Principles and construction of MSD adder in ternary optical computer

Yi JIN, YunFu SHEN, JunJie PENG, ShiYi XU, GuangTai DING, DongJian YUE and HaiHang YOU

Citation: SCIENCE CHINA Information Sciences **53**, 2159 (2010); doi: 10.1007/s11432-010-4091-9

View online: http://engine.scichina.com/doi/10.1007/s11432-010-4091-9

View Table of Contents: http://engine.scichina.com/publisher/scp/journal/SCIS/53/11

Published by the Science China Press

SCIENCE CHINA
Information Sciences

• RESEARCH PAPERS •

November 2010 Vol. 53 No. 11: 2159–2168
doi: 10.1007/s11432-010-4091-9

Principles and construction of MSD adder in ternary optical computer

JIN Yi[1]*, SHEN YunFu[1], PENG JunJie[1], XU ShiYi[1], DING GuangTai[1], YUE DongJian[1] & YOU HaiHang[2]

[1]*School of Computer Engineering & Science, Shanghai University, Shanghai,* 200072 *China;*
[2]*National Institute for Computational Sciences, Oak Ridge, TN*37831*, USA*

Received April 23, 2010; accepted August 9, 2010

Abstract The two remarkable features of ternary values and a massive unit with thousands bits of parallel computation will make the ternary optical computer (TOC) with modified signed-digit (MSD) adder more powerful and efficient than ever before for numerical calculations. Based on the decrease-radix design presented previously, a TOC can satisfy either a user requiring huge capacity for data calculations or one with a moderate amount of data, if it is equipped with a prepared adder. Furthermore, with the application of pipelined operations and the proposed data editing technique, the efficiency of the prepared adder can be greatly improved, so that each calculated result can be obtained almost within one clock cycle. It is hopeful that by employing a MSD adder, users will be able to enter a new dimension with the creation of a new multiplier, new divider, as well as new matrix operator in a TOC in the near future.

Keywords ternary optical computer (TOC), modified signed-digit (MSD) adder, pipeline computing, data editing technique

Citation Jin Y, Shen Y F, Peng J J, et al. Principles and construction of MSD adder in ternary optical computer. Sci China Inf Sci, 2010, 53: 2159–2168, doi: 10.1007/s11432-010-4091-9

With the current rapid increase in the complexity of computer architectures, the power consumption of large scale systems has risen prohibitively. Much attention has been focused on reducing the power consumption in different ways. One of the ways of solving the problem is to use of an optical computer with its special non-electron characteristics of high speed, parallelism, multi-valued, and low power consumption. Considering these properties, researchers have been focusing mainly on improving the operating speed [1–3] and enlarging the number of parallel bits in these computers [4–6], but have often neglected the problem of reducing the power consumption.

A TOC prototype recently developed in our laboratory at Shanghai University is a typical optical computer with a huge number of data bits [6, 7]. Based on the decrease-radix design proposed in 2008 [8], we can configure any number of bits as specific groups of tri-valued logic units at any time in the TOC. However, as thousands of bits exist in an adder, the ripple-carry technique is infeasible in a TOC because of the terrible carry delay. In addition, the look-ahead carry technique does not suit the construction of optical elements due to the high complexity of its tree type architecture. For these reasons, we proposed a new technique called the direct parallel carry channel (DPCC) aiming at accelerating the carry operation [9]. Unfortunately, this scheme has failed to be put into practice for various reasons.

*Corresponding author (email: yijin@shu.edu.cn)

However, based on further study of TOC applications [10, 11], we proposed a new modified signed-digit (MSD) adder, and new multiplication, division, and matrix multiplication algorithms, in which the MSD adder is considered to be the crucial technique for fixing the current problems. Therefore, the rest of this paper is dedicated mainly to a discussion of the MSD adder.

The paper is organized as follows. Section 1 introduces the optical processor (OP) of a TOC. Section 2 discusses the MSD numerical system, including MSD expressions and MSD addition. Section 3 focuses on the construction of a MSD adder for the TOC, while section 4 presents the pipeline computing technique in the MSD adder. Section 5 proposes a new technique called data editing. Section 6 discusses the results of our experiments using the MSD adder, while section 7 presents our conclusions.

1 Characteristics of a ternary optical processor

In a TOC, three values are used, i.e., 1, 0, and -1, where 0 represents darklight, while the other two values, 1 and -1, denote two polarized lights whose polarization directions are orthogonal to each other. In a TOC two polaroid sheets and a liquid crystal array are used to create the ternary optical processor. Because there are a great number of pixels in a liquid crystal array, one of the obvious characteristics of the TOC is the ability to process massive data [2, 12]. We implemented a prototype in our laboratory in 2007 with 360 bits for parallel data operations. Recently, we have increased the word length up to one thousand in our experiments and are considering adding more bits in the near future. On the other hand, the power consumption of optical elements is significantly lower than that of electronic components, such that only a few milliwatts of power are required for a liquid crystal screen with millions of pixels. Thus, our prototype with 360 bits consumes only 100 watts, of which the optical part uses only 2 watts for its LED components.

