s}$1…

Thus, it's obvious that, during addition, if two positive integers within range but with $t^e_{2\cdot(\frac{n}{2}+1)} = 1$, then the initial sum should be 1-1$\underbrace{0\ldots0}_{all\ 0s}$-1… + 1-1$\underbrace{0\ldots0}_{all\ 0s}$-1… = 2 -2 0/-1/-2… If we eliminate the carries from right to left, the initial sum could not have a

carry 1 to the position $t^e_{2\cdot(\frac{n}{2}+2)}$, which has been emphasized in the preceding section. Since we have simple complementation of 1 <-> -1 and 0 <-> 0 in the STFRe{-1,0,1}, if we use two negative integers instead of two positive integers in addition, there is still no carry -1 to the position $t^e_{2\cdot(\frac{n}{2}+2)}$. Besides, we know that subtraction is actually the same as addition, and multiplication & division are just implementation of multiple additions and subtractions. Then it is apparent that if n is even, there is no need to append a second extra digit as $t^e_{2\cdot(\frac{n}{2}+2)}$.

2) If n is odd,

In this case, the STFRe{-1,0,1} is of size $k_o = \frac{n+1}{2} + 1$ and we have the maximal non-overflowing integer $M = F_{n+1} - 1$. After the UBFR(min)-to-STFRe{-1,0,1} conversion, the integer's STFRe{-1,0,1} should be in $\frac{n+1}{2}$ digits. If $\left|t^e_{2\cdot(\frac{n+1}{2}+1)}\right| = 1$, the corresponding minimal positive integer $N_{min-pos}$ is

$$\begin{aligned} N_{min-pos} &= F_{2\cdot(\frac{n+1}{2}+1)} - F_{2\cdot\left(\frac{n+1}{2}\right)} - F_{2\cdot\left(\frac{n+1}{2}-1\right)} - \cdots - F_2 \\ &= F_{2\cdot(\frac{n+1}{2}+1)} - (F_{2\cdot\left(\frac{n+1}{2}\right)+1} - 1) \\ &= F_{n+1} + 1 \\ &= M + 2 \end{aligned}$$

and we have

$$N_{min-pos} > M$$

Therefore, we have the following conclusion:

I. If $t^e_{2\cdot(\frac{n+1}{2}+1)} = \pm 1$, it must be out of range. This also indicates it is unnecessary to append a second extra digit as $t^e_{2\cdot(\frac{n+1}{2}+2)}$.

Under the premise of $t^e_{2\cdot(\frac{n+1}{2}+1)} = 0$:

II. If $t^e_{2\cdot\frac{n+1}{2}} = 0$, it must be within range, since we have $|M'| = |-M'| = F_{2\cdot(\frac{n+1}{2}-1)} + F_{2\cdot\left(\frac{n+1}{2}-2\right)} + \cdots F_{2\cdot 1} = F_n - 1 < M$.

III. If $t^e_{2\cdot\frac{n+1}{2}} = 1$, implying the integer must be positive and we have weight F_{n+1}, in order to satisfy the condition that the valid positive integer N is $N < M$, we only need to ensure that the most significant non-zero digit from $t^e_{2\cdot(\frac{n+1}{2}-1)}$ to t^e_2 is -1. Thus, the valid STFRe{-1,0,1} with $t^e_{2\cdot\frac{n+1}{2}} = 1$ is as $1\underbrace{0\ldots0}_{all\ 0s}$-1…

IV. If $t^e_{2\cdot\frac{n+1}{2}} = -1$, implying the integer must be negative, according to the opposite property of 1 and -1, we should have the most significant non-zero digit from $t^e_{2\cdot(\frac{n+1}{2}-1)}$ to t^e_2 is 1. Thus, the valid STFRe{-1,0,1} with $t^e_{2\cdot\frac{n+1}{2}} = -1$ is as $-1\underbrace{0\ldots0}_{all\ 0s}1$…

In conclusion, given two n-bit UBFR(min)s, regardless of n being even or odd, we only need ($\left\lceil\frac{n}{2}\right\rceil$+1)-bit STFRe{-1,0,1} to implement all the arithmetic operations as well as check overflow. When there occurs overflow, usually we terminate the procedure and send a warning. But sometimes overflow is a signal of termination of a certain part.

3.3 Logic System Realization

In this section, we are going to display the realization of a Fibonacci logic system. We will perform and test all the practical algorithms for addition, subtraction, multiplication and division based on the STFRe{-1,0,1} on a computer. Although we have explained the principle of the four fundamental Fibonacci arithmetic operations in the 3.2 section, we will emphasize some key points during the concrete implementation in the first subsection. And we will show the executive outcomes in the second subsection.

3.3.1 Some Key Points

1) In the preliminary preparations, we need to use the algorithm in figure 2.1 to convert the absolute values of the two input integers from the decimal form to the STFRe{-1,0,1}. If we find one integer or both integers' absolute values are out of range, then we should terminate the whole procedure and send a warning message about overflow.

2) For convenience, we use the absolute values of the two input integers during the concrete procedure of multiplication and division. The reason is that we only need to consider the case of the positive integers when checking overflow. What's more, in division, we can not directly compare multiples to the dividend with an opposite sign, which will result in wrong results.

3) Since the multiplication is the most complicated one among the four arithmetic operations, we should develop some clever ways to avoid the unnecessary steps as much as possible.

We know the biggest problem in the multiplication is overflow because of a given limited capacity. Discovery of overflow as soon as possible not only reduces unnecessary work, but also prevents the problem of insufficient capacity for storing an integer in the STFRe{-1,0,1}.

During the first part of multiplication, namely making Fibonacci multiples of the multiplier, the most important task is to keep checking whether the multiples overflow. And we know the fact that the first overflowing multiple can be correctly represented in the STFRe{-1,0,1}, since it resulted from two non-overflowing multiples.

Then the overflow problems can be mainly divided into three conditions:

① If the first overflowing multiple is with odd-subscript, namely M_{2k+1}: Check the digits of the dividend's absolute value from left to right starting with the most significant one.

I. Once we find a non-zero digit with subscript larger than 2k+2, the final product must overflow and we terminate the procedure.

II. If the most significant non-zero digit is equal to 2k+2, i.e. t^e_{2k+2} = 1, then we check whether t^e_{2k} = -1. If not, the final product must overflow and we terminate the procedure. If yes, we finish this part and go to the second part of multiplication, with an "overflow" label used in the second multiplication part.

III. If no non-zero digit with even subscript larger than 2k is found, we finish this part and go to the second part of multiplication, with a "non-overflow" label.

② If the first overflowing multiple is with an even subscript, namely M_{2k+2}: Check the digits of the dividend's absolute value from left to right starting with the most significant one.

I. Once we find a non-zero digit with subscript larger than 2k+2, the final product must overflow and we terminate the procedure.

II. If no non-zero digit with subscript larger than 2k+2 is found, we finish this part and go to the second part of multiplication, with an "overflow" label.

③ If every necessary multiple is within range, then we finish this part and go to the second part of multiplication, with a "non-overflow" label.

Then, during the second part of multiplication, namely accumulation of multiples weighted by the digits of the multiplicand, we start accumulation from the currently largest even-subscripted multiple, add it to the accumulated sum with initial value 0, and repeat the following steps:

1. If the label is "overflow", then we check the next largest even-subscripted multiple whose corresponding digit in the multiplicand is non-zero:

1.1. If the corresponding digit is 1 or we can not find any non-zero digit, then the final product must overflow and we terminate. If the corresponding digit is -1, then add that multiple to the accumulated sum and check overflow.

1.2. If the accumulated sum is now within range, then change the label as "non-overflow".

1.3. If the accumulated sum is still out of range, keep the label as "overflow".

1.4. Set the next largest even-subscripted multiple as the currently largest one.

2. If the label is "non-overflow", then we directly add the next largest even-subscripted multiple, weighted by the corresponding digit in the dividend, to the accumulated sum, and check whether the accumulated sum overflows now

2.1 If the accumulated sum is still within range, then keep the label as "non-overflow".

2.2 If the accumulated sum is now out of range, then change the label as "overflow".

2.3 Set the next largest even-subscripted multiple as the currently largest one.

After the repeating of these steps, if the label shows "overflow", then the final product overflows and we terminate. If the label shows "non-overflow", then the final product doesn't overflow and we print it.

3.3.2 Results

Given the n-bit UBFR(min), we first want to show the relationship between n, namely the number of bits allowed, and the maximal integer that can be represented within range.

We have proved that if the range of the n-bit UBFR(min) is $[0, M]$, where M is the valid maximal integer, then the range of the converted STFRe{-1,0,1} becomes $[-M, M]$. Since $|-M| = |M|$, we only need to show the relationship between n and the maximal integer M. What's more, according to the property of the UBFR(min), we should have $n \geq 2$.

