A VLSI Design for Implementation of Transform Domain Adaptive Filters

ALI NAJAFI, BEHROUZ FARHANG-BOROUJENY and GANESH S. SAMUDRA*

Department of Electrical Engineering, National University of Singapore, Singapore 0511

(Received 25 June 1995; In final form 6 October 1996)

A VLSI implementation of a dedicated digital signal processor is presented. The processor is tailored for efficient implementation of transform domain adaptive filters. It incorporates a butterfly processor which performs butterfly operation to implement the required transformation. It is also able to perform complex addition, subtraction and multiplication. The butterfly processor makes use of a redundant binary tree multiplier with a recently proposed coding of signed-digit numbers which reduces the number of levels in the tree by one. An on-chip read only memory holds the transformation coefficients. The contents of the ROM determine the type of transform. The processor incorporates an ALU to perform integer arithmetic, address calculations and implementation of circular memory scheme. For fastest accessibility, the essential variables of the algorithm are implemented in a register file.

Keywords: LMS, FBLMS, FFT, VLSI, DSP, butterfly

1. INTRODUCTION

Extensive research on Least Mean Square (LMS) algorithm [1, 2] has resulted in various implementations. To speed up the convergence of the algorithm, the input samples are uncorrelated by an orthonormal transform. The resulting structure is called transform domain adaptive filter (TDAF) and the adaptive algorithm is called transform domain LMS (TDLMS). The effect of different transformations (e.g., Discrete Fourier Transform (DFT) and different Discrete Sine and Cosine Transforms (DST and DCT)) on the performance of LMS algorithm has been recently studied [3]. Implementation of the above mentioned transforms by structures similar to FFT makes the transformation to be of $O(N \log N)$ in computational complexity where $N$ is the length of the transformation and $O(\cdot)$ is order of. To perform the transformation, $N/2 \log_2 N$ butterflies (which is defined later in this paper) should be executed. In a recent paper [4], it is shown that the decimation-in-time FFT performed in sliding fashion can be implemented with an order of

* Corresponding author. e-mail: gss@vlsi.ee.nus.sg
complexity of as low as $N$. In sliding implementation of the transform, the transformation is performed after each input sample enters the transformation structure. A further generalization of the Sliding FFT, where a block of $L$ input samples are slid into an $N$-point FFT structure has been proposed in [5]. The number of butterflies to be executed in this case is

$$n_{b, \text{SFFT}} = \frac{N}{2} (\log_2 L + 2) - L. \quad (1)$$

In order to implement the sliding FFT efficiently, one should make use of a circular memory scheme [6]. Implementing transformation of the input samples in a sliding fashion with a circular memory scheme introduces a major improvement in the time requirement of TDLMS.

Another technique which is also used to improve the efficiency is to use the Frequency Domain Block LMS (FBLMS) structure, where the time domain convolution in the adaptive algorithm is replaced by frequency domain element by element multiplication [7]. This structure can improve the time requirement of the algorithm for the case $L > 64$. The FBLMS structure of [7] requires 3 DFT and 2 inverse DFT transforms. In this structure, one DFT and one inverse DFT can be removed. The resulting structure is called unconstrained FBLMS. The latter has the same speed of convergence as the former at the expense of a higher misadjustment. The misadjustment, however, is close to the one in the former for the case $L \ll N$, [5, 8]. The structure of unconstrained FBLMS is shown in Figure 1. In the rest of this paper, the term FBLMS refers to the structure of unconstrained FBLMS. In the figure, FFT transform 1 transforms the input samples. This transformation can be implemented in a sliding fashion and is shown as SFFT (sliding FFT) in the figure. For implementing the FBLMS structure, in the inverse FFT transform 2, only the last $L$ terms of the transform are required, and in the FFT transform 3, the first $M-1$ terms are zero where $M$ is the length of the adaptive filter and satisfies the relation $M = N - L + 1$. Therefore it is possible to implement these transformations with pruned FFT or pruned inverse FFT to achieve further improvement in the time requirement of the algorithm [5]. The number of butterflies for execution of pruned FFT and inverse FFT can be obtained by equations similar

\[ \text{FIGURE 1 Unconstrained frequency domain block LMS structure (FBLMS).} \]
to (1) as follows

\[ n_{b, \text{PIFFT}} = \frac{N}{2} (\log_2 L + 2) - L \]  