The new optical processor (OP) currently under construction has been designed based on the decrease-radix principle. It has been theoretically proven that any one of the 3^9 (=19683) ternary logic units can be constructed by combining not more than 6 of the 18 basic operators. In other words, we can design and construct any kind of ternary logic unit in any section of the OP by configuring the 18 basic operators, as long as we have enough types of operators on hand. This is what we refer to as the re-configurability property of a TOC [8].

Because all bits in a logic unit are totally independent of one another, different sections of an OP can be configured into different logic units. Therefore, with one machine instruction the different sections of an OP can execute different logic operations in parallel, which is impossible at all in an electronic computer. Moreover, there is a special feature called OP extensibility, which means that a bigger OP can be constructed by adding a new OP to the old one instead of using a multi-OP. Theoretically, the vast number of data bits in a TOC can satisfy almost any users' requirements and greatly reduce the burden of programming without the need for communication between CPUs as in traditional electronic computers.

For several reasons, we selected a liquid crystal array as the main material with which to build the OP. In this case, the 18 basic operators can be constructed in a similar way: holding a liquid pixel in the middle of two polaroid sheets. The only differences between the 18 basic operators are the polarization directions of the polaroid and the innate optical rotation of the liquid crystal. Figure 1 shows the design of a ternary OP, where LCA depicts the liquid crystal array; and $P1$ and $P2$ represent the two sheets of polaroid. All basic operators are created from the LCA, $P1$ and $P2$. The reconstructer in the control module includes various circuits to control the optical rotation of the LCA and routines to configure the basic operators into specific ternary logic units. These routines are used by the data-bit management program included in the TOC's monitoring system to produce signal e from the reconstructer to meet the user's needs. Signal e is sent to the rest of the OP to reconfigure the designated location of the OP into a specialized ternary logic unit with a sufficient number of data-bits. This is what we refer to as reconfiguring the OP.

Input lights, a and b, are applied in parallel to the OP from two openings. One of these, say b, enters the Photoelectric transducer and produces a control signal d. Signal d is then sent to the LCA to control

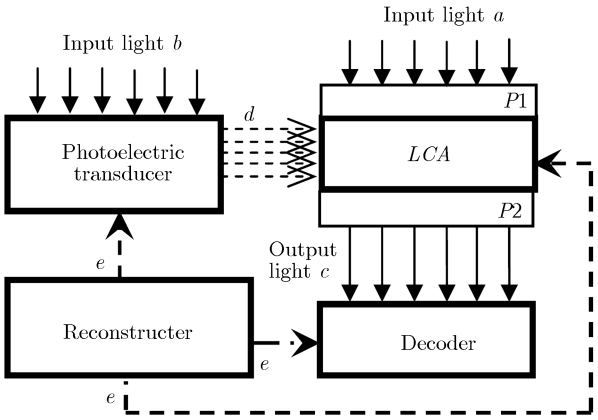

Figure 1 Block diagram of ternary optical processor.

the LCA's pixels of the corresponding data-bit. The other input, a, passes through the $P1$, LCA, and $P2$ to produce the resulting output light c, by changing its light state in the LCA according to the electric signal d. The decoder transforms the output light c into an electric signal so that the result can be conveniently sent to an electronic computer.

Because of the characteristics of re-configurability and vast number of data bits, the TOC is much suited for logic operations where there is no relationship among the data bits. Since the relativity among the data bits of the carries in normal addition, the TOC is not suited to normal adder. Thanks to the efforts of mathematicians in the 1960s, several difficult issues have been solved theoretically. Some promising results of redundant binary representations and counts have been proposed without the need of a carry operation [13, 14]. Unfortunately, these achievements cannot be employed in binary electronic computers, due to the use of the binary system. However, a new dimension has been opened using a TOC which operates on three values, of which one is able to serve as a redundant value, and redundant binary operations can be carried out without any difficulty on a TOC.

2 MSD representation

In a normal n digital system, there should be n digits expressing 0 to $n-1$, and a carry will occur in the highest position if the value of a digit is greater than $n-1$. However, in a redundant $n+r$ digital system, each digit could have $n+r$ states, where n and r are integers, although this still obeys the carry regulation. A digit with $n+r$ states should have r redundant states, which is called redundant counting. In a redundant $n+r$ digital system, therefore, one digit could take several different forms, something which is quite unfamiliar to human beings. Nevertheless, a redundant digital system happens to be most useful in multi-valued calculations. We take advantage of a redundant digital system in our TOC system, where we set $n=2$ and $r=1$, which is called a (2, 1) redundant digital system. One of these (2, 1) redundant digital systems is referred to as the MSD as explained in the next subsection.