To illustrate the relationship, we draw a plot, which is displayed in figure 3.5.

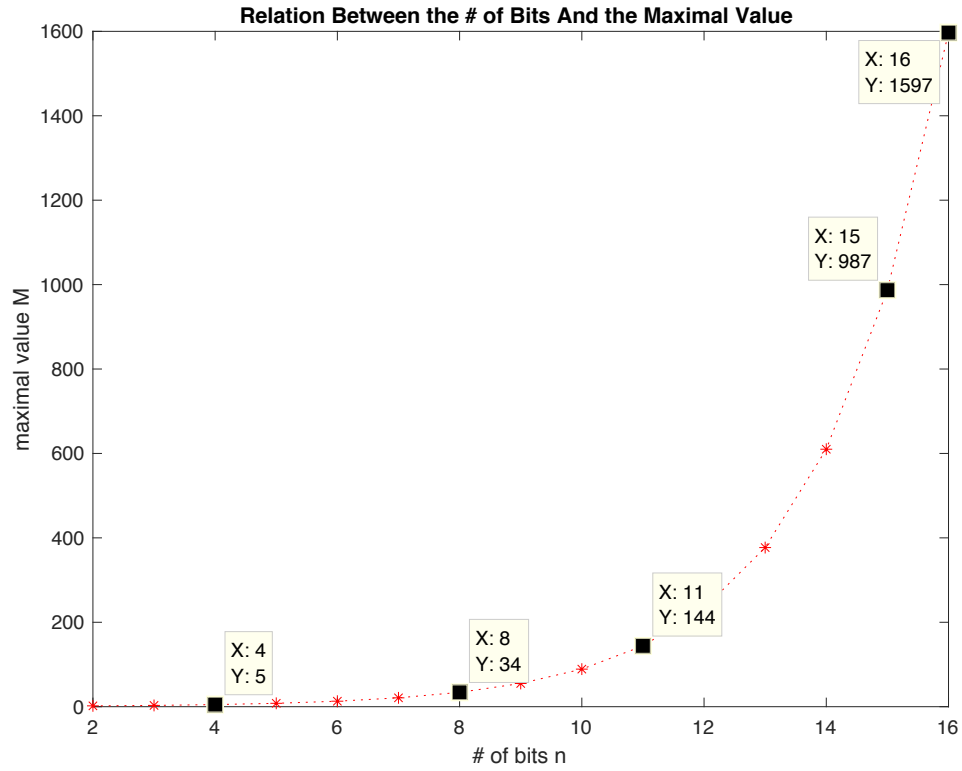

Figure 3.5 Relation between the # of bits n and the maximal value M

In the plot, the x axis is the # of bits, and the y axis is the maximal value corresponding to # of bits. Here, we choose n's range in [2, 16] as an example. It's obvious that as n increases, the maximal value increases. And the rate of increase is faster and faster, since we find the line goes vertically up with n's increment.

And the precise relationship between n and M is

$$M = F_{n+1} - 1$$

where F_{n+1} can be obtained either by the recurrence relation (1.2), or directly by the general term formula (1.3).

Next, we show the flowcharts and some results of the arithmetic operations as examples. The arithmetic operations are written and run on CLion using C++.

1) Addition:

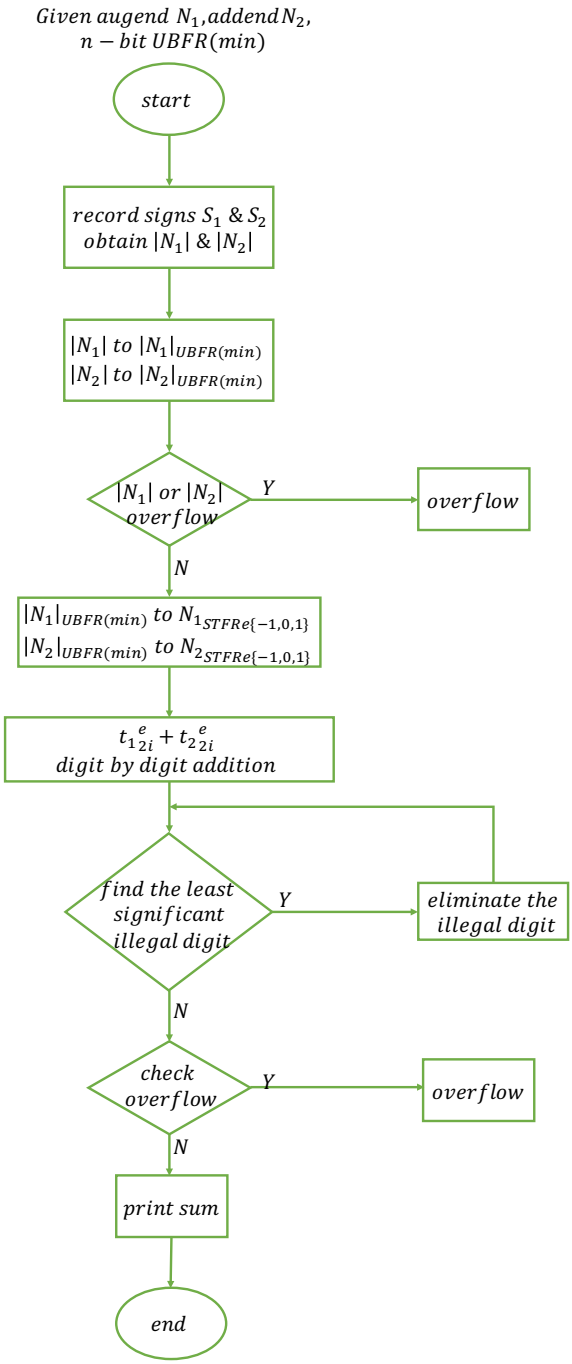

Figure 3.6 Flowchart for the procedure of addition

```
Please input the # of bits logical system you want to create:
8
Please input 2 values to do the operation:
17
11
Please input one operator among '+', '-', '*' and '/':
+
The STFRe expression from the most significant digit to the least significant digit is:
0 1 1 0 -1
The corresponding decimal expression is:
28
```

Figure 3.7(a) An example result of addition

```
Please input the # of bits logical system you want to create:
8
Please input 2 values to do the operation:
-34
8
Please input one operator among '+', '-', '*' and '/':
+
Input1 is overflow!
```

Figure 3.7(b) An example result of addition

```
Please input the # of bits logical system you want to create:
8
Please input 2 values to do the operation:
25
26
Please input one operator among '+', '-', '*' and '/':
+
Overflow! The result exceeds the maximum value!
```

Figure 3.7(c) An example result of addition

Flowchart 3.6 shows the whole procedure of addition. Figures 3.7(a)-3.7(c) show three examples of addition. Given 8-bit UBFR(min), the maximal integer is F_9 - 1 = 33. Therefore, the range of the STFRe{-1,0,1} is [-33, 33]. In figure 3.7(a), the two inputs and the sum $N = 28$ are within range; in figure 3.7(b), the first input overflows and we directly terminate the procedure; in figure 3.7(c), the sum is out of range since it's larger than 33.

2) Subtraction:

