A HIGH-LEVEL INTERACTIVE SYSTEM FOR DESIGNING VLSI SIGNAL PROCESSORS

by

Robert Michael Owens
and
Joseph Ja' Ja'

Dept. of Electrical Engineering
University of Maryland
College Park, MD 20742
Abstract

This paper describes a high-level interactive system that can be used to generate VLSI designs for various operations in signal processing such as filtering, convolution and computing the discrete Fourier transform. The overall process is fully automated and requires that the user specifies only few parameters such as operation, precision, size and architecture type. The built-in architectures are new digit-on-line bit-serial architectures that are based on recently derived fast algorithms for the above operations. The basic elements are compact and have a very small gate delay. We feel that our system will offer a flexible and easy to use tool that can produce practical designs which are easy to test, efficient and fast.
1. Introduction

This paper introduces a preliminary version of a high-level interactive system that allows a user, with only a rudimentary knowledge of VLSI, to generate a design at the layout level for performing various operations required in signal processing. The overall process is completely automated and does not require that the user supplies his own algorithm or architecture. At the same time, the system is flexible enough to allow the user to compare between different possible design alternatives and to tailor his design according to his needs. Our system is different than previous systems for signal processing applications ([B], [PB]) in several significant ways. The fundamental differences will be outlined below.

The F.I.R.S.T. Silicon Compiler ([B]) and the Macrocell Design ([PB]) both allow a user to implement the standard multiply/add algorithms by using built-in cells. The F.I.R.S.T. compiler has a relatively fixed floor plan and requires that the user specifies the various cells (from a set of operators) by using the F.I.R.S.T. programming language. A layout program is then used to arrange the corresponding blocks in a simple way and a two-layer router is used to make the connections between the cells. The Macrocell Design is based on a library of macrocells which are especially designed for signal processing applications. The overall design is mostly left to the user.

While both of the above systems revolve around the standard algorithms/architectures ([A], [D]) for various operations such as filtering, convolution and computing the discrete Fourier transform, our scheme is based on a new set of algorithms ([ER], [KP], [W]) that in most cases use substantially fewer arithmetic operations and that have a general structure that lends itself well to highly parallel, pipelined designs. The speed-up obtained can be mathematically shown to be optimal ([JO]). As
a matter of fact, preliminary simulation results lead us to believe that a discrete Fourier transform (DFT) processor can be constructed which is capable of performing about 64,000 pipelined 1024 point DFT's per second. Some of the components of such a DFT, which were produced by our system, are actually being fabricated by MOSIS. An example of such a design is given in section 3 of this paper.

The basic cells supplied by other systems could be relatively large. This can severely limit the freedom of the designer. Our basic cells are digit-on-line bit serial adders, subtractors, multipliers, delay elements, and dynamic shift-registers that have been specifically designed for our system. These cells are quite compact with a very small gate delay (typically two to four gate delays). Moreover, the process of putting the pieces together is completely automated and can be done by specifying the architecture type and the size as will be explained later. The resulting designs can be implemented on a single or few VLSI chips.

Other advantages of our system include the properties that the designs produced require no significant routing or placement and almost no control. Compare this with the fact that routing is usually time consuming, wasteful, and introduces long wires. Moreover, designing the control is usually a complex task that can substantially slow down the design process.

On the other hand, our system is very easy to use and allows the user to compare between three different types of architectures (already in the system) and to tailor the overall design according to his needs. It will also provide the user with estimates of the area and the delay for each constructed component. This flexibility is largely due to the new structures of the basic algorithms and to the new basic digit-on-line bit-serial architectures. We feel that overall our system will offer a flexible and easy-to-use tool that can produce practical designs that are efficient, easy to test, and fast.
The rest of the paper is organized as follows. The basic algorithms/architectures are presented in section 2 together with the hardware description of one basic component. Section 3 introduces the software components of the system and several extensions will be outlined in section 4.