2.1 MSD expression

The value of a digit A can be expressed as follows:

$$A = \sum_i a_i \times 2^i. \tag{1}$$

Here, the symbol set of a_i is $\{\bar{1}, 0, 1\}$, with $\bar{1}$ denoting the value -1. The 2^i shows that the MSD is still a binary system. Because there are three values for a_i, the MSD expressions of a digit could have more forms. For example, the MSD expressions of the digit 5 can be described as follows:

$$(5)_{10} = (101)_2 = (101)_{\text{MSD}} = (10\bar{1}\bar{1})_{\text{MSD}} = (1\bar{1}0\bar{1}\bar{1})_{\text{MSD}}.$$

Table 1 T, W, T′, and W′ transformations used in MSD addition

T or T_2 transform		W transform		T′ transform		W′ transform	
b or w' \ a or t'	$\bar{1}$ 0 1	b \ a	$\bar{1}$ 0 1	w \ t	$\bar{1}$ 0 1	w \ t	$\bar{1}$ 0 1
$\bar{1}$	$\bar{1}$ $\bar{1}$ 0	$\bar{1}$	0 1 0	$\bar{1}$	$\bar{1}$ 0 0	$\bar{1}$	0 $\bar{1}$ 0
0	$\bar{1}$ 0 1	0	1 0 $\bar{1}$	0	0 0 0	0	$\bar{1}$ 0 1
1	0 1 1	1	0 $\bar{1}$ 0	1	0 0 1	1	0 1 0

The MSD expressions of the digit -5 are as follows:

$$(-5)_{10} = (-101)_2 = (\bar{1}0\bar{1})_{\mathrm{MSD}} = (\bar{1}011)_{\mathrm{MSD}} = (\bar{1}1011)_{\mathrm{MSD}}.$$

From the above examples it can be seen that by complementing each bit of a digit, its complement digit can easily be obtained in an MSD expression. Taking advantage of this, we can effectively omit the expression of a negative number.

2.2 MSD addition

The MSD addition operation is very different from the normal binary one. MSD addition is completed via four tri-value logical transformations, T (or T_2), W, T′, and W′, as listed in Table 1 [15, 16]. The operation is divided into five steps.

Step 1. Apply T and W to the entered operands a and b bit by bit. The result of T is u and the result of W is w.

Step 2. Append one 0 to the tail of u. The result is called the intermediate result t.

Step 3. Apply T′ and W′ separately to t and w bit by bit. The result of T′ is u' and the result of W′ is w'.

Step 4. Append one 0 to the tail of u'. The result is called the intermediate result t'.

Step 5. Apply T to t' and w' bit by bit. The result is c which is the final result of the addition. To distinguish the two T transformations in Steps 1 and 5, the transformation in Step 5 is henceforth (and in Table 1) written as T_2.

For example, assume a=110100=$(52)_{\mathrm{D}}$, b=1$\bar{1}$100$\bar{1}$=$(23)_{\mathrm{D}}$, and suffix D specifies decimal digits, while no suffix means MSD.

The addition is given as: $c = a + b$=110100+1$\bar{1}$100$\bar{1}$

Step 1. Apply T to operands a and b to produce u=10110$\bar{1}$.
Apply W to operands a and b to produce w=00$\bar{1}\bar{1}$01.

Step 2. Append one 0 to the tail of u to produce t=10110$\bar{1}$0.

Step 3. Apply T′ to operands t and w to produce u'=0000000.
Apply W′ to operands t and w to produce w'=1010$\bar{1}\bar{1}$1.

Step 4. Append one 0 to the tail of u' to produce t'=00000000.

Step 5. Apply T_2 to operands t' and w' to produce c=01010$\bar{1}\bar{1}$1.

The decimal expression of c is $(75)_{\mathrm{D}}$ and the decimal addition can be expressed as

$$c = a + b = (52)_{\mathrm{D}} + (23)_{\mathrm{D}} = (75)_{\mathrm{D}}.$$

Both additions are accordant.

Meanwhile, b has another form in MSD, which is the same as binary b=10111=$(23)_{\mathrm{D}}$. With this form of b, the intermediate results of the MSD addition are as follows:

$$u = 110111, w = \bar{1}000\bar{1}\bar{1}, t = 1101110; u' = 0000000, w' = 100110\bar{1}, t' = 00000000.$$

The final result is c=100110$\bar{1}$. After translation into decimal, c is still $(75)_{\mathrm{D}}$. It is clear that binary is a form of MSD too, so using binary in MSD addition is the same as using other forms of MSD.

3 MSD adder for a TOC

In theory, it is equivalent which of the physical states of the TOC expresses which symbol of the MSD. Nevertheless, in this paper, darklight is used to express the symbol 0, one polarized state of light the symbol 1, and the other polarization symbol $\bar{1}$. So each MSD digit is expressed explicitly by hardware in the TOC, something that a binary computer is incapable of achieving.