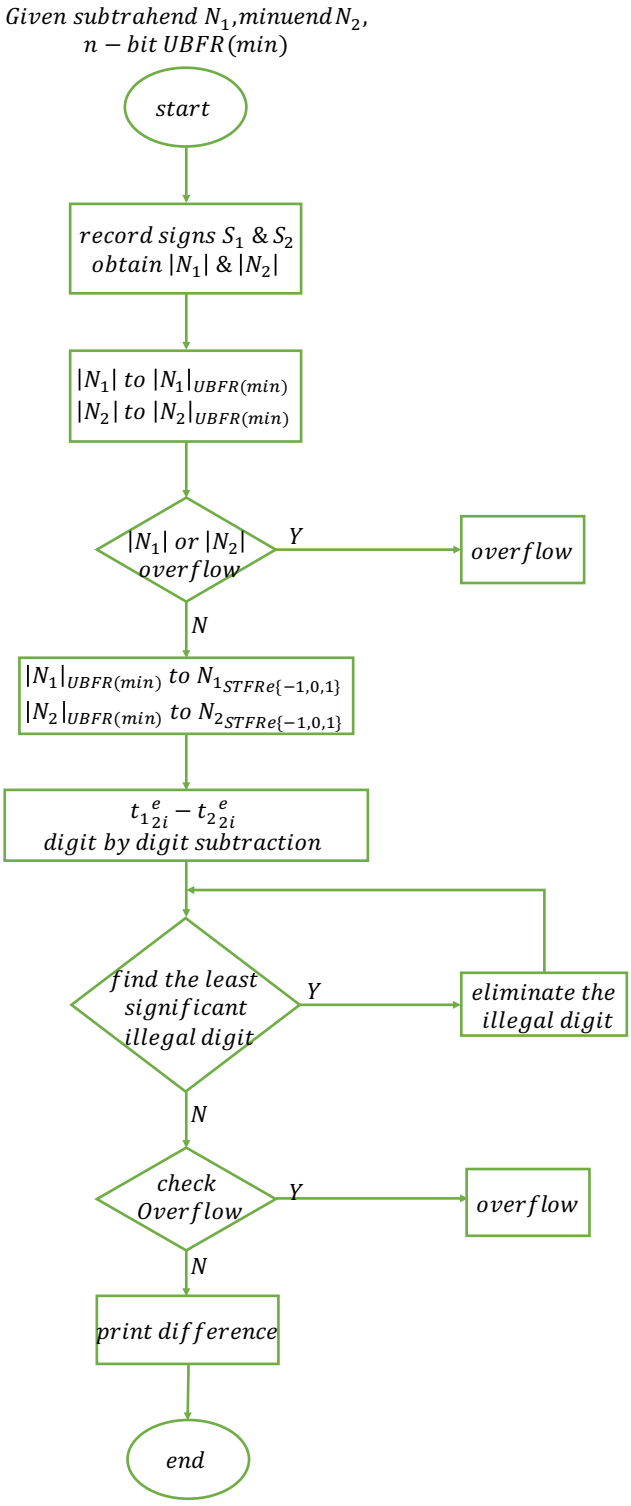

Figure 3.8 Flowchart for the procedure of subtraction

```
Please input the # of bits logical system you want to create:
9
Please input 2 values to do the operation:
7
38
Please input one operator among '+', '-', '*' and '/':
-
The STFRe expression from the most significant digit to the least significant digit is:
0 -1 1 0 1 0
The corresponding decimal expression is:
-31
```

Figure 3.9(a) An example result of subtraction

```
Please input the # of bits logical system you want to create:
9
Please input 2 values to do the operation:
-56
-60
Please input one operator among '+', '-', '*' and '/':
-
Both Inputs are overflow!
```

Figure 3.9(b) An example result of subtraction

```
Please input the # of bits logical system you want to create:
9
Please input 2 values to do the operation:
28
-29
Please input one operator among '+', '-', '*' and '/':
-
Overflow! The result exceeds the maximum value!
```

Figure 3.9(c) An example result of subtraction

Flowchart 3.8 shows the whole procedure of subtraction. Figures 3.9(a)-3.9(c) show three examples of subtraction. Given 9-bit UBFR(min), the maximal integer is F_{10} - 1 = 54. Therefore, the range of the STFRe{-1,0,1} is [-54, 54]. In figure 3.9(a), the two inputs and the difference N = -31 are within range; in figure 3.9(b), both inputs overflow and we directly terminate the procedure; in figure 3.9(c), the difference is out of range since it's larger than 54.

3) Multiplication:

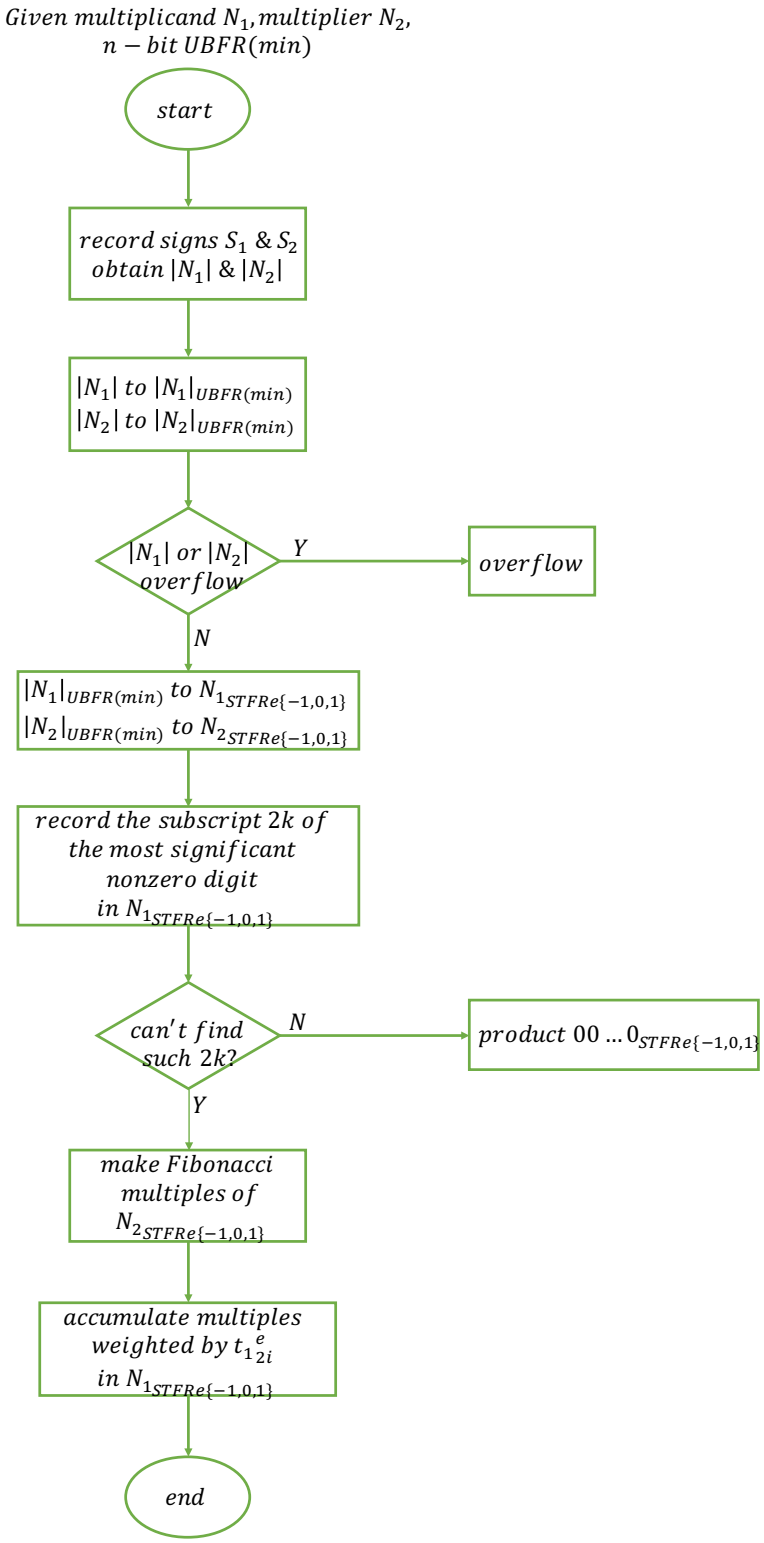

Figure 3.10(a) Flowchart for the procedure of multiplication

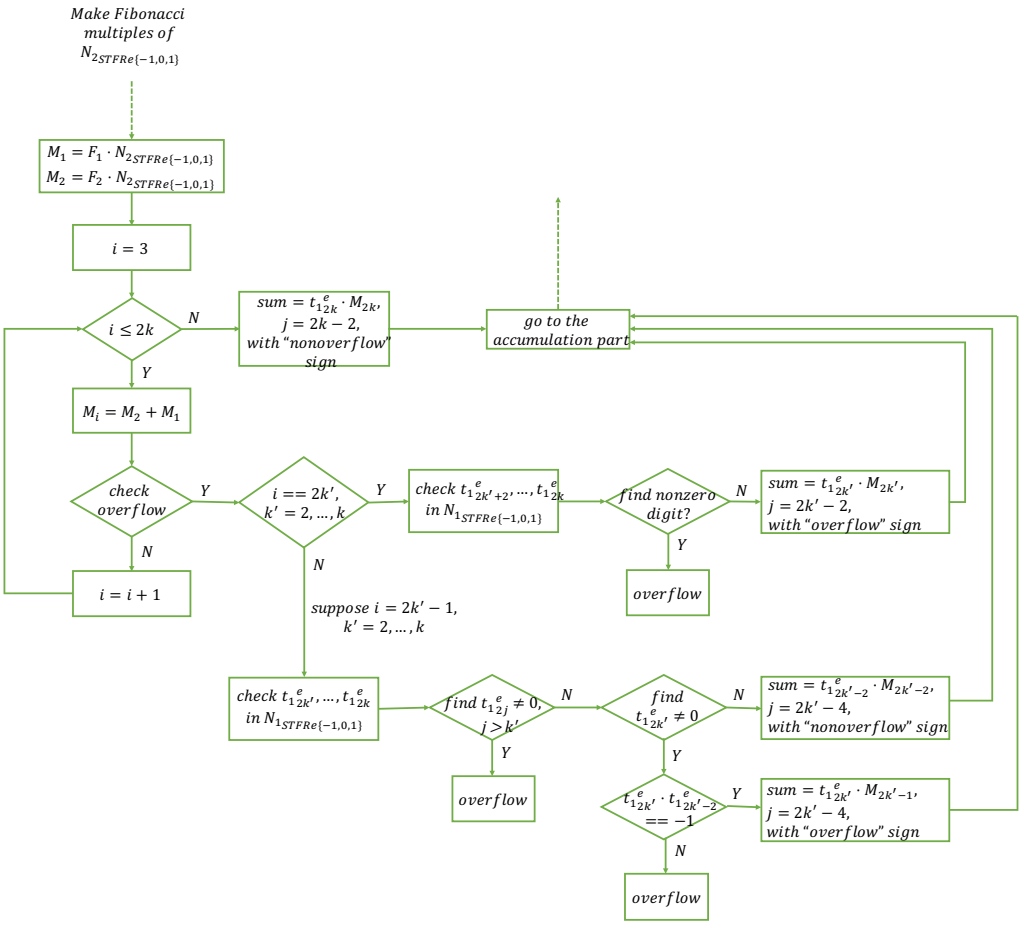

Figure 3.10(b) Flowchart for the details in the first part of multiplication

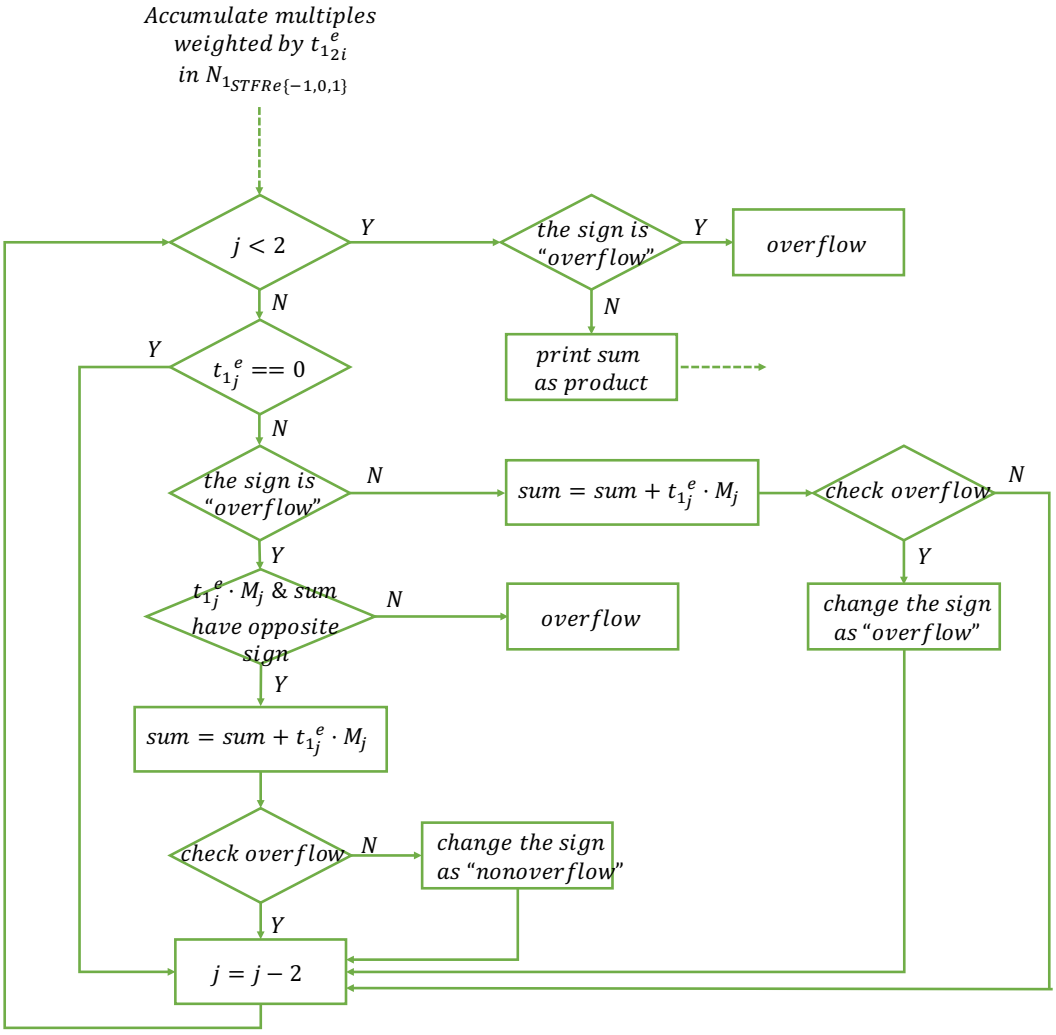

Figure 3.10(c) Flowchart for the details in the second part of multiplication

```
Please input the # of bits logical system you want to create:
10
Please input 2 values to do the operation:
12
-7
Please input one operator among '+', '-', '*' and '/':
*
The STFRe expression from the most significant digit to the least significant digit is:
0 -1 -1 -1 0 0
The corresponding decimal expression is:
-84
```

Figure 3.11(a) An example result of multiplication

```
Please input the # of bits logical system you want to create:
10
Please input 2 values to do the operation:
13
-7
Please input one operator among '+', '-', '*' and '/':
*
Overflow! The result exceeds the minimum value!
```

Figure 3.11(b) An example result of multiplication

Flowchart 3.10(a) shows the whole procedure of multiplication. Flowcharts 3.10(b) and 3.10(c) show details in the first part, namely making Fibonacci multiples of the multiplier, and second part, namely accumulating weighted Fibonacci multiples, of multiplication respectively. Figures 3.11(a) and 3.11(b) show two examples of subtraction. Given 10-bit UBFR(min), the maximal integer is F_{11} - 1 = 88. Therefore, the range of the STFRe{-1,0,1} is [-88, 88]. In figure 3.11(a), the two inputs and the product N = -84 are within range; in figure 3.11(b), the product is out of range since it's less than -88.

4) Division:

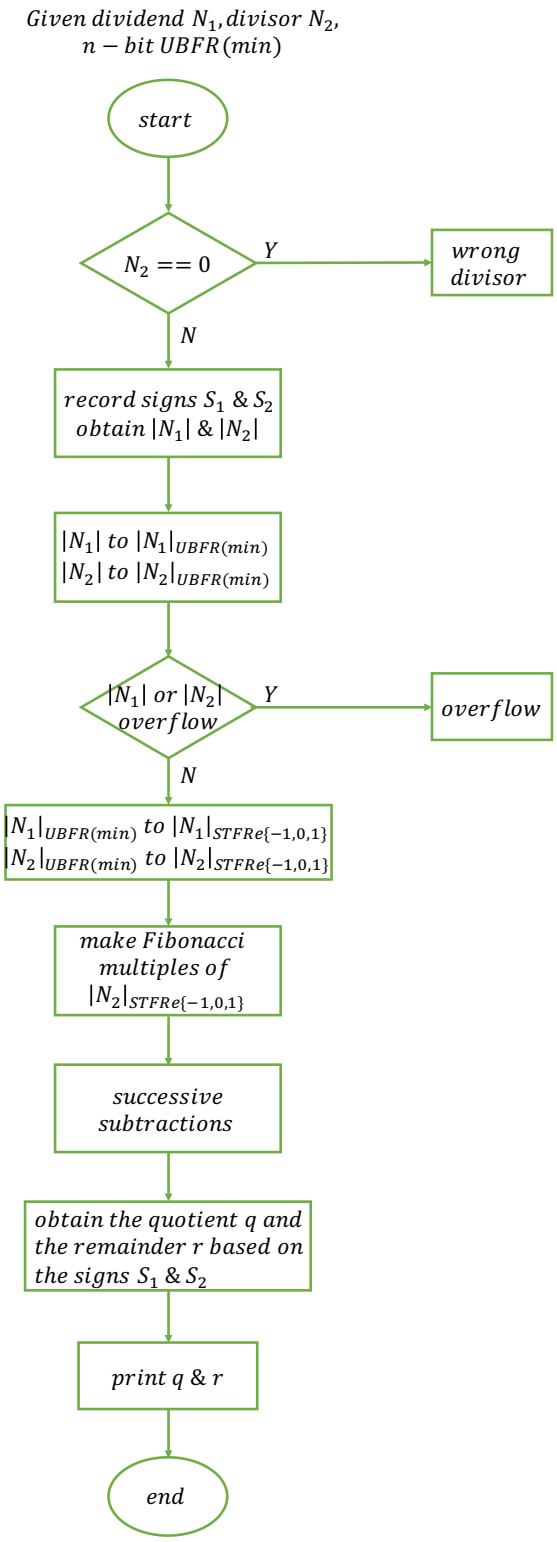

Figure 3.12(a) Flowchart for the procedure of division

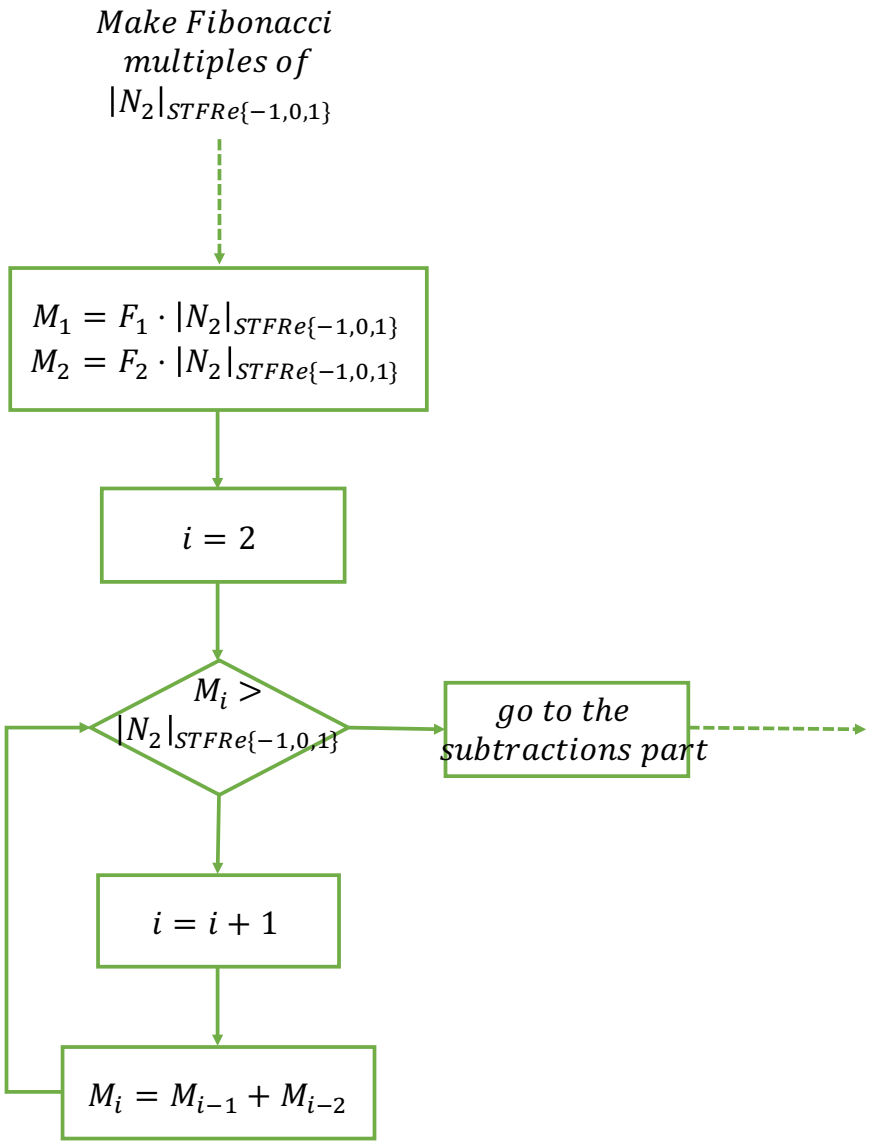

Figure 3.12(b) Flowchart for the details in the first part of division

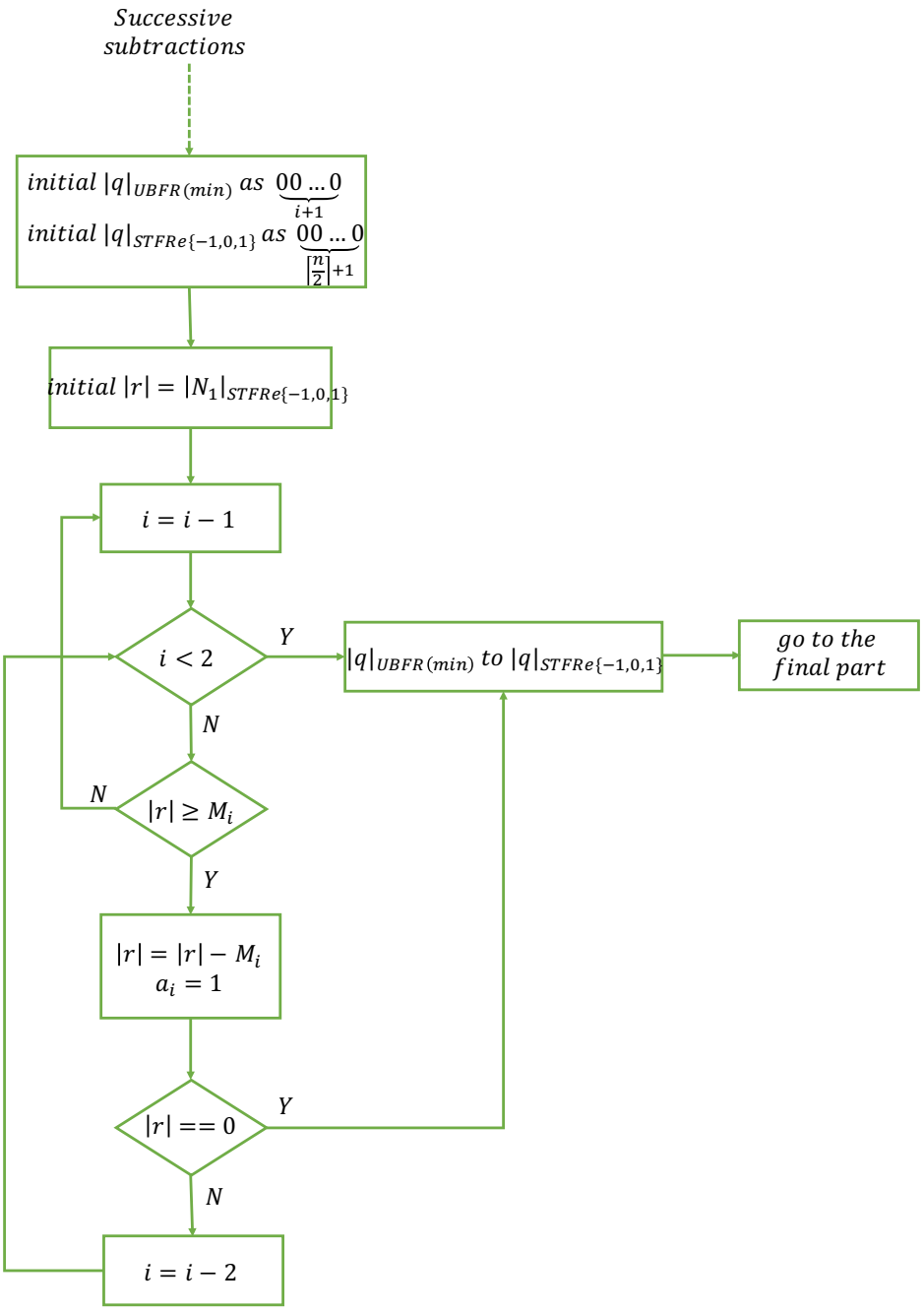

Figure 3.12(c) Flowchart for the details in the second part of division

```
Please input the # of bits logical system you want to create:
11
Please input 2 values to do the operation:
121
-11
Please input one operator among '+', '-', '*' and '/':
/
Here is the quotient of N1/N2:
The STFRe expression from the most significant digit to the least significant digit is:
0 0 0 0 -1 -1 0
The corresponding decimal expression is:
-11
Here is the remainder of N1/N2:
The STFRe expression from the most significant digit to the least significant digit is:
0 0 0 0 0 0 0
The corresponding decimal expression is:
0
```

Figure 3.13(a) An example result of division

```
Please input the # of bits logical system you want to create:
11
Please input 2 values to do the operation:
137
-25
Please input one operator among '+', '-', '*' and '/':
/
Here is the quotient of N1/N2:
The STFRe expression from the most significant digit to the least significant digit is:
0 0 0 0 -1 1 0
The corresponding decimal expression is:
-5
Here is the remainder of N1/N2:
The STFRe expression from the most significant digit to the least significant digit is:
0 0 0 1 -1 0 -1
The corresponding decimal expression is:
12
```

Figure 3.13(b) An example result of division

```
Please input the # of bits logical system you want to create:
11
Please input 2 values to do the operation:
-12
136
Please input one operator among '+', '-', '*' and '/':
/
Here is the quotient of N1/N2:
The STFRe expression from the most significant digit to the least significant digit is:
0 0 0 0 0 0 0
The corresponding decimal expression is:
0
Here is the remainder of N1/N2:
The STFRe expression from the most significant digit to the least significant digit is:
0 0 0 0 -1 -1 -1
The corresponding decimal expression is:
-12
```

Figure 3.13(c) An example result of division

```
Please input the # of bits logical system you want to create:
11
Please input 2 values to do the operation:
13
0
Please input one operator among '+', '-', '*' and '/':
/
The divisor can't be zero!
```

Figure 3.13(d) An example result of division

Flowchart 3.12(a) shows the whole procedure of division. Flowcharts 3.12(b) and 3.12(c) show details in the first part, namely making Fibonacci multiples of the divisor, and second part, namely successive subtractions, of division respectively. Figures 3.13(a)-3.13(d) show four examples of division. Given 11-bit UBFR(min), the maximal integer is F_{12} - 1 = 143. Therefore, the range of the STFRe{-1,0,1} is [-143, 143]. In figure 3.13(a), the quotient of 121/-11 is -11 and there is no remainder; in figure 3.13(b), the quotient of 137/-25 is -5 and the remainder is 12; in figure 3.13(c), since the absolute value of dividend is less than the absolute value of divisor, we have quotient of -12/136 equal to 0 and remainder equal to -12; in figure 3.13(d), since the divisor can't be zero, we terminate the procedure and send a warning message.