2. Basic Strategy

In this section, we will briefly outline the classes of architectures that our system will provide and give an outline of the hardware description of one of the basic components. We will restrict our discussions to the problem of computing the discrete Fourier transform. Precisely the same strategy will work for designing filters and convolvers. Our approach will be radically different than previous approaches and will result in fully pipelined bit serial architectures that can be implemented on a single or few chips and that can achieve a provably optimal area-time tradeoff ([JO], [OJ]). These architectures are based on different iterative schemes that use the so called small $n$ algorithms. A small $n$ algorithm for computing the discrete Fourier transform on $n$ points $DFT(n)$ can be expressed as follows:

$$y = S C T x,$$

where $T$ is a $\delta \times n$ matrix whose elements are either $-1$, $0$, or $1$, representing the input additions and subtractions, $C$ is a diagonal $\delta \times \delta$ matrix, representing the multiplications, $S$ is an $n \times \delta$ matrix whose elements are either $-1$, $0$, or $1$, representing the output additions/subtractions, and $\delta \approx n$ is the number of multiplications. Optimal primitive small $n$ algorithms have been derived for $n = 2, 3, 4, 5, 7, 8, 9, 16$. A composite small $n$ algorithm for $DFT(n)$ can be obtained by combining the primitive small $n$ algorithms for $DFT(n_1), DFT(n_2), \ldots, DFT(n_l)$, where $n = n_1 n_2 \ldots n_l$ and the $n_i$'s are relatively prime.
Let \( n = n_1 n_2 \), where \( n_1 \) and \( n_2 \) are relatively prime, and let \((S^1, C^1, T^1)\) and \((S^2, C^2, T^2)\) be the primitive or composite small \( n \) algorithms for \( DFT(n_1) \) and \( DFT(n_2) \) respectively. The pair \((n_1, n_2)\) is called a valid decomposition for \( n \). We consider three types of iterations:

A) **The Good (prime factor) iteration:** \( DFT(n) \) is computed by the following equation:

\[
Y = (S^1 C^1 T^1(S^2 C^2 T^2 X)^T)^T,
\]

where \( X \) and \( Y \) are \( n_1 \times n_2 \) arrays holding the input and the output numbers respectively in row major order form, and \( T \) is the transpose operation.

B) **The Winograd iteration:** \( DFT(n) \) is given by the following equation:

\[
Y = (S^2(S^1 C \mathcal{O} T^1(T^2 X)^T))^T,
\]

where \( \mathcal{O} \) is element-wise multiplication is \( C \) is a constant \( n_1 \times n_2 \) array given by:

\[
C_{i,j} = C_{i,1}^1 C_{j,2}^2, \quad 0 \leq i < n_1, \quad 0 \leq j < n_2.
\]

C) **The Mixed (FFT-like) iteration:** In this case \( n_1 \) and \( n_2 \) need not be relatively prime. The output \( Y \) is given by:

\[
Y = S^2 C^2 T^2(W \mathcal{O} S^1 C^1 T^1 X)^T,
\]

where \( W \) is a constant \( n_1 \times n_2 \) array given by:

\[
W_{ij} = \omega^{ij}, \quad 0 \leq i < n_1, \quad 0 \leq j < n_2.
\]

Notice that one valid decomposition may be better than another decomposition.
For example, $30 = 3 \times 10 = 5 \times 6$, both of which are valid decompositions. Note that 
$DFT(10) = DFT(5) \otimes DFT(2)$ and $DFT(6) = DFT(3) \otimes DFT(2)$ are both composite small $n$ algorithms. However, the corresponding algorithm for the decomposition $(3,10)$ uses 48 multiplications while the decomposition $(5,6)$ uses 36 multiplications (and hence results in a smaller scaling components, see below). As we will see in the next section, the system will inform the user about the best decomposition but will allow the user to choose a different decomposition for his design.

A close look at the above schemes reveals that three types of hardware components are required:

1) A *summation component* is needed to compute expressions of the form $AX$, where the elements of $A$ are either $-1$, $0$, or $1$. This component will therefore be used to compute the input/output additions/subtractions. It will be constructed from an array of serial adders.

2) A *scaling component* is needed to compute expressions of the form $X \oplus Y$ or $DX$ for a diagonal matrix $D$. This component will be constructed from a number of linear array of digit online multipliers which will perform all the corresponding multiplications.