From section 1, any logical unit can be configured in any section of the ternary OP (i.e., any section of data-bits). Moreover from section 2.2, MSD addition is completed via four tri-valued logic units T (T_2), W, T′, and W′. This means that there are three ways to complete the MSD addition in a TOC.

Option 1. The T, W, T′, W′, and T_2 are configured orderly using all the data-bits of the OP, and the transformation is executed as soon as one logic unit has been configured. The adder can process operands with as many data-bits as the OP, but it will spend additional time on the five reconfigurations.

Option 2. The OP is divided into two parts, $z1$ and $z2$. After configuring $z1$ into T and $z2$ into W, the transformations, T and W, are executed in parallel. Then, after reconfiguring $z1$ into T′ and $z2$ into W′, the transformations, T′ and W′, are executed in parallel. Lastly, reconfiguring $z1$ into T again, the transformation T_2 is executed. The adder can process operands with half as many data-bits as the OP, but it will spend less time on the three reconfigurations.

Option 3. The OP is divided into five parts, $z1, z2, z3, z4$, and $z5$. The $z1$ is configured into T, $z2$ into W, $z3$ into T′, $z4$ into W′, and $z5$ into T_2. After passing through T and W at the same time, the entered operands, a and b, produce the intermediate results t and w. Then, t and w pass through T′ and W′ at the same time to produce the next intermediate results t' and w'. Lastly, t' and w' pass through T_2 to produce the final result c. The adder can only process operands with one fifth as many data-bits as the OP, but it only configures the OP once.

In Option 3, the OP is only configured at the beginning. So, not only can we reconfigure a special adder for a user at any time, but we can also prepare a prepared MSD adder in the initialization phase of the TOC for general users. Most users will use the prepared adder instead of reconfiguring the OP to save time. Nevertheless, it is difficult to determine how many data-bits are sufficient for the prepared MSD adder. To increase the efficiency of the prepared adder we have developed a new technique, called data editing, which is introduced in section 5 of this paper. Meanwhile, having introduced pipeline computation, the prepared adder outputs one result nearly every clock cycle. Pipeline processing is discussed in section 4. Based on pipeline computation and data editing, the MSD adder in a TOC is configured according to Option 3. An outline of its operation process is given below.

Step 1. Divide the OP into five areas which are separately configured into transformations T, W, T′, W′, and T_2, respectively. If T and W have q bits, then T′ and W′ will have $q+1$ bits and T_2 will have $q+2$ bits. This task can be completed in the initialization phase of the TOC, so it will not impact on the user's time.

Step 2. Send operands a and b to T and W at the same time. Obtain intermediate results u and w.

Step 3. Add a 0 at the front of w and the tail of u, and then change u into t.

Step 4. Send w and t into T′ and W′ at the same time. Obtain the intermediate results u' and w'.

Step 5. Add a 0 at the front of w' and the tail of u', and then change u' into t'.

Step 6. Send w' and t' into T_2, and obtain the final result c.

4 Pipeline computing in a MSD adder

In modern computers the execution process of instructions is accomplished by using a pipeline. The pipeline technique is also used in the multiplication unit. However, the carry relativity of ordinary binary addition prevents the adder of electronic computer from using pipeline processing. Since no-carry in the MSD addition, the pipeline computing can be implemented in the MSD adder. In a TOC, the MSD adder is divided into five independent logical units, the order of use of which is fixed in operation. The right order of using the five logical units is the prior condition for introducing pipeline processing into the MSD adder. Pipeline computing in a TOC's MSD adder can be described as follows.

Assume that there are many pairs of data waiting for computation. After the first pair of data passing through T and W, their results are sent to T′ and W′, while the second pair of data is sent to T and W. After the first pair of data passing through T′ and W′, their results are sent to T_2, while the results of the second pair of data are sent to T′ and W′, and a third pair of data is sent to T and W. After the first pair of data passing through T_2, the first addition is complete. Once the final result of the first pair of data has been obtained, one addition will be completed every clock cycle thereafter, until all the final results have been obtained.

5 Data editing in MSD adder

A remarkable advantage of the TOC is that it has a large number of data bits. Therefore, a TOC is more suited to two specific cases of computing; one is the computation of very large data, while the other is the computation of many data items. A TOC can be reconfigured into a specific large MSD adder at any time for any user. However, the needs of users are different for large data and many data items. If an adder needs to be reconfigured for each type of user, a great deal of time and computing resources of the TOC will be consumed. To save time, it is necessary to build a prepared adder in the TOC to serve most users, and to reconfigure a large adder only for specific users.

With many input data of the requisite length, the efficiency of the prepared adder can be improved through pipeline computing. To improve the efficiency of the prepared adder, we break a large operand into several shorter ones or join several short operands into one that has the optimal length as a preliminary process. This is what we call data editing. Some skill is required for data editing because of appending a 0 at the tail of the intermediate results of T and T′.

