Journal of Discrete Algorithms 7 (2009) 267–277

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Boolean circuit programming: A new paradigm to design parallel algorithms ☆,☆☆

Kunsoo Park [a,*], Heejin Park [b], Woo-Chul Jeun [a], Soonhoi Ha [a]

[a] School of Computer Science and Engineering, Seoul National University, Republic of Korea
[b] College of Information and Communications, Hanyang University, Republic of Korea

ARTICLE INFO

Article history:
Received 21 March 2007
Received in revised form 16 July 2008
Accepted 8 August 2008
Available online 31 January 2009

Keywords:
Parallel algorithms
Parallel models
Boolean circuit
Description language

ABSTRACT

The Boolean circuit has been an important model of parallel computation, but not many parallel algorithms have been designed on this model because it is 'awkward to program.' To overcome this drawback, we propose a description language for designing parallel algorithms on the Boolean circuit. This description language is to parallel algorithms what the pseudo-code is to sequential algorithms. Through example codes, we show that the description language is a convenient tool to design parallel algorithms due to its general iterative and recursive structures and the ease of modular design.

1. Introduction

Numerous parallel algorithms have been developed on various models of parallel computation, which can be roughly classified into the following three.

1. Network models such as mesh and hypercube [17]: These network models can be actually used as the architectures of parallel machines. However, a parallel algorithm designed for a network model is usually too specific to the fixed connection that the network model has. Also a network model itself imposes some constraints on problem complexity. For example, sorting in an $n \times n$ mesh requires $\Omega(n)$ time, and a similar result applies to a three-dimensional mesh, which are widely used topologies in massively parallel processors [14].
2. PRAM [12,15]: PRAM has been the most popular model to design and analyse parallel algorithms, but it is too idealistic in that all processors work synchronously and interprocessor communication is free [10]. Hence, some problems have $o(\log n)$ time complexity on PRAM, which is unrealistic [25]. In fact, many parallel algorithms of $o(\log n)$ time have been developed on PRAM (e.g., [8]), and in this sense PRAM is quite a misleading model.
3. Bridging models such as BSP [27] and LogP [10]: These models have been proposed as ones to bridge the gap between parallel programs and actual machines. They are excellent bridging models for existing parallel machines, but they may not be appropriate in characterizing parallel complexity of problems just as a most important notion, NC, in parallel computation cannot be well grasped in existing parallel machines [13].

☆ A preliminary version of this paper appeared in the 15th Australasian Workshop on Combinatorial Algorithms 2004.
☆☆ This work was supported by Korea Research Council of Fundamental Science and Technology.
* Corresponding author.
E-mail addresses: kpark@theory.snu.ac.kr (K. Park), hjpark@hanyang.ac.kr (H. Park), wcjeun@iris.snu.ac.kr (W.-C. Jeun), sha@iris.snu.ac.kr (S. Ha).

1570-8667/$ – see front matter
doi:10.1016/j.jda.2008.08.004

Another important model of parallel computation is the Boolean circuit [29,30]. Uniform Boolean circuits have long been considered as a parallel computation model [23], and the classes NC^i for $i \geqslant 1$ and NC are defined by them. Sipser [24] describes the following advantages and disadvantages of the Boolean circuit.

- It is simple and realistic.
- It is 'awkward to program' because individual processors (AND, OR, NOT) are so weak.

The latter is the main reason why not many parallel algorithms have been designed on this model.

When we talk about practice in regard to parallel algorithms, there are two different fields: one is parallel computing in existing parallel machines where each processor is quite powerful [7,11], and the other is parallel programming in logic circuits where the standard set of gates is AND, OR, NOT. In the second field 'programming logic circuits' using VHDL [4] and Verilog [6,28], which is inherently parallel programming, is a main task for hardware designers. However, parallel algorithms, except for basic problems such as multiplication and FFT [5,16,18], that are readily available to hardware designers are very scarce. Hence, there is a need from the area of hardware design to develop parallel algorithms for various problems by using logic circuits.

In this paper we will use the Boolean circuit as the model of parallel algorithm design, and present a way to overcome its drawback (awkward to program). We propose a description language like VHDL [4] for designing parallel algorithms, which we call the BC (Boolean Circuit) language. We will explain the language features of BC by giving example codes for the problems of counting and sorting. Although VHDL and Verilog also describe Boolean circuits, they are very complicated languages to describe all physical behaviors of hardware logic circuits. The language BC is such a simplified version of VHDL that only features necessary for algorithm design are retained. Some features such as recursion are generalised in BC to facilitate algorithm description. Still, the mapping from the BC language to VHDL is straightforward.

Our work has analogies with algorithm design in sequential computation. The model of sequential computation is the 'random access machine' (RAM) [1,23], and its instruction set is in a level of assembly languages. Thus, designing an algorithm with RAM's instruction set will be painful. Fortunately, most algorithms are described in pseudo-codes [9], because any codes in a high-level language can be translated to assembly codes by compilers and this process is well understood [3]. Similarly, VHDL-like codes can be translated to logic circuits by high-level synthesizers [19] and even to silicon layouts by silicon compilers [26]. In fact, the codes in this paper were synthesized to logic circuits by VHDL synthesis tools [19,26].

Our approach to parallel computation meets the needs from both theory and practice. It provides a convenient tool (description language) to parallel algorithm designers, and any parallel algorithm designed in this description language can be automatically translated to logic circuits, which means that this approach is very realistic. Our approach puts forward the Boolean circuit not just as a model to study complexity classes but also as a model to design parallel algorithms.