3) A *transpose component* is needed to compute expressions of the form $X^T$. A two dimensional array of dynamic shift registers will be used to implement this component.

Using the above components and the iteration methods outlined above, we obtain three different types of architectures:

A) *Good's type architecture* whose interconnection is given in figure 2.1.
X → \[ T^2 \times \]
\[ C^2 \times \]
\[ S^2 \times \rightarrow X^T \rightarrow T^1 \times \]

S^1 \times → X^T → Y
\[ C^1 \times \]

Figure 2.1 Structure of Good's Type Architecture.

B) Winograd's type architecture whose interconnection is given in figure 2.2.

X → \[ T^2 \times \]
\[ X^T \]
\[ T^1 \times \rightarrow X \odot Y \rightarrow S^1 \times \]

S^2 \times → Y
\[ X^T \]
\[ C \]

Figure 2.2 Structure of Winograd's Type Architecture.

C) Mixed type architecture whose interconnection is given in figure 2.3.

X → \[ T^1 \times \]
\[ C^1 \times \]
\[ S^1 \times \rightarrow X \odot Y \rightarrow X^T \rightarrow T^2 \times \]

S^2 \times → X^T → Y
\[ C^2 \times \]
\[ X^T \]
\[ W \]

Figure 2.3 Structure of Mixed Type Architecture.

The system will query the user as to which of the above architectures he wants to use and once this decision is made and the size specified the system will display the logical interconnection layout. On the other hand, a new architecture type could be easily introduced as long as it is based on using summation, scaling, and transpose components.
As an example of the actual hardware implementation, we give a brief description of the summation component taken from ([OJ]). Recall that the mathematical operation performed by the summation component is given by $Z = S \cdot X$, where

$$S = \begin{bmatrix} S_{i,j}, & 0 \leq i < n_0, 0 \leq j < n_1 \end{bmatrix},$$

$$X = \begin{bmatrix} X_{i,j}, & 0 \leq i < n_1, 0 \leq j < n_2 \end{bmatrix},$$

and

$$Z = \begin{bmatrix} Z_{i,j}, & 0 \leq i < n_0, 0 \leq j < n_2 \end{bmatrix}.$$

The elements of $S$ are predefined and are either $-1$, $0$, or $1$. We will make use of this fact by building them into the summation component itself rather than supplying them to the component as the operation is performed. Being able to tailor the component in this manner will decrease its area and the number of inputs. To reduce the area and number of inputs and outputs further, bit serial arithmetic is utilized in the design of the component. By using bit serial arithmetic, we will be able to reduce the perimeter required from $O(p \cdot (n_1 + n_0))$ to $O(n_1 + n_0)$ and the area required from $O(p \cdot n_1 \cdot n_0)$ to $O(n_1 \cdot n_0)$. We feel that serial arithmetic is necessary given the size of the problems we would like to solve (as least 1024 point DFT) and the maximum complexity of VLSI chips in the foreseeable future.

For reasons given in [Ow], we propose left directed (least to most significant digit) bit serial arithmetic if the input is integer data and right directed (most to least significant bit) if the input is fractional data. The following is the left directed design since it is the most complex of the two. The summation component can be constructed by interconnecting smaller subcomponents of only three different types (addition, subtraction, and delay), which share the same rectangular shape. Each of these subcomponents can be represented as shown in figure 2.4.
The input $r_{in}$ is used to indicate that the first digit of an element is being supplied to the subcomponent. The functional description of the digit online addition subcomponent is given by figure 2.5.

**Digit Online Addition Subcomponent**

\[
x_{out} = x_{in}
\]

\[
r_{out} = r_{in}
\]

If $r_{in} \equiv 0$ then

\[
s = x_{in} + z_{in} - b \cdot c \quad \text{where } c \text{ is chosen such that}
\]

\[-b+2 \leq s \leq b-2
\]

\[
z_{out} = t + c
\]

else

\[
s = x_{in} + z_{in}
\]

\[
z_{out} = t
\]

endif

\[
t = s
\]

Figure 2.5. Subcomponent Functional Definition.

where $x_{in}$, $r_{in}$, and $z_{in}$ are the values supplied to inputs $x_{in}$, $r_{in}$, and $z_{in}$ respectively in
the time interval prior to some clocking, $x_{out}$, $r_{out}$, and $z_{out}$ are the values generated at outputs $x_{out}$, $r_{out}$, and $z_{out}$ respectively in the interval after that clocking, and $b$ is the base of the number system being used. The functional description of the subtraction and delay subcomponents follows from the description of the addition subcomponent.