Chapter 4: Conclusions

4.1 Conclusions of The Thesis

In this section, we are going to recall and conclude what we did in the whole thesis.

1. Starting with the introduction of basic properties of the Fibonacci numbers, we put forward the idea to build a Fibonacci logic system.
2. By analogy with the binary system, we use unsigned binary Fibonacci representation, namely UBFR, to express non-negative integers. We find for the same non-negative integer, there may exist several UBFRs to express it.
3. To unify the form in Fibonacci arithmetic operations as well as use as little number of 1s as possible, we select the minimal form of UBFR, namely the UBFR(min), to represent the non-negative integers, which is unique.
4. In order to include the negative integers and double the range, we introduce the signed ternary Fibonacci representation using only even subscripts, namely STFRe{-1,0,1}, converted from the corresponding UBFR(min).
5. Compared with the binary Fibonacci representations, the STFRe{-1,0,1} can convert the integer N into –N just by a simple form of complementation as 1 <-> -1, 0 <-> 0, without any extra bit.
6. We develop Fibonacci addition, subtraction, multiplication and division based on the STFRe{-1,0,1}, through the combination of analogy with the conventional arithmetic operations' methods as well as injection of the STFRe{-1,0,1}'s special properties.
7. Illustrate some clever ways to check overflow occurring in the arithmetic operations based on the STFRe{-1,0,1} and to avoid as many unnecessary steps as possible.

4.2 Open Problems

In the final section, we provide some open problems, among which some are further improvement measures to better realize the Fibonacci computer, others are several innate defects of the Fibonacci system compared with the conventional binary system.

4.2.1 Further Improvements

1. Since we can use different Fibonacci coding forms to represent the same integer, it may be a good idea to take advantage of this redundancy property to improve the Fibonacci computer's error detection ability [16-18].
2. We can implement mixed Fibonacci arithmetic operations among more than two operands. And we should have an investigation for the order of precedence in the mixed operations.
3. We can simplify the steps of making Fibonacci multiples in multiplication by using prefix coding.

4. In the conventional binary system, it is natural to extend to fractional values. Therefore, we wonder whether we can further include fractional values in the four fundamental arithmetic operations.

5. We can try to include more operators to the Fibonacci system, such as power operator, integral or differential.

6. We can try to re-implement all the representations and calculations in this thesis based on the Lucas number system.

7. Finally, we desire to apply all the ideas and improvements to hardware implementations using three-state devices or two-bit per cell binary equivalents [19].

4.2.2 Disadvantages of The Fibonacci System

1. Although we have realized the four fundamental arithmetic operations based on the STFRe{-1,0,1}, we have to admit that those arithmetic operations developed in Chapter 3 are much more complicated compared with the arithmetic operations in the conventional binary system. The conventional binary system only has a single carry-propagation direction to more significant digits in any kind of arithmetic operations. And it can implement multiplication and division easily by "left shifting" or "right shifting".

2. In the preliminary preparations, the algorithms converting the decimal to the UBFR(min) and then converting the UBFR(min) to the STFRe{-1,0,1} are themselves bulkier than the simple conversion from the decimal to the conventional binary representation.

3. The integer range is $[-2^{n-1}, 2^{n-1}-1]$ in the conventional n-bit signed binary system, but is $[-F_{n+1}+1, F_{n+1}-1]$ in the even-subscripted signed ternary Fibonacci system where the ternary coefficients take values from the set {-1,0,1}. It's obvious that given the same n-bit size, the integer range in the conventional binary system is much larger than in the ternary Fibonacci system. And the gap widens more and more sharply as n grows larger and larger.

Finally, we want to say that, even though the Fibonacci computer has its own advantages compared with the conventional binary one, while it is unlikely to remain more than a curiosity. At least, there is still a long way to explore.

