Binary addition and multiplication in cellular space

By E. KATONA

Cellular automata are highly parallel bitprocessors, so they are suitable for the bitparallel execution of distinct computational tasks. In this paper powerful bitparallel algorithms are given for fixed point binary addition and multiplication, taking into account the cellprocessor architecture developed by T. LEGENDI [1]. For this architecture there have been constructed more then 100 cellular algorithms solving different computational tasks [6]. In a large cellular space a high number of cellular adders, multipliers and other processing elements may be embedded, and more complex tasks may be computed in parallel, as matrix multiplication [4], certain data processing tasks [5], etc.

1. Introduction

A cellular automaton is a highly parallel processor, but the economical programming of such a processor is not an easy task. If *macro-cells* are applied (a cell works as a microprocessor), then the programming of the cellular structure is somewhat easier [7], but the architecture has lower flexibility (fixed operations, fixed word length, etc.) and in general the bitparallel execution of the operations is impossible.

If *micro-cells* are applied (having maximum 16 states) with variable transition functions, then the cellprocessor has high flexibility and a totally bitparallel processing is possible. In [3], [4], [5], [6] and in this paper it is shown that a cellprocessor consisting of micro-cells is economically programmable, and the speed of the cellular algorithms is *wordlength-independent* in most cases.

The cellprocessor architecture proposed in [1] is based on the micro-cell conception, and has — from the point of view of this paper — the following characteristic properties:

(i) The cellular space is a *two-dimensional rectangle-form cell-matrix* which is bounded by *dummy-cells* (the dummy cells have no transition funtion, but their states can be set from the outside world). In the cellular net the *von Neumann neighbourhood* is assumed.

(ii) *The cells do not have a fixed transition function,* but receive commands (microinstructions) from a central control (CCPU), and arbitrary local transition function may be realized by the execution of a certain sequence of microinstructions. This implies that the cellprocessor can work with an *arbitrary local transition function,* and — moreover — it can work with *time-varying transition function.*

(iii) The cellular space is *inhomogeneous,* that is, the individual cells may work with different transition functions at the same time. To ensure this property, each cell has an *internal state.* The cells having different internal states may work with different transition functions. So, if there are n different internal states, then maximum n different transition functions may work in parallel. The internal states are set at $t=0$, and during the working of the cellprocessor they are unchanged.

The transition functions will be defined according to [2] by *microconfiguration terms.* A microconfiguration term has the form:

the state of a group of cells at time t	→	the required state of (another) group of cells at time $t+1$

Each cell on the right side occurs on the left side, too, and is marked by double frame for the identification. Because of the inhomogeneity a microconfiguration term may describe more transition functions together.

The notation $[x_k x_{k-1} \ldots x_1]$ will be used often in the text, which means a k-digit binary number having the digits $x_k, x_{k-1}, \ldots, x_1$ $(x_i \in \{0, 1\})$.

2. Binary addition

Binary addition is the most fundamental arithmetic operation. The cellular algorithm described below is applied in many further cellular processing elements (see the cellular multiplier in this paper, and [4], [5], [6]).

The cellular binary addition is based on the "carry save" addition algorithm. Let $x=[x_k \ldots x_1]$, $y=[y_k \ldots y_1]$ and $z=[z_k \ldots z_1]$ be binary numbers of k digits to be added. In the first step x and y are added *in a parallel way:* a (partial) sum $s=[s_k \ldots s_1]$ and a carry vector $c=[c_k \ldots c_1]$ is computed as follows

$$[c_i s_i] := x_i + y_i \qquad \text{for any } i. \tag{1}$$

In the second step the number z can be added to s and c by the formula

$$[c_i' s_i'] := z_i + s_i + c_{i-1} \qquad \text{for any } i. \tag{2}$$

(The sign $'$ serves for the distinction between the old and new values of s and c.)