Digit online subcomponents of the three different types are interconnected as illustrated by figure 2.6 to form a digit online summation component.

![Diagram of digit online summation component](image)

Figure 2.6. Summation Component.

Preliminary implementations of our design show that each of the subcomponents can be laid out on an area of 170 microns by 170 microns and that the subcomponents have a delay of 40 nanoseconds. Hence, given the current restrictions imposed by MOSIS, we will be able to fabricate an array of $24 \times 24$ subcomponents on a single chip. This subcomponent has been submitted to MOSIS for fabrication.
3. **Overview of the Software System**

In this section we describe the software system that will automatically generate the VLSI chips for a given design from its formal description. A description of this software is not only interesting in itself but it also illustrates some of major features of our overall approach. Our system presently consists of three main programs which are illustrated in figure 3.1.

![Diagram of software environment]

The *logical* descriptions for each component are created using an interactive program called *Generate*. *Generate* not only allows the user to create and change the logical description of VLSI chips, but it also assists him by indicating those choices which the program believes lead to an optimal design and by listing the logical parameters of the design after the logical description has been fully specified. The compiler (actually more of a translator) *Compose* converts the logical description of each component into a *physical* description. At present a cif file [HS] is generated to physically describe each component. The program *Logic* is used in a semiautomatic fashion to create the physical description of an element from its logical description (essentially a truth table). This
will help the user to introduce new elements and, hence, extend the capabilities of *Generate/Compose*. For example the current implementation uses base four for value representation. A user may wish to use a different base which can be done by creating a new set of elements using *Logic*.

To create the logical description, *Generate* needs to know 1) the small $n$ algorithms, 2) the user's name for the design, 3) the overall architecture type to be used, 4) the data type desired, 5) the precision (in digits), and 6) the size (number of points) of the input. *Generate* uses an interactive session to determine all of these items except the first. The first, a description of each of the primitive small $n$ algorithms (in terms of the matrices $S$, $C$, and $T$), is compiled into the program. *Generate* at present has the description for the small $n$ algorithms where $n = 2, 3, 4, 5, 7, 8$ compiled into it ([ER]). Furthermore, *Generate* can compute the description for the composite small $n$ algorithms whenever $n$ has a valid decomposition by using the Kronecker product operation ([JO]). Hence, *Generate* can compute the descriptions for the composite small $n$ algorithms, where $n = 6, 10, 12, 14, 15, \ldots$.

We will illustrate how *Generate* works by giving the description of a typical interactive session (figure 3.2). Note the that user input is indicated by bold face type.
% generate

P1 - Design name? [ ] - example
P2 - Architecture type.
    Good's? [n] -
    Winograd's? [n] - y
P3 - Data type.
    Real to real? [n] - y
P4 - Precision? [8] -
P5 - Transform size.
    Range search? [y] -
    Upper limit? [512] - 36
P6 - Num -> 24 (3,8) (8,3)
    Num -> 28 (4,7) (7,4)
    Num -> 30 (2,15) (3,10) (6,5) (5,6) (10,3) (15,2)
    Num -> 35 (5,7) (7,5)
P7 - More than one size with a valid decomposition found. Select one? [y] -
    Size? [35] - 30
    Num -> 30 (2,15) (3,10) (6,5) (5,6) (10,3) (15,2)
P8 - Best computed decomposition for 30 is (5,6)
P9 - Do you want to try to do better? [n] - y
    Use (2,15) as the decomposition for 30? [n] -
    Use (3,10) as the decomposition for 30? [n] - y
P10 - Design name is example
    Architecture type is Winograd's
    Data type is real to real
    Precision is 8 digits
    Size is 30 (3 x 10)
    Summation Component S1 is 510 lambda by 510 lambda
    Summation Component T1 is 1700 lambda by 2040 lambda
    Summation Component S2 is 510 lambda by 510 lambda
    Summation Component T2 is 2040 lambda by 1700 lambda
P11 - Change entry? [n] -
P12 - Layout chip? [y] -
    Generating layout for S1
    Generating layout for T1
    Generating layout for S2
    Generating layout for T2
    Quit? [n] - y
    Normally exiting generate
%

Figure 3.2 User example dialogue.