Coding Part

```
class Fibonacci{
public:
   vector<int> num1_UBFR;
   vector<int> num2_UBFR;
   vector<int> num1_STFR;
   vector<int> num2_STFR;
   vector<int> result_STFR;
   vector<int> base_fibonacci;
   int digit;
   int value1, value2;
   int sign1, sign2;

   Fibonacci(int input, int input1, int input2) {
      value1 = abs(input1);
      value2 = abs(input2);
      sign1 = input1 < 0 ? -1 : 1;
      sign2 = input2 < 0 ? -1 : 1;
      digit = input;

      int base0 = 0, base1 = 1;
      for (int i = 0; i <= input; i++) {
         base_fibonacci.push_back(base0 + base1); // F_2 to F_(digit+2)
         base0 = base1;
         base1 = base_fibonacci[i];
         num1_UBFR.push_back(0); // F_1 to F_(digit+1)
         num2_UBFR.push_back(0);
         if (i % 2 == 0) {
            num1_STFR.push_back(0); // F_2,...,F_(digit+2), if digit is even
            num2_STFR.push_back(0);
            result_STFR.push_back(0);
         }
      }

      if (digit % 2 == 1) {
         num1_UBFR.push_back(0); // For odd digit, F_1 to F_(digit+2)
         num2_UBFR.push_back(0);
         num1_STFR.push_back(0); // For odd digit, F_2,F_4,...,F_(digit+3)
         num2_STFR.push_back(0);
         result_STFR.push_back(0); // For odd digit, F_2,F_4,...,F_(digit+3)
      }
   }


   int Decimal_UBFR() {
      int start = digit - 1; // Start from F_digit, end in F_2
      while (start > 0 && value1 > 0) {
         if (value1 < base_fibonacci[start - 1]) {
            start -= 1;
         } else {
```

```
            num1_UBFR[start] = 1;
            value1 -= base_fibonacci[start - 1];
            start -= 2;
        }
    }

    start = digit - 1;
    while (start > 0 && value2 > 0) {
        if (value2 < base_fibonacci[start - 1]) {
            start -= 1;
        } else {
            num2_UBFR[start] = 1;
            value2 -= base_fibonacci[start - 1];
            start -= 2;
        }
    }

    if (value1 == 0 && value2 == 0) {
        return 1;
    } else {
        if (value1 > 0 && value2 > 0) {
            cout << "Both Inputs are overflow!" << endl;
        } else if (value1 > 0) {
            cout << "Input1 is overflow!" << endl;
        } else {
            cout << "Input2 is overflow!" << endl;
        }
        return -1;
    }
}

void UBFR_STFR(){
    for(int i = 0; i < num1_STFR.size()-1; i++){
        num1_STFR[i] = (num1_UBFR[i * 2] + num1_UBFR[i * 2 + 1] - num1_UBFR[i * 2 + 2])
* sign1;
        num2_STFR[i] = (num2_UBFR[i * 2] + num2_UBFR[i * 2 + 1] - num2_UBFR[i * 2 + 2])
* sign2;
    }

}

/*int check_highest() {
    int i = result_STFR.size()-2;
    while(i >= 0) {
        if(abs(result_STFR[i]) >= 2) {
            break;
        } else {
            i--;
        }
    }
    return i;
} */
```

```
int check_lowest() {
   int i = 0;
   while(i < result_STFR.size()) {
      if(abs(result_STFR[i]) >= 2) {
         break;
      } else {
         i++;
      }
   }
   return i;
}

void correct(int i_l) { //correct from lowest
   if(result_STFR[i_l] == 2 || result_STFR[i_l] == 3) {
      result_STFR[i_l+1]++;
      if(i_l != 0) {
         result_STFR[i_l - 1]++;
      }
      if(result_STFR[i_l] == 2) {
         result_STFR[i_l] = -1;
      } else {
         result_STFR[i_l] = 0;
      }
   } else {
      result_STFR[i_l+1]--;
      if(i_l != 0) {
         result_STFR[i_l - 1]--;
      }
      if(result_STFR[i_l] == -2) {
         result_STFR[i_l] = 1;
      } else {
         result_STFR[i_l] = 0;
      }
   }
}

int check_overflow() {
   int len = result_STFR.size();
   if(digit % 2 == 0) {
      if(result_STFR[len - 1] == 1) {
         if(result_STFR[len - 2] == 1 || result_STFR[len - 2] == 0) {
            return 1;
         } else {
            for(int i = len - 3; i >= 0; i--) {
               if(result_STFR[i] == 0) {
                  continue;
               } else if(result_STFR[i] == -1) {
                  return 0;
               } else {
                  return 1;
               }
            }
            return 1;
```

```
        }
    } else if(result_STFR[len - 1] == -1) {
        if(result_STFR[len - 2] == -1 || result_STFR[len - 2] == 0) {
            return -1;
        } else {
            for(int i = len - 3; i >= 0; i--) {
                if(result_STFR[i] == 0) {
                    continue;
                } else if(result_STFR[i] == 1) {
                    return 0;
                } else {
                    return -1;
                }
            }
            return -1;
        }
    } else {
        return 0;
    }
} else {
    if(result_STFR[len - 1] == 1) {
        return 1;
    }
    if(result_STFR[len - 1] == -1) {
        return -1;
    }

    if(result_STFR[len - 2] == 1) {
        for(int i = len - 3; i >= 0; i--) {
            if(result_STFR[i] == 0) {
                continue;
            } else if(result_STFR[i] == -1) {
                return 0;
            } else {
                return 1;
            }
        }
        return 1;
    } else if(result_STFR[len - 2] == -1) {
        for(int i = len - 3; i >= 0; i--) {
            if(result_STFR[i] == 0) {
                continue;
            } else if(result_STFR[i] == 1) {
                return 0;
            } else {
                return -1;
            }
        }
        return -1;
    } else {
        return 0;
    }
}
```

```
}

int multi_addition(vector<vector<int>>& multiples, int start, bool flag_overflow) {
   while(start > 0) {
      if(flag_overflow == true && num1_STFR[(start - 1) / 2] == 1) {
         return sign1 * sign2;
      }

      if(num1_STFR[(start - 1) / 2] == 1) {
         for(int i = 0; i < result_STFR.size(); i++) {
            result_STFR[i] = result_STFR[i] + multiples[start][i];
         }
      } else if(num1_STFR[(start - 1) / 2] == -1) {
         for(int i = 0; i < result_STFR.size(); i++) {
            result_STFR[i] = result_STFR[i]- multiples[start][i];
         }
      }

      bool flag = true;
      while(flag) {
         int i_l = check_lowest();
         if(i_l == result_STFR.size()) {
            flag = false;
         } else {
            correct(i_l);
         }
      }

      int sign = check_overflow();
      if(sign == 1) {
         flag_overflow = true;
      } else {
         flag_overflow = false;
      }

      start -= 2;
   }

   if(flag_overflow ==  true) {
      return sign1 * sign2;
   } else {
      if(sign1 * sign2 == -1) {
         for(int i = 0; i < result_STFR.size(); i++) {
            result_STFR[i] = result_STFR[i] * -1;
         }
      }
      return 0;
   }

}

void print_answer() {
```

```
        cout << "The STFRe expression from the most significant digit to the least significant digit
is:" << endl;
        for(int i = result_STFR.size() - 1; i >= 0; i--) {
            cout << result_STFR[i] << " ";
        }
        cout << endl;

        int sum = 0;
        for(int i = 0; i < result_STFR.size() - 1; i++) {
            sum += result_STFR[i] * base_fibonacci[2 * i];
        }
        if(digit % 2 == 0) {
            sum += result_STFR[result_STFR.size() - 1] * base_fibonacci[2 * (result_STFR.size() -
1)];
        }
        cout << "The corresponding decimal expression is:" << endl << sum << endl;
    }


    void Addition() {
        if(Decimal_UBFR() == -1) {
            return;
        }
        UBFR_STFR();
        for(int i = 0; i < result_STFR.size(); i++) {
            result_STFR[i] = num1_STFR[i] + num2_STFR[i];
        }

        bool flag = true;
        while(flag) {
            int i_l = check_lowest();
            if(i_l == result_STFR.size()) {
                flag = false;
            } else {
                correct(i_l);
            }
        }

        int sign = check_overflow();
        if(sign == 1) {
            cout << "Overflow! The result exceeds the maximum value!" << endl;
        } else if(sign == -1) {
            cout << "Overflow! The result exceeds the minimum value!" << endl;
        } else {
            print_answer();
        }

    }

    void Subtraction() {
        if(Decimal_UBFR() == -1) {
            return;
        }
```

```
        UBFR_STFR();
        for(int i = 0; i < result_STFR.size(); i++) {
            result_STFR[i] = num1_STFR[i] - num2_STFR[i];
        }

        bool flag = true;
        while(flag) {
            int i_l = check_lowest();
            if(i_l == result_STFR.size()) {
                flag = false;
            } else {
                correct(i_l);
            }
        }

        int sign = check_overflow();
        if(sign == 1) {
            cout << "Overflow! The result exceeds the maximum value!" << endl;
        } else if(sign == -1) {
            cout << "Overflow! The result exceeds the minimum value!" << endl;
        } else {
            print_answer();
        }
    }

    void Multiplication() {
        if(Decimal_UBFR() == -1) {
            return;
        }
        for(int i = 0; i < num1_STFR.size() - 1; i++){ // UBFR_STFR_absolute
            num1_STFR[i] = num1_UBFR[i * 2] + num1_UBFR[i * 2 + 1] - num1_UBFR[i * 2 + 2];
            num2_STFR[i] = num2_UBFR[i * 2] + num2_UBFR[i * 2 + 1] - num2_UBFR[i * 2 + 2];
        }

        int h_index = num1_STFR.size() - 2;
        while(h_index >= 0) { // find the highest position that equals to 1(here value1 must be
positive)
            if(num1_STFR[h_index] == 1) {
                break;
            }
            h_index--;
        } // no need to do the rest multiples

        if(h_index < 0) { // the result is 0
            print_answer();
            return;
        }
        if(h_index == 0) { // the result is value2 * sign
            for(int i = 0; i < result_STFR.size(); i++) {
                result_STFR[i] = num2_STFR[i] * sign1 * sign2;
            }
            print_answer();
            return;
```

```
}


int result_sign = 0; // record whether the result is overflow or not
int sign = 0; // record whether there exists an overflow in the outside loop

vector<vector<int>> multiples((h_index + 1) * 2); //must be even elements
multiples[0] = num2_STFR; // F_1 * value2
multiples[1] = num2_STFR; // F_2 * value2

for(int i = 2; i < multiples.size(); i++) {
   for (int j = 0; j < num2_STFR.size(); j++) {
      result_STFR[j] = multiples[i - 1][j] + multiples[i - 2][j];
   } // multiple of F_(i+1) * value2

   bool flag = true; // correct expression
   while (flag) {
      int i_l = check_lowest();
      if (i_l == result_STFR.size()) {
         flag = false;
      } else {
         correct(i_l);
      }
   }

   multiples[i] = result_STFR;

   sign = check_overflow();
   if (sign == 1) { // if the multiple is overflow(the multiple must be positive)
      if ((i + 1) % 2 == 1) { // if the first overflow multiple is odd
         if(((multiples.size() - 1 - i) == 1) && (num1_STFR[i / 2 - 1] == -1) && i > 3) {
            result_sign = multi_addition(multiples, i - 3, true); // case_1: first odd overflows
         } else {
            if(sign1 * sign2 == 1) {
               result_sign = 1;
            } else {
               result_sign = -1;
            }
         }
      } else { // if the first overflow multiple is even
         if(i == multiples.size() - 1) {
            result_sign = multi_addition(multiples, i - 2, true); // case_2: first even overflows
         } else {
            if(sign1 * sign2 == 1) {
               result_sign = 1;
            } else {
               result_sign = -1;
            }
         }
      }

      break;
   }
```

```
    }

    if(sign == 0) { // if there is not overflow during the whole loop
        result_sign = multi_addition(multiples, multiples.size() - 3, false); // case_3: no overflow
    }

    if(result_sign == 1) {
        cout << "Overflow! The result exceeds the maximum value!" << endl;
    } else if(result_sign == -1) {
        cout << "Overflow! The result exceeds the minimum value!" << endl;
    } else {
        print_answer();
    }

}

void Division() {
    if(Decimal_UBFR() == -1) {
        return;
    }
    for(int i = 0; i < num1_STFR.size() - 1; i++){ // UBFR_STFR_absolute
        num1_STFR[i] = num1_UBFR[i * 2] + num1_UBFR[i * 2 + 1] - num1_UBFR[i * 2 + 2];
        num2_STFR[i] = num2_UBFR[i * 2] + num2_UBFR[i * 2 + 1] - num2_UBFR[i * 2 + 2];
    }

    bool non_zero = false; // check if the divisor is 0
    for(int i = num2_STFR.size() - 1; i >= 0; i--) { // value2 must be non_negative
        if(num2_STFR[i] == 1) {
            non_zero = true;
            break;
        }
    }
    if(non_zero == false) {
        cout << "The divisor can't be zero!" << endl;
        return;
    }

    vector<vector<int>> multiples; // largest capacity is "digit"
    multiples.push_back(num2_STFR); // M_1 = F_1 * value2
    multiples.push_back(num2_STFR); // M_2 = F_2 * value2

    int sign = 0; // record the multiple: 1. overflow or not; 2. exceed the dividend or not

   for(int i = 2; i < digit; i++) { // start from M_3
        for (int j = 0; j < result_STFR.size(); j++) {
            result_STFR[j] = multiples[i - 1][j] + multiples[i - 2][j];
        } // multiple of F(i+1) * value2

        bool flag = true; // correct expression
        while (flag) {
            int i_l = check_lowest();
            if (i_l == result_STFR.size()) {
                flag = false;
```

```
        } else {
            correct(i_l);
        }
    }

    sign = check_overflow();
    if (sign == 1) { // if the multiple is overflow(the multiple must be positive)
        break;
    } else { // if the multiple is not overflow
        vector<int> temple = result_STFR;
        for (int j = result_STFR.size() - 1; j >= 0; j--) {
            result_STFR[j] = num1_STFR[j] - result_STFR[j]; // difference can't be overflow
        }

        flag = true; // correct expression
        while (flag) {
            int i_l = check_lowest();
            if (i_l == result_STFR.size()) {
                flag = false;
            } else {
                correct(i_l);
            }
        }

        for (int j = result_STFR.size() - 1; j >= 0; j--) { // check whether it is negative
            if (result_STFR[j] == 0) {
                continue;
            } else {
                sign = result_STFR[j];
                break;
            }
        }
        if (sign == -1) { // the multiple exceeds the dividend
            break;
        } else {
            multiples.push_back(temple);
        }
    }
}

vector<int> b_system(result_STFR.size() * 2 + 1, 0);

//int t_h; // the highest non-negative index in t_system
//if(multiples.size() % 2 == 1) { // if the highest index in b_system is odd
    //t_h = (multiples.size() - 1) / 2;
//} else { // if the highest index in b_system is even
    //t_h = multiples.size() / 2 - 1;
//}

vector<int> rest = num1_STFR;
sign = 0; // sign records whether the previous bit is 1 or not
for(int i = multiples.size() - 1; i > 0; i--) {
    if(sign == 0) { // the previous bit is 0
```

```
            for(int j = 0; j < num1_STFR.size(); j++) {
               result_STFR[j] = rest[j] - multiples[i][j];
            }

            bool flag = true; // correct expression
            while (flag) {
               int i_l = check_lowest();
               if (i_l == result_STFR.size()) {
                  flag = false;
               } else {
                  correct(i_l);
               }
            }


            int test = result_STFR.size() - 1;

            while(test >= 0) {
               if(result_STFR[test] == 1) {
                  b_system[i] = 1;
                  rest = result_STFR;
                  sign = 1;
                  break;
               }
               if(result_STFR[test] == -1) {
                  break;
               }
               test --;
            }
            if(test < 0) {
               b_system[i] = 1;
               rest = result_STFR;
               break;
            }

         } else {
            sign = 0;
         }
      }

      //result_STFR[t_h] = 1 * sign1 * sign2;
      //for(int i = result_STFR.size() - 1; i > t_h; i--) { // set 0s after index t_h in result_STFR
         //result_STFR[i] = 0;
      //}

      for(int i = 0; i < result_STFR.size(); i++) {
         result_STFR[i] = (b_system[i * 2] + b_system[i * 2 + 1] - b_system[i * 2 + 2]) * sign1 *
sign2;
      }

      cout << "Here is the quotient of N1/N2:" << endl; // print the quotient
      print_answer();
```

```
        result_STFR = rest;
        if(sign1 == -1) {
            for(int i = 0; i <= result_STFR.size(); i++) {
                result_STFR[i] *= -1;
            }
        }
        cout << "Here is the remainder of N1/N2:" << endl; // print the remainder
        print_answer();
    }
};


int main() {
    int input1, input2, bits;
    char oper;
    cout << "Please input the # of bits logical system you want to create:" << endl;
    cin >> bits;
    cout << "Please input 2 values to do the operation:" << endl;
    cin >> input1 >> input2;
    Fibonacci Fib(bits, input1, input2);
    cout << "Please input one operator among '+', '-', '*' and '/':" << endl;
    cin >> oper;
    switch(oper) {
        case '+': {
            Fib.Addition();
            break;
        }
        case '-': {
            Fib.Subtraction();
            break;
        }
        case '*': {
            Fib.Multiplication();
            break;
        }
        case '/': {
            Fib.Division();
            break;
        }
        default:
            cout << "Wrong operator!" << endl;
    }

    return 0;
}
```