If there are more numbers to be added, then they can be added to s and c also by formula (2). The complete sum of the operands should be computed from the last s and c in $k-1$ steps applying the formula

$$[c_i' s_i'] := s_i + c_{i-1} \qquad \text{for any } i. \tag{3}$$

On the basis of the described parallel addition algorithm it is easy to construct a cellular automaton for binary addition. It consists of k adder cells, each containing a sum bit S and a carry bit C (4-state cells). A dummy cell is connected to each adder cell as upper neighbour (Fig. 1).

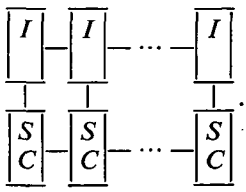
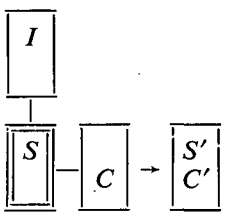


Fig. 1

At $t=0$ the bits S and C are 0, and the bits I contain the first number to be added. In any further step a new number will be written into the bits I and the adder cells work with the transition function:



where $[C'S'] = S+C+I.$

After the input of the last operand the dummy cells are set into 0 and after $k-1$ steps the complete sum of the operands is computed in the bits S of the cell-row. (In this way the above transition function includes the formulas (1), (2), (3).)

The addition of n numbers each consisting of k bits, needs $n+k-1$ *steps*, so the parallel addition algorithm is economical for many operands.

Remark. To prevent the overflow, for n operands a cellular adder consisting of $k+\log_2 n$ cells should be used. If only k cells are applied, then the leftmost cell needs a special overflow-watching transition function (inhomogeneity).

The above cellular adder has many simple applications, as the binary counter, the computation of certain number-rows (e.g. Fibonacci-numbers), vector addition, etc. [6]; but the most important application is the binary multiplication discussed in the next point.

3. The multiplication of two binary numbers

The cellular multiplication algorithm is based, as usual, on the addition: the partial products will be generated in a special cell-row, and another cell-row under it works as an adder (Fig. 2).

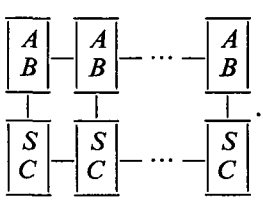


Fig. 2

The partial products are generated in an overlapped manner. Between the digits of the multiplicand $a=[a_k \dots a_1]$ and the multiplier $b=[b_k \dots b_1]$ zero digits are inserted, and in such a form they move step by step one against another in the upper cell-row (Fig. 3).

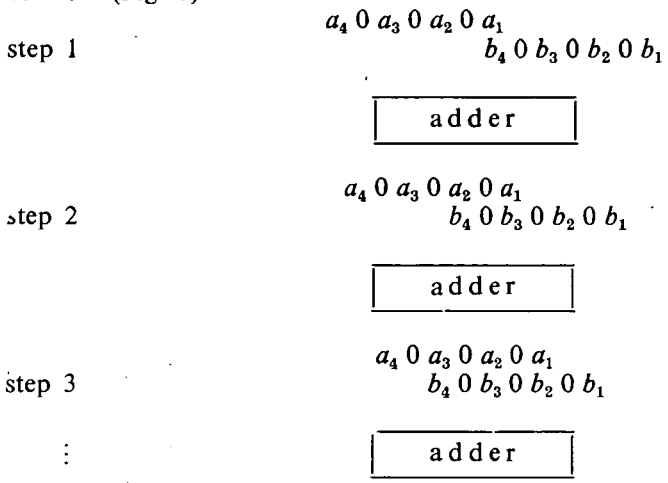
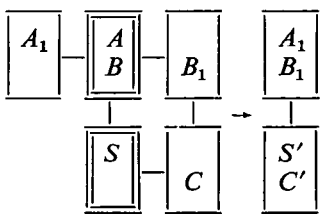


Fig. 3
Cellular algorithm for binary multiplication in the case $k=4$.