We now briefly describe the purpose of each step. At points P1, P2, P3, and P4 in figure 3.2, the user specifies, respectively, the name to be used for the design, selects the architecture type (Good's, Winograd's, and Mixed are available), selects the data type (real to real, real to complex, complex to complex, and complex to real are available),
and specifies the precision. If all the data to be entered were of this nature, a stand-
alone compiler approach would have been more appropriate. However, the next item,
the size of the input, illustrates the power of a more interactive approach.

Recall that the size of the input for a given architecture type must have at least
one valid decomposition before the VLSI chips with that architecture can be designed.
At point P5, the user has the option of either specifying the size directly (in which case,
only a check is made to see if the size has at least one valid decomposition) or of speci-
fying a range of sizes to determine which sizes have valid decompositions. In the case of
the example given in figure 3.2, a range search from 24 to 36 inclusive was chosen. The
program then determines and lists (starting at point P6) the sizes with valid decomposi-
tions along with those decompositions. For example, 30 has six valid decompositions
(2,15), (3,10), (6,5), (5,6), (10,3), (15,2) for the specified architecture type (Wnograd's).
Note that decomposition (6,5) does not lead to the same design as the decomposition
(5,6).

At point P7, the user has the option of specifying one of the listed sizes (if at
least one size with a valid decomposition was located in the range) or trying a different
range. At point P8, the program lists the decomposition which it believes will be the
optimal one with respect to area and time considerations and at point P9 the system
gives the user the option of overriding this choice. The user specified data and the lo-
gical size of each component is listed starting at point P10 and the user is given a chance
to reread the overall description at point P11. At point P12, the user indicates that
the logical description for each component should be created.

Since each component is constructed from compatible (both physically and logi-
cally) elements, the logical structure of the components which make up a design can be
described in a straightforward manner. A component can be logically visualized as a
two dimensional grid as illustrated in figure 3.3 which gives the logical structure for the component associated with operation $T^1 X$.

\[
\begin{array}{cccccc}
\text{left} & \text{positive} & \text{delay} & \text{delay} & \text{right} \\
| & | & | & | & | \\
\text{left} & \text{positive} & \text{positive} & \text{negative} & \text{right} \\
| & | & | & | & | \\
\text{left} & \text{positive} & \text{positive} & \text{positive} & \text{right} \\
\end{array}
\]

Figure 3.3 Logical Structure for Component T1.

The grid can be thought to have equally spaced adjacent grid lines, since no physical separation is implied by the distance between any two grid points. Two grid points connected by a line imply that the elements at those two points should be physically connected together by placing the one beside the other (with possibly some overlap) in the orientation indicated by the grid. The file produced by Generate has a structure as illustrated in figure 3.4 to actually describe logically the structure of a component.

```
T1 /enee/owens/vls/l1lib/cfl/lib
1 delay
2 positive
3 negative
4 left
5 right

4 2 1 1 5
4 2 2 3 5
4 2 2 2 5
```

Figure 3.4 Logical Description File for Component T1.

The first line specifies the name of the component and the library to be used to physically construct the component. The following five lines specify integer aliases for the elements. The blank line separates the header from the logical description. The last three lines specify element to be placed at each grid point.
Recall that component T1 was included in the design to compute $T^1 X$, where $T^1$ is given by:

$$T^1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & -1 & 1 \end{bmatrix}$$

as taken from [ER]. The difference between the expression $T^1 X$ and the logical structure of component $T1$ can be explained by as follows. Recall that a 1, 0, or $-1$ in $T^1$ imply that the expression $x + y$, $x$, or $x - y$, respectively should be performed by corresponding VLSI element. However, in designing the VLSI elements, we found that elements of almost equal size could be laid out for the operations $(x + y)$ (positive), $x$ (delay), and $-(x + y)$ (negative). A simple recoding of the matrix $T^1$ (which is taken care of by Generate) permits us to use these VLSI elements instead.