Bibliography

[1] Alexey Stakhov, Alexey Borisenko and Svetlana Matsenko, “Fibonacci Counter based on Zeckendorf’s Theorem (Boolean Realization)”, http://elib.mi.sanu.ac.rs/files/journals/vm/57/vmn57p1-19.pdf.

[2] V. E. Hoggatt, Jr., “Fibonacci and Lucas Numbers”, Boston: Houghton Mifflin Co., 1969.

[3] P.Ligomenides and R. Newcomb, “Multilevel Fibonacci Conversion And Addition”, The Fibonacci Quarterly, 1984, pp. 196-203.

[4] Takahiro Yamamoto, “Fibonacci Numbers and Chebyshev Polynomials”, Department of Physics and Astronomy, University of Utah, December 2, 2015. http://www.physics.utah.edu/~lebohec/P3730/Labs/Lab13/fibonacci.pdf

[5] M. Livio, “The Golden Ratio: The Story of PHI, the World’s Most Astonishing Number”, Broadway Books, 2003.

[6] Vorobyov N.N, “Fibonacci Numbers”, first ed. 1961; Moscow: Nauka, Russia, 1978; p.192.

[7] P. Ligomenides and R. Newcomb, “Complement Representations in the Fibonacci Computer”, Proceedings of the Fifth Symposium on Computer Arithmetic, Ann Arbor, Michigan, May 1981, pp. 6-9.

[8] Zeckendorf's theorem. http://www.maths.surrey.ac.uk/hostedsites/R.Knott/Fibonacci/fibrep.html http://en.wikipedia.org/wiki/Zeckendorf's_theorem

[9] V. D. Hoang, "A Class of Arithmetic Burst-Error-Correcting Codes for the Fibonacci Computer", Ph.D. Dissertation, University of Maryland, December 1979.

[10] D. E. Daykin, "Representation of Natural Numbers as Sums of Generalized Fibonacci Numbers", Journal of the London Math. Society, Vol. 35 (1960), pp. 143-160.

[11] J. L Brown, Jr., "Zeckendorfs Theorem and Some Applications", The Fibonacci Quarterly, Vol. 2, No. 3 (Oct.1964), pp. 163-168.

[12] J. L Brown, Jr., "A New Characteristic of the Fibonacci Numbers", The Fibonacci Quarterly, Vol. 3, No. 1 (Feb.1965), pp. 1-8.

[13] Monteiro P. and R.W. Newcomb, “Minimal and maximal Fibonacci Representations: Boolean Generation”, The Fibonacci Quarterly, 1976, Vol.14, No 1.

[14] Peter Fenwick, “Zeckendorf Integer Arithmetic”, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.2030&rep=rep1&type=pdf

[15] P. Ligomenides and R. Newcomb, “Equivalence of Some Binary, Ternary and Quaternary Fibonacci Computers”, Proceedings of the Eleventh International Symposium on Multiple-Valued Logic, Norman, Oklahoma, May 1981, pp. 82-84.

[16] Stakhov A.P., “By the Principle of the Golden Ratio: a New Way for the Development of Computing Technology”, Bulletin of the Ukrainian Academy of Sciences, Ukraine, 1990, No.2, 14-25.

[17] Stakhov A.P., “Redundant Binary Positional Number Systems”, Homogeneous digital computing and integral structures, Taganrog Radio Institute, Russia, 1974, 5-41.

[18] Stakhov A.P., “An Use of Natural Redundancy of the Fibonacci Number Systems for Checking Computing Structures”, Automation and Computer Engineering, Russia, 1975, No 6, 80-87.

[19] M. Stark, "Two Bits Per Cell ROM", Digest of Papers, Compcon 81, February 1981, San Francisco, pp. 209-212.