(2)

\[ n_{b, \text{PFFT}} = \frac{N}{2} (\log_2 (N - L) + 2) - (N - L) \]  

(3)

where \( n_{b, \text{PIFFT}} \) and \( n_{b, \text{PFFT}} \) are the number of butterflies required to be executed to obtain pruned inverse FFT and pruned FFT respectively. Based on the above discussion, by implementing the TDAF by FBLMS structure and using sliding FFT for transforming the input samples, the less time requirement of this structure reduces the order of complexity of transformation. Moreover, using the pruned transform in the remaining parts of FBLMS structure, reduces the time requirement of the adaptive algorithm.

Another issue in enhancing the performance of the transformations is to use a special purpose processor to perform certain tasks. There have been many versatile designs to implement a hardware FFT. [9] shows a 4K FFT processor, which is based on a computational element that performs a radix 4 butterfly computation which, in turn, uses radix 2 butterfly to perform the operation. Each computational element incorporates 8 complex adders and 3 complex multipliers. The circuit elements used to implement this design were ECL and TTL commercial integrated circuits. In this design, 4 FFT processors are incorporated to perform filtering in frequency domain and converting back to time domain. Another example is [10] where a radix-2 FFT is implemented. Shift registers are used in this design to change the order of the results of each state. This design was made of discrete elements [11] gives a clear explanation of radix 2,4, and 8 butterflies. Examples of VLSI implementation of FFT processors can be found in [12]. A more recent paper proposed a single FFT processor which made use of 2 multiplier-accumulators and was able to perform one butterfly in 4 clock cycles [13]. Incorporating an FFT processor in a digital signal processor enables it to perform other functions while the FFT processor is performing the butterfly operations.

In the remaining parts of this paper, implementation of the circular memory scheme which is crucial in devising an efficient sliding FFT is discussed. Next a VLSI design of butterfly processor is presented. It is capable of performing one butterfly or inverse butterfly of FFT. An architecture which makes use of the butterfly processor is also proposed. It is also able to implement the circular memory scheme and can implement the FBLMS structure efficiently.

2. SOFTWARE IMPLEMENTATION OF SLIDING FFT

Efficient implementation of sliding FFT requires that the part of the memory which contains the results of the transformation be accessed in a circular fashion where, instead of shifting the contents of a (usually) big array, only the index to the array will be modified [14, 16]. Special hardware should be therefore available in the processor which can implement the circular memory scheme efficiently.

In this section, an efficient software implementation of sliding FFT is presented. Figure 2 shows the structure of decimation-in-time FFT of length 8. Considering \( L \) to be 1, after one transformation is completed, in order to perform the transformation at the next sampling time, only the highlighted butterflies in the figure should be executed for the results of the rest are already calculated at the previous sampling time. Therefore these results should be kept in the memory. [6] has discussed the difficulties that one encounters in keeping and accessing the results in memory, has considered different approaches to the solution of the problem and has finally proposed an efficient approach which is briefly discussed here.

Let \( N \) be a power of 2 indicating the length of the transform. The transformation is performed in \( \log_2 N \) stages. In order to hold the results of each stage, \( N \) memory elements are needed. The input samples also require the same amount of memory. This adds up to \( N(\log_2 N + 1) \) elements. Therefore
an array called $x$ is allocated with a length equal to the smallest integer power of 2 that is greater than or equal to this number. This is given by

$$L_x = 2^\lceil \log_2(N + 1) \rceil$$

where $L_x$ denotes the length of $x$ and $\lceil i \rceil$ denotes the smallest integer which is greater then or equal to $i$.

In order to access $x$, an index called $I_x$, is used. It is first initialized to 0 and after each transformation is accomplished, it is incremented according to