Once the logical structure for each component has been created by Generate, each can be compiled into a clf file by Compose. To create the physical description of a component, Compose reads the logical description along with the library specified in the logical description. The library contains information about the physical properties of each element. The library used to create the component T1 is given in figure 3.5.

```
delay 1600 -800 68000 68000 /enee/owens/vlsi/llb/delay.clf
positive -3600 -3600 68000 68000 /enee/owens/vlsi/llb/positive.clf
negative -800 -800 68000 68000 /enee/owens/vlsi/llb/negative.clf
left -11200 0 11200 68000 /enee/owens/vlsi/llb/left.clf
right 67200 -800 11200 68000 /enee/owens/vlsi/llb/right.clf
```

Figure 3.5 Physical library.

Each line of the library describes one of the elements to be used to construct a given component. The first entry gives the name of the element. The next two entries give the clf offset of the origin of the element. The next two entries give the clf size of the element. The last entry gives the name of the clf file which physically describes the element.
Upon reading the logical description of the component and the library, Compose 1) insures that each of the elements along a given row has the same height (the height of a row is then determined by the height of any element along that row), 2) insures that each of the elements along a given column has the same width (the width of a column is then determined by the width of any element along the column), 4) reads the clf file for each of the elements actually used in the component, and 5) produces the physical description in the form of a composite clf file. Note that the clf file for each of the elements is the unmodified clf file produced by the graphics station which was used to layout the elements. Figure 3.6 illustrates VLSI layout for component T1 as produced by clfplot.
Figure 3.6 Clfplot for Component T1.
Note that routing is not needed to create the physical description for a component. Furthermore, components themselves can be interconnected with little or no routing. We feel that this is one of the main advantages which result from using our architecture. It reduces the complexity of the compiler and it reduces the chip area and the delay since long interconnection wires are not needed. Moreover, our approach of using an Interactive editor and compiler combination has the advantage of making the system technology independent. The present physical descriptions for the elements are based on an NMOS technology. The ongoing switch to CMOS requires modifying the CIF library but no changes to the present software are needed.

Since the performance (in terms of both area and time) of any design created by Generate/Compose depends directly on the performance of any one element, the optimization of the layouts for each element is of utmost importance to us. Moreover, the user should be allowed to easily create new elements that have optimal or near optimal layouts. To attack this problem in a rigorous manner, we developed an expression factoring program called Logic. Given a formal description of a set of functions, Logic generates a set of factored expressions. These expressions can be used to obtain a physical layout in a straightforward way. The task of writing a program like Logic is not trivial, since generation of an "optimal" factored expression for any but the simplest of cases leads to hard problems (well known NP-complete problems). Hence, solutions to problems of even modest size may be impossible to find in a reasonable amount of running time.

To produce a set of optimal factored expressions, Logic uses essentially a depth first search of the exponentially large state space of possible factoring of the expressions. Since most of the time, the enumeration of expressions, which satisfy a given set of restrictions (particularly depth and fan in), is spent in the lower levels of the search tree,
Logic uses a hash table to quickly search the bottom levels for previously computed solutions. Logic also uses known facts (such that the depth of any factored expression for a certain subfunction must be larger than some easy to compute bound) and certain symmetric reductions to reduce the number of paths which must be searched. Given these techniques, we were able to use Logic to find near optimal factored expressions for the type of functions we encountered in creating the elements we are presently using.

Another interesting feature of Logic is that the standard NAND or AND/OR basis is not used. Instead Logic uses a basis whose realization as a gate is illustrated in figure 3.6 (cf. Weinberger array).

![Figure 3.7 Gate Cell.](image)

Each of the $x_{i,j}$'s represents a possible input and each of the $l_{i,j}$'s a link which may be either closed (conducting) or open (nonconducting). The advantage of this cell is the density of logic which can be packed into a given chip area. The goal of Logic is to minimize the area occupied by the rectangular cells and their pullups (in the case of NMOS) needed to realize a set of factored expression. To illustrate the use of Logic, we present the following example. Figure 3.8 gives the truth tables (input values for which the output is equal to 1) for least and most significant bits of the carry output, $c_0$ and
$c_1$ used by both the positive (addition) and negative (subtraction) elements.