5.1 Cutting long operands into several shorter ones

Assume operands a and b are a pair of long digits with $n+1$ bits, and compute $c = a + b$.

$$a = a_n a_{n-1} \dots a_k a_j a_i a_h \dots a_1 a_0;$$
$$b = b_n b_{n-1} \dots b_k b_j b_i b_h \dots b_1 b_0.$$

After transformation by T and W the intermediate results are

$$t = t_n t_{n-1} \dots t_k t_j t_i t_h \dots t_1 t_0 0;$$
$$w = 0 w_n w_{n-1} \dots w_k w_j w_i w_h \dots w_1 w_0.$$

After transformation by T′ and W′ the intermediate results are

$$t' = t'_{n+1} t'_n t'_{n-1} \dots t'_k t'_j t'_i t'_h \dots t'_1 t'_0 0;$$
$$w' = 0 w'_{n+1} w'_n w'_{n-1} \dots w'_k w'_j w'_i w'_h \dots w'_1 w'_0.$$

After transformation by T_2 the final result is

$$c = c_{n+2} c_{n+1} c_n c_{n-1} \dots c_{k+1} c_k c_j c_i c_h \dots c_1 c_0. \tag{2}$$

Now, we cut operand a into two parts a^1 and a^2 between a_j and a_i, and b into b^1 and b^2 between b_j and b_i.

$$a^1 = a_n a_{n-1} \dots a_{k+1} a_k a_j; \quad a^2 = a_i a_h \dots a_1 a_0;$$
$$b^1 = b_n b_{n-1} \dots b_{k+1} b_k b_j; \quad b^2 = b_i b_h \dots b_1 b_0.$$

After transformation by T and W the intermediate results are

$$t^1 = t_n t_{n-1} \dots t_{k+1} t_k t_j 0; \qquad t^2 = t_i t_h \dots t_1 t_0 0;$$
$$w^1 = 0 w_n w_{n-1} \dots w_{k+1} w_k w_j; \quad w^2 = 0 w_i w_h \dots w_1 w_0.$$

After transformation by T′ and W′ the intermediate results are

$$t'^1 = t'_{n+1}t'_n t'_{n-1}\dots t'_{k+1}t'_k t'^*_j 0; \quad t'^2 = t'_{i+1}t'_i t'_h \dots t'_1 t'_0 0;$$
$$w'^1 = 0w'_{n+1}w'_n w'_{n-1}\dots w'_{k+1}w'_k w'^*_j; \quad w'^2 = 0w'_{i+1}w'_i w'_h \dots w'_1 w'_0.$$

After transformation by T_2 the final results are

$$c^1 = c_{n+2}c_{n+1}c_n c_{n-1}\dots c_{k+1}c^*_k c^*_j; \quad c^2 = c_{i+2}c_{i+1}c_i c_h \dots c_1 c_0. \tag{3}$$

Two differences are evident between the two sequences from c_0 to c_{n+2} in eqs. (2) and (3). One difference is that c_{i+2} and c_{i+1} are two extra items in (3), and the other is that $c^*_k c^*_j$ is not equal to $c_k c_j$. After amending the two differences, c^1 and c^2 can be joined into c. It is easy to amend the first difference by deleting c_{i+2} and c_{i+1} from (3). Amending the second difference requires the following techniques: 1) copying the first two items of a^2 and b^2 (i.e., $a_i a_h$ and $b_i b_h$) to the ends of a^1 and b^1, respectively; and 2) cutting out the $c^*_k c^*_j$ from the end of c^1. This technique is referred to as "copying the breakpoint and trimming the connection point". We explain it as follows.

If

$$a^1 = a_n a_{n-1}\dots a_{k+1}a_k a_j \boldsymbol{a}_i \boldsymbol{a}_h; \quad a^2 = \boldsymbol{a}_i \boldsymbol{a}_h \dots a_1 a_0;$$
$$b^1 = b_n b_{n-1}\dots b_{k+1}b_k b_j \boldsymbol{b}_i \boldsymbol{b}_h; \quad b^2 = \boldsymbol{b}_i \boldsymbol{b}_h \dots b_1 b_0.$$

then

$$\begin{aligned}
&t^1 = t_n t_{n-1}\dots t_{k+1}t_k t_j \boldsymbol{t}_i \boldsymbol{t}_h 0; && t^2 = \boldsymbol{t}_i \boldsymbol{t}_h \dots t_1 t_0 0;\\
&w^1 = 0w_n w_{n-1}\dots w_{k+1}w_k w_j \boldsymbol{w}_i \boldsymbol{w}_h; && w^2 = 0\boldsymbol{w}_i \boldsymbol{w}_h \dots w_1 w_0;\\
&t'^1 = t'_{n+1}t'_n t'_{n-1}\dots t'_{k+1}t'_k t'_j \boldsymbol{t'}_i t'^*_h 0; && t'^2 = t'_{i+1}\boldsymbol{t'}_i t'_h \dots t'_1 t'_0 0;\\
&w'^1 = 0w'_{n+1}w'_n w'_{n-1}\dots w'_{k+1}w'_k w'_j \boldsymbol{w'}_i w'^*_h; && w'^2 = 0w'_{i+1}\boldsymbol{w'}_i w'_h \dots w'_1 w'_0;\\
&c^1 = c_{n+2}c_{n+1}c_n c_{n-1}\dots c_{k+1}c_k c_j c^*_i c^*_h; && c^2 = c_{i+2}c_{i+1}c_i c_h \dots c_1 c_0.
\end{aligned} \tag{4}$$