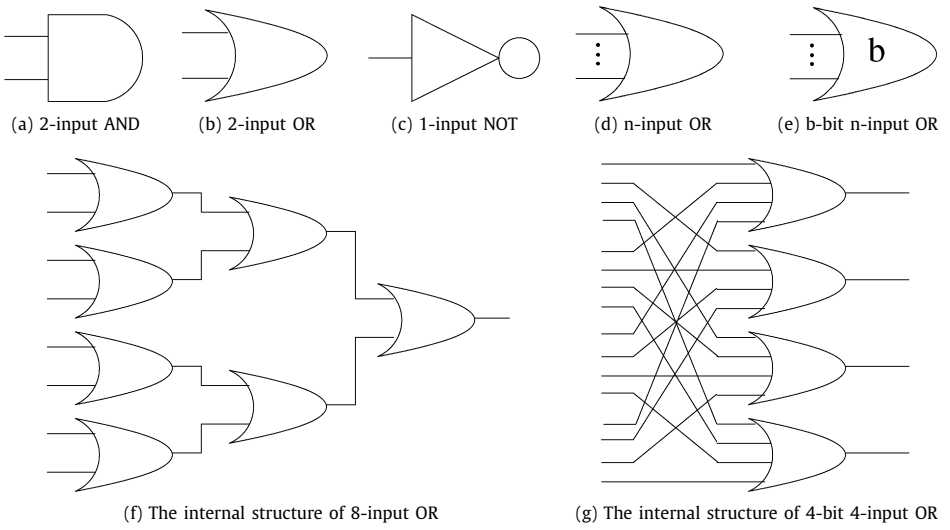
The rest of the paper is organized as follows. In Section 2, we describe basic Boolean circuits such as MUX, DEMUX, and Adder. In Section 3, we introduce the basic features of the BC language. In Sections 4–6, we show the Boolean circuits for comparison, counting, and sorting, and present their descriptions using BC. We conclude with some remarks in Section 7.

2. Preliminaries

A *Boolean circuit* consists of inputs, outputs, logic gates, and directed wires. The inputs, outputs and logics gates are connected by the wires. If we consider the inputs, outputs, and logic gates as nodes and the wires as edges, a boolean circuit can be represented as a graph. A *combinatorial Boolean circuit* is a Boolean circuit that can be represented as a directed acyclic graph, i.e., there are no cycles in the graph (circuit). Since we only consider the combinatorial Boolean circuit in this paper, we will say just 'circuit' instead of combinatorial Boolean circuit. We define the *depth* and the *size* of a circuit. The depth of a circuit is the number of logic gates in a longest path from an input to an output. The depth of a circuit is well-defined because the circuit is acyclic, and it corresponds to the worst-case running time of the circuit. The size of a circuit is the number of inputs, outputs, and logic gates in it. Note that the number of wires is proportional to the number of logic gates and outputs because the number of inputs to a logic gate is either one or two and the number of wires connected to an output is one.

We describe some circuits that are used as building blocks in composing more complicated circuits. The basic circuits are logic gates (AND, OR, NOT) and additional components are multiplexers (MUX), demultiplexers (DEMUX), adders (ADD), and carry save adders (CSADD).

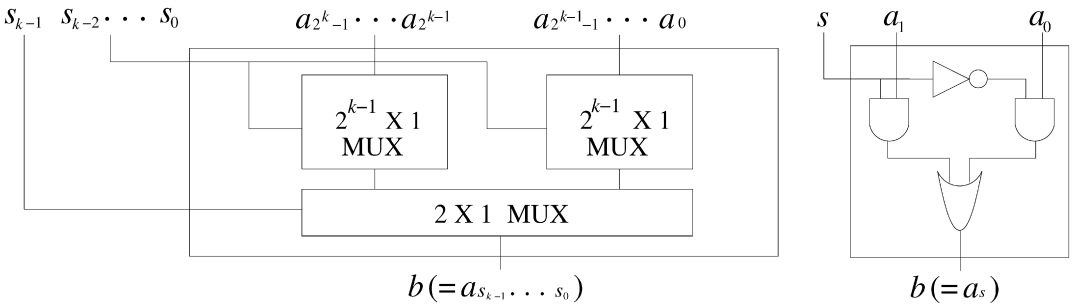
- Logic gates (AND, OR, NOT): Normally, an AND (resp. OR) gate has two input wires (Fig. 1(a) and (b)) and a NOT gate has one input wire (Fig. 1(c)). However, we allow the AND gate and the OR gate to have n input wires for arbitrary integer $n \geqslant 2$ to simplify the description of circuits (Fig. 1(d)). Actually, the n-input OR gate is a circuit composed of $n-1$ 2-input OR gates that are connected in a tree-like fashion (Fig. 1(f)). The internal structure of an n-input AND gate is similar. Thus, the depth of the n-input OR (AND) gate is $O(\log n)$ and its size is $O(n)$. Furthermore, we allow the OR gate and the AND gate to have b-bit integers as inputs and output (Fig. 1(e)). A b-bit n-input OR gate is a circuit composed of b n-input OR gates such that each n-input OR gate gets the ith bits, $0 \leqslant i \leqslant b-1$, of n input integers as



(a) 2-input AND (b) 2-input OR (c) 1-input NOT (d) n-input OR (e) b-bit n-input OR

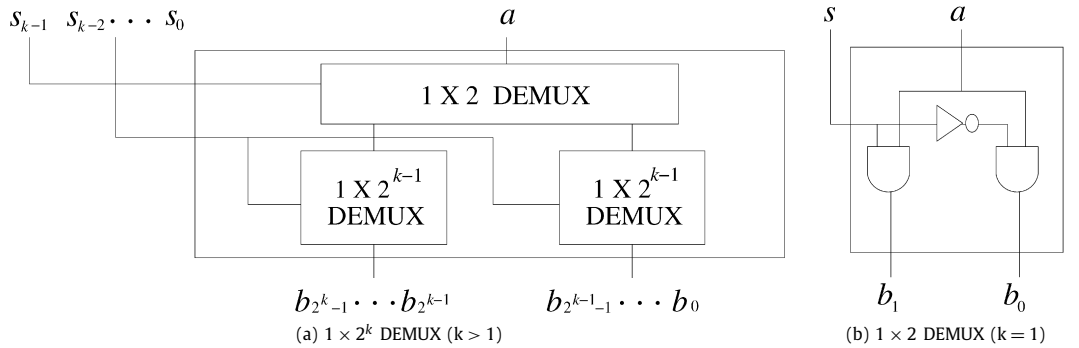
(f) The internal structure of 8-input OR (g) The internal structure of 4-bit 4-input OR

Fig. 1. Gates.