<table>
<thead>
<tr>
<th>Truth Table for $c_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100</td>
</tr>
<tr>
<td>001100</td>
</tr>
<tr>
<td>001000</td>
</tr>
<tr>
<td>010000</td>
</tr>
<tr>
<td>110000</td>
</tr>
<tr>
<td>110100</td>
</tr>
<tr>
<td>111100</td>
</tr>
<tr>
<td>111000</td>
</tr>
<tr>
<td>100000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Truth Table for $c_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
</tr>
<tr>
<td>110000</td>
</tr>
<tr>
<td>110100</td>
</tr>
<tr>
<td>111100</td>
</tr>
<tr>
<td>111000</td>
</tr>
<tr>
<td>100000</td>
</tr>
</tbody>
</table>

Figure 3.8 Logic Input

Given these truth tables as input, Logic produced the factored decomposition as given by figure 3.9 as its output. Note that N and O represent the NAND and OR operations, respectively, and that the numbers at the leaves represent the input variables. In the case of figure 3.9, a NAND operation and its children OR operations can be combined quite efficiently with our gate cell. Hence, only five gates are needed to compute $c_0$ and $c_1$, which are two reasonably complex functions.
Figure 3.9 Expression Factorization.

Note the presence of two common subexpressions in the factored form. This leads to the sharing of gates for two functions and the reduction of the overall VLSI area needed. Output such as that given in figure 3.9 was used to create the elements used in the present system.

4. Future Directions

The current version of our system should be considered as a preliminary stage of a more comprehensive system that we are planning to develop. Our main goal is to provide a user with a flexible and easy-to-use tool that will allow him to generate a set of good designs, for any problem from a large class, without having to know the details of the best possible algorithms/architectures for this problem.

Currently Generate and Compose assume that each of the components of a design (summation, scaling, and transpose) is to be laid out on a separate chip. A relatively easy refinement would be to allow Generate and Compose to try to pack as many of the components as possible on a single chip. The corresponding placement and rout-
ing problems are not difficult because of the structure of the components.

Another possible extension is to enlarge the architecture types and the classes of transforms and composite transforms that the system can handle. It is rather straightforward to include convolution and filtering in our system because similar schemes have been developed for these transforms. These schemes will require the same types of components (summation, scaling, and transpose) as well as similar interconnection patterns. Moreover, we feel that our designs will offer several advantages over exiting designs. For example consider the filtering transforms which can be expressed as $y = Ax$, where $A$ is a $n \times n$, $k$ banded, Toeplitz matrix. Many existing designs to compute $Ax$ in time $O(n/p)$ require $\Omega(kp)$ processors. However, since $A$ is cyclic, Generate can be extended to create designs based on our architectures which require only $\Omega(p)$ multipliers to compute $Ax$ in time $O(n/p)$. Note that the number of processors required is not a function of $k$ in our design. This is due to the fact that the algorithms we use require $O(n)$ multiplications at the expense of slightly more additions/subtractions. Hence, if multipliers are much larger than adders or if $k$ is sufficiently large, the designs produced by our system will be better than existing designs.

Finally, we are planning to extend Logic so that it can produce an actual layout from the factored expression using the gate cell of figure 3.7. Moreover, since Logic will require too much time whenever the functions have a large number of input variables ($> 10$), we are exploring another approach which tries to exploit the partial symmetries of the given functions. At present, we developed optimal schemes for fully symmetric functions and functions that can be easily mapped into symmetric functions.
5. Conclusion

We have presented a brief outline of our system for generating special-purpose VLSI devices for signal processing. An automated system for specialized processors is justified only if it can produce designs that are as good as those that can be obtained by hand. We feel that our system does satisfy this condition. The architectures/algorithms are new and can be shown to offer a definite advantage over existing ones. Moreover, we have shown that our system works at a very high level and requires little effort from the user.
6. References