After cutting out $c^*_i c^*_h$ from c^1 and $c_{i+2}c_{i+1}$ from c^2, the remnants are joined into c. Therefore, a long input can be cut into a number of smaller inputs by employing the technique of copying the breakpoint and trimming the connection point. The smaller inputs are suited to the prepared adder and can be processed in the pipeline computation.

5.2 Joining several small inputs into one with an optimal length

If several of the input operands are shorter than half the length of the prepared adder, it is viable to join some of the small inputs into a larger one, which is a little shorter than the adder length. We call the larger input a digit chain. The skill required is explained below.

Assume there are pairs of small inputs a^3 and b^3, and other pairs of small inputs a^4 and b^4.

$$a^3 = a^3_p a^3_{p-1}\dots a^3_1 a^3_0; \quad a^4 = a^4_m a^4_{m-1}\dots a^4_1 a^4_0;$$
$$b^3 = b^3_p b^3_{p-1}\dots b^3_1 b^3_0; \quad b^4 = b^4_m b^4_{m-1}\dots b^4_1 b^4_0.$$

After transformation by T and W the intermediate results are

$$t^3 = t^3_p t^3_{p-1}\dots t^3_1 t^3_0 0; \quad t^4 = t^4_m t^4_{m-1}\dots t^4_1 t^4_0 0;$$
$$w^3 = 0w^3_p w^3_{p-1}\dots w^3_1 w^3_0; \quad w^4 = 0w^4_m w^4_{m-1}\dots w^4_1 w^4_0.$$

After transformation by T′ and W′ the intermediate results are

$$t^{3'} = t^{3'}_{p+1}t^{3'}_p t^{3'}_{p-1}\dots t^{3'}_1 t^{3'}_0 0; \quad t^{4'} = t^{4'}_{m+1}t^{4'}_m t^{4'}_{m-1}\dots t^{4'}_1 t^{4'}_0 0;$$
$$w^{3'} = 0w^{3'}_{p+1}w^{3'}_p w^{1'}_{p-1}\dots w^{3'}_1 w^{3'}_0; \quad w^{4'} = 0w^{4'}_{m+1}w^{4'}_m w^{4'}_{m-1}\dots w^{4'}_1 w^{4'}_0.$$

After transformation by T_2 the final results are

$$c^3 = c^3_{p+2}c^3_{p+1}c^3_p c^3_{p-1}\dots c^3_1 c^3_0; \quad c^4 = c^4_{m+2}c^4_{m+1}c^4_m c^4_{m-1}\dots c^4_1 c^4_0. \tag{5}$$

On the other hand, after appending two 0′s at the end of the a_0^3 and b_0^3 severally, one can join the a^4 and b^4 behind the a_0^3 and b_0^3 severally to form operands a and b as follows:

$$a = a_p^3 a_{p-1}^3 \dots a_1^3 a_0^3 00 a_m^4 a_{m-1}^4 \dots a_1^4 a_0^4;$$
$$b = b_p^3 b_{p-1}^3 \dots b_1^3 b_0^3 00 b_m^4 b_{m-1}^4 \dots b_1^4 b_0^4.$$

After transformation by T and W the intermediate results are

$$t = t_p^3 t_{p-1}^3 \dots t_1^3 t_0^3 00 t_m^4 t_{m-1}^4 \dots t_1^4 t_0^4 0;$$
$$w = 0 w_p^3 w_{p-1}^3 \dots w_1^3 w_0^3 00 w_m^4 w_{m-1}^4 \dots w_1^4 w_0^4.$$

After transformation by T′ and W′ the intermediate results are

$$t' = t_{p+1}^{3'} t_p^{3'} t_{p-1}^{3'} \dots t_1^{3'} t_0^{3'} 0 t_{m+1}^{4'} t_m^{4'} t_{m-1}^{4'} \dots t_1^{4'} t_0^{4'} 0;$$
$$w' = 0 w_{p+1}^{3'} w_p^{3'} w_{p-1}^{3'} \dots w_1^{3'} w_0^{3'} 0 w_{m+1}^{4'} w_m^{4'} w_{m-1}^{4'} \dots w_1^{4'} w_0^{4'}.$$

After transformation by T_2 the final results are

$$c = c_{p+2}^3 c_{p+1}^3 c_p^3 c_{p-1}^3 \dots c_1^3 c_0^3 c_{m+2}^4 c_{m+1}^4 c_m^4 c_{m-1}^4 \dots c_1^4 c_0^4. \tag{6}$$

By comparing eqs. (5) and (6), it is obvious that c^3 and c^4 can be obtained by cutting c between c_0^3 and c_{m+2}^4. c^4 is two bits longer than a^4 which comprises the rear digits of a. We call this technique "splice with double zeros, cut off the two surplus bits".