(a) $2^k \times 1$ MUX (k > 1) (b) 2×1 MUX (k = 1)

Fig. 2. MUX.



(a) 1×2^k DEMUX (k > 1) (b) 1×2 DEMUX (k = 1)

Fig. 3. DEMUX.

inputs and it produces the ith bit of the b-bit output (Fig. 1(g)). The internal structure of a b-bit n-input AND gate is similar. Since a b-bit n-input OR (AND) gate is composed of b n-input OR (AND) gates that performs in parallel, the depth of b-bit n-input OR (AND) gate is $O(\log n)$ and its size is $O(bn)$.

- MUX (Fig. 2): The $2^k \times 1$ MUX gets two bit strings $s_{k-1}\cdots s_0$ and $a_{2^k-1}\cdots a_0$ as inputs and it outputs a bit a_x such that the binary representation of x corresponds to $s_{k-1}\cdots s_0$. One can show by induction that the depth of $2^k \times 1$ MUX is $O(k)$ and its size is $O(2^k)$.
- DEMUX (Fig. 3): The 1×2^k DEMUX gets a bit string $s_{k-1}\cdots s_0$ and a bit a as inputs and it outputs a to b_x such that the binary representation of x corresponds to $s_{k-1}\cdots s_0$. The depth and the size of 1×2^k DEMUX are $O(k)$ and $O(2^k)$, respectively.

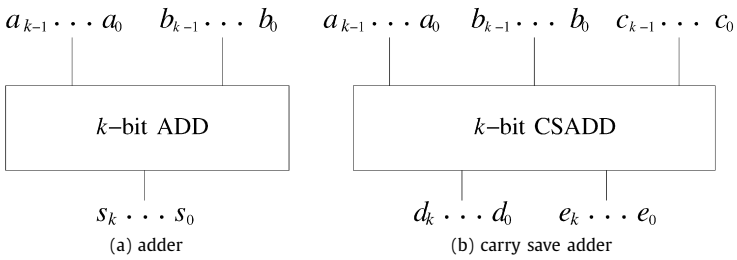


Fig. 4. Adder and carry save adder.

- A b-bit MUX (DEMUX): The b-bit $2^k \times 1$ MUX is the same as $2^k \times 1$ MUX except that all inputs and outputs are b-bit integers. The b-bit $2^k \times 1$ MUX consists of b $2^k \times 1$ MUXs such that each $2^k \times 1$ MUX multiplexes each bit of the b-bits in parallel. Thus, the depth and the size of the b-bit $2^k \times 1$ MUX are $O(k)$ and $O(b \cdot 2^k)$, respectively. Similarly, the b-bit 1×2^k DEMUX consists of b 1×2^k DEMUXs, and its depth and size are $O(k)$ and $O(b \cdot 2^k)$, respectively.
- Adder (Fig. 4(a)): The k-bit adder gets two k-bit integers $a_{k-1} \cdots a_0$ and $b_{k-1} \cdots b_0$ as inputs and it outputs a $(k+1)$-bit integer $s_k \cdots s_0$ such that $s_k \cdots s_o = a_{k-1} \cdots a_0 + b_{k-1} \cdots b_0$. The depth of the k-bit adder is $O(\log k)$ and its size is $O(k)$ [9].
- Carry save adder (Fig. 4(b)): The k-bit carry save adder gets three k-bit integers $a_{k-1} \cdots a_0$, $b_{k-1} \cdots b_0$, and $c_{k-1} \cdots c_0$ as inputs and it outputs two $(k+1)$-bit integers $d_k \cdots d_0$ and $e_k \cdots e_0$ such that $d_k \cdots d_o + e_k \cdots e_o = a_{k-1} \cdots a_0 + b_{k-1} \cdots b_0 + c_{k-1} \cdots c_0$. The depth of the k-bit carry save adder is $O(1)$ and its size is $O(k)$ [9].

3. The BC language

We explain how to describe Boolean circuits using the language BC. The basic features of BC are essentially the same as those of VHDL with some exceptions. Those features of VHDL that are not directly related to algorithm design were eliminated or simplified. Some features such as recursion were generalised to facilitate algorithm description. We first introduce the basic rules to describe the circuit elements and the interconnections between them, and then advanced features that are necessary to describe the iterative structures in circuits.

The basic rules to describe circuit elements (logic gates, components, wires) and the interconnections between them are as follows.