The products of the operand digits staying on the same position are summed by the adder (on Fig. 3 in the first step a_1b_4, in the second step a_2b_4 and a_1b_3 are summed). Fig. 3 shows well that in steps 1, 2, 3 and 4 the bit b_4 is multiplied by a_1, a_2, a_3 and a_4, thus the partial product $[a_4a_3a_2a_1] \cdot b_4$ is generated for the adder. The partial products corresponding to b_3, b_2 and b_1 are computed in a similar way, and each is created on the appropriate position.

The two rows of the cellular multiplier have distinct transition functions, which may be defined together as follows:



where $[C'S'] = S+C+A \cdot B$.

If k-bit numbers are multiplied, then the product has $2k$ bits, therefore an adder of length $2k$ should be used. Thus the multiplier needs $4k$ 4-*state cells*.

If at $t=0$ the configuration of Fig. 3 (step 1) is assumed, then at $t=2k-1$ all the partial products are generated. It is easy to see that at $t=2k$ the rightmost k cells of the adder have zero carry bits. Therefore to compute the complete product further k steps are needed, thus the whole multiplication process uses $3k$ steps.

Remark. If between the digits of a and b the digits of further two k-bit numbers x and y are written (instead of the zeros), then the multiplier computes the expression $a \cdot b + x \cdot y$! The cellular multiplier may be used for vector-multiplication in a similar way [4].

4. Multiplication of more then two numbers

In this section a cellular algorithm is given to compute the product $x_1 \dots x_n$ where x_i is a k-bit number and $0 \leqq x_i < 1$ holds for any i (the leftmost digit of x_i has the positional value 2^{-1}). To solve this task the cellular multiplier of section 3 will be modified: 3-bit cells (i.e. 8-state cells) will be used where the third bits in the adder cells serve for control (Fig. 4).

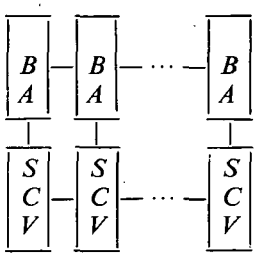
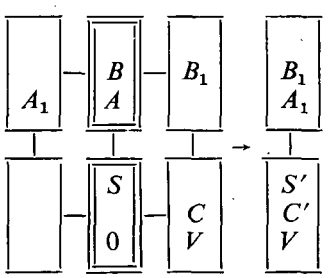


Fig. 4
Cellular multiplier for more then two numbers. The control bits are marked by *V*.

At $t=0$ the number x_1 is stored in the bits "S" of the adder. The numbers $x_2, \dots, x_n$ come from the outside world and go left on the bits "B". Before each number x_i a control signal of value 1 is sent, which goes left on the control bits and copies the bits "S" into the bits "A" (at the same time the adder is cleared). Thus the number x_i coming from the outside world is multiplied by the product $x_1 \cdot \dots \cdot x_{i-1}$, and the process may be repeated until it is necessary.

According to the above principle, the transition functions of section 3 should be modified as follows.

If the adder cell contains a control signal 0:

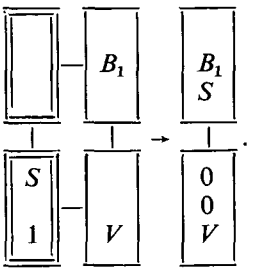


where $[C'\,S'] = S + C + A \cdot B.$

Listing

```
STEP  0:  .  .  .  .  .  .  .  .  .
          .  .  .  .  .  .  .  .  .
          1  0  0  1  0  0  0  0  0
          .  .  .  .  .  .  .  .  <

STEP  1:  .  .  .  .  .  .  .  .  .
          .  .  .  .  .  .  .  .  0
          1  0  0  1  0  0  0  0  0
          .  .  .  .  .  .  .  <  .

STEP  2:  .  .  .  .  .  .  .  .  .
          .  .  .  .  .  .  .  0  .
          1  0  0  1  0  0  0  0  0
          .  .  .  .  .  .  <  .  .

STEP  3:  .  .  .  .  .  .  .  .  1
          .  .  .  .  .  .  0  .  0
          1  0  0  1  0  0  0  0  0
          .  .  .  .  .  <  .  .  .

STEP  4:  .  .  .  .  .  .  .  1  .
          .  .  .  .  .  0  .  0  .
          1  0  0  1  0  0  0  0  0
          .  .  .  .  <  .  .  .  .

STEP  5:  .  .  .  .  .  .  1  .  1
          .  .  .  .  1  .  0  .  0
          1  0  0  0  0  0  0  0  0
          .  .  .  <  .  .  .  .  .

STEP  6:  .  .  .  .  .  1  .  1  .
          .  .  .  0  .  1  .  0  .
          1  0  0  0  0  0  0  0  0
          .  .  <  .  .  .  .  .  .

STEP  7:  .  .  .  .  1  .  1  .  0
          .  .  0  .  0  .  1  .  0
          1  0  0  0  0  1  0  0  0
          <  .  .  .  .  .  .  .  .

STEP  8:  .  .  .  1  .  1  .  0  .
          .  1  .  0  .  0  .  1  .
          0  0  0  0  0  1  1  0  0
          .  .  .  .  .  .  .  .  .

STEP  9:  .  .  1  .  1  .  0  .  1
          .  .  1  .  0  .  0  .  1
          0  0  0  0  0  1  1  0  0
          .  .  .  .  .  .  .  .  .

STEP 10:  1  .  .  1  .  0  .  1  .
          .  .  .  1  .  0  .  0  .
          0  1  0  0  0  1  1  0  1
          .  .  .  .  .  .  .  .  <

STEP 11:  .  1  .  .  0  .  1  .  .
          .  .  .  .  1  .  0  .  1
          0  1  1  0  0  1  1  0  0
          .  .  .  .  .  .  .  <  .

STEP 12:  1  .  0  .  .  1  .  .  .
          .  .  .  .  .  1  .  0  .
          0  1  1  0  0  1  1  0  0
          .  .  .  .  .  .  <  .  .

STEP 13:  .  0  .  1  .  .  .  .  1
          .  .  .  .  .  .  1  .  0
          0  1  1  0  2  0  0  0  0
          .  .  .  .  .  <  .  .  .

STEP 14:  0  .  1  .  .  .  .  1  .
          .  .  .  .  .  0  .  1  .
          0  1  1  1  0  0  0  0  0
          .  .  .  .  <  .  .  .  .

STEP 15:  .  1  .  .  .  .  1  .  1
          .  .  .  1  .  0  .  1  .
          0  1  1  0  0  0  1  0  0
          .  .  .  <  .  .  .  .  .

STEP 16;  1  .  .  .  .  1  .  1  .
          .  .  .  1  .  1  .  0  .
          0  1  0  0  0  0  1  1  0
          .  .  <  .  .  .  .  .  .

STEP 17:  .  .  .  .  1  .  1  .  1
          .  .  1  .  1  .  1  .  0
          0  0  0  0  1  0  1  1  0
          <  .  .  .  .  .  .  .  .

STEP 18:  .  .  .  1  .  1  .  1  .
          0  .  1  .  1  .  1  .  .
          0  0  0  1  1  1  1  1  0
          .  .  .  .  .  .  .  .  .

STEP 19:  .  1  .  1  .  1  .  .  0
          .  0  .  1  .  1  .  .  1
          0  0  1  1  2  1  2  1  0
          .  .  .  .  .  .  .  .  .

STEP 20:  1  .  1  .  1  .  0  .  .
          .  .  0  .  1  .  1  .  .
          0  0  1  3  0  3  0  1  0
          .  .  .  .  .  .  .  .  .

STEP 21:  .  1  .  1  .  0  .  .  .
          .  .  .  0  .  1  .  1  .
          0  0  2  1  2  1  0  1  0
          .  .  .  .  .  .  .  .  .

STEP 22:  1  .  1  .  0  .  .  .  .
          .  .  .  .  0  .  1  .  .
          0  1  0  2  0  1  0  1  0
          .  .  .  .  .  .  .  .  .

STEP 23:  .  1  .  0  .  .  .  .  .
          .  .  .  .  .  0  .  1  .
          0  1  1  0  0  1  0  1  0
          .  .  .  .  .  .  .  .  .
```