The data editing is invisible or transparent to the user because it is handled by a software module of the TOC's monitor. The module automatically edits the input operands, queues the operands in the MSD adder, and reversely edits the results.

6 Simulation experiments

Because the MSD adder of the TOC is still under construction, it is worthwhile simulating the tri-valued MSD adder, data editing, and pipeline computing. Recently, a simulation experiment was carried out at Shanghai University. The simulation results agree to some extent with the above theory. The crucial points of the simulation are discussed below.

6.1 Simulation project

1) Choice of simulation object: Because of the length restriction of this paper, a 15-bit MSD adder was chosen as the simulation object. Thus, the simulation program includes five transformation modules, that is, a 15-bit T, 15-bit W, 16-bit T′, 16-bit W′, and 17-bit T_2. These five modules constitute the MSD adder simulator. Accordingly, an array $D[79]$ is defined to store the results of the five transformation modules. $D[0]$ to $D[14]$ store the result of T, $D[15]$ to $D[29]$ the result of W, $D[30]$ to $D[45]$ the result of T′, $D[46]$ to $D[61]$ the result of W′, and $D[62]$ to $D[78]$ the result of T_2, which is the final result c.

2) Test case: The input data consists of the following: a pair of 8-bit operands, a pair of 15-bit operands, a pair of 3-bit operands, a pair of 3-bit operands, a pair of 4-bit operands, a pair of 18-bit operands, and a pair of 5-bit operands. The values of the input data are listed in Table 2.

6.2 Strategies of the simulation

The simulation program is written in Dev C^{++}. A file is used to input/output data. Arrays $D[79]$ and $weishu[10]$ are defined to store the results. Each $weishu[i]$ is used to record the data editing for the ith data. $weishu[i] = 0$ signifies that the ith data is unedited. $0 < weishu[i] < 15$ signifies that the ith data and its adjacent data are joined into a larger data, and the value of $weishu[i]$ is the bit-number of the ith data. $weishu[i] = 15$ signifies that the ith data is the lowest 15 bits of a larger operand.

Table 2 Test case

	1	2	3	4	5	6	7
a	1011 0101	101 1100 1001 1110	110	101	1010	10 1111 0111 0101 1011	1 1001
b	0100 1101	111 0110 0000 1111	101	011	1110	01 1000 0111 0110 0111	1 1100
c	01 0000 0010	0 1101 01$\bar{1}$1 $\bar{1}$010 111$\bar{1}$	0 110$\bar{1}$	0 1000	01 1000	0100 100$\bar{1}$ 1110 1100 0010	011 1$\bar{1}$1$\bar{1}$

$weishu[i] = 16$ signifies that the ith data is the intermediate 15 bits of a larger operand. $weishu[i] = 17$ signifies that the ith data is the most significant bits of a larger operand. In the pipeline computation, i is incremented by one at each step.

The editing strategy for input data must consider the following seven options:

1) If the length of the current data is less than 16, and the sum of the lengths of the current data and the next data is greater than 13, and $weishu[i-1] < 15$, the current data pair is sent into the adder and $weishu[i] = 0$.

2) If the total lengths of the current data and next data is less than 14, and $weishu[i-1] = 0$ or 17, the current data is the lowest section of a new interim digit chain. The bit-number of the current data is entered in $weishu[i]$.

3) If the total lengths of the current data and interim data chain is less than 14, and $weishu[i-1]$ is neither 0 nor 17, after appending two 0′s behind the current data, it is joined to the front of the interim data chain. The bit-number of the current data is entered in $weishu[i]$.

4) If the total lengths of the current data and interim data chain is greater than 13, and $weishu[i-1]$ is less than 15 and not 0, $weishu[i]$ is set to 0 and the interim data chain is sent into the Adder. The current data is tested again for compliance with strategy options 1, 2 and 5.

5) If the length of the current data is greater than 15, and $weishu[i-1]$ is 0, the 14th and 15th bits are copied into the 15th and 16th bits of the intermediate result. The lowest 15 bits of the larger data are sent into the adder as a new operand, and 15 is entered in $weishu[i]$. The remainder of the current data becomes the next current data.

6) If the length of the current data is greater than 15, and $weishu[i-1]$ is 15 or 16, the 14th and 15th bits are copied into the 15th and 16th bits of the intermediate result. The lowest 15 bits of the larger data are sent into the adder, and 16 is entered in $weishu[i]$. The remainder of the current data becomes the next current data.

7) If the length of the current data is less than 15, and $weishu[i-1]$ is 15 or 16, the data is sent into the adder, and 17 is entered in $weishu[i]$.