- A wire is represented as a variable such as A and B. A set of wires is represented by an array of variables such as A(0..7) and C(0..$n-1$)(0..$b-1$). For brevity, an omitted index in an array means the whole range, e.g., C for C(0..$n-1$)(0..$b-1$) and C(1) for C(1)(0..$b-1$).
- Logic gates are represented as operators such as AND, OR, and NOT.
- The interconnection between wires and logic gates are described using a set of assignment statements. For example, if two wires A and B are connected to the inputs of an AND gate and a wire C is connected to the output of the AND gate, the description of the interconnection is 'C ← A AND B.'
- Components are represented as functions. A function for a component is composed of two parts. In the first part, we define the wires (variables) associated with the component. The input and output wires of the component are defined using type identifier **in** and **out**, respectively. The wires inside the component are defined using type identifier **signal**. In the second part that is closed by **begin** and **end**, we describe the internal structure of the component. For example, a function for the 2×1 MUX (Fig. 2(b)), named `2X1MUX`, is as follows.

```
2X1MUX
in:  A(0..1), S;
out:  B;
signal:  T(0..1);
begin
     T(0) ← (NOT S) AND A(0);
     T(1) ← S AND A(1);
     B ← T(0) OR T(1);
end
```

- Some functions may have a varying number of elements in an array for inputs and/or outputs. For example, the function for the b-bit $2^k \times 1$ MUX (Fig. 2(a)) should have $b \cdot 2^k + k$ elements in its input arrays and b elements in its output array. To describe a function like this, we use keyword **generic**. Keyword **generic** indicates the use of special parameters such that the numbers of elements in arrays for inputs and outputs are represented as mathematical expressions of the

special parameters. Function MUX for describing the b-bit $2^k \times 1$ MUX is as follows. In function MUX, k and b are defined as special parameters and the numbers of inputs and outputs are defined using k and b.

```
MUX generic (k, b)
in:  A(0..2^k − 1)(0..b − 1), S(0..k − 1);
out: B(0..b − 1);
signal: T(0..1)(0..b − 1);
begin
if ( k = 1 ) {
     pfor i from 0 to b − 1 {
          T(0)(i) ← (NOT S(0)) AND A(0)(i);
          T(1)(i) ← S(0) AND A(1)(i);
     }
     ORF generic (1, b) (T(0..1), B);
}
else {
     MUX generic (k − 1, b) (A(0..2^(k−1) − 1), S(0..k − 2), T(0));
     MUX generic (k − 1, b) (A(2^(k−1)..2^k − 1), S(0..k − 2), T(1));
     MUX generic (1, b) (T(0..1), S(k − 1), B);
}
end
```

- In the above function MUX, keywords **if** and **else** are used. In this case, the keywords are used to describe a conditional structure that is dependent on special parameters indicated by **generic**. The keywords are also used when describing parallel or sequential iterative structures using recursion, which will be explained later.
- The interconnection between a component and wires connected to the component are represented as a function call to the function for the component where the parameters of the function call are variables representing the wires connected to the component. For example, if A(0), A(1), and S (S is for selection) are inputs to a 2×1 MUX and B is its output, the description of the interconnection is a function call `2X1MUX (A(0..1), S, B)`.

We show how to describe iterative structures in circuits. In an iterative structure, a similar structure appears multiple times with small regular variations. The iterative structures are divided into two categories. They are parallel iterative structures and sequential iterative structures. In a parallel iterative structure, there are no wires connecting the structures constituting the parallel iterative structure. In a sequential iterative structure, there are. The parallel and sequential iterative structures are not distinguished in VHDL, but we distinguish them in BC to enhance readability of codes. We first show how to describe parallel iterative structures and then sequential iterative structures.

- Parallel iterative structures appear in the 8-input OR gate in Fig. 1(f). The 8-input OR gate is a circuit composed of seven 2-input OR gates that are connected in a tree-like fashion: The 8-input wires are connected to four 2-input OR gates, then the four output wires of the four 2-input OR gates are connected to two 2-input OR gates, and finally the two output wires are connected to an 2-input OR gate. Thus, there are three parallel iterative structures in the 8-input OR gate which are composed of four, two, and one 2-input OR gates, respectively. To describe the parallel iterative structure, we use keyword **pfor**. Let us assume that 8 input wires are represented as A(i) for $0 \leqslant i \leqslant 7$ and 4 output wires as B(i) for $0 \leqslant i \leqslant 3$. Then, description for the leftmost parallel iterative structure in the 8-input OR gate is as follows.

```
pfor i from 0 to 3
          { B(i) ← A(2i) OR A(2i + 1); }
```

- Three parallel iterative structures consisting of 4, 2, and 1 OR gates form a sequential iterative structure in the 8-input OR gate. To describe the sequential iterative structure, we use keyword **sfor**. Let us assume that wires are represented A(i)(j) for $0 \leqslant i \leqslant 3$ and $0 \leqslant j \leqslant 2^i - 1$, where A($i$)($j$) denotes the jth wire to the ith parallel iterative structure. Then, they can be described as follows.

```
sfor i from 2 downto 0 {
          pfor j from 0 to 2^i − 1 {
               A(i)(j) ← A(i + 1)(2j) OR A(i + 1)(2j + 1); } }
```

- Describing (and reading) an iterative structure is much easier if we replace it with recursion. Keywords **if** and **else** are used to describe a conditional structure dependent on indices of **pfor** or **sfor** loops. A recursive function ORF describing the b-bit 2^k-input OR gate, $k \geqslant 1$, is as follows.

```
ORF   generic (k, b)
in:   A(0..2^k − 1)(0..b − 1);
out:   B(0..b − 1);
signal:   T(0..1)(0..b − 1);
begin
if ( k = 1 ) {
  pfor i from 0 to b − 1
        { B(i) ← A(0)(i) OR A(1)(i); }
}
else {
  ORF generic (k − 1, b) (A(0..2^(k−1) − 1), T(0));
  ORF generic (k − 1, b) (A(2^(k−1)..2^k − 1), T(1));
  pfor i from 0 to b − 1
        { B(i) ← T(0)(i) OR T(1)(i); }
}
end
```

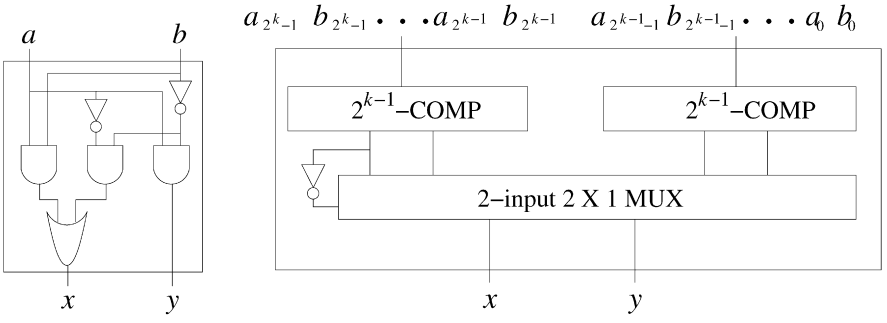
Now, `MUX generic (k, b)`, the full description of the b-bit $2^k \times 1$ MUX using recursion, can be understood.

Remark. A Boolean circuit can be described either recursively or iteratively, and either way the final logic circuits produced by a synthesis tool are the same. Hence, using recursion in circuit design is highly recommendable. In contrast, recursion in sequential computation suffers performance degradation due to the use of system stacks, when compared to iteration.

4. Comparison

We describe a 2^k-bit magnitude comparator (2^k-COMP), $k \geqslant 0$, that gets two 2^k-bit integers a and b as inputs and produces a pair of bits (x, y). The output (x, y) is $(1, 0)$ if $a = b$, $(0, 1)$ if $a > b$, and $(0, 0)$ if $a < b$. A 2^0-COMP (1-COMP) consists of three AND gates, two NOT gates, and an OR gate (Fig. 5(a)). A 2^k-COMP, $k \geqslant 1$, is constructed recursively from two 2^{k-1}-COMPs and a 2-input 2×1 MUX (Fig. 5(b)). The correctness of a 2^k-COMP is as follows. Let a_h and b_h denote the most significant 2^{k-1} bits of a and b and a_l and b_l the least significant 2^{k-1} bits of them, respectively. One 2^{k-1}-COMP compares a_h and b_h and outputs (x_h, y_h) and the other compares a_l and b_l and outputs (x_l, y_l). If $a_h \neq b_h$, the result of comparing a and b is the same as that of comparing a_h and b_h. Otherwise (if $a_h = b_h$), the result of a and b is the same as that of a_l and b_l. Hence, if $a_h \neq b_h$, i.e., $x_h = 0$, the 2-input 2×1 MUX outputs (x_h, y_h), and it outputs (x_l, y_l), otherwise. Since the depth and the size of both 1-COMP and 2-input 2×1 MUX are O(1), the depth and the size of 2^k-COMP are $O(k)$ and $O(2^k)$, respectively. The recursive structure of 2^k-COMP is well reflected in the following function `COMP`. Note that the wires inside 1-COMP (i.e., T(0..1)) and the wires inside 2^k-COMP for $k \geqslant 1$ (i.e., P(0..1)(0..1), Q(0..1), and R) are not the same, and their definitions are affected by parameter k.

```
COMP generic (k)
in:   A(0..2^k − 1), B(0..2^k − 1);
out:   X, Y;
if( k = 0 ) {          /* 1-COMP */
signal:   T(0..1);
begin
      T(0) ← A(0) AND B(0);
```



(a) 1-COMP ($k = 0$) (b) 2^k-COMP ($k > 0$)

Fig. 5. Comparator.

```
      T(1) ← (NOT A(0)) AND (NOT B(0));
      X ← T(0) OR T(1);
      Y ← A(0) AND (NOT B(0));

end
}
else {
signal: P(0..1)(0..1), Q(0..1), R;
begin
      COMP generic (k − 1) (A(0..2^(k−1) − 1),
                B(0..2^(k−1) − 1), P(0)(1), P(0)(0));
      COMP generic (k − 1) (A(2^(k−1)..2^k − 1),
                B(2^(k−1)..2^k − 1), P(1)(1), P(1)(0));
      R ← NOT P(1)(1);
      MUX generic (1, 2) (P(0..1)(0..1), R, Q(0..1));
      X ← Q(1);
      Y ← Q(0);
end
}
```

5. Bit counting

We describe a bit counter that gets n bits as inputs and produces the number of 1 bits among the n bits, i.e., the sum of the n bits. Thus, the output can be as large as $\lceil \log(n+1) \rceil$ bits. The bit counter consists of carry save adders and an adder. In fact, the bit counter is a variant of a Wallace-tree [9] that is designed to add n n-bit numbers with $O(\log n)$ depth and $O(n^2)$ size. However, since the bit counter adds 1-bit numbers, the size of the bit counter is reduced to $O(n)$. The internal structure of the bit counter can be divided into stages. Each stage i, $i \geqslant 1$, gets i-bit integers. Let x_i denote the number of i-bit integers. Stage i (except the last) outputs $\lceil 2x_i/3 \rceil (= x_i - \lfloor x_i/3 \rfloor)$ number of $(i+1)$-bit integers. The last stage takes two integers and it outputs the sum of the integers. The number of stages is $O(\log n)$ because the number of output integers of each stage (except the last) is 2/3 of the number of input integers of the stage. Hence, carry save adders in stage i are i-bit carry save adders and the adder in the last stage is an $O(\log n)$-bit adder.

For example, a bit counter taking 6 bits as inputs is shown in Fig. 6. In the first stage, the 6 input bits are reduced to four 2-bit integers. In the second and the third stage, the four 2-bit integers are reduced to three 3-bit integers and then to two 4-bit integers. Note that the integer that is an output of a 1-CSADD and directly connected to a 3-CSADD should be expanded by one bit. In the last stage, the two 4-bit integers are added by an adder. Although the adder outputs a 5-bit integer, we take only 3 least significant bits of them because the sum of 6 input bits is at most 6 and thus 3 bits are sufficient for the sum.

Consider the depth and the size of the bit counter. Since the depth of a carry save adder is $O(1)$ and the depth of the $O(\log n)$-bit adder is $O(\log n)$, the depth of the bit counter is $O(\log n)$. We compute the size of the bit counter by adding the sizes of all carry save adders and the $O(\log n)$-bit adder. Because in stage i (except the last), we have $O(n \cdot (2/3)^{i-1})$ number of i-bit carry save adders whose sizes are i, the size of all carry save adders in the bit counter is $O(n \cdot (1 + 2 \cdot 2/3 + \cdots + i \cdot (2/3)^{i-1} + \cdots)) = O(n)$. Since the size of the $O(\log n)$-bit adder is $O(n)$, the size of the bit counter is $O(n)$.

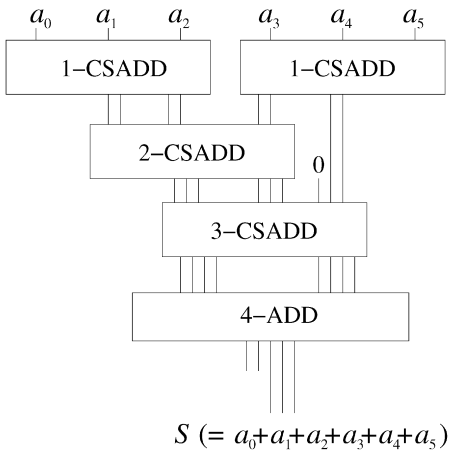


Fig. 6. A bit counter.

The bit counter is described as a function call to recursive function ITADD. The function ITADD, taking n b-bit numbers and producing the sum of the n numbers, is as follows. The bit counter for n inputs is described as a function call ITADD **generic** $(n, 1, \lceil \log(n+1) \rceil)$ $(X(0..n-1),\ Y(0..\lceil \log(n+1) \rceil - 1))$.

```
ITADD generic (n, b, f)
in:   X(0..n−1)(0..b−1);
out:   Y(0..f−1);
if (n=2) {
signal:  S(0..b);
begin
      ADD generic (b) (X(0), X(1), S);
      Y(0..f−1) ← S(0..f−1);
end
}
else {
signal:  T(0..⌈2n/3⌉−1)(0..b);
begin
      pfor j from 0 to ⌊n/3⌋−1
            {CSADD generic (b) (X(3j), X(3j+1), X(3j+2),
                  T(2j), T(2j+1));}
      if ((n mod 3)=1) {
            T(⌈2n/3⌉−1)(0..b−1) ← X(n−1)(0..b−1);
            T(⌈2n/3⌉−1)(b) ← 0; }
      else if ((n mod 3)=2) {
            T(⌈2n/3⌉−2)(0..b−1) ← X(n−2)(0..b−1);
            T(⌈2n/3⌉−2)(b) ← 0;
            T(⌈2n/3⌉−1)(0..b−1) ← X(n−1)(0..b−1);
            T(⌈2n/3⌉−1)(b) ← 0; }
      ITADD generic (⌈2n/3⌉, b+1, f)
            (T(0..⌈2n/3⌉−1)(0..b), Y(0..f−1));
end
}
```

6. Sorting

We describe a sorting circuit that sorts n integers (of b bits each) with $O(\log n + \log b)$ depth and $O(bn^2)$ size. We will represent the sequence of input integers as a and each input integer as a_i $(0 \leqslant i \leqslant n-1)$. We will represent the sequence of (sorted) output integers as a' and each output integer as a'_i $(0 \leqslant i \leqslant n-1)$. The sequence of output integers is sorted in nondecreasing order. When two integers a_i and a_j are equal, a_i appears after a_j in the output sequence if a_i is after a_j in the input sequence, i.e., $i > j$. Hence, a_i will be after a_j in the output sequence if and only if a Boolean expression $(a_i > a_j) \vee (a_i = a_j \wedge i > j)$ is true. We assume that n and b are powers of 2. (If not, we can take the smallest power of 2 that is greater than n or b.)

This sorting circuit consists of three top-level components, which are PAIRWISE_COMP-ARISON, COMPUTE_RANK, and PERMUTATION. This is a naive sorting algorithm that is also described in [29], but it shows many interesting aspects of Boolean circuit design. In addition, it produces a sorting circuit of optimal depth $O(\log n + \log b)$. We first overview these top-level components, then describe the internal structures of these components, and finally consider their depths and sizes. The code for the top-level interconnection (Fig. 7) is as follows.

```
SORT generic (n, b);
in:  A(0..n−1)(0..b−1);
out:   O(0..n−1)(0..b−1);
signal:   C(0..n−1)(0..n−1),
          R(0..n−1)(0..⌊log⌈(n+1)⌉⌋−1);
begin
PAIRWISE_COMPARISON generic(n, b) (A, C);
COMPUTE_RANK generic(n) (C, R);
PERMUTATION generic(n, b) (A, R, O);
end
```

We now describe the internal structures of these three components. Component PAIRWISE_COMPARISON (Fig. 8) compares each pair a_i and a_j $(0 \leqslant i, j \leqslant n-1)$ and outputs c_{ij} as 1 if $(a_i > a_j) \vee (a_i = a_j \wedge i > j)$ and 0 otherwise. The component

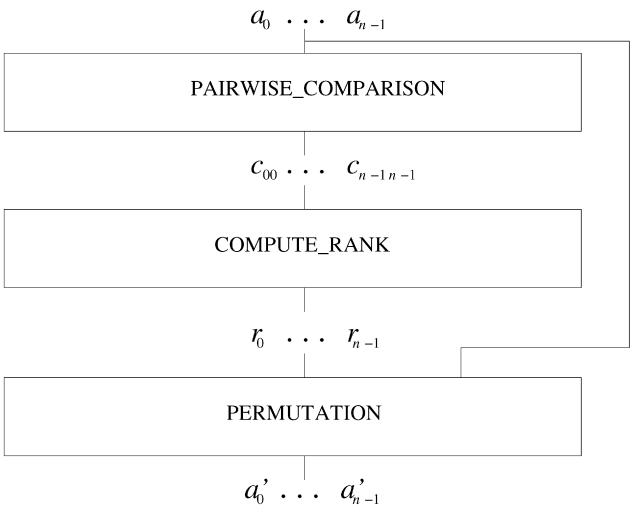


Fig. 7. Three top-level components and their input and output.

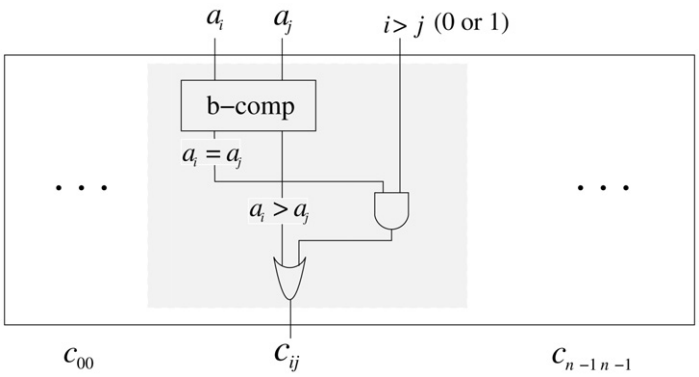


Fig. 8. PAIRWISE_COMPARISON. A shaded region is a block for computing c_{ij}.

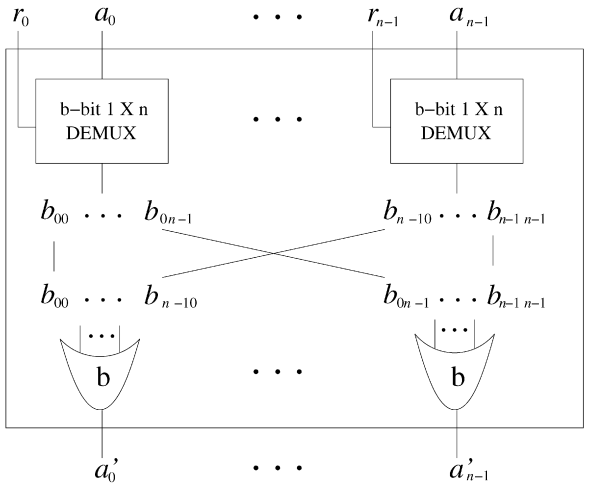


Fig. 9. PERMUTATION.

consists of n^2 blocks such that each block computes c_{ij} in parallel. Each block has a b-bit comparator to compare a_i and a_j and an AND gate and an OR gate additionally. Note that the Boolean expression $i > j$ can be computed at compile time of a synthesis tool because values i and j are fixed regardless of values a_i and a_j and thus $(i > j)$ is either 0 or 1. The depth

of PAIRWISE_COMPARISON is $O(\log b)$ due to the b-bit comparator, and its size is n^2 times the size of the b-bit comparator, which is $O(bn^2)$. The code for PAIRWISE_COMPARISON is given as follows.

```
PAIRWISE_COMPARISON generic (n, b)
in:   A(0..n−1)(0..b−1);
out:   C(0..n−1)(0..n−1);
signal:   D(0..n−1)(0..n−1), E(0..n−1)(0..n−1);
begin
  pfor i from 0 to n−1 {
       pfor j from 0 to n−1 {
             COMP generic (⌈log(b+1)⌉)
             (A(i), A(j), D(i)(j), E(i)(j));
             C(i)(j) ← E(i)(j) OR (D(i)(j) AND (i > j)); } }
end
```

Component COMPUTE_RANK computes the rank r_i for each a_i in the output sequence, i.e., the number of input integers preceding a_i in the output sequence. The rank r_i is obtained by computing $c_{i0} + \cdots + c_{i,n-1}$. This component consists of n bit counters such that each bit counter computes r_i in parallel. Its depth is the same as the depth of the bit counter, which is $O(\log n)$. Its size is n times the size of the bit counter and thus it is $O(n^2)$. The code for COMPUTE_RANK is as follows.

```
COMPUTE_RANK generic (n)
in:   C(0..n−1)(0..n−1);
out:   R(0..n−1)(0..⌈log(n+1)⌉−1);
begin
pfor i from 0 to n−1
       { ITADD generic (n, 1, ⌈log(n+1)⌉)
             (C(i)(0..n−1), R(i)(0..⌈log(n+1)⌉−1)); }
end
```

Component PERMUTATION generates a sorted sequence a' by outputting a_i to a'_{r_i}. It consists of n b-bit $1 \times n$ DEMUXs and n b-bit n-input OR gates (Fig. 9). The ith DEMUX, $0 \leqslant i \leqslant n-1$, from the left gets a_i and r_i and sends a_i to its r_ith output wire which is connected to the r_ith OR gate from the left. The depth of this component is the sum of the depth of the DEMUX and the depth of the OR gate, which is $O(\log n)$ because the depths of the DEMUX and the OR gate are all $O(\log n)$. Its size is n times the sum of the sizes of the DEMUX and the OR gate, which is $O(bn^2)$, because the sizes of the DEMUX and the OR gate are all $O(bn)$. The code for PERMUTATION is as follows.

```
PERMUTATION generic (n, b)
in:   A(0..n−1)(0..b−1), R(0..n−1)(0..⌈log(n+1)⌉−1);
out:   O(0..n−1)(0..b−1);
signal:   B(0..n−1)(0..n−1)(0..b−1), C(0..n−1)(0..n−1)(0..b−1);
begin
pfor i from 0 to n−1
      { DEMUX generic (⌈log(n+1)⌉, b)
            (A(i), R(i)(0..⌈log(n+1)⌉−1), B(i)(0..n−1)); }
pfor i from 0 to n−1
     pfor j from 0 to n−1
           C(i)(j) ← B(j)(i);
pfor i from 0 to n−1
      { ORF generic (⌈log(n+1)⌉, b) (C(i)(0..n−1), O(i)); }

end
```

Theorem 1. *The depth and the size of the sorting circuit are $O(\log n + \log b)$ and $O(bn^2)$, respectively.*

7. Concluding remarks

We have introduced the language BC for Boolean circuit programming, and explained its features by example codes for counting and sorting. As can be seen in the example codes, the language BC is a convenient tool to design parallel algorithms due to its general iterative and recursive structures and the ease of modular design.

In developing parallel algorithms on the Boolean circuit model, one has to deal with input data in terms of *bits* rather than *words*. But it increases the accuracy of the model rather than imposing unnecessary details, because without specifying the number of bits in a word the complexity of a problem in the model might change. For example, Paul and Simon showed that the $\Omega(n \log n)$ bound for sorting in the RAM model does not hold if one can pack many bits into one word [22]. Moreover, handling bits on the Boolean circuit model incurs only one more level of modular design when compared to handling words.

Many parallel algorithms can be developed in the Boolean circuit model by using the BC language. Recently, E. Park and K. Park gave a Boolean circuit of $O(\log^2 n(\log b + \log\log n))$ depth and $O(bn^2 \log n)$ size that finds maximum matching in a convex bipartite graph [20]. The AKS and Paterson's sorting networks [2,21] are translated into Boolean circuits of depth $O(\log n \log b)$ and size $O(bn \log n)$ using the optimal comparator in Section 4, while the sorting circuit in Section 6 has depth $O(\log n + \log b)$. An open problem is whether there is a sorting circuit of depth $O(\log n + \log b)$ and size $o(bn^2)$.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
[2] M. Ajtai, J. Komlós, E. Szemerédi, An $O(n \log n)$ sorting network, in: Proc. 15th Ann. ACM Symp. on Theory of Computing 1983, pp. 1–9.
[3] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.
[4] P.J. Ashenden, The Designer's Guide to VHDL, 2nd ed., Morgan Kaufmann, 2001.
[5] G.D. Bergland, Fast Fourier Transform hardware implementation – A survey, IEEE Transactions on Audio and Electroacoustics 17 (2) (1969) 104–108.
[6] M.D. Ciletti, Advanced Digital Design with the Verilog HDL, Prentice Hall, 2003.
[7] K.M. Chandy, J. Misra, Parallel Program Design, Addison-Wesley, 1988.
[8] M. Crochemore, Z. Galil, L. Gasieniec, K. Park, W. Rytter, Constant-time randomized parallel string matching, SIAM Journal on Computing 26 (4) (1997) 950–960.
[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, 1990.
[10] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: towards a realistic model of parallel computation, in: Proc. 4th ACM S1GPLAN Symp. On Principles and Practice of Parallel Programming (PPoPP '93), San Diego, CA, May 1993.
[11] D.E. Culler, J.P. Singh, Parallel Computer Architecture – A Hardware/Software Approach, Morgan Kaufmann, 1999.
[12] A. Gibbons, W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, 1988.
[13] R. Greenlaw, H.J. Hoover, W.L. Ruzzo, Limits to Parallel computation: P-completeness Theory, Oxford University Press, 1995.
[14] K. Hwang, Z. Xu, Scalable Parallel Computing, McGraw-Hill, 1998.
[15] J. JaJa, An Introduction to Parallel Algorithms, Addison-Wesley, 1992.
[16] I. Koren, Computer Arithmetic Algorithms, Prentice Hall, 1993.
[17] F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann, 1992.
[18] Z. Liu, Y. Song, T. Ikenaga, S. Goto, A VLSI array processing oriented Fast Fourier Transform algorithm and hardware implementation, IEICE Transactions on Fundamentals E88-A (12) (2005) 3523–3530.
[19] MentorGraphics Inc., ModelSim, vcom compiler, http://www.model.com.
[20] E. Park, K. Park, A new Boolean circuit for maximum matching in a convex bipartite graph, in: 17th Australasian Workshop on Combinatorial Algorithms, July 2006.
[21] M.S. Paterson, Improved sorting networks with $O(\log N)$ depth, Algorithmica 5 (1990) 75–92.
[22] W. Paul, J. Simon, Decision trees and random access machines, Monographie de L'Enseigment Mathematique 30 (1982) 331–340.
[23] J.E. Savage, Models of Computation: Exploring the Power of Computing, Addison-Wesley, 1998.
[24] M. Sipser, Introduction to the Theory of Computation, PWS 1997.
[25] L. Snyder, Type architectures, shared memory, and the corollary of modest potential, Annual Review of Computer Science 1 (1986) 289–317.
[26] Synopsis Inc., Design Analyzer, http://www.synopsis.com.
[27] L.G. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (8) (1990) 103–111.
[28] Verilog Formal Syntax Specification, http://www.verilog.com/VerilogBNF.html.
[29] H. Vollmer, Introduction to Circuit Complexity, Springer, 1999.
[30] I. Wegener, The Complexity of Boolean Functions, Wiley-Teubner, 1987.