$$I_x = (I_x + 1) \text{ AND } (L_x - 1).$$

In fact, a circular memory is implemented by this relation. Figure 3 shows the circular memory for the case $N = 4$. By using (4) the length of the array is found to be $L_x = 16$. However the number of stages required to perform the transformation is 2. In this way, $x$ has 3 parts. Therefore only 12 elements are involved in each transformation. At first $I_x$ is set to zero pointing to the physical start of $x$. The first input sample, $x(0)$, is put into the $(I_x + N - 1) \text{ AND } (L_x - 1)$'th element. All other elements are set to zero and then the transformation is performed (Fig. 3a). To perform the next transformation, $I_x$ is incremented according to the assignment statement (5). The next input sample, $x(1)$, is then put into the $(I_x + N - 1) \text{ AND } (L_x - 1)$'th element of $x$. This is the element next to the element containing $x(0)$. In this way, the first part of the array is rotated by one position. This is equivalent to shifting the contents of this part all by one position as well as shifting all other parts (Fig. 3b). The same procedure is followed for the next three input samples (Fig. 3c). As the next transformation is going to be performed, the last part of the array which contains the transform of the input samples is broken to two sections. This, however, is transparent to the algorithm for it uses the AND operator to index into the array (Fig. 3d).

The circular memory scheme discussed above is implemented on conventional computers as well as digital signal processors, and the time requirement of the algorithm is measured. This is reported in [6]. Based on the results discussed in [6], the circular memory scheme enables one to implement the sliding FFT with $O(N)$ complexity, provided

![FIGURE 2 Decimation-in-time FFT of length 8.](image-url)
that there is enough hardware in the processor which can implement the scheme without any overhead instructions.

3. ARCHITECTURE OF THE MAIN PROCESSOR

Figure 4 shows the block diagram of the processor. It can implement the FBLMS structure. The transformation of input samples is performed by sliding FFT. The processor incorporates a butterfly processor as its major block. The butterfly processor incorporates a recently developed binary-tree multiplier. The processor incorporates an ALU to perform integer arithmetic and address calculations and also implements the circular memory scheme efficiently. The processor implements the essential variables of the FBLMS structure in registers of a register file for fastest accessibility. Most of the registers, however, can be used for other purposes as well. The processor incorporates two output data buses and one input data bus to the register file. Therefore, it is possible to implement instructions with three operands. An on-chip ROM called program memory contains the adaptive algorithm program. Another on-chip ROM called the transformation coefficients memory contains the coefficients of FFT for the case the length of transform is 512. However, the same coefficients can be used to perform FFT of smaller lengths. This can be obtained by simply skipping the unnecessary coefficients. By changing the coefficients appropriately, any other transformation which follows the structure of decimation in time FFT can be implemented. Addressing and address calculation of the ROM is performed independent of the ALU to enable the processor to perform another instruction while the ROM is being accessed.
4. BUTTERFLY PROCESSOR

The butterfly operation of FFT is defined as
\[ y_1 = x_1 + W \cdot x_2 \]  
(6)
\[ y_2 = x_1 - W \cdot x_2 \]  
(7)
and the butterfly operation of inverse FFT is defined as
\[ y_1 = \frac{1}{2} (x_1 + x_2) \]  
(8)
\[ y_2 = \frac{1}{2} W(x_1 + x_2) \]  
(9)
where \( x_1 \) and \( x_2 \) are the inputs to the butterfly, \( W \) is a transformation coefficient and \( y_1 \) and \( y_2 \) are the outputs. (6) and (7) can be written as follows:
\[ y_1 \cdot r = x_1 \cdot r + W \cdot r \cdot x_2 \cdot r - W \cdot i \cdot x_2 \cdot i \]  
(10)
\[ y_1 \cdot i = x_1 \cdot i + W \cdot r \cdot x_2 \cdot i + W \cdot i \cdot x_2 \cdot r \]  
(11)
\[ y_2 \cdot r = x_1 \cdot r - W \cdot r \cdot x_2 \cdot r + W \cdot i \cdot x_2 \cdot i \]  
(12)
\[ y_2 \cdot i = x_1 \cdot i - W \cdot r \cdot x_2 \cdot i - W \cdot i \cdot x_2 \cdot r \]  
(13)
Where \( \cdot r \) and \( \cdot i \) refer to real and imaginary parts respectively. Similarly, (8) and (9) can be written as follows:

\[
y_1 \cdot r = \frac{1}{2} (x_1 \cdot r + x_2 \cdot r)
\]

\[
y_1 \cdot i = \frac{1}{2} (x_1 \cdot i + x_2 \cdot i)
\]

\[
y_2 \cdot r = \frac{1}{2} (W \cdot r(x_1 \cdot r - x_2 \cdot r) - W \cdot i(x_1 \cdot i - x_2 \cdot i))
\]

\[
y_2 \cdot i = \frac{1}{2} (W \cdot r(x_1 \cdot i - x_2 \cdot i) + W \cdot i(x_1 \cdot r - x_2 \cdot r))
\]

In this section, an efficient butterfly processor is presented. This processor makes use of one multiplier only and is able to implement one butterfly of FFT in 5 cycles. For the butterfly of inverse FFT, however, it can accomplish the calculations in 4 clock cycles only. Compared to the butterfly processor of [13], this processor incorporates one multiplier instead of two. This reduces the area requirement of the processor substantially. This processor takes the same number of cycles for calculating inverse butterfly as the one in [13]. For calculating butterfly, it takes one more cycle. However, as it will become clear later, similar to the concept of pipelining, the first and last cycles can be performed simultaneously. Therefore we can claim that the butterfly operation is performed in 4 cycles. At each cycle, controller of the butterfly processor provides its different parts with the required control signals. Figure 5 Shows the architecture of the butterfly processor. In the figure, the signals highlighted with \(\text{*}\) are the ones provided by the main processor. Others are generated by the controller of the butterfly processor (not shown in the figure).

In addition to the butterfly operation, the butterfly processor is able to perform complex and real multiplications, as well as addition and subtraction. Complex multiplication can be performed by substituting \(x_1\) in (6) and (7) with zero. Complex addition and subtraction can be performed by substituting \(W\) in (6) and (7) with one. After presenting the basic operations, a detailed explanation of the architecture of the butterfly processor now follows.

The communication between the butterfly processor and the main processor is accomplished through input and output buffers. The input buffers contain the 2 inputs to the butterfly and the twiddle factor (\textit{i.e.,} the transformation coefficient). They are called \(x_1\text{InBuf}, x_2\text{InBuf}, \text{and } W\text{InBuf}.\) The output buffers contain the 2 outputs of the butterfly. There are called \(y_1\text{OutBuf} \) and \(y_2\text{OutBuf}\) and are shown in the top row of the figure. These buffers are internally latched. Therefore, while the butterfly processor is busy performing current operation, it is possible to read the results of the last operation from the output buffer and write the inputs and the twiddle factor of the next one into the input buffers in the same clock cycle. The width of the input and output buffers are 24 bits for real part and 24 bits for imaginary part. The input buffers of the twiddle factors are 16 bits for real part and 16 bits for imaginary part. Each read or write to any of the buffers will access both the real and imaginary parts.

The operation of signals mentioned in Figure 5 is described here.

- Signals whose name start with “Load”, load the latch to which they are connected with their input. For instance, “Loadrl” loads r1 with its input. The load operation is performed at the end of the cycle that the signal is active. Exceptions are signals Loadx1InBuf, Loadx2InBuf, LoadWN and LoadMulOutWin which load the input at the beginning of the cycle. These signals are highlighted by “\*” on Figure 5.
- Signals whose name end in “iSel” are connected to multiplexers which select either real or imaginary part of the latch to which their inputs are connected. Each of these signals, when active, selects the imaginary part of the latch.
FIGURE 5 Architecture of butterfly processor.
For instance, signal xInSel selects the imaginary part of x1.

- Signals whose name end in “InSel” are connected to multiplexers. The inputs of the multiplexers, on the other hand, are named “0” and “1”, respectively (Fig. 5). Each of these signals, when active, selects the input named “1”. For instance, r1InSel selects the input “1” of multiplexer r1IMux, which is the output of Adder3.

- Signals whose name start with “Out” are connected to tri-state buffers and activate the buffers. For instance, signal OutShifter3 activates the tri-state buffers connected to the output of Shifter3.

- When signal SubMBar is active, the output of Adder4 is the subtraction of its inputs. Otherwise, its output is the addition of its inputs.

- When signal Shifter3Div2 is active, the output of Shifter3 is its input times 0.5. Otherwise, its output is equal to its input.

- Signal MulOutWinValue contains the value to be loaded in MulOutWinReg.

4.1. Sequence of Executions in the Butterfly Processor

In this section the sequence of the operations performed for calculating the butterfly and inverse butterfly are explained.

For the butterfly operation, the butterfly processor performs the steps shown in Table I. In this table, identifiers are the names of the latches in the butterfly processor shown in Figure 5. Furthermore, suffix “.r” refers to the real part and suffix “.i” refers to the imaginary part of the latches. All the assignments shown in Table I are performed at the end of the cycles. The exceptions are the assignments highlighted by “*”. The reason is that the value of the latches W, x1 and x2 are required during Cycle (1) and should be loaded at the start of the cycle. Table I also shows which of the signals in the architecture of Figure 5 should be activated to perform each operation. The steps shown in Table I are in accordance with (10), (11), (12) and (13). By tracing the value of the variables mentioned in this table, y1.r, y1.i, y2.r and y2.i are found to contain values of the above equations at the end of Cycle (5).

An interesting feature of the butterfly processor is that while it is performing Cycle (5) of one butterfly operation, it can also perform Cycle (1) of another one. This is similar to the concept of pipelining. The reason is, as can be seen in Table I, except for x1Buf, no other latches are being referred to in Cycles (1) and (5) both. Furthermore, in Cycle (1), the value of x1Buf is being changed and in Cycle (5), its value is being read.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Operation</th>
<th>Signals Activated</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>W = WInBuf*</td>
<td>Load WN</td>
</tr>
<tr>
<td></td>
<td>x1 = x1InBuf*</td>
<td>Load x</td>
</tr>
<tr>
<td></td>
<td>x2 = x2InBuf*</td>
<td>Load x</td>
</tr>
<tr>
<td></td>
<td>r1 = x1.r + W.r*x2.r</td>
<td>Loadr1, r1InSel</td>
</tr>
<tr>
<td></td>
<td>x1Buf = x1.r</td>
<td>Loadx1Buf</td>
</tr>
<tr>
<td></td>
<td>y1.r = r2 = r1 - W.i*x2.i</td>
<td>Loady1r, Loadr2, OutShifter3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SubMBar, WNISel, x2ISel</td>
</tr>
<tr>
<td>(2)</td>
<td>y2.r = -r2 + 2*x1Buf</td>
<td>Loady2r, OutAdder5</td>
</tr>
<tr>
<td></td>
<td>r1 = x1.i + W.r*x2.i</td>
<td>Loadr1, r1InSel, xlISel</td>
</tr>
<tr>
<td></td>
<td>x1Buf = x1.i</td>
<td>Loadx1Buf, xlISel</td>
</tr>
<tr>
<td>(3)</td>
<td>y1.i = r2 = r1 + W.i*x2.r</td>
<td>Loady1i, Load2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OutShifter3, WNISel</td>
</tr>
<tr>
<td>(4)</td>
<td>y2.i = -r2 + 2*x1Buf</td>
<td>OutAdder5, LoadyOutBuf</td>
</tr>
<tr>
<td></td>
<td>y1OutBuf = y1</td>
<td>LoadyOutBuf</td>
</tr>
<tr>
<td></td>
<td>y2OutBuf = y2</td>
<td>LoadyOutBuf</td>
</tr>
</tbody>
</table>

* These operations are performed at the start of the cycle.
However, the change occurs at the end of Cycle (1) only and by that time, the value of \( x_1 \text{Buf} \) is no longer needed in Cycle (5).

The sequence of executions for performing an inverse butterfly operation is shown in Table II. All the assignments shown in the table are performed at the end of the cycles except the ones highlighted by "***", for the value of the latches \( W' \), \( x_1 \) and \( x_2 \) are required during Cycle (1) and should be loaded at the start of the cycle. Table II also shows which of the signals in the architecture of Figure 5 should be activated to perform each operation. The steps shown in Table II are in accordance with (14), (15), (16) and (17). By tracing the value of the variables mentioned in this table, \( y_1 \cdot r, \ y_1 \cdot i, \ y_2 \cdot r \) and \( y_2 \cdot i \) are found to contain the values of the above equations at the end of Cycle (4).

As can be seen in the table, the butterfly processor performs the inverse butterfly operation in 4 cycles.

4.2. Multiplier

The butterfly processor incorporates an integer 24-bit \( \times 16 \)-bit signed digit binary-tree multiplier. The output of the multiplier is 40 bits. The coding of the signed-digit is the one discussed in [15] where designing the Booth encoders based on that coding reduces the number of levels in the tree by one. The binary tree has 3 levels only. In other words, there are only 4 partial products in the multiplier. The 16 bit input is connected to the Booth encoders.

4.3. Multiplier Output Window

The calculations in the main processor are in fixed point representation and the position of the decimal point changes in different stages of the adaptive algorithm which causes the precision of the output of the multiplier to change. On the other hand, the butterfly processor is able to make use of only 24 bits of the multiplication. This requires that there be a mechanism to select the appropriate 24 bits out of the result. For this purpose, a window called (MulOutWin) is provided at the output of the multiplier. The contents of a register called multiplier output window register determine which 24 bits are selected. This register is loaded by the main processor.

The multiplier output window is a multiplexer. The principle behind it is the same as that of the design of the barrel shifter made from pass transistors [16]. The window, however, is made based on barrel cell which uses a transmission gate. Each barrel cell consists of two transmission gates and has two data inputs, namely Data0 and Data1, and one data output, namely DataOut. It also has four command lines. The value of the

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Operation</th>
<th>Signals Activated</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>( W = W'\text{InBuf} ) *</td>
<td>Load WN</td>
</tr>
<tr>
<td></td>
<td>( x_1 = x_2\text{InBuf} ) *</td>
<td>Loadx</td>
</tr>
<tr>
<td></td>
<td>( x_2 = x_3\text{InBuf} ) *</td>
<td>Loadx</td>
</tr>
</tbody>
</table>
|       | \( r_1 = (x_1 \cdot r - x_2 \cdot r) \cdot W' \cdot r 
|       | \( y_1 \cdot r = (x_1 \cdot r + x_2 \cdot r) \cdot 1/2 
| (2)   | \( y_2 \cdot r = (r_1 - (x_1 \cdot i - x_2 \cdot i) \cdot W' \cdot i) \cdot 1/2 
| (3)   | \( y_1 \cdot i = (x_1 \cdot i + x_2 \cdot i) \cdot 1/2 
|       | \( r_1 = (x_1 \cdot i - x_2 \cdot i) \cdot W' \cdot r 
| (4)   | \( y_2 \cdot i = (r_1 + (x_1 \cdot r - x_2 \cdot r) \cdot W' \cdot i) \cdot 1/2 
|       | \( y_\text{OutBuf} = y_1 
|       | \( y_2\text{OutBuf} = y_2 

* These operations are performed at the start of the cycle.
command lines determines which of the data input lines should be connected to the output. The transistor level schematic of the barrel cell is shown in Figure 6. A signal called Shift and its complement are connected to the control lines. Once the signal Shift is active, Datal is connected to the line DataOut. Otherwise, Data0 is connected to that line.

In the barrel multiplexer, the barrel cells are arranged as shown in Figure 7. Each box in the figure represents one barrel cell. Each row of barrel cells in the figure is controlled by one Shift signal. The first row from the top, controlled by signal Shift1.1 selects 39 of 40 bits of the output of the multiplier. It behaves similar to a sliding window which can shift one position and show 39 bits. When Shift1,1 is set to 1, the first row outputs the least significant 39 bits of its input, and when set to 0, it outputs the most significant 39 bits. The second row, controlled by signal Shift1.2, selects 38 bits of the first row. When Shift1,2 is set to 1, the second row outputs the least significant 38 bits of its input, and when set to 0, it outputs the most significant 38 bits. The third row, controlled by signal Shift2 selects 36 bits of 38 bits of the second row. When Shift2 is set to 1, the third row outputs the least significant 36 bits of its input, and when set to 0, it outputs the most significant 36 bits. The forth row, controlled by signal Shift4 selects 32 bits of the bits of the third row. When Shift4 is set to 1, the forth row outputs the least significant 32 bits of its input, and when set to 0, it outputs the most significant 32 bits. The last row, controlled by signal Shifts selects 24 bits of the bits of the forth row. It is similar to a sliding window which can shift 8 positions. When Shifts is set to 1, the last row outputs the least significant 24 bits of its input, and when set to 0, it outputs the most significant 24 bits. Each Shift, signal, when active, makes the corresponding row to select the least significant bits of its input. The index of the least
significant bit of the output of the Multiplier Output Window is:

\[ \text{Shift}_8 \times 8 + \text{Shift}_4 \times 4 + \text{Shift}_2 \times 2 + \text{Shift}_{1.2} \times 1 + \text{Shift}_{1.1} \times 1 \]  \tag{18} \]

In other words, when all signals are set, the output of the window is the least significant bits of the output of the multiplier, and when they are all reset, the output of the window is the most significant bits of the output of the multiplier. For instance, if the signals of (18) are set to 11001, respectively, the least significant bit of the output of the Multiplier Output Window is bit 3 of the output of the multiplier. As another example, if these signals are set to 00000, bit 16 of the output of the Multiplier is assigned to the least significant bit of the Multiplier Output Window. The two rows of inverters are there to provide sufficient current drive between the rows of barrel cells.

5. IMPLEMENTATION OF THE PROCESSOR

The main processor is implemented in VLSI by software MAGIC, using the double metal CMOS \( \lambda \)-based technology and 1.2\( \mu \)m process (\( \lambda = 0.6 \mu \)m). First the layout of the required logic cells were designed. They were then manually incorporated into larger blocks to implement their layout, until the processor was fully implemented. A ROM generator and the layout of the required memory, address decoding and buffer cells were also developed to automatically implement the layout of ROMs. The size of the chip is 22184 \( \times \) 18371 \( \lambda \) which is equivalent to 13.31 mm \( \times \) 11.02 mm. The size of butterfly processor is 8656 \( \times \) 8938\( \lambda \) which is equivalent to 5.2 mm \( \times \) 5.4 mm. In order to test the functionality of the individual blocks, the schematic netlist and capacitances are extracted from the layout. The extraction rules are specified to account for loading in CMOS technology. The simulator IRSIM which is integrated into software MAGIC is then used to simulate the extracted netlist by applying input test vectors and analyzing the outputs. Due to limitations in facilities and budget at the time of development of the project, the chip could not be fabricated.

![Figure 8 Clock pulse](image)

<table>
<thead>
<tr>
<th>( N )</th>
<th>( L )</th>
<th>Number of Instructions</th>
<th>FBLMS Rate (Hz)</th>
<th>Sampling Rate (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1</td>
<td>1857</td>
<td>769.290</td>
<td>769.290</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>1929</td>
<td>740.576</td>
<td>1481.152</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>2010</td>
<td>710.732</td>
<td>2842.928</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>2106</td>
<td>678.334</td>
<td>5426.672</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>2235</td>
<td>639.182</td>
<td>10226.909</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>2425</td>
<td>589.102</td>
<td>18851.251</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>3496</td>
<td>408.630</td>
<td>408.630</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>3632</td>
<td>393.329</td>
<td>786.658</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>3777</td>
<td>378.229</td>
<td>1512.916</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>3938</td>
<td>362.766</td>
<td>2902.126</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>4131</td>
<td>345.817</td>
<td>5533.077</td>
</tr>
<tr>
<td>128</td>
<td>32</td>
<td>4389</td>
<td>325.489</td>
<td>10415.649</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>4768</td>
<td>299.616</td>
<td>19175.455</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>6799</td>
<td>210.115</td>
<td>210.115</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
<td>7063</td>
<td>202.261</td>
<td>404.523</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>7336</td>
<td>194.734</td>
<td>778.938</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>7625</td>
<td>187.354</td>
<td>1498.829</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>7947</td>
<td>179.762</td>
<td>2876.198</td>
</tr>
<tr>
<td>256</td>
<td>32</td>
<td>8334</td>
<td>171.415</td>
<td>5485.275</td>
</tr>
<tr>
<td>256</td>
<td>64</td>
<td>8849</td>
<td>161.439</td>
<td>10332.079</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>9067</td>
<td>148.701</td>
<td>19033.740</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>13494</td>
<td>105.867</td>
<td>105.867</td>
</tr>
<tr>
<td>512</td>
<td>2</td>
<td>14014</td>
<td>101.939</td>
<td>203.878</td>
</tr>
<tr>
<td>512</td>
<td>4</td>
<td>14543</td>
<td>98.231</td>
<td>392.923</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
<td>15088</td>
<td>94.683</td>
<td>757.461</td>
</tr>
<tr>
<td>512</td>
<td>16</td>
<td>15666</td>
<td>91.189</td>
<td>1459.029</td>
</tr>
<tr>
<td>512</td>
<td>32</td>
<td>16310</td>
<td>87.589</td>
<td>2802.838</td>
</tr>
<tr>
<td>512</td>
<td>64</td>
<td>17064</td>
<td>83.620</td>
<td>5351.707</td>
</tr>
<tr>
<td>512</td>
<td>128</td>
<td>18115</td>
<td>78.861</td>
<td>10094.239</td>
</tr>
<tr>
<td>512</td>
<td>256</td>
<td>19630</td>
<td>72.775</td>
<td>18630.376</td>
</tr>
</tbody>
</table>
The timing of the blocks of the chip has been simulated by SPICE circuit simulator. The slowest blocks of the chip are the ROM's. Based on the simulations performed by SPICE circuit simulator, the ROM's provide data in 80 nano-seconds. Another slow device in the processor is the multiplier which is located in-between the path of input buffers of butterfly processor, multiplier output window and the adders Adder3 and Adder4 (Fig. 5). Based on simulations performed by SPICE circuit simulator on sub-blocks of this path, the approximate delay in this path is 70 nano-seconds. Based on the simulations made on long routes in the main processor, the delay is at most 20 nano-seconds. This includes the routing of control signals generated by controller.

A two-phase clock pulse is chosen to avoid the race hazard. Figure 8 shows the clock pulse. Phase 1 is used to load the state flip flops of the controller and stage flip flops of the butterfly processor, as well as loading some of the registers which should be loaded at the start of the cycle (Tabs. I and II). Phase 2 is used to load data in other registers. The limiting circuit in determining the clock frequency is the ROM. Therefore, at least 80 nano-seconds are required. The width of the clock pulses are set to the maximum delay it takes for the control
signals to reach their destinations, i.e., 20 nano-seconds. The margin between the two phases are also set to the same amount so that, because of the delay, the two phases won’t be active at any location in the chip at the same time. The period of the clock pulse is therefore 140 nano-seconds or equivalently, the clock frequency is selected to be 7 MHz. Using this as the highest clock frequency, the time requirement of an unconstrained FBLMS algorithm developed for this processor is shown in Table III. Figure 9 shows the floor plan of the butterfly and Figure 10 shows its layout. Figure 11 shows the floor plan of the main processor and Figure 12 shows its layout.

6. CONCLUSION

A special purpose processor was proposed to efficiently implement the FBLMS algorithm. The butterfly processor performs one butterfly operation in 4 clock cycles and has only one multiplier compared to a previously known structure which requires the same number of cycles but two multipliers. The processor incorporated a dedicated butterfly processor. This, in turn, makes use of a modern multiplier. Other essential blocks of the processor include ALU and controller. The architecture of the processor as well as the VLSI implementation were discussed. The timing performance of the processor was also presented.

References

Authors’ Biographies

AU Najafi has received his Bachelor’s degree in computer engineering with first rank honors from Isfahan University of Technology, Isfahan, Iran, in 1989. He has completed his Master of Engineering degree in the Department of Electrical Engineering of National University of Singapore. He is currently an ASIC development engineer in Aztech Systems Ltd., Singapore. His research interests include VLSI design and relational database systems. He has been a member of IEEE since 1984.

B. Farhang-Boroujeny received the B.S. degree in electrical engineering from Teheran University, Iran, in 1976, the M.Eng. degree in system test technology from University of Wales Institute of Science and Technology, UK, in 1977, and the Ph.D. degree from Imperial College, University of London, UK, in 1981. From 1981 to 1989 he was with Isfahan University of Technology, Isfahan, Iran. Since 1989 he has been with National University of Singapore. His current scientific interests are adaptive signal processing, digital filter design, and VLSI implementation of signal processing systems. He received the UNESCO Regional Office of Science and Technology for South and Central Asia young scientists award, in 1987, in recognition of his outstanding contribution in the field of computer applications and informatics.

Ganesh Samudra was born in India. He received the M.Sc. from Indian Institute of Technology, Bombay in 1976 and M.S., M.S.E.E., and Ph.D. degrees from Purdue University in 1985. In 1985, he joined Texas Instruments, Inc., Dallas, where he worked on development and support of semiconductor process and device simulation tools. In 1987, he joined Texas Instruments, Bangalore, India where he was section manager for simulation. He was elected Member of Group Technical Staff in 1988 for outstanding technical contributions in simulation. Since 1989, he is with the department of Electrical Engineering, National University of Singapore where he is presently a senior lecturer. He is presently working in application and development of Technology and Design CAD tools.
Submit your manuscripts at
http://www.hindawi.com