If the adder cell contains a control signal 1:



The multiplication process is demonstrated on a simulation example (see Listing). The product of $x_1=0.1001$, $x_2=0.1101$ and $x_3=0.1110$ will be computed by an 8-bit multiplier. The multiplier is displayed in 4 rows, according to Fig. 4, but in the third row the bits S and C are printed together in the form $[CS]$ (that is, for example the value 2 means $C=1$ and $S=0$). The points mean insignificant zeros in each row.

At $t=0$, x_1 is stored in the adder, and a control signal marked by "$<$" starts on the right end of the multiplier. Between $t=1$ and $t=8$ the number x_1 is copied into the bits "A" and it is shifted right (hereby zeros are inserted between the digits). The number x_2 comes from outside and will be multiplied by x_1. At $t=10$ the rightmost digit of x_1x_2 is computed. Already at this moment a new control signal may be started which ensures the multiplication of x_1x_2 by x_3, thus an overlapping is possible between the consecutive multiplications.

For the multiplication of n numbers $(2k+2)(n-1)+2k \approx 2kn$ *steps* are required, and the modified multiplier consists of $4k$ 8-*state cells*. The product contains $2k$ digits (the leftmost digit has the positional value 2^{-1}) and the first $?k-\log_2 k-\log_2 n$ bits are always correct.

5. Concluding remarks

In this paper three fundamental *cellular processing elements* have been discussed, *each designed for the same cellprocessor architecture* [1]. Each processing element is based on a *bitparallel cellular algorithm* where nearly all cells work effectively in each time-step. By the interconnection of such simple processing elements more complex tasks may be solved in bitparallel by a cellprocessor.

RESEARCH GROUP ON THEORY OF AUTOMATA
HUNGARIAN ACADEMY OF SCIENCES
SOMOGYI U. 7.
SZEGED, HUNGARY
H-6720

References

]1] LEGENDI, T., Cellprocessors in computer architecture, *Computational Linguistics and Computer Languages*, v. 11, 1977, pp. 147—167.
[2] LEGENDI, T., A 2D transition function definition language for a subsystem of the CELLAS cellular processor simulation language, *Computational Linguistics and Computer Languages*, v. 13, 1979, pp. 169—194.
[3] KATONA, E., T. LEGENDI, Cellular algorithms for fixed point decimal addition and multiplication, *Elektron. Informationsverarb. Kybernet.*, v. 17, 1981, pp. 637—644.
[4] KATONA, E., Cellular algorithms for fixed point vector- and matrix-multiplication, *Proceedings of the Conference Programming Systems'* 81, pp. 262—280, in Hungarian.
[5] KATONA, E., The application of cellprocessors in conventional data processing, *Proceedings of the Third Hungarian Computer Science Conference*, Publishing House of the Hungarian Academy of Sciences, Budapest, 1981, pp. 295—306.
[6] KATONA, E., Cellular algorithms (Selected results of the cellprocessor team led by T. Legendi), Von Neumann Society, Budapest, 160 pages in Hungarian, 1981
[7] DOMÁN, A., A 3-dimensional cellular space, *Sejtautomaták*, Gondolat Kiadó, Budapest, 1978, in Hungarian.
[8] VOLLMAR, R., *Algorithmen in Zellularautomaten*, B. G. Teubner, Stuttgart, 1979.
[9] NISHIO, H., Real time sorting of binary numbers by 1-dimensional cellular automaton, *Proceedings of the International Symposium on Uniformly Structured Automata and Logic*, Tokyo, 1975, pp. 153—162.

(Received April 10, 1981)