The editing strategy for the results must consider the following five options:

1) If $weishu[i] = 0$, c is the final result.

2) If $weishu[i]$ is neither 0 nor 15, a section of the result, with length $=weishu[i] + 2$, has been cut off from the rear of c. The section is a final result. After setting $i = i + 1$, the step is repeated until the remainder of c is null.

3) If $weishu[i] =15$, the highest two bits of c have been cut off. The remainder of c is the lowest bits of the interim result.

4) If $weishu[i] = 16$, the lowest two bits and highest two bits of c have been cut off. The remainder of c is joined to the previous interim result to form a new result.

5) If $weishu[i] = 17$, the lowest two bits of c have been cut off. The remainder of c is joined to the forepart of the interim result. Now, the interim result becomes the final result.

6.3 Simulation results

The simulation results are listed in row c of Table 2. They are all correct and data editing is a success. After analyzing array $D[79]$ for each computation, we can confirm that the pipeline computation in the MSD adder also functions correctly. Due to the length restriction of this paper, the analysis of $D[79]$ has been omitted.

7 Conclusions

Taking advantage of the carry-free nature of the MSD adder, we proposed several new methods for numerical calculations. Thanks to pipeline calculation techniques and the proposed data editing approach, the efficiency of the prepared MSD adder in the TOC has been greatly improved. The MSD adder proposed in this paper shows that the TOC can certainly make a significant contribution to those fields involving massive numerical calculations, and will establish a milestone in the maturity process thereof. In addition, the MSD adder lays a solid foundation in the area of multiplication, division, as well as matrix operation routines. Furthermore, the low power consumption of the liquid crystal intended for use in the TOC makes it a promising new green calculation dimension that will be available in the near future.

Acknowledgements

This work was supported by the Shanghai Leading Academic Discipline Project (Grant No. J50103), and the Doctorate Foundation of Education Ministry of China (Grant No. 20093108110016).

References

1 Caulfield H J, Vikram C S, Zavalin A. Optical logic redux. Optik, 2006, 117: 199–209
2 Wong W M, Blow K J. Design and analysis of an all-optical processor for modular arithmetic. Opt Commun, 2006, 265: 425–433
3 Jitendra N R, Dilip K G. Integrated all-optical logic and arithmetic operations with the help of a TOAD-based interferometer device-alternative approach. Appl Opt, 2007, 46: 5304–5310
4 Nishimuraa N, Awatsujib Y, Kubota T. Two-dimensional arrangement of spatial patterns representing numerical data in input images for effective use of hardware resources in digital optical computing system based on optical array logic. Parallel Distrib Comput, 2004, (64): 1027–1040
5 Nishimuea N, Awatsuj Y, Kubota T. Performance comparison and evaluation of options for arranging data in digital optical parallel computing. Opt Soc Japan, 2003, 4: 523–533
6 Jin Y, He H C, Lu Y T. Ternary optical computer principle. Sci China Ser F-Inf Sci, 2003, 46: 145–150
7 Jin Y. Management strategy of data bits in ternary optical computer. J Shanghai Univ (Nat Sci), 2007, 13: 519–523
8 Yan J Y, Jin Y, Zuo K Z. Decrease-radix design principle for carrying/borrowing free multi-valued and application in ternary optical computer. Sci China Ser F-Inf Sci, 2008, 51: 1415–1426
9 Jin Y, He H C, Ai L R. Lane of parallel through carry in ternary optical adder. Sci China Ser F-Inf Sci, 2005, 48: 107–116
10 Wang X C, Peng J J, Jin Y, et al. Vector-matrix multiplication based on ternary optical computer. In: The 2nd International Conference on High Performance Computing and Application (HPCA2009), 10-12th, August 2009, Shanghai China, LNCS 5938, 2010. 426–432
11 Teng L, Peng J J, Jin Y, et al. A cellular automata calculation model based on ternary optical computer. In: The 2nd International Conference on High Performance Computing and Application, HPCA2009, 10-12th, August 2009, Shanghai China, LNCS 5938, 2010. 377–383
12 Zhan X Q, Peng J J, Jin Y, et al. Static allocation strategy of data-bits of terry optical computer. J Shanghai Univ (Nat Sci), 2009, 15: 528–533
13 Avizienis A. Signed-digit number representations for fast parallel arithmetic. IRE Trans Electron Comp, 1961, EC: 389–400
14 Ha B, Li Y. Parallel modified signed-digit arithmetic using an optoelectronic shared content-addressable-memory processor. Appl Opt, 1994, (33): 3647–3662
15 Casasent D, Woodford P. Symbolic substitution modified signed-digit optical adder. Appl Opt, 1994, (33): 1498–1506
16 Li G Q, Qian F, Ruan H, et al. Compact parallel optical modified-signed-digit arithmetic-logic array processor with electron-trapping device. Appl Opt, 1999, (38): 5